gate recurrent unit(gru)

# example code

|  |
| --- |
| import os  import torch  import torch.nn as nn  import torch.nn.functional as F  from torchtext.legacy import data, datasets  import torchtext  # 하이퍼파라미터 정의  BATCH\_SIZE = 64  lr = 0.001  EPOCHS = 10  USE\_CUDA = torch.cuda.is\_available()  DEVICE = torch.device("cuda" if USE\_CUDA else "cpu")#GPU, CPU  print("다음 기기로 학습합니다:", DEVICE)  # 데이터 로딩하기  print("데이터 로딩중...")  # data.Field 설명 #  # sequential인자 : TEXT는 Sequential 데이터라 True, Lable은 비Sequential이라 False로 설정  # batch\_first : Batch를 우선시 하여, Tensor 크기를 (BATCH\_SIZE, 문장의 최대 길이)로 설정  # lower : 소문자 전환 인자  # # # # # # # # # #  TEXT = data.Field(sequential=True, batch\_first=True, lower=True)  LABEL = data.Field(sequential=False, batch\_first=True)  #IMDB 데이터 로딩  trainset, testset = datasets.IMDB.splits(TEXT, LABEL)  # data.Field.build\_vocab() 라이브러리  # 문장 내 단어와 Integer index 를 매칭시키는 단어장(vocab)을 생성 == 워드 임베딩을 위한 Vocab 생성  # <UNK> = 0, <PAD> = 1 토큰도 추가.  # min\_freq : 최소 5번 이상 등장한 단어들만 사전에 담겠다는 것.  # 5번 미만으로 등장하는 단어는 UNK라는 토큰으로 대체  TEXT.build\_vocab(trainset, min\_freq=5)# TEXT 데이터를 기반으로 Vocab 생성  LABEL.build\_vocab(trainset)# LABEL 데이터를 기반으로 Vocab 생성  # 학습용 데이터를 학습셋 80% 검증셋 20% 로 나누기  trainset, valset = trainset.split(split\_ratio=0.8)  # 매 배치마다 비슷한 길이에 맞춰 줄 수 있도록 iterator 정의  train\_iter, val\_iter, test\_iter = data.BucketIterator.splits(          (trainset, valset, testset), batch\_size=BATCH\_SIZE,          shuffle=True, repeat=False)  vocab\_size = len(TEXT.vocab)  n\_classes = 2  # Positive, Negative Class가 두 개  '''  # Test #  for batch in train\_iter:      print(batch.text)      print(batch.label)      break  '''  print("[학습셋]: %d [검증셋]: %d [테스트셋]: %d [단어수]: %d [클래스] %d"        % (len(trainset),len(valset), len(testset), vocab\_size, n\_classes))  class BasicGRU(nn.Module):      def \_\_init\_\_(self, n\_layers, hidden\_dim, n\_vocab, embed\_dim, n\_classes, dropout\_p=0.2):          super(BasicGRU, self).\_\_init\_\_()          print("Building Basic GRU model...")          self.n\_layers = n\_layers # 일반적으로는 2          #n\_vocab : Vocab 안에 있는 단어의 개수, embed\_dim : 임베딩 된 단어 텐서가 갖는 차원 값(dimension)          self.embed = nn.Embedding(n\_vocab, embed\_dim)          # hidden state vector의 dimension과 dropout 정의          self.hidden\_dim = hidden\_dim          self.dropout = nn.Dropout(dropout\_p)          #앞에서 정의한 하이퍼 파라미터를 넣어 GRU 정의          self.gru = nn.GRU(embed\_dim, self.hidden\_dim,                            num\_layers=self.n\_layers,                            batch\_first=True)            #Input: GRU의 hidden state vector(context), Output : Class probability vector          self.out = nn.Linear(self.hidden\_dim, n\_classes)      def forward(self, x):          # Input data: 한 batch 내 모든 영화 평가 데이터            x = self.embed(x)# 영화 평 임베딩          h\_0 = self.\_init\_state(batch\_size=x.size(0)) # 초기 hidden state vector를 zero vector로 생성          x, \_ = self.gru(x, h\_0)  # [i, b, h] 출력값 :  (batch\_size, 입력 x의 길이, hidden\_dim)          # h\_t : Batch 내 모든 sequential hidden state vector의 제일 마지막 토큰을 내포한 (batch\_size, 1, hidden\_dim)형태의 텐서 추출          # 다른 의미로 영화 리뷰 배열들을 압축한 hidden state vector          h\_t = x[:,-1,:]          self.dropout(h\_t)# dropout 설정 후,          # linear layer의 입력으로 주고, 각 클래스 별 결과 logit을 생성.          logit = self.out(h\_t)  # [b, h] -> [b, o]          return logit        def \_init\_state(self, batch\_size=1):          weight = next(self.parameters()).data          return weight.new(self.n\_layers, batch\_size, self.hidden\_dim).zero\_()  def train(model, optimizer, train\_iter):      model.train()      for b, batch in enumerate(train\_iter):          x, y = batch.text.to(DEVICE), batch.label.to(DEVICE)          y.data.sub\_(1)  # 레이블 값을 (기존 1, 2)에서 0과 1로 변환          optimizer.zero\_grad()# 매번 기울기를 새로 계산하기 위해서 zero 로 초기화          logit = model(x)#모델의 예측값 logit 계산          loss = F.cross\_entropy(logit, y)# logit과 실제 label간의 오차를 구하고 기울기 계산          loss.backward()          optimizer.step()          if b % 30 == 0:              print("Train Epoch: {} [{}/{} ({:.0f}%)]\tTrain Loss: {:.6f}".format(                  e, b \* len(x),                  len(train\_iter.dataset), 100. \* b / len(train\_iter),                  loss.item()))    def evaluate(model, val\_iter):      """evaluate model"""      model.eval()      corrects, total\_loss = 0, 0      for batch in val\_iter:# Validation 데이터셋에 대하여          x, y = batch.text.to(DEVICE), batch.label.to(DEVICE)          y.data.sub\_(1) # 레이블 값을 0과 1로 변환          logit = model(x)          loss = F.cross\_entropy(logit, y, reduction='sum')          total\_loss += loss.item()          corrects += (logit.max(1)[1].view(y.size()).data == y.data).sum()      #전체 validation 셋에 대한 평균 loss와 accuracy를 구하는 과정      size = len(val\_iter.dataset)      avg\_loss = total\_loss / size      avg\_accuracy = 100.0 \* corrects / size      return avg\_loss, avg\_accuracy  model = BasicGRU(1, 256, vocab\_size, 128, n\_classes, 0.5).to(DEVICE)  optimizer = torch.optim.Adam(model.parameters(), lr=lr)  best\_val\_loss = None  for e in range(1, EPOCHS+1):      train(model, optimizer, train\_iter)      val\_loss, val\_accuracy = evaluate(model, val\_iter)      print("[이폭: %d] 검증 오차:%5.2f | 검증 정확도:%5.2f" % (e, val\_loss, val\_accuracy)) |

