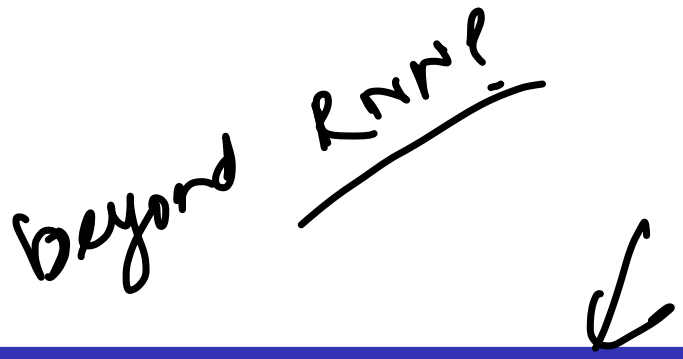


Transformers: Attention is all you need

Pawan Goyal

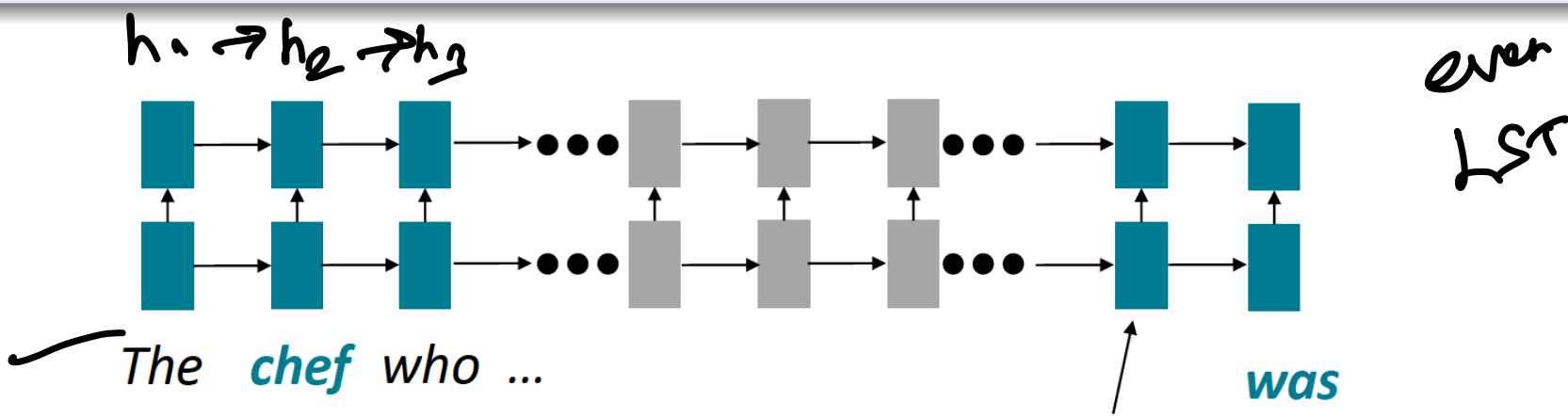
CSE, IIT Kharagpur

CS60010

Issues with recurrent models

O(sequence length) steps for distant word pairs

- Hard to learn long-distance dependencies (gradient problems!)
- Linear order of words is "baked in"; not the right way to think about sentences

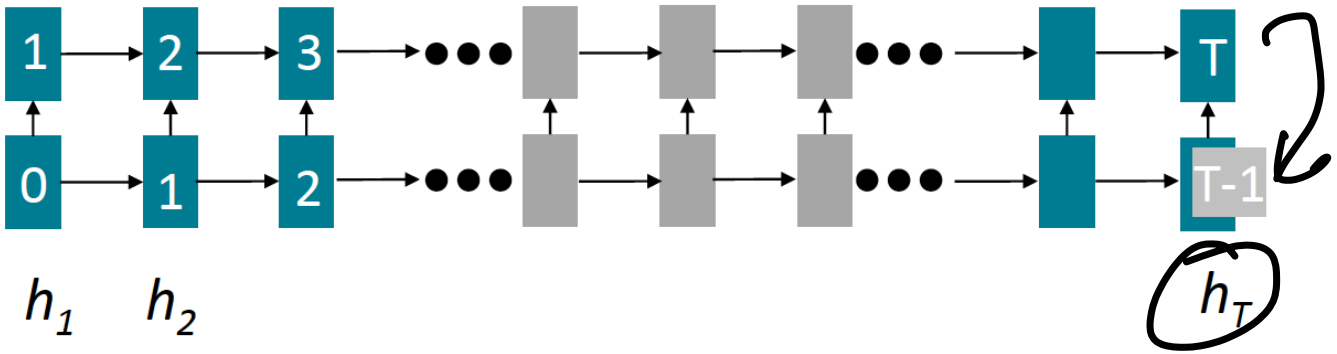


Info of ***chef*** has gone through O(sequence length) many layers!

Issues with recurrent models

Lack of parellilizability

- Future RNN hidden states can't be computed in full before past RNN hidden states have been computed

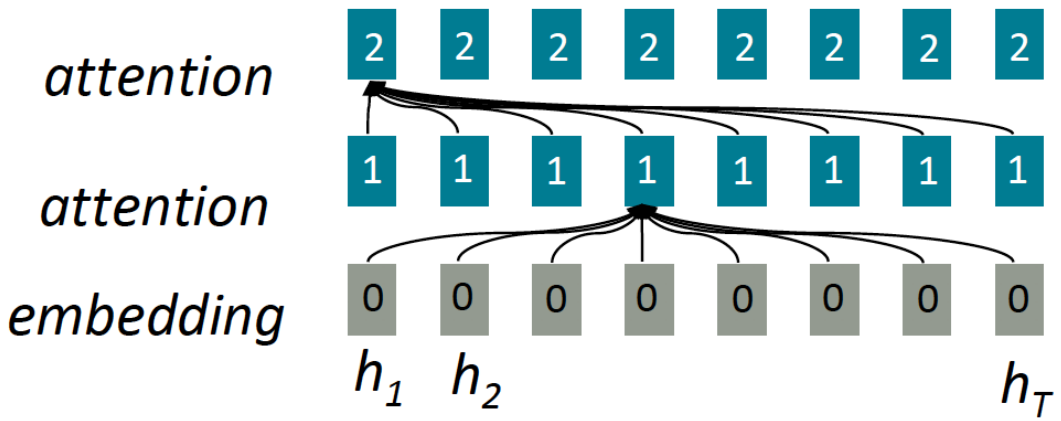


Numbers indicate min # of steps before a state can be computed

If not recurrence, then what?

Attention

- Given a word as query, attention can be used to access and incorporate information from a set of values (other words)
- Can we do this within a single sentence?
- All words can interact with each other and computation can be done in parallel!!



All words attend to all words in previous layer; most arrows here are omitted

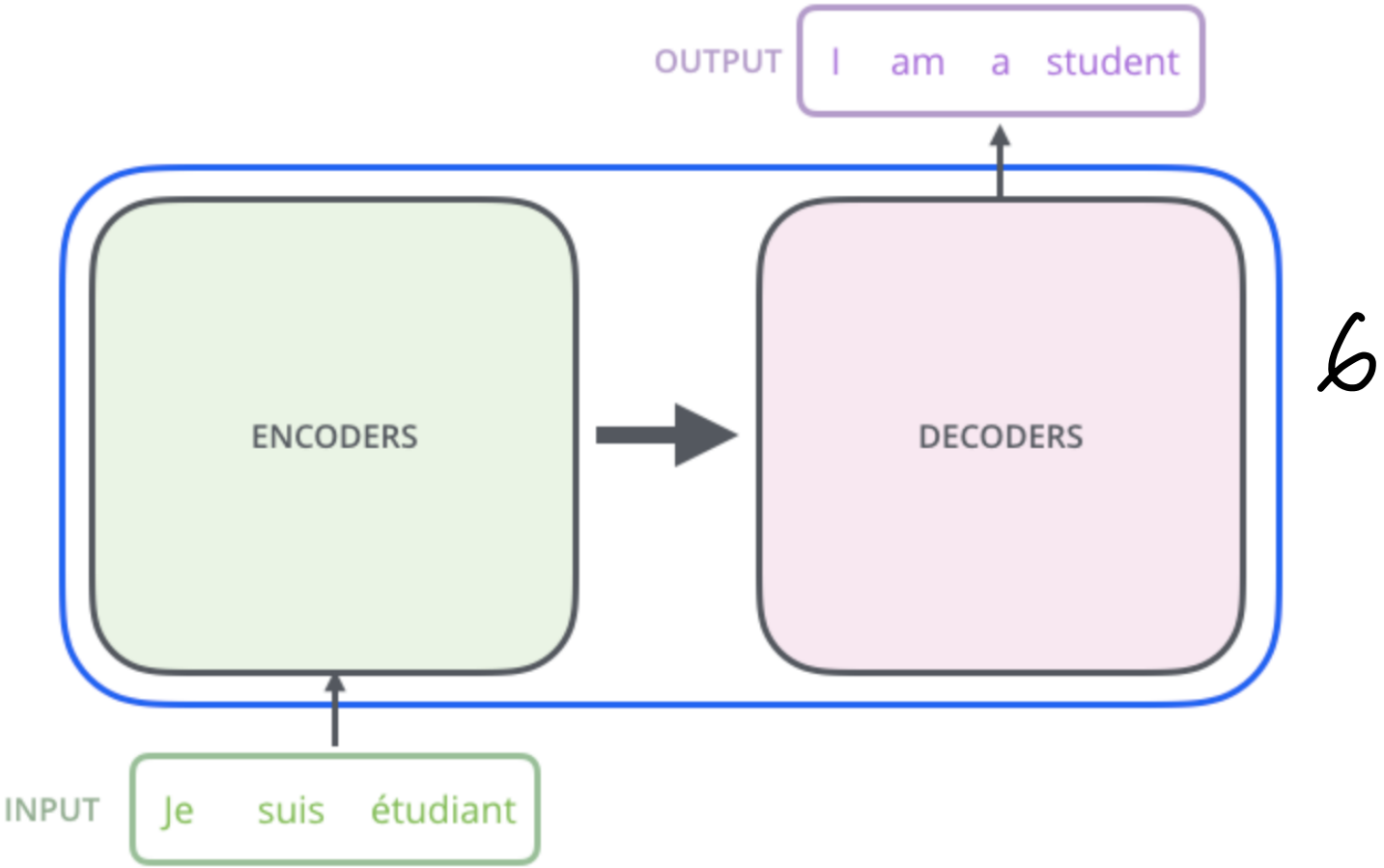
Transformers: A High-Level Look

In a Machine Translation application, it would take a sentence in one language and output translation in another.

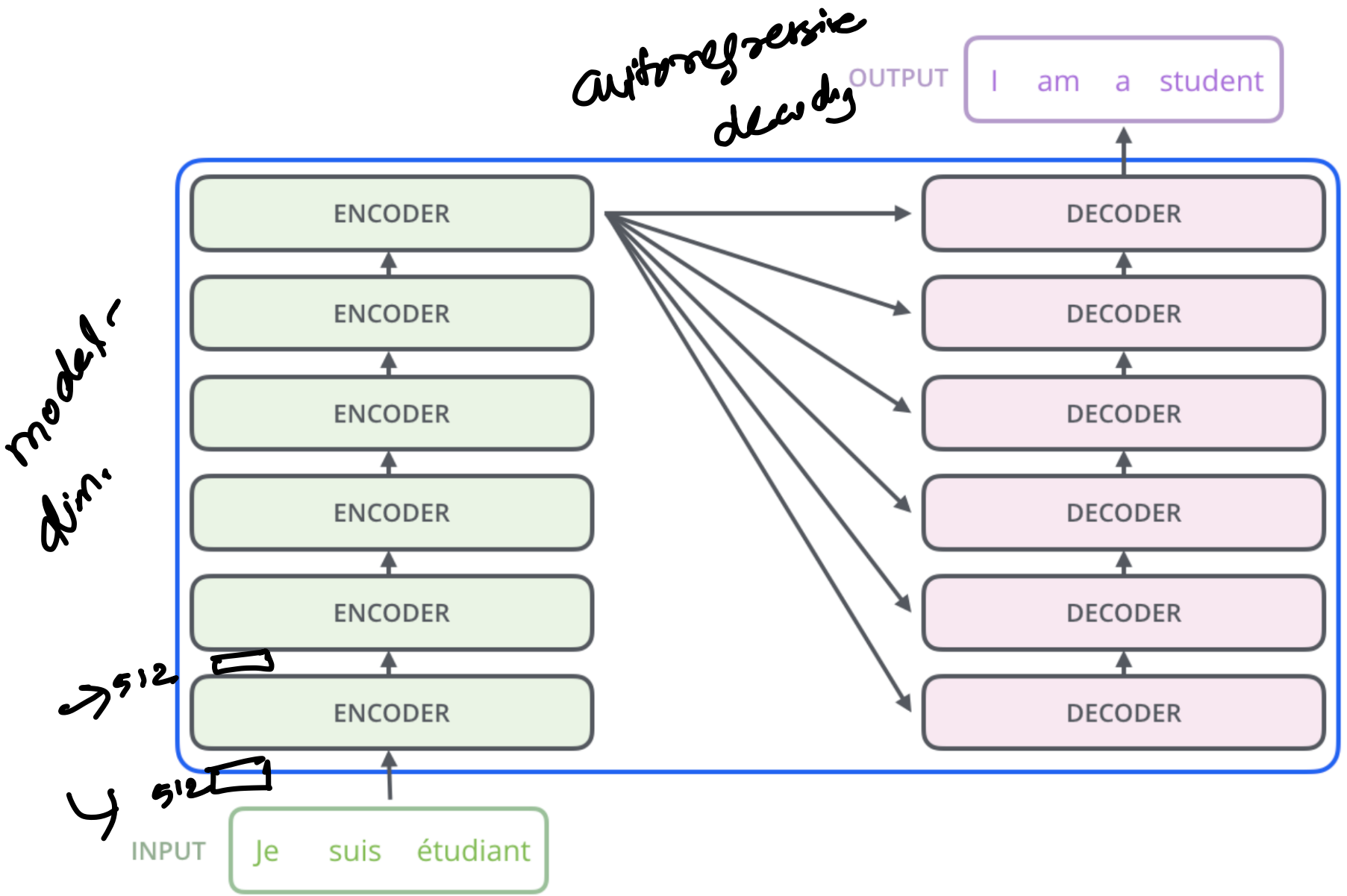
http://jalammar.github.io/illustrated-transformer/

A High-Level Look

On a high-level, we can also see an encoding component, a decoding component and connections between them.

