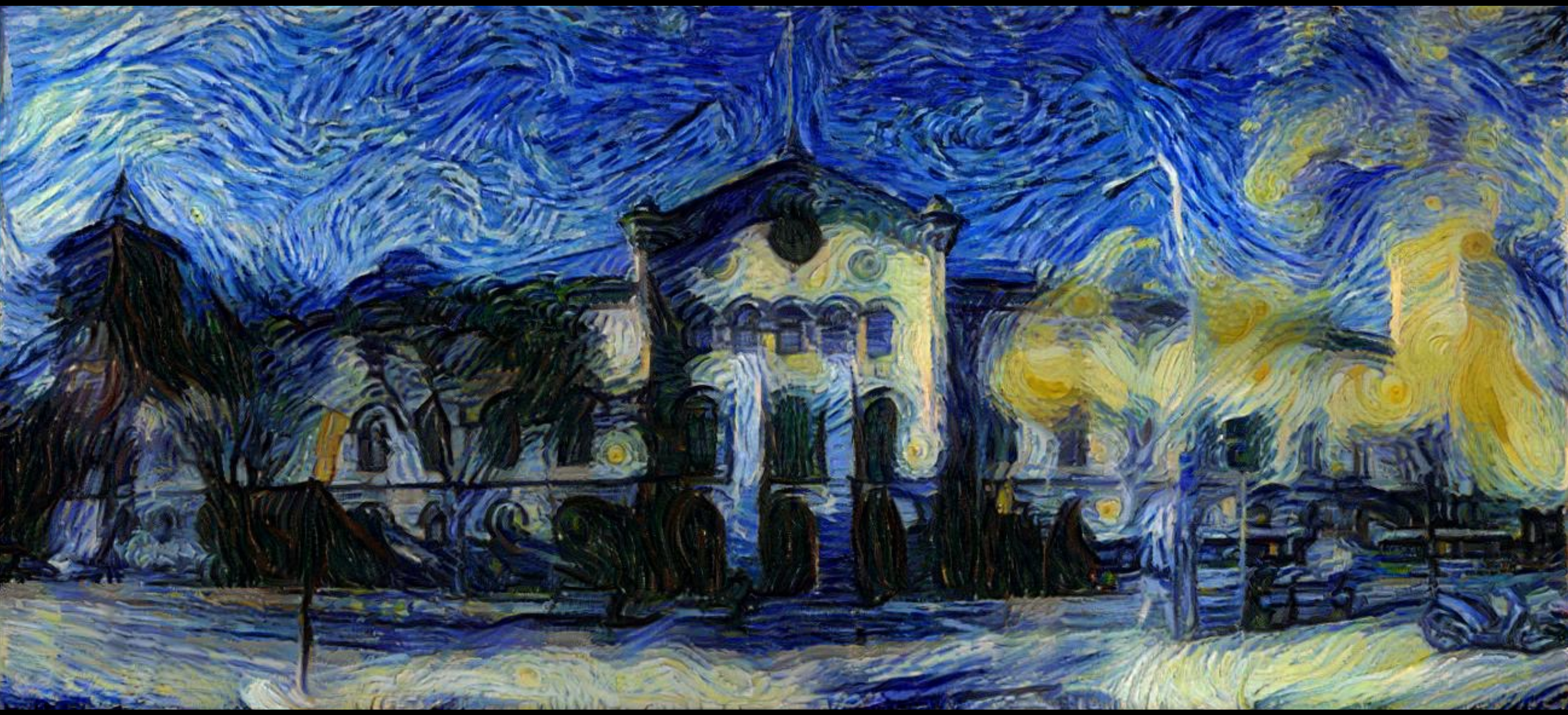
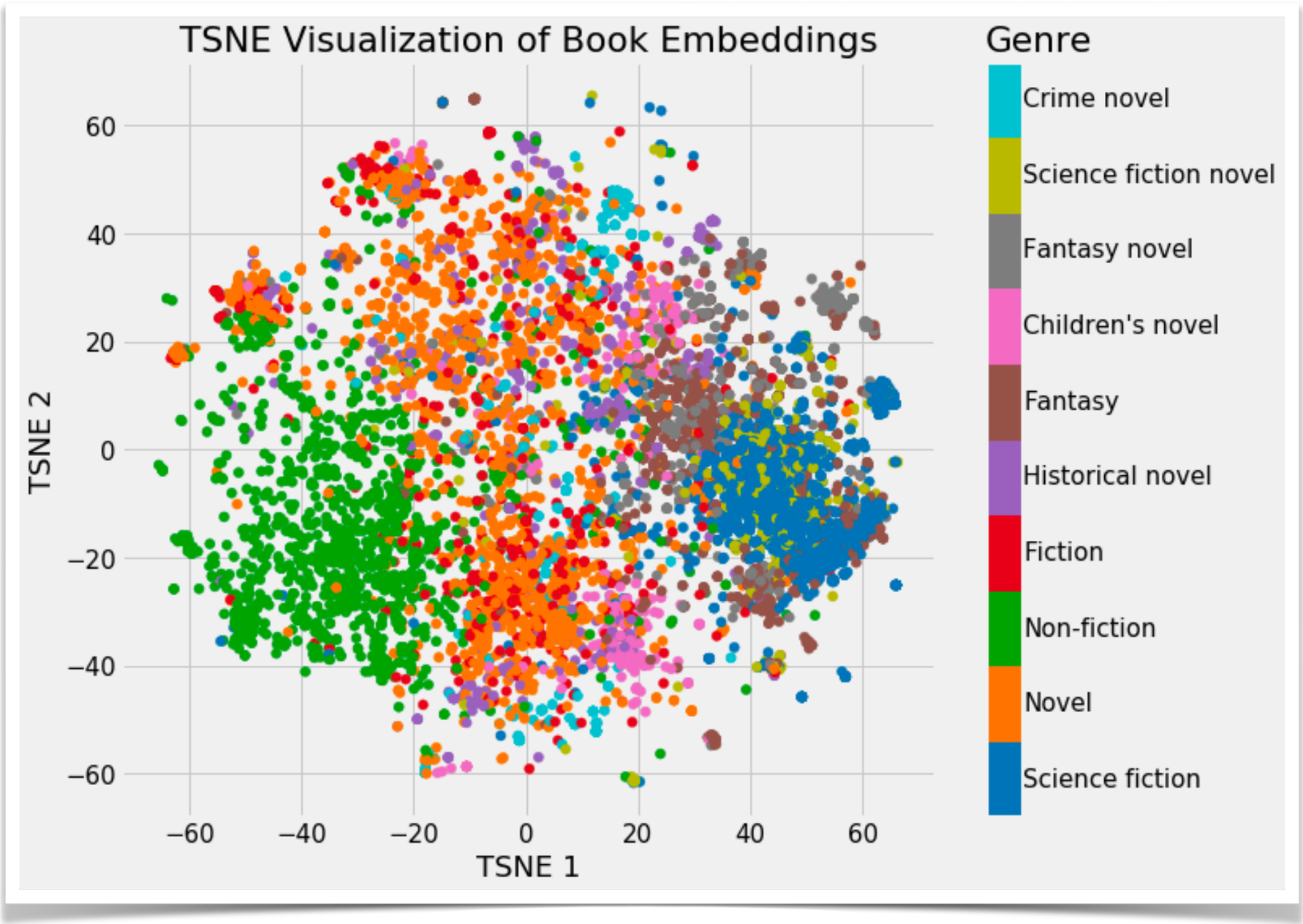


