Learning Embeddings

CS246: Mining Massive Datasets
Jure Leskovec, Stanford University
Mina Ghashami, Amazon
http://cs246.stanford.edu

What is Machine Learning?

- Machine learning is about *Optimization*
- Three key components:

  1. Training Data D $= \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$
  2. Loss function $\mathcal{L}$
  3. Model $\boldsymbol{f_\theta(x)}$

- Optimize $\boldsymbol{f_\theta(x)}$ on D w.r.t loss function $\mathcal{L}$:
  - find the parameter θ that minimizes the expected loss on the training data

$$\min_{f} J(f) = \min_{f} \frac{1}{\mathrm{n}} \sum_{i=1}^{\mathrm{n}} \mathcal{L}(f_\theta(x_i), y_i)$$

Two Dominant ML Paradigms

- **Supervised learning**:
  - Given “labeled data” $\{x, y\}$, learn $f(x) = y$
  - Ex: classification, regression
    - In linear regression, the model $f_\theta(x) = Wx + b$
    - Parameters are $\theta = \{W, b\}$
    - The loss function is mean square error (MSE)

- **Unsupervised learning**:
  - Given only “unlabeled data” $\{x\}$, learn $f(x)$
  - Ex: Dimensionality reduction, clustering
    - In SVD, the model is $f(x) = \hat{x} = VVTx$ where V is right singular vectors of input matrix.
    - The loss function is L2 loss: $L(x, \hat{x}) = \sum ||x - \hat{x}||^{2}$

Input Feature Vectors

- All ML methods work with the input feature vectors $\{x_1, x_2, \ldots, x_n\}$ 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

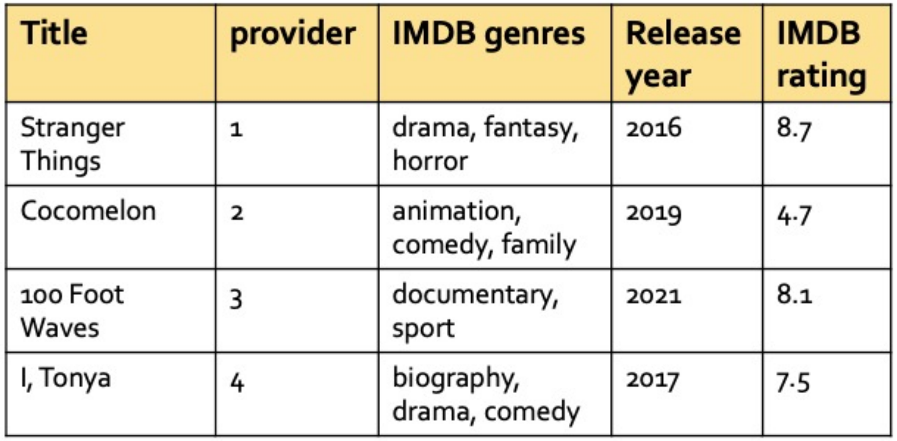
- There are two ways to encode categorical var:
  - Integer encoding
  - One-hot encoding (and multi-hot encoding)
- Consider the following movie dataset:

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.*

Stranger things	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
Cocomelon	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
100 foot wave	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
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	0	0	0