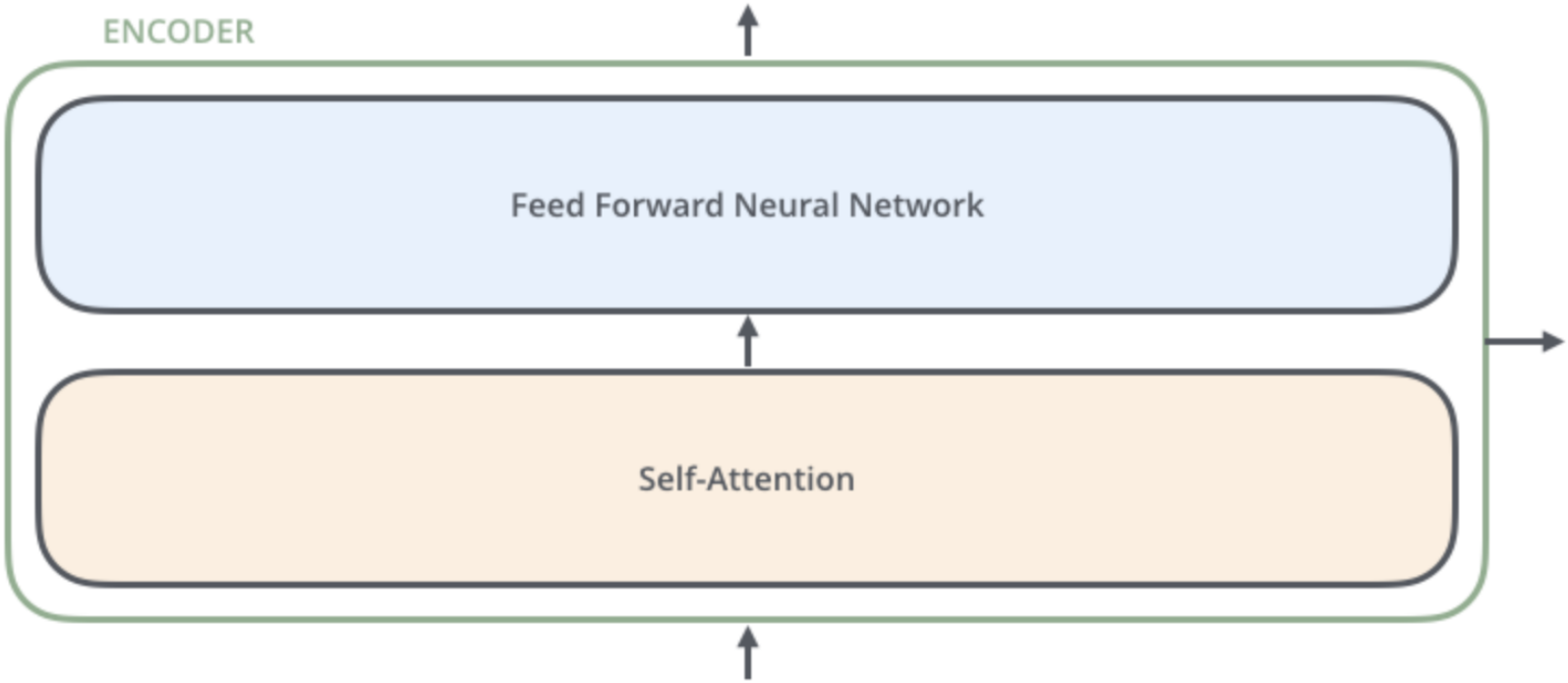


Encoders and Decoders are stacked



Let's zoom in on an encoder

Each encoder can be broken down into two sub-layers.

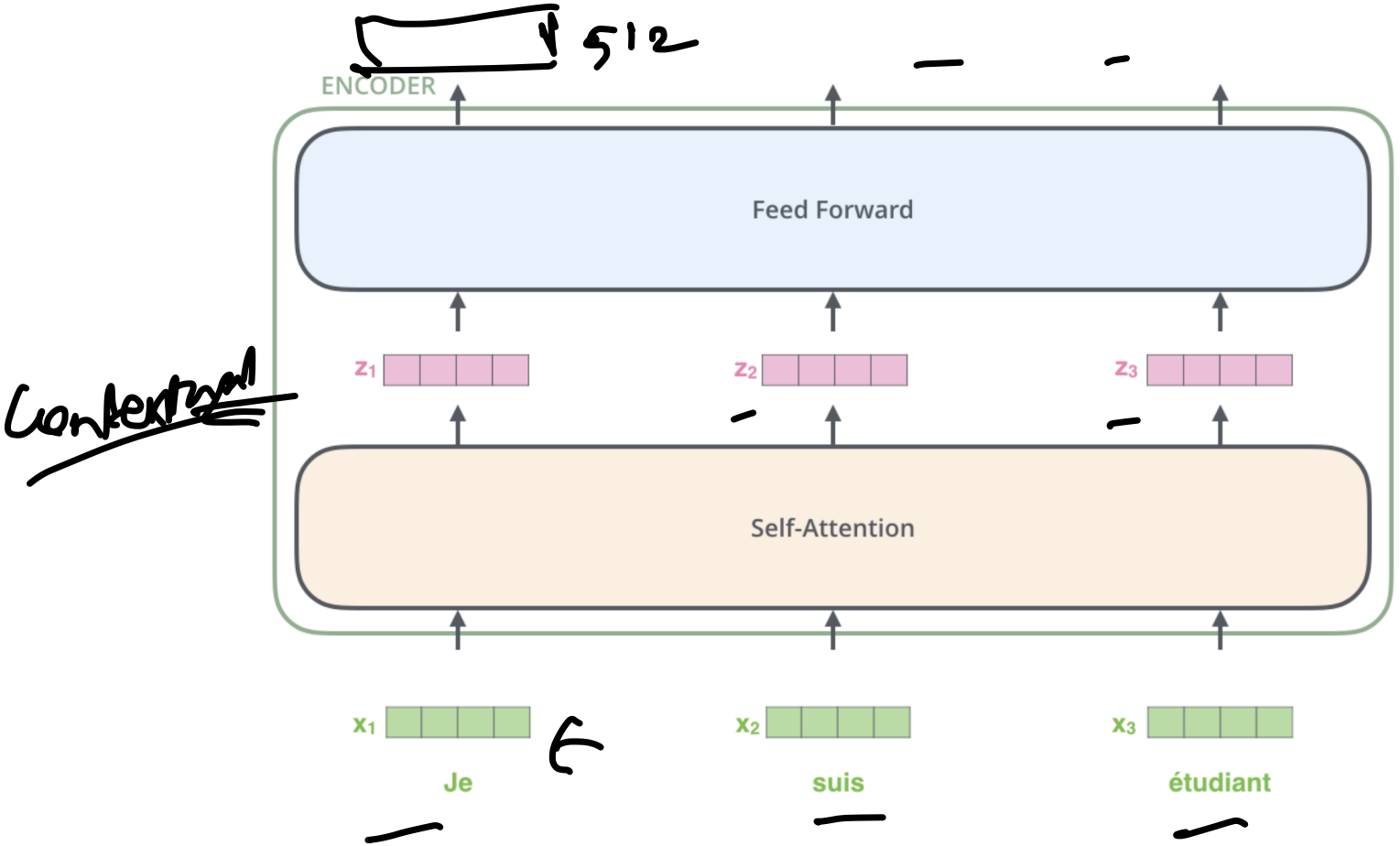


How is an input processed through an encoder?

- Each encoder receives a list of vectors each of size 512.
- In the bottom encoder, this would be the word embeddings.
- In other encoders, it would be the output of the encoder that is directly below.

Each word is embedded into a vector of size 512. We'll represent those vectors with these simple boxes.

Flow through an encoder



- The word in each position flows through its own path in the encoder.
- There are dependencies between these paths in the self-attention layer but not in the feed-forward layer.

The need for self-attention

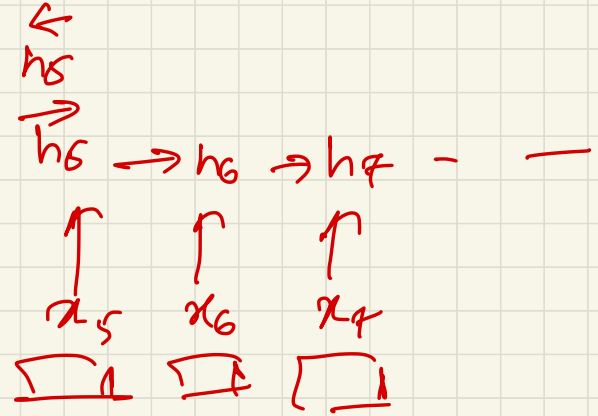
Will it also help get a better representation?

Consider the two sentences

- The animal didn't cross the street because it was too tired
- The animal didn't cross the street because it was too wide

This is very difficult challenge for a ~~machine~~ translation system – *coreference resolution*

How will RNN get a contextual representation

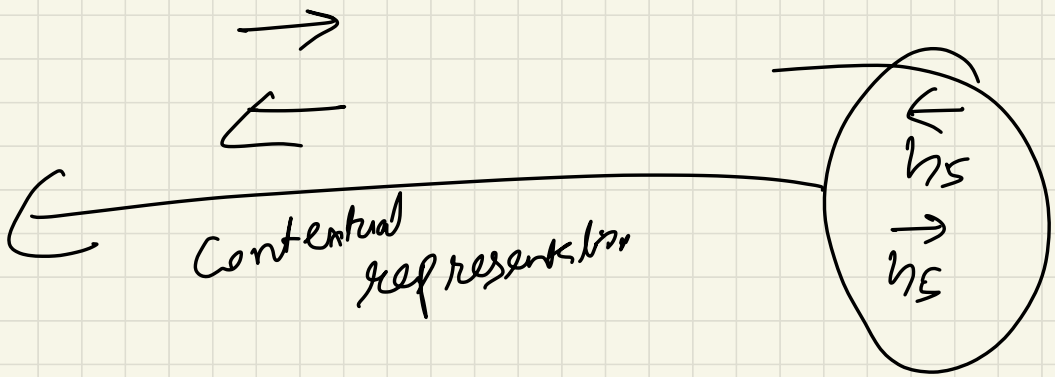


if RNN is artificially trained

I went to the bank to withdraw money

ELMo (2018)

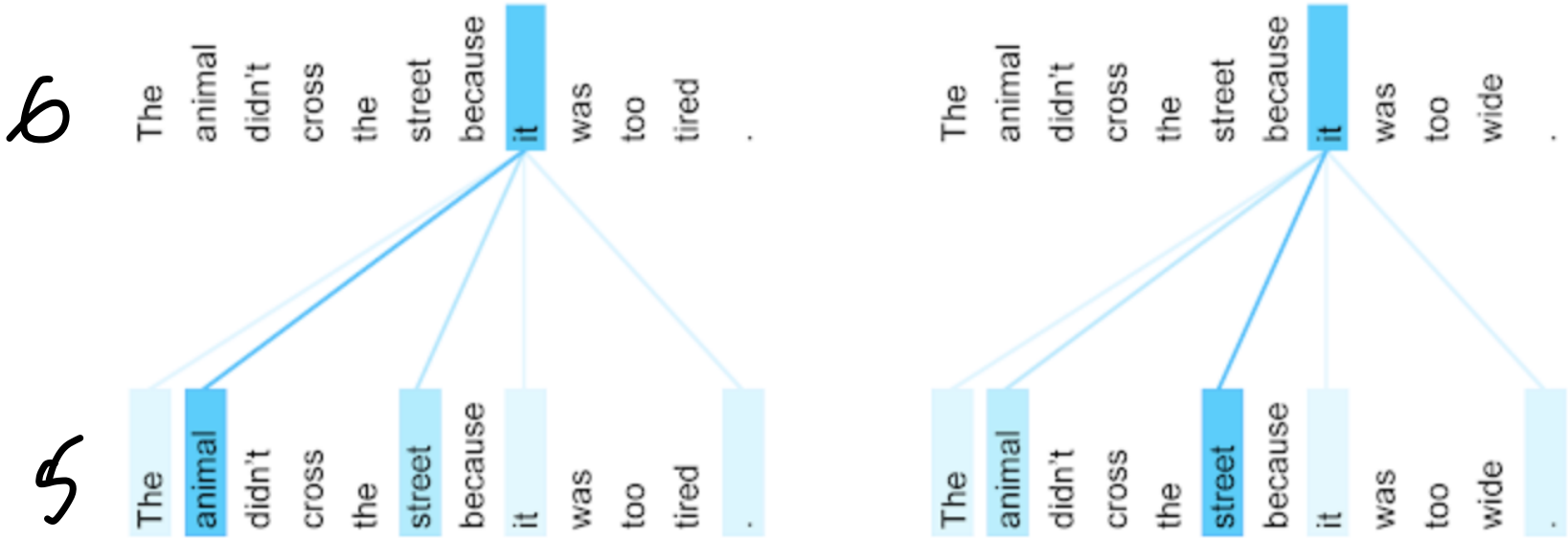
Train RNN for LM



I went to the bank

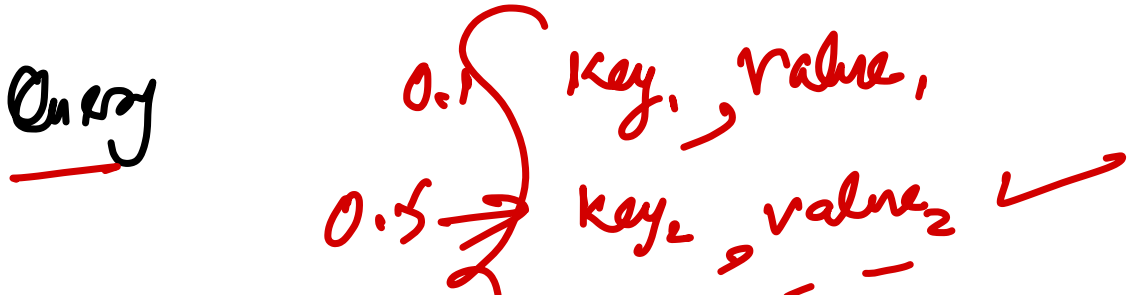
Motivation for self-attention

Can a better representation be obtained in the encoder by attending to the relevant context?



https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html

Intuitive Understanding



The key/value/query concept is analogous to retrieval systems.

When you search for videos on Youtube

- The search engine will map your query (text in the search bar) against a set of keys (video title, description, etc.) associated with candidate videos in their database
- It will then present you the best matched videos (values).

Generalized Attention: queries, keys and values

- Recall: Attention operates on **queries**, **keys**, and **values.**
  - We have some **queries** $q_1, q_2, \ldots, q_T$. Each query is $q_i \in \mathbb{R}^d$
  - We have some **keys** $k_1, k_2, \ldots, k_T$. Each key is $k_i \in \mathbb{R}^d$
  - We have some **values** $v_1, v_2, \ldots, v_T$. Each value is $v_i \in \mathbb{R}^d$

The number of queries can differ from the number of keys and values in practice.

