

Learning Embeddings

Fall 2026 -- CS594 Vector Databases

A. Asudeh

asudeh@uic.edu

University of Illinois Chicago



Input Feature Vectors

- All ML methods work with the **input feature vectors** $\{x_1, x_2, \dots, x_n\}$ and almost all of them require input features to be **numerical**
- From ML perspective, there are four types of features:
 - Numerical (continues or discrete)
 - Continues: height
 - Discrete: age
 - **Categorical** (ordinal or nominal)
 - Ordinal: level={beginner, intermediate, advanced}
 - Nominal: gender={male, female}, color={red, blue, green}
 - Time series:
 - Average of home sale price over years
 - Text
 - Bag of words

Categorical Features

- There are two ways to encode categorical var:
 - Integer encoding
 - One-hot encoding (and multi-hot encoding)
- Consider the following movie dataset:

Title	provider	IMDB genres	Release year	IMDB rating
Stranger Things	Netflix	drama, fantasy, horror	2016	8.7
Cocomelon	Prime Video	animation, comedy, family	2019	4.7
100 Foot Waves	HBO Max	documentary, sport	2021	8.1
I, Tonya	Hulu	biography, drama, comedy	2017	7.5

Integer Encoding

- Assigns each category value with an integer
 - *provider := [Netflix, Prime Video, HBO Max, Hulu]*, we assign them integers 1, 2, 3 and 4 respectively.

- **Pros:** dense representation
- **Cons:** It implies ordering between different categories:
Netflix < Prime Video < HBO Max < Hulu

Title	provider	IMDB genres	Release year	IMDB rating
Stranger Things	1	drama, fantasy, horror	2016	8.7
Cocomelon	2	animation, comedy, family	2019	4.7
100 Foot Waves	3	documentary, sport	2021	8.1
I, Tonya	4	biography, drama, comedy	2017	7.5

- Makes more sense to use it for ordinal variables:
 - Such as “Education”= {Diploma, Undergrad, Masters, PhD }
 - But still it implies values are equally spaced out

One-hot Encoding

- First do integer encoding, then create a **binary vector** that represents the numerical values
 - Ex: following integer encoding on provider:
Netflix -> 1, Prime Video -> 2, HBO Max -> 3, Hulu -> 4
 - create a binary vector of length 4 for each value:

Netflix	1	0	0	0
Prime Video	0	1	0	0
HBO Max	0	0	1	0
Hulu	0	0	0	1

The integer encoding is the index into the vector

Multi-hot Encoding

- An extension of one-hot encoding when categorical variable can take **multiple values at the same time**
 - Ex: There are 28 distinct IMDB genres 
a movie can take multiple genres, e.g. *stranger things* is drama, fantasy, horror.

IMDB genres

```
[ (1, 'Action'),
  (2, 'Comedy'),
  (3, 'Short'),
  (4, 'Western'),
  (5, 'Drama'),
  (6, 'Horror'),
  (7, 'Music'),
  (8, 'Thriller'),
  (9, 'Animation'),
  (10, 'Adventure'),
  (11, 'Family'),
  (12, 'Fantasy'),
  (13, 'Sport'),
  (14, 'Romance'),
  (15, 'Crime'),
  (16, 'Sci-Fi'),
  (17, 'Biography'),
  (18, 'Musical'),
  (19, 'Mystery'),
  (20, 'History'),
  (21, 'Documentary'),
  (22, 'Film-Noir'),
  (23, 'News'),
  (24, 'Game-Show'),
  (25, 'Reality-TV'),
  (26, 'War'),
  (27, 'Talk-Show'),
  (28, 'Adult')]
```

Stranger things

[illegible]

Cocomelon

[illegible]

100 foot
wave

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

I, Tonya

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

Applying encodings on Movies dataset

	provider								IMDB genres																												
Stranger things	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2016	8.7	
cocomelon	0	1	0	0	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2019	4.7	
100foot waves	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	2021	8.1	
I, Tonya	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	2017	7.5	

- Data dimensions increased from 5 to 35. It will blow up to thousands or a million if we multi-hot encode title!
- One-hot and multi-hot encodings are not practical for features with large value sets.

From encoding to embedding

- **One-hot/multi-hot encodings:**
 - **pros:** simple and robust
 - **cons:** **sparse** and **high dimensional**, **does not capture semantic similarity**
 - In a corpus of documents with one **million** distinct words:
 - **high dimensional:** multi-hot encodings are 1-million dimensional
 - **Sparse:** an average document contains 500 words therefore the multi-hot encodings are > 99.95% sparse
 - **lack of semantic:** **encoding of two words 'good' and 'great' are as different as encoding of 'good' and 'bad'!**
- An **embedding** is a translation of a high-dim vector into a low-dim space. An embedding is a:
 - Dense representation (floating-point value)
 - Low-dimensional vector
 - Captures semantic similarity

Review: SVD as an embedder

- Standard dimensionality Reduction methods
 - Singular value decompositions (SVD)

The diagram illustrates the SVD decomposition of matrix A . Matrix A is represented by a pink rectangle with dimensions m (height) and n (width). It is equal to the product of three matrices: U (a green rectangle with dimensions m and r), Σ (a purple square with dimensions r and r), and V^T (an orange rectangle with dimensions n and r). The dimensions are labeled around each matrix: m and n for A ; m and r for U ; r and r for Σ ; and n and r for V^T .

- **A: Input data matrix:** $m \times n$ matrix (e.g., m documents, n terms)
(r : rank of the matrix **A** – often $r < \min(m,n)$)
- **U: Left singular vectors:** $m \times r$ matrix (m documents, r concepts)
- **Σ : Singular values:** $r \times r$ diagonal matrix (strength of each ‘concept’)
- **V: Right singular vectors:** $n \times r$ matrix (n terms, r concepts)

Review: SVD as an embedder

- U, V : column orthonormal
 - $U^T U = I; V^T V = I$ (I : identity matrix)
 - Columns are orthogonal unit vectors hence they define an r -dimensional subspace
 - U defines an r -dim subspace in $\mathbf{R}^m$
 - V defines an r -dim subspace in $\mathbf{R}^n$
- Projecting A onto V and U produces embeddings:
 - Since $A = U \Sigma V^T$ then $AV = U \Sigma$ are row embeddings
 - Since $A = U \Sigma V^T$ then $U^T A = \Sigma V^T$ are col embeddings

Review: SVD as an embedder

Ex: compute document & word embeddings

Step 1: given a corpus of documents convert it to BOW vectors → get a term-document matrix

- Use term frequencies (tf), or normalize using tf-idf

	data	science	spark	Stanford	learning
document 1	10	15	3	0	10
document 2	0	9	2	8	2
document 3	1	2	20	0	4
document 4	14	11	1	32	2
document 5	5	1	7	12	5
document 6	6	3	5	1	1
document 7	2	3	5	2	7

Review: SVD as an embedder

Step 2: apply SVD on the term-document matrix and pick a value $r \leq \text{rank}(A)$

Here, we set $r = 3$.

10	15	3	0	10
0	9	2	8	2
1	2	20	0	4
14	11	1	32	2
5	1	7	12	5
6	3	5	1	1
2	3	5	2	7

A

~

-0.30	0.41	-0.79
-0.25	0.03	-0.12
-0.14	0.74	0.5
-0.83	-0.40	0.12
-0.33	0.11	0.31
-0.13	0.20	-0.04
-0.14	0.27	-0.05

U

X

42.7	0	0
0	23.8	0
0	0	16.7

Σ

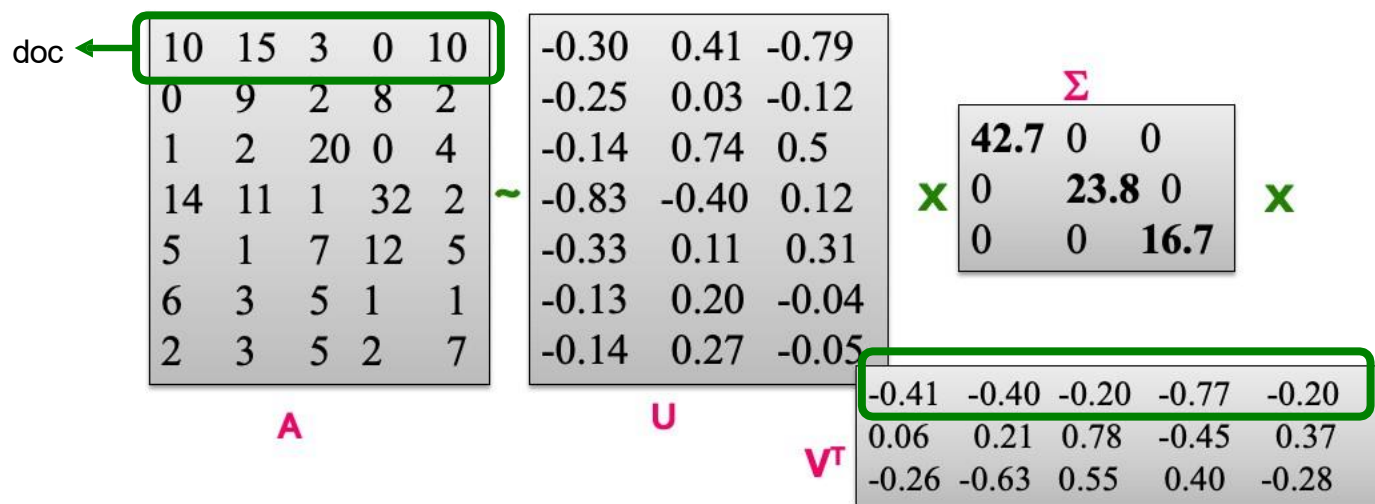
X

-0.41	-0.40	-0.20	-0.77	-0.20
0.06	0.21	0.78	-0.45	0.37
-0.26	-0.63	0.55	0.40	-0.28

V^T

Review: SVD as an embedder

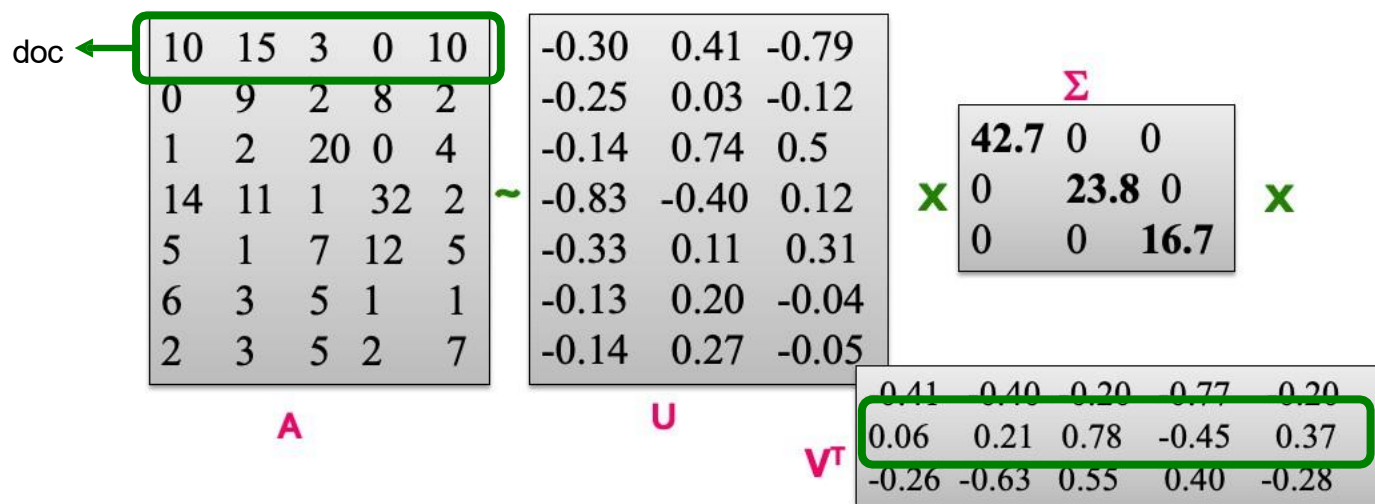
Step 3: compute embedding of documents as
 $\text{emb} = [\langle \text{doc}, v_1 \rangle, \langle \text{doc}, v_2 \rangle, \langle \text{doc}, v_3 \rangle]$



- $\langle \text{doc}, v_1 \rangle = \langle [10, 15, 3, 0, 10], v_1 \rangle = -12.7$

