!pip install transformers

import numpy as np

import pandas as pd

import torch

import time

import torch.nn as nn

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

from sklearn.metrics import classification\_report

import transformers

from transformers import AutoModel, BertTokenizerFast

from torch.utils.data import TensorDataset, DataLoader, RandomSampler, SequentialSampler

# paramters

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

df = pd.read\_csv("newdatasetwithcoviddata.csv")

df.head()

df = df.sample(100000)

df.shape

# check class distribution

df['label'].value\_counts(normalize = True)

# #Split train dataset into train, validation and test sets

train\_text, temp\_text, train\_labels, temp\_labels = train\_test\_split(df['text'], df['label'],

random\_state=2018,

test\_size=0.4,

stratify=df['label'])

# we will use temp\_text and temp\_labels to create validation and test set

val\_text, test\_text, val\_labels, test\_labels = train\_test\_split(temp\_text, temp\_labels,

random\_state=2018,

test\_size=0.5,

stratify=temp\_labels)

test\_text

print(type(test\_text))

print(type(test\_labels))

# import BERT-base pretrained model

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

# Load the BERT tokenizer

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

# sample data

text = ["this is a bert model tutorial", "we will fine-tune a bert model"]

# encode text

sent\_id = tokenizer.batch\_encode\_plus(text, padding=True, return\_token\_type\_ids=False)

# output

print(sent\_id)

# get length of all the messages in the train set

seq\_len = [len(i.split()) for i in train\_text]

pd.Series(seq\_len).hist(bins = 30)

max\_seq\_len = 25

# tokenize and encode sequences in the training set

tokens\_train = tokenizer.batch\_encode\_plus(

train\_text.tolist(),

max\_length = max\_seq\_len,

pad\_to\_max\_length=True,

truncation=True,

return\_token\_type\_ids=False

)

# tokenize and encode sequences in the validation set

tokens\_val = tokenizer.batch\_encode\_plus(

val\_text.tolist(),

max\_length = max\_seq\_len,

pad\_to\_max\_length=True,

truncation=True,

return\_token\_type\_ids=False

)

# tokenize and encode sequences in the test set

tokens\_test = tokenizer.batch\_encode\_plus(

test\_text.tolist(),

max\_length = max\_seq\_len,

pad\_to\_max\_length=True,

truncation=True,

return\_token\_type\_ids=False

)

# for train set

train\_seq = torch.tensor(tokens\_train['input\_ids'])

train\_mask = torch.tensor(tokens\_train['attention\_mask'])

train\_y = torch.tensor(train\_labels.tolist())

# for validation set

val\_seq = torch.tensor(tokens\_val['input\_ids'])

val\_mask = torch.tensor(tokens\_val['attention\_mask'])

val\_y = torch.tensor(val\_labels.tolist())

# for test set

test\_seq = torch.tensor(tokens\_test['input\_ids'])

test\_mask = torch.tensor(tokens\_test['attention\_mask'])

test\_y = torch.tensor(test\_labels.tolist())

#define a batch size

batch\_size = 32

# wrap tensors

train\_data = TensorDataset(train\_seq, train\_mask, train\_y)

# sampler for sampling the data during training

train\_sampler = RandomSampler(train\_data)

# dataLoader for train set

train\_dataloader = DataLoader(train\_data, sampler=train\_sampler, batch\_size=batch\_size)

# wrap tensors

val\_data = TensorDataset(val\_seq, val\_mask, val\_y)

# sampler for sampling the data during training

val\_sampler = SequentialSampler(val\_data)

# dataLoader for validation set

val\_dataloader = DataLoader(val\_data, sampler = val\_sampler, batch\_size=batch\_size)

# freeze all the parameters

for param in bert.parameters():

param.requires\_grad = False

# #Define Model Architecture

class BERT\_Arch(nn.Module):

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

super(BERT\_Arch, self).\_\_init\_\_()

self.bert = bert

# dropout layer

self.dropout = nn.Dropout(0.1)

# relu activation function

self.relu = nn.ReLU()

# dense layer 1

self.fc1 = nn.Linear(768,512)

# dense layer 2 (Output layer)

self.fc2 = nn.Linear(512,2)

#softmax activation function

self.softmax = nn.LogSoftmax(dim=1)

#define the forward pass

def forward(self, sent\_id, mask):

#pass the inputs to the model

\_, cls\_hs = self.bert(sent\_id, attention\_mask=mask, return\_dict=False)

x = self.fc1(cls\_hs)

x = self.relu(x)

x = self.dropout(x)

# output layer

x = self.fc2(x)

