Codes for tasks

May 22, 2022

[ ]:
```
###task1.py
```

[ ]:
```
from torch.autograd import Variable
import torch
import random
import wandb
wandb.init(project = 'assign4', name = 'task1')
import numpy as np
import pickle
import torchtext
from collections import Counter
from torchtext.vocab import Vocab
from models import Encoder, Decoder, Seq2Seq
import pickle
import io
import math
import time
import torch.nn as nn
from tqdm import tqdm
from nltk import word_tokenize
from transformers import AutoModel, AutoTokenizer, BertTokenizerFast
from torch.nn.utils.rnn import pad_sequence
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchtext import vocab
from torchtext.data.utils import get_tokenizer
import numpy as np
import spacy
spacy_eng = spacy.load("en")


PAD_IDX_EN = 0
BOS_IDX_EN = 1
EOS_IDX_EN = 2
UNK_IDX_EN = 3
GLOVE_TEXT_PATH = '/scratch/tanay/exp/glove.6B.300d.txt'
```

```
def save_pickle(data, path):
  with open(path, 'wb') as f:
    pickle.dump(data, f)
def load_pickle(path):
  with open(path, 'rb') as f:
    return pickle.load(f)
def add_specials(vocab):
    vocab["<unk>"] = UNK_IDX_EN
    vocab["<pad>"] = PAD_IDX_EN
    vocab["<bos>"] = BOS_IDX_EN
    vocab['<eos>'] = EOS_IDX_EN
    return vocab

def save_pickle(data, path):
  with open(path, 'wb') as f:
    pickle.dump(data, f)
def load_pickle(path):
  with open(path, 'rb') as f:
    return pickle.load(f)

def load_embeds_enc(root_dir):
    embeddings_index = dict()
    f = open(root_dir)
    c =4
    for line in f:
        values = line.split()
        word = values[0]
        embeddings_index[word] = c
        c +=1
    f.close()
    return embeddings_index

en_tokenizer = get_tokenizer('spacy', language='en')
bert_model = AutoModel.from_pretrained('ai4bharat/indic-bert')
bert_tokenizer = AutoTokenizer.from_pretrained('ai4bharat/indic-bert')


train_filepaths = ['/scratch/tanay/exp/en-gu/train.en', '/scratch/tanay/exp/
↪en-gu/train.gu']
val_filepaths = ['/scratch/tanay/exp/en-gu/dev.en', '/scratch/tanay/exp/en-gu/
↪dev.gu']
test_filepaths = ['/scratch/tanay/exp/en-gu/test.en', '/scratch/tanay/exp/en-gu/
↪test.gu']

def entokenizer(text_list, tokenizer):
  if isinstance(text_list, str):
    text_list = [text_list]
```

```
  tokenized_text = []
  for text in text_list:
    ls= []
    for tok in tokenizer(text.strip()):
      ls.append(tok.lower())
    tokenized_text.append(ls)
  return tokenized_text

def gu_tokenizer(text, tokenizer):
    return tokenizer(text, add_special_tokens=False)['input_ids']


def build_vocab(filepath, lang, _tokeniz):
  vocab = {}
  c = 4
  with io.open(filepath, encoding="utf8") as f:
    data = f.readlines()
    for i in tqdm(range(0, len(data), 512), desc = f'Building vocab {lang}'):
        if lang == 'en':
          for k in entokenizer(data[i:i + 512], en_tokenizer):
            # print(k)
            # exit()
            for token in k:
              if token not in vocab:
                vocab[token] = c
                c +=1
        elif lang == 'gu':
          # print(counter)
          for k in gu_tokenizer(data[i: i + 512], _tokeniz):
            for token in k:
              if token not in vocab:
                vocab[token] = c
                c +=1
  return vocab

if False:
    gu_vocab = build_vocab(train_filepaths[1], lang = 'gu', _tokeniz =␣
 ↪bert_tokenizer)
    en_vocab = build_vocab(train_filepaths[0], lang = 'en', _tokeniz =␣
 ↪en_tokenizer)

    specials=['<unk>', '<pad>', '<bos>', '<eos>']
    en_vocab = add_specials(en_vocab)
    gu_vocab = add_specials(gu_vocab)

    save_pickle(gu_vocab, 'vocab_gu_task1.pkl')
    save_pickle(en_vocab, 'vocab_en_task1.pkl')
```

```
gu_vocab = load_pickle('vocab_gu_task1.pkl')
en_vocab = load_pickle('vocab_en_task1.pkl')



def read_data(filepaths, k = 1):
  with open(filepaths[0], 'r') as fen:
    en_data = fen.readlines()
  with open(filepaths[1],'r') as fgu:
    gu_data = fgu.readlines()
  en_data2 = []
  for i in en_data[:int(len(en_data)*k)]:
    en_data2.append(i.strip())
  gu_data2 = []
  for i in gu_data[:int(len(gu_data)*k)]:
    gu_data2.append(i.strip())
  return en_data2, gu_data2

def data_process_val_test(path, k ):
  data = [];a =0
  en_data2, gu_data2 = read_data(path, k = k)
  assert len(en_data2) == len(gu_data2), f"EN{len(en_data2)}/GU{len(gu_data2)}"
  for i in tqdm(range(0,len(en_data2), 512), desc = f'Running'):
    c =0
    en_list =[];gu_list =[]
    for en in entokenizer(en_data2[i:i +512], en_tokenizer):
 ↪#gu_tokenizer(gu_data2[i:i + 512], en_bert_tokenizer):
      tk =[]
      for token in en:
        if token in en_vocab:
          tk.append(en_vocab[token])
        else:
          a +=1
          tk.append(en_vocab["<unk>"])
      en_tensor_ = torch.tensor(tk,
                        dtype=torch.long)
      en_list.append(en_tensor_)
    for gu in gu_tokenizer(gu_data2[i:i + 512], bert_tokenizer):
      tk =[]
      for token in gu:
        if token in gu_vocab:
          tk.append(gu_vocab[token])
        else:
          c +=1
          tk.append(gu_vocab["<unk>"])
```

```
      gu_tensor_ = torch.tensor(tk,
                     dtype=torch.long)
      gu_list.append(gu_tensor_)

    assert len(en_list) == len(gu_list)
    for i,j in zip(en_list, gu_list):
      data.append((i, j))
    del en_list
    del gu_list
  print(len(data), a, c)
  return data


def data_process(path):
  data = []
  en_data2, gu_data2 = read_data(path, k = 1)
  assert len(en_data2) == len(gu_data2), f"EN{len(en_data2)}/GU{len(gu_data2)}"
  for i in tqdm(range(0,len(en_data2), 512), desc = f'Running'):
    c =0
    en_list =[];gu_list =[]
    for en in entokenizer(en_data2[i:i +512], en_tokenizer):
        en_tensor_ = torch.tensor([en_vocab[token] for token in en],
                         dtype=torch.long)
        en_list.append(en_tensor_)

    for gu in gu_tokenizer(gu_data2[i:i + 512], bert_tokenizer):
        gu_tensor_ = torch.tensor([gu_vocab[token] for token in gu],
                           dtype=torch.long)
        gu_list.append(gu_tensor_)

    assert len(en_list) == len(gu_list)
    for i,j in zip(en_list, gu_list):
      data.append((i, j))
    del en_list
    del gu_list
  print(len(data))
  return data

# train_data = data_process(train_filepaths)
# val_data  = data_process_val_test(val_filepaths, 1)
test_data  = data_process_val_test(test_filepaths, 1)

import gc
gc.collect()
```

```python
def load_embeds_dec(model, tokenizer, vocab, embed_dim= 128):
    vocab_to_embedding_convertor = model.get_input_embeddings()
    # pass the tokens to get the embeddings
    embeddings_index = {}
    for tokens in tqdm(vocab):
      try:
        embeddings = vocab_to_embedding_convertor(torch.tensor(tokens))
      except:
        print(tokens)
        embeddings = np.random.normal(scale=0.5, size=(embed_dim, ))
      embeddings_index[tokens] = embeddings

    return embeddings_index

def load_embeds_enc(root_dir):
    embeddings_index = {}
    f = open(root_dir)

    for line in f:
        values = line.split()
        word = values[0]
        coefs = np.asarray(values[1:], dtype='float32')
        embeddings_index[word] = coefs

    f.close()
    return embeddings_index

def load_embed_weights_enc(embeddings_index, embed_dim, vocab, vocab_size):
    matrix_len = vocab_size
    print("ENC", vocab_size)
    weights_matrix = np.zeros((matrix_len, embed_dim))
    words_found = 0
    for word,i in vocab.items():
        try:
            weights_matrix[i] = embeddings_index[word]
            words_found += 1
        except:
            weights_matrix[i] = np.random.normal(scale=0.5, size=(embed_dim, ))
    print(words_found/vocab_size)
    weights_matrix = torch.tensor(weights_matrix)
    return weights_matrix


def load_embed_weights_dec(embeddings_index, embed_dim, vocab, vocab_size):

    matrix_len = vocab_size
    print("DEC", vocab_size)
```

```
    weights_matrix = np.zeros((matrix_len, embed_dim))
    words_found = 0
    for word,i in tqdm(vocab.items(), desc = 'DEC'):
        try:
            weights_matrix[i] = embeddings_index[word]
            words_found += 1
        except:
            weights_matrix[i] = np.random.normal(scale=0.5, size=(embed_dim, ))
    print(words_found/vocab_size)
    weights_matrix = torch.tensor(weights_matrix)
    return weights_matrix

embeddings_index = None #load_embeds_enc(GLOVE_TEXT_PATH) #load_embeds_dec(en_bert_model, en_bert_tokenizer, en_vocab)
weights_matrix = None #load_embed_weights_enc(embeddings_index, 300, en_vocab,len(en_vocab)) #load_embed_weights_dec(embeddings_index, 128, en_vocab, len(en_vocab))

