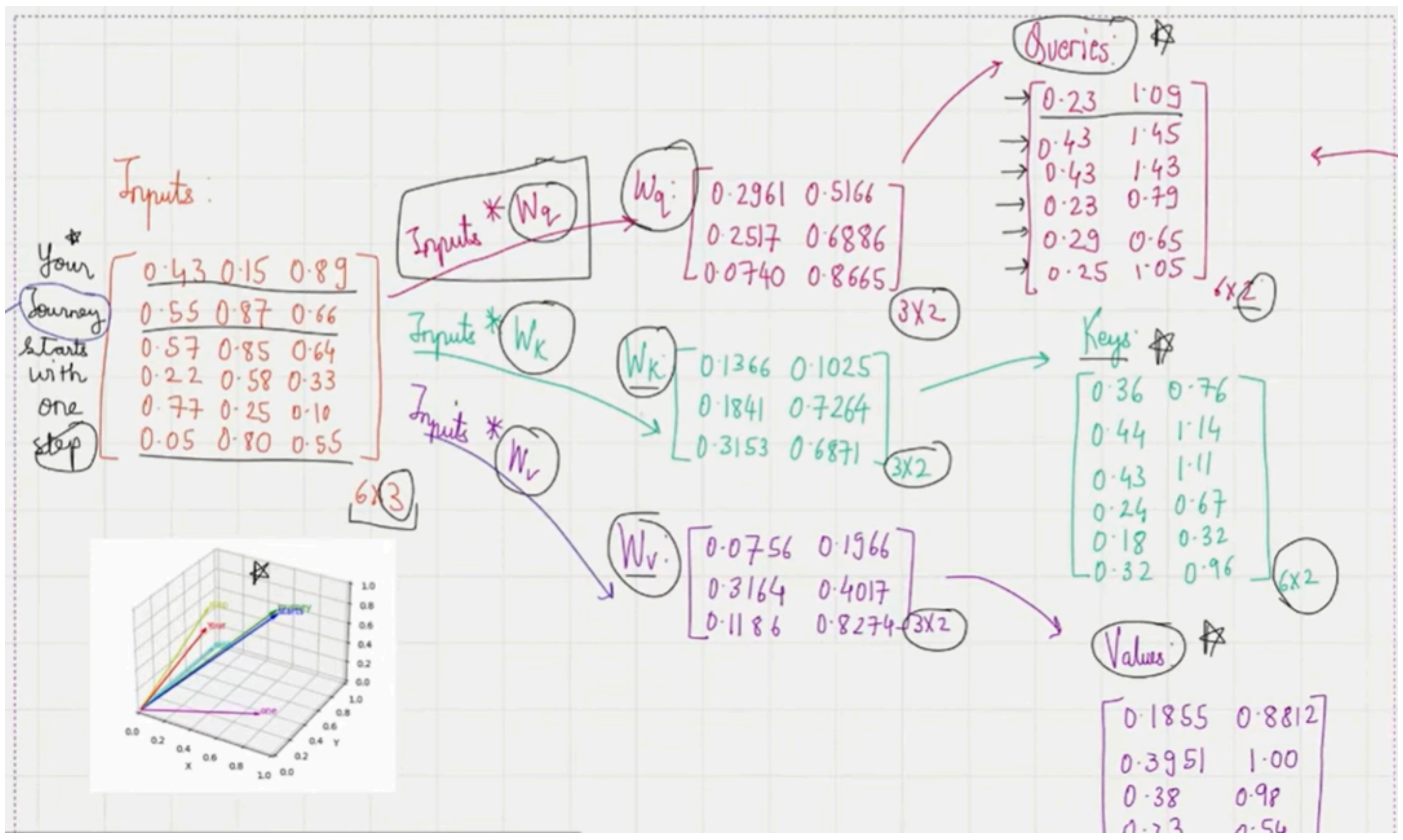
Lecture 15: Self Attention

In this lecture we will be introduced with **Weight Matrices,** that are updated during model training.

