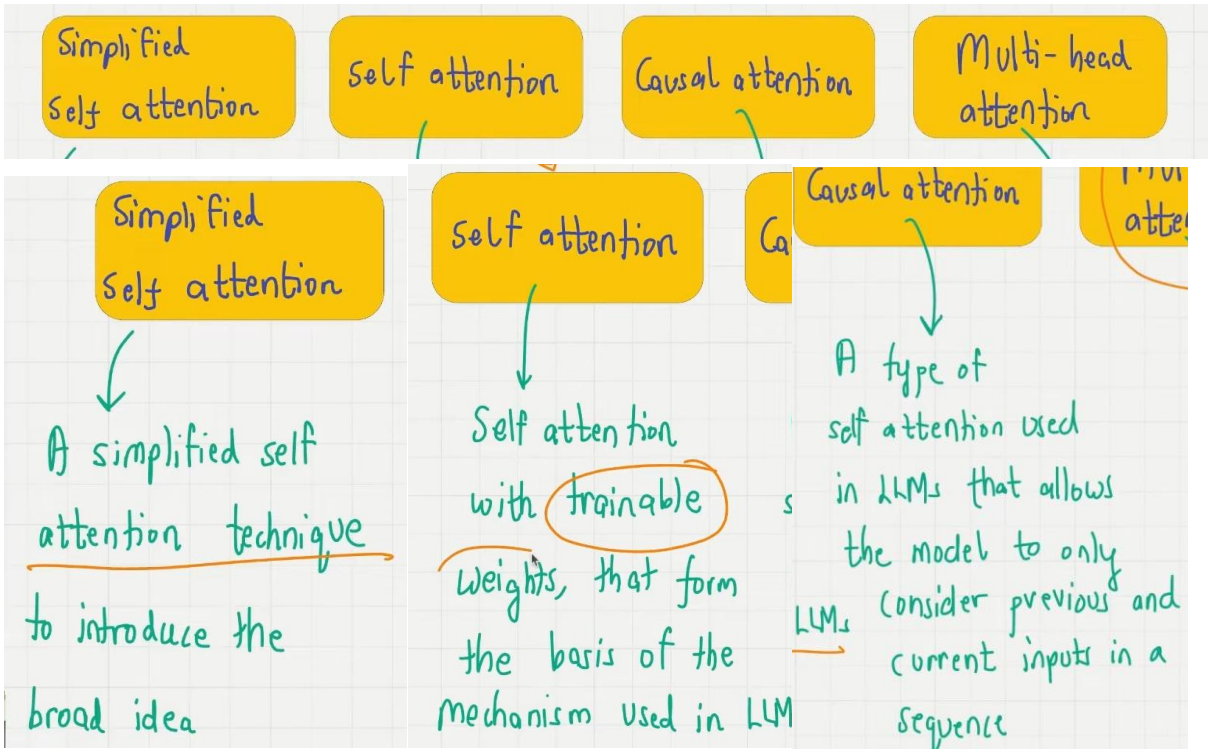
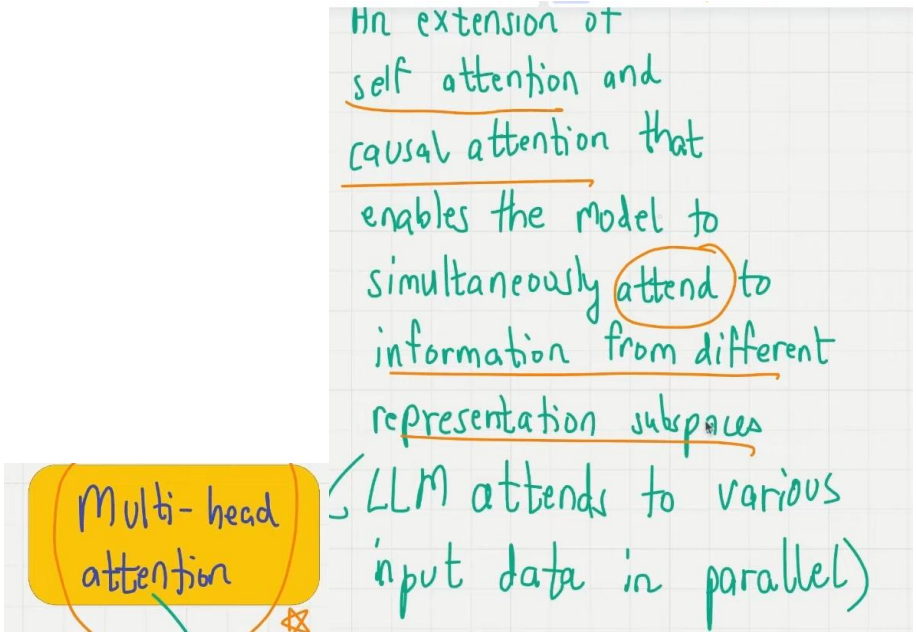
"The cat that was sitting on the mat, which was next to the dog, jumped"

① The 4 types of attention mechanisms:





Simplified self attention

A simplified self attention technique to introduce the broad idea

Self attention

Self attention with trainable weights, that form the basis of the mechanism used in LLMs

Causal attention

A type of self attention used in LLMs that allows the model to only consider previous and current inputs in a sequence

Multi-head attention

An extension of self attention and causal attention that enables the model to simultaneously attend to information from different representation subspaces

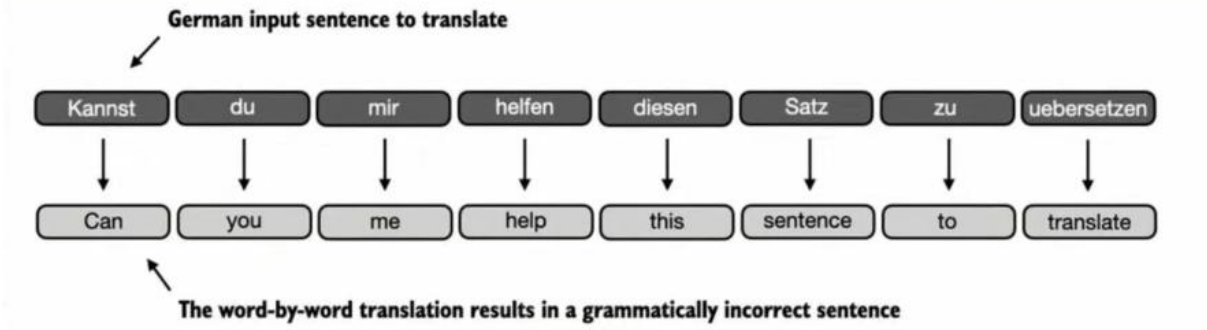
(LLM attends to ...

8%

(2) The problem with modeling long sequences:

(a) What is the problem with architectures without the attention mechanism which came before LLMs?

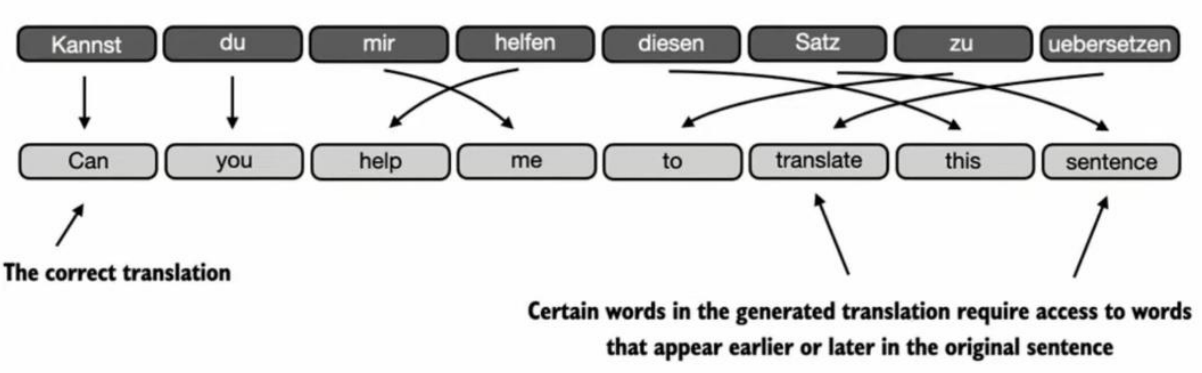
* let's consider a language translation model *



Word by word translation does not work!

Can You Help me!

क्या तुम मेरी मदद करोगे?



The translation process requires (contextual) understanding and grammar alignment.

(b) To address this issue that we cannot translate text word by word, it is common to use a neural network with two submodules:

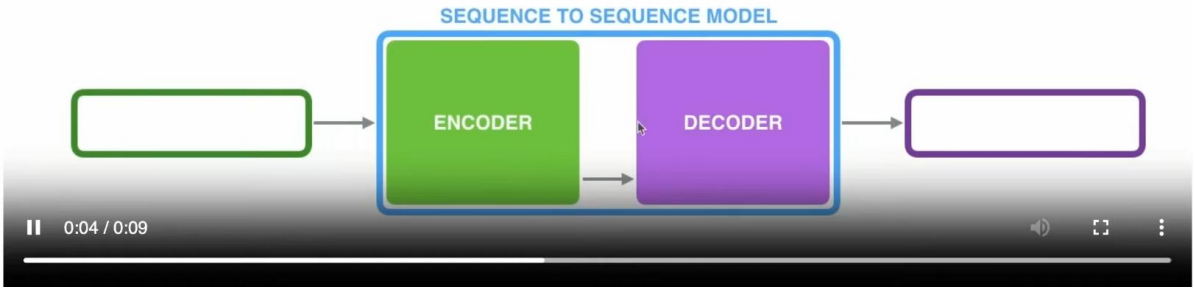
Encoder Decoder

jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

Looking under the hood

Under the hood, the model is composed of an encoder and a decoder.

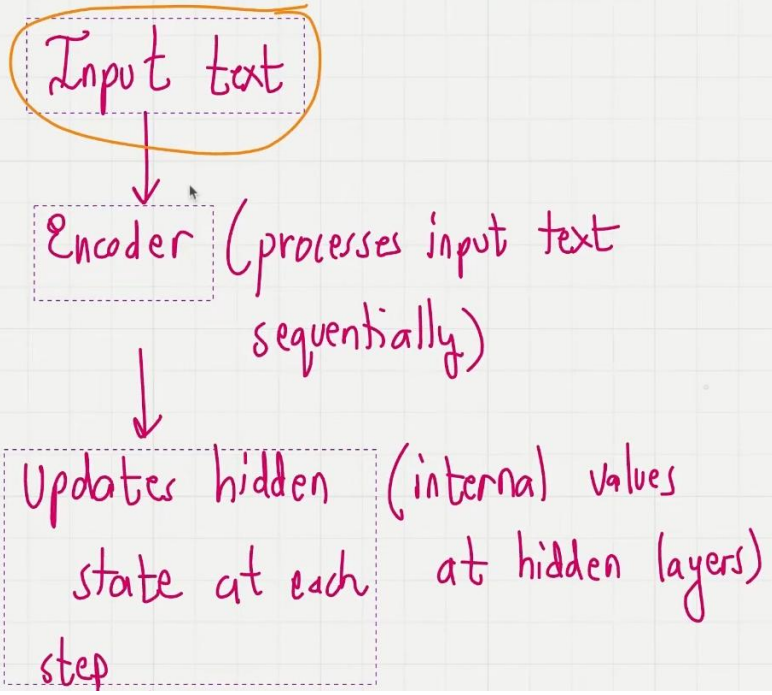
The encoder processes each item in the input sequence, it compiles the information it captures into a vector (called the context). After processing the entire input sequence, the encoder sends the context over to the decoder, which begins producing the output sequence item by item.



(c) Before transformers, Recurrent Neural Networks (RNNs) were the most popular encoder-decoder architecture for language translation.

(d) RNN: Output from previous step is fed as input to current text

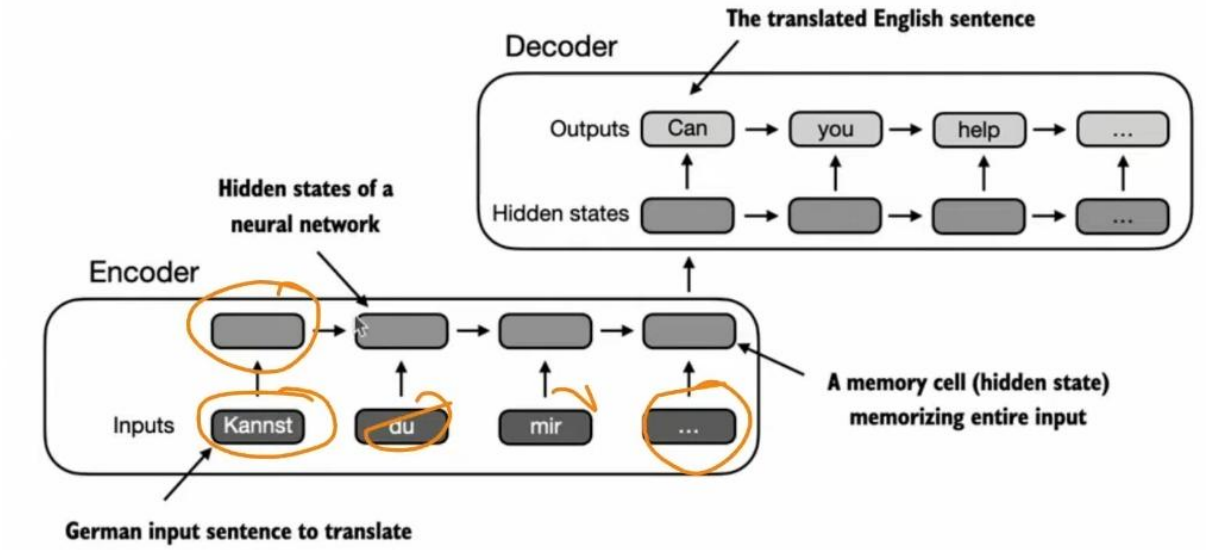
(e) Here's how the encoder-decoder RNN works:



Final hidden state (encoder tries to capture sentence meaning)

Decoder uses this final hidden state to generate translated sentence (one word at a time)

(decoder also updates it's hidden state at each time)



(f) The encoder processes the entire input text into a hidden state (memory cell). Decoder takes this hidden state to produce an

↳ think of this as an embedding vector.

Decoder takes this hidden state to produce an output.

(g) * Big issue: RNN can't directly access earlier hidden states from the encoder, during the decoding phase.

It relies solely on current hidden state.

This leads to a loss of context, especially in complex sentences where dependencies might span long distances.

Encoder compresses entire input sequence into a single hidden state vector

If the sentence is very long, it becomes very difficult for the RNN to capture all information in a single vector

"The cat that was sitting on the mat, which was next to the dog, jumped"

↓

"Le chat qui était assis sur le tapis, qui était à côté du chien, a sauté"

Here, the key action "jumped" depends

Here, the key action "jumped" depends

on the subject "cat" but also on understanding the longer dependencies ("that was sitting on the mat, next to the dog")

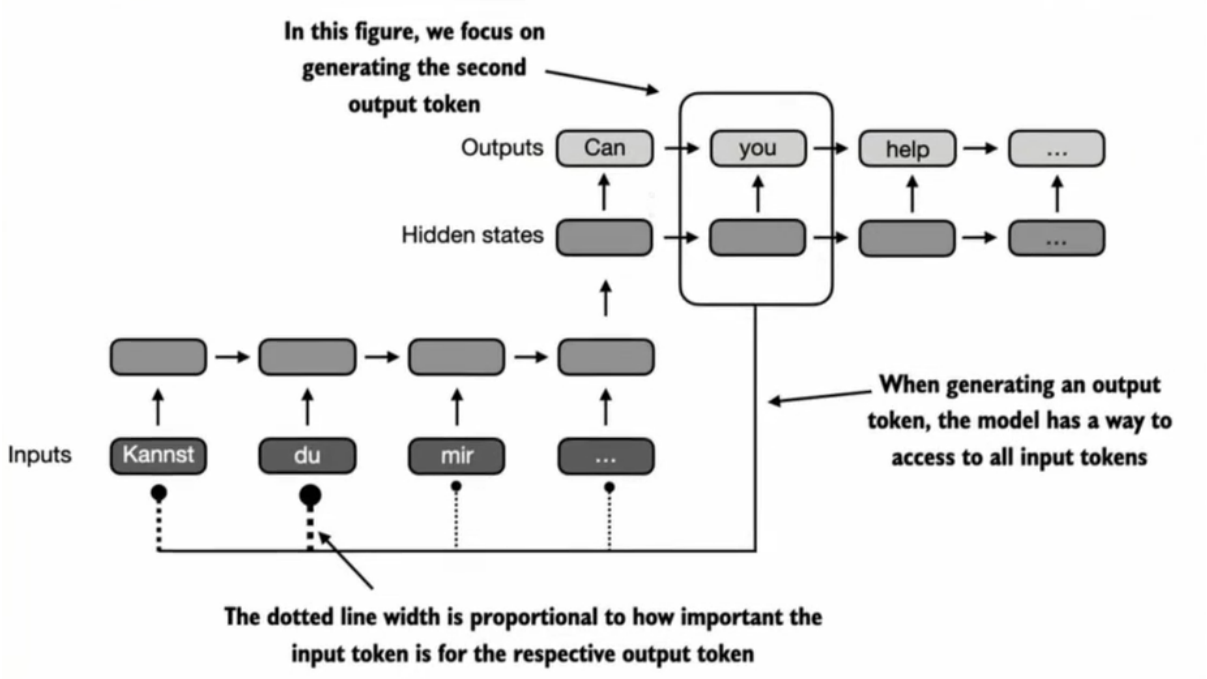
The RNN decoder might struggle with this

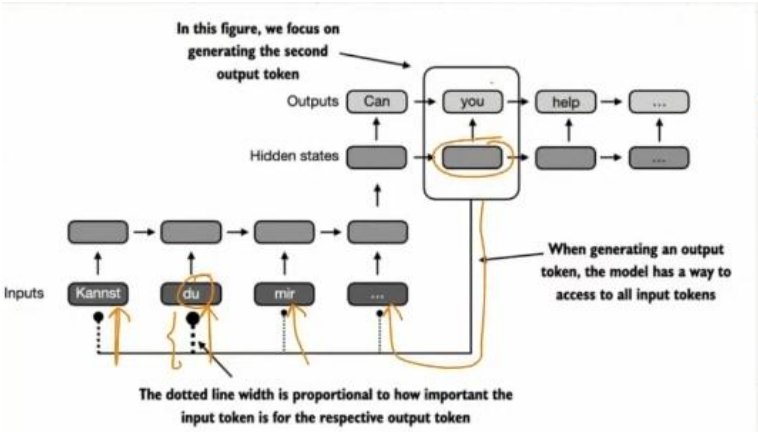
③ Capturing data dependencies with attention mechanisms

(a) RNNs work fine for translating short sentences, but don't work for long texts as they don't have direct access to previous words in the input.

(b) One major shortcoming in this approach is that: RNN must remember the entire encoded input in a single hidden state before passing it to the decoder

(c) In 2014, researchers developed the so called "Bahdanau attention mechanism for RNNs" modifies the encoder-decoder RNN such that the decoder can selectively access different parts of the input sequence at each decoding step.





General idea behind the Bahdanau mechanism.

- Using an attention mechanism, the text generating decoder part of the network can access all input tokens selectively.

- This means that some input tokens are more important than others for generating a given output token.
- This importance is determined by the so called attention weights.

(d) Only 3 years later, researchers found that RNN architectures are not required for building deep neural networks for natural language processing and proposed the original transformer architecture; with a self attention mechanism inspired by the Bahdanau attention mechanism.

"The cat that was sitting on the mat, which was next to the dog, jumped"

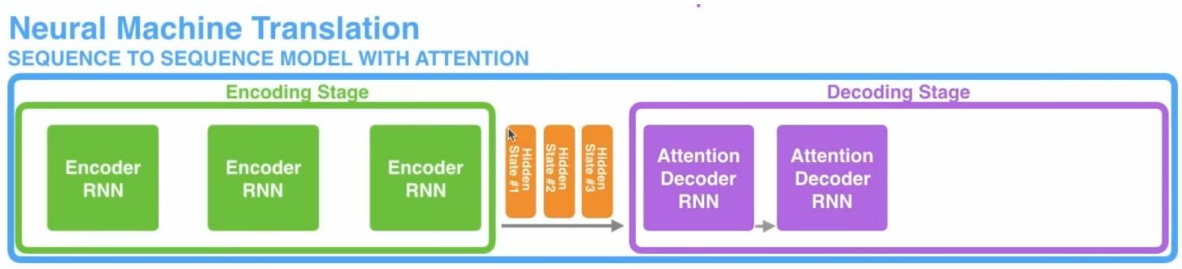
↓

"Le chat qui était assis sur le tapis, qui était à côté du chien, a sauté"

At each decoding step, the model can look back at the entire input sequence and decide which parts are most relevant to generate current word

When decoder is predicting "saute," the attention mechanism allows it to focus on part of input that corresponds to "jumped".

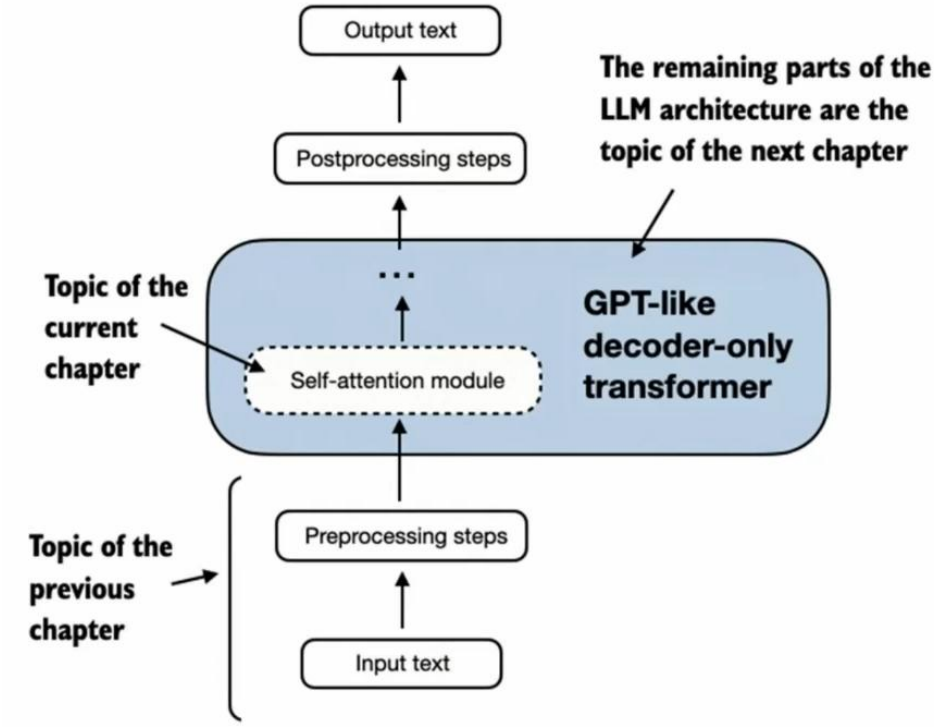
Dynamic focus on different parts of input sequence allows models to learn long range dependencies more effectively.



RNNs	LSTMs	Attention	Transformers
1980	1997	2014	2017

(e) Self attention is a mechanism that allows each position of the input sequence to attend to all positions in the same sequence when computing the representation of a sequence.

(f) Self attention is a key component of contemporary LLMs based on the transformer architecture, such as the GPT series.



④ Attending to different parts of the input with self attention

(a) In "self attention", the "self" refers to the mechanisms' ability to compute attention weights by relating different positions in a single input sequence.

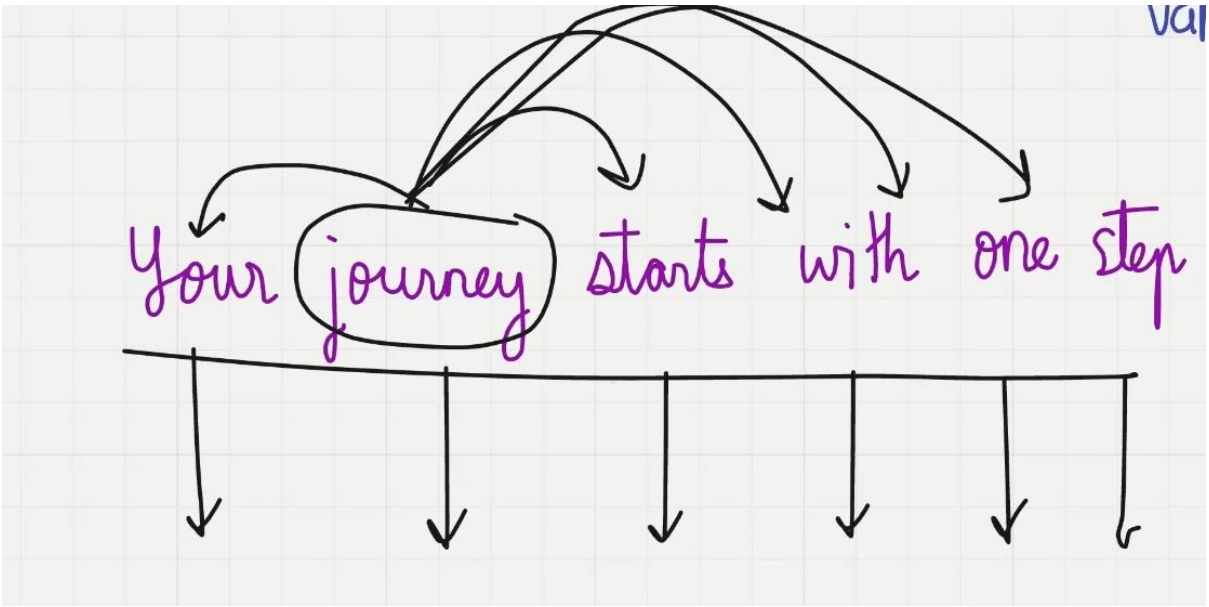
b) It learns the relationship between various parts of the input itself, such as words in a sentence.

