Additive Attention Mechanism

Overview

Additive Attention, introduced by Bahdanau et al. (2015), is a mechanism commonly used in sequence-to-sequence models. It computes attention scores between a query and a set of keys, then uses these scores to weight the corresponding values. The core of the mechanism is an energy function that computes compatibility between the query and keys.

Why Use Additive Attention?

- **Alignment Learning**: Helps in determining which parts of the input sequence are important for producing the output.
- **Sequence-to-Sequence Models**: Typically used in tasks such as machine translation, speech recognition, and text summarization.
- **Improved Contextualization**: Adds the ability to focus on different parts of the input sequence for different outputs.

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**:

   Here, random data for input sequences, a single query vector, and keys/values for the attention mechanism are generated.

   ```
   X = np.random.randn(10, 20)  # Random data for input (10 sequences, 20 features)
   query = torch.randn(1, 20)  # A single query vector
   key = torch.randn(10, 20)  # Key for all sequences
   value = torch.randn(10, 20)  # Value for all sequences
   ```

   - `X` : The input data (for example, the output from an encoder).
   - `query` : A vector representing the query (typically the output from a decoder).
   - `key` : The set of keys, representing all possible input sequences.
   - `value` : The corresponding values for each key.

2. **Additive Attention Class**:

   The `AdditiveAttention` class defines the layers and forward pass of the attention mechanism.

```
class AdditiveAttention(nn.Module):
    def __init__(self, input_dim, hidden_dim):
        super(AdditiveAttention, self).__init__()
        self.query_layer = nn.Linear(input_dim, hidden_dim)
        self.key_layer = nn.Linear(input_dim, hidden_dim)
        self.energy_layer = nn.Linear(hidden_dim, 1)

    def forward(self, query, key, value):
        query = self.query_layer(query).unsqueeze(0)
        key = self.key_layer(key)
        energy = torch.tanh(query + key)
        attention_weights = torch.softmax(self.energy_layer(energy).squeeze(), dim=
        weighted_sum = torch.sum(attention_weights.unsqueeze(1) * value, dim=0)
        return weighted_sum, attention_weights
```

- `query_layer` and `key_layer`: Linear layers to transform the query and key vectors into a shared hidden space.
- `energy_layer`: Computes a scalar energy score from the sum of transformed query and key vectors.
- `forward`: The attention mechanism that computes the attention weights and the weighted sum of values.

3. **Attention Computation**:

   The attention mechanism is applied to the input data, computing the weighted sum and attention weights.

```
attention = AdditiveAttention(input_dim=20, hidden_dim=32)
weighted_sum, attention_weights = attention(query, key, value)
```

- The `weighted_sum` is the final output, which is a weighted sum of the value vectors based on the computed attention weights.
- The `attention_weights` represent how much focus each input sequence (key-value pair) receives for this query.

4. **Output**:

   The weighted sum and attention weights are printed for inspection.

```
print("Weighted Sum:", weighted_sum)
print("Attention Weights:", attention_weights)
```

Expected Outputs

- **Weighted Sum**: The resulting weighted sum of the value vectors based on attention scores.
- **Attention Weights**: A tensor indicating how much attention is given to each key-value pair in the sequence.

Use Cases

- **Machine Translation**: Additive attention can be applied in machine translation models to focus on relevant parts of the input sentence when generating the translated output.
- **Speech Recognition**: In speech-to-text models, attention mechanisms help focus on specific parts of the speech signal when transcribing it to text.
- **Text Summarization**: For generating summaries, the model can focus on the most relevant portions of the document.

Future Enhancements

1. **Multi-Head Attention**: Implement multi-head attention to learn different attention representations from different parts of the input.
2. **Scalability**: Improve the mechanism to handle longer sequences efficiently (e.g., by using sparse attention or performing attention in chunks).
3. **Visualization**: Add functionality to visualize the attention weights to understand which parts of the input are being focused on during prediction.

References

- Bahdanau, D., Cho, K., & Bengio, Y. (2015). Neural Machine Translation by Jointly Learning to Align and Translate. arXiv:1409.0473.