# apply softmax activation

x = self.softmax(x)

return x

# pass the pre-trained BERT to our define architecture

model = BERT\_Arch(bert)

# push the model to GPU

model = model.to(device)

# optimizer from hugging face transformers

from transformers import AdamW

# define the optimizer

optimizer = AdamW(model.parameters(), lr = 1e-3)

# #Find Class Weights

from sklearn.utils.class\_weight import compute\_class\_weight

#compute the class weights

class\_wts = compute\_class\_weight('balanced', np.unique(train\_labels), train\_labels)

print(class\_wts)

# convert class weights to tensor

weights= torch.tensor(class\_wts,dtype=torch.float)

weights = weights.to(device)

# loss function

cross\_entropy = nn.NLLLoss(weight=weights)

# number of training epochs

epochs = 5

# function to train the model

def train():

model.train()

total\_loss, total\_accuracy = 0, 0

# empty list to save model predictions

total\_preds=[]

# iterate over batches

for step,batch in enumerate(train\_dataloader):

# progress update after every 50 batches.

if step % 50 == 0 and not step == 0:

print(' Batch {:>5,} of {:>5,}.'.format(step, len(train\_dataloader)))

# push the batch to gpu

batch = [r.to(device) for r in batch]

sent\_id, mask, labels = batch

# clear previously calculated gradients

model.zero\_grad()

# get model predictions for the current batch

preds = model(sent\_id, mask)

# compute the loss between actual and predicted values

loss = cross\_entropy(preds, labels)

# add on to the total loss

total\_loss = total\_loss + loss.item()

# backward pass to calculate the gradients

loss.backward()

# clip the the gradients to 1.0. It helps in preventing the exploding gradient problem

torch.nn.utils.clip\_grad\_norm\_(model.parameters(), 1.0)

# update parameters

optimizer.step()

# model predictions are stored on GPU. So, push it to CPU

preds=preds.detach().cpu().numpy()

# append the model predictions

total\_preds.append(preds)

# compute the training loss of the epoch

avg\_loss = total\_loss / len(train\_dataloader)

# predictions are in the form of (no. of batches, size of batch, no. of classes).

# reshape the predictions in form of (number of samples, no. of classes)

total\_preds = np.concatenate(total\_preds, axis=0)

#returns the loss and predictions

return avg\_loss, total\_preds

# function for evaluating the model

def evaluate():

print("\nEvaluating...")

# deactivate dropout layers

model.eval()

total\_loss, total\_accuracy = 0, 0

# empty list to save the model predictions

total\_preds = []

# iterate over batches

for step,batch in enumerate(val\_dataloader):

# Progress update every 50 batches.

if step % 50 == 0 and not step == 0:

# Calculate elapsed time in minutes.

# elapsed = format\_time(time.time() - t0)

# Report progress.

print(' Batch {:>5,} of {:>5,}.'.format(step, len(val\_dataloader)))

# push the batch to gpu

batch = [t.to(device) for t in batch]

sent\_id, mask, labels = batch

# deactivate autograd

with torch.no\_grad():

# model predictions

preds = model(sent\_id, mask)

# compute the validation loss between actual and predicted values

loss = cross\_entropy(preds,labels)

total\_loss = total\_loss + loss.item()

preds = preds.detach().cpu().numpy()

total\_preds.append(preds)

# compute the validation loss of the epoch

avg\_loss = total\_loss / len(val\_dataloader)

# reshape the predictions in form of (number of samples, no. of classes)

total\_preds = np.concatenate(total\_preds, axis=0)

return avg\_loss, total\_preds

# #Start Model Training

# set initial loss to infinite

best\_valid\_loss = float('inf')

# empty lists to store training and validation loss of each epoch

train\_losses=[]

valid\_losses=[]

#for each epoch

for epoch in range(epochs):

print('\n Epoch {:} / {:}'.format(epoch + 1, epochs))

#train model

train\_loss, \_ = train()

#evaluate model

valid\_loss, \_ = evaluate()

#save the best model

if valid\_loss < best\_valid\_loss:

best\_valid\_loss = valid\_loss

torch.save(model.state\_dict(), 'saved\_weights.pt')

# append training and validation loss

train\_losses.append(train\_loss)

valid\_losses.append(valid\_loss)

print(f'\nTraining Loss: {train\_loss:.3f}')

print(f'Validation Loss: {valid\_loss:.3f}')

import matplotlib.pyplot as plt

plt.plot(range(len(train\_losses)), train\_losses, label='Train')

plt.plot(range(len(valid\_losses)), valid\_losses, label='Valid')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.legend()

plt.show()