Attention for your Transformers

Attention on Attention

Vipul Koti
16 min read · 2 hours ago

Analogy and Definition-

Let's start with an Analogy of the Attention Mechanism to Human Reading Using "The Lord of the Rings"

[Source- Generated using DALL-E]

Imagine you're reading "The Lord of the Rings," a complex story with a rich narrative and many characters. As you read, your brain naturally prioritizes certain details over others, depending on the context, you don't give equal attention to every word or sentence. Instead, you:

1. **Focus on Key Characters and Events**: For example, when reading about Frodo's journey to destroy the One Ring, you might pay close attention to his interactions with Gollum because you know Gollum's behavior is crucial for Frodo's quest.
2. **Recall Important Details**: If Frodo mentions something about Rivendell, you might recall the significance of Rivendell from earlier chapters and use that context to understand the current situation better.
3. **Adjust Focus Based on Context**: When the narrative shifts to a different storyline, like Aragorn's path to becoming king, you switch focus and remember the details relevant to Aragorn, such as his lineage and destiny.

When we read a sentence in a book, we use the context built and saved from previous sentences in the short-term or working memory. The context helps us understand the meaning of the current sentence. This selective focus is similar to how the **attention** mechanism works in deep learning models. The attention mechanism is analogous to short-term memory for transformers.

Definition- An attention mechanism in machine learning allows models to dynamically focus on the most relevant parts of the input data while processing the data, thereby enhancing the model's efficiency and accuracy in processing the data.

What we will learn next-

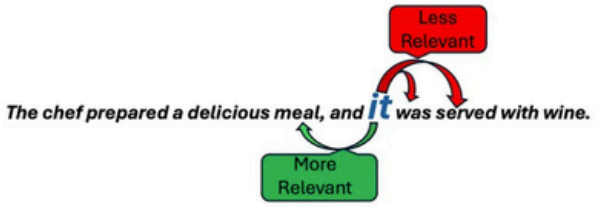
1. **Level 1 & Level 2** — We will first understand Self-Attention (with and without weights)
2. **Level 3**— On our understanding of Self-Attention, we will understand Causal attention
3. **Level 4** — Last we will discuss Multi-head attention
4. **Bonus**- We will discuss Cross Attention

Self-Attention-

Self-attention is a mechanism that calculates attention weights by analyzing the relationships between different parts of a single input sequence, such as words in a sentence or pixels in an image. It learns **how these elements relate to each other within the same input**.

Traditional Sequential models, like RNN process data sequentially and depend on the context built so far, these models have trouble maintaining context over long sequences (exploding or vanishing gradient problem, RNN training doesn't allow parallel processing). Another important thing to understand is that the importance of previous words generally doesn't depend on the order in which they appear.

Let's understand this and the need for self-attention from an **example**-



[src-Image prepared by author]

Consider the sentence: "*The chef prepared a delicious meal, and it was served with wine.*" This sentence has 12 words, or tokens. If we focus on the word "it," we need to understand what "it" refers to in the context of the sentence. The words "was" and "served" are close to "it" in terms of proximity, but they don't help us understand the meaning of "it." Instead, the word "meal," which appears earlier in the sentence, is much more relevant because "it" refers to the "meal."

In this example, *proximity isn't the key factor for understanding; the context provided by the word "meal" is what clarifies the meaning of "it."*

When a computer processes this sentence, each word is represented as a token with a word embedding, which is a numerical vector representing the word's meaning. In a word embedding space, words with similar meanings or that are used in similar contexts have embeddings that are close to each other. For instance, the word "chef" might be close to "cook" and "kitchen," while "wine" might be close to "beverage" and "drink." To learn more about word embedding. **link**

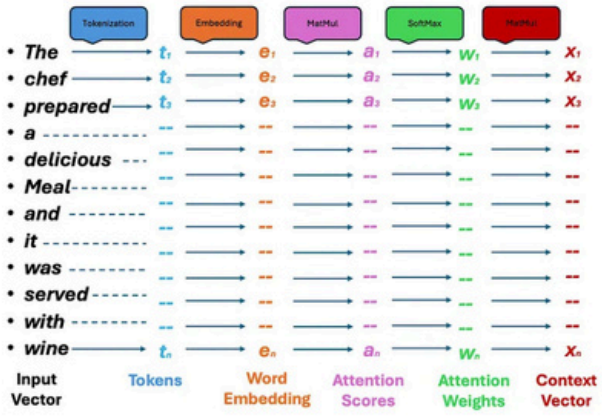
Initially, these embeddings don't capture the relationships between words like "it" and "wine" in our sentence; they lack context for our sentences. The goal of self-attention is to refine these general embeddings with current context so that they include more context about the current sentence as a whole.

