Coding Quiz

With the given dataset, Please compare your best possible version of

```
(1) BiLSTM,
(2) BiLSTM with multiplicative attention (you have to fix e), and
(3) BERT
```

Report the accuracy, precision, recall, and f1-score of each model.

For (1) and (2), use the following hyperparameters:

```
Optimizer: SG
Embedding: GloVe (https://pytorch.org/text/stable/vocab.html#torchtext.voca
b.GloVe) >> Please change the embed_dim accordingly.
Epochs: 2
Batch size: 32
Save the model with the best params
```

Anything not stated, please assume accordingly

For (2), Multiplicative attention differs from the General Attention (in Assignment 4) such that, for the *Alignment Scores* (or Energy), we multiply the Keys with some weights first before we dot the Keys with the Query.

$$\mathbf{e}_i = \mathbf{q}^T \ \mathbf{W}\mathbf{k}_t$$

where $\mathbf{W} \in \mathbb{R}^{h,h}$

- Hint : The shape of the Keys before and after multiplying with the weights should be the same

For (3), use this tutorial [https://huggingface.co/docs/transformers/training (https://huggingface.co/docs/transformers/training)](https://huggingface.co/docs/transformers/training) as your guide.

In [2]:

```
# import os

# os.environ['http_proxy'] = 'http://192.41.170.23:3128'
# os.environ['https_proxy'] = 'http://192.41.170.23:3128'
```

In [1]:

```
import torchtext
import torch
from torch import nn
import math
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
```

```
cuda
```

1. Load the IMDB Review dataset from TorchText ([https://pytorch.org/text/stable/datasets.html#id10 (https://pytorch.org/text/stable/datasets.html#id10)](https://pytorch.org/text/stable/datasets.html#id10))

In [11]:

```
from torchtext.data.utils import get_tokenizer
tokenizer = get_tokenizer('spacy', language='en_core_web_sm')
# tokens = tokenizer("We are learning torchtext in U.K.!")  #some test
# tokens
```

In [12]:

```
from torchtext.vocab import build_vocab_from_iterator
def yield_tokens(data_iter):
    for _, text in data_iter:
        yield tokenizer(text)

vocab = build_vocab_from_iterator(yield_tokens(train_iter), specials=['<unk>', '<pac
vocab.set_default_index(vocab["<unk>"])
```

In [13]:

```
text_pipeline = lambda x: vocab(tokenizer(x))
label_pipeline = lambda x: 1 if x == 'pos' else 0

from torch.utils.data import DataLoader
from torch.nn.utils.rnn import pad_sequence #++

def collate_batch(batch):
    label_list, text_list, length_list = [], [], []
    for (_label, _text) in batch:
        label_list.append(label_pipeline(_label))
        processed_text = torch.tensor(text_pipeline(_text), dtype=torch.int64)
        text_list.append(processed_text)
        length_list.append(processed_text.size(0))  #++<-----packed padded sequences
    #criterion expects float labels
    return torch.tensor(label_list, dtype=torch.float64), pad_sequence(text_list, pa

from torchtext.datasets import IMDB
from torch.utils.data.dataset import random_split
from torchtext.data.functional import to_map_style_dataset

train_iter = IMDB(split='train')
test_iter = IMDB(split='test')

train_dataset = to_map_style_dataset(train_iter)
test_dataset = to_map_style_dataset(test_iter)

batch_size = 32
num_train = int(len(train_dataset) * 0.15)
num_val = int(len(train_dataset) * 0.10)
num_test = int(len(test_dataset) * 0.05)

split_train_, split_valid_, _ = \
    random_split(train_dataset, [num_train, num_val,len(train_dataset)- num_train -

split_test_, _ = \
    random_split(train_dataset, [num_test, len(test_dataset) - num_test])

train_loader = DataLoader(split_train_, batch_size=batch_size,
                              shuffle=True, collate_fn=collate_batch)
valid_loader = DataLoader(split_valid_, batch_size=batch_size,
                              shuffle=True, collate_fn=collate_batch)
test_loader = DataLoader(split_test_, batch_size=batch_size,
                             shuffle=True, collate_fn=collate_batch)
```

In [16]:

```
# print('valid',len(valid_loader))
# print('train',len(train_loader))
```

