Tutorial: Understanding of Attention Mechanisms in Transformers
GitHub: https://github.com/PKSR-DS/Attention-Mechanisms-in-Transformers
Student Name: Praveen Kumar Savariraj
Student ID: 23089117

Introduction

- Transformers made revolutionary changes in Natural Language Processing (NLP) by enabling the processing of long-range dependencies effectively.
- Unlike earlier models, they use attention mechanisms to dynamically prioritize different parts of the input, leading to better understanding and performance.
- In this tutorial, we will see two key concepts, self-attention, multi-head attention and how they enhance NLP models..

Objectives

By the end of this tutorial, we can:

- Understand the concept of self-attention and its significance in Transformers.
- We can learn about multi-head attention and how it enhances the model's ability to capture complex relationships in data.
- Also it will give learning experience by implementing both self-attention and multi-head attention using Python and PyTorch.

---

1. Self-Attention Mechanism

- Self-attention which allows each word in a sentence to attend to every other word, making the model capable of understanding context more effectively.
- For example consider this sentence, "The cat sat on the mat," the word "cat" will focus on "sat" for better understanding.

Formula for Self-Attention:

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

where:

- Q is the Query matrix
- K is the Key matrix
- V is the Value matrix
- d_k is the dimensionality of the key vectors

Code:

- The core of self-attention is the scaled dot-product attention.

[1]:
```python
import numpy as np
import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt

# Function to compute self-attention
def compute_self_attention(queries, keys, values):
    """
    This function calculates the self-attention output and the corresponding attention weights.

    Parameters:
    queries: Tensor representing the queries
    keys: Tensor representing the keys
    values: Tensor representing the values

    Returns:
    output: The resulting attention output
    attention_weights: The weights assigned to each input
    """
    # Calculate the attention scores using the scaled dot-product method
    scores = torch.matmul(queries, keys.T) / np.sqrt(keys.size(-1))

    # Apply softmax to convert scores into attention weights
    attention_weights = F.softmax(scores, dim=-1)

    # Compute the output as a weighted sum of the values
    output = torch.matmul(attention_weights, values)
    return output, attention_weights

# Create a sample input tensor representing three words with four-dimensional embeddings
sample_input = torch.tensor([[1, 0, 1, 0],
                             [0, 1, 0, 1],
                             [1, 1, 0, 0]], dtype=torch.float32)

# For simplicity, we will use the same input for queries, keys, and values
queries = keys = values = sample_input

# Calculate the self-attention output and weights
output, attention_weights = compute_self_attention(queries, keys, values)
print("Attention Output:\n", output)
print("Attention Weights:\n", attention_weights)
```

 # Example input

```python
sample_input = torch.tensor([[1, 0, 1, 0],
                             [0, 1, 0, 1],
                             [1, 1, 0, 0]], dtype=torch.float32)
```

Output,

Attention Output:
tensor([[0.8137, 0.4935, 0.5065, 0.1863],
[0.4935, 0.8137, 0.1863, 0.5065],
[0.7259, 0.7259, 0.2741, 0.2741]])
Attention Weights:
tensor([[0.5065, 0.1863, 0.3072],
[0.1863, 0.5065, 0.3072],
[0.2741, 0.2741, 0.4519]])

- The compute_self_attention function calculates the attention scores by performing a dot product between the query (Q) and key (K).
- These scores were scaled and then passed through a softmax function to normalize them into attention weights.
- The weighted sum of the values (V) is computed and returned as the attention output.

2. Visualizing Attention Weights

- Visualizing attention weights which helps us to understand which words influence each other in a sentence.

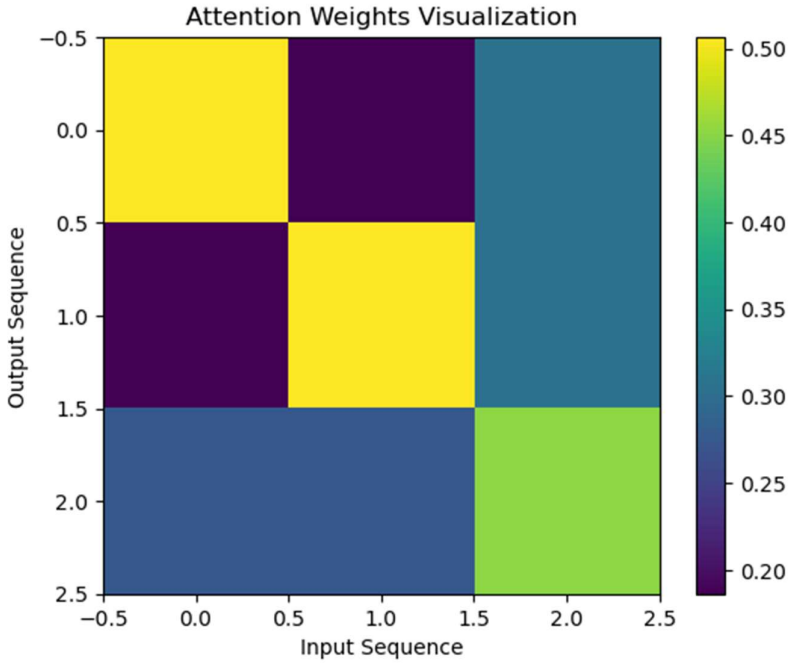
Code:

```
# Function to visualize the attention weights
def display_attention_weights(weights):
    """
    Visualizes the attention weights using a heatmap.

    Parameters:
    weights: The attention weights to visualize
    """
    plt.imshow(weights.detach().numpy(), cmap='viridis')
    plt.colorbar()
    plt.title('Attention Weights Visualization')
    plt.xlabel('Input Sequence')
    plt.ylabel('Output Sequence')
    plt.show()

# Visualize the attention weights
display_attention_weights(attention_weights)
```

- This heatmap shows how much each word (query) attends to every other word (key).
- Darker colors which represent stronger attention, allowed us to visualize the focus of the model.



