

Artificial Neural Networks

Course-12

Gavrilit Dragos

rev 3

π

AGENDA FOR TODAY

- › Introduction
- › Tokenization
- › Transformers
 - Embeddings
 - Positional encoding
- › Attention mechanism
- › Feed Forward network
- › Prediction layer

Introduction

Introduction



How can we take a phrase, a sentence and translate it into some numbers (something that a computer might understand)

π

Introduction

So ... how can we test/validate if someone has learned something ?

The simple's way of doing this it to **ASK QUESTIONS** and **VALIDATE THE ANSWERS**

"Understanding is a cognitive process related to an abstract or physical object, such as a person, situation, or message whereby one is able to use concepts to model that object. Understanding is often, though not always, related to learning concepts, and sometimes also the theory or theories associated with those concepts."

wikipedia

A computer however

But what is **understanding** ?

*How can we take a phrase, a sentence and translate it into some numbers (something that a computer might **understand**)*

Introduction

- › So ... we are looking at a process where:
 1. We ask a question
 2. That question is somehow translated to numbers that a machine can understand
 3. The machine provides an answer based on the question we have provided
 4. We validate the answer.


Introduction

› So ... we are looking at a process where:

1. We ask a question

2. That question is converted into numbers that a machine can understand

This is where the actual **magic** happens !

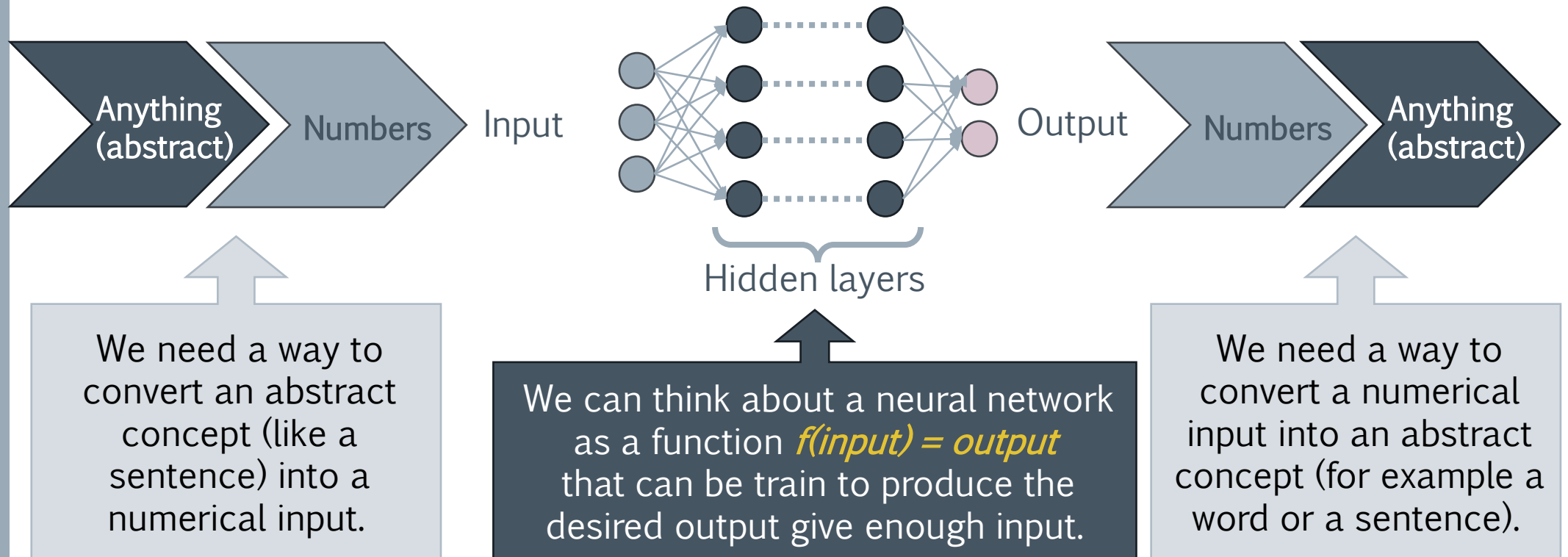


3. The machine provides an answer based on the question we have provided

4. We validate the answer.

Introduction

- › To answer the question: “*how can A machine provide an answer based on a question*” we must first discuss a couple of basic things about neural networks.

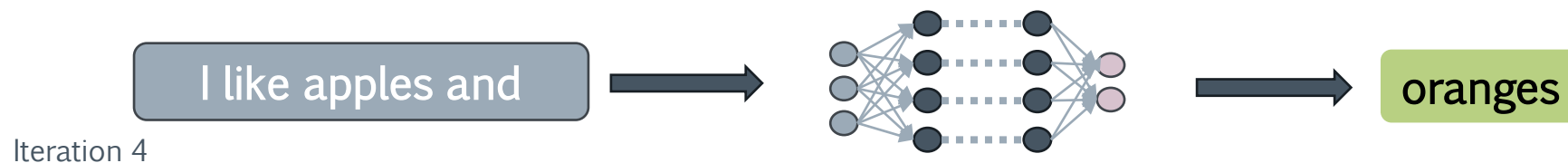
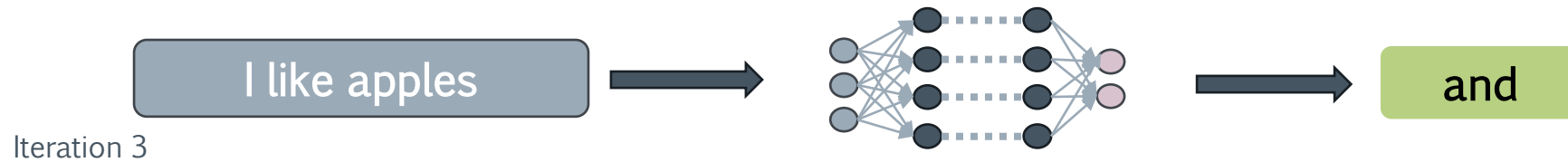
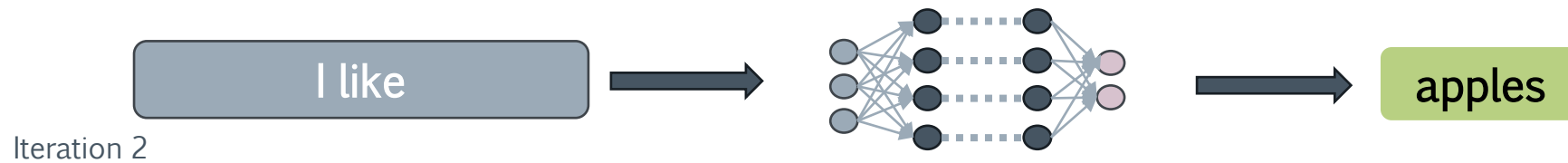
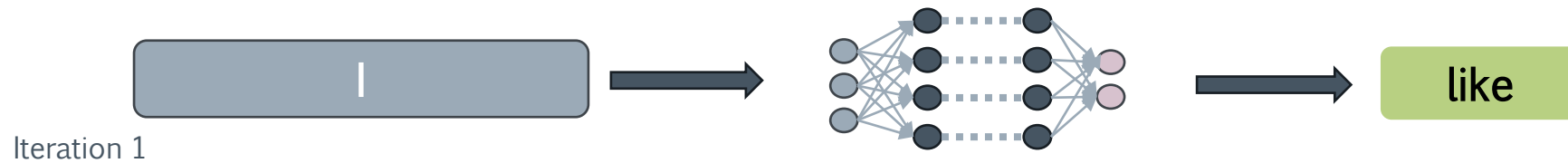


Introduction

- › So ... assuming we have a way to convert a questions into some numbers, the ability to create a generic function f that for a specific input (set of numbers) can produce the desire output (numeric values) and the ability to transform that output into another sentence, how do we build a system that can answer questions ?

Introduction

- › Let's take the following sentence: “*I like apples and oranges*” and we try to create a function that for each word predicts (outputs) the next one. In this case we will have:



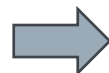
Introduction

- › Let's take the following sentence: “*I like apples and oranges*” and we try to create a function that for each word predicts (outputs) the next one. In this case we will have
- › This means that given *enough sentences* and a way to convert sentences into numeric values and vice-versa, we can train a model that can predict the next word from an incomplete sentence.

Introduction

- › So now we have way to *generate* words from an incomplete sentence. How can we make it answer a question ?
- › Well ... instead of feeding the model sentences, why not formulate them as a Q & A example:

I like apples and oranges



The response for the question “What do you like?”
is “*It like apples and oranges*”

Introduction

- › So ... after training the model, when we ask him the question “**What do you like?**” in reality we will send to the model the following sentence:

The response for the question “**What do you like?**” is

and just ask the model to generate the next word for this question 😊

... and the next word after that

... and the next work after that

...

Tokenization

Tokenization

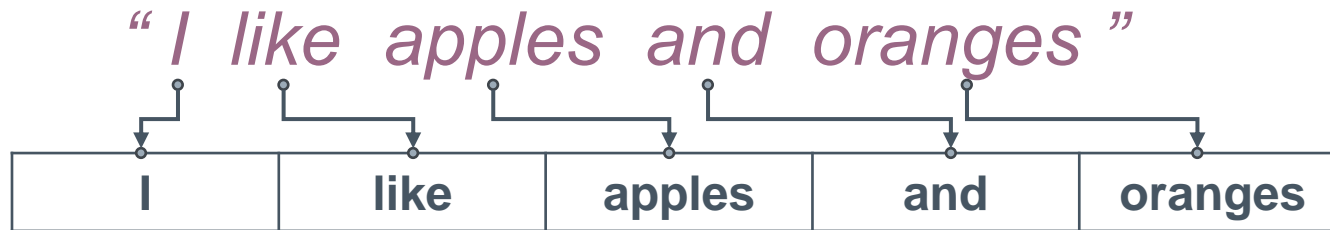
Let's discuss how we can convert a sentence into numbers

We will use the following sentence as an example for our task:

"I like apples and oranges"

Tokenization

First approach: let's split the sentence into words, and assign a number for each word, based on the word index within a vocabulary.

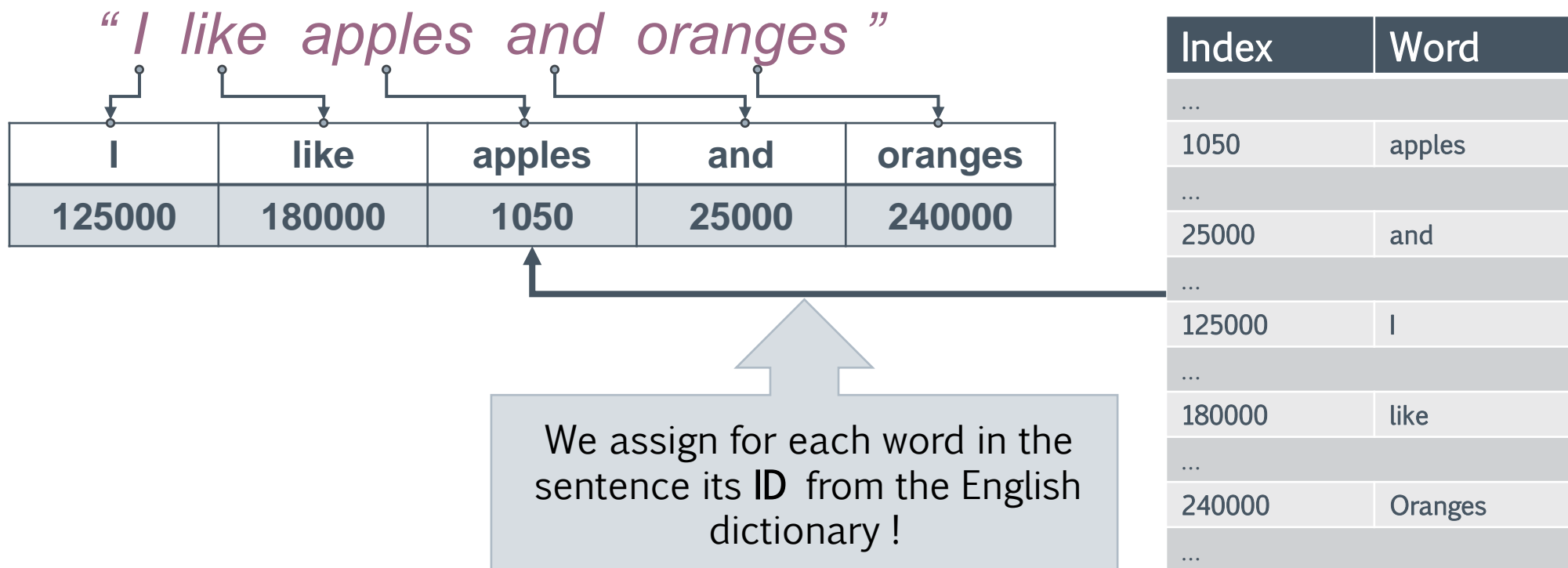


Index	Word
...	
1050	apples
...	
25000	and
...	
125000	I
...	
180000	like
...	
240000	Oranges
...	

According to Oxford English Dictionary 2nd edition, there are more than 270.000 words in English. We can create a map where for each word we have its index in the English dictionary

Tokenization

First approach: let's split the sentence into words, and assign a number for each word, based on the word index within a vocabulary.



Tokenization

First approach: let's split the sentence into words, and assign a number for each word, based on the word index within a vocabulary.

Sentence: *" I like apples and oranges "*

Vector: [12500,180000,1050,25000,240000]

This translation matrix is often referred to as **VOCABULARY**

Index	Word
...	
1050	apples
...	
25000	and
...	
125000	I
...	
180000	like
...	
240000	Oranges
...	

Tokenization

But is this good enough ?

What is differences between the next two sentences ?

Sentence - 1: “ *I like apples and oranges* ”

Sentence - 2: “ *I like apples and orangrs* ”

Tokenization

But is this good enough ?

What is differences between the next two sentences ?

Sentence - 1: *“ I like apples and oranges ”*

Sentence - 2: *“ I like apples and oranges ”*

Notice that the keys “e” and “r” are one next to each other. As such, it is possible (especially if typing on a phone) to make a mistake !



Tokenization

But is this good enough ?

Well → “*orangrs*” (with “r” instead of “e”) is not found in the English dictionary. As such, its representation will be a different number (usually a very large number that implies an unknown word).

“*I like apples and oranges*” → [12500, 180000, 1050, 25000, 240000]

but

“*I like apples and orangrs*” → [12500, 180000, 1050, 25000, +∞]

Tokenization

But is this good enough ?

- › What if we write two words incorrect ?
- › Or what if we write 3 or all ?
- › How could a model be able to work if the input data is incorrect ?

In reality, this simple tokenization technique is good from a theoretical point of view. We need another method, more resilient to convert a sentence into words.

Tokenization

Second approach: Character encoding (split everything at character level).

“ I like apples and oranges ”

I	—	l	i	k	e	—	a	p	p	l	e	s	—	a	n	d	—	o	r	r	a	n	g	e	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

and we can use ASCII/UNICODE codes to convert this to numerical values

I	—	l	i	k	e	—	a	p	p	l	e	s	—	a	n	d	—	o	r	r	a	n	g	e	s
73	32	108	105	107	101	32	97	112	112	108	101	115	32	97	110	100	32	111	114	114	97	110	103	101	115

Sentence: *“ I like apples and oranges ”*

Vector: [73,32,108,105,107,101,32,97,112,112,108,101,115,32,97,110,100,32,111,114,114,97,110,103,101,115]

Tokenization

Second approach: Character encoding (split everything at character level).

The main advantage with this method is that the method does not suffer from OOV (Out Of Vocabulary), meaning that since every word is form out of characters (ASCII or UNICODE) we can represent every word like this.

Tokenization

Second approach: Character encoding (split everything at character level).

However, the *disadvantages* are:

1. **The resulted vectors are too large** (we will end up making too many computation)
2. **No cross lingual benefits** (character “a” might be followed by “b” in one language and by “c” in another). If the training set contains example in multiple languages the result might be less efficient.
3. **No generalization** (words like happy or happiness or unhappiness are related and have a common component *happ*). However with character encodings we can not capture this characteristic.

Tokenization

Third approach: BPE (Byte Pair Encoding) method.

This method tries to break down words in subunits (form out of one or more characters) that are relevant.

For example: “happy”, “unhappy”, “happiness”, “unhappiness” will be break into the following components:

Word	SubUnit	SubUnit	SubUnit	Sub-unit
happy		happ	y	
unhappy	un	happ	y	
happiness		happ	i	ness
unhappiness	un	happ	i	ness

Tokenization

Third approach: BPE (Byte Pair Encoding) method.

This method tries to break down words in subunits (form out of one or more characters) that are relevant.

For example: “happiness”, “unhappiness” will be break into the subunits:

Notice that pieces of words that are common on multiple words are extracted as a subunit.

Word	SubUnit	SubUnit	SubUnit	Sub-unit
happy		happ	y	
unhappy	un	happ	y	
happiness		happ	i	ness
unhappiness	un	happ	i	ness

Tokenization

Third approach: BPE (Byte Pair Encoding) method.

This method tries to break down words in subunits (form out of one or more characters) that are relevant.

For example: “happy” will be break into the f

At the same time, suffixes and prefixes are also extracted (as they are relevant to multiple words).

Word	SubUnit	SubUnit	SubUnit	Sub-unit
happy		happ	y	
unhappy	un	happ	y	
happiness		happ	i	ness
unhappiness	un	happ	i	ness

Tokenization

Third approach: BPE (Byte Pair Encoding) method.


So ... how this algorithm works ?

1. Split all questions in the training example into words
2. Split every resulted words into characters
3. Search the most common pair of consecutive characters among all existing words
4. Merge that pair on all words and add it to a vocabulary
5. Repeat step 3 for a number of steps or until the vocabulary reaches a desired size.

Tokenization

Third approach: BPE (Byte Pair Encoding) method.

Let`s see a simple example.

- › We will consider the following words as part of our dictionary: "**low**", "**new**", "**lowest**", "**newest**", "**widest**", "**lower**"
- › First, we concatenate all and use a special separator  to indicate when a word ends. The result will be:

"**low****new****lowest****newest****widest****lower**

Tokenization

Third approach: BPE (Byte Pair Encoding) method.

Let`s see a simple example.

l	o	w	▢	n	e	w	▢	l	o	w	e	s	t	▢	n	e	w	e	s	t	▢	w	i	d	e	s	t	▢	l	o	w	e	r	▢
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Let`s see what is the sequence of two that is the most common:

π

Tokenization

Third approach: BPE (Byte Pair Encoding) method.

Let`s see a simple example.



Let's see what is the sequence of two that is the most common:

Seq	Count
“l” followed by “o”	3

Tokenization

Third approach: BPE (Byte Pair Encoding) method.

Let`s see a simple example.

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

Let`s see what is the sequence of two that is the most common:

Seq	Count
“l” followed by “o”	3
“o” followed by “w”	3


Tokenization

Third approach: BPE (Byte Pair Encoding) method.

Let's see a simple example.

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

Let's see what is the sequence of two that is the most common:

Seq	Count
"l" followed by "o"	3
"o" followed by "w"	3
"w" followed by " 	2



Tokenization

Third approach: BPE (Byte Pair Encoding) method.

Let's see a simple example.

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

Let's see what is the sequence of two that is the most common:

Seq	Count
“l” followed by “o”	3
“o” followed by “w”	3
“w” followed by “  ”	2
“  ” followed by “n”	2



Tokenization



Third approach: BPE (Byte Pair Encoding) method.