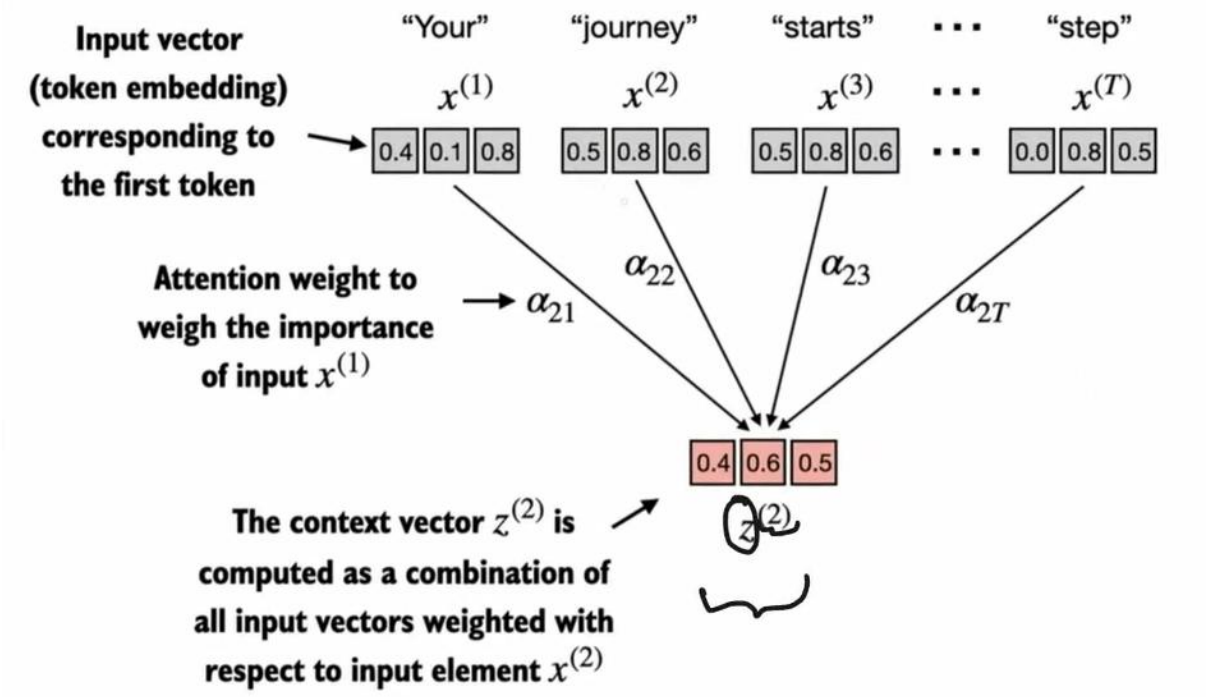
c) This is in contrast to traditional attention mechanisms, where the focus is on relationships between elements of 2 different sequences.

A simple self attention mechanism without trainable weights

In this section, we will implement a simple variant of self-attention, free from any trainable weights.

Your journey starts with one step

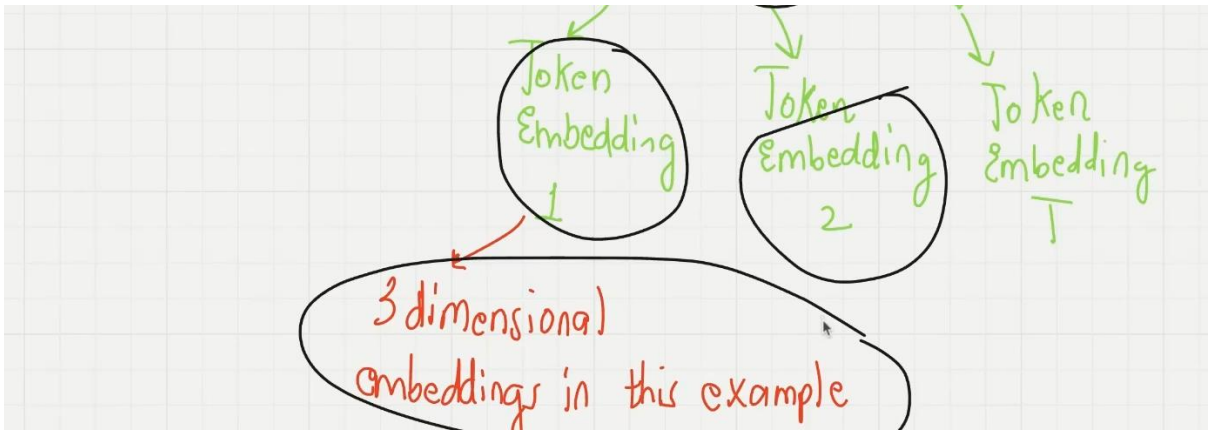




① Input Sequence: x

consists of T elements:

$x^{(1)}$, $x^{(2)}$ ..... $x^{(T)}$



② Goal of self attention:

"Calculate a context vector $z^{(i)}$ for each element $x^{(i)}$"

Context vector: Enriched embedding vector

Let us focus on 2nd element ($x^{(2)}$)

↳ Corresponding context vector is $z^{(2)}$.

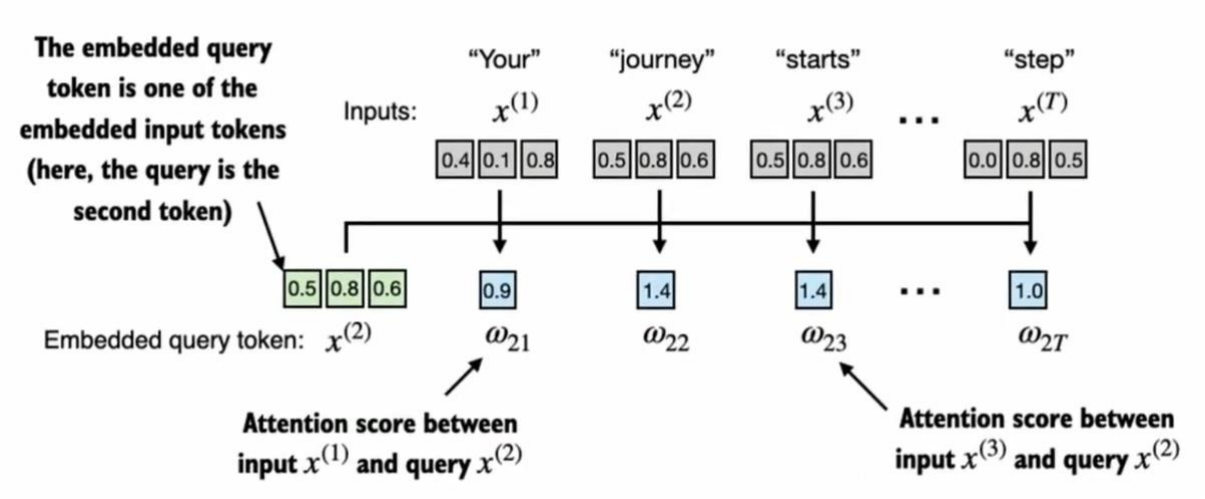
$z^{(2)}$ is an embedding which contains information about $x^{(2)}$ and all other input elements $x^{(1)}$ to $x^{(T)}$

2nd element ($x^{(2)}$)

context vector is $z^{(2)}$. → query

Later, we will add trainable weights that help the LLM learn to construct these context vectors; so that they are relevant for the LLM to generate the next token.

First step of implementing self-attention is to compute the intermediate values w, also referred to as attention scores.



6. Intermediate attention scores calculated between the query token and each input taken

Query: Dot products quantifies how much two vectors aligned

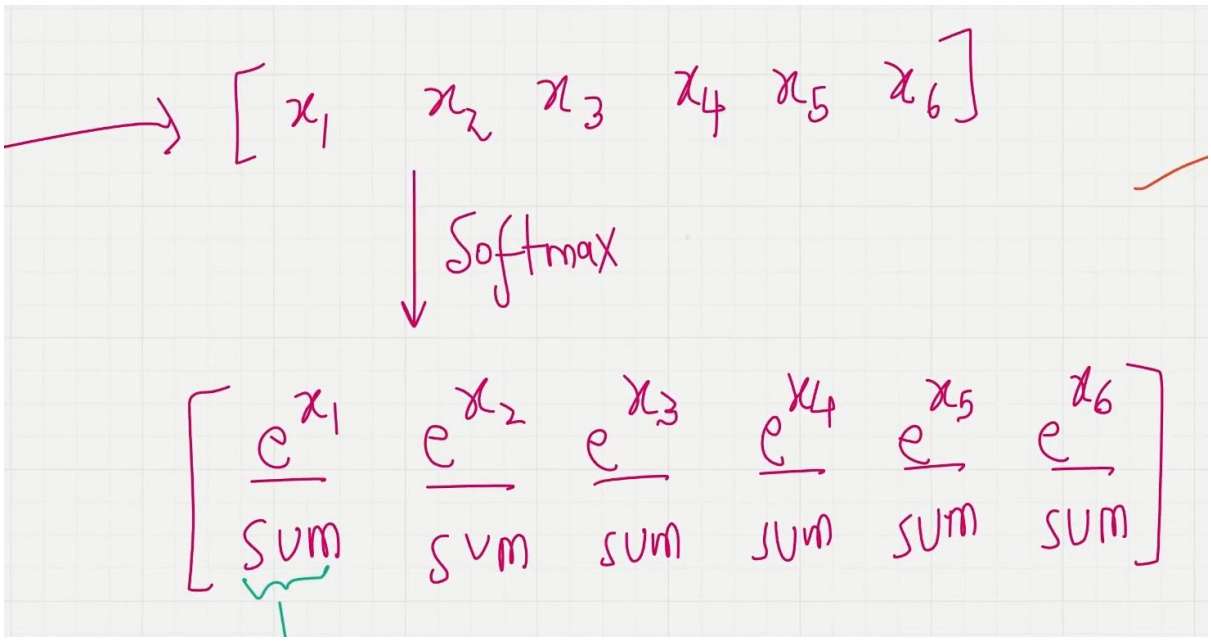
In the context of self attention mechanisms, dot product determines the extent to which elements of a sequence attend to one another → higher the dot product, high the similarity and attention scores betwe two elements.

In the next step, we normalize each of the attention scores that we computed previously.

In practice, it's more common and advisable to use the softmax function for normalization.

This approach is better at managing extreme values and offers more favorable gradient properti

Below is a basic implementation of the softmax function for normalizing the attention scores:



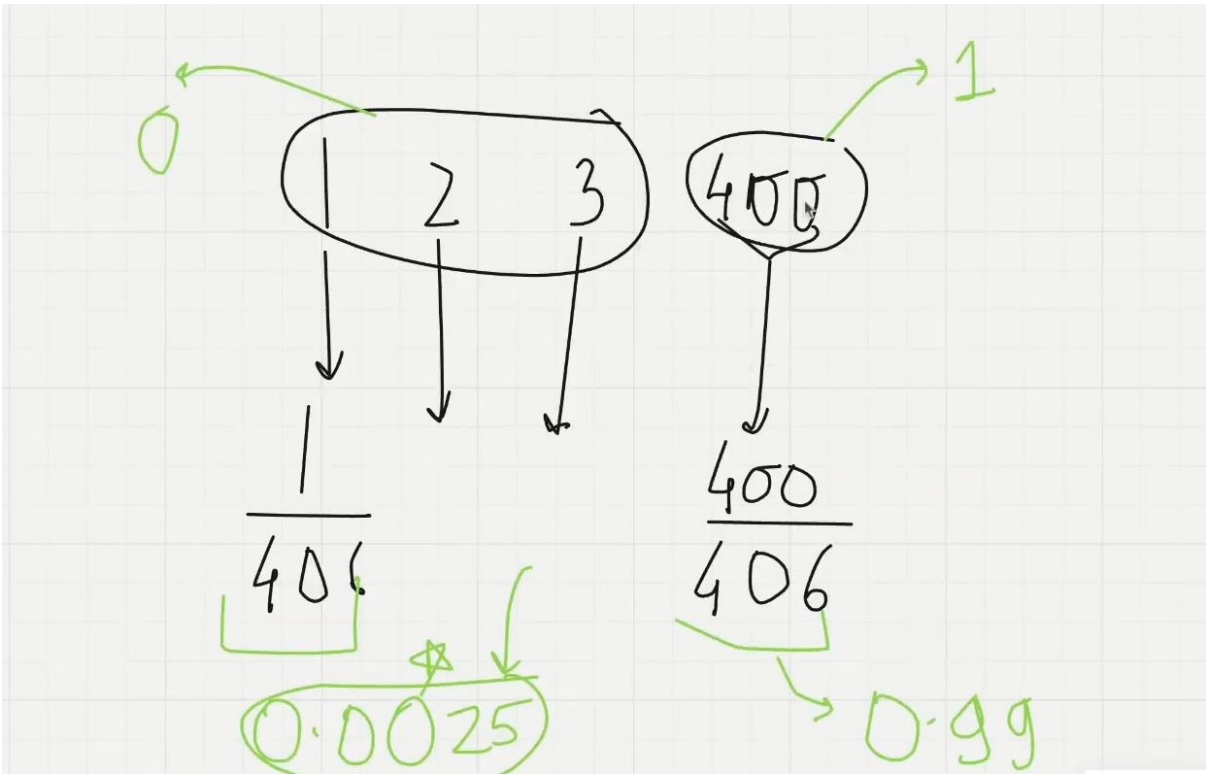
$$\left[\frac{e^{x_1}}{\text{sum}} \quad \frac{e^{x_2}}{\text{sum}} \quad \frac{e^{x_3}}{\text{sum}} \quad \frac{e^{x_4}}{\text{sum}} \quad \frac{e^{x_5}}{\text{sum}} \quad \frac{e^{x_6}}{\text{sum}}\right]$$

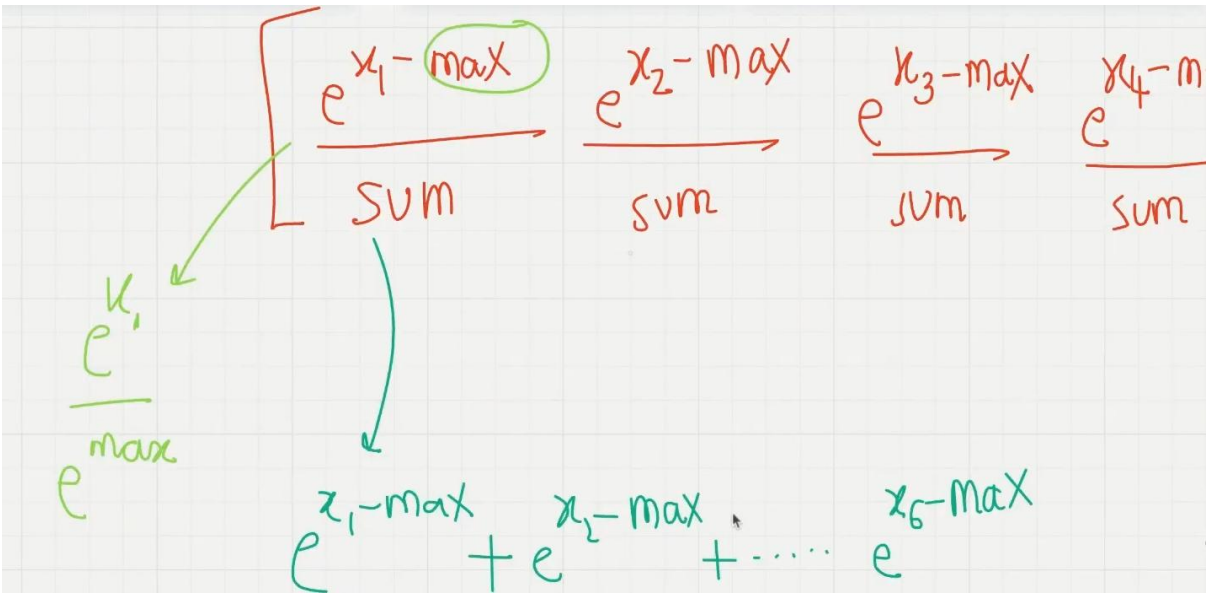
$$\text{sum} = e^{x_1} + e^{x_2} + e^{x_3} + e^{x_4} + e^{x_5} + e^{x_6}$$

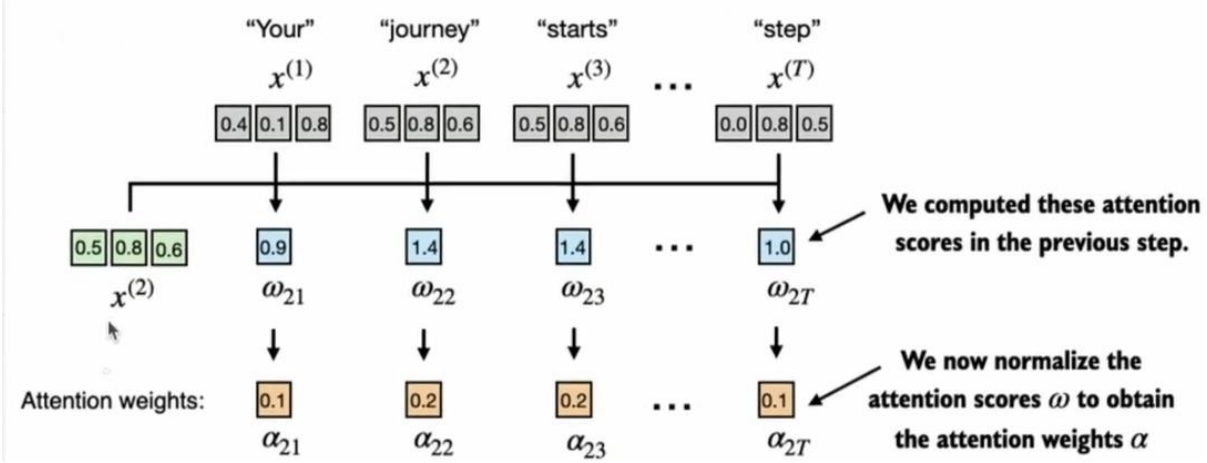
PyTorch Softmax

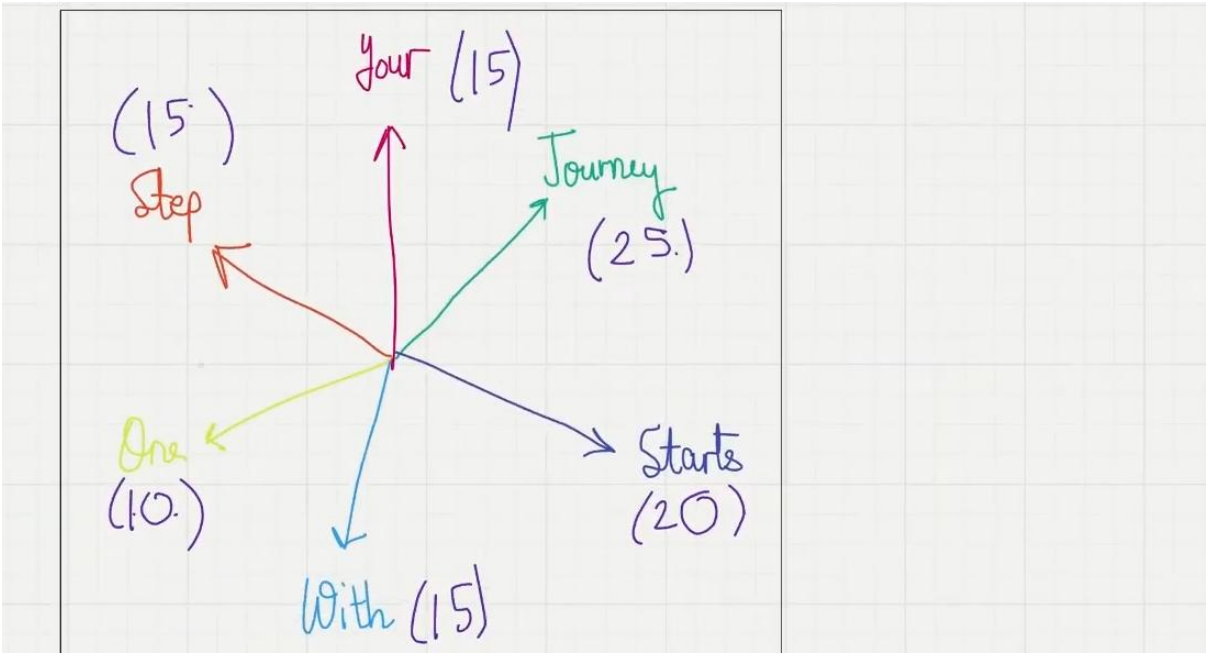
$$\left[x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6\right]$$

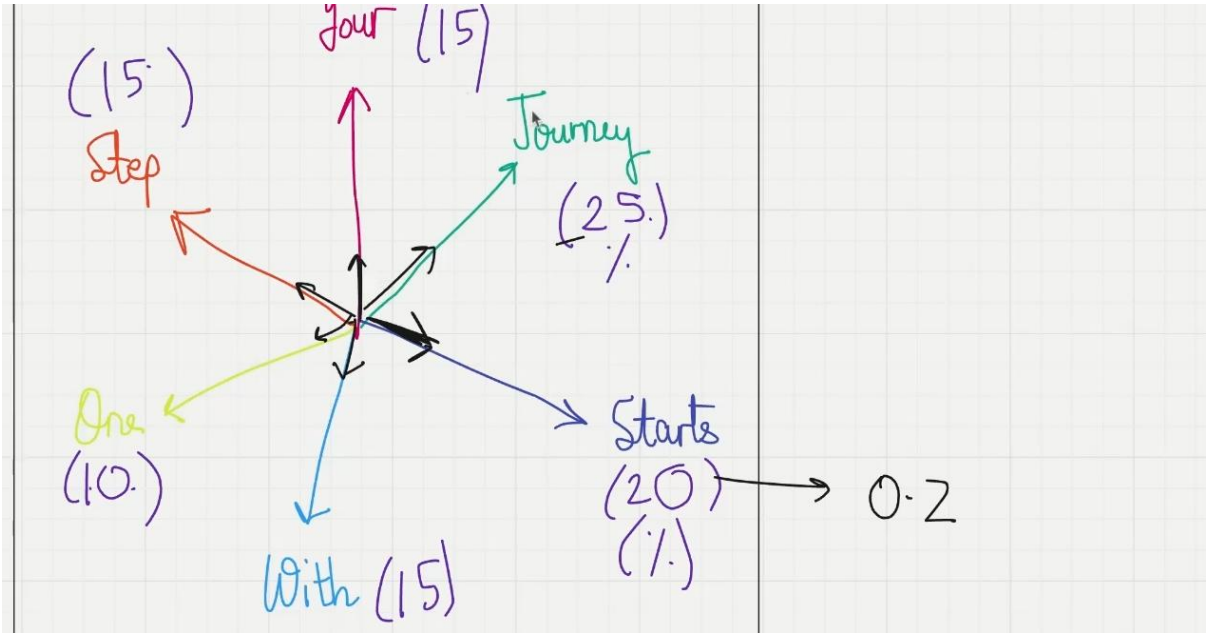
$$\left[\frac{e^{x_1 - \max}}{\text{sum}} \quad \frac{e^{x_2 - \max}}{\text{sum}} \quad \frac{e^{x_3 - \max}}{\text{sum}} \quad \frac{e^{x_4 - m}}{\text{sum}}\right.$$

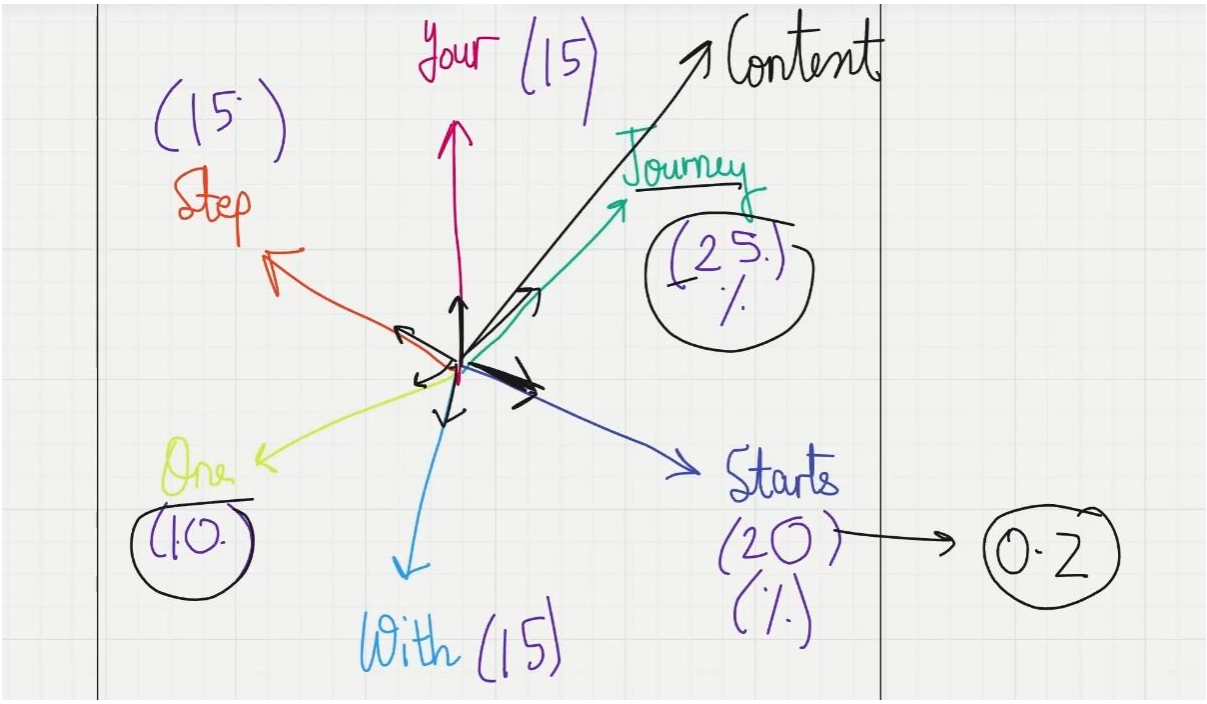

0
1 2 3
400
1
406
400
406
1
0.0025
0.99


$e^{x_1 - max}$
$e^{x_2 - max}$
$e^{x_3 - max}$
$e^{x_4 - m}$
sum
sum
sum
sum
e^{x_1}
e^{max}
$e^{x_1 - max} + e^{x_2 - max} + \cdots e^{x_6 - max}$