- In **self-attention**, the queries, keys, and values are drawn from the same source.
  - For example, if the output of the previous layer is $x_1, \ldots, x_T$, (one vec per word) we could let $v_i = k_i = q_i = x_i$ (that is, use the same vectors for all of them!)
- The (dot product) self-attention operation is as follows:

$$e_{ij} = q_i^\top k_j$$

Compute **key-query** affinities

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{j'} \exp(e_{ij'})}$$

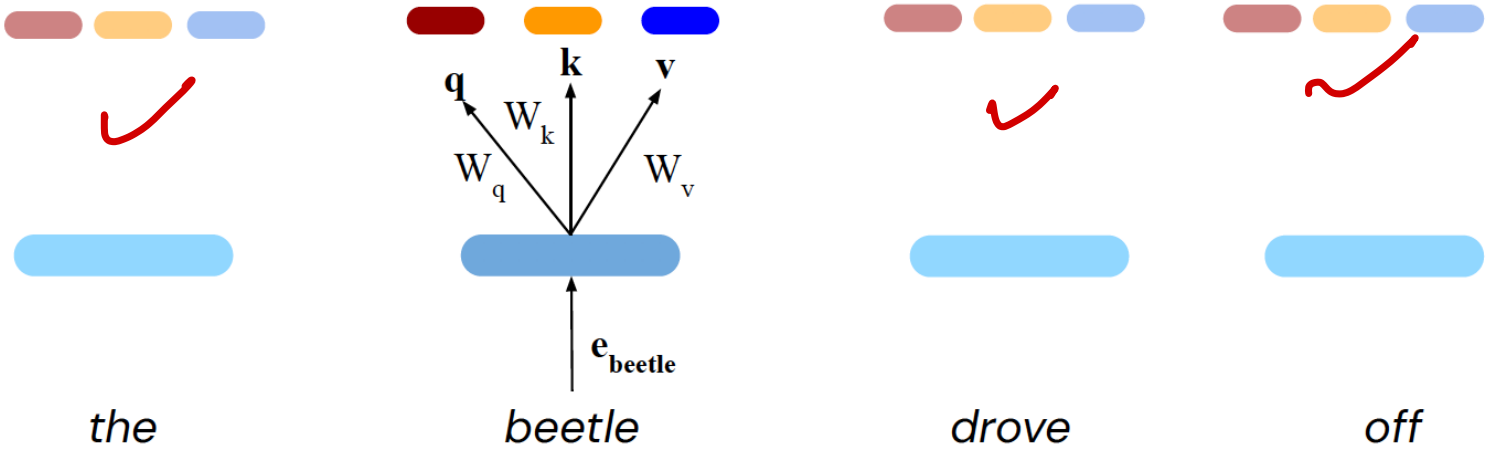
Compute attention weights from affinities (softmax)

$$\text{output}_i = \sum_j \alpha_{ij} v_j$$

Compute outputs as weighted sum of **values**

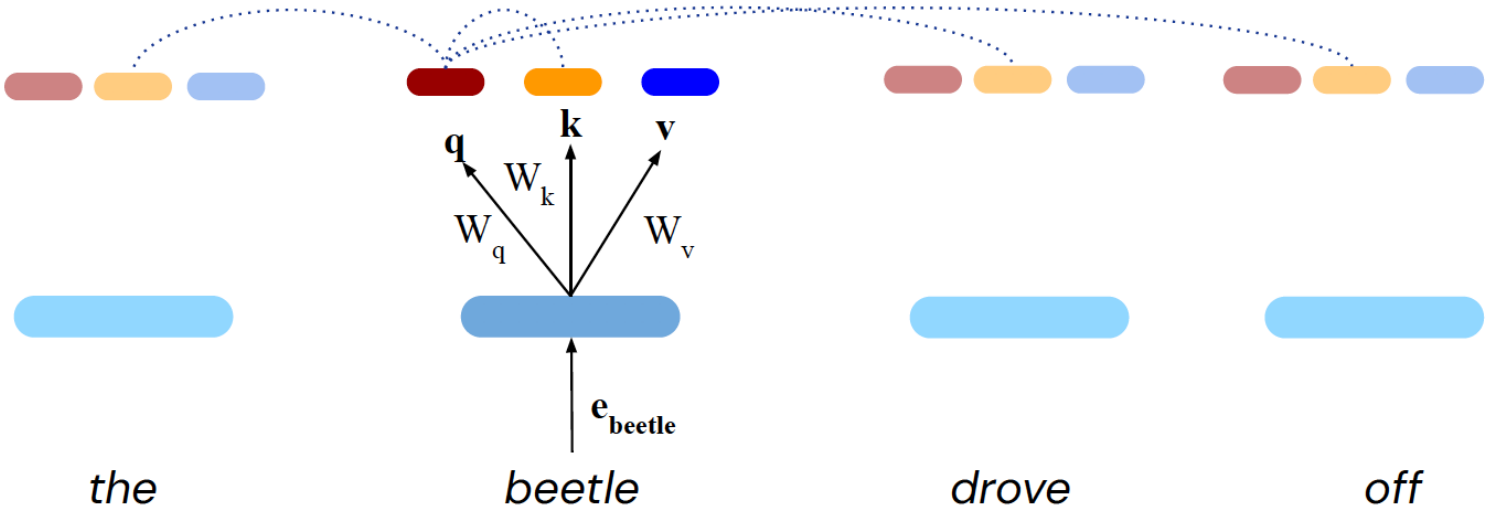
Transformers: Self-attention over input embeddings

$$\mathbf{q} = \mathbf{e}_{beetle} W_q$$
$$\mathbf{k} = \mathbf{e}_{beetle} W_k$$
$$\mathbf{V} = \mathbf{e}_{beetle} W_v$$



https://deepmind.com/learning-resources/deep-learning-lecture-series-2020

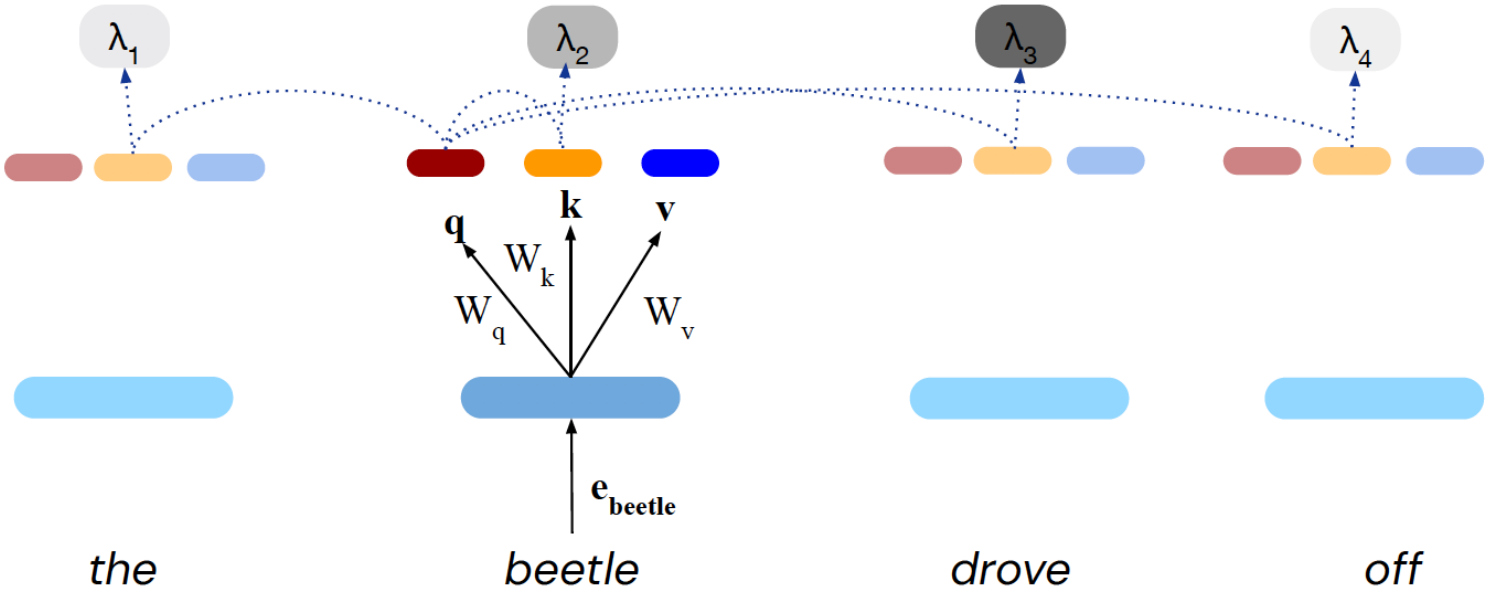
Self-attention over input embeddings



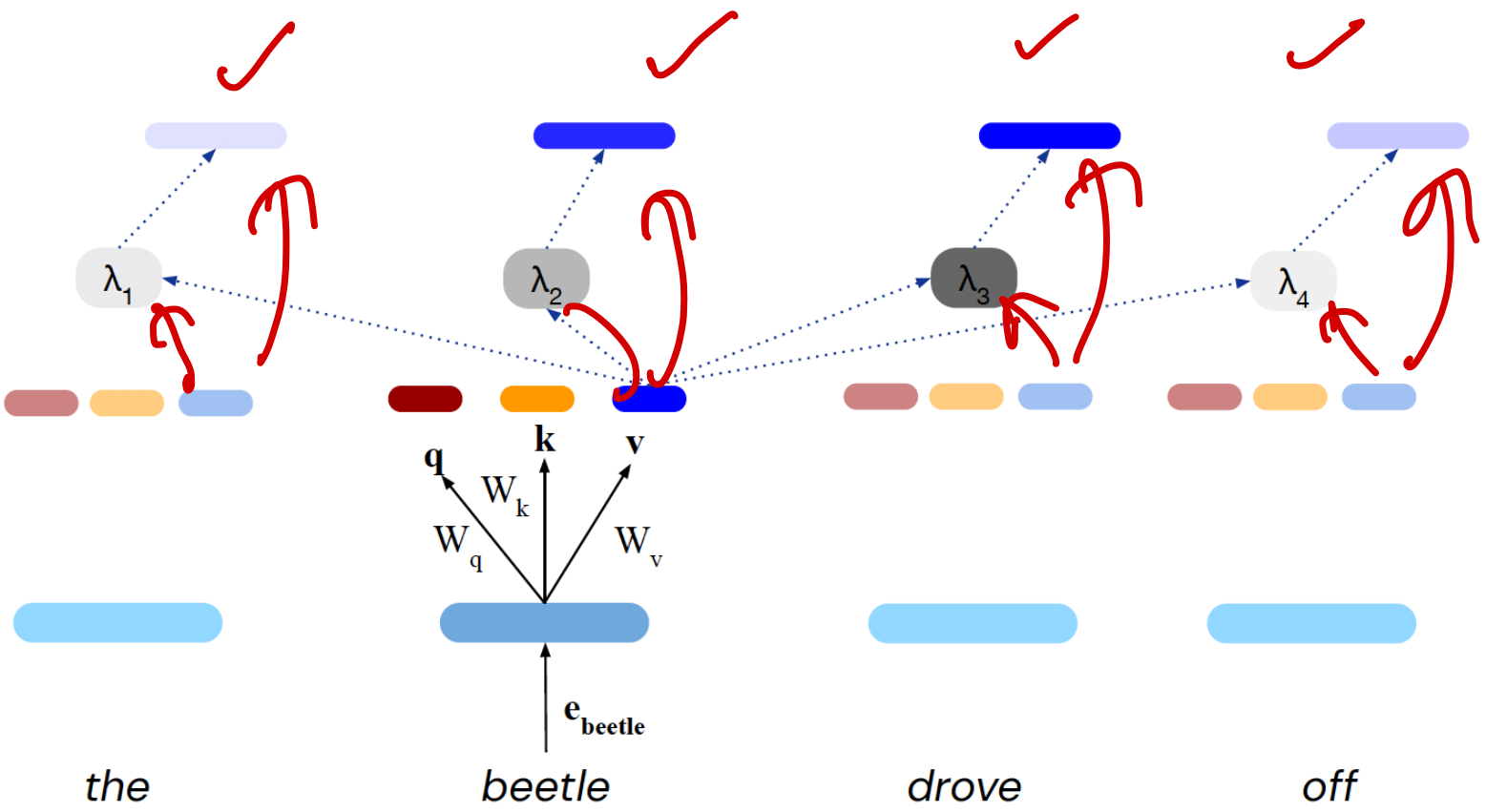
Self-attention over input embeddings

$$\lambda_i = \frac{e^{\mathbf{q}\cdot\mathbf{k_i}}}{\sum_{i=1}^{4} e^{\mathbf{q}\cdot\mathbf{k_i}}}$$

