



CS 394E: Contemporary Topics in LLMs

Lecture-3: Tokenization and Vector Embeddings

Marco Canini

Professor of Computer Science, KAUST

Agenda

- Recap
- Tokenization
- Vector Embeddings

Recap

- Language Models aim to find the **joint probability**

$$P(W) = P(w_1, w_2, \dots, w_n)$$



$$P(W) = P(w_1) \cdot P(w_2 | w_1) \cdot P(w_3 | w_1, w_2) \cdot \dots \cdot P(w_n | w_1, \dots, w_{n-1})$$



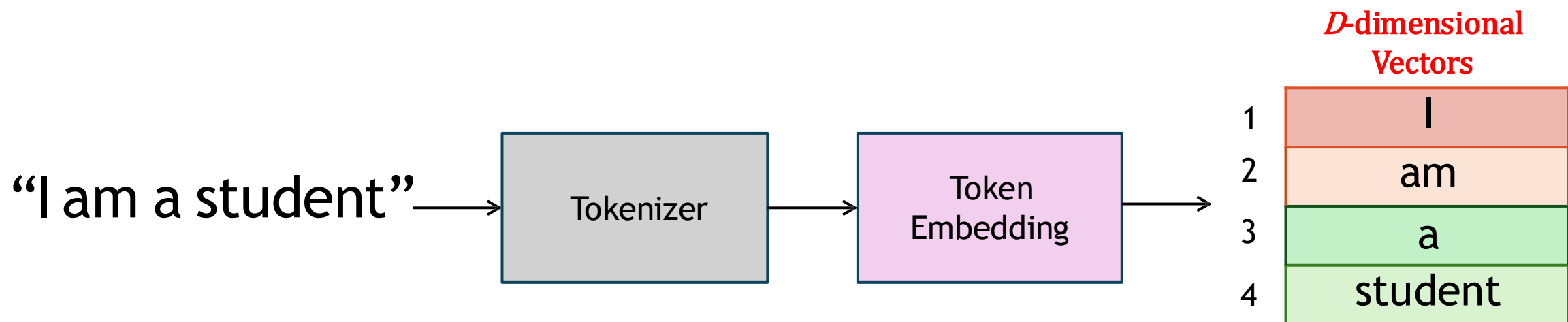
N-gram $P(w_n | w_1, \dots, w_{n-1}) \approx P(w_n | w_{n-N+1}, \dots, w_{n-1})$

Limitations: Can't capture long-range dependencies, curse of dimensionality, and lack of semantic abstraction/generalization

