# Attention Is All You Need

October 3, 2023

## 1 Decrypting Transformer

### 1.1 What is Self-Attention?

As the name of the paper 'Attention Is All You Need' implies, **Transformer** model basically works as a sequence of **attention** modules. In the paper, **Scaled Dot Product Attention** is introduced, which revolves around the interplay of three components: **key**, **query**, and **value**. These are vital for understanding how information is weighted and propagated in attention models, such as the Transformer.

#### 1.1.1 Self-Attention as a Matrix Multiplication

$$\textbf{Attention}(Q, K, V) = \textbf{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \cdot V \tag{1}$$

When $Q = K$, the term $QK^T$ captures the self-attention, indicating how similar elements within the matrix $Q$ are to one another. Let's break this equation down into parts. An equation $\frac{QK^T}{\sqrt{d_k}} \cdot V$ suggests that a given matrix $Q$ and its transpose $K$ are multiplied, and this attention weight (level of similarity) is multiplied into value. The algorithm is basically so simple, that it becomes new $Q$ to be multiplied. And it becomes the new query of next layer, that deepens the quality of model.

#### 1.1.2 Scaling Factor

You might have noticed that I did not mention $\sqrt{d_k}$. Why is it used as a scaling factor? It requires a form of proof. Under the assumption that the components of $q$ and $k$ are independent random variables with mean 0 and variance 1 (it is quite theoretical assumption that is not realistic for most cases), their dot product, $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$ has mean 0 and variance $d_k$.

The mean can be determined using the **linearity of expectation**:

$$E[q \cdot k] = E\left[\sum_{i=1}^{d_k} q_i k_i\right]$$

$$= \sum_{i=1}^{d_k} E[q_i k_i]$$

Given the assumption that random variables are i.i.d (independently identically distributed):

$$= \sum_{i=1}^{d_k} E[q_i]E[k_i] = 0$$

Thus, the mean of $q \cdot k$ equals 0.

For variance, although variance is not strictly linear in the way that expectation is, in this context, since the random variables are independent, the variance of their sum is the sum of their variances. Hence, using a principle similar to the **linearity of expectation**:

$$\mathbf{var}[q \cdot k] = \mathbf{var}\left[\sum_{i=1}^{d_k} q_i k_i\right]$$

$$= \sum_{i=1}^{d_k} \mathbf{var}[q_i k_i] = d_k$$

To make the dot product have a mean of 0 and standard deviation of 1, it's divided by $\sqrt{d_k}$. However, nowadays, this normalization is often omitted since a normal distribution is not always assumed, especially when layer normalization is not used.

### 1.1.3 Calculation in Detail

**Scaled Dot Product Attention** refers to the process of this calculation. Given that **Query**, **Key**, and **Value** are all $3 \times 1$ matrices:

$$Q = K = V = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

Since $QK^T$ results in a $3 \times 3$ matrix:

$$QK^T = \begin{bmatrix} v_1 \cdot v_1 & v_1 \cdot v_2 & v_1 \cdot v_3 \\ v_2 \cdot v_1 & v_2 \cdot v_2 & v_2 \cdot v_3 \\ v_3 \cdot v_1 & v_3 \cdot v_2 & v_3 \cdot v_3 \end{bmatrix}$$

We then divide $QK^T$ by $\sqrt{d_k}$, obtaining the **attention weight**:

$$\mathbf{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}$$

Given the value matrix, we compute:

$$\mathbf{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \times V = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

The attention mechanism gauges the similarity between a *query* (the word we're focusing on) and a *key* (the word we're comparing against). The resulting similarity scores are then used to weigh the importance of words in the **Value** matrix. See below example code to understand how it goes:

### 1.1.4  PyTorch Sample Code

```python
# !python3 -m pip install torch torchtext
import torch
import torch.nn as nn
import torch.nn.functional as F

class ScaledDotProductAttention(nn.Module):
    def __init__(self, temperature):
        super(ScaledDotProductAttention, self).__init__()
        self.temperature = temperature
        self.softmax = nn.Softmax(dim=2)

    def forward(self, q, k, v, mask=None):
        attn = torch.bmm(q, k.transpose(1, 2))
        attn = attn / self.temperature

        if mask is not None:
            attn = attn.masked_fill(mask, -float('inf'))

        attn = self.softmax(attn)
        output = torch.bmm(attn, v)
        return output, attn
```

### 1.1.5  Multi-head Attention

Now, it is the moment to introduce **Multi-head Attention** - where self-attention is trained for $h$ times in parallel. It enables transformer to pay attention to various parts of input, and thus consider inter-dependency and diversity of input sentences.

