import torch

import torch.nn as nn

import torch.optim as optim

import torch.nn.functional as F

class Transformer(nn.Module):

def \_\_init\_\_(self, input\_dim, output\_dim, hid\_dim, n\_layers, n\_heads, pf\_dim, dropout):

super().\_\_init\_\_()

self.encoder\_layers = nn.ModuleList([EncoderLayer(hid\_dim, n\_heads, pf\_dim, dropout) for \_ in range(n\_layers)])

self.fc\_out = nn.Linear(hid\_dim, output\_dim)

self.dropout = nn.Dropout(dropout)

self.input\_dim = input\_dim

self.output\_dim = output\_dim

def forward(self, src):

# src = [batch size, src len, input dim]

for layer in self.encoder\_layers:

src = layer(src)

# src = [batch size, src len, hid dim]

output = self.fc\_out(src[:, -1, :])

return output

class EncoderLayer(nn.Module):

def \_\_init\_\_(self, hid\_dim, n\_heads, pf\_dim, dropout):

super().\_\_init\_\_()

self.self\_attn\_layer\_norm = nn.LayerNorm(hid\_dim)

self.fc\_layer\_norm = nn.LayerNorm(hid\_dim)

self.self\_attention = MultiHeadAttentionLayer(hid\_dim, n\_heads, dropout)

self.positionwise\_feedforward = PositionwiseFeedforwardLayer(hid\_dim, pf\_dim, dropout)

self.dropout = nn.Dropout(dropout)

def forward(self, src):

# src = [batch size, src len, hid dim]

# self attention

\_src, \_ = self.self\_attention(src, src, src) # all source

# dropout, residual connection and layer norm

src = self.self\_attn\_layer\_norm(src + self.dropout(\_src))

# src = [batch size, src len, hid dim]

# positionwise feedforward

\_src = self.positionwise\_feedforward(src)

# dropout, residual and layer norm

src = self.fc\_layer\_norm(src + self.dropout(\_src))

# src = [batch size, src len, hid dim]

return src

class MultiHeadAttentionLayer(nn.Module):

def \_\_init\_\_(self, hid\_dim, n\_heads, dropout):

super().\_\_init\_\_()

assert hid\_dim % n\_heads == 0

self.hid\_dim = hid\_dim

self.n\_heads = n\_heads

self.head\_dim = hid\_dim // n\_heads

self.fc\_q = nn.Linear(hid\_dim, hid\_dim)

self.fc\_k = nn.Linear(hid\_dim, hid\_dim)

self.fc\_v = nn.Linear(hid\_dim, hid\_dim)

self.fc\_o = nn.Linear(hid\_dim, hid\_dim)

self.dropout = nn.Dropout(dropout)

self.scale = torch.sqrt(torch.FloatTensor([self.head\_dim])).to(device)

def forward(self, query, key, value, mask=None):

batch\_size = query.shape[0]

# query = [batch size, query len, hid dim]

# key = [batch size, key len, hid dim]

# value = [batch size, value len, hid dim]

Q = self.fc\_q(query)

K = self.fc\_k(key)

V = self.fc\_v(value)

# Q = [batch size, query len, hid dim]

# K = [batch size, key len, hid dim]

# V = [batch size, value len, hid dim]

Q = Q.view(batch\_size, -1, self.n\_heads, self.head\_dim).permute(0, 2, 1, 3)

K = K.view(batch\_size, -1, self.n\_heads, self.head\_dim).permute(0, 2, 1, 3)

V = V.view(batch\_size, -1, self.n\_heads, self.head\_dim).permute(0, 2, 1, 3)

# Q = [batch size, n heads, query len, head dim]

# K = [batch size, n heads, key len, head dim]

# V = [batch size, n heads, value len, head dim]

energy = torch.matmul(Q, K.permute(0, 1, 3, 2)) / self.scale

# energy = [batch size, n heads, query len, key len]

if mask is not None:

energy = energy.masked\_fill(mask == 0, -1e10)

attention = torch.softmax(energy, dim=-1)

# attention = [batch size, n heads, query len, key len]

x = torch.matmul(self.dropout(attention), V)

# x = [batch size, n heads, query len, head dim]

x = x.permute(0, 2, 1, 3).contiguous()

# x = [batch size, query len, n heads, head dim]

x = x.view(batch\_size, -1, self.hid\_dim)

# x = [batch size, query len, hid dim]

x = self.fc\_o(x)

# x = [batch size, query len, hid dim]

return x, attention

class PositionwiseFeedforwardLayer(nn.Module):

def \_\_init\_\_(self, hid\_dim, pf\_dim, dropout):

super().\_\_init\_\_()

self.fc\_1 = nn.Linear(hid\_dim, pf\_dim)

self.fc\_2 = nn.Linear(pf\_dim, hid\_dim)

self.dropout = nn.Dropout(dropout)

def forward(self, x):

# x = [batch size, seq len, hid dim]

x = self.dropout(torch.relu(self.fc\_1(x)))

# x = [batch size, seq len, pf dim]

x = self.fc\_2(x)

# x = [batch size, seq len, hid dim]

return x