Deep Learning

Deep representations of words|items

Embedding


TSNE Visualization of Book Embeddings
Genre
Crime novel
Science fiction novel
Fantasy novel
Children's novel
Fantasy
Historical novel
Fiction
Non-fiction
Novel
Science fiction
TSNE 1
TSNE 2
60
40
20
0
−20
−40
−60
−60
−40
−20
0
20
40
60

Item representation: One-hot encoding

Words in documents and other categorical features such as **user/product ids** in recommenders, **names of places**, visited **URL**s, 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:

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

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.

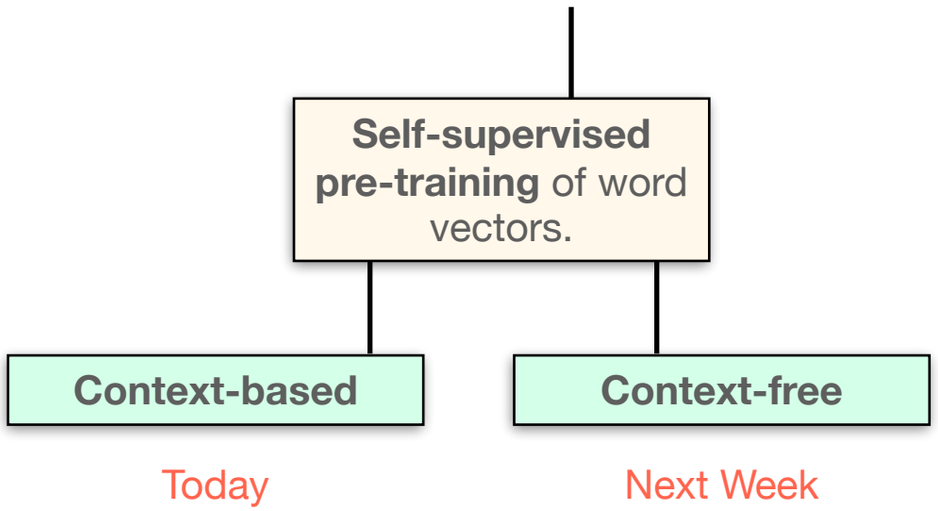
Item representation: One-hot encoding

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?

Embedding

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

Embedding matrix

$$\underbrace{[0,0,1]^T}_{\text{Item index = 3}} \cdot \begin{bmatrix} 0.29572738 & 0.88443109 & 0.21831979 \\ 0.03157878 & 0.71250614 & 0.22703532 \\ 0.12386669 & 0.74266196 & 0.91580261 \end{bmatrix} = \begin{bmatrix} 0.21831979 \\ 0.22703532 \\ 0.91580261 \end{bmatrix}$$

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 "finding" a set of elements of the embedding matrix that "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).

$$[[4], [20]] \text{ is turned into } [[0.25, 0.1], [0.6, -0.2]]$$

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())
```

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

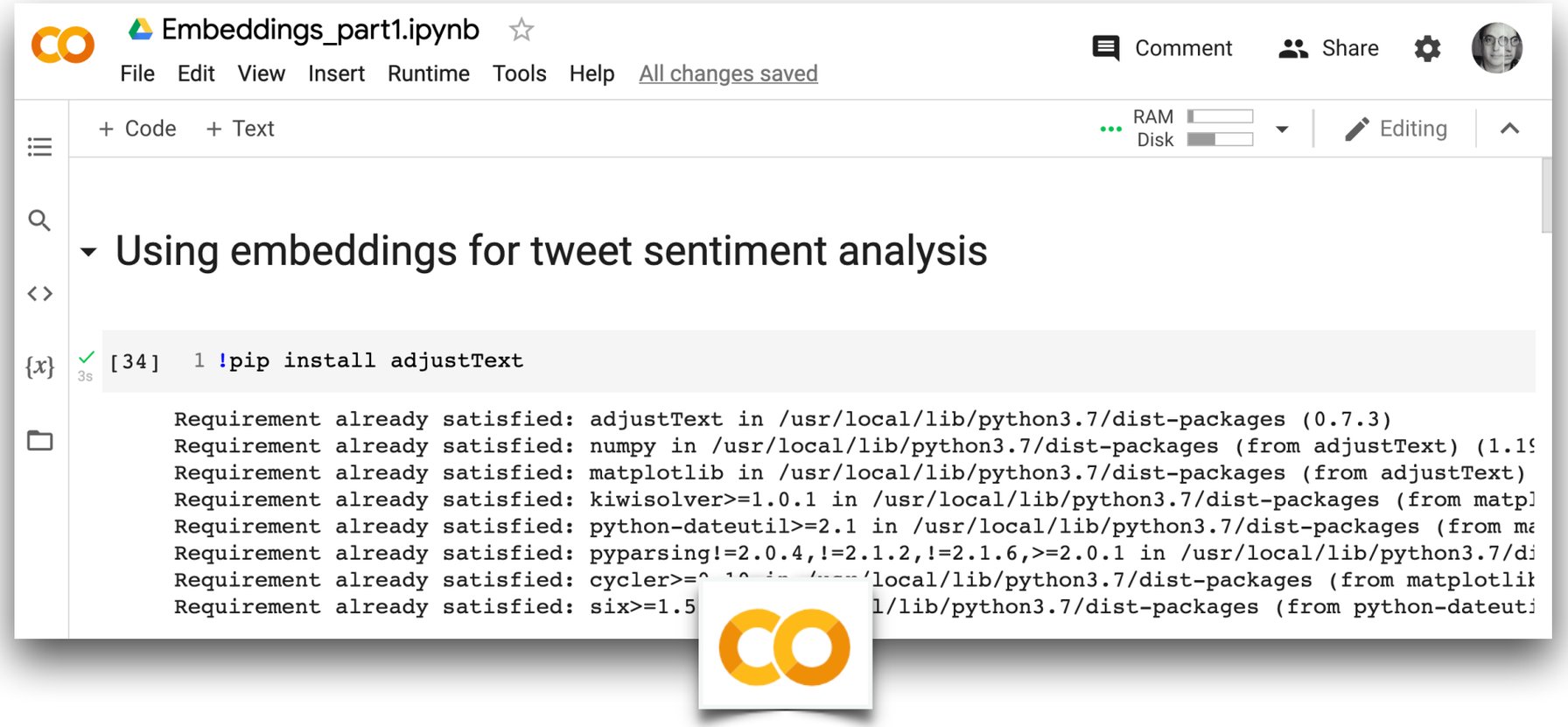
Lets see how to do it using restaurant reviews data.

