#!/usr/bin/env python

# coding=utf-8

# Copyright 2020 The HuggingFace Inc. team. All rights reserved.

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

# http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

# limitations under the License.

""" Finetuning the library models for sequence classification on GLUE."""

# You can also adapt this script on your own text classification task. Pointers for this are left as comments.

import logging

import os

import pickle

import sys

from dataclasses import dataclass, field

from datetime import datetime

from typing import List, Optional

sys.path.append("../hilat")

import datasets

import pandas as pd

from pathlib import Path

import torch

import transformers

from datasets import load\_dataset

from torch.utils.data import Subset

from transformers import (

AutoConfig,

AutoTokenizer,

DataCollatorWithPadding,

EvalPrediction,

HfArgumentParser,

Trainer,

TrainingArguments,

default\_data\_collator,

set\_seed,

)

from transformers.file\_utils import ExplicitEnum

from transformers.trainer\_utils import get\_last\_checkpoint

from transformers.utils import check\_min\_version

from transformers.utils.versions import require\_version

# Will error if the minimal version of Transformers is not installed. Remove at your own risks.

from models.modeling import CodingModel, CodingModelConfig

from models.utils import initial\_code\_title\_vectors, calculate\_scores, tokenize\_dataset, MimicIIIDataset, \

segment\_tokenize\_dataset, LazyMimicIIIDataset

# for tpu cores

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

logger = logging.getLogger("hilat")

check\_min\_version("4.10.0")

require\_version("datasets>=1.8.0", "To fix: pip install -r examples/pytorch/text-classification/requirements.txt")

task\_to\_keys = {

"mimic3-50": ("mimic3-50"),

"mimic3-full": ("mimic3-full"),

}

class TransformerLayerUpdateStrategy(ExplicitEnum):

NO = "no"

LAST = "last"

ALL = "all"

class DocumentPoolingStrategy(ExplicitEnum):

FLAT = "flat"

MAX = "max"

MEAN = "mean"

@dataclass

class DataTrainingArguments:

"""

Arguments pertaining to what data we are going to input our model for training and eval.

Using `HfArgumentParser` we can turn this class

into argparse arguments to be able to specify them on

the command line.

"""

task\_name: Optional[str] = field(

default=None,

metadata={"help": "The name of the task to train on: " + ", ".join(task\_to\_keys.keys())},

)

dataset\_name: Optional[str] = field(

default=None, metadata={"help": "The name of the dataset to use (via the datasets library)."}

)

dataset\_config\_name: Optional[str] = field(

default=None, metadata={"help": "The configuration name of the dataset to use (via the datasets library)."}

)

max\_seq\_length: int = field(

default=128,

metadata={

"help": "The maximum total input sequence length after tokenization. Sequences longer "

"than this will be truncated, sequences shorter will be padded."

},

)

overwrite\_cache: bool = field(

default=False, metadata={"help": "Overwrite the cached preprocessed datasets or not."}

)

pad\_to\_max\_length: bool = field(

default=True,

metadata={

"help": "Whether to pad all samples to `max\_seq\_length`. "

"If False, will pad the samples dynamically when batching to the maximum length in the batch."

},

)

max\_train\_samples: Optional[int] = field(

default=None,

metadata={

"help": "For debugging purposes or quicker training, truncate the number of training examples to this "

"value if set."

},

)

max\_eval\_samples: Optional[int] = field(

default=None,

metadata={

"help": "For debugging purposes or quicker training, truncate the number of evaluation examples to this "

"value if set."

},

)

max\_predict\_samples: Optional[int] = field(

default=None,

metadata={

"help": "For debugging purposes or quicker training, truncate the number of prediction examples to this "

"value if set."

