```
# -*- coding: utf-8 -*-
"""
Created on Sun Apr 25 19:55:35 2021

@author: GS63
"""

import os
import json
# import glob
from tqdm import tqdm
import torch
import random
import numpy as np
from torch.utils.data import (DataLoader, RandomSampler, SequentialSampler,
TensorDataset)
from torch.utils.data.distributed import DistributedSampler
#from transformers import BertTokenizer


#from params import MCTest_test_file_path, MCTest_testAns_file_path
from params import datasetsDREAM, datasetsRACE, datasetsMCTEST
from params import race_raw_dev_path, race_raw_test_path, race_raw_train_path
from pytorch_pretrained_bert.modeling import BertConfig, BertForMultipleChoice
from pytorch_pretrained_bert.file_utils import PYTORCH_PRETRAINED_BERT_CACHE,
WEIGHTS_NAME, CONFIG_NAME
from pytorch_pretrained_bert.optimization import BertAdam, WarmupLinearSchedule
from pytorch_pretrained_bert.tokenization import BertTokenizer


class Race1(object):
    """We are going to train race dataset with bert."""
    def __init__(self,
                 race_id,
                 context_sentence,
                 start_ending,
                 ending_0,
                 ending_1,
                 ending_2,
                 ending_3,
                 label = None):
        self.race_id = race_id
        self.context_sentence = context_sentence # sentance
        self.start_ending = start_ending  # Question
        self.endings = [
            ending_0,
            ending_1,
            ending_2,
            ending_3,
        ]
```

```
        self.label = label

    def __str__(self):
        return self.__repr__()

    def __repr__(self):
        l = [
            "race_id: {}".format(self.race_id),
            "context_sentence: {}".format(self.context_sentence),
            "start_ending: {}".format(self.start_ending),
            "ending_0: {}".format(self.endings[0]),
            "ending_1: {}".format(self.endings[1]),
            "ending_2: {}".format(self.endings[2]),
            "ending_3: {}".format(self.endings[3]),
        ]

        if self.label is not None:
            l.append("label: {}".format(self.label))

        return ", ".join(l)


class InputFeatures1(object):
    def __init__(self,
                 example_id,
                 choices_features,
                 label

    ):
        self.example_id = example_id
        self.choices_features = [
            {
                'input_ids': input_ids,
                'input_mask': input_mask,
                'segment_ids': segment_ids
            }
            for _, input_ids, input_mask, segment_ids in choices_features
        ]
        self.label = label


class MCQATrainModel:
    #set Trains
    ansLabel_map = {"A": 0, "B": 1, "C": 2, "D": 3}
    #define difficulty
    difficulty_set = ["middle", "high"]
    #define dataset type
    dataset_type = ["train", "dev", "test"]

    max_seq_length = 450 #512#256
```

```
    gradient_accumulation_steps = 3
    train_batch_size = 6
    eval_batch_size = 3
    test_batch_size = 3
    numEpoch = 2
    maxNumFileForTestDirectory = 200  # for both middle/ high directory
    maxNumFileforTrainDirectory = 3000 # for both middle/ high directory
    maxNumFileForEvalDirectory = 200  # for both middle/ high directory
    warmup_proportion= 0.1
    learning_rate = 5e-5
    seed= 42
    trainedRaceModelFile = "raceTrainedModel.sav"
    # # setup GPU/CPU
    device = torch.device('cuda') if torch.cuda.is_available() else
torch.device('cpu')
    tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')


    train_batch_size = train_batch_size // gradient_accumulation_steps


    def __init__(self):
        self = self
        # self.dataset = dataset


    def loadRaceJsonDataFile(self, fileName):
        with open(fileName, 'r', encoding="utf-8") as f:
            data =  json.loads(f.read())
            return data["id"], data["article"], data["questions"], data["options"],
data["answers"]


    def scanFileList(self, path):
        fileList = []
        for file in os.listdir(path):
            # print(file)
            fileList.append(os.path.join(path, file))
        return fileList


    def read_raceModify(self, input_dir, maxSentence):
        samples = []
        data_grade = ["middle","high"]
        for grade in data_grade:
            print("level", grade)
            dir_name = input_dir + grade + '/'
            #
            fileList = []
            for file in os.listdir(dir_name):
```

```
                # print(file)
                fileList.append(os.path.join(dir_name, file))
            fileList = sorted(fileList, key=lambda x:
int((x.split('/')[-1]).split('.')[0]))
            print("After sorted:",fileList[0])
            sentenceCnt = 0
            for file_name in fileList:
                f = open(file_name,'r',encoding='utf-8')
                sentenceCnt += 1
                if(sentenceCnt >= maxSentence):
                    break
                sample = json.load(f)
                answers = sample['answers']
                text = sample["article"]
                questions = sample['questions']
                options = sample['options']
                #rid = file_name[:-4]
                rid = sample['id']
                #print(file_name)
                for i in range(len(answers)):
                    samples.append(Race1(
                        race_id = rid+":"+str(i),
                        context_sentence = text,
                        start_ending = questions[i],
                        ending_0 = options[i][0],
                        ending_1 = options[i][1],
                        ending_2 = options[i][2],
                        ending_3 = options[i][3],
                        label = self.ansLabel_map[answers[i]]#ord(answers[i])-65
                        ))
        return samples

    def convert_examples_to_features1(self, examples, tokenizer, max_seq_length,
                                      is_training):
        """Loads a data file into a list of `InputBatch`s."""

        # RACE is a multiple choice task like Swag. To perform this task using
Bert,
        #
        # Each choice will correspond to a sample on which we run the
        # inference. For a given Race example, we will create the 4
        # following inputs:
        # - [CLS] context [SEP] choice_1 [SEP]
        # - [CLS] context [SEP] choice_2 [SEP]
        # - [CLS] context [SEP] choice_3 [SEP]
        # - [CLS] context [SEP] choice_4 [SEP]
        # The model will output a single value for each input. To get the
        # final decision of the model, we will run a softmax over these 4
        # outputs.
        features = []
```

```
        print("Length of Example: ", len(examples),examples[0])
        for example_index, example in enumerate(examples):
            context_tokens = tokenizer.tokenize(example.context_sentence) #
tokenize the sentance
            start_ending_tokens = tokenizer.tokenize(example.start_ending) #
question

            choices_features = []
            for ending_index, ending in enumerate(example.endings): #extract
options
                # We create a copy of the context tokens in order to be
                # able to shrink it according to ending_tokens
                context_tokens_choice = context_tokens[:]
                ending_tokens = start_ending_tokens + tokenizer.tokenize(ending) #
question + option convert to tokenize
                # Modifies `context_tokens_choice` and `ending_tokens` in
                # place so that the total length is less than the
                # specified length.  Account for [CLS], [SEP], [SEP] with
                # "- 3"
                self._truncate_seq_pair(context_tokens_choice, ending_tokens,
max_seq_length - 3)

                # generate full token with label ( sentence+ qustion+ option)
                tokens = ["[CLS]"] + context_tokens_choice + ["[SEP]"] +
ending_tokens + ["[SEP]"]
                #generate segment_id for represent sentence 0= context , 1 =
question
                #segment_ids = [0] * (len(context_tokens_choice) + 2) + [1] *
(len(start_ending_tokens)) +[2] * (len(tokenizer.tokenize(ending)) + 1)
                segment_ids = [0] * (len(context_tokens_choice) + 2) + \
                [1] * (len(ending_tokens) + 1) # article =0 ，question + option =1


                input_ids = tokenizer.convert_tokens_to_ids(tokens) #convert full
sentance + optiom  into BERT input ids , input token related id
                input_mask = [1] * len(input_ids) # mask 1 for input sentenace , 0
for padding