In [17]:

```
# from torchtext.vocab import FastText
from torchtext.vocab import GloVe

glove_vector = torchtext.vocab.GloVe(name='6B', dim=300)
fast_embedding = glove_vector.get_vecs_by_tokens(vocab.get_itos()).to(device)
```

In [18]:

```
input_dim = len(vocab)
hidden_dim = 256
embed_dim = 300
output_dim = 1

pad_idx = vocab['<pad>']
num_layers = 2
bidirectional = True
dropout = 0.5


num_epochs = 2
lr=0.0001
```

In [19]:

```
#explicitly initialize weights for better learning
def initialize_weights(m):
    if isinstance(m, nn.Linear):
        nn.init.xavier_normal_(m.weight)
        nn.init.zeros_(m.bias)
    elif isinstance(m, nn.RNN):
        for name, param in m.named_parameters():
            if 'bias' in name:
                nn.init.zeros_(param)
            elif 'weight' in name:
                nn.init.orthogonal_(param) #<---here

def binary_accuracy(preds, y):
    """
    Returns accuracy per batch, i.e. if you get 8/10 right, this returns 0.8, NOT 8
    """
    #round predictions to the closest integer
    rounded_preds = torch.round(torch.sigmoid(preds))
    correct = (rounded_preds == y).float() #convert into float for division
    acc = correct.sum() / len(correct)
    return acc
```

In [20]:

```
def train(model, loader, optimizer, criterion):
    epoch_loss = 0
    epoch_acc = 0
    model.train() #useful for batchnorm and dropout
    for i, (label, text, text_length) in enumerate(loader):
        label = label.to(device) #(batch_size, )
        text = text.to(device) #(batch_size, seq len)

        #predict
        predictions = model(text, text_length) #output by the fc is (batch_size, 1),
        predictions = predictions.squeeze(1)

        #calculate loss
        loss = criterion(predictions, label)
        acc = binary_accuracy(predictions, label)

        #backprop
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

        if i == 10:
            break

    return epoch_loss / len(loader), epoch_acc / len(loader)


def evaluate(model, loader, criterion):
    epoch_loss = 0
    epoch_acc = 0
    model.eval()

    with torch.no_grad():
        for i, (label, text, text_length) in enumerate(loader):
            label = label.to(device) #(batch_size, )
            text = text.to(device) #(batch_size, seq len)

            predictions = model(text, text_length)
            predictions = predictions.squeeze(1)

            loss = criterion(predictions, label)
            acc = binary_accuracy(predictions, label)

            epoch_loss += loss.item()
            epoch_acc += acc.item()

            if i == 10:
                break

    return epoch_loss / len(loader), epoch_acc / len(loader)
```

