【Pytorch】【Transformers】一个基于transformers的自定义命名。。。

github:

本篇博客希望展示如何基于**transformers**提供的功能进行模型的开发，减少代码量，提高开发速度。

```
import torch
import warnings
import torch.nn as nn

from torch import Tensor
from typing import List, Dict
from dataclasses import dataclass, field
from torch.nn.utils.rnn import pad_sequence
from torch.utils.data import Dataset, DataLoader
from transformers.file_utils import logger, logging
from transformers.trainer_utils import EvalPrediction
from transformers.modeling_outputs import TokenClassifierOutput
from sklearn.metrics import f1_score, precision_score, recall_score
from transformers import TrainingArguments, Trainer, BertTokenizer, BertModel, BertPreTrainedModel

warnings.filterwarnings("ignore")
```

一、定义参数

```
@dataclass
class ModelArguments:
    use_lstm: bool = field(default=True, metadata={"help": "是否使用LSTM"})
    lstm_hidden_size: int = field(default=500, metadata={"help": "LSTM隐藏层输出的维度"})
    lstm_layers: int = field(default=1, metadata={"help": "堆叠LSTM的层数"})
    lstm_dropout: float = field(default=0.5, metadata={"help": "LSTM的dropout"})
    hidden_dropout: float = field(default=0.5, metadata={"help": "预训练模型输出向量表示的dropout"})
    ner_num_labels: int = field(default=12, metadata={"help": "需要预测的标签数量"})


@dataclass
class OurTrainingArguments:
    checkpoint_dir: str = field(default="./models/checkpoints", metadata={"help": "训练过程中的checkpoints的保存路径"})
    best_dir: str = field(default="./models/best", metadata={"help": "最优模型的保存路径"})
    do_eval: bool = field(default=True, metadata={"help": "是否在训练时进行评估"})
    epoch: int = field(default=5, metadata={"help": "训练的epoch"})
    train_batch_size: int = field(default=8, metadata={"help": "训练时的batch size"})
    eval_batch_size: int = field(default=8, metadata={"help": "评估时的batch size"})


@dataclass
class DataArguments:
    train_file: str = field(default="./data/train.txt", metadata={"help": "训练数据的路径"})
    dev_file: str = field(default="./data/dev.txt", metadata={"help": "测试数据的路径"})
```

二、读取数据

这里定义了一个用于保存数据的数据结构，这样的方法能够提高代码的可阅读性。

```
@dataclass
class Example:
    text: List[str] # ner的文本
    label: List[str] = None # ner的标签

    def __post_init__(self):
        if self.label:
            assert len(self.text) == len(self.label)
```

定义将文件中的ner数据保存为Example列表的函数

```
def read_data(path):
    examples = []
    with open(path, "r", encoding="utf-8") as file:
        text = []
        label = []
        for line in file:
            line = line.strip()
            # 一条文本结束
            if len(line) == 0:
                examples.append(Example(text, label))
                text = []
                label = []
                continue
            text.append(line.split()[0])
            label.append(line.split()[1])
    return examples

train_data = read_data("./data/train.txt")
eval_data = read_data("./data/dev.txt")
print(train_data[0])
```

```
Example(text=['回', '眸', '"', '9', '7', '房', '地', '产', '景', '气', '水', '平', '缓', '缓', '回', '升'], label=['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O'])
```

加载标签数据并分配对于的id

```
def load_tag(path):
    with open(path, "r", encoding="utf-8") as file:
        lines = file.readlines()
        tag2id = {tag.strip(): idx for idx, tag in enumerate(lines)}
        id2tag = {idx: tag for tag, idx in tag2id.items()}
    return tag2id, id2tag

tag2id, id2tag = load_tag("./data/tag.txt")
print(tag2id)
print(id2tag)
```

```
{'<pad>': 0, 'O': 1, 'B-ORG': 2, 'I-ORG': 3, 'B-LOC': 4, 'I-LOC': 5, 'B-TIME': 6, 'I-TIME': 7, 'B-PER': 8, 'I-PER': 9, '<start>': 10, '<eos>': 11}
{0: '<pad>', 1: 'O', 2: 'B-ORG', 3: 'I-ORG', 4: 'B-LOC', 5: 'I-LOC', 6: 'B-TIME', 7: 'I-TIME', 8: 'B-PER', 9: 'I-PER', 10: '<start>', 11: '<eos>'}
```

读取tokenizer

```
tokenizer = BertTokenizer.from_pretrained("bert-base-chinese")
```

