Jordi Vitrià

Deep Learning

Deep representations of words

One-hot encoding

Deep Learning algorithms require the input to be represented as (sequences of) **fixed-length feature tensors**.

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:

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

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

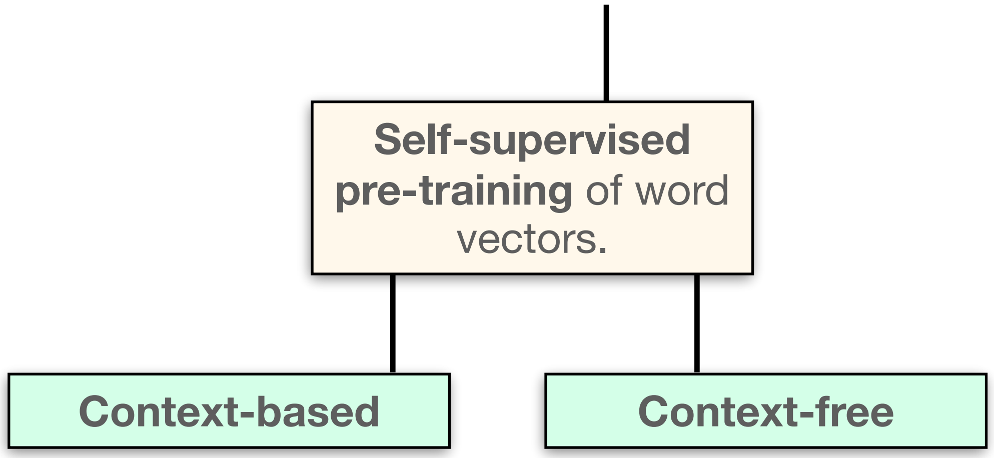
One-hot encoding

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 [predictive] meaning)!

What other alternatives are there?



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.

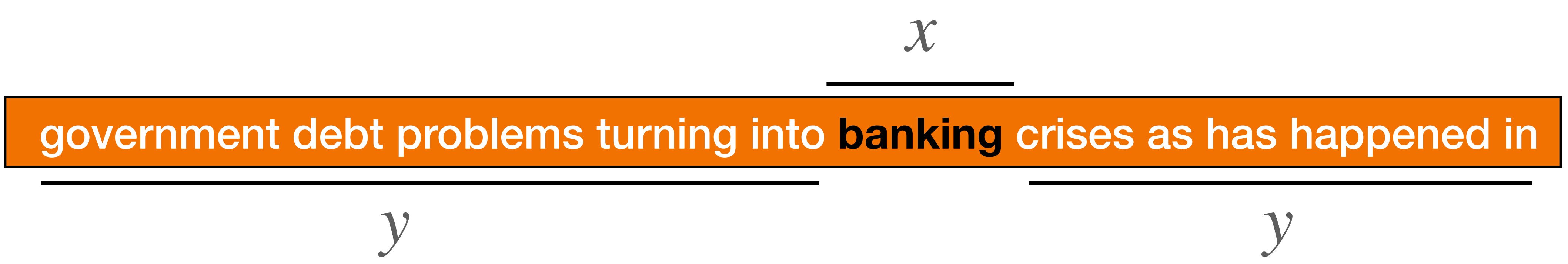
These vectors can then be used across a **wide range of natural language understanding tasks**.

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?


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

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?**

$$\underbrace{\text{government debt problems turning into}}_{x}\ \overset{y}{\overline{\textbf{banking}}}\ \underbrace{\text{crises as has happened in}}_{x}$$

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)

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 $(v_{c-m} = \mathcal{V}x^{(c-m)}, v_{c-m+1} = \mathcal{V}x^{(c-m+1)}, \dots, v_{c+m} = \mathcal{V}x^{(c+m)} \in \mathbb{R}^n)$

$$\begin{array}{cccc} \mathrm{V} & \mathrm{x} & & \mathrm{v} \\ \begin{bmatrix} 17 & 23 & 4 & 10 & 11 \\ 24 & 5 & 6 & 12 & 18 \\ 1 & 7 & 13 & 19 & 25 \end{bmatrix} & \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} & = & \begin{bmatrix} 10 & 12 & 19 \end{bmatrix} \end{array}$$

- 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 $\hat{y} = \mathrm{softmax}(z) \in \mathbb{R}^{|V|}$.
- 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)