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, n_head, d_model, d_k, d_v, dropout=0.1):
        super(MultiHeadAttention, self).__init__()
        self.n_head = n_head
        self.d_k = d_k
        self.d_v = d_v

        self.w_qs = nn.Linear(d_model, d_k * n_head, bias=False)
        self.w_ks = nn.Linear(d_model, d_k * n_head, bias=False)
        self.w_vs = nn.Linear(d_model, d_v * n_head, bias=False)
        self.fc = nn.Linear(n_head * d_v, d_model)
        self.attention = ScaledDotProductAttention(temperature=torch.sqrt(torch.
    tensor(d_k).float()))
```

```python
        self.dropout = nn.Dropout(dropout)
        self.layer_norm = nn.LayerNorm(d_model)

    def forward(self, q, k, v, mask=None):
        d_k, d_v, n_head = self.d_k, self.d_v, self.n_head
        sz_b, len_q = q.size(0), q.size(1)

        # query, key, value
        q = self.w_qs(q).view(sz_b, len_q, n_head, d_k)
        k = self.w_ks(k).view(sz_b, len_q, n_head, d_k)
        v = self.w_vs(v).view(sz_b, len_q, n_head, d_v)

        q = q.permute(2, 0, 1, 3).contiguous().view(-1, len_q, d_k)
        k = k.permute(2, 0, 1, 3).contiguous().view(-1, len_q, d_k)
        v = v.permute(2, 0, 1, 3).contiguous().view(-1, len_q, d_v)

        if mask is not None:
            mask = mask.repeat(n_head, 1, 1)
        output, attn = self.attention(q, k, v, mask=mask)

        output = output.view(n_head, sz_b, len_q, d_v)
        output = output.permute(1, 2, 0, 3).contiguous().view(sz_b, len_q, -1)
        output = self.dropout(self.fc(output))

        # residual connection to prevent accelerated vanish
        output = self.layer_norm(output + q)
        return output, attn
```

### 1.1.6 Transformer Encoder

In Encoding, **Transformer** repeats scaled dot product attention $n$ times, as to extract feature from given data.

```python
[3]: class TransformerEncoderLayer(nn.Module):
    def __init__(self, d_model, d_k, d_v, d_ff, n_head, dropout=0.1):
        super(TransformerEncoderLayer, self).__init__()
        self.self_attn = MultiHeadAttention(n_head, d_model, d_k, d_v,
    →dropout=dropout)
        self.pos_ffn = PositionwiseFeedForward(d_model, d_ff, dropout=dropout)

    def forward(self, src, mask=None):
        enc_output, attn = self.self_attn(src, src, src, mask=mask)
        enc_output = self.pos_ffn(enc_output)
        return enc_output, attn

class TransformerEncoder(nn.Module):
    def __init__(self, d_model, d_k, d_v, d_ff, n_head, n_layers, dropout=0.1):
```

```python
        super(TransformerEncoder, self).__init__()
        self.layers = nn.ModuleList([
            TransformerEncoderLayer(d_model, d_k, d_v, d_ff, n_head,␣
↪dropout=dropout)
            for _ in range(n_layers)
        ])

    def forward(self, src, mask=None):
        attns = []
        for layer in self.layers:
            src, attn = layer(src, mask=mask)
            attns.append(attn)
        return src, attns

