The Transformer encoder *visualized*

By Ulf Mertens

In this tutorial, I will try to visualise and explain the calculations performed under the hood of the Transformer encoder used for masked language modeling, i.e. predicting missing words. In order to do so, the dimensions of the tensors involved will be much lower than in real applications.

This tutorial should be seen as an add-on to the tutorials that have already been written. Be aware that this is not meant as an extensive introduction into all the details but rather yet another view that (I hope) sheds even more light on the inner workings.

We will start with only three sentences. For each sentence, one or more words will get masked. The Transformers task then is to predict these missing words within each sentence.

Input representation

1) She likes the summer better

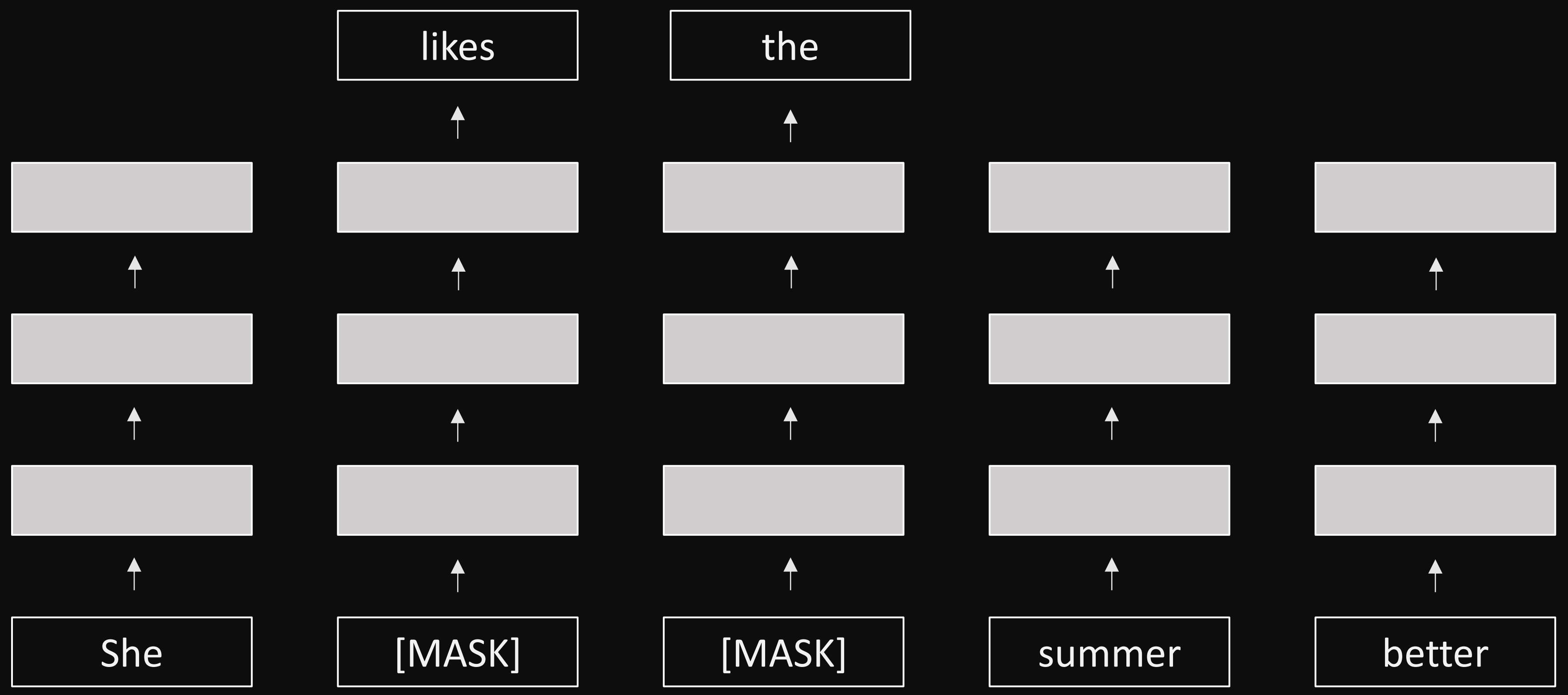
2) He loves to cook healthy

3) They go hiking quite often

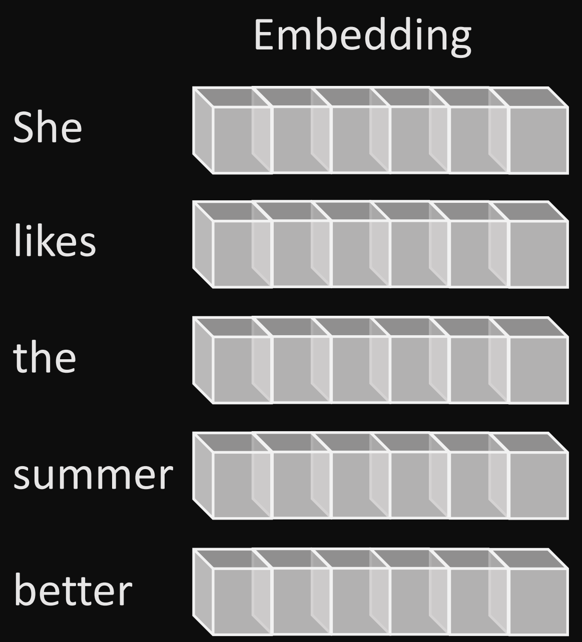
1) She **[MASK]** **[MASK]** summer better

2) He loves to **[MASK]** healthy

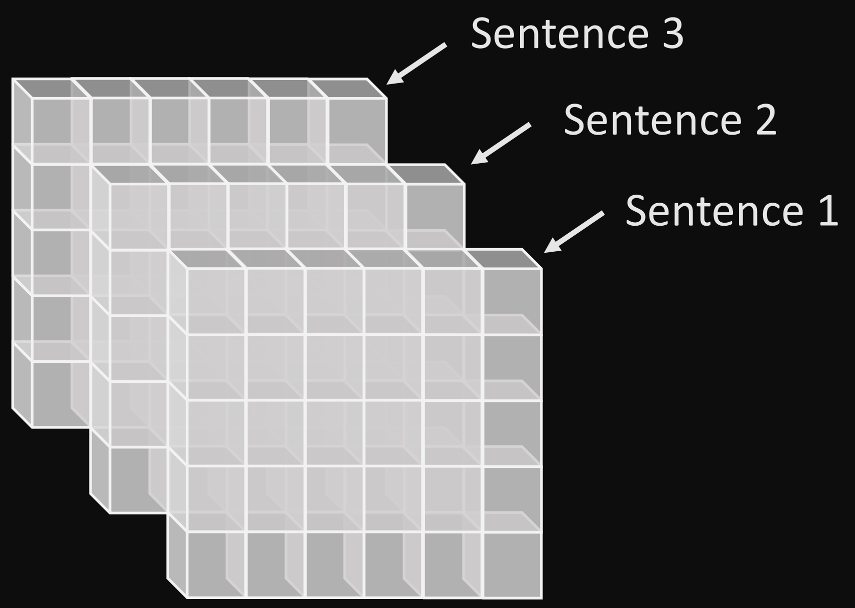
3) They go hiking quite **[MASK]**


likes
the
She
[MASK]
[MASK]
summer
better

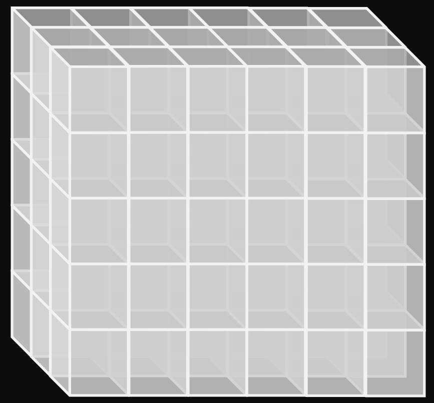
Each of the words will be replaced with a higher-dimensional representation (embedding). We set the dimensionality of the embedding to six. Let’s now take a look at the tensor we are working with…



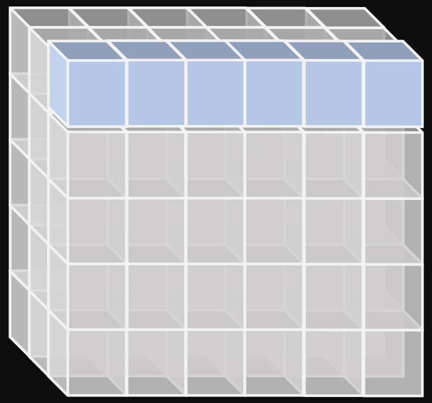
The representations of all three sentences are combined into a single 3d tensor



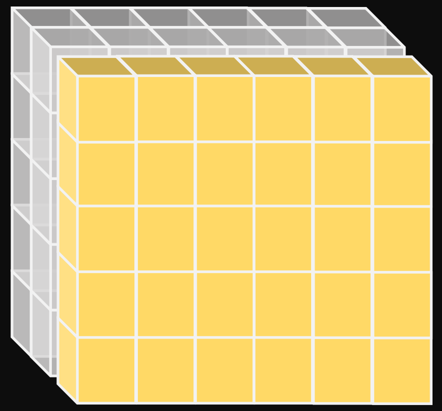
This is a 3d tensor representation of the three sentences. The depth corresponds to the number of sentences, i.e. the batch size, the height is the maximum number of words in each sentence and the width equals the embedding dimensionality.



The highlighted area corresponds to the **word representation of the first word in the first sentence** → *She*

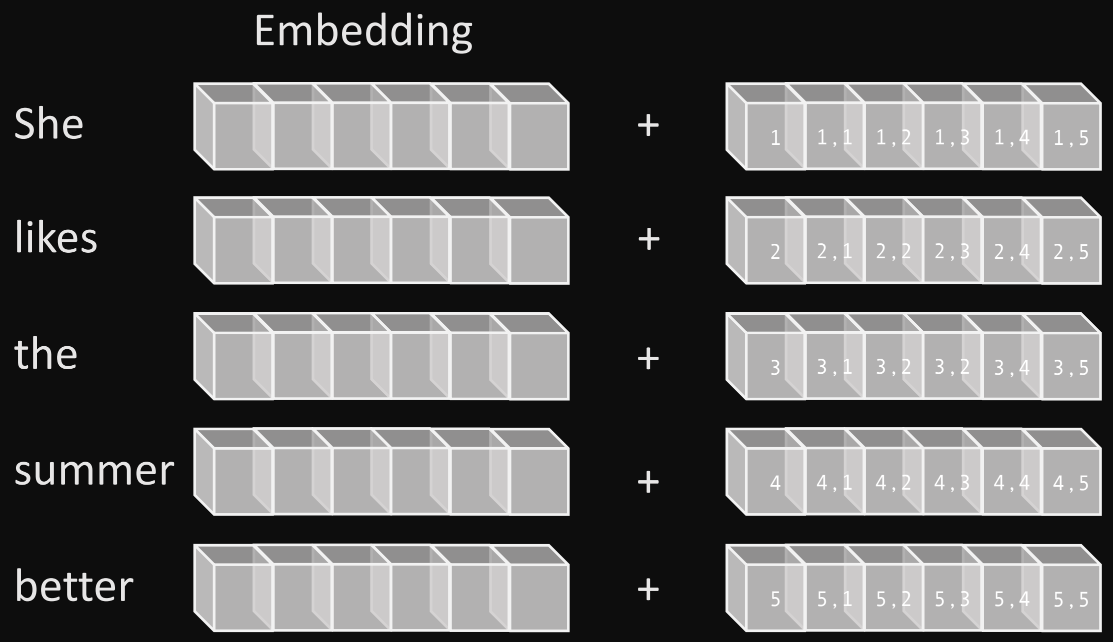


The highlighted area corresponds to the **word representations of the first sentence** → *She likes the summer better*

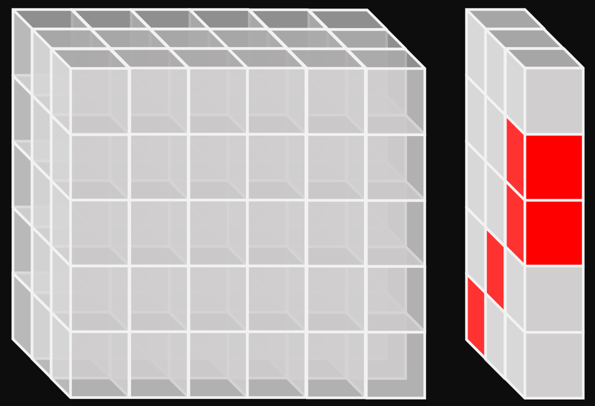


Positional encoding

The sequence of words within a sentence is important for any NLP task. However, the Transformer architecture by itself does not care about the actual order of words, i.e. you could switch words as you like and the final result would stay the same. To circumvent this permutation invariance, a tensor representing the position of each word and also the position of each scalar within the embedding is added to the actual embedding. Note that the numbers on the right do not resemble the numbers that are actually used but should only illustrate the idea in general.

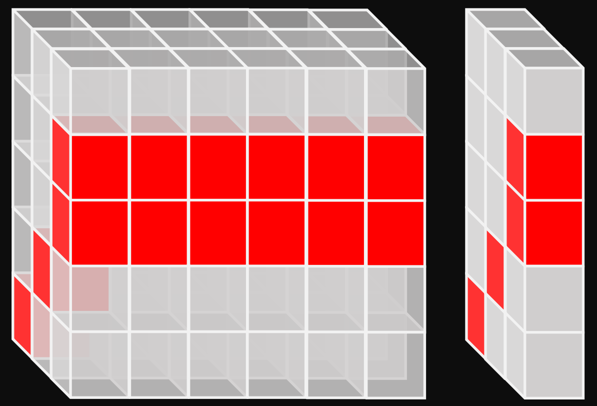


Since our goal is to predict the masked words in each sentence, we should ensure to not present those words to the model. The calculations done within the Transformer are not allowed to process the masked words. A (boolean) mask tensor is needed for this. We see how it is applied in later steps but you need to make sure that the embedding vectors of masked words are set to all zeros.