$\sum \lambda_i v_i$

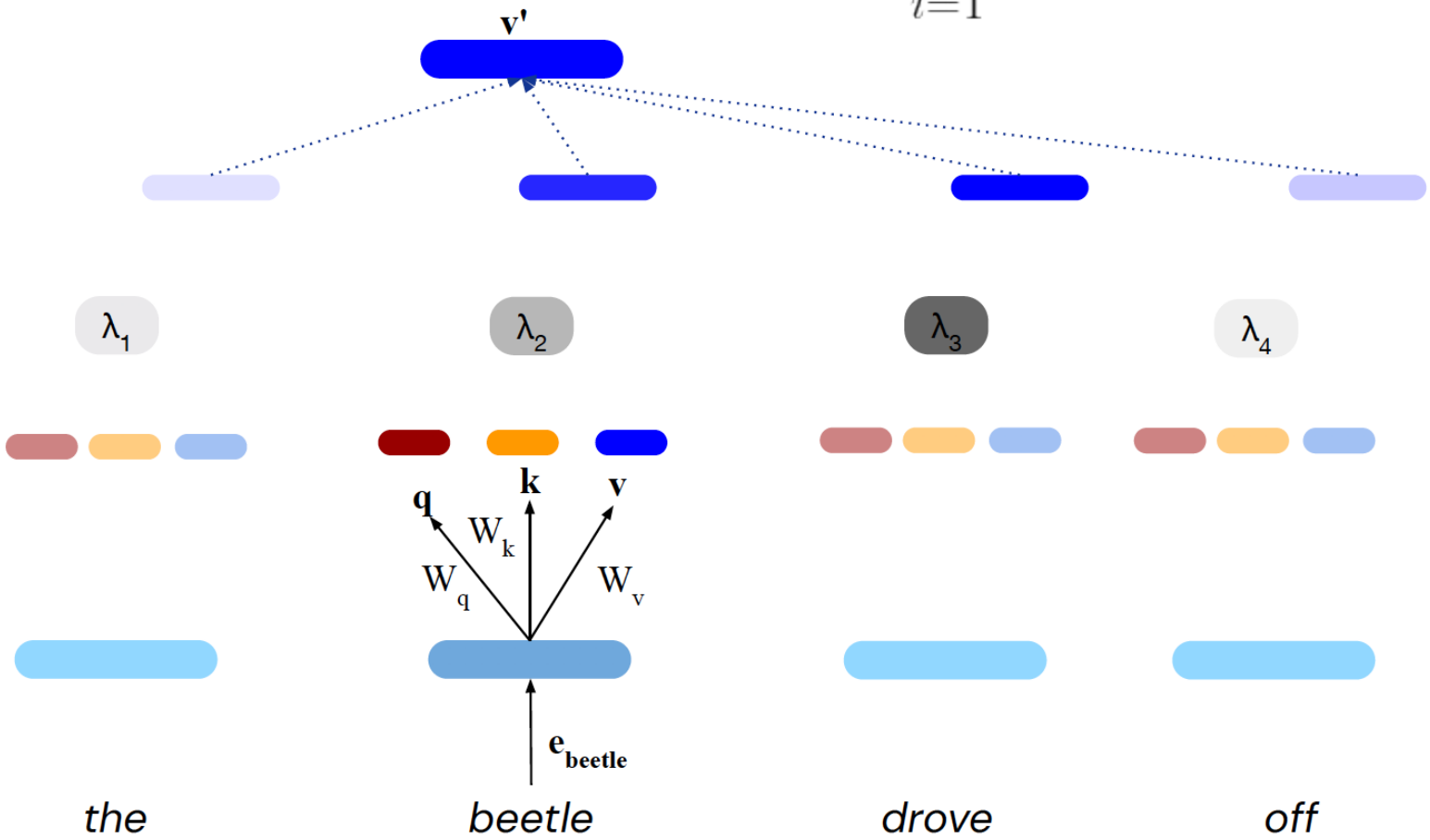


Self-attention over input embeddings

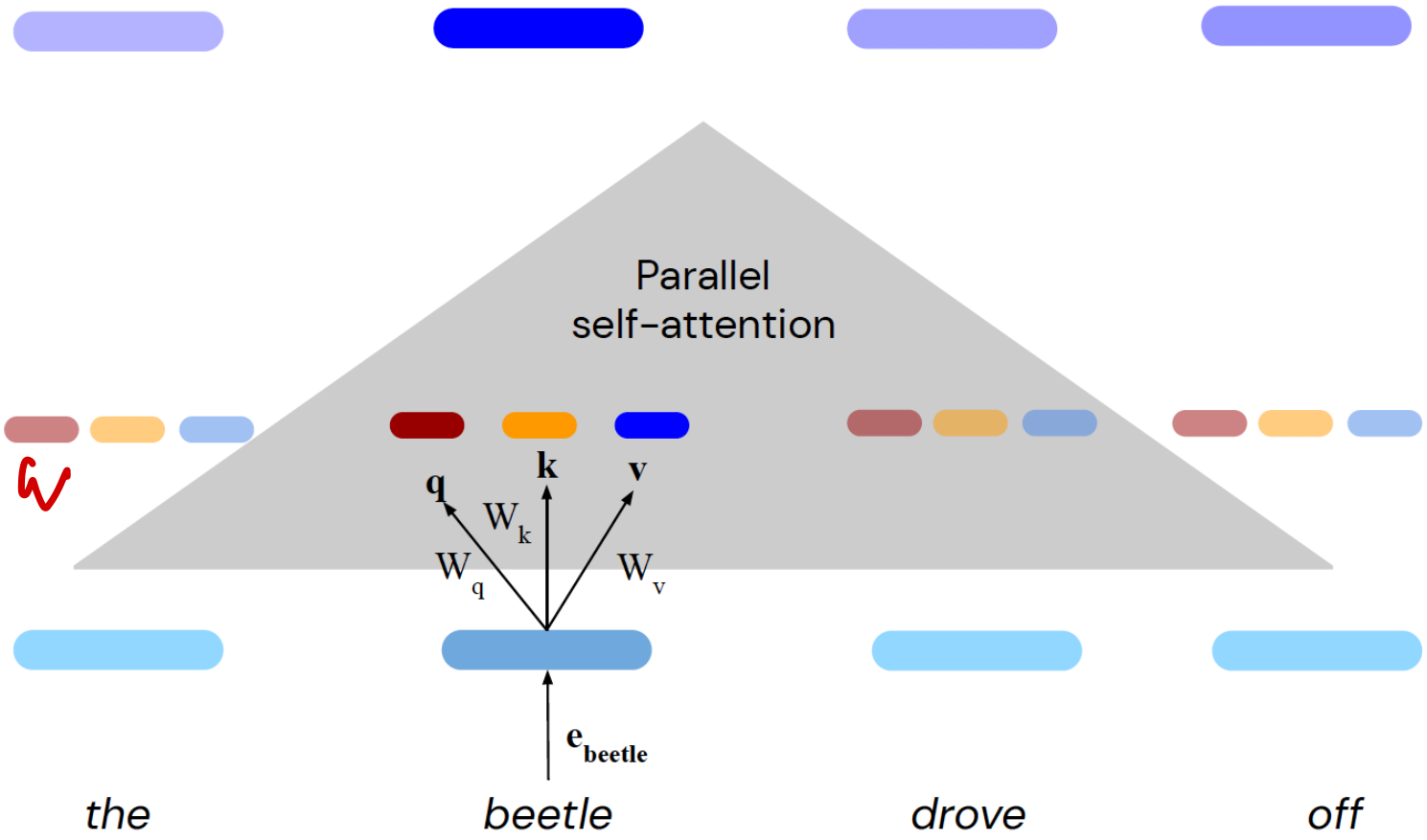


Self-attention over input embeddings

$$\mathbf{v}' = \sum_{i=1}^{4} \lambda_i \mathbf{v}_i$$



Self-attention over all words in input (in parallel)



A worked out example

(a) Input, and transformation matrices for (b) Key (c) Query (d) Value

```
Input 1: [1, 0, 1, 0]
Input 2: [0, 2, 0, 2]
Input 3: [1, 1, 1, 1]
```

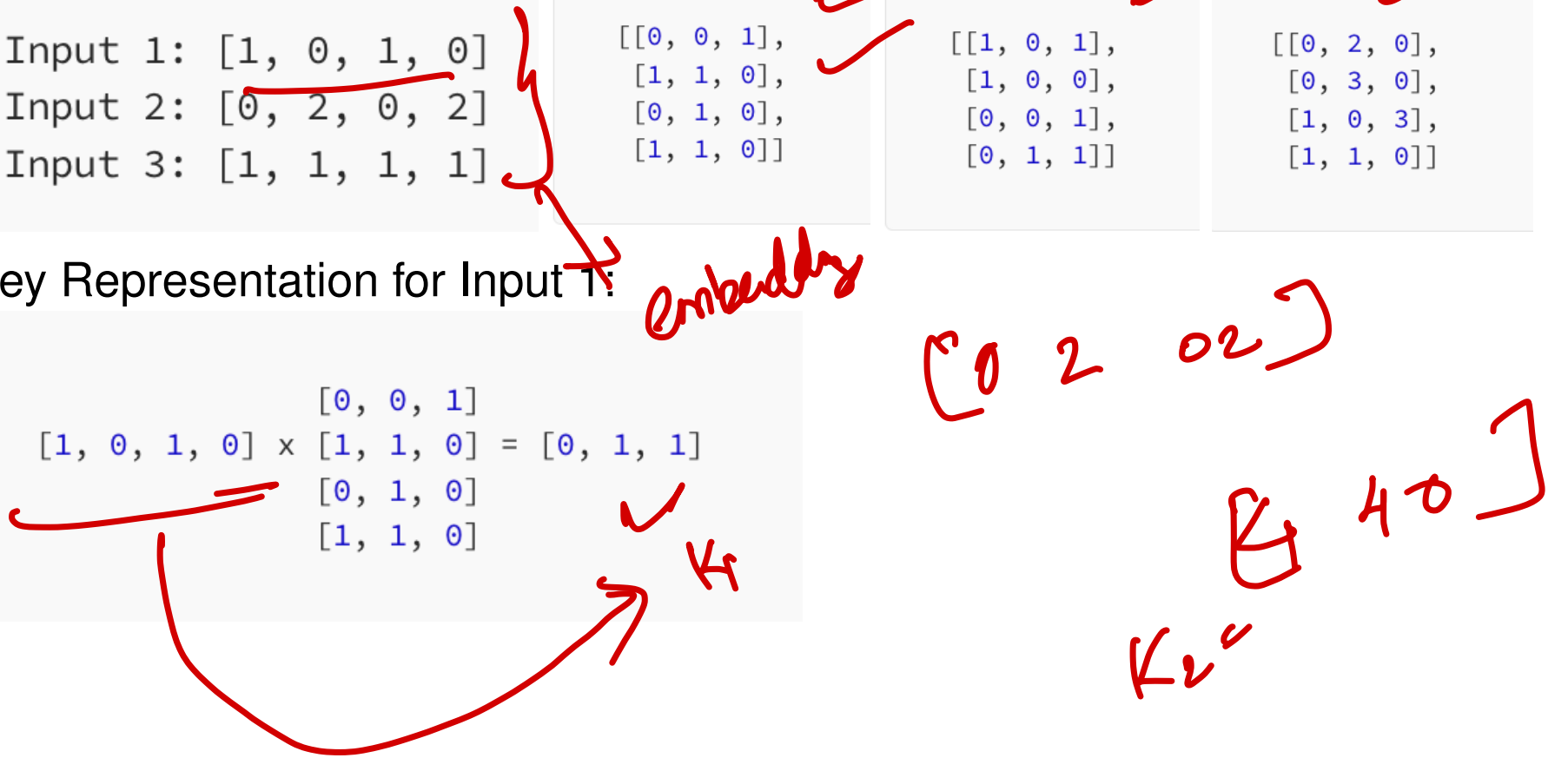
```
[[0, 0, 1],
 [1, 1, 0],
 [0, 1, 0],
 [1, 1, 0]]
```

```
[[1, 0, 1],
 [1, 0, 0],
 [0, 0, 1],
 [0, 1, 1]]
```

```
[[0, 2, 0],
 [0, 3, 0],
 [1, 0, 3],
 [1, 1, 0]]
```

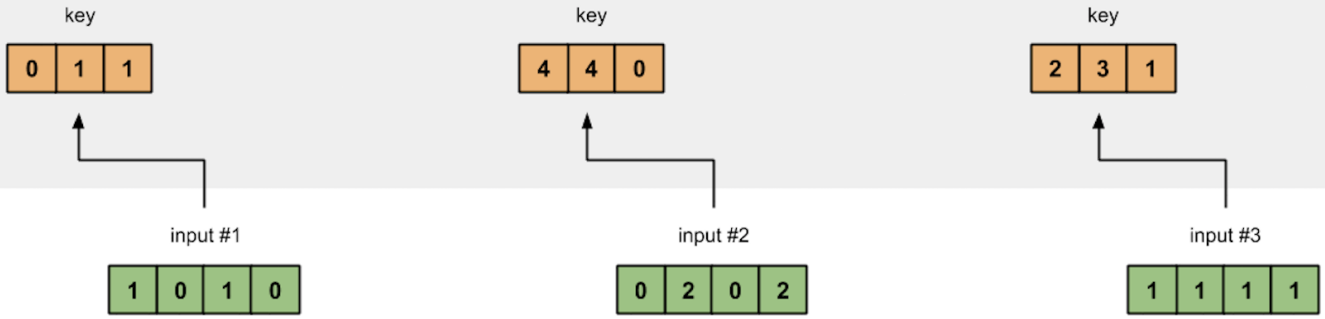
Key Representation for Input 1:

```
                [0, 0, 1]
[1, 0, 1, 0] x  [1, 1, 0]  =  [0, 1, 1]
                [0, 1, 0]
                [1, 1, 0]
```



https://towardsdatascience.com/illustrated-self-attention-2d627e33b20a

A worked out example



A worked out example

(a) Input, and transformation matrices for (b) Key (c) Query (d) Value

```
Input 1: [1, 0, 1, 0]
Input 2: [0, 2, 0, 2]
Input 3: [1, 1, 1, 1]
```

```
[[0, 0, 1],
 [1, 1, 0],
 [0, 1, 0],
 [1, 1, 0]]
```

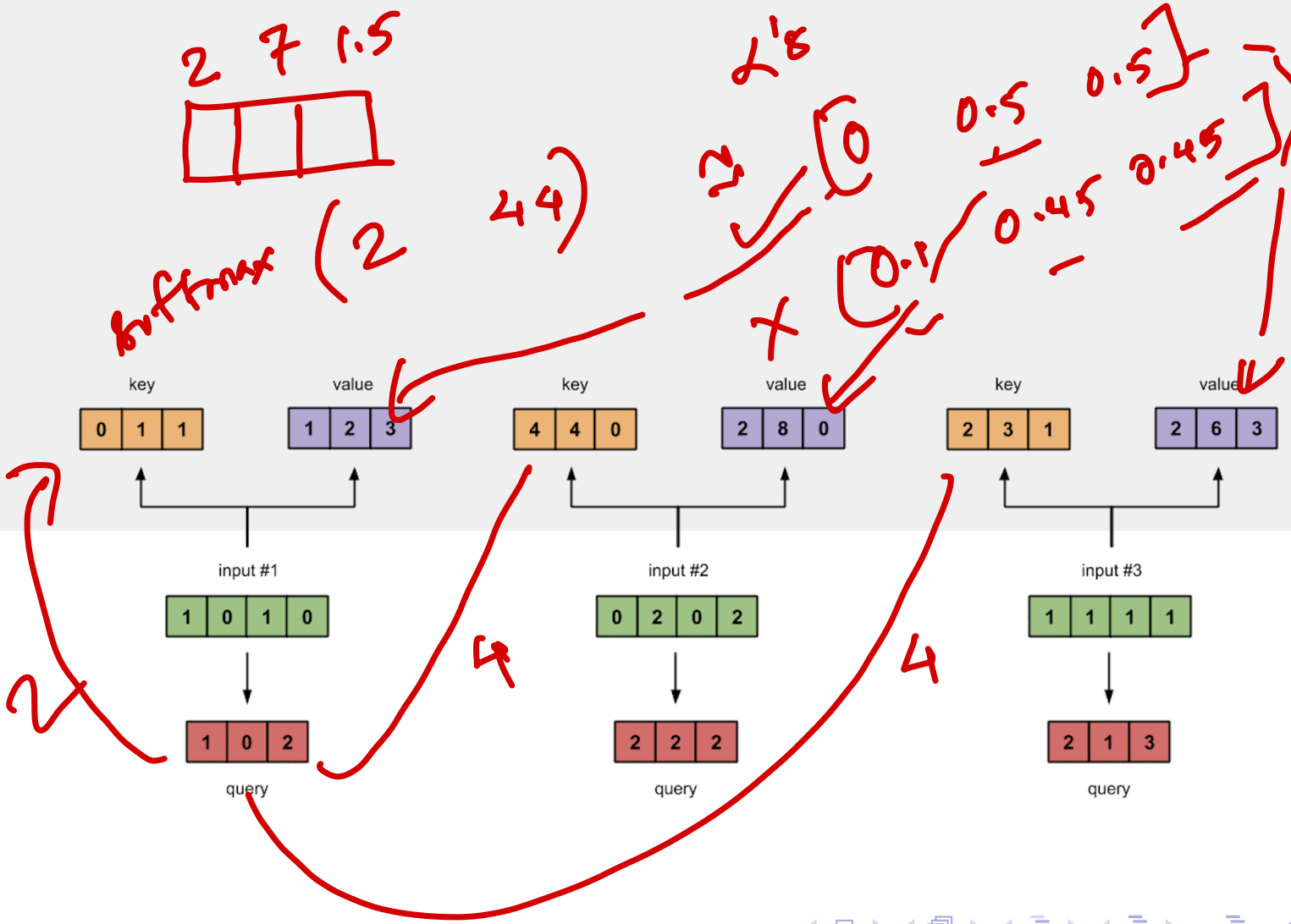
```
[[1, 0, 1],
 [1, 0, 0],
 [0, 0, 1],
 [0, 1, 1]]
```

```
[[0, 2, 0],
 [0, 3, 0],
 [1, 0, 3],
 [1, 1, 0]]
```