BiLSTM

In [21]:

```
ass new_LSTM_cell(nn.Module):
  def __init__(self, input_dim: int, hidden_dim: int, lstm_type: str):
      super().__init__()

      self.hidden_dim = hidden_dim
      self.lstm_type = lstm_type

      # initialise the trainable Parameters
      self.U_i = nn.Parameter(torch.Tensor(input_dim, hidden_dim))
      self.W_i = nn.Parameter(torch.Tensor(hidden_dim, hidden_dim))
      self.b_i = nn.Parameter(torch.Tensor(hidden_dim))

      self.U_f = nn.Parameter(torch.Tensor(input_dim, hidden_dim))
      self.W_f = nn.Parameter(torch.Tensor(hidden_dim, hidden_dim))
      self.b_f = nn.Parameter(torch.Tensor(hidden_dim))

      self.U_g = nn.Parameter(torch.Tensor(input_dim, hidden_dim))
      self.W_g = nn.Parameter(torch.Tensor(hidden_dim, hidden_dim))
      self.b_g = nn.Parameter(torch.Tensor(hidden_dim))

      self.U_o = nn.Parameter(torch.Tensor(input_dim, hidden_dim))
      self.W_o = nn.Parameter(torch.Tensor(hidden_dim, hidden_dim))
      self.b_o = nn.Parameter(torch.Tensor(hidden_dim))

      if self.lstm_type == 'peephole' :
          self.P_i = nn.Parameter(torch.Tensor(hidden_dim, hidden_dim))
          self.P_f = nn.Parameter(torch.Tensor(hidden_dim, hidden_dim))
          self.P_o = nn.Parameter(torch.Tensor(hidden_dim, hidden_dim))

      self.init_weights()

  def init_weights(self):
      stdv = 1.0 / math.sqrt(self.hidden_dim)
      for weight in self.parameters():
          weight.data.uniform_(-stdv, stdv)

  def forward(self, x, init_states=None):
      bs, seq_len, _ = x.shape
      output = []

      # initialize the hidden state and cell state for the first time step
      if init_states is None:
          h_t  = torch.zeros(bs, self.hidden_dim).to(x.device)
          c_t  = torch.zeros(bs, self.hidden_dim).to(x.device)
      else:
          h_t, c_t = init_states

      # For each time step of the input x, do ...
      for t in range(seq_len):
          x_t = x[:, t, :] # get x data of time step t (SHAPE: (batch_size, input_di

          if self.lstm_type in ['vanilla', 'coupled'] :
              f_t = torch.sigmoid(    h_t @ self.W_f  +  x_t @ self.U_f  +  self.b_f
              o_t = torch.sigmoid(    h_t @ self.W_o  +  x_t @ self.U_o  +  self.b_o
              if self.lstm_type == 'vanilla':
                  i_t = torch.sigmoid(    h_t @ self.W_i  +  x_t @ self.U_i  +  self.
              if self.lstm_type == 'coupled':
                  i_t = (1 - f_t)
          if self.lstm_type == 'peephole' :
```

```
            i_t = torch.sigmoid( h_t @ self.W_i + x_t @ self.U_i + c_t @ self.P_i +
            f_t = torch.sigmoid( h_t @ self.W_f + x_t @ self.U_f + c_t @ self.P_f +
            o_t = torch.sigmoid( h_t @ self.W_o + x_t @ self.U_o + c_t @ self.P_o +

        g_t = torch.tanh(       h_t @ self.W_g  +  x_t @ self.U_g   + self.b_g)
        c_t = (f_t * c_t) + (i_t * g_t)
        h_t = o_t * torch.tanh(c_t)

        output.append(h_t.unsqueeze(0)) # reshape h_t to (1, batch_size, hidden_di

    output = torch.cat(output, dim = 0) # concatenate h_t of all time steps into SH
    output = output.transpose(0, 1).contiguous() # just transpose to SHAPE :(seq_le
    return output, (h_t, c_t)
```

In [22]:

```
class BiLSTM_model(nn.Module):
    def __init__(self, input_dim: int, embed_dim: int, hidden_dim: int, output_dim:
        super().__init__()
        self.num_directions = 2
        self.embedding = nn.Embedding(input_dim, embed_dim, padding_idx=pad_idx)
        self.hidden_dim = hidden_dim

        self.forward_lstm   =  new_LSTM_cell(embed_dim, hidden_dim, lstm_type = 'var
        self.backward_lstm  =  new_LSTM_cell(embed_dim, hidden_dim, lstm_type = 'var

        # These should be torch Parameters
        self.W_h = nn.Parameter(torch.Tensor(hidden_dim*self.num_directions, hidden_
        self.b_h = nn.Parameter(torch.Tensor(hidden_dim*self.num_directions))

        self.fc  = nn.Linear(hidden_dim*self.num_directions, output_dim)

        self.init_weights()

    def init_weights(self):
        stdv = 1.0 / math.sqrt(self.hidden_dim)
        for weight in self.parameters():
            weight.data.uniform_(-stdv, stdv)

    def forward(self, text, text_lengths):
        embedded      = self.embedding(text)
        embedded_flip =  torch.flip(embedded, [1])

        output_forward, (hn_forward, cn_forward)    = self.forward_lstm(embedded, in
        output_backward, (hn_backward, cn_backward) = self.backward_lstm(embedded_fl

        concat_hn = torch.cat( (hn_forward, hn_backward), dim=1 )
        ht        = torch.sigmoid( concat_hn @ self.W_h + self.b_h)

        return self.fc(ht)
```

