```python
# Code by Sarah Wiegreffe (saw@gatech.edu)
# Fall 2019

import numpy as np

import torch
from torch import nn
import random

####### Do not modify these imports.

class ClassificationTransformer(nn.Module):
    """
    A single-layer Transformer which encodes a sequence of text and
    performs binary classification.

    The model has a vocab size of V, works on
    sequences of length T, has an hidden dimension of H, uses word vectors
    also of dimension H, and operates on minibatches of size N.
    """
    def __init__(self, word_to_ix, hidden_dim=128, num_heads=2, dim_feedforward=2048, dim_k=96, dim_v=96, dim_q=96, max_length=43):
        '''
        :param word_to_ix: dictionary mapping words to unique indices
        :param hidden_dim: the dimensionality of the output embeddings that go into the final layer
        :param num_heads: the number of Transformer heads to use
        :param dim_feedforward: the dimension of the feedforward network model
        :param dim_k: the dimensionality of the key vectors
        :param dim_q: the dimensionality of the query vectors
        :param dim_v: the dimensionality of the value vectors
        '''
        super(ClassificationTransformer, self).__init__()
        assert hidden_dim % num_heads == 0

        self.num_heads = num_heads
        self.word_embedding_dim = hidden_dim
        self.hidden_dim = hidden_dim; #print(f"H: {hidden_dim}")
        self.dim_feedforward = dim_feedforward
        self.max_length = max_length
        self.vocab_size = len(word_to_ix)

        self.dim_k = dim_k
        self.dim_v = dim_v
        self.dim_q = dim_q

        seed_torch(0)

        ##############################################################################
        # Deliverable 1: Initialize what you need for the embedding lookup (1 line). #
        # Hint: you will need to use the max_length parameter above.                 #
        ##############################################################################
        #num_embeddings (python:int) – size of the dictionary of embeddings
        #embedding_dim (python:int) – the size of each embedding vector
        self.token_embeddings = nn.Embedding(num_embeddings=self.vocab_size, embedding_dim=self.hidden_dim)
        self.position_embeddings = nn.Embedding(num_embeddings=self.max_length, embedding_dim=self.hidden_dim)
        ##############################################################################
        #                               END OF YOUR CODE                             #
        ##############################################################################


        ##############################################################################
        # Deliverable 2: Initializations for multi-head self-attention.              #
        # You don't need to do anything here. Do not modify this code.               #
        ##############################################################################

        # Head #1
        self.k1 = nn.Linear(self.hidden_dim, self.dim_k)
        self.v1 = nn.Linear(self.hidden_dim, self.dim_v)
        self.q1 = nn.Linear(self.hidden_dim, self.dim_q)

        # Head #2
        self.k2 = nn.Linear(self.hidden_dim, self.dim_k)
        self.v2 = nn.Linear(self.hidden_dim, self.dim_v)
        self.q2 = nn.Linear(self.hidden_dim, self.dim_q)

        self.softmax = nn.Softmax(dim=2)
        self.attention_head_projection = nn.Linear(self.dim_v * self.num_heads, self.hidden_dim)
        self.norm_mh = nn.LayerNorm(self.hidden_dim)


        ##############################################################################
        # Deliverable 3: Initialize what you need for the feed-forward layer.        #
        # Don't forget the layer normalization.                                      #
        ##############################################################################
        h = self.hidden_dim
        self.W1 = nn.Linear(h, self.dim_feedforward)
        self.W2 = nn.Linear(self.dim_feedforward, h)
        self.norm_ffn = nn.LayerNorm(h)
        ##############################################################################
        #                               END OF YOUR CODE                             #
        ##############################################################################


        ##############################################################################
        # Deliverable 4: Initialize what you need for the final layer (1-2 lines).   #
        ##############################################################################
        self.final_dense = nn.Linear(h, 1)
        self.final_sigmoid = nn.Sigmoid()
        ##############################################################################
        #                               END OF YOUR CODE                             #
        ##############################################################################


    def forward(self, inputs):
        '''
        This function computes the full Transformer forward pass.
        Put together all of the layers you've developed in the correct order.

        :param inputs: a PyTorch tensor of shape (N,T). These are integer lookups.

        :returns: the model outputs. Should be normalized scores of shape (N,1).