},

)

train\_file: Optional[str] = field(

default=None, metadata={"help": "A csv or a json file containing the training data."}

)

validation\_file: Optional[str] = field(

default=None, metadata={"help": "A csv or a json file containing the validation data."}

)

test\_file: Optional[str] = field(default=None, metadata={"help": "A csv or a json file containing the test data."})

# customized data arguments

label\_dictionary\_file: Optional[str] = field(

default=None, metadata={"help": "The name of the test data file."}

)

code\_max\_seq\_length: int = field(

default=128,

metadata={

"help": "The maximum total input sequence length after tokenization for code long titles"

},

)

code\_batch\_size: int = field(

default=8,

metadata={

"help": "The batch size for generating code representation"

},

)

ignore\_keys\_for\_eval: Optional[List[str]] = field(

default=None, metadata={"help": "The list of keys to be ignored during evaluation process."}

)

use\_cached\_datasets: bool = field(

default=True,

metadata={"help": "if use cached datasets to save preprocessing time. The cached datasets were preprocessed "

"and saved into data folder."})

data\_segmented: bool = field(

default=False,

metadata={"help": "if dataset is segmented or not"})

lazy\_loading: bool = field(

default=False,

metadata={"help": "if dataset is larger than 500MB, please use lazy\_loading"})

def \_\_post\_init\_\_(self):

if self.task\_name is not None:

self.task\_name = self.task\_name.lower()

if self.task\_name not in task\_to\_keys.keys():

raise ValueError("Unknown task, you should pick one in " + ",".join(task\_to\_keys.keys()))

elif self.dataset\_name is not None:

pass

elif self.train\_file is None or self.validation\_file is None:

raise ValueError("Need a training/validation file")

elif self.label\_dictionary\_file is None:

raise ValueError("label dictionary must be provided")

else:

train\_extension = self.train\_file.split(".")[-1]

assert train\_extension in ["csv", "json"], "`train\_file` should be a csv or a json file."

validation\_extension = self.validation\_file.split(".")[-1]

assert (

validation\_extension == train\_extension

), "`validation\_file` should have the same extension (csv or json) as `train\_file`."

@dataclass

class ModelArguments:

"""

Arguments pertaining to which model/config/tokenizer we are going to fine-tune from.

"""

model\_name\_or\_path: str = field(

metadata={"help": "Path to pretrained model or model identifier from huggingface.co/models"}

)

config\_name: Optional[str] = field(

default=None, metadata={"help": "Pretrained config name or path if not the same as model\_name"}

)

tokenizer\_name: Optional[str] = field(

default=None, metadata={"help": "Pretrained tokenizer name or path if not the same as model\_name"}

)

cache\_dir: Optional[str] = field(

default=None,

metadata={"help": "Where do you want to store the pretrained models downloaded from huggingface.co"},

)

use\_fast\_tokenizer: bool = field(

default=True,

metadata={"help": "Whether to use one of the fast tokenizer (backed by the tokenizers library) or not."},

)

model\_revision: str = field(

default="main",

metadata={"help": "The specific model version to use (can be a branch name, tag name or commit id)."},

)

use\_auth\_token: bool = field(

default=False,

metadata={

"help": "Will use the token generated when running `transformers-cli login` (necessary to use this script "

"with private models)."

},

)

# Customized model arguments

d\_model: int = field(default=768, metadata={"help": "hidden size of model. should be the same as base transformer "

"model"})

dropout: float = field(default=0.1, metadata={"help": "Dropout of transformer layer"})

dropout\_att: float = field(default=0.1, metadata={"help": "Dropout of label-wise attention layer"})

num\_chunks\_per\_document: int = field(default=0.1, metadata={"help": "Num of chunks per document"})

transformer\_layer\_update\_strategy: TransformerLayerUpdateStrategy = field(

default="all",

metadata={"help": "Update which transformer layers when training"})

use\_code\_representation: bool = field(

default=True,

metadata={"help": "if use code representation as the "

"initial parameters of code vectors in attention layer"})

multi\_head\_attention: bool = field(

default=True,

metadata={"help": "if use multi head attention for different chunks"})

chunk\_attention: bool = field(

default=True,

metadata={"help": "if use chunk attention for each label"})

multi\_head\_chunk\_attention: bool = field(

default=True,

metadata={"help": "if use multi head chunk attention for each label"})

linear\_init\_mean: float = field(default=0.0, metadata={"help": "mean value for initializing linear layer weights"})

linear\_init\_std: float = field(default=0.03, metadata={"help": "standard deviation value for initializing linear "

