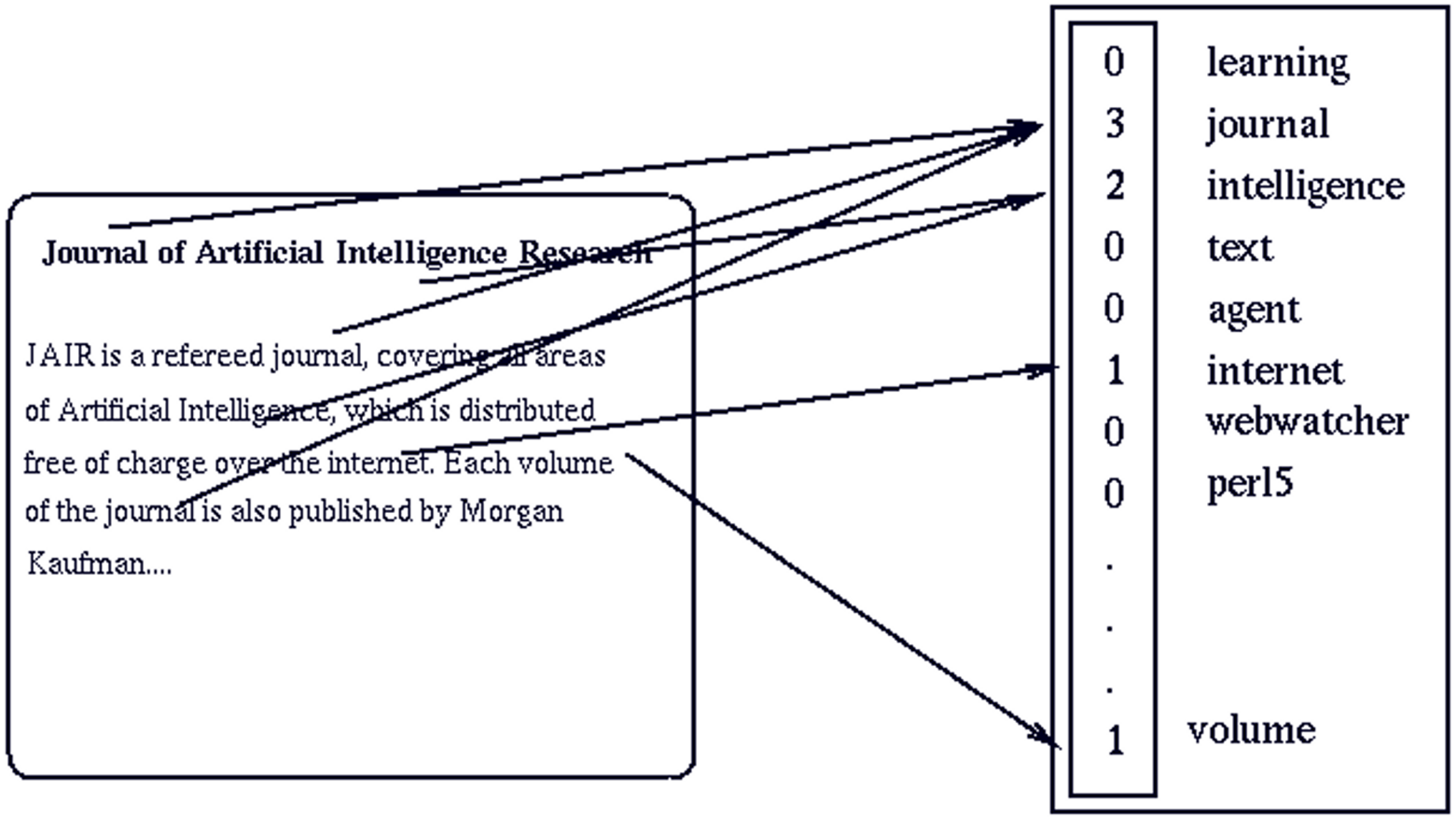
Word embeddings

Michel RIVEILL
Michel.RIVEILL@univ-cotedazur.fr

BOW model

- A bag-of-words is a representation of text that describes the occurrence of words within a document. It involves two things:
  - A vocabulary of known words.
  - A measure of the presence of known words.



BOW model – limitations

- Simple to understand and implement
- Offers a lot of flexibility for customization on your specific text data.
  - It has been used with great success in problems such as document classification or sentiment analysis

Main limitation

- Vocabulary: Vocabulary requires careful design, specifically to manage size, which impacts the sparsity of document representations.
- Sparsity: Sparse representations are more difficult to model, both for computational reasons (space and time complexity) and for informational reasons, where the challenge is that models exploit so little information in such a large representation space.
- Signification: Le fait d'ignorer l'ordre des mots ne tient pas compte du contexte et, par conséquent, de la signification des mots dans le document (sémantique).

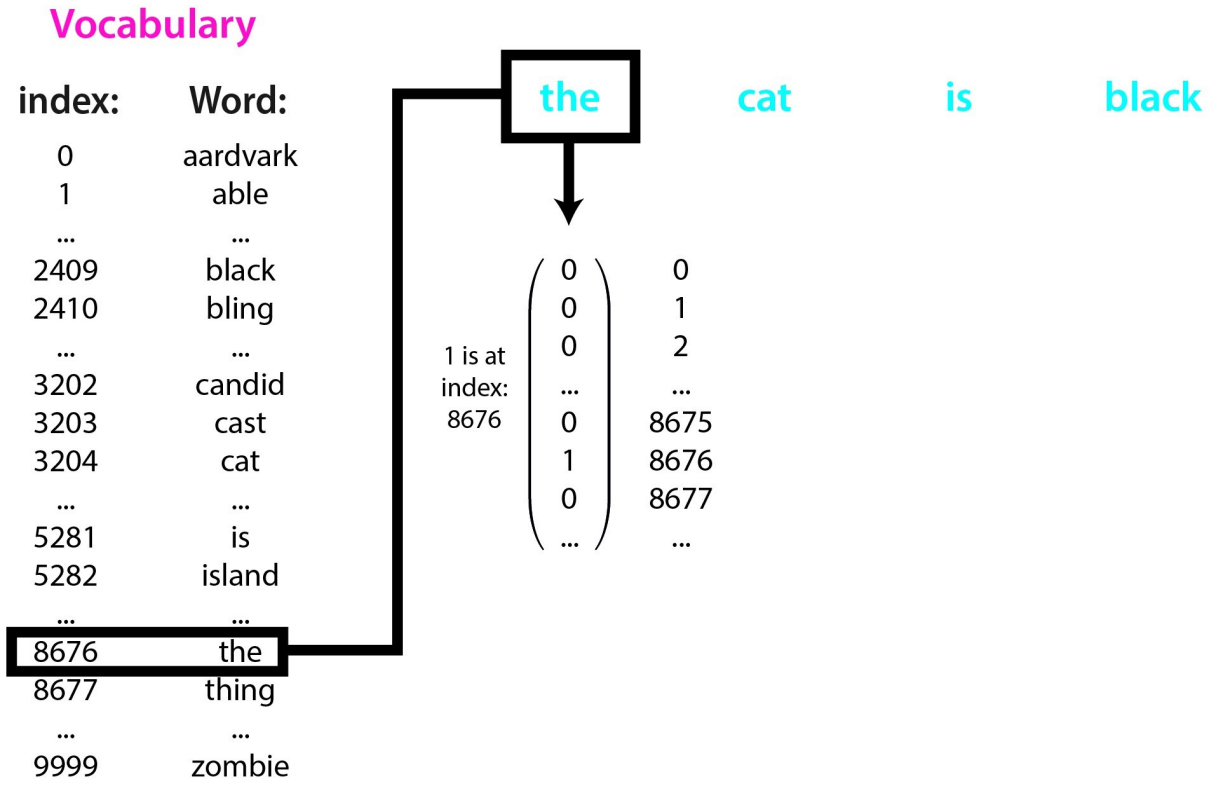
Vector Embedding of Words

- A word is represented as a vector.
- Word embeddings depend on a notion of ***word similarity***.
  - Similarity is computed using cosine.
- A very useful definition is paradigmatic similarity:
  - ***Similar words*** occur in ***similar contexts.*** They are ***exchangeable.***
  - Yesterday { POTUS / The President Biden } called a press conference.
  - "POTUS: President of the United States."

Represent a word by an One Hot Vector

First idea

- Step 1 : build a dictionary
- Step 2 : replace each word by it's rank inside the dictionary
- Step 3 : One Hot Encode each word



How to represent a word

First idea

- Build a dictionary
  - For each word, we obtain a number
- One hot encode each word

Not really a good idea

- Very sparse representation,
- Very large representation vector representation = vocabulary size
- Not express similarity between different word
  - Cosine similarity (x, y) = $\frac{x^T y}{|x||y|} \in [-1, 1]$
  - With one hot encoding, similarity is always equal to 0

The objective of the vectorization approach is to try to grasp the similarity and analogy relationships between different words.

From sparse vector to dense vector

- We take 5 words from our vocabulary ("aardvark","black","cat","duvet" and "zombie")
- Their embedding vectors created by the one-hot encoding method look like:

sparse one-hot encoding of words

aardvark	1	0	0	...	0	0	0
black	0	0	...	1	...	0	0
cat	0	0	...	1	...	0	0
duvet	0	0	...	1	...	0	0
zombie	0	0	0	...	0	0	1

An aardwark

- These vectors can be used to represent a word but do not carry any meaning.
- Whatever the 2 words chosen, whatever similarity we choose:
  - similarity(W_1,W_2) = 0

From sparse vector to dense vector

- We know that words are these rich entities with many layers of connotation and meaning.
- Let's hand-craft some semantic features for these 5 words.
- Specifically, let's represent each word as having some sort of value between 0 and 1 for four semantic qualities,:
  - "animal", "fluffiness (duveteux)", "dangerous", and "spooky (effrayant)":

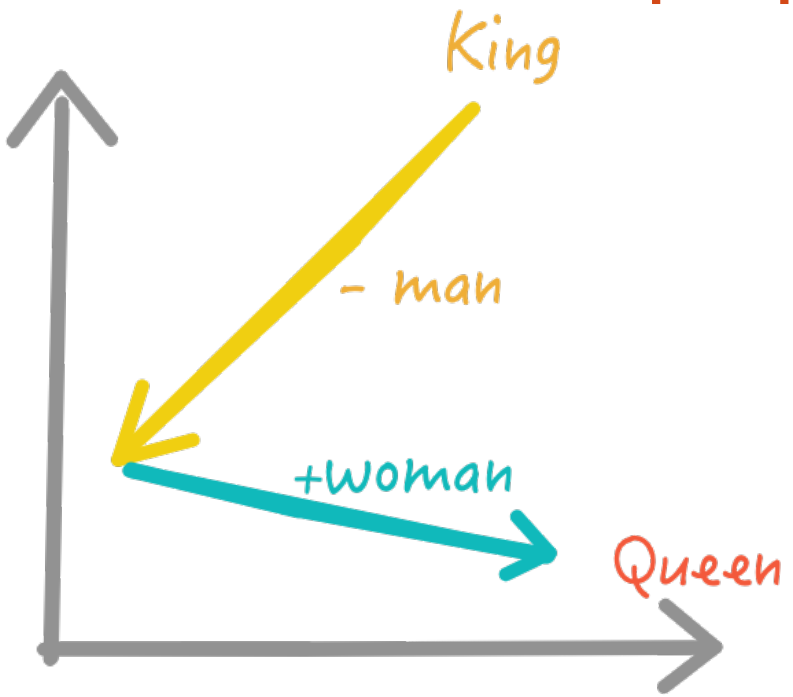
	animal	fluffiness	dangerous	spooky
aardvark	0.97	0.03	0.15	0.04
black	0.07	0.01	0.20	0.95
cat	0.98	0.98	0.45	0.35
duvet	0.01	0.84	0.12	0.02
zombie	0.74	0.05	0.98	0.93

