Pytorch Lightning 框架

需要安装pytorch，pytorch-lightning (>=1.5)，yacs

整体框架分为四个主要文件夹：data，models，tb_logs，utils以及配置文件(config.py&config.yml)，main.py，lit_mode.py，subfunction.py

```
Pytorch-Lightning-Framework
|  config.py
|  config.yml
|  lit_model.py
|  main.py
|  result.txt
|  subfunction.py
|
├─data
|      data.py
|      dataset.py
|      __init__.py
|
├─models
|      model.py
|      __init__.py
|
├─utils
|      token_utils.py
|      __init__.py
|
└─tb_logs
    └─2022-03-09
        └─version_0
            |  hparams.yaml
            |
            └─checkpoints
                    model.ckpt
```

data

在data文件夹下实现数据集的操作，如初始化，载入数据集等

其中主要实现的是两个，一个是正常pytorch框架下的 **Dataset**，另一个是 **pl.LightningDataModule**

```
class MyDataSet(Dataset):
    def __init__(self, args):
        super(MyDataSet, self).__init__()
        # 数据集初始化如载入数据集等

    def __len__(self):
        # 统计全部数据个数
        return len(self.data)

    def __getitem__(self, item):
        # 返回item对应的数据
        data = self.data[item]
        return data
```

在 **MyDataset** 中一般实现对数据的读取和统计数据个数即可

```
import pytorch_lightning as pl

class MyDataModule(pl.LightningDataModule):
    def __init__(self, config):
        """
        完成数据初始化，如定义数据集路径，需要的数据集名称等
        """
        super(MyDataModule, self).__init__()
        # 获取预训练模型名字
        self.config = config
        self.pre_trained_name = config.PRETRAINED.tokenizer
        self.train_path = config.DATASETS.train
        self.test_path = config.DATASETS.test
        self.valid_path = config.DATASETS.valid

    def prepare_data(self):
        """
        如果数据集需要下载可以在这里定义方法
        """
        pass

    def setup(self, stage=None):
        """
        数据集建立方法，框架自动调用
        :param stage: 当前阶段 fit/test
        """
        if stage == 'fit' or stage is None:

            self.train_dataset = MyDataSet(self.train_path)
            self.valid_dataset = MyDataSet(self.test_path)
            pass
        if stage == 'test' or stage is None:
            self.test_dataset = MyDataSet(self.valid_path)
            pass

    # 以下三个函数均需要返回对应的dataloader
    def train_dataloader(self):
        return DataLoader(self.train_dataset,
batch_size=self.config.SOLVER.train_batch_size, num_workers=12,
collate_fn=collect_fn)
```

```
    def val_dataloader(self):
        return DataLoader(self.valid_dataset,
batch_size=self.config.SOLVER.valid_batch_size, num_workers=12,
collate_fn=collect_fn)

    def test_dataloader(self):
        return DataLoader(self.test_dataset,
batch_size=self.config.SOLVER.test_batch_size, num_workers=12,
collate_fn=collect_fn)
```

而在 **DataModule** 中则需要实现 **setup，train_dataloader，val_dataloader，test_dataloader** 这四个方法

其中 **setup** 负责根据不同阶段实例化之前的 **MyDatase**

train_dataloader，val_dataloader，test_dataloader 则是调用正常的 **Dataloader** 构建出迭代器

models&utils

在这两个文件夹下存放模型和相关的工具函数，这一部分就是正常的pytorch代码

tb_logs

负责保存模型以及记录关键信息，在这里使用的是 **tensorboard** 记录并按照日期存储

lit_model

整个框架最不同的就是该文件，其将正常pytorch中需要写的train/test for循环隐藏了起来，取而代之的是钩子函数，这一部分函数都会自动调用

首先是函数的初始化，在这里会完成网络模型的实例化，参数的保存

```
import pytorch_lightning as pl

class LitModel(pl.LightningModule):
    """
    pytorch lightning 模型
    """

    def __init__(self, config):
        """
        初始化模型
        Args:
            config: 模型基本参数
        """
        super(LitModel, self).__init__()

        self.config = config
        config_dict = yaml.load(config.dump(), Loader=yaml.FullLoader)
        self.save_hyperparameters(config_dict)
        self.model = MyModel(config)
```

接下来就是定义**training_step，validation_step，test_step**

这三个部分就是将之前for循环中写的内容放到这里，其中一些共同的步骤可以开一个新的函数share_step

```
def training_step(self, batch, batch_idx):
        """
        train 步骤
        Args:
            batch: 输入数据
            batch_idx: batch的索引
        Returns:
        """
        loss, f1, acc, precision, recall = self.share_step(batch, batch_idx)
        self.log("f1", f1, prog_bar=True, logger=False)
        return {"loss": loss, "train_f1": f1}

 def validation_step(self, batch, batch_idx):
        """
        validation 步骤
        """
        loss, f1, acc, precision, recall = self.share_step(batch, batch_idx)
        self.log("f1", f1, prog_bar=True, logger=False)
        return {"val_loss": loss, "val_f1": f1}

 def test_step(self, batch, batch_idx):
        """
        测试步骤
        Args:
            batch:
            batch_idx:
        Returns:
        """
        loss, f1, acc, precision, recall = self.share_step(batch, batch_idx)
        self.log("f1", f1)
        self.log("acc", acc)
        self.log("precision", precision)
        self.log("recall", recall)
```

