

# ResNet

## 0. Abstract

1. 깊은 학습이 어려워짐  $\Rightarrow$  전자 학습:  $F(x) = H(x) - \gamma$

$$\rightarrow \text{결과: } y = F(x) + \gamma$$

2. 계산 복잡도 ↓

## 1. Introduction

1. 이전: CNN으로 차·중·고 수준 특징을 end-to-end로 통합하여 depth 키워감

$\rightarrow$  gradient vanishing, exploding (학습 초기 단계에서 수렴이 X)

$\hookrightarrow$  정규화된 가중치 초기화, Batch Normalization 등으로 해결

$\rightarrow$  "degradation" 문제: 깊이↑ 성능 평화 후 악화

$\hookrightarrow$  training error↑ (교차오류 X)

$\hookrightarrow$  깊은 구조는 얕은 구조 복사 + 추가 층

$\hookrightarrow$  항등 매핑 (identity mapping) 인

해를 구성한다면 얕은 모델보다는 성능 좋아야 함

$\rightarrow$  but, optimizer가 그 항등매핑 해에 도달X ( $H(x)$  직접 최적화 힘들)

$\Rightarrow$  해결하기 위한 "전자 매핑" 학습

$\hookrightarrow$  underlying mapping (underlying mapping) 을 예측

$$= H(x)$$

$\Rightarrow$  대신: 전자  $F(x) := H(x) - \gamma$  를 학습 후 원래 매핑  $H(x)$  는  $F(x) + \gamma$ 로 대체 / 전자  $F(x) \approx 0$

$\hookrightarrow H(x) = h_L \circ \dots \circ h_1(x)$  (각 층 결과 연쇄합성; 즉, 차례대로 통과하는 학습)

$\rightarrow H(x) = \gamma$ 라는 항등 매핑 문제의 해를 직접 구하지 X

$\rightarrow H(x) - \gamma = F(x)$  를 두고  $F(x)$ 를 학습 (out( $H(x)$ ) = out( $x$ ) + identity( $F(x)$ ))

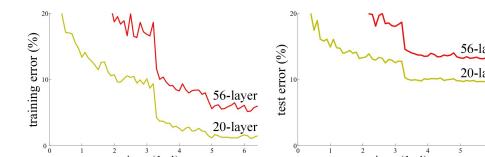


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

## 2. "잔차구조"

= shortCut connection (ffn)  
= skip  
= identity mapping

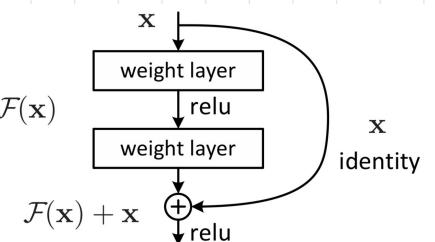


Figure 2. Residual learning: a building block.

## 2. Related Work

### 1. residual net vs highway net

residual net: x  
highway net:  $H(x)$   
Projection skip:  $\pi(x)$

## 3. Deep Residual Learning

### 3.1 Residual Learning

$$1. F(x) := H(x) - x$$

→ x: 입력 x의 첫 번째 layer 값

→ x와  $F(x)$ 의 차이가 작아지도록 하면

→ 학습 속도 증가(증명):  $H(x) = x + F(x) \approx x$

↓

"가중치를 0으로 몰자": 초기화가 더 쉬워짐 (CIFAR-10)

↳ 사전 조건화 (pre-conditioning) 효과

: 문제를 해결하기 쉬운 형태로 학습을 촉진

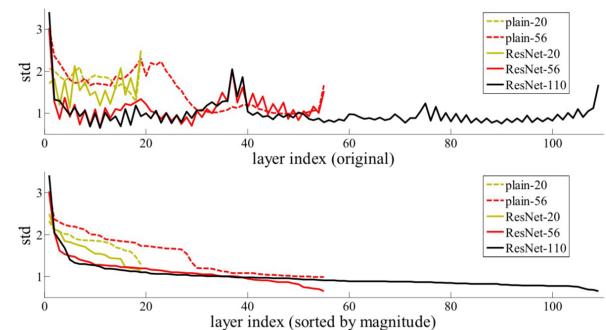


Figure 7. Standard deviations (std) of layer responses on CIFAR-10. The responses are the outputs of each  $3 \times 3$  layer, after BN and before nonlinearity. **Top:** the layers are shown in their original order. **Bottom:** the responses are ranked in descending order.

→ std가 plain net vs ResNet의 차이

: 작은 std = 작은  $f(x) = 작은 \Delta x$   
( $x$ 는 크다 | 원소가 많다)

## 3.2 Identity mapping by shortcuts

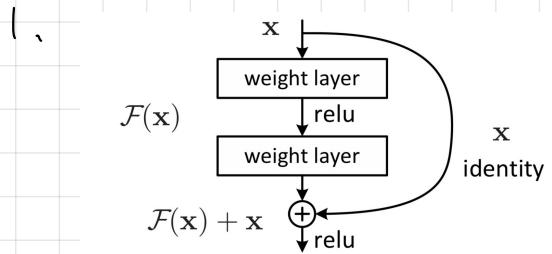


Figure 2. Residual learning: a building block.

