ResNet

20176359 신수현

ResNet에서 알고자 하는 것

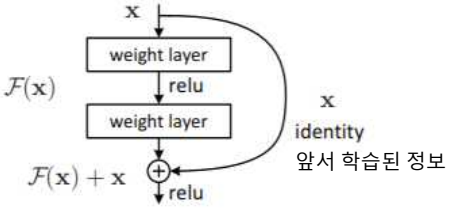
- 더 많은 레이어를 쌓은 것 만큼 Network의 성능이 좋아질까?
- Vanishing/exploding gradients 으로 인해 Degradation Problem 발생
- Degradation Probelm (Degradation : network 가 깊어질수록 accuracy가 떨어지는 현상) 해소하기 ➡ deep residual learning 제안

논문의 핵심 아이디어 : 잔여 블록 (Residual Block)

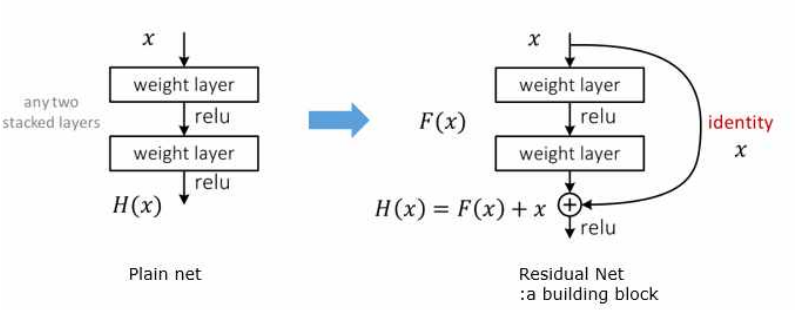
- Residual block을 이용해 네트워크의 최적화(optimization) 난이도를 낮춘다
- 실제 내재된 mapping H(x) 를 곧바로 학습하기 어려우므로 F(x) = H(x)−x를 대신 학습한다.

※ Building Block

x 는 input / Model 인 F(x) (잔여 정보) 라는 일련의 과정을 거치면서 자신(identity)인 x가 더해져서 output으로 F(x) + x 가 나오는 구조



Plain layers VS Residual Block

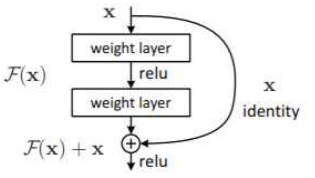


Plain layers에서는 weight layer 가 각각 분리되어, 가중치가 개별적으로 학습되어야한다. 따라서 난이도가 증가하고 층이 깊어질수록 심하게 발생하게된다.

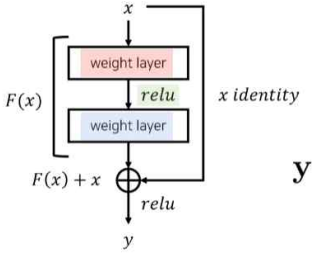
Residual Block 에서, 기존 학습 정보 x는 그대로 가져오고 잔여정보인 F(x)만 추가적으로 더해준다. 전체를 학습하는거보다 쉬워 학습이 더 빠르고 높은 성능을 보여준다.

Basic Block

```
class BasicBlock(nn.Module):
    def __init__(self, in_planes, planes, stride=1):
        super(BasicBlock, self).__init__()

        # 3x3 필터를 사용 (너비와 높이를 줄일 때는 stride 값 조절)
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes) # 배치 정규화(batch normalization)

        # 3x3 필터를 사용 (패딩을 1만큼 주기 때문에 너비와 높이가 동일)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes) # 배치 정규화(batch normalization)

        self.shortcut = nn.Sequential() # identity인 경우
        if stride != 1: # stride가 1이 아니라면, Identity mapping이 아닌 경우 (입력값 =/= 출력값 dim)
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, planes, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(planes)
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x) # (핵심) skip connection
        out = F.relu(out)
        return out
```



Residual Block



$$\mathcal{F} = W_2\sigma(W_1\mathbf{x})$$

일반적인 형태

$$\mathbf{y} = \underbrace{\mathcal{F}(\mathbf{x}, \{W_i\})}_{\text{multiple convolutional layers}} + \underbrace{W_s\mathbf{x}}_{\text{shortcut}}$$

- F(x) 함수를 오른쪽 식과 같이 정의했는데, 입력 x 가 들어왔을때 first weight W1을 곱한 후 activation 함수 즉 relu함수를 적용하고, second weight W2 값을 곱해준다.
- 마지막에 추가적으로 x가 더해진다.
- Weight값을 단순히 2번이 아니라 여러번 사용할 수 있다고 가정하고 (= multiple convolutional layers), 추가적으로 shortcut connection을 이용해 기존의 입력값을 그대로 가져와서 더해준다.
- 입력값의 차원과 output 차원이 동일시 identity mapping 적용 가능하다.
- 다르다면, linear 하게 projection 을 시켜서 mapping 가능하다고한다.

