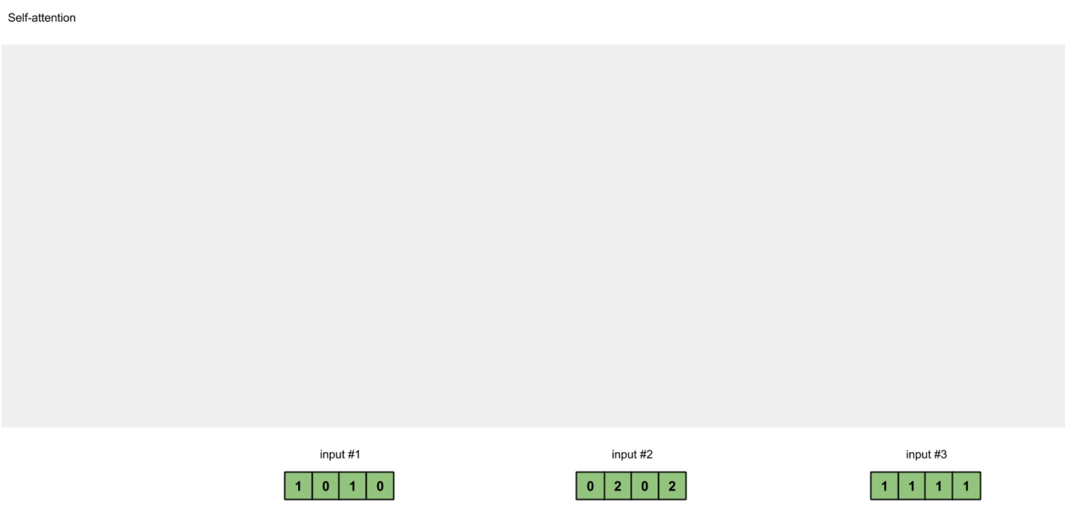
towardsdatascience.com

Illustrated: Self-Attention - Towards Data Science

Raimi Karim

9-12 minutes



Inside AI

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

The illustrations are best viewed on Desktop. A Colab version can be found here, (thanks to Manuel Romero!).

What do BERT, RoBERTa, ALBERT, SpanBERT, DistilBERT, SesameBERT, SemBERT, SciBERT, BioBERT, 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 and instead trust entirely on **self-attention** mechanisms to draw global dependencies between inputs and outputs. But what's the math behind this?

That's what we're going to find out today. The main content of this post is to walk you through the mathematical operations involved in a self-attention module. By the end of this article, you should be able to write or code a self-attention module from scratch.

This article does not aim to provide the intuitions and explanations behind the different numerical representations and mathematical operations in the self-attention module. It also does not aim to demonstrate the why's and how-exactly's of self-attention in Transformers (I believe there's a lot out there already). Note that the difference between attention and self-attention is also not detailed in this article.

Content

Now let's get on to it!

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.

1. Illustrations

The illustrations are divided into the following steps:

1. Prepare inputs
2. Initialise weights
3. Derive **key**, **query** and **value**
4. Calculate attention scores for Input 1
5. Calculate softmax
6. Multiply scores with **values**
7. Sum **weighted values** to get Output 1

8. Repeat steps 4–7 for Input 2 & Input 3

> ***Note***
> *In practice, the mathematical operations are vectorised, i.e. all the inputs undergo the mathematical operations together. We'll see this later in the Code section.*

Step 1: Prepare inputs

Fig. 1.1: Prepare inputs

For this tutorial, we start with 3 inputs, each with dimension 4.

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

> ***Note***
> *We'll see later that the dimension of **value** is also the dimension of the output.*

Fig. 1.2: Deriving **key**, **query** and **value** representations from each input

In order to obtain these representations, every input (green) is multiplied with a set of weights for **keys**, a set of weights for **querys** (I know that's not the right spelling), and a set of weights for **values**. In our example, we initialise the three sets of weights as follows.

Weights for **key**:

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

Weights for **query**:

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

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

Weights for **value**:

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

> ***Notes***
> *In a neural network setting, these weights are usually small numbers, initialised randomly using an appropriate random distribution like Gaussian, Xavier and Kaiming distributions. This initialisation is done once before training.*

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.

