

Deep Learning **Deep representations of words|items**

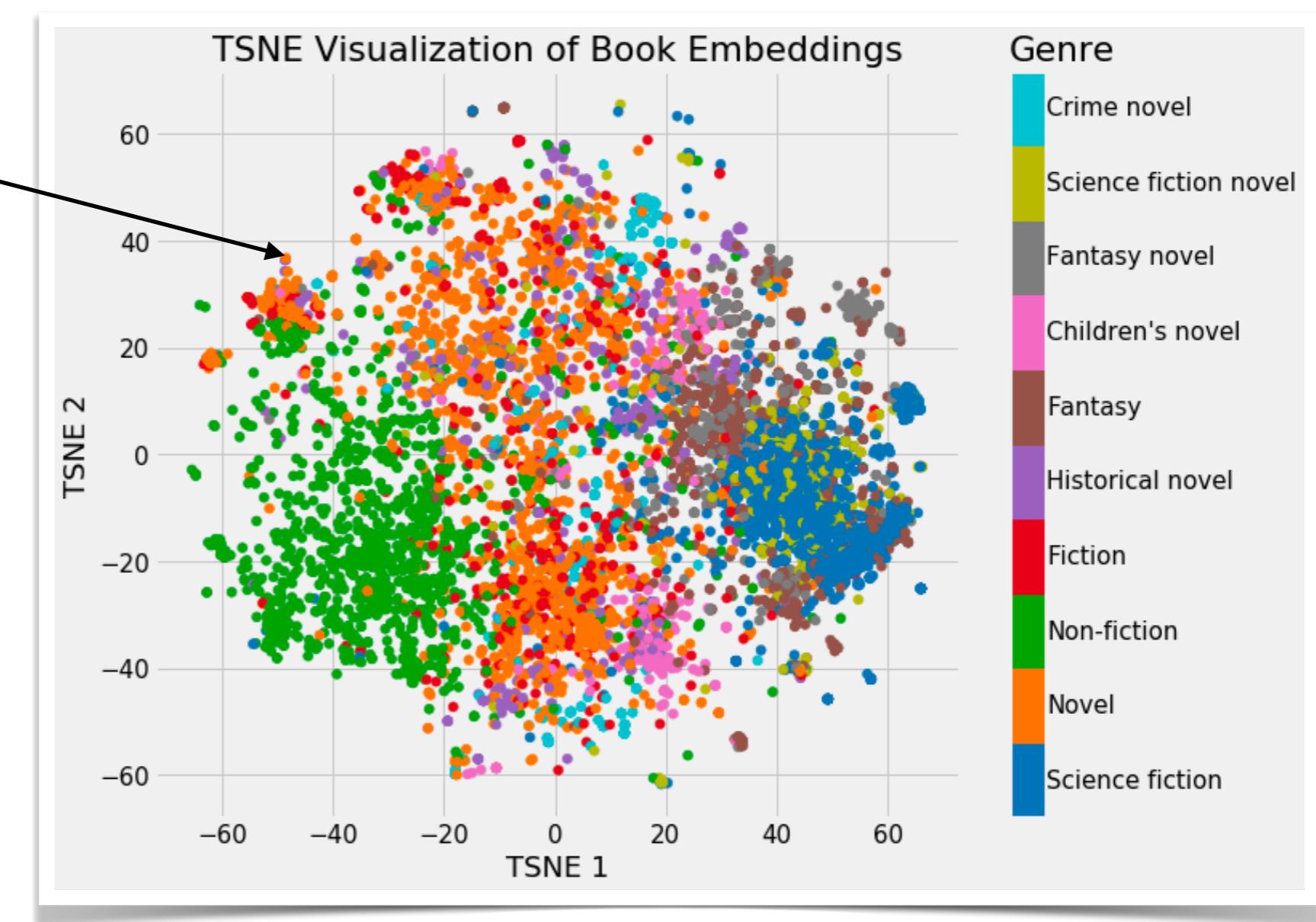
Embedding

An **embedding** is a relatively low-dimensional space into which you can translate high-dimensional vectors.

Ideally, an embedding captures some of the semantics of the input by placing **semantically similar inputs close together** in the embedding space.

```
array([-0.5968882 , -0.33086956, -0.32643065, -0.3670732 ,  0.628059 ,  
-0.3692328 , -0.37902787, -0.12308089, -0.38124698, -0.03940517,  
0.2260839 ,  0.2873845, -0.29775823, -0.99628943, -0.54497653, -0.11718027,  
-0.07114276, -0.24311952,  0.43098116,  0.09587188, -0.2503798 ,  0.06768776,  0.3311586 ,  
0.18462623, -0.45676082,  0.5662387 ,  0.69908774,  0.48064467,  
0.27378514, -0.45430255,  0.17282294, -0.40275463, -0.38083532,  
0.47487524,  0.31950948, -0.1189335 ,  0.2165357 ,  0.034114 ,  
0.05689918,  0.20939653,  0.15209009, -0.24204595,  0.03478364,  
0.1616051 , -0.5827333 , -0.47017908,  0.26226178, -0.11884775,  
0.48180743, -0.5173988 , -0.19270805,  0.660391 , -0.24518126,  
-0.42860952, -0.22274768,  0.4887834 ,  0.49302152,  0.38799986,  
-0.041193 , -0.38600504, -0.37632987,  0.04570564,  0.50462466,  
-0.14396582,  0.33490512, -0.15964787, -0.21363072, -0.25445372,  
0.52389127,  0.5747422 , -0.25075617, -0.5339069 ,  0.2582965 ,  
-0.16139959,  0.09748188,  0.04540966, -0.27768216, -0.51260656,  
-0.06189002, -0.54032195, -0.21863565,  0.06233869,  0.13287479,  
0.49741864,  0.1772418 ,  0.02864824, -0.04775626, -0.16884916,  
0.4643644 ,  0.5546319 ,  0.68051434,  0.7790246 ,  0.5617202 ],  
dtype=float32)
```

t-SNE is a tool to visualize high-dimensional data. It converts similarities between data points to joint probabilities and tries to minimize the Kullback-Leibler divergence between the joint probabilities of the low-dimensional embedding and the high-dimensional data.



Item representation: One-hot encoding

Words in documents and other categorical features such as **user/product ids** in recommenders, **names of places**, visited **URLs**, etc. are usually represented by using a one-of-K scheme (**one-hot encoding**).

If we represent every English word as an $\mathbb{R}^{|V| \times 1}$ vector with all 0's and one 1 at the index of that word in the sorted english language, word vectors in this type of encoding would appear as the following:

$$w^{aardvark} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, w^a = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, w^{at} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \dots, w^{zebra} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

There are an estimated 13 million tokens for the English language. The dimensionality of these vectors is huge!

Item representation: One-hot encoding

One hot encoding represents each word as a completely **independent** entity:

$$(w^{hotel})^T w^{motel} = (w^{hotel})^T w^{cat} = 0$$

But words are not independent (from the point of view of their meaning)!

What other alternatives are there?

Self-supervised
pre-training of word
vectors.