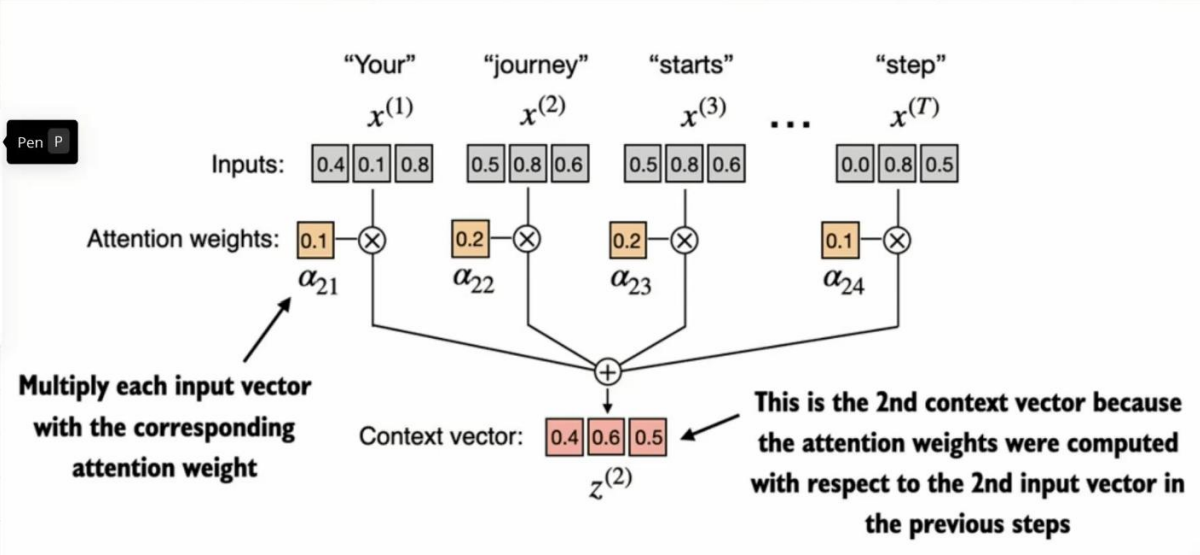

"Your"
"journey"
"starts"
"step"
$x^{(1)}$
$x^{(2)}$
$x^{(3)}$
$x^{(T)}$
0.4 0.1 0.8
0.5 0.8 0.6
0.5 0.8 0.6
0.0 0.8 0.5
0.5 0.8 0.6
$x^{(2)}$
0.9
1.4
1.4
1.0
ω_{21}
ω_{22}
ω_{23}
ω_{2T}
We computed these attention scores in the previous step.
Attention weights:
0.1
0.2
0.2
0.1
α_{21}
α_{22}
α_{23}
α_{2T}
We now normalize the attention scores ω to obtain the attention weights α


Your (15)
(15·)
Step
Journey
(25·)
One
(10·)
Starts
(20)
With (15)


Your (15)
(15·)
Step
Journey
(25·)
%
One
(10·)
Starts
(20)
(%)
0·2
With (15)

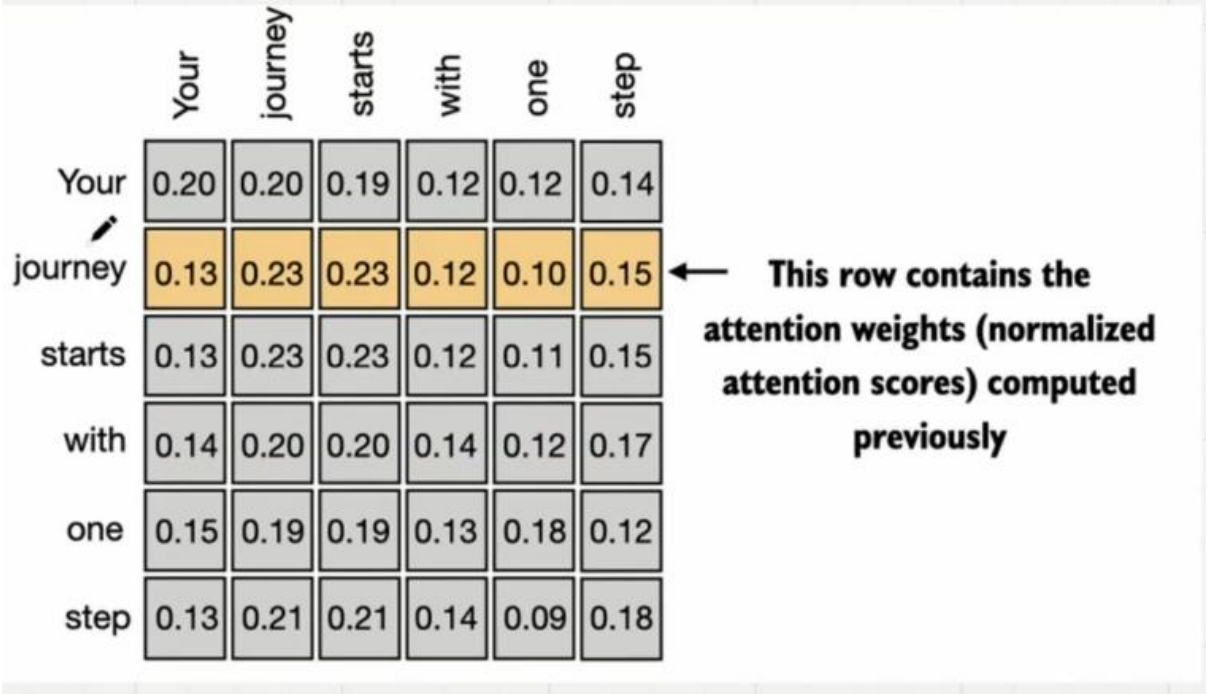


(8) After computing the normalized attention weights, we calculate the context vector $z^{(2)}$ by multiplying the embedded input tokens $x^{(i)}$, with the corresponding attention weights and then summing the resultant vectors.



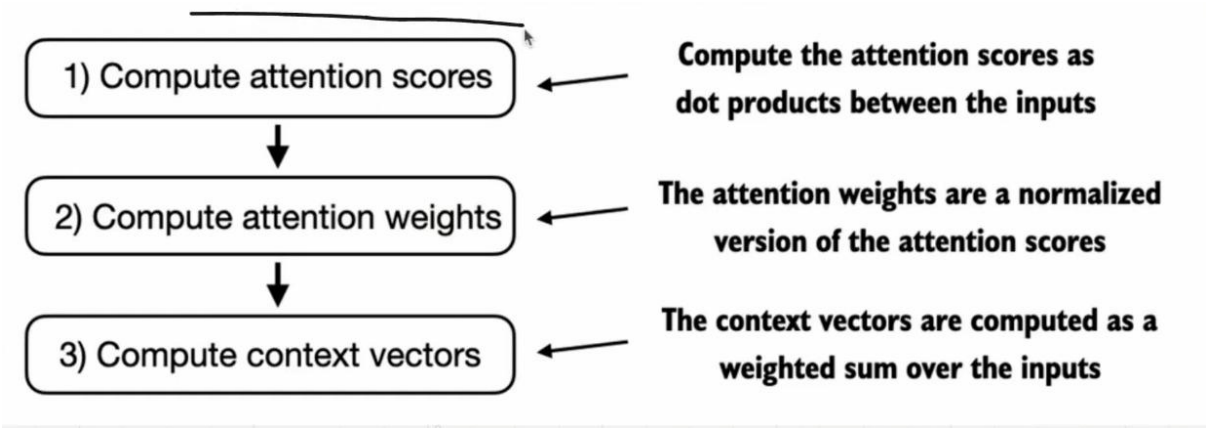
(9) Computing attention weights for all input tokens:

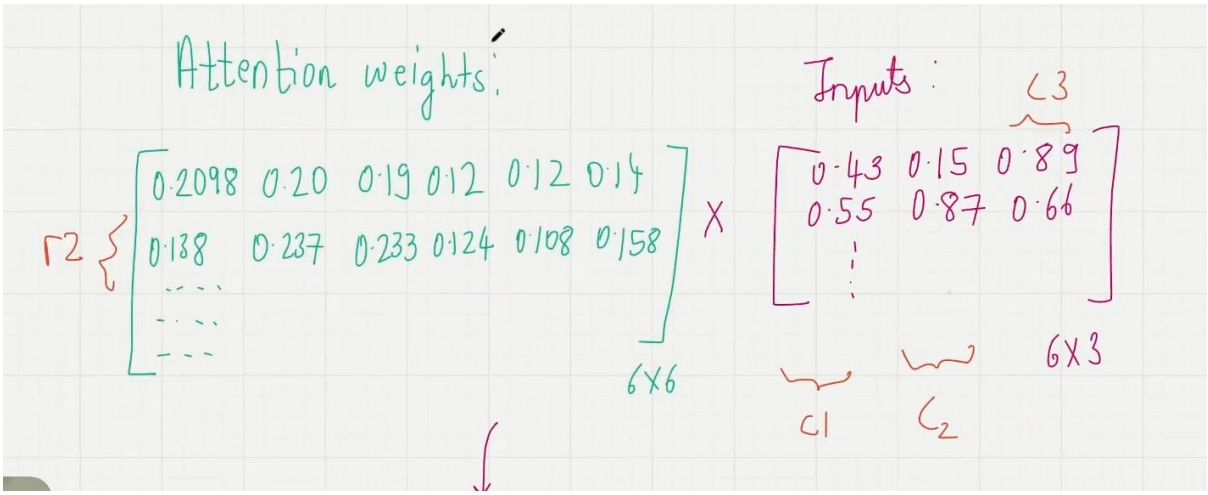
↳ We can extend similar computation as before to calculate attention weights and context vectors for all inputs

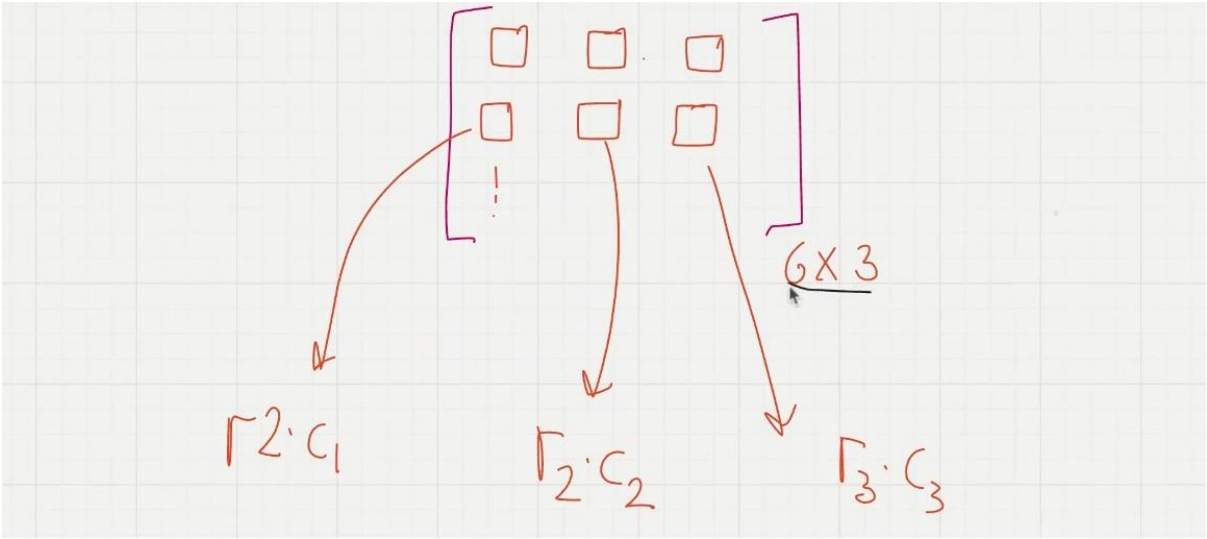


	Your	journey	starts	with	one	step
Your	0.20	0.20	0.19	0.12	0.12	0.14
journey	0.13	0.23	0.23	0.12	0.10	0.15
starts	0.13	0.23	0.23	0.12	0.11	0.15
with	0.14	0.20	0.20	0.14	0.12	0.17
one	0.15	0.19	0.19	0.13	0.18	0.12
step	0.13	0.21	0.21	0.14	0.09	0.18

(10) We will follow the same 3 steps as before:







$$= 0.138 * [0.43\ 0.15\ 0.89]$$

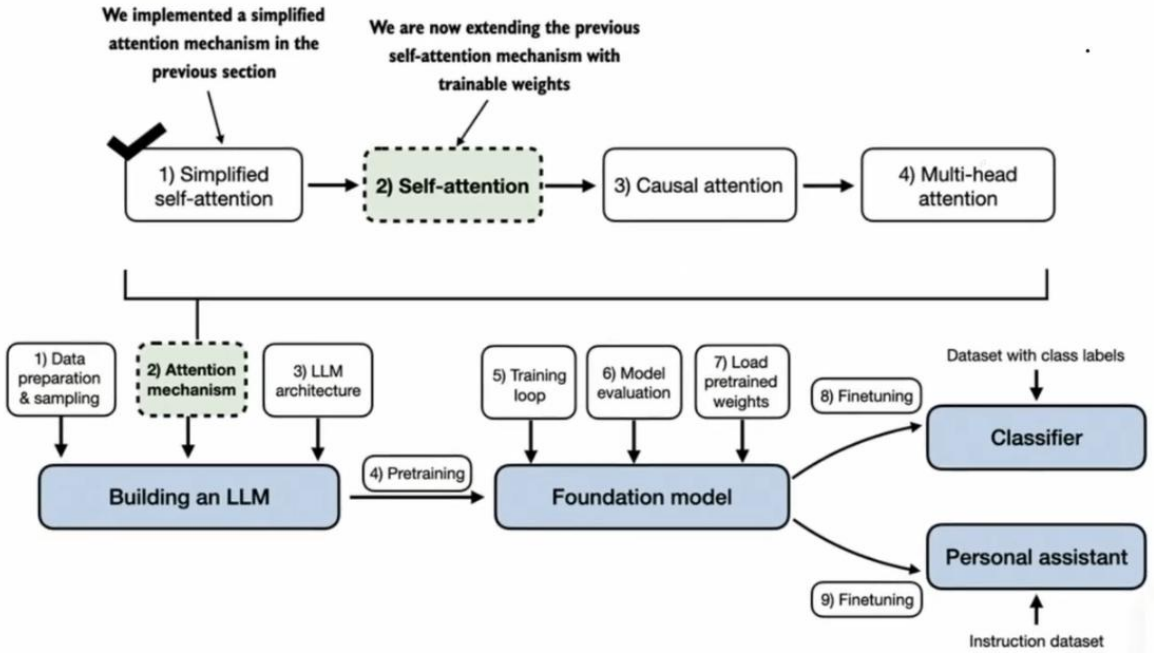
$$+ 0.237 * [0.55\ 0.87\ 0.66]$$

$$+ \ldots\ldots$$

Lecture → Self attention with trainable weights

↳ In this section, we will learn about the self attention mechanism used in the original transformer architecture, the GPT models and most other popular LLMs.

↳ This self attention mechanism is also called as "scaled dot-product attention"

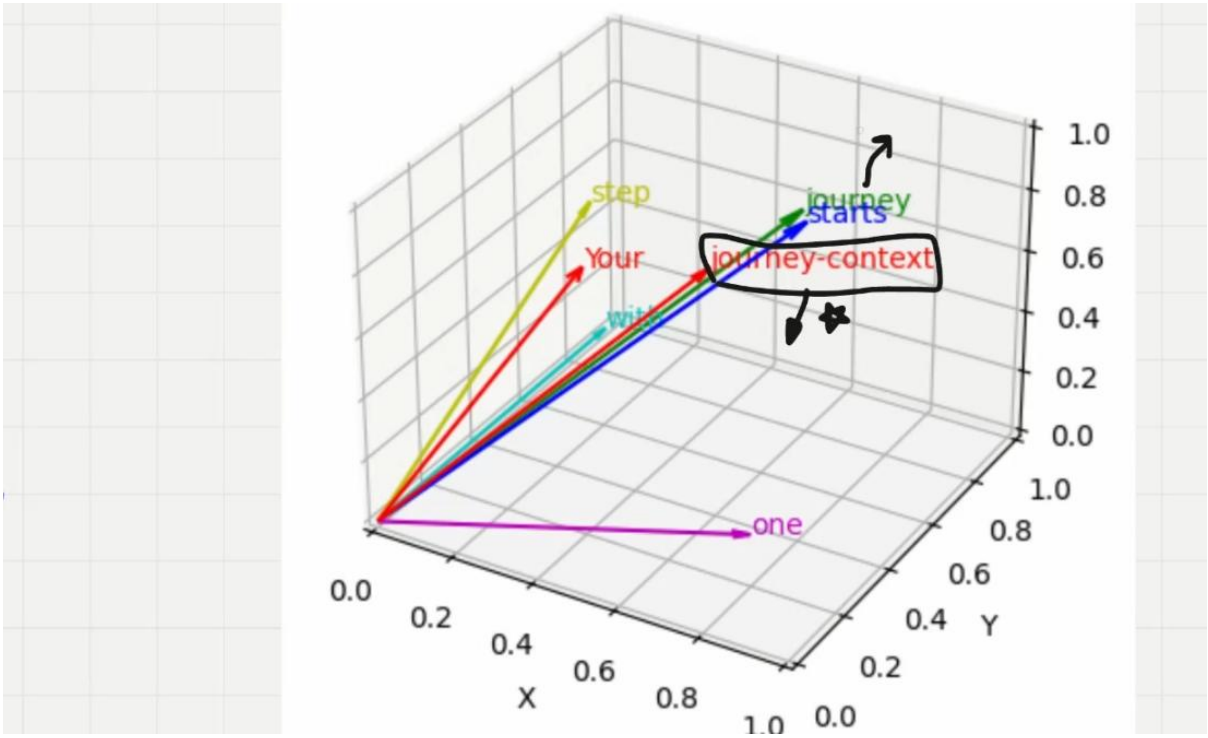


① We want to compute context vectors as weighted sums over the input vectors specific to a certain input element.

② We will introduce weight matrices that are updated during model training.

(2) We will introduce weight matrices that are updated during model training.

(3) These trainable weight matrices are crucial so that the model can learn to produce "good" context vectors.



(4) We will implement the self attention mechanism step by step by introducing 3 trainable weight matrices: W_q, W_K and W_V

↳ These 3 matrices are used to project the embedded input tokens $x^{(i)}$ into query, key and value vectors

① Converting input embeddings into key, query, value vectors

Inputs:

Your	0.43	0.15	0.89
Journey	0.55	0.87	0.66
starts	0.57	0.85	0.64
with	0.22	0.58	0.33
one	0.77	0.25	0.10
step	0.05	0.80	0.55

6X3

x_2

Inputs * W_q

W_q:

0.2961	0.5166
0.2517	0.6886
0.0740	0.8665

3X2

Inputs * W_K

W_K:

0.1366	0.1025
0.1841	0.7264
0.3153	0.6871

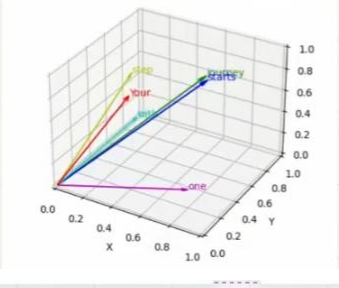
3X2

Inputs * W_v

W_v:

0.0756	0.1966
0.3164	0.4017
0.1186	0.8274

3X2



Inputs * W_q

W_q:

0.2961	0.5166
0.2517	0.6886
0.0740	0.8665

3X2

Queries:

0.23	1.09
0.43	1.45
0.43	1.43
0.23	0.79
0.29	0.65
0.25	1.05

6X2

Inputs * W_K

W_K:

0.1366	0.1025
0.1841	0.7264
0.3153	0.6871

3X2

Keys:

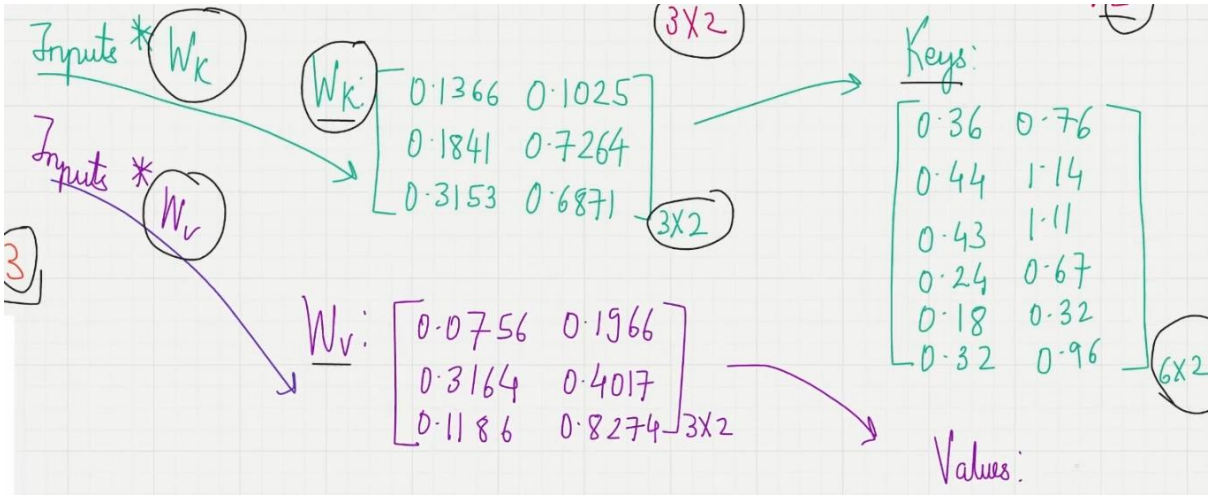
0.36	0.76
0.44	1.14
0.43	1.11
0.24	0.67
0.18	0.32
0.32	0.96