三、构建**Dataset**和**collate_fn**

构建Dataset

```
class NERDataset(Dataset):
    def __init__(self, examples: List[Example], max_length=128):
        self.max_length = 512 if max_length > 512 else max_length
        """
        1. 将文本的长度控制在max_length - 2，减2的原因是为[CLS]和[SEP]空出位置；
        2. 将文本转换为id序列；
        3. 将id序列转换为Tensor；
        """
        self.texts = [torch.LongTensor(tokenizer.encode(example.text[: self.max_length - 2])) for example in examples]
        self.labels = []
        for example in examples:
            label = example.label
            """
            1. 将字符的label转换为对于的id；
            2. 控制label的最长长度；
            3. 添加开始位置和结束位置对应的标签，这里<start>对应输入中的[CLS],<eos>对于[SEP]；
            4. 转换为Tensor；
            """
            label = [tag2id["<start>"]] + [tag2id[l] for l in label][: self.max_length - 2] + [tag2id["<eos>"]]
            self.labels.append(torch.LongTensor(label))
        assert len(self.texts) == len(self.labels)
        for text, label in zip(self.texts, self.labels):
            assert len(text) == len(label)

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, item):
        return {
            "input_ids": self.texts[item],
            "labels": self.labels[item]
        }

train_dataset = NERDataset(train_data)
eval_dataset = NERDataset(eval_data)
print(train_dataset[0])
```

```
{'input_ids': tensor([ 101, 1726, 4704,  100, 8037, 8035, 2791, 1765,  772, 3250, 3698, 3717,
        2398, 5353, 5353, 1726, 1285,  102]), 'labels': tensor([10,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1, 11])}
```

定义collate_fn，collate_fn的作用在Dataloader生成batch数据时会被调用。
这里的作用是对每个batch进行padding

```
def collate_fn(features) -> Dict[str, Tensor]:
    batch_input_ids = [feature["input_ids"] for feature in features]
    batch_labels = [feature["labels"] for feature in features]
    batch_attentiton_mask = [torch.ones_like(feature["input_ids"]) for feature in features]
    # padding
    batch_input_ids = pad_sequence(batch_input_ids, batch_first=True, padding_value=tokenizer.pad_token_id)
    batch_labels = pad_sequence(batch_labels, batch_first=True, padding_value=tag2id["<pad>"])
    batch_attentiton_mask = pad_sequence(batch_attentiton_mask, batch_first=True, padding_value=0)
    assert batch_input_ids.shape == batch_labels.shape
    return {"input_ids": batch_input_ids, "labels": batch_labels, "attention_mask": batch_attentiton_mask}
```

测试一下collate_fn

```
dataloader = DataLoader(train_dataset, shuffle=True, batch_size=2, collate_fn=collate_fn)
batch = next(iter(dataloader))
print(batch.keys())
print(type(batch["input_ids"]))
print(batch["input_ids"].shape)
print(type(batch["labels"]))
print(batch["labels"].shape)
print(type(batch["attention_mask"]))
print(batch["attention_mask"].shape)
```

```
dict_keys(['input_ids', 'labels', 'attention_mask'])
<class 'torch.Tensor'>
torch.Size([2, 19])
<class 'torch.Tensor'>
torch.Size([2, 19])
<class 'torch.Tensor'>
torch.Size([2, 19])
```

四、定义一个评估函数

```
def ner_metrics(eval_output: EvalPrediction) -> Dict[str, float]:
    """
    该函数是回调函数，Trainer会在进行评估时调用该函数。
    (如果使用Pycharm等IDE进行调试，可以使用断点的方法来调试该函数，该函数在进行评估时被调用)
    """
    preds = eval_output.predictions
    preds = np.argmax(preds, axis=-1).flatten()
    labels = eval_output.label_ids.flatten()
    # labels为0表示为<pad>，因此计算时需要去掉该部分
    mask = labels != 0
    preds = preds[mask]
    labels = labels[mask]
    metrics = dict()
    metrics["f1"] = f1_score(labels, preds, average="macro")
    metrics["precision"] = precision_score(labels, preds, average="macro")
    metrics["recall"] = recall_score(labels, preds, average="macro")
    # 必须以字典的形式返回，后面会用到字典的key
    return metrics
```

