[딥러닝 파이토치 교과서] 6장 합성곱 신경망2(6.1.3)

작성일시	@2024년 11월 12일 오후 5:41
분야	DL
주제	VGGNet
type	필사
날짜	@2024년 11월 12일

6.1.3 VGGNet

- 카렌 시모니안, 앤드류 지서만, <Very deep convolutional networks for large-scale image recognition(2025)>, ICLR
- 합성곱층 파라미터 수 감소, 훈련 시간 개선
  - VGG : 네트워크를 깊게 만드는 것이 성능에 어떤 영향을 미치는지 확인하고자 함.
    - 최대한 **깊이**의 영향만 확인하고자 합성곱층에서 사용하는 필터/커널의 크기를 가장 작은 3×3으로 고정
- 네트워크 계층의 총 개수에 따른 여러 유형의 VGGNet(VGG16, VGG19 등) 존재

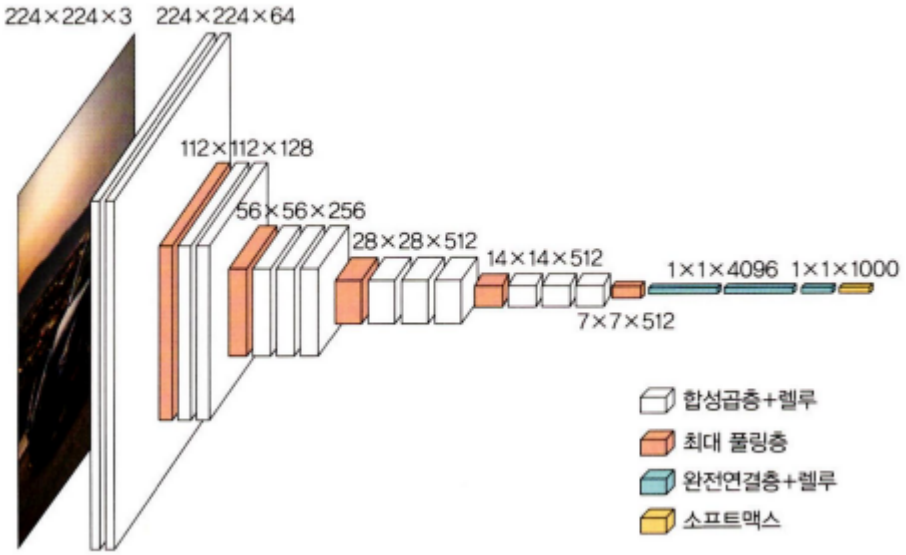
EX1) VGG16

- 파라미터 1억 3300만 개
- 합성곱 커널의 크기 3×3
- 최대 풀링 커널의 크기 2×2
- 스트라이드 2

➡️ 64개의 224×224 특성 맵($224 \times 224 \times 64$) 생성

- 마지막 16번째 계층을 제외하고 모두 ReLU 활성화 함수 적용

❤ 그림 6-13 VGG16 구조



❤ 표 6-3 VGG16 구조 상세

계층 유형	특성 맵	크기	커널 크기	스트라이드	활성화 함수
이미지	1	224×224	–	–	–
합성곱층	64	224×224	3×3	1	렐루(ReLU)
합성곱층	64	224×224	3×3	1	렐루(ReLU)
최대 풀링층	64	112×112	2×2	2	–
합성곱층	128	112×112	3×3	1	렐루(ReLU)
합성곱층	128	112×112	3×3	1	렐루(ReLU)
최대 풀링층	128	56×56	2×2	2	–
합성곱층	256	56×56	3×3	1	렐루(ReLU)
합성곱층	256	56×56	3×3	1	렐루(ReLU)
합성곱층	256	56×56	3×3	1	렐루(ReLU)
합성곱층	256	56×56	3×3	1	렐루(ReLU)
최대 풀링층	256	28×28	2×2	2	–
합성곱층	512	28×28	3×3	1	렐루(ReLU)
합성곱층	512	28×28	3×3	1	렐루(ReLU)
합성곱층	512	28×28	3×3	1	렐루(ReLU)
합성곱층	512	28×28	3×3	1	렐루(ReLU)
최대 풀링층	512	14×14	2×2	2	–