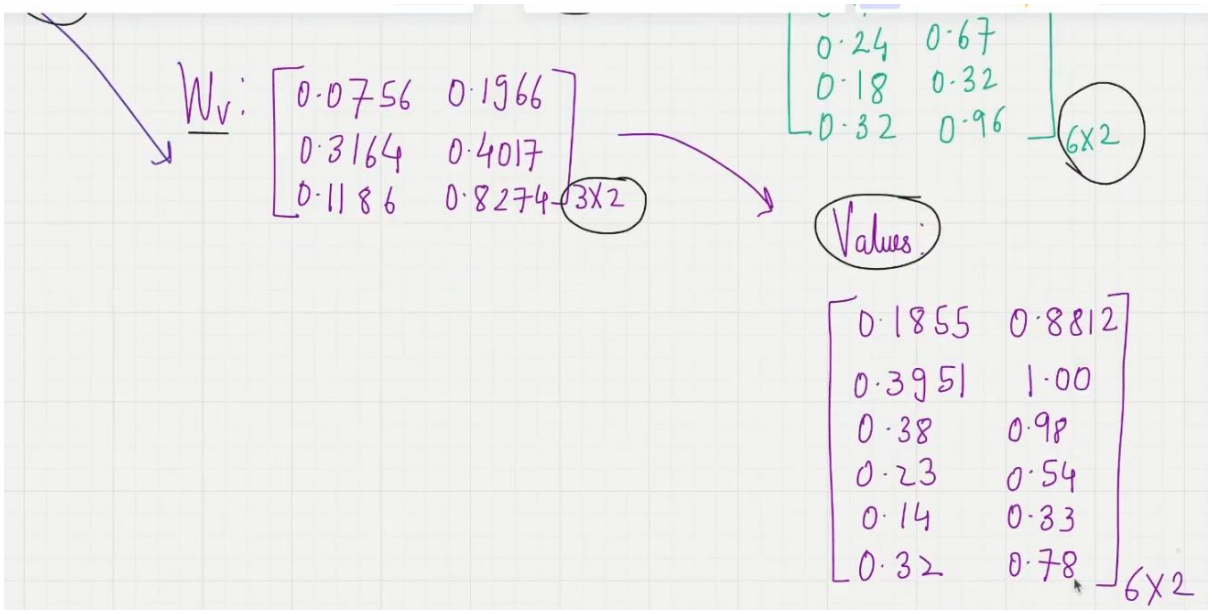
Inputs * W_v

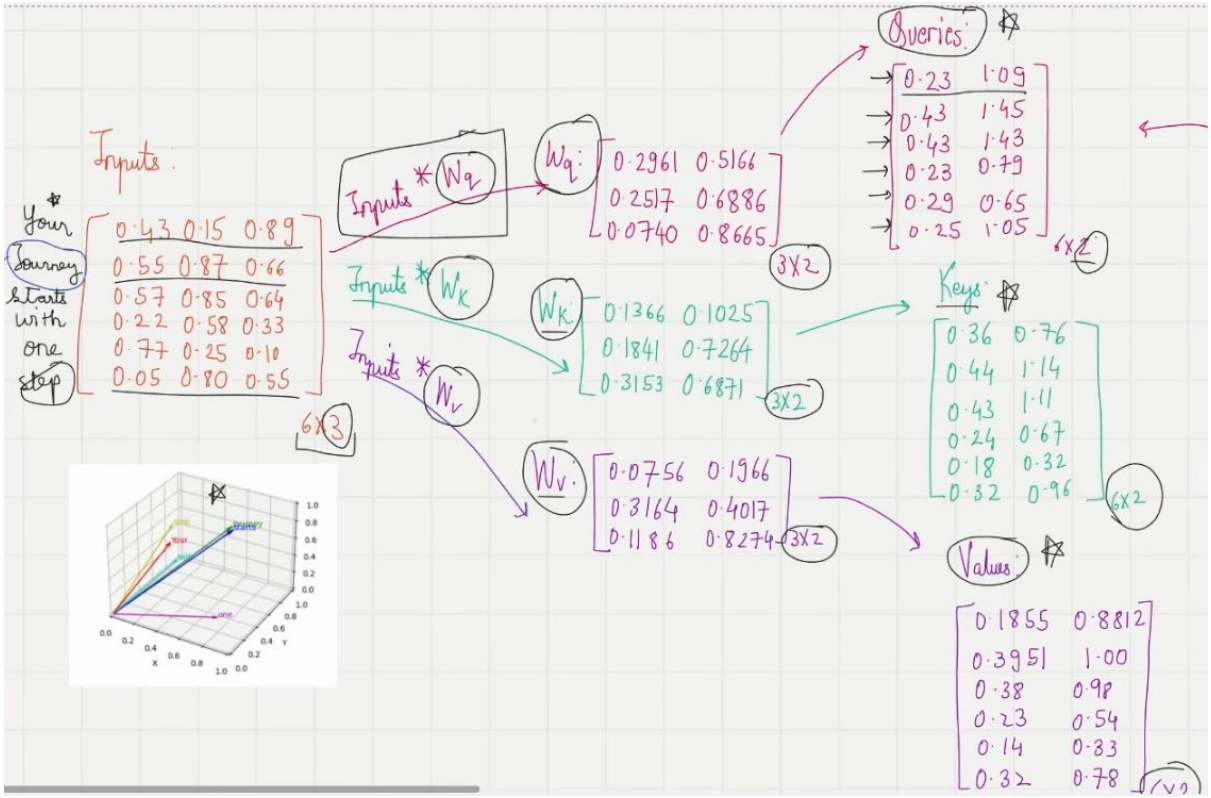
6X3

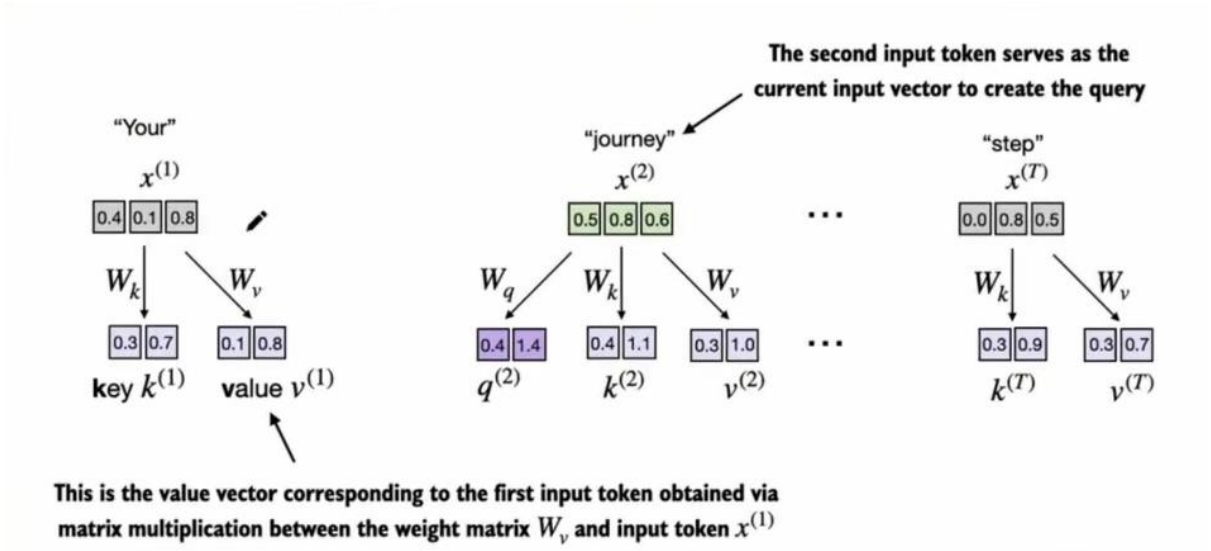
W_v:

0.0756	0.1966
0.3164	0.4017


Inputs * W_K
W_K
0·1366 0·1025
0·1841 0·7264
0·3153 0·6871
3X2
3X2
Keys:
0·36 0·76
0·44 1·14
0·43 1·11
0·24 0·67
0·18 0·32
0·32 0·96
6X2
Inputs * W_V
3
W_V:
0·0756 0·1966
0·3164 0·4017
0·1186 0·8274
3X2
Values:

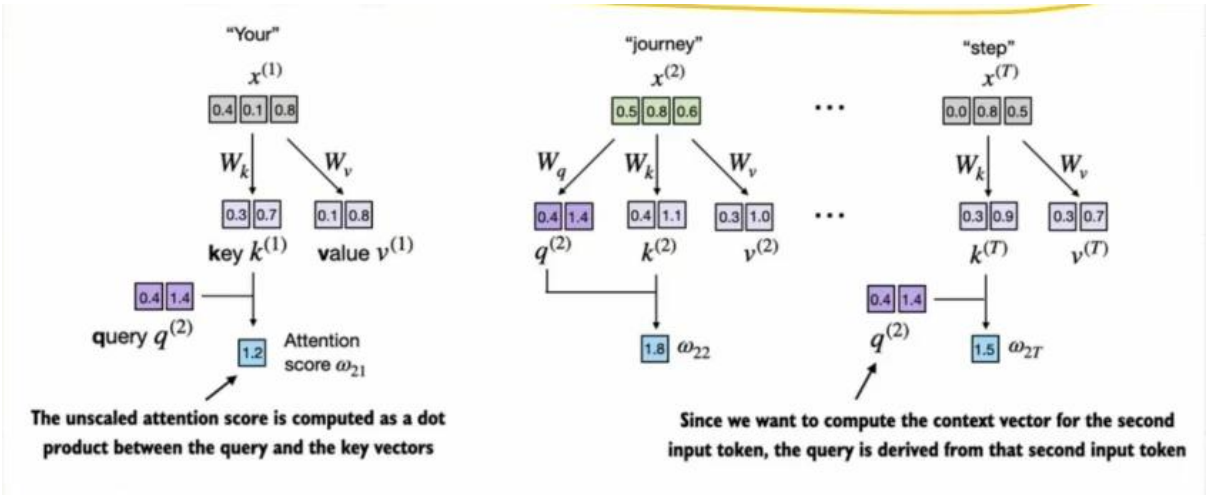

W_V:
0·0756 0·1966
0·3164 0·4017
0·1186 0·8274
3X2
0·24 0·67
0·18 0·32
0·32 0·96
6X2
Values:
0·1855 0·8812
0·3951 1·00
0·38 0·98
0·23 0·54
0·14 0·33
0·32 0·78
6X2

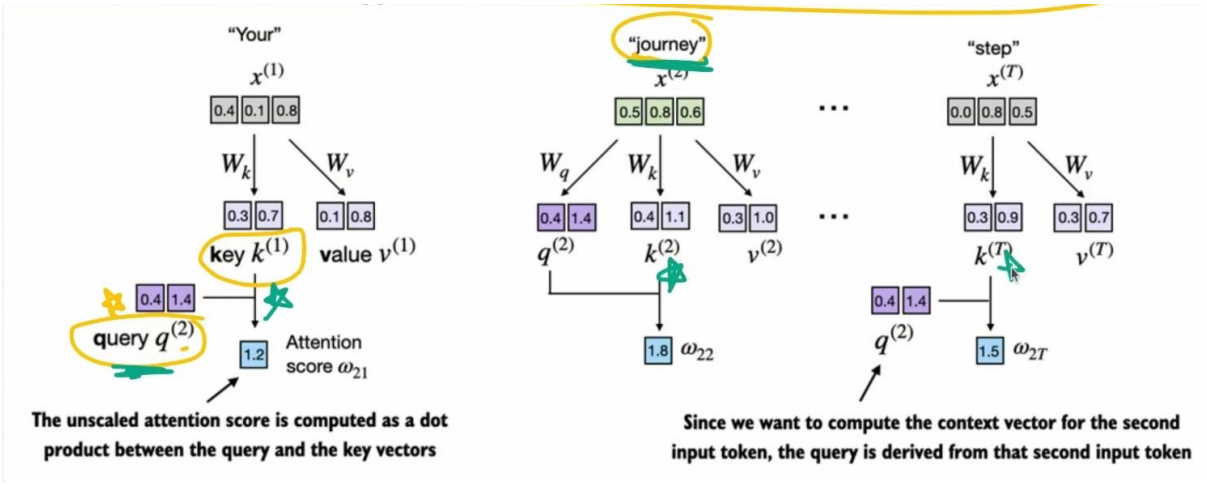




(5) We will start by computing only one context vector $z^{(2)}$ for illustration purposes

↳ Computing the attention scores:





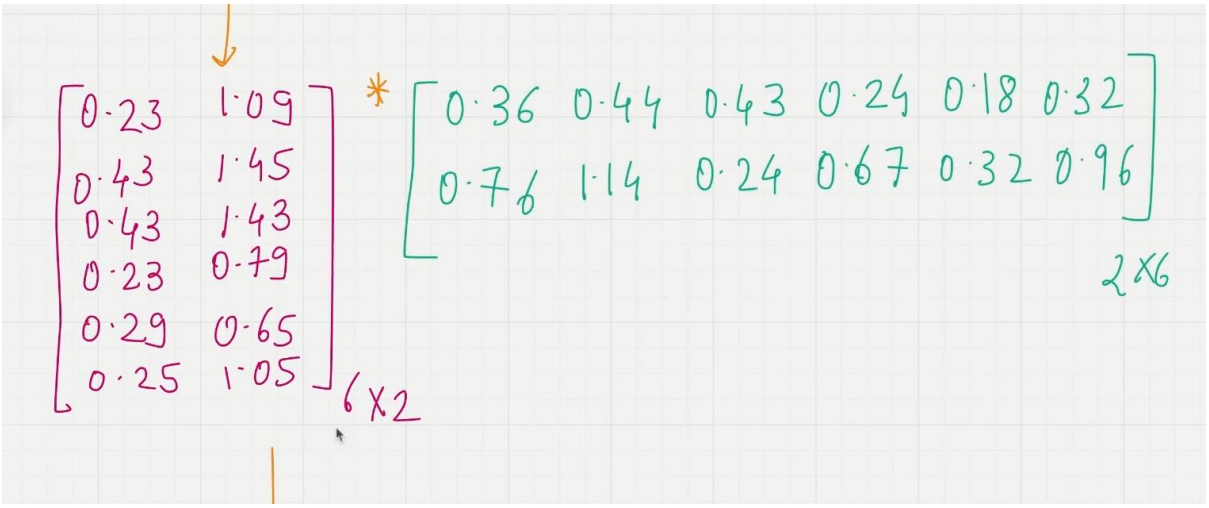
Queries:

$$\begin{bmatrix} 0.23 & 1.09 \\ 0.43 & 1.45 \\ 0.43 & 1.43 \\ 0.23 & 0.79 \\ 0.29 & 0.65 \\ 0.25 & 1.05 \end{bmatrix}_{6\times 2}$$

Keys:

$$\begin{bmatrix} 0.36 & 0.76 \\ 0.44 & 1.14 \\ 0.43 & 1.11 \\ 0.24 & 0.67 \\ 0.18 & 0.32 \\ 0.32 & 0.96 \end{bmatrix}_{6\times 2}$$

Queries * Keys^T




```
tensor([[0.9231, 1.3545, 1.3241, 0.7910, 0.4032, 1.1330],
        [1.2705, 1.8524, 1.8111, 1.0795, 0.5577, 1.5440],
        [1.2544, 1.8284, 1.7877, 1.0654, 0.5508, 1.5238],
        [0.6973, 1.0167, 0.9941, 0.5925, 0.3061, 0.8475],
        [0.6114, 0.8819, 0.8626, 0.5121, 0.2707, 0.7307],
        [0.8995, 1.3165, 1.2871, 0.7682, 0.3937, 1.0996]])
```

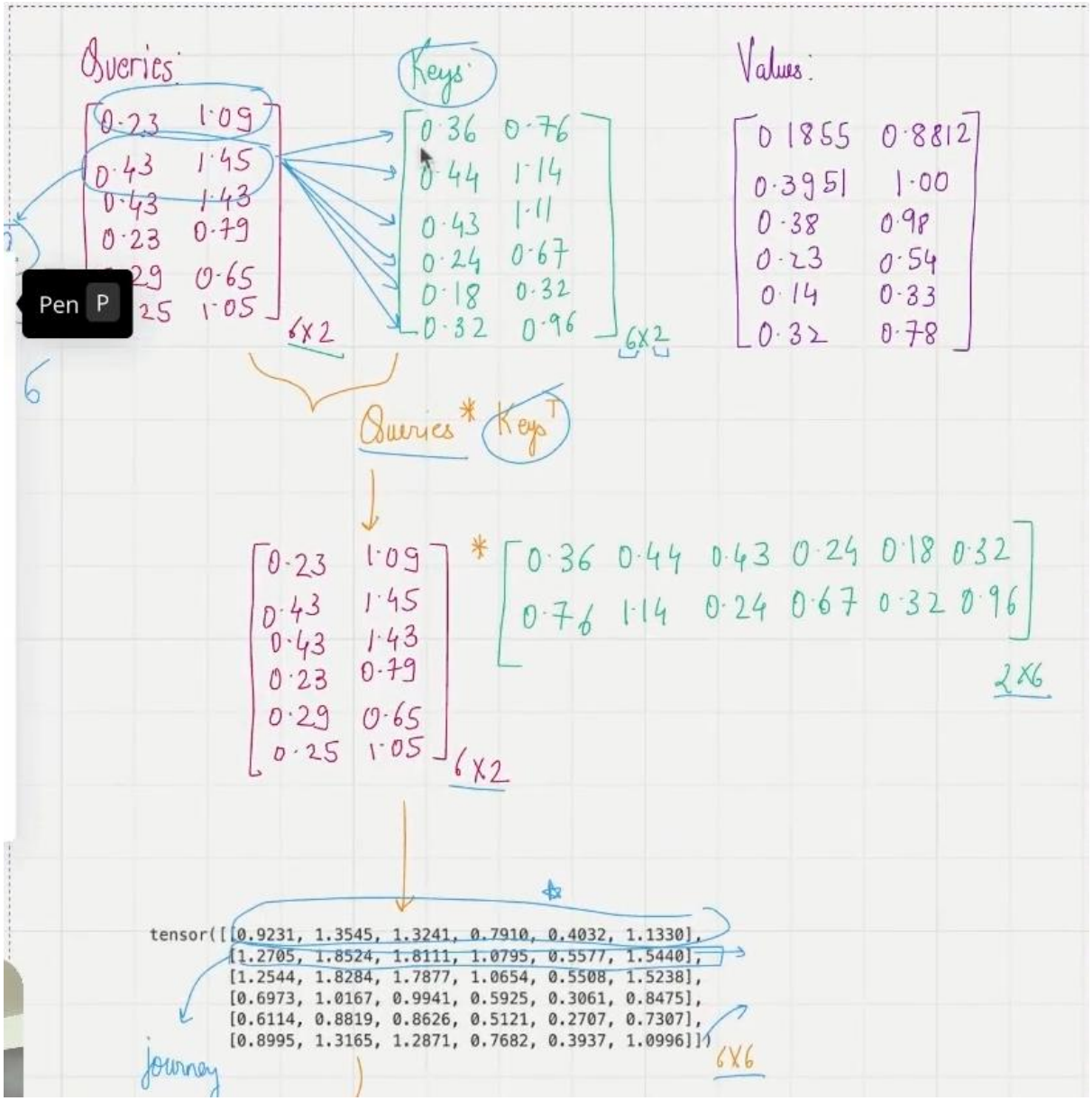
6X6

Attention scores

③ Attention weights

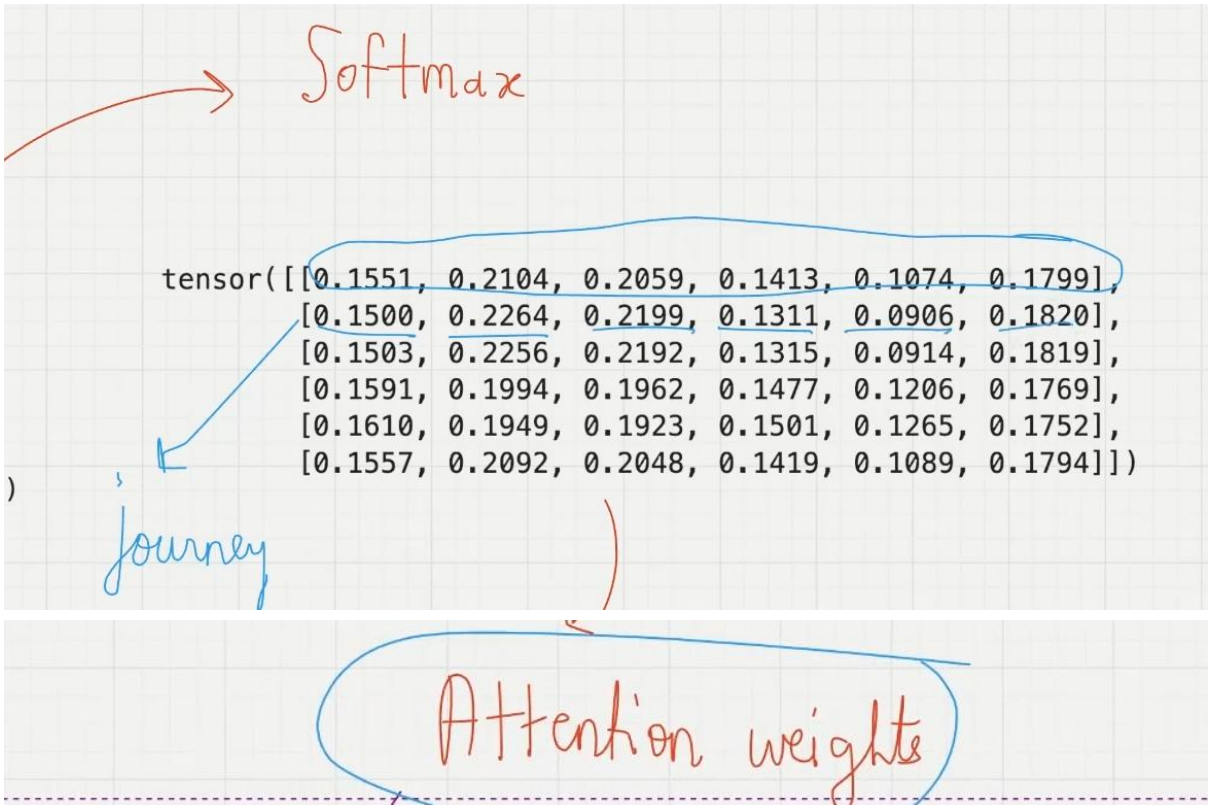
Scale by $\sqrt{d\_keys}$

```
tensor([[0.6528, 0.9578, 0.9363, 0.5593, 0.2851, 0.8011],
        [0.8984, 1.3098, 1.2806, 0.7633, 0.3944, 1.0918],
        [0.8870, 1.2929, 1.2641, 0.7534, 0.3895, 1.0775],
        [0.4930, 0.7189, 0.7029, 0.4190, 0.2164, 0.5993],
        [0.4323, 0.6236, 0.6099, 0.3621, 0.1914, 0.5167],
        [0.6361, 0.9309, 0.9101, 0.5432, 0.2784, 0.7776]])
```



Scale by $\sqrt{d\_keys}$

```
tensor([[0.6528, 0.9578, 0.9363, 0.5593, 0.2851, 0.8011],
        [0.8984, 1.3098, 1.2806, 0.7633, 0.3944, 1.0918],
        [0.8870, 1.2929, 1.2641, 0.7534, 0.3895, 1.0775],
        [0.4930, 0.7189, 0.7029, 0.4190, 0.2164, 0.5993],
        [0.4323, 0.6236, 0.6099, 0.3621, 0.1914, 0.5167],
        [0.6361, 0.9309, 0.9101, 0.5432, 0.2784, 0.7776]])
```



WHY DIVIDE BY SQRT (DIMENSION)

Reason 1: For stability in learning

The softmax function is sensitive to the magnitudes of its inputs. When the inputs are large, the differe much more pronounced. This causes the softmax output to become "peaky," where the highest value r very little.

In attention mechanisms, particularly in transformers, if the dot products between query and key vecto the attention scores can become very large. This results in a very sharp softmax distribution, making th distributions can make learning unstable,

- The **softmax function** is sensitive to the magnitudes of its inputs.
- When inputs are large, the exponential values become more pronounced.
- This causes the softmax output to become **"peaky,"** where one value dominates almost all the probability mass, and others receive very little.
- In **attention mechanisms** (especially in transformers), large dot products between query and key vectors can produce large attention scores.
- Large scores lead to **sharp softmax distributions**, making the model **overly confident** in a single key.
- This can cause **training instability**.
- Dividing by the **square root of the dimension** normalizes these scores.
- This normalization helps **prevent scores from becoming excessively large** and results in **more stable learning**.

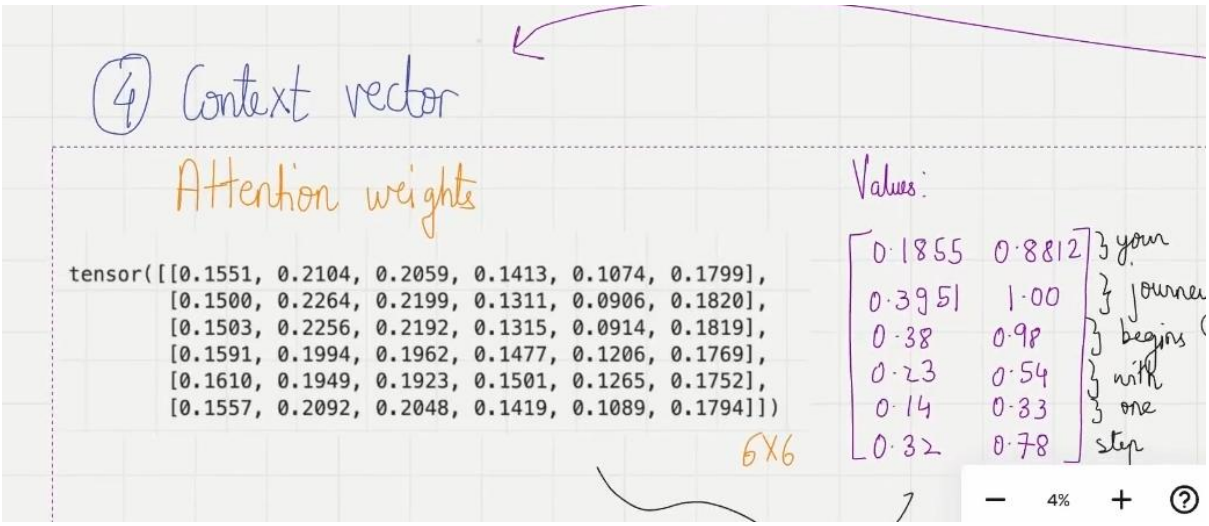
BUT WHY SQRT?

