CODE. <https://www.kaggle.com/code/wht1996/feedback-nn-train>

import copy

import numpy as np

import pandas as pd

def calc\_overlap(row):

set\_pred = set(row.predictionstring\_pred.split(' '))

set\_gt = set(row.predictionstring\_gt.split(' '))

len\_gt = len(set\_gt)

len\_pred = len(set\_pred)

inter = len(set\_gt.intersection(set\_pred))

return inter/max(len\_gt,len\_pred)

//The calc\_overlap(row) function calculates the overlap between the predicted entities and the ground truth entities for a given row of data (presumably representing a single text example). This overlap calculation helps in determining whether the predicted entities match closely with the ground truth entities.

def get\_f1\_score(test\_pred,test\_df,log=print,slient=True):

if **not** slient:

log('test\_pred.shape:',test\_pred.shape,'**\t**test\_df.shape:',test\_df.shape,)

log('pred class:**\n**',test\_pred['class'].value\_counts())

log('true class:**\n**',test\_df['discourse\_type'].value\_counts())

f1s = []

for c **in** sorted(test\_pred['class'].unique()):

pred\_df = test\_pred.loc[test\_pred['class']==c].copy()

gt\_df = test\_df.loc[test\_df['discourse\_type']==c].copy()

gt\_df = gt\_df[['id','discourse\_type','predictionstring']].reset\_index(drop=True).copy()

pred\_df = pred\_df[['id','class','predictionstring']].reset\_index(drop=True).copy()

pred\_df['pred\_id'] = pred\_df.index

gt\_df['gt\_id'] = gt\_df.index

*# Step 1. all ground truths and predictions for a given class are compared.*

joined = pred\_df.merge(gt\_df,

left\_on=['id','class'],

right\_on=['id','discourse\_type'],

how='outer',

suffixes=('\_pred','\_gt')

)

joined['predictionstring\_gt'] = joined['predictionstring\_gt'].fillna(' ')

joined['predictionstring\_pred'] = joined['predictionstring\_pred'].fillna(' ')

*# print(joined.head())*

joined['min\_overlaps'] = joined.apply(calc\_overlap, axis=1)

joined['potential\_TP'] = (joined['min\_overlaps'] >= 0.5)

matched\_pred\_ids = joined.query('potential\_TP')['pred\_id'].unique()

fp\_pred\_ids = [p for p **in** joined['pred\_id'].unique() if p **not** **in** matched\_pred\_ids]

matched\_gt\_ids = joined.query('potential\_TP')['gt\_id'].unique()

fn\_gt\_ids = [c for c **in** joined['gt\_id'].unique() if c **not** **in** matched\_gt\_ids]

*# Get numbers of each type*

TP = len(matched\_gt\_ids)

FP = len(fp\_pred\_ids)

FN = len(fn\_gt\_ids)

*#calc microf1*

f1\_score = TP / (TP + 0.5\*(FP+FN))

if **not** slient:

log(f'**{**c**:**<20**}** f1 score:**\t{**f1\_score**}**')

f1s.append(f1\_score)

log('**\n**Overall f1 score **\t**',np.mean(f1s))

return np.mean(f1s)

//calculating the f1 score instead of using scikit-learn gives more filexibility and better understanding of the calculation.

import time

class Log:

def \_\_init\_\_(self,log\_path,time\_key=True):

self.path = log\_path

if time\_key:

self.path = self.path.replace('.','**{}**.'.format(time.strftime('\_%Y%m**%d**%H%M%S',time.localtime(time.time()))))

print(time.strftime('%Y-%m-**%d** %H:%M:%S',time.localtime(time.time())),file=open(self.path,'a+'))

print('log path:', self.path)

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*begin\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*',file=open(self.path,'a+'))

def \_\_call\_\_(self,\*content):

t1 = time.strftime('%H:%M:%S',time.localtime(time.time()))

print(\*content)

print(t1,content,file=open(self.path,'a+'))

def clean(self):

print(time.strftime('%Y-%m-**%d** %H:%M:%S',time.localtime(time.time())),file=open(self.path,'w'))

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*begin\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*',file=open(self.path,'a+'))

import math

import torch

from torch.optim.lr\_scheduler import ReduceLROnPlateau,\_LRScheduler

class CosineAnnealingWarmupRestarts(\_LRScheduler):

*"""*

*optimizer (Optimizer): Wrapped optimizer.*

*first\_cycle\_steps (int): First cycle step size.*

*cycle\_mult(float): Cycle steps magnification. Default: -1.*

*max\_lr(float): First cycle's max learning rate. Default: 0.1.*

*min\_lr(float): Min learning rate. Default: 0.001.*

*warmup\_steps(int): Linear warmup step size. Default: 0.*

*gamma(float): Decrease rate of max learning rate by cycle. Default: 1.*

*last\_epoch (int): The index of last epoch. Default: -1.*

*"""*

def \_\_init\_\_(self,

optimizer: torch.optim.Optimizer,

first\_cycle\_steps: int,

cycle\_mult: float = 1.,

max\_lr: float = 0.1,

min\_lr: float = 0.001,

warmup\_steps: int = 0,

gamma: float = 1.,

last\_epoch: int = -1

):

assert warmup\_steps < first\_cycle\_steps

self.first\_cycle\_steps = first\_cycle\_steps *# first cycle step size*

self.cycle\_mult = cycle\_mult *# cycle steps magnification*

self.base\_max\_lr = max\_lr *# first max learning rate*

self.max\_lr = max\_lr *# max learning rate in the current cycle*

self.min\_lr = min\_lr *# min learning rate*

self.warmup\_steps = warmup\_steps *# warmup step size*

self.gamma = gamma *# decrease rate of max learning rate by cycle*

self.cur\_cycle\_steps = first\_cycle\_steps *# first cycle step size*

self.cycle = 0 *# cycle count*

self.step\_in\_cycle = last\_epoch *# step size of the current cycle*

super(CosineAnnealingWarmupRestarts, self).\_\_init\_\_(optimizer, last\_epoch)

*# set learning rate min\_lr*

self.init\_lr()

def init\_lr(self):

self.base\_lrs = []

for param\_group **in** self.optimizer.param\_groups:

param\_group['lr'] = self.min\_lr

self.base\_lrs.append(self.min\_lr)

def get\_lr(self):

if self.step\_in\_cycle == -1:

return self.base\_lrs

elif self.step\_in\_cycle < self.warmup\_steps:

return [(self.max\_lr - base\_lr) \* self.step\_in\_cycle / self.warmup\_steps + base\_lr for base\_lr **in** self.base\_lrs]

else:

return [base\_lr + (self.max\_lr - base\_lr)

