CS 541-A-Homework 4

Self attention

Fill your details below

Name: Sneha Venkatesh

CWID: 20027527

Email ID: svenkate1@stevens.edu

References: ***Cite your references here***

Submission guidelines:

1. Submit this notebook along with its PDF version. You can do this by clicking File->Print->"Save as PDF"

2. Name the file as "<mailID_HWnumber.extension>". For example, mailID is abcdefg @stevens.edu then name the files as abcdefg_HW1.ipynb and abcdefg_HW1.pdf.

3. Please do not Zip your files.

Illustrated: Self-Attention

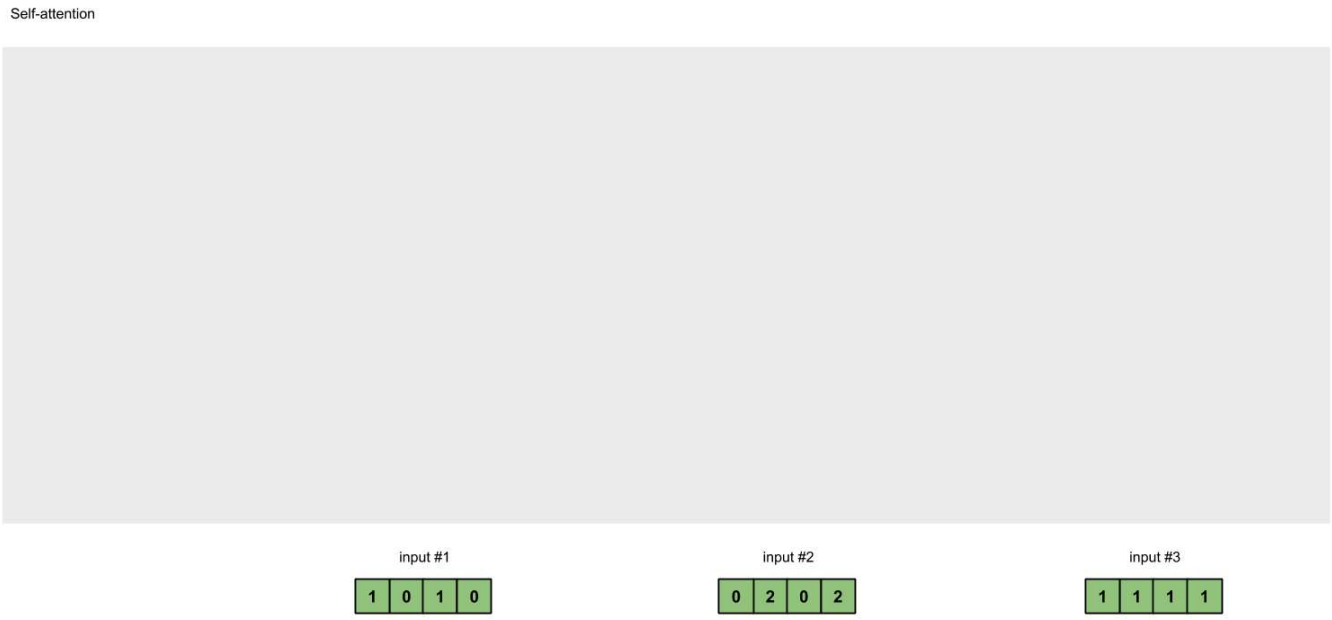
Step-by-step guide to self-attention with illustrations and code

Reference

medium article

Article author

Colab reference: Manuel Romero



What do *BERT, RoBERTa, ALBERT, SpanBERT, DistilBERT, SesameBERT, SemBERT, MobileBERT, TinyBERT and CamemBERT* all have in common? And I'm not looking for the answer "BERT" 🤭. Answer: **self-attention** 🤗. We are not only talking about architectures bearing the name "BERT', but more correctly **Transformer-based architectures**. Transformer-based architectures, which are primarily used in modelling language understanding tasks, eschew the use of recurrence in neural network (RNNs) and instead trust entirely on self-attention mechanisms to draw global dependencies between inputs and outputs. But what's the math behind this?

The main content of this kernel is to walk you through the mathematical operations involved in a self-attention module.

Step 0. What is self-attention?

If you're thinking if self-attention is similar to attention, then the answer is yes! They fundamentally share the same concept and many common mathematical operations. A self-attention module takes in n inputs, and returns n outputs. What happens in this module? In layman's terms, the self-attention mechanism allows the inputs to interact with each other ("self") and find out who they should pay more attention to ("attention"). The outputs are aggregates of these interactions and attention scores.