Since our goal is to predict the masked words in each sentence, we should ensure to not present those words to the model. The calculations done within the Transformer are not allowed to process the masked words. A (boolean) mask tensor is needed for this. We see how it is applied in later steps but you need to make sure that the embedding vectors of masked words are set to all zeros.

The cubes shown in red are the masked words (represented e.g. as zeros) whereas the grey cubes are words we have access to (represented as e.g. ones)



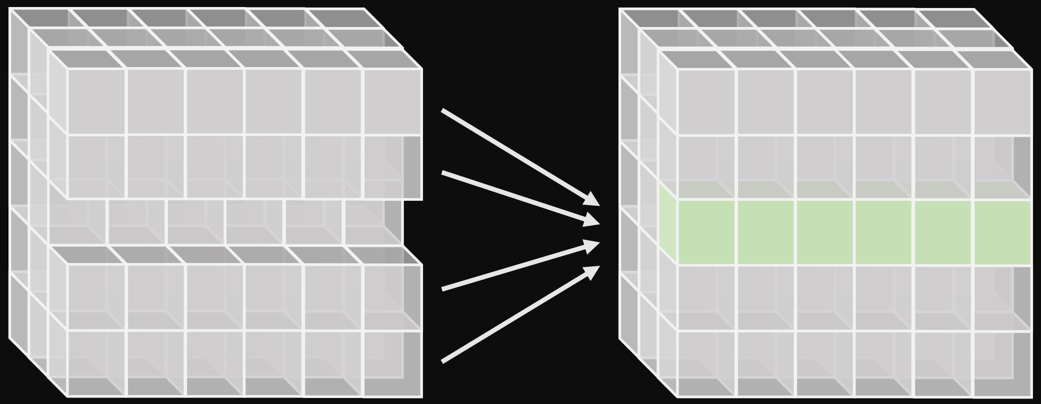
Step 0
What do we want to achieve?

Loosely speaking,the overall goal is to *transform* our words represented by embeddings into embeddings that are aware of their soroundings/context.

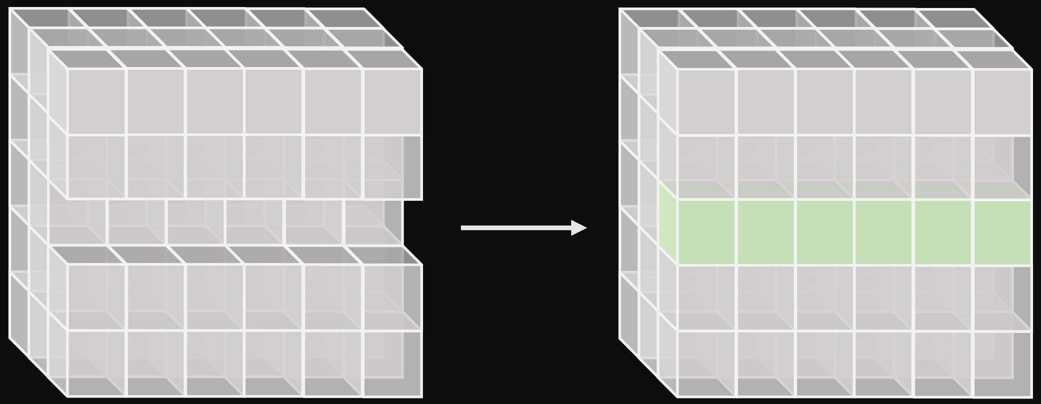
The final embedding of a masked word contains knowledge about what the other words in the sentence are.

For an intuitive understanding, you can think of the resulting embeddings (as the one shown in green) as a weighted combination of the other embeddings in the sentence.

Note that the input embedding is missing on the right since it is a masked word and we do not want access to the respective embedding.

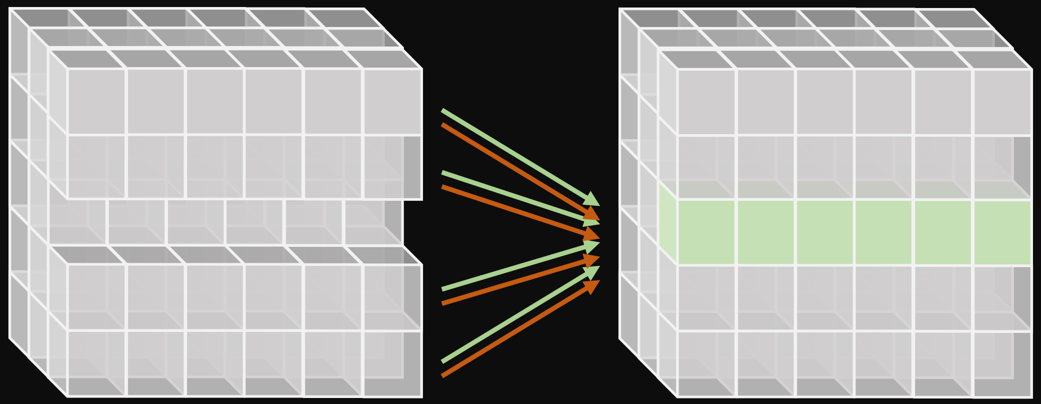


The Transformer learns how to weight each word in a sentence, in order to make the best guess about each missing word. Put differently, for the prediction of a masked word, it can learn how much *attention* to pay to each other word.

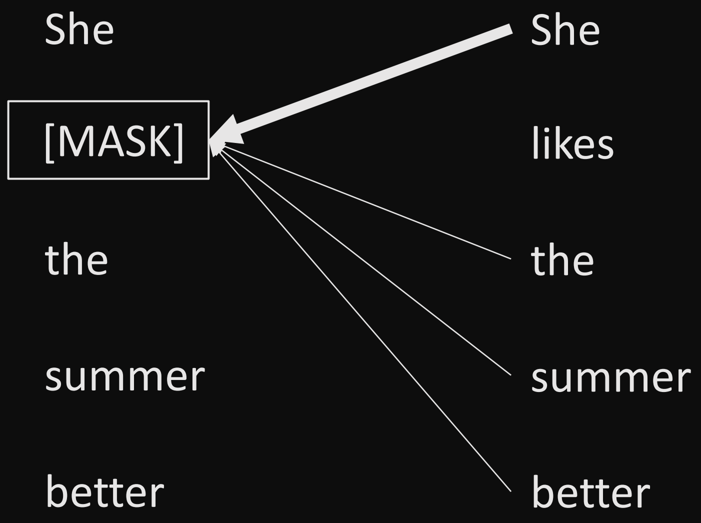


Sometimes, it might be beneficial to not be restricted to weight each word only once. To make a good guess about a missing word, the Transformer chooses in one so-called *attention head* a specific set of weights, whereas another *attention head* might choose a different set of weights.

Depending on the application, several such *attention heads* might be applied. Let’s be more concrete on the next slide…

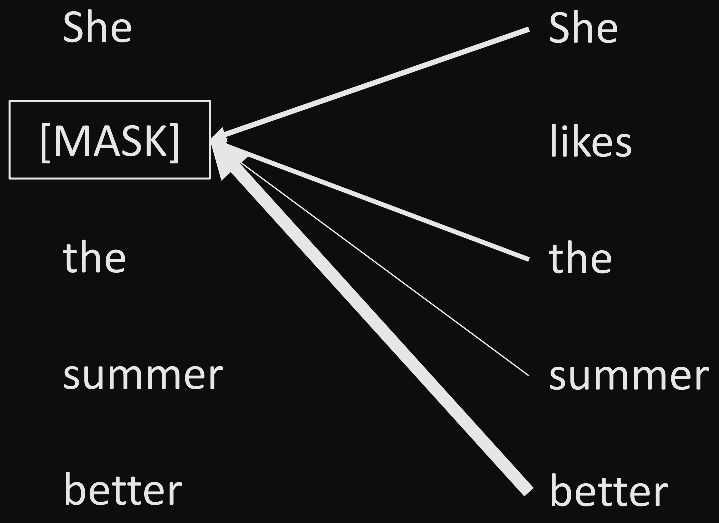


Head I



Putting most attention to ***She***

Head II



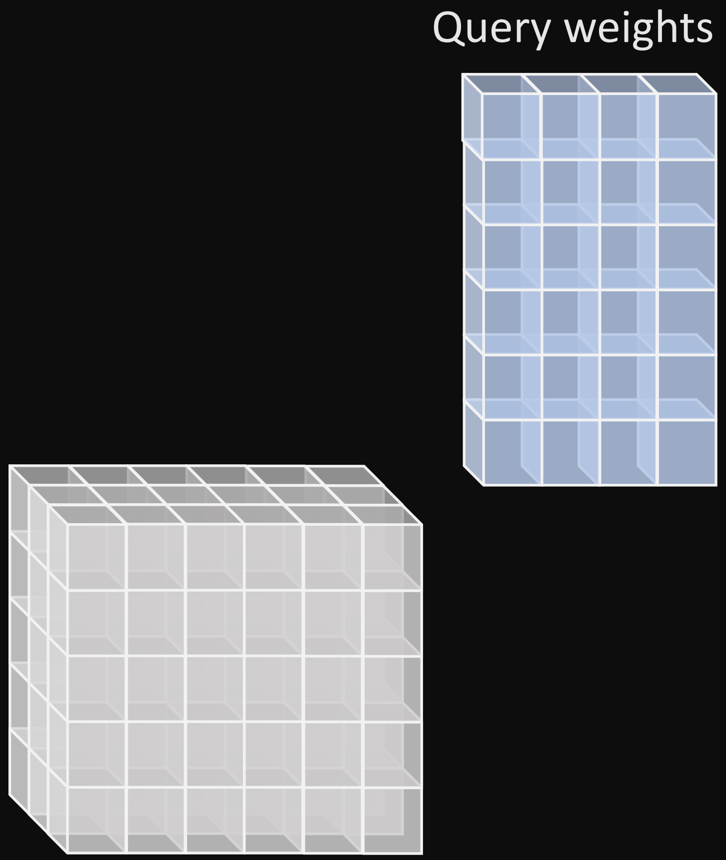
Putting most attention to ***better***

Step I

Generating attention weights

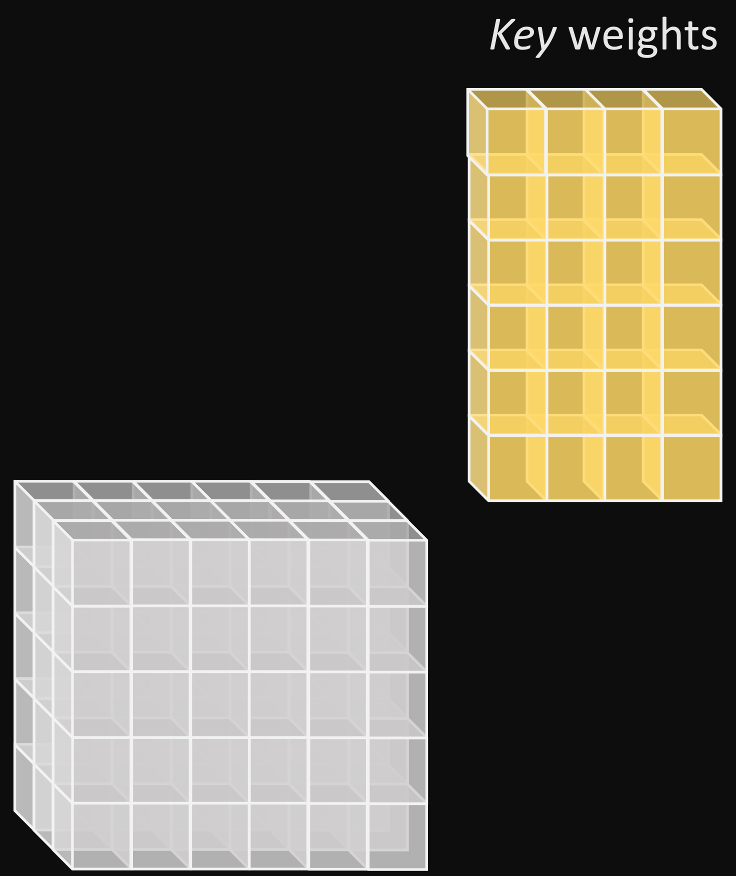
In principle, for each word (token) in a sentence, we would like to get a score of all remaining words in the sentence (including the word itself). So we need a way to compare each word with all others.

Instead of comparing the embedding vectors directly, the embeddings are first transformed using two linear layers. One such transformation generates the *query* tensor, the other transformation (with different weights) leads to the so-called *key* tensor. On the right, the *query* weight tensor is shown. We set the *query* dimensionality to 4.



In principle, for each word (token) in a sentence, we would like to get a score of all remaining words in the sentence (including the word itself). So we need a way to compare each word with all others.