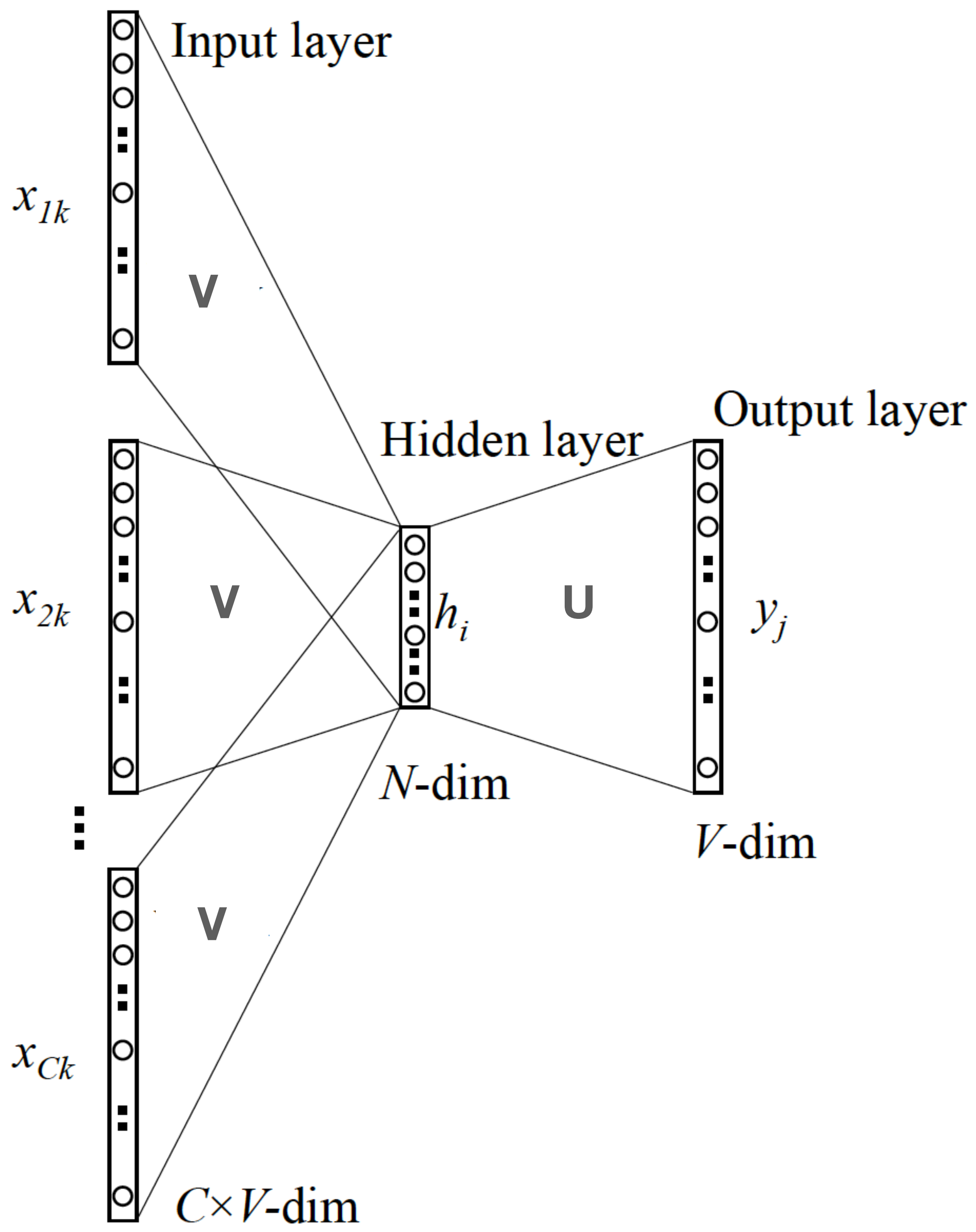
How can we learn these two matrices? Well, we need to create an objective function. We choose a classification loss function.

Here, we use a popular choice: *cross entropy*.