\* (1 + math.cos(math.pi \* (self.step\_in\_cycle - self.warmup\_steps)

/ (self.cur\_cycle\_steps - self.warmup\_steps))) / 2

for base\_lr **in** self.base\_lrs]

def step(self, epoch=None):

if epoch **is** None:

epoch = self.last\_epoch + 1

self.step\_in\_cycle = self.step\_in\_cycle + 1

if self.step\_in\_cycle >= self.cur\_cycle\_steps:

self.cycle += 1

self.step\_in\_cycle = self.step\_in\_cycle - self.cur\_cycle\_steps

self.cur\_cycle\_steps = int((self.cur\_cycle\_steps - self.warmup\_steps) \* self.cycle\_mult) + self.warmup\_steps

else:

if epoch >= self.first\_cycle\_steps:

if self.cycle\_mult == 1.:

self.step\_in\_cycle = epoch % self.first\_cycle\_steps

self.cycle = epoch // self.first\_cycle\_steps

else:

n = int(math.log((epoch / self.first\_cycle\_steps \* (self.cycle\_mult - 1) + 1), self.cycle\_mult))

self.cycle = n

self.step\_in\_cycle = epoch - int(self.first\_cycle\_steps \* (self.cycle\_mult \*\* n - 1) / (self.cycle\_mult - 1))

self.cur\_cycle\_steps = self.first\_cycle\_steps \* self.cycle\_mult \*\* (n)

else:

self.cur\_cycle\_steps = self.first\_cycle\_steps

self.step\_in\_cycle = epoch

self.max\_lr = self.base\_max\_lr \* (self.gamma\*\*self.cycle)

self.last\_epoch = math.floor(epoch)

for param\_group, lr **in** zip(self.optimizer.param\_groups, self.get\_lr()):

param\_group['lr'] = lr

class EarlyStopping:

def \_\_init\_\_(self, patience=6, mode="max", max\_epoch=1e6, min\_epoch=0, at\_last\_score=None):

self.patience = patience

self.mode = mode

self.max\_epoch = max\_epoch

self.min\_epoch = min\_epoch

self.at\_last\_score = at\_last\_score if at\_last\_score **is** **not** None else -np.Inf

self.epoch = 0

self.early\_stop = False

self.best\_model = None

self.best\_epoch = 0

self.model\_path = None

self.best\_score = -np.Inf if self.mode == "max" else np.Inf

def \_\_call\_\_(self, epoch\_score, model=None, model\_path=None):

self.model\_path = model\_path

self.epoch += 1

score = -epoch\_score if self.mode == "min" else epoch\_score

if score <= self.best\_score:

counter = self.epoch - self.best\_epoch

print('EarlyStopping counter: **{}** out of **{}**'.format(counter, self.patience))

if (counter >= self.patience) **and** (self.best\_score > self.at\_last\_score) **and** (self.epoch >= self.min\_epoch):

self.early\_stop = True

self.\_save\_checkpoint()

else:

self.best\_score = score

self.best\_epoch = self.epoch

self.best\_model = copy.deepcopy(model).cpu()

if self.max\_epoch <= self.epoch:

self.early\_stop = True

self.\_save\_checkpoint()

def \_save\_checkpoint(self):

if self.model\_path **is** **not** None **and** self.best\_model **is** **not** None:

torch.save(self.best\_model.state\_dict(), self.model\_path.replace('\_score','\_'+str(self.best\_score)))

print('model saved at: ',self.model\_path.replace('\_score','\_'+str(self.best\_score)))

import os

import time

import torch

import pickle

import numpy as np

import pandas as pd

from tqdm import tqdm

from torch.utils.data import Dataset

def mapping\_to\_ids(mapping,text):

mapping[0] = 0 if mapping[0]==1 else mapping[0]

start = len(text[:mapping[0]].split())

end = len(text[:mapping[1]].split())

return [str(i) for i **in** range(start,end)]

def is\_head(token):

return len(token.split())>0 **and** token **not** **in** '.,;?!'

def get\_offset\_mapping(text,tokens):

offset\_mapping = []

start\_index = 0

for t **in** tokens:

while start\_index<len(text) **and** text[start\_index] **in** ['**\xa0**', '**\n**', ' ']:

start\_index += 1

if t **in** ('[CLS]', '[SEP]'):

mapping = (0,0)

elif t == '[UNK]':

mapping = (start\_index,start\_index+1)

else:

if t[0]=='▁':

t = t[1:]

t\_len = len(t)

if t\_len == 0:

mapping = (start\_index,start\_index+1)

else:

sample = False

for i **in** [0, 1, 2, 3, 4, 5, -1,0]:

if t.lower() == text[start\_index+i:start\_index+t\_len+i].lower():

sample = True

break

if sample:

mapping = (start\_index+i,start\_index+t\_len+i)

elif t[0]=='<' **and** t[-1]=='>':

mapping = (start\_index,start\_index+1)

else:

mapping = (start\_index+i,start\_index+t\_len+i)

start\_index = mapping[1]

offset\_mapping.append(mapping)

if len(offset\_mapping) != len(tokens):

raise **ValueError**('offset\_mapping error！')

if abs(offset\_mapping[-2][1] - len(text))>2:

raise **ValueError**('offset\_mapping erroe！')

return offset\_mapping

def encode(text,tokenizer,data,labels\_to\_ids):

if 'deberta-v' **in** tokenizer.name\_or\_path:

encoding = tokenizer.encode\_plus(text)

tokens = tokenizer.convert\_ids\_to\_tokens(encoding['input\_ids'])

encoding['offset\_mapping'] = get\_offset\_mapping(text,tokens)

else:

encoding = tokenizer.encode\_plus(text,

return\_offsets\_mapping=True,

)

input\_ids = encoding['input\_ids']

attention\_mask = encoding['attention\_mask']

offset\_mapping = encoding['offset\_mapping']

unmask\_length = sum(attention\_mask)

token\_label = [0] \* unmask\_length

if 'discourse\_start' **in** data.columns:

data = data.sort\_values('discourse\_start').reset\_index(drop=True)

char\_label = list(zip(data.discourse\_start.astype(int),data.discourse\_end.astype(int),data.discourse\_type))

is\_first = True

for i,mapping **in** enumerate(offset\_mapping):

while len(char\_label) > 0 **and** mapping[1] > char\_label[0][1]:

char\_label.pop(0)

is\_first = True

if len(char\_label) == 0:

break

if mapping[1] >= char\_label[0][0] **and** mapping != (0,0):

if is\_first:

if is\_head(text[mapping[0]:mapping[1]]):

token\_label[i] = labels\_to\_ids['B-'+char\_label[0][2]]