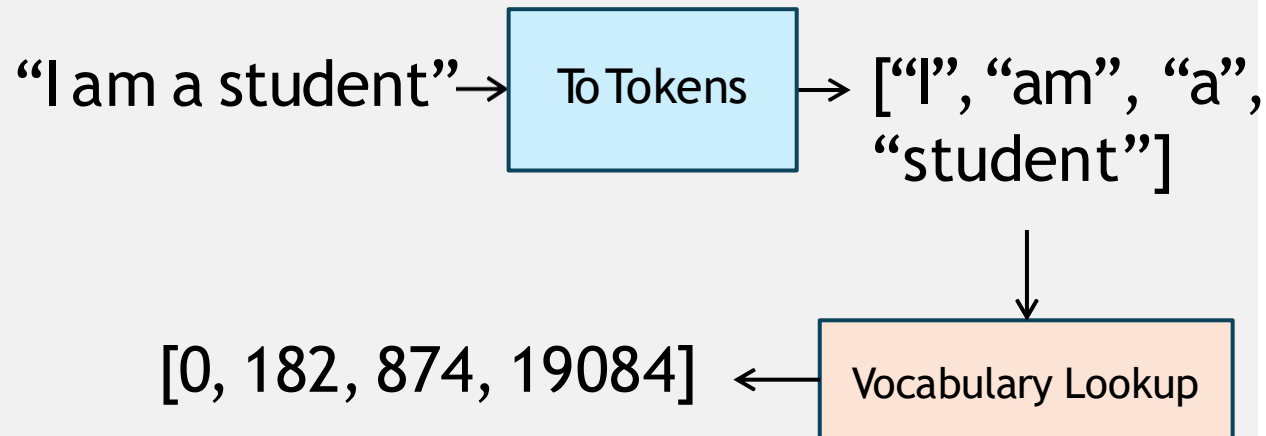
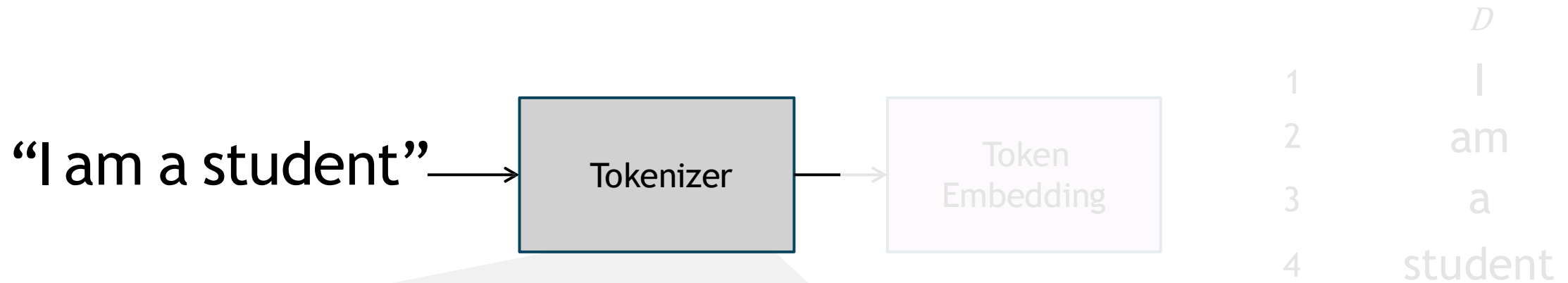
Recap: Modeling Tokens, not Words

$$P(W) = P(w_1, w_2, \dots, w_n) \quad w_1 \text{ to } w_n \text{ represent tokens}$$

Recap: Language to Tokens to Embeddings



Recap: Language to Tokens



The tokenizer breaks the text into tokens, basic elements of processing

Vocabulary is the collection of all tokens

Recap: A Few Simple Tokenizers

- **Properties**

- Balanced vocabulary size
- Out-of-Vocabulary or unknown tokens
- Language independence

- **A Few Simple Tokenizers**

- Word-level tokenization
- Character-level tokenization
- Morpheme-level tokenization

Subword Tokenization (BPE, WordPiece, Unigram)

- A data-driven approach of building a **vocabulary of pieces** (substrings) that:
 - includes very common words or morphemes as single tokens, and
 - can guarantee coverage by also including units so small (characters or bytes) that any new string can be expressed as a sequence of known pieces
- Because you can always fall back to smaller units, there's no true OOV: the model never needs a <unk>
- **Example: New word not seen in training “COVID”**
 - BPE: [“COV”, “ID”], if those merges exist; otherwise it falls back to smaller pieces, eventually characters

Byte Pair Encoding (BPE)

- BPE was designed as a **compression algorithm** in 1994 by Gage
- Sennrich et al, 2016 applied the BPE algorithm to produce one of the most commonly used **tokenizers**

Neural Machine Translation of Rare Words with Subword Units

Rico Sennrich and **Barry Haddow** and **Alexandra Birch**

School of Informatics, University of Edinburgh

`{rico.sennrich,a.birch}@ed.ac.uk,bhaddow@inf.ed.ac.uk`

BPE: Learning the Tokenizer

- It has two parts
 - Token **learner**: takes a raw training corpus and induces a vocabulary of tokens
 - Token **segmenter**: takes a raw test sentence and tokenizes it according to that vocabulary
- **Intuition**: start with **each character as its own token** and **combine tokens that co-occur** a lot

Byte Pair Encoding (BPE) - token learner

Let the initial vocabulary be the set of all individual characters

= {A, B, C, D, ..., a, b, c, d....}

- Repeat:

- Choose the **two symbols** that are **most frequently adjacent** in the training corpus (say 'A', 'B'); break ties randomly
- Add a new merged symbol 'AB' to the vocabulary
- Replace every adjacent 'A' 'B' in the corpus with 'AB'

