Lecture 6: Attention and Transformer Basics

CSC 352 / CSC 552 · 2026-09-15

Open in Colab → **File → Save a copy in Drive**. You cannot edit the class original.

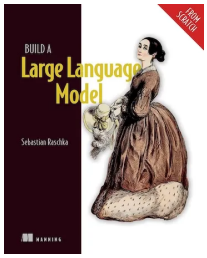
Start-of-class quiz

Type answers in the Canvas attendance quiz (empty boxes only). The questions stay here.

1. In a training window, what is `y` relative to `x`?
2. What does `nn.Embedding` do with a token id?

Supplementary code for the Build a Large Language Model From Scratch book by Sebastian Raschka

Code repository:
https://github.com/rasbt/LLMs-from-scratch

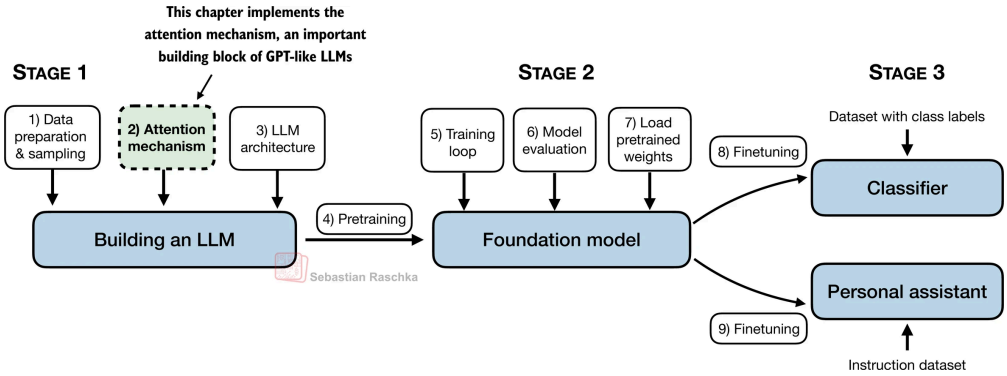


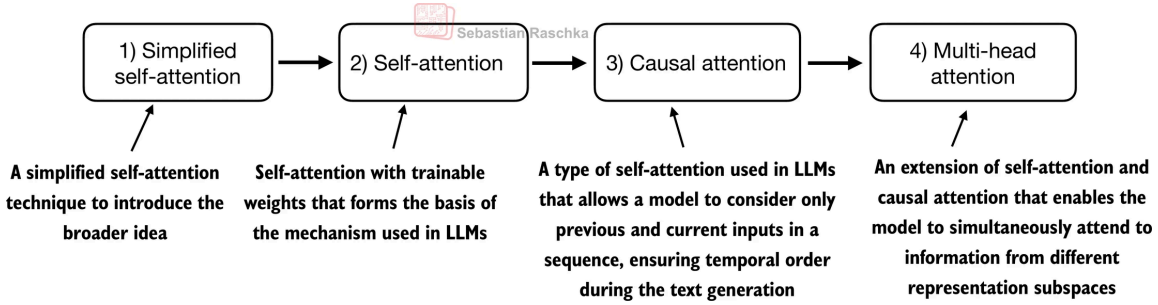
Packages that are being used in this notebook:

In [5]:
```
from importlib.metadata import version

print("torch version:", version("torch"))
```

```
torch version: 2.8.0
```

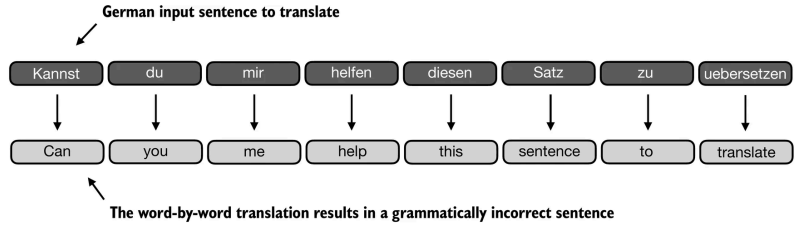
- This chapter covers attention mechanisms, the engine of LLMs:

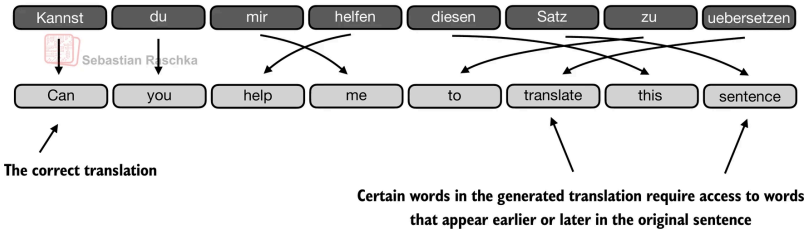




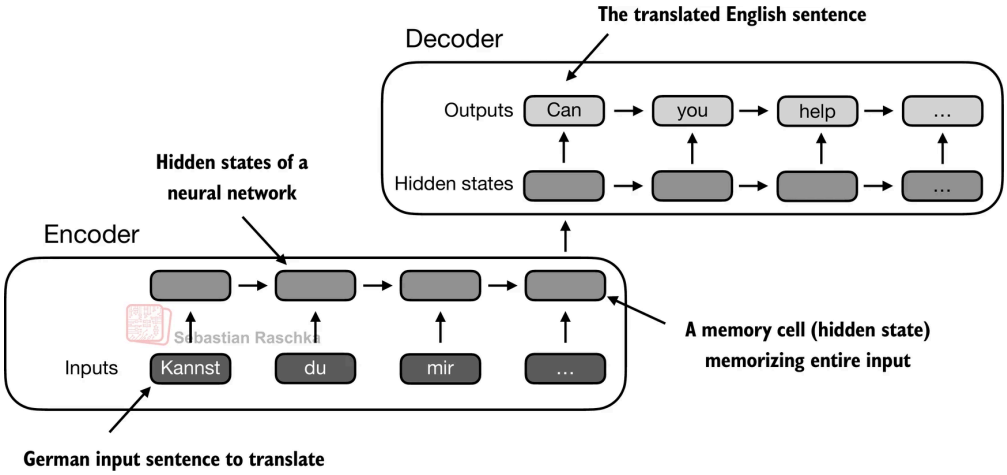
3.1 The problem with modeling long sequences

