Word Embedding

Nguyen Van Vinh

VNU-UET

word2vec: The foundation of NLP

- https://kharshit.github.io/blog/2018/07/27/word2vec-the-basic-of-nlp

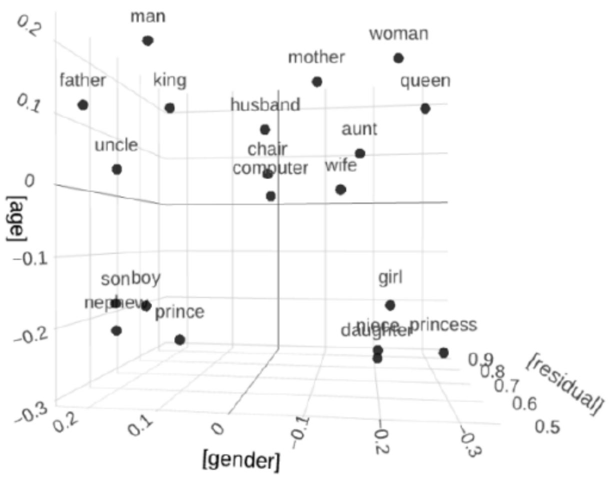
The important idea: model of meaning focusing on similarity

Each word = a vector

$$v_{\text{cat}} = \begin{pmatrix} -0.224 \\ 0.130 \\ -0.290 \\ 0.276 \end{pmatrix} \qquad v_{\text{dog}} = \begin{pmatrix} -0.124 \\ 0.430 \\ -0.200 \\ 0.329 \end{pmatrix}$$

$$v_{\text{the}} = \begin{pmatrix} 0.234 \\ 0.266 \\ 0.239 \\ -0.199 \end{pmatrix} \qquad v_{\text{language}} = \begin{pmatrix} 0.290 \\ -0.441 \\ 0.762 \\ 0.982 \end{pmatrix}$$

Similar words are "**nearby in the vector space**"



(Bandyopadhyay et al. 2022)

Why word meaning in NLP models?

- With words, a feature is a word identity (= string)
  - Feature 5: `The previous word was "terrible"'
  - Requires **exact same word** to be in the training and testing set

"terrible" $\neq$ "horrible"

- If we can represent word meaning in vectors:
  - The previous word was vector [35, 22, 17, …]
  - Now in the test set we might see a similar vector [34, 21, 14, …]
  - We can generalize to **similar but unseen** words!!!

Content

- **Representing words**
- **Word2Vec**
- **Application of Word2Vec**

How do we represent the meaning of a word?

Definition: **meaning** (Webster dictionary)

- the idea that is represented by a word, phrase, etc.
- the idea that a person wants to express by using words, signs, etc.
- the idea that is expressed in a work of writing, art, etc.

How do we have usable meaning in a computer?

Common solution: Use e.g. WordNet, a thesaurus containing lists of **synonym sets** and **hypernyms** ("is a" relationships).

e.g. synonym sets containing "good":

```
from nltk.corpus import wordnet as wn
poses = { 'n':'noun', 'v':'verb', 's':'adj (s)', 'a':'adj', 'r':'adv'}
for synset in wn.synsets("good"):
    print("{}: {}".format(poses[synset.pos()],
          ", ".join([l.name() for l in synset.lemmas()])))
```

```
noun: good
noun: good, goodness
noun: good, goodness
noun: commodity, trade_good, good
adj: good
adj (sat): full, good
adj: good
adj (sat): estimable, good, honorable, respectable
adj (sat): beneficial, good
adj (sat): good
adj (sat): good, just, upright
…
adverb: well, good
adverb: thoroughly, soundly, good
```

e.g. hypernyms of "panda":

```
from nltk.corpus import wordnet as wn
panda = wn.synset("panda.n.01")
hyper = lambda s: s.hypernyms()
list(panda.closure(hyper))
```

```
[Synset('procyonid.n.01'),
Synset('carnivore.n.01'),
Synset('placental.n.01'),
Synset('mammal.n.01'),
Synset('vertebrate.n.01'),
Synset('chordate.n.01'),
Synset('animal.n.01'),
Synset('organism.n.01'),
Synset('living_thing.n.01'),
Synset('whole.n.02'),
Synset('object.n.01'),
Synset('physical_entity.n.01'),
Synset('entity.n.01')]
```

Problems with resources like WordNet

- Great as a resource but missing nuance
    - e.g. “proficient” is listed as a synonym for “good”. This is only correct in some contexts.
- Missing new meanings of words
    - e.g., wicked, badass, nifty, wizard, genius, ninja, bombest
    - Impossible to keep up-to-date!
- Subjective
- Requires human labor to create and adapt
- Can’t compute accurate word similarity →

Representing words as discrete symbols

In traditional NLP, we regard words as discrete symbols:
hotel, conference, motel – a localist representation

Means one 1, the rest 0s

Words can be represented by one-hot vectors:

motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
hotel = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]

Vector dimension = number of words in vocabulary (e.g., 500,000)

Problem with words as discrete symbols

Example: in web search, if user searches for "Seattle motel", we would like to match documents containing "Seattle hotel".

But:

motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
hotel = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]

These two vectors are orthogonal.

There is no natural notion of **similarity** for one-hot vectors!

Solution:

- Could try to rely on WordNet's list of synonyms to get similarity?
  - But it is well-known to fail badly: incompleteness, etc.
- **Instead: learn to encode similarity in the vectors themselves**

Representing words by their context

- Distributional semantics: **A word's meaning is given by the words that frequently appear close-by**

  - *"You shall know a word by the company it keeps"* (J. R. Firth 1957: 11)
  - *"Words which frequently appear in similar contexts have similar meaning"*
  - **One of the most successful ideas of modern statistical NLP!**

- When a word *w* appears in a text, its **context** is the set of words that appear nearby (within a fixed-size window)
- Use the many contexts of *w* to build up a representation of *w*

…government debt problems turning into	***banking***	*crises as happened in 2009…*
…saying that Europe needs unified	***banking***	*regulation to replace the hodgepodge…*
…India has just given its	***banking***	*system a shot in the arm…*

These **context words** will represent ***banking***

Distributed representation

- Vector representation that encodes information about the distribution of contexts a word appears in
- Words that appear in similar contexts have similar representations
- We have several different ways we can encode the notion of “context.

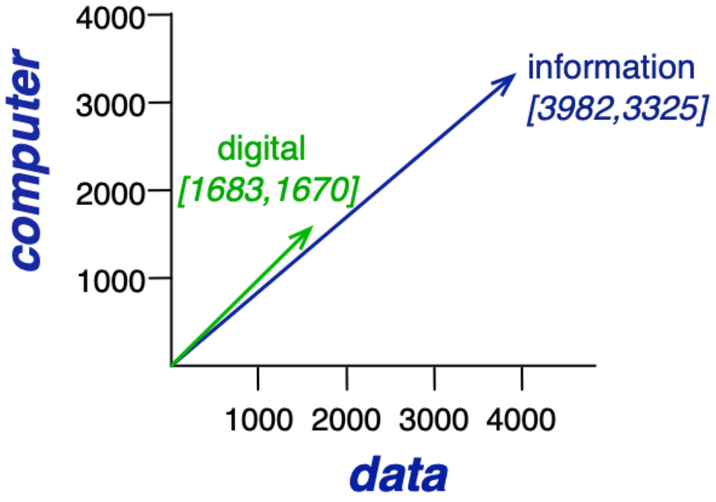
Term-document matrix

- Context = appearing in the same document.

	Hamlet	Macbeth	Romeo & Juliet	Richard III	Julius Caesar	Tempest	Othello	King Lear
knife	1	1	4	2		2		10
dog				6	12	2		
sword	2	2	7	5		5		17
love	64		135	63		12		48
like	75	38	34	36	34	41	27	44

Measuring similarity

$$\cos(x,y) = \frac{\sum_{i=1}^{F} x_i y_i}{\sqrt{\sum_{i=1}^{F} x_i^2}\sqrt{\sum_{i=1}^{F} y_i^2}}$$



- We can calculate the cosine similarity of two vectors to judge the degree of their similarity [Salton 1971]
- Cosine similarity measures their orientation
- A common similarity metric: **cosine** of the angle between the two vectors (the larger, the more similar the two vectors are)

cos(knife, knife)	1
cos(knife, dog)	0.11
cos(knife, sword)	0.99
cos(knife, love)	0.65
cos(knife, like)	0.61

Sparse vs dense vectors

- The vectors in the word-word occurrence matrix are
  - **Long**: vocabulary size
  - **Sparse**: most are 0’s
- Alternative: we want to represent words as **short** (50-300 dimensional) & **dense** (real-valued) vectors
  - The basis for modern NLP systems

$$v_{\text{cat}} = \begin{pmatrix} -0.224 \\ 0.130 \\ -0.290 \\ 0.276 \end{pmatrix} \qquad v_{\text{dog}} = \begin{pmatrix} -0.124 \\ 0.430 \\ -0.200 \\ 0.329 \end{pmatrix}$$

$$v_{\text{the}} = \begin{pmatrix} 0.234 \\ 0.266 \\ 0.239 \\ -0.199 \end{pmatrix} \qquad v_{\text{language}} = \begin{pmatrix} 0.290 \\ -0.441 \\ 0.762 \\ 0.982 \end{pmatrix}$$

Word vectors

- We will build a **dense vector** for each word, chosen so that it is similar to vectors of words that appear in similar contexts