Using self-attention, we calculate **how much each word in the sentence should "attend to" or focus on every other word to understand its context better**. For the word "it" in our example, the attention mechanism would give more weight to "meal" than to "was" or "served" because "meal" provides the necessary context to understand what "it" refers to. Generally, we can say, that by using self-attention, we effectively enhance the word

embeddings so that they carry more contextual information, enabling the model to understand the sentence more accurately.

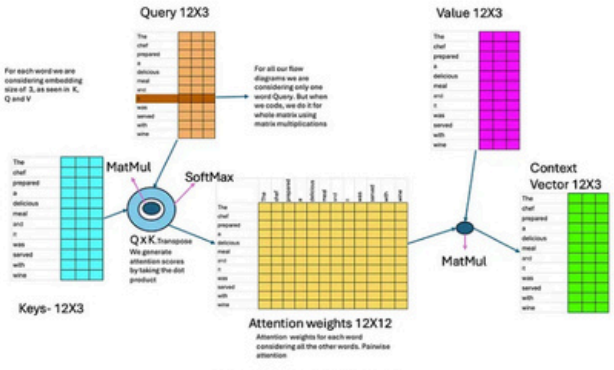
Level 1 — We will understand the basic self-attention calculations without weights (we will add weights in the next section)

Think of the above sentence "***The chef prepared a delicious meal, and it was served with wine.***" First, we calculate vocabulary and turn words into tokens {t1, t2, t3... tn}, these tokens are then converted into word embedding {e1,e2,e3,...en}. (Covered here in word Embedding discussion).
Now these word embeddings are used pairwise (e2 with all e1,e2,e3,...en) to get relevant attention score {a1,a2,a3,...an}(how important all the other words are for word e2, if we do for the word "it", the word "meal" will have the highest attention score). These scores here are calculated using the dot product of word embeddings(we are not considering weights in this section). Then the attention scores are converted to attention weights {w1,w2,w3,... wn} by normalizing the attention scores. In the last step, using these weights, a new representation for each word is generated as the weighted sum of all the other words. (Note weight for each word is defined by the attention/relevance scores)



[src-Image prepared by author]

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$



[src-Image prepared by author]

Summarizing steps for self-attention (without weight matrices)

1. Represent each word as a vector (word to token {t1, t2, t3... tn}, and token to embedding {e1,e2,e3,...en})
2. Pairwise compare each word with every other word to compute attention score {a1,a2,a3,...an}. e.g. Calculate how much focus should be placed on "meal" when understanding "it" (one pair).
3. Normalize the attention score to get attention weights {w1,w2,w3,...wn}. Convert scores to probabilities (weights) that sum to 1. We use softmax

usually.

4. Create a new context-aware vector representation {x1, x2, x3... xn} for each word as a weighted sum of all words, using the attention weights from step 3. e.g. Create a new representation of "it" that incorporates information about "chef", "prepared", "meal", etc.

```
### Self-Attention (Without Weights) ###
import torch
#Step 1
word_embeddings = torch.tensor(
  [[0.32, 0.68, 0.45], # The      (x^1)
   [0.71, 0.23, 0.89], # chef     (x^2)
   [0.55, 0.92, 0.37], # prepared (x^3)
   [0.18, 0.79, 0.60], # a        (x^4)
   [0.84, 0.41, 0.13], # delicious(x^5)
   [0.29, 0.63, 0.76], # meal     (x^6)
   [0.50, 0.15, 0.95], # and      (x^7)
   [0.67, 0.38, 0.82], # it       (x^8)
   [0.43, 0.91, 0.26], # was      (x^9)
   [0.75, 0.20, 0.58], # served   (x^10)
   [0.36, 0.72, 0.49], # with     (x^11)
   [0.88, 0.54, 0.11]] # wine     (x^12)
)

# Step 2 Calculate attention scores (Pairwise compare)
attn_scores = word_embeddings @ word_embeddings.T

# Step 3 Calculate attention weights (Normalize)
attn_weights = torch.softmax(attn_scores, dim=-1)

# Step 4 Calculate context vectors
all_context_vecs = attn_weights @ word_embeddings
```

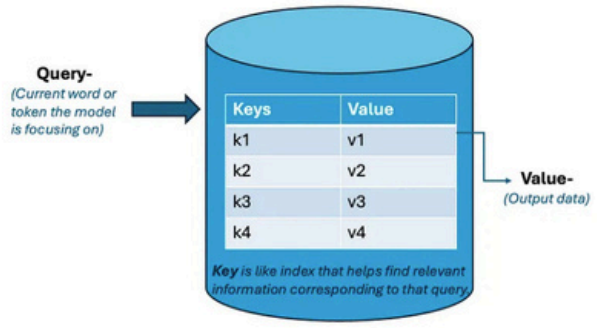
```
print(all_context_vecs)
```

```
tensor([[0.5270, 0.5664, 0.5374],
        [0.5533, 0.5059, 0.5825],
        [0.5316, 0.5783, 0.5197],
        [0.5150, 0.5726, 0.5456],
        [0.5655, 0.5434, 0.5146],
        [0.5233, 0.5521, 0.5616],
        [0.5458, 0.5044, 0.5926],
        [0.5477, 0.5204, 0.5716],
        [0.5280, 0.5851, 0.5142],
        [0.5601, 0.5151, 0.5592],
        [0.5271, 0.5664, 0.5382],
        [0.5638, 0.5516, 0.5077]])
```

