mmdetection2.6 自定义Runtime设置

一、 自定义优化器设置

1.1 自定义pytorch支持的优化器

支持很多的pytorch定义的优化器，唯一需要修改的就是去在`config`文件中修改`optimizer`字段的值。例如如果想要使用`ADAM`，可以使用如下的方式修改：

```
optimizer = dict(type='Adam', lr=0.0003, weight_decay=0.0001)
```

如果想要修改学习率，使用者可以在`optimizer`config文件中修改`lr`。使用者可以直接按照pytorch的API doc 设置arguments。

1.2 自定义优化器

1.2.1 定义新的优化器

假设想要添加`MyOptimizer`的新优化器，有`a`，`b`，`c`三个参数，需要创建一个新的路径`mmdet/core/optimizer`。然后在config文件中应用一个新的优化器`mmdet/core/optimizer/my_optimizer.py`：

```
from .registry import OPTIMIZERS
from torch.optim import Optimizer


@OPTIMIZERS.register_module()
class MyOptimizer(Optimizer):

  def __init__(self, a, b, c)
```

1.2.2 将新的优化器添加到注册器中

上边定义完成的模块想要在config文件中使用，就必须import到一个明明空间中，两种方法，实现这个工作：

- 直接在`__init__.py`文件中，导入，在`mmdet/core/optimizer/__init__.py`中

  ```
  from .my_optimizer import MyOptimizer
  ```

- 在config文件中实现

```
custom_imports = dict(imports=['mmdet.core.optimizer.my_optimizer'],
allow_failed_imports=False)
```

该模块`mmdet.core.optimizer.my_optimizer`将在程序开始时导入，然后自动注册`MyOptimizer`这个类。请注意，仅应该导入包含`MyOptimizer`的包。`mmdet.core.optimizer.my_optimizer.MyOptimizer`**无法**直接导入。

实际上，使用这种导入的方式，使用者可以直接使用不同的文件路径结构，只要模块的路径在`PYTHONPATH`中可以找到即可。

1.2.3 在config文件中指定优化器

在config文件中的`optimizer`字段中使用`MyOptimizer`：

```
# 原始的优化器使用方法
optimizer = dict(type='SGD', lr=0.02, momentum=0.9, weight_decay=0.0001)
# 自定义优化器使用方法
optimizer = dict(type='MyOptimizer', a=a_value, b=b_value, c=c_value)
```

1.3 自定义优化器constructor

一些模型对于优化器的参数有一些特定的设置，例如对于BN层的权重衰减。使用者可以使用自定义优化器的构建器来执行参数的微调。

```
from mmcv.utils import build_from_cfg

from mmcv.runner.optimizer import OPTIMIZER_BUILDERS, OPTIMIZERS
from mmdet.utils import get_root_logger
from .my_optimizer import MyOptimizer


@OPTIMIZER_BUILDERS.register_module()
class MyOptimizerConstructor(object):

  def __init__(self, optimizer_cfg, paramwise_cfg=None):

  def __call__(self, model):

    return my_optimizer
```

默认优化器构建器在这里实现,可以作为一个新的优化器构建器的模板。

1.4 其他的设置

优化器没有实现的技巧，可以通过优化器构建器实现。下边列出了一些常用稳定与加速训练的通用设置。

- 使用梯度裁剪来稳定训练

  ```
  optimizer_config = dict(
    _delete_=True, grad_clip=dict(max_norm=35, norm_type=2))
  ```

  如果config文件继承自base config，那么已经设置了optimizer_config， 可能需要`_delete_=True`来去除不必要的设置。

- 使用动量表加速模型收敛

支持使用动量进度表来一句学习率修改模型的动量，这种方式可以使得模型收敛的更快。动量进程表通常使用LR scheduler，例如接下来的config文件用在3D目标检测中来加速模型的收敛。

```
lr_config = dict(
  policy='cyclic',
  target_ratio=(10, 1e-4),
  cyclic_times=1,
  step_ratio_up=0.4,
)
momentum_config = dict(
  policy='cyclic',
  target_ratio=(0.85 / 0.95, 1),
  cyclic_times=1,
  step_ratio_up=0.4,
)
```

二、 自定义训练schedules

默认情况下，我们使用步长调整的学习率使用`1x`的方式。在MMCV中这称为`StepLRHook`， 我们也支持许多其他的schedule。例如 `CosineAnnealing` and `Poly` schedule。

- Poly schedule:

  ```
  lr_config = dict(policy='poly', power=0.9, min_lr=1e-4, by_epoch=False)
  ```

- ConsineAnnealing schedule:

```
lr_config = dict(
    policy='CosineAnnealing',
    warmup='linear',
    warmup_iters=1000,
    warmup_ratio=1.0 / 10,
    min_lr_ratio=1e-5)
```