class PositionwiseFeedForward(nn.Module):
    def __init__(self, d_in, d_hid, dropout=0.1):
        super(PositionwiseFeedForward, self).__init__()
        self.w_1 = nn.Linear(d_in, d_hid)
        self.w_2 = nn.Linear(d_hid, d_in)
        self.layer_norm = nn.LayerNorm(d_in)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        residual = x
        x = self.w_2(F.relu(self.w_1(x)))
        x = self.dropout(x)
        x = self.layer_norm(x + residual)
        return x
```

### 1.1.7 Understanding Cross Attention in Transformers

Delving deeper into the intricacies of the Transformer model, one inevitably stumbles upon a vital variant of the self-attention mechanism: **Cross Attention**. Unraveling its essence helps demystify how Transformers facilitate understanding between two distinct sequences, often a source and a target, especially paramount in tasks like machine translation, summarization, and question-answering.

**Bridging Sequences: The Core of Cross Attention**   The mechanism of cross attention primarily functions to correlate information between two sequences, which we can term as the "input" sequence and the "context" sequence. In the realm of machine translation, for instance, the input might be a sentence in English, while the context would be its counterpart in French. How, then, does the Transformer ensure a coherent relationship between these disparate sets of information?

When we delineate self-attention, all the queries, keys, and values are derived from the same input sequence. Contrarily, in the cross-attention mechanism, while the queries originate from one sequence (let's say the input sequence), the keys and values are derived from a different sequence (the context).

Mathematically, let's symbolize our input sequence as $X$ and context sequence as $Y$. The associated query, key, and value matrices for these sequences would be $Q_X$, $K_Y$, and $V_Y$ respectively. The attention calculation modifies as:

$$\text{Attention}(Q_X, K_Y, V_Y) = \text{softmax}\left(\frac{Q_X K_Y^T}{\sqrt{d_k}}\right) V_Y$$

**An Illustration: Machine Translation**  Let's elucidate with the model translating English to French. An input sequence in English, "Hello, how are you?", is passed through the encoder, giving rise to a context that the decoder can utilize. The decoder, tasked with producing the French translation, generates queries from its previous outputs (initially seeded, often with a start token). These queries attend to the keys and values from the encoder's output, ensuring the translated words are not just grammatically and contextually apt, but also correspond accurately to the source sentence.

**Dovetailing Encoders and Decoders: A Symbiotic Interplay**  The beauty of the Transformer model, especially in tasks like translation, lies in the symbiotic interplay between the encoder and decoder via cross attention. The encoder processes the input sequence and encapsulates it into a context, rich with syntactic and semantic cues. When the decoder endeavors to generate the target sequence, it employs its own previous outputs as queries in the cross-attention layer, aligning them with the keys and values from the encoder's context, thereby forging a coherent, contextual bridge between source and target languages.

In a nutshell, cross attention capacitates the model to meticulously weave information across sequences, ensuring that the output is not a mere syntactic replica, but a semantically coherent translation, resonating with the nuances of the input. It's through this meticulous attention to inter-sequence relationships that models can generate translations, summaries, or answers that are not just syntactically correct but semantically faithful to the source content.

**Implications and Applications of Cross Attention**  From machine translation to question-answering systems, cross attention empowers the model to consider different sequences simultaneously, providing a dynamic, context-aware lens through which the input sequence is perceived and processed. Particularly in dialogue systems or chatbots, when a user poses a question, cross attention aids the model in correlating this input with the preceding conversation, enabling generation of responses that are contextually anchored and relevant to the ongoing interaction.

In summary, the advent of cross attention has enabled Transformers to elevate their capacity for sequence transduction tasks, adroitly managing the correlation and transfer of semantic and syntactic information between distinct sequences, thereby realizing more contextually robust and coherent outputs in myriad applications across the NLP domain. This mechanism not only reflects the architecture's capacity for handling diverse NLP tasks but also epitomizes its versatility and adaptability in managing complex, inter-sequence dependencies.

### 1.1.8 Sample Code to Run

```
[4]: # # Assuming you have all the previous classes defined as provided
     # import torch
     # from torchtext.vocab import build_vocab_from_iterator
     # from torchtext.data.utils import get_tokenizer

     # # 1. Tokenization and Vocabulary
     # tokenizer = get_tokenizer('basic_english')

     # def yield_tokens(data_iter):
     #     for sentence in data_iter:
     #         yield tokenizer(sentence)

     # sentences = ["Hello World"]
     # vocab = build_vocab_from_iterator(yield_tokens(sentences), specials=['<unk>',
     ↪'<pad>', '<bos>', '<eos>'])
     # vocab.set_default_index(vocab["<unk>"])

     # tokenized_sentence = [vocab[token] for token in tokenizer("Hello World")]
     # sentence_tensor = torch.tensor(tokenized_sentence, dtype=torch.long).
     ↪unsqueeze(0)   # (1, len(sentence))

     # # 2. Embedding
     # embedding_dim = 512
     # embedding = torch.nn.Embedding(len(vocab), embedding_dim)
     # embedded_sentence = embedding(sentence_tensor)

