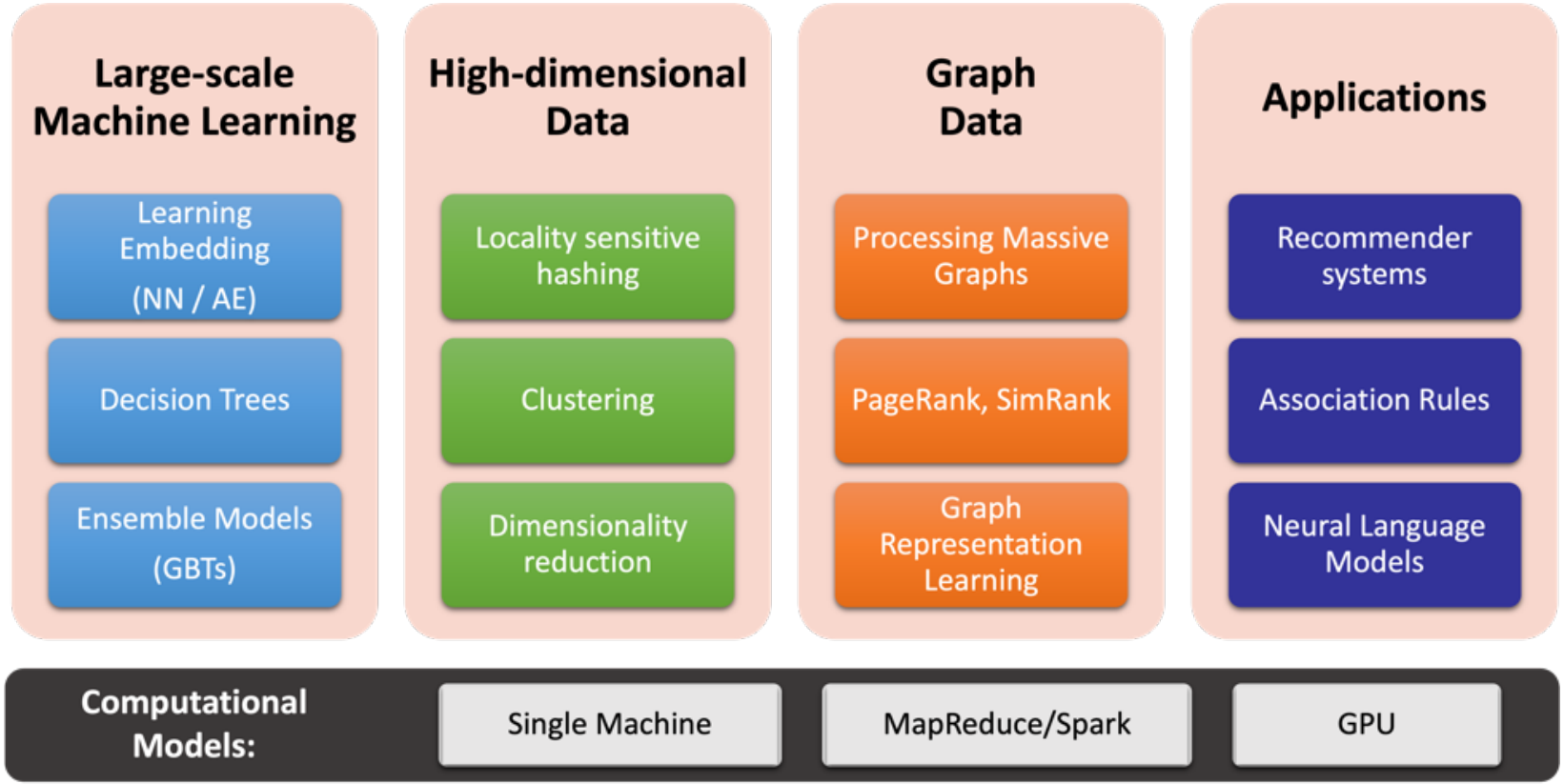
MIE524 Data Mining

Neural Networks (Part 2)

Slides Credits:
Slides from Leskovec, Rajaraman, Ullman (http://www.mmds.org),
Dan Jurafsky & James H. Martin (https://web.stanford.edu/~jurafsky/slp3/),
Roger Grosse, Stephen Scott, Geoffrey Hinton, Arun Kumar, Leonid Sigal, Vagelis Hristidis

MIE524: Course Topics (Tentative)



Agenda

- Motivation: why embeddings?
- Neural network fundamentals
- Training Neural Networks
- Computation Graphs and Backward Differentiation
- **Why is this scalable?**
- The XOR problem
- Learning embeddings: Autoencoders

Using slides by:
Jure Leskovec & Mina Ghashami, Dan Jurafsky & James H. Martin,
Roger Grosse, Stephen Scott, Geoffrey Hinton

Scalable ML Systems

- ML systems that do *not* require the (training) dataset to fit entirely in main memory (DRAM) of one node
  - Conversely, if the system *thrashes* when data file does not fit in RAM, it is not scalable

Basic Idea: Split data file (virtually or physically) and *stage reads* (and writes) of pages to DRAM and processor

Scalable ML Systems

4 main approaches to scale ML to large data:

- **Single-node disk**: Paged access from file on local disk
- **Remote read**: Paged access from disk(s) over network
- **Distributed memory**: Fits on a cluster's total DRAM
- **Distributed disk**: Fits on a cluster's full set of disks

ML Algorithm = Program Over Data

Basic Idea: Split data file (virtually or physically) and *stage reads* (and writes) of pages to DRAM and processor

- To scale an ML program's computations, split them up to operate over "chunks" of data at a time
- *How* to split up an ML program this way can be non-trivial!
  - Depends on *data access pattern* of the algorithm
  - A large class of ML algorithms do just *sequential scans* for iterative numerical optimization

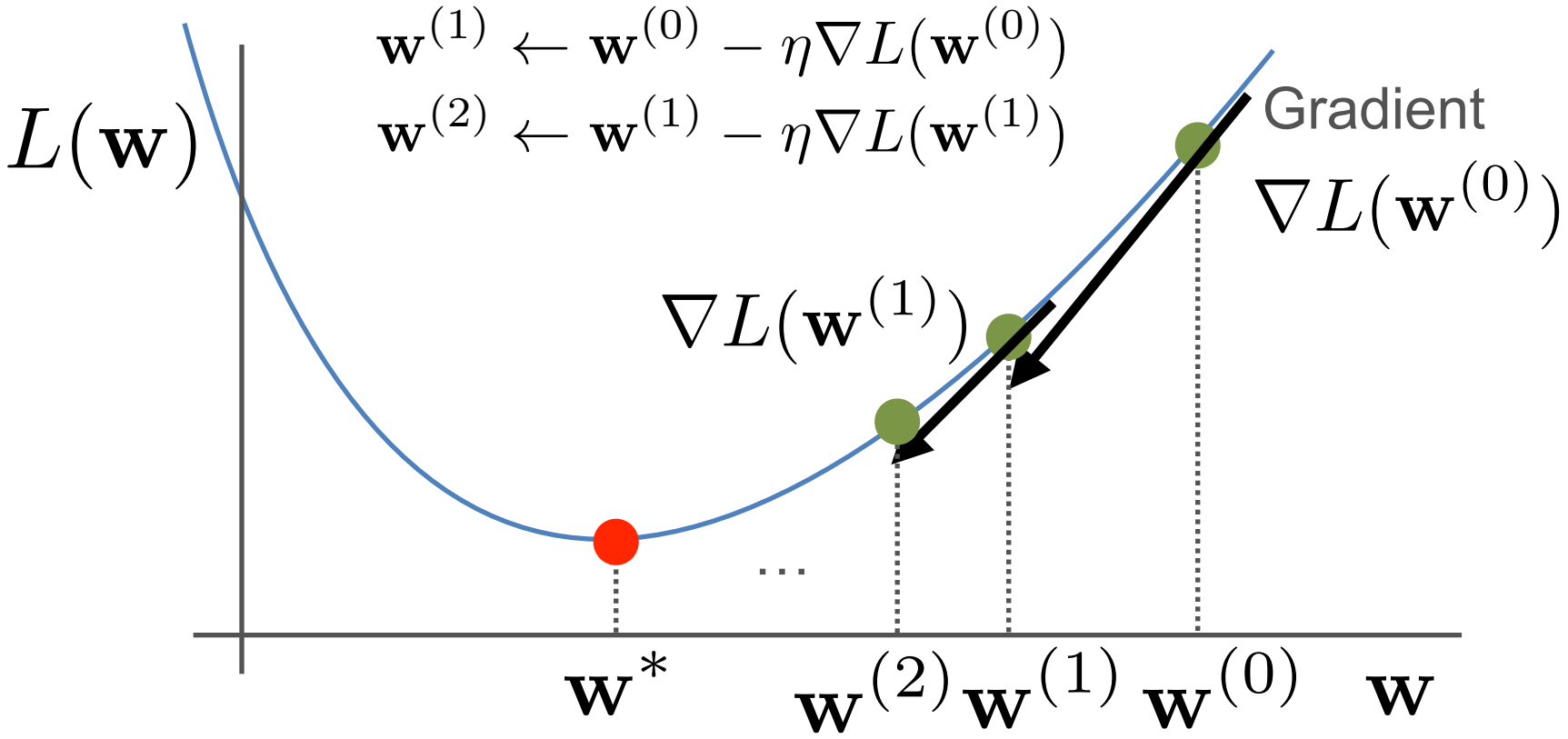
Batch Gradient Descent for ML

$$L(\mathbf{w}) = \sum_{i=1}^{n} l(y_i, f(\mathbf{w}, x_i))$$

- **Batch Gradient Descent:**
  - Iterative numerical procedure to find an optimal **w**
  - Initialize **w** to some value $\mathbf{w}^{(0)}$
  - Compute **gradient**: $\nabla L(\mathbf{w}^{(k)}) = \sum_{i=1}^{n} \nabla l(y_i, f(\mathbf{w}^{(k)}, x_i))$
  - Descend along gradient: (Aka **Update Rule**) $\mathbf{w}^{(k+1)} \leftarrow \mathbf{w}^{(k)} - \eta \nabla L(\mathbf{w}^{(k)})$
  - Repeat until we get close to **w***, aka **convergence**

Slides by Arun Kumar

Batch Gradient Descent for ML



- Learning rate is a **hyper-parameter** selected by user or "AutoML" tuning procedures
- Number of iterations/epochs of BGD also hyper-parameter

Slides by Arun Kumar

Data Access Pattern of BGD at Scale

- The data-intensive computation in BGD is the gradient
  - In scalable ML, dataset D may not fit in DRAM
  - Model **w** is typically small and DRAM-resident

$$\nabla L(\mathbf{w}^{(k)}) = \sum_{i=1}^{n} \nabla l(y_i, f(\mathbf{w}^{(k)}, x_i))$$

- At each epoch, 1 **filescan** over D to get gradient
- Update of **w** happens normally in DRAM
- Monitoring across epochs for convergence needed

Scaling BGD to Disk

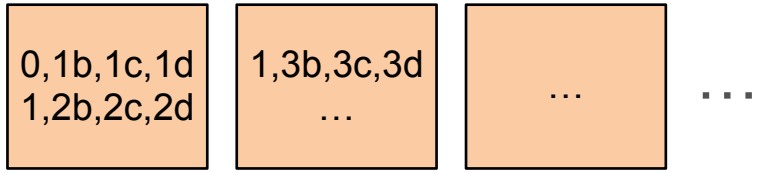
- Sequential scan to read pages from disk to DRAM
  - Modern DRAM sizes can be 10s of GBs; so we read a "chunk" of file at a time (say, 1000s of pages)
  - Compute partial gradient on each chunk and add all up

$$\nabla L(\mathbf{w}^{(k)}) = \sum_{i=1}^{n} \nabla l(y_i, f(\mathbf{w}^{(k)}, x_i))$$

D

Y	X1	X2	X3
0	1b	1c	1d
1	2b	2c	2d
1	3b	3c	3d
0	4b	4c	4d
…	…	…	…

Data pages



$\nabla L_1(w^{(k)})$ $\nabla L_2(w^{(k)})$ …

$$\nabla L(w^{(k)}) = \nabla L_1(w^{(k)}) + \nabla L_2(w^{(k)}) + \ldots$$

Slides by Arun Kumar

Scaling with Remote Reads

Basic Idea: Split data file (virtually or physically) and *stage reads* (and writes) of pages to DRAM and processor

- Similar to scaling to disk but instead read pages/chunks over the network from remote disk/disks (e.g., from S3)
  - Good in practice for a one-shot *filescan* access pattern
  - For iterative ML, repeated reads over network
  - Can combine with caching on local disk / DRAM
- Increasingly popular for cloud-native ML workloads

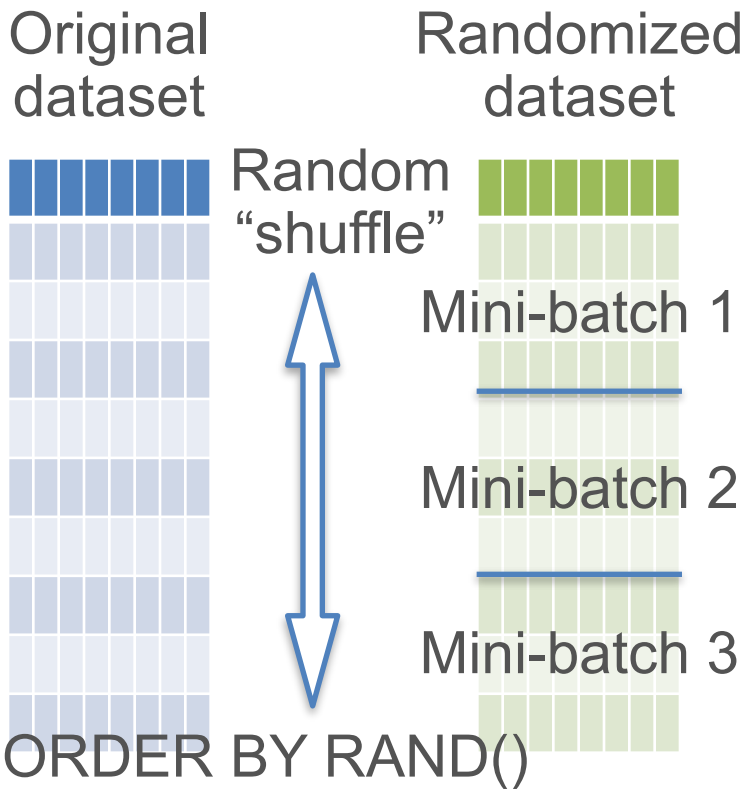
Stochastic Gradient Descent for ML

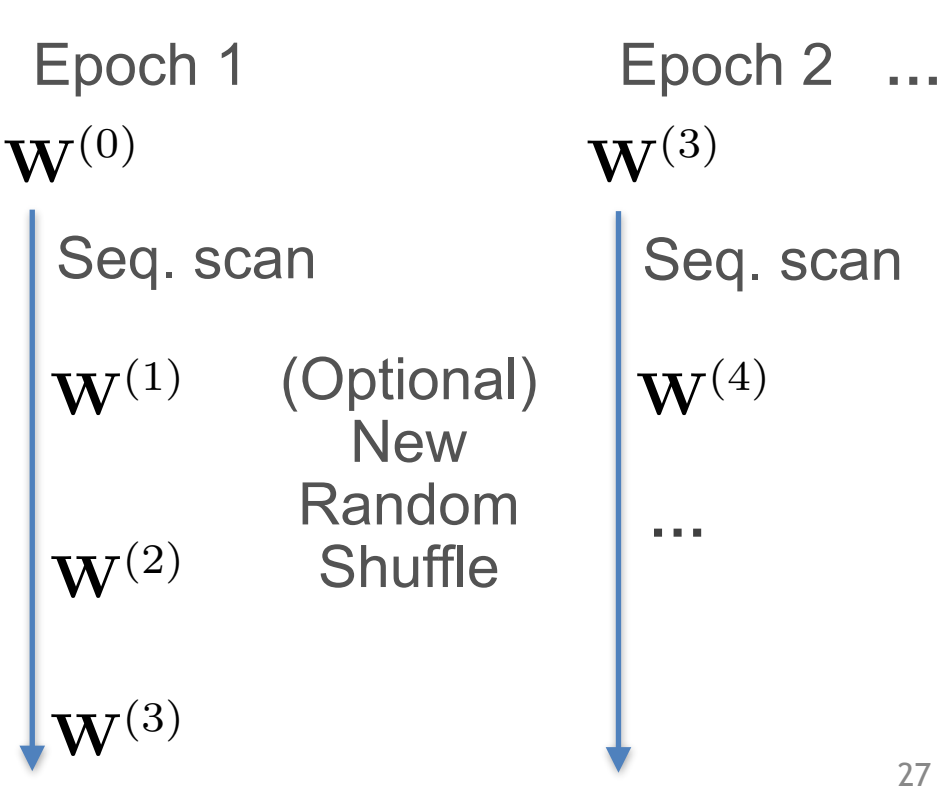
- Two key cons of BGD:
  - Slow to converge to optimal (too many epochs)
  - Costly full scan of D for each update of **w**
- Stochastic GD (SGD) mitigates both issues
- **Basic Idea:** Use a *sample* (called **mini-batch**) of D to approximate gradient instead of "full batch" gradient
  - *Without replacement* sampling
  - Randomly shuffle D before each epoch
  - One pass = sequence of mini-batches
- SGD works well for *non-convex* loss functions too, unlike BGD; "workhorse" of scalable ML

Data Access Pattern of Scalable SGD

$$\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \eta \nabla \tilde{L}(\mathbf{w}^{(t)}) \qquad \nabla \tilde{L}(\mathbf{w}^{(k)}) = \sum_{(y_i, x_i) \in B \subset D} \nabla l(y_i, f(\mathbf{w}^{(k)}, x_i))$$