```
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'])
```

This 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



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**, **homonony**, **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 to 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

```
morphomes > 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 generate **a single** word embedding representation (a **vector** of numbers) for each word in the vocabulary.

On the other hand, **context-based** models generate 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-ocurrences by using SVD

Some previous methods were based on **counting** co-ocurrences 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-ocurrences by using SVD

Some previous methods were based on **counting** co-ocurrences 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)

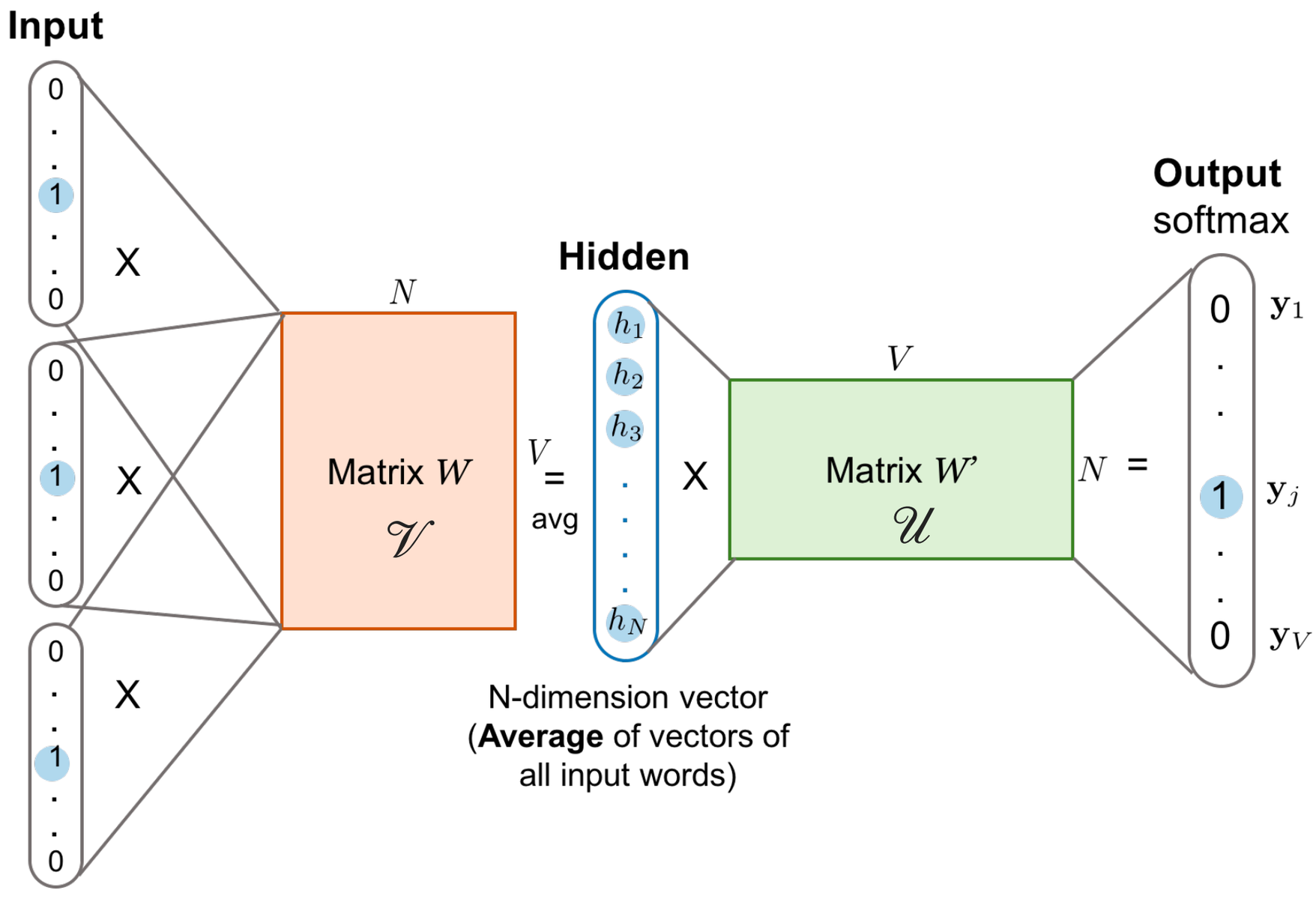
We create two matrices, $\mathscr{V} \in \mathbb{R}^{n \times |V|}$ and $\mathscr{U} \in \mathbb{R}^{|V| \times n}$, where n is an arbitrary size which defines the size of our embedding space.

$\mathscr{V}$ is the input word matrix such that the i-th column of $\mathscr{V}$ 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, $\mathscr{U}$ is the output word matrix. The j-th row of $\mathscr{U}$ is an n-dimensional embedded vector for word w_j when it is an output of the model. We denote this row of $\mathscr{U}$ 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)

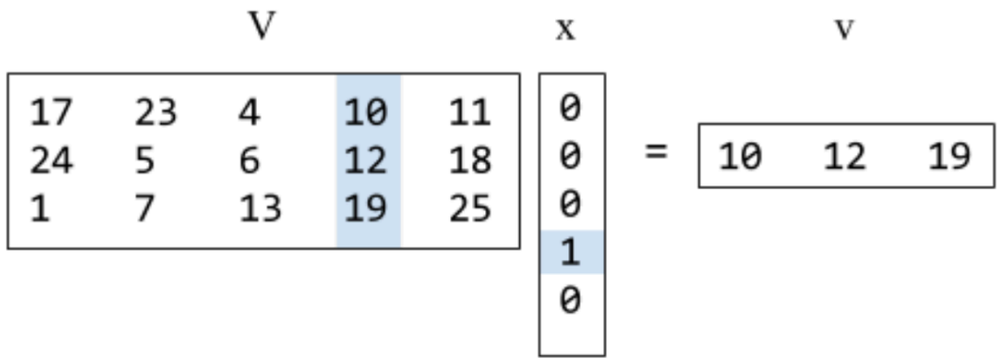


https://lilianweng.github.io/lil-log/2017/10/15/learning-word-embedding.html#loss-functions

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



- Average these vectors to get $h = \frac{v_{c-m}+v_{c-m+1}+\ldots+v_{c+m}}{2m} \in \mathbb{R}^{n}$
- 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*.