계층 유형	특성 맵	크기	커널 크기	스트라이드	활성화 함수
합성곱층	512	14×14	3×3	1	렐루(ReLU)
합성곱층	512	14×14	3×3	1	렐루(ReLU)
합성곱층	512	14×14	3×3	1	렐루(ReLU)
합성곱층	512	14×14	3×3	1	렐루(ReLU)
최대 풀링층	512	7×7	2×2	2	–
완전연결층	–	4096	–	–	렐루(ReLU)
완전연결층	–	4096	–	–	렐루(ReLU)
완전연결층	–	1000	–	–	소프트맥스(softmax)

EX 2) VGG11

- VGG16이나 VGG19를 구현하려면 VGG11에서 더 깊게 쌓아 올리면 된다.

1. 라이브러리 호출

```
import copy
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.utils.data as data
import torchvision
import torchvision.transforms as transforms
import torchvision.datasets as Datasets

device=torch.device('cuda' if torch.cuda.is_available() else
```

객체 복사

`import copy` : **객체 복사**를 위해 사용

1. 단순 객체 복사
2. 크게 얕은 복사(shallow copy)
3. 깊은 복사(deep copy)

1. 단순 객체 복사

```
original=[1, 2, 3]
copy_o=original
print(copy_o)

copy_o[2]=10
print(copy_o)
print(original)
```

✅ `copy_o`와 `original` 모두 바뀐다.

2. 얕은 복사

```
import copy

original=[[1, 2], 3]
copy_o=copy.copy(original)
print(copy_o)
copy_o[0]=100
print(copy_o)

append=copy.copy(original)
append[0].append(4)
print(append)
print(original)
```

✅ `copy_o`에서 `[1, 2]` 값을 `100`으로 변경했더니 `copy_o`만 바뀌었다.

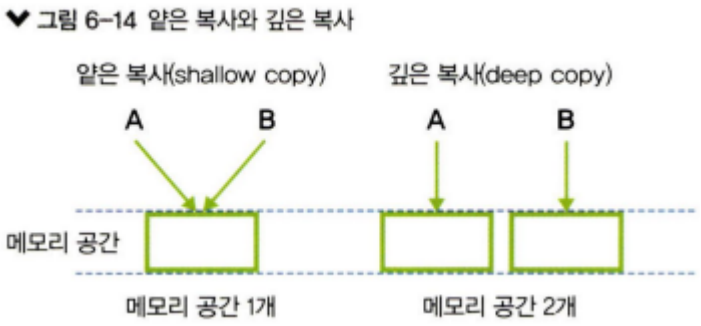
✅ `[1, 2]`에 `4`를 추가했더니 `original`과 `copy_o` 모두 바뀌었다.

3. 깊은 복사

```
import copy

original=[[1, 2], 3]
copy_o=copy.deepcopy(original)
print(copy_o)
copy_o[0]=100
print(copy_o)
```

```
print(original)

append=copy.deepcopy(original)
append[0].append(4)
print(append)
print(original)
```

✅ `copy_o`은 변경되었지만 `original`은 그대로이다.



