

Data Science

Lecture 8: Text Data Processing



UNIVERSITY
OF AMSTERDAM

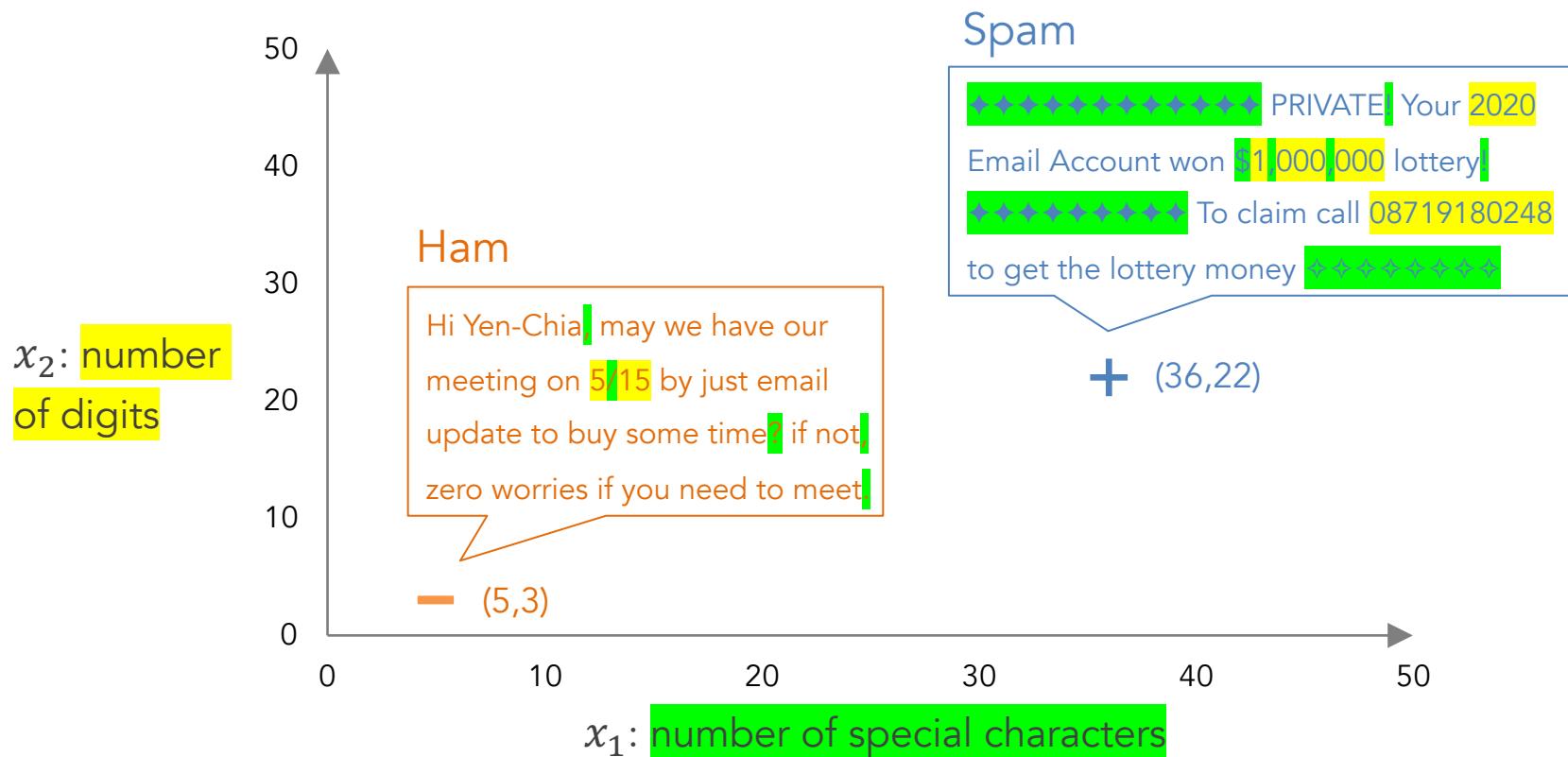
Lecturer: Yen-Chia Hsu

Date: Mar 2023

This lecture covers the pipeline of **Natural Language Processing (NLP)**:

- Text preprocessing
- Bag of words and TF-IDF
- Topic modeling
- Word embeddings and Word2Vec
- Sentence/document representations
- Attention mechanism

Previously, we have learned the spam classification example about how to represent messages as data points on a 2-dimensional space, using some hand-crafted features.



Typically, before the deep learning era, we need to preprocess text using **tokenization** (i.e., separating words) and **normalization** (i.e., standardizing the word format).

Google, headquartered in Mountain View (1600 Amphitheatre Pkwy, Mountain View, CA 940430), unveiled the new Android phone for \$799 at the Consumer Electronic Show. Sundar Pichai said in his keynote that users love their new Android phones.



```
['google', 'headquarter', 'mountain', 'view', 'amphitheatre', 'pkwy', 'mountain', 'view', 'ca', 'unveil',  
'new', 'android', 'phone', 'consumer', 'electronic', 'show', 'sundar', 'pichai', 'say', 'keynote', 'user',  
'love', 'new', 'android', 'phone']
```

The [tokenization](#) step separates a sentence into word fragments (i.e., an array of words).

We can lower the cases first before tokenization.

Google, headquartered in Mountain View (1600 Amphitheatre Pkwy, Mountain View, CA 940430), unveiled the new Android phone for \$799 at the Consumer Electronic Show. Sundar Pichai said in his keynote that users love their new Android phones.

```
>>> import nltk  
>>> tokens = nltk.tokenize.word_tokenize(s.lower())
```

[`'google', ''', 'headquartered', 'in', 'mountain', 'view', '(', '1600', 'amphitheatre', 'pkwy', ''', 'mountain', 'view', ''', 'ca', '940430', ')', ''', 'unveiled', 'the', 'new', 'android', 'phone', 'for', '$', '799', 'at', 'the', 'consumer', 'electronic', 'show', '.', 'sundar', 'pichai', 'said', 'in', 'his', 'keynote', 'that', 'users', 'love', 'their', 'new', 'android', 'phones', '.'`]

During tokenization, we can also [remove unwanted tokens](#), such as punctuations, digits, symbols, emojis, stop words (i.e., high frequency words, like "the"), etc.

```
['google', ',', 'headquartered', 'in', 'mountain', 'view', '(', '1600', 'amphitheatre', 'pkwy', ',', 'mountain',
'view', ',', 'ca', '940430', ')', ',', 'unveiled', 'the', 'new', 'android', 'phone', 'for', '$', '799', 'at', 'the',
'consumer', 'electronic', 'show', '.', 'sundar', 'pichai', 'said', 'in', 'his', 'keynote', 'that', 'users', 'love',
'their', 'new', 'android', 'phones', '']
```

```
>>> stws = nltk.corpus.stopwords.words('english')
>>> tokens = [t for t in tokens if t.isalpha() and t not in stws]
```



```
['google', 'headquartered', 'mountain', 'view', 'amphitheatre', 'pkwy', 'mountain', 'view', 'ca', 'unveiled',
'new', 'android', 'phone', 'consumer', 'electronic', 'show', 'sundar', 'pichai', 'said', 'keynote', 'users',
'love', 'new', 'android', 'phones']
```

One way to perform normalization is **stemming**, which chops or replaces word tails (e.g., removing "s") with the goal of approximate the word's original form.

```
['google', 'headquartered', 'mountain', 'view', 'amphitheatre', 'pkwy', 'mountain', 'view', 'ca', 'unveiled',
 'new', 'android', 'phone', 'consumer', 'electronic', 'show', 'sundar', 'pichai', 'said', 'keynote', 'users',
 'love', 'new', 'android', 'phones']
```

```
>>> stemmer = nltk.stem.porter.PorterStemmer()
>>> clean_tokens = [stemmer.stem(t) for t in tokens]
```

```
['googl', 'headquart', 'mountain', 'view', 'amphitheatr', 'pkwi', 'mountain', 'view', 'ca', 'unveil', 'new',
 'android', 'phone', 'consum', 'electron', 'show', 'sundar', 'pichai', 'said', 'keynot', 'user', 'love', 'new',
 'android', 'phone']
```



Another way to perform normalization is [lemmatization](#), which uses dictionaries and full morphological analysis to correctly identify the lemma (i.e., base form) for each word.

```
['google', 'headquartered', 'mountain', 'view', 'amphitheatre', 'pkwy', 'mountain', 'view', 'ca', 'unveiled',
 'new', 'android', 'phone', 'consumer', 'electronic', 'show', 'sundar', 'pichai', 'said', 'keynote', 'users',
 'love', 'new', 'android', 'phones']
```

```
>>> from nltk.corpus import wordnet
>>> lemmatizer = nltk.stem.WordNetLemmatizer()
>>> pos = [wordnet_pos(p) for p in nltk.pos_tag(tokens)]
>>> clean_tokens = [lemmatizer.lemmatize(t,p) for t, p in pos]
```

```
['google', 'headquarter', 'mountain', 'view', 'amphitheatre', 'pkwy', 'mountain', 'view', 'ca', 'unveil',
 'new', 'android', 'phone', 'consumer', 'electronic', 'show', 'sundar', 'pichai', 'say', 'keynote', 'user',
 'love', 'new', 'android', 'phone']
```

To perform lemmatization appropriately, we need POS (Part Of Speech) tagging, which means labeling the role of each word in a particular part of speech.

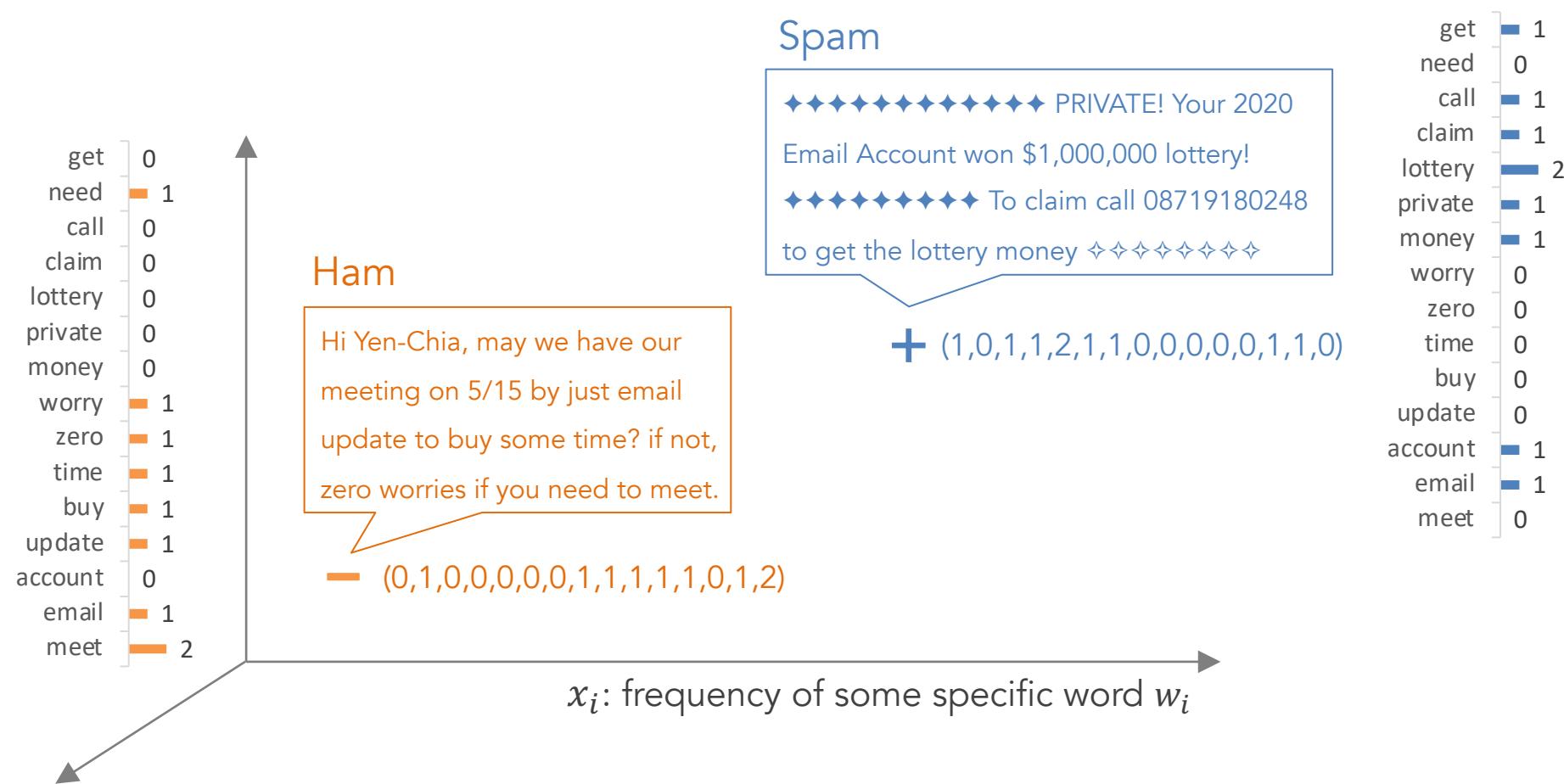
	nsubj	p		vmod	prep	nn	pobj	p	num	nn	appos	p
Google	,		headquartered	in	Mountain	View	(1600	Amphitheatre	Pkwy	,	
NOUN	PUNCT		VERB	ADP	NOUN	NOUN	PUNCT	NUM	NOUN	NOUN	PUNCT	

nn	appos	p	appos	num	p	p	root	det	amod	nn	dobj	prep	pobj	prep	det	nn	nn	pobj	p
Mountain	View	,	CA	940430)	,	unveiled	the	new	Android	phone	for	\$799	at	the	Consumer	Electronic	Show	.
NOUN	NOUN	PUNCT	NOUN	NUM	PUNCT	PUNCT	VERB	DET	ADJ	NOUN	NOUN	ADP	NUM	ADP	DET	NOUN	NOUN	NOUN	PUNCT

nn	nsubj	root	prep	poss	pobj	mark	nsubj	ccomp	poss	amod	nn	dobj
Sundar	Pichai	said	in	his	keynote	that	users	love	their	new	Android	phones
NOUN	NOUN	VERB	ADP	PRON	NOUN	ADP	NOUN	VERB	PRON	ADJ	NOUN	NOUN

```
>>> from nltk.corpus import wordnet
>>> def wordnet_pos(nltk_pos):
...     if nltk_pos[1].startswith('V'): return (nltk_pos[0], wordnet.VERB)
...     if nltk_pos[1].startswith('J'): return (nltk_pos[0], wordnet.ADJ)
...     if nltk_pos[1].startswith('R'): return (nltk_pos[0], wordnet.ADV)
...     else: return (nltk_pos[0], wordnet.NOUN)
```

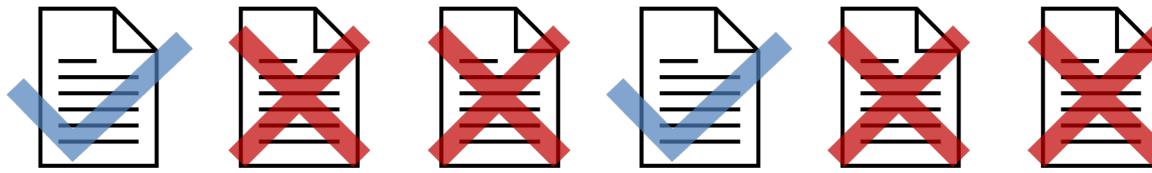
Now we have the cleaned tokens that represent a sentence. We need to transform them to data points in some high-dimensional space. One example is Bag of Words.