                # Zero-pad up to the sequence length.
                padding = [0] * (max_seq_length - len(input_ids))
                input_ids += padding    # padding 0 the full sentence
                input_mask += padding   # padding 0 the input mask
                segment_ids += padding  # padding 0 for segment ids

                assert len(input_ids) == max_seq_length
                assert len(input_mask) == max_seq_length
                assert len(segment_ids) == max_seq_length

                choices_features.append((tokens, input_ids, input_mask,
segment_ids))
```

```
            label = example.label
            if example_index == 0:
                print("*** Example ***")
                print("race_id: {}".format(example.race_id))
                for choice_idx, (tokens, input_ids, input_mask, segment_ids) in
enumerate(choices_features):
                    print("choice: {}".format(choice_idx))
                    print("tokens: {}".format(' '.join(tokens)))
                    print("input_ids: {}".format(' '.join(map(str, input_ids))))
                    print("input_mask: {}".format(' '.join(map(str, input_mask))))
                    print("segment_ids: {}".format(' '.join(map(str,
segment_ids))))
                if is_training:
                    print("label: {}".format(label))
            if (example_index%5000 ==0): print(example_index)
            features.append(
                InputFeatures1(
                    example_id = example.race_id,
                    choices_features = choices_features,
                    label = label
                    )
                )

        return features

    def _truncate_seq_pair(self, tokens_a, tokens_b, max_length):
        """Truncates a sequence pair in place to the maximum length."""

        # This is a simple heuristic which will always truncate the longer sequence
        # one token at a time. This makes more sense than truncating an equal
percent
        # of tokens from each, since if one sequence is very short then each token
        # that's truncated likely contains more information than a longer sequence.
        while True:
            total_length = len(tokens_a) + len(tokens_b)
            if total_length <= max_length:
                break
            if len(tokens_a) > len(tokens_b):
                tokens_a.pop()
            else:
                tokens_b.pop()

    def selectField(self, features, field):
        return [
                [
                    choice[field]
                    for choice in feature.choices_features
                ]
                for feature in features
```

```
            ]

    def accuracy(self, out, labels):
        outputs = np.argmax(out, axis=1)
        #print(outputs,outputs == labels)
        return np.sum(outputs == labels)

    def preprocessTrain(self, numEpoch):
        tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

        model = BertForMultipleChoice.from_pretrained("bert-base-uncased",

cache_dir=os.path.join(str(PYTORCH_PRETRAINED_BERT_CACHE)),

num_choices=4)
        #getTrain sample
        trainSamples =  self.read_raceModify(race_raw_train_path,
self.maxNumFileforTrainDirectory)
        num_train_optimization_steps  = int(len(trainSamples)
/self.train_batch_size / self.gradient_accumulation_steps) * numEpoch

        print("Optimzation Step: ", num_train_optimization_steps)



        print("Freeze network")
        for name, param in model.named_parameters():
            ln = 24
            if name.startswith('bert.encoder'):
                 l = name.split('.')
                 ln = int(l[3])

            if name.startswith('bert.embeddings') or ln < 6:
                print(name)
                param.requires_grad = False


        # Prepare optimizer
        param_optimizer = list(model.named_parameters())

        # hack to remove pooler, which is not used
        # thus it produce None grad that break apex
        param_optimizer = [n for n in param_optimizer if 'pooler' not in n[0]]

        no_decay = ['bias', 'LayerNorm.bias', 'LayerNorm.weight']
        optimizer_grouped_parameters = [
            {'params': [p for n, p in param_optimizer if not any(nd in n for nd in
no_decay)], 'weight_decay': 0.01},
            {'params': [p for n, p in param_optimizer if any(nd in n for nd in
