

Vector Semantics and Embeddings

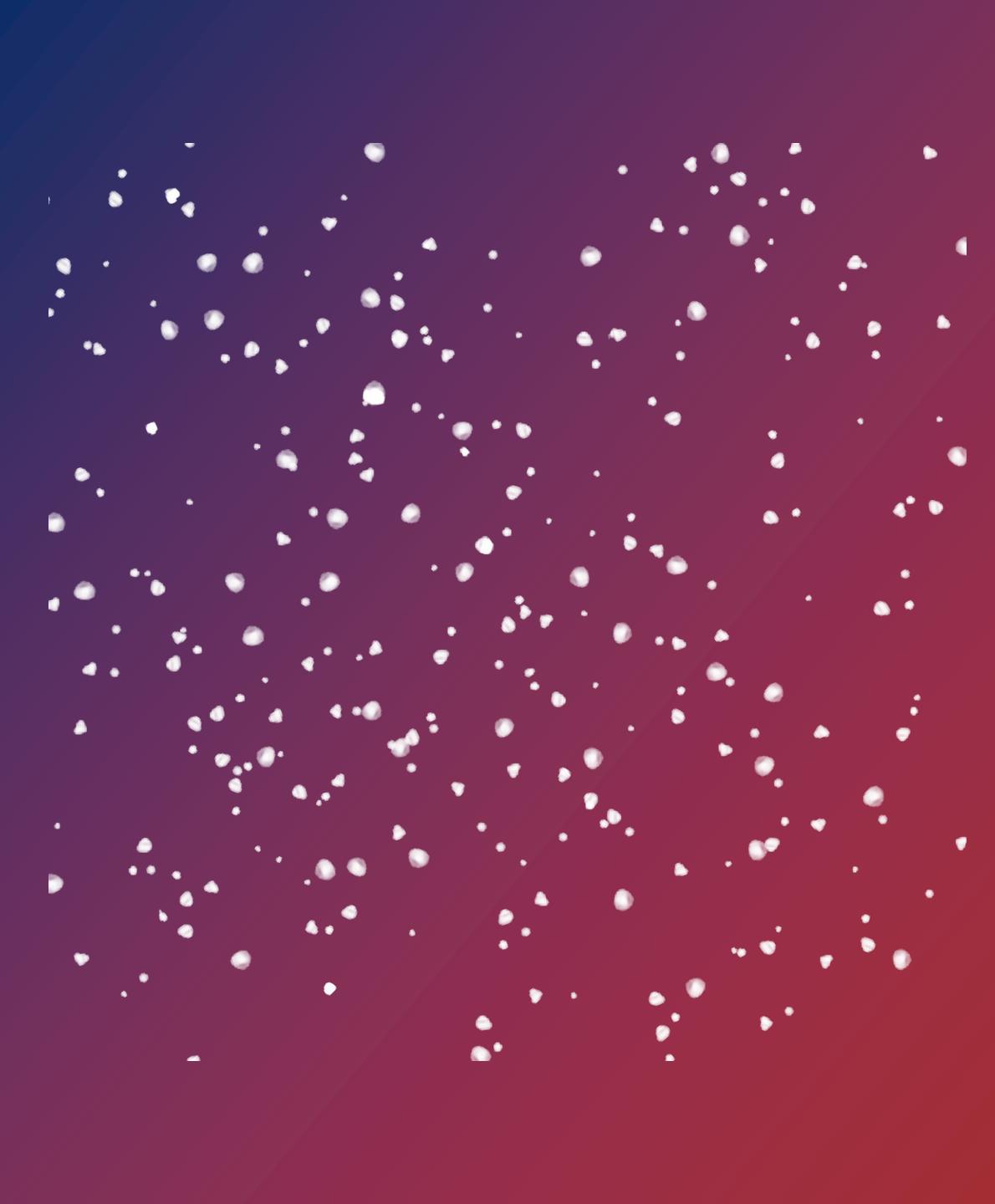
Natalie Parde
UIC CS 421

What we know so far....

- **Word vectors:** Vectors of numbers used to encode language
- Simple techniques to create word vectors:
 - Co-occurrence frequency (**bag of words**)
 - **TF-IDF**

1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1

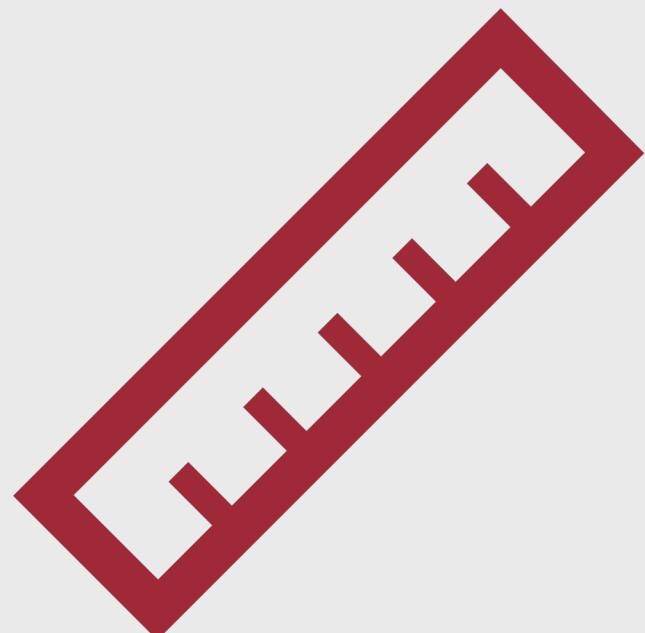
0.7 | 0 | 0 | 0 | 0 | 0.9 | 0.1 | 0 | 0 | 0.5



Word vectors indicate a word's meaning with respect to other words!

- Each vector represents a point in the n -dimensional semantic space
 - $|V|$ -dimensional word embedding $\rightarrow |V|$ -dimensional semantic space

Now that we know how to create a vector space model, how can we use it to compute similarity between words?



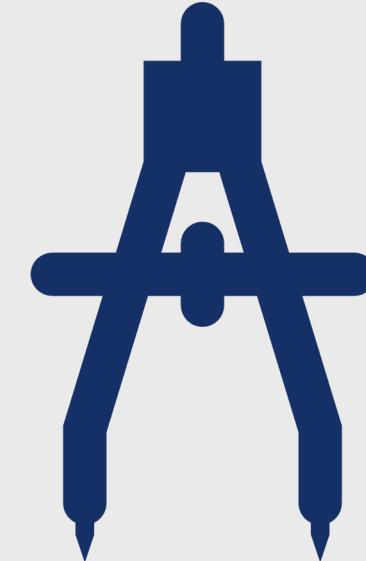
- **Cosine similarity**
 - Based on the **dot product** (also called **inner product**) from linear algebra
 - $\text{dot product}(\mathbf{v}, \mathbf{w}) = \mathbf{v} \cdot \mathbf{w} = \sum_{i=1}^N v_i w_i = v_1 w_1 + v_2 w_2 + \dots + v_N w_N$
 - Similar vectors (those with large values in the same dimensions) will have high values; dissimilar vectors (those with zeros or low values in different dimensions) will have low values

Why don't we just use the dot product?

- More frequent words tend to co-occur with more words and have higher co-occurrence values with each of them
- Thus, the **raw dot product will be higher for frequent words**
- This isn't good! 😞
 - We want our similarity metric to tell us how similar two words are regardless of frequency
- The simplest way to fix this problem is to **normalize for the vector length** (divide the dot product by the lengths of the two vectors)



Normalized Dot Product = Cosine of the angle between two vectors



- 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

Example: Computing Cosine Similarity

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

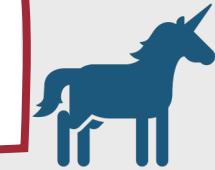
$$\cos(\text{unicorn}, \text{information}) = ?$$



Example: Computing Cosine Similarity

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

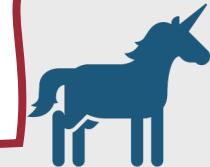
$$\cos(\text{unicorn}, \text{information}) = \frac{[442, 8, 2] \cdot [5, 3982, 3325]}{\sqrt{442^2 + 8^2 + 2^2} \sqrt{5^2 + 3982^2 + 3325^2}}$$



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*!

Limitations of Classic Word Representation Strategies

- No capacity to infer deeper semantic content
- Can't encode the following using a bag-of-words vector:
 - Synonyms
 - Antonyms
 - Positive/negative connotations
 - Related contexts

**Additionally,
remember that
bag of words
representations
are sparse.**

- Very **high-dimensional**
- Lots of **empty** (zero-valued) cells

- **Lower-dimensional** (~ 50-1000 cells)
- Most cells with **non-zero** values
- We'd also prefer to be able to encode other dimensions of meaning than word type alone
 - *Good* should be:
 - Far from *bad*
 - Close to *great*

We'd
prefer to
have dense
vectors.



It turns out that dense vectors are preferable for NLP tasks for many reasons!

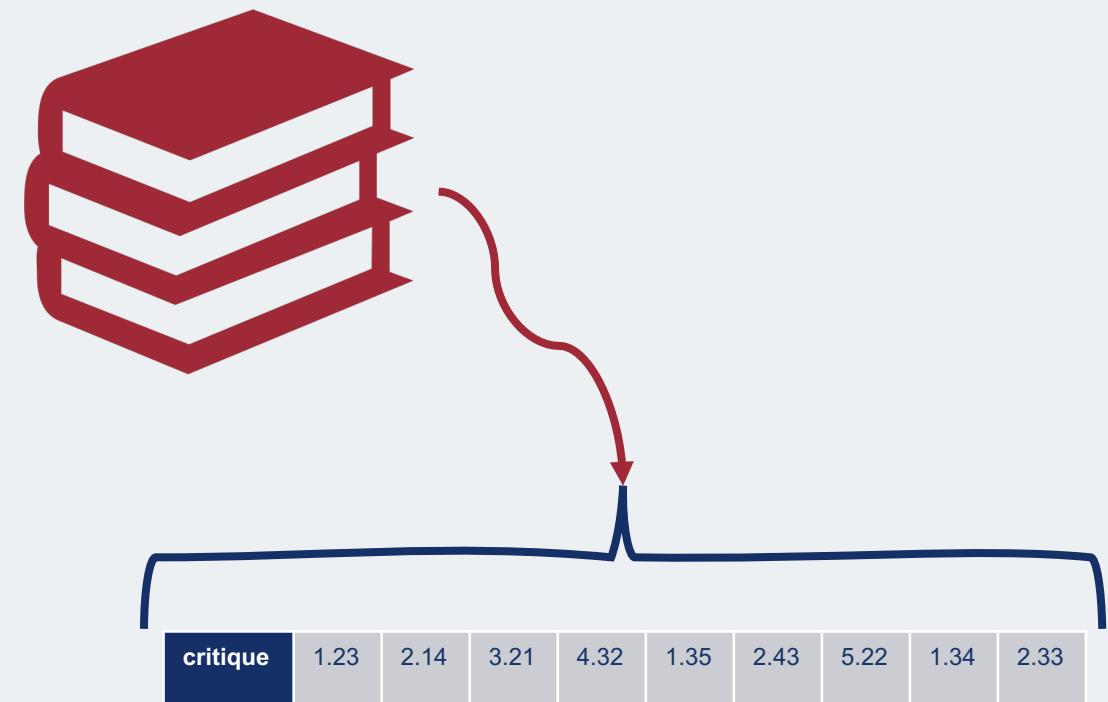
- Easier to include as **features** in machine learning systems
 - Classifiers have to learn ~100 weights instead of ~50,000
- Fewer **parameters** → lower chance of overfitting
 - May generalize better to new data
- Better at capturing **synonymy**
 - Words are not distinct dimensions; instead, **dimensions correspond to meaning components**

What is the best way to generate dense word vectors?

- The answer changes quite frequently:
 - <https://super.gluebenchmark.com/leaderboard/>
- Current state-of-the-art models are **bidirectional** (trained to represent words using both their left and right context), **contextual** (produce different vectors for different word senses) models built using **Transformers** (a type of neural network)

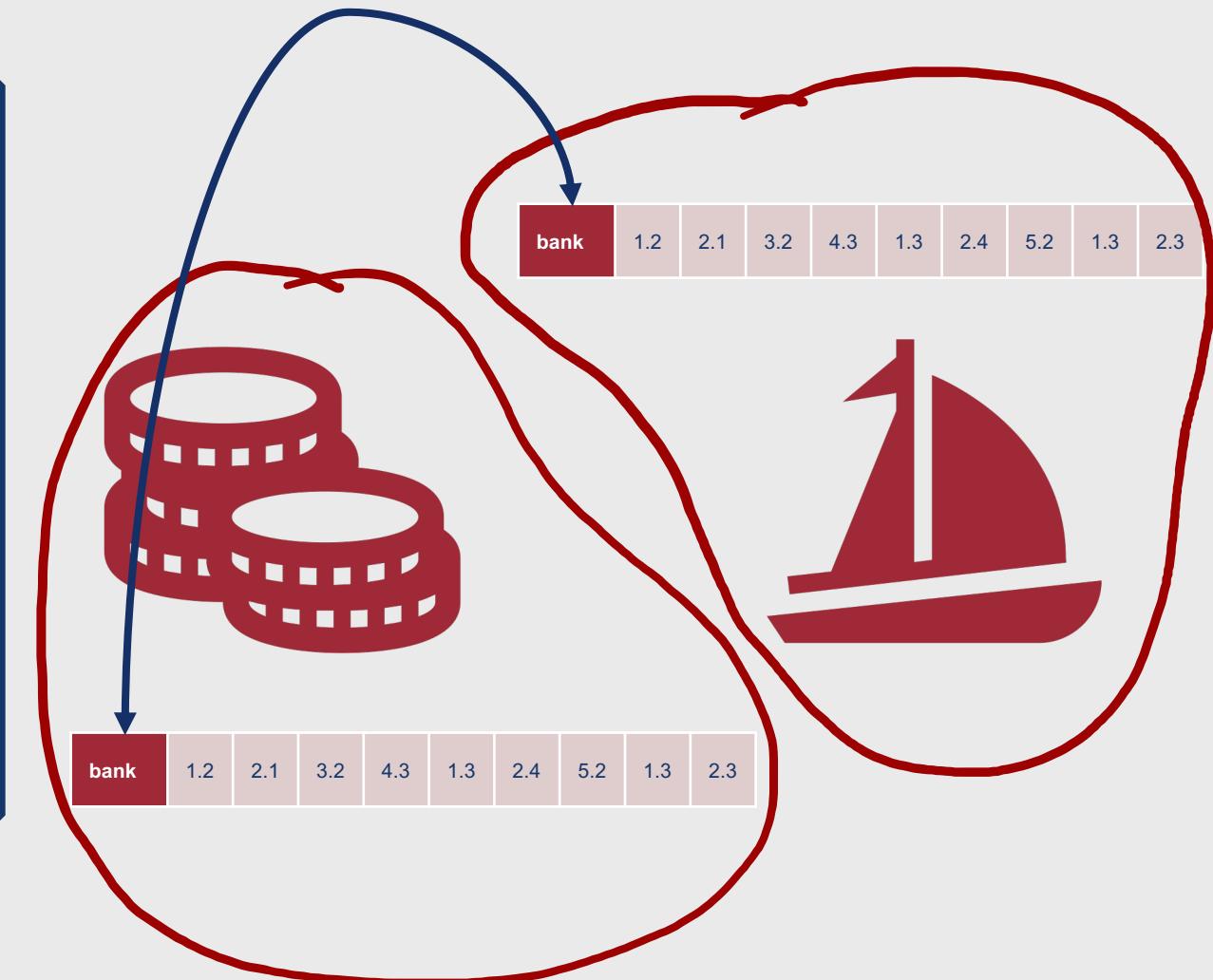
Somewhere between TF-IDF and contextual word vectors....

- **Word2Vec:** A method for automatically learning dense word representations from large text corpora



Characteristics of Word2Vec

- Non-contextual
- Fast
- Efficient to train





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**

How does Word2Vec work?

- 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”
 - This idea comes from **neural language modeling** (neural networks that predict the next word based on prior words)
 - However, Word2Vec is simpler than a neural language model:
 - It has fewer layers
 - It makes **binary yes/no predictions** rather than predicting words

this sunday, watch the super bowl at 5:30 p.m.

c1

c2

t

c3

c4

What does the classification task look like?

- Assume the following:
 - **Text fragment:** this sunday, watch the super bowl at 5:30 p.m.
 - **Target word:** super
 - **Context window:** ± 2 words

this sunday, watch the super bowl at 5:30 p.m.

c1

c2

t

c3

c4

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
 - $P(+ | t, c)$



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

- We base this decision on the similarity between the input vectors for t and c
- More similar vectors → more likely that c occurs near t

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

super bowl

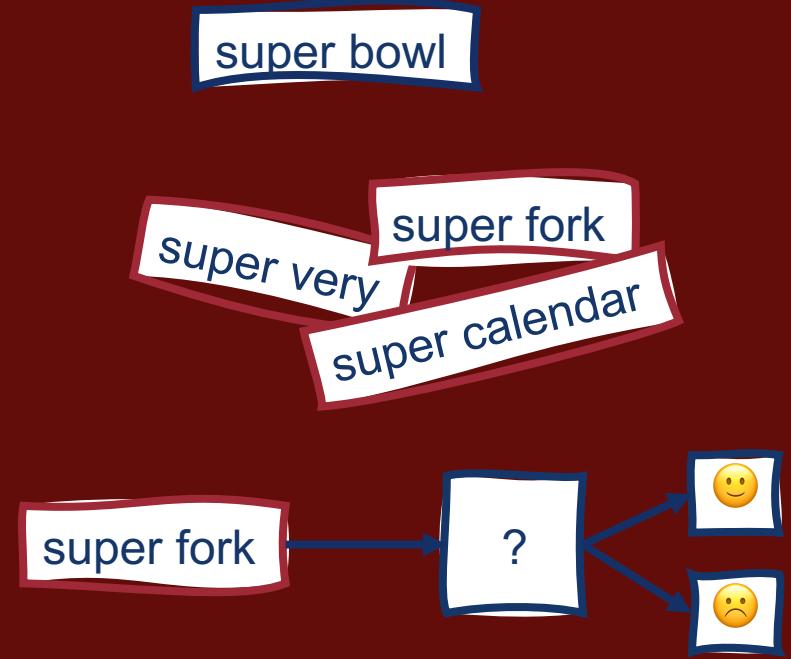
super very

super fork

super calendar

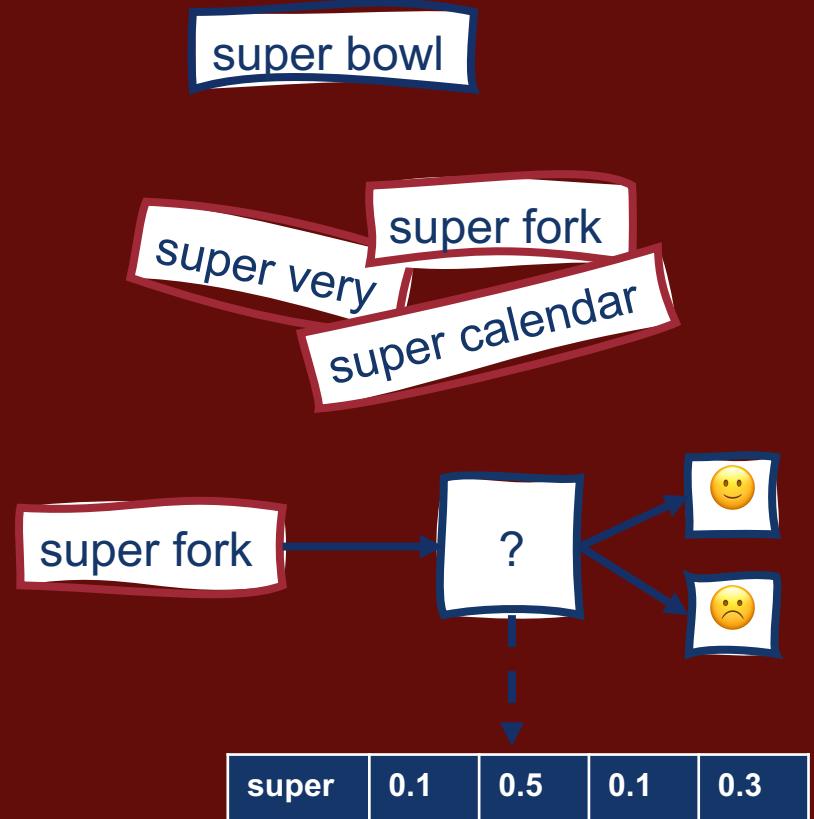
High-Level Overview: How Word2Vec Works

- 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



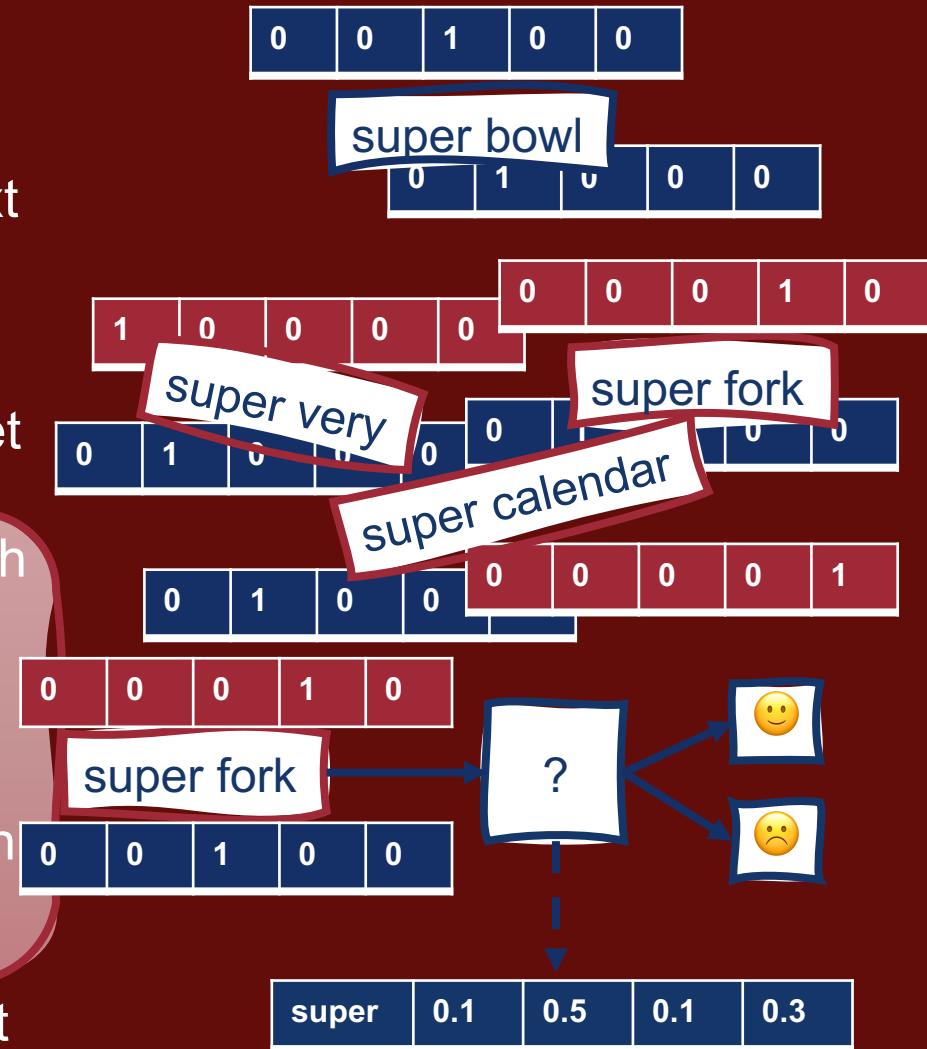
High-Level Overview: How Word2Vec Works

- 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* $P(+ \mid t, c)$?

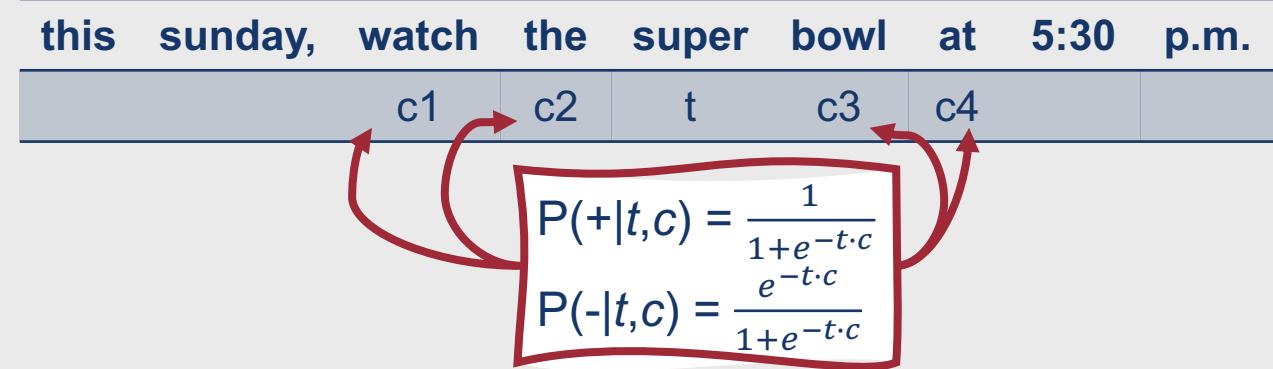
- This is based on vector similarity
- We can assume that vector similarity is proportional to the dot product between two vectors
 - $\text{Similarity}(t, c) \propto t \cdot c$

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:
 - $P(+ | t,c) = \frac{1}{1+e^{-t\cdot c}}$
 - $P(- | t,c) = 1 - P(+ | t,c) = \frac{e^{-t\cdot c}}{1+e^{-t\cdot c}}$



We're usually not just looking at words in isolation.



- What if we're considering the probability of a span of text occurring in the context of a 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....