其中的 **self.log()** 可以记录过程中的变量，如果开启prog_bar=True则会显示在训练中的进度条上

除了以上的钩子函数，还有**validation_epoch_end，test_epoch_end，optimizer_step**等，具体可以查看pytorch-lightning官网

在模型 **inference** 的时候使用的是 **lit_model** 中的 **forward** 函数

main&subfunction

这两个函数主要用于确定当前阶段并调用相关函数，具体的阶段函数负责在subfunction中实现

这其中有一个特殊类 **CustomProgressBar** 其作用是为了自定义进度条从而删除其中的 **v_num** 项

```
class CustomProgressBar(TQDMProgressBar):
    def __init__(self):
        super(CustomProgressBar, self).__init__()

    def get_metrics(self, trainer, model):
        # don't show the version number
        items = super().get_metrics(trainer, model)
        items.pop("v_num", None)
        return items
```

在 **train** 和 **test** 阶段需要配置 **trainer**

```
def train(config):
    model = LitModel(config)
    data_module = MyDataModule(config)
    bar = CustomProgressBar()
    checkpoint_callback = ModelCheckpoint(
        filename='{epoch}-{f1:.2f}',
        monitor='f1',
        save_top_k=config.SAVE.save_top_k,
        mode='max',
        every_n_epochs=config.SAVE.every_n_epochs)
    trainer = pl.Trainer(
        gpus=config.SOLVER.gpus,
        strategy=config.SOLVER.accelerator,
        max_epochs=config.SOLVER.max_epochs,
        callbacks=[checkpoint_callback, bar],
        logger=TensorBoardLogger(config.SAVE.logger,
name=config.SAVE.tb_log_path))
    trainer.fit(model, data_module)

def test(check_point, config_file):
    bar = CustomProgressBar()
    if os.path.exists(config_file):
        print("加载配置文件{}".format(config_file))
        cfgf = open(config_file)
        config = CN().load_cfg(cfgf)
    else:
        print("没有找到目标配置，将加载默认配置")
        from config import _C as config
    config.merge_from_file("config.yml")  # 可以修改测试数据集等
    trainer = pl.Trainer(gpus=config.SOLVER.gpus,
strategy=config.SOLVER.accelerator, logger=False,
                         callbacks=[bar])
    data_module = MyDataModule(config)
    model = LitModel(config)
    model.load_model(check_point)
```

其中包括使用的显卡，后端框架（ddp、dp等），最大训练epoch，自动保存文件的检测选项，以及 **logger** 的选择，这里面还有很多别的有用的选项如自动确定batchsize，限制训练集大小等

配置文件

框架使用的是 **yacs** 来帮助实现参数的记录，首先在 **config.py** 中的是默认参数

```
from yacs.config import CfgNode as CN
```

```
import os
import time

_C = CN()

_C.PRETRAINED = CN()
_C.PRETRAINED.tokenizer = "tb_logs/PreTrained/tokenizer/chinese-roberta-wwm-
ext/"  # 一部分预训练模型已经保存在本地，建议从本地直接加载
_C.PRETRAINED.model = "tb_logs/PreTrained/model/chinese-roberta-wwm-ext/"
_C.PRETRAINED.config = "tb_logs/PreTrained/model/chinese-roberta-wwm-
ext/config.json"

# 模型参数
_C.MODEL = CN()
_C.MODEL.window_size = 3
_C.MODEL.embedding_dim = 256
_C.MODEL.lstm_hidden_dim = 512
_C.MODEL.fc_hidden = 256
```

每一个类型的参数都可以定义一个主名称，如预训练模型的参数就在 **_C.PRETRAINED** 里

调用的时候只需要使用 **.** 即可调用

而 config.yml 中保存的是修改的参数，可以通过如下格式来覆盖默认参数

```
SOLVER:
  gpus: [0,1]
  max_epochs: 50
  train_batch_size: 256
  valid_batch_size: 256
  test_batch_size: 256
  threshold: 0.5  # 置信门限
```

整体流程

整个框架的运行流程是先判断当前运行的阶段：目前分为 **train、test、inference**

train

通过模型框架自动调用 **MyDataModule** 中的 **setup** 方法，然后获取相应的 **dataloader** 传入到 **training_step** 中，在 **training_step** 中计算的是一个 **batch** 的数据，等一轮 **epoch** 结束自动开始 val 步骤，如此重复，并且根据你设定的监控选项 **monitor** 自动保存模型，直到最大max_epoch

test&inference

这两个阶段都是需要指定对应的 **ckpt** 文件和与之对应的 **yaml** 配置文件，框架将会从 **yaml** 文件加载参数，并用以初始化模型，之后再从 **ckpt** 文件中加载权重，在 **test** 的时候是自动调用 **test** 的 **dataloader** 和 **test_step** ，在 **inference** 则是调用模型的 **forward** 部分