Artificial Intelligence Final Report Assignment 問題2 (Problem 2)

レポート解答用紙 (Report Answer Sheet)

Group Leader

学生証番号 (Student ID):

名前(Name):

Group Members

学生証番号 (Student ID):

名前(Name):

学生証番号 (Student ID):

名前(Name):

問題2 (Problem 2)のレポート

I. Program:

!pip install torchtext==0.17.0

!pip install portalocker

import torch

import torch.nn.functional as F

import torchtext

from torchtext.data.utils import get\_tokenizer

# Load the IMDB dataset

train\_iter, test\_iter = torchtext.datasets.IMDB(split=('train', 'test'))

tokenizer = get\_tokenizer('basic\_english')

# Hyperparameters

MODEL\_NAME = 'imdb-bilstm.model'

EPOCHS = 10

BATCH\_SIZE = 64

LEARNING\_RATE = 1e-4

EMBEDDING\_DIM = 300

HIDDEN\_DIM = 128

DROPOUT = 0.5

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

print(DEVICE)

# Tokenize and sort the data

train\_data = [(label, tokenizer(line)) for label, line in train\_iter]

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

test\_data = [(label, tokenizer(line)) for label, line in test\_iter]

test\_data.sort(key=lambda x: len(x[1]))

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

    vocab = {}

    for label, 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>', 3)]

    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, vocabidx = make\_vocab(train\_data, 10)

def preprocess(data, vocabidx):

    rr = []

    for label, tokenlist in data:

        tkl = ['<cls>']

        for token in tokenlist:

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

        tkl.append('<eos>')

        rr.append((label, tkl))

    return rr

train\_data = preprocess(train\_data, vocabidx)

test\_data = preprocess(test\_data, vocabidx)

def make\_batch(data, batchsize):

    bb = []

    blabel = []

    btokenlist = []

    for label, tokenlist in data:

        blabel.append(label)

        btokenlist.append(tokenlist)

        if len(blabel) >= batchsize:

            bb.append((btokenlist, blabel))

            blabel = []

            btokenlist = []

    if len(blabel) > 0:

        bb.append((btokenlist, blabel))

    return bb

train\_data = make\_batch(train\_data, BATCH\_SIZE)

test\_data = make\_batch(test\_data, BATCH\_SIZE)

def padding(bb):

    for tokenlists, labels in bb:

        maxlen = max([len(x) for x in tokenlists])

        for tkl in tokenlists:

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

                tkl.append('<pad>')

    return bb

train\_data = padding(train\_data)

test\_data = padding(test\_data)

def word2id(bb, vocbidx):

    rr = []

    for tokenlists, labels in bb:

        id\_labels = [label - 1 for label in labels]

        id\_tokenlists = []

        for tokenlist in tokenlists:

            id\_tokenlists.append([vocabidx[token] for token in tokenlist])

        rr.append((id\_tokenlists, id\_labels))

    return rr

train\_data = word2id(train\_data, vocabidx)

test\_data = word2id(test\_data, vocabidx)

# Load GloVe embeddings

glove\_vectors = torchtext.vocab.GloVe(name='6B', dim=EMBEDDING\_DIM)

def build\_embedding\_matrix(vocab, glove\_vectors, embedding\_dim):

    embedding\_matrix = torch.zeros((len(vocab), embedding\_dim))

    for word, idx in vocabidx.items():

        if word in glove\_vectors.stoi:

            embedding\_matrix[idx] = glove\_vectors[word]

        else:

            embedding\_matrix[idx] = torch.randn(embedding\_dim)

    return embedding\_matrix

embedding\_matrix = build\_embedding\_matrix(vocablist, glove\_vectors, EMBEDDING\_DIM)

# Model definition

class MyBiLSTM(torch.nn.Module):

    def \_\_init\_\_(self):

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

        vocab\_size = len(vocablist)

        self.embedding = torch.nn.Embedding.from\_pretrained(embedding\_matrix, freeze=False, padding\_idx=vocabidx['<pad>'])

        self.lstm = torch.nn.LSTM(EMBEDDING\_DIM, HIDDEN\_DIM, batch\_first=True, bidirectional=True)

        self.dropout = torch.nn.Dropout(DROPOUT)

        self.fc = torch.nn.Linear(HIDDEN\_DIM \* 2, 2)

    def forward(self, x):

        embedded = self.embedding(x)

        lstm\_out, (h\_n, c\_n) = self.lstm(embedded)

        lstm\_out = self.dropout(lstm\_out[:, -1, :])

        return self.fc(lstm\_out)

# Training function

def train():

    model = MyBiLSTM().to(DEVICE)

    optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING\_RATE)

    for epoch in range(EPOCHS):

        total\_loss = 0

        for tokenlists, labels in train\_data:

            tokenlists = torch.tensor(tokenlists, dtype=torch.int64).to(DEVICE)

            labels = torch.tensor(labels, dtype=torch.int64).to(DEVICE)

            optimizer.zero\_grad()

            outputs = model(tokenlists)

            loss = F.cross\_entropy(outputs, labels)

            loss.backward()

            optimizer.step()

            total\_loss += loss.item()

        print(f"Epoch {epoch+1}, Loss: {total\_loss:.4f}")

    torch.save(model.state\_dict(), MODEL\_NAME)

# Testing function

def test():

    model = MyBiLSTM().to(DEVICE)

    model.load\_state\_dict(torch.load(MODEL\_NAME))

    model.eval()

    total = 0

    correct = 0

    with torch.no\_grad():

        for tokenlists, labels in test\_data:

            tokenlists = torch.tensor(tokenlists, dtype=torch.int64).to(DEVICE)

            labels = torch.tensor(labels, dtype=torch.int64).to(DEVICE)

            outputs = model(tokenlists)

            \_, predicted = torch.max(outputs, 1)

            total += labels.size(0)

            correct += (predicted == labels).sum().item()

    print(f"Accuracy: {correct / total:.4f}")

# Run the training and testing

train()

test()

II. Execution Results

A computer screen with white text

Description automatically generated

III. Explanation

* Bidirectional LSTM: The LSTM layer is now bidirectional, which allows the model to capture dependencies from both past and future states.
* Pretrained Embeddings: The GloVe embeddings are used to initialize the embedding layer, leveraging pretrained word vectors to provide richer semantic information.
* Embedding Matrix: The embedding matrix is constructed using GloVe vectors, enhancing the model's ability to understand word meanings.
* Dropout: Dropout is applied to the LSTM outputs to prevent overfitting.