In [23]:

```
import torch.optim as optim

bilstm = BiLSTM_model(input_dim, embed_dim, hidden_dim, output_dim).to(device)
bilstm.apply(initialize_weights)
bilstm.embedding.weight.data = fast_embedding

optimizer = optim.SGD(bilstm.parameters(), lr=lr)
criterion = nn.BCEWithLogitsLoss() #combine sigmoid with binary cross entropy

train_losses = []
train_accs = []
valid_losses = []
valid_accs = []
best_valid_loss = float('inf')


for epoch in range(num_epochs):
    train_loss, train_acc = train(bilstm, train_loader, optimizer, criterion)
    valid_loss, valid_acc = evaluate(bilstm, valid_loader, criterion)

    train_losses.append(train_loss)
    train_accs.append(train_acc)
    valid_losses.append(valid_loss)
    valid_accs.append(valid_acc)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(bilstm.state_dict(), 'BiLSTM-model.pt')

    print(f'Epoch: {epoch+1:02} | Train Loss: {train_loss:.3f} | Train Acc: {train_a
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')

# del bilstm
# del optimizer
# del criterion
```

```
Epoch: 01 | Train Loss: 0.069 | Train Acc: 4.98%
         Val. Loss: 0.106 |  Val. Acc: 7.08%
Epoch: 02 | Train Loss: 0.074 | Train Acc: 4.26%
         Val. Loss: 0.101 |  Val. Acc: 7.63%
```

In [38]:

```
# Test
bilstm.load_state_dict(torch.load('BiLSTM-model.pt'))
test_loss, test_acc = evaluate(bilstm, test_iter, criterion)
print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 0.000 | Test Acc: 0.00%
```

In [27]:

```
epoch_loss = 0
epoch_acc = 0
#bilstm = BiLSTM_model(input_dim, embed_dim, hidden_dim, output_dim).to(device)
bilstm.eval()

with torch.no_grad():
    for i, (label, text, text_length) in enumerate(train_loader):
        label = label.to(device) #(batch_size, )
        text = text.to(device) #(batch_size, seq len)

        predictions = bilstm(text, text_length)
        predictions = predictions.squeeze(1)

        loss = criterion(predictions, label)
        acc = binary_accuracy(predictions, label)

        epoch_loss += loss.item()
        epoch_acc += acc.item()
```

In [28]:

```
print(predictions)
print(label)
```

```
tensor([-0.7532, -0.7532, -0.7532, -0.7532, -0.7532, -0.7532], device
='cuda:0')
tensor([1., 1., 1., 0., 1., 0.], device='cuda:0', dtype=torch.float64)
```

In [31]:

```
def confusion(prediction, truth):

    rounded_preds = torch.round(torch.sigmoid(prediction))
    confusion_vector = rounded_preds / truth
    true_positives = torch.sum(confusion_vector == 1).item()
    false_positives = torch.sum(confusion_vector == float('inf')).item()
    true_negatives = torch.sum(torch.isnan(confusion_vector)).item()
    false_negatives = torch.sum(confusion_vector == 0).item()

    return true_positives, false_positives, true_negatives, false_negatives
```

In [32]:

```
true_positives, false_positives, true_negatives, false_negatives = confusion(predict
```

In [35]:

```
total = true_positives + false_positives + true_negatives + false_negatives

accuracy = (true_positives + true_negatives) / (total * 1.0)
precision = (1.0 * true_positives) / (true_positives + false_positives)
recall = (1.0 * true_positives) / (true_positives + false_negatives)
f1 = 2.0 / ((1.0 / precision) + (1.0 / recall))
```