Level 2- Introducing trainable weights in Self-Attention

Trainable Weights: In self-attention, three sets of weights (represented as matrices) are used to help the model focus on different aspects of the input:

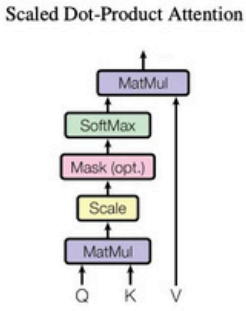
- **Query (Q) Matrix**: This matrix helps determine what specific information or context the model is searching for when analyzing each word. (Helps to determine the question we ask about each word. Q- *What do you want?*)
- **Key (K) Matrix**: This matrix helps identify which information within each word is relevant to the query, similar to an indexing system. (Helps in identifying key information in each word. Q- *Who do you get it from?*)
- **Value (V) Matrix**: This matrix holds the actual content or meaning of the words that the model will use once it has determined which parts of the input are important.(Q- *What do you get?*)



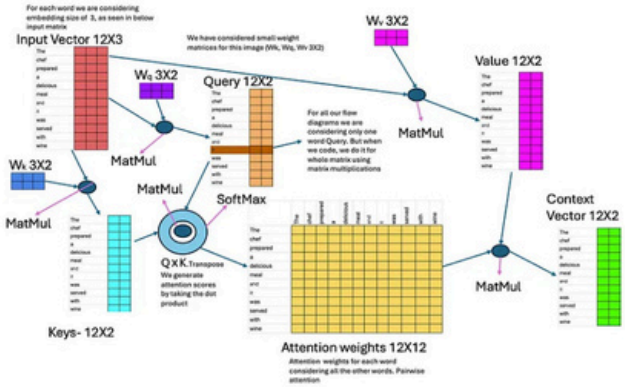
[src-Image prepared by author]

To explain this further, think of a "query" like a search term in a database that represents the current word or token the model is focusing on. The "key" is like an index that helps find relevant information corresponding to

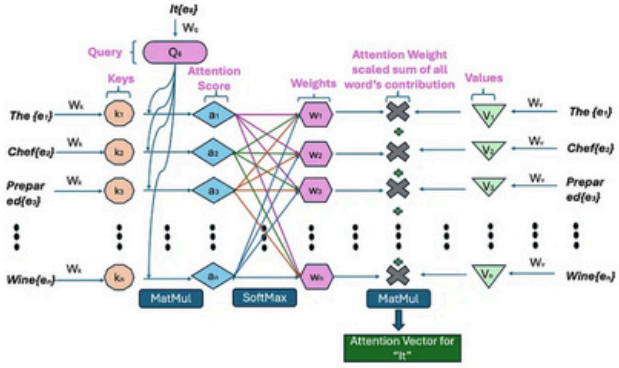
that query. The “value” is similar to the actual data retrieved from the database once a match is found. By using these three components together, the model can decide how much attention to give to different parts of the input, allowing it to better understand and process information.



[Source- Attention is all you need. NeurIPS, 2017]



[src-Image prepared by author]



[src-Image prepared by author]

Summarizing steps for self-attention (with weight matrices)

1. Generate Query, Key, Value vectors- Matrix multiplication of Vector embedding for input with Q,k, V weight matrices to get Query, Key, Value vectors
2. Compute Attention scores- Matrix multiply Query and Key matrices to get raw attention score matrix containing scores for all input words.
3. Normalize scores- Appy softmax to the raw attention scores to get the normalized attention weights.
4. Generate weight-scaled context vector- Matrix multiple attention weight vector with the values matrix, to generate context vector matrix, which contains context vector for each word (matrix multiply generate normalized weighted sum context vector for each word considering all the words in the input)

```
### Self-Attention (with weights) ###
```

```
import torch
import torch.nn as nn

# Set random seed for reproducibility
torch.manual_seed(1240)

# Define input and output dimensions
d_in = word_embeddings.shape[1]  # Input embedding size (3 in this case)
d_out = 2  # Output embedding size

class SelfAttention(nn.Module):
    def __init__(self, d_in, d_out, use_bias=False):
        super().__init__()
        # Linear transformations for Query, Key, and Value
        self.W_query = nn.Linear(d_in, d_out, bias=use_bias)
        self.W_key = nn.Linear(d_in, d_out, bias=use_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=use_bias)

    def forward(self, x):
        # Step 1 Transform input to Query, Key, and Value
        queries = self.W_query(x)
        keys = self.W_key(x)
        values = self.W_value(x)

        # Step 2 Calculate attention scores
        attn_scores = queries @ keys.T

        # Step 3 Apply scaling factor and softmax to get attention weights
        scaling_factor = keys.shape[-1] ** 0.5
        attn_weights = torch.softmax(attn_scores / scaling_factor, dim=-1)

        # Step 4 Compute context vectors
        context_vectors = attn_weights @ values
        return context_vectors

# Initialize the SelfAttention module
torch.manual_seed(1240)  # Reset seed for consistent results
self_attention = SelfAttention(d_in, d_out)

# Apply self-attention to word embeddings
result = self_attention(word_embeddings)
print(result)
```