     # # 3. Encoding with TransformerEncoder
     # # Assuming d_model=512, d_k=d_v=64, d_ff=2048, n_head=8, n_layers=6
     # encoder = TransformerEncoder(d_model=512, d_k=64, d_v=64, d_ff=2048,
     ↪n_head=8, n_layers=6)
     # enc_output, attns = encoder(embedded_sentence)

     # print("Encoder Output Shape:", enc_output.shape)
```

```
     ---------------------------------------------------------------------------
     RuntimeError                              Traceback (most recent call last)
     /Users/seunghyeonshin/Deep-Learning-Study/Attention Is All You Need.ipynb  9
     ↪line 2
           <a href='vscode-notebook-cell:/Users/seunghyeonshin/Deep-Learning-Study/
     ↪Attention%20Is%20All%20You%20Need.ipynb#X10sZmlsZQ%3D%3D?line=24'>25</a> # 3.
     ↪Encoding with TransformerEncoder
           <a href='vscode-notebook-cell:/Users/seunghyeonshin/Deep-Learning-Study/
     ↪Attention%20Is%20All%20You%20Need.ipynb#X10sZmlsZQ%3D%3D?line=25'>26</a> #
     ↪Assuming d_model=512, d_k=d_v=64, d_ff=2048, n_head=8, n_layers=6
```

```
        <a href='vscode-notebook-cell:/Users/seunghyeonshin/Deep-Learning-Study/
↪Attention%20Is%20All%20You%20Need.ipynb#X10sZmlsZQ%3D%3D?line=26'>27</a>␣
↪encoder = TransformerEncoder(d_model=512, d_k=64, d_v=64, d_ff=2048, n_head=8␣
↪n_layers=6)
---> <a href='vscode-notebook-cell:/Users/seunghyeonshin/Deep-Learning-Study/
↪Attention%20Is%20All%20You%20Need.ipynb#X10sZmlsZQ%3D%3D?line=27'>28</a>␣
↪enc_output, attns = encoder(embedded_sentence)
        <a href='vscode-notebook-cell:/Users/seunghyeonshin/Deep-Learning-Study/
↪Attention%20Is%20All%20You%20Need.ipynb#X10sZmlsZQ%3D%3D?line=29'>30</a>␣
↪print("Encoder Output Shape:", enc_output.shape)

File /opt/homebrew/lib/python3.11/site-packages/torch/nn/modules/module.py:1501␣
↪in Module._call_impl(self, *args, **kwargs)
   1496 # If we don't have any hooks, we want to skip the rest of the logic in
   1497 # this function, and just call forward.
   1498 if not (self._backward_hooks or self._backward_pre_hooks or self.
↪_forward_hooks or self._forward_pre_hooks
   1499         or _global_backward_pre_hooks or _global_backward_hooks
   1500         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1501     return forward_call(*args, **kwargs)
   1502 # Do not call functions when jit is used
   1503 full_backward_hooks, non_full_backward_hooks = [], []

/Users/seunghyeonshin/Deep-Learning-Study/Attention Is All You Need.ipynb  9␣
↪line 2
        <a href='vscode-notebook-cell:/Users/seunghyeonshin/Deep-Learning-Study/
↪Attention%20Is%20All%20You%20Need.ipynb#X10sZmlsZQ%3D%3D?line=20'>21</a> attns␣
↪= []
        <a href='vscode-notebook-cell:/Users/seunghyeonshin/Deep-Learning-Study/
↪Attention%20Is%20All%20You%20Need.ipynb#X10sZmlsZQ%3D%3D?line=21'>22</a> for␣
↪layer in self.layers:
---> <a href='vscode-notebook-cell:/Users/seunghyeonshin/Deep-Learning-Study/
↪Attention%20Is%20All%20You%20Need.ipynb#X10sZmlsZQ%3D%3D?line=22'>23</a>    ␣
↪src, attn = layer(src, mask=mask)
        <a href='vscode-notebook-cell:/Users/seunghyeonshin/Deep-Learning-Study/
↪Attention%20Is%20All%20You%20Need.ipynb#X10sZmlsZQ%3D%3D?line=23'>24</a>    ␣
↪attns.append(attn)
        <a href='vscode-notebook-cell:/Users/seunghyeonshin/Deep-Learning-Study/
↪Attention%20Is%20All%20You%20Need.ipynb#X10sZmlsZQ%3D%3D?line=24'>25</a>␣
↪return src, attns