        '''
        outputs = None
        #############################################################################
        # Deliverable 5: Implement the full Transformer stack for the forward pass. #
        # You will need to use all of the methods you have previously defined above.#
        # You should only be calling ClassificationTransformer class methods here.  #
        #############################################################################
        """
```

```
    def compare(a, b):
        print(torch.sum(torch.pairwise_distance(a, b)).item())

    cache = inputs
    embeds = self.embed(inputs)
    compare(cache, inputs)

    cache = embeds
    hidden_states = self.multi_head_attention(embeds)
    compare(cache, embeds)

    cache = hidden_states
    outputs = self.feedforward_layer(hidden_states)
    compare(cache, hidden_states)

    cache = outputs
    scores = self.final_layer(outputs)
    compare(cache, outputs)

    outputs = scores
    print(outputs.shape)
    """
    outputs = self.final_layer(self.feedforward_layer(self.multi_head_attention(self.embed(inputs))))
    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################
    return outputs


def embed(self, inputs):
    """
    :param inputs: intTensor of shape (N,T)
    :returns embeddings: floatTensor of shape (N,T,H)
    """
    embeddings = None
    #############################################################################
    # Deliverable 1: Implement the embedding lookup.                            #
    # Note: word_to_ix has keys from 0 to self.vocab_size - 1                   #
    # This will take a few lines.                                               #
    #############################################################################
    #num_embeddings (python:int) – size of the dictionary of embeddings
    #embedding_dim (python:int) – the size of each embedding vector
    embeddings = self.token_embeddings(inputs)

    N, T = inputs.shape
    copy_inputs = inputs[:, :]
    for i in range(T):
        copy_inputs[:, i] = i
    embeddings += self.position_embeddings(copy_inputs)

    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################
    return embeddings

def multi_head_attention(self, inputs, verbose=False):
    """
    :param inputs: float32 Tensor of shape (N,T,H)
    :returns outputs: float32 Tensor of shape (N,T,H)

    Traditionally we'd include a padding mask here, so that pads are ignored.
    This is a simplified implementation.
    """

    outputs = None
    #############################################################################
    # Deliverable 2: Implement multi-head self-attention followed by add + norm.#
    # Use the provided 'Deliverable 2' layers initialized in the constructor.   #
    #############################################################################
    """
    # Head #1
    self.k1 = nn.Linear(self.hidden_dim, self.dim_k)
    self.v1 = nn.Linear(self.hidden_dim, self.dim_v)
    self.q1 = nn.Linear(self.hidden_dim, self.dim_q)
    # Head #2
    self.k2 = nn.Linear(self.hidden_dim, self.dim_k)
    self.v2 = nn.Linear(self.hidden_dim, self.dim_v)
    self.q2 = nn.Linear(self.hidden_dim, self.dim_q)

    self.softmax = nn.Softmax(dim=2)
    self.attention_head_projection = nn.Linear(self.dim_v * self.num_heads, self.hidden_dim)
    self.norm_mh = nn.LayerNorm(self.hidden_dim)
    """
    q1 = self.q1(inputs)
    k1 = self.k1(inputs).permute(0, 2, 1)
    v1 = self.v1(inputs)

    if verbose:
        print(q1.shape)
        print(k1.T.shape)
    x, y, z = q1.shape

    temp = torch.matmul(q1,k1)
    temp = temp / np.sqrt(self.dim_k)
    if verbose: print(f"temp: {temp.shape}")
    a1 = torch.matmul(self.softmax(temp), v1)
    if verbose: print(f"a1: {a1.shape}")

    q2 = self.q2(inputs)
    k2 = self.k2(inputs).permute(0, 2, 1)
    v2 = self.v2(inputs)

    temp = torch.matmul(q2,k2)
    temp = temp / np.sqrt(self.dim_k)
    a2 = torch.matmul(self.softmax(temp), v2)
    if verbose: print(f"a2: {a2.shape}")

    a = torch.cat((a1, a2), dim=2) #this dim=2 is key

    if verbose:
        print(f"attention : {a.shape}")
        print(f"att_head_proj : ({self.dim_v * self.num_heads}, {self.hidden_dim})")
    mh = self.attention_head_projection(a)

    if verbose:
        print(f"mh : {mh.shape}")
        print(f"inputs : {inputs.shape}")
    outputs = self.norm_mh(inputs + mh)
```

```
        ############################################################################
        #                               END OF YOUR CODE                           #
        ############################################################################
        return outputs


    def feedforward_layer(self, inputs):
        """
        :param inputs: float32 Tensor of shape (N,T,H)
        :returns outputs: float32 Tensor of shape (N,T,H)
        """
        outputs = None
        ###########################################################################
        # Deliverable 3: Implement the feedforward layer followed by add + norm.  #
        # Use a ReLU activation and apply the linear layers in the order you      #
        # initialized them.                                                       #
        # This should not take more than 3-5 lines of code.                       #
        ###########################################################################
        outputs = self.W1(inputs)
        outputs = nn.functional.relu(outputs)
        outputs = self.W2(outputs)
        outputs = self.norm_ffn(inputs + outputs)
        ############################################################################
        #                               END OF YOUR CODE                           #
        ############################################################################
        return outputs


    def final_layer(self, inputs):
        """
        :param inputs: float32 Tensor of shape (N,T,H)
        :returns outputs: float32 Tensor of shape (N,1)
        """
        outputs = None
        ###########################################################################
        # Deliverable 4: Implement the final layer for the Transformer classifier. #
        # This should not take more than 2 lines of code.                        #
        ###########################################################################
        N, T, H = inputs.shape
        copy_inputs = inputs[:, 0, :].reshape(N, H) #use CLS token only to make classification. idk why.
        outputs = self.final_dense(copy_inputs)
        outputs = self.final_sigmoid(outputs)

        ############################################################################
        #                               END OF YOUR CODE                           #
        ############################################################################
        return outputs


def seed_torch(seed=0):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.backends.cudnn.benchmark = False
    torch.backends.cudnn.deterministic = True
```