is\_first = False

else:

token\_label[i] = labels\_to\_ids['I-'+char\_label[0][2]]

return input\_ids,attention\_mask,token\_label,offset\_mapping

def get\_feat\_helper(args, tokenizer, df, train\_ids):

training\_samples = []

for idx **in** train\_ids:

filename = args.text\_path + idx + ".txt"

with open(filename, "r") as f:

text = f.read().rstrip()

input\_ids, attention\_mask, token\_label, offset\_mapping = \

encode(text,tokenizer,df[df['id']==idx],args.labels\_to\_ids)

training\_samples.append({'id':idx,'text':text,'input\_ids':input\_ids,

'attention\_mask':attention\_mask,'token\_label':token\_label,

'offset\_mapping':offset\_mapping})

return training\_samples

//tokenizing part above

from joblib import Parallel, delayed

def get\_feat(df, tokenizer, args, data\_key):

data\_path = args.cache\_path + 'feat\_**{}**.pkl'.format(data\_key)

if os.path.exists(data\_path) & (args.load\_feat):

data = pickle.load(open(data\_path,'+rb'))

else:

num\_jobs = 8

data = []

train\_ids = df["id"].unique()

train\_ids\_splits = np.array\_split(train\_ids, num\_jobs)

results = Parallel(n\_jobs=num\_jobs, backend="multiprocessing")(

delayed(get\_feat\_helper)(args, tokenizer, df, idx) for idx **in** train\_ids\_splits

)

for result **in** results:

data.extend(result)

data = pd.DataFrame(sorted(data,key=lambda x:len(x['input\_ids'])))

pickle.dump(data,open(data\_path,'+wb'))

return data

class dataset:

def \_\_init\_\_(self, data):

self.data = data

def \_\_len\_\_(self):

return len(self.data)

def \_\_getitem\_\_(self, index):

id\_ = self.data.id[index]

input\_ids = self.data.input\_ids[index]

attention\_mask = self.data.attention\_mask[index]

token\_label = self.data.token\_label[index]

item = {'id':id\_,

'input\_ids': torch.tensor(input\_ids, dtype=torch.long),

'attention\_mask': torch.tensor(attention\_mask, dtype=torch.long),

'labels': torch.tensor(token\_label, dtype=torch.long),

}

return item

// It splits the training data into multiple chunks and processes each chunk in parallel.

The dataset class integrates with PyTorch and is designed to work seamlessly with the rest of the training pipeline.

The get\_feat function handles feature generation efficiently by parallelizing the process, making use of joblib's Parallel function.

import gc

import time

import torch

import numpy as np

import pandas as pd

from tqdm import tqdm

import torch.nn.functional as F

from sklearn.metrics import accuracy\_score

def feat\_padding(input\_ids,attention\_mask,token\_label,batch\_length,padding\_dict,padding\_side):

random\_seed = None

if padding\_side == 'right':

random\_seed = 0

elif padding\_side == 'left':

random\_seed = 1

else:

random\_seed = np.random.rand()

mask\_index = attention\_mask.nonzero().reshape(-1)

input\_ids = input\_ids.index\_select(0,mask\_index)

token\_label = token\_label.index\_select(0,mask\_index)

attention\_mask = attention\_mask.index\_select(0,mask\_index)

ids\_length = len(input\_ids)

if ids\_length>batch\_length:

if random\_seed<=0.33:

input\_ids = input\_ids[:batch\_length]

attention\_mask = attention\_mask[:batch\_length]

token\_label = token\_label[:batch\_length]

elif random\_seed >= 0.66:

input\_ids = input\_ids[-batch\_length:]

attention\_mask = attention\_mask[-batch\_length:]

token\_label = token\_label[-batch\_length:]

else:

sub\_length = ids\_length - batch\_length

strat\_idx = np.random.randint(sub\_length+1)

input\_ids = input\_ids[strat\_idx:strat\_idx+batch\_length]

attention\_mask = attention\_mask[strat\_idx:strat\_idx+batch\_length]

token\_label = token\_label[strat\_idx:strat\_idx+batch\_length]

if ids\_length<batch\_length:

add\_length = batch\_length-ids\_length

if random\_seed<=0.33:

input\_ids = F.pad(input\_ids, (0,add\_length), "constant", padding\_dict['input\_ids'])

attention\_mask = F.pad(attention\_mask, (0,add\_length), "constant", padding\_dict['attention\_mask'])

token\_label = F.pad(token\_label, (0,add\_length), "constant", padding\_dict['input\_ids'])

elif random\_seed >= 0.66:

input\_ids = F.pad(input\_ids, (add\_length,0), "constant", padding\_dict['input\_ids'])

attention\_mask = F.pad(attention\_mask, (add\_length,0), "constant", padding\_dict['attention\_mask'])

token\_label = F.pad(token\_label, (add\_length,0), "constant", padding\_dict['input\_ids'])

else:

add\_length1 = np.random.randint(add\_length+1)

add\_length2 = add\_length - add\_length1

input\_ids = F.pad(input\_ids, (add\_length1,add\_length2), "constant", padding\_dict['input\_ids'])

attention\_mask = F.pad(attention\_mask, (add\_length1,add\_length2), "constant", padding\_dict['attention\_mask'])

token\_label = F.pad(token\_label, (add\_length1,add\_length2), "constant", padding\_dict['input\_ids'])

return input\_ids,attention\_mask,token\_label

// Overall, this function efficiently handles padding of input features to match the desired batch length, ensuring compatibility with batch processing in neural network training. It also provides flexibility by allowing both deterministic and random padding strategies.

class Collate:

def \_\_init\_\_(self, model\_length=None,max\_length=None,padding\_side='right',padding\_dict={}):

self.model\_length = model\_length

self.max\_length = max\_length

self.padding\_side = padding\_side

self.padding\_dict = padding\_dict

def \_\_call\_\_(self, batch):

output = dict()

output["input\_ids"] = [sample["input\_ids"] for sample **in** batch]

output["attention\_mask"] = [sample["attention\_mask"] for sample **in** batch]

output["labels"] = [sample["labels"] for sample **in** batch]

*# calculate max token length of this batch*

batch\_length = None

if self.model\_length **is** **not** None:

batch\_length = self.model\_length

else:

batch\_length = max([len(ids) for ids **in** output["input\_ids"]])

if self.max\_length **is** **not** None:

batch\_length = min(batch\_length,self.max\_length)

for i **in** range(len(output["input\_ids"])):

output\_fill = feat\_padding(output["input\_ids"][i], output["attention\_mask"][i],output["labels"][i],

batch\_length,self.padding\_dict,padding\_side=self.padding\_side)

output["input\_ids"][i],output["attention\_mask"][i], output["labels"][i] = output\_fill

*# convert to tensors*

output["input\_ids"] = torch.stack(output["input\_ids"])

output["attention\_mask"] = torch.stack(output["attention\_mask"])

output["labels"] = torch.stack(output["labels"])

return output

//Collate Class:

This class defines a collate function to process batches of data before feeding them into the model.