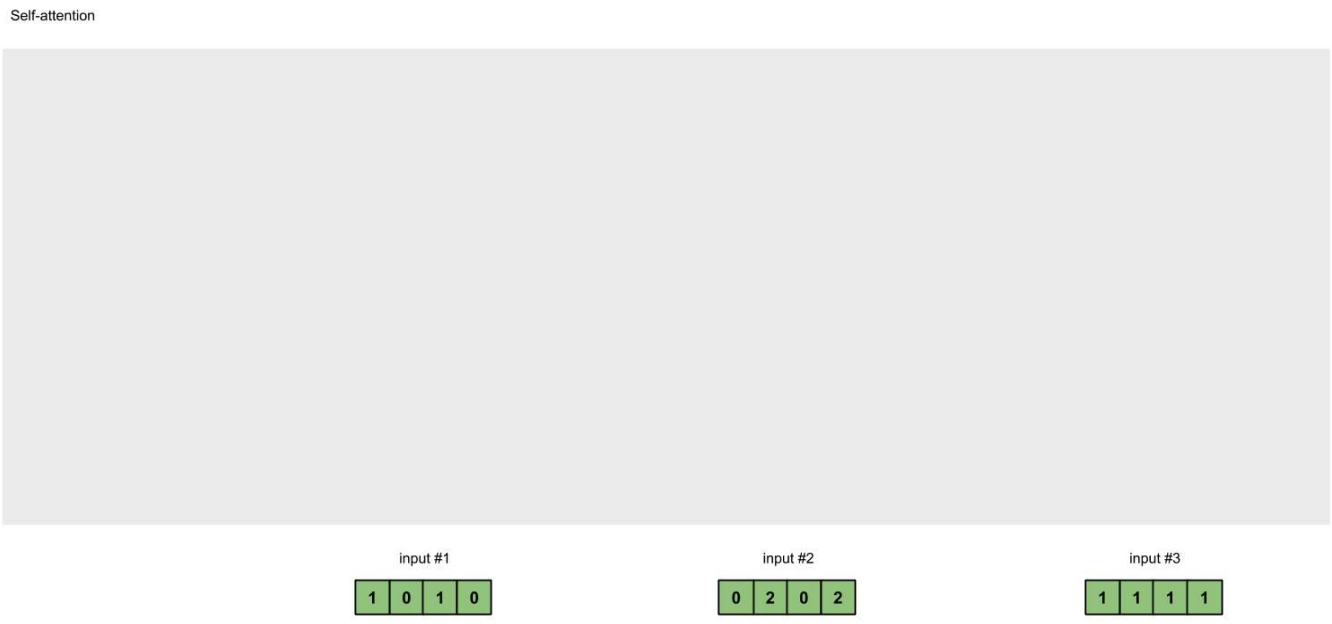
Following, we are going to explain and implement:

- Prepare inputs
- Initialise weights
- Derive key, query and value
- Calculate attention scores for Input 1
- Calculate softmax
- Multiply scores with values
- Sum weighted values to get Output 1
- Repeat steps 4–7 for Input 2 & Input 3

```
1 import torch
```

Step 1: Prepare inputs

For this tutorial, for the sake of simplicity, we start with 3 inputs, each with dimension 4.

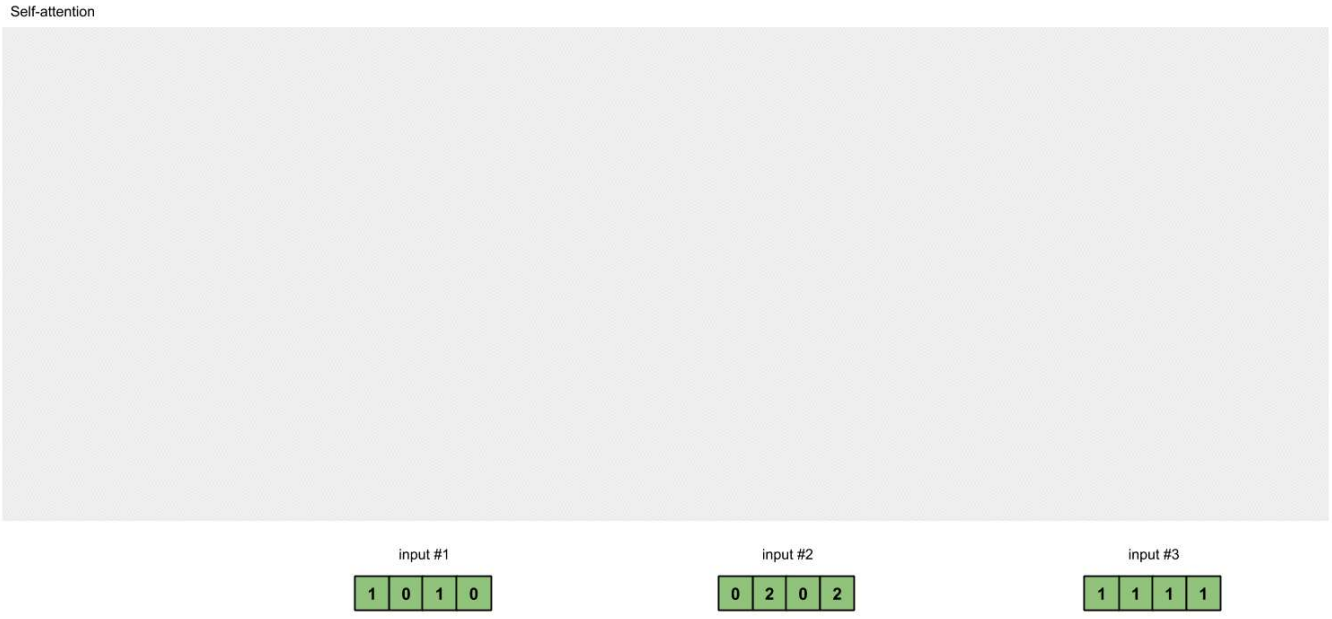


```
x = [
  [1, 0, 1, 0], # Input 1
  [0, 2, 0, 2], # Input 2
  [1, 1, 1, 1]  # Input 3
 ]
x = torch.tensor(x, dtype=torch.float32)
x
```

```
tensor([[1., 0., 1., 0.],
        [0., 2., 0., 2.],
        [1., 1., 1., 1.]])
```