Sample mini-batch from dataset without replacement





Slides by Arun Kumar

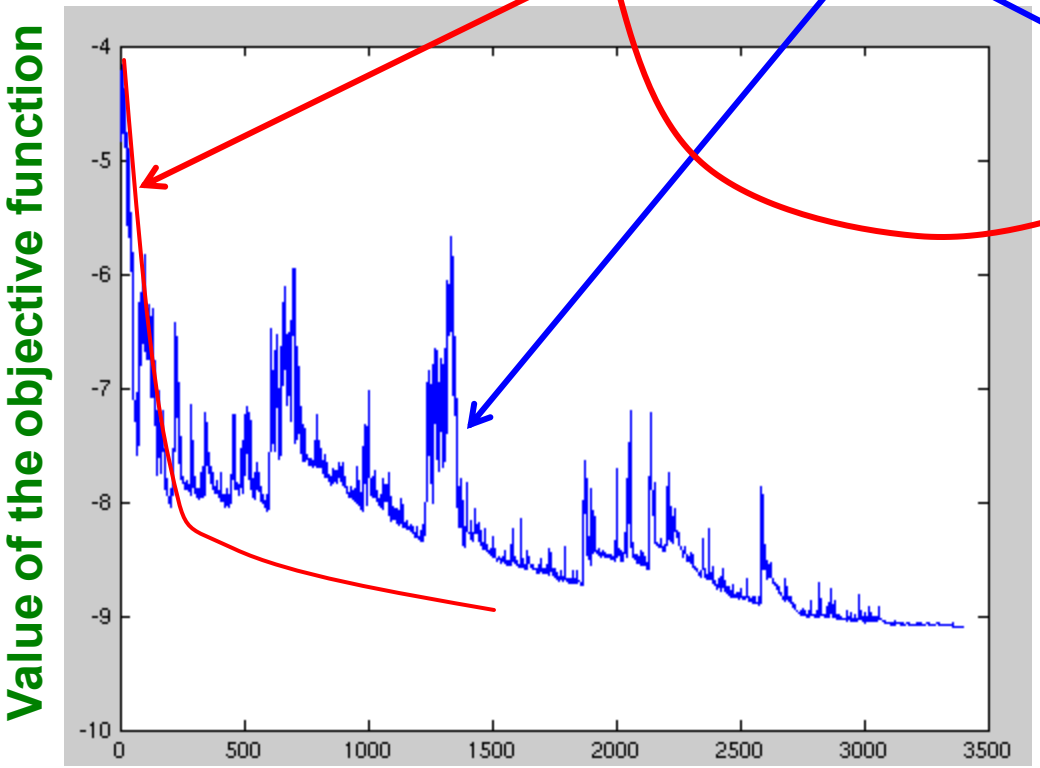
Data Access Pattern of Scalable SGD

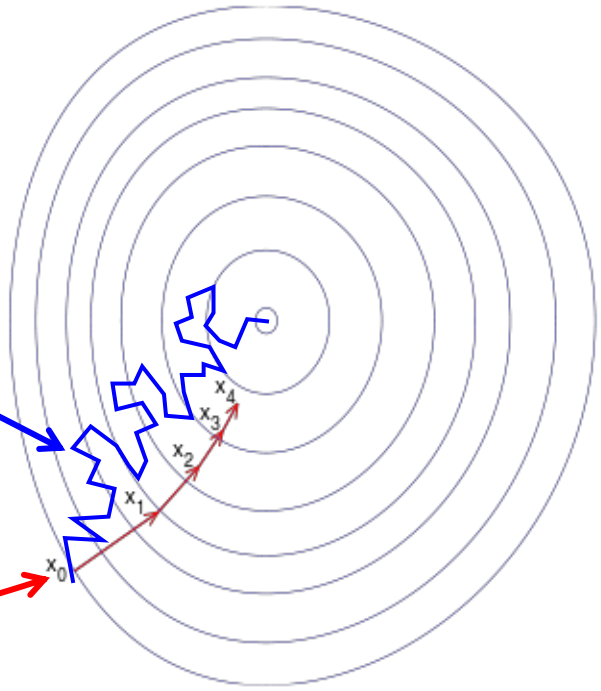
- Mini-batch gradient computations: 1 **filescan** per epoch
- Update of **w** happens in DRAM
- During filescan, count number of examples seen and update per mini-batch
  - Typical Mini-batch sizes: 10s to 1000s
  - Orders of magnitude more updates than BGD!
- Random shuffle is not trivial to scale; requires "external merge sort" (roughly 2 scans of file)
  - ML practitioners often shuffle dataset *only once up front*; good enough in most cases in practice

Slides by Arun Kumar

SGD vs. GD

- **Convergence of GD vs. SGD**





GD improves the value of the objective function at every step.
SGD improves the value but in a "noisy" way.
GD takes fewer steps to converge but each step takes much longer to compute.
In practice, **SGD** is much faster!

Agenda

- Motivation: why embeddings?
- Neural network fundamentals
- Training Neural Networks
- Computation Graphs and Backward Differentiation
- Why is this scalable?
- **The XOR problem**
- Learning embeddings: Autoencoders

Using slides by:
Jure Leskovec & Mina Ghashami, Dan Jurafsky & James H. Martin,
Roger Grosse, Stephen Scott, Geoffrey Hinton

Simple Neural Networks and Neural Language Models

The XOR problem

Slides by Dan Jurafsky and James H. Martin

The XOR problem

Minsky and Papert (1969)

Can neural units compute simple functions of input?

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

Perceptrons

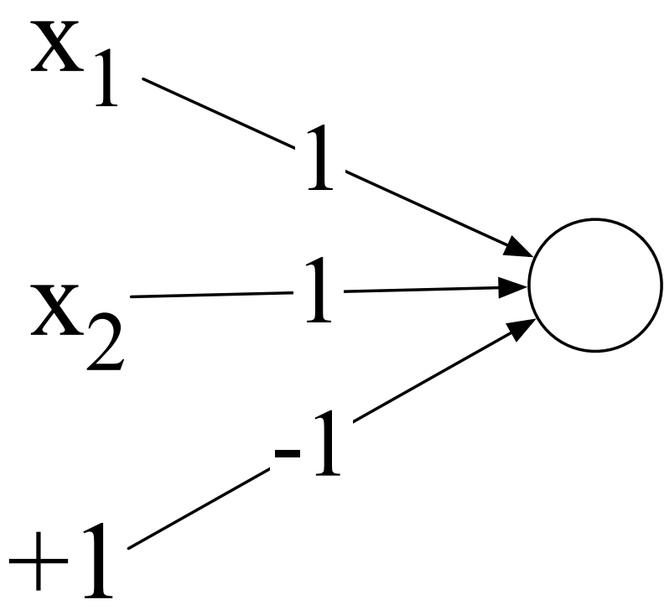
A very simple neural unit

- Binary output (0 or 1)
- No non-linear activation function

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

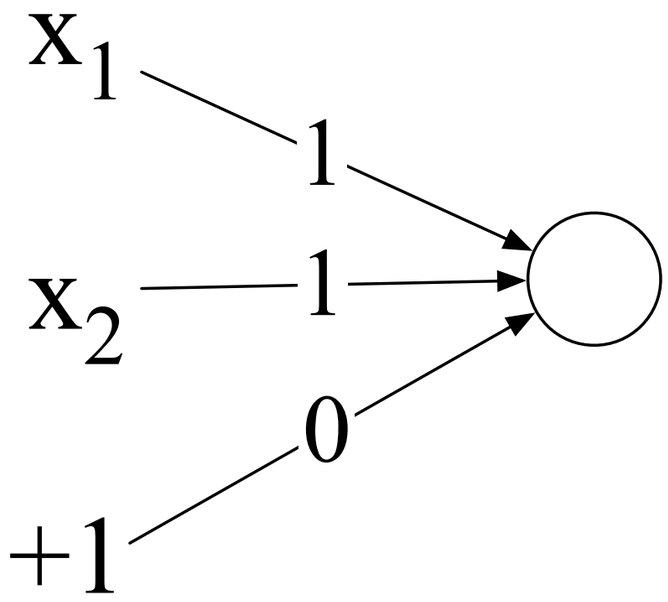
Easy to build AND or OR with perceptrons

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



AND

AND		
x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1



OR

OR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

Not possible to capture XOR with perceptrons

Pause the lecture and try for yourself!

Why? Perceptrons are linear classifiers

Perceptron equation given x_1 and x_2, is the equation of a line

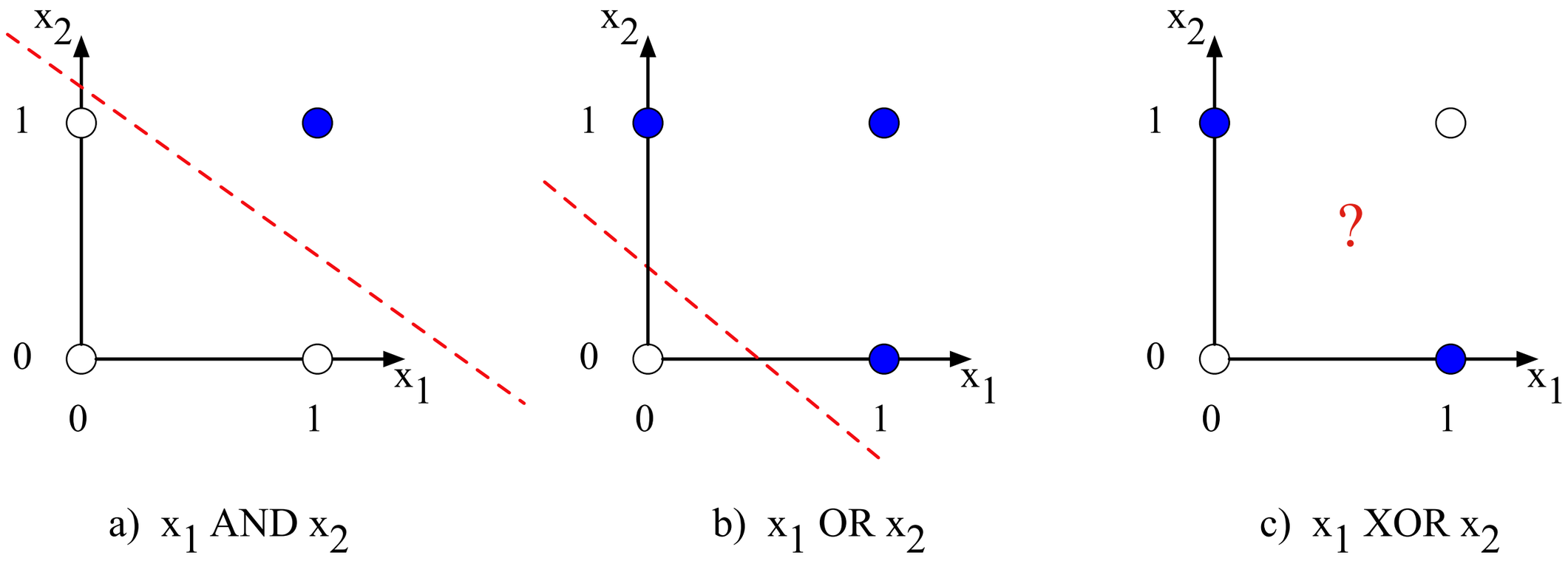
$$w_1 x_1 + w_2 x_2 + b = 0$$

(in standard linear format: $x_2 = (-w_1/w_2)x_1 + (-b/w_2)$)

This line acts as a **decision boundary**

- 0 if input is on one side of the line
- 1 if on the other side of the line

Decision boundaries



a) x_1 AND x_2

b) x_1 OR x_2

c) x_1 XOR x_2

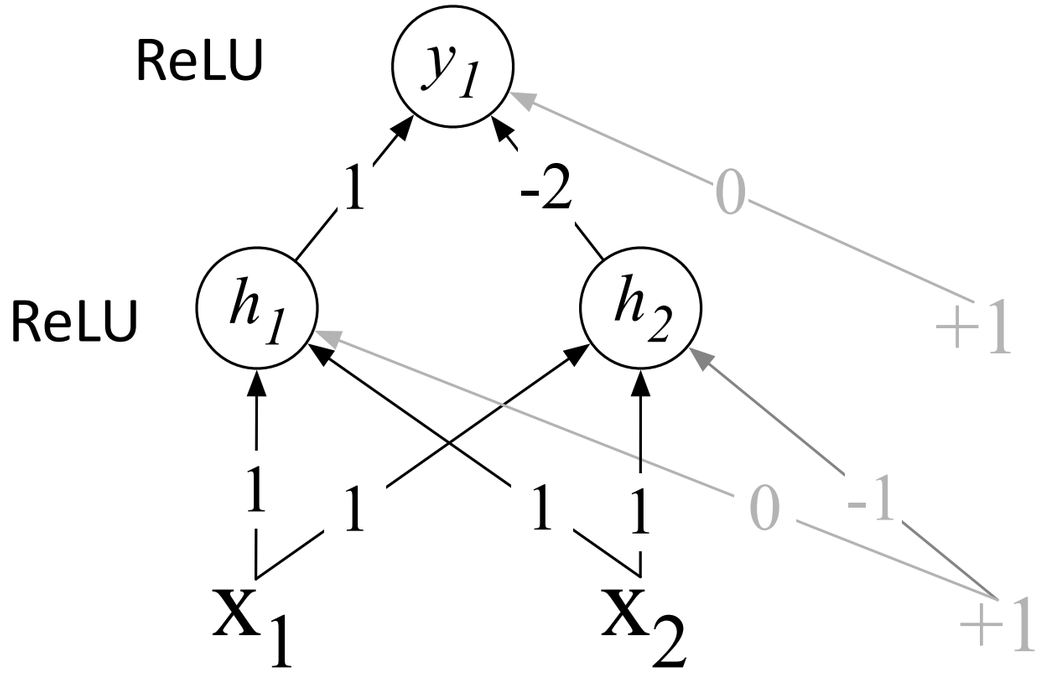
XOR is not a **linearly separable** function!

Solution to the XOR problem

XOR **can't** be calculated by a single perceptron

XOR **can** be calculated by a layered network of units.

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

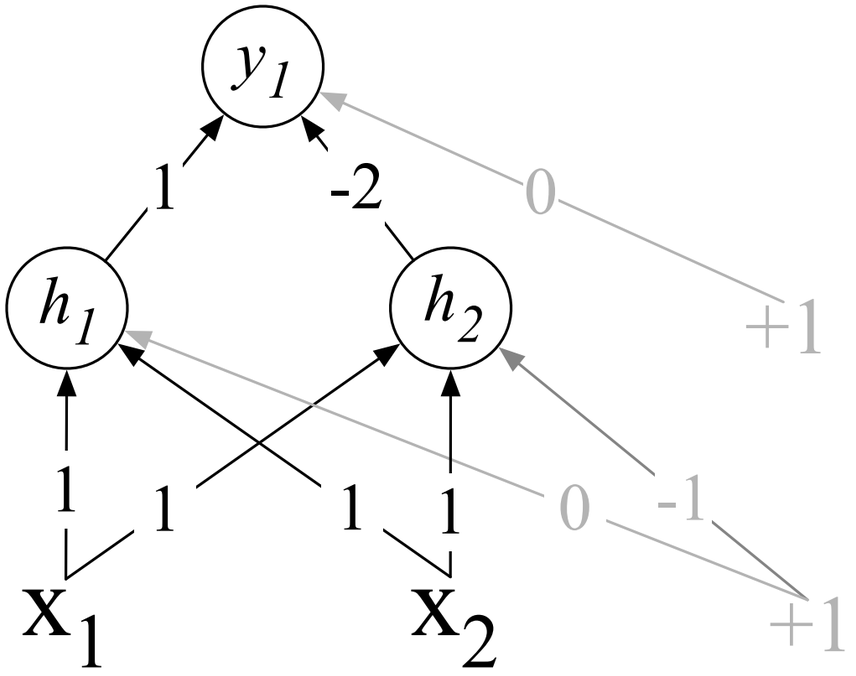


Solution to the XOR problem

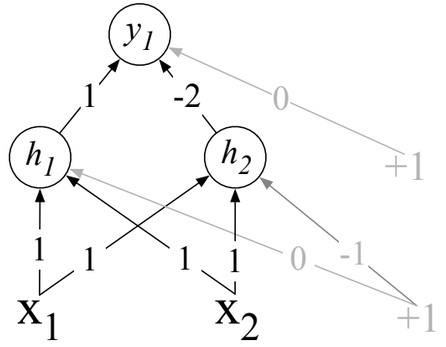
XOR **can't** be calculated by a single perceptron

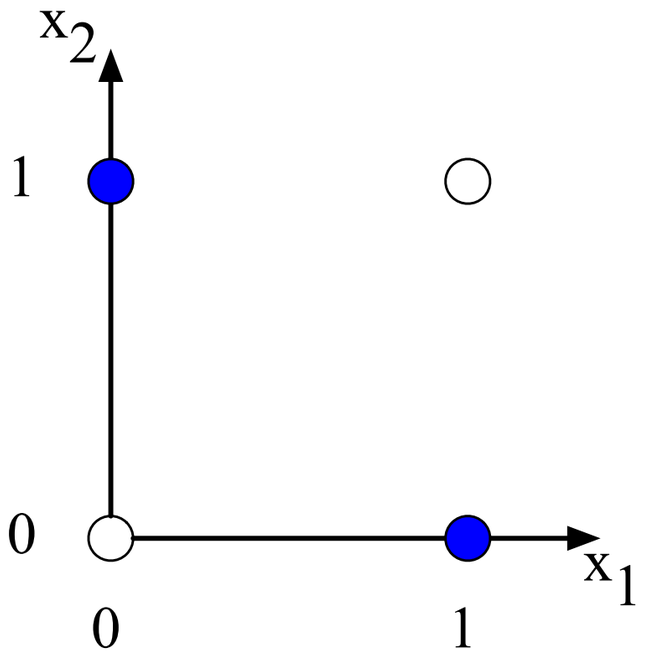
XOR **can** be calculated by a layered network of units.