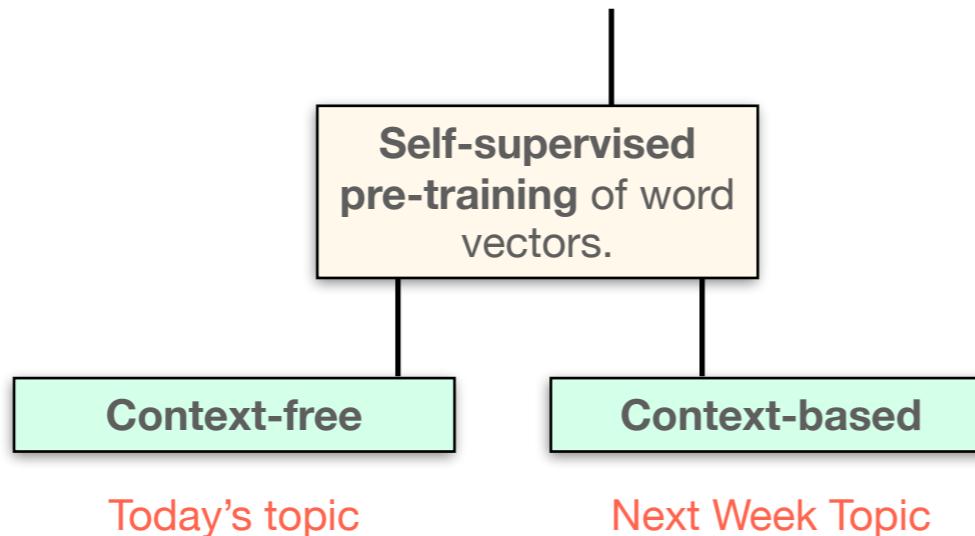
Item representation: One-hot encoding

One hot encoding represents each word as a completely **independent** entity:

$$(w^{hotel})^T w^{motel} = (w^{hotel})^T w^{cat} = 0$$

But words are not independent (from the point of view of their meaning)!

What other alternatives are there?



Embedding

Embeddings can be thought of as an alternative to one-hot encoding vectors along with dimensionality reduction.

Insight:

In order to use one-hot encoding within a machine learning system, we can represent each sparse vector as a dense vector of numbers **so that semantically similar items** (movies, words, products, etc.) **have similar representations in the new vector space**.

Problems:

- How can we learn a mapping between items and dense vectors? (mathematical question)?
- Do we need a task-dependent or a task-independent representation (empirical question)?

Embedding

Let's consider **linear embedding maps**:

$$[0,0,1]^T \cdot \begin{bmatrix} 0.29572738 & 0.88443109 & 0.21831979 \\ 0.03157878 & 0.71250614 & 0.22703532 \\ 0.12386669 & 0.74266196 & 0.91580261 \end{bmatrix} = \begin{bmatrix} 0.21831979 \\ 0.22703532 \\ 0.91580261 \end{bmatrix}$$

Item index = 3 **Embedding matrix** **Embedding vector**

You can learn a **linear embedding mapping** (that maps indices to real valued vectors) for your data in two different ways:

- By considering the elements of the embedding matrix as **parameters** and optimizing their value with respect to a loss function that represents a task (f.e. classifying the polarity of a tweet).
- By “learning” a set of elements of the embedding matrix that are “good” for a **large series of tasks**.

Keras embedding layer

The Keras embedding layer receives a sequence of **non-negative integer indices** and learns to embed those into a high dimensional vector (the size of which is specified by output dimension).

`[[4], [20]]` is turned into `[[0.25,0.1,0.3, - 0.99], [0.6, - 0.2,1.03, - 2.09]]`

This layer can only be used as the **first** layer in a model (after the input layer). It **learns a representation that is optimal for a task.**

```
● ● ●
model = Sequential()
model.add(Embedding(vocab_size, 8, input_length=max_length))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
# compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# summarize the model
print(model.summary())
```

You could implement this with a one-hot encoding input + a fully connected layer W , but this implementation is more efficient.

Keras embedding layer

Lets see how to do it using restaurant reviews data.

```
# Define 10 restaurant reviews
reviews =[
    'Never coming back!',
    'horrible service',
    'rude waitress',
    'cold food',
    'horrible food!',
    'awesome',
    'awesome services!',
    'rocks',
    'poor work',
    'couldn\'t have done better'
]
#Define labels
labels = array([1,1,1,1,1,0,0,0,0,0])

Vocab_size = 50
encoded_reviews = [one_hot(d,Vocab_size) for d in reviews]
```

encoded reviews:

```
[[18, 39, 17], [27, 27],
[5, 19], [41, 29], [27,
29], [2], [2, 1], [49],
[26, 9], [6, 9, 11, 21]]
```

Keras embedding layer

Lets see how to do it using restaurant reviews data.

```
● ● ●  
max_length = 4  
padded_reviews = pad_sequences(encoded_reviews,maxlen=max_length,padding='post')  
print(padded_reviews)
```

encoded reviews:

```
[ [18 39 17 0]  
[27 27 0 0]  
[ 5 19 0 0]  
[41 29 0 0]  
[27 29 0 0]  
[ 2 0 0 0]  
[ 2 1 0 0]  
[49 0 0 0]  
[26 9 0 0]  
[ 6 9 11 21]]
```

Keras embedding layer

Lets see how to do it using restaurant reviews data.

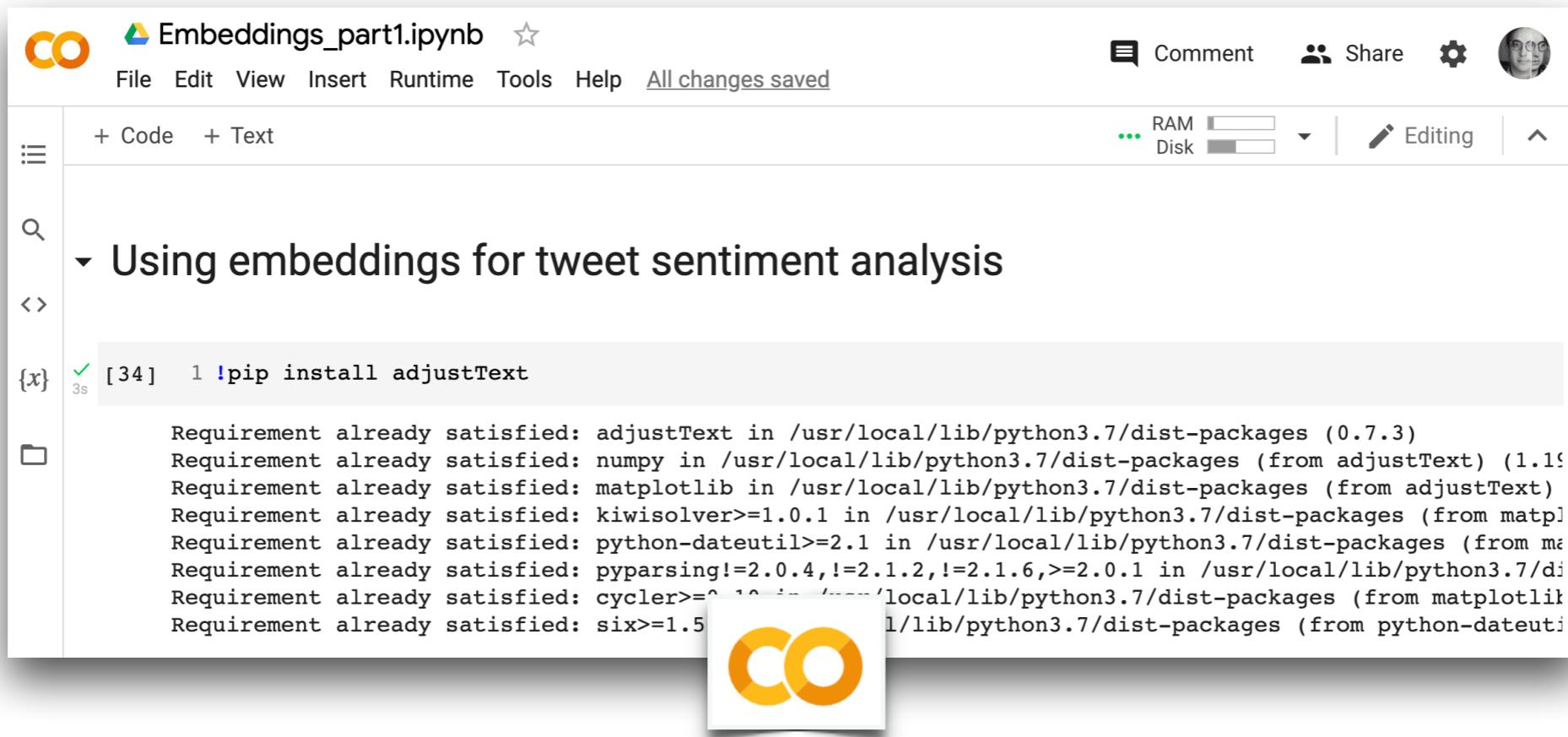


```
model = Sequential()
embedding_layer = Embedding(input_dim=Vocab_size, output_dim=8, input_length=max_length)
model.add(embedding_layer)
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=[ 'acc' ])
```

