import os

PATH = "./src/automatic\_video\_chaptering/video\_preparation\_scripts"

def run\_stage(stage\_name, script\_path):

print(f">>>>>>Stage {stage\_name} started <<<<<<")

res = os.system(f"python {script\_path}")

if res == 0:

print(f">>>>>>Stage {stage\_name} completed <<<<<<\n\nx==========x")

else:

print(f">>>>>>Stage {stage\_name} exited with error code {res} <<<<<<\n\nx==========x")

exit()

def main():

# Define stages

stages = [

# ("Downloading Videos from YouTube", f"{PATH}/download\_all\_videos/download\_videos.py"),

# ("Extracting Videos into Frames", f"{PATH}/extract\_videos\_to\_frames/extract\_video\_to\_frames.py"),

# ("Finding Videos Based on Criteria", f"{PATH}/find\_and\_discard\_all\_bad\_videos/find\_bad\_vid.py"),

# ("Discarding Videos Based on Criteria", f"{PATH}/find\_and\_discard\_all\_bad\_videos/discard\_bad\_vid.py"),

]

# Execute stages

for stage\_name, script\_path in stages:

run\_stage(stage\_name, script\_path

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

main()

print("All stages completed successfully.")

--------------------------

import random

import multiprocessing

import os, glob

# import youtube\_dl

import yt\_dlp as youtube\_dl

import pandas as pd

from automatic\_video\_chaptering.utilities.data\_utils import multiple\_process\_utils

from automatic\_video\_chaptering.utilities.data\_utils.load\_dataset\_utils import parse\_csv\_to\_list

def download\_youtube\_video(save\_dir, all\_vids):

idx = 0

for vid in all\_vids:

save\_filename = os.path.join(save\_dir, f"{vid}.mp4")

print(f"downloading {vid}...")

link = "https://www.youtube.com/watch?v=" + vid

try:

ydl\_opts = {

'outtmpl': save\_filename,

"format": "18"

}

with youtube\_dl.YoutubeDL(ydl\_opts) as ydl:

dictMeta = ydl.extract\_info(f"https://www.youtube.com/watch?v={vid}", download=False)

duration = dictMeta["duration"]

if duration <= 600: # do not consider videos greater than 10 minutes

ydl.download([link])

idx+=1

print(f"{idx} videos downloaded from a total of size {len(all\_vids)}")

print("Note: Only videos under 20 minutes will be downloaded")

print("Hence: The number of downloaded videos might not be equal to the total size")

except Exception as e:

print(f"Exception to download {vid}")

print(f"{e}")

print()

return None

def multiple\_process\_run(save\_dir, all\_vids, process\_num=10):

pool = multiprocessing.Pool(process\_num)

chunked\_data = multiple\_process\_utils.split\_data(process\_num, all\_vids)

for i, d in enumerate(chunked\_data):

pool.apply\_async(download\_youtube\_video,

args=(save\_dir, d),

error\_callback=multiple\_process\_utils.subprocess\_print\_err)

print('Waiting for all subprocesses done...')

pool.close()

pool.join()

print('All subprocesses done.')

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

# video\_save\_dir = "/videos/"

video\_save\_dir = "./videos/"

df = pd.read\_csv("./dataset/all\_in\_one\_with\_subtitle.csv")

all\_vids = df.videoId

all\_vids = list(set(all\_vids))

need\_to\_download\_vids = []

# Skip all already downloaded videos

for vid in all\_vids:

if os.path.exists(os.path.join(video\_save\_dir, f"{vid}.mp4")):

print(f"{vid}.mp4 already exists and will not be downloaded!")

continue

need\_to\_download\_vids.append(vid)

print(f"{len(all\_vids) - len(need\_to\_download\_vids)}/{len(all\_vids)}, {len(need\_to\_download\_vids)} videos need to be downloaded")

# Download video for only a subset of the whole data

num\_values\_to\_return = 100

need\_to\_download\_vids = random.sample(need\_to\_download\_vids, num\_values\_to\_return)

# download\_youtube\_video(video\_save\_dir, all\_vids)

# download\_youtube\_video(video\_save\_dir, need\_to\_download\_vids)

multiple\_process\_run(video\_save\_dir, need\_to\_download\_vids)

--------------------------

import os, glob

import shutil

import pandas as pd

import multiprocessing

from automatic\_video\_chaptering.utilities.data\_utils import multiple\_process\_utils

""" extract all video """

def extract\_frame\_fn(process\_idx, video\_files, durations, video\_frame\_dir, extract\_fps=1):

for i, video\_file in enumerate(video\_files):

if durations is not None and durations[i] > 600:

continue

vid = os.path.basename(video\_file).split(".")[0]

print(f"process {process\_idx}, extract video {i}/{len(video\_files)}, {vid}...")

frame\_save\_dir = os.path.join(video\_frame\_dir, vid)

if os.path.exists(frame\_save\_dir):

if durations is not None:

img\_num = len(glob.glob(frame\_save\_dir + "/\*.jpg"))

if img\_num < int(durations[i]) - 1:

shutil.rmtree(frame\_save\_dir)

else:

continue

else:

shutil.rmtree(frame\_save\_dir)

os.makedirs(frame\_save\_dir)

save\_path = frame\_save\_dir + "/%05d.jpg"

os.system(f"ffmpeg -i {video\_file} -s 224x224 -r {extract\_fps} {save\_path}")

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

data = pd.read\_csv("./dataset/all\_in\_one\_with\_subtitle.csv")

vids = list(data["videoId"].values)

durations = list(data["duration"].values)

video\_dir = "./videos"

video\_files = [video\_dir + "/" + x + ".mp4" for x in vids]

video\_frame\_dir = "./youtube\_video\_frame\_dataset/"

os.makedirs(video\_frame\_dir, exist\_ok=True)

extract\_fps = 1

# multi processing

process\_num = 8

pool = multiprocessing.Pool(process\_num)

chunked\_data = multiple\_process\_utils.split\_data(process\_num, video\_files)

chunked\_duration\_data = multiple\_process\_utils.split\_data(process\_num, durations)

for i, d in enumerate(chunked\_data):

chunked\_dura = chunked\_duration\_data[i]

pool.apply\_async(extract\_frame\_fn,

args=(i, d, chunked\_dura, video\_frame\_dir, extract\_fps),

error\_callback=multiple\_process\_utils.subprocess\_print\_err)

print('Waiting for all subprocesses done...')

pool.close()

pool.join()

print('All subprocesses done.')