LSTM Attention

In [36]:

```
import torch.nn as nn
from torch.nn import functional as F

class LSTM_GAtt(nn.Module):
    def __init__(self, input_dim: int, embed_dim: int, hidden_dim: int, output_dim:
        super().__init__()

        self.embedding = nn.Embedding(input_dim, embed_dim, padding_idx=pad_idx)

        # let's use pytorch's LSTM
        self.lstm = nn.LSTM(embed_dim,
                            hidden_dim,
                            num_layers=num_layers,
                            bidirectional=bidirectional,
                            dropout=dropout,
                            batch_first=True)

        # Linear Layer for binary classification
        self.fc = nn.Linear(hidden_dim * 2, output_dim)
        self.W = nn.Parameter(torch.Tensor(batch_size, hidden_dim * 2,hidden_dim * 2
    def attention_net(self, lstm_output, hn):

        h_t      = hn.unsqueeze(2)
        H_keys   = torch.clone(lstm_output)
        H_values = torch.clone(lstm_output)
        H_query = torch.clone(lstm_output)

        # k_w = torch.bmm(H_keys, self.W)
        #   #alignment_score = torch.bmm(h_t, k_w.transpose(1,2))  # SHAPE : (bs, se
        # alignment_score   = torch.bmm(k_w, h_t).squeeze(2) # SHAPE : (bs, seq_len,

        #alignment_score   = torch.bmm(H_keys, h_t).squeeze(2) # SHAPE : (bs, seq_le
        score = torch.bmm(H_keys, self.W)
        # # score   = self.W @  H_keys
        alignment_score = torch.bmm(score,h_t).squeeze(2)
        # alignment_score = (torch.bmm(self.W, H_keys).squeeze(2)

        soft_attn_weights = F.softmax(alignment_score, 1) # SHAPE : (bs, seq_len, 1)

        context           = torch.bmm(H_values.transpose(1, 2), soft_attn_weights.un

        return context

    def forward(self, text, text_lengths):

        embedded = self.embedding(text) # SHAPE : (batch_size, seq_len, embed_dim)

        lstm_output, (hn, cn) = self.lstm(embedded)

        # This is how we concatenate the forward hidden and backward hidden from Pyt
        hn = torch.cat((hn[-2,:,:], hn[-1,:,:]), dim = 1)

        attn_output = self.attention_net(lstm_output, hn)

        return self.fc(attn_output)
```

In [37]:

```
#m_attmodel = LSTM_GAtt(input_dim, embed_dim, hidden_dim, output_dim, len_reduction
m_attmodel = LSTM_GAtt(input_dim, embed_dim, hidden_dim, output_dim).to(device)
m_attmodel.apply(initialize_weights)
m_attmodel.embedding.weight.data = fast_embedding

optimizer = optim.SGD(m_attmodel.parameters(), lr=lr) #<----changed to Adam
criterion = nn.BCEWithLogitsLoss() #combine sigmoid with binary cross entropy

train_losses = []
train_accs = []
valid_losses = []
valid_accs = []
best_valid_loss = float('inf')

for epoch in range(num_epochs):
    train_loss, train_acc = train(m_attmodel, train_loader, optimizer, criterion)
    valid_loss, valid_acc = evaluate(m_attmodel, valid_loader, criterion)

    train_losses.append(train_loss)
    train_accs.append(train_acc)
    valid_losses.append(valid_loss)
    valid_accs.append(valid_acc)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(m_attmodel.state_dict(), 'LSTMMultiAtt-model.pt')

    print(f'Epoch: {epoch+1:02} | Train Loss: {train_loss:.3f} | Train Acc: {train_a
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')

# del g_attmodel
# del optimizer
# del criterion
```

```
Epoch: 01 | Train Loss: 0.065 | Train Acc: 4.90%
         Val. Loss: 0.096 |  Val. Acc: 7.87%
Epoch: 02 | Train Loss: 0.065 | Train Acc: 4.40%
         Val. Loss: 0.096 |  Val. Acc: 7.44%
```

In [39]:

```
# Test
m_attmodel.load_state_dict(torch.load('LSTMMultiAtt-model.pt'))
test_loss, test_acc = evaluate(m_attmodel, test_iter, criterion)
print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 0.000 | Test Acc: 0.00%
```

In [ ]:

```
```

In [42]:

```
def confusion(prediction, truth):

    rounded_preds = torch.round(torch.sigmoid(prediction))
    confusion_vector = rounded_preds / truth
    true_positives = torch.sum(confusion_vector == 1).item()
    false_positives = torch.sum(confusion_vector == float('inf')).item()
    true_negatives = torch.sum(torch.isnan(confusion_vector)).item()
    false_negatives = torch.sum(confusion_vector == 0).item()

    return true_positives, false_positives, true_negatives, false_negatives
```

In [44]:

```
true_positives, false_positives, true_negatives, false_negatives = confusion(predict
```

In [ ]:

```
total = true_positives + false_positives + true_negatives + false_negatives

accuracy = (true_positives + true_negatives) / (total * 1.0)
precision = (1.0 * true_positives) / (true_positives + false_positives)
recall = (1.0 * true_positives) / (true_positives + false_negatives)
f1 = 2.0 / ((1.0 / precision) + (1.0 / recall))
```

