

# Distributed Representation

SLP 6.8. Word2Vec

<https://web.stanford.edu/~jurafsky/slp3/6.pdf>

# Outline

- Sparse vs. Dense representations
- Word2Vec
- Skip-Gram algorithm

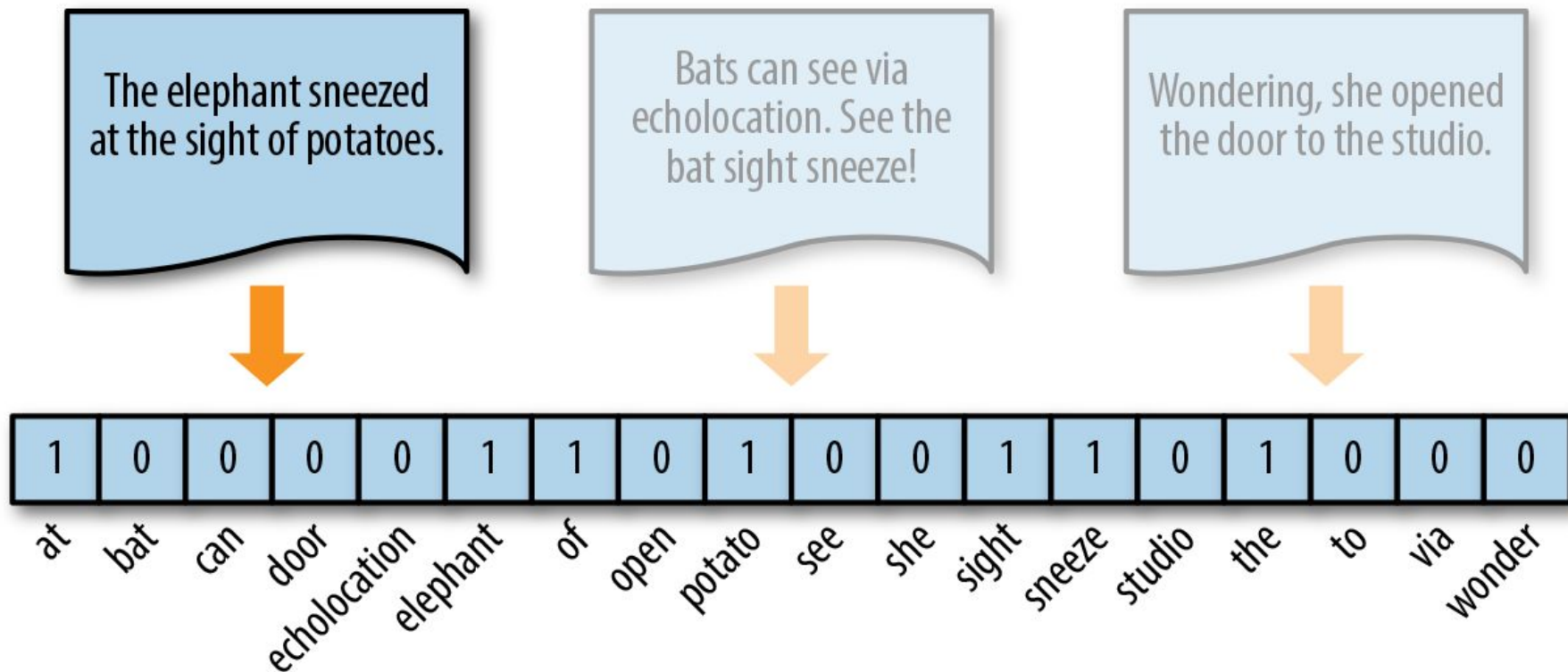
# Tf-idf and PPMI are sparse representations

- long (length  $|V|$  = 20,000 to 50,000)
- sparse (most elements are zero)

# Why dense vectors?

- Easier to use short vectors as features in machine learning
- Dense vectors may generalize better than storing explicit counts
- May do better at capturing synonymy
- In practice, they work better

# One-hot encoding



# Dense vector

The elephant sneezed  
at the sight of potatoes.



Bats can see via  
echolocation. See the  
bat sight sneeze!



Wondering, she opened  
the door to the studio.



-0.0225403

-0.0212964

0.02708783

0.0049877

0.0492694

-0.03268785

-0.0320941

# Dense embeddings you can download!

- Word2vec (Mikolov et al.)
  - <https://code.google.com/archive/p/word2vec/>
- Fasttext
  - <http://www.fasttext.cc/>
- Glove (Pennington, Socher, Manning)
  - <http://nlp.stanford.edu/projects/glove/>
- Elmo
- BERT
- XLNet

# Word2vec (Mikolov, 2013)

- A group of models that tries to represent each word in a large text as a vector
- Popular embedding method
- Very fast to train
- **Idea: predict rather than count**
- Instead of counting how often each word occurs near "apricot"
- Train a classifier on a **binary prediction task**:
  - Is  $w$  likely to show up near "apricot"?
- We don't actually care about this task
- But we'll **take the learned classifier weights as the word embeddings**



# Use text as implicitly supervised training data!

- A word **s** near *apricot*
  - Acts as gold 'correct answer' to the question
  - "Is word *w* likely to show up near apricot?"
- No need for hand-labeled supervision
- The idea comes from **neural language modeling**
  - Bengio et al. (2003)
  - Collobert et al. (2011)
- We will learn "skip-gram with negative sampling" (SGNS)

# Skip-gram algorithm

word2vec is a much simpler model than the neural network language model

- simplifies the task (binary classification instead of word prediction)
  - simplifies the architecture (training a logistic regression classifier instead of a multi-layer neural network with hidden layers)
1. Treat the target word and a neighboring context word as **positive examples**
  2. Randomly sample other words in the lexicon to get **negative samples**
  3. Use logistic regression to train a **classifier to distinguish those two cases**
  4. Use the **weights as the embeddings**

# Skip-Gram Training Data

Training sentence:

... lemon, a tablespoon of apricot jam a pinch ...

c1 c2 target c3 c4

Assume context words are those in +/- 2 word window

# Skip-Gram Goal

- Given a tuple (**t**, **c**) = target, context

(**apricot**, **jam**)

(**apricot**, **aardvark**)

- Return probability that c is a real context word:

$$P(+|t, c)$$

$$P(-|t, c) = 1 - P(+|t, c)$$

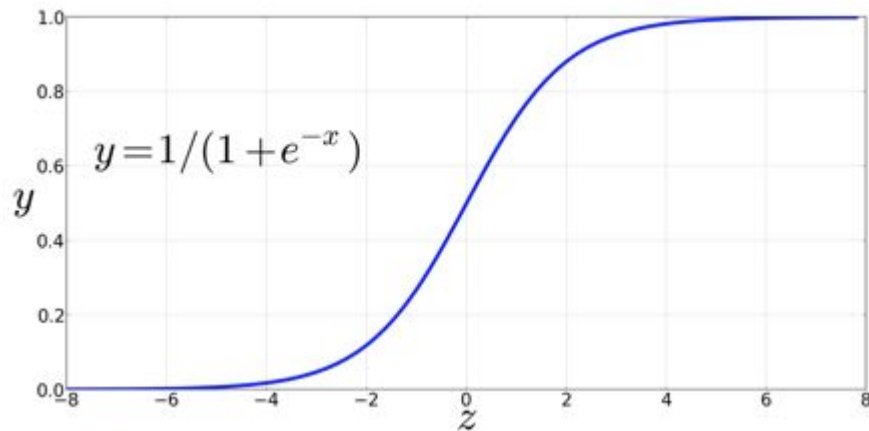
# How to compute $p(+|t,c)$ ?

- Intuition:
  - Words are likely to appear near similar words
  - Model similarity with dot-product!
- ***Similarity(t,c)***  $\propto t \cdot c$
- Problem:
  - Dot product is not a probability!
  - (Neither is cosine)

# Turning dot product into a probability

- The sigmoid lies between 0 and 1:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



## Turning dot product into a probability

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

$$\begin{aligned} P(-|t, c) &= 1 - P(+|t, c) \\ &= \frac{e^{-t \cdot c}}{1 + e^{-t \cdot c}} \end{aligned}$$

# For all the context words:

- Assume all context words in the window ( $c_{1:k}$ ) are independent

$$P(+|t, c_{1:k}) = \prod_{i=1}^{\kappa} \frac{1}{1 + e^{-t \cdot c_i}}$$

$$\log P(+|t, c_{1:k}) = \sum_{i=1}^k \log \frac{1}{1 + e^{-t \cdot c_i}}$$



# Skip-Gram Training Data

- Training sentence:

... lemon, a **tablespoon of** **apricot** **jam** **a** pinch ...

**c1**    **c2** **target** **c3**    **c4**

- Training data: input/output pairs centering on **apricot**
- Assume context words are those in +/- 2 word window

# Skip-Gram Training Data

- Training sentence:

... lemon, a **tablespoon of** **apricot jam** **a** pinch ...

**c1**    **c2** **target** **c3**    **c4**

## positive examples +

t	c
apricot	tablespoon
apricot	of
apricot	preserves
apricot	or

- For each positive example, create  $k$  negative examples.
- Using any random word that isn't  $t$

# Skip-Gram Training Data

- Training sentence:

... lemon, a **tablespoon of** **apricot jam** **a** pinch ...

**c1**    **c2** **target** **c3**    **c4**

## positive examples +

t	c
apricot	tablespoon
apricot	of
apricot	preserves
apricot	or

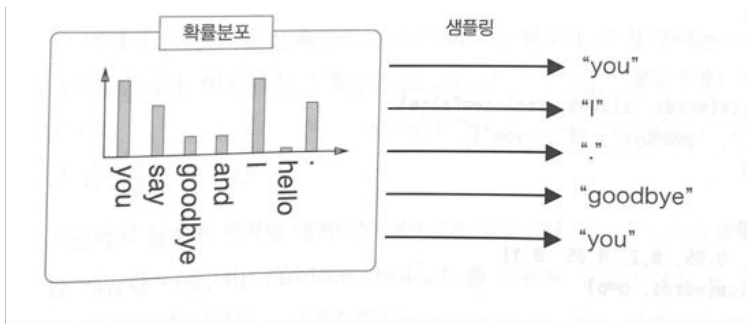
## negative examples -

k=2

t	c	t	c
apricot	aardvark	apricot	twelve
apricot	puddle	apricot	hello
apricot	where	apricot	dear
apricot	coaxial	apricot	forever

# Negative Sampling

- 단어별 출현횟수를 바탕으로 확률 분포를 구해서 그에 따라서 샘플링
- 자주 등장하는 단어가 선택될 확률이 높음



# Choosing noise words

- Pick  $w$  according to their unigram frequency  $P(w)$  or weighted unigram frequency  $p_\alpha(w)$

$$P_\alpha(w) = \frac{\text{count}(w)^\alpha}{\sum_w \text{count}(w)^\alpha}$$

$\alpha=0.75$  works well because it gives rare noise words slightly higher probability

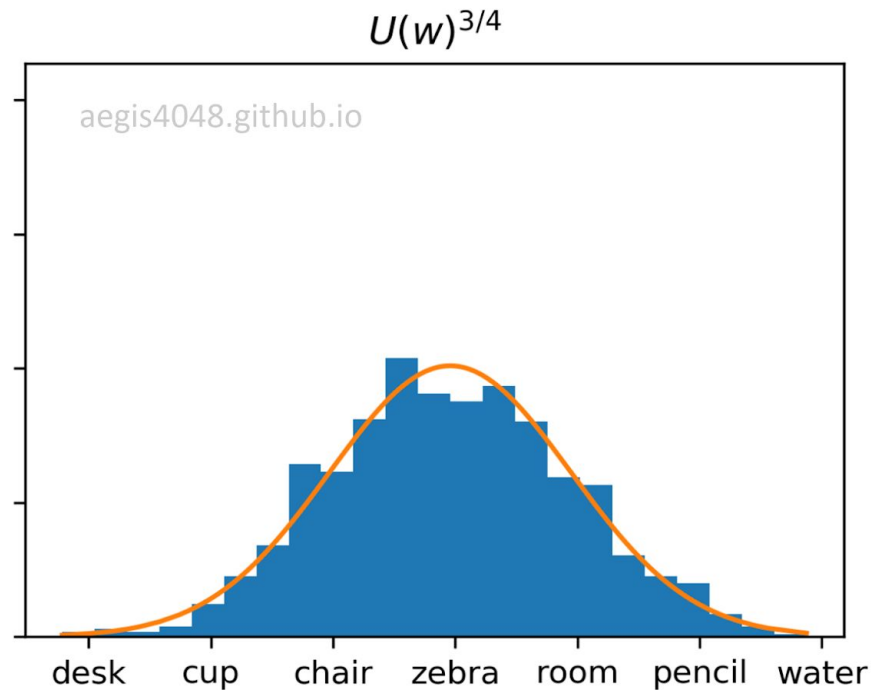
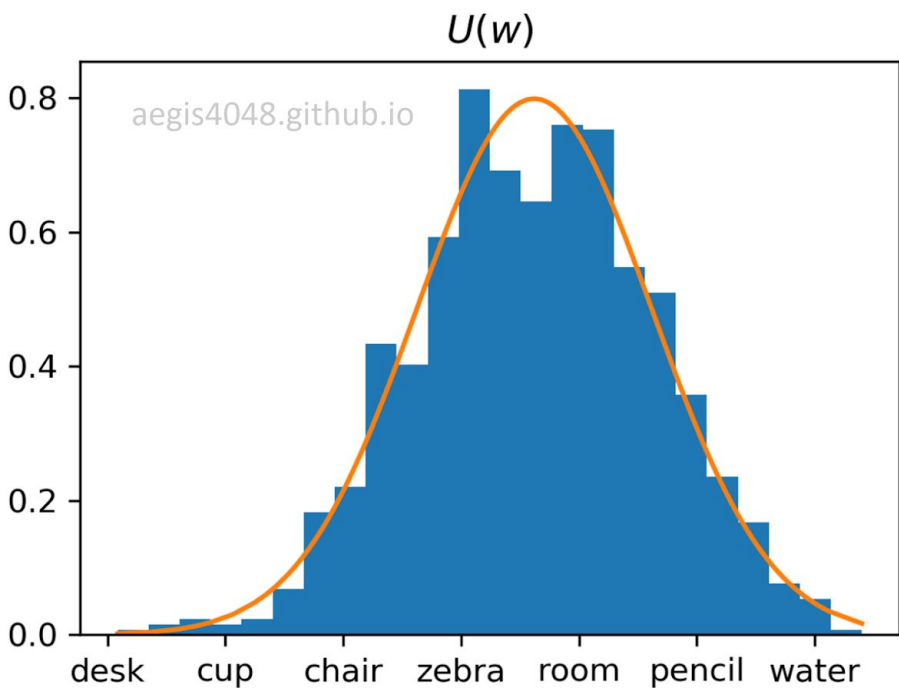
- **For rare words,  $P_\alpha(w) > P(w)$**