File /opt/homebrew/lib/python3.11/site-packages/torch/nn/modules/module.py:1501␣
↪in Module._call_impl(self, *args, **kwargs)
   1496 # If we don't have any hooks, we want to skip the rest of the logic in
   1497 # this function, and just call forward.
   1498 if not (self._backward_hooks or self._backward_pre_hooks or self.
↪_forward_hooks or self._forward_pre_hooks
   1499         or _global_backward_pre_hooks or _global_backward_hooks
   1500         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1501     return forward_call(*args, **kwargs)
```

```
1502 # Do not call functions when jit is used
1503 full_backward_hooks, non_full_backward_hooks = [], []
```

/Users/seunghyeonshin/Deep-Learning-Study/Attention Is All You Need.ipynb 9
  ↪line 8
```
      <a href='vscode-notebook-cell:/Users/seunghyeonshin/Deep-Learning-Study/
↪Attention%20Is%20All%20You%20Need.ipynb#X10sZmlsZQ%3D%3D?line=6'>7</a> def
↪forward(self, src, mask=None):
----> <a href='vscode-notebook-cell:/Users/seunghyeonshin/Deep-Learning-Study/
↪Attention%20Is%20All%20You%20Need.ipynb#X10sZmlsZQ%3D%3D?line=7'>8</a>
↪enc_output, attn = self.self_attn(src, src, src, mask=mask)
      <a href='vscode-notebook-cell:/Users/seunghyeonshin/Deep-Learning-Study/
↪Attention%20Is%20All%20You%20Need.ipynb#X10sZmlsZQ%3D%3D?line=8'>9</a>
↪enc_output = self.pos_ffn(enc_output)
      <a href='vscode-notebook-cell:/Users/seunghyeonshin/Deep-Learning-Study/
↪Attention%20Is%20All%20You%20Need.ipynb#X10sZmlsZQ%3D%3D?line=9'>10</a>
↪return enc_output, attn
```

File /opt/homebrew/lib/python3.11/site-packages/torch/nn/modules/module.py:1501
  ↪in Module._call_impl(self, *args, **kwargs)
```
1496 # If we don't have any hooks, we want to skip the rest of the logic in
1497 # this function, and just call forward.
1498 if not (self._backward_hooks or self._backward_pre_hooks or self.
↪_forward_hooks or self._forward_pre_hooks
1499         or _global_backward_pre_hooks or _global_backward_hooks
1500         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1501     return forward_call(*args, **kwargs)
1502 # Do not call functions when jit is used
1503 full_backward_hooks, non_full_backward_hooks = [], []
```

/Users/seunghyeonshin/Deep-Learning-Study/Attention Is All You Need.ipynb 9
  ↪line 3
```
      <a href='vscode-notebook-cell:/Users/seunghyeonshin/Deep-Learning-Study/
↪Attention%20Is%20All%20You%20Need.ipynb#X10sZmlsZQ%3D%3D?line=35'>36</a>
↪output = self.dropout(self.fc(output))
      <a href='vscode-notebook-cell:/Users/seunghyeonshin/Deep-Learning-Study/
↪Attention%20Is%20All%20You%20Need.ipynb#X10sZmlsZQ%3D%3D?line=37'>38</a> #
↪residual connection to prevent accelerated vanish
---> <a href='vscode-notebook-cell:/Users/seunghyeonshin/Deep-Learning-Study/
↪Attention%20Is%20All%20You%20Need.ipynb#X10sZmlsZQ%3D%3D?line=38'>39</a>
↪output = self.layer_norm(output + q)
      <a href='vscode-notebook-cell:/Users/seunghyeonshin/Deep-Learning-Study/
↪Attention%20Is%20All%20You%20Need.ipynb#X10sZmlsZQ%3D%3D?line=39'>40</a>
↪return output, attn
```

RuntimeError: The size of tensor a (512) must match the size of tensor b (64) at
  ↪non-singleton dimension 2

```
[ ]: # embedded_sentence.shape, enc_output.shape
```

### 1.1.9 Into PDF

```python
from base_lib import convert_ipynb_to_pdf

convert_ipynb_to_pdf('./Attention Is All You Need.ipynb')
```