DesneNet Pytorch code 뜯어보기

Reference

- https://github.com/pytorch/vision/blob/master/torchvision/models/densenet.py

목표: pytorch에 구현된 densenet 모델을 뜯어보자

코드 전부를 뜯어보지는 않고 핵심이 되는 bottle neck, dense layer, dense block, transition layer 등과 이를 어떻게 쌓는지 살펴볼 것이다.

1. _DenseLayer class

```
class _DenseLayer(nn.Module):
    def __init__(self, num_input_features, growth_rate, bn_size, drop_rate,
memory_efficient=False):
        super(_DenseLayer, self).__init__()
        self.add_module('norm1', nn.BatchNorm2d(num_input_features)),
        self.add_module('relu1', nn.ReLU(inplace=True)),
        self.add_module('conv1', nn.Conv2d(num_input_features, bn_size *
                                           growth_rate, kernel_size=1, stride=1,
                                           bias=False)),
        self.add_module('norm2', nn.BatchNorm2d(bn_size * growth_rate)),
        self.add_module('relu2', nn.ReLU(inplace=True)),
        self.add_module('conv2', nn.Conv2d(bn_size * growth_rate, growth_rate,
                                           kernel_size=3, stride=1, padding=1,
                                           bias=False)),
        self.drop_rate = float(drop_rate)
        self.memory_efficient = memory_efficient

    def bn_function(self, inputs):
        # type: (List[Tensor]) -> Tensor
        concated_features = torch.cat(inputs, 1)
        bottleneck_output =
self.conv1(self.relu1(self.norm1(concated_features)))  # noqa: T484
        return bottleneck_output
```

`__init__`에서는 모델에 필요한 Conv, BN, ReLU를 정의한다. 주목해야할 점은 conv1, conv2를 정의할 때 growth_rate를 곱한다는 점이다. conv1에서는 output_channels에 growth_rate를 곱하고 conv2에서는 input_channels에 growth_rate를 곱한다. 여기서 정의하는 conv1, conv2는 이후에 bottleneck에 사용할 것이므로 growth_rate와 함께 저의된다.
`bn_function`은 bottleneck 함수이다. 이 함수의 핵심은 `concated_features = torch.cat(inputs, 1)`이다. 이 명령어를 통해 l번째 layer의 input으로 받은 $l-1$번째 layer까지의 output을 모두 concat한다. 그리고 BN-ReLU-Conv(1×1)을 수행한다. 논문에서 소개된 bottleneck은 여기에 BN-ReLU-Conv(3×3)을 또 한다. 이 과정은 곧 바로 forward 함수에서 구현된다.

```
class _DenseLayer(nn.Module):
```

```
    def forward(self, input):  # noqa: F811
        if isinstance(input, Tensor):
            prev_features = [input]
        else:
            prev_features = input

        if self.memory_efficient and self.any_requires_grad(prev_features):
            if torch.jit.is_scripting():
                raise Exception("Memory Efficient not supported in JIT")

            bottleneck_output = self.call_checkpoint_bottleneck(prev_features)
        else:
            bottleneck_output = self.bn_function(prev_features)

        new_features = self.conv2(self.relu2(self.norm2(bottleneck_output)))
        if self.drop_rate > 0:
            new_features = F.dropout(new_features, p=self.drop_rate,
                                     training=self.training)
        return new_features
```

가장 눈여겨 봐야할 곳은 bottleneck을 정의하는 부분, 그리고 new_features을 통해서 bottleneck의 나머지 과정인 BN-ReLU-Conv(3×3)을 수행하는 부분이다.

요약하면 _DenseLayer는 모델을 만드는데 필요한 뼈대 모듈, 특히 bottleneck을 정의한다고 보면 될 것 같다.

2. _DenseBlock class

```
class _DenseBlock(nn.ModuleDict):
    _version = 2

    def __init__(self, num_layers, num_input_features, bn_size, growth_rate,
drop_rate, memory_efficient=False):
        super(_DenseBlock, self).__init__()
        for i in range(num_layers):
            layer = _DenseLayer(
                num_input_features + i * growth_rate,
                growth_rate=growth_rate,
                bn_size=bn_size,
                drop_rate=drop_rate,
                memory_efficient=memory_efficient,
            )
            self.add_module('denselayer%d' % (i + 1), layer)
```

`_DenseBlock`은 densenet에서 denseblock을 만드는 클래스이다. num_layers 인자를 살펴보자. num_layers는 후에 나올 `DenseNet` 클래스에서 [6, 12, 24, 16]을 받는다. 즉, 각 denseblock의 layer 갯수이다. 이는 논문에서 ImageNet에 사용된 DenseNet 아키텍쳐를 보면 알 수 있다.