Step 2: Initialise weights

Every input must have three representations (see diagram below). These representations are called **key** (orange), **query** (red), and **value** (purple). For this example, let's take that we want these representations to have a dimension of 3. Because every input has a dimension of 4, this means each set of the weights must have a shape of 4×3.



```
w_key = [
  [0, 0, 1],
  [1, 1, 0],
  [0, 1, 0],
  [1, 1, 0]
]
w_query = [
  [1, 0, 1],
  [1, 0, 0],
  [0, 0, 1],
  [0, 1, 1]
]
w_value = [
  [0, 2, 0],
  [0, 3, 0],
  [1, 0, 3],
  [1, 1, 0]
]
w_key = torch.tensor(w_key, dtype=torch.float32)
w_query = torch.tensor(w_query, dtype=torch.float32)
w_value = torch.tensor(w_value, dtype=torch.float32)

print("Weights for key: \n", w_key)
print("Weights for query: \n", w_query)
print("Weights for value: \n", w_value)
```

```
Weights for key:
 tensor([[0., 0., 1.],
        [1., 1., 0.],
        [0., 1., 0.],
        [1., 1., 0.]])
Weights for query:
 tensor([[1., 0., 1.],
        [1., 0., 0.],
        [0., 0., 1.],
        [0., 1., 1.]])
Weights for value:
 tensor([[0., 2., 0.],
        [0., 3., 0.],
        [1., 0., 3.],
        [1., 1., 0.]])
```

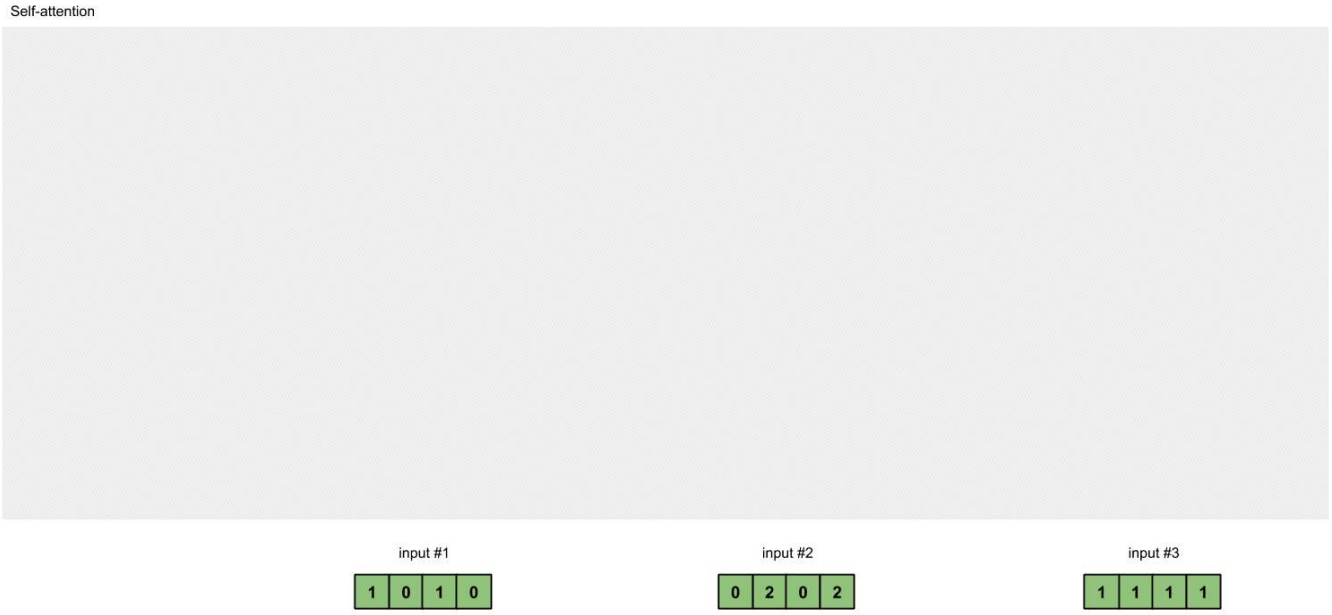
Note: *In a neural network setting, these weights are usually small numbers, initialised randomly using an appropriate random distribution like Gaussian, Xavier and Kaiming distributions.*