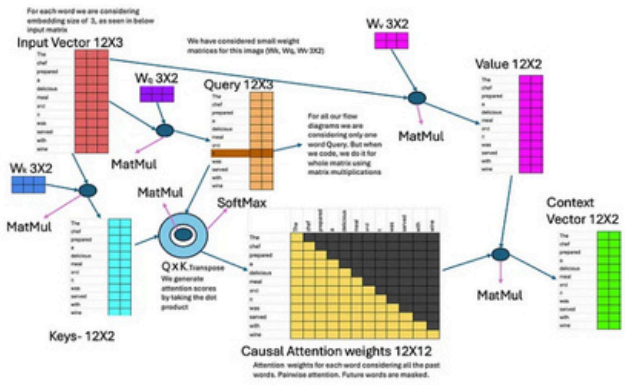
```
tensor([[0.1348, 0.1801],
        [0.1358, 0.1782],
        [0.1361, 0.1776],
        [0.1346, 0.1803],
        [0.1358, 0.1782],
        [0.1349, 0.1798],
        [0.1348, 0.1799],
        [0.1359, 0.1780],
        [0.1355, 0.1788],
        [0.1355, 0.1787],
        [0.1351, 0.1796],
        [0.1362, 0.1774]], grad_fn=<MmBackward0>)
```

Level 3- We will add Causal mask to the self-attention we discussed in the earlier section.

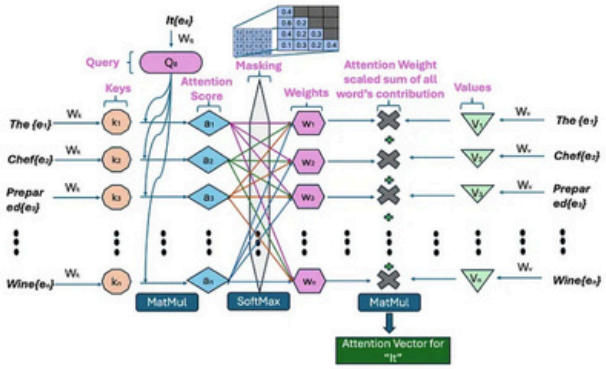
Causal Attention-

In some problems, e.g. during the model's testing on the next word prediction task, we need the model to be causal. Model to be causal means, the model can't look at the future when we are trying to predict the output at the current time stamp. e.g. During test time, the Auto Complete model will only see the past words to predict the next word, it will not have access to the future words. (In case it has access, it is cheating).

Causal Attention is also known as masked attention as it is implemented by extending the self-attention mechanism we discussed in the last section by adding a causal mask. A causal mask hides the future words. This mask is applied to set the attention scores of these future words to zero before the model uses the attention scores.



[src-Image prepared by author]



[src-Image prepared by author]

