Artificial Intelligence Final Report Assignment 問題3 (Problem 3)

レポート解答用紙 (Report Answer Sheet)

Group Leader

学生証番号 (Student ID): 21520910

名前(Name): Bùi Minh Huy

Group Members

学生証番号 (Student ID): 2152

名前(Name): Lê Tấn Hòa

学生証番号 (Student ID): 21521963

名前(Name): Nguyễn Tiến Đức

問題3 (Problem 3)のレポート

I. Program:

import numpy as np

import matplotlib.pyplot as plt

import torch

import torch.nn.functional as F

import torchtext

import requests

import tarfile

from tqdm import tqdm

url="https://nlp.stanford.edu/projects/nmt/data/iwslt15.en-vi/"

train\_en = [line.split() for line in requests.get(url+"train.en").text.splitlines()]

train\_vi= [line.split() for line in requests.get(url+"train.vi").text.splitlines()]

test\_en = [line.split() for line in requests.get(url+"tst2013.en").text.splitlines()]

test\_vi = [line.split() for line in requests.get(url+"tst2013.vi").text.splitlines()]

EPOCH = 10

BATCHSIZE = 16

LR = 0.0001

DEVICE = 'cuda' if torch.cuda.is\_available() else 'cpu'

import tarfile

def iswlt15(train\_test):

    url = "https://github.com/stefan-it/nmt-en-vi/raw/master/data/"

    r = requests.get(url + train\_test + "-en-vi.tgz")

    filename = train\_test + "-en-vi.tar.gz"

    with open(filename, 'wb') as f:

        f.write(r.content)

        tarfile.open(filename, 'r:gz').extractall("iwslt15")

iswlt15("train")

iswlt15("test-2013")

f = open("iwslt15/train.en")

train\_en = [line.split() for line in f]

f.close()

f = open("iwslt15/train.vi")

train\_vi = [line.split() for line in f]

f.close()

f = open("iwslt15/tst2013.en")

test\_en = [line.split() for line in f]

f.close()

f = open("iwslt15/tst2013.vi")

test\_vi = [line.split() for line in f]

f.close()

for i in range(10):

  print(train\_en[i])

  print(train\_vi[i])

print('# of line', len(train\_en), len(train\_vi), len(test\_en), len(test\_vi))

def make\_vocab(train\_data, min\_freq):

  vocab = {}

  for tokenlist in train\_data:

    for token in tokenlist:

      if token not in vocab:

        vocab[token] = 0

      vocab[token] += 1

  vocablist = [('<unk>',0),('<pad>', 0), ('<cls>', 0), ('<eos>', 0)]

  vocabidx = {}

  for token, freq in vocab.items():

    if freq >= min\_freq:

      idx = len(vocablist)

      vocablist.append((token, freq))

      vocabidx[token] = idx

  vocabidx['<unk>'] = 0

  vocabidx['<pad>'] = 1

  vocabidx['<cls>'] = 2

  vocabidx['<eos>'] = 3

  return vocablist, vocabidx

vocablist\_en, vocabidx\_en = make\_vocab(train\_en, 3)

vocablist\_vi, vocabidx\_vi = make\_vocab(train\_vi, 3)

print('vocab size en: ', len(vocablist\_en))

print('vocab size vi: ', len(vocablist\_vi))

def preprocess(data, vocabidx):

  rr = []

  for tokenlist in data:

    tkl =['<cls>']

    for token in tokenlist:

      tkl.append(token if token in vocabidx else '<unk>')

    tkl.append('<eos>')

    rr.append((tkl))

  return rr

train\_en\_prep = preprocess(train\_en, vocabidx\_en)

train\_vi\_prep = preprocess(train\_vi, vocabidx\_vi)

test\_en\_prep = preprocess(test\_en, vocabidx\_en)

for i in range(5):

  print(train\_en\_prep[i])

  print(train\_vi\_prep[i])

  print(test\_en\_prep[i])

train\_data = list(zip(train\_en\_prep, train\_vi\_prep))

train\_data.sort(key = lambda x: (len(x[0]), len(x[1])))

test\_data = list(zip(test\_en\_prep, test\_en, test\_vi))

for i in range(10):

  print(train\_data[i])

for i in range(10):

  print(test\_data[i])

def make\_batch(data, batchsize):

  bb = []

  ben = []

  bvi = []

  for en,vi in data:

    ben.append(en)

    bvi.append(vi)

    if len(ben) >= batchsize:

      bb.append((ben, bvi))

      ben = []

      bvi = []

  if len(ben) > 0:

    bb.append((ben, bvi))

  return bb

train\_data = make\_batch(train\_data, BATCHSIZE)

for i in range(2):

  print(train\_data[i])

def padding\_batch(b):

  maxlen = max([len(x) for x in b])

  for tkl in b:

    for i in range(maxlen - len(tkl)):

      tkl.append('<pad>')

def padding(bb):

  for ben, bvi in bb:

    padding\_batch(ben)

    padding\_batch(bvi)

padding(train\_data)

for i in range(3):

  print(train\_data[i])

train\_data = [([[vocabidx\_en[token] for token in tokenlist] for tokenlist in ben],