Step 3: Derive key, query and value

Now that we have the three sets of weights, let's actually obtain the **key**, **query** and **value** representations for every input.

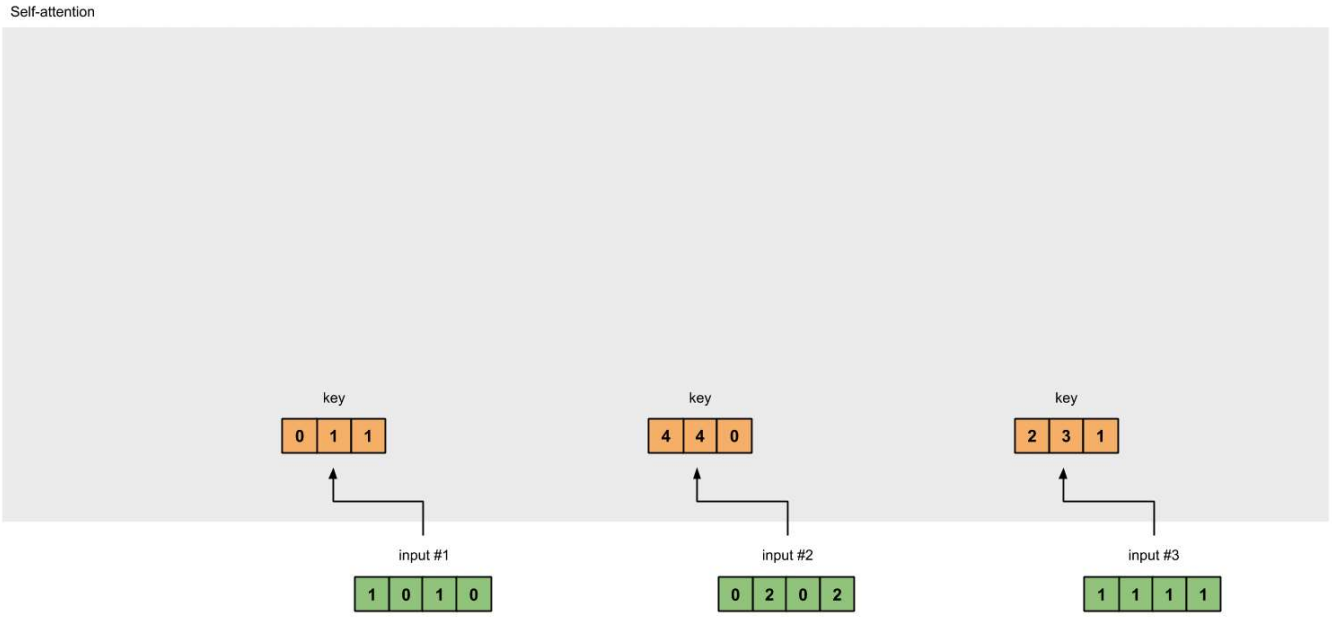
Obtaining the keys:

```
                [0, 0, 1]
[1, 0, 1, 0]    [1, 1, 0]   [0, 1, 1]
[0, 2, 0, 2] x  [0, 1, 0] = [4, 4, 0]
[1, 1, 1, 1]    [1, 1, 0]   [2, 3, 1]
```