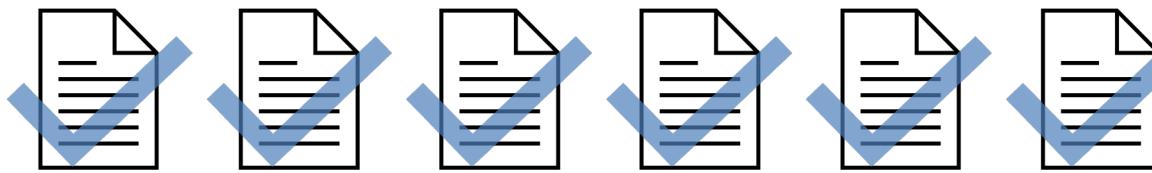
These data points are also called **vectors**, which means arrays of numbers that encode both the direction and length information.



The Bag of Words approach can be problematic since it weights all words equally, even after removing stop words. For example, "play" can appear many times in sports news.



If a word appears in only a few documents (and frequently in these documents), it contains more information and should be more important.

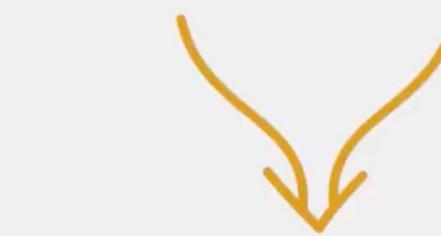


If a word appears in almost all documents, it should be less important, since seeing this word does not give us much information.

So, we can use TF-IDF (term frequency-inverse document frequency) to transform sentences or documents into vectors. Intuitively, TF-IDF means weighted Bag of Words.

Final TF-IDF score for a term in a document

$$w_{t,d} = \underbrace{\text{tf}(t, d)}_{\text{The more frequently a term appears in a given document...}} \times \underbrace{\text{idf}(t, D)}_{\dots \text{and the fewer times it appears in other documents...}}$$



The higher its TF-IDF value.

Term Frequency (TF) measures how frequently a term (word) appears in a document.

There are different implementations, such as using a log function to scale it down.

Term Frequency (TF)

$$\text{tf}(t, d) = f_{t,d}$$

Given a word(**t**) in a document(**d**)...

The *term frequency* is just how many times the term occurs in the document.

Alternative Implementation: $\text{tf}(t, d) = \log_{10}(f_{t,d} + 1)$

Inverse Document Frequency (IDF) weights each word by considering how frequently it shows in different documents. IDF is higher when the term appears in fewer documents.

Inverse Document Frequency (IDF)

$$\text{idf}(t, D) = \log_{10} \left(\frac{N}{n_t} \right)$$

Given a term(t) and a **corpus**(D)...

We take the log here as well.

N is the number of documents.

n_t is the number of documents t appears in.

We can also use **topic modeling** to encode a sentence/document into a distribution of topics. Below is an intuition of how the Latent Dirichlet Allocation method works.

Topic Vectors

gene	0.04
dna	0.02
genetic	0.01
...	

life	0.02
evolve	0.01
organism	0.01
...	

brain	0.04
neuron	0.02
nerve	0.01
...	

data	0.02
number	0.02
computer	0.01
...	

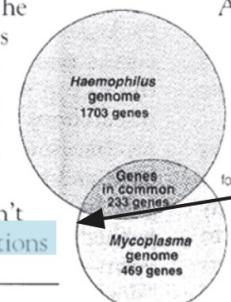
Documents

Seeking Life's Bare (Genetic) Necessities

COLD SPRING HARBOR, NEW YORK—How many **genes** does an **organism** need to **survive**? Last week at the genome meeting here,* two genome researchers with radically different approaches presented complementary views of the basic genes needed for **life**. One research team, using **computer analyses** to compare known **genomes**, concluded that today's **organisms** can be sustained with just 250 genes, and that the earliest life forms required a mere 128 **genes**. The other researcher mapped genes in a simple parasite and estimated that for this organism, 800 genes are plenty to do the job—but that anything short of 100 wouldn't be enough.

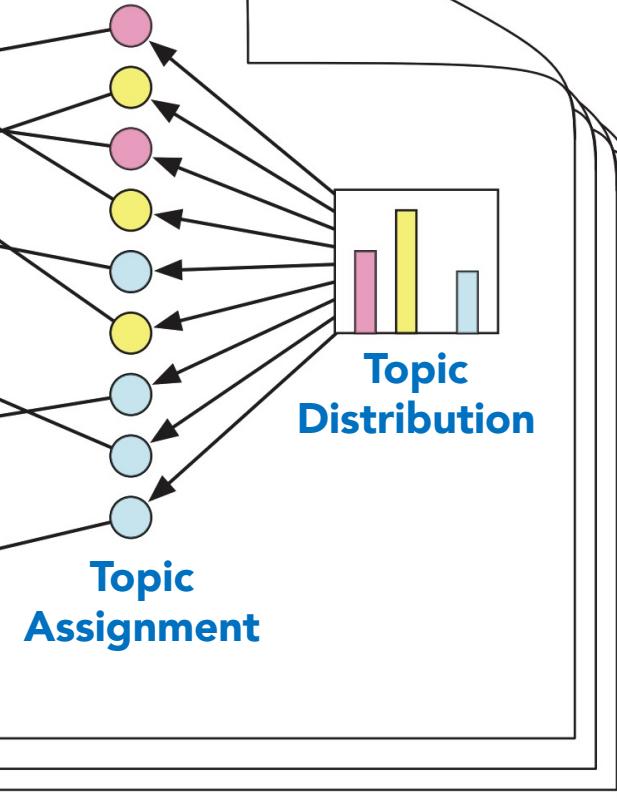
Although the numbers don't match precisely, those predictions

"are not all that far apart," especially in comparison to the 75,000 **genes** in the **human genome**, notes Siv Andersson of Uppsala University in Sweden, who arrived at the 800 number. But coming up with a consensus answer may be more than just a **genetic numbers game**, particularly as more and more **genomes** are completely mapped and sequenced. "It may be a way of organizing any newly **sequenced genome**," explains Arcady Mushegian, a **computational molecular biologist** at the National Center for Biotechnology Information (NCBI) in Bethesda, Maryland. Comparing an

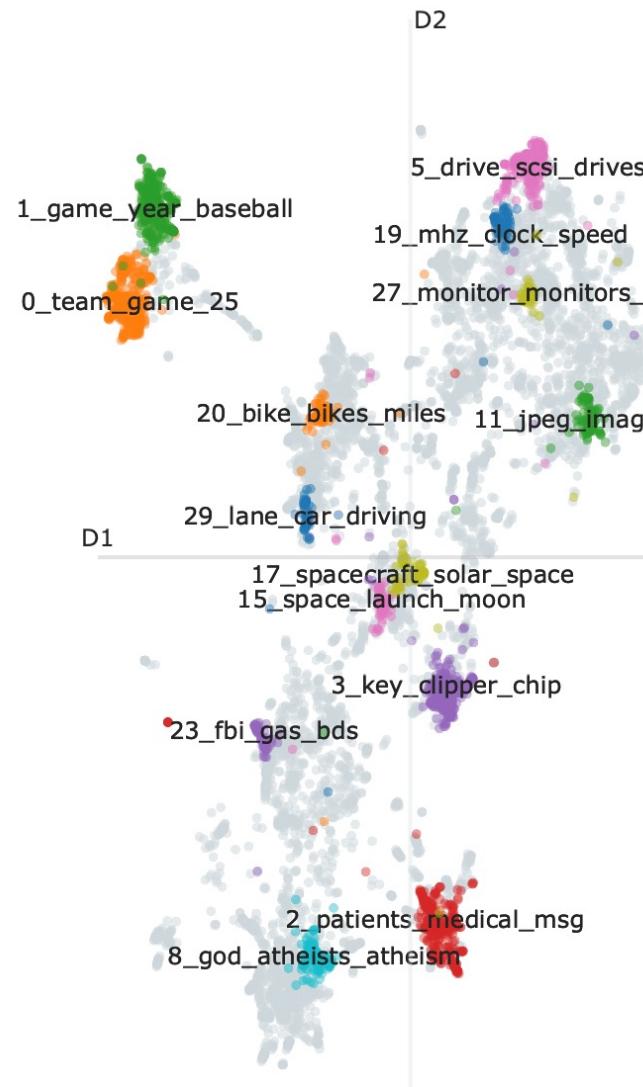


Stripping down. Computer analysis yields an estimate of the minimum modern and ancient genomes.

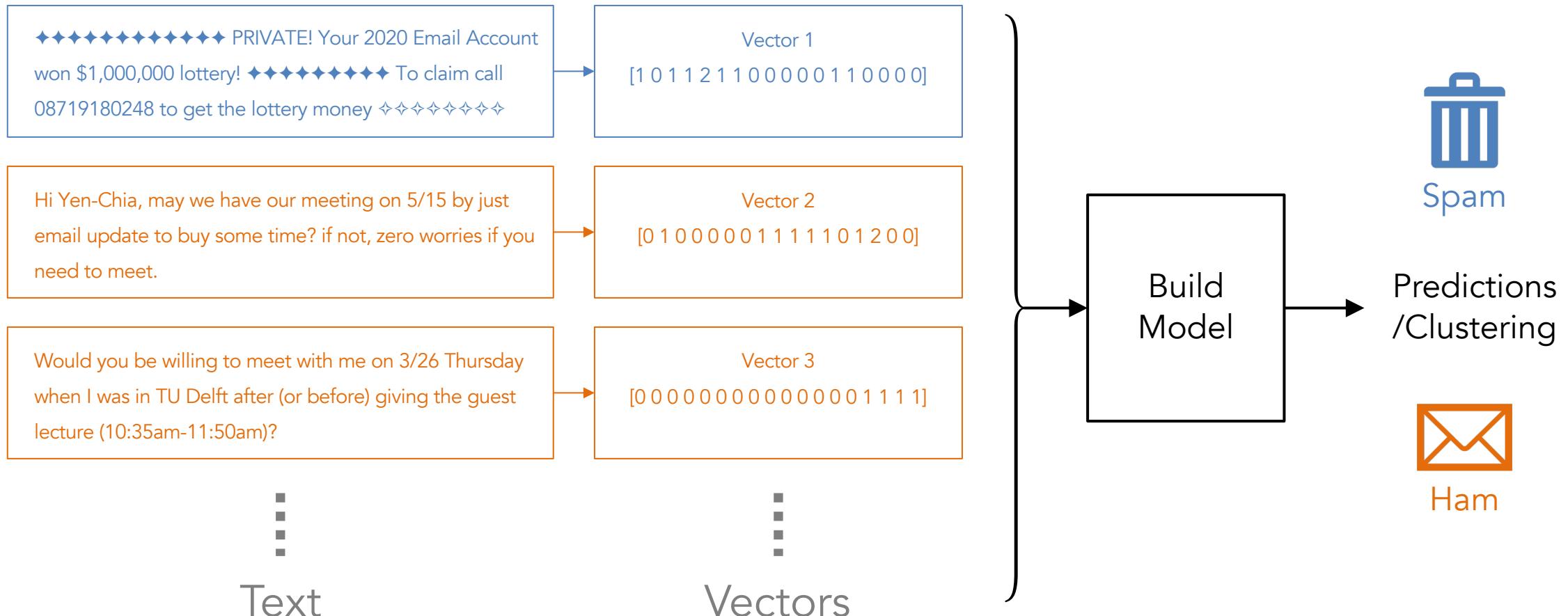
* Genome Mapping and Sequencing, Cold Spring Harbor, New York, May 8 to 12.