- No code in this section
- Translating a text word by word isn't feasible due to the differences in grammatical structures between the source and target languages:



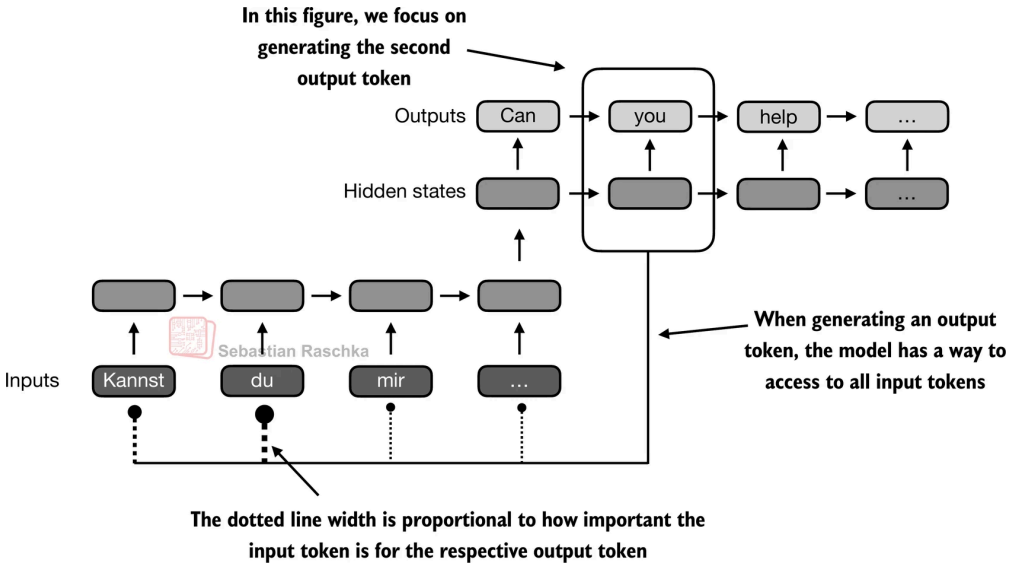


- Prior to the introduction of transformer models, encoder-decoder RNNs were commonly used for machine translation tasks
- In this setup, the encoder processes a sequence of tokens from the source language, using a hidden state—a kind of intermediate layer within the neural network—to generate a condensed representation of the entire input sequence:

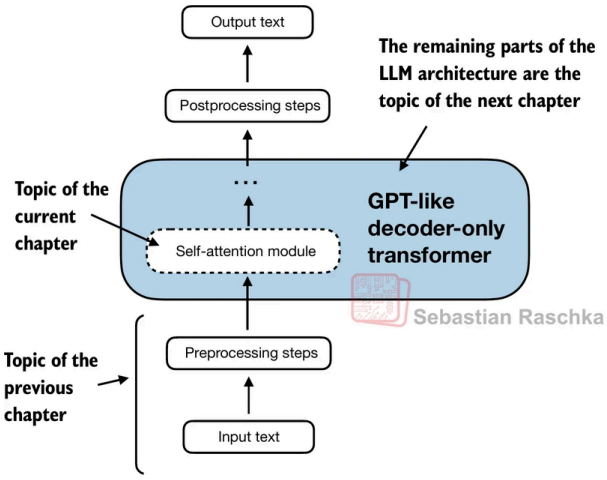


3.2 Capturing data dependencies with attention mechanisms

- No code in this section
- Through an attention mechanism, the text-generating decoder segment of the network is capable of selectively accessing all input tokens, implying that certain input tokens hold more significance than others in the generation of a specific output token:



- Self-attention in transformers is a technique designed to enhance input representations by enabling each position in a sequence to engage with and determine the relevance of every other position within the same sequence

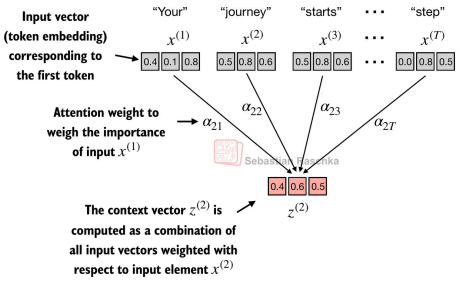


3.3 Attending to different parts of the input with self-attention

3.3.1 A simple self-attention mechanism without trainable weights

- This section explains a very simplified variant of self-attention, which does not contain any trainable weights
- This is purely for illustration purposes and NOT the attention mechanism that is used in transformers
- The next section, section 3.3.2, will extend this simple attention mechanism to implement the real self-attention mechanism
- Suppose we are given an input sequence $x^{(1)}$ to $x^{(T)}$
  - The input is a text (for example, a sentence like "Your journey starts with one step") that has already been converted into token embeddings as described in chapter 2
  - For instance, $x^{(1)}$ is a d-dimensional vector representing the word "Your", and so forth
- **Goal:** compute context vectors $z^{(i)}$ for each input sequence element $x^{(i)}$ in $x^{(1)}$ to $x^{(T)}$ (where z and x have the same dimension)
  - A context vector $z^{(i)}$ is a weighted sum over the inputs $x^{(1)}$ to $x^{(T)}$
  - The context vector is "context"-specific to a certain input
    - Instead of $x^{(i)}$ as a placeholder for an arbitrary input token, let's consider the second input, $x^{(2)}$
    - And to continue with a concrete example, instead of the placeholder $z^{(i)}$, we consider the second output context vector, $z^{(2)}$

    - The second context vector, $z^{(2)}$, is a weighted sum over all inputs $x^{(1)}$ to $x^{(T)}$ weighted with respect to the second input element, $x^{(2)}$
    - The attention weights are the weights that determine how much each of the input elements contributes to the weighted sum when computing $z^{(2)}$
    - In short, think of $z^{(2)}$ as a modified version of $x^{(2)}$ that also incorporates information about all other input elements that are relevant to a given task at hand