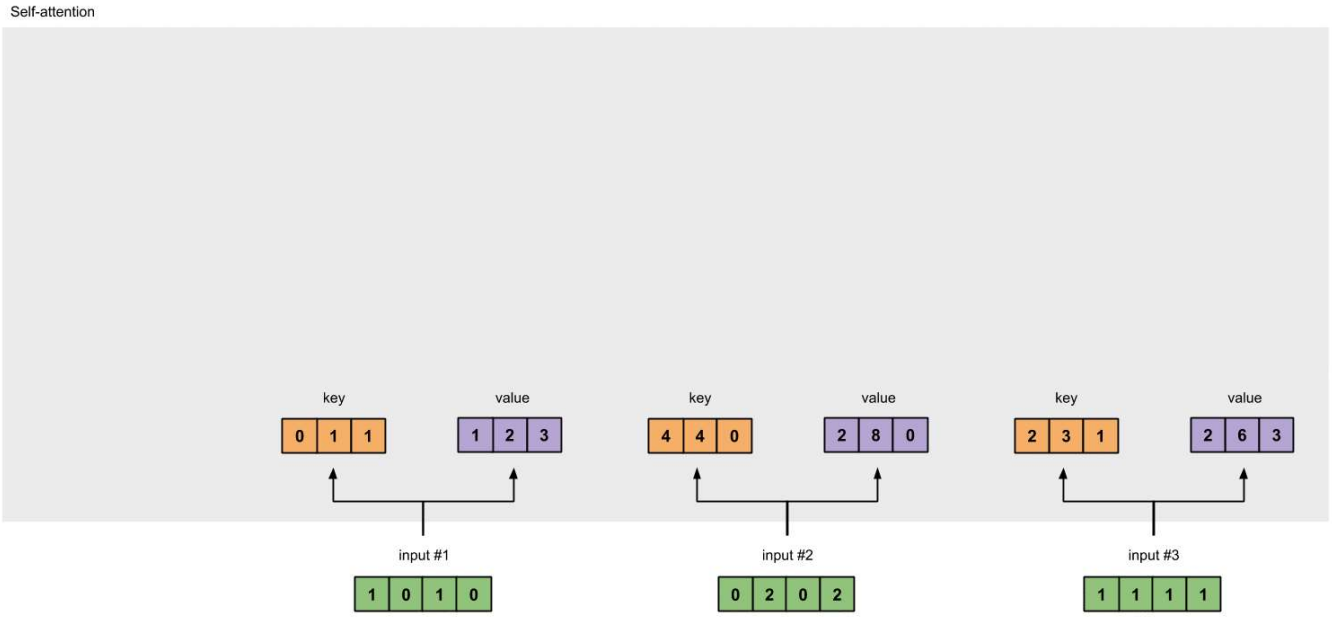
Obtaining the values:

```
                [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]
```



Obtaining the querys:

```
               [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]
```



Notes: *Notes In practice, a bias vector may be added to the product of matrix multiplication.*

Q1. (45 points)

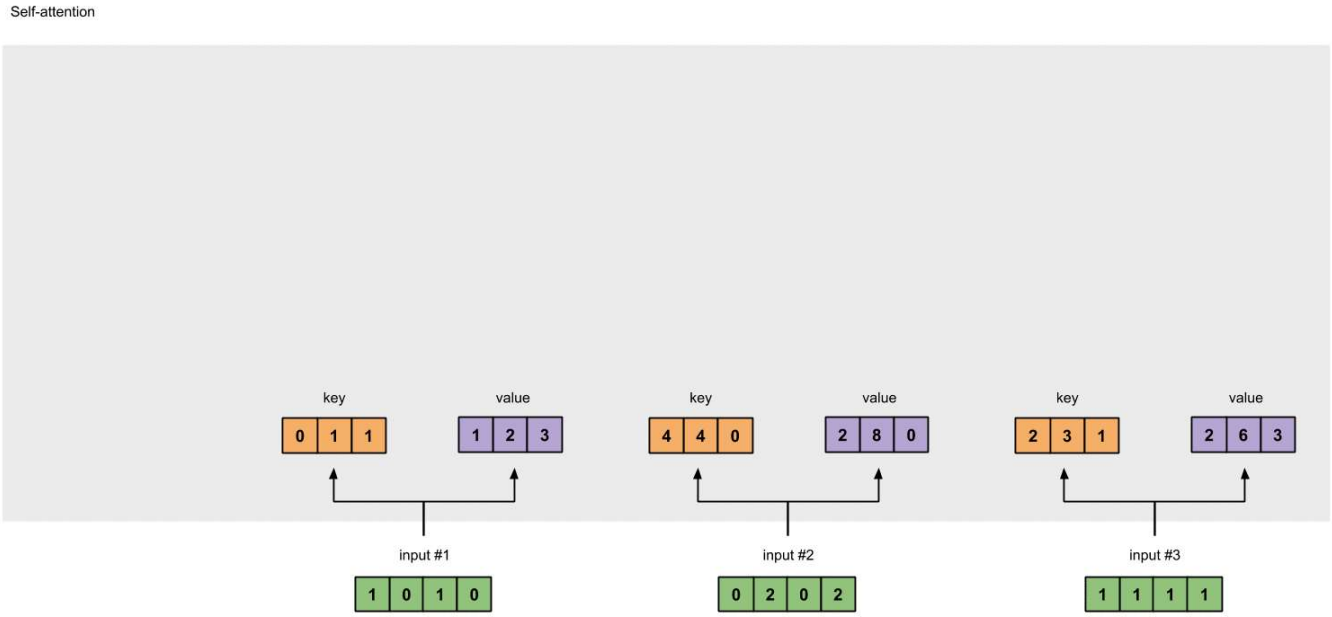
1. Obtain and print the keys. (15 points)
2. Obtain and print the querys. (15 points)
3. Obtain and print the values. (15 points)