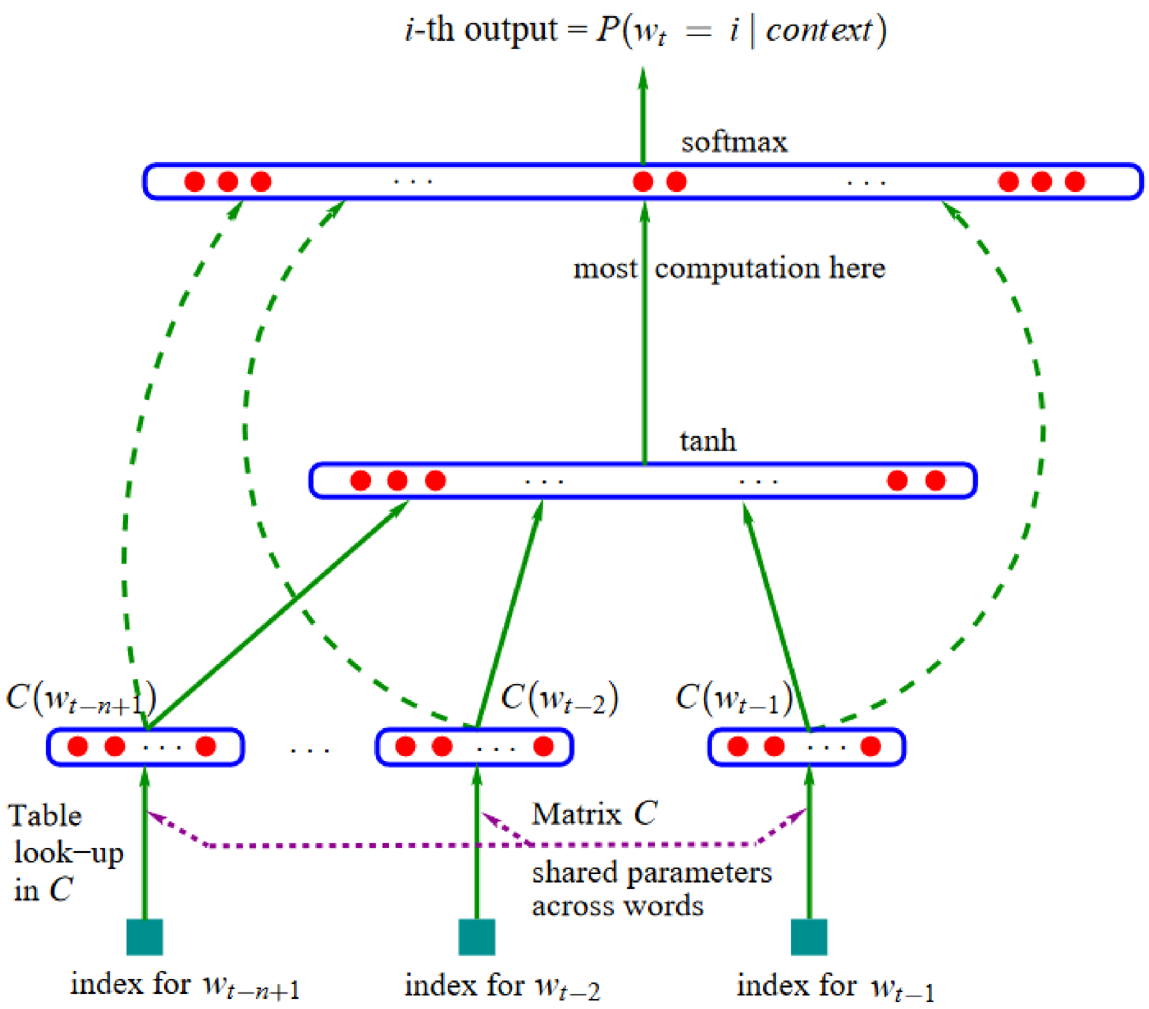
$$\textit{banking} = \begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \end{pmatrix}$$

- **Note**: **word vectors** are sometimes called **word embeddings** or **word representations**. They are a distributed representation.

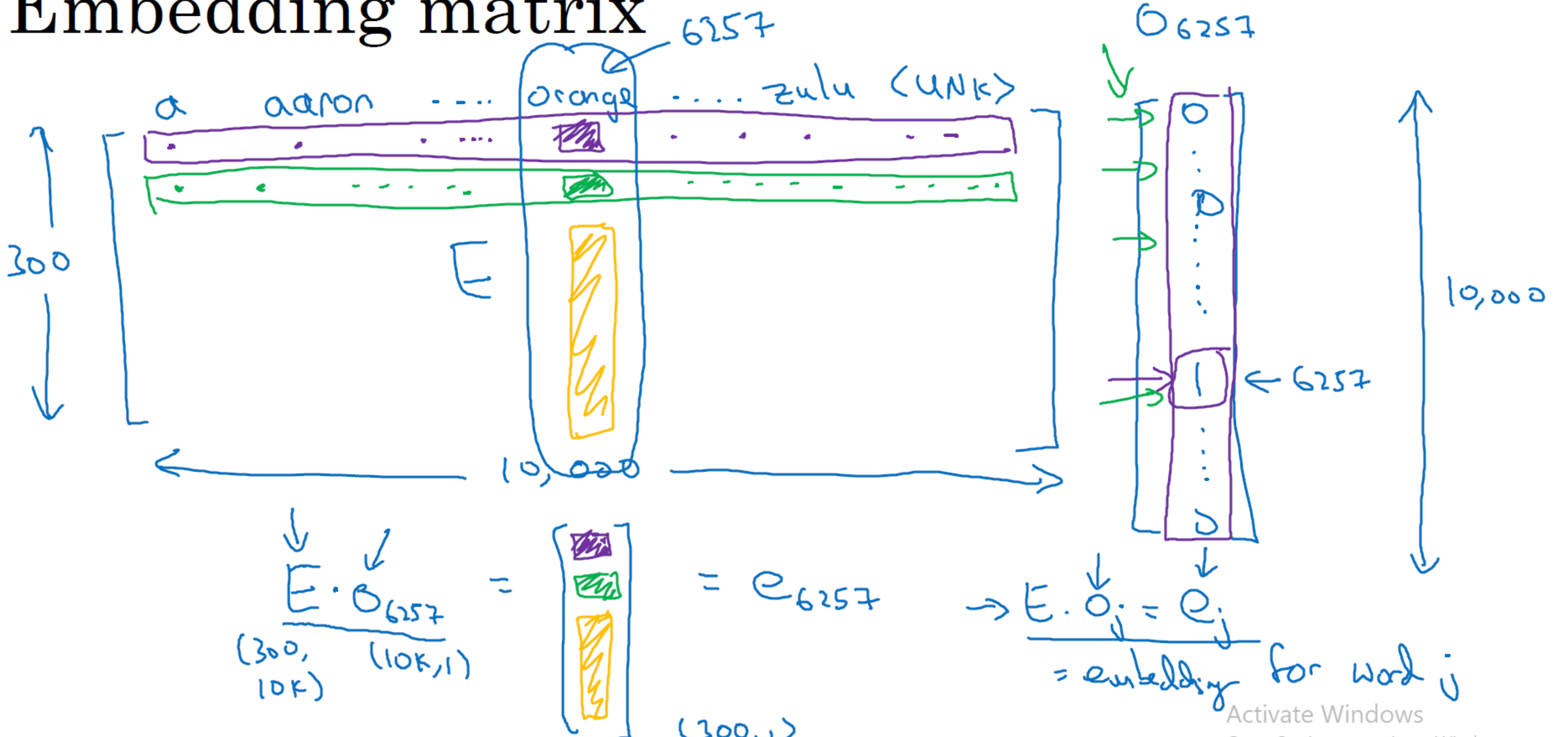
Word embeddings: idea

- **Idea:** We have to put information about contexts into word vectors.
- **How:** Learn word vectors by teaching them to predict contexts
- **Prior work:**
  - Learning representations by back-propagating errors. (Rumelhart et al., 1986)
  - A neural probabilistic language model (**Bengio et al., 2003**)
  - NLP (almost) from Scratch (Collobert & Weston, 2008)
  - A recent, even simpler and faster model: word2vec (Mikolov et al. 2013)

Language model based on neural networks (Bengio et al., 2003)



Embedding matrix



In practice, use specialized function to look up an embedding.

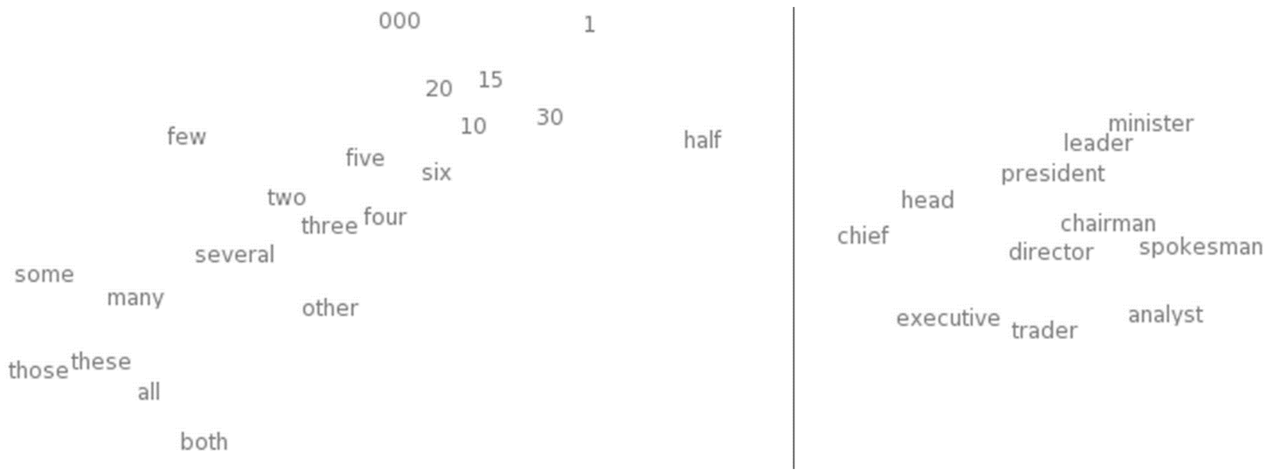
→Embedding

Andrew Ng

Source: https://cs230.stanford.edu/files/C5M2.pdf

Word embeddings: similarity

- Hope to have similar words nearby



Word embeddings: relationships

- Hope to preserve some language structure (relationships between words).

Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

Word embeddings: questions

- How big should the embedding space be?
  - Trade-offs like any other machine learning problem – greater capacity versus efficiency and overfitting.

- How do we find W matrix (vectors)?
  - Often as part of a prediction or classification task involving neighboring words.

Word2vec: Overview

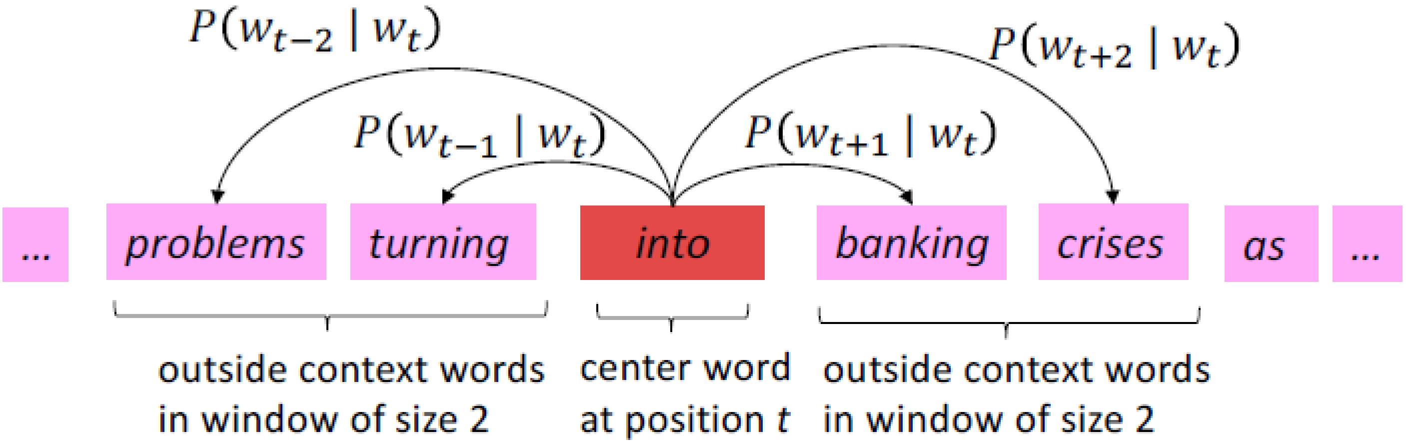
- Word2vec (Mikolov et al. 2013) is a framework for learning word vectors

