```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 f
inal 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
        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.                 #
        ##############################################################################
        self.embed_layer = nn.Embedding(self.vocab_size + max_length, 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.hidd
en_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.                                      #
        ##############################################################################
        #element-wise feed-forward layer consisting of two linear transformers with a ReL
U layer in between
        self.linear1 = nn.Linear(hidden_dim, dim_feedforward)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(dim_feedforward, hidden_dim)

        #layer normalization
        self.layer_norm = nn.LayerNorm(hidden_dim)
        ##############################################################################
        #                               END OF YOUR CODE                             #
        ##############################################################################


        ##############################################################################
        # Deliverable 4: Initialize what you need for the final layer (1-2 lines).   #
        ##############################################################################
        #linear and softmax layers
        self.linear3 = nn.Linear(hidden_dim, 1)
        self.soft = 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.  #
        #############################################################################
        #1. form embeddings
        embeddings = self.embed(inputs)
        #2. multi head attention
        mha = self.multi_head_attention(embeddings)
        #3. feed forward
        ffw = self.feedforward_layer(mha)
        #4. final layer
        outputs = self.final_layer(ffw)
        ##############################################################################
        #                               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.                                                #
        ##############################################################################
        #token embeddings
        token_embeddings = self.embed_layer(inputs)
        # print(token_embeddings.size)

        #segment embeddings not considered
        #positional embeddings
        encoding = np.arange(self.vocab_size, self.vocab_size + inputs.shape[1])
        encoding = torch.as_tensor(encoding)
        positional_embeddings = self.embed_layer(encoding)

        #add the two embeddings to obtain final embeddings
        embeddings = token_embeddings + positional_embeddings
        ##############################################################################
        #                               END OF YOUR CODE                             #
        ##############################################################################
        return embeddings

    def multi_head_attention(self, inputs):
        """
        :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.   #
        ##############################################################################
        #head1 calculations
        softmax1 = self.softmax((torch.bmm(self.q1(inputs), self.k1(inputs).transpose(1,2
)))/ np.sqrt(self.dim_k))
        attention1 = torch.bmm(softmax1, self.v1(inputs))

        #head2 calculations
        softmax2 = self.softmax((torch.bmm(self.q2(inputs), self.k2(inputs).transpose(1,2
)))/ np.sqrt(self.dim_k))
        attention2 = torch.bmm(softmax2, self.v2(inputs))

        #concatenate the heads
        head = torch.cat((attention1, attention2), 2)

        #residual connection + layer normalzation
        outputs = self.norm_mh(inputs + self.attention_head_projection(head))
        ##############################################################################
        #                               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.                          #
```

```
        ##############################################################################
        #feedforward layer
        h1 = self.linear1(inputs)
        relu1 = self.relu(h1)
        h2 = self.linear2(relu1)

        #layer norm
        outputs = self.layer_norm(h2 + inputs)
        ##############################################################################
        #                               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.                          #
        ##############################################################################
        outputs = self.soft(self.linear3(inputs[:,0,:].squeeze()))
        ##############################################################################
        #                               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
```