An aardwark

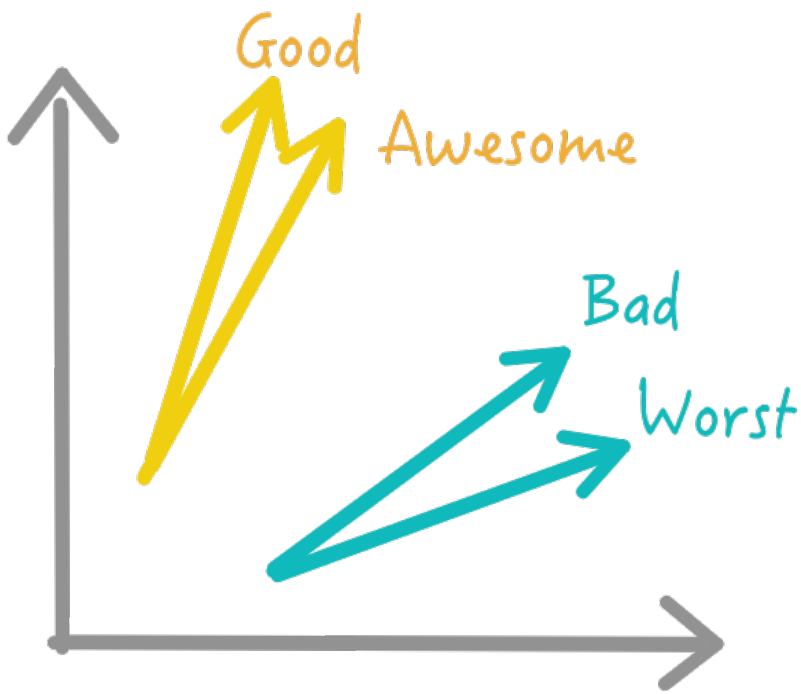
From sparse vector to dense vector

The process to transform word to vectors are called
→ word embeddings or word representations

Main properties searched



a) Learns Analogy



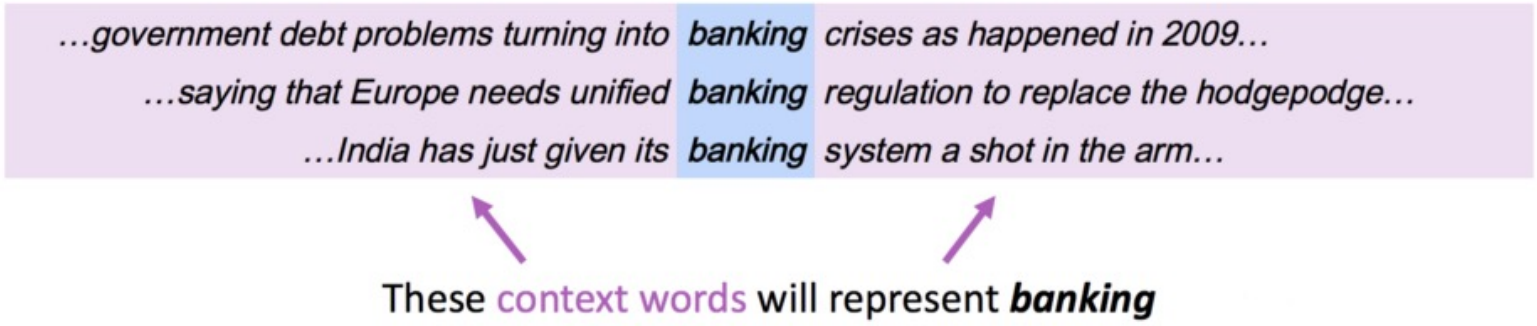
b) Similar Words have same angles

Representing words by their context

- How to learn the vector… core idea:
  - A word's meaning is given by the words that frequently appear close-by
  - 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



Word Representations

Traditional Method - Bag of Words Model + OneHot representation	Word Embeddings
• Uses one hot encoding • Each word in the vocabulary is represented by one bit position in a HUGE vector. • For example, if we have a vocabulary of 10000 words, and "Hello" is the 4th word in the dictionary, it would be represented by: 0 0 0 1 0 0 . . . . . 0 0 0 0 • Context information is not utilized	• Stores each word in as a point in space, where it is represented by a vector of fixed number of dimensions (generally 300) • Unsupervised, built just by reading huge corpus • For example, "Hello" might be represented as : [0.4, -0.11, 0.55, 0.3 . . . 0.1, 0.02] • Dimensions are basically projections along different axes, more of a mathematical concept.

How to build these magic vectors...

- How do you build these super-intelligent vectors, which seem to have such magical powers?
  - Latent Semantic Analysis/Indexing (1988)
    - Term weighting-based model
    - Consider occurrences of terms at document level.
  - Word2Vec (2013) -- Prediction-based model.
  - GloVe (2014) -- Count-based model.
    - Both consider occurrences of terms at context level.
  - ELMo (2018) – use character-level tokens as input + Recurrent Neural Network
  - BERT (2018) – use token piece tokenizer as input + Transformer based approach
    - Both are language model-based.
    - Both build a context sensitive embedding
- But, we can also use another solution:
  - Keras' Embedding layer can transform an integer into a vector of a given size specifically trained for a given problem on the vocabulary of the train set.

With Keras embedding layer

See notebook

Keras Text preprocessing.

- The preprocessing functions of text have radically changed during the year 2021.
- Be careful when you look at the tutorials on the Internet.
- The notebook example uses the new interface.

One Hot Encoding in Keras:

- Use Embedding layer in order to OneHotEncode a sequence of Integer

- from keras.layers import Input, Embedding

- inputs = Input(shape=(SEQUENCE_SIZE,))
- embedding = Embedding(vocabulary_size,
  EMBEDDING_SIZE,
  input_length=SEQUENCE_SIZE)(inputs)

- Layer (type) Output Shape Param #
  ==========
  input (InputLayer) (None, 100) 0
  _____
  embedding (Embedding) (None, 100, 300) 600000

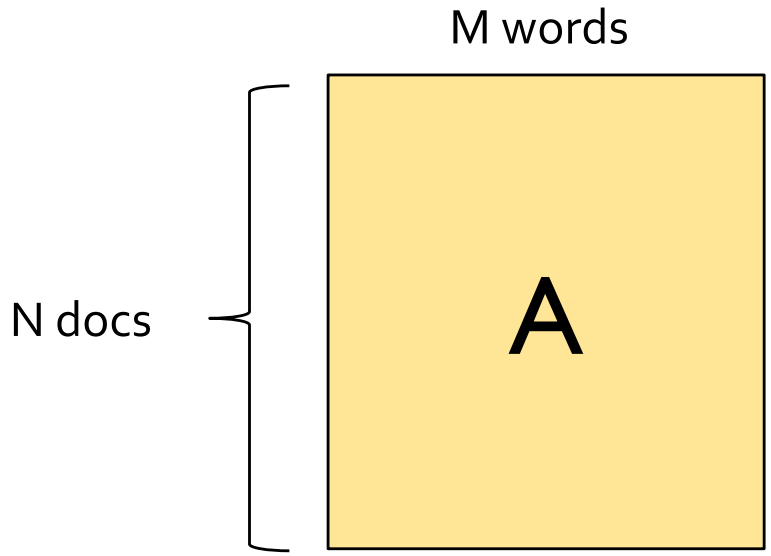
- In this example
  - SEQUENCE_SIZE = 100
  - EMBEDDING_SIZE = 300

Latent Semantic Analysis

Deerwester, Scott, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. "Indexing by latent semantic analysis." Journal of the American society for information science 41, no. 6 (1990): 391-407.

Embedding: Latent Semantic Analysis

- Latent semantic analysis studies documents in Bag-Of-Words model (1988).
  - i.e. given a matrix ***A*** encoding some documents: A_{ij} is the count* of word ***j*** in document ***i***. Most entries are 0.



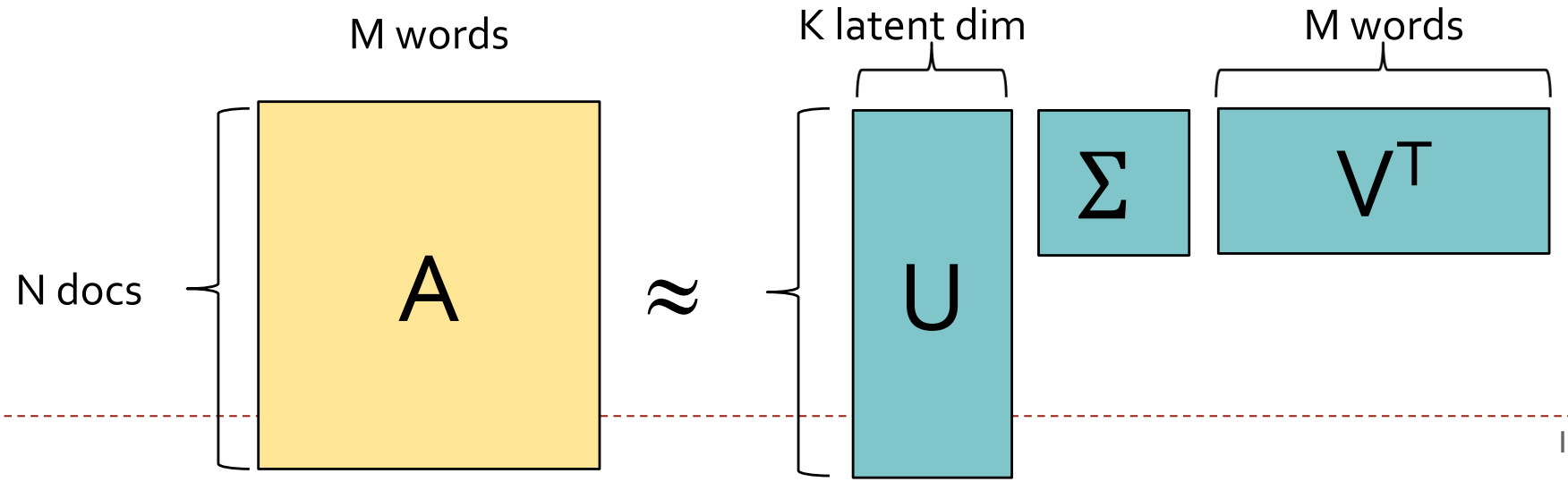
* Often tf-idf or other "squashing" functions of the count are used.