7

- (Please note that the numbers in this figure are truncated to one digit after the decimal point to reduce visual clutter; similarly, other figures may also contain truncated values)

- By convention, the unnormalized attention weights are referred to as **"attention scores"** whereas the normalized attention scores, which sum to 1, are referred to as **"attention weights"**

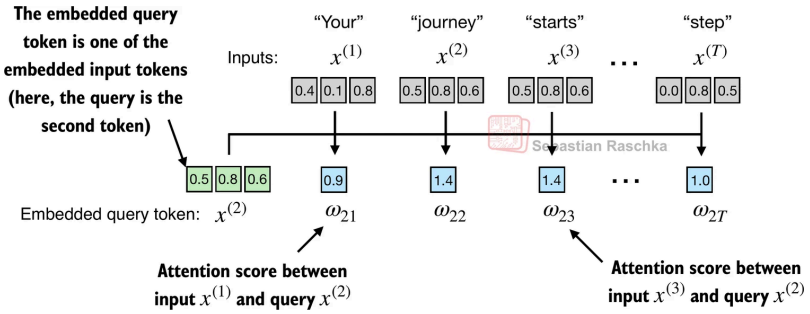
- The code below walks through the figure above step by step

- **Step 1:** compute unnormalized attention scores ω
- Suppose we use the second input token as the query, that is, $q^{(2)} = x^{(2)}$, we compute the unnormalized attention scores via dot products:
    - $\omega_{21} = x^{(1)} q^{(2)\top}$
    - $\omega_{22} = x^{(2)} q^{(2)\top}$
    - $\omega_{23} = x^{(3)} q^{(2)\top}$
    - ...
    - $\omega_{2T} = x^{(T)} q^{(2)\top}$
- Above, ω is the Greek letter "omega" used to symbolize the unnormalized attention scores
    - The subscript "21" in ω_{21} means that input sequence element 2 was used as a query against input sequence element 1

- Suppose we have the following input sentence that is already embedded in 3-dimensional vectors as described in chapter 3 (we use a very small embedding dimension here for illustration purposes, so that it fits onto the page without line breaks):

In [26]:
```
import torch

inputs = torch.tensor(
  [[0.43, 0.15, 0.89], # Your     (x^1)
   [0.55, 0.87, 0.66], # journey  (x^2)
   [0.57, 0.85, 0.64], # starts   (x^3)
   [0.22, 0.58, 0.33], # with     (x^4)
   [0.77, 0.25, 0.10], # one      (x^5)
   [0.05, 0.80, 0.55]] # step     (x^6)
)
```

- (In this book, we follow the common machine learning and deep learning convention where training examples are represented as rows and feature values as columns; in the case of the tensor shown above, each row represents a word, and each column represents an embedding dimension)
- The primary objective of this section is to demonstrate how the context vector $z^{(2)}$ is calculated using the second input sequence, $x^{(2)}$, as a query
- The figure depicts the initial step in this process, which involves calculating the attention scores ω between $x^{(2)}$ and all other input elements through a dot product operation



- We use input sequence element 2, $x^{(2)}$, as an example to compute context vector $z^{(2)}$; later in this section, we will generalize this to compute all context vectors.
- The first step is to compute the unnormalized attention scores by computing the dot product between the query $x^{(2)}$ and all other input tokens:

In [30]:
```
query = inputs[1]  # 2nd input token is the query

attn_scores_2 = torch.empty(inputs.shape[0])
for i, x_i in enumerate(inputs):
```

```
    attn_scores_2[i] = torch.dot(x_i, query) # dot product (transpose not ne

print(attn_scores_2)
```

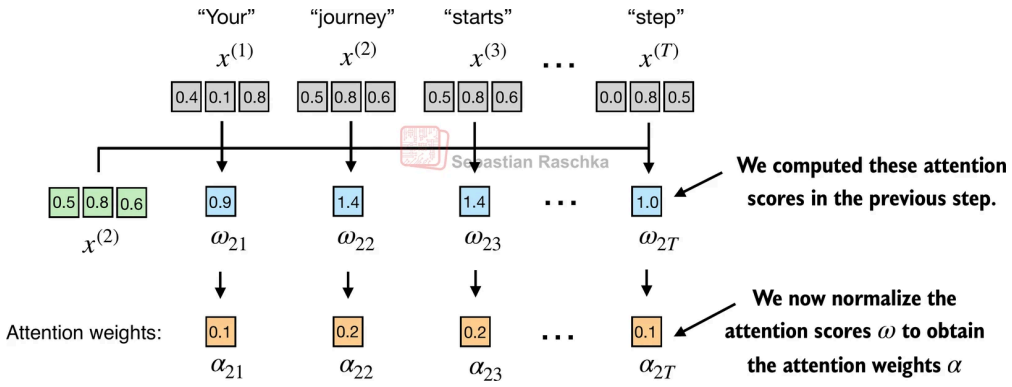
```
tensor([0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865])
```

- Side note: a dot product is essentially a shorthand for multiplying two vectors elements-wise and summing the resulting products:

In [32]:
```
res = 0.

for idx, element in enumerate(inputs[0]):
    res += inputs[0][idx] * query[idx]

print(res)
print(torch.dot(inputs[0], query))
```

```
tensor(0.9544)
tensor(0.9544)
```

- **Step 2:** normalize the unnormalized attention scores ("omegas", ω) so that they sum up to 1
- Here is a simple way to normalize the unnormalized attention scores to sum up to 1 (a convention, useful for interpretation, and important for training stability):



In [35]:
```
attn_weights_2_tmp = attn_scores_2 / attn_scores_2.sum()

print("Attention weights:", attn_weights_2_tmp)
print("Sum:", attn_weights_2_tmp.sum())
```

```
Attention weights: tensor([0.1455, 0.2278, 0.2249, 0.1285, 0.1077, 0.1656])
Sum: tensor(1.0000)
```

- However, in practice, using the softmax function for normalization, which is better at handling extreme values and has more desirable gradient properties during training, is common and recommended.
- Here's a naive implementation of a softmax function for scaling, which also normalizes the vector elements such that they sum up to 1:

In [37]:
```
def softmax_naive(x):
    return torch.exp(x) / torch.exp(x).sum(dim=0)

attn_weights_2_naive = softmax_naive(attn_scores_2)

print("Attention weights:", attn_weights_2_naive)
print("Sum:", attn_weights_2_naive.sum())
```

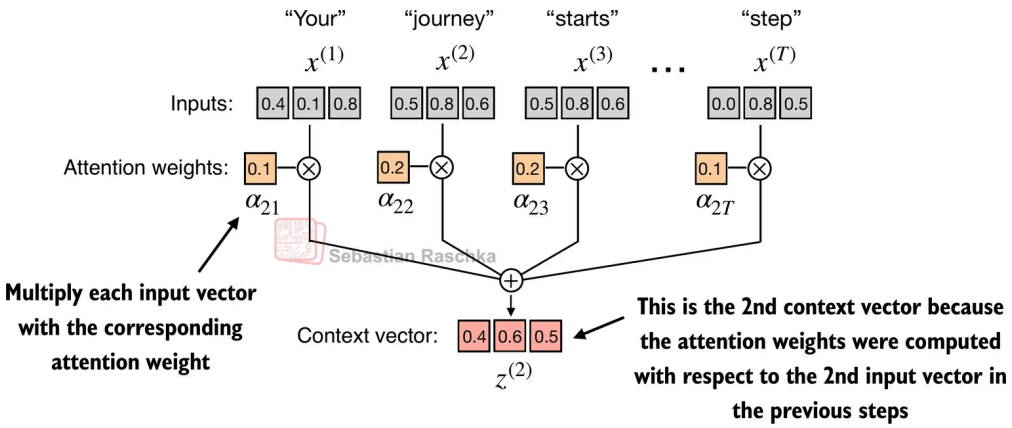
```
Attention weights: tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
Sum: tensor(1.)
```

- The naive implementation above can suffer from numerical instability issues for large or small input values due to overflow and underflow issues
- Hence, in practice, it's recommended to use the PyTorch implementation of softmax instead, which has been highly optimized for performance:

In [39]:
```
attn_weights_2 = torch.softmax(attn_scores_2, dim=0)

print("Attention weights:", attn_weights_2)
print("Sum:", attn_weights_2.sum())
```

```
Attention weights: tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
Sum: tensor(1.)
```

- **Step 3**: compute the context vector $z^{(2)}$ by multiplying the embedded input tokens, $x^{(i)}$ with the attention weights and sum the resulting vectors:



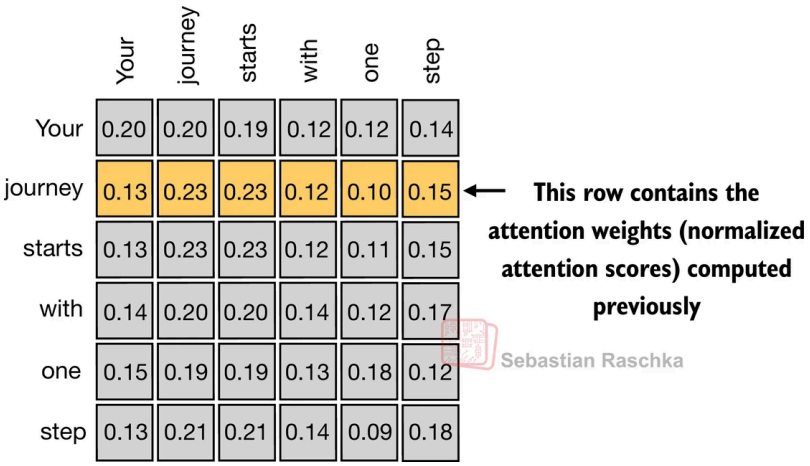
In [42]:
```
query = inputs[1] # 2nd input token is the query

context_vec_2 = torch.zeros(query.shape)
for i,x_i in enumerate(inputs):
    context_vec_2 += attn_weights_2[i]*x_i

print(context_vec_2)
```

```
tensor([0.4419, 0.6515, 0.5683])
```

3.3.2 Computing attention weights for all input tokens

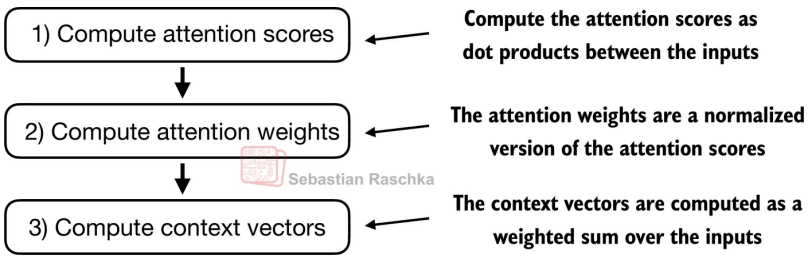
Generalize to all input sequence tokens:

- Above, we computed the attention weights and context vector for input 2 (as illustrated in the highlighted row in the figure below)
- Next, we are generalizing this computation to compute all attention weights and context vectors



- (Please note that the numbers in this figure are truncated to two digits after the decimal point to reduce visual clutter; the values in each row should add up to 1.0 or 100%; similarly, digits in other figures are truncated)

- In self-attention, the process starts with the calculation of attention scores, which are subsequently normalized to derive attention weights that total 1
- These attention weights are then utilized to generate the context vectors through a weighted summation of the inputs



- Apply previous **step 1** to all pairwise elements to compute the unnormalized attention score matrix:

In [49]:
```
attn_scores = torch.empty(6, 6)

for i, x_i in enumerate(inputs):
    for j, x_j in enumerate(inputs):
        attn_scores[i, j] = torch.dot(x_i, x_j)
```

```
print(attn_scores)
```

```
tensor([[0.9995, 0.9544, 0.9422, 0.4753, 0.4576, 0.6310],
        [0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865],
        [0.9422, 1.4754, 1.4570, 0.8296, 0.7154, 1.0605],
        [0.4753, 0.8434, 0.8296, 0.4937, 0.3474, 0.6565],
        [0.4576, 0.7070, 0.7154, 0.3474, 0.6654, 0.2935],
        [0.6310, 1.0865, 1.0605, 0.6565, 0.2935, 0.9450]])
```