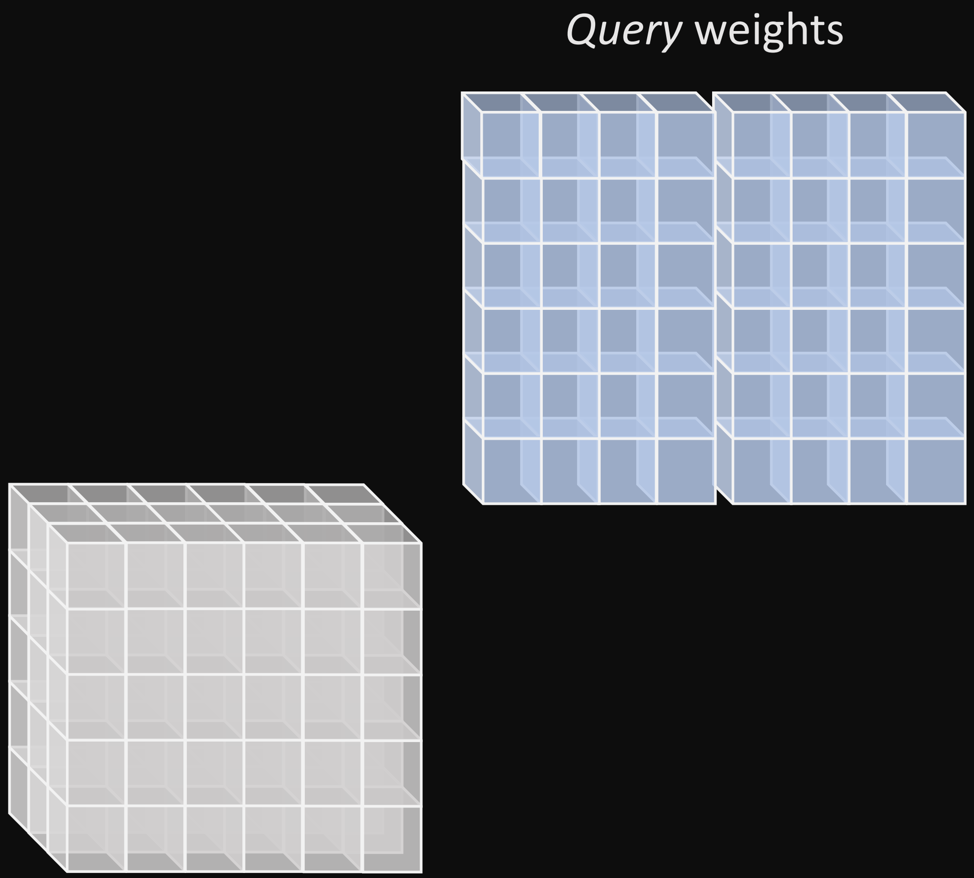
Instead of comparing the embedding vectors directly, the embeddings are first transformed using two linear layers. One such transformation generates the *query* tensor, the other transformation (with different weights) leads to the so-called *key* tensor. On the right, the *key* weight tensor is shown. We set the *query/key* dimensionality to 4.



As shown before, it might be good to have not only one score for each word, but multiple scores (*multi-head attention*). To get another set of scores ,we simply concatenate another weight tensor as shown on the right side.

Both *query* and *key* weight tensor therefore consist 6 x (4 x 2) weights.

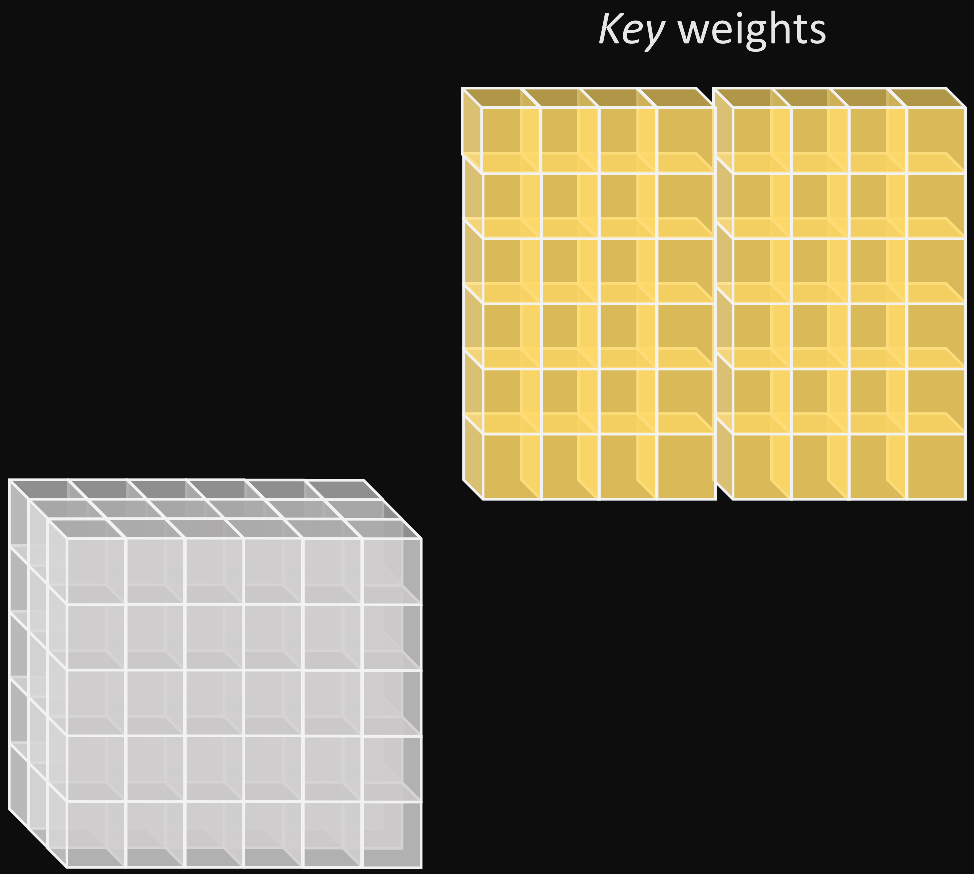
The multiplication of *embedding tensor* and weight tensor is possible by expanding the weight tensor in batch dimension (i.e. broadcasting).

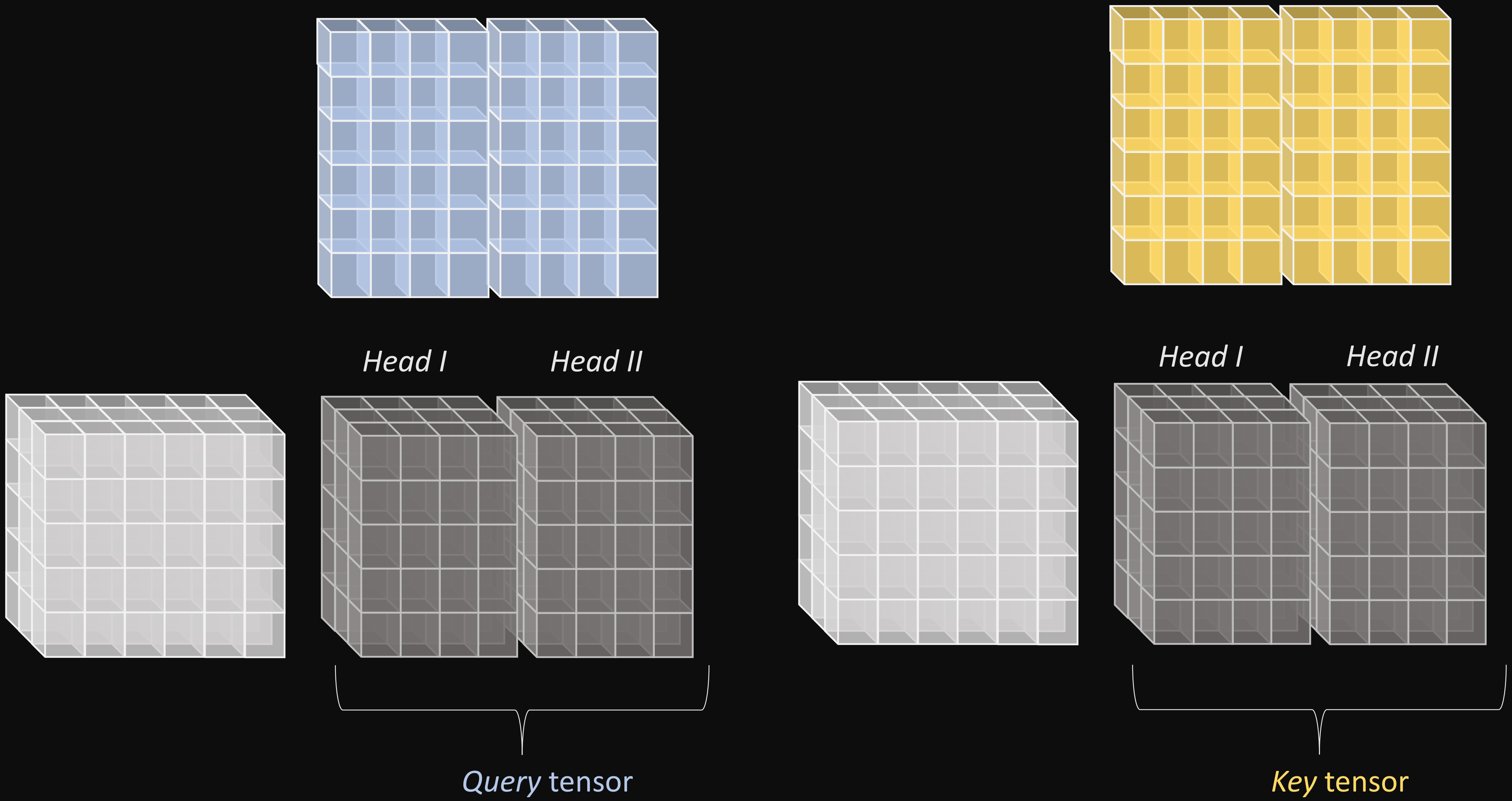


As shown before, it might be good to have not only one score for each word, but multiple scores (*multi-head attention*). To get another set of scores ,we simply concatenate another weight tensor as shown on the right side.

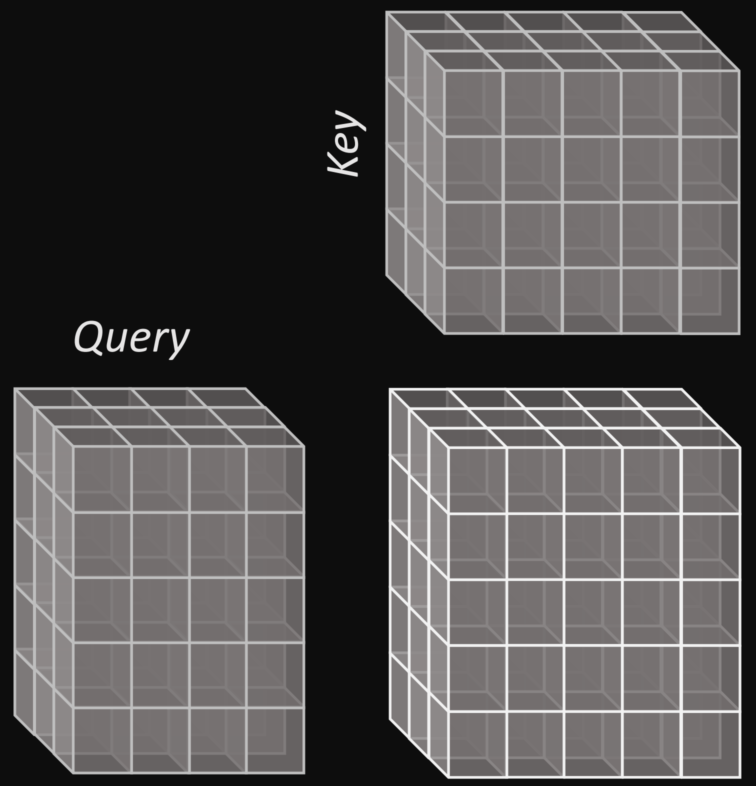
Both *query* and *key* weight tensor therefore consist 6 x (4 x 2) weights.