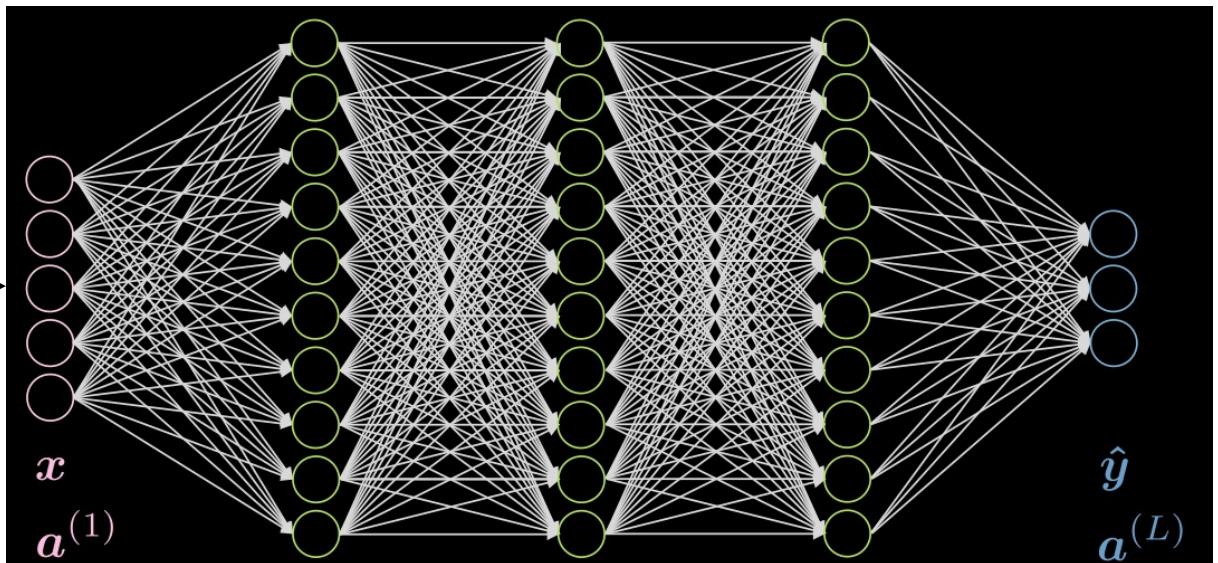
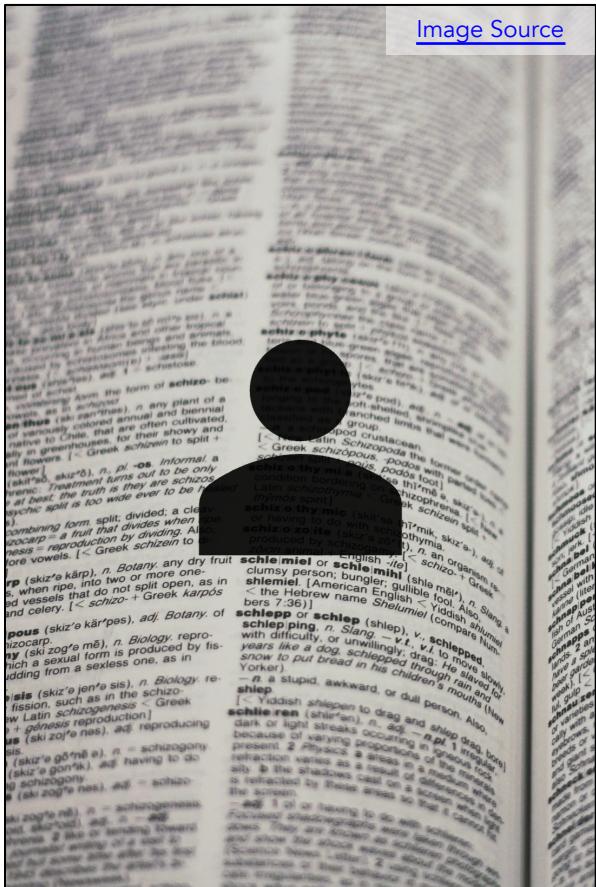
Each topic vector is represented by a list of words with different weights.



After transforming text into vectors, we can use these vectors for national language processing tasks, such as sentence/document classification (or clustering).



We have seen the approach of crafting features manually. But we can use deep learning to automate feature engineering. What should be the input vectors in this case?



We can use one-hot encoding. But this approach is inefficient (in terms of computation) because it creates long vectors with many zeros, which uses a lot of computer memory.

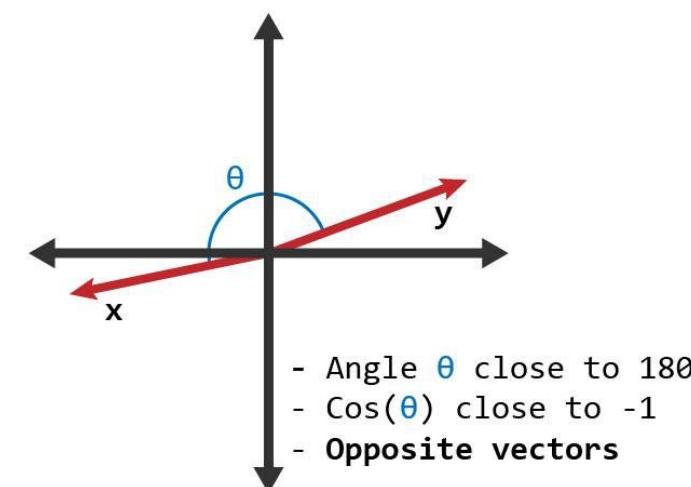
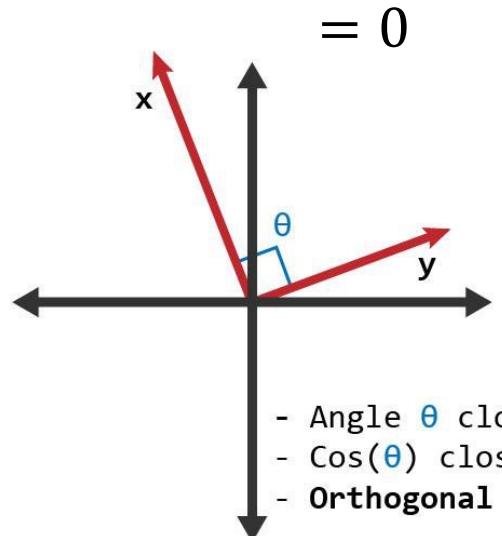
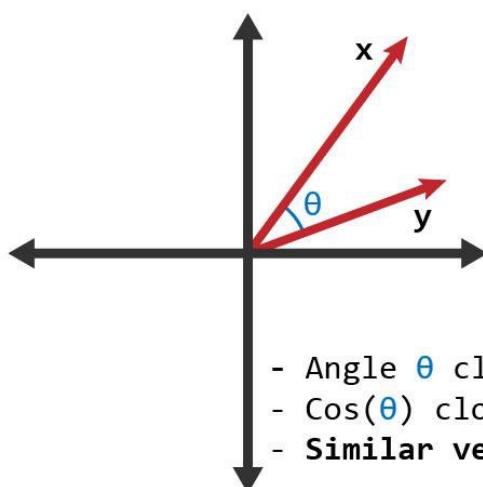
One-hot encoding

		cat	mat	on	sat	the	All other possible words	
w_{the}	the	=>	0	0	0	0	1	0 0 0 0 0 0 0 ... 0
w_{cat}	cat	=>	1	0	0	0	0	0 0 0 0 0 0 0 ... 0
w_{sat}	sat	=>	0	0	0	1	0	0 0 0 0 0 0 0 ... 0
					

Another problem of **one-hot encoding** is that it does not encode similarity. For example, the **cosine similarity** between two one-hot encoded vectors are always zero.

$$\text{similarity}(w_{cat}, w_{sat}) = \cos(\theta) = \frac{\langle w_{cat} \cdot w_{sat} \rangle}{\|w_{cat}\| \|w_{sat}\|} = [1 \ 0 \ 0 \ 0 \ 0] \cdot \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

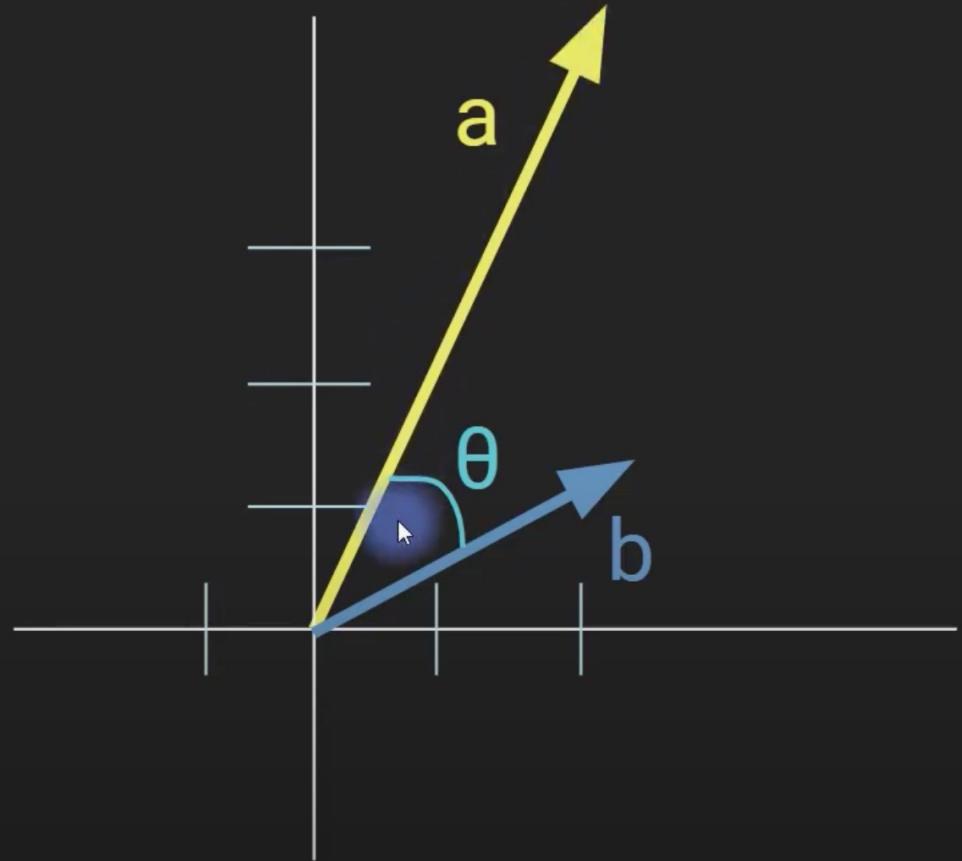
dot product = $w_{cat}^T w_{sat}$



The dot product of two vectors can also be used to measure similarity, which considers both the angle and the vector lengths. Cosine similarity is just a normalized dot product.

Magnitudes of vectors
scaled by angle between them

$$a = |a||b|\cos(\theta_{ab})$$



We can use **word embeddings** to efficiently represent text as vectors, in which **similar words have a similar encoding** in a high-dimensional space.

A 4-dimensional embedding

cat =>

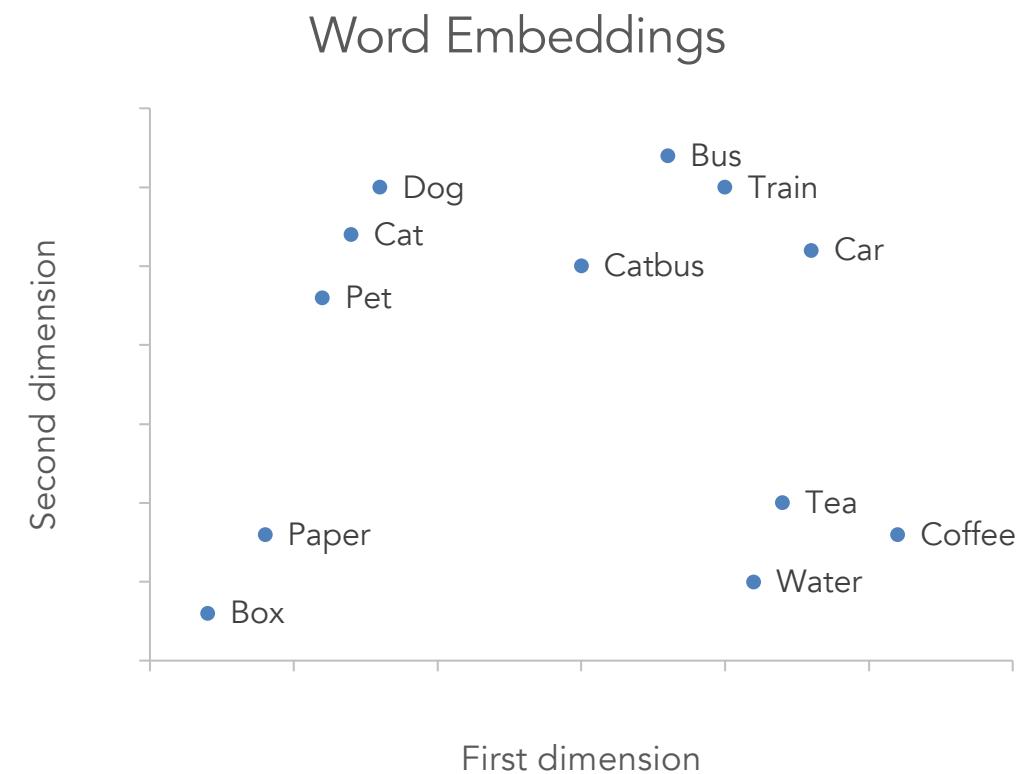
1.2	-0.1	4.3	3.2
0.4	2.5	-0.9	0.5
2.1	0.3	0.1	0.4

mat =>

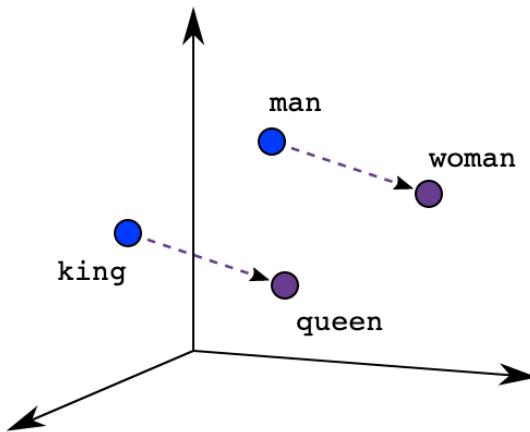
on =>

...