Summarizing Steps for Causal Attention

1. Compute Attention scores- Similar to the self-attention mechanism, calculate the importance scores for all word positions relative to each other.
2. Appy Causal Mask- Apply causal mask to mask out the score for future words by setting them to negative infinity (-∞) before the normalization. To summarize, we mask out the attention scores above the diagonal for a given input matrix.
3. Normalize and Apply Attention- After normalization, the future words will have 0 attention weight as their attention score was negative infinity. (Softmax function converts its inputs into a probability distribution. When negative infinity values (-∞) are present in a row, the softmax function treats them as zero probability. (Because $e^{-\infty}$ approaches 0). New attention weights are then applied to the model task.

```
### Caual Attention with causal and dropout mask on batched input###

import torch
import torch.nn as nn

# Create a batch by stacking the word embeddings twice
batch = torch.stack((word_embeddings, word_embeddings), dim=0)
print(batch.shape)  # Shape: (2, 12, 3) - 2 inputs, 12 tokens each, 3-dimensiona

class CausalAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length, dropout, qkv_bias=False):
        super().__init__()
        self.d_out = d_out
        # Linear transformations for Query, Key, and Value
        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)
        # Dropout layer for regularization
        self.dropout = nn.Dropout(dropout)
        # Create an upper triangular mask to ensure causality
        self.register_buffer('mask', torch.triu(torch.ones(context_length, conte

    def forward(self, x):
        b, num_tokens, d_in = x.shape  # b: batch size, num_tokens: sequence len

        # Transform input to Query, Key, and Value
        queries = self.W_query(x)
        keys = self.W_key(x)
        values = self.W_value(x)

        # Calculate attention scores
        attn_scores = queries @ keys.transpose(1, 2)

        # Apply causal mask to prevent attending to future tokens
        attn_scores.masked_fill_(
            self.mask.bool()[:num_tokens, :num_tokens],
            -torch.inf
        )

        # Apply scaling factor and softmax to get attention weights
        scaling_factor = keys.shape[-1] ** 0.5
        attn_weights = torch.softmax(attn_scores / scaling_factor, dim=-1)

        # Apply dropout to attention weights
        attn_weights = self.dropout(attn_weights)

        # Compute context vectors
        context_vectors = attn_weights @ values
        return context_vectors

# Set random seed for reproducibility
torch.manual_seed(1240)

# Initialize CausalAttention module
```

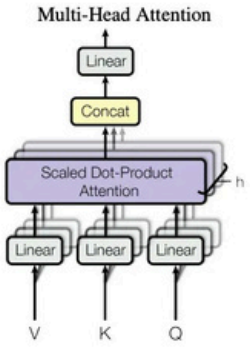
```
context_length = batch.shape[1]
d_in = batch.shape[2]
d_out = 2  # Output embedding size
causal_attention = CausalAttention(d_in, d_out, context_length, dropout=0.0)

# Apply causal attention to the batch
context_vectors = causal_attention(batch)

print(context_vectors)
print("context_vectors.shape:", context_vectors.shape)
```

```
torch.Size([2, 12, 3])
tensor([[[0.0872, 0.2233],
         [0.1996, 0.0526],
         [0.1382, 0.1952],
         [0.1382, 0.1830],
         [0.1097, 0.2292],
         [0.1286, 0.1924],
         [0.1608, 0.1259],
         [0.1748, 0.1068],
         [0.1528, 0.1494],
         [0.1562, 0.1402],
         [0.1502, 0.1500],
         [0.1362, 0.1774]],

        [[0.0872, 0.2233],
         [0.1996, 0.0526],
         [0.1382, 0.1952],
         [0.1382, 0.1830],
         [0.1097, 0.2292],
         [0.1286, 0.1924],
         [0.1608, 0.1259],
         [0.1748, 0.1068],
         [0.1528, 0.1494],
         [0.1562, 0.1402],
         [0.1502, 0.1500],
         [0.1362, 0.1774]]], grad_fn=<UnsafeViewBackward0>)
context_vectors.shape: torch.Size([2, 12, 2])
```

Level 4- We will discuss multi-head attention, which we will build on Causal attention we discussed in earlier section.

Multi-Headed Attention-

The core concept of multi-head attention is to apply the attention mechanism several times in parallel, each with its own set of learned linear transformations. This approach enables **the model to simultaneously focus on information from different representation subspaces and at various positions within the input**, capturing a richer set of dependencies and patterns.



[Source- Attention is all you need. NeurIPS, 2017]

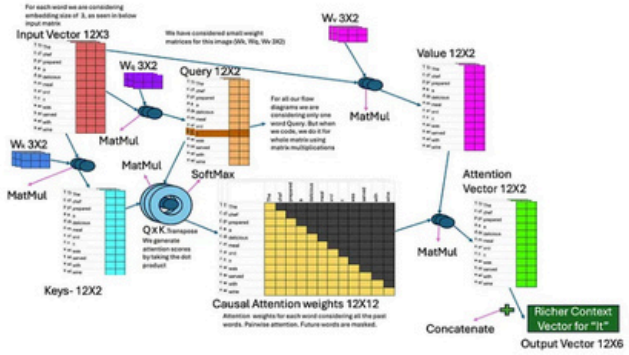
$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_\text{i}^Q, KW_\text{i}^K, VW_\text{i}^V)$$

Let's go back to our old example, "**The chef prepared a delicious meal, and it was served with wine.**" If we focus on the word "meal," we can see that there are several other words in the sentence that have significant relevance to it. Specifically, the words "chef," "prepared," "delicious," and "wine" all provide important context that enhances our understanding of the word "meal."