--------------------------------------------

"""

delete bad vid (wrong data) from a list

"""

import pandas as pd

from automatic\_video\_chaptering.utilities.data\_utils.load\_dataset\_utils import parse\_csv\_to\_list

from automatic\_video\_chaptering.utilities.data\_utils.constants import TIMESTAMP\_DELIMITER

def remove\_bad\_vids():

with open("./bad\_videos/bad\_video\_ids.txt", "r", encoding="utf-8") as f:

bad\_vid\_ids = f.readlines()

bad\_vid\_ids = [line.strip() for line in bad\_vid\_ids]

data\_file = "./dataset/all\_in\_one\_with\_subtitle.csv"

new\_data\_file = "./dataset/all\_in\_one\_with\_subtitle\_new.csv"

all\_vids, all\_titles, all\_durations, all\_timestamps = parse\_csv\_to\_list(data\_file)

vids = []

titles = []

durations = []

timestamps = []

for i in range(len(all\_vids)):

vid = all\_vids[i]

if vid in bad\_vid\_ids:

continue

vids.append(vid)

titles.append(all\_titles[i])

durations.append(all\_durations[i])

# convert timestamp list to string

timestamp = all\_timestamps[i]

timestamp = TIMESTAMP\_DELIMITER.join(timestamp)

timestamps.append(timestamp)

all\_in\_one = {

"videoId": vids,

"title": titles,

"duration": durations,

"timestamp": timestamps

}

data = pd.DataFrame(all\_in\_one)

data.to\_csv(new\_data\_file)

print("Bad videos have been discarded.")

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

remove\_bad\_vids()

------------------------------------

import os

import pandas as pd

from automatic\_video\_chaptering.utilities.data\_utils.load\_dataset\_utils import parse\_csv\_to\_list

from automatic\_video\_chaptering.utilities.data\_utils.load\_dataset\_utils import extract\_first\_timestamp

def save\_bad\_video\_ids(bad\_vids\_ids):

os.makedirs("./bad\_videos", exist\_ok=True)

with open("./bad\_videos/bad\_video\_ids.txt", "w", encoding="utf-8") as f:

for vid\_id in bad\_vids\_ids:

f.write(vid\_id + "\n")

def find\_timestamp\_too\_close(all\_timestamps, time\_gap=8):

bad\_indices = []

for i, timestamps in enumerate(all\_timestamps):

timepoint\_secs = []

descriptions = []

for line in timestamps:

sec, description = extract\_first\_timestamp(line)

if len(timepoint\_secs) > 0:

if sec - timepoint\_secs[-1] < time\_gap:

bad\_indices.append(i)

break

timepoint\_secs.append(sec)

descriptions.append(description)

return bad\_indices

def find\_duration\_too\_short(durations, threshold=100):

bad\_indices = []

for i, duration in enumerate(durations):

if duration < threshold:

bad\_indices.append(i)

return bad\_indices

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

data\_file = "./dataset/all\_in\_one\_with\_subtitle.csv"

all\_vids, all\_titles, all\_durations, all\_timestamps = parse\_csv\_to\_list(data\_file)

# a = find\_timestamp\_too\_close(all\_timestamps, 10)

bad\_indices = find\_timestamp\_too\_close(all\_timestamps) + find\_duration\_too\_short(all\_durations)

bad\_vids = [all\_vids[i] for i in bad\_indices]

print(len(bad\_vids))

print(bad\_vids)

save\_bad\_video\_ids(bad\_vids)

----------------------------------------

#dataset scripting

import os

PATH = "./src/automatic\_video\_chaptering/data\_preparation\_scripts"

def run\_stage(stage\_name, script\_path):

print(f">>>>>>Stage {stage\_name} started <<<<<<")

res = os.system(f"python {script\_path}")

if res == 0:

print(f">>>>>>Stage {stage\_name} completed <<<<<<\n\nx==========x")

else:

print(f">>>>>>Stage {stage\_name} exited with error code {res} <<<<<<\n\nx==========x")

exit()

def main():

# Define stages

stages = [

# ("Wikihow Scraping Stage", f"{PATH}/wikihow\_web\_scraping/script.py"),

# ("Cleaning and Saving Scraped Wikihow Data", f"{PATH}/clean\_scraped\_wikihow\_data/clean\_scraped\_data.py"),

("Querying YT API and Extracting Videos into Chapters", f"{PATH}/extract\_videos\_into\_chapters/extract\_videos\_into\_chapters.py"),

("Merge all Data with Subtitles", f"{PATH}/merge\_all\_youtube\_data/merge\_all\_data.py"),

("Splitting Dataset into Train, and Test Sets", f"{PATH}/split\_dataset\_into\_train\_test/split\_dataset.py"),

("Obtaining final Dataset", f"{PATH}/obtain\_final\_dataset/obtain\_final\_dataset.py")

]

# Execute stages

for stage\_name, script\_path in stages:

run\_stage(stage\_name, script\_path)

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

main()

print("All stages completed successfully.")

----------------------------------

import os

from flask import Flask, render\_template, request

from werkzeug.utils import secure\_filename

import json

app = Flask(\_\_name\_\_, static\_folder='static')

UPLOAD\_FOLDER = 'static'

ALLOWED\_EXTENSIONS = {'mp4', 'avi', 'mov', 'mkv'}

app.config['UPLOAD\_FOLDER'] = UPLOAD\_FOLDER

def allowed\_file(filename):

return '.' in filename and filename.rsplit('.', 1)[1].lower() in ALLOWED\_EXTENSIONS

def get\_timestamps\_subtitles():

# Read JSON file

with open('model/summarize.json', 'r') as file:

timestamps\_summaries = json.load(file)

# Get the timestamps and corresponding subtitles

timestamps = list(timestamps\_summaries.keys())

summaries = list(timestamps\_summaries.values())

return timestamps, summaries

@app.route('/')

def index():

return render\_template('index.html')

@app.route('/upload', methods=['POST'])

def upload():

if 'file' not in request.files:

return render\_template('index.html', error='No file part')

file = request.files['file']

if file.filename == '':

return render\_template('index.html', error='No selected file')

if file and allowed\_file(file.filename):

filename = secure\_filename(file.filename)

file\_path = os.path.join(app.config['UPLOAD\_FOLDER'], filename)

file.save(file\_path)

timestamps, summaries = get\_timestamps\_subtitles()

print(timestamps)

print(summaries)

return render\_template('index.html', filename=filename, timestamps=timestamps, summaries=summaries)

return render\_template('index.html', error='Invalid file format')

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

--------------------------------------

import setuptools

\_\_version\_\_ = "0.0.0"

SRC\_REPO = "automatic\_video\_chaptering"

setuptools.setup(

name=SRC\_REPO,

version=\_\_version\_\_,

package\_dir={"": "src"},

packages=setuptools.find\_packages(where="src")

)