- Until **k merges have been done**

BPE token learner algorithm

function BYTE-PAIR ENCODING(strings C , number of merges k) **returns** vocab V

$V \leftarrow$ all unique characters in C # initial set of tokens is characters

for $i = 1$ **to** k **do** # merge tokens til k times

$t_L, t_R \leftarrow$ Most frequent pair of adjacent tokens in C

$t_{NEW} \leftarrow t_L + t_R$ # make new token by concatenating

$V \leftarrow V + t_{NEW}$ # update the vocabulary

 Replace each occurrence of t_L, t_R in C with t_{NEW} # and update the corpus

return V

BPE token learner algorithm

function BYTE-PAIR ENCODING(strings C , number of merges k) **returns** vocab V

$V \leftarrow$ all unique characters in C # initial set of tokens is characters

for $i = 1$ **to** k **do** # merge tokens til k times

$t_L, t_R \leftarrow$ Most frequent pair of adjacent tokens in C

$t_{NEW} \leftarrow t_L + t_R$ # make new token by concatenating

$V \leftarrow V + t_{NEW}$ # update the vocabulary

 Replace each occurrence of t_L, t_R in C with t_{NEW} # and update the corpus

return V

BPE token learner algorithm

function BYTE-PAIR ENCODING(strings C , number of merges k) **returns** vocab V

$V \leftarrow$ all unique characters in C # initial set of tokens is characters

for $i = 1$ **to** k **do** # merge tokens til k times

$t_L, t_R \leftarrow$ Most frequent pair of adjacent tokens in C

$t_{NEW} \leftarrow t_L + t_R$ # make new token by concatenating

$V \leftarrow V + t_{NEW}$ # update the vocabulary

Replace each occurrence of t_L, t_R in C with t_{NEW} # and update the corpus

return V

BPE token learner algorithm

function BYTE-PAIR ENCODING(strings C , number of merges k) **returns** vocab V

$V \leftarrow$ all unique characters in C # initial set of tokens is characters

for $i = 1$ **to** k **do** # merge tokens til k times

$t_L, t_R \leftarrow$ Most frequent pair of adjacent tokens in C

$t_{NEW} \leftarrow t_L + t_R$ # make new token by concatenating

$V \leftarrow V + t_{NEW}$ # update the vocabulary

Replace each occurrence of t_L, t_R in C with t_{NEW} # and update the corpus

return V

Byte Pair Encoding (BPE) Addendum

- An additional complication is that in practice, BPE is **run only inside words**
- So, we commonly first add a **special end-of-word symbol '___'** before space in training corpus
 - The marker tells the algorithm where the word stops so it can learn word endings and also reconstruct spaces later

BPE token learner

Original corpus:

low low low low low lowest lowest newer newer newer newer newer newer
wider wider wider new new

Add end-of-word tokens, resulting in this vocabulary:

vocabulary

—, d, e, i, l, n, o, r, s, t, w

BPE token learner

corpus

5 l o w _
2 l o w e s t _
6 n e w e r _
3 w i d e r _
2 n e w _

vocabulary

_, d, e, i, l, n, o, r, s, t, w

Merge **e r** to **er**

corpus

5 l o w _
2 l o w e s t _
6 n e w e r _
3 w i d e r _
2 n e w _

vocabulary

_, d, e, i, l, n, o, r, s, t, w, er

BPE

corpus

5 l o w _
2 l o w e s t _
6 n e w e r _
3 w i d e r _
2 n e w _

vocabulary

_, d, e, i, l, n, o, r, s, t, w, e r

Merge **er _** to **er_**

corpus

5 l o w _
2 l o w e s t _
6 n e w e r_
3 w i d e r_
2 n e w _

vocabulary

, d, e, i, l, n, o, r, s, t, w, e r, e r

BPE

corpus

5 l o w _
2 l o w e s t _
6 n e w er_
3 w i d er_
2 n e w _

vocabulary

, d, e, i, l, n, o, r, s, t, w, er, er

Take a few mins to find the next 5 merges

Merge **n e** to **ne**

corpus

5 l o w _
2 l o w e s t _
6 ne w er_
3 w i d er_
2 ne w _

vocabulary

, d, e, i, l, n, o, r, s, t, w, er, er, ne

BPE

The next merges are:

Merge	Current Vocabulary
(ne, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new
(l, o)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo
(lo, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low
(new, er_)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_
(low, _)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_, low_

BPE token segmenter algorithm

On the **test data**, run each merge learned from the training data:

- Greedily + In the order we learned them (test frequencies don't play a role)

So: merge every **e r** to **er**, then merge **er _** to **er_**, etc.