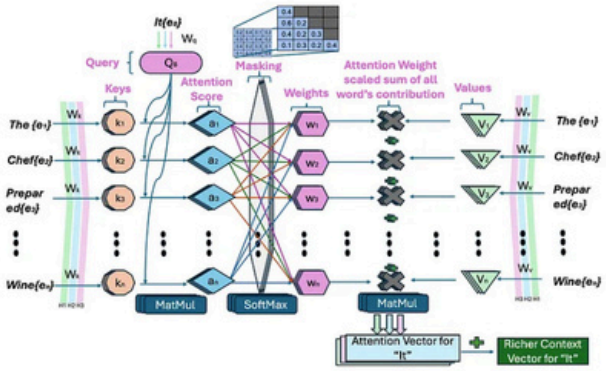
- The word "chef" tells us who is responsible for making the meal.
- The word "prepared" indicates the action taken to create the meal.
- The word "delicious" describes the quality of the meal.
- The phrase "with wine" suggests what accompanies the meal when served.

In a simple attention mechanism, a single attention head might struggle to simultaneously capture all these nuances and associations. This could lead to missing some relevant words or not fully capturing the relationships between “meal” and other words.

However, with **multi-head attention**, we can allocate different attention heads to focus on different aspects of the sentence. For instance, one head might focus on the relationship between “meal” and “chef,” another on “meal” and “delicious,” and yet another on “meal” and “wine.” By distributing the attention across multiple heads, the model can capture a richer and more nuanced understanding of the word “meal” in context. This richer understanding of all relevant words and relationships enhances the model’s ability.



[src-Image prepared by author]



[src-Image prepared by author]

Summarizing steps for multi-head attention

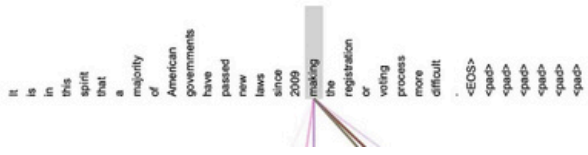
1. Learn Different Relationships- Run multiple heads of attention mechanism on the input. Each head will use different weight matrices and generate different attention weights.
2. Combine attention scores- All the attention scores from the multiple heads are concatenated and then combined using linear layer projection
3. Generate context vector- The combined attention score from step 2 is used to generate the final enriched representation of the input data.

```
### Multi-Head Attention ###
import torch
import torch.nn as nn


class MultiHeadAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length, dropout, num_heads, qkv_bias
        super().__init__()
        assert d_out % num_heads == 0, "d_out must be divisible by num_heads"

        self.d_out = d_out
        self.num_heads = num_heads
        self.head_dim = d_out // num_heads  # Dimension of each attention head

        # Linear projections for Query, Key, and Value
        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)
```

```
        # Output projection
        self.out_proj = nn.Linear(d_out, d_out)

        # Dropout for regularization
        self.dropout = nn.Dropout(dropout)

        # Causal mask to prevent attending to future tokens
        self.register_buffer(
            "mask",
            torch.triu(torch.ones(context_length, context_length), diagonal=1)
        )

    def forward(self, x):
        b, num_tokens, d_in = x.shape  # b: batch size, num_tokens: sequence len

        # Linear projections
        keys = self.W_key(x)      # Shape: (b, num_tokens, d_out)
        queries = self.W_query(x) # Shape: (b, num_tokens, d_out)
        values = self.W_value(x)  # Shape: (b, num_tokens, d_out)

        # Reshape for multi-head attention
        # (b, num_tokens, d_out) -> (b, num_tokens, num_heads, head_dim)
        keys = keys.view(b, num_tokens, self.num_heads, self.head_dim)
        queries = queries.view(b, num_tokens, self.num_heads, self.head_dim)
        values = values.view(b, num_tokens, self.num_heads, self.head_dim)

        # Transpose for attention computation
        # (b, num_tokens, num_heads, head_dim) -> (b, num_heads, num_tokens, hea
        keys = keys.transpose(1, 2)
        queries = queries.transpose(1, 2)
        values = values.transpose(1, 2)

        # Compute scaled dot-product attention
        attn_scores = queries @ keys.transpose(2, 3)  # Shape: (b, num_heads, nu

        # Apply causal mask
        mask_bool = self.mask.bool()[:num_tokens, :num_tokens]
        attn_scores.masked_fill_(mask_bool, -torch.inf)

        # Compute attention weights
        attn_weights = torch.softmax(attn_scores / (self.head_dim ** 0.5), dim=-
        attn_weights = self.dropout(attn_weights)

        # Apply attention weights to values
        context_vec = attn_weights @ values  # Shape: (b, num_heads, num_tokens,

        # Reshape and combine heads
        context_vec = context_vec.transpose(1, 2).contiguous()  # (b, num_tokens
        context_vec = context_vec.view(b, num_tokens, self.d_out)  # (b, num_tok

        # Final output projection
        context_vec = self.out_proj(context_vec)

        return context_vec

# Set random seed for reproducibility
torch.manual_seed(1240)

# Get dimensions from the batch
batch_size, context_length, d_in = batch.shape
d_out = 2  # Output dimension

# Initialize MultiHeadAttention
mha = MultiHeadAttention(d_in, d_out, context_length, dropout=0.0, num_heads=2)

# Apply multi-head attention to the batch
context_vecs = mha(batch)

print(context_vecs)
print("context_vecs.shape:", context_vecs.shape)
```

```
tensor([[[ 0.6238, -0.3816],
         [ 0.4784, -0.3510],
         [ 0.5800, -0.3691],
         [ 0.5747, -0.3686],
         [ 0.6151, -0.3764],
         [ 0.5870, -0.3709],
         [ 0.5370, -0.3618],
         [ 0.5201, -0.3580],
         [ 0.5527, -0.3642],
         [ 0.5459, -0.3633],
         [ 0.5536, -0.3648],
         [ 0.5761, -0.3689]],

        [[ 0.6238, -0.3816],
         [ 0.4784, -0.3510],
         [ 0.5800, -0.3691],
         [ 0.5747, -0.3686],
         [ 0.6151, -0.3764],
         [ 0.5870, -0.3709],
         [ 0.5370, -0.3618],
         [ 0.5201, -0.3580],
         [ 0.5527, -0.3642],
         [ 0.5459, -0.3633],
         [ 0.5536, -0.3648],
         [ 0.5761, -0.3689]]], grad_fn=<ViewBackward0>)
context_vecs.shape: torch.Size([2, 12, 2])
```