Let`s see a simple example.

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

Let`s see what is the sequence of two that is the most common:

Seq	Count
“l” followed by “o”	3
“o” followed by “w”	3
“w” followed by “  ”	2
“  ” followed by “n”	2
“n” followed by “e”	2
“e” followed by “w”	2

Seq	Count
“w” followed by “  ”	2
“  ” followed by “l”	2
“l” followed by “o”	2
“o” followed by “w”	2
“w” followed by “e”	3
“e” followed by “s”	3

Seq	Count
“s” followed by “t”	3
“t” followed by “  ”	3
“  ” followed by “w”	1
“w” followed by “i”	1
“i” followed by “d”	1
..... and so on	



Tokenization

Third approach: BPE (Byte Pair Encoding) method.




Let's see a simple example.

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

Let's see what is the sequence of two that is the most common:

Seq	Count
"l" followed by "o"	3
"o" followed by "w"	3
"w" followed by 	2
 followed by "n"	2
"n" followed by "e"	2
"e" followed by "w"	2

The first combination with the maximum number of appearances is "l" followed by "o" with **3 appearances**.

Seq	Count	Seq	Count
 followed by "l"	2	"l" followed by 	3
"l" followed by "o"	2	 followed by "w"	1
"o" followed by "w"	2	"w" followed by "i"	1
"w" followed by "e"	3	"i" followed by "d"	1
"e" followed by "s"	3 and so on	

Tokenization

Third approach: BPE (Byte Pair Encoding) method.

Let`s see a simple example.

lo	w		n	e	w		lo	w	e	s	t		n	e	w	e	s	t		w	i	d	e	s	t		lo	w	e	r	
----	---	--	---	---	---	--	----	---	---	---	---	--	---	---	---	---	---	---	--	---	---	---	---	---	---	--	----	---	---	---	--

Now we repeat the process ...

Tokenization

Third approach: BPE (Byte Pair Encoding) method.

Let`s see a simple example.

lo w new lo west new west widest lo wer

Now we repeat the process ...

Seq	Count
“lo” followed by “w”	3


Tokenization

Third approach: BPE (Byte Pair Encoding) method.

Let`s see a simple example.

lo w  n e w  lo w e s t  n e w e s t  w i d e s t  lo w e r 

Now we repeat the process ...

Seq	Count
“lo” followed by “w”	3
“w” followed by “  ”	2
..... and so on	

π

Tokenization

Third approach: BPE (Byte Pair Encoding) method.

Let`s see a simple example.

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

Now we repeat the process ...

Seq	Count
"lo" followed by "w"	3
"w" followed by "	2
..... and so on	

The first combination with the maximum number of appearances is "lo" followed by "w" with **3 appearances**.

Tokenization

Third approach: BPE (Byte Pair Encoding) method.

Let's see a simple example.

low	▤	n	e	w	▤	low	e	s	t	▤	n	e	w	e	s	t	▤	w	i	d	e	s	t	▤	low	e	r	▤
-----	---	---	---	---	---	-----	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	-----	---	---	---

› Next merge will be “e” with “s” (3 appearances)

low	▤	n	e	w	▤	low	es	t	▤	n	e	w	es	t	▤	w	i	d	es	t	▤	low	e	r	▤
-----	---	---	---	---	---	-----	----	---	---	---	---	---	----	---	---	---	---	---	----	---	---	-----	---	---	---

› Next merge will be “es” with “t” (3 appearances)

low	▤	n	e	w	▤	low	est	▤	n	e	w	est	▤	w	i	d	est	▤	low	e	r	▤
-----	---	---	---	---	---	-----	-----	---	---	---	---	-----	---	---	---	---	-----	---	-----	---	---	---

› Next merge will be “esr” with “▤” (3 appearances) **[this is a suffix]**

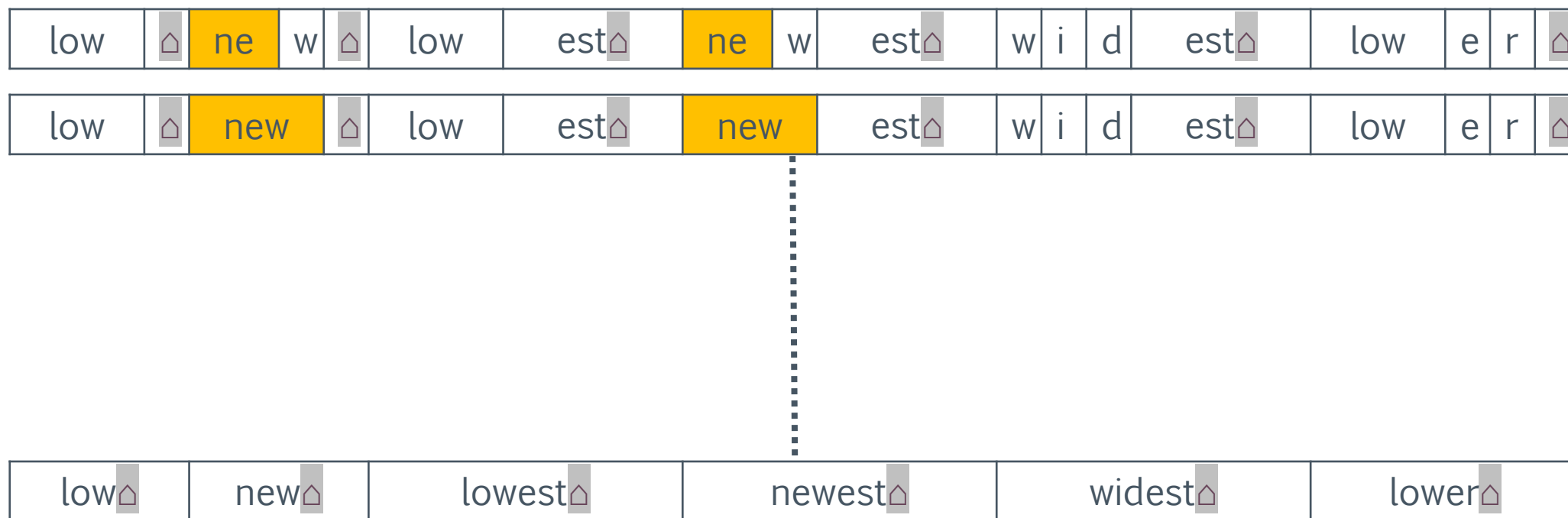
low	▤	n	e	w	▤	low	est▤	n	e	w	est▤	w	i	d	est▤	low	e	r	▤
-----	---	---	---	---	---	-----	------	---	---	---	------	---	---	---	------	-----	---	---	---

π

Tokenization

Third approach: BPE (Byte Pair Encoding) method.


Let's see a simple example.



Tokenization

Third approach: BPE (Byte Pair Encoding) method.

Keep in mind that on every step we **STORE** the rule (meaning that for the previous example, at the end of the process we will have a list of rules that we have used):

- “l” followed by “o”
- “lo” followed by “w”
- “w” followed by “”
- ... and soon

These rules will be used in the reverse process (converting a sentence into smaller subunits), and the vocabulary in this case will be form out of these sub-units.

Tokenization

Third approach: BPE (Byte Pair Encoding) method.

Advantages:

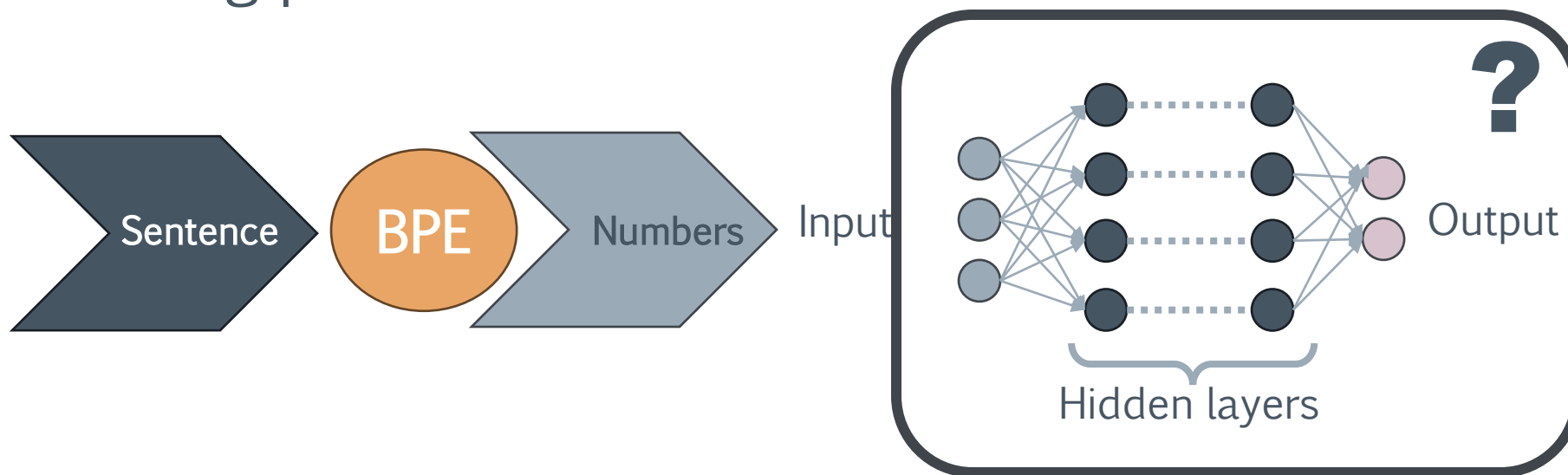
1. **Cross-lingual benefits** (identifying sub-units of words can help separate between languages)
2. **Linguistic Generalization** (the way the words are split can capture word roots, suffixes, prefixes, etc).
3. **Resilient to errors** (typos might impact a subunit, but not the entire word).
4. **Resilient to OOW** (unknown words are split in subunits – and in worst case scenario, characters)
5. **Domain specific** (if the training set is specific to a domain, the subunits will be specific as well)

Transformers

Prepare the data

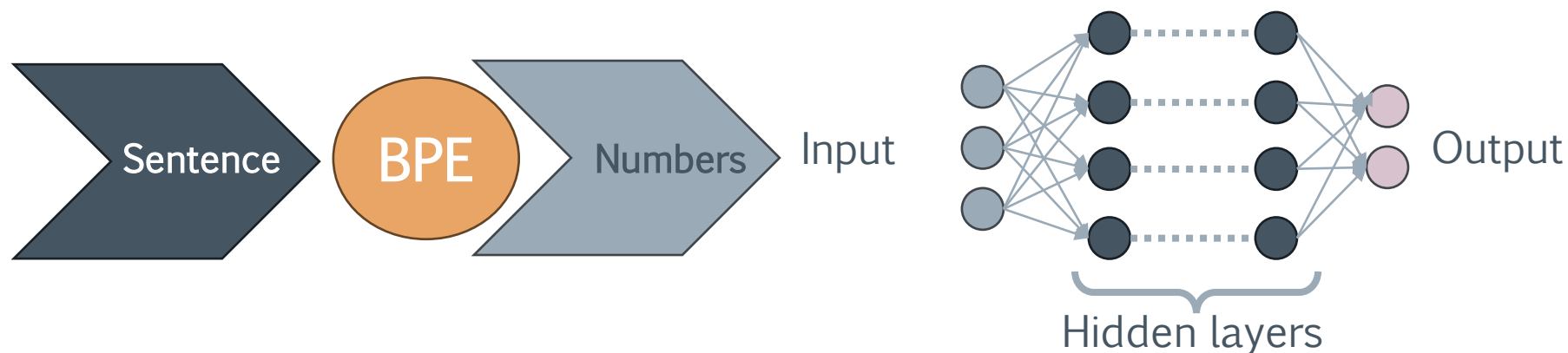
Transformers

- › Up to this moment, we know how to convert a text into a vector of numbers (using BPE).
- › The next step is to figure out how the model from the following picture works:



Transformers

- › So ... we have a way of converting a sentence to a vector of numbers (so that we can send it to a neural network)



- › But, the values that we get, while are **unique** between themselves, store no meaning / no relation one to another.

Embeddings

- › In other words, let's consider the sentence “I like apples” and let's assume that instead of using BPE we just split the sentence in words (for simplicity).
- › Then:
 - “I like apples” \Rightarrow [1,2,3] (where 1=“I”, 2=“like” and 3=“apples”)
 - “I like apples” \Rightarrow [3,1,9] (where 3=“I”, 1=“like” and 9=“apples”)
 - “I like apples” \Rightarrow [5,2,1] (where 5=“I”, 2=“like” and 1=“apples”)
 - *(other combinations)* ...
- › In practice, these numbers hold no real meaning, no context, no relationship (they are just an arbitrary decision to assign indexes)

Embeddings

- › This means that those relations and meanings between words would have to be created by us (or obtained via training).
- › This is done through “**word embeddings**” / “**byte-pair embeddings**” – a way to map the index of a word (or byte-pair) into another space.
- › These embeddings have different sizes (depending on the model). For example, BERT uses embeddings with sizes from 768 to 1024.
- › You can also use pre-train embeddings (e.g. GloVE, word2vec,)

Embeddings

- › Let's take two simple sentences and split them into words and discuss what kind of relationship we would like to have between the words:
- › Sentence 1: “**I love apples**”
Sentence 2: “**You love oranges**”
- › Now let's assume that the vector conversion for these two sentences is as follows:
 - “**I love apples**” \Rightarrow [1,2,3]
 - “**You love oranges**” \Rightarrow [4,2,5]

with: “I” = 1, “love” = 2, “apples” = 3, “You” = 4 and “oranges” = 5

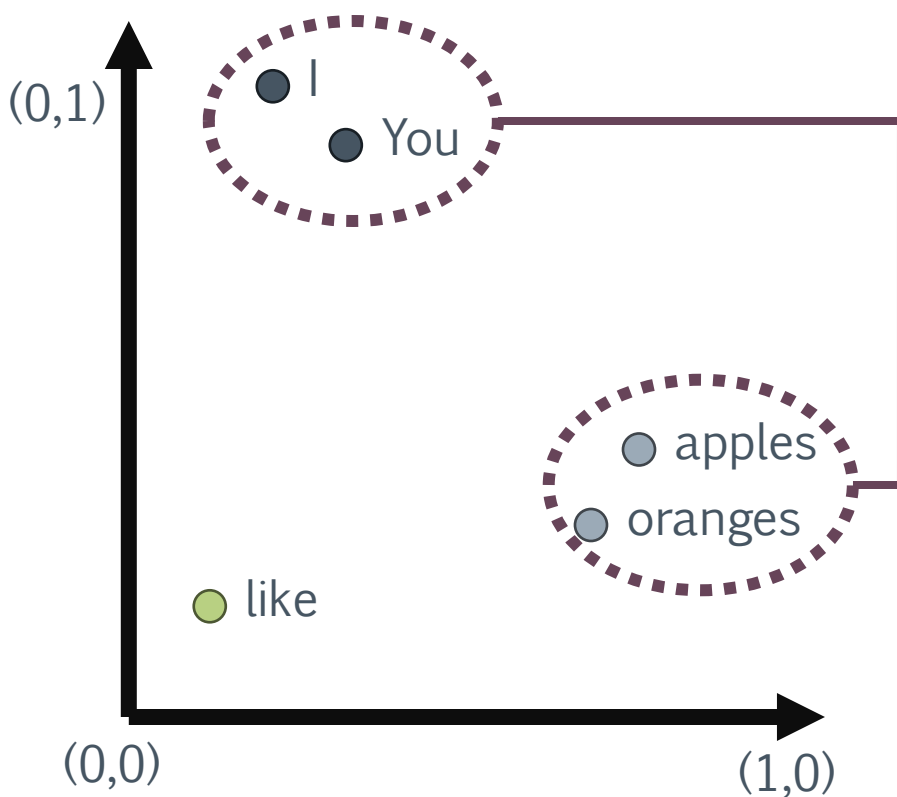
Embeddings

- › So .. what kind of relationships we would like to represent for “I love apples” and “You love oranges”
 - “apples” and “oranges” are something that people love
 - “You” and “I” refers to people
- › *But having a bunch of numbers like 1,2,3,4,2,5 means nothing and does not reflect the desired relationships.*

π

Embeddings

- › What we would like to capture looks more like in the next picture:

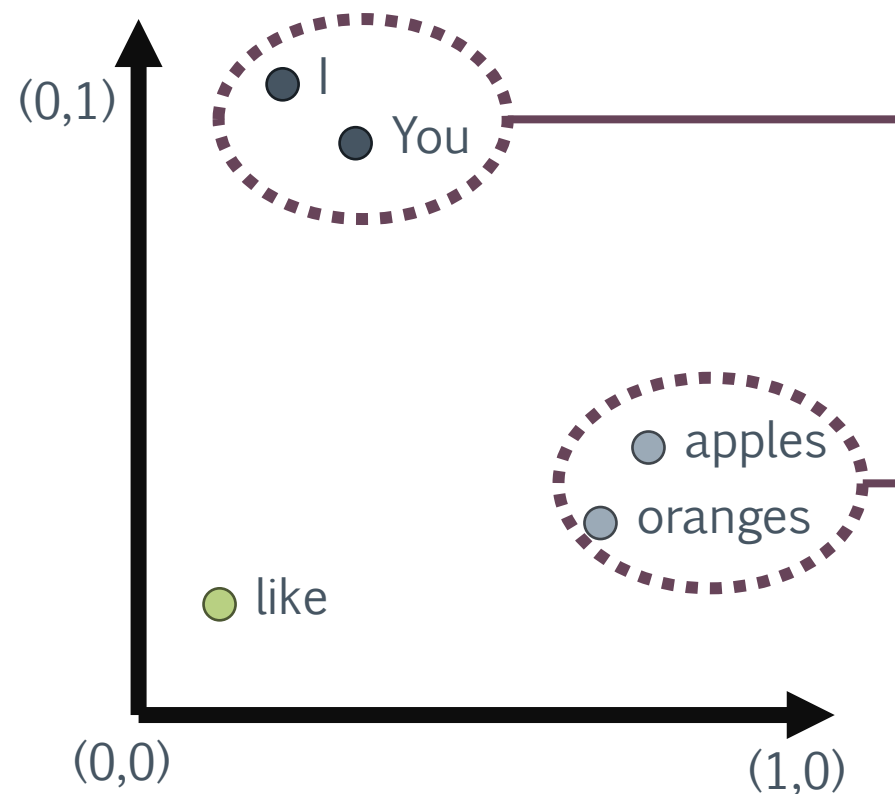


Notice that words that relate one to another
are clustered together

Embeddings

- › So ... we can represent the word in those two sentences with indexes or with coordinates in a two-dimensional space:

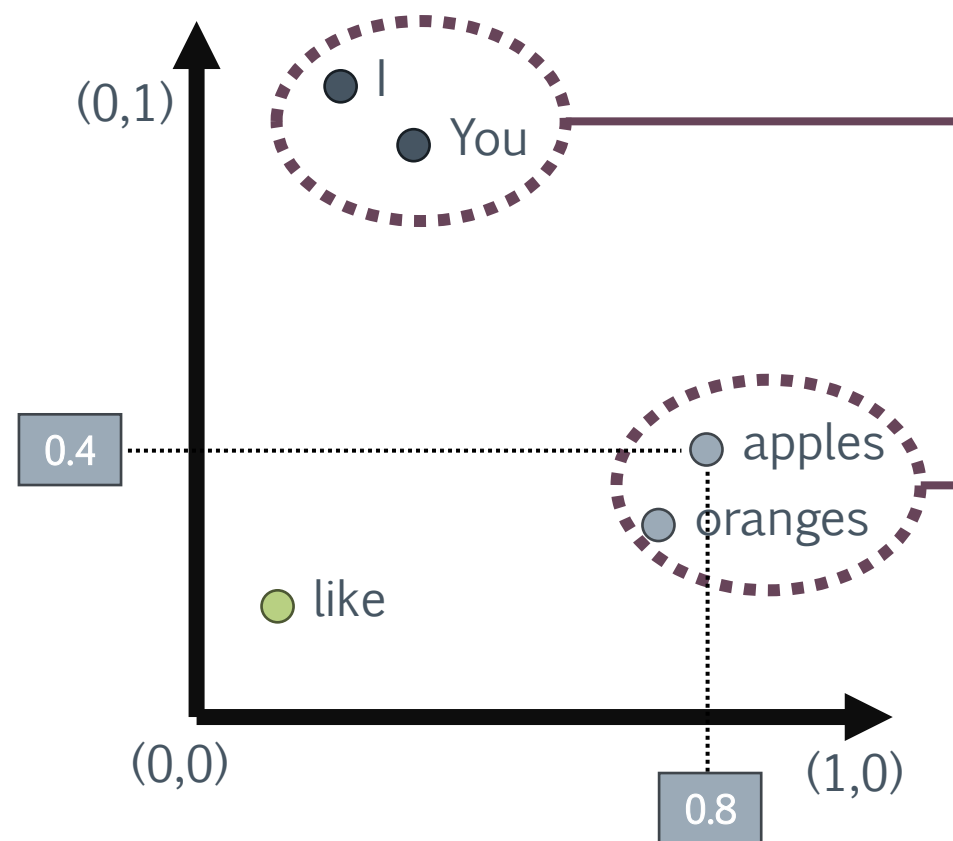
Word	Index	X	Y
apples	3		
I	1		
like	2		
oranges	5		
you	4		



Embeddings

- › So ... we can represent the word in those two sentences with indexes or with coordinates in a two-dimensional space:

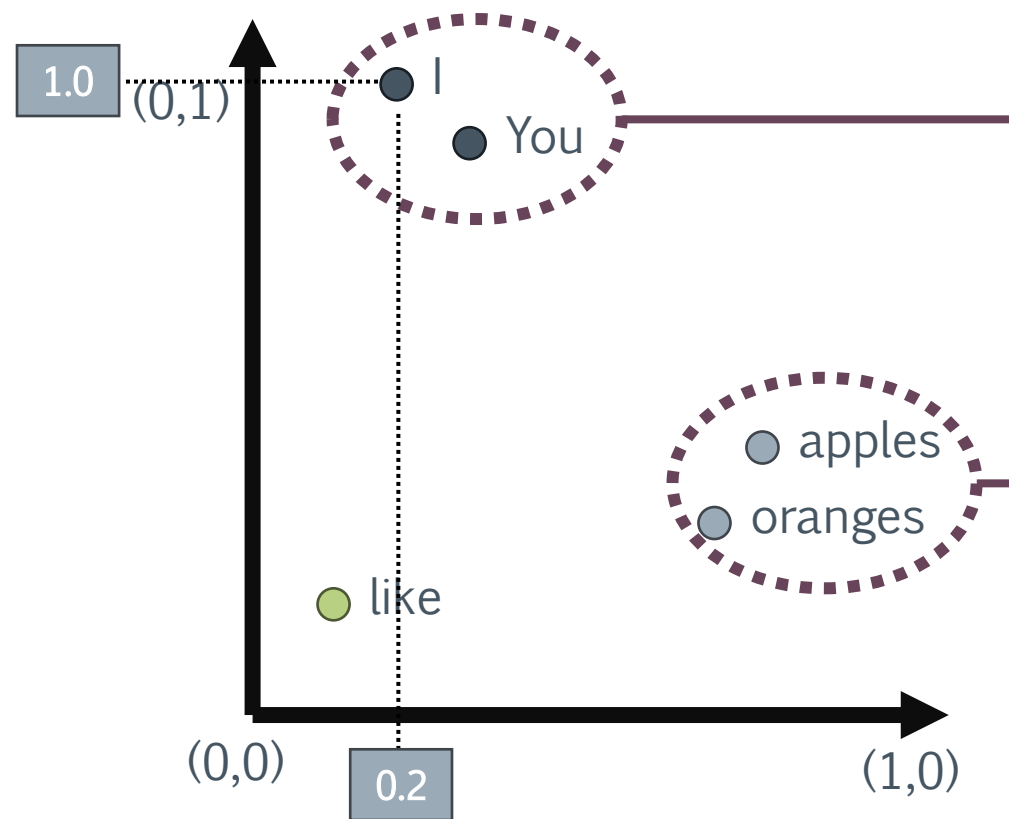
Word	Index	X	Y
apples	3	0.8	0.4
I	1		
like	2		
oranges	5		
you	4		



Embeddings

- › So ... we can represent the word in those two sentences with indexes or with coordinates in a two-dimensional space:


Word	Index	X	Y
apples	3	0.8	0.4
I	1	0.2	1.0
like	2		
oranges	5		
you	4		



Embeddings

- › So ... we can represent the word in those two sentences with indexes or with coordinates in a two-dimensional space:

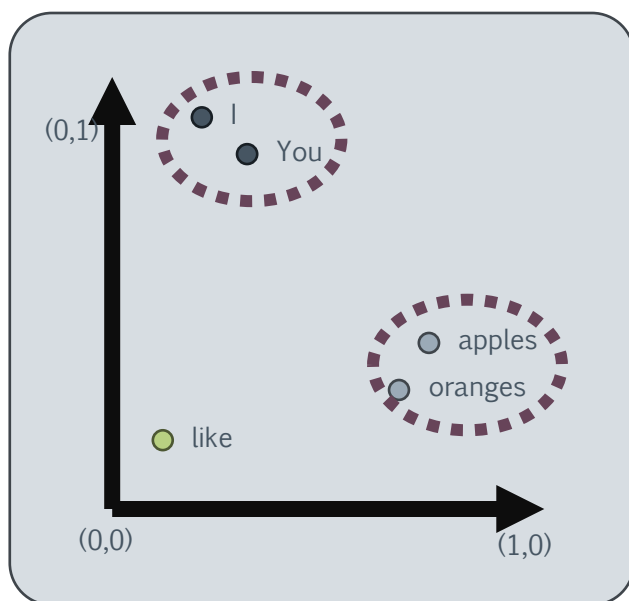
Word	Index	X	Y
apples	3	0.8	0.4
I	1	0.2	1.0
like	2	0.1	0.1
oranges	5	0.7	0.3
you	4	0.4	0.9



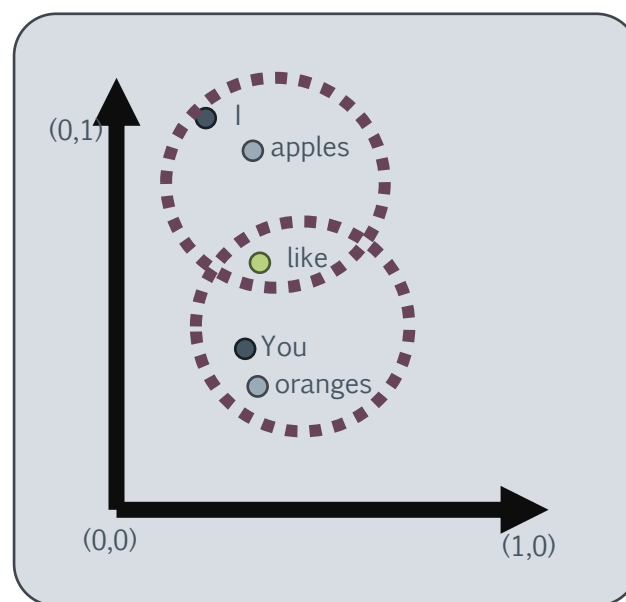
In reality, if for each word instead of using the index, we use (in our case) a two-dimensional coordinate, we store/provide more information on the relations of that word to another.

Embeddings

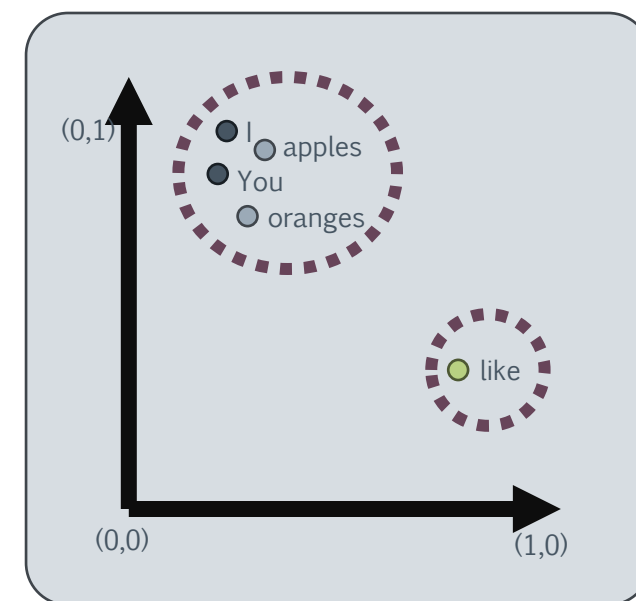
› But ... there are many relations that could be capture:



This is one where we group base on similarity



In this case we group base on how sentence look like



Or we can group them base on generic associations

Embeddings

- › The reality is that using a two-dimensional space is not enough to capture all possible relations between words. As such, an “*n-dimensional*” space is preferred.


Word	Index	Dim ₁	Dim ₂	Dim ₃	Dim ₄	Dim ₅	Dim ₆	...	Dim _n
apples	3	0.8	0.4	0.2	0.5	0.3	0.6	...	0.9
I	1	0.2	1.0	0.4	0.1	0.6	0.3	...	0.4
like	2	0.1	0.1	0.7	0.8	0.1	0.9	...	0.8
oranges	5	0.7	0.3	0.1		1.0	1.0	...	0.4
you	4								1.0

The vector that contains all of the values for each dimension of the space is called **embedding** (either word or byte-pair embedding). In our case, the embedding for word “I” will be [0.2, 1.0, 0.4, 0.1, 0.6, 0.3, ... 0.4]


Embeddings

› So ... lets see what we have up to this point:

Word / Byte-pair	Index	Dim ₁	Dim ₂	Dim ₃	Dim ₄	...	Dim _k
Word ₁ / BytePair ₁	1	?	?	?	?	...	?
Word ₂ / BytePair ₂	2	?	?	?	?	...	?
Word ₃ / BytePair ₃	3	?	?	?	?	...	?
...	...	?	?	?	?	...	?
Word _n / BytePair _n	n	?	?	?	?	...	?



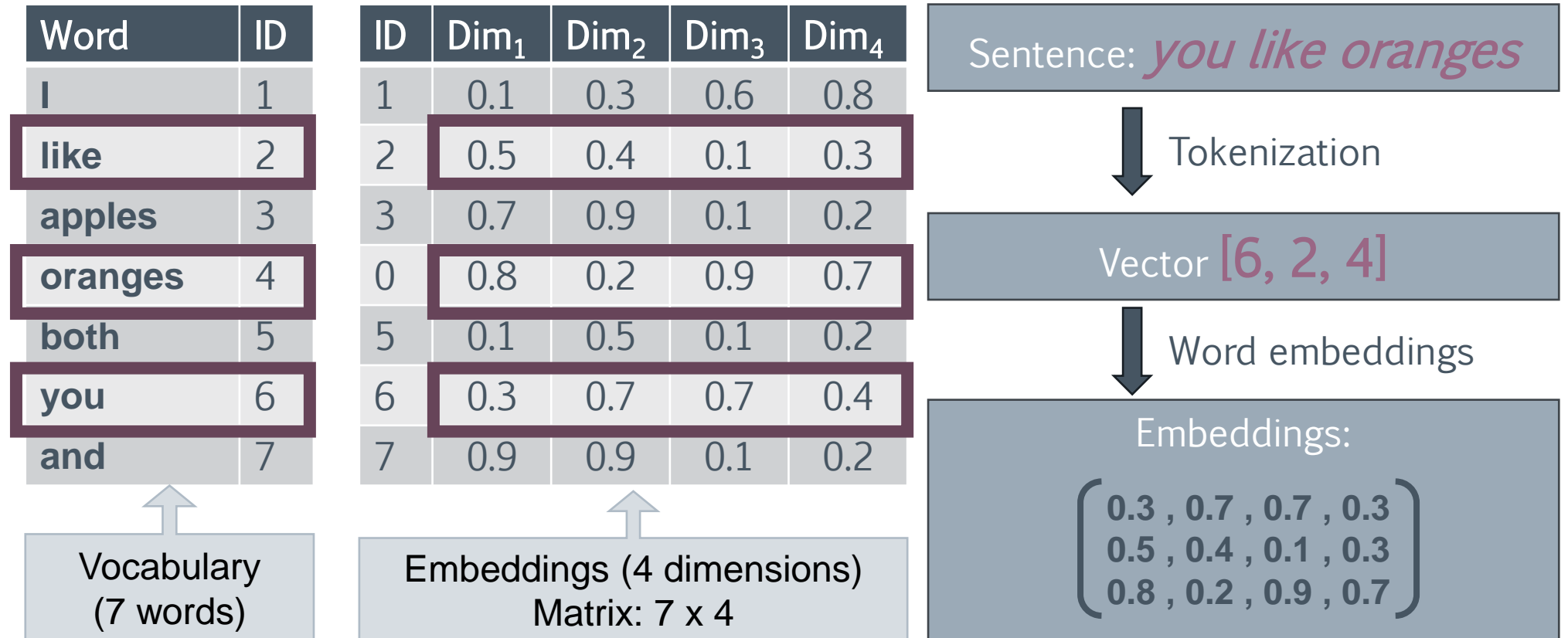
A vocabulary (a list of all words/byte-pairs and their associated index that are supported for our language).



An embedding (a vector of size “k”) for every word or byte-pair in the vocabulary. This vector can be precomputed, or can be initialized with random values and adjusted in the learning process.

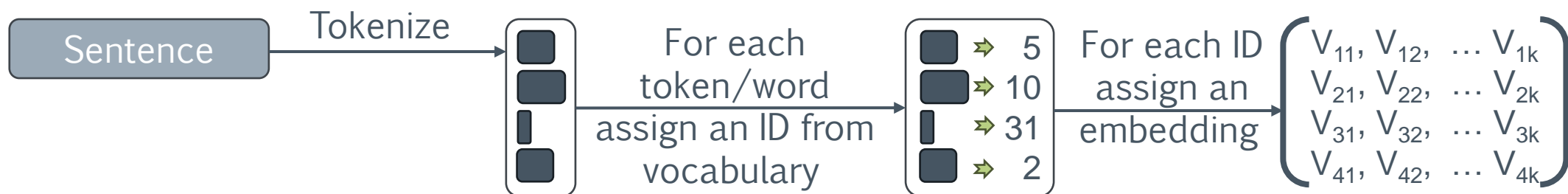
Embeddings

- › Let's see an example on how this process works. We will assume a vocabulary of 7 words and embeddings with dimension 4.



Embeddings

› So ... the process up to this point looks like this:

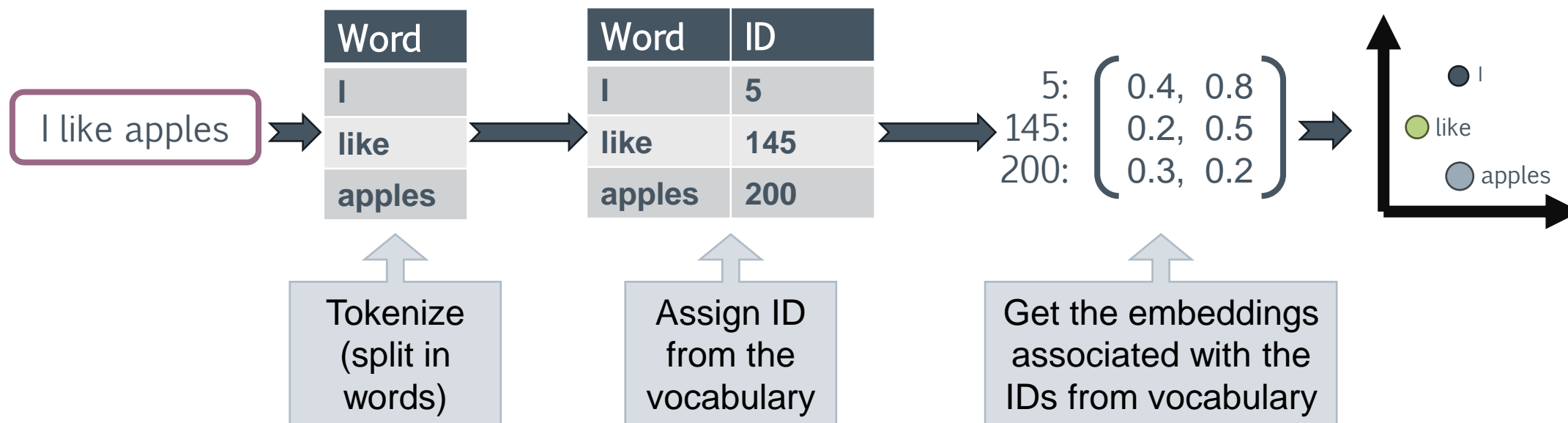


Positional encoding

- › Now that we have the embeddings, what is the next step ?
- › First , let's analyze the following example:
 - Tho sentences: “*I like apples*” and the “*apples like I*”
 - Embeddings in a 2-dimensional space
 - A vocabulary with 200 words
 - We just split words (we do not use BPE)
- › Let's see how the processing of the two sentences looks like:

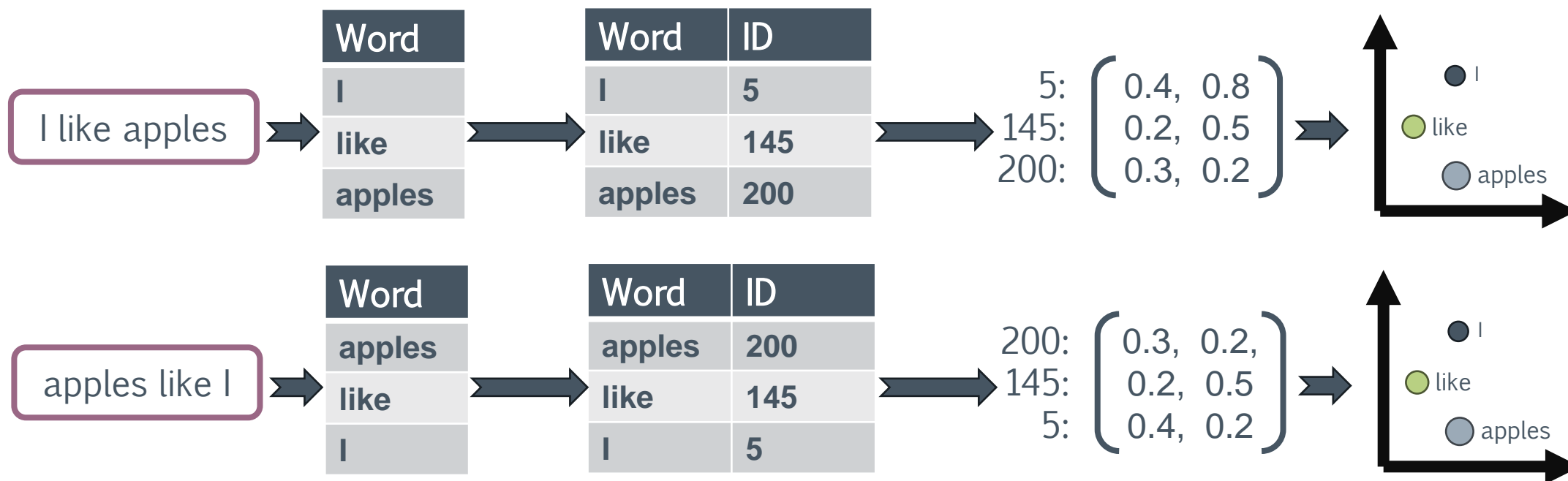
π

Positional encoding



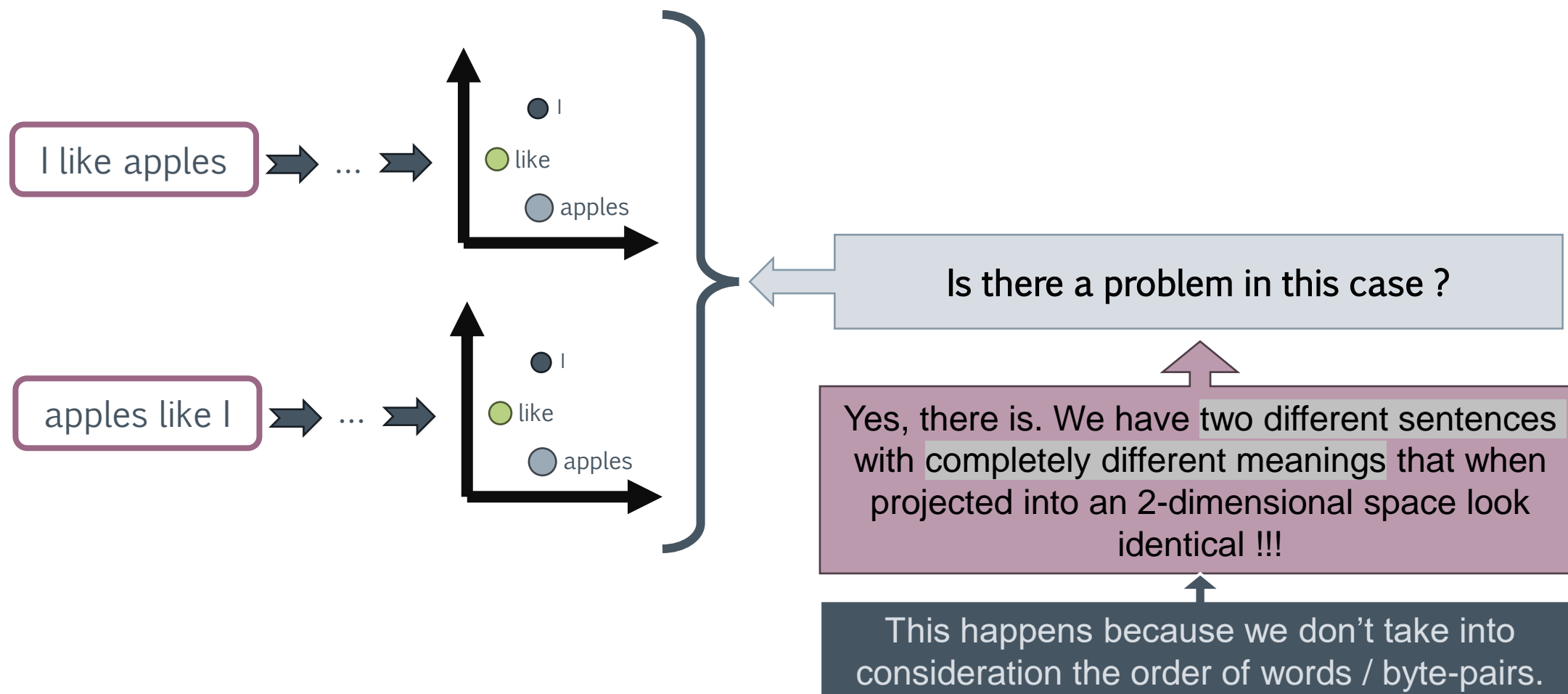
π

Positional encoding



π

Positional encoding



Positional encoding

- › This means that we would need to modify the embeddings so that they would reflect the order of words/byte pairs.
- › But ... we can not modify them too much as we don't want to disrupt any kind of relation between words that those embeddings have built.

Positional encoding

- › As a general concept, we can define positional encoding as a list of vectors (that have the same size as the word/byte-pair encoding vectors) that will be added to the embedding vector.

Positional encoding

- › Before we go deeper into what positional encoding is, let's explain a couple of notions.
 - When we send a sentence to such a model, that sentence is split into subunits (words, characters or byte-pairs)
 - However, not all sentences have the same number of subunits (for example: “*I love apples*” has 3 words while “*You and I love oranges*” have 5 words.
 - Further more, the input of the used model is fixed.

*The solution is to set up the maximum number of subunit that can be sent to a model at one time. To that extend some special tokens will be added to the vocabulary to reflect the start of a sentence, its end or padding. This maximum number of tokens reflect a “**SEQUENCE**” (a unit that measures how many tokens can be sent at one time to the model).*

Positional encoding

- › This means that upon tokenization, some sentences might be padded with a special token. Let's see some example where we consider the vocabulary of size 8 and the maximum sequence length of size 10.

Word	ID
I	1
like	2
apples	3
oranges	4
both	5
you	6
and	7
<PAD>	-1

Example 1: *I like apples*

Tokens	I	like	apples	<PAD>	<PAD>	<PAD>	<PAD>	<PAD>	<PAD>	<PAD>
Vector	1	2	3	-1	-1	-1	-1	-1	-1	-1

Example 2: *you and I like apples*

Tokens	you	and	I	like	apples	<PAD>	<PAD>	<PAD>	<PAD>	<PAD>
Vector	6	7	1	2	3	-1	-1	-1	-1	-1

Positional encoding

- › Now that we know that any sentence will be padded to fit the maximum sequence size, we can say that the position of any word in the sequence is **between 1 and maximum sequence size**.
- › This also means that the number of positional encoding vector is the maximum sequence size !

if $V = \begin{bmatrix} word_1 \\ word_2 \\ \dots \\ word_n \end{bmatrix}$ is the vocabulary with $<n>$ words,

and $E = \begin{bmatrix} value_{(1,1)} & \dots & value_{(1,d)} \\ \vdots & \ddots & \vdots \\ value_{(n,1)} & \dots & value_{(n,d)} \end{bmatrix}$ is the embedding matrix of size $n \times d$,

then $PE = \begin{bmatrix} pos_{(1,1)} & \dots & pos_{(1,d)} \\ \vdots & \ddots & \vdots \\ pos_{(k,1)} & \dots & pos_{(k,d)} \end{bmatrix}$ is the position encoding matrix of size $k \times d$,

π

Positional encoding

Let's see an example on how this process works. We will assume a vocabulary of 7 words and embeddings with dimension 4, with max sequence length of 10.

Word	ID
I	1
like	2
apples	3
oranges	4
both	5
you	6
and	7

Vocabulary
(7 words)

ID	Dim ₁	Dim ₂	Dim ₃	Dim ₄
1	0.1	0.3	0.6	0.8
2	0.5	0.4	0.1	0.3
3	0.7	0.9	0.1	0.2
0	0.8	0.2	0.9	0.7
5	0.1	0.5	0.1	0.2
6	0.3	0.7	0.7	0.4
7	0.9	0.9	0.1	0.2

Embeddings
(4 dimensions)

Order	Dim ₁	Dim ₂	Dim ₃	Dim ₄
First word	0.1	0.1	0.1	0.1
2 nd word	0.2	0.2	0.2	0.2
3 rd word	0.3	0.3	0.3	0.3
4 th word	0.4	0.4	0.4	0.4
5 th word	0.5	0.5	0.5	0.5
6 th word	0.6	0.6	0.6	0.6
7 th word	0.7	0.7	0.7	0.7
8 th word	0.8	0.8	0.8	0.8
9 th word	0.9	0.9	0.9	0.9
10 th word	1.0	1.0	1.0	1.0

Positional encodings (up to 10
words)

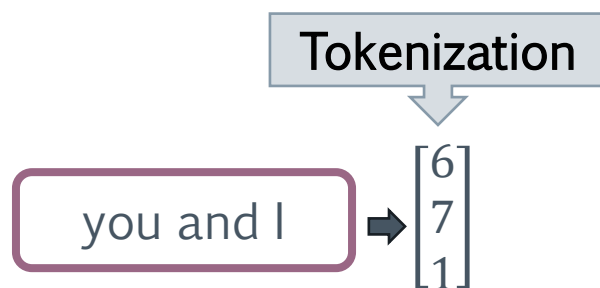
Positional encoding

Let's see an example on how this process works. We will assume a vocabulary of 7 words and embeddings with dimension 4, with max sequence length of 10.

Word	ID
I	1
like	2
apples	3
oranges	4
both	5
you	6
and	7

ID	Dim ₁	Dim ₂	Dim ₃	Dim ₄
1	0.1	0.3	0.6	0.8
2	0.5	0.4	0.1	0.3
3	0.7	0.9	0.1	0.2
0	0.8	0.2	0.9	0.7
5	0.1	0.5	0.1	0.2
6	0.3	0.7	0.7	0.4
7	0.9	0.9	0.1	0.2

Order	Dim ₁	Dim ₂	Dim ₃	Dim ₄
First word	0.1	0.1	0.1	0.1
2 nd word	0.2	0.2	0.2	0.2
3 rd word	0.3	0.3	0.3	0.3
4 th word	0.4	0.4	0.4	0.4
5 th word	0.5	0.5	0.5	0.5
6 th word	0.6	0.6	0.6	0.6
7 th word	0.7	0.7	0.7	0.7
8 th word	0.8	0.8	0.8	0.8
9 th word	0.9	0.9	0.9	0.9
10 th word	1.0	1.0	1.0	1.0



Positional encoding

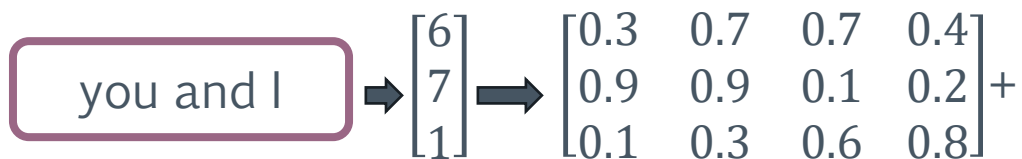
Let's see an example on how this process works. We will assume a vocabulary of 7 words and embeddings with dimension 4, with max sequence length of 10.

Word	ID
I	1
like	2
apples	3
oranges	4
both	5
you	6
and	7

ID	Dim ₁	Dim ₂	Dim ₃	Dim ₄
1	0.1	0.3	0.6	0.8
2	0.5	0.4	0.1	0.3
3	0.7	0.9	0.1	0.2
0	0.8	0.2	0.9	0.7
5	0.1	0.5	0.1	0.2
6	0.3	0.7	0.7	0.4
7	0.9	0.9	0.1	0.2

Order	Dim ₁	Dim ₂	Dim ₃	Dim ₄
First word	0.1	0.1	0.1	0.1
2 nd word	0.2	0.2	0.2	0.2
3 rd word	0.3	0.3	0.3	0.3
4 th word	0.4	0.4	0.4	0.4
5 th word	0.5	0.5	0.5	0.5
6 th word	0.6	0.6	0.6	0.6
7 th word	0.7	0.7	0.7	0.7
8 th word	0.8	0.8	0.8	0.8
9 th word	0.9	0.9	0.9	0.9
10 th word	1.0	1.0	1.0	1.0

Word Embeddings



Positional encoding

Let's see an example on how this process works. We will assume a vocabulary of 7 words and embeddings with dimension 4, with max sequence length of 10.

Word	ID
I	1
like	2
apples	3
oranges	4
both	5
you	6
and	7

ID	Dim ₁	Dim ₂	Dim ₃	Dim ₄
1	0.1	0.3	0.6	0.8
2	0.5	0.4	0.1	0.3
3	0.7	0.9	0.1	0.2
0	0.8	0.2	0.9	0.7
5	0.1	0.5	0.1	0.2
6	0.3	0.7	0.7	0.4
7	0.9	0.9	0.1	0.2

Order	Dim ₁	Dim ₂	Dim ₃	Dim ₄
First word	0.1	0.1	0.1	0.1
2 nd word	0.2	0.2	0.2	0.2
3 rd word	0.3	0.3	0.3	0.3
4 th word	0.4	0.4	0.4	0.4
5 th word	0.5	0.5	0.5	0.5
6 th word	0.6	0.6	0.6	0.6
7 th word	0.7	0.7	0.7	0.7
8 th word	0.8	0.8	0.8	0.8
9 th word	0.9	0.9	0.9	0.9
10 th word	1.0	1.0	1.0	1.0

“you” (ID: 6) is the **first word** (so we will add its embedding [0.3,0.7,0.7,0.4] the *first row* from the position encoding list)

$$\begin{array}{|c|} \hline \text{you and I} \\ \hline \end{array} \rightarrow \begin{bmatrix} 6 \\ 7 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0.3 & 0.7 & 0.7 & 0.4 \\ 0.9 & 0.9 & 0.1 & 0.2 \\ 0.1 & 0.3 & 0.6 & 0.8 \end{bmatrix} + \begin{bmatrix} 0.1 & 0.1 & 0.1 & 0.1 \end{bmatrix} =$$

Positional encoding

Let's see an example on how this process works. We will assume a vocabulary of 7 words and embeddings with dimension 4, with max sequence length of 10.

Word	ID
I	1
like	2
apples	3
oranges	4
both	5
you	6
and	7

ID	Dim ₁	Dim ₂	Dim ₃	Dim ₄
1	0.1	0.3	0.6	0.8
2	0.5	0.4	0.1	0.3
3	0.7	0.9	0.1	0.2
0	0.8	0.2	0.9	0.7
5	0.1	0.5	0.1	0.2
6	0.3	0.7	0.7	0.4
7	0.9	0.9	0.1	0.2

Order	Dim ₁	Dim ₂	Dim ₃	Dim ₄
First word	0.1	0.1	0.1	0.1
2 nd word	0.2	0.2	0.2	0.2
3 rd word	0.3	0.3	0.3	0.3
4 th word	0.4	0.4	0.4	0.4
5 th word	0.5	0.5	0.5	0.5
6 th word	0.6	0.6	0.6	0.6
7 th word	0.7	0.7	0.7	0.7
8 th word	0.8	0.8	0.8	0.8
9 th word	0.9	0.9	0.9	0.9
10 th word	1.0	1.0	1.0	1.0

“and” (ID: 7) is the **second word** (so we will add its embedding [0.9,0.9,0.1,0.2] the *second row* from the position encoding list)

you and I $\rightarrow \begin{bmatrix} 6 \\ 7 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0.3 & 0.7 & 0.7 & 0.4 \\ 0.9 & 0.9 & 0.1 & 0.2 \\ 0.1 & 0.3 & 0.6 & 0.8 \end{bmatrix} + \begin{bmatrix} 0.1 & 0.1 & 0.1 & 0.1 \\ 0.2 & 0.2 & 0.2 & 0.2 \end{bmatrix}$

Positional encoding

Let's see an example on how this process works. We will assume a vocabulary of 7 words and embeddings with dimension 4, with max sequence length of 10.

Word	ID
I	1
like	2
apples	3
oranges	4
both	5
you	6
and	7

ID	Dim ₁	Dim ₂	Dim ₃	Dim ₄
1	0.1	0.3	0.6	0.8
2	0.5	0.4	0.1	0.3
3	0.7	0.9	0.1	0.2
0	0.8	0.2	0.9	0.7
5	0.1	0.5	0.1	0.2
6	0.3	0.7	0.7	0.4
7	0.9	0.9	0.1	0.2

Order	Dim ₁	Dim ₂	Dim ₃	Dim ₄
First word	0.1	0.1	0.1	0.1
2 nd word	0.2	0.2	0.2	0.2
3 rd word	0.3	0.3	0.3	0.3
4 th word	0.4	0.4	0.4	0.4
5 th word	0.5	0.5	0.5	0.5
6 th word	0.6	0.6	0.6	0.6
7 th word	0.7	0.7	0.7	0.7
8 th word	0.8	0.8	0.8	0.8
9 th word	0.9	0.9	0.9	0.9
10 th word	1.0	1.0	1.0	1.0

Finally, for "I" (ID: 1) is the **third word** (so we will add its embedding [0.1,0.3,0.6,0.8] the *third row* from the position encoding list)

$$\begin{array}{|c|} \hline \text{you and I} \\ \hline \end{array} \rightarrow \begin{bmatrix} 6 \\ 7 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0.3 & 0.7 & 0.7 & 0.4 \\ 0.9 & 0.9 & 0.1 & 0.2 \\ 0.1 & 0.3 & 0.6 & 0.8 \end{bmatrix} + \begin{bmatrix} 0.1 & 0.1 & 0.1 & 0.1 \\ 0.2 & 0.2 & 0.2 & 0.2 \\ 0.3 & 0.3 & 0.3 & 0.3 \end{bmatrix} =$$

Positional encoding

Let's see an example on how this process works. We will assume a vocabulary of 7 words and embeddings with dimension 4, with max sequence length of 10.

Word	ID
I	1
like	2
apples	3
oranges	4
both	5
you	6
and	7

ID	Dim ₁	Dim ₂	Dim ₃	Dim ₄
1	0.1	0.3	0.6	0.8
2	0.5	0.4	0.1	0.3
3	0.7	0.9	0.1	0.2
0	0.8	0.2	0.9	0.7
5	0.1	0.5	0.1	0.2
6	0.3	0.7	0.7	0.4
7	0.9	0.9	0.1	0.2

Order	Dim ₁	Dim ₂	Dim ₃	Dim ₄
First word	0.1	0.1	0.1	0.1
2 nd word	0.2	0.2	0.2	0.2
3 rd word	0.3	0.3	0.3	0.3
4 th word	0.4	0.4	0.4	0.4
5 th word	0.5	0.5	0.5	0.5
6 th word	0.6	0.6	0.6	0.6
7 th word				
8 th word				
9 th word				
10 th word				

The final resulted matrix (or list of encodings)

$$\begin{array}{|c|} \hline \text{you and I} \\ \hline \end{array} \rightarrow \begin{bmatrix} 6 \\ 7 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0.3 & 0.7 & 0.7 & 0.4 \\ 0.9 & 0.9 & 0.1 & 0.2 \\ 0.1 & 0.3 & 0.6 & 0.8 \end{bmatrix} + \begin{bmatrix} 0.1 & 0.1 & 0.1 & 0.1 \\ 0.2 & 0.2 & 0.2 & 0.2 \\ 0.3 & 0.3 & 0.3 & 0.3 \end{bmatrix} = \begin{bmatrix} 0.4 & 0.8 & 0.8 & 0.5 \\ 1.1 & 1.1 & 0.3 & 0.4 \\ 0.4 & 0.6 & 0.9 & 1.1 \end{bmatrix}$$

Positional encoding

- › Now that we know how position encoding looks like, lets discuss how we should decide what values to use. Lets start with some rules:
 1. *Small Values*: values should be small enough so that they don't modify the original encodings too much.
 2. *Maintain order information*: values should be selected so that the 4th word and the 5th word are relatively close one to another
 3. *Dimensions variance*: an embedding is represented by a vector (with one value for different dimensions). If we modify each value with a constant we will not add variance among different relations between two words.