- Result:

- Test set “newer_” would be tokenized as a full word
- Test set “lower_” would be two tokens: “**low** **er_**”

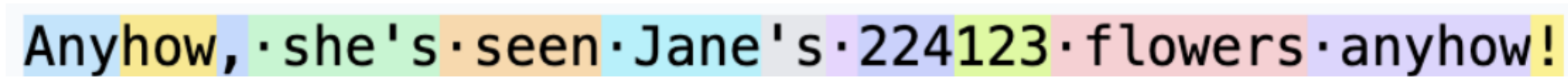
d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_, low_

Properties of BPE tokens

- Usually include frequent words
- And frequent subwords
 - Which are often morphemes like *-est* or *-er*

Tokenization Visualization

Checkout the following tokenizer to see the number of tokens in a given sentence for a system like OpenAI GPT4o: <https://tiktokenizer.vercel.app/>



Anyhow, · she ' s · seen · Jane ' s · 224123 · flowers · anyhow !

- The output is a sequence of unique token ids:
 - 11865, 8923, 11, 31211, 6177, 23919, 885, 220, 19427, 7633, 18887, 147065, 0

Questions to reflect on?

- How do decide the value of k in BPE?
- How does BPE deal with non-English languages?

Choosing k in BPE

- k is a hyperparameter: set by the desired vocabulary size (e.g., 30k / 50k / 100k tokens)

The tradeoff: efficiency vs. flexibility

- Larger vocab (bigger k) → fewer tokens per sentence, faster training and inference, but a larger embedding table and higher memory use
- Smaller vocab (smaller k) → more tokens per sentence, but a smaller embedding table and better handling of rare or unseen words

Typical ranges in practice

- Machine translation (early days): 30k–50k BPE merges
- LLMs (GPT, BERT, etc.): 30k–100k vocabulary; GPT-3 used ~50k tokens

BPE in Practice

- We have been showing BPE learning from sequences of **ASCII bytes**
- In practice, BPE is normally on the **individual bytes of UTF-8 encoded text**
 - We represent text as Unicode code points, encode it in UTF-8 bytes, and use these bytes as BPE input.

Agenda

- Quick Recap
- Tokenization
- Vector Embeddings

One Challenge

- How do we generalize better for unseen words/sentences?

Capturing Word Relationships

- Two words are *synonymous* if they are substitutable for one another in any sentence without changing the truth conditions of the sentence, the situations in which the sentence would be true
- Principle of contrast
- Relationship of *approximate or rough synonymy*

