# HuggingFace Transformers

## 常见的自然语言处理任务:

情感分析、文本生成、命名实体识别、阅读理解、掩码填充、文本摘要、机器翻译、特征提取、对话机器人

## 自然语言处理的新阶段:

神经网络 + 更大的预训练模型 + Prompt

ChatGPT、Bloom、LlaMA、文心一言、通义千问

## Transformers及相关库:

Transformers:核心库，模型加载，模型训练，流水线等;

Tokenizer:分词器，对数据进行预处理，文本到token序列的互相转换;

Datasets:数据集库，提供了数据集加载、处理等方法;

Evaluate:评估函数,提供各种评价指标的计算函数;

PEFT:高效微调模型的库,提供几种高效微调的方法,小参数量撬动大模型;

Accelerate:分布式训练,提供分布式训练解决方案,包括大模型的加载与推理解决方案

Optimum:优化加速库,支持多种后端,如Onnxruntime、OpenVino等

Gradio:可视化部署库,几行代码快速实现基于Web交互的算法演示系统

## Transformers主要模型

自回归：GPT2、Transformer-XL、XLNet

自编码：BERT、ALBERT、RoBERTa、ELECTRA

S2S：BART、Pegasus、T5

## Transformers环境搭建:

pip install jupyterlab scikit-learn pandas matplotlib tensorboard nltk rouge

pip install transformers datasets evaluate peft accelerate gradio optimum sentencepiece

(python==3.9.19、torch==1.13.1+cu116、numpy==1.23.5)

补充:

1.python3.6+torch1.10.2,transformers需要指定版本

(pip install transformers==3.4.0)

2.pip设置清华镜像源

(pip config set global.index-url <https://pypi.tuna.tsinghua.edu.cn/simple>)

3.配置HOSTS文件

185.199.108.133 raw.githubusercontent.com

185.199.109.133 raw.githubusercontent.com

185.199.110.133 raw.githubusercontent.com

185.199.111.133 raw.githubusercontent.com

2606:50c0:8000::154 raw.githubusercontent.com

2606:50c0:8001::154 raw.githubusercontent.com

2606:50c0:8002::154 raw.githubusercontent.com

2606:50c0:8003::154 raw.githubusercontent.com

## Transformers基础组件

1. pipeline:

简介:pipeline指定了数据会经过预处理、模型、后处理三个步骤

I.支持的任务类型：

Eg:

from transformers.pipelines import SUPPORTED\_TASKS

for k, v in SUPPORTED\_TASKS.items():

print(k, v)

for k, v in SUPPORTED\_TASKS.items():

print(k, v["type"])

II.Pipeline创建与使用

1).根据任务类型直接创建pipeline

pipe = pipeline("text-classification"**,**model="uer/roberta-base-finetuned-dianping-chinese"**,** device=0)

print(pipe("我觉得不太行"))

第一个参数:任务 第二个参数:模型 第三个参数:指定device为GPU(0,1,2)/CPU(cpu)

2).提前先加载好模型和分词器,再创建pipeline

model = AutoModelForSequenceClassification.from\_pretrained("uer/roberta-base-finetuned-dianping-chinese")

tokenizer = AutoTokenizer.from\_pretrained("uer/roberta-base-finetuned-dianping-chinese")

pipe = pipeline("text-classification"**,** model=model**,** tokenizer=tokenizer)

print(pipe("我觉得不太行"))

3).预训练模型的例子——零样本检测

Eg:

from PIL import Image

from transformers import \*

pipe = pipeline(task="zero-shot-object-detection", model="google/owlvit-base-patch32")

img = Image.open('demo.jpg')

predictions=pipe(img,candidate\_labels=["hat", "sunglasses"])

print(predictions)

4).pipeline的背后实现

Step 1.初始化Tokenizer

tokenizer = AutoTokenizer.from\_pretrained("uer/roberta-base-finetuned-chinanews-chinese")

Step 2.初始化Model

model = AutoModel.from\_pretrained('uer/roberta-base-finetuned-chinanews-chinese')

Step 3.数据预处理

input\_text = "我觉得不太行!"

inputs = tokenizer(input\_text**,** return\_tensors="pytorch")

使用**PreTrainedTokenizerBase**的\_\_call\_\_()的方法

Step 4.模型预测

res = model(\*\*inputs).logits

Step 5.结果后处理

pred = torch.argmax(torch.softmax(logits**,** dim=-**1**)).item()

result = model.config.id2lbel.get(pred)

Eg:

import torch

from transformers import \*

tokenizer = AutoTokenizer.from\_pretrained('uer/roberta-base-finetuned-chinanews-chinese')

model = AutoModelForSequenceClassification.from\_pretrained('uer/roberta-base-finetuned-chinanews-chinese')

input\_text = "巴黎奥运会"

inputs = tokenizer(input\_text**,** return\_tensors="pt")

print(inputs)

res = model(\*\*inputs)

print(res)

logits = res.logits

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

print(logits)

pred = torch.argmax(logits).item()

print(pred)

print(model.config.id2label[pred])

1. Tokenizer

简介: 数据预处理

Step 1 分词:使用分词器对文本数据进行分词（字、字词）;

Step 2 构建词典:根据数据集分词的结果，构建词典映射（不绝对,如果采用预训练词向量,词典映射要根据词向量文件进行处理）;

Step 3 数据转换:根据构建好的词典,将分词处理后的数据做映射,将文本序列转换为数字序列;

Step 4 数据填充与截断:在以batch输入到模型的方式中,需要对过短的数据进行填充,过长的数据进行截断,保证数据长度符合模型能接收的范围,同时batch内的数据维度大小一致。

Tip:不同的模型可能对应不同的Tokenizer,为了解决该问题,提出了AutoTokenizer。

1. 分词器的加载与保存

Eg:

import torch

from transformers import \*

tokenizer = AutoTokenizer.from\_pretrained('uer/roberta-base-finetuned-dianping-chinese') ## BertTokenizerFast（**PreTrainedTokenizerFast**）

print(tokenizer)

tokenizer.save\_pretrained("./roberta\_tokenizer")

tokenizer = AutoTokenizer.from\_pretrained("./roberta\_tokenizer")

print(tokenizer)

1. 句子分词

Eg:

tokenizer = AutoTokenizer.from\_pretrained('uer/roberta-base-finetuned-dianping-chinese')

tokens = tokenizer.tokenize("我可太想进步了")

print(tokens)

1. 查看词典

Eg:

tokenizer = AutoTokenizer.from\_pretrained('uer/roberta-base-finetuned-dianping-chinese')

vocab = tokenizer.vocab

print(vocab)

1. 索引转换

Eg:

from transformers import \*

tokenizer = AutoTokenizer.from\_pretrained('uer/roberta-base-finetuned-chinanews-chinese')

print(type(tokenizer))

input\_text="篮球大于生活"

tokens = tokenizer.tokenize(input\_text)

print(tokens)

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

print(ids)

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

print(tokens)

strs = tokenizer.convert\_tokens\_to\_string(tokens)

print(strs)

1. 更便捷的实现方法

Eg:

from transformers import \*

tokenizer = AutoTokenizer.from\_pretrained('uer/roberta-base-finetuned-chinanews-chinese')

print(type(tokenizer))

input\_text="篮球大于生活"

# 将字符串转换为id序列,又称之为编码

ids = tokenizer.encode(input\_text**,** add\_special\_tokens=False)

print(ids)

# 将id序列转换为字符串,又称之为解码

strs = tokenizer.decode(ids)

print(strs)

1. 填充与截断

Eg1:填充

input\_text = "篮球大于生活"

# 将字符串转换为id序列,又称之为编码

ids = tokenizer.encode(input\_text**,** padding='max\_length'**,** max\_length=**15,** add\_special\_tokens=True)

print(ids)

# 将id序列转换为字符串,又称之为解码

strs = tokenizer.decode(ids**,** skip\_special\_tokens=True)

print(strs)

Eg2:截断

input\_text = '篮球大于生活篮球大于生活篮球大于生活'

# 将字符串转换为id序列,又称之为编码

ids = tokenizer.encode(input\_text**,** max\_length=**15,** truncation=True**,** add\_special\_tokens=True)

print(ids)

# 将id序列转换为字符串,又称之为解码

strs = tokenizer.decode(ids**,** skip\_special\_tokens=True)

print(strs)

1. 快速调用方式(会自动实现token\_type\_ids和attention\_mask)

Eg1:

ids = tokenizer.encode(input\_text)

print(ids)

token\_type\_ids = [0] \* len(ids)

print(token\_type\_ids)

attention\_mask = [1 if idx != 0 else 0 for idx in ids]

print(attention\_mask)

Eg2:

inputs = tokenizer.encode\_plus(input\_text**,** padding='max\_length'**,** max\_length=**15**)

print(inputs)

Eg3:（Eg3等价于Eg2）

inputs = tokenizer(input\_text**,** padding='max\_length'**,** max\_length=**15**)

print(inputs)

