COLX-531: Neural Machine Translation

Muhammad Abdul-Mageed

muhammad.mageed@ubc.ca

Natural Language Processing Lab

The University of British Columbia

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Transformers Motivated

Background

- **Goal:** to reduce sequential computation
- **Extended Neural GPU, ByteNet**, and **ConvS2Sb:** use convnets for computing hidden representations in parallel for all input and output positions.
- Still difficult to learn dependencies between distant positions
- **Transformer:** **Constant number of operations** to relate signals from two arbitrary input or output positions
- This is at the cost of reduced effective resolution due to **averaging attention-weighted positions**
- Counteracted with **Multi-Head Attention**

Self-Attention (*aka* intra-attention)

What is self-attention?

- An attention mechanism that relates different positions of a single sequence in order to compute a representation of the sequence
- Has been successfully applied to **reading comprehension, abstractive summarization, textual entailment**, and **learning task-independent sentence representations**

Self-Attention

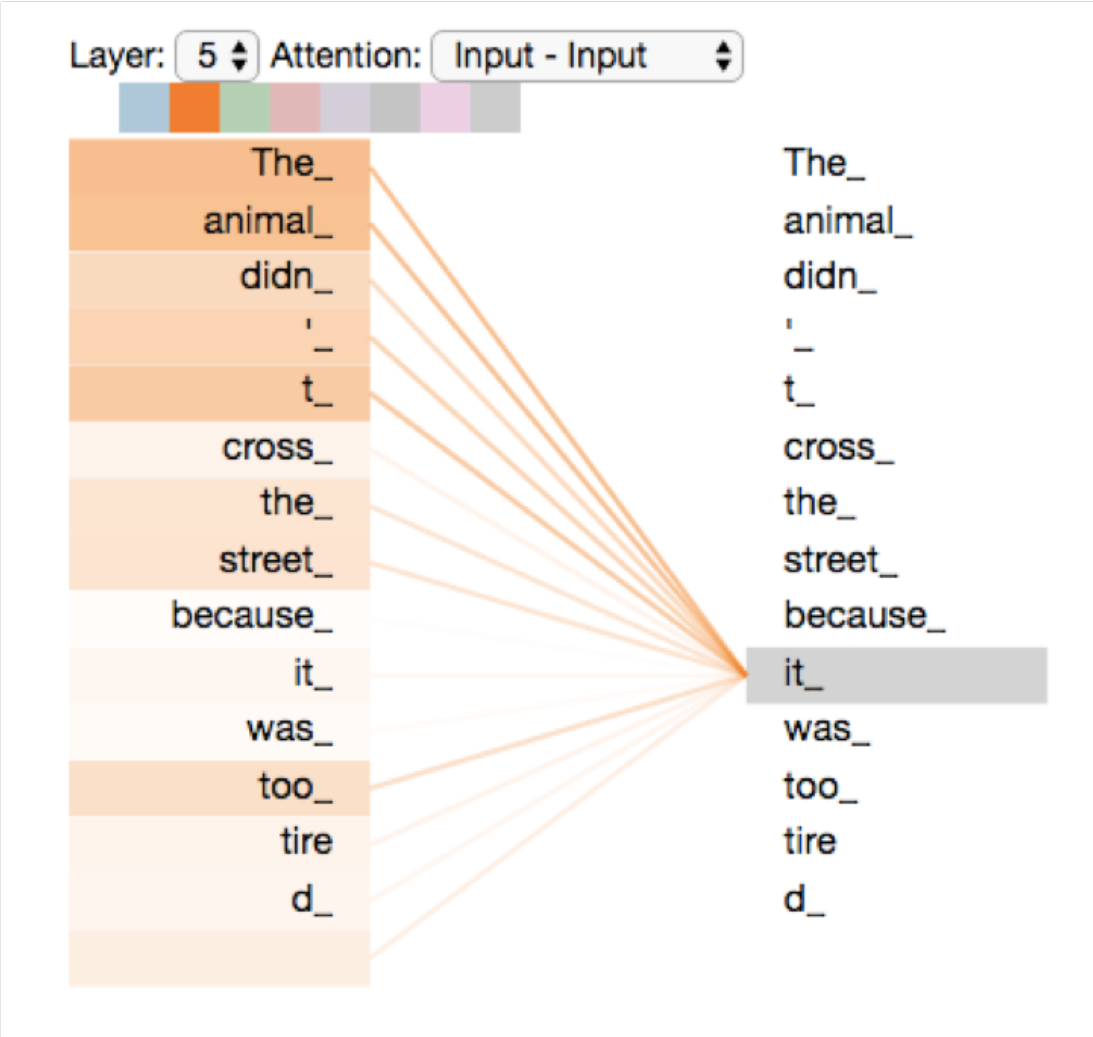


Figure: Self-attention. (Source: `http://jalammar.github.io`).

Transformer Architecture

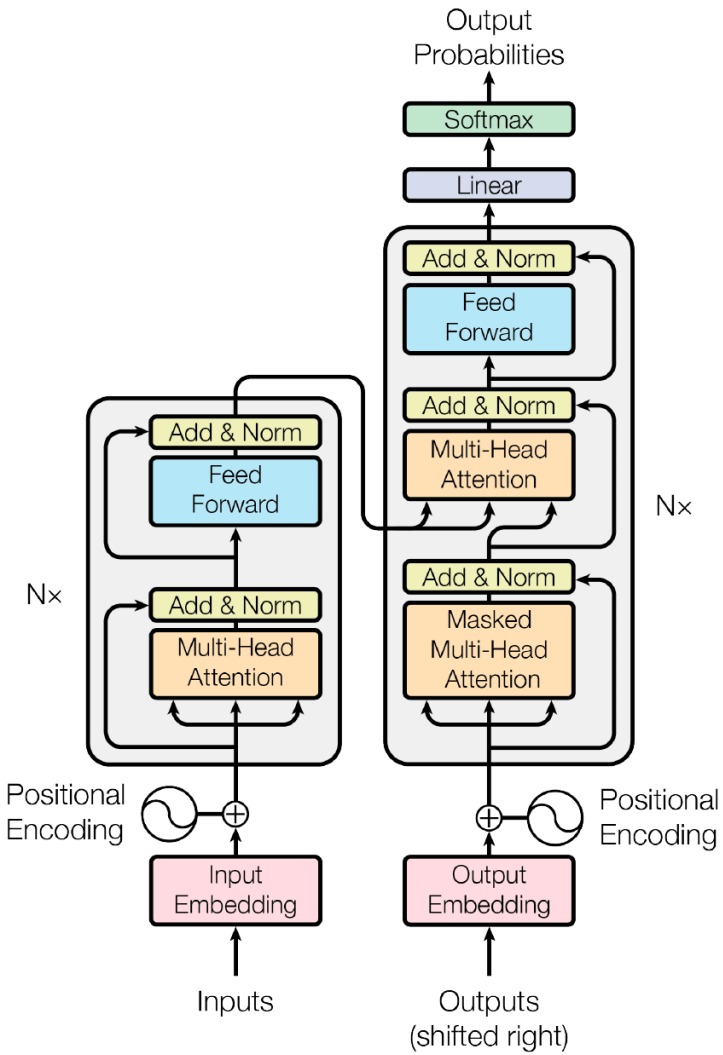


Figure: Transformer: Model Architecture.

Recall: Basic Encoder Decoder Idea

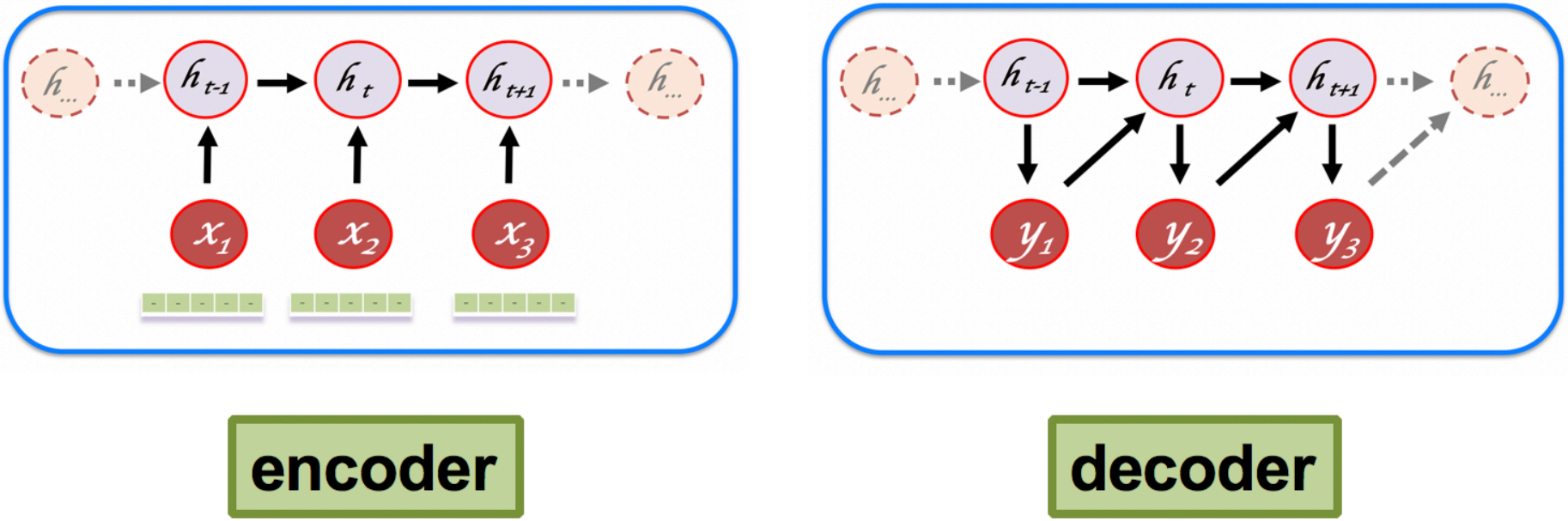


Figure: Transformer has an encoder and a decoder.

