

WORKING WITH TEXT

WHAT IS A REGULAR EXPRESSION?

A regular expression (*regex*) describes a set of possible input strings.

Regular expressions descend from a fundamental concept in Computer Science called *finite automata* theory

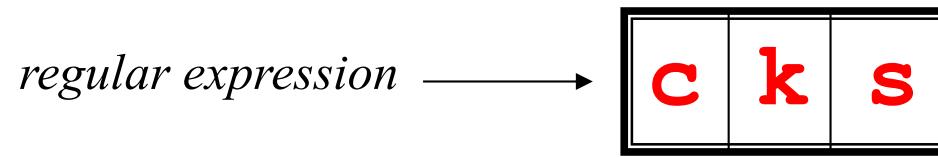
Regular expressions are used in many *nix tools

- vi, ed, sed, and emacs
- awk, tcl, perl and Python
- grep, egrep, fgrep
- compilers

REGULAR EXPRESSIONS

The simplest regular expressions are a string of literal characters to match.

The string *matches* the regular expression if it contains the substring.



Unix rocks.



match

UNIX sucks.



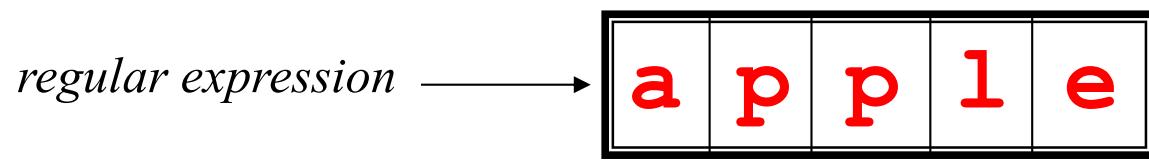
match

UNIX is okay.

no match

REGULAR EXPRESSIONS

A regular expression can match a string in more than one place.



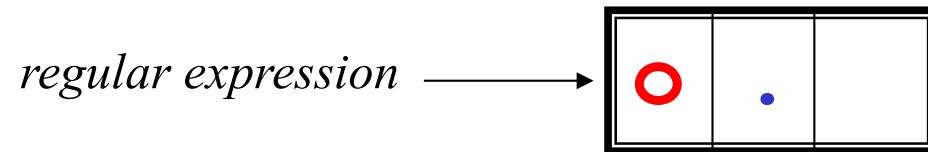
Sc**apple** from the **apple**.

match 1 *match 2*

Two arrows point upwards from the text "Sc**apple** from the **apple**." to the second and fourth boxes of the string "apple" in the diagram above. The first arrow is labeled "match 1" and the second is labeled "match 2".

REGULAR EXPRESSIONS

The `.` regular expression can be used to match any character.



For me to open

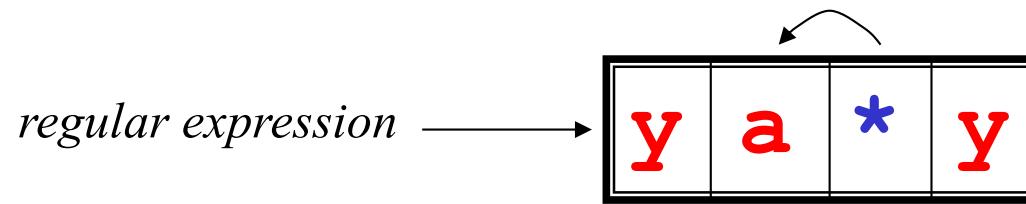
match 1 *match 2*

The word "For me to open" is shown. The letters "For" are enclosed in a dashed square, with an arrow below it labeled "match 1". The letters "open" are also enclosed in a dashed square, with an arrow below it labeled "match 2".

REPETITION

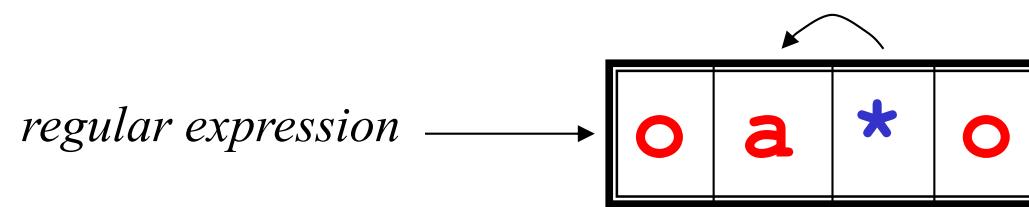
The ***** is used to define **zero or more** occurrences of the *single* regular expression preceding it.

- + Matches one or more occurrences



I got mail, **yaaaaaaaaay!**

↑
match



I sat on the stoop

↑
match

REPETITION RANGES

Ranges can also be specified

- { } notation can specify a range of repetitions for the immediately preceding regex
- {*n*} means exactly *n* occurrences
- {*n*, } means at least *n* occurrences
- {*n*,*m*} means at least *n* occurrences but no more than *m* occurrences

Example:

- .{0,} same as .*
- a{2,} same as aaa*

OR

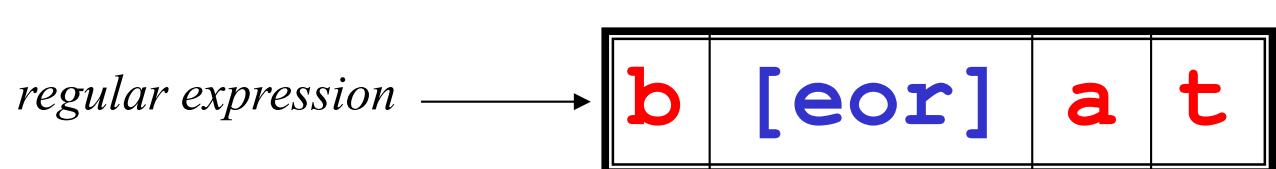
$a|b^*$ denotes $\{\epsilon, "a", "b", "bb", "bbb", \dots\}$

$(a|b)^*$ denotes the set of all strings with no symbols other than "a" and "b", including the empty string: $\{\epsilon, "a", "b", "aa", "ab", "ba", "bb", "aaa", \dots\}$

$ab^*(c)$ denotes the set of strings starting with "a", then zero or more "b"s and finally optionally a "c": $\{"a", "ac", "ab", "abc", "abb", "abbc", \dots\}$

CHARACTER CLASSES – OR SHORTHAND

Character classes [] can be used to match any specific set of characters.

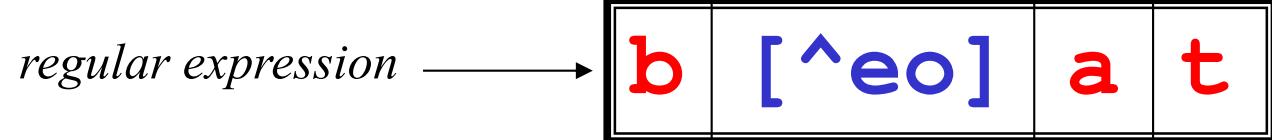


beat a brat on a boat

match 1 match 2 match 3

NEGATED CHARACTER CLASSES

Character classes can be negated with the [^] syntax.



beat a brat on a boat

↑
match

MORE ABOUT CHARACTER CLASSES

- `[aeiou]` will match any of the characters **a**, **e**, **i**, **o**, or **u**
- `[kK]orn` will match **korn** or **Korn**

Ranges can also be specified in character classes

- `[1-9]` is the same as `[123456789]`
- `[abcde]` is equivalent to `[a-e]`
- You can also combine multiple ranges
 - `[abcde123456789]` is equivalent to `[a-e1-9]`
- Note that the `-` character has a special meaning in a character class *but only* if it is used within a range, `[-123]` would match the characters `-`, `1`, `2`, or `3`

NAMED CHARACTER CLASSES

Commonly used character classes can be referred to by name (*alpha*, *lower*, *upper*, *alnum*, *digit*, *punct*, *cntrl*)

Syntax [:name:]

- [a-zA-Z] [:alpha:]
- [a-zA-Z0-9] [:alnum:]
- [45a-zA-Z] [45[:lower:]]

Important for portability across languages

CLICKER QUESTION

Select the string for which the regular expression
'..\\19..' would find a match:

- a) "12.1000"
- b) "123.1900"
- c) "12.2000"
- d) the regular expression does not
match any of the strings above

ANCHORS

Anchors are used to match at the beginning or end of a line (or both).

^ means beginning of the line

\$ means end of the line

regular expression



^	b	[eor]	a	t
---	---	-------	---	---

beat a brat on a boat

↑
match

regular expression

b	[eor]	a	t	\$
---	-------	---	---	----

beat a brat on a **boat**

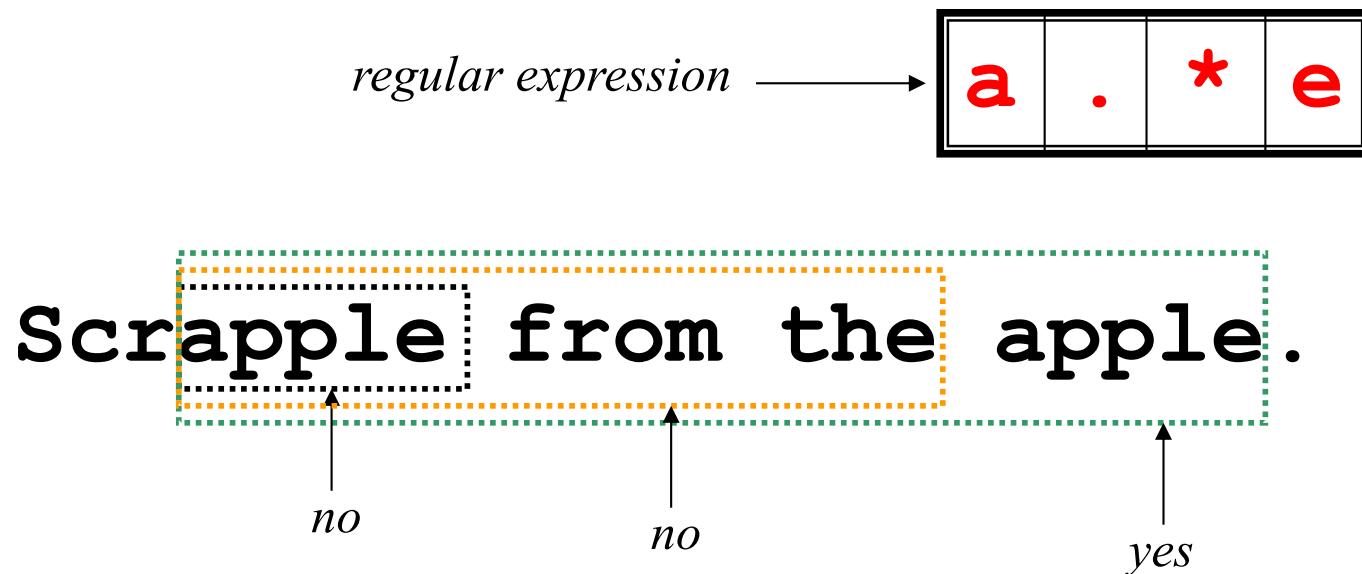
↑
match

^word\$

^\$

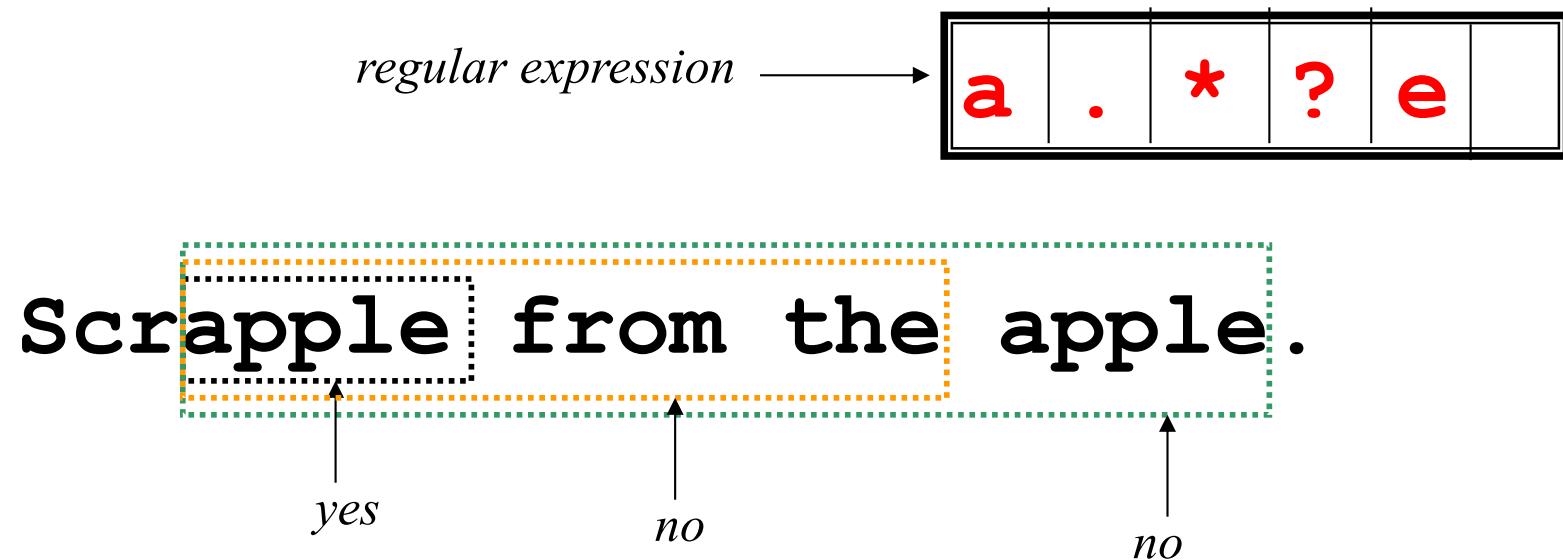
MATCH LENGTH

By default, a match will be the longest string that satisfies the regular expression.



MATCH LENGTH

Append a ? to match the shortest string possible:



PRACTICAL REGEX EXAMPLES

Dollar amount with optional cents

- `\$ [0-9] + (\. [0-9] [0-9]) ?`

Time of day

- `(1 [012] | [1-9]) : [0-5] [0-9] (am | pm)`

HTML headers <h1> <H1> <h2> ...

- `< [hH] [1-4] >`

THREE POWERFUL TOOLS

1) grep – find text matching a regular expression

Basic syntax:

```
grep 'regexp' filename
```

or equivalently (using UNIX pipelining):

```
cat filename | grep 'regexp'
```

2) sed – stream editor

3) awk – general purpose text processing language

GREP

- grep comes from the ed (Unix text editor) search command “global regular expression print” or g/re/p
- This was such a useful command that it was written as a standalone utility
- There are two other variants, *egrep* and *fgrep* that comprise the *grep* family
- *grep* is the answer to the moments where you know you want the file that contains a specific phrase but you can't remember its name

GREP DEMO

```
grep '\"text\": \".*location.*\"' twitter.json
```

"text": "RT @TwitterMktg: Starting today, businesses can request and share locations when engaging with people in Direct Messages.
<https://t.co/rpYn...>",

"text": "Starting today, businesses can request and share locations when engaging with people in Direct Messages.
<https://t.co/rpYndqWfQw>",

CLICKER QUESTION

Choose the pattern that finds all filenames in which

1. the first letters of the filename are chap,
2. followed by two digits,
3. followed by some additional text,
4. and ending with a file extension of .doc

For example : chap23Production.doc

- a) chap[0-9]*.doc
- b) chap*[0-9]doc
- c) chap[0-9][0-9].*\.\.doc
- d) chap*doc

THREE POWERFUL TOOLS

1) grep

Basic syntax:

```
grep 'regexp' filename
```

or equivalently (using UNIX pipelining):

```
cat filename | grep 'regexp'
```

2) sed – stream editor

Basic syntax

```
sed 's/regexp/replacement/g' filename
```

For each line in the input, the portion of the line that matches regexp (if any) is replaced with replacement.

Sed is quite powerful within the limits of operating on single line at a time.

You can use \(\) to refer to parts of the pattern match.

SED EXAMPLE

File = Trump is the president. His job is to tweet.

```
sed 's/Trump/Biden/g' file
```

```
sed 's/\\(His job is to\\).*/\\1 run the country./g' file
```

Biden is the president. His job is to tweet.

Trump is the president. His job is to run the country.

COMBINING TOOLS

Suppose we want to extract all the “screen_name” fields from twitter data

```
[  
 {  
   "created_at": "Thu Apr 06 15:28:43 +0000 2017",  
   "id": 850007368138018817,  
   "id_str": "850007368138018817",  
   "text": "RT @TwitterDev: 1/ Today we're sharing our vision for the  
future of the Twitter API platform!nhttps://t.co/XweGngmxlP",  
   "truncated": false,  
 ...  
   "user": {  
     "id": 6253282,  
     "id_str": "6253282",  
     "name": "Twitter API",  
     "screen_name": "twitterapi",  
   }  
 }
```

```
grep '\"screen_name\":' twitter.json |  
sed 's/[ ]*\"screen_name\": \"\\\"\\(.*)\\\",/\1/g'
```

COMBINING TOOLS

Suppose we want to extract all the “screen_name” fields from twitter data

```
[  
 {  
   "created_at": "Thu Apr 06 15:28:43 +0000 2017",  
   "id": 850007368138018817,  
   "id_str": "850007368138018817",  
   "text": "RT @TwitterDev: 1/ Today we're sharing our vision for the  
future of the Twitter API platform!nhttps://t.co/XweGngmx1P",  
   "truncated": false,  
 ...  
   "user": {  
     "id": 6253282,  
     "id_str": "6253282",  
     "name": "Twitter API",  
     "screen_name": "twitterapi",  
_____
```

```
grep \"screen_name\": twitter.json |  
sed 's/[ ]*\"screen_name\": \"\\\"\\(.*)\\\",/\\1/g'
```

EXAMPLE 2: LOG PARSING

```
192.168.2.20 -- [28/Jul/2006:10:27:10 -0300] "GET /cgi-bin/try/ HTTP/1.0" 200 3395  
127.0.0.1 -- [28/Jul/2006:10:22:04 -0300] "GET / HTTP/1.0" 200 2216
```

```
sed -E 's/^([0-9]+\.[0-9]+\.[0-9]+\.[0-9]+) ([^"]*"([^\"]*)"\.*)/\1,\2/g' apache.txt
```

IP Address	Stuff	URL
	up to quote	

```
192.168.2.20,GET /cgi-bin/try/ HTTP/1.0  
127.0.0.1,GET / HTTP/1.0
```

THREE POWERFUL TOOLS

Awk

Finally, awk is a powerful scripting language (not unlike perl). The basic syntax of awk is:

```
awk -F',' ' BEGIN{ commands }
          /regexp1/ { command1 } /regexp2/ { command2 }
          END{ commands } '
```

- For each line, the regular expressions are matched in order, and if there is a match, the corresponding command is executed (multiple commands may be executed for the same line).
- BEGIN and END are both optional.
- The -F',' specifies that the lines should be split into fields using the separator ",", and those fields are available to the regular expressions and the commands as \$1, \$2, etc.
- See the manual (man awk) or online resources for further details.

AWK COMMANDS

{ print \$1 } – *Match any line, print the 1st field*

\$1=="Obama">{print \$2}'

If the first field is “Obama”, print the 2nd field

'\$0 ~ /Obama/ {t = gsub("Obama", "Trump", "g", \$0); print t}'

If the line contains Obama, globally replace “Trump” for “Obama” and assign the result to the variable “txt”. Then print it.

Awk commands:

https://www.gnu.org/software/gawk/manual/html_node/Built_002din.html

WRANGLING IN AWK

Input data

Reported crime in Alabama,

,
2004,4029.3

2005,3900

2006,3937

2007,3974.9

2008,4081.9

,

Reported crime in Alaska,

,
2004,3370.9

2005,3615

2006,3582

2007,3373.9

2008,2928.3

,

Reported crime in Arizona,

,
2004,5073.3

2005,4827

Desired Output:

2004,Alabama,4029.3

2005,Alabama,3900

2006,Alabama,3937

2007,Alabama,3974.9

2008,Alabama,4081.9

2004,Alaska,3370.9

2005,Alaska,3615

2006,Alaska,3582

2007,Alaska,3373.9

2008,Alaska,2928.3

2004,Arizona,5073.3

2005,Arizona,4827

2006,Arizona,4741.6

2007,Arizona,4502.6

2008,Arizona,4087.3

2004,Arkansas,4033.1

2005,Arkansas,4068

AWK EXAMPLE

Reported crime in Alabama,
,

2004,4029.3
2005,3900
2006,3937
2007,3974.9
2008,4081.9

```
BEGIN {FS="[, ]"}  
$1=="Reported" {           line begins w/ reported  
state = $4" "$5;          some states are two words  
gsub(/[\t]+$/ , "" , state) strip trailing spaces  
}  
$1 ~ 20 {print $1 , "state" , "$2"}  
line begins with 20           print year, state, and amount
```

TEXT AS DATA

What might we want to do?

Find similar documents

E.g., for document clustering

Find similarity between a document and a string

E.g., for document search

Answer questions from documents

Assess document sentiment

Extract information from documents

Focus today:
Given two
pieces of
text, how do
we measure
similarity?

TOKENIZATION

- A **token** is an instance of a sequence of characters

Input: “*Friends, Romans and Countrymen*”

Output: Tokens

- *Friends*
- *Romans*
- *and*
- *Countrymen*

- What are valid tokens?
- Typically, just words, but can be complicated

E.g., how many tokens is

Lebensversicherungsgesellschaftsangestellter, meaning ‘life insurance company employee’ in German?

WHY TOKENIZE?

- Often useful to think of text as a bag of words, or as a table of words and their frequencies
- Need a standard way to define a word, and correct for differences in formatting, etc.
- LLMs are trained to consume and predict tokens
- Very common in information retrieval (IR) / keyword search
 - Typical goal: find similar documents based on their words or n-grams (length n word groups)

DOCUMENT SIMILARITY EXAMPLE

Suppose we have the following strings, and want to measure their similarity?

```
sen = [  
    "Tim loves the band Korn.",  
    "Tim adores the rock group Korn.",  
    "Tim loves eating corn.",  
    "Tim used to love Korn, but now he hates them.",  
    "Tim absolutely loves Korn.",  
    "Tim completely detests the performers named Korn",  
    "Tim has a deep passion for the outfit the goes by the name of Korn",  
    "Tim loves listening to the band Korn while eating corn."  
]
```

BAG-OF-WORDS MODEL

- Treat documents as sets
- Measure similarity of sets

Standard set similarity metric: Jaccard Similarity

$$\text{sim}(s1, s2) = \frac{s1 \cap s2}{s1 \cup s2}$$

$\text{sim}(\{\text{tim}, \text{loves}, \text{korn}\}, \{\text{tim}, \text{loves}, \text{eating}, \text{corn}\}) = 2 / 5$

$\text{sim}(\{\text{tim}, \text{absolutely}, \text{adores}, \text{the}, \text{band}, \text{korn}\}, \{\text{tim}, \text{loves}, \text{korn}\}) = 2 / 7$

Problems:

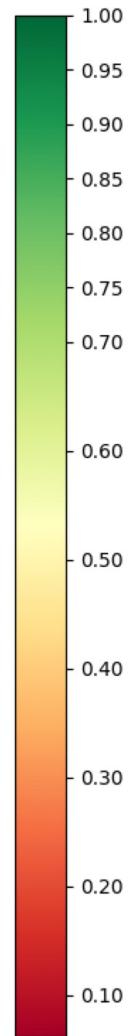
All words weighted equally

Same word with different suffix treated differently (e.g., love & loves)

Semantic significance ignored (e.g., adores & loves are the same)

Duplicates are ignored ("Tim really, really loves Korn")

EXAMPLE



STOP WORDS

With a stop list, you exclude from the dictionary entirely the commonest words. Intuition:

- They have little semantic content: *the, a, and, to, be*
- There are a lot of them: ~30% of postings for top 30 words

Sometimes you want to include them, as they affect meaning

- Phrase queries: “King of Denmark”
- Various song titles, etc.: “Let it be”, “To be or not to be”
- “Relational” queries: “flights to London”

STOP WORDS IN PYTHON

```
from nltk.corpus import stopwords  
print(stopwords.words('english'))
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've",  
"you'll", "you'd", 'your', 'yours', 'yourself', 'yourselves', 'he', 'him', 'his',  
'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's", 'its', 'itself', 'they', 'them',  
'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'that', "that'll",  
'these', 'those', 'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has',  
'had', 'having', 'do', 'does', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or',  
'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against',  
'between', 'into', 'through', 'during', 'before', 'after', 'above', 'below', 'to',  
'from', 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further', 'then',  
'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'each',  
'few', 'more', 'most', 'other', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'same',  
'so', 'than', 'too', 'very', 's', 't', 'can', 'will', 'just', 'don', "don't", 'should',  
"should've", 'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'aren', "aren't", 'couldn',  
"couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't",  
'haven', "haven't", 'isn', "isn't", 'ma', 'mightn', "mightn't", 'mustn', "mustn't",  
'needn', "needn't", 'shan', "shan't", 'shouldn', "shouldn't", 'wasn', "wasn't",  
'weren', "weren't", 'won', "won't", 'wouldn', "wouldn't"]
```

STEMMING

- Reduce terms to their “roots” before indexing
- “Stemming” performs crude affix chopping
 - language dependent
 - e.g., *automate(s)*, *automatic*, *automation* all reduced to *automat.*

for example compressed and compression are both accepted as equivalent to compress.



for exampl compress and compress ar both accept as equival to compress

PORTRER'S ALGORITHM

Most common algorithm for stemming English

- Other options exist, e.g., snowball

Conventions + 5 phases of reductions

- phases applied sequentially
- each phase consists of a set of commands
- sample convention: *Of the rules in a compound command, select the one that applies to the longest suffix.*

TYPICAL RULES IN PORTER

sses → *ss*

ies → *i*

ational → *ate*

tional → *tion*

Weight of word sensitive rules

$(m > 1)$ *EMENT* →

- *replacement* → *replac*
- *cement* → *cement*

STEMMING IN PYTHON

```
import nltk.stem.porter  
  
stemmer = nltk.stem.porter.PorterStemmer()  
for w in sen[0].split(" "):  
    print(stemmer.stem(w))
```

tim
love
the
band
korn

STEP WORDS + STEMMING

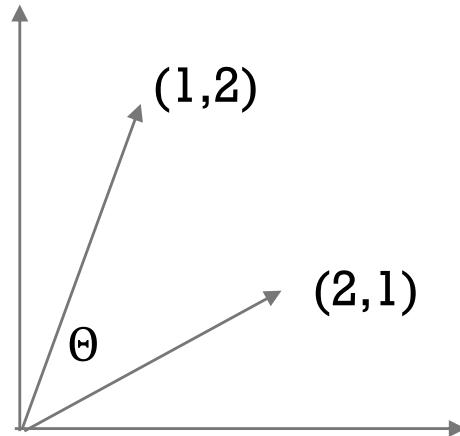
```
sen = [  
    "Tim loves the band Korn.",  
    "Tim adores the rock group Korn.",  
    "Tim loves eating corn.",  
    "Tim used to love Korn, but now he hates them.",  
    "Tim absolutely loves Korn.",  
    "Tim completely detests the performers named Korn",  
    "Tim has a deep passion for the outfit the goes by the name of Korn",  
    "Tim loves listening to the band Korn while eating corn."  
]
```

tim love band korn
tim ador rock group korn
tim love eat corn
tim use love korn hate
tim absolut love korn
tim complet detest perform name korn
tim deep passion outfit goe korn
tim love listen band korn eat corn

COSINE SIMILARITY

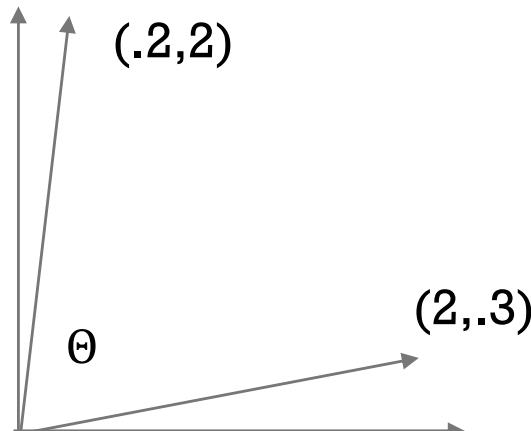
Given two vectors, a standard way to measure how similar they are

$\text{Cos}(v_1, v_2) = \text{closeness of two vectors (smaller is closer)}$



$$\text{Cos}(\Theta) = \mathbf{V1} \cdot \mathbf{V2} / ||\mathbf{V1}|| \times ||\mathbf{V2}||$$

$$\begin{aligned}\text{Cos}(\Theta) &= [1 \ 2] \cdot [2 \ 1] / (\sqrt{5})^2 \\ \text{Arcos}(4/5) &= 36.8^\circ\end{aligned}$$



$$\begin{aligned}||\mathbf{V1}|| &= 2.01, \quad ||\mathbf{V2}|| = 2.02 \\ \text{Cos}(\Theta) &= [.2 \ 2] \cdot [2 \ .3] / 2.015 \\ &= 1/2.015 \\ \text{Arcos}(1/2.015) &= 60.2^\circ\end{aligned}$$

COSINE SIMILARITY OF WORD VECTORS

$$\text{Cos}(\Theta) = \mathbf{V1} \bullet \mathbf{V2} / \|\mathbf{V1}\| \times \|\mathbf{V2}\|$$

$\begin{matrix} 1 & 2 & 3 \\ S1 = \text{Tim loves Korn} & & \\ & 4 & 5 \end{matrix}$
 $S2 = \text{Tim loves eating corn}$

$$\begin{matrix} \mathbf{V1} = 1 & 1 & 1 & 0 & 0 \\ \mathbf{V2} = 1 & 1 & 0 & 1 & 1 \end{matrix}$$

$$\begin{matrix} \mathbf{V1} \bullet \mathbf{V2} = 2 \\ \|\mathbf{V1}\| = \text{sqrt}(3) \\ \|\mathbf{V2}\| = \text{sqrt}(4) \\ \\ 2 / \text{sqrt}(3) * \text{sqrt}(4) = .58 \end{matrix}$$

$\begin{matrix} 1 & 2 & 3 \\ S1 = \text{Tim loves Korn} & & \\ & 4 & 5 & 6 & 7 \\ S2 = \text{Tim absolutely adores the band Korn} & & & & \end{matrix}$

$$\begin{matrix} \mathbf{V1} = 1 & 1 & 1 & 0 & 0 & 0 \\ \mathbf{V2} = 1 & 0 & 1 & 1 & 1 & 1 \end{matrix}$$

$$\begin{matrix} \mathbf{V1} \bullet \mathbf{V2} = 2 \\ \|\mathbf{V1}\| = \text{sqrt}(3) \\ \|\mathbf{V2}\| = \text{sqrt}(6) \end{matrix}$$

$$2 / \text{sqrt}(3) * \text{sqrt}(6) = .47$$

Typically, when using cosine similarity, we don't take the acos of the values (since acos is expensive)

JACCARD VS COSINE

S1 = Tim loves Korn

S2 = Tim loves eating corn

CosSim(S1,S2) = .29

Jaccard(S1,S2) = .4

S3 = Tim absolutely adores the band Korn

CosSim(S1,S3) = .43

Jaccard(S1,S3) = .28

Jaccard more sensitive to different document lengths than CosSim

CosSim can incorporate repeated words (by using non-binary vectors)

CLICKER

<https://clicker.csail.mit.edu/6.s079/>

Consider two sentences:

Sam loves limp bizkit

Sam eats limp biscuits

What is their Jaccard similarity?

- A. 4/6
- B. 2/8
- C. 2/6
- D. Something else

{Sam, limp}

{Sam, loves, limp, bizkit, eats, biscuits}

What is their Cosine similarity?

- A. 1/4
- B. 2/4
- C. 4/6
- D. Something else

S1: 1 1 1 1 0 0

S2: 1 0 1 0 1 1

$$S1 \cdot S2 = 2$$

$$||S1|| = ||S2|| = \sqrt{4}$$

COSINE SIMILARITY PLOT

Includes
stemming



WHICH WORDS MATTER: TF-IDF

Problem: neither Jaccard nor Cosine Similarity have a way to understand which words are important

TF-IDF tries to estimate the importance of words based on

- 1) Their Term Frequency (TF) in a document
- 2) Their Inter-document Frequency (IDF), across all documents

Assumptions: If a term appears frequently in a document, it's more important in that document

If a term appears frequently in all documents, its less important

TF-IDF EQUATIONS

t = t

d = document

f_{t,d} = frequency of t in d

$$tf(t, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

Larger the more times document d uses term

For each term t in d, $tf(t, d)$ is the fraction of words in d that are t

$$idf(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

Approaches 0 as more documents use term

N = number of documents

D = set of all documents

$|\{d \in D : t \in d\}|$ = # documents which use term t

For each term t in all D, $idf(t, D)$ is inversely proportional to the number of documents that use t

TF-IDF EXAMPLE

S1 = Tim loves Korn

S2 = Tim loves eating corn

Terms = Tim, loves, Korn, eating Korn

$$\text{tf-idf}(\text{Tim}, s1) = \text{tf}(\text{Tim}, s1) \times \text{idf}(\text{Tim}) = 1/3 \times \log(2/2) = 0$$

$$\text{tf-idf}(\text{loves}, s1) = \text{tf}(\text{loves}, s1) \times \text{idf}(\text{loves}) = 1/3 \times \log(2/2) = 0$$

$$\text{tf-idf}(\text{Korn}, s1) = \text{tf}(\text{Korn}, s1) \times \text{idf}(\text{Korn}) = 1/3 \times \log(2/1) = 1/3 \times .69 = 0.23$$

$$\text{tf-idf}(\text{eating}, s2) = \text{tf}(\text{eating}, s2) \times \text{idf}(\text{eating}) = 1/4 \times \log(2/1) = 0.17$$

$$\text{tf-idf}(\text{corn}, s2) = \text{tf}(\text{corn}, s2) \times \text{idf}(\text{corn}) = 1/4 \times \log(2/1) = 0.17$$

$$tf(t, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

$$idf(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

$$S1 = [0, 0, .23]$$

$$S2 = [0, 0, .17, .17]$$

Words in all documents aren't helpful if we're trying to rank documents according to their similarity or do keyword search

TF-IDF IN PYTHON

These parameters make it match equations on previous slide

```
#TF-IDF using sklearn
f = sklearn.feature_extraction.text.TfidfVectorizer(smooth_idf=False, norm='l1')
X = f.fit_transform(sen)
print(X.toarray())
cos_sim = cosine_similarity(X)
print(cos_sim)
```

Tim loves the band Korn	[[1. 0.13 0.26 0.29 0.37 0.11 0.11 0.57]
Tim adores the rock group Korn	[0.13 1. 0.05 0.09 0.11 0.06 0.06 0.07]
Tim loves eating corn	[0.26 0.05 1. 0.17 0.22 0.04 0.04 0.68]
Tim used to love Korn,	[0.29 0.09 0.17 1. 0.25 0.07 0.07 0.16]
but now he hates them	[0.37 0.11 0.22 0.25 1. 0.1 0.1 0.21]
	[0.11 0.06 0.04 0.07 0.1 1. 0.06 0.06]
	[0.11 0.06 0.04 0.07 0.1 0.06 1. 0.06]
	[0.57 0.07 0.68 0.16 0.21 0.06 0.06 1.]]

TF-IDF not a great choice for these sentences, because it downweights frequent words (Tim, Korn and loves)

MODERN ML TECHNIQUES

Modern deep learning has completely transformed text processing tasks like this

NLP models, e.g., BERT and GPT-3/4 trained to *understand* documents

Models are trained to predict missing words:

Tim loves the ___ Korn

Tim loves eating ___

We're going to try
BERT, which is a
slightly older model
than GPT-3/4

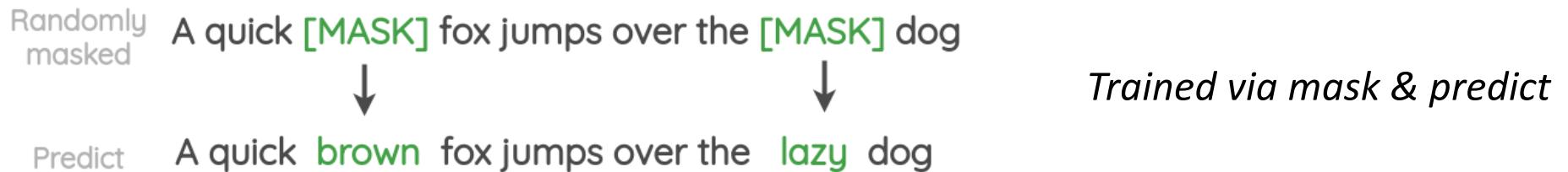
Using billions of documents on the Web (training takes years of GPU time!!!)

