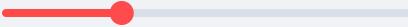
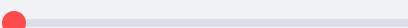


### Settings

Semantic search results  
 5

Prerequisite depth  
 2

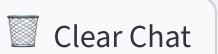
Max concepts  
 10

---

### Knowledge Graph Stats

Concepts in ChromaDB  
3120

---

 Clear Chat

# Erica - Your AI Tutor

Ask questions about AI/ML concepts from the Introduction to AI course



What is attention in transformers and can you provide a python example of how it is used ?



## Understanding Attention in Transformers

### 1. Intuitive Overview: Why Attention Matters

**Attention** is a mechanism in deep learning that helps models focus on specific parts of the input data when making predictions. In the context of transformers, attention allows the model to weigh the importance of different tokens in a sequence, enabling it to handle long-range dependencies more effectively than traditional recurrent neural networks (RNNs).

Imagine you are reading a long document to answer a specific question. Instead of reading every word equally, you might focus more on certain sentences or paragraphs that are more relevant to the question. Attention in transformers works similarly, allowing the model to "focus" on the most relevant parts of the input sequence.

## 2. Prerequisite Concepts

Before diving into the details of attention in transformers, it's important to understand a few foundational concepts:

- **Sequence-to-Sequence Models:** These models are used to map an input sequence to an output sequence, such as translating a sentence from one language to another. They typically consist of an encoder and a decoder.
- **Self-Attention:** A type of attention mechanism where the model attends to different parts of the same input sequence.
- **Positional Encoding:** A method to add information about the position of tokens in a sequence to the model, which is crucial since transformers do not have an inherent way to understand the order of tokens.

## 3. Deep Dive into Attention in Transformers

### 3.1. Self-Attention Mechanism

Self-attention, also known as intra-attention, is a mechanism where each token in the sequence is allowed to attend to all other tokens. This is achieved through a series of linear transformations and a softmax function.

#### Step-by-Step Walkthrough:

1. **Input Embeddings:** Each token in the input sequence is first converted into a dense vector representation, often called an embedding.
  - Let  $X$  be the input matrix of shape  $[N, D]$ , where  $N$  is the sequence length and  $D$  is the embedding dimension.

2. **Linear Transformations:** Three linear transformations are applied to the input embeddings to produce the query ( $Q$ ), key ( $K$ ), and value ( $V$ ) matrices.

- $Q = XW_Q$
- $K = XW_K$
- $V = XW_V$
- Here,  $W_Q$ ,  $W_K$ , and  $W_V$  are learnable weight matrices.

3. **Attention Scores:** The attention scores are computed by taking the dot product of the query and key matrices, scaled by the square root of the key dimension.

- Attention Scores =  $\frac{QK^T}{\sqrt{d_k}}$
- $d_k$  is the dimension of the key vectors.

4. **Softmax:** The attention scores are passed through a softmax function to obtain the attention weights.

- Attention Weights =  $\text{softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right)$

5. **Weighted Sum:** The attention weights are used to compute a weighted sum of the value vectors.

- Output = Attention Weights  $\cdot V$

### 3.2. Multi-Head Attention

Multi-head attention allows the model to attend to different subspaces of the input embeddings, capturing different types of information.

- Step-by-Step:

1. **Split the Embeddings:** The input embeddings are split into multiple heads.
2. **Apply Self-Attention:** Self-attention is applied to each head independently.
3. **Concatenate the Outputs:** The outputs from each head are concatenated.
4. **Linear Transformation:** A final linear transformation is applied to the concatenated output.

### 3.3. Positional Encoding

Since transformers do not have an inherent way to understand the order of tokens, positional encoding is added to the input embeddings to provide this information.

- **Formula:**

$$\text{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right)$$

$$\text{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

- $pos$  is the position of the token in the sequence.
- $i$  is the position within the embedding dimension.
- $d$  is the embedding dimension.