Transformer is based on encoder-decoder design

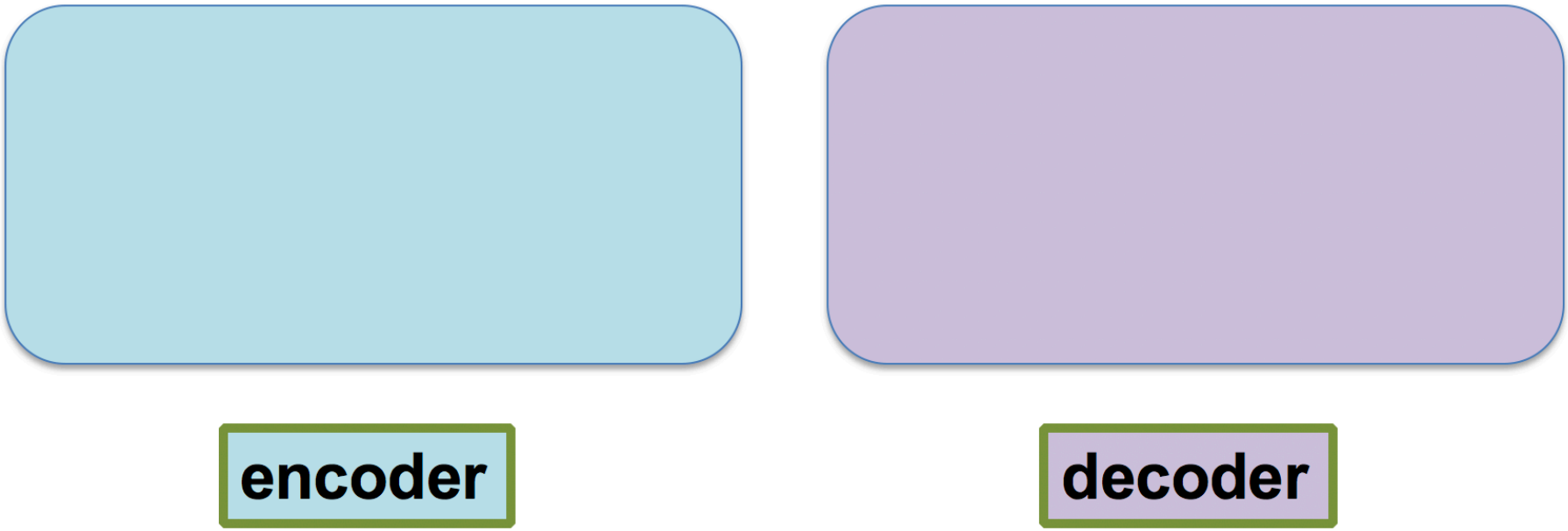


Figure: Transformer has an encoder and a decoder, **each with specialized building blocks**.

Encoder-Decoder

Basic Encoder-Decoder Operations

- **Encoder** maps an input sequence of symbol representations $(x_1, \ldots, x_n)$ to a sequence of continuous representations $\mathbf{z} = (z_1, \ldots, z_n)$.
- Given $\mathbf{z}$, the **decoder** then generates an output sequence $(y_1, \ldots, y_m)$ of symbols one element at a time.
- **Auto-regressive:** At each step the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next.

Input and Output **Embeddings**

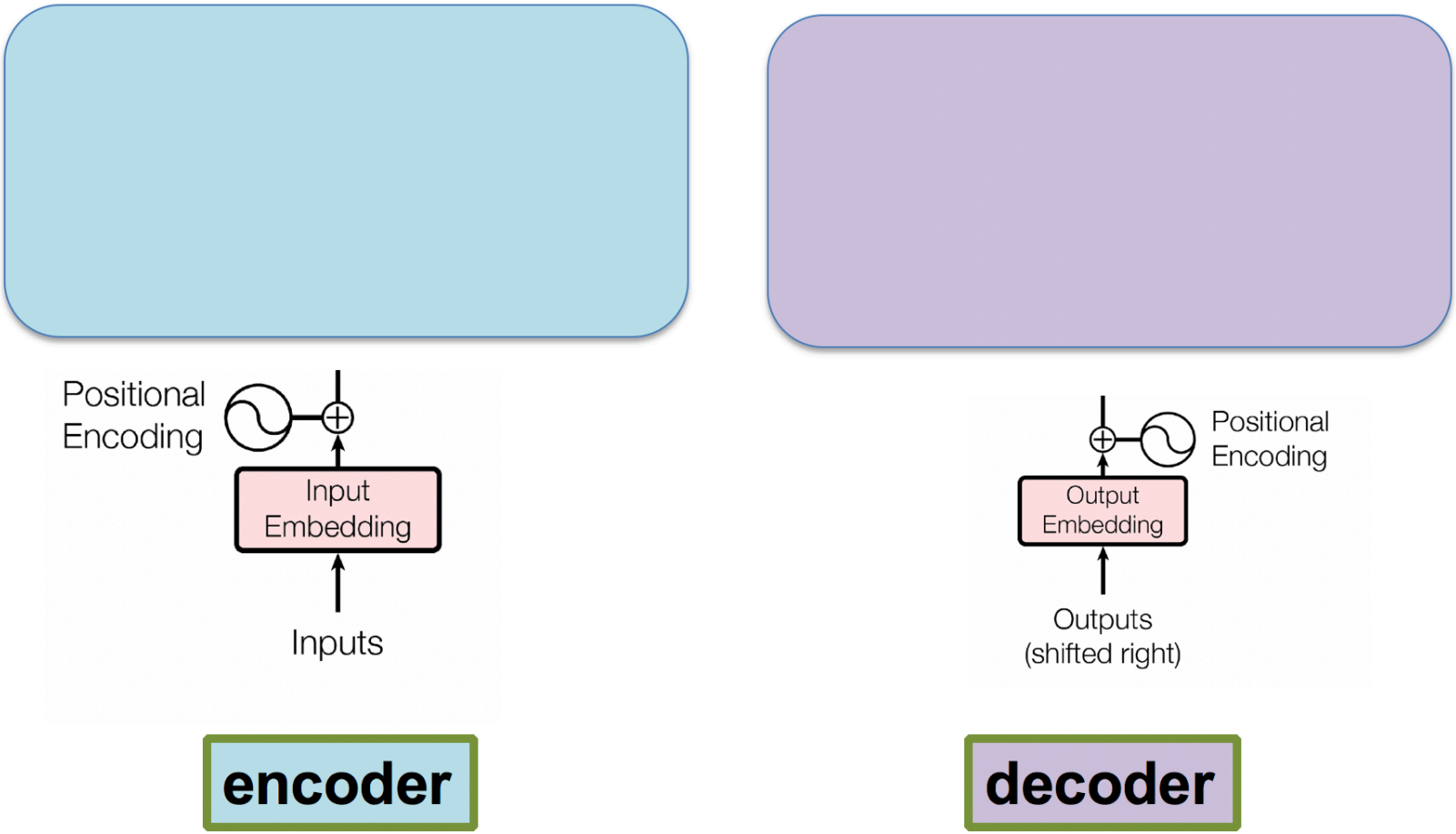


Figure: Embeddings for Encoder and Decoder.

Encoding Word Position

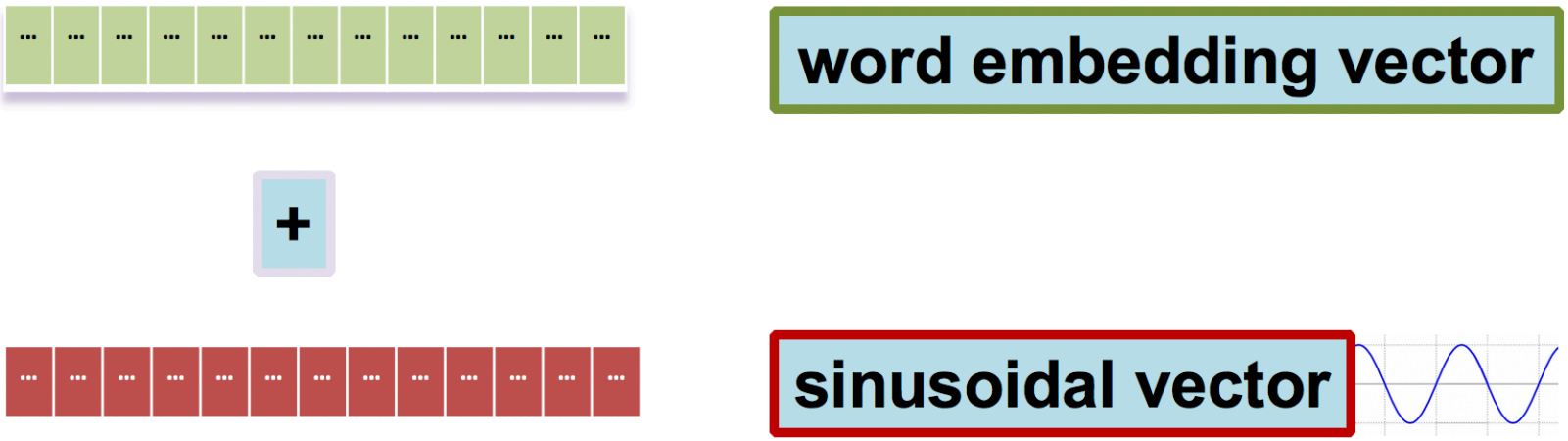


Figure: Simple Addition for encoding word positions

Positional Encoding

Recipe for PE

- **Positional encodings** added to the **input embeddings** at the bottoms of encoder and decoder stacks
- PEs ***have the same dimension*** d_{model} as the embeddings, so the two can be summed:
  $PE(pos, 2i) = sin(pos/10000^{2i/d_{model}})$ &
  $PE(pos, 2i+1) = cos(pos/10000^{2i/d_{model}})$
  where pos is the position and i is the dimension.