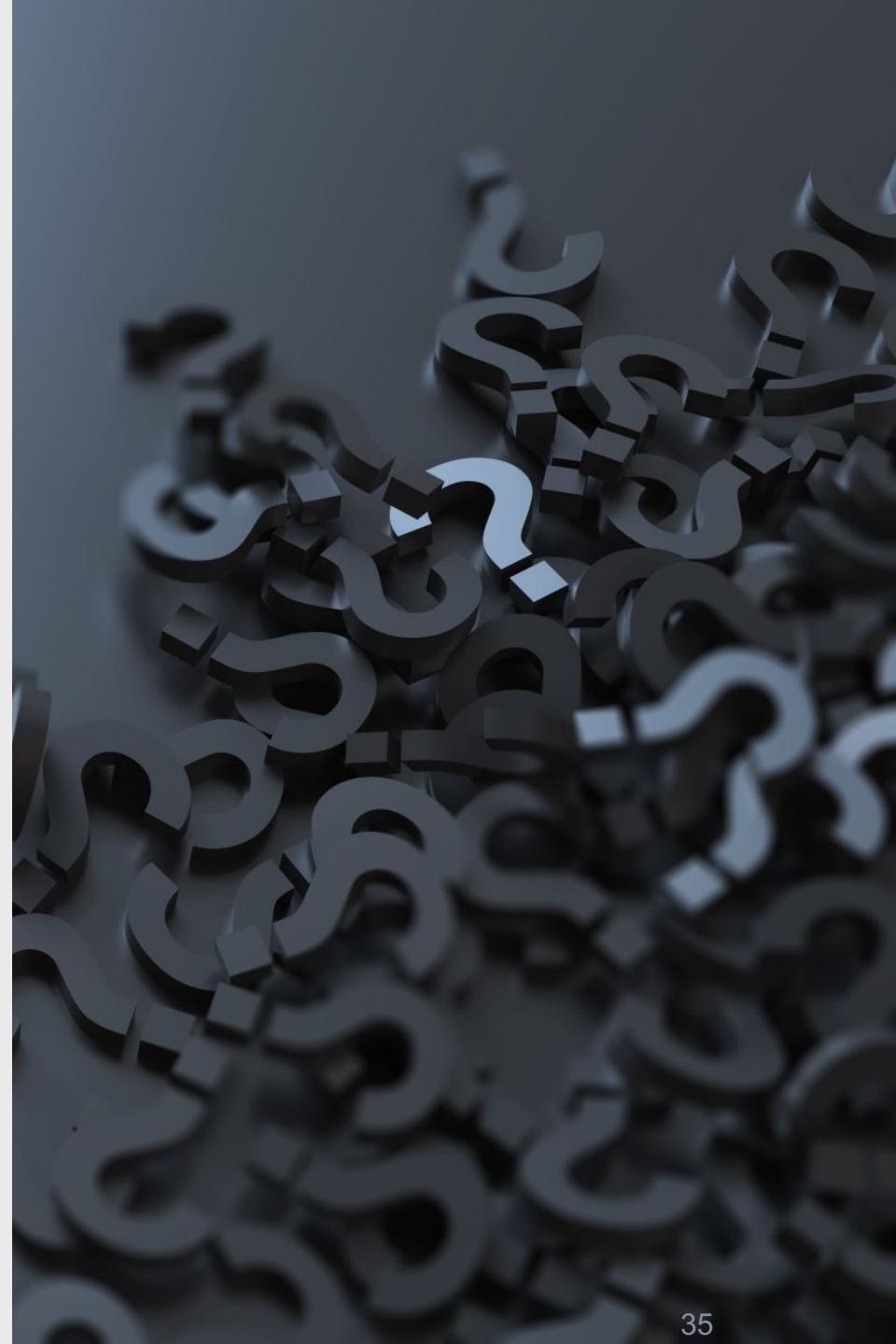
this	sunday,	watch	the	super	bowl	at	5:30	p.m.
c1	c2	t	c3	c4				
	P(+ super, watch) = .7	P(+ super, the) = .5		P(+ super, bowl) = .9		P(+ super at) = .5		

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

- 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
- We do so by applying the logistic function to the dot product of the embeddings of t with each context word c

Computing $P(+ | t, c)$ and $P(- | t, c)$: ✓

- 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: ✓

- Input words are 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
---	---	---	---	---	---	---	---	---	---

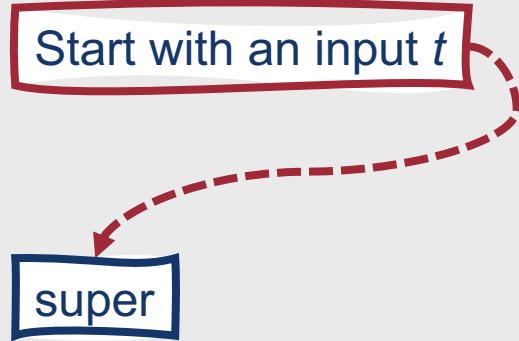
bowl

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

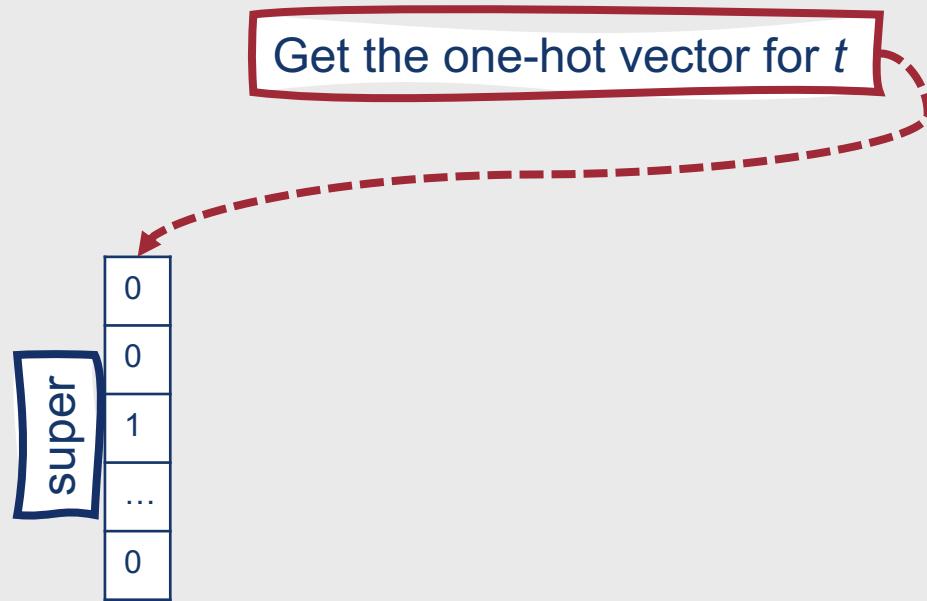
Learned Embeddings....

- 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:
 - $P(+ | t, c) = \frac{1}{1+e^{-t \cdot c}}$

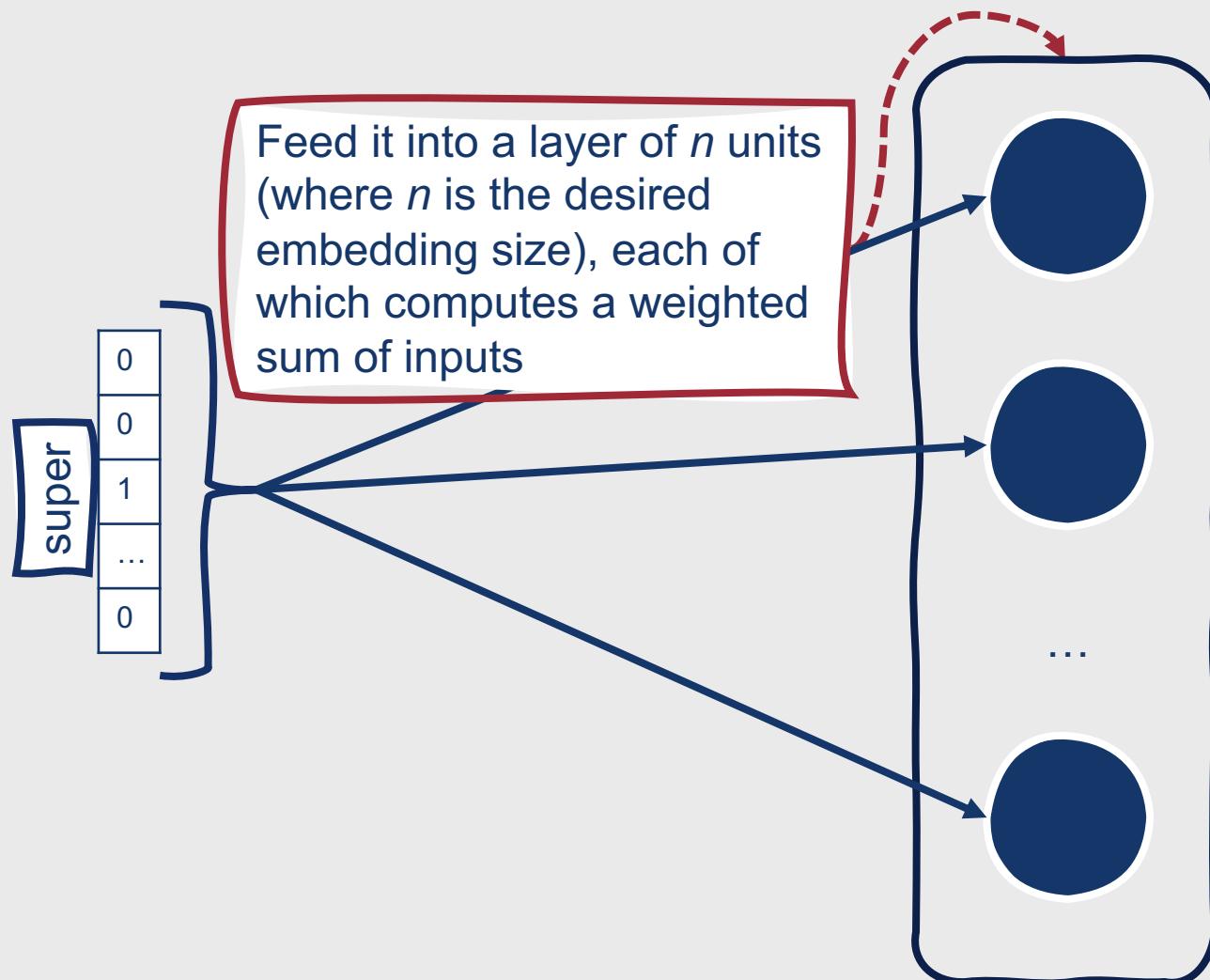
What does this look like?



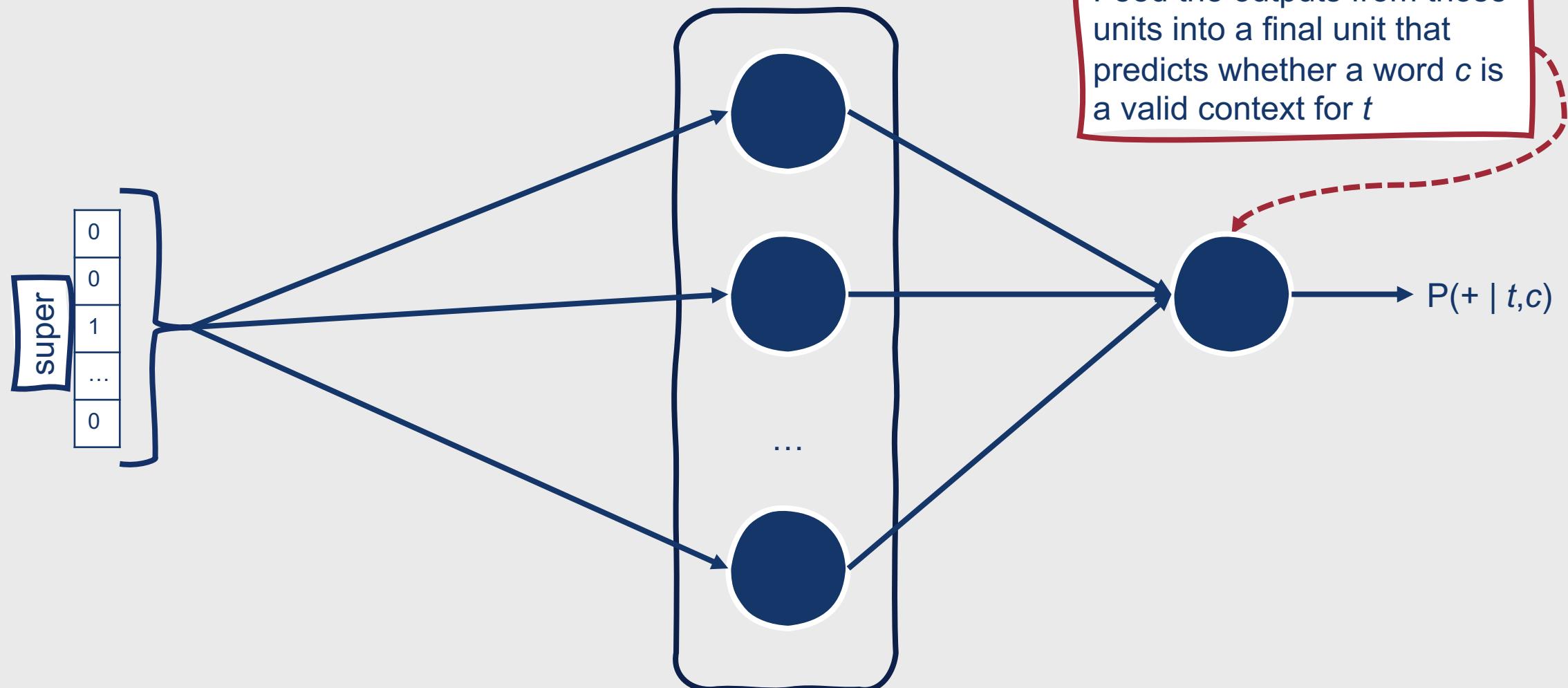
What does this look like?



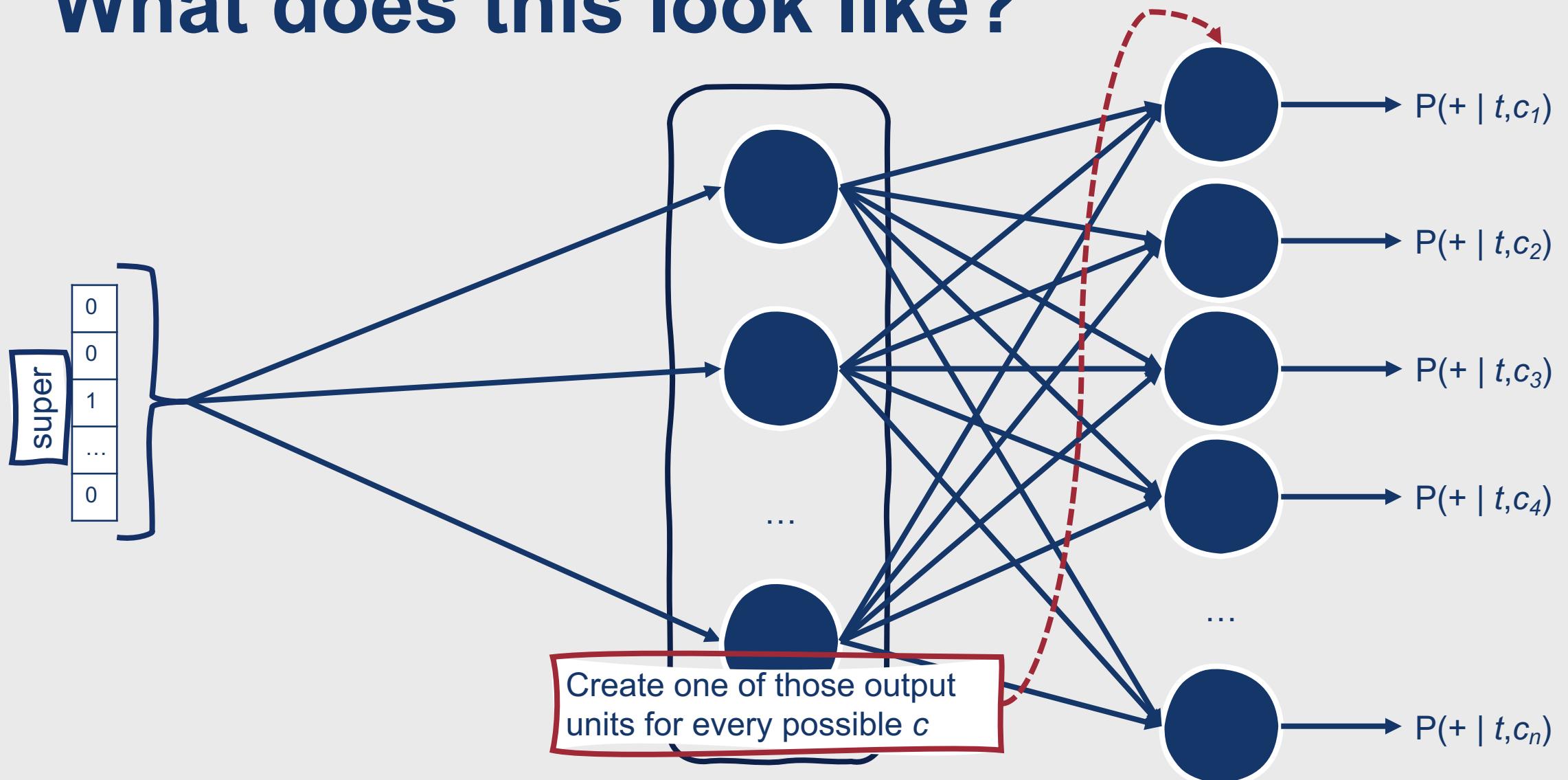
What does this look like?



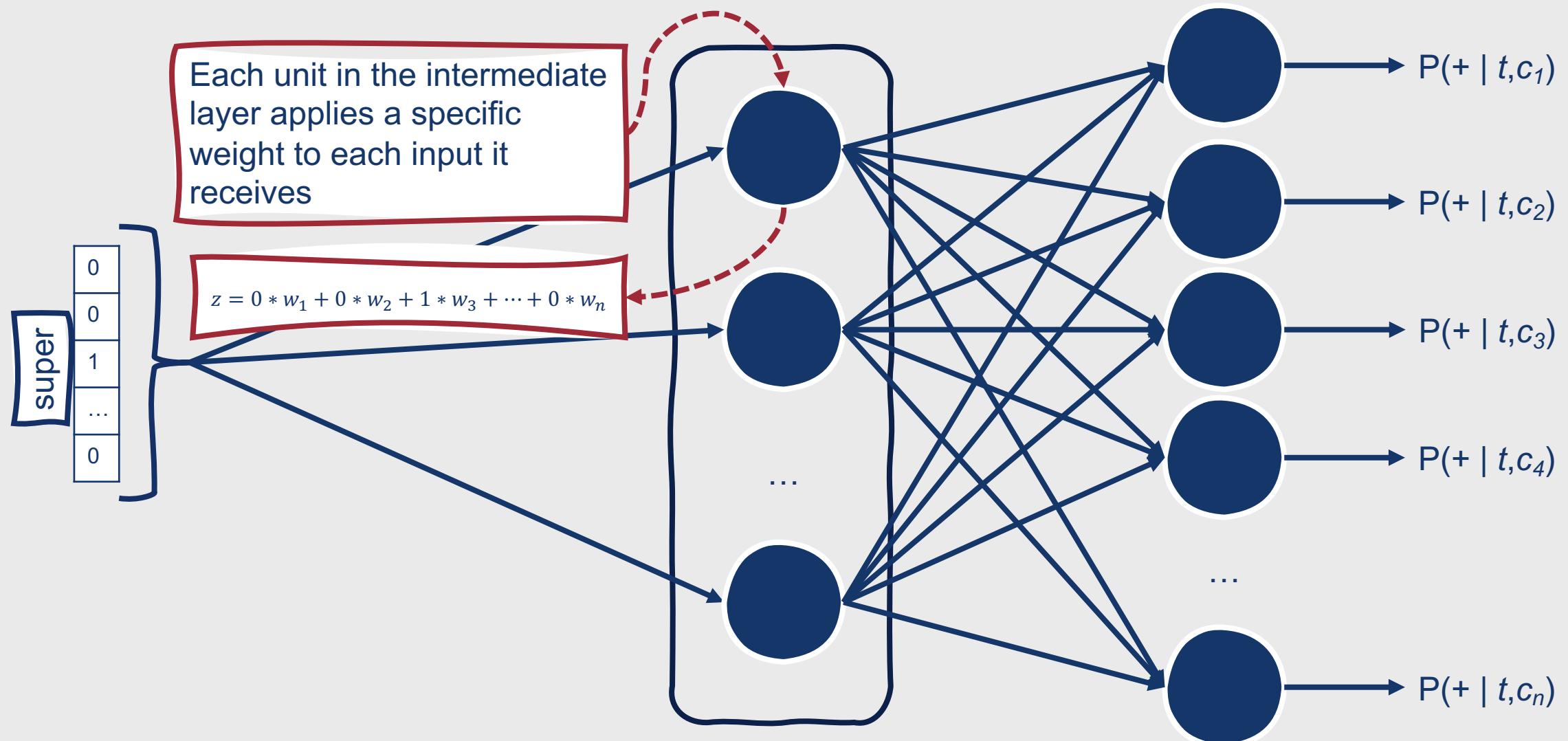
What does this look like?



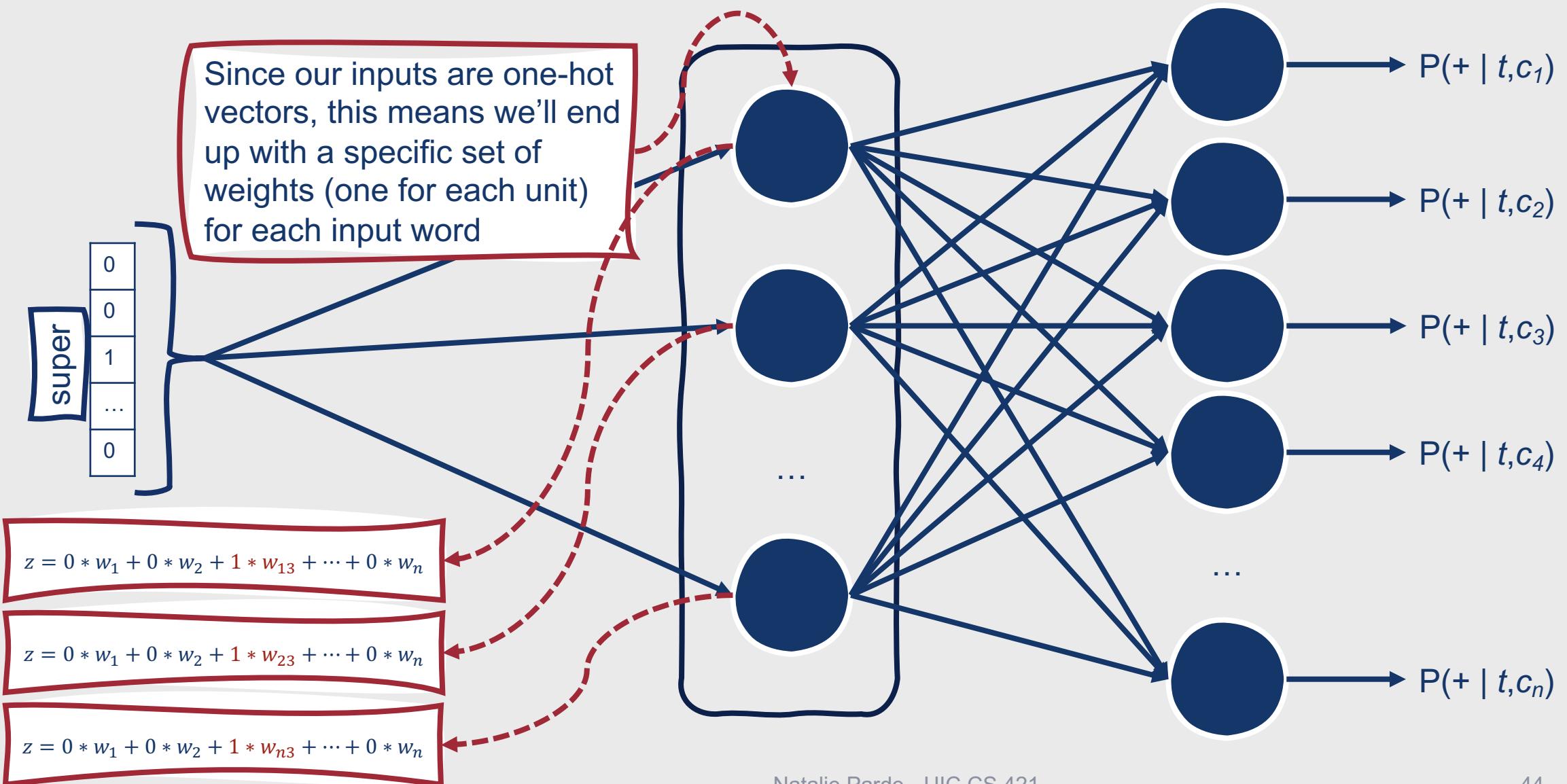
What does this look like?



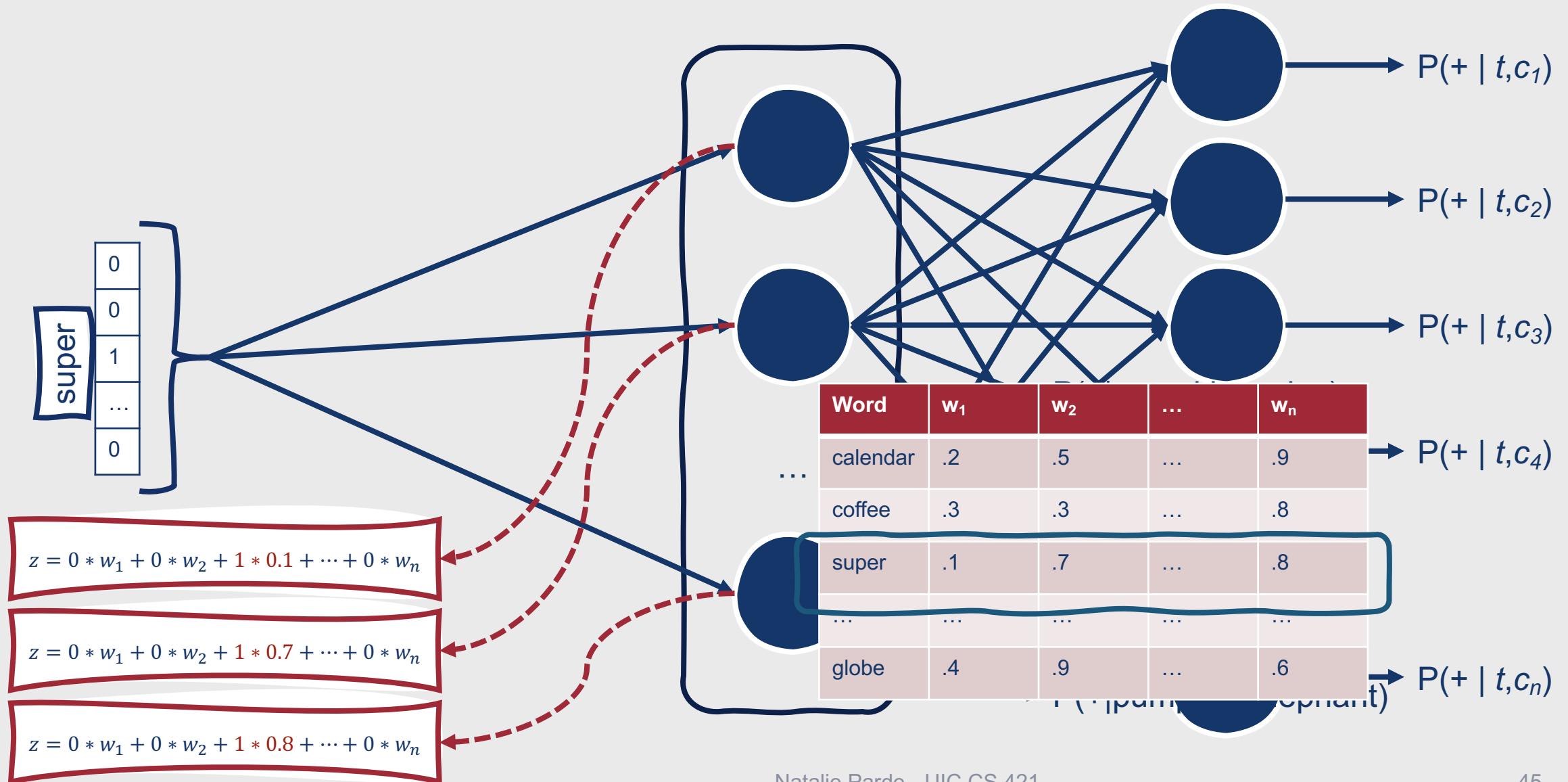
Behind the scenes....



Behind the scenes....