We thus formulate our optimization objective as:

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

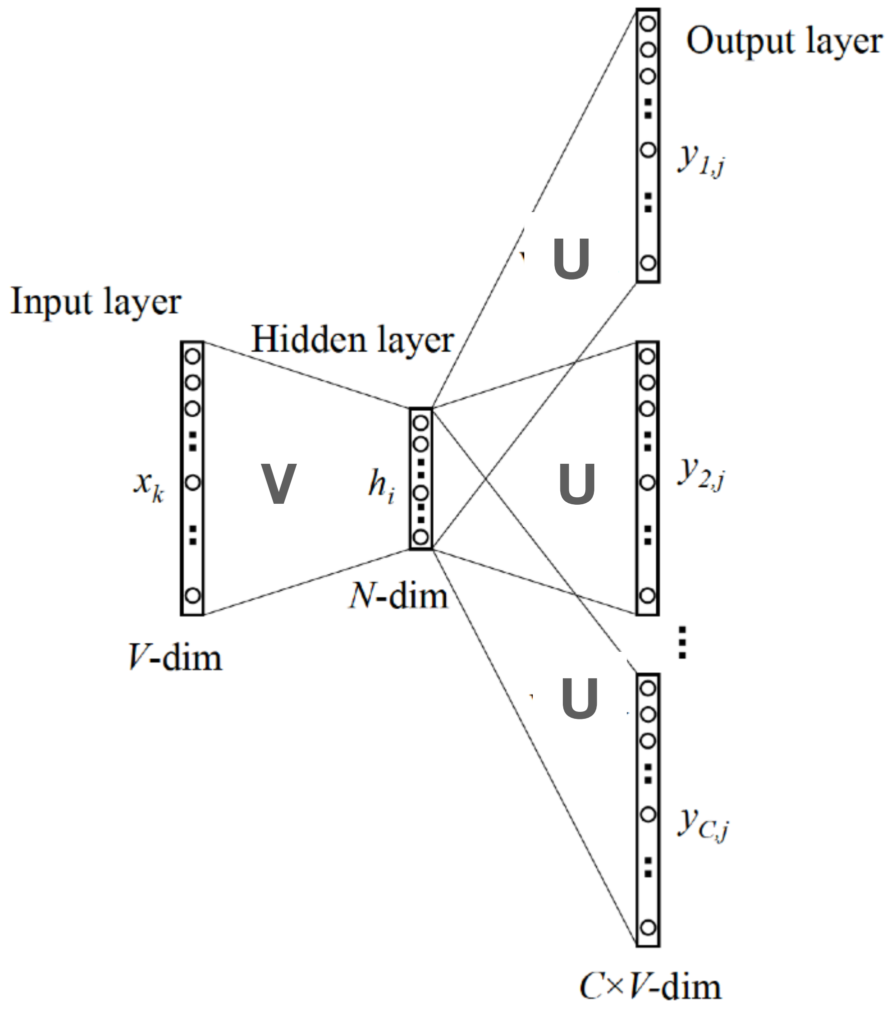
Continuous Bag of Words Model (CBOW)



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

We thus formulate our optimization objective as:

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

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 that takes into account this fact…

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.

Other 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.