BERT

In [45]:

```
!pip install transformers
```

```
Requirement already satisfied: transformers in /usr/local/lib/python3.
7/dist-packages (4.17.0)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.
7/dist-packages (from transformers) (1.21.5)
Requirement already satisfied: huggingface-hub<1.0,>=0.1.0 in /usr/loc
al/lib/python3.7/dist-packages (from transformers) (0.4.0)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/pyt
hon3.7/dist-packages (from transformers) (2019.12.20)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/pytho
n3.7/dist-packages (from transformers) (21.3)
Requirement already satisfied: filelock in /usr/local/lib/python3.7/di
st-packages (from transformers) (3.6.0)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.7/
dist-packages (from transformers) (4.63.0)
Requirement already satisfied: requests in /usr/local/lib/python3.7/di
st-packages (from transformers) (2.23.0)
Requirement already satisfied: importlib-metadata in /usr/local/lib/py
thon3.7/dist-packages (from transformers) (4.11.2)
Requirement already satisfied: tokenizers!=0.11.3,>=0.11.1 in /usr/loc
al/lib/python3.7/dist-packages (from transformers) (0.11.6)
Requirement already satisfied: sacremoses in /usr/local/lib/python3.7/
dist-packages (from transformers) (0.0.47)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.
7/dist-packages (from transformers) (6.0)
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/loca
l/lib/python3.7/dist-packages (from huggingface-hub<1.0,>=0.1.0->trans
formers) (3.10.0.2)
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in /usr/local/
lib/python3.7/dist-packages (from packaging>=20.0->transformers) (3.0.
7)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/d
ist-packages (from importlib-metadata->transformers) (3.7.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/py
thon3.7/dist-packages (from requests->transformers) (2021.10.8)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/pyt
hon3.7/dist-packages (from requests->transformers) (3.0.4)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.
7/dist-packages (from requests->transformers) (2.10)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1
 in /usr/local/lib/python3.7/dist-packages (from requests->transformer
s) (1.24.3)
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-pa
ckages (from sacremoses->transformers) (1.15.0)
Requirement already satisfied: joblib in /usr/local/lib/python3.7/dist
-packages (from sacremoses->transformers) (1.1.0)
Requirement already satisfied: click in /usr/local/lib/python3.7/dist-
packages (from sacremoses->transformers) (7.1.2)
```

In [46]:

```
### BERT

from transformers import AutoModelForSequenceClassification
from transformers import AutoTokenizer
from transformers import TrainingArguments
#from transformers import DistilBertTokenizerFast

model_name = "bert-base-cased"
tokenizer = AutoTokenizer.from_pretrained(model_name)

def tokenize_function(examples):
    return tokenizer(examples["text"], padding="max_length", truncation=True)

print(tokenizer)


#tokenizer = DistilBertTokenizerFast.from_pretrained('distilbert-base-uncased')

# tokenized_train_datasets = train_dataset.map(tokenize_function)
# tokenized_test_datasets = test_dataset.map(tokenize_function, b)
```

```
PreTrainedTokenizerFast(name_or_path='bert-base-cased', vocab_size=289
96, model_max_len=512, is_fast=True, padding_side='right', truncation_
side='right', special_tokens={'unk_token': '[UNK]', 'sep_token': '[SE
P]', 'pad_token': '[PAD]', 'cls_token': '[CLS]', 'mask_token': '[MAS
K]'})
```

In [47]:

```
model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", num_la
```

```
Some weights of the model checkpoint at bert-base-cased were not used
when initializing BertForSequenceClassification: ['cls.predictions.tra
nsform.dense.weight', 'cls.predictions.transform.LayerNorm.bias', 'cl
s.predictions.decoder.weight', 'cls.seq_relationship.bias', 'cls.predi
ctions.transform.LayerNorm.weight', 'cls.predictions.transform.dense.b
ias', 'cls.seq_relationship.weight', 'cls.predictions.bias']
- This IS expected if you are initializing BertForSequenceClassificati
on from the checkpoint of a model trained on another task or with anot
her architecture (e.g. initializing a BertForSequenceClassification mo
del from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertForSequenceClassifi
cation from the checkpoint of a model that you expect to be exactly id
entical (initializing a BertForSequenceClassification model from a Ber
tForSequenceClassification model).
Some weights of BertForSequenceClassification were not initialized fro
m the model checkpoint at bert-base-cased and are newly initialized:
['classifier.weight', 'classifier.bias']
You should probably TRAIN this model on a down-stream task to be able
to use it for predictions and inference.
```