These are the weights we're interested in! ✓

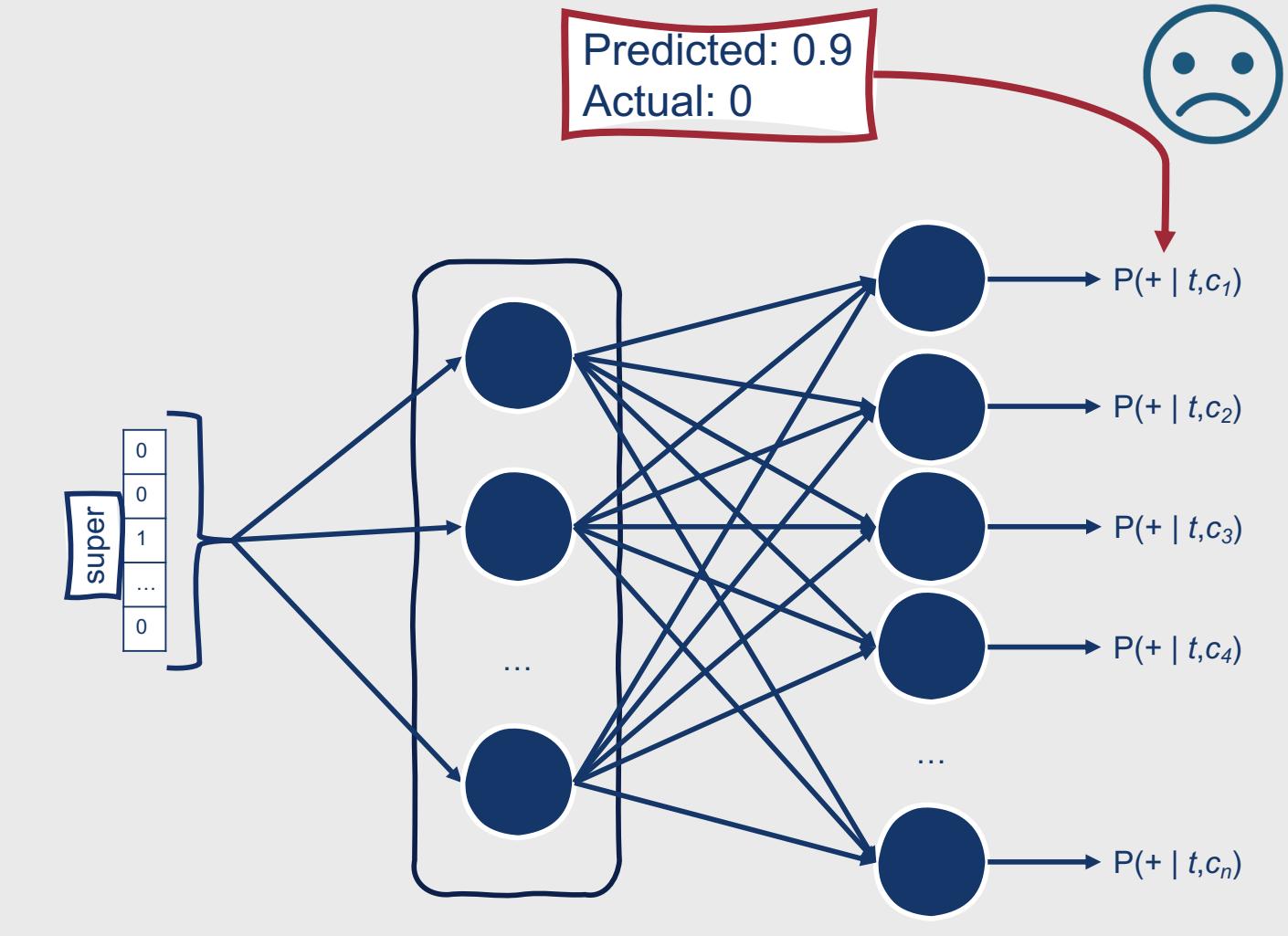




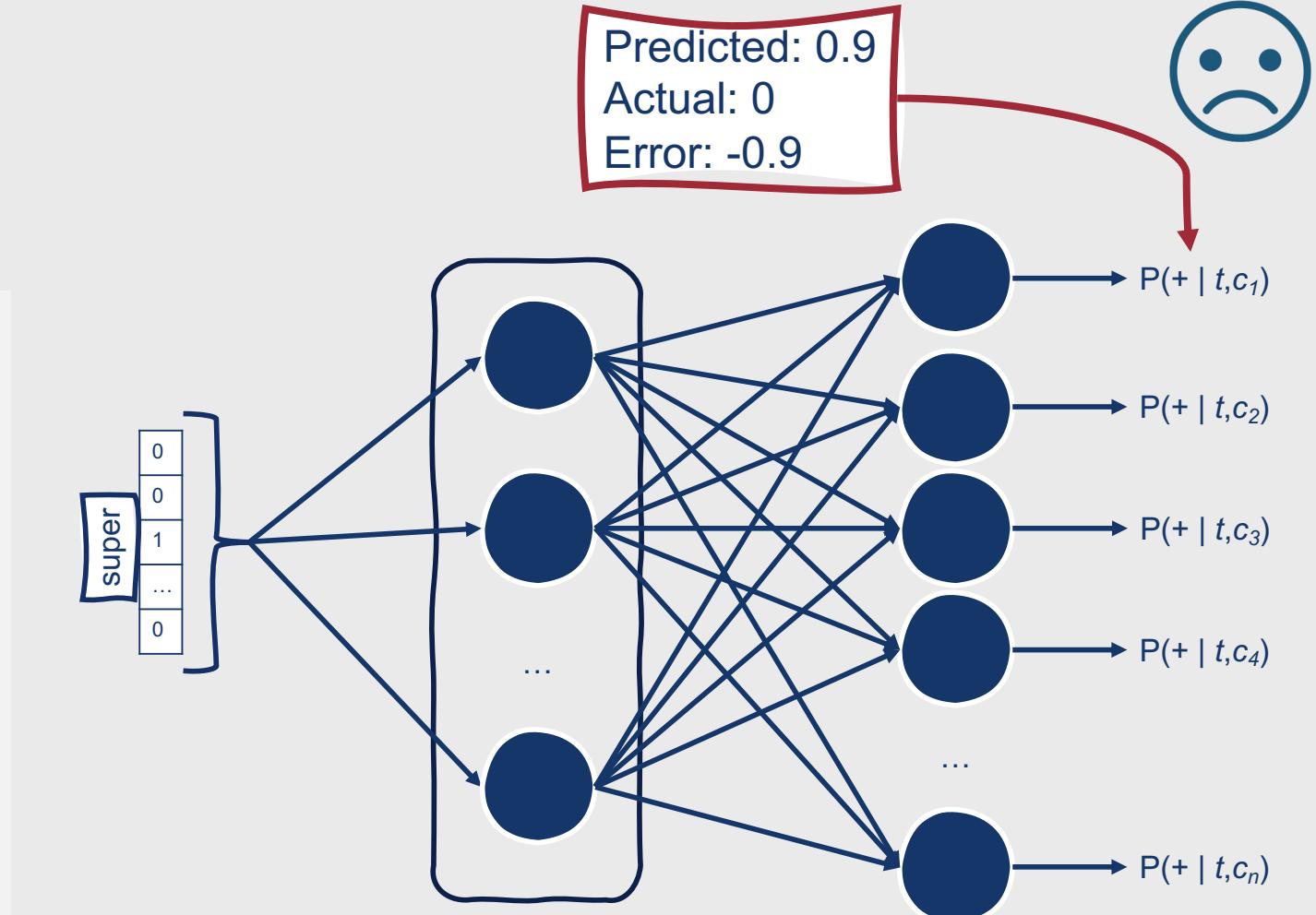
How do we optimize these weights over time?

- The 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

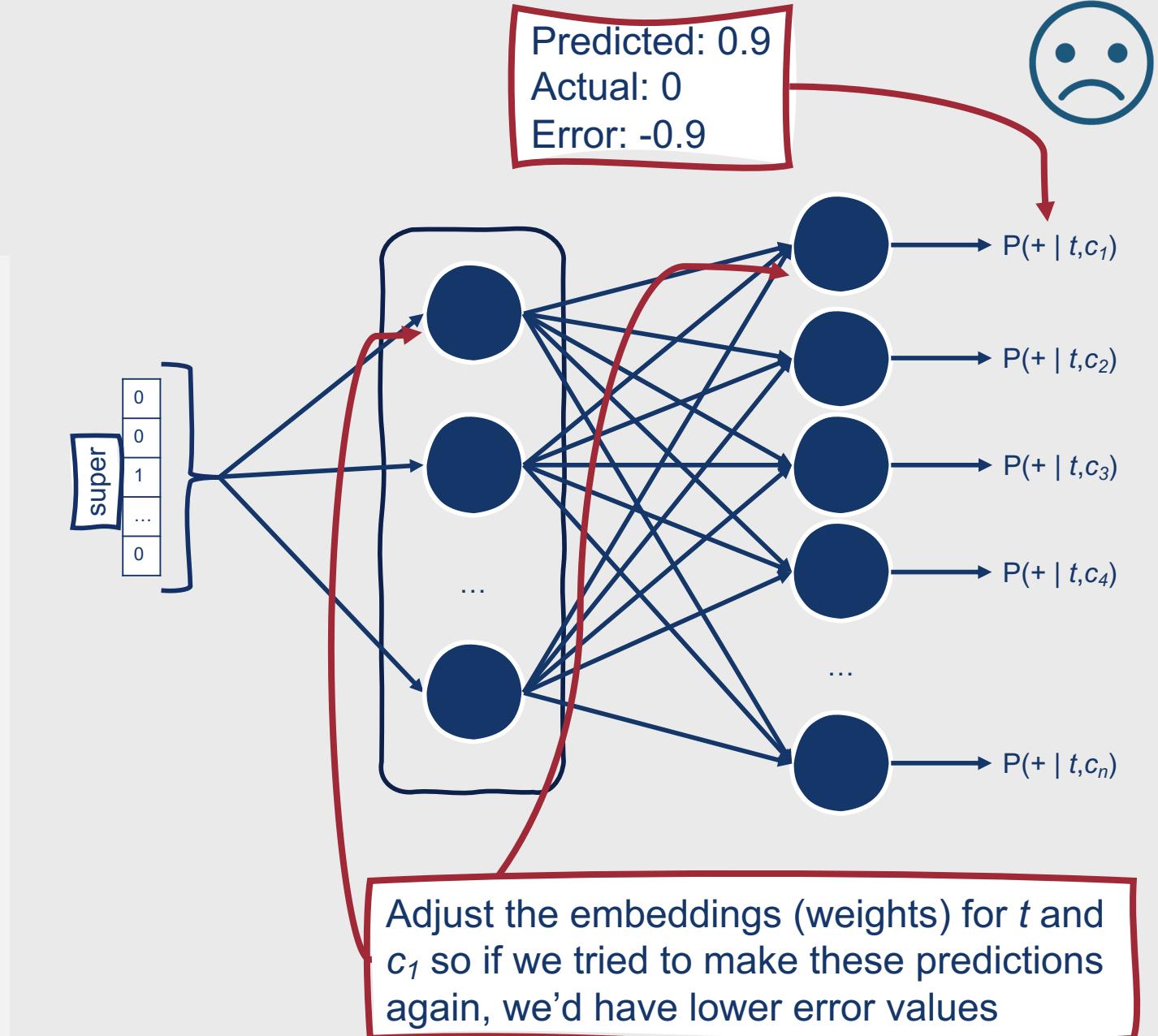
Since we initialize our weights randomly, the classifier's first prediction will almost certainly be wrong.



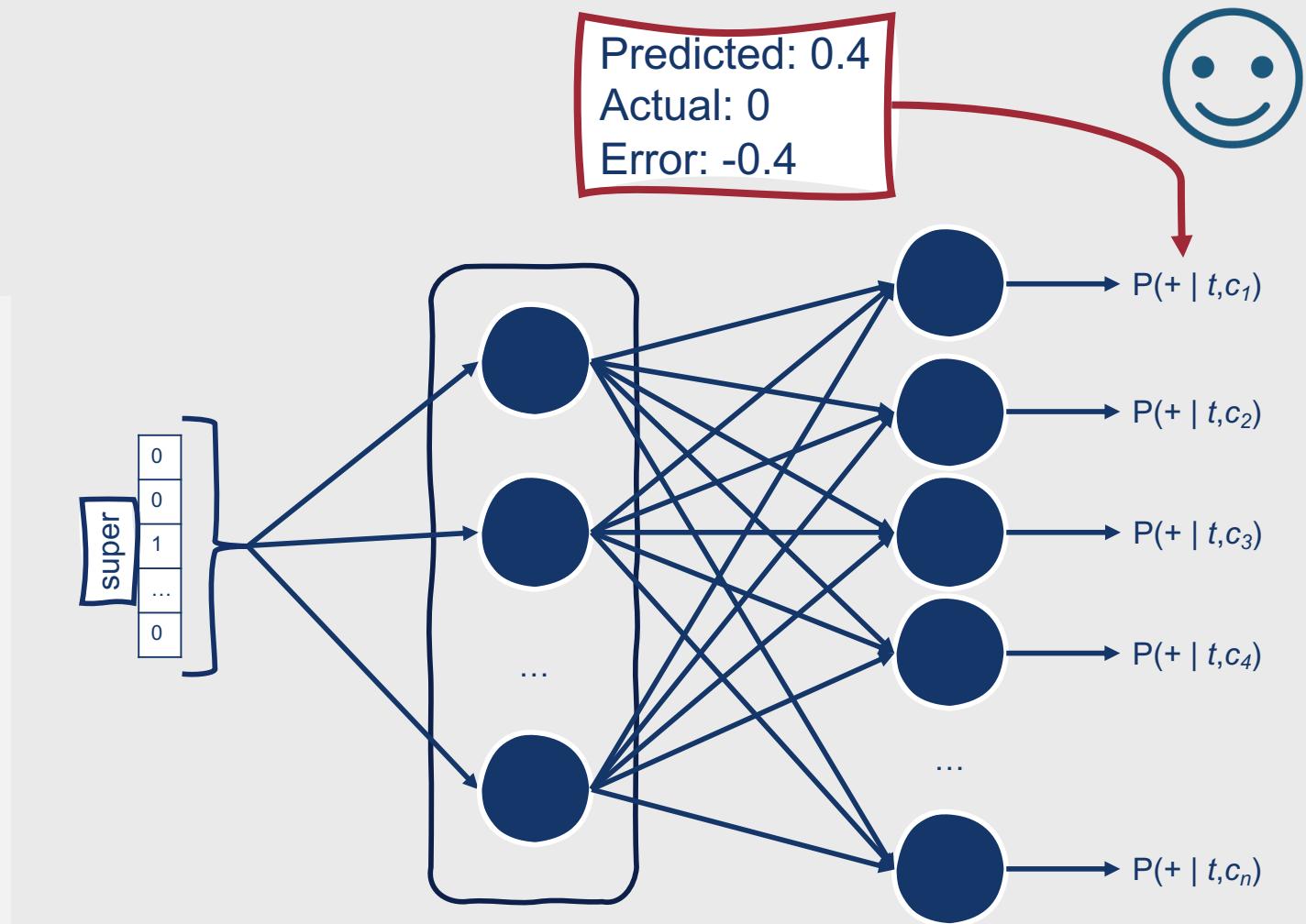
However, the error values from our incorrect guesses are what allow us to improve our embeddings over time.



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 are able to 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?

- 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:
 - $$p_\alpha(w) = \frac{\text{count}(w)^\alpha}{\sum_{w'} \text{count}(w')^\alpha}$$

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



What is our training data?

- How to create negative examples?
 - Often, $\alpha = 0.75$ to give rarer noise words slightly higher probability of being randomly sampled
 - Assuming we want twice as many negative samples as positive samples, we can thus randomly select noise words according to weighted unigram frequency

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

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

Learning Skip-Gram Embeddings

- Even though we're maintaining two embeddings for each word during training (the target vector and the context vector), we only need one of them
- When we're finished learning the embeddings, we can just discard the context vector
- Alternately, we can add them together to create a **summed embedding** of the same dimensionality, or we can **concatenate them into a longer embedding** with twice as many dimensions



Context window size can impact performance!

- Because of this, context window size is often tuned on a validation or development set
- Larger window size → more required computations (important to consider when using very large datasets)



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!
 - The difference:
 - Instead of learning to predict a context word from a target word vector, you learn to **predict a target word from a set of context word vectors**

Skip-Gram vs. CBOW Embeddings

In general, skip-gram embeddings are good with:

- Small datasets
- Rare words and phrases

CBOW embeddings are good with:

- Larger datasets (they're faster to train)
- Frequent words

Are there any other variations of Word2Vec?

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

fastText

- Each word is represented as:

- Itself
- A bag of constituent n-grams

$$\text{super} = \langle \text{super} \rangle + \langle \text{su, sup, upe, per, er} \rangle$$



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/>

Word2Vec and fastText embeddings are nice ...but what's another alternative?

- 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
- Intuitive solution to increase scalability?
 - Dimensionality reduction!
 - However, typical dimensionality reduction strategies may result in too much computational overhead
 - GloVe learns to predict weights in a lower-dimensional space that correspond to the co-occurrence probabilities between words

GloVe

- Why is this useful?
 - Predictive models → black box
 - They work, but why?
 - GloVe models are easier to interpret
- GloVe models also encode the ratios of co-occurrence probabilities between different words ...this makes these vectors particularly useful for word analogy tasks

How does GloVe work?

	c_1	\dots	c_n
t_1	123	\dots	456
\dots	\dots	\dots	\dots
t_n	0	\dots	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?

	c₁	...	c_n
t₁	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?

	c₁	...	c_n
t₁	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?

	c₁	...	c_n
t₁	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

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

0.4 0.7 1.2 4.3 0.9 6.7 1.3 0.5 0.7 5.3

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

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 best ...Word2Vec or GloVe?

- It depends on your data!
- 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/>)

Summary: Word2Vec and GloVe

- **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**
- **fastText** is an extension of Word2Vec that also incorporates **subword models**
- **GloVe** is a **count-based** word embedding approach that learns an optimized, lower-dimensional version of a co-occurrence matrix

A large, abstract graphic on the left side of the slide features a complex pattern of overlapping triangles in shades of blue, purple, and white, creating a sense of depth and geometric complexity.

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



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

Other Common Evaluation Tasks

Semantic Textual Similarity

- Evaluates the performance of sentence-level similarity algorithms, rather than word-level similarity

Analogy

- Evaluates the performance of algorithms at solving analogies
 - Chicago is to Illinois as Omaha is to (Nebraska)
 - Embedding is to embeddings as assignment is to (assignments)



Semantic Properties of Embeddings

- Major advantage of dense word embeddings: Ability to capture elements of meaning
- Context window size impacts what type of meaning is captured
 - 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., waiter and menu, rather than spoon and fork)

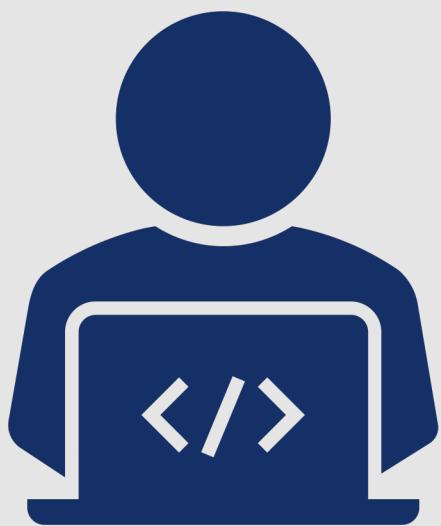


Analogy

- Word embeddings can also capture **relational meanings**
- This is done 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

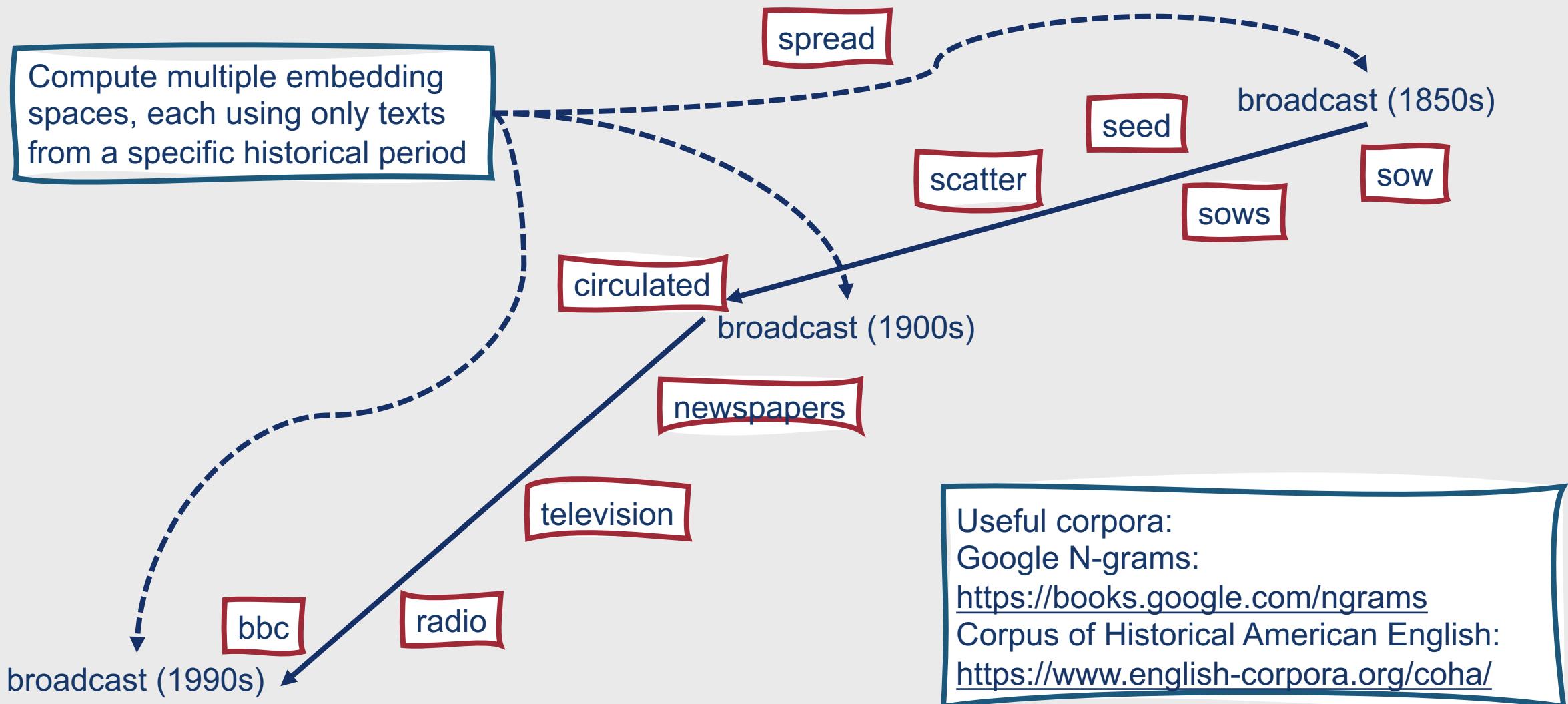


Word embeddings have many practical applications.



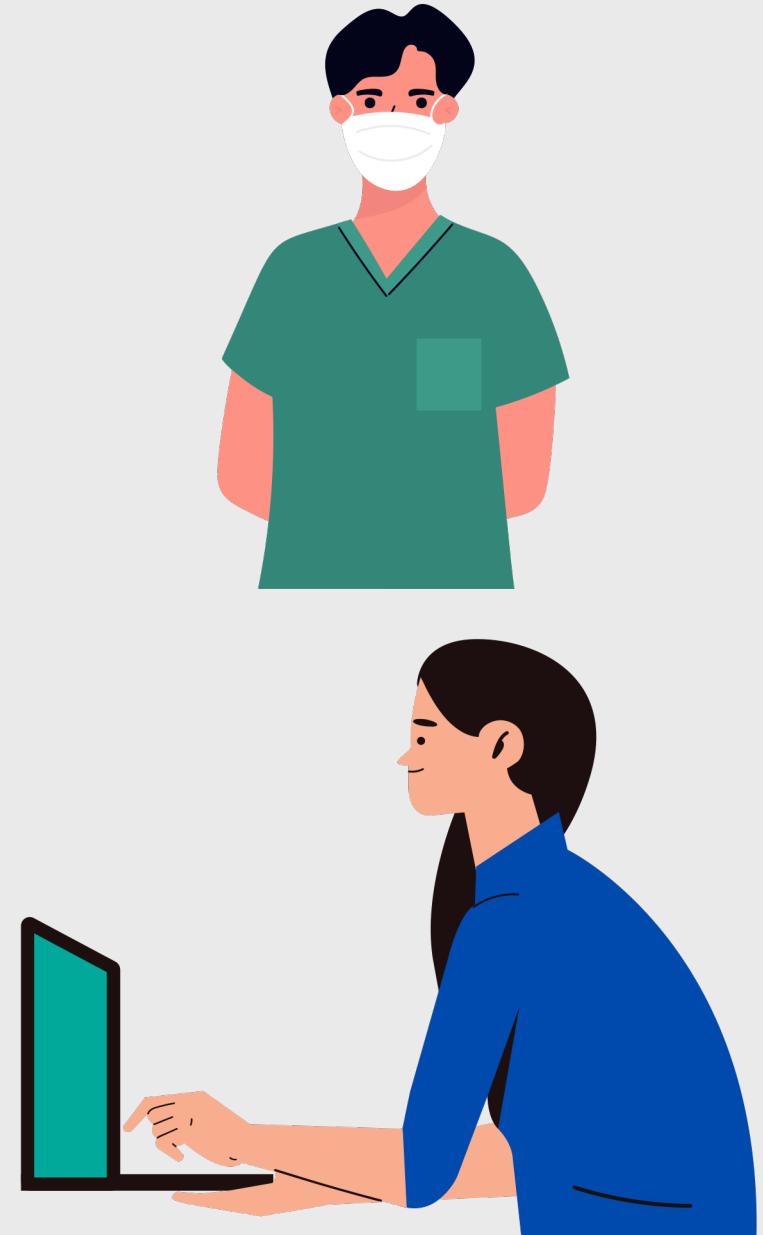
- Incorporated as features in nearly every modern NLP task
- Useful for computational social science
 - Studying word meaning over time
 - Studying implicit associations between words

Embeddings and Historical Semantics



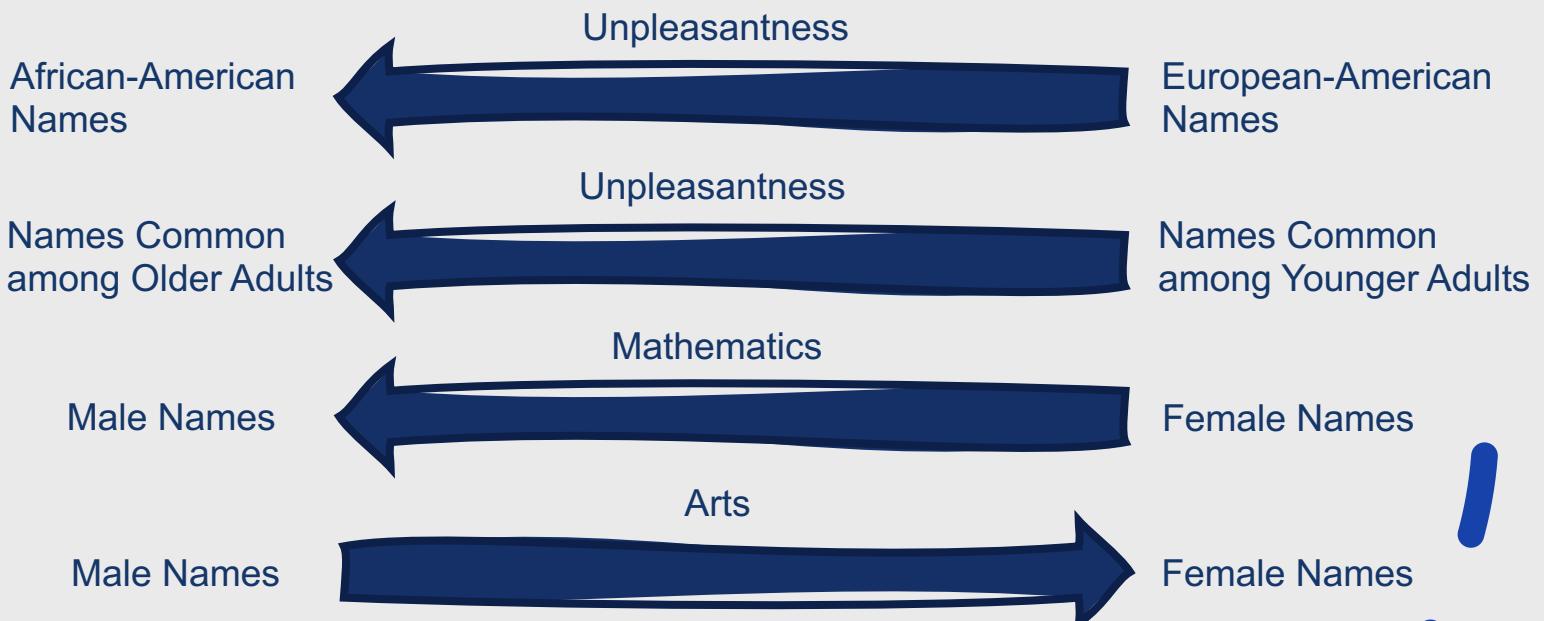
Unfortunately, word embeddings can also end up reproducing implicit biases and stereotypes latent in text.