三、 自定义workflow

wokflow是一个（phrase， epoch）的列表，使用这个参数来指定运行的顺序与epoches，默认的设置为

```
workflow = [('train', 1)]
```

这意味着训练1epoch， 有事使用者可能想要在验证集上检查某些评价指标，例如loss与准确率，这种情况下，我们可以设置：

```
[('train', 1), ('val', 1)]
```

这样训练一个epoch，验证一个epoch交替执行。

注意：

- 验证时，模型的参数不能更改
- config文件中的total_epochs，仅仅影响训练过程，对于val的过程没有影响。
- Workflows `[('train', 1), ('val', 1)]` and `[('train', 1)]` 不会改变 `EvalHook` 的行为，因为e `EvalHook` 在`after_train_epoch`调用验证的workflow仅仅影响在 `after_val_epoch`中调用的hooks. 因此，`[('train', 1), ('val', 1)]` 和 `` `[( ‘train’ , 1)] `` 唯一的不同点就是 runner将会在每个训练过程之后计算验证集的损失。

四、 自定义hooks

4.1 自定义hooks

4.1.1 实现新的hook

mmdetection支持自定义训练过程的hooks，使用者可以简单的修改config文件直接在mmdet或者给予mmdet的自己代码库中应用自己的hook

```
from mmcv.runner import HOOKS, Hook


@HOOKS.register_module()
class MyHook(Hook):

    def __init__(self, a, b):
        pass
```

```
  def before_run(self, runner):
    pass

  def after_run(self, runner):
    pass

  def before_epoch(self, runner):
    pass

  def after_epoch(self, runner):
    pass

  def before_iter(self, runner):
    pass

  def after_iter(self, runner):
    pass
```

基于hook的函数功能，仅仅需要指定在训练过程中这个hook需要完成的功能即可。在`before_run`, `after_run`, `before_epoch`, `after_epoch`, `before_iter`, and `after_iter`这些函数中修改即可。

4.1.2 注册新的hook

然后我们需要将，自定义的`MyHook`导入，假设存在`mmdet/core/utils/my_hook.py`这个文件中，两种方法来导入这个模块：

- 修改`mmdet/core/utils/__init__.py`来导入模块

  ```
  from .my_hook import MyHook
  ```

- 在config文件中修改

  ```
  custom_imports = dict(imports=['mmdet.core.utils.my_hook'], allow_failed_imports=False)
  ```

4.1.3 修改config文件

```
custom_hooks = [
  dict(type='MyHook', a=a_value, b=b_value)
]
```

也可以设置hook的优先级通过加入这个键值对`priority`值为`'NORMAL'`或`'HIGHEST'`：

```
custom_hooks = [
  dict(type='MyHook', a=a_value, b=b_value, priority='NORMAL')
]
```

默认的优先级为 NORMAL

4.2 使用MMCV实现的HOOKs

如果mmcv中存在对应的hook，可以直接使用：

```
custom_hooks = [
   dict(type='MyHook', a=a_value, b=b_value, priority='NORMAL')
]
```

4.3 修改默认运行时的hooks

有一些通用的hook，这些hook没有通过 custom_hooks 注册，这些hooks是：

- log_config
- checkpoint_config
- evaluation
- lr_config
- optimizer_config
- momentum_config

在这些hooks中，只有 log_config 具有 VERY_LOW 的优先级其他的都是 NORMAL 优先级。上面的介绍已经说明了如何修改 optimizer_config , momentum_config 和 lr_config 。这里介绍 log_config , checkpoint_config , and evaluation 这几个hooks。

4.3.1 checkpoint config

mmcv使用 checkpoint_config 来初始化 CheckpointHook 。

```
checkpoint_config = dict(interval=1)
```

使用者可以设置 max_keep_ckpts 来仅仅保存少数的几个checkpoint文件，或者使用 save_optimizer 决定是否保存优化器的状态.

4.3.2 log config

log_config 打包了很多的logger hooks并且能够设置间隔。现在MMCV支持 WandbLoggerHook , MlflowLoggerHook , and TensorboardLoggerHook 。更多细节参考：
https://mmcv.readthedocs.io/en/latest/api.html#mmcv.runner.LoggerHook

```
log_config = dict(
   interval=50,
   hooks=[
      dict(type='TextLoggerHook'),
      dict(type='TensorboardLoggerHook')
   ])
```

4.3.3 evaluation config

这个config文件用来初始化`EvalHook`，除了intervals这个参数之外，其他的变量例如`metric`将会传递给`dataset.evaluate()`

```
evaluation = dict(interval=1, metric='bbox')
```

专栏所有文章请点击下列文章列表查看：

知乎专栏：小哲AI专栏文章分类索引跳转查看

AI研习社专栏：小哲AI专栏文章分类索引