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')]
```

Applying encodings on Movies dataset

	provider				IMDB genres																													
Stranger things	1	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	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	2019	4.7
100 foot 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	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	1	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!

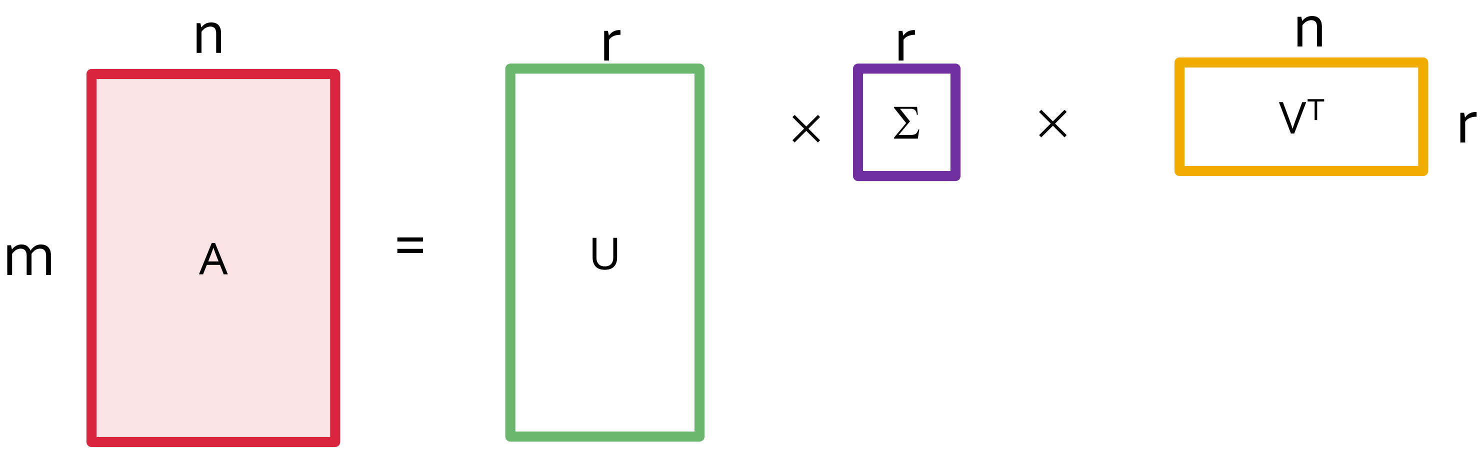
- One-hot and multi-hot encodings are not practical for features with large value sets.

From encoding to embedding

- One-hot/multi-hot encodings:
  - pros: simple, robust, the observation that simple models trained on huge amounts of data outperform complex systems trained on less data
  - cons: sparse and high dimensional, don't 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)



- **A**: **Input data matrix:** *m* x *n* matrix (e.g., *m* documents, *n* terms)
  (*r* : rank of the matrix **A –** often r < min(m,n))
- **U**: **Left singular vectors:** *m* x *r* matrix (*m* documents, *r* concepts)
- **Σ**: **Singular values:** *r* x *r* diagonal matrix (strength of each 'concept')
- **V**: **Right singular vectors:** *n* x *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 $\boldsymbol{R}^m$
    - ***V*** defines an r-dim subspace in $\boldsymbol{R}^n$

- 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

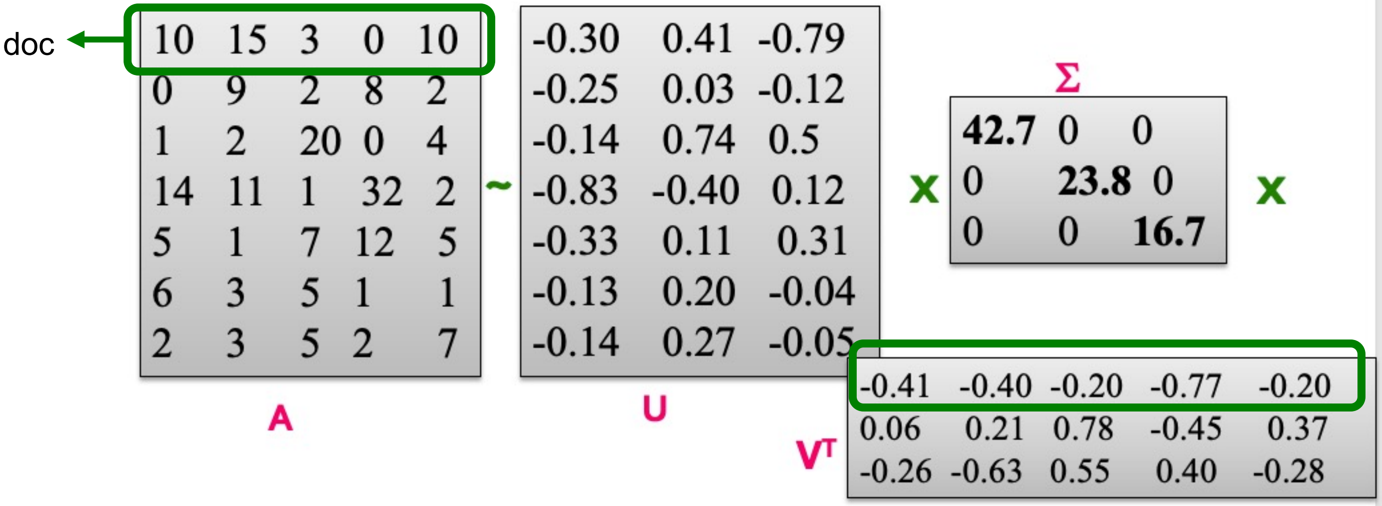
Step 2: apply SVD on the term-document matrix and pick a value $r \leq rank(A)$

Here, we set r = 3.

$$\underset{A}{\begin{bmatrix} 10 & 15 & 3 & 0 & 10 \\ 0 & 9 & 2 & 8 & 2 \\ 1 & 2 & 20 & 0 & 4 \\ 14 & 11 & 1 & 32 & 2 \\ 5 & 1 & 7 & 12 & 5 \\ 6 & 3 & 5 & 1 & 1 \\ 2 & 3 & 5 & 2 & 7 \end{bmatrix}} \sim \underset{U}{\begin{bmatrix} -0.30 & 0.41 & -0.79 \\ -0.25 & 0.03 & -0.12 \\ -0.14 & 0.74 & 0.5 \\ -0.83 & -0.40 & 0.12 \\ -0.33 & 0.11 & 0.31 \\ -0.13 & 0.20 & -0.04 \\ -0.14 & 0.27 & -0.05 \end{bmatrix}} \times \overset{\Sigma}{\begin{bmatrix} \mathbf{42.7} & 0 & 0 \\ 0 & \mathbf{23.8} & 0 \\ 0 & 0 & \mathbf{16.7} \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -0.41 & -0.40 & -0.20 & -0.77 & -0.20 \\ 0.06 & 0.21 & 0.78 & -0.45 & 0.37 \\ -0.26 & -0.63 & 0.55 & 0.40 & -0.28 \end{bmatrix}}$$

Review: SVD as an embedder

Step 3: compute embedding of documents as emb = [<doc, v1> , <doc, v2> , <doc, v3>]

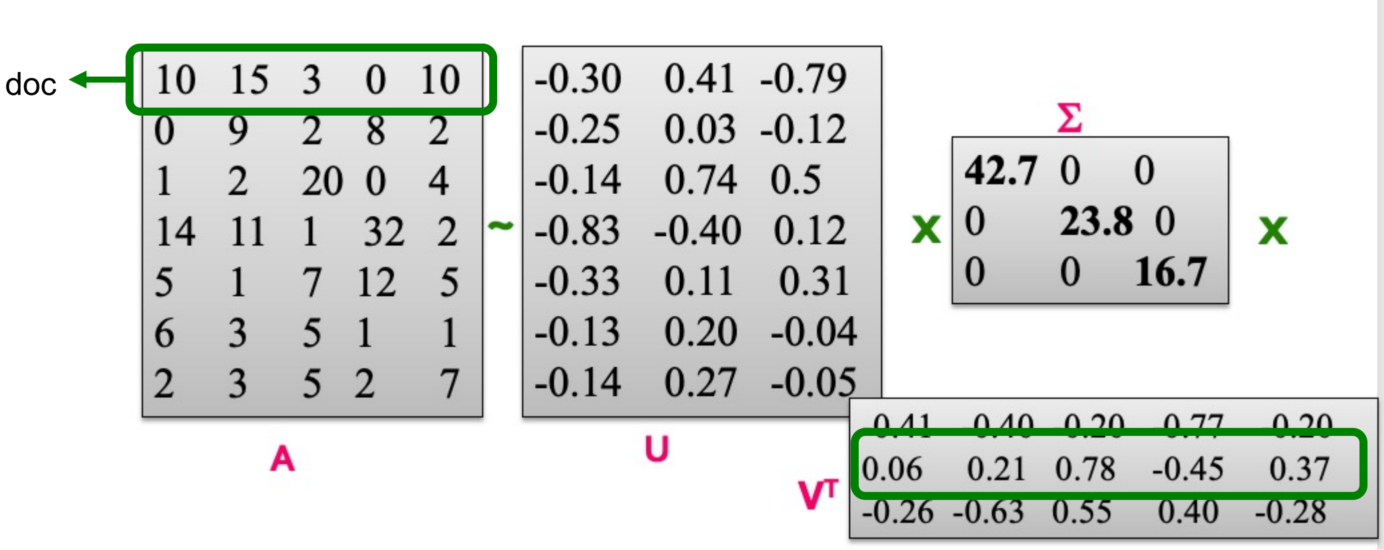


$$\underbrace{\begin{bmatrix} 10 & 15 & 3 & 0 & 10 \\ 0 & 9 & 2 & 8 & 2 \\ 1 & 2 & 20 & 0 & 4 \\ 14 & 11 & 1 & 32 & 2 \\ 5 & 1 & 7 & 12 & 5 \\ 6 & 3 & 5 & 1 & 1 \\ 2 & 3 & 5 & 2 & 7 \end{bmatrix}}_{A} \sim \underbrace{\begin{bmatrix} -0.30 & 0.41 & -0.79 \\ -0.25 & 0.03 & -0.12 \\ -0.14 & 0.74 & 0.5 \\ -0.83 & -0.40 & 0.12 \\ -0.33 & 0.11 & 0.31 \\ -0.13 & 0.20 & -0.04 \\ -0.14 & 0.27 & -0.05 \end{bmatrix}}_{U} \times \overset{\Sigma}{\begin{bmatrix} 42.7 & 0 & 0 \\ 0 & 23.8 & 0 \\ 0 & 0 & 16.7 \end{bmatrix}} \times \underbrace{\begin{bmatrix} -0.41 & -0.40 & -0.20 & -0.77 & -0.20 \\ 0.06 & 0.21 & 0.78 & -0.45 & 0.37 \\ -0.26 & -0.63 & 0.55 & 0.40 & -0.28 \end{bmatrix}}_{V^T}$$

- <doc, v1> = <[10,15,3,0,10] , v1>= -12.7

Review: SVD as an embedder

