bert

# example code

|  |
| --- |
| import re  import sys  import random  import torch  import torch.nn as nn  import torch.nn.functional as F  from torchtext.legacy import data  from torchtext.legacy import datasets  from transformers import BertTokenizer, BertModel  tokenizer = BertTokenizer.from\_pretrained('bert-base-uncased')  sentence = "My dog is cute. He likes playing. I bought a  pet food for him"  # sentence = '나는 책상 위에 사과를 먹었다. 알고 보니 그 사과는 Jason 것이었다. 그래서 Jason에게 사과를 했다'  print(tokenizer.tokenize(sentence))  len(tokenizer.vocab)  max\_input\_length = tokenizer.max\_model\_input\_sizes['bert-base-uncased']  print(max\_input\_length)  def new\_tokenizer(sentence):      tokens = tokenizer.tokenize(sentence)      tokens = tokens[:max\_input\_length-2]      return tokens  def PreProcessingText(input\_sentence):      input\_sentence = input\_sentence.lower() # 소문자화      input\_sentence = re.sub('<[^>]\*>', repl= ' ', string = input\_sentence) # "<br />" 처리      input\_sentence = re.sub('[!"$%&\()\*+,-./:;<=>?@[\\]^\_`{|}~]', repl= ' ', string = input\_sentence) # 특수문자 처리 ("'" 제외)      input\_sentence = re.sub('\s+', repl= ' ', string = input\_sentence) # 연속된 띄어쓰기 처리      if input\_sentence:          return input\_sentence  def PreProc(list\_sentence):      return [tokenizer.convert\_tokens\_to\_ids(PreProcessingText(x)) for x in list\_sentence]  TEXT = data.Field(batch\_first = True,                    use\_vocab = False,                    tokenize = new\_tokenizer,                    preprocessing = PreProc,                    init\_token = tokenizer.cls\_token\_id,                    eos\_token = tokenizer.sep\_token\_id,                    pad\_token = tokenizer.pad\_token\_id,                    unk\_token = tokenizer.unk\_token\_id)  LABEL = data.LabelField(dtype = torch.float)  train\_data, test\_data = datasets.IMDB.splits(TEXT, LABEL)  LABEL.build\_vocab(train\_data)  train\_data, valid\_data = train\_data.split(random\_state = random.seed(0), split\_ratio=0.8)  # Data Length  print(f'Train Data Length : {len(train\_data.examples)}')  print(f'Test Data Length : {len(test\_data.examples)}')  # Data Sample  print('---- Data Sample ----')  print('Input : ')  print(tokenizer.convert\_ids\_to\_tokens(vars(train\_data.examples[2])['text']))  # Label Info  print(f'Label Size : {len(LABEL.vocab)}')  print('Lable Examples : ')  for idx, (k, v) in enumerate(LABEL.vocab.stoi.items()):      print('\t', k, v)  model\_config = {}  model\_config['batch\_size'] = 8  device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')  train\_iterator, valid\_iterator, test\_iterator = data.BucketIterator.splits(      (train\_data, valid\_data, test\_data),      batch\_size=model\_config['batch\_size'],      device=device)  # Check batch data  sample\_for\_check = next(iter(train\_iterator))  print(sample\_for\_check)  print(sample\_for\_check.text)  print(sample\_for\_check.label)  bert = BertModel.from\_pretrained('bert-base-uncased')  model\_config['emb\_dim'] = bert.config.to\_dict()['hidden\_size']  class SentenceClassification(nn.Module):      def \_\_init\_\_(self, \*\*model\_config):          super(SentenceClassification, self).\_\_init\_\_()          self.bert = bert          self.fc = nn.Linear(model\_config['emb\_dim'],                              model\_config['output\_dim'])        def forward(self, x):          pooled\_cls\_output = self.bert(x)[1]          return self.fc(pooled\_cls\_output)  def train(model, iterator, optimizer, loss\_fn, idx\_epoch, \*\*model\_params):        epoch\_loss = 0      epoch\_acc = 0        model.train()      batch\_size = model\_params['batch\_size']      for idx, batch in enumerate(iterator):            # Initializing          optimizer.zero\_grad()            # Forward          predictions = model(batch.text).squeeze()          loss = loss\_fn(predictions, batch.label)          acc = binary\_accuracy(predictions, batch.label)            sys.stdout.write(                      "\r" + f"[Train] Epoch : {idx\_epoch:^3}"\                      f"[{(idx + 1) \* batch\_size} / {len(iterator) \* batch\_size} ({100. \* (idx + 1) / len(iterator) :.4}%)]"\                      f"  Loss: {loss.item():.4}"\                      f"  Acc : {acc.item():.4}"\                      )          # Backward          loss.backward()          optimizer.step()            # Update Epoch Performance          epoch\_loss += loss.item()          epoch\_acc += acc.item()        return epoch\_loss/len(iterator) , epoch\_acc/len(iterator)  def evaluate(model, iterator, loss\_fn, idx\_epoch, \*\*model\_params):        epoch\_loss = 0      epoch\_acc = 0        batch\_size = model\_params['batch\_size']        # evaluation mode      model.eval()      with torch.no\_grad():          for idx, batch in enumerate(iterator):              predictions = model(batch.text).squeeze()              loss = loss\_fn(predictions, batch.label)              acc = binary\_accuracy(predictions, batch.label)              epoch\_loss += loss.item()              epoch\_acc += acc.item()              sys.stdout.write(                      "\r" + f"[Eval] Epoch : {idx\_epoch:^3}"\                      f"[{(idx + 1) \* batch\_size} / {len(iterator) \* batch\_size} ({100. \* (idx + 1) / len(iterator) :.4}%)]"\                      f"  Loss: {loss.item():.4}"\                      f"  Acc : {acc.item():.4}"\                      )        return epoch\_loss / len(iterator), epoch\_acc / len(iterator)  model\_config.update(dict(output\_dim = 1))  def binary\_accuracy(preds, y):      # rounded\_preds = torch.argmax(preds, axis=1)      rounded\_preds = torch.round(torch.sigmoid(preds))      correct = (rounded\_preds == y).float()      acc = correct.sum()/len(correct)      return acc  model = SentenceClassification(\*\*model\_config)  optimizer = torch.optim.Adam(model.parameters(), lr=3e-5)  loss\_fn = nn.BCEWithLogitsLoss().to(device)  model = model.to(device)  def count\_parameters(model):      return sum(p.numel() for p in model.parameters() if p.requires\_grad)  count\_parameters(model)  N\_EPOCH = 4  best\_valid\_loss = float('inf')  model\_name = "BERT"  print('---------------------------------')  print(f'Model name : {model\_name}')  print('---------------------------------')  for epoch in range(N\_EPOCH):      train\_loss, train\_acc = train(model, train\_iterator, optimizer, loss\_fn, epoch, \*\*model\_config)      print('')      print(f'\t Epoch : {epoch} | Train Loss : {train\_loss:.4} | Train Acc : {train\_acc:.4}')      valid\_loss, valid\_acc = evaluate(model, valid\_iterator, loss\_fn, epoch, \*\*model\_config)      print('')      print(f'\t Epoch : {epoch} | Valid Loss : {valid\_loss:.4} | Valid Acc : {valid\_acc:.4}')      # print('')      if valid\_loss < best\_valid\_loss:          best\_valid\_loss = valid\_loss          torch.save(model.state\_dict(), f'./{model\_name}.pt')          print(f'\t Model is saved at {epoch}-epoch')      # Test set  # model.load\_state\_dict(torch.load(f'./{model\_name}.pt'))  epoch = 0  test\_loss, test\_acc = evaluate(model, test\_iterator, loss\_fn, epoch, \*\*model\_config)  print('')  print(f'Test Loss : {test\_loss:.4} | Test Acc : {test\_acc:.4}') |