In [48]:

```
print(tokenizer)
```

```
PreTrainedTokenizerFast(name_or_path='bert-base-cased', vocab_size=289
96, model_max_len=512, is_fast=True, padding_side='right', truncation_
side='right', special_tokens={'unk_token': '[UNK]', 'sep_token': '[SE
P]', 'pad_token': '[PAD]', 'cls_token': '[CLS]', 'mask_token': '[MAS
K]'})
```

In [49]:

```
tokens = tokenizer("We are learning torchtext in U.K.!")  #some test
tokens
```

Out[49]:

```
{'input_ids': [101, 1284, 1132, 3776, 16328, 17380, 1107, 158, 119, 14
8, 119, 106, 102], 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0], 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

In [50]:

```
tokenized_train_datasets = train_dataset.map(tokenize_function)
tokenized_test_datasets = test_dataset.map(tokenize_function)
```

In [52]:

```
print(type(train_loader))
```

```
<class 'torch.utils.data.dataloader.DataLoader'>
```

In [54]:

```
print(tokenized_train_datasets)
```

```
<torch.utils.data.datapipes.map.callable.MapperMapDataPipe object at 0
x7f98d3c7abd0>
```

In [55]:

```
print(type(train_dataset))
```

```
<class 'torchtext.data.functional.to_map_style_dataset.<locals>._MapSt
yleDataset'>
```

In [56]:

```
#from transformers import TrainingArguments

training_args = TrainingArguments(output_dir="test_trainer")
```

In [59]:

```
# import torch.optim as optim

# bert = model(input_dim, embed_dim, hidden_dim, output_dim).to(device)
# bert.apply(initialize_weights)
# bert.embedding.weight.data = fast_embedding

# optimizer = optim.SGD(bert.parameters(), lr=lr)
# criterion = nn.BCEWithLogitsLoss() #combine sigmoid with binary cross entropy

# train_losses = []
# train_accs = []
# valid_losses = []
# valid_accs = []
# best_valid_loss = float('inf')


# for epoch in range(num_epochs):
#     train_loss, train_acc = train(bert, train_loader, optimizer, criterion)
#     valid_loss, valid_acc = evaluate(bert, valid_loader, criterion)

#     train_losses.append(train_loss)
#     train_accs.append(train_acc)
#     valid_losses.append(valid_loss)
#     valid_accs.append(valid_acc)

#     if valid_loss < best_valid_loss:
#         best_valid_loss = valid_loss
#         torch.save(bilstm.state_dict(), 'bert-model.pt')

#     print(f'Epoch: {epoch+1:02} | Train Loss: {train_loss:.3f} | Train Acc: {trair
#     print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')
```

In [60]:

```
# model.to(device)
# model.train()
```

In [61]:

```
from torch.optim.sgd import SGD
optim = SGD(model.parameters(), lr = 5e-5)
```

In [62]:

```
#model.eval()
```

In [63]:

```
for epoch in range(num_epochs):
    for step, batch in enumerate(train_loader):

  #for batch in train_loader:
        optim.zero_grad()
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)

        outputs = model(input_ids, attention_mask = attention_mask,labels=labels )

        loss = outputs[0]
        loss.backward()
        optim.step()

model.eval()
```

```
---------------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
 last)
<ipython-input-63-8496c87ac3b2> in <module>()
      1 for epoch in range(num_epochs):
----> 2   for step, batch in enumerate(train_loader):
      3 
      4   #for batch in train_loader:
      5     optim.zero_grad()

/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py
 in __next__(self)
    519             if self._sampler_iter is None:
    520                 self._reset()
--> 521             data = self._next_data()
    522             self._num_yielded += 1
    523             if self._dataset_kind == _DatasetKind.Iterable and
\

/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py
 in _next_data(self)
    559     def _next_data(self):
    560         index = self._next_index()  # may raise StopIteration
--> 561         data = self._dataset_fetcher.fetch(index)  # may raise
StopIteration
    562         if self._pin_memory:
    563             data = _utils.pin_memory.pin_memory(data)

/usr/local/lib/python3.7/dist-packages/torch/utils/data/_utils/fetch.p
y in fetch(self, possibly_batched_index)
     50         else:
     51             data = self.dataset[possibly_batched_index]
---> 52         return self.collate_fn(data)