Review: SVD as an embedder

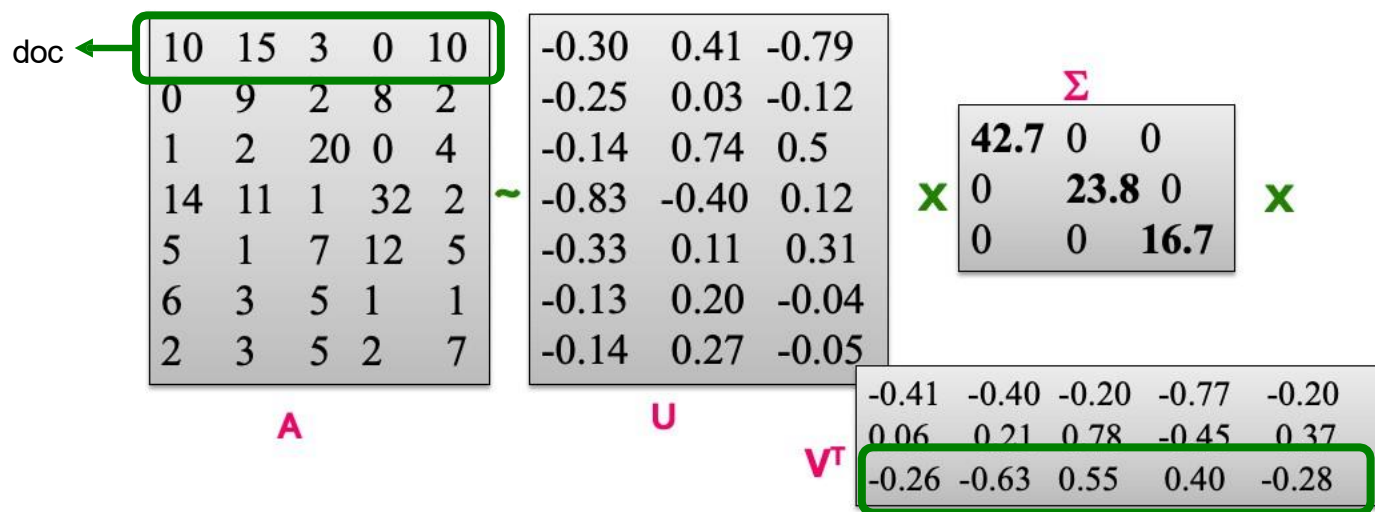
- **Step 3:** compute embedding of documents as
 $\text{emb} = [\langle \text{doc}, v_1 \rangle, \langle \text{doc}, v_2 \rangle, \langle \text{doc}, v_3 \rangle]$



- $\langle \text{doc}, v_1 \rangle = \langle [10, 15, 3, 0, 10], v_1 \rangle = -12.7$
- $\langle \text{doc}, v_2 \rangle = \langle [10, 15, 3, 0, 10], v_2 \rangle = 9.79$

Review: SVD as an embedder

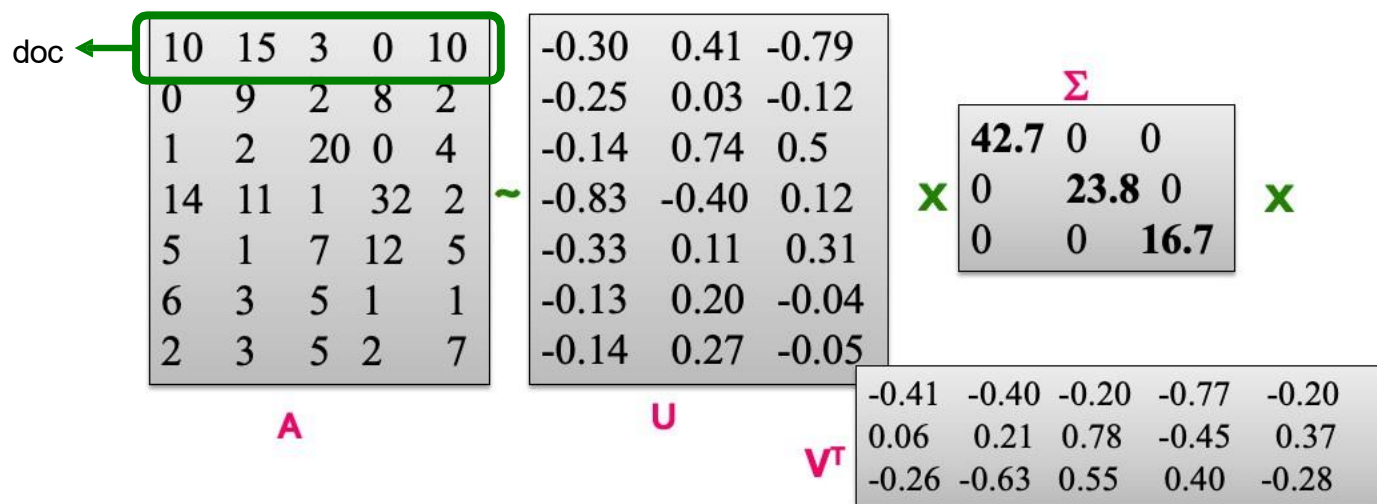
- **Step 3:** compute embedding of documents as
 $\text{emb} = [\langle \text{doc}, v_1 \rangle, \langle \text{doc}, v_2 \rangle, \langle \text{doc}, v_3 \rangle]$



- $\langle \text{doc}, v_1 \rangle = \langle [10, 15, 3, 0, 10], v_1 \rangle = -12.7$
- $\langle \text{doc}, v_2 \rangle = \langle [10, 15, 3, 0, 10], v_2 \rangle = 9.79$
- $\langle \text{doc}, v_3 \rangle = \langle [10, 15, 3, 0, 10], v_3 \rangle = -13.9$

Review: SVD as an embedder

- **Step 3:** compute embedding of documents as
 $\text{emb} = [\langle \text{doc}, v_1 \rangle, \langle \text{doc}, v_2 \rangle, \langle \text{doc}, v_3 \rangle]$



- $\langle \text{doc}, v_1 \rangle = \langle [10, 15, 3, 0, 10], v_1 \rangle = -12.7$
 - $\langle \text{doc}, v_2 \rangle = \langle [10, 15, 3, 0, 10], v_2 \rangle = 9.79$
 - $\langle \text{doc}, v_3 \rangle = \langle [10, 15, 3, 0, 10], v_3 \rangle = -13.9$
- emb1 = [-12.7, 9.79, -13.9]

Review: SVD as an embedder

- SVD is impractical on real-world datasets
 - There are 0.5 billion wiki pages, and 4 billion words.
 - SVD is computationally prohibitive, as it requires to load all data in memory
 - SVD is a linear embedder
 - Not utilizing sparsity
 - Orthonormality constraint is an overkill

From SVD to Neural Networks

- State of the art embedders are among **neural networks**
- Can we use neural networks to create non-linear embedding?

Learning Embeddings using Neural Networks

Agenda

- We will work with three examples:
 1. Word embeddings
 - Word embeddings produced by **Word2Vec** model
 - Converts one-hot encoding to dense embedding
 - Unsupervised mode
 2. Video recommendation
 - Converts one-hot encoding to embedding
 - Item-item collaborative filtering
 3. Autoencoders:
 - Learn representation by reconstructing input
 - Unsupervised mode

Task Independent Embedding

- Word Embedding
 - Many techniques: Word2Vec, Glove, BERT, fastText
 - **Today's lecture: Word2Vec**
- Word2Vec was Developed at Google in 2013([paper](#))
- Word2Vec is a **statistical method**
 - Efficient for learning word embedding
 - Unsupervised and task independent.

Word2Vec

- Word2Vec comes in two architectures:
 - Continuous bag of words (CBOW)
 - Skip Gram(We will discuss **skip-gram** model today)
- The two methods are very similar, both use a shallow neural network (**only 1 hidden layer**) to learn word representations.
- The key idea of **word2Vec** is that words with **similar context** have similar meanings.
 - It learns embedding based on **the usage of words**.

Word2Vec: target and context

The key idea: The more often a word appears in the **context** of certain other words, the closer they are in meaning.

This is how we define context:

Given a document, set the **window size=N**. Window size is a hyperparameter.

For any given word (call it “target word”), **N** words to its left & **N** words to its right are the “context words”.

- Document “**I read sci-fi books**”, and window size = 2
- Target = “**I**” → context words = “**read**”, “**sci-fi**” .
- Target = “**read**” → context words = “**I**”, “**sci-fi**”, “**books**”

Given a document, we can slide the window from left to right and find all pairs of (*target*, *context*) words

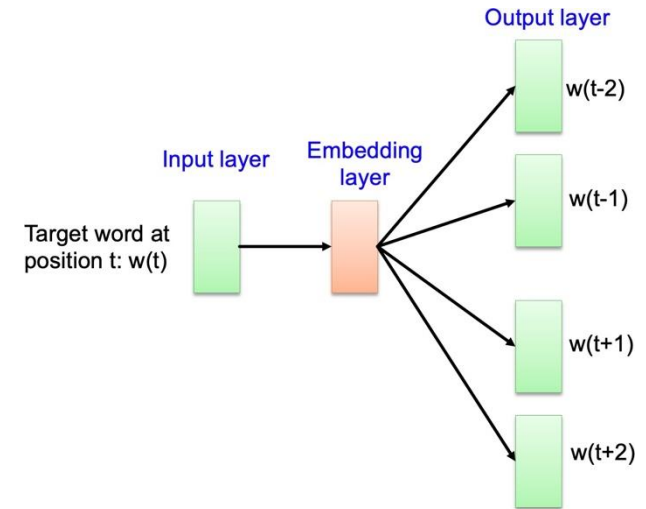
Word2Vec: target and context

Document: "I read sci-fi books and drink orange juice". Let window size = 2
The **highlighted** word is the **target** word. Other words in the box are context words.

[illegible]

Word2Vec: architecture

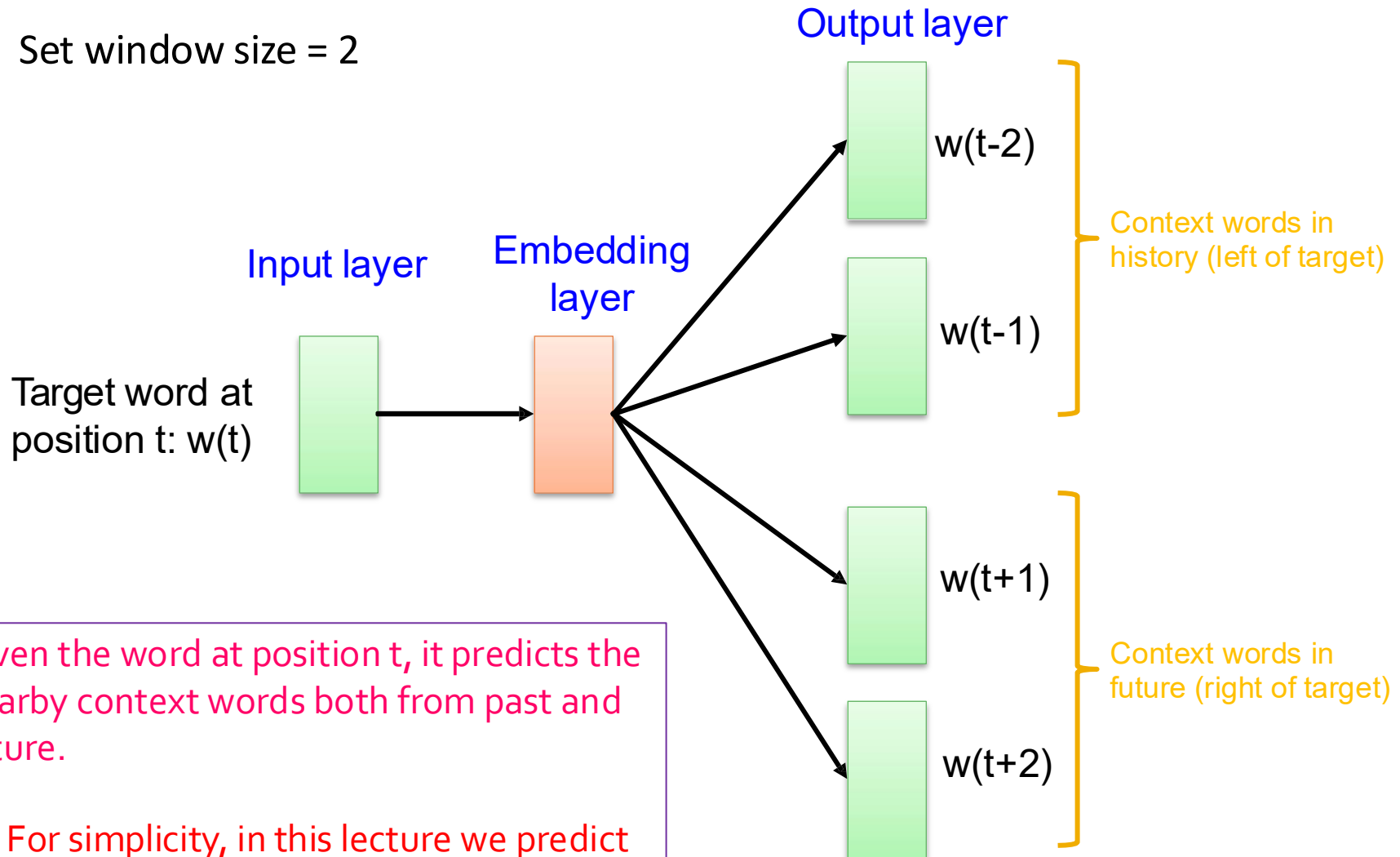
- Word2Vec is a 2 layers NN



- Given the target word as input, it predicts context words
 - In our example, if **target word** = "I" → **context** = "read", "sci-fi"
- Why is it called Skip-Gram?
 - window size** = N , the model predicts the **N-grams** words except the current word as it is the input to the model, hence the name **skip-gram**.