Embedding: Latent Semantic Analysis

- Low rank SVD decomposition:

$$A_{[m\times n]} = U_{[m\times r]}\Sigma_{[r\times r]}(V_{[n\times r]})^T$$

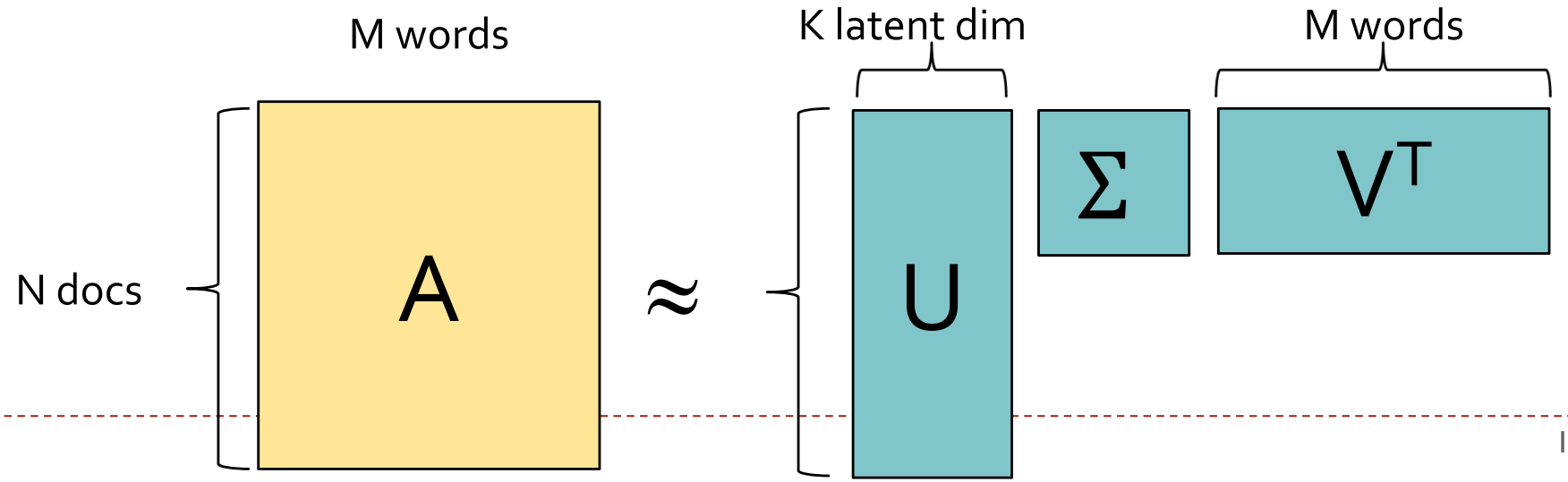
  - U : document-to-concept similarities matrix (orthogonal matrix).
  - V : word-to-concept similarities matrix (orthogonal matrix).
  - Σ : strength of each concept.
- Then given a word ***w*** (column of ***A***):
  - $\varsigma = w^T \times U$ is the embedding (encoding) of the word ***w*** in the latent space.
  - $w \approx U \times \varsigma^T = U \times (w^T \times U)^T$ is the decoding of the word ***w*** from its embedding.



18

Embedding: Latent Semantic Analysis

- $w \approx U \times \varsigma^T = U \times (w^T \times U)^T$ is the decoding of the word ***w*** from its embedding.
  - An SVD factorization gives the **best possible reconstructions** of the a word ***w*** from its embedding.
- Note:
  - The problem with this method, is that we may end up with matrices having billions of rows and columns, which makes **SVD computationally expensive and restrictive**.



Latent Semantic Analysis revisited try to capture a context

- A toy example:
  - Corpus = ["I like deep learning.", "I like NLP.", "I enjoy flying."]
- The co-occurrence matrix : put +1 if the line word is before/after the column word

counts	I	like	enjoy	deep	learning	NLP	flying	.
I	0	2	1	0	0	0	0	0
like	2	0	0	1	0	1	0	0
enjoy	1	0	0	0	0	0	1	0
deep	0	1	0	0	1	0	0	0
learning	0	0	0	1	0	0	0	1
NLP	0	1	0	0	0	0	0	1
flying	0	0	1	0	0	0	0	1
.	0	0	0	0	1	1	1	0

- It is possible to encode a word using this co-occurrence matrix.
  - For example the vector of "like" is [2, 0, 0, 1, 0, 1, 0, 0]
- But the size of a vector is equal to the size of the vocabulary.
- To reduce the size of the vocabulary,
  - an SVD decomposition or a PCA approach can be used.

Word2vec
Use pretrained embedding

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

Word2vec family

- Word2vec is a group of related algorithms
- Word2vec models are two-layer neural networks
  - Trained on very large corpus (not on the train set)
  - Produce a vector space (dimension 50, 100, 150, 300)
  - Try to capture the linguistic contexts of words.
- Word2vec associates
  - For each word in the corpus a corresponding vector in space.
  - Words that share common contexts in the corpus are located close to each other in space.

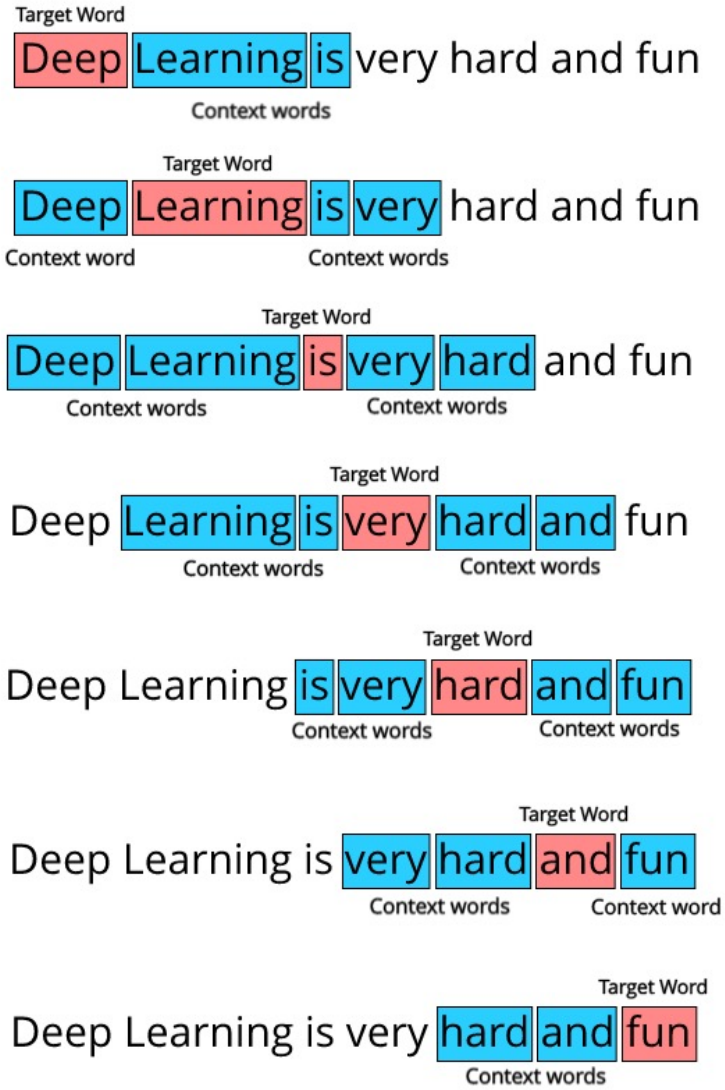
- Word2vec was created and published in 2013 by a team of researchers led by Tomas Mikolov at Google and patented.

- The algorithm was then analyzed and explained by other researchers.

- The incorporation of vectors created using the Word2vec algorithm has many advantages over previous algorithms, such as co-occurrence matrices.

word2Vec: Local contexts

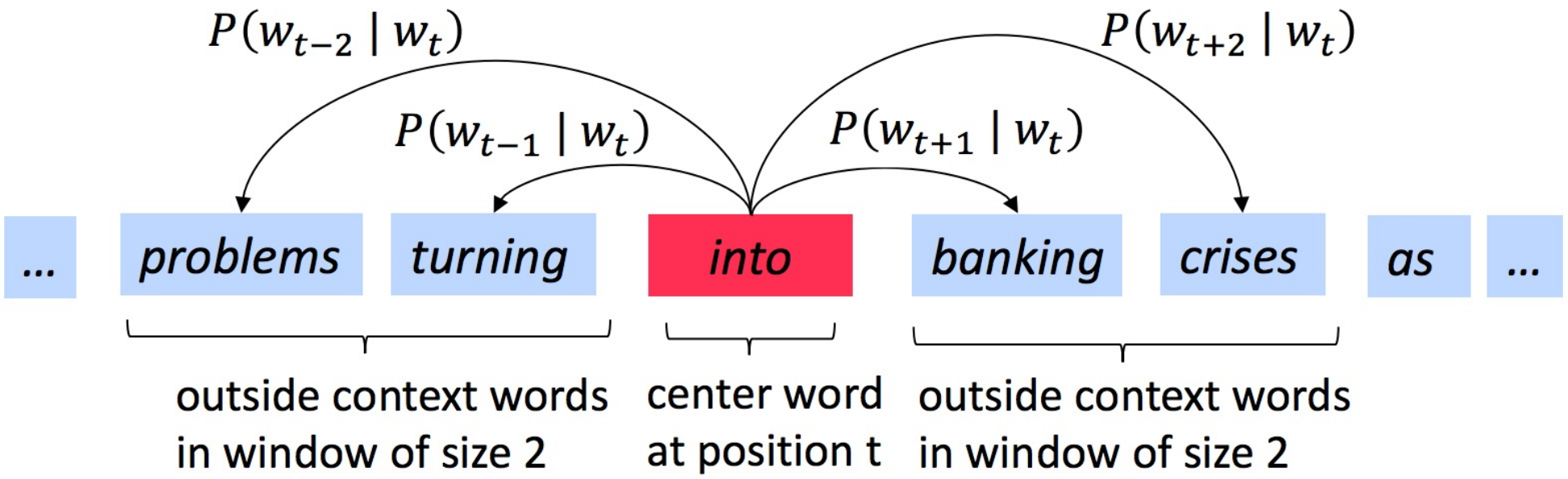
- Instead of entire documents, ***Word2Vec*** uses words ***k*** positions away from each center word.
  - These words are called **context words**.
- Example for ***k=2***:
  - Center word: red (also called focus word).
  - Context words: blue (also called target words).
- Word2Vec considers all words as center words, and all their context words.