Positional encoding

› Let's see some possible options:

1. Use a linear equation (*first word is increased by k , second word is increased by $2 \times k$, third word by $3 \times k$...*)

Problems:

- *Even if “ k ” is small, if we have 100 words, $100 \times k$ could be high enough to change the encoding relationships*
- *We would apply the same value for all dimensions*
- *When “ k ” is large, differences between “ k ” and “ $k+1$ ” will be large as well.*

Positional encoding

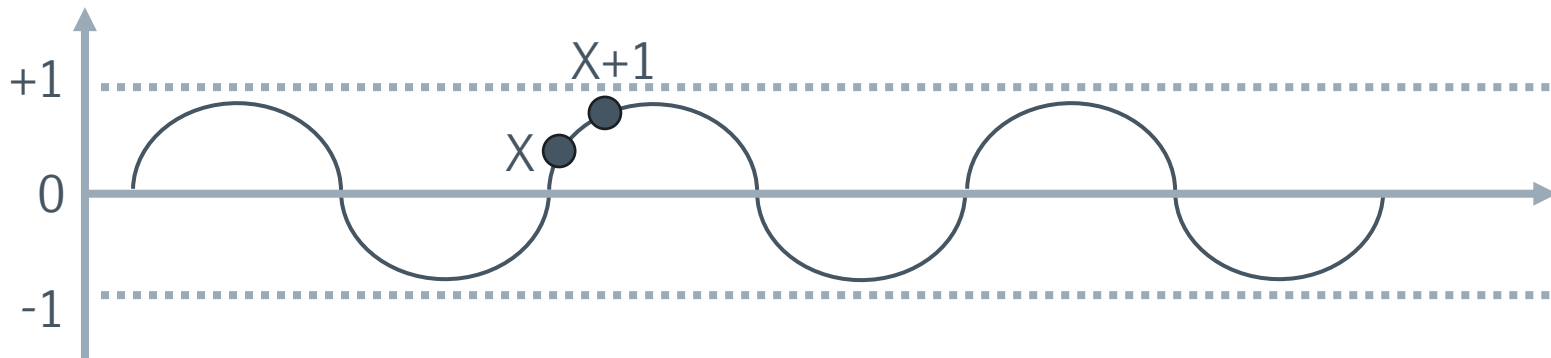
› Let's see some possible options:

2. Use *sin* or *cos* functions : we rely on the fact that :

- both ***sin*** and ***cos*** produce small values between -1 and 1
- ***sin***(x) and ***sin***($x+1$) are close to each other

Problems:

- We would apply the same value for all dimensions



Positional encoding

› Let's see some possible options:

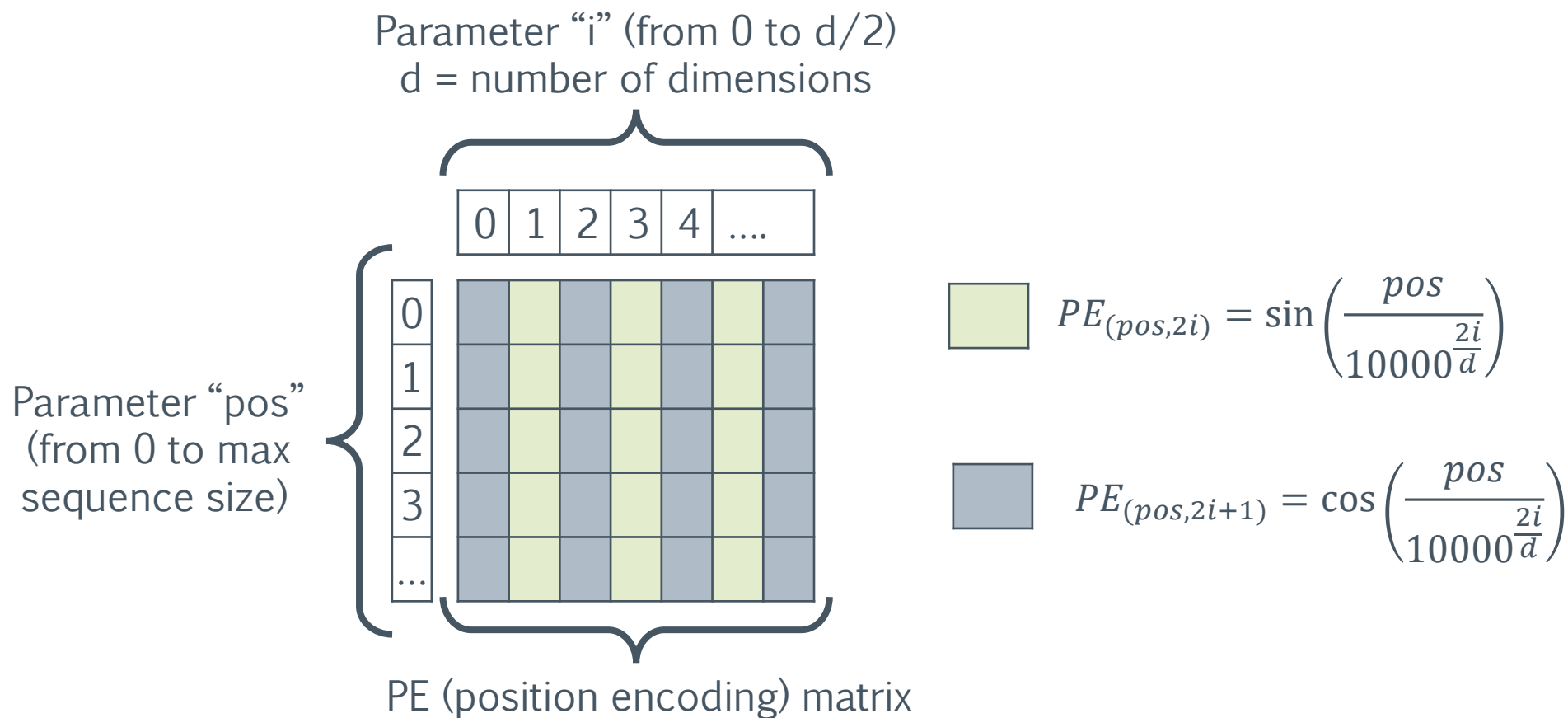
3. Use *sin and cos functions combined*. This solution was discussed in the “*Attention is all you need paper*” and implies using sin and cos in the following way:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

π

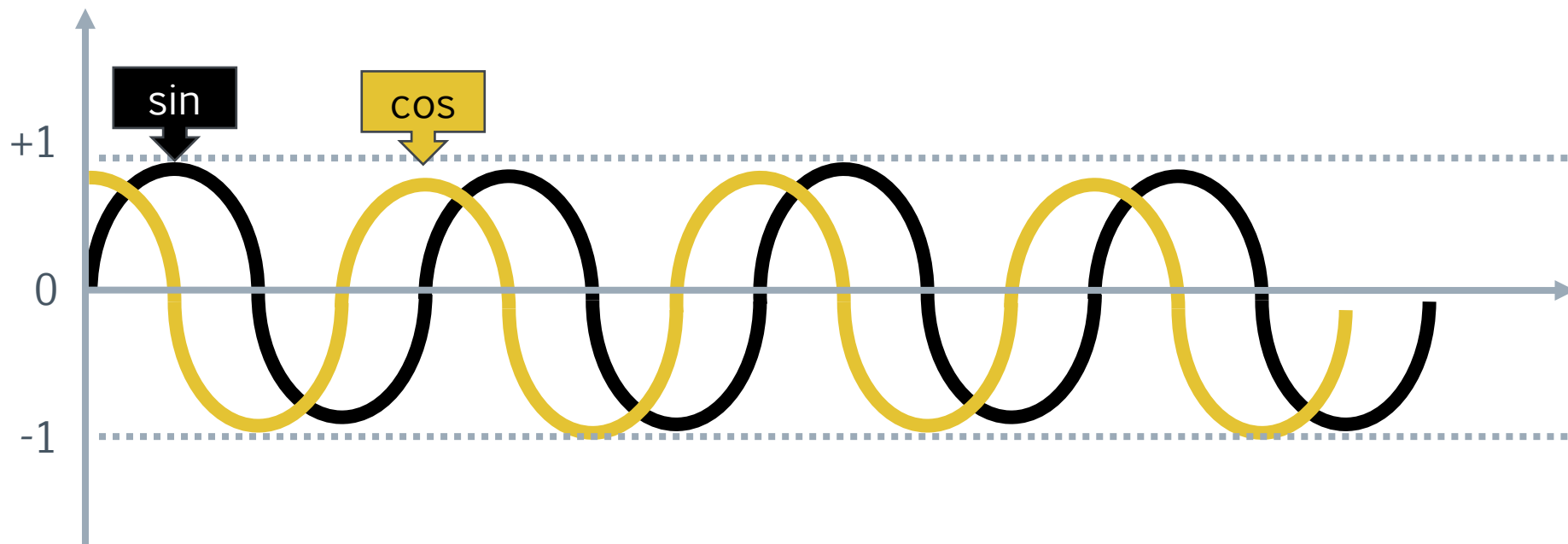
Positional encoding



π

Positional encoding

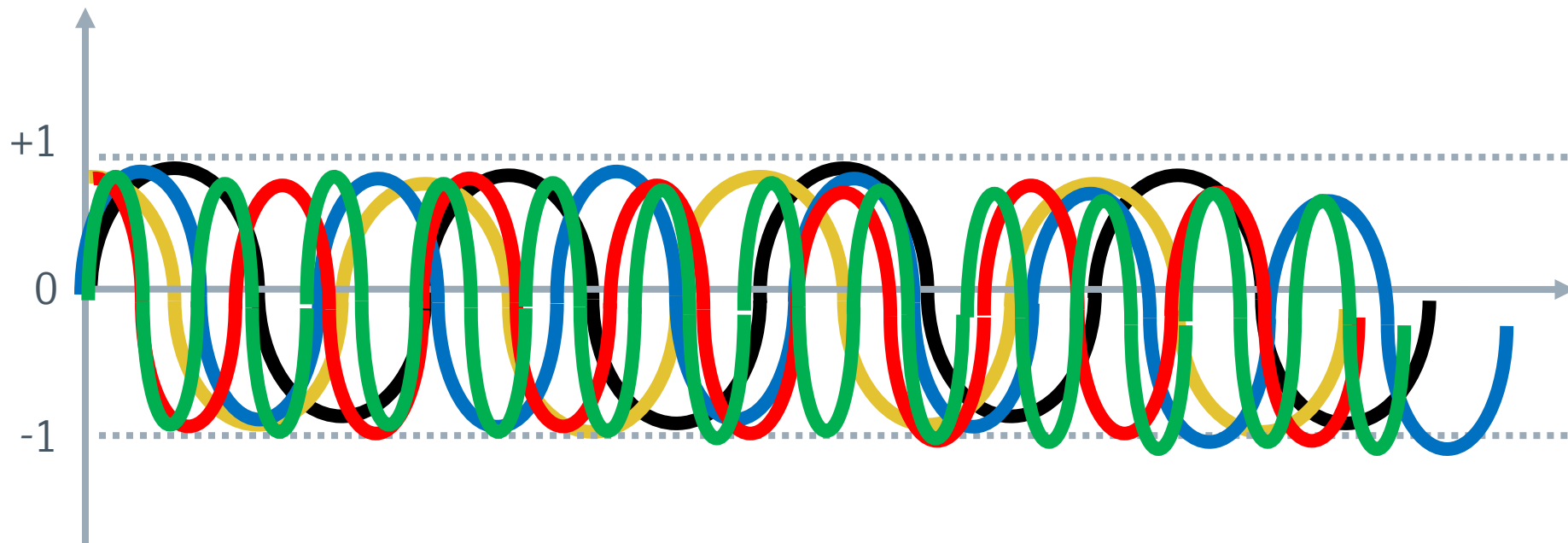
- › But why is this mode so effective ?
 - **cos** and **sin** complement each other (increase variance in dimensions)



π

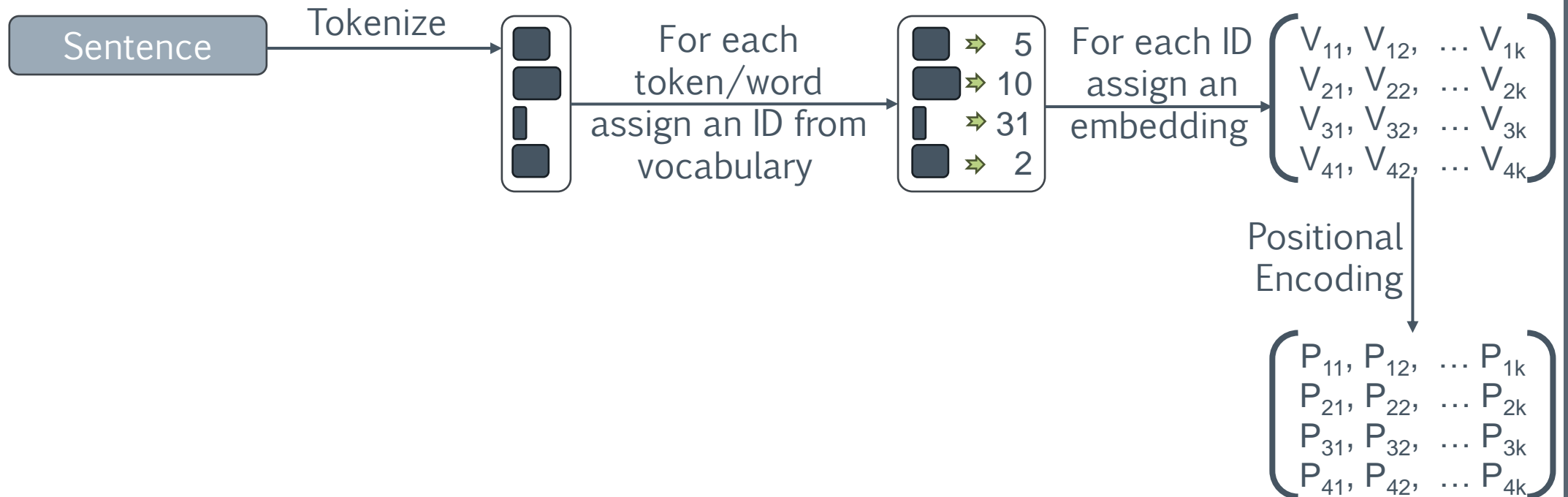
Positional encoding

- › But why is this mode so effective ?
 - **cos** and **sin** complement each other (increase variance in dimensions)
 - the more we increase the “l” parameter, the amplitude of waves decreases, and more space is occupied (better variance)



Positional encoding

› So ... the process up to this point looks like this:



Transformers

Attention mechanism

Attention

- › So ... what is attention ? It's a mechanism that tells us at what word we need to pay attention to (given an existing state). One way of doing this is to build a matrix where we compute some scores that tell us how much attention we need to pay for a specific word.

	I	like	oranges	and	apples
I	-	80%	5%	10%	5%
like	5%	-	45%	5%	45%
oranges	5%	10%	-	80%	10%
and	20%	10%	35%	-	35%
apples	5%	10%	10%	75%	-

This translates that we should pay more attention to word “and” when processing word “apples”

Attention

- › So ... how can we get from embeddings to this type relations ?
- › First let's define some vectors:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \dots & \dots & \dots & \dots & \dots \\ a_{k1} & a_{k2} & a_{k3} & \dots & a_{kn} \end{pmatrix} \quad Q = \begin{pmatrix} proj_1(n \rightarrow m) \\ proj_2(n \rightarrow m) \\ proj_3(n \rightarrow m) \\ \dots \\ proj_n(n \rightarrow m) \end{pmatrix} \quad V = \begin{pmatrix} proj_1(n \rightarrow m) \\ proj_2(n \rightarrow m) \\ proj_3(n \rightarrow m) \\ \dots \\ proj_n(n \rightarrow m) \end{pmatrix} \quad K = \begin{pmatrix} proj_1(n \rightarrow m) \\ proj_2(n \rightarrow m) \\ proj_3(n \rightarrow m) \\ \dots \\ proj_n(n \rightarrow m) \end{pmatrix}$$

A = the output from the positional encoding step, with “k” vectors (k = number of words), each vector of size “n” (the dimension of the embeddings)

3 vectors of projection matrixes from “n” to “m”, where “m” is a divisor of “n” (we will discuss about this later).
 $m \leq n$

Attention

- › We define the similarity measure as follows:

$$\text{Sim}(A) = \begin{pmatrix} S_{11} & S_{12} & S_{13} & \dots & S_{1k} \\ S_{21} & S_{22} & S_{23} & \dots & S_{2k} \\ S_{31} & S_{32} & S_{33} & \dots & S_{3k} \\ \dots & \dots & \dots & \dots & \dots \\ S_{k1} & S_{k2} & S_{k3} & \dots & S_{kk} \end{pmatrix}$$

Sim(S) size = $k \times k$
(notice that we no longer
have “n” dimensions)

The *scale factor* is required to normalize the output. It is more a practical reason (keep all elements within a certain numerical range).

$$\text{where } S_{(i,j)} = (Q_i \cdot K_j) \times \text{scale factor}$$

This Dot product computes the similarity
between the project of i -th word and j -th word

Attention

- › The next step is to normalize the values from $\text{Sim}(A)$ via a softmax function

$$\text{Sim}(A) = \begin{pmatrix} S_{11} & S_{12} & S_{13} & \dots & S_{1k} \\ S_{21} & S_{22} & S_{23} & \dots & S_{2k} \\ S_{31} & S_{32} & S_{33} & \dots & S_{3k} \\ \dots & \dots & \dots & \dots & \dots \\ S_{k1} & S_{k2} & S_{k3} & \dots & S_{kk} \end{pmatrix} \rightarrow \text{NormSim}(A) = \begin{pmatrix} \text{Softmax}(S_{11} & S_{12} & S_{13} & \dots & S_{1k}) \\ \text{Softmax}(S_{21} & S_{22} & S_{23} & \dots & S_{2k}) \\ \text{Softmax}(S_{31} & S_{32} & S_{33} & \dots & S_{3k}) \\ \dots & \dots & \dots & \dots & \dots \\ \text{Softmax}(S_{k1} & S_{k2} & S_{k3} & \dots & S_{kk}) \end{pmatrix}$$

Using softmax translates the values into similarity percentages

Attention

› Next we multiply the output to the V:

$$\text{Sim}(A) = \begin{pmatrix} S_{11} & S_{12} & S_{13} & \dots & S_{1k} \\ S_{21} & S_{22} & S_{23} & \dots & S_{2k} \\ S_{31} & S_{32} & S_{33} & \dots & S_{3k} \\ \dots & \dots & \dots & \dots & \dots \\ S_{k1} & S_{k2} & S_{k3} & \dots & S_{kk} \end{pmatrix} \rightarrow \begin{pmatrix} \text{Softmax}(S_{11} \ S_{12} \ S_{13} \ \dots \ S_{1k}) \\ \text{Softmax}(S_{21} \ S_{22} \ S_{23} \ \dots \ S_{2k}) \\ \text{Softmax}(S_{31} \ S_{32} \ S_{33} \ \dots \ S_{3k}) \\ \dots \\ \text{Softmax}(S_{k1} \ S_{k2} \ S_{k3} \ \dots \ S_{kk}) \end{pmatrix} \times V = \begin{bmatrix} o_{(1,1)} & \dots & o_{(1,m)} \\ \vdots & \ddots & \vdots \\ o_{(k,1)} & \dots & o_{(k,m)} \end{bmatrix}$$

Finally, we multiply the normalized softmaxed similarity values with V. This computes the probability that a specific word relates to another one.