E.g., two events  $p(a)=.99$  and  $p(b) = .01$ :

$$P_\alpha(a) = \frac{.99^{.75}}{.99^{.75} + .01^{.75}} = .97$$

$$P_\alpha(b) = \frac{.01^{.75}}{.99^{.75} + .01^{.75}} = .03$$

# Effect of raising power of unigram distribution $U(w)$



# Setup

- Represent words as vectors of some length (e.g., 300)
- Start with  $300 * V$  random parameters
- Over the entire training set, adjust those word vectors such that
  - **Maximize** the similarity of the **target word, context word pairs**  $(t,c)$  drawn from the **positive data**
  - **Minimize** the similarity of the  $(t,c)$  pairs drawn from the **negative data**

**Maximize**

<b>positive examples +</b>	
t	c
apricot	tablespoon
apricot	of
apricot	preserves
apricot	or

**Minimize**

<b>negative examples -</b>			
t	c	t	c
apricot	aardvark	apricot	twelve
apricot	puddle	apricot	hello
apricot	where	apricot	dear
apricot	coaxial	apricot	forever

# Learning the classifier

Iterative process over the entire training set:

- Start with 0 or random weights
- Then **adjust the word weights** to
  - Make the **positive pairs more likely**
  - Make the **negative pairs less likely**



# Objective Criteria

- Maximize + label for **the pairs from the positive training data**, and the – label for the pairs sample from the negative data

$$\sum_{(t,c) \in +} \log P(+|t, c) + \sum_{(t,c) \in -} \log P(-|t, c)$$

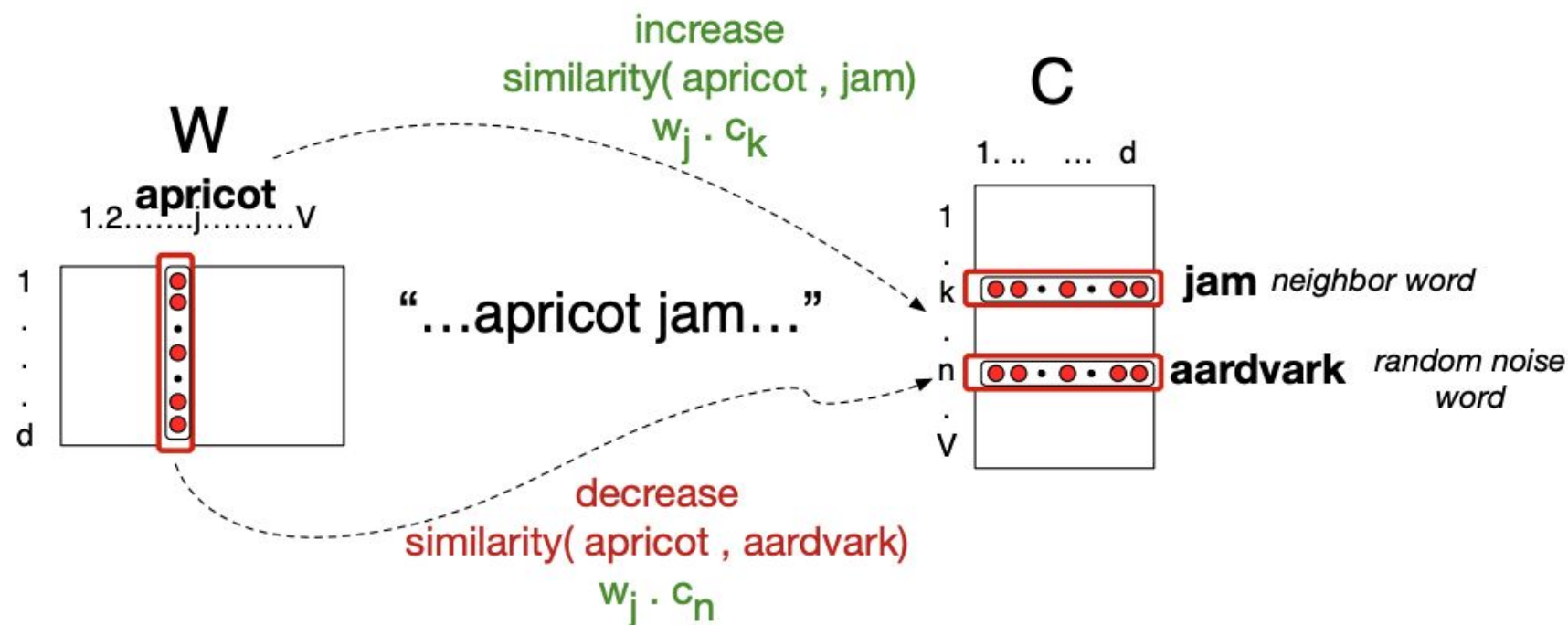
# Focusing on one target word $t$ :

- Or, focus on one word/context pair  $(t, c)$  with its  $k$  noise words  $n_1 \dots n_k$
- **maximize** the dot product of the word with the actual context words, and **minimize** the dot products of the word with the  $k$  negative sampled non-neighbor words

$$\log P(+|t, c) + \sum_{i=1} \log P(-|t, n_i)$$

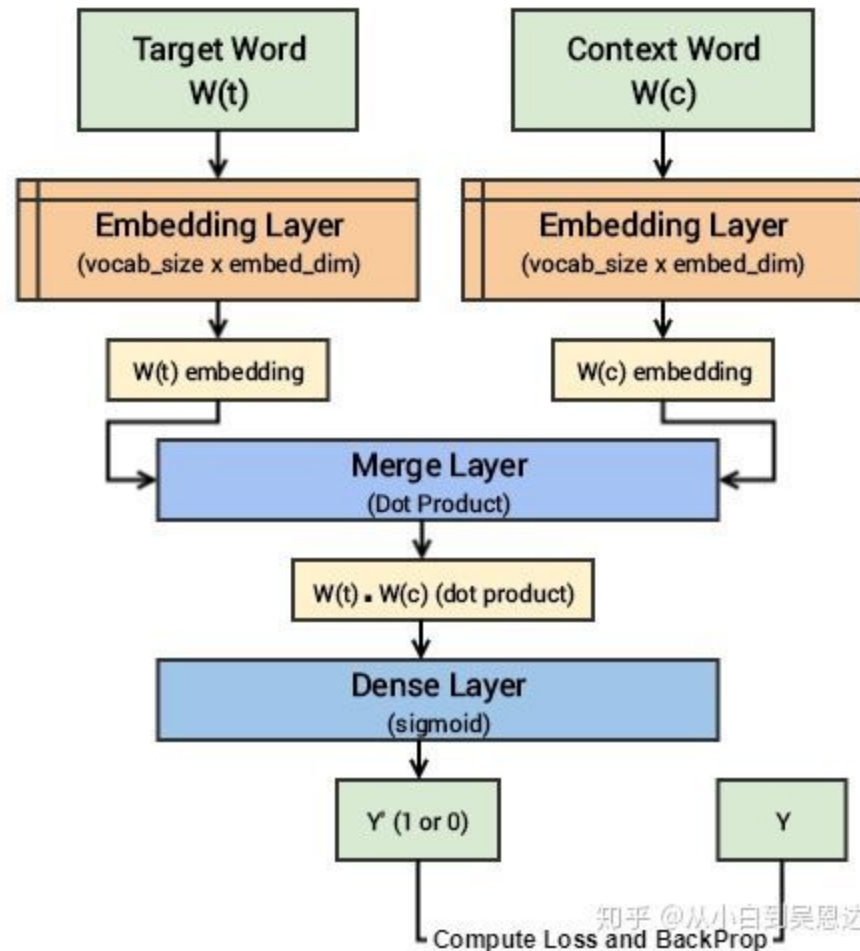
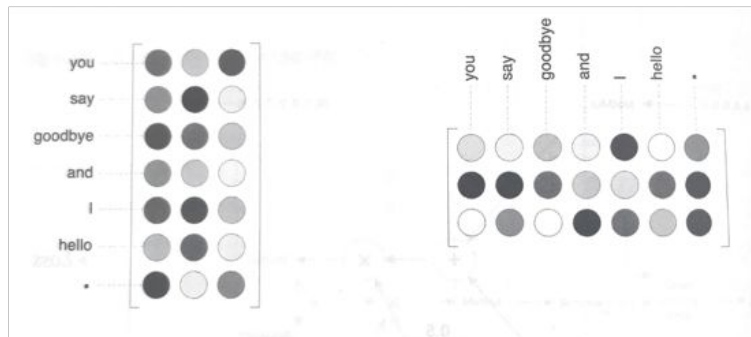
$$\log \sigma(c \cdot t) + \sum_{i=1}^k \log \sigma(\boxed{-} n_i \cdot t)$$

$$\log \frac{1}{1 + e^{-c \cdot t}} + \sum_{i=1}^k \log \frac{1}{1 + e^{n_i \cdot t}}$$



**Figure 6.13** The skip-gram model tries to shift embeddings so the target embedding (here for *apricot*) are closer to (have a higher dot product with) context embeddings for nearby words (here *jam*) and further from (have a lower dot product with) context embeddings for words that don't occur nearby (here *aardvark*).

# Overview

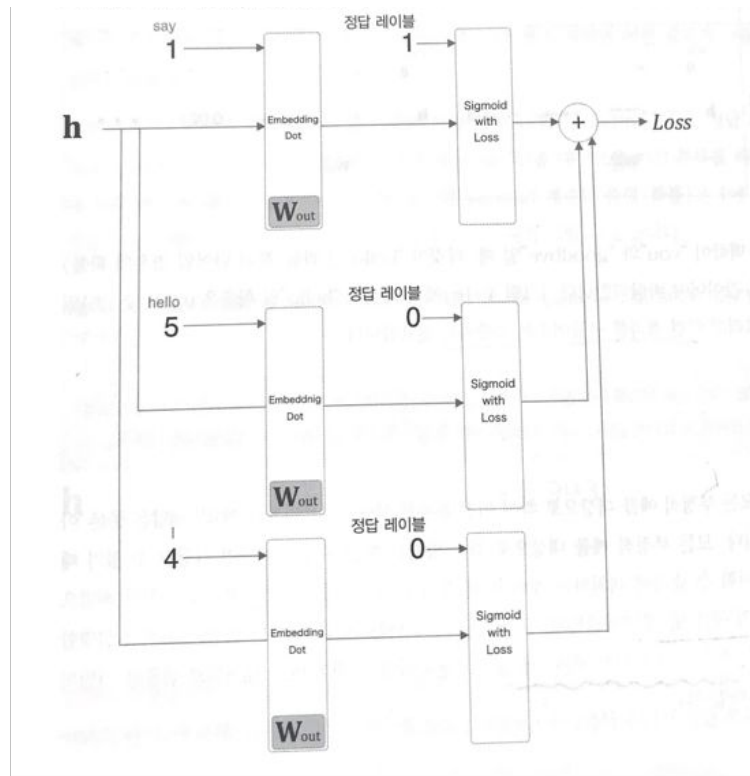


# Learn with Negative samples

$$\log P(+|t, c) + \sum_{i=1} \log P(-|t, n_i)$$

$$\log \sigma(c \cdot t) + \sum_{i=1}^k \log \sigma(-n_i \cdot t)$$

$$\log \frac{1}{1 + e^{-c \cdot t}} + \sum_{i=1}^k \log \frac{1}{1 + e^{n_i \cdot t}}$$



# Train using gradient descent

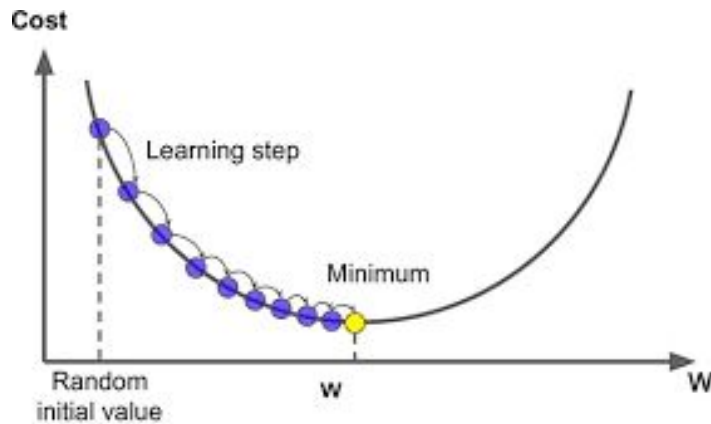
- **Learns two separate embedding matrices**  $\mathbf{W}$  and  $\mathbf{C}$  using stochastic gradient descent to train to this objective, iteratively modifying the parameters to maximize the objective
- Each **row**  $i$  of the target matrix  $T$  is the  $1 \times d$  vector embedding  $t_i$  for word  $i$  in the vocabulary  $V$ , and each column  $i$  of the context matrix  $C$  is a  $d \times 1$  vector embedding  $c_i$  for word  $i$  in  $V$
- We can choose to throw away the  $C$  matrix and **just keep  $\mathbf{W}$** , in which case each word  $i$  will be represented by the vector  $t_i$
- Or, we can **add** the two embeddings together,  $t_i + c_i$ , or we can **concatenate** them into an embedding of dimensionality  $2d$