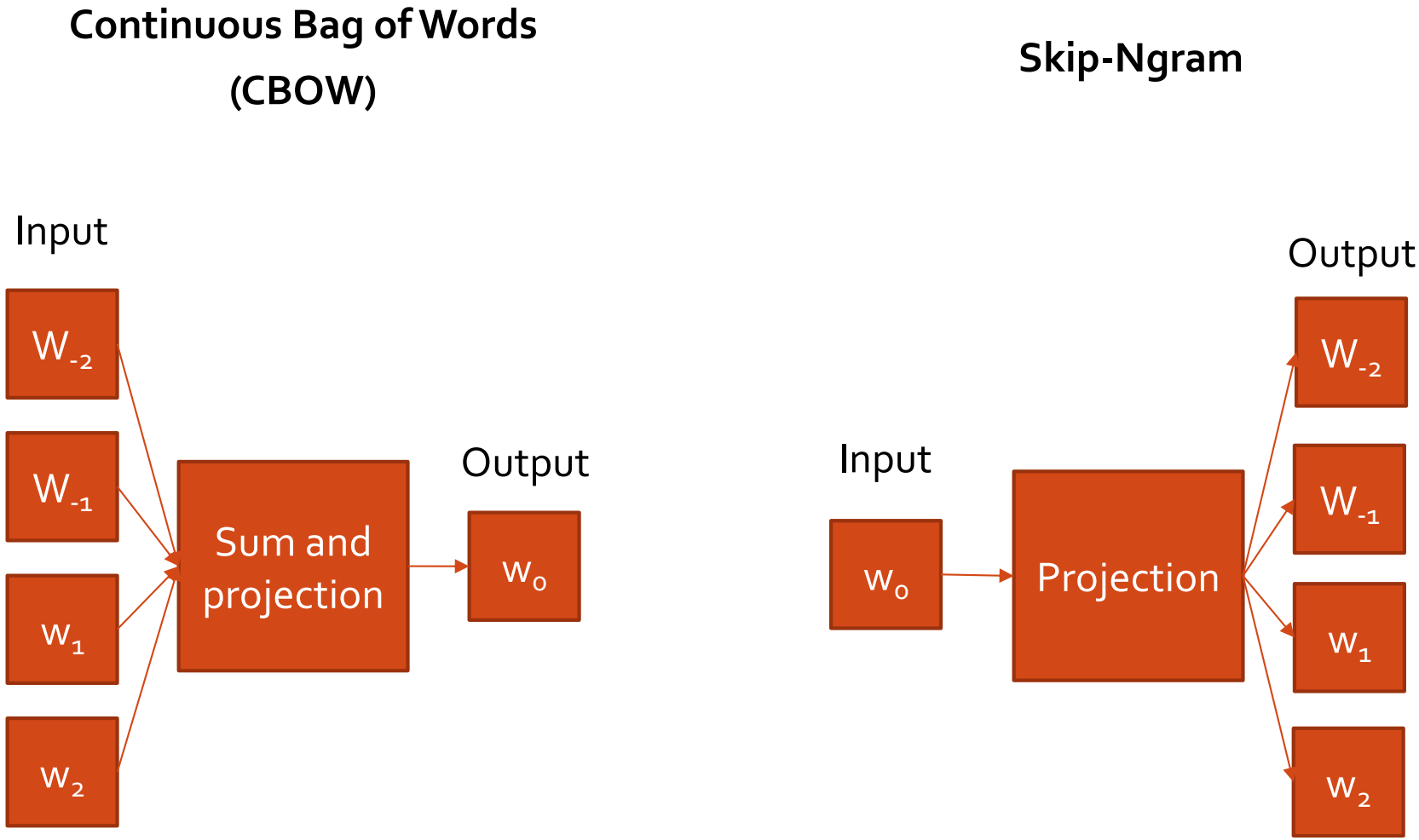
25

Word2vec

- In order to construct a Word2vec embedding
  - We need have a large corpus of text (Wikipedia ?)
  - 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



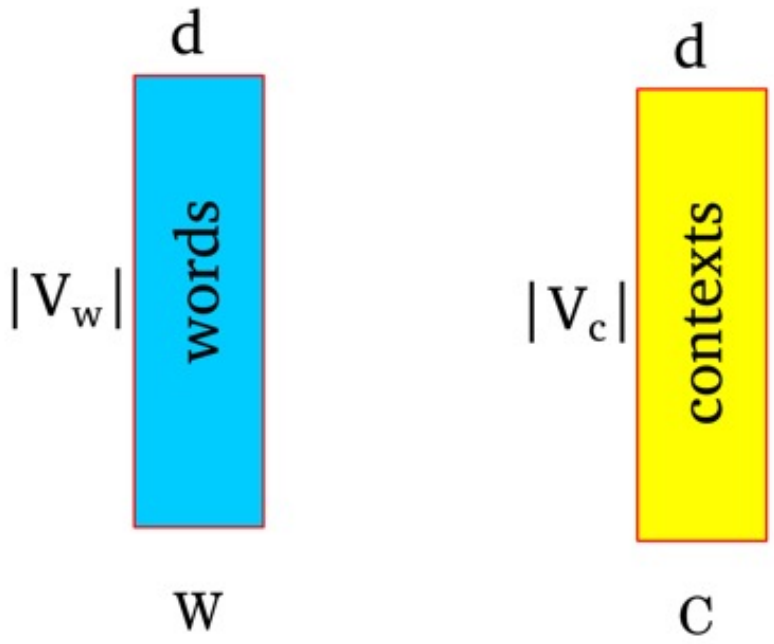
Word2Vec: main context representation models



Word2Vec is a predictive model

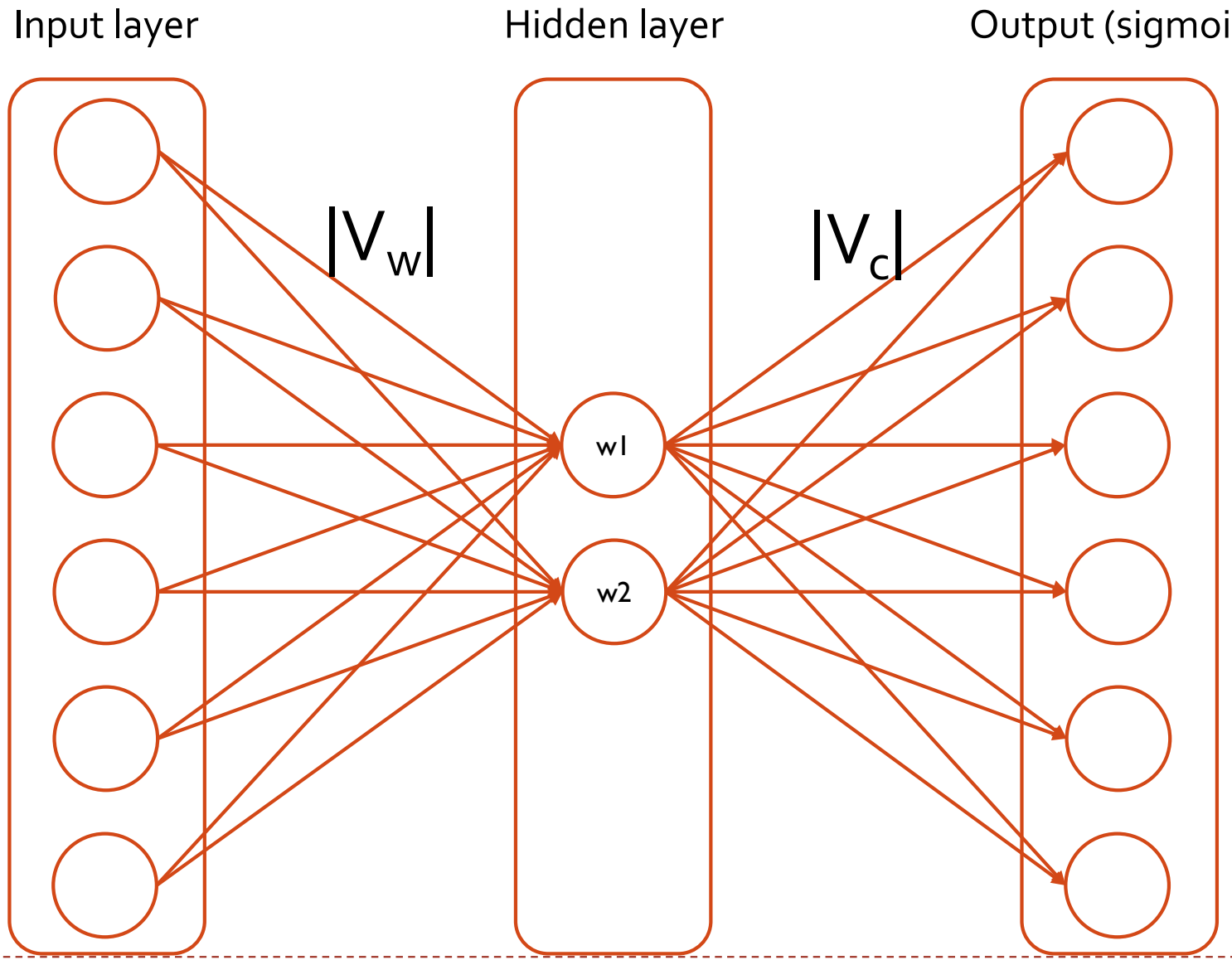
How does word2Vec work? Skip-gram approach

- Represent each word as a ***d*** dimensional vector.
- Represent each context as a ***d*** dimensional vector.
- Initialize all vectors to random weights.
- Arrange vectors in two matrices, W and C.