- Step 3: compute embedding of documents as emb = [<doc, v1> , <doc, v2> , <doc, v3>]

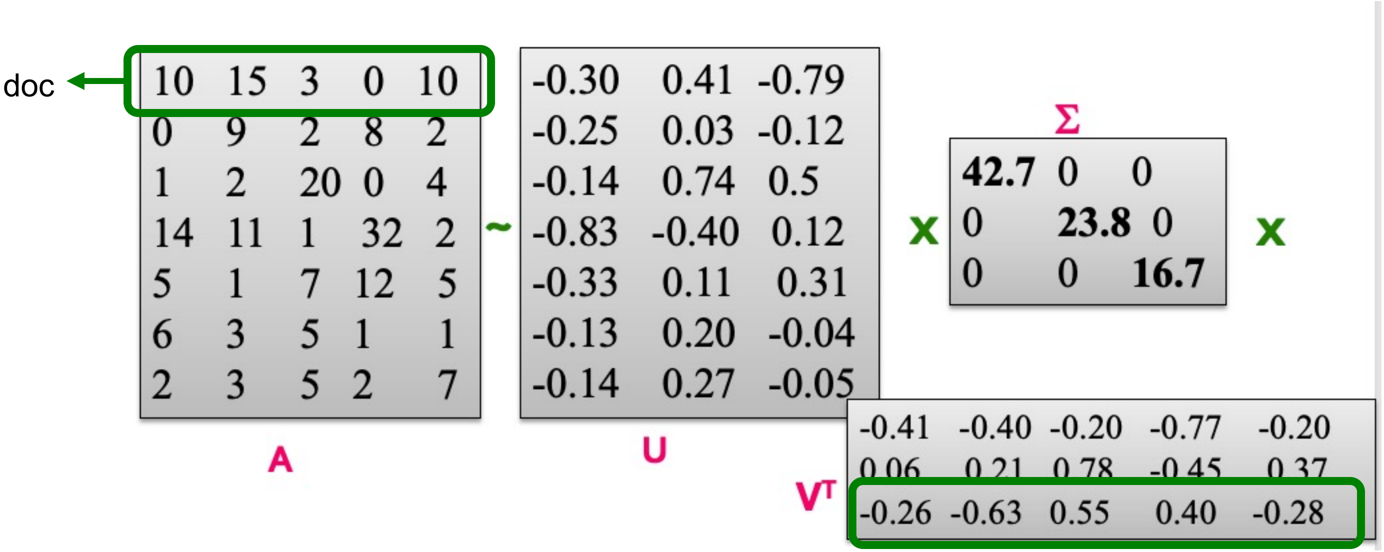


$$\underset{A}{\begin{bmatrix} 10 & 15 & 3 & 0 & 10 \\ 0 & 9 & 2 & 8 & 2 \\ 1 & 2 & 20 & 0 & 4 \\ 14 & 11 & 1 & 32 & 2 \\ 5 & 1 & 7 & 12 & 5 \\ 6 & 3 & 5 & 1 & 1 \\ 2 & 3 & 5 & 2 & 7 \end{bmatrix}} \sim \underset{U}{\begin{bmatrix} -0.30 & 0.41 & -0.79 \\ -0.25 & 0.03 & -0.12 \\ -0.14 & 0.74 & 0.5 \\ -0.83 & -0.40 & 0.12 \\ -0.33 & 0.11 & 0.31 \\ -0.13 & 0.20 & -0.04 \\ -0.14 & 0.27 & -0.05 \end{bmatrix}} \times \overset{\Sigma}{\begin{bmatrix} 42.7 & 0 & 0 \\ 0 & 23.8 & 0 \\ 0 & 0 & 16.7 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} 0.41 & 0.40 & 0.20 & 0.77 & 0.20 \\ 0.06 & 0.21 & 0.78 & -0.45 & 0.37 \\ -0.26 & -0.63 & 0.55 & 0.40 & -0.28 \end{bmatrix}}$$

- <doc, v1> = <[10,15,3,0,10] , v1>= -12.7
- <doc, v2> = <[10,15,3,0,10] , v2> = 9.79

Review: SVD as an embedder

- Step 3: compute embedding of documents as emb = [<doc, v1> , <doc, v2> , <doc, v3>]

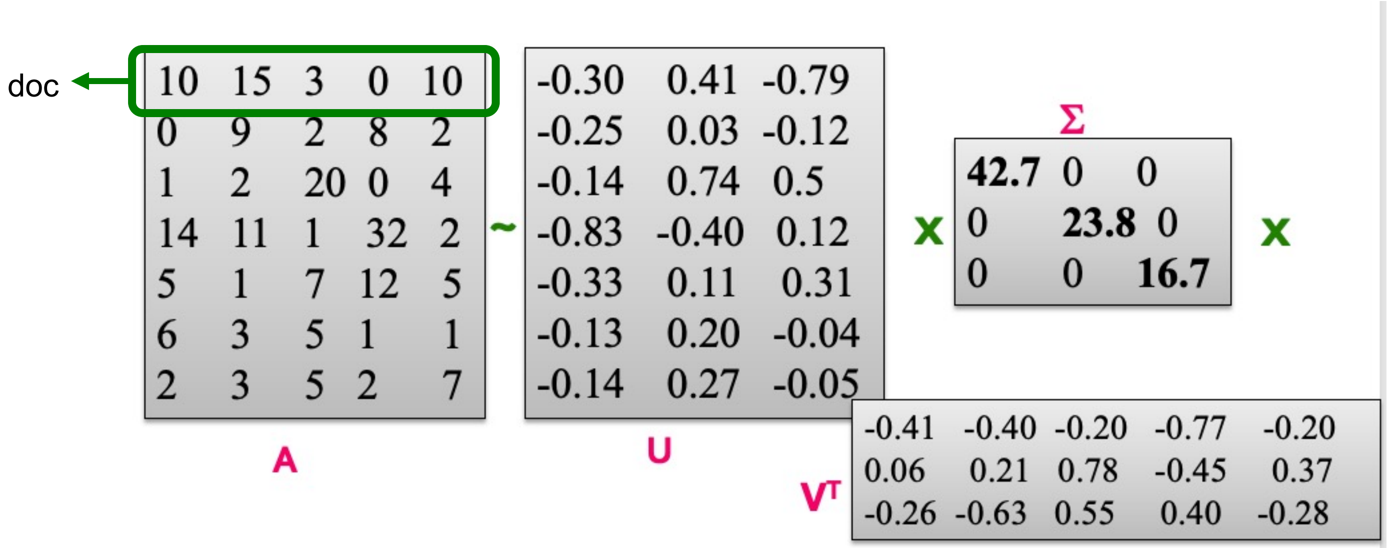


$$
\underbrace{\begin{matrix} 10 & 15 & 3 & 0 & 10 \\ 0 & 9 & 2 & 8 & 2 \\ 1 & 2 & 20 & 0 & 4 \\ 14 & 11 & 1 & 32 & 2 \\ 5 & 1 & 7 & 12 & 5 \\ 6 & 3 & 5 & 1 & 1 \\ 2 & 3 & 5 & 2 & 7 \end{matrix}}_{A}
\sim
\underbrace{\begin{matrix} -0.30 & 0.41 & -0.79 \\ -0.25 & 0.03 & -0.12 \\ -0.14 & 0.74 & 0.5 \\ -0.83 & -0.40 & 0.12 \\ -0.33 & 0.11 & 0.31 \\ -0.13 & 0.20 & -0.04 \\ -0.14 & 0.27 & -0.05 \end{matrix}}_{U}
\times
\overset{\Sigma}{\begin{matrix} 42.7 & 0 & 0 \\ 0 & 23.8 & 0 \\ 0 & 0 & 16.7 \end{matrix}}
\times
V^T \begin{matrix} -0.41 & -0.40 & -0.20 & -0.77 & -0.20 \\ 0.06 & 0.21 & 0.78 & -0.45 & 0.37 \\ -0.26 & -0.63 & 0.55 & 0.40 & -0.28 \end{matrix}
$$

- <doc, v1> = <[10,15,3,0,10] , v1>= -12.7
- <doc, v2> = <[10,15,3,0,10] , v2> = 9.79
- <doc, v3> = <[10,15,3,0,10] , v3>= -13.9

Review: SVD as an embedder

- Step 3: compute embedding of documents as emb = [<doc, v1> , <doc, v2> , <doc, v3>]



$$\underset{A}{\begin{bmatrix} \boxed{10 & 15 & 3 & 0 & 10} \\ 0 & 9 & 2 & 8 & 2 \\ 1 & 2 & 20 & 0 & 4 \\ 14 & 11 & 1 & 32 & 2 \\ 5 & 1 & 7 & 12 & 5 \\ 6 & 3 & 5 & 1 & 1 \\ 2 & 3 & 5 & 2 & 7 \end{bmatrix}} \sim \underset{U}{\begin{bmatrix} -0.30 & 0.41 & -0.79 \\ -0.25 & 0.03 & -0.12 \\ -0.14 & 0.74 & 0.5 \\ -0.83 & -0.40 & 0.12 \\ -0.33 & 0.11 & 0.31 \\ -0.13 & 0.20 & -0.04 \\ -0.14 & 0.27 & -0.05 \end{bmatrix}} \times \overset{\Sigma}{\begin{bmatrix} 42.7 & 0 & 0 \\ 0 & 23.8 & 0 \\ 0 & 0 & 16.7 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -0.41 & -0.40 & -0.20 & -0.77 & -0.20 \\ 0.06 & 0.21 & 0.78 & -0.45 & 0.37 \\ -0.26 & -0.63 & 0.55 & 0.40 & -0.28 \end{bmatrix}}$$

- <doc, v1> = <[10,15,3,0,10] , v1>= -12.7
- <doc, v2> = <[10,15,3,0,10] , v2> = 9.79
- <doc, v3> = <[10,15,3,0,10] , v3>= -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

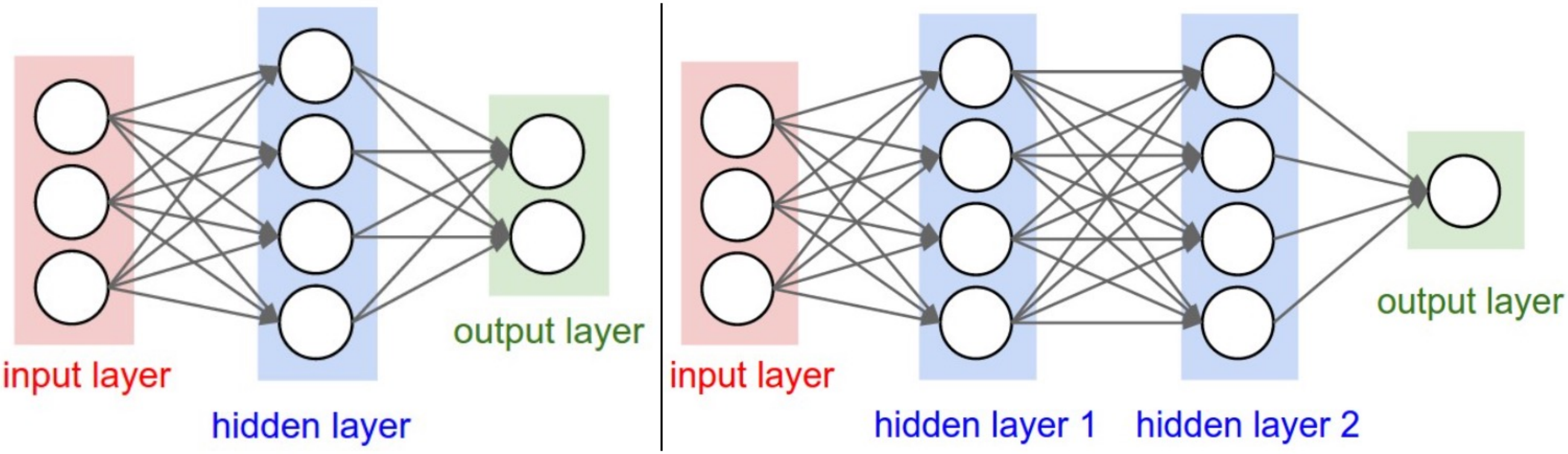
- State of the art embedders are among neural networks

- Can we use neural networks to create non-linear embedding?

Neural Networks Fundamentals

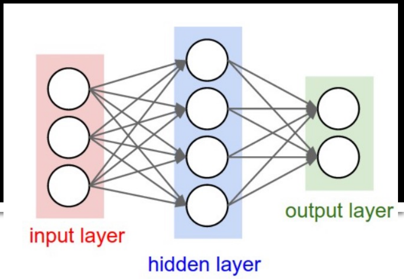
Neural Network: Architecture

- A neural network is a collection of neurons that are connected in an *acyclic graph*