The multiplication of *embedding tensor* and weight tensor is possible by expanding the weight tensor in batch dimension (i.e. broadcasting)

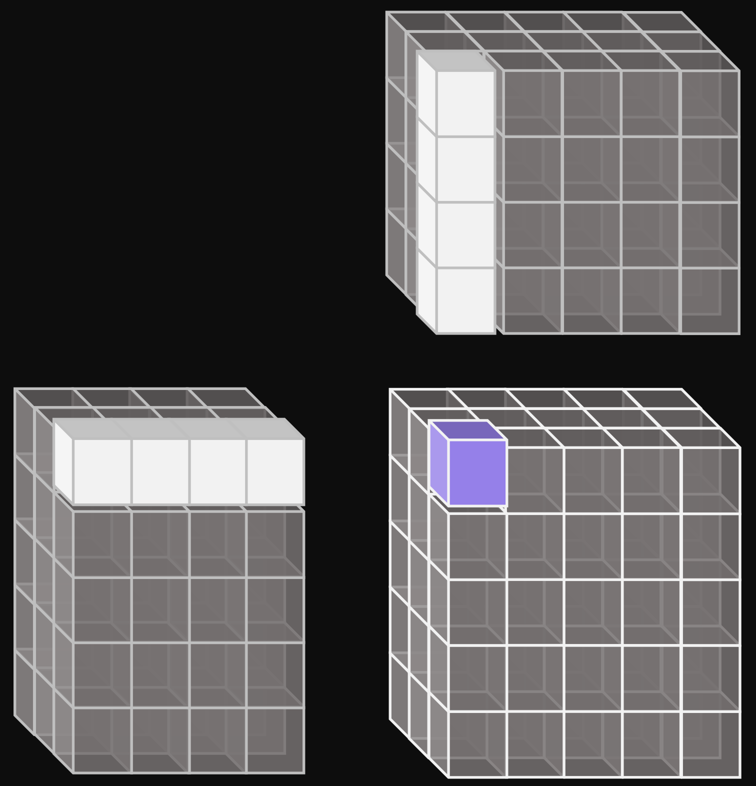



Head I
Head II
Query tensor
Head I
Head II
Key tensor

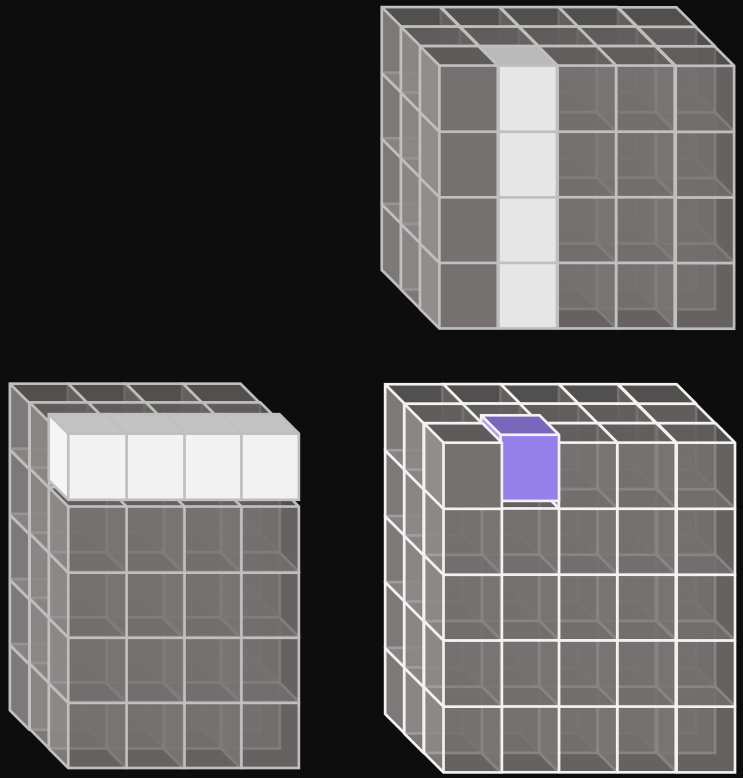
The way the attention scores are computed is via dot products. Shown on the right, we compute the first set of attention scores using only *Head I* of both *query* and *key*.



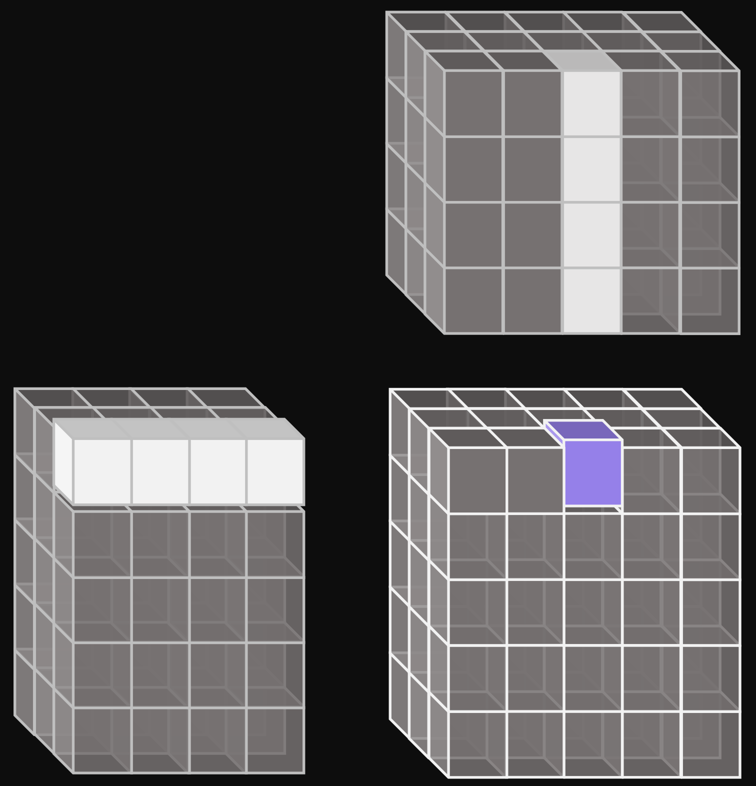
Computing the similarity (score) between query of the first word with key of the **first** word.



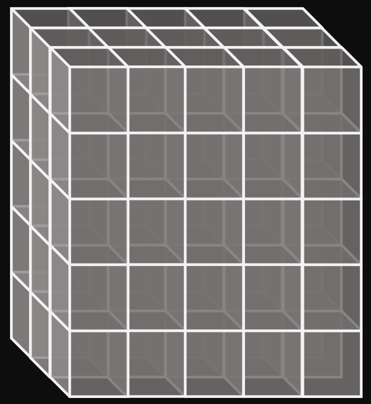
Computing the similarity (score) between query of the first word with key of the **second** word.



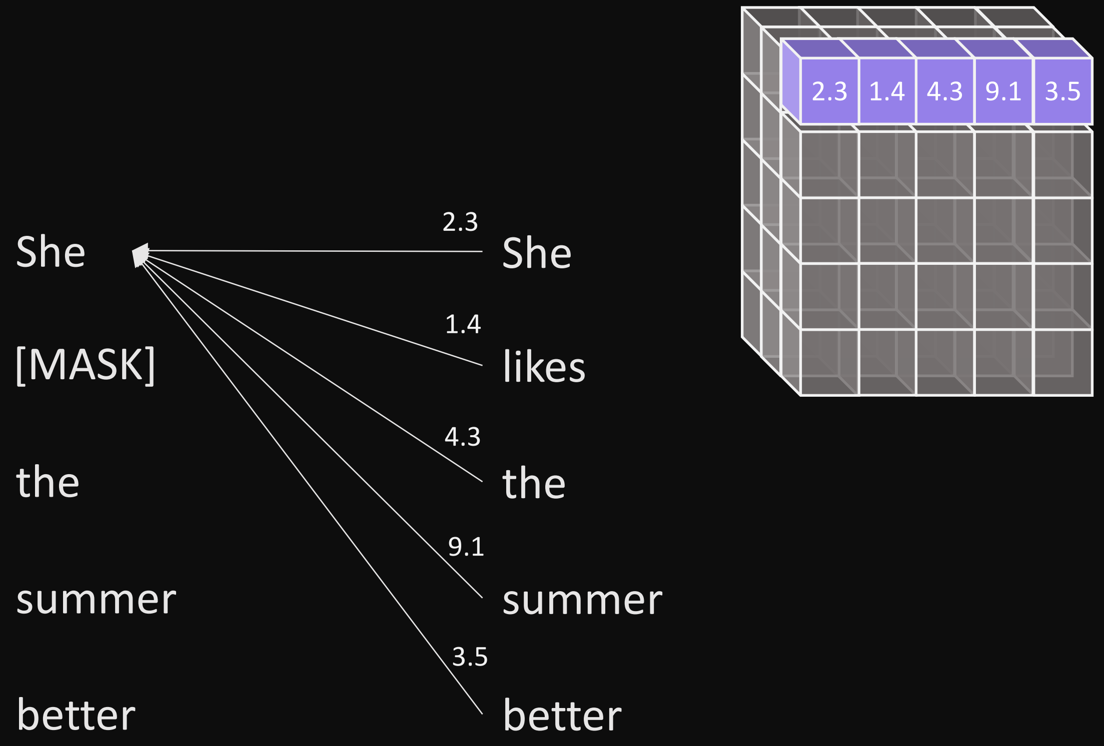
Computing the similarity (score) between query of the first word with key of the **third** word.



This is the final *score* tensor
containing all attention scores.

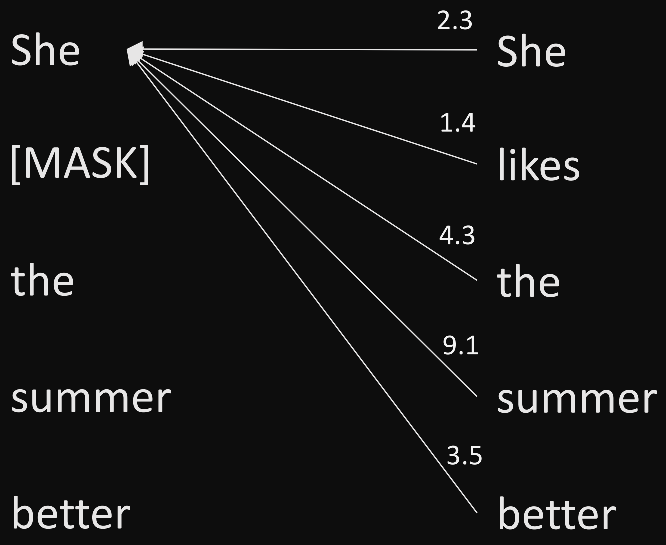


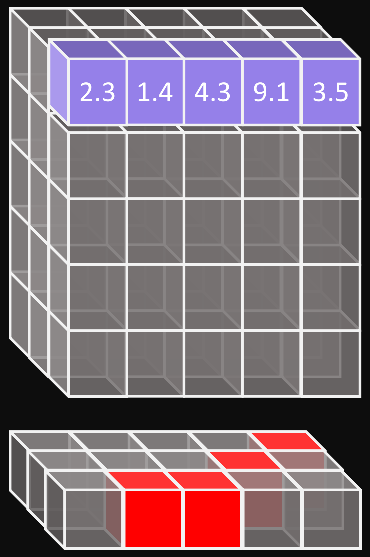
These are the scores which are related to the **first** word in the first sentence.



Remember the mask tensor from the beginning. It is now applied so that the scores from the masked words do not have any influence for further processing.

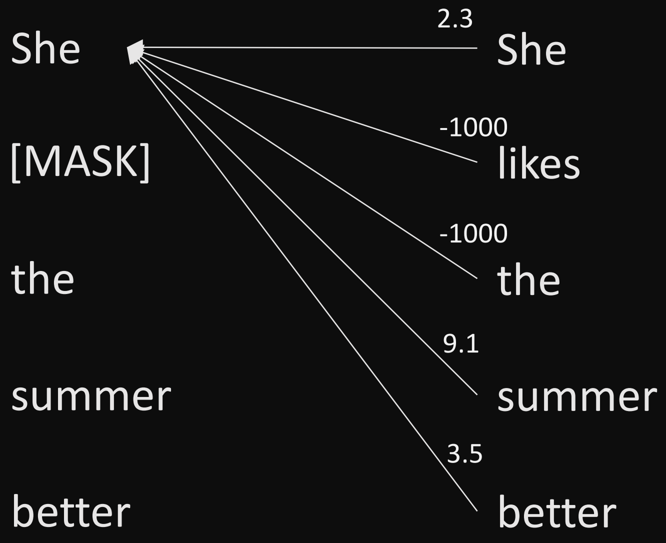
For the first sentence, the scores for *likes* and for *the* should be masked.

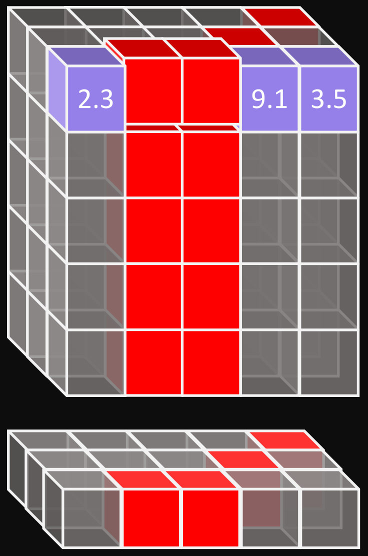




Masking is done by first setting the weights of those words to high negative values (here -1000).

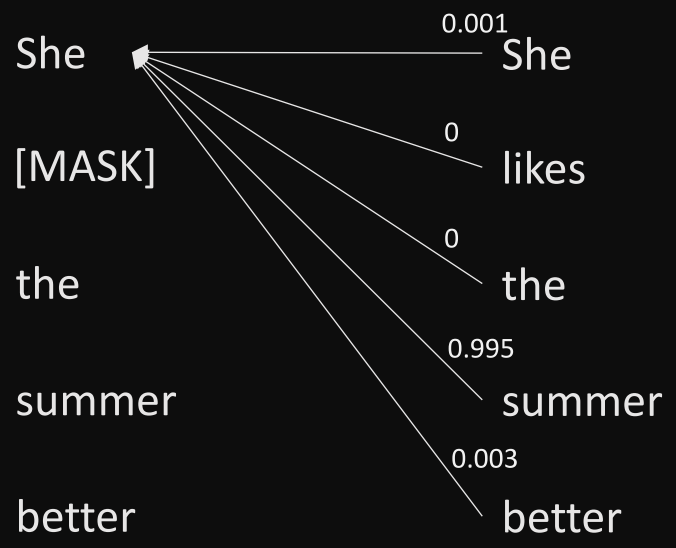
Afterwards, to get proper weights (that sum to 1) instead of scores, a transformation (*softmax*) is applied.

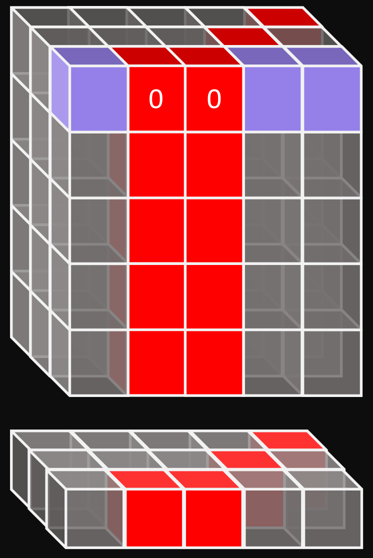




After the *softmax*, the values can be interpreted as weights. As seen in the graphic below, the -1000 was 'translated' into zero.