# Gradient Descent

- 그래프에서 y축이 손실값일때, 기울기가 음수인 방향으로 가중치를 변경
- 전체 훈련셋에 대하여 계산하고 평균 손실치 구함
- 기울기가 양이면 x축으로 음의 방향, 기울기가 음이면 x축으로 양의 방향으로 가중치 갱신

$$w := w - \eta \nabla Q(w) = w - \eta \sum_{i=1}^n \nabla Q_i(w) / n,$$

step size 혹은 learning rate



# Train

[https://aegis4048.github.io/optimize\\_computational\\_efficiency\\_of\\_skip-gram\\_with\\_negative\\_sampling](https://aegis4048.github.io/optimize_computational_efficiency_of_skip-gram_with_negative_sampling)



# Summary: learn word2vec (skip-gram) embeddings

- Start with  $V$  random 300-dimensional vectors as initial embeddings
- Use logistic regression, the second most basic classifier used in machine learning after naïve bayes
  - Take a corpus and take pairs of words that co-occur as positive examples
  - Take pairs of words that don't co-occur as negative examples
  - Train the classifier to distinguish these using stochastic gradient descent, iteratively modifying the parameters to maximize the objective
  - Throw away the classifier code and keep the embeddings

# Evaluating embeddings

Compare to human scores on **word similarity-type tasks**:

- WordSim-353 (Finkelstein et al., 2002)
- SimLex-999 (Hill et al., 2015)
- Stanford Contextual Word Similarity (SCWS) dataset (Huang et al., 2012)
- TOEFL dataset: *Levied is closest in meaning to: imposed, believed, requested, correlated*

# Properties of embeddings

Similarity depends on window size  $C$

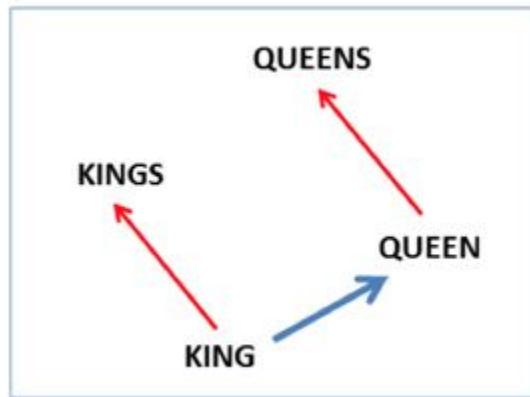
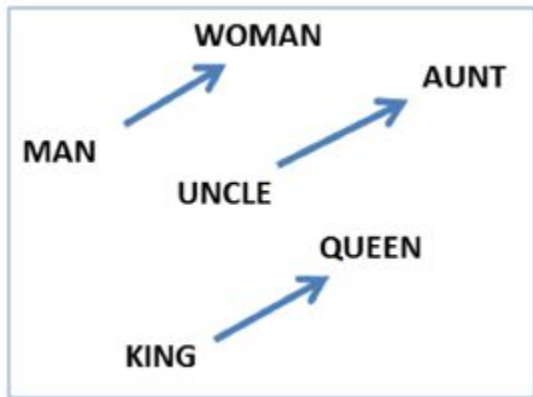
- Dataset: Harry Porters
- $C = \pm 2$  The nearest words to Hogwarts:
  - Sunnydale
  - Evernight
- $C = \pm 5$  The nearest words to Hogwarts:
  - Dumbledore
  - Malfoy
  - halfblood



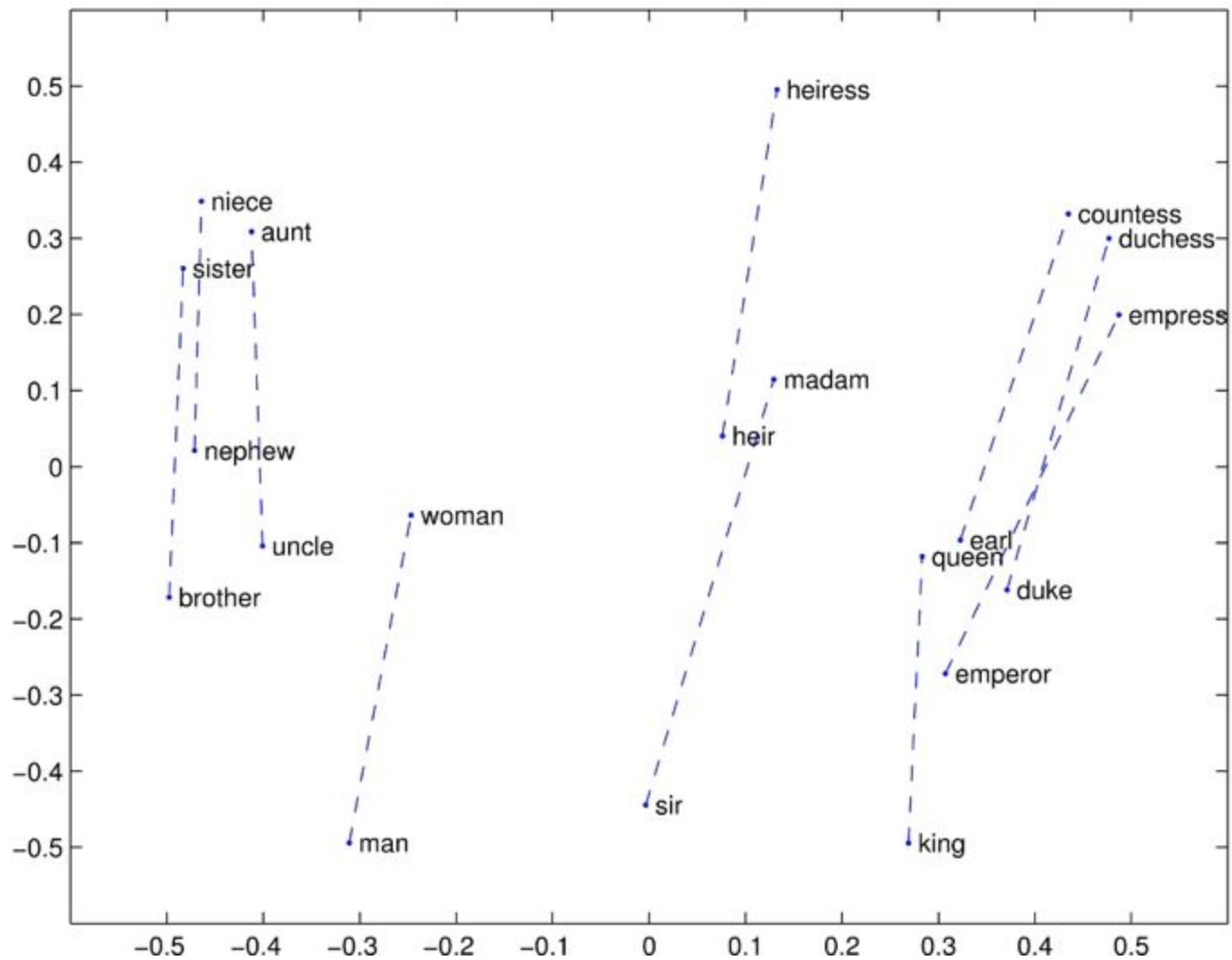
# Analogy: Embeddings capture relational meaning!

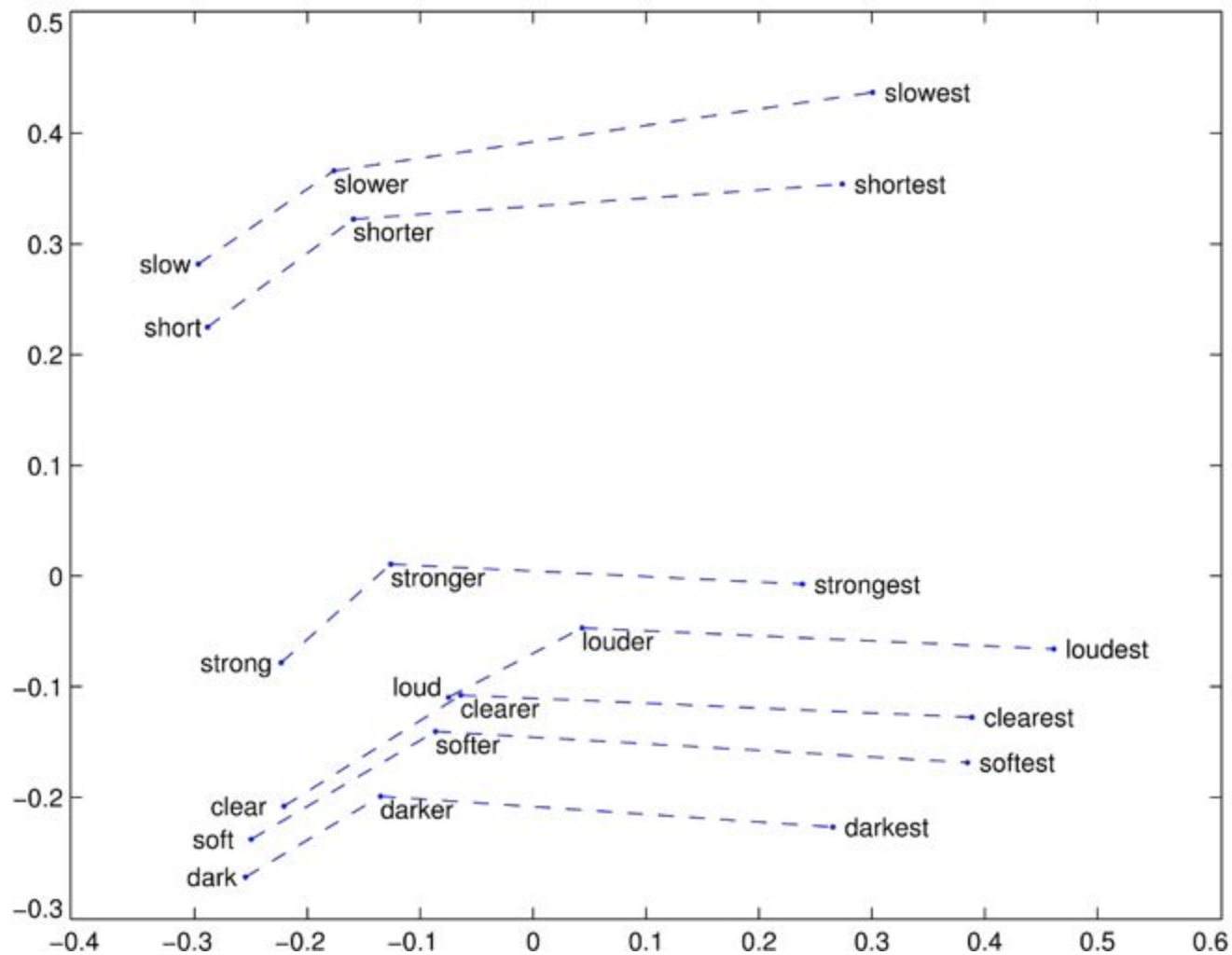
$\text{vector}(\text{'king'}) - \text{vector}(\text{'man'}) + \text{vector}(\text{'woman'}) \approx \text{vector}(\text{'queen'})$

$\text{vector}(\text{'Paris'}) - \text{vector}(\text{'France'}) + \text{vector}(\text{'Italy'}) \approx \text{vector}(\text{'Rome'})$



- Several factors influence the quality of the word vectors: \* amount and quality of the training data \* size of the vectors \* training algorithm
- The quality of the vectors is crucial for any application
- 한국어 : <http://w.elnn.kr/search/>





	Most Similar Words for	
	Static Channel	Non-static Channel
<b><i>bad</i></b>	<i>good</i> <i>terrible</i> <i>horrible</i> <i>lousy</i>	<i>terrible</i> <i>horrible</i> <i>lousy</i> <i>stupid</i>
<b><i>good</i></b>	<i>great</i> <i>bad</i> <i>terrific</i> <i>decent</i>	<i>nice</i> <i>decent</i> <i>solid</i> <i>terrific</i>
<b><i>n't</i></b>	<i>os</i> <i>ca</i> <i>ireland</i> <i>wo</i>	<i>not</i> <i>never</i> <i>nothing</i> <i>neither</i>
<b><i>!</i></b>	<i>2,500</i> <i>entire</i> <i>jez</i> <i>changer</i>	<i>2,500</i> <i>lush</i> <i>beautiful</i> <i>terrific</i>
<b><i>,</i></b>	<i>decasia</i> <i>abysmally</i> <i>demise</i> <i>valiant</i>	<i>but</i> <i>dragon</i> <i>a</i> <i>and</i>

*Words that appear in the same context share semantic meaning*

Convolutional Neural Networks for Sentence Classification, Yoon Kim, 2014 EMNLP



# Use Pre-trained Google Word2Vec model

- Load the word2vec model from Google
- Download the google vector file at  
<https://drive.google.com/file/d/0B7XkCwpl5KDYNINUTTlSS21pQmM/edit>

```
from gensim.models import KeyedVectors

filename = 'GoogleNews-vectors-negative300.bin'
# load the google word2vec model
model = KeyedVectors.load_word2vec_format(filename, binary=True)
# calculate: (king - man) + woman = ?
result = model.most_similar(positive=['woman', 'king'], negative=['man'], topn=1)
print(result)

[Run]

[ ('queen', 0.7118192315101624) ]
```