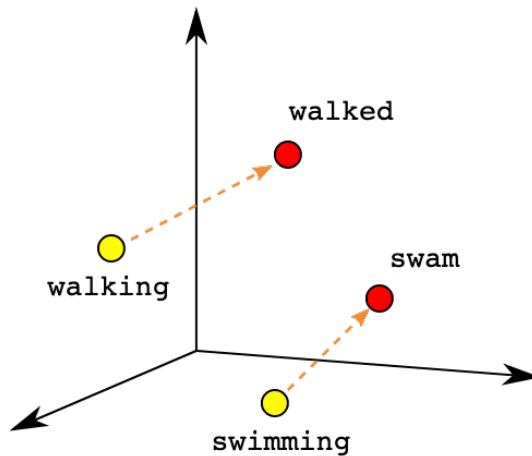
...



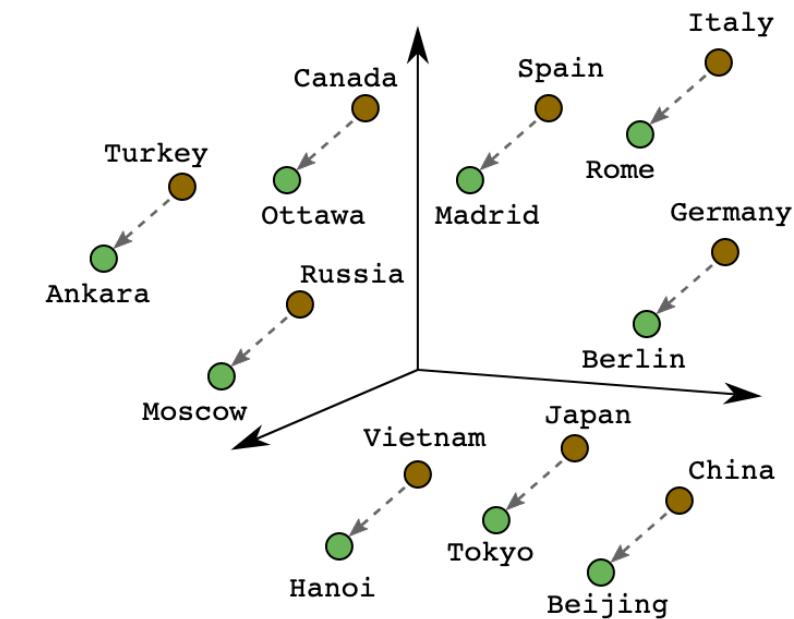
Position (e.g., distance and direction) in the word embedding vector space can **encode semantic relations**, such as the relation between a country and its capital.



Male-Female

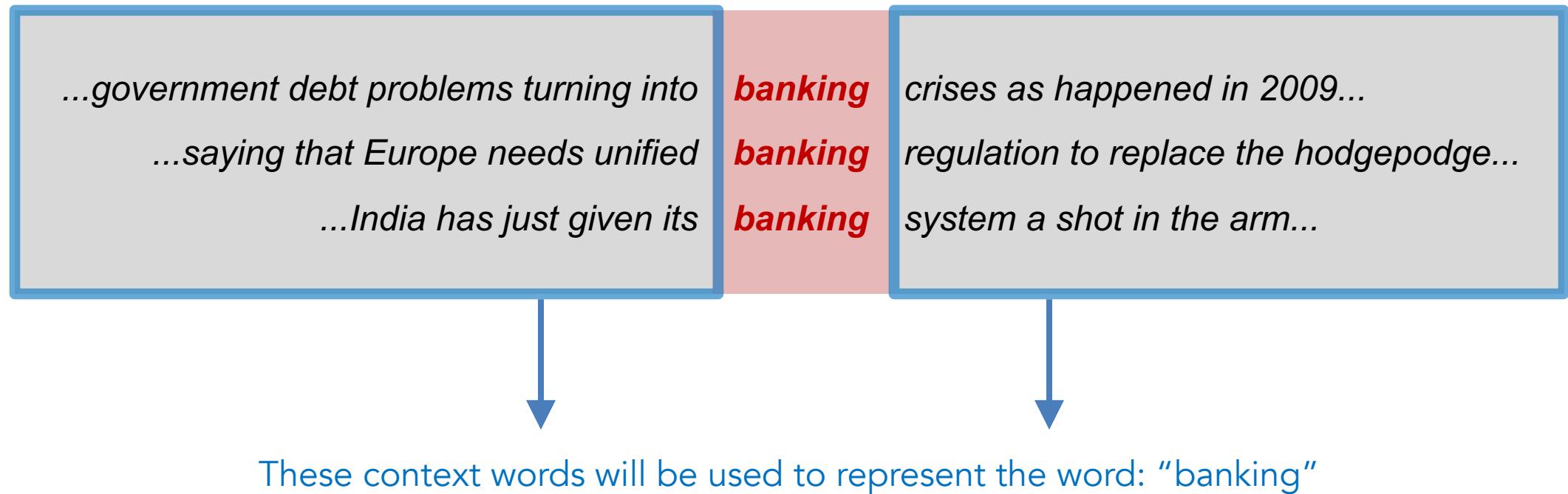


Verb Tense



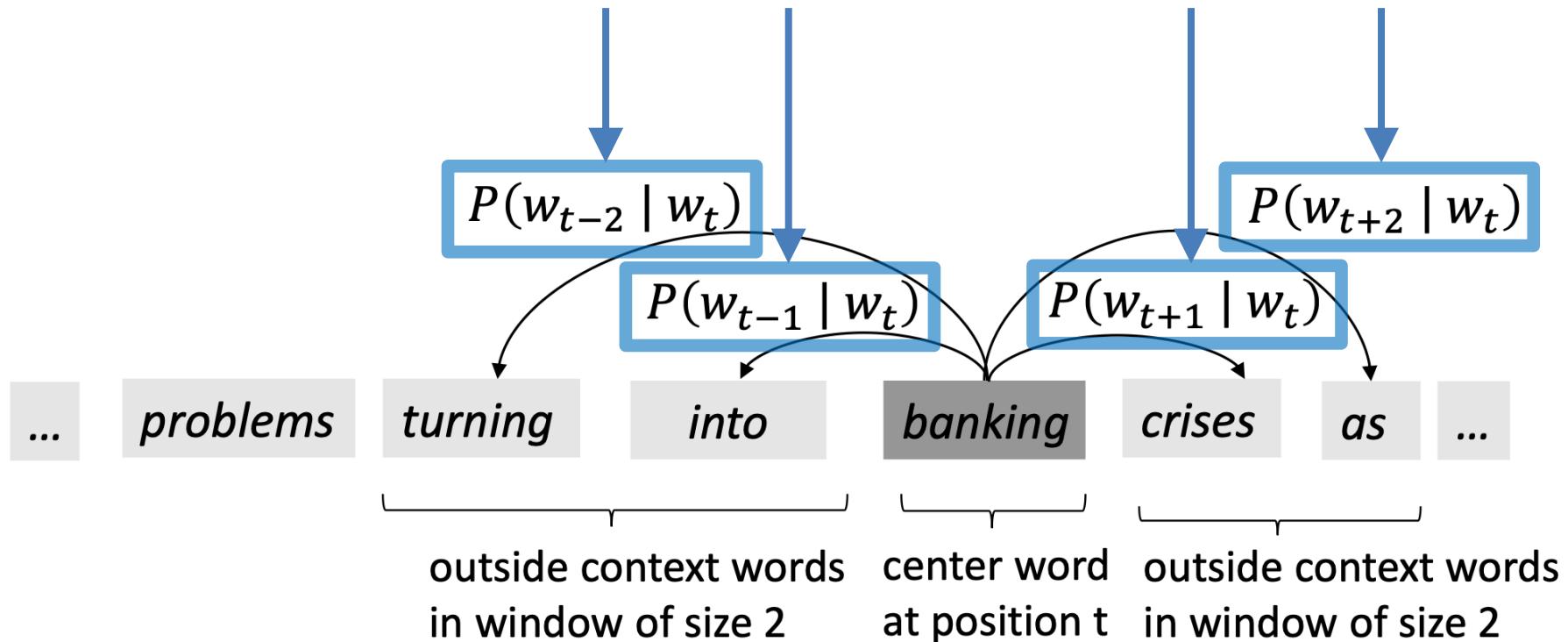
Country-Capital

But how can we train the word embeddings? Intuitively, we can represent words by their **context** (i.e., the nearby words within a fixed-size window).

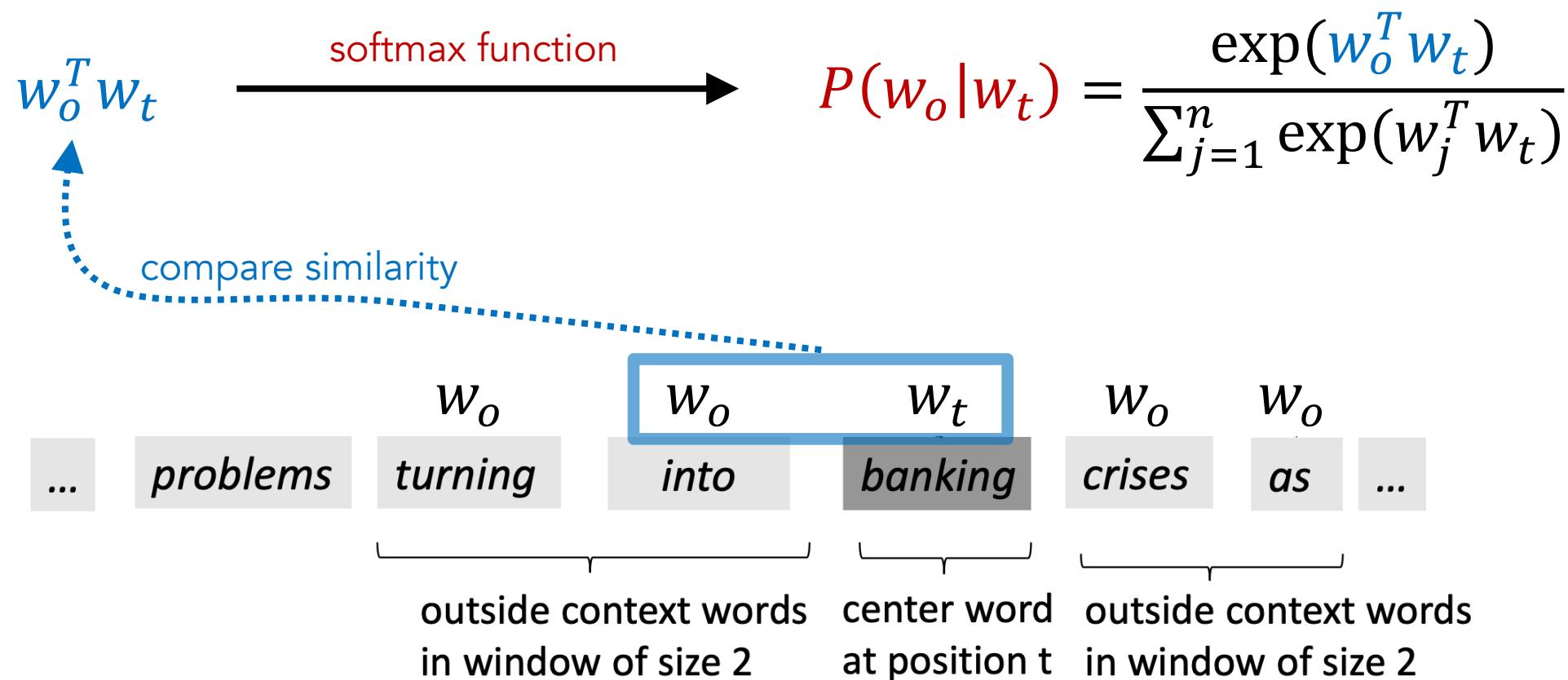


[Word2Vec](#) is a method to train word embeddings by context. The goal is to use the center word to predict nearby words as accurate as possible, based on probabilities.

We need to maximize these probabilities for all words in the text body



How is probability related to word vectors? We use the **dot product similarity** of word vectors to calculate **probabilities**, with the help of the **softmax function**.



The softmax function maps any arbitrary values to a probability distribution.