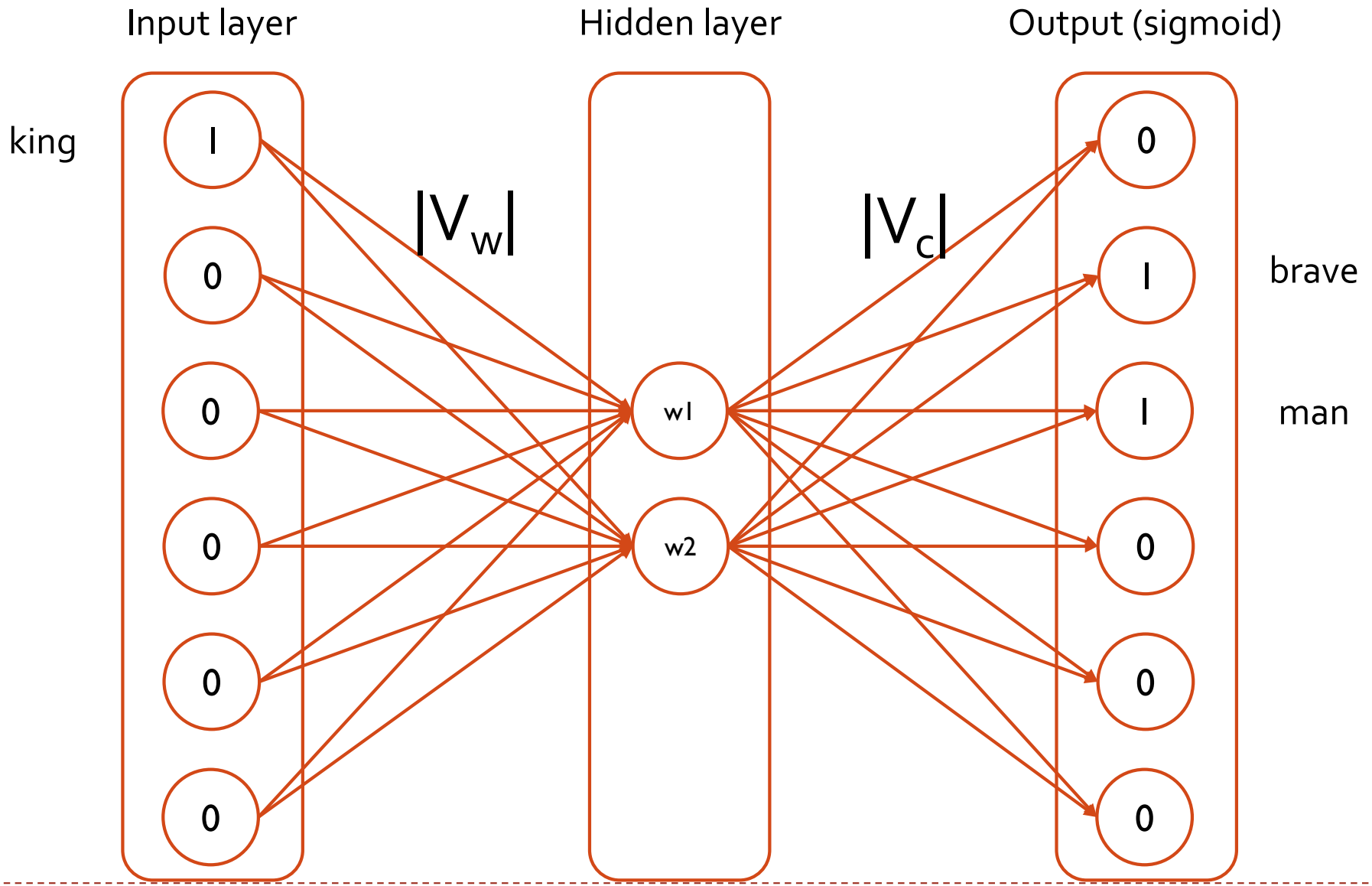


28

How does word2Vec work? Skip-gram approach

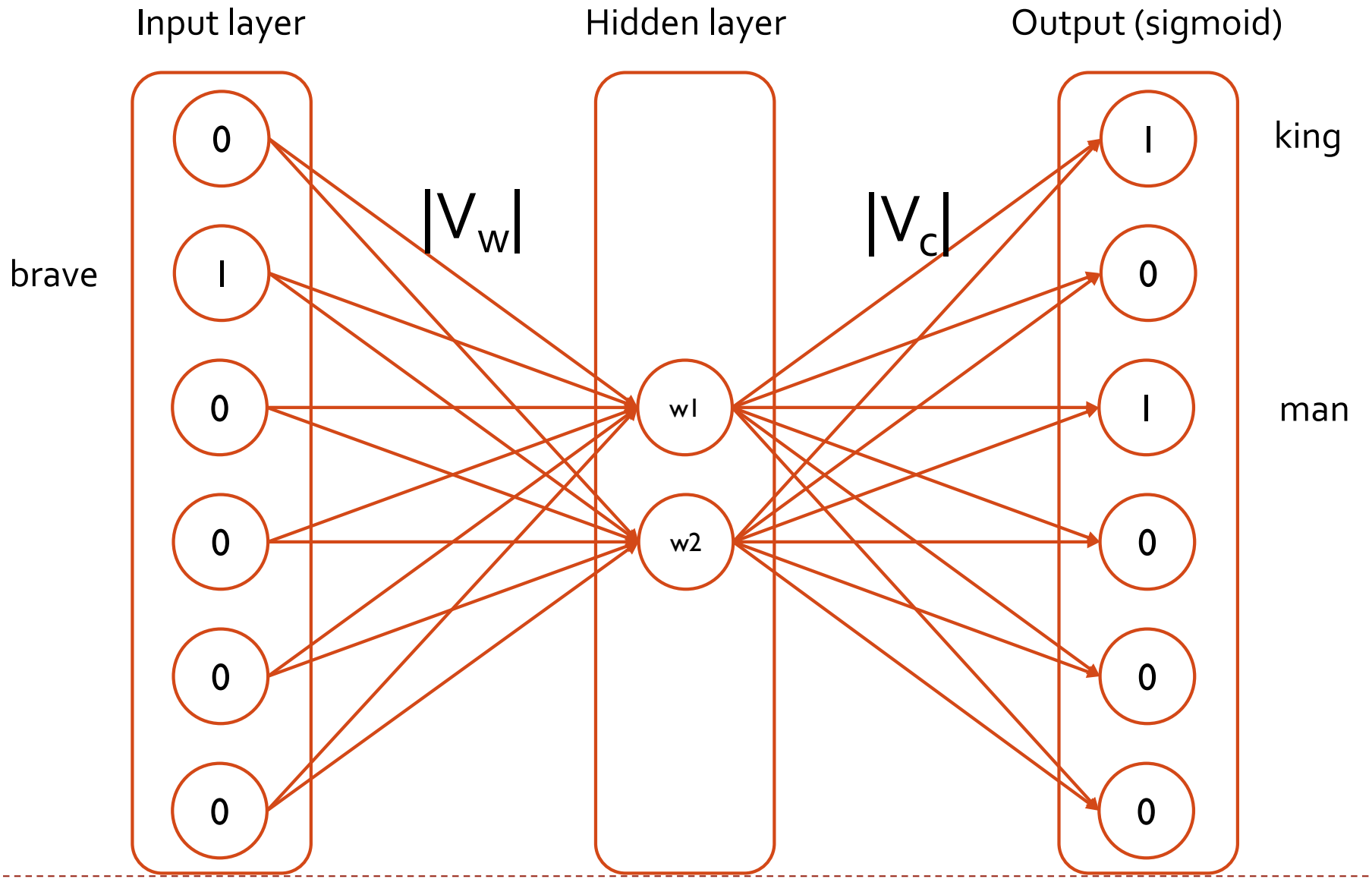


How does word2Vec work? Skip-gram approach



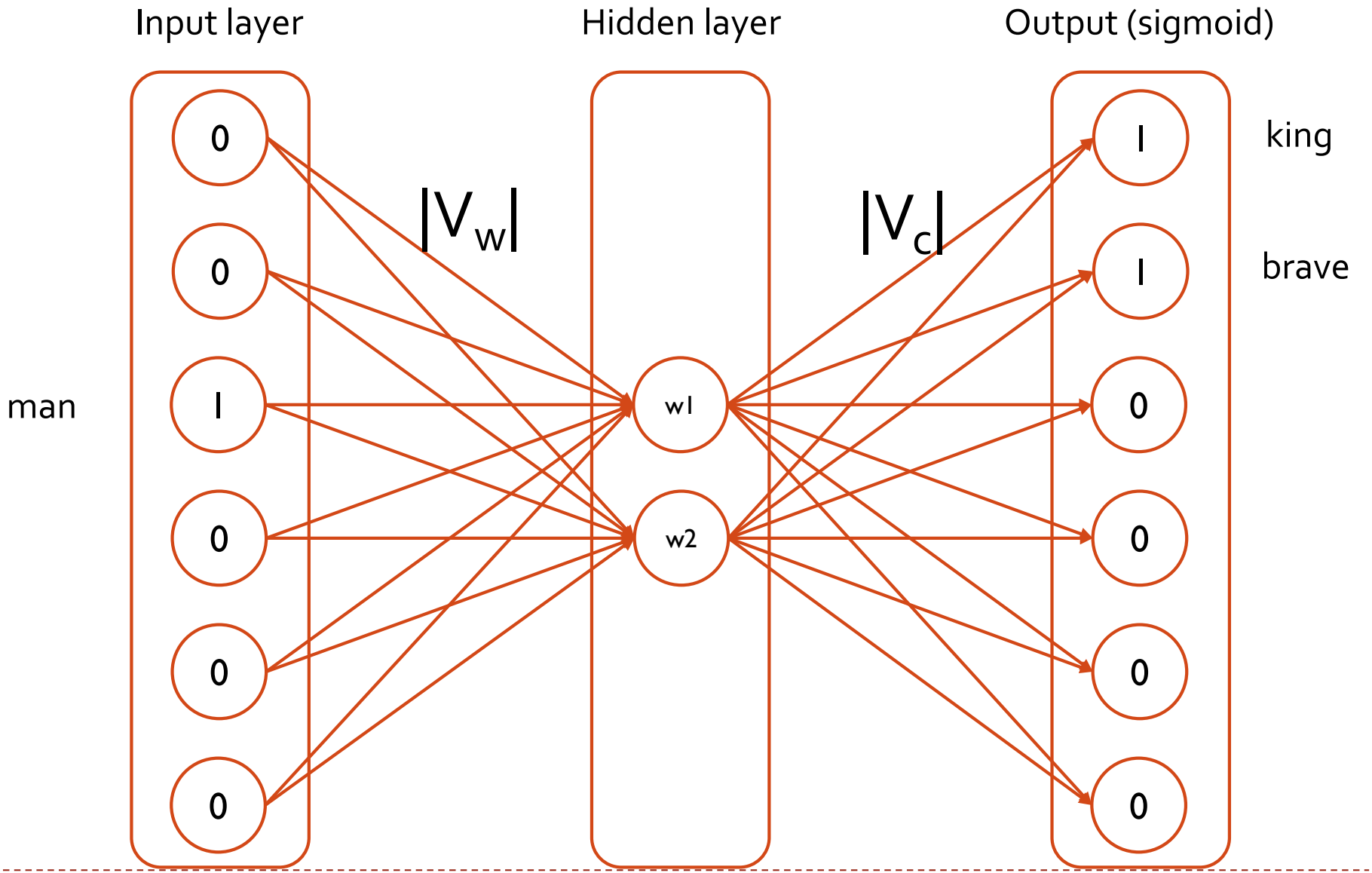