no_decay)], 'weight_decay': 0.0}
```

```
            ]


        optimizer = BertAdam(optimizer_grouped_parameters,
                             lr=self.learning_rate,
                             warmup=self.warmup_proportion,
                             t_total=num_train_optimization_steps)


        trainFeatures = self.convert_examples_to_features1(trainSamples, tokenizer,
self.max_seq_length, True)
        trainLen = len(trainFeatures)
        print("\n***** Running training *****")
        print("  Num examples = {}".format(len(trainSamples)))
        print("  Batch size = {}".format(self.train_batch_size))
        print("  Num steps = {}".format(num_train_optimization_steps))

        #convert into tensor
        all_input_ids =  torch.tensor(self.selectField(trainFeatures, 'input_ids')
,dtype=torch.long)
        all_input_mask = torch.tensor(self.selectField(trainFeatures, 'input_mask')
,dtype=torch.long)
        all_segment_ids =  torch.tensor(self.selectField(trainFeatures,
'segment_ids') ,dtype=torch.long)
        all_label= torch.tensor([f.label for f in trainFeatures], dtype=torch.long)

        trainData = TensorDataset(all_input_ids, all_input_mask, all_segment_ids,
all_label)
        #use RandomSampler
        train_sampler = RandomSampler(trainData)
        #user DistributedSamper
        #train_sampler = DistributedSampler(trainData)
        trainDataLoader = DataLoader(trainData, sampler=train_sampler, batch_size=
self.train_batch_size)


        # #device = 'cpu'
        # # move model over to detected device
        self.train(numEpoch, model, optimizer, trainDataLoader)


    def train(self, numEpoch, model, optimizer, trainDataLoader):
        device = torch.device('cuda') if torch.cuda.is_available() else
torch.device('cpu')
        #device = 'cpu'
        # move model over to detected device
        model.to(device)

        model.train()
```

```
        global_step = 0
        for epoch in range(numEpoch):
            tr_loss = 0
            last_tr_loss = 0
            nb_tr_examples, nb_tr_steps = 0, 0
            for step, batch in enumerate(tqdm(trainDataLoader, desc="Iteration")):
                batch = tuple(t.to(device) for t in batch)
                input_ids, input_mask, segment_ids, label_ids = batch

                loss = model(input_ids, segment_ids, input_mask, label_ids) # for
pytorch_pretrained_bert only
                # loss = outputs[0]
                loss = loss / self.gradient_accumulation_steps
                tr_loss += loss.item()
                nb_tr_examples += input_ids.size(0)
                nb_tr_steps +=1
                loss.backward()
                # optim.step()
                if (step + 1) % self.gradient_accumulation_steps == 0:
                    optimizer.step()
                    optimizer.zero_grad()
                    global_step +=1

                if nb_tr_examples % 512 ==0:
                    loss_log = (tr_loss - last_tr_loss)* 1.0/512
                    print("\nNum Of Epoch: {} , Num Of Step: {} , Loss:
{:.3f}".format(epoch, nb_tr_examples, loss_log))
                    last_tr_loss = tr_loss

        #save trained model
        torch.save(model, self.trainedRaceModelFile) # save entire model

    def preprocssEval(self):
        model = torch.load(self.trainedRaceModelFile)
        evalSamples =  self.read_raceModify(race_raw_dev_path,
self.maxNumFileForEvalDirectory)
        evalFeatures = self.convert_examples_to_features1(evalSamples,
self.tokenizer, self.max_seq_length, True)
        print("\n***** Running evaluation *****")
        print("  Num examples = {}".format(len(evalSamples)))
        print("  Batch size = {}".format(self.eval_batch_size))

        #convert into tensor
        all_input_ids =  torch.tensor(self.selectField(evalFeatures, 'input_ids')
,dtype=torch.long)
        all_input_mask = torch.tensor(self.selectField(evalFeatures, 'input_mask')
,dtype=torch.long)
        all_segment_ids =  torch.tensor(self.selectField(evalFeatures,
'segment_ids') ,dtype=torch.long)
        all_label= torch.tensor([f.label for f in evalFeatures], dtype=torch.long)
```

```
        evalData = TensorDataset(all_input_ids, all_input_mask, all_segment_ids,
all_label)

        # Run prediction for full data
        eval_sampler = SequentialSampler(evalData)
