long short-term memory(LSTM)

# example code

|  |
| --- |
| from google.colab import drive  drive.mount('/gdrive')  import pandas as pd  from sklearn.model\_selection import train\_test\_split  raw\_data\_path = '/content/drive/MyDrive/news.csv'  destination\_folder = '/content/drive/MyDrive'  train\_test\_ratio = 0.10  train\_valid\_ratio = 0.80  first\_n\_words = 200  def trim\_string(x):      x = x.split(maxsplit=first\_n\_words)      x = ' '.join(x[:first\_n\_words])      return x  df\_raw = pd.read\_csv(raw\_data\_path)  # column 준비  df\_raw['label'] = (df\_raw['label'] == 'FAKE').astype('int')  df\_raw['titletext'] = df\_raw['title'] + '.' + df\_raw['text']  df\_raw = df\_raw.reindex(columns = ['label', 'title','text','titletext'])  # => ['label', 'title','text','titletext'] 순으로 데이터를 정렬하고 인덱스를 다시 매김  # 비어있는 텍스트가 존재하는 열 제거  df\_raw.drop(df\_raw[df\_raw.text.str.len()<5].index, inplace = True)  # 텍스트를 자르고 titletext를 first\_n\_words까지 자름  df\_raw['text'] = df\_raw['text'].apply(trim\_string)  df\_raw['titletext'] = df\_raw['titletext'].apply(trim\_string)  # label을 기준으로 데이터를 나눈다  df\_real = df\_raw[df\_raw['label'] == 0]  df\_fake = df\_raw[df\_raw['label'] == 1]  # 훈련/테스트 분할  df\_real\_full\_train, df\_real\_test = train\_test\_split(df\_real, train\_size = train\_test\_ratio, random\_state = 1)  df\_fake\_full\_train, df\_fake\_test = train\_test\_split(df\_fake, train\_size = train\_test\_ratio, random\_state = 1)  # 훈련/검증 분할  df\_real\_train, df\_real\_valid = train\_test\_split(df\_real\_full\_train, train\_size = train\_valid\_ratio, random\_state = 1)  df\_fake\_train, df\_fake\_valid = train\_test\_split(df\_fake\_full\_train, train\_size = train\_valid\_ratio, random\_state = 1)  # 분할된 데이터를 연결한다. ( real + fake / 행 방향 )  df\_train = pd.concat([df\_real\_train, df\_fake\_train], ignore\_index=True, sort=False)  df\_valid = pd.concat([df\_real\_valid, df\_fake\_valid], ignore\_index=True, sort=False)  df\_test = pd.concat([df\_real\_test, df\_fake\_test], ignore\_index=True, sort=False)  # 전처리된 데이터셋 저장  df\_train.to\_csv(destination\_folder + '/train.csv', index=False)  df\_valid.to\_csv(destination\_folder + '/valid.csv', index=False)  df\_test.to\_csv(destination\_folder + '/test.csv', index=False)  import matplotlib.pyplot as plt  import pandas as pd  import torch  # Preliminaries  from torchtext.legacy.data import Field, TabularDataset, BucketIterator  # Models  import torch.nn as nn  from torch.nn.utils.rnn import pack\_padded\_sequence, pad\_packed\_sequence  device = torch.device("cuda")  # Training  import torch.optim as optim  # Evaluation  from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix  import seaborn as sns  label\_field = Field(sequential = False, # 순서가 있는 데이터일경우 True / Label은 순서가 필요 없으므로 False                      use\_vocab = False, # 단어장(vocab) 객체 사용 여부 / 텍스트 데이터에만 True                      batch\_first = True, # 배치를 우선시                      dtype = torch.float)  text\_field = Field(tokenize = 'spacy',                     lower = True, # 소문자 전환 여부 (엉어)                     include\_lengths = True, # 패딩된 미니 배치의 튜플과 각 샘플의 길이가 포함 된 목록을 반환 or 패딩된 미니 배치만 반환                     batch\_first = True)  fields = [('label', label\_field), ('title', text\_field), ('text', text\_field), ('titletext', text\_field)]  # Tabular Dataset  train, valid, test = TabularDataset.splits(path=destination\_folder, train='train.csv', validation='valid.csv', test='test.csv',                                             format='CSV', fields=fields, skip\_header=True)  # Iterators  # BucketIterator - batch learning을 함. 비슷한 길이를 가진 텍스트를 한 batch에 할당하여 패딩을 최소화하는 기능을 가짐.  train\_iter = BucketIterator(train, batch\_size=32, sort\_key=lambda x: len(x.text),                              device=device, sort=True, sort\_within\_batch=True)  valid\_iter = BucketIterator(valid, batch\_size=32, sort\_key=lambda x: len(x.text),                              device=device, sort=True, sort\_within\_batch=True)  test\_iter = BucketIterator(test, batch\_size=32, sort\_key=lambda x: len(x.text),                              device=device, sort=True, sort\_within\_batch=True)  # Vocabulary  text\_field.build\_vocab(train, min\_freq=3)  class LSTM(nn.Module): # nn.Module 상속      def \_\_init\_\_(self, dimension = 128):          super(LSTM, self).