02. Decorator

CheolSeong Park
tjd4987@naver.com

02. Decorator

- 모델에서 서브 모듈을 **추가** 혹은 **변경**할 때마다, 모듈 선언 등의 코드 수정이 불가피함

- python decorator를 사용하여 모델이나 메인 코드의 수정 없이, 모듈을 변경하는 방법을 알아봅시다

02. Decorator

- 무슨 장점이 있나요?
  - DenseLayer를 ConvLayer로 바꾸고 싶을 때 코드 수정을 **덜** 할 수 있습니다.
    - 기존

```
class MyModel(Model):
    def __init__(self, cfg):
        super(MyModel, self).__init__()
        self.pre_layer = DenseLayer(cfg)

        self.d1 = Dense(128, activation=cfg['activation'])
        self.d2 = Dense(10)
```

```
class MyModel(Model):
    def __init__(self, cfg):
        super(MyModel, self).__init__()
        self.pre_layer = ConvLayer(cfg)

        self.d1 = Dense(128, activation=cfg['activation'])
        self.d2 = Dense(10)
```

    - Decorator 사용시
      - 코드 수정 x

```
class MyModel(Model):
    def __init__(self, cfg):
        super(MyModel, self).__init__()
        self.pre_layer = get_layer(cfg['prelayer'], cfg)

        self.d1 = Dense(128, activation=cfg['activation'])
        self.d2 = Dense(10)
```

02. Decorator

- 무슨 장점이 있나요?
  - 새로운 클래스 모듈이 추가되어, 기존 모델에 적용시에도 코드 수정이 필요하지 않습니다.

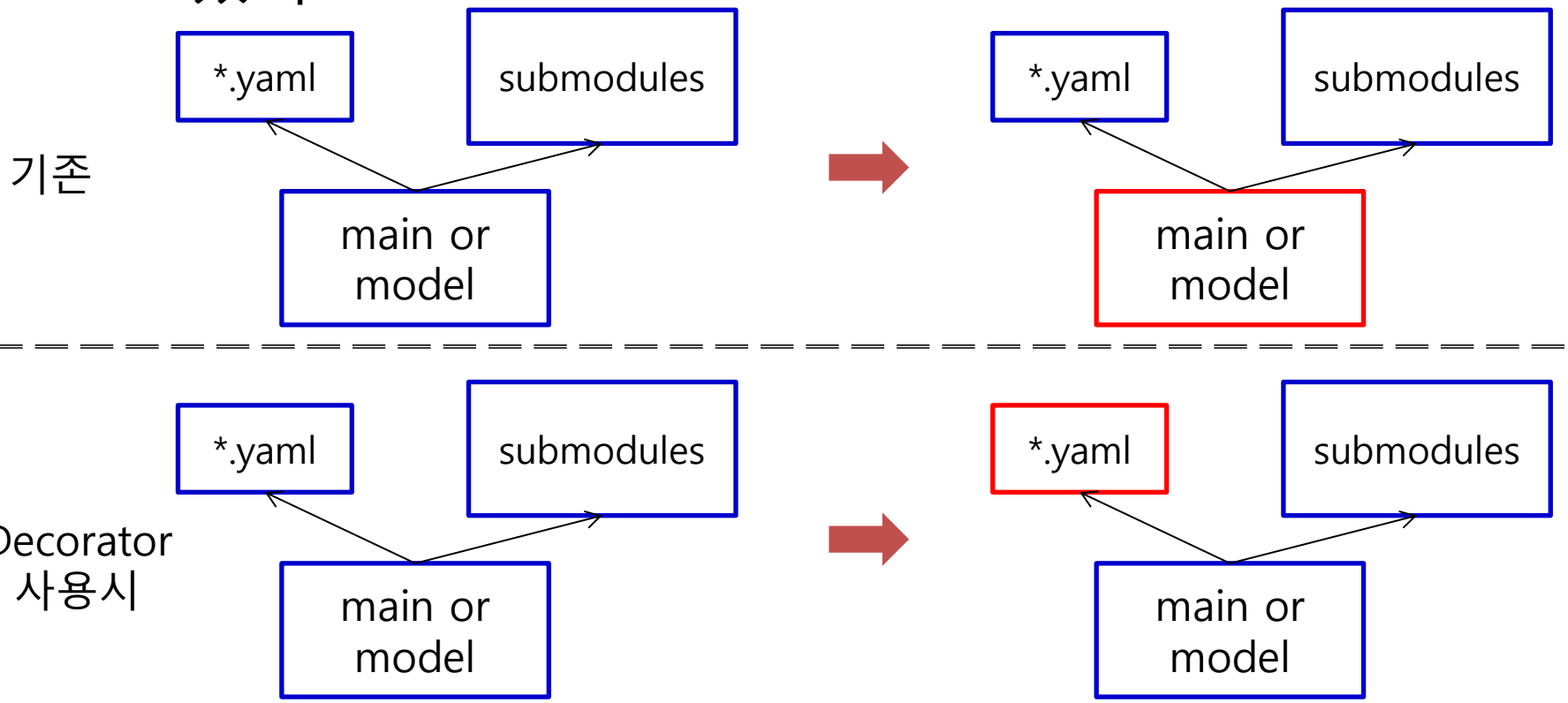
- Decorator 사용시
  - 코드 수정 x

```
class MyModel(Model):
    def __init__(self, cfg):
        super(MyModel, self).__init__()
        self.pre_layer = get_layer(cfg['prelayer'], cfg)

        self.d1 = Dense(128, activation=cfg['activation'])
        self.d2 = Dense(10)
```