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
BATCH_SIZE = 512

embeddings_index_dec = load_embeds_dec(bert_model, bert_tokenizer, gu_vocab)
weight_matrix_dec = load_embed_weights_dec(embeddings_index_dec, 128, gu_vocab, len(gu_vocab))

print("STARTING TO CREATE DATALOADERS")
def generate_batch(data_batch, max_len = 40):
  gu_batch, en_batch = [], []
  for gu_item, en_item in data_batch:
    gu_batch.append(torch.cat([torch.tensor([BOS_IDX_EN]), gu_item[:max_len], torch.tensor([EOS_IDX_EN])], dim=0))
    en_batch.append(torch.cat([torch.tensor([BOS_IDX_EN]), en_item[:max_len], torch.tensor([EOS_IDX_EN])], dim=0))

  gu_batch = pad_sequence(gu_batch, padding_value=PAD_IDX_EN)
  en_batch = pad_sequence(en_batch, padding_value=PAD_IDX_EN)
  return gu_batch, en_batch

# train_iter = DataLoader(train_data, batch_size=BATCH_SIZE,
#                           shuffle=True, collate_fn=generate_batch)
# valid_iter = DataLoader(val_data, batch_size=BATCH_SIZE,
#                         shuffle=True, collate_fn=generate_batch)
test_iter = DataLoader(test_data, batch_size=1,
                       shuffle=False, collate_fn=generate_batch)


INPUT_DIM = len(en_vocab)
```

```
ENC_EMB_DIM = 128
DEC_EMB_DIM = 128
OUTPUT_DIM = len(gu_vocab)
ENC_LAYERS = 3
DEC_LAYERS = 3
ENC_HEADS = 8
DEC_HEADS = 8
ENC_PF_DIM = 512
DEC_PF_DIM = 512
ENC_DROPOUT = 0.1
DEC_DROPOUT = 0.1

enc = Encoder(INPUT_DIM,
              ENC_EMB_DIM,
              ENC_LAYERS,
              ENC_HEADS,
              ENC_PF_DIM,
              ENC_DROPOUT,
              device,
              weights_matrix)

dec = Decoder(OUTPUT_DIM,
              DEC_EMB_DIM,
              DEC_LAYERS,
              DEC_HEADS,
              DEC_PF_DIM,
              DEC_DROPOUT,
              device,
              weight_matrix_dec)

model = Seq2Seq(enc, dec, PAD_IDX_EN, PAD_IDX_EN, device).to(device)

params_to_update = model.parameters()
params_to_update = []
for name,param in model.named_parameters():
    if param.requires_grad == True:
        params_to_update.append(param)

optimizer = optim.Adam(params_to_update, lr = 0.001)


criterion = nn.CrossEntropyLoss(ignore_index=0)

def train(model, iterator, val_iterator, optimizer, criterion, clip):

    model.train()
```

```
    epoch_loss = 0

    for i, (src, trg)  in enumerate(iterator):

        src, trg = src.to(device), trg.to(device)

        optimizer.zero_grad()
        output, _ = model(src, trg[:-1,:])

        output_dim = output.shape[-1]

        output = output.contiguous().view(-1, output_dim)
        trg = trg[1:,:].contiguous().view(-1)

        loss = criterion(output, trg)
        wandb.log({"train_loss": loss.item(), 'step': i})

        loss.backward()

        torch.nn.utils.clip_grad_norm_(model.parameters(), clip)

        optimizer.step()

        epoch_loss += loss.item()
        if i % 1000 ==0:
          val_loss = evaluate(model, val_iterator, criterion)
          wandb.log({"val_loss": val_loss, 'step': i})



    return epoch_loss / len(iterator)

def evaluate(model, iterator, criterion):

    epoch_loss = 0
    gold =[]; inputs =[]; outputs =[]
    print("EVALUATION")
    with torch.no_grad():

        for i, (src, trg)  in enumerate(iterator):

            src, trg = src.to(device), trg.to(device)

            output, _ = model(src, trg[:-1,:])

            #output = [batch size, trg len - 1, output dim]
            #trg = [batch size, trg len]
```

```
            output_dim = output.shape[-1]

            output = output.contiguous().view(-1, output_dim)
            trg = trg[1:,:].contiguous().view(-1)

            #output = [batch size * trg len - 1, output dim]
            #trg = [batch size * trg len - 1]
            loss = criterion(output, trg)
            # output = output.argmax(dim =1)
            # print(output.shape, trg.shape)
            # outputs.append(output.cpu().numpy())
            # gold.append(trg.cpu().numpy())
            # inputs.append(src.cpu().numpy())

            epoch_loss += loss.item()
    # save_pickle(outputs, 'test_ped_val.pkl')
    # save_pickle(gold, 'test_gold_val.pkl')
    # save_pickle(inputs, 'test_sc_val.pkl')

    return epoch_loss / len(iterator)

N_EPOCHS = 10
CLIP = 2

best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):

    train_loss = train(model, train_iter, valid_iter,  optimizer, criterion,␣
 ↪CLIP)
    valid_loss = evaluate(model, valid_iter, criterion)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'task1-model.pt')
    wandb.log({"val_loss_epoch": valid_loss, "train_loss_epoch":train_loss,␣
 ↪'epoch': epoch})
    print(f'\tTrain Loss: {train_loss:.3f} | Train PPL: {math.exp(train_loss):7.
 ↪3f}')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. PPL: {math.exp(valid_loss):7.
 ↪3f}')


# TESTING

# model.load_state_dict(torch.load('task1-model.pt'))
```

```
# print('model loaded')
# model.to(device)
# model.eval()


# test_loss = evaluate(model, test_iter, criterion)

# print(f'| Test Loss: {test_loss:.3f} | Test PPL: {math.exp(test_loss):7.3f} |')
```

[ ]:
```
##models.py
```

[ ]:
```
from transformers import BertPreTrainedModel, BertModel
import torch.nn as nn
from typing import Optional, Union, Tuple
import torch

class CustomBertForQuestionAnswering(nn.Module):

    def __init__(self, config):
        super().__init__()
        self.num_labels = config.num_labels
        self.bert = BertModel.from_pretrained('bert-base-uncased', config =␣
 ↪config, add_pooling_layer=False)
        self.qa_outputs = nn.Linear(config.hidden_size, config.num_labels)


    def forward(self, input_ids, attention_mask, token_type_ids, position_ids=␣
 ↪None):

        outputs = self.bert(input_ids,
            attention_mask=attention_mask,
            token_type_ids=token_type_ids,
            position_ids=position_ids)

        sequence_output = outputs[0]
        logits = self.qa_outputs(sequence_output)
        start_logits, end_logits = logits.split(1, dim=-1)
        start_logits = start_logits.squeeze(-1).contiguous()
        end_logits = end_logits.squeeze(-1).contiguous()

        return start_logits, end_logits

    def save_pretrained(self, path = None):
        pass


class CustomBertForSequenceClassification(nn.Module):
```

```python
    def __init__(self, config):
        super().__init__()
        self.num_labels = config.num_labels
        self.config = config
        self.bert = BertModel.from_pretrained('bert-base-uncased', config =␣
↪config)
        self.dropout = nn.Dropout(0.2)
        self.classifier = nn.Linear(config.hidden_size, config.num_labels)

    def forward(self, input_ids, attention_mask, token_type_ids, position_ids=␣
↪None):
        outputs = self.bert(input_ids,
            attention_mask=attention_mask,
            token_type_ids=token_type_ids,
            position_ids=position_ids)
        pooled_output = outputs[1]
        pooled_output = self.dropout(pooled_output)
        logits = self.classifier(pooled_output)
        return logits

    def save_pretrained(self, path = None):
        pass

# PART OF THE MODEL IMPLEMNETATION WAS BORROWED FROM THE OFFICIAL TRANSFORMERS␣
↪IMPLETEMENTATION
# https://github.com/bentrevett/pytorch-seq2seq

class Encoder(nn.Module):
    def __init__(self,
                 input_dim,
                 hid_dim,
                 n_layers,
                 n_heads,
                 pf_dim,
                 dropout,
                 device,
                 weights_matrix,
                 max_length = 100):
        super().__init__()

        self.device = device

        self.tok_embedding = nn.Embedding(input_dim, hid_dim)
        #self.tok_embedding.weight.requires_grad = True
        #self.tok_embedding.load_state_dict({'weight': weights_matrix})
        self.pos_embedding = nn.Embedding(max_length, hid_dim)
```

```
        self.layers = nn.ModuleList([EncoderLayer(hid_dim,
                                                  n_heads,
                                                  pf_dim,
                                                  dropout,
                                                  device)
                                     for _ in range(n_layers)])

        self.dropout = nn.Dropout(dropout)

        self.scale = torch.sqrt(torch.FloatTensor([hid_dim])).to(device)

    def forward(self, src, src_mask):

        #src = [batch size, src len]
        #src_mask = [batch size, 1, 1, src len]
        src = src.permute((1,0))
        #print(src.shape)
        batch_size = src.shape[0]
        src_len = src.shape[1]

        pos = torch.arange(0, src_len).unsqueeze(0).repeat(batch_size, 1).