---------------------------------------

from tqdm import tqdm

import pandas as pd

import matplotlib.pyplot as plt

import torch

from transformers import PegasusForConditionalGeneration, PegasusTokenizer

# from transformers import pipeline, set\_seed

# import warnings

# warnings.filterwarnings("ignore")

-----------------------------------------

model\_name = "pegasus\_trained\_model"

tokenizer\_name = "pegasus\_trained\_tokenizer"

model = PegasusForConditionalGeneration.from\_pretrained(model\_name)

tokenizer = PegasusTokenizer.from\_pretrained(tokenizer\_name)

def generate\_summary(input\_text, max\_length=20, length\_penalty=0.8, num\_beams=4):

inputs = tokenizer.encode("summarize: " + input\_text, return\_tensors="pt", max\_length=1024, truncation=True)

summary\_ids = model.generate(inputs, max\_length=max\_length, length\_penalty=length\_penalty, num\_beams=num\_beams, early\_stopping=True)

summary = tokenizer.decode(summary\_ids[0], skip\_special\_tokens=True)

return summary

summary = generate\_summary(input\_text)

print("Input Text:", input\_text)

print("Generated Summary:", summary)

-------------------------------------

import os

import argparse

import math

import logging

from tqdm import tqdm

import numpy as np

import torch

import torch.optim as optim

from torch.nn import functional as F

from torch.optim.lr\_scheduler import LambdaLR

from torch.utils.data import DataLoader

from torch.utils.tensorboard import SummaryWriter

from transformers import PegasusForConditionalGeneration, PegasusTokenizer

from all\_utility\_scripts import \*

-----------------------------------------

logger = logging.getLogger(\_\_name\_\_)

# sample a chapter for each video

class YoutubeChapterTitleDataset:

def \_\_init\_\_(self, data\_file, vid\_file, tokenizer, max\_text\_len=512, chapter\_title\_text\_len=30, transform=None, target\_transform=None):

self.tokenizer = tokenizer

self.max\_text\_len = max\_text\_len

self.chapter\_title\_text\_len = chapter\_title\_text\_len

all\_vids, titles, durations, timestamps = parse\_csv\_to\_list(data\_file)

self.vid2title = dict()

self.vid2timestamps = dict()

self.vid2durations = dict()

for i in range(len(all\_vids)):

vid = all\_vids[i]

self.vid2title[vid] = titles[i]

self.vid2timestamps[vid] = timestamps[i]

self.vid2durations[vid] = durations[i]

with open(vid\_file, "r") as f:

vids = f.readlines()

vids = [x.strip() for x in vids]

self.vids = vids

# get asr file

asr\_file\_list = glob.glob(os.path.dirname(data\_file) + "/\*/subtitle\_\*.json")

self.vid2asr\_files = dict()

for asr\_file in asr\_file\_list:

filename = os.path.basename(asr\_file)

vid = filename.split(".")[0][9:]

self.vid2asr\_files[vid] = asr\_file

self.transform = transform

self.target\_transform = target\_transform

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

vid = self.vids[i]

# print(vid)

timestamp = self.vid2timestamps[vid]

asr\_file = self.vid2asr\_files[vid]

duration = round(self.vid2durations[vid] - 1) # equal to video total duration (seconds)

with open(asr\_file, "r") as f:

subtitle = json.load(f)

# extract timestamp

timepoint\_secs = []

descriptions = []

for line in timestamp:

sec, description = extract\_first\_timestamp(line)

timepoint\_secs.append(sec)

descriptions.append(description)

# randomly select a chapter

chapter\_idx = random.randint(0, len(descriptions) - 1)

description = descriptions[chapter\_idx]

description = clean\_str(description)

description = remove\_timestamp(description)

description = description.lower()

chapter\_start\_t = timepoint\_secs[chapter\_idx]

if chapter\_idx + 1 < len(timepoint\_secs):

chapter\_end\_t = timepoint\_secs[chapter\_idx + 1]

else:

chapter\_end\_t = duration

# get subtitle within selected chapter

time\_gap = 1

text\_within\_chapter = ""

for sub in subtitle:

text = sub["text"]

start = sub["start"]

if chapter\_start\_t - time\_gap < start < chapter\_end\_t + time\_gap:

if text\_within\_chapter == "":

text\_within\_chapter = text

else:

text\_within\_chapter += " " + text

if start >= chapter\_end\_t + time\_gap:

break

token\_list = text\_within\_chapter.split(" ")

text\_within\_chapter = " ".join(token\_list)

text\_within\_chapter = text\_within\_chapter.lower()

# process input text (truncate and pad)

pad\_token = self.tokenizer.pad\_token

tokens = self.tokenizer.tokenize(text\_within\_chapter)

tokens = tokens[:self.max\_text\_len]

attention\_mask = [1] \* len(tokens)

if len(tokens) < self.max\_text\_len:

zero\_pad\_list = [0] \* (self.max\_text\_len - len(tokens))

pad\_list = [pad\_token] \* (self.max\_text\_len - len(tokens))

tokens += pad\_list

attention\_mask += zero\_pad\_list

# process summarization text (truncate and pad and shift right to make label)

# bos\_token = self.tokenizer.bos\_token

# if bos\_token is None: # google/pegasus-large has no bos\_token, but model config use pad token as decoder\_start\_token\_id

# bos\_token = pad\_token

bos\_token = pad\_token

decode\_tokens = self.tokenizer.tokenize(description)

input\_decode\_tokens = [bos\_token] + decode\_tokens # bos\_token is decoder\_start\_token

input\_decode\_tokens = input\_decode\_tokens[:self.chapter\_title\_text\_len]

eos\_token = self.tokenizer.eos\_token

if len(decode\_tokens) >= self.chapter\_title\_text\_len:

target\_decode\_tokens = decode\_tokens

target\_decode\_tokens[self.chapter\_title\_text\_len - 1] = eos\_token

else:

target\_decode\_tokens = decode\_tokens + [eos\_token]

target\_decode\_tokens = target\_decode\_tokens[:self.chapter\_title\_text\_len]

decode\_attention\_mask = [1] \* (len(decode\_tokens) + 1)

decode\_attention\_mask = decode\_attention\_mask[:self.chapter\_title\_text\_len]

if len(decode\_attention\_mask) < self.chapter\_title\_text\_len:

zero\_pad\_list = [0] \* (self.chapter\_title\_text\_len - len(decode\_attention\_mask))

pad\_list = [eos\_token] \* (self.chapter\_title\_text\_len - len(decode\_attention\_mask))

input\_decode\_tokens += pad\_list

target\_decode\_tokens += pad\_list

decode\_attention\_mask += zero\_pad\_list

# Convert token to vocabulary indices

ids = self.tokenizer.convert\_tokens\_to\_ids(tokens)

text\_ids = torch.from\_numpy(np.array(ids)).long()

attention\_mask = torch.from\_numpy(np.array(attention\_mask)).long()

input\_decode\_ids = self.tokenizer.convert\_tokens\_to\_ids(input\_decode\_tokens)

input\_decode\_ids = torch.from\_numpy(np.array(input\_decode\_ids)).long()

decode\_attention\_mask = torch.from\_numpy(np.array(decode\_attention\_mask)).long()

target\_decode\_ids = self.tokenizer.convert\_tokens\_to\_ids(target\_decode\_tokens)

target\_decode\_ids = torch.from\_numpy(np.array(target\_decode\_ids)).long()

return text\_ids, attention\_mask, input\_decode\_ids, decode\_attention\_mask, target\_decode\_ids