Larger dot product means larger probability

$$w_o^T w_t \xrightarrow{\text{softmax function}} P(w_o | w_t) = \frac{\exp(w_o^T w_t)}{\sum_{j=1}^n \exp(w_j^T w_t)}$$

The entire denominator is just for normalization

$$x_i \xrightarrow{\text{softmax function}} P(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^n \exp(x_j)}$$

The exponential function makes things positive: $\exp(x) = e^x$

Below is an example of how the softmax function maps numbers to probabilities.

$$\begin{bmatrix} 2 \\ 1 \\ 0.1 \end{bmatrix} \xrightarrow{\text{softmax function}} \begin{bmatrix} 0.7 \\ 0.2 \\ 0.1 \end{bmatrix}$$

Diagram illustrating the softmax function mapping:

- Input vector: $\begin{bmatrix} 2 \\ 1 \\ 0.1 \end{bmatrix}$
- Output probabilities: $\begin{bmatrix} 0.7 \\ 0.2 \\ 0.1 \end{bmatrix}$
- Intermediate steps:
 - Top row: $\frac{\exp(2)}{\exp(2) + \exp(1) + \exp(0.1)}$ (blue)
 - Middle row: $\frac{\exp(1)}{\exp(2) + \exp(1) + \exp(0.1)}$ (red)
 - Bottom row: $\frac{\exp(0.1)}{\exp(2) + \exp(1) + \exp(0.1)}$ (teal)

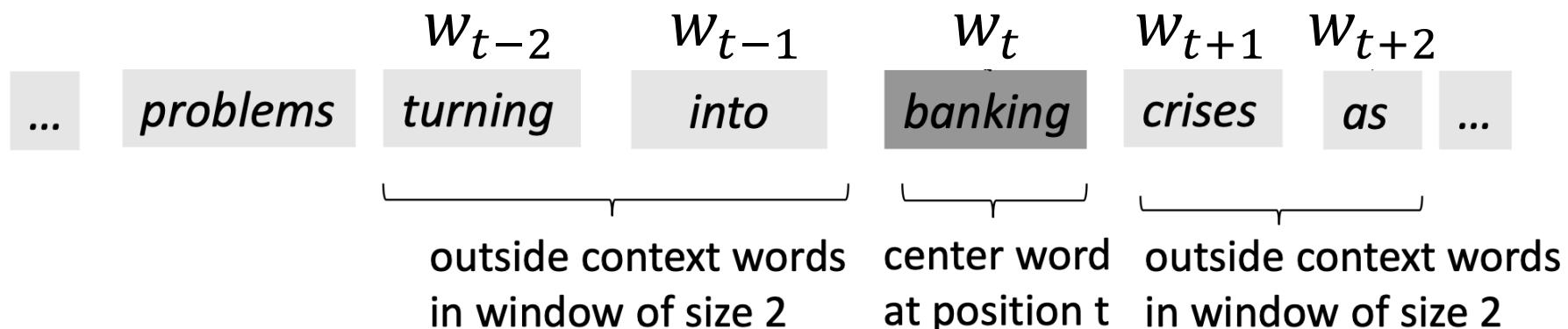
$$x_i \xrightarrow{\text{softmax function}} P(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^n \exp(x_j)}$$

Remember that the denominator is just for normalization

For each word position $t = 1, \dots, T$ with window size m , we can adjust the word vectors to [maximize the likelihood function](#), based on the probability that we calculated.

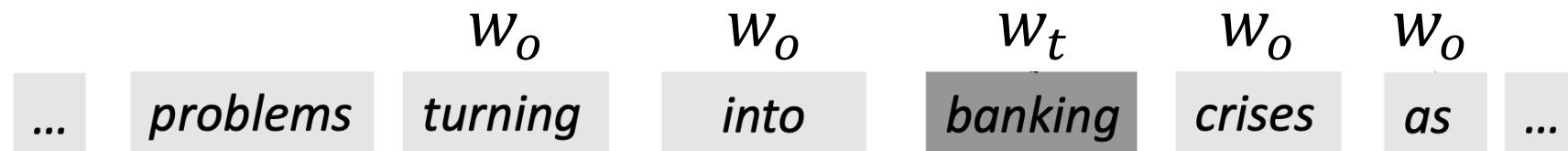
$$\text{Likelihood} = L(\theta) = \prod_{t=1}^T \prod_{\substack{-m \leq j \leq m \\ j \neq 0}} P(w_{t+j} | w_t; \theta)$$

θ is all variables
to be optimized

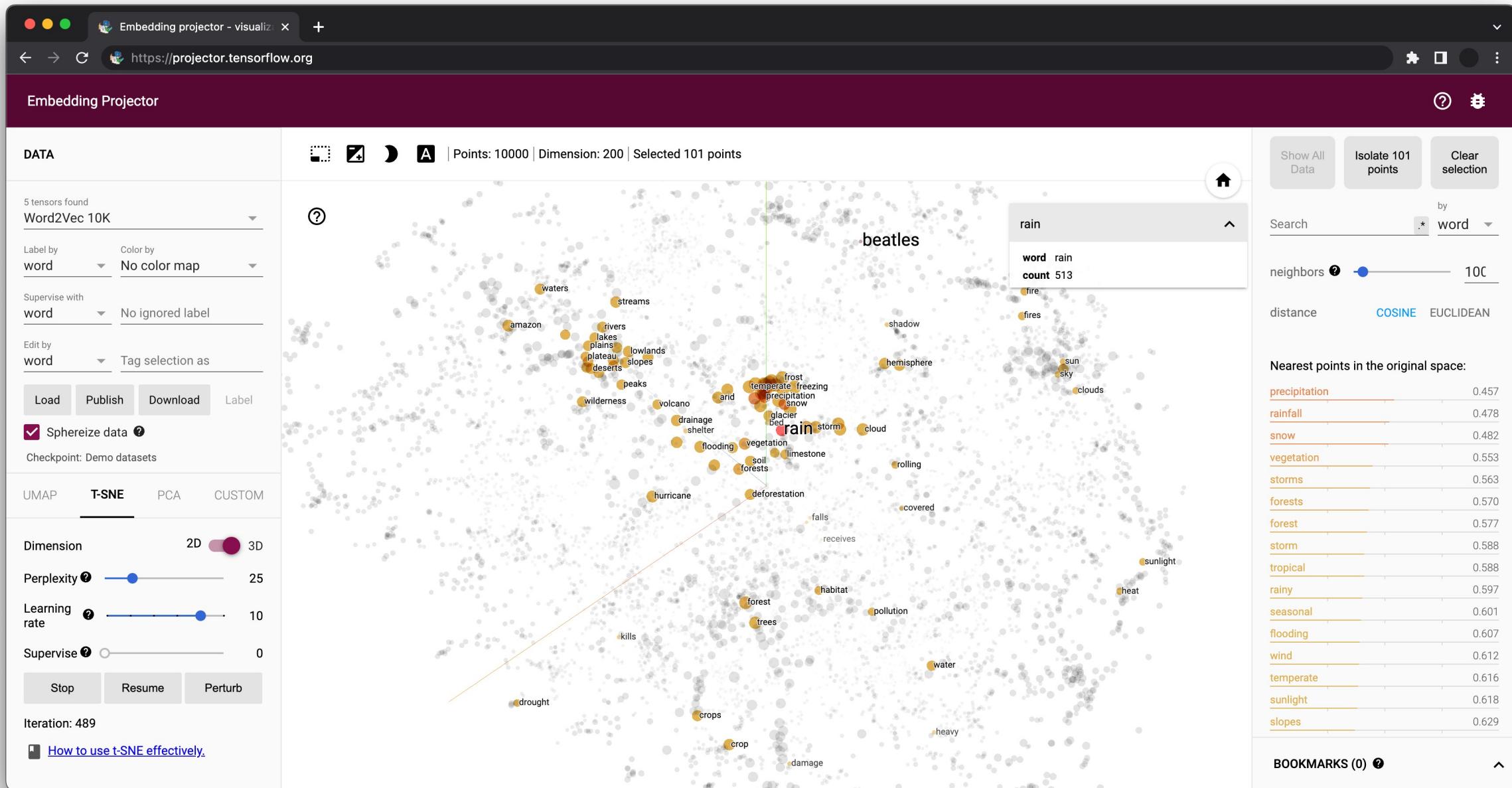


To sum up, below are the high-level ideas for training word embeddings:

- We have a large text corpus (i.e., text body) with a long list of words
- Every word is represented by a vector w
- For each position t in the text, determine the center word w_t and the context words w_o (i.e., the words that are nearby w_t)



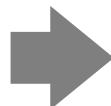
- For each word w_t , compute the probability of $P(w_o|w_t)$ using the dot product similarity of word vectors w_o and w_t
- Keep adjusting the word vectors to maximize this probability



Word embeddings represent words in vectors.
But how to represent **s**entences in vectors?

We can stack all the word vectors into a matrix, where each column means a dimension of the word vector, and the number of rows means sentence length.

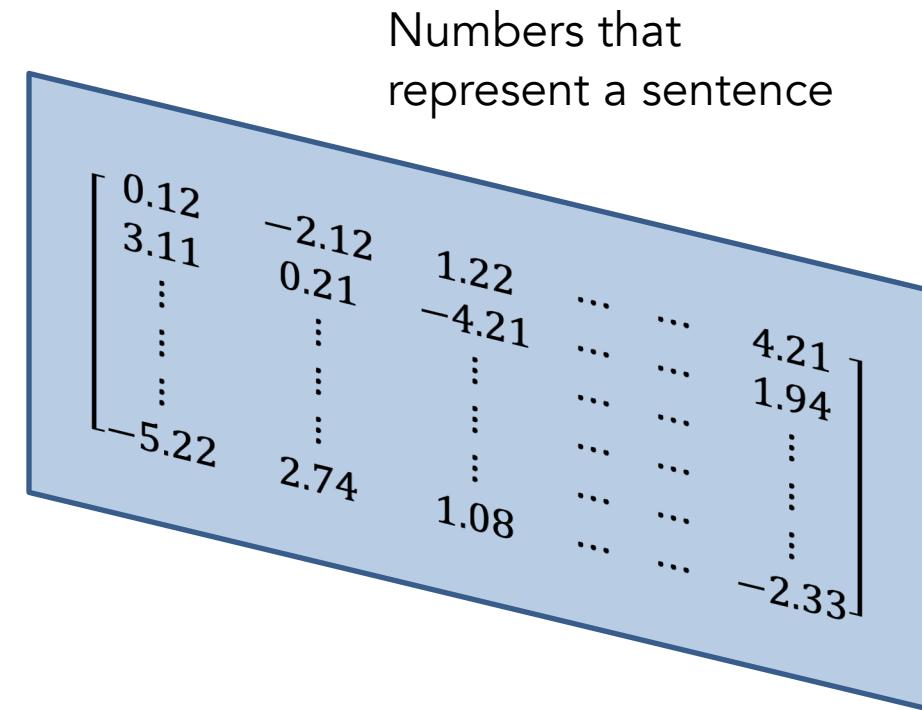
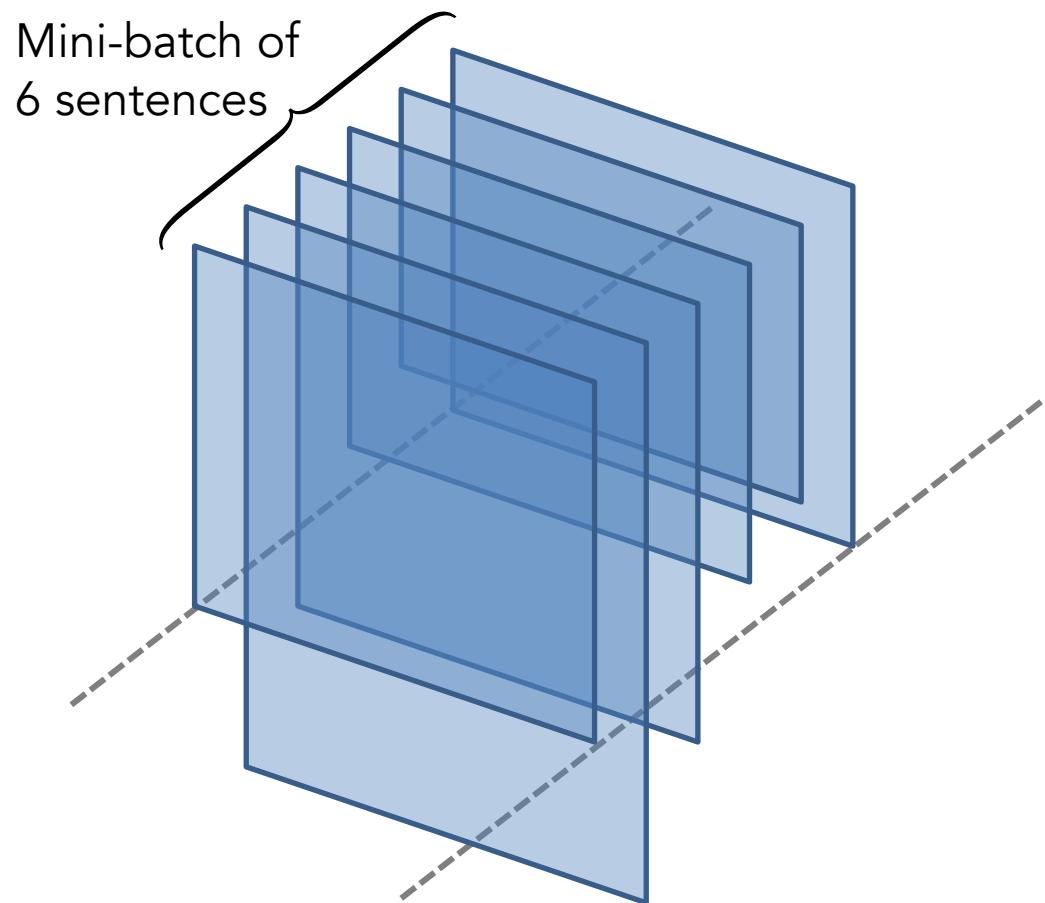
```
['google', 'headquarter',  
 'mountain', 'view',  
 'amphitheatre', 'pkwy',  
 'mountain', 'view', 'ca', 'unveil',  
 'new', 'android', 'phone',  
 'consumer', 'electronic', 'show',  
 'sundar', 'pichai', 'say',  
 'keynote', 'user', 'love', 'new',  
 'android', 'phone']
```