"layer weights"})

document\_pooling\_strategy: DocumentPoolingStrategy = field(

default="flat",

metadata={"help": "how to pool document representation after label-wise attention layer for each label"})

def main():

# See all possible arguments in src/transformers/training\_args.py

# or by passing the --help flag to this script.

# We now keep distinct sets of args, for a cleaner separation of concerns.

parser = HfArgumentParser((ModelArguments, DataTrainingArguments, TrainingArguments))

if len(sys.argv) == 2 and sys.argv[1].endswith(".json"):

# If we pass only one argument to the script and it's the path to a json file,

# let's parse it to get our arguments.

model\_args, data\_args, training\_args = parser.parse\_json\_file(json\_file=os.path.abspath(sys.argv[1]))

else:

model\_args, data\_args, training\_args = parser.parse\_args\_into\_dataclasses()

# Setup logging

now = datetime.now().strftime("%Y%m%d%H%M%S")

file\_handler = logging.FileHandler(

os.path.join(training\_args.output\_dir, "log\_{}.txt".format(now)))

logging.basicConfig(

format="%(asctime)s - %(levelname)s - %(filename)s:%(lineno)s] %(message)s",

datefmt="%d/%m/%Y %H:%M:%S",

handlers=[logging.StreamHandler(sys.stdout), file\_handler],

)

log\_level = training\_args.get\_process\_log\_level()

logger.setLevel(log\_level)

datasets.utils.logging.set\_verbosity(log\_level)

transformers.utils.logging.set\_verbosity(log\_level)

transformers.utils.logging.enable\_default\_handler()

transformers.utils.logging.enable\_explicit\_format()

transformers.utils.logging.add\_handler(file\_handler)

# Log on each process the small summary:

logger.warning(

f"Process rank: {training\_args.local\_rank}, device: {training\_args.device}, n\_gpu: {training\_args.n\_gpu}"

+ f"distributed training: {bool(training\_args.local\_rank != -1)}, 16-bits training: {training\_args.fp16}"

)

logger.info(f"DataTraining parameters {data\_args}")

logger.info(f"Model parameters {model\_args}")

logger.info(f"Training/evaluation parameters {training\_args}")

# Detecting last checkpoint.

last\_checkpoint = None

if os.path.isdir(training\_args.output\_dir) and training\_args.do\_train and not training\_args.overwrite\_output\_dir:

last\_checkpoint = get\_last\_checkpoint(training\_args.output\_dir)

if last\_checkpoint is None and len(os.listdir(training\_args.output\_dir)) > 0:

raise ValueError(

f"Output directory ({training\_args.output\_dir}) already exists and is not empty. "

"Use --overwrite\_output\_dir to overcome."

)

elif last\_checkpoint is not None and training\_args.resume\_from\_checkpoint is None:

logger.info(

f"Checkpoint detected, resuming training at {last\_checkpoint}. To avoid this behavior, change "

"the `--output\_dir` or add `--overwrite\_output\_dir` to train from scratch."

)

# Set seed before initializing model.

set\_seed(training\_args.seed)

if data\_args.task\_name is not None and data\_args.task\_name not in task\_to\_keys.keys():

# Downloading and loading a dataset from the hub.

raw\_datasets = load\_dataset("glue", data\_args.task\_name, cache\_dir=model\_args.cache\_dir)

elif data\_args.dataset\_name is not None:

# Downloading and loading a dataset from the hub.

raw\_datasets = load\_dataset(

data\_args.dataset\_name, data\_args.dataset\_config\_name, cache\_dir=model\_args.cache\_dir

)

else:

# Loading a dataset from your local files.

# CSV/JSON training and evaluation files are needed.

data\_files = {"train": data\_args.train\_file, "validation": data\_args.validation\_file,

"label\_dict": data\_args.label\_dictionary\_file}

# Get the test dataset: you can provide your own CSV/JSON test file (see below)

# when you use `do\_predict` without specifying a GLUE benchmark task.

if training\_args.do\_predict:

if data\_args.test\_file is not None:

train\_extension = data\_args.train\_file.split(".")[-1]

test\_extension = data\_args.test\_file.split(".")[-1]

assert (

test\_extension == train\_extension

), "`test\_file` should have the same extension (csv or json) as `train\_file`."