Word2Vec: high level architecture

- Set window size = 2

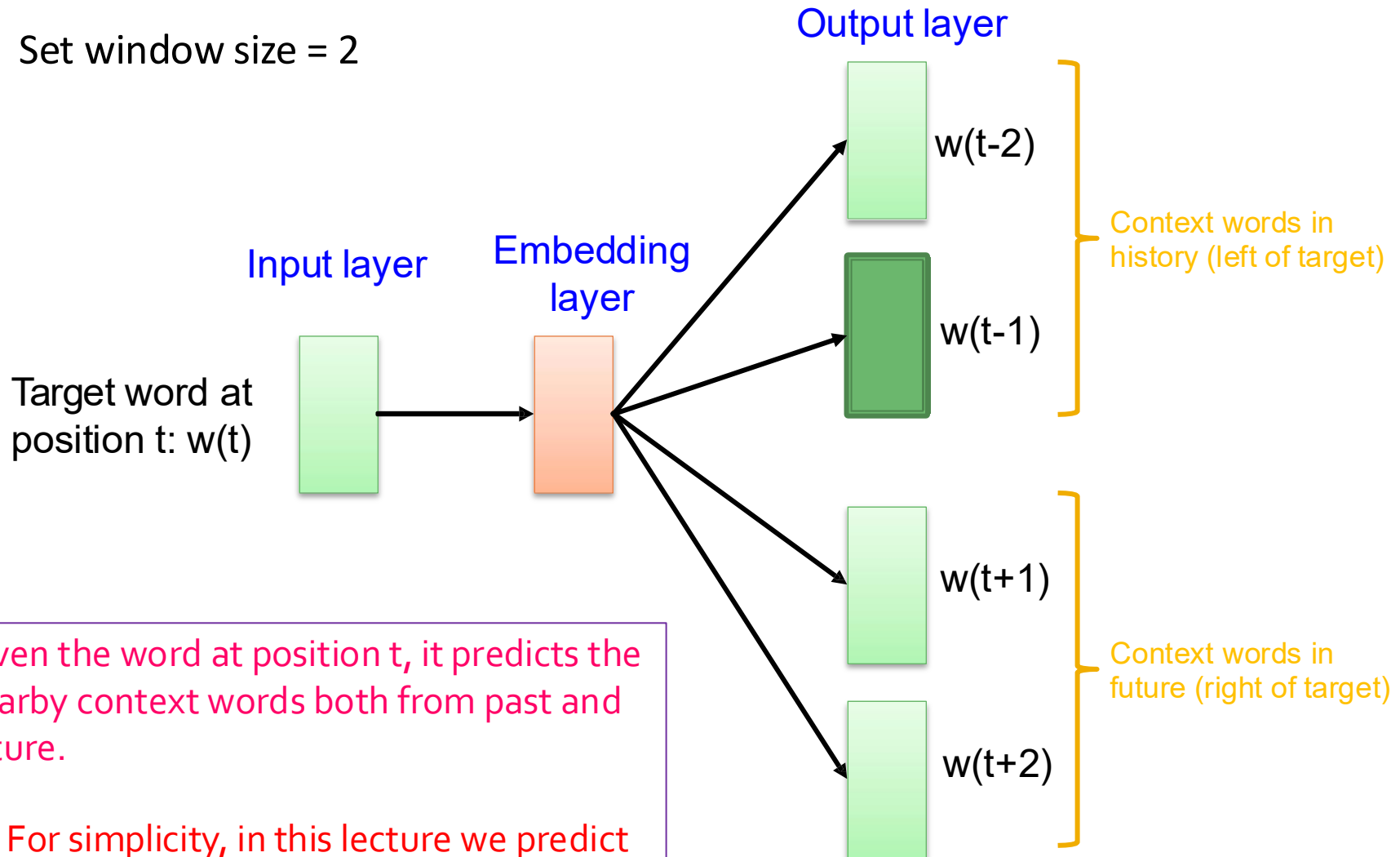


Given the word at position t , it predicts the nearby context words both from past and future.

** For simplicity, in this lecture we predict only one context word.

Word2Vec: high level architecture

- Set window size = 2

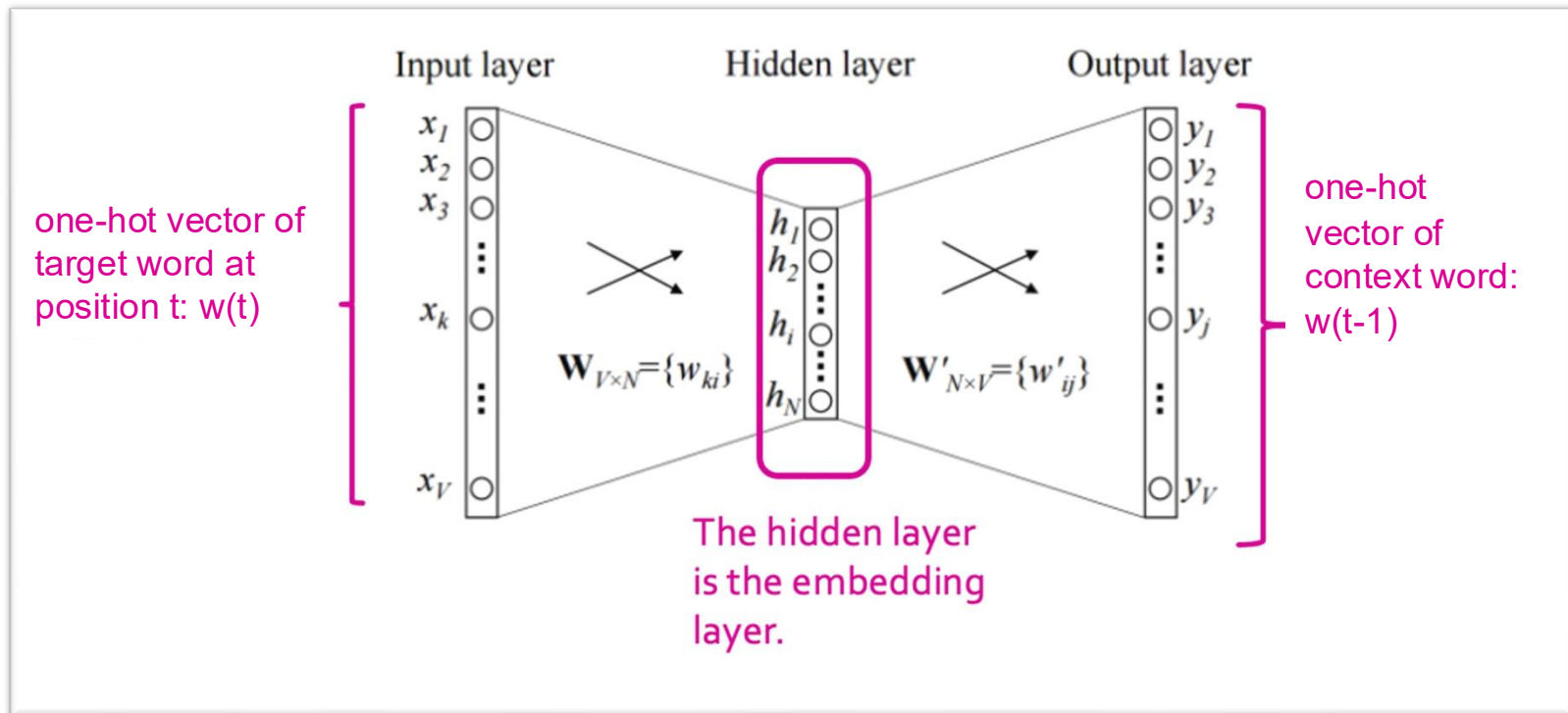
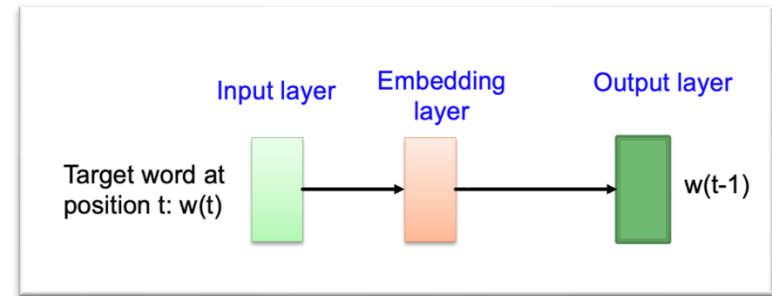


Given the word at position t , it predicts the nearby context words both from past and future.