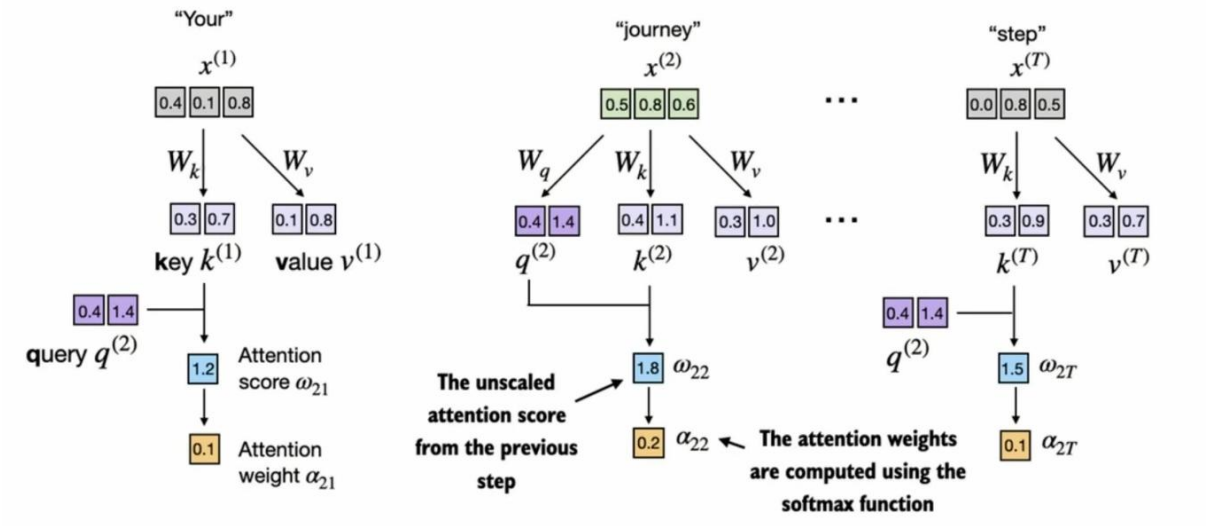
Reason 2: To make the variance of the dot product stable

The dot product of Q and K increases the variance because multiplying two random numbers

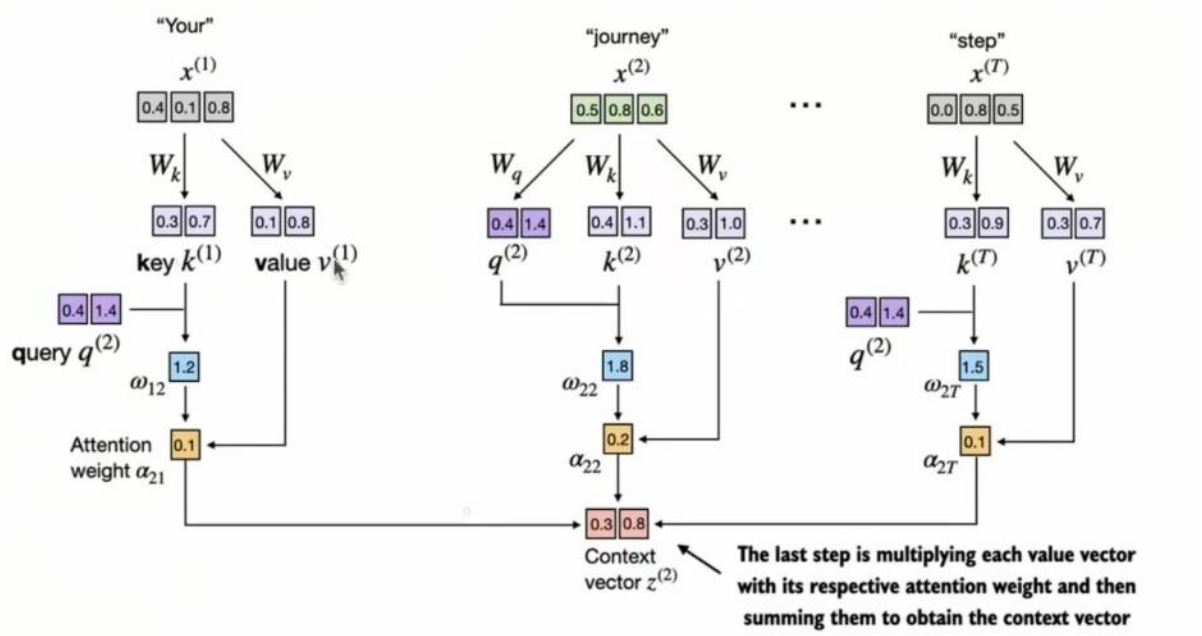
The increase in variance grows with the dimension.

Dividing by sqrt (dimension) keeps the variance close to 1





↳ Final step: Compute context vectors



④ Context vector

Attention weights

```
tensor([[0.1551, 0.2104, 0.2059, 0.1413, 0.1074, 0.1799],
        [0.1500, 0.2264, 0.2199, 0.1311, 0.0906, 0.1820],
        [0.1503, 0.2256, 0.2192, 0.1315, 0.0914, 0.1819],
        [0.1591, 0.1994, 0.1962, 0.1477, 0.1206, 0.1769],
        [0.1610, 0.1949, 0.1923, 0.1501, 0.1265, 0.1752],
        [0.1557, 0.2092, 0.2048, 0.1419, 0.1089, 0.1794]])
```

6X6

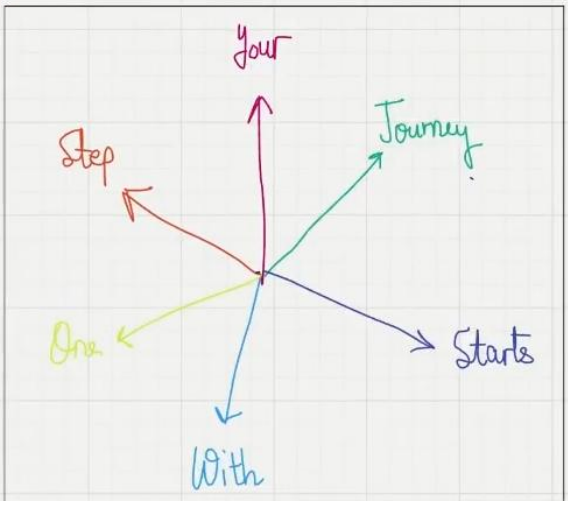
Values:

$$\begin{bmatrix} 0.1855 & 0.8812 \\ 0.3951 & 1.00 \\ 0.38 & 0.98 \\ 0.23 & 0.54 \\ 0.14 & 0.33 \\ 0.32 & 0.78 \end{bmatrix}$$

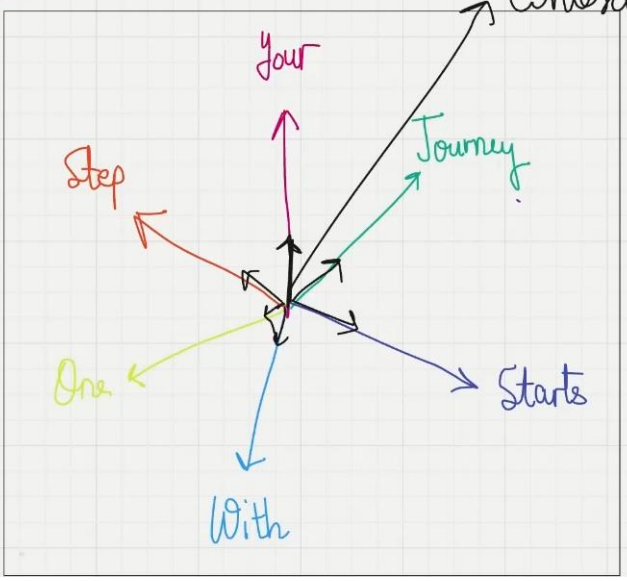
} your
} journey
} begins
} with
} one
} step

6X2

What this means



What this means



Attention weights * Values

```
tensor([[0.2996, 0.8053],
        [0.3061, 0.8210],
        [0.3058, 0.8203],
        [0.2948, 0.7939],
        [0.2927, 0.7891],
        [0.2990, 0.8040]])
```

Context vector matrix

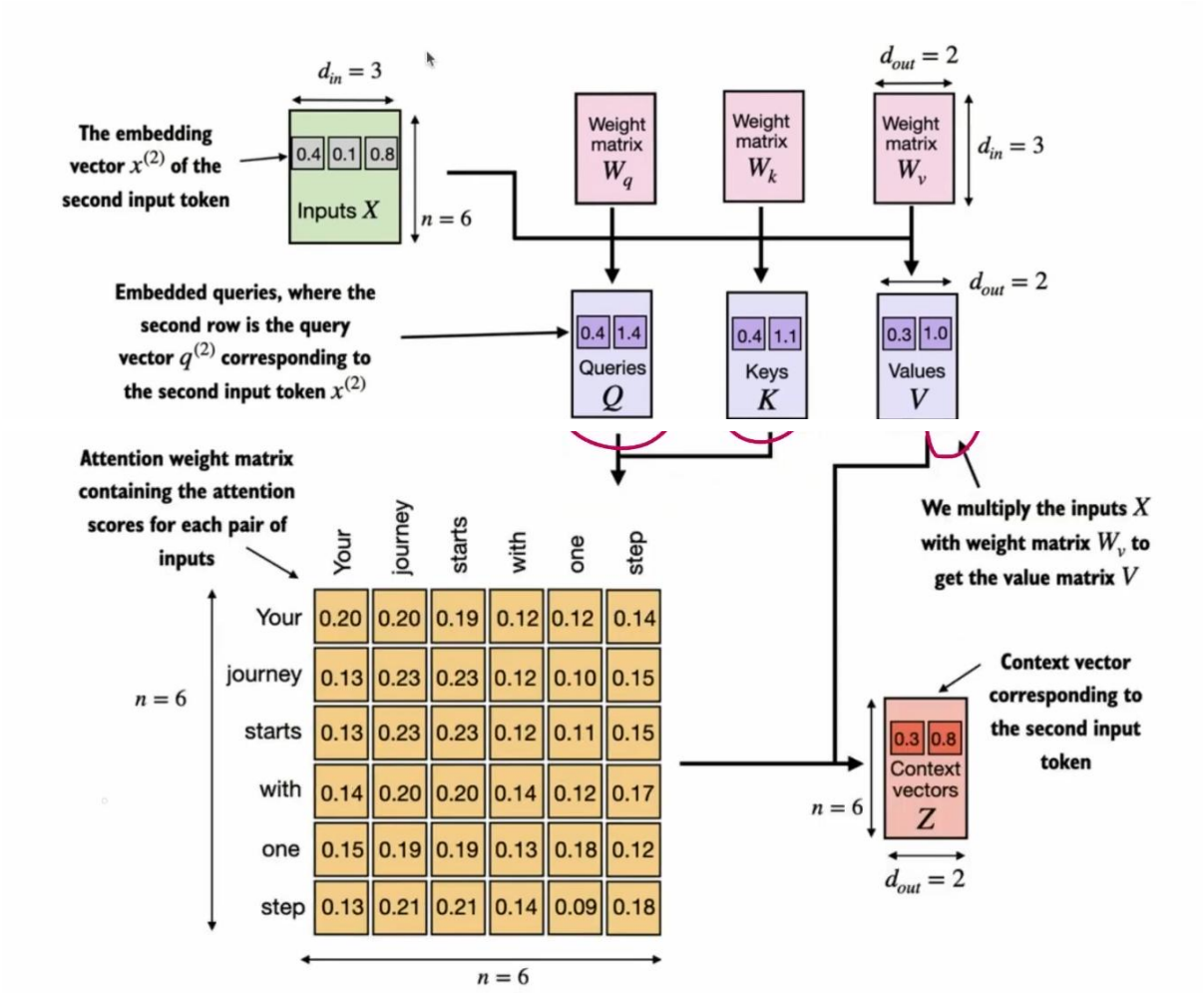
* Why do we use the terms: key, query and value?

Query: Analogous to search query in a database. It represents the current token the model focuses on

Key In attention mechanism, each item in input sequence has a key. Keys are used to match with the query.

Value It represents the actual content or representation of the input items. Once the model determines which keys (which parts of the input) are most relevant to the query (current focus item), it retrieves the corresponding values.

⑥ Implementing a compact self attention Python class



* Self attention involves the trainable weight matrices: W_q, W_K and W_v

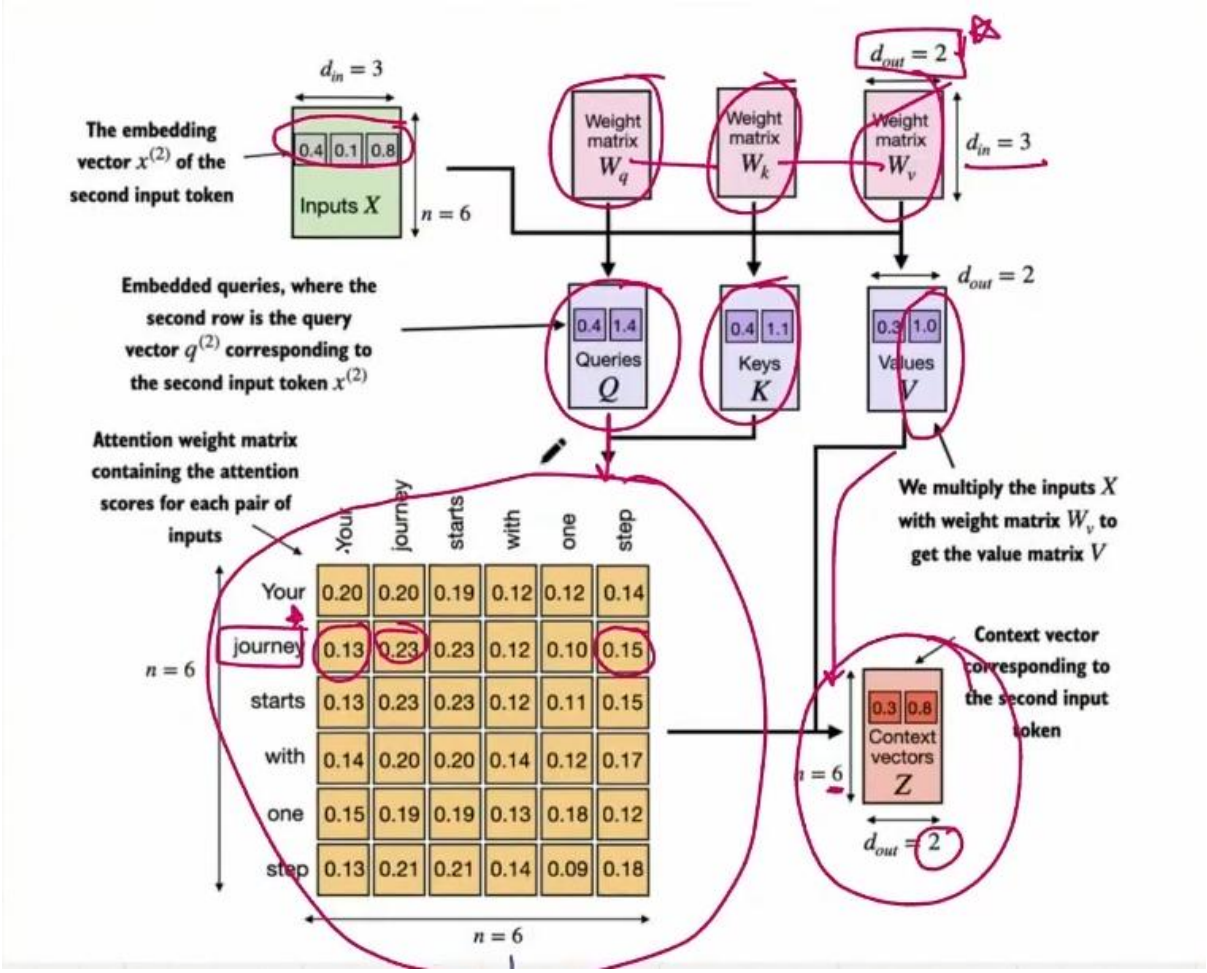
* These matrices transform input data into queries, keys and values; which are crucial components of the attention mechanism.

* Why do we use the terms: key, query and value?

Query: Analogous to search query in a database. It represents the current token the model focuses on

Key: In attention mechanism, each item in input sequence has a key. Keys are used to match with the query

Value: It represents the actual content or representation of the input items. Once the model determines which keys (which parts of the input) are most relevant to the query (current focus item), it retrieves the corresponding values.



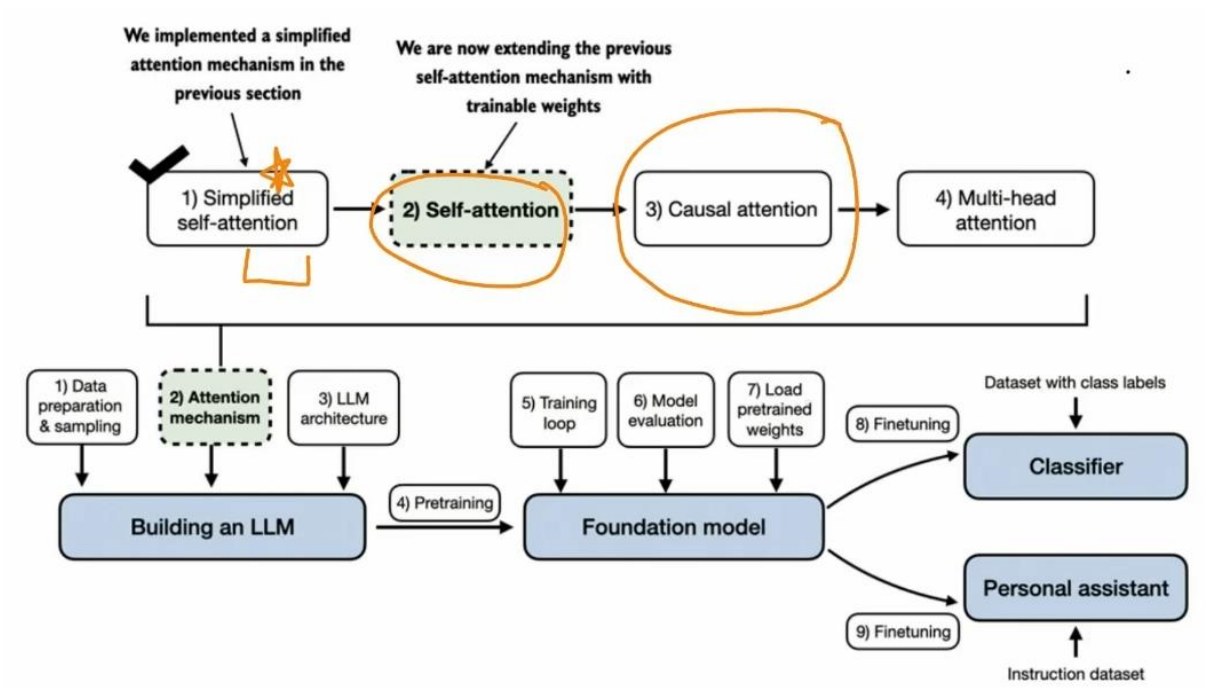
* Self attention involves the trainable weight matrices: W_q, W_K and W_v

* These matrices transform input data into queries, keys and values; which are crucial components of the attention mechanism.

In the next section, we will look at:

- Causal attention
  - Modify attention mechanism → prevents model from accessing future information in a sequence, which is crucial for tasks like language modeling.
- Multi head attention
  - Split the attention mechanism into multiple "heads". Each head learns different aspects of the data → improves model performance in complex tasks.

Lecture → Causal attention



Recap (until now)

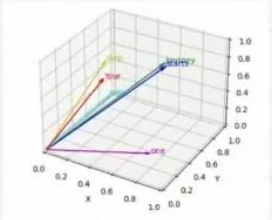
Inputs

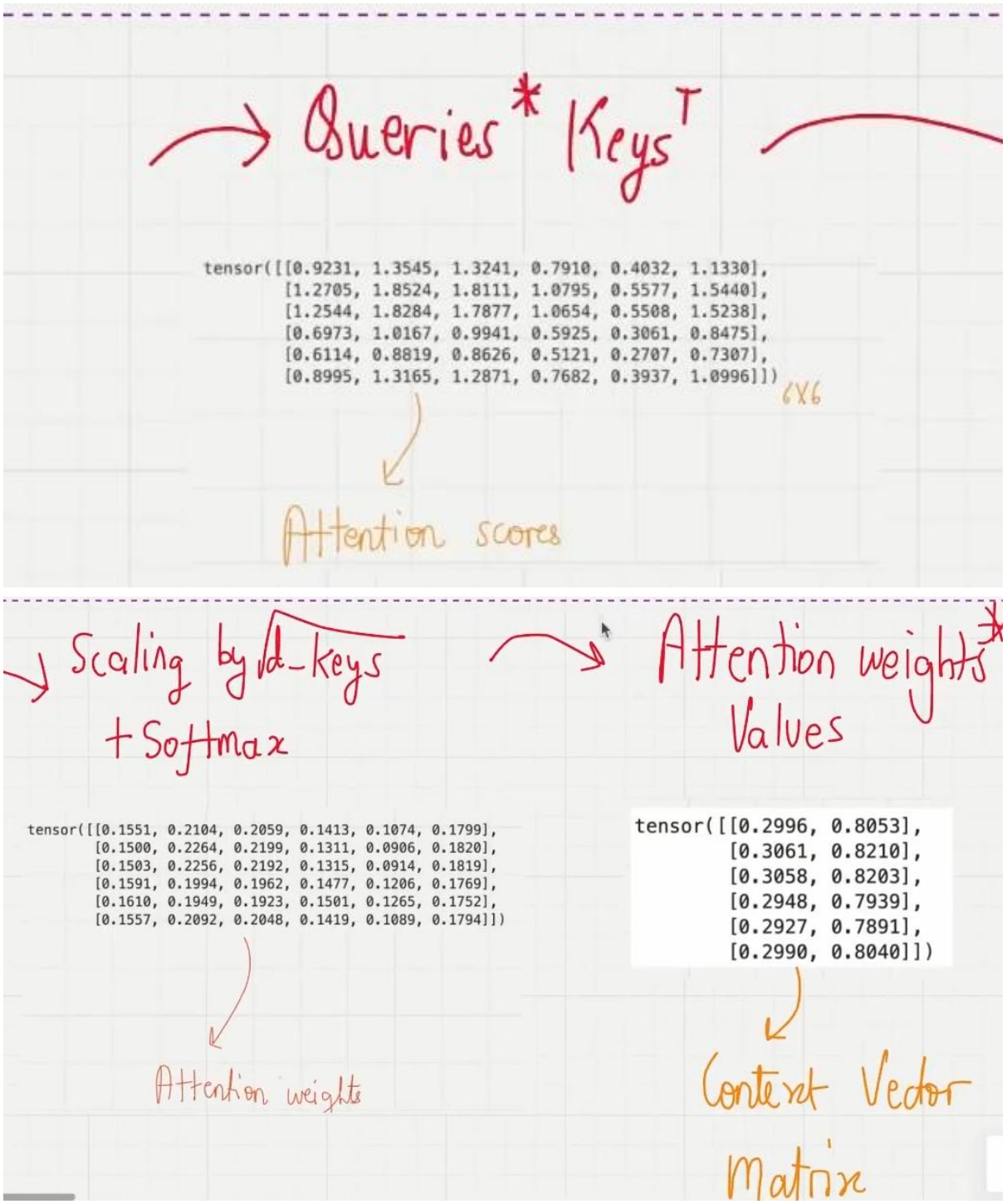
Your [0.43 0.15 0.89]
Journey [0.55 0.87 0.66]
Starts [0.57 0.85 0.64]
with [0.22 0.58 0.33]
one [0.77 0.25 0.10]
step [0.05 0.80 0.55]
6X3

Inputs * W_q → W_q: [0.2961 0.5166; 0.2517 0.6886; 0.0740 0.8665] 3X2 → Queries: [0.23 1.09; 0.43 1.45; 0.43 1.43; 0.23 0.79; 0.29 0.65; 0.25 1.05] 6X2

Inputs * W_k → W_k: [0.1366 0.1025; 0.1841 0.7264; 0.3153 0.6871] 3X2 → Keys: [0.36 0.76; 0.44 1.14; 0.43 1.11; 0.24 0.67; 0.18 0.32; 0.32 0.96] 6X2

Inputs * W_v → W_v: [0.0756 0.1966; 0.3164 0.4017; 0.1186 0.8274] 3X2 → Values: [0.1855 0.8812; 0.3951 1.00; 0.38 0.98; 0.23 0.54; 0.14 0.33; 0.32 0.78] 6X2




Queries * Keys^T
tensor([[0.9231, 1.3545, 1.3241, 0.7910, 0.4032, 1.1330],
[1.2705, 1.8524, 1.8111, 1.0795, 0.5577, 1.5440],
[1.2544, 1.8284, 1.7877, 1.0654, 0.5508, 1.5238],
[0.6973, 1.0167, 0.9941, 0.5925, 0.3061, 0.8475],
[0.6114, 0.8819, 0.8626, 0.5121, 0.2707, 0.7307],
[0.8995, 1.3165, 1.2871, 0.7682, 0.3937, 1.0996]])
6X6
Attention scores
Scaling by √d_keys
+ Softmax
tensor([[0.1551, 0.2104, 0.2059, 0.1413, 0.1074, 0.1799],
[0.1500, 0.2264, 0.2199, 0.1311, 0.0906, 0.1820],
[0.1503, 0.2256, 0.2192, 0.1315, 0.0914, 0.1819],
[0.1591, 0.1994, 0.1962, 0.1477, 0.1206, 0.1769],
[0.1610, 0.1949, 0.1923, 0.1501, 0.1265, 0.1752],
[0.1557, 0.2092, 0.2048, 0.1419, 0.1089, 0.1794]])
Attention weights
Attention weights * Values
tensor([[0.2996, 0.8053],
[0.3061, 0.8210],
[0.3058, 0.8203],
[0.2948, 0.7939],
[0.2927, 0.7891],
[0.2990, 0.8040]])
Context Vector Matrix

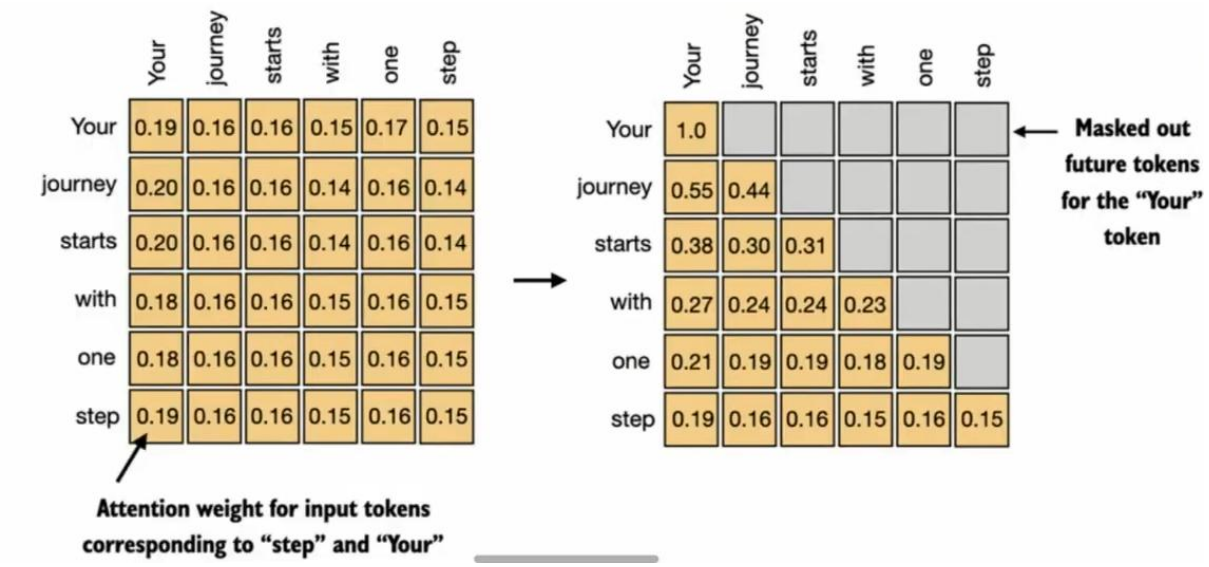
(1) Causal attention, also known as masked attention is a special form of self attention.