Par2Vec

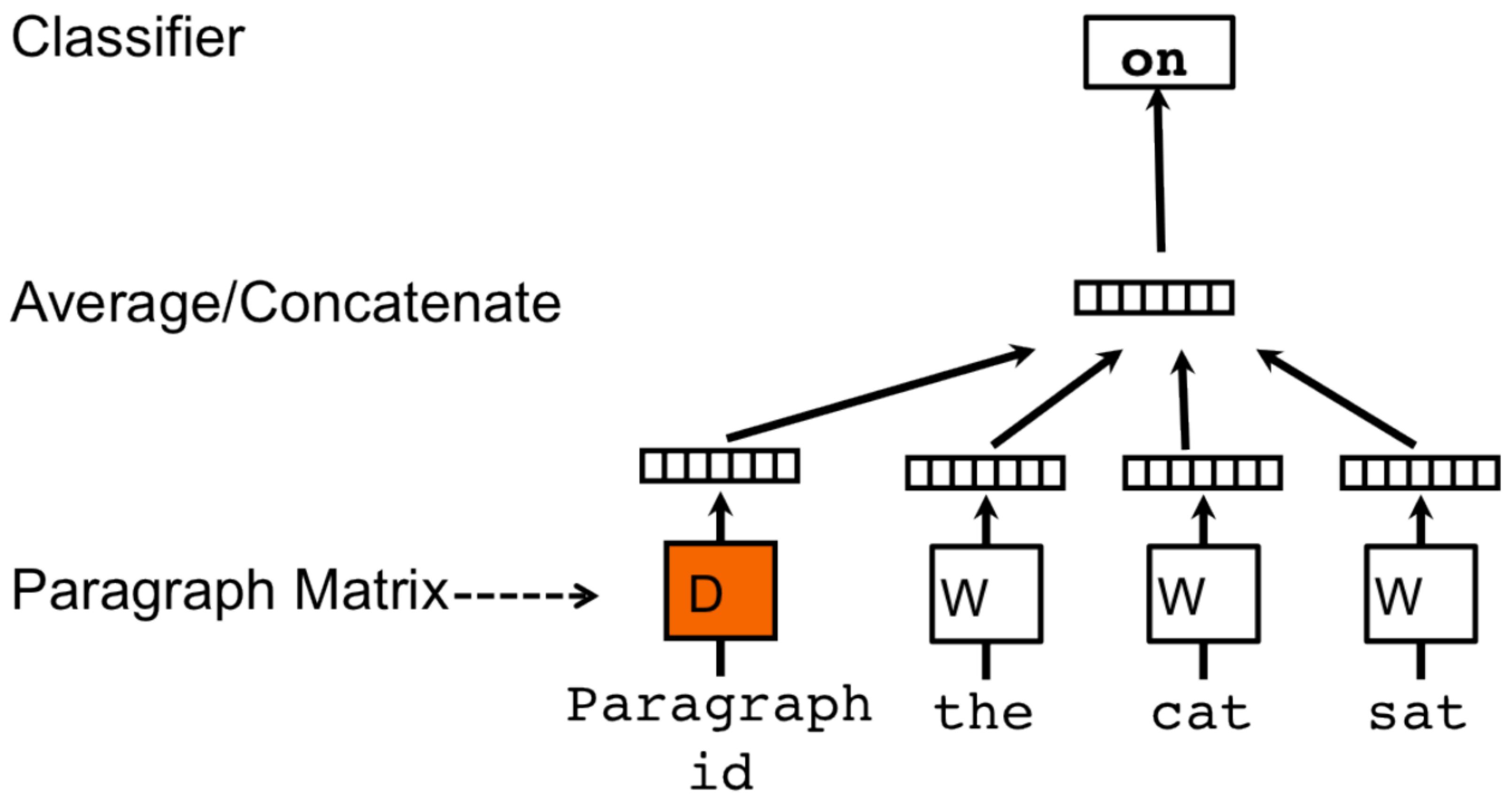
What about a **vector representation for phrases**/paragraphs/documents?

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