The correct values for all 3 have been commented in the cell below so you can verify your output.

```
keys = torch.matmul(x, w_key) #x @ w_key
querys = torch.matmul(x, w_query) #x @ w_query
values = torch.matmul(x, w_value) #x @ w_value

print("Keys: \n", keys)
# tensor([[0., 1., 1.],
#         [4., 4., 0.],
#         [2., 3., 1.]])

print("Querys: \n", querys)
# tensor([[1., 0., 2.],
#         [2., 2., 2.],
#         [2., 1., 3.]])
print("Values: \n", values)
# tensor([[1., 2., 3.],
#         [2., 8., 0.],
#         [2., 6., 3.]])
```

```
Keys:
 tensor([[0., 1., 1.],
        [4., 4., 0.],
        [2., 3., 1.]])
Querys:
 tensor([[1., 0., 2.],
        [2., 2., 2.],
        [2., 1., 3.]])
Values:
 tensor([[1., 2., 3.],
        [2., 8., 0.],
        [2., 6., 3.]])
```

Step 4: Calculate attention scores



To obtain **attention scores**, we start off with taking a dot product between Input 1's **query** (red) with **all keys** (orange), including itself. Since there are 3 key representations (because we have 3 inputs), we obtain 3 attention scores (blue).

```
            [0, 4, 2]
[1, 0, 2] x [1, 4, 3] = [2, 4, 4]
            [1, 0, 1]
```

Notice that we only use the query from Input 1. Later we'll work on repeating this same step for the other querys.

Note: *The above operation is known as dot product attention, one of the several score functions. Other score functions include scaled dot product and additive/concat.*

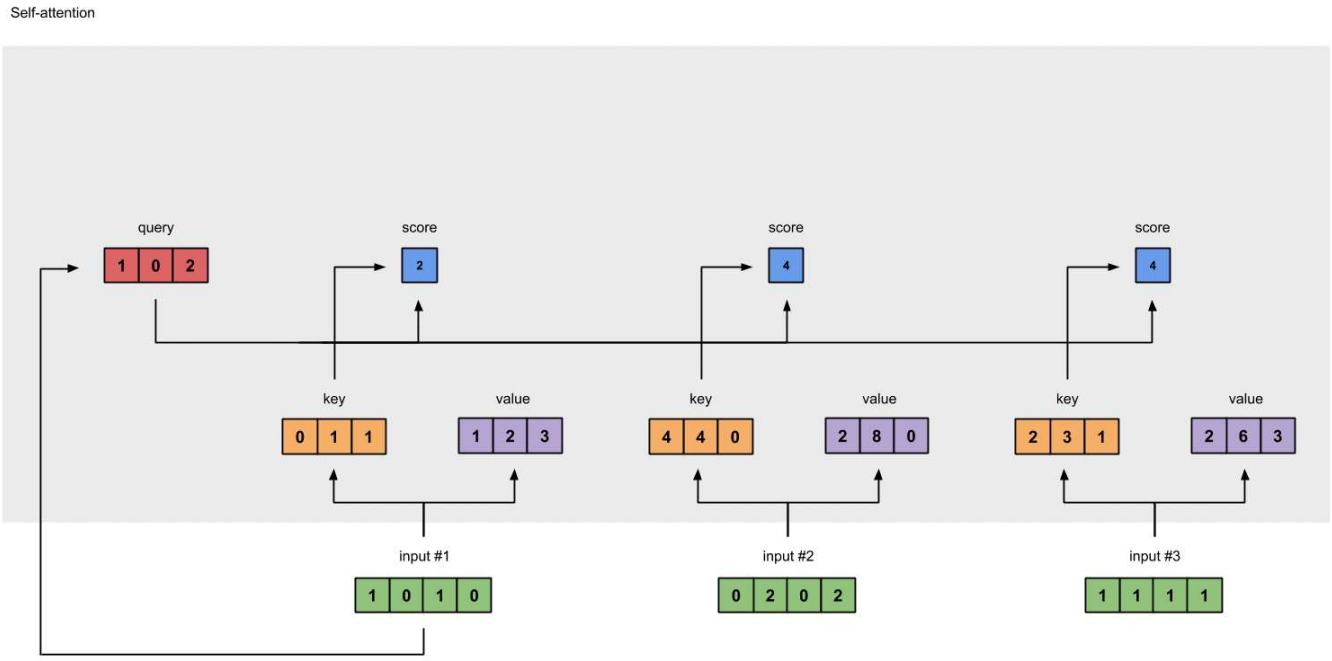
Q2. Calculate and print the attention scores. (25 points)