data\_files["test"] = data\_args.test\_file

else:

raise ValueError("Need either a GLUE task or a test file for `do\_predict`.")

for key in data\_files.keys():

logger.info(f"load a local file for {key}: {data\_files[key]}")

label\_dict = pd.read\_csv(data\_args.label\_dictionary\_file)

num\_labels = label\_dict.shape[0]

tokenizer = AutoTokenizer.from\_pretrained(

model\_args.tokenizer\_name if model\_args.tokenizer\_name else model\_args.model\_name\_or\_path,

cache\_dir=model\_args.cache\_dir,

use\_fast=model\_args.use\_fast\_tokenizer,

revision=model\_args.model\_revision,

use\_auth\_token=True if model\_args.use\_auth\_token else None,

padding\_side="right"

)

# Generate code title representation from base transformer model

d\_model = model\_args.d\_model

coding\_model\_config = CodingModelConfig(model\_args.model\_name\_or\_path,

model\_args.tokenizer\_name,

model\_args.transformer\_layer\_update\_strategy,

model\_args.num\_chunks\_per\_document,

data\_args.max\_seq\_length,

model\_args.dropout,

model\_args.dropout\_att,

d\_model,

label\_dict,

num\_labels,

model\_args.use\_code\_representation,

data\_args.code\_max\_seq\_length,

data\_args.code\_batch\_size,

model\_args.multi\_head\_attention,

model\_args.chunk\_attention,

model\_args.linear\_init\_mean,

model\_args.linear\_init\_std,

model\_args.document\_pooling\_strategy,

model\_args.multi\_head\_chunk\_attention)

model = CodingModel(coding\_model\_config, training\_args)

def count\_parameters(model):

return sum(p.numel() for p in model.parameters() if p.requires\_grad)

logger.info("Model parameters: {}".format(count\_parameters(model)))

def load\_data(data\_file):

# Input: data\_file. Output: contains features: input\_ids, attention\_mask, token\_type\_ids

cached\_file = Path(data\_file).parent / Path(data\_file).name\

.replace(".csv", "\_seq-{}.pkl"

.format(data\_args.max\_seq\_length))

if data\_args.use\_cached\_datasets:

# laod dataset from pickle file

if cached\_file.exists():

with open(cached\_file, "rb") as f:

dataset = pickle.load(f)

else:

logger.info("There is no cached data file found for {}".format(data\_file))

data\_args.use\_cached\_datasets = False

if not data\_args.use\_cached\_datasets:

data = pd.read\_csv(data\_file)

if data\_args.data\_segmented:

text = data.loc[:, data.columns.str.startswith("Chunk")].fillna("").apply(

lambda x: [seg for seg in x],

axis=1).tolist()

labels = data.iloc[:, 11:].apply(lambda x: [seg for seg in x], axis=1).tolist()

results = tokenize\_dataset(tokenizer, text, labels, data\_args.max\_seq\_length)

else:

text = data["text"].tolist()

import ast

labels = data["labels"].apply(ast.literal\_eval).tolist()

results = segment\_tokenize\_dataset(tokenizer, text, labels,

data\_args.max\_seq\_length,

model\_args.num\_chunks\_per\_document)

dataset = MimicIIIDataset(results)

with open(cached\_file, 'wb') as f:

pickle.dump(dataset, f)

return dataset

if training\_args.do\_train:

if data\_args.lazy\_loading:

train\_dataset = LazyMimicIIIDataset(data\_args.train\_file, data\_args.task\_name, 'train')

else:

train\_dataset = load\_data(data\_args.train\_file)

if data\_args.max\_train\_samples is not None:

train\_dataset = Subset(train\_dataset, list(range(data\_args.max\_train\_samples)))

if training\_args.do\_eval:

if data\_args.lazy\_loading:

eval\_dataset = LazyMimicIIIDataset(data\_args.validation\_file, data\_args.task\_name, 'dev')

else:

eval\_dataset = load\_data(data\_args.validation\_file)

if data\_args.max\_eval\_samples is not None:

eval\_dataset = Subset(eval\_dataset, list(range(data\_args.max\_eval\_samples)))

if training\_args.do\_predict or data\_args.task\_name is not None or data\_args.test\_file is not None:

if data\_args.lazy\_loading:

predict\_dataset = LazyMimicIIIDataset(data\_args.test\_file, data\_args.task\_name, 'test')

else:

predict\_dataset = load\_data(data\_args.test\_file)

if data\_args.max\_predict\_samples is not None:

predict\_dataset = Subset(predict\_dataset, list(range(data\_args.max\_predict\_samples)))