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

return len(self.vids)

-------------------------------------

class PegasusHugface(nn.Module):

def \_\_init\_\_(self, reinit\_head=False):

super().\_\_init\_\_()

# load pretrained base model

self.base\_model = PegasusForConditionalGeneration.from\_pretrained('google/pegasus-large', output\_attentions=True)

self.vocab\_size = self.base\_model.config.vocab\_size

self.embed\_size = self.base\_model.config.hidden\_size

# init head

if reinit\_head:

self.base\_model.lm\_head = nn.Linear(self.embed\_size, self.vocab\_size, bias=False)

self.base\_model.lm\_head.weight.data.normal\_(mean=0.0, std=0.02)

if self.base\_model.lm\_head.bias is not None:

self.base\_model.lm\_head.bias.data.zero\_()

# number of parameters: 109482240

print("number of parameters: ", sum(p.numel() for p in self.base\_model.parameters()))

def fix\_backbone(self):

# param\_dict = {pn: p for pn, p in self.named\_parameters()}

for pn, p in self.named\_parameters():

if "pooler" in pn or "head" in pn:

continue

p.requires\_grad = False

def configure\_optimizers(self, train\_config):

# separate out all parameters to those that will and won't experience regularizing weight decay

decay = set()

no\_decay = set()

# for mn, m in self.named\_modules():

for pn, p in self.named\_parameters():

# fpn = '%s.%s' % (mn, pn) if mn else pn # full param name

fpn = pn

if pn.endswith('bias'):

# all biases will not be decayed

no\_decay.add(fpn)

elif "layer\_norm" in fpn:

no\_decay.add(fpn)

elif "bn" in fpn:

no\_decay.add(fpn)

elif "emb" in fpn:

no\_decay.add(fpn)

else:

# weights of whitelist modules will be weight decayed

decay.add(fpn)

# validate that we considered every parameter

param\_dict = {pn: p for pn, p in self.named\_parameters()}

inter\_params = decay & no\_decay

union\_params = decay | no\_decay

assert len(inter\_params) == 0, "parameters %s made it into both decay/no\_decay sets!" % (str(inter\_params), )

assert len(param\_dict.keys() - union\_params) == 0, "parameters %s were not separated into either decay/no\_decay set!" % (str(param\_dict.keys() - union\_params), )

# create the pytorch optimizer object

optim\_groups = [

{"params": [param\_dict[pn] for pn in list(decay)], "weight\_decay": train\_config.weight\_decay},

{"params": [param\_dict[pn] for pn in list(no\_decay)], "weight\_decay": 0.0},

]

optimizer = torch.optim.AdamW(optim\_groups, lr=train\_config.learning\_rate, betas=train\_config.betas)

return optimizer

def forward(self, x, attention\_mask=None, decoder\_input\_ids=None, decoder\_attention\_mask=None):

"""

"""

inputs = {

"input\_ids": x,

"attention\_mask": attention\_mask,

"decoder\_input\_ids": decoder\_input\_ids,

"decoder\_attention\_mask": decoder\_attention\_mask

}

base\_output = self.base\_model(\*\*inputs)

logits = base\_output.logits

return logits

def generate(self, text, tokenizer, device, max\_text\_len=512, max\_len=30, temperature=1.0, sample=False, top\_k=None):

tokens = tokenizer.tokenize(text)

tokens = tokens[:max\_text\_len]

ids = tokenizer.convert\_tokens\_to\_ids(tokens)

input\_ids = torch.from\_numpy(np.array(ids)).long().to(device)

input\_ids = input\_ids.unsqueeze(0)

decoder\_start\_token\_id = self.base\_model.config.decoder\_start\_token\_id

# decoder\_start\_token = tokenizer.convert\_ids\_to\_tokens([decoder\_start\_token\_id])

decoder\_input\_ids = torch.from\_numpy(np.array([decoder\_start\_token\_id])).long().to(device)

decoder\_input\_ids = decoder\_input\_ids.unsqueeze(0)

i = 0

sentence\_ids = []

sentence\_logits = []

while i < max\_len:

with torch.no\_grad():

logits = self.forward(input\_ids, decoder\_input\_ids=decoder\_input\_ids)

sentence\_logits.append(logits)

logits = logits[:, -1, :] / temperature

# optionally crop probabilities to only the top k options

if top\_k is not None:

logits = top\_k\_logits(logits, top\_k)

# apply softmax to convert to probabilities

probs = F.softmax(logits, dim=-1)

# sample from the distribution or take the most likely

if sample:

ix = torch.multinomial(probs, num\_samples=1)

else:

\_, ix = torch.topk(probs, k=1, dim=-1)

# append to the sequence and continue

decoder\_input\_ids = torch.cat((decoder\_input\_ids, ix), dim=1)

sentence\_ids.append(ix.squeeze(0).item())

if sentence\_ids[-1] == self.base\_model.config.eos\_token\_id:

break

i += 1

sentence\_logits = torch.cat(sentence\_logits, dim=1)

gen\_text = tokenizer.decode(sentence\_ids)

return gen\_text, sentence\_logits

---------------------------------------------

class PegasusHugface(nn.Module):

def \_\_init\_\_(self, reinit\_head=False):

super().\_\_init\_\_()

# load pretrained base model

self.base\_model = PegasusForConditionalGeneration.from\_pretrained('google/pegasus-large', output\_attentions=True)

self.vocab\_size = self.base\_model.config.vocab\_size

self.embed\_size = self.base\_model.config.hidden\_size

# init head

if reinit\_head:

self.base\_model.lm\_head = nn.Linear(self.embed\_size, self.vocab\_size, bias=False)

self.base\_model.lm\_head.weight.data.normal\_(mean=0.0, std=0.02)

if self.base\_model.lm\_head.bias is not None:

self.base\_model.lm\_head.bias.data.zero\_()

# number of parameters: 109482240

print("number of parameters: ", sum(p.numel() for p in self.base\_model.parameters()))

def fix\_backbone(self):

# param\_dict = {pn: p for pn, p in self.named\_parameters()}

for pn, p in self.named\_parameters():

if "pooler" in pn or "head" in pn:

continue

p.requires\_grad = False

def configure\_optimizers(self, train\_config):

"""

This long function is unfortunately doing something very simple and is being very defensive:

We are separating out all parameters of the model into two buckets: those that will experience

weight decay for regularization and those that won't (biases, and layernorm/embedding weights).

We are then returning the PyTorch optimizer object.