 ↪to(self.device)

        #pos = [batch size, src len]
        h = (self.tok_embedding(src) * self.scale) + self.pos_embedding(pos)

        src = self.dropout(h)

        #src = [batch size, src len, hid dim]

        for layer in self.layers:
            src = layer(src, src_mask)
        return src

class EncoderLayer(nn.Module):
    def __init__(self,
                 hid_dim,
                 n_heads,
                 pf_dim,
                 dropout,
                 device):
        super().__init__()

        self.self_attn_layer_norm = nn.LayerNorm(hid_dim)
        self.ff_layer_norm = nn.LayerNorm(hid_dim)
        self.self_attention = MultiHeadAttentionLayer(hid_dim, n_heads, dropout,␣
 ↪device)
```

```python
        self.positionwise_feedforward = PositionwiseFeedforwardLayer(hid_dim,
                                                                     pf_dim,
                                                                     dropout)
        self.dropout = nn.Dropout(dropout)

    def forward(self, src, src_mask):

        #src = [batch size, src len, hid dim]
        #src_mask = [batch size, 1, 1, src len]

        #self attention
        #print(f'Layee: {src.shape}: {src_mask.shape}')
        _src, _ = self.self_attention(src, src, src, src_mask)

        #dropout, residual connection and layer norm
        src = self.self_attn_layer_norm(src + self.dropout(_src))

        #src = [batch size, src len, hid dim]

        #positionwise feedforward
        _src = self.positionwise_feedforward(src)

        #dropout, residual and layer norm
        src = self.ff_layer_norm(src + self.dropout(_src))

        #src = [batch size, src len, hid dim]

        return src
class MultiHeadAttentionLayer(nn.Module):
    def __init__(self, hid_dim, n_heads, dropout, device):
        super().__init__()

        assert hid_dim % n_heads == 0

        self.hid_dim = hid_dim
        self.n_heads = n_heads
        self.head_dim = hid_dim // n_heads

        self.fc_q = nn.Linear(hid_dim, hid_dim)
        self.fc_k = nn.Linear(hid_dim, hid_dim)
        self.fc_v = nn.Linear(hid_dim, hid_dim)

        self.fc_o = nn.Linear(hid_dim, hid_dim)

        self.dropout = nn.Dropout(dropout)

        self.scale = torch.sqrt(torch.FloatTensor([self.head_dim])).to(device)
```

```python
    def forward(self, query, key, value, mask = None):

        #print('query', query.shape)
        batch_size = query.shape[0]

        #query = [batch size, query len, hid dim]
        #key = [batch size, key len, hid dim]
        #value = [batch size, value len, hid dim]

        Q = self.fc_q(query)
        K = self.fc_k(key)
        V = self.fc_v(value)
        #print("K", K.shape)
        #Q = [batch size, query len, hid dim]
        #K = [batch size, key len, hid dim]
        #V = [batch size, value len, hid dim]

        Q = Q.view(batch_size, -1, self.n_heads, self.head_dim).permute(0, 2, 1,␣
 ↪3)
        K = K.view(batch_size, -1, self.n_heads, self.head_dim).permute(0, 2, 1,␣
 ↪3)
        V = V.view(batch_size, -1, self.n_heads, self.head_dim).permute(0, 2, 1,␣
 ↪3)

        #Q = [batch size, n heads, query len, head dim]
        #K = [batch size, n heads, key len, head dim]
        #V = [batch size, n heads, value len, head dim]


        energy = torch.matmul(Q, K.permute(0, 1, 3, 2)) / self.scale

        #energy = [batch size, n heads, query len, key len]
        #print("energy",energy.shape)
        #print("mask", mask.shape)
        if mask is not None:
            energy = energy.masked_fill(mask == 0, -1e10)

        attention = torch.softmax(energy, dim = -1)

        #attention = [batch size, n heads, query len, key len]

        x = torch.matmul(self.dropout(attention), V)

        #x = [batch size, n heads, query len, head dim]

        x = x.permute(0, 2, 1, 3).contiguous()
```

```
        #x = [batch size, query len, n heads, head dim]

        x = x.view(batch_size, -1, self.hid_dim)

        #x = [batch size, query len, hid dim]

        x = self.fc_o(x)

        #x = [batch size, query len, hid dim]

        return x, attention

class PositionwiseFeedforwardLayer(nn.Module):
    def __init__(self, hid_dim, pf_dim, dropout):
        super().__init__()

        self.fc_1 = nn.Linear(hid_dim, pf_dim)
        self.fc_2 = nn.Linear(pf_dim, hid_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, x):

        #x = [batch size, seq len, hid dim]

        x = self.dropout(torch.relu(self.fc_1(x)))

        #x = [batch size, seq len, pf dim]

        x = self.fc_2(x)

        #x = [batch size, seq len, hid dim]

        return x

class Decoder(nn.Module):
    def __init__(self,
                 output_dim,
                 hid_dim,
                 n_layers,
                 n_heads,
                 pf_dim,
                 dropout,
                 device,
                 weights_matrix,
                 max_length = 100):
```

```
        super().__init__()

        self.device = device

        self.tok_embedding = nn.Embedding(output_dim, hid_dim)
        self.tok_embedding.weight.requires_grad = False
        self.tok_embedding.load_state_dict({'weight': weights_matrix})
        self.pos_embedding = nn.Embedding(max_length, hid_dim)

        self.layers = nn.ModuleList([DecoderLayer(hid_dim,
                                                  n_heads,
                                                  pf_dim,
                                                  dropout,
                                                  device)
                                     for _ in range(n_layers)])

        self.fc_out = nn.Linear(hid_dim, output_dim)

        self.dropout = nn.Dropout(dropout)

        self.scale = torch.sqrt(torch.FloatTensor([hid_dim])).to(device)

    def forward(self, trg, enc_src, trg_mask, src_mask):

        #trg = [batch size, trg len]
        #enc_src = [batch size, src len, hid dim]
        #trg_mask = [batch size, 1, trg len, trg len]
        #src_mask = [batch size, 1, 1, src len]
        trg = trg.permute((1,0))
        #print("trg_mask",trg_mask.shape)
        #print("trg",trg.shape)
        batch_size = trg.shape[0]
        trg_len = trg.shape[1]

        pos = torch.arange(0, trg_len).unsqueeze(0).repeat(batch_size, 1).
↪to(self.device)

        #pos = [batch size, trg len]

        trg = self.dropout((self.tok_embedding(trg) * self.scale) + self.