** For simplicity, in this lecture we predict only one context word.

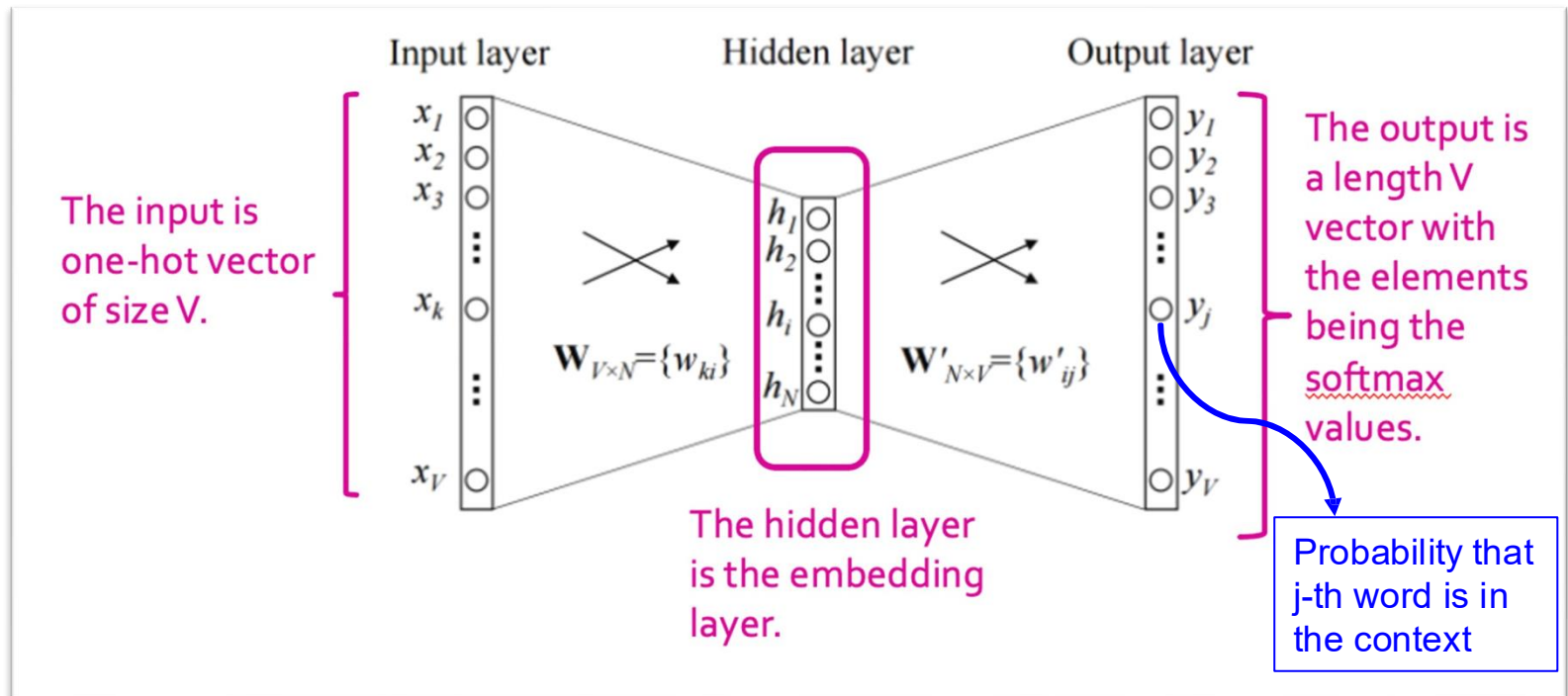
Word2Vec: architecture

- Let V = size of vocabulary
- Let N = embedding dimension



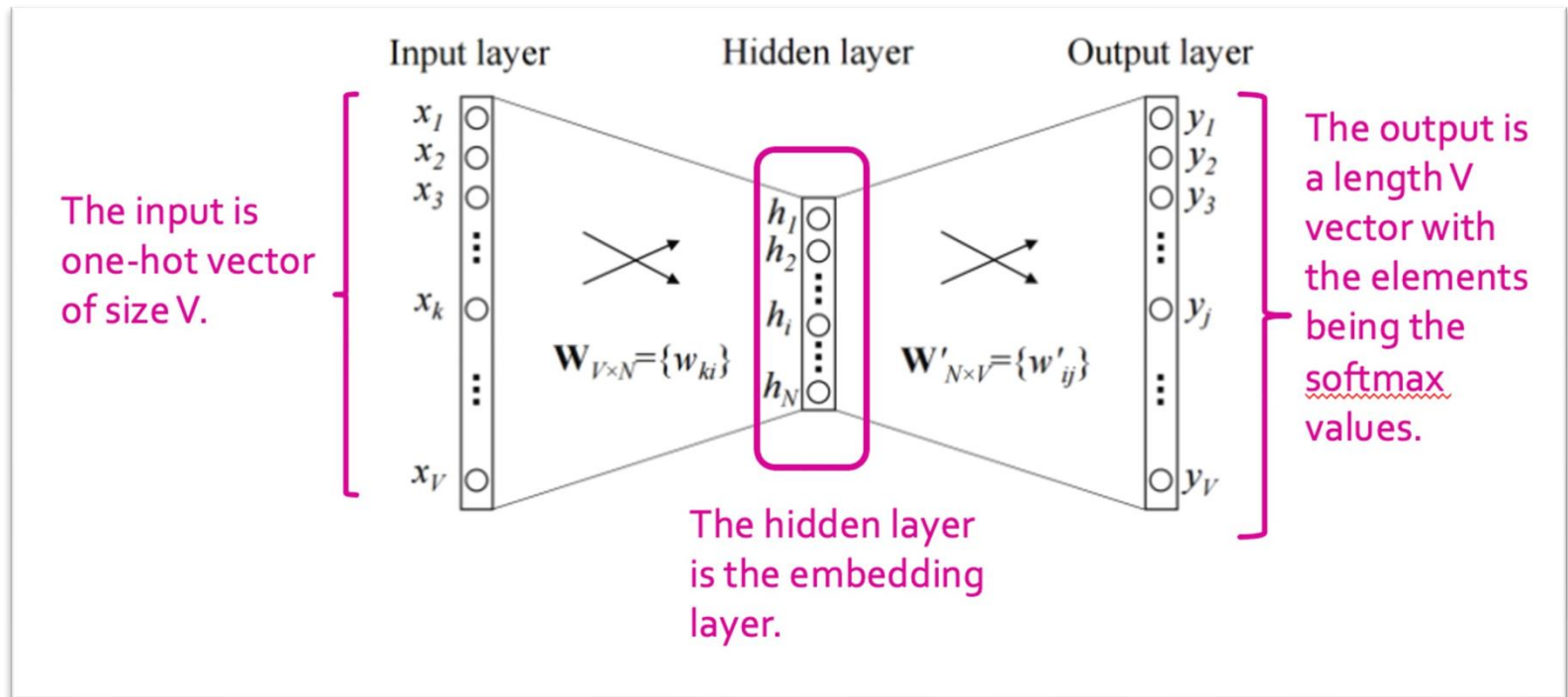
Word2Vec: architecture

- A softmax function is applied on output layer



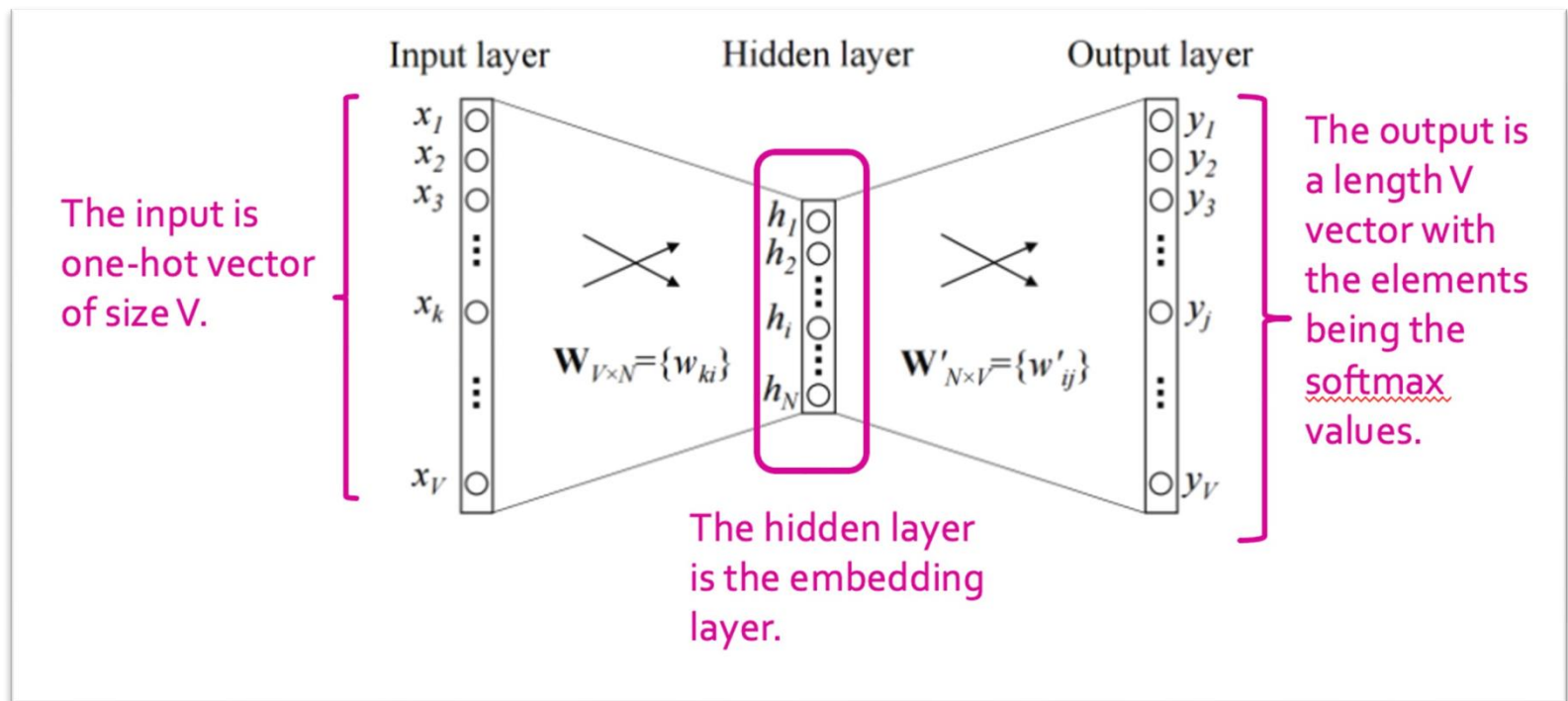
Word2Vec: architecture

- The hidden layer are all linear neurons, **no activation**!
- Two different weight matrices:
 - $W_{V \times N}$: from input to hidden layer
 - $W'_{N \times V}$: from hidden to output layer



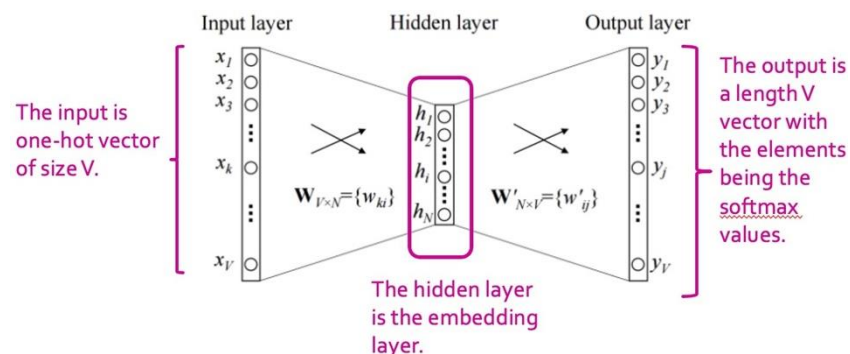
Word2Vec: architecture

- After training the network:
 - The embedding of a word is obtained by $x^T W$ i.e. matrix multiplication between word's one-hot vector and learned weights $W_{V \times N}$



Word2Vec: Training the network

- How is the network trained?
 - There are no labels. It is an unsupervised task (it is a statistical method based on co-occurrence of words in one window).
 - We therefore create a fake task!
- Fake task = given a target word, predict its context words
- Decisions to make:
 - How to make training data?
 - What is the loss function?



Word2Vec: Training data

- Take all your corpus of data = $\{d_1, d_2, \dots\}$
 - Millions of documents, wiki pages, blot posts, etc.
- Tokenize all documents and build vocabulary
 - Many methods: wordpeice, BytePairEncoding (BPE)
 - Simplest method: k-gram
- If you take $k=1$ and do tokenization by words then 1-gram is equivalent to split sentences by words

Word2Vec: Training data

- Move sliding window over tokenized documents and collect training data
- Ex: Document = {I read sci-fi books and drink orange juice}

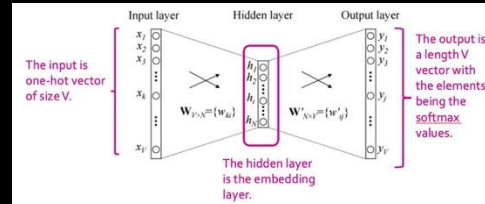
Input document (window size = 2)

I	read	sci-fi	books	and	drink	orange	juice
I	read	sci-fi	books	and	drink	orange	juice
I	read	sci-fi	books	and	drink	orange	juice
I	read	sci-fi	books	and	drink	orange	juice
I	read	sci-fi	books	and	drink	orange	juice
I	read	sci-fi	books	and	drink	orange	juice
I	read	sci-fi	books	and	drink	orange	juice
I	read	sci-fi	books	and	drink	orange	juice
I	read	sci-fi	books	and	drink	orange	juice

Training data (target, context)

- (I, read) (I, sci-fi)
- (read, I) (read, sci-fi) (read, books)
- (sci-fi, I) (sci-fi, read) (sci-fi, books)...
- (books, read) (books, sci-fi) ...
- (and, sci-fi) (and, books) (and, drink)
- (drink, books) (drink, and)
- (orange, drink) (orange, juice)
- (juice, drink) (juice, orange)

Word2Vec: Loss function



- Given the topology of the network, if $\mathbf{x}$ is the input and $\mathbf{y}$ is the output, then

$$\mathbf{y} = \text{softmax}(W'^T W^T \mathbf{x})$$

- We train against target-context pairs (w_t, w_c)
 - The context word w_c represents the ideal prediction, given the target word w_t
 - W_c is represented as one-hot, i.e. it has value 1 at some position j and other positions are 0

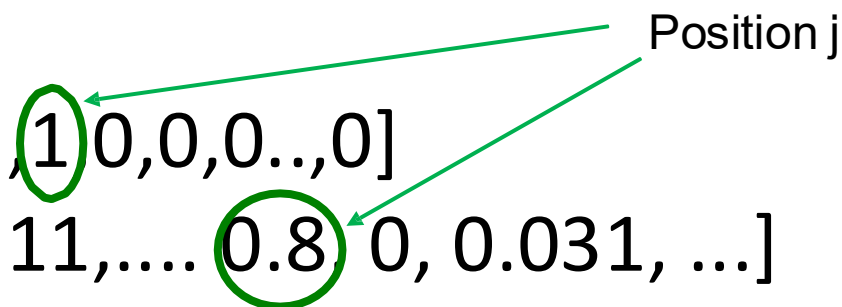
$$w_c = [0, 0, 0, \dots, 1, 0, 0, 0, \dots, 0]$$

Position j

Word2Vec: Loss function

- The loss function needs to evaluate the output layer at the **same position j** , i.e. y_j (remember y is a probability distribution; ideal value of y_j is being 1)
- We use **cross-entropy** loss function:

$$CE(w_c, y) = -\log(y_{\text{correct class}})$$

- Since $w_c = [0, 0, 0, \dots, \textcircled{1}, 0, 0, 0, \dots, 0]$
And $y = [0.02, 0.11, \dots, \textcircled{0.8}, 0, 0.031, \dots]$
- 

the loss value would be $L = -\log(0.8)$

Word2Vec: Backpropagation

- Now that loss function is clear, we want to find the values of $\mathbf{W}$ and $\mathbf{W}'$ that **minimize** it.
 - We want our model to **learn the weights**.
- We use gradient descent to tackle this
 - We find derivatives $\partial L / \partial \mathbf{W}$ and $\partial L / \partial \mathbf{W}'$ and update weights as $\mathbf{W}_{new} = \mathbf{W}_{old} - \mu \partial L / \partial \mathbf{W}$