- Outputs of some neurons are inputs to other neurons, and they are organized into layers



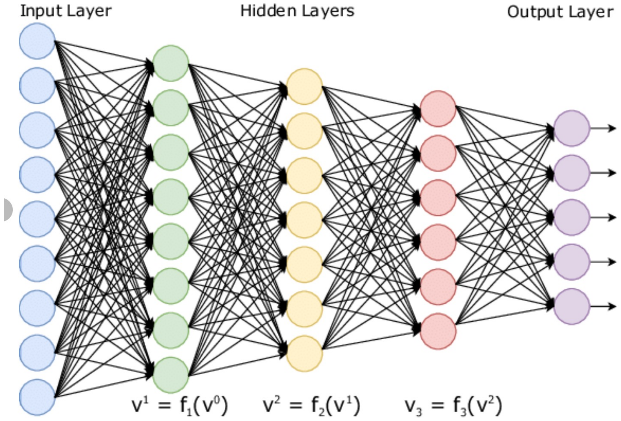
credit: cs231

Neural Network



- **Fully-connected layer** is the most common layer type:
  - neurons between two adjacent layers are fully pairwise connected
  - neurons within a single layer share no connections

- Number of hidden layers and neurons in each hidden layer are hyperparameters of the network

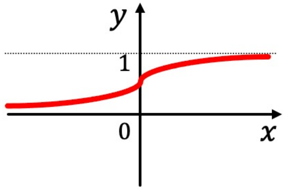


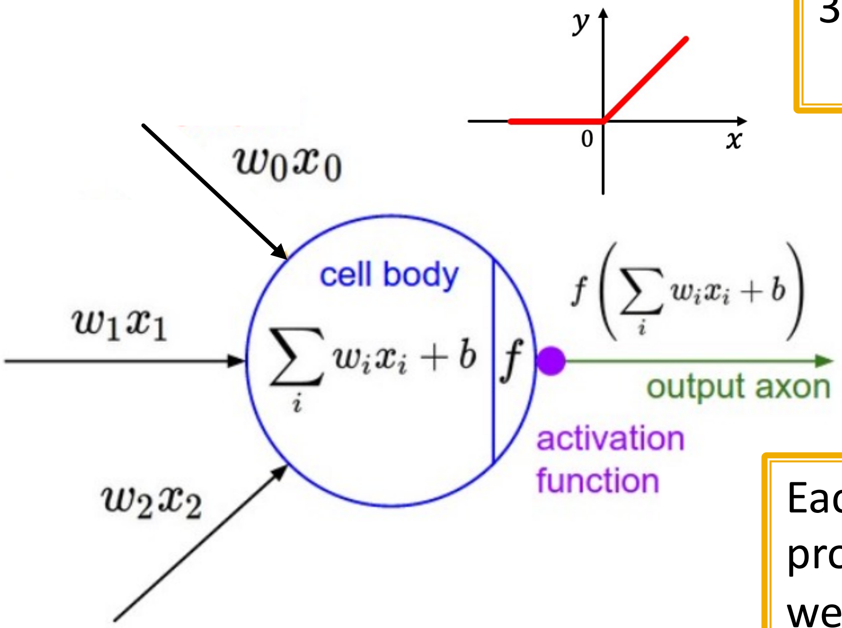
Neural Network: A Neuron

- A neuron is a classifier
  - Input: $[x_0, x_1, x_2]$
  - Output = $f(\sum w_i x_i + b)$

f is the activation function, It takes a single number and performs an operation on it. Some choices are:

1. Sigmoid $\sigma(x) = 1/(1 + e^{-x})$
2. Tanh $\tanh(x) = 2\sigma(2x) - 1.$
3. Relu $f(x) = \max(0, x)$



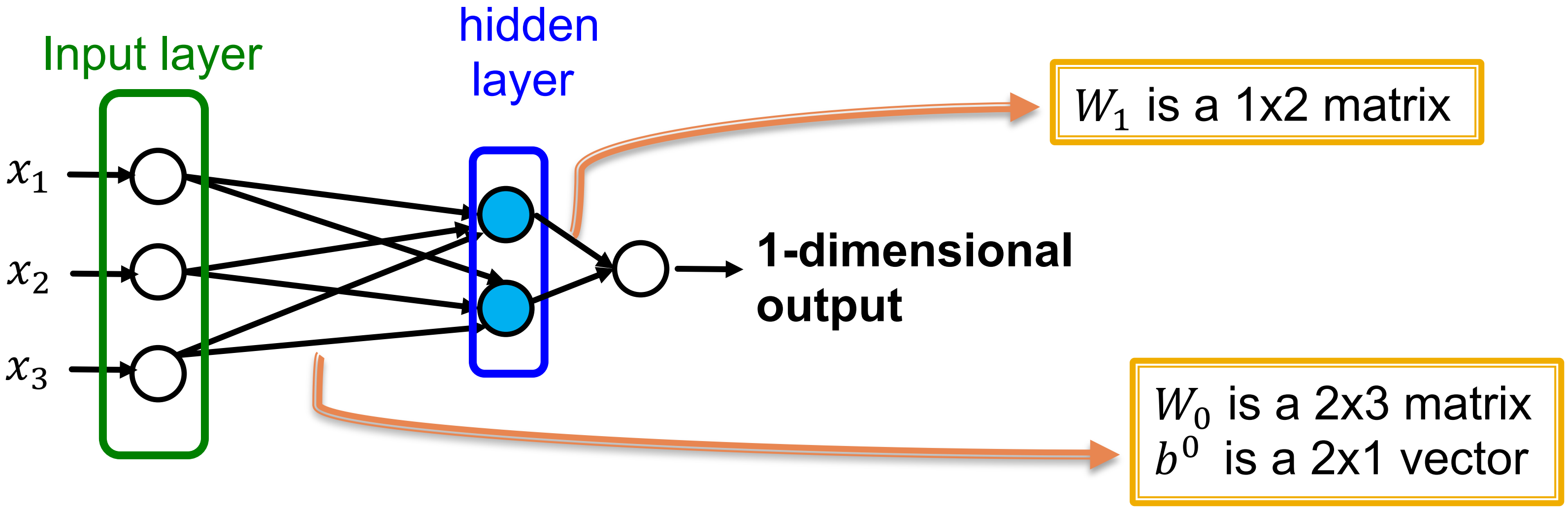


Each neuron performs a dot product with the input and its weights, adds the bias and applies the activation function

credit: cs231

Neural Network: A Layer

- Consider two neurons in a hidden layer



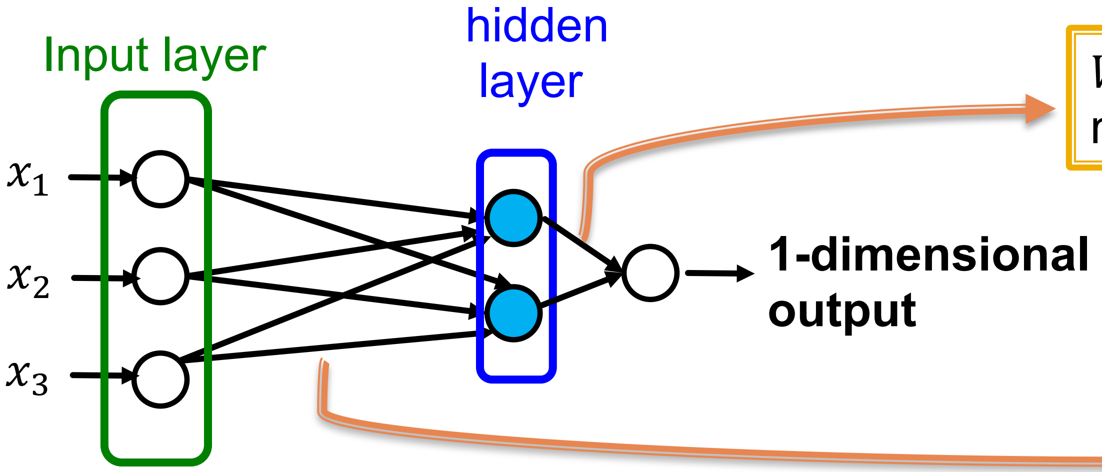
W_1 is a 1x2 matrix

W_0 is a 2x3 matrix
b^0 is a 2x1 vector

- Each layer computes $\boldsymbol{x}^{(l+1)} = \sigma(W_l \boldsymbol{x}^{(l)} + b^l)$
- W_l is weight matrix that transforms representation at layer l to layer $l+1$
- b^l is bias at layer l, and is added to the linear transformation of $\boldsymbol{x}$
- σ is sigmoid activation function

Neural Network: A Layer

- This network computes $f(x) = W_1(\sigma(W_0 \boldsymbol{x}^{(0)} + b^0) + b^1)$



W_1 is a 1x2 matrix

W_0 is a 2x3 matrix

- Notice without activation functions, *f(x) will be linear in x* !!

$$f(x) = W_1 W_0 \boldsymbol{x}^{(0)} + W_1 b^0 + b^1$$

Neural Network: Loss Function

- A loss function $\mathcal{L}$ is required to train the NN.
  - Example: L2 loss

$$\mathcal{L}\big(\boldsymbol{y}, f(\boldsymbol{x})\big) = \|y - f(x)\|_2$$

- Common loss functions for **regression**:
  - L2 loss, L1 loss, huber loss, …
- Common loss functions for **classification**:
  - Cross entropy, max margin (hinge loss), …
- Example
  - See https://pytorch.org/docs/stable/nn.html#loss-functions

Cross Entropy Loss

- Common loss for classification tasks
  - Defined between one-hot of true label and the predicted probability distribution over classes

- Ex: Task = multi-class classification with 5 classes
  - True label y belongs to class 3, so one-hot of y =

0	0	1	0	0

  - Predicted probability distribution $\hat{y} = f(x)$ =

0.1	0.3	0.4	0.1	0.1

  - $\mathrm{CE}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{i=1}^{C}(y_i \log \hat{y}_i) = -\log(\widehat{y}_{\text{correct class}})$

  - $y_i, \hat{y}_i$ are the **actual** and **predicted** value of the i-th class.

- **Intuition:** the lower the loss, the closer the prediction is to one-hot $\boldsymbol{y}$

Cross Entropy Loss

- Often model's output is a score for each class, not a probability distribution
- To convert to probability distribution:

$$f(\boldsymbol{x}) = \text{Softmax}\big(g(\boldsymbol{x})\big)$$

Probability distribution over classes

Output score for each class

  - $$f(\boldsymbol{x})_i = \frac{e^{g(x)_i}}{\sum_{j=1}^{C} e^{g(x)_j}}$$