\_\_init\_\_()          self.embedding = nn.Embedding(len(text\_field.vocab), 300)          self.dimension = dimension          self.lstm = nn.LSTM(input\_size=300,                              hidden\_size=dimension,                              num\_layers=1,                              batch\_first=True,                              bidirectional=True) # 양방향          self.drop = nn.Dropout(p=0.5)          self.fc = nn.Linear(2\*dimension, 1) #full connect      def forward(self, text, text\_len):          text\_emb = self.embedding(text)          # pack\_padded\_sequence : padding된 문장을 padding 기준으로 sorting 하는 역할          # 참고할 글 : https://simonjisu.github.io/nlp/2018/07/05/packedsequence.html          packed\_input = pack\_padded\_sequence(text\_emb,                                              text\_len.cpu(),                                              batch\_first=True,                                              enforce\_sorted=False)          packed\_output, \_ = self.lstm(packed\_input)          output, \_ = pad\_packed\_sequence(packed\_output, batch\_first = True)          out\_forward = output[range(len(output)), text\_len - 1, :self.dimension]          out\_reverse = output[:, 0, self.dimension:]          out\_reduced = torch.cat((out\_forward, out\_reverse), 1) #[range(len(output)), text\_len -1 + 0 , self.dimension]          text\_fea = self.drop(out\_reduced)          text\_fea = self.fc(text\_fea)          text\_fea = torch.squeeze(text\_fea, 1)          text\_out = torch.sigmoid(text\_fea)          return text\_out  # 저장 / 불러오기 함수  def save\_checkpoint(save\_path, model, optimizer, valid\_loss):      if save\_path == None:          return      state\_dict = {'model\_state\_dict': model.state\_dict(),                    'optimizer\_state\_dict': optimizer.state\_dict(),                    'valid\_loss': valid\_loss}      torch.save(state\_dict, save\_path)      print(f'Model saved to ==> {save\_path}')  def load\_checkpoint(load\_path, model, optimizer):      if load\_path==None:          return      state\_dict = torch.load(load\_path, map\_location=device)      print(f'Model loaded from <== {load\_path}')      model.load\_state\_dict(state\_dict['model\_state\_dict'])      optimizer.load\_state\_dict(state\_dict['optimizer\_state\_dict'])      return state\_dict['valid\_loss']  def save\_metrics(save\_path, train\_loss\_list, valid\_loss\_list, global\_steps\_list):      if save\_path == None:          return      state\_dict = {'train\_loss\_list': train\_loss\_list,                    'valid\_loss\_list': valid\_loss\_list,                    'global\_steps\_list': global\_steps\_list}      torch.save(state\_dict, save\_path)      print(f'Model saved to ==> {save\_path}')  def load\_metrics(load\_path):      if load\_path==None:          return      state\_dict = torch.load(load\_path, map\_location=device)      print(f'Model loaded from <== {load\_path}')      return state\_dict['train\_loss\_list'], state\_dict['valid\_loss\_list'], state\_dict['global\_steps\_list']  def train(model,            optimizer,            criterion = nn.BCELoss(),            train\_loader = train\_iter,            valid\_loader = valid\_iter,            num\_epochs = 5,            eval\_every = len(train\_iter) // 2,            file\_path = destination\_folder,            best\_valid\_loss = float("Inf")):      # running values 초기화      running\_loss = 0.0      valid\_running\_loss = 0.0      global\_step = 0      train\_loss\_list = []      valid\_loss\_list = []      global\_steps\_list = []      # training loop      model.train()      for epoch in range(num\_epochs) :          for (labels, (title, title\_len), (text, text\_len), (titletext, titletext\_len)), \_ in train\_loader:              labels = labels.to(device)              titletext = titletext.to(device)              titletext\_len = titletext\_len.to(device)              output = model(titletext, titletext\_len)              loss = criterion(output, labels)              optimizer.zero\_grad()              loss.backward()              optimizer.step()              # update running values              running\_loss += loss.item()              global\_step += 1              # evaluation step              if global\_step % eval\_every == 0:                  model.eval()                  with torch.no\_grad():                      # validation loop                      for (labels, (title, title\_len), (text, text\_len), (titletext, titletext\_len)), \_ in valid\_loader:                          labels = labels.to(device)                          titletext = titletext.to(device)                          titletext\_len = titletext\_len.to(device)                          output = model(titletext, titletext\_len)                          loss = criterion(output, labels)                          valid\_running\_loss += loss.item()                  # evaluation                  average\_train\_loss = running\_loss / eval\_every                  average\_valid\_loss = valid\_running\_loss / len(valid\_loader)                  train\_loss\_list.append(average\_train\_loss)                  valid\_loss\_list.append(average\_valid\_loss)                  global\_steps\_list.append(global\_step)                  # resetting running values                  running\_loss = 0.0                  valid\_running\_loss = 0.0                  model.train()                  # print progress                  print('Epoch [{}/{}], Step [{}/{}], Train Loss: {:.4f}, Valid Loss: {:.4f}'                        .format(epoch+1, num\_epochs, global\_step, num\_epochs\*len(train\_loader),                                average\_train\_loss, average\_valid\_loss))                  # checkpoint                  if best\_valid\_loss > average\_valid\_loss:                      best\_valid\_loss = average\_valid\_loss                      save\_checkpoint(file\_path + '/model.pt', model, optimizer, best\_valid\_loss)                      save\_metrics(file\_path + '/metrics.pt', train\_loss\_list, valid\_loss\_list, global\_steps\_list)      save\_metrics(file\_path + '/metrics.pt', train\_loss\_list, valid\_loss\_list, global\_steps\_list)      print('Finished Training!')  model = LSTM().to(device)  optimizer = optim.Adam(model.parameters(), lr=0.001)  train(model=model, optimizer=optimizer, num\_epochs=10)  train\_loss\_list, valid\_loss\_list, global\_steps\_list = load\_metrics(destination\_folder + '/metrics.pt')  plt.plot(global\_steps\_list, train\_loss\_list, label='Train')  plt.plot(global\_steps\_list, valid\_loss\_list, label='Valid')  plt.xlabel('Global Steps')  plt.ylabel('Loss')  plt.legend()  plt.show()  def evaluate(model, test\_loader, version='title', threshold=0.5):      y\_pred = []      y\_true = []      model.eval()      with torch.no\_grad():          for (labels, (title, title\_len), (text, text\_len), (titletext, titletext\_len)), \_ in test\_loader:              labels = labels.to(device)              titletext = titletext.to(device)              titletext\_len = titletext\_len.to(device)              output = model(titletext, titletext\_len)              output = (output > threshold).int()              y\_pred.extend(output.tolist())              y\_true.extend(labels.tolist())      print('Classification Report:')      print(classification\_report(y\_true, y\_pred, labels=[1,0], digits=4))      cm = confusion\_matrix(y\_true, y\_pred, labels=[1,0])      ax= plt.subplot()      sns.heatmap(cm, annot=True, ax = ax, cmap='Blues', fmt="d")      ax.set\_title('Confusion Matrix')      ax.set\_xlabel('Predicted Labels')      ax.set\_ylabel('True Labels')      ax.xaxis.set\_ticklabels(['FAKE', 'REAL'])      ax.yaxis.set\_ticklabels(['FAKE', 'REAL'])  best\_model = LSTM().to(device)  optimizer = optim.Adam(best\_model.parameters(), lr=0.001)  load\_checkpoint(destination\_folder + '/model.pt', best\_model, optimizer)  evaluate(best\_model, test\_iter) |

# testing result

|  |
| --- |
| Epoch [1/10], Step [8/160], Train Loss: 0.6990, Valid Loss: 0.6755  Model saved to ==> /content/drive/MyDrive/model.pt  Model saved to ==> /content/drive/MyDrive/metrics.pt  Epoch [1/10], Step [16/160], Train Loss: 0.6938, Valid Loss: 0.6606  Model saved to ==> /content/drive/MyDrive/model.pt  Model saved to ==> /content/drive/MyDrive/metrics.pt  Epoch [2/10], Step [24/160], Train Loss: 0.5682, Valid Loss: 0.6484  Model saved to ==> /content/drive/MyDrive/model.pt  Model saved to ==> /content/drive/MyDrive/metrics.pt  Epoch [2/10], Step [32/160], Train Loss: 0.5969, Valid Loss: 0.6341  Model saved to ==> /content/drive/MyDrive/model.pt  Model saved to ==> /content/drive/MyDrive/metrics.pt  Epoch [3/10], Step [40/160], Train Loss: 0.4826, Valid Loss: 0.6228  Model saved to ==> /content/drive/MyDrive/model.pt  Model saved to ==> /content/drive/MyDrive/metrics.pt  Epoch [3/10], Step [48/160], Train Loss: 0.4923, Valid Loss: 0.6052  Model saved to ==> /content/drive/MyDrive/model.pt  Model saved to ==> /content/drive/MyDrive/metrics.pt  Epoch [4/10], Step [56/160], Train Loss: 0.3639, Valid Loss: 0.5911  Model saved to ==> /content/drive/MyDrive/model.pt  Model saved to ==> /content/drive/MyDrive/metrics.pt  Epoch [4/10], Step [64/160], Train Loss: 0.3506, Valid Loss: 0.5491  Model saved to ==> /content/drive/MyDrive/model.pt  Model saved to ==> /content/drive/MyDrive/metrics.pt  Epoch [5/10], Step [72/160], Train Loss: 0.2037, Valid Loss: 0.5513  Epoch [9/10], Step [144/160], Train Loss: 0.0192, Valid Loss: 0.6540 Epoch [10/10], Step [152/160], Train Loss: 0.0188, Valid Loss: 0.5264 Epoch [10/10], Step [160/160], Train Loss: 0.0099, Valid Loss: 0.5189 Model saved to ==> /content/drive/MyDrive/metrics.pt Finished Training! |