Models take a window of text (e.g., 512 words) and produce an output vector (e.g., 768 floats) for each word

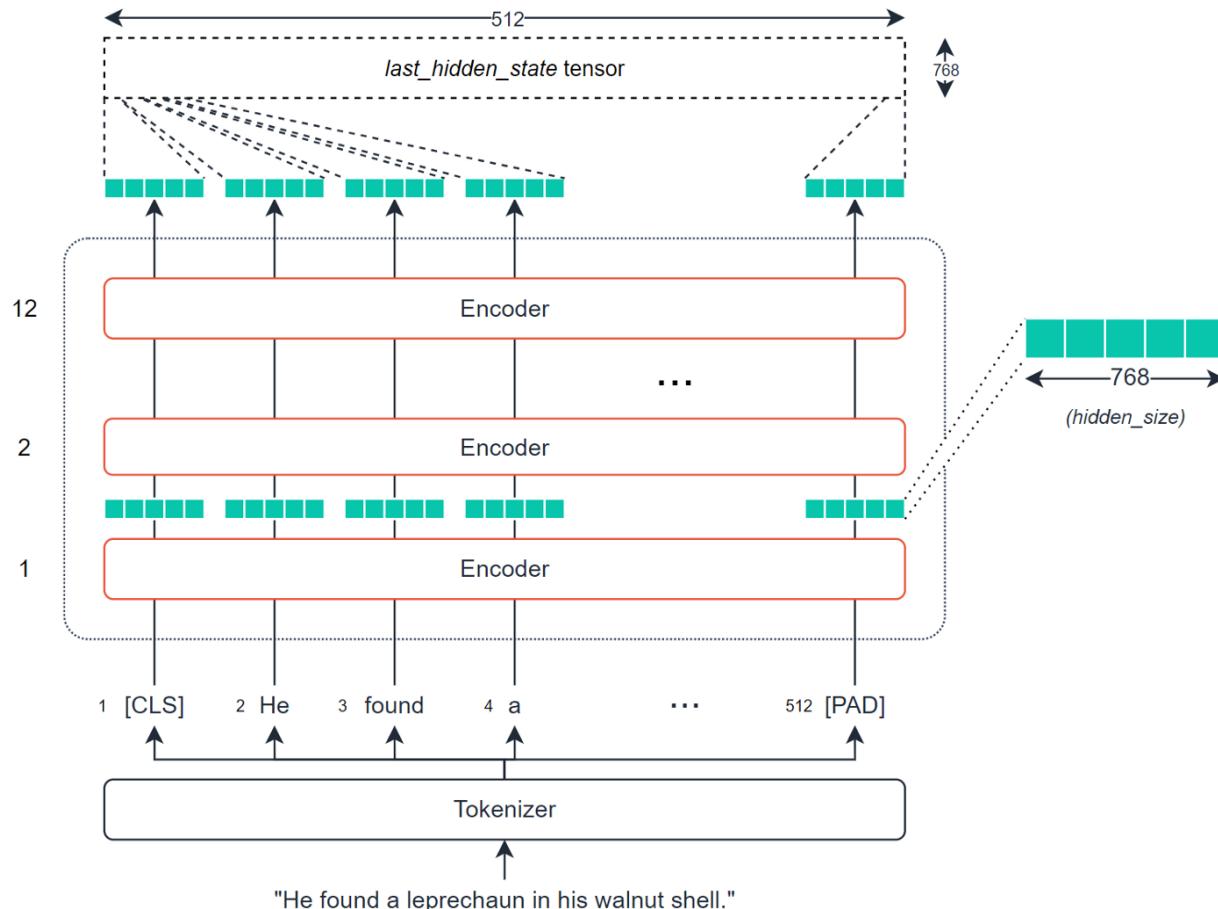
Vector represents the “meaning” of that word in **the context** of the natural language in which it appears

This vector can be used to predict the next word, or to measure the similarity of meaning of two words

BERT



Transformer Architecture



Each word in input assigned a 768 element output vector, that depends on its context (before and after)

(GPT3 is 175B!)

110M params

<https://towardsdatascience.com/bert-for-measuring-text-similarity-eec91c6bf9e1>

USING BERT VECTORS

Each word is represented by a set of 768-element outputs

Convert to a single element 768-vector for each sentence by averaging words in document

Compute similarity between vectors (e.g., using Cosine Similarity)

Python sentence-transformers package makes this trivial

```
from sentence_transformers import SentenceTransformer  
  
model = SentenceTransformer('all-mpnet-base-v2')  
sen_embeddings = model.encode(sen)  
  
cos_sim = cosine_similarity(sen_embeddings)  
  
print(cos_sim)
```

A popular BERT-like model known to perform well

Does averaging across documents

Contains a 768-element vector for each document

USING BERT VECTORS

```
from sentence_transformers import SentenceTransformer  
  
model = SentenceTransformer('all-mpnet-base-v2')  
sen_embeddings = model.encode(sen)  
  
cos_sim = cosine_similarity(sen_embeddings)  
  
print(cos_sim)
```

Tim loves the band Korn	[[1. 0.97 0.49 0.83 0.92 0.81 0.93 0.78]
Tim adores the rock group Korn	[0.97 1. 0.46 0.82 0.91 0.81 0.93 0.77]
Tim loves eating corn	[0.49 0.46 1. 0.42 0.52 0.41 0.43 0.81]
Tim used to love Korn,	[0.83 0.82 0.42 1. 0.83 0.86 0.8 0.67]
but now he hates them	[0.92 0.91 0.52 0.83 1. 0.79 0.87 0.76] [0.81 0.81 0.41 0.86 0.79 1. 0.8 0.66] [0.93 0.93 0.43 0.8 0.87 0.8 1. 0.71] [0.78 0.77 0.81 0.67 0.76 0.66 0.71 1.]]

Captures meaning of sentences much better than other metrics

HEAT MAP



Here are a bunch of sentences: "Tim loves the band Korn.", "Tim adores the rock group Korn.", "Tim loves eating corn.", "Tim used to love Korn, but now he hates them.", "Tim absolutely loves Korn.", "Tim completely detests the performers named Korn", "Tim has a deep passion for the outfit the goes by the name of Korn", "Tim loves listening to the band Korn while eating corn."

Which is most similar to the phrase "**Sam loves the band Korn**", and why?



ChatGPT

Based on the text provided in your message, the sentence most similar to "Sam loves the band Korn" is "**Tim loves the band Korn**."

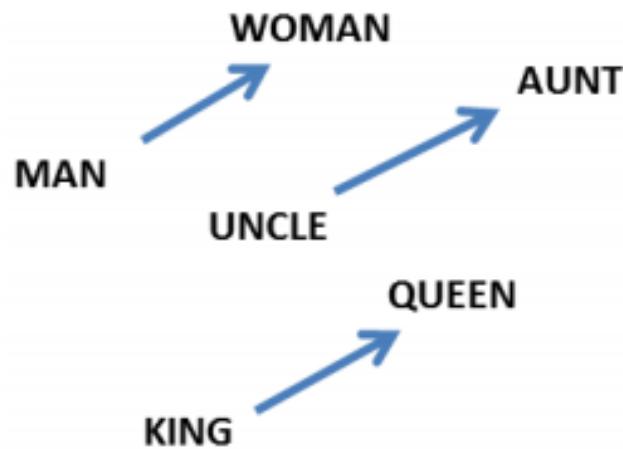
This is because both sentences express a positive sentiment about the subject's feeling towards the band Korn, with the only difference being the subject of the sentence (Sam vs. Tim). The structure and the main verb "loves" are the same, making them closely similar in both meaning and sentiment.

Word embeddings

- Idea: learn a high-dimensional representation of each word
Cat: {0.002, 0.244, 0.546, ..., 0.345}
- Need to have a function $W(\text{word})$ that returns a vector encoding that word.
- Applications: ???

Word embeddings: properties

Relationships between words correspond to difference between vectors.



$$W(\text{"woman"}) - W(\text{"man"}) \simeq W(\text{"aunt"}) - W(\text{"uncle"})$$

$$W(\text{"woman"}) - W(\text{"man"}) \simeq W(\text{"queen"}) - W(\text{"king"})$$

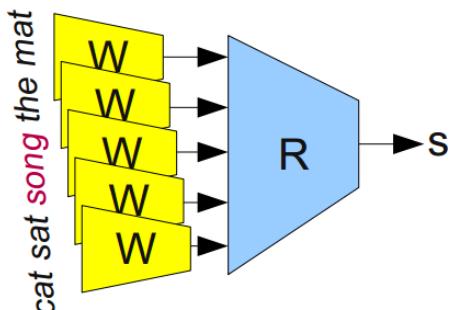
Word embeddings: questions

- How big should the embedding space be?
 - Trade-offs like any other machine learning problem – greater capacity versus efficiency and overfitting.
- How do we find W ?
 - Often as part of a prediction or classification task involving neighboring words.

Learning word embeddings

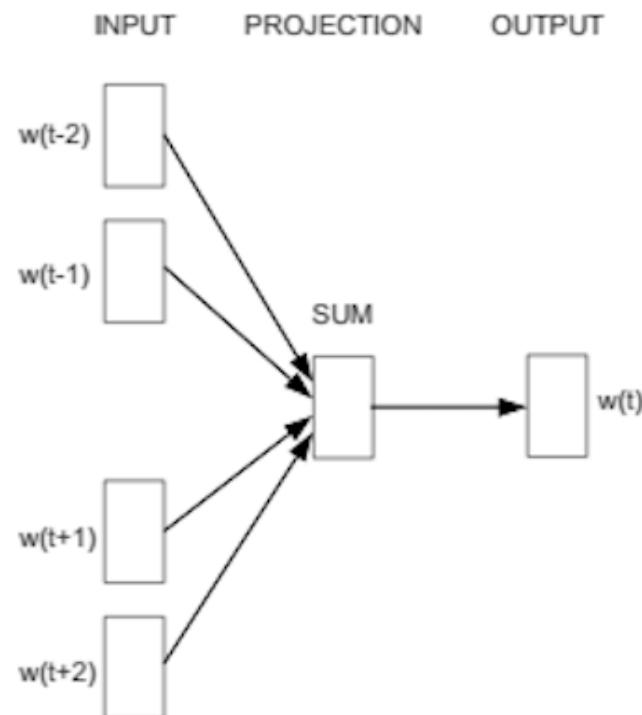
<https://arxiv.org/ftp/arxiv/papers/1102/1102.1808.pdf>

- First attempt:
 - Input data is sets of 5 words from a meaningful sentence. E.g., “one of the best places”. Modify half of them by replacing middle word with a random word. “one of function best places”
 - W is a map (depending on parameters, Q) from words to 50 dim'l vectors.
 - Feed 5 embeddings into a module R to determine ‘valid’ or ‘invalid’
 - Optimize over Q to predict better

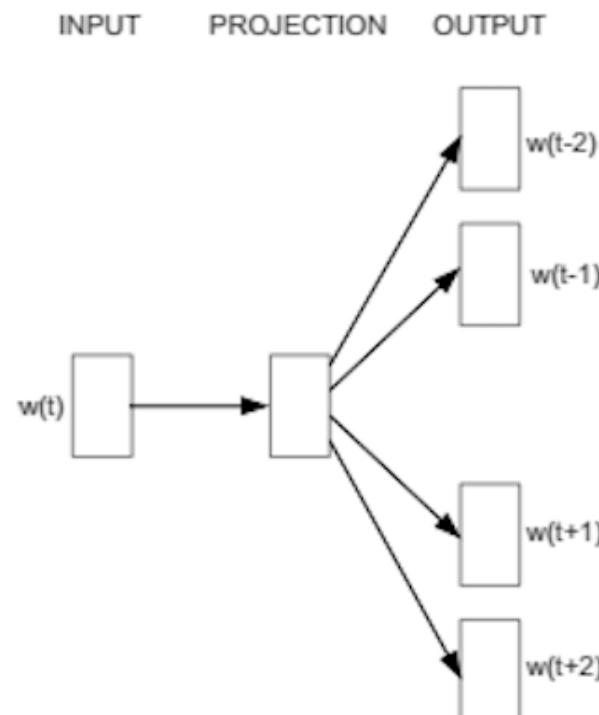


word2vec

- A set of (relatively) simple methods to produce word embeddings
 - Predict words using context
 - Two versions: CBOW and Skip-gram



CBOW



Skip-gram

Bag of Words (BOW)

- Bag of words (BOW)
 - A vector representation of word frequency
 - Vector has to be as long as vocabulary size
 - Very old method
 - Gets rid of word order!
- Exploits the “distributional hypothesis”
 - The degree of semantic similarity between two words reflects the similarity of the linguistic contexts in which those words can appear
- Words are points in an n-dimensional boolean space

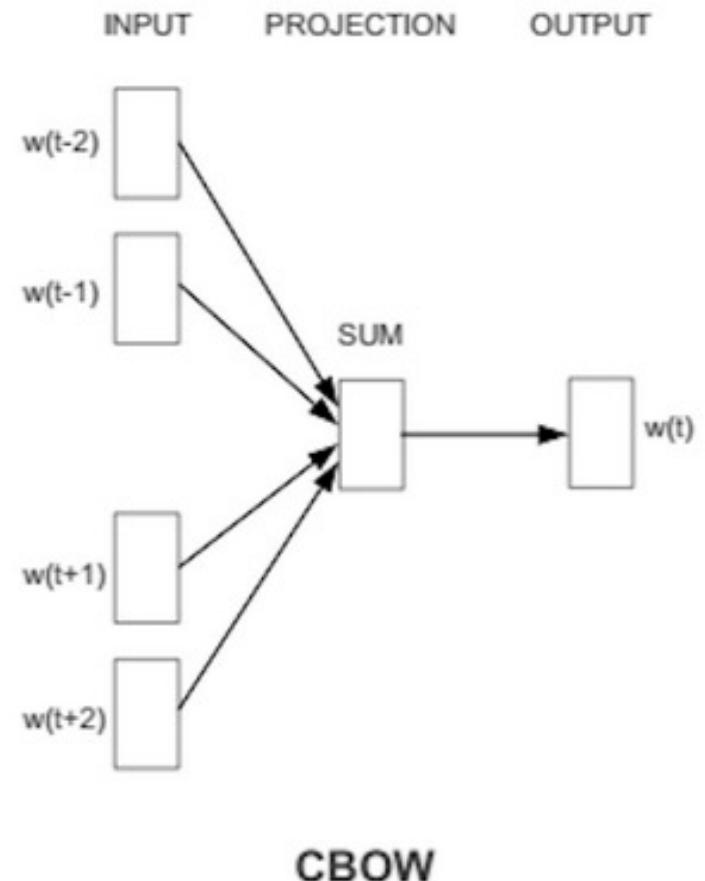
“Cat jumped on dog” [1, 0, 0, 1, 1, 1, 0]

“Dog jumped on cat” [1, 0, 0, 1, 1, 1, 0]

What are the weaknesses with this approach?

Continuous Bag of Words (CBOW)

- Takes vector embeddings of n words before target and n words after and adds them (as vectors).
- It is a “fill-in-the-blank” task
- The embedding describes how the “missing word” impacts the probability of seeing the words in its context window
- It works well even though we remove word order. The vector sum is meaningful enough to deduce missing word.
- Key idea: Use combined representations of contextual words to predict the missing word



Continuous Bag of Words - Window Size 2

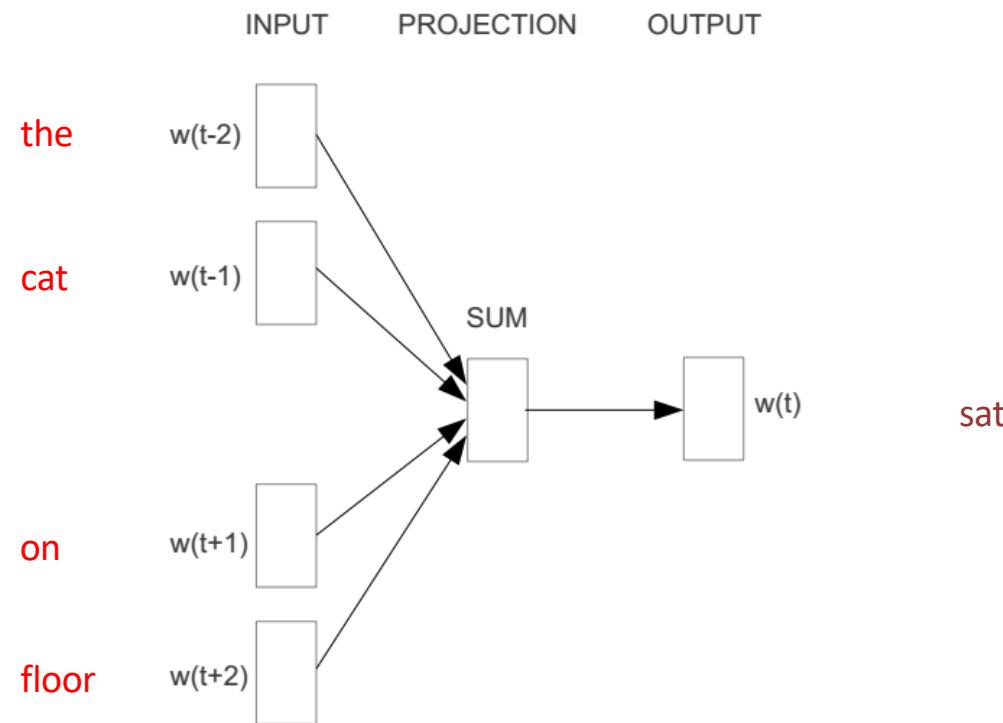
Jay was hit by a _____ bus in...

by	a	red	bus	in
----	---	-----	-----	----

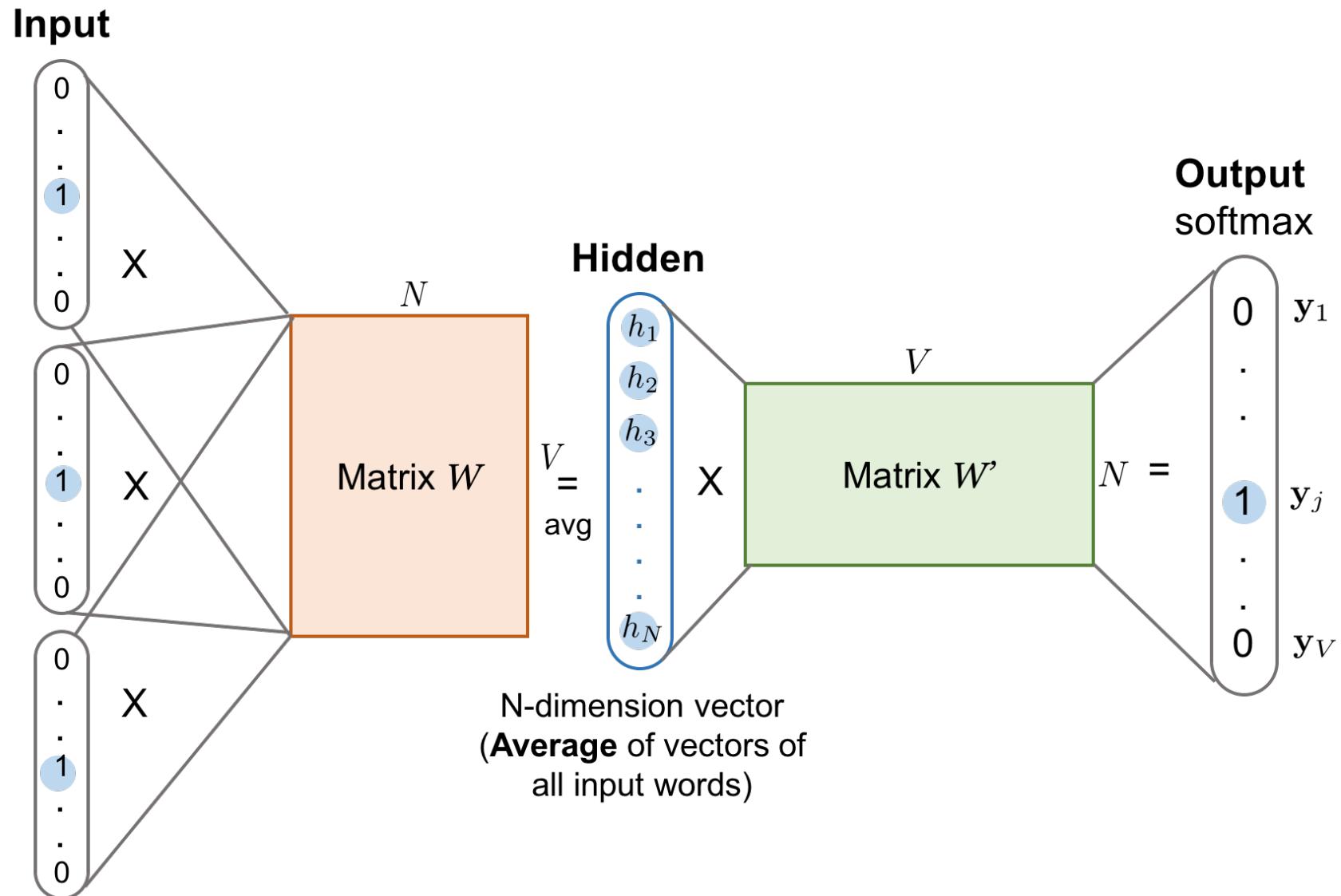
input 1	input 2	input 3	input 4	output
by	a	bus	in	red

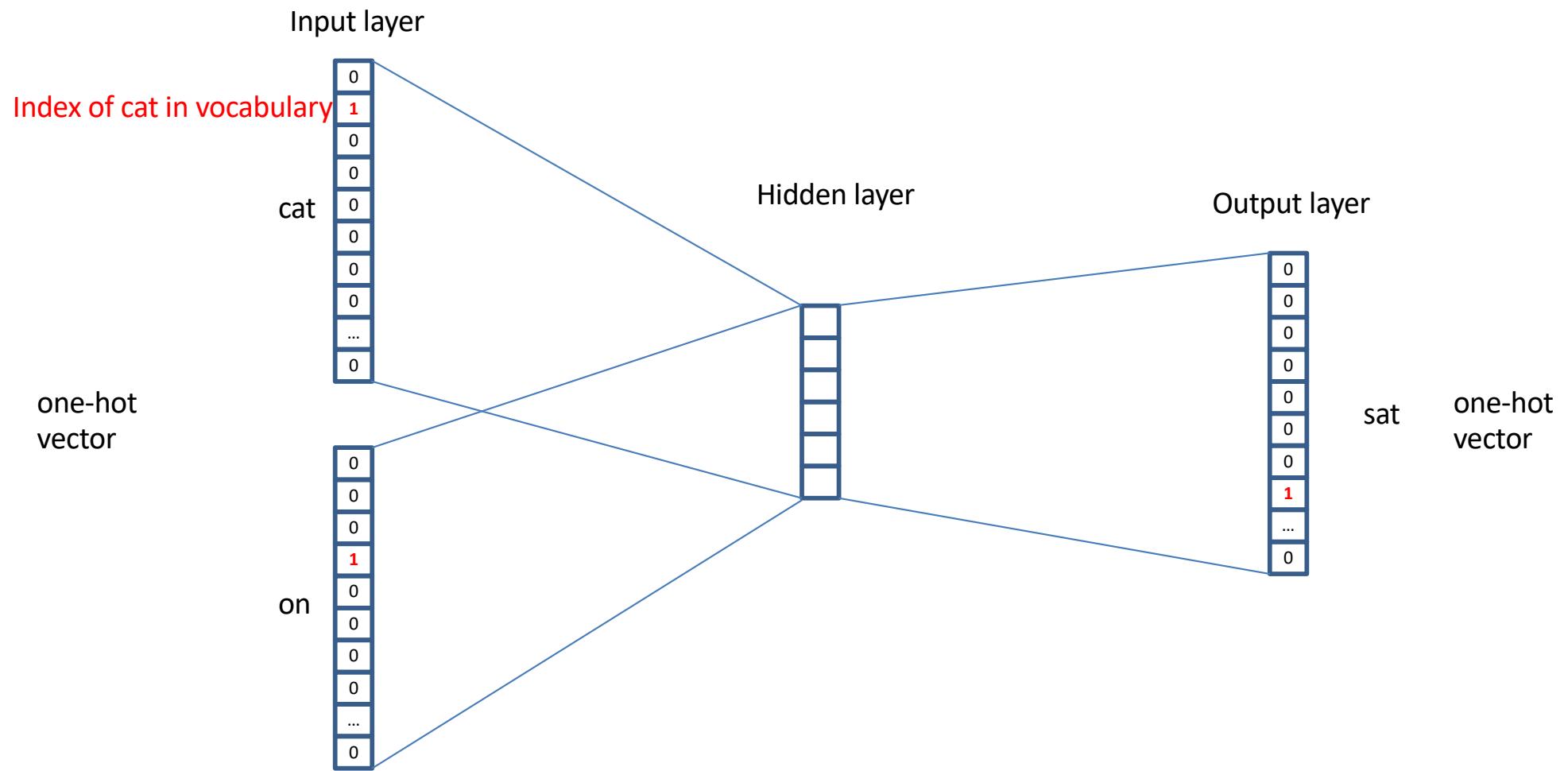
Continuous Bag of Word

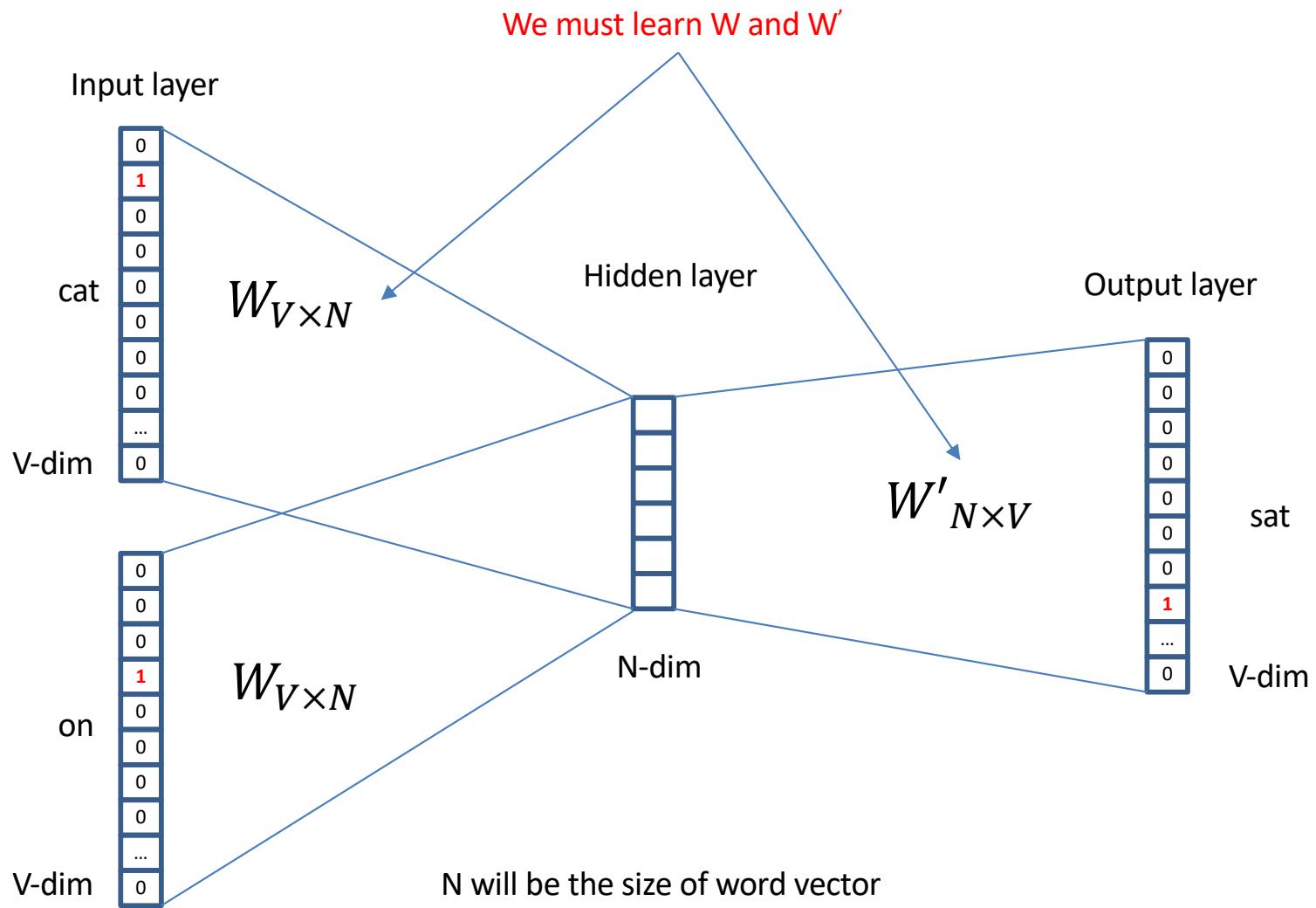
- E.g. “The cat **sat** on floor”
 - Window size = 2