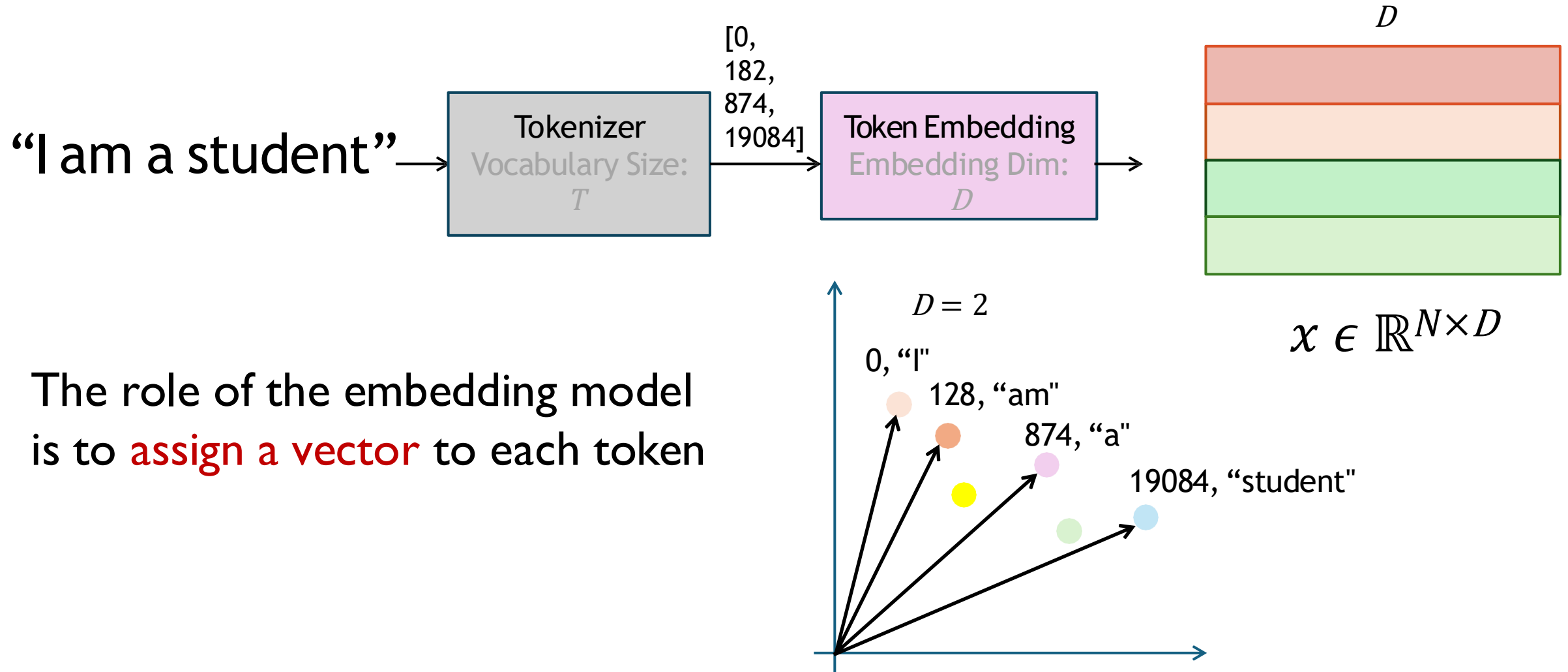
One Idea: Change Word Representation

Represent words in a **multi-dimensional semantic space**

Vector Embeddings

- Vector embeddings are **numerical representations** of objects (e.g., words) in a continuous vector space
- They capture the **semantic properties** of the objects, allowing similar objects to be closer in the space and dissimilar ones farther apart

Language to Embeddings



Strawman: Representing Vectors

- Vector of dimension V , where V is the vocabulary
- Any issues with such a vector representation?
- Consider the words king, queen, man and woman

```
king = [1, 0, 0, 0]  
queen = [0, 1, 0, 0]  
man = [0, 0, 1, 0]  
woman = [0, 0, 0, 1]
```

Dense Vector Spaces

- When using embeddings, the words are represented in a **dense vector space**
 - Typically, vectors with a few hundred to a few thousand dimensions
 - In this space, words with similar meanings or relationships are closer together
 - For example, the words **king** and **queen** might be **closer in space** because they represent similar objects (both royalty)

`king` = [0.5, 0.1, 0.3, ..., 0.8]

