Transformer_Encoding-3 (Score: 0.0 / 100.0)

1. Test cell (Score: 0.0 / 100.0)

Step 1: Tokenizer and Vocabulary Creation (Add more words if needed)¶

1. Import Necessary Libraries¶

Import the required PyTorch modules.

In [ ]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

You are working on building a simple machine translation model. The first step in the pipeline is to tokenize the given sentences and create vocabulary mappings for both the source and target languages.

In [ ]:

```
# Step 1: Tokenizer and Vocabulary Creation (Add more words if needed)
sentence_en = "I love AI ."
sentence_fr = "J' adore l'IA ."

word_map_en = {"<pad>": 0, "I": 1, "love": 2, "AI": 3, ".": 4}
word_map_fr = {"<pad>": 0, "J'": 1, "adore": 2, "l'IA": 3, ".": 4}
```

2. Tokenizing sentences¶

Implement the tokenize function to map each word in the sentence to the corresponding index from the word_map. Ensure that the function returns a PyTorch tensor containing the word indices.

In [ ]:

```
# Tokenizing sentences
def tokenize(sentence, word_map):
    return torch.tensor([word_map[word] for word in sentence.split()])
```

3.Tokenize the input and target sentences¶

Tokenize the input (sentence_en) and target (sentence_fr) sentences. Add the extra dimension to each tokenized tensor using .unsqueeze(0) to make the shape (1, N).

In [ ]:

```
# Tokenize the input and target sentences
input_tensor = tokenize(sentence_en, word_map_en).unsqueeze(0)  # Shape (1, 6)
target_tensor = tokenize(sentence_fr, word_map_fr).unsqueeze(0)  # Shape (1, 6)
```

Step 2: Positional Encoding¶

You are tasked with implementing a Positional Encoding layer in PyTorch, which will be used to inject information about the position of tokens in a sequence. This encoding is crucial for models like the Transformer, which lacks any inherent notion of token position in its architecture. 1.Positional Encoding Initialization: The PositionalEncoding class will be initialized with the following parameters: d_model: The dimension of the embeddings (e.g., size of word vectors). max_len: The maximum sequence length that can be processed (default is 5000). The positional encoding should be created based on the formula using sine and cosine functions as shown in the code. 2.Encoding Calculation: You need to calculate the positional encoding matrix using the formula: For even indices (i.e., positions 0, 2, 4,...), use the sine function: sin(position / 10000^(2i/d_model)) For odd indices (i.e., positions 1, 3, 5,...), use the cosine function: cos(position / 10000^(2i/d_model)) This positional encoding matrix will be added to the input sequence in the forward method. 3.Forward Pass: In the forward method, the input tensor x will be added to the positional encoding corresponding to its sequence length. The output tensor will be the sum of the input tensor and the positional encoding, which gives the input tokens with position information. Expected Input and Output: Given an input sequence of shape (batch_size, seq_len, d_model)

In [ ]:

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.encoding = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * -(torch.log(torch.tensor(10000.0)) / d_model))
        self.encoding[:, 0::2] = torch.sin(position * div_term)
        self.encoding[:, 1::2] = torch.cos(position * div_term)
        self.encoding = self.encoding.unsqueeze(0)

    def forward(self, x):
        return x + self.encoding[:, :x.size(1)]
```

Step 3: Multi-Head Attention¶

You are building a Multi-Head Attention mechanism, which is a key component of the Transformer model architecture. The goal is to implement the MultiHeadAttention class, which will allow the model to focus on different parts of the input sequence simultaneously using multiple attention heads. 1.Initialization: The MultiHeadAttention class should be initialized with: d_model: The dimensionality of the input and output features (i.e., the size of the word vectors). num_heads: The number of attention heads. The following layers should be initialized: query, key, value: Linear transformations for computing the query, key, and value vectors for attention. fc: A linear layer to project the concatenated attention outputs from all heads back to d_model. 2.Forward Pass: Split the input tensor x into multiple heads. For each head, compute the query, key, and value vectors. Compute the attention scores using the scaled dot-product attention formula: where Q is the query, K is the key, and d_k is the dimension of the key vectors. Apply the attention mask if provided (to handle padding or certain tokens). Use the softmax function to compute the attention weights. Compute the output by applying the attention weights to the value vectors. Finally, concatenate the outputs from all attention heads and pass them through the final linear layer to obtain the final output. Expected Input and Output: Given: An input tensor x of shape (batch_size, seq_len, d_model). The output should be of shape (batch_size, seq_len, d_model), which is the attention-weighted output for the input sequence.

In [ ]:

```
# Step 3: Multi-Head Attention
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(MultiHeadAttention, self).__init__()
        self.num_heads = num_heads
        self.d_model = d_model
        self.d_k = d_model // num_heads
        self.d_v = d_model // num_heads

        self.query = nn.Linear(d_model, d_model)
        self.key = nn.Linear(d_model, d_model)
        self.value = nn.Linear(d_model, d_model)

        self.fc = nn.Linear(d_model, d_model)

    def forward(self, x, mask=None):
        batch_size, seq_len, _ = x.size()

        q = self.query(x).view(batch_size, seq_len, self.num_heads, self.d_k).transpose(1, 2)
        k = self.key(x).view(batch_size, seq_len, self.num_heads, self.d_k).transpose(1, 2)
        v = self.value(x).view(batch_size, seq_len, self.num_heads, self.d_v).transpose(1, 2)

        attn_scores = torch.matmul(q, k.transpose(-2, -1)) / torch.sqrt(torch.tensor(self.d_k, dtype=torch.float))
        if mask is not None:
            attn_scores = attn_scores.masked_fill(mask == 0, float('-inf'))  # Ensure mask matches input length
        attn_weights = F.softmax(attn_scores, dim=-1)

        attention_output = torch.matmul(attn_weights, v).transpose(1, 2).contiguous().view(batch_size, seq_len, self.d_model)
        output = self.fc(attention_output)

        return output
```

Step 4: Feed-Forward Network¶

You are tasked with implementing a Feed-Forward Network (FFN), a critical part of the Transformer architecture. The FFN processes each token's representation independently, consisting of two fully connected layers. The first layer transforms the input features into a larger space (d_ff), and the second layer projects it back to the original feature dimension (d_model). A ReLU activation function is applied between these two layers. 1.Initialization: The FeedForward class should be initialized with: d_model: The dimensionality of the input and output feature vectors. d_ff: The dimensionality of the hidden layer (default value is 512). 2.Forward Pass: The forward pass will apply two fully connected layers: First, the input tensor x (of shape [batch_size, seq_len, d_model]) will be passed through a linear layer (fc1) that projects it to the higher dimension d_ff. Then, a ReLU activation function is applied to the output of fc1. Finally, the result will be passed through a second linear layer (fc2), which projects the output back to the original dimension d_model. 3.Expected Input and Output: Given an input tensor x of shape [batch_size, seq_len, d_model], the output will be the same shape [batch_size, seq_len, d_model], but with transformed features.

In [ ]:

Student's answer (Top)

```
# YOUR CODE HERE
#raise NotImplementedError()
class FeedForward(nn.Module):
    def __init__(self,d_model,d_ff=512):
        super(FeedForward,self).__init__()
        self.fc1=nn.Linear(d_model,d_ff)
        self.fc2=nn.Linear(d_ff,d_model)
    def forward(self,x):
        return self.fc2(F.relu(self.fc1(x)))
```

Step 5: Encoder Layer¶

You are tasked with implementing an Encoder Layer for a Transformer model. The encoder layer consists of two main components:

Multi-Head Attention: This mechanism allows the model to focus on different parts of the input sequence simultaneously. Feed-Forward Network (FFN): This component applies two fully connected layers to each position independently. In addition, each of these components is followed by Layer Normalization and Residual Connections to stabilize training.

Task Breakdown: Initialization: The EncoderLayer class should be initialized with: d_model: The dimensionality of the input and output features. num_heads: The number of attention heads in the multi-head attention mechanism. d_ff: The dimensionality of the hidden layer in the feed-forward network (default is 512). The following components should be initialized: multihead_attn: A MultiHeadAttention layer. feedforward: A FeedForward layer. norm1, norm2: Two Layer Normalization layers.

Forward Pass:

The input tensor x is first passed through the multi-head attention mechanism (multihead_attn). The output is added to the original input x (residual connection), and then Layer Normalization (norm1) is applied. The resulting tensor is then passed through the feed-forward network (feedforward), and again the residual connection is added followed by Layer Normalization (norm2). The output of this forward pass will be the transformed tensor with attention and feed-forward operations applied.

In [ ]:

Student's answer (Top)

```
# YOUR CODE HERE
#raise NotImplementedError()
class EncoderLayer(nn.Module):
    def _init_(self, d_model, num_heads, d_ff=512):
        super(EncoderLayer, self)._init_()

        self.multihead_attn = nn.MultiheadAttention(embed_dim=d_model, num_heads=num_heads)

        self.feedforward = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
            nn.Linear(d_ff, d_model)
        )

        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)


        self.dropout1 = nn.Dropout(0.1)
        self.dropout2 = nn.Dropout(0.1)

    def forward(self, x, mask=None):

        attn_output, _ = self.multihead_attn(x, x, x, attn_mask=mask)
        x = self.norm1(x + self.dropout1(attn_output))



        ff_output = self.feedforward(x)
        x = self.norm2(x + self.dropout2(ff_output))

        return x
```

Test Case 3: Check Encoder Layer¶

In [ ]:

Grade cell: `cell-13c589c77c5e763f` Score: 0.0 / 100.0 (Top)