(2) It restricts the model to only consider previous and current inputs in a sequence, when processing any given token.

(3) This is in contrast to the self attention mechanism, which allows access to the entire input sequence.

(4) When computing attention scores, the causal attention mechanism ensures that the model only factors in tokens that occur at or before the current token in the sequence.

(5) To achieve this in GPT like LLMs, for each token processed, we mask out the future tokens, which come after the current token in the input text.

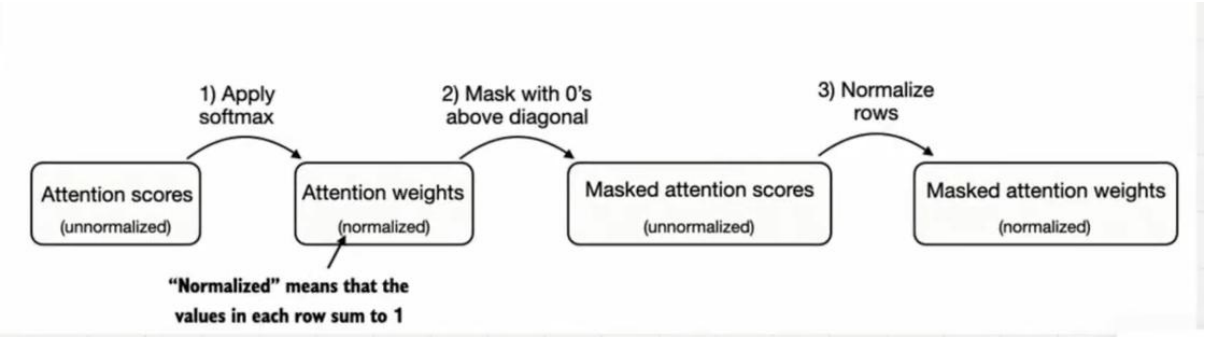


↳ We mask out the attention weights above the diagonal, and we normalize the non masked attention weights, such that the attention weights sum upto 1 in each row.

* Applying a Causal Attention Mask *

Strategy: Get attention weights

↳ Zero out elements above the diagonal and normalize the resulting matrix.



More efficient way

Attention scores → Upper Triangular infinity mask → Softmax

Masking additional attention weights with dropout *

↳ Dropout is a deep learning technique where randomly selected hidden layer units are ignored during training

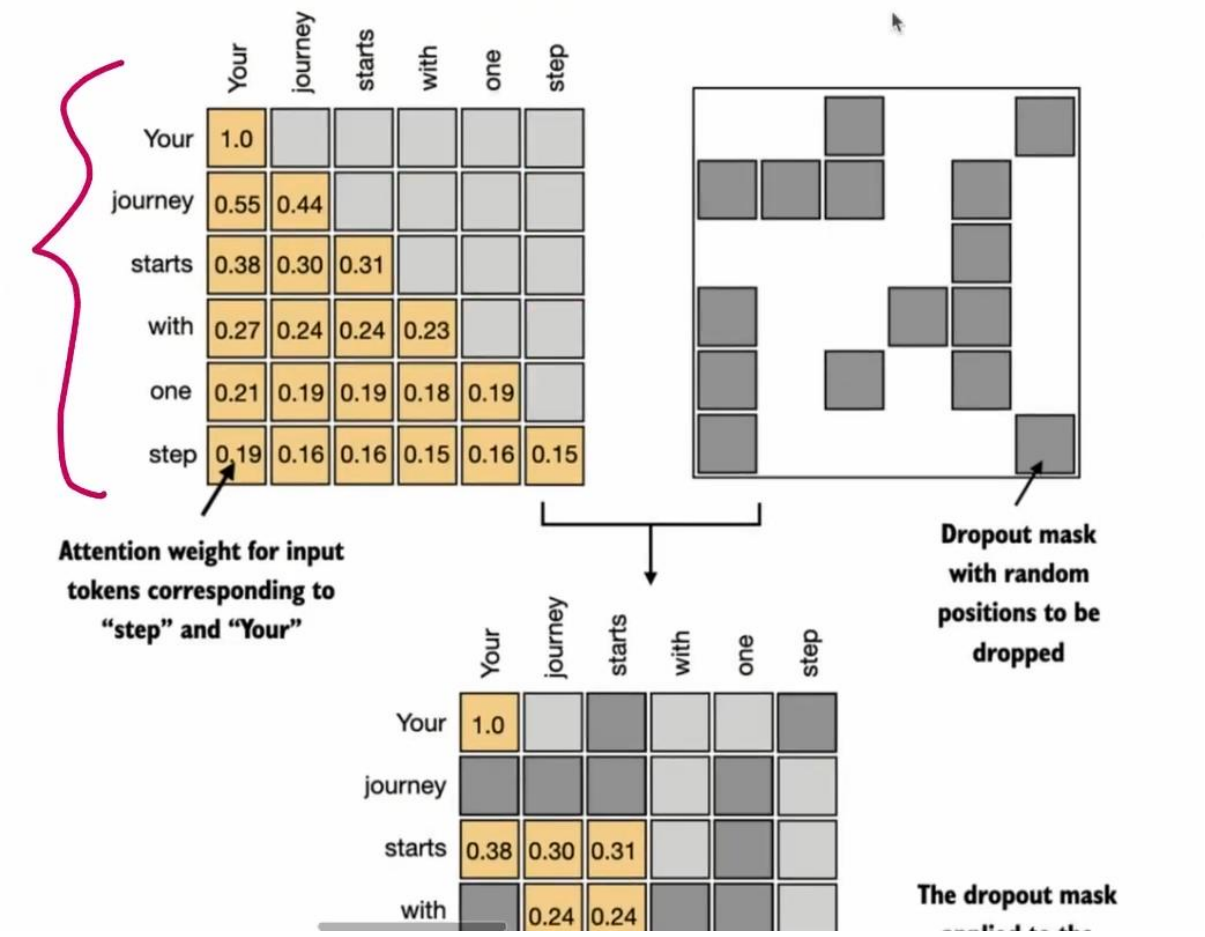
This prevents overfitting and improves generalization performance.

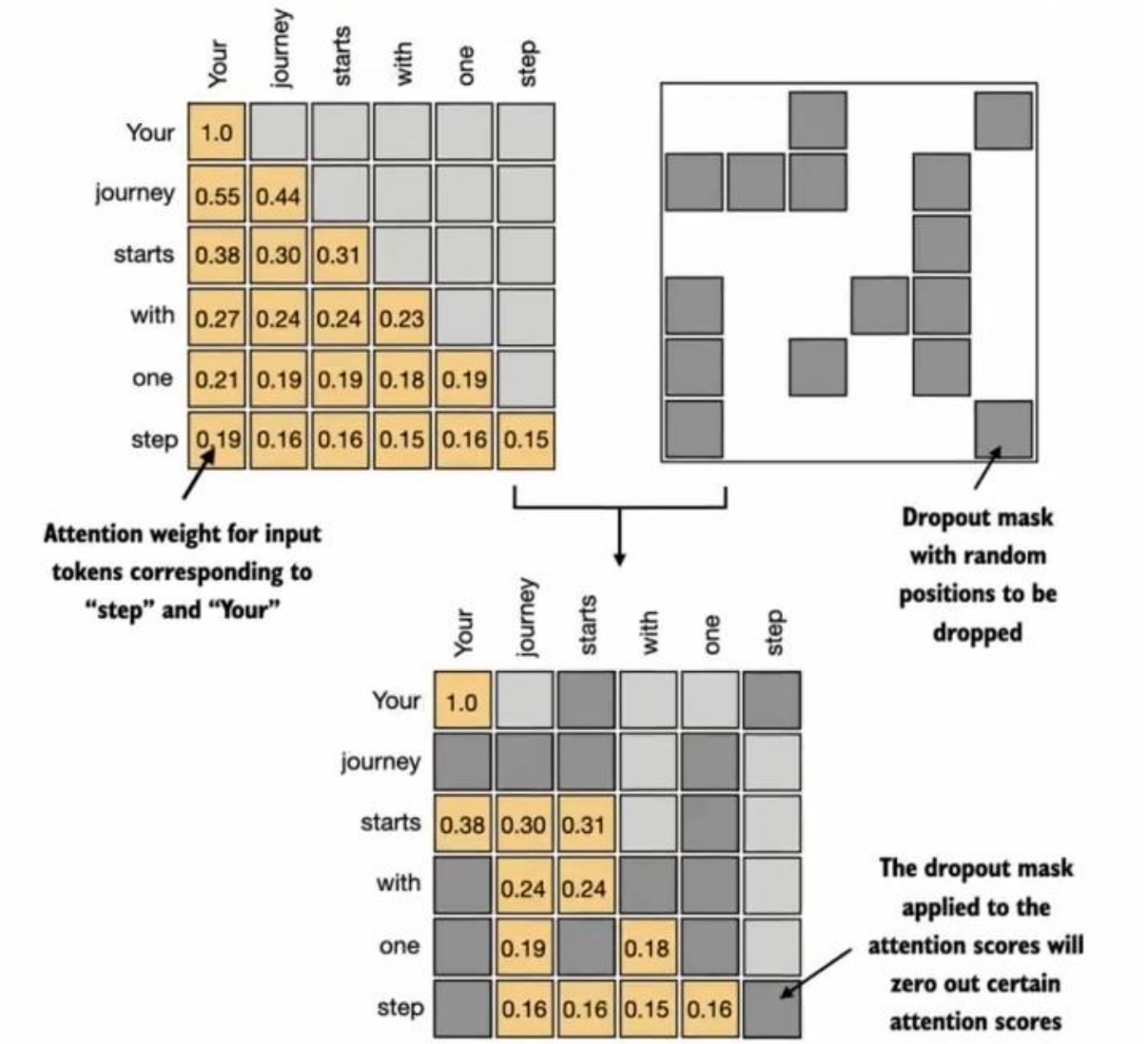
↳ In transformer architecture, including models like GPT, dropout in the attention mechanism is applied in 2 specific areas

after calculating attention scores

after applying attention weights to value vectors

Applying dropout after calculating attention weights is more common, and we consider that.





* Now, we can implement a Causal Attention class, which incorporates Causal Attention and Dropout modifications into the Self Attention class we implemented earlier.

MASKING ADDITIONAL ATTENTION WEIGHTS WITH DROPOUT

In the following code example, we use a dropout rate of 50%, which means masking out half of the attention weights.

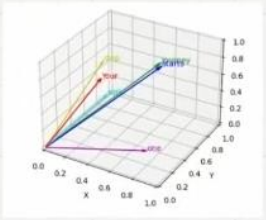
When we train the GPT model in later chapters, we will use a lower dropout rate, such as 0.1 or 0.2.

In the following code, we apply PyTorch's dropout implementation first to a 6×6 tensor consisting of ones for illustration

Inputs

Your (Journey) (starts) (with) (one) (step)

$$\begin{bmatrix} 0.43 & 0.15 & 0.89 \\ 0.55 & 0.87 & 0.66 \\ 0.57 & 0.85 & 0.64 \\ 0.22 & 0.58 & 0.33 \\ 0.77 & 0.25 & 0.10 \\ 0.05 & 0.80 & 0.55 \end{bmatrix}_{6\times3}$$



Inputs * W_q

$$W_q = \begin{bmatrix} 0.2961 & 0.5161 \\ 0.2517 & 0.6886 \\ 0.0740 & 0.8665 \end{bmatrix}_{3\times2}$$

Queries:

$$\begin{bmatrix} 0.23 & 1.09 \\ 0.43 & 1.45 \\ 0.43 & 1.43 \\ 0.23 & 0.79 \\ 0.29 & 0.65 \\ 0.25 & 1.05 \end{bmatrix}_{6\times2}$$

Inputs * W_k

$$W_k = \begin{bmatrix} 0.1366 & 0.1025 \\ 0.1841 & 0.7264 \\ 0.3153 & 0.6871 \end{bmatrix}_{3\times2}$$

Keys:

$$\begin{bmatrix} 0.36 & 0.76 \\ 0.44 & 1.14 \\ 0.43 & 1.11 \\ 0.24 & 0.67 \\ 0.18 & 0.32 \\ 0.32 & 0.96 \end{bmatrix}_{6\times2}$$

Inputs * W_v

$$W_v = \begin{bmatrix} 0.0756 & 0.1966 \\ 0.3164 & 0.4017 \\ 0.1186 & 0.8274 \end{bmatrix}_{3\times2}$$

Values:

$$\begin{bmatrix} 0.1855 & 0.8812 \\ 0.3951 & 1.00 \\ 0.38 & 0.98 \\ 0.23 & 0.54 \\ 0.14 & 0.33 \\ 0.32 & 0.78 \end{bmatrix}_{6\times2}$$

Batch 1

→ Queries * Keys^T → Scaling by $\sqrt{d\_keys}$ + Dropout + softmax → Attention weights * Values

```
tensor([[0.9231, 1.3545, 1.3241, 0.7910, 0.4032, 1.1330],
        [1.2705, 1.8524, 1.8111, 1.0795, 0.5577, 1.5440],
        [1.2544, 1.8284, 1.7877, 1.0654, 0.5508, 1.5238],
        [0.6973, 1.0167, 0.9941, 0.5925, 0.3061, 0.8475],
        [0.6114, 0.8819, 0.8626, 0.5121, 0.2707, 0.7307],
        [0.8995, 1.3165, 1.2871, 0.7682, 0.3937, 1.0996]])
```

6X6

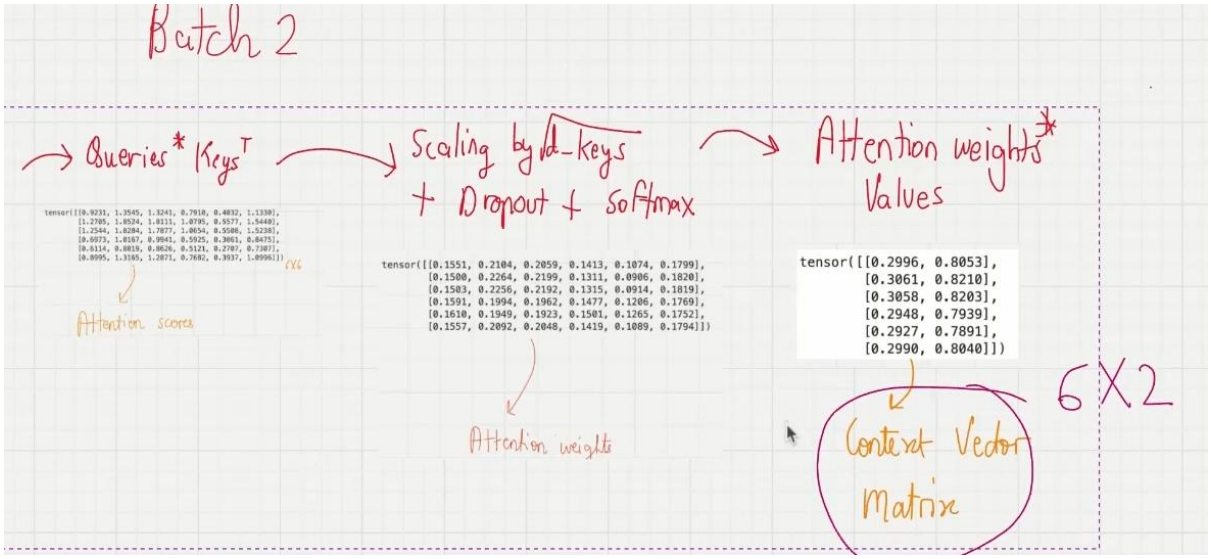
Attention scores

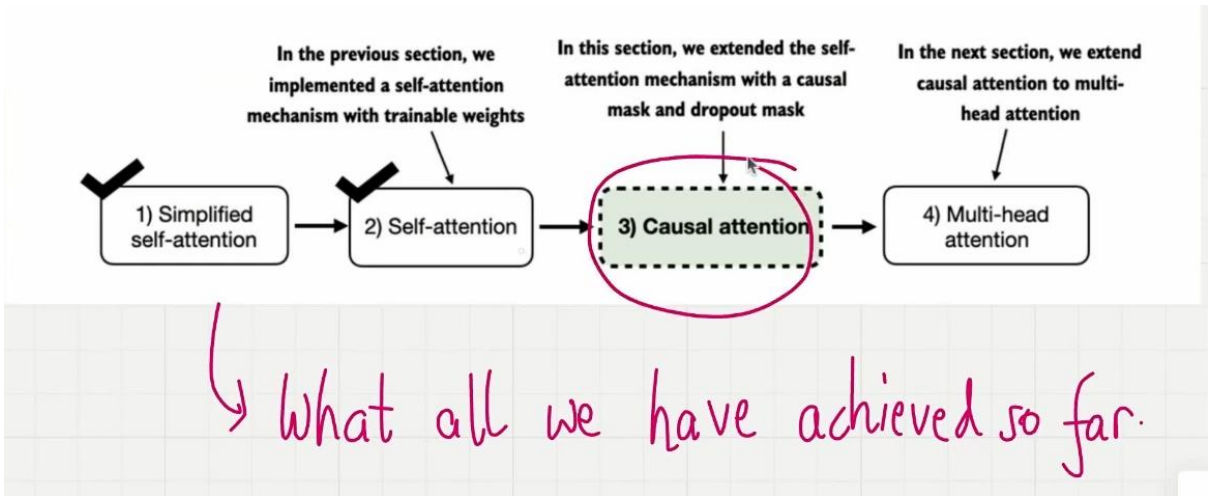
```
tensor([[0.1551, 0.2104, 0.2059, 0.1413, 0.1074, 0.1799],
        [0.1500, 0.2264, 0.2199, 0.1311, 0.0906, 0.1820],
        [0.1503, 0.2256, 0.2192, 0.1315, 0.0914, 0.1819],
        [0.1591, 0.1994, 0.1962, 0.1477, 0.1206, 0.1769],
        [0.1610, 0.1949, 0.1923, 0.1501, 0.1265, 0.1752],
        [0.1557, 0.2092, 0.2048, 0.1419, 0.1089, 0.1794]])
```

Attention weights

```
tensor([[0.2996, 0.8053],
        [0.3061, 0.8210],
        [0.3058, 0.8203],
        [0.2948, 0.7939],
        [0.2927, 0.7891],
        [0.2990, 0.8040]])
```

Context Vector Matrix





What all we have achieved so far.

Lecture → Multi-head attention mechanism

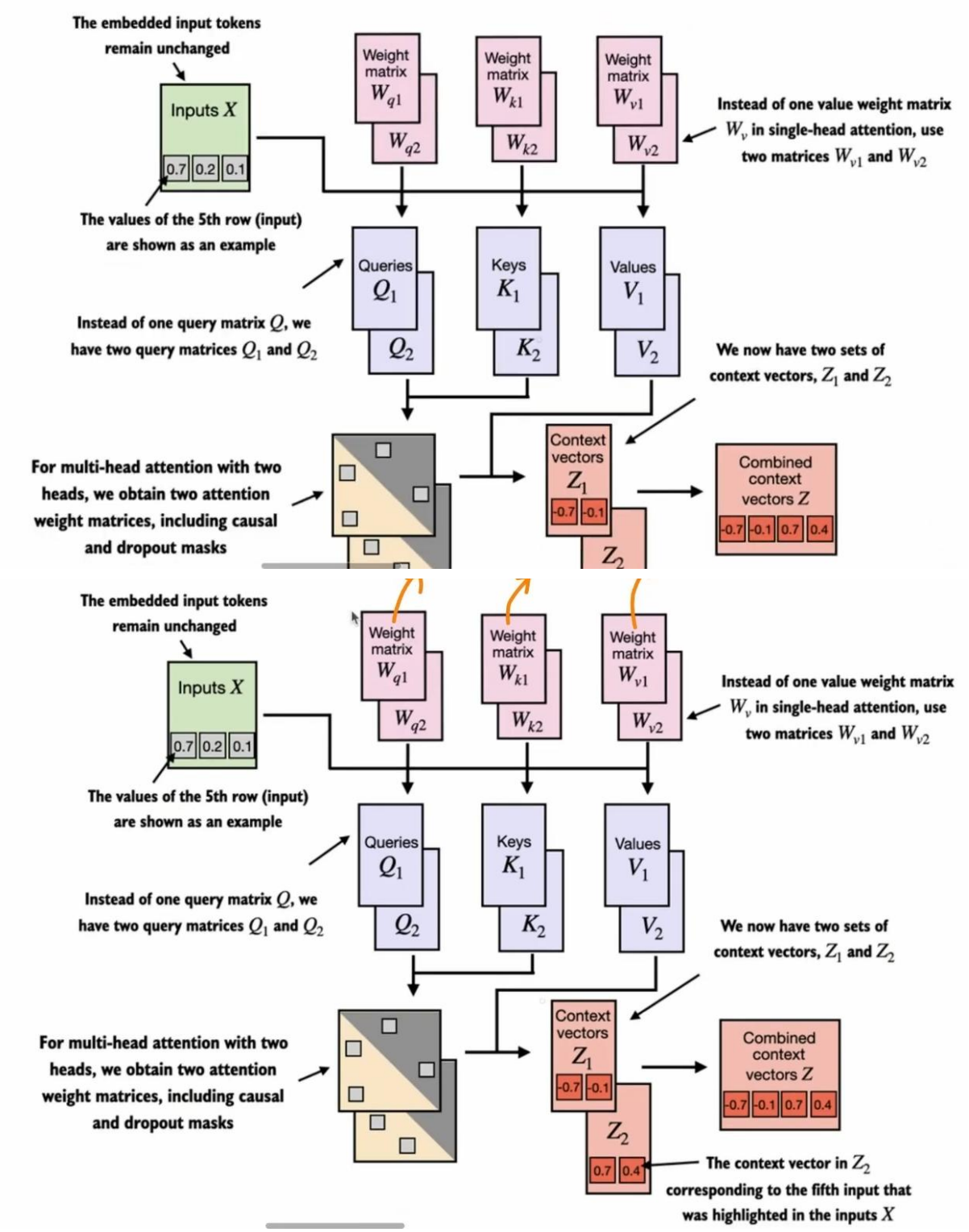
The term "multi-head" refers to dividing the attention mechanism into multiple heads.

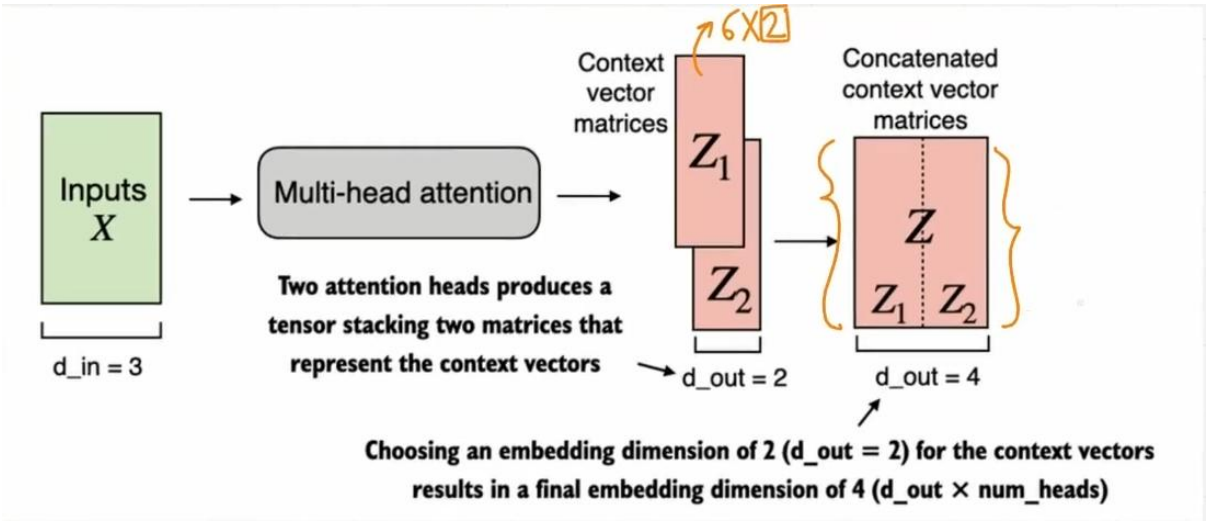
↳ each operating independently

(a) Stacking multiple single head attention layers

(1) Implementing multi-head attention involves creating multiple instances of the self-attention mechanism; each with it's own weights; and then combining their outputs.

(2) This can be computationally intensive, but it makes LLMs powerful at complex pattern recognition tasks.

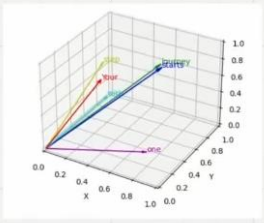

The embedded input tokens remain unchanged
Inputs X
0.7 0.2 0.1
The values of the 5th row (input) are shown as an example
Weight matrix W_{q1}
W_{q2}
Weight matrix W_{k1}
W_{k2}
Weight matrix W_{v1}
W_{v2}
Instead of one value weight matrix W_v in single-head attention, use two matrices W_{v1} and W_{v2}
Queries Q_1
Q_2
Keys K_1
K_2
Values V_1
V_2
Instead of one query matrix Q, we have two query matrices Q_1 and Q_2
We now have two sets of context vectors, Z_1 and Z_2
For multi-head attention with two heads, we obtain two attention weight matrices, including causal and dropout masks
Context vectors Z_1
-0.7 -0.1
Z_2
Combined context vectors Z
-0.7 -0.1 0.7 0.4
0.7 0.4
The context vector in Z_2 corresponding to the fifth input that was highlighted in the inputs X



↳ Main idea: Run the attention mechanism multiple times (in parallel) with different, learned linear projections: the results of multiplying input data (like query, key value vectors) by a weight matrix.