The embedding matrix is essentially a lookup table of 50 rows and 8 columns.

```
[0] -> [ 0.056933 0.0951985 0.07193055 0.13863552 -0.13165753 0.07380469 0.10305451 -0.10652688]
```

Embeddings and Sentiment Analysis



The screenshot shows a Jupyter Notebook interface with the following details:

- Title Bar:** CO Embeddings_part1.ipynb
- Toolbar:** File Edit View Insert Runtime Tools Help All changes saved
- Header Buttons:** Comment, Share, Settings, User Profile
- Code Cell:** {x} [34] 1 !pip install adjustText
Requirement already satisfied: adjustText in /usr/local/lib/python3.7/dist-packages (0.7.3)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from adjustText) (1.19.2)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-packages (from adjustText)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib)
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: cycler>=0.10.0 in /usr/local/lib/python3.7/dist-packages (from matplotlib)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packages (from python-dateutil)

Words, NLP and meaning

Understanding **the way words fit together with structure and meaning** is a field of study connected to linguistics.

In linguistics, **ambiguity** is at the sentence rather than word level. Words with multiple meanings combine to make ambiguous sentences and phrases become increasingly difficult to understand.

This compelling field faces **unsolved problems**: **ambiguity, polysemy, homonymy, synonomy, coreference, anaphora**, etc.

Words with multiple meanings are considered **polysemous** or homonymous.

The verb ‘get’, a polysemous word, for example, could mean ‘to procure’, ‘to acquire’, or ‘to understand’.

Homonyms are the other main type of word with multiple meanings. For example, “rose,” which is a homonym, could mean to “rise up” or it could be a flower. These two-word meanings are not related at all.

Synonymous words mean the same as each other (or very similar), but are different words. An example of synonymous words would be the adjectives “tiny,” “little” and “mini” as synonyms of “small.”

Anaphora resolution is the problem of trying to tie mentions of items as pronouns or noun phrases from earlier in a piece of text (such as people, places, things). “John helped Mary. He was kind.”

Distributional and compositional semantics

Linguistics assumes two important hypotheses:

1. **Distributional Hypothesis:** words that occur in the same contexts tend to have similar meanings (Harris, 1954).

government debt problems turning into **banking crises** as has happened in

These words will represent banking

These words will represent banking

saying that Europe needs unified **banking regulation** to replace the hodgepodge

The meaning of a word is determined by its contexts

2. **Compositionality:** Semantic complex entities can be built from its simpler constituents by using **compositional** rules

morphemes > words > sentences > paragraphs > text

Word models.

Researchers have developed various techniques for **training general purpose word models (vector models)** using the enormous piles of unannotated text on the web (this is known as **pre-training**).

Assumption: These general purpose pre-trained models are general and can then be **fine-tuned** on smaller task-specific datasets, e.g., when working with problems like question answering and sentiment analysis or even be used as it.

Word models.

Word models can either be **context-free** or **context-based**.

Context-free models learn **a single** word embedding representation (a **vector** of numbers) for each word in the vocabulary.

On the other hand, **context-based** models generate (at training or at inference time) a representation of each word that **is based on the other words in the sentence**.

Context-based representations can then be unidirectional or bidirectional. For example, in the sentence “I accessed the bank account,” a unidirectional contextual model would represent “bank” based on “I accessed the” but not “account.”

Context-free Word Embeddings

Simply put, context-free **word embeddings** are a way to represent, in a mathematical space, words which “live” in similar contexts in a real-world collection of text, otherwise known as a **text corpus**.

Word embeddings take the notion of distributional similarity, with words simply mapped to their company and stored in vector spaces.

These vectors can then be used across a **wide range of natural language understanding tasks**.

F.e. Analyzing word co-occurrences by using SVD

Some previous methods were based on **counting** co-occurrences of words, but today the most successful are those based on **prediction**:

$y = f(x)$ Instead of counting how often each word y occurs near x ,
 train a classifier on a binary prediction task: Is y likely to
 show up near x ?

x

government debt problems turning into **banking** crises as has happened in

y

y

Context-free Word Embeddings

Simply put, context-free **word embeddings** are a way to represent, in a mathematical space, words which “live” in similar contexts in a real-world collection of text, otherwise known as a **text corpus**.

Word embeddings take the notion of distributional similarity, with words simply mapped to their company and stored in vector spaces.

These vectors can then be used across a **wide range of natural language understanding tasks**.

F.e. Analyzing word co-occurrences by using SVD

Some previous methods were based on **counting** co-occurrences of words, but today the most successful are those based on **prediction**:

$y = f(x)$ Instead of counting how often each word y occurs near x ,
 train a classifier on a binary prediction task: Is y likely to
 show up near x ?

y

government debt problems turning into **banking** crises as has happened in

x

x

Word2Vec

The **context of a word** is the set of m surrounding words.

For instance, the $m = 2$ context of the word ‘fox’ in the sentence

‘The quick brown fox jumped over the lazy dog’

is [‘quick’, ‘brown’, ‘jumped’, ‘over’].

The idea is to design a model whose parameters are the word vectors.

Then, train the (**discriminative**) model on a certain objective.

Mikolov presented a simple, probabilistic model in 2013 that is known as **word2vec**. In fact, **word2vec** includes 2 algorithms (**CBOW** and **skip-gram**).

Continuous Bag of Words Model (**CBOW**)

The approach is to treat {`The`, `cat`, `over`, `the`, `puddle`} as a context and from these words, be able to predict or generate the center word `jumped`.

First, we set up our known parameters. Let the known parameters in our model be the sentence represented by **one-hot word vectors**.

- The **input** one-hot vectors or context will be represented with an $x^{(c)}$.
- And the **output** as y which is the one hot vector of the unknown center word.