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

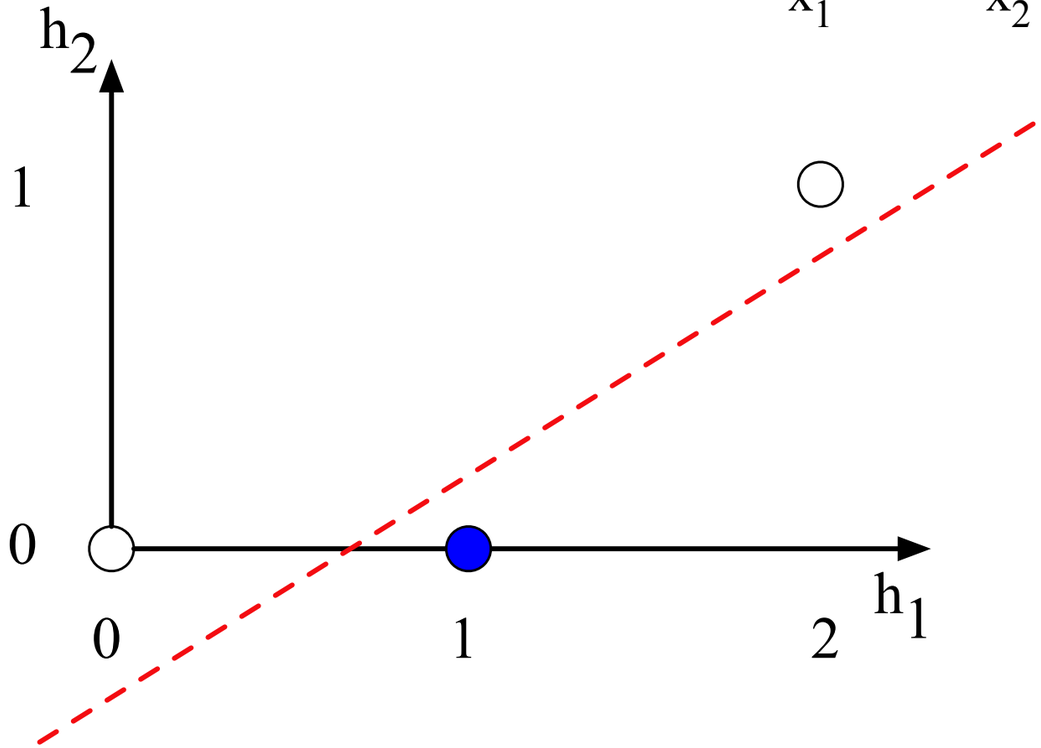


The hidden representation h





a) The original *x* space



b) The new (linearly separable) *h* space

(With learning: hidden layers will learn to form useful representations)

Agenda

- Motivation: why embeddings?
- Neural network fundamentals
- Training Neural Networks
- Computation Graphs and Backward Differentiation
- Why is this scalable?
- The XOR problem
- **Learning embeddings: Autoencoders**

Using slides by:
Jure Leskovec & Mina Ghashami, Dan Jurafsky & James H. Martin,
Roger Grosse, Stephen Scott, Geoffrey Hinton

Learning Embeddings using Neural Networks

Agenda

- We will work with three examples:

  Word embeddings

  - Word embeddings produced by **Word2Vec** model
  - Converts one-hot encoding to dense embedding

  Autoencoders:

  - Learn representation by reconstructing input
  - Unsupervised mode

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 task-independent , and unsupervised.

Word2Vec

- Word2Vec comes in two architectures:
  - Continuous bag of words (CBOW)
  - Skip Gram

    (We will discuss skip-gram model today)

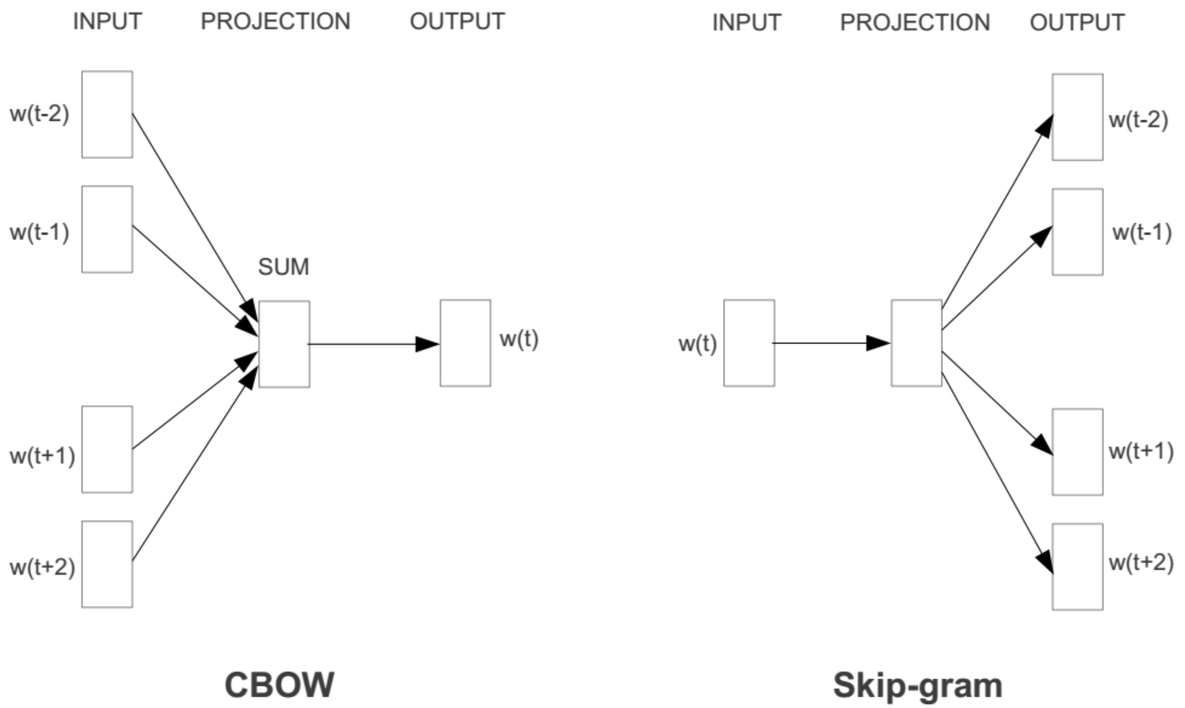
  - The two methods are very similar, both use a shallow neural network (only 1 hidden layer) to learn word representations.

  - The key idea of **word2Vec** is that words with similar context have similar meanings.
    - It learns embedding based on the usage of words.

word2vec: Representing the Meaning of Words [Mikolov et al., 2013]

Key idea: Predict surrounding words of every word

Benefits: Faster and easier to incorporate new document, words, etc.



Continuous Bag of Words (**CBOW**): use context words in a window to predict middle word

Skip-gram: use the middle word to predict surrounding ones in a window

*slide from Vagelis Hristidis

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:

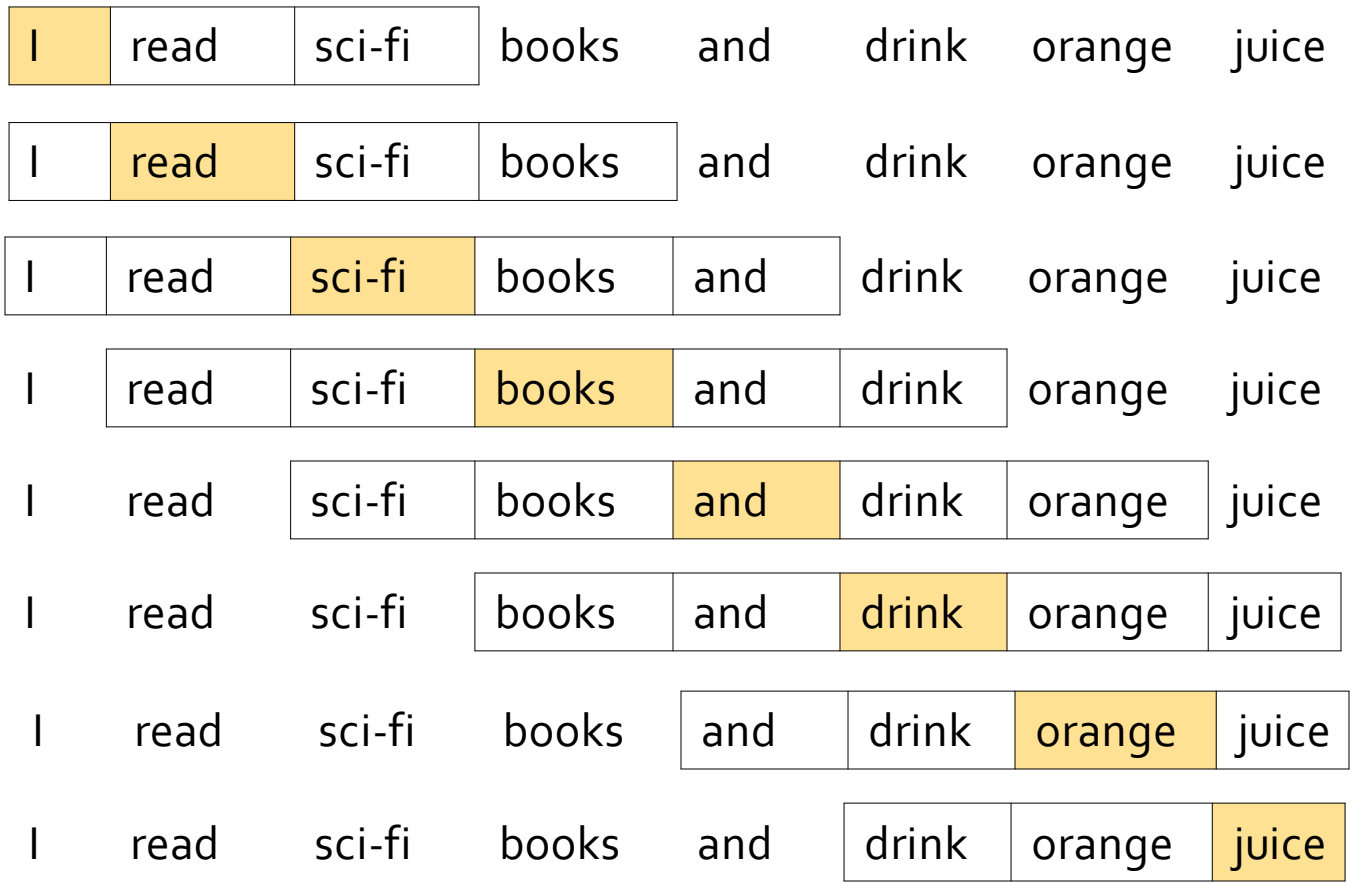
Set a window size (e.g. window=2). For any given word (aka target word), 2 words to its left & 2 words to its right are the context words. Window size is a hyperparameter.

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.

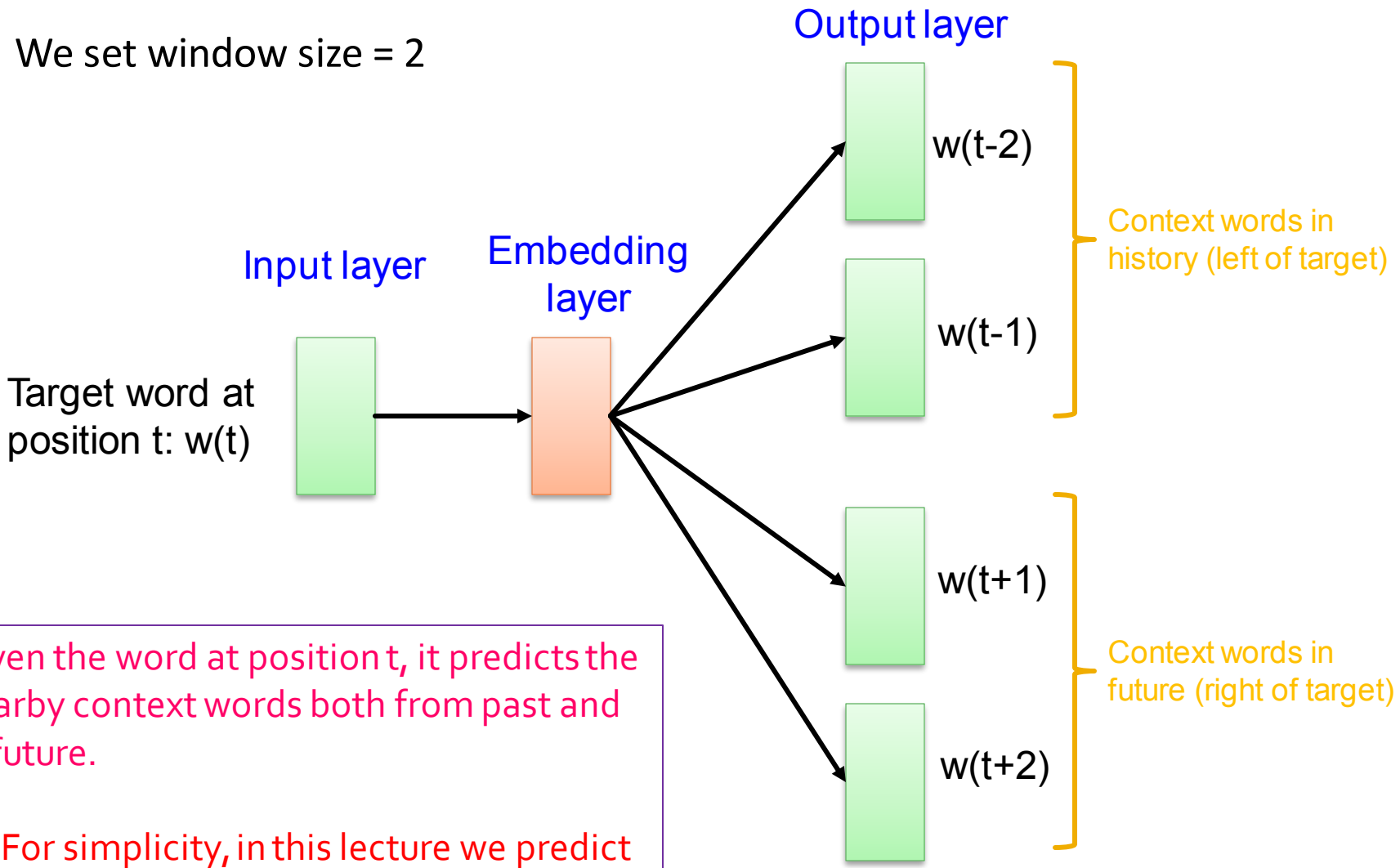


Word2Vec: architecture

- 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

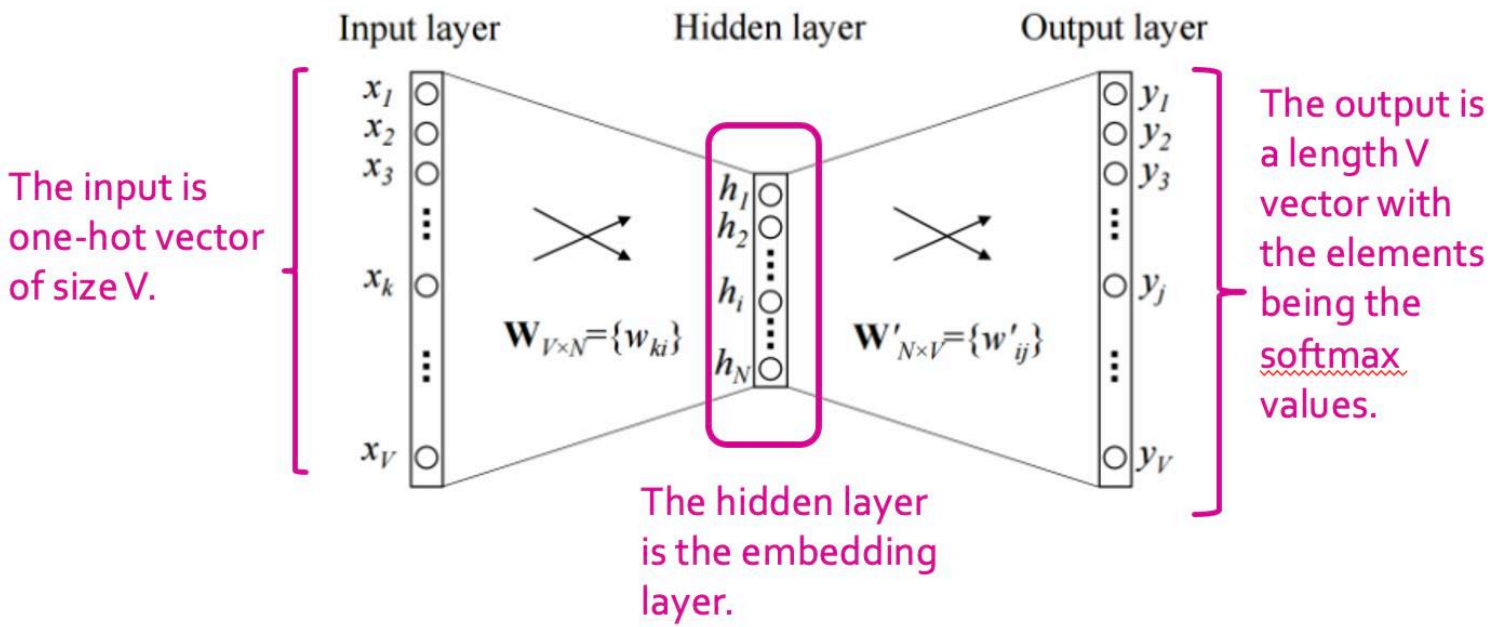


Given the word at position t, it predicts the nearby context words both from past and in future.