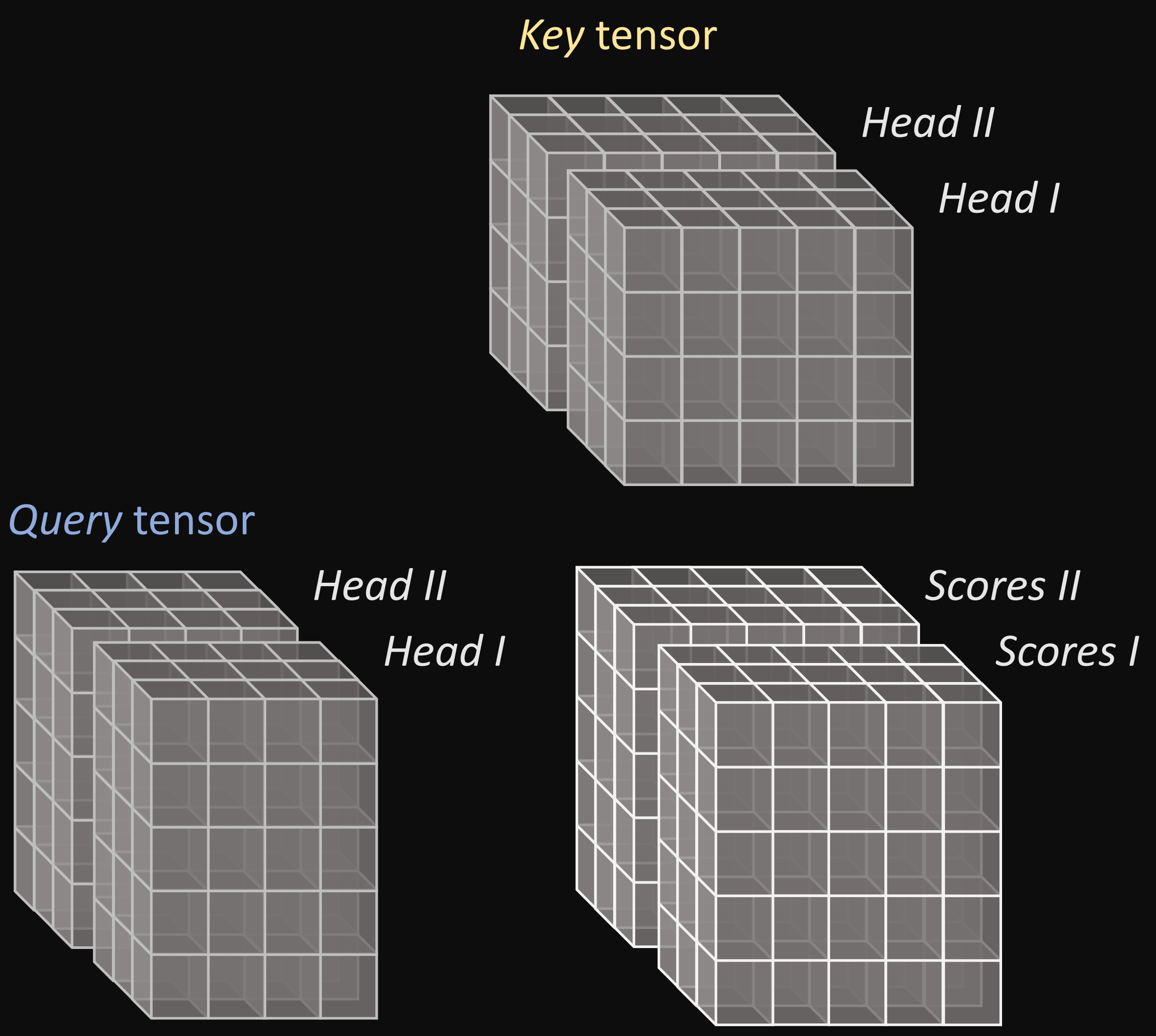
Masking hence means to set the attention weights of some words to zero.





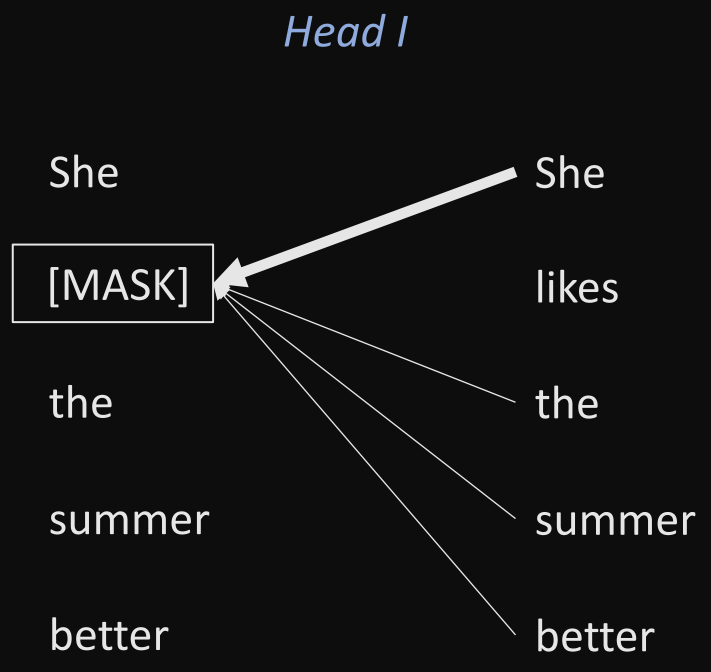
Since we have two attention heads, we want to have not only one but two attention score tensors. The logic as shown previously stays exactly the same though.

The matrix multiplication shown on the right essentially illustrates two separate computations*. The multiplication of *Query* Head I with Key Head I and *Query* Head II with *Key* Head II.

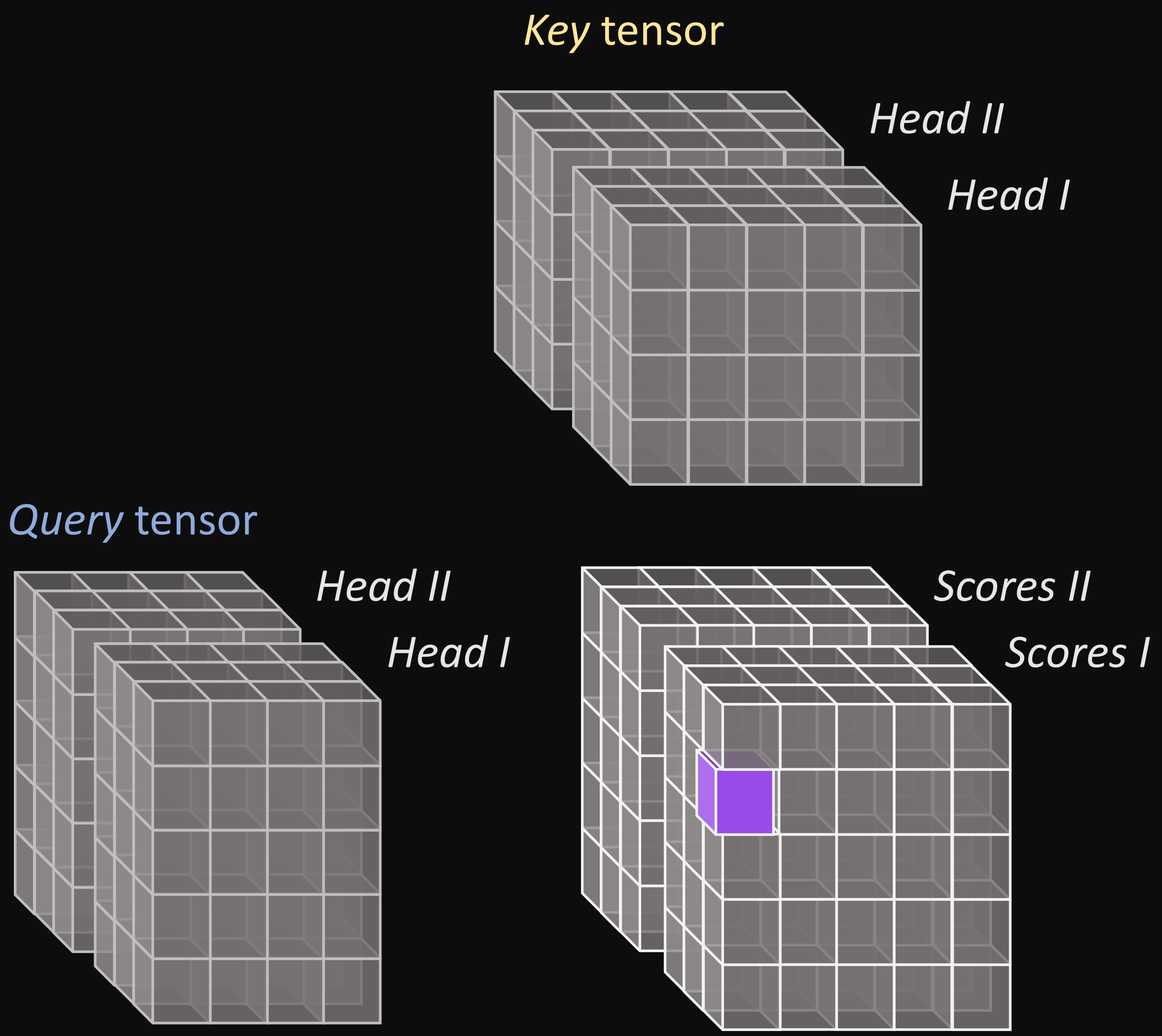


*implemented as a single matmul of 4d tensors

The score of the word *She* for the masked word *likes* is illustrated on the right.