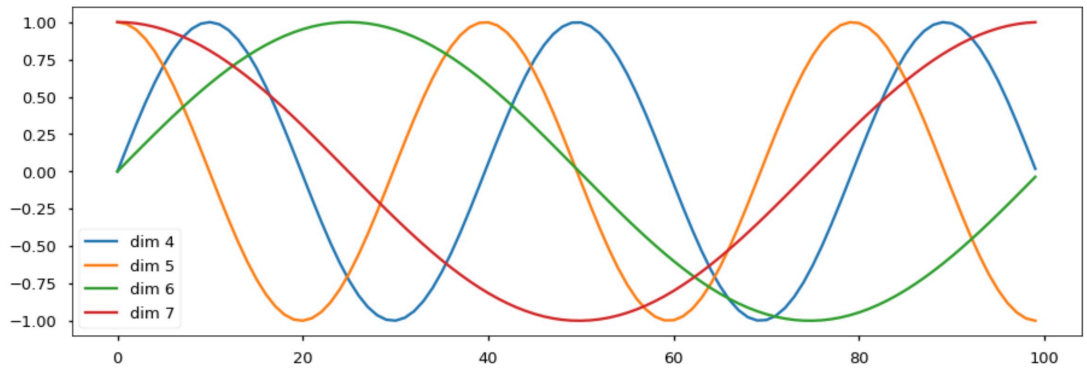


Figure: `https://nlp.seas.harvard.edu/2018/04/03/attention.html`.

Positional Encoding Example

PE Example

- For w at position $pos \in [0, L-1]$, in sequence $s = (w_1, \ldots w_4)$, with 4-dimensional embedding e_w and $d_{model} = 4$:

$$
\begin{aligned}
e'_w &= e_w + \left[sin\left(\frac{pos}{10000^0}\right), cos\left(\frac{pos}{10000^0}\right), sin\left(\frac{pos}{10000^{2/4}}\right), cos\left(\frac{pos}{10000^{2/4}}\right) \right] \\
&= e_w + \left[sin\left(pos\right), cos\left(pos\right), sin\left(\frac{pos}{100}\right), cos\left(\frac{pos}{100}\right) \right]
\end{aligned}
$$

where the formula for positional encoding is as follows

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

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

with $d_{model} = 512$ (thus $i \in [0, 255]$) in the original paper.

Figure: Source: `https://datascience.stackexchange.com/questions/51065/what-is-the-positional-encoding-in-the-transformer-model`.

Positional Encoding Illustrated

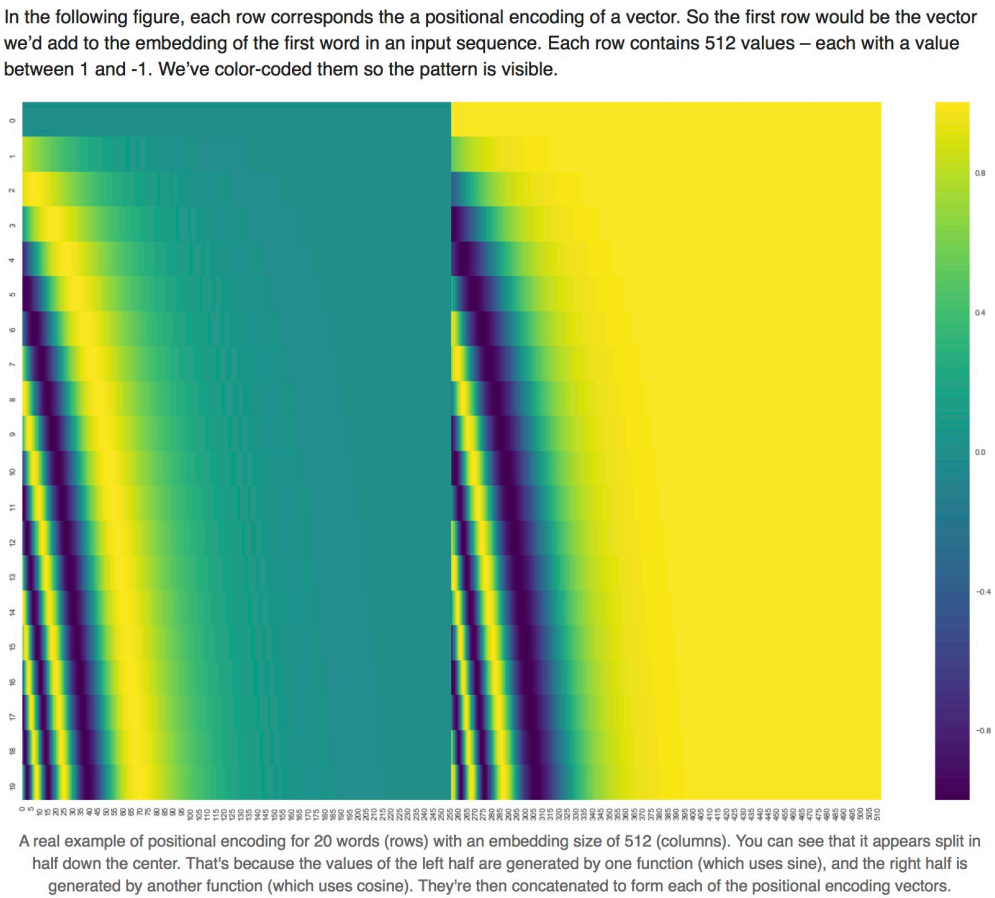


Figure: Source: `http://jalammar.github.io`. This illustration is unfortunately incorrect as it uses `sin` for the first half of the embedding and `cos` for the second half, instead of `sin` for even indices and `cos` for odd indices.

Torch Tensor for Positional Encodings

```
for pos in range(max_seq_len):
    for i in range(0, d_model, 2):
        pe[pos, i] = math.sin(pos / (10000 ** ((2 * i)/d_model)))
        pe[pos, i + 1] = math.cos(pos / (10000 ** ((2 * (i + 1))/d_model)))
print(pe)
```

```
tensor([[ 0.0000,  1.0000,  0.0000,  1.0000,  0.0000,  1.0000,  0.0000,  1.0000,
          0.0000,  1.0000],
        [ 0.8415,  0.9875,  0.0251,  1.0000,  0.0006,  1.0000,  0.0000,  1.0000,
          0.0000,  1.0000],
        [ 0.9093,  0.9502,  0.0502,  1.0000,  0.0013,  1.0000,  0.0000,  1.0000,
          0.0000,  1.0000],
        [ 0.1411,  0.8891,  0.0753,  0.9999,  0.0019,  1.0000,  0.0000,  1.0000,
          0.0000,  1.0000],
        [-0.7568,  0.8057,  0.1003,  0.9999,  0.0025,  1.0000,  0.0001,  1.0000,
          0.0000,  1.0000],
        [-0.9589,  0.7021,  0.1253,  0.9998,  0.0032,  1.0000,  0.0001,  1.0000,
          0.0000,  1.0000],
        [-0.2794,  0.5809,  0.1501,  0.9997,  0.0038,  1.0000,  0.0001,  1.0000,
          0.0000,  1.0000],
        [ 0.6570,  0.4452,  0.1749,  0.9996,  0.0044,  1.0000,  0.0001,  1.0000,
          0.0000,  1.0000],
        [ 0.9894,  0.2983,  0.1996,  0.9995,  0.0050,  1.0000,  0.0001,  1.0000,
          0.0000,  1.0000],
        [ 0.4121,  0.1439,  0.2241,  0.9994,  0.0057,  1.0000,  0.0001,  1.0000,
          0.0000,  1.0000]])
```

Figure: Tensor of 10 x 10. (max_seq_len and d_model both set to 10).

Creating New Encodings

```
print("\n-- Positional encoding for w in position 3:\n\n", pe3)
print("\n-- Embedding vector for w in position 3:\n\n", word_embedding)
```

```
-- Positional encoding for w in position 3:

 tensor([1.4112e-01, 8.8908e-01, 7.5285e-02, 9.9993e-01, 1.8929e-03, 1.0000e+00,
        4.7547e-05, 1.0000e+00, 1.1943e-06, 1.0000e+00])

-- Embedding vector for w in position 3:

 [0.15127356 0.11721583 0.8375494  0.2517023  0.52421439 0.63624648
 0.52809084 0.21888487 0.93466182 0.419567  ]
```

```
# We simply add the two vectors
print("\n-- Embedding vector and positional encoding added:\n\n", word_embedding + pe3)
```

```
--Embedding vector and positional encoding added:

 tensor([0.2924, 1.0063, 0.9128, 1.2516, 0.5261, 1.6362, 0.5281, 1.2189, 0.9347,
        1.4196], dtype=torch.float64)
```

Figure: New encoding is simply created by adding up embedding vector and sinusoidal vector.

Encoder Blocks (6 Identical Blocks)

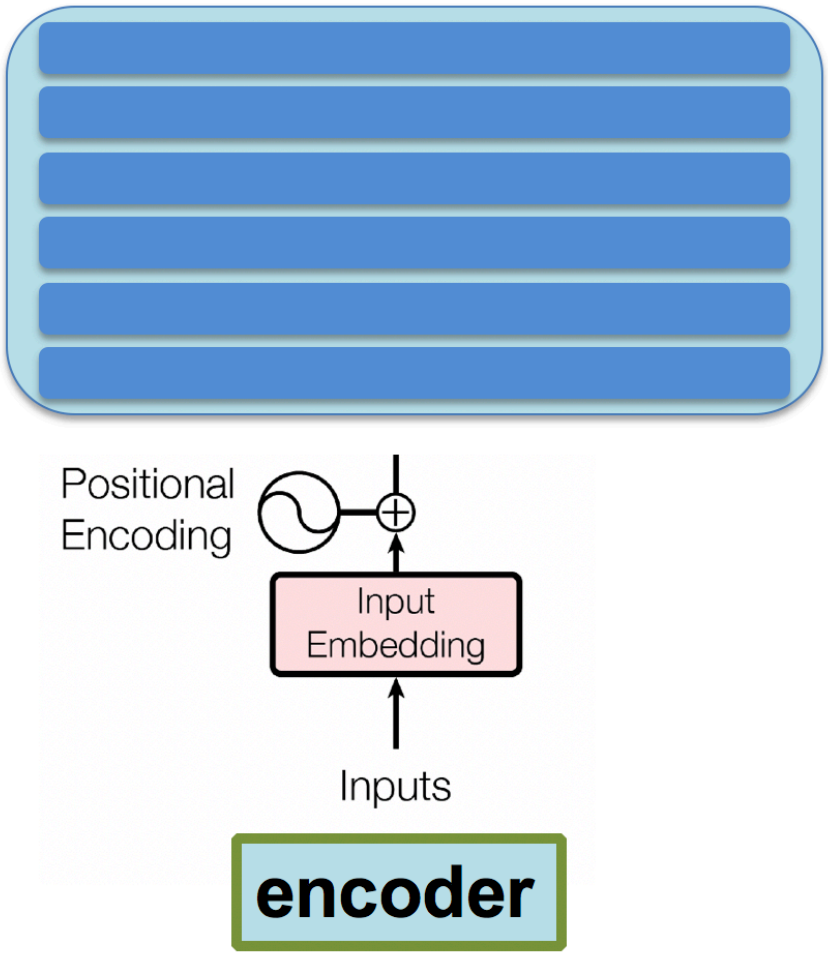


Figure: Encoder's 6 identical blocks.