Cross entropy loss sometimes is referred to as softmax loss

  - It normalizes a vector into a probability distribution that sums to 1

Neural Network: Training

- **How to optimize the Loss function?**

  Gradient descent:

  Partial derivative

$$\nabla_{\Theta}\mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial \Theta_1}, \frac{\partial \mathcal{L}}{\partial \Theta_2}, \dots\right)$$

  Θ_1, Θ_2 … : components of Θ

- repeatedly update weights in the (opposite) direction of gradients until convergence

$$\Theta \leftarrow \Theta - \eta \nabla_{\Theta}\mathcal{L}$$

- **Learning rate (LR) η:**
  - Hyperparameter that controls the size of gradient step
- **Ideal termination condition: 0** gradient
  - In practice, we stop training if it no longer improves performance on the **validation set** (part of dataset we hold out from training)

Neural Network

- There are much more about NN including:
  - Minibatch Stochastic gradient descent
  - Batch size, Epoch
  - Learning rate scheduling
  - Optimizers to improve over SGD

- However they are not the focus of today's lecture.
- Now that we know fundamentals, let's use NN to learn embeddings

Learning Embeddings using Neural Networks

Agenda

- We will work with three examples:

1. Word embeddings produced by **Word2Vec** model
   - Converts one-hot encoding to dense embedding
   - Task independent and unsupervised
2. Example of task specific embedding
   - converts one-hot encoding to embedding in supervised mode
3. **Autoencoders:**
   - learn short representation from a large feature vector

Word Embedding

- There are many techniques to learn word embeddings:Word2Vec, Glove, BERT, fastText
- **Today's lecture: Word2Vec**
- Word2Vec was Developed at Google in 2013(paper)
- Word2Vec is a statistical method for efficiently learning word embedding. It is 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:

Set a window size (e.g. window=2). For any given word (aka target word), 2 words to its left & 2 words to its right are the context words. Window size is a hyperparameter.

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:

Set a window size (e.g. window=2). For any given word (aka target word), 2 words to its left & 2 words to its right are the context words. Window size is a hyperparameter.

For example, in sentence “I read sci-fi books”:
Target = “I” → context words = “read” , “sci-fi” . No words to the left of “I”.
Target = “read” → context words = “I” , “sci-fi”, “books”

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:

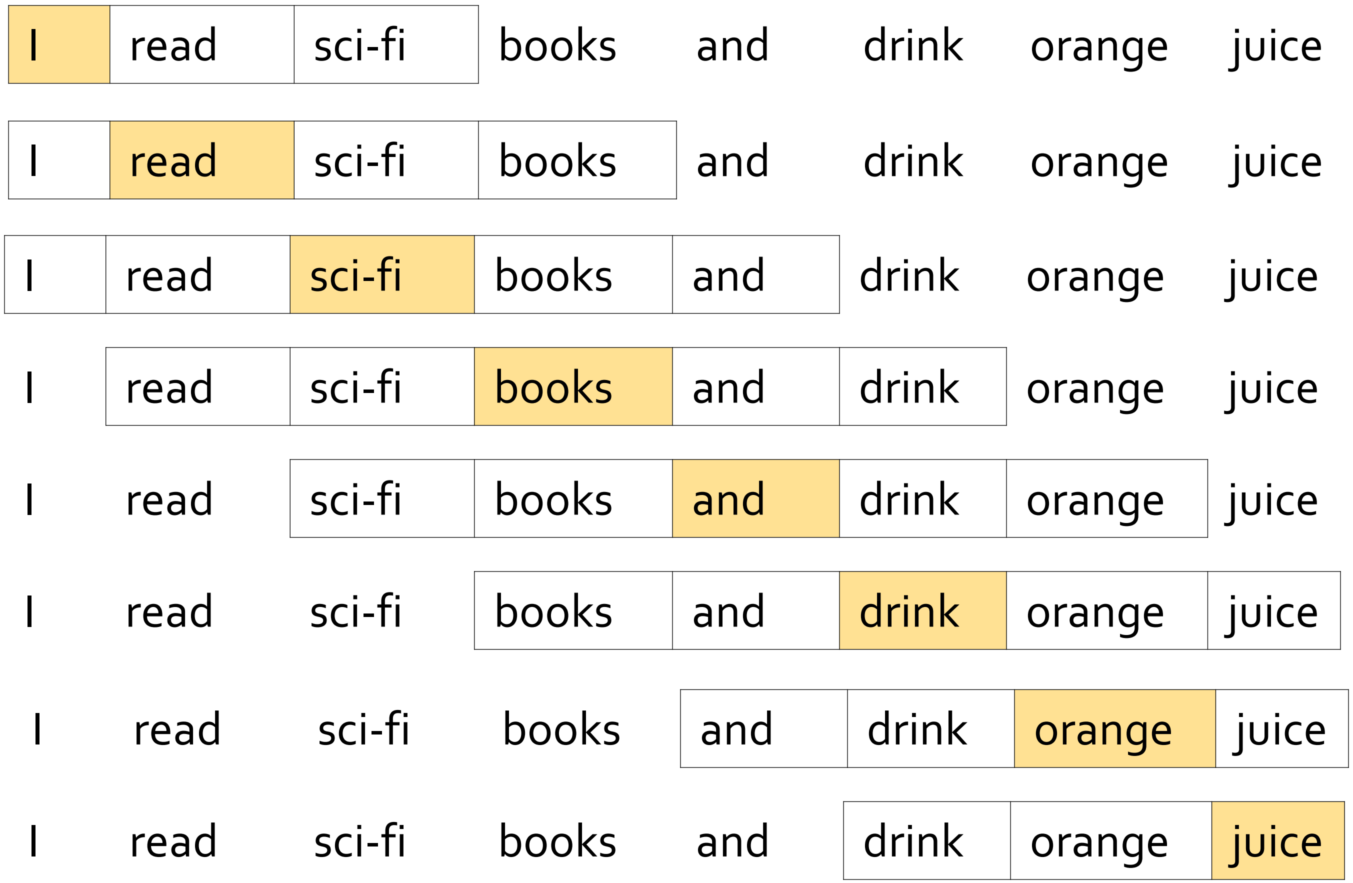
Set a window size (e.g. window=2). For any given word (aka target word), 2 words to its left & 2 words to its right are the context words. Window size is a hyperparameter.

For example, in sentence "I read sci-fi books":
Target = "I" → context words = "read" , "sci-fi" . No words to the left of "I".
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

Window size is a hyper-parameter. Here window size = 2
The highlighted word is the target word. Other words in the box are context words.



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**

Word2Vec: architecture