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

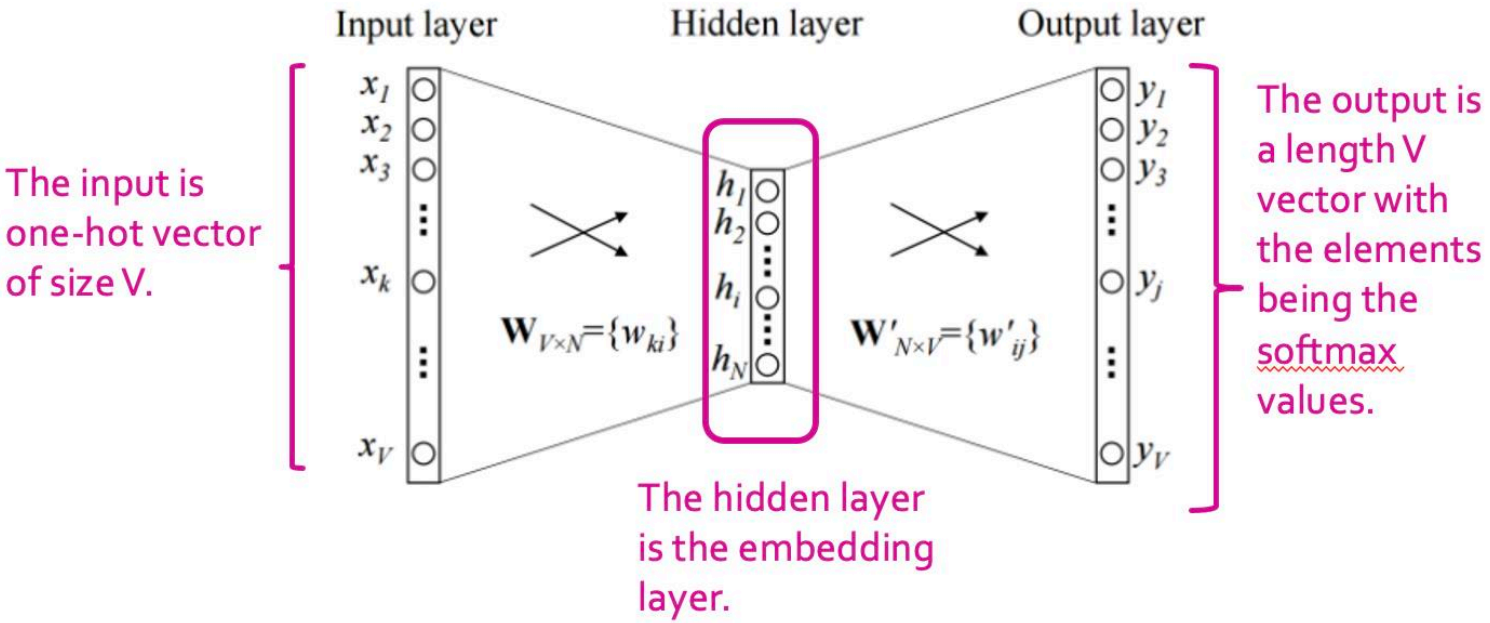
Word2Vec: architecture

- Word2Vec is a 2 layers NN (i.e. only 1 hidden layer)
- V = size of vocabulary
- N = embedding dimension



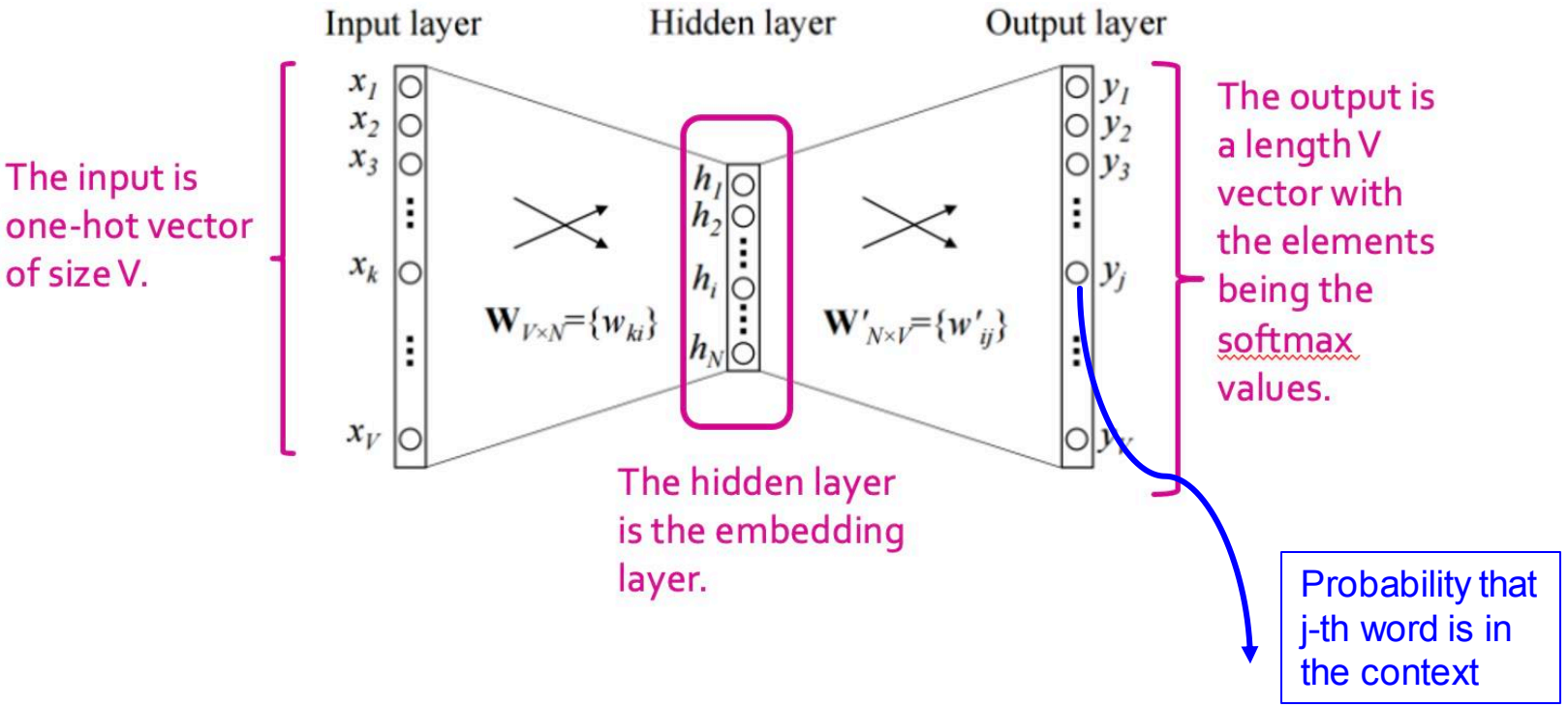
Word2Vec: architecture

- 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



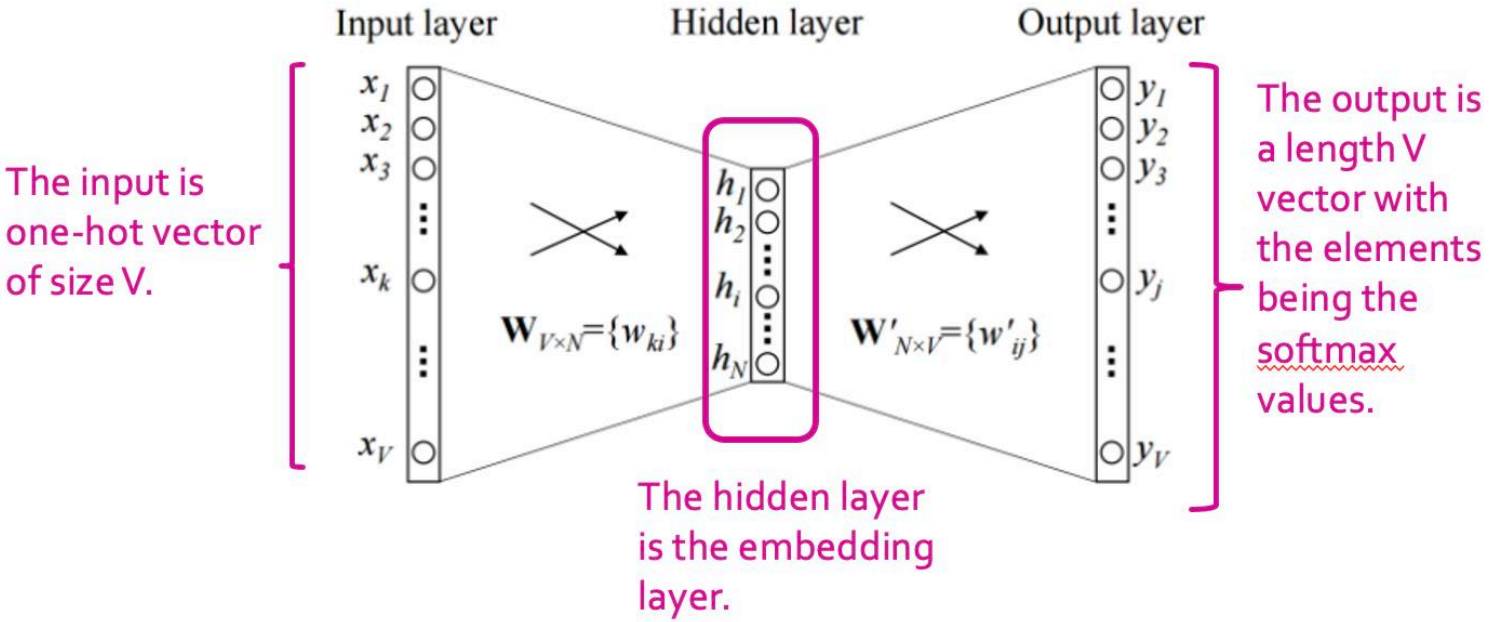
Word2Vec: architecture

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



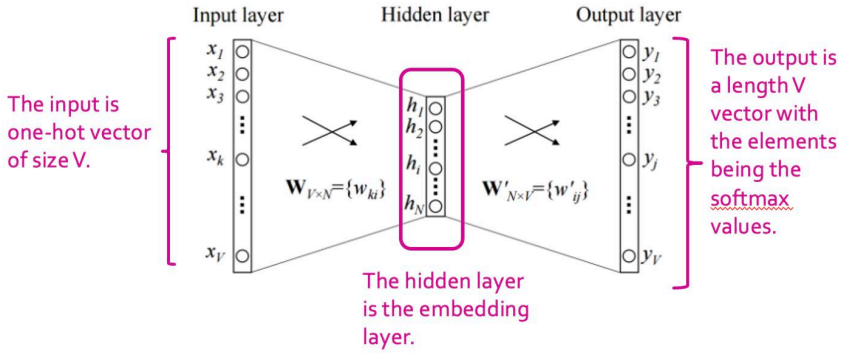
Word2Vec: architecture

- 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}



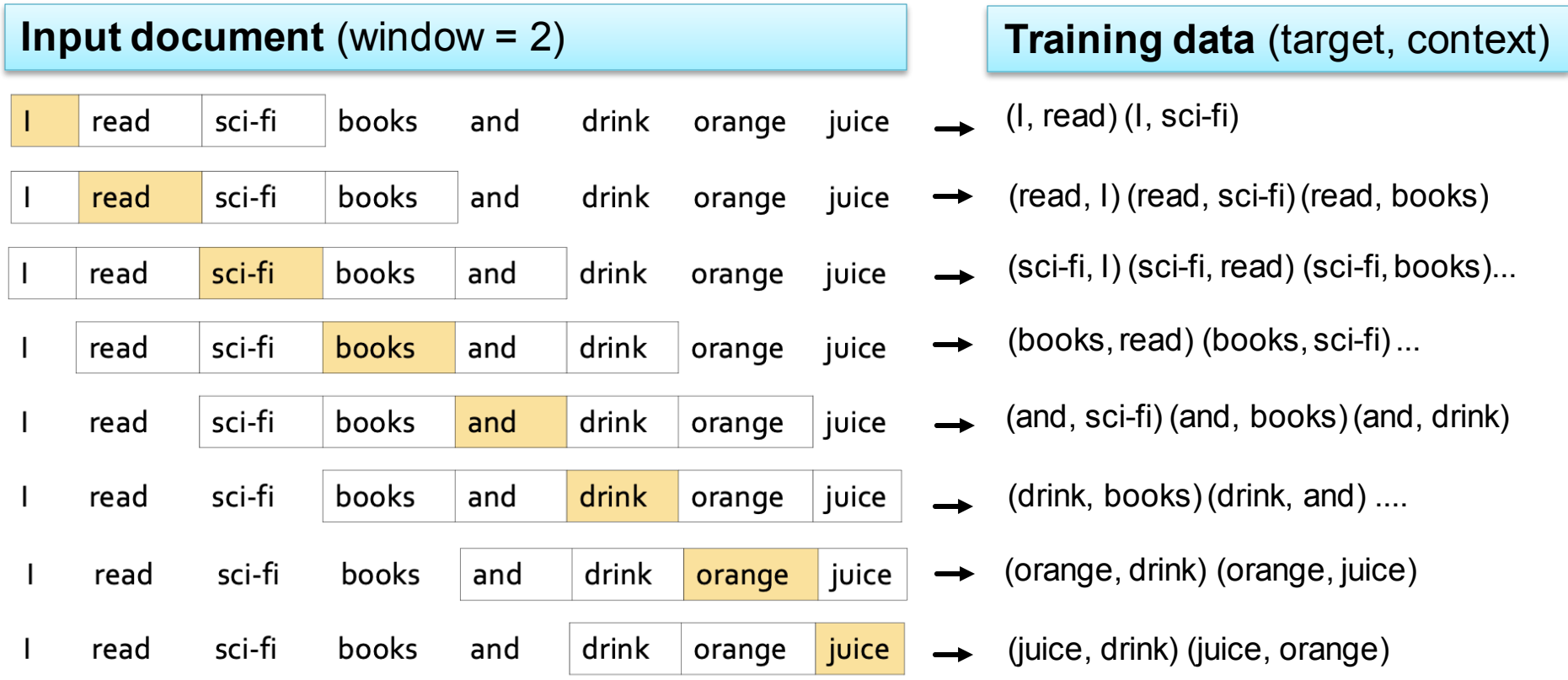
Word2Vec: training the network

- 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?

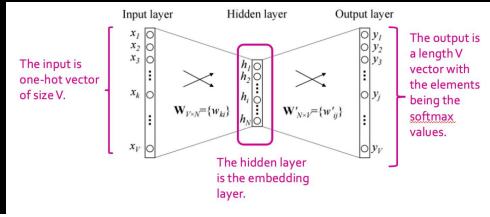


Word2Vec: Training data

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



Word2Vec: Loss function



- 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:

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

- 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 $W_{new} = Wold - \mu\, \partial L/\partial W$

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

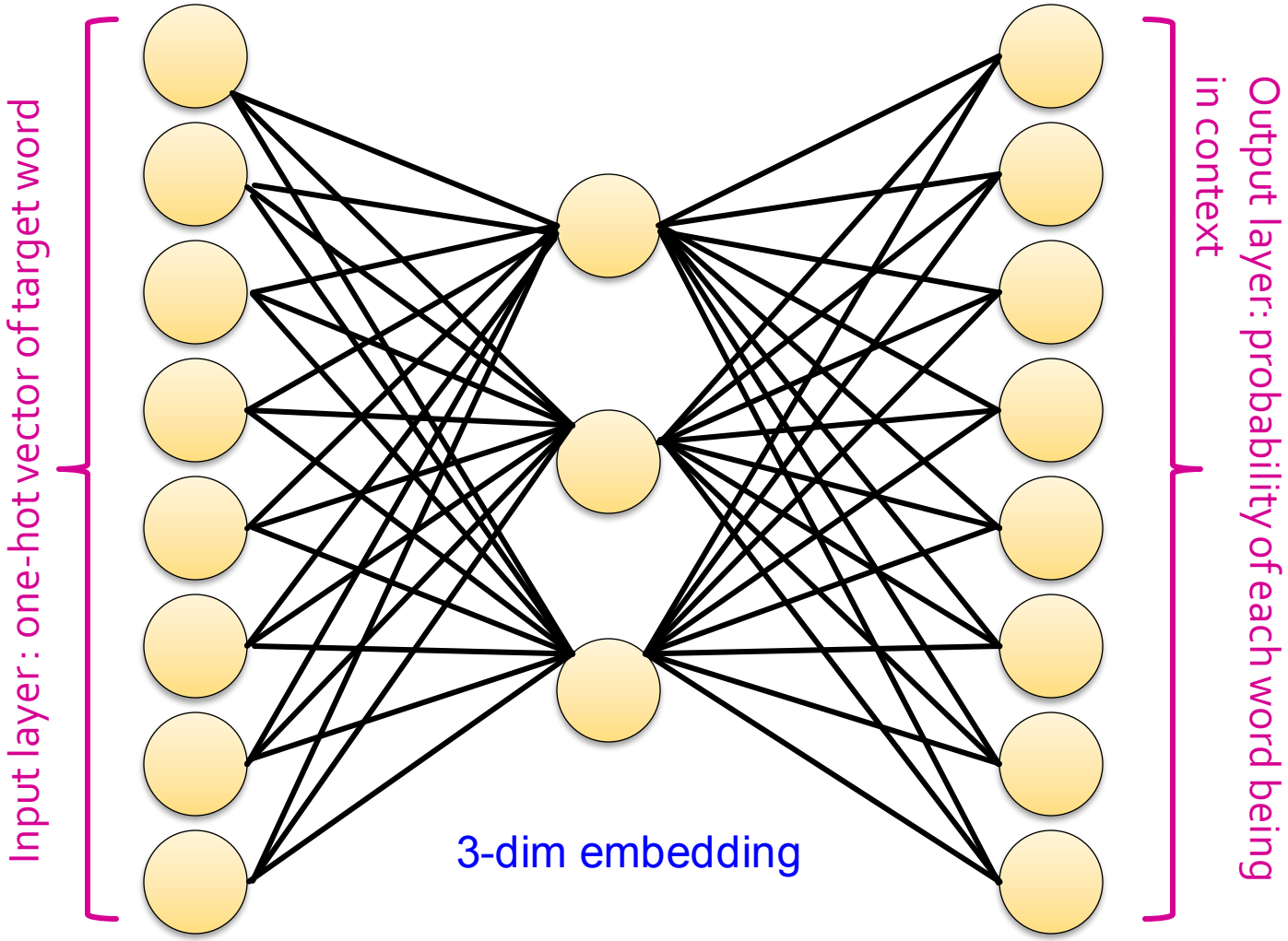
- We execute one forward pass using above document