"""

# separate out all parameters to those that will and won't experience regularizing weight decay

decay = set()

no\_decay = set()

# for mn, m in self.named\_modules():

for pn, p in self.named\_parameters():

# fpn = '%s.%s' % (mn, pn) if mn else pn # full param name

fpn = pn

if pn.endswith('bias'):

# all biases will not be decayed

no\_decay.add(fpn)

elif "layer\_norm" in fpn:

no\_decay.add(fpn)

elif "bn" in fpn:

no\_decay.add(fpn)

elif "emb" in fpn:

no\_decay.add(fpn)

else:

# weights of whitelist modules will be weight decayed

decay.add(fpn)

# validate that we considered every parameter

param\_dict = {pn: p for pn, p in self.named\_parameters()}

inter\_params = decay & no\_decay

union\_params = decay | no\_decay

assert len(inter\_params) == 0, "parameters %s made it into both decay/no\_decay sets!" % (str(inter\_params), )

assert len(param\_dict.keys() - union\_params) == 0, "parameters %s were not separated into either decay/no\_decay set!" % (str(param\_dict.keys() - union\_params), )

# create the pytorch optimizer object

optim\_groups = [

{"params": [param\_dict[pn] for pn in list(decay)], "weight\_decay": train\_config.weight\_decay},

{"params": [param\_dict[pn] for pn in list(no\_decay)], "weight\_decay": 0.0},

]

optimizer = torch.optim.AdamW(optim\_groups, lr=train\_config.learning\_rate, betas=train\_config.betas)

return optimizer

def forward(self, x, attention\_mask=None, decoder\_input\_ids=None, decoder\_attention\_mask=None):

"""

We only finetune the pretrained model on abstractive summarization generation, without MLM task

"""

inputs = {

"input\_ids": x,

"attention\_mask": attention\_mask,

"decoder\_input\_ids": decoder\_input\_ids,

"decoder\_attention\_mask": decoder\_attention\_mask

}

base\_output = self.base\_model(\*\*inputs)

logits = base\_output.logits

return logits

def generate(self, text, tokenizer, device, max\_text\_len=512, max\_len=30, temperature=1.0, sample=False, top\_k=None):

tokens = tokenizer.tokenize(text)

tokens = tokens[:max\_text\_len]

ids = tokenizer.convert\_tokens\_to\_ids(tokens)

input\_ids = torch.from\_numpy(np.array(ids)).long().to(device)

input\_ids = input\_ids.unsqueeze(0)

decoder\_start\_token\_id = self.base\_model.config.decoder\_start\_token\_id

# decoder\_start\_token = tokenizer.convert\_ids\_to\_tokens([decoder\_start\_token\_id])

decoder\_input\_ids = torch.from\_numpy(np.array([decoder\_start\_token\_id])).long().to(device)

decoder\_input\_ids = decoder\_input\_ids.unsqueeze(0)

i = 0

sentence\_ids = []

sentence\_logits = []

while i < max\_len:

with torch.no\_grad():

logits = self.forward(input\_ids, decoder\_input\_ids=decoder\_input\_ids)

sentence\_logits.append(logits)

logits = logits[:, -1, :] / temperature

# optionally crop probabilities to only the top k options

if top\_k is not None:

logits = top\_k\_logits(logits, top\_k)

# apply softmax to convert to probabilities

probs = F.softmax(logits, dim=-1)

# sample from the distribution or take the most likely

if sample:

ix = torch.multinomial(probs, num\_samples=1)

else:

\_, ix = torch.topk(probs, k=1, dim=-1)

# append to the sequence and continue

decoder\_input\_ids = torch.cat((decoder\_input\_ids, ix), dim=1)

sentence\_ids.append(ix.squeeze(0).item())

if sentence\_ids[-1] == self.base\_model.config.eos\_token\_id:

break

i += 1

sentence\_logits = torch.cat(sentence\_logits, dim=1)

gen\_text = tokenizer.decode(sentence\_ids)

return gen\_text, sentence\_logits

-------------------------------------------

class TrainerConfig:

# optimization parameters

max\_epochs = 10

block\_size = 512

batch\_size = 5

learning\_rate = 3e-4

betas = (0.9, 0.95)

grad\_norm\_clip = 1.0

weight\_decay = 0.01 # only applied on matmul weights

# learning rate decay params: linear warmup followed by cosine decay to 10% of original

lr\_decay = False

lr\_decay\_type = "cosine"

warmup\_epochs = 30

final\_epoch = 2700

# checkpoint settings

ckpt\_path = None

num\_workers = 0 # for DataLoader

# tensorboard writer

tensorboard\_writer = None

def \_\_init\_\_(self, \*\*kwargs):

for k,v in kwargs.items():

setattr(self, k, v)

------------------------------------

class Trainer:

def \_\_init\_\_(self, model, tokenizer, train\_dataset, test\_dataset, config):

self.model = model

self.tokenizer = tokenizer

self.train\_dataset = train\_dataset

self.test\_dataset = test\_dataset

self.config = config

# take over whatever gpus are on the system

self.device = 'cpu'

if torch.cuda.is\_available():

self.device = torch.cuda.current\_device()

self.model = self.model.to(self.device)

self.model = torch.nn.DataParallel(self.model).to(self.device)

def save\_checkpoint(self, epoch, best\_result):

# DataParallel wrappers keep raw model object in .module attribute

raw\_model = self.model.module if hasattr(self.model, "module") else self.model

os.makedirs(os.path.dirname(self.config.ckpt\_path), exist\_ok=True)

print("saving %s" % self.config.ckpt\_path)

torch.save({"epoch": epoch, "best\_result": best\_result, "model\_state\_dict": raw\_model.state\_dict()}, self.config.ckpt\_path)

def train(self):

raw\_model = self.model.module if hasattr(self.model, "module") else self.model

self.optimizer = raw\_model.configure\_optimizers(self.config)

best\_result = float('-inf')

test\_result = float('-inf')

for epoch in range(self.config.max\_epochs):

train\_acc = self.run\_epoch('train', epoch)

if self.test\_dataset is not None and epoch % 20 == 0:

test\_result = self.run\_epoch('test', epoch)

# supports early stopping based on the test loss, or just save always if no test set is provided

good\_model = self.test\_dataset is None or test\_result > best\_result

if self.config.ckpt\_path is not None and good\_model:

best\_result = test\_result

self.save\_checkpoint(epoch, best\_result)

def run\_epoch(self, split, epoch):

is\_train = split == 'train'

self.model.train(is\_train)

data = self.train\_dataset if is\_train else self.test\_dataset

loader = DataLoader(data, shuffle=True, pin\_memory=True, batch\_size=self.config.batch\_size, num\_workers=self.config.num\_workers)

losses = []

accs = []

pbar = tqdm(enumerate(loader), total=len(loader)) if is\_train else enumerate(loader)

for it, (text\_ids, attention\_mask, input\_decode\_ids, decode\_attention\_mask, target\_decode\_ids) in pbar:

text\_ids = text\_ids.to(self.device)

attention\_mask = attention\_mask.to(self.device)

input\_decode\_ids = input\_decode\_ids.to(self.device)

decode\_attention\_mask = decode\_attention\_mask.to(self.device)

target\_decode\_ids = target\_decode\_ids.to(self.device)