Correct output has been commented for verification.

```
attn_scores = torch.matmul(querys, keys.T) #querys @ keys.T

print(attn_scores)

# tensor([[ 2.,  4.,  4.],  # attention scores from Query 1
#         [ 4., 16., 12.],  # attention scores from Query 2
#         [ 4., 12., 10.]]) # attention scores from Query 3
```

```
tensor([[ 2.,  4.,  4.],
        [ 4., 16., 12.],
        [ 4., 12., 10.]])
```

Step 5: Calculate softmax



Take the **softmax** across these **attention scores** (blue).

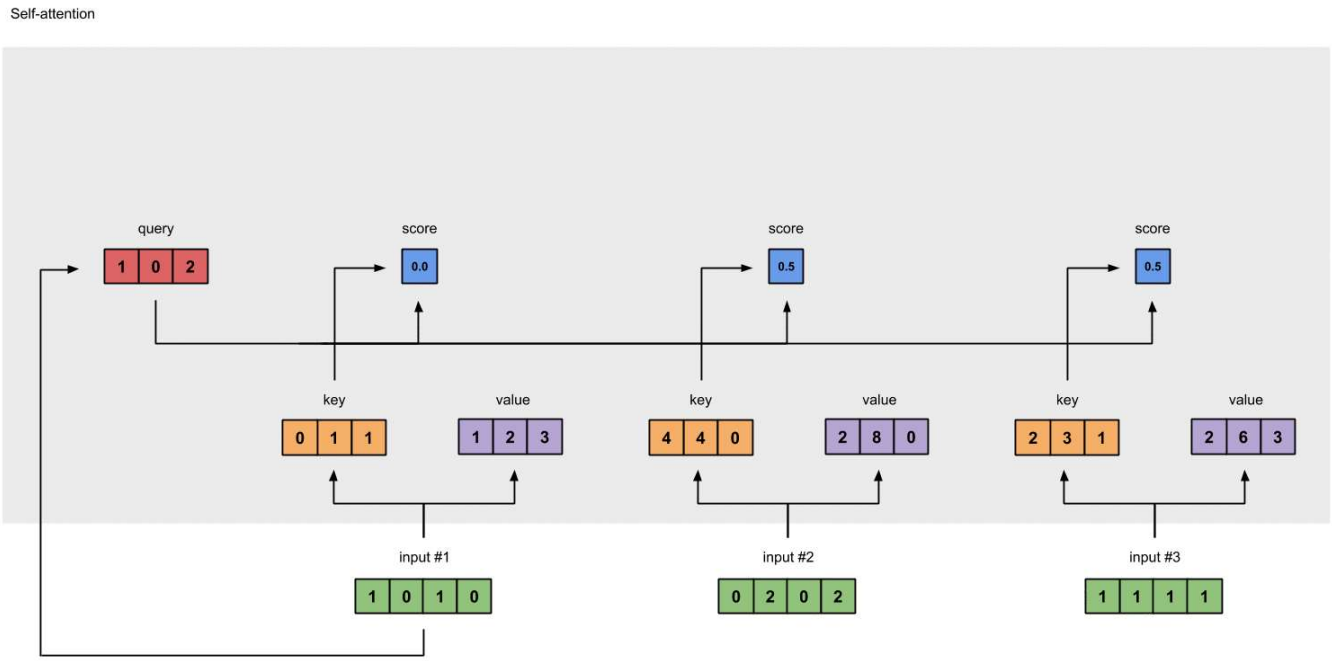
```
softmax([2, 4, 4]) = [0.0, 0.5, 0.5]
```

Q3. Calculate and print the softmax of the attention scores. (30 points)

```
from torch.nn.functional import softmax

# Q3 30 points

attn_scores_softmax = softmax(attn_scores, dim=-1)

print(attn_scores_softmax)
# tensor([[6.3379e-02, 4.6831e-01, 4.6831e-01],
#         [6.0337e-06, 9.8201e-01, 1.7986e-02],
#         [2.9539e-04, 8.8054e-01, 1.1917e-01]])

# For readability, approximate the above as follows
attn_scores_softmax = [
  [0.0, 0.5, 0.5],
  [0.0, 1.0, 0.0],
  [0.0, 0.9, 0.1]
]
attn_scores_softmax = torch.tensor(attn_scores_softmax)
print(attn_scores_softmax)
```

```
tensor([[6.3379e-02, 4.6831e-01, 4.6831e-01],
        [6.0337e-06, 9.8201e-01, 1.7986e-02],
        [2.9539e-04, 8.8054e-01, 1.1917e-01]])
tensor([[0.0000, 0.5000, 0.5000],
        [0.0000, 1.0000, 0.0000],
        [0.0000, 0.9000, 0.1000]])
```