Putting most attention to ***She***

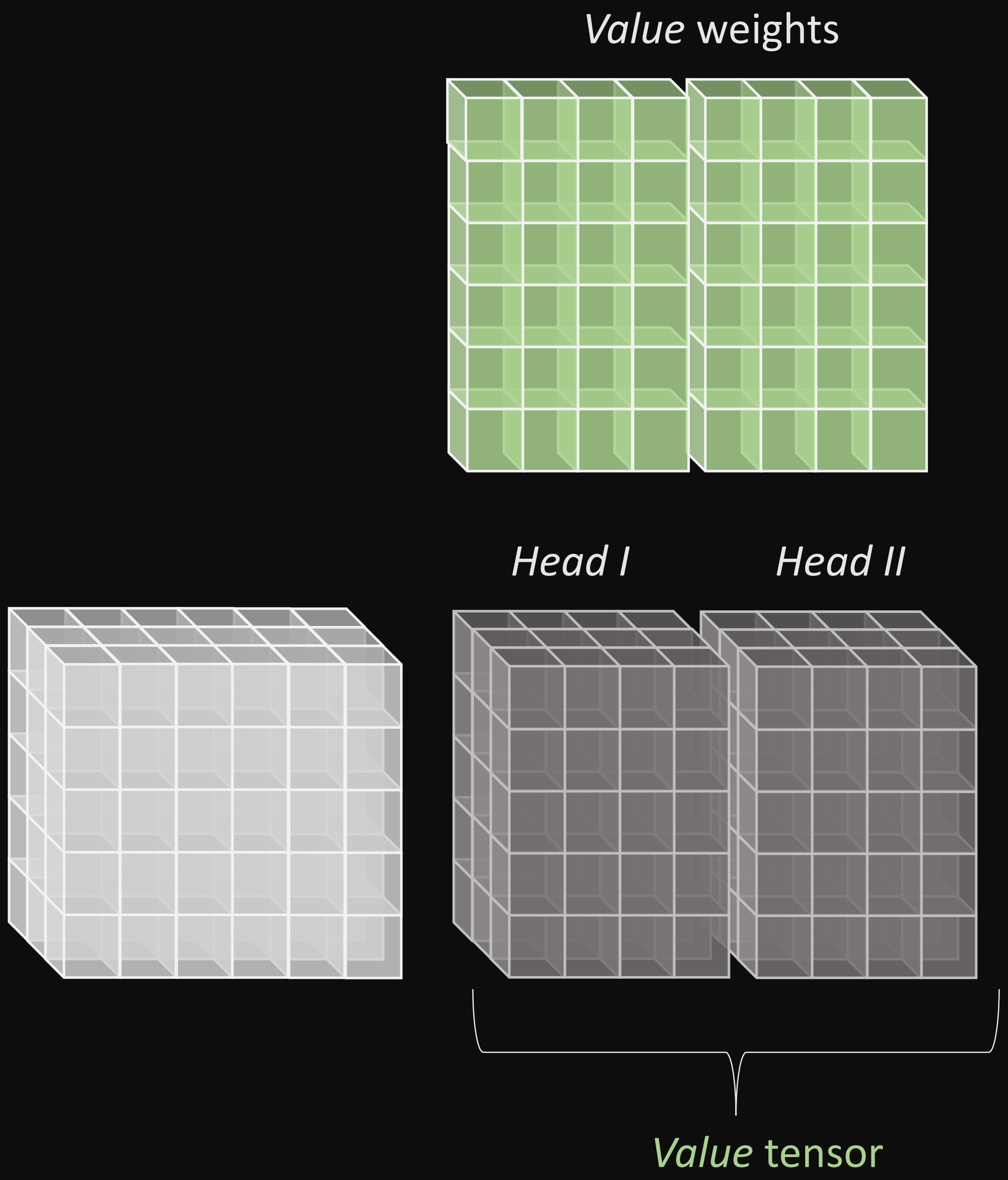


*implemented as a single matmul of 4d tensors

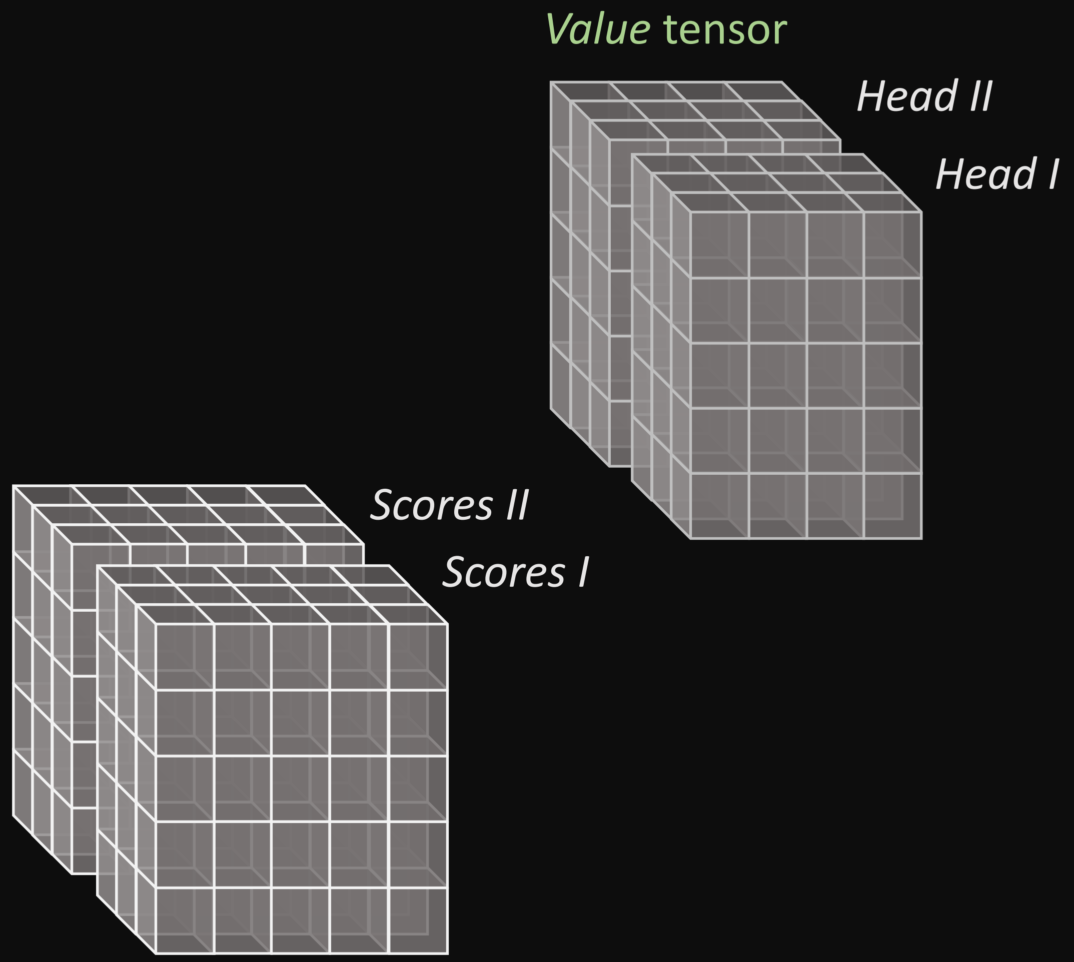
Step II

Generating weighted word representations

We have introduced *Query* and *Key* tensors so far. The last piece missing until now is the *Value* tensor. The *Value* tensor is computed just as Q and K with a single linear layer. The dimensionality of the Value tensor can differ from Q and K but we choose the same dimensionality here.



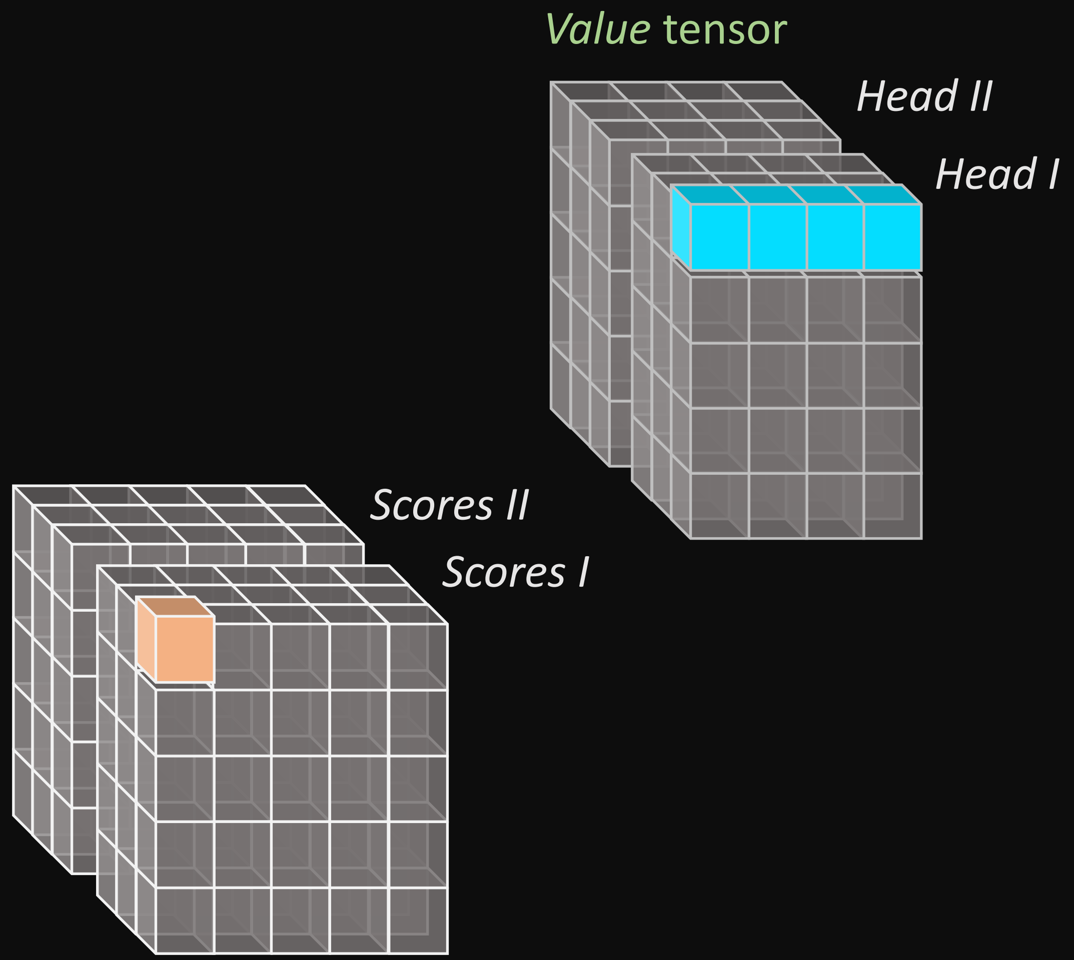
Lastly, we weight each word representation (in the *Value* tensor) with the attention weights we computed earlier, again using two separate computations*. The multiplication of *Scores* I with *Value* Head I and *Scores* II with *Value* Head II.



*implemented as a single matmul of 4d tensors

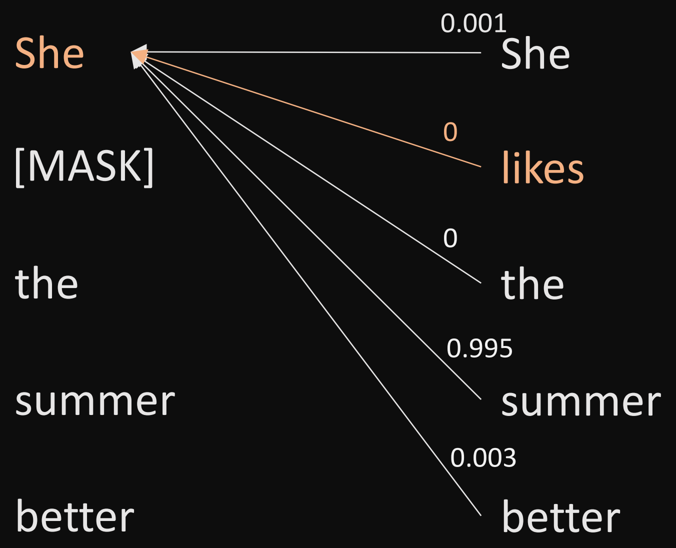
Let's go through a few steps to retrieve the final weighted word representation for the first word of the first sentence.

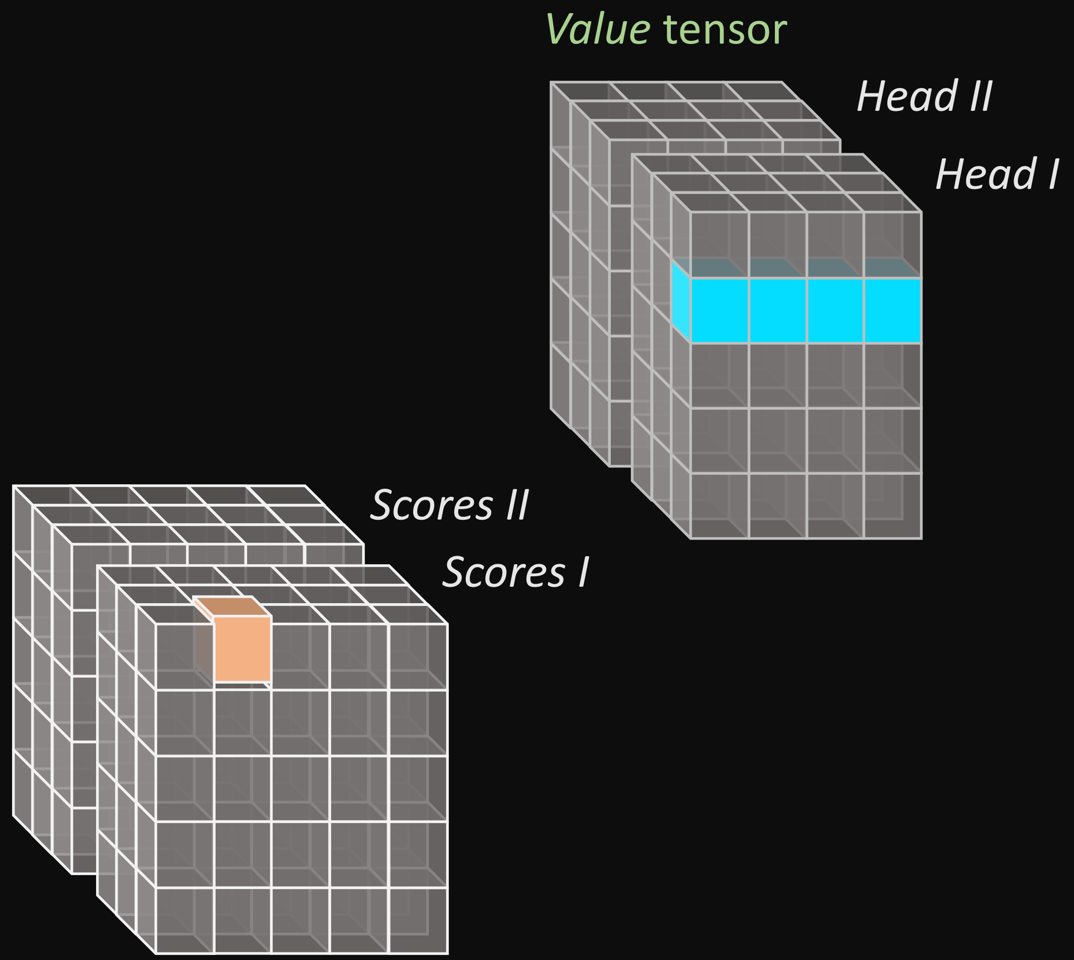
The weight of the first word with respect to the first word is multiplied by the raw vector of the first word



Let's got through a few steps to retrieve the final weighted word representation for the first word of the first sentence.