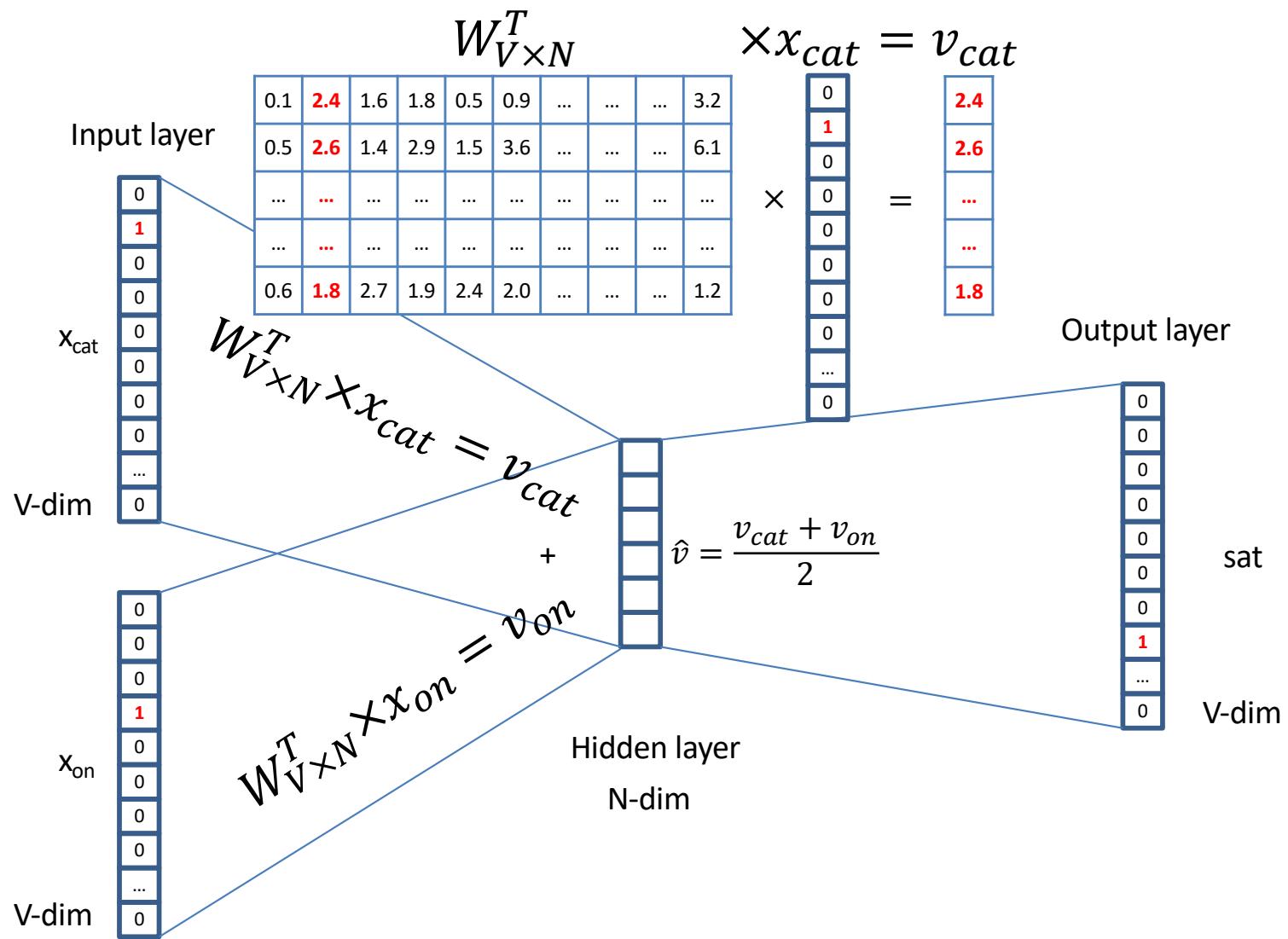


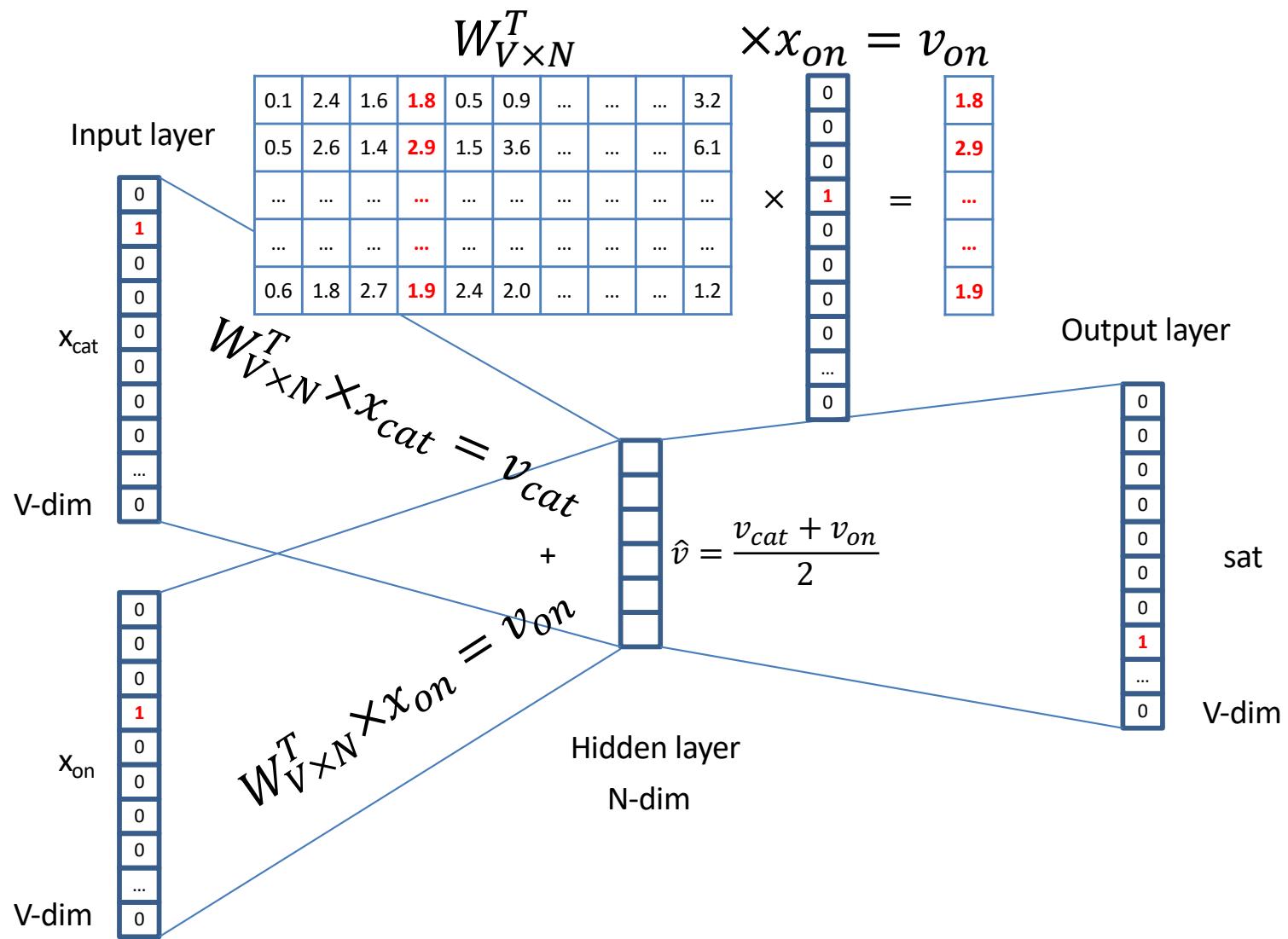
COBW





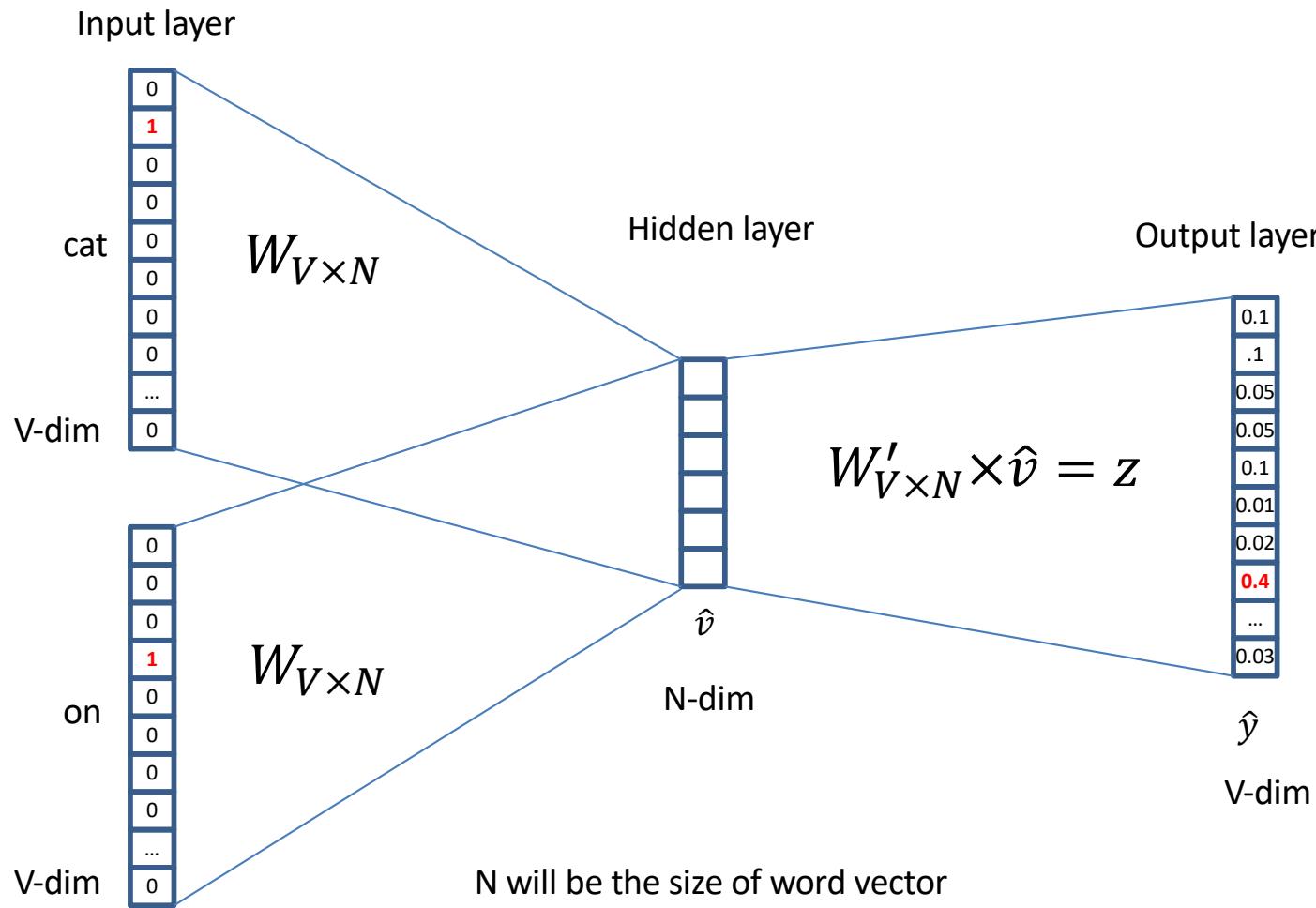




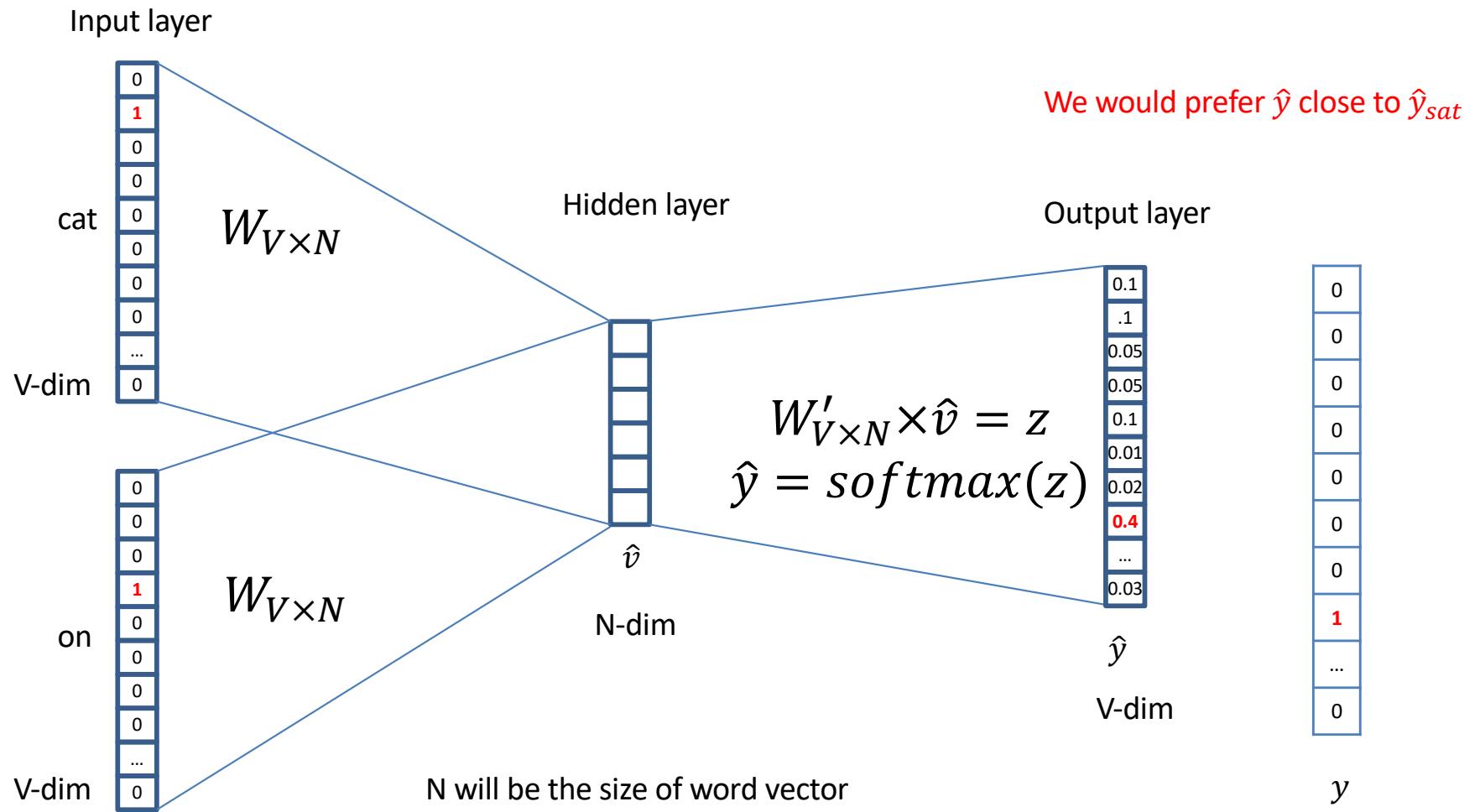


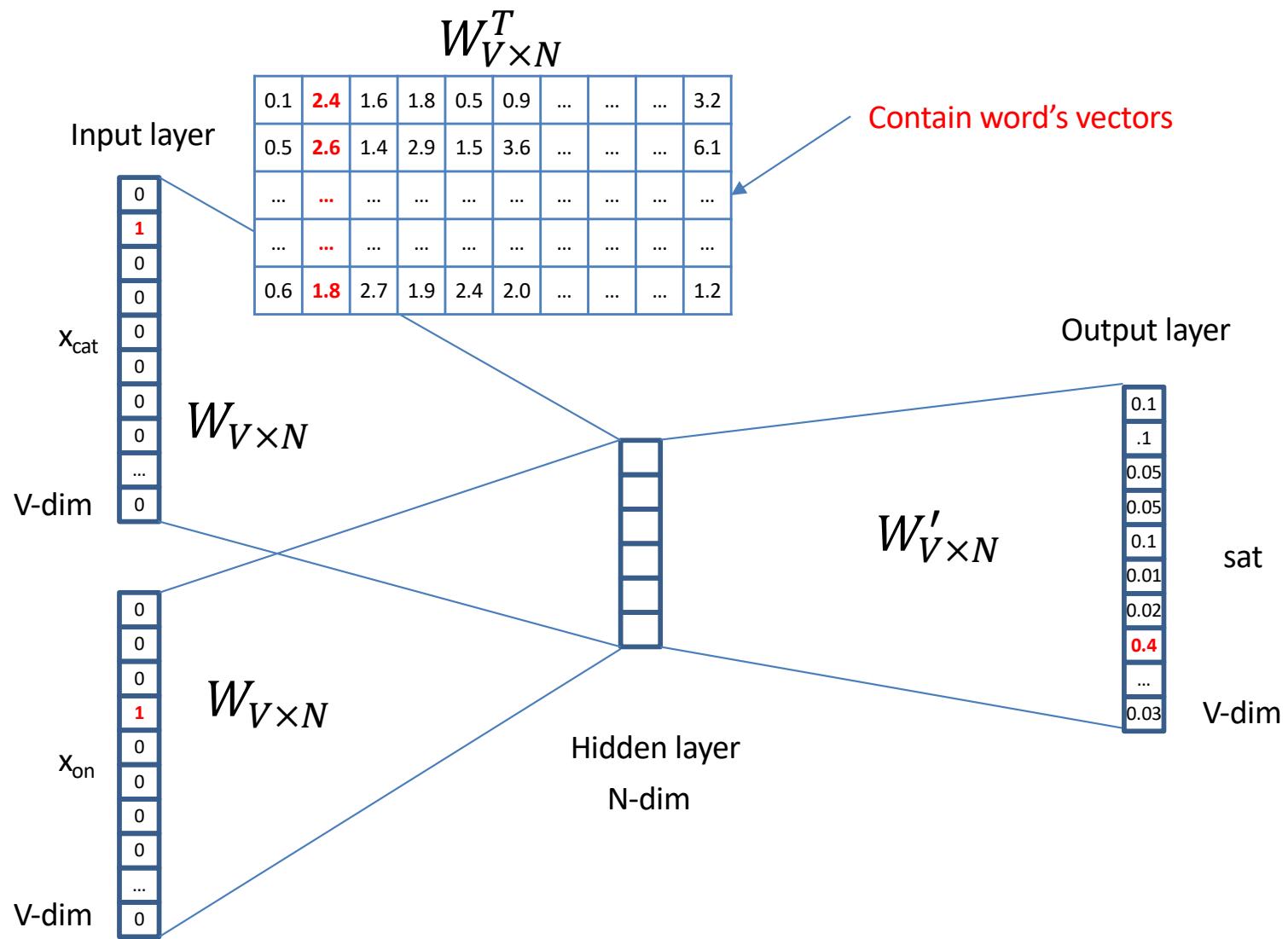
$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$$

Softmax turns the vector into probabilities



$$\hat{y} = softmax(z)$$





We can consider either W or W' as the word's representation. Or even take the average.

Word Analogies

Test for linear relationships, examined by Mikolov et al. (2014)

$$a:b :: c:?$$



$$d = \arg \max_x \frac{(w_b - w_a + w_c)^T w_x}{\|w_b - w_a + w_c\|}$$

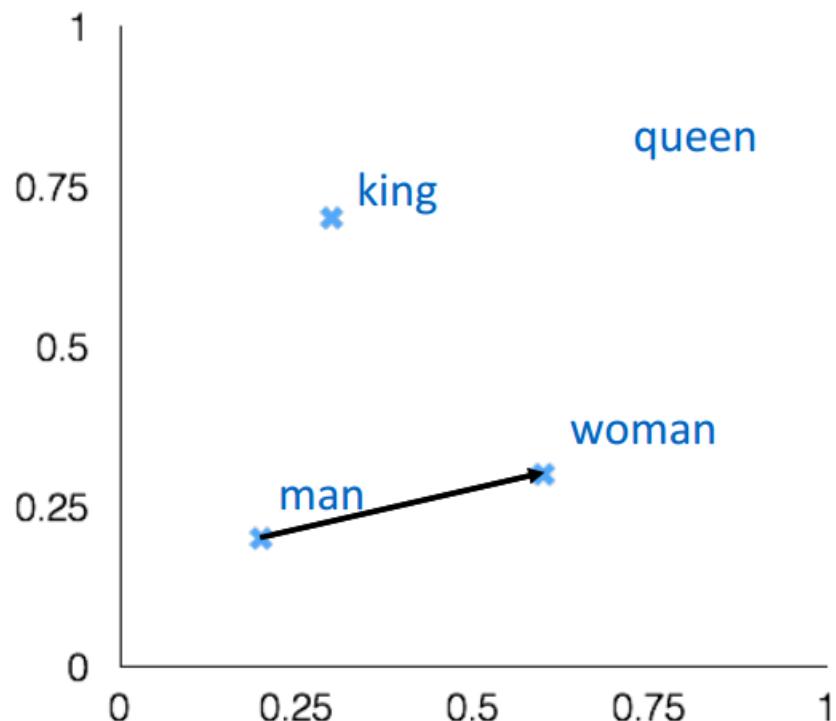
man:woman :: king:?

+ king [0.30 0.70]

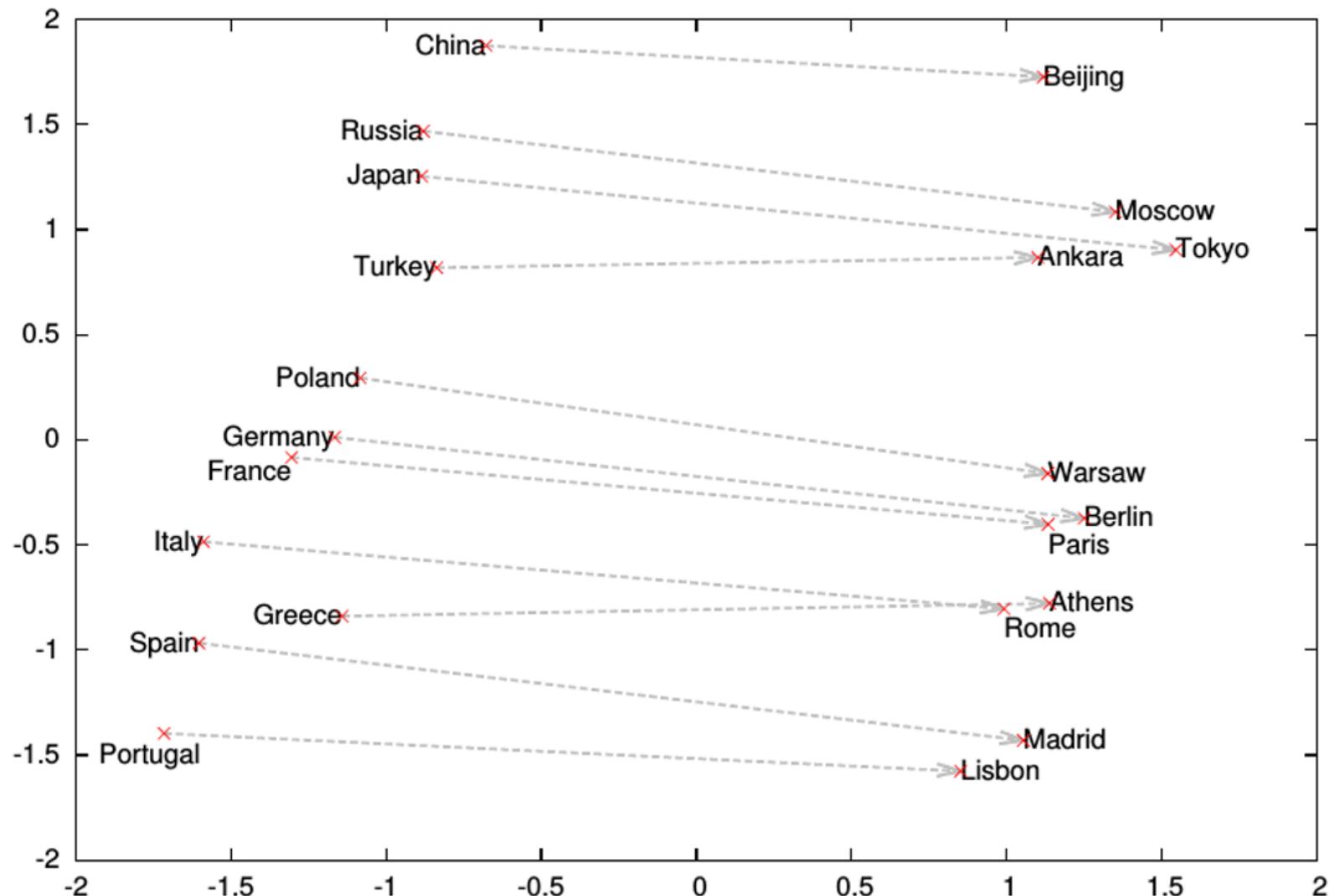
- man [0.20 0.20]

+ woman [0.60 0.30]

queen [0.70 0.80]



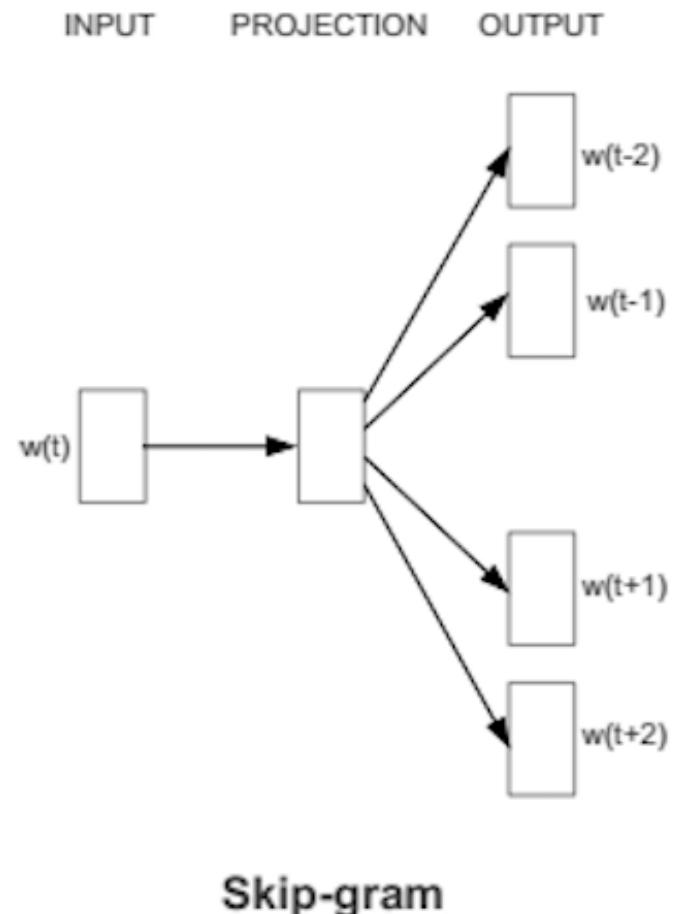
Word analogies



Demo!

Skip gram

- It's a different architecture for implementing word2vec
- Key idea: **Use representation of the input word to predict the context**
 - (This is reflection of CBOW, which uses the context to predict the word)
- Start with a single word embedding and try to predict the surrounding words.
- How can this possibly work??? The embedding is what we're trying to find!



Skip Gram (window 2)

Sam	likes	Celine	Dion	and	biking
-----	-------	--------	------	-----	--------

Sam	likes	Celine	Dion	and	biking
-----	-------	--------	------	-----	--------

Sam	likes
Sam	Celine

Sam	likes	Celine	Dion	and	biking
-----	-------	--------	------	-----	--------

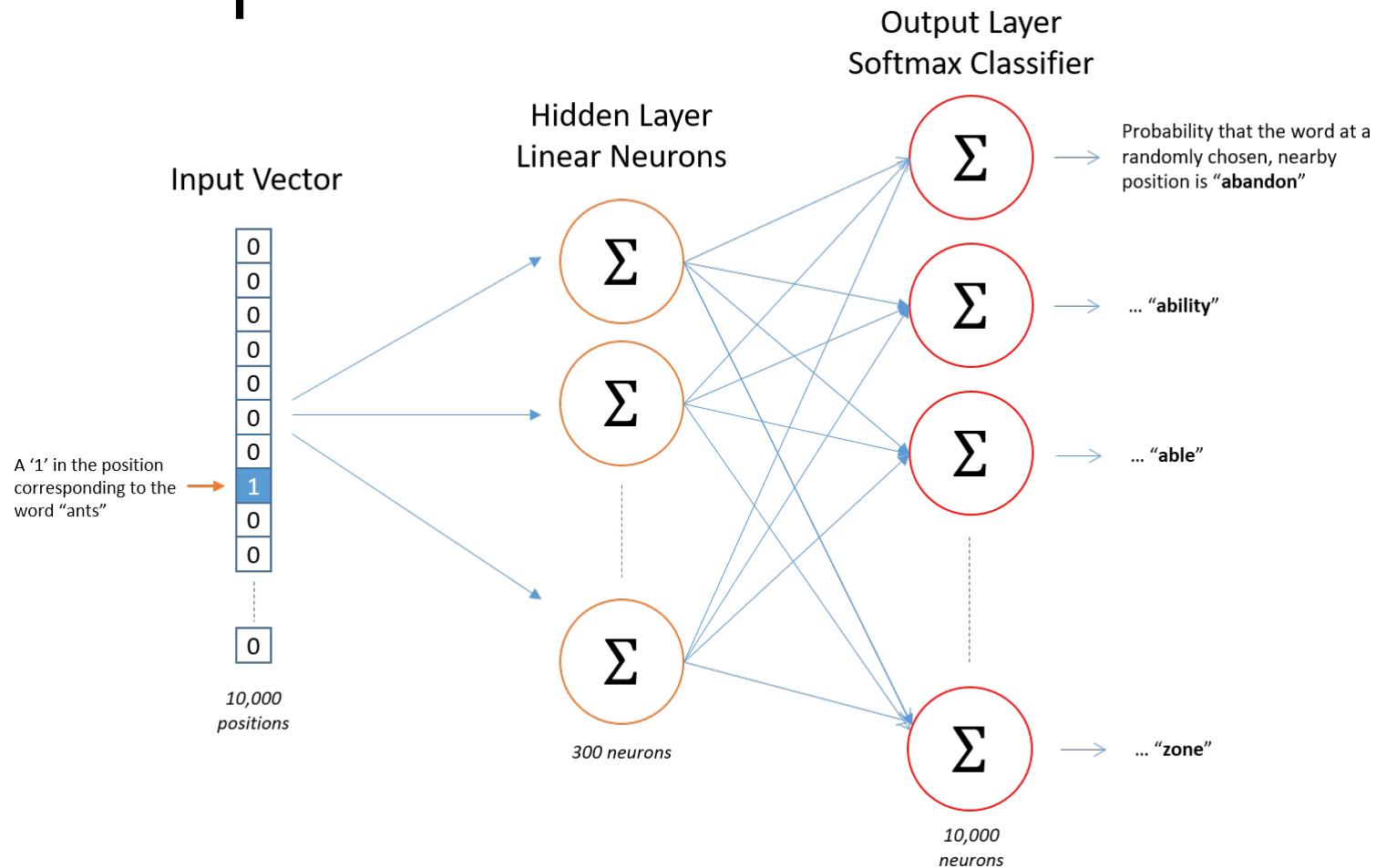
Sam	likes
Sam	Celine
likes	Sam
likes	Celine
likes	Dion

Sam	likes	Celine	Dion	and	biking
-----	-------	--------	------	-----	--------

Sam	likes
Sam	Celine
likes	Sam
likes	Celine
likes	Dion
Celine	Sam
Celine	likes
Celine	Dion
Celine	and

First, generate a lot of target input/output pairs

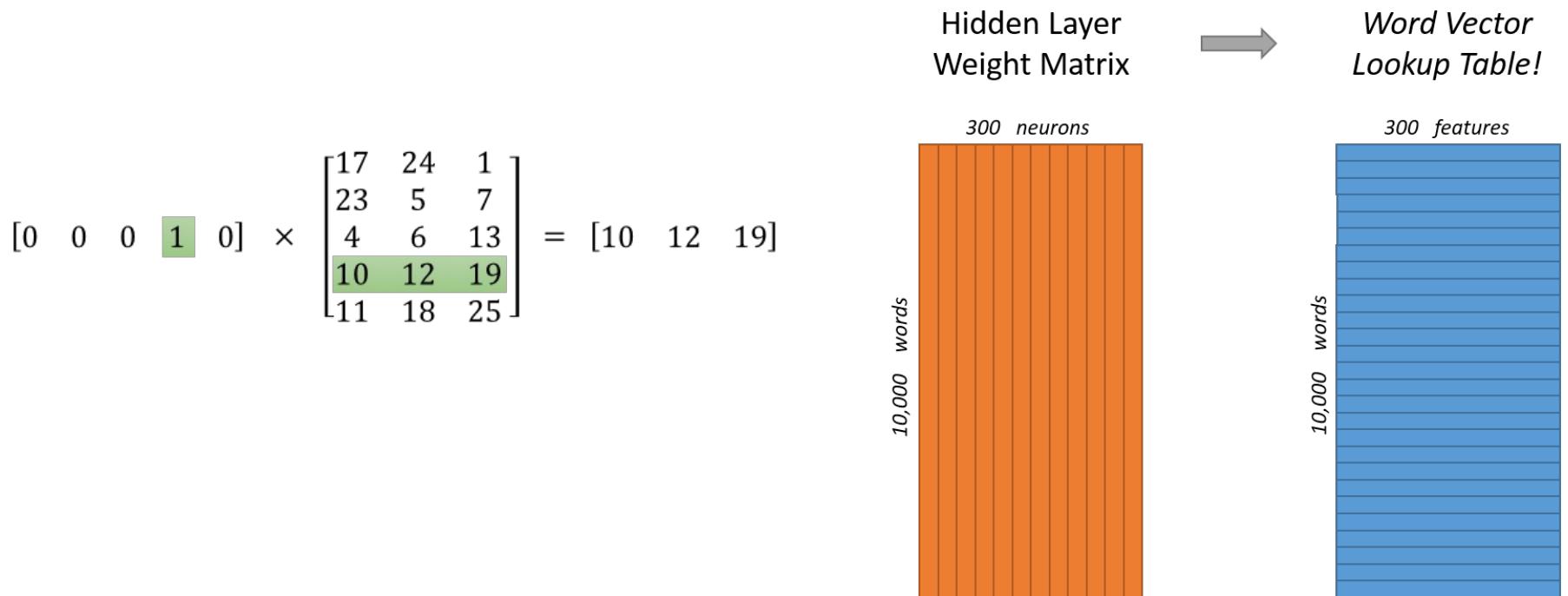
Skip Gram Architecture



- Input is a one-hot encoding of the word. It's size $1 \times V$.
- Hidden layer has size $V \times E$ (where E is the embedding size, a hyperparameter)
- Output layer is size $1 \times E$, which we feed into softmax to obtain probabilities
- We have a target value for the output layer (from previous training data step)

Skip gram example

- Vocabulary of 10,000 words.
- Embedding vectors with 300 features.
- So the hidden layer is going to be represented by a weight matrix of size 300 with 10,000 rows
- Note that if we want to emit similar results for two words that appear in similar contexts, their vectors must be similar



CBOW vs Skip gram

- CBOW needs less data, typically worse for rare words
- Skip gram needs more data, better for rare words

Word2vec shortcomings

- **Problem:** 10,000 words and 300 dim embedding gives a large parameter space to learn. And 10K words is minimal for real applications.
- Slow to train, and need lots of data, particularly to learn uncommon words.

Any ideas how to make the approach more scalable?

Word2vec improvements: word pairs and phrases

- **Idea:** Treat common word pairs or phrases as single “words.”
 - E.g., Boston Globe (newspaper) is different from Boston and Globe separately. Embed Boston Globe as a single word/phrase.
- **Method:** make phrases out of words which occur together often relative to the number of individual occurrences. Prefer phrases made of infrequent words in order to avoid making phrases out of common words like “and the” or “this is”.
- **Pros/cons:** Increases vocabulary size but decreases training expense.

Word2vec improvements: subsample frequent words

- **Idea:** Subsample frequent words to decrease the number of training examples.
 - The probability that we cut the word is related to the word's frequency. More common words are cut more.
 - Uncommon words (anything < 0.26% of total words) are kept
 - E.g., remove some occurrences of "the."
- **Method:** For each word, cut the word with probability related to the word's frequency.
- **Benefits:** If we have a window size of 10, and we remove a specific instance of "the" from our text:
 - As we train on the remaining words, "the" will not appear in any of their context windows.

Word2vec improvements: selective updates

- **Idea:** Use “Negative Sampling”, which causes each training sample to update only a small percentage of the model’s weights.
- **Observation:** A “correct output” of the network is a one-hot vector. That is, one neuron should output a 1, and *all* of the other thousands of output neurons to output a 0.
- **Method:** With negative sampling, randomly select just a small number of “negative” words (let’s say 5) to update the weights for. (In this context, a “negative” word is one for which we want the network to output a 0 for). We will also still update the weights for our “positive” word.

Agenda

1. Embeddings
2. Applications and Retrieval Augmented Generation
3. Vector Databases

Vector Similarity Applications

- Clustering
- Next word prediction
- Semantic Search
- Translation
- If we can obtain a joint embedding of different input types, then we can do cross-modal similarity
 - Image of a cat should map close to the word “cat”

LLMs

- Their inner workings are too much for today. For the moment, let's consider them only as systems (not as consumers of embeddings themselves)

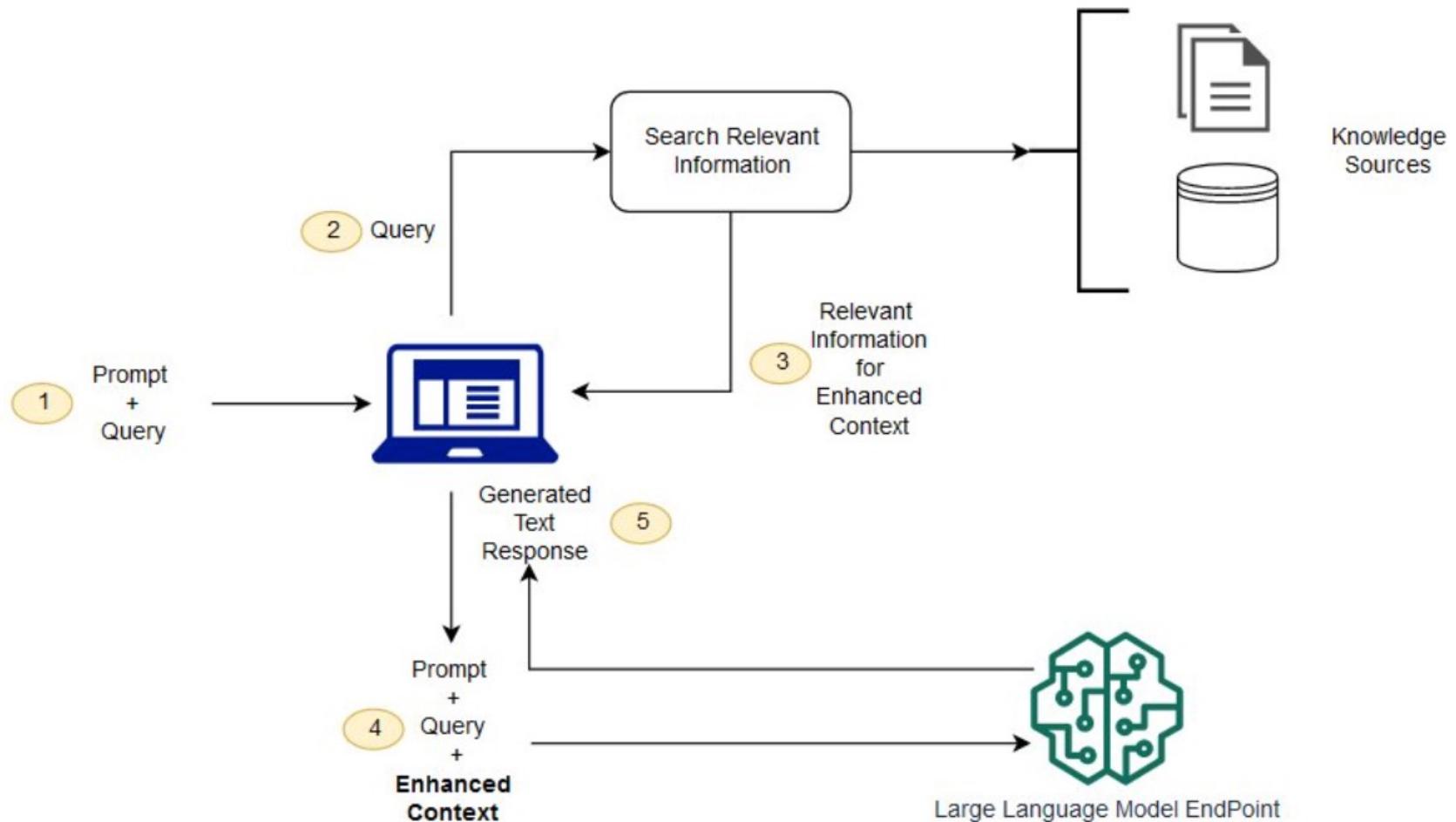
LLMs

- “Write me a poem about data science”
- “Summarize the above paragraph”
- “Why is my printer not working? The answer can be found in one of the above 45,293 technical support articles”
- “Which product is most relevant? <here are 14.3M products>”
- Today’s LLM architectures have limited max context lengths (usually 2k-32k tokens)

Retrieval Augmented Generation

- RAG “pre-builds” the LLM context

<https://docs.aws.amazon.com/sagemaker/latest/dg/jumpstart-foundation-models-customize-rag.html>



Retrieval Augmented Generation

- Simple, but offers lots of advantages!
- Get around the context length limit
- Improve results by editing your article database
- Avoid retraining costs
 - Expensive in general
 - Impossible if the database is changing
- Don't have to include possibly-sensitive data in the training set
- Potentially reduces hallucinations

Other Uses

- Extended LLM “memory”
 - What if you have a ChatGPT conversation that goes very, very long?
 - The chatbot can use RAG to selectively swap in past memories from the record
- LLM answer cache
 - LLM answer generation is extremely computationally expensive
 - When a new query arrives, go back and check if we’ve ever answered something like that before

So What's The Problem?

```
for w in words:  
    curDist = distance(q, w)  
    if curDist < bestDist:  
        bestDist = curDist  
        bestAnswer = w
```

What is `len(words)`?

What if it was `len(products)` instead?

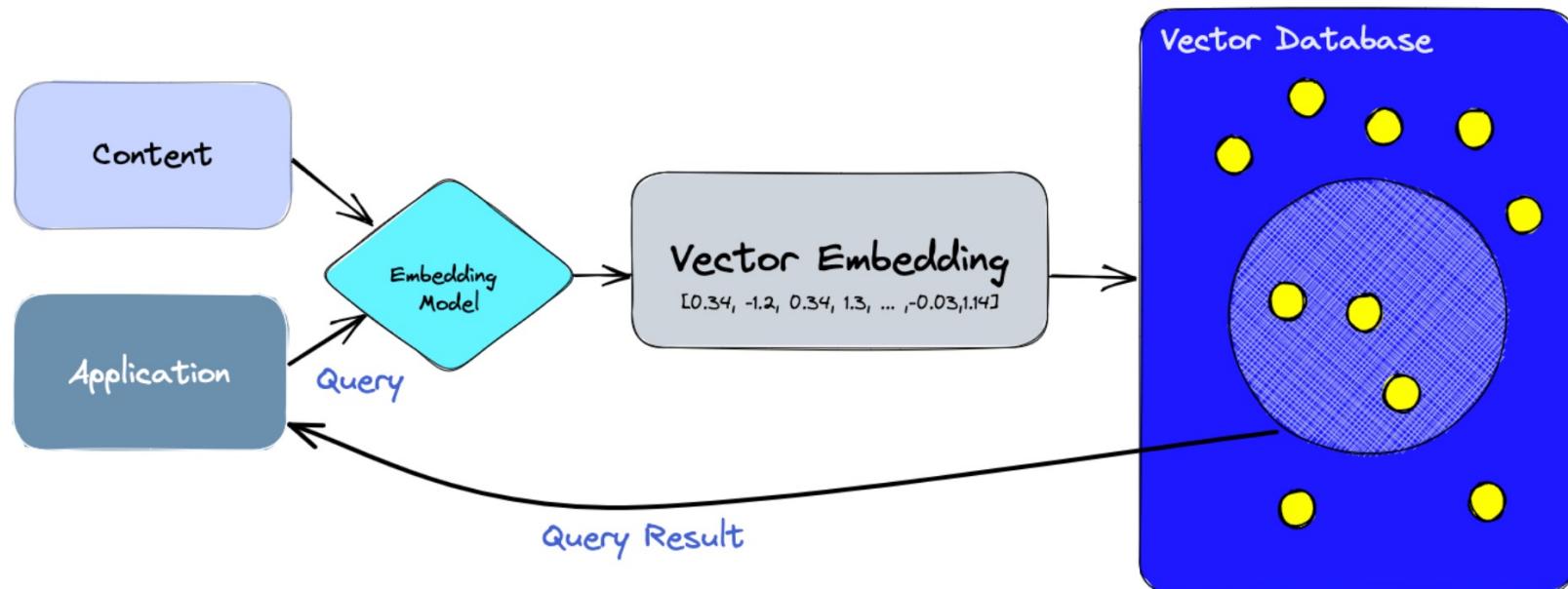
Query Efficiency

- For n points in d dimensions...
 - Linear search for nearest-neighbor is $O(nd)$
- If d is 300, and n is vast, then this is bad
- We need approximate nearest neighbor

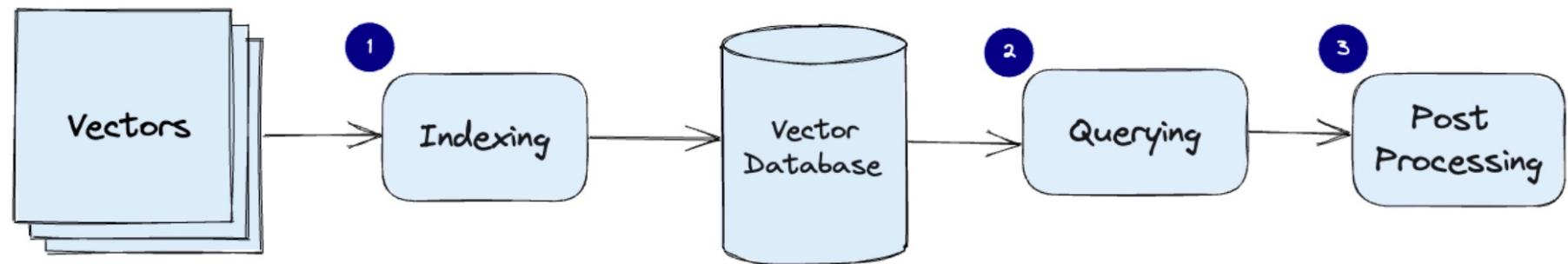
Vector Databases

- Relational databases w/VECTOR datatype
- Allows fast querying over 100Ms or even Bs of data objects
- Handles distributed operation, failure, etc

