Artificial Intelligence Lab Work (6)

レポート解答用紙 (Report Answer Sheet)

学生証番号 (Student ID): 18521071

名前(Name):ズオン・ミン・ルオン (Duong Minh Luong)

問題1.

|  |
| --- |
| (プログラム)  import tarfile  def iwlt15(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")  iwslt15("train")  iwslt15("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()  import requests  import torch  import torch.nn.functional as F  import torchtext  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()]  for i in range(10):  print(train\_en[i])  print(train\_vi[i])  print("# line", len(train\_en), len(train\_vi), len(test\_en), len(test\_vi))  MODELNAME = 'iwslt15-en-vi-rnn.model'  EPOCH = 10  BATCHSIZE = 128  LR = 0.0001  DEVICE = 'cuda' if torch.cuda.is\_available() else 'cpu'  """##1. Data preparation  ###Make vocab  """  def make\_vocab(train\_data, min\_freq):  vocab = {} # tap tu vung  for tokenlist in train\_data:  for token in tokenlist:  #辞書を使ってトークンの出現回数をカウント  if token not in vocab:  vocab[token] = 0  vocab[token] += 1  #語彙リストの0〜3番⽬を<unk>,<pad>, <cls>, <eos>で予約  vocablist = [('<unk>', 0), ('<pad>', 0), ('<cls>', 0), ('<eos>', 0)]  vocabidx = {}  for token, freq in vocab.items():  if freq >= min\_freq:  idx = len(vocablist)  #min\_freq以上の出現回数のトークンだけ語彙リスト(vocablist)と語彙インデックス(vocabidx)に登録  vocablist.append((token, freq))  vocabidx[token] = idx  #<unk>,<pad>,<cls>,<eos>のインデックス登録  vocabidx['<unk>'] = 0  vocabidx['<pad>'] = 1  vocabidx['<cls>'] = 2  vocabidx['<eos>'] = 3  return vocablist, vocabidx  #今回はmin\_freqを3に設定  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))  """###Pre-proces"""  def preprocess(data, vocabidx):  rr = []  for tokenlist in data:  #テキストの先頭に<cls>トークンを追加  tkl = ['<cls>']  for token in tokenlist:  #トークンの追加。語彙リストに無いトークンは<unk>に変換  tkl.append(token if token in vocabidx else '<unk>')  #テキストの末尾に<eos>トークンを追加  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)  #この部分はあっても無くても良い  for i in range(5):  print(train\_en\_prep[i])  print(train\_vi\_prep[i])  print(test\_en\_prep[i])  #訓練データのzip化(前処理済en, 前処理済vi)  train\_data = list(zip(train\_en\_prep, train\_vi\_prep))  #訓練データのソーティング  train\_data.sort(key = lambda x: (len(x[0]), len(x[1])))  #テストデータのZip化(前処理済en, en, vi)  test\_data = list(zip(test\_en\_prep, test\_en, test\_vi))  #この部分はあっても無くても良い  for i in range(5):  print(train\_data[i])  for i in range(5):  print(test\_data[i])  """###Make batch"""  def make\_batch(data, batchsize):  bb = []  ben = []  bvi = []  for en, vi in data:  #英語⽂だけbenにまとめ、ベトナム語⽂だけbviにまとめる  ben.append(en)  bvi.append(vi)  if len(ben) >= batchsize:  #バッチサイズと同じ⼤きさになったら、たまったバッチデータをbbに追加  bb.append((ben, bvi))  ben = []  bvi = []  if len(ben) > 0:  #残ったbenと bviを忘れずにbbに追加  bb.append((ben, bvi))  return bb  train\_data = make\_batch(train\_data, BATCHSIZE)  for i in range(5):  print(train\_data[i])  """###Padding batch"""  def padding\_batch(b):  #ミニバッチ内で⼀番⻑いトークン列の⻑さを得る  maxlen = max([len(x) for x in b])  for tokenlists in b:  for i in range(maxlen - len(tokenlists)):  #最⼤⻑と同じ⻑さになるように<pad>トークンの追加を繰り返す  tokenlists.append('<pad>')  return b  def padding(bb):  for ben, bvi in bb:  #英語バッチとベトナム語バッチと両⽅にパディングをする  ben = padding\_batch(ben)  bvi = padding\_batch(bvi)  return bb  padding(train\_data)  for i in range(3):  print(train\_data[i])  """###Encoding"""  #語彙インデックスを使ってトークンをID化  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])  for i in range(3):  print(test\_data[i])  """##2. RNN Model"""  class RNNEncDec(torch.nn.Module):  def \_\_init\_\_(self, vocablist\_x, vocabidx\_x, vocablist\_y, vocabidx\_y):  super(RNNEncDec, self).