Layers	Output Size	DenseNet-121($k=32$)	DenseNet-169($k=32$)	DenseNet-201($k=32$)	DenseNet-161($k=48$)
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	[1 × 1 conv, 3 × 3 conv] × 24	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48	[1 × 1 conv, 3 × 3 conv] × 36
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	[1 × 1 conv, 3 × 3 conv] × 16	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 24
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

즉, 각 denseblock에서 각각 [6, 12, 24, 16] 개의 bottleneck이 사용된 것이다.
따라서 for문을 돌며 각 dense block을 만드는데, 앞서 정의한 `_DenseLayer` 클래스를 사용한다. 그런데 여기에 `num_input_features + i * growth_rate`를 주목하자. num_input_features는 이전 dense block의 output features map의 갯수이고 growth_rate는 논문에서 언급되었다. 즉,

$k_0 + k \times (l-1)$을 코드로 구현한 것이다. i는 num_layers의 값이 아니라 0부터 len(num_layers)-1의 값을 갖는 것에 주목하자. 즉, $l-1$의 역할을 하는 것이다. 이와 같이 각 dense block을 만든 후에, `self.add_module`을 통해서 self 인자에 추가한다.

```
class _DenseBlock(nn.ModuleDict):
    def forward(self, init_features):
        features = [init_features]
        for name, layer in self.items():
            new_features = layer(features)
            features.append(new_features)
        return torch.cat(features, 1)
```

앞서 `__init__`에서 만든 layer module을 사용한다. for loop의 name, layer는 `__init__`에서 정의한 layer module인 것 같다 (정확히 파악은 못함) 이들 layer을 통해 얻은 new_features를 features에 추가하고 이를 torch.cat으로 합친다. 즉, 이 과정이 이전 layer의 output을 concat하는 과정인 것 같다.

3. _Transition class

```
class _Transition(nn.Sequential):
    def __init__(self, num_input_features, num_output_features):
        super(_Transition, self).__init__()
        self.add_module('norm', nn.BatchNorm2d(num_input_features))
        self.add_module('relu', nn.ReLU(inplace=True))
        self.add_module('conv', nn.Conv2d(num_input_features,
num_output_features,
                                          kernel_size=1, stride=1, bias=False))
        self.add_module('pool', nn.AvgPool2d(kernel_size=2, stride=2))
```

이름에서도 알 수 있듯이 dense block을 연결해주는 transition class이다.

4. DenseNet

```
class DenseNet(nn.Module):
    def __init__(self, growth_rate=32, block_config=(6, 12, 24, 16),
                 num_init_features=64, bn_size=4, drop_rate=0, num_classes=1000,
memory_efficient=False):
```

앞서 정의한 class을 이용하여 최종 DenseNet class을 정의한다. 우선 클래스의 인자부터 살펴보자. 클래스 인자의 기본 값은 모두 논문의 DenseNet-121 기준으로 설정되었다. block_config는 각 dense block마다 사용할 bottleneck의 갯수이다. num_init_features는 첫 번째 convolution layer에서 학습할 filters의 갯수이다. 논문에서 이를 $2k$로 설정한다고 나와있고 $k = 32$이므로 64로 설정되었다.

```
class DenseNet(nn.Module):
    def __init__():

        # Each denseblock
        num_features = num_init_features
        for i, num_layers in enumerate(block_config):
            block = _DenseBlock(
                num_layers=num_layers,
                num_input_features=num_features,
                bn_size=bn_size,
                growth_rate=growth_rate,
                drop_rate=drop_rate,
                memory_efficient=memory_efficient
            )

            self.features.add_module('denseblock%d' % (i + 1), block)
            num_features = num_features + num_layers * growth_rate
            if i != len(block_config) - 1:
                trans = _Transition(num_input_features=num_features,
                                    num_output_features=num_features // 2)
                self.features.add_module('transition%d' % (i + 1), trans)
                num_features = num_features // 2
```

이제 dense block을 만든다. num_features는 이전 layers의 output feature maps의 갯수이다. for loop을 시작하기 전에 num_init_features로 지정했는데, 이는 dense block을 만들기 이전의 first convolution의 output feature maps의 갯수가 num_init_features이기 때문이다. 이후에는 block_config에서 지정한 layers의 갯수만큼 dense block을 만든다.
for loop을 한 번 돌때마다 dense block, transition이 한 세트씩 만들어진다고 생각하자. block을 만든 후에는 .add_module을 통해 block 모듈을 추가한다.
다음으로 growth_rate을 통해 num_features을 다시 조정한다.
조정된 num_features를 transition의 input_features로 넣는다. transition 모듈을 통해 feature maps의 갯수를 반으로 줄인다.
이 과정을 block_config에 명시된 갯수만큼 반복한다.

```
class DenseNet(nn.Module):
    def __init__():

        # Final batch norm
        self.features.add_module('norm5', nn.BatchNorm2d(num_features))

        # Linear layer
        self.classifier = nn.Linear(num_features, num_classes)

    def forward(self, x):
```

```
        features = self.features(x)
        out = F.relu(features, inplace=True)
        out = F.adaptive_avg_pool2d(out, (1, 1))
        out = torch.flatten(out, 1)
        out = self.classifier(out)
        return out
```

이후에는 final batch norm, classifier을 정의한다.
forward에서는 4개의 DenseBlock과 3개의 Transition, 그리고 relu, pooling 및 classifier까지의 흐름을 정의한다.