ResNet 핵심 개념

ResNet은 2015년 Kaiming He 등이 발표한 **"Deep Residual Learning for Image Recognition"** 논문에서 아래 문제를 해결하기 위해 제안되었다.

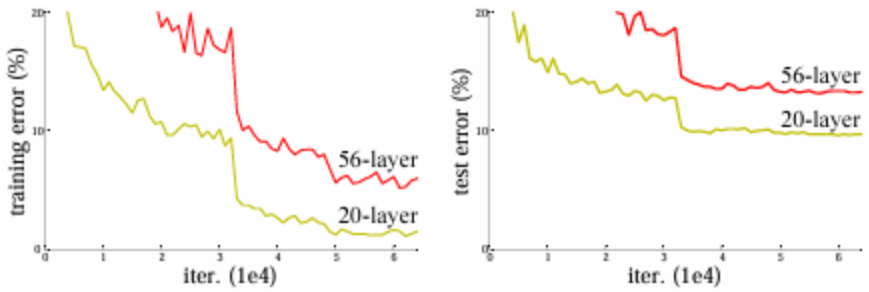
딥러닝에서 깊이 증가할 때의 문제

- 네트워크를 깊게 만들면 이론상 성능이 좋아야 하지만, 실제로는 **훈련 오차(training error)가 더 커지는 현상**이 발생한다.
- 이 현상을 **degradation problem**이라 부른다.
- 이는 **vanishing gradient** 문제와는 다르며, BN 등으로 기울기 문제는 완화했음에도 불구하고 발생한다.
- **해결책** : 네트워크가 직접 $H(x)$를 학습하는 대신, $H(x) - x$를 학습하도록함.
  - 즉, 입력 x에 대해 원하는 출력 $H(x)$를 그대로 학습하는 대신, **잔차(Residual)** $F(x) = H(x) - x$를 학습하고, 최종 출력은 $F(x) + x$로 만든다.

> degradation problem이 정확히 무엇이고, 잔차 학습으로 어떻게 해결이 되는가?

degradation problem : 네트워크 깊이를 늘리면 일반적으로 더 복잡한 패턴을 학습 할 수 있어 성능이 좋아져야 하는데, 실제로는 깊이가 깊어질수록 훈련 오류(training error)와 테스트 오류(test error)가 오히려 증가하는 현상을 의미함

- 이 문제는 overfitting 때문이 아님(테스트 에러 뿐 아니라 훈련 에러도 증가하기 때문)
- 즉, 모델 용량이 커졌음에도 최적화가 더 어려워짐을 의미
- 아래 그림은, ResNet 논문 CIFAR10 실험 결과 그래프 : 20-layer plain network는 잘 학습되지만, 56-layer plain network는 훈련 에러 조차 높아짐을 확인 가능



발생 이유

- 깊은 네트워크는 **identity mapping**(즉, 입력을 그대로 전달하는 단순한 매핑)조차 쉽게 학습하지 못한다.
- 깊이를 늘릴수록 연속된 비선형 변환을 통해 출력이 왜곡되고, 최적화 경로가 복잡해진다.(SGD가 좋은 해를 찾기 어려워짐)

- 단순히 초기화나 BN으로 해결되지 않는, 구조적 문제가 발생

Residual Learning으로 해결하는 방식

- 기존 방식 : $H(x)$을 직접 학습
- ResNet 방식

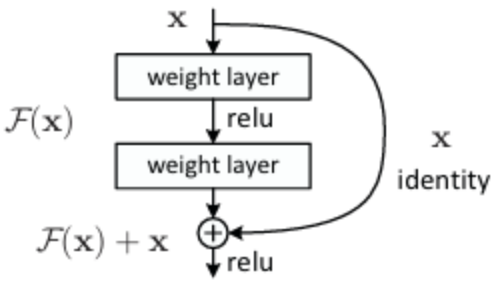


Figure 2. Residual learning: a building block.