               [[vocabidx\_vi[token] for token in tokenlist] for tokenlist in bvi]) for ben, bvi in train\_data]

test\_data = [([vocabidx\_en[token] for token in enprep], en, vi) for enprep, en, vi in test\_data]

for i in range(3):

  print(train\_data[i])

  print(test\_data[i])

from torch import Tensor

from torch import nn

from torch.nn import Transformer

import math

class PositionalEncoding(nn.Module):

    def \_\_init\_\_(self,

                 emb\_size: int,

                 dropout: float,

                 maxlen: int = 900):

        super(PositionalEncoding, self).\_\_init\_\_()

        den = torch.exp(- torch.arange(0, emb\_size, 2)\* math.log(10000) / emb\_size)

        pos = torch.arange(0, maxlen).reshape(maxlen, 1)

        pos\_embedding = torch.zeros((maxlen, emb\_size))

        pos\_embedding[:, 0::2] = torch.sin(pos \* den)

        pos\_embedding[:, 1::2] = torch.cos(pos \* den)

        pos\_embedding = pos\_embedding.unsqueeze(-2)

        self.dropout = nn.Dropout(dropout)

        self.register\_buffer('pos\_embedding', pos\_embedding)

    def forward(self, token\_embedding: Tensor):

        return self.dropout(token\_embedding + self.pos\_embedding[:token\_embedding.size(0), :])

class TokenEmbedding(nn.Module):

    def \_\_init\_\_(self, vocab\_size: int, emb\_size):

        super(TokenEmbedding, self).\_\_init\_\_()

        self.embedding = nn.Embedding(vocab\_size, emb\_size)

        self.emb\_size = emb\_size

    def forward(self, tokens: Tensor):

        return self.embedding(tokens.long()) \* math.sqrt(self.emb\_size)

class Seq2SeqTransformer(nn.Module):

    def \_\_init\_\_(self,

                 num\_encoder\_layers: int,

                 num\_decoder\_layers: int,

                 emb\_size: int,

                 nhead: int,

                 src\_vocab\_size: int,

                 tgt\_vocab\_size: int,

                 dim\_feedforward: int = 512,

                 dropout: float = 0.01):

        super(Seq2SeqTransformer, self).\_\_init\_\_()

        self.transformer = Transformer(d\_model=emb\_size,

                                       nhead=nhead,

                                       num\_encoder\_layers=num\_encoder\_layers,

                                       num\_decoder\_layers=num\_decoder\_layers,

                                       dim\_feedforward=dim\_feedforward,

                                       dropout=dropout)

        self.generator = nn.Linear(emb\_size, tgt\_vocab\_size)

        self.src\_tok\_emb = TokenEmbedding(src\_vocab\_size, emb\_size)

        self.tgt\_tok\_emb = TokenEmbedding(tgt\_vocab\_size, emb\_size)

        self.positional\_encoding = PositionalEncoding(

            emb\_size, dropout=dropout)

    def forward(self,

                src: Tensor,

                trg: Tensor,

                src\_mask: Tensor,

                tgt\_mask: Tensor,

                src\_padding\_mask: Tensor,

                tgt\_padding\_mask: Tensor,

                memory\_key\_padding\_mask: Tensor):

        src\_emb = self.positional\_encoding(self.src\_tok\_emb(src))

        tgt\_emb = self.positional\_encoding(self.tgt\_tok\_emb(trg))

        outs = self.transformer(src\_emb, tgt\_emb, src\_mask, tgt\_mask, None,

                                src\_padding\_mask, tgt\_padding\_mask, memory\_key\_padding\_mask)

        return self.generator(outs)

    def encode(self, src: Tensor, src\_mask: Tensor):

        return self.transformer.encoder(self.positional\_encoding(

                            self.src\_tok\_emb(src)), src\_mask)

    def decode(self, tgt: Tensor, memory: Tensor, tgt\_mask: Tensor):

        return self.transformer.decoder(self.positional\_encoding(

                          self.tgt\_tok\_emb(tgt)), memory,

                          tgt\_mask)

import time

torch.manual\_seed(0)

MODELNAME = 'transfomers.model'

SRC\_VOCAB\_SIZE = len(vocablist\_en)

TGT\_VOCAB\_SIZE = len(vocablist\_vi)

EMB\_SIZE = 512

NHEAD = 8

FFN\_HID\_DIM = 512

NUM\_ENCODER\_LAYERS = 3

NUM\_DECODER\_LAYERS = 3

model = Seq2SeqTransformer(NUM\_ENCODER\_LAYERS, NUM\_DECODER\_LAYERS, EMB\_SIZE,

                                 NHEAD, SRC\_VOCAB\_SIZE, TGT\_VOCAB\_SIZE, FFN\_HID\_DIM)

for p in model.parameters():

    if p.dim() > 1:

        nn.init.xavier\_uniform\_(p)

model = model.to(DEVICE)