An Encoder Block

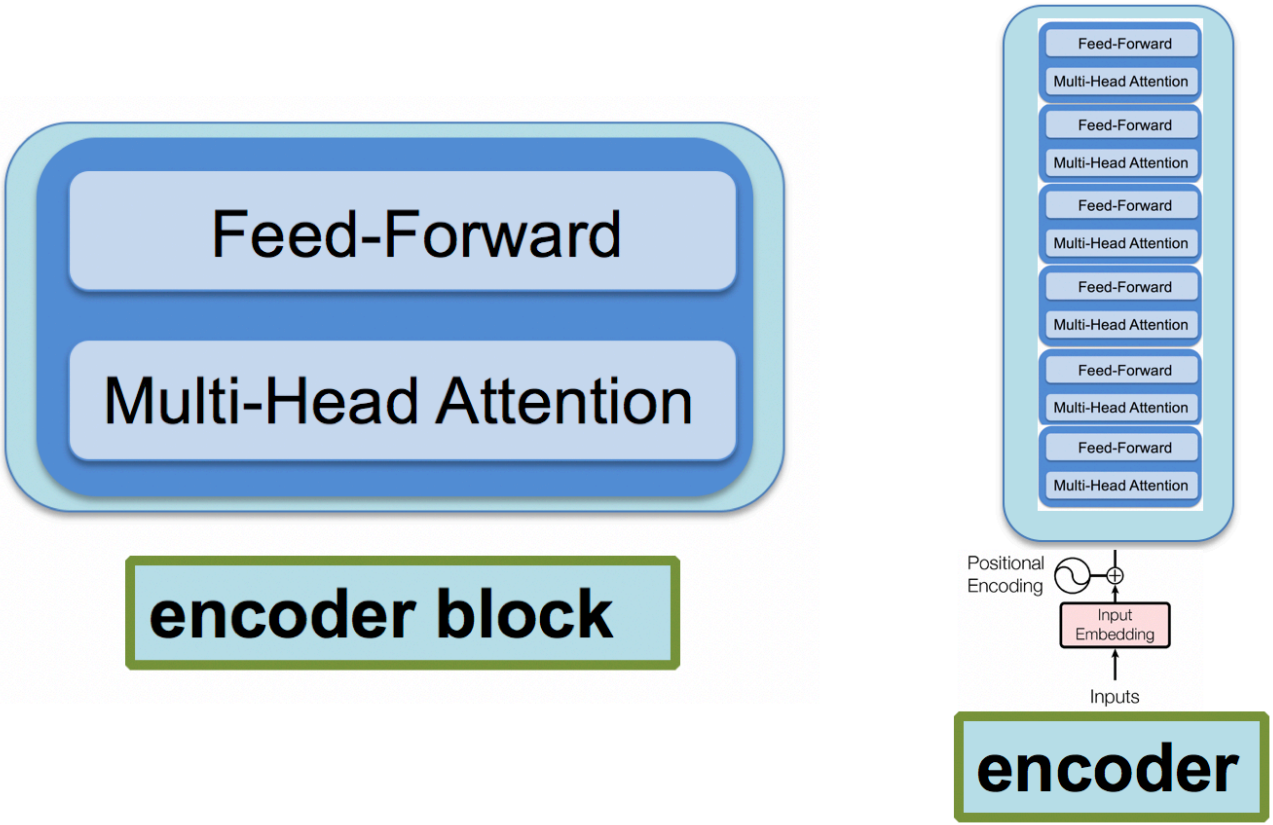


Figure: Each encoder block has two sub-layers, a multi-head attention and a feed-forward.

Self-Attention

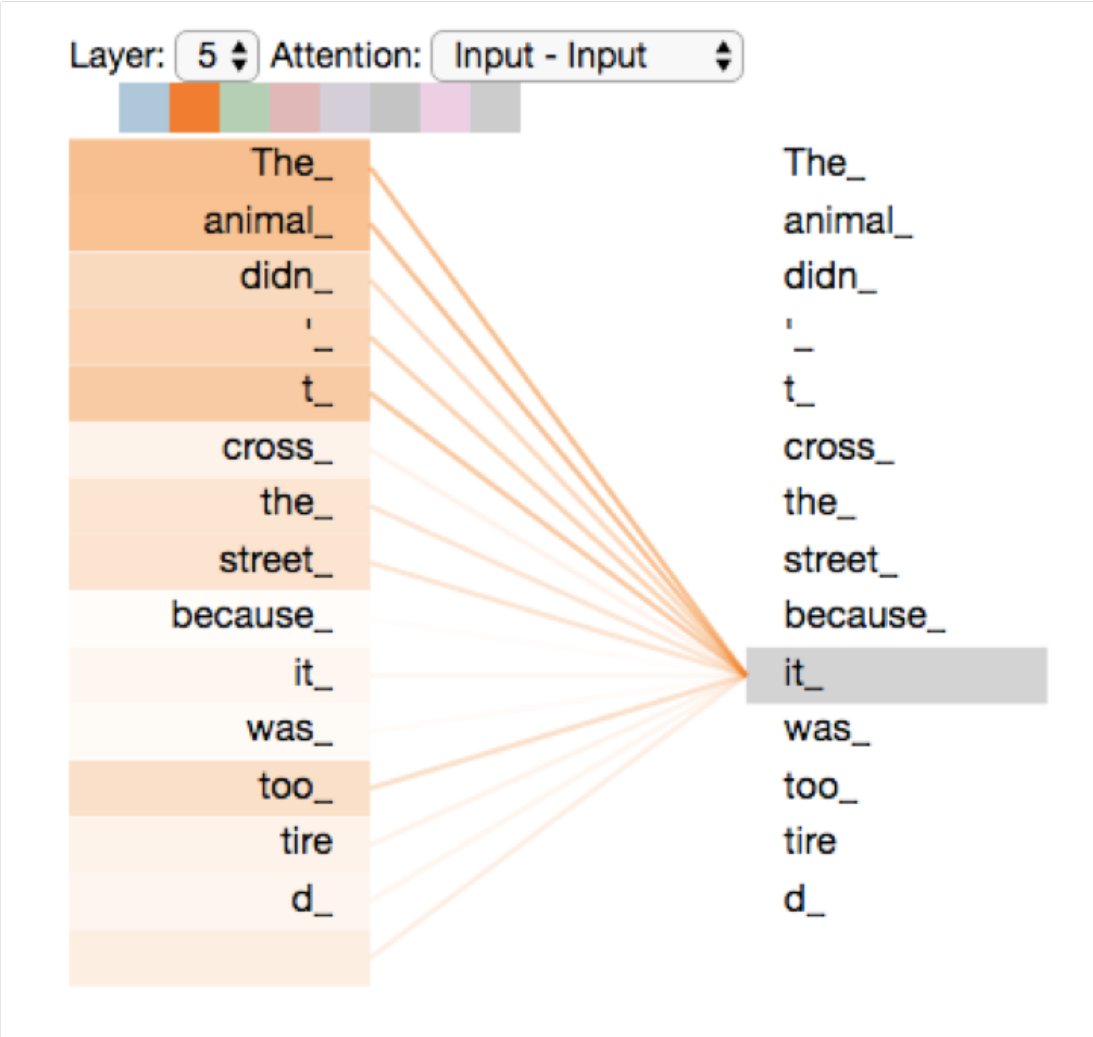
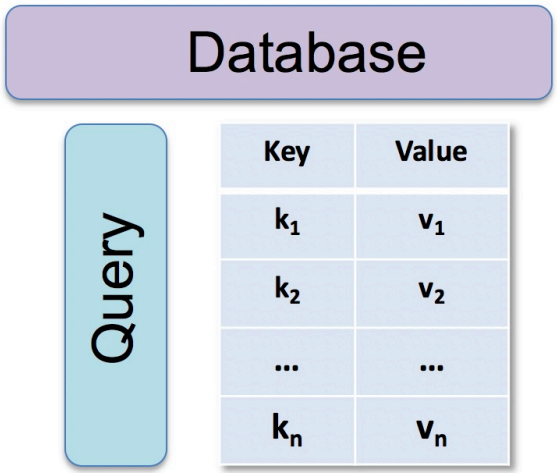


Figure: Self-attention. (Source: `http://jalammar.github.io`).