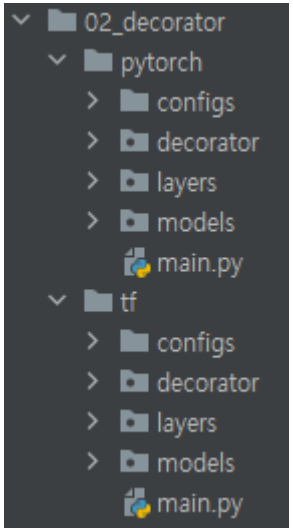
02. Decorator

- 무슨 장점이 있나요?
  - 즉, 기존 코드의 수정 없이 YAML만 변경할 수 있다



02. Decorator

- Hierarchy
  - configs
    - *.yaml이 들어있는 폴더
  - decorator
    - 모듈을 손쉽게 가져오도록 하는 클래스
  - layers
    - Dense 모듈과 Conv 모듈이 들어있는 폴더
    - Decorator를 이용하여 가져올 모듈
  - models
    - MyModel



02. Decorator

- configs/dense.yaml과 conv.yaml
  - 각각 DenseLayer클래스와 ConvLayer 클래스를 가져오도록 하는 yaml입니다.

```yaml
hyper_parameters:
  batch_size : 32
  epochs : 5
  learning_rate : 0.001

network_parameters:
  activation : 'relu'
  prelayer : "DenseLayer"

data_scale_factor : 2

dataset_root : '../../00_data'
```

dense.yaml

```yaml
hyper_parameters:
  batch_size : 32
  epochs : 5
  learning_rate : 0.001

network_parameters:
  activation : 'relu'
  prelayer : "ConvLayer"

data_scale_factor : 2

dataset_root : '../../00_data'
```

conv.yaml

02. Decorator

- MyModel
  - get_layer함수를 통하여, pre_layer를 정의할 수 있는 모델로 변경해 봅시다.

```tensorflow
from layers.build import get_layer

class MyModel(Model):
    def __init__(self, cfg):
        super(MyModel, self).__init__()
        self.pre_layer = get_layer(cfg['prelayer'], cfg)

        self.d1 = Dense(128, activation=cfg['activation'])
        self.d2 = Dense(10)

    def call(self, x):
        x = self.pre_layer(x)

        x = self.d1(x)
        return self.d2(x)
```

tensorflow

```pytorch
from layers.build import get_layer

class MyModel(nn.Module):
    def __init__(self, cfg):
        super(MyModel, self).__init__()
        self.pre_layer = get_layer(cfg['prelayer'], cfg)

        self.d1 = nn.Linear(self.pre_layer.outdim, 128)
        self.d2 = nn.Linear(128, 10)

        if cfg['activation'] == 'sigmoid':
            self.act = nn.Sigmoid()
        elif cfg['activation'] == 'relu':
            self.act = nn.ReLU()
        else:
            self.act = nn.Identity()

    def forward(self, x):
        x = self.pre_layer(x)
        x = self.act(self.d1(x))
        return self.d2(x)
```

pytorch

02. Decorator

- ClsDecorator
  - set()함수로 가져올 클래스들을 저장하고, get() 함수로 저장된 클래스를 가져옵니다.

```
class ClsDecorator():
    def __init__(self):
        self.dic = {}

    def set(self):
        def deco_fn(_cls):
            name = _cls.__name__
            self.dic[name] = _cls
            return _cls

        return deco_fn

    def get(self, name):
        return self.dic.get(name)
```