↪pos_embedding(pos))

        #trg = [batch size, trg len, hid dim]
        #print("dopout post trg",trg.shape)
        for layer in self.layers:
            trg, attention = layer(trg, enc_src, trg_mask, src_mask)
```

```
        #trg = [batch size, trg len, hid dim]
        #attention = [batch size, n heads, trg len, src len]

        output = self.fc_out(trg)

        #output = [batch size, trg len, output dim]

        return output, attention

class DecoderLayer(nn.Module):
    def __init__(self,
                 hid_dim,
                 n_heads,
                 pf_dim,
                 dropout,
                 device):
        super().__init__()

        self.self_attn_layer_norm = nn.LayerNorm(hid_dim)
        self.enc_attn_layer_norm = nn.LayerNorm(hid_dim)
        self.ff_layer_norm = nn.LayerNorm(hid_dim)
        self.self_attention = MultiHeadAttentionLayer(hid_dim, n_heads, dropout,␣
 ↪device)
        self.encoder_attention = MultiHeadAttentionLayer(hid_dim, n_heads,␣
 ↪dropout, device)
        self.positionwise_feedforward = PositionwiseFeedforwardLayer(hid_dim,
                                                                     pf_dim,
                                                                     dropout)
        self.dropout = nn.Dropout(dropout)

    def forward(self, trg, enc_src, trg_mask, src_mask):

        #trg = [batch size, trg len, hid dim]
        #enc_src = [batch size, src len, hid dim]
        #trg_mask = [batch size, 1, trg len, trg len]
        #src_mask = [batch size, 1, 1, src len]
        #print("***")
        #print(trg.shape)
        #print(enc_src.shape)
        #print(trg_mask.shape)
        #print(src_mask.shape)
        #print("***")
        #self attention
        _trg, _ = self.self_attention(trg, trg, trg, trg_mask)

        #dropout, residual connection and layer norm
        trg = self.self_attn_layer_norm(trg + self.dropout(_trg))
```

```
        #trg = [batch size, trg len, hid dim]

        #encoder attention
        _trg, attention = self.encoder_attention(trg, enc_src, enc_src, src_mask)

        #dropout, residual connection and layer norm
        trg = self.enc_attn_layer_norm(trg + self.dropout(_trg))

        #trg = [batch size, trg len, hid dim]

        #positionwise feedforward
        _trg = self.positionwise_feedforward(trg)

        #dropout, residual and layer norm
        trg = self.ff_layer_norm(trg + self.dropout(_trg))

        #trg = [batch size, trg len, hid dim]
        #attention = [batch size, n heads, trg len, src len]

        return trg, attention

class Seq2Seq(nn.Module):
    def __init__(self,
                 encoder,
                 decoder,
                 src_pad_idx,
                 trg_pad_idx,
                 device):
        super().__init__()

        self.encoder = encoder
        self.decoder = decoder
        self.src_pad_idx = src_pad_idx
        self.trg_pad_idx = trg_pad_idx
        self.device = device

    def make_src_mask(self, src):

        #src = [batch size, src len]

        src_mask = (src != self.src_pad_idx).unsqueeze(1).unsqueeze(2)

        #src_mask = [batch size, 1, 1, src len]
        src_mask = src_mask.permute((3,1,2,0))

        return src_mask
```

```
    def make_trg_mask(self, trg):

        #trg = [batch size, trg len]
        trg = trg.permute((1,0))
        #print('Gene tg', trg.shape)

        trg_pad_mask = (trg != self.trg_pad_idx).unsqueeze(1).unsqueeze(2)

        #trg_pad_mask = [batch size, 1, 1, trg len]

        trg_len = trg.shape[1]

        trg_sub_mask = torch.tril(torch.ones((trg_len, trg_len), device = self.
↪device)).bool()

        #trg_sub_mask = [trg len, trg len]

        trg_mask = trg_pad_mask & trg_sub_mask
        #print("generated", trg_mask.shape)
        #trg_mask = [batch size, 1, trg len, trg len]

        return trg_mask

    def forward(self, src, trg):

        #src = [batch size, src len]
        #trg = [batch size, trg len]

        src_mask = self.make_src_mask(src)
        trg_mask = self.make_trg_mask(trg)

        #src_mask = [batch size, 1, 1, src len]
        #trg_mask = [batch size, 1, trg len, trg len]
        #print(f"AT START {src_mask.shape} and {trg_mask.shape}")
        enc_src = self.encoder(src, src_mask)

        #enc_src = [batch size, src len, hid dim]

        output, attention = self.decoder(trg, enc_src, trg_mask, src_mask)

        #output = [batch size, trg len, output dim]
        #attention = [batch size, n heads, trg len, src len]

        return output, attention
```

[ ]:
```
##task2.py
```

[ ]:
```python
# Inspired from https://github.com/huggingface/transformers/tree/main/examples/
↪pytorch

import argparse
import json
import logging
import math
import os
import wandb
import random
from pathlib import Path
from models import CustomBertForSequenceClassification
import datasets
import torch.nn as nn
import torch
from datasets import load_dataset, load_metric
from torch.utils.data import DataLoader
from tqdm.auto import tqdm

import transformers
from accelerate import Accelerator
from accelerate.logging import get_logger
from accelerate.utils import set_seed
from huggingface_hub import Repository
from transformers import (
    AdamW,
    AutoConfig,
    AutoTokenizer,
    DataCollatorWithPadding,
    PretrainedConfig,
    SchedulerType,
    default_data_collator,
    get_scheduler,
)
from transformers.utils import get_full_repo_name
from transformers.utils.versions import require_version


logger = get_logger(__name__)


task_to_keys = {
    "cola": ("sentence", None),
    "mnli": ("premise", "hypothesis"),
    "mrpc": ("sentence1", "sentence2"),
    "qnli": ("question", "sentence"),
    "qqp": ("question1", "question2"),
```

```
    "rte": ("sentence1", "sentence2"),
    "sst2": ("sentence", None),
    "stsb": ("sentence1", "sentence2"),
    "wnli": ("sentence1", "sentence2"),
}


def parse_args():
    parser = argparse.ArgumentParser(description="Finetune a transformers model␣
↪on a text classification task")
    parser.add_argument(
        "--task_name",
        type=str,
        default=None,
        help="The name of the glue task to train on.",
        choices=list(task_to_keys.keys()),
    )
    parser.add_argument(
        "--train_file", type=str, default=None, help="A csv or a json file␣
↪containing the training data."
    )
    parser.add_argument(
        "--validation_file", type=str, default=None, help="A csv or a json file␣
↪containing the validation data."
    )
    parser.add_argument(
        "--max_length",
        type=int,
        default=128,
        help=(
            "The maximum total input sequence length after tokenization.␣
↪Sequences longer than this will be truncated,"
            " sequences shorter will be padded if `--pad_to_max_lengh` is passed.
↪"
        ),
    )
    parser.add_argument(
        "--pad_to_max_length",
        action="store_true",
        help="If passed, pad all samples to `max_length`. Otherwise, dynamic␣
↪padding is used.",
    )
    parser.add_argument(
        "--model_name_or_path",
        type=str,
```

```
        help="Path to pretrained model or model identifier from huggingface.co/
↪models.",
        required=True,
    )
    parser.add_argument(
        "--use_slow_tokenizer",
        action="store_true",
        help="If passed, will use a slow tokenizer (not backed by the ␣
↪Tokenizers library).",
    )
    parser.add_argument(
        "--per_device_train_batch_size",
        type=int,
        default=8,
        help="Batch size (per device) for the training dataloader.",
    )
    parser.add_argument(
        "--per_device_eval_batch_size",
        type=int,
        default=8,
        help="Batch size (per device) for the evaluation dataloader.",
    )
    parser.add_argument(
        "--learning_rate",
        type=float,
        default=5e-5,
        help="Initial learning rate (after the potential warmup period) to use.",
    )
    parser.add_argument("--weight_decay", type=float, default=0.0, help="Weight␣
↪decay to use.")
    parser.add_argument("--num_train_epochs", type=int, default=3, help="Total␣
↪number of training epochs to perform.")
    parser.add_argument(
        "--max_train_steps",
        type=int,
        default=None,
        help="Total number of training steps to perform. If provided, overrides␣
↪num_train_epochs.",
    )
    parser.add_argument(
        "--gradient_accumulation_steps",
        type=int,
        default=1,
        help="Number of updates steps to accumulate before performing a backward/
↪update pass.",
    )
```

```
    parser.add_argument(
        "--lr_scheduler_type",
        type=SchedulerType,
        default="linear",
        help="The scheduler type to use.",
        choices=["linear", "cosine", "cosine_with_restarts", "polynomial",␣
↪"constant", "constant_with_warmup"],
    )
    parser.add_argument(
        "--num_warmup_steps", type=int, default=0, help="Number of steps for the␣
↪warmup in the lr scheduler."
    )
    parser.add_argument("--output_dir", type=str, default=None, help="Where to␣
↪store the final model.")
    parser.add_argument("--seed", type=int, default=None, help="A seed for␣
↪reproducible training.")
    parser.add_argument("--push_to_hub", action="store_true", help="Whether or␣
↪not to push the model to the Hub.")
    parser.add_argument(
        "--hub_model_id", type=str, help="The name of the repository to keep in␣
↪sync with the local `output_dir`."