Attention Visualizations



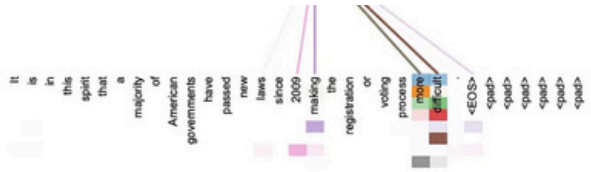


Figure 3: An example of the attention mechanism following long-distance dependencies in the encoder self-attention in layer 5 of 6. Many of the attention heads attend to a distant dependency of the verb 'making', completing the phrase 'making...more difficult'. Attentions here shown only for the word 'making'. Different colors represent different heads. Best viewed in color.

[Source- Attention is all you need. NeurIPS, 2017]

Bonus- Cross Attention

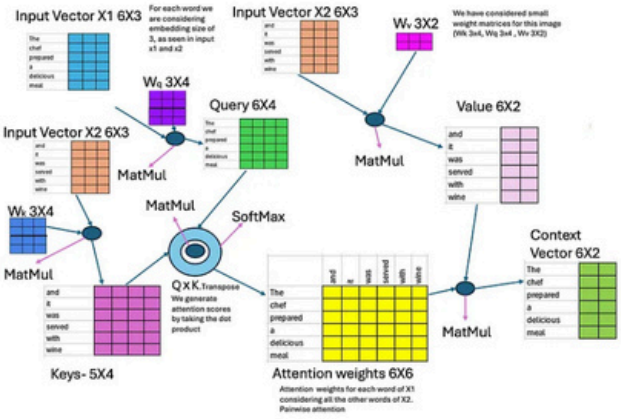
Cross-attention allows a model to focus on relevant parts of one sequence while processing another. When we compare to self-attention, where a model weighs the importance of different parts of the same input sequence (i.e., token to token in a sentence), cross-attention relates to different inputs. The two sequences could belong to different modalities or represent different sets of information.

For code, we will break our original sentence into two sequences and compute cross-attention between those two sequences.

We split the sentence into two parts:

- x_1: "The chef prepared a delicious meal" (first 6 words)
- x_2: "and it was served with wine" (last 6 words)

The cross-attention mechanism will allow the first part of the sentence to attend to the second part. This means each word in the first part will compute attention weights for each word in the second part.