- 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., resume scoring models)



Bias and Embeddings

- Caliskan et al. (2017) identified known, harmful implicit associations in GloVe embeddings
- Thus, learning word representations is an ethically complex topic!



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:
 - Developing transformations of embedding spaces that remove gender stereotypes but preserve definitional gender
 - Changing training procedures to eliminate these issues before they arise
- Although these methods reduce bias, they do not eliminate it
- Increasingly active area of study:
 - <https://factconference.org>



**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 networks**

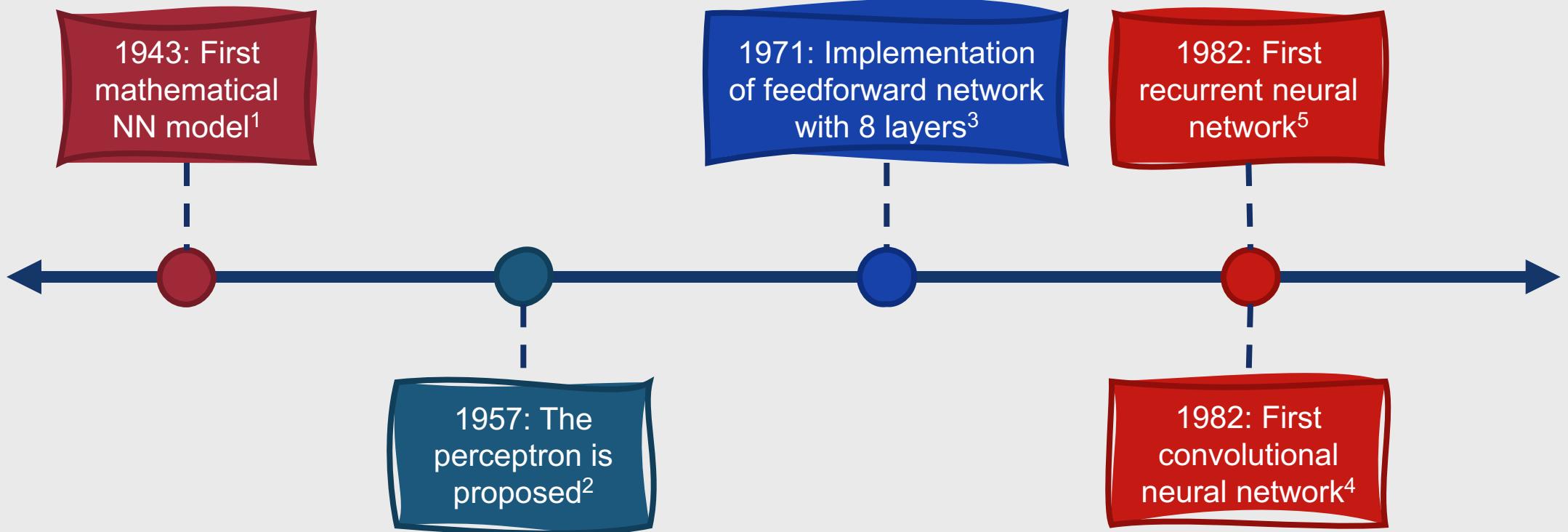
What are neural networks?

- Classification models comprised of interconnected computing units, or **neurons**, (loosely!) mirroring the interconnected neurons in the human brain

**Neural networks
are fundamental to
many modern NLP
tasks.**

ACL Year	# Paper Titles with “Neural”	% Paper Titles with “Neural”
2000	0	0
2001	0	0
2002	0	0
2003	0	0
2004	1	1/89 = 1.1%
2005	0	0
2006	0	0
2007	1	1/132 = 0.8%
2008	0	0
2009	1	1/216 = 0.5%
2010	0	0
2011	0	0
2012	0	0
2013	5	5/330 = 1.5%
2014	11	11/288 = 3.8%
2015	37	37/320 = 11.6%
2016	47	47/330 = 14.2%
2017	77	77/304 = 25.3%
2018	81	81/383 = 21.1%
2019	108	108/661 = 16.3%
2020	93	93/779 = 11.9%
2021	68	68/712 = 9.55%
2022	47	47/701 = 6.7%

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.

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.

Natalie Parde - UIC CS 421

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

- Data
- Computing power



Natalie Parde - UIC CS 421



Neural networks are everywhere!

Augmenting Neural Networks with First-order Logic

Tao Li
University of Utah
tli@cs.utah.edu

Vivek Srikumar
University of Utah
svivek@cs.utah.edu

Abstract

Today, the dominant paradigm for training neural networks involves minimizing task loss on a large dataset. Using world knowledge to inform a model, and yet retain the ability to perform end-to-end training remains an open question. In this paper, we present a novel framework for introducing declarative knowledge to neural network architectures in order to guide training and prediction. Our framework systematically compiles logical statements into computation graphs that augment

Paragraph: Gaius Julius Caesar (July 100 BC – 15 March 44 BC) **Roman general**, statesman, Consul and notable **military leader**. He played a crucial role in the events that led to the decline of the Roman Republic and the rise of the Roman Empire through his various military campaigns.

Question: Which **Roman general** is known for **writing books**?

Figure 1: An example of reading comprehension that illustrates alignments/attention. In this paper, we consider the problem of incorporating external knowledge about such alignments into training neural networks.

Neural Relation Extraction for Knowledge Base Enrichment

Bayu Distiwana Trisedy¹, Gerhard Weikum², Jianzhong Qi¹, Rui Zhang^{1*}

¹ The University of Melbourne, Australia

² Max Planck Institute for Informatics, Saarland Informatics Campus, Germany
btrisedy@student.unimelb.edu.au, rui.zhang@unimelb.edu.au, weikum@mpi-inf.mpg.de

Abstract

We study relation extraction for knowledge base (KB) enrichment. Specifically, we aim to extract entities and their relationships from sentences in the form of triples and map the elements of the extracted triples to an existing KB in an end-to-end manner. Previous studies focus on rule extraction tasks and rely on Named Entity Disambiguation (NED) to map triples into the KB space. This way, NED errors may cause extraction errors that affect the overall precision and recall. To address this

Input sentence:
"New York University is a private university in Manhattan."

Unsupervised approach output:
(NYU, is, private university)
(NYU, is, private university, in, Manhattan)

Supervised approach output:
(NYU, instance of, Private University)
(NYU, located in, Manhattan)

Canonicalized output:
(Q49210, P31, Q902104)
(Q49210, P131, Q11299)

Table 1: Relation extraction example.

Cross-Domain Generalization of Neural Constituency Parsers

Daniel Fried^{*}, Nikita Kitaev^{*}, Dan Klein

Computer Science Division
University of California, Berkeley
[\(dfried,kitaev,klein\)@cs.berkeley.edu](mailto:(dfried,kitaev,klein)@cs.berkeley.edu)

Abstract

Neural parsers obtain state-of-the-art results on benchmark treebanks for constituency parsing—but to what degree do they generalize? We present three sets of results that shed light on the generalization of neural parsers in a zero-shot setting: training on trees from one corpus and evaluating on out-of-domain corpora. First, neural and non-neural parsers generalize comparably to new domains. Second, incorporating pre-trained encoder representations into neural parsers substantially improves their performance across all domains, but does not give a larger relative improvement for out-of-domain treebanks. Finally, despite the rich input representations they learn, neural parsers still benefit from structured output

treebanks still transfer to out-of-domain improvements (McClosky et al., 2006).

Is the success of neural constituency parsers (Henderson 2004; Vinyals et al. 2015; Dyer et al. 2016; Cross and Huang 2016; Cho and Charniak 2016; Stern et al. 2017; Liu and Zhang 2017; Kitaev and Klein 2018, *inter alia*) similarly transferable to out-of-domain treebanks? In this work, we focus on *zero-shot generalization*: training parsers on a single treebank (e.g. WSD) and evaluating on a range of broad-coverage, out-of-domain treebanks (e.g. Broz, Frank, and Kokša, 1979; Genia (Tateisi et al., 2005), the English Web Treebank (Petrov and McDonald, 2012)). We ask three questions about zero-shot generalization properties of state-of-the-art neural constituency parsers:

Do Neural Dialog Systems Use the Conversation History Effectively? An Empirical Study

Chinnadhurai Sankar^{1,2,4*}, Sandeep Subramanian^{1,2,5}

Christopher Pal^{1,3,5}, Sarath Chandar^{1,2,4}, Yoshua Bengio^{1,2}

¹Mila

²Université de Montréal

³École Polytechnique de Montréal

⁴Google Research, Brain Team

⁵Element AI, Montréal

Abstract

Neural generative models have been become increasingly popular when building conversational agents. They offer flexibility, can be easily adapted to new domains, and require minimal domain engineering. A common criticism of these systems is that they seldom understand or use the available dialog history effectively. In this paper, we take an empirical approach to understanding how these mod-

they still lack the ability to “understand” and process the dialog history to produce coherent and interesting responses. They often produce boring and repetitive responses like “Thank you.” (Li et al., 2015; Serban et al., 2017a) or meander away from the topic of conversation. This has been often attributed to the manner and extent to which these models use the dialog history when generating responses. However, there has been little empirical investigation to validate these speculations.

Effective Adversarial Regularization for Neural Machine Translation

Motoki Sato¹, Jun Suzuki^{2,3}, Shun Kiyono^{3,2}

¹Preferred Networks, Inc., ²Tohoku University,
³RIKEN Center for Advanced Intelligence Project
sato@preferred.jp, jun.suzuki@ecei.tohoku.ac.jp, shun.kiyono@riken.jp

Abstract

A regularization technique based on adversarial perturbation, which was initially developed in the field of image processing, has been successfully applied to text classification tasks and has yielded attractive improvements. We aim to further leverage this promising methodology into more sophisticated and critical neural models in the natural language processing field, i.e., neural machine translation (NMT) models. However, it is not trivial to apply this

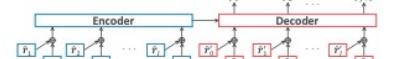


Figure 1: An intuitive sketch that explains how we add adversarial perturbations to a typical NMT model structure for adversarial regularization. The definitions of e_i and f_j can be found in Eq. 2. Moreover, those of \hat{e}_i and \hat{f}_j are in Eq. 8 and 13, respectively.

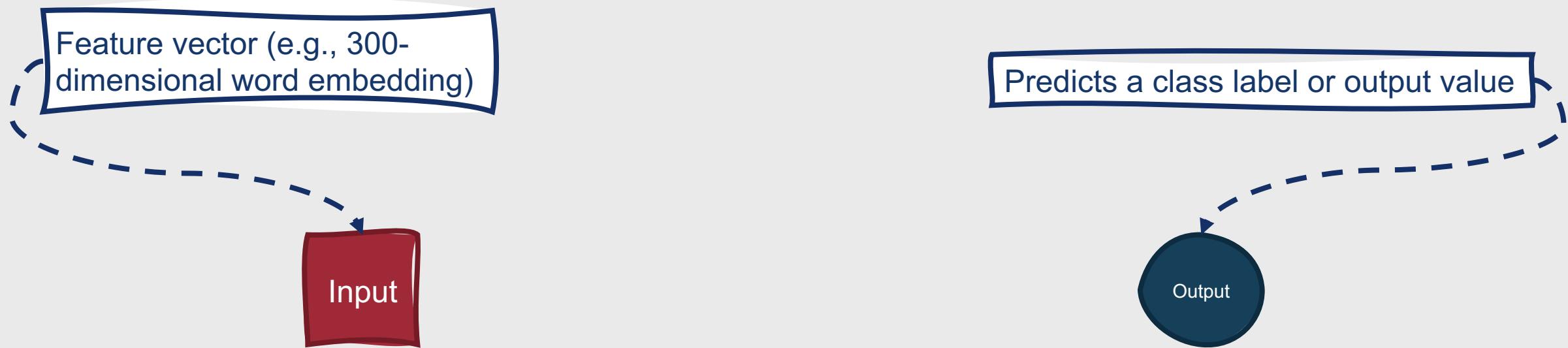
There are many types of neural networks!