    )
    parser.add_argument("--hub_token", type=str, help="The token to use to push␣
↪to the Model Hub.")
    parser.add_argument(
        "--checkpointing_steps",
        type=str,
        default=100,
        help="Whether the various states should be saved at the end of every n␣
↪steps, or 'epoch' for each epoch.",
    )
    parser.add_argument(
        "--resume_from_checkpoint",
        type=str,
        default=None,
        help="If the training should continue from a checkpoint folder.",
    )
    parser.add_argument(
        "--with_tracking",
        action="store_true",
        help="Whether to load in all available experiment trackers from the␣
↪environment and use them for logging.",
    )
    parser.add_argument(
        "--ignore_mismatched_sizes",
        action="store_true",
```

```
        help="Whether or not to enable to load a pretrained model whose head␣
↪dimensions are different.",
    )
    args = parser.parse_args()

    # Sanity checks
    if args.task_name is None and args.train_file is None and args.
↪validation_file is None:
        pass
        #raise ValueError("Need either a task name or a training/validation file.
↪")
    else:
        if args.train_file is not None:
            extension = args.train_file.split(".")[-1]
            assert extension in ["csv", "json"], "`train_file` should be a csv␣
↪or a json file."
        if args.validation_file is not None:
            extension = args.validation_file.split(".")[-1]
            assert extension in ["csv", "json"], "`validation_file` should be a␣
↪csv or a json file."

    if args.push_to_hub:
        assert args.output_dir is not None, "Need an `output_dir` to create a␣
↪repo when `--push_to_hub` is passed."

    return args


def main():
    args = parse_args()

    # Initialize the accelerator. We will let the accelerator handle device␣
↪placement for us in this example.
    # If we're using tracking, we also need to initialize it here and it will␣
↪pick up all supported trackers in the environment
    accelerator = Accelerator(log_with="all", logging_dir=args.output_dir) if␣
↪args.with_tracking else Accelerator()
    # Make one log on every process with the configuration for debugging.
    logging.basicConfig(
        format="%(asctime)s - %(levelname)s - %(name)s - %(message)s",
        datefmt="%m/%d/%Y %H:%M:%S",
        level=logging.INFO,
    )
    logger.info(accelerator.state, main_process_only=False)
    if accelerator.is_local_main_process:
        datasets.utils.logging.set_verbosity_warning()
```

```
        transformers.utils.logging.set_verbosity_info()
    else:
        datasets.utils.logging.set_verbosity_error()
        transformers.utils.logging.set_verbosity_error()

    # If passed along, set the training seed now.
    if args.seed is not None:
        set_seed(args.seed)

    # Handle the repository creation
    if accelerator.is_main_process:
        if args.output_dir is not None:
            os.makedirs(args.output_dir, exist_ok=True)
    accelerator.wait_for_everyone()

    # Get the datasets: you can either provide your own CSV/JSON training and␣
↪evaluation files (see below)
    # or specify a GLUE benchmark task (the dataset will be downloaded␣
↪automatically from the datasets Hub).

    # For CSV/JSON files, this script will use as labels the column called␣
↪'label' and as pair of sentences the
    # sentences in columns called 'sentence1' and 'sentence2' if such column␣
↪exists or the first two columns not named
    # label if at least two columns are provided.

    # If the CSVs/JSONs contain only one non-label column, the script does␣
↪single sentence classification on this
    # single column. You can easily tweak this behavior (see below)

    # In distributed training, the load_dataset function guarantee that only one␣
↪local process can concurrently
    # download the dataset.
    data_files ={'train':'/scratch/tanay/exp2/Reviews.csv',
                'validation': '/scratch/tanay/exp2/Reviews.csv',
                'test': '/scratch/tanay/exp2/Reviews.csv'}

    raw_datasets = load_dataset("csv", data_files=data_files)

    def train_filter_function(example):
        return example['Id'] > 95000 and example['Id']< 99999

    def val_filter_function(example):
        return example['Id'] > 250000 and example['Id']< 251000

    def test_filter_function(example):
```

```python
        return example['Id'] > 251001 and example['Id']< 252000

    raw_datasets['train'] = raw_datasets['train'].filter(train_filter_function)
    raw_datasets['validation'] = raw_datasets['validation'].filter(val_filter_function)
    raw_datasets['test'] = raw_datasets['test'].filter(test_filter_function)

    # Labels
    label_list = list(range(1, 6))
    num_labels = len(label_list)
    sentence1_key, label_key = 'Text', 'Score'
    # Load pretrained model and tokenizer
    #
    # In distributed training, the .from_pretrained methods guarantee that only one local process can concurrently
    # download model & vocab.
    config = AutoConfig.from_pretrained(args.model_name_or_path, num_labels=len(label_list), finetuning_task=args.task_name)
    tokenizer = AutoTokenizer.from_pretrained(args.model_name_or_path, use_fast=not args.use_slow_tokenizer)
    model = CustomBertForSequenceClassification(
    config = config
    )

    # Preprocessing the datasets
    label_to_id = {v: i for i, v in enumerate(label_list)}
    model.config.label2id = label_to_id
    model.config.id2label = {id: label for label, id in label_to_id.items()}
    max_seq_length = 128
    padding = "max_length"

    def preprocess_function(examples):
        # Tokenize the texts
        texts = (
            (examples[sentence1_key],)
        )
        result = tokenizer(*texts, padding=padding, max_length=max_seq_length, truncation=True)

        result["label"] = [(label_to_id[l] if l != -1 else -1) for l in examples[label_key]]
        return result

    with accelerator.main_process_first():
        processed_datasets = raw_datasets.map(
            preprocess_function,
```

```
            batched=True,
            remove_columns=raw_datasets["train"].column_names,
            desc="Running tokenizer on dataset",
        )

    train_dataset = processed_datasets["train"]
    eval_dataset = processed_datasets["validation"]
    test_dataset = processed_datasets['test']

    # Log a few random samples from the training set:
    for index in random.sample(range(len(train_dataset)), 3):
        logger.info(f"Sample {index} of the training set: {train_dataset[index]}.
↪")

    # DataLoaders creation:
    if args.pad_to_max_length:
        # If padding was already done ot max length, we use the default data␣
↪collator that will just convert everything
        # to tensors.
        data_collator = default_data_collator
    else:
        # Otherwise, `DataCollatorWithPadding` will apply dynamic padding for us␣
↪(by padding to the maximum length of
        # the samples passed). When using mixed precision, we add␣
↪`pad_to_multiple_of=8` to pad all tensors to multiple
        # of 8s, which will enable the use of Tensor Cores on NVIDIA hardware␣
↪with compute capability >= 7.5 (Volta).
        data_collator = DataCollatorWithPadding(tokenizer, pad_to_multiple_of=(8␣
↪if accelerator.use_fp16 else None))

    train_dataloader = DataLoader(
        train_dataset, shuffle=True, collate_fn=data_collator, batch_size=args.
↪per_device_train_batch_size
    )
    eval_dataloader = DataLoader(eval_dataset, collate_fn=data_collator,␣
↪batch_size=args.per_device_eval_batch_size)

    test_dataloader = DataLoader(test_dataset, shuffle=False,␣
↪collate_fn=data_collator, batch_size=args.per_device_eval_batch_size)
    # Optimizer
    # Split weights in two groups, one with weight decay and the other not.
    no_decay = ["bias", "LayerNorm.weight"]
    optimizer_grouped_parameters = [
        {
            "params": [p for n, p in model.named_parameters() if not any(nd in n␣
↪for nd in no_decay)],
```

```
            "weight_decay": args.weight_decay,
        },
        {
            "params": [p for n, p in model.named_parameters() if any(nd in n for␣
↪nd in no_decay)],
            "weight_decay": 0.0,
        },
    ]
    optimizer = AdamW(optimizer_grouped_parameters, lr=args.learning_rate)

    # Scheduler and math around the number of training steps.
    num_update_steps_per_epoch = math.ceil(len(train_dataloader) / args.
↪gradient_accumulation_steps)
    if args.max_train_steps is None:
        args.max_train_steps = args.num_train_epochs * num_update_steps_per_epoch
    else:
        args.num_train_epochs = math.ceil(args.max_train_steps /␣
↪num_update_steps_per_epoch)

    lr_scheduler = get_scheduler(
        name=args.lr_scheduler_type,
        optimizer=optimizer,
        num_warmup_steps=args.num_warmup_steps,
        num_training_steps=args.max_train_steps,
    )

    # Prepare everything with our `accelerator`.
    model, optimizer, train_dataloader, eval_dataloader, test_dataloader,␣
↪lr_scheduler = accelerator.prepare(
        model, optimizer, train_dataloader, eval_dataloader, test_dataloader,␣
↪lr_scheduler
    )

    # We need to recalculate our total training steps as the size of the␣
↪training dataloader may have changed.
    num_update_steps_per_epoch = math.ceil(len(train_dataloader) / args.