Word2Vec: Example

- Document = {I read sci-fi books and drink orange juice}

Since this is only doc in our corpus, our vocab is

vocab = ["I", "read", "sci-fi", "books", "and", "drink", "orange", "juice"] and $V = 8$

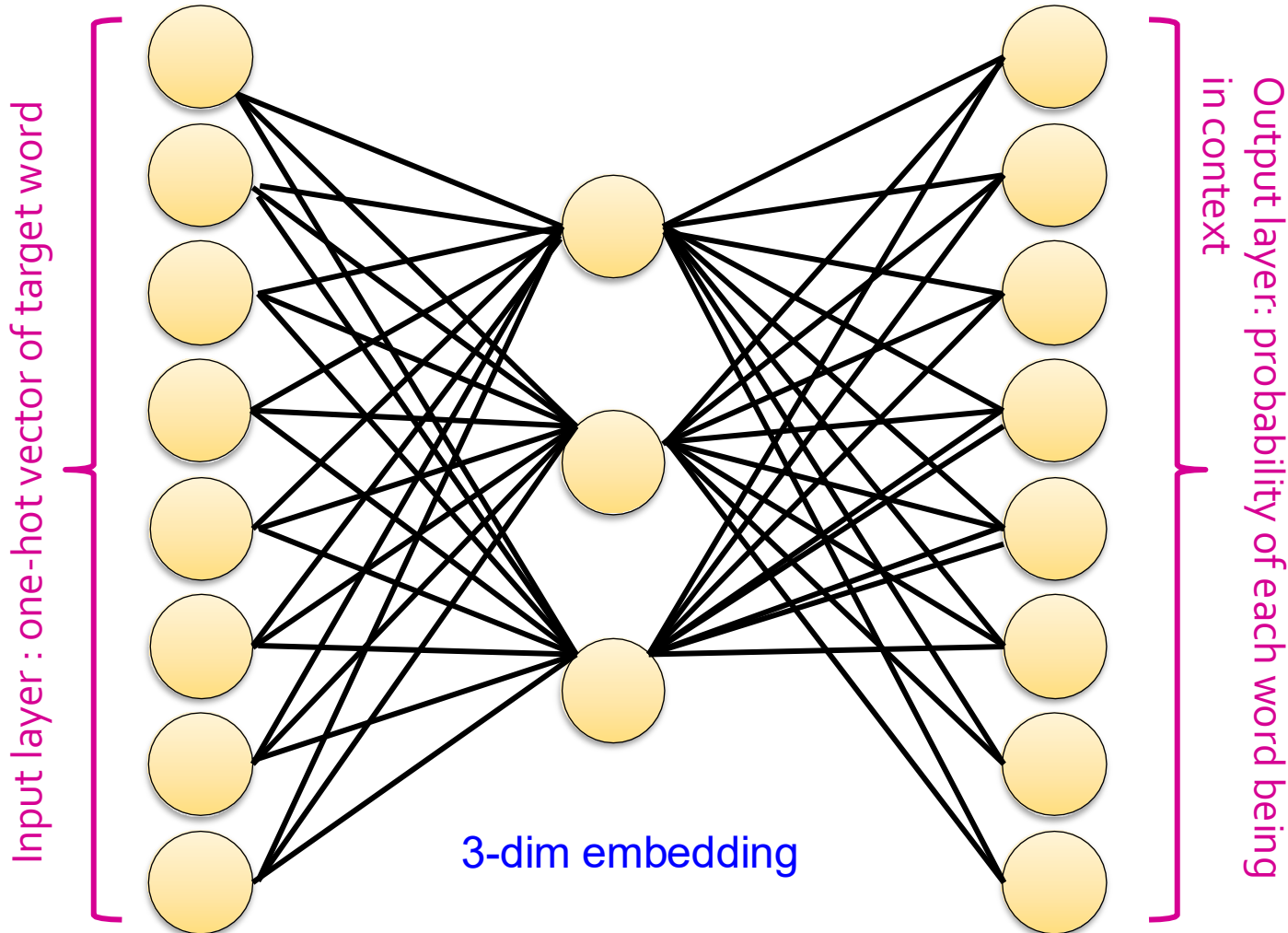
- We execute one forward pass using above document

- **Step 1:** assign one-hot vectors to words

I	:	[1, 0, 0, 0, 0, 0, 0, 0]
read	:	[0, 1, 0, 0, 0, 0, 0, 0]
sci-fi	:	[0, 0, 1, 0, 0, 0, 0, 0]
books	:	[0, 0, 0, 1, 0, 0, 0, 0]
and	:	[0, 0, 0, 0, 1, 0, 0, 0]
drink	:	[0, 0, 0, 0, 0, 1, 0, 0]
orange	:	[0, 0, 0, 0, 0, 0, 1, 0]
juice	:	[0, 0, 0, 0, 0, 0, 0, 1]

Word2Vec: Example

- size of vocabulary = 8, Let's set embedding dim = 3



Word2Vec: Example

- If target word = books and weight matrix $W_{V \times N}$ be as following:

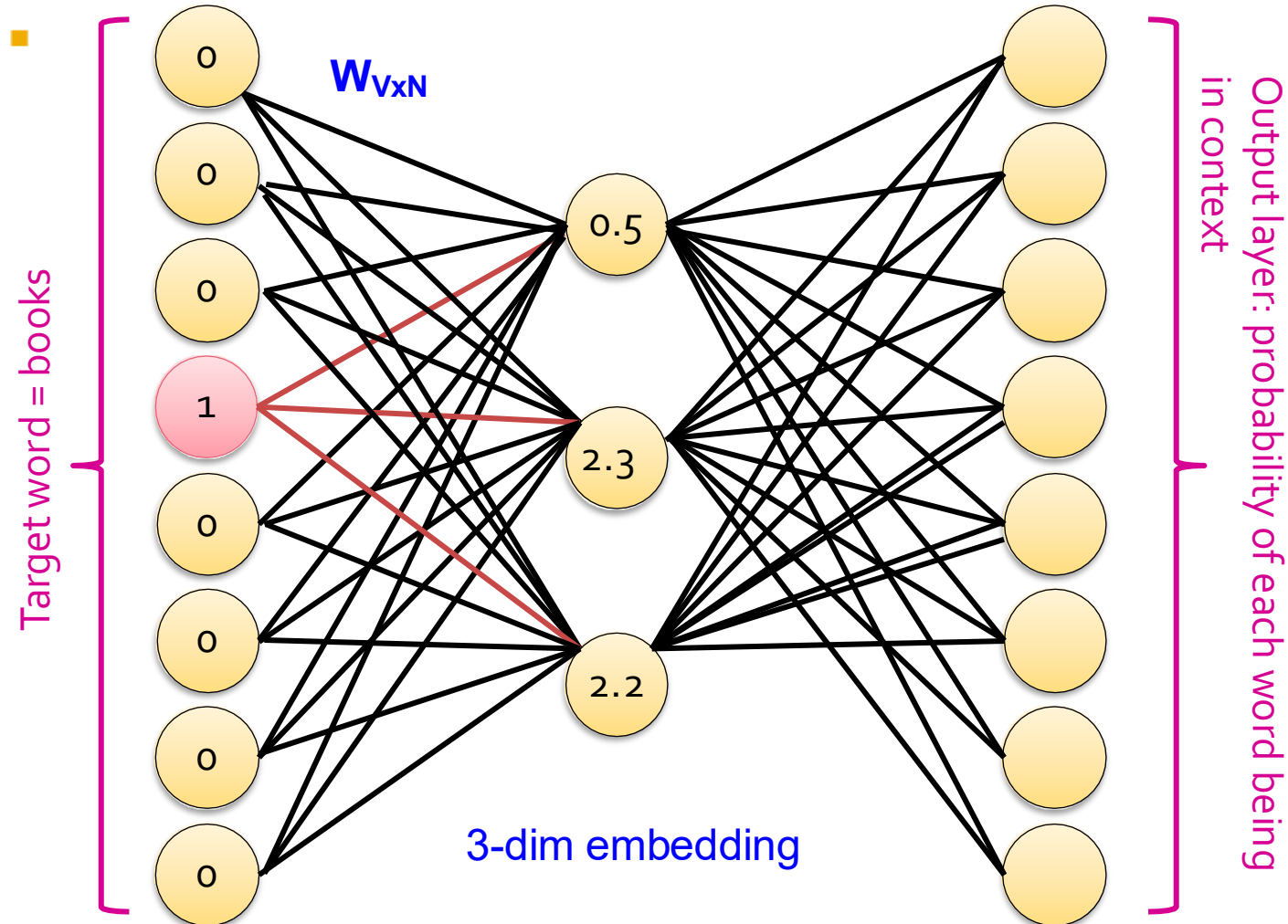
$$W_{V \times N} = \begin{bmatrix} 1 & 2 & 2 \\ -1.2 & -3 & -2 \\ 1.2 & 1.1 & 0.5 \\ 0.5 & 2.3 & 2 \\ -1.1 & 0.6 & -1 \\ 1 & -1 & 2 \\ 0.3 & 1.2 & 0.7 \end{bmatrix}$$

- Then

$$[0,0,0,1,0,0,0,0] \times \begin{bmatrix} 1 & 2 & 2 \\ -1.2 & -3 & -2 \\ 1.2 & 1.1 & 0.5 \\ 0.5 & 2.3 & 2 \\ -1.1 & 0.6 & -1 \\ 1 & -1 & 2 \\ 0.3 & 1.2 & 0.7 \end{bmatrix} = [0.5, 2.3, 2.2]$$

Word2Vec: Example

- If target word = books, and $W_{V \times N}$ given:



Word2Vec: Example

- If the weight matrix $W'_{N \times V}$ be as following:

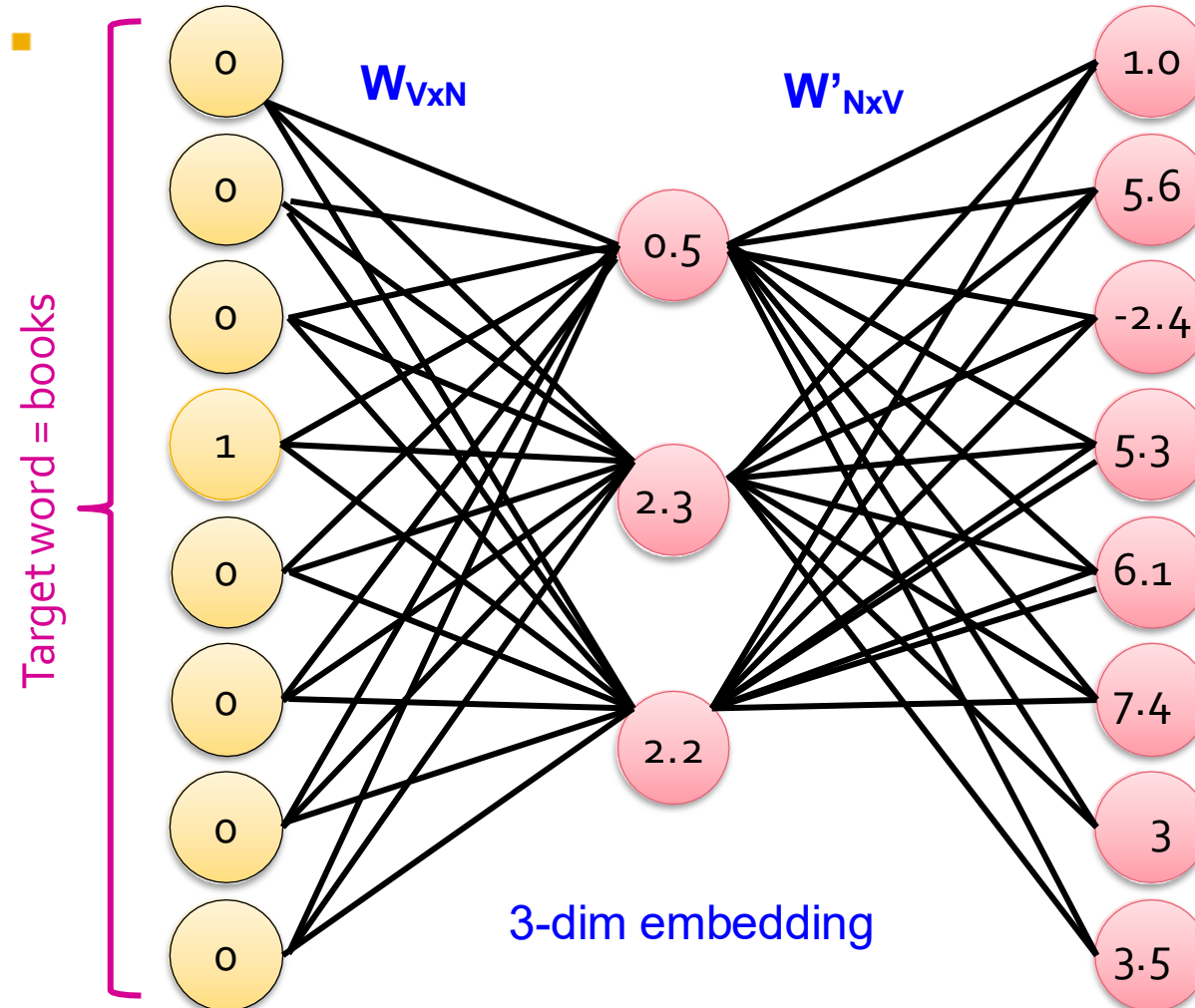
$$W'_{N \times V} = \begin{bmatrix} 1 & 2 & 2 & 0 & 0.7 & 1.3 & -1 & -0.1 \\ 1.2 & 0.5 & -1 & 1 & 0.3 & 2 & .6 & 1 \\ -1 & 1.6 & -0.5 & 1.4 & 2.3 & 1 & 1 & 0.6 \end{bmatrix}$$