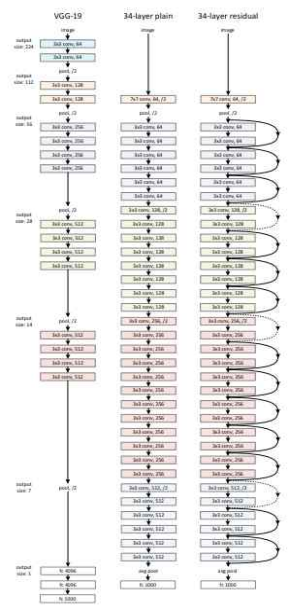
Residual Block 사용한 결과

	plain	ResNet
18 layers	27.94	27.88
34 layers	28.54	**25.03**

method	top-5 err. (**test**)
VGG [40] (ILSVRC'14)	7.32
GoogLeNet [43] (ILSVRC'14)	6.66
VGG [40] (v5)	6.8
PReLU-net [12]	4.94
BN-inception [16]	4.82
ResNet (ILSVRC'15)	**3.57**

method	top-1 err.	top-5 err.
VGG [40] (ILSVRC'14)	-	8.43†
GoogLeNet [43] (ILSVRC'14)	-	7.89
VGG [40] (v5)	24.4	7.1
PReLU-net [12]	21.59	5.71
BN-inception [16]	21.99	5.81
ResNet-34 B	21.84	5.71
ResNet-34 C	21.53	5.60
ResNet-50	20.74	5.25
ResNet-101	19.87	4.60
ResNet-152	**19.38**	**4.49**

Residual Block 을 사용하여 돌린 결과가 더 나은 것을 확인할 수 있다.

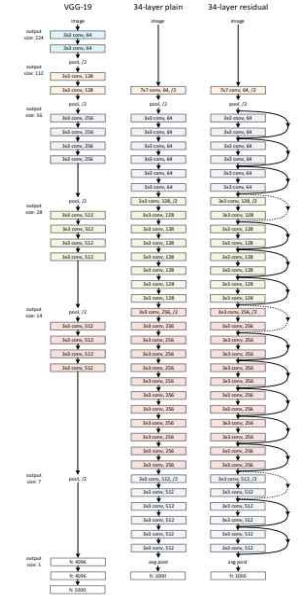


34-layer residual ※ FLOP의 감소 (계산 복잡도)

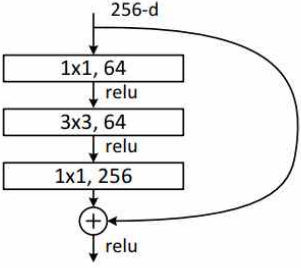
- 3x3 convolutional filter 사용
- 이 필터를 2개씩 묶어서 residual function 형태로 학습 진행
- 점선은 입력값과 출력값의 dim이 일치하지않아서 맞춰주는 short cut connection을 이용한 것
- 2개씩 묶는것을 3번 반복하고 크기를 바꿔서 4번 반복하고, 크기 바꿔서 6번 반복 그리고 마지막으로 크기 바꿔서 3번 반복한 것을 볼 수 있다.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

```python
# ResNet 클래스 정의
class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=10):
        super(ResNet, self).__init__()
        self.in_planes = 64

        # 64개의 3x3 필터(filter)를 사용
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False) #conv layer 1개
        self.bn1 = nn.BatchNorm2d(64)
        self.layer1 = self._make_layer(block, 64, num_blocks[0], stride=1) #basic block 당 conv layer 2개
                                                                           #그게 2개이므로 총 4개
        self.layer2 = self._make_layer(block, 128, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 256, num_blocks[2], stride=2)
        self.layer4 = self._make_layer(block, 512, num_blocks[3], stride=2)
        #총 16개의 conv layer

        self.linear = nn.Linear(512, num_classes)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1] * (num_blocks - 1)#첫번째 conv 연산에 의해서만 너비와 높이가 줄어들 수 있도록
                                                   #나머지는 stride = 1 로하여 같은 너비와 높이 유지
                                                #filter 개수가 증가할때마다 너비와 높이는 줄어들수 있게 설정
        layers = []
        for stride in strides:
            layers.append(block(self.in_planes, planes, stride))
            self.in_planes = planes # 다음 레이어를 위해 채널 수 변경
        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = F.avg_pool2d(out, 4)
        out = out.view(out.size(0), -1)
        out = self.linear(out)
        return out

# ResNet18 함수 정의
def ResNet18():
    return ResNet(BasicBlock, [2, 2, 2, 2])
```