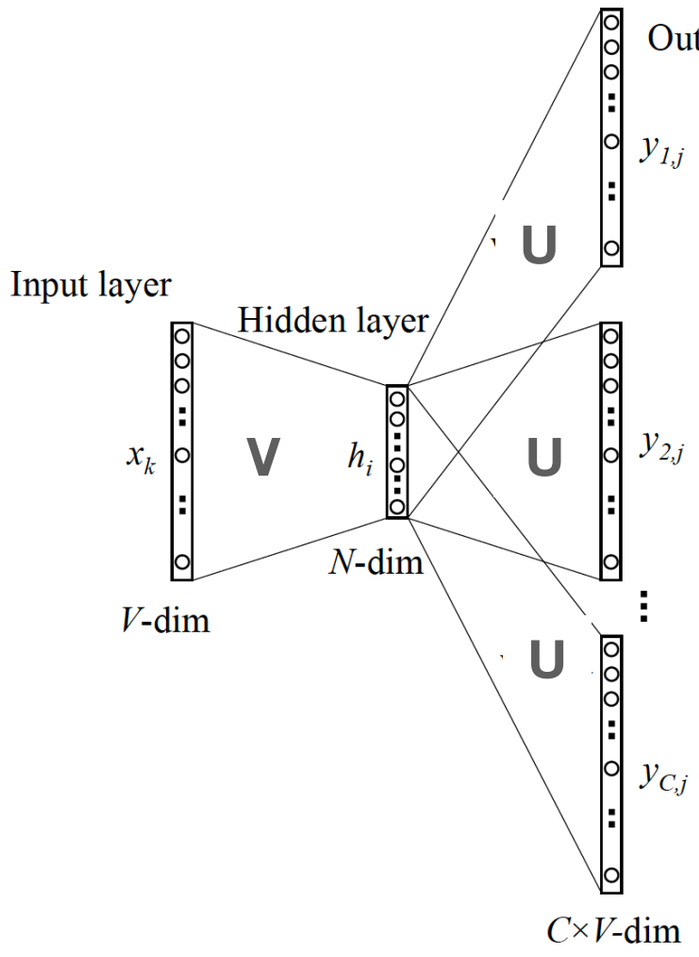
We thus formulate our optimization objective as:

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

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`.

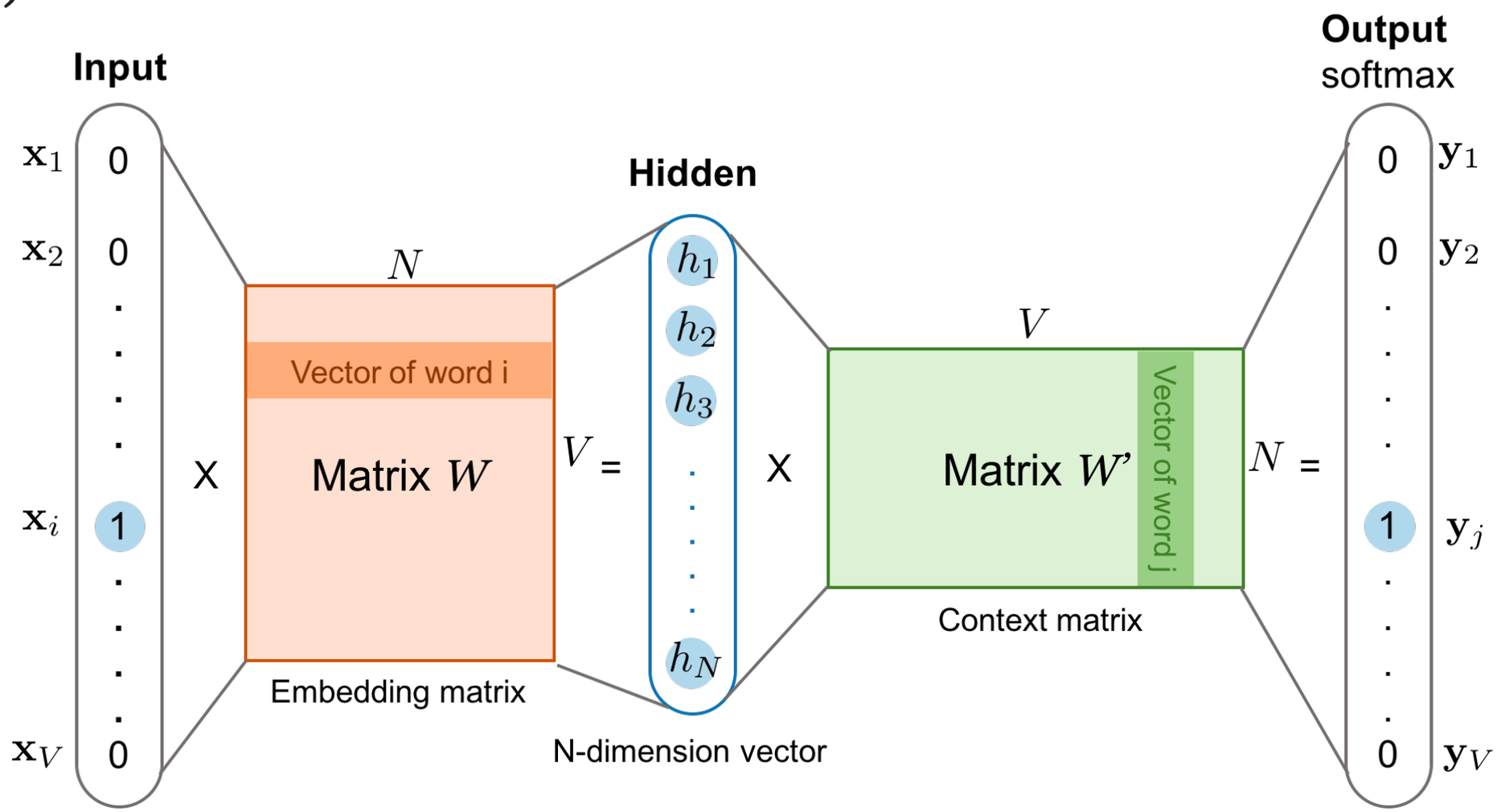
Here we call the word `jumped` the context. We call this type of model a **Skip-Gram model**.



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"*).