- Then

$$\begin{aligned} & [0.5, 2.3, 2.2] \times \begin{bmatrix} 1 & 2 & 2 & 0 & 0.7 & 1.3 & -1 & -0.1 \\ 1.2 & 0.5 & -1 & 1 & 0.3 & 2 & 0.6 & 1 \\ -1 & 1.6 & -0.5 & 1.4 & 2.3 & 1 & 1 & 0.6 \end{bmatrix} \\ & = [1.0, 5.6, -2.4, 5.3, 6.1, 7.4, 3.0, 3.5] \end{aligned}$$

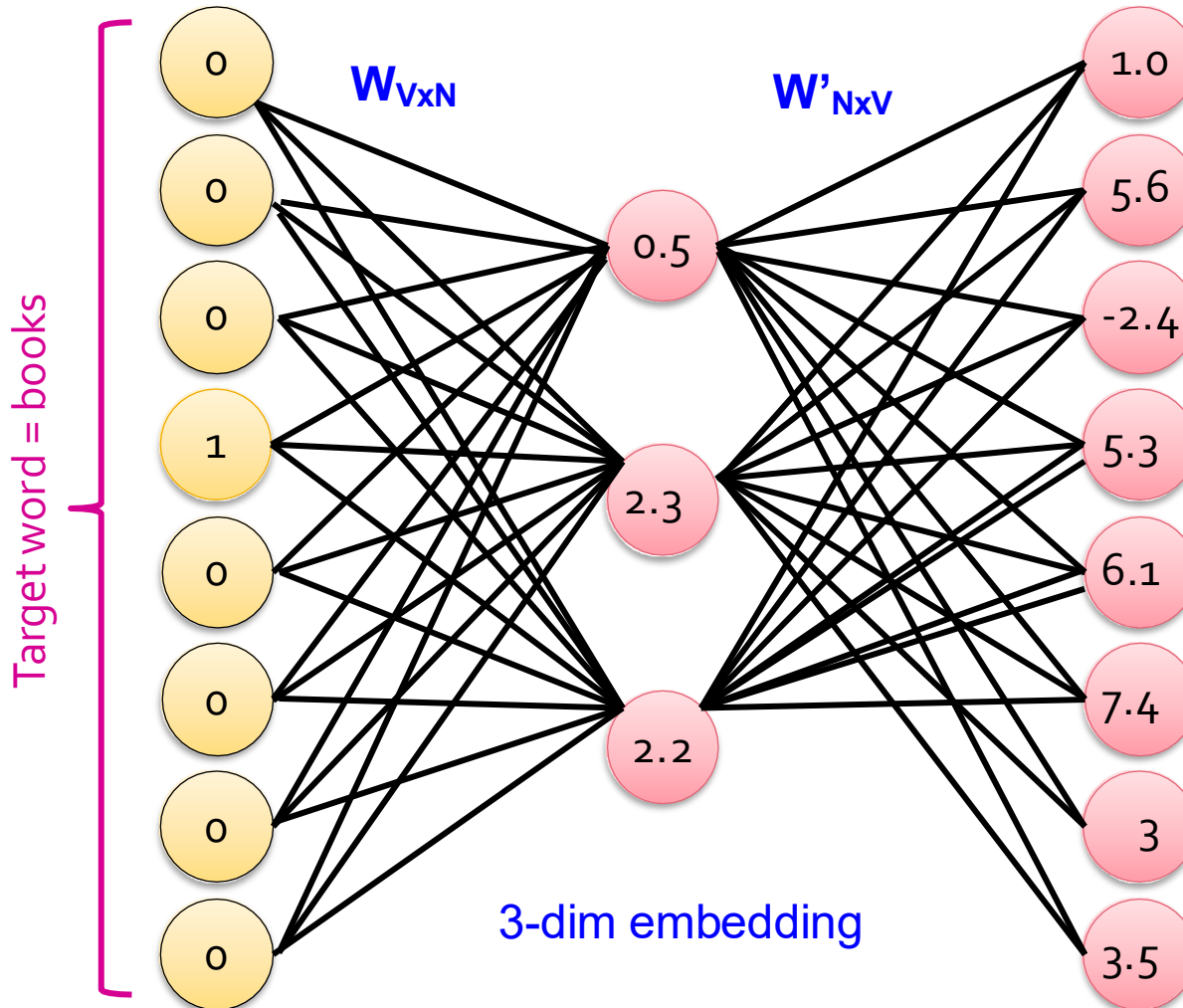
Word2Vec: Example

- If $W'_{N \times V}$ given:



Word2Vec: Example

- If $W'_{N \times V}$ given:



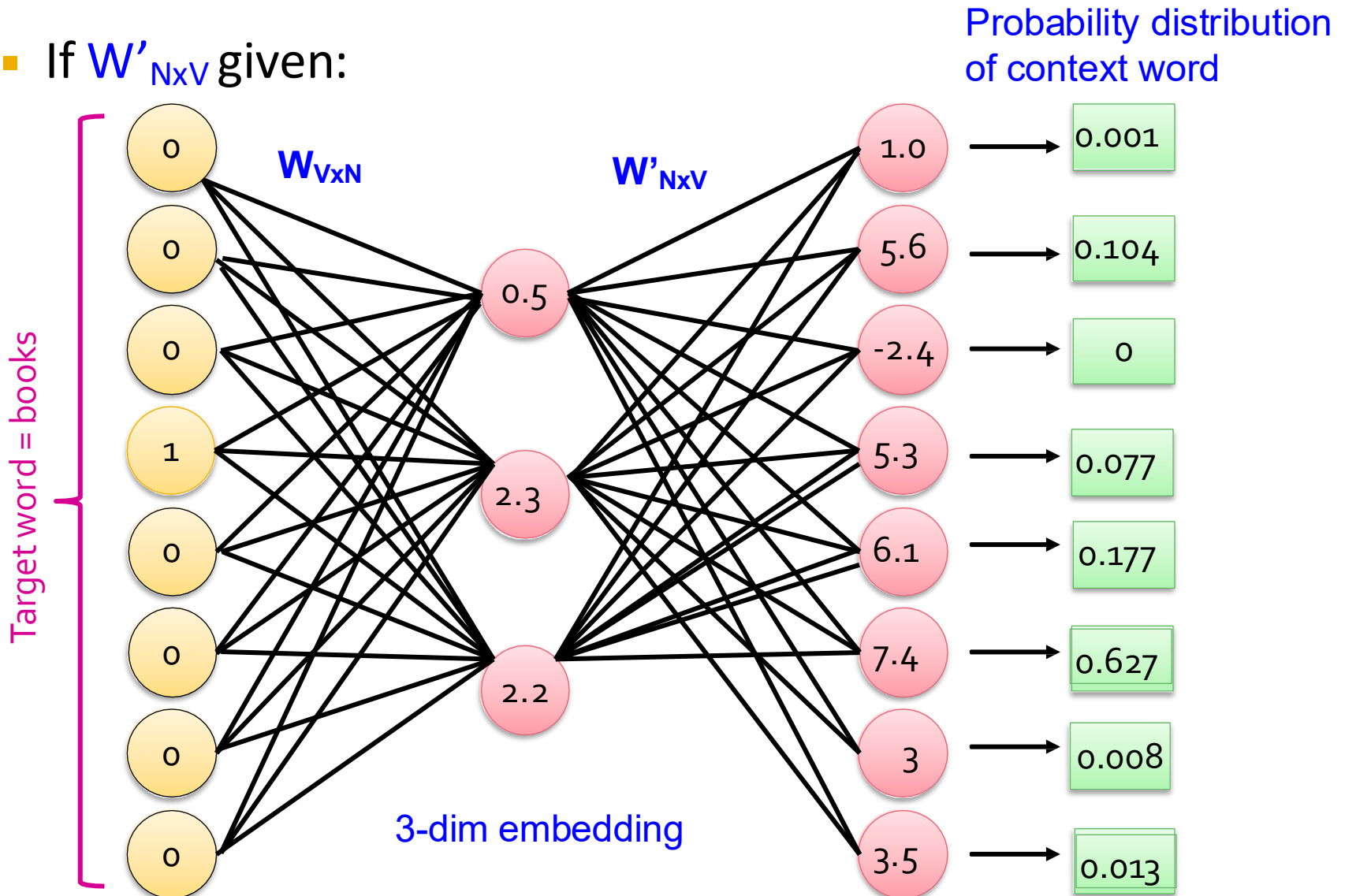
We apply softmax functions to turn them into probabilities.

Each output

$$\text{neuron } x = \frac{e^x}{\sum e^x}$$

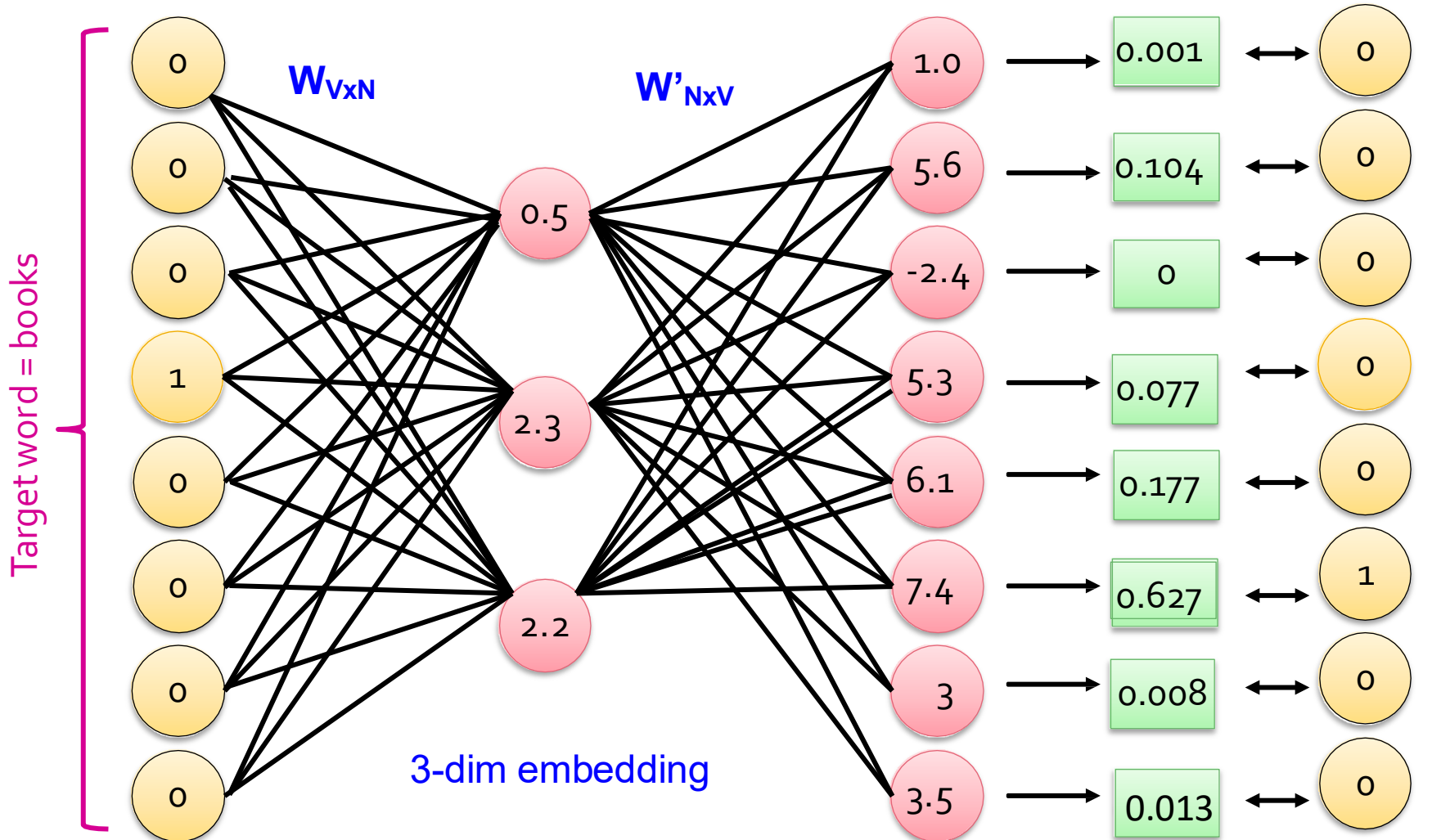
Word2Vec: Example

- If $W'_{N \times V}$ given:



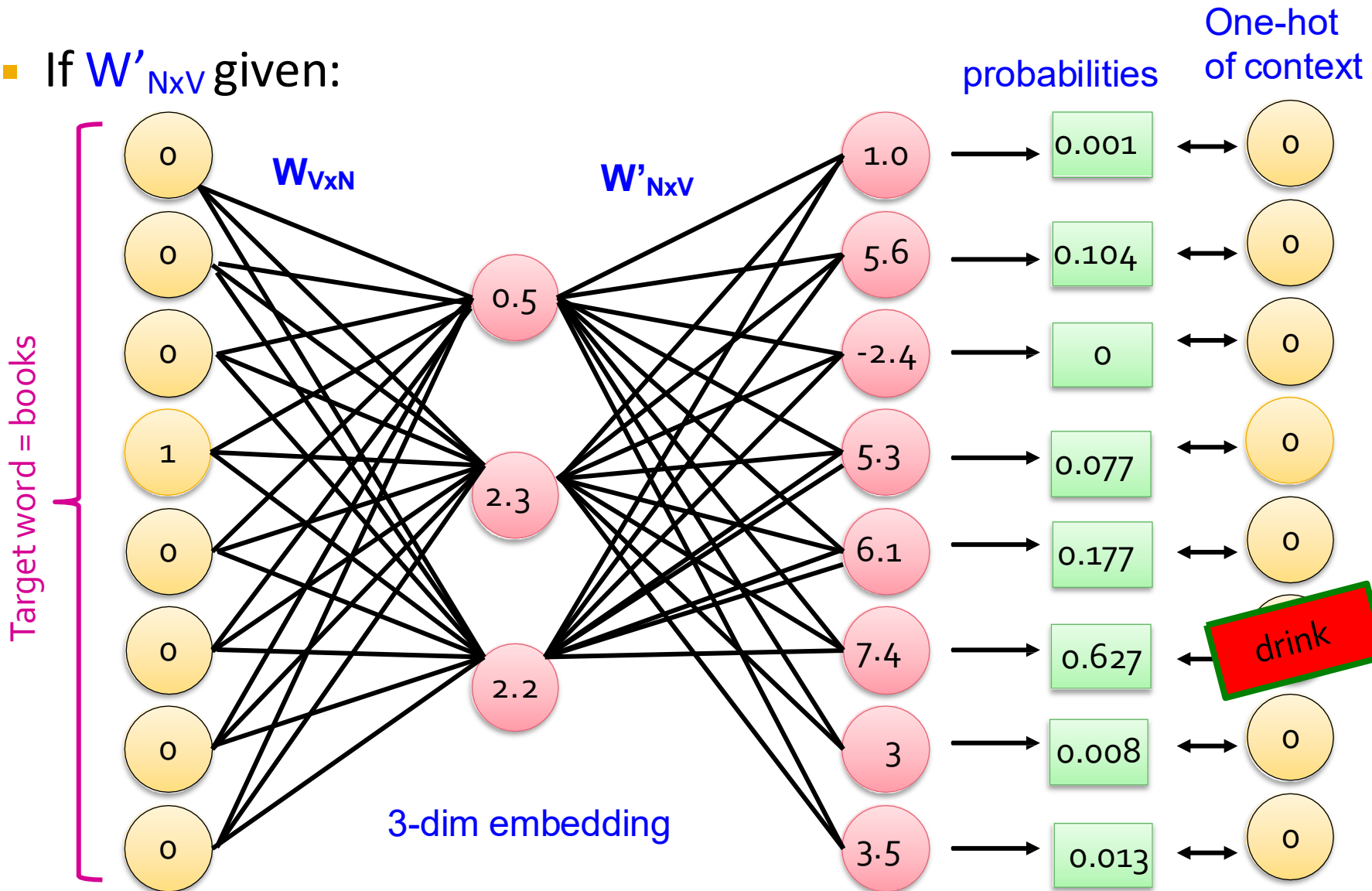
Word2Vec: Example

- If $W'_{N \times V}$ given:



Word2Vec: Example

- If $W'_{N \times V}$ given:



Word2Vec: Example

Document = {I read sci-fi books
and drink orange juice}

- The network learns by comparing softmax vector to the one-hot of true context word.
- In our example **target** = “books”, one correct **context** = “read” but we predicted “drink”
 - Predicted vector =
[0.001, 0.104, 0, 0.077, 0.177, 0.627, 0.008, 0.013]
 - One-hot of “read” = [0, 1, 0, 0, 0, 0, 0, 0]

$$L = -\ln(0.104) = 2.26$$

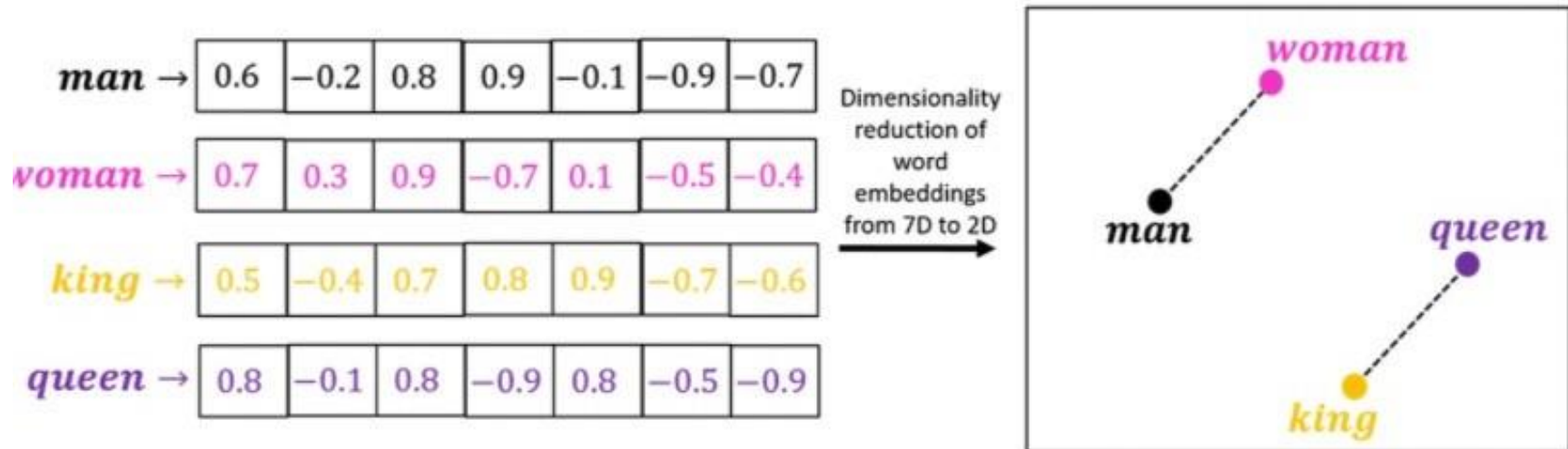
Then

Word2Vec: Summary

- Word2Vec comes in two architecture:
 - CBOW: Given context words, it predicts target word
 - Skip-Gram: Given target word, predicts context words
- Skip Gram method:
 - works well with small amount of data and is found to represent rare words well
- CBOW method:
 - is faster and more suitable for large data, it has better representations for more frequent words.

Word2Vec: Summary

- Word2vec assigns an embedding to every word in the vocabulary
- Embedding dimension $\ll$ size of the vocabulary



Task independent vs Task specific

- So far we worked with unsupervised data
 - A corpus of documents
- We learned embeddings that was not tied to any classification or regression task
 - We created a fake task of predicting nearby words
- Alternatively, we can learn embeddings for a specific tasks such as classification

Video Recommendation

Example: Recommending movies

- **Input:** 1 million movies, and 500k users who have watched some of these movies
- **Task:** recommend movies to users
- We solved this problem before using collaborative filtering, and latent factor models
- Here, we formulate it as **multi-class classification** where each movie is a class. We use neural network to learn **embeddings for movies such that similar movies have similar embeddings.**

Example: Recommending movies

- **Train-Test split:** First split data into train and test. For every user, randomly hold out few movies they have watched as test and use the rest to build train data.

Full data		Train	Test
Alice -> m1, m2, m3, m4, m5	split →	Alice -> m1, m4, m5	Alice -> m3, m2
Bob -> m8, m9, m21		Bob -> m8, m9	Bob -> m21
Sam -> m2, m6, m10		Sam -> m6, m10	Sam -> m2

Example: Recommending movies

- **Build train data:** We then build train data as pairs (movie1, movie2) where both movies are watched by same user

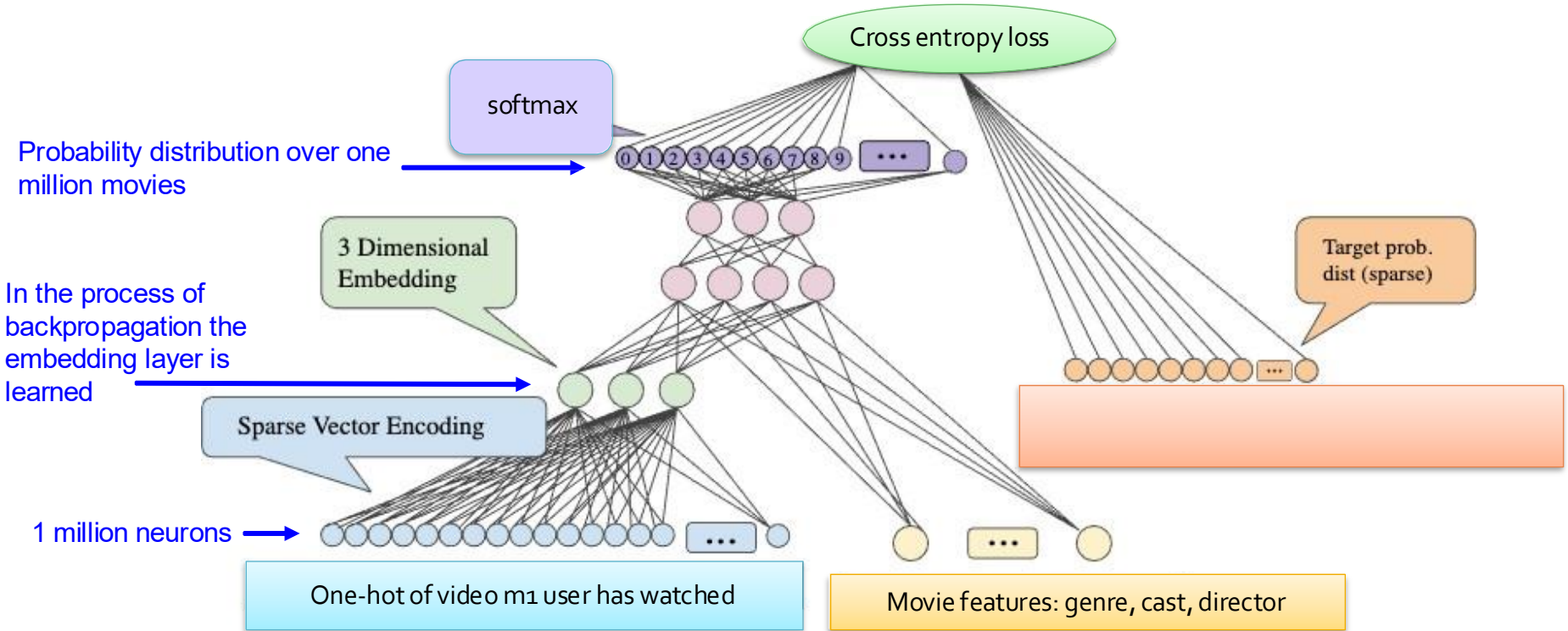
Train	Test
Alice -> m1, m4, m5	Alice -> m3, m2
Bob -> m8, m9	Bob -> m21
Sam -> m6, m10	Sam -> m2

Prepare
train data
for NN

Train	Test
(m1, m4) (m4, m5) ...	m3, m2
(m8, m9) (m9, m8)	m21
(m6, m10) (m10, m6)	m2

Example: Recommending movies

- We then build a **neural network** that performs **item-item collaborative filtering** while learning 3-dim embeddings



Example: Recommending movies

- How to recommend a movie to a user e.g. Alice?
 - Alice has watched m_1 , m_4 , m_5 in train
 - Find movies that have similar embeddings to m_1
 - Similarity score = $\langle \text{emb}(m_1), \text{emb}(v) \rangle$ for any movie v
 - Find top 5 movies with highest similarity score
 - Recommend them to Alice
 - Or even better: repeat above for m_1 , m_4 and m_5
 - Recommend movies in intersection of above sets

So far...

