Multi-Head Attention Mechanism

Overview

Multi-Head Attention is a powerful extension of the attention mechanism. It allows the model to focus on different parts of the input sequence simultaneously by splitting the attention mechanism into multiple "heads." Each head learns a different attention representation, which is then concatenated and processed through a final linear layer. Multi-Head Attention is an essential component of the Transformer architecture.

Why Use Multi-Head Attention?

- **Parallel Attention**: Allows the model to focus on multiple aspects of the input simultaneously, providing richer representations.
- **Improved Learning**: By using multiple attention heads, the model can capture various relationships in the data, improving its understanding of the sequence.
- **Scalability**: It scales well with longer sequences, as it can attend to different parts of the sequence in parallel.

Prerequisites

- **Python 3.x**: Ensure Python 3.x is installed.

- **PyTorch**: Install PyTorch compatible with your system.

  ```
  pip install torch
  ```

Code Description

1. **Data Preparation**:

   Random data for the input sequence, query, key, and value vectors are created.

   ```
   query = torch.randn(1, 20)
  key = torch.randn(10, 20)
  value = torch.randn(10, 20)
   ```

   - `query` : The query vector, usually the output from a decoder.
   - `key` and `value` : Represent the set of keys and corresponding values (typically the output from an encoder).

2. **Multi-Head Attention Class**:

   The `MultiHeadAttention` class defines the layers and forward pass for the multi-head attention mechanism.

```
class MultiHeadAttention(nn.Module):
    def __init__(self, input_dim, num_heads):
        super(MultiHeadAttention, self).__init__()
        self.num_heads = num_heads
        self.head_dim = input_dim // num_heads
        self.query_layer = nn.Linear(input_dim, input_dim)
        self.key_layer = nn.Linear(input_dim, input_dim)
        self.value_layer = nn.Linear(input_dim, input_dim)
        self.output_layer = nn.Linear(input_dim, input_dim)

    def forward(self, query, key, value):
        batch_size = query.size(0)

        # Apply linear projections to query, key, value
        query = self.query_layer(query).view(batch_size, -1, self.num_heads, self.h
        key = self.key_layer(key).view(batch_size, -1, self.num_heads, self.head_di
        value = self.value_layer(value).view(batch_size, -1, self.num_heads, self.h

        # Compute Scaled Dot-Product Attention for each head
        attention_output, attention_weights = self.scaled_dot_product_attention(que

        # Concatenate heads and apply final output linear layer
        attention_output = attention_output.view(batch_size, -1, self.num_heads * s
        output = self.output_layer(attention_output)
        return output, attention_weights
```

- **Initialization**: The class initializes the layers for projecting the input query, key, and value vectors into different subspaces (one for each attention head).
- **Forward Method**: Applies the linear projections to the query, key, and value vectors, then computes the attention output for each head, concatenates the results, and applies the final output layer.

3. **Scaled Dot-Product Attention**:

   This function computes the attention for each head by calculating the dot product of the query and key, followed by a softmax normalization to obtain the attention weights. These weights are then used to get the weighted sum of the value vectors.

```
def scaled_dot_product_attention(self, query, key, value):
    # Compute Scaled Dot-Product Attention
    scores = torch.matmul(query, key.transpose(-2, -1)) / np.sqrt(self.head_dim)
    attention_weights = torch.softmax(scores, dim=-1)
    attention_output = torch.matmul(attention_weights, value)
    return attention_output, attention_weights
```

   - `scores` : The dot product between the query and key is scaled by the square root of the dimension of the key (for numerical stability).
   - `attention_weights` : The softmax over the dot products generates a probability distribution.
   - `attention_output` : The weighted sum of the value vectors, based on the attention weights.

4. **Attention Computation**:

   The `MultiHeadAttention` module is initialized, and the attention output and attention weights are computed by passing the query, key, and value vectors.

```
multi_head_attention = MultiHeadAttention(input_dim=20, num_heads=4)
output, attention_weights = multi_head_attention(query, key, value)
```

   - The `output` is the concatenated attention result after applying the linear output layer.
   - The `attention_weights` represent how much attention each key receives for the query.

5. **Output**:

   The attention output and the attention weights are printed for inspection.

```
print("Multi-Head Attention Output:", output)
print("Attention Weights:", attention_weights)
```

Expected Outputs

- **Multi-Head Attention Output**: The final output vector after the attention mechanism is applied.
- **Attention Weights**: A tensor indicating the attention weights assigned to each key-value pair.

Use Cases

- **Machine Translation**: Multi-Head Attention is a core component of the Transformer architecture used in machine translation.
- **Speech Recognition**: Attention mechanisms help focus on relevant parts of the input audio sequence.
- **Text Summarization**: In summarization models, Multi-Head Attention allows the model to focus on different parts of the input text when generating summaries.

Future Enhancements

1. **Layer Normalization**: Add layer normalization before or after each attention step to stabilize training.
2. **Global Attention**: Extend the attention to incorporate context from a larger portion of the input sequence.
3. **Efficiency Improvements**: Use sparse attention mechanisms or approximation techniques to reduce computation costs for long sequences.

References

- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. A., Kaiser, ?., & Polosukhin, I. (2017). Attention is All You Need. arXiv:1706.03762.