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

Skip-gram model

We thus formulate our optimization objective as:

$$\text{minimize } J = -\log P(w_{c-m}, \ldots, w_{c-1}, w_{c+1}, \ldots, w_{c+m} \mid w_c)$$

$$= -\log \prod_{j=0; j\neq m}^{2m} \frac{\exp(u_{c-m+j}^T v_c)}{\sum_{k=1}^{|V|} \exp(u_k^T v_c)}$$

$$= -\log \prod_{j=0; j\neq m}^{2m} u_{c-m+j}^T v_c + 2m \log \sum_{k=1}^{|V|} \exp(u_k^T v_c)$$

Skip-gram model

We thus formulate our optimization objective as:

$$\text{minimize } J = -\log P(w_{c-m}, \ldots, w_{c-1}, w_{c+1}, \ldots, w_{c+m} \mid w_c)$$

$$= -\log \prod_{j=0; j \neq m}^{2m} \frac{\exp(u_{c-m+j}^T v_c)}{\sum_{k=1}^{|V|} \exp(u_k^T v_c)}$$

$$= -\log \prod_{j=0; j \neq m}^{2m} u_{c-m+j}^T v_c + 2m \log \sum_{k=1}^{|V|} \exp(u_k^T v_c)$$

Negative Sampling

Note that 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:

`model.most_similar('king', topn=10)`	`model.most_similar('queen', topn=10)`
(word, similarity with 'king')	(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.

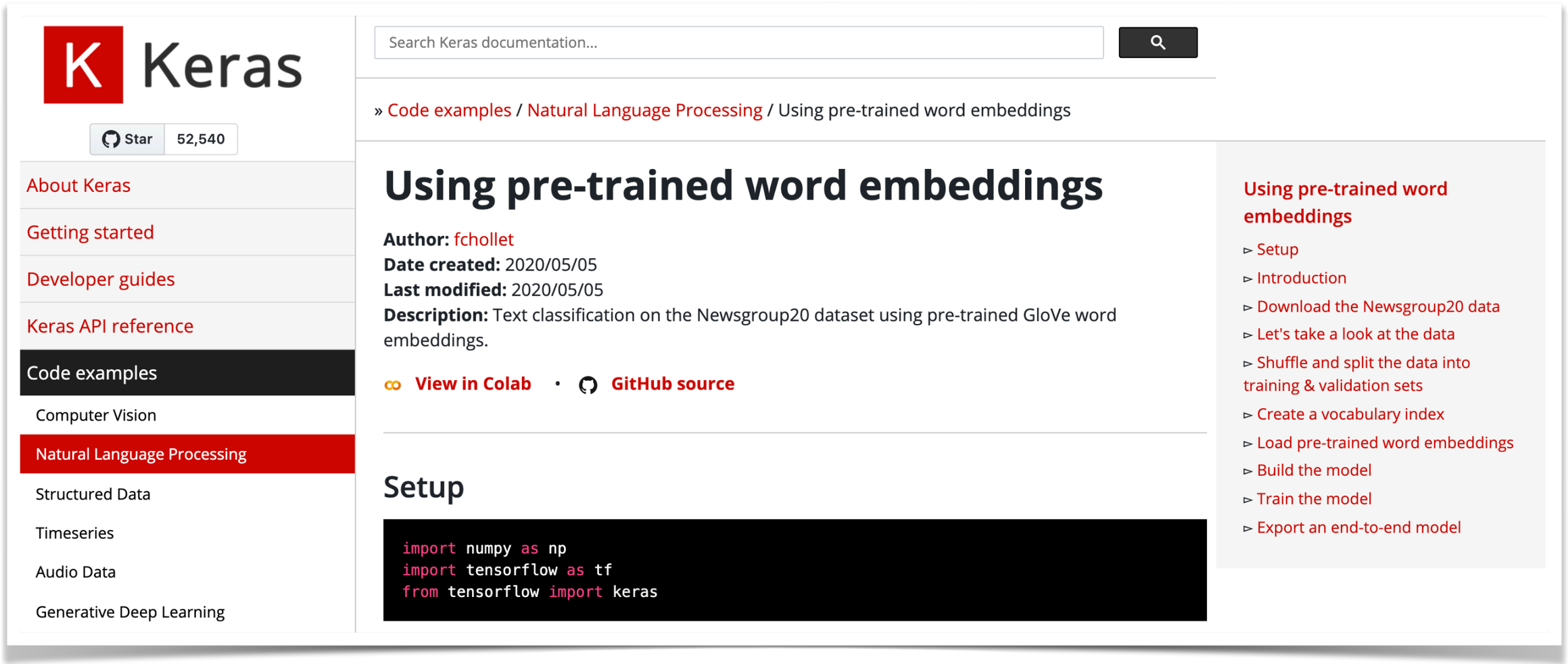
Related models

GloVe: *GloVe* is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space.

fastText: *fastText* embeddings exploit **subword information** to construct word embeddings. Representations are learnt of character *n*-grams, and words represented as the sum of the *n*-gram vectors. This extends the *word2vec* type models with subword information.