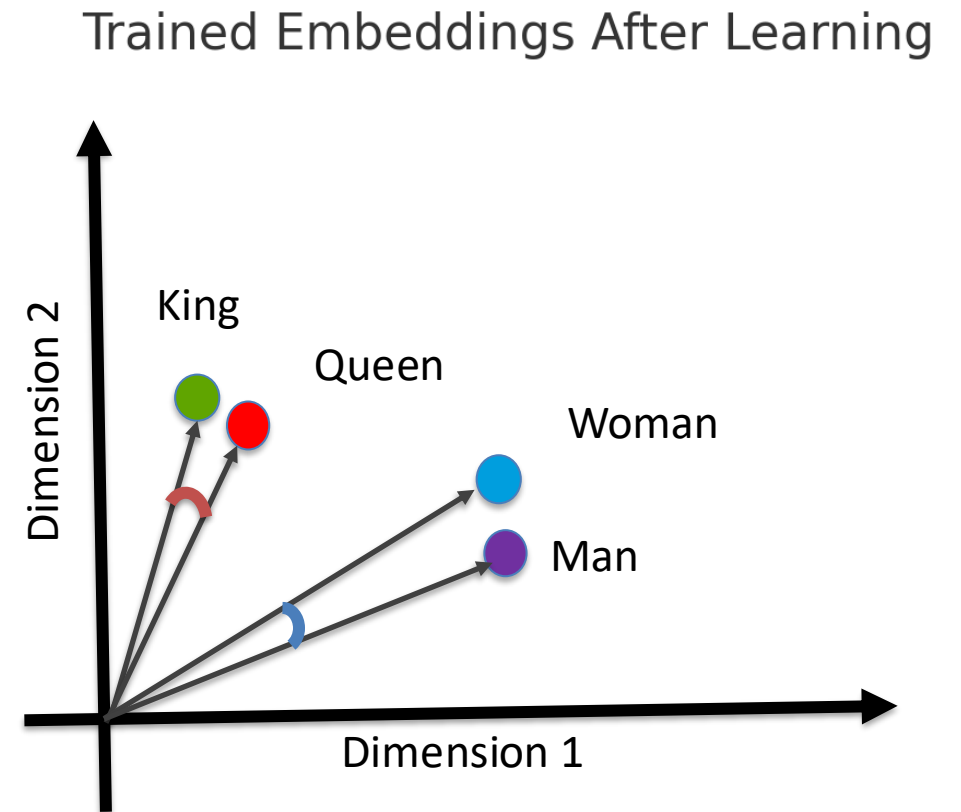
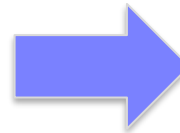
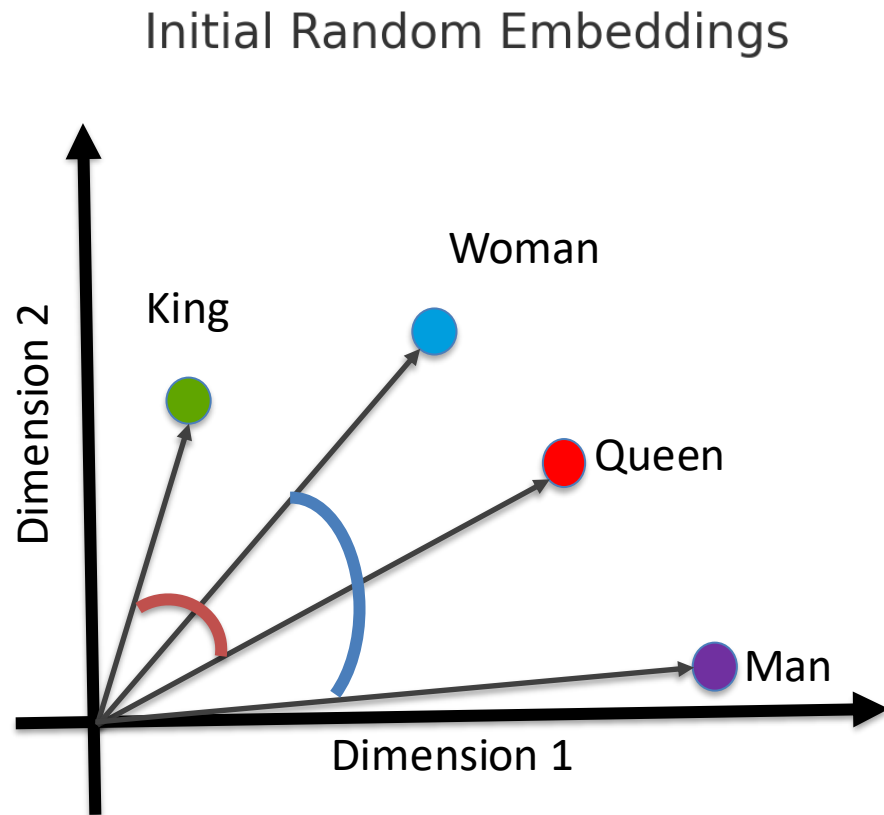
`queen` = [0.6, 0.15, 0.28, ..., 0.75]

`man` = [0.2, 0.8, 0.1, ..., 0.4]

`woman` = [0.25, 0.85, 0.12, ..., 0.45]

Learning Vector Embeddings

- After training, the embeddings form a **vector space** where the distance/angle between the vectors reflect the relationships between them



To learn vector embeddings, we first need a metric to capture similarity between two vector embeddings?