PAD\_IDX= 1

loss\_fn = torch.nn.CrossEntropyLoss(ignore\_index=PAD\_IDX)

def generate\_square\_subsequent\_mask(sz):

    mask = (torch.triu(torch.ones((sz, sz), device=DEVICE)) == 1).transpose(0, 1)

    mask = mask.float().masked\_fill(mask == 0, float('-inf')).masked\_fill(mask == 1, float(0.0))

    return mask

def create\_mask(src, tgt):

    src\_seq\_len = src.shape[0]

    tgt\_seq\_len = tgt.shape[0]

    tgt\_mask = generate\_square\_subsequent\_mask(tgt\_seq\_len)

    src\_mask = torch.zeros((src\_seq\_len, src\_seq\_len),device=DEVICE).type(torch.bool)

    src\_padding\_mask = (src == vocabidx\_en['<pad>']).transpose(0, 1)

    tgt\_padding\_mask = (tgt == vocabidx\_vi['<pad>']).transpose(0, 1)

    return src\_mask, tgt\_mask, src\_padding\_mask, tgt\_padding\_mask

def train():

  optimizer = torch.optim.Adam(model.parameters(), lr=0.0001, betas=(0.9, 0.98), eps=1e-9)

  model.train()

  all\_time= 0

  for epoch in range(10):

    start= time.time()

    loss = 0

    for en, vi in train\_data:

      en = torch.tensor(en, dtype=torch.int64).transpose(0,1).to(DEVICE)

      vi = torch.tensor(vi, dtype=torch.int64).transpose(0,1).to(DEVICE)

      tgt\_input = vi[:-1, :]

      optimizer.zero\_grad()

      src\_mask, tgt\_mask, src\_padding\_mask, tgt\_padding\_mask = create\_mask(en, tgt\_input)

      y = model(en, tgt\_input, src\_mask, tgt\_mask, src\_padding\_mask, tgt\_padding\_mask, src\_padding\_mask)

      tgt\_out = vi[1:, :]

      batchloss = loss\_fn(y.reshape(-1, y.shape[-1]), tgt\_out.reshape(-1))

      batchloss.backward()

      optimizer.step()

      loss = loss + batchloss.item()

    end= time.time()

    all\_time+=(end-start)

    print("epoch", epoch, ": loss", loss, "Epoch time",end-start)

  torch.save(model.state\_dict(), MODELNAME)

def evaluate(model, src, max\_len, start\_symbol):

    num\_tokens = src.shape[0]

    src\_mask = torch.zeros((num\_tokens, num\_tokens),device=DEVICE).type(torch.bool)

    memory = model.encode(src, src\_mask)

    ys = torch.ones(1, 1).fill\_(start\_symbol).type(torch.long).to(DEVICE)

    pred = []

    for i in range(max\_len-1):

        memory = memory.to(DEVICE)

        tgt\_mask = (generate\_square\_subsequent\_mask(ys.size(0)).type(torch.bool)).to(DEVICE)

        out = model.decode(ys, memory, tgt\_mask)

        out = out.transpose(0, 1)

        prob = model.generator(out[:, -1])

        \_, next\_word = torch.max(prob, dim=1)

        next\_word = next\_word.item()

        ys = torch.cat([ys,

                        torch.ones(1, 1).type\_as(src.data).fill\_(next\_word)], dim=0)

        if next\_word == vocabidx\_en['<eos>']:

            break

        pred\_y = vocablist\_vi[next\_word][0]

        pred.append(pred\_y)

    return pred

train()

import torchtext.data.metrics

def test():

  model = Seq2SeqTransformer(NUM\_ENCODER\_LAYERS, NUM\_DECODER\_LAYERS, EMB\_SIZE,

                                 NHEAD, SRC\_VOCAB\_SIZE, TGT\_VOCAB\_SIZE, FFN\_HID\_DIM).to(DEVICE)

  model.load\_state\_dict(torch.load(MODELNAME))

  model.eval()

  ref = []

  pred = []

  for enprep, en, vi in test\_data:

    input = torch.tensor([enprep], dtype=torch.int64).transpose(0,1).to(DEVICE).view(-1, 1)

    p = evaluate(model, input, 50, vocabidx\_en['<cls>'])

    ref.append([vi])

    pred.append(p)

  bleu = torchtext.data.metrics.bleu\_score(pred, ref)

  print("total: ", len(test\_data))

  print("bleu: ", bleu)

test()

II. Execution Results

A screenshot of a computer screen

Description automatically generated A number on a black background

Description automatically generated

III. Explanation

I made several improvements to increase the accuracy of the program. First, I added functions to download and extract the dataset efficiently. I created vocabularies with special tokens <unk>, <pad>, <cls>, and <eos> for better handling during training and inference. The preprocessing and batching functions were optimized to ensure consistency and efficiency. I replaced the RNN model with a Transformer model (Seq2SeqTransformer) to enhance the ability to capture long-term dependencies. Positional encoding was implemented to provide positional information to the model. Lastly, I refined the training and evaluation loops to include optimizations and detailed loss calculations, ensuring efficient training and accurate evaluation. These improvements enhance the model's performance, efficiency, and accuracy in this tasks.