Idea:

- We have a large corpus of text
- Every word in a fixed vocabulary is represented by a vector
- Go through each position t in the text, which has a center word c and context ("outside") words o
- Use the similarity of the word vectors for c and o to calculate the probability of o given c (or vice versa)
- Keep adjusting the word vectors to maximize this probability

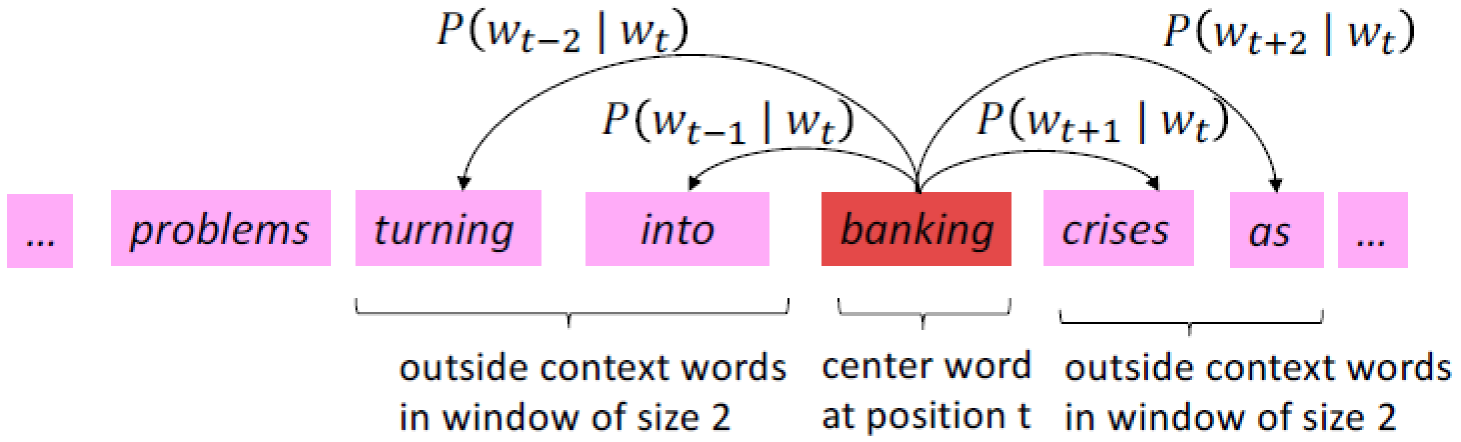
Word2Vec Overview

- Example windows and process for computing P(Wt+j|Wt)



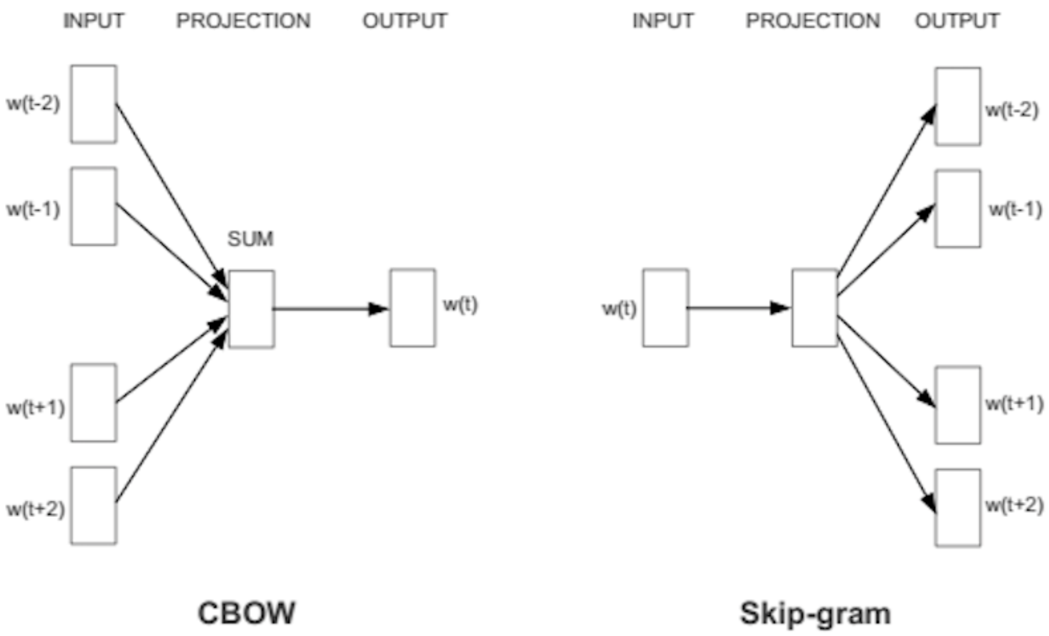
Word2Vec Overview

- Example windows and process for computing $P(W_{t+j}|W_t)$



word2vec

- Predict words using context
- Two versions: CBOW (continuous bag of words) and Skip-gram



https://skymind.ai/wiki/word2vec

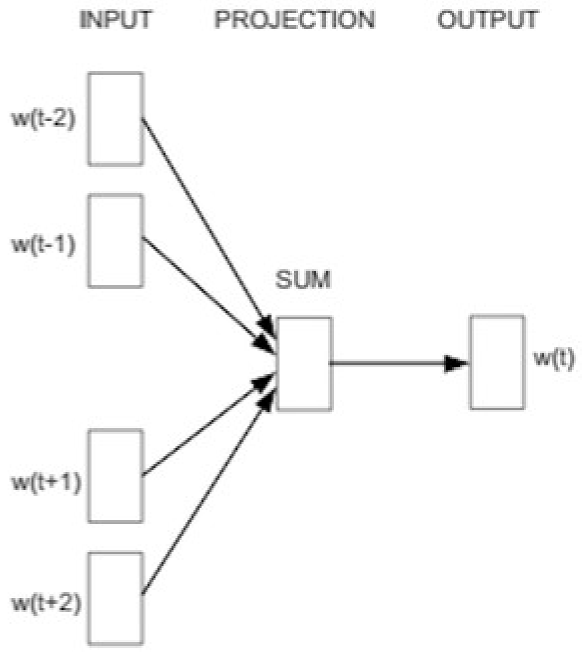
Skip gram/CBOW intuition

- Similar “contexts” (that is, what words are likely to appear around them), lead to similar embeddings for two words.
- One way for the network to output similar context predictions for these two words is if *the word vectors are similar*. So, if two words have similar contexts, then the network is motivated to learn similar word vectors for these two words!

http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

CBOW

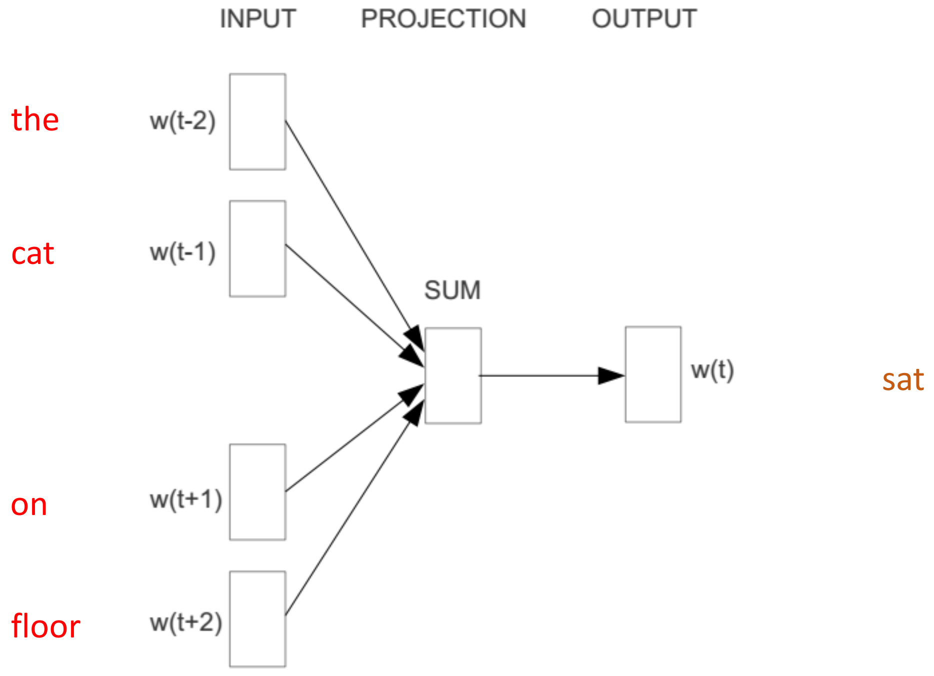
- Bag of words
  - Gets rid of word order. Used in discrete case using counts of words that appear.
- CBOW
  - Takes vector embeddings of n words before target and n words after and adds them (as vectors).
  - Also removes word order, but the vector sum is meaningful enough to deduce missing word.