It pads input features within each batch to ensure uniform batch sizes.

The padding side and maximum batch length can be specified during initialization.

def test(model,test\_loader,test\_feat,test\_df,args,log):

model.eval()

te\_loss,te\_accuracy = [],[]

test\_pred = []

scaler = torch.cuda.amp.GradScaler()

with torch.no\_grad():

for batch **in** test\_loader:

*# with torch.cuda.amp.autocast():*

*# loss, te\_logits = model(input\_ids=input\_ids, attention\_mask=attention\_mask, labels=labels)*

*# loss = loss.mean()*

if args.test\_padding\_side **in** ['right','left']:

loss, te\_logits = model\_predict(model, batch, model\_length=args.model\_length,max\_length=args.max\_length,

padding\_dict=args.padding\_dict,padding\_side=args.test\_padding\_side)

elif args.test\_padding\_side == 'double':

loss1, te\_logits1 = model\_predict(model, batch, model\_length=args.model\_length,max\_length=args.max\_length,

padding\_dict=args.padding\_dict,padding\_side='right')

loss2, te\_logits2 = model\_predict(model, batch, model\_length=args.model\_length,max\_length=args.max\_length,

padding\_dict=args.padding\_dict,padding\_side='left')

loss = (loss1+loss2)/2

te\_logits = [(l1+l2)/2 for l1,l2 **in** zip(te\_logits1,te\_logits2)]

*# compute training accuracy*

for i **in** range(len(te\_logits)):

pred = te\_logits[i].argmax(axis=-1)

token\_label = batch['labels'][i][batch['attention\_mask'][i]>0].cpu().numpy().reshape(-1)

te\_accuracy.append(accuracy\_score(pred, token\_label))

te\_loss.append(loss.item())

test\_pred.extend(te\_logits)

te\_accuracy = np.mean(te\_accuracy)

te\_loss = np.mean(te\_loss)

gc.collect()

torch.cuda.empty\_cache()

model.train()

test\_feat['pred'] = test\_pred

segment\_param = {

"Lead": {'min\_proba':[0.47,0.41],'begin\_proba':1.00,'min\_sep':40,'min\_length': 5},

"Position": {'min\_proba':[0.45,0.40],'begin\_proba':0.90,'min\_sep':21,'min\_length': 3},

"Evidence": {'min\_proba':[0.50,0.40],'begin\_proba':0.56,'min\_sep': 2,'min\_length':21},

"Claim": {'min\_proba':[0.40,0.30],'begin\_proba':0.30,'min\_sep':10,'min\_length': 1},

"Concluding Statement": {'min\_proba':[0.58,0.25],'begin\_proba':0.93,'min\_sep':50,'min\_length': 5},

"Counterclaim": {'min\_proba':[0.45,0.25],'begin\_proba':0.70,'min\_sep':35,'min\_length': 6},

"Rebuttal": {'min\_proba':[0.37,0.34],'begin\_proba':0.70,'min\_sep':45,'min\_length': 5},

}

test\_predictionstring = after\_deal(test\_feat, args.labels\_to\_ids, segment\_param,log)

f1\_score = get\_f1\_score(test\_predictionstring,test\_df,log)

return te\_loss,te\_accuracy,f1\_score,test\_pred

def train(model,train\_loader,test\_loader,test\_feat,test\_df,args,model\_path,log):

optimizer = torch.optim.Adam(params=model.parameters(), lr=args.lr)

scheduler = CosineAnnealingWarmupRestarts(optimizer = optimizer,

first\_cycle\_steps = args.num\_train\_steps, cycle\_mult = 1,

max\_lr = args.max\_lr, min\_lr = args.min\_lr, warmup\_steps = args.num\_train\_steps \* 0.2,

gamma = 1.,last\_epoch = -1

)

es = EarlyStopping(patience=4,max\_epoch=args.epochs)

t0 = time.time()

scaler = torch.cuda.amp.GradScaler()

awp = AWP(model,

optimizer,

adv\_lr=args.adv\_lr,

adv\_eps=args.adv\_eps,

start\_epoch=args.num\_train\_steps/args.epochs,

scaler=scaler

)

step = 0

f1\_score = 0

for epoch **in** range(100):

tr\_loss, tr\_accuracy = [], []

nb\_tr\_steps = 0

model.train()

for idx, batch **in** enumerate(train\_loader):

step += 1

input\_ids = batch['input\_ids'].to(args.device, dtype = torch.long)

attention\_mask = batch['attention\_mask'].to(args.device, dtype = torch.long)

labels = batch['labels'].to(args.device, dtype = torch.long)

with torch.cuda.amp.autocast():

loss, tr\_logits = model(input\_ids=input\_ids, attention\_mask=attention\_mask, labels=labels)

loss = loss.mean()

optimizer.zero\_grad()

*# loss.backward()*

*# optimizer.step()*

scaler.scale(loss).backward()

if f1\_score > 0.64:

awp.attack\_backward(input\_ids,labels,attention\_mask,step)

*# gradient clipping*

torch.nn.utils.clip\_grad\_norm\_(

parameters=model.parameters(), max\_norm=10

)

scaler.step(optimizer)

scaler.update()

scheduler.step()

tr\_loss.append(loss.item())

nb\_tr\_steps += 1

*# compute training accuracy*

for i **in** range(len(tr\_logits)):

pred = tr\_logits[i][attention\_mask[i]>0].detach().cpu().numpy().argmax(axis=-1)

token\_label = labels[i][attention\_mask[i]>0].cpu().numpy().reshape(-1)

tr\_accuracy.append(accuracy\_score(pred, token\_label))

if idx % 200==0:

del input\_ids,attention\_mask,labels,loss, tr\_logits

gc.collect()

torch.cuda.empty\_cache()

tr\_loss\_ = np.mean(tr\_accuracy)

tr\_accuracy\_ = np.mean(tr\_accuracy)

log(f"step: **\t{**idx**:**04d**}**, train loss: **\t{**tr\_loss\_**:**.4f**}**, train acc: **\t{**tr\_accuracy\_**:**.4f**}**, time: **\t{**int(time.time()-t0)**}**s")

gc.collect()

torch.cuda.empty\_cache()

tr\_loss\_ = np.mean(tr\_loss)