<https://www.pinecone.io/learn/vector-database/>



RAG Management with Vector Databases



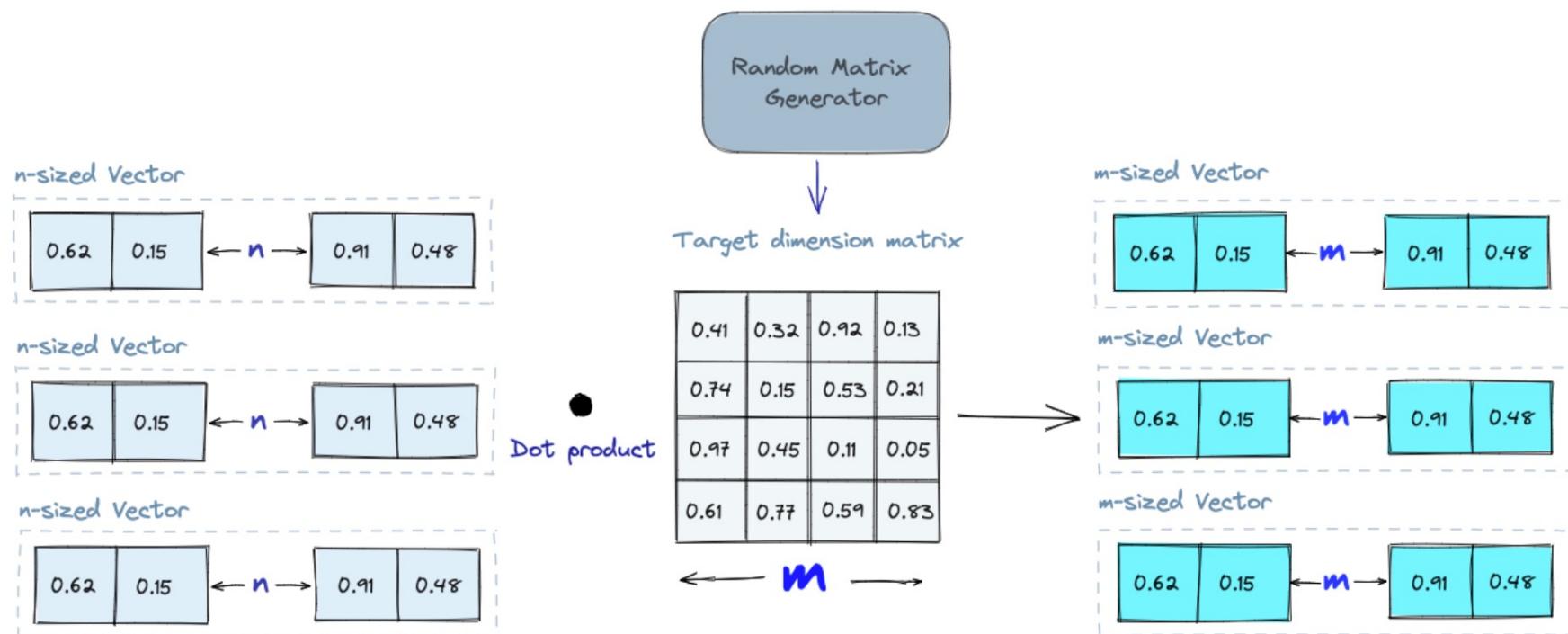
<https://www.pinecone.io/learn/vector-database/>

Query Efficiency

- Three basic ways to make search faster
- Vector Compression
- Hashing
- Indexing

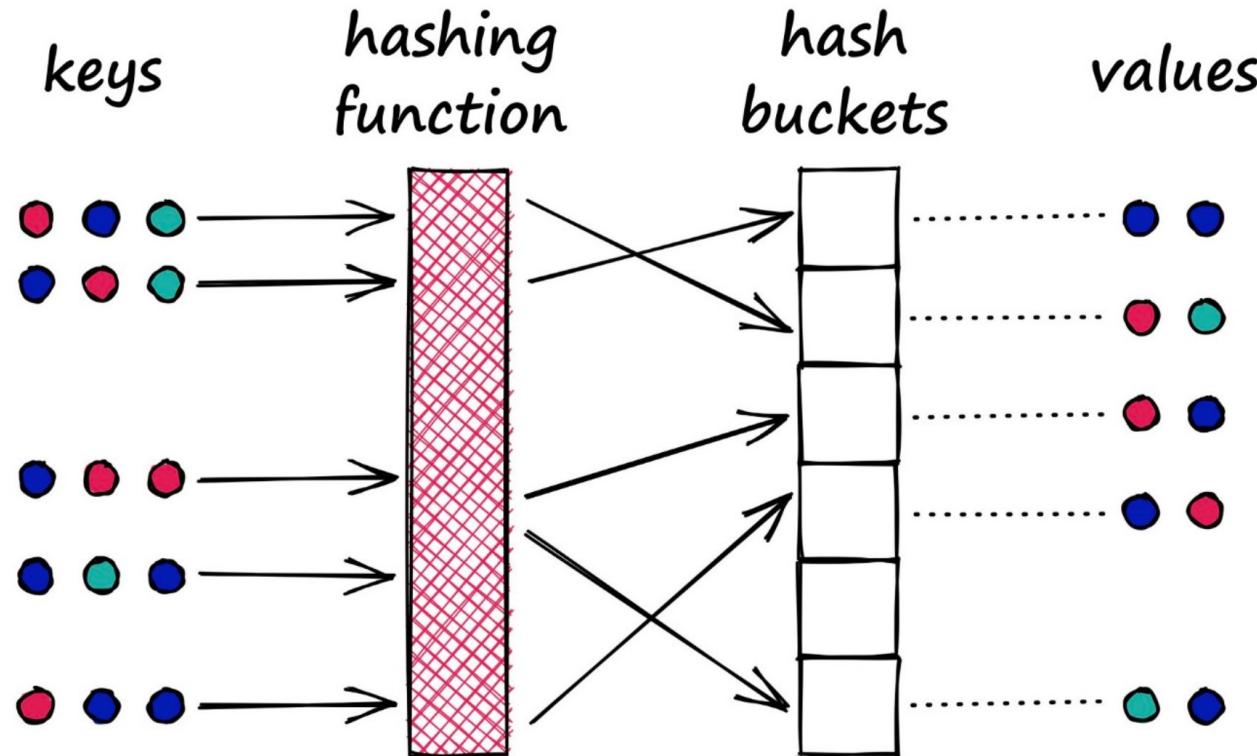
1. Vector Compression

- Reduce work by using lower-dim vectors
- “Random projection” is one simple method



2. Hashing

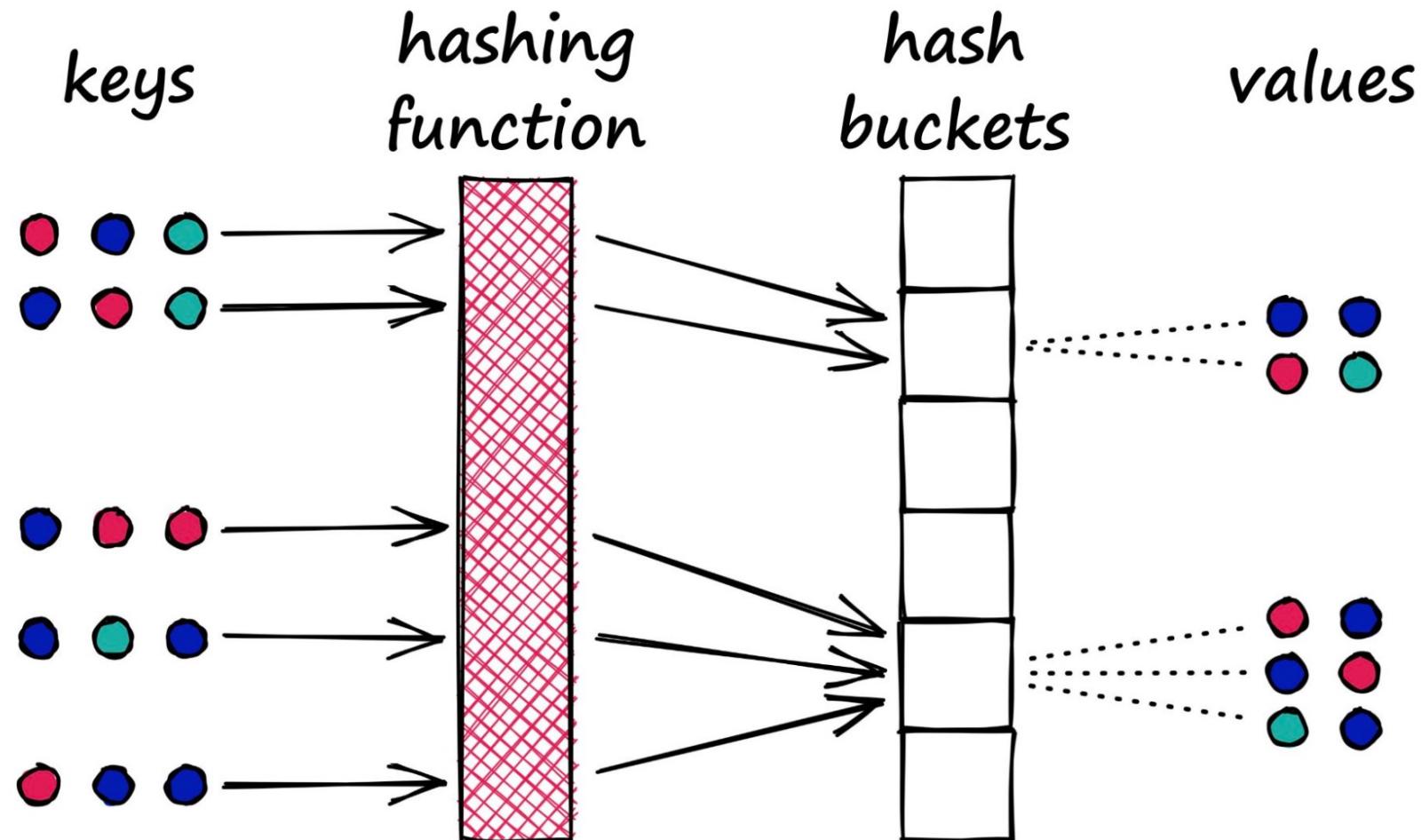
- Standard hashing gives lookups in $O(1)$



- Why won't they work here?
- We need **Locality Sensitive Hashing**

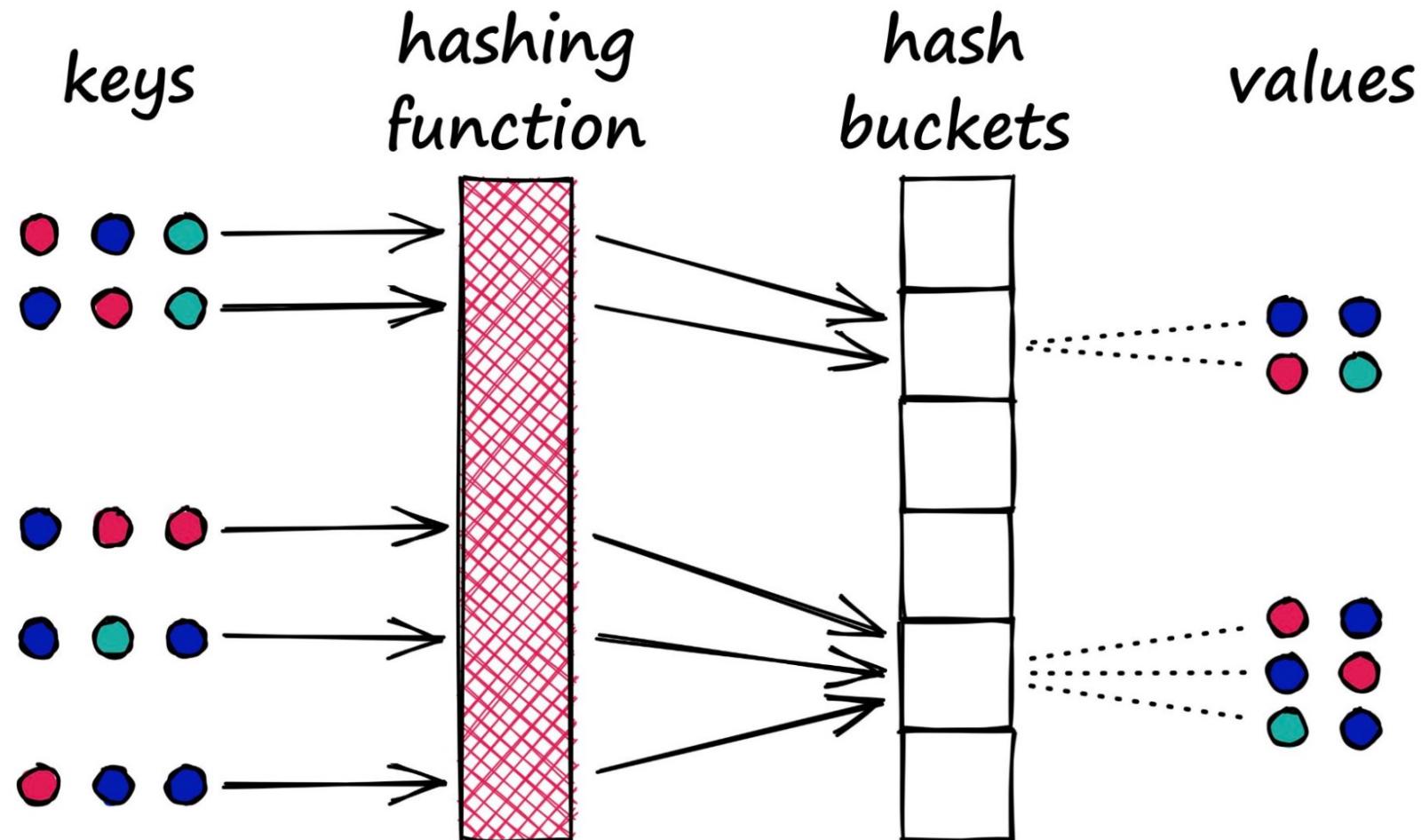
Locality Sensitive Hashing

- Traditional hashing minimizes collisions
- LSH maximizes collisions (when similar)



Locality Sensitive Hashing

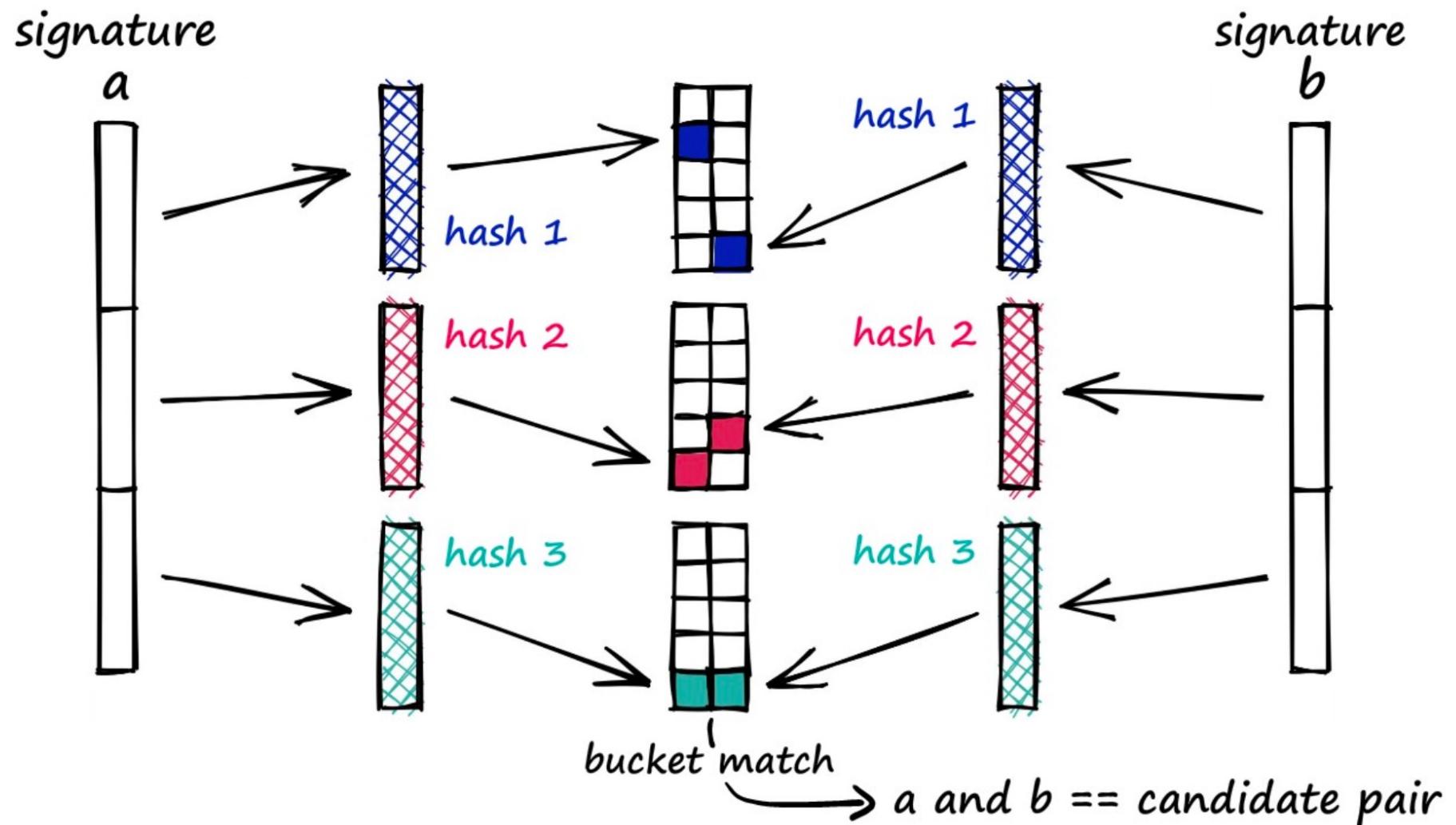
- Traditional hashing minimizes collisions
- LSH maximizes collisions (when similar)



Locality Sensitive Hashing

- When inserting into database...
 - Break input v into subvectors (“signatures”)
 - Hash and store each subvector separately
- When querying database...
 - Break input q into subvectors
 - Hash each subvector separately
 - When q and some data object d share a bucket match, d is a match candidate
 - Compute traditional distance between d and all candidates

Locality Sensitive Hashing



More formally...

- Assume you have a set of hash functions where probability of bucket collision goes up when distance is smaller (aka similarity is greater)
1. Create L hash tables, each with k hash functions
 2. Hash all n d -size vectors into the L tables
 3. For a query q , hash it into the L tables
 4. Get candidates from the tables
 5. Either evaluate every candidate or terminate after “enough” items retrieved

More formally...

- Important parameters L and k
 - L is the number of tables
 - k is the “width” of hash functions associated with each $I \in L$
- Insert time: $O(nLkt)$ (t is hashing time)
- Storage: $O(nL)$
- Larger L and k improve probability of finding a “close” point
- You can tune this probability via L, k

3. Indexing

- The same problems we saw in hashing prevent us from using conventional indexes
 - Vectors can be very similar but not identical
 - How do we order objects in 300 dimensions?
- Let's try **Navigable Small World** graph search instead, for fast approximate nearest neighbor search

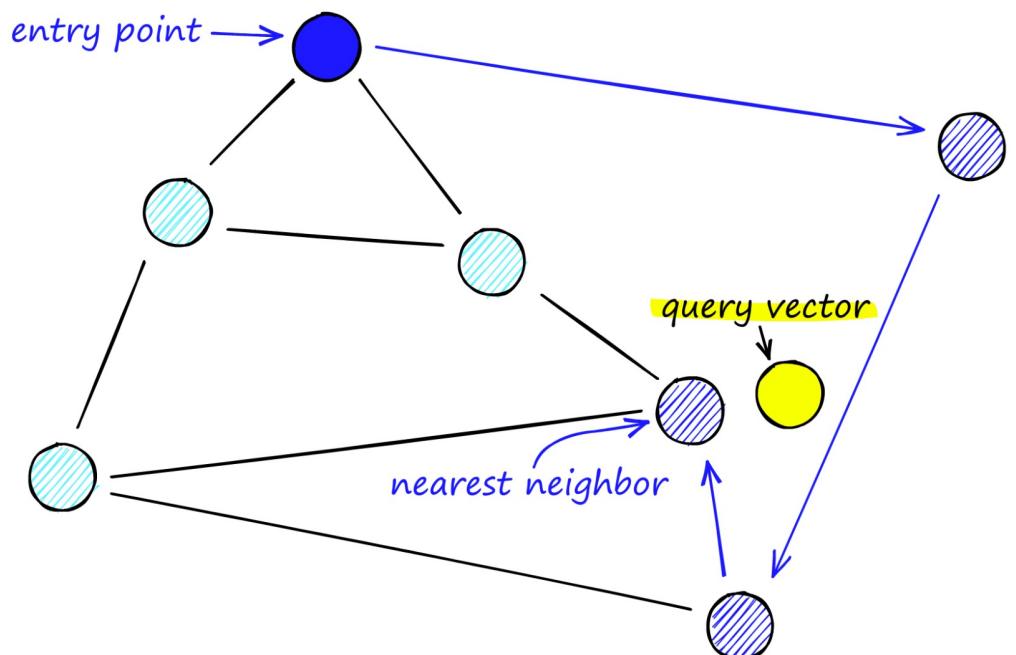
This Slide Is NSW!

```
Greedy_Search(q: object, ventry_point: object)
```

```
1  vcurr←ventry_point;  
2  δmin←δ(q, vcurr); vnext←NIL;  
3  foreach vfriend∈vcurr.getFriends() do  
4    δfr←d(query, vfriend)  
5    if δfr<δmin then  
6      δmin←δfr;  
7      vnext←vfriend;  
8  if vnext=NIL then return vcurr;  
9  else return Greedy_Search(q, vnext);
```

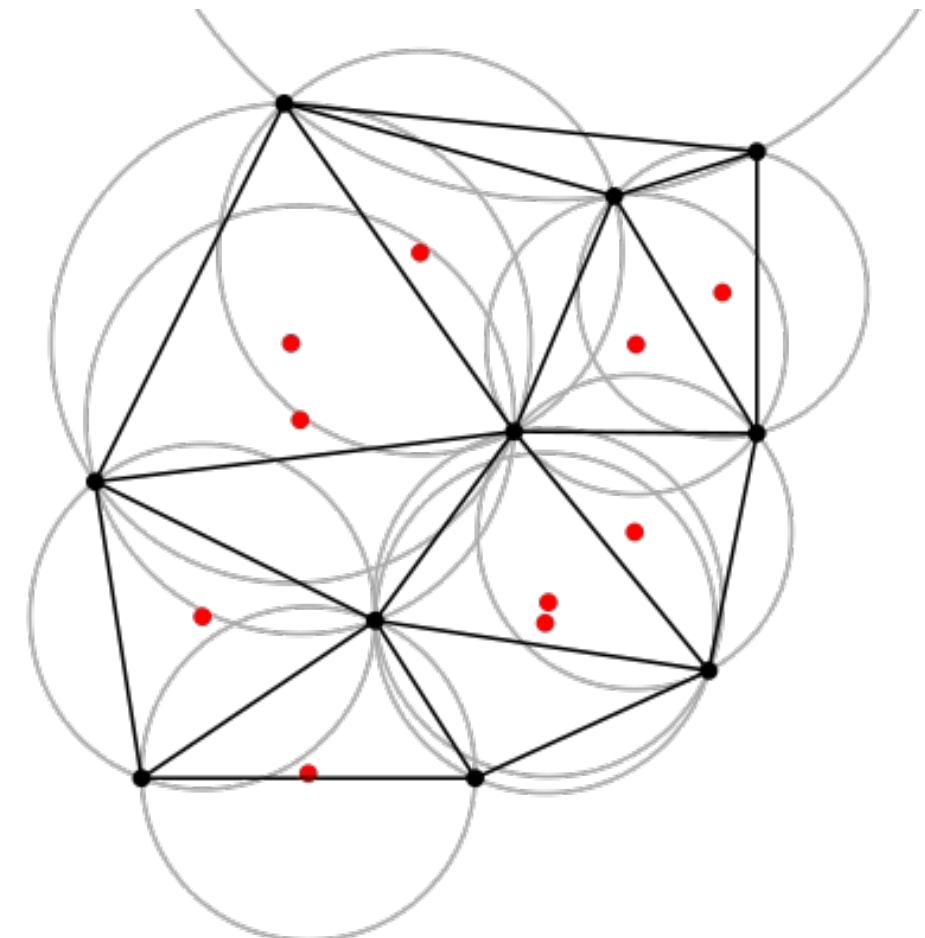
Malkov et al, "Approximate nearest neighbor algorithm based on navigable small world graphs "

Will this always work?



Delaunay Triangulation

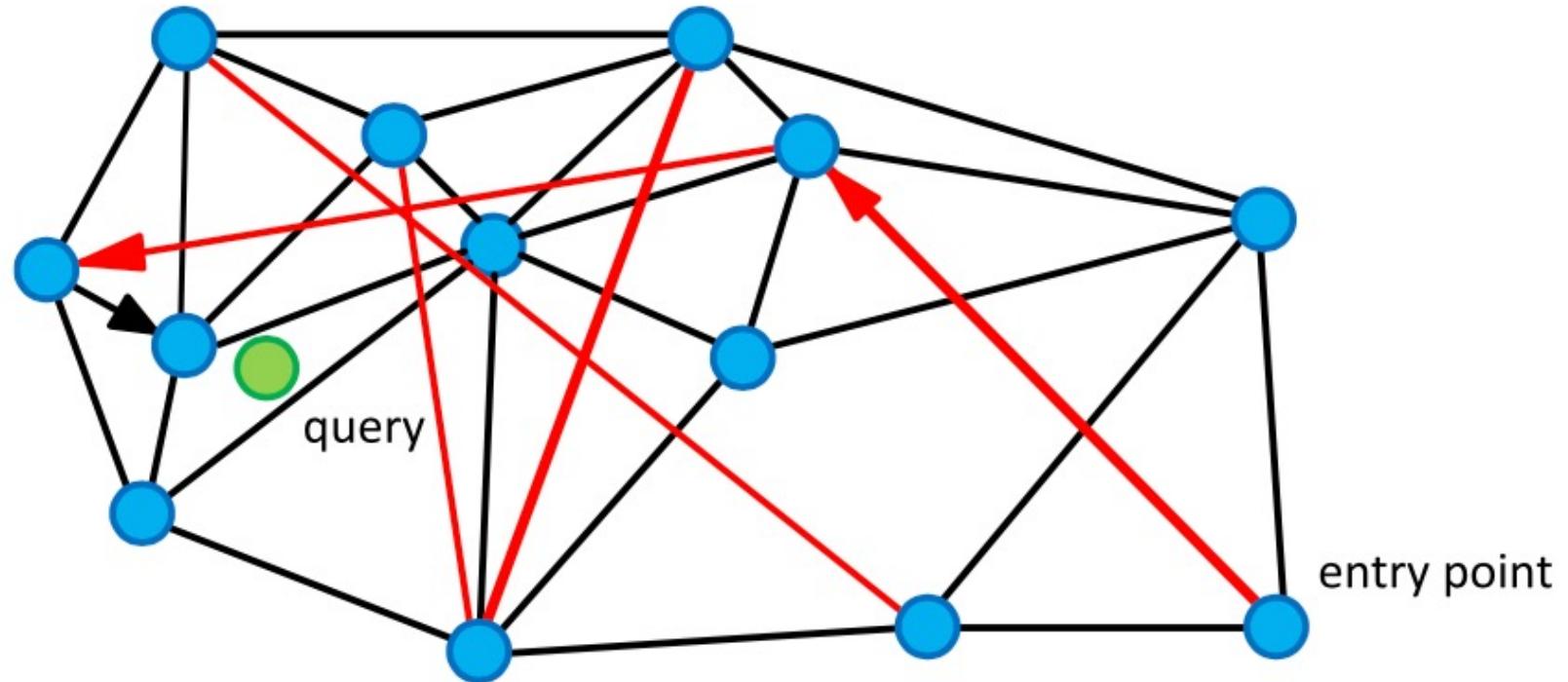
- We are only guaranteed to find the true closest node if each node's neighbors consist of its Voronoi neighbors
- If that doesn't hold, greedy search is approximate



Still, what's the problem?

NSW

- Black links are local; red are “long distance”



Malkov et al, “Approximate nearest neighbor algorithm based on navigable small world graphs”

How does this work?

- The graph is constructed s.t:
- Some links are “short-range” and approximate the Delaunay graph. This provides basic connectivity so greedy search works
- Some links are “long-range” and are used for logarithmic scaling

Insertion Algorithm

- For each new incoming object:
 - Find its set of f closest neighbors. (Try w times)
 - Connect incoming object to every neighbor (and vice versa)
- That's it! Though you have to tune f, w
- Key idea: as objects are inserted, the graph is “filled-in” and old short-range links are “stretched” to become long-range ones
- It's not perfect: f -nearest neighbors aren't guaranteed to give us a Delaunay graph. Increasing w gives more chances to avoid greedy search failure

Malkov et al, “Approximate nearest neighbor algorithm based on navigable small world graphs”

Behavior

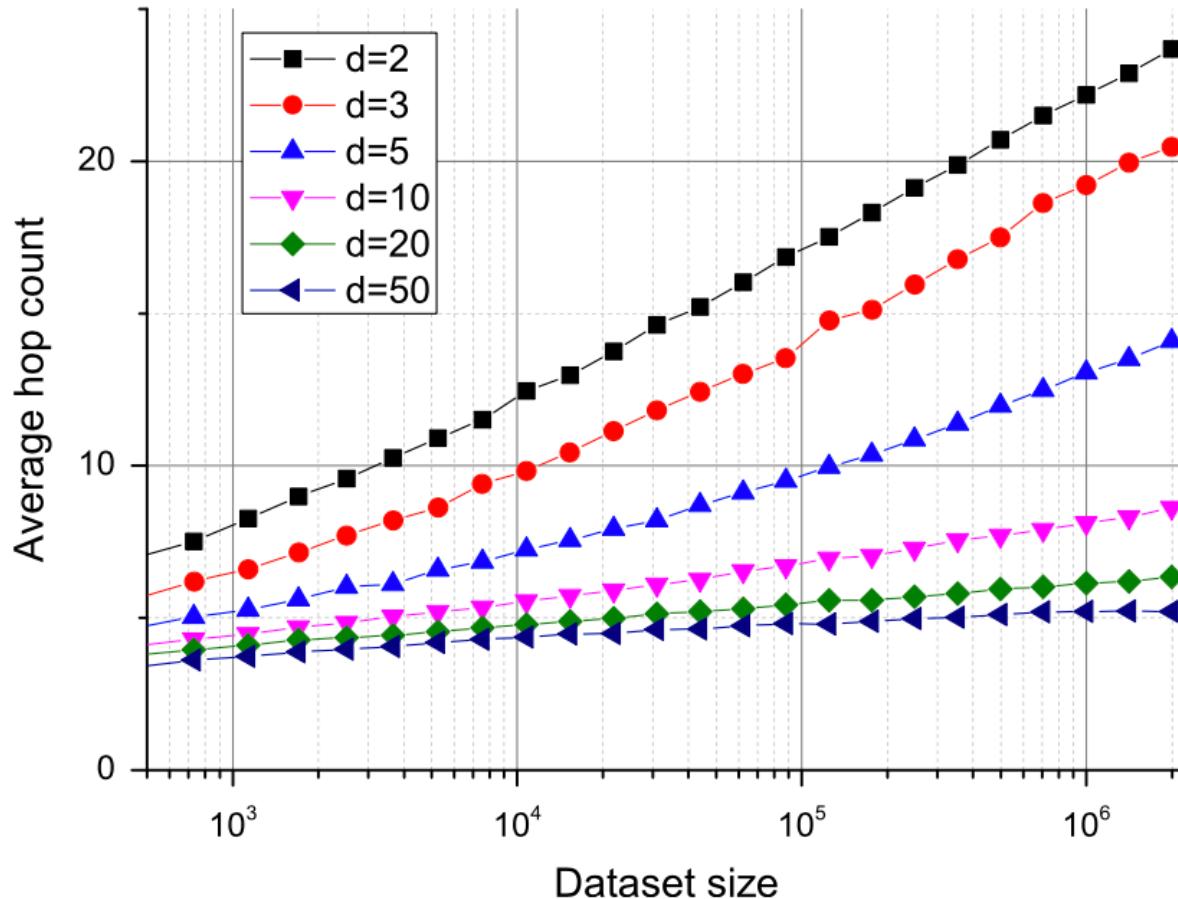


Fig. 2. The average hop count induced by a greedy search algorithm for different dimensionality Euclidean data ($k=10$, $w=20$). The navigable small world properties are evident from the logarithmic scaling.

Malkov et al, “Approximate nearest neighbor algorithm based on navigable small world graphs”

Behavior

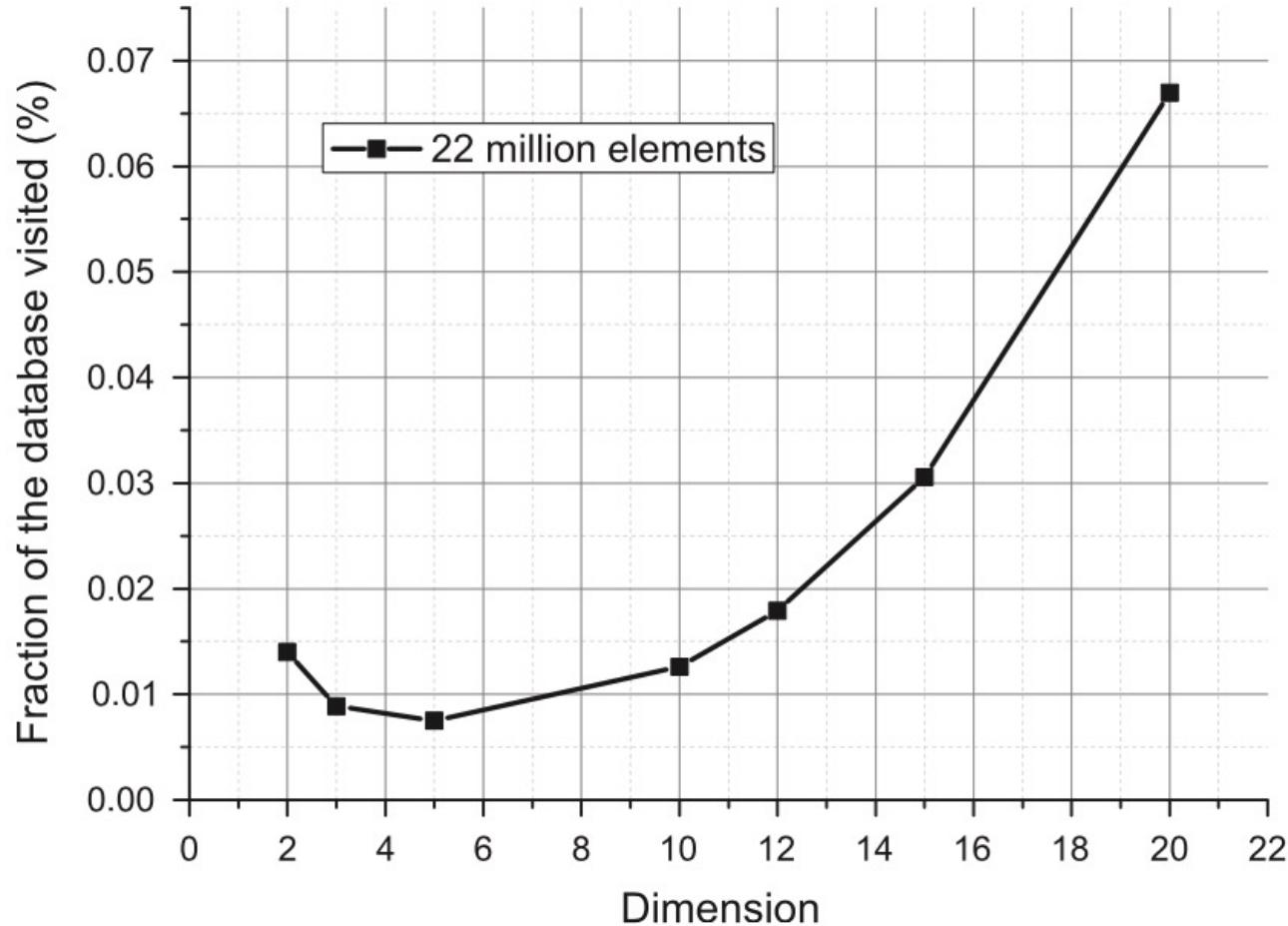
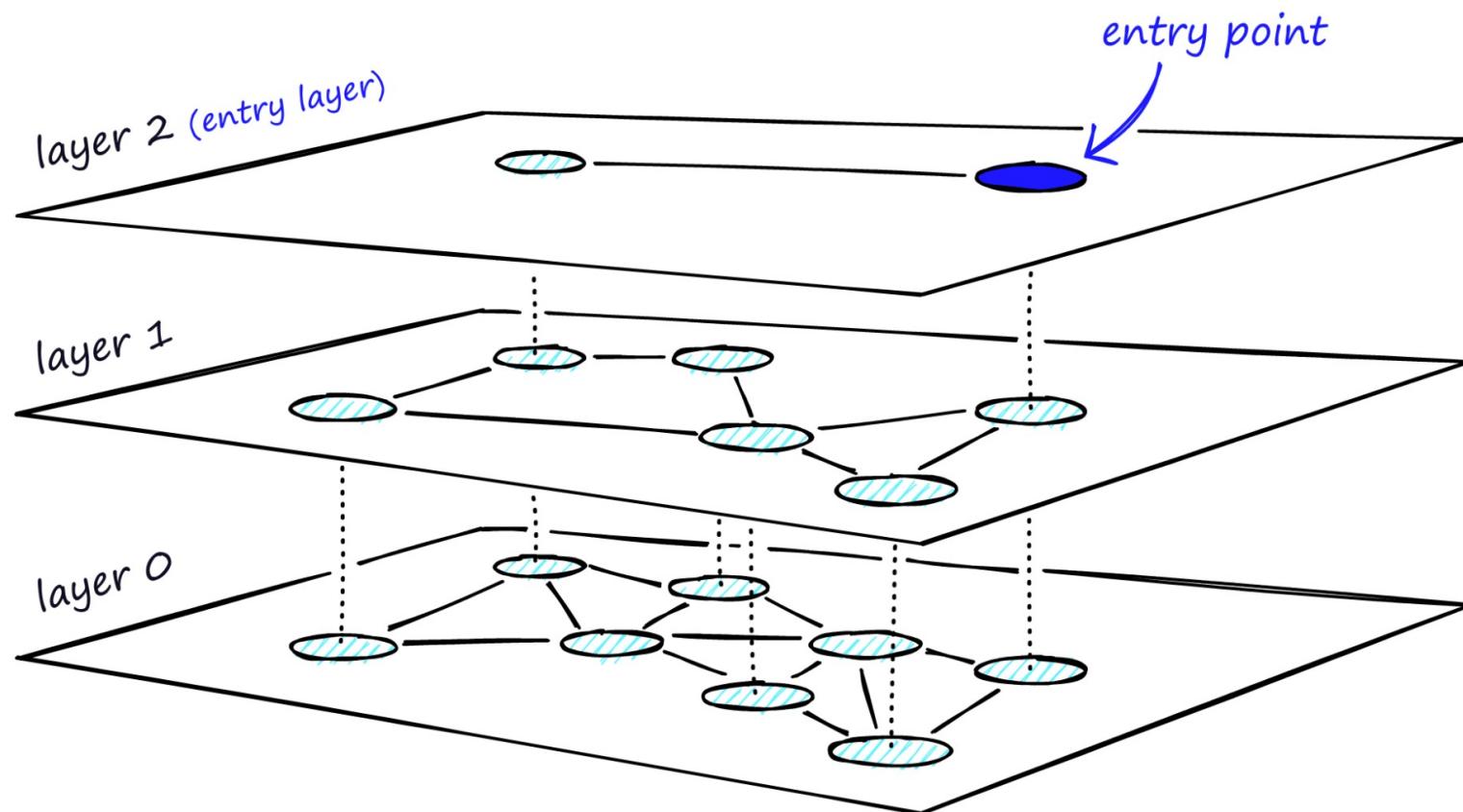


Fig. 5. Average fraction of visited elements within a single 10-NN-search with 0.999 recall for about 22 million elements dataset.