1. 处理batch数据

Eg:

tokenizer = AutoTokenizer.from\_pretrained('uer/roberta-base-finetuned-dianping-chinese')

input\_text = ["弱小的我也有大梦想"**,** "有梦想谁都了不起"**,** "追逐梦想的心,比梦想本身更可贵"]

res = tokenizer(input\_text)

print(res)

1. 基础组件之Tokenizer

Fast / Slow Tokenizer

* FastTokenizer

基于Rust实现,速度快

offsets\_mapping,word\_ids

* SlowTokenizer

基于Pytorch实现,速度慢

1. 特殊Tokenizer的加载

HuggingFace上的很多模型并不是官方实现的，可以将一些自己写的模型，包括一些文件传上去（chatglm平面级的对话模型,分词器是自己实现的,其实是在它的远程仓库里面,必须指定trust\_remote\_code=True）

1. Model:

简介:

**Transformer**

* 原始的Transformer为编码器(Encoder)、解码器（Decoder）模型
* Encoder部分接收输入并构建其完整特征表示，Decoder部分使用Encoder的编码结果以及其他的输入生成目标序列
* 无论是编码器还是解码器，均由多个TransformerBlock堆叠而成
* TransformerBlock由注意力机制(Attention)和前馈神经网络(FFN)组成

注意力机制

注意力机制的使用是Transformer的一个核心特性，在计算当前词的特征表示时，可以通过注意力机制有选择性地告诉模型要使用哪些上下文

**模型类型**

编码器模型:自编码模型，使用Encoder，拥有双向的注意力机制，即计算每一个词的特征时都看到完整上下文;（ALBERT、BERT、DistilBERT、RoBERTa——文本分类、命名实体识别、阅读理解）

解码器模型:自回归模型，使用Decoder，拥有单向的注意力机制，即计算每一个词的特征时都只能看到上文，无法看到下文;（GPT、GPT-2、Bloom、LLaMA——文本生成）

编码器解码器模型:序列到序列模型，使用Encoder+Decoder，Encoder部分使用双向的注意力，Decoder部分使用单向注意力;（BART、T5、mBART、GLM——文本摘要、机器翻译）

**Model Head (Transformers模型中不存在任务头)**

* Model Head是连接在模型后的层，通常为1个或多个全连接层;
* Model Head将模型的编码的表示结果进行映射，以解决不同类型的任务;

(输入数据到Transformer Network网络中，最终输出的数据是HidenState状态（比如输入的序列长度是128，编码后的维度是768，最终输出的维度是1\*128\*768)，Transformers中的Model Head（\*Model（模型本身，只返回编码结果）、\*ForCausalLM、\*ForMaskedLM、\*ForSeq2SeqLM、\*ForMultipleChoice、\*ForQuestionAnswering、\*ForSequenceClassification、\*ForTokenClassification）

1. 模型的加载

I.远程下载

model = AutoModel.from\_pretrained("hfl/rbt3"**,** force\_download=True)

II.本地下载（使用网页、git将模型文件下载）

git clone [git@hf.co:hfl/rbt3](mailto:git@hf.co:hfl/rbt3)

git lfs clone [git@hf.co:hfl/rbt3](mailto:git@hf.co:hfl/rbt3) --include=“\*.bin”

1. 加载模型

model = AutoModel.from\_pretrained("hlfrbt3")

1. 加载模型参数（可更改的参数）

config = AutoConfig.from\_pretrained("hlfrbt3")

1. 模型调用
2. Transformer简单模型（使用AutoModel来进行加载,该模型不存在任务头Model Head）

Eg:

model = AutoModel.from\_pretrained("hlfrbt3")

output = model(\*\*inputs)

print(output)

**输出向量的格式为:[batch,seq\_len,embedding\_size]**

last\_hidden\_state:是模型最后一层输出的隐藏状态，序列标注中通常使用;

pooler\_output:是将Bert模型中的[CLS]Token的向量取出来;

1. Transformer 分类模型（使用AutoModelForSequenceClassification来进行加载）

Eg:

model = AutoModelForSequenceClassification.from\_pretrained("hlfrbt3")

output = model(\*\*inputs)

print(output)

Print:

SequenceClassifierOutput(loss=None, logits=tensor([[ 0.2809, -0.2431]], grad\_fn=<AddmmBackward0>), hidden\_states=None, attentions=None)

1. 模型的加载与保存
2. 模型的加载

Eg:

model = AutoModel.from\_pretrained('rbt3')

1. 模型的保存

Eg:

model.save\_pretrained('./model')

**Transformer实战:**

* **Bert文本分类案例**

Eg:

from transformers import AutoModelForSequenceClassification, AutoTokenizer, pipeline  
import pandas as pd  
from torch.utils.data import Dataset, DataLoader, random\_split  
import torch  
from torch.optim import Adam  
  
  
# 1.加载数据集  
# data = pd.read\_csv('ChnSentiCorp\_htl\_all.csv')  
# data.dropna()  
# print(data.head())  
  
# 2.创建DataSet  
class ClassificationDataset(Dataset):  
 def \_\_init\_\_(self):  
 super(ClassificationDataset, self).\_\_init\_\_()  
 self.data = pd.read\_csv('ChnSentiCorp\_htl\_all.csv')  
 self.data = self.data.dropna()  
  
 def \_\_len\_\_(self):  
 return len(self.data)  
  
 def \_\_getitem\_\_(self, idx):  
 return self.data.iloc[idx]['review'], self.data.iloc[idx]['label']  
  
  
datasets = ClassificationDataset()  
train\_dataset, valid\_dataset = random\_split(datasets, lengths=[0.9, 0.1])  
  
# 3.创建DataLoader  
tokenizer = AutoTokenizer.from\_pretrained('rbt3')  
  
  
def collate\_fn(batch):  
 review, labels = [], []  
 for sample in batch:  
 review.append(sample[0])  
 labels.append(sample[1])  
 inputs = tokenizer(review, padding='max\_length', max\_length=128, truncation=True, return\_tensors='pt')  
 inputs['labels'] = torch.tensor(labels)  
 return inputs  
  
  
train\_loader = DataLoader(dataset=train\_dataset, batch\_size=128, collate\_fn=collate\_fn)  
valid\_loader = DataLoader(dataset=valid\_dataset, batch\_size=128, collate\_fn=collate\_fn)  
  
# 4.创建模型和优化器  
device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')  
model = AutoModelForSequenceClassification.from\_pretrained('rbt3').to(device)  
optimizer = Adam(model.parameters(), lr=2e-5)  
  
  
# 5.训练和验证  
def train(epoch=5, log\_step=5):  
 total\_train = 0  
 for epo in range(epoch):  
 model.train()  
 for batch in train\_loader:  
 batch = {k: v.to(device) for k, v in batch.items()}  
 optimizer.zero\_grad()  
 outputs = model(\*\*batch)  
 outputs.loss.backward()  
 optimizer.step()  
 if total\_train % log\_step == 0:  
 print(f'Epoch {epo}, Total\_Train: {total\_train}, Loss: {outputs.loss.item()}')  
 total\_train += 1  
 acc = eval()  
 print(f'Epoch {epoch}, Total\_Train: {total\_train}, Acc: {acc}')  
 model.save\_pretrained('./model')  
  
  
def eval():  
 model.eval()  
 with torch.inference\_mode():  
 acc\_num = 0  
 for batch in valid\_loader:  
 batch = {k: v.to(device) for k, v in batch.items()}  
 outputs = model(\*\*batch)  
 preds = torch.argmax(outputs.logits, dim=-1)  
 acc\_num += (preds == batch['labels']).sum().item()  
 return acc\_num \* 1.0 / len(valid\_dataset)   
 #不是valid\_loader,len(valid\_dataset)与len(valid\_ loader)长度不一致  
  
train()  
  
# 6.模型预测  
sens = "这家店面环境还不错,饭也好吃"  
tokenizer = AutoTokenizer.from\_pretrained('rbt3')  
model = AutoModelForSequenceClassification.from\_pretrained("./model").to(device)  
model.config.id2label = {0: '差评', 1: '好评'}  
pipe = pipeline(task='text-classification', tokenizer=tokenizer, model=model)  
print(pipe(sens))

Eg:

1. pandas中loc函数、iloc函数:

dataset = pd.read\_csv('data/ChnSentiCorp\_htl\_all.csv')  
dataset.dropna()

loc函数:是基于行标签和列标签进行索引的

eg:

print(dataset.loc[5:8, 'label':'review'])

label review

5 1 总的来说，这样的酒店配这样的价格还算可以，希望他赶快装修，给我的客人留些好的印象

6 1 价格比比较不错的酒店。这次免费升级了，感谢前台服务员。房子还好，地毯是新的，比上次的好些。早...

7 1 不错，在同等档次酒店中应该是值得推荐的！

iloc函数:是基于行和列的位置进行索引的,索引值从0开始,并且不包括最后一个位置

eg:

print(dataset.iloc[5:8])

label review

5 1 总的来说，这样的酒店配这样的价格还算可以，希望他赶快装修，给我的客人留些好的印象

6 1 价格比比较不错的酒店。这次免费升级了，感谢前台服务员。房子还好，地毯是新的，比上次的好些。早...

