```python
import numpy as np
import copy

import torch
import torch.nn as nn

from ..transformer_layers import *


class CaptioningTransformer(nn.Module):
    """
    A CaptioningTransformer produces captions from image features using a
    Transformer decoder.

    The Transformer receives input vectors of size D, has a vocab size of V,
    works on sequences of length T, uses word vectors of dimension W, and
    operates on minibatches of size N.
    """
    def __init__(self, word_to_idx, input_dim, wordvec_dim, num_heads=4,
                 num_layers=2, max_length=50):
        """
        Construct a new CaptioningTransformer instance.

        Inputs:
        - word_to_idx: A dictionary giving the vocabulary. It contains V entries.
          and maps each string to a unique integer in the range [0, V).
        - input_dim: Dimension D of input image feature vectors.
        - wordvec_dim: Dimension W of word vectors.
        - num_heads: Number of attention heads.
        - num_layers: Number of transformer layers.
        - max_length: Max possible sequence length.
        """
        super().__init__()

        vocab_size = len(word_to_idx)
        self.vocab_size = vocab_size
        self._null = word_to_idx["<NULL>"]
        self._start = word_to_idx.get("<START>", None)
        self._end = word_to_idx.get("<END>", None)

        self.visual_projection = nn.Linear(input_dim, wordvec_dim)
        self.embedding = nn.Embedding(vocab_size, wordvec_dim, padding_idx=self._null)
        self.positional_encoding = PositionalEncoding(wordvec_dim, max_len=max_length)
```

```python
        decoder_layer = TransformerDecoderLayer(input_dim=wordvec_dim, num_heads=num_heads)
        self.transformer = TransformerDecoder(decoder_layer, num_layers=num_layers)
        self.apply(self._init_weights)

        self.output = nn.Linear(wordvec_dim, vocab_size)

    def _init_weights(self, module):
        """
        Initialize the weights of the network.
        """
        if isinstance(module, (nn.Linear, nn.Embedding)):
            module.weight.data.normal_(mean=0.0, std=0.02)
            if isinstance(module, nn.Linear) and module.bias is not None:
                module.bias.data.zero_()
        elif isinstance(module, nn.LayerNorm):
            module.bias.data.zero_()
            module.weight.data.fill_(1.0)

    def forward(self, features, captions):
        """
        Given image features and caption tokens, return a distribution over the
        possible tokens for each timestep. Note that since the entire sequence
        of captions is provided all at once, we mask out future timesteps.

        Inputs:
         - features: image features, of shape (N, D)
         - captions: ground truth captions, of shape (N, T)

        Returns:
         - scores: score for each token at each timestep, of shape (N, T, V)
        """
        N, T = captions.shape
        # Create a placeholder, to be overwritten by your code below.
        scores = torch.empty((N, T, self.vocab_size))
        ############################################################################
        # TODO: Implement the forward function for CaptionTransformer.             #
        # A few hints:                                                             #
        #  1) You first have to embed your caption and add positional              #
        #     encoding. You then have to project the image features into the same  #
        #     dimensions.                                                          #
        #  2) You have to prepare a mask (tgt_mask) for masking out the future     #
        #     timesteps in captions. torch.tril() function might help in preparing #
        #     this mask.                                                           #
        #  3) Finally, apply the decoder features on the text & image embeddings   #
        #     along with the tgt_mask. Project the output to scores per token      #
        ############################################################################
```

```python
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        pass

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        ############################################################################
        #                             END OF YOUR CODE                             #
        ############################################################################

        return scores

    def sample(self, features, max_length=30):
        """
        Given image features, use greedy decoding to predict the image caption.

        Inputs:
         - features: image features, of shape (N, D)
         - max_length: maximum possible caption length

        Returns:
         - captions: captions for each example, of shape (N, max_length)
        """
        with torch.no_grad():
            features = torch.Tensor(features)
            N = features.shape[0]

            # Create an empty captions tensor (where all tokens are NULL).
            captions = self._null * np.ones((N, max_length), dtype=np.int32)

            # Create a partial caption, with only the start token.
            partial_caption = self._start * np.ones(N, dtype=np.int32)
            partial_caption = torch.LongTensor(partial_caption)
            # [N] -> [N, 1]
            partial_caption = partial_caption.unsqueeze(1)

            for t in range(max_length):

                # Predict the next token (ignoring all other time steps).
                output_logits = self.forward(features, partial_caption)
                output_logits = output_logits[:, -1, :]

                # Choose the most likely word ID from the vocabulary.
                # [N, V] -> [N]
                word = torch.argmax(output_logits, axis=1)

                # Update our overall caption and our current partial caption.
```

```python
                captions[:, t] = word.numpy()
                word = word.unsqueeze(1)
                partial_caption = torch.cat([partial_caption, word], dim=1)

            return captions


class TransformerDecoderLayer(nn.Module):
    """
    A single layer of a Transformer decoder, to be used with TransformerDecoder.
    """
    def __init__(self, input_dim, num_heads, dim_feedforward=2048, dropout=0.1):
        """
        Construct a TransformerDecoderLayer instance.

        Inputs:
         - input_dim: Number of expected features in the input.
         - num_heads: Number of attention heads
         - dim_feedforward: Dimension of the feedforward network model.
         - dropout: The dropout value.
        """
        super().__init__()
        self.self_attn = MultiHeadAttention(input_dim, num_heads, dropout)
        self.multihead_attn = MultiHeadAttention(input_dim, num_heads, dropout)
        self.linear1 = nn.Linear(input_dim, dim_feedforward)
        self.dropout = nn.Dropout(dropout)
        self.linear2 = nn.Linear(dim_feedforward, input_dim)

        self.norm1 = nn.LayerNorm(input_dim)
        self.norm2 = nn.LayerNorm(input_dim)
        self.norm3 = nn.LayerNorm(input_dim)
        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)
        self.dropout3 = nn.Dropout(dropout)

        self.activation = nn.ReLU()


    def forward(self, tgt, memory, tgt_mask=None):
        """
        Pass the inputs (and mask) through the decoder layer.

        Inputs:
        - tgt: the sequence to the decoder layer, of shape (N, T, W)
        - memory: the sequence from the last layer of the encoder, of shape (N, S, D)
        - tgt_mask: the parts of the target sequence to mask, of shape (T, T)
```

```python
        Returns:
        - out: the Transformer features, of shape (N, T, W)
        """
        # Perform self-attention on the target sequence (along with dropout and
        # layer norm).
        tgt2 = self.self_attn(query=tgt, key=tgt, value=tgt, attn_mask=tgt_mask)
        tgt = tgt + self.dropout1(tgt2)
        tgt = self.norm1(tgt)

        # Attend to both the target sequence and the sequence from the last
        # encoder layer.
        tgt2 = self.multihead_attn(query=tgt, key=memory, value=memory)
        tgt = tgt + self.dropout2(tgt2)
        tgt = self.norm2(tgt)

        # Pass
        tgt2 = self.linear2(self.dropout(self.activation(self.linear1(tgt))))
        tgt = tgt + self.dropout3(tgt2)
        tgt = self.norm3(tgt)
        return tgt

def clones(module, N):
    "Produce N identical layers."
    return nn.ModuleList([copy.deepcopy(module) for _ in range(N)])

class TransformerDecoder(nn.Module):
    def __init__(self, decoder_layer, num_layers):
        super().__init__()
        self.layers = clones(decoder_layer, num_layers)
        self.num_layers = num_layers

    def forward(self, tgt, memory, tgt_mask=None):
        output = tgt

        for mod in self.layers:
            output = mod(output, memory, tgt_mask=tgt_mask)

        return output
```