tr\_accuracy\_ = np.mean(tr\_accuracy)

te\_loss, te\_accuracy,f1\_score, test\_pred = test(model,test\_loader,test\_feat,test\_df,args,log)

log(f"epoch: **\t{**epoch**:**04d**}**, train loss: **\t{**tr\_loss\_**:**.4f**}**, train acc: **\t{**tr\_accuracy\_**:**.4f**}**, test loss: **\t{**te\_loss**:**.4f**}**, test acc: **\t{**te\_accuracy**:**.4f**}**, test f1: **\t{**f1\_score**:**.4f**}**, time: **\t{**int(time.time()-t0)**}**s")

es(f1\_score,model,model\_path=model\_path)

if es.early\_stop:

break

return es.best\_model.to(next(model.parameters()).device),test\_pred

class AWP:

def \_\_init\_\_(

self,

model,

optimizer,

adv\_param="weight",

adv\_lr=1,

adv\_eps=0.2,

start\_epoch=0,

adv\_step=1,

scaler=None

):

self.model = model

self.optimizer = optimizer

self.adv\_param = adv\_param

self.adv\_lr = adv\_lr

self.adv\_eps = adv\_eps

self.start\_epoch = start\_epoch

self.adv\_step = adv\_step

self.backup = {}

self.backup\_eps = {}

self.scaler = scaler

def attack\_backward(self, x, y, attention\_mask,epoch):

if (self.adv\_lr == 0) **or** (epoch < self.start\_epoch):

return None

self.\_save()

for i **in** range(self.adv\_step):

self.\_attack\_step()

with torch.cuda.amp.autocast():

adv\_loss, tr\_logits = self.model(input\_ids=x, attention\_mask=attention\_mask, labels=y)

adv\_loss = adv\_loss.mean()

self.optimizer.zero\_grad()

self.scaler.scale(adv\_loss).backward()

self.\_restore()

def \_attack\_step(self):

e = 1e-6

for name, param **in** self.model.named\_parameters():

if param.requires\_grad **and** param.grad **is** **not** None **and** self.adv\_param **in** name:

norm1 = torch.norm(param.grad)

norm2 = torch.norm(param.data.detach())

if norm1 != 0 **and** **not** torch.isnan(norm1):

r\_at = self.adv\_lr \* param.grad / (norm1 + e) \* (norm2 + e)

param.data.add\_(r\_at)

param.data = torch.min(

torch.max(param.data, self.backup\_eps[name][0]), self.backup\_eps[name][1]

)

*# param.data.clamp\_(\*self.backup\_eps[name])*

def \_save(self):

for name, param **in** self.model.named\_parameters():

if param.requires\_grad **and** param.grad **is** **not** None **and** self.adv\_param **in** name:

if name **not** **in** self.backup:

self.backup[name] = param.data.clone()

grad\_eps = self.adv\_eps \* param.abs().detach()

self.backup\_eps[name] = (

self.backup[name] - grad\_eps,

self.backup[name] + grad\_eps,

)

def \_restore(self,):

for name, param **in** self.model.named\_parameters():

if name **in** self.backup:

param.data = self.backup[name]

self.backup = {}

self.backup\_eps = {}

def mapping\_to\_ids(mapping, text):

word\_start = len(text[:mapping[0]].split())

word\_end = word\_start + len(text[mapping[0]:mapping[1]].split())

word\_end = min(word\_end, len(text.split()))

output = " ".join([str(x) for x **in** range(word\_start, word\_end)])

return output

sentence\_split = [';', ',', '.', '?', '!', '"']

null\_symbol = ['**\xa0**', '**\n**', '**\x93**', ' ']

def get\_sentence\_split(text,pred\_head=[]):

start = []

end = True

for i,t **in** enumerate(text):

if t == ' ':

continue

if i **in** pred\_head:

start.append(i)

end = False

if (t **in** sentence\_split **and** text[i-1] **not** **in** null\_symbol) **or** \

(t **in** null\_symbol):

end = True

if (t **in** null\_symbol) **and** (text[i-1] **not** **in** null\_symbol):

start.append(i)

elif end == True:

start.append(i)

end = False

else:

pass

result = []

for start,end **in** zip(start,start[1:]+[len(text)]):

*# lenght = len(text[start:end].strip())*

result.append([start,end])

return result

def word\_decode(pred,b\_pred,offset\_mapping,text,\*\*kwargs):

pred\_head = []

for i,(p,b\_p,mapping) **in** enumerate(zip(pred,b\_pred,offset\_mapping)):

if mapping == (0,0):

continue

if b\_p >= kwargs['begin\_proba'] **or** (abs(p-pred[i-1])>0.1):

pred\_head.append(mapping[0])

sentence = get\_sentence\_split(text,pred\_head)

sentence\_offset\_mapping = [[]]

sentence\_pred = [[]]

sentence\_b\_pred = [[]]

sentence\_idx = [[]]

i\_sentence = 0

for i,(p,b\_p,mapping) **in** enumerate(zip(pred,b\_pred,offset\_mapping)):

if mapping == (0,0) **or** mapping[0]==mapping[1]:

continue

if i\_sentence == len(sentence):

print(text)

print(mapping)

if mapping[1] <= sentence[i\_sentence][1]:

sentence\_offset\_mapping[i\_sentence].append(mapping)

sentence\_pred[i\_sentence].append(p)

sentence\_b\_pred[i\_sentence].append(b\_p)

sentence\_idx[i\_sentence].append(i)

else:

i\_sentence += 1

sentence\_offset\_mapping.append([mapping])

sentence\_pred.append([p])

sentence\_b\_pred.append([b\_p])

sentence\_idx.append([i])

sentence\_offset\_mapping2 = []

sentence\_pred2 = []

sentence\_b\_pred2 = []

sentence\_idx2 = []

for t,mapping\_,pred\_,b\_pred\_,idx\_ **in** zip(sentence,sentence\_offset\_mapping,sentence\_pred,sentence\_b\_pred,sentence\_idx):