- 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



Word2Vec: Example

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

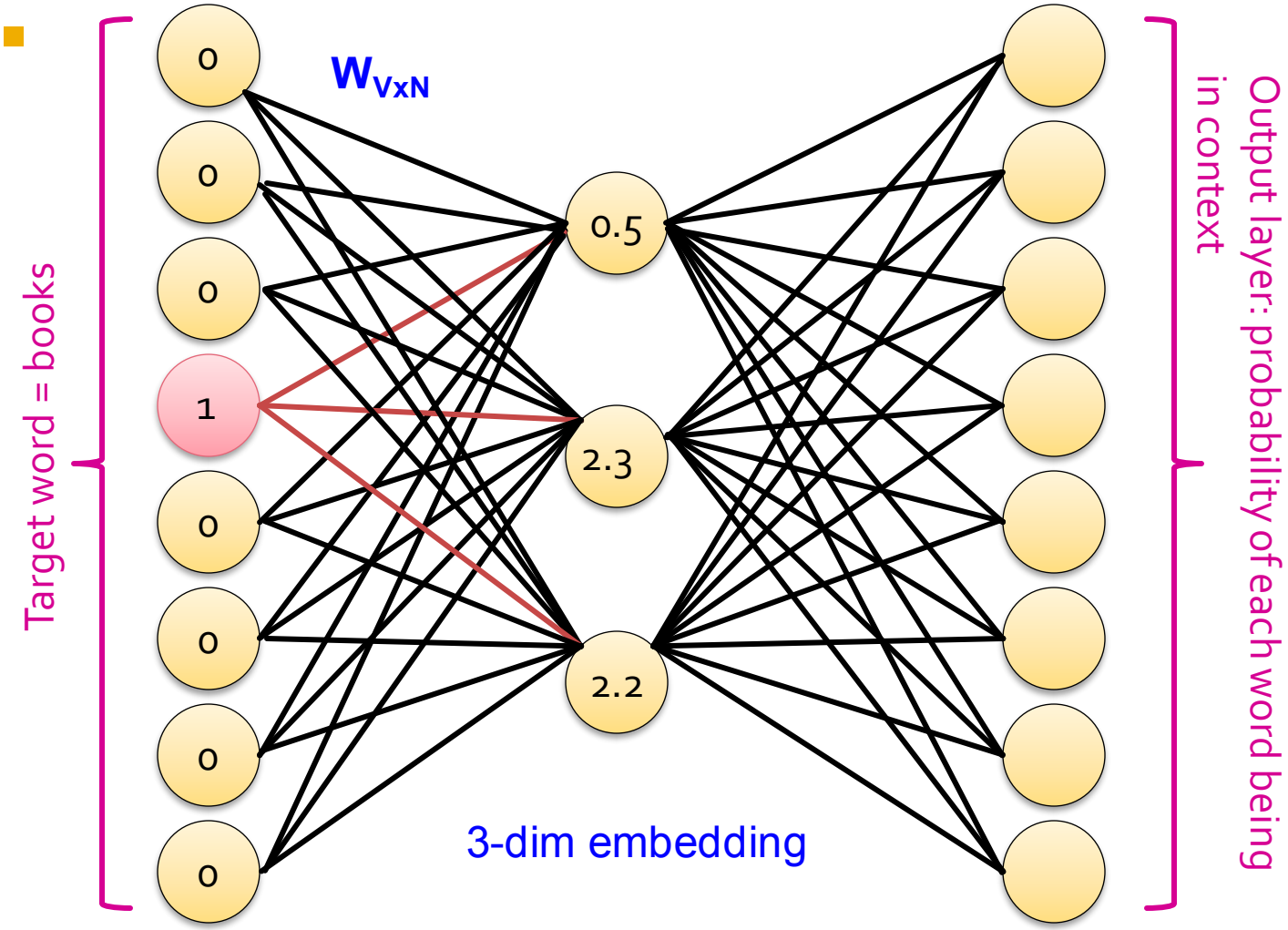
$$W_{V \times N} = \begin{bmatrix} 1 & 2 & 2 \\ -1.2 & -3 & -2 \\ 1.2 & 1.1 & 0.5 \\ 0.5 & 2.3 & 2 \\ -1.1 & 0.6 & -1 \\ 1 & -1 & 2 \\ 0.3 & 1.2 & 0.7 \end{bmatrix}$$

- 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:

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

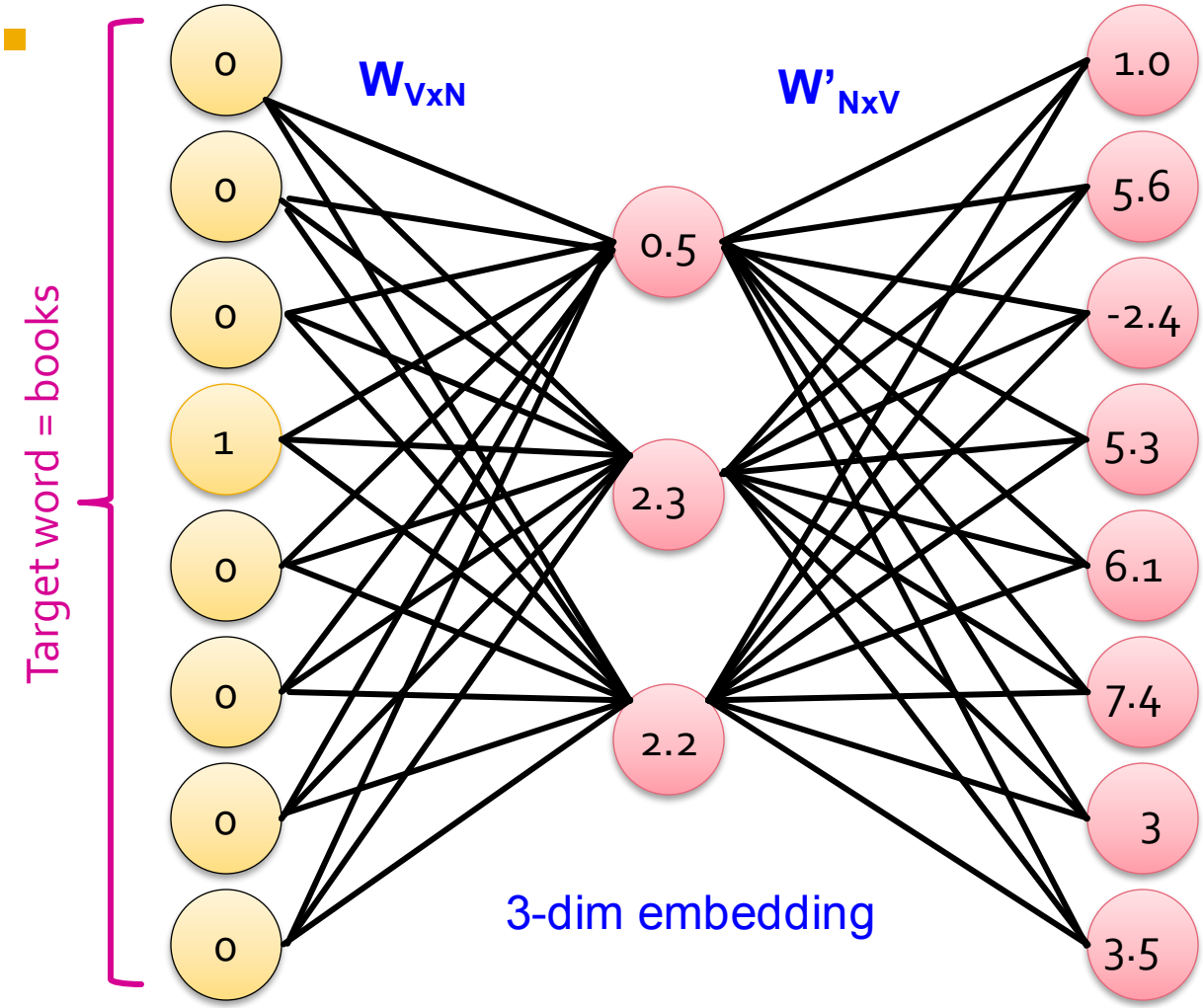
- Then

$$[0.5,2.3,2.2] \times \begin{bmatrix} 1 & 2 & 2 & 0 & 0.7 & 1.3 & -1 & -0.1 \\ 1.2 & 0.5 & -1 & 1 & 0.3 & 2 & 0.6 & 1 \\ -1 & 1.6 & -0.5 & 1.4 & 2.3 & 1 & 1 & 0.6 \end{bmatrix}$$

$$= [1.0, 5.6, -2.4, 5.3,6.1,7.4,3.0,3.5]$$

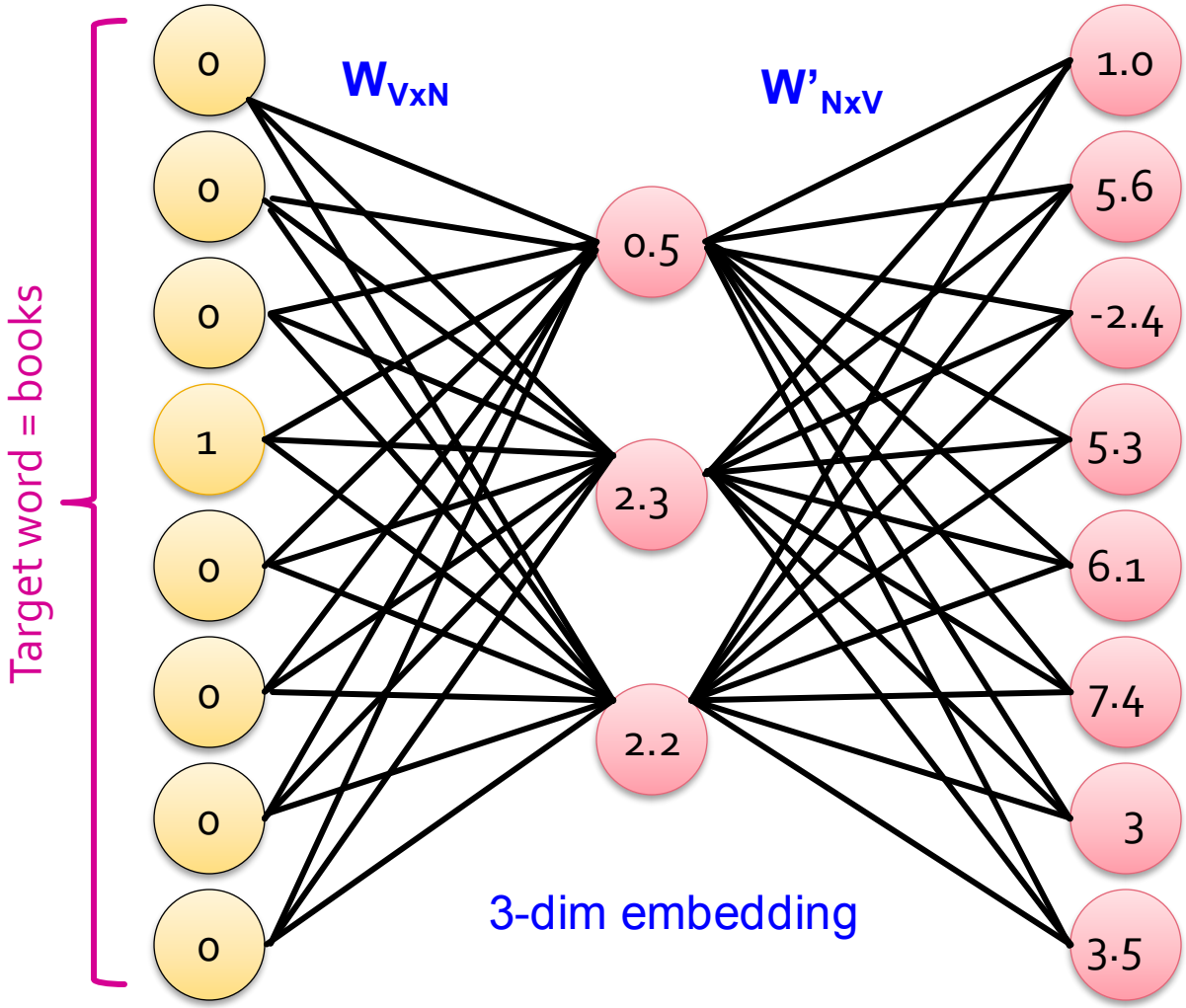
Word2Vec: Example

- If W'_{NxV} given:



Word2Vec: Example

- If W'_{NxV} given:

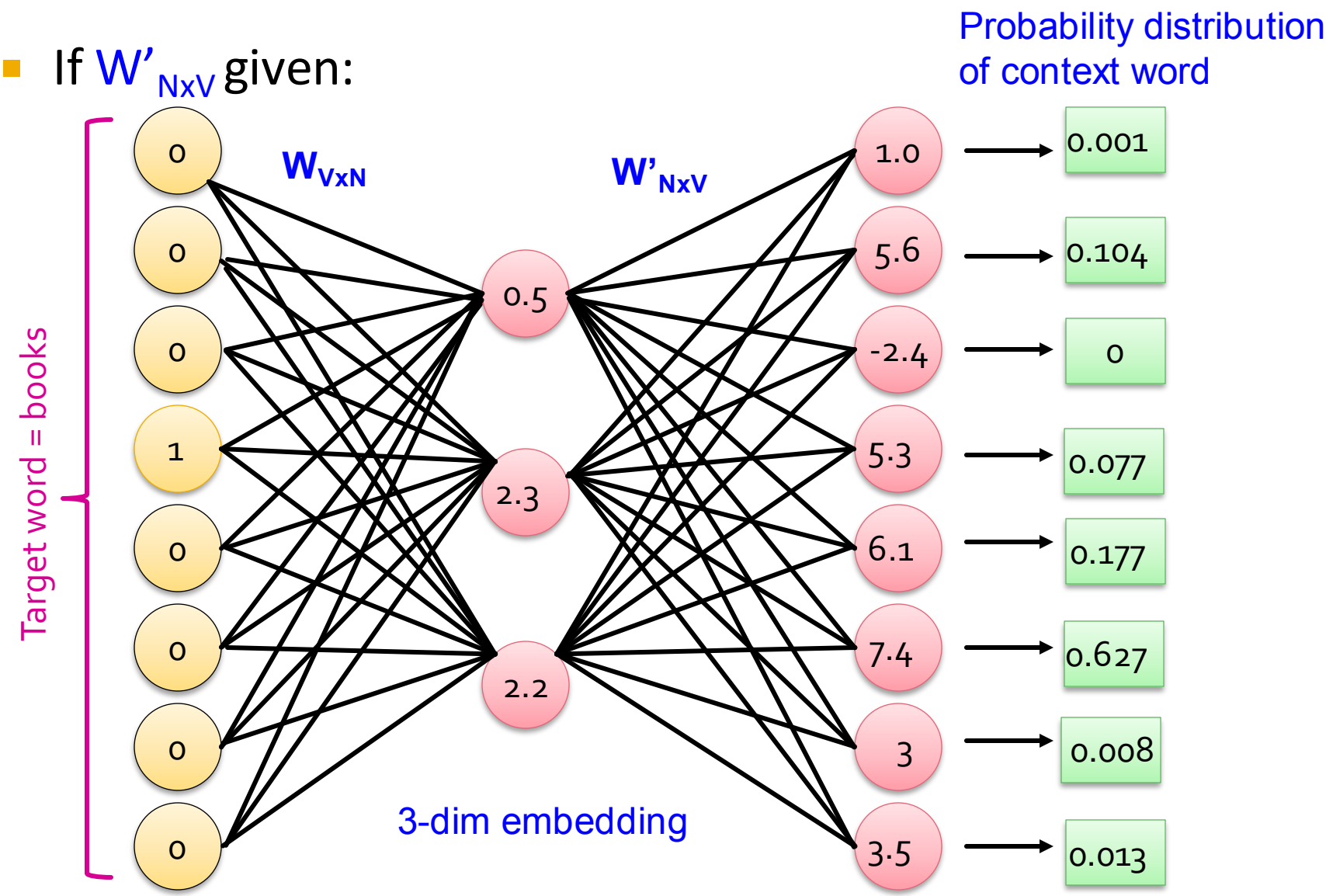


We apply softmax functions to turn them into probabilities.

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

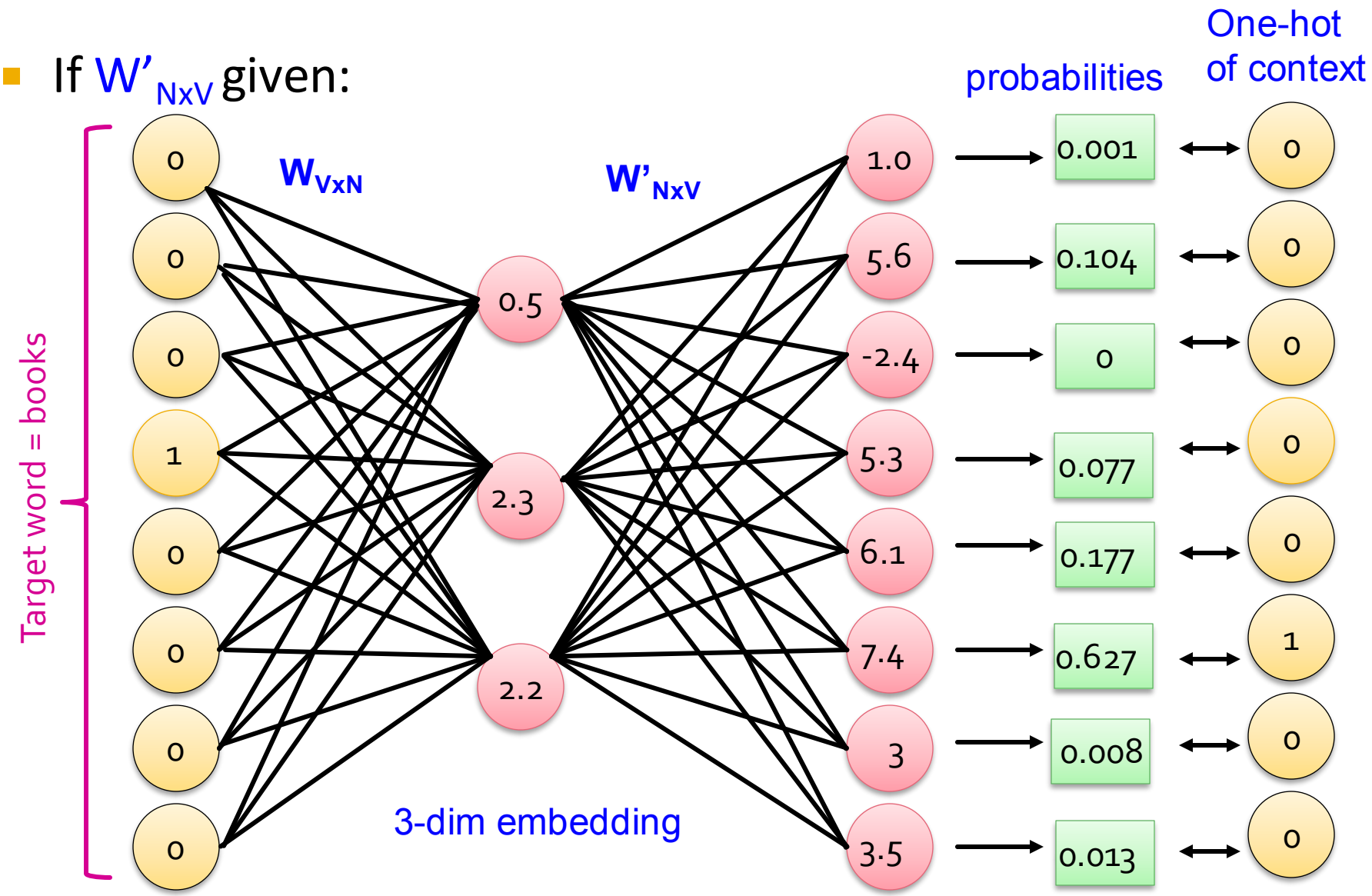
Word2Vec: Example

- If W'_{NxV} given:



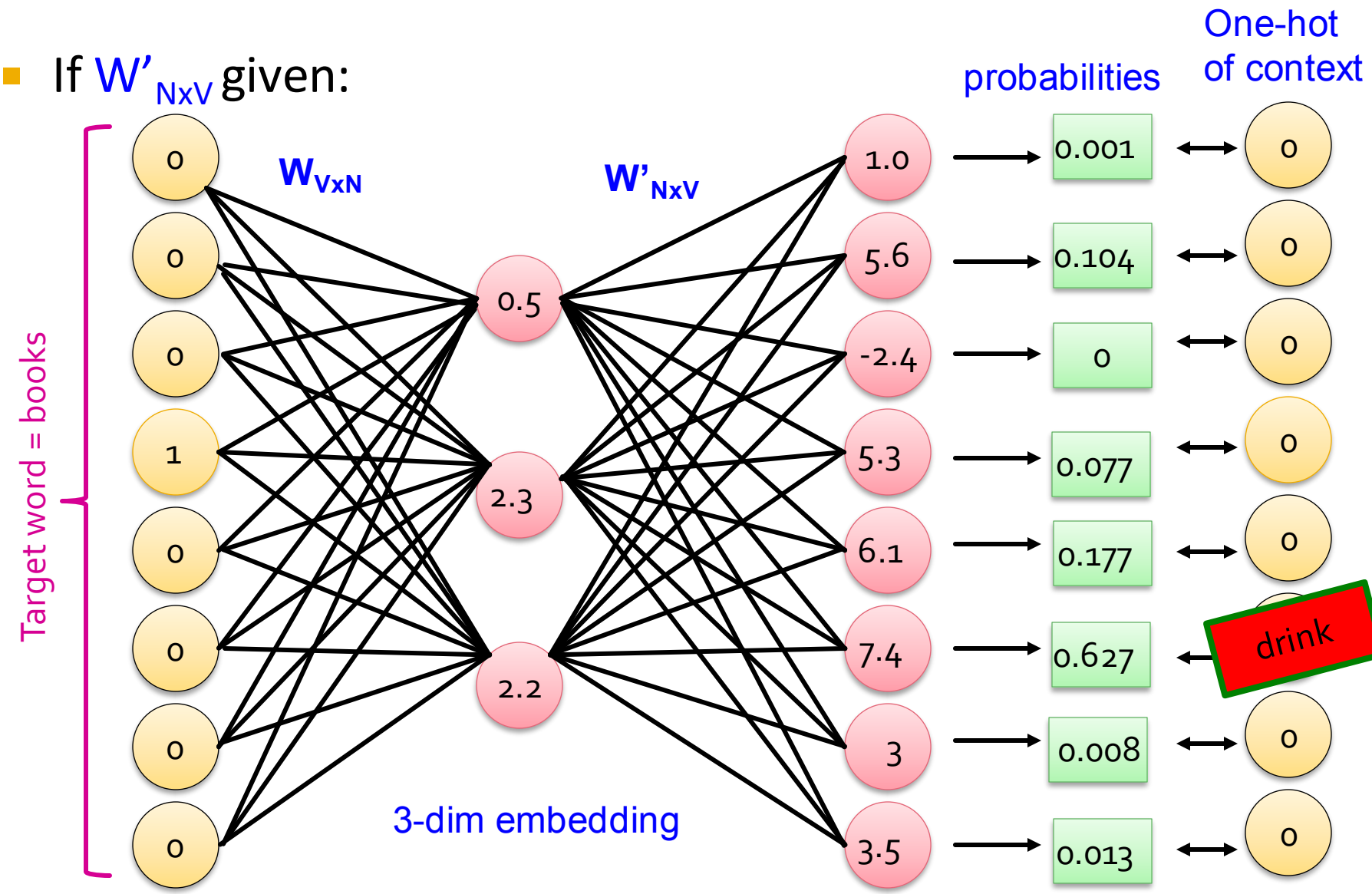
Word2Vec: Example

- 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]

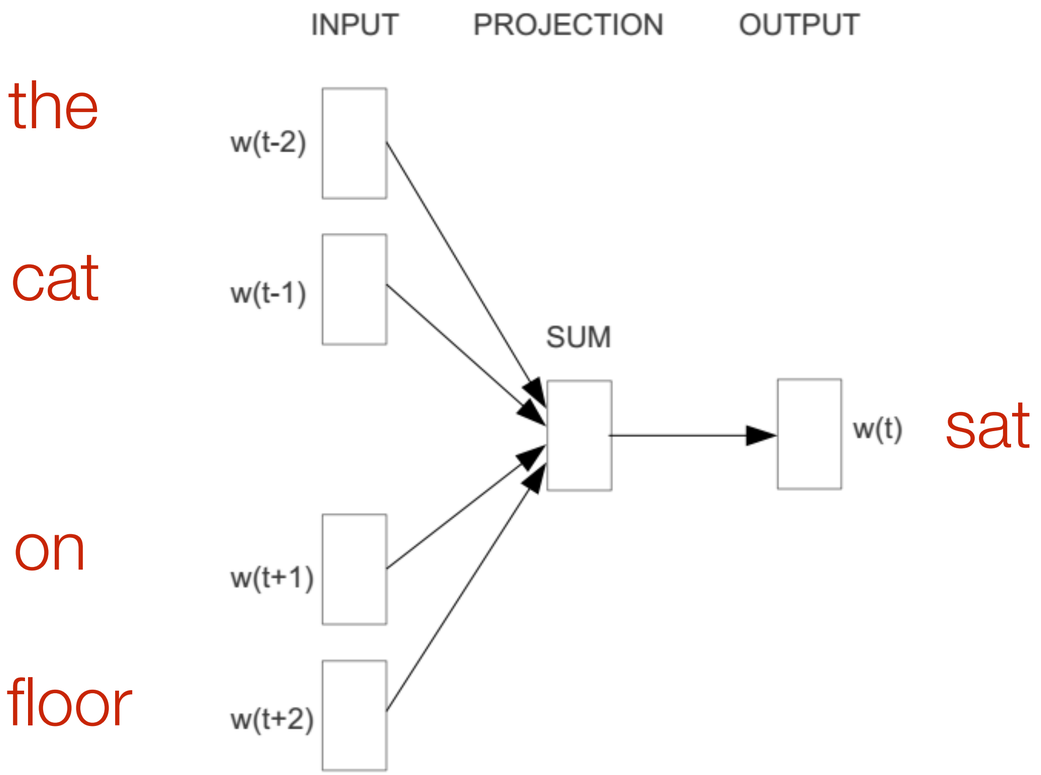
$$L = -\ln(0.104) = 2.26$$

Then

CBOW: Continuous Bag of Words

[Mikolov et al., 2013]

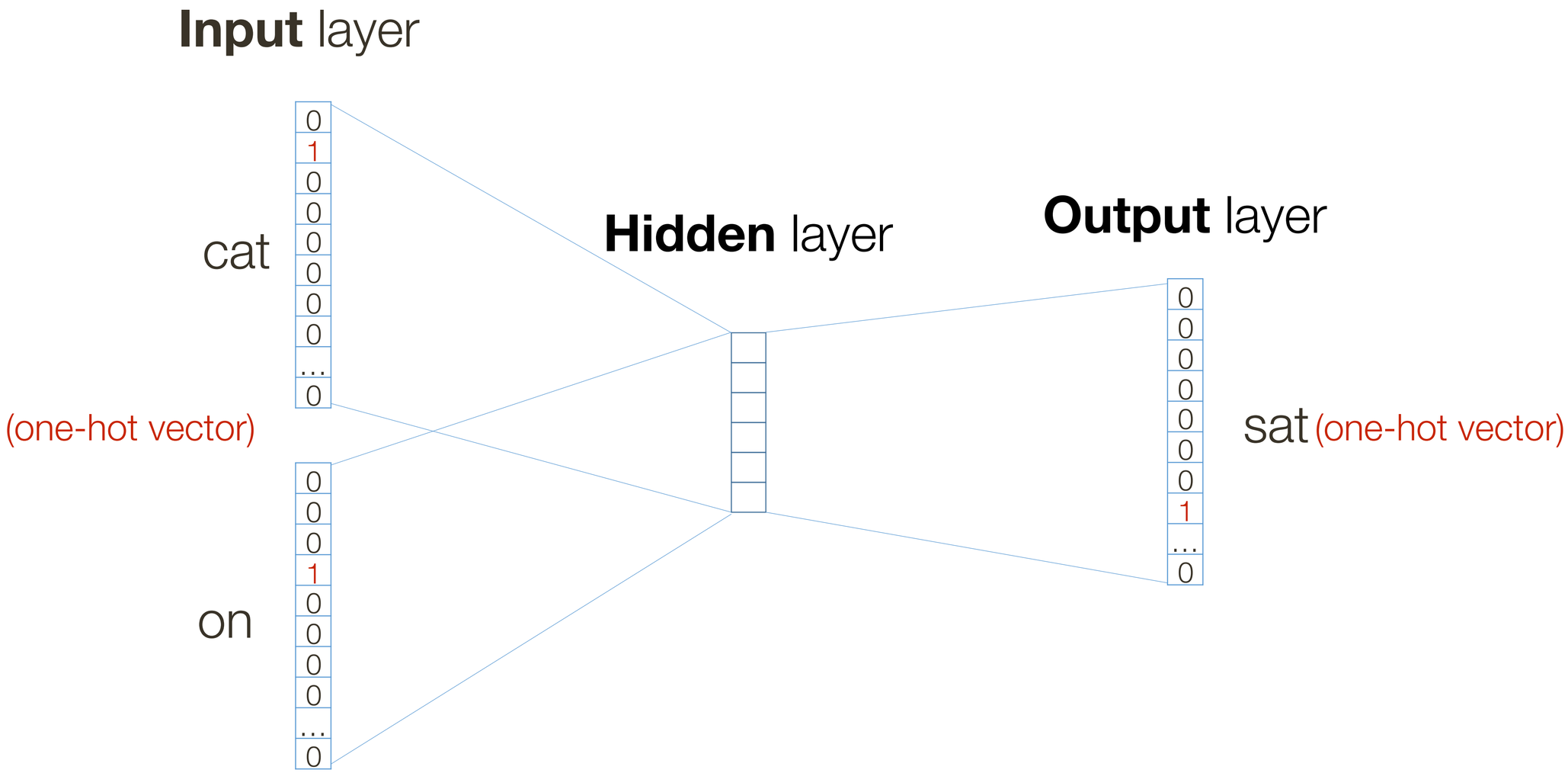
Example: “The cat sat on floor” (window size 2)



*slide from Vagelis Hristidis

CBOW: Continuous Bag of Words

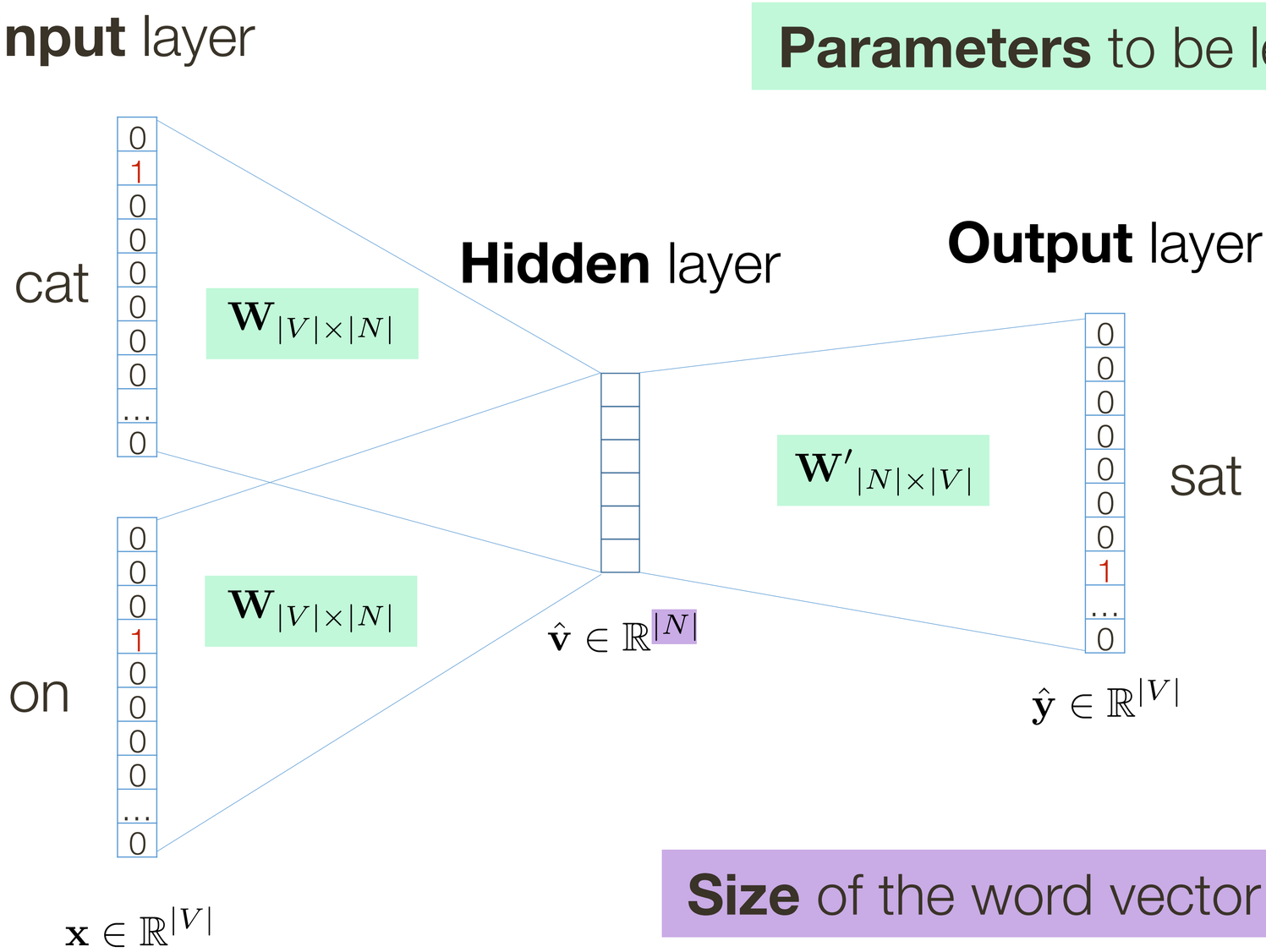
[Mikolov et al., 2013]



*slide from Vagelis Hristidis

CBOW: Continuous Bag of Words

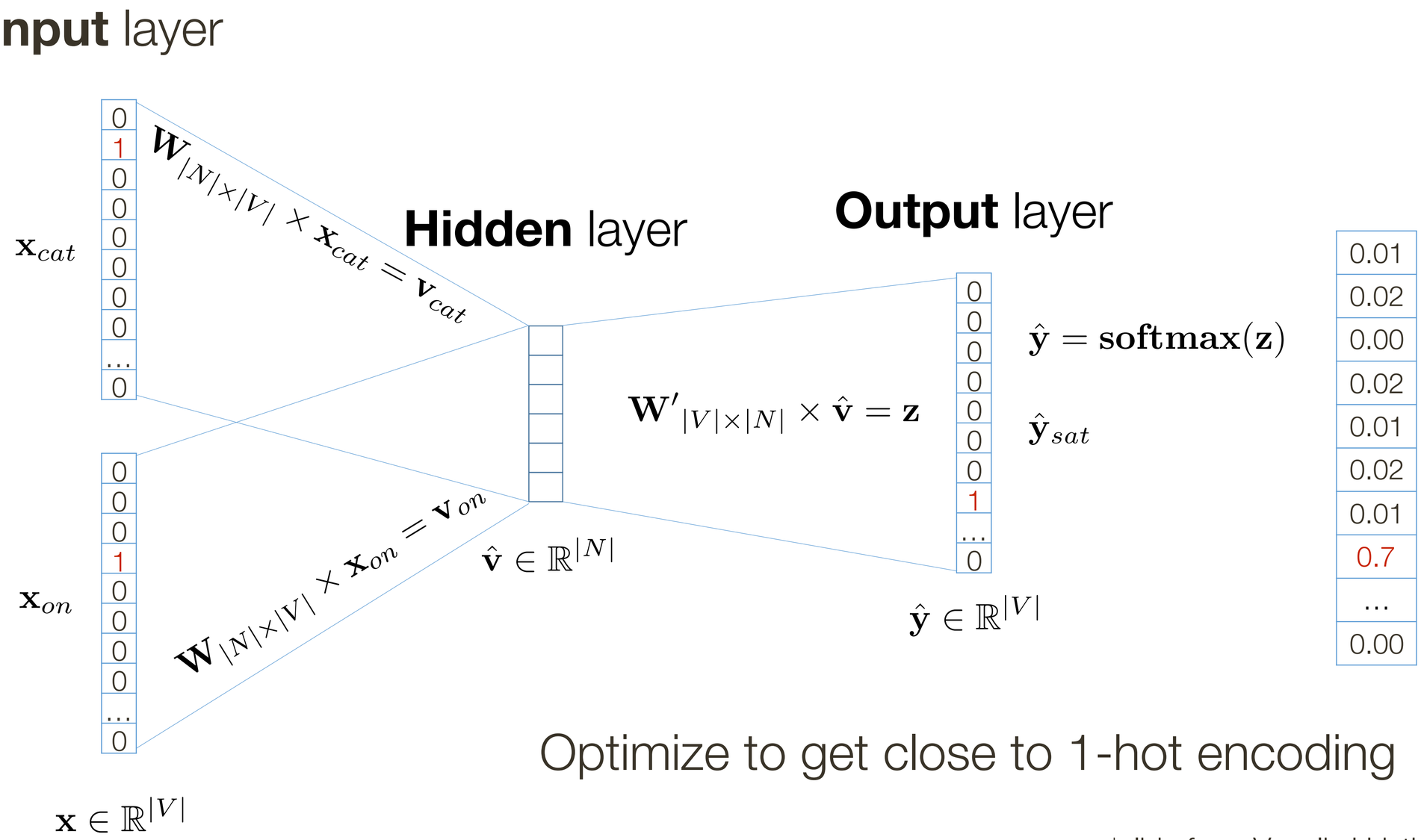
[Mikolov et al., 2013]



*slide from Vagelis Hristidis

CBOW: Continuous Bag of Words

[Mikolov et al., 2013]