Query Representation:

```
                  [1, 0, 1]
[1, 0, 1, 0]      [1, 0, 0]   [1, 0, 2]
[0, 2, 0, 2]  x   [0, 0, 1] = [2, 2, 2]
[1, 1, 1, 1]      [0, 1, 1]   [2, 1, 3]
```

A worked out example



A worked out example

(a) Input, and transformation matrices for (b) Key (c) Query (d) Value

```
Input 1: [1, 0, 1, 0]
Input 2: [0, 2, 0, 2]
Input 3: [1, 1, 1, 1]
```

```
[[0, 0, 1],
 [1, 1, 0],
 [0, 1, 0],
 [1, 1, 0]]
```

```
[[1, 0, 1],
 [1, 0, 0],
 [0, 0, 1],
 [0, 1, 1]]
```

```
[[0, 2, 0],
 [0, 3, 0],
 [1, 0, 3],
 [1, 1, 0]]
```

Value Representation:

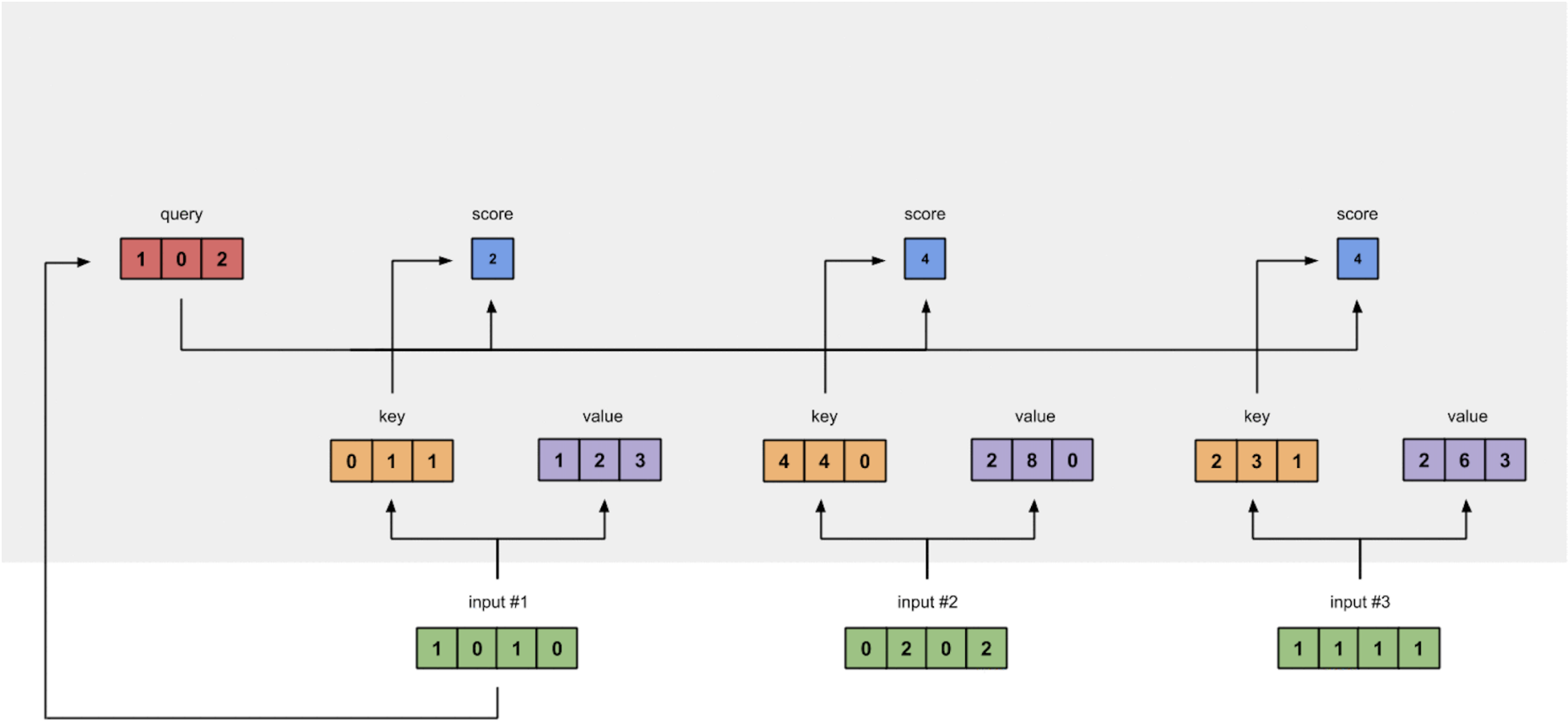
```
                [0, 2, 0]
[1, 0, 1, 0]    [0, 3, 0]    [1, 2, 3]
[0, 2, 0, 2] x  [1, 0, 3] =  [2, 8, 0]
[1, 1, 1, 1]    [1, 1, 0]    [2, 6, 3]
```

Computing Attention

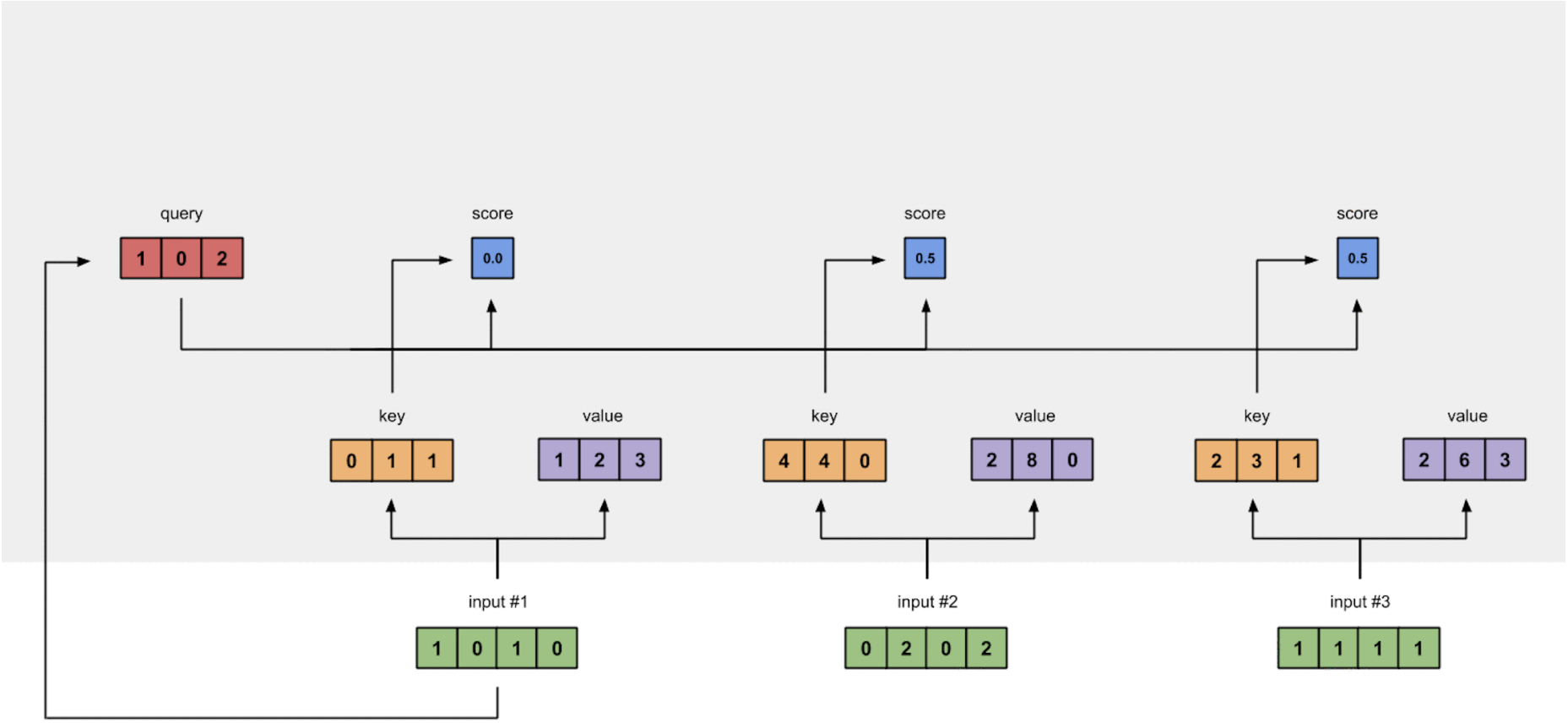
We take dot product between Input 1's query and all keys, including itself.

$$[1, 0, 2] \times \begin{matrix}[0, 4, 2] \\ [1, 4, 3] \\ [1, 0, 1]\end{matrix} = [2, 4, 4]$$

Computing Attention



Computing Attention: Taking softmax



Weighted Values and Output

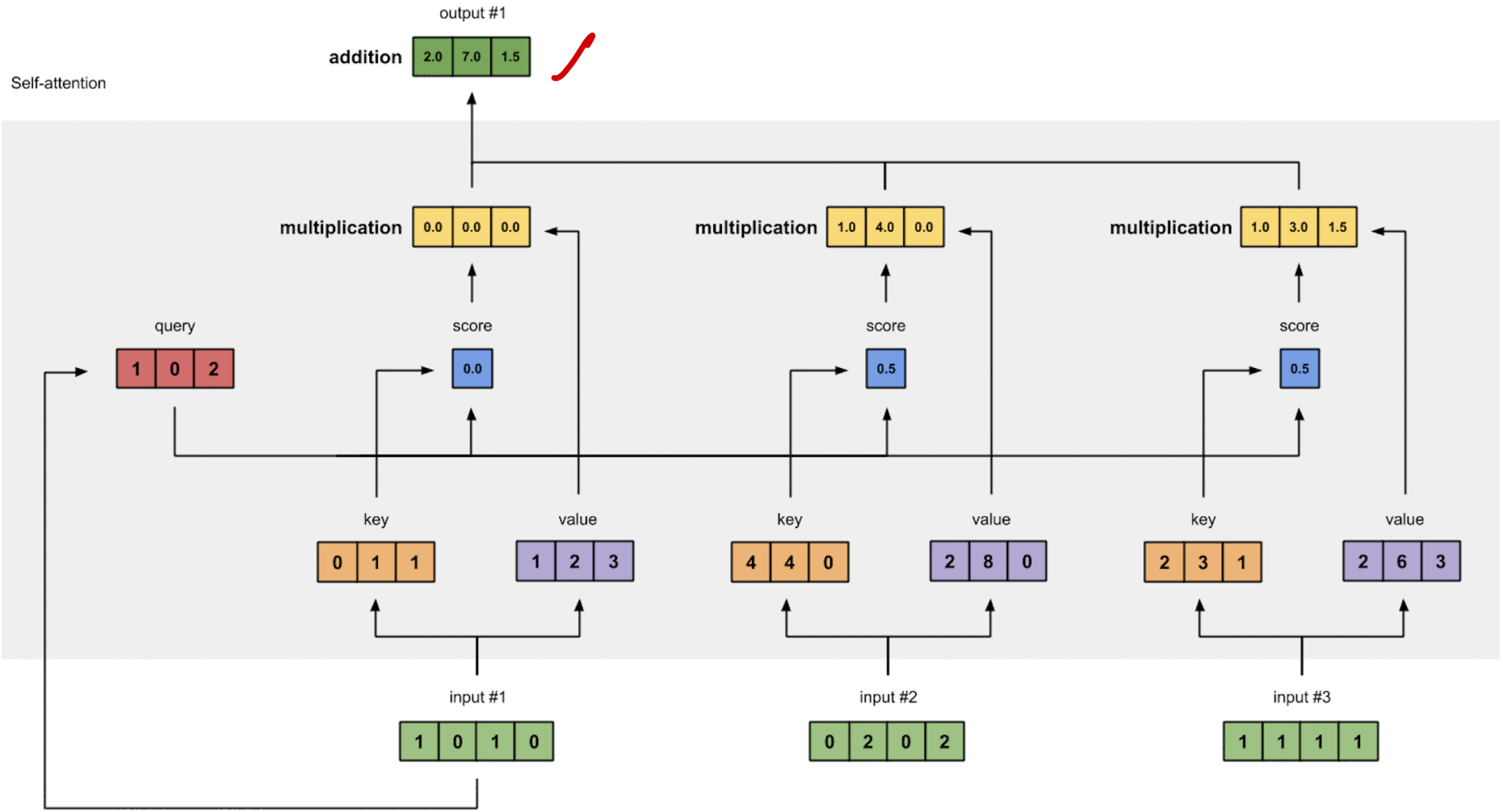
The softmaxed attention scores for each input is multiplied by its corresponding value. This results in 3 weighted values.

```
1: 0.0 * [1, 2, 3] = [0.0, 0.0, 0.0]
2: 0.5 * [2, 8, 0] = [1.0, 4.0, 0.0]
3: 0.5 * [2, 6, 3] = [1.0, 3.0, 1.5]
```