<https://machinelearningmastery.com/develop-word-embeddings-python-gensim/>

# word2vec with Korean

```
>>> from gensim.models import word2vec
>>> model = word2vec.Word2Vec.load("toji.model")
>>> model.most_similar(positive=["땅"])
>>> model["땅"]
```



```
import codecs
from bs4 import BeautifulSoup
from konlpy.tag import Okt
from gensim.models import word2vec
fp = codecs.open("BEXX0003.txt", "r", encoding="utf-16")
soup = BeautifulSoup(fp, "html.parser")
body = soup.select_one("body > text")
text = body.getText()
twitter = Okt()
results = []
lines = text.split("\n\n")
for line in lines: # 텍스트를 한 줄씩 처리
    malist = twitter.pos(line, norm=True, stem=True)
    r = []
    for word in malist: # 어미/조사/구두점 등은 대상에서 제외
        if not word[1] in ["Josa", "Eomi", "Punctuation"]:
            r.append(word[0])
    results.append((" ".join(r)).strip())
with open('toji.wakati', 'w', encoding='utf-8') as fp:
    fp.write("\n".join(results))
# Word2Vec 모델 만들기 --- (※5)
data = word2vec.LineSentence('toji.wakati')
model = word2vec.Word2Vec(data, size=200, window=10, hs=1,
min_count=2, sg=1)
```

# Doc2Vec Train using Gensim

Reimplements word2vec, starting with the hierarchical softmax skip-gram model, which reports the best accuracy

## Result

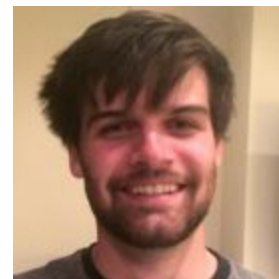
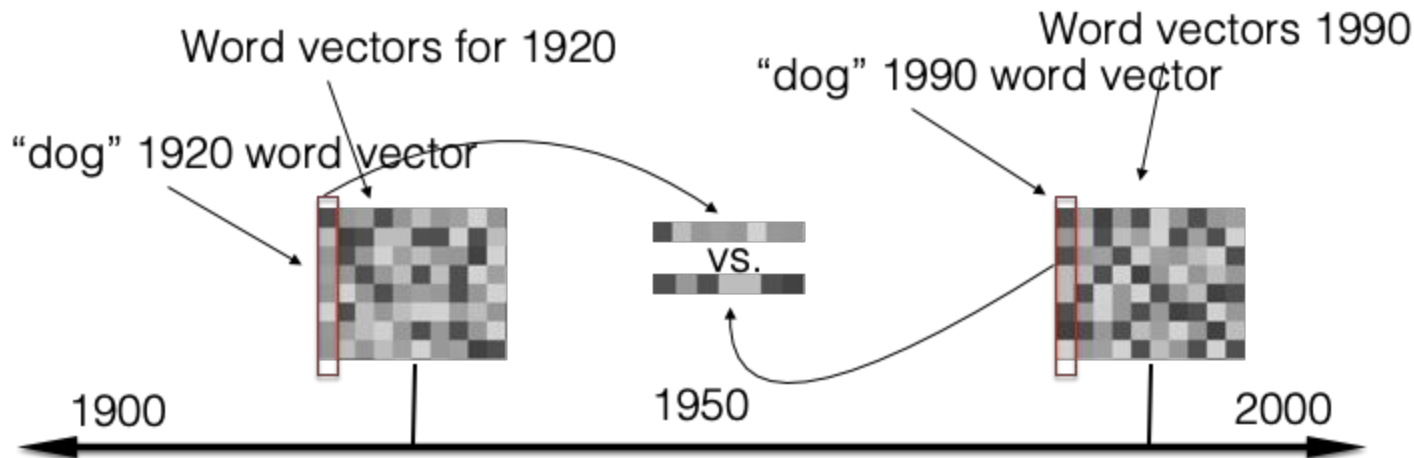
```
[TaggedDocument(words=['This', 'is', 'the', 'first',  
'document', '.'], tags=['d0']),  
TaggedDocument(words=['This', 'document', 'is', 'the',  
'second', 'document', '.'], tags=['d1']),  
TaggedDocument(words=['And', 'this', 'is', 'the', 'third',  
'one', '.'], tags=['d2']), TaggedDocument(words=['Is', 'this',  
'the', 'first', 'document', '?'], tags=['d3'])]
```

```
[-0.08623783 0.01574823 -0.07002052  
-0.03521339 0.03276849]
```

```
from gensim.models.doc2vec import TaggedDocument, Doc2Vec  
from nltk.tokenize import word_tokenize  
corpus = [ 'This is the first document.',  
           'This document is the second document.',  
           'And this is the third one.',  
           'Is this the first document?' ]  
  
corpus = [list(word_tokenize(doc)) for doc in corpus]  
corpus = [ TaggedDocument(words, ['d{}'.format(idx)])  
           for idx, words in enumerate(corpus) ]  
  
print(corpus)  
model = Doc2Vec(corpus, vector_size=5, min_count=0)  
print(model.docvecs[0])
```

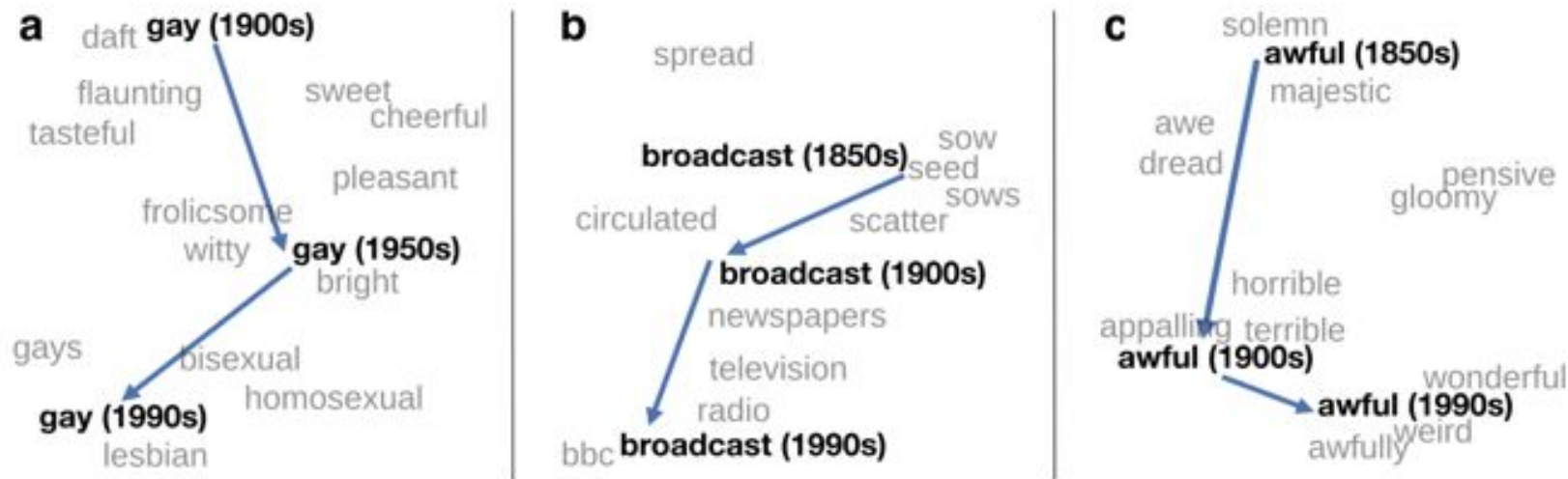
# Embeddings can help study word history!

Train embeddings on old books to study changes in word meaning!!



Will Hamilton

# Project 300 dimensions down into 2



~30 million books, 1850-1990, Google Books data

# Embeddings reflect cultural bias

Bolukbasi, Tolga, Kai-Wei Chang, James Y. Zou, Venkatesh Saligrama, and Adam T. Kalai. "Man is to computer programmer as woman is to homemaker? debiasing word embeddings." In Advances in Neural Information Processing Systems, pp. 4349-4357. 2016.

- Ask “Paris : France :: Tokyo : x”
  - x = Japan
- Ask “father : doctor :: mother : x”
  - x = nurse
- Ask “man : computer programmer :: woman : x”
  - x = homemaker

# Embeddings reflect cultural bias (Caliskan et al., 2017)

- Psychological findings on US participants:
  - African-American names are associated with unpleasant words (more than European-American names)
  - Male names associated more with math, female names with arts
  - Old people's names with unpleasant words, young people with pleasant words.
- Caliskan et al. replication with embeddings:
  - African-American names (Leroy, Shaniqua) had a higher GloVe cosine with unpleasant words (abuse, stink, ugly)
  - European American names (Brad, Greg, Courtney) had a higher cosine with pleasant words (love, peace, miracle)
- Embeddings reflect and replicate all sorts of pernicious biases

# Distributed representations (Word embeddings)

- Based on the distributional hypothesis: words with similar meanings tend to occur in similar context
- Words from the vocabulary are mapped to vectors of real numbers
- Embedding from a space with one dimension per word to a continuous vector space with a much lower dimension
- Used as the underlying input representation
- Word vectors capture the characteristics of the neighbors of a word
- Typically pre-trained
- Software for training and using word embeddings: Word2vec, GloVe, Elmo, fastText, Gensim
- Advantage of distributional vectors
  - Capture similarity between words
  - Fast and efficient in computing core NLP tasks



# Summary

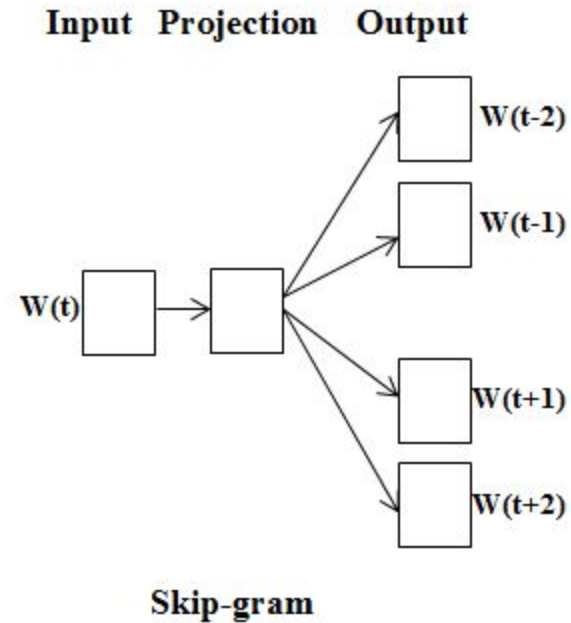
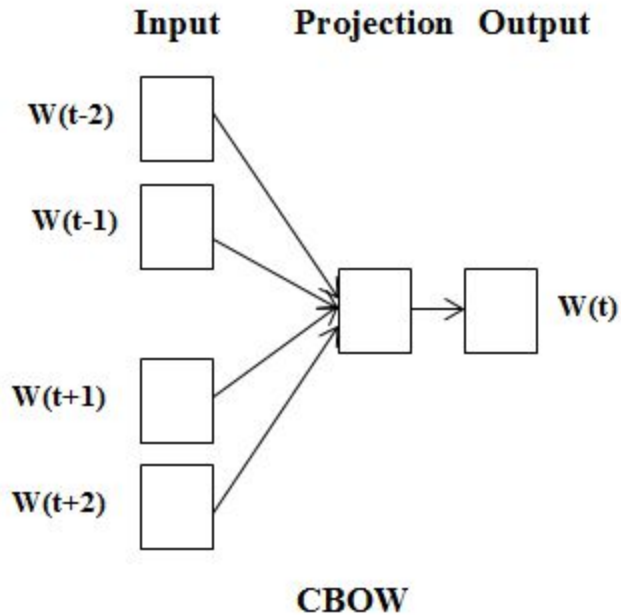
## **Concepts** or word senses

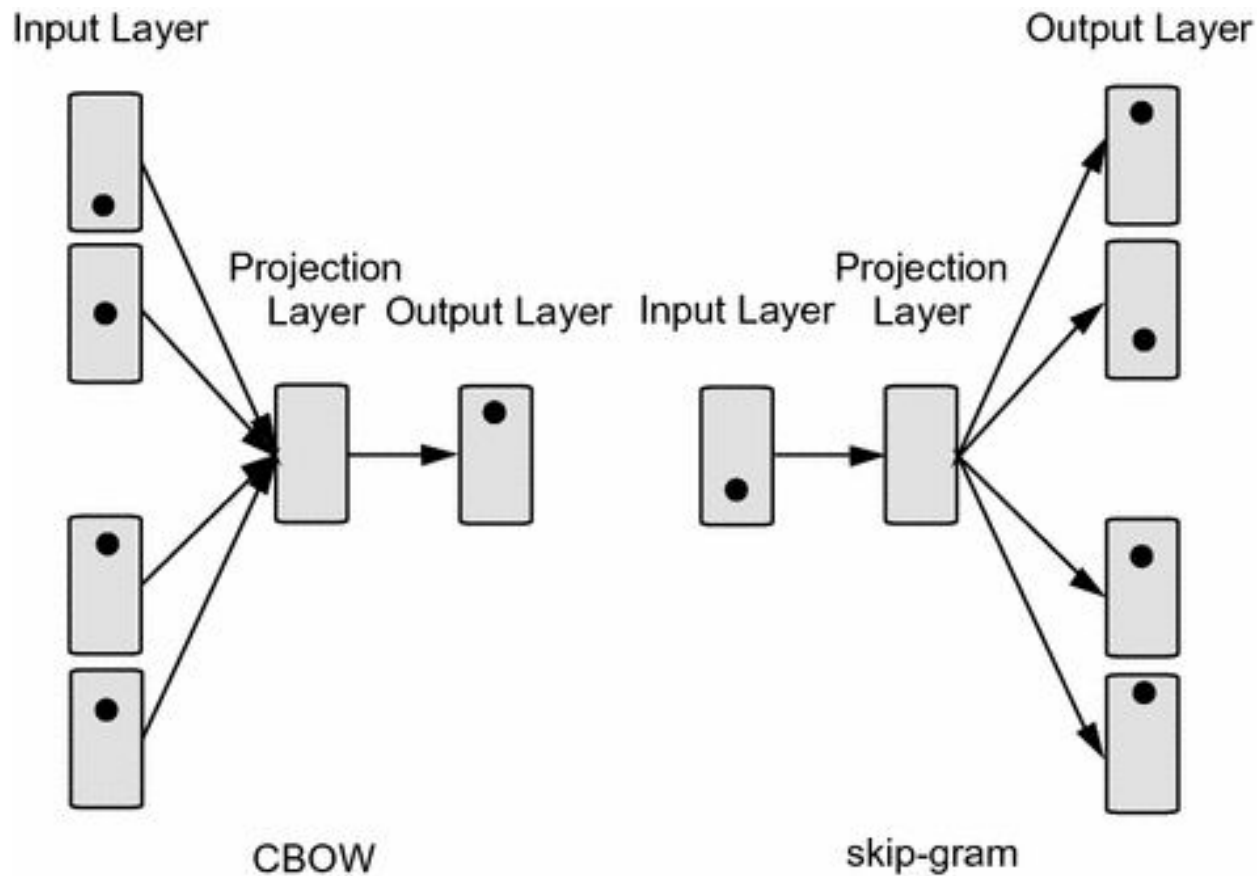
- Have a complex many-to-many association with words (multiple senses)
- Have relations with each other: Synonymy, Antonymy, Superordinate
- But are hard to define formally