Your Journey starts with one step

$$\begin{bmatrix} 0.43 & 0.15 & 0.89 \\ 0.55 & 0.87 & 0.66 \\ 0.57 & 0.85 & 0.64 \\ 0.22 & 0.58 & 0.33 \\ 0.77 & 0.25 & 0.10 \\ 0.05 & 0.80 & 0.55 \end{bmatrix}_{6\times3}$$



$$\begin{bmatrix} 0.0740 & 0.8665 \end{bmatrix}_{3\times2} \qquad \begin{bmatrix} 0.25 & 1.05 \end{bmatrix}_{6\times2}$$

Inputs * W_k

$$W_k: \begin{bmatrix} 0.1366 & 0.1025 \\ 0.1841 & 0.7264 \\ 0.3153 & 0.6871 \end{bmatrix}_{3\times2}$$

Keys:

$$\begin{bmatrix} 0.36 & 0.76 \\ 0.44 & 1.14 \\ 0.43 & 1.11 \\ 0.24 & 0.67 \\ 0.18 & 0.32 \\ 0.32 & 0.96 \end{bmatrix}_{6\times2}$$

Inputs * W_v

$$W_v: \begin{bmatrix} 0.0756 & 0.1966 \\ 0.3164 & 0.4017 \\ 0.1186 & 0.8274 \end{bmatrix}_{3\times2}$$

Values:

$$\begin{bmatrix} 0.1855 & 0.8812 \\ 0.3951 & 1.00 \\ 0.38 & 0.98 \\ 0.23 & 0.54 \\ 0.14 & 0.33 \\ 0.32 & 0.78 \end{bmatrix}_{6\times2}$$

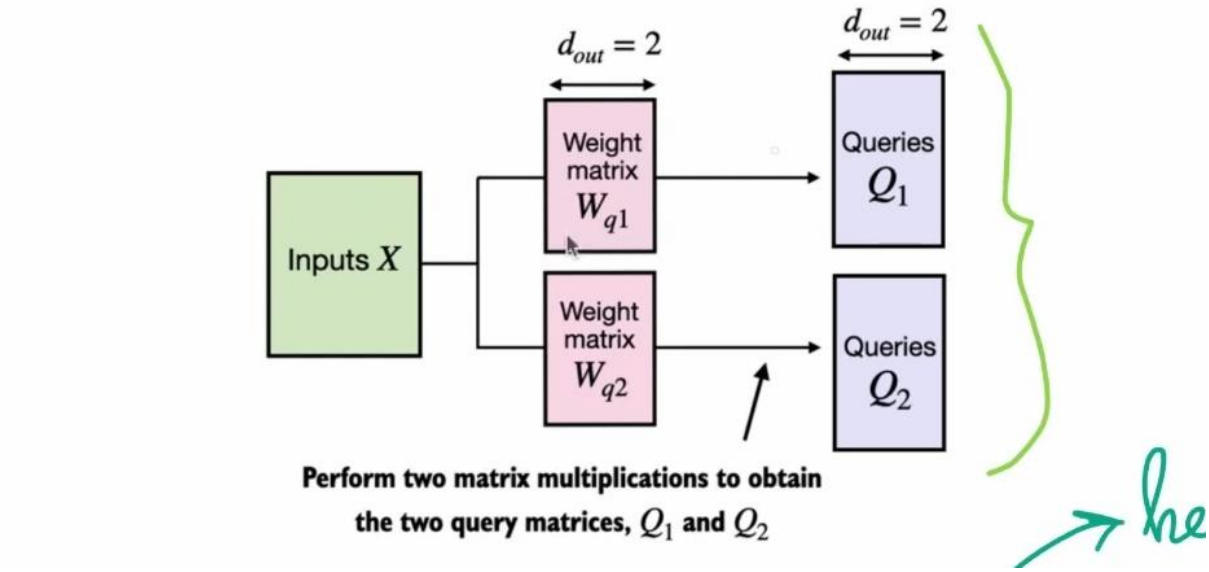
In multi-head attention, we have multiple W_q, W_k, W_v matrices

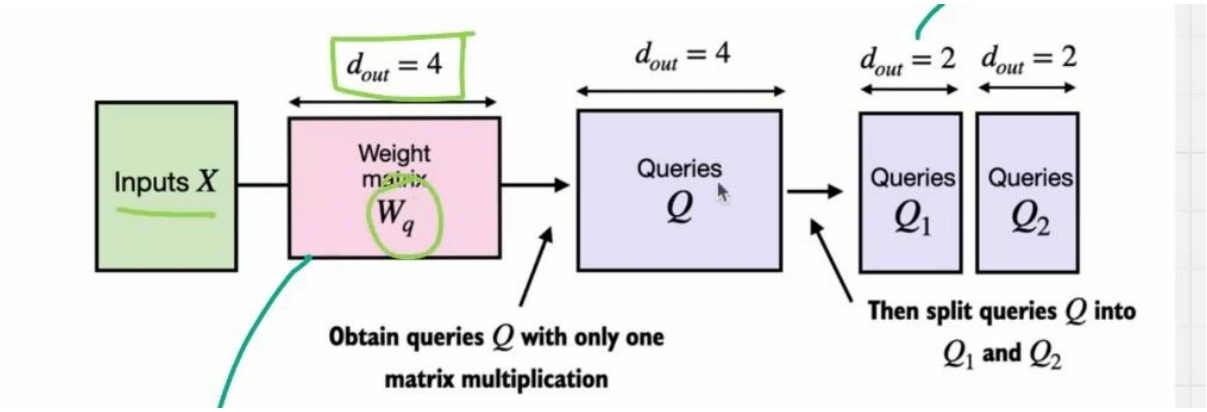
(b) Implementing multi-head attention with weight splits

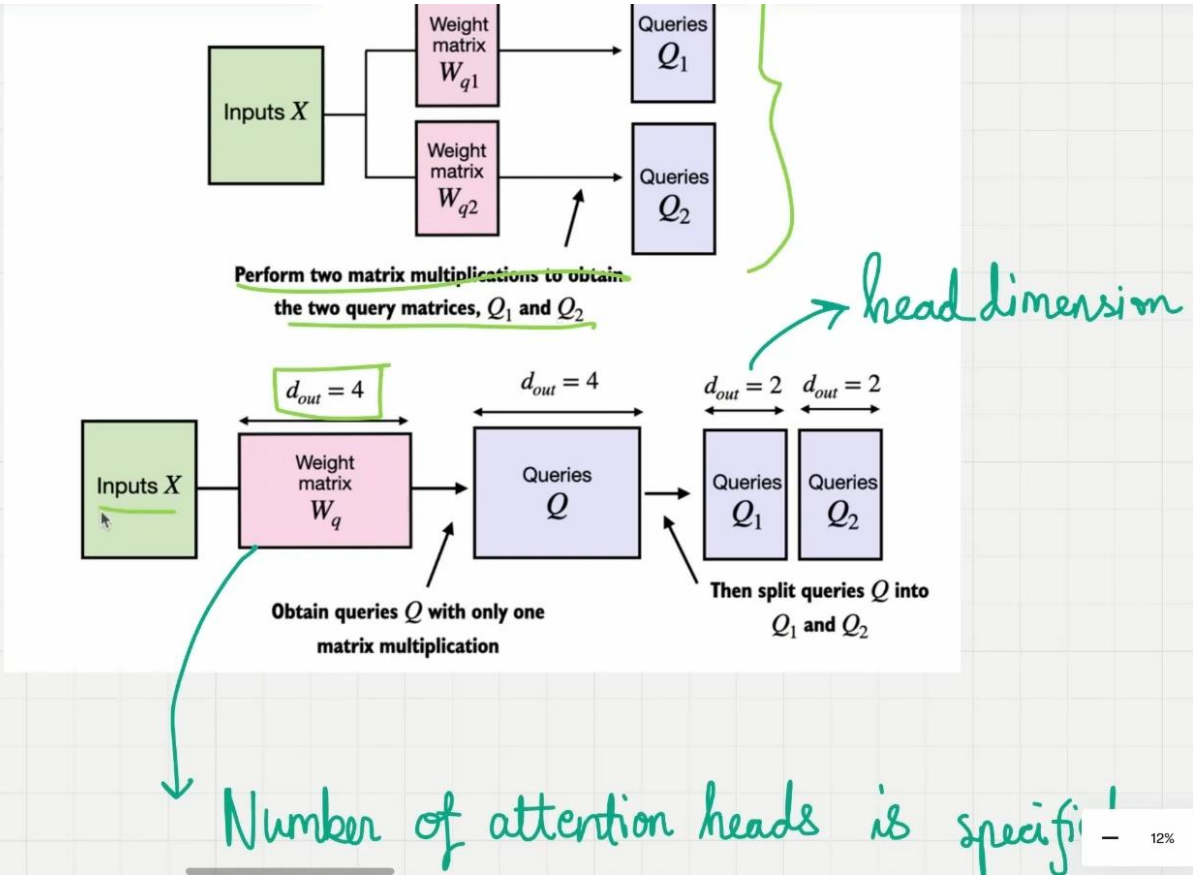
(1) Instead of maintaining 2 separate classes: MultiHeadAttentionWrapper and CausalAttention, we combined both of these into a single MultiHeadAttention class.

Implementing multi-head attention with weight splits

(1) Instead of maintaining 2 separate classes: MultiHeadAttentionWrapper and CausalAttention we combined both of these into a single MultiHeadAttention class.







12%

IMPLEMENTING MULTI-HEAD ATTENTION WITH WEIGHT SPLITS

Instead of maintaining two separate classes, MultiHeadAttentionWrapper and CausalAttention, we can combine both of these concepts into a single MultiHeadAttention class.

Also, in addition to just merging the MultiHeadAttentionWrapper with the CausalAttention code, we will make some other modifications to implement multi-head attention more efficiently.

In the MultiHeadAttentionWrapper, multiple heads are implemented by creating a list of CausalAttention objects (self.heads), each representing a separate attention head.

The CausalAttention class independently performs the attention mechanism, and the results from each head are concatenated.

In contrast, the following MultiHeadAttention class integrates the multi-head functionality within a single class.

It splits the input into multiple heads by reshaping the projected query, key, and value tensors and then combines the results from these heads after computing attention.

Number of attention heads is specified.

Head dimension = $\frac{d\_out}{n\_heads}$ (2)

* Let us take a simple example:

Step ①: Start with the input

b, num_tokens, d_in = (1, 3, 6)

the cat sleeps

```
x = torch.tensor([[[1.0, 2.0, 3.0, 4.0, 5.0, 6.0],
                   [6.0, 5.0, 4.0, 3.0, 2.0, 1.0],
                   [1.0, 1.0, 1.0, 1.0, 1.0, 1.0]]])
```

batch = 1,
num_tokens = , d_in = 6

Step ②: Decide d_out, num_heads

6 2

Step ②: Decide d_out, num_heads

6 2

head_dim = $\frac{6}{2} = 3$

Step ③: Initialize trainable weight matrices for key, query, value (W_k, W_q, W_v)

(6 × 6)

t

(6X6) → d_{in} X d_{out}

```
Wq = tensor([[ 0.6323, -0.2366,  1.2455,  0.3465,  1.2458,  0.3229],
        [ 0.6571, -0.2378, -0.5311, -0.2610, -1.4819, -1.6418],
        [-0.2990,  0.4216,  0.2114, -0.0271, -0.5682,  0.6937],
        [-1.1291, -1.0102,  0.6946,  0.1094,  0.5130, -0.8669],
        [ 0.3480,  0.2593,  0.4412,  1.0017, -0.3913, -0.2878],
        [ 0.2484,  0.2846, -0.3386, -0.6164,  1.2722,  0.5754]])
```

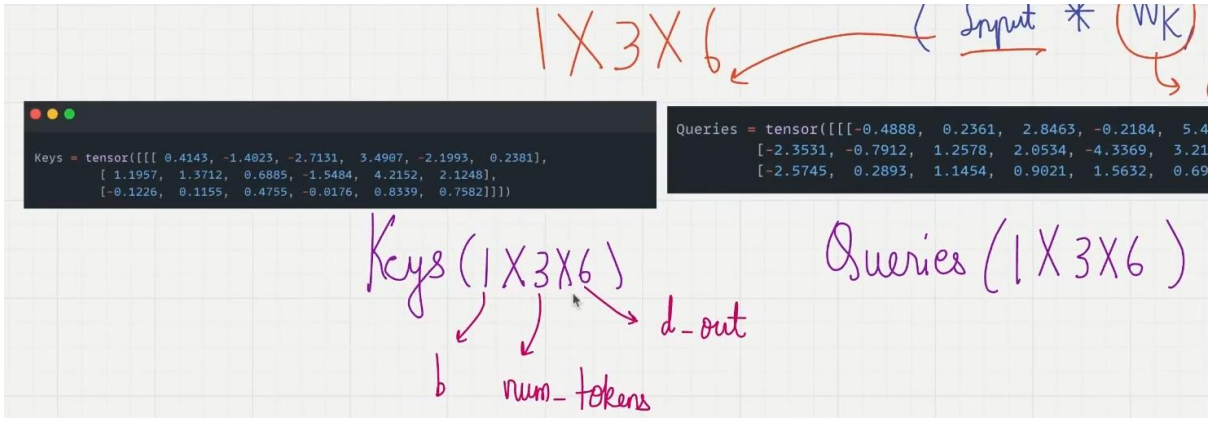
```
Wk = tensor([[-0.3703,  0.5431, -0.0372, -0.4406,  0.4103, -0.1773],
        [ 1.5993, -0.2777, -1.1909, -0.4301,  0.6927, -1.3304],
        [ 1.2470, -0.1872, -0.1670,  1.4302,  1.2927,  0.4822],
        [-0.0984, -0.8983,  0.3334, -0.6312,  0.1022, -1.0715],
        [-0.7647,  0.1734,  0.6305,  1.0155,  0.8474,  0.1454],
        [-1.5085, -0.4529,  0.0997, -0.1084,  0.8046,  0.3459]])
```

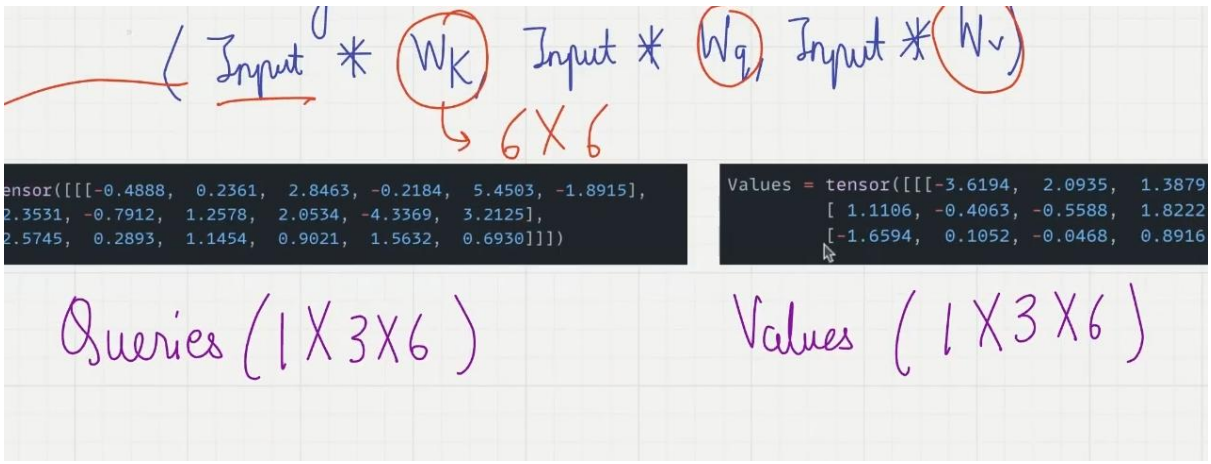
(6X6)

```
Wv = tensor([[ 1.6395,  1.1234, -0.1001,  0.5021, -1.0590,  0.1412
        [-0.4271,  0.5681,  0.4164, -1.2534,  1.3061,  0.3610],
        [-0.2824, -0.4314,  1.2358,  0.1181, -1.2467,  0.1893],
        [ 1.3440,  0.1487, -0.6174,  0.8890, -0.3282,  1.4662],
        [ 0.1814, -0.4761, -0.0402,  0.7326,  0.7654, -0.1080],
        [-0.8974,  0.6786,  0.5602, -0.2443, -0.4883,  1.3996]])
```

Step ④: Calculate Keys, Queries, Value Matrix

(Input * W_K, Input * W_q, Input * W_v)





Step (5): Unroll last dimension of Keys, Queries and Values to include num_heads and head_dim

head_dim = d_out / num_heads = 6/2 = 3

(b, num_tokens, d_out) → (b, num_tokens, num_heads, head_dim)

(1, 3, 6) → (1, 3, 2, 3)

2

(b, num_tokens, d_out) → (b, num_tokens, num_heads, head_dim)

(1, 3, 6) → (1, 3, 2, 3)

```
Reshaped Queries matrix:
tensor([[[[-0.4888,  0.2361,  2.8463],
          [-0.2184,  5.4503, -1.8915]],

         [[-2.3531, -0.7912,  1.2578],
          [ 2.0534, -4.3369,  3.2125]],

         [[-2.5745,  0.2893,  1.1454],
          [ 0.9021,  1.5632,  0.6930]]]], dtype=torch.float64)
```

(1, 3, 2, 3)

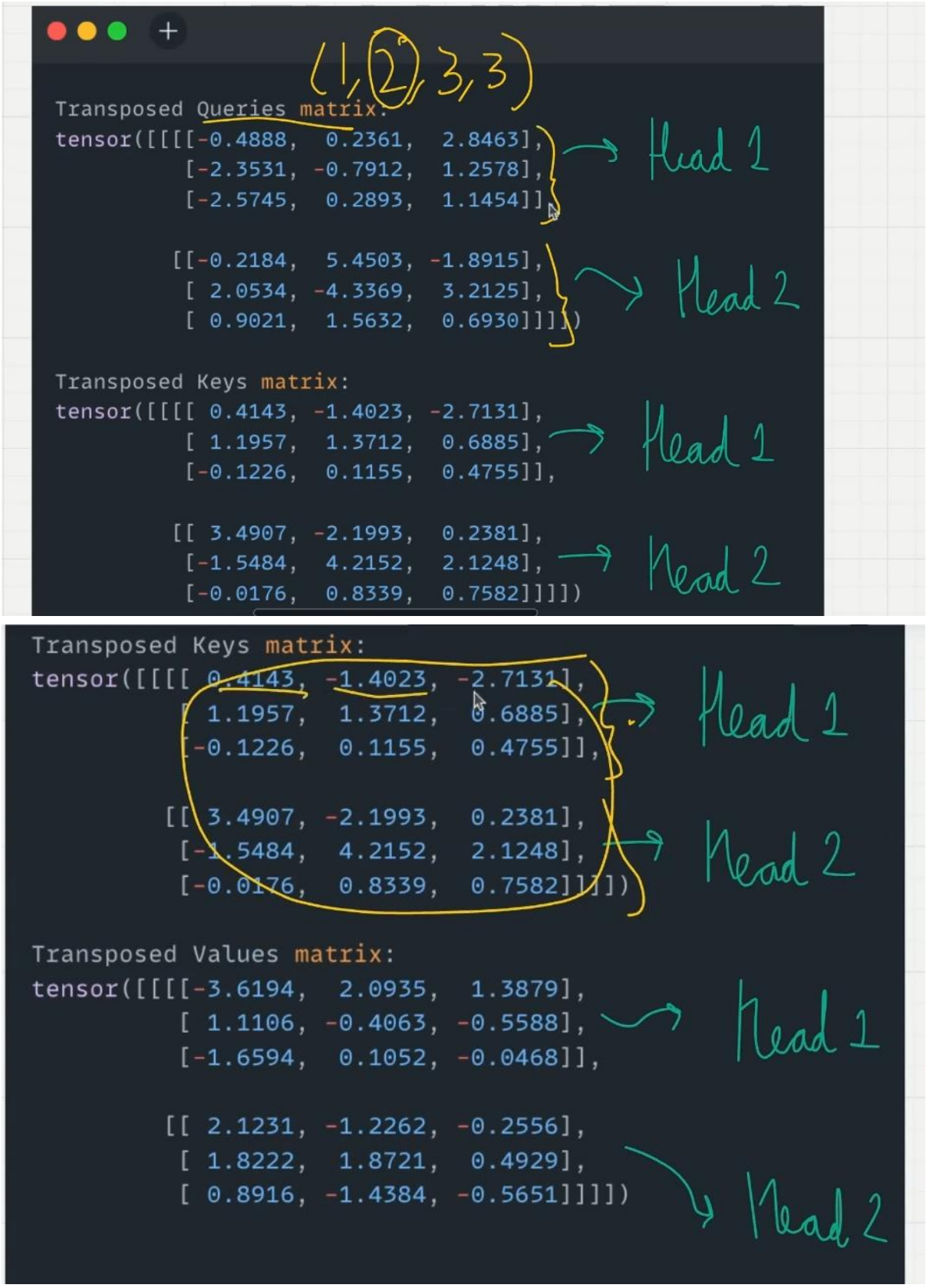
```
Reshaped Keys matrix:
tensor([[[[ 0.4143, -1.4023, -2.7131],
          [ 3.4907, -2.1993,  0.2381]],

         [[ 1.1957,  1.3712,  0.6885],
          [-1.5484,  4.2152,  2.1248]],

         [[-0.1226,  0.1155,  0.4755],
          [-0.0176,  0.8339,  0.7582]]]], dtype=torch.float64)

Reshaped Values matrix:
tensor([[[[-3.6194,  2.0935,  1.3879],
          [ 2.1231, -1.2262, -0.2556]],
```

(1, 3, 2, 3)

Step ⑥: Group matrices by "number of heads"

(b, num_tokens, num_heads, head_dim) → (b, num_heads, num_tokens, head_d

(1, 3, 2, 3) → (1, 2, 3, 3)


(1, 2, 3, 3)
Transposed Queries matrix:
tensor([[[[-0.4888, 0.2361, 2.8463],
[-2.3531, -0.7912, 1.2578],
[-2.5745, 0.2893, 1.1454]],
[[-0.2184, 5.4503, -1.8915],
[2.0534, -4.3369, 3.2125],
[0.9021, 1.5632, 0.6930]]]])
Head 1
Head 2
Transposed Keys matrix:
tensor([[[[0.4143, -1.4023, -2.7131],
[1.1957, 1.3712, 0.6885],
[-0.1226, 0.1155, 0.4755]],
[[3.4907, -2.1993, 0.2381],
[-1.5484, 4.2152, 2.1248],
[-0.0176, 0.8339, 0.7582]]]])
Head 1
Head 2

Transposed Keys matrix:
tensor([[[[0.4143, -1.4023, -2.7131],
[1.1957, 1.3712, 0.6885],
[-0.1226, 0.1155, 0.4755]],
[[3.4907, -2.1993, 0.2381],
[-1.5484, 4.2152, 2.1248],
[-0.0176, 0.8339, 0.7582]]]])
Head 1
Head 2
Transposed Values matrix:
tensor([[[[-3.6194, 2.0935, 1.3879],
[1.1106, -0.4063, -0.5588],
[-1.6594, 0.1052, -0.0468]],
[[2.1231, -1.2262, -0.2556],
[1.8222, 1.8721, 0.4929],
[0.8916, -1.4384, -0.5651]]]])
Head 1
Head 2

Step (7): Find attention scores:

Queries * Keys.Transpose (2,3)

Queries * Keys.Transpose (2,3)

```
Queries matrix:

tensor([[[[-0.4888,  0.2361,  2.8463],
          [-2.3531, -0.7912,  1.2578],
          [-2.5745,  0.2893,  1.1454]],

         [[-0.2184,  5.4503, -1.8915],
          [ 2.0534, -4.3369,  3.2125],
          [ 0.9021,  1.5632,  0.6930]]]])
```

Head 1

Head 2

```
Keys.transpose(2, 3) =

tensor([[[[ 0.4143,  3.4907, -0.1226],
          [-1.4023, -2.1993,  0.1155],
          [-2.7131,  0.2381,  0.4755]],

         [[ 1.1957, -1.5484, -0.0176],
          [ 1.3712,  4.2152,  0.8339],
          [ 0.6885,  2.1248,  0.7582]]]])
```

Head 1

Head 2

(1, 2, 3, 3) × (1, 2, 3, 3)

(b, num_heads, num_tokens, head_dim) (b, num_heads, head_dim, num_tokens)

Attention scores

```
queries * keys.transpose(2, 3):
tensor([[[[-8.2252,  2.0863,  1.2960],
          [-1.3722, -7.0720, -1.4290],
          [-5.8961, -2.7236, -1.0160]],

         [[ 4.6567, -7.1312,  4.7274],
          [-1.3167,  1.3964, -0.6018],
          [ 2.3820,  2.7213,  0.8448]]]])
```

(1, 2, 3, 3)

(b, num_heads, num_tokens, num_tokens)

Step (8): Find attention weights

↓

Mask attention scores to implement causal attention

```
tensor([[[[-8.2252,    -inf,    -inf],
          [-1.3722,  -7.0720,    -inf],
          [-5.8961,  -2.7236,  -1.0160]],

         [[ 4.6567,    -inf,    -inf],
          [-1.3167,   1.3964,    -inf],
          [ 2.3820,   2.7213,   0.8448]]]])
```

Divide by $\sqrt{\text{head\_dim}} = \sqrt{\text{d\_out}/\text{num\_heads}} = \sqrt{6/2} = \sqrt{3}$

```
tensor([[[[-4.7496,    -inf,    -inf],
          [-0.7920,  -4.0838,    -inf],
          [-3.4033,  -1.5727,  -0.5862]],

         [[ 2.6880,    -inf,    -inf],
          [-0.7603,   0.8060,    -inf],
          [ 1.3757,   1.5708,   0.4877]]]])
```

↓ Apply softmax

```
tensor([[[[1.0000, 0.0000, 0.0000],
          [0.9641, 0.0359, 0.0000],
          [0.0417, 0.2603, 0.6980]],

         [[1.0000, 0.0000, 0.0000],
          [0.1727, 0.8273, 0.0000],
          [0.3807, 0.4627, 0.1566]]]])
```

Attention weights
(1, 2, 3, 3)
(b, num-heads, num_tokens

Attention weights
(1, 2, 3, 3)
(b, num-heads, num_tokens, num_tokens)

↓ We can also implement Dropout after this.