HNSW

- Explicitly hierarchical versions --- where searcher transits different levels --- are also possible



6.S079 FROM EMBEDDINGS TO LANGUAGE

MARCH 19, 2024
MIKE CAFARELLA

THANKS TO MIKEY SHULMAN
FOR SOME SLIDE IDEAS

The Core Thing

- We need a model that can compute $P(\text{someSentence})$
- We can use it to build a sequence model
 - Input: a sequence of tokens
 - Output: a sequence of tokens
- Lots of applications can be built this way

Translation

INPUT: “J’ai vu un chat noir”

OUTPUT A: “I saw a cat black”

vs

OUTPUT B: “I saw a black cat”

Can we compute that $P(A) < P(B)$?

Speech Recognition



*

INPUT: <some waveform>

OUTPUT A: “Yesterday I met Prince Harry”

vs

OUTPUT B: “Yesterday I met prints hairy”

Can we compute that $P(A) > P(B)$?

*The GPT image for “prints hairy” is too weird and alarming to put onscreen

Naive Text Generation

- If you can compute
 $P(\text{nextWord} \mid \text{precedingWords})\dots$
- $P(\text{nextWord} \mid \text{"My favorite food is"})$:
 - nextWord = “pizza”
 - nextWord = “love”
 - nextWord = “antagonist”
- Just try every possible word, pick the best

Computing Sentence Probability

- For a sequence with n words

$$w^n_1 = w_1, w_2, \dots, w_n$$

- Every word w is drawn from a fixed vocabulary

$$\bullet P(w^n_1) = P(w_1)P(w_2|w_1)P(w_3|w^2_1)\dots P(w_n|w^{n-1}_1)$$

$$= \prod_{k=1}^n P(w_k|w_1^{k-1})$$

The Simplest Model

- Where do we get evidence for $P(w_k | w^{k-1}_1)$?
- If you want to be really basic, just ignore the context (that is, a k-gram model where $k=1$)

Or, put another way, assume that

$$P(w_k | w^{k-1}_1) = P(w_k)$$

- Compute as
 $P(w_k) = \text{Count}(w_k) / \text{TotalTrainingWords}$

Expanding Context

- We can do better with more context
- For $k=2$ ("bigrams"), we model
 $P(w_k|w_{k-1}) = P(w_k|w_{k-1})$
- Count the bigram $P(w_k|w_{k-1})$, divide by count of all bigrams starting with w_{k-1}
$$P(w_k|w_{k-1}) = \frac{C(w_{k-1}w_k)}{\sum_{w'} C(w_{k-1}w'_k)}$$
- Use special tokens for start/end sentence

Discussion

- Why does a larger corpus help?
- How big does your corpus have to be?
- How could we tell if corpus was too small?
- Why word-grams? Why not characters?

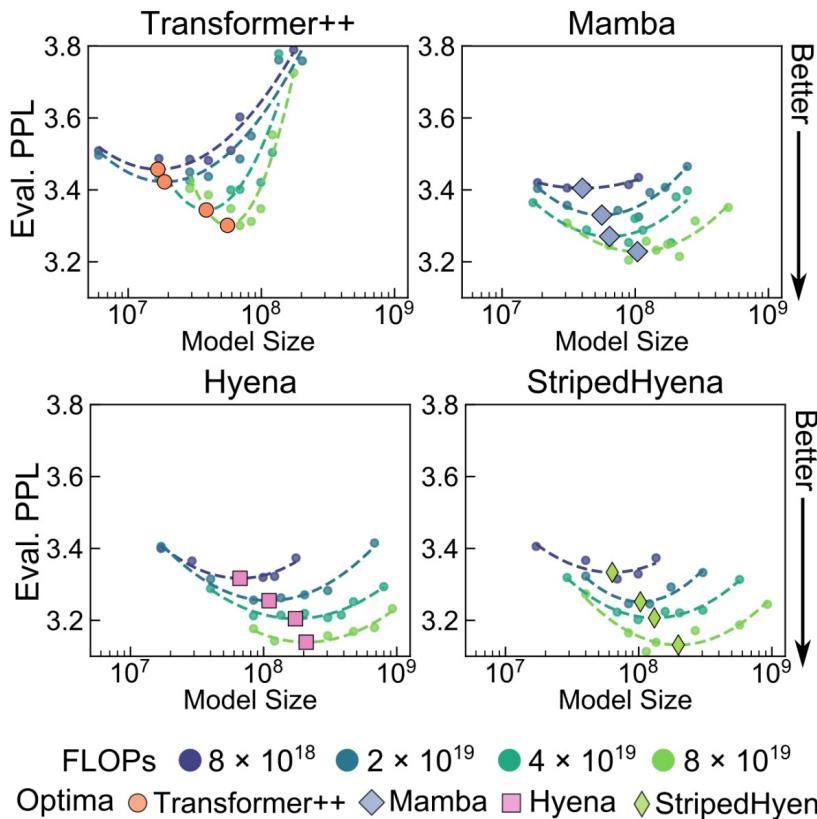
Evaluation

- How can we tell if the model is good?
 - Maybe it helps with a downstream task
 - A general-purpose metric would be nice
- “Perplexity” measures the inverse probability of an unseen test corpus with a particular language model
 - If text is real, then its probability should be high
 - Lower is better

$$PP(W) = P(w_1 \dots w_N)^{-\frac{1}{N}}$$

Perplexity

- What's nice about Perplexity?
 - It's easy to compute
 - You don't need a concrete task
 - You don't need to understand the language!



Measures of Perplexity on different language models' predictions of single-nucleotide sequences from prokaryotic genomes

From Ngyuen et al, "Sequence modeling and design from molecular to genome scale with Evo", 2024

Perplexity

- What's bad about Perplexity?
 - It only works if your model gives a real probability (so: no rule-based methods)
 - Can't compare language models with different vocabularies
 - What is a “good” Perplexity number?
- Discussion: is a bigram language model an example of supervised or unsupervised learning?
- Discussion: when might you see overfitting in this setting?

Data, Models, Features

- More context is better
- ...but we will run out of data for statistics when k-grams get big enough
- We need some combination of:
 - More informative features
 - Constrained models to avoid overfitting
- ...but feature engineering for language is extremely hard
- ...and expressivity of the model is hard to engineer

Neural Methods

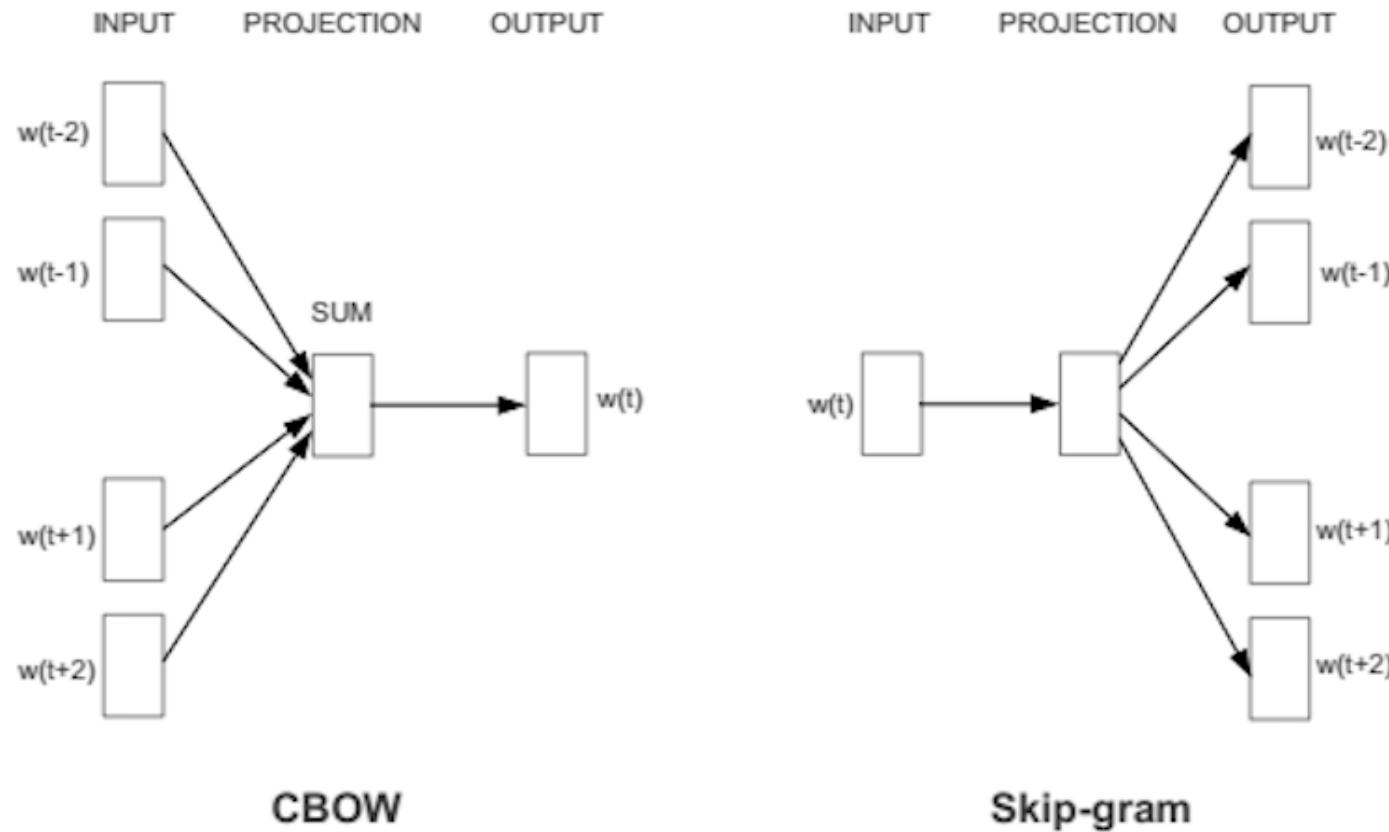
- Neural methods like CBOW let us avoid feature engineering
- Managing overfitting is poorly-understood but works in practice through model architecture, dropout, and other methods

Encoder-Decoders

- Most sequence models use Encoder-Decoder architecture
 - The encoder converts the input into a compressed embedding-style representation
 - The decoder converts an encoded representation back into the target language
- Nice qualities:
 - You can train them separately
 - You can mix/match them for different input and output types
- word2vec has encoder/decoder architecture

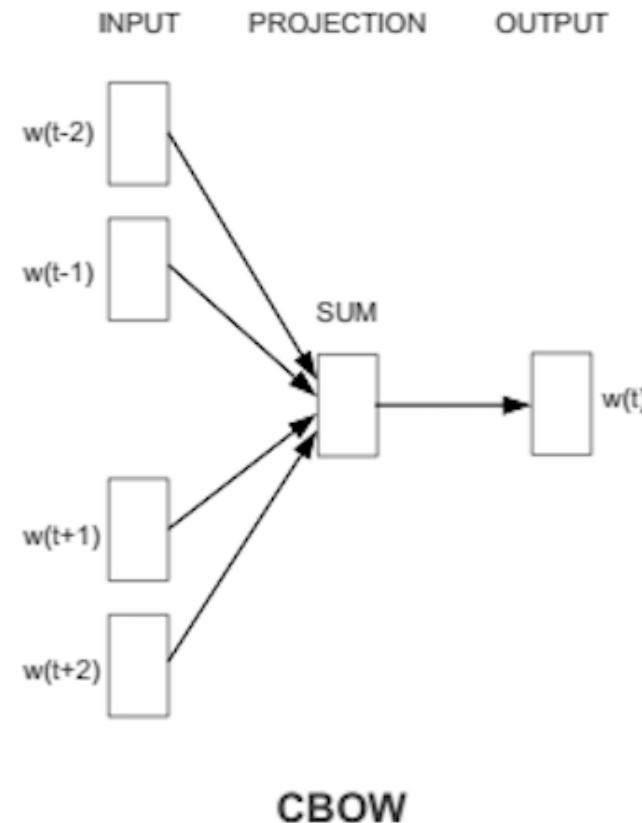
Beyond w2v

- For a chatbot, what's bad about the w2v encoder/decoder architecture?



Beyond w2v

- (Focus on CBOW for now)
- What's bad about this for chat?



Weaknesses of w2v for chat

- Input architecture “looks into the future”
(this is easy to fix)
- Each word has a single embedding,
regardless of usage
 - “I am going to **stick** to it”
 - “I am going to throw the **stick**”
 - The w2v embedding for stick will reflect both
senses, even though in some contexts the
correct sense is obvious to a human
- Can’t handle truly huge vocabularies
- Sentence modeling is very primitive

BERT, ELMO, and the Transformer

- Ideas in these papers led to incredible improvements in the last 7 years
- We'll cover these after break

6.S079 LLMS IN A NUTSHELL

APRIL 2, 2024
MIKE CAFARELLA

Agenda

- Text Modeling
- Sequence Models
- Attention
- Transformers
- Fine-Tuning and RLHF

Agenda

- Text Modeling
- Sequence Models
- Attention
- Transformers
- Fine-Tuning and RLHF

The Core Thing

- We need a model that can compute $P(\text{someSentence})$
- We can use it to build a sequence model
 - Input: a sequence of tokens
 - Output: a sequence of tokens
- Lots of applications can be built this way

Autoregressive Text Generation

- If you can compute
 $P(\text{nextWord} \mid \text{precedingWords})\dots$
- $P(\text{nextWord} \mid \text{"My favorite food is"})$:
 - nextWord = “pizza”
 - nextWord = “love”
 - nextWord = “antagonist”
- Just try every possible word, pick the best

Autoregressive Text Generation

```
currentCtx = getUserInput()

while True:
    w = predictMostLikelyNextWord(currentCtx)
    emit(w)
    currentCtx += w

    if computeTerminationCriterion():
        break
```

Computing Sentence Probability

- For a sequence with n words

$$w^n_1 = w_1, w_2, \dots, w_n$$

- Every word w is drawn from a fixed vocabulary

$$\begin{aligned} P(w^n_1) &= P(w_1)P(w_2|w_1)P(w_3|w^2_1)\dots P(w_n|w^{n-1}_1) \\ &= \prod_{k=1}^n P(w_k|w_1^{k-1}) \end{aligned}$$

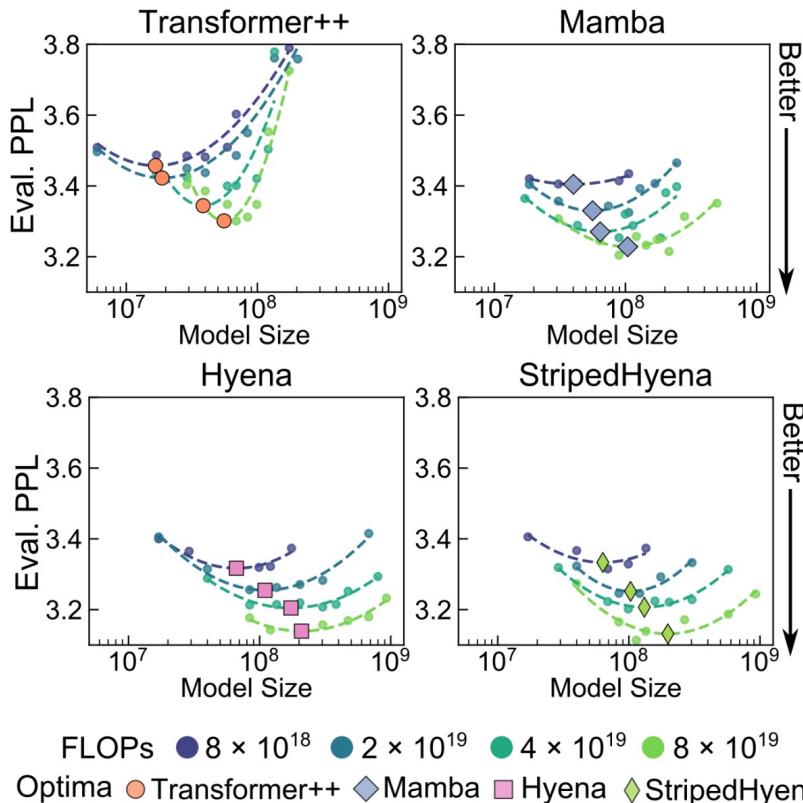
Evaluation

- How can we tell if the model is good?
 - Maybe it helps with a downstream task
 - A general-purpose metric would be nice
- “Perplexity” measures the inverse probability of an unseen test corpus with a particular language model
 - If text is real, then its probability should be high
 - Lower is better

$$PP(W) = P(w_1 \dots w_N)^{-\frac{1}{N}}$$

Perplexity

- What's nice about Perplexity?
 - It's easy to compute
 - You don't need a concrete task
 - You don't need to understand the language!



Measures of Perplexity on different language models' predictions of single-nucleotide sequences from prokaryotic genomes

From Ngyuen et al, “Sequence modeling and design from molecular to genome scale with Evo”, 2024

Perplexity

- What's bad about Perplexity?
 - It only works if your model gives a real probability (so: no rule-based methods)
 - Can't compare language models with different vocabularies
 - What is a "good" Perplexity number?

Data, Models, Features

- More context is better
- ...but we will run out of data for statistics when k-grams get big enough
- We need some combination of:
 - More informative features
 - Constrained models to avoid overfitting
- ...but feature engineering for language is extremely hard
- ...and expressivity of the model is hard to engineer

Neural Methods

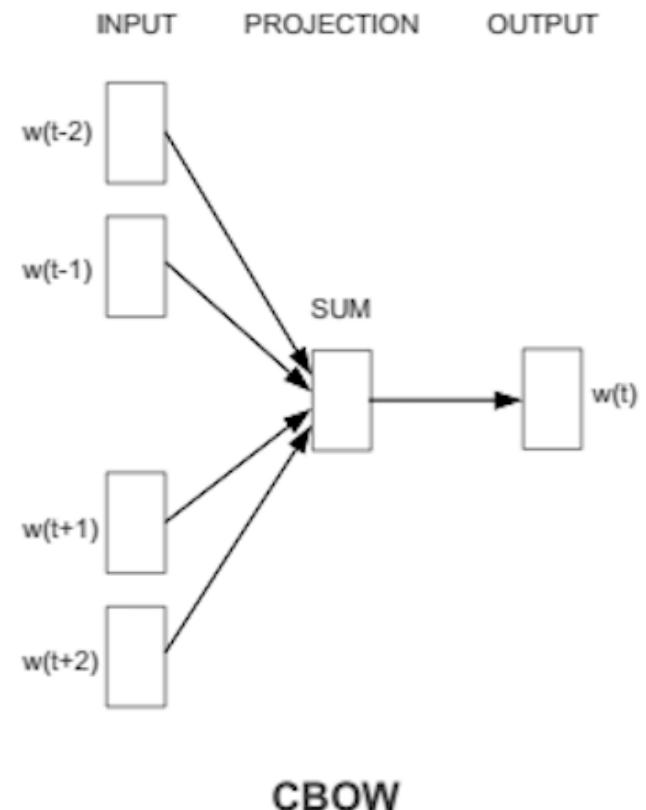
- Neural methods like CBOW let us avoid feature engineering
- Managing overfitting is poorly-understood but works in practice through model architecture, dropout, and other methods

Encoder-Decoders

- Most sequence models use Encoder-Decoder architecture
 - The encoder converts the input into a compressed embedding-style representation
 - The decoder converts an encoded representation back into the target language
- Nice qualities:
 - You can train them separately
 - You can mix/match them for different input and output types
- word2vec has encoder/decoder architecture

Beyond w2v

- Remember CBOW? It aims to predict the output word given its context
- What's bad about this for chat?



Weaknesses of w2v for chat

- Input architecture “looks into the future” (this is easy to fix)
- Each word has a single embedding, regardless of usage
 - “I am going to stick to it”
 - “I am going to throw the **stick**”
 - The w2v embedding for stick will reflect both senses, even though in some contexts the correct sense is obvious to a human
- Can’t handle truly huge vocabularies
- Sentence modeling is very primitive

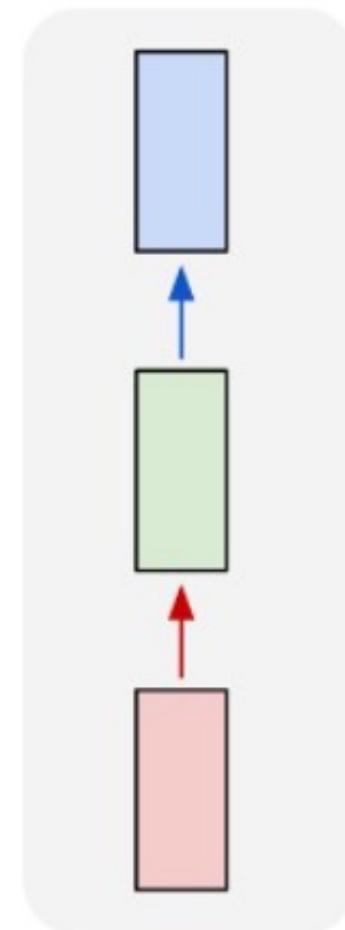
Agenda

- Text Modeling
- Sequence Models
- Attention
- Transformers
- Fine-Tuning and RLHF

Sequence Modeling

- The input/output model of w2v is really basic
 - You put in some fixed-size context
 - You get out a word
- What if input/outputs are variable length?
- What if they have complicated structure?

one to one

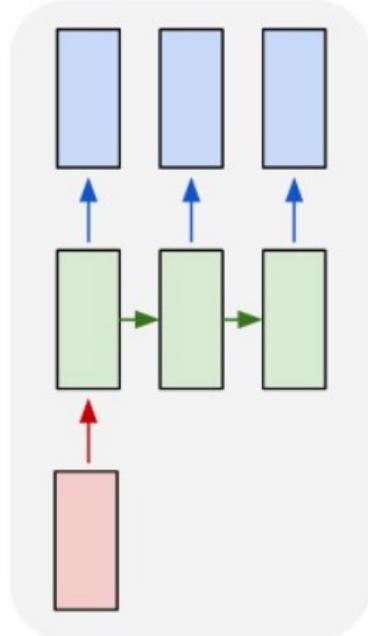


Images thanks to Karpathy, “Unreasonable Effectiveness of Recurrent Neural Networks”

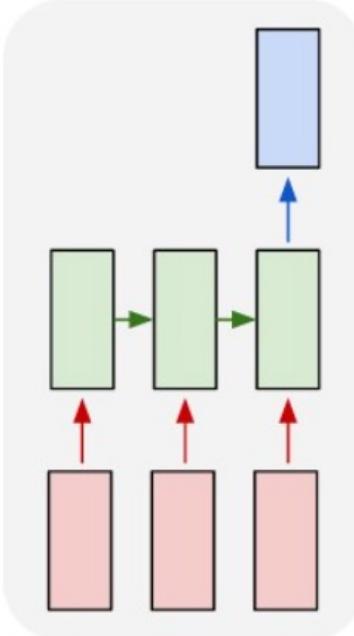
Sequence Modeling

- Sequence models offer more flexibility

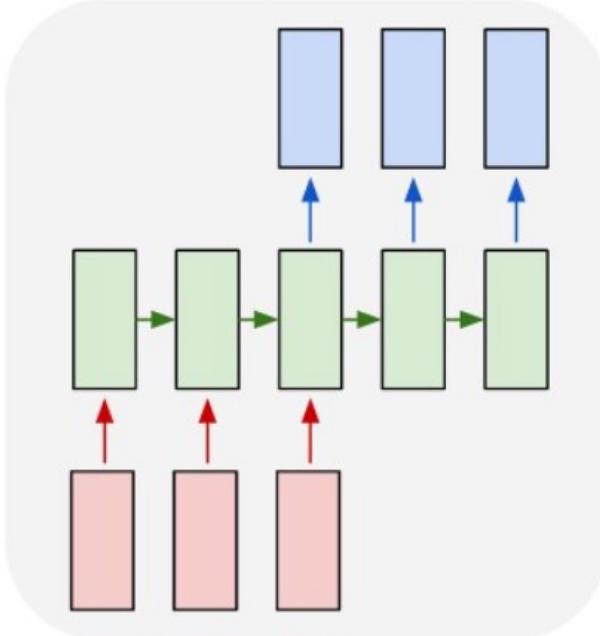
one to many



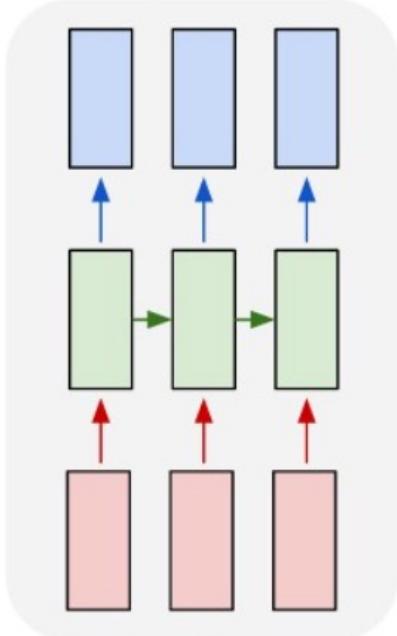
many to one



many to many



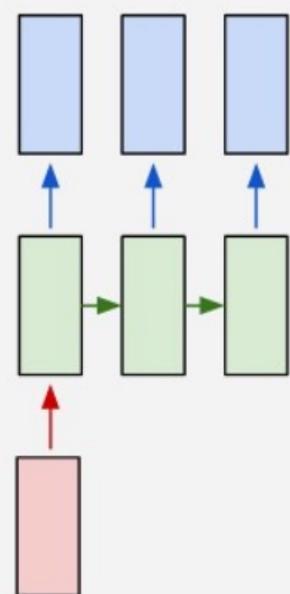
many to many



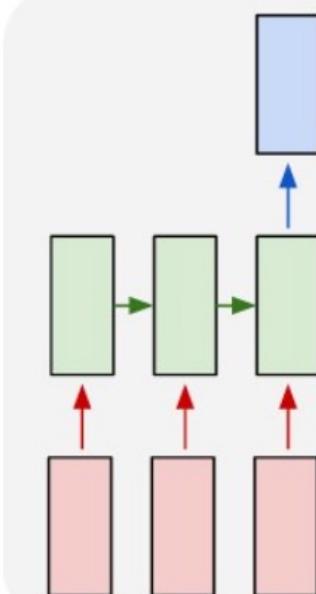
- Red: input vectors, blue: output vectors
- Green: internal state

Sequence Modeling

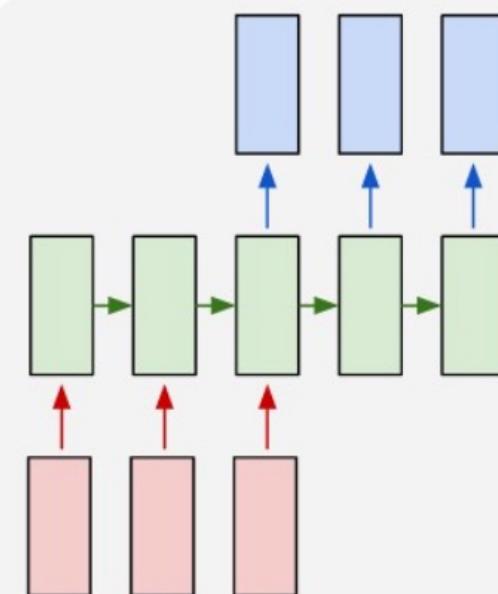
one to many



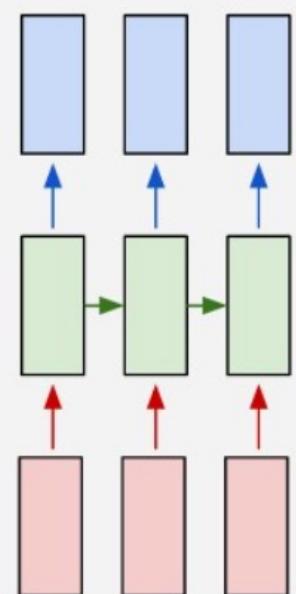
many to one



many to many



many to many

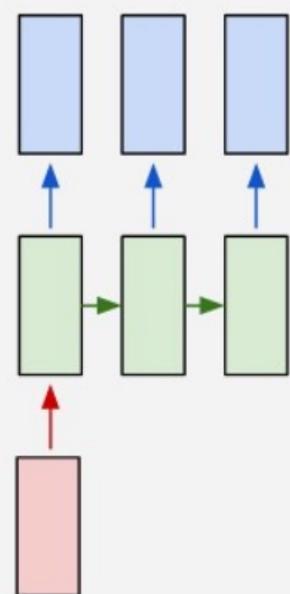


???

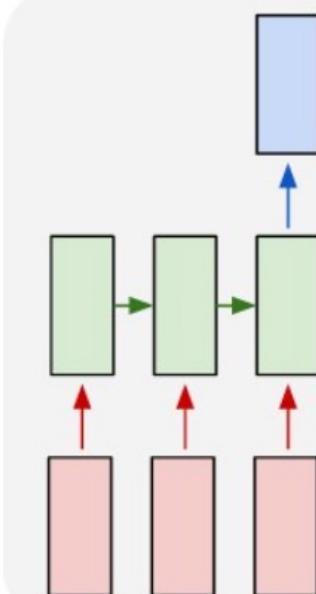
???

Sequence Modeling

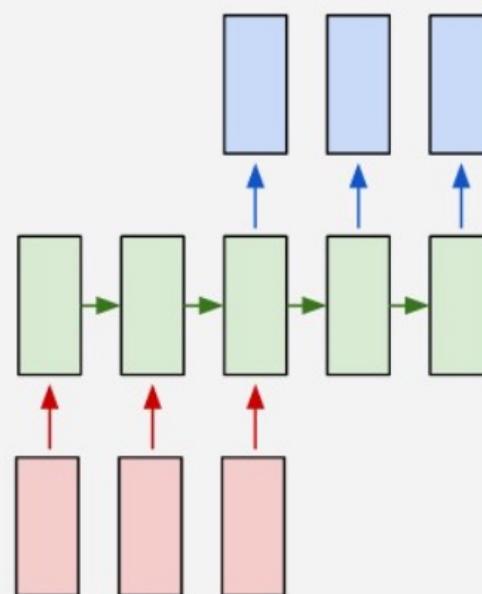
one to many



many to one



many to many



many to many

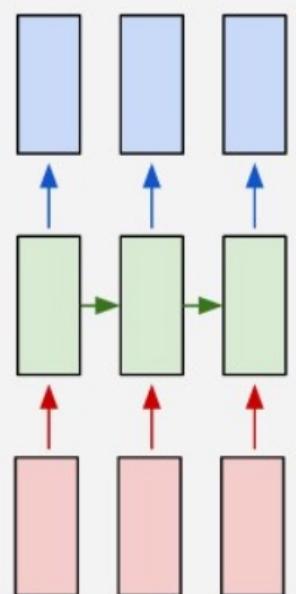
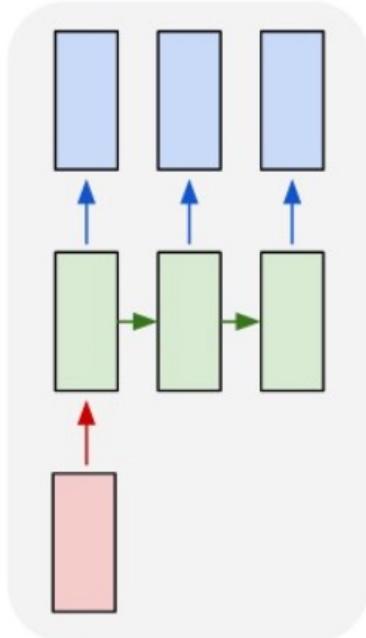


Image
captioning

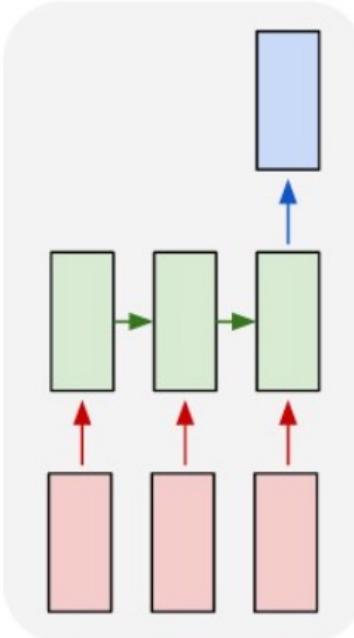
Text
labelling

Sequence Modeling

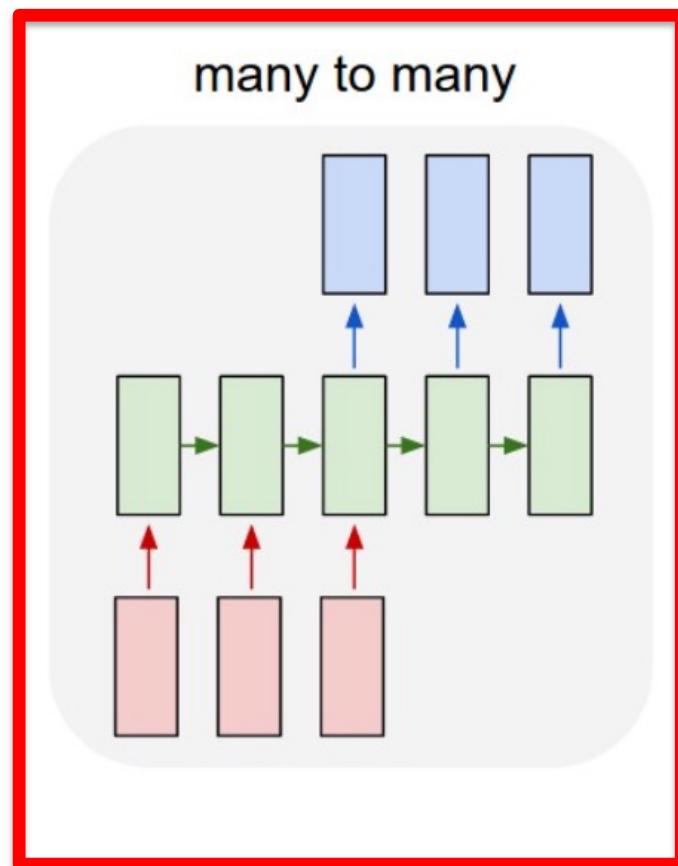
one to many



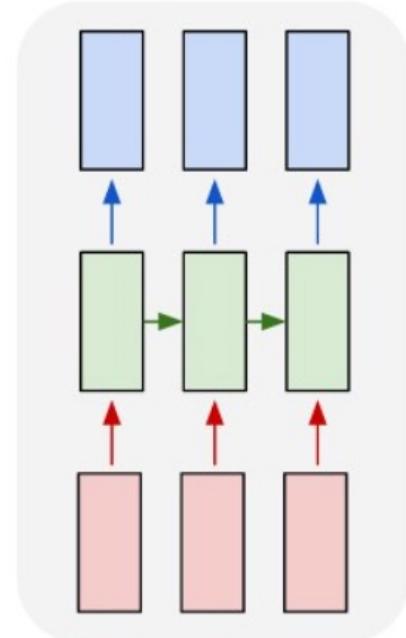
many to one



many to many



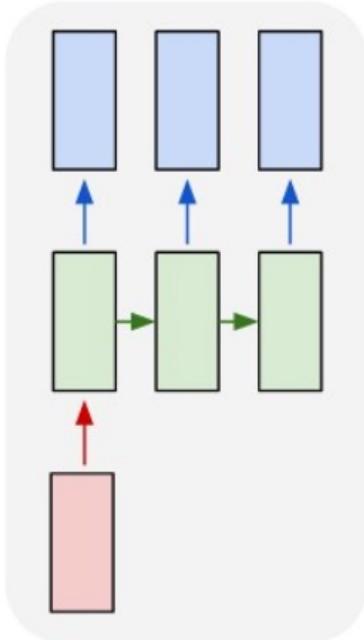
many to many



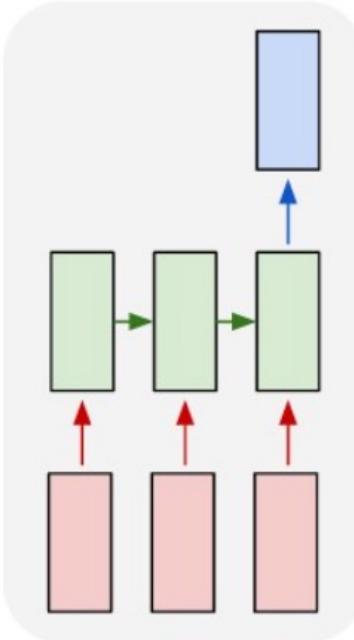
????