7 1 不错，在同等档次酒店中应该是值得推荐的！

8 1 入住丽晶，感觉很好。因为是新酒店，的确有淡淡的油漆味，房间内较新。房间大小合适，卫生间设备齐...

1. Datasets:

简介:datasets库是一个非常简单易用的数据集加载库,可以方便快捷的从本地或者Hugging Face Hub加载数据集

公开数据集地址:https://huggingface.co/datasets

文档地址:https://huggingface.co/docs/datasets/index

A. 加载在线数据集

Eg:

from datasets import \*  
dataset = load\_dataset('madao33/new-title-chinese')  
print(dataset)

参数指定:指定split参数,返回的是Dataset类型，否则返回的是DatesetDict类型

split：train、validation、test、train[:100]、train[:50%]、[“train[:50%]”,”validation[:10%]“]

Eg:

from datasets import \*  
dataset = load\_dataset('data/madao', split="train[10:100]")  
print(dataset)

1. 离线加载数据集
2. data文件夹中含.py文件

Eg:

from datasets import \*  
dataset = load\_dataset('zero-shot/data/super\_glue.py', 'axg', trust\_remote\_code=True)  
print(dataset)

1. data文件夹中含dataset\_info.json、xxx.arrow

Eg1:

from datasets import \*  
dataset = load\_dataset('data')  
print(dataset)

Eg2:

from datasets import \*

dataset = load\_dataset('data/madao', split=["train[:50%]","train[50%:]"])  
print(dataset)

1. 查看数据集
2. load\_dataset函数不含split参数

dataset = load\_dataset('data/madao')  
print(dataset['train'][0])  
print(dataset['train'][:2])

1. load\_dataset函数含split参数

dataset = load\_dataset('data/madao',split=['train[:50%]','train[50%:]'])  
print(dataset[0][0])  
print(dataset[0][:2])

1. 查看数据中包含哪些字段

dataset = load\_dataset('data/madao',split=['train[:50%]','train[50%:]'])  
print(dataset[0].column\_names)  
print(dataset[0].features)

1. 数据集划分

分类数据集按比例划分（boolq\_dataset、dataset都是DatasetDict类型）

boolq\_dataset = load\_dataset('aps/super\_glue','boolq')  
print(boolq\_dataset)  
dataset=boolq\_dataset['train'].train\_test\_split(train\_size=0.8, stratify\_by\_column='label')  
print(dataset)

1. 数据的选取、过滤、映射

数据的过滤或映射,可以是针对Dataset,也可以是针对DatasetDict

1. 选取

select\_dataset = dataset['train'].select([0, 1])

1. 过滤

filter\_dataset = dataset['train'].filter(lambda example: '中国' in example['title'])

1. 映射(map会导致一组batch多增加一个field字段)

I.基础数据映射处理

Eg1:

def add\_prefix(examples):  
 examples['title'] = 'Prefix:' + examples['title']  
 return examples  
  
reflect\_dataset = dataset.map(add\_prefix)['train'][:10]['title']  
print(reflect\_dataset)

II.数据映射处理加速(batched=True、num\_proc=4)

dataset = load\_dataset('data/madao')  
tokenizer = AutoTokenizer.from\_pretrained("bert-base-chinese")

Eg2:[!!!important]

def preprocess\_function(examples):  
 model\_inputs = tokenizer(examples['content'], truncation=True, padding="max\_length", max\_length=512)  
 labels = tokenizer(examples['title'], truncation=True, padding="max\_length", max\_length=32)  
 model\_inputs['label'] = labels['input\_ids']  
 return model\_inputs

reflect\_dataset = dataset['train'].map(preprocess\_function, batched=True)  
# batched=True速度处理会非常快

print(reflect\_dataset)

Eg3:[!!!important]

def preprocess\_function(examples, tokenizer=tokenizer):  
 model\_inputs = tokenizer(examples['content'], truncation=True, padding="max\_length", max\_length=512)  
 labels = tokenizer(examples['title'], truncation=True, padding="max\_length", max\_length=32)  
 model\_inputs['label'] = labels['input\_ids']  
 return model\_inputs  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 reflect\_dataset = dataset['train'].map(preprocess\_function, num\_proc=4)  
 print(reflect\_dataset)

III.数据映射处理[去除原始字段]

reflect\_dataset = dataset['train'].map(preprocess\_function, num\_proc=4,  
 remove\_columns=dataset['train'].column\_names)

1. 数据的保存与加载

1).arrow格式的数据集

I.数据保存

reflect\_dataset.save\_to\_disk('data/madao\_pretrained')

II.数据加载

reflect\_dataset = datasets.load\_from\_disk('data/madao\_pretrained')

2).json格式的数据集

I.数据保存

dataset = load\_dataset('data/madao', split="train")

dataset.to\_json('data/madao.json')

II.数据加载

dataset = Dataset.from\_json('data/madao.json')

3).csv格式的数据集

I.数据保存

dataset = load\_dataset('data/madao')  
dataset['train'].to\_csv("data/madao.csv")

II.数据加载

dataset = Dataset.from\_csv('data/madao.csv')

1. 加载本地数据集

I. 指定数据集所在文件夹data\_dir

dataset = load\_dataset('csv', data\_dir='./classification/data', split='train')  
print(dataset)

II.指定数据集文件名data\_file(单文件、多文件(列表))

dataset = load\_dataset('csv', data\_files='./classification/data/ChnSentiCorp\_htl\_all.csv', split='train')  
print(dataset)

III.通过预先加载的其他格式转换加载数据集

data = pd.read\_csv('classification/data/ChnSentiCorp\_htl\_all.csv')  
dataset = Dataset.from\_pandas(data)  
print(dataset)

IV.通过集合形式转换加载数据集(list(dict))

list=[{'title': '体育', 'content': '篮球大于生活'}, {'title': '饮食', 'content': '红烧肉可真不错'}]  
dataset = Dataset.from\_list(list)

V.通过自定义加载脚本加载数据集

problem:

dataset = load\_dataset('json', data\_files='data/cmrc/cmrc2018\_trial.json', field="data")加载数据集时,会导致数据加载不完整,并产生field嵌套

solve:

class CMRC2018TRIAL(datasets.GeneratorBasedBuilder):  
  
 def \_info(self):

# info方法,定义数据集的信息,这里要对数据的字段进行定义 return DatasetInfo(description,features)  
 pass  
  
 def \_split\_generators(self, dl\_manager):

# split\_generators方法,定义数据集的划分,  
# 涉及两个参数:name和gen\_kwargs name:指定数据集的划分 gen\_kwargs:指定要读取的文件的路径(filepath),与\_generate\_examples的传入参数(filepath)一致  
# param dl\_manager return [ datasets.SplitGenerator ]

pass  
  
 def \_generate\_examples(self, \*\*kwargs):

# generate\_examples方法,生成具体的样本,使用yield (key, example) tuples from the dataset  
# 需要额外指定key,id从0开始自增就可以  
# param :gen\_kwargs  
 pass

load\_script.py

Eg:

import json  
import datasets  
from datasets import DownloadManager, DatasetInfo  
  
  
class CMRC2018TRIAL(datasets.GeneratorBasedBuilder):  
  
 def \_info(self):  
 # info方法,定义数据集的信息,这里要对数据的字段进行定义 return DatasetInfo(description,features)  
 return DatasetInfo(  
 description="CMRC 2018 Trial Dataset",  
 features=datasets.Features({  
 "id", datasets.Value("string"),  
 "context", datasets.Value("string"),  
 "question", datasets.Value("string"),  
 "answer", datasets.Sequence(  
 {"answer\_starts": datasets.Value("int32"), "answer\_texts": datasets.Value("string")}  
 )  
 })  
)

def \_split\_generators(self, dl\_manager: DownloadManager):  
 # split\_generators方法,定义数据集的划分,  
 # 涉及两个参数:name和gen\_kwargs name:指定数据集的划分 gen\_kwargs:指定要读取的文件的路径(filepath),与\_generate\_examples的传入参数(filepath)一致  
 # param dl\_manager return [ datasets.SplitGenerator ]  
 return [  
 datasets.SplitGenerator(name=datasets.Split.TRAIN,  
 gen\_kwargs={"filepath": "./data/cmrc/cmrc2018\_train.json"})]  
  
 def \_generate\_examples(self, filepath):  
 # generate\_examples方法,生成具体的样本,使用yield (key, example) tuples from the dataset  
 # 需要额外指定key,id从0开始自增就可以  
 # param :gen\_kwargs  
  
 with open(filepath, encoding="utf-8") as f:  
 data = json.load(f)  
 for example in data["data"]:  
 for paragraph in example["paragraphs"]:  
 context = paragraph["context"].strip()  
 for qa in paragraph["qas"]:  
 question = qa["question"].strip()  
 id\_ = qa["id"]  
 answer\_starts = [answer['answer\_start'] for answer in qa["answers"]]  
 answer\_texts = [answer['text'] for answer in qa["answers"]]  
 yield id\_, {  
 "context": context,  
 "question": question,  
 "id": id\_,  
 "answer": {  
 "answer\_starts": answer\_starts,  
 "texts": answer\_texts,  
 }  
 }

main.py

dataset = load\_dataset('load\_script.py', split='train', trust\_remote\_code=True)  
print(dataset)