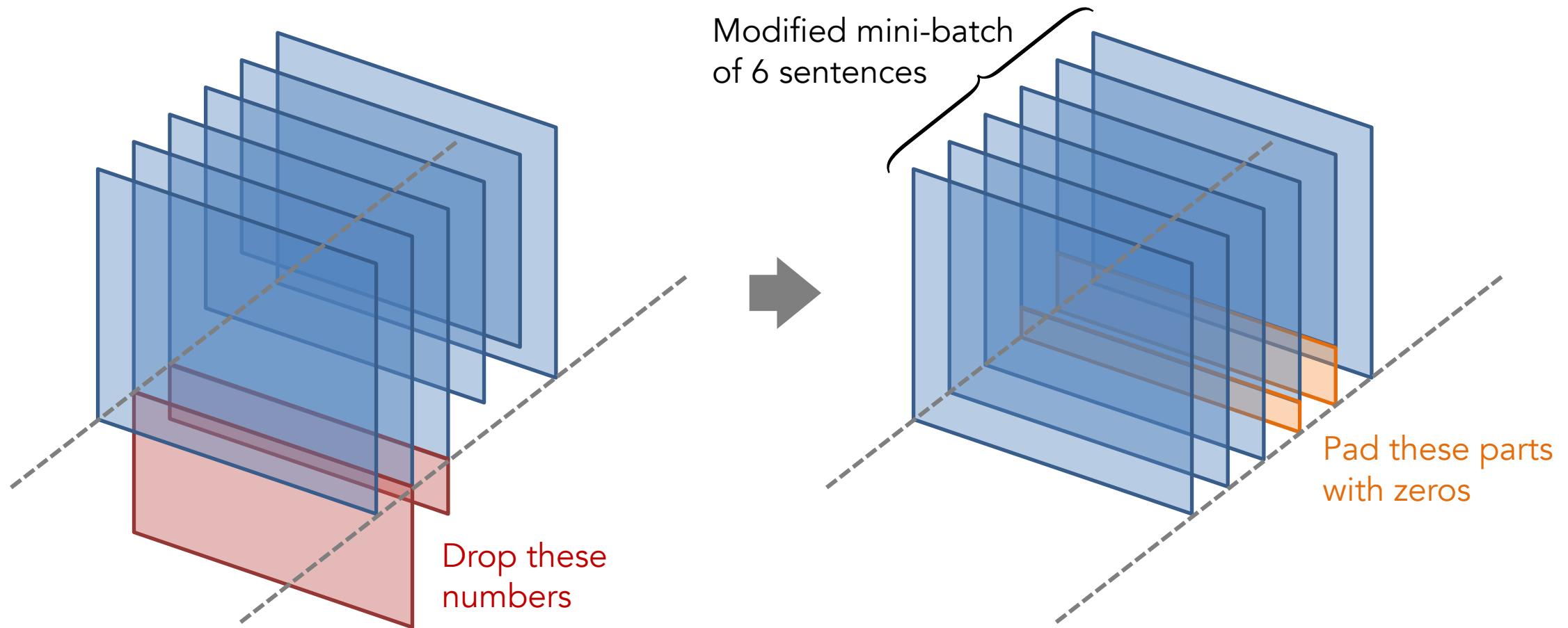
First dimension
of word vector

	0.12	-2.12	1.22	4.21	google
	3.11	0.21	-4.21	1.94	headquarter
:	:	:	:	:	
:	:	:	:	:	
:	:	:	:	:	
	-5.22	2.74	1.08	-2.33	phone

For a deep feedforward network (or convolutional neural network), all inputs need to have the same size. But [sentences can have different length](#). So, what should we do?



We can **drop** the parts that are too long and **pad** the parts that are too short with zeros.



After we make sure that **all input data have the same size**, we can put them into deep neural networks for different tasks, such as sentence/document classification.

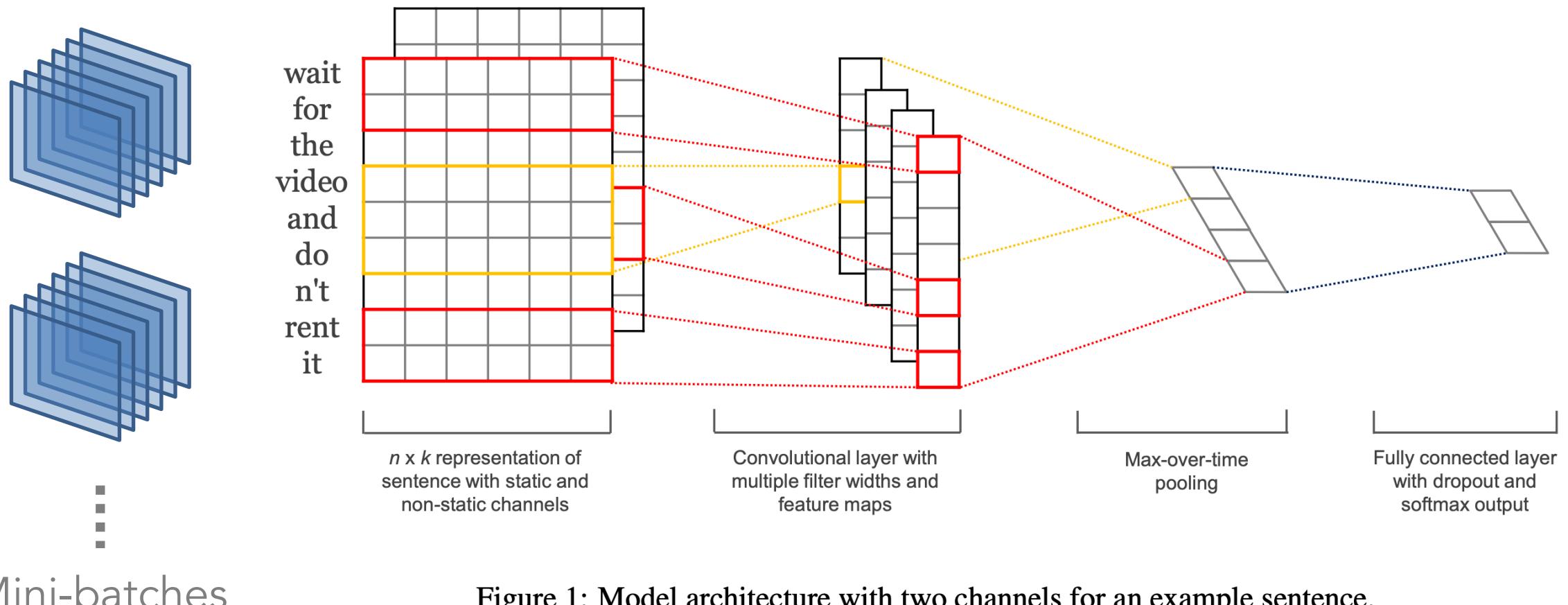
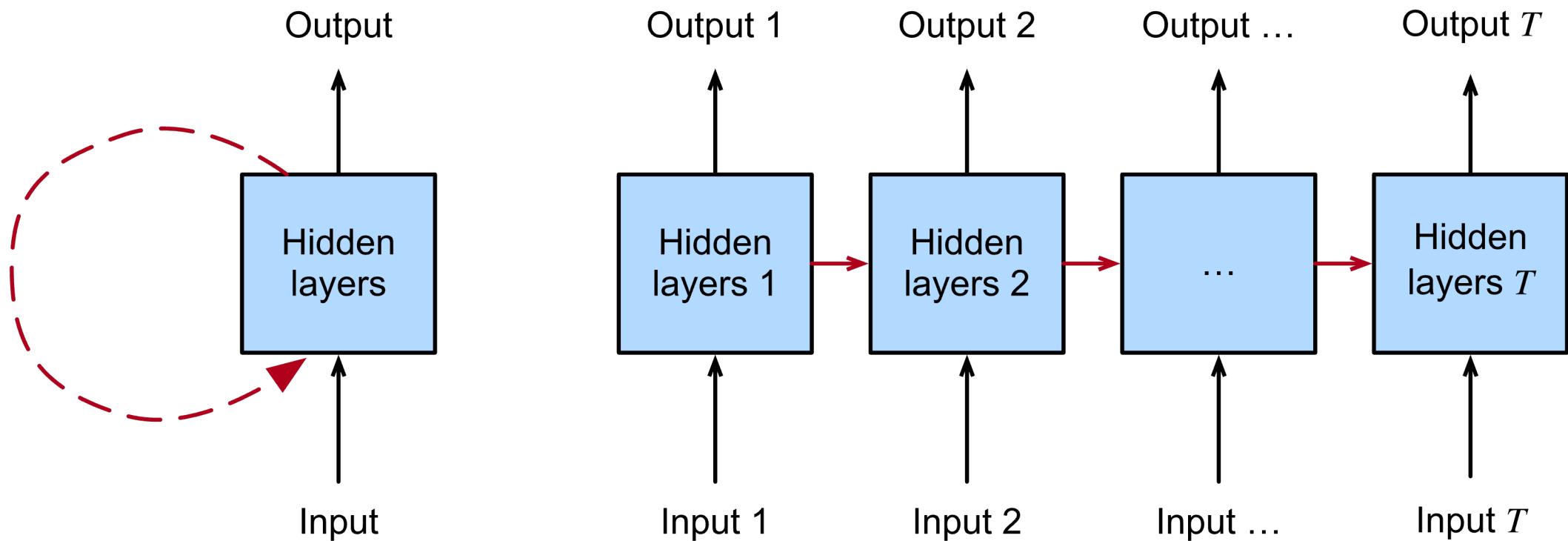


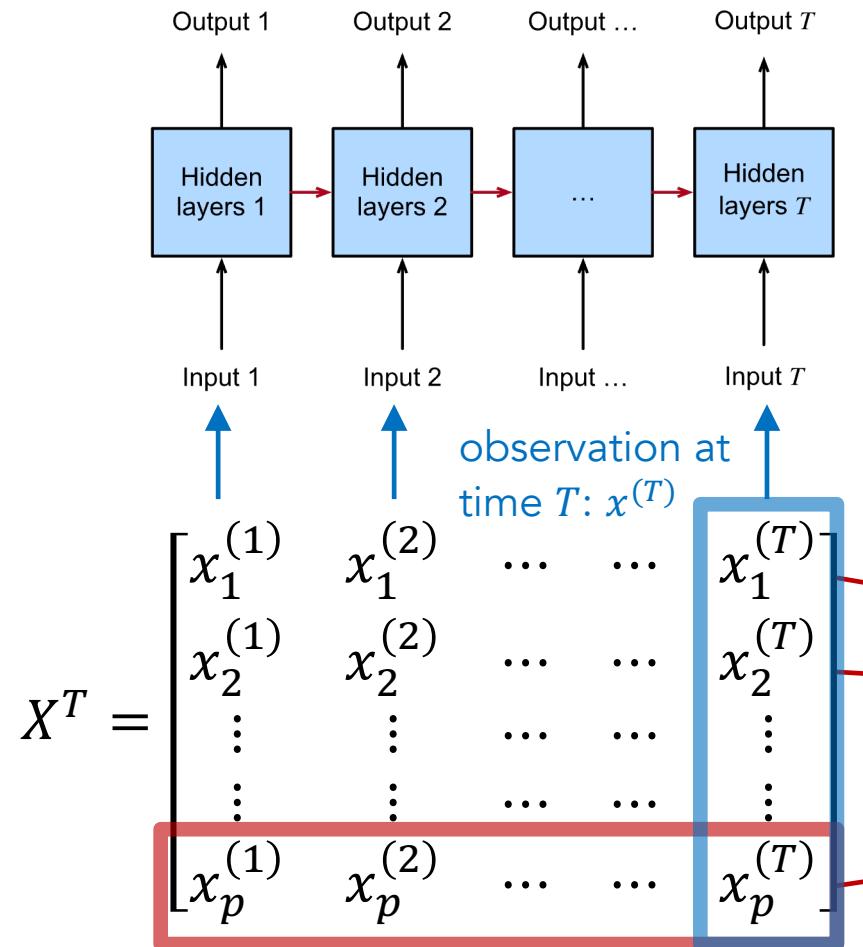
Figure 1: Model architecture with two channels for an example sentence.

We can also use the recurrent neural network (RNN) to takes inputs with various lengths.

Recurrent connections are shown in the red cyclic edges (and unfolded into red arrows).

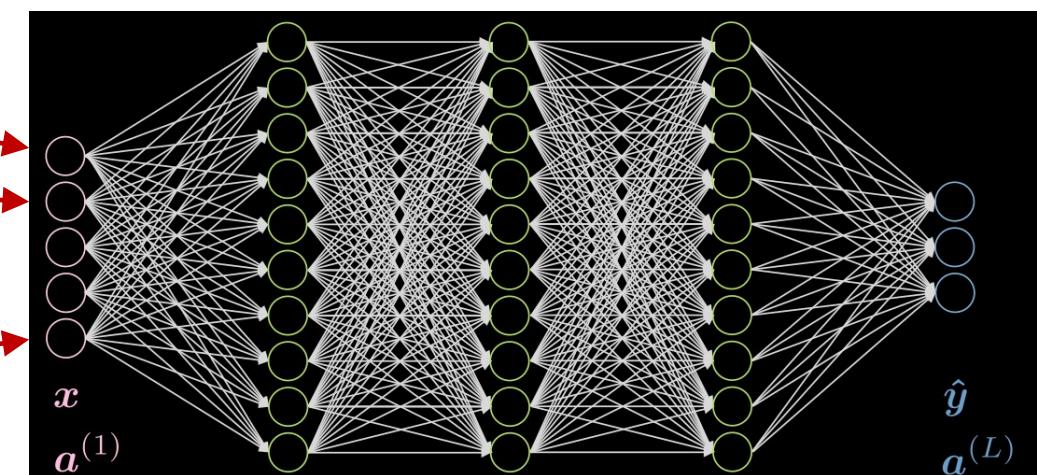


Typically, we **feed features to the deep neural net**, but we **feed observations (for each time step)** to the recurrent neural net. Notice that the input X below is transposed.