Optimize to get close to 1-hot encoding

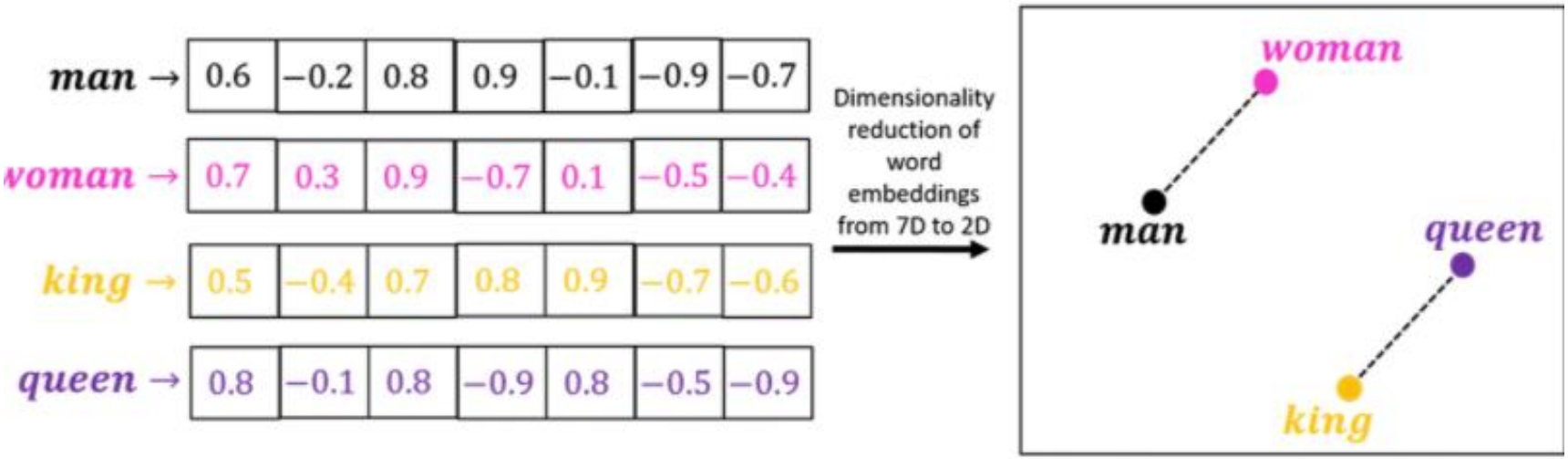
*slide from Vagelis Hristidis

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



Interesting Results: **Word Analogies**

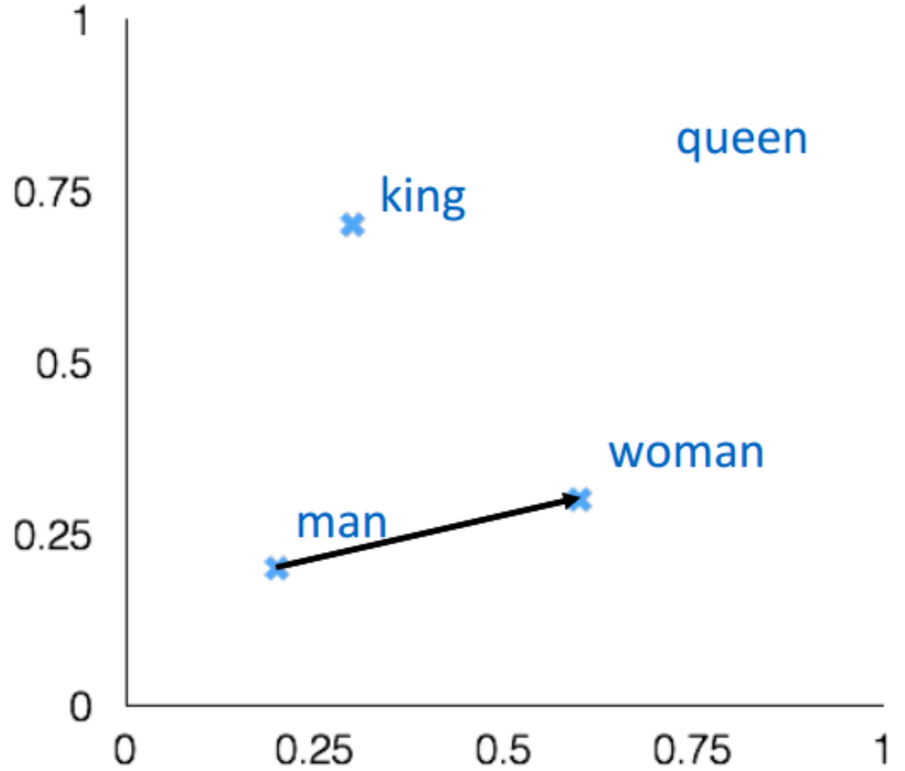
Test for linear relationships, examined by Mikolov et al. (2014)

a:b :: c:?

$$d = \arg\max_x \frac{(w_b - w_a + w_c)^T w_x}{||w_b - w_a + w_c||}$$

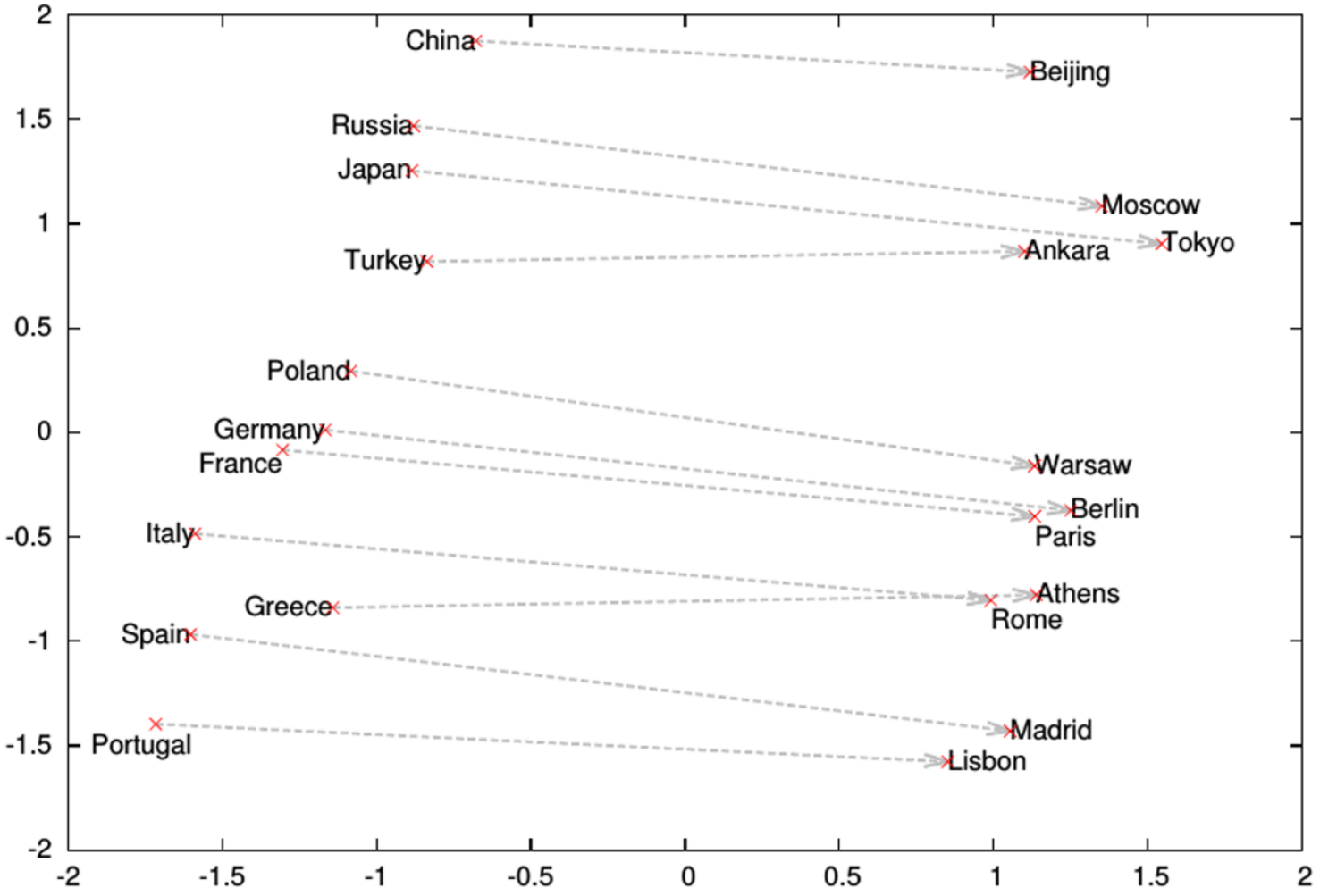
man:woman :: king:?

+	king	[0.30 0.70]
-	man	[0.20 0.20]
+	woman	[0.60 0.30]
	queen	[0.70 0.80]



Interesting Results: **Word Analogies**

[Mikolov et al., 2013]



Autoencoders

Autoencoder

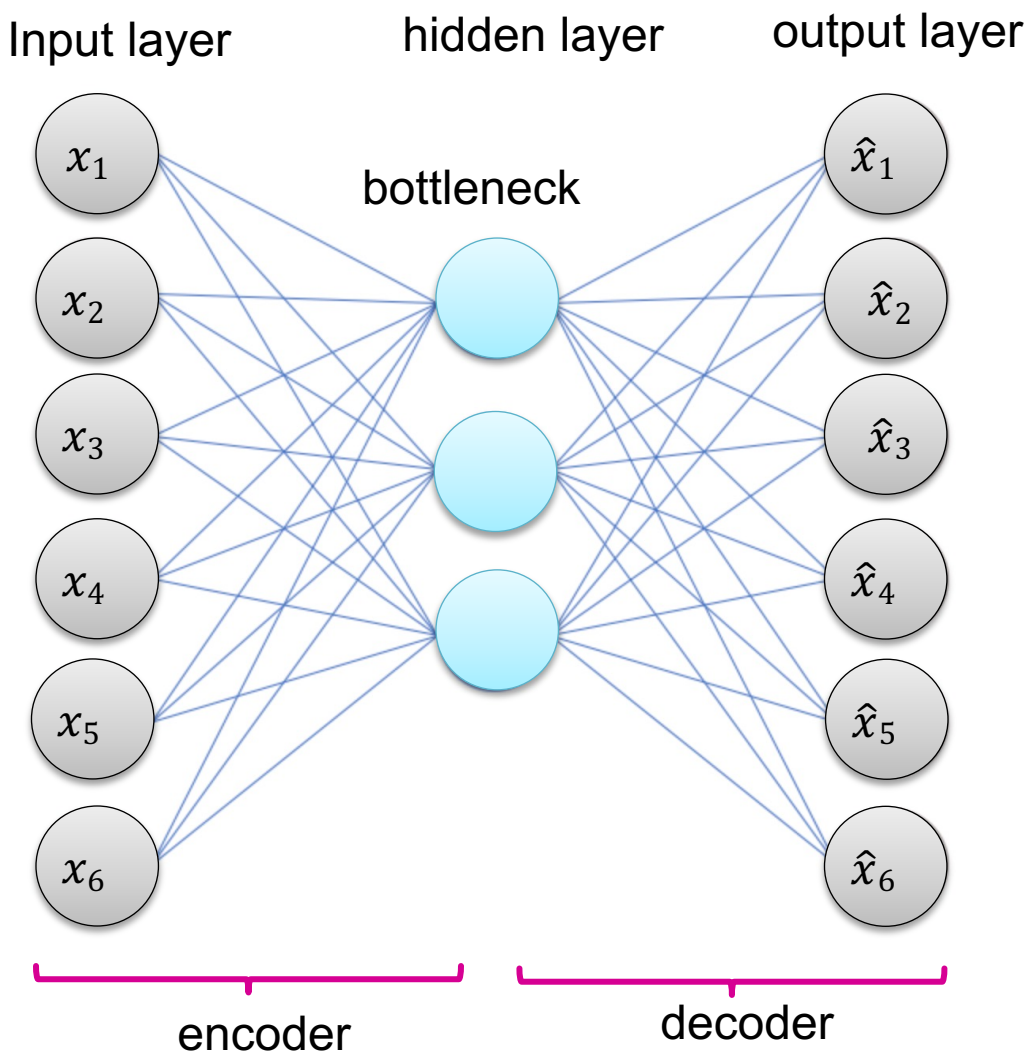
- Autoencoder are an extension of PCA to non-linear space
- They are a special type of neural network that is trained to copy input to output **except** that it has to go through a bottleneck
  - They are unsupervised too
- It learns to compress the data while minimizing the *reconstruction error*.

Autoencoder

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

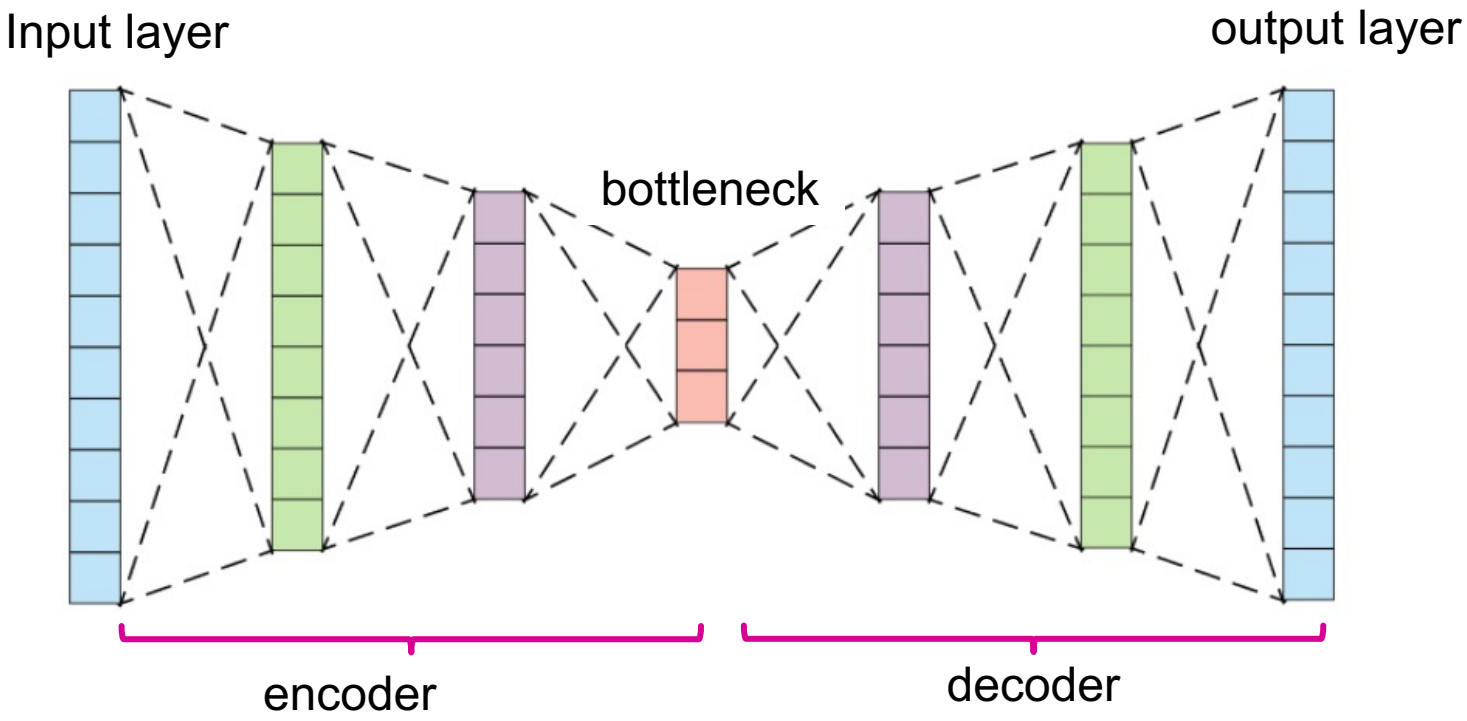
bottleneck layer: is the bottleneck as it projects down 6-dim vector to 3-dim space. It constrains the amount of information that traverses the network

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

Autoencoders

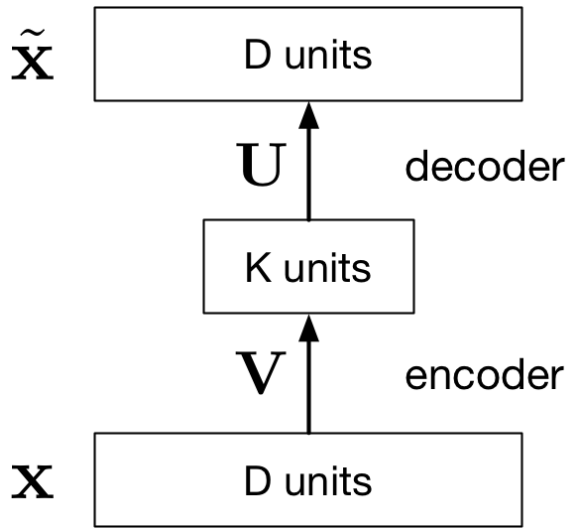
Why autoencoders?

- Map high-dimensional data to two dimensions for visualization
- Compression (i.e. reducing the file size)
  - Note: autoencoders don't do this for free — it requires other ideas as well.
- Learn abstract features in an unsupervised way so you can apply them to a supervised task
  - Unlabled data can be much more plentiful than labeled data

Principal Component Analysis

- The simplest kind of autoencoder has one hidden layer, linear activations, and squared error loss.

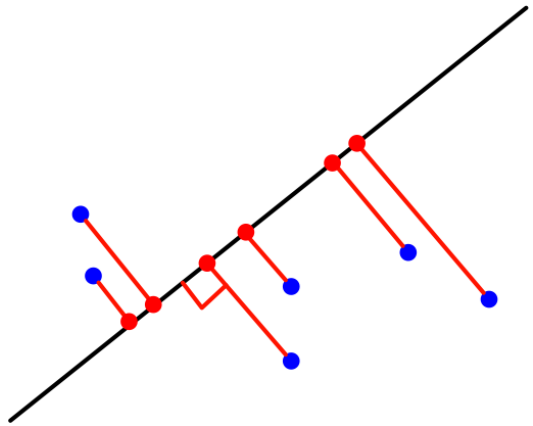
$$\mathcal{L}(\mathbf{x}, \tilde{\mathbf{x}}) = \|\mathbf{x} - \tilde{\mathbf{x}}\|^2$$



- This network computes $\tilde{\mathbf{x}} = \mathbf{UVx}$, which is a linear function.
- If $K \geq D$, we can choose $\mathbf{U}$ and $\mathbf{V}$ such that $\mathbf{UV}$ is the identity. This isn't very interesting.
- But suppose $K < D$:
  - $\mathbf{V}$ maps $\mathbf{x}$ to a K-dimensional space, so it's doing dimensionality reduction.
  - The output must lie in a K-dimensional subspace, namely the column space of $\mathbf{U}$.