补充:

1.参数传递

Eg:

def fun1(name, age):  
 print(f"{name}今年{age}岁了")  
  
fun1(\*\*{'name': '林泽源', 'age': 23})  
  
  
def fun2(\*\*kwargs):  
 name, age = '', ''  
 for key, value in kwargs.items():  
 if (key == 'name'):  
 name = value  
 else:  
 age = value  
 print(f"{name}今年{age}岁了")  
  
  
fun2(name="林泽远", age=23)

2.函数(具有关键字yield)初始化时,不会执行,而是得到生成器generator

Eg:

def fun():  
 yield 'a'  
 yield '1'  
 yield '$'  
 b = yield '%'  
 print(b)  
  
  
b = fun()  
print(next(b)) # a  
print(next(b)) # 1  
print(next(b)) # $  
print(next(b)) # %  
print(next(b)) # None

3.Dataset的get\_item参数返回值是一个元组。

4.DataLoader的collate\_fn参数说明:

collate\_fn:将从dataset的\_\_getitem\_\_函数中获取的一组数据batch,通过collate\_fn对这组数据进行预处理(补齐、删减)操作,并最终将数据重新堆叠成一组数据batch。

5.\_\_str\_\_是一个特殊方法,用于定义对象的可读字符串表示:

Eg:

class Person(object):  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
  
 def \_\_str\_\_(self):  
 return str(self.\_\_dict\_\_)  
  
  
person = Person('John', 23)  
print(person)

1. Dataset with DataCollator

DataCollator只会对原始字段input\_ids、token\_type\_ids、attention\_mask、labels进行填充padding操做。如果有自定义字段，则需要自定义进行填充padding；否则需要指定remove\_columns=dataset.column\_names。

DataCollatorWithPadding:会对一组数据进行动态填充且返回tensor格式数据类型，前提是tokenizer没有指定填充padding的大小。

Eg:

from datasets import \*  
from torch.utils.data import DataLoader  
from transformers import AutoTokenizer, DataCollatorWithPadding  
  
dataset = load\_dataset('csv', data\_files='classification/data/ChnSentiCorp\_htl\_all.csv', split='train')  
dataset = dataset.filter(lambda x: x['review'] is not None)  
print(dataset)  
  
tokenizer = AutoTokenizer.from\_pretrained('classification/rbt3\_tokenizer')  
  
  
def process\_function(example, tokenizer=tokenizer):  
 tokenized\_input = tokenizer(example['review'], truncation=True, max\_length=128)  
 tokenized\_input['labels'] = example['label']  
 return tokenized\_input  
  
  
tokenized\_dataset = dataset.map(process\_function, batched=True, remove\_columns=dataset.column\_names)  
print(tokenized\_dataset)  
  
collator = DataCollatorWithPadding(tokenizer=tokenizer)  
dl = DataLoader(dataset=tokenized\_dataset, batch\_size=4, collate\_fn=collator, shuffle=True)  
  
for batch in dl:  
 print(batch['input\_ids'].size())

1. Evaluate:

简介:evaluate库是一个非常简单易用的机器学习模型评估函数库,只需要一行代码便可以加载各种任务的评估函数

函数库地址: https://huggingface.co/evaluate-metric

文档地址: <https://huggingface.co/docs/evaluate/index>

1. 查看支持的评估函数

Eg:

import evaluate

evaluate.list\_evaluation\_modules(include\_community=False, with\_details=True)

1. 加载评估函数

Eg:

import evaluate  
  
accuracy = evaluate.load("accuracy")  
print(accuracy)  
print("===========================================")  
print(accuracy.description)  
print("===========================================")  
print(accuracy.inputs\_description)

1. 评估指标计算
2. 全局计算

Eg:

import evaluate  
  
acc\_metric = evaluate.load("accuracy")  
  
ref = [1, 2, 3, 4, 5]  
pred = [1, 5, 2, 4, 5]  
  
acc1 = acc\_metric.compute(references=ref, predictions=pred)  
print(acc1)  
acc2 = acc\_metric.compute(references=ref, predictions=pred, normalize=False)  
print(acc2)  
acc3 = acc\_metric.compute(references=ref, predictions=pred, sample\_weight=[1, 1, 1, 1, 1])  
print(acc3)

II. 迭代计算

1).add (既可以针对element,又可以针对batch)

Eg:

import evaluate  
  
acc\_metrix = evaluate.load('accuracy')  
  
for refs, preds in zip([0, 1, 1, 0], [1, 0, 1, 0]):  
 acc\_metrix.add(references=refs, predictions=preds)  
  
print(acc\_metrix.compute())

2).add\_batch（只可以针对batch）

Eg:

import evaluate  
  
acc\_metric = evaluate.load('accuracy')  
  
for refs, preds in zip([[0, 1], [1]], [[1, 0], [1]]):  
 acc\_metric.add\_batch(references=refs, predictions=preds)  
  
print(acc\_metric.compute())

补充:

1. batch与zip类型相似 — zip返回元组的集合 batch.item()返回元组的集合

{'input\_ids': tensor([

[ 101, 6370, 4638, ..., 0, 0, 0],

[ 101, 2990, 3193, ..., 4638, 1377, 102],

[ 101, 3680, 3613, ..., 0, 0, 0],

...,

[ 101, 3683, 6772, ..., 0, 0, 0],

[ 101, 2595, 817, ..., 0, 0, 0],

[ 101, 6821, 2157, ..., 1765, 7188, 102]]),

'labels': tensor([0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0,

1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1,

0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1])}

1. 多个评估指标计算

Eg:

import evaluate  
  
eval\_metric = evaluate.combine(['accuracy', 'f1', 'precision', 'recall'])  
  
for refs, preds in zip([[0, 1], [1]], [[1, 0], [1]]):  
 eval\_metric.add\_batch(references=refs, predictions=preds)  
  
print(eval\_metric.compute())

补充:

1. 多分类任务,指标f1\_score没法直接调用,需要指定参数average

1. 评估结果对比可视化（matpylib、seaborn）

from evaluate.visualization import radar\_plot  
  
data = [  
 {"accuracy": 0.99, "f1": 0.95, "recall": 0.95, "precision": 0.95},  
 {"accuracy": 0.49, "f1": 0.35, "recall": 0.65, "precision": 0.75},  
 {"accuracy": 0.69, "f1": 0.75, "recall": 0.85, "precision": 0.95},  
 {"accuracy": 0.89, "f1": 0.85, "recall": 0.95, "precision": 0.98}  
]  
model\_names = ["Model 1", "Model 2", "Model 3", "Model 4"]  
plot = radar\_plot(data=data, model\_names=model\_names)  
plot.show()

补充:

1.zip()函数的使用:

Eg:

x = ['a', 'b', 'c', 'd', 'e']  
y = [1, 2, 3, 4, 5]  
xy = zip(x, y)  
for item in xy:  
 print(item)

2.f1 score评估指标

1. Trainer:

简介:Trainer是transformers库中提供的训练的函数,内部封装了完整的训练、评估逻辑，并集成了多种的后端，如DeepSpeed、Pytorch FSDP等，搭配TrainingArguments对训练过程中的各项参数进行配置,可以非常方便快捷地启动模型单机/分布式训练

[!important]需要注意的是:

1.使用Trainer进行模型训练对模型的输入输出是有限制的,要求模型返回元组或者ModelOutput的子类;

2.如果输入中提供了labels,模型要能返回loss结果;如果是元组,要求loss为元组中第一个值。

Eg:

train\_args = TrainingArguments(  
 output\_dir='./checkpoints',  
 per\_device\_train\_batch\_size=64,  
 per\_device\_eval\_batch\_size=128,  
 logging\_steps=10,  
 eval\_strategy='epoch',  
 save\_strategy='epoch',  
 learning\_rate=2e-5,  
 weight\_decay=0.01,  
 metric\_for\_best\_model='f1',  
 load\_best\_model\_at\_end=True  
)  
trainer = Trainer(model=model, args=train\_args, train\_dataset=tokenized\_dataset['train'],  
 eval\_dataset=tokenized\_dataset['test'], data\_collator=DataCollatorWithPadding(tokenizer=tokenizer),  
 compute\_metrics=eval\_metric)

标准代码

Eg:

import evaluate  
from transformers import AutoTokenizer, AutoModelForSequenceClassification, Trainer, TrainingArguments, \  
 DataCollatorWithPadding, pipeline  
from datasets import load\_dataset  
  
# 1.加载数据集  
dataset = load\_dataset("csv", data\_files='classification/data/ChnSentiCorp\_htl\_all.csv', split='train')  
dataset = dataset.filter(lambda x: x['review'] is not None)  
  