Attention

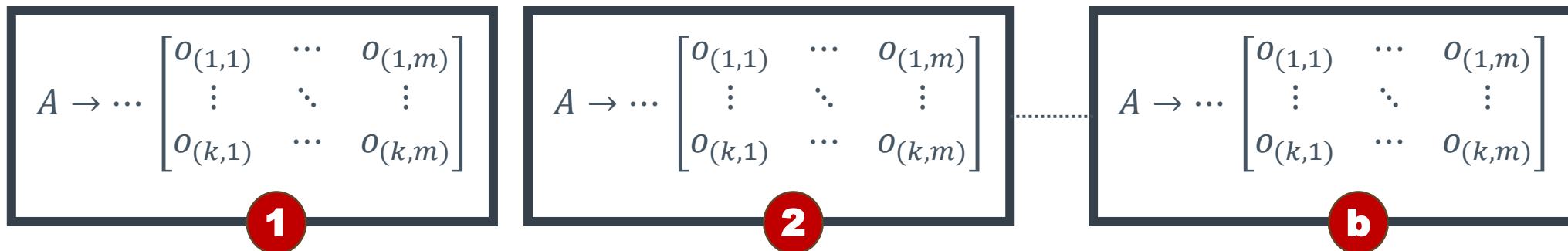
- › This entire process where we go from the input A and reach the final value is called **attention head**

$$A \rightarrow Sim(A) \rightarrow SoftMax(Sim(A)) \times V = \begin{bmatrix} O_{(1,1)} & \cdots & O_{(1,m)} \\ \vdots & \ddots & \vdots \\ O_{(k,1)} & \cdots & O_{(k,m)} \end{bmatrix}$$

A very simplified form of attention head. The most important thing is to notice that the final output is a matrix of size $(k \times m)$

Attention

- › In practice, several attention heads are use (each one focusing on a different relationship between words).

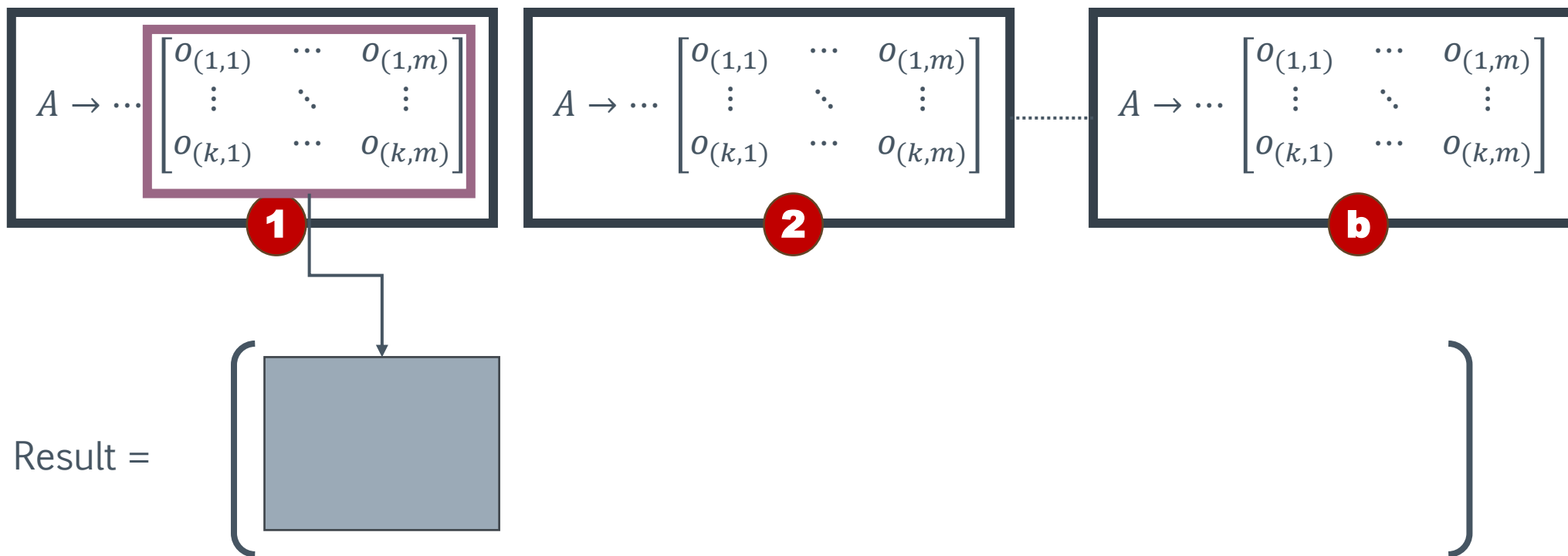


Where “ $b * m = n$ ”
In practice “b” is a hyperparameter, and “m” is computed based on its value. For example if $n=128$ and $b=4$ then m will be $128/4 = 32$

π

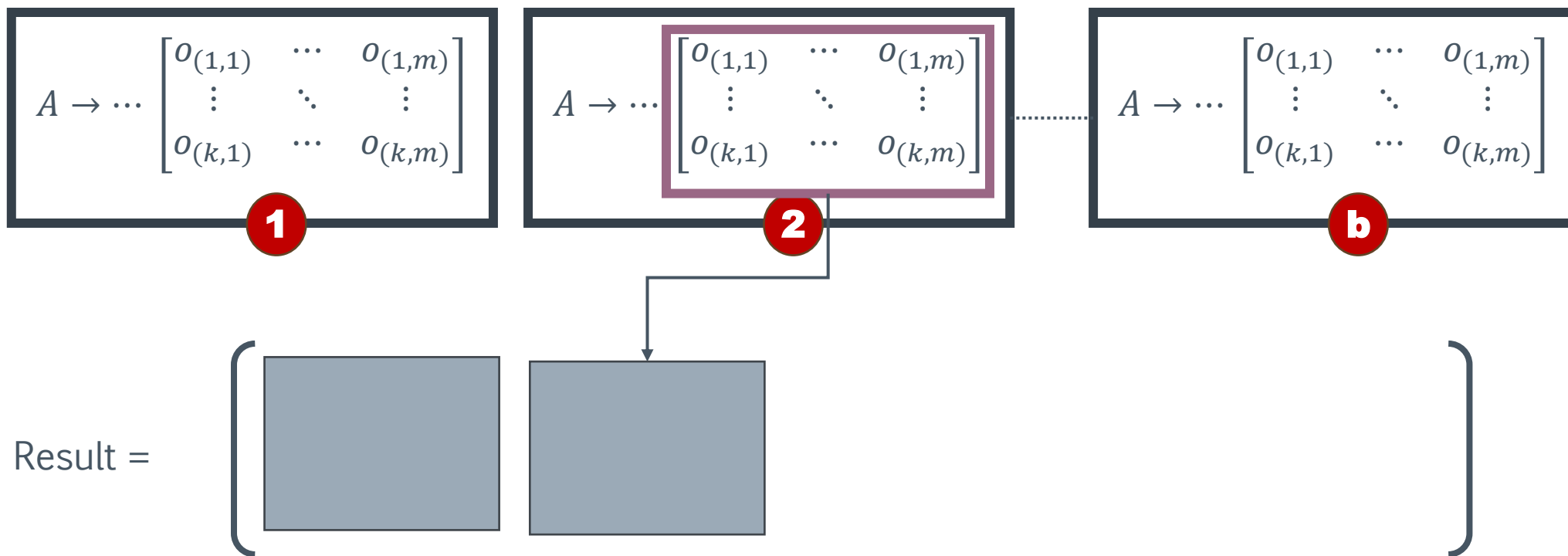
Attention

- › For the final output of the attention layer, we are going to concatenate all of the results from attention heads.



Attention

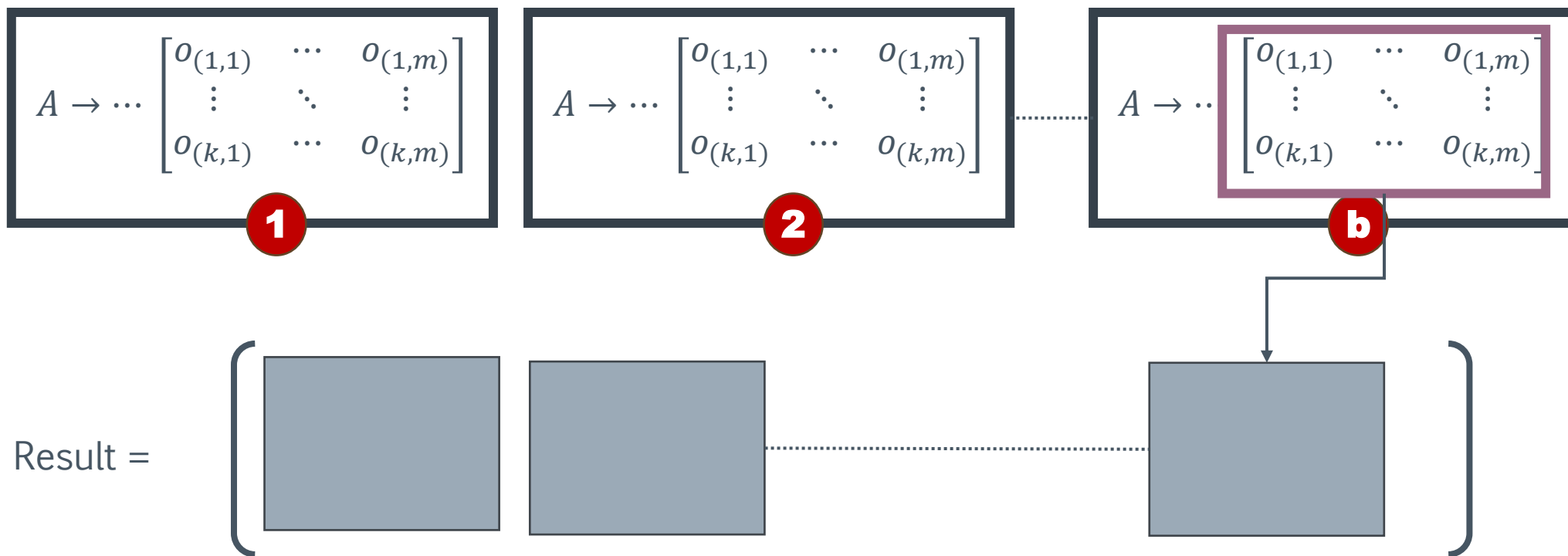
- › For the final output of the attention layer, we are going to concatenate all of the results from attention heads.



π

Attention

- › For the final output of the attention layer, we are going to concatenate all of the results from attention heads.



Attention

- › One final observation → notice that after the final step, the result is no longer $(k \times m)$, now it is $(k \times n)$, as $m \times b = n$

Transformers

Feed Forward network

π

Feed Forward network

- › Now that we have the output from the attention layer, what are the next steps.
- › At step we feed this output to a feedforward neural network:

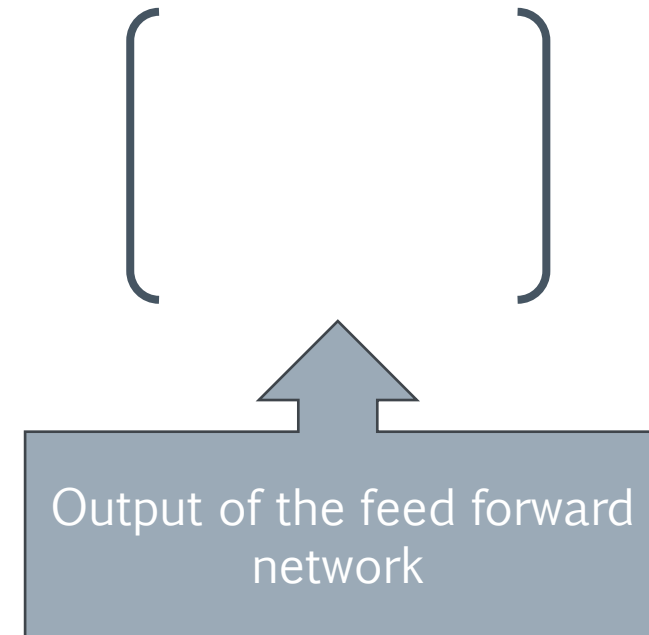
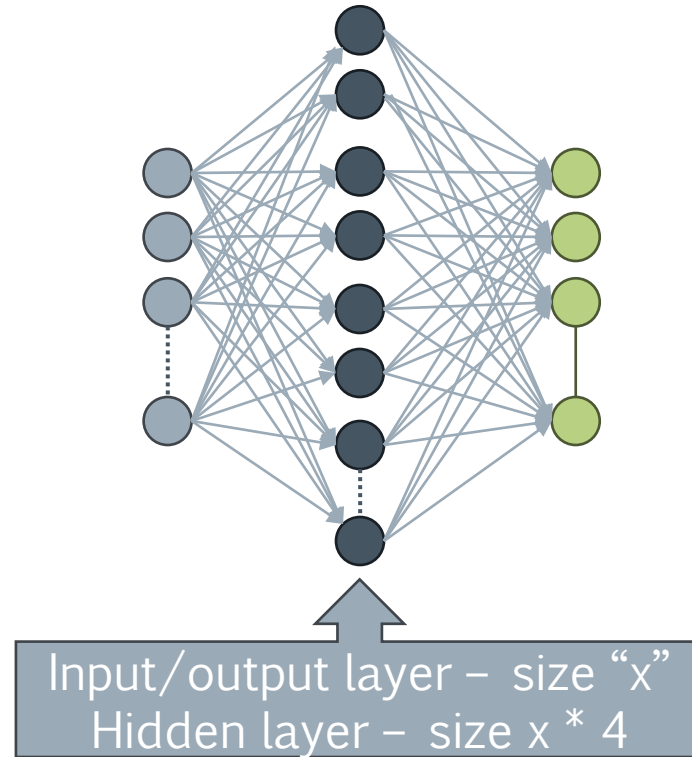
$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \dots & \dots & \dots & \dots & \dots \\ a_{k1} & a_{k2} & a_{k3} & \dots & a_{kn} \end{pmatrix}$$

Output from the attention layer.
“n” embeddings size, k = number
of words/byte pairs

Feed Forward network

- › Now that we have the output from the attention layer, what are the next steps.
- › At step we feed this output to a feedforward neural network:

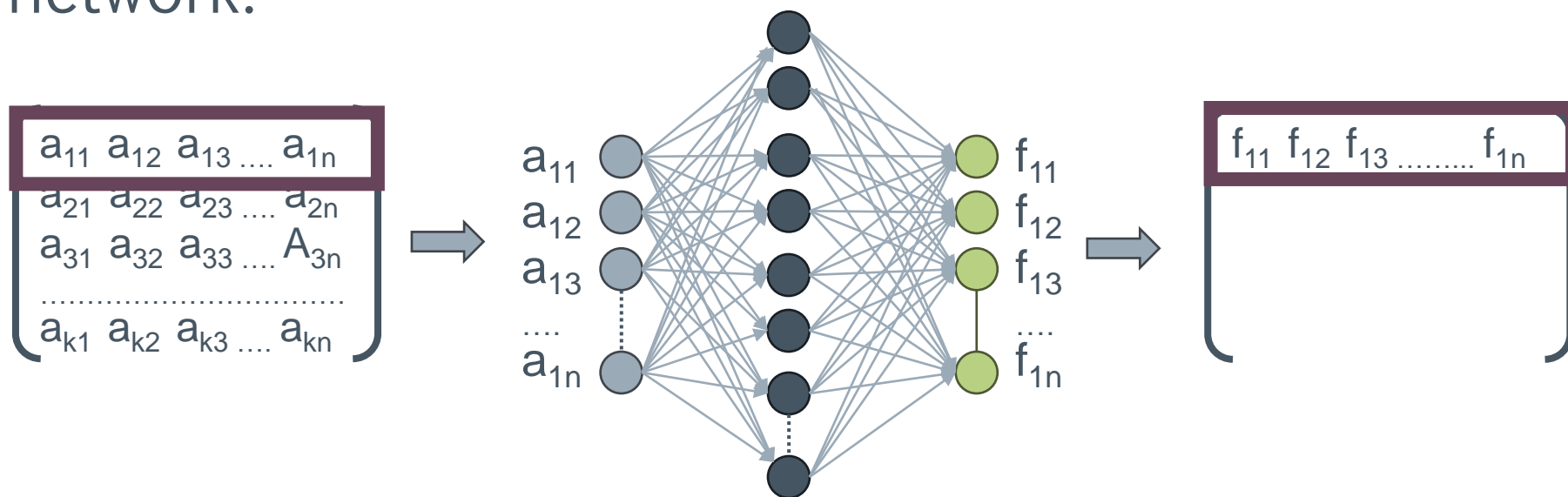
$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \dots & \dots & \dots & \dots & \dots \\ a_{k1} & a_{k2} & a_{k3} & \dots & a_{kn} \end{pmatrix}$$



π

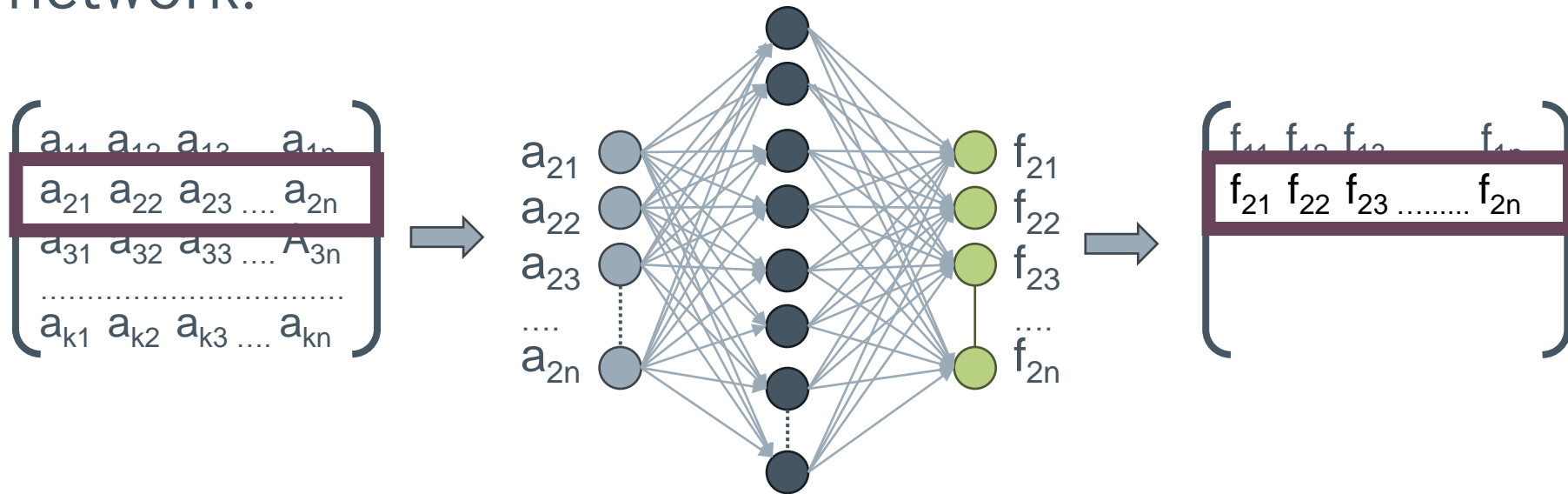
Feed Forward network

- › Now that we have the output from the attention layer, what are the next steps.
- › At step we feed this output to a feedforward neural network:



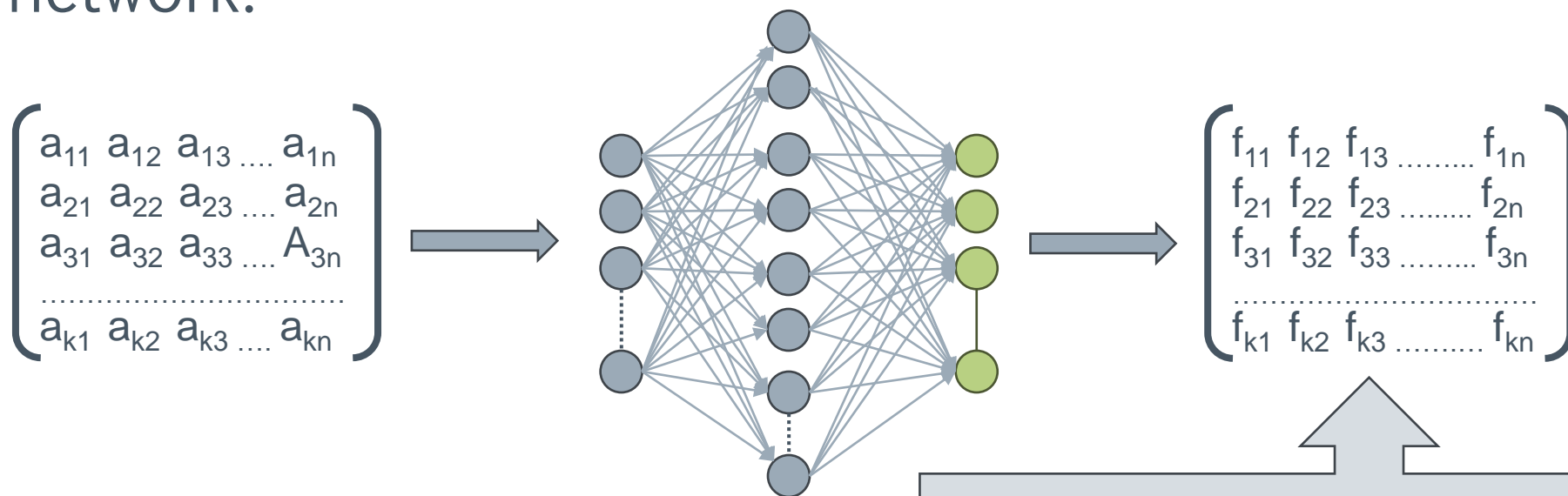
Feed Forward network

- › Now that we have the output from the attention layer, what are the next steps.
- › At step we feed this output to a feedforward neural network:



Feed Forward network

- › Now that we have the output from the attention layer, what are the next steps.
- › At step we feed this output to a feedforward neural network:

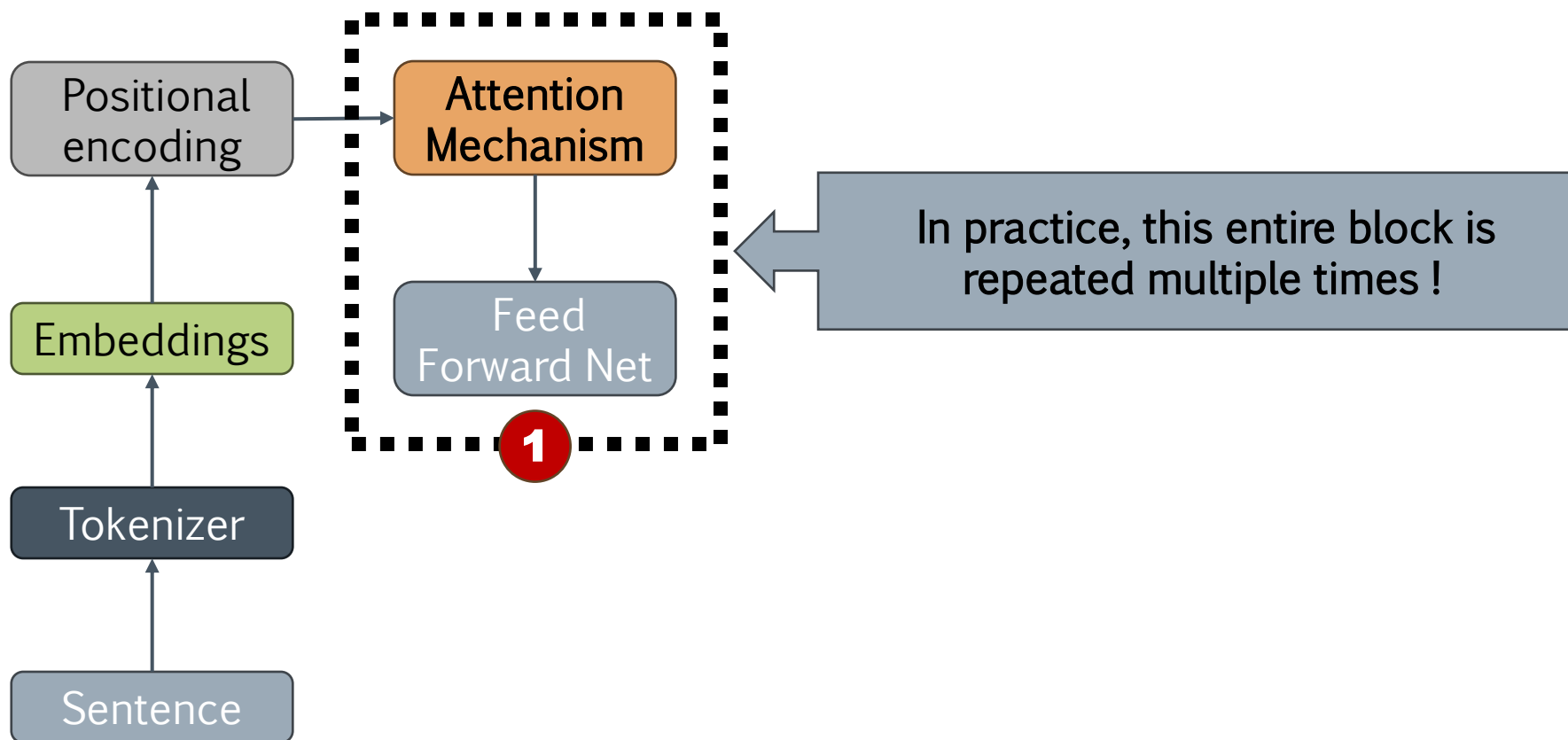


It should be mentioned that at this point, since the weights are frozen, all of computation can be done in **parallel**.

π

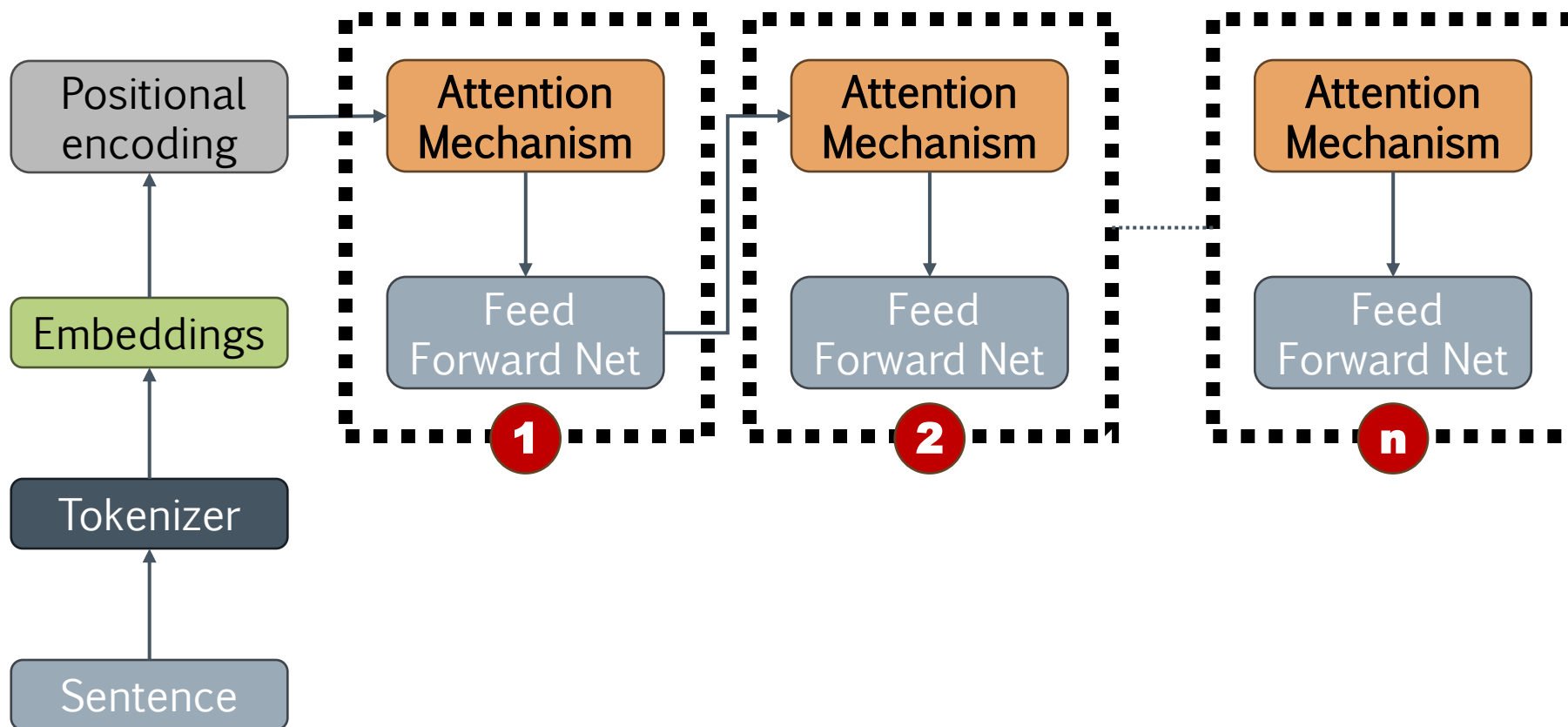
Feed Forward network

› So ... what is the architecture up to this point ?



Feed Forward network

› So ... what is the architecture up to this point ?



Transformers

Final layer (prediction)

π

Final layer (prediction)

- › Now that we have the output from the previous layers, how can we predict the word to come ?

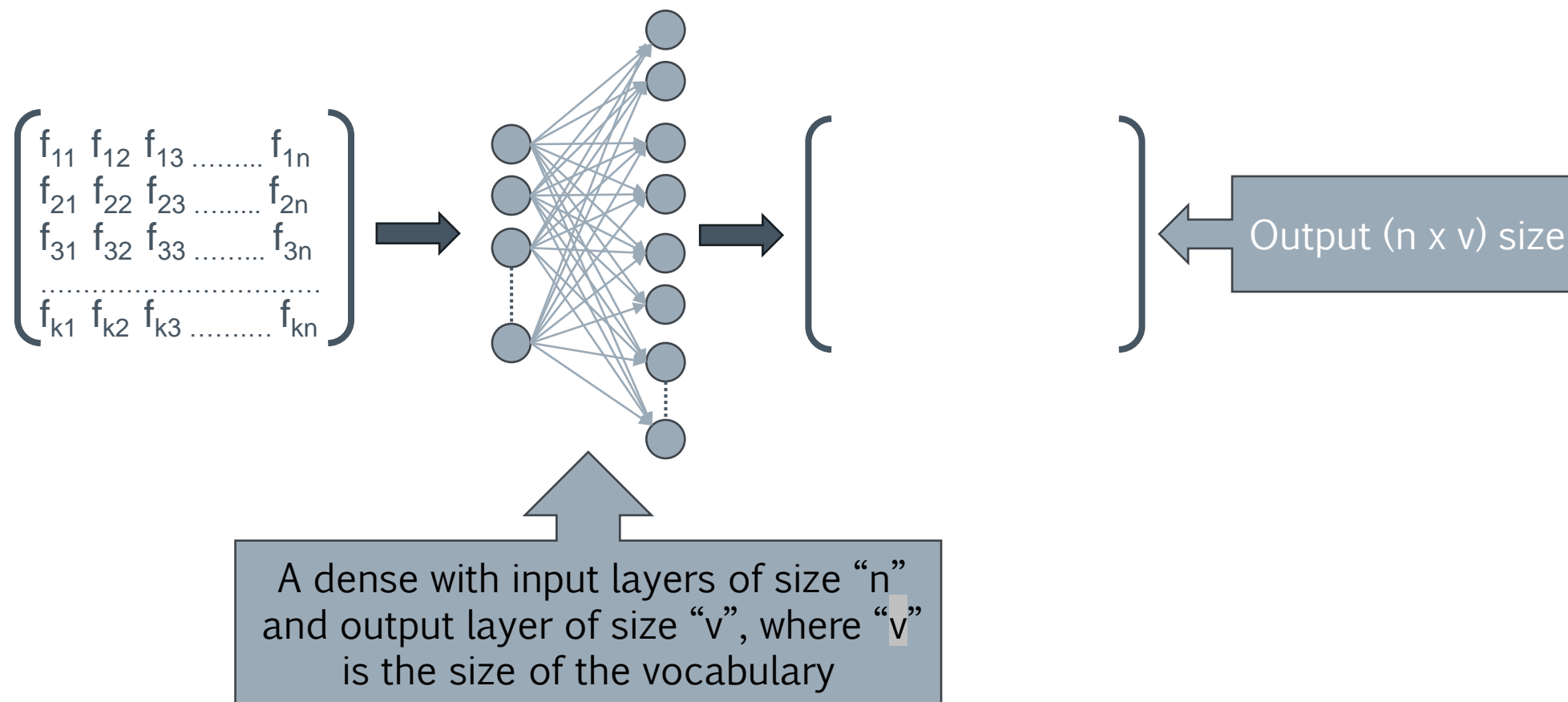
$$\begin{pmatrix} f_{11} & f_{12} & f_{13} & \dots & f_{1n} \\ f_{21} & f_{22} & f_{23} & \dots & f_{2n} \\ f_{31} & f_{32} & f_{33} & \dots & f_{3n} \\ \dots & \dots & \dots & \dots & \dots \\ f_{k1} & f_{k2} & f_{k3} & \dots & f_{kn} \end{pmatrix}$$

Output from the previous layer.
“n” embeddings size, k = number
of words/byte pairs

π

Final layer (prediction)

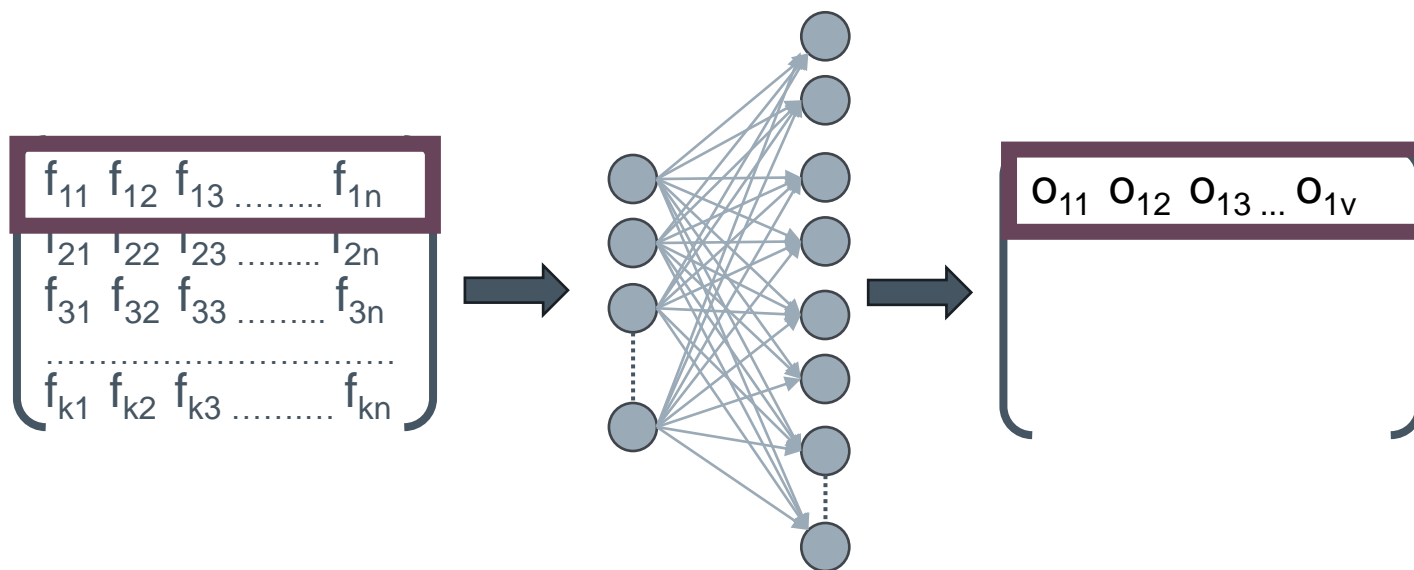
- › Now that we have the output from the previous layers, how can we predict the word to come ?



π

Final layer (prediction)

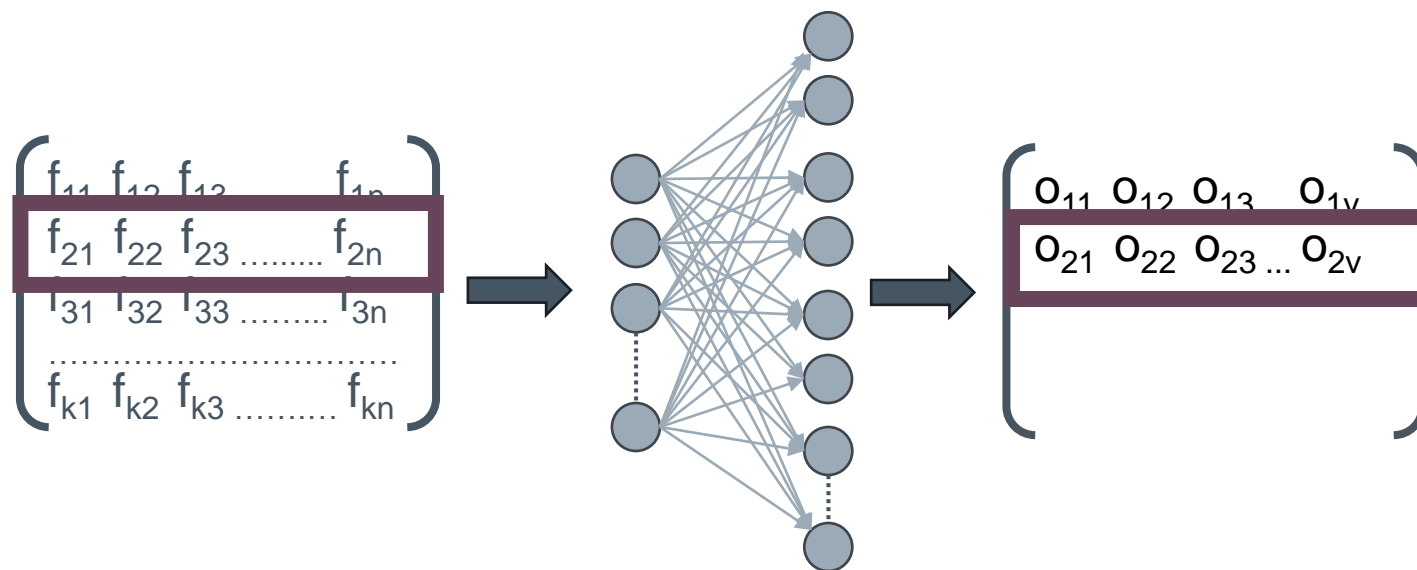
- › Now that we have the output from the previous layers, how can we predict the word to come ?