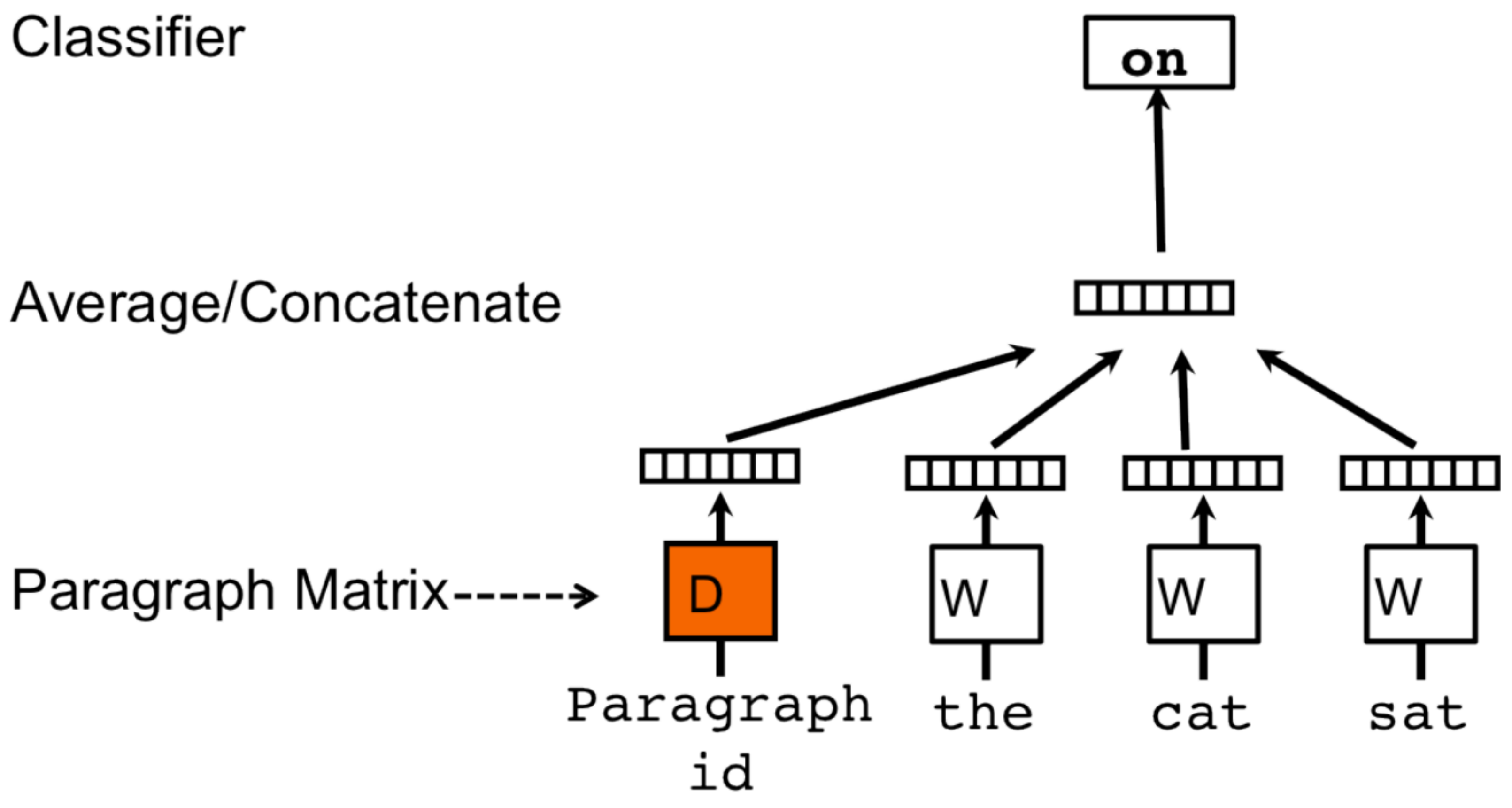
The paragraph token can be thought of as another word. It acts as a memory that remembers what is missing from the current context – or the topic of the paragraph.