# 2.划分数据集  
dataset\_dict = dataset.train\_test\_split(train\_size=0.9, test\_size=0.1)  
print(dataset\_dict)  
  
# 3.数据集预处理  
tokenizer = AutoTokenizer.from\_pretrained('classification/prefix\_model')  
  
  
def process\_function(examples):  
 tokenized\_examples = tokenizer(examples['review'], truncation=True, max\_length=128)  
 tokenized\_examples['labels'] = examples['label']  
 return tokenized\_examples  
  
  
tokenized\_dataset = dataset\_dict.map(process\_function, batched=True, remove\_columns=dataset\_dict['train'].column\_names)  
  
# 4.创建模型  
model = AutoModelForSequenceClassification.from\_pretrained('classification/prefix\_model')  
model.config.id2label = {0: '差评', 1: '好评'} # id2label中的0、1没有引号  
  
# 5.创建评估函数  
acc\_metrics = evaluate.load('accuracy')  
f1\_metrics = evaluate.load('f1')  
  
  
def eval\_metric(eval\_predict):  
 predictions, labels = eval\_predict  
 predictions = predictions.argmax(axis=-1)  
 acc = acc\_metrics.compute(references=labels, predictions=predictions)  
 f1 = f1\_metrics.compute(references=labels, predictions=predictions)  
 acc.update(f1)  
 return acc  
  
  
# 6.模型训练  
train\_args = TrainingArguments(  
 output\_dir='./checkpoints',  
 per\_device\_train\_batch\_size=64,  
 per\_device\_eval\_batch\_size=128,  
 logging\_steps=10,  
 eval\_strategy='epoch', # 含有该参数,才会对模型进行评估  
 save\_strategy='epoch',  
 learning\_rate=2e-5,  
 weight\_decay=0.01,  
 metric\_for\_best\_model='f1',  
 load\_best\_model\_at\_end=True  
)  
trainer = Trainer(model=model, args=train\_args, train\_dataset=tokenized\_dataset['train'],  
 eval\_dataset=tokenized\_dataset['test'], data\_collator=DataCollatorWithPadding(tokenizer=tokenizer),  
 compute\_metrics=eval\_metric)  
  
trainer.train()  
  
# 7.模型评估  
eval\_res = trainer.evaluate(tokenized\_dataset['test'])  
# 8.模型预测  
predict\_res = trainer.predict(tokenized\_dataset['test'])  
  
sen = "我觉得不错！"  
pipe = pipeline('text-classification', model=model, tokenizer=tokenizer, device=0)  
print(pipe(sen))

补充:

1.在Hugging Face的Trainer中,默认的迭代次数（即epochs）设置为3次,该参数可以通过TrainingArguments中的num\_train\_epochs参数进行调整。

2.评估函数—acc.compute() / f1.compute()返回的是dict字典类型,f1.update(acc)会更新字典f1,但其返回值为None。

3.模型训练Trainer中的参数compute\_metrics是一个回调函数。

compute\_metrics(`Callable[[EvalPrediction],Dict]`,\*optional\*)*,*该回调函数的参数是EvalPrediction(predictions、label\_ids、inputs(可缺省))。

4.评估指标函数eval\_metric(eval\_predict)的参数 eval\_predict是ndarray类型,而不是tensor类型。

## Transformers实战演练篇

* + - 1. Transformers显存优化

简介:显存优化策略,4G显存也可以跑BERT-Large

A.显存占用简单分析:

模型权重:

-4Bytes \* 模型参数量

优化器状态:

-8Bytes \* 模型参数量,对于常用的Adam、AdamW优化器而言

梯度:

-4Bytes \* 模型参数量

前向激活值

-取决于序列长度、隐层维度、Batch大小等多个因素

Eg:

1.bert-base是110M — 0.1b的一个模型; bert-large是330M — 0.3b的一个模型

2.深度学习中,每个权重参数、梯度等都是用FP32（32bit浮点数表示）,因为能够提供足够的精度，并且大部分硬件(GPU/CPU)都对32bit浮点数进行了优化。

3.前向激活值—中间变量,需要保存下来,更好的去计算梯度

B.显存优化策略

模型:hfl/chinese-macbert-large, 330M 设备:4090显卡

每一当前优化策略是在前面全部优化策略的基础上进行的

|  |  |  |  |
| --- | --- | --- | --- |
| 优化策略 | 优化对象 | 显存占用 | 训练时间 |
| 基线(bs 32,ml 128) | - | 15.2G | 64s |
| +梯度累加(bs 1,ga 32) | 前向激活值 | 7.4G | 259s |
| +梯度检查(bs 1,ga 32) | 前向激活值 | 7.2G | 422s |
| +自适应优化器(bs 1,ga 32) | 优化器状态 | 5.0G | 406s |
| +冻结模型(bs 1,ga 32) | 前向激活值/模型 | 3.5G | 178s |
| +数据长度(bs 1,ga 32,ml 32) | 前向激活值 | 3.4G | 126s |

bs:batch\_size 32、ml:max\_length 128、ga:gradient\_accumulation、

基线情况下: GPU 显存不足，无法为任务分配所需的 16MB 显存（CUDA out of memory. Tried to allocate 16.00 MiB.）

+梯度累加:TrainingArgument中设置参数gradient\_accumulation\_steps=32;

+梯度检查:梯度累加基础上,TrainingArgument中设置参数gradient\_checkpointing=True（只保存一部分激活值来降低显存消耗,未保存的激活值在反向传播时通过重新计算得到。）

+自适应优化器:梯度检查基础上,TrainingArgument中设置参数optim=‘adafactor’

+冻结模型:可以独立于梯度累加、梯度检查、自适应优化器、数据长度的显存优化策略

（不仅可以降低显存,还可以提升速度;训练模型的时候,冻结模型中的一些参数;比如文本分类的任务,其结构是由一个bert结构和全连接层结构构成的,可以把预训练中bert结构部分的参数全部冻结,只训练全连接层结构部分的参数;冻结模型策略让模型可训练的参数变少了,最终模型的**效果可能就不好了**。）

Eg:

for name, param in model.bert.named\_parameters():   
 param.requires\_grad = False

# 该处的model是AutoModelForSequenceClassification,包含bert结构和全连接层结构

+数据长度:可以独立于梯度累加、梯度检查、自适应优化器、冻结模型的显存优化策略

（会很影响模型效果的;长度是很重要的,有些数据就是很长,强行截断得很短,其**可能的效果就不明显**。）

Eg:

1.批量梯度下降法,batch\_size太大或太小都不适合(优化效果均不好)?

batch\_size太大:内存占用太大,整体训练速度下降。

batch\_size太小:梯度更新不稳定,收敛速度变慢,训练时间会延长。

2.其他显存优化策略?

参数高效微调（如Lora）、cpu offload、flash attention等技巧将在LLM章节进行讲解。

* + - 1. 命名实体识别

简介:命名实体识别（简称NER）是指识别文本中具有特定意义的实体,主要包括人名、地名、机构名、专有名词等。通常包括两部分: 1).**实体边界识别**; 2).**确定实体类别**（人名、地名、机构名或其他）。

I. 数据标注体系: IOB1、IOB2、IOE1、IOE2、IOBES、BILOU

* IOB2标注

I — 表示实体内部 O — 表示实体外部 B — 表示实体开始

B/I-XXX, XXX表示具体的类别

|  |  |
| --- | --- |
| 标记 | 说明 |
| B-Person | 人名开始 |
| I-Person | 人名中间 |
| B-Organization | 组织名开始 |
| I-Organization | 组织名中间 |
| O | 非命名实体 |

* IOBES标注

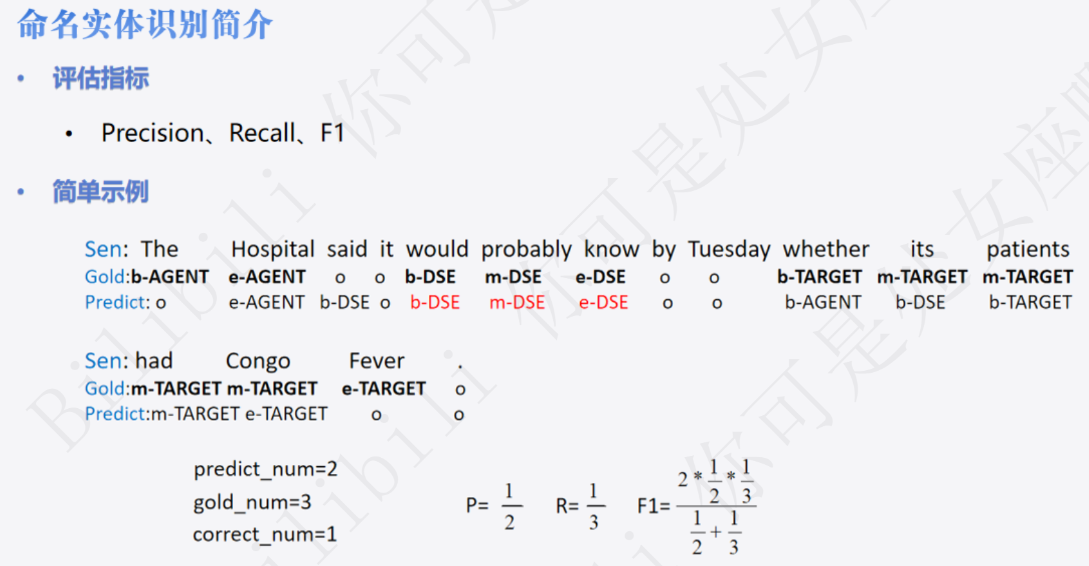
I/M — 表示实体内部 O — 表示实体外部 B — 表示实体开始

E — 表示实体结束 S — 表示一个词单独形成一个命名实体

II. 评估指标:

* + Precision、Recall、F1

简单示例:



Eg:

1.F1值越大时,意味着模型的精准率和召回率之间取得更好的平衡,同时意味着模型的精准率和召回率都较大;

— **分类错误率低** — **对正样本的识别能力强**

III. 模型结构:

BertForTokenClassification

1. 源码:

class BertForTokenClassification(BertPreTrainedModel):  
 def \_\_init\_\_(self, config):  
 super().\_\_init\_\_(config)  
 self.num\_labels = config.num\_labels  
  
 self.bert = BertModel(config, add\_pooling\_layer=False)  
 classifier\_dropout = (  
 config.classifier\_dropout if config.classifier\_dropout is not None else config.hidden\_dropout\_prob  
 )  
 self.dropout = nn.Dropout(classifier\_dropout)  
 self.classifier = nn.Linear(config.hidden\_size, config.num\_labels)  
  
 # Initialize weights and apply final processing  
 self.post\_init()  
  
 def forward(  
 self,  
 input\_ids: Optional[torch.Tensor] = None,  
 attention\_mask: Optional[torch.Tensor] = None,  
 token\_type\_ids: Optional[torch.Tensor] = None,  
 position\_ids: Optional[torch.Tensor] = None,  
 head\_mask: Optional[torch.Tensor] = None,  
 inputs\_embeds: Optional[torch.Tensor] = None,  
 labels: Optional[torch.Tensor] = None,  
 output\_attentions: Optional[bool] = None,  
 output\_hidden\_states: Optional[bool] = None,  
 return\_dict: Optional[bool] = None,  
 ) -> Union[Tuple[torch.Tensor], TokenClassifierOutput]:  
 *r"""  
 labels (`torch.LongTensor` of shape `(batch\_size, sequence\_length)`, \*optional\*):  
 Labels for computing the token classification loss. Indices should be in `[0, ..., config.num\_labels - 1]`.  
 """* return\_dict = return\_dict if return\_dict is not None else self.config.use\_return\_dict  
  
 outputs = self.bert(  
 input\_ids,  
 attention\_mask=attention\_mask,  
 token\_type\_ids=token\_type\_ids,  
 position\_ids=position\_ids,  
 head\_mask=head\_mask,  
 inputs\_embeds=inputs\_embeds,  
 output\_attentions=output\_attentions,  
 output\_hidden\_states=output\_hidden\_states,  
 return\_dict=return\_dict,  
 )  
  
 sequence\_output = outputs[0]  
  
 sequence\_output = self.dropout(sequence\_output)  
 logits = self.classifier(sequence\_output)  
  
 loss = None  
 if labels is not None:  
 loss\_fct = CrossEntropyLoss()  
 loss = loss\_fct(logits.view(-1, self.num\_labels), labels.view(-1))  
  
 if not return\_dict:  
 output = (logits,) + outputs[2:]  
 return ((loss,) + output) if loss is not None else output  
  
 return TokenClassifierOutput(  
 loss=loss,  
 logits=logits,  
 hidden\_states=outputs.hidden\_states,  
 attentions=outputs.attentions,  
 )

补充:

1. BertModel的返回值是BaseModelOutputWithPoolingAndCrossAttentions类型(outputs = self.bert());其中outputs[0]是last\_hidden\_state, last\_hidden\_state表示输入序列中每个 token 对应的最后一层隐藏状态,它是一个三维张量(batch\_size, seq\_len, embedding\_size)，包含了每个 token 的上下文嵌入;其中outputs[1]是pooler\_output, pooler\_output 是通过对 [CLS] token 的隐藏状态(即 last\_hidden\_state 的第一个 token)进行池化操作获得的。

2.Union联合类型

-> Union[Tuple[Tensor],BaseModelOutputWithPoolingAndCrossAttentions]

返回值的类型可以是BaseModelOutputWithPoolingAndCrossAttentions,也可以是Tuple[torch.Tensor]

Eg1:

var\_1: Union[str, int] = 0  
var\_2: Union[str, int] = 'hello world'

Eg2:

var\_list: list[Union[int, str]] = [1, 2, 'LinZeyuan']  
print(var\_list)

3. BaseModelOutputWithPoolingAndCrossAttentions通过索引[下标]访问的思路

Eg:

class Res(**OrderedDict**):  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
  
 def \_\_str\_\_(self):  
 return str(self.\_\_dict\_\_)  
  
 def \_\_getitem\_\_(self, item):  
 return list(self.\_\_dict\_\_.values())[item]  
  
  
res = Res(name='<NAME>', age=18)  
print(res) # {'name': '<NAME>', 'age': 18}  
print(res[0]) # <NAME>  
print(res[1]) # 18

IV. 评估函数:

pip install seqeval evaluate.load(evaluate)

V. 代码实战演练

数据集:

* peoples\_daily\_ner(人民日报的ner数据集）

预训练模型:

* hfl/chinese-macbert-base

Eg:

# 1.数据集加载

ner\_datasets = load\_dataset(path='./data')

print(ner\_datasets['train'][0])

#{'id': '0', 'tokens': ['海', '钓', '比', '赛', '地', '点', '在', '厦', '门', '与', '金', '门', '之', '间', '的', '海', '域', '。'], 'ner\_tags': [0, 0, 0, 0, 0, 0, 0, 5, 6, 0, 5, 6, 0, 0, 0, 0, 0, 0]}

print(ner\_datasets['train'].features)

#{'id': Value(dtype='string', id=None), 'tokens': Sequence(feature=Value(dtype='string', id=None), length=-1, id=None), 'ner\_tags': Sequence(feature=ClassLabel(names=['O', 'B-PER', 'I-PER', 'B-ORG', 'I-ORG', 'B-LOC', 'I-LOC'], id=None), length=-1, id=None)}

补充:

1.load\_dataset()中的cache\_dir参数可以指定数据集的下载位置

dataset=load\_dataset(path="data/peoples\_daily\_ner.py",trust\_remote\_code=True,cache\_dir='data');

2.peoples\_daily\_ner数据集包括了id、tokens、ner\_tags字段,每个tokens分词对应一个ner\_tags类别标记;

3.sequence(datasets中的数据结构)可以是单一类型的特征列表或不同类型的特征字典,通常用于指定数据集(datasets)的特征(features);

Eg:

import datasets

from datasets import Dataset, Sequence, Value, Features

from transformers import AutoTokenizer

# 创建数据（text列的数据、label列的数据，且text与label要一一对应）

dataset = {  
 'text': [{  
 'question': 'hello world',  
 'answer': 'hello world'  
 }, {  
 'question': 'hello world',  
 'answer': 'hello world'  
 }, {  
 'question': 'hello world',  
 'answer': 'hello world'  
 }],  
 'label': [0, 1, 2],  
}

# 定义特征（定义text列、label列的数据类型）

feature = {  
 'text': Sequence({  
 'question': Value('string'),  
 'answer': Value('string')  
 }),  
 'label': Value('int32'),  
}  
  
dataset = Dataset.from\_dict(dataset, features=Features(feature))

VI. 代码部分:

import torch  
from datasets import load\_dataset  
from transformers import AutoTokenizer, AutoModelForTokenClassification, TrainingArguments, Trainer, \  
 DataCollatorForTokenClassification, pipeline  
from evaluate import load  
import numpy as np  
# 分布式训练  
import os  
  
os.environ["NCCL\_P2P\_DISABLE"] = '1'  
os.environ["NCCL\_IB\_DISABLE"] = '1'  
os.environ["CUDA\_VISIBLE\_DEVICES"] = '0,1'  
  
# 1.数据集加载  
ner\_dataset = load\_dataset("data")  
label\_list = ner\_dataset['train'].features['ner\_tags'].feature.names  
  
# 2.数据集预处理  
tokenizer = AutoTokenizer.from\_pretrained('prefix\_model')  
  
  
def preprocess(batch):  
 tokenized\_batch = tokenizer(batch['tokens'], max\_length=128, truncation=True, padding='max\_length',  
 is\_split\_into\_words=True)  
 labels = []  
 for i, label in enumerate(batch['ner\_tags']):  
 word\_ids = tokenized\_batch.word\_ids(batch\_index=i)  
 label\_ids = []  
 for word\_id in word\_ids:  
 if word\_id is None:  
 label\_ids.append(-100)  
 else:  
 label\_ids.append(label[word\_id])  
 labels.append(label\_ids)  
 tokenized\_batch['labels'] = labels  
 return tokenized\_batch  
  
  
ner\_dataset\_map = ner\_dataset.map(preprocess, batched=True, remove\_columns=ner\_dataset['train'].column\_names)  
  