Sequence Modeling

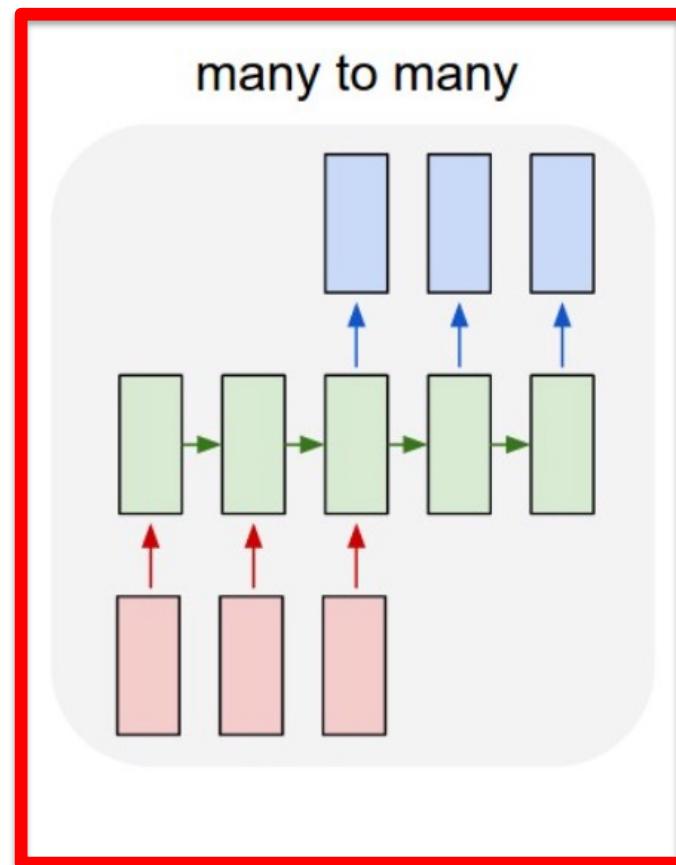
one to many



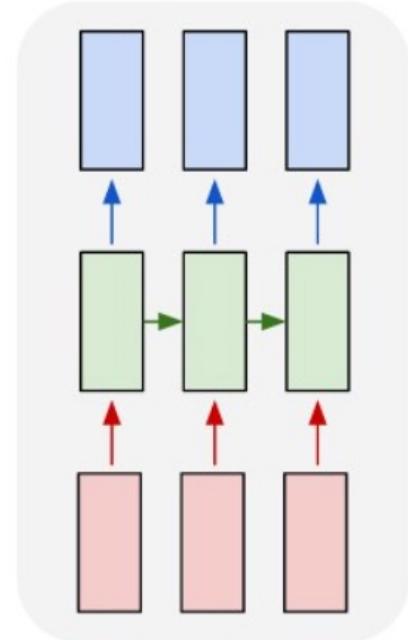
many to one



many to many



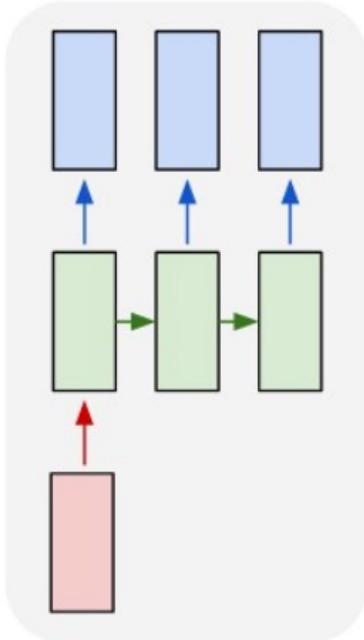
many to many



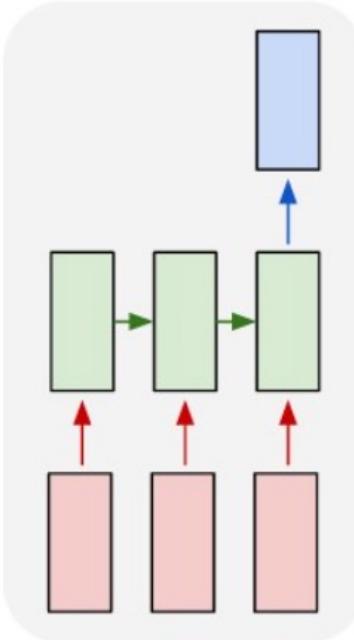
Translation

Sequence Modeling

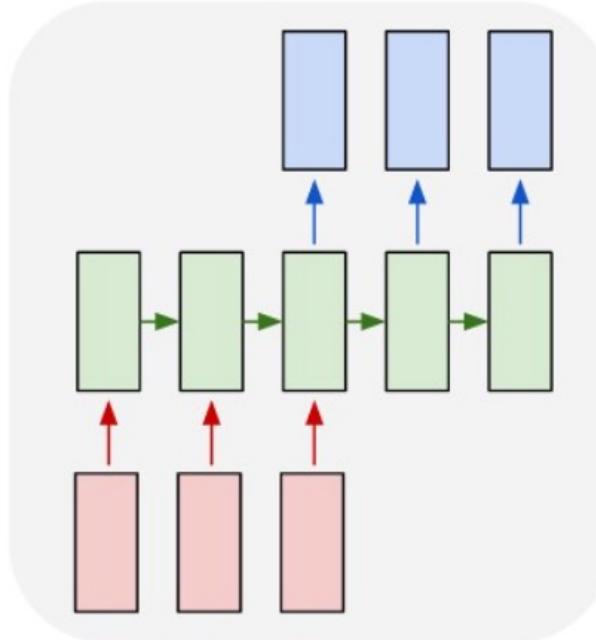
one to many



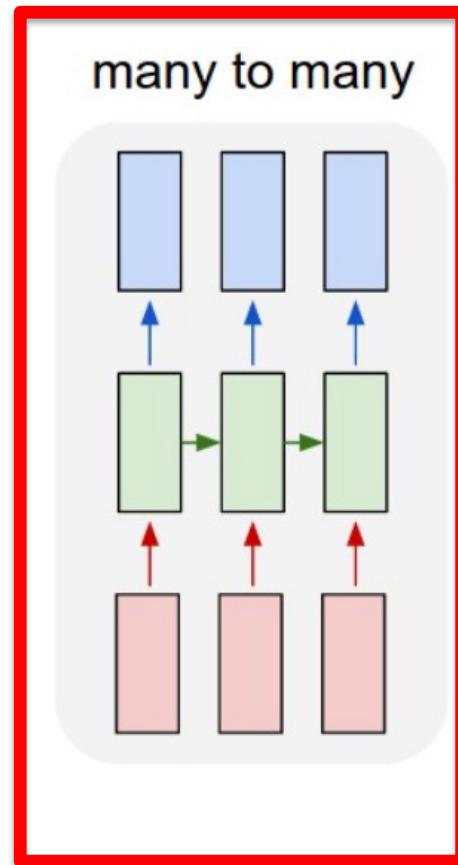
many to one



many to many



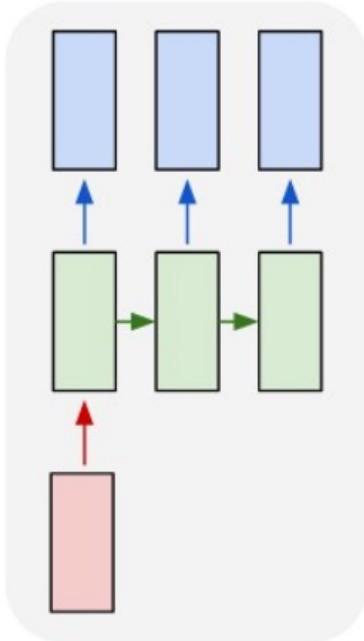
many to many



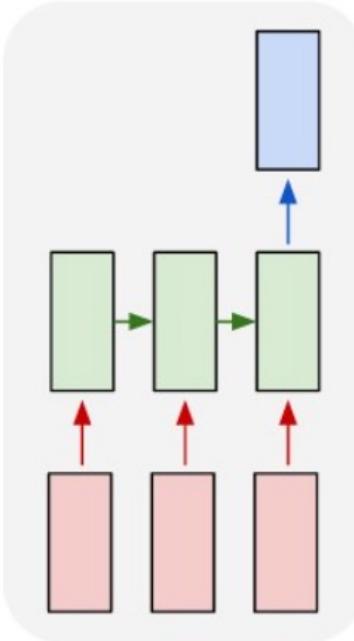
?????

Sequence Modeling

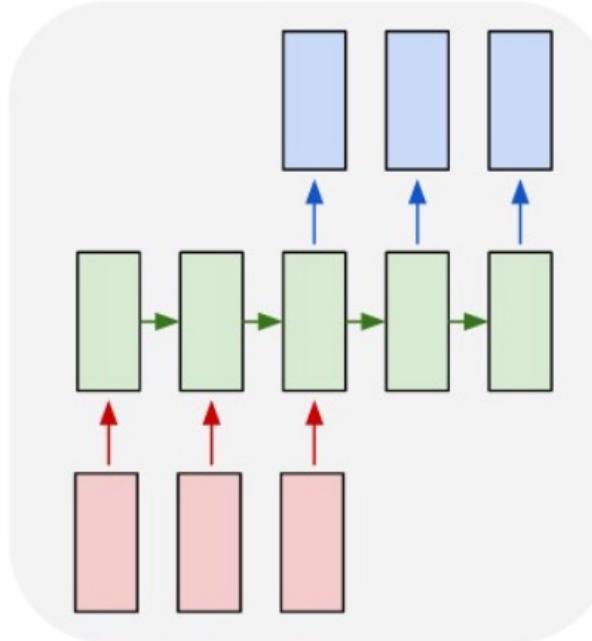
one to many



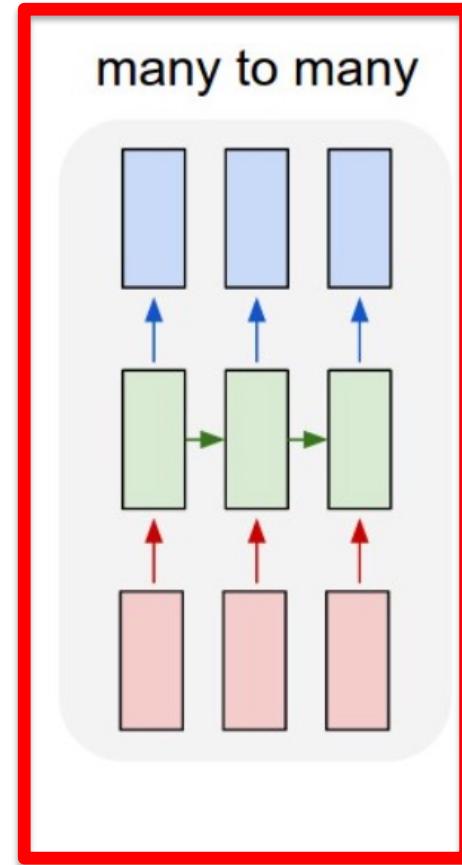
many to one



many to many



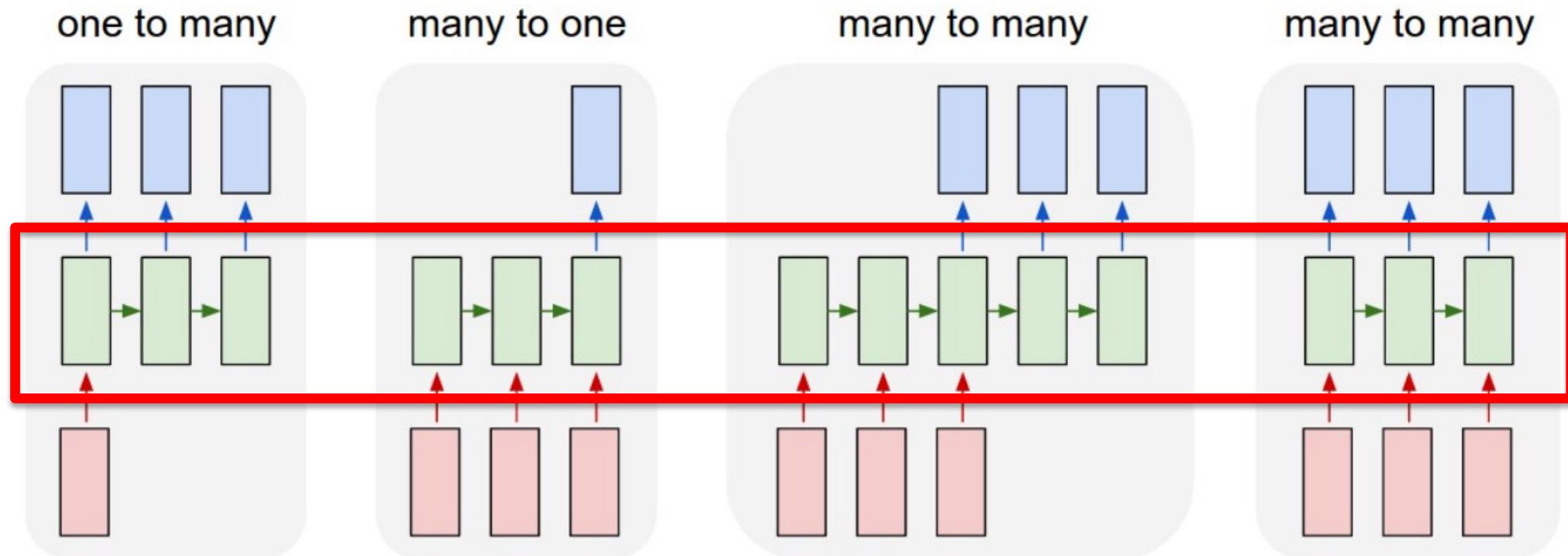
many to many



Time-locked
translation

(Hold off on chatbots for a bit...)

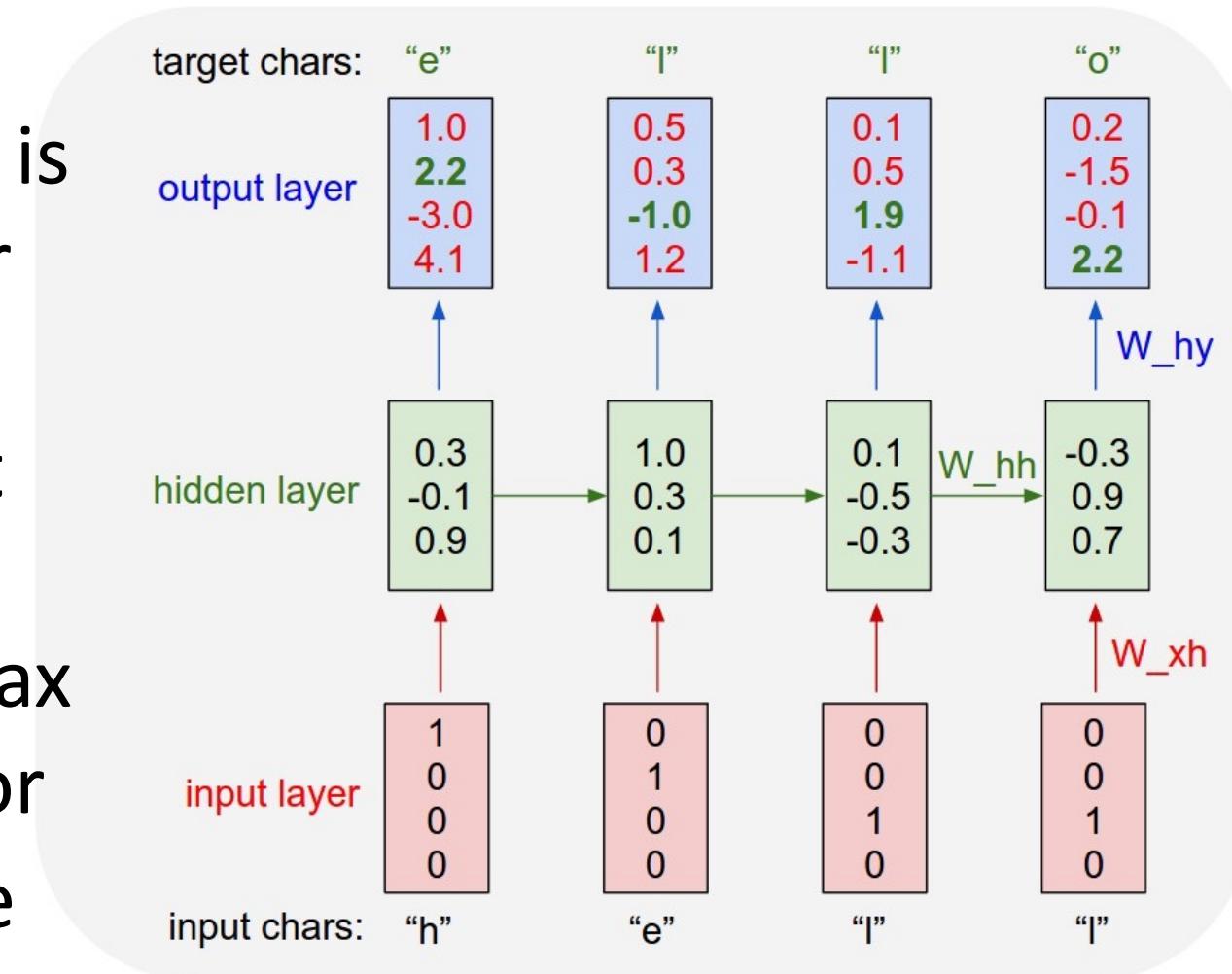
Sequence Modeling



- Internal state represents summary of inputs
- This breaks out of CBOB's fixed-size context
- Simplest version is a Recurrent Neural Network

Detailed View of Char Model

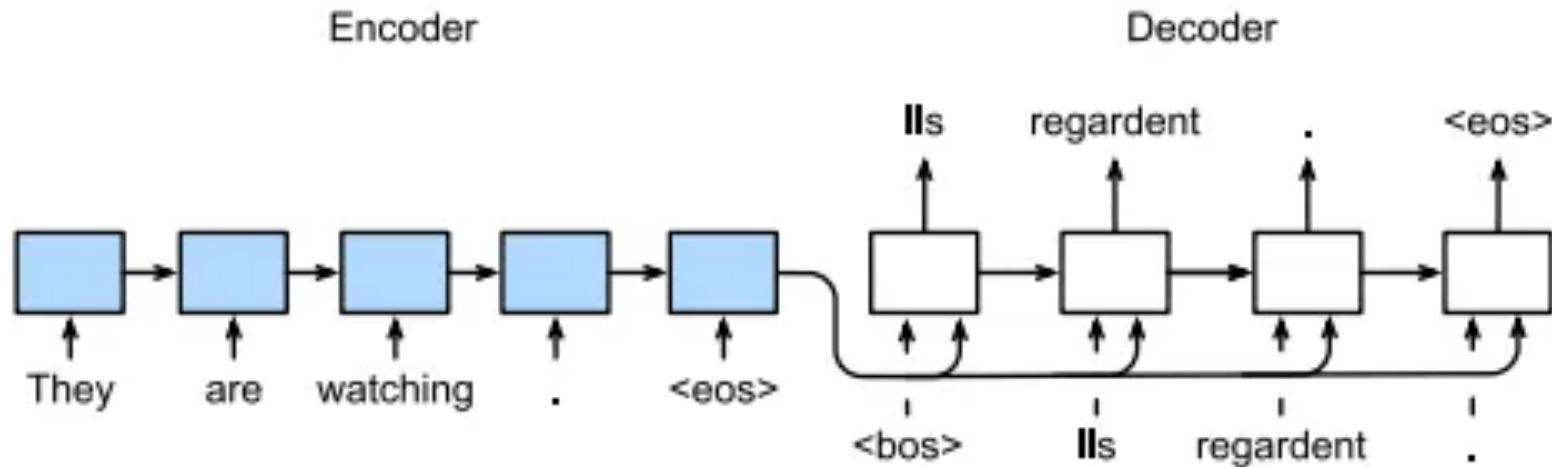
- Task: predict the next character
- One input word is broken into four “labeled” pairs
- Input is one-hot encoded
- Output is softmax prediction vector
- W_{xh} and W_{hy} are init’ed randomly, then learned via training



Karpathy, “Unreasonable ...”

RNNs and seq2seq

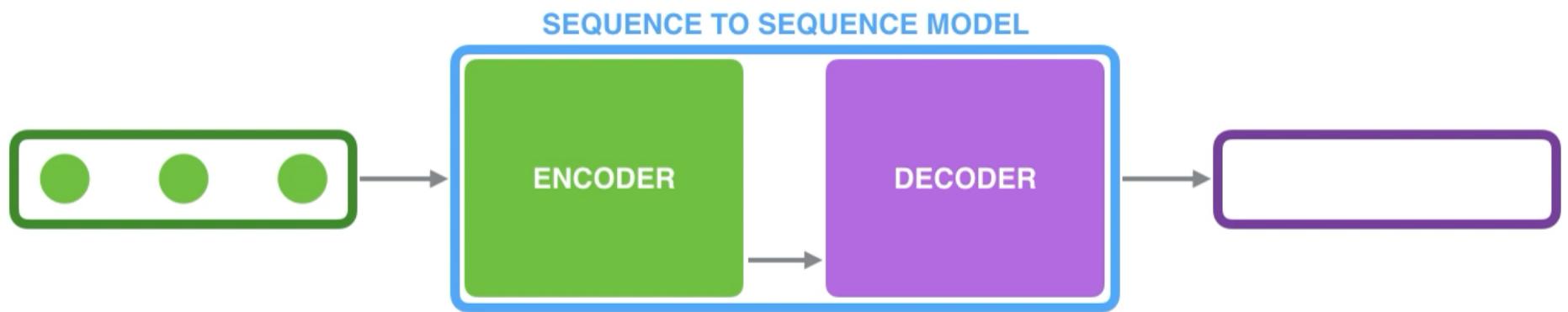
- Sometimes we have an entire sequence to translate, summarize, or answer
- We can train and combine two RNNs in an encoder/decoder “seq2seq” architecture



Sebastian Raschka, Vahid Mirjalili. Python Machine Learning

- Encoder RNN predicts next input. Decoder RNN takes encoder state and predicts next output

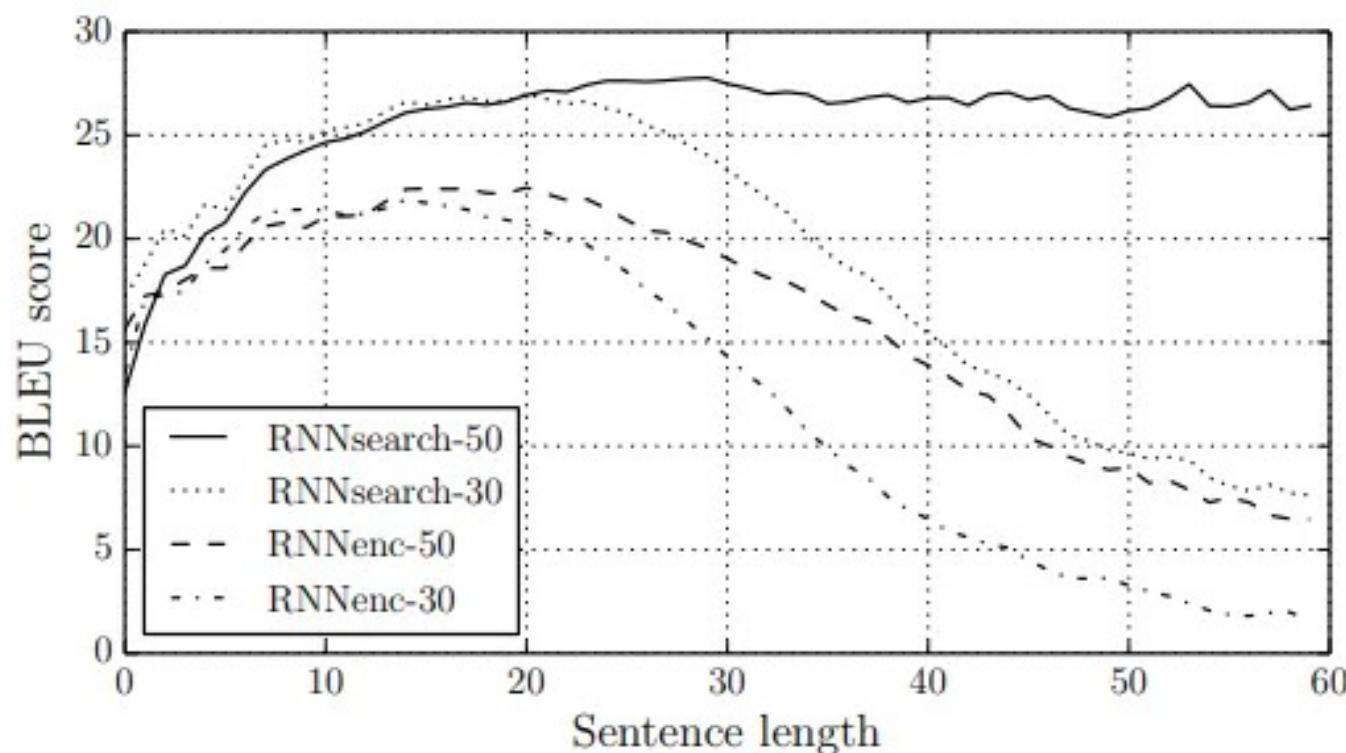
RNNs and seq2seq



Allamar, Visualizing A Neural Machine Translation Model

RNN Problems

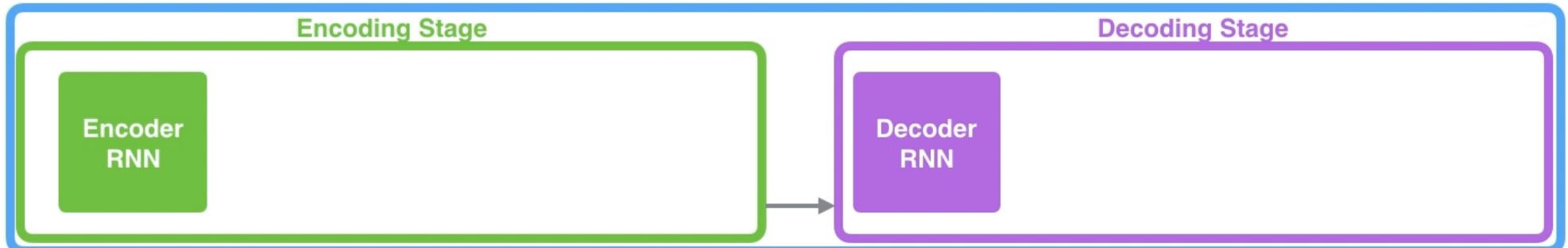
- Long-distance information passing isn't great.
First word of long seq may be “forgotten” by end
- In seq2seq architecture, the entire input sentence must be captured in one vector sent to decoder



Bahdanau, Cho, Bengio, “Neural Machine Translation By Jointly Learning to Align and Translate”

RNN Problems

Neural Machine Translation SEQUENCE TO SEQUENCE MODEL



Je suis étudiant

RNN Problems

- Long-distance information passing isn't great.
First word of long seq may be “forgotten” by end
- In seq2seq architecture, the entire input sentence must be captured in one vector sent to decoder (the “encoder bottleneck”)
- Various mechanisms invented to address this, such as Long Short-Term Memories
- LSTMs had trainable components to intentionally “forget” parts of input and alleviate bottleneck
- The “attention” mechanism was the most successful of these

Agenda

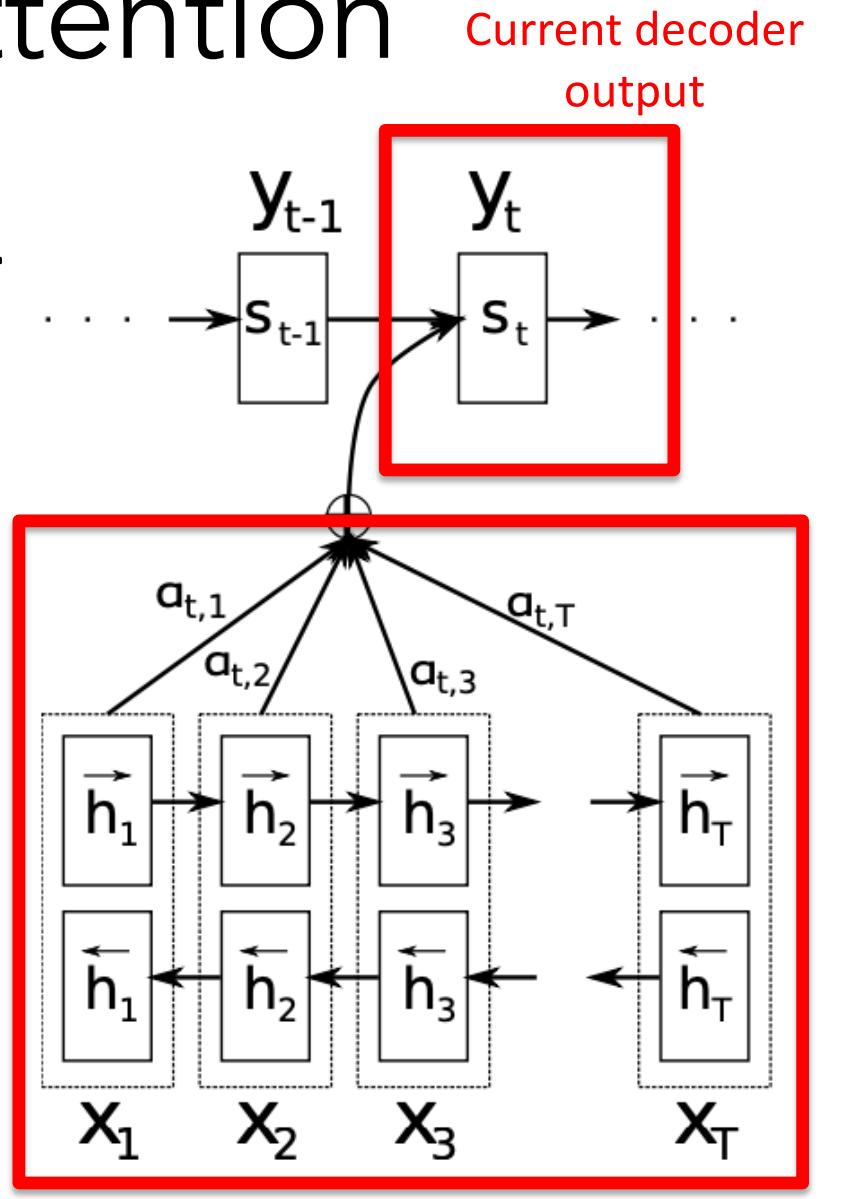
- Text Modeling
- Sequence Models
- Attention
- Transformers
- Fine-Tuning and RLHF

Paying Attention

- Core idea: given some input, figure out the important parts and (mainly) ignore the rest
- How does your attention work when reading?
- **The attention mechanism itself should be learned**
- First introduced in “Neural Machine Translation By Jointly Learning to Align and Translate”, by Bahdanau, Cho, Bengio
- The decoder “chooses a subset of [encoder input] vectors adaptively while decoding the translation”

Bahdanau Attention

- Karpathy, AI's bard: “The context vector from encoder is a weighted sum of hidden states of words of the encoding”
- Those attention weights are themselves computed by looking at current decoder state and encoder values



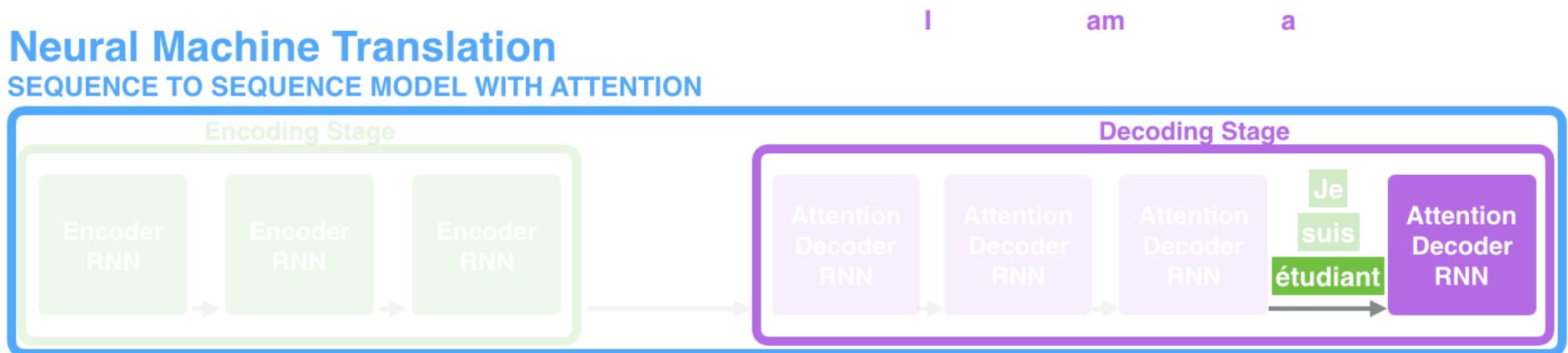
Attention-weighted encoder context vectors

Bahdanau, Cho, Bengio, “Neural Machine Translation By Jointly Learning to Align and Translate”

Bahdanau Attention

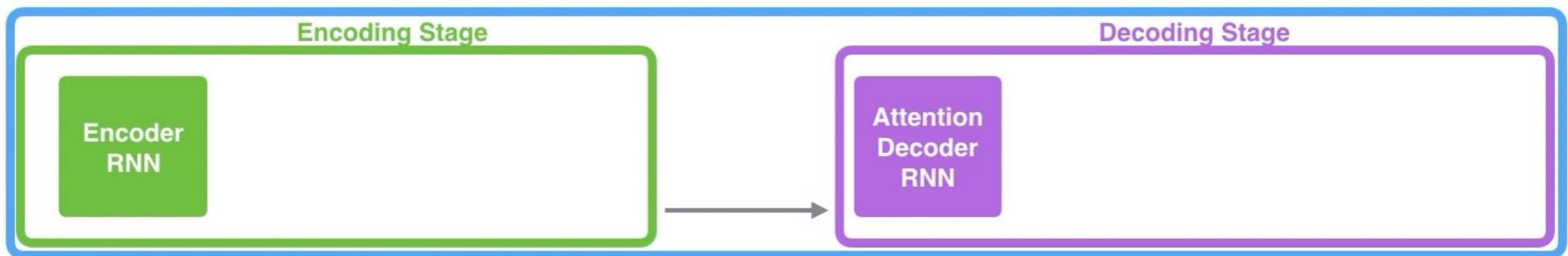
Time step: 7

Neural Machine Translation SEQUENCE TO SEQUENCE MODEL WITH ATTENTION



Bahdanau Attention

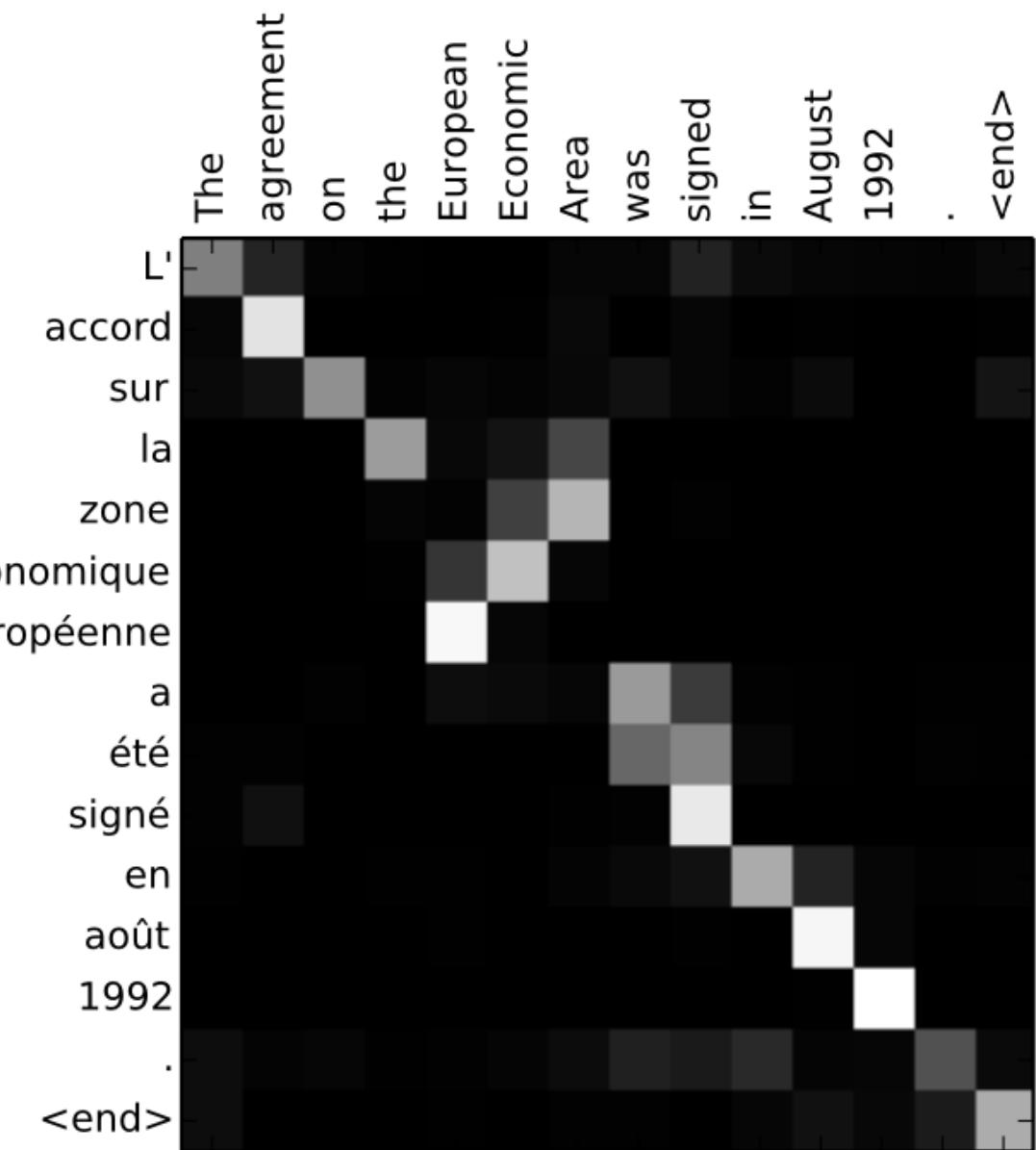
Neural Machine Translation SEQUENCE TO SEQUENCE MODEL WITH ATTENTION