## **Embeddings** = vector models of meaning

- More fine-grained than just a string or index
- Good at modeling similarity/analogy: Just download them and use cosines!!
- Can use sparse models (tf-idf) or dense models (word2vec, GLoVE)
- Useful in practice but know they encode cultural stereotypes

# Word2vec algorithms





# Skip-gram vs. CBOW

- Skip-gram: works well with small amount of the training data, represents well even rare words or phrases
- CBOW: several times faster to train than the skip-gram, slightly better accuracy for the frequent words

# Word2vec skip-gram using Pytorch without negative sampling

출처:

[https://aegis4048.github.io/demystifying\\_neural\\_network\\_in\\_skip\\_gram\\_language\\_modeling#weight\\_matrix](https://aegis4048.github.io/demystifying_neural_network_in_skip_gram_language_modeling#weight_matrix)  
x  
<https://towardsdatascience.com/implementing-word2vec-in-pytorch-skip-gram-model-e6bae040d2fb>

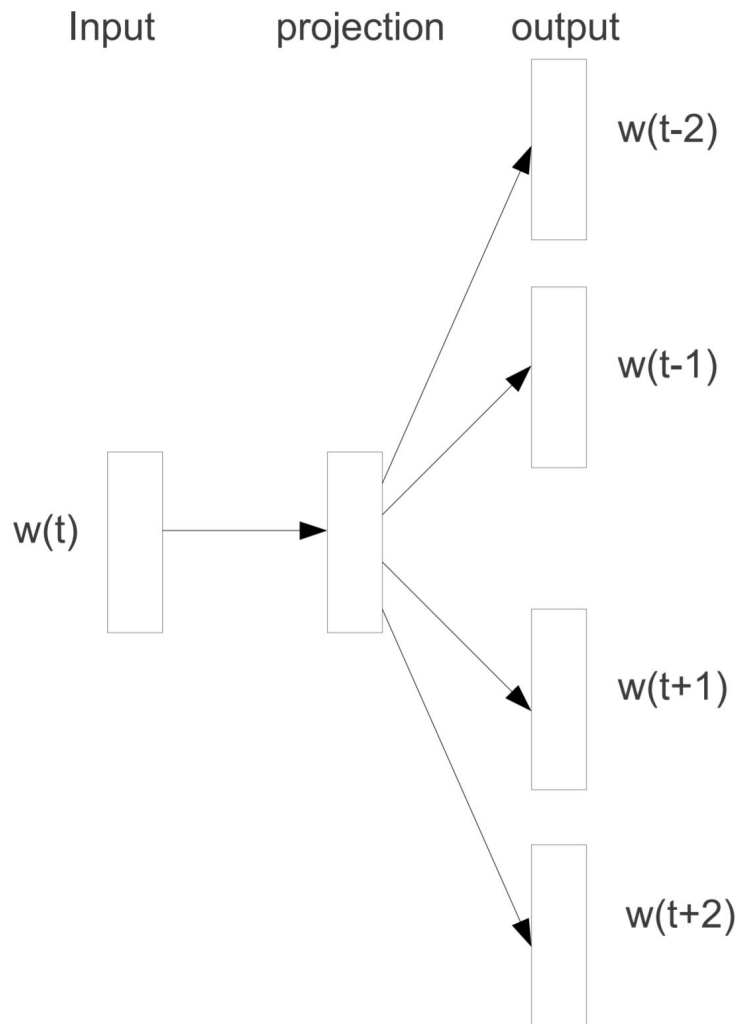
# Skip-gram

- Attempts to maximize the probability of observing all four context words together, given a center word
- Training objective is **to learn word vector representations** that are good at predicting the nearby words

$$\operatorname{argmax}_{\theta} p(w_1, w_2, \dots, w_C | w_{center}; \theta)$$



$$\operatorname{argmax}_{\theta} \log p(w_1, w_2, \dots, w_C | w_{center}; \theta)$$



## Source Text

## Training Samples

The quick brown fox jumps over the lazy dog. ➡

(the, quick)  
(the, brown)

The quick brown fox jumps over the lazy dog. ➡

(quick, the)  
(quick, brown)  
(quick, fox)

The quick brown fox jumps over the lazy dog. ➡

(brown, the)  
(brown, quick)  
(brown, fox)  
(brown, jumps)

The quick brown fox jumps over the lazy dog. ➡

(fox, quick)  
(fox, brown)  
(fox, jumps)  
(fox, over)

center word context words

I like playing football with my friends →

Center Word

[1, 0, 0, 0, 0, 0, 0]

Context Words

[0, 1, 0, 0, 0, 0, 0]  
[0, 0, 1, 0, 0, 0, 0]

I like playing football with my friends →

[0, 1, 0, 0, 0, 0, 0]

[1, 0, 0, 0, 0, 0, 0]  
[0, 0, 1, 0, 0, 0, 0]  
[0, 0, 0, 1, 0, 0, 0]

I like playing football with my friends →

[0, 0, 1, 0, 0, 0, 0]

[1, 0, 0, 0, 0, 0, 0]  
[0, 1, 0, 0, 0, 0, 0]  
[0, 0, 0, 1, 0, 0, 0]  
[0, 0, 0, 0, 1, 0, 0]

I like playing football with my friends →

[0, 0, 0, 1, 0, 0, 0]

[0, 1, 0, 0, 0, 0, 0]  
[0, 0, 1, 0, 0, 0, 0]  
[0, 0, 0, 0, 1, 0, 0]  
[0, 0, 0, 0, 0, 1, 0]

I like playing football with my friends →

[0, 0, 0, 0, 1, 0, 0]

[0, 0, 1, 0, 0, 0, 0]  
[0, 0, 0, 1, 0, 0, 0]  
[0, 0, 0, 0, 0, 1, 0]  
[0, 0, 0, 0, 0, 0, 1]

I like playing football with my friends →

[0, 0, 0, 0, 0, 1, 0]

[0, 0, 0, 1, 0, 0, 0]  
[0, 0, 0, 0, 1, 0, 0]  
[0, 0, 0, 0, 0, 0, 1]

I like playing football with my friends →

[0, 0, 0, 0, 0, 0, 1]

[0, 0, 0, 0, 1, 0, 0]  
[0, 0, 0, 0, 0, 1, 0]



# 전처리-word tokenization

```
corpus = [  
    'he is a king',  
    'she is a queen',  
    'he is a man',  
    'she is a woman',  
    'warsaw is poland capital',  
    'berlin is germany capital',  
    'paris is france capital',  
]
```

```
def tokenize_corpus(corpus):  
    tokens = [x.split() for x in corpus]  
    return tokens
```

```
tokenized_corpus = tokenize_corpus(corpus)
```

Word tokenization



```
[[ 'he', 'is', 'a', 'king'],  
 [ 'she', 'is', 'a', 'queen'],  
 [ 'he', 'is', 'a', 'man'],  
 [ 'she', 'is', 'a', 'woman'],  
 [ 'warsaw', 'is', 'poland', 'capital'],  
 [ 'berlin', 'is', 'germany', 'capital'],  
 [ 'paris', 'is', 'france', 'capital']]
```

# 전처리 - Create Vocabulary

```
vocabulary = []  
for sentence in tokenized_corpus:  
    for token in sentence:  
        if token not in vocabulary:  
            vocabulary.append(token)  
  
word2idx = {w: idx for (idx, w) in enumerate(vocabulary)}  
idx2word = {idx: w for (idx, w) in enumerate(vocabulary)}  
  
vocabulary_size = len(vocabulary)
```



사전

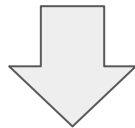
0: 'he',  
1: 'is',  
2: 'a',  
3: 'king',  
4: 'she',  
5: 'queen',  
6: 'man',  
7: 'woman',  
8: 'warsaw',  
9: 'poland',  
10: 'capital',  
11: 'berlin',  
12: 'germany',  
13: 'paris',  
14: 'france'

# 전처리 - Generate pairs center, context

```
window_size = 2
idx_pairs = []

for sentence in tokenized_corpus: # for each sentence
    indices = [word2idx[word] for word in sentence]
    # for each word as center word
    for center_word_pos in range(len(indices)):
        # for each window position
        for w in range(-window_size, window_size + 1):
            context_word_pos = center_word_pos + w
            # make sure index is not out of sentence
            if context_word_pos < 0 or context_word_pos >= len(indices) or
center_word_pos == context_word_pos:
                continue
            context_word_idx = indices[context_word_pos]
            idx_pairs.append((indices[center_word_pos], context_word_idx))
```

array([[ 0, 1],  
 [ 0, 2],  
 ...