↪gradient_accumulation_steps)
    args.max_train_steps = args.num_train_epochs * num_update_steps_per_epoch

    # Figure out how many steps we should save the Accelerator states
    checkpointing_steps = 500

    # We need to initialize the trackers we use, and also store our configuration
    if args.with_tracking:
        experiment_config = vars(args)
        # TensorBoard cannot log Enums, need the raw value
```

```
        experiment_config["lr_scheduler_type"] =␣
↪experiment_config["lr_scheduler_type"].value
        accelerator.init_trackers("glue_no_trainer", experiment_config)

    # Get the metric function
    if args.task_name is not None:
        metric = load_metric("glue", args.task_name)
    else:
        metric = load_metric("accuracy")

    # Train!
    total_batch_size = args.per_device_train_batch_size * accelerator.
↪num_processes * args.gradient_accumulation_steps

    logger.info("***** Running training *****")
    logger.info(f"  Num examples = {len(train_dataset)}")
    logger.info(f"  Num Epochs = {args.num_train_epochs}")
    logger.info(f"  Instantaneous batch size per device = {args.
↪per_device_train_batch_size}")
    logger.info(f"  Total train batch size (w. parallel, distributed &␣
↪accumulation) = {total_batch_size}")
    logger.info(f"  Gradient Accumulation steps = {args.
↪gradient_accumulation_steps}")
    logger.info(f"  Total optimization steps = {args.max_train_steps}")
    # Only show the progress bar once on each machine.
    progress_bar = tqdm(range(args.max_train_steps), disable=not accelerator.
↪is_local_main_process)
    completed_steps = 0
    starting_epoch = 0
    loss_fct = nn.CrossEntropyLoss()
    for epoch in range(starting_epoch, args.num_train_epochs):
        model.train()
        if args.with_tracking:
            total_loss = 0
        for step, batch in enumerate(train_dataloader):
            # We need to skip steps until we reach the resumed step
            if args.resume_from_checkpoint and epoch == starting_epoch:
                if resume_step is not None and step < resume_step:
                    completed_steps += 1
                    continue
            input_ids, attention_mask, token_type_ids = batch.input_ids, batch.
↪attention_mask, batch.token_type_ids
            logits = model(input_ids, attention_mask, token_type_ids)
            labels = batch.labels
            loss = loss_fct(logits.view(-1, num_labels), labels.view(-1))
            # We keep track of the loss at each epoch
```

```
            if args.with_tracking:
                total_loss += loss.detach().float()
            loss = loss / args.gradient_accumulation_steps
            if accelerator.is_main_process:
                wandb.log(
                    {
                        'train_loss': loss,
                        'step': step
                    }
                )
            accelerator.backward(loss)
            if step % args.gradient_accumulation_steps == 0 or step == len(train_dataloader) - 1:
                optimizer.step()
                lr_scheduler.step()
                optimizer.zero_grad()
                progress_bar.update(1)
                completed_steps += 1

            if isinstance(checkpointing_steps, int):
                if completed_steps % checkpointing_steps == 0:
                    output_dir = f"step_{completed_steps }"
                    if args.output_dir is not None:
                        output_dir = os.path.join(args.output_dir, output_dir)
                    accelerator.save_state(output_dir)

            if completed_steps >= args.max_train_steps:
                break
        if accelerator.is_main_process:
            wandb.log({'train_loss_epoch': total_loss, 'epoch': epoch})

        model.eval()
        samples_seen = 0
        total_val_loss =0
        for step, batch in enumerate(eval_dataloader):
            with torch.no_grad():
                input_ids, attention_mask, token_type_ids = batch.input_ids, batch.attention_mask, batch.token_type_ids
                logits = model(input_ids, attention_mask, token_type_ids)
                labels = batch.labels
                loss = loss_fct(logits.view(-1, num_labels), labels.view(-1))
                total_val_loss += loss.detach().float()
            predictions = logits.argmax(dim=-1)
            predictions, references = accelerator.gather((predictions, batch["labels"]))
            # If we are in a multiprocess environment, the last batch has duplicates
```

```
            if accelerator.num_processes > 1:
                if step == len(eval_dataloader):
                    predictions = predictions[: len(eval_dataloader.dataset) -␣
↪samples_seen]
                    references = references[: len(eval_dataloader.dataset) -␣
↪samples_seen]
                else:
                    samples_seen += references.shape[0]
            metric.add_batch(
                predictions=predictions,
                references=references,
            )
        if accelerator.is_main_process:
            wandb.log({'val_loss': total_val_loss/len(eval_dataloader), 'epoch':␣
↪epoch})

        eval_metric = metric.compute()
        logger.info(f"epoch {epoch}: {eval_metric} \t val_loss: {total_val_loss/
↪len(eval_dataloader)}")

    # PREDICTION


    model.eval()
    samples_seen = 0
    for step, batch in enumerate(test_dataloader):
        with torch.no_grad():
            input_ids, attention_mask, token_type_ids = batch.input_ids, batch.
↪attention_mask, batch.token_type_ids
            logits = model(input_ids, attention_mask, token_type_ids)
        predictions = logits.argmax(dim=-1)
        predictions, references = accelerator.gather((predictions,␣
↪batch["labels"]))
        # If we are in a multiprocess environment, the last batch has duplicates
        if accelerator.num_processes > 1:
            if step == len(eval_dataloader):
                predictions = predictions[: len(eval_dataloader.dataset) -␣
↪samples_seen]
                references = references[: len(eval_dataloader.dataset) -␣
↪samples_seen]
            else:
                samples_seen += references.shape[0]
        metric.add_batch(
            predictions=predictions,
            references=references,
        )
```

```
    # with open(os.path.join(args.output_dir, "ouputs.json"), 'w') as f:
    #     for i,j in zip(predictions,references):
    #         ls ={
    #             'predictions': i.cpu().numpy()[0],
    #             'references': j.cpu().numpy()[0]
    #         }
    #         json.dump(ls, f)
    #         f.write("\n")

    test_metric = metric.compute()

    if args.with_tracking:
        logger.info(
                f"test_accuracy: {test_metric}"
        )


    if args.output_dir is not None:
        accelerator.wait_for_everyone()
        unwrapped_model = accelerator.unwrap_model(model)
        unwrapped_model.save_pretrained(
            path = args.output_dir
        )
        if accelerator.is_main_process:
            tokenizer.save_pretrained(args.output_dir)
            if args.push_to_hub:
                repo.push_to_hub(commit_message="End of training",␣
↪auto_lfs_prune=True)

    if args.output_dir is not None:
        with open(os.path.join(args.output_dir, "all_results.json"), "w") as f:
            json.dump({"test_accuracy": test_metric["accuracy"]}, f)


if __name__ == "__main__":
    wandb.init(project = 'assign4', name = 'senti')
    main()
```

[ ]:
```
##task3.py
```

[ ]:
```
import argparse
import json
import logging
import math
from models import CustomBertForQuestionAnswering
import os
import random
```

```python
from pathlib import Path
import torch.nn as nn
import datasets
import numpy as np
import torch
from datasets import load_dataset, load_metric
from torch.utils.data import DataLoader
from tqdm.auto import tqdm
import wandb
import transformers
from accelerate import Accelerator
from accelerate.logging import get_logger
from accelerate.utils import set_seed
from huggingface_hub import Repository
from transformers import (
    AdamW,
    AutoConfig,
    AutoTokenizer,
    DataCollatorWithPadding,
    EvalPrediction,
    SchedulerType,
    default_data_collator,
    get_scheduler,
)
from utils_qa import *



logger = get_logger(__name__)


def save_prefixed_metrics(results, output_dir, file_name: str = "all_results.
 ↪json", metric_key_prefix: str = "eval"):
    """
    Save results while prefixing metric names.

    Args:
        results: (:obj:`dict`):
            A dictionary of results.
        output_dir: (:obj:`str`):
            An output directory.
        file_name: (:obj:`str`, `optional`, defaults to :obj:`all_results.json`):
            An output file name.
        metric_key_prefix: (:obj:`str`, `optional`, defaults to :obj:`eval`):
            A metric name prefix.
    """
    # Prefix all keys with metric_key_prefix + '_'
```

```
    for key in list(results.keys()):
        if not key.startswith(f"{metric_key_prefix}_"):
            results[f"{metric_key_prefix}_{key}"] = results.pop(key)

    with open(os.path.join(output_dir, file_name), "w") as f:
        json.dump(results, f, indent=4)


def parse_args():
    parser = argparse.ArgumentParser(description="Finetune a transformers model␣
 ↪on a Question Answering task")
    parser.add_argument(
        "--train_file", type=str, default=None, help="A csv or a json file␣
 ↪containing the training data."
    )
    parser.add_argument("--do_predict", action="store_true", help="To do␣
 ↪prediction on the question answering model")
    parser.add_argument(
        "--validation_file", type=str, default=None, help="A csv or a json file␣
 ↪containing the validation data."
    )
    parser.add_argument(
        "--test_file", type=str, default=None, help="A csv or a json file␣
 ↪containing the Prediction data."