# forward the model

with torch.set\_grad\_enabled(is\_train):

logits = self.model(text\_ids, attention\_mask, decoder\_input\_ids=input\_decode\_ids, decoder\_attention\_mask=decode\_attention\_mask)

# calculate loss and acc

mask = torch.nonzero(decode\_attention\_mask == 1)

valid\_logits = logits[mask[:, 0], mask[:, 1], :]

valid\_targets = target\_decode\_ids[mask[:, 0], mask[:, 1]]

loss = F.cross\_entropy(valid\_logits.view(-1, valid\_logits.size(-1)), valid\_targets.view(-1))

# acc

cpu\_y = valid\_targets.cpu().numpy()

topk\_scores, topk\_labels = valid\_logits.data.topk(1, 1, True, True)

topk\_ind = topk\_labels.squeeze(1).cpu().numpy()

correct = np.sum(topk\_ind == cpu\_y)

count = len(cpu\_y)

acc = correct / count

losses.append(loss.item())

accs.append(acc)

if is\_train:

# backprop and update the parameters

self.model.zero\_grad()

loss.backward()

torch.nn.utils.clip\_grad\_norm\_(self.model.parameters(), self.config.grad\_norm\_clip)

self.optimizer.step()

# decay the learning rate based on our progress

if self.config.lr\_decay:

# self.tokens += (attention\_mask > 0).sum()

if epoch < self.config.warmup\_epochs:

# linear warmup

lr\_mult = max(epoch / self.config.warmup\_epochs, 1e-2)

else:

if epoch < self.config.final\_epochs:

progress = epoch / self.config.final\_epochs

else:

progress = 1.0

# cosine learning rate decay

if self.config.lr\_decay\_type == "cosine":

lr\_mult = max(0.001, 0.5 \* (1.0 + math.cos(math.pi \* progress)))

# exponential learning rate decay

elif self.config.lr\_decay\_type == "exp":

decay\_progress\_threshold = 1/5

if progress < decay\_progress\_threshold:

lr\_mult = 1

elif decay\_progress\_threshold < progress < decay\_progress\_threshold \* 2:

lr\_mult = 0.1

elif decay\_progress\_threshold \* 2 < progress < decay\_progress\_threshold \* 3:

lr\_mult = 0.01

else:

lr\_mult = 0.001

else:

raise RuntimeError("Unknown learning rate decay type")

lr = self.config.learning\_rate \* lr\_mult

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

param\_group['lr'] = lr

else:

lr = self.config.learning\_rate

# report progress

n\_iter = epoch \* len(loader) + it

self.config.tensorboard\_writer.add\_scalar('Train/loss', loss.item(), n\_iter)

self.config.tensorboard\_writer.add\_scalar('Train/acc', acc, n\_iter)

pbar.set\_description(f"epoch {epoch+1} iter {it}: train loss {loss.item():.5f}, acc {acc:.5f}. lr {lr:e}")

if not is\_train:

test\_loss = float(np.mean(losses))

test\_acc = float(np.mean(accs))

print("test loss: %f, acc %f"%(test\_loss, test\_acc))

self.config.tensorboard\_writer.add\_scalar('Test/loss', test\_loss, epoch)

self.config.tensorboard\_writer.add\_scalar('Test/acc', test\_acc, epoch)

return test\_acc

else:

train\_loss = float(np.mean(losses))

train\_acc = float(np.mean(accs))

return train\_acc

------------------------------------------

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

parser = argparse.ArgumentParser(description='video chapter title generation model')

parser.add\_argument('-f', '--f', default=0)

parser.add\_argument('-a', '--gpu', default=0)

parser.add\_argument('-b', '--epoch', default=3000)

parser.add\_argument('-c', '--batch\_size', default=5)

parser.add\_argument('-d', '--lr\_decay\_type', default="cosine", type=str)

parser.add\_argument('-e', '--model\_type', default="pegasus", type=str)

args = parser.parse\_args()

ckpt\_path = f"/model\_checkpoints/checkpoint/chapter\_title\_gen/chapter\_title\_hugface\_{args.model\_type}\_validation/batch\_{args.batch\_size}\_lr\_decay\_{args.lr\_decay\_type}/checkpoint.pth"

data\_file = "dataset/all\_in\_one\_with\_subtitle.csv"

# train\_vid\_file = "/opt/tiger/video\_chapter\_youtube\_dataset/dataset/train.txt"

# test\_vid\_file = "/opt/tiger/video\_chapter\_youtube\_dataset/dataset/test.txt"

train\_vid\_file = "dataset/new\_train.txt"

test\_vid\_file = "dataset/new\_validation.txt"

tensorboard\_log = os.path.dirname(ckpt\_path)

tensorboard\_writer = SummaryWriter(tensorboard\_log)

set\_random\_seed.use\_fix\_random\_seed()

batch\_size = args.batch\_size

num\_workers = 2

chapter\_title\_text\_len = 30

max\_text\_len = 512

# tokenizer and model

if args.model\_type == "pegasus":

tokenizer = PegasusTokenizer.from\_pretrained('google/pegasus-large')

model = PegasusHugface(reinit\_head=True).to(args.gpu)

# tokenizer = PegasusTokenizer.from\_pretrained('automatic\_video\_chaptering/utilities/train\_utils/local\_pegasus-large\_tokenizer')

# model = PegasusHugface(reinit\_head=True).to('cpu')

else:

raise RuntimeError(f"Unknown model\_type {args.model\_type}")

model = torch.nn.DataParallel(model, device\_ids=[0, 1, 2, 3, 4, 5, 6, 7])

# dataset

train\_dataset = YoutubeChapterTitleDataset(data\_file, train\_vid\_file, tokenizer, max\_text\_len, chapter\_title\_text\_len)

test\_dataset = YoutubeChapterTitleDataset(data\_file, test\_vid\_file, tokenizer, max\_text\_len, chapter\_title\_text\_len)

# initialize a trainer instance and kick off training

tconf = TrainerConfig(max\_epochs=args.epoch, batch\_size=batch\_size, learning\_rate=1e-5, block\_size=max\_text\_len,

lr\_decay\_type=args.lr\_decay\_type, lr\_decay=True, warmup\_epochs=args.epoch//100, final\_epochs=args.epoch//100\*90,

num\_workers=num\_workers, ckpt\_path=ckpt\_path, tensorboard\_writer=tensorboard\_writer)

trainer = Trainer(model, tokenizer, train\_dataset, test\_dataset, tconf)

trainer.device = args.gpu

# trainer.device = 'cpu'

trainer.train()

print('Done!')