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

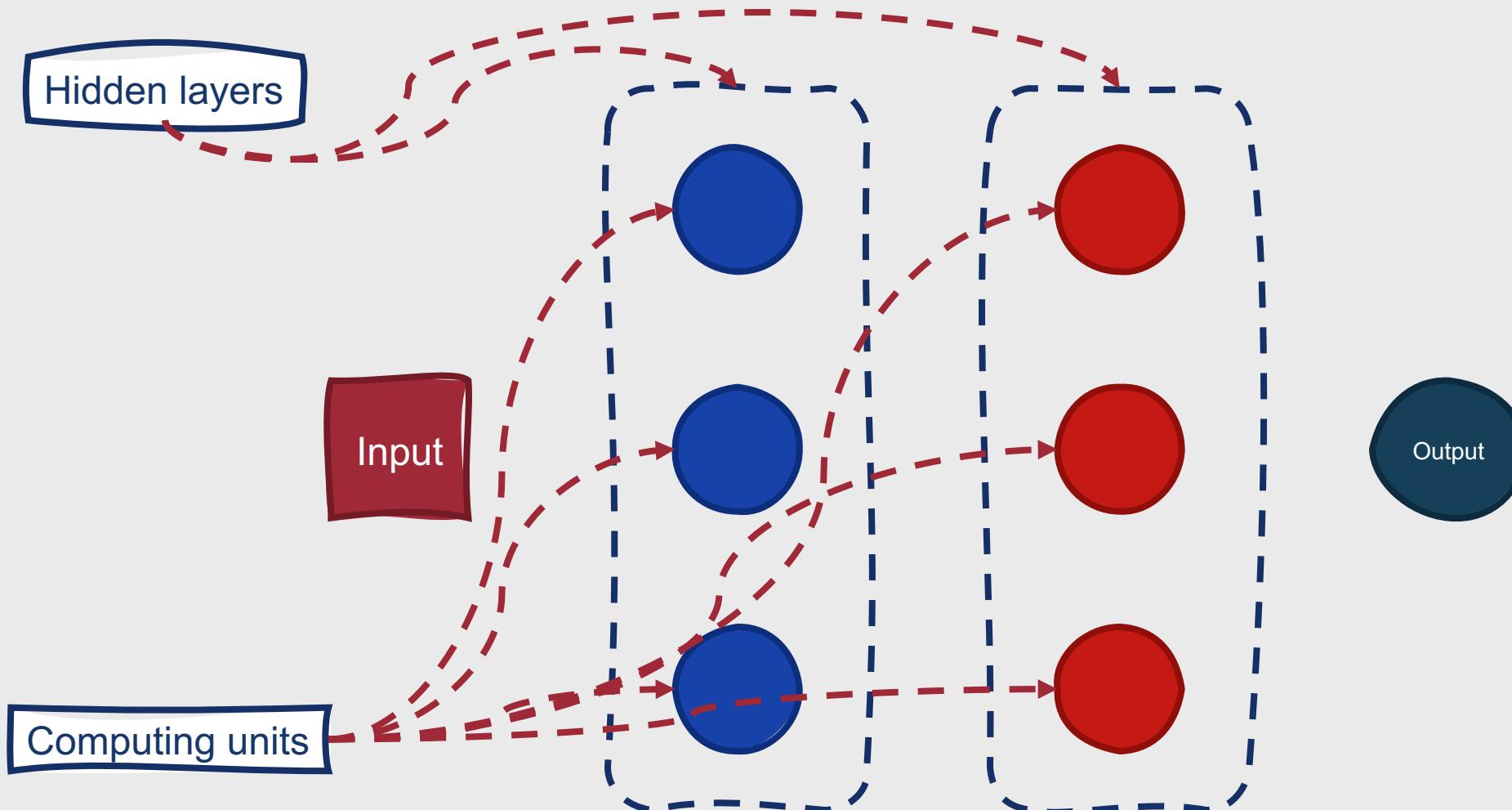
Feedforward Neural Networks

- 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

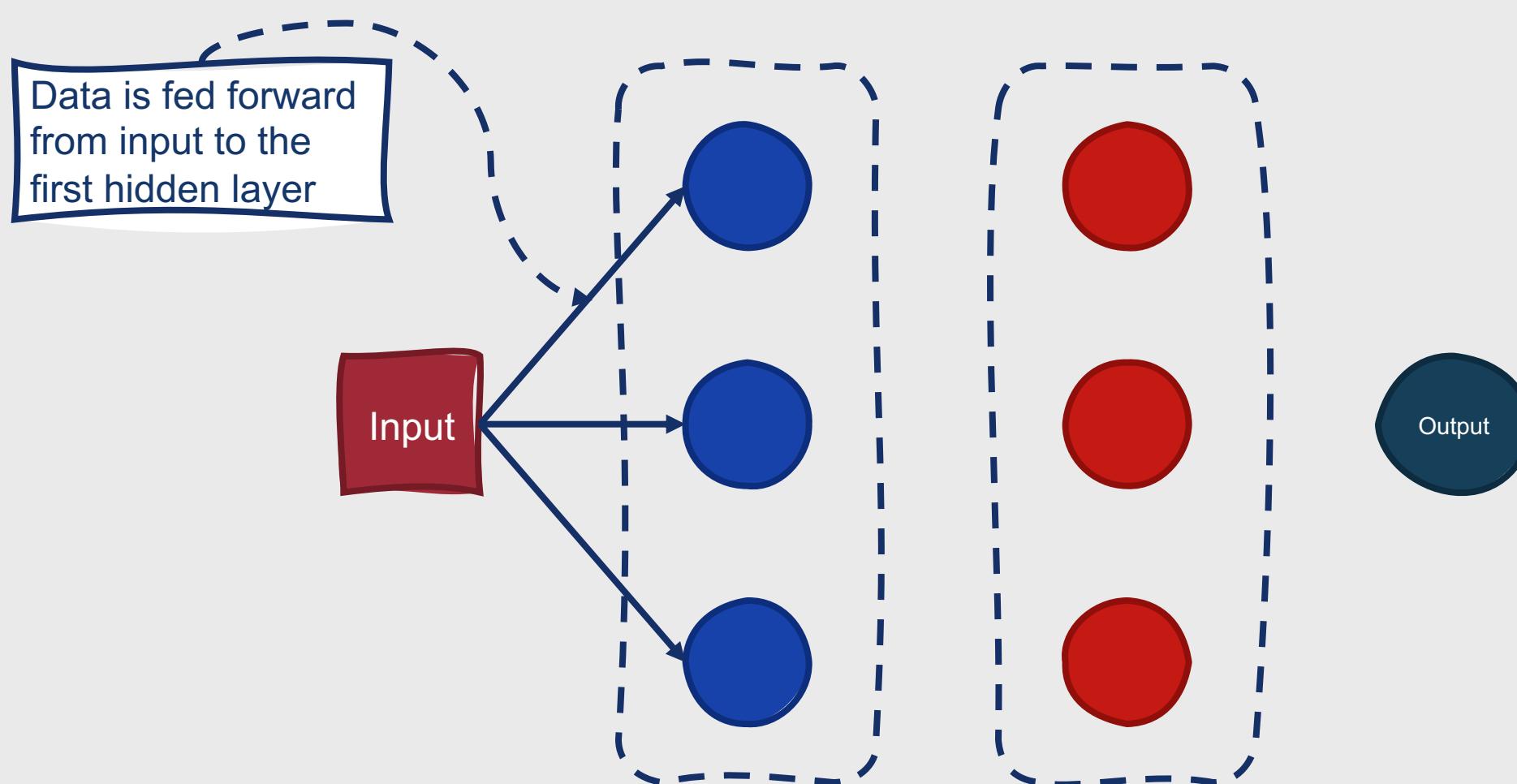
Feedforward Neural Networks



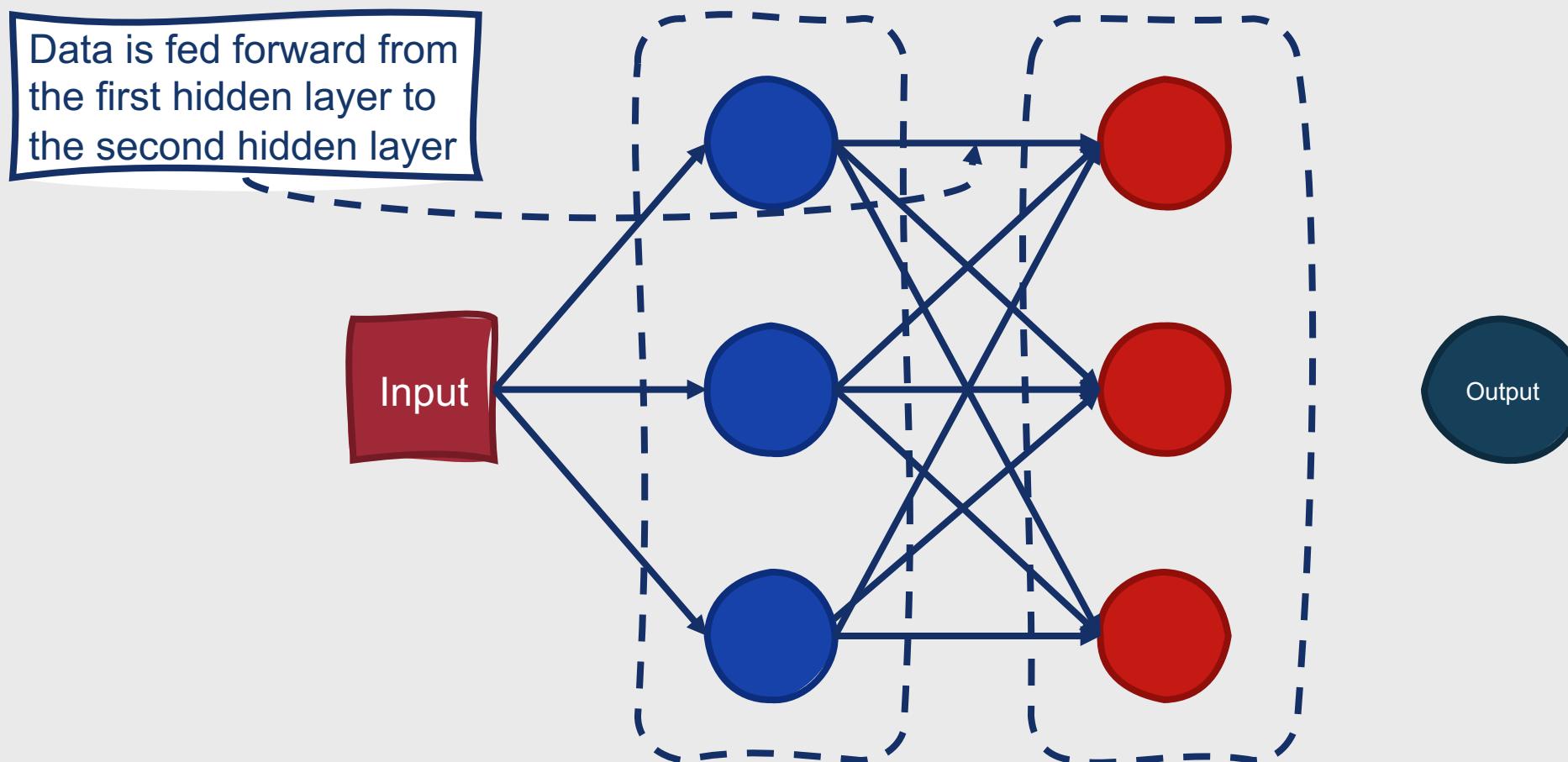
Feedforward Neural Networks



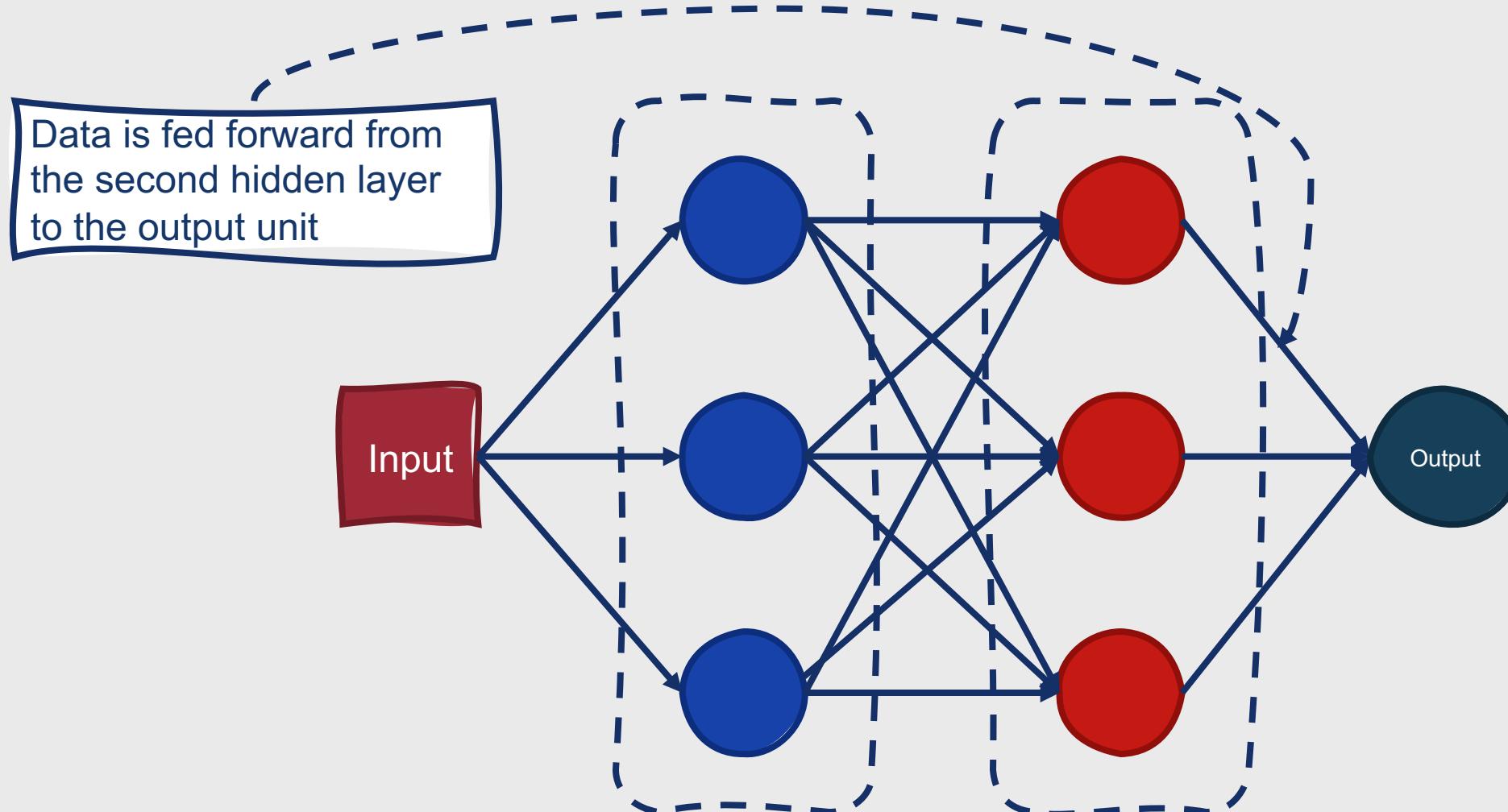
Feedforward Neural Networks



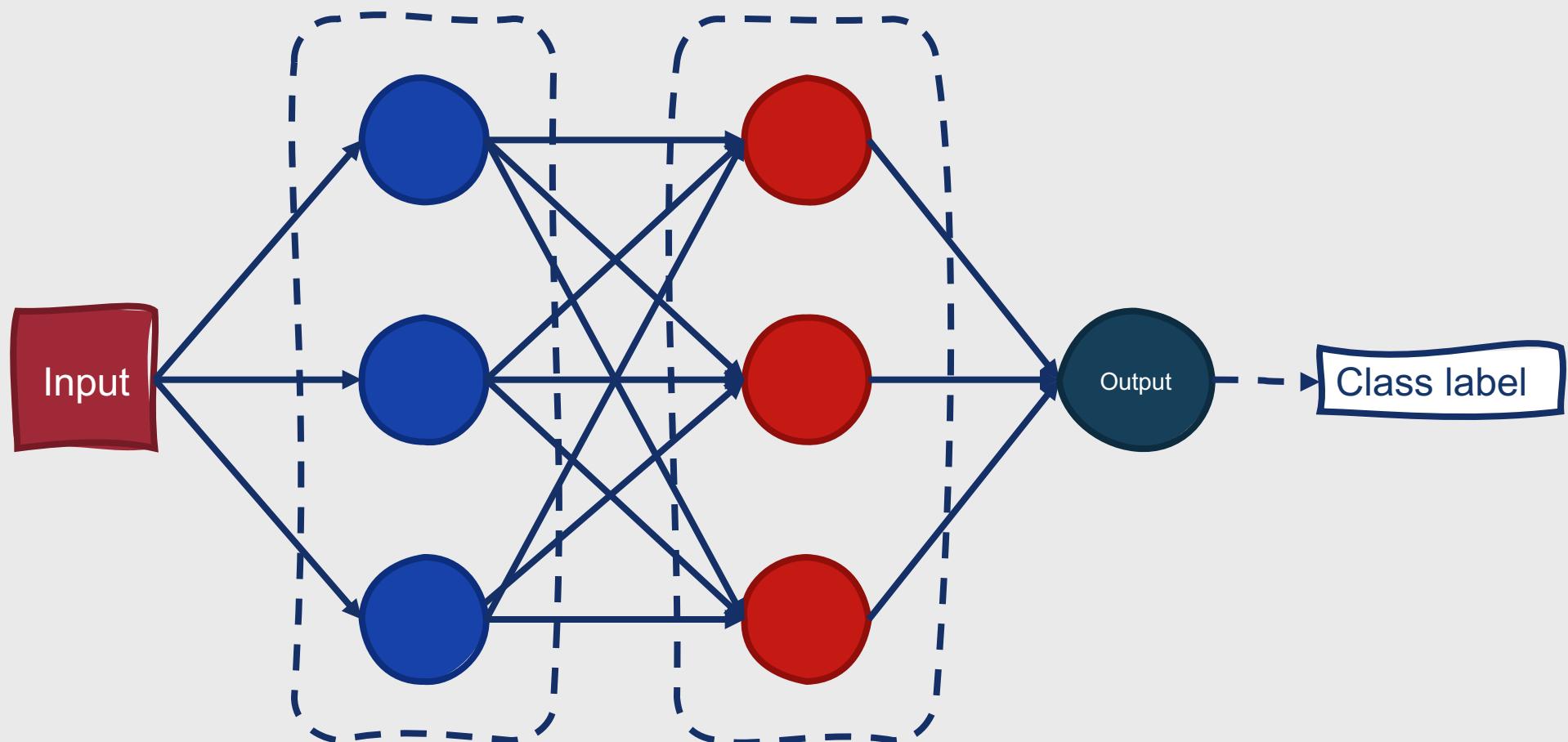
Feedforward Neural Networks

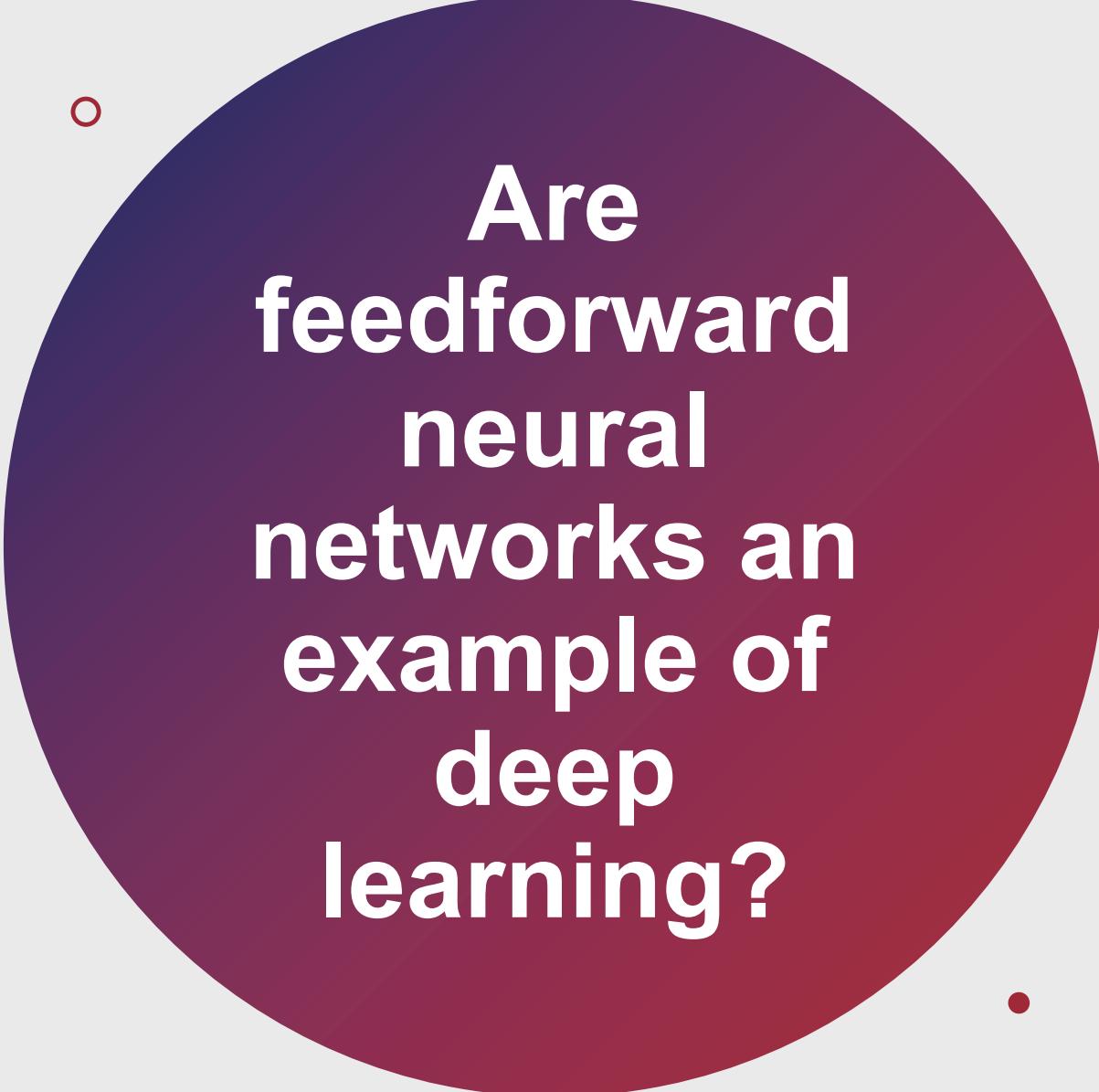


Feedforward Neural Networks



Feedforward Neural Networks

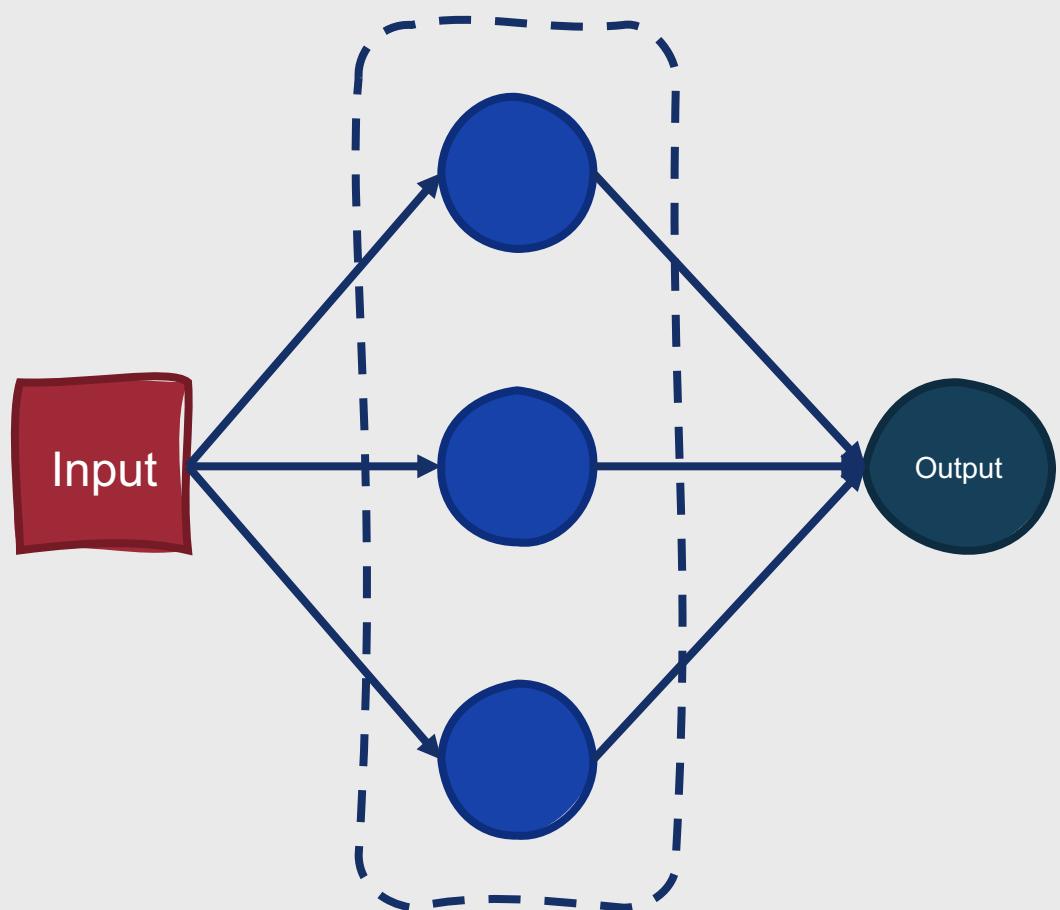




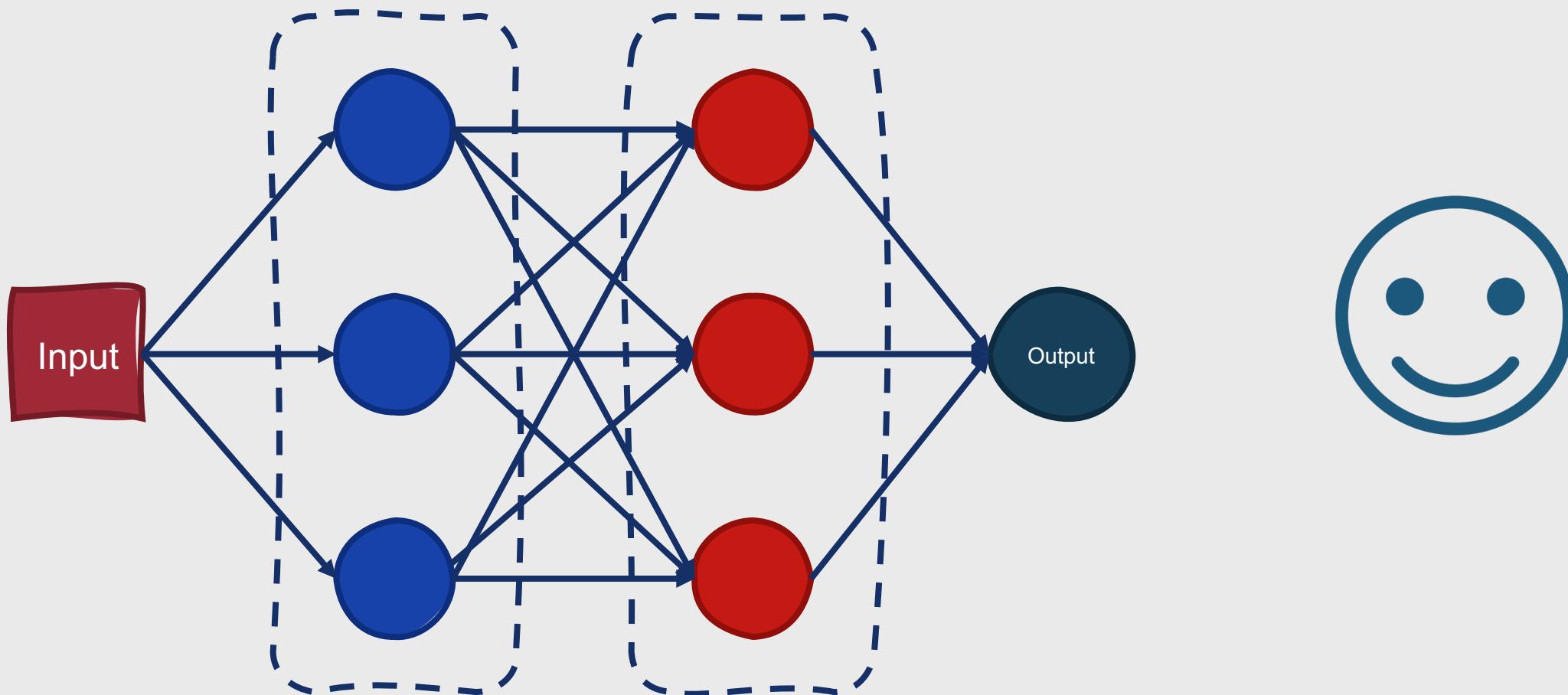
Are feedforward neural networks an example of deep learning?

- Yes ...if they have multiple layers
- People often tend to refer to neural network-based machine learning as **deep learning**
 - Why?
 - Modern networks often have many layers (in other words, they're **deep**)

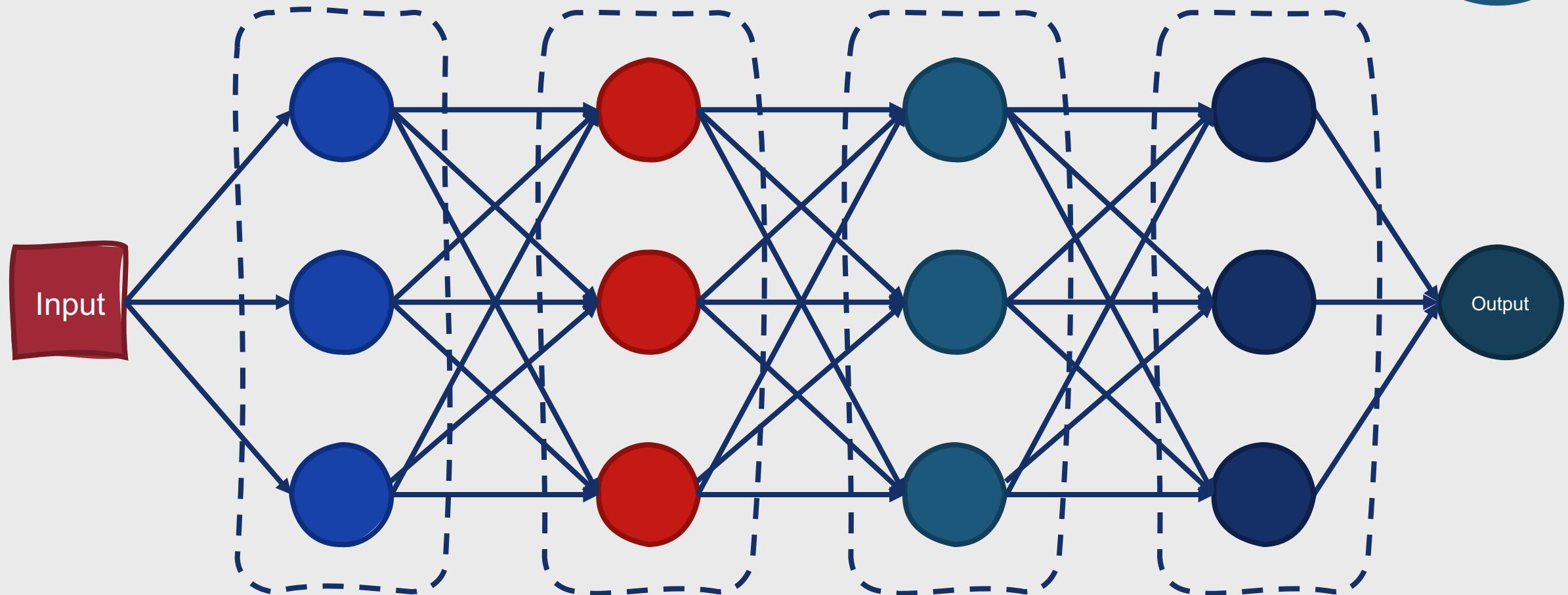
How many layers is “deep?”



How many layers is “deep?”

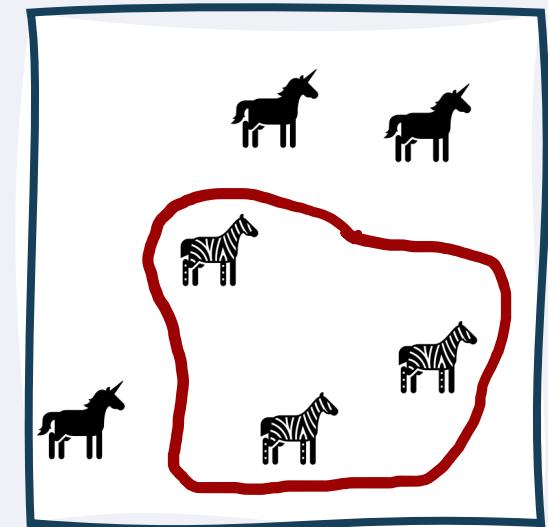
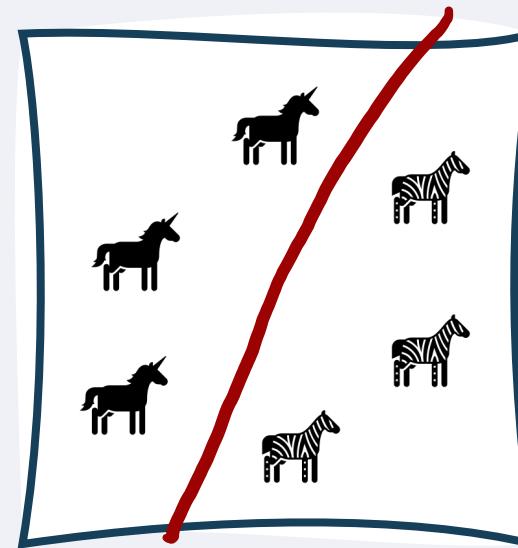


How many layers is “deep?”



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

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



Neural networks also commonly use different types of features from traditional classification algorithms.

Traditional classification

- **Manually engineer** a set of features and extract them for each instance
 - Part-of-speech label
 - Number of exclamation marks
 - Sentiment score

Neural networks

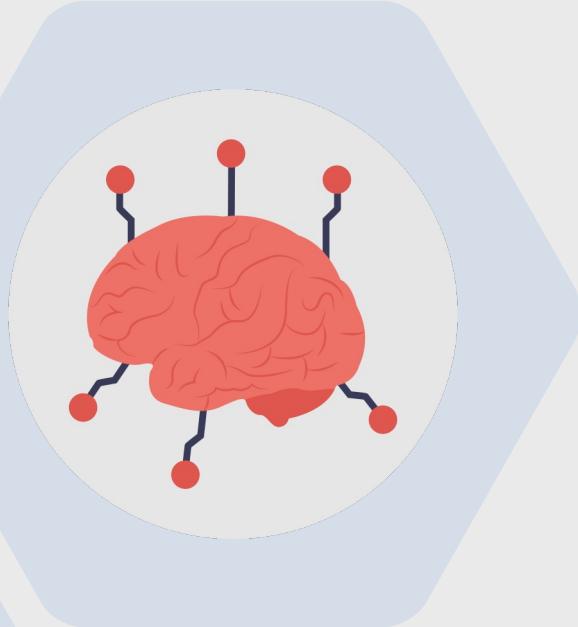
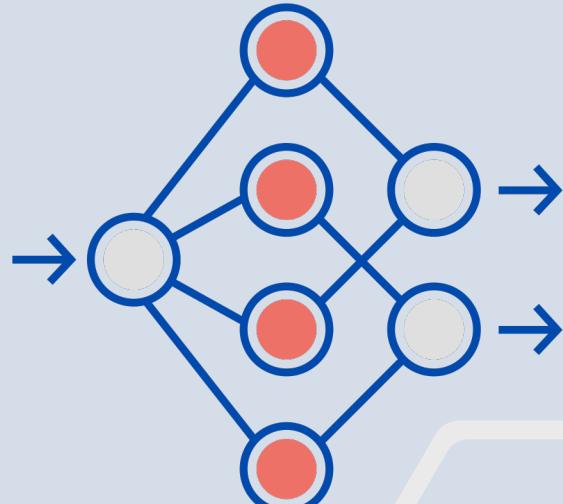
- **Implicitly learn** features and extract those for each instance
 - Word embeddings

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

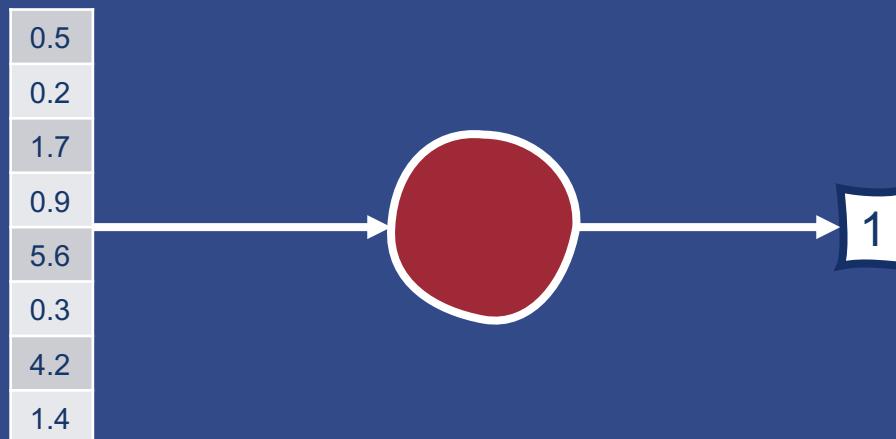
Neural nets tend to work very well
for large-scale problems, but not
as well for small-scale problems



How do you build a neural network?

Building Blocks for Neural Networks

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



Computational Units

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

Sound familiar?

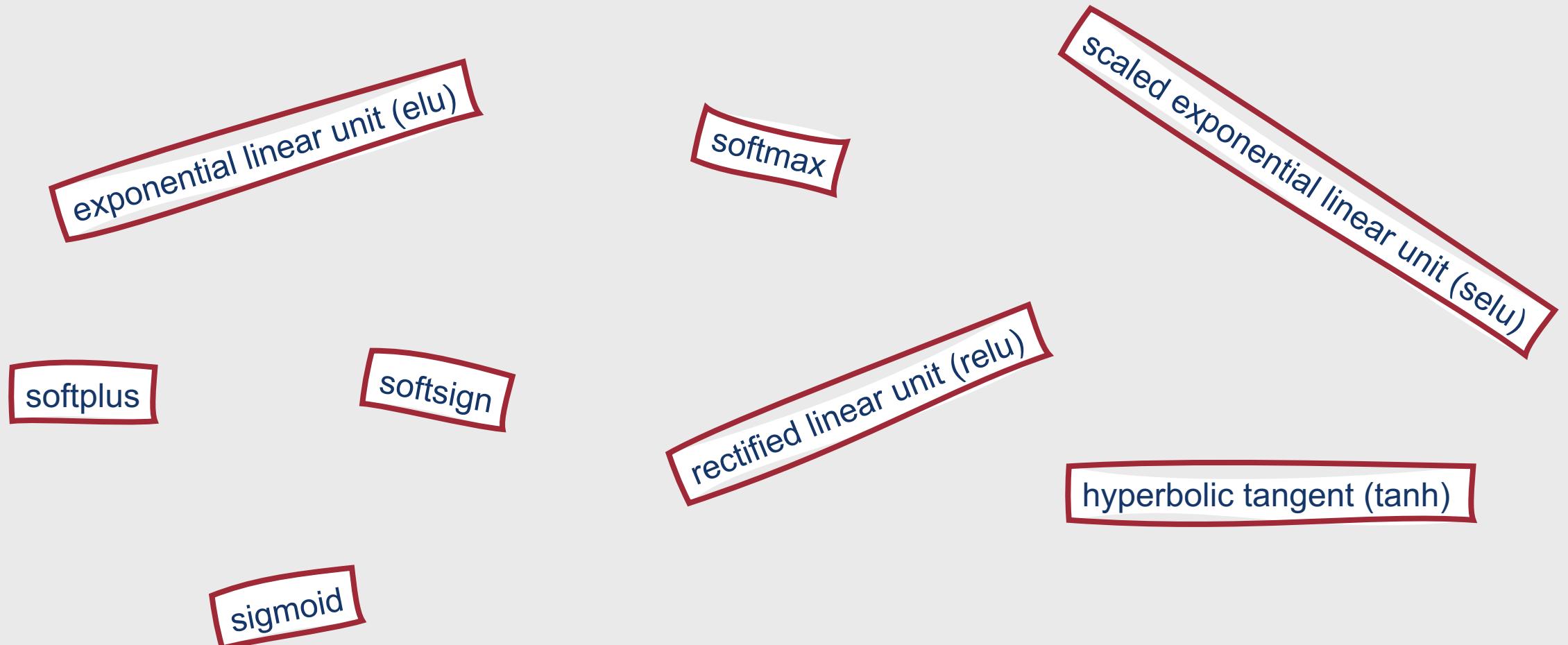
- This is exactly the same sort of weighted sum of inputs that we needed to find with logistic regression!
- Recall that we can also represent the weighted sum z using vector notation:
 - $z = w \cdot x + b$