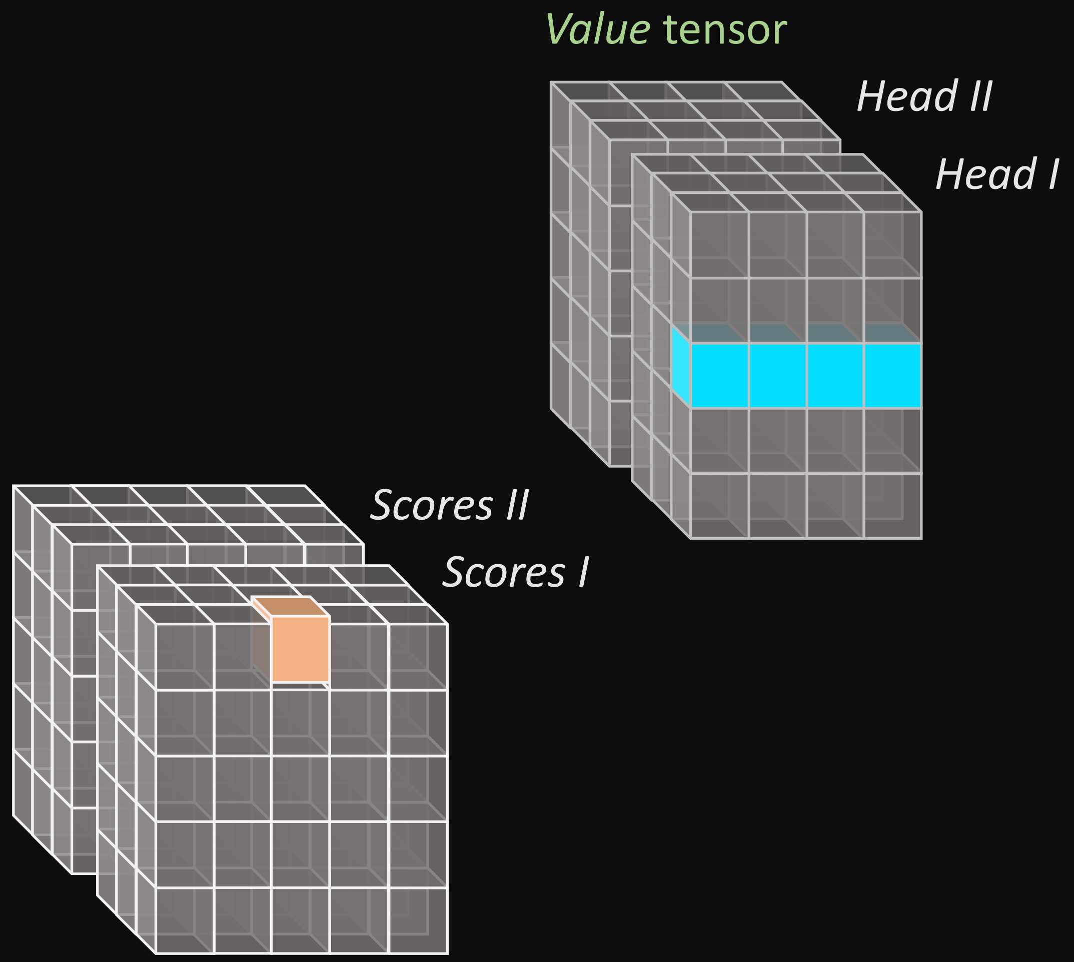
The weight of the second word with respect to the first word is multiplied by the raw vector of the second word



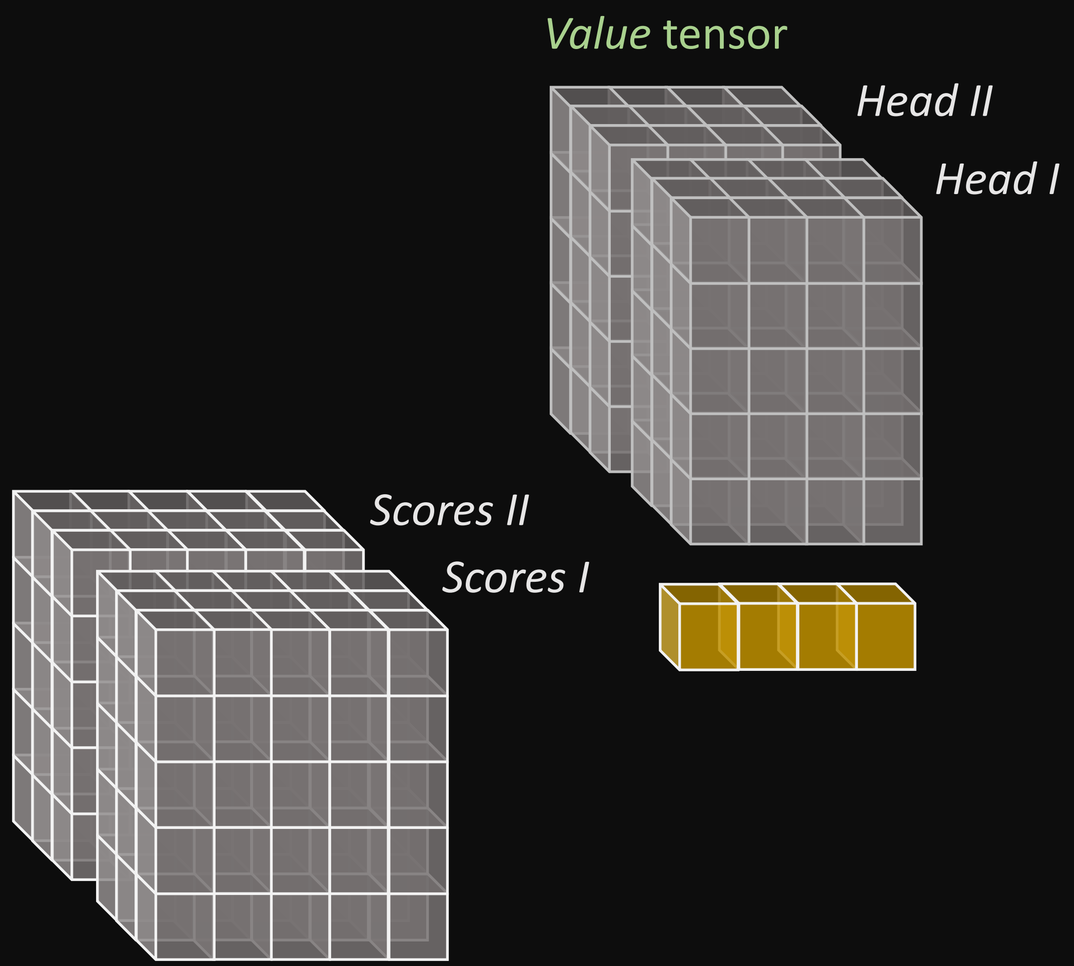


Let’s got through a few steps to retrieve the final weighted word representation for the first word of the first sentence.

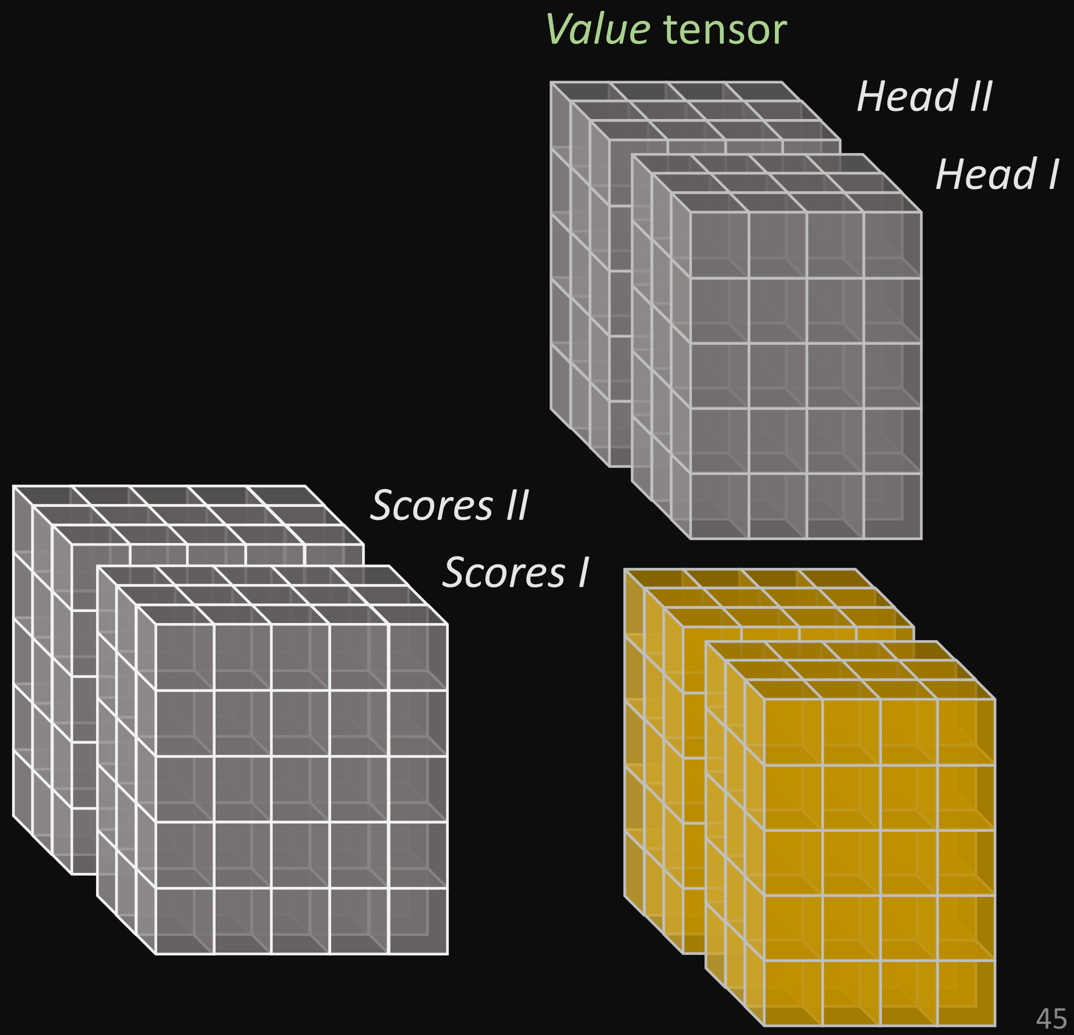
The weight of the third word with respect to the first word is multiplied by the raw vector of the second word



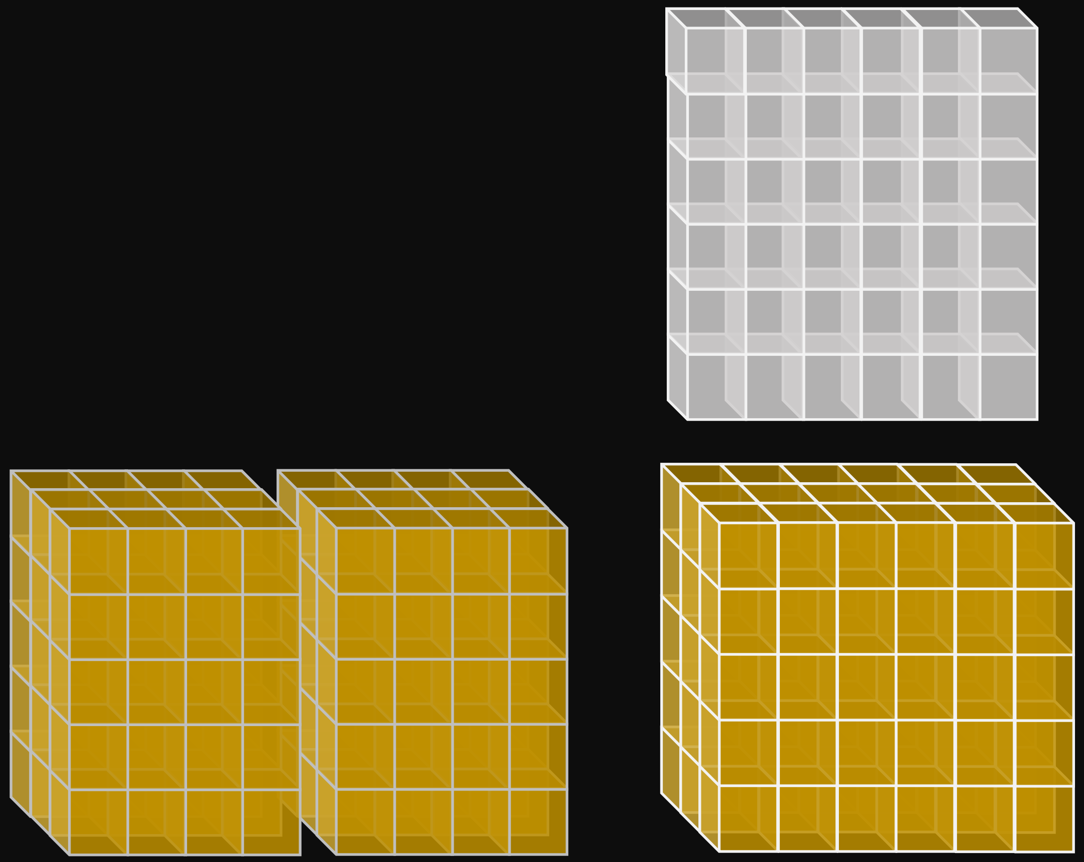
Step 4 and 5 are not shown here. In the end, you get the weighted word representation of the first word in the first sentence.



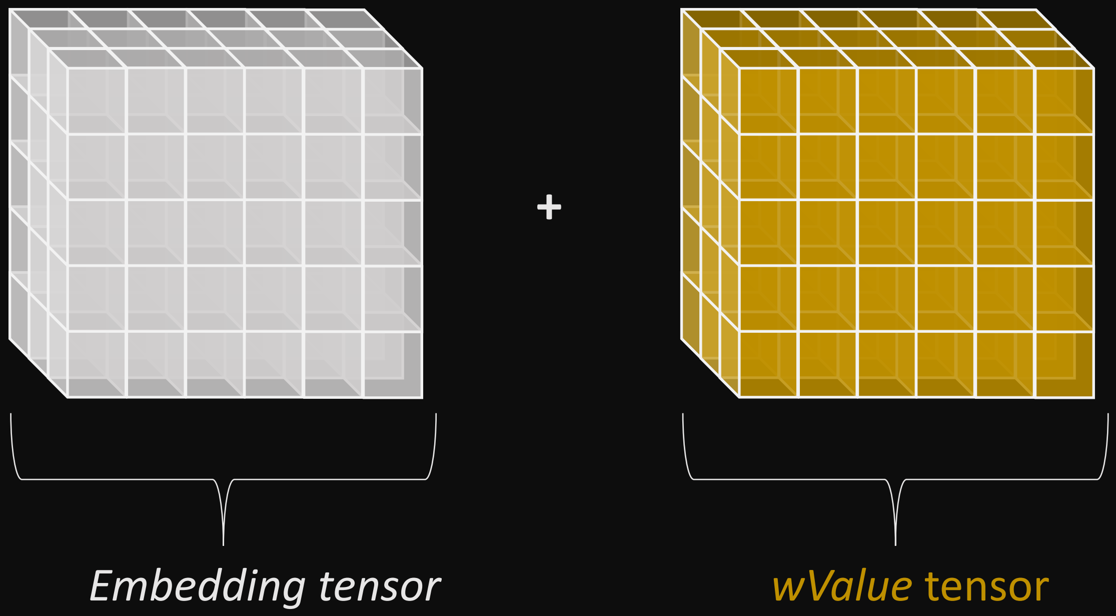
The final weighted Value tensor is shown on the right.