We need a metric that takes two vectors of the same dimensionality and gives a measure of their similarity

How about using the dot product?

$$\text{dot product}(\mathbf{v}, \mathbf{w}) = \mathbf{v} \cdot \mathbf{w} = \sum_{i=1}^N v_i w_i = v_1 w_1 + v_2 w_2 + \dots + v_N w_N$$

Any issues with using the dot product?

- **Norm leaks in:** the score grows with vector norms, not only with direction
- **Frequent words have large-norm vectors:** they co-occur with almost everything
- **Result:** frequency bias — common words look “similar” to unrelated words

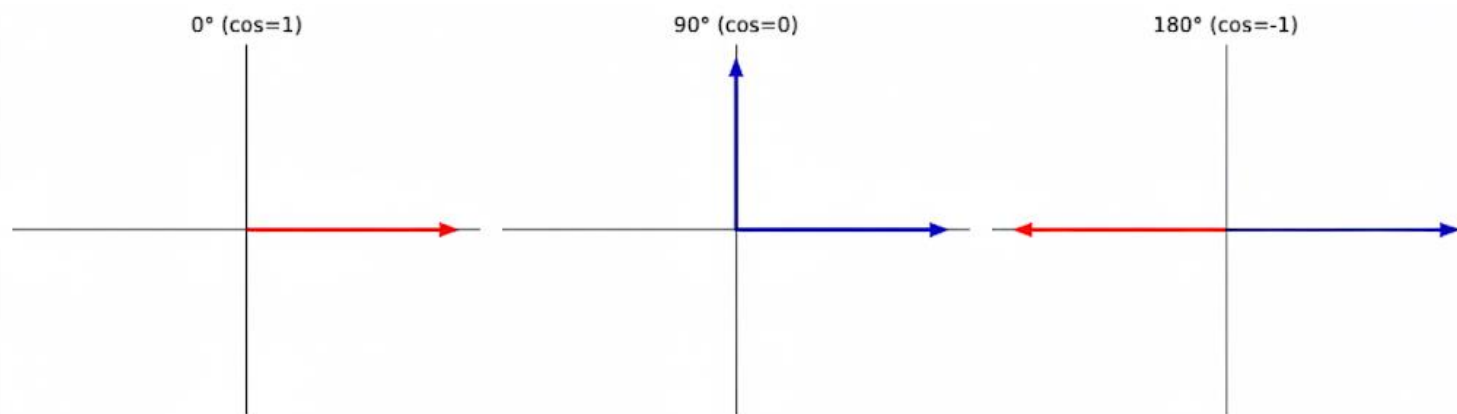
What we want: a measure that reflects direction and is independent of the norms

Cosine Similarity between Word Vectors

- When the model predicts words correctly, the **angle between their vectors** becomes smaller
 - Smaller angles = higher similarity = vectors closer in embedding space.

$$\text{cosine similarity} = \frac{v_1 \cdot v_2}{\|v_1\| \|v_2\|} = \cos(\theta)$$

θ	Cosine Similarity	Meaning
0°	1	Perfectly aligned (same meaning)
90°	0	Unrelated
180°	-1	Opposite meaning



Cosine Similarity, Angles and Dot Product

$$\mathbf{v}_1 \cdot \mathbf{v}_2$$

Intuitively, it can be thought of as a measure of how much two vectors “align” or “agree” with each other.

+

It helps us understand how much of $\mathbf{v}_1$ is in the direction of $\mathbf{v}_2$ or vice versa.