五、构建模型

- 自定义的模型需要继承BertPreTrainedModel

```
class BertForNER(BertPreTrainedModel):
    def __init__(self, config, *model_args, **model_kargs):
        super().__init__(config) # 初始化父类(必要的步骤)
        if "model_args" in model_kargs:
            model_args = model_kargs["model_args"]
            """
            必须将额外的参数更新至self.config中，这样在调用save_model保存模型时才会将这些参数保存；
            这种在使用from_pretrained方法加载模型时才不会出错；
            """
            self.config.__dict__.update(model_args.__dict__)
        self.num_labels = self.config.ner_num_labels
        self.bert = BertModel(config, add_pooling_layer=False)
        self.dropout = nn.Dropout(self.config.hidden_dropout)
        self.lstm = nn.LSTM(self.config.hidden_size, # 输入的维度
                            self.config.lstm_hidden_size, # 输出维度
                            num_layers=self.config.lstm_layers, # 堆叠lstm的层数
                            dropout=self.config.lstm_dropout,
                            bidirectional=True, # 是否双向
                            batch_first=True)
        if self.config.use_lstm:
            self.classifier = nn.Linear(self.config.lstm_hidden_size * 2, self.num_labels)
        else:
            self.classifier = nn.Linear(self.config.hidden_size, self.num_labels)
        self.init_weights()

    def forward(
            self,
            input_ids=None,
            attention_mask=None,
            token_type_ids=None,
            position_ids=None,
            head_mask=None,
```

```
        inputs_embeds=None,
        labels=None,
        pos=None,
        output_attentions=None,
        output_hidden_states=None,
        return_dict=None,
    ):
        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 = self.dropout(outputs[0])
        if self.config.use_lstm:
            sequence_output, _ = self.lstm(sequence_output)
        logits = self.classifier(sequence_output)

        loss = None
        if labels is not None:
            loss_fct = nn.CrossEntropyLoss()
            # 如果attention_mask不为空，则只计算attention_mask中为1部分的Loss
            if attention_mask is not None:
                active_loss = attention_mask.view(-1) == 1
                active_logits = logits.view(-1, self.num_labels)
                active_labels = torch.where(
                    active_loss, labels.view(-1), torch.tensor(loss_fct.ignore_index).type_as(labels)
                )
                loss = loss_fct(active_logits, active_labels)
            else:
                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, # 该部分在评估时，会作为EvalPrediction对象的predictions进行返回
            hidden_states=outputs.hidden_states,
            attentions=outputs.attentions,
        )
```

测试一下模型是否符合预期

```
model_args = ModelArguments(use_lstm=True)
model = BertForNER.from_pretrained("bert-base-chinese", model_args=model_args)
output = model(**batch)
print(type(output))
print(output.loss)
print(output.logits.shape)
```

```
<class 'transformers.modeling_outputs.TokenClassifierOutput'>
tensor(2.5061, grad_fn=<NllLossBackward>)
torch.Size([2, 19, 12])
```

六、模型训练

```python
def run(model_args: ModelArguments, data_args: DataArguments, args: OurTrainingArguments):
    # 设定训练参数
    training_args = TrainingArguments(output_dir=args.checkpoint_dir,  # 训练中的checkpoint保存的位置
                                      num_train_epochs=args.epoch,
                                      do_eval=args.do_eval,  # 是否进行评估
                                      evaluation_strategy="epoch",  # 每个epoch结束后进行评估
                                      per_device_train_batch_size=args.train_batch_size,
                                      per_device_eval_batch_size=args.eval_batch_size,
                                      load_best_model_at_end=True,  # 训练完成后加载最优模型
                                      metric_for_best_model="f1"  # 评估最优模型的指标，该指标是ner_metrics返回评估指标中的key
                                      )
    # 构建dataset
    train_dataset = NERDataset(read_data(data_args.train_file))
    eval_dataset = NERDataset(read_data(data_args.dev_file))
    # 加载预训练模型
    model = BertForNER.from_pretrained("bert-base-chinese", model_args=model_args)
    # 初始化Trainer
    trainer = Trainer(model=model,
                      args=training_args,
                      train_dataset=train_dataset,
                      eval_dataset=eval_dataset,
                      tokenizer=tokenizer,
                      data_collator=collate_fn,
                      compute_metrics=ner_metrics)
    # 模型训练
    trainer.train()
    # 训练完成后，加载最优模型并进行评估
    logger.info(trainer.evaluate(eval_dataset))
    # 保存训练好的模型
    trainer.save_model(args.best_dir)

# 定义各类参数并训练模型
model_args = ModelArguments(use_lstm=True)
data_args = DataArguments()
training_args = OurTrainingArguments(train_batch_size=16, eval_batch_size=32)
run(model_args, data_args, training_args)
```