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]
```

Use the same set of weights to get the **key** representation for Input 2:

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

Use the same set of weights to get the **key** representation for Input 3:

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

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

A faster way is to vectorise the above operations:

```
               [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]
```

Fig. 1.3a: Derive **key** representations from each input

Let's do the same to obtain the **value** representations for every input:

```
               [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]
```

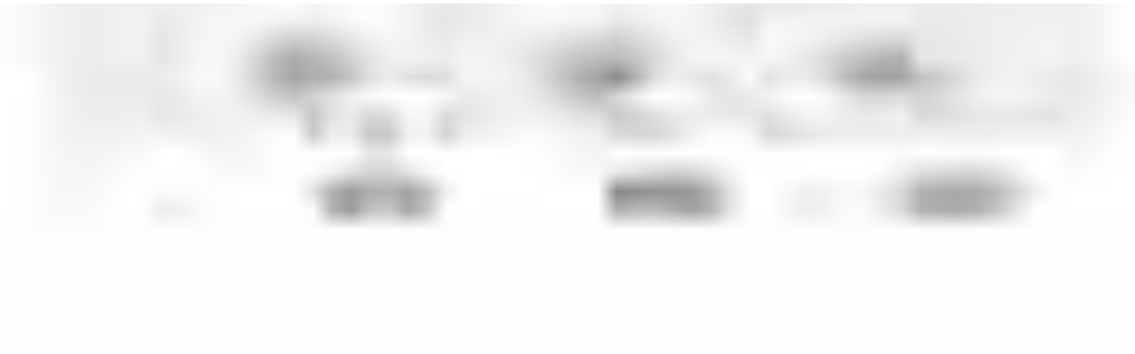


Fig. 1.3b: Derive **value** representations from each input

and finally the **query** representations:

```
                  [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]
```

Fig. 1.3c: Derive **query** representations from each input

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

Step 4: Calculate attention scores for Input 1

Fig. 1.4: Calculating attention scores (blue) from query 1

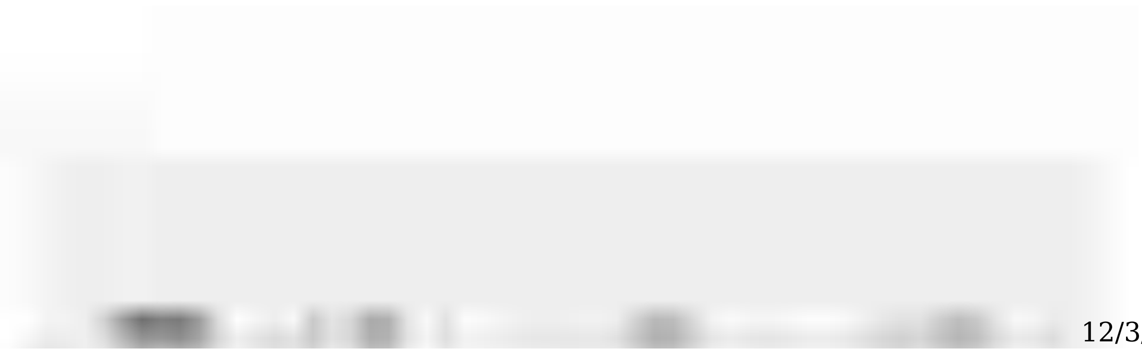
To obtain *attention scores*, we start off with taking a dot product between Input 1's **query** (red) with all **keys** (orange), including itself. Since there are 3 **key** representations (because we have 3 inputs), we obtain 3 attention scores (blue).

```
            [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.

Step 5: Calculate softmax



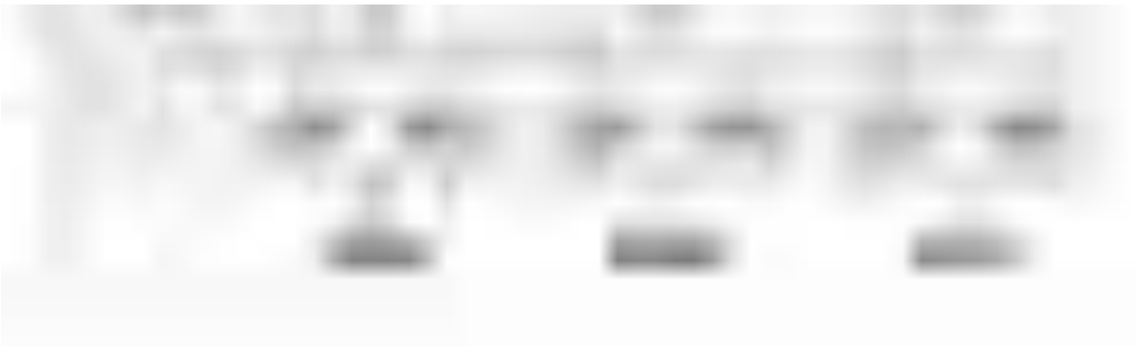


Fig. 1.5: Softmax the attention scores (blue)

Take the softmax across these attention scores (blue).