*# print(np.round(pred\_,2),'\t',text[t[0]:t[1]])*

if np.mean(pred\_) > kwargs['min\_proba'][0]:

sentence\_offset\_mapping2.append([mapping\_[0][0],mapping\_[-1][1]])

sentence\_pred2.append(pred\_)

sentence\_b\_pred2.append(b\_pred\_)

sentence\_idx2.append(idx\_)

sentence\_offset\_mapping3 = []

sentence\_idx3 = []

for i,mapping\_ **in** enumerate(sentence\_offset\_mapping2):

sep\_start = sentence\_idx2[i-1][-1]+1

sep\_end = sentence\_idx2[i][0]

sep\_length = sep\_end-sep\_start

if i == 0:

sentence\_offset\_mapping3.append(mapping\_)

sentence\_idx3.append(sentence\_idx2[i])

elif sentence\_b\_pred2[i][0] > kwargs['begin\_proba']:

sentence\_offset\_mapping3.append(mapping\_)

sentence\_idx3.append(sentence\_idx2[i])

elif sep\_length >= kwargs['min\_sep']:

sentence\_offset\_mapping3.append(mapping\_)

sentence\_idx3.append(sentence\_idx2[i])

elif sep\_length/kwargs['min\_sep'] - np.mean(pred[sep\_start:sep\_end])/kwargs['min\_proba'][1] > 0:

sentence\_offset\_mapping3.append(mapping\_)

sentence\_idx3.append(sentence\_idx2[i])

else:

sentence\_offset\_mapping3[-1][1] = mapping\_[1]

sentence\_idx3[-1].extend(sentence\_idx2[i])

sentence\_offset\_mapping4 = []

for i,mapping **in** enumerate(sentence\_offset\_mapping3):

word\_length = len(text[mapping[0]: mapping[1]].split())

if word\_length <= kwargs['min\_length']:

continue

*# if word\_length >= kwargs['min\_length'][1]:*

*# continue*

*# if sum(pred[sentence\_idx3[i][0] : sentence\_idx3[i][-1]+1]) <= kwargs['min\_proba'][0]\*kwargs['min\_length'][1]\*1.2: # 整体概率大于 p也算数*

*# continue*

sentence\_offset\_mapping4.append(mapping)

result = [mapping\_to\_ids(mapping,text) for mapping **in** sentence\_offset\_mapping4]

return result

def after\_deal\_helper(train\_ids\_sub,labels\_to\_ids, segment\_param):

y\_pred = []

for i,row **in** train\_ids\_sub:

attention\_mask = np.array(row.attention\_mask)

offset\_mapping = [mapping for mapping,mask **in** zip(row.offset\_mapping,attention\_mask) if mask > 0]

*# token\_label = np.array(row.token\_label)[attention\_mask>0]*

*# b\_pred = row.pred[:,8:].sum(axis=1)*

for discourse **in** [

'Claim',

'Evidence',

'Position',

'Concluding Statement',

'Lead',

'Counterclaim',

'Rebuttal'

]:

if row.pred.shape[1]!=9:

b\_pred = row.pred[:,labels\_to\_ids['B-'+discourse]].copy()

pred = row.pred[:,labels\_to\_ids['I-'+discourse]].copy()

pred = pred + b\_pred

else:

pred = row.pred[:,labels\_to\_ids['I-'+discourse]].copy()

b\_pred = row.pred[:,8].copy()

b\_pred = b\_pred \* pred

pred\_type = word\_decode(pred,b\_pred,offset\_mapping,row.text,

\*\*segment\_param[discourse],

)

for pred\_type\_ **in** pred\_type:

y\_pred.append({'id':row['id'],'class':discourse,'predictionstring':pred\_type\_})

return y\_pred

def after\_deal(data\_pred, labels\_to\_ids, segment\_param, log):

num\_jobs = 16

y\_pred = []

train\_ids = list(data\_pred.sort\_values('id').iterrows())

train\_ids\_splits = np.array\_split(train\_ids, num\_jobs)

results = Parallel(n\_jobs=num\_jobs, backend="multiprocessing")(

delayed(after\_deal\_helper)(train\_ids\_sub, labels\_to\_ids, segment\_param) for train\_ids\_sub **in** train\_ids\_splits

)

for result **in** results:

y\_pred.extend(result)

y\_pred = pd.DataFrame(y\_pred)

print('y\_pred.shape:',y\_pred.shape)

return y\_pred

def model\_predict(model,batch,model\_length,max\_length,padding\_dict,padding\_side='right',duplicate\_cnt=100):

batch\_length = batch["input\_ids"].shape[1]

model\_length = min(batch\_length,max\_length) if model\_length **is** None else model\_length

batch\_length = batch\_length if batch\_length>=model\_length else model\_length

new\_batch = []

for i **in** range(len(batch["input\_ids"])):

new\_batch.append(feat\_padding(batch["input\_ids"][i], batch["attention\_mask"][i],batch["labels"][i],

batch\_length,padding\_dict=padding\_dict,padding\_side=padding\_side))

batch["input\_ids"] = torch.stack([c[0] for c **in** new\_batch])

batch["attention\_mask"] = torch.stack([c[1] for c **in** new\_batch])

batch["labels"] = torch.stack([c[2] for c **in** new\_batch])

ids\_length = batch['input\_ids'].shape[1]

loops = int(np.ceil((ids\_length-duplicate\_cnt)/(model\_length-duplicate\_cnt)))

loops = max(loops,1)

loops\_start = [i\*(model\_length-duplicate\_cnt) for i **in** range(loops)]

loops\_end = [i\*(model\_length-duplicate\_cnt)+model\_length for i **in** range(loops)]

if loops > 1:

loops\_start[-1] = ids\_length-model\_length

loops\_end[-1] = ids\_length

if padding\_side=='left':

loops\_start = [ids\_length-idx for idx **in** loops\_end][::-1]

loops\_end = [idx+model\_length for idx **in** loops\_start]

losses = []

preds = None

for i,(start,end) **in** enumerate(zip(loops\_start,loops\_end)):

device = next(model.parameters()).device

input\_ids = batch["input\_ids"][:,start:end].to(device, dtype = torch.long)

attention\_mask = batch["attention\_mask"][:,start:end].to(device, dtype = torch.long)

labels = batch["labels"][:,start:end].to(device, dtype = torch.long)

model.eval()

with torch.cuda.amp.autocast():

loss, logits = model(input\_ids=input\_ids,attention\_mask=attention\_mask,labels=labels)

loss = loss.mean()

if i == 0:

preds = logits

else:

if i == (loops-1):

if i==1:

inter\_length = loops\_end[0]-loops\_start[-1]

weight = torch.floor\_divide(torch.arange(inter\_length), inter\_length-1).reshape(1,-1,1).to(device)

intersection = preds[:,start:]\*(1-weight) + logits[:,:inter\_length]\*(weight)

preds = torch.cat([preds[:,:start],intersection,logits[:,inter\_length:]],dim=1)

else:

preds = torch.cat([preds[:,:start],logits],dim=1)

else:

preds = torch.cat([preds,logits[:,duplicate\_cnt:]],dim=1)

losses.append(loss)

pred\_list = []

for p,m **in** zip(preds,batch["attention\_mask"]):

m\_index = m.nonzero().reshape(-1).to(device)