    )
    parser.add_argument(
        "--max_seq_length",
        type=int,
        default=256,
        help=(
            "The maximum total input sequence length after tokenization.␣
 ↪Sequences longer than this will be truncated,"
            " sequences shorter will be padded if `--pad_to_max_lengh` is passed.
 ↪"
        ),
    )
    parser.add_argument(
        "--model_name_or_path",
        type=str,
        help="Path to pretrained model or model identifier from huggingface.co/
 ↪models.",
        required=True,
    )
    parser.add_argument(
        "--per_device_train_batch_size",
        type=int,
```

```
        default=8,
        help="Batch size (per device) for the training dataloader.",
    )
    parser.add_argument(
        "--per_device_eval_batch_size",
        type=int,
        default=8,
        help="Batch size (per device) for the evaluation dataloader.",
    )
    parser.add_argument(
        "--learning_rate",
        type=float,
        default=5e-5,
        help="Initial learning rate (after the potential warmup period) to use.",
    )
    parser.add_argument("--weight_decay", type=float, default=0.0, help="Weight
 decay to use.")
    parser.add_argument("--num_train_epochs", type=int, default=3, help="Total
 number of training epochs to perform.")
    parser.add_argument(
        "--max_train_steps",
        type=int,
        default=None,
        help="Total number of training steps to perform. If provided, overrides
 num_train_epochs.",
    )
    parser.add_argument("--output_dir", type=str, default=None, help="Where to
 store the final model.")
    parser.add_argument("--seed", type=int, default=None, help="A seed for
 reproducible training.")
    parser.add_argument(
        "--with_tracking",
        action="store_true",
        help="Whether to load in all available experiment trackers from the
 environment and use them for logging.",
    )
    args = parser.parse_args()

    return args


def main():
    args = parse_args()

    accelerator = Accelerator(log_with="all", logging_dir=args.output_dir) if
 args.with_tracking else Accelerator()
```

```python
    # Make one log on every process with the configuration for debugging.
    logging.basicConfig(
        format="%(asctime)s - %(levelname)s - %(name)s - %(message)s",
        datefmt="%m/%d/%Y %H:%M:%S",
        level=logging.INFO,
    )
    logger.info(accelerator.state, main_process_only=False)
    # If passed along, set the training seed now.
    if args.seed is not None:
        set_seed(args.seed)

    # Handle the repository creation
    if accelerator.is_main_process:
        if args.output_dir is not None:
            os.makedirs(args.output_dir, exist_ok=True)
    accelerator.wait_for_everyone()


    data_files ={
    'train':'qa_train_data_clean.json',
    'validation': 'qa_val_data_clean.json',
    'test': 'qa_test_data_clean.json'
    }

    raw_datasets = load_dataset("json", data_files=data_files)


    config = AutoConfig.from_pretrained(args.model_name_or_path)
    tokenizer = AutoTokenizer.from_pretrained(args.model_name_or_path,␣
↪use_fast=True)
    model = CustomBertForQuestionAnswering(config = config)

    # Preprocessing the datasets.

    column_names = raw_datasets["train"].column_names
    question_column_name = "question"
    context_column_name = "context"
    answer_column_name = "answers"

    # Padding side determines if we do (question|context) or (context|question).
    pad_on_right = tokenizer.padding_side == "right"
    max_seq_length = args.max_seq_length

    # Training preprocessing

    train_dataset = raw_datasets["train"]
```

```
    with accelerator.main_process_first():
        train_dataset = train_dataset.map(
            prepare_train_features,
            batched=True,
            num_proc=args.preprocessing_num_workers,
            remove_columns=column_names,
            load_from_cache_file=not args.overwrite_cache,
            desc="Running tokenizer on train dataset",
        )

    # Validation preprocessing


    eval_examples = raw_datasets["validation"]

    with accelerator.main_process_first():
        eval_dataset = eval_examples.map(
            prepare_train_features,
            batched=True,
            num_proc=args.preprocessing_num_workers,
            remove_columns=column_names,
            load_from_cache_file=not args.overwrite_cache,
            desc="Running tokenizer on validation dataset",
        )


    predict_examples = raw_datasets["test"]

    # Predict Feature Creation
    with accelerator.main_process_first():
        predict_dataset = predict_examples.map(
            prepare_validation_features,
            batched=True,
            num_proc=args.preprocessing_num_workers,
            remove_columns=column_names,
            load_from_cache_file=not args.overwrite_cache,
            desc="Running tokenizer on prediction dataset",
        )


    # DataLoaders creation:

    data_collator = DataCollatorWithPadding(tokenizer, pad_to_multiple_of=(8 if␣
↪accelerator.use_fp16 else None))

    train_dataloader = DataLoader(
```

```
        train_dataset, shuffle=True, collate_fn=data_collator, batch_size=args.
↪per_device_train_batch_size
    )

    eval_dataloader = DataLoader(
        eval_dataset, collate_fn=data_collator, batch_size=args.
↪per_device_eval_batch_size
    )

    predict_dataset_for_model = predict_dataset.remove_columns(["example_id",␣
↪"offset_mapping"])

    predict_dataloader = DataLoader(
            predict_dataset_for_model, collate_fn=data_collator, batch_size=args.
↪per_device_eval_batch_size)

    # Post-processing:
    def post_processing_function(examples, features, predictions, stage="eval"):
        # Post-processing: we match the start logits and end logits to answers␣
↪in the original context.
        predictions = postprocess_qa_predictions(
            examples=examples,
            features=features,
            predictions=predictions,
            version_2_with_negative=args.version_2_with_negative,
            n_best_size=args.n_best_size,
            max_answer_length=args.max_answer_length,
            null_score_diff_threshold=args.null_score_diff_threshold,
            output_dir=args.output_dir,
            prefix=stage,
        )
        # Format the result to the format the metric expects.
        if args.version_2_with_negative:
            formatted_predictions = [
                {"id": k, "prediction_text": v, "no_answer_probability": 0.0}␣
↪for k, v in predictions.items()
            ]
        else:
            formatted_predictions = [{"id": k, "prediction_text": v} for k, v in␣
↪predictions.items()]

        references = [{"id": ex["id"], "answers": ex[answer_column_name]} for ex␣
↪in examples]
        return EvalPrediction(predictions=formatted_predictions,␣
↪label_ids=references)
```

```python
    metric = load_metric("squad_v2" if args.version_2_with_negative else "squad")

    # Create and fill numpy array of size len_of_validation_data *␣
↪max_length_of_output_tensor
    def create_and_fill_np_array(start_or_end_logits, dataset, max_len):
        """
        Create and fill numpy array of size len_of_validation_data *␣
↪max_length_of_output_tensor

        Args:
            start_or_end_logits(:obj:`tensor`):
                This is the output predictions of the model. We can only enter␣
↪either start or end logits.
            eval_dataset: Evaluation dataset
            max_len(:obj:`int`):
                The maximum length of the output tensor. ( See the model.eval()␣
↪part for more details )
        """

        step = 0
        # create a numpy array and fill it with -100.
        logits_concat = np.full((len(dataset), max_len), -100, dtype=np.float64)
        # Now since we have create an array now we will populate it with the␣
↪outputs gathered using accelerator.gather
        for i, output_logit in enumerate(start_or_end_logits):  # populate␣
↪columns
            # We have to fill it such that we have to take the whole tensor and␣
↪replace it on the newly created array
            # And after every iteration we have to change the step

            batch_size = output_logit.shape[0]
            cols = output_logit.shape[1]

            if step + batch_size < len(dataset):
                logits_concat[step : step + batch_size, :cols] = output_logit
            else:
                logits_concat[step:, :cols] = output_logit[: len(dataset) - step]

            step += batch_size

        return logits_concat

    # Optimizer
    # Split weights in two groups, one with weight decay and the other not.
    no_decay = ["bias", "LayerNorm.weight"]
    optimizer_grouped_parameters = [
```

```
        {
            "params": [p for n, p in model.named_parameters() if not any(nd in n␣
↪for nd in no_decay)],
            "weight_decay": args.weight_decay,
        },
        {
            "params": [p for n, p in model.named_parameters() if any(nd in n for␣
↪nd in no_decay)],
            "weight_decay": 0.0,
        },
    ]
    optimizer = AdamW(optimizer_grouped_parameters, lr=args.learning_rate)

    # Scheduler and math around the number of training steps.
    num_update_steps_per_epoch = math.ceil(len(train_dataloader) / args.