Je suis étudiant

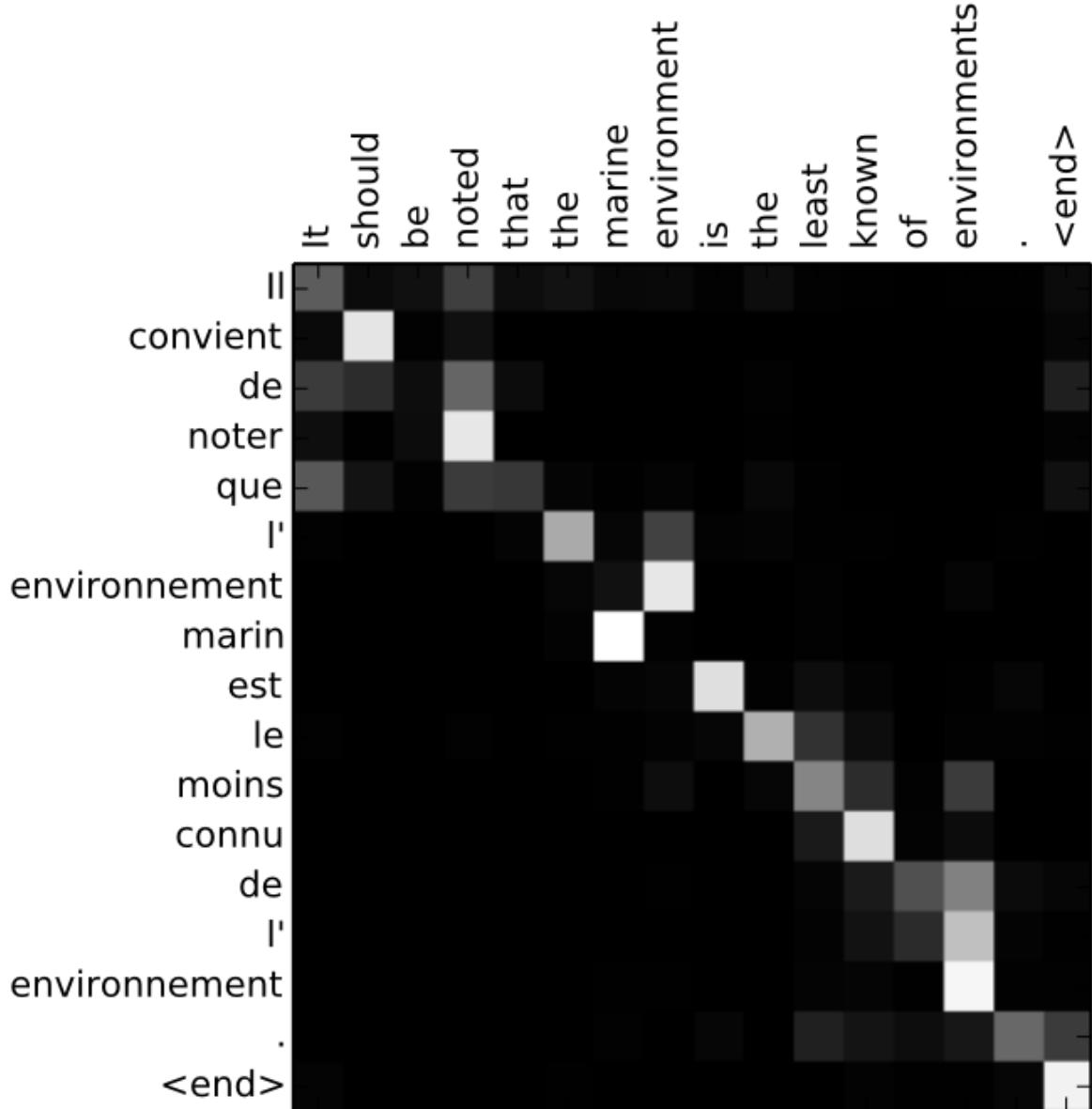
Attention Visualization

- Consider translating English to French
- Pixel values indicate relevance of input word to output word



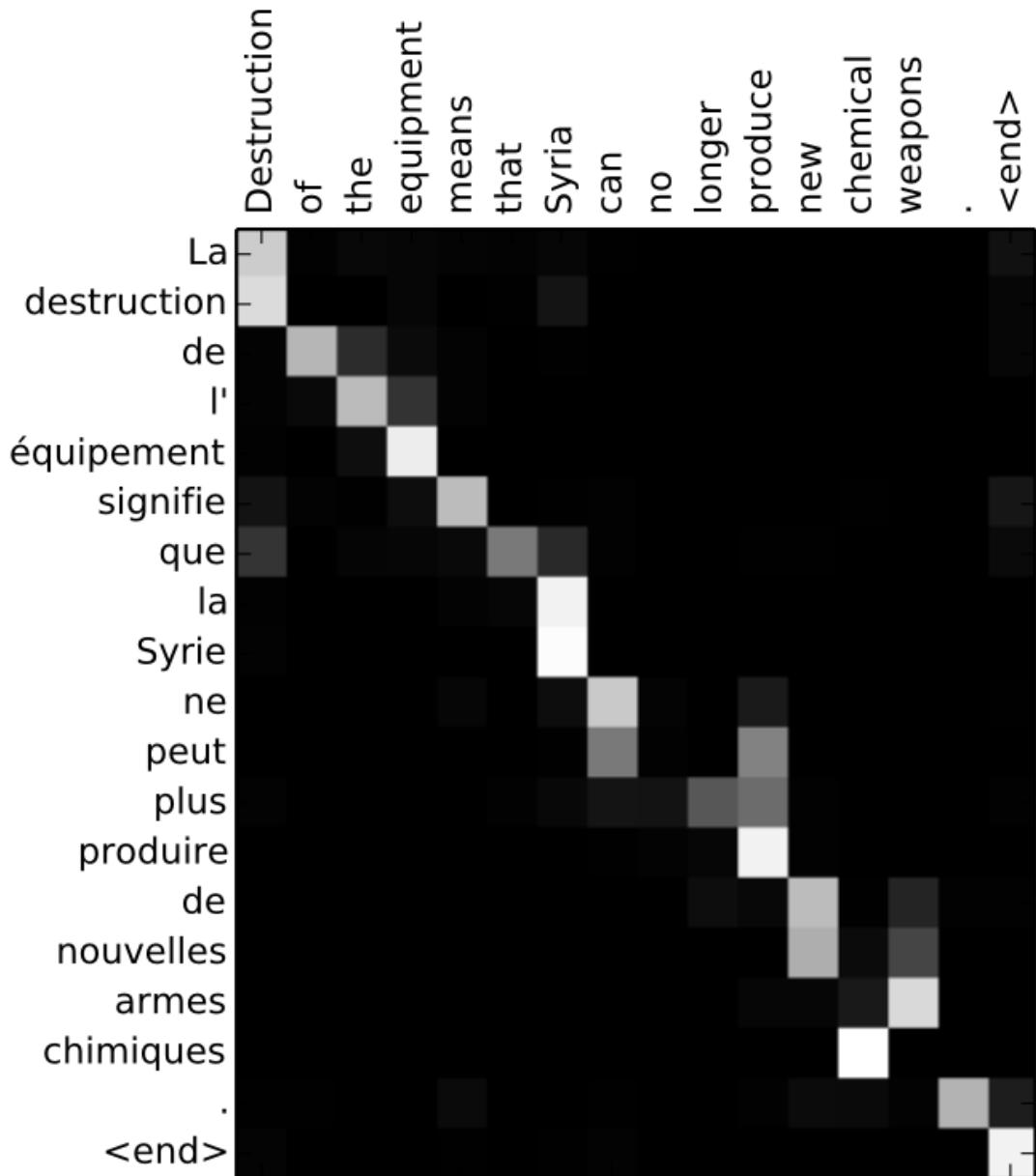
Attention Visualization

- Consider translating English to French
- Pixel values indicate relevance of input word to output word



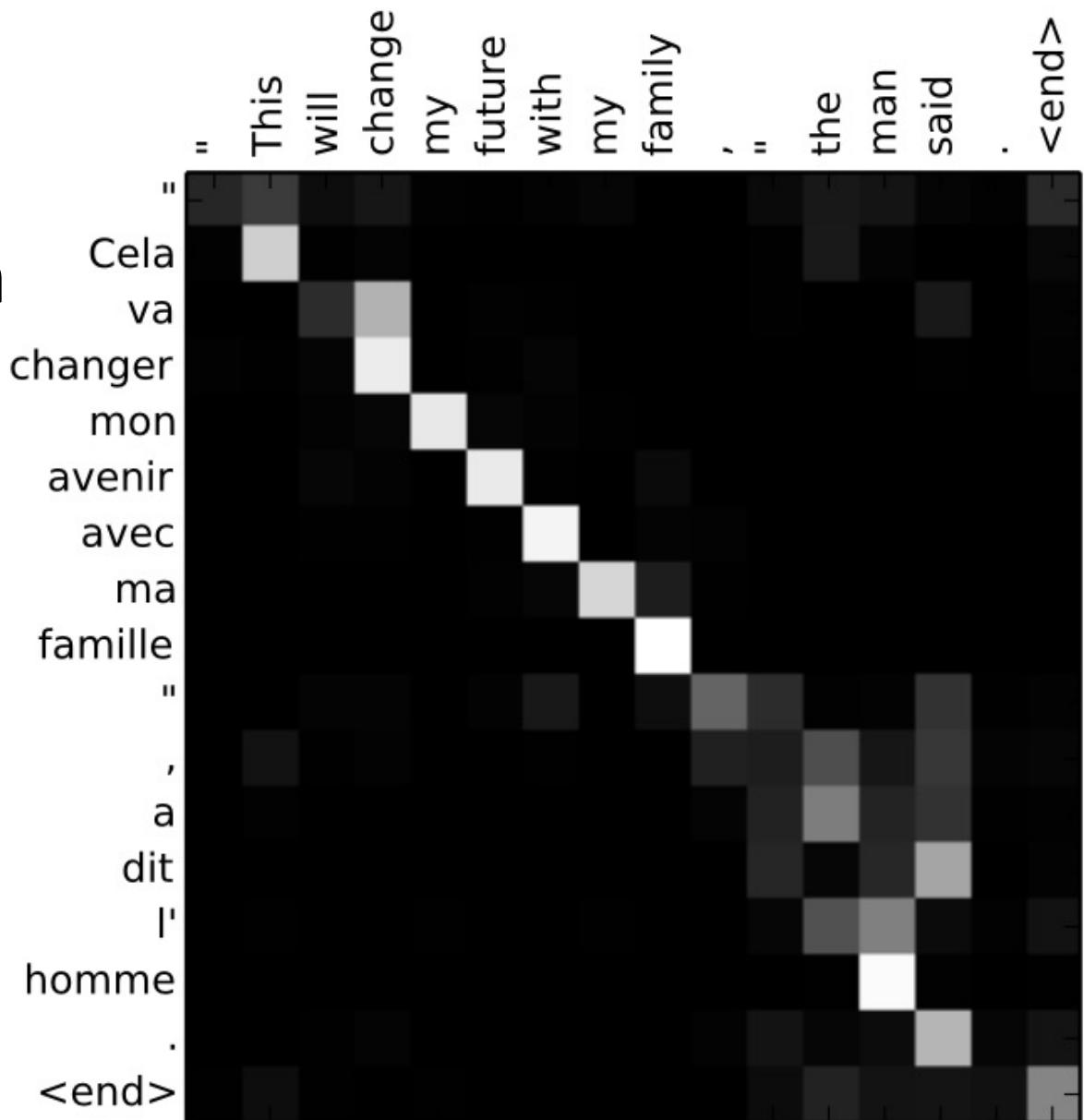
Attention Visualization

- Consider translating English to French
- Pixel values indicate relevance of input word to output word



Attention Visualization

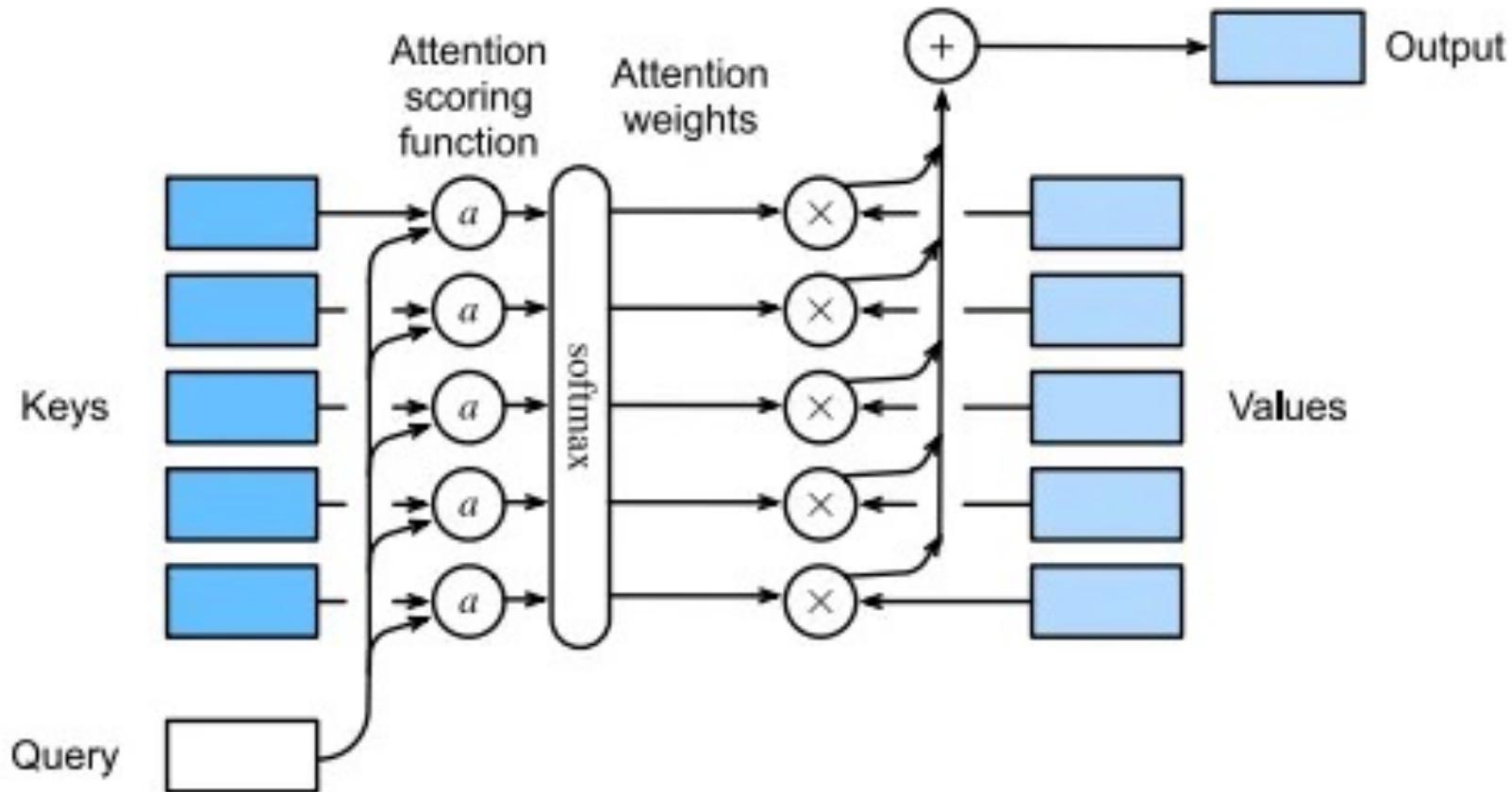
- Consider translating English to French
- Pixel values indicate relevance of input word to output word



Attention Model

- Think about Queries, Keys, and Values
 - A Query describes what you're trying to do (“find the predator”)
 - A Key describes a particular input (“red things”)
 - A Value describes the value of that input (“one big red thing near the lower right”)

Attention Model

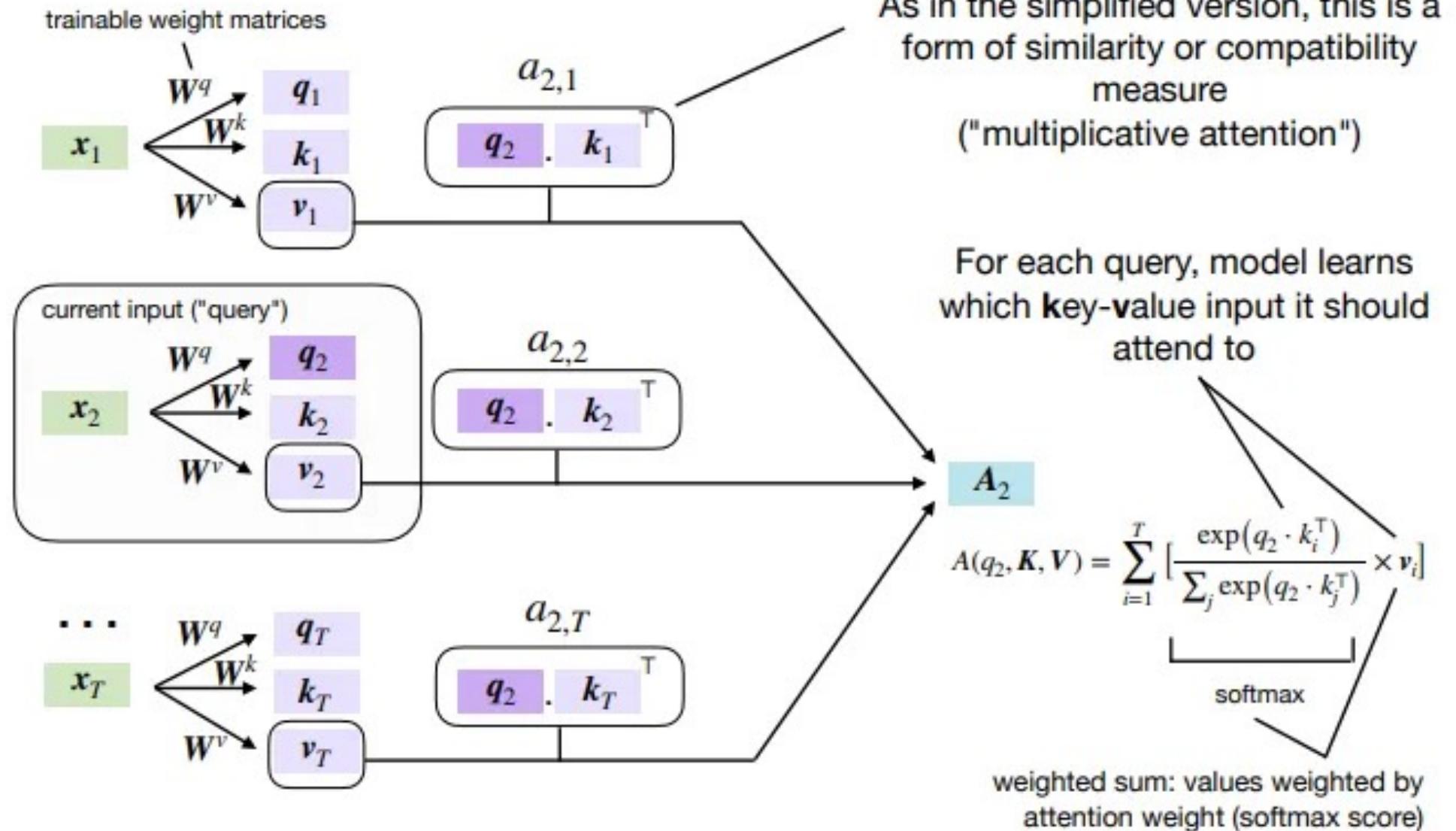


- Queries, Keys, and Values are vectors

Self-Attention

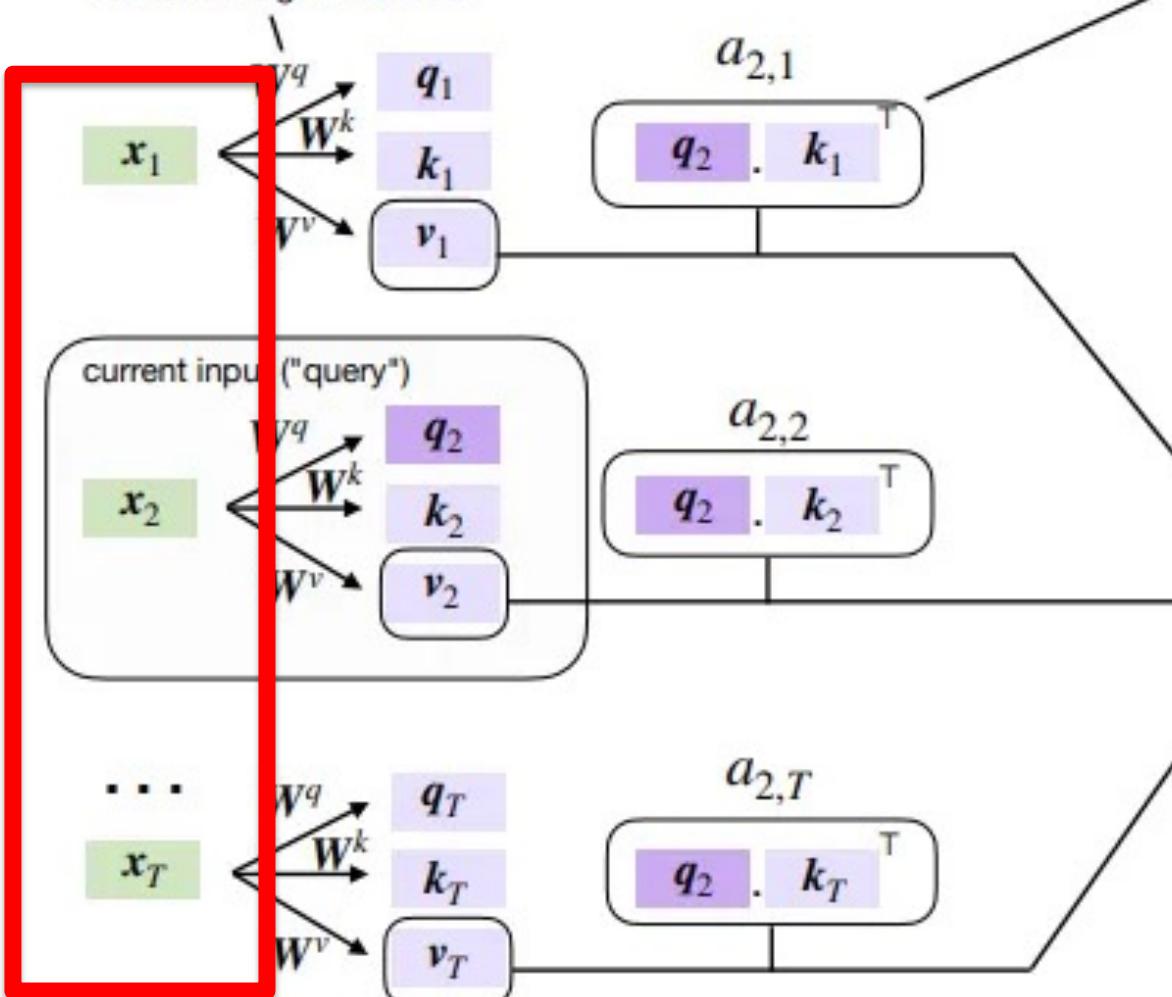
- Self-Attention works on an input sequence, allows us to drop the task-specific stuff
- We can encode both local and global dependencies

Self-Attention



Self-Attention

trainable weight matrices



As in the simplified version, this is a form of similarity or compatibility measure ("multiplicative attention")

For each query, model learns which **key-value** input it should attend to

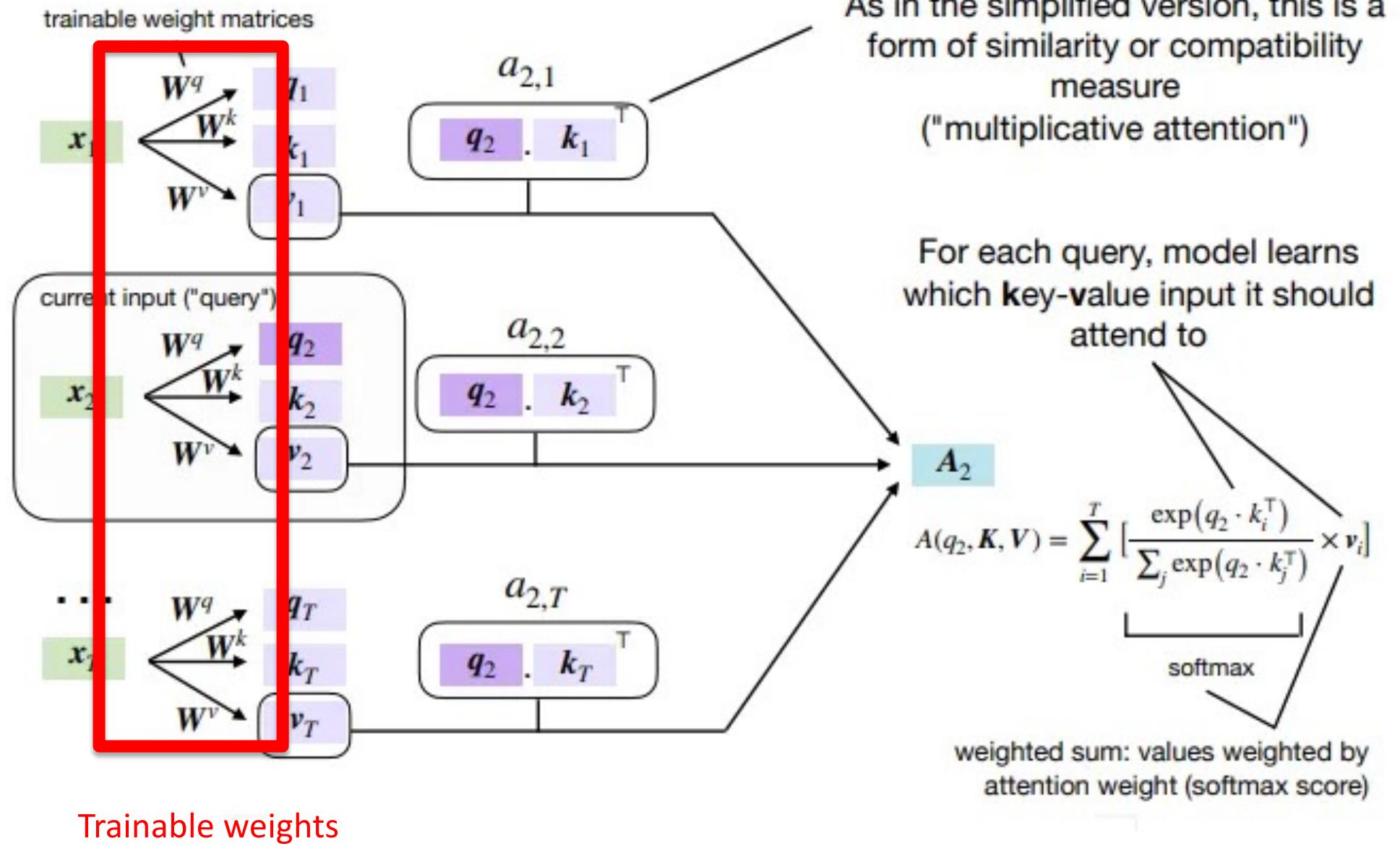
$$A(q_2, K, V) = \sum_{i=1}^T \left[\frac{\exp(q_2 \cdot k_i^\top)}{\sum_j \exp(q_2 \cdot k_j^\top)} \times v_i \right]$$

softmax

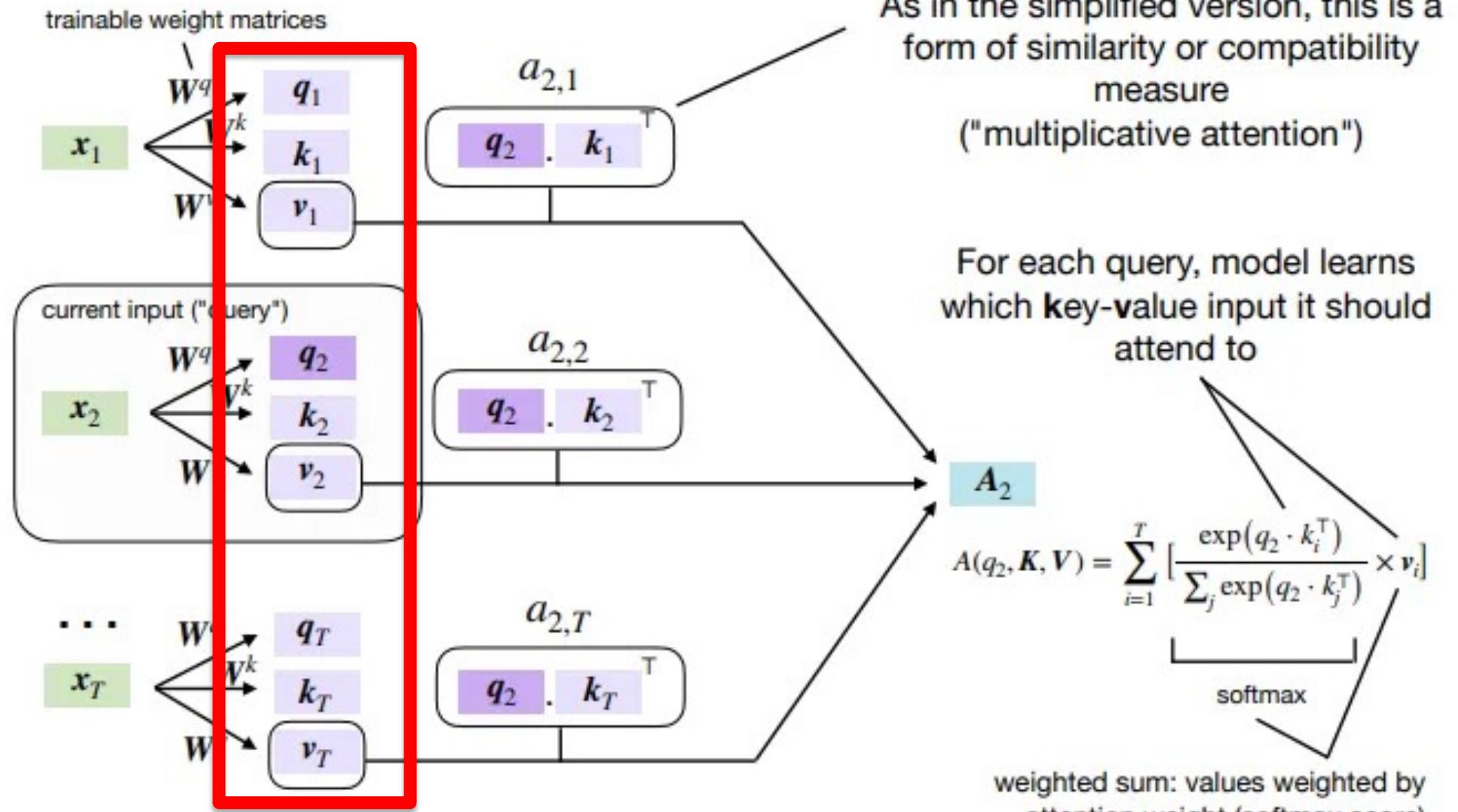
weighted sum: values weighted by attention weight (softmax score)

Input elements

Self-Attention



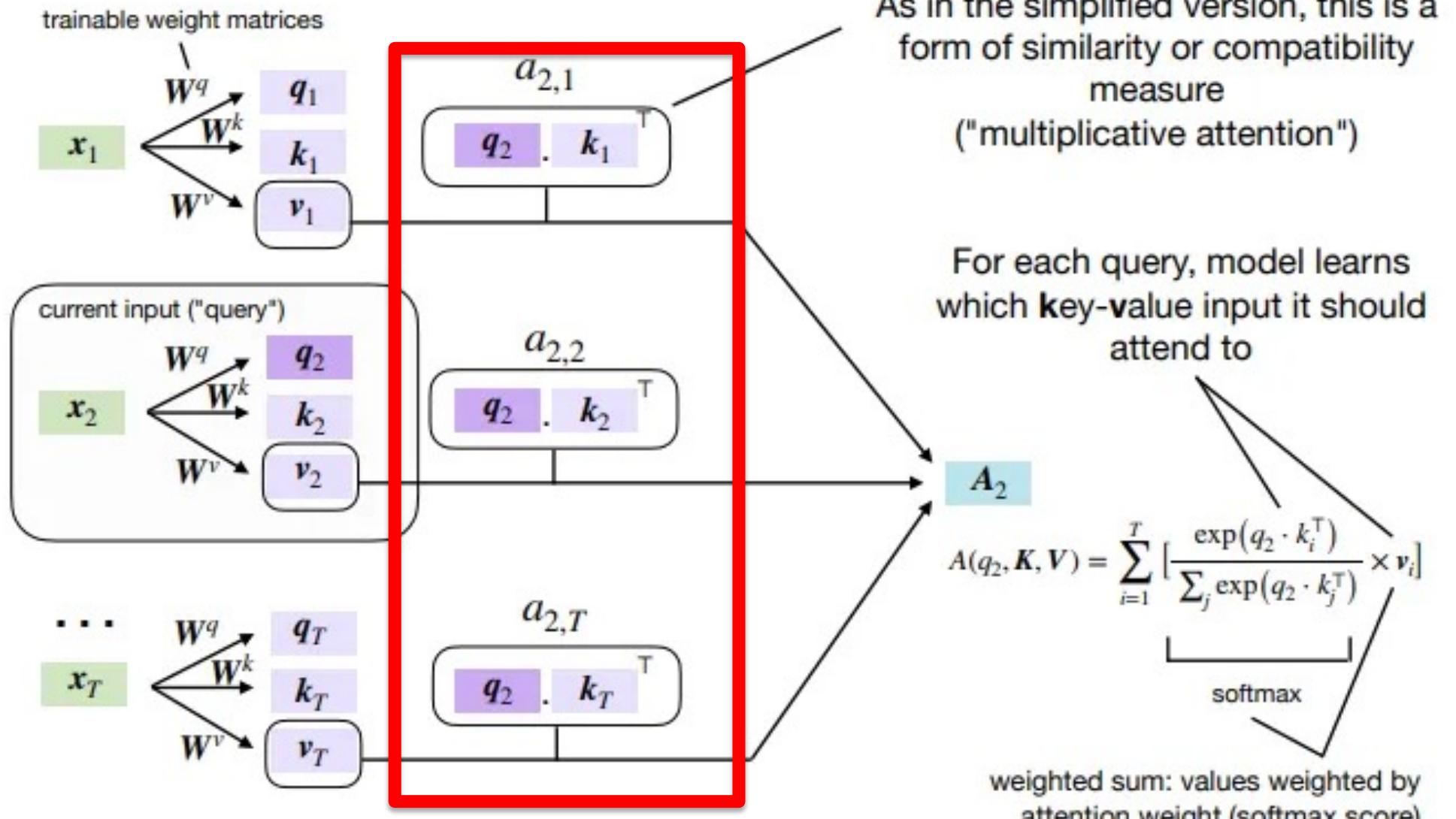
Self-Attention



Query, key, value vectors are result of combining input with learned weights

Intro to Deep Learning by Sebastian Raschka

Self-Attention

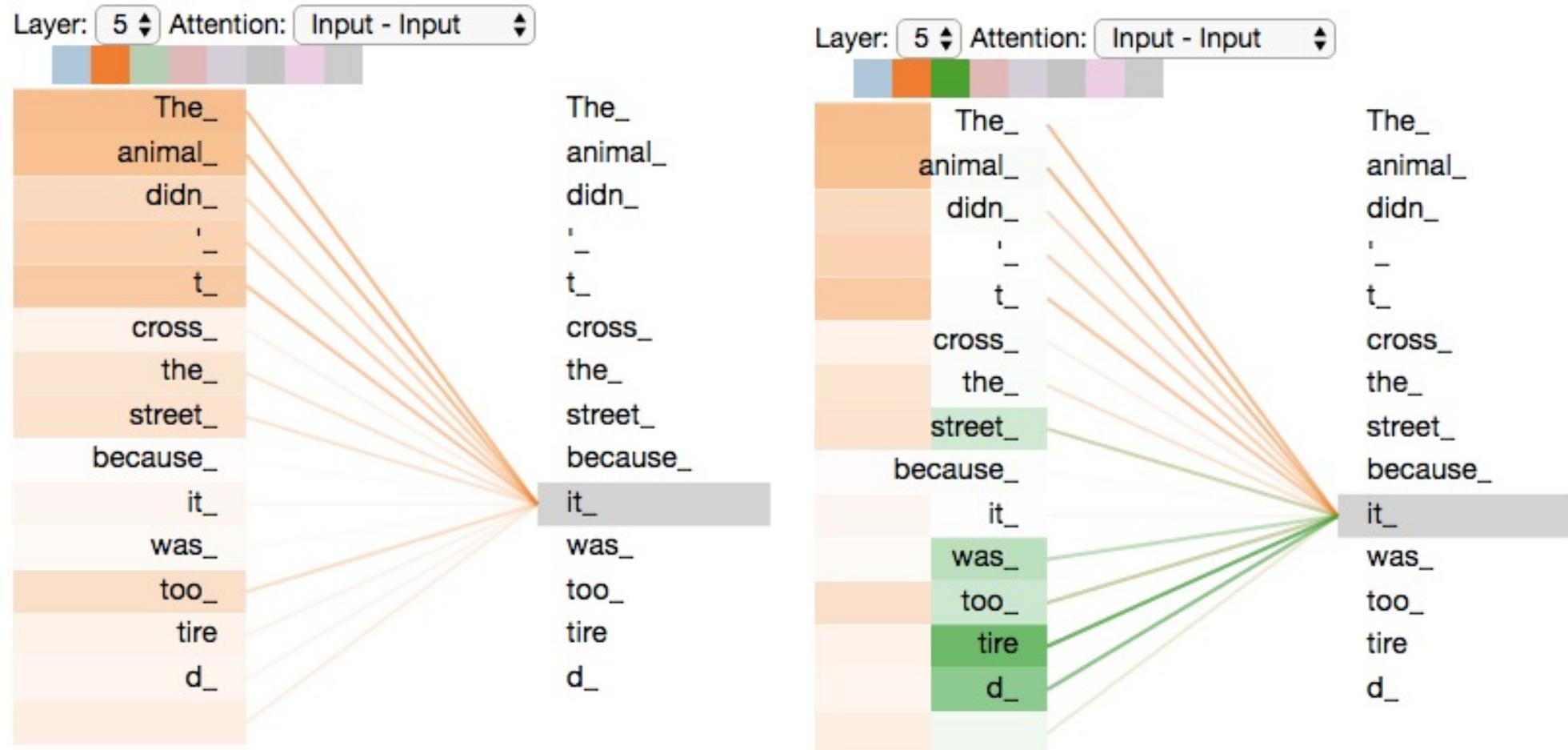


We combine q_2 with each of the k_i in order to obtain the attention we should pay to v_i and we do this for all queries