pred\_list.append(p.index\_select(0,m\_index).cpu().numpy().astype('float64'))

return torch.tensor(losses).mean(), pred\_list

def sorted\_quantile(array, q):

array = np.array(array)

n = len(array)

index = (n - 1) \* q

left = np.floor(index).astype(int)

fraction = index - left

right = left

right = right + (fraction > 0).astype(int)

i, j = array[left], array[right]

return i + (j - i) \* fraction

def get\_label(text,mapping,predictionstring):

word\_start = len(text[:mapping[0]].split())

word\_end = word\_start + len(text[mapping[0]:mapping[1]].split())

word\_end = min(word\_end, len(text.split()))

pred\_idx = list(range(word\_start, word\_end))

pred\_cnt = len(pred\_idx)

for true\_idx **in** predictionstring:

if len(pred\_idx)==0 **or** true\_idx[0] > pred\_idx[-1] **or** true\_idx[-1] < pred\_idx[0]:

continue

inter\_cnt = len(set(pred\_idx) & set(true\_idx))

true\_cnt = len(true\_idx)

inter\_rate = min(inter\_cnt/pred\_cnt,pred\_cnt/true\_cnt)

if inter\_rate > 0.5:

return 1,inter\_rate,[true\_idx[0],true\_idx[-1]],[word\_start,word\_end]

return 0,0,[-1,-1],[word\_start,word\_end]

import re

def tuple\_map(offset\_mapping,threshold):

paragraph\_rk = []

rk = 0

last = 1

for token\_index **in** offset\_mapping:

if len(threshold) == 0:

paragraph\_rk.append(1)

elif token\_index[1] <= threshold[rk][1]:

last = max(rk+1,last)

paragraph\_rk.append(last)

else:

last = max(rk+2,last)

paragraph\_rk.append(last)

if rk + 1 < len(threshold) - 1:

rk += 1

return paragraph\_rk

def get\_pos\_feat(text, offset\_mapping):

paragraph\_cnt = len(text.split('**\n\n**')) + 1

paragraph\_th = [m.span() for m **in** re.finditer('**\n\n**',text)]

paragraph\_rk = tuple\_map(offset\_mapping,paragraph\_th)

paragraph\_rk\_r = [paragraph\_cnt-rk+1 if rk!=0 else 0 for rk **in** paragraph\_rk]

sentence\_th = []

for i,v **in** enumerate([m.span() for m **in** re.finditer('**\n\n**|\.|,|\?|\!',text)]):

if i == 0:

sentence\_th.append(list(v))

else:

if v[0]==sentence\_th[-1][-1]:

sentence\_th[-1][-1] = v[-1]

else:

sentence\_th.append(list(v))

sentence\_cnt = len(sentence\_th) + 1

sentence\_rk = tuple\_map(offset\_mapping,sentence\_th)

sentence\_rk\_r = [sentence\_cnt-rk+1 if rk!=0 else 0 for rk **in** sentence\_rk]

last\_garagraph\_cnt = 0

sentence\_rk\_of\_paragraph = []

for i **in** range(len(offset\_mapping)):

sentence\_rk\_of\_paragraph.append(sentence\_rk[i]-last\_garagraph\_cnt)

if i+1 == len(offset\_mapping) **or** paragraph\_rk[i]!=paragraph\_rk[i+1]:

last\_garagraph\_cnt = sentence\_rk[i]

sentence\_cnt\_of\_paragraph = []

last\_max = None

for i **in** range(1,len(offset\_mapping)+1):

if i==1 **or** paragraph\_rk[-i] != paragraph\_rk[-i+1]:

last\_max = sentence\_rk\_of\_paragraph[-i]

sentence\_cnt\_of\_paragraph.append(last\_max)

sentence\_cnt\_of\_paragraph = sentence\_cnt\_of\_paragraph[::-1]

sentence\_rk\_r\_of\_paragraph = [s\_cnt-rk+1 if rk!=0 else 0 for s\_cnt,rk **in** zip(sentence\_cnt\_of\_paragraph,sentence\_rk\_of\_paragraph)]

return paragraph\_cnt,sentence\_cnt,paragraph\_rk,paragraph\_rk\_r,sentence\_rk,sentence\_rk\_r, \

sentence\_cnt\_of\_paragraph,sentence\_rk\_of\_paragraph,sentence\_rk\_r\_of\_paragraph

// Overall, these functions facilitate the extraction of positional features from text data, which can be useful for various natural language processing tasks, such as sequence labeling or classification.

In [ ]:

import copy

import torch

from torch import nn

import torch.nn.functional as F

from sklearn.metrics import accuracy\_score

from torch.nn import CrossEntropyLoss

from transformers import AutoTokenizer, AutoModel, AutoConfig

class TextModel(nn.Module):

def \_\_init\_\_(self,model\_name=None,num\_labels=1):

super(TextModel,self).\_\_init\_\_()

config = AutoConfig.from\_pretrained(model\_name)

self.model = AutoModel.from\_pretrained(model\_name) *# 768*

self.drop\_out = nn.Dropout(0.1)

self.dropout1 = nn.Dropout(0.1)

self.dropout2 = nn.Dropout(0.2)

self.dropout3 = nn.Dropout(0.3)

self.dropout4 = nn.Dropout(0.4)

self.dropout5 = nn.Dropout(0.5)

self.output = nn.Linear(config.hidden\_size,num\_labels)

if 'deberta-v2-xxlarge' **in** model\_name:

self.model.embeddings.requires\_grad\_(False)

self.model.encoder.layer[:24].requires\_grad\_(False) *# 冻结24/48*

if 'deberta-v2-xlarge' **in** model\_name:

self.model.embeddings.requires\_grad\_(False)

self.model.encoder.layer[:12].requires\_grad\_(False) *# 冻结12/24*

def forward(self, input\_ids, attention\_mask, labels=None):

if 'gpt' **in** self.model.name\_or\_path:

emb = self.model(input\_ids)[0]

else:

emb = self.model(input\_ids,attention\_mask)[0]

preds1 = self.output(self.dropout1(emb))

preds2 = self.output(self.dropout2(emb))

preds3 = self.output(self.dropout3(emb))

preds4 = self.output(self.dropout4(emb))

preds5 = self.output(self.dropout5(emb))

preds = (preds1 + preds2 + preds3 + preds4 + preds5) / 5

logits = torch.softmax(preds, dim=-1)

if labels **is** **not** None:

loss = self.get\_loss(preds,labels,attention\_mask)

return loss,logits

else:

return logits

def get\_loss(self, outputs, targets, attention\_mask):

loss\_fct = nn.CrossEntropyLoss()

active\_loss = attention\_mask.reshape(-1) == 1

active\_logits = outputs.reshape(-1, outputs.shape[-1])

true\_labels = targets.reshape(-1)

idxs = np.where(active\_loss.cpu().numpy() == 1)[0]

active\_logits = active\_logits[idxs]

true\_labels = true\_labels[idxs].to(torch.long)

loss = loss\_fct(active\_logits, true\_labels)

return loss

In [ ]:

import os

os.environ["TOKENIZERS\_PARALLELISM"] = "false"