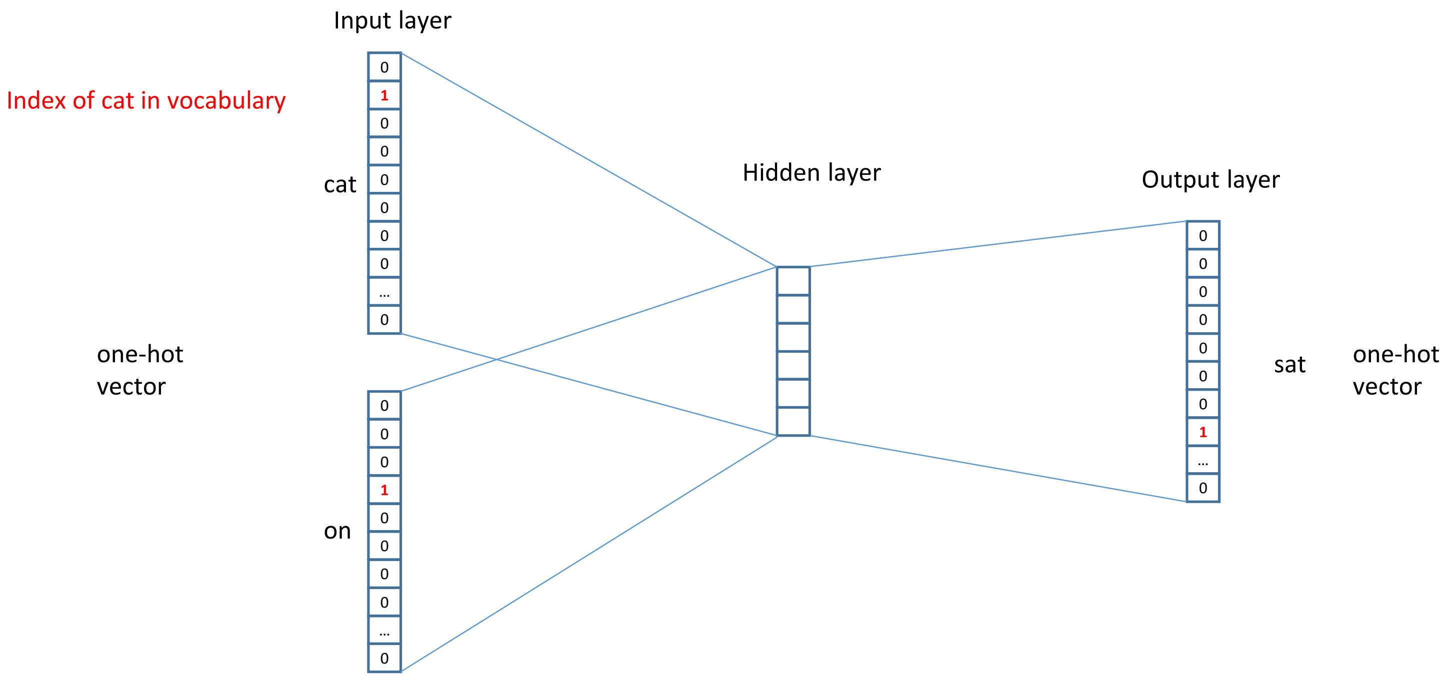
CBOW

Word2vec – Continuous Bag of Word

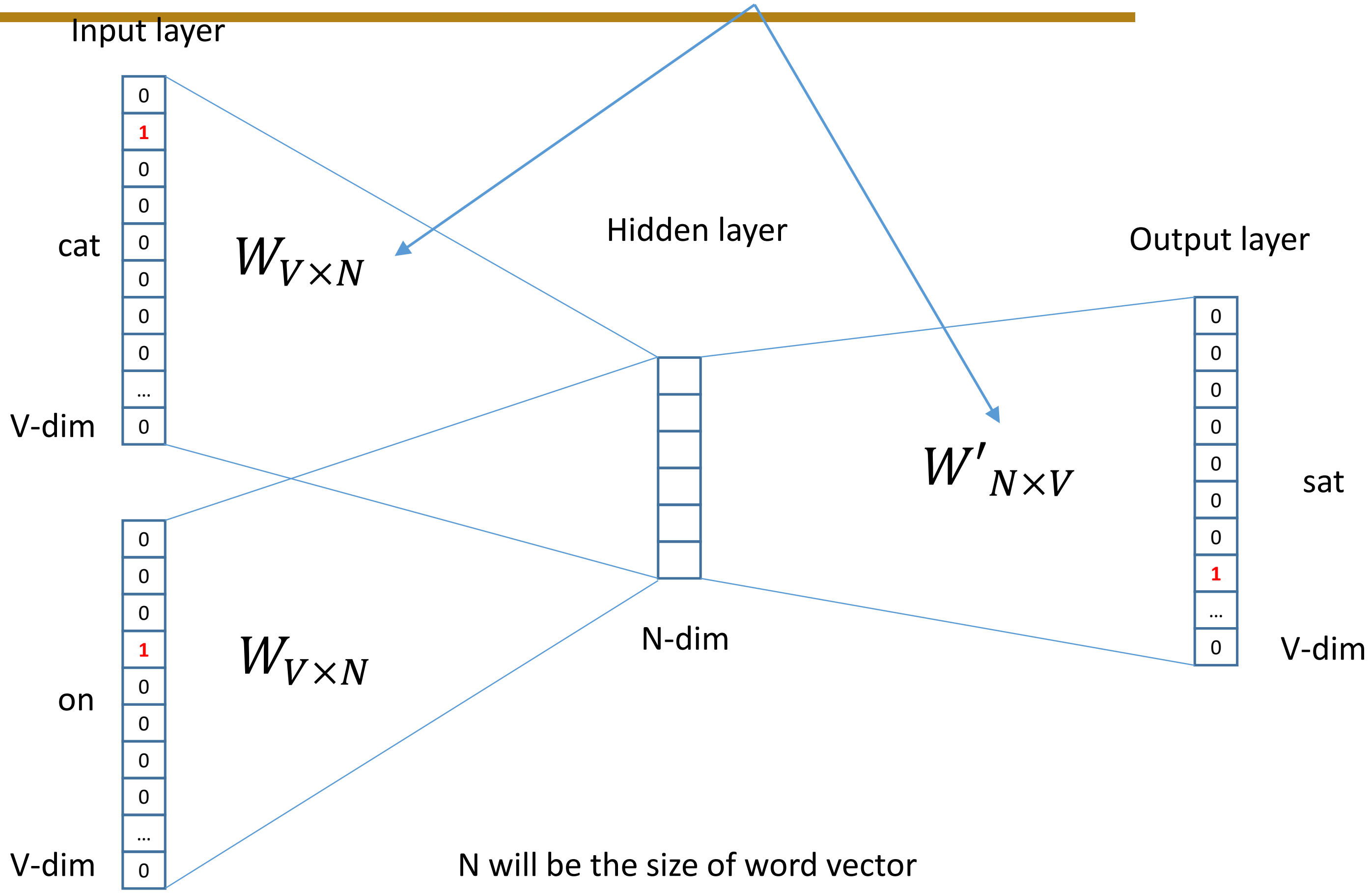
- E.g. "The cat sat on floor"
  - Window size = 2

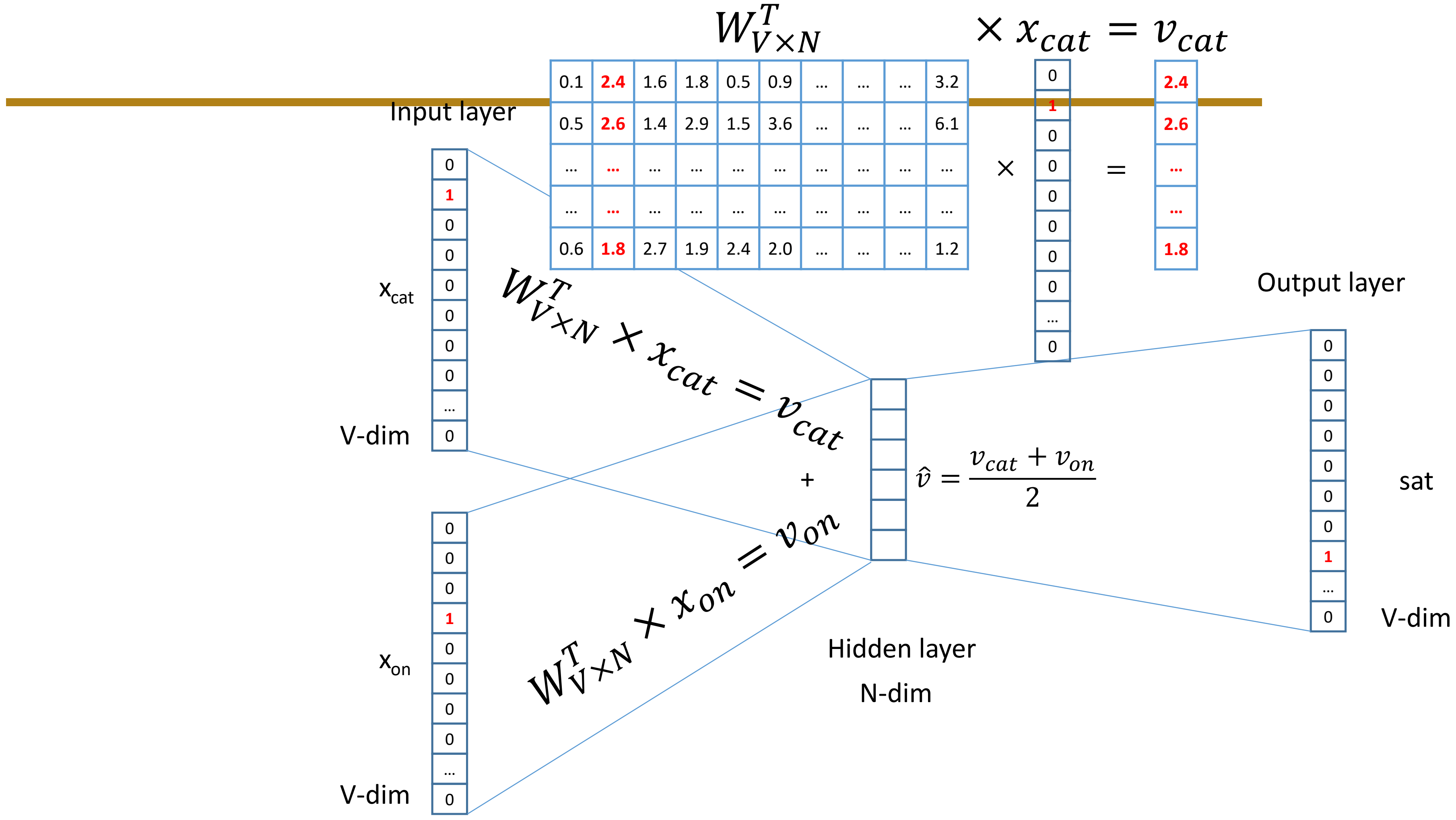


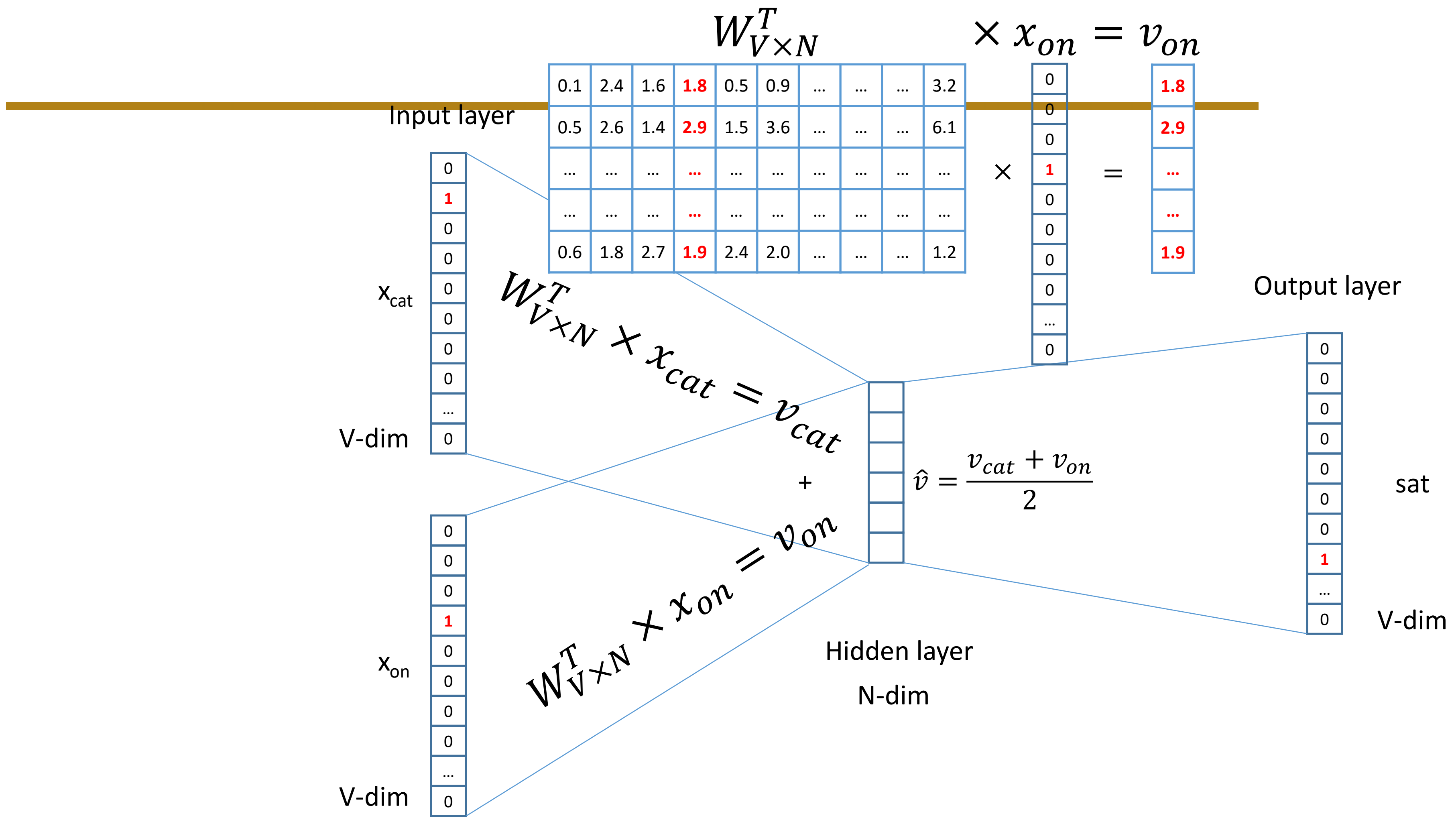
www.cs.ucr.edu/~vagelis/classes/CS242/slides/word2vec.pptx