Example

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



Example

Embeddings, Transformers and Transfer Learning

Using transformer embeddings like BERT in spaCy

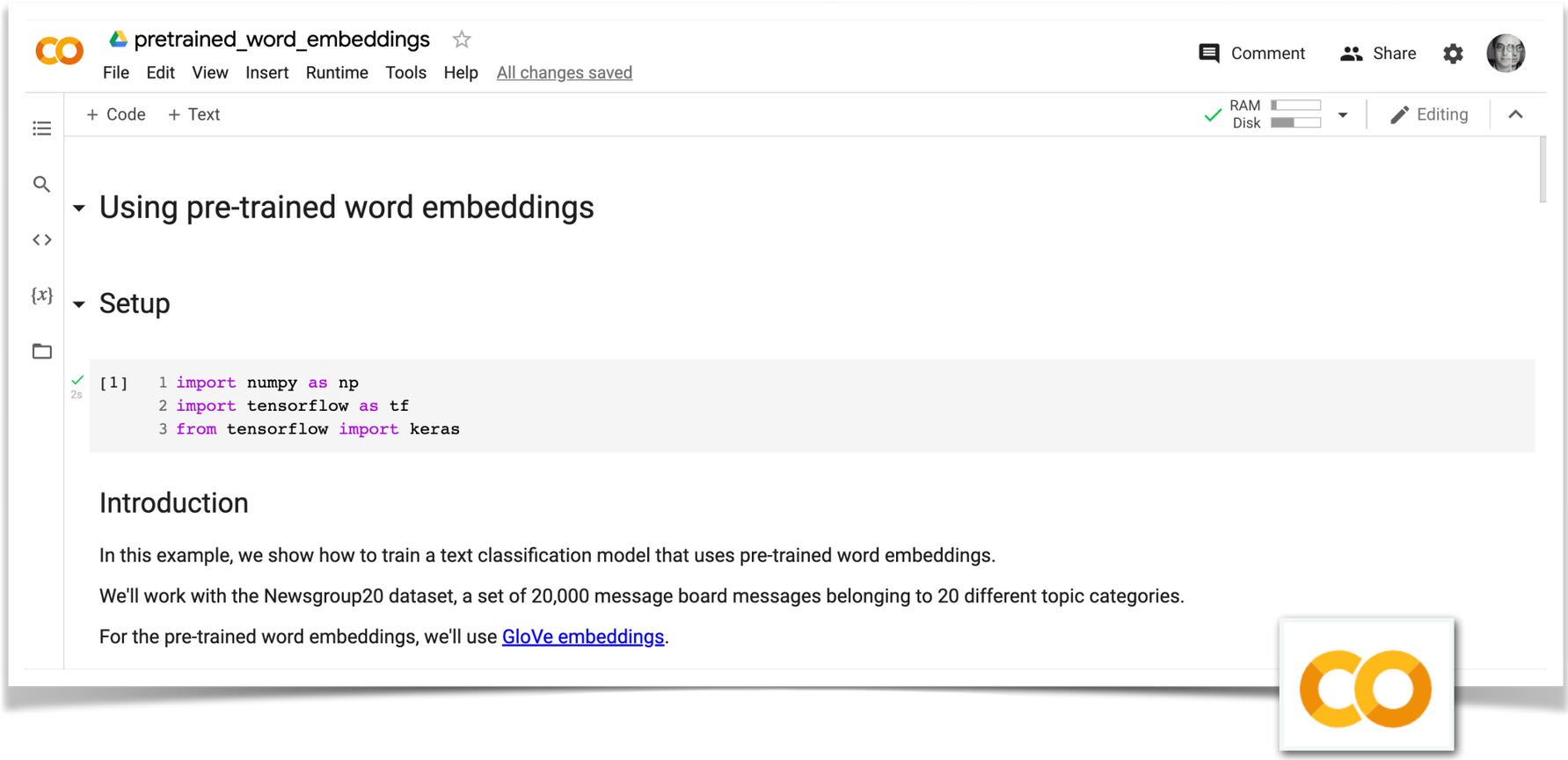
spaCy supports a number of **transfer and multi-task learning** workflows that can often help improve your pipeline's efficiency or accuracy. Transfer learning refers to techniques such as word vector tables and language model pretraining. These techniques can be used to import knowledge from raw text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

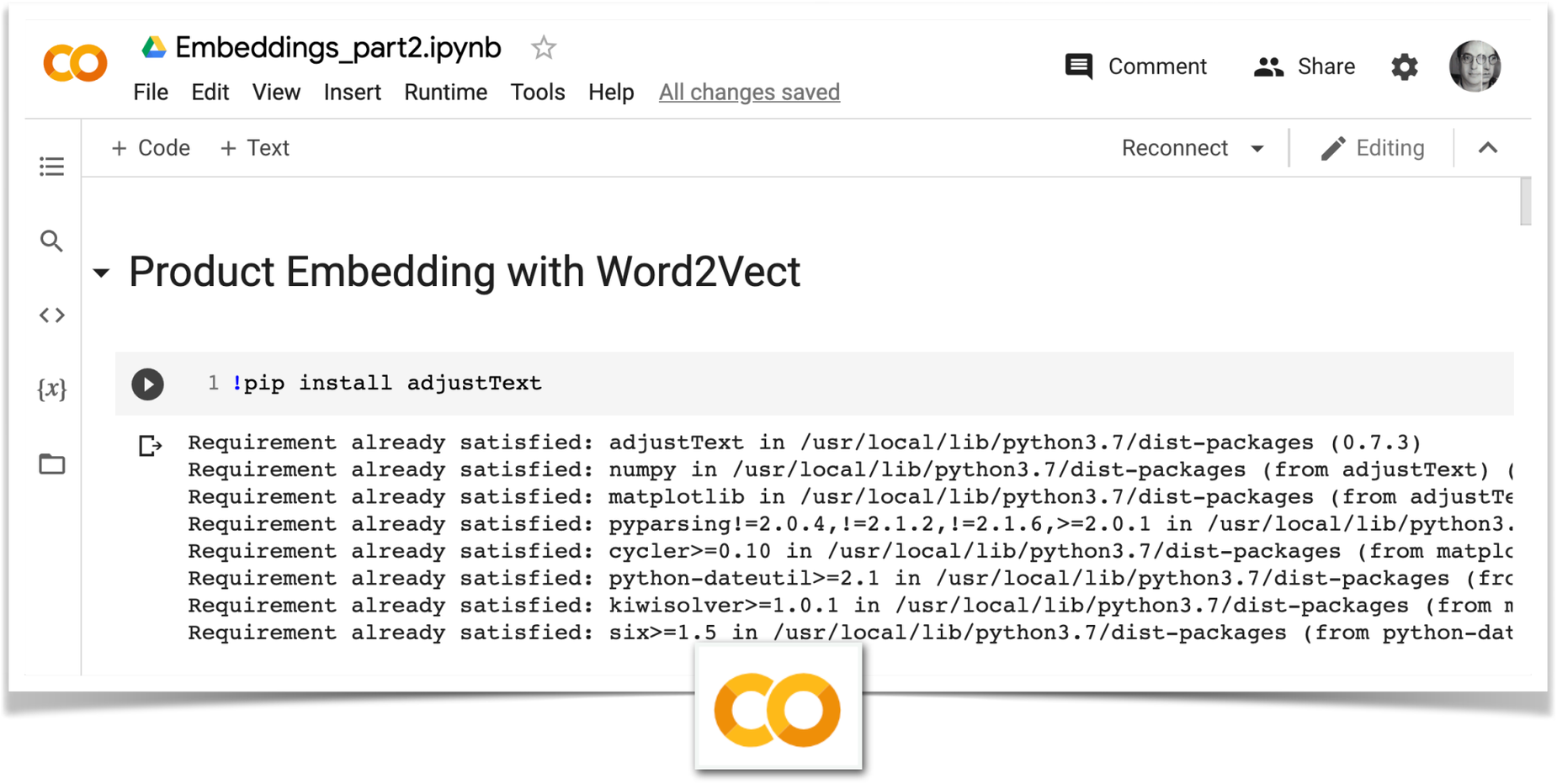
What's the difference between word vectors and language models?

When should I add word vectors to my model?

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



How to learn word embeddings?