Continuous Bag of Words Model (CBOW)

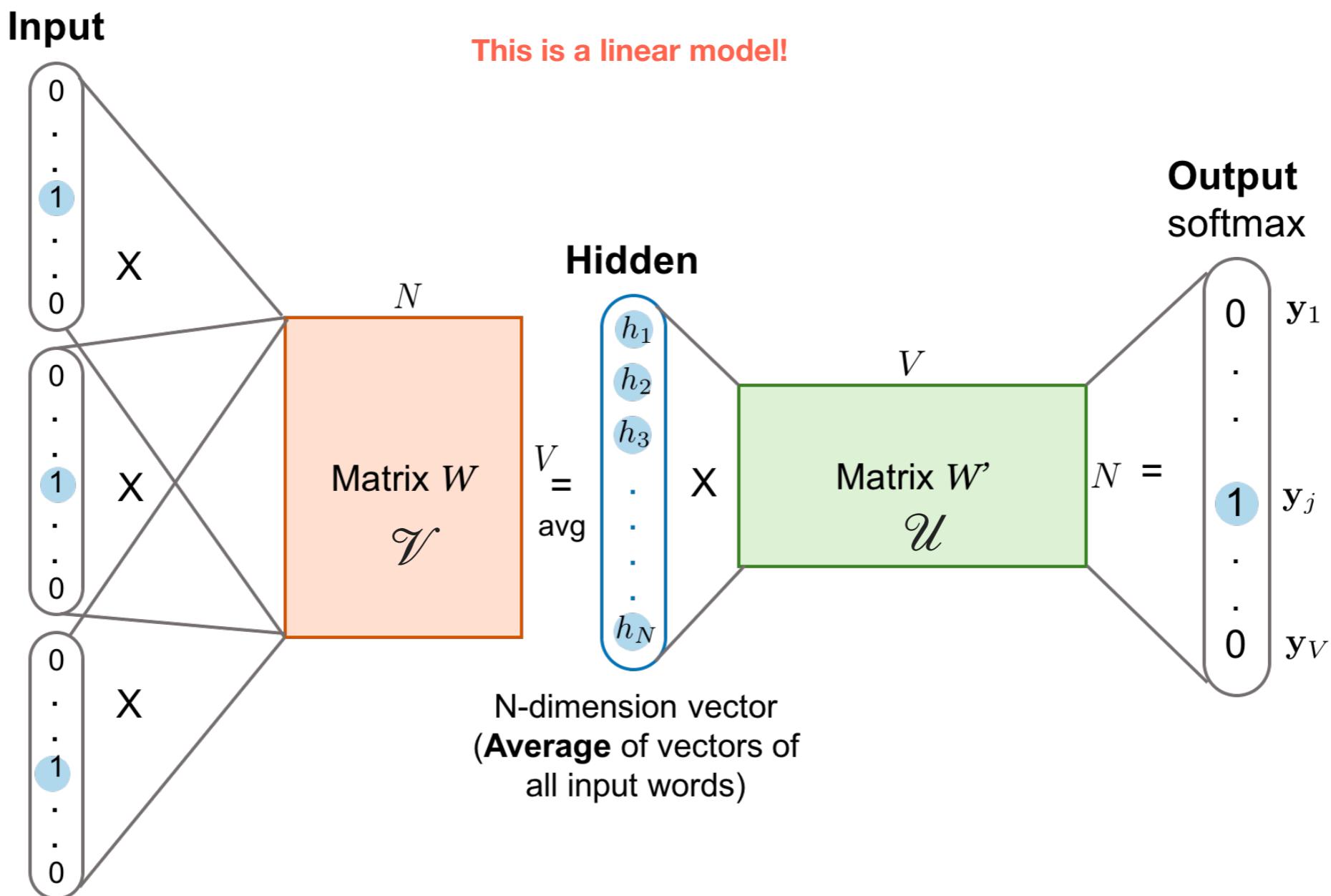
We create two matrices, $\mathcal{V} \in \mathbb{R}^{n \times |V|}$ and $\mathcal{U} \in \mathbb{R}^{|V| \times n}$, where n is an arbitrary size which defines the size of our embedding space.

\mathcal{V} is the input word matrix such that the i -th column of \mathcal{V} is the n -dimensional embedded vector for word w_i when it is an input to this model. We denote this $n \times 1$ vector as v_i .

Similarly, \mathcal{U} is the output word matrix. The j -th row of \mathcal{U} is an n -dimensional embedded vector for word w_j when it is an output of the model. We denote this row of \mathcal{U} as u_j .

Note that we do in fact learn two vectors for every word w_i (i.e. input word vector v_i and output word vector u_i).

Continuous Bag of Words Model (CBOW)



Continuous Bag of Words Model (CBOW)

We breakdown the way this model works in these steps:

- We generate our one hot word vectors for the input context of size m : $(x^{(c-m)}, \dots, x^{(c-1)}, x^{(c+1)}, \dots, x^{(c+m)} \in \mathbb{R}^{|V|})$.
- We get our embedded word vectors for the context ($v_{c-m} = \mathcal{V}x^{(c-m)}, v_{c-m+1} = \mathcal{V}x^{(c-m+1)}, \dots, v_{c+m} = \mathcal{V}x^{(c+m)} \in \mathbb{R}^n$)

$$\begin{array}{ccccc}
 & V & & x & \\
 \begin{matrix} 17 & 23 & 4 & 10 & 11 \\ 24 & 5 & 6 & 12 & 18 \\ 1 & 7 & 13 & 19 & 25 \end{matrix} & \times & \begin{matrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{matrix} & = & \boxed{10 \quad 12 \quad 19}
 \end{array}$$

- Average these vectors to get $h = \frac{v_{c-m} + v_{c-m+1} + \dots + v_{c+m}}{2m} \in \mathbb{R}^n$
- Generate a score vector $z = \mathcal{U}h \in \mathbb{R}^{|V|}$. As the dot product of similar vectors is higher, it will push similar words close to each other in order to achieve a high score.
- Turn the scores into probabilities $\hat{y} = \text{softmax}(z) \in \mathbb{R}^{|V|}$.
- We desire our probabilities generated, $\hat{y} \in \mathbb{R}^{|V|}$, to match the true probabilities, $y \in \mathbb{R}^{|V|}$, which also happens to be the one hot vector of the actual word.

Continuous Bag of Words Model (CBOW)

How can we learn these two matrices?

Well, we need to create an **objective function**. We choose a classification loss function. Here, we use a popular choice: cross entropy, **as a proxy function of the conditional probability** $P(u_c | h)$.