\_\_init\_\_()  #encemb: エンコーダーの畳み込み層(300次元)  self.encemb = torch.nn.Embedding(len(vocablist\_x), 300, padding\_idx = vocabidx\_x['<pad>'])  #encrnn: エンコーダーのRNN計算ユニット(300✕300のFC層)  self.encrnn = torch.nn.Linear(300, 300)  #decemb: デコーダーの畳み込み層(300次元)  self.decemb = torch.nn.Embedding(len(vocablist\_x), 300, padding\_idx = vocabidx\_y['<pad>'])  #decrnn: デコーダーのRNN計算ユニット(300✕300のFC層)  self.decrnn = torch.nn.Linear(300, 300)  #decout: デコーダーの出力(300✕目的言語の語彙数)  self.decout = torch.nn.Linear(300, len(vocabidx\_y))    def forward(self, x):  #x: input (⽂⻑✕バッチサイズ)  #y:output (⽂⻑✕バッチサイズ)  x, y = x[0], x[1]  # encoder  e\_x = self.encemb(x)  n\_x = e\_x.size()[0]  h = torch.zeros(300, dtype = torch.float32).to(DEVICE)  for i in range(n\_x):  h = F.relu(e\_x[i] + self.encrnn(h))  # decoder  e\_y = self.decemb(y)  #n\_y=⽂⻑(J)+2 (<cls>と <eos>)  n\_y = e\_y.size()[0]  loss = torch.tensor(0, dtype = torch.float32).to(DEVICE)  for i in range(n\_y - 1):  #⼊⼒はi=0から Jまで  h = F.relu(e\_y[i] + self.decrnn(h))  #出⼒はi=1から J+1まで  loss += F.cross\_entropy(self.decout(h), y[i+1])  return loss  def evaluate(self, x, vocablist\_y, vocabidx\_y):  # encoder  #推論は1⽂ずつ⾏うので、 xには⽂⻑✕バッチサイズ1のミニバッチが⼊っている。  e\_x = self.encemb(x)  n\_x = e\_x.size()[0]  #エンコーダー部はforwardとほぼ同じ。  h = torch.zeros(300, dtype = torch.float32).to(DEVICE)  for i in range(n\_x):  h = F.relu(e\_x[i] + self.encrnn(h))  # decoder  #デコーダーの⼊⼒(バッチサイズ1)を作る。最初は<cls>トークンを⼊⼒する  y = torch.tensor([vocabidx\_y['<cls>']]).to(DEVICE)  e\_y = self.decemb(y)  pred = []  for i in range(30):  h = F.relu(e\_y + self.decrnn(h))  pred\_id = self.decout(h).squeeze().argmax()  #pred\_idが予測する出⼒単語ID pred\_idが<eos>のIDと等しければ推論終了  if pred\_id == vocabidx\_y['<eos>']:  break  pred\_y = vocablist\_y[pred\_id][0]  pred.append(pred\_y)  #デコーダーは1単語ずつ処理をし、得られた出⼒を次の⼊⼒とする  y[0] = pred\_id  e\_y = self.decemb(y)    return pred  def train():  model = RNNEncDec(vocablist\_en, vocabidx\_en, vocablist\_vi, vocabidx\_vi).to(DEVICE)  optimizer = torch.optim.Adam(model.parameters(), lr = LR)  for epoch in range(EPOCH):  loss = 0  step = 0  for ben, bvi in train\_data:  ben = torch.tensor(ben, dtype = torch.int64).transpose(0,1).to(DEVICE)  bvi = torch.tensor(bvi, dtype = torch.int64).transpose(0,1).to(DEVICE)  optimizer.zero\_grad()  batchloss = model((ben, bvi))  batchloss.backward()  optimizer.step()  loss = loss + batchloss.item()  if step % 100 == 0:  print("step {}, batchloss = {}".format(step, batchloss.item()))  step += 1  print("Epoch {} with loss = {}".format(epoch, loss))  torch.save(model.state\_dict(), MODELNAME)  def test():  total = 0  correct = 0  model = RNNEncDec(vocablist\_en, vocabidx\_en, vocablist\_vi, vocabidx\_vi).to(DEVICE)  model.load\_state\_dict(torch.load(MODELNAME))  model.eval()  ref = []  pred = []  #テストデータはミニバッチ化されていないので、 1⽂ずつ処理をする。 (enprep,en, vi)にそれぞれ  #前処理済み英⽂、英⽂、ベトナム語⽂がはいっている  for enprep, en, vi in test\_data:  input = torch.tensor([enprep], dtype = torch.int64).transpose(0,1).to(DEVICE)  p = model.evaluate(input, vocablist\_vi, vocabidx\_vi)  print("INPUT: ", en)  print("REF: ", vi)  print("MT:", p)  ref.append([vi])  pred.append(p)  bleu = torchtext.data.metrics.bleu\_score(pred, ref)  print("total: {}".format(len(test\_data)))  print("BLEU = {}".format(bleu))  train()  test() |

|  |
| --- |
| (実行結果) |

問題2.