30

How does word2Vec work? Skip-gram approach

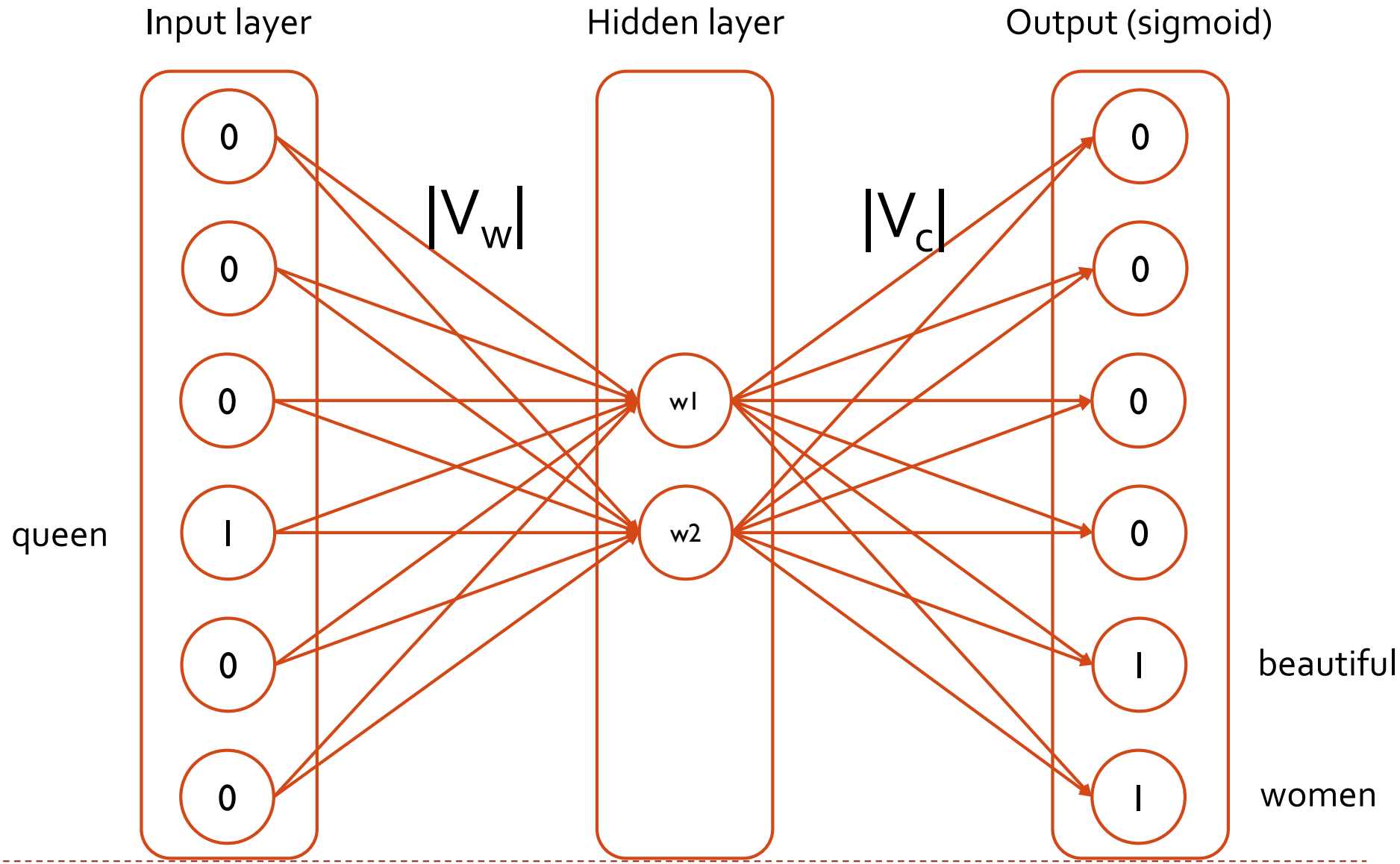


31

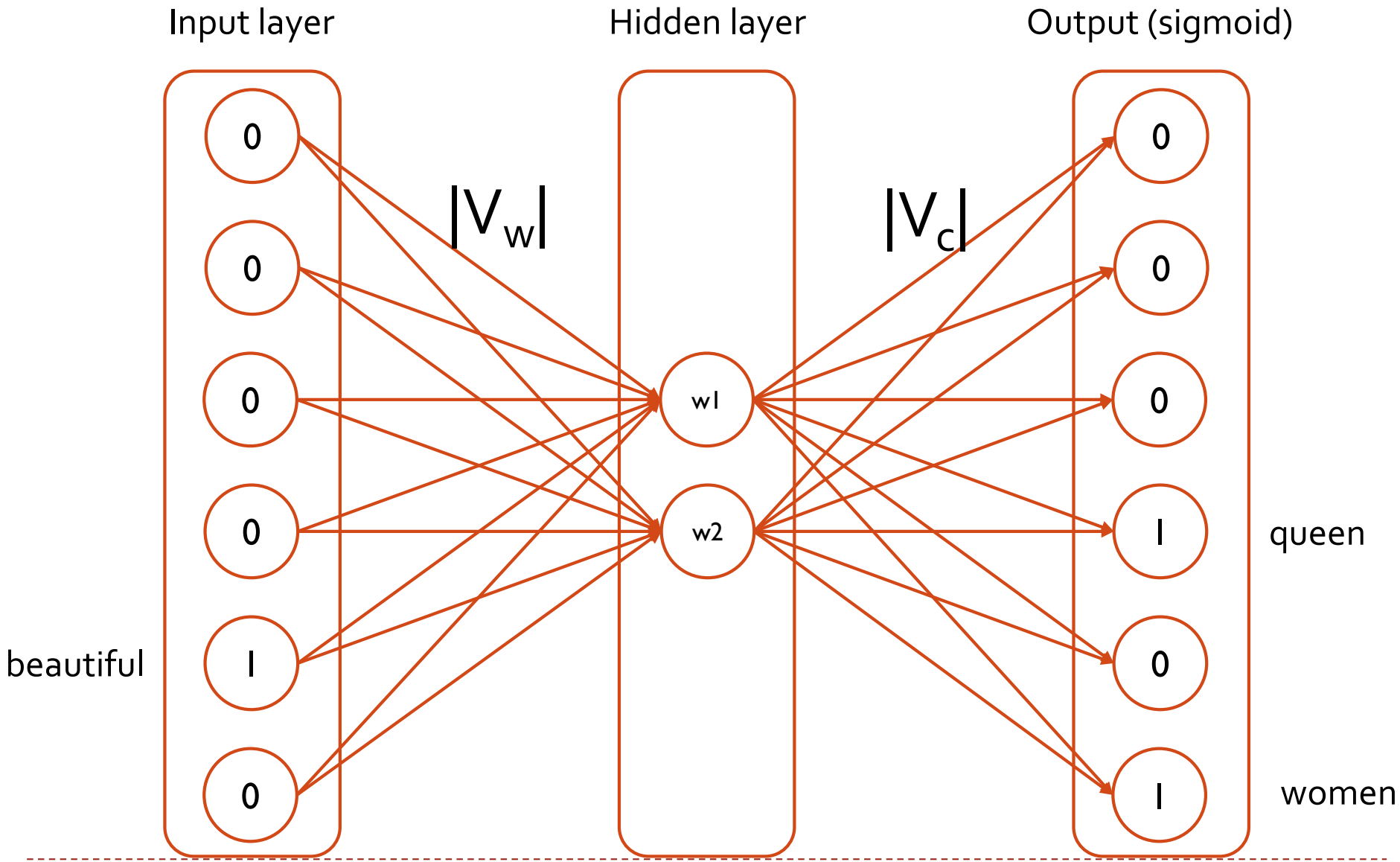
How does word2Vec work? Skip-gram approach