↪gradient_accumulation_steps)
    if args.max_train_steps is None:
        args.max_train_steps = args.num_train_epochs * num_update_steps_per_epoch
    else:
        args.num_train_epochs = math.ceil(args.max_train_steps /␣
↪num_update_steps_per_epoch)

    lr_scheduler = get_scheduler(
        name=args.lr_scheduler_type,
        optimizer=optimizer,
        num_warmup_steps=args.num_warmup_steps,
        num_training_steps=args.max_train_steps,
    )

    # Prepare everything with our `accelerator`.
    model, optimizer, train_dataloader, eval_dataloader,predict_dataloader,␣
↪lr_scheduler = accelerator.prepare(
        model, optimizer, train_dataloader, eval_dataloader,␣
↪predict_dataloader,lr_scheduler
    )

    # We need to recalculate our total training steps as the size of the␣
↪training dataloader may have changed.
    num_update_steps_per_epoch = math.ceil(len(train_dataloader) / args.
↪gradient_accumulation_steps)
    args.max_train_steps = args.num_train_epochs * num_update_steps_per_epoch

    # Figure out how many steps we should save the Accelerator states
    if hasattr(args.checkpointing_steps, "isdigit"):
        checkpointing_steps = args.checkpointing_steps
        if args.checkpointing_steps.isdigit():
```

```
            checkpointing_steps = int(args.checkpointing_steps)
    else:
        checkpointing_steps = None

    # We need to initialize the trackers we use, and also store our configuration
    if args.with_tracking:
        experiment_config = vars(args)
        # TensorBoard cannot log Enums, need the raw value
        experiment_config["lr_scheduler_type"] =␣
↪experiment_config["lr_scheduler_type"].value
        accelerator.init_trackers("qa_no_trainer", experiment_config)

    # Train!
    total_batch_size = args.per_device_train_batch_size * accelerator.
↪num_processes * args.gradient_accumulation_steps

    logger.info("***** Running training *****")
    logger.info(f"  Num examples = {len(train_dataset)}")
    logger.info(f"  Num Epochs = {args.num_train_epochs}")
    logger.info(f"  Instantaneous batch size per device = {args.
↪per_device_train_batch_size}")
    logger.info(f"  Total train batch size (w. parallel, distributed &␣
↪accumulation) = {total_batch_size}")
    logger.info(f"  Gradient Accumulation steps = {args.
↪gradient_accumulation_steps}")
    logger.info(f"  Total optimization steps = {args.max_train_steps}")

    # Only show the progress bar once on each machine.
    progress_bar = tqdm(range(args.max_train_steps), disable=not accelerator.
↪is_local_main_process)
    completed_steps = 0
    starting_epoch = 0
    for epoch in range(starting_epoch, args.num_train_epochs):
        model.train()
        if args.with_tracking:
            total_loss = 0
        for step, batch in enumerate(train_dataloader):
            input_ids, attention_mask, token_type_ids = batch.input_ids, batch.
↪attention_mask, batch.token_type_ids
            start_logits, end_logits = model(input_ids, attention_mask,␣
↪token_type_ids)
            start_positions = batch.start_positions
            end_positions = batch.end_positions
            if start_positions is not None and end_positions is not None:
                # If we are on multi-GPU, split add a dimension
                if len(start_positions.size()) > 1:
```

```
                start_positions = start_positions.squeeze(-1)
            if len(end_positions.size()) > 1:
                end_positions = end_positions.squeeze(-1)
            # sometimes the start/end positions are outside our model␣
↪inputs, we ignore these terms
            ignored_index = start_logits.size(1)
            loss_fct = nn.CrossEntropyLoss(ignore_index=ignored_index)
            start_positions = start_positions.clamp(0, ignored_index)
            end_positions = end_positions.clamp(0, ignored_index)
            start_loss = loss_fct(start_logits, start_positions)
            end_loss = loss_fct(end_logits, end_positions)
            total_loss = (start_loss + end_loss) / 2
        loss = total_loss
        # We keep track of the loss at each epoch
        if args.with_tracking:
            total_loss += loss.detach().float()
        loss = loss / args.gradient_accumulation_steps
        if accelerator.is_main_process:
            wandb.log(
                {
                    'train_loss_step': loss,
                    'step': step
                }
            )
        accelerator.backward(loss)
        if step % args.gradient_accumulation_steps == 0 or step ==␣
↪len(train_dataloader) - 1:
            optimizer.step()
            lr_scheduler.step()
            optimizer.zero_grad()
            progress_bar.update(1)
            completed_steps += 1

        if isinstance(checkpointing_steps, int):
            if completed_steps % checkpointing_steps == 0:
                output_dir = f"step_{completed_steps }"
                if args.output_dir is not None:
                    output_dir = os.path.join(args.output_dir, output_dir)
                accelerator.save_state(output_dir)

        if completed_steps >= args.max_train_steps:
            break

    if args.checkpointing_steps == "epoch":
        output_dir = f"epoch_{epoch}"
        if args.output_dir is not None:
            output_dir = os.path.join(args.output_dir, output_dir)
```

```
            accelerator.save_state(output_dir)

        if accelerator.is_main_process:
            wandb.log(
                {
                    'train_loss_epoch': total_loss,
                    'epoch': epoch
                }
            )
        # EVALUATION
        model.eval()
        for step, batch in enumerate(eval_dataloader):
            with torch.no_grad():
                input_ids, attention_mask, token_type_ids = batch.input_ids,␣
↪batch.attention_mask, batch.token_type_ids
                start_logits, end_logits = model(input_ids, attention_mask,␣
↪token_type_ids)
                start_positions = batch.start_positions
                end_positions = batch.end_positions
                if start_positions is not None and end_positions is not None:
                    # If we are on multi-GPU, split add a dimension
                    if len(start_positions.size()) > 1:
                        start_positions = start_positions.squeeze(-1)
                    if len(end_positions.size()) > 1:
                        end_positions = end_positions.squeeze(-1)
                    # sometimes the start/end positions are outside our model␣
↪inputs, we ignore these terms
                    ignored_index = start_logits.size(1)
                    start_positions = start_positions.clamp(0, ignored_index)
                    end_positions = end_positions.clamp(0, ignored_index)
                    start_loss = loss_fct(start_logits, start_positions)
                    end_loss = loss_fct(end_logits, end_positions)
                    total_loss = (start_loss + end_loss) / 2

        if accelerator.is_main_process:
            wandb.log(
                {
                    'val_loss': total_loss,
                    'epoch': epoch
                }
            )


    # Prediction
    logger.info("***** Running Prediction *****")
    logger.info(f"  Num examples = {len(predict_dataset)}")
    logger.info(f"  Batch size = {args.per_device_eval_batch_size}")
```

```
    all_start_logits = []
    all_end_logits = []

    model.eval()

    for step, batch in enumerate(predict_dataloader):
        with torch.no_grad():
            input_ids, attention_mask, token_type_ids = batch.input_ids, batch.
↪attention_mask, batch.token_type_ids
            start_logits, end_logits = model(input_ids, attention_mask,␣
↪token_type_ids)
            if not args.pad_to_max_length:  # necessary to pad predictions and␣
↪labels for being gathered
                start_logits = accelerator.pad_across_processes(start_logits,␣
↪dim=1, pad_index=-100)
                end_logits = accelerator.pad_across_processes(end_logits, dim=1,␣
↪pad_index=-100)

            all_start_logits.append(accelerator.gather(start_logits).cpu().
↪numpy())
            all_end_logits.append(accelerator.gather(end_logits).cpu().numpy())

    max_len = max([x.shape[1] for x in all_start_logits])  # Get the max_length␣
↪of the tensor
    # concatenate the numpy array
    start_logits_concat = create_and_fill_np_array(all_start_logits,␣
↪predict_dataset, max_len)
    end_logits_concat = create_and_fill_np_array(all_end_logits,␣
↪predict_dataset, max_len)

    # delete the list of numpy arrays
    del all_start_logits
    del all_end_logits

    outputs_numpy = (start_logits_concat, end_logits_concat)
    prediction = post_processing_function(predict_examples, predict_dataset,␣
↪outputs_numpy)
    predict_metric = metric.compute(predictions=prediction.predictions,␣
↪references=prediction.label_ids)
    logger.info(f"Predict metrics: {predict_metric}")

    if args.with_tracking:
        log = {
            "squad_v2" if args.version_2_with_negative else "squad":␣
↪predict_metric,
```

```
        }

    log["squad_v2_predict" if args.version_2_with_negative else "squad_predict"]␣
↪= predict_metric

    accelerator.log(log)

    if args.output_dir is not None:
        accelerator.wait_for_everyone()
        unwrapped_model = accelerator.unwrap_model(model)
        unwrapped_model.save_pretrained(
            args.output_dir, is_main_process=accelerator.is_main_process,␣
↪save_function=accelerator.save
        )
        if accelerator.is_main_process:
            tokenizer.save_pretrained(args.output_dir)
            logger.info(json.dumps(eval_metric, indent=4))
            save_prefixed_metrics(eval_metric, args.output_dir)


if __name__ == "__main__":
    wandb.init(project = 'assign4', name = 'qa')
    main()
```