## 4. Python Example

Here's a Python example using PyTorch to illustrate the self-attention mechanism:

```
import torch
import torch.nn as nn

class SelfAttention(nn.Module):
    def __init__(self, embed_size, heads):
        super(SelfAttention, self).__init__()
        self.embed_size = embed_size
        self.heads = heads
        self.head_dim = embed_size // heads

        assert (self.head_dim * heads == embed_size), "Embed size must be divisible by heads"

        self.values = nn.Linear(self.head_dim, self.head_dim)
        self.keys = nn.Linear(self.head_dim, self.head_dim,
                             bias=False)
        self.queries = nn.Linear(self.head_dim, self.head_dim,
                               bias=False)
        self.fc_out = nn.Linear(heads * self.head_dim, embed_size)

    def forward(self, values, keys, query, mask):
        N = query.shape[0]
        value_len, key_len, query_len = values.shape[1], keys.shape[1], query.shape[1]

        # Split the embedding into self.heads different pieces
        values = values.reshape(N, value_len, self.heads, self.head_dim)
        keys = keys.reshape(N, key_len, self.heads, self.head_dim)
        queries = query.reshape(N, query_len, self.heads, self.head_dim)

        values = self.values(values)
        keys = self.keys(keys)
        queries = self.queries(queries)

        # Compute the dot product of queries and keys
        energy = torch.einsum("nqhd,nkhd->nhqk", [queries,
```

```
if mask is not None:
    energy = energy.masked_fill(mask == 0, float("-inf"))

attention = torch.softmax(energy / (self.embed_size ** 0.5), dim=-1)

out = torch.einsum("nhql,nlhd->nqhd", [attention, v], [N, query_len, self.heads * self.head_dim])
out = self.fc_out(out)
return out

# Example usage
embed_size = 256
heads = 8
N = 2 # Batch size
seq_len = 10 # Sequence length

# Random input tensor
x = torch.randn((N, seq_len, embed_size))

# Initialize the self-attention layer
attention = SelfAttention(embed_size, heads)

# Forward pass
output = attention(x, x, x, None)
print(output.shape) # Output shape: [2, 10, 256]
```

## 5. Visualization with Bertviz

To better understand how attention works, you can use the `bertviz` library to visualize the attention weights. Here's how you can do it:

```
from bertviz import model_view
import transformers

# Load a pre-trained BERT model and tokenizer
model_name = 'bert-base-uncased'
model = transformers.BertModel.from_pretrained(model_name)
tokenizer = transformers.BertTokenizer.from_pretrained(mode

# Example input sentence
sentence = "The quick brown fox jumps over the lazy dog"

# Tokenize the input
tokens = tokenizer.tokenize(sentence)
tokens = ['[CLS]'] + tokens + ['[SEP]']
input_ids = torch.tensor([tokenizer.convert_tokens_to_ids(t

# Forward pass through the model
outputs = model(input_ids)
attention = outputs[-1] # Attention weights

# Visualize the attention
model_view(attention, tokens)
```

## 6. Summary and Next Steps

Summary:

- **Attention** is a mechanism that allows models to focus on specific parts of the input sequence.
- **Self-attention** computes attention scores by comparing query, key, and value vectors.
- **Multi-head attention** allows the model to attend to different subspaces of the input embeddings.
- **Positional encoding** provides information about the position of tokens in the sequence.

#### Next Steps:

- **Explore Multi-Head Attention:** Dive deeper into how multi-head attention works and its benefits.
- **Understand Transformer Architecture:** Learn about the full transformer architecture, including the encoder and decoder layers.
- **Hands-On Projects:** Apply transformers to real-world tasks such as text classification, translation, or image captioning.

## Resources for Further Reading

- **Bertviz and Attention Weights:** [<https://pantelis.github.io/aiml-common/lectures/nlp/transformers/singlehead-self-attention.html>]
- **Transformers and Language Modeling:** [<https://pantelis.github.io/book/multimodal/index.html>]
- **Transformer Models and Self-Attention Networks:** [<https://www.aclweb.org/anthology/P18-1238.pdf>]
- **Positional Encoding:** [[https://pantelis.github.io/aiml-common/lectures/vision-transformers/vision\\_transformer\\_tutorial.html](https://pantelis.github.io/aiml-common/lectures/vision-transformers/vision_transformer_tutorial.html)]

- **Transformer-Based Models:** [[https://pantelis.github.io/aiml-common/lectures/transfer-learning/transfer\\_learning\\_tutorial.html](https://pantelis.github.io/aiml-common/lectures/transfer-learning/transfer_learning_tutorial.html)]

Feel free to ask more questions or explore these resources to deepen your understanding of AI/ML.

Ask a question about AI/ML... 