# 3.加载模型  
model = AutoModelForTokenClassification.from\_pretrained('prefix\_model', num\_labels=len(label\_list))  
model.config.id2label = {idx: label for idx, label in enumerate(label\_list)}  
# 4.加载评估函数  
seqeval\_metrics = load(path='metrics/seqeval.py')  
  
  
def evaluate\_compute(evaluation\_parms):  
 predictions, labels = evaluation\_parms  
 predictions = np.argmax(predictions, axis=-1)  
 true\_predictions = [  
 [label\_list[p] for p, l in zip(prediction, label) if l != -100]  
 for prediction, label in zip(predictions, labels)  
 ]  
 true\_labels = [  
 [label\_list[l] for p, l in zip(prediction, label) if l != -100]  
 for prediction, label in zip(predictions, labels)  
 ]  
 metrics = seqeval\_metrics.compute(predictions=true\_predictions, references=true\_labels, scheme='IOB2',  
 mode='strict')  
 return {  
 'overall\_f1': metrics['overall\_f1'],  
 'person\_f1': metrics['PER']['f1'],  
 'location\_f1': metrics['LOC']['f1'],  
 }  
  
  
# 5.模型训练  
traning\_args = TrainingArguments(  
 output\_dir='logs',  
 per\_device\_train\_batch\_size=32,  
 per\_device\_eval\_batch\_size=128,  
 num\_train\_epochs=3,  
 logging\_steps=10,  
 eval\_strategy='epoch',  
 save\_strategy='epoch',  
 metric\_for\_best\_model='person\_f1',  
 load\_best\_model\_at\_end=True,  
)  
trainer = Trainer(model=model, tokenizer=tokenizer, args=traning\_args,  
 train\_dataset=ner\_dataset\_map['train'], eval\_dataset=ner\_dataset\_map['test'],  
 compute\_metrics=evaluate\_compute)  
  
trainer.train()  
  
# 6.模型预测  
pipe = pipeline(task='token-classification', model=model, tokenizer=tokenizer, aggregation\_strategy='simple', device=0)  
sens = '爱迪生在美国洛杉矶发明了灯泡'  
results = pipe(sens)  
  
person\_entity = []  
location\_entity = []  
for result in results:  
 start = result['start']  
 end = result['end']  
 if (result['entity\_group'] == 'PER'):  
 person\_entity.append(sens[start:end])  
 elif (result['entity\_group'] == 'LOC'):  
 location\_entity.append(sens[start:end])  
print(person\_entity)  
print(location\_entity)

补充:

1.token是分词;input\_ids是token映射成向量表中的id(并不一定是一一对应的);word\_ids()方法(**BatchEncoding中的方法**),用于获取输入文本中每个标记对应的原始词ID;[!important]确保input\_ids和labels的长度匹配很重要。

2.num\_labels和id2label参数的区别:

num\_labels:表示模型可以预测的标签数量,即分类数;

id2label:用于将标签的整数ID映射到标签的实际名称上,即ID-分类名称映射;

3.模型输入参数、评估函数评估的测试方法:

# 模型输入参数的测试

input = {k: torch.tensor(v).cuda() for k, v in ner\_dataset\_map['train'][0:100].items()}  
output = model(\*\*input)

# 评估函数评估的测试  
res = evaluate\_compute((output['logits'].cpu().detach().numpy(), np.array(ner\_dataset\_map['train'][0:100]['labels'])))

4.多分类任务,在模型训练前,要指定id2label[自动指定label2id],确保softmax-onehot环节不出错

5.Trainer中的参数metric\_for\_best\_model,根据单一指标来保存模型;

根据多个指标保存模型的方式:

1）.将平均值作为新的综合指标,来保存合适的模型;

combined\_metric = (accuracy + f1) / 2  
  
return {  
 "accuracy": accuracy,  
 "f1": f1,  
 "combined\_metric": combined\_metric # 新的综合指标  
}

2）.自定义回调函数,实现根据多个指标保存模型的逻辑;

from transformers import TrainerCallback, TrainerControl  
  
  
class MultiMetricSaveCallback(TrainerCallback):  
 def \_\_init\_\_(self):  
 self.best\_f1 = -float("inf")  
 self.best\_accuracy = -float("inf")  
  
 def on\_evaluate(self, args, state, control: TrainerControl, metrics=None, \*\*kwargs):  
 # 获取指标  
 current\_f1 = metrics.get("eval\_f1")

current\_accuracy = metrics.get("eval\_accuracy")  
  
 # 比较并保存模型  
 if current\_f1 > self.best\_f1 and current\_accuracy > self.best\_accuracy:  
 print(f"New best model found! Accuracy: {current\_accuracy}, F1: {current\_f1}")  
 self.best\_f1 = current\_f1  
 self.best\_accuracy = current\_accuracy  
 control.should\_save = True # 触发保存模型  
 else:  
 control.should\_save = False # 不保存模型

6.pipeline中的参数aggregation\_strategy='simple',可以指定将token聚合成命名实体。

7.tokenizer在decoder过程中,会默认以空格来分隔

[problem]:

sens = "爱迪生在美国洛杉矶发明了电灯炮"  
tokenizer = AutoTokenizer.from\_pretrained('prefix\_model')  
tokenized\_sens = tokenizer(sens)  
print(tokenized\_sens)  
print(tokenizer.decode(tokenized\_sens['input\_ids']))

[solve]:

pipe = pipeline(task='token-classification', model=model, tokenizer=tokenizer, aggregation\_strategy='simple', device=0)  
sens = '爱迪生在美国洛杉矶发明了灯泡'  
results = pipe(sens)  
  
person\_entity = []  
location\_entity = []  
for result in results:  
 start = result['start']  
 end = result['end']  
 if (result['entity\_group'] == 'PER'):  
 person\_entity.append(sens[start:end])  
 elif (result['entity\_group'] == 'LOC'):  
 location\_entity.append(sens[start:end])  
print(person\_entity)  
print(location\_entity)

* + - 1. 机器阅读理解

简介:机器阅读理解是一项通过让机器回答基于给定上下文的问题来测试机器理解自然语言的程度的任务,简单来说即给定一个或者多个文档P以及一个问题Q,输出问题Q的答案A。

机器阅读理解任务的形式是多样化的,常见的类型包括完型填空式、答案选择式、片段抽取式、自由生成式,本次课程讲解的内容为片段抽取式的机器阅读理解,即问题Q的答案A在文档P中,A是P中的一个连续片段。

I. 建模:定位答案在文档中的起始位置和结束位置。(类似命名实体识别的方式也可以)

II. 数据集格式:

CMRC2018

{

“context”: “战国无双3“是由光荣和w-Force开发的战国无双系列的正统第三续作.本作

以三大故事为主轴，分别是以武田信玄等人为主的“关东三国志”，织田信长等人为主的“战国三杰“，石田三成等人为主的”关原的年轻武者“，丰富游戏内的剧情.此部份专门介绍角色，

欲知武…”,

“id”: “DEV\_0\_QUERY\_0”,

“question”: “《战国无双3》是由哪两个公司合作开发的？”,

“answers”: {

“answer\_start”: [11, 11],

“text”: [“光荣和w-force”, “光荣和w-force”]

},

}

III. 评估指标

精准匹配度(EM):计算预测结果与标准答案是否完全匹配。

模糊匹配度(F1):计算预测结果与标准答案之间字级别的匹配程度。

简单示例:

* 数据
  + - 模型预测结果: 北京
    - 真实标签结果: 北京天安门
* 计算结果
  + - EM = 0 P = 2/2 R = 2/5 F1 = (2\*2/2\*2/5)/(2/2+2/5) = 4/7 ~ 0.57
* 相关包安装
  + - pip install nltk
    - nltk.download(“punkt”)

IV. 数据预处理

* 数据处理格式

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| [CLS] | Question | [SEP] | Context | [SEP] |

* 如何准确定位答案位置
  + - start\_positions / end\_positions
    - offset\_mapping (与命名实体识别部分的word\_id
* Context过长如何解决
  + - 策略1：直接截断,简单易实现,但是会损失答案靠后的数据,因为无法定位答案
    - 策略2：**滑动窗口,实现较为复杂,会丢失部分上下文,但是综合来看损失较小**

V. 模型结构

BertForQuestionAnswering:

1.源码:

class BertForQuestionAnswering(BertPreTrainedModel):  
 def \_\_init\_\_(self, config):  
 super().\_\_init\_\_(config)  
 self.num\_labels = config.num\_labels  
  
 self.bert = BertModel(config, add\_pooling\_layer=False)  
 self.qa\_outputs = nn.Linear(config.hidden\_size, config.num\_labels)  
  