# testing result

|  |
| --- |
| ---------------------------------  Model name : BERT  ---------------------------------  [Train]  Epoch : 0 [20000 / 20000 (100.0%)] Loss: 0.0814 Acc : 1.075  Epoch : 0 | Train Loss : 0.3003 | Train Acc : 0.8651  [Eval]  Epoch : 0 [5000 / 5000 (100.0%)] Loss: 0.2644 Acc : 0.875  Epoch : 0 | Valid Loss : 0.2061 | Valid Acc : 0.9166  Model is saved at 0-epoch  [Train]  Epoch : 1 [20000 / 20000 (100.0%)] Loss: 0.06414 Acc : 1.055  Epoch : 1 | Train Loss : 0.1619 | Train Acc : 0.9413  [Eval]  Epoch : 1 [5000 / 5000 (100.0%)] Loss: 0.1703 Acc : 1.075  Epoch : 1 | Valid Loss : 0.1944 | Valid Acc : 0.928  Model is saved at 1-epoch  [Train]  Epoch : 2 [20000 / 20000 (100.0%)] Loss: 0.01884 Acc : 1.005  Epoch : 2 | Train Loss : 0.09785 | Train Acc : 0.9665  [Eval]  Epoch : 2 [5000 / 5000 (100.0%)] Loss: 0.1581 Acc : 0.8755  Epoch : 2 | Valid Loss : 0.2447 | Valid Acc : 0.9246  [Train]  Epoch : 3 [20000 / 20000 (100.0%)] Loss: 0.001591 Acc : 1.05  Epoch : 3 | Train Loss : 0.06446 | Train Acc : 0.9799  [Eval]  Epoch : 3 [5000 / 5000 (100.0%)] Loss: 0.004881 Acc : 1.0  Epoch : 3 | Valid Loss : 0.296 | Valid Acc : 0.9272  [Eval] Epoch : 0 [25000 / 25000 (100.0%)] Loss: 0.6764 Acc : 0.6255 Test Loss : 0.1905 | Test Acc : 0.9264 |