|  |
| --- |
| (プログラム)  import requests  import torch  import torch.nn.functional as F  import torchtext  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()]  for i in range(10):  print(train\_en[i])  print(train\_vi[i])  print("# line", len(train\_en), len(train\_vi), len(test\_en), len(test\_vi))  MODELNAME = 'LSTM\_Dropout-translation'  EPOCH = 10  BATCHSIZE = 128  LR = 0.001  DEVICE = 'cuda' if torch.cuda.is\_available() else 'cpu'  """##1. Data preparation  ###Make vocab  """  def make\_vocab(train\_data, min\_freq):  vocab = {} # tap tu vung  for tokenlist in train\_data:  for token in tokenlist:  #辞書を使ってトークンの出現回数をカウント  if token not in vocab:  vocab[token] = 0  vocab[token] += 1  #語彙リストの0〜3番⽬を<unk>,<pad>, <cls>, <eos>で予約  vocablist = [('<unk>', 0), ('<pad>', 0), ('<cls>', 0), ('<eos>', 0)]  vocabidx = {}  for token, freq in vocab.items():  if freq >= min\_freq:  idx = len(vocablist)  #min\_freq以上の出現回数のトークンだけ語彙リスト(vocablist)と語彙インデックス(vocabidx)に登録  vocablist.append((token, freq))  vocabidx[token] = idx  #<unk>,<pad>,<cls>,<eos>のインデックス登録  vocabidx['<unk>'] = 0  vocabidx['<pad>'] = 1  vocabidx['<cls>'] = 2  vocabidx['<eos>'] = 3  return vocablist, vocabidx  #今回はmin\_freqを3に設定  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))  """###Pre-proces"""  def preprocess(data, vocabidx):  rr = []  for tokenlist in data:  #テキストの先頭に<cls>トークンを追加  tkl = ['<cls>']  for token in tokenlist:  #トークンの追加。語彙リストに無いトークンは<unk>に変換  tkl.append(token if token in vocabidx else '<unk>')  #テキストの末尾に<eos>トークンを追加  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)  #この部分はあっても無くても良い  for i in range(5):  print(train\_en\_prep[i])  print(train\_vi\_prep[i])  print(test\_en\_prep[i])  #訓練データのzip化(前処理済en, 前処理済vi)  train\_data = list(zip(train\_en\_prep, train\_vi\_prep))  #訓練データのソーティング  train\_data.sort(key = lambda x: (len(x[0]), len(x[1])))  #テストデータのZip化(前処理済en, en, vi)  test\_data = list(zip(test\_en\_prep, test\_en, test\_vi))  #この部分はあっても無くても良い  for i in range(5):  print(train\_data[i])  for i in range(5):  print(test\_data[i])  """##Make batch"""  def make\_batch(data, batchsize):  bb = []  ben = []  bvi = []  for en, vi in data:  #英語⽂だけbenにまとめ、ベトナム語⽂だけbviにまとめる  ben.append(en)  bvi.append(vi)  if len(ben) >= batchsize:  #バッチサイズと同じ⼤きさになったら、たまったバッチデータをbbに追加  bb.append((ben, bvi))  ben = []  bvi = []  if len(ben) > 0:  #残ったbenと bviを忘れずにbbに追加  bb.append((ben, bvi))  return bb  train\_data = make\_batch(train\_data, BATCHSIZE)  for i in range(5):  print(train\_data[i])  """##Padding batch"""  def padding\_batch(b):  #ミニバッチ内で⼀番⻑いトークン列の⻑さを得る  maxlen = max([len(x) for x in b])  for tokenlists in b:  for i in range(maxlen - len(tokenlists)):  #最⼤⻑と同じ⻑さになるように<pad>トークンの追加を繰り返す  tokenlists.append('<pad>')  return b  def padding(bb):  for ben, bvi in bb:  #英語バッチとベトナム語バッチと両⽅にパディングをする  ben = padding\_batch(ben)  bvi = padding\_batch(bvi)  return bb  train\_data\_pd = padding(train\_data)  for i in range(3):  print(train\_data\_pd[i])  """##Encoding"""  #語彙インデックスを使ってトークンをID化  train\_data\_encoding=[([[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\_encoding=[([vocabidx\_en[token] for token in enprep],en ,vi) for enprep, en, vi in test\_data]  for i in range(3):  print(train\_data\_encoding[i])  print(test\_data\_encoding[i])  """##2. LSTM + dropout"""  class LSTM(torch.nn.Module):  def \_\_init\_\_(self, vocablist\_x, vocabidx\_x, vocablist\_y, vocabidx\_y):  super(LSTM, self).