        evalDataLoader = DataLoader(evalData, sampler=eval_sampler, batch_size=
self.eval_batch_size)


        _, _,  eval_loss, eval_accuracy = self.evalutionORtest(model,
evalDataLoader, "Evaluating")

        print("\n****Eval Reult****")
        print("Evalate loss {:.3f}".format(eval_loss))
        print("Accuaracy    {:.3f}%".format(eval_accuracy* 100))

    def preprocessTest(self):
        model = torch.load(self.trainedRaceModelFile)
        testSamples =  self.read_raceModify(race_raw_test_path,
self.maxNumFileForEvalDirectory)
        testFeatures = self.convert_examples_to_features1(testSamples,
self.tokenizer, self.max_seq_length, True)


        print("\n***** Running test *****")
        print("  Num examples = {}".format(len(testSamples)))
        print("  Batch size = {}".format(self.test_batch_size))

        #convert into tensor
        all_input_ids =  torch.tensor(self.selectField(testFeatures, 'input_ids')
,dtype=torch.long)
        all_input_mask = torch.tensor(self.selectField(testFeatures, 'input_mask')
,dtype=torch.long)
        all_segment_ids =  torch.tensor(self.selectField(testFeatures,
'segment_ids') ,dtype=torch.long)
        all_label= torch.tensor([f.label for f in testFeatures], dtype=torch.long)

        testData = TensorDataset(all_input_ids, all_input_mask, all_segment_ids,
all_label)


        test_sampler = SequentialSampler(testData)
        testDataLoader = DataLoader(testData, sampler=test_sampler, batch_size=
self.test_batch_size)


        _, _,  eval_loss, eval_accuracy = self.evalutionORtest(model,
testDataLoader, "Testing")
```

```
        print("\n****TEST Reult****")
        print("Test loss {:.3f}".format(eval_loss))
        print("Test Accuaracy    {:.3f}%".format(eval_accuracy* 100))

    def evalutionORtest(self, model, dataLoader, des):
        model.eval()
        tr_loss = 0
        eval_loss, eval_accuracy = 0, 0
        nb_eval_steps, nb_eval_examples = 0, 0
        total_logits = []
        total_labels = []
        for input_ids, input_mask, segment_ids, label_ids in tqdm(dataLoader,
desc=des):
            input_ids = input_ids.to(self.device)
            input_mask = input_mask.to(self.device)
            segment_ids = segment_ids.to(self.device)
            label_ids = label_ids.to(self.device)

            with torch.no_grad():
                tmp_eval_loss = model(input_ids, segment_ids, input_mask,
label_ids)
                logits = model(input_ids, segment_ids, input_mask)

            logits = logits.detach().cpu().numpy()
            label_ids = label_ids.to('cpu').numpy()
            tmp_eval_accuracy = self.accuracy(logits, label_ids)

            eval_loss += tmp_eval_loss.mean().item()
            eval_accuracy += tmp_eval_accuracy

            nb_eval_examples += input_ids.size(0)
            nb_eval_steps += 1

            total_logits.append(logits)
            total_labels.append(label_ids)

        total_logits = np.concatenate(total_logits)
        total_labels = np.concatenate(total_labels)

        # np.save(args.output_dir+"/logits.npy",total_logits)
        # np.save(args.output_dir+"/labels.npy",total_labels)

        eval_loss = eval_loss / nb_eval_steps
        eval_accuracy = eval_accuracy / nb_eval_examples
        return total_logits, total_labels , eval_loss, eval_accuracy

    def main(self):
        random.seed(self.seed)
        np.random.seed(self.seed)
        torch.manual_seed(self.seed)
```

```
        #Model inital
        #mcaqTrainModel = MCQATrainModel() # inital

        device = torch.device('cuda') if torch.cuda.is_available() else
torch.device('cpu')

        self.preprocessTrain(self.numEpoch)

        # #load pytorch model
        # model = torch.load(self.trainedRaceModelFile)

        # #prepare eval sample
        #mcaqTrainModel.preprocssEval()

        # #for test
        #mcaqTrainModel.preprocessTest()



if __name__ == "__main__":
    #Model inital
        mcaqTrainModel = MCQATrainModel() # inital
        mcaqTrainModel.main()



#use for external call function
def raceTest():
    mcaqTrainModel = MCQATrainModel()
    mcaqTrainModel.preprocessTest()

def raceEval():
    mcaqTrainModel = MCQATrainModel()
    mcaqTrainModel.preprocssEval()

def raceTrain():
    mcaqTrainModel = MCQATrainModel()
    mcaqTrainModel.preprocessTrain(2)
```