3. Multi-Head Attention

- Multi-head attention which splits the self-attention mechanism into multiple attention heads, each learning different relationships in the data.
- Which allows the model to capture a richer representation by attending to multiple aspects of the input sequence simultaneously.

Formula for Multi-Head Attention:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h)W^O$$

where each head is calculated as:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

- h is the number of heads
- W_i^Q, W_i^K, W_i^V are learned weight matrices for each attention head.

Code:

- **Define the MultiHeadAttentionLayer class to split the attention mechanism into multiple heads.**

```
# Class definition for Multi-Head Attention
class MultiHeadAttentionLayer(torch.nn.Module):
    def __init__(self, embedding_size, num_heads):
        super(MultiHeadAttentionLayer, self).__init__()
        self.num_heads = num_heads
        self.embedding_size = embedding_size
        self.head_dim = embedding_size // num_heads

        # Ensure the embedding size is divisible by the number of heads
        assert (self.head_dim * num_heads == embedding_size), "Embedding size must be divisible by number of heads"

        # Linear layers for values, keys, and queries
        self.values_layer = torch.nn.Linear(embedding_size, embedding_size, bias=False)
        self.keys_layer = torch.nn.Linear(embedding_size, embedding_size, bias=False)
        self.queries_layer = torch.nn.Linear(embedding_size, embedding_size, bias=False)
        self.output_layer = torch.nn.Linear(embedding_size, embedding_size)

    def forward(self, x):
        """
        Forward pass for the multi-head attention layer.

        Parameters:
        x: Input tensor of shape (batch_size, seq_length, embedding_size)

        Returns:
        output: The output of the multi-head attention layer
        """
        batch_size = x.shape[0]
        seq_length = x.shape[1]

        # Transform the input into multiple heads
        values = self.values_layer(x).view(batch_size, seq_length, self.num_heads, self.head_dim)
        keys = self.keys_layer(x).view(batch_size, seq_length, self.num_heads, self.head_dim)
        queries = self.queries_layer(x).view(batch_size, seq_length, self.num_heads, self.head_dim)

        # Rearrange dimensions for multi-head attention
        values = values.permute(0, 2, 1, 3)  # (batch_size, num_heads, seq_length, head_dim)
        keys = keys.permute(0, 2, 1, 3)      # (batch_size, num_heads, seq_length, head_dim)
        queries = queries.permute(0, 2, 1, 3)  # (batch_size, num_heads, seq_length, head_dim)
```

```
        # Calculate attention scores
        energy = torch.einsum("nqhd,nkhd->nhqk", [queries, keys])  # (batch_size, num_heads, query_length, key_length)
        attention_scores = F.softmax(energy / (self.embedding_size ** (1 / 2)), dim=3)

        # Compute the output using the attention scores
        out = torch.einsum("nhql,nlhd->nqhd", [attention_scores, values]).reshape(batch_size, seq_length, self.num_heads * self.head_dim)
        return self.output_layer(out)

# Example usage of the Multi-Head Attention class
if __name__ == "__main__":
    embedding_size = 4  # Size of the embedding
    num_heads = 2       # Number of attention heads
    multi_head_attention_layer = MultiHeadAttentionLayer(embedding_size, num_heads)

    # Perform a forward pass with the sample input
    output_multihead = multi_head_attention_layer(sample_input.unsqueeze(0))  # Add batch dimension
    print("Output from Multi-Head Attention:\n", output_multihead)
```

- The MultiHeadAttentionLayer class splits the input into multiple heads. Each head computes attention independently, and the results were combined.

- Forward Method: The input was split into heads, each computing its attention scores. The results are then combined and passed through a final layer.

Output from Multi-Head Attention:

```
tensor([[[-0.9761,  0.0010, -0.6647,  0.3468],
         [-0.2477, -0.5708, -0.3520,  0.4833],
         [-0.5272, -0.4351, -0.5420,  0.2334]]], grad_fn=<ViewBackward0>)
```

Conclusion

- Self-attention and multi-head attention were central to Transformer models which enabling them to capture relationships in sequential data.
- This tutorial provided a deep dive into the mechanics behind these attention techniques, including both theory and practical implementation using Python and PyTorch.
- Understanding these techniques will help us to build and modify Transformer models more effectively.

References

1. Vaswani, A., et al. (2017). Attention is All You Need. NeurIPS.
2. Brown, T. B., et al. (2020). Language Models are Few-Shot Learners. NeurIPS.
3. Wolf, T., et al. (2020). Transformers: State-of-the-Art Natural Language Processing. arXiv preprint arXiv:1910.03771.
4. Self-Attention in Transformer Neural Networks (with Code!) Link
5. Attention in transformers, step-by-step | DL6 Link
6. Scaled Dot Product Attention Explained + Implemented Link

Accessibility Consideration:

- Visual Impairments: The visualizations use a distinguishable color palette for individuals with color blindness. Alternative text descriptions are provided for images.
- Hearing Impairments: Given youtube video tutorials have subtitles/transcripts included which ensures accessibility for people with hearing impairments.

Summary:

- Transformers which revolutionize NLP with attention mechanisms to handle long-range dependencies.
- Self-Attention which allows words in a sentence to focus on each other for better context understanding.
    - Formula: Dot product between Query (Q) and Key (K), scaled and passed through softmax.
- Visualizing Attention: Heatmaps shows which words influence each other.
- Multi-Head Attention: Splits attention into multiple heads, each capturing different relationships.