 # Initialize weights and apply final processing  
 self.post\_init()  
  
 def forward(  
 self,  
 input\_ids: Optional[torch.Tensor] = None,  
 attention\_mask: Optional[torch.Tensor] = None,  
 token\_type\_ids: Optional[torch.Tensor] = None,  
 position\_ids: Optional[torch.Tensor] = None,  
 head\_mask: Optional[torch.Tensor] = None,  
 inputs\_embeds: Optional[torch.Tensor] = None,  
 start\_positions: Optional[torch.Tensor] = None,  
 end\_positions: Optional[torch.Tensor] = None,  
 output\_attentions: Optional[bool] = None,  
 output\_hidden\_states: Optional[bool] = None,  
 return\_dict: Optional[bool] = None,  
 ) -> Union[Tuple[torch.Tensor], QuestionAnsweringModelOutput]:  
return\_dict = return\_dict if return\_dict is not None else self.config.use\_return\_dict  
  
 outputs = self.bert(  
 input\_ids,  
 attention\_mask=attention\_mask,  
 token\_type\_ids=token\_type\_ids,  
 position\_ids=position\_ids,  
 head\_mask=head\_mask,  
 inputs\_embeds=inputs\_embeds,  
 output\_attentions=output\_attentions,  
 output\_hidden\_states=output\_hidden\_states,  
 return\_dict=return\_dict,  
 )  
  
 sequence\_output = outputs[0]  
  
 logits = self.qa\_outputs(sequence\_output)  
 start\_logits, end\_logits = logits.split(1, dim=-1)  
 start\_logits = start\_logits.squeeze(-1).contiguous()  
 end\_logits = end\_logits.squeeze(-1).contiguous()  
  
 total\_loss = None  
 if start\_positions is not None and end\_positions is not None:  
 # If we are on multi-GPU, split add a dimension  
 if len(start\_positions.size()) > 1:  
 start\_positions = start\_positions.squeeze(-1)  
 if len(end\_positions.size()) > 1:  
 end\_positions = end\_positions.squeeze(-1)  
 # sometimes the start/end positions are outside our model inputs, we ignore these terms  
 ignored\_index = start\_logits.size(1)  
 start\_positions = start\_positions.clamp(0, ignored\_index)  
 end\_positions = end\_positions.clamp(0, ignored\_index)  
  
 loss\_fct = CrossEntropyLoss(ignore\_index=ignored\_index)  
 start\_loss = loss\_fct(start\_logits, start\_positions)  
 end\_loss = loss\_fct(end\_logits, end\_positions)  
 total\_loss = (start\_loss + end\_loss) / 2  
  
 if not return\_dict:  
 output = (start\_logits, end\_logits) + outputs[2:]  
 return ((total\_loss,) + output) if total\_loss is not None else output  
  
 return QuestionAnsweringModelOutput(  
 loss=total\_loss,  
 start\_logits=start\_logits,  
 end\_logits=end\_logits,  
 hidden\_states=outputs.hidden\_states,  
 attentions=outputs.attentions,  
 )

2.forward部分代码注释:

sequence\_output = outputs[0] # [batch\_size, seq\_len, embedding\_size]  
logits = self.qa\_outputs(sequence\_output) # [batch\_size, seq\_len, 2]  
start\_logits, end\_logits = logits.split(1, dim=-1)   
# [batch\_size, seq\_len, 1] 、[batch\_size, seq\_len, 1]

start\_logits = start\_logits.squeeze(-1).contiguous() # [batch\_size, seq\_len]  
end\_logits = end\_logits.squeeze(-1).contiguous() # [batch\_size, seq\_len]

补充:

1. token\_type\_id是为了区分当前段文本是来自第一段,还是来自第二段;[SEP]可以用来划分文本的段落层次;

2. 在多GPU训练中,需要在不同的GPU之间分配数据时,通常会将输入数据的维度进行扩展,以便每个GPU能够处理其分配到的部分,通常涉及到对张量进行维度的调整;

3. 交叉熵函数CrossEntropyLoss的input参数的向量维度为[batch\_size, num\_classes] / [batch\_size, num\_classes, seq\_len],target参数的向量维度为[batch\_size] / [batch\_size, seq\_len];

VI. 代码实战演练

* 数据集 — cmrc2018
* 预训练模型 — hfl/chinese-macbert-base
* 数据集处理方式 — 对context进行截断处理
* 代码实现[基于**滑动窗口**的数据处理]

VII. 代码部分

import os

from datasets import load\_dataset

from transformers import AutoTokenizer, AutoModelForQuestionAnswering, TrainingArguments, Trainer, DataCollatorWithPadding, pipeline

os.environ['NCCL\_P2P\_DISABLE'] = '1'

os.environ['NCCL\_IB\_DISABLE'] = '1'

os.environ['CUDA\_VISIBLE\_DEVICES'] = '0,1'

# 1.数据集加载

dataset = load\_dataset('data')

print(dataset)

# 2.数据集预处理

tokenizer = AutoTokenizer.from\_pretrained('prefix\_model')

def preProcess(batch):

tokenized\_batch = tokenizer(batch['question'], text\_pair=batch['context'], max\_length=512, truncation='only\_second',padding='max\_length', return\_offsets\_mapping=True)

start\_positions = []

end\_positions = []

for idx, offset in enumerate(tokenized\_batch['offset\_mapping']):

start\_char = batch['answers'][idx]['answer\_start'][0]

end\_char = start\_char + len(batch['answers'][idx]['text'][0])

start\_context = tokenized\_batch.sequence\_ids(idx).index(1)

end\_context = tokenized\_batch.sequence\_ids(idx).index(None, start\_context) - 1

if offset[end\_context][1] < start\_char or offset[start\_context][0] > end\_char:

start\_position = 0

end\_position = 0

else:

token\_id = start\_context

while token\_id <= end\_context and offset[token\_id][0] < start\_char:

token\_id += 1

start\_position = token\_id

token\_id = end\_context

while token\_id >= start\_context and offset[token\_id][1] > end\_char:

token\_id -= 1

end\_position = token\_id

start\_positions.append(start\_position)

end\_positions.append(end\_position)

tokenized\_batch['start\_positions'] = start\_positions

tokenized\_batch['end\_positions'] = end\_positions

return tokenized\_batch

map\_dataset=dataset.map(preProcess,batched=True,remove\_columns=dataset['train'].column\_names)

train\_dataset = map\_dataset['train']

test\_dataset = map\_dataset['test']

# 3.创建模型

model = AutoModelForQuestionAnswering.from\_pretrained('prefix\_model')

# 4.模型训练

training\_args = TrainingArguments(

output\_dir='fine\_tuned\_model\_1',

per\_device\_train\_batch\_size=32,

per\_device\_eval\_batch\_size=128,

num\_train\_epochs=3,

logging\_steps=15,

learning\_rate=5e-5,

evaluation\_strategy='epoch',

save\_strategy='epoch',

load\_best\_model\_at\_end=True,

)

trainer=Trainer(tokenizer=tokenizer,model=model,args=training\_args, train\_dataset=train\_dataset,eval\_dataset=test\_dataset, data\_collator=DataCollatorWithPadding(tokenizer=tokenizer))

trainer.train()

# 5.模型预测

pipe = pipeline('question-answering', model=model, tokenizer=tokenizer, device=0)

print(pipe(question="谁总有一天会完成sci二区并成功实习",

context="我虽然讨厌读书,但总有一天我林泽源一定会完成sci二区作品，并成功实习"))

VIII. [进阶]

A.基于滑动窗口的数据处理

滑动窗口:

* Context:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

* Query:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |

* Input:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [cls] | 1 | 2 | 3 | 4 | [sep] | 1 | 2 | 3 | 4 | 5 | [sep] |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [cls] | 1 | 2 | 3 | 4 | [sep] | 5 | 6 | 7 | 8 | 9 | [sep] |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [cls] | 1 | 2 | 3 | 4 | [sep] | 9 | 10 | 11 | 12 | [pad] | [sep] |

B.基于滑动窗口的结果预测

最终预测的时候,把三条输入数据的预测结果都取到,假设第一条数据取到2、7,第二条数据和第三条数据均取到CLS,综合三条数据作对比,最终结果取2-7;

补充:

1). Transformer中,一个token对应一个input\_id,但是每个分词token并不一定是一个完整的单词(单词“NLP”对应两个分词token['nl', '##p']);tokenizer中\_\_call\_\_()的参数return\_offsets\_mapping=True,返回一个列表,记录在原始文本中对应的字符的起始位置和终止位置,offsets\_mapping区分不同段的文本;tokenizer中word\_ids(batch\_index = i)的返回值,会返回每个token对应的原始文本中的单词ID。

2). tokenizer的\_\_call\_\_()返回值的token\_type\_ids(),虽然能区分不同的段,但不能将特殊标记标出来;sequence\_ids()可以表示该token属于哪个段,可以通过sequence\_ids()取得其对应的段并清除特殊标记的影响。

3). map的操作 合并操作 — update 取值操作 — pop

# HuggingFace 模型训练基本步骤

HuggingFace任务地址: <https://huggingface.co/tasks/text-classification>（介绍了不同任务可以使用的models、datasets、metrics）

1).Torch训练步骤:

1.加载数据集

2.创建DataSet

3.创建DataLoader

4.创建模型和优化器

5.训练和验证

6.模型预测

2).Trainer训练步骤（没有Dataset、DataLoader组件）

1.加载数据集

2.划分数据集

3.数据集预处理

4.创建模型

5.创建评估函数

6.模型训练

7.模型评估

8.模型预测