Input layer
Index of cat in vocabulary
cat
0
1
0
0
0
0
0
0
...
0
one-hot
vector
on
0
0
0
1
0
0
0
0
...
0
Hidden layer
Output layer
0
0
0
0
0
0
0
1
...
0
sat
one-hot
vector

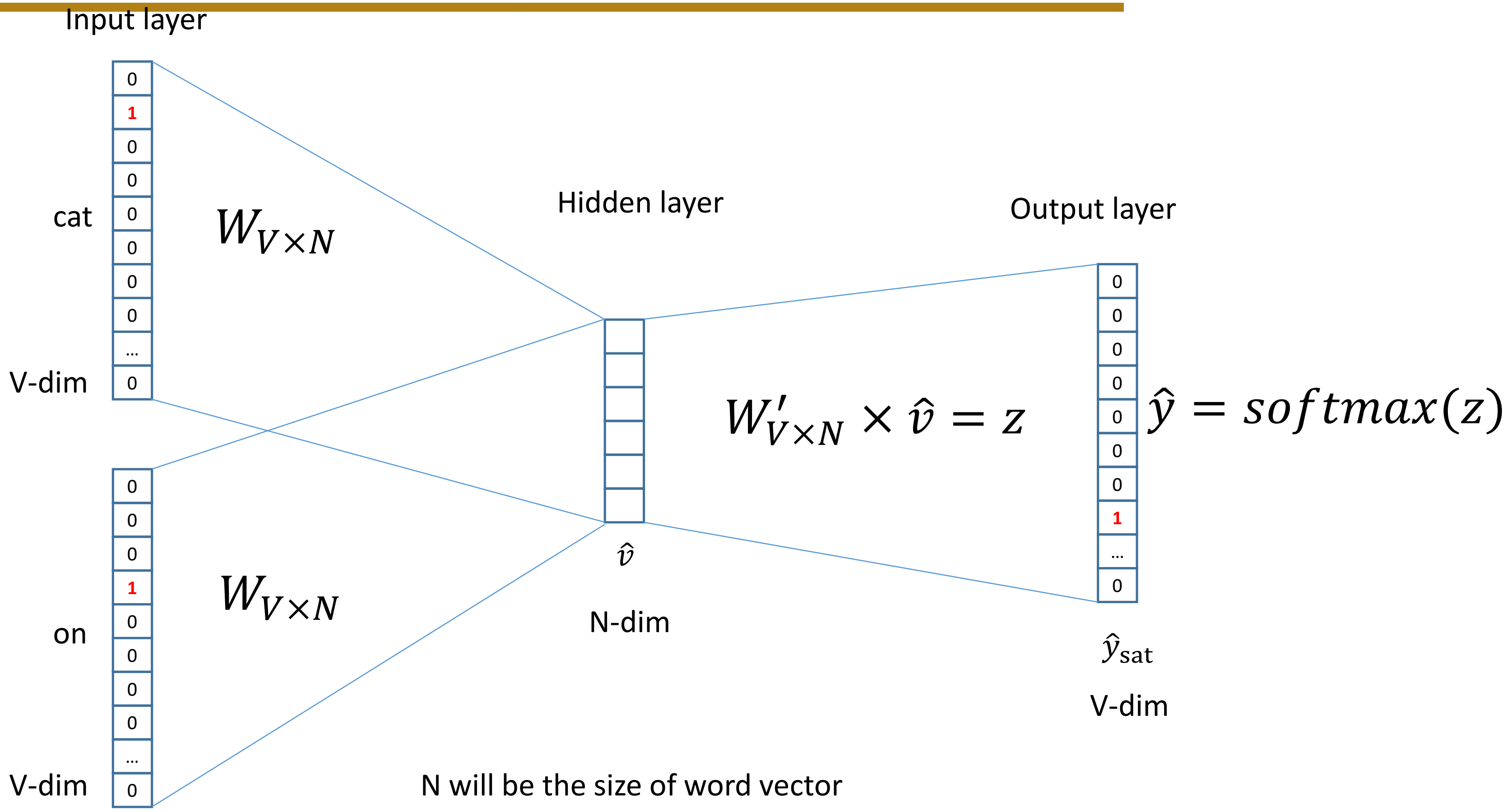
We must learn W and W′



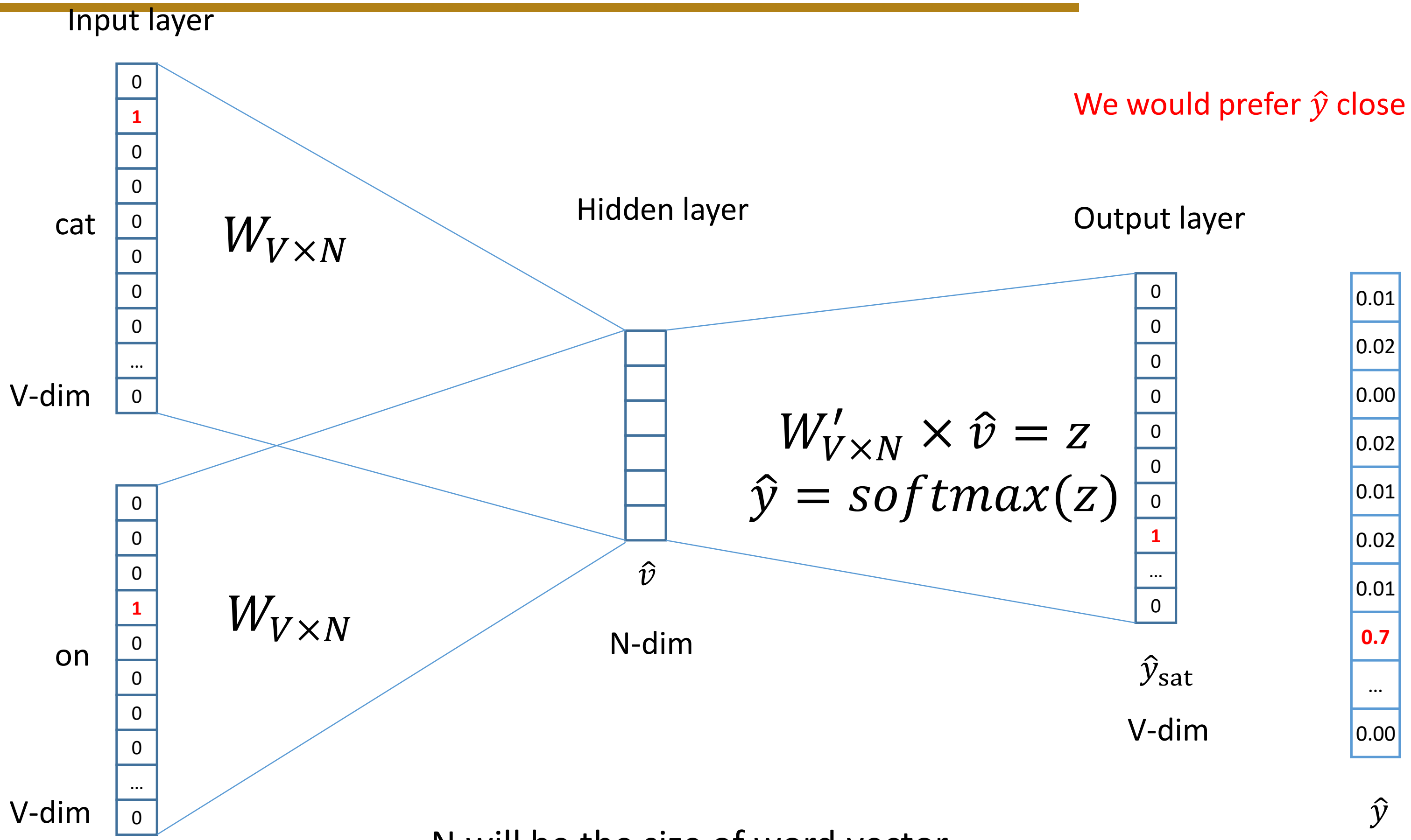

$W^T_{V\times N} \times x_{cat} = v_{cat}$
Input layer
0.1 2.4 1.6 1.8 0.5 0.9 … … … 3.2
0.5 2.6 1.4 2.9 1.5 3.6 … … … 6.1
0.6 1.8 2.7 1.9 2.4 2.0 … … … 1.2
2.4
2.6
1.8
x_{cat}
V-dim
$W^T_{V\times N} \times x_{cat} = v_{cat}$
+
$W^T_{V\times N} \times x_{on} = v_{on}$
$\hat{v} = \frac{v_{cat} + v_{on}}{2}$
Output layer
sat
V-dim
x_{on}
V-dim
Hidden layer
N-dim