- 네트워크가 학습하는 대상 함수를 재정의

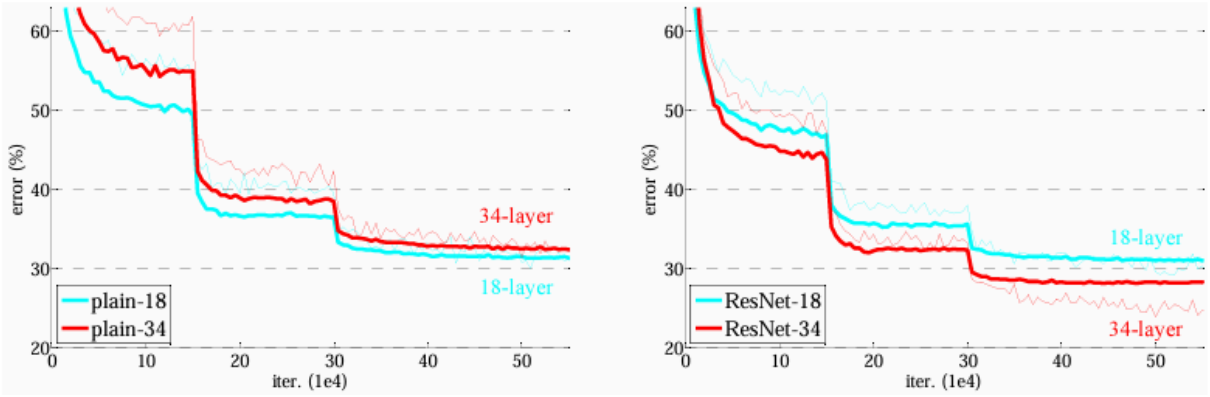
$$H(x) \text{ 대신 } F(x) = H(x) - x \text{를학습}$$

- 최종 출력

$$y = F(x) + x$$

- 즉, 네트워크는 **입력에서 얼마나 변화(잔차)를 줄 것인가**만 학습하면 된다.

효과



- 훈련 오류 감소 : 깊은 네트워크도 얕은 네트워크보다 항상 더 잘 학습이된다.
- 일반화 향상 : 깊이 증가에 따라 정확도도 꾸준히 향상

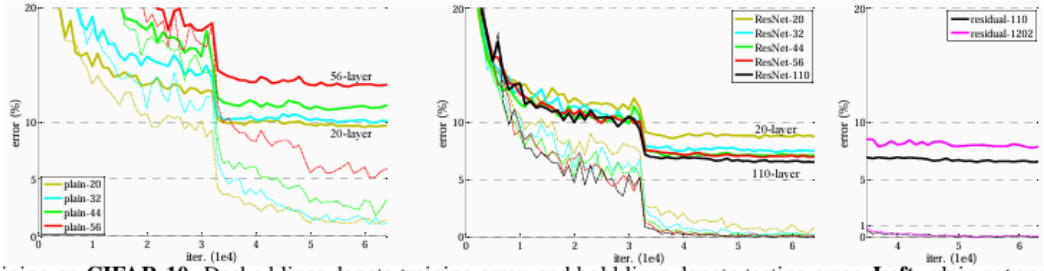


Figure 6. Training on **CIFAR-10**. Dashed lines denote training error, and bold lines denote testing error. **Left**: plain networks. The error of plain-110 is higher than 60% and not displayed. **Middle**: ResNets. **Right**: ResNets with 110 and 1202 layers.

- 위 그림은 각 레이어의 출력 값의 표준편차(std)를 보여준다.
- ResNet의 반응 크기는 일반적인 plain 네트워크보다 작다 즉, Residual function이 일반 함수보다 0에 더 가깝다는 논문 가설.
- 더 깊은 ResNet이 얕은 ResNet보다 반응 크기가 더 작다 : 깊어질수록 개별 레이어가 입력 신호를 조금씩만 수정한다는 의미
- 즉, Residual 학습은 신호를 크게 바꾸지 않고 작은 변화를 누적하며 깊이를 늘릴 수 있도록 함

1000 층 이상의 네트워크 실험
결과 : 최적화 문제 없음, 훈련 에러 0.1% 미만, 테스트 에러 7.93%
문제점

- ResNet-110보다 테스트 성능 낮음(Overfitting 발생)
- CIFAR-10 데이터 셋이 너무 작아 1202층 모델은 19.4M 파라미터로 과대적합
- 과도한 깊이로 모델 크기가 커져 불필요한 복잡성이 증가

Residual Block의 구조

기존 CNN 블록 : Conv $\rightarrow$ BN $\rightarrow$ ReLU$\rightarrow$Conv$\rightarrow$BN$\rightarrow$Add$\rightarrow$ReLU
여기서, $F(x) + x$ 구현을 위해 shorcut connection을 추가

입력과 출력의 크기가 다를 경우 : **Projection shortcut** 1x1 Conv로 차원을 맞춘다.