2. VGG 모델 정의

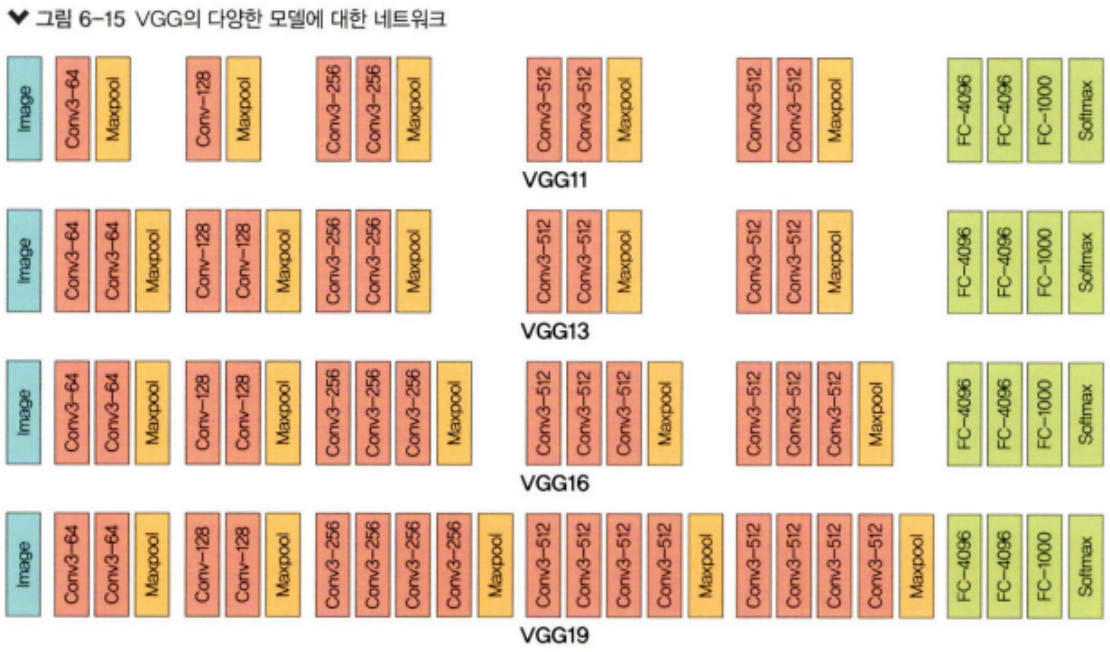
```
class VGG(nn.Module):
  def __init__(self, features, output_dim):
    super().__init__()
    self.features=features
    self.avgpool=nn.AdaptiveAvgPool2d(7)
    self.classifier=nn.Sequential(
        nn.Linear(512*7*7, 4096),
        nn.ReLU(inplace=True),
        nn.Dropout(0.5),
        nn.Linear(4096, 4096),
        nn.ReLU(inplace=True),

        ## 마지막 층
        nn.Dropout(0.5),
        nn.Linear(4096, output_dim)
    )

  def forward(self, x):
```

```
    x=self.features(x)
    x=self.avgpool(x)
    h=x.view(x.shape[0], -1)
    x=self.classifier(h)
    return x, h
```

VGG11, VGG13, VGG16, VGG19 모델의 계층 정의

- 숫자(output channel, 출력 채널)는 Conv2d를 수행하라는 의미
- 출력 채널(output channel)은 다음 계층의 입력 채널(input channel)
- ❓ `M` : 최대 풀링(max pooling)

```
vgg11_config=[64, 'M', 128, 'M', 256, 256, 'M', 512, 512, 'M'
vgg13_config=[64, 64, 'M', 128, 128, 'M', 256, 256, 'M', 512,
vgg16_config=[64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M',
vgg19_config=[64, 64, 'M', 128, 128, 'M', 256, 256, 256, 256,
```



VGG 계층 정의

```
def get_vgg_layers(config, batch_norm):
  layers=[]
  in_channels=3
```

```
  for c in config:
    assert c == 'M' or isinstance(c, int) ①
    if c=='M':
      layers += [nn.MaxPool2d(kernel_size=2)]
    else:
      conv2d=nn.Conv2d(in_channels, c, kernel_size=3, padding
      if batch_norm:
        layers += [conv2d, nn.BatchNorm2d(c), nn.ReLU(inplace
      else:
        layers += [conv2d, nn.ReLU(inpalce=True)]
      in_channels=c

  return nn.Sequential(*layers) # 네트워크의 모든 계층 반환
```

① 조건문 정의

`assert c == 'M'`

- `assert` : 가정 설정문. 뒤의 조건이 `True`가 아니면 에러 발생
  - `c == 'M'`이 아니면 오류 발생

`isinstance` : 주어진 조건이 `True`인지 판단.