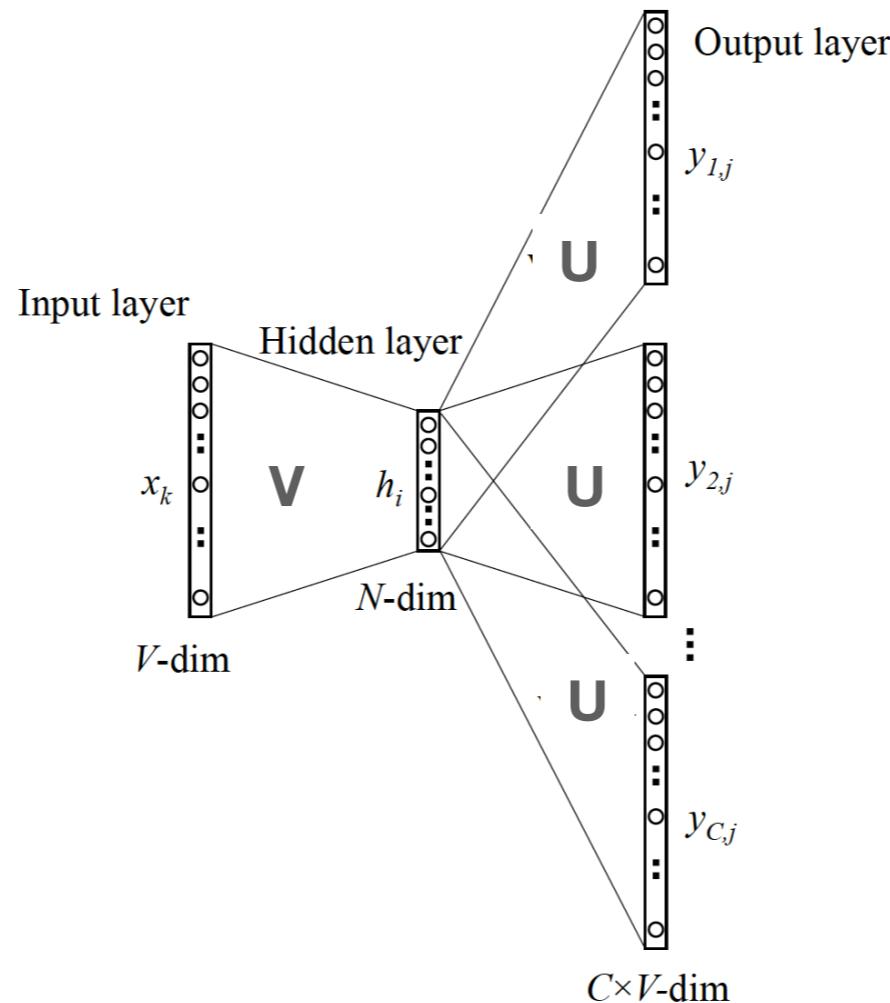
We thus formulate our optimization objective as:

$$\begin{aligned} \text{minimize } J &= -\log P(w_c | w_{c-m}, \dots, w_{c-1}, w_{c+1}, \dots, w_{c+m}) \\ &= -\log P(u_c | h) \\ &= -\log \frac{\exp(u_c^T h)}{\sum_{j=1}^{|V|} \exp(u_j^T h)} \\ &= -u_c^T h + \log \sum_{j=1}^{|V|} \exp(u_j^T h) \end{aligned}$$

Skip-gram model

Another approach, the **skip-gram model**, is to create a model such that given the center word ‘jumped’, the model will be able to predict or generate the surrounding words ‘The’, ‘cat’, ‘over’, ‘the’, ‘puddle’.

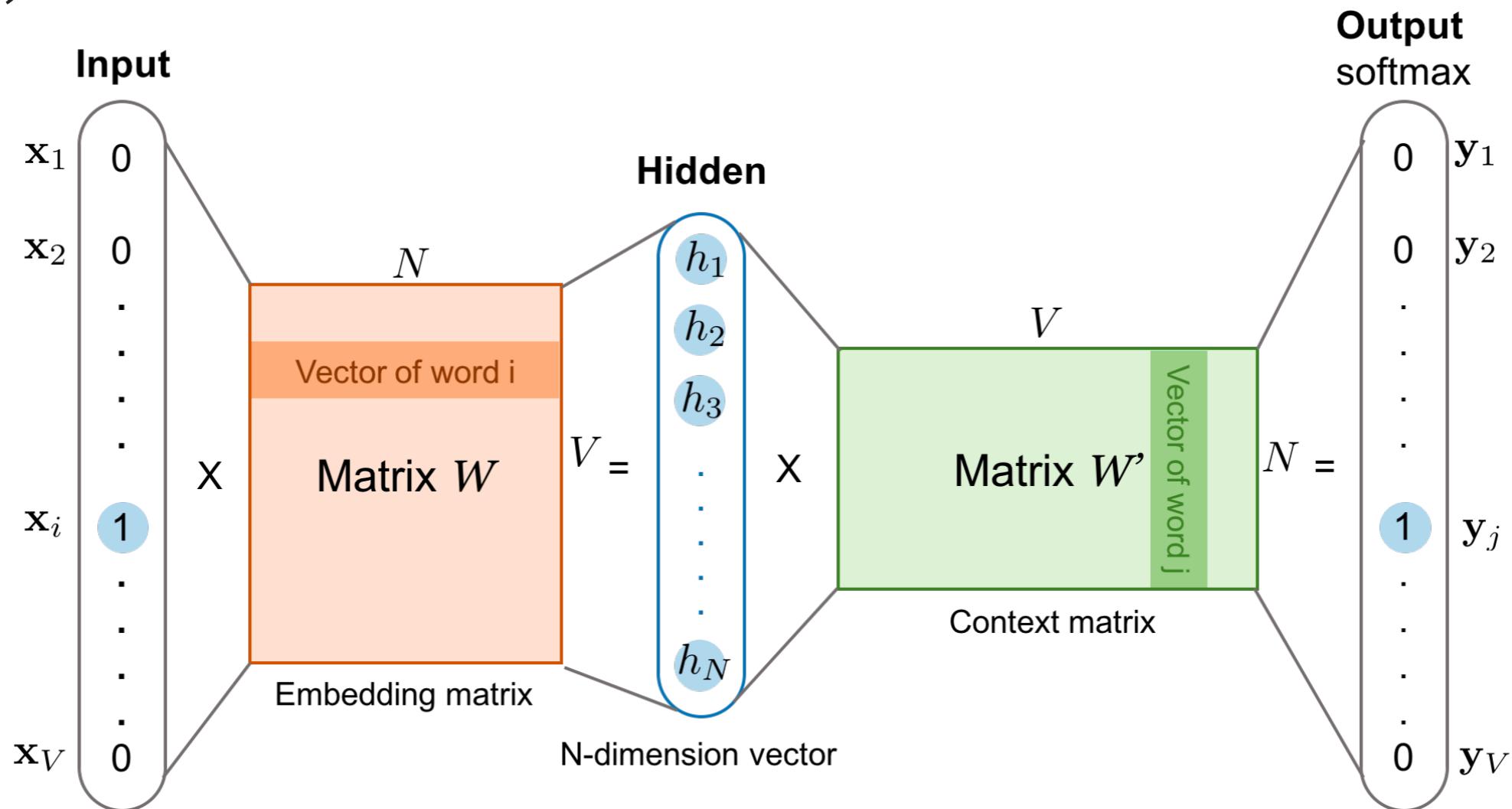
Here we call the word ‘jumped’ the context. We call this type of model a **Skip-Gram model**.



Skip-gram model

In fact, we can consider that **each context-target pair** is treated as a new observation in the data.

For example, the target word “swing” in the above case produces four training samples: (“swing”, “sentence”), (“swing”, “should”), (“swing”, “the”), and (“swing”, “sword”).



Skip-gram model

We thus formulate our optimization objective as:

$$\begin{aligned} \text{minimize } J &= -\log P(w_{c-m}, \dots, w_{c-1}, w_{c+1}, \dots, w_{c+m} | w_c) \\ &= -\log \prod_{j=0; j \neq m}^{2m} \frac{\exp(u_{c-m+j}^T v_c)}{\sum_{k=1}^{|V|} \exp(u_k^T v_c)} \\ &= -\log \prod_{j=0; j \neq m}^{2m} u_{c-m+j}^T v_c + 2m \log \sum_{k=1}^{|V|} \exp(u_k^T v_c) \end{aligned}$$

Negative Sampling

The summation over $|V|$ is computationally huge!