Query, Key, and Value

Intuition

- Retrieving from a database, we issue a **query (q)** to identify a **key (k)** that has a similarity, based on a value (v), to q.
- To do **backpropagation** on the output, we will not just want one similarity value, but a **similarity distribution** over all keys.



$$Attention(q, \mathbf{k}, \mathbf{v}) = \sum_i sim(q, k_i) v_i$$

MT Example of Query, Key, and Value

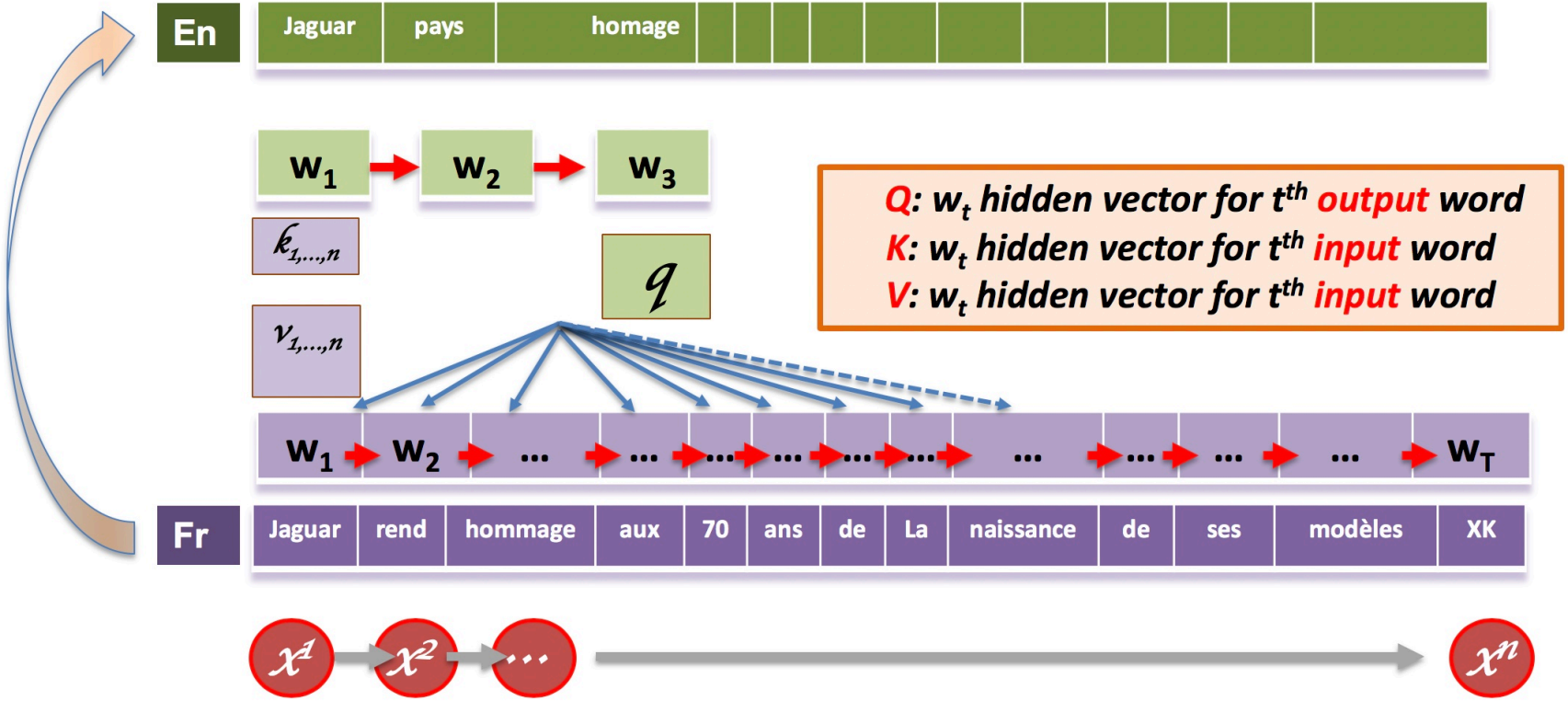


Figure: q, k, and v in MT.

Self-Attention in Transformer with q, k, v

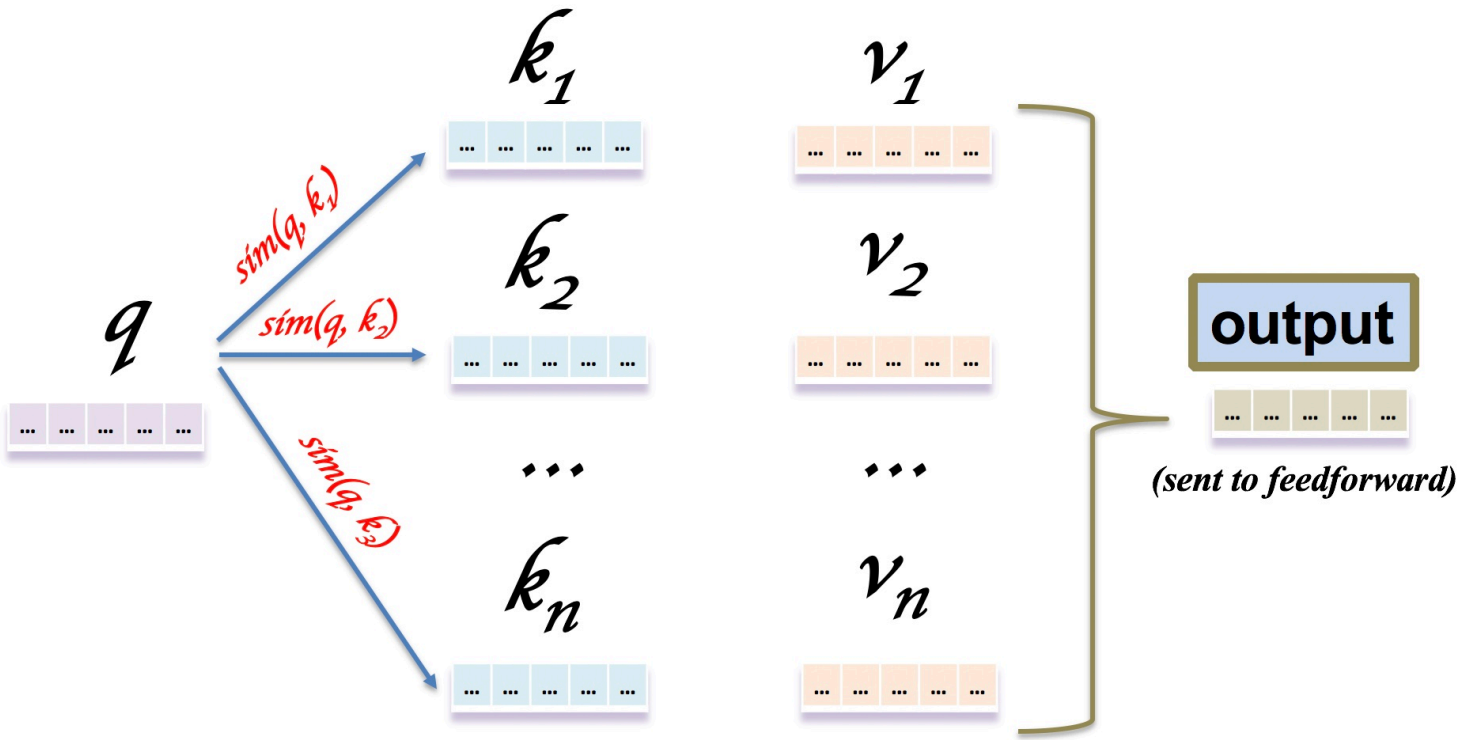


Figure: Mapping a **query** and a set of **key-value** pairs to an output. **Similarity s** will be based on **dot product**. Well, *scaled dot product*...

Z weighted sum of $\sum_i \text{softmax}(\text{sim}(q, k_i))v_i$

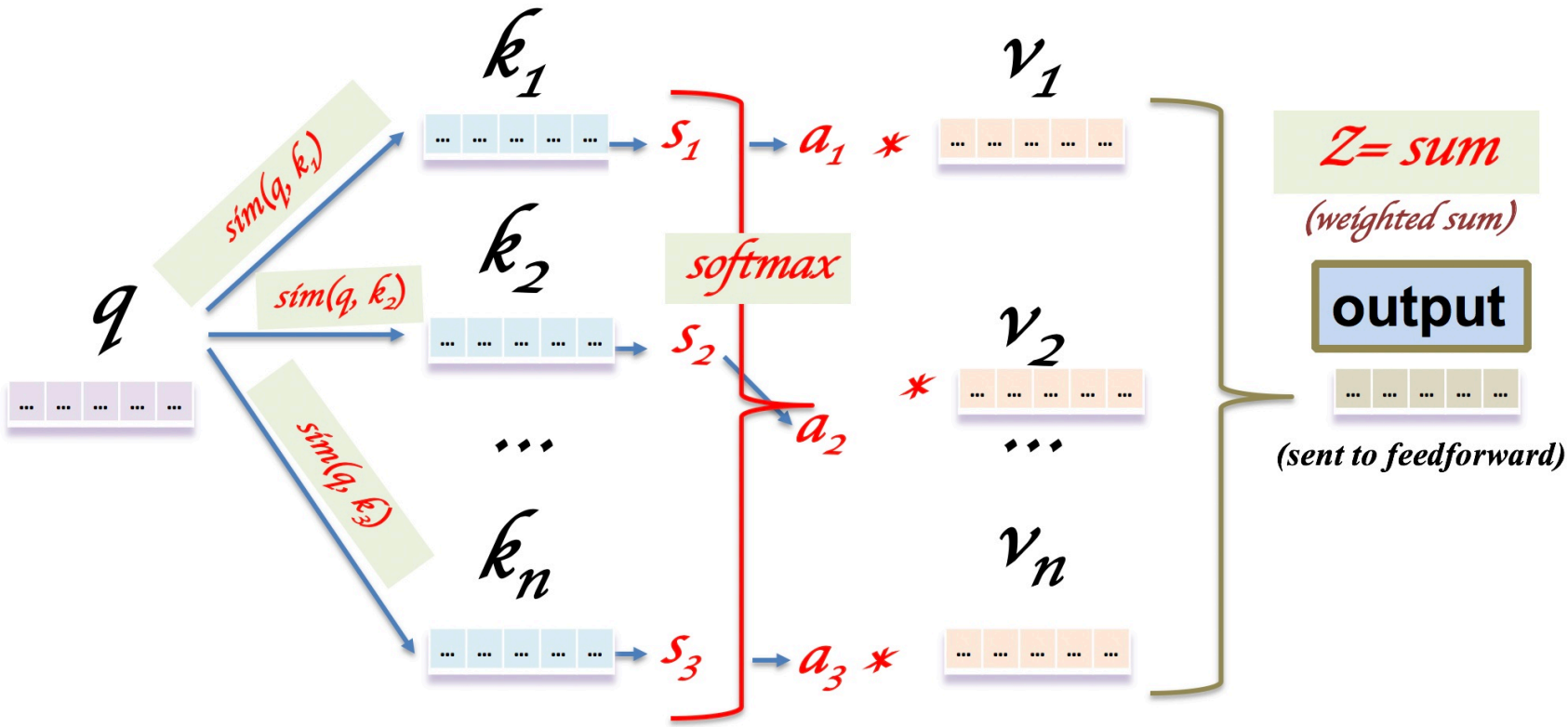


Figure: s=sim(q, k_i). $\alpha = softmax(s)$. $z = \sum_i a_i v_i$

Self-Attention Illustrated: Query "the"