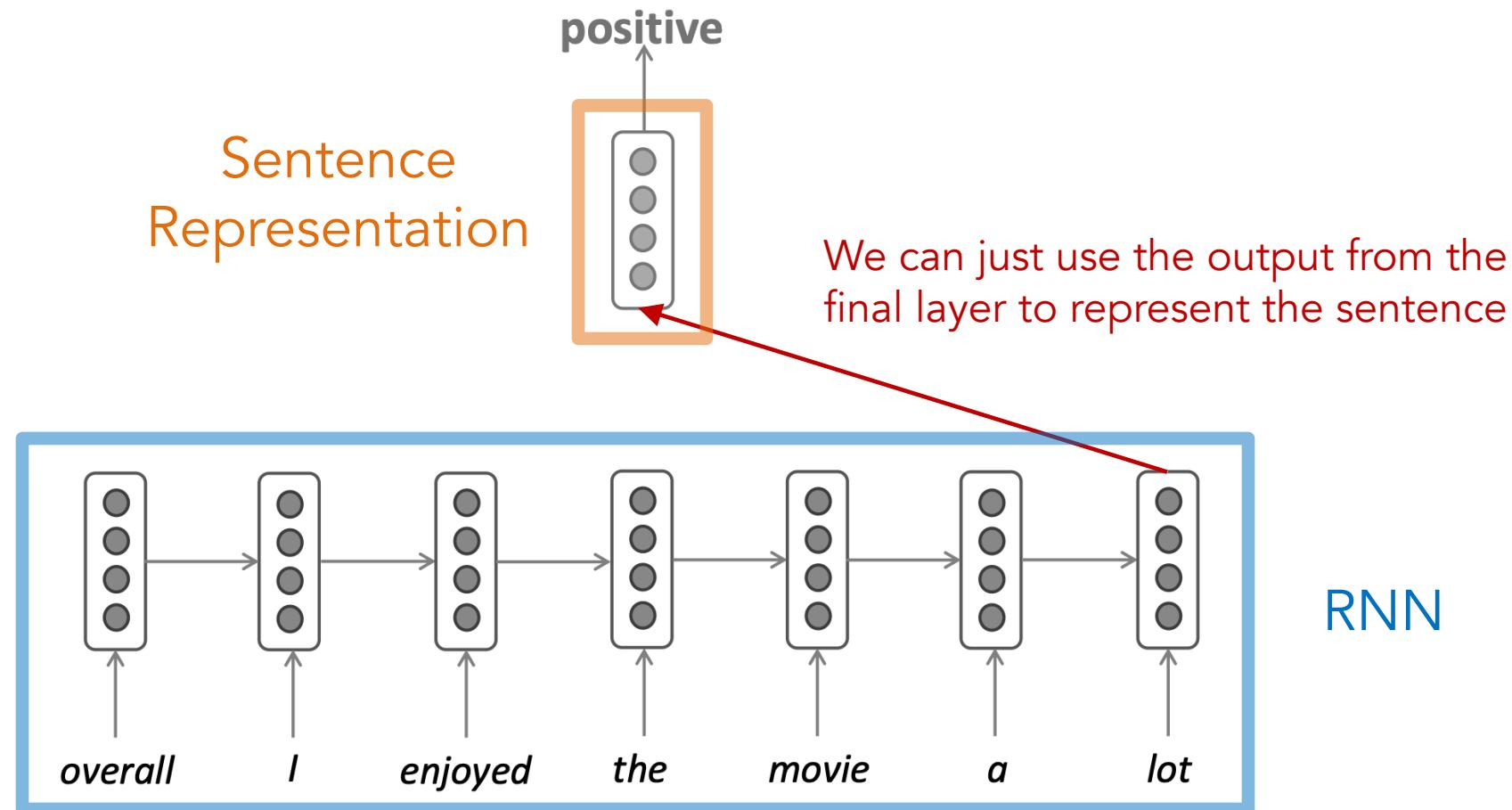


An example for natural language processing:

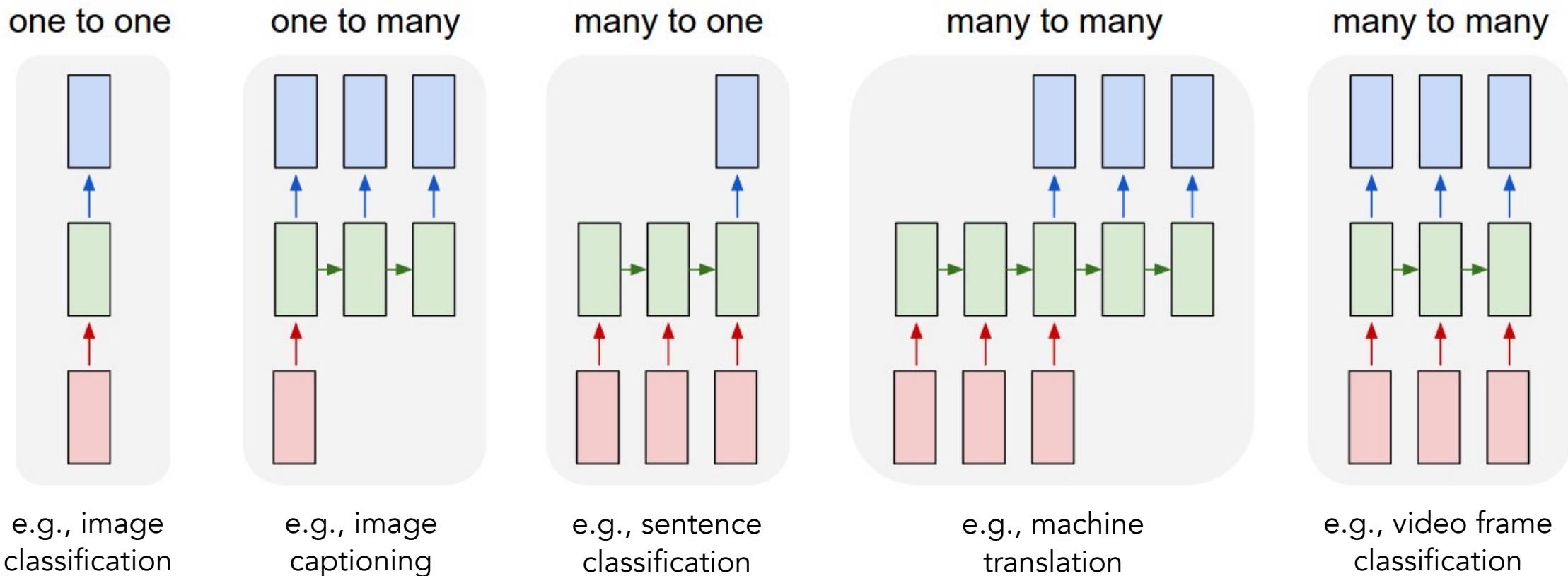
- Feature: the word embedding dimensions
- Observation: the word at position T in a sentence



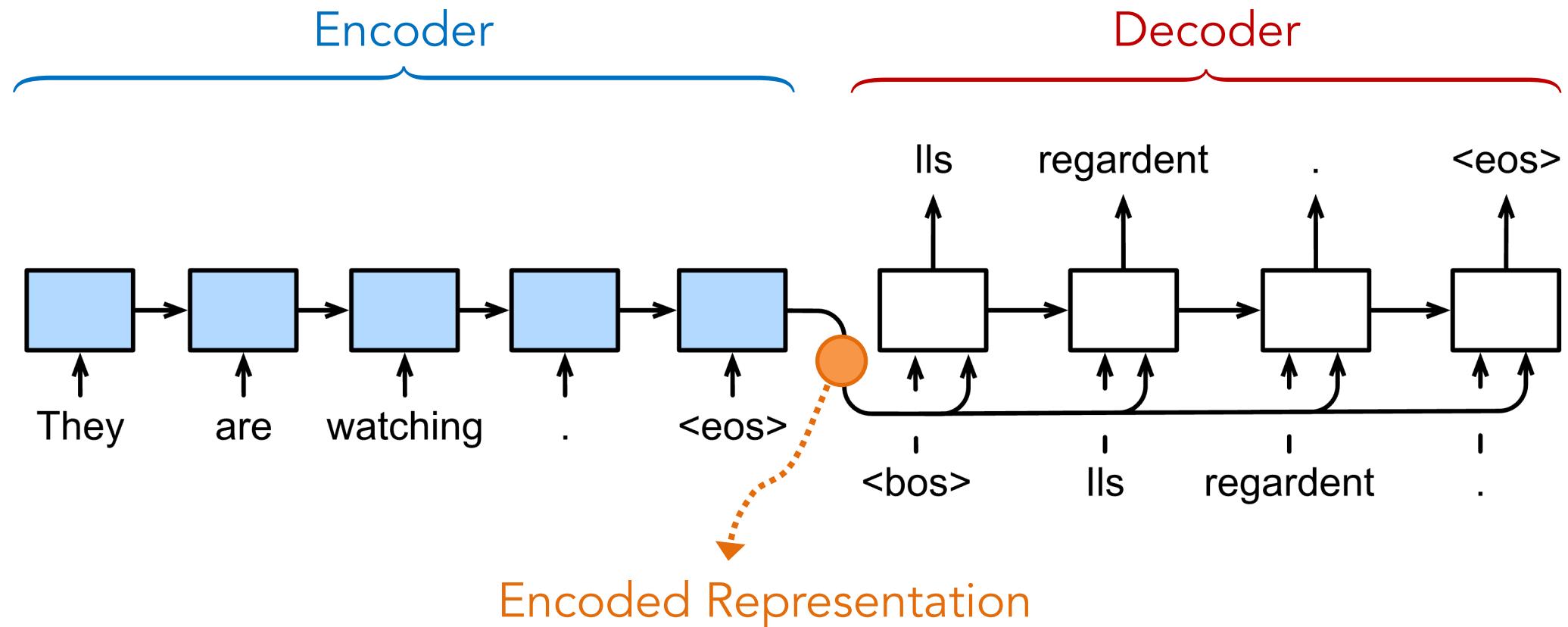
We can combine RNNs into a sequence-to-sequence (Seq2Seq) model for sentence classification or sentiment analysis. In this case, the output sequence has only one label.



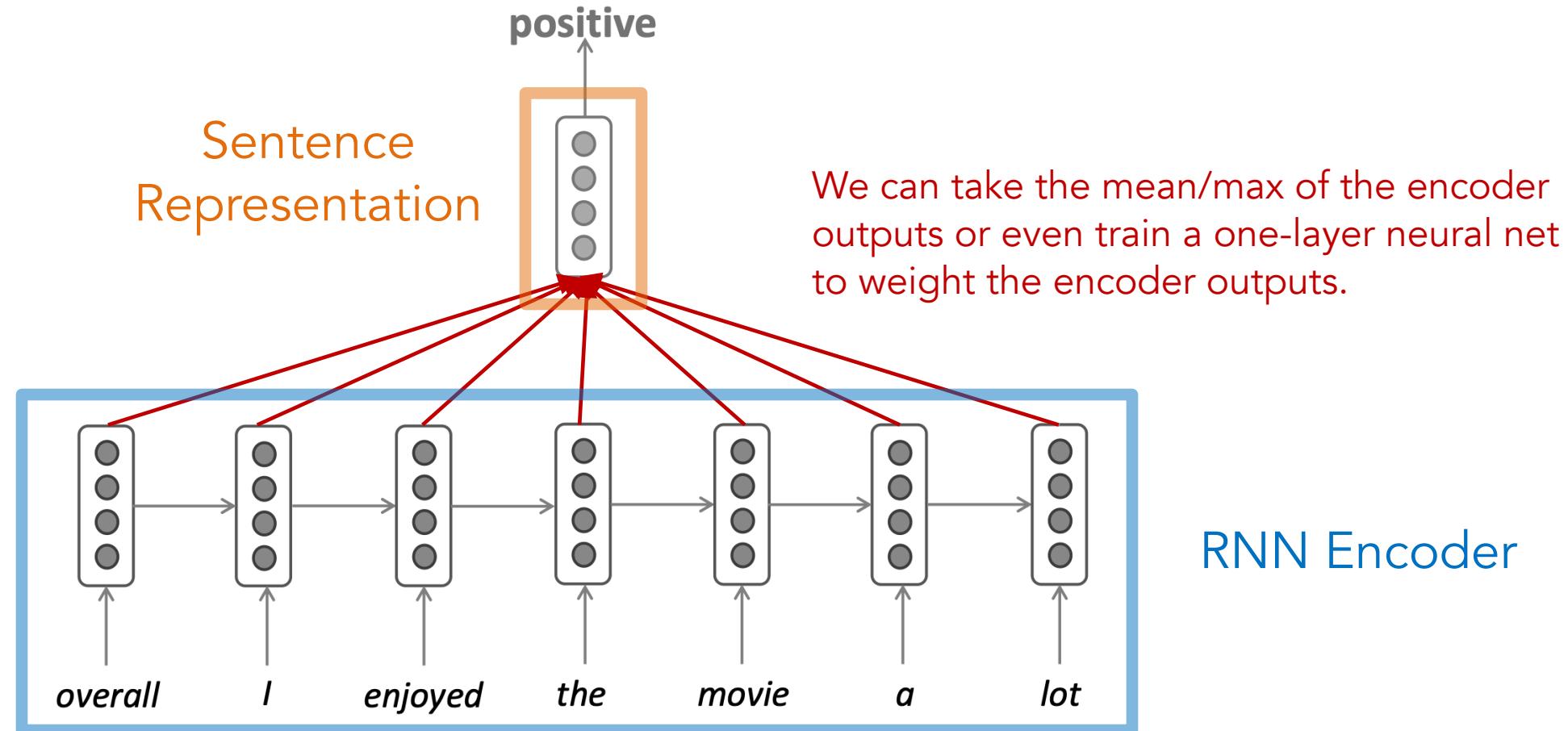
Seq2Seq models are **flexible in the input and output sizes**. The rectangles in the graph below mean vectors, red rectangles mean inputs, and blue rectangles mean outputs.



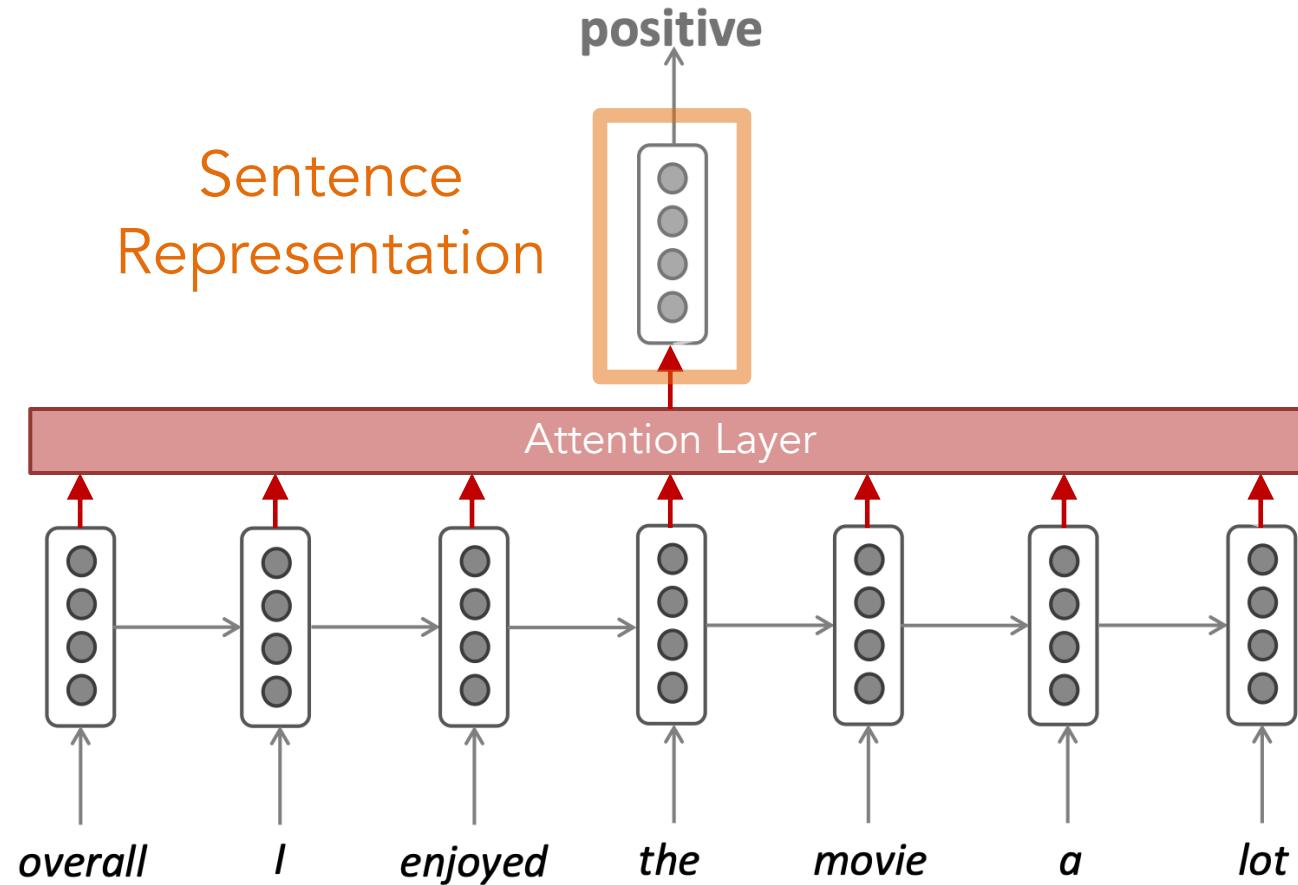
We can generalize the Seq2Seq model further to the **encoder-decoder** structure, where the encoder produces an **encoded representation** of the entire input sequence.

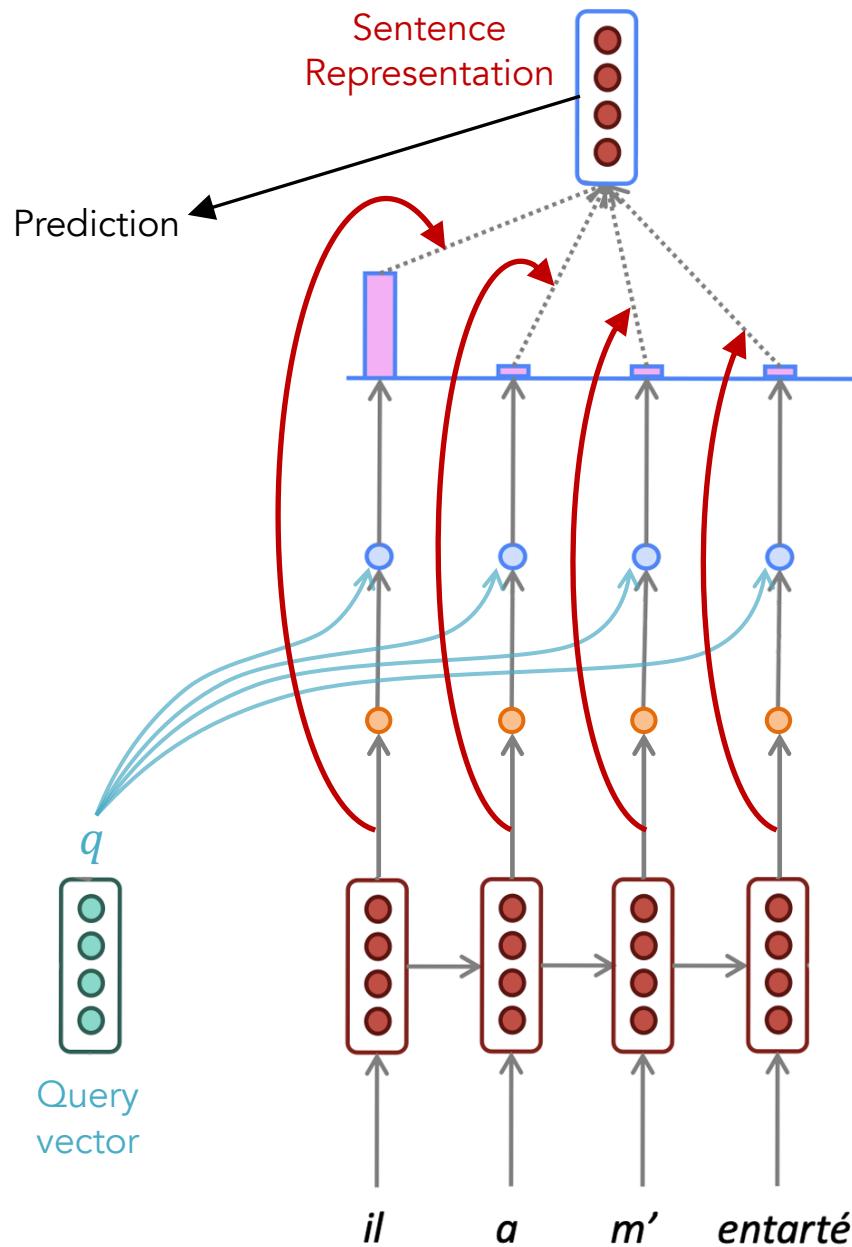


The problem of using only the final encoder output is that it is hard for the model to remember previous information. Instead, we can have the model **considers all outputs**.



But, using the same weights may be insufficient, as we may want the weights to change according to different inputs. We can use the **attention mechanism** to achieve this.





} Step 5: Compute attention-weighted sum of encoder output:

- $\sum_{t=1}^T a_t h_t$

} Step 4: Compute the attention distribution using softmax:

- $[a_1 \ a_2 \ \dots \ a_T] = \text{softmax}([e_1 \ e_2 \ \dots \ e_T])$

} Step 3: Compute attention scores (dot product similarity):

- $e_t = q^T u_t$ q is trainable

} Step 2: Transform encoder outputs (dimension reduction):

- $u_t = \tanh(W h_t)$ W is trainable

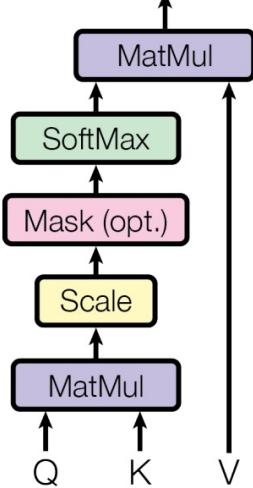
} Step 1: Get the encoder output values (from the RNN):

- h_t

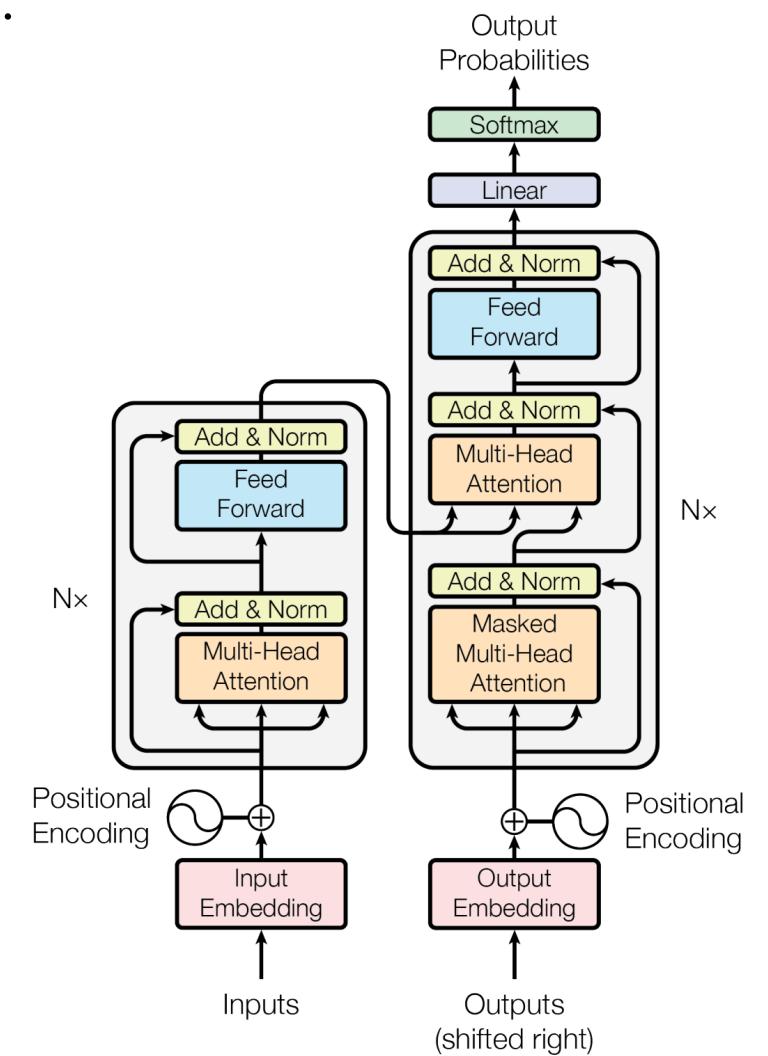
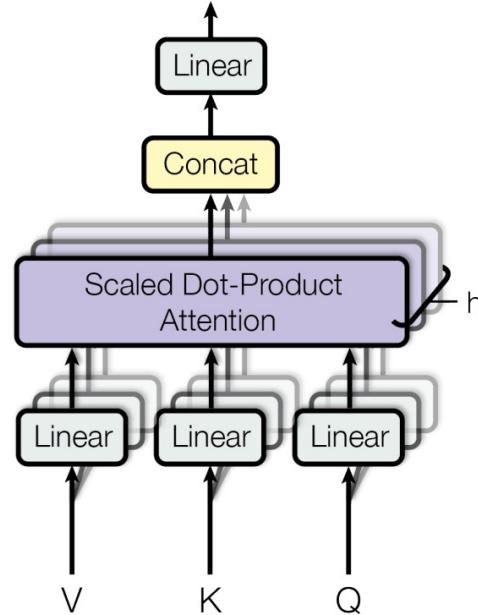
There are many ways of doing step 2 and 3

There is a more complicated attention mechanism, “multi-head self attention”, which is the building block of the Transformer network architecture.

Scaled Dot-Product Attention



Multi-Head Attention



Take-Away Messages

- We need to represent text as numbers for Natural Language Processing tasks.
- We can train word embeddings (vectors) to map words into data points in a high dimensional space.
- One way to train word embeddings is to use the context (e.g., nearby words) to represent a word.
- Word embeddings also encode semantics, which means similar words are close to each other.
- Cosine similarity and dot product can be used to measure how vectors are close to each other.
- Softmax is a commonly used function in deep learning to map arbitrary values to probabilities.
- Recurrent Neural Network can take inputs with various lengths (e.g., sentences).
- Attention helps the model learn information from the past and focus on a certain part of the source.



Questions?