Any update we do or **evaluation of the objective function** would take $O(|V|)$ time which if we recall is in the millions.

A simple idea is we could instead just **approximate** it.

For every training step, instead of looping over the entire vocabulary, we can just sample several negative examples!

But instead of this, we can also optimize a different objective function (**Negative Sampling Loss**) that takes into account this fact...

(See <https://lilianweng.github.io/lil-log/2017/10/15/learning-word-embedding.html#loss-functions>)

Example

Let's try a little experiment in word embedding extracted from "the Games of Thrones corpus".

Extract words:

```
import sys
from nltk.corpus import stopwords
from nltk.tokenize import sent_tokenize

STOP_WORDS = set(stopwords.words('english'))

def get_words(txt):
    return filter(
        lambda x: x not in STOP_WORDS,
        re.findall(r'\b(\w+)\b', txt)
    )

def parse_sentence_words(input_file_names):
    """Returns a list of a list of words. Each sublist is a sentence."""
    sentence_words = []
    for file_name in input_file_names:
        for line in open(file_name):
            line = line.strip().lower()
            line = line.decode('unicode_escape').encode('ascii','ignore')
            sent_words = map(get_words, sent_tokenize(line))
            sent_words = filter(lambda sw: len(sw) > 1, sent_words)
            if len(sent_words) > 1:
                sentence_words += sent_words
    return sentence_words

# You would see five .txt files after unzip 'a_song_of_ice_and_fire.zip'
input_file_names = ["001ssb.txt", "002ssb.txt", "003ssb.txt",
                    "004ssb.txt", "005ssb.txt"]
GOT_SENTENCE_WORDS= parse_sentence_words(input_file_names)
```

Example

Let's try a little experiment in word embedding extracted from "the Games of Thrones corpus".

Feed a word2vec model:

```
● ● ●

from gensim.models import Word2Vec

# size: the dimensionality of the embedding vectors.
# window: the maximum distance between the current and predicted word within a sentence.
model = Word2Vec(GOT_SENTENCE_WORDS, size=128, window=3, min_count=5, workers=4)
model.wv.save_word2vec_format("got_word2vec.txt", binary=False)
```

Example

Let's try a little experiment in word embedding extracted from "the Games of Thrones corpus".

Check the results:

<code>model.most_similar('king', topn=10)</code> (word, similarity with 'king')	<code>model.most_similar('queen', topn=10)</code> (word, similarity with 'queen')
('kings', 0.897245)	('cersei', 0.942618)
('baratheon', 0.809675)	('joffrey', 0.933756)
('son', 0.763614)	('margaery', 0.931099)
('robert', 0.708522)	('sister', 0.928902)
('lords', 0.698684)	('prince', 0.927364)
('joffrey', 0.696455)	('uncle', 0.922507)
('prince', 0.695699)	('varys', 0.918421)
('brother', 0.685239)	('ned', 0.917492)
('aerys', 0.684527)	('melisandre', 0.915403)
('stannis', 0.682932)	('robb', 0.915272)

Limitations

- **Low probability words** are represented by one symbol [UNK]. How to deal with rare words, names, etc?
- Always the same representation for a word type regardless of the context in which a word token occurs. We do not have fine-grained **word sense disambiguation**.
- We just have one representation for a word, but words have **different aspects**, including semantics, syntactic behavior, and register/connotations.

Example

We can load a pre-trained word embeddings matrix into an **Embedding** layer!

The screenshot shows the Keras documentation website. The left sidebar has a navigation menu with links like 'About Keras', 'Getting started', 'Developer guides', 'Keras API reference', 'Code examples' (which is highlighted in black), 'Computer Vision', 'Natural Language Processing' (which is highlighted in red), 'Structured Data', 'Timeseries', 'Audio Data', and 'Generative Deep Learning'. The main content area has a search bar at the top. Below it, a breadcrumb trail shows '» Code examples / Natural Language Processing / Using pre-trained word embeddings'. The title 'Using pre-trained word embeddings' is displayed in large bold text. Below the title, author information ('Author: fchollet'), creation date ('Date created: 2020/05/05'), and last modification date ('Last modified: 2020/05/05') are listed. A description follows: 'Text classification on the Newsgroup20 dataset using pre-trained GloVe word embeddings.' Below this, there are two links: 'View in Colab' and 'GitHub source'. To the right of the main content, a sidebar titled 'Using pre-trained word embeddings' lists several sub-sections with right-pointing arrows: 'Setup', 'Introduction', 'Download the Newsgroup20 data', 'Let's take a look at the data', 'Shuffle and split the data into training & validation sets', 'Create a vocabulary index', 'Load pre-trained word embeddings', 'Build the model', 'Train the model', and 'Export an end-to-end model'. A code block at the bottom contains Python code for importing numpy, tensorflow, and keras.

Search Keras documentation...

» Code examples / Natural Language Processing / Using pre-trained word embeddings

Using pre-trained word embeddings

Author: fchollet
Date created: 2020/05/05
Last modified: 2020/05/05
Description: Text classification on the Newsgroup20 dataset using pre-trained GloVe word embeddings.

[View in Colab](#) · [GitHub source](#)