# You can define your custom compute\_metrics function. It takes an `EvalPrediction` object (a namedtuple with a

# predictions and label\_ids field) and has to return a dictionary string to float.

def compute\_metrics(p: EvalPrediction):

logits = p.predictions[0] if isinstance(p.predictions, tuple) else p.predictions

metric\_scores = calculate\_scores(p.label\_ids, logits)

micro\_scores = calculate\_scores(p.label\_ids, logits, average="micro")

metric\_scores.update(micro\_scores)

return metric\_scores

# Data collator will default to DataCollatorWithPadding, so we change it if we already did the padding.

if data\_args.pad\_to\_max\_length:

data\_collator = default\_data\_collator

elif training\_args.fp16:

data\_collator = DataCollatorWithPadding(tokenizer, pad\_to\_multiple\_of=8)

else:

data\_collator = None

# Initialize our Trainer

trainer = Trainer(

model=model,

args=training\_args,

train\_dataset=train\_dataset if training\_args.do\_train else None,

eval\_dataset=eval\_dataset if training\_args.do\_eval else None,

compute\_metrics=compute\_metrics,

tokenizer=tokenizer,

data\_collator=data\_collator,

)

# Training

if training\_args.do\_train:

checkpoint = None

if training\_args.resume\_from\_checkpoint is not None:

checkpoint = training\_args.resume\_from\_checkpoint

elif last\_checkpoint is not None:

checkpoint = last\_checkpoint

torch.cuda.empty\_cache()

train\_result = trainer.train(resume\_from\_checkpoint=checkpoint,

ignore\_keys\_for\_eval=data\_args.ignore\_keys\_for\_eval)

metrics = train\_result.metrics

max\_train\_samples = (

data\_args.max\_train\_samples if data\_args.max\_train\_samples is not None else len(train\_dataset)

)

metrics["train\_samples"] = min(max\_train\_samples, len(train\_dataset))

trainer.save\_model() # Saves the tokenizer too for easy upload

trainer.log\_metrics("train", metrics)

trainer.save\_metrics("train", metrics)

trainer.save\_state()

# Evaluation

if training\_args.do\_eval:

logger.info("\*\*\* Evaluate \*\*\*")

metrics = trainer.evaluate(eval\_dataset=eval\_dataset, ignore\_keys=data\_args.ignore\_keys\_for\_eval)

max\_eval\_samples = (

data\_args.max\_eval\_samples if data\_args.max\_eval\_samples is not None else len(eval\_dataset)

)

metrics["eval\_samples"] = min(max\_eval\_samples, len(eval\_dataset))

trainer.log\_metrics("eval", metrics)

trainer.save\_metrics("eval", metrics)

if training\_args.do\_predict:

logger.info("\*\*\* Predict \*\*\*")

# output: label\_ids, metrics, predictions(logits, pred\_probs, attention\_weights)

output = trainer.predict(predict\_dataset, metric\_key\_prefix="predict")

output\_predict\_file = os.path.join(training\_args.output\_dir, f"predict\_results\_{data\_args.task\_name}.pkl")

logger.info("Metrics on test datasets: {}".format(output.metrics))

predict\_result = {"labels": output.label\_ids, "metrics": output.metrics, "predictions": output.predictions}

if trainer.is\_world\_process\_zero():

logger.info("Save predictions into {}".format(output\_predict\_file))

with open(output\_predict\_file, "wb") as writer:

pickle.dump(predict\_result, writer, pickle.HIGHEST\_PROTOCOL)

def \_mp\_fn(index):

# For xla\_spawn (TPUs)

main()

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

main()