Context Vector = Attention Weights * Values

(b, num_heads, num_tokens, num_tokens)

(b, num_heads, num_tokens, head_dim)

```
# Final attention scores
attn_scores = torch.tensor([[[[1.0000, 0.0000, 0.0000],
                              [0.9641, 0.0359, 0.0000],
                              [0.0417, 0.2603, 0.6980]],

                             [[1.0000, 0.0000, 0.0000],
                              [0.1727, 0.8273, 0.0000],
                              [0.3807, 0.4627, 0.1566]]]])
```

(1, 2, 3, 3)

*

```
values = torch.tensor([[[[-3.6194,  2.0935,  1.3879],
                         [ 1.1106, -0.4063, -0.5588],
                         [-1.6594,  0.1052, -0.0468]],

                        [[ 2.1231, -1.2262, -0.2556],
                         [ 1.8222,  1.8721,  0.4929],
                         [ 0.8916, -1.4384, -0.5651]]]])
```

(1, 2, 3, 3)

```
Context vectors:
tensor([[[[ -3.6194,   2.0935,   1.3879],
          [ -1.8728,   1.6494,   0.7163],
          [ -0.3553,  -0.5607,  -0.1181]],

         [[  1.5961,  -0.9367,  -0.3400],
          [  1.3712,   0.7805,   0.2233],
          [  0.6997,  -0.4487,  -0.0530]]]])
```

(1, 2, 3, 3)

(b, num_heads, num_tokens, head_dim)

Step (10) Reformat context vectors:

(b, num_heads, num_tokens, head_dim) → (b, num_tokens, num_heads, head_dim)

```
tensor([[[[-3.6194,  2.0935,  1.3879],
          [ 1.5961, -0.9367, -0.3400]],

         [[-1.8728,  1.6494,  0.7163],
          [ 1.3712,  0.7805,  0.2233]],

         [[-0.3553, -0.5607, -0.1181],
          [ 0.6997, -0.4487, -0.0530]]]])
```

→ (1, 3, 2, 3)

(b, num_tokens, num_heads, head_dim)

Step (11): Combine heads:

Flatten each

```
tensor([[[[-3.6194,  2.0935,  1.3879],
          [ 1.5961, -0.9367, -0.3400]],

         [[-1.8728,  1.6494,  0.7163],
          [ 1.3712,  0.7805,  0.2233]],

         [[-0.3553, -0.5607, -0.1181],
          [ 0.6997, -0.4487, -0.0530]]]])
```

Token 1

Token 2

Token 3

(b, num_tokens, num_heads, head_dim)

(1, 3, 2, 3)

6%

Flatten each token output into each row

Token 1

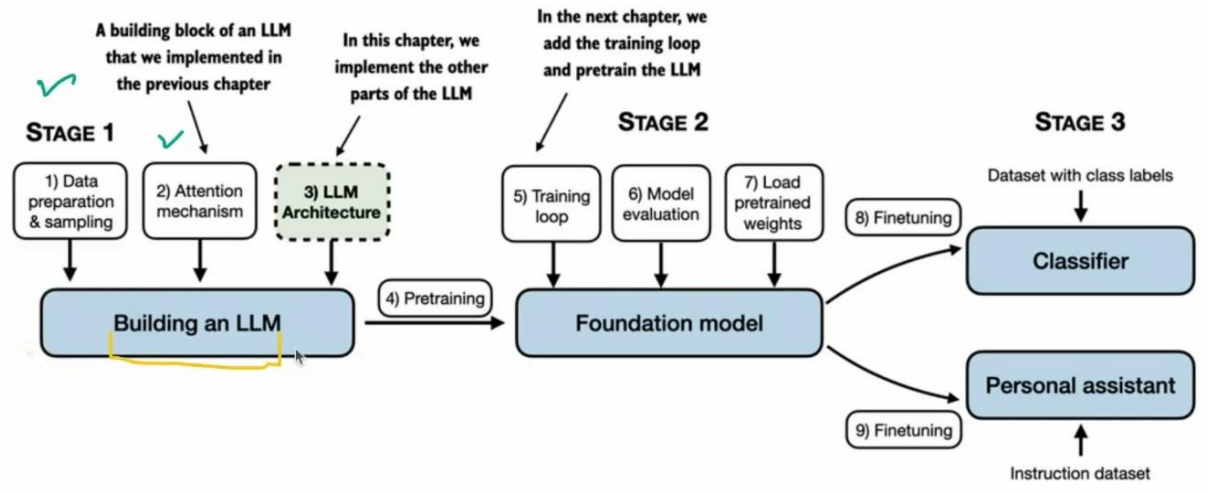
Token 2

Token 3

```
tensor([[[-3.6194,  2.0935,  1.3879,  1.5961, -0.9367, -0.3400],
         [-1.8728,  1.6494,  0.7163,  1.3712,  0.7805,  0.2233],
         [-0.3553, -0.5607, -0.1181,  0.6997, -0.4487, -0.0530]]])
```

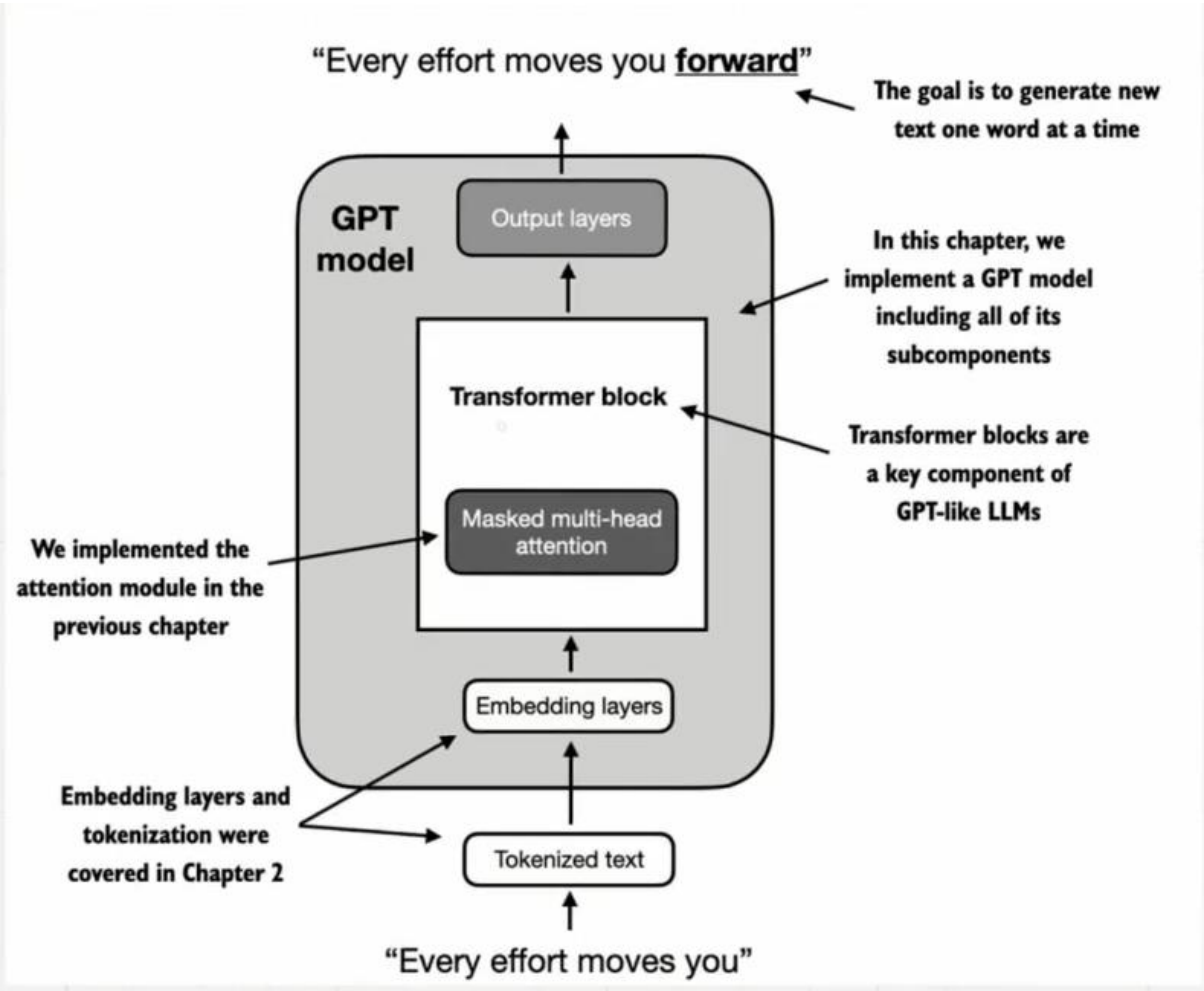
(b, num_tokens, d_out)

(1, 3, 6)

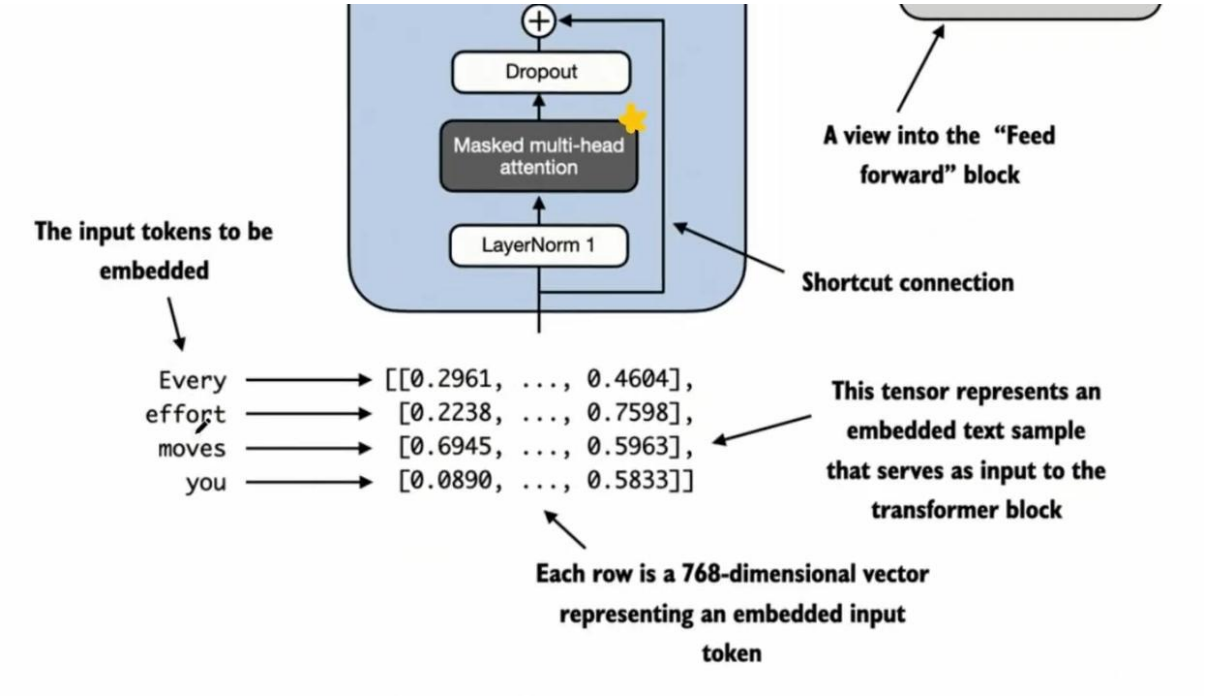


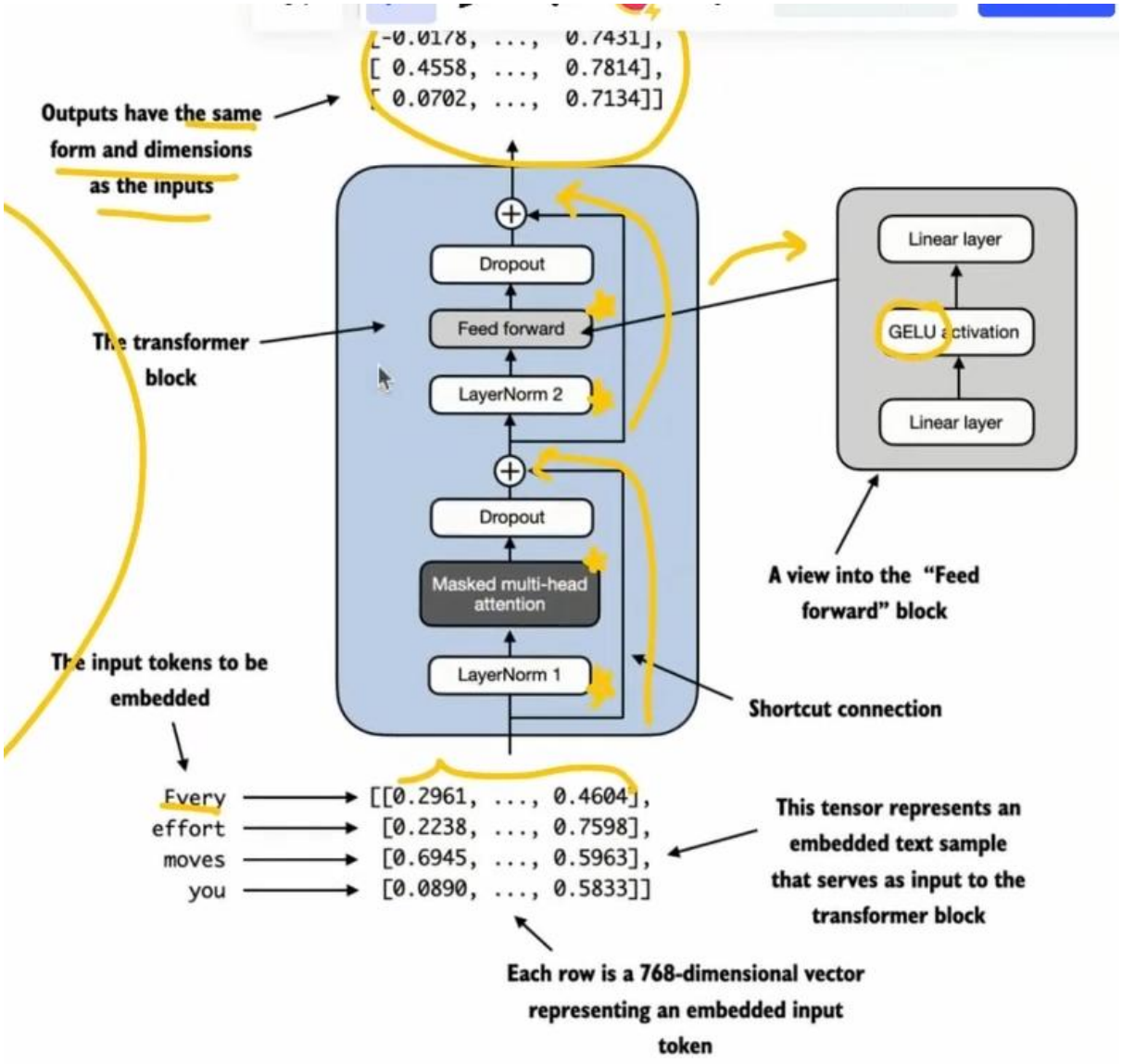
After learning about the attention mechanism, let us learn about the LLM architecture now.

) Coding an LLM architecture:



Zoom In Transformer Block


Dropout
Masked multi-head attention
LayerNorm 1
A view into the "Feed forward" block
The input tokens to be embedded
Shortcut connection
Every → [[0.2961, ..., 0.4604],
effort → [0.2238, ..., 0.7598],
moves → [0.6945, ..., 0.5963],
you → [0.0890, ..., 0.5833]]
This tensor represents an embedded text sample that serves as input to the transformer block
Each row is a 768-dimensional vector representing an embedded input token

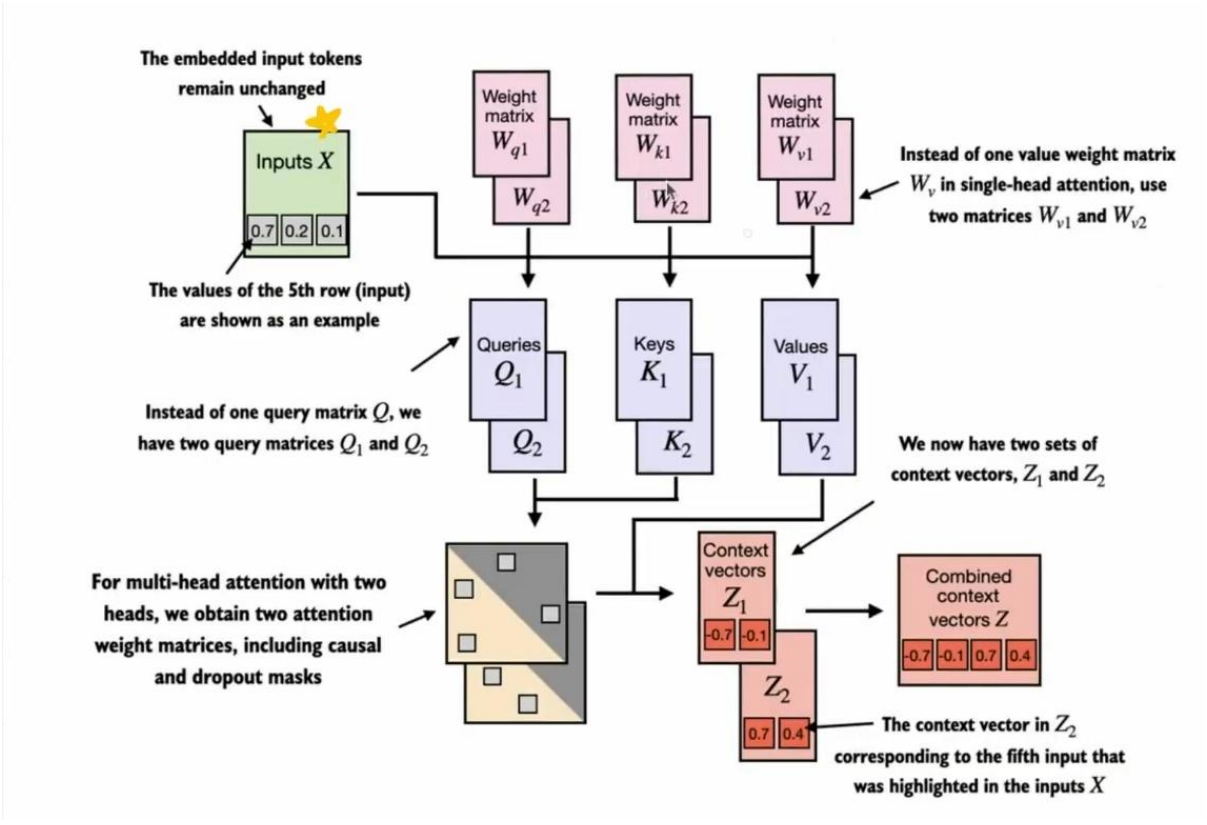


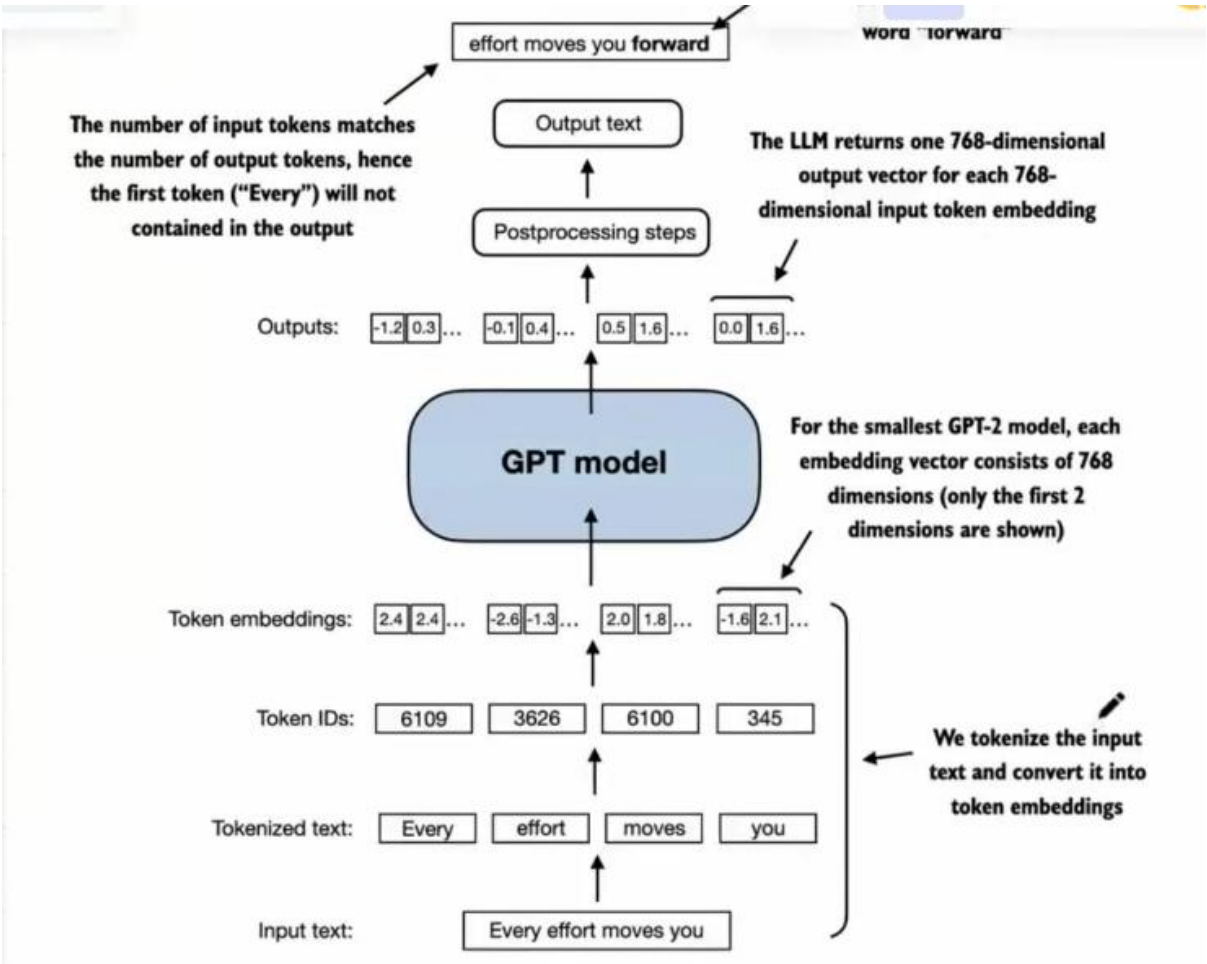
(2) What we have learnt so far:

(a) Input tokenization

(b) Embedding (Token + Positional)

(c) Masked multi-head attention


The embedded input tokens remain unchanged
Inputs X
0.7 0.2 0.1
The values of the 5th row (input) are shown as an example
Weight matrix W_{q1}
W_{q2}
Weight matrix W_{k1}
W_{k2}
Weight matrix W_{v1}
W_{v2}
Instead of one value weight matrix W_v in single-head attention, use two matrices W_{v1} and W_{v2}
Queries Q_1
Q_2
Keys K_1
K_2
Values V_1
V_2
Instead of one query matrix Q, we have two query matrices Q_1 and Q_2
We now have two sets of context vectors, Z_1 and Z_2
For multi-head attention with two heads, we obtain two attention weight matrices, including causal and dropout masks
Context vectors Z_1
-0.7 -0.1
Z_2
0.7 0.4
Combined context vectors Z
-0.7 -0.1 0.7 0.4
The context vector in Z_2 corresponding to the fifth input that was highlighted in the inputs X


effort moves you forward
The number of input tokens matches the number of output tokens, hence the first token ("Every") will not contained in the output
Output text
Postprocessing steps
The LLM returns one 768-dimensional output vector for each 768-dimensional input token embedding
Outputs: -1.2 0.3 ... -0.1 0.4 ... 0.5 1.6 ... 0.0 1.6 ...
GPT model
For the smallest GPT-2 model, each embedding vector consists of 768 dimensions (only the first 2 dimensions are shown)
Token embeddings: 2.4 2.4 ... -2.6 -1.3 ... 2.0 1.8 ... -1.6 2.1 ...
Token IDs: 6109 3626 6100 345
Tokenized text: Every effort moves you
Input text: Every effort moves you
We tokenize the input text and convert it into token embeddings

③ What we are yet to learn:

ⓐ Transformer blocks

④ We will scale upto the size of a small GPT-2 model → 124 million parameters

---

Language Models are Unsupervised Multitask Learners

Parameters	Layers	d_{model}
117M	12	768
345M	24	1024
762M	36	1280
1542M	48	1600

↗ This was corrected to 124 later

⑤ Open AI has made GPT-2 weights public.

GPT-3,4 weights have not yet been made public.

→ We will use these paramet

GPT_CONFIG_124M = {