Step 6: Multiply scores with values



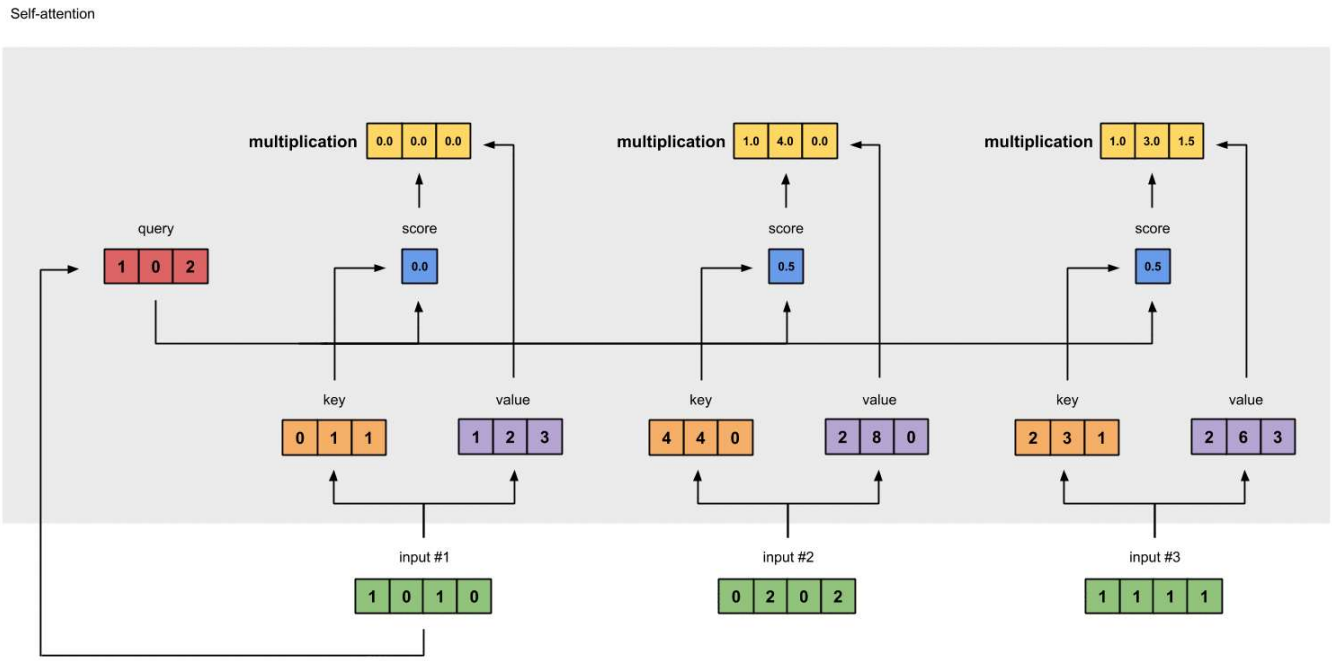
The softmaxed attention scores for each input (blue) is multiplied with its corresponding **value** (purple). This results in 3 alignment vectors (yellow). In this tutorial, we'll refer to them as **weighted values**.

```
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]
```

```
1 weighted_values = values[:,None] * attn_scores_softmax.T[:,:,None]
2 print(weighted_values)
```

```
    tensor([[[0.0000, 0.0000, 0.0000],
             [0.0000, 0.0000, 0.0000],
             [0.0000, 0.0000, 0.0000]],

            [[1.0000, 4.0000, 0.0000],
             [2.0000, 8.0000, 0.0000],
             [1.8000, 7.2000, 0.0000]],

            [[1.0000, 3.0000, 1.5000],
             [0.0000, 0.0000, 0.0000],
             [0.2000, 0.6000, 0.3000]]])
```

Step 7: Sum weighted values



Take all the **weighted values** (yellow) 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]
```

The resulting vector `[2.0, 7.0, 1.5]` (dark green) **is Output 1**, which is based on the **query representation from Input 1** interacting with all other keys, including itself.

Step 8: Repeat for Input 2 & Input 3


```
1 outputs = weighted_values.sum(dim=0)
2 print(outputs)
3
4 # tensor([[2.0000, 7.0000, 1.5000],  # Output 1
5 #         [2.0000, 8.0000, 0.0000],  # Output 2
6 #         [2.0000, 7.8000, 0.3000]]) # Output 3
```

```
tensor([[2.0000, 7.0000, 1.5000],
        [2.0000, 8.0000, 0.0000],
        [2.0000, 7.8000, 0.3000]])
```