# testing result

|  |
| --- |
| Train Epoch: 1 [0/20000 (0%)] Train Loss: 0.707375  Train Epoch: 1 [1920/20000 (10%)] Train Loss: 0.706541  Train Epoch: 1 [3840/20000 (19%)] Train Loss: 0.691229  Train Epoch: 1 [5760/20000 (29%)] Train Loss: 0.702576  Train Epoch: 1 [7680/20000 (38%)] Train Loss: 0.688980  Train Epoch: 1 [9600/20000 (48%)] Train Loss: 0.692748  Train Epoch: 1 [11520/20000 (58%)] Train Loss: 0.696841  Train Epoch: 1 [13440/20000 (67%)] Train Loss: 0.731383  Train Epoch: 1 [15360/20000 (77%)] Train Loss: 0.689197  Train Epoch: 1 [17280/20000 (86%)] Train Loss: 0.688158  Train Epoch: 1 [19200/20000 (96%)] Train Loss: 0.709162  [이폭: 1] 검증 오차: 0.69 | 검증 정확도:49.70  Train Epoch: 2 [0/20000 (0%)] Train Loss: 0.696883  Train Epoch: 2 [1920/20000 (10%)] Train Loss: 0.696512  Train Epoch: 2 [3840/20000 (19%)] Train Loss: 0.690508  Train Epoch: 2 [5760/20000 (29%)] Train Loss: 0.698974  Train Epoch: 2 [7680/20000 (38%)] Train Loss: 0.684536  Train Epoch: 2 [9600/20000 (48%)] Train Loss: 0.693768  Train Epoch: 2 [11520/20000 (58%)] Train Loss: 0.722033  Train Epoch: 2 [13440/20000 (67%)] Train Loss: 0.704838  Train Epoch: 2 [15360/20000 (77%)] Train Loss: 0.713810  Train Epoch: 2 [17280/20000 (86%)] Train Loss: 0.680727  Train Epoch: 2 [19200/20000 (96%)] Train Loss: 0.683486  [이폭: 2] 검증 오차: 0.69 | 검증 정확도:52.08  Train Epoch: 3 [0/20000 (0%)] Train Loss: 0.704421  Train Epoch: 10 [13440/20000 (67%)] Train Loss: 0.007024 Train Epoch: 10 [15360/20000 (77%)] Train Loss: 0.004479 Train Epoch: 10 [17280/20000 (86%)] Train Loss: 0.019669 Train Epoch: 10 [19200/20000 (96%)] Train Loss: 0.002866 [이폭: 10] 검증 오차: 0.39 | 검증 정확도:87.00 |