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

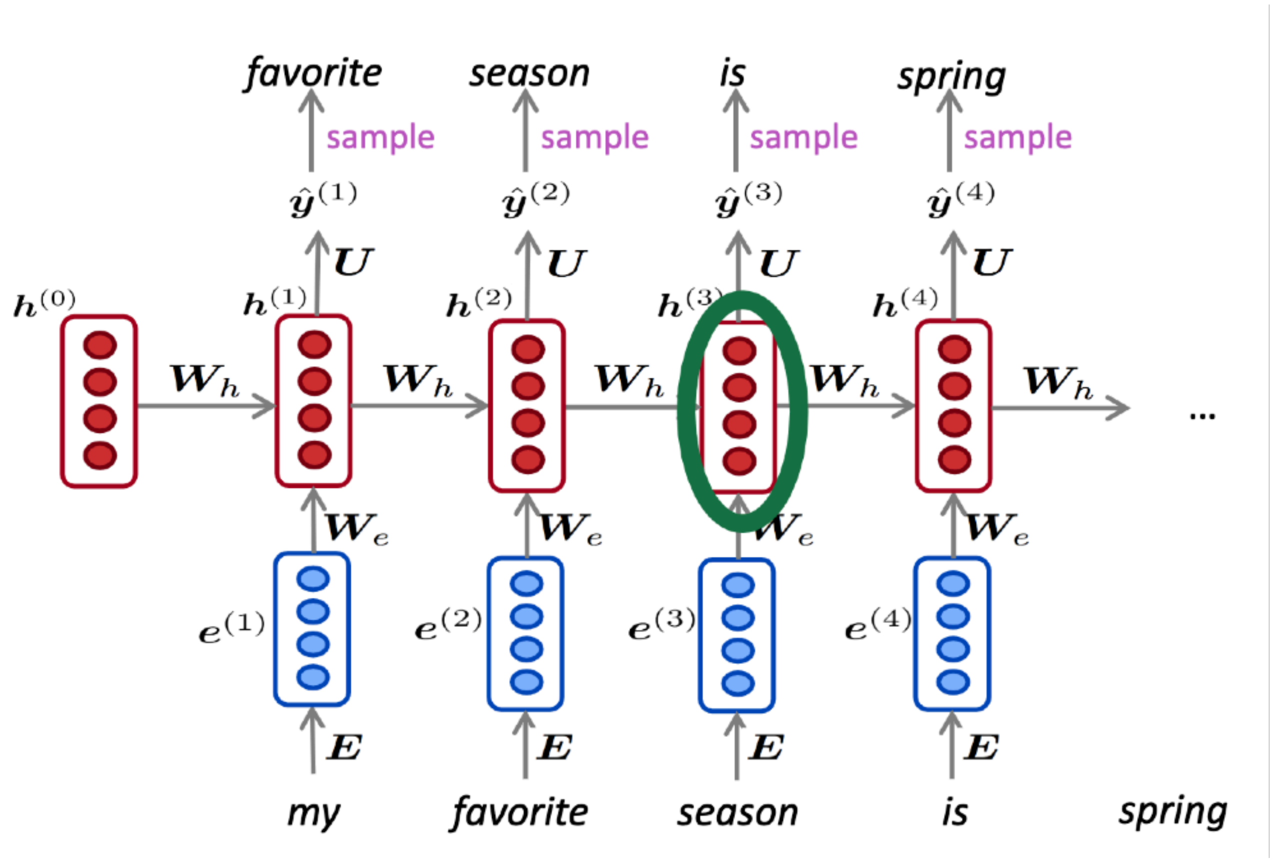


At prediction time, one needs to perform an inference step to compute the paragraph vector for a new paragraph.

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!



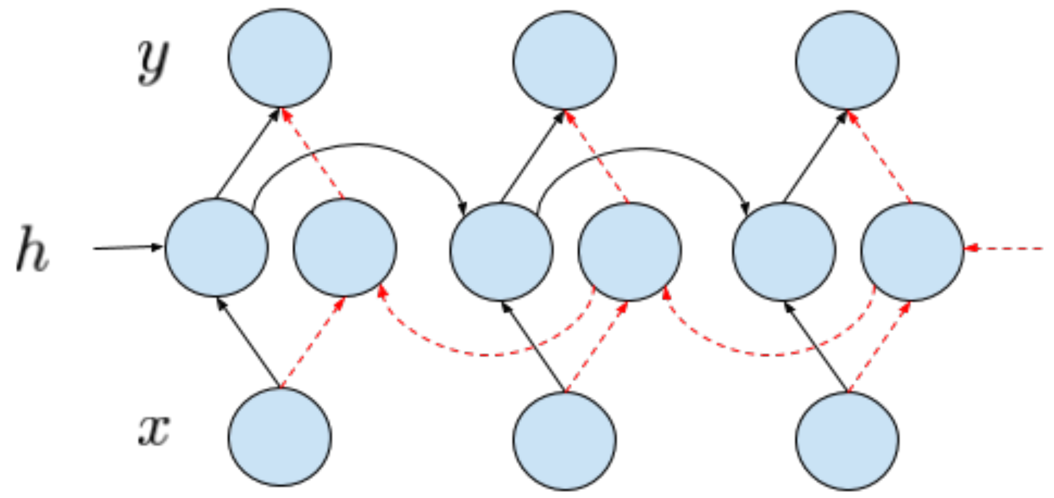
Limitations:
- This is a right-to-left model.
- This is task-dependent.

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):