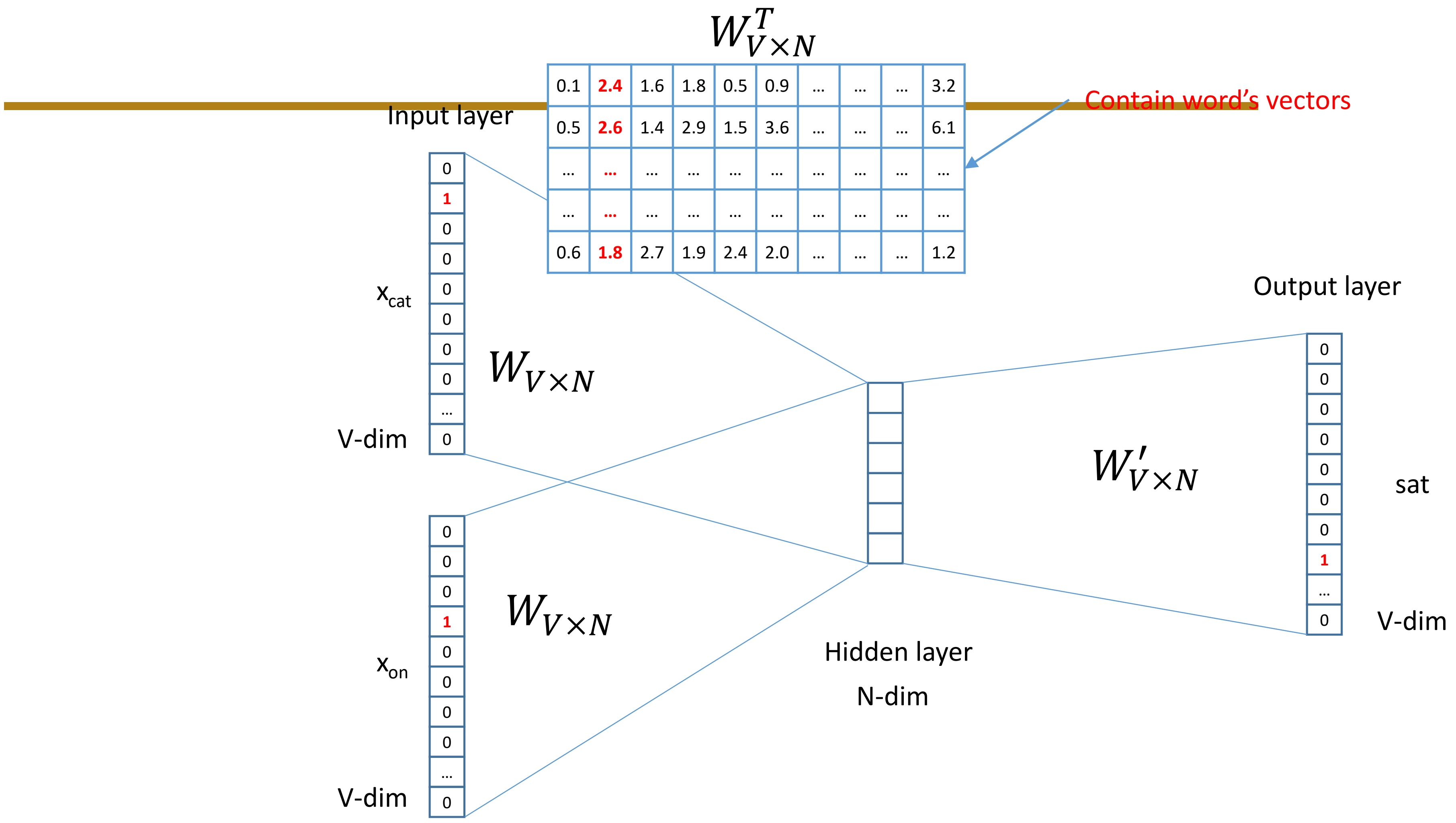
$W_{V\times N}^{T}$
$\times\ x_{on} = v_{on}$
0.1 2.4 1.6 1.8 0.5 0.9 … … … 3.2
0.5 2.6 1.4 2.9 1.5 3.6 … … … 6.1
0.6 1.8 2.7 1.9 2.4 2.0 … … … 1.2
1.8
2.9
1.9
Input layer
Output layer
x_{cat}
V-dim
x_{on}
V-dim
$W_{V\times N}^{T} \times x_{cat} = v_{cat}$
+
$W_{V\times N}^{T} \times x_{on} = v_{on}$
$\hat{v} = \frac{v_{cat} + v_{on}}{2}$
Hidden layer
N-dim
sat
V-dim


Input layer
cat
V-dim
on
V-dim
$W_{V\times N}$
$W_{V\times N}$
Hidden layer
$\hat{v}$
N-dim
$W'_{V\times N} \times \hat{v} = z$
Output layer
$\hat{y} = softmax(z)$
$\hat{y}_{\text{sat}}$
V-dim
N will be the size of word vector

We would prefer $\hat{y}$ close to $\hat{y}_{sat}$



N will be the size of word vector



We can consider either W or W' as the word's representation. Or even take the average.

Details

$$\text{Loss} = J(\theta) = -\frac{1}{T}\log L(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{-m\le j\le m, j\ne 0}\log P(w_{t+j}|w_t,\theta) = \frac{1}{T}\sum_{t=1}^{T}\sum_{-m\le j\le m, j\ne 0} J_{t,j}(\theta).$$

... I saw a cute grey cat playing in the garden ...

with the central word cat, and four context words. Since we are going to look at just one step, we will pick only one of the context words; for example, let's take cute. Then the loss term for the central word cat and the context word cute is:

$$J_{t,j}(\theta) = -\log P(cute|cat) = -\log\frac{\exp u_{cute}^T v_{cat}}{\sum\limits_{w\in Voc}\exp u_w^T v_{cat}} = -u_{cute}^T v_{cat} + \log\sum_{w\in Voc}\exp u_w^T v_{cat}.$$

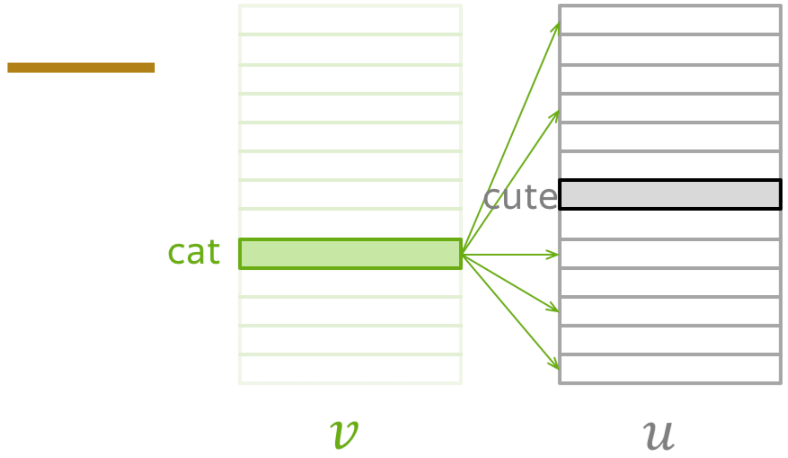
Note which parameters are present at this step:

- from vectors for central words, only v_{cat};
- from vectors for context words, all u_w (for all words in the vocabulary).

1. Take dot product of v_{cat} with **all** u

2. exp

3. sum all



$$u_{w1}^T v_{cat} \rightarrow \exp(u_{w1}^T v_{cat})$$

$$u_{w3}^T v_{cat} \rightarrow \exp(u_{w3}^T v_{cat})$$

$$\vdots$$

$$\boxed{u_{cute}^T v_{cat}} \rightarrow \exp(u_{cute}^T v_{cat})$$

1

$$\vdots$$

$$u_{wn}^T v_{cat} \rightarrow \exp(u_{wn}^T v_{cat})$$

$$\boxed{\sum_{w \in V} \exp(u_w^T v_{cat})}$$

2

4. get loss (for this one step)

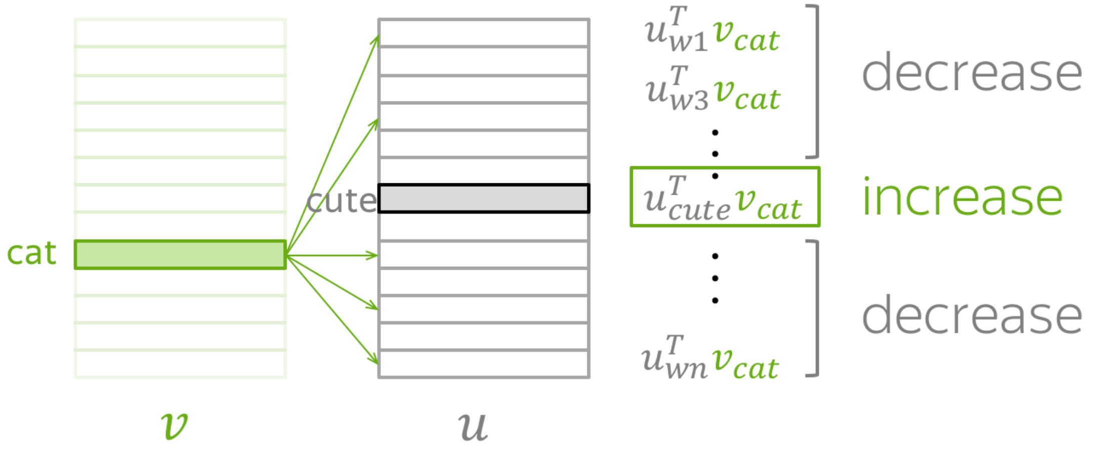
$$J_{t,j}(\theta) = \underbrace{-u_{cute}^T v_{cat}}_{1} + \underbrace{\log \sum_{w \in V} \exp(u_w^T v_{cat})}_{2}$$

5. evaluate the gradient, make an update

$$v_{cat} := v_{cat} - \alpha \frac{\partial J_{t,j}(\theta)}{\partial v_{cat}}$$

$$u_w := u_w - \alpha \frac{\partial J_{t,j}(\theta)}{\partial u_w} \quad \forall\, w \in V$$

By making an update to minimize $J_{t,j}(\theta)$, we force the parameters to <u>increase</u> similarity (dot product) of v_{cat} and u_{cute} and, at the same time, to <u>decrease</u> similarity between v_{cat} and u_w for all other words w in the vocabulary.