he is  
he a  
is he  
is a  
is king  
a he  
a is  
a king

# Training data

## Source Text

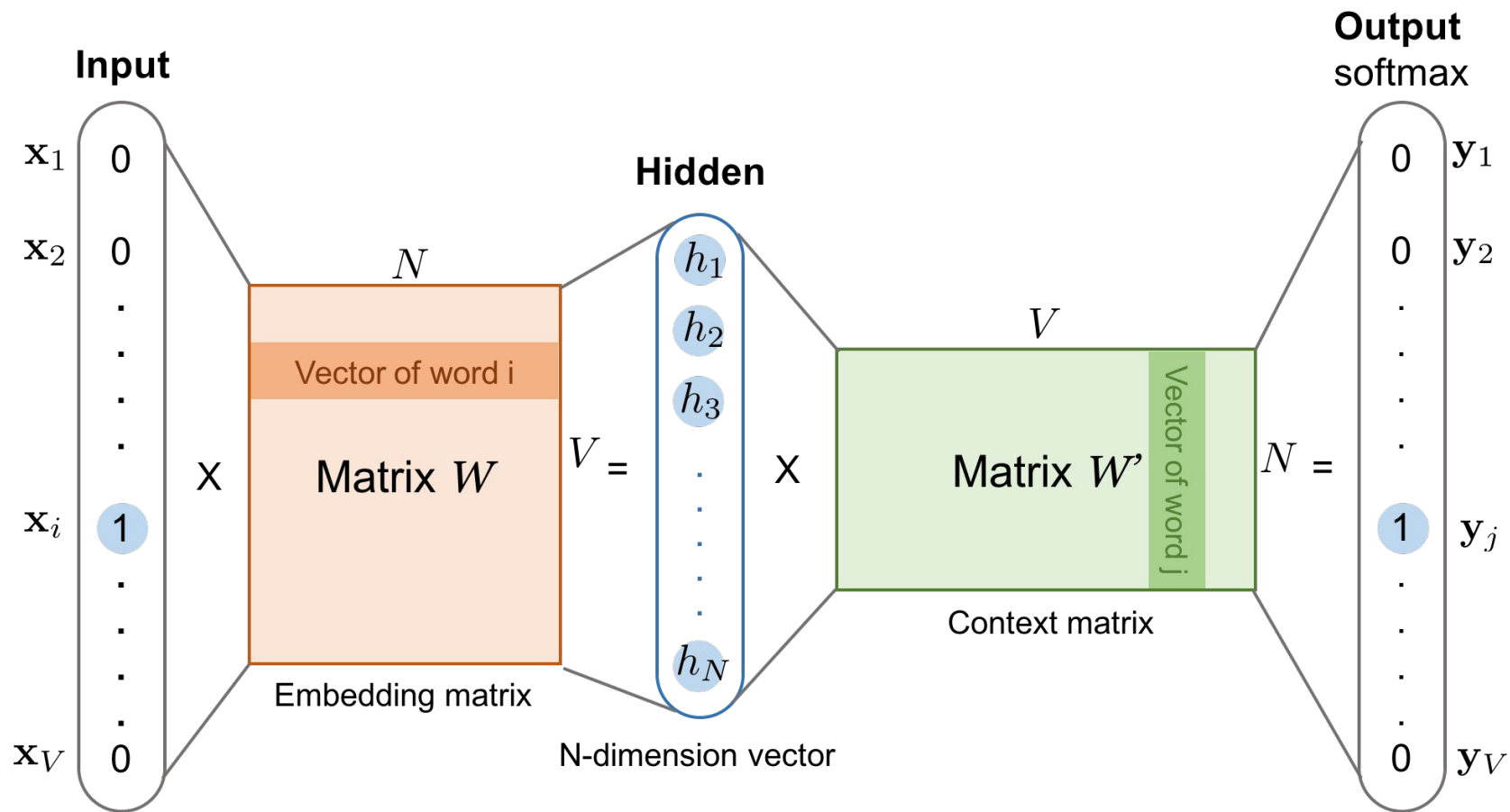
## Training Samples

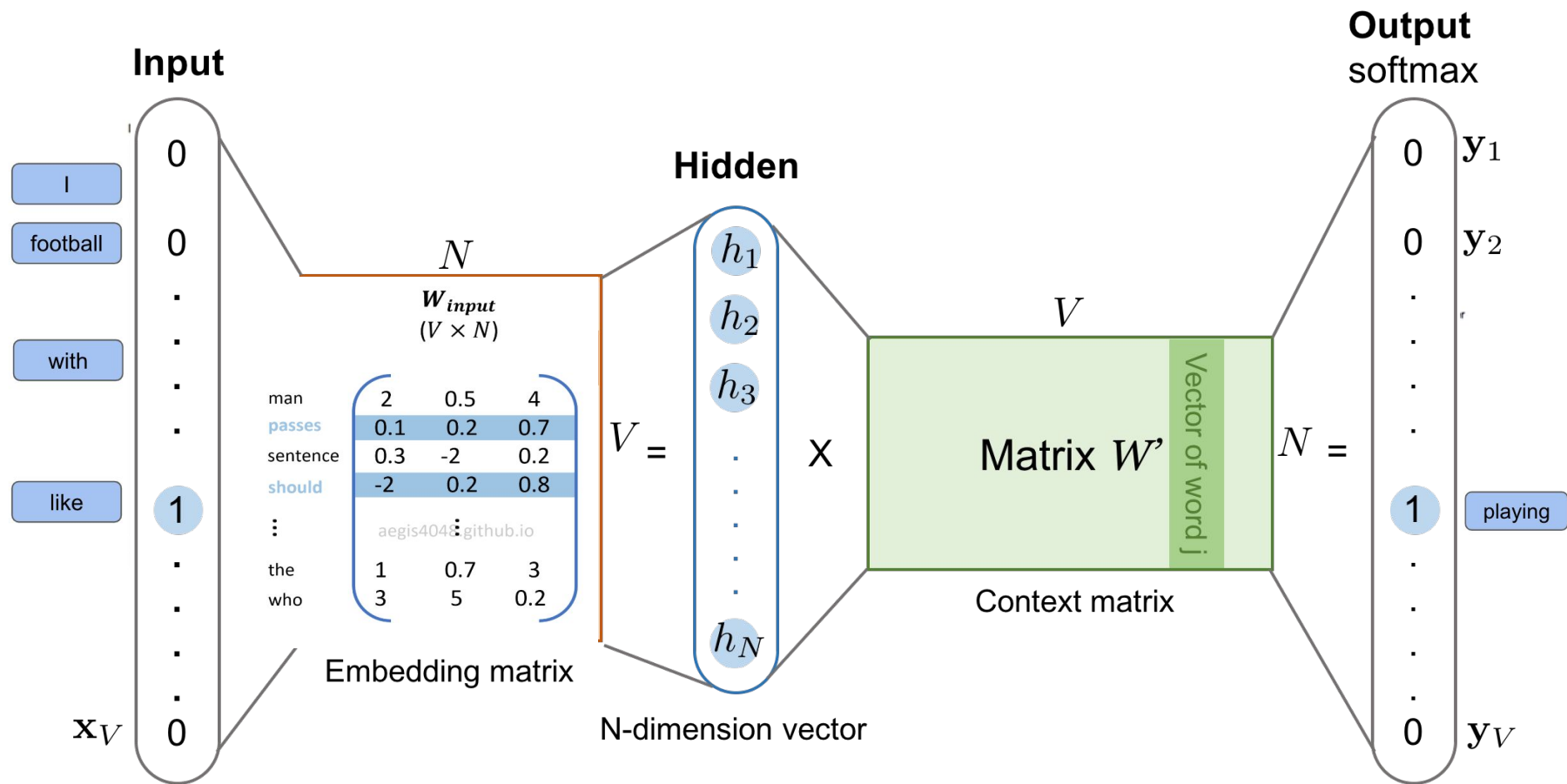
The man who passes the sentence should swing the sword. →

(passes, who)  
(passes, the)

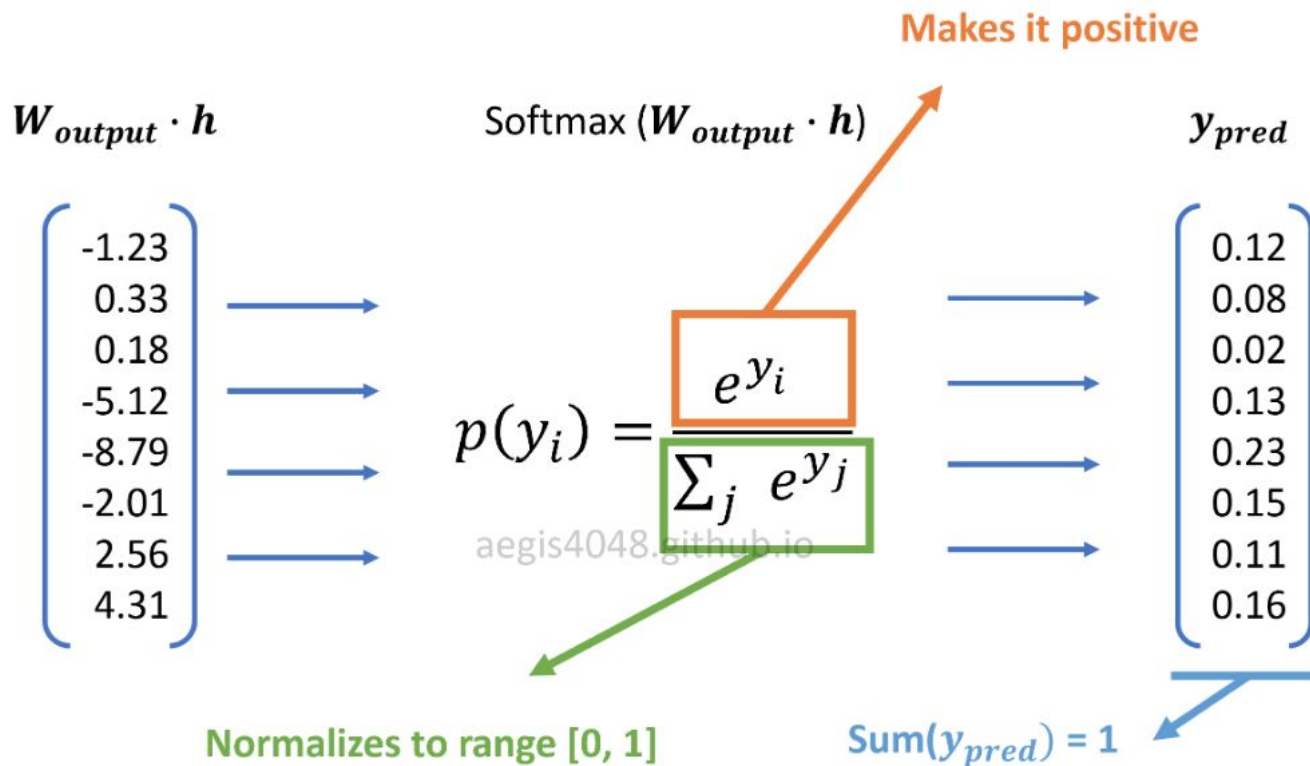
```
from gensim.models import Word2Vec  
  
model = Word2Vec(corpus, size=3, window=1)
```

- input weight matrix ( $W_{\text{input}}$ ) has a size of  $8 \times 3$
- output weight matrix ( $W_{\text{output}}^T$ ) will have a size of  $3 \times 8$
- The corpus, "The man who passes the sentence should swing the sword", has 8 unique vocabularies ( $V=8$ )