*# os.environ['CUDA\_VISIBLE\_DEVICES'] = '1'*

CUDA\_LAUNCH\_BLOCKING=1

import pickle

import random

import numpy as np

import pandas as pd

from tqdm import tqdm

from torch.utils.data import Dataset, DataLoader

from transformers import AutoTokenizer

import pdb

import torch

from torch import nn

from torch import cuda

import warnings

warnings.filterwarnings('ignore')

import argparse

def get\_args():

parser = argparse.ArgumentParser()

parser.add\_argument('--seed', type=int, default=66)

parser.add\_argument('--device', type=str, default='cuda')

parser.add\_argument('--log\_path', type=str, default="log.txt")

parser.add\_argument('--data\_path', type=str, default="feedback/")

parser.add\_argument('--text\_path', type=str, default="feedback/train/")

parser.add\_argument('--cache\_path', type=str, default="cache/")

parser.add\_argument('--model\_name', type=str, default="longformer-base-4096/")

parser.add\_argument("--fold", type=int, default=0)

parser.add\_argument('--train\_batch\_size', type=int, default=4)

parser.add\_argument('--valid\_batch\_size', type=int, default=4)

parser.add\_argument('--max\_length', type=int, default=1024)

parser.add\_argument('--epochs', type=int, default=4)

parser.add\_argument('--lr', type=float, default=0.00001)

parser.add\_argument('--min\_lr', type=float, default=0.000001)

parser.add\_argument('--max\_lr', type=float, default=0.00001)

parser.add\_argument('--adv\_lr', type=float, default=0.0000)

parser.add\_argument('--adv\_eps', type=float, default=0.001)

parser.add\_argument('--max\_grad\_norm', type=float, default=10)

parser.add\_argument('--debug', action='store\_true')

parser.add\_argument('--load\_model', action='store\_true')

parser.add\_argument('--load\_feat', action='store\_true')

parser.add\_argument('--key\_string', type=str, default='')

args = parser.parse\_args()

if args.debug:

args.epochs = 2

args.key\_string = args.model\_name.split('/')[-2] + \

'\_v2\_15class\_adv' + \

(f'\_adv**{**args.adv\_lr**}**' if args.adv\_lr>0 else '') + \

f'\_fold**{**args.fold**}**' + \

('\_debug' if args.debug else '')

log = Log(f'log/**{**args.key\_string**}**.log',time\_key=False)

log('args:**{}**'.format(str(args)))

return args,log

*# Function to seed everything*

def seed\_everything(seed: int):

random.seed(seed)

os.environ["PYTHONHASHSEED"] = str(seed)

np.random.seed(seed)

torch.manual\_seed(seed)

torch.cuda.manual\_seed(seed)

torch.backends.cudnn.deterministic = True

torch.backends.cudnn.benchmark = True

if \_\_name\_\_ == "\_\_main\_\_":

args,log = get\_args()

seed\_everything(args.seed)

discourse\_type = ['Claim','Evidence', 'Position','Concluding Statement','Lead','Counterclaim','Rebuttal']

i\_discourse\_type = ['I-'+i for i **in** discourse\_type]

b\_discourse\_type = ['B-'+i for i **in** discourse\_type]

args.labels\_to\_ids = {k:v for v,k **in** enumerate(['O']+i\_discourse\_type+b\_discourse\_type)}

args.ids\_to\_labels = {k:v for v,k **in** args.labels\_to\_ids.items()}

df = pd.read\_csv(os.path.join(args.data\_path, "train\_folds.csv"))

train\_df = df[df["kfold"] != args.fold].reset\_index(drop=True)

test\_df = df[df["kfold"] == args.fold].reset\_index(drop=True)

if args.debug:

sample\_id = train\_df['id'].drop\_duplicates().sample(frac=0.1).values

train\_df = train\_df[train\_df['id'].isin(sample\_id)].reset\_index(drop=True)

sample\_id = test\_df['id'].drop\_duplicates().sample(frac=0.1).values

test\_df = test\_df[test\_df['id'].isin(sample\_id)].reset\_index(drop=True)

log('train\_df.shape:',train\_df.shape,'**\t**','test\_df.shape:',test\_df.shape)

tokenizer = AutoTokenizer.from\_pretrained(args.model\_name)

train\_feat = get\_feat(train\_df,tokenizer,args,'train\_feat'+args.key\_string)

test\_feat = get\_feat(test\_df,tokenizer,args,'test\_feat'+args.key\_string)

*# train\_feat = prepare\_training\_data(train\_df, tokenizer, args, 8)*

*# test\_feat = prepare\_training\_data(test\_df, tokenizer, args, 8)*

log("train\_feat.shap: **{}**".format(train\_feat.shape),'**\t**',"test\_feat.shape: **{}**".format(test\_feat.shape))

train\_params = {'batch\_size': args.train\_batch\_size,

'shuffle': True, 'num\_workers': 2, 'pin\_memory':True,

'collate\_fn':Collate(args.max\_length)

}

test\_params = {'batch\_size': args.valid\_batch\_size,

'shuffle': False, 'num\_workers': 2,'pin\_memory':True,

'collate\_fn':Collate(4096)

}

train\_loader = DataLoader(dataset(train\_feat), \*\*train\_params)

test\_loader = DataLoader(dataset(test\_feat), \*\*test\_params)

args.num\_train\_steps = len(train\_feat) \* args.epochs / args.train\_batch\_size

*# CREATE MODEL*

model = TextModel(args.model\_name, num\_labels=len(args.labels\_to\_ids))

model = torch.nn.DataParallel(model)

model.to(args.device)

model\_path = f'**{**args.cache\_path + args.key\_string**}**.pt'

if args.load\_model **and** os.path.exists(model\_path):

model.load\_state\_dict(torch.load( model\_path))

log(f'Model loaded from **{**model\_path**}**')

te\_loss, te\_accuracy,f1\_score, test\_pred = test(model,test\_loader,test\_feat,test\_df,args,log)

else:

model,test\_pred = train(model,train\_loader,test\_loader,test\_feat,test\_df,args,model\_path,log)

torch.save(model.state\_dict(), "model.bin")