Some interesting results

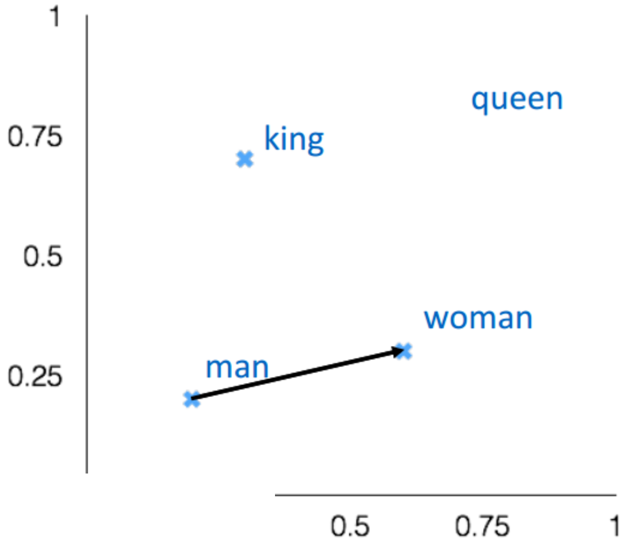
Word Analogies

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

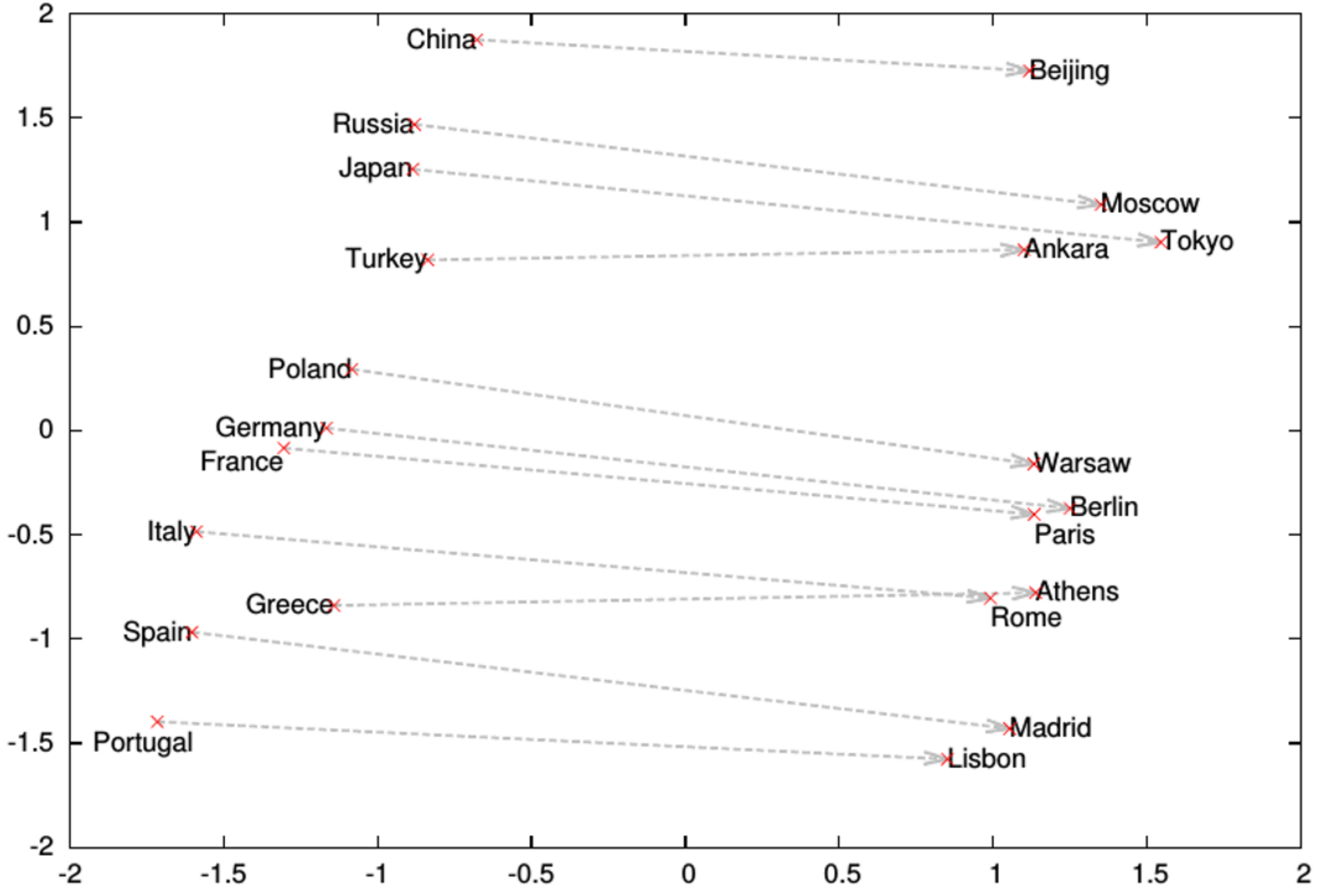
a:b :: c:? $\longrightarrow$ $$d = \arg\max_x \frac{(w_b - w_a + w_c)^T w_x}{||w_b - w_a + w_c||}$$

man:woman :: king:?

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



Word analogies



Word embeddings: properties

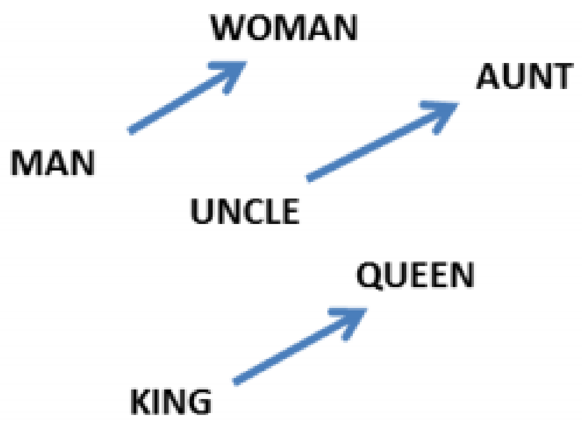
- Need to have a function W(word) that returns a vector encoding that word.

- Similarity of words corresponds to nearby vectors.
  - Director – chairman, scratched – scraped

- Relationships between words correspond to difference between vectors.
  - Big – bigger, small – smaller

Word embeddings: properties

- Relationships between words correspond to difference between vectors.



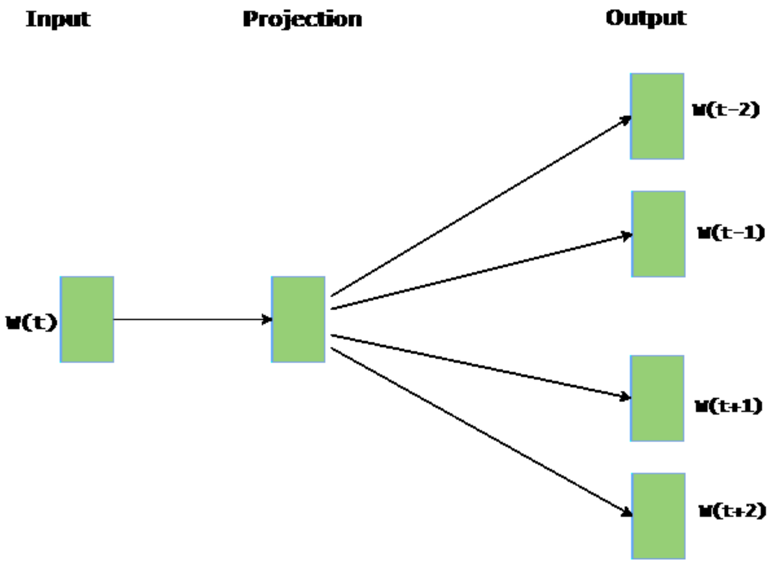
$$W(\text{``woman''}) - W(\text{``man''}) \simeq W(\text{``aunt''}) - W(\text{``uncle''})$$

$$W(\text{``woman''}) - W(\text{``man''}) \simeq W(\text{``queen''}) - W(\text{``king''})$$

http://colah.github.io/posts/2014-07-NLP-RNNs-Representations/

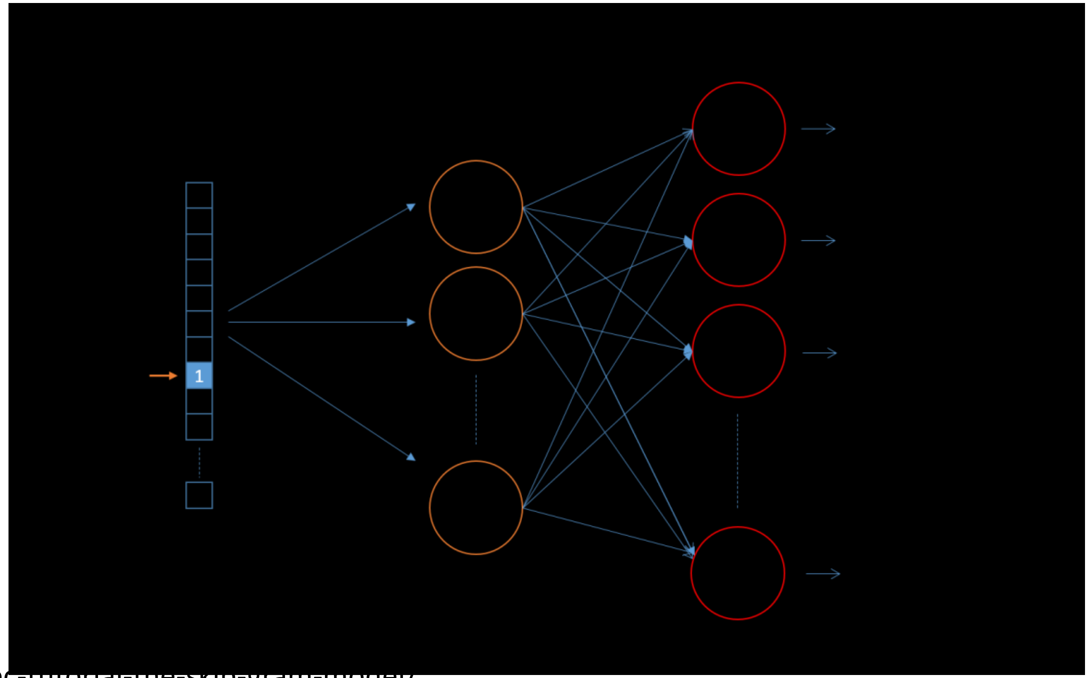
Skip gram

- Skip gram – alternative to CBOW
  - Start with a single word embedding and try to predict the context words (surrounding words).
  - Skip-Gram works well with small datasets, and can better represent less frequent words.



Skip gram

- Map from center word to probability on surrounding words. One input/output unit below.
  - There is no activation function on the hidden layer neurons, but the output neurons use softmax.

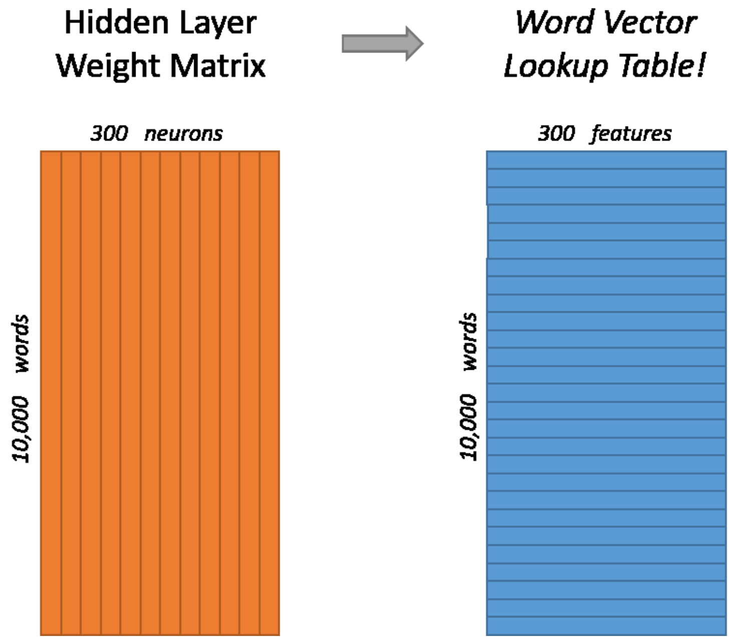


http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

Skip gram example

- Vocabulary of 10,000 words.
- Embedding vectors with 300 features.
- So the hidden layer is going to be represented by a weight matrix with 10,000 rows (multiply by vector on the left).