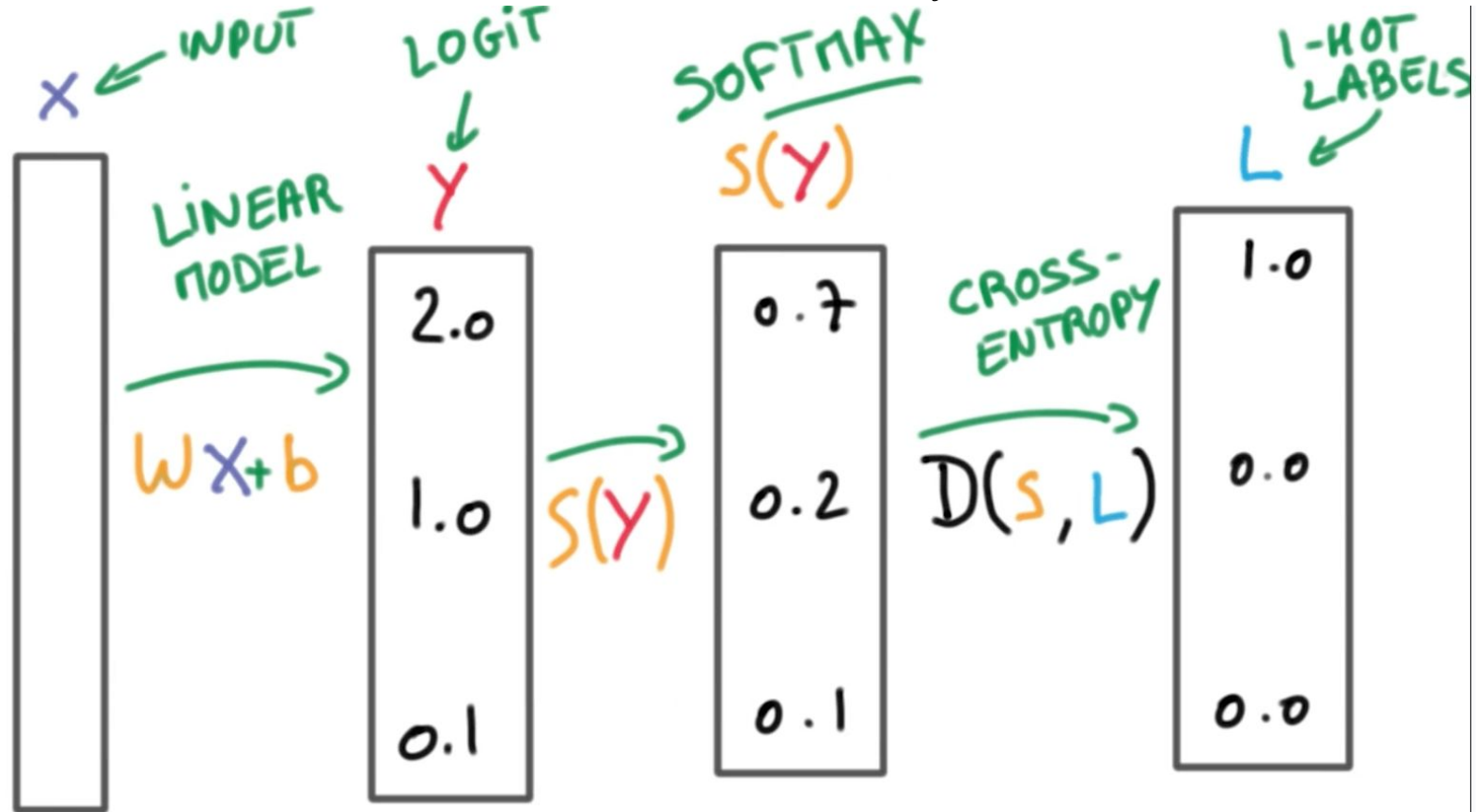


# Predict context word

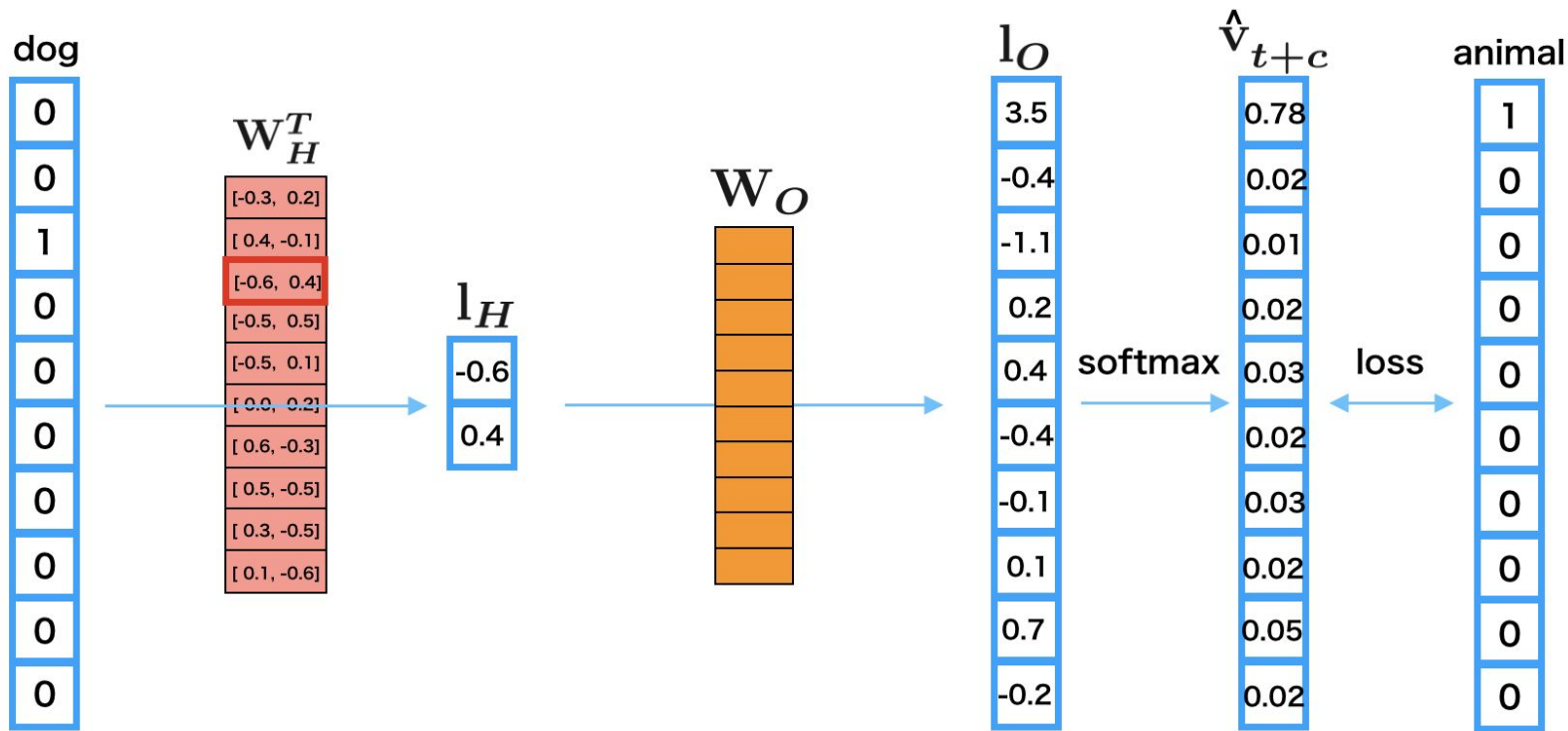


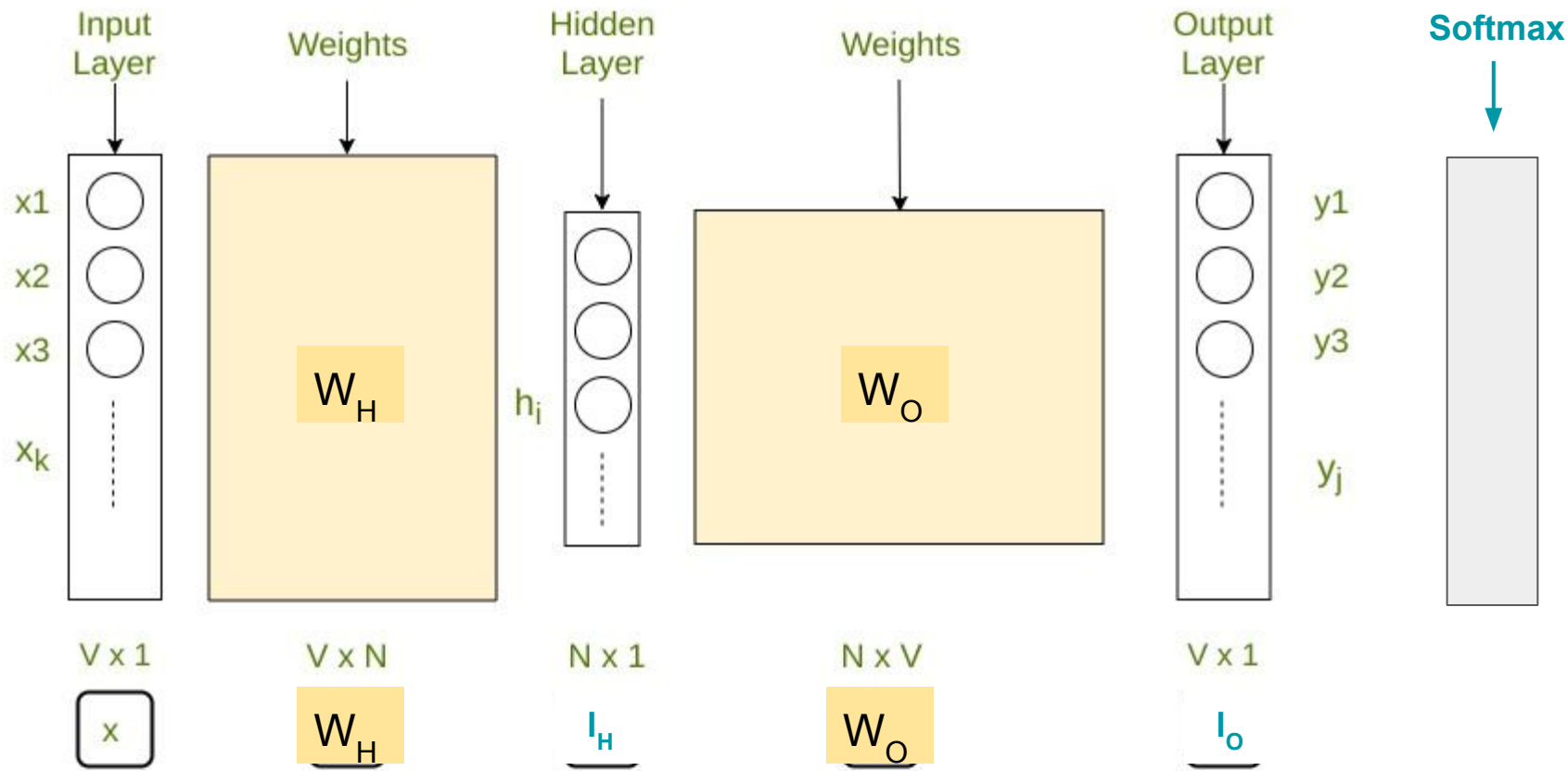
$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

$$L = - \sum_k t_k \log y_k$$









# Objective

Probability distribution for single pair

$$P(\text{context}|\text{center}; \theta)$$

E.g.:  $P(\text{king} | \text{is})$

Maximize it through all word/context pairs.

$$\max \prod_{\text{center}} \prod_{\text{context}} P(\text{context}|\text{center}; \theta)$$

# Transform for computation

- Minimize


$$\min_{\theta} - \log \prod_{\text{center}} \prod_{\text{context}} P(\text{context}|\text{center}, \theta)$$

- Product to summation

$$-\frac{1}{T} \sum_{\text{center}} \sum_{\text{context}} \log P(\text{context}|\text{center}, \theta)$$

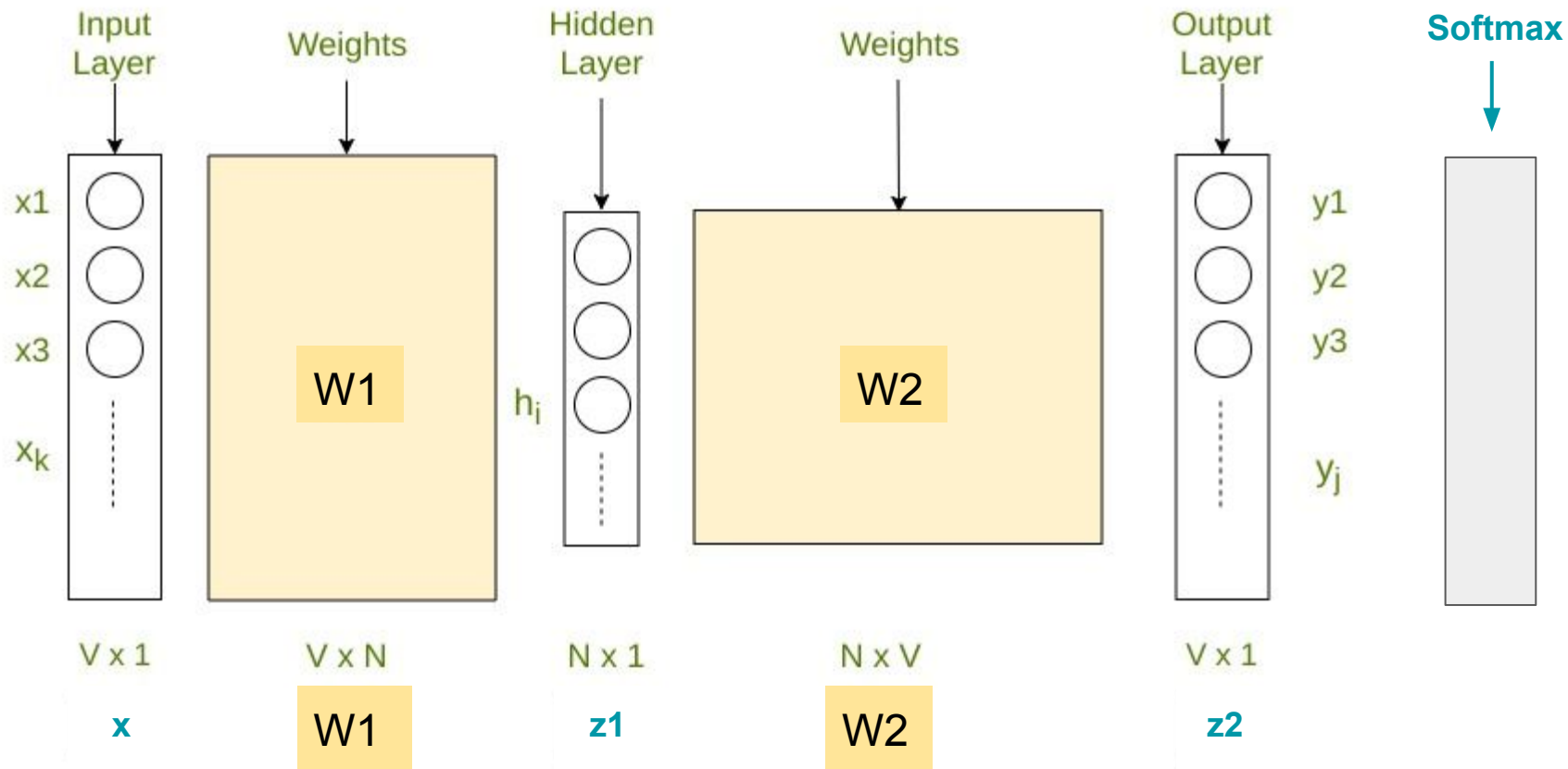
Define  $P(\text{context}|\text{center})$

softmax  $\frac{\exp(\cdot)}{\sum \exp(\cdot)}$



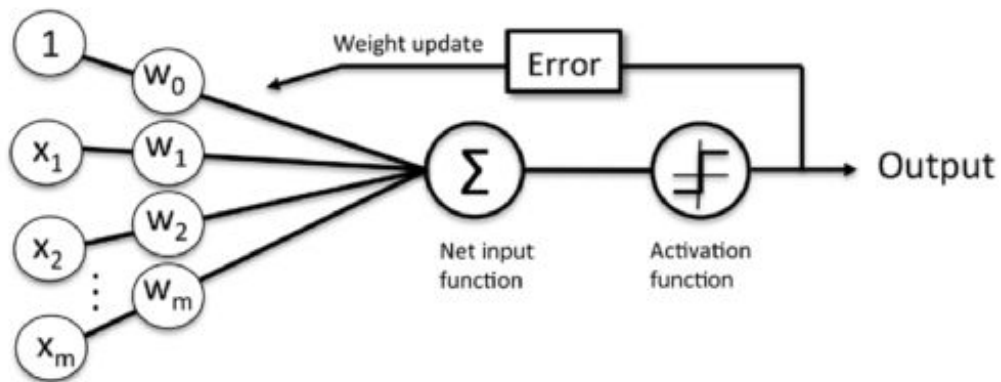
$$P(\text{context}|\text{center}) = \frac{\exp(u_{\text{context}}^T v_{\text{center}})}{\sum_{w \in \text{vocab}} \exp(u_w^T v_{\text{center}})}$$

Bigger as they are  
more similar to  
each other



# Backpropagation

- 다층 신경망에서 출력에서 계산한 오차를 이전 단으로 전달
- 계산그래프에서 계산을 왼쪽에서 오른쪽으로 진행하는 단계를 순전파(forward propagation), 오른쪽에서 왼쪽으로 진행하는 단계를 역전파(backward propagation)



# 학습 - Forward Process

## Input Layer

```
#dimensions are [1, vocabulary_size]
def get_input_layer(word_idx):
    x = torch.zeros(vocabulary_size).float()
    x[word_idx] = 1.0
    return x
```

## Hidden layer

```
#W1 weight matrix: [embedding_dims, vocabulary_size]
#each column of W1 stores v vector for single word
embedding_dims = 5
W1 = Variable(torch.randn(embedding_dims, vocabulary_size).float(), requires_grad=True)
z1 = torch.matmul(W1, x)
```



# 학습 - Forward Process

## Output layer

```
#generates probabilities for each word: W2 [vocabulary_size, embedding_dims]  
W2 = Variable(torch.randn(vocabulary_size, embedding_dims).float(), requires_grad=True)  
z2 = torch.matmul(W2, z1)
```

## On Top

```
#softmax combined with log—regular softmax is not numerically stable:  
log_softmax = F.log_softmax(a2, dim=0)
```

## Compute loss

```
#nll_loss computes negative-log-likelihood on logsoftmax. y_true is context word—we want to make this  
as high as possible—because pair x, y_true is from training data—center, context  
loss = F.nll_loss(log_softmax.view(1,-1), y_true)
```