The *reshaped weighted Value tensor* is passed through a linear layer to retrieve the original input tensor shape we started with in the first place.



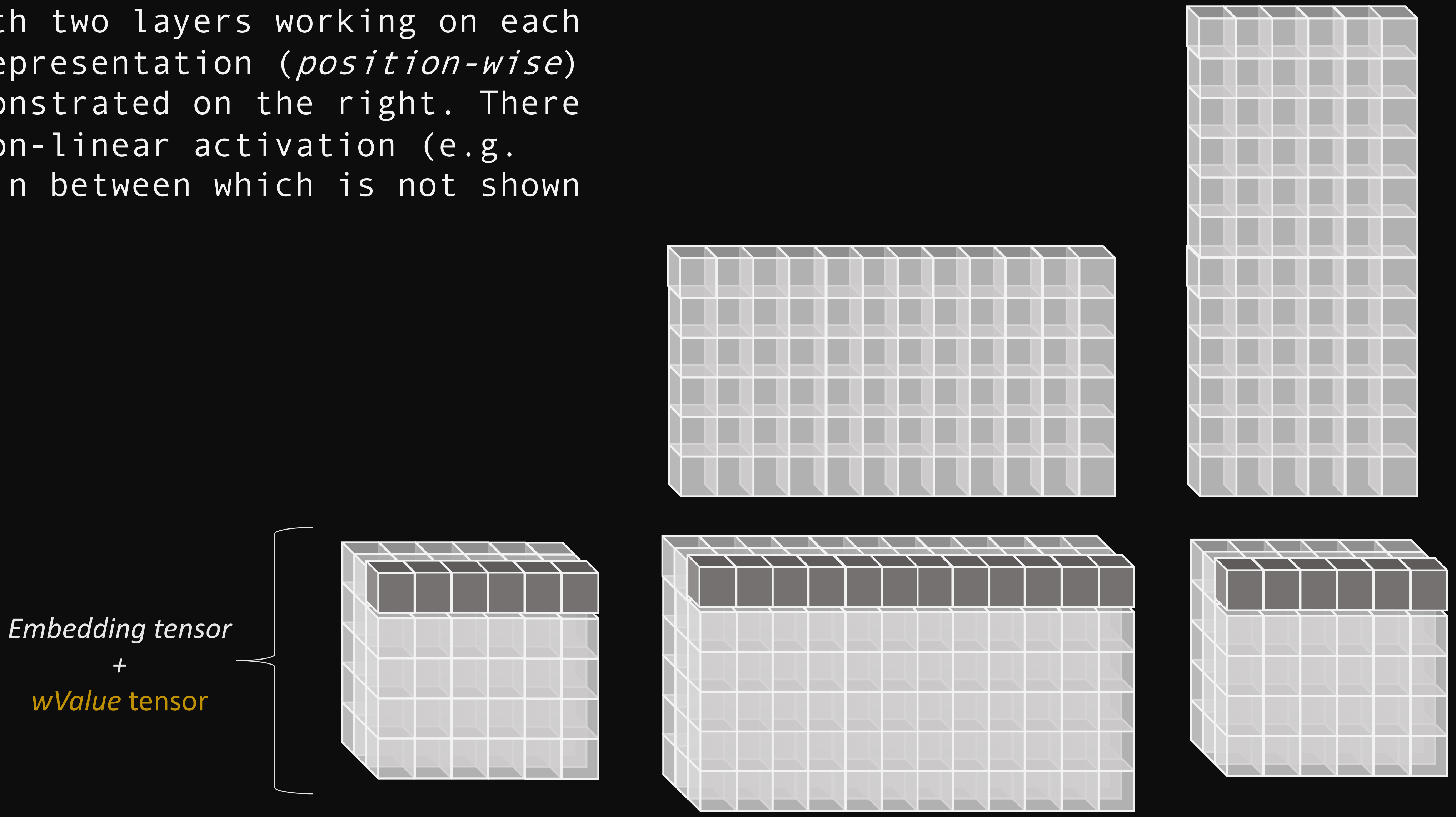
Along the way, we might have lost some crucial information inherent in the data. For this reason, the *input tensor with the embeddings* is added back to the transformed *weighted Value* tensor. You see now why it is important to set embedding vectors of masked words to zero (see slide 14) since otherwise information about masked words would be leaked.



Step III

Position-wise feed forward

The last step is a small feed forward net with two layers working on each word representation (*position-wise*) as demonstrated on the right. There is a non-linear activation (e.g. relu) in between which is not shown here.

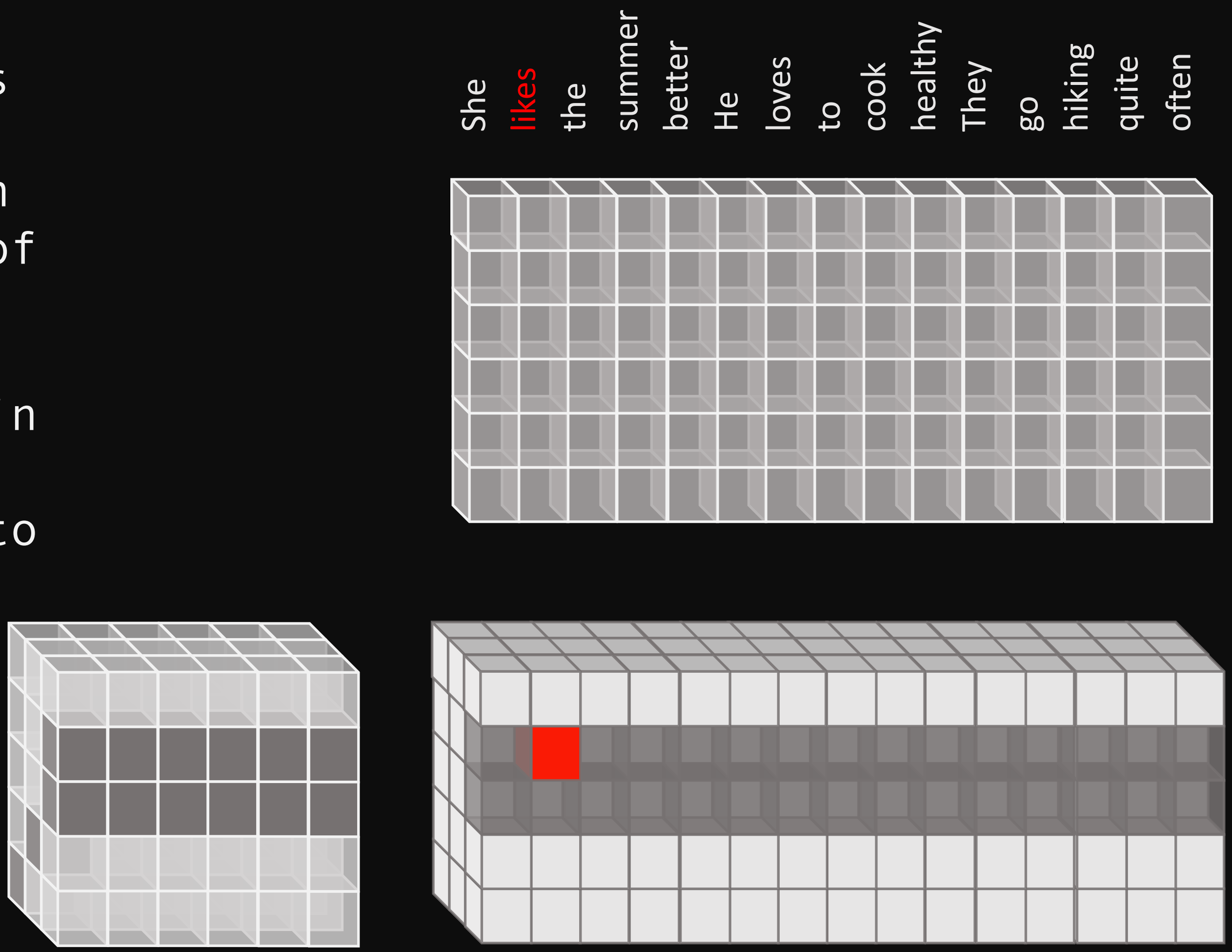


Step IV

Predict masked words

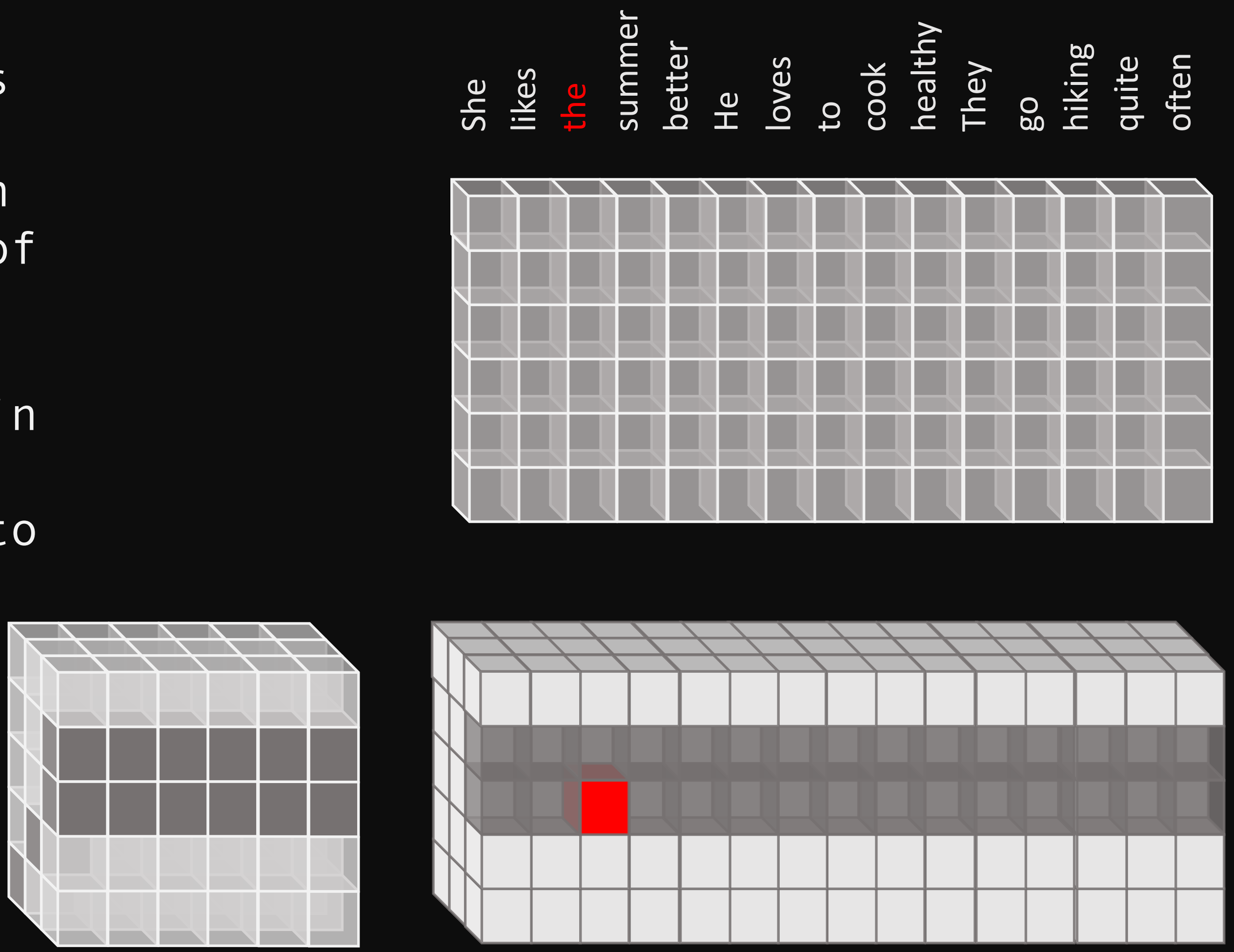
Last but not least, we can apply a linear layer followed by a softmax activation to transform the tensor with weighted embeddings into a vector containing as many values as there are words in the vocabulary (here 15). Each resulting value can be interpreted as the probability of being each word in the vocabulary.

We would like the value displayed in red to have the highest number (probability) since it is related to the word *likes*



Last but not least, we can apply a linear layer followed by a softmax activation to transform the tensor with weighted embeddings into a vector containing as many values as there are words in the vocabulary (here 15). Each resulting value can be interpreted as the probability of being each word in the vocabulary.

We would like the value displayed in red to have the highest number (probability) since it is related to the word *the*



I hope you enjoyed it!

If you have any suggestions or questions, don’t hesitate to contact me

https://twitter.com/UlfMertens