--------------------------------------------

"""

Visualize attention map for language model

"""

import torch

import numpy as np

from torch.nn import functional as F

import matplotlib.pyplot as plt

from torch.utils.data import DataLoader

from transformers import BertTokenizer

from model.lang import bert\_hugface

from data.youtube\_dataset import YoutubeClipDataset

from data.infer\_youtube\_video\_dataset import InferYoutubeVideoDataset

from common\_utils import set\_random\_seed

from visualization\_lib.lang.integrated\_gradient import IntegratedGradient

from IPython.display import display, HTML

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

set\_random\_seed.use\_fix\_random\_seed()

device = "cuda:3"

model\_type = "bert"

lr\_decay\_type = "cosine"

ckpt\_path = f"/opt/tiger/video\_chapter\_generation/checkpoint/text/bert/batch\_32\_head\_type\_mlp\_clip\_frame\_num\_16/checkpoint.pth"

data\_file = "/opt/tiger/video\_chapter\_youtube\_dataset/dataset/all\_in\_one\_with\_subtitle.csv"

img\_dir = "/opt/tiger/youtube\_video\_frame\_dataset"

train\_vid\_file = "/opt/tiger/video\_chapter\_youtube\_dataset/dataset/train.txt"

test\_vid\_file = "/opt/tiger/video\_chapter\_youtube\_dataset/dataset/test.txt"

# tokenizer and model

tokenizer = BertTokenizer.from\_pretrained('bert-base-uncased')

model = bert\_hugface.BertHugface(pretrain\_stage=False)

model.build\_chapter\_head()

model = model.to(device)

# dataset

data\_mode = "text" # text (text only), image (image only) or all (multiple-model)

clip\_frame\_num = 16

max\_text\_len = 100

# load checkpoint

checkpoint = torch.load(ckpt\_path)

start\_epoch = checkpoint["epoch"]

best\_result = checkpoint["best\_result"]

model.load\_state\_dict(checkpoint["model\_state\_dict"])

model.eval()

# test on video clip infer dataset

infer\_video\_dataset = InferYoutubeVideoDataset(img\_dir, data\_file, train\_vid\_file, tokenizer, clip\_frame\_num, max\_text\_len=max\_text\_len, mode=data\_mode)

infer\_video\_loader = DataLoader(infer\_video\_dataset, shuffle=False, pin\_memory=True, batch\_size=1, num\_workers=0)

# gradient based visualization

integrated\_grad\_visualization = IntegratedGradient(model, tokenizer, encoder="base\_model")

# run on dataset

# infer\_video\_dataset.random\_choose\_vid()

# OJLk8qNd2O8, Nohke4UXGIM

infer\_video\_dataset.manual\_choose\_vid(vid="OJLk8qNd2O8")

duration = infer\_video\_dataset.get\_duration()

print(f"infer video {infer\_video\_dataset.infer\_vid}, duration {duration}")

t = 0

batch\_i = -1

for img\_clip, text\_ids, attention\_mask, label in infer\_video\_loader:

batch\_i += 1

img\_clip = img\_clip.float().to(device)

text\_ids = text\_ids.to(device)

attention\_mask = attention\_mask.to(device)

label = label.to(device)

instances = integrated\_grad\_visualization.saliency\_interpret((img\_clip, text\_ids, attention\_mask, label))

coloder\_string = integrated\_grad\_visualization.colorize(instances[0])

pred\_label = instances[0]["pred\_label"]

gt\_label = label.item()

# if pred\_label == 1:

if gt\_label == 1:

display(HTML(coloder\_string))

display(HTML("<br>"))

-------------------------------------------

from tqdm import tqdm

import pandas as pd

import matplotlib.pyplot as plt

from datasets import load\_metric

import torch

from transformers import AutoModelForSeq2SeqLM, AutoTokenizer

from transformers import pipeline, set\_seed

import nltk

from nltk.tokenize import sent\_tokenize

nltk.download("punkt")

import warnings

warnings.filterwarnings("ignore")

--------------------------------------

device = "cuda" if torch.cuda.is\_available() else "cpu"

model\_ckpt = "google/pegasus-large"

tokenizer = AutoTokenizer.from\_pretrained(model\_ckpt)

model\_pegasus = AutoModelForSeq2SeqLM.from\_pretrained(model\_ckpt).to(device)

----------------------------------------

# df = pd.read\_csv('/kaggle/input/bolanle-dataset/cache/final\_dataset.csv')

df = pd.read\_csv('/content/drive/MyDrive/final\_dataset.csv')

-----------------------------------------

df.dropna(inplace=True)

=---------------------------------------

ndex(['videoID', 'videoTitle', 'chapterDescription', 'chapterText',

'chapterStartTime', 'chapterEndTime'],

dtype='object')

(12751, 6)

----------------------------------------

from sklearn.model\_selection import train\_test\_split

X\_train, X\_val = train\_test\_split(df, test\_size=0.1, random\_state=42)

X\_train, X\_test = train\_test\_split(X\_train, test\_size=0.1, random\_state=42)

print(f"The shape of the training set is {X\_train.shape}")

print(f"The shape of the validation set is {X\_val.shape}")

print(f"The shape of the test set is {X\_test.shape}")

Model Summary:

>accelerate) (3.1.2)

(12751, 6)

The shape of the training set is (10327, 6)

The shape of the validation set is (1276, 6)

The shape of the test set is (1148, 6)

100%|██████████| 8/8 [00:22<00:00, 2.80s/it]

{'rouge1': AggregateScore(low=Score(precision=0.006249576793527329, recall=0.16666666666666666, fmeasure=0.01203486859224564), mid=Score(precision=0.014893250688705234, recall=0.37083333333333335, fmeasure=0.028493705528188286), high=Score(precision=0.024675472978447573, recall=0.5896874999999999, fmeasure=0.04721842150933824)),

'rouge2': AggregateScore(low=Score(precision=0.0, recall=0.0, fmeasure=0.0), mid=Score(precision=0.004184322033898305, recall=0.13541666666666666, fmeasure=0.008114502799253534), high=Score(precision=0.010434322033898307, recall=0.3229166666666667, fmeasure=0.020211276992801924)),

'rougeL': AggregateScore(low=Score(precision=0.005234156162464987, recall=0.14166666666666666, fmeasure=0.010096351459416234), mid=Score(precision=0.013559752413362037, recall=0.34375, fmeasure=0.026156619891725845), high=Score(precision=0.022625265498761486, recall=0.5645833333333333, fmeasure=0.043210121851642476)),

'rougeLsum': AggregateScore(low=Score(precision=0.005182651573025904, recall=0.13541666666666666, fmeasure=0.009985688264376787), mid=Score(precision=0.013826336319651827, recall=0.34375, fmeasure=0.026548766481365855), high=Score(precision=0.02284881256221497, recall=0.5625, fmeasure=0.04395747462919594))}

rouge1 rouge2 rougeL rougeLsum

precision 0.014893 0.004184 0.013560 0.013826

recall 0.370833 0.135417 0.343750 0.343750

fmeasure 0.028494 0.008115 0.026157 0.026549

[3041/6450 3:46:30 < 4:14:04, 0.22 it/s, Epoch 4.71/10]

Step Training Loss Validation Loss

500 3.108000 3.045014

1000 2.805800 2.835346

1500 2.469500 2.749536

2000 2.508400 2.695641

2500 2.342300 2.651484

3000 2.066000 2.654372

---------------------------------------------------------------------------

KeyboardInterrupt Traceback (most recent call last)