# 학습 - Backward Process

## Optimization

```
#For optimization SDG is used  
W1.data -= 0.01 * W1.grad.data  
W2.data -= 0.01 * W2.grad.data
```

## Lastly

```
#To zero gradients to make next pass clear  
W1.grad.data.zero_()  
W2.grad.data.zero_()
```

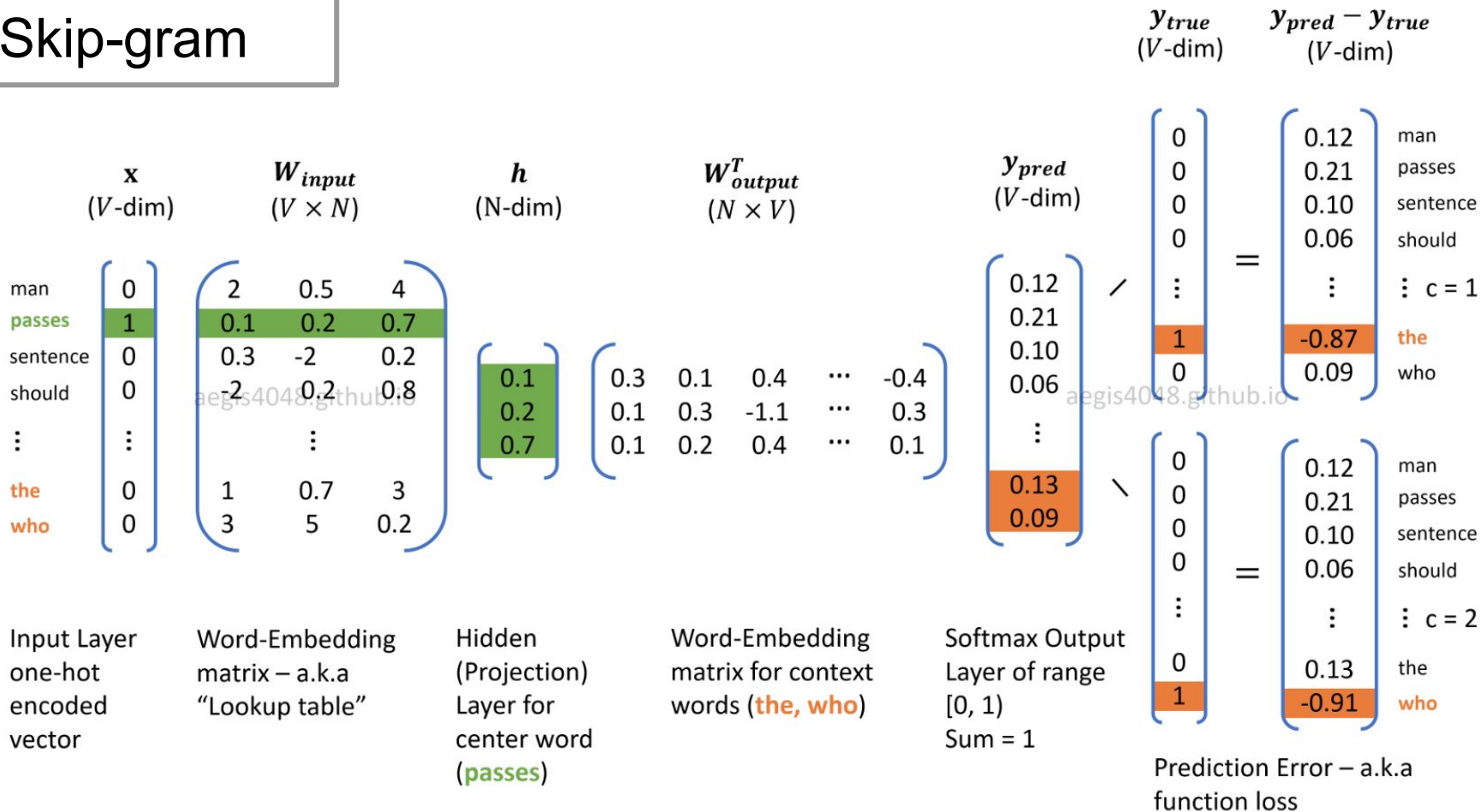
# Training

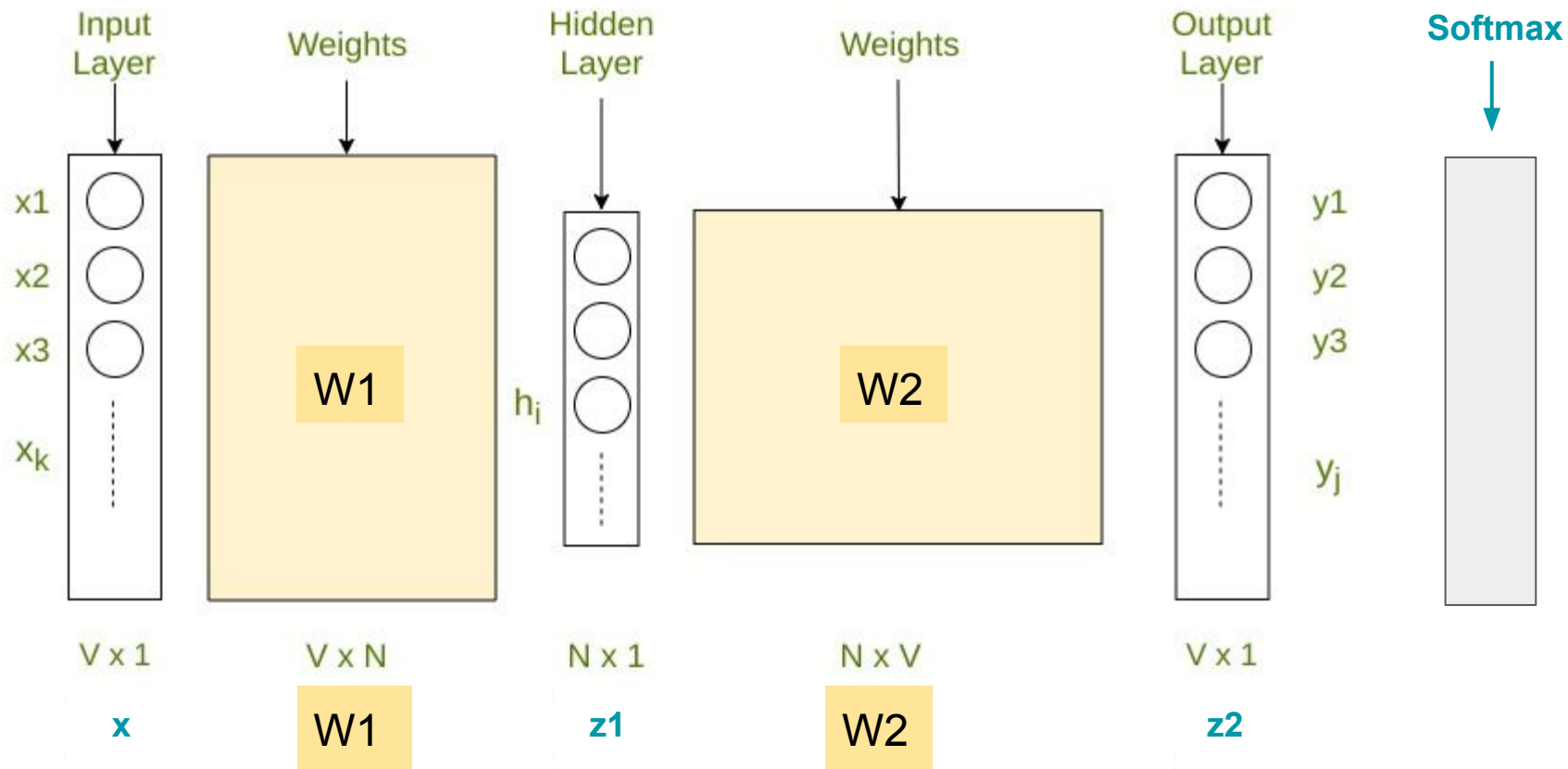
```
embedding_dims = 5
W1 = Variable(torch.randn(embedding_dims, vocabulary_size).float(), requires_grad=True)
W2 = Variable(torch.randn(vocabulary_size, embedding_dims).float(), requires_grad=True)
num_epochs, learning_rate = 100, 0.001
for epo in range(num_epochs):
    loss_val = 0
    for data, target in idx_pairs:
        x = Variable(get_input_layer(data)).float()
        y_true = Variable(torch.from_numpy(np.array([target])).long())
        z1 = torch.matmul(W1, x)
        z2 = torch.matmul(W2, z1)
        log_softmax = F.log_softmax(z2, dim=0)
        loss = F.nll_loss(log_softmax.view(1,-1), y_true)
        loss_val += loss.data[0]
        loss.backward()
        W1.data -= learning_rate * W1.grad.data
        W2.data -= learning_rate * W2.grad.data

    W1.grad.data.zero_()
    W2.grad.data.zero_()
    if epo % 10 == 0:
        print('Loss at epo {epo}: {loss_val/len(idx_pairs)}')
```

```
Loss at epo 0: 4.241989389487675
Loss at epo 10: 3.8398486052240646
Loss at epo 20: 3.5548086541039603
Loss at epo 30: 3.343840673991612
Loss at epo 40: 3.183084646293095
Loss at epo 50: 3.05673006943294
Loss at epo 60: 2.953996729850769
Loss at epo 70: 2.867735825266157
Loss at epo 80: 2.79331214427948
Loss at epo 90: 2.727727291413716
Loss at epo 100: 2.6690095041479385
```

# Skip-gram





# Negative sampling is faster than skip-gram

- Computing softmax is costly
- Weight update
  - Assume that the training corpus has 10,000 vocabs ( $V = 10000$ ) and the hidden layer is 300 ( $N = 300$ )  $\Rightarrow$  3,000,000 neurons in the output weight matrix ( $W_2$ ) that need to be updated for each training sample (Notes: for the input weight matrix ( $W_1$ ), only 300 neurons are updated for each training sample)
  - With negative sampling, if you set  $K=9$ , the model will update  $(9 + 1) * 300 = 3000$  neurons, which is only 0.1% of the 3M neurons in  $W_2$ . This is computationally much cheaper than the original Skip-Gram, and yet maintains a good quality of word vectors

## Vanilla Skip-Gram

$$\begin{array}{c}
 \text{W\_output (old)} \\
 \begin{array}{|c|c|c|}
 \hline
 -0.560 & 0.340 & 0.160 \\
 -0.910 & -0.440 & 1.560 \\
 -1.210 & -0.130 & -1.320 \\
 1.670 & -0.150 & -1.030 \\
 1.720 & -1.460 & 0.730 \\
 0.000 & 1.390 & -0.120 \\
 -0.060 & 1.520 & -0.790 \\
 0.800 & 1.850 & -1.670 \\
 -1.370 & 1.320 & -0.480 \\
 0.670 & 1.990 & -1.850 \\
 -1.520 & -1.740 & -1.860 \\
 \hline
 \end{array} \\
 (11 \times 3)
 \end{array}
 - \boxed{0.05} \times \begin{array}{c} \text{grad\_W\_output} \\ \begin{array}{|c|c|c|}
 \hline
 0.064 & 0.071 & -0.014 \\
 0.098 & 0.015 & 0.063 \\
 0.069 & 0.089 & 0.045 \\
 0.014 & 0.085 & 0.079 \\
 -0.021 & 0.067 & 0.071 \\
 -0.098 & -0.088 & 0.091 \\
 -0.072 & -0.078 & -0.089 \\
 0.046 & -0.079 & -0.053 \\
 -0.049 & -0.087 & 0.025 \\
 -0.060 & 0.092 & 0.042 \\
 0.074 & 0.050 & 0.070 \\
 \hline
 \end{array} \\
 (11 \times 3)
 \end{array} = \begin{array}{c} \text{W\_output (new)} \\ \begin{array}{|c|c|c|}
 \hline
 -0.563 & 0.336 & 0.161 \\
 -0.915 & -0.441 & 1.557 \\
 -1.213 & -0.134 & -1.322 \\
 1.669 & -0.154 & -1.034 \\
 1.721 & -1.463 & 0.726 \\
 0.005 & 1.394 & -0.125 \\
 -0.056 & 1.524 & -0.786 \\
 0.798 & 1.854 & -1.667 \\
 -1.368 & 1.324 & -0.481 \\
 0.673 & 1.985 & -1.852 \\
 -1.524 & -1.743 & -1.864 \\
 \hline
 \end{array} \\
 (11 \times 3)
 \end{array}$$

## Negative Sampling

$$\begin{array}{c}
 \text{W\_output (old)} \\
 \begin{array}{|c|c|c|}
 \hline
 -0.560 & 0.340 & 0.160 \\
 -0.910 & -0.440 & 1.560 \\
 -1.210 & -0.130 & -1.320 \\
 1.670 & -0.150 & -1.030 \\
 1.720 & -1.460 & 0.730 \\
 0.000 & 1.390 & -0.120 \\
 -0.060 & 1.520 & -0.790 \\
 0.800 & 1.850 & -1.670 \\
 -1.370 & 1.320 & -0.480 \\
 0.670 & 1.990 & -1.850 \\
 -1.520 & -1.740 & -1.860 \\
 \hline
 \end{array} \\
 (11 \times 3)
 \end{array}
 - \boxed{0.05} \times \begin{array}{c} \text{grad\_W\_output} \\ \begin{array}{|c|c|c|}
 \hline
 \text{Not computed!} \\
 \hline
 \end{array} \\
 (11 \times 3)
 \end{array} = \begin{array}{c} \text{W\_output (new)} \\ \begin{array}{|c|c|c|}
 \hline
 -0.560 & 0.340 & 0.160 \\
 -0.910 & -0.440 & 1.560 \\
 -1.210 & -0.130 & -1.320 \\
 1.670 & -0.150 & -1.030 \\
 1.720 & -1.460 & 0.730 \\
 0.000 & 1.390 & -0.120 \\
 -0.060 & 1.520 & -0.790 \\
 \text{0.798} & \text{1.849} & \text{-1.672} \\
 \text{-1.366} & \text{1.318} & \text{-0.477} \\
 \text{0.667} & \text{1.985} & \text{-1.847} \\
 \text{-1.523} & \text{-1.744} & \text{-1.858} \\
 \hline
 \end{array} \\
 (11 \times 3)
 \end{array}$$

Positive sample, w\_o  
 Negative sample, k=1  
 Negative sample, k=2  
 Negative sample, k=3

# Q&A