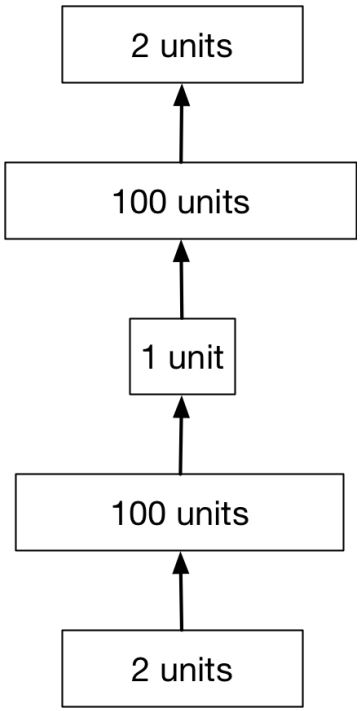
Principal Component Analysis

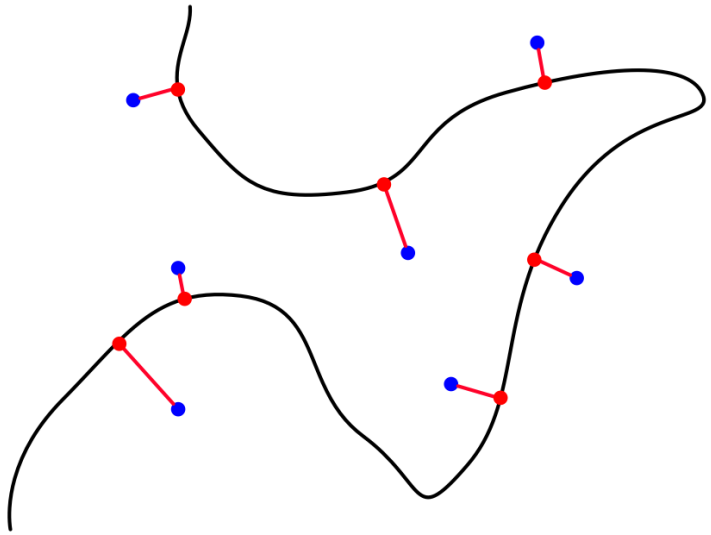
- We just saw that a linear autoencoder has to map D-dimensional inputs to a K-dimensional subspace $\mathcal{S}$.
- Knowing this, what is the best possible mapping it can choose?
  - By definition, the projection of $\mathbf{x}$ onto $\mathcal{S}$ is the point in $\mathcal{S}$ which minimizes the distance to $\mathbf{x}$.



Deep Autoencoders

- Deep nonlinear autoencoders learn to project the data, not onto a subspace, but onto a nonlinear manifold
- This manifold is the image of the decoder.
- This is a kind of nonlinear dimensionality reduction.



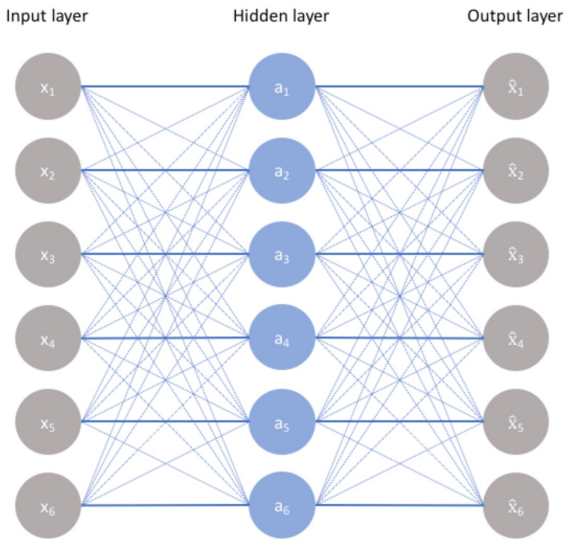


Deep Autoencoders

- Nonlinear autoencoders can learn more powerful codes for a given dimensionality, compared with linear autoencoders (PCA)

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

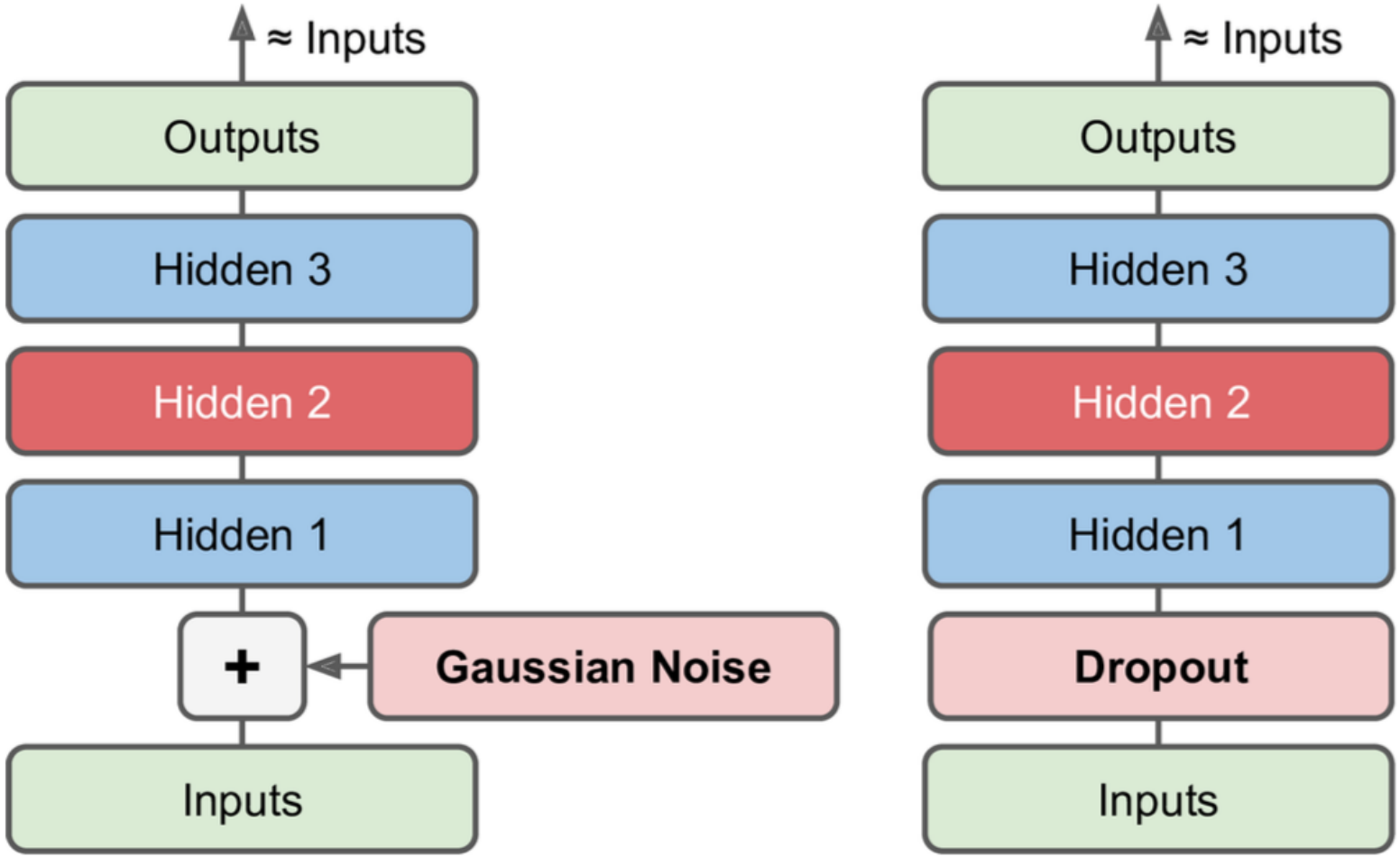
Denoising Autoencoders

Vincent et al. (2010)

- Can train an autoencoder to learn to **denoise** input by giving input **corrupted** instance $\tilde{x}$ and targeting **uncorrupted** instance x
- Example noise models:
  - **Gaussian noise:** $\tilde{x} = x + z$, where $z \sim \mathcal{N}(\mathbf{0}, \sigma^2 I)$
  - **Masking noise:** zero out some fraction ν of components of x
  - **Salt-and-pepper noise:** choose some fraction ν of components of x and set each to its min or max value (equally likely)

Slides by Stephen Scott, based on Paul Quint and Ian Goodfellow

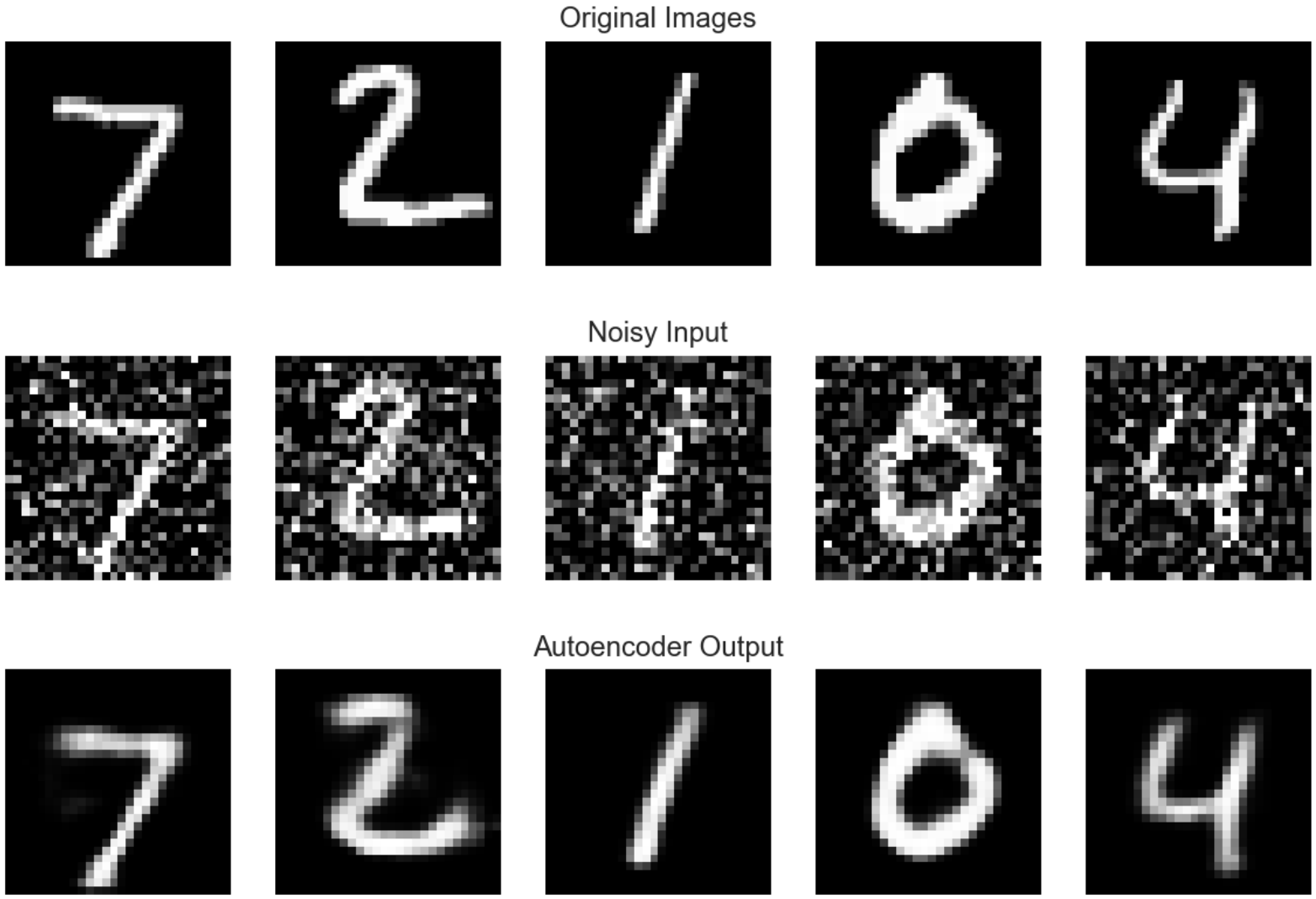
Denoising Autoencoders



Slides by Stephen Scott, based on Paul Quint and Ian Goodfellow

Denoising Autoencoders

Example



Slides by Stephen Scott, based on Paul Quint and Ian Goodfellow

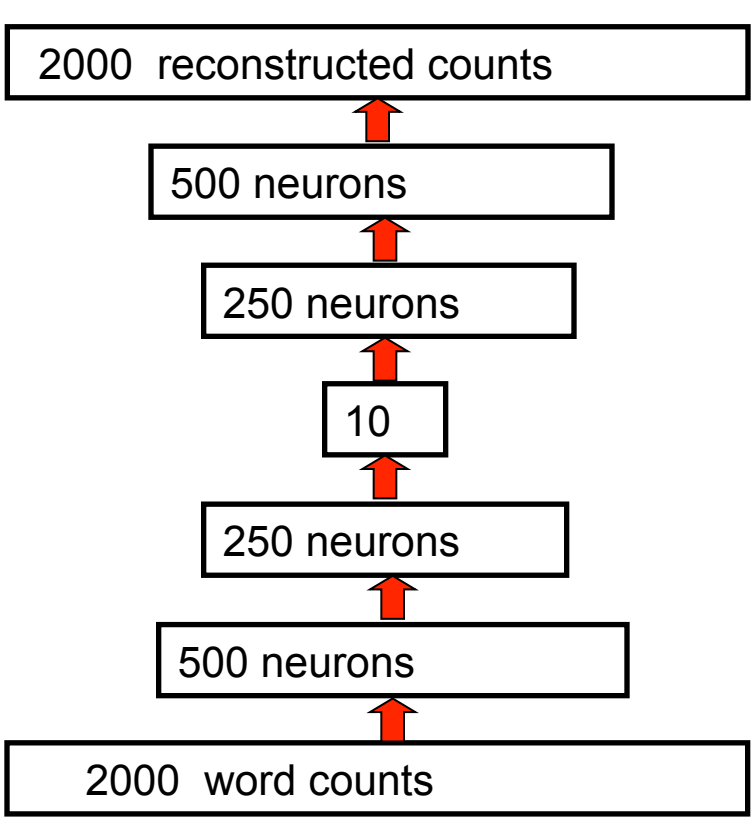
How to find documents that are similar to a query document

- Convert each document into a “bag of words”.
  - This is a vector of word counts ignoring order.
  - Ignore stop words (like “the” or “over”)
- We could compare the word counts of the query document and millions of other documents but this is too slow.
  - So we reduce each query vector to a much smaller vector that still contains most of the information about the content of the document.

0	fish
0	cheese
2	vector
2	count
0	school
2	query
1	reduce
1	bag
0	pulpit
0	iraq
2	word

Slides by Geoffrey Hinton

How to compress the count vector



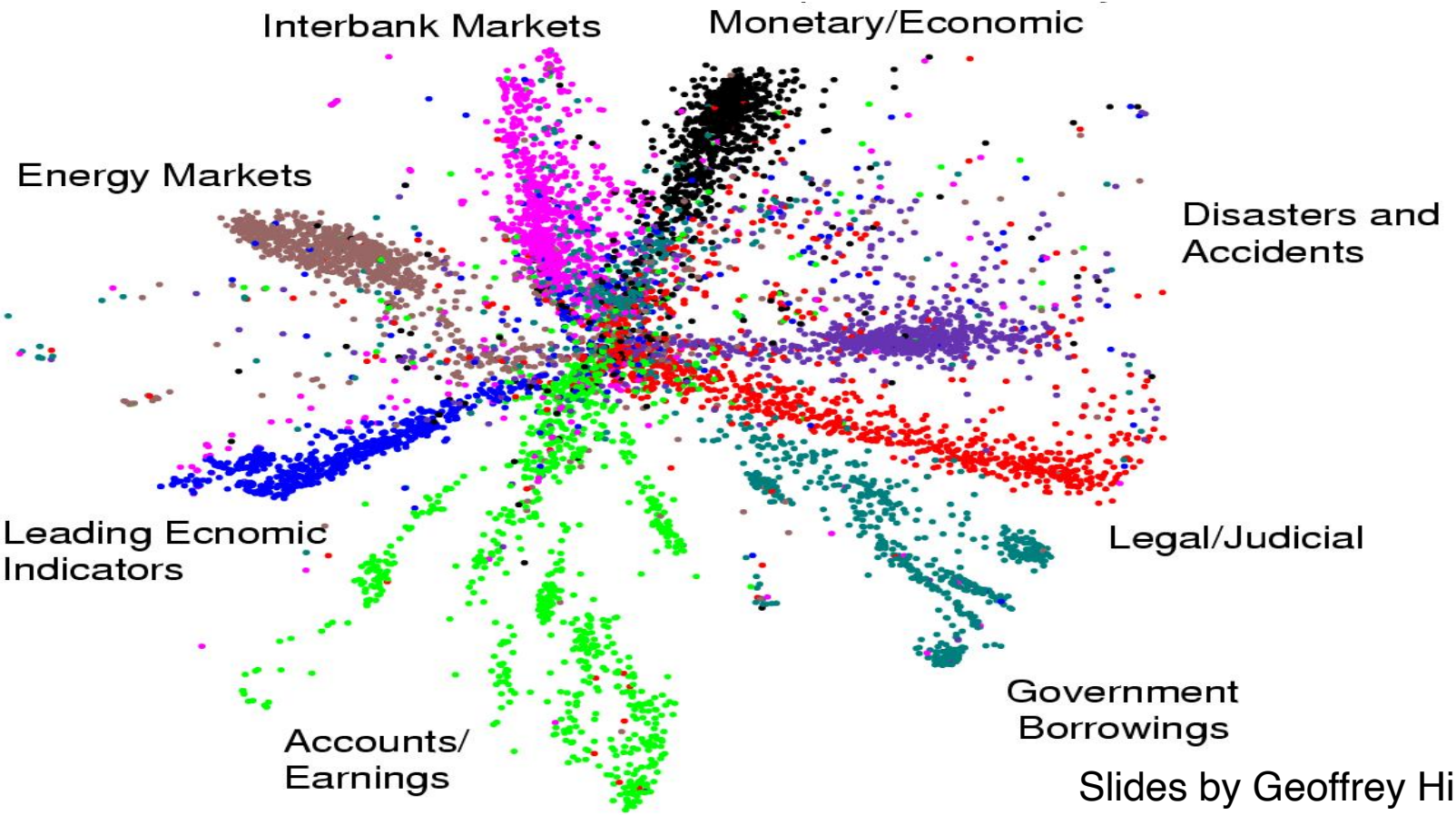
- We train the neural network to reproduce its input vector as its output
- This forces it to compress as much information as possible into the 10 numbers in the central bottleneck.
- These 10 numbers are then a good way to compare documents.

Slides by Geoffrey Hinton

First compress all documents to 2 numbers using PCA on log(1+count). Then use different colors for different categories.

Slides by Geoffrey Hinton

First compress all documents to 2 numbers using deep auto.
Then use different colors for different document categories



Slides by Geoffrey Hinton