<ipython-input-30-3435b262f1ae> in <cell line: 1>()

----> 1 trainer.train()

/usr/local/lib/python3.10/dist-packages/transformers/trainer.py in train(self, resume\_from\_checkpoint, trial, ignore\_keys\_for\_eval, \*\*kwargs)

1553 hf\_hub\_utils.enable\_progress\_bars()

1554 else:

-> 1555 return inner\_training\_loop(

1556 args=args,

1557 resume\_from\_checkpoint=resume\_from\_checkpoint,

/usr/local/lib/python3.10/dist-packages/transformers/trainer.py in \_inner\_training\_loop(self, batch\_size, args, resume\_from\_checkpoint, trial, ignore\_keys\_for\_eval)

1863 args.logging\_nan\_inf\_filter

1864 and not is\_torch\_tpu\_available()

-> 1865 and (torch.isnan(tr\_loss\_step) or torch.isinf(tr\_loss\_step))

1866 ):

1867 # if loss is nan or inf simply add the average of previous logged losses

KeyboardInterrupt:

100%|██████████| 8/8 [00:13<00:00, 1.67s/it]

rouge1 rouge2 rougeL rougeLsum

precision 0.014893 0.004184 0.013560 0.013826

recall 0.370833 0.135417 0.343750 0.343750

fmeasure 0.028494 0.008115 0.026157 0.026549

rouge1 rouge2 rougeL rougeLsum

precision 0.204167 0.135417 0.200000 0.202083

recall 0.183333 0.135417 0.183333 0.183333

fmeasure 0.190278 0.135417 0.187500 0.187500

('trained-tokenizer/tokenizer\_config.json',

'trained-tokenizer/special\_tokens\_map.json',

'trained-tokenizer/spiece.model',

'trained-tokenizer/added\_tokens.json',

'trained-tokenizer/tokenizer.json')

-------------------------------------------------

def generate\_batch\_sized\_chunks(list\_of\_elements, batch\_size):

"""split the dataset into smaller batches"""

for i in range(0, len(list\_of\_elements), batch\_size):

return list(list\_of\_elements[i : i + batch\_size].values)

-------------------------------------------------

def calculate\_metric\_on\_test\_ds(dataset, metric, model, tokenizer,

batch\_size=16, device=device,

column\_text="article",

column\_summary="highlights"):

article\_batches = generate\_batch\_sized\_chunks(dataset[column\_text], batch\_size)

target\_batches = generate\_batch\_sized\_chunks(dataset[column\_summary], batch\_size)

for article\_batch, target\_batch in tqdm(

zip(article\_batches, target\_batches), total=len(article\_batches)):

inputs = tokenizer(article\_batch, max\_length=1024, truncation=True,

padding="max\_length", return\_tensors="pt")

summaries = model.generate(input\_ids=inputs["input\_ids"].to(device),

attention\_mask=inputs["attention\_mask"].to(device),

length\_penalty=0.8, num\_beams=8, max\_length=128)

''' parameter for length penalty ensures that the model does not generate sequences that are too long. '''

# Finally, we decode the generated texts,

# replace the token, and add the decoded texts with the references to the metric.

decoded\_summaries = [tokenizer.decode(s, skip\_special\_tokens=True,

clean\_up\_tokenization\_spaces=True)

for s in summaries]

decoded\_summaries = [d.replace("", "") for d in decoded\_summaries]

target\_batch = [target\_batch]

metric.add\_batch(predictions=decoded\_summaries, references=target\_batch)

# Finally compute and return the ROUGE scores.

score = metric.compute()

return score

------------------------------------------------

def rouge\_score\_df(score):

rouge\_metrics = ['rouge1', 'rouge2', 'rougeL', 'rougeLsum']

rouge\_output = {

metric: {

'precision': score[metric].mid.precision,

'recall': score[metric].mid.recall,

'fmeasure': score[metric].mid.fmeasure

} for metric in rouge\_metrics

}

df = pd.DataFrame.from\_dict(rouge\_output, orient='index')

df = df.transpose()

return df

-------------------------------------------------

def convert\_examples\_to\_features(dataframe):

input\_encodings = tokenizer(dataframe['chapterText'], max\_length=1024, truncation=True )

with tokenizer.as\_target\_tokenizer():

target\_encodings = tokenizer(dataframe['chapterDescription'], max\_length=128, truncation=True )

return {

'input\_ids' : input\_encodings['input\_ids'],

'attention\_mask': input\_encodings['attention\_mask'],

'labels': target\_encodings['input\_ids']

}

X\_train\_processed = X\_train.apply(convert\_examples\_to\_features, axis=1).reset\_index()[0]

X\_val\_processed = X\_val.apply(convert\_examples\_to\_features, axis=1).reset\_index()[0]

X\_test\_processed = X\_test.apply(convert\_examples\_to\_features, axis=1).reset\_index()[0]

---------------------------------------------------

trainer\_args = TrainingArguments(

output\_dir='pegasus-large', num\_train\_epochs=10, warmup\_steps=500,

per\_device\_train\_batch\_size=1, per\_device\_eval\_batch\_size=1,

weight\_decay=0.01, logging\_steps=10,

evaluation\_strategy='steps', eval\_steps=500, save\_steps=1e6,

gradient\_accumulation\_steps=16

)

trainer = Trainer(model=model\_pegasus, args=trainer\_args,

tokenizer=tokenizer, data\_collator=seq2seq\_data\_collator,

train\_dataset=X\_train\_processed,

eval\_dataset=X\_val\_processed)

------------------------------------------

sample\_text = X\_test["chapterText"].values[1][:1024]

reference = X\_test["chapterDescription"].values[1]

print(f"Sample Text: {sample\_text}")

print(f"Sample Reference: {reference}")

------------------------------------

import zipfile

import os

def zip\_folder(folder\_path, output\_zip):

"""

Compresses the contents of a folder into a zip file.

:param folder\_path: Path to the folder to be zipped

:param output\_zip: Path for the output zip file

"""

with zipfile.ZipFile(output\_zip, 'w', zipfile.ZIP\_DEFLATED) as zipf:

for root, \_, files in os.walk(folder\_path):

for file in files:

file\_path = os.path.join(root, file)

zipf.write(file\_path, os.path.relpath(file\_path, folder\_path))

---------------------------------------

# Provide the folder path and output zip file name

folder\_to\_zip = '/content/pegasus-trained-model'

output\_zip\_file = 'pegasus\_trained\_model.zip'

zip\_folder(folder\_to\_zip, output\_zip\_file)

---------------------------------------

# Provide the folder path and output zip file name

folder\_to\_zip = '/content/trained-tokenizer'

output\_zip\_file = 'pegasus\_trained\_tokenizer.zip'

zip\_folder(folder\_to\_zip, output\_zip\_file)