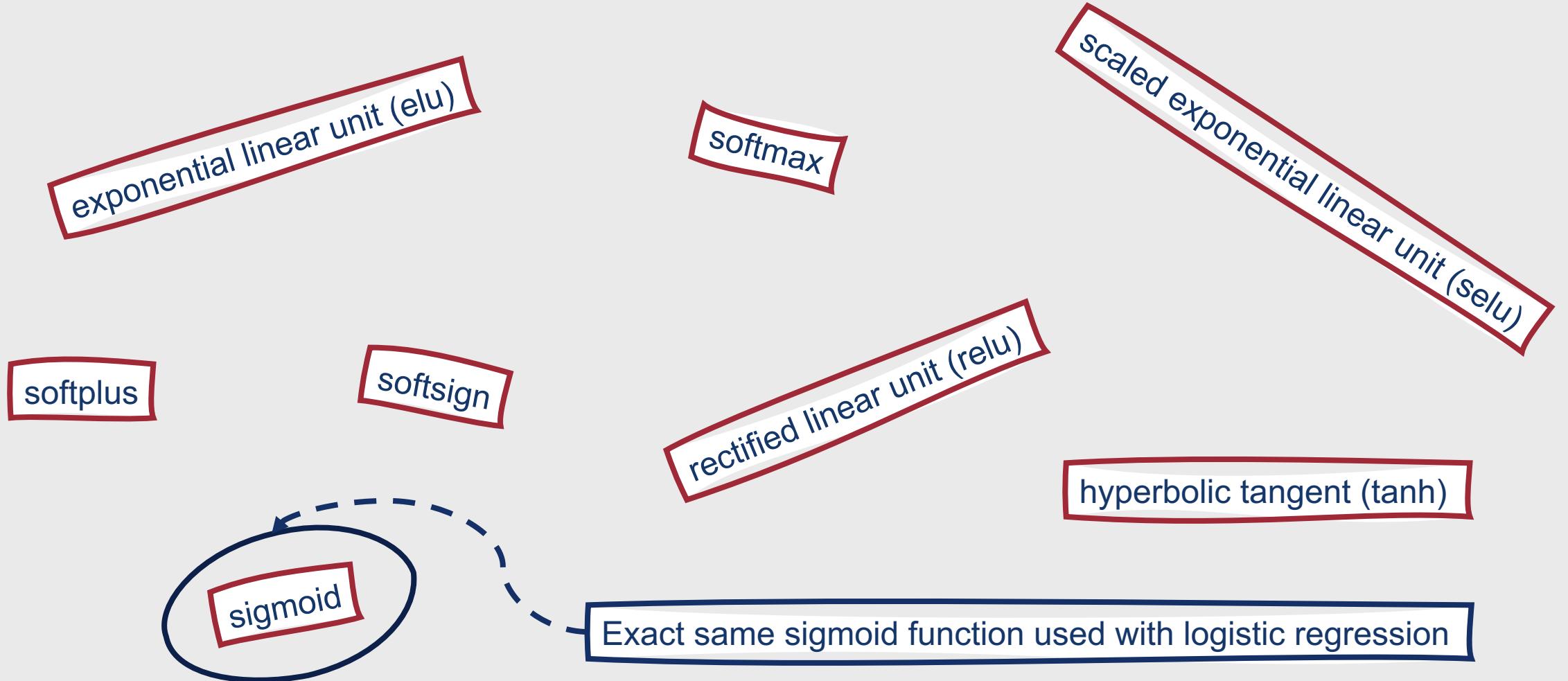
Computational Units

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

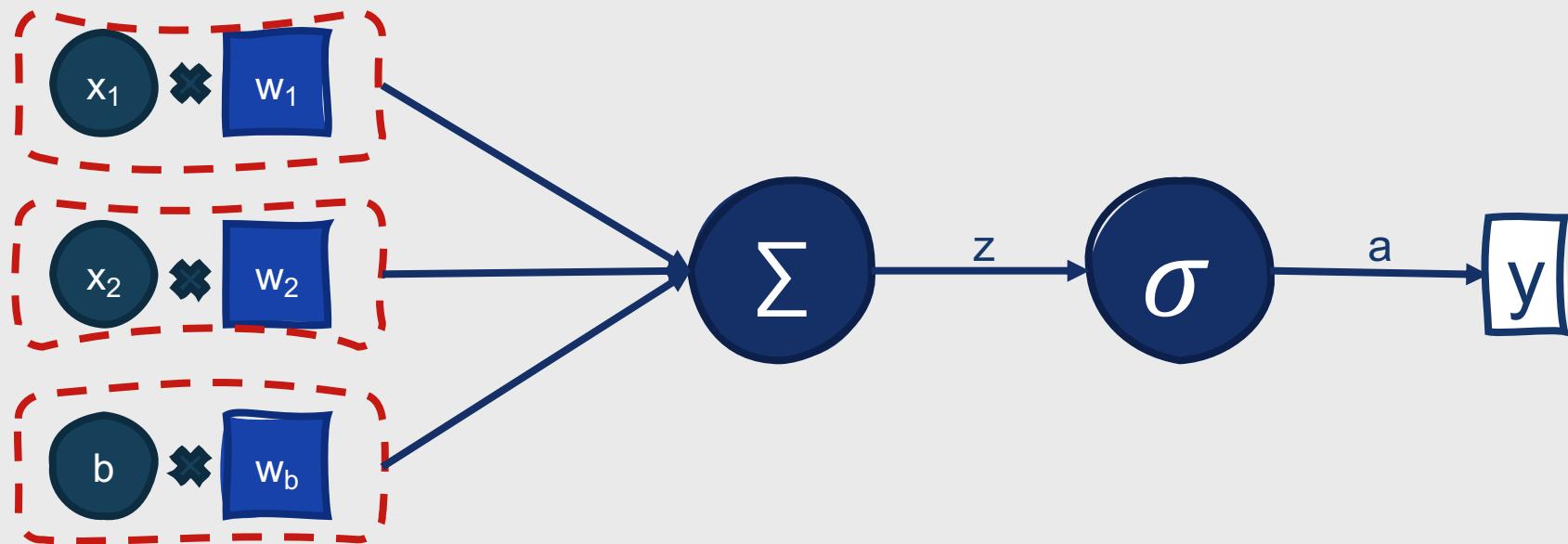
There are many different activation functions!



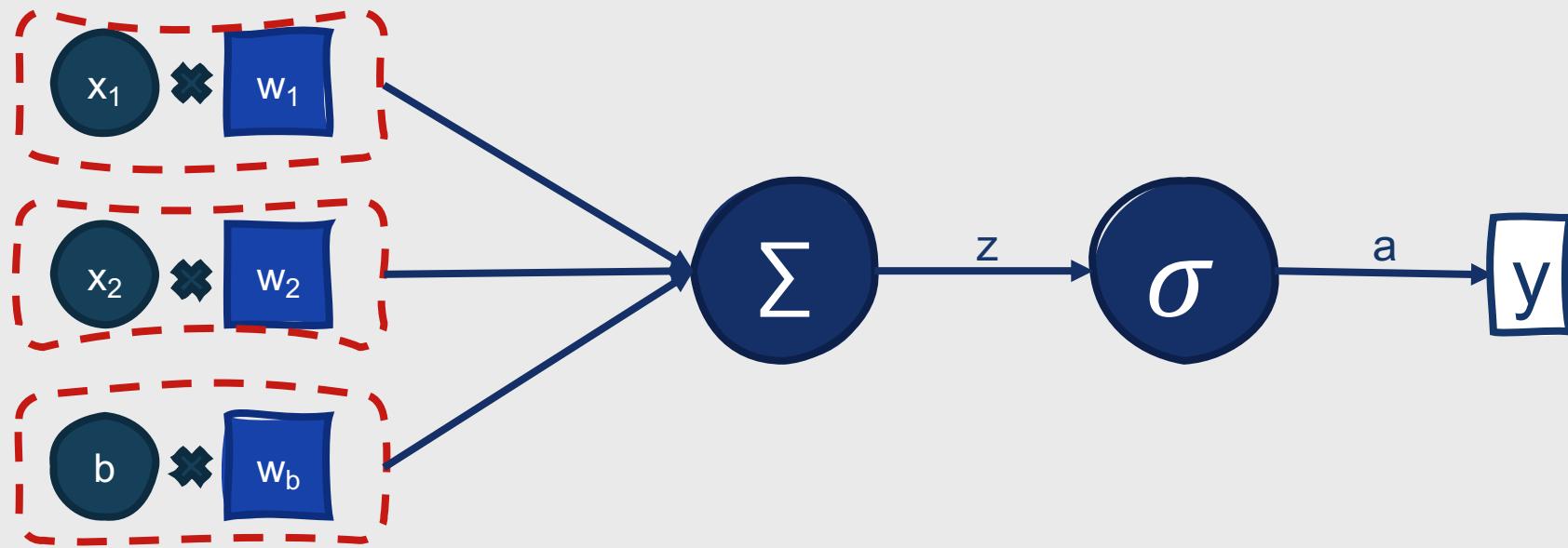
There are many different activation functions!



Computational Unit with Sigmoid Activation



Example: Computational Unit with Sigmoid Activation

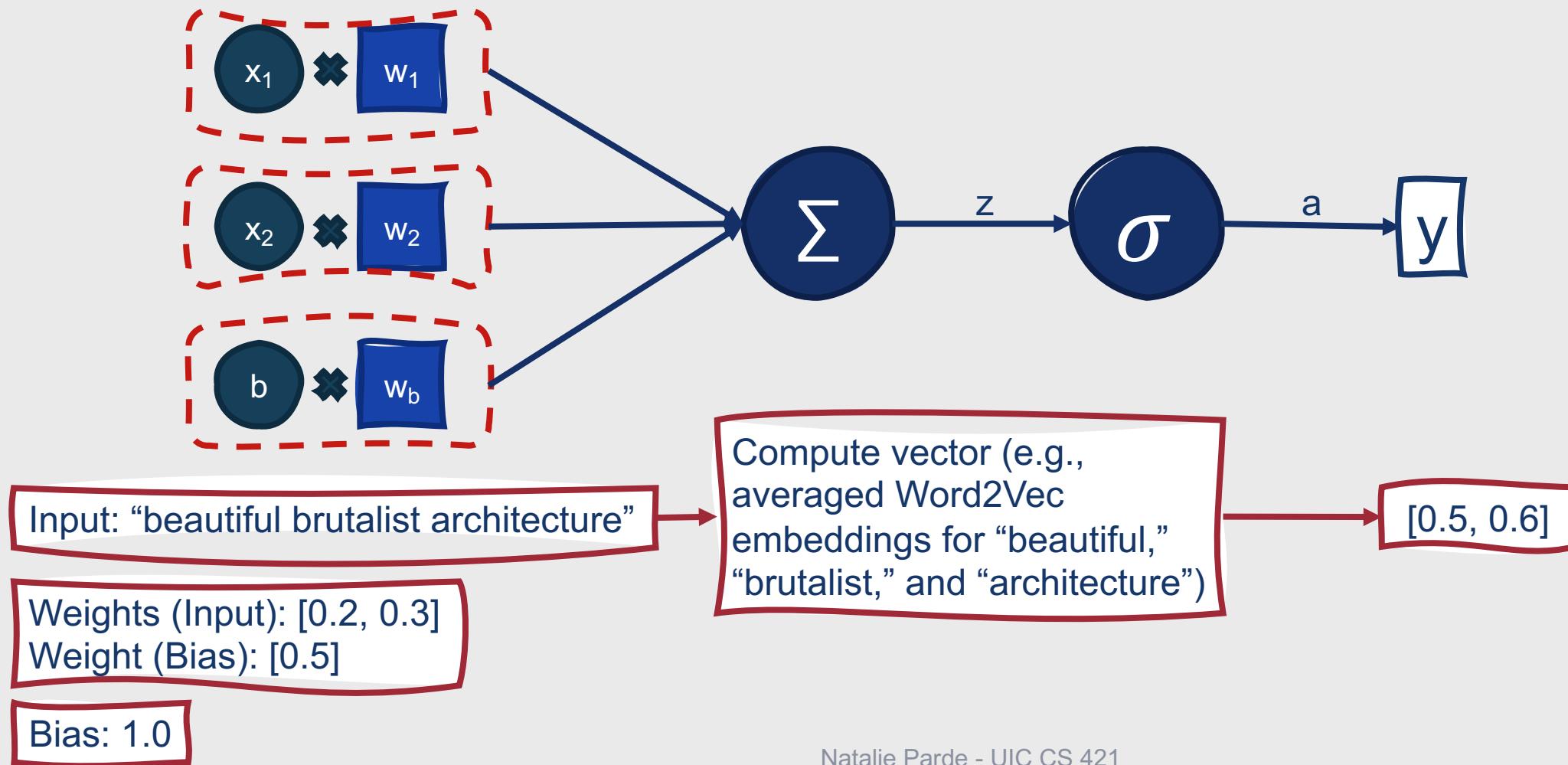


Input: “beautiful brutalist architecture”

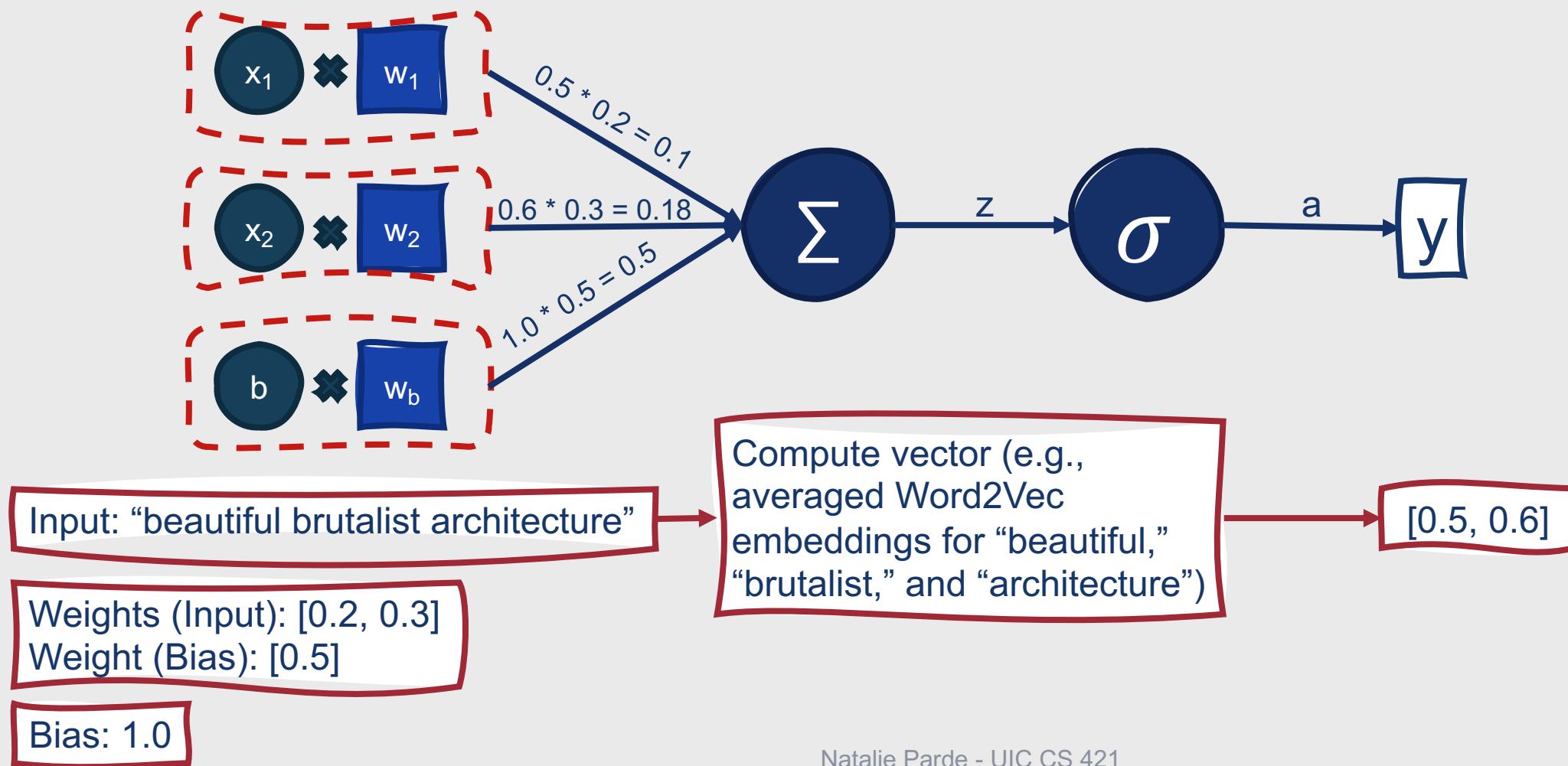
Weights (Input): [0.2, 0.3]
Weight (Bias): [0.5]

Bias: 1.0

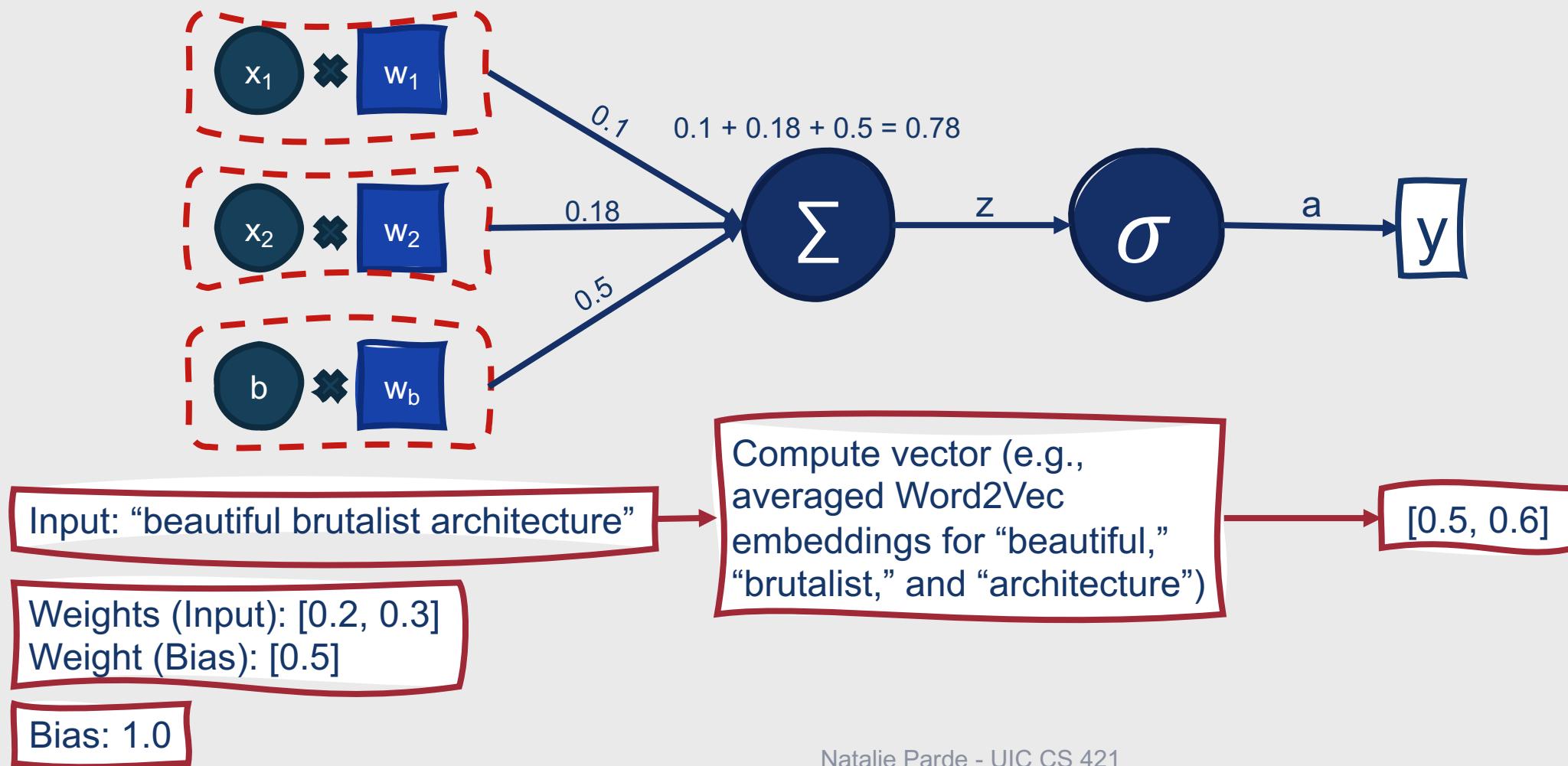
Example: Computational Unit with Sigmoid Activation



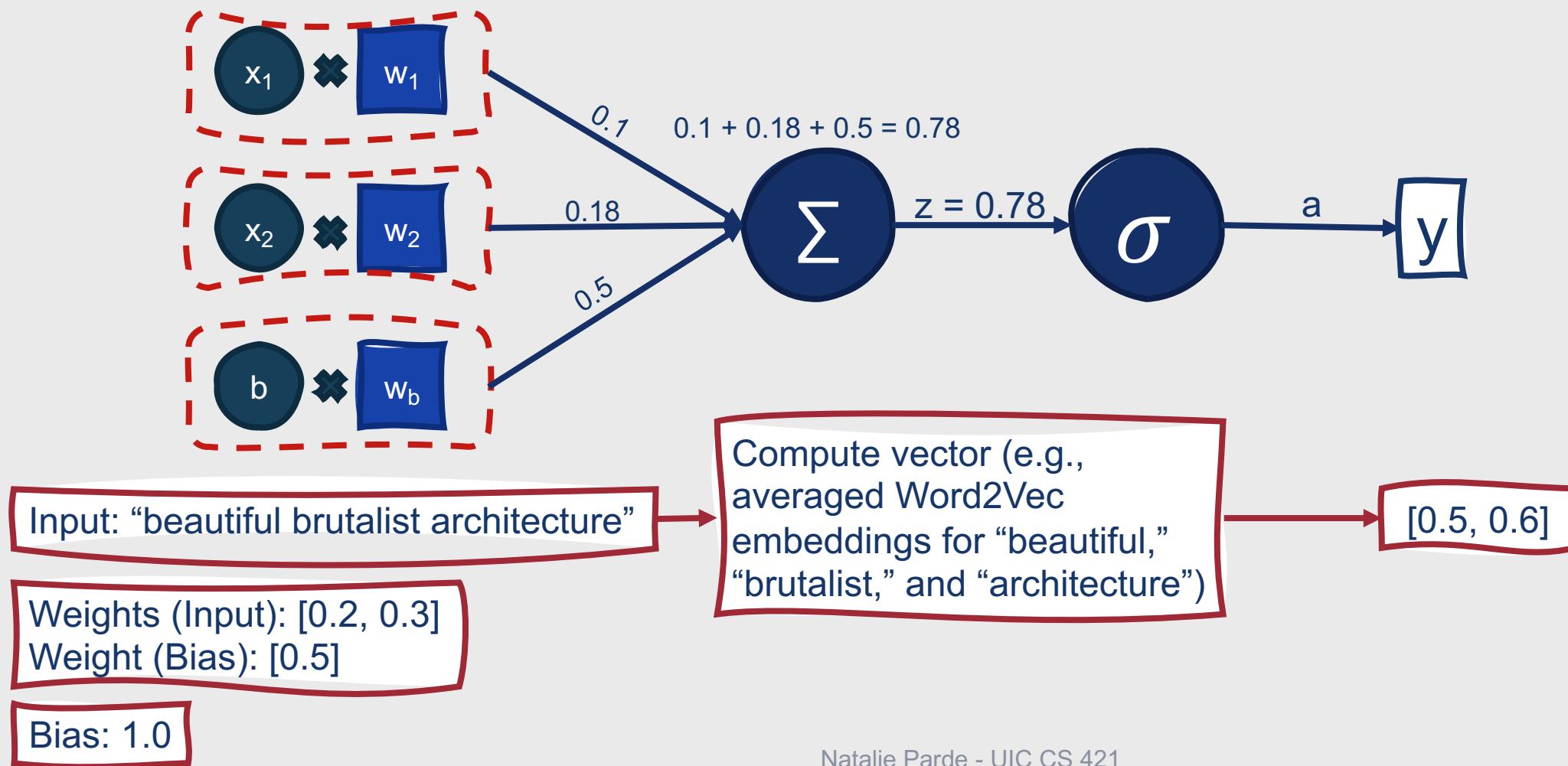
Example: Computational Unit with Sigmoid Activation



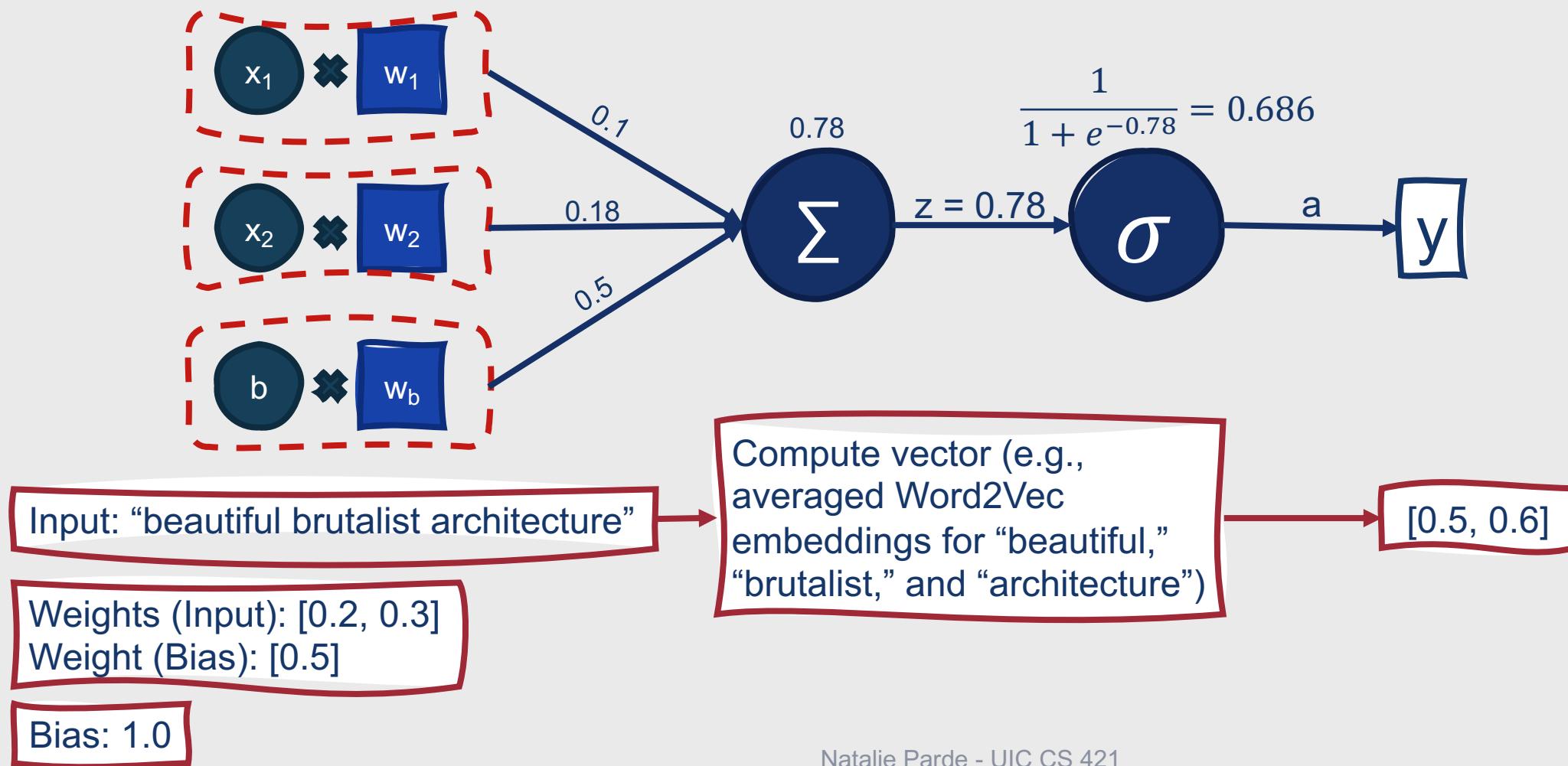
Example: Computational Unit with Sigmoid Activation