Multi-Head Attention

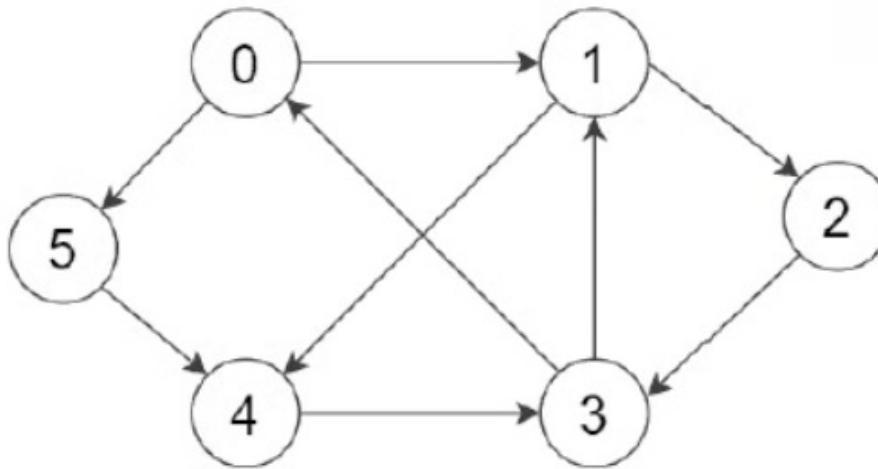
- Compute the Self-Attention mechanism multiple times in parallel
- Using different sets of learned W^q , W^k , W^v weights for each head
- More heads let us capture different kinds of attention

Multi-Head Attention



Karpathy's View of Attention

- Consider a directed graph, where each node stores a vector



- During “communication” nodes compute:
 - Key vector:** What the node has
 - Query vector:** What the node is looking for
 - Value vector:** What the node will emit

Karpathy's View of Attention

- Encoder attention creates “communication” or “message-passing” process
 - Loop over the nodes randomly
 - Each node combines its query vector with incoming nodes’ key vectors; yields “interestingness” score of each input
 - Each node weights incoming values by score to yield an update to the node
- Attention in left-hand encoder “input” layer is fully connected, but masked MHA in decoder limits connectivity to reflect left-to-right text gen

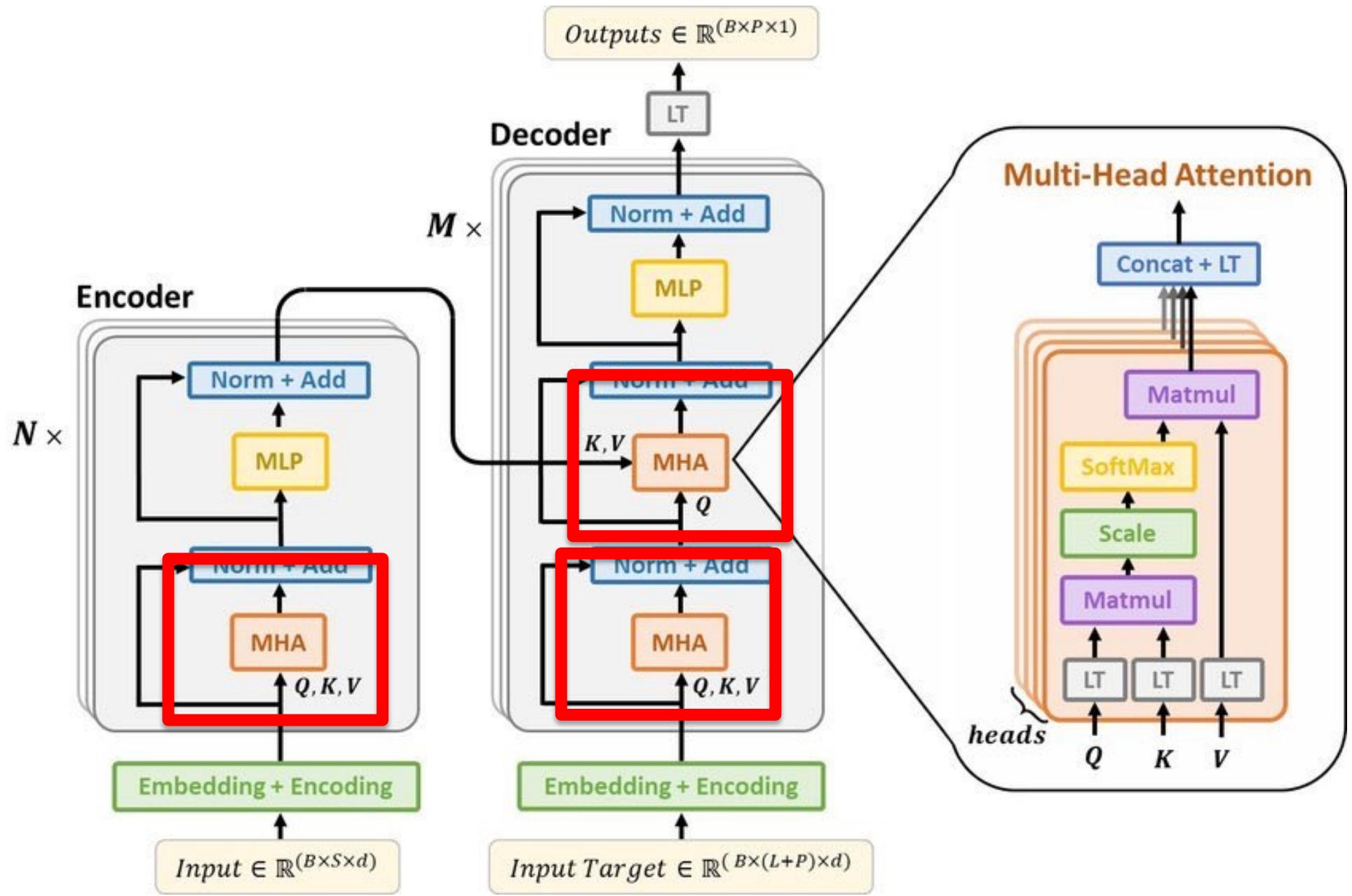
Agenda

- Text Modeling
- Sequence Models
- Attention
- **Transformers**
- Fine-Tuning and RLHF

Transformers

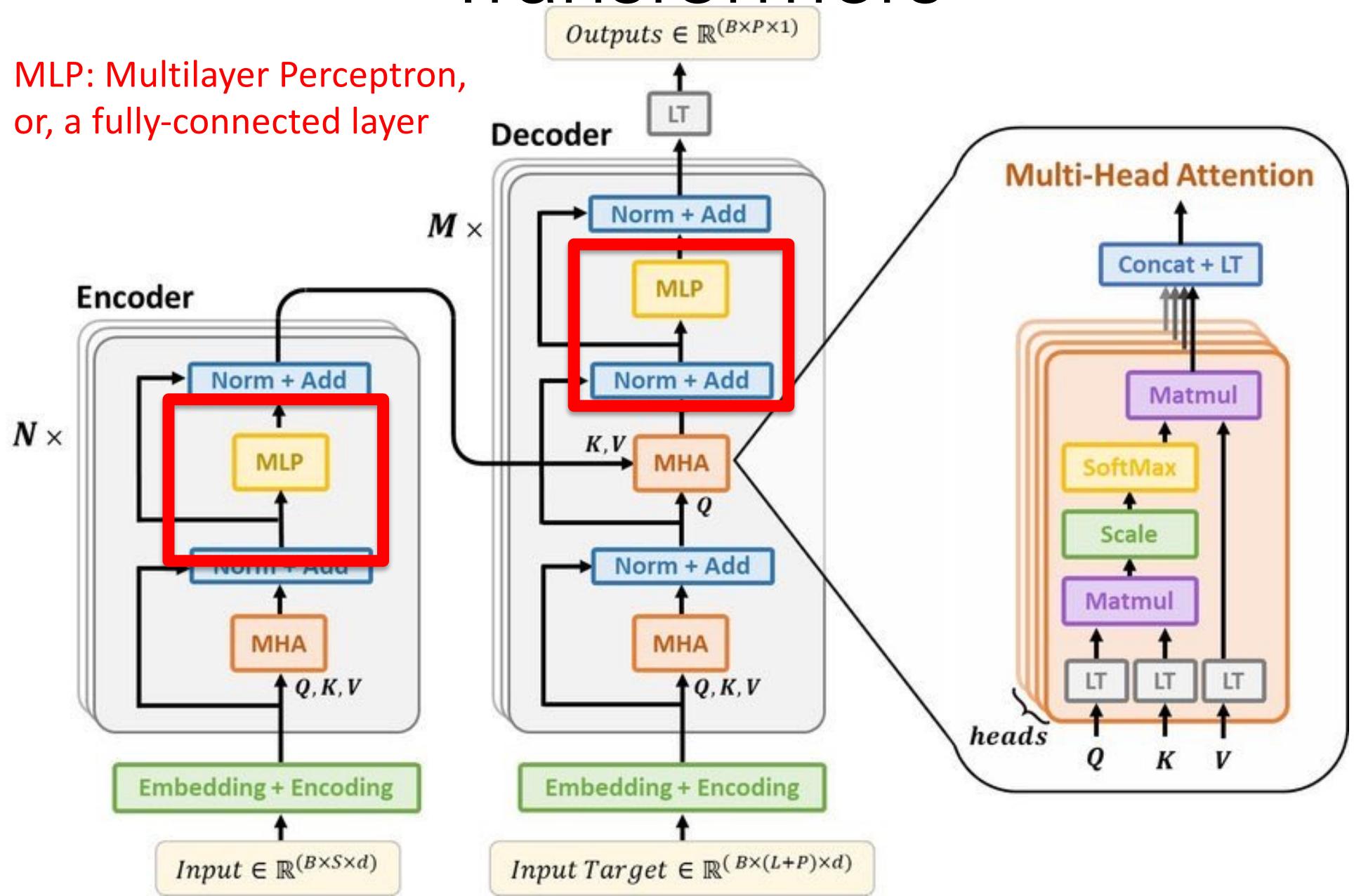
- Most common architecture today is The Transformer
- “Attention is All You Need,” by Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, Polosukhin (2017)
- Drops recurrence; eats entire input
- Still autoregressive

Transformers



Transformers

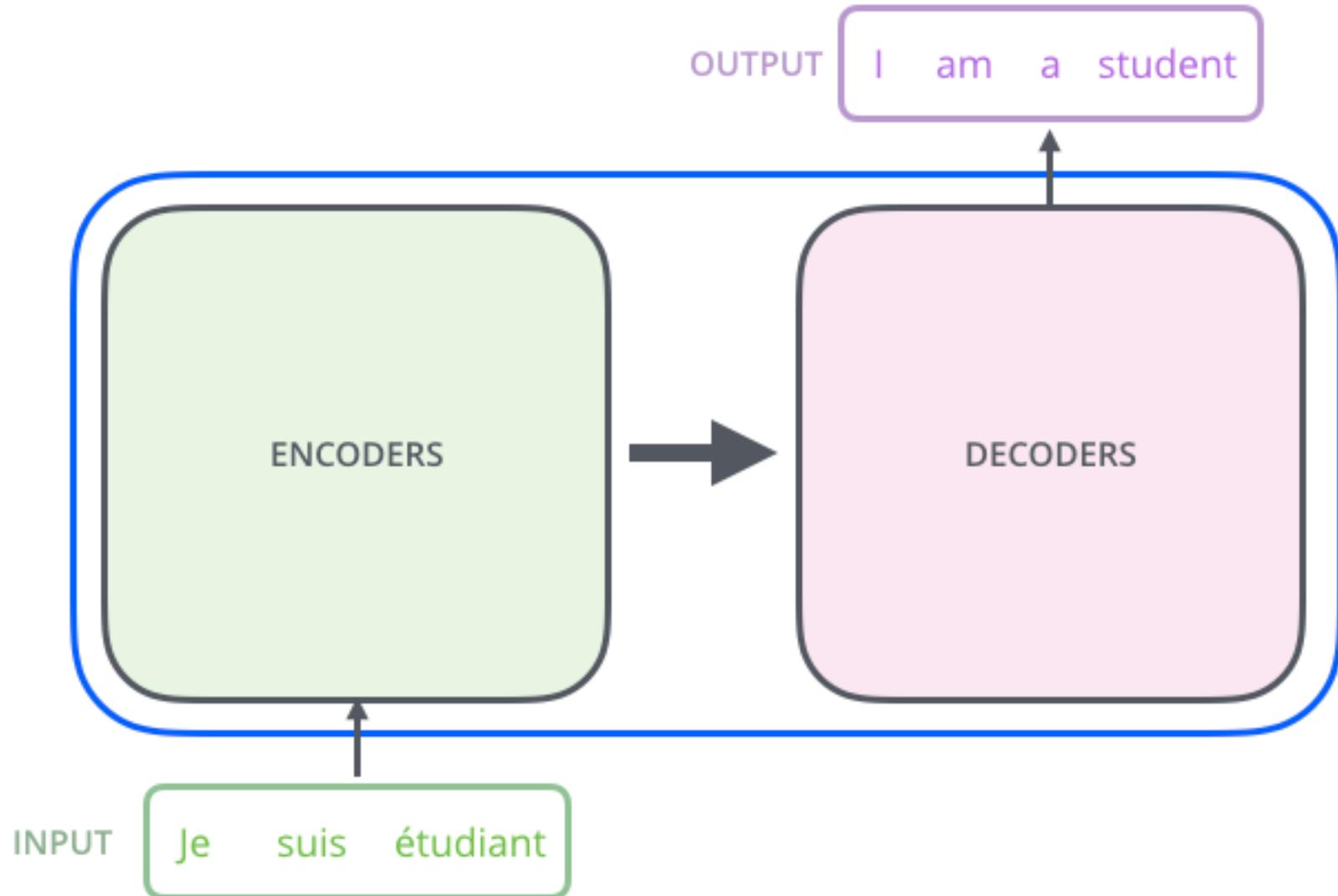
MLP: Multilayer Perceptron,
or, a fully-connected layer



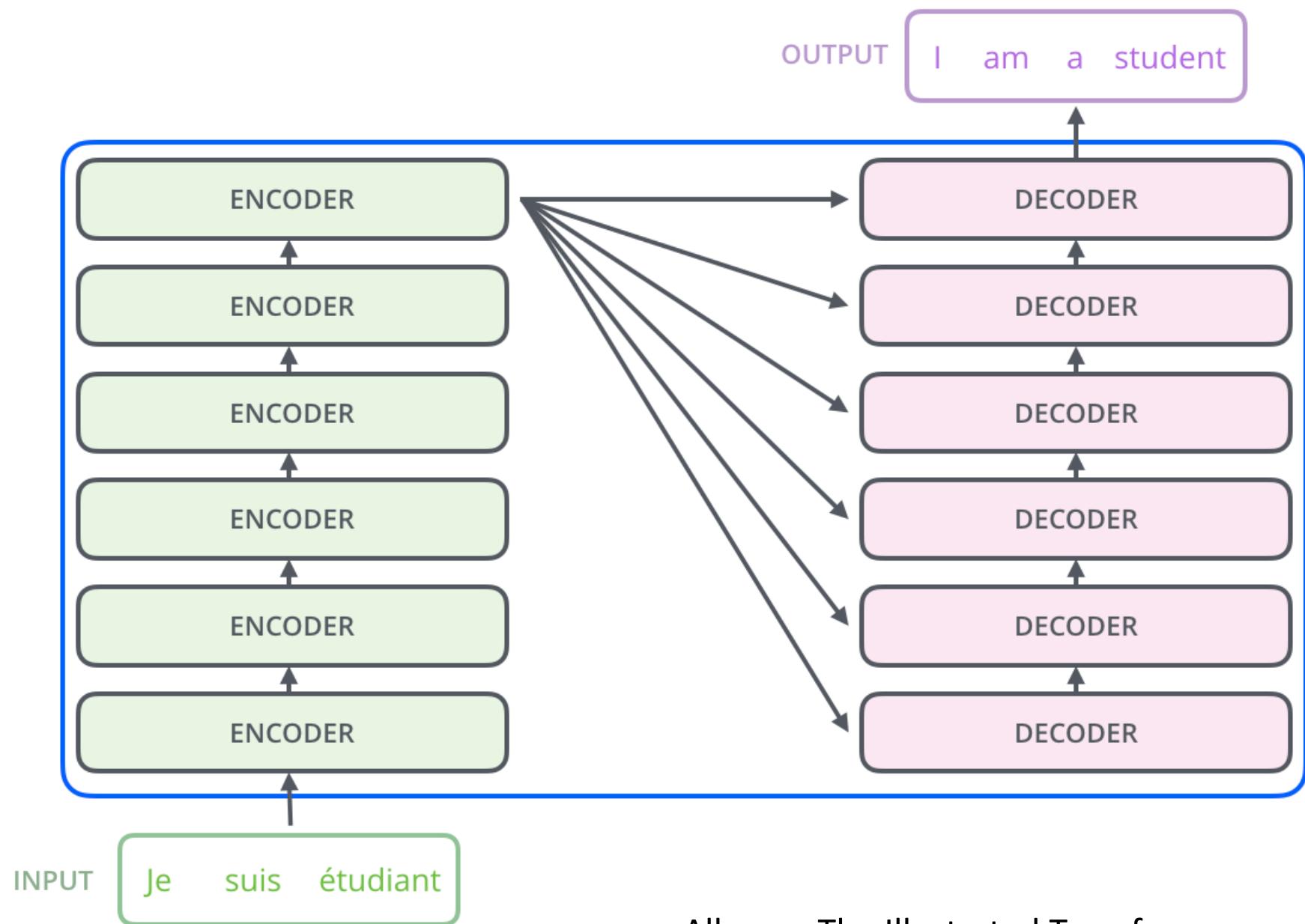
Transformers

- Multi-Head attention is computed in parallel
- Each “pane” is replicated in series N times
- Different weights in each setting
- Masked MHA disallows attention to “future” tokens

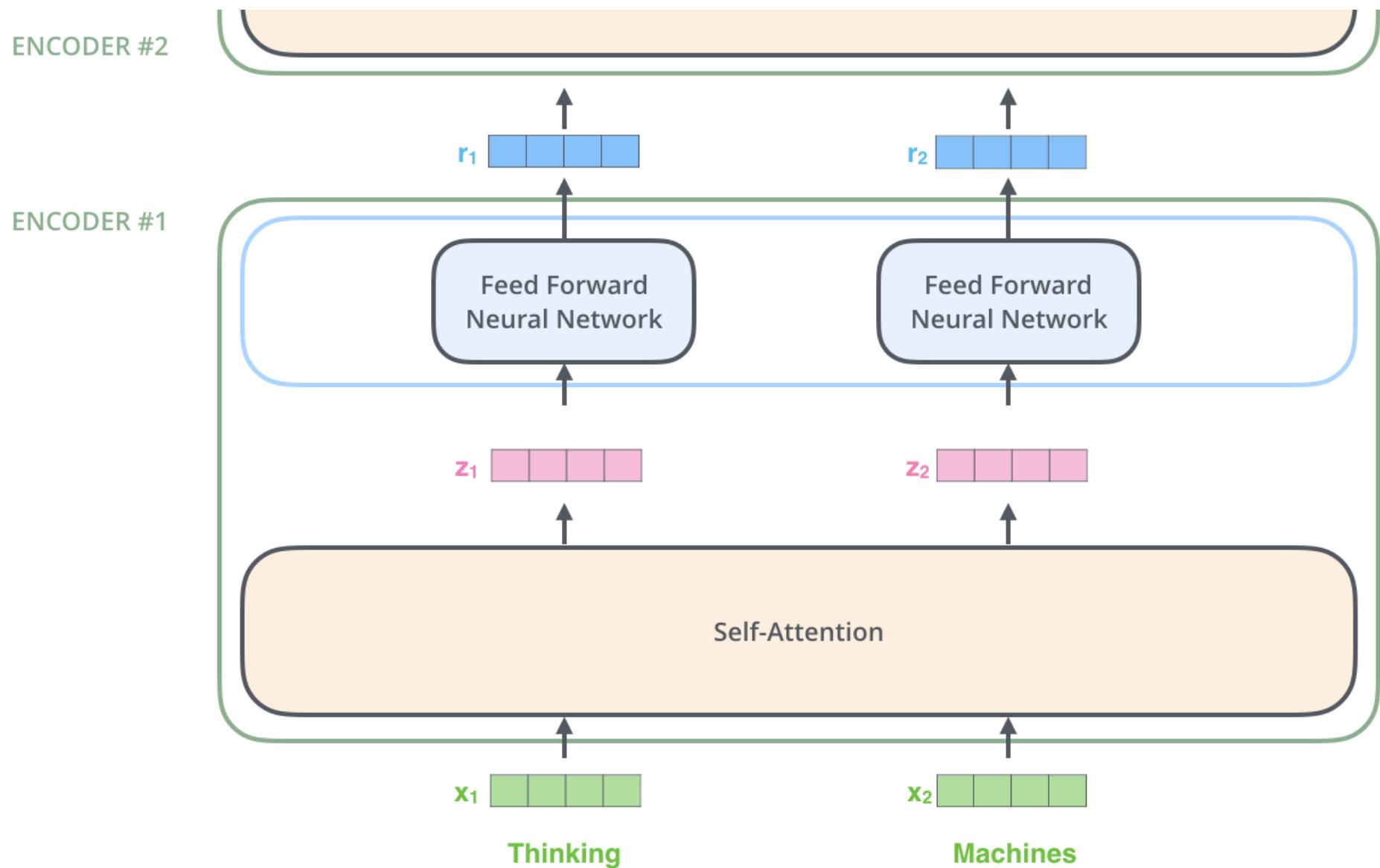
Transformers



Transformers



Transformers

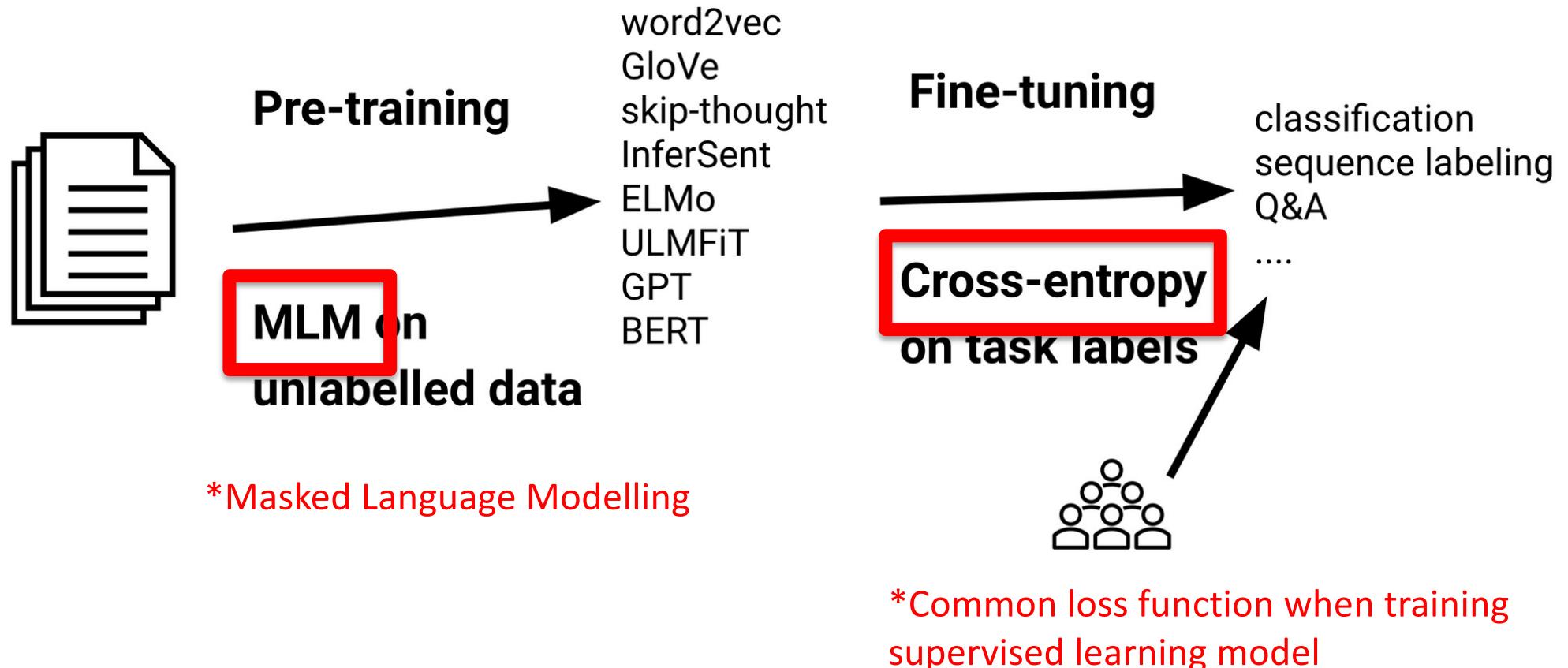


Agenda

- Text Modeling
- Sequence Models
- Attention
- Transformers
- Fine-Tuning and RLHF

Putting It All Together

- We train a big language model, then fine-tune



- FT retrains, using small dataset & limited weight updates

Ruder, <https://www.ruder.io/recent-advances-lm-fine-tuning/>

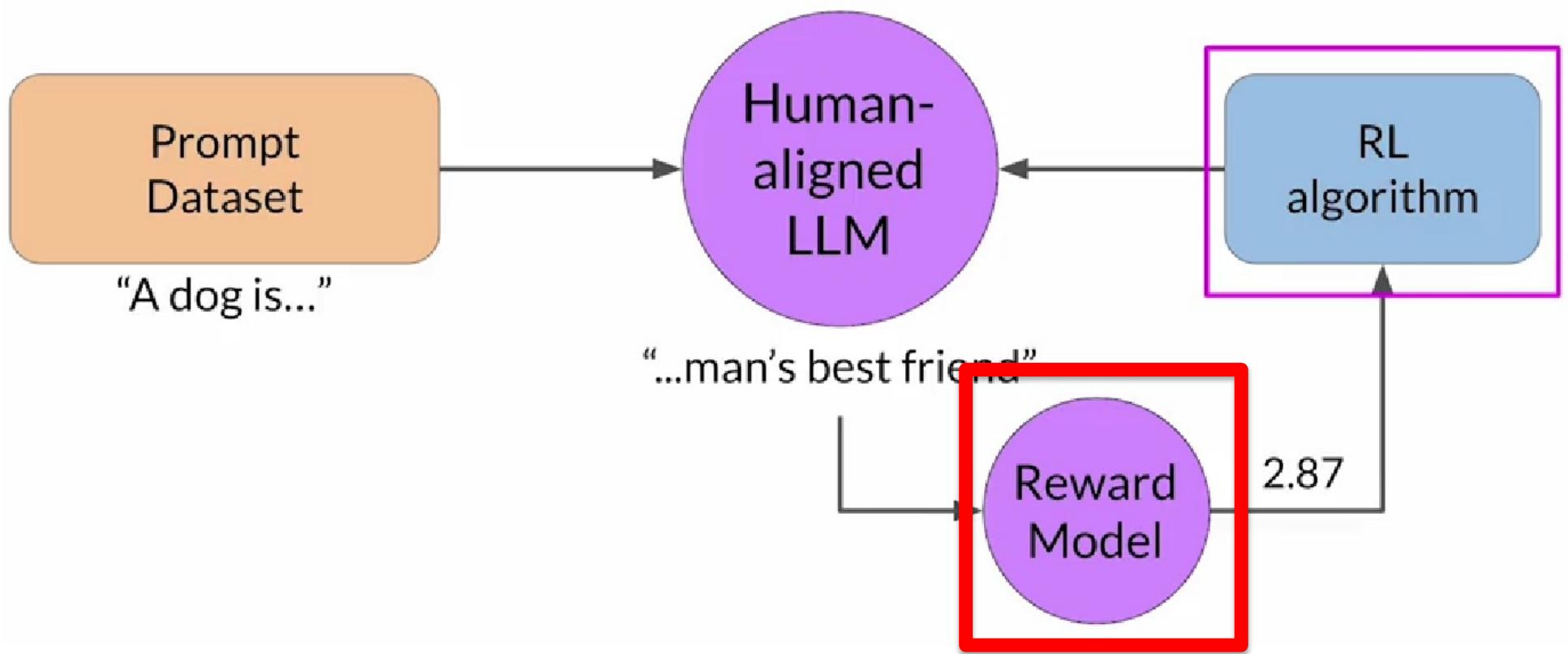
Fine Tuning

- Fine-tuning allows us to adapt to a particular domain or particular task
- Often enables real improvements with relatively small training sets
- **But**, what if we don't have labeled data?
- **Also**, for a given output text, labeling it might be hard for a human, but choosing A vs B is easy

Reinforcement Learning with Human Feedback

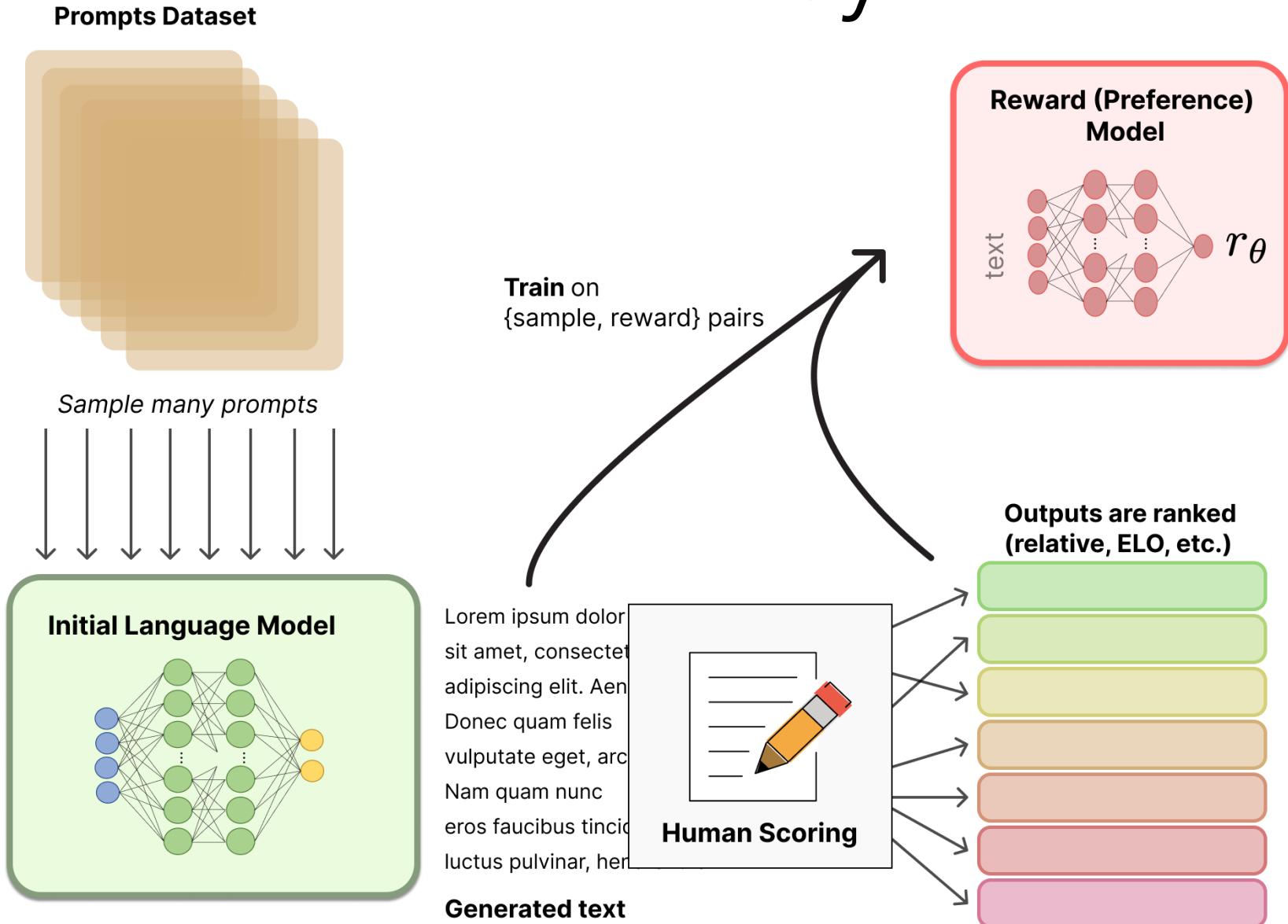
- RLHF lets us use direct human feedback to fine-tune a model
- Often used to modify LLM tone, content guidelines, reducing “toxicity”
- Reinforcement Learning is AI area that uses “good dog/bad dog” signals instead of supervision

The RLHF Cycle



But how do we turn user preferences into a reward value?

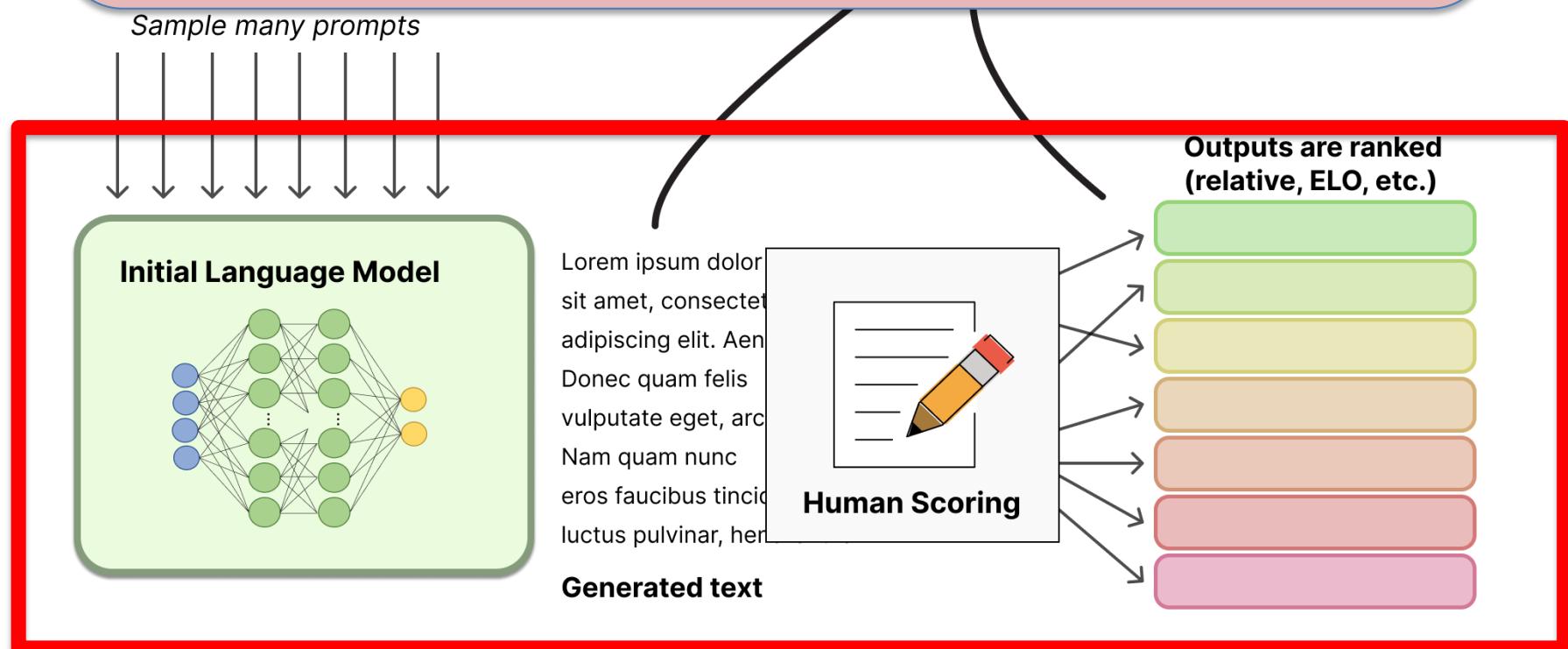
The RLHF Cycle



The RLHF Cycle

Prompts Dataset

1. Human beings rank lots of sample outputs
2. We turn each pairwise element in the ranking into a supervised label

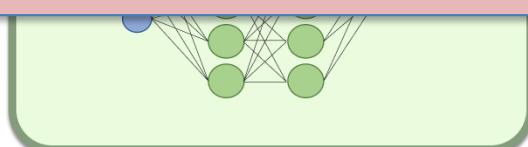


The RLHF Cycle

Prompts Dataset

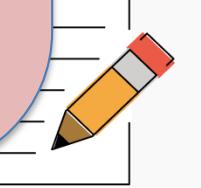
3. Build a supervised model that takes an input text and predicts the users' judgment

4. This supervised model is the reward model for the RLHF cycle

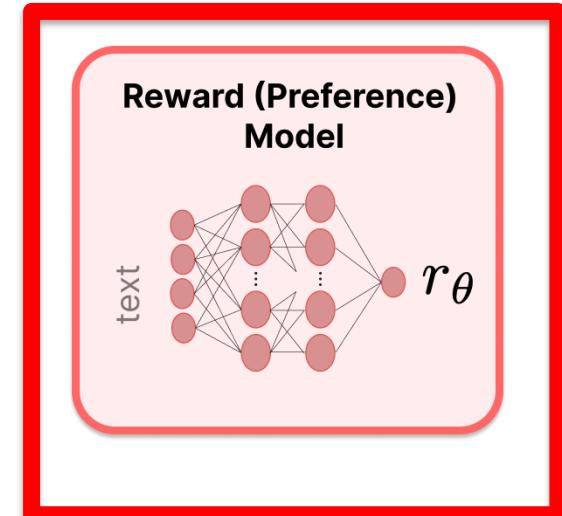


Nam quam nunc
eros faucibus tincid
luctus pulvinar, her

Generated text



Human Scoring



Outputs are ranked
(relative, ELO, etc.)



RLHF Has Its Problems

- How much RL is too much? Hard to figure out when to stop
- Training the reward model is one place we can make mistakes, then fine-tuning adds another
- **Direct Preference Optimization** (Rafailov et al) is a new scheme that avoids constructing the reward model; part of Lab 5!