Par2Vec

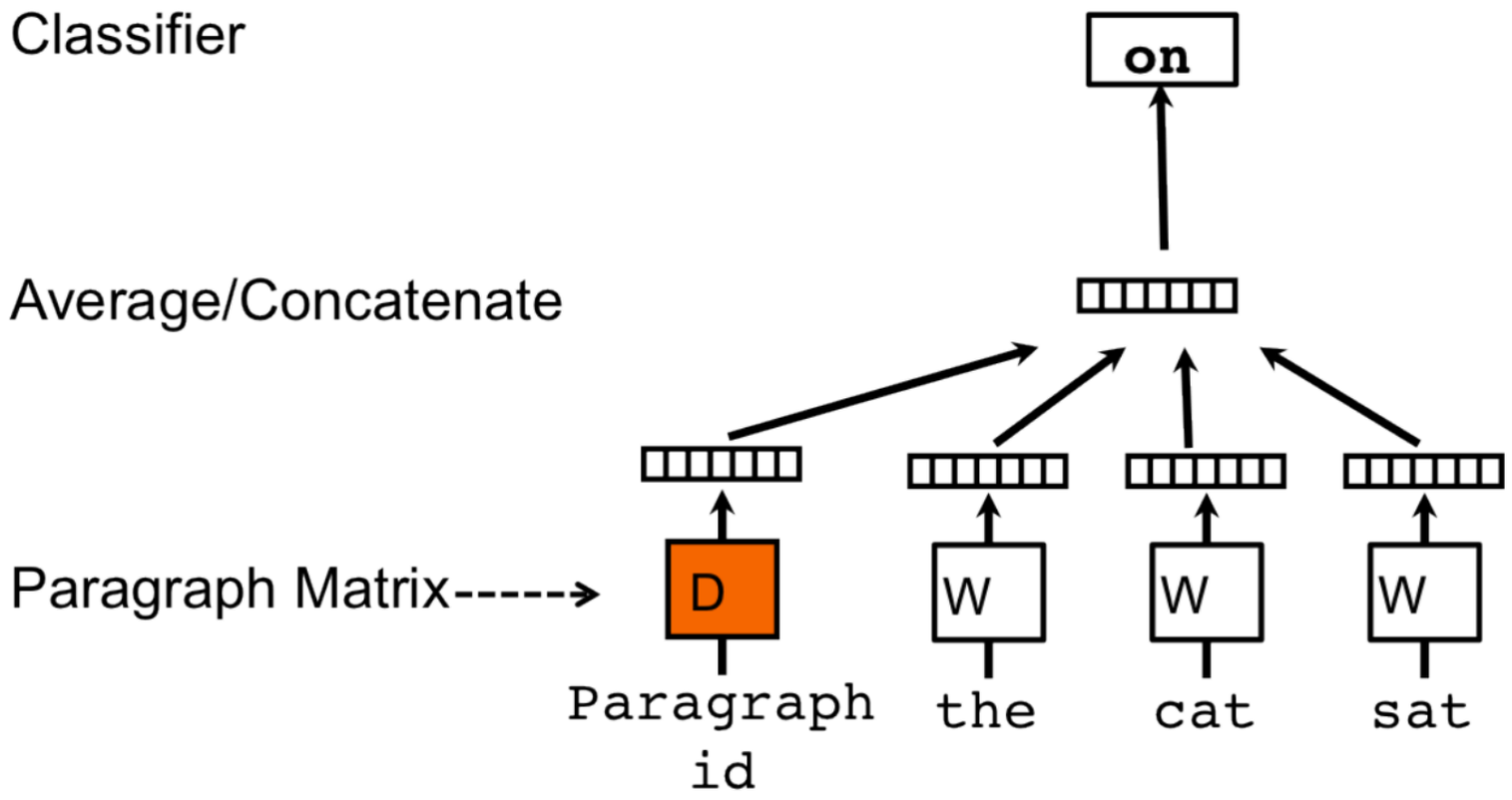
What about a **vector representation for phrases**/paragraphs/documents?

The ***par2vec*** approach for learning paragraph vectors is inspired by the methods for learning the word vectors.

We will consider a paragraph vector. The paragraph vectors are also asked to contribute to the prediction task of the next word given many contexts sampled from the paragraph.

In *par2vec* framework, every paragraph is mapped to a unique vector, represented by a column in matrix D and every word is also mapped to a unique vector, represented by a column in matrix W. The paragraph vector and word vectors are averaged or concatenated to predict the next word in a context.

Par2Vec



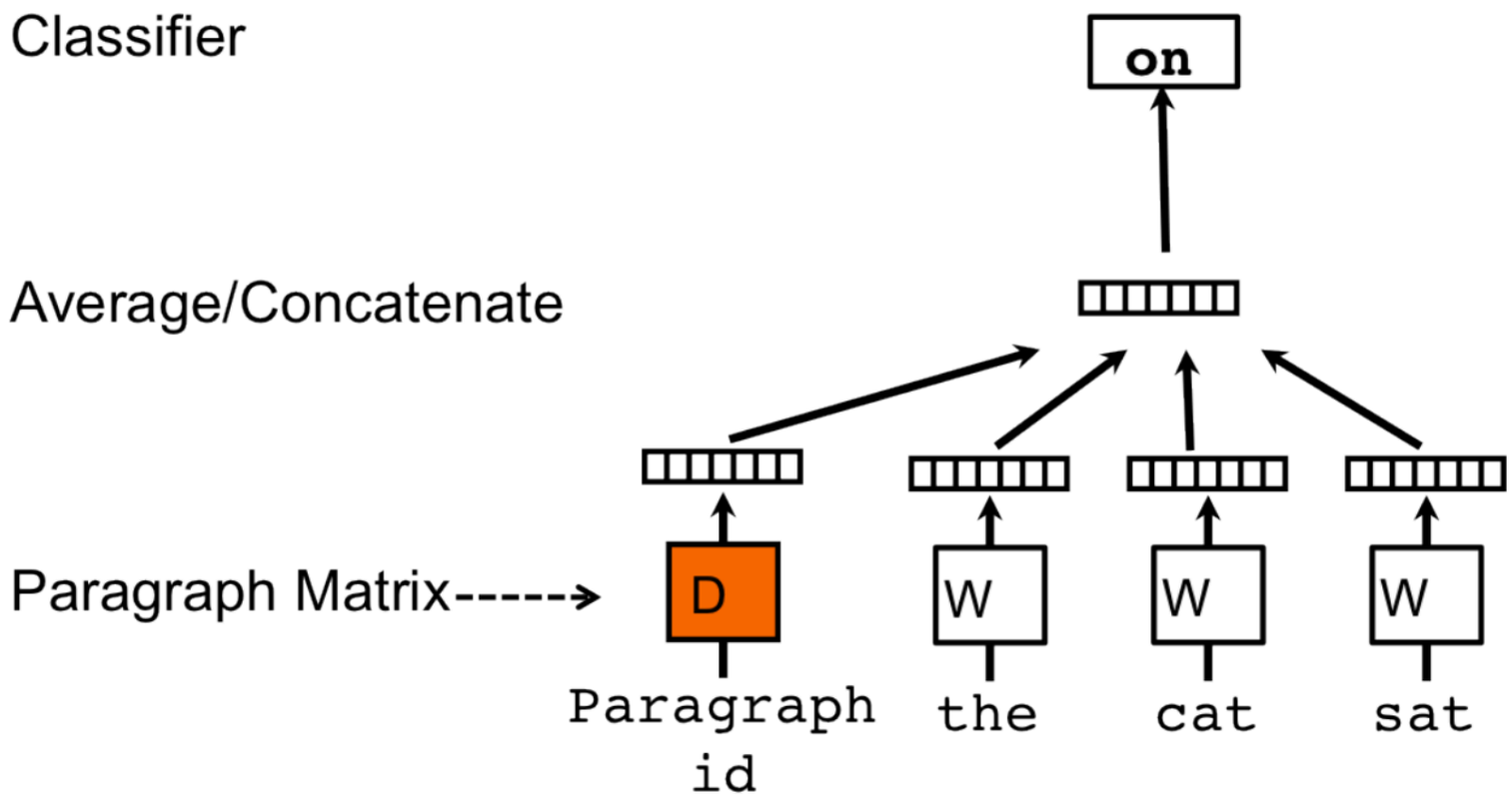
The paragraph token can be thought of as another word. It acts as a memory that remembers what is missing from the current context – or the topic of the paragraph.