- We can achieve the same as above more efficiently via matrix multiplication:

In [51]:
```
attn_scores = inputs @ inputs.T
print(attn_scores)
```

```
tensor([[0.9995, 0.9544, 0.9422, 0.4753, 0.4576, 0.6310],
        [0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865],
        [0.9422, 1.4754, 1.4570, 0.8296, 0.7154, 1.0605],
        [0.4753, 0.8434, 0.8296, 0.4937, 0.3474, 0.6565],
        [0.4576, 0.7070, 0.7154, 0.3474, 0.6654, 0.2935],
        [0.6310, 1.0865, 1.0605, 0.6565, 0.2935, 0.9450]])
```

- Similar to **step 2** previously, we normalize each row so that the values in each row sum to 1:

In [53]:
```
attn_weights = torch.softmax(attn_scores, dim=-1)
print(attn_weights)
```

```
tensor([[0.2098, 0.2006, 0.1981, 0.1242, 0.1220, 0.1452],
        [0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581],
        [0.1390, 0.2369, 0.2326, 0.1242, 0.1108, 0.1565],
        [0.1435, 0.2074, 0.2046, 0.1462, 0.1263, 0.1720],
        [0.1526, 0.1958, 0.1975, 0.1367, 0.1879, 0.1295],
        [0.1385, 0.2184, 0.2128, 0.1420, 0.0988, 0.1896]])
```

- Quick verification that the values in each row indeed sum to 1:

In [55]:
```
row_2_sum = sum([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
print("Row 2 sum:", row_2_sum)

print("All row sums:", attn_weights.sum(dim=-1))
```

```
Row 2 sum: 1.0
All row sums: tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000])
```

- Apply previous **step 3** to compute all context vectors:

In [57]:
```
all_context_vecs = attn_weights @ inputs
print(all_context_vecs)
```

```
tensor([[0.4421, 0.5931, 0.5790],
        [0.4419, 0.6515, 0.5683],
        [0.4431, 0.6496, 0.5671],
        [0.4304, 0.6298, 0.5510],
        [0.4671, 0.5910, 0.5266],
        [0.4177, 0.6503, 0.5645]])
```

- As a sanity check, the previously computed context vector $z^{(2)} = [0.4419, 0.6515, 0.5683]$ can be found in the 2nd row in above:

In [59]:
```
print("Previous 2nd context vector:", context_vec_2)
```

```
Previous 2nd context vector: tensor([0.4419, 0.6515, 0.5683])
```

Exercise (3.3 Self-attention without weights)

Three short product reviews are already encoded as 3-d vectors (row 0 is the product you are looking at; rows 1 and 2 are other reviews). Using the **unweighted** recipe from this section — scores from a dot product with row 0, then softmax — decide which **other** review is closer. Do not reuse `attn_weights` from the demo. Print that index.

In [ ]:
```
import torch

reviews = torch.tensor(
    [
        [0.90, 0.10, 0.00],
        [0.80, 0.20, 0.00],
        [0.00, 0.10, 0.90],
    ],
    dtype=torch.float32,
)
query = reviews[0]
scores = None
weights = None
best_other = None
print('most related other review', best_other)
```

Solution (3.3). Same unweighted recipe as the demo, on a **new** table: scores, softmax, then the largest weight that is not position 0.

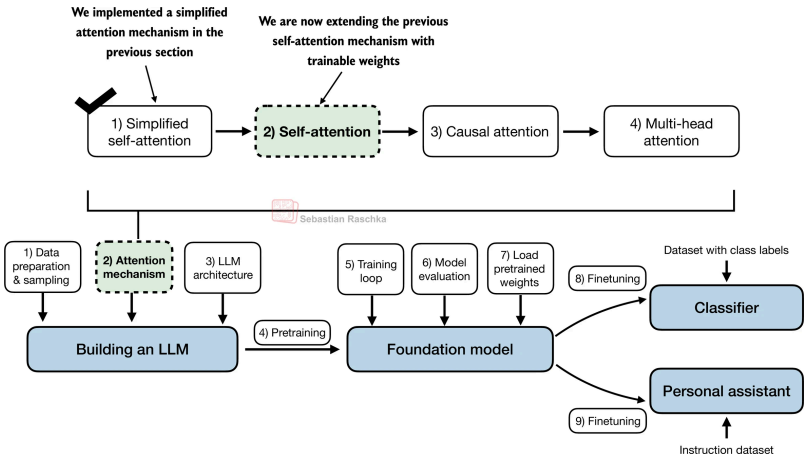
In [1]:
```
import torch

reviews = torch.tensor(
    [
        [0.90, 0.10, 0.00],
        [0.80, 0.20, 0.00],
        [0.00, 0.10, 0.90],
    ],
    dtype=torch.float32,
)
query = reviews[0]
scores = reviews @ query
```

```
weights = torch.softmax(scores, dim=0)
masked = weights.clone()
masked[0] = 0
best_other = int(torch.argmax(masked))
print('most related other review', best_other)
```

```
most related other review 1
```

3.4 Implementing self-attention with trainable weights

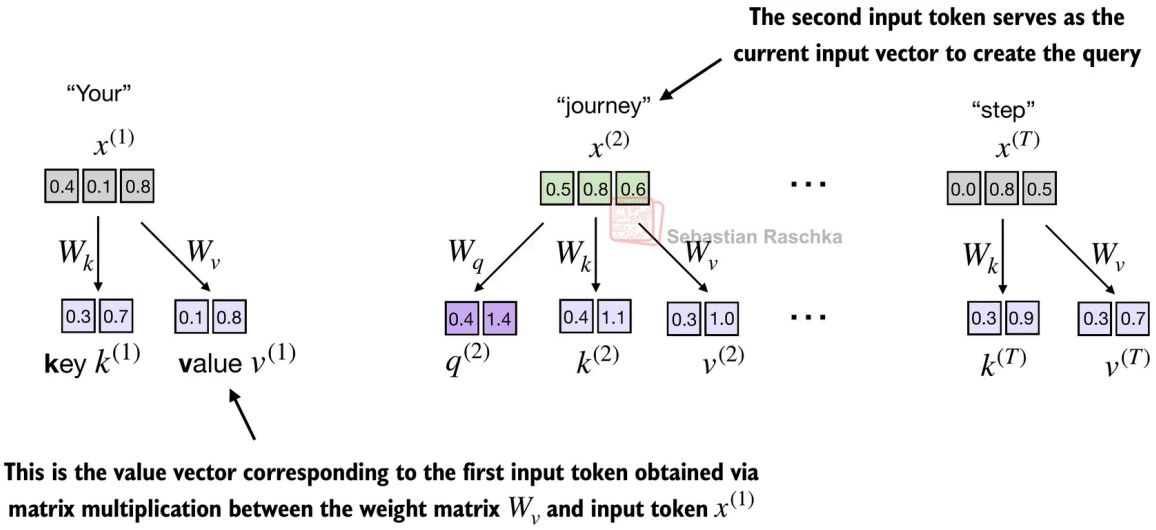
- A conceptual framework illustrating how the self-attention mechanism developed in this section integrates into the overall narrative and structure of this book and chapter



3.4.1 Computing the attention weights step by step

- In this section, we are implementing the self-attention mechanism that is used in the original transformer architecture, the GPT models, and most other popular LLMs
- This self-attention mechanism is also called "scaled dot-product attention"
- The overall idea is similar to before:
  - We want to compute context vectors as weighted sums over the input vectors specific to a certain input element
  - For the above, we need attention weights
- As you will see, there are only slight differences compared to the basic attention mechanism introduced earlier:
  - The most notable difference is the introduction of weight matrices that are updated during model training

- These trainable weight matrices are crucial so that the model (specifically, the attention module inside the model) can learn to produce "good" context vectors



- Implementing the self-attention mechanism step by step, we will start by introducing the three training weight matrices W_q, W_k, and W_v
- These three matrices are used to project the embedded input tokens, $x^{(i)}$, into query, key, and value vectors via matrix multiplication:
  - Query vector: $q^{(i)} = x^{(i)} W_q$
  - Key vector: $k^{(i)} = x^{(i)} W_k$
  - Value vector: $v^{(i)} = x^{(i)} W_v$

- The embedding dimensions of the input x and the query vector q can be the same or different, depending on the model's design and specific implementation
- In GPT models, the input and output dimensions are usually the same, but for illustration purposes, to better follow the computation, we choose different input and output dimensions here:

In [72]:
```
x_2 = inputs[1] # second input element
d_in = inputs.shape[1] # the input embedding size, d=3
d_out = 2 # the output embedding size, d=2
```

- Below, we initialize the three weight matrices; note that we are setting `requires_grad=False` to reduce clutter in the outputs for illustration purposes, but if we were to use the weight matrices for model training, we would set `requires_grad=True` to update these matrices during model training

In [74]:
```
torch.manual_seed(123)

W_query = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
```

```
W_key   = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
W_value = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
```

- Next we compute the query, key, and value vectors:

In [76]:
```
query_2 = x_2 @ W_query # _2 because it's with respect to the 2nd input elem
key_2 = x_2 @ W_key
value_2 = x_2 @ W_value

print(query_2)
```

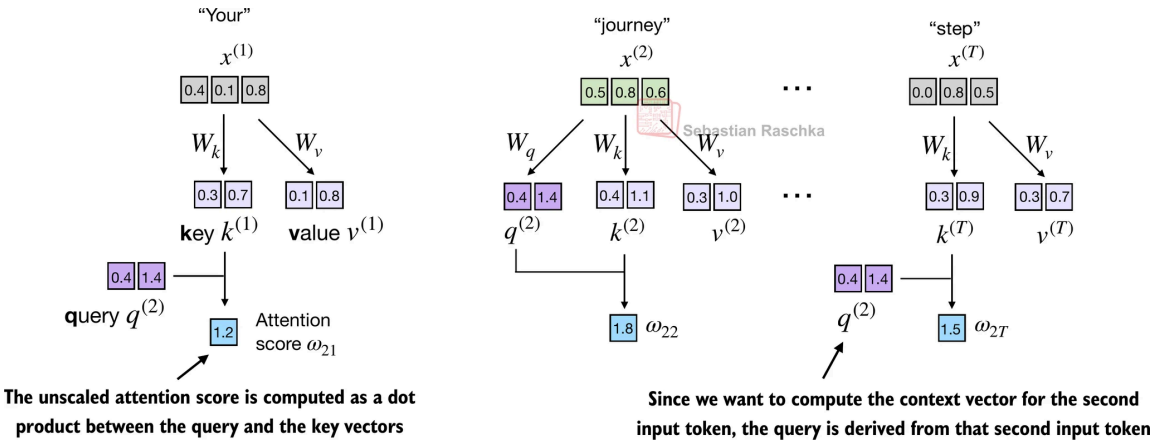
```
tensor([0.4306, 1.4551])
```

- As we can see below, we successfully projected the 6 input tokens from a 3D onto a 2D embedding space:

In [78]:
```
keys = inputs @ W_key
values = inputs @ W_value

print("keys.shape:", keys.shape)
print("values.shape:", values.shape)
```

```
keys.shape: torch.Size([6, 2])
values.shape: torch.Size([6, 2])
```

- In the next step, **step 2**, we compute the unnormalized attention scores by computing the dot product between the query and each key vector:



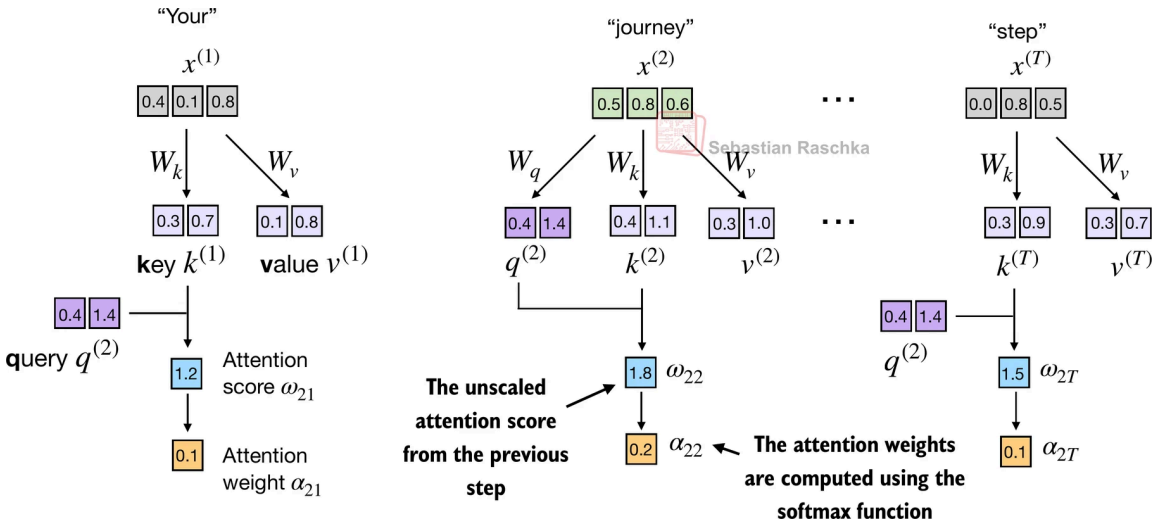
In [81]:
```
keys_2 = keys[1] # Python starts index at 0
attn_score_22 = query_2.dot(keys_2)
print(attn_score_22)
```

```
tensor(1.8524)
```

- Since we have 6 inputs, we have 6 attention scores for the given query vector:

In [83]:
```
attn_scores_2 = query_2 @ keys.T # All attention scores for given query
print(attn_scores_2)
```

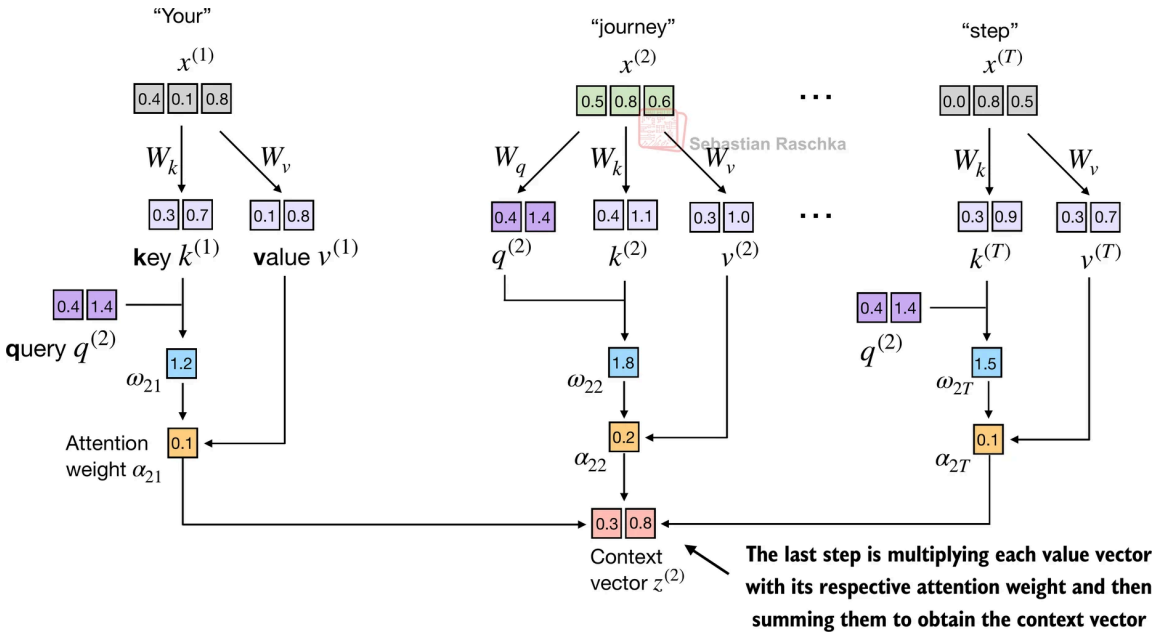
```
tensor([1.2705, 1.8524, 1.8111, 1.0795, 0.5577, 1.5440])
```



- Next, in **step 3**, we compute the attention weights (normalized attention scores that sum up to 1) using the softmax function we used earlier
- The difference to earlier is that we now scale the attention scores by dividing them by the square root of the embedding dimension, $\sqrt{d_k}$ (i.e., `d_k**0.5`):

In [86]:
```
d_k = keys.shape[1]
attn_weights_2 = torch.softmax(attn_scores_2 / d_k**0.5, dim=-1)
print(attn_weights_2)
```

```
tensor([0.1500, 0.2264, 0.2199, 0.1311, 0.0906, 0.1820])
```



- In **step 4**, we now compute the context vector for input query vector 2:

In [89]:
```
context_vec_2 = attn_weights_2 @ values
print(context_vec_2)
```

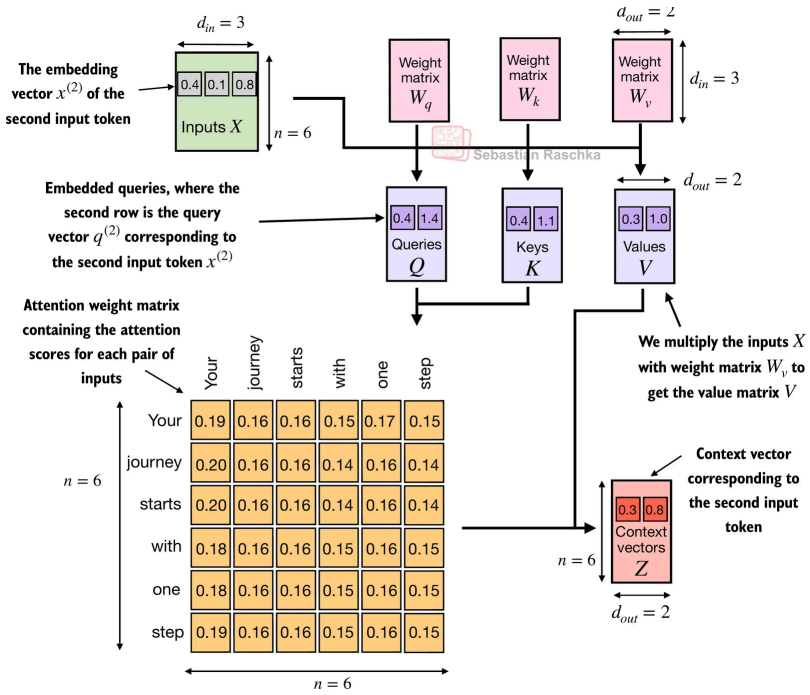
```
tensor([0.3061, 0.8210])
```

3.4.2 Implementing a compact SelfAttention class

- Putting it all together, we can implement the self-attention mechanism as follows:

In [92]:
```python
import torch.nn as nn

class SelfAttention_v1(nn.Module):

    def __init__(self, d_in, d_out):
        super().__init__()
        self.W_query = nn.Parameter(torch.rand(d_in, d_out))
        self.W_key   = nn.Parameter(torch.rand(d_in, d_out))
        self.W_value = nn.Parameter(torch.rand(d_in, d_out))

    def forward(self, x):
        keys = x @ self.W_key
        queries = x @ self.W_query
        values = x @ self.W_value

        attn_scores = queries @ keys.T # omega
        attn_weights = torch.softmax(
            attn_scores / keys.shape[-1]**0.5, dim=-1
        )

        context_vec = attn_weights @ values
        return context_vec

torch.manual_seed(123)
sa_v1 = SelfAttention_v1(d_in, d_out)
print(sa_v1(inputs))
```

```
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]], grad_fn=<MmBackward0>)
```



- We can streamline the implementation above using PyTorch's Linear layers, which are equivalent to a matrix multiplication if we disable the bias units
- Another big advantage of using `nn.Linear` over our manual `nn.Parameter(torch.rand(...)` approach is that `nn.Linear` has a preferred weight initialization scheme, which leads to more stable model training

In [95]:
```
class SelfAttention_v2(nn.Module):

    def __init__(self, d_in, d_out, qkv_bias=False):
        super().__init__()
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key   = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)

    def forward(self, x):
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)

        attn_scores = queries @ keys.T
        attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim=

        context_vec = attn_weights @ values
        return context_vec

torch.manual_seed(789)
sa_v2 = SelfAttention_v2(d_in, d_out)
print(sa_v2(inputs))
```

```
tensor([[-0.0739,  0.0713],
        [-0.0748,  0.0703],
        [-0.0749,  0.0702],
        [-0.0760,  0.0685],
        [-0.0763,  0.0679],
        [-0.0754,  0.0693]], grad_fn=<MmBackward0>)
```

- Note that `SelfAttention_v1` and `SelfAttention_v2` give different outputs because they use different initial weights for the weight matrices

Exercise (3.4 Trainable Q, K, V)

A 3-token window and small learned `W_q` / `W_k` are given below (not the demo matrices). Project token 0 into a query and every token into a key, then print which token the learned query looks at (largest dot product).

In [ ]:
```python
import torch

window = torch.tensor([[1.0, 0.0], [0.0, 1.0], [1.0, 1.0]])
W_q = torch.tensor([[1.0, 0.0], [0.0, 1.0]])
W_k = torch.tensor([[0.0, 1.0], [1.0, 0.0]])
q0 = None
keys = None
dots = None
looks_at = None
print('token 0 looks at', looks_at)
```

Solution (3.4). `q0 = window[0] @ W_q`, `keys = window @ W_k`, then argmax of the dots.

In [1]:
```python
import torch

window = torch.tensor([[1.0, 0.0], [0.0, 1.0], [1.0, 1.0]])
W_q = torch.tensor([[1.0, 0.0], [0.0, 1.0]])
W_k = torch.tensor([[0.0, 1.0], [1.0, 0.0]])
q0 = window[0] @ W_q
keys = window @ W_k
dots = keys @ q0
looks_at = int(torch.argmax(dots))
print('token 0 looks at', looks_at)
```

```
token 0 looks at 1
```

Comprehensive exercises

Exercise (comprehensive)

On the same 3-token `window` as 3.4, compare **unweighted** 3.3 (dot with token 0, softmax) against **learned** 3.4 (`W_q` / `W_k` from that exercise). Print the other-token

index each method picks for token 0. Did the projection change who we listen to?

In [ ]:
```
import torch

window = torch.tensor([[1.0, 0.0], [0.0, 1.0], [1.0, 1.0]])
W_q = torch.tensor([[1.0, 0.0], [0.0, 1.0]])
W_k = torch.tensor([[0.0, 1.0], [1.0, 0.0]])
unweighted_other = None
learned_other = None
print('unweighted other', unweighted_other)
print('learned other', learned_other)
```

Solution. Unweighted: softmax of `window @ window[0]`, hide index 0. Learned: same hide-self step after `W_q` / `W_k` dots.

In [1]:
```
import torch

window = torch.tensor([[1.0, 0.0], [0.0, 1.0], [1.0, 1.0]])
W_q = torch.tensor([[1.0, 0.0], [0.0, 1.0]])
W_k = torch.tensor([[0.0, 1.0], [1.0, 0.0]])

w = torch.softmax(window @ window[0], dim=0).clone()
w[0] = 0
unweighted_other = int(torch.argmax(w))

dots = (window @ W_k) @ (window[0] @ W_q)
lw = torch.softmax(dots, dim=0).clone()
lw[0] = 0
learned_other = int(torch.argmax(lw))
print('unweighted other', unweighted_other)
print('learned other', learned_other)
```

```
unweighted other 2
learned other 1
```

Takeaways

1. Self-attention: every token is a **query** that scores **keys**, softmax, then a weighted sum of **values**.
2. A row of attention weights adds to 1.
3. Trainable attention uses `W_query`, `W_key`, and `W_value`.
4. Next class: hide the future (causal), then several heads.