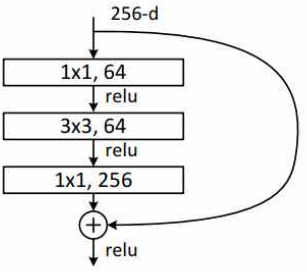


Bottle Neck



- 복잡도를 증가시키지 않기 위해 사용된 것
- 초반에 1x1 filter를 64개 사용하고 중간에는 3x3 filter 64개, 마지막에 1x1 filter 64개 사용
- 작은 커널을 사용함으로써 파라미터 수를 감소시킨다
- identity short cut 이 더욱 효과적
- 깊이가 50 이상인 Resnet 더욱더 좋은 성능을 보임

```python
    #layer가 50이상일때
class Bottleneck(nn.Module):
    expansion = 4

    def __init__(self, in_planes, planes, stride=1):
        super(Bottleneck, self).__init__()
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)

        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3,
                               stride=stride, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)

        self.conv3 = nn.Conv2d(planes, self.expansion *
                               planes, kernel_size=1, bias=False)
        self.bn3 = nn.BatchNorm2d(self.expansion*planes)

        self.shortcut = nn.Sequential()

        if stride != 1 or in_planes != self.expansion*planes:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, self.expansion*planes,
                          kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(self.expansion*planes)
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = F.relu(self.bn2(self.conv2(out)))
        out = self.bn3(self.conv3(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out
```



Implementation

※ 입력값 출력값 서로 다른 차원?

(A) 사이드에 zero padding 한 후 dim늘린 후 identity mapping 사용
(B) projection 연산을 활용한 short cut connection 이용 (dim이 증가할때만)
(C)모든 조건에 대해 항상 projection 사용 (필수일만큼 높은 개선은 아님!)

※ 구현

- random crop (224x224), horizontal flip 사용
- 매 conv layer 거칠때마다 batch 정규화 이용
- learning rate 점진적으로 줄여나가기
- weight decay 0.0001
- momentum 0.9
- Dropout 사용하지 않음

```
#learning rate를 바꾸기
def adjust_learning_rate(optimizer, epoch):
    lr = learning_rate
    if epoch >= 100:
        lr /= 10 #1/10 로 줄이기
    if epoch >= 150:
        lr /= 10  #1/10 로 줄이기
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr
```

```
learning_rate = 0.1
file_name = 'resnet18_cifar10.pt'

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=learning_rate, momentum=0.9, weight_decay=0.0001)
```

ResNet with CIFAR-10

- 입력 이미지 크기가 작은 cifar10 에 맞게 파라미터 수를 줄여서 별도의 Resnet 을 사용
- 파라미터 수는 더 적지만 성능은 좋은 것을 볼 수 있다.

method			error (%)
Maxout [9]			9.38
NIN [25]			8.81
DSN [24]			8.22
	# layers	# params	
FitNet [34]	19	2.5M	8.39
Highway [41, 42]	19	2.3M	7.54 (7.72±0.16)
Highway [41, 42]	32	1.25M	8.80
ResNet	20	0.27M	8.75
ResNet	32	0.46M	7.51
ResNet	44	0.66M	7.17
ResNet	56	0.85M	6.97
ResNet	110	1.7M	**6.43** (6.61±0.16)
ResNet	1202	19.4M	7.93

(middle/right). The network inputs are 32×32 images, with the per-pixel mean subtracted. The first layer is 3×3 convolutions. Then we use a stack of $6n$ layers with 3×3 convolutions on the feature maps of sizes $\{32, 16, 8\}$ respectively, with $2n$ layers for each feature map size. The numbers of filters are $\{16, 32, 64\}$ respectively. The subsampling is performed by convolutions with a stride of 2. The network ends with a global average pooling, a 10-way fully-connected layer, and softmax. There are totally $6n$+2 stacked weighted layers. The following table summarizes the architecture:

output map size	32×32	16×16	8×8
# layers	1+2n	2n	2n
# filters	16	32	64

참고 자료

- ndb796 github, “vgg.py 코드 & 논문 리뷰 참고”, https://github.com/ndb796/Deep-Learning-Paper-Review-and-Practice/tree/master/code_practices (2021.10.1)
- kuangliu github, “vgg.py, main.py 코드 참고”, https://github.com/kuangliu/pytorch-cifar, (2021.9.25)