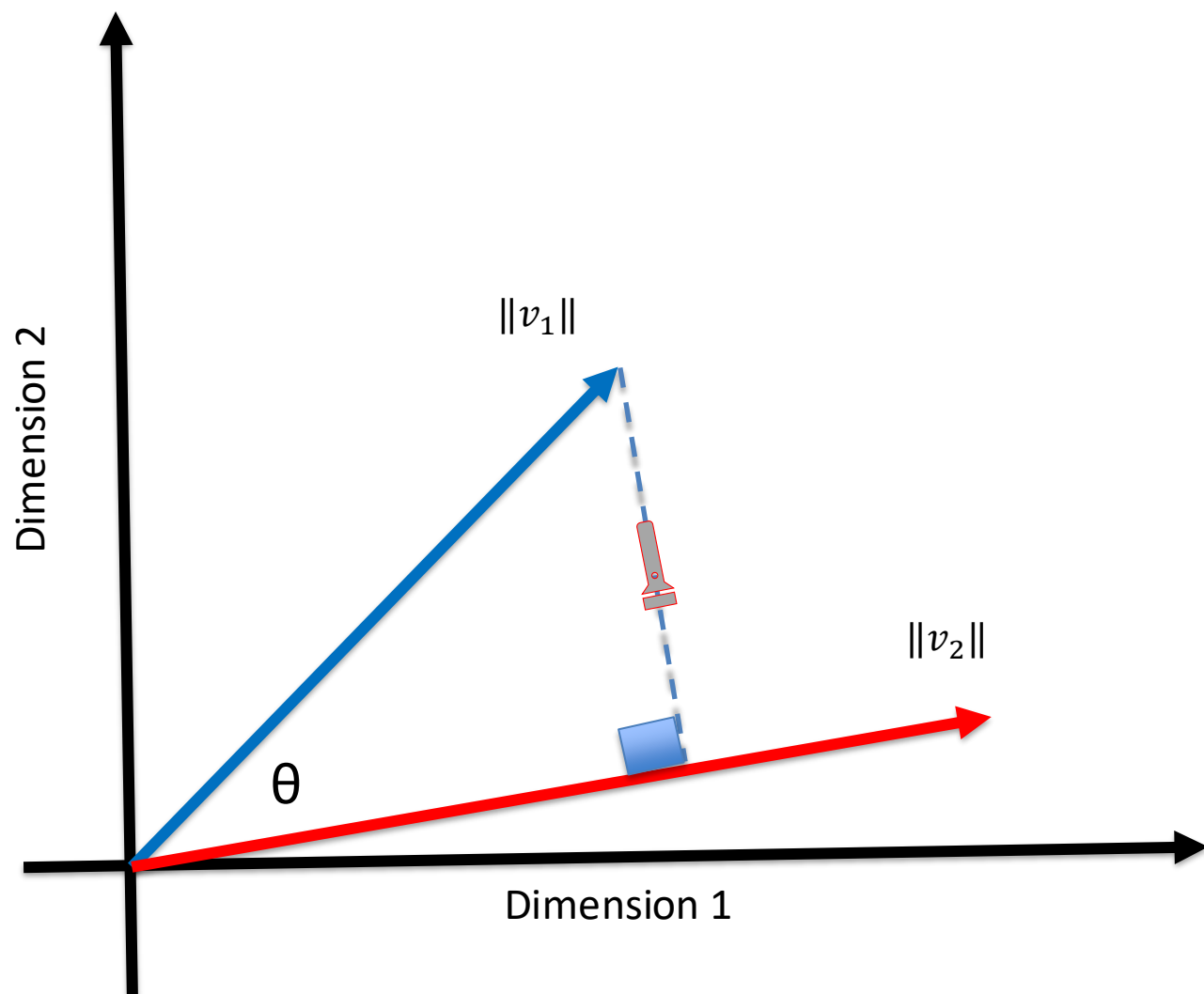
$$\mathbf{v}_1 \cdot \mathbf{v}_2 = v_{1x} \cdot v_{2x} + v_{1y} \cdot v_{2y} + \dots + v_{1n} \cdot v_{2n}$$

$$\|\mathbf{v}_1\| = \sqrt{v_{1x}^2 + v_{1y}^2 + \dots + v_{1n}^2}$$

$$\|\mathbf{v}_2\| = \sqrt{v_{2x}^2 + v_{2y}^2 + \dots + v_{2n}^2}$$

The **cosine of the angle** measures the alignment of the two vectors:

$$\mathbf{v}_1 \cdot \mathbf{v}_2 = \|\mathbf{v}_1\| \cdot \|\mathbf{v}_2\| \cdot \cos(\theta)$$



Take a 5 min break; Next

- How are vector embeddings learned?
- A probabilistic neural language model (Bengio et al., 2003)