➡️ `assert c == 'M' or isinstance(c, int)` : `c`가 `'M'`이 아니거나 `int`가 아니면 오류 발생

모델 계층 생성

`get_vgg_layers()` 함수

- 배치 정규화(batch normalization)에 대한 계층 추가

```
vgg11_layers=get_vgg_layers(vgg11_config, batch_norm=True) ①
```

① `batch_norm` : 데이터 평균을 0, 표준편차를 1로 분포시킴. (8장)

⚠️ 각 계층에서 입력 데이터의 분포는 앞 계층에서 업데이트된 가중치에 따라 변한다. 즉, 각 계층에서 변화되는 분포는 학습 속도를 늦추고, 학습도 어렵게 한다.

➡️ 각 계층의 입력에 대한 분산을 평균 0, 표준편차 1로 분포시킨다.

모델 계층 생성

```
vgg11_layers=get_vgg_layers(vgg11_config, batch_norm=True)
```

VGG11 계층 확인

```
Sequential(
  (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), paddi
  (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
  (2): ReLU(inplace=True)
  (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation
  (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), pad
  (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
  (6): ReLU(inplace=True)
  (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation
  (8): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), pa
  (9): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
  (10): ReLU(inplace=True)
  (11): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), p
  (12): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True
  (13): ReLU(inplace=True)
  (14): MaxPool2d(kernel_size=2, stride=2, padding=0, dilatio
  (15): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), p
  (16): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True
  (17): ReLU(inplace=True)
  (18): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), p
  (19): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True
  (20): ReLU(inplace=True)
  (21): MaxPool2d(kernel_size=2, stride=2, padding=0, dilatio
  (22): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), p
  (23): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True
  (24): ReLU(inplace=True)
  (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), p
  (26): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True
  (27): ReLU(inplace=True)
  (28): MaxPool2d(kernel_size=2, stride=2, padding=0, dilatio
)
```

➡️배치 정규화(`BatchNorm2d`) 추가 확인

VGG11 전체에 대한 네트워크 확인

```
OUTPUT_DIM=2
model=VGG(vgg11_layers, OUTPUT_DIM)
print(model)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), pad
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilati
    (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), pa
    (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=Tru
    (6): ReLU(inplace=True)
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilati
    (8): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1),
    (9): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=Tru
    (10): ReLU(inplace=True)
    (11): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
    (12): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=Tr
    (13): ReLU(inplace=True)
    (14): MaxPool2d(kernel_size=2, stride=2, padding=0, dilat
    (15): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1),
    (16): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=Tr
    (17): ReLU(inplace=True)
    (18): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
    (19): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=Tr
    (20): ReLU(inplace=True)
    (21): MaxPool2d(kernel_size=2, stride=2, padding=0, dilat
    (22): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
    (23): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=Tr
    (24): ReLU(inplace=True)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
    (26): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=Tr
    (27): ReLU(inplace=True)
    (28): MaxPool2d(kernel_size=2, stride=2, padding=0, dilat
  )
  (avgpool): AdaptiveAvgPool2d(output_size=7)
```

```
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=Tr
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=Tru
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=2, bias=True)
  )
)
```

➡️ `vgg11_layers`와 `VGG()`에서 정의했던 완전연결층과 출력층이 포함된 구성

VGG11 사전 훈련된 모델 사용

```
import torchvision.models as models

pretrained_model=models.vgg11_bn(pretrained=True)
print(pretrained_model)
```

① `pretrained_model=models.vgg11_bn(pretrained=True)`

`vgg_bn` : VGG11 기본 모델에 배치 정규화가 적용된 모델 사용

`pretrained`를 `True`로 설정 : 사전 훈련된 모델 사용(미리 학습된 파라미터 값 사용)

새로운 `config` 정의 : `My_Vgg`

```
vgg11_config=[64, 'M', 128, 'M', 256, 256, 'M', 512, 512, 'M'
vgg13_config=[64, 64, 'M', 128, 128, 'M', 256, 256, 'M', 512,
vgg16_config=[64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M',
vgg19_config=[64, 64, 'M', 128, 128, 'M', 256, 256, 256, 256,
My_Vgg=[64, 64, 64, 'M', 128, 128, 128, 'M', 256, 256, 256, '
```