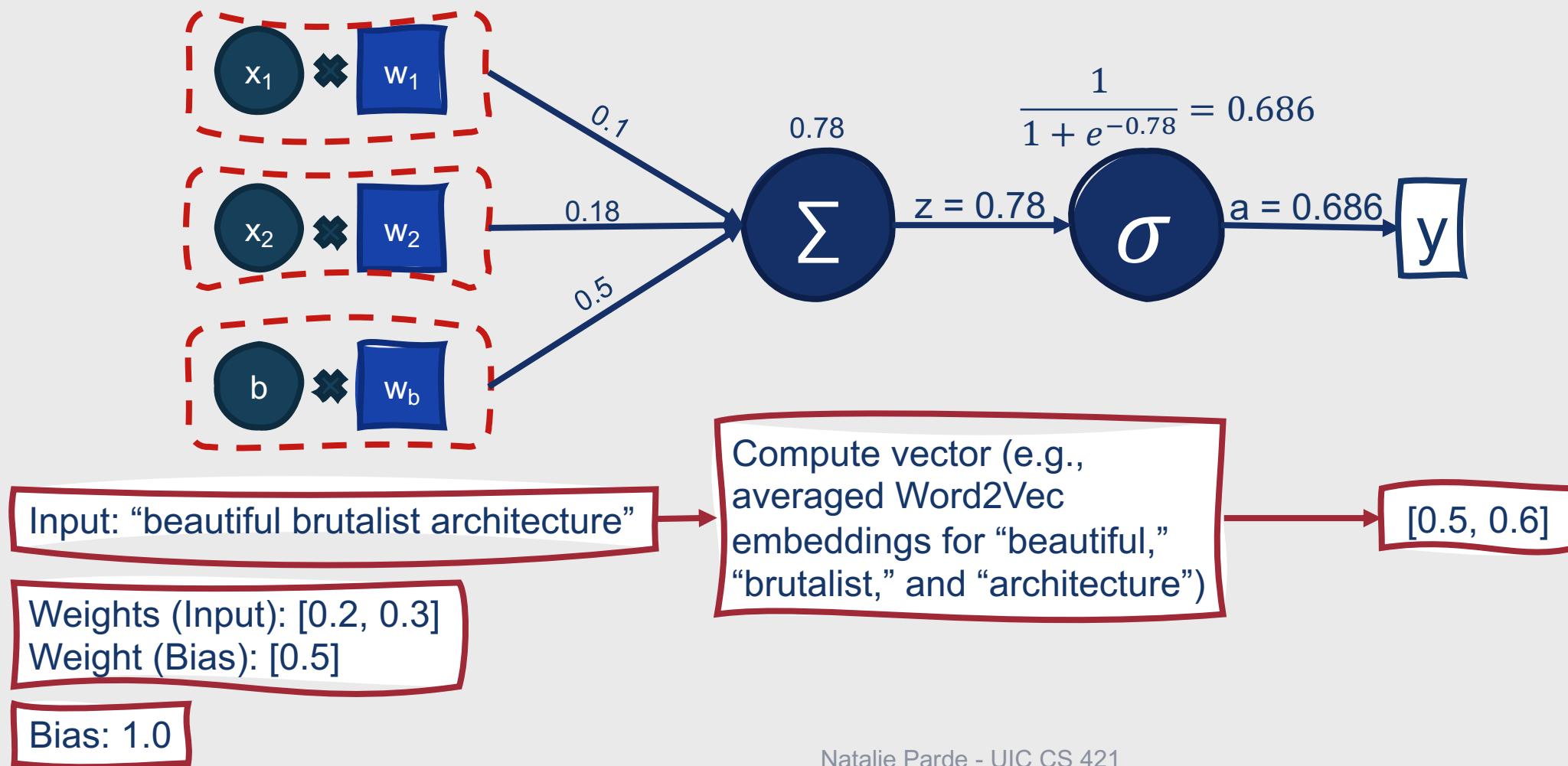
Example: Computational Unit with Sigmoid Activation



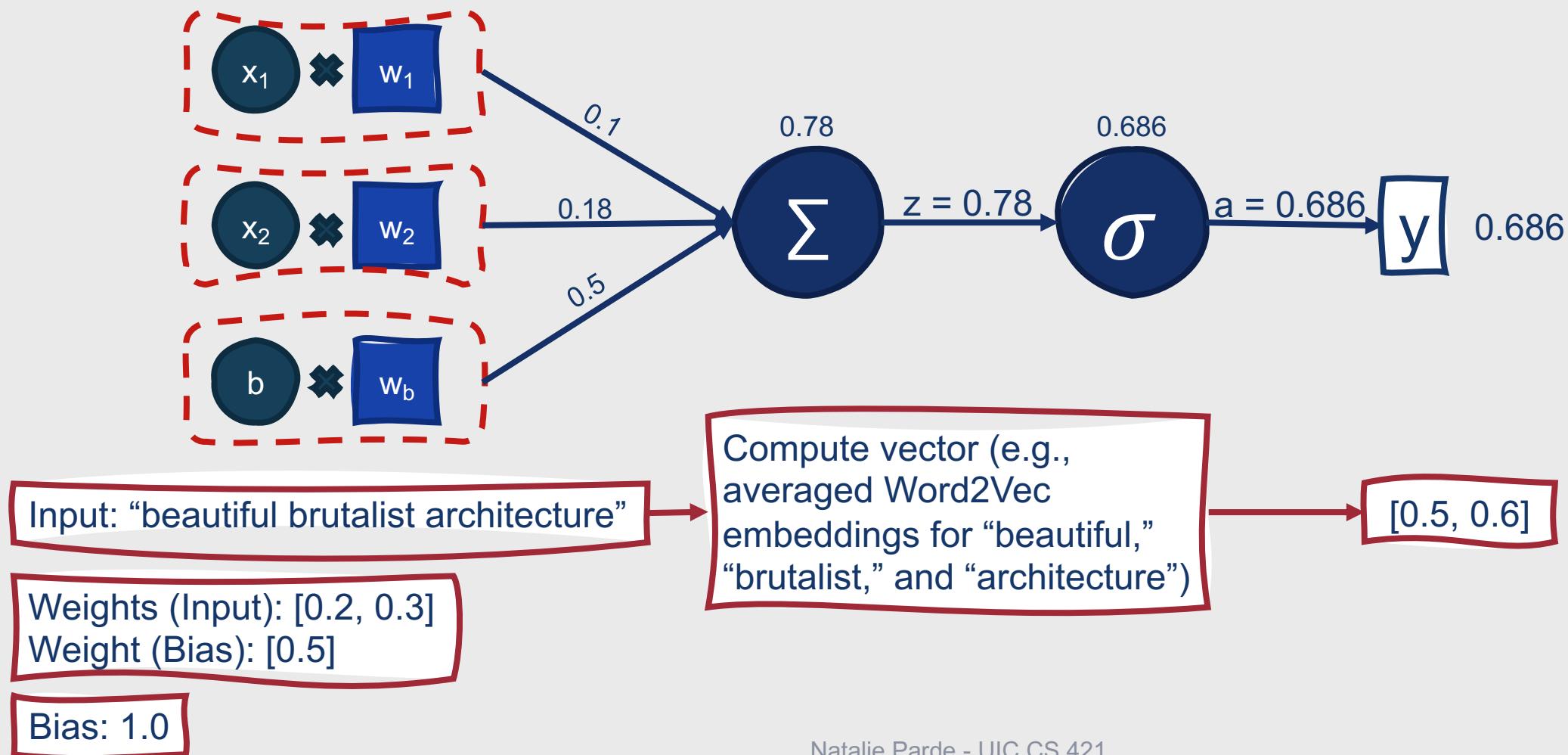
Example: Computational Unit with Sigmoid Activation



Example: Computational Unit with Sigmoid Activation



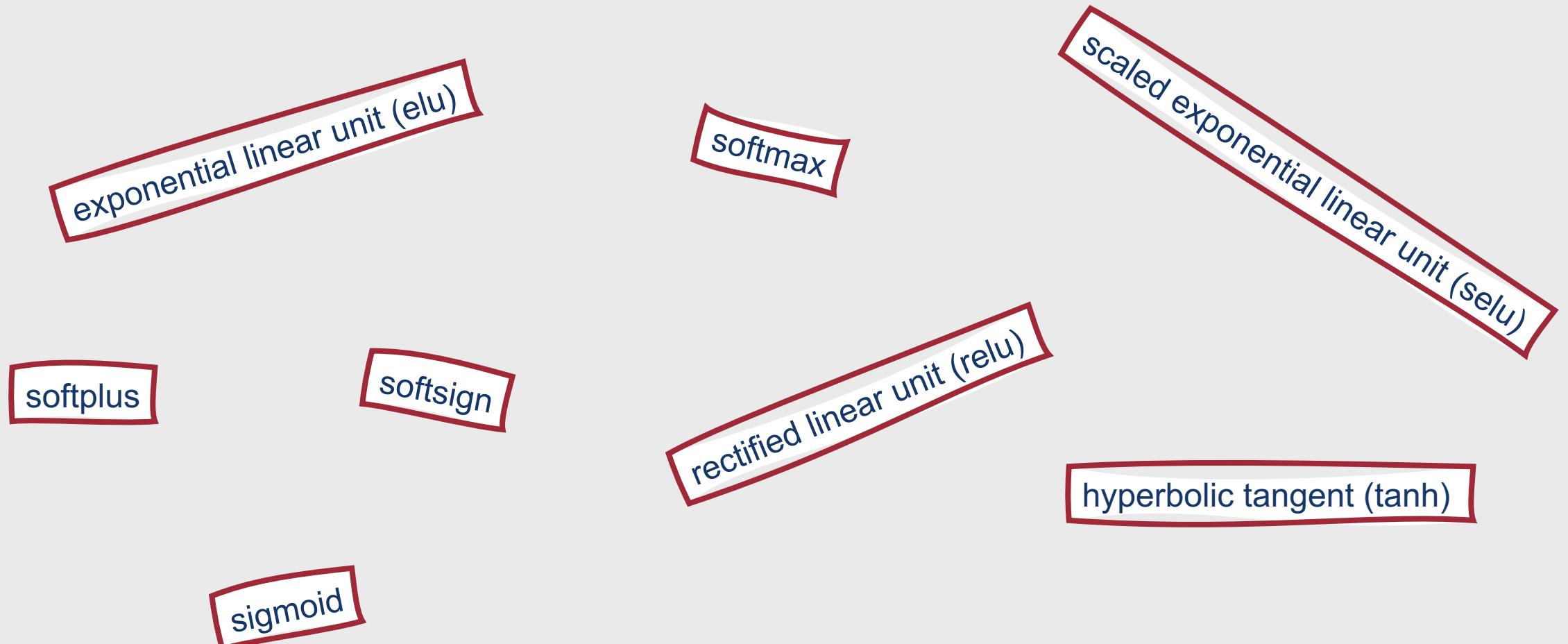
Example: Computational Unit with Sigmoid Activation



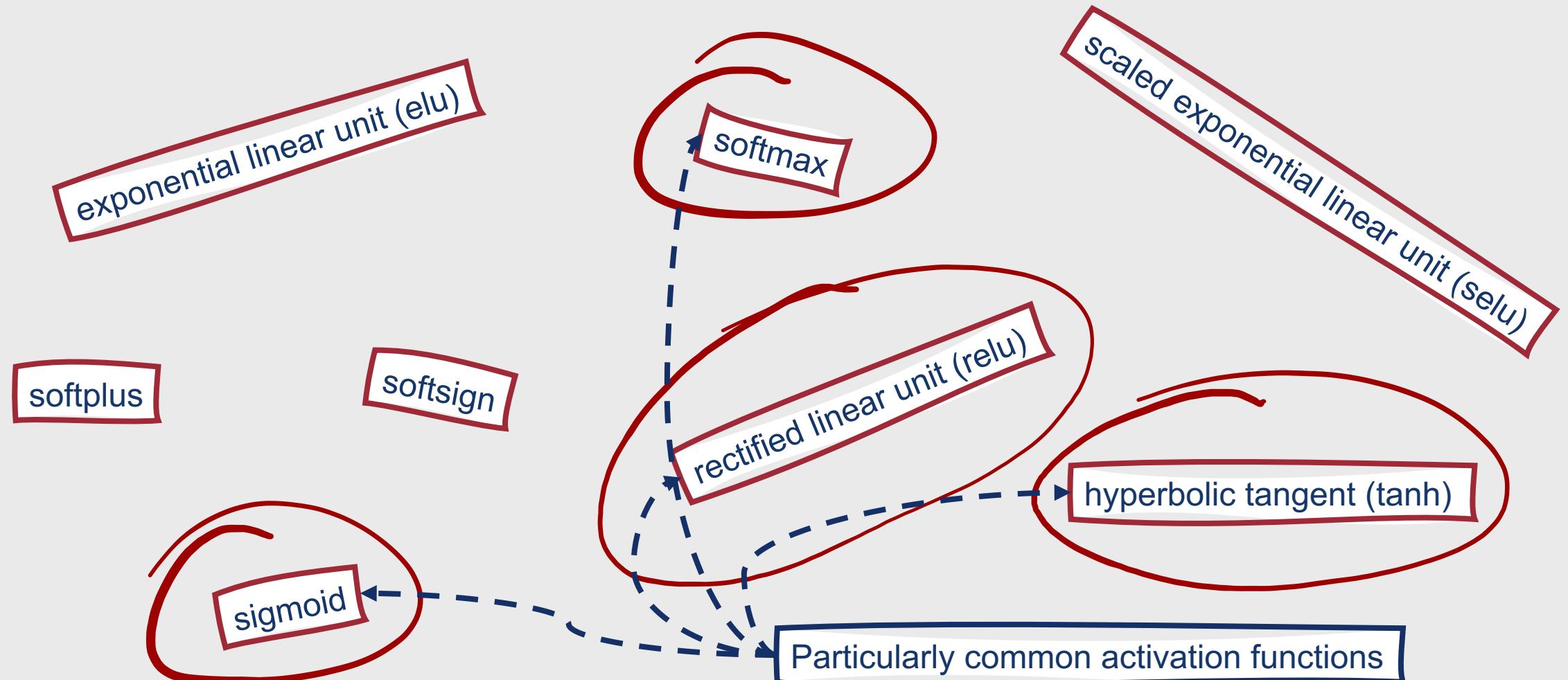
**Although some
neural networks
look like logistic
regression, they
can be
customized in
many ways.**



There are many different activation functions!



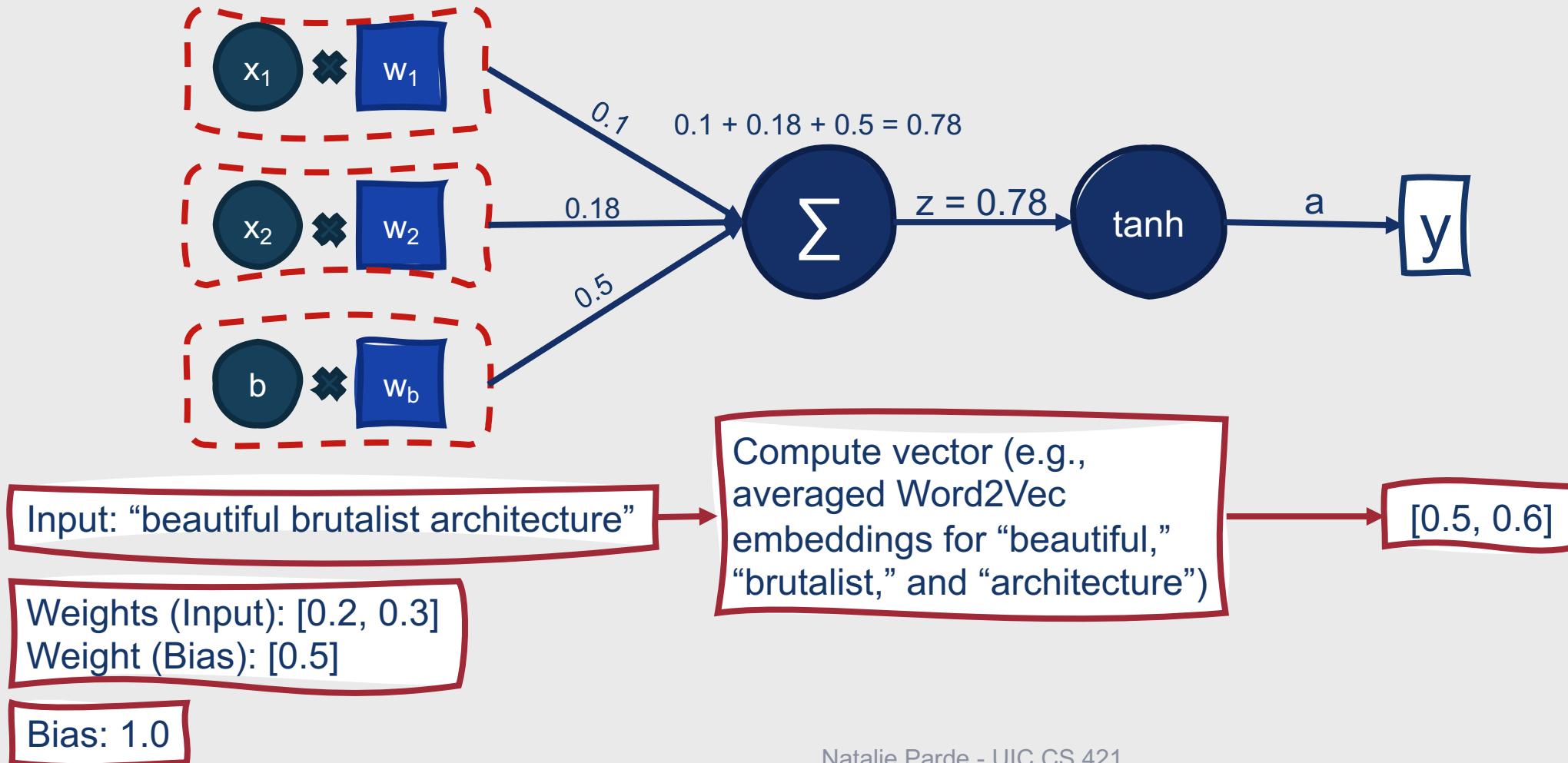
There are many different activation functions!



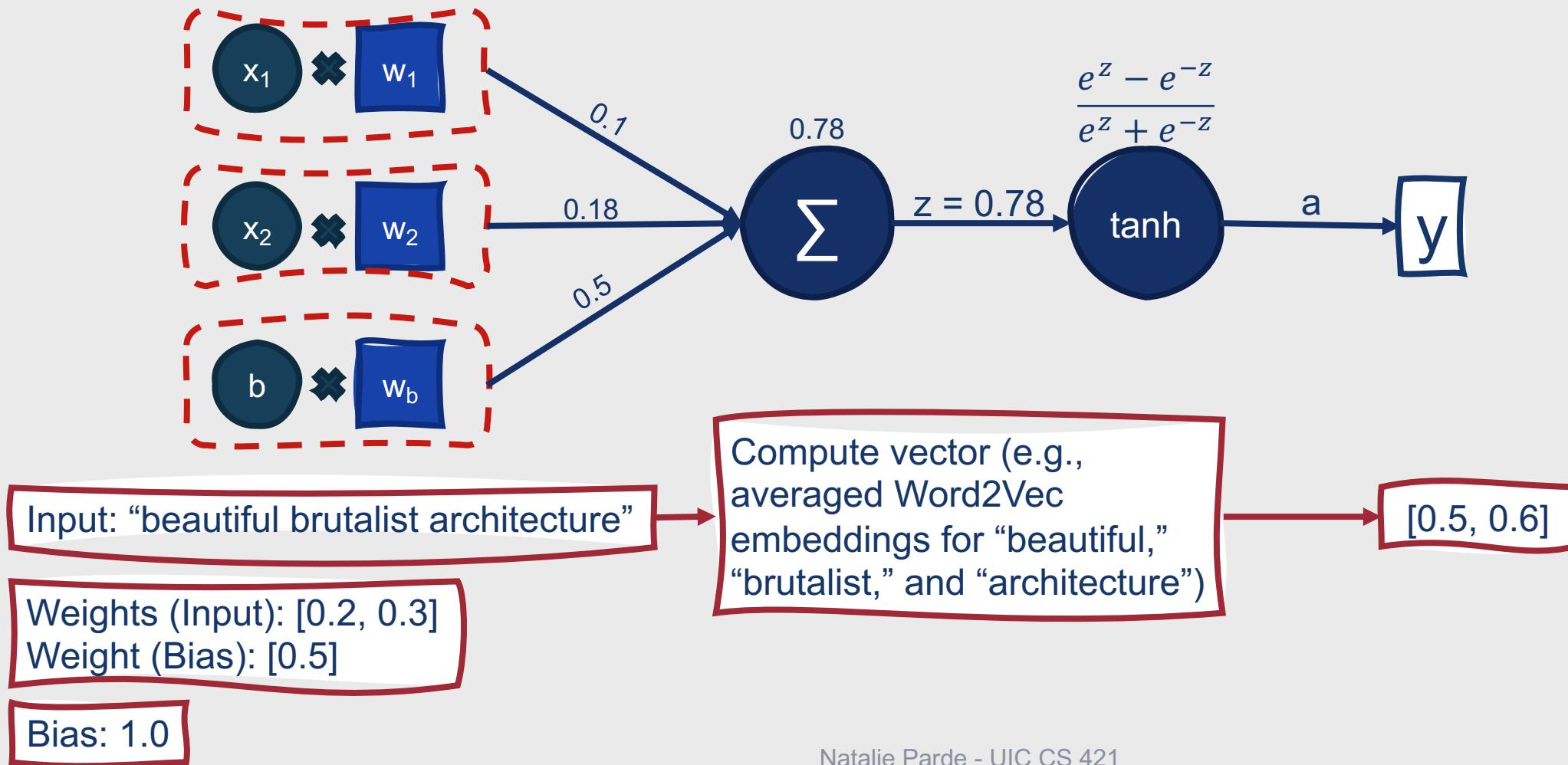
Activation: tanh

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

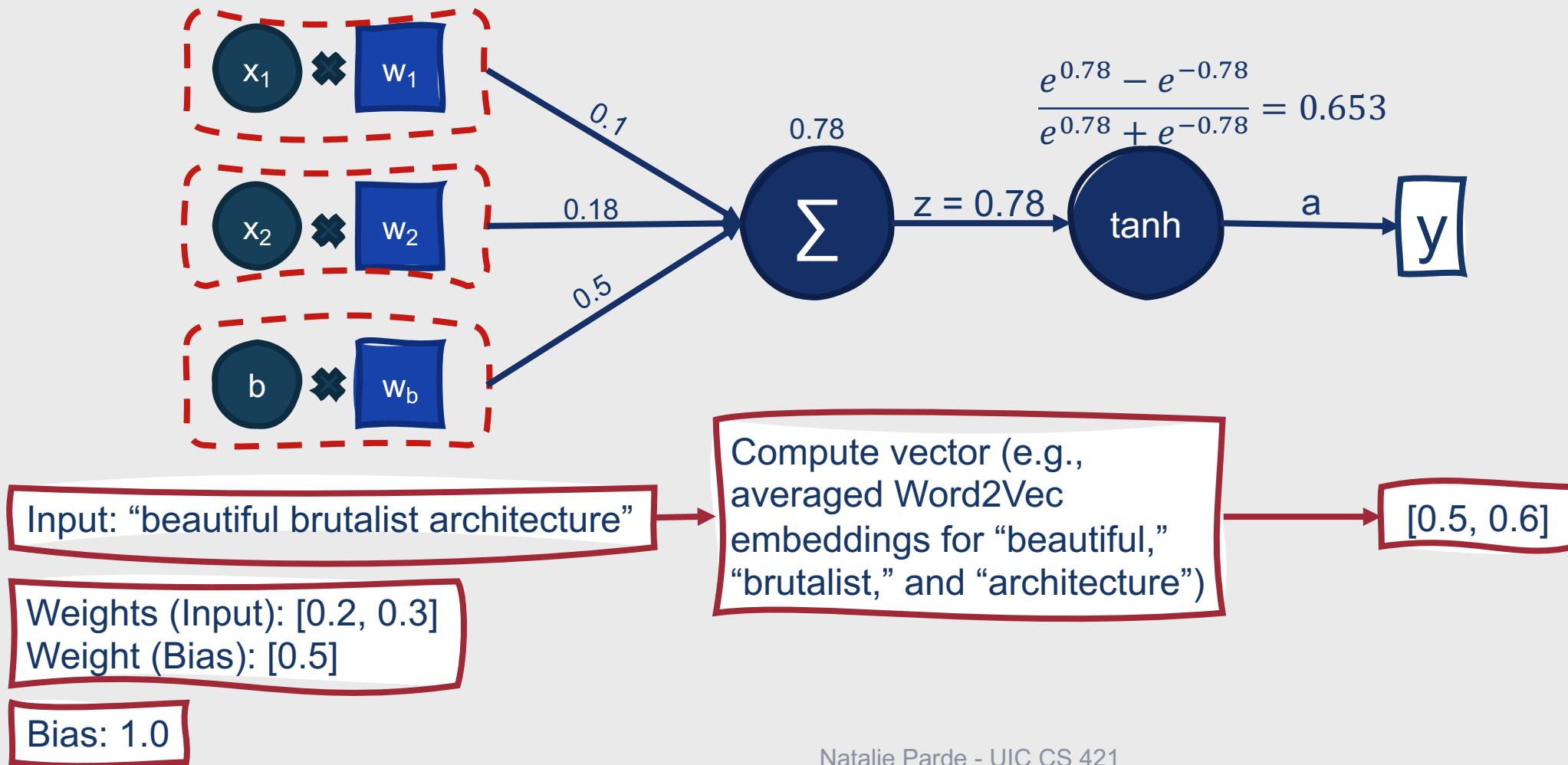
Example: Computational Unit with tanh Activation



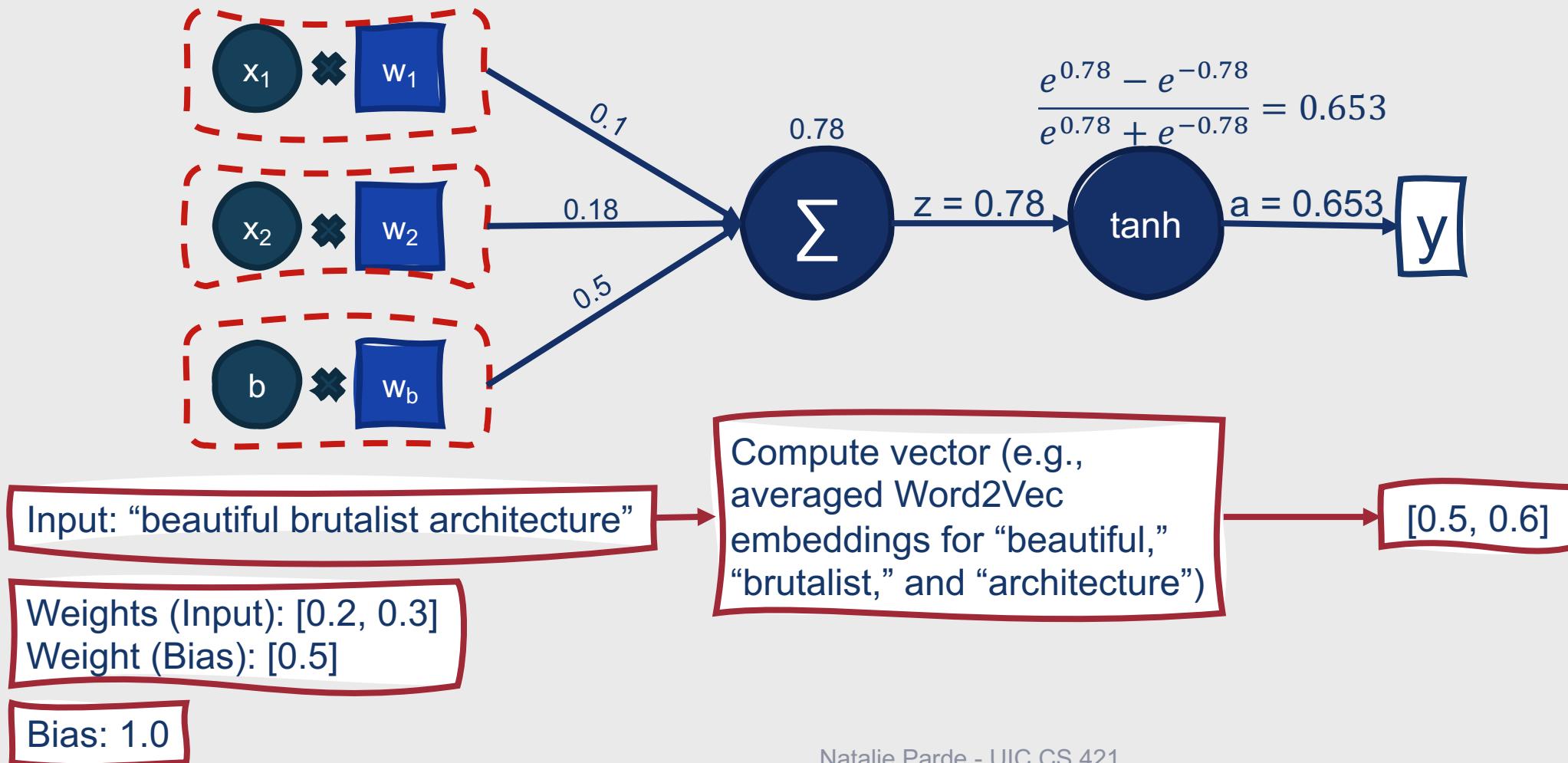
Example: Computational Unit with tanh Activation