[src-Image prepared by author]

```
### Cross Attention ###
import torch
import torch.nn as nn

word_embeddings = torch.tensor(
  [[0.32, 0.68, 0.45], # The      (x^1)
   [0.71, 0.23, 0.89], # chef     (x^2)
   [0.55, 0.92, 0.37], # prepared (x^3)
   [0.18, 0.79, 0.60], # a        (x^4)
   [0.84, 0.41, 0.13], # delicious(x^5)
   [0.29, 0.63, 0.76], # meal     (x^6)
   [0.50, 0.15, 0.95], # and      (x^7)
   [0.67, 0.38, 0.82], # it       (x^8)
   [0.43, 0.91, 0.26], # was      (x^9)
   [0.75, 0.20, 0.58], # served   (x^10)
   [0.36, 0.72, 0.49], # with     (x^11)
   [0.88, 0.54, 0.11]] # wine     (x^12)
)

class CrossAttention(nn.Module):
    def __init__(self, d_in, d_out_kq, d_out_v):
        super().__init__()
        self.d_out_kq = d_out_kq

        # Learnable weight matrices for Query, Key, and Value
        self.W_query = nn.Parameter(torch.rand(d_in, d_out_kq))
        self.W_key = nn.Parameter(torch.rand(d_in, d_out_kq))
        self.W_value = nn.Parameter(torch.rand(d_in, d_out_v))

    def forward(self, x_1, x_2):
```

```
        # Compute queries from x_1
        queries = x_1 @ self.W_query

        # Compute keys and values from x_2
        keys = x_2 @ self.W_key
        values = x_2 @ self.W_value

        # Compute attention scores
        attn_scores = queries @ keys.T

        # Apply scaling factor and softmax to get attention weights
        scaling_factor = self.d_out_kq ** 0.5
        attn_weights = torch.softmax(attn_scores / scaling_factor, dim=-1)

        # Compute context vectors
        context_vectors = attn_weights @ values

        return context_vectors

# Set random seed for reproducibility
torch.manual_seed(42)

# Define dimensions
d_in = word_embeddings.shape[1]  # Input dimension (3 in this case)
d_out_kq = 4  # Output dimension for queries and keys
d_out_v = 2   # Output dimension for values

# Initialize CrossAttention module
cross_attention = CrossAttention(d_in, d_out_kq, d_out_v)

# Split the sentence into two parts for demonstration
x_1 = word_embeddings[:6]  # "The chef prepared a delicious meal"
x_2 = word_embeddings[6:]  # "and it was served with wine"

# Apply cross-attention
result = cross_attention(x_1, x_2)

print("Cross-attention result shape:", result.shape)
print("Cross-attention result:\n", result)

# Interpret the results
for i, word in enumerate(["The", "chef", "prepared", "a", "delicious", "meal"]):
    print(f"{word}: {result[i].tolist()}")
```

```
Cross-attention result shape: torch.Size([6, 2])
Cross-attention result:
 tensor([[0.5326, 0.2634],
        [0.5321, 0.2654],
        [0.5345, 0.2637],
        [0.5325, 0.2636],
        [0.5334, 0.2634],
        [0.5322, 0.2642]], grad_fn=<MmBackward0>)
The: [0.5325765609741211, 0.26338499784469604]
chef: [0.5320825576782227, 0.2653651535511017]
prepared: [0.534509539604187, 0.2637292146682739]
a: [0.5324926376342773, 0.2635837197303772]
delicious: [0.533425509929657, 0.2634294927120209]
meal: [0.532193660736084, 0.26424530148506165]
```

Note- The weights are randomly initialized and not trained, so the output doesn't have any meaningful interpretation.

The example above, demonstrates how cross-attention allows one part of a sentence to gather information from another part. In more complex models, this mechanism can be used to create dependencies between different parts of the input or between encoder and decoder in sequence-to-sequence models or transformers. Another prominent example of cross-attention usage is in the Stable Diffusion model.

Thanks for Reading!

Thank you for reading! By now, you should have a foundational understanding of different types of attention mechanisms and their roles in improving model performance. This knowledge will help you understand the transformers model much better.

Resource Suggestion

1. https://youtu.be/eMlx5fFNoYc?si=PERGzhjd9nHizYMS

Code Repo

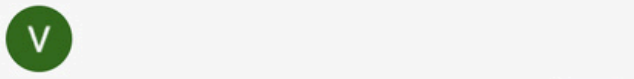
https://github.com/vipulkoti/AttentionOnAttention

References

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. NeurIPS, 2017.

[2] Raschka, S. (2023). *Build a Large Language Model (From Scratch)* [Https://www.manning.com/books/build-a-large-language-model-from-scratch]. Manning Publications Co. https://www.manning.com/books/build-a-large-language-model-from-scratch

Attention Mechanism | Deep Learning | Transformers | AI | Naturallanguageprocessing

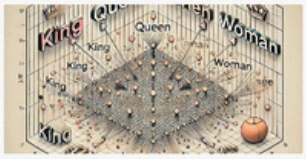


Written by Vipul Koti

5 Followers

Edit profile

More from Vipul Koti



Vipul Koti

Word Embeddings (NLP View)

Word embeddings are a technique for extracting features from text, enabling us to...

Aug 25 9

Vipul Koti

Linguistic representations of words, phrases, clauses, and...

Every language has a structure and languages are built on these structures....

Aug 17

Vipul Koti

Exploring Lexical Semantics: Understanding Word Meaning [in...

The term "lexical semantics." Lexical refers to the lexicon, which is the list of words in a...

Aug 18

See all from Vipul Koti