\_\_init\_\_()  self.encemb = torch.nn.Embedding(len(vocablist\_x), 256, padding\_idx = vocabidx\_x['<pad>'])  self.dropout = torch.nn.Dropout(0.5)  self.enclstm = torch.nn.LSTM(256,516,2,dropout=0.5)  self.decemb = torch.nn.Embedding(len(vocablist\_x), 256, padding\_idx = vocabidx\_y['<pad>'])  self.declstm = torch.nn.LSTM(256,516,2,dropout=0.5)  self.decout = torch.nn.Linear(516, len(vocabidx\_y))    def forward(self,x):  x, y = x[0], x[1]  e\_x = self.dropout(self.encemb(x))    outenc,(hidden,cell) = self.enclstm(e\_x)  n\_y=y.shape[0]  outputs = torch.zeros(n\_y,BATCHSIZE,len(vocablist\_vi)).to(DEVICE)  loss = torch.tensor(0.,dtype=torch.float32).to(DEVICE)  for i in range(n\_y-1):  input = y[i]  input = input.unsqueeze(0)  input = self.dropout(self.decemb(input))  outdec, (hidden,cell) = self.declstm(input,(hidden,cell))  output = self.decout(outdec.squeeze(0))  input = y[i+1]  loss += F.cross\_entropy(output, y[i+1])  return loss  def evaluate(self,x,vocablist\_y,vocabidx\_y):  e\_x = self.dropout(self.encemb(x))  outenc,(hidden,cell)=self.enclstm(e\_x)    y = torch.tensor([vocabidx\_y['<cls>']]).to(DEVICE)  pred=[]  for i in range(30):  input = y  input = input.unsqueeze(0)  input = self.dropout(self.decemb(input))  outdec,(hidden,cell)= self.declstm(input,(hidden,cell))  output = self.decout(outdec.squeeze(0))  pred\_id = output.squeeze().argmax().item()  if pred\_id == vocabidx\_y['<eos>']:  break  pred\_y = vocablist\_y[pred\_id][0]  pred.append(pred\_y)  y[0]=pred\_id  input=y  return pred  def train():  model = LSTM(vocablist\_en, vocabidx\_en, vocablist\_vi, vocabidx\_vi).to(DEVICE)  optimizer = torch.optim.Adam(model.parameters(), lr = LR)  for epoch in range(EPOCH):  loss = 0  step = 0  for ben, bvi in train\_data\_encoding:  ben = torch.tensor(ben, dtype = torch.int64).transpose(0,1).to(DEVICE)  bvi = torch.tensor(bvi, dtype = torch.int64).transpose(0,1).to(DEVICE)  optimizer.zero\_grad()  batchloss = model((ben, bvi))  batchloss.backward()  optimizer.step()  loss = loss + batchloss.item()  if step % 100 == 0:  print("step {}, batchloss = {}".format(step, batchloss.item()))  step += 1  print("Epoch {} with loss = {}".format(epoch, loss))  torch.save(model.state\_dict(), MODELNAME)  def test():  total = 0  correct = 0  model = LSTM(vocablist\_en, vocabidx\_en, vocablist\_vi, vocabidx\_vi).to(DEVICE)  model.load\_state\_dict(torch.load(MODELNAME))  model.eval()  ref = []  pred = []    for enprep, en, vi in test\_data\_encoding:  input = torch.tensor([enprep], dtype = torch.int64).transpose(0,1).to(DEVICE)  p = model.evaluate(input, vocablist\_vi, vocabidx\_vi)  print("INPUT: ", en)  print("REF: ", vi)  print("MT:", p)  ref.append([vi])  pred.append(p)  bleu = torchtext.data.metrics.bleu\_score(pred, ref)  print("total: {}".format(len(test\_data)))  print("BLEU = {}".format(bleu))  train()  test() |

|  |
| --- |
| (実行結果) |