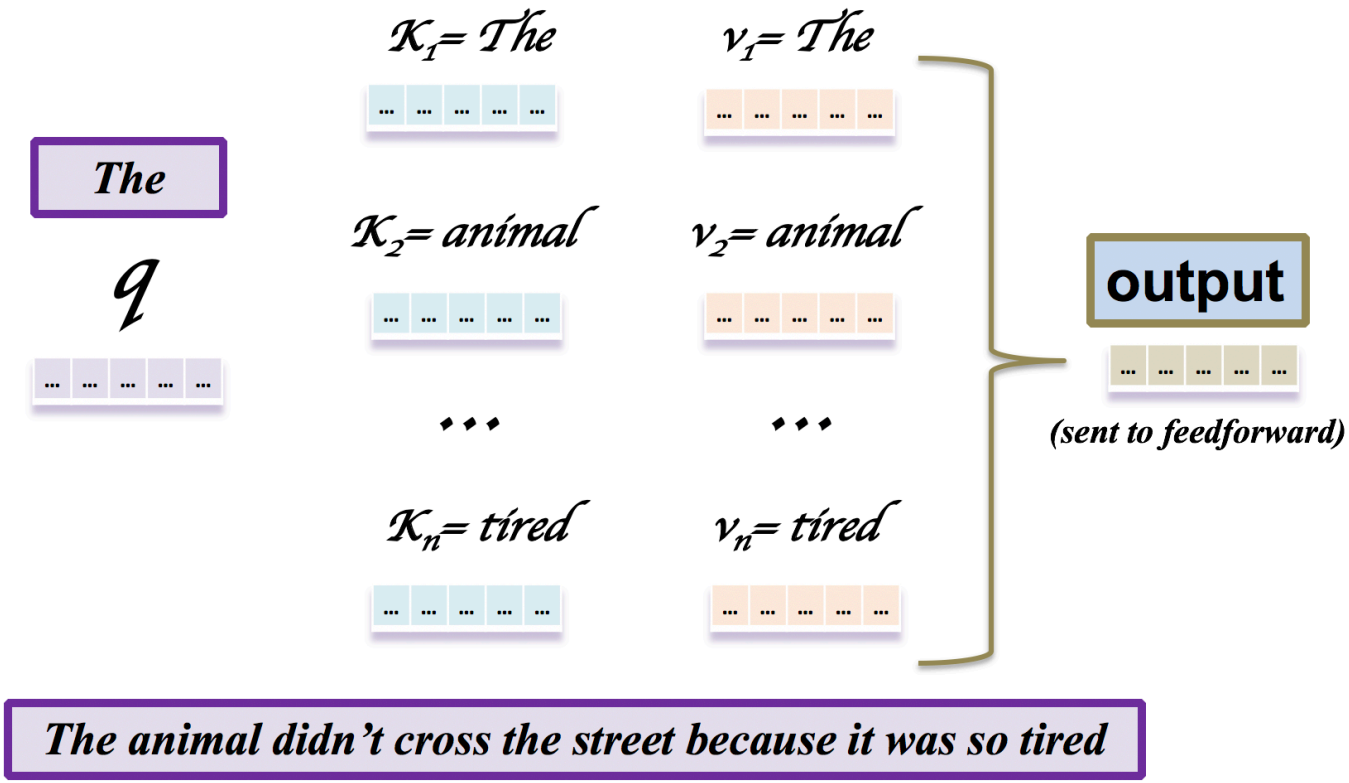


Figure: Mapping the **query** "the" and a set of **key-value** pairs to an output.

Self-Attention Illustrated: Query "animal"

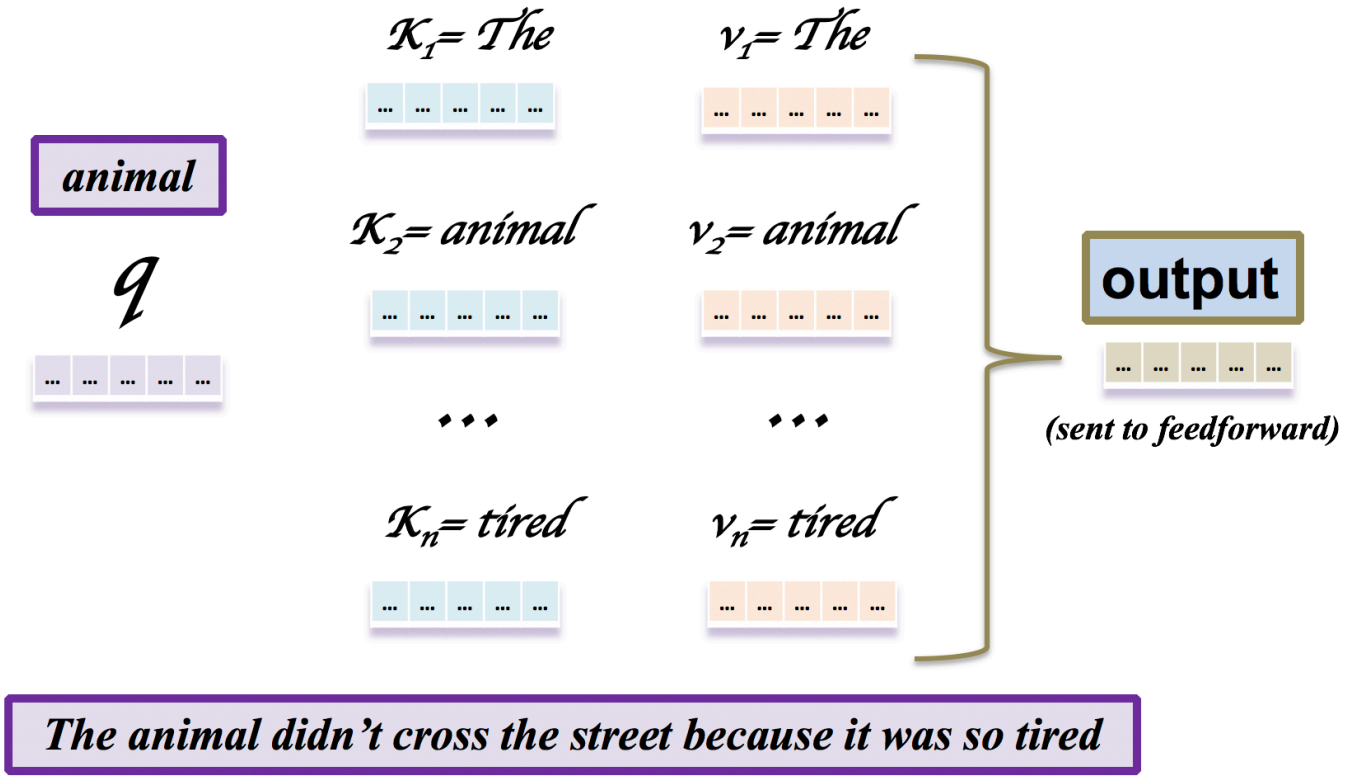


Figure: Mapping the **query** "animal" and a set of **key-value** pairs to an output.

Self-Attention Illustrated: Query "tired"

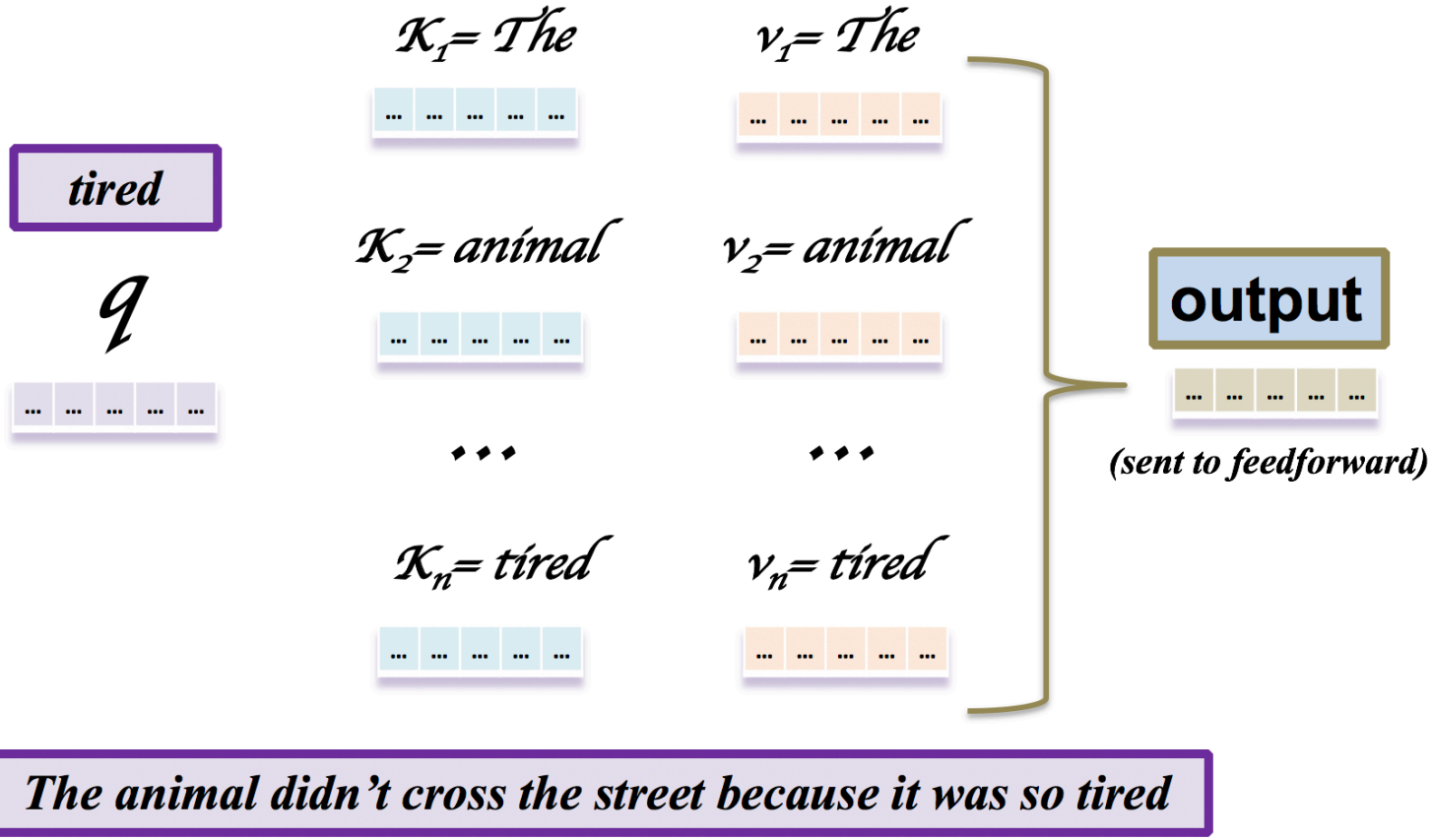


Figure: Mapping the **query** "tired" and a set of **key-value** pairs to an output.