02. Decorator

- layers/build.py
  - ClsDecorator instance를 선언하고, get_layer() 함수를 정의합니다.
  - _my_decorator를 이용하여, 원하는 모듈을 가져오는 함수입니다.

```
from decorator.decorator import ClsDecorator

_my_decorator = ClsDecorator()

def get_layer(name, cfg):

    return _my_decorator.get(name)(cfg)
```

02. Decorator

- layers/dense_layer.py
  - 코드 빌드 시, python decorator를 이용하여 해당 클래스를 set 합니다.

```tensorflow
from .build import _my_decorator

@_my_decorator.set()
class DenseLayer(tf.keras.Model):
    def __init__(self, cfg=None):
        super(DenseLayer, self).__init__()
        self.flatten = Flatten()
        self.d0 = Dense(256, activation=cfg['activation'])

    def call(self,x):
        x = self.flatten(x)
        x = self.d0(x)
        return x
```

tensorflow

```pytorch
from .build import _my_decorator

@_my_decorator.set()
class DenseLayer(nn.Module):
    def __init__(self, cfg=None):
        super(DenseLayer, self).__init__()
        self.flatten = nn.Flatten()
        self.d0 = nn.Linear(28*28,256)

        if cfg['activation'] == 'sigmoid':
            self.act = nn.Sigmoid()
        elif cfg['activation'] == 'relu':
            self.act = nn.ReLU()
        else:
            self.act = nn.Identity()

        self.outdim = 256

    def forward(self,x):
        x = self.flatten(x)
        x = self.act(self.d0(x))
        return x
```

pytorch

02. Decorator

- layers/conv_layer.py
  - 코드 빌드 시, python decorator를 이용하여 해당 클래스를 set 합니다.

```tensorflow
from .build import _my_decorator

@_my_decorator.set()
class ConvLayer(tf.keras.Model):
    def __init__(self, cfg=None):
        super(ConvLayer, self).__init__()
        self.conv = Conv2D(32, 3, activation=cfg['activation'])
        self.flatten = Flatten()

    def call(self,x):
        x = self.conv(x)
        x = self.flatten(x)
        return x
```

tensorflow

```pytorch
from .build import _my_decorator

@_my_decorator.set()
class ConvLayer(nn.Module):
    def __init__(self, cfg=None):
        super(ConvLayer, self).__init__()
        self.conv = nn.Conv2d(1, 32, 3, padding=1)
        self.flatten = nn.Flatten()

        if cfg['activation'] =='sigmoid':
            self.act = nn.Sigmoid()
        elif cfg['activation'] =='relu':
            self.act = nn.ReLU()
        else:
            self.act = nn.Identity()

        self.outdim = 28*28*32

    def forward(self,x):
        x = self.act(self.conv(x))
        x = self.flatten(x)
        return x
```

pytorch

02. Decorator

- layers/__init__.py
  - 구현한 모듈을 ClsDecorator에 등록하도록 import해줍니다.

```
from .build import _my_decorator, get_layer

from .dense_layer import DenseLayer
from .conv_layer import ConvLayer
```

02. Decorator

- 나머지 코드는 지난번 01.YAML과 거의 변화가 없습니다.
- main에서 model 생성자 인자로 config만 바꿔주면 됩니다.

```
model = MyNetwork.MyModel(config['network_parameters'])
```

02. Decorator

- 실행 결과 예시
  - python main.py --config ./configs/dense.yaml

```
MyModel(
  (pre_layer): DenseLayer(
    (flatten): Flatten()
    (d0): Linear(in_features=784, out_features=256, bias=True)
    (act): ReLU()
  )
  (d1): Linear(in_features=256, out_features=128, bias=True)
  (d2): Linear(in_features=128, out_features=10, bias=True)
  (act): ReLU()
)
```

  - python main.py --config ./configs/conv.yaml

```
MyModel(
  (pre_layer): ConvLayer(
    (conv): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (flatten): Flatten()
    (act): ReLU()
  )
  (d1): Linear(in_features=25088, out_features=128, bias=True)
  (d2): Linear(in_features=128, out_features=10, bias=True)
  (act): ReLU()
)
```

02. Decorator

- 이제 새로운 모듈을 추가 할 때, layers/__init__.py에 decorator가 set하도록 추가 import만 해주면 됩니다.