How does word2Vec work? Skip-gram approach

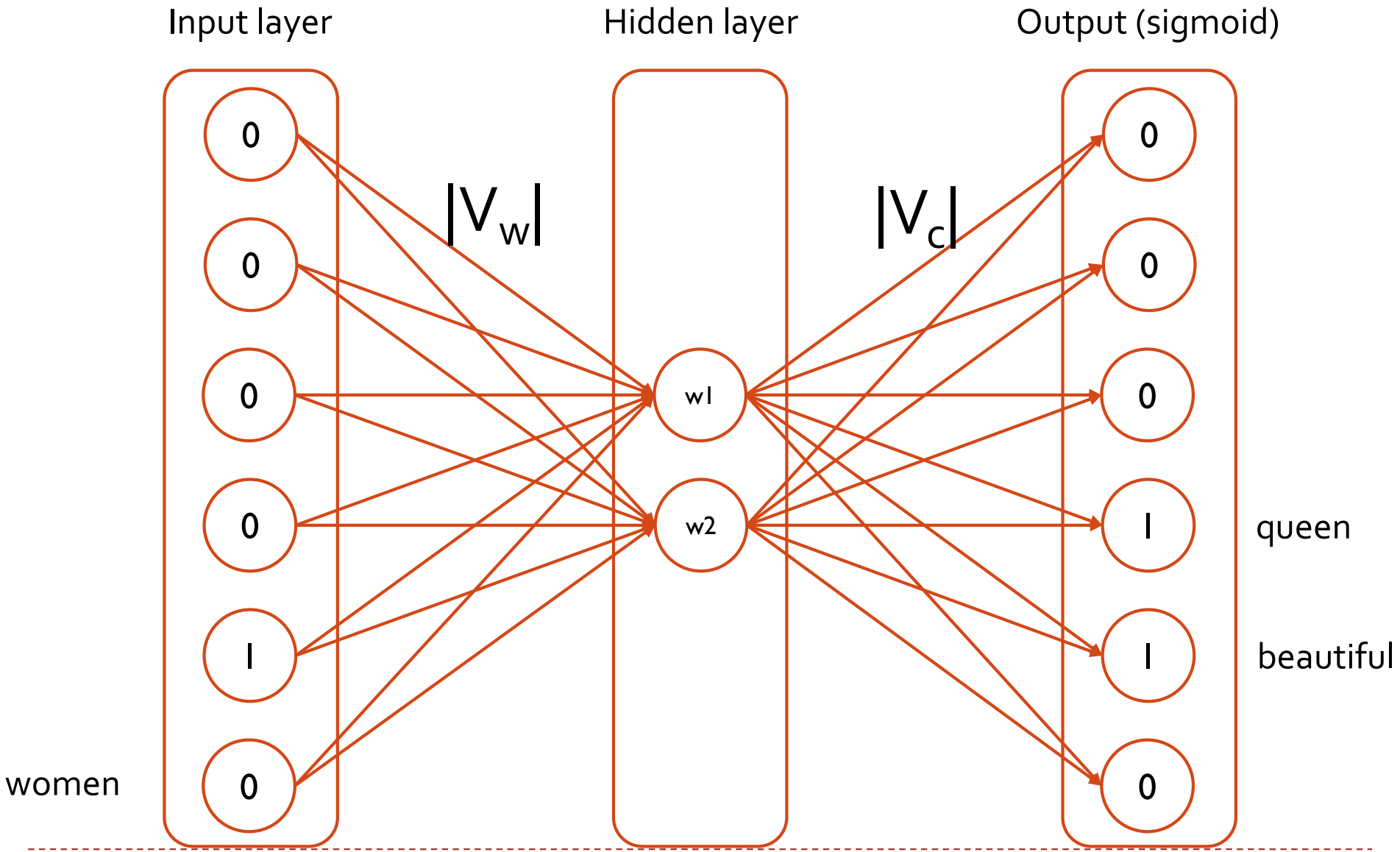


How does word2Vec work? Skip-gram approach



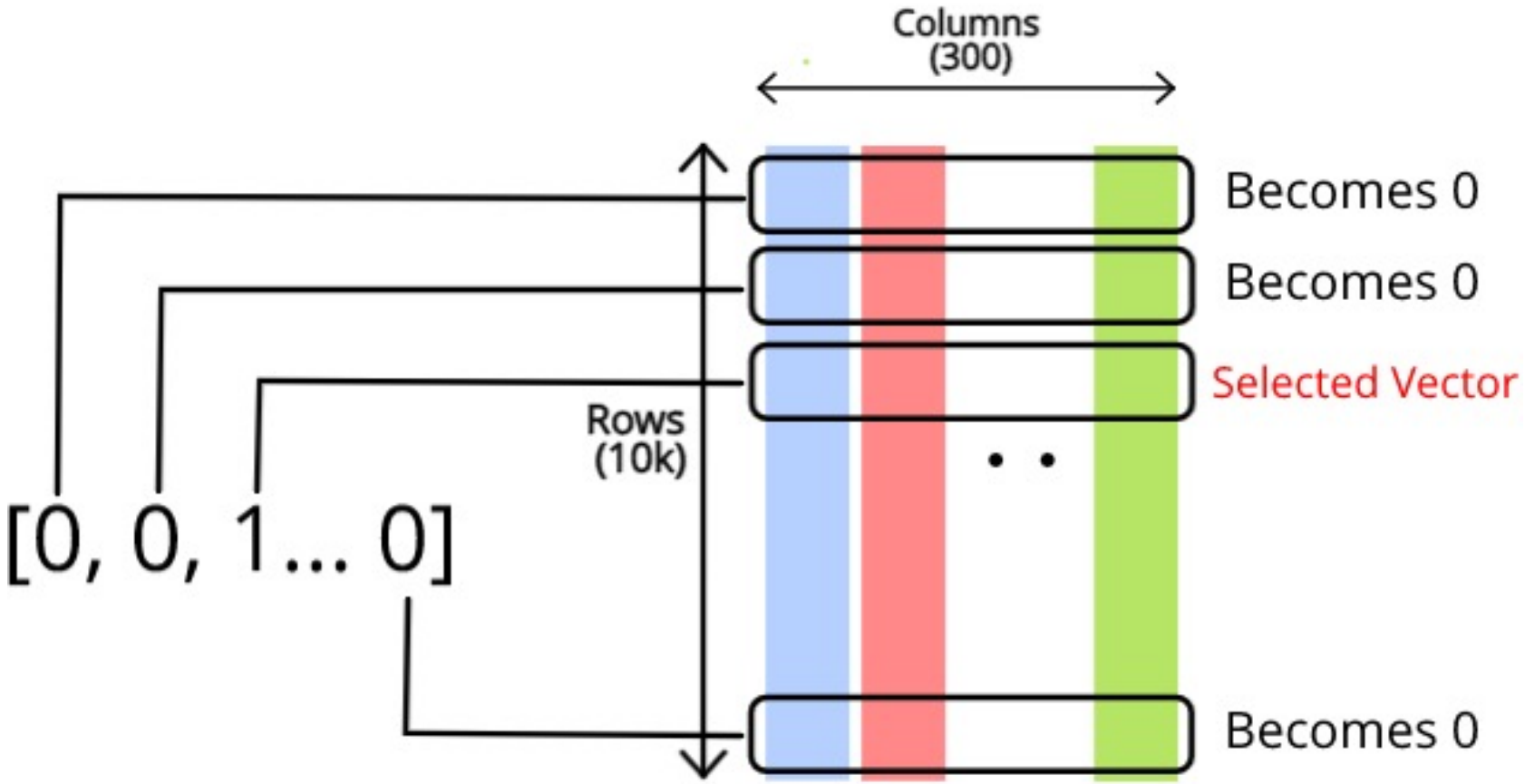
34

How does word2Vec work? Skip-gram approach

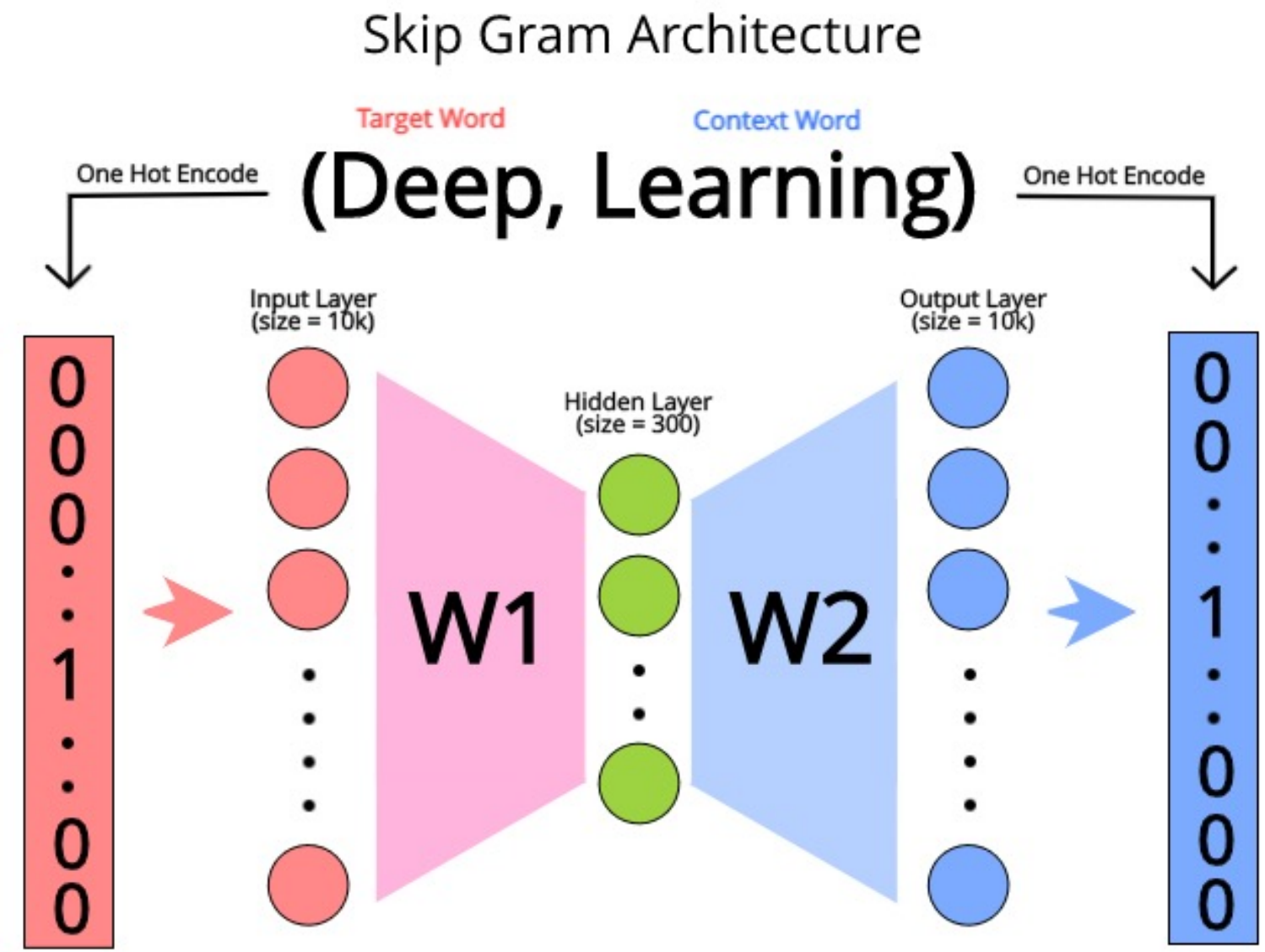


How does word2Vec work? Skip-gram approach

Use the $|V_W|$ vector as word embedding matrix



Skip Gram approach



Skip Gram in Keras

- inputs = Input(shape=(1,), dtype='int32', name="input")

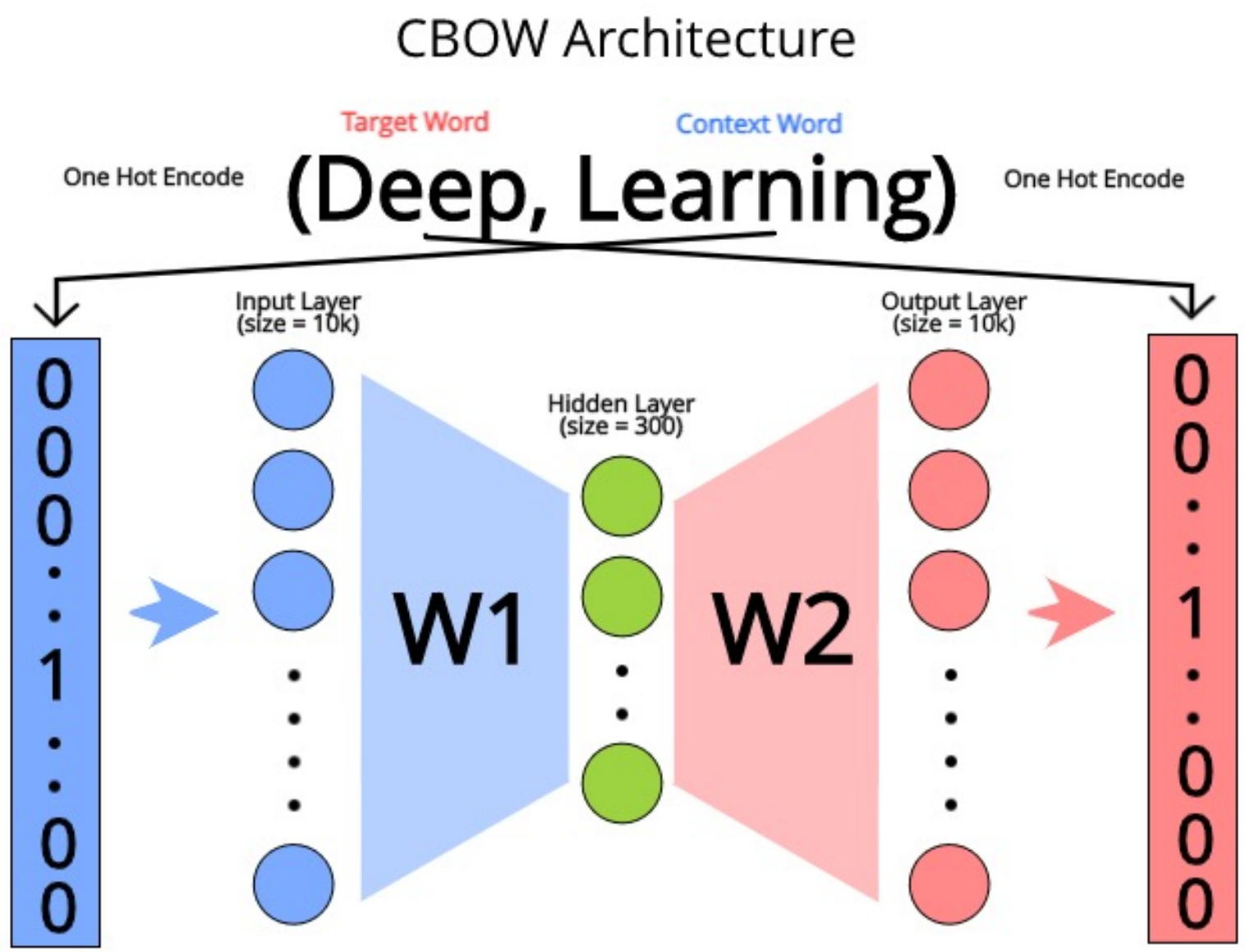
- embedding = Embedding(vocabulary_size, EMBEDDING_SIZE,
  input_length=1)(inputs)
- flatten = Flatten()embedding)
- output = Dense(vocabulary_size, activation='softmax')(flatten)

- # Model compilation
- model_skipgram = Model(inputs=inputs, outputs=output)
- model_skipgram.compile(optimizer=op, loss='categorical_crossentropy')