Setup

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
```

Using pre-trained word embeddings

- ▷ Setup
- ▷ Introduction
- ▷ Download the Newsgroup20 data
- ▷ Let's take a look at the data
- ▷ Shuffle and split the data into training & validation sets
- ▷ Create a vocabulary index
- ▷ Load pre-trained word embeddings
- ▷ Build the model
- ▷ Train the model
- ▷ Export an end-to-end model

Example

spaCy ★ Out now: spaCy v3.1 USAGE

GET STARTED

- [Installation](#)
- [Models & Languages](#)
- [Facts & Figures](#)
- [spaCy 101](#)
- [New in v3.0](#)
- [New in v3.1](#)

GUIDES

- [Linguistic Features](#)
- [Rule-based Matching](#)
- [Processing Pipelines](#)
- Embeddings & Transformers NEW**
 - Embedding Layers
 - Transformers
 - Static Vectors
 - Pretraining
- [Training Models NEW](#)
- [Layers & Model](#)
- [Architectures NEW](#)
- [spaCy Projects NEW](#)
- [Saving & Loading](#)
- [Visualizers](#)

Embeddings, Transformers and Transfer Learning

Using transformer embeddings like BERT in spaCy

spaCy supports a number of **transfer and multi-task learning** workflows that can often help improve your pipeline's efficiency or accuracy. Transfer learning refers to techniques such as word vector tables and language model pretraining. These techniques can be used to import knowledge from raw text into your pipeline, so that your models are able to generalize better from your annotated examples.

You can convert **word vectors** from popular tools like [FastText](#) and [Gensim](#), or you can load in any pretrained **transformer model** if you install [spacy-transformers](#). You can also do your own language model pretraining via the `spacy pretrain` command. You can even **share** your transformer or other contextual embedding model across multiple components, which can make long pipelines several times more efficient. To use transfer learning, you'll need at least a few annotated examples for what you're trying to predict. Otherwise, you could try using a "one-shot learning" approach using [vectors and similarity](#).

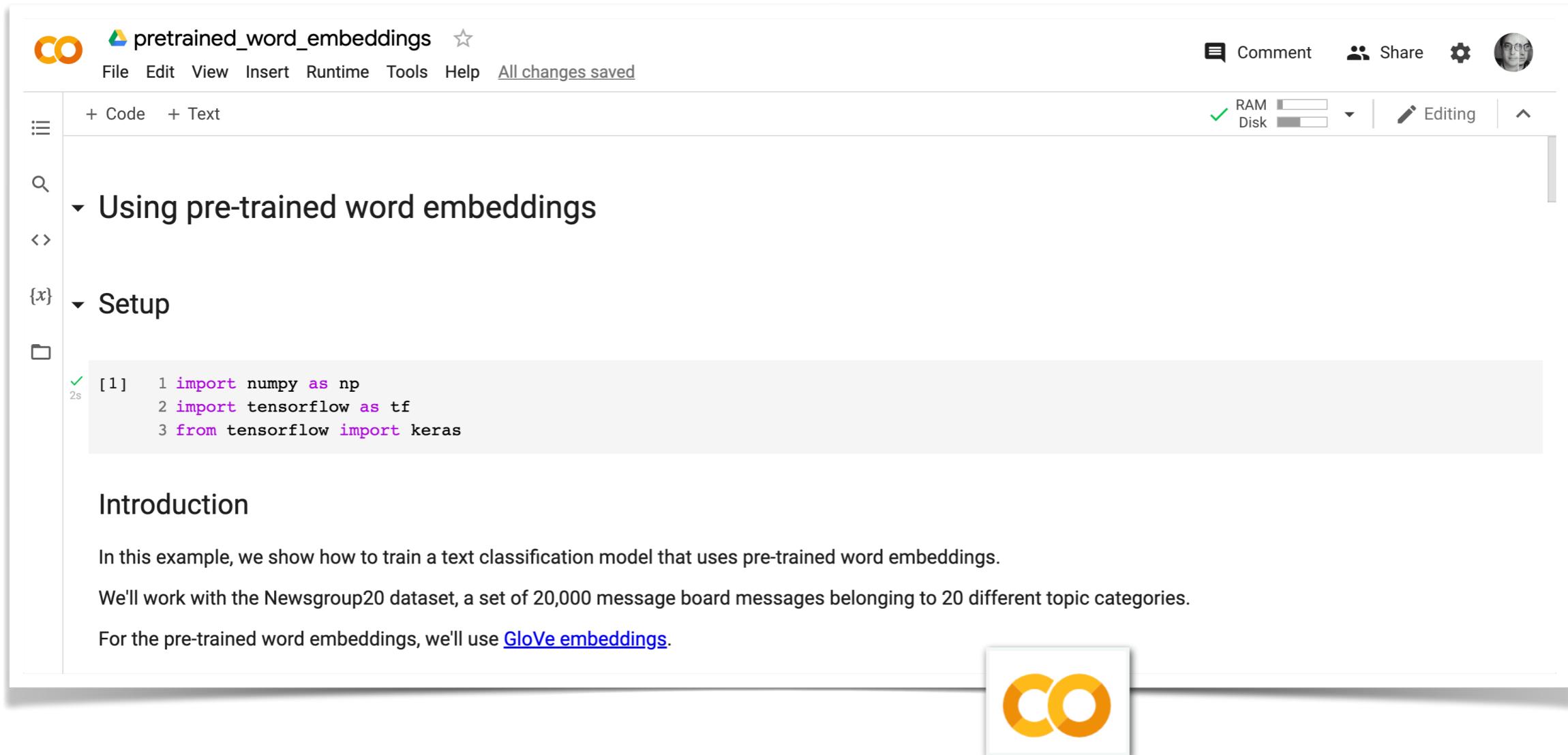
What's the difference between word vectors and language models?

+

When should I add word vectors to my model?

+

How to use pre-trained (GloVe) word embeddings?



The screenshot shows a Jupyter Notebook interface with the following details:

- Title Bar:** CO pretrained_word_embeddings
- Toolbar:** File Edit View Insert Runtime Tools Help All changes saved
- Header Buttons:** Comment, Share, Settings, User Profile
- Code Cell:** RAM: 100%, Disk: 100% | Editing
- Sidebar:** + Code, + Text
- Table of Contents:**
 - Using pre-trained word embeddings
 - {x} - Setup
- Code Block:** [1] 1 import numpy as np
2 import tensorflow as tf
3 from tensorflow import keras
- Section Header:** Introduction
- Text:** In this example, we show how to train a text classification model that uses pre-trained word embeddings.
- Text:** We'll work with the Newsgroup20 dataset, a set of 20,000 message board messages belonging to 20 different topic categories.
- Text:** For the pre-trained word embeddings, we'll use [GloVe embeddings](#).

How to learn word embeddings?

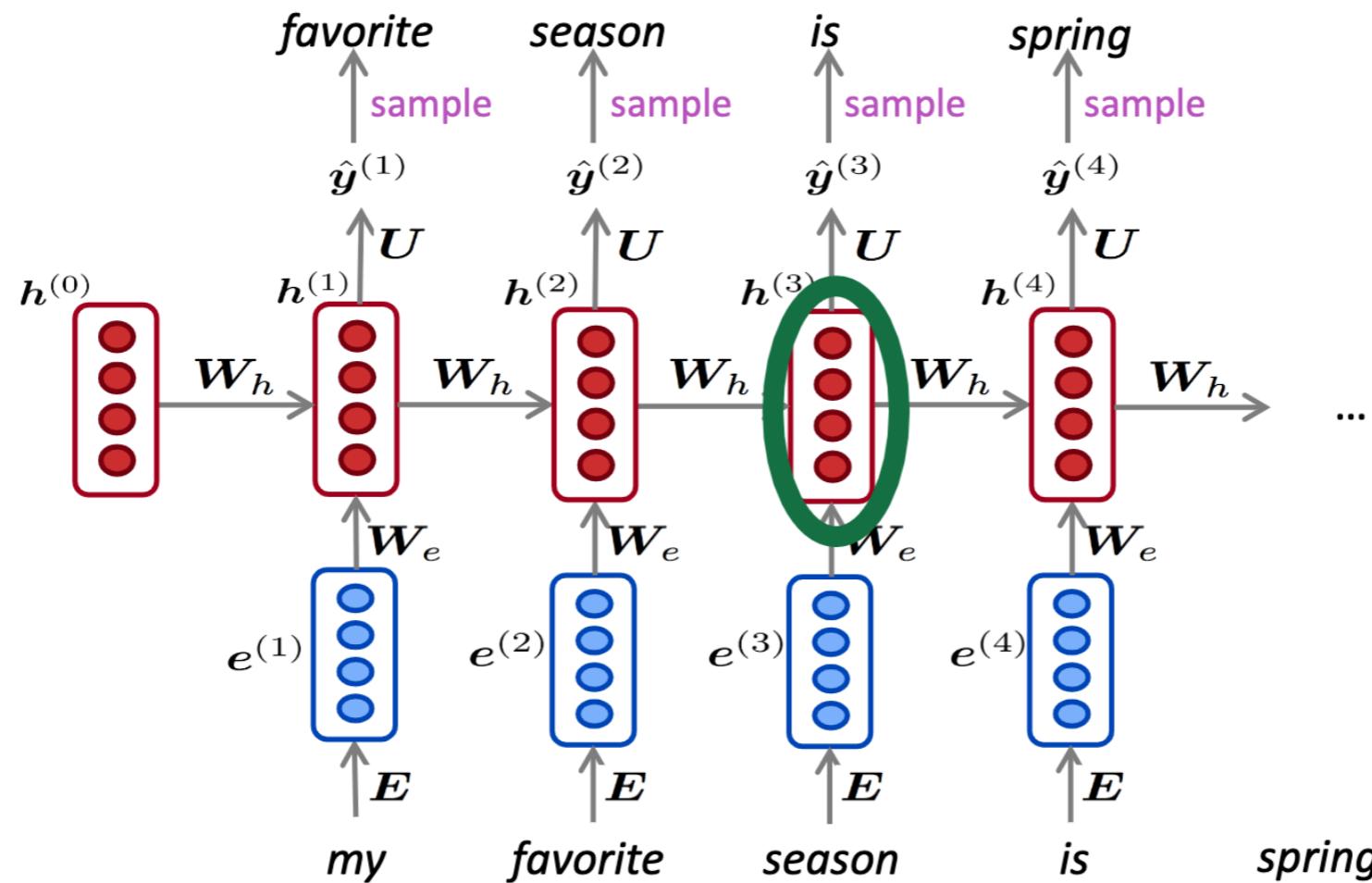
The screenshot shows a Jupyter Notebook interface with the following details:

- Title:** Embeddings_part2.ipynb
- Toolbar:** File, Edit, View, Insert, Runtime, Tools, Help, All changes saved
- Header Buttons:** Comment, Share, Settings, User Profile
- Code Cell:** + Code, + Text, Reconnect, Editing
- Sidebar Icons:** Filter, Search, Compare, {x}, File
- Section:** Product Embedding with Word2Vect
- Output:** A command-line terminal window showing the output of a pip install command:

```
1 !pip install adjustText
Requirement already satisfied: adjustText in /usr/local/lib/python3.7/dist-packages (0.7.3)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from adjustText)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-packages (from adjustText)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/local/lib/python3.7/dist-packages (from adjustText)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-packages (from matplotlib)
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packages (from python-dateutil)
```

From context-free to context-based representations

Recurrent models can generate context-dependent word representations when trained in a self-supervised way!



Limitation:

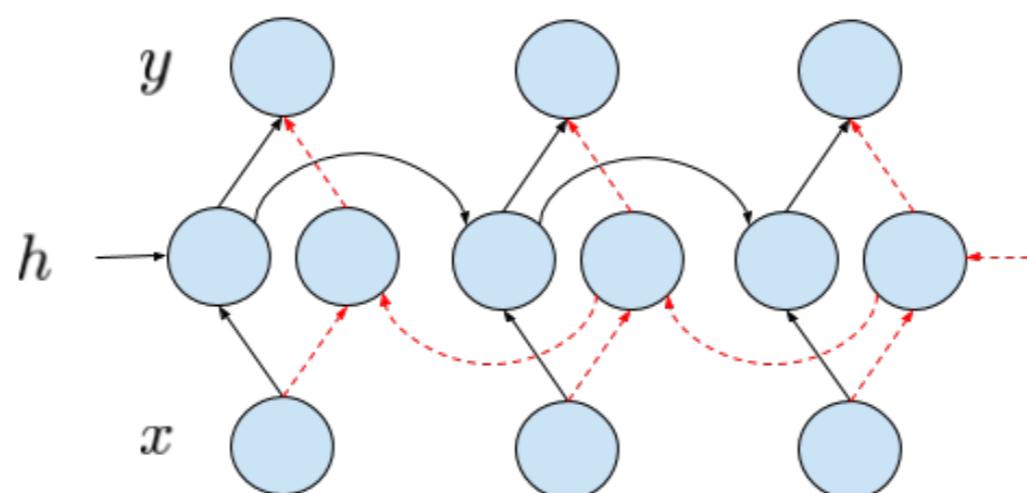
- This is a right-to-left model.

Bi-directional LSTM

Bidirectional LSTMs are an extension of traditional LSTMs that can improve model performance on sequence classification problems by extending left-to-right processing.

Bi-directional deep neural networks, at each time-step, t , maintain two hidden layers, one for the left-to-right propagation and another for the right-to-left propagation (hence, consuming twice as much memory space).

The final classification result, \hat{y} , is generated through combining the score results produced by both RNN hidden layers.



Bi-directional LSTM

The equations are (arrows are for designing left-to-right and right-to-left tensors):

$$\vec{h}_t = f(\vec{W}x_t + \vec{V}\vec{h}_{t-1} + \vec{b})$$

$$\overleftarrow{h}_t = f(\overleftarrow{W}x_t + \overleftarrow{V}\overleftarrow{h}_{t+1} + \overleftarrow{b})$$

$$\hat{y}_t = g(Uh_t + c) = g(U[\vec{h}_t; \overleftarrow{h}_t] + c)$$

$[\vec{h}_t; \overleftarrow{h}_t]$ summarizes the past and future of a single element of the sequence.

Bidirectional RNNs can be stacked as usual!

Bi-directional LSTM

```
1 # Input for variable-length sequences of integers
2 inputs = keras.Input(shape=(None,), dtype="int32")
3 # Embed each integer in a 128-dimensional vector
4 x = layers.Embedding(max_features, 128)(inputs)
5 # Add 2 bidirectional LSTMs
6 x = layers.Bidirectional(layers.LSTM(64, return_sequences=True))(x)
7 x = layers.Bidirectional(layers.LSTM(64))(x)
8 # Add a classifier
9 outputs = layers.Dense(1, activation="sigmoid")(x)
10 model = keras.Model(inputs, outputs)
11 model.summary()
12
```

Model: "functional_1"

Layer (type)	Output Shape	Param #
<hr/>		
input_1 (InputLayer)	[(None, None)]	0
embedding (Embedding)	(None, None, 128)	2560000
bidirectional (Bidirectional (None, None, 128))		98816
bidirectional_1 (Bidirection (None, 128))		98816
dense (Dense)	(None, 1)	129
<hr/>		
Total params: 2,757,761		
Trainable params: 2,757,761		
Non-trainable params: 0		



Bi-directional LSTM

```
1 # Input for variable-length sequences of integers
2 inputs = keras.Input(shape=(None,), dtype="int32")
3 # Embed each integer in a 128-dimensional vector
4 x = layers.Embedding(max_features, 128)(inputs)
5 # Add 2 bidirectional LSTMs
6 x = layers.Bidirectional(layers.LSTM(64, return_sequences=True))(x)
7 x = layers.Bidirectional(layers.LSTM(64))(x)
8 # Add a classifier
9 outputs = layers.Dense(1, activation="sigmoid")(x)
10 model = keras.Model(inputs, outputs)
11 model.summary()
12
```

Model: "functional_1"

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, None)]	0
embedding (Embedding)	(None, None, 128)	2560000
bidirectional (Bidirectional (None, None, 128))		98816
bidirectional_1 (Bidirection (None, 128))		98816
dense (Dense)	(None, 1)	129
<hr/>		
Total params: 2,757,761		
Trainable params: 2,757,761		
Non-trainable params: 0		

Here we can
use *Word2Vec*
embeddings
instead of
learning task-
specific vectors.

This could be a
good word
representation
...