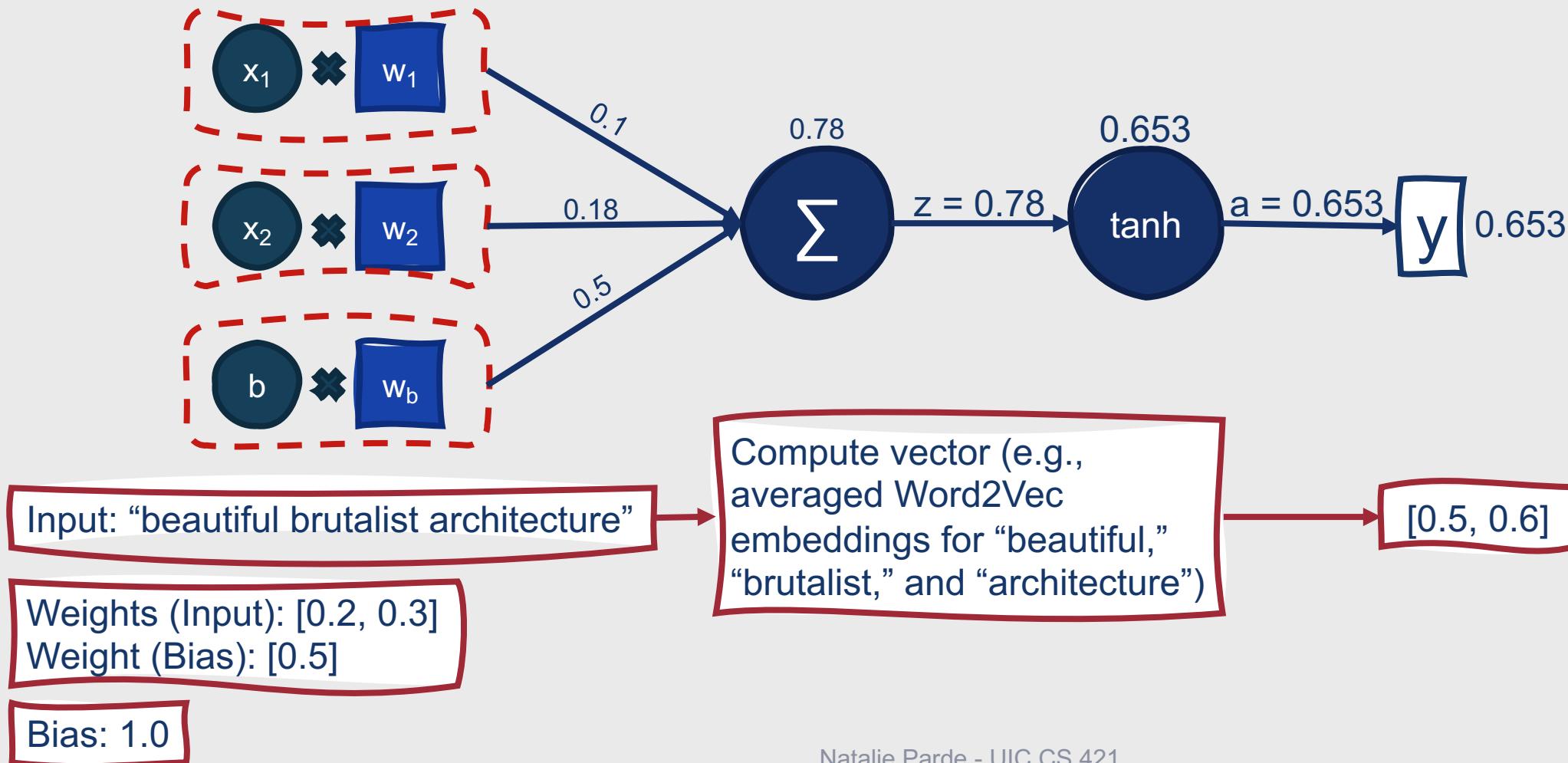
Example: Computational Unit with tanh Activation



Example: Computational Unit with tanh Activation



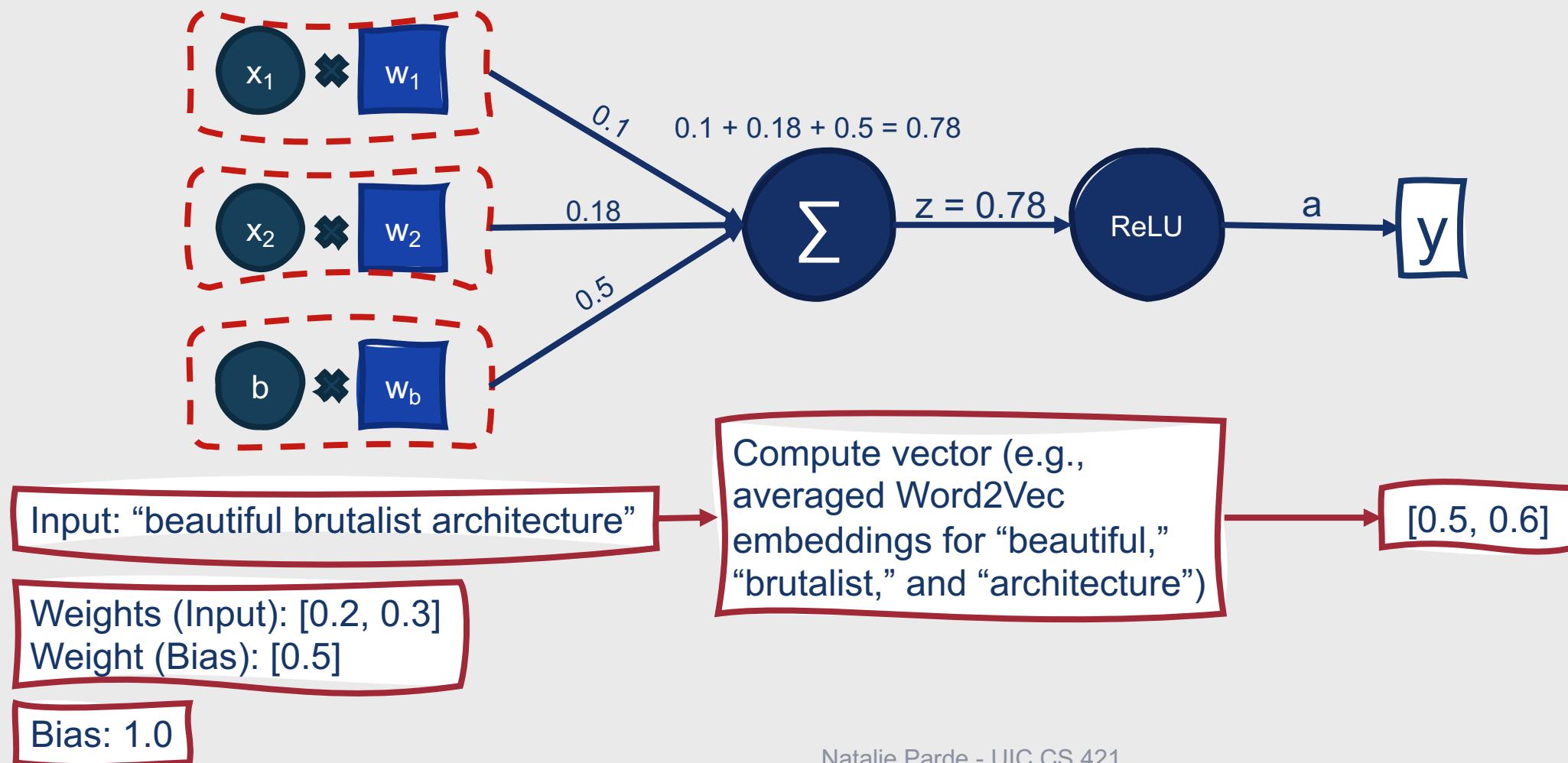
Example: Computational Unit with tanh Activation



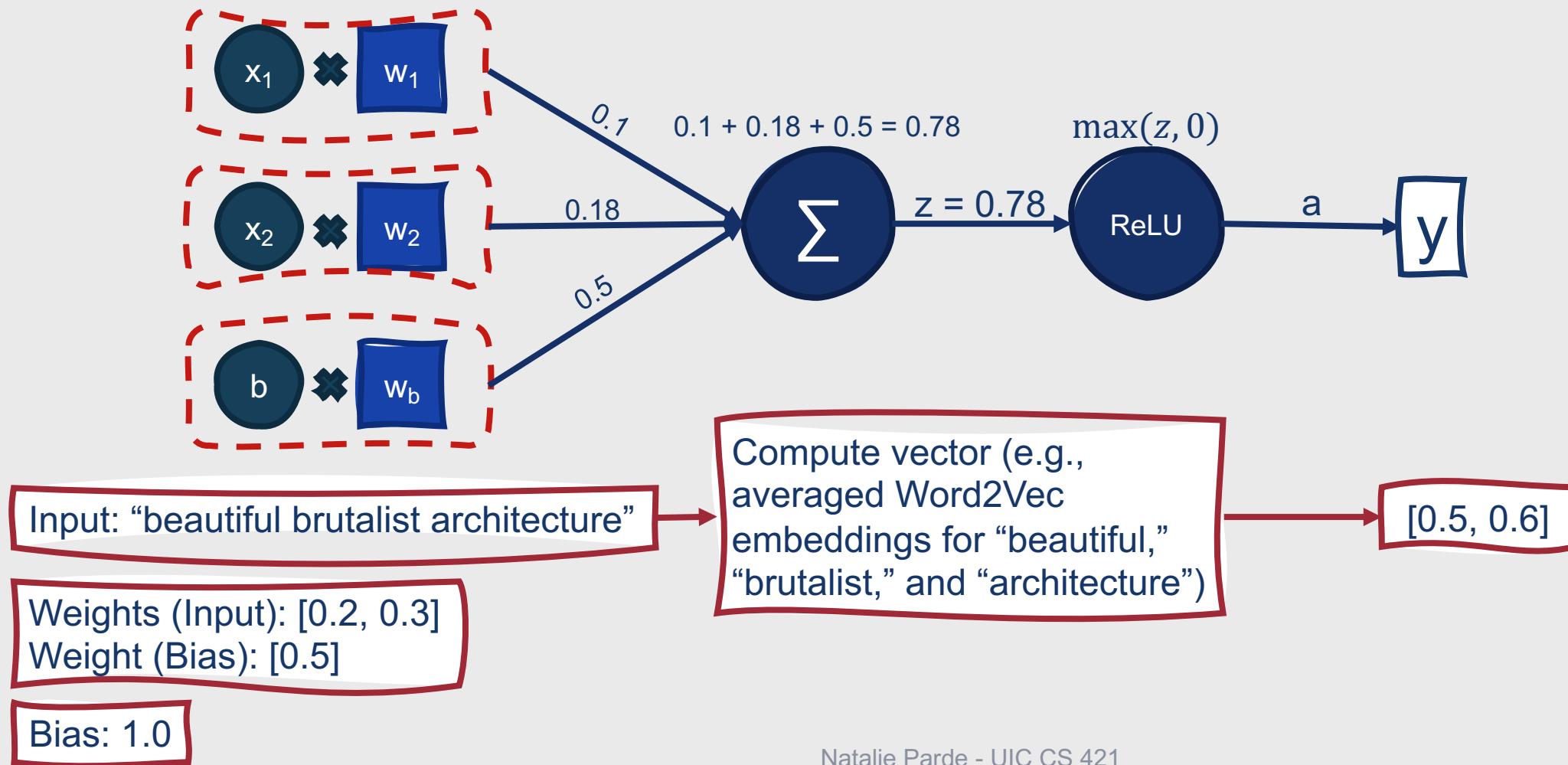
Activation: ReLU

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

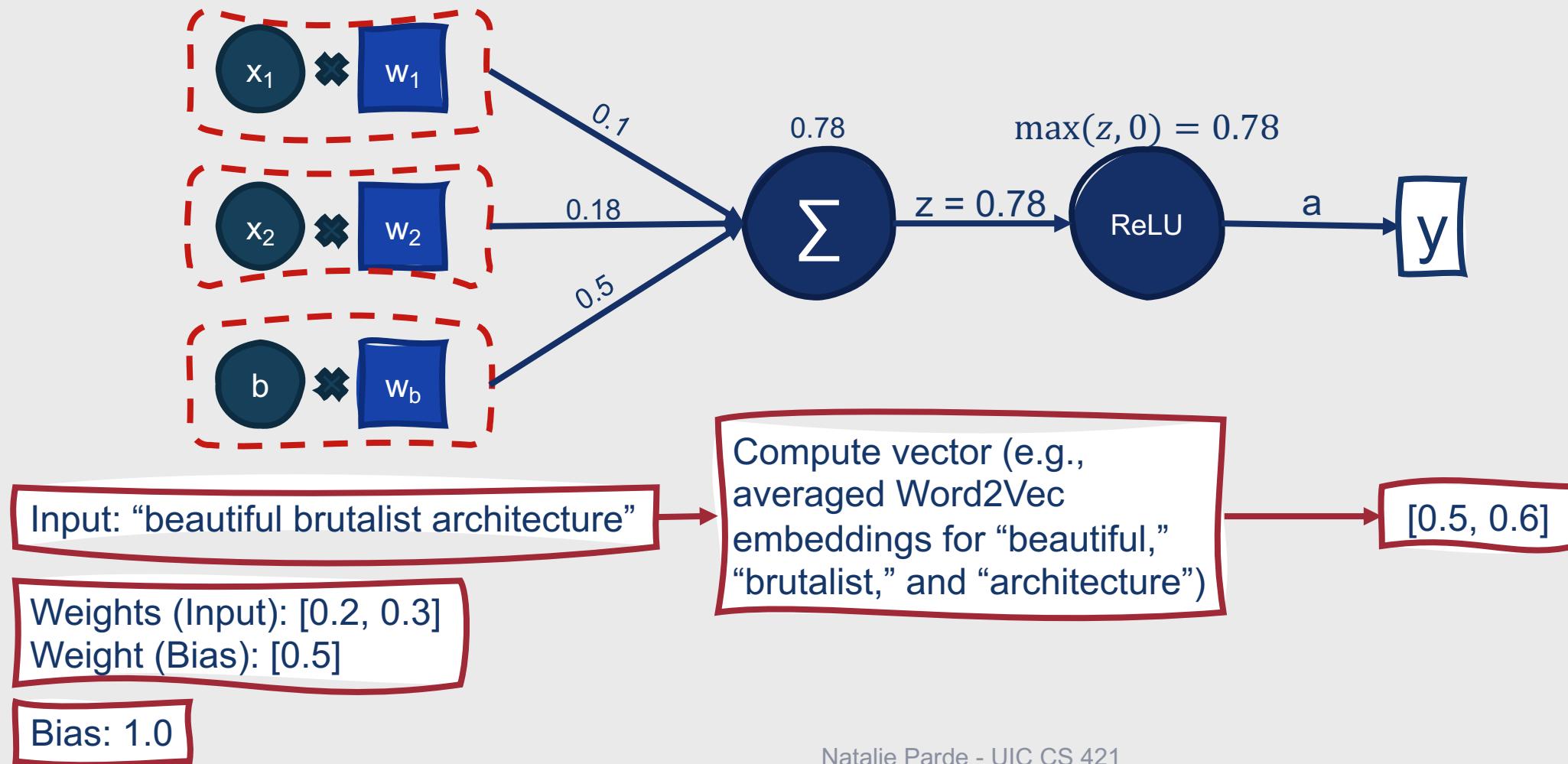
Example: Computational Unit with ReLU Activation



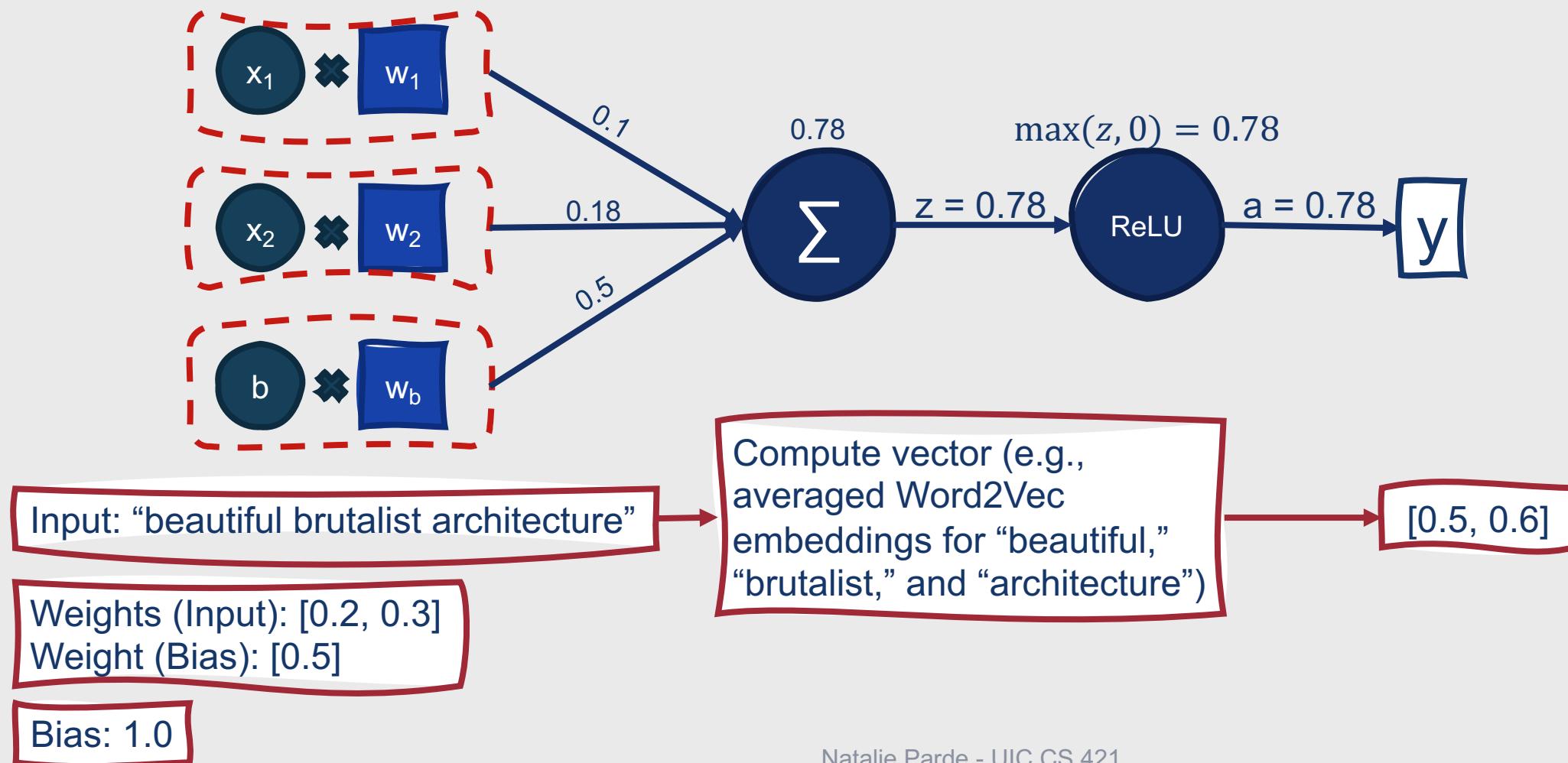
Example: Computational Unit with ReLU Activation



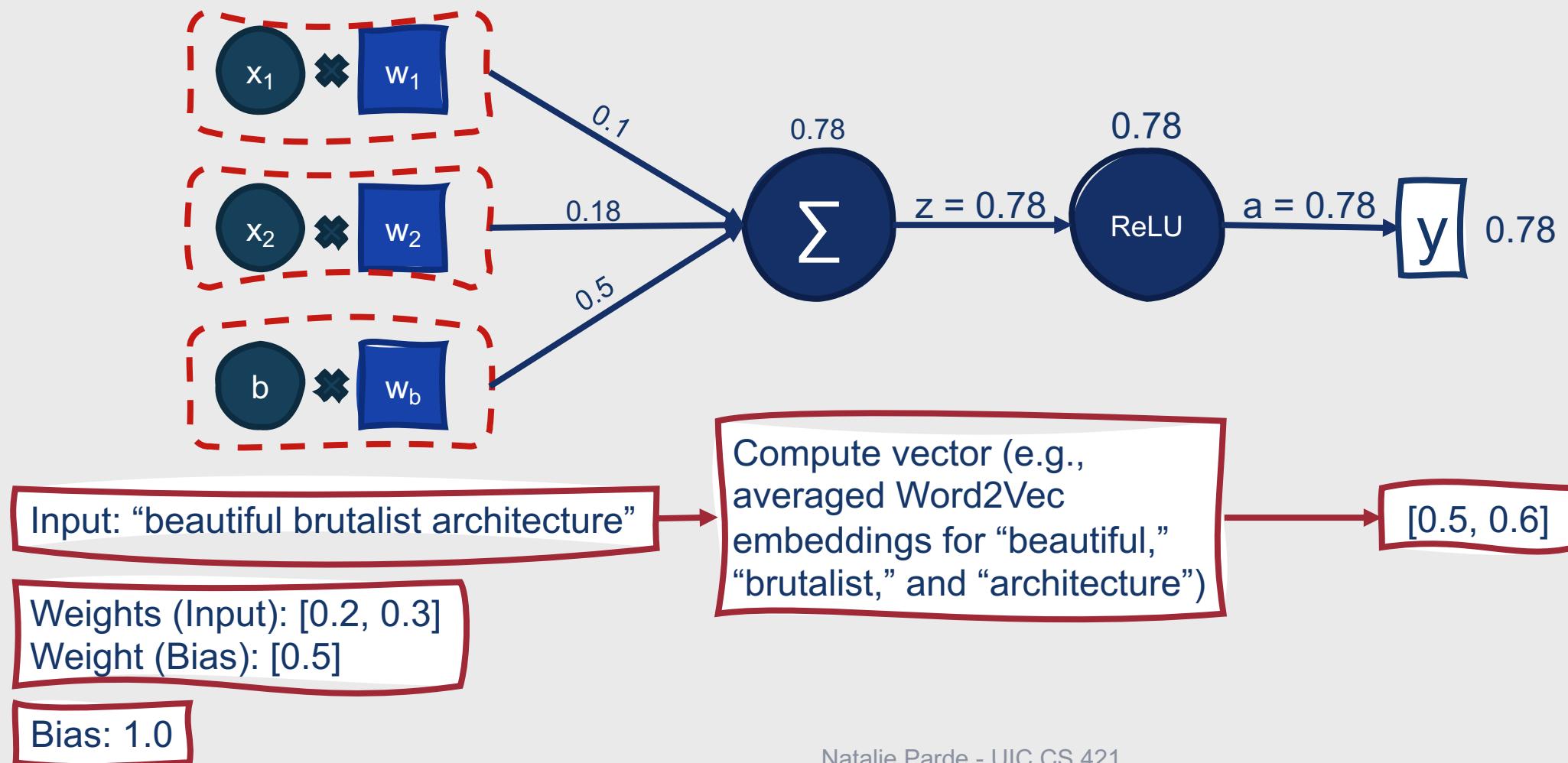
Example: Computational Unit with ReLU Activation



Example: Computational Unit with ReLU Activation



Example: Computational Unit with ReLU Activation



Comparing sigmoid, tanh, and ReLU

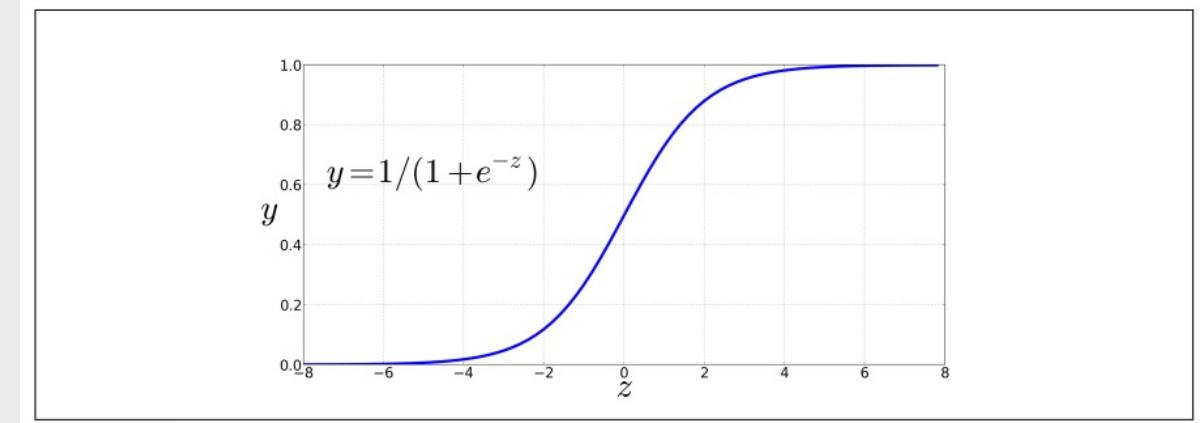


Figure 7.1 The sigmoid function takes a real value and maps it to the range [0, 1]. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

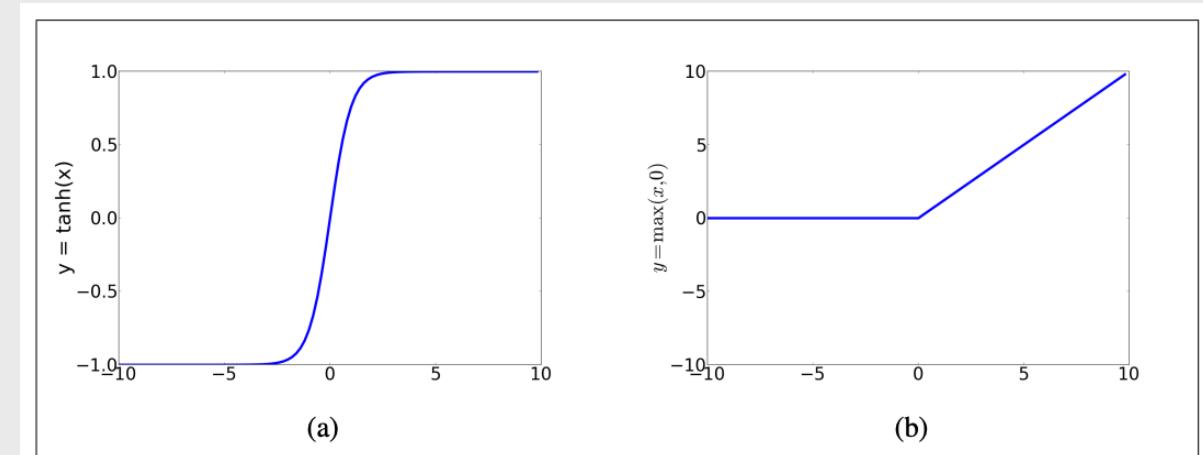


Figure 7.3 The tanh and ReLU activation functions.

Summary: Word Embeddings and Introduction to Neural Networks

- **Word embeddings** can be evaluated through their incorporation in other language tasks
- By altering their training sets and hyperparameters, word embeddings can be used to model syntactic and semantic properties and even the evolution of language over time
- Word embeddings may reflect the same **biases** found in the data used to train them
- Neural networks are classification models that **implicitly learn** sophisticated feature representations during their training process
- **Feedforward neural networks** are the simplest type of neural network, and are comprised of interconnected layers of computing units through which information is passed forward from one layer to the next
- An **activation function** is one of many possible non-linear functions applied to the weighted sum of inputs for a computing unit