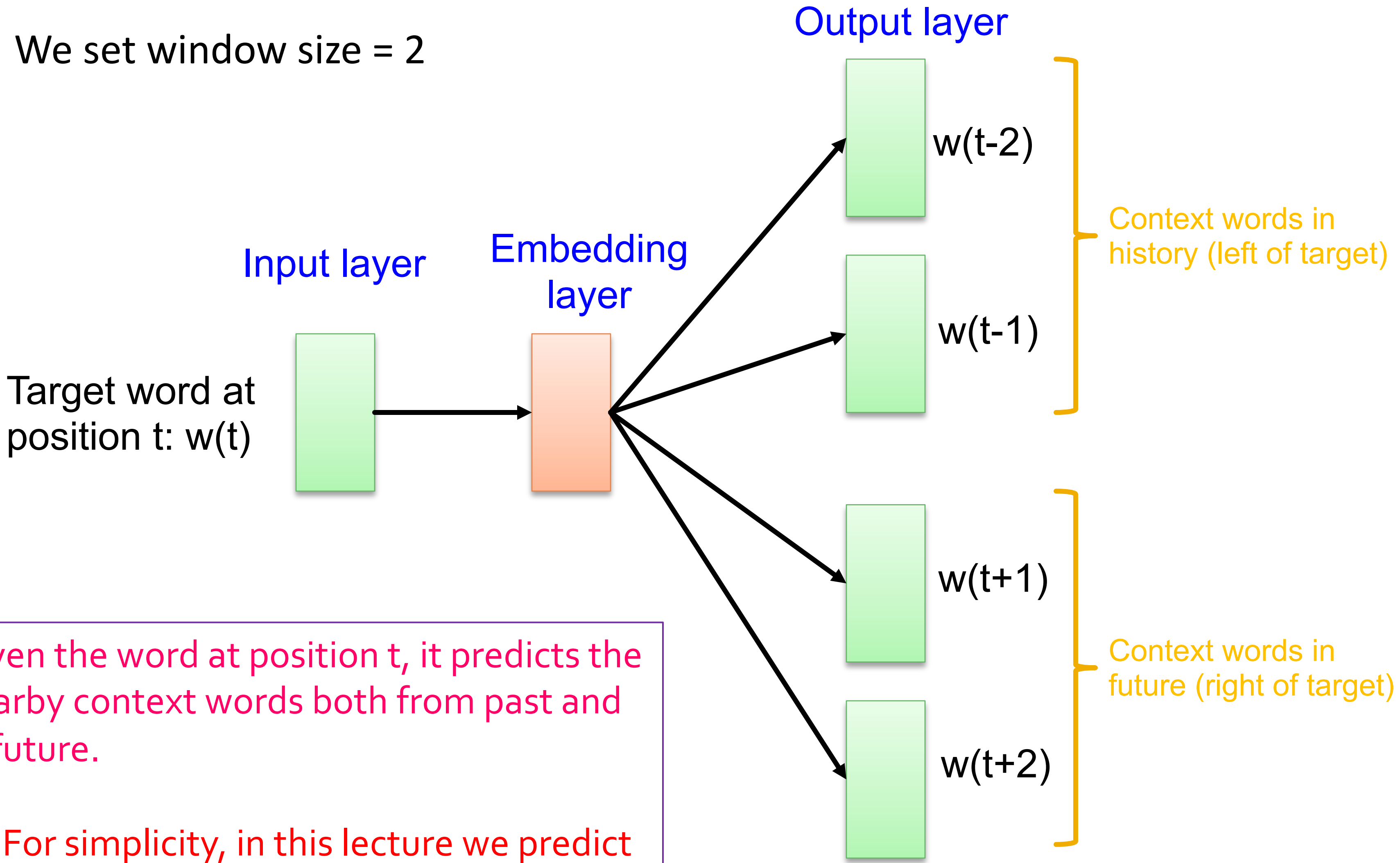
- Word2Vec is a 2 layers NN (i.e. only 1 hidden layer)

- Given one-hot encoding of the target word, it predicts context words
  - In our example, if target word = “I” → context = “read”, “sci-fi”
  - Given one-hot encoding of word “I”, it predicts the context words

- If 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

- We set window size = 2



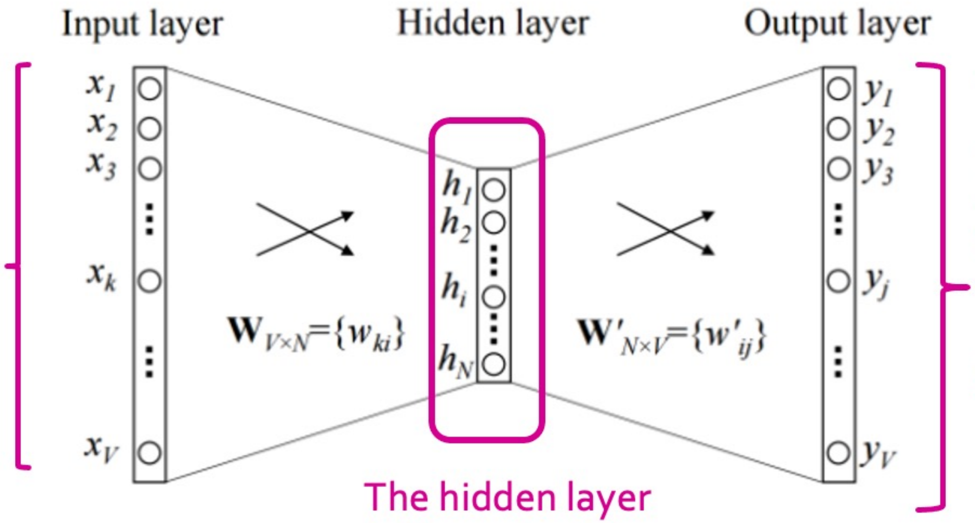
Output layer
Input layer
Embedding layer
Target word at position t: w(t)
w(t-2)
w(t-1)
w(t+1)
w(t+2)
Context words in history (left of target)
Context words in future (right of target)

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

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

Word2Vec: architecture

- Word2Vec is a 2 layers NN (i.e. only 1 hidden layer)
- Let V = size of vocabulary and N = embedding dimension
  - We build vocabulary by extracting 1-gram tokens from corpus



The input is one-hot vector of size V.

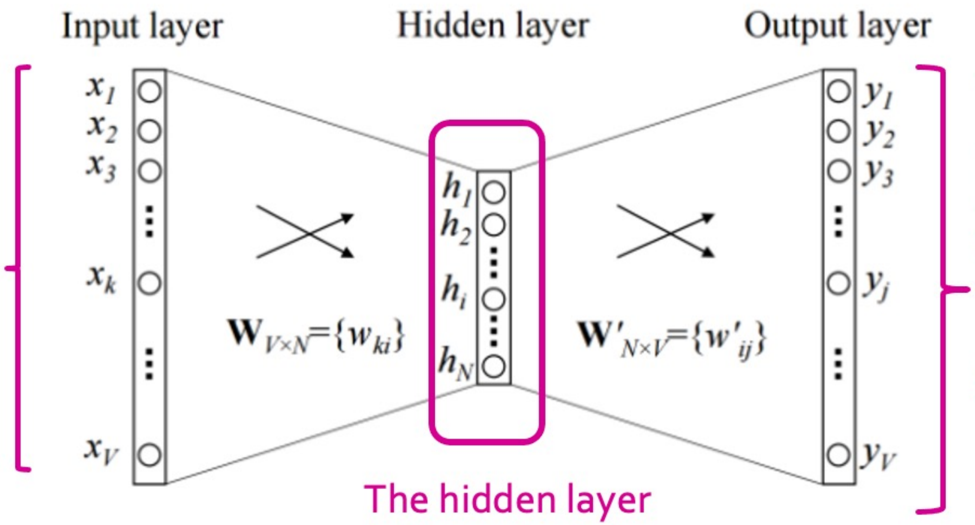
The hidden layer is the embedding layer.

The output is a length V vector with the elements being the softmax values.

Word2Vec: architecture

- Let V = size of vocabulary and N = embedding dimension
- Two different weight matrices:
  - W_{VxN} : from input to hidden layer
  - W'_{NxV} : from hidden to output layer

The input is one-hot vector of size V.

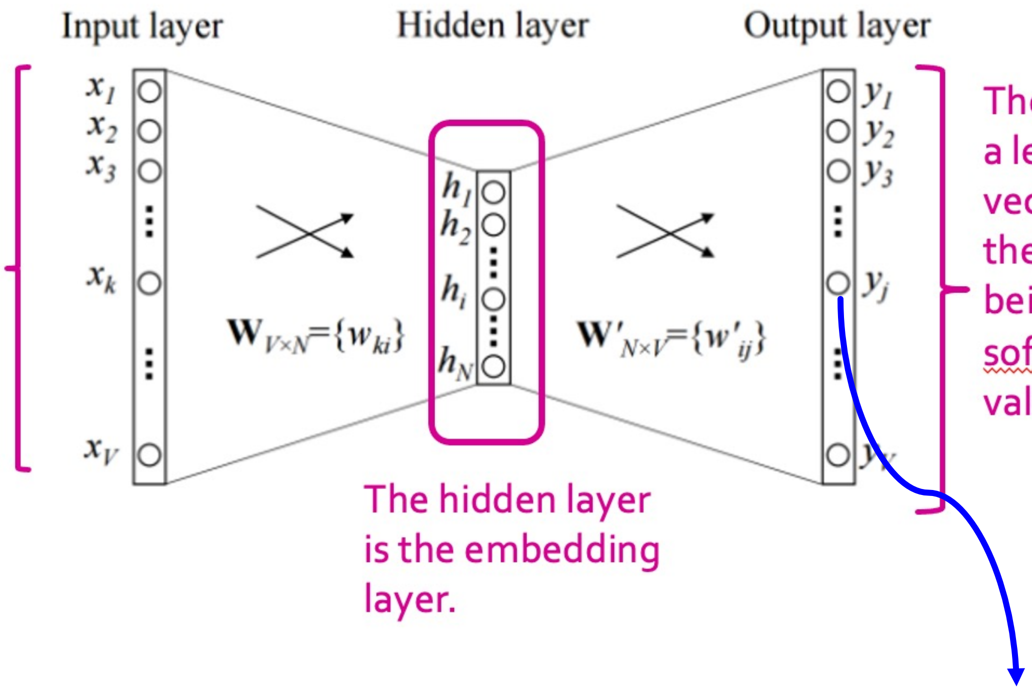


The output is a length V vector with the elements being the softmax values.

The hidden layer is the embedding layer.

Word2Vec: architecture

- Let V = size of vocabulary and N = embedding dimension
- A softmax function is applied on output layer to convert output to probability distribution



The input is one-hot vector of size V.

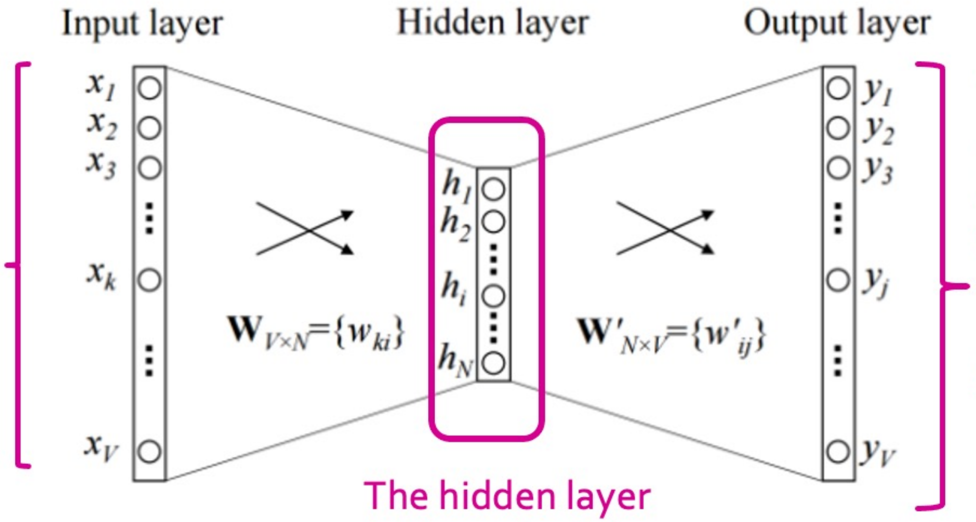
The hidden layer is the embedding layer.

The output is a length V vector with the elements being the softmax values.

Probability that j-th word is in the context

Word2Vec: architecture

- Let V = size of vocabulary and N = embedding dimension
- The hidden layer are all linear neurons, no activation!
- After training the network, embedding of a word is obtained by $x^T W$ i.e. matrix multiplication between word’s one-hot vector and learned weights W_{VxN}



The input is one-hot vector of size V.

The output is a length V vector with the elements being the softmax values.

The hidden layer is the embedding layer.

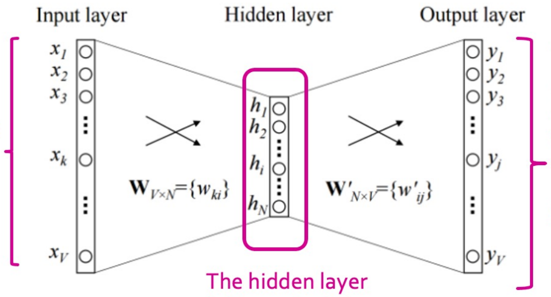
Word2Vec: training the network

- 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?

The input is one-hot vector of size V.

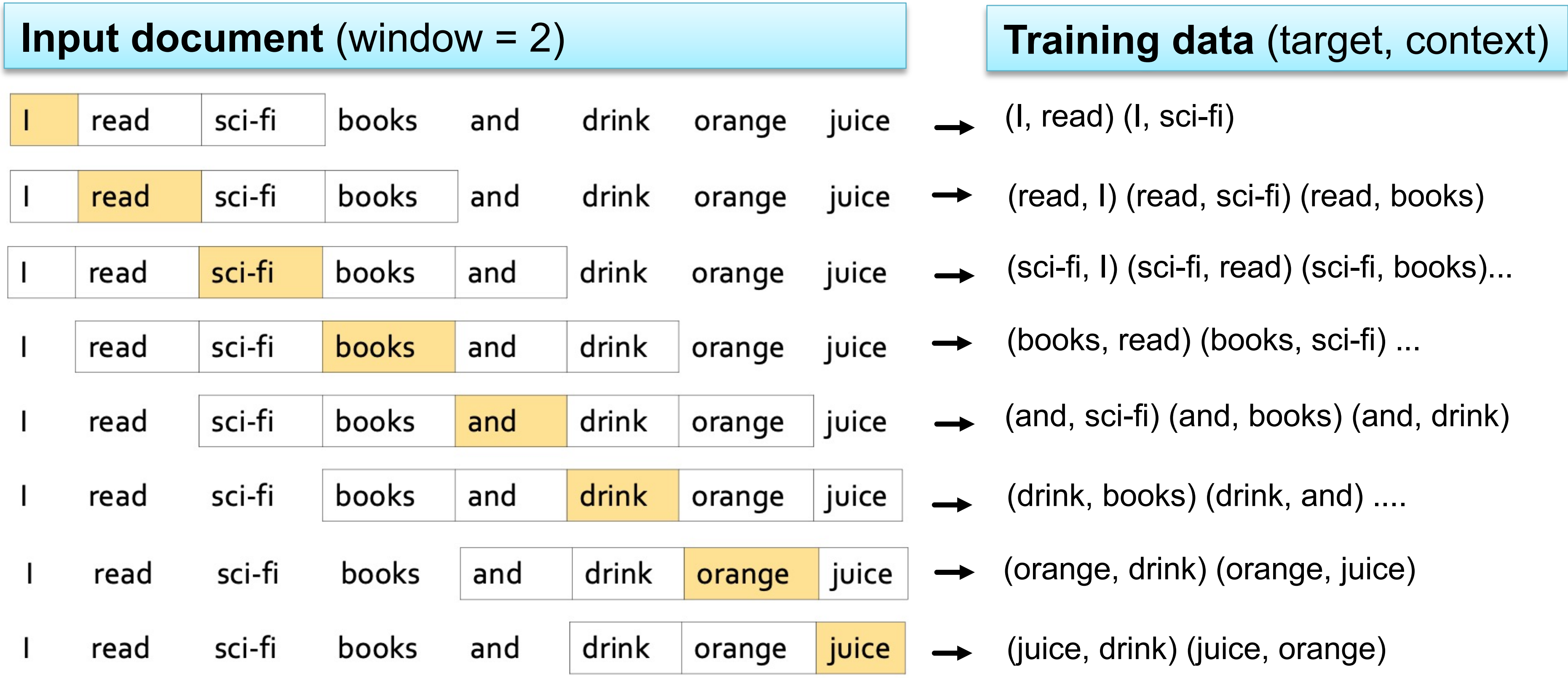


The output is a length V vector with the elements being the softmax values.

The hidden layer is the embedding layer.

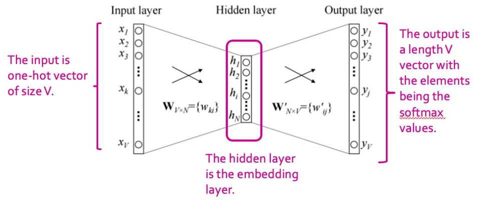
Word2Vec: Training data

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



Input document (window = 2)		Training data (target, context)
I read sci-fi books and drink orange juice	→	(I, read) (I, sci-fi)
I read sci-fi books and drink orange juice	→	(read, I) (read, sci-fi) (read, books)
I read sci-fi books and drink orange juice	→	(sci-fi, I) (sci-fi, read) (sci-fi, books)...