<ipython-input-13-481ca85b07da> in collate_batch(batch)
      9     for (_label, _text) in batch:
     10         label_list.append(label_pipeline(_label))
---> 11         processed_text = torch.tensor(text_pipeline(_text), dt
ype=torch.int64)
     12         text_list.append(processed_text)
```

```
     13         length_list.append(processed_text.size(0))  #++<-----p
acked padded sequences require length

<ipython-input-13-481ca85b07da> in <lambda>(x)
----> 1 text_pipeline = lambda x: vocab(tokenizer(x))
      2 label_pipeline = lambda x: 1 if x == 'pos' else 0
      3
      4 from torch.utils.data import DataLoader
      5 from torch.nn.utils.rnn import pad_sequence #++

/usr/local/lib/python3.7/dist-packages/torch/nn/modules/module.py in _
call_impl(self, *input, **kwargs)
   1100         if not (self._backward_hooks or self._forward_hooks or
self._forward_pre_hooks or _global_backward_hooks
   1101                 or _global_forward_hooks or _global_forward_pr
e_hooks):
-> 1102             return forward_call(*input, **kwargs)
   1103         # Do not call functions when jit is used
   1104         full_backward_hooks, non_full_backward_hooks = [], []

/usr/local/lib/python3.7/dist-packages/torchtext/vocab/vocab.py in for
ward(self, tokens)
     32             The indices associated with a list of `tokens`.
     33         """
---> 34         return self.vocab.lookup_indices(tokens)
     35
     36     @torch.jit.export

TypeError: lookup_indices(): incompatible function arguments. The foll
owing argument types are supported:
    1. (self: torchtext._torchtext.Vocab, arg0: list) -> List[int]

Invoked with: <torchtext._torchtext.Vocab object at 0x7f995daaccf0>,
 {'input_ids': [101, 1409, 1128, 1176, 5367, 5558, 1114, 7424, 1104, 1
892, 1105, 1301, 1874, 117, 5606, 1104, 5152, 118, 13671, 4899, 1105,
 8362, 9261, 3452, 1158, 117, 13936, 7867, 3798, 4429, 1104, 4252, 166
5, 5082, 26346, 1473, 117, 1173, 1440, 6890, 119, 1409, 1128, 1176, 35
89, 117, 6601, 1183, 117, 17873, 5367, 1134, 27486, 1892, 4783, 1107,
 5010, 1104, 170, 10416, 2296, 1104, 18410, 117, 1173, 23158, 22039, 1
110, 1111, 1128, 119, 133, 9304, 120, 135, 133, 9304, 120, 135, 15255,
2365, 117, 13713, 1149, 1667, 117, 1117, 15604, 21155, 23516, 17449, 2
050, 1676, 4246, 1105, 1147, 1685, 1488, 7726, 1132, 5312, 1149, 1106,
1103, 4883, 1183, 11408, 1111, 170, 1263, 5138, 12020, 1283, 1121, 110
3, 1331, 119, 1212, 1103, 1236, 1146, 117, 1667, 4919, 170, 188, 2136
5, 1114, 1117, 1610, 119, 1109, 13202, 1150, 1125, 1151, 12137, 1103,
 10064, 1132, 1136, 17278, 1165, 1152, 1525, 1115, 1667, 1144, 2207, 1
147, 9839, 119, 1130, 2440, 117, 4167, 4993, 1174, 11151, 20579, 2274,
1122, 7572, 119, 1124, 3226, 1103, 1266, 1106, 1147, 12020, 1313, 117,
1543, 1612, 1152, 1267, 1140, 119, 1124, 21761, 1113, 1667, 1105, 424
6, 1112, 1152, 1138, 2673, 119, 1124, 8966, 1194, 1147, 3751, 1114, 11
17, 6658, 1165, 1152, 4597, 112, 189, 1313, 117, 5074, 1172, 7290, 110
3, 26858, 7996, 1107, 1147, 3751, 1105, 2928, 1165, 1152, 1862, 119, 1
332, 4246, 2274, 7726, 1106, 1103, 5557, 24612, 1107, 1411, 117, 7726,
1110, 5666, 1106, 170, 1353, ...
```

In [63]: