알고리즘 응용(00) Lab08

201802161 조은빈

1. Goal

1. Data 전처리
2. 딥러닝 네트워크 구현하기
3. 학습하기(colab)
4. Kaggle에 결과 제출하기

2. 코드 분석

```
import os
from collections import defaultdict
import numpy as np
import pandas as pd
import torch.nn as nn
import torch
from torch import optim
import matplotlib.pyplot as plt
from tqdm import tqdm, trange
```

코드를 구현하는 데 필요한 라이브러리 패키지를 import 한다.

```
def make_vocab(vocab_path, train=None):
    vocab = {}
    if os.path.isfile(vocab_path):
        file = open(vocab_path, 'r', encoding='utf-8')
        for line in file.readlines():
            line = line.rstrip()
            key, value = line.split('₩t')
            vocab[key] = value
        file.close()
    else:
        count_dict = defaultdict(int)
        for index, data in tqdm(train.iterrows(), desc='make vocab', total=len(train)):
            sentence = data['Phrase'].lower()
            tokens = sentence.split(' ')
            for token in tokens:
                count_dict[token] += 1

        file = open(vocab_path, 'w', encoding='utf-8')
        file.write('[UNK]₩t0₩n[PAD]₩t1₩n')
        vocab = {'[UNK]' : 0, '[PAD]' : 1}
        for index, (token, count) in enumerate(sorted(count_dict.items(), reverse=True,key=lambda item: item[1])):
            vocab[token] = index + 2
            file.write(token + '₩t' + str(index + 2) + '₩n')
        file.close()

    return vocab
```

make_vocab은 단어 사전을 만드는 함수이다. main에서 vocab을 저장할 경로와 train data를 파라미터로 받아온다. train data는 아래와 같은 형태이다.

PhraseId	SentenceId	Phrase	Sentiment
1	1	A series of escapades demonstrating the adage that what is good for the goose is also good for the gander , some of which occasionally amuses but none of which amounts to much of a story .	1
2	1	A series of escapades demonstrating the adage that what is good for the goose	2
3	1	A series	2
4	1	A	2
5	1	series	2
6	1	of escapades demonstrating the adage that what is good for the goose	2
7	1	of	2
8	1	escapades demonstrating the adage that what is good for the goose	2
9	1	escapades	2
10	1	demonstrating the adage that what is good for the goose	2
11	1	demonstrating the adage	2
12	1	demonstrating	2
13	1	the adage	2
14	1	the	2
15	1	adage	2
16	1	that what is good for the goose	2
17	1	that	2
18	1	what is good for the goose	2
19	1	what	2
20	1	is good for the goose	2
21	1	is	2
22	1	good for the goose	3
23	1	good	3
24	1	for the goose	2
25	1	for	2

train data에는 PhraseId, SetenceId, Phrase, Sentiment 항목이 있다. 단어 사전을 만들 때 필요한 정보는 Pharse이다. Pharse 데이터를 소문자로 바꾸고 띄어쓰기를 기준으로 나눠서 문장들을 단어들로 쪼갠다. 딕셔너리를 만드는 함수인 defaultdict를 통해 빈도수를 체크할 count_dict을 만든다. 'count_dict[token] +=1'은 단어들을 key로 사용해 동일한 단어가 나올 경우 value가 1 증가하게 한다. 이를 통해 각 단어들의 빈도수를 구한다.

빈도수를 기준으로 오름차순으로 정렬해 vocab에 저장한다. 이때 vocab에 [UNK]와 [PAD]를 추가해준다. UNK는 unknown 토큰을 의미하고 PAD는 padding 토큰을 의미한다. [UNK]와 [PAD]를 0, 1로 먼저 추가해줬기 때문에 count_dict의 데이터는 index에 2를 더해서 vocab에 넣어준다. 만들어진 단어 사전은 아래와 같다.

```
[UNK]   0
[PAD]   1
the     2
,       3
a       4
of      5
and     6
to      7
.       8
's      9
in      10
is      11
that    12
it      13
as      14
with    15
for     16
its     17
```

만약 이전에 만든 단어 사전이 있다면 새로 만들지 않고 이전에 만든 단어 사전을 열어서 읽는다.

```python
def read_data(train, test, vocab, max_len):
    x_train = np.ones(shape=(len(train), max_len))
    for i, data in tqdm(enumerate(train['Phrase']), desc='make x_train data', total=len(train)):
        data = data.lower() # 소문자로 바꾼다
        tokens = data.split(' ')    # 공백을 기준으로 토큰을 자른다
        for j, token in enumerate(tokens):
            if j == max_len:
                break
            x_train[i][j] = vocab[token]

    x_test = np.ones(shape=(len(test), max_len))
    for i, data in tqdm(enumerate(test['Phrase']), desc='make x_test data', total=len(test)):
        data = data.lower()
        tokens = data.split(' ')
        for j, token in enumerate(tokens):
            if j == max_len:
                break
            if token not in vocab.keys():   # test 데이터에는 unknown 토큰이 존재
                x_test[i][j] = 0
            else:
                x_test[i][j] = vocab[token]

    y_train = train['Sentiment'].to_numpy()

    return x_train, y_train, x_test
```

read_data는 train과 test에서 데이터를 읽어와 전처리하는 함수이다. 1로 채워진 np 어레이 x_train을 만든다. train의 Phrase 항목 즉, 문장들을 하나씩 읽어와 소문자로 바꾸고 공백을 기준으로 토큰을 자른다. 토큰들을 돌면서 vocab을 이용해 단어를 숫자로 바꿔준다. 숫자를 x_train에 저장한다.

A series of escapades demonstrating → a series of escapades demonstrating
[a, series, of, escapades, demonstrating]
[32, 3845, 92, 9798, 12934]

max_len은 문장의 최대 길이이다. 문장의 길이가 max_len을 넘더라도 max_len까지만 사용한다. 데이터의 길이가 너무 길어지면 성능이 하락하기 때문이다.

위 과정이 진행되는 것을 tqdm를 통해 진행표시바로 표시해준다.

```
make x_train data:  16%|          | 24404/156060 [00:00<00:00, 244035.58it/s]
```

test 데이터도 마찬가지의 과정을 거친다. 다른 점은 test 데이터에는 unknown 토큰이 존재한다는 점이다. 따라서 추가적으로 아래 코드가 작성된다.

```python
if token not in vocab.keys():   # test 데이터에는 unknown 토큰이 존재
    x_test[i][j] = 0
```

vocab에 저장되어 있지 않은 단어일 경우 x_test에 0을 삽입한다.

train 데이터에서 Sentiment 항목은 정답 항목이다. 정답 항목을 numpy 배열 객체로 만들어 y_train에 저장한다.

전처리 완료한 x_train, y_train, x_test를 리턴한다.

```python
class RNN(nn.Module):
    def __init__(self, input_size, embed_size, hidden_size, output_size, num_layers=1, bidirec=True, device='cuda'):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size
        self.num_layers = num_layers
        if bidirec:
            self.num_directions = 2
        else:
            self.num_directions = 1
        self.device = device

        self.embed = nn.Embedding(input_size, embed_size, padding_idx=1)
        self.lstm = nn.LSTM(embed_size, hidden_size, num_layers, batch_first=True, bidirectional=bidirec)
        self.linear = nn.Linear(hidden_size * self.num_directions, output_size)

    def init_hidden(self, batch_size):
        hidden = torch.zeros(self.num_layers * self.num_directions, batch_size, self.hidden_size).to(self.device)
        cell = torch.zeros(self.num_layers * self.num_directions, batch_size, self.hidden_size).to(self.device)
        return hidden, cell

    def forward(self, inputs):
        embed = self.embed(inputs)
        hidden, cell = self.init_hidden(inputs.size(0))
        output, (hidden, cell) = self.lstm(embed, (hidden, cell))

        hidden = hidden[-self.num_directions:]
        hidden = torch.cat([h for h in hidden], 1)
        output = self.linear(hidden)

        return output
```

RNN 클래스는 Network model을 정의한 클래스이다. RNN model을 만들 때 input_size, embed_size, hidden_size, output_size, num_layer, bidirec, device를 파라미터로 받는다.

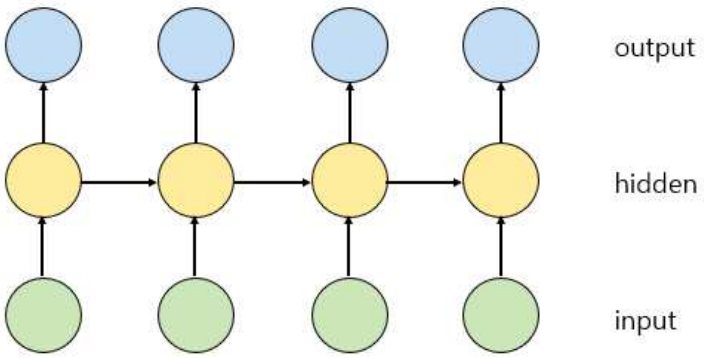
input_size : vocab의 길이

embed_size : 임베딩 할 벡터의 차원

hidden_size : RNN의 은닉층 사이즈

output_size : 0~5 범위로 저장된 Sentiment 항목을 예측해야하므로 5.

num_layer : RNN의 은닉층 개수. num_layer가 1일 경우 아래와 같이 은닉층이 1개.



bidirec : True일 시 양방향 RNN

nn.Embedding을 통해 Embedding 한다. nn.LSTM을 통해 lstm model을 만든다. LSTM은 RNN의 일종인 Long Short-Term Memory models이다. nn.Linear을 통해 데이터를 output_size로 추상화한다.

init_hidden은 hidden과 cell을 초기화하는 함수이다.

```
def get_acc(pred, answer):
    correct = 0
    for p, a in zip(pred, answer):
        pv, pi = p.max(0)
        if pi == a: # a는 One-Hot Encoding 값이 아니라서 바로 비교 가능
            correct += 1
    return correct / len(pred)
```

get_acc는 예측값과 정답을 받아서 accuracy를 구하는 함수이다. 예측 데이터 p와 정답 데이터 a를 비교하는데 이때 a는 One-Hot Encoding 된 값이 아니기 때문에 max를 찾는 과정은 생략되었다. p에서 max인 index 값을 추출해 a와 비교한 후 서로 동일하면 예측값이 정답이라는 의미가 되므로 corrent를 1 증가시킨다.

```python
def train(x, y, max_len, embed_size, hidden_size, output_size, batch_size, epochs, lr, device, model = None):
    x = torch.from_numpy(x).long()
    y = torch.from_numpy(y).long()
    if model is None:
        model = RNN(max_len, embed_size, hidden_size, output_size, device=device)
    model.to(device)
    model.train()
    loss_function = nn.CrossEntropyLoss()
    #loss_function = nn.MSELoss(reduction="mean")
    optimizer = optim.Adam(model.parameters(), lr=lr)
    data_loader = torch.utils.data.DataLoader(list(zip(x, y)), batch_size, shuffle=True)
    loss_total = []
    acc_total = []
    for epoch in trange(epochs):
        epoch_loss = 0
        epoch_acc = 0
        for batch_data in data_loader:
            x_batch, y_batch = batch_data
            x_batch = x_batch.to(device)    # to(device)는 GPU를 쓰겠다는 의미
            y_batch = y_batch.to(device)

            pred = model(x_batch)

            loss = loss_function(pred, y_batch)
            optimizer.zero_grad()
            loss.backward()

            optimizer.step()

            epoch_loss += loss.item()
            epoch_acc += get_acc(pred, y_batch)
        epoch_loss /= len(data_loader)
        epoch_acc /= len(data_loader)
        loss_total.append(epoch_loss)
        acc_total.append(epoch_acc)
        print("₩nEpoch [%d] Loss: %.3f₩tAcc:%.3f"%(epoch+1, epoch_loss, epoch_acc))

    torch.save(model, 'model.out')

    return model, loss_total, acc_total
```

train은 RNN 클래스로 model을 생성해서 학습시키는 함수이다. mnist 구현 코드와 유사하다.
x_train과 y_train을 받아 텐서 자료형으로 변환하고 RNN model을 생성한다.
loss_function으로는 nn.CrossEntropyLoss()를 사용한다.
optimizer는 Adam 알고리즘으로 파라미터를 업데이트하면서 모델을 최적화시킨다.
data_loader는 하나의 커다란 data를 batch 사이즈로 쪼개서 볼 수 있게 해준다.
지정한 epoch만큼 data를 돈다. data_loader로 일정하게 쪼개진 데이터를 돌아가면서 본다. 일정하게 쪼개진 데이터인 batch_data를 x_batch와 y_batch로 나눈다. to(device)로 x_batch와 y_batch를 GPU 연산하도록 한다. x_data를 model에 넣어서 결과를 예측한다. loss_function에 예측값인 pred와 정답 데이터인 y_batch를 넣고 둘 사이의 loss를 구한다. loss 값으로 학습시킨 수 step 함수를 통해 학습된 결과를 반영한다. 나온 loss 값을 epoch_loss에 더해준다. get_acc 함수를 통해 accuracy를 구해서 epoch_acc에 더해준다. 1 epoch이 완료되면 loss 확률과 accuracy 확률을 구해 각각 loss_total, acc_total에 저장한다.

```
def test(model, x, batch_size, devise):
    model.to(device)
    model.eval()
    x = torch.from_numpy(x).long()
    data_loader = torch.utils.data.DataLoader(x, batch_size, shuffle=False)

    predict = []
    for batch_data in data_loader:
        batch_data = batch_data.to(device)
        pred = model(batch_data)
        for p in pred:
            pv, pi = p.max(0)
            predict.append(pi.item())

    return predict
```

test 함수는 학습시킨 model에 test data를 넣어서 확인해보는 함수이다. model을 테스트할 경우 model.eval()을 선언해준다. data_loader로 data를 batch 사이즈로 분리한다. epoch 없이 data_loader를 돌려서 결과를 예측한다. 결과를 예측하고 예측값을 predict에 담아 반환한다.

```
def draw_graph(data):
    plt.plot(data)
    plt.show()

def save_submission(pred):
    data = {
        "PhraseId" : np.arange(156061, len(pred)+ 156061),
        "Sentiment" : pred
        }
    df = pd.DataFrame(data)
    df.to_csv('/content/drive/My Drive/Colab Notebooks/my_submission.csv', mode='w', index=False)
```

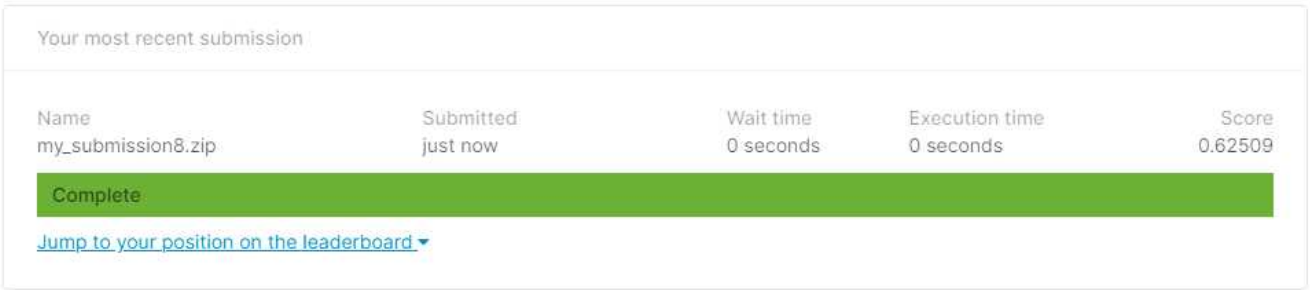
draw_graph 함수는 data를 받아 그래프를 그리고 시각화한다. save_submission 함수는 예측값을 csv 파일로 저장한다. 주어진 sampleSubmission.csv 파일에서 PhraseId가 156061부터 시작하므로 똑같이 만들어준다. 정답 항목인 Sentiment에는 예측값을 넣는다.

```
if __name__ == '__main__':
    train_path = '/content/drive/My Drive/Colab Notebooks/train.tsv'
    test_path = '/content/drive/My Drive/Colab Notebooks/test.tsv'
    vocab_path = '/content/drive/My Drive/Colab Notebooks/vocab.txt'

    train_data = pd.read_csv(train_path, sep='₩t')
    test_data = pd.read_csv(test_path, sep='₩t')
    vocab = make_vocab(vocab_path, train_data)
    #model = torch.load('model.out')

    device = torch.device('cuda:0') # 0 번 GPU 사용
    max_len = 50
    input_size = len(vocab)
    embed_size = 50
    hidden_size = 100
    output_size = 5
    batch_size = 1024
    epochs = 20
    lr = 0.001

    x_train, y_train, x_test = read_data(train_data, test_data, vocab, max_len)

    model, loss_total, acc_total = train(x_train, y_train, input_size, embed_size, hidden_size, output_size, batch_size, epochs, lr, device)
    draw_graph(loss_total)
    draw_graph(acc_total)
    predict = test(model, x_test, batch_size, device)
    save_submission(predict)
```

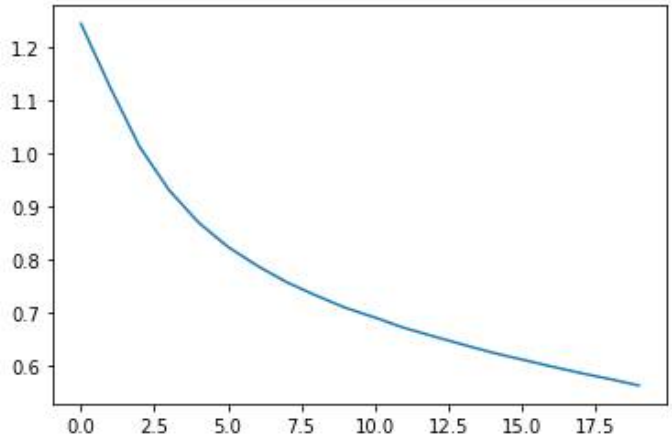
train_path, test_path, vocab_path를 지정한다.
read_csv를 통해 train_data와 test_data를 읽어오고 make_vocab으로 단어사전 vocab을 만든다.
GPU 연산을 하기 위해 torch.device('cuda:0')로 GPU 디바이스 객체를 생성한다.
파라미터로 사용할 값들을 각각 세팅한다. epoch를 20으로 늘려 accuracy를 증가시킨다.

3. 실행 결과

- Kaggle 결과



- Loss 그래프



- Accuracy 그래프