π

Final layer (prediction)

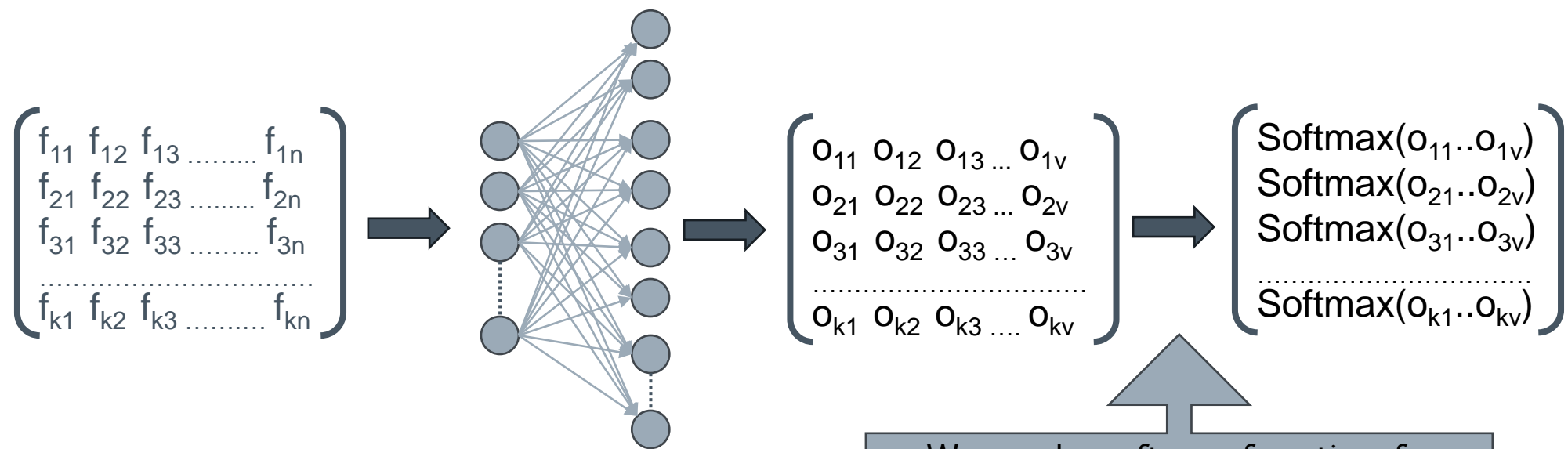
- › Now that we have the output from the previous layers, how can we predict the word to come ?



π

Final layer (prediction)

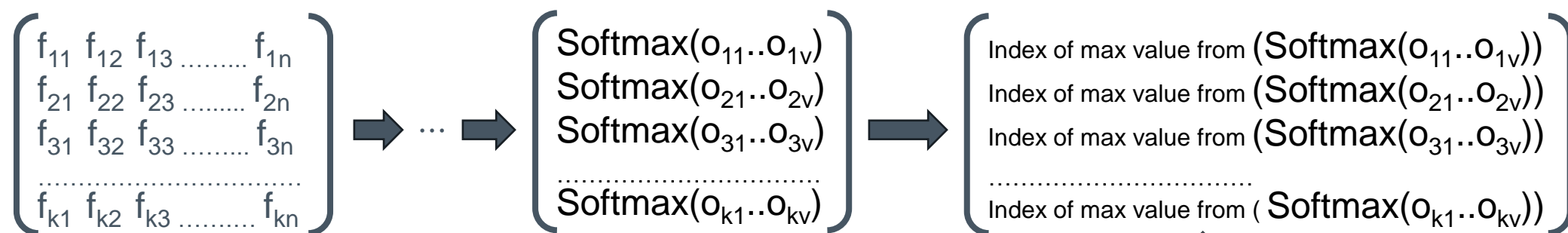
- › Now that we have the output from the previous layers, how can we predict the word to come ?



We apply softmax function for every line (vector) in the matrix. The result is that we normalize the matrix into probabilities.

Final layer (prediction)

- › Now that we have the output from the previous layers, how can we predict the word to come ?



For every element on every row, we select the index of the largest values. Since the index is between 0 and v, we actually select a token from the vocabulary !

π

Final layer (prediction)

- › Let's see an example (vocabulary with size 7, embeddings with dimension 2:

Word	ID
I	1
like	2
apples	3
oranges	4
both	5
you	6
and	7

Vocabulary
(7 words)

I like oranges

$\begin{pmatrix} 5 & 9 \\ 6 & 7 \\ 1 & 8 \end{pmatrix}$

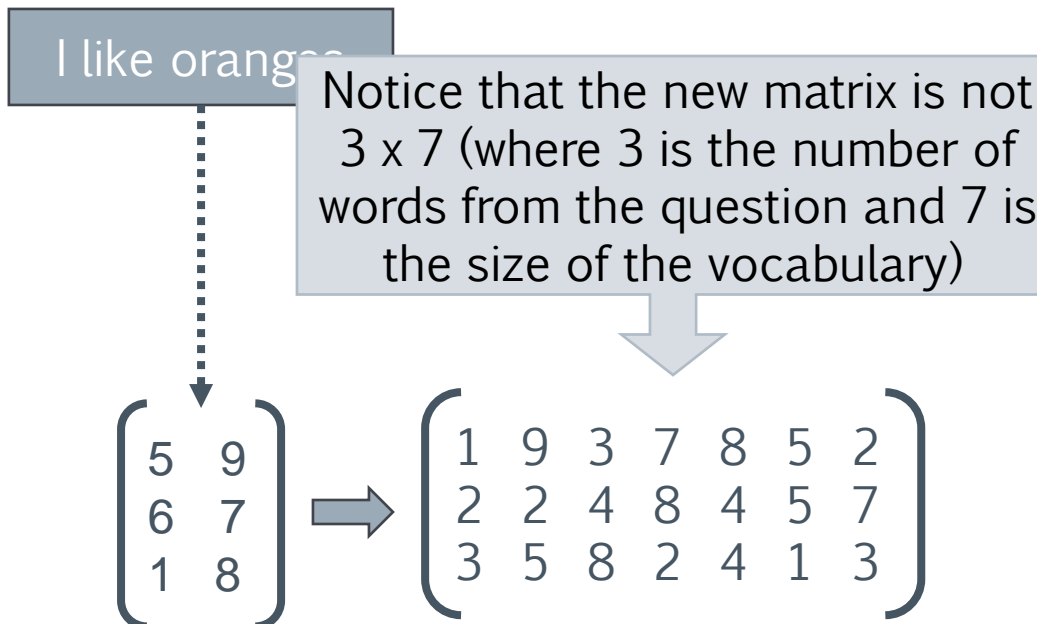
All of the steps until the final one are executed:
embeddings, positional encoding, attention, feed forward

π

Final layer (prediction)

- › Let's see an example (vocabulary with size 7, embeddings with dimension 2):

Word	ID
I	1
like	2
apples	3
oranges	4
both	5
you	6
and	7



Final layer (prediction)

- › Let's see an example (vocabulary with size 7, embeddings with dimension 2):

Word	ID
I	1
like	2
apples	3
oranges	4
both	5
you	6
and	7

I like oranges

$\begin{pmatrix} 5 & 9 \\ 6 & 7 \\ 1 & 8 \end{pmatrix}$



$\begin{pmatrix} 1 & 9 & 3 & 7 & 8 & 5 & 2 \\ 2 & 2 & 4 & 8 & 4 & 5 & 7 \\ 3 & 5 & 8 & 2 & 4 & 1 & 3 \end{pmatrix}$

$\begin{pmatrix} 2\% & 25\% & 8\% & 20\% & 22\% & 14\% & 5\% \\ 6\% & 6\% & 12\% & 25\% & 12\% & 15\% & 21\% \\ 11\% & 19\% & 30\% & 7\% & 15\% & 3\% & 11\% \end{pmatrix}$

We compute the softmax function for every row in the matrix. For example, for the first row: [1,9,3,7,8,5,2] the result will be: [2%,25%,8%,20%,22%,14%,5%]

Final layer (prediction)

- › Let's see an example (vocabulary with size 7, embeddings with dimension 2):

Word	ID
I	1
like	2
apples	3
oranges	4
both	5
you	6
and	7

I like oranges

Now we create a vector with the indexes of the highest values on each row. For example, for the 1st row the highest value is 25% that corresponds to index 2. As a result we will add 2 on the first position of the vector.

$\begin{pmatrix} 5 & 9 \\ 6 & 7 \\ 1 & 8 \end{pmatrix}$



$\begin{pmatrix} 1 & 9 & 3 & 7 & 8 & 5 & 2 \\ 2 & 2 & 4 & 8 & 4 & 5 & 7 \\ 3 & 5 & 8 & 2 & 4 & 1 & 3 \end{pmatrix}$



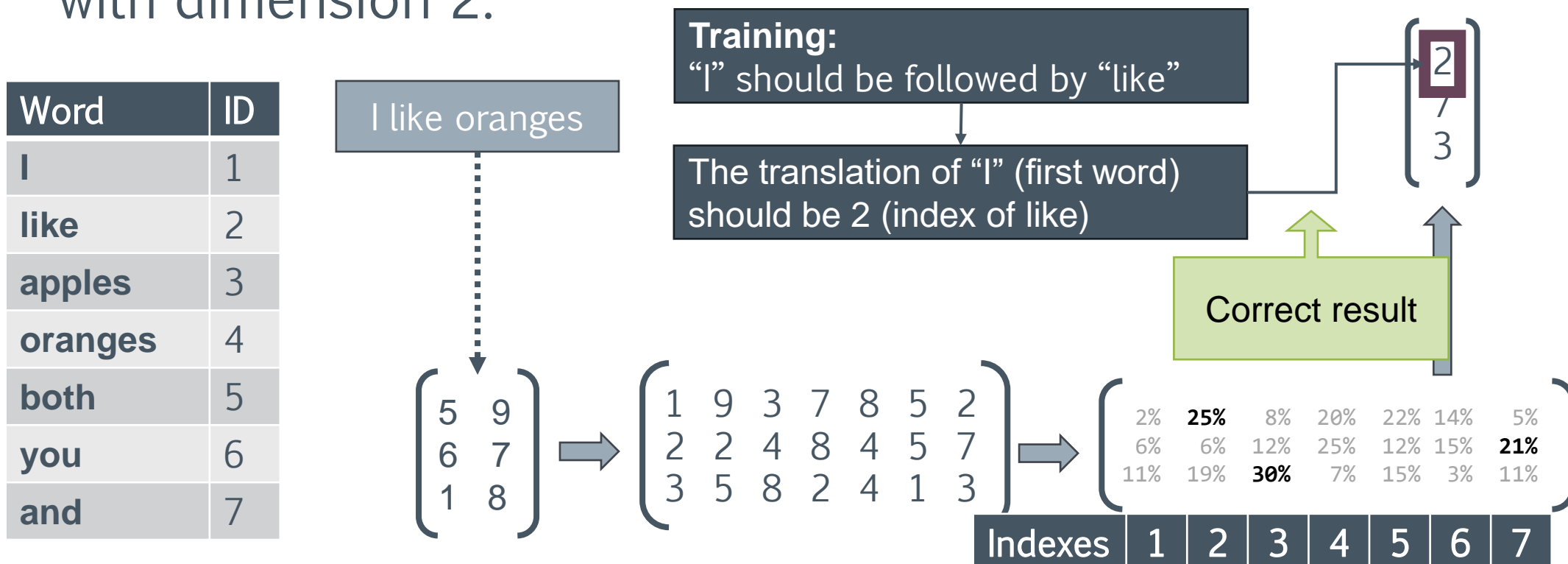
$\begin{pmatrix} 2\% & \mathbf{25\%} & 8\% & 20\% & 22\% & 14\% & 5\% \\ 6\% & 6\% & 12\% & 25\% & 12\% & 15\% & \mathbf{21\%} \\ 11\% & 19\% & \mathbf{30\%} & 7\% & 15\% & 3\% & 11\% \end{pmatrix}$

Indexes 1 2 3 4 5 6 7

$\begin{pmatrix} 2 \\ 7 \\ 3 \end{pmatrix}$

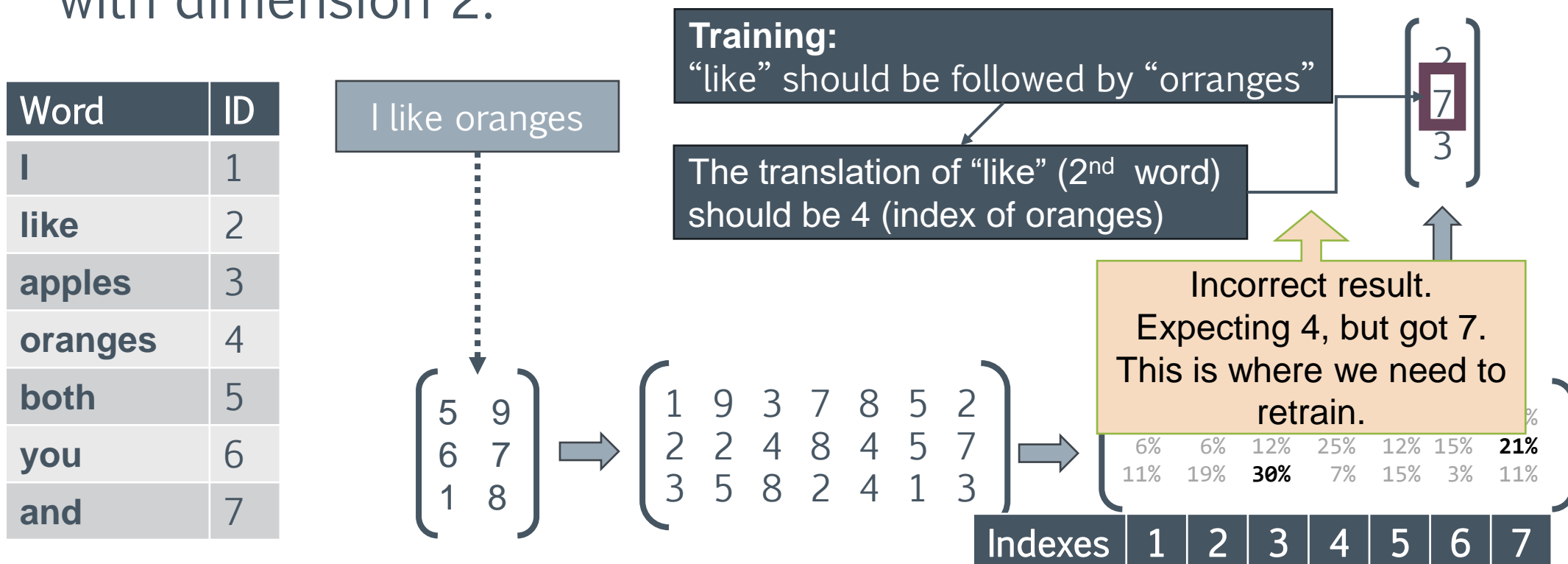
Final layer (prediction)

- › Let's see an example (vocabulary with size 7, embeddings with dimension 2):



Final layer (prediction)

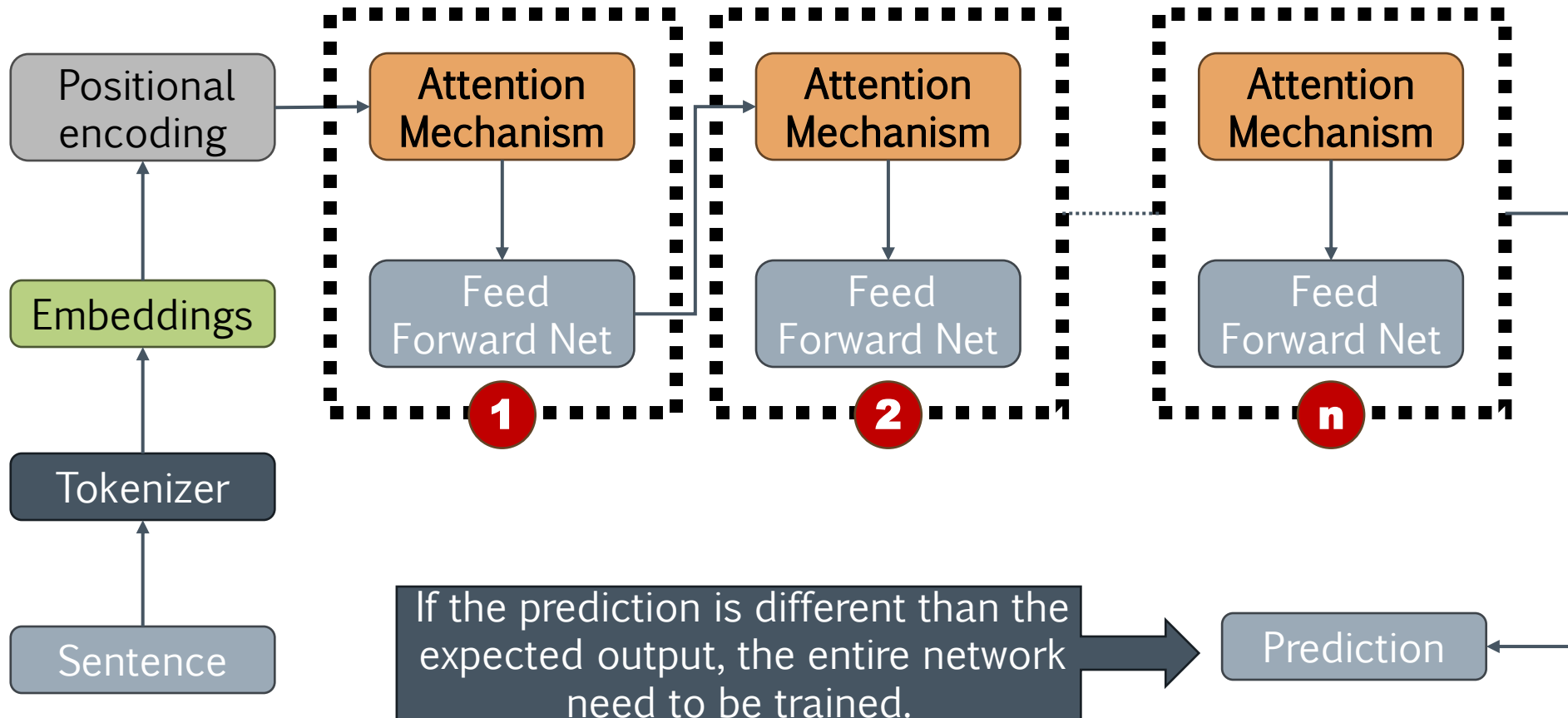
- › Let's see an example (vocabulary with size 7, embeddings with dimension 2):



π

Architecture

› So ... what is the complete architecture



π

Q & A