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

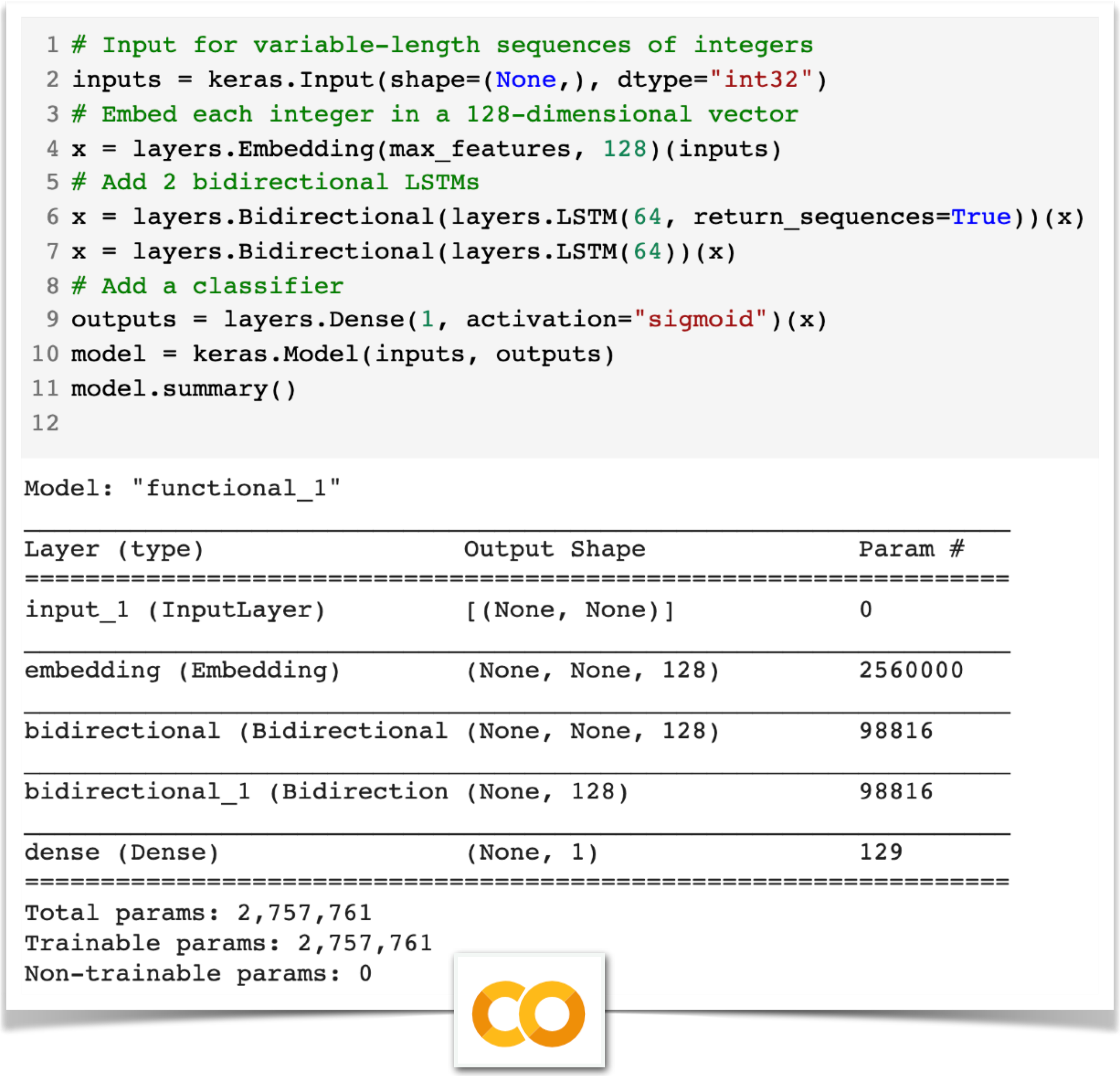
$$\overleftarrow{h}_t = f(\overleftarrow{W}x_t + \overleftarrow{V}\overleftarrow{h}_{t+1} + \overleftarrow{b})$$

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

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

Bidirectional RNNs can be stacked as usual!