Self-Attention Illustrated: All-Queries Overview'

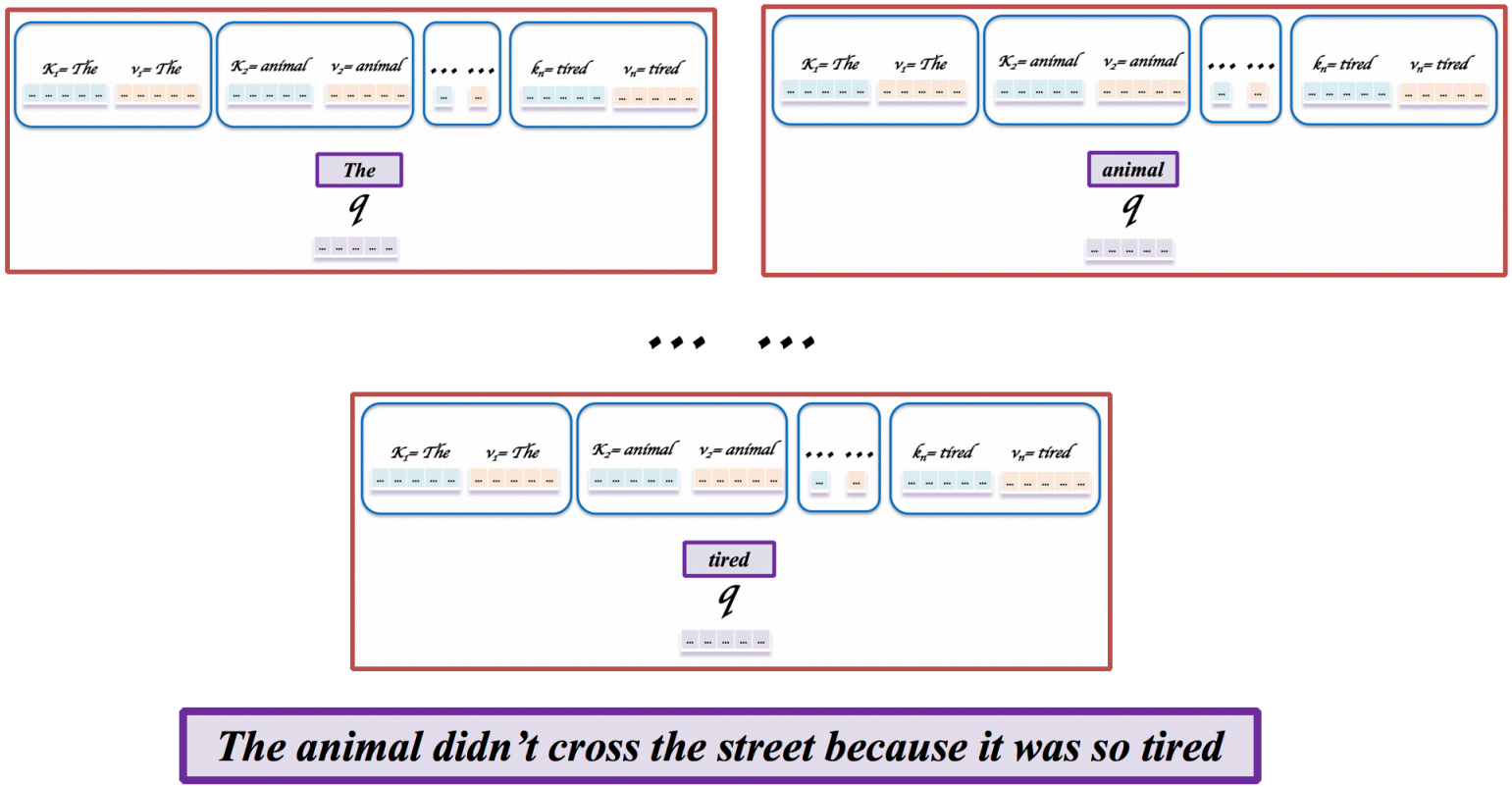


Figure: Mapping all **queries** and a set of **key-value** pairs to an output.

Scaled Dot-Product Attention

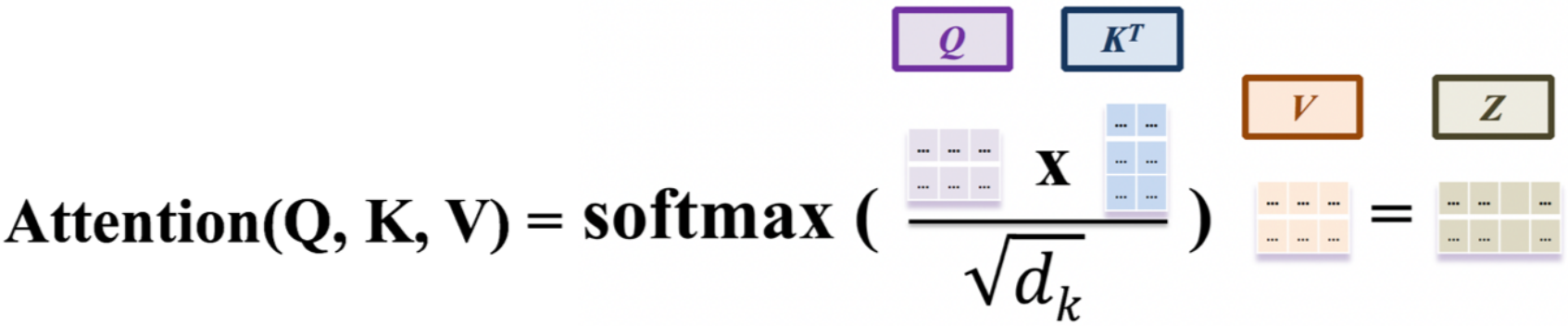


Figure: **Scaled dot-product attention** in the Transformer.

Scaled Dot-Product Attention: Paper Illustration

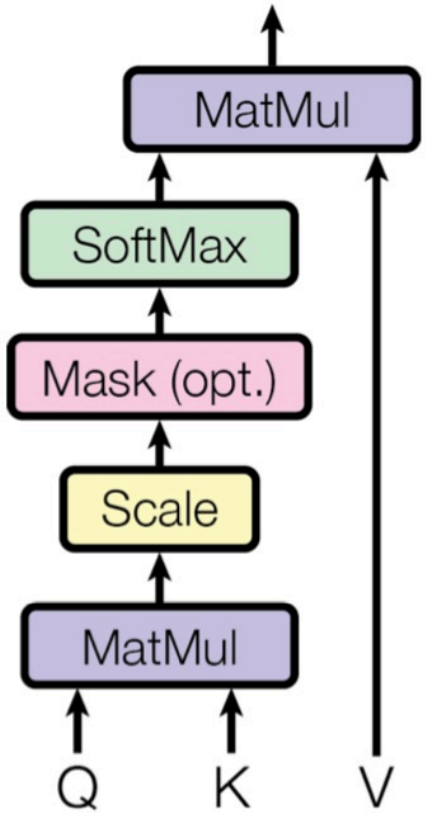


Figure: **Scaled dot-product attention**

Linear Projections of Query, Key, and Value (for Multi-Head Attention)