- So far we have seen examples of converting one-hot encodings to embeddings
 - word2Vec
 - Supervised NN with one-hot input vector
- We can use NN to learn embedding from dense feature vectors
 - What other method does the same? SVD, PCA

Autoencoders

Autoencoder

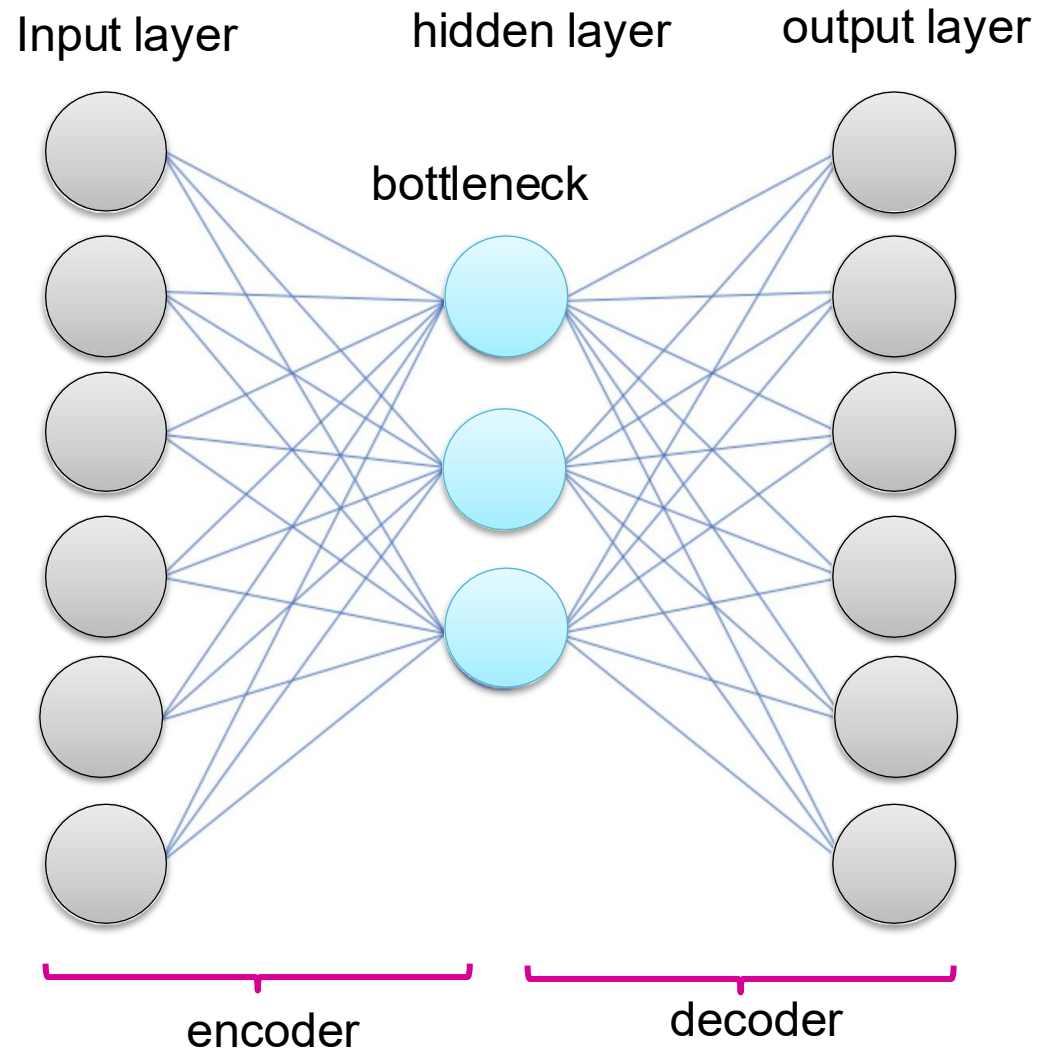
- Autoencoder are an extension of PCA to non-linear space
- They are a special type of neural network that is trained to copy input to output **except** that it has to go through a **bottleneck**
 - They are unsupervised too
- It learns to **compress** the data while **minimizing the *reconstruction error***.

Autoencoder

Input layer: is input feature vector. It does not need to be one-hot vectors. Here, input data is 6-dim vectors

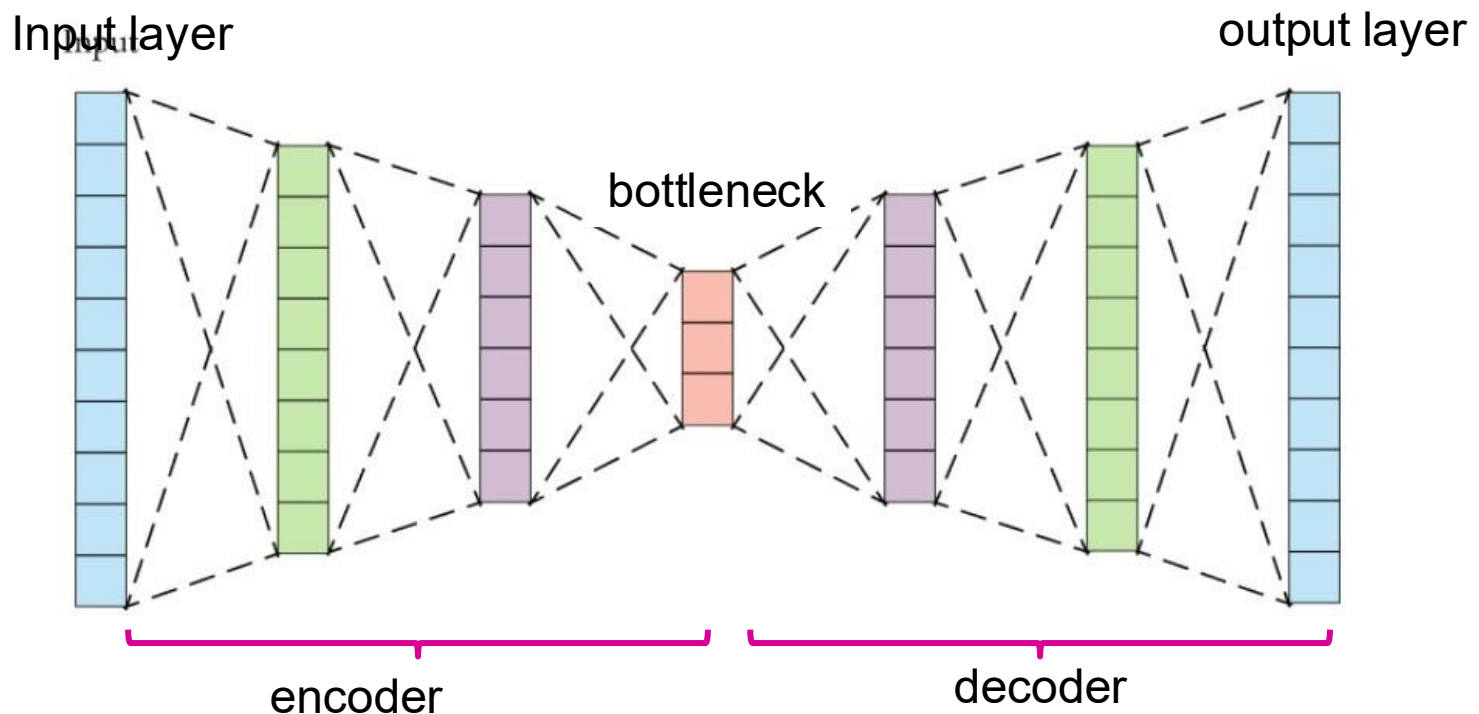
bottleneck layer: is the bottleneck as it projects down 6-dim vector to 3-dim space. It constrains the amount of information that traverses the network

Output layer: is the reconstructed input from 3-dim to 6-dim.



Autoencoder

- There can be multiple hidden layers between Input layer and bottleneck layer, similarly between bottleneck layer and output layer.



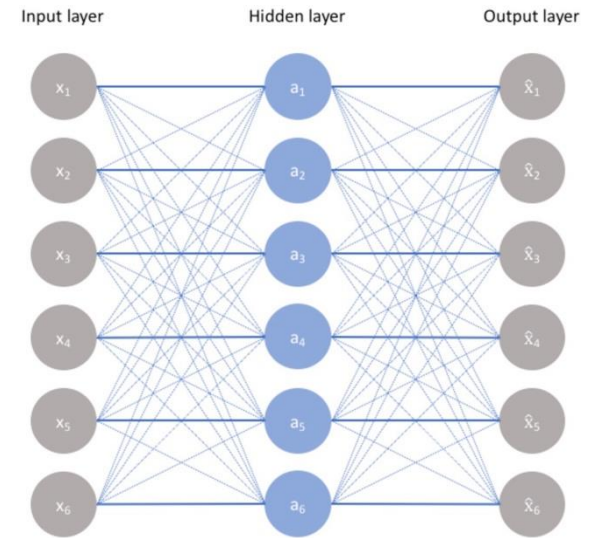
Autoencoder

Two main component in their architecture:

- **Encoder:** a function f that compresses the input into a latent-space representation
 - $f(x) = h$ such that $\text{dimension}(h) < \text{dimension}(x)$
- **Decoder:** a function g that reconstruct the input from the latent space representation
 - $g(h) \sim x$, i.e. bring h back to the original space

Autoencoder

- The bottleneck is the key:
 - Without an information bottleneck, autoencoder could learn to memorize the input data!!
- There are different types of autoencoders:
 - Undercomplete, denoising, sparse, variational
 - Today, we talk about **undercomplete autoencoder**
 - i.e **bottleneck dimension < input dimension**



Autoencoder: Training

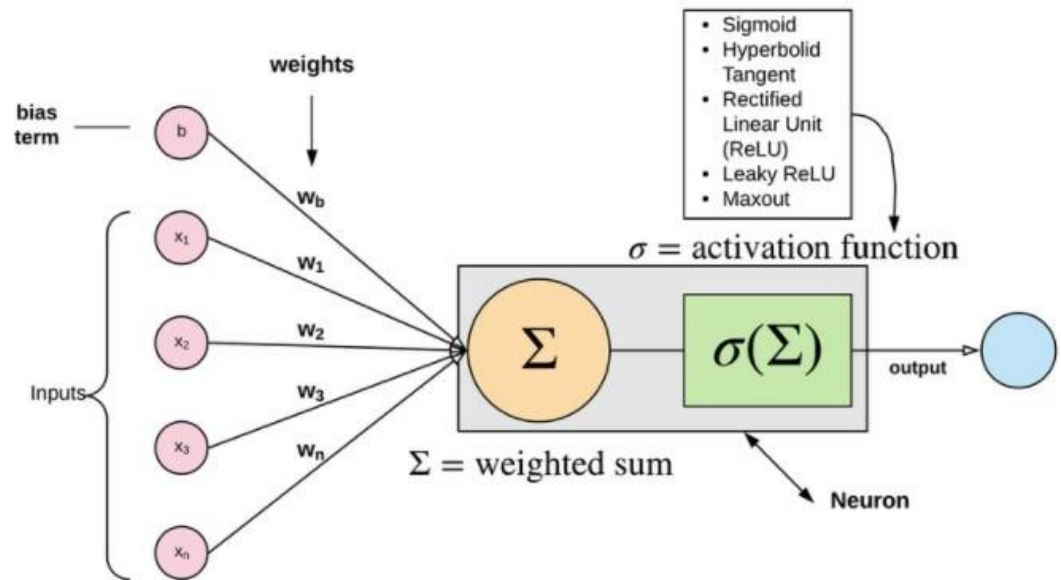
- The loss function to train an undercomplete AE is *reconstruction loss*:

$$L(x, \hat{x}) = \|x - \hat{x}\|_{1,2}$$

- No regularization term is needed in undercomplete AE. To ensure the model is not memorizing the input data we regulate:
 - size of the bottleneck layer
 - number of hidden layers

Autoencoder is a non-linear PCA

A neuron has activation functions. As long as activation function is not Identity, we learn non-linear embedding.



If we use Identity activation functions in hidden layers we convert back to PCA and produce similar dimensionality reduction as PCA.

Autoencoder: summary

- Non-linear PCA
- A neural network that is trained to copy input to output
 - it passes data through a bottleneck
 - Reconstruction loss function: L1, KL divergence
 - Unsupervised
- There are different types of autoencoders:
 - Undercomplete, denoising, sparse, variational
 - We studied undercomplete AE.

Summary

- categorical variables
 - Integer encoding
 - One-hot encoding
 - Multi-hot
- How to transform encodings to embeddings
 - SVD
 - Neural networks
- Task independent vs task specific embedding
 - Word2Vec architecture
 - Autoencoder architecture