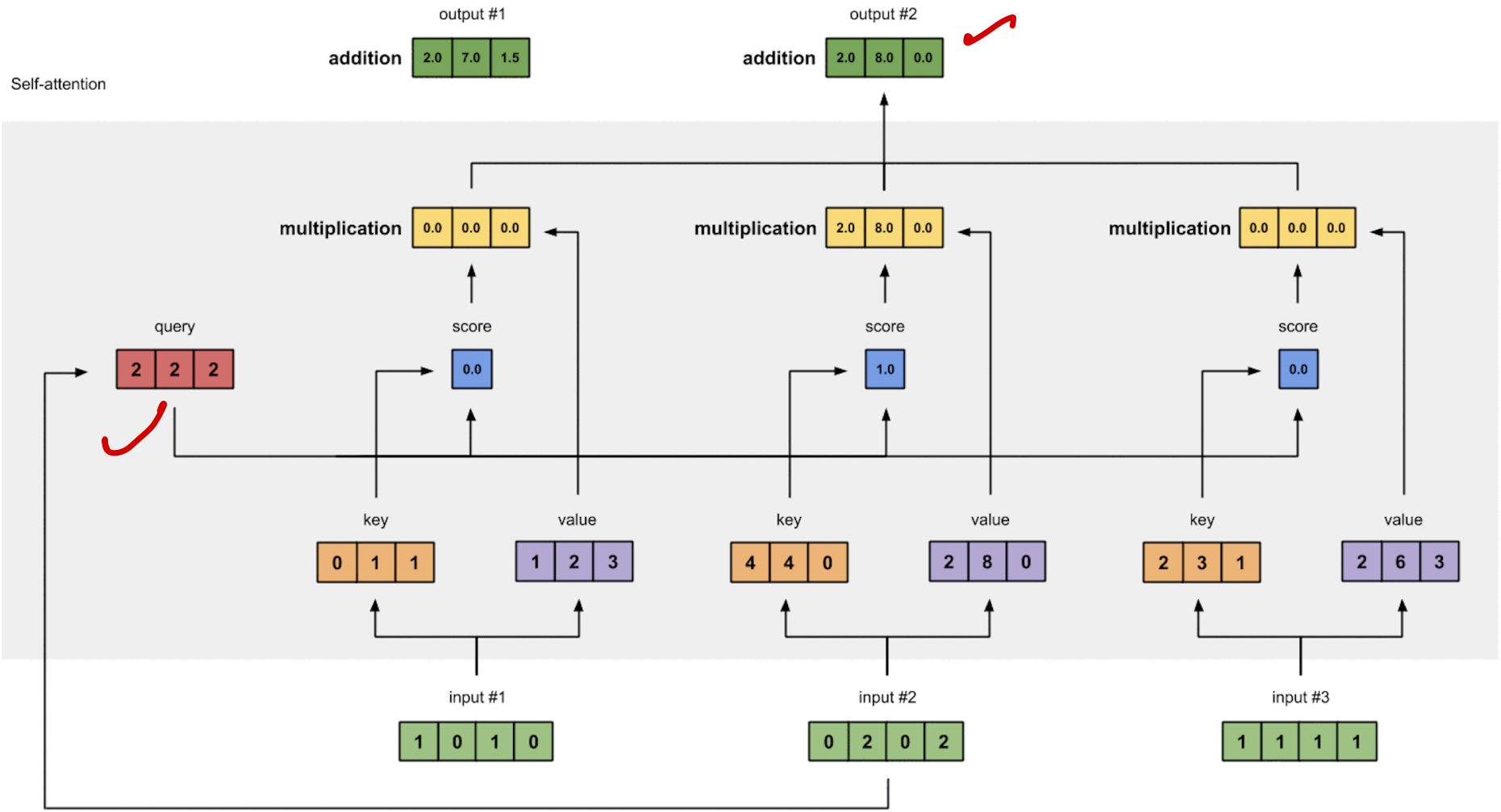
Take all the weighted values and sum them element-wise:

```
  [0.0, 0.0, 0.0]
+ [1.0, 4.0, 0.0]
+ [1.0, 3.0, 1.5]
-----------------
= [2.0, 7.0, 1.5]
```