Bi-directional LSTM



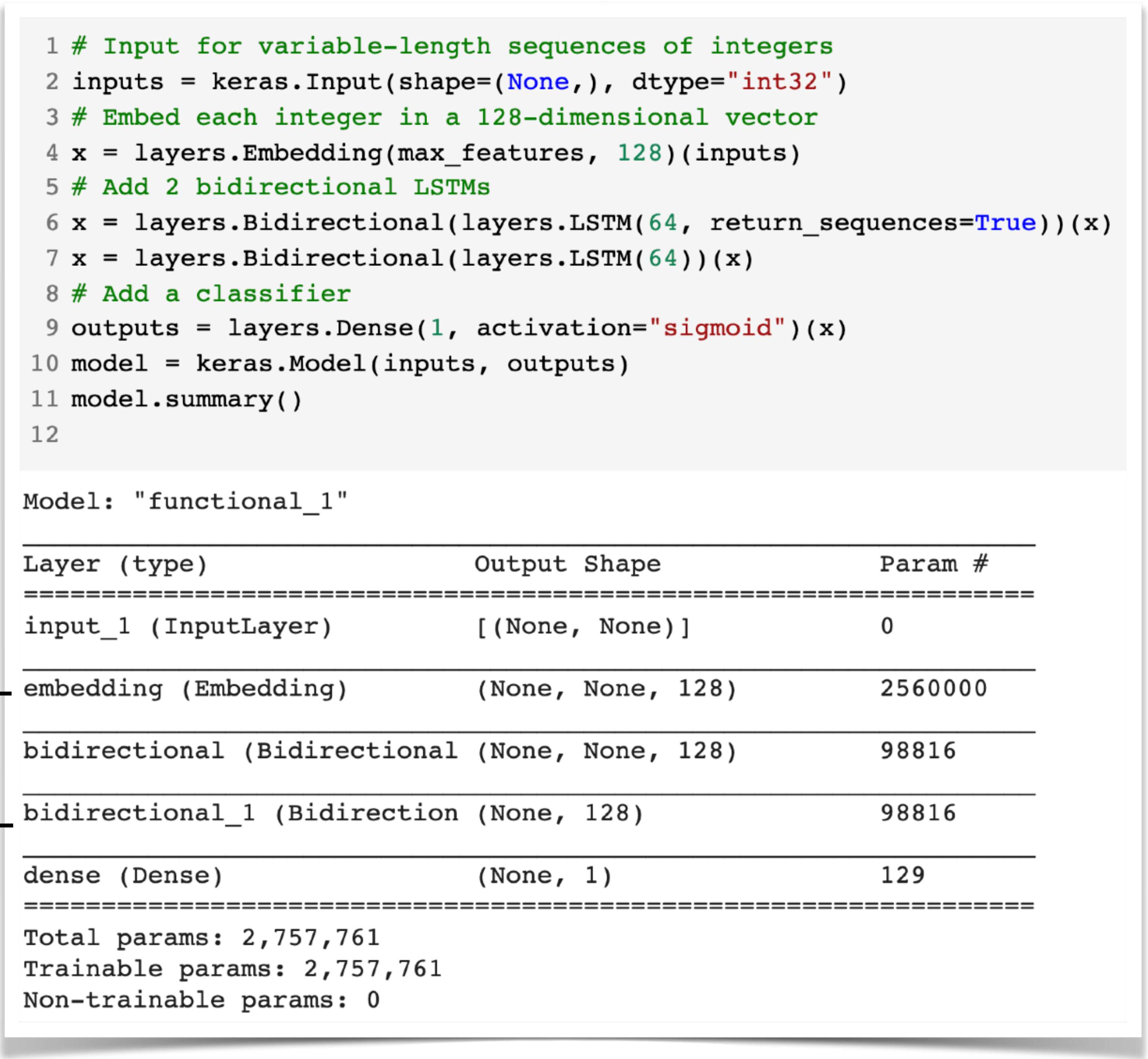
```python
# Input for variable-length sequences of integers
inputs = keras.Input(shape=(None,), dtype="int32")
# Embed each integer in a 128-dimensional vector
x = layers.Embedding(max_features, 128)(inputs)
# Add 2 bidirectional LSTMs
x = layers.Bidirectional(layers.LSTM(64, return_sequences=True))(x)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

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

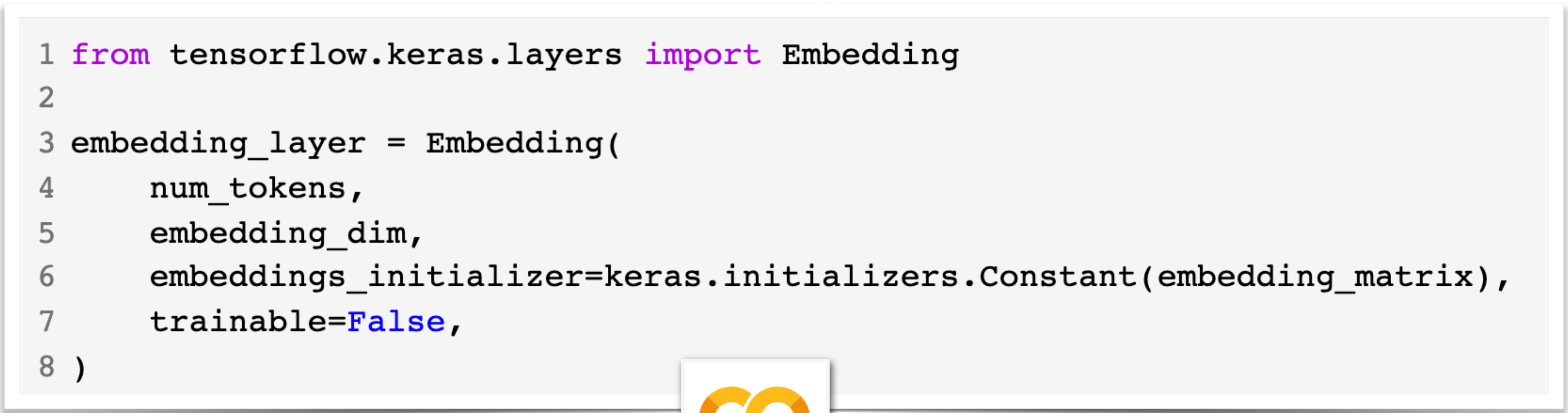
This could be a good word representation …



```
# 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
```

How to use pre-trained word embeddings?



```python
from tensorflow.keras.layers import Embedding

embedding_layer = Embedding(
    num_tokens,
    embedding_dim,
    embeddings_initializer=keras.initializers.Constant(embedding_matrix),
    trainable=False,
)
```