Projection Shorcut

$$y = F(x) + x$$

- 여기서 $F(x)$는 블록의 Conv 연산 결과, x는 입력
- 두 항의 Shape(차원)이 동일해야 element-wise addition이 가능하다.
- **문제점**
- Stage변경 시 stride=2로 다운 샘플링을 하면 feature map 크기가 절반으로 줄어든다.
- 채널수(out_channels)가 stage별로 증가(예: 64 $\rightarrow$ 128 $\rightarrow$ 256) 따라서 입력 x와 출력 F(x)의 크기, 채널이 불일치하는 문제가 발생한다.

이 문제를 해결하기 위해 입력 x를 1x1 Convolution으로 변환해서 출력 크기와 동일하게 맞춘다.

$$x' = W_s * x$$

$$y = F(x) + x'$$

여기서 W_s는 1x1 Conv의 weight

1x1 Convolution 사용 이유

1. 채널 수 조정
   - 1x1 Conv는 feature map의 공간적 크기를 유지하면서 채널 수를 바꿀 수 있다.
   - 예를 들어 입력 채널 64, 출력 채널 128로 바꿀때, 1x1 Conv는 가장 간단하고 효율적이다.
2. 연산 효율
   - 커널이 1x1이므로 연산량이 매우 적다

$$연산량 = H \times W \times C_{in} \times C_{out}$$

   - 공간 크기(H x W)는 유지하면서, stride로 다운 샘플링이 가능하다.
3. 정보 손실 최소화
   - 1x1 Conv는 각 픽셀 위치에서 채널 간 선형 조합을 수행한다.
   - 공간 구조를 바꾸지 않고 차원 정렬만 수행하므로 shortcut 연결에 적합하다.
4. 다른 대안 보다 단순
   - Zero-padding은 차원은 맞춰주지만, 학습 가능한 가중치가 없어 표현력이 부족하다.
   - 3x3 conv는 가능하지만, 연산량 증가로 비효율적이다.

ResNet-34 아키텍쳐

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$
	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

- conv1: 7×7, 64 filters, stride 2 → MaxPool(3×3, stride 2)
- conv2_x: BasicBlock × 3
- conv3_x: BasicBlock × 4

- conv4_x: BasicBlock × 6
- conv5_x: BasicBlock × 3
- Global AvgPool → FC(1000)

Pytorch 구현 코드

BasicBlock

```
class BasicBlock(nn.Module)
    expansion = 1
```

- ResNet-34에서 사용하는 기본 블록
- expansion: 채널 확장 비율, Bottleneck(ResNet-50 이상)은 4, BasicBlock은 1
- BasicBlock은 ResNet의 기본 형태로 ResNet-18, ResNet-34에서 사용된다.(위 표 이미지 참고)
- ResNet-18, ResNet-34에서는 두 개의 3x3 Conv만 사용 > 채널 수 변화 없음
- ResNet-50이상 > Bottleneck Block , 세개의 conv으로 구성
- 1×1 Conv (채널 축소) → 3×3 Conv → 1×1 Conv (채널 확장)
- 마지막 1x1 Conv에서 채널을 4배로 늘림(expansion = 4)
- 깊이를 늘리면서 연산량을 줄이기 위해 병목 구조 이용
  Bottleneck
  ResNet-50 이상에서 깊이를 늘리려면 파라미터와 연산량 폭발을 피해야한다.
  예를 들어, 3x3 Conv 연산량은

$$연산량 \propto (k^2) \times C_{in} \times C_{out} \times H \times W$$

- k = 커널 크기, H x W = feature map 크기
- 채널 수가 크면 연산량이 기하급수적으로 커짐
- 해결책 : 3x3 Conv 전에 채널 수를 줄이고, 끝에서만 다시 늘림(bottleneck)
- 즉, 첫번째 1x1 Conv에서 채널 수를 1/4로 줄이면 3x3 Conv는 작은 채널 크기에서 수행(연산량이 크게 절감)
- 마지막 1x1 Conv에서 다시 확장(expansion=4)
  plain 구조
  입력 : 256채널
  연산량 : $3 \times 3 \times 256 \times 256 = 589824$

Bottleneck 구조
입력 : 256 (1x1 Conv > 64 > 3x3 Conv > 64 > 1x1 Conv > 256)
연산량:

$$(1 \times 1 \times\ 256 \times 64) + (3 \times 3 \times\ 64 \times 64) + (1 \times 1 \times 64 \times 256) = 98304$$

$$\frac{589824}{98304} \approx 6$$

최종 연산량 약 6배 절감 가능

```
self.conv1 = nn.Conv2d(in_channels,
                       out_channels,
                       kernel_size = 3
                       stride=stride,
                       padding=1,
                       bias=False)
self.bn1 = nn.BatchNorm2d(out_channels)
self.relu = nn.ReLU(inplace=True)
```

- 첫 번째 Conv, stride로 다운 샘플링 가능

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

- 두 번째 Conv, 크기 유지

```
self.downsample = downsample
```

- 입력과 출력 크기가 다를 때 shortcut을 위해 사용

```
def forward(self, x):
        identity = x
        if self.downsample is not None:
            identity = self.downsample(x)

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)

        out += identity
        out = self.relu(out)
```

```
        return out
```

- $F(x) + x$ 구현
- 다운샘플링이 필요하면 identity를 변환
- identity?