The contexts are fixed-length and sampled from a sliding window over the paragraph.

The paragraph vector is shared across all contexts generated from the same paragraph but not across paragraphs.

The word vector matrix W, however, is shared across paragraphs.

Par2Vec

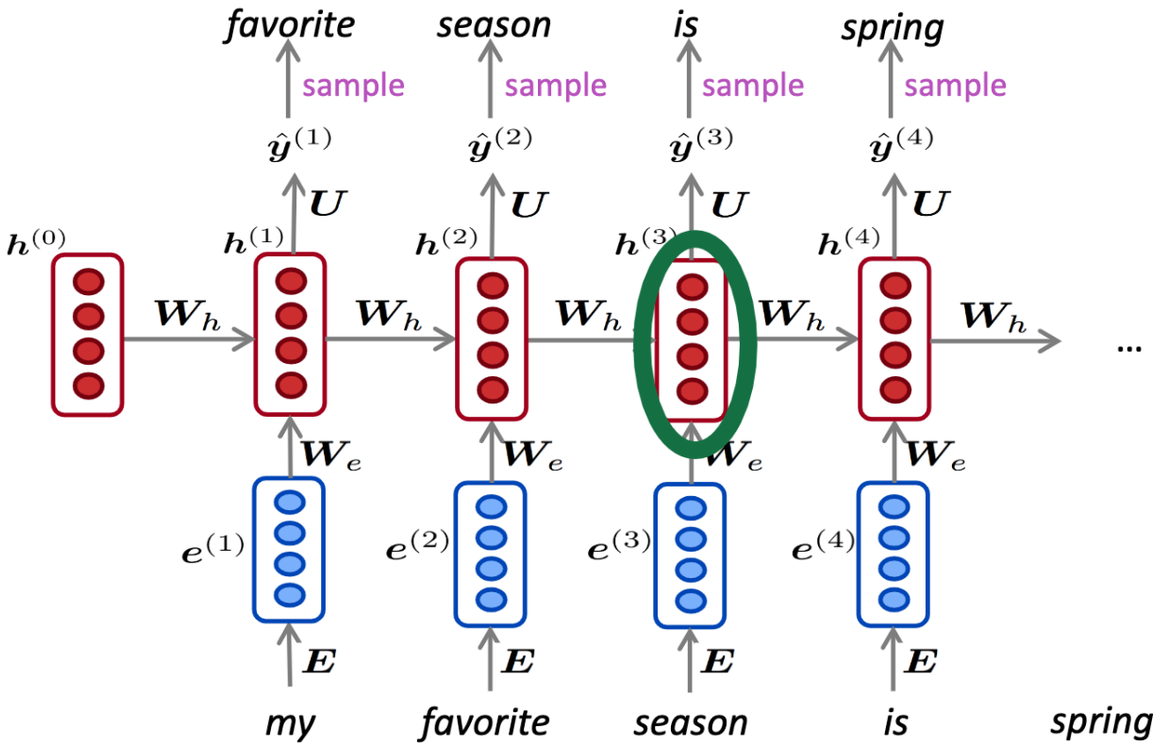


At prediction time, one needs to perform an inference step to compute the paragraph vector for a new paragraph.

This is also obtained by gradient descent. In this step, the parameters for the rest of the model, the word vectors and the softmax weights, are fixed.

From context-free to context-based respresentations

Recurrent models can generate context-dependent word representations!



Limitations:

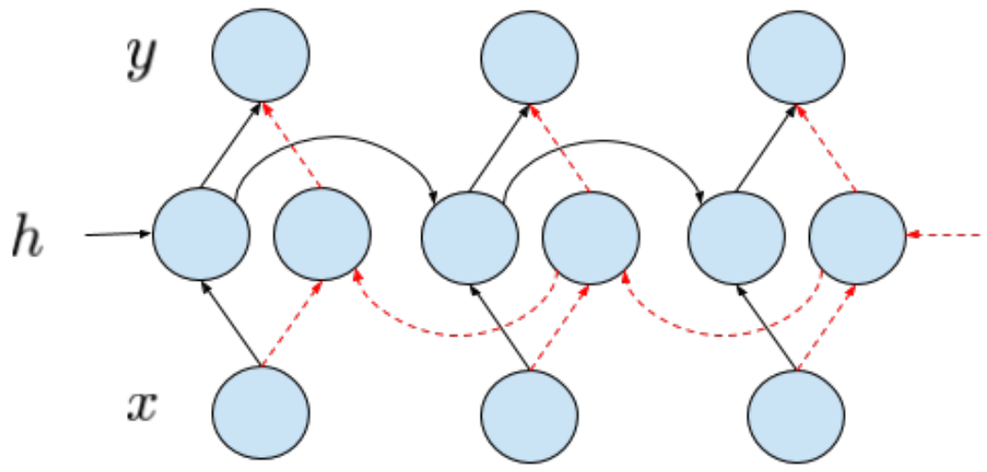
- This is a right-to-left model.
- This is task-dependent.

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 networs, 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):

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

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

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

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

Bidirectional RNNs can be stacked as usual!

Bi-directional LSTM

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

```

```
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
=================================================================
Total params: 2,757,761
Trainable params: 2,757,761
Non-trainable params: 0
```

Bi-directional LSTM

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

```

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

This could be a good word representation …

```
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
=================================================================
Total params: 2,757,761
Trainable params: 2,757,761
Non-trainable params: 0
```

Other resources

https://colab.research.google.com/drive/16cM5NXedlrvU2mp-HcYKs9OIMkYItTS1?usp=sharing

https://nlp-css-201-tutorials.github.io/nlp-css-201-tutorials/