import torch

import torch.nn as nn

import torch.optim as optim

import torchtext

from torchtext.datasets import TranslationDataset, Multi30k

from torchtext.data import Field, BucketIterator

import random

import math

import time

class Encoder(nn.Module):

def \_\_init\_\_(self, input\_dim, emb\_dim, hid\_dim, n\_layers, dropout):

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

self.input\_dim = input\_dim

self.emb\_dim = emb\_dim

self.hid\_dim = hid\_dim

self.n\_layers = n\_layers

# self.dropout = dropout

self.embedding = nn.Embedding(

num\_embeddings=input\_dim,

embedding\_dim=emb\_dim

)

# <YOUR CODE HERE>

self.rnn = nn.LSTM(

input\_size=emb\_dim,

hidden\_size=hid\_dim,

num\_layers=n\_layers,

dropout=dropout

)

# <YOUR CODE HERE>

self.dropout = nn.Dropout(p=dropout)# <YOUR CODE HERE>

def forward(self, src):

#src = [src sent len, batch size]

# Compute an embedding from the src data and apply dropout to it

embedded = self.embedding(src)# <YOUR CODE HERE>

embedded = self.dropout(embedded)

output, (hidden, cell) = self.rnn(embedded)

#embedded = [src sent len, batch size, emb dim]

# Compute the RNN output values of the encoder RNN.

# outputs, hidden and cell should be initialized here. Refer to nn.LSTM docs ;)

# <YOUR CODE HERE>

#outputs = [src sent len, batch size, hid dim \* n directions]

#hidden = [n layers \* n directions, batch size, hid dim]

#cell = [n layers \* n directions, batch size, hid dim]

#outputs are always from the top hidden layer

return hidden, cell

class Decoder(nn.Module):

def \_\_init\_\_(self, output\_dim, emb\_dim, hid\_dim, n\_layers, dropout):

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

self.emb\_dim = emb\_dim

self.hid\_dim = hid\_dim

self.output\_dim = output\_dim

self.n\_layers = n\_layers

self.dropout = dropout

self.embedding = nn.Embedding(

num\_embeddings=output\_dim,

embedding\_dim=emb\_dim

)

# <YOUR CODE HERE>

self.rnn = nn.LSTM(

input\_size=emb\_dim,

hidden\_size=hid\_dim,

num\_layers=n\_layers,

dropout=dropout

)

# <YOUR CODE HERE>

self.out = nn.Linear(

in\_features=hid\_dim,

out\_features=output\_dim

)

# <YOUR CODE HERE>

self.dropout = nn.Dropout(p=dropout)# <YOUR CODE HERE>

def forward(self, input, hidden, cell):

#input = [batch size]

#hidden = [n layers \* n directions, batch size, hid dim]

#cell = [n layers \* n directions, batch size, hid dim]

#n directions in the decoder will both always be 1, therefore:

#hidden = [n layers, batch size, hid dim]

#context = [n layers, batch size, hid dim]

input = input.unsqueeze(0)

#input = [1, batch size]

# Compute an embedding from the input data and apply dropout to it

embedded = self.dropout(self.embedding(input))# <YOUR CODE HERE>

#embedded = [1, batch size, emb dim]

# Compute the RNN output values of the encoder RNN.

# outputs, hidden and cell should be initialized here. Refer to nn.LSTM docs ;)

# <YOUR CODE HERE>

#output = [sent len, batch size, hid dim \* n directions]

#hidden = [n layers \* n directions, batch size, hid dim]

#cell = [n layers \* n directions, batch size, hid dim]

#sent len and n directions will always be 1 in the decoder, therefore:

#output = [1, batch size, hid dim]

#hidden = [n layers, batch size, hid dim]

#cell = [n layers, batch size, hid dim]

output, (hidden, cell) = self.rnn(embedded, (hidden, cell))

prediction = self.out(output.squeeze(0))

#prediction = [batch size, output dim]

return prediction, hidden, cell

class Seq2Seq(nn.Module):

def \_\_init\_\_(self, encoder, decoder, device):

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

self.encoder = encoder

self.decoder = decoder

self.device = device

assert encoder.hid\_dim == decoder.hid\_dim, \

"Hidden dimensions of encoder and decoder must be equal!"

assert encoder.n\_layers == decoder.n\_layers, \

"Encoder and decoder must have equal number of layers!"

def forward(self, src, trg, teacher\_forcing\_ratio = 0.5):

#src = [src sent len, batch size]

#trg = [trg sent len, batch size]

#teacher\_forcing\_ratio is probability to use teacher forcing

#e.g. if teacher\_forcing\_ratio is 0.75 we use ground-truth inputs 75% of the time

# Again, now batch is the first dimention instead of zero

batch\_size = trg.shape[1]

max\_len = trg.shape[0]

trg\_vocab\_size = self.decoder.output\_dim

#tensor to store decoder outputs

outputs = torch.zeros(max\_len, batch\_size, trg\_vocab\_size).to(self.device)

#last hidden state of the encoder is used as the initial hidden state of the decoder

hidden, cell = self.encoder(src)

#first input to the decoder is the <sos> tokens

input = trg[0,:]

for t in range(1, max\_len):

output, hidden, cell = self.decoder(input, hidden, cell)

outputs[t] = output

teacher\_force = random.random() < teacher\_forcing\_ratio

top1 = output.max(1)[1]

input = (trg[t] if teacher\_force else top1)

return outputs