I read sci-fi books and drink orange juice	→	(books, read) (books, sci-fi) ...
I read sci-fi books and drink orange juice	→	(and, sci-fi) (and, books) (and, drink)
I read sci-fi books and drink orange juice	→	(drink, books) (drink, and) ....
I read sci-fi books and drink orange juice	→	(orange, drink) (orange, juice)
I read sci-fi books and drink orange juice	→	(juice, drink) (juice, orange)

Word2Vec: Loss function



The input is one-hot vector of size V.

The hidden layer is the embedding layer.

The output is a length V vector with the elements being the softmax values.

- Given the topology of the network, if ***x*** is the input and ***y*** is the output, then

$$output = y = softmax(W'^T W^T x)$$

- 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

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. Given two probability distributions ***p*** and ***q***, it is defined:

$$\text{CE}(w_c, y) = -\log(ycorr_{ect\ class})$$

Position j

- Since w_c = [0,0,0,...,1,0,0,0..,0]

And y = [0.02, 0.11,.... 0.8, 0, 0.031, ...]

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

Word2Vec: Backpropagation

- Now that loss function is clear, we want to find the values of ***W*** and ***W'*** that minimize it.
  - We want our model to *learn* the weights.

- We use gradient descent to tackle this
  - We find derivatives $\partial L/\partial W$ and $\partial L/\partial W'$ and update weights as $W_{new} = Wo_{ld} - \mu\, \partial L/\partial W$

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

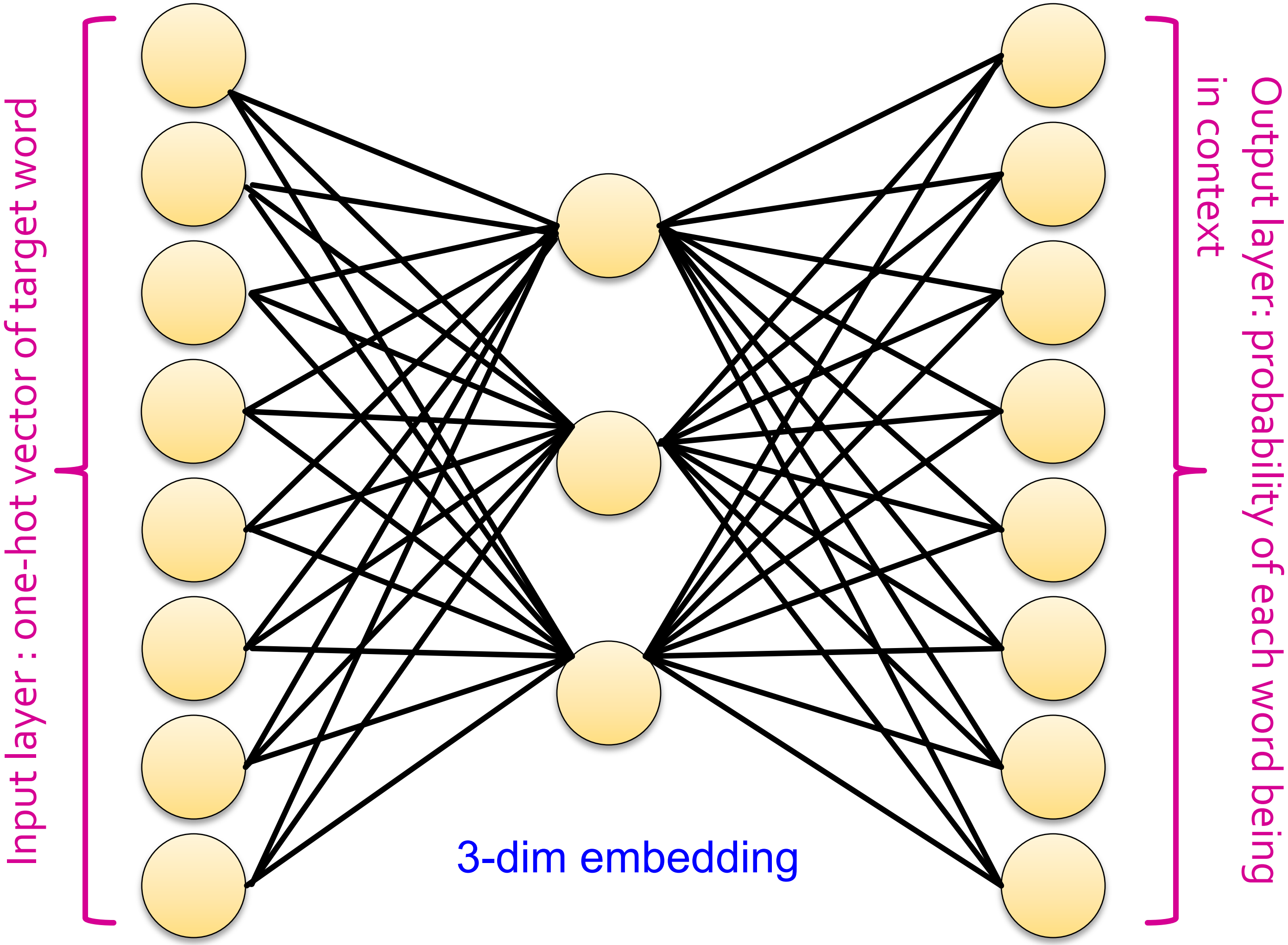
- We execute one forward pass using above document

- 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


Input layer : one-hot vector of target word
3-dim embedding
Output layer: probability of each word being in context

Word2Vec: Example

- If target word = books and weight matrix W_{VxN} be as following:

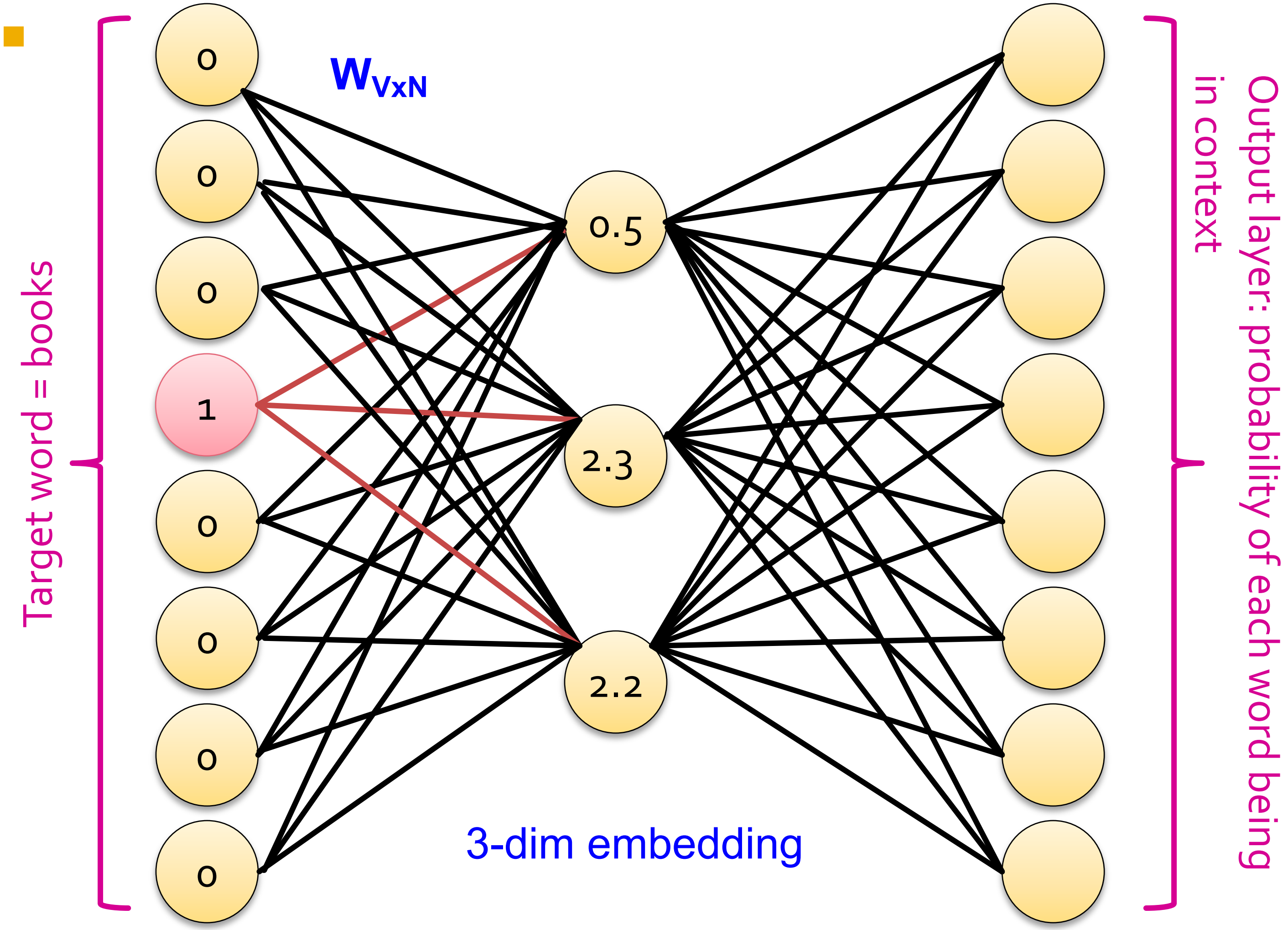
$$W_{V\times N} = \begin{bmatrix} 1 & 2 & 2 \\ -1.2 & -3 & -2 \\ 1.2 & 1.1 & 0.5 \\ 0.5 & 2.3 & 2 \\ -1.1 & 0.6 & -1 \\ 1 & -1 & 2 \\ 0.3 & 1.2 & 0.7 \end{bmatrix}$$

- 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_{VxN} given:
-



Word2Vec: Example

- If the weight matrix W’$_{NxV}$ be as following:

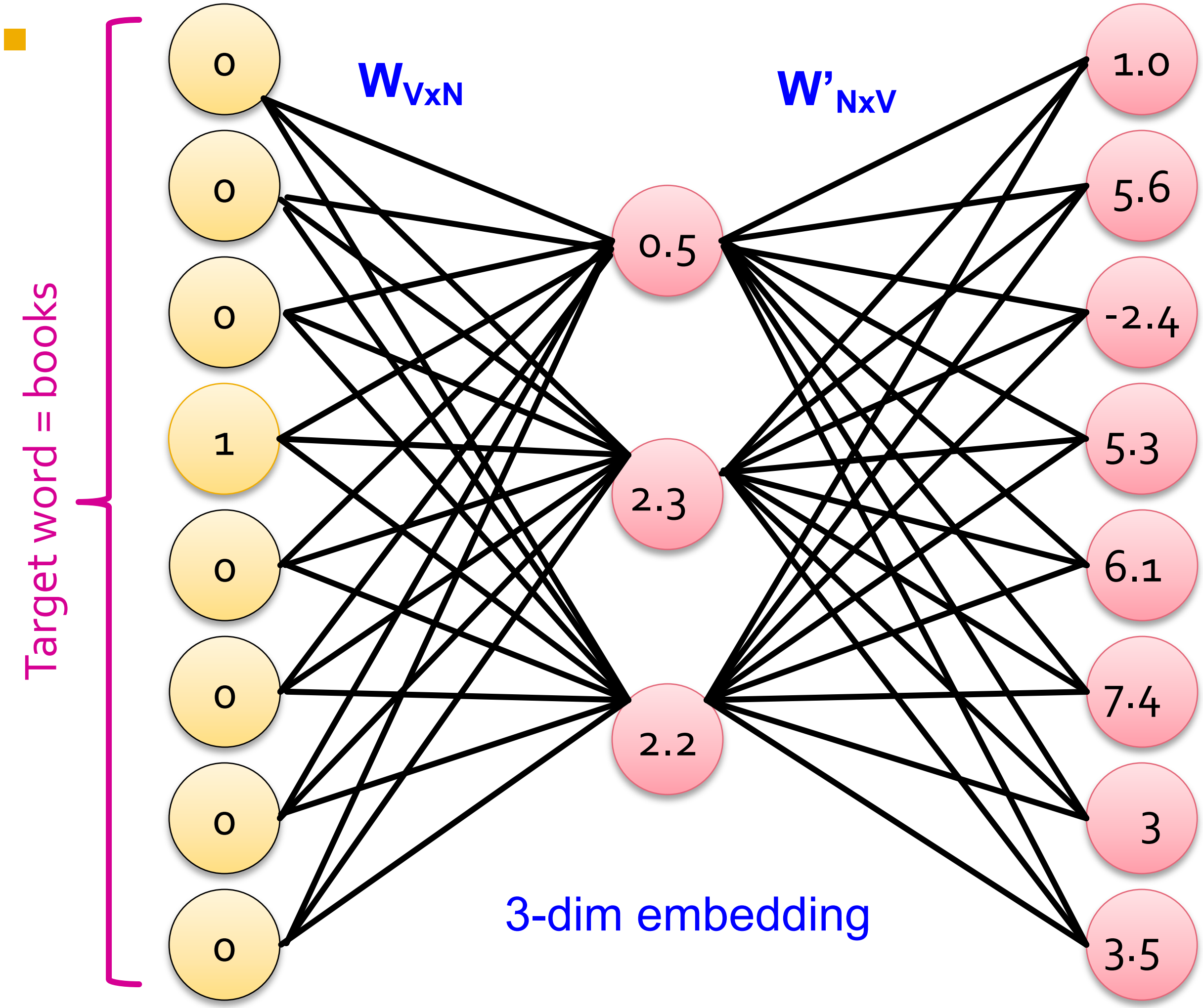
$$W_{N\times V} = \begin{bmatrix} 1 & 2 & 2 & 0 & 0.7 & 1.3 & -1 & -0.1 \\ 1.2 & 0.5 & -1 & 1 & 0.3 & 2 & .6 & 1 \\ -1 & 1.6 & -0.5 & 1.4 & 2.3 & 1 & 1 & 0.6 \end{bmatrix}$$

- Then

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

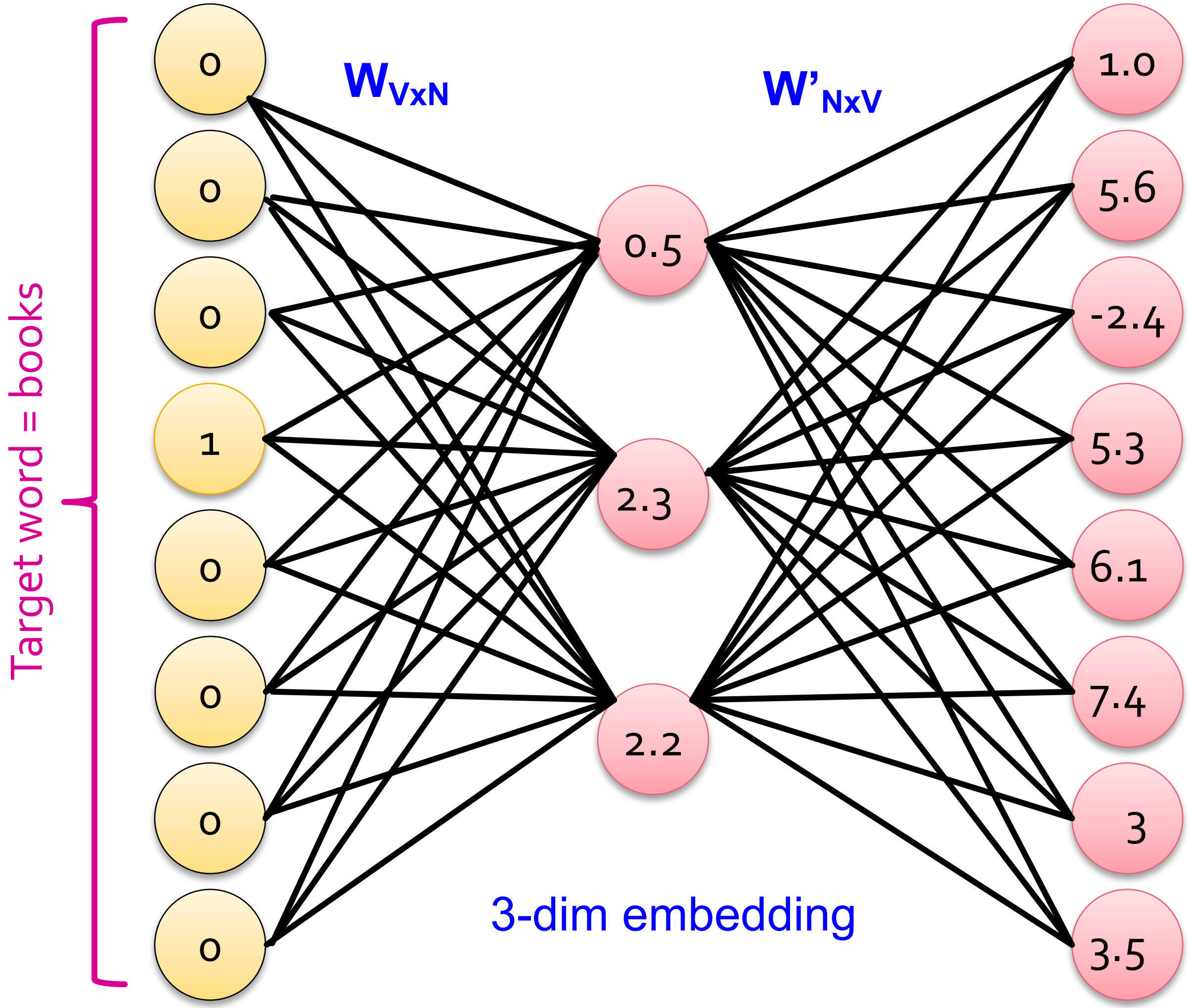
Word2Vec: Example

- If W'_{NxV} given:
-



Word2Vec: Example

- If W'_{NxV} given:

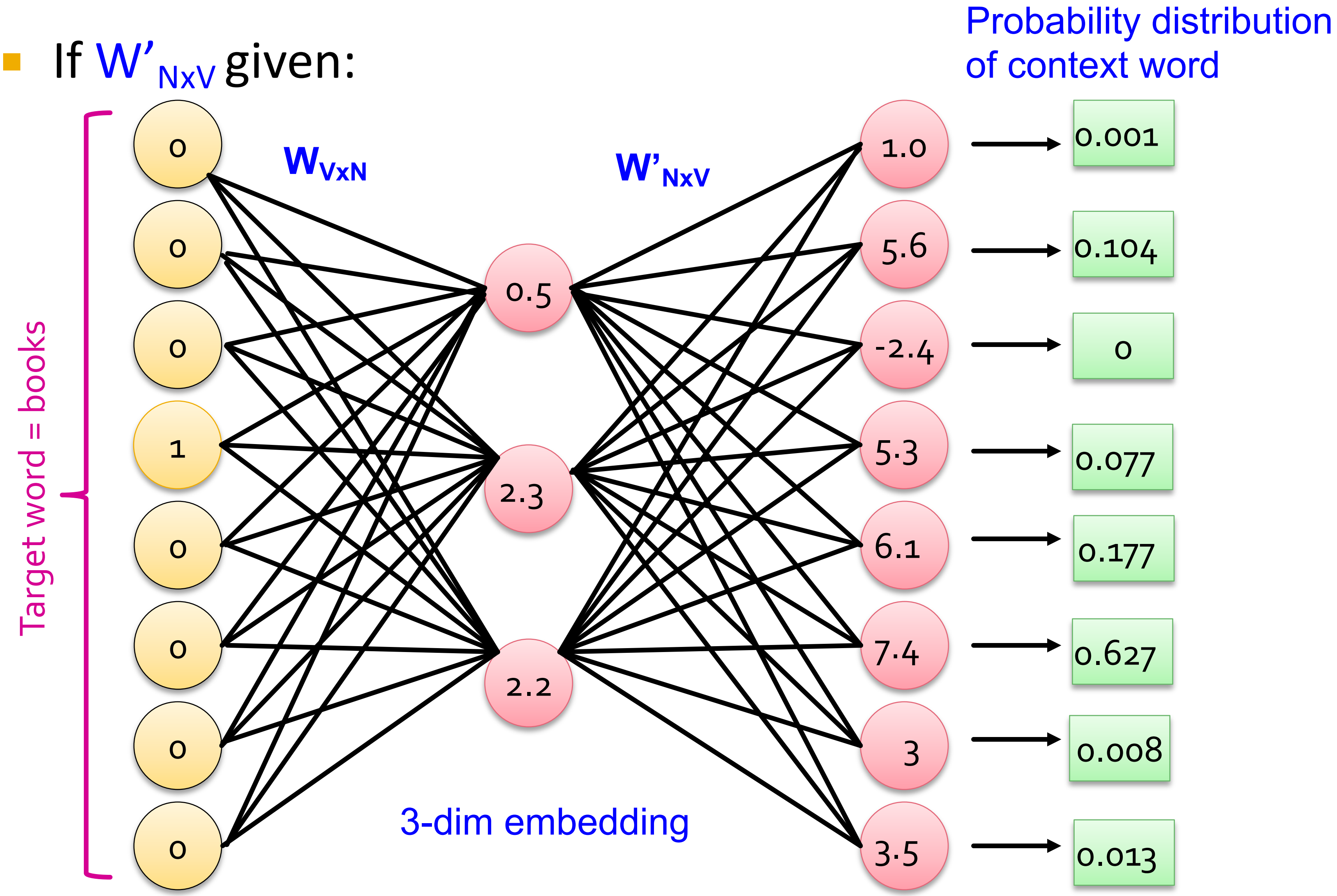


We apply softmax functions to turn them into probabilities.

Each output node x = $\frac{e^x}{\sum e^x}$

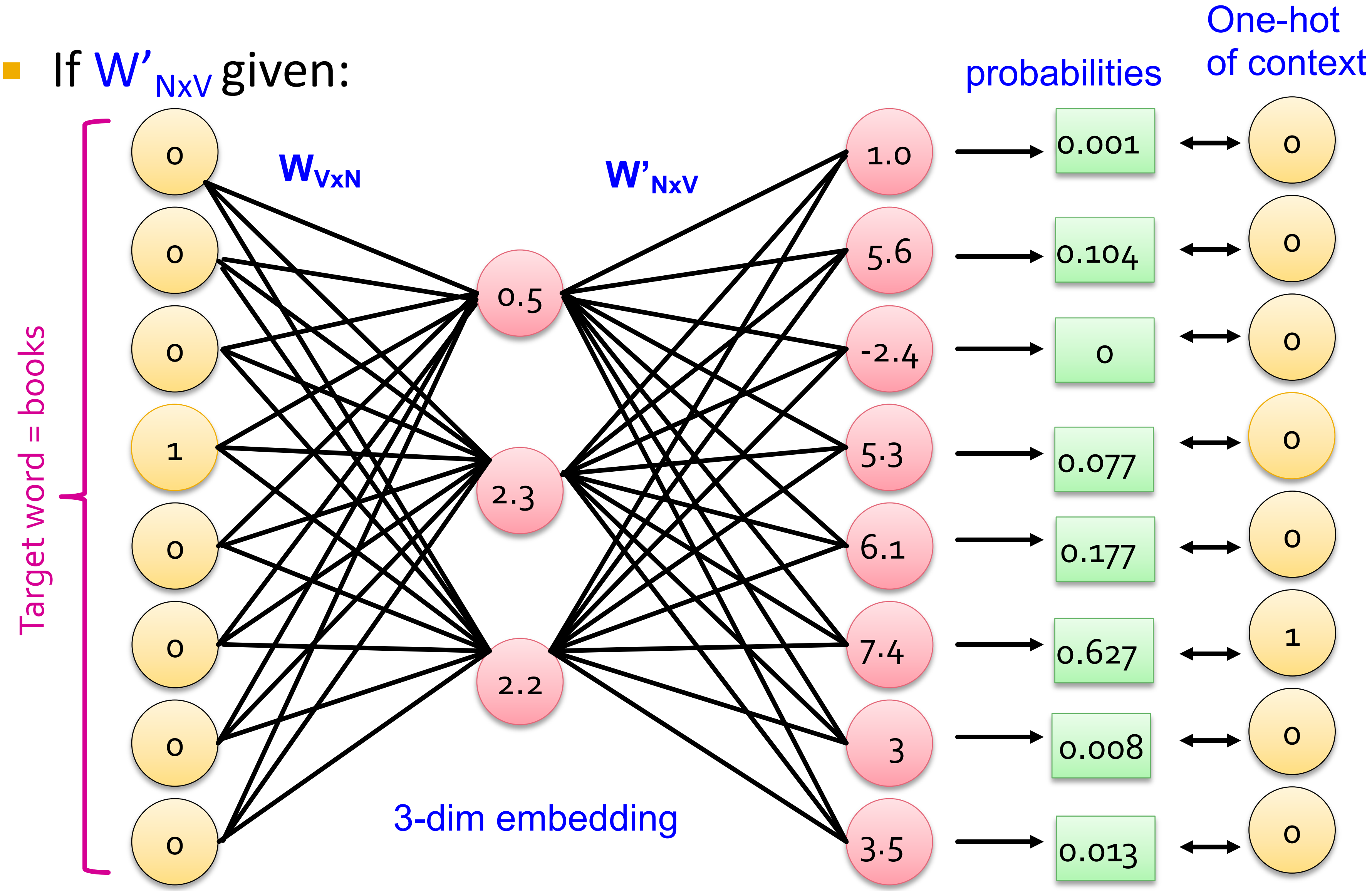
Word2Vec: Example

- If W'$_{NxV}$ given:



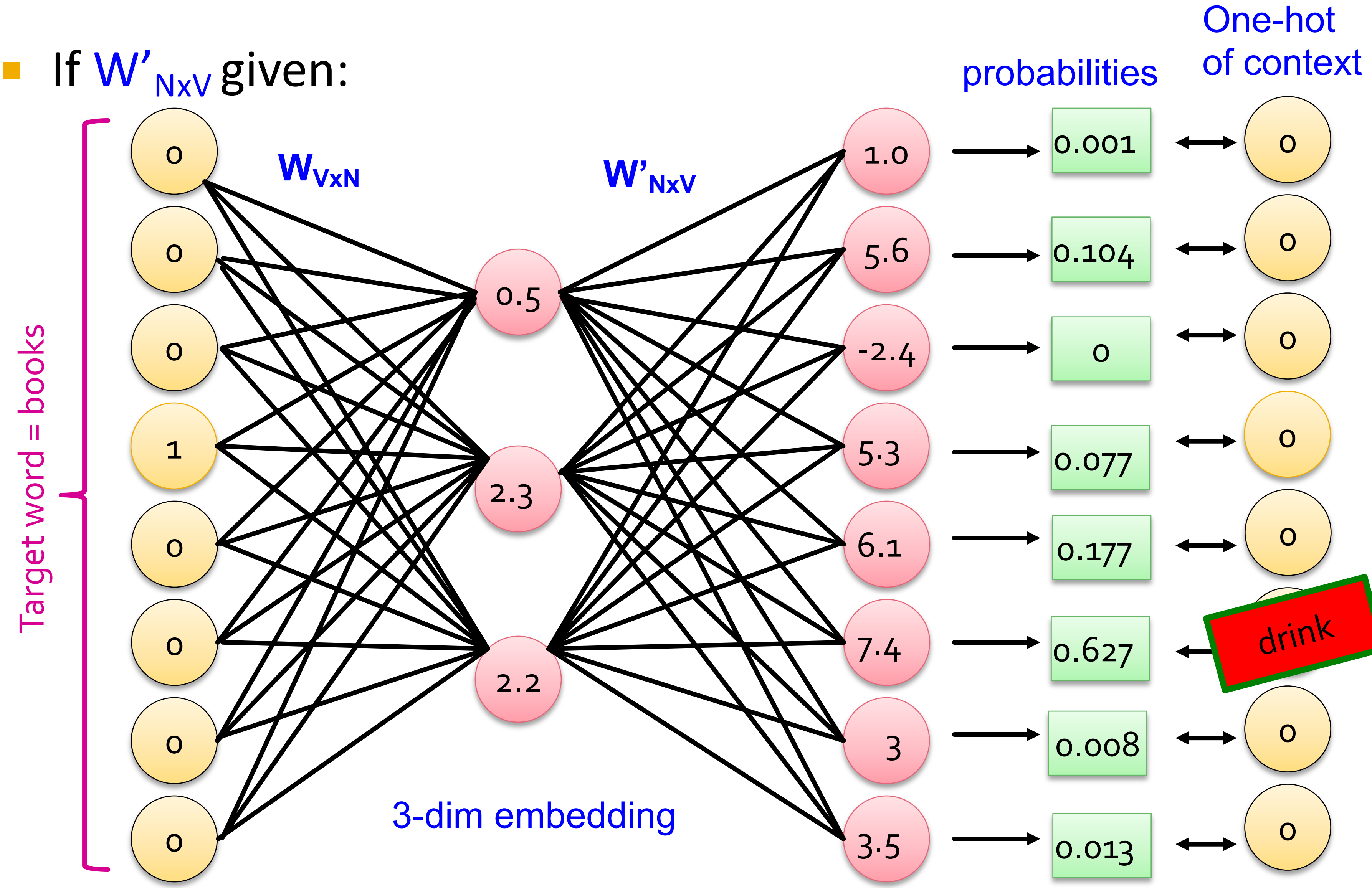
Word2Vec: Example

- If W'_{NxV} given:



Word2Vec: Example

- If W'_{NxV} given:



Word2Vec: Example

- 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]

Then

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

Word2Vec: Summary

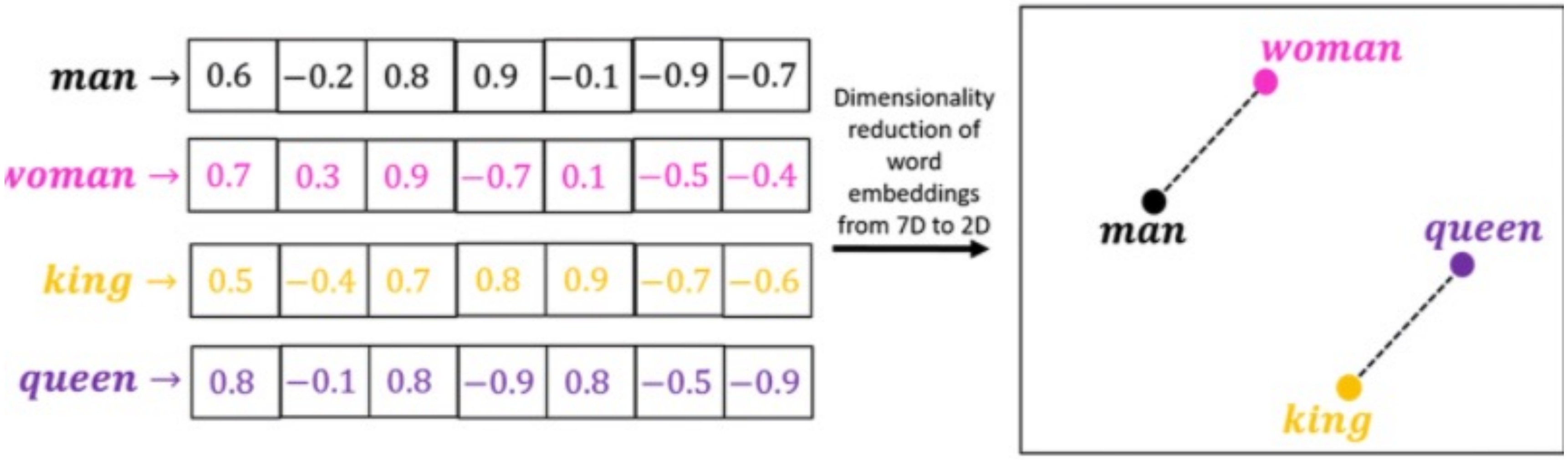
- Word2Vec comes in two architecture:
  - CBOW: Given context words, it predicts target word
  - Skip-Gram: Given target word, predicts context words

- 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 << size of the vocabulary



man →	0.6	−0.2	0.8	0.9	−0.1	−0.9	−0.7
woman →	0.7	0.3	0.9	−0.7	0.1	−0.5	−0.4
king →	0.5	−0.4	0.7	0.8	0.9	−0.7	−0.6
queen →	0.8	−0.1	0.8	−0.9	0.8	−0.5	−0.9

Dimensionality reduction of word embeddings from 7D to 2D

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

Task Specific Embedding

Embedding Layer

- We can learn an embedding as part of a neural network for a given task, e.g. prediction task
  - Pros: produces an embedding customized for a particular task
  - Cons: takes longer than training the embedding separately
- Embedding layer is a regular hidden layer of size *d* (i.e *d* hidden units)
- We combine the embedding layer with any other features needed for our specific task

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

split →

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

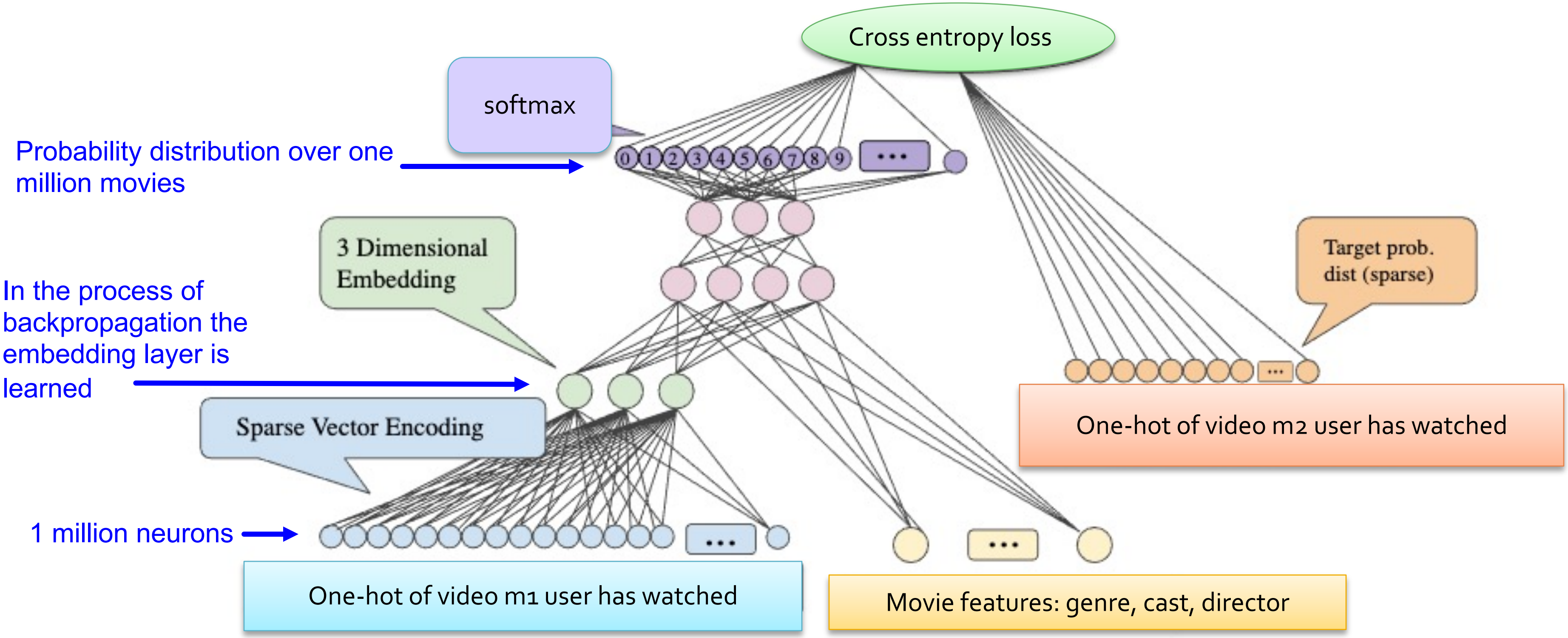
Train	Test
Alice -> m1, m4, m5	Alice -> m3, m2
Bob -> m8, m9	Bob -> m21
Sam -> m6, m10	Sam -> m2

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 collaborative filtering while learning 3-dim embeddings


Cross entropy loss
softmax
Probability distribution over one million movies
0
1
2
3
4
5
6
7
8
9
...
3 Dimensional Embedding
Target prob. dist (sparse)
In the process of backpropagation the embedding layer is learned
...
One-hot of video m2 user has watched
Sparse Vector Encoding
1 million neurons
...
...
One-hot of video m1 user has watched
Movie features: genre, cast, director

Example: Recommending movies

- How to recommend a movie to a user e.g. Alice?
  - Alice has watched m1, m4, m5 in train
  - Find movies that have similar embeddings to m1
    - Similarity score = <emb(m1), emb(v)> for any movie v
    - Find top 5 movies with highest similarity score
    - Recommend them to Alice
  - Or even better: repeat above for m1, m4 and m5
    - 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

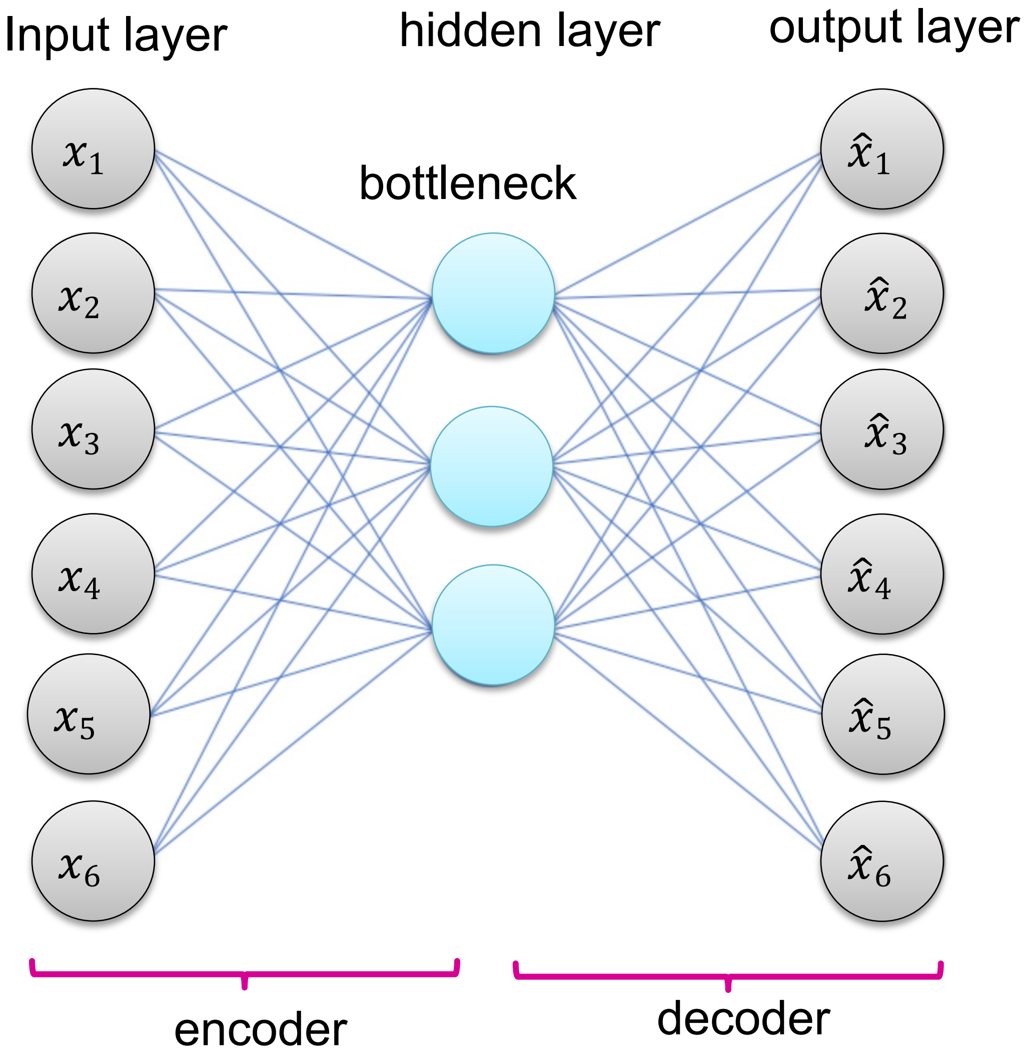
- It learns to compress the data while minimizing the *reconstruction error*.

Autoencoder

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

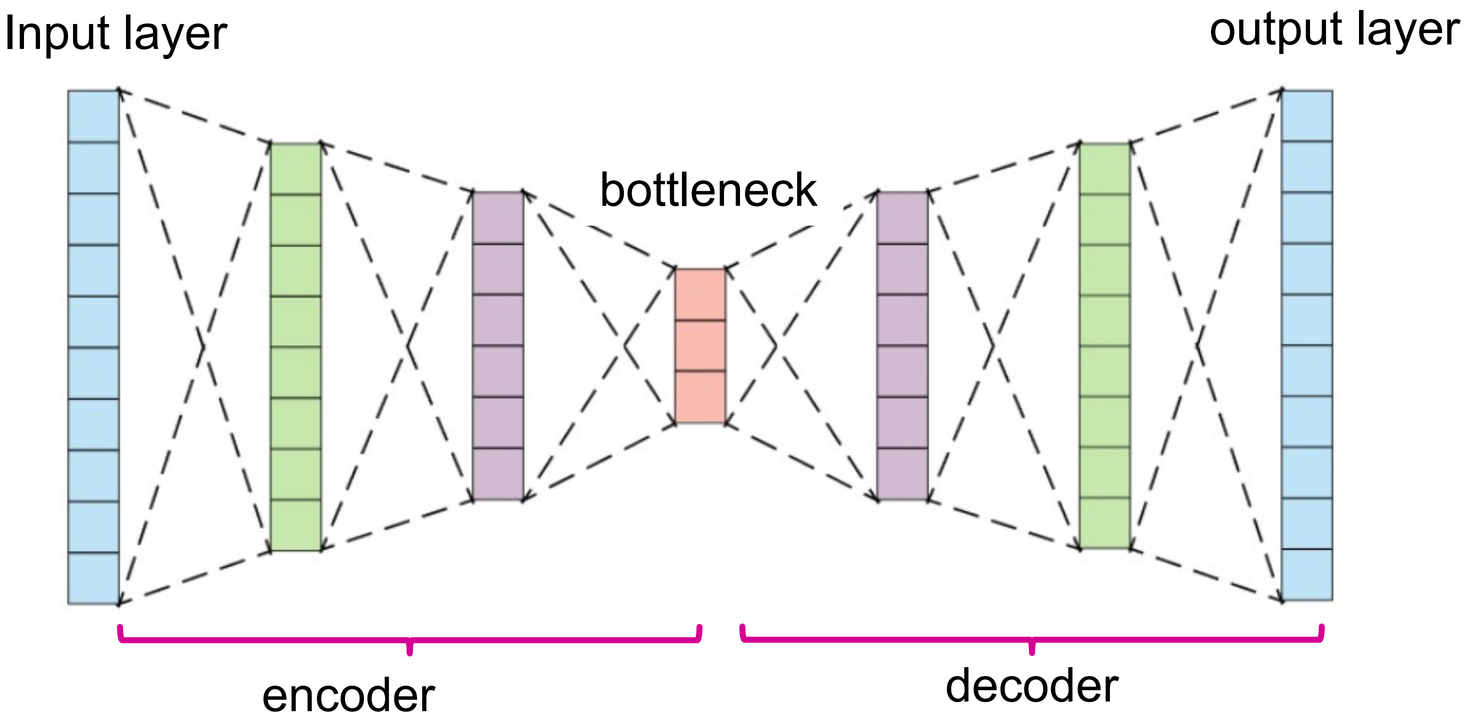
bottleneck layer: is the bottleneck as it projects down 6-dim vector to 3-dim space. It constrains the amount of information that traverses the network

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.



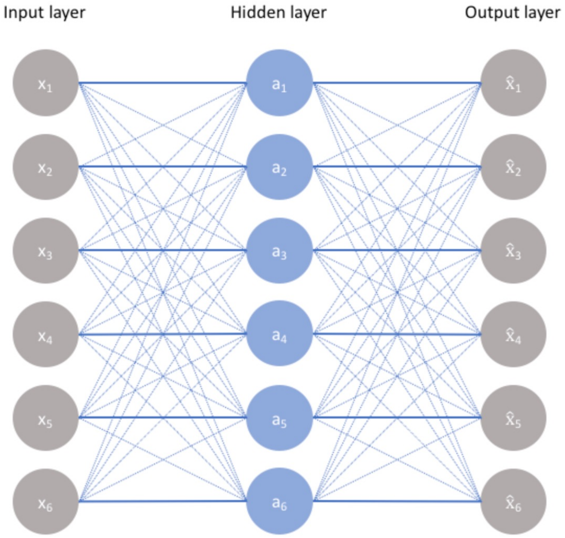
Autoencoder

Two main component in their architecture:

- **Encoder**: a function **f** that compresses the input into a latent-space representation
  - f(x) = h such that dimension(h) < dimension(x)
- **Decoder**: a function **g** that reconstruct the input from the latent space representation
  - g(h) ~ 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

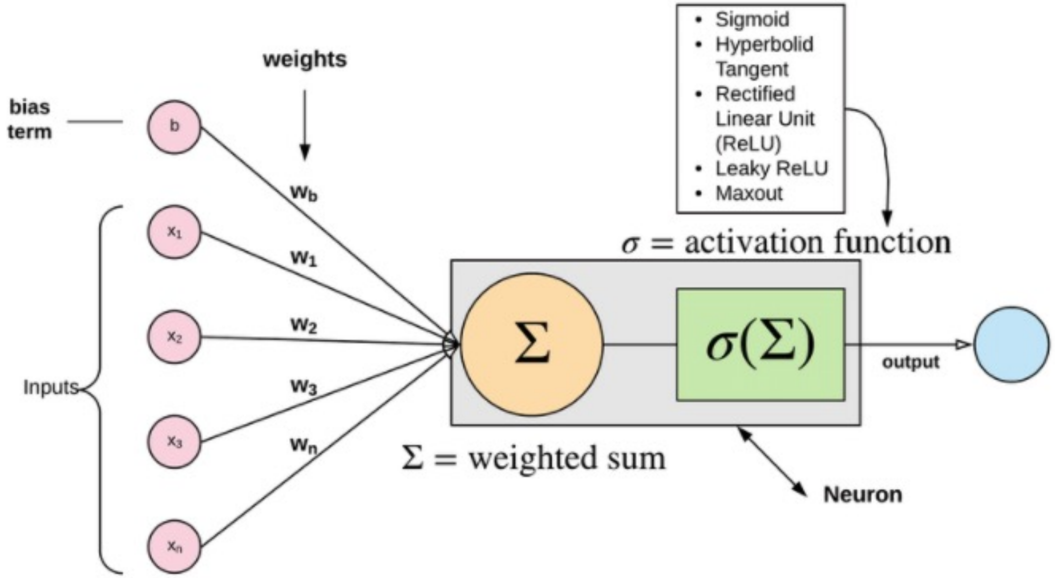
- The loss function to train an undercomplete AE is *reconstruction loss:*

$$L(x, \hat{x}) = \|x - \hat{x}\|_1$$

- 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.

Today's lecture

- 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