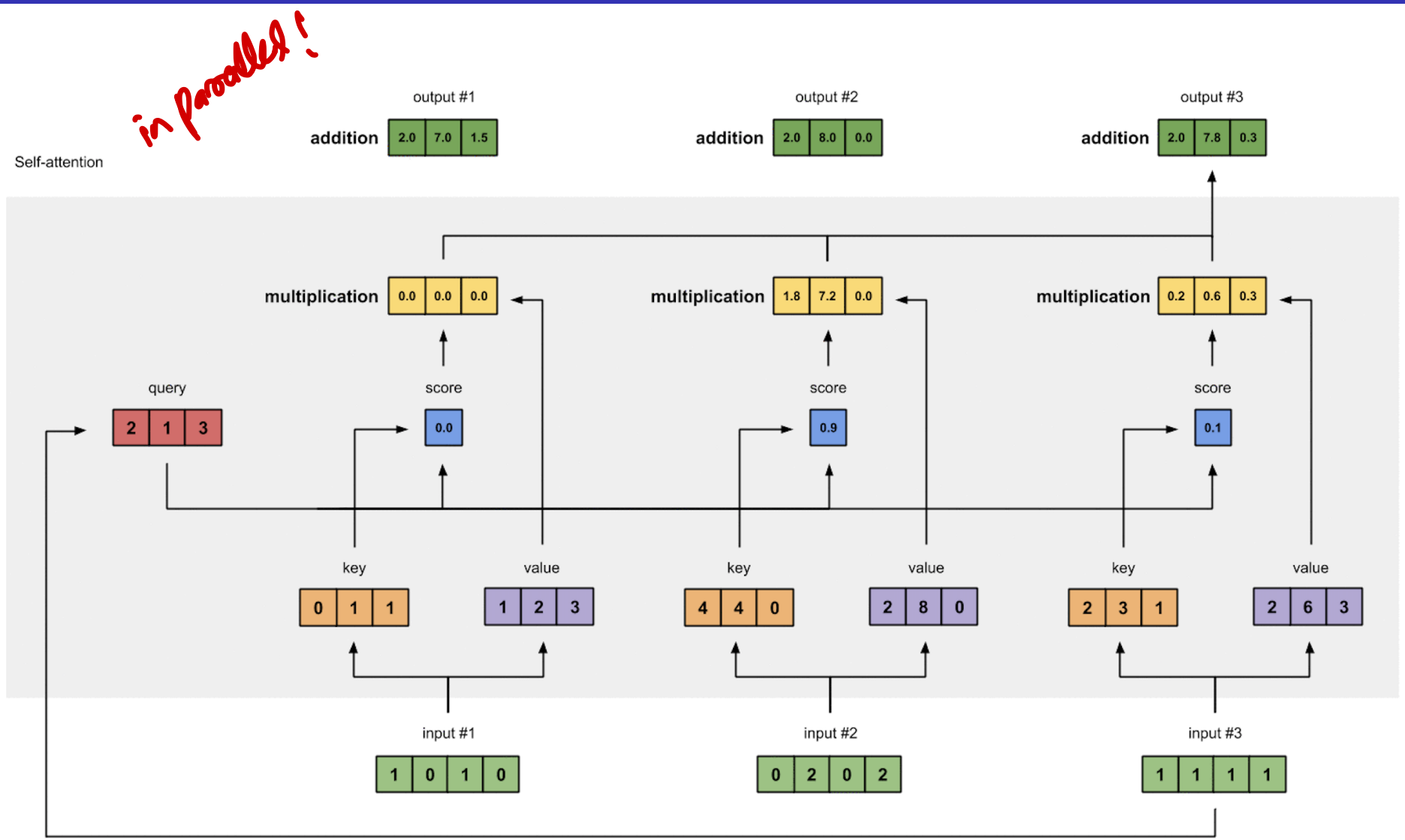
Getting Outputs



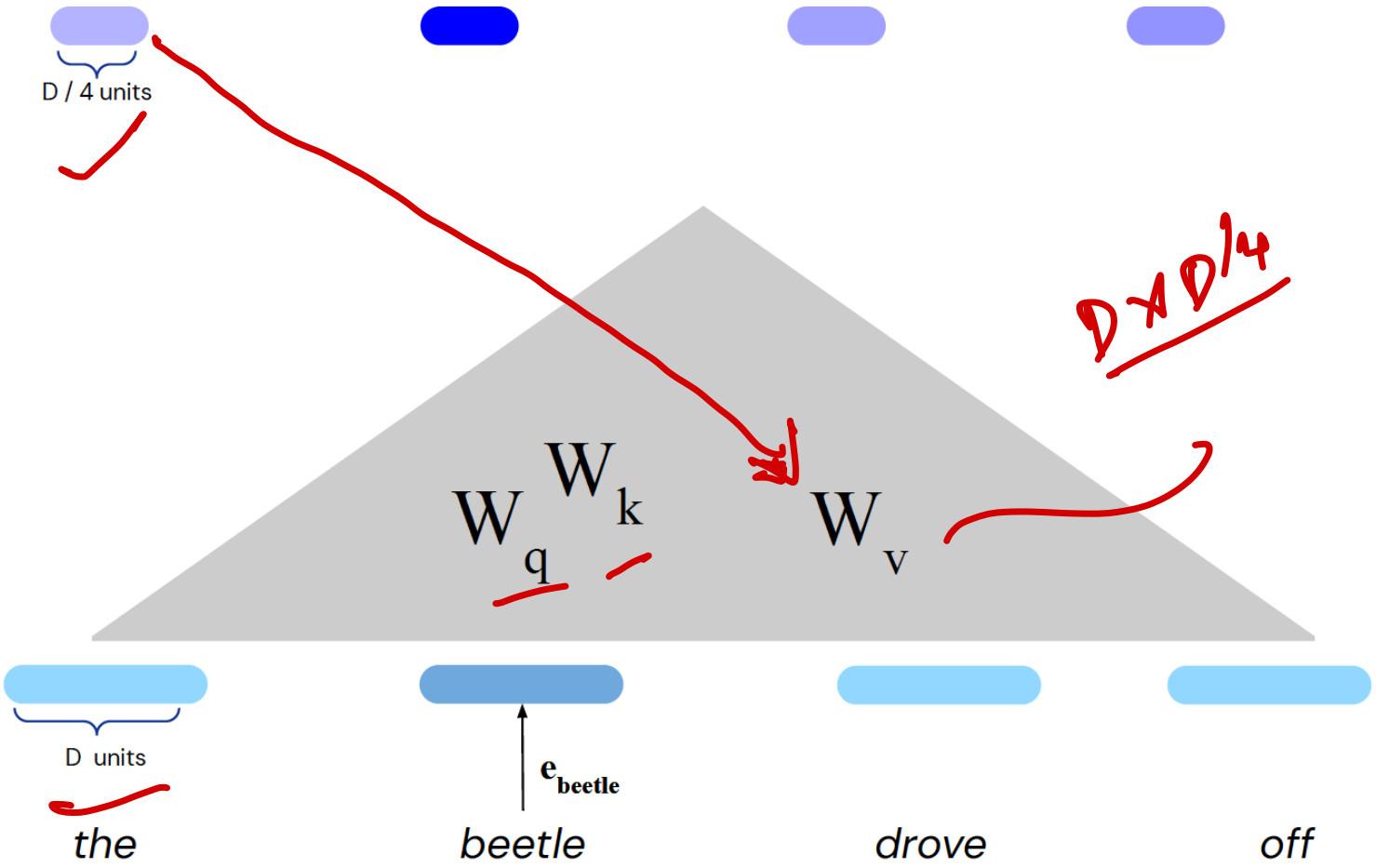
Getting Outputs



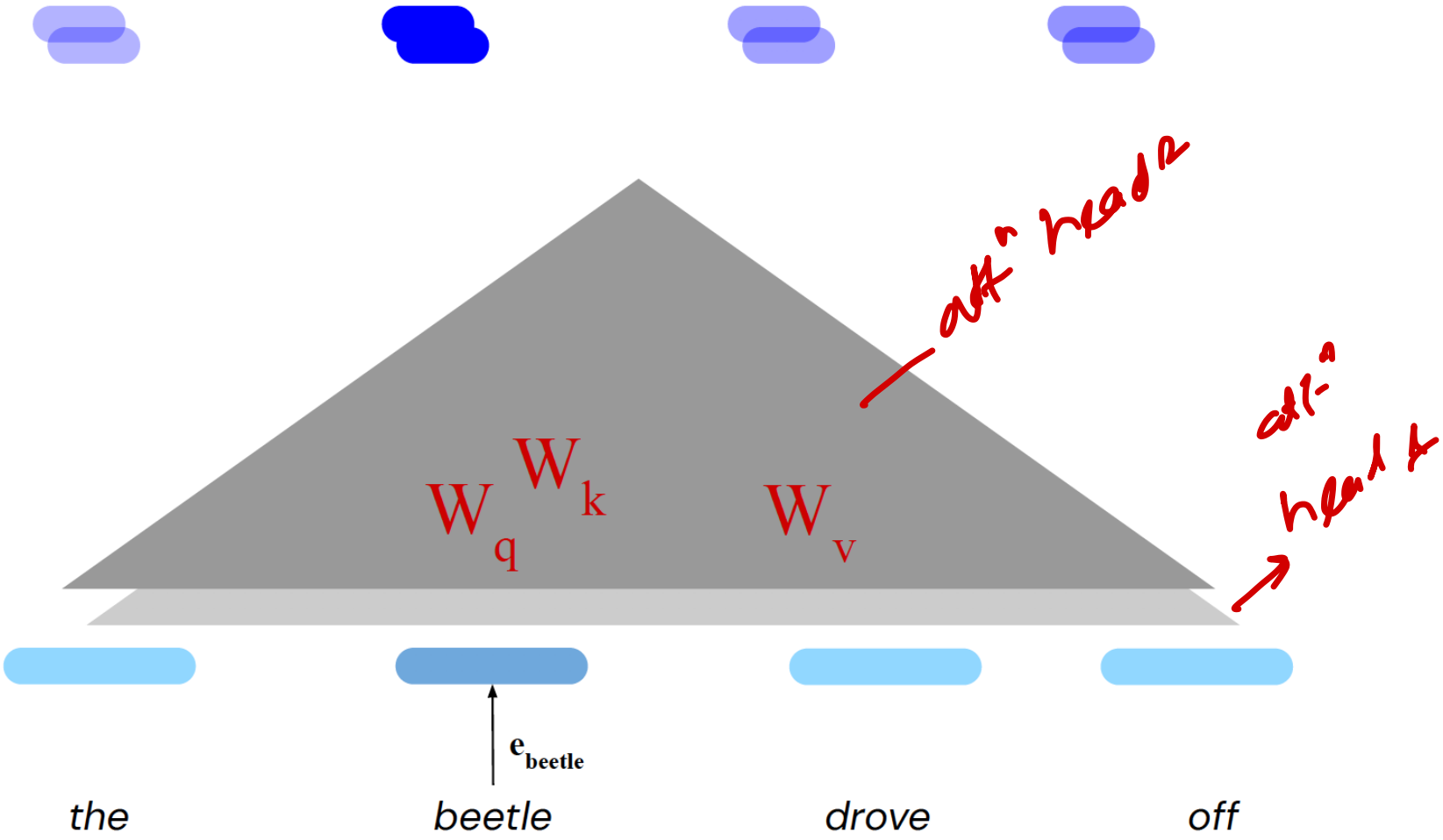
Getting Outputs



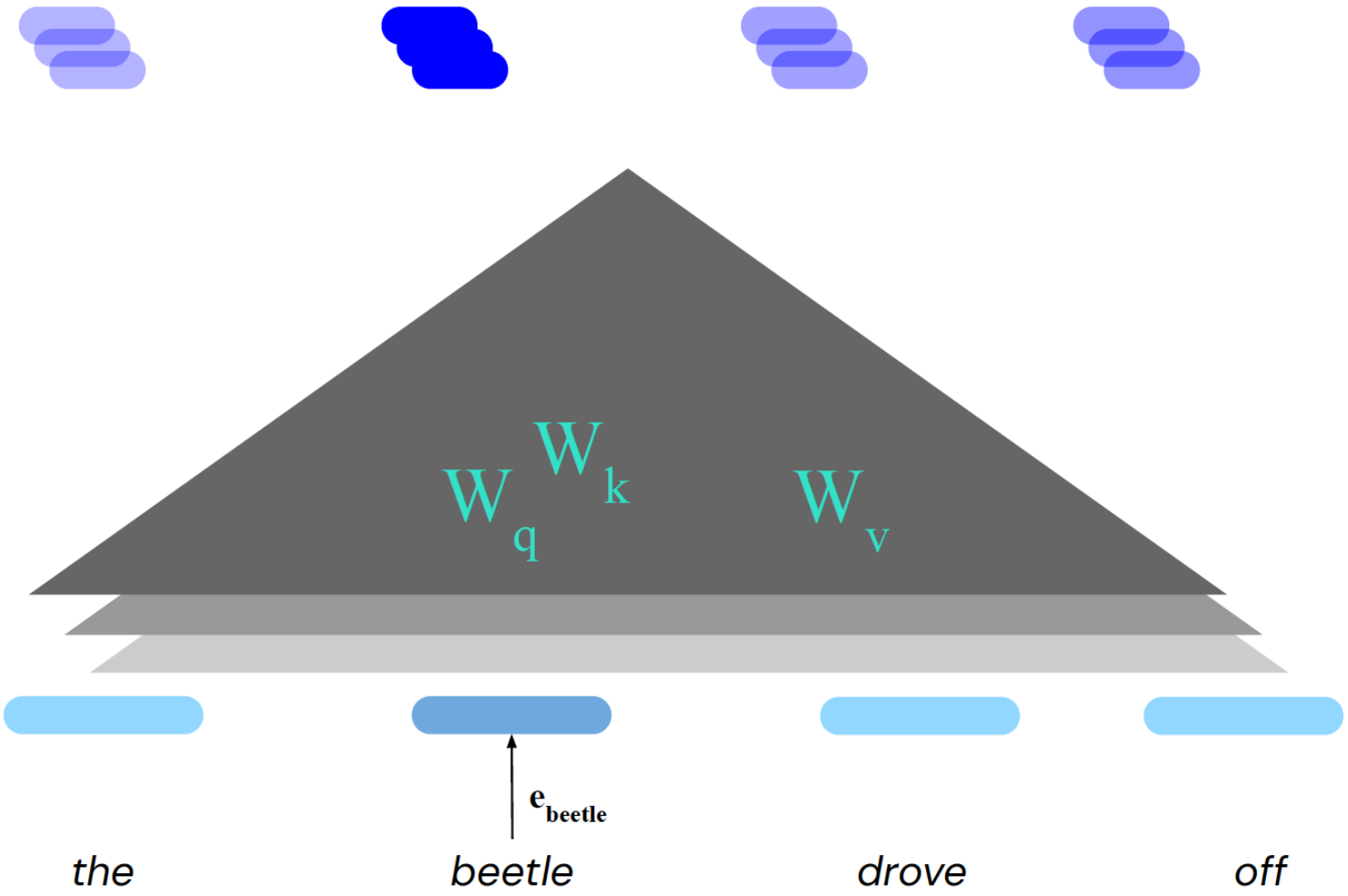
Multi-head Attention



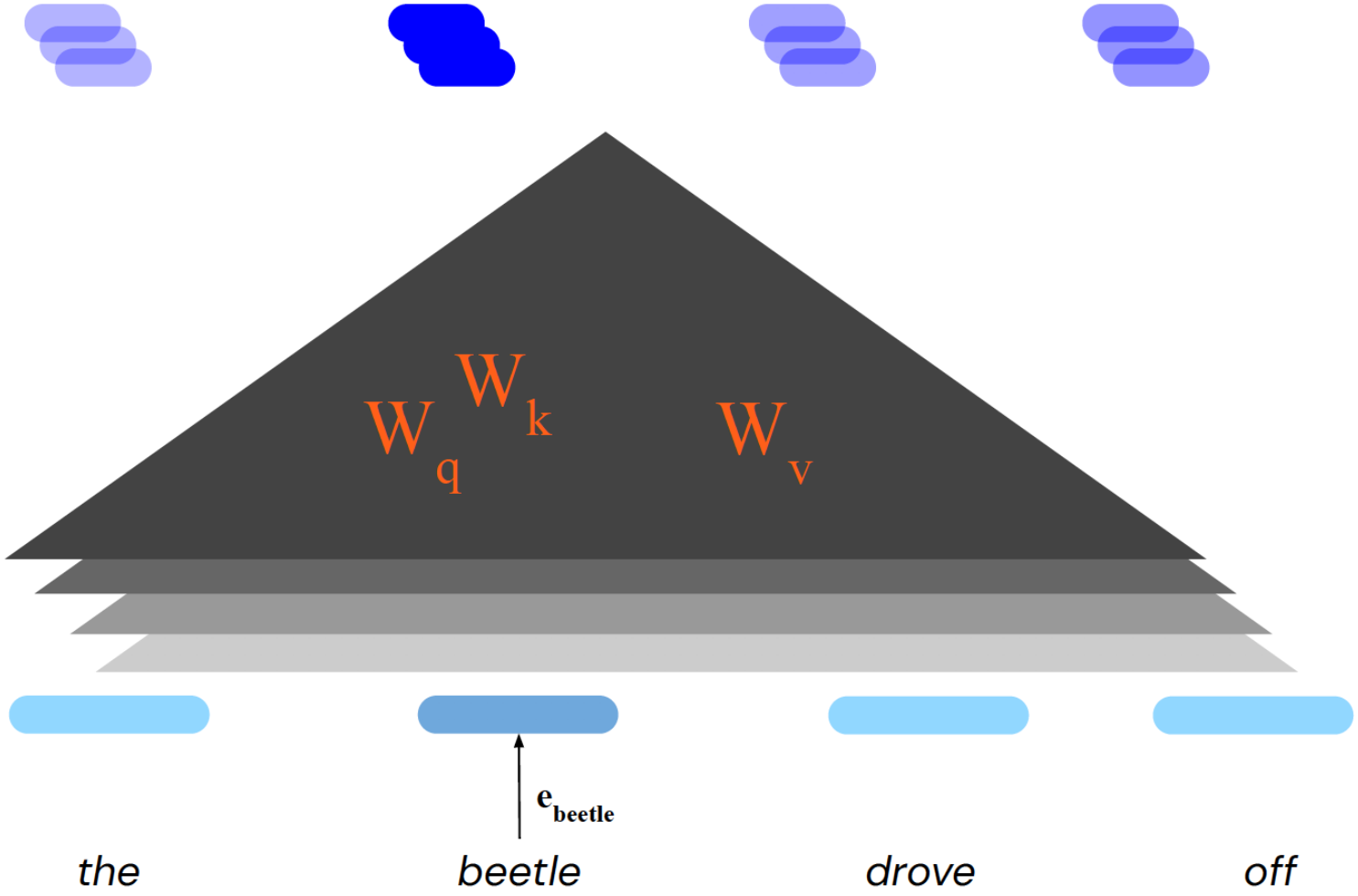
Multi-head Attention



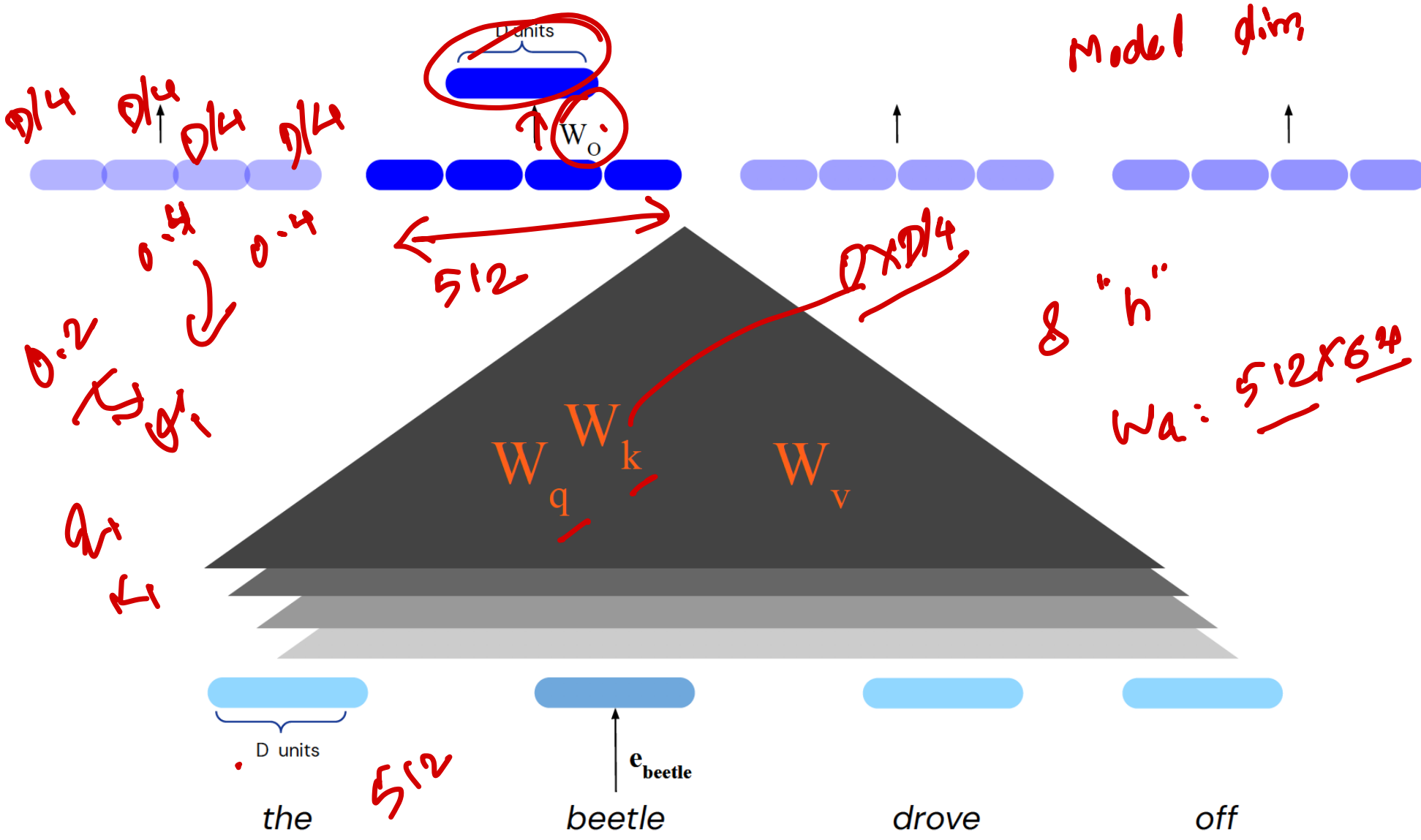
Multi-head Attention



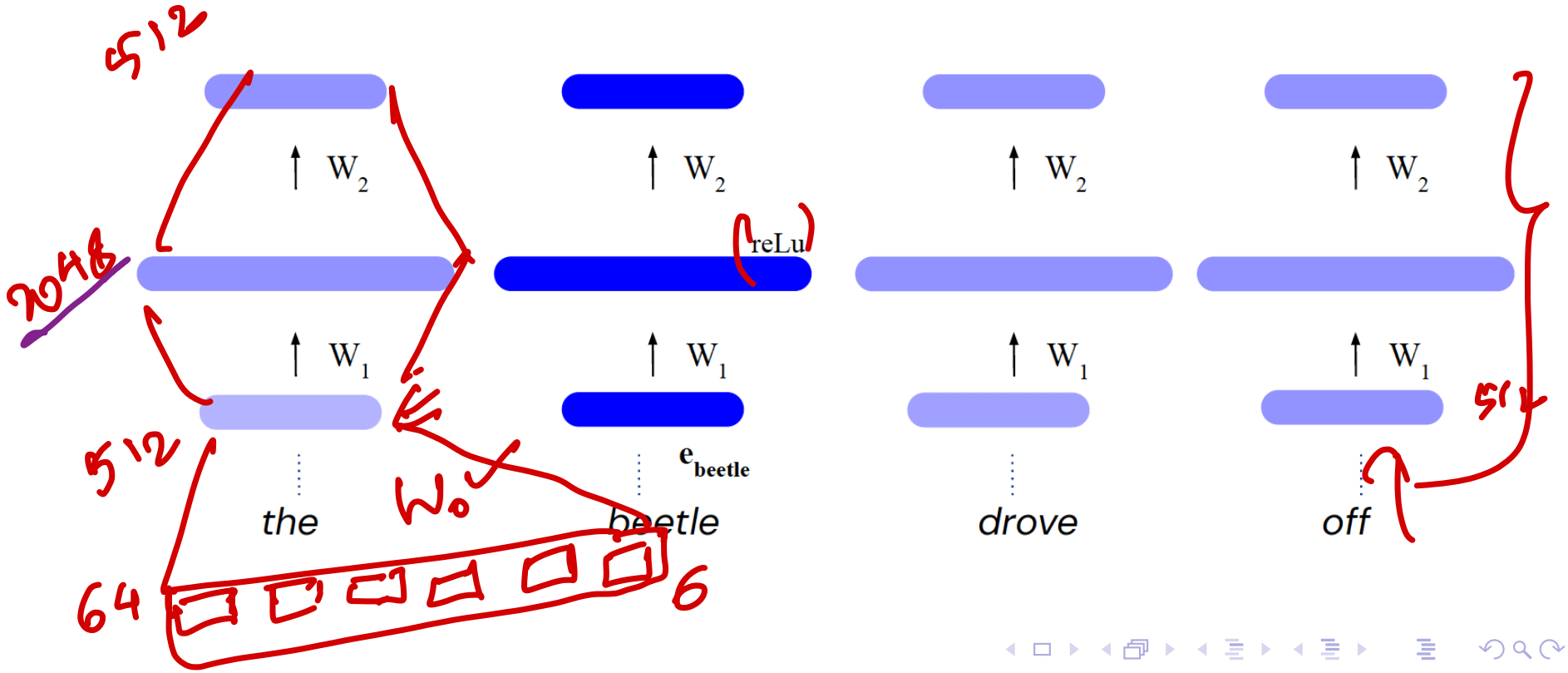
Multi-head Attention



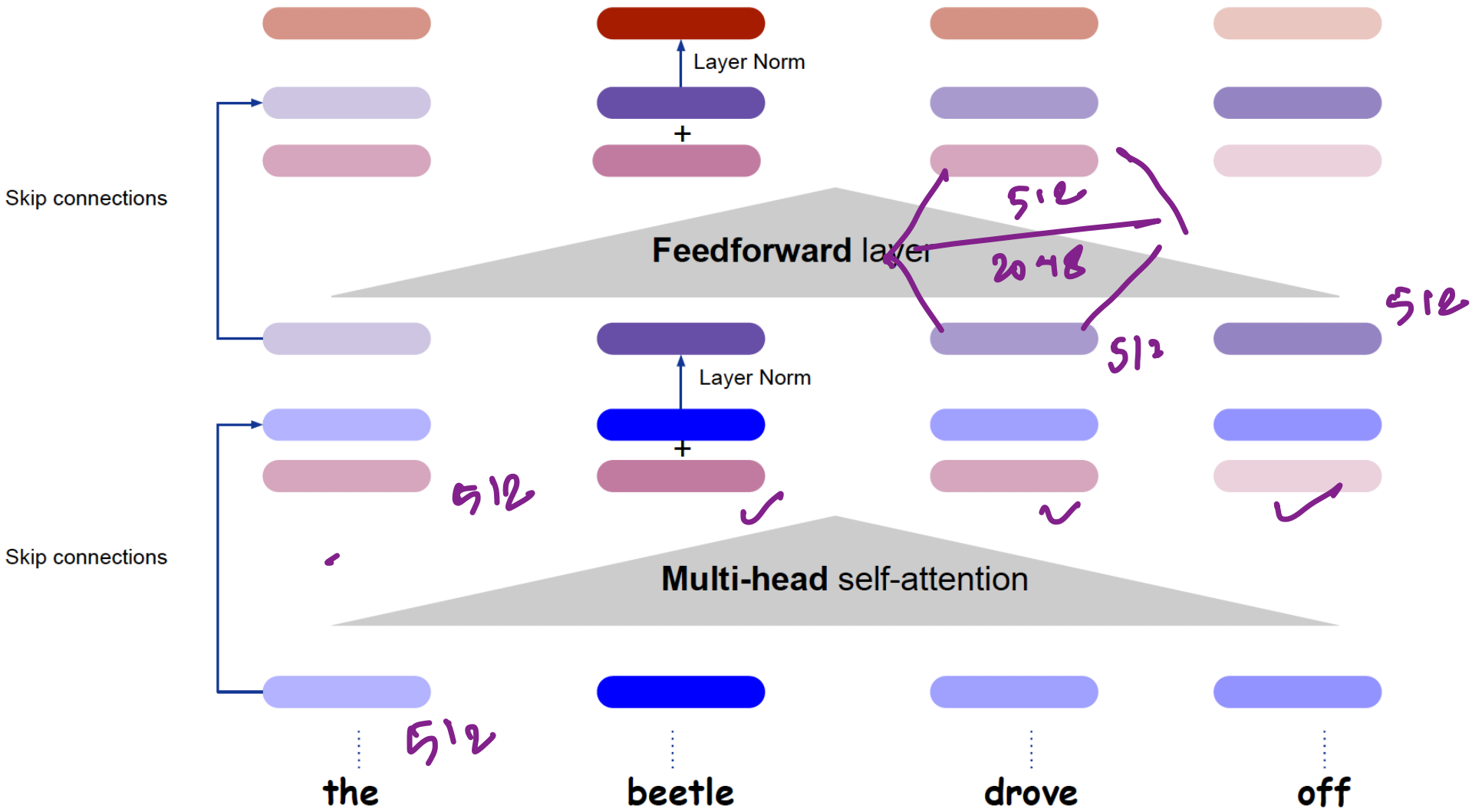
Multi-head Attention



Feed-forward Layer



A Complete Transformer Block



A residual stream view

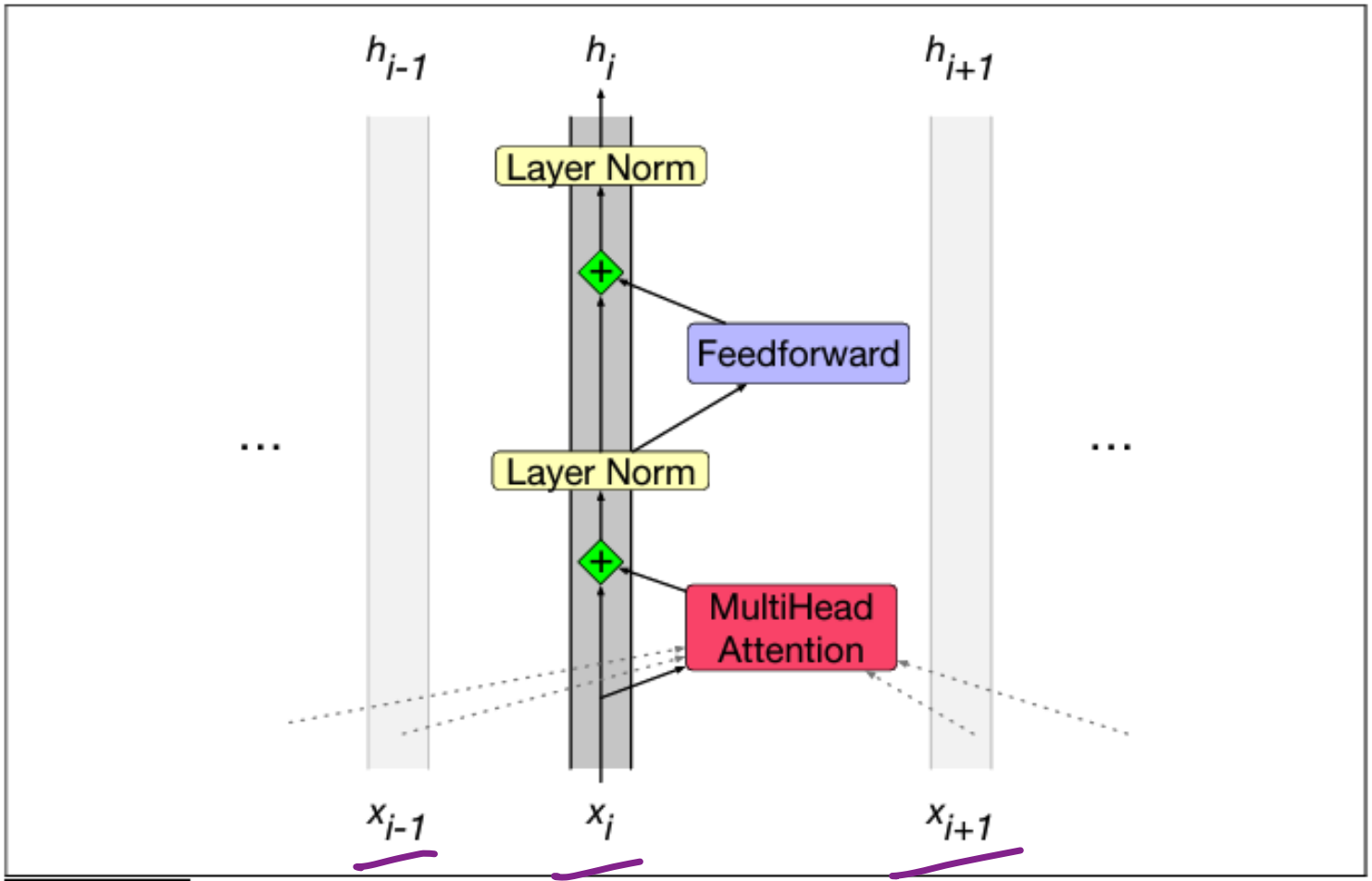


Figure 10.7 The residual stream for token x_i, showing how the input to the transformer block x_i is passed up through residual connections, the output of the feedforward and multi-head attention layers are added in, and processed by layer norm, to produce the output of this block, h_i, which is used as the input to the next layer transformer block. Note that of all the components, only the MultiHeadAttention component reads information from the other residual streams in the context.

A residual stream view:n Equations

$$
\begin{aligned}
\mathbf{t}_i^1 &= \text{MultiHeadAttention}(\mathbf{x}_i, [\mathbf{x}_1, \cdots, \mathbf{x}_N]) \\
\mathbf{t}_i^2 &= \mathbf{t}_i^1 + \mathbf{x}_i \\
\mathbf{t}_i^3 &= \text{LayerNorm}(\mathbf{t}_i^2) \\
\mathbf{t}_i^4 &= \text{FFN}(\mathbf{t}_i^3)) \\
\mathbf{t}_i^5 &= \mathbf{t}_i^4 + \mathbf{t}_i^3 \\
\mathbf{h}_i &= \text{LayerNorm}(\mathbf{t}_i^5)