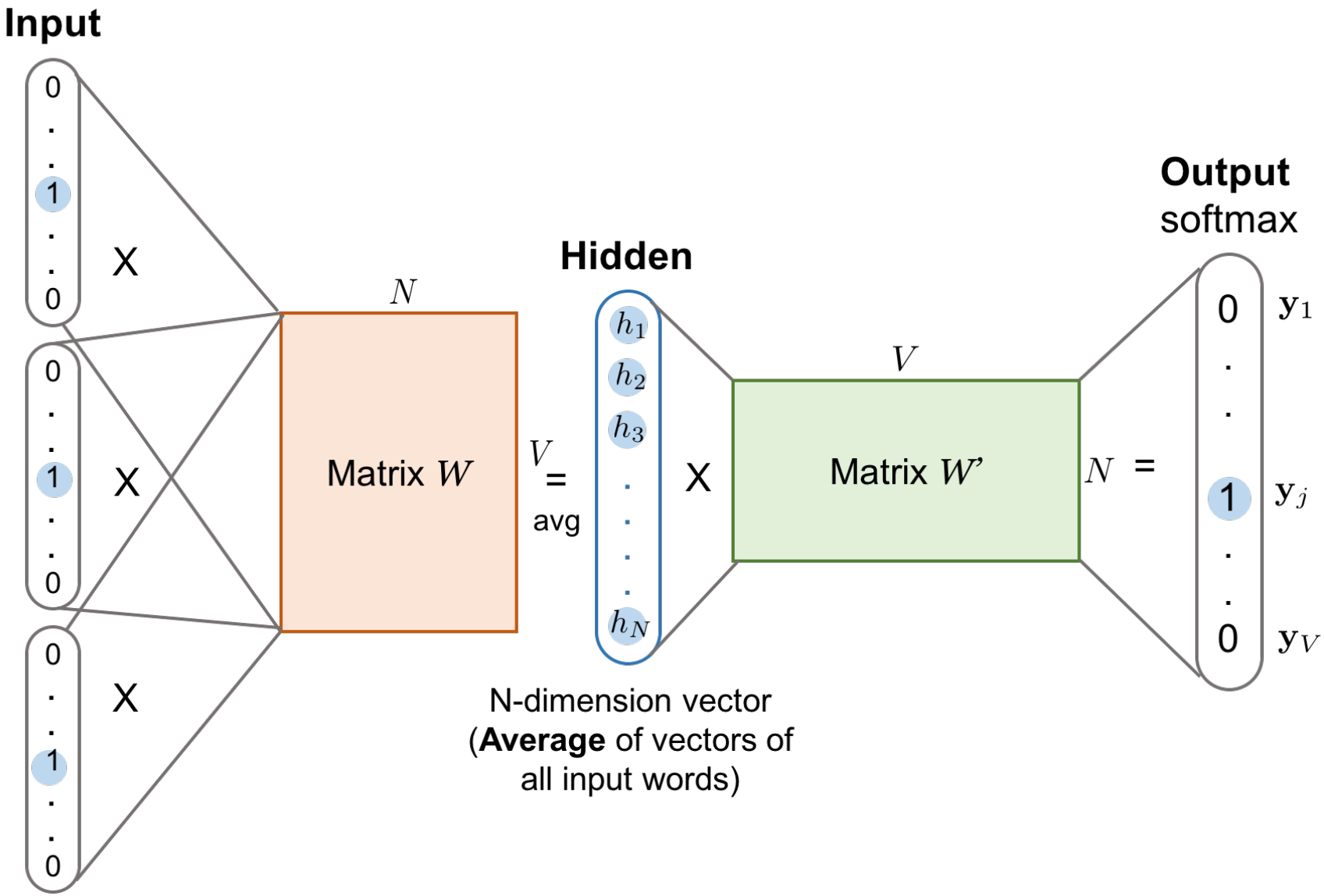
- # Keep embedding weight
- embedding_weights_skipgram = model_skipgram.get_weights()[0]

- # Use embedding weight
- weigths[embedding_weights_skipgram['my_word']]

CBow


CBOW Architecture
Target Word
Context Word
One Hot Encode
(Deep, Learning)
One Hot Encode
Input Layer
(size = 10k)
Hidden Layer
(size = 300)
Output Layer
(size = 10k)
0
0
0
.
.
1
.
.
0
0
W1
W2
0
0
.
.
1
.
.
0
0
0

Cbow
We can use directly all context word

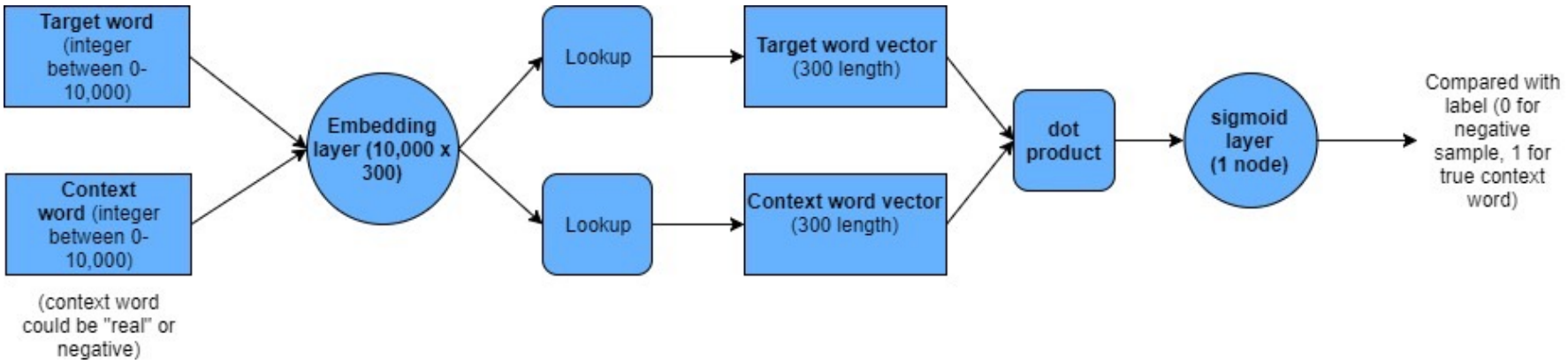


Cbow in Keras

- inputs = Input(shape=(2*SLIDDING_WINDOWS,), dtype='int32', name="input")
- embedding = Embedding(vocabulary_size, EMBEDDING_SIZE,
  input_length=2*SLIDDING_WINDOWS)(inputs)
- mean_embedding = Lambda(lambda x: K.mean(x, axis=1),
  output_shape=(EMBEDDING_SIZE,))(embedding)
- flatten = Flatten()(mean)
- output = Dense(vocabulary_size, activation='softmax')(added)
- # Model compilation
- model_cbow = Model(inputs=inputs, outputs=output)
- model_cbow.compile(optimizer=op, loss='categorical_crossentropy')
- # Keep embedding weight
- embedding_weights_cbow = model_cbow.get_weights()[0]
- # Use embedding weight
- weigths[embedding_weights_cbow['my_word']]

Skip gram model with negative sampling

- It is quite difficult to converge a Skip Gram or CBow model in particular because of the use of a One Hot encoding as a label.
- The idea of a sampling approach is to build a list of word pairs and to associate as a label True or False depending on whether the two words can be found in the same context or not.



Negative sampling in Keras

- Step 1: build sampling list probabilities
  - Use sequence module from Keras
  - *""" Generates a word rank-based probabilistic sampling table.*

    Used for generating the sampling_table argument for skipgrams.
    sampling_table[i] is the probability of sampling the word i-th most common word in a dataset (more common words should be sampled less frequently, for balance).

    The sampling probabilities are generated according to the sampling distribution used in word2vec

    Arguments
    ***size:** Int, number of possible words to sample.*
    ***sampling_factor:** The sampling factor in the word2vec formula (1e-05 by default).*
    """

  - from keras.preprocessing.sequence import make_sampling_table
  - sampling_table = make_sampling_table(vocabulary_size)

Negative sampling in Keras

- Step 2: build a list of pair of words based of the previous list of probabilities
  - Use sequence module from Keras
  - *'''This function transforms a sequence of word indexes (list of integers) into tuples of words of the form:*
    (word, word in the same window), with label 1 (positive samples).
    (word, random word from the vocabulary), with label 0 (negative samples).

    Arguments
    ***sequence**: A word sequence (sentence), encoded as a list of word indices (integers). Word indices are expected to match the rank of the words in a reference dataset (e.g. 10 would encode the 10-th most frequently occurring token).*
    ***vocabulary_size**: Int, maximum possible word index + 1 (0 is used for unknow word)*
    ***window_size**: Int, size of sampling windows (technically half-window). The window of a word w_i will be [i - window_size, i + window_size+1].*
    ***sampling_table**: 1D array of size vocabulary_size where the entry i encodes the probability to sample a word of rank i.*
    '''

  - from keras.preprocessing.sequence import skipgrams
  - couples, labels = skipgrams(data, vocabulary_size,
    window_size=SLIDDING_WINDOWS,
    sampling_table=sampling_table)

Negative sampling in Keras
The network

- input_target = Input((1,), name="target")
- input_context = Input((1,), name="context")

- embedding = Embedding(vocabulary_size, EMBEDDING_SIZE, input_length=1, name='embedding')
- target = embedding(input_target)
- target = Reshape((EMBEDDING_SIZE, 1), name="target_reshape")(target)
- context = embedding(input_context)
- context = Reshape((EMBEDDING_SIZE, 1), name="context_reshape")(context)

- dot_product = Dot(axes=1, normalize=False, name="dot_product")([target, context])
- dot_product = Flatten()(dot_product)

- output = Dense(1, activation='sigmoid', name="sofmax")(dot_product)

- # create the primary training model
- model_skneg = Model(input=[input_target, input_context], output=output)
- model_skneg.compile(loss='binary_crossentropy', optimizer='rmsprop')

- # Keep embedding weight
- embedding_weights_skneg = model_skneg.get_weights()[0]

Word2vec

- These representations are very good at encoding dimensions of similarity
  - We can visualize the learned vectors by projecting them down to 2 dimensions using for instance something like the t-SNE dimensionality reduction technique.
- Some fun word2vec analogies

Expression	*Nearest token*
Paris - France + Italy	Rome
bigger - big + cold	colder
sushi - Japan + Germany	bratwurst
Cu - copper + gold	Au
Windows - Microsoft + Google	Android
Montreal Canadiens - Montreal + Toronto	Toronto Maple Leafs

Using word2vec in your research . . .

- Easiest way to use it is via the Gensim libarary for Python (tends to be slowish, even though it tries to use C optimizations like Cython, NumPy)
  - https://radimrehurek.com/gensim/models/word2vec.html

- Original word2vec C code by Google
  - https://code.google.com/archive/p/word2vec/

Train your own Word2Vec model with Gensim lib in Python

- Unsupervized approach → you only need a set of text
  - Here brown corpus from nltk
- from gensim.models import Word2Vec
- from nltk.corpus import brown

- # build a word2vec model from a corpus
- b = Word2Vec(brown.sents())

- b.most_similar('man', topn=4)
- For brown
  - [('woman', 0.874), ('girl', 0.870), ('boy', 0.838), ('young', 0.784)]
- For gutenberg
  - [('person', 0.736), ('body', 0.713), ('woman', 0.710), ('lady', 0.677)]

Word2vec in Python

- There are also pre-trained corpora
- Google's pre-trained model for gensim
  - https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit
  - Vocabulary of 3 million words
  - Only english
  - The vector length is 300 features
- Wiki pre-trained model
  - https://fasttext.cc/docs/en/pretrained-vectors.html
  - 294 languages
  - Vector length is 300

Word2vec in Python

- See Keras tutorial in order to use efficiently this embedding
  - https://blog.keras.io/using-pre-trained-word-embeddings-in-a-keras-model.html
  - from keras.layers import Embedding
  - embedding_layer = Embedding(len(word_index) + 1,
    EMBEDDING_DIM,
    weights=[embedding_matrix],
    input_length=MAX_SEQ_LENGTH,
    trainable=False)

Summary

2 approaches are preferable

- Build a specific embedding
  - for the vocabulary used and the target task
  - i.e. use the Embedding proposed by Keras.
  - **Disadvantage:** all the words of the **test set** not known by the train set will have a null embedding
- Reuse an existing embedding
  - it is necessary to find an embedding as close as possible to the vocabulary contained in the dataset
  - **Disadvantage:** all words from the **train** and the **test** set that are not embedding dataset will have a null embedding
  - It's some time possible to fine tune the embedding
- Main problem with these approach
  - OOV (Out-of-Vocabulary): the unknown vocabulary has a null vector
  - Polysemy: a word always has the same vector regardless of the context

Conclusion

- The embedding of words has become very important in the last two years with the appearance of new models (BERT, Elmo, etc.) that we will study later as they do not yet have sufficient neural networks.