ResNet34 클래스

```
self.conv1 = nn.Conv2d(3,
                       64,
                       kernel_size=7,
                       stride=2,
                       padding=3,
                       bias=False)
```

- ImageNet기준 구조 유지(CIFAR-10에선 불필요하게 크기가 축소되어 뒤에서 수정)

```
self.layer1 = self._make_layer(block, 64, layers[0])
self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
```

- 4개의 stage [3, 4, 6, 3] 블록 쌓음

```
self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
self.fc = nn.Linear(512 * block.expansion, num_classes)
```

- Global Average Pooling → FC

_make_layer()

```
if stride != 1 or self.in_channels != out_channels * block.expansion:
    downsample = nn.Sequential(
        nn.Conv2d(self.in_channels, out_channels * block.expansion,
                  kernel_size=1, stride=stride, bias=False),
        nn.BatchNorm2d(out_channels * block.expansion)
    )
```

- 크기나 채널이 다르면 projection shortcut 적용

CIFAR-10에 맞게 수정된 코드

논문에서 CIFAR-10 실험 시, 다음과 같이 구조를 변경:

- **첫 Conv**: 3×3 kernel, stride=1 → CIFAR는 32×32이므로 초기 다운샘플링 불필요.
- **MaxPool 제거**: 크기가 너무 작아지는 것을 방지.
- Feature Map 크기 변화: {32→16→8} 구조 사용.
- 필터 수: {16, 32, 64}로 설정 (원본은 {64, 128, 256, 512}).
- Shortcut은 모두 **identity** (채널 다를 때 projection만 사용).
  기존 ResNet

```
Conv(7×7, stride=2, channels=64)
MaxPool(3×3, stride=2)
```

CIFAR-10 맞춤형(모델 구조 자체를 변경함)

```
Conv(3×3, stride=1, channels=16)
#MaxPool(3×3, stride=2) 제거
```

> 첫 Conv를 3×3으로 바꾸고, MaxPool 제거하면 CIFAR-10 정확도 향상 예상

```
class BasicBlock(nn.Module):
    expansion = 1
    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,
                               stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,
                               stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = downsample

    def forward(self, x):
        identity = x
        if self.downsample is not None:
            identity = self.downsample(x)

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
```

```
        out += identity
        out = self.relu(out)

        return out

class ResNet34(nn.Module):
    def __init__(self, block, layers, num_classes=10):
        super(ResNet34, self).__init__()
        self.in_channels = 16 # 채널 수 줄임
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1,
bias=False)
        #self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.bn1 = nn.BatchNorm2d(16)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(block, 16, layers[0])
        self.layer2 = self._make_layer(block, 32, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 64, layers[2], stride=2)
        self.layer4 = self._make_layer(block, 128, layers[3], stride=2)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(128 * block.expansion, num_classes)

        # 가중치 초기화 He 초기화
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out',
nonlinearity='relu')
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

    def _make_layer(self, block, out_channels, blocks, stride=1):
        downsample = None
        if stride != 1 or self.in_channels != out_channels * block.expansion:
            downsample = nn.Sequential(
                nn.Conv2d(self.in_channels, out_channels * block.expansion,
                          kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels * block.expansion)

            )
        layers = []
        layers.append(block(self.in_channels, out_channels, stride,
downsample))

        self.in_channels = out_channels * block.expansion
```

```
        for _ in range(1, blocks):
            layers.append(block(self.in_channels, out_channels))

        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        # x = self.maxpool(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)
        return x
```

Test Result

항목	원래 ResNet 논문 구조	CIFAR-10 맞춤 구조
첫 Conv	7×7, stride=2	3×3, stride=1
MaxPool	있음 (3×3, stride=2)	없음
Stage 채널 변화	64 → 128 → 256 → 512	16 → 32 → 64 → 128
Test Loss	0.7190	**0.5886**
Test Acc	75.19%	**79.85%**