\end{aligned}
$$

More on Input Embeddings

UNK

- The embeddings can be learnt along with the architecture, given a pre-defined vocabulary
- Assume there are $|V|$ words in vocabulary, and a d-dimensional embedding vector for each word, initialized as an embedding matrix E.

transformer-xl

atleast 5 times in corpus

50K unique words

OOV

More on Input Embeddings

Suppose we have a word “thanks" in the input, with index 5 in the vocabulary. It’s one-hot vector will be

```
[0 0 0 0 1 0 0 ... 0 0 0 0]
 1 2 3 4 5 6 7 ...  ... |V|
```

This one-hot vector multiplies with E and selects the relevant row for “thanks"

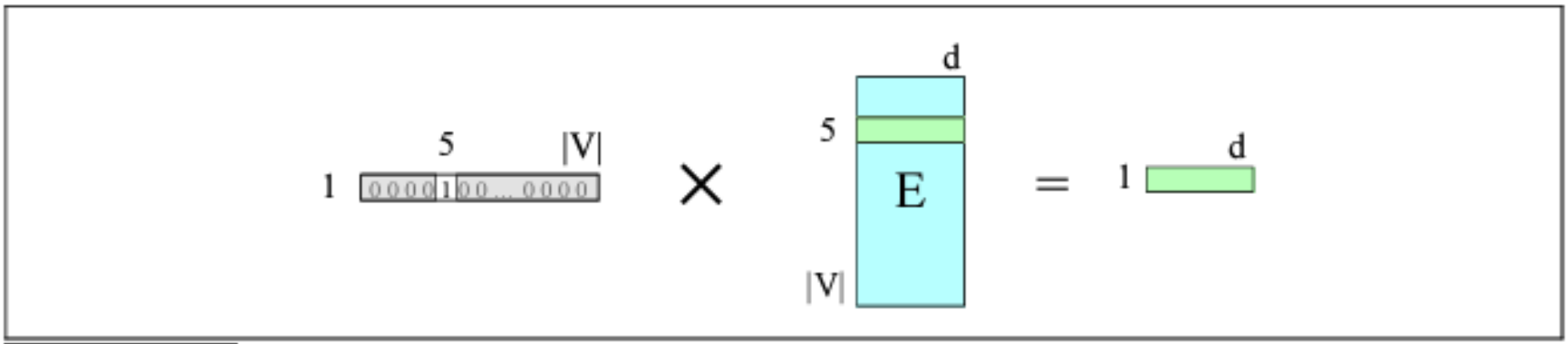


Figure 10.10 Selecting the embedding vector for word V_5 by multiplying the embedding matrix **E** with a one-hot vector with a 1 in index 5.

Positional Embeddings

'The man ate the fish' vs. 'The fish ate the man'

The word embeddings are independent of the position

Why is this needed in transformers (and not in RNNs)?

Add $d-$dimensional embeddings for each position

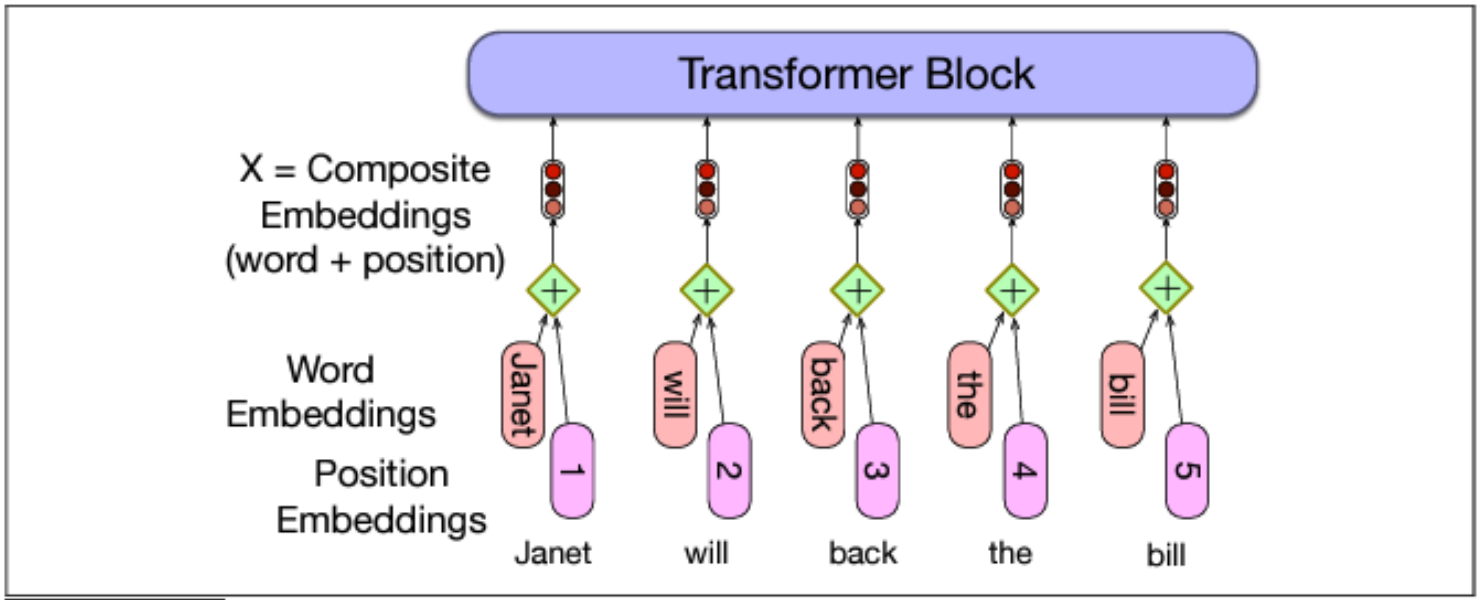


Figure 10.12 A simple way to model position: add an embedding of the absolute position to the token embedding to produce a new embedding of the same dimenionality.

Positional Encoding: Use Sinusoids

Use Sinusoids of varying periods:

$$p_i = \begin{pmatrix} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \vdots \\ \sin(i/10000^{2*\frac{d}{2}/d}) \\ \cos(i/10000^{2*\frac{d}{2}/d}) \end{pmatrix}$$