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ 10 & 12 & 19 \\ 11 & 18 & 25 \end{bmatrix} = \begin{bmatrix} 10 & 12 & 19 \end{bmatrix}$$



http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

Word2vec shortcomings

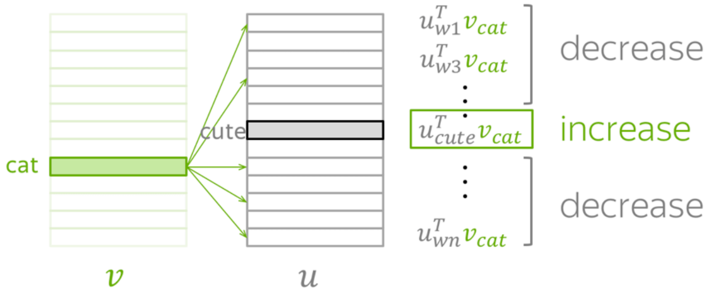
- **Problem:** 10,000 words and 300 dim embedding gives a large parameter space to learn. And 10K words is minimal for real applications.

- **Slow to train**, and need lots of data, particularly to learn uncommon words.
- Both of these layers would have a weight matrix with 300 x 10,000 = 3 million weights each

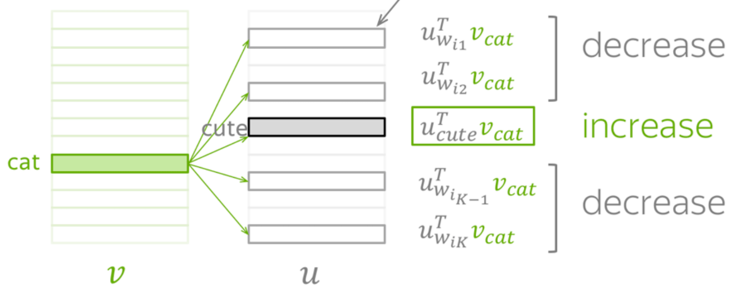
Faster Training: Negative Sampling

Dot product of v_{cat}:
- with u_{cute} - increase,
- with **all other** u - decrease

Dot product of v_{cat}:
- with u_{cute} - increase,
- with **a subset of other** u - decrease



Negative samples: randomly selected K words



Parameters to be updated:
- v_{cat}
- u_w for all w in the vocabulary

|V| + 1 vectors

Parameters to be updated:
- v_{cat}
- u_{cute} and u_w for w in K negative examples

K + 2 vectors

Defining a new learning problem

I want a glass of orange juice to go along with my cereal.

x → y

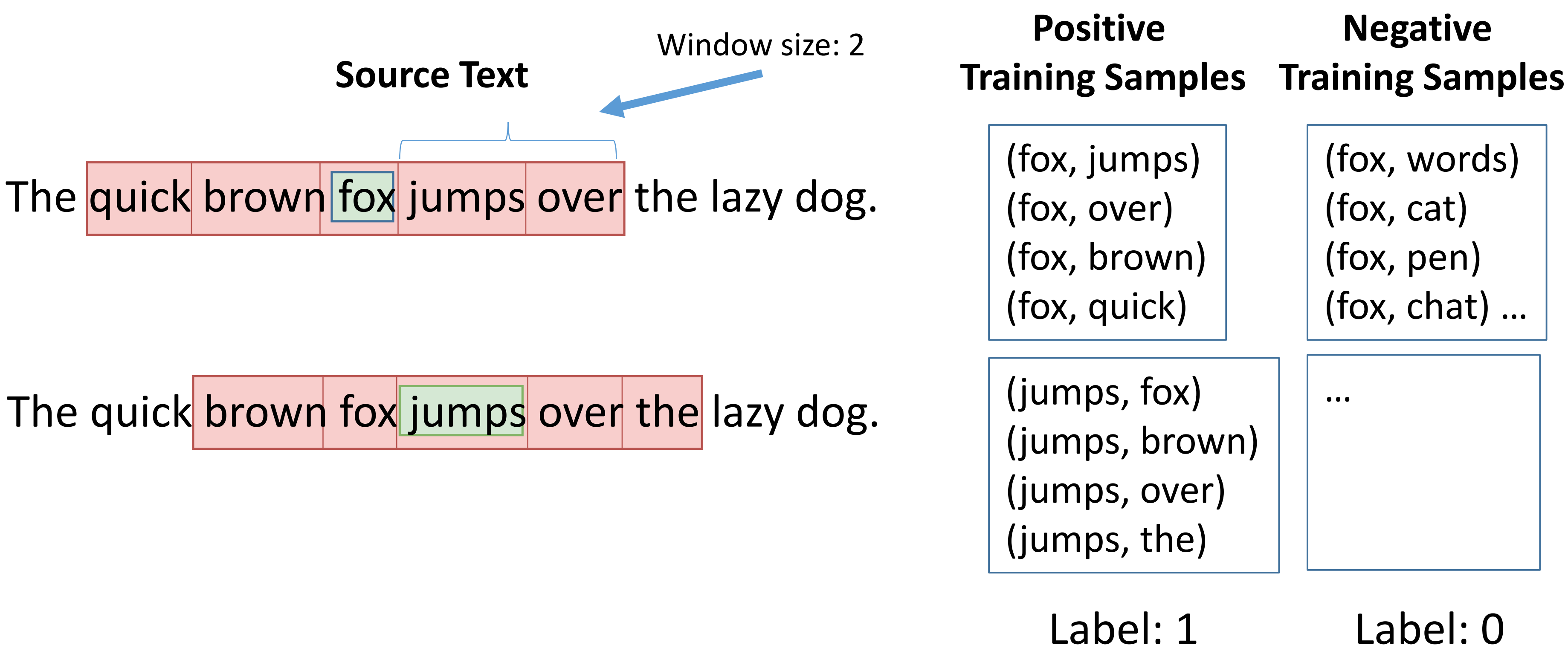
Context	word	target?
orange	juice	1
orange	king	0
orange	book	0
orange	the	0
orange	of	0

k

k = 5-20 smaller datasets

k = 2-5 larger dataset

Word2vec improvements: selective updates



Word2vec improvements: selective updates

- Selecting 5-20 words works well for smaller datasets, and you can get away with only 2-5 words for large datasets.
- Updating the weights for our positive word ("*jump*"), plus the weights for 5 other words that we want to output 0. That's a total of 6 output neurons, and 1,800 weight values total. That's only **0.06% of the 3M weights** in the output layer!

The Glove Model (2014)

- GloVe observes that ratios of word-word co-occurrence probabilities have the potential for encoding some form of meaning. To consider the co-occurrence probabilities for target words ***ice*** and ***steam*** with various probe words from the vocabulary:
  - As one might expect, **ice** co-occurs more frequently with **solid** than it does with **gas**, whereas **steam** co-occurs more frequently with **gas** than it does with **solid**.
  - Both words co-occur with their shared property **water** frequently, and both co-occur with the unrelated word **fashion** infrequently.

Word2Vec vs. GloVe

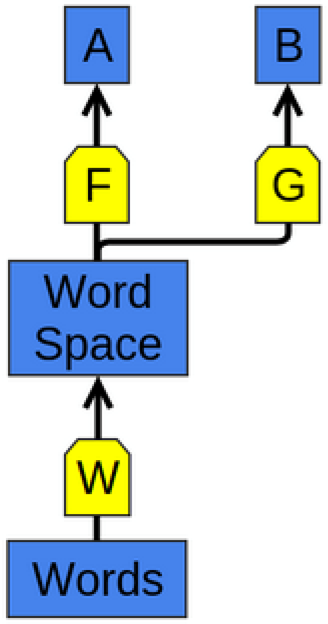
- The advantage of GloVe is that, unlike Word2vec, GloVe does not rely just on local statistics (local context information of words), but incorporates global statistics (word co-occurrence) to obtain word vectors.

FastText vs. Word2Vec

- Fasttext invented by Facebook (2016)
- Fastext solving unknown word problem.
- FastText operates at a more granular level with character n-grams. Where words are represented by the sum of the character n-gram vectors.

Word embedding applications

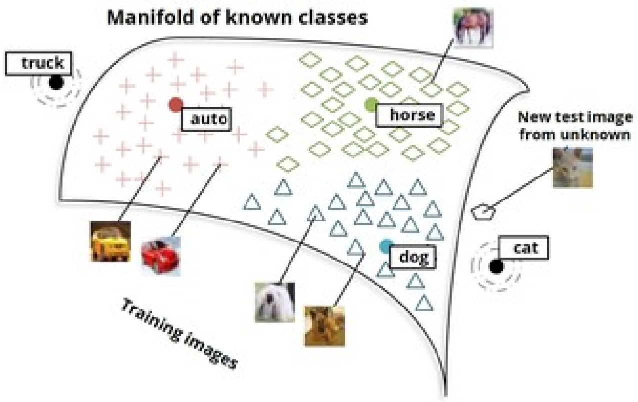
- The use of word representations… has become a key “secret sauce” for the success of many NLP systems in recent years, across tasks including named entity recognition, part-of-speech tagging, parsing, and semantic role labeling. ([Luong *et al.* (2013)](#))
- Learning a good representation on a task A and then using it on a task B is one of the major tricks in the Deep Learning toolbox.
  - Pretraining, transfer learning, and multi-task learning, **Large Language Models**.
  - Can allow the representation to learn from more than one kind of data.
- StarSpace: Embed All The Things (Ledell Wu et al, 2017)



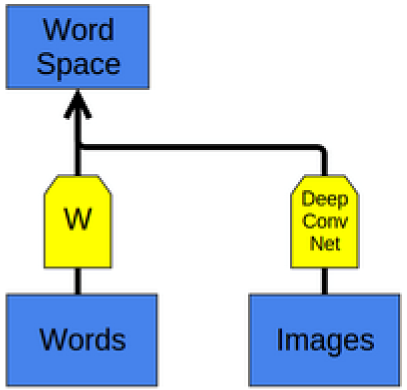
W and F learn to perform task A. Later, G can learn to perform B based on W.

Word embedding applications

- Can apply to get a joint embedding of words and images or other multi-modal data sets.
- New classes map near similar existing classes: e.g., if ‘cat’ is unknown, cat images map near dog.



(Socher *et al.* (2013b))



Conclusion

- Representing words
- Word2vec
  - Skip-gram
  - CBOW
- Application of word2vec

References

- Slide of NLP Course, Stanford, 2022
- Slide of NLP Course, Princeton, 2023
- Other documents on Internet.