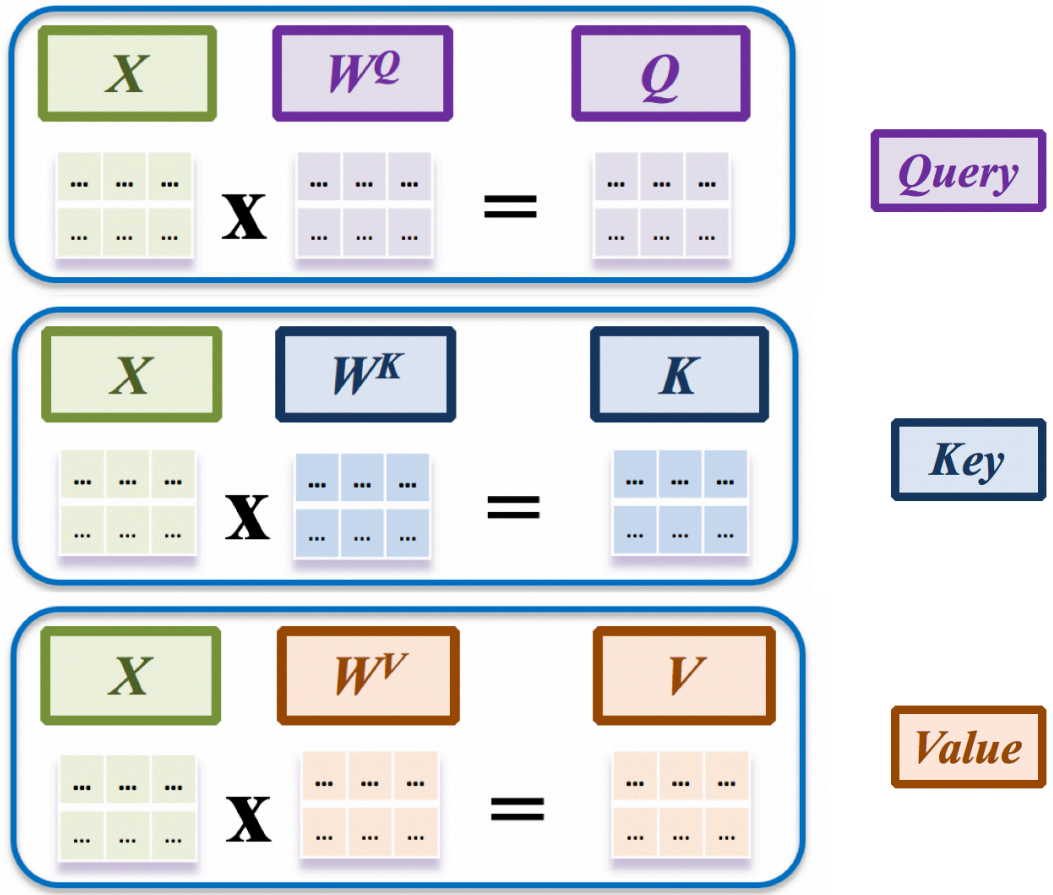


Figure: Linear projections of Q, K, and V. **W** matrixes are learned linear projections.

Multi-Head Attention

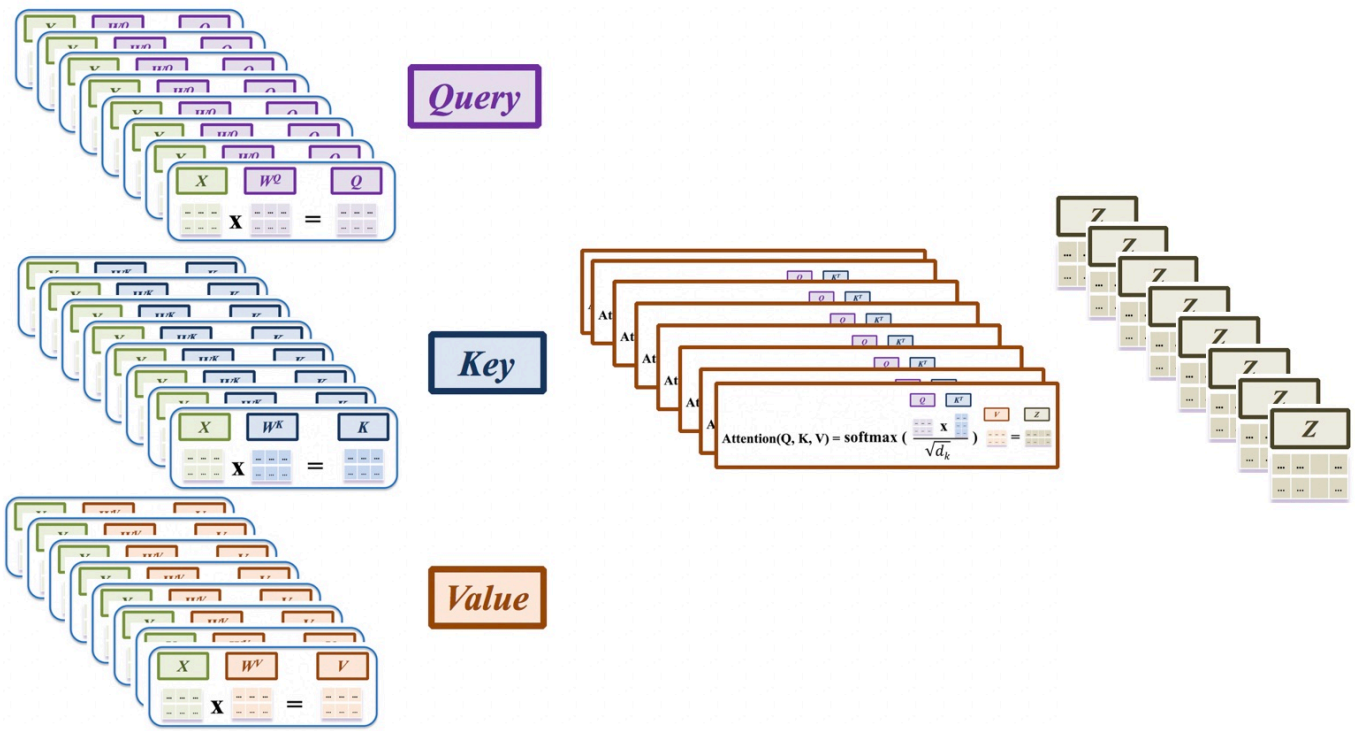


Figure: Multi-Head Attention with 8 attention heads.

Multi-Head Attention in Paper

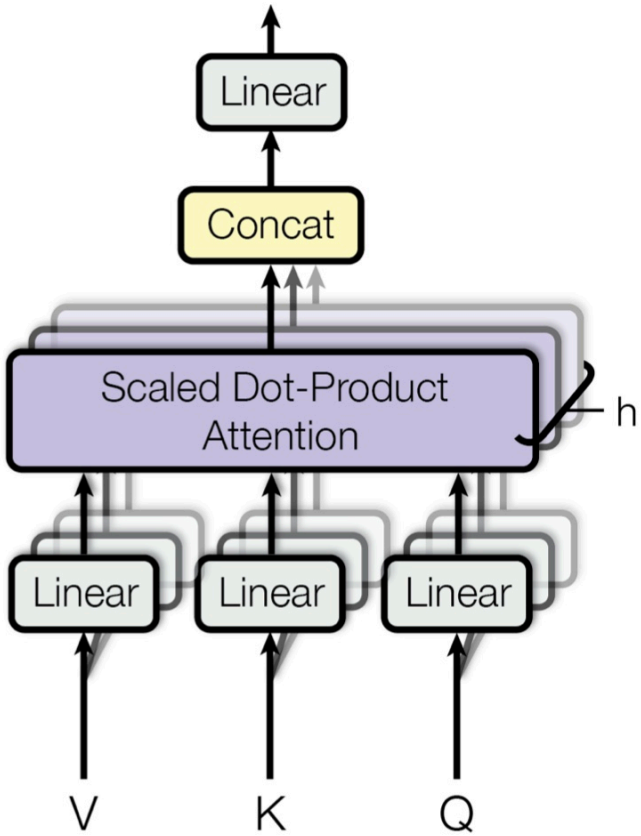
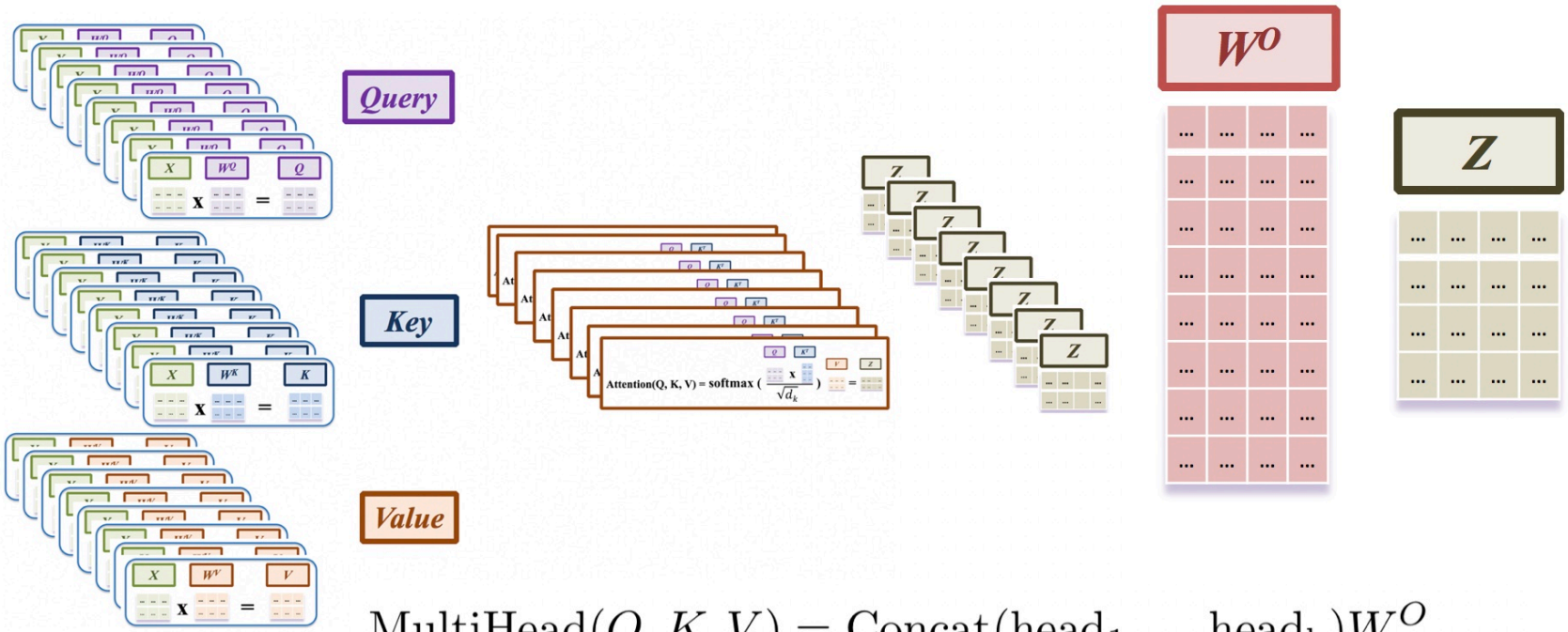


Figure: **Multi-Head Attention**

Multi-Head Attention: Output Concatenation



$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$

$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Figure: Output of attention heads are concatenated and projected to a linear layer.

Detailed Encoder Block