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

Step 6: Multiply scores with values

Fig. 1.6: Derive **weighted value** representation (yellow) from multiply **value** (purple) and score (blue)

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]
```

Step 7: Sum weighted values to get Output 1

Fig. 1.7: Sum all **weighted values** (yellow) to get Output 1 (dark green)

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

Now that we're done with Output 1, we repeat Steps 4 to 7 for Output 2 and Output 3. I trust that I can leave you to work out the operations yourself 👍🏼.

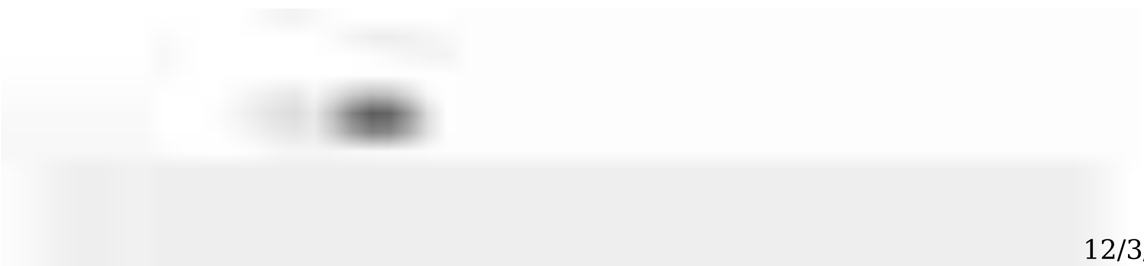


Fig. 1.8: Repeat previous steps for Input 2 & Input 3

> ***Notes***
> *The dimension of **query** and **key** must always be the same because of the dot product score function. However, the dimension of **value** may be different from **query** and **key**. The resulting output will consequently follow the dimension of **value**.*

2. Code

Here is the code in PyTorch 🤗, a popular deep learning framework in Python. To enjoy the APIs for `@` operator, `.T` and **`None`** indexing in the following code snippets, make sure you're on Python≥3.6 and PyTorch 1.3.1. Just follow along and copy paste these in a Python/IPython REPL or Jupyter Notebook.

Step 1: Prepare inputs

Step 2: Initialise weights

Step 3: Derive key, query and value

Step 4: Calculate attention scores

Step 5: Calculate softmax

Step 6: Multiply scores with values

Step 7: Sum weighted values

> ***Note***

> *PyTorch has provided an API for this called `nn.MultiheadAttention`. However, this API requires that you feed in key, query and value PyTorch tensors. Moreover, the outputs of this module undergo a linear transformation.*

3. Extending to Transformers

So, where do we go from here? Transformers! Indeed we live in exciting times of deep learning research and high compute resources. Transformer is the incarnation from Attention Is All You Need, orginally born to perform neural machine translation. Researchers picked up from here, reassembling, cutting, adding and extending the parts, and extend its usage to more language tasks.

Here I will briefly mention how we can extend self-attention to a Transformer architecture.

Within the self-attention module:

- Dimension
- Bias

Inputs to the self-attention module:

- Embedding module
- Positional encoding
- Truncating
- Masking

Adding more self-attention modules:

- Multihead
- Layer stacking

Modules between self-attention modules:

- Linear transformations

- LayerNorm

That's all folks! Hope you find the content easy to digest. Is there something that you think I should add or elaborate further in this article? Do drop a comment! Also, do check out an illustration I created for attention below

References

Attention Is All You Need (arxiv.org)

The Illustrated Transformer (jalammar.github.io)

Related Articles

Attn: Illustrated Attention (towardsdatascience.com)

Credits

Special thanks to Xin Jie, Serene, Ren Jie, Kevin and Wei Yih for ideas, suggestions and corrections to this article.

Follow me on Twitter @remykarem for digested articles and other tweets on AI, ML, Deep Learning and Python.