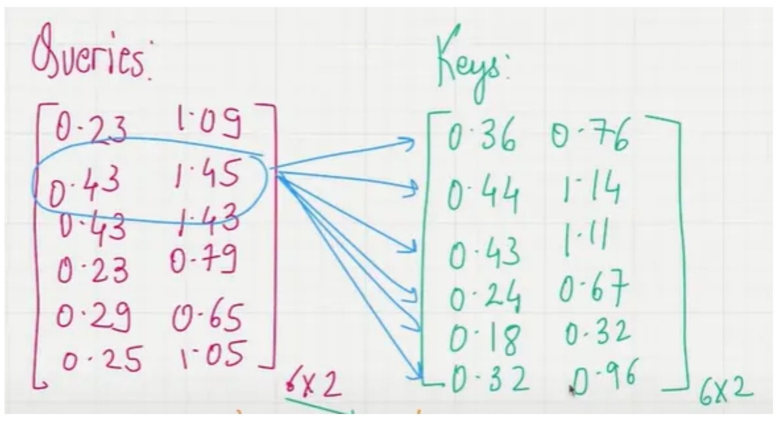
These trainable weight matrices are crucial so that the model can learn to produce “good” context vectors.

There are 3 trainable weight matrices: Wq, Wk, Wv.

The input embeddings are converted into Wq, Wk and Wv. After this conversion, the previous input embeddings are no longer needed.



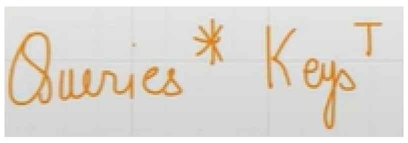
To find the attention score for the second query, we need to take that particular row and find the dot product with all the other rows of the key vector’ transpose. So we’ll have 6 attention scores.

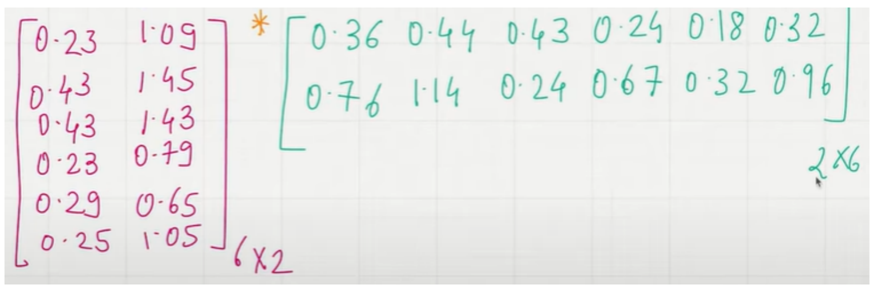


These scores give the information that, when we look at the query for journey, how much importance should be given to other words.

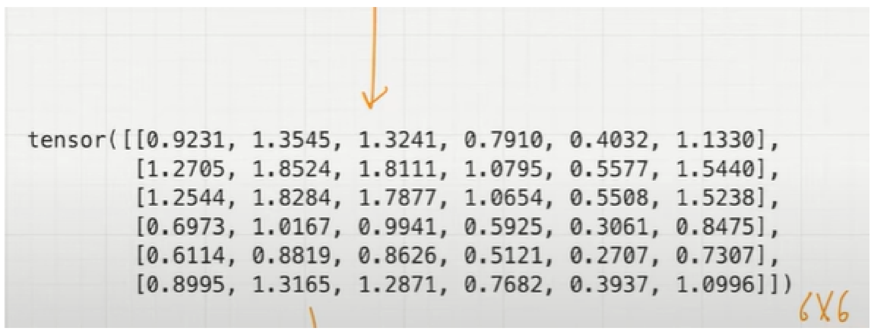
The similar operation is done for all the queries, so we simply do this operation:

This is our **Attention Score matrix.**



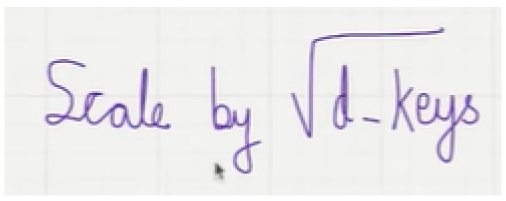


$$\begin{bmatrix} 0.23 & 1.09 \\ 0.43 & 1.45 \\ 0.43 & 1.43 \\ 0.23 & 0.79 \\ 0.29 & 0.65 \\ 0.25 & 1.05 \end{bmatrix}_{6\times2} * \begin{bmatrix} 0.36 & 0.44 & 0.43 & 0.24 & 0.18 & 0.32 \\ 0.76 & 1.14 & 0.24 & 0.67 & 0.32 & 0.96 \end{bmatrix}_{2\times6}$$



```
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]])
```

Now we will calculate **Attention Weights** by normalizing the attention score**.** Attention weights sum upto 1.   We will use **Softmax** to normalize. But before applying softmax, we will do:

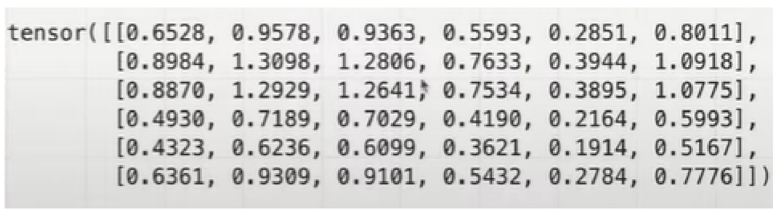


Scale by $\sqrt{d\_keys}$

Here, d_keys = dimension of keys

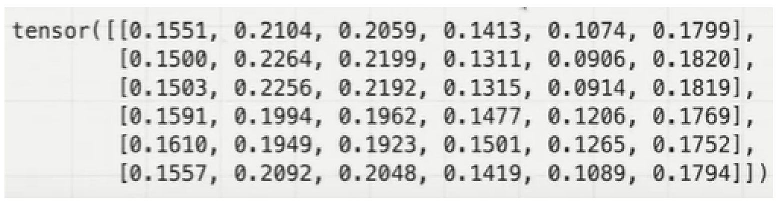
In this case, d_keys = 2

After scaling by **√2** (dividing by **√2)**:



```
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]])
```

Then applying Softmax on it:



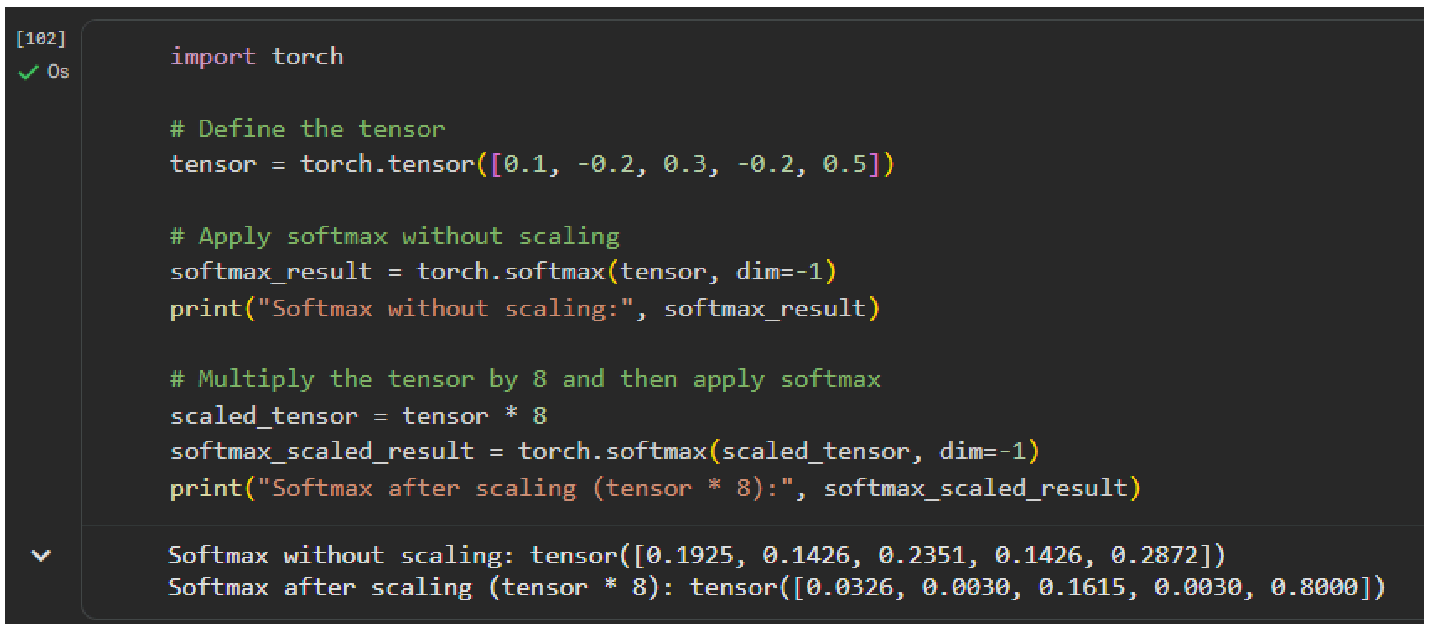
```
tensor([[0.1551, 0.2104, 0.2059, 0.1413, 0.1074, 0.1799],
        [0.1500, 0.2264, 0.2199, 0.1311, 0.0906, 0.1820],
        [0.1503, 0.2256, 0.2192, 0.1315, 0.0914, 0.1819],
        [0.1591, 0.1994, 0.1962, 0.1477, 0.1206, 0.1769],
        [0.1610, 0.1949, 0.1923, 0.1501, 0.1265, 0.1752],
      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

This is the **Attention Weights matrix.** These weights are not optimized right now. They will be optimized after training.

Why divide by SQRT?

Reason 1: For stability in learning

The softmax function is sensitive to the magnitudes of its inputs. When the inputs are large, the differences between the exponential values of each input become much more pronounced. This causes the softmax output to become **"peaky,"** where the **highest value receives almost all the probability mass**, and the rest receive very little.



```
[102]
 0s
    import torch

    # Define the tensor
    tensor = torch.tensor([0.1, -0.2, 0.3, -0.2, 0.5])

    # Apply softmax without scaling
    softmax_result = torch.softmax(tensor, dim=-1)
    print("Softmax without scaling:", softmax_result)

    # Multiply the tensor by 8 and then apply softmax
    scaled_tensor = tensor * 8
    softmax_scaled_result = torch.softmax(scaled_tensor, dim=-1)
    print("Softmax after scaling (tensor * 8):", softmax_scaled_result)
```

```
Softmax without scaling: tensor([0.1925, 0.1426, 0.2351, 0.1426, 0.2872])
Softmax after scaling (tensor * 8): tensor([0.0326, 0.0030, 0.1615, 0.0030, 0.8000])
```

In attention mechanisms, particularly in transformers, if the dot products between query and key vectors become too large (like multiplying by 8 in this example), the attention scores can become very large. This results in a very sharp softmax distribution, making the model **overly confident in one particular "key."** Such sharp distributions can make learning unstable.

But why SQRT?

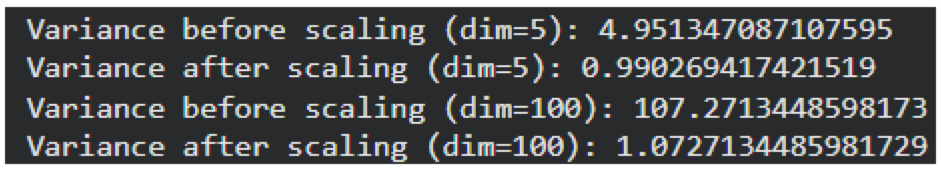
Reason 2: To make the variance of the dot product stable

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

The increase in variance grows with the dimension.

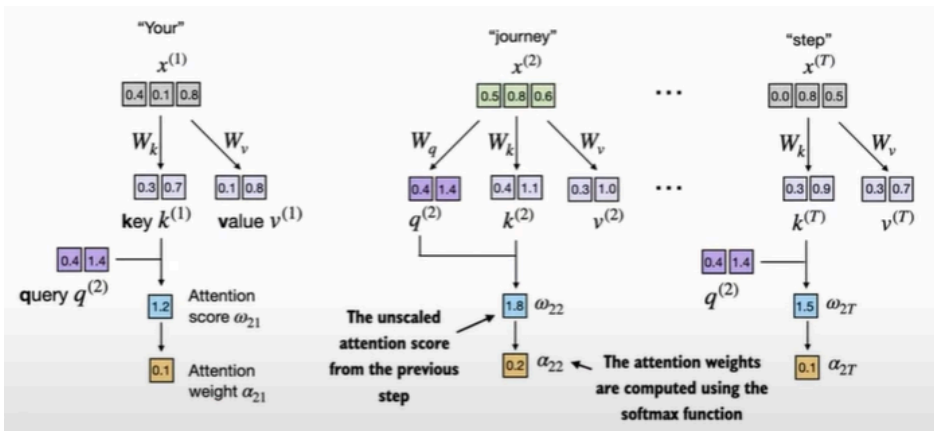
Dividing by sqrt (dimension) keeps the variance close to 1.

Why variance close to 1 is important? Because this ensures stability in training.

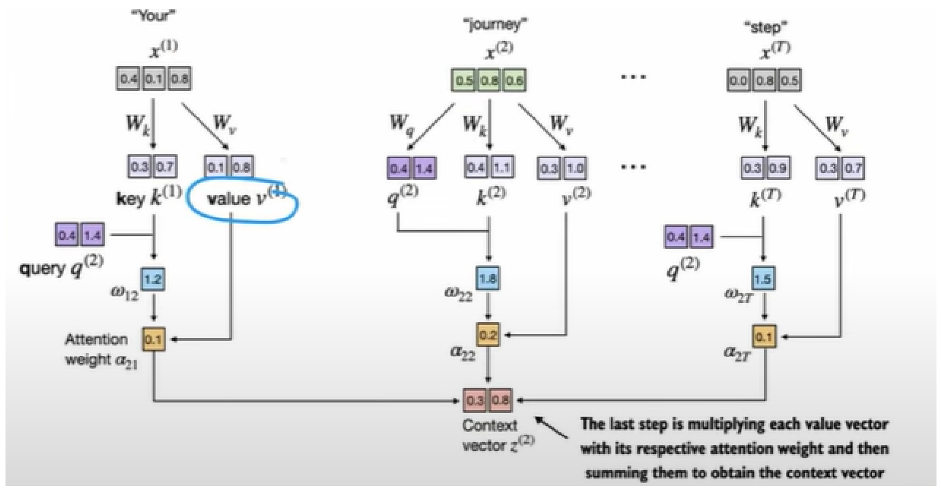


```
Variance before scaling (dim=5): 4.951347087107595
Variance after scaling (dim=5): 0.990269417421519
Variance before scaling (dim=100): 107.2713448598173
Variance after scaling (dim=100): 1.0727134485981729
```

What we have done until now:

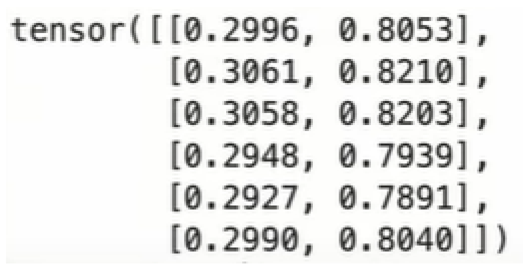


Now, to get the **Context Vector matrix,** we multiply the **attention weight matrix with the Value vector.**



The last step is multiplying each value vector with its respective attention weight and then summing them to obtain the context vector

The Context vector matrix looks like this:



```
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]])
```

So 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 in.

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

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