$$y = \mathcal{F}(x, \{W_i\}) + x$$

한국어 해석  
(H(x) 학습되는 것  
x와 더해짐)

shortcut connection (+ 노드: 학습되는 x, 학습되는 x)

```
import torch
import torch.nn as nn

def conv3x3(in_c, out_c, stride=1):
    return nn.Conv2d(in_c, out_c, 3, stride=stride, padding=1, bias=False)

def conv1x1(in_c, out_c, stride=1):
    return nn.Conv2d(in_c, out_c, 1, stride=stride, bias=False)

# 1) Plain: H(x)만 학습 (초기 없음)
class PlainBlock(nn.Module):
    def __init__(self, in_c, out_c, stride=1):
        super().__init__()
        self.net = nn.Sequential(
            conv3x3(in_c, out_c, stride),
            nn.BatchNorm2d(out_c),
            nn.ReLU(inplace=True),
            conv3x3(out_c, out_c),
            nn.BatchNorm2d(out_c),
            nn.ReLU(inplace=True),
        )
        # 스케일이 있으니 in_c == out_c 이므로 신경 x
    def forward(self, x):
        return self.net(x) # y = G(x) (그냥 H(x))
```

```
# 2) Residual (post-activation, 원 논문 스타일)
class ResBlockPostAct(nn.Module):
    def __init__(self, in_c, out_c, stride=1):
        super().__init__()
        self.f = nn.Sequential(
            conv3x3(in_c, out_c, stride),
            nn.BatchNorm2d(out_c),
            nn.ReLU(inplace=True),
            conv3x3(out_c, out_c),
            nn.BatchNorm2d(out_c),
        ) # 마지막에 ReLU 없음 (함수 위에 ReLU)
        self.skip = nn.Identity() if (in_c == out_c and stride == 1) \
            else nn.Sequential(conv1x1(in_c, out_c, stride),
                               nn.BatchNorm2d(out_c))
        self.act = nn.ReLU(inplace=True)

    # * 활성화 층 기준: 마지막 BN의 gamma를 0으로 두면 F(x)=0에서 시작
    def init_(self, f=-1):
        nn.init.constant_(self.f[-1].weight, 0.0)

    def forward(self, x):
        y = self.skip(x) + self.f(x) # y_pre = x + F(x)
        return self.act(y) # post-activation
```

Plain

ResNet

$$\rightarrow F, \gamma \text{ 차원 } \neq : y = \mathcal{F}(x, \{W_i\}) + W_s x$$

한국어 해석  
차원 맞추기

2.

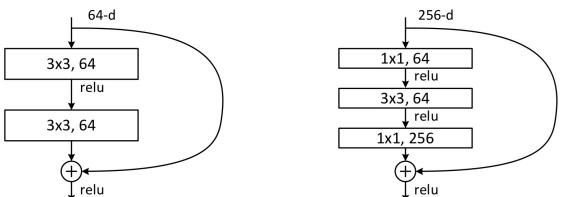


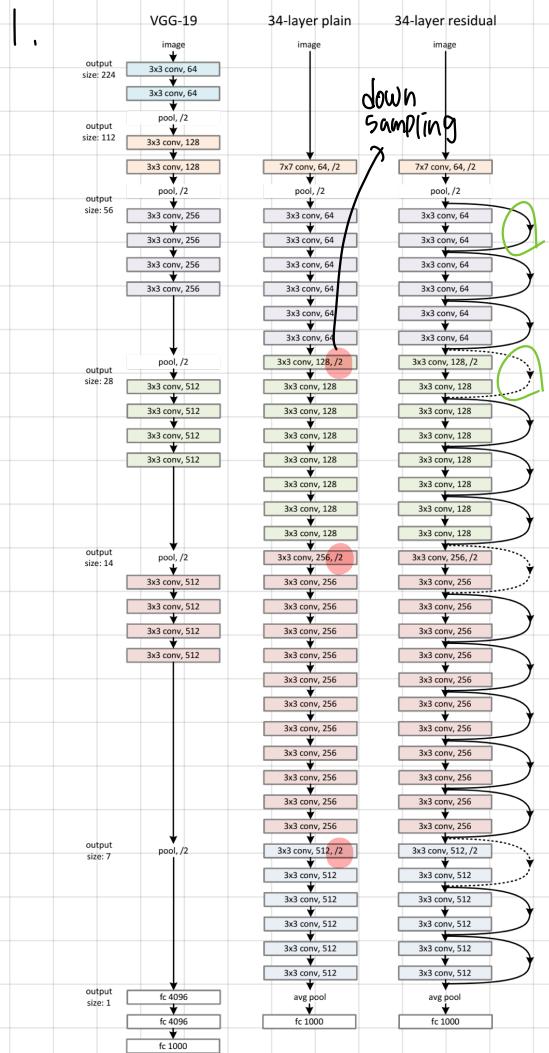
Figure 5. A deeper residual function  $\mathcal{F}$  for ImageNet. Left: a building block (on  $56 \times 56$  feature maps) as in Fig. 3 for ResNet-34. Right: a “bottleneck” building block for ResNet-50/101/152.

2/2

3/2

$$\rightarrow \text{단종}: y = W_s x + \gamma \circ \text{ReLU}(x)$$

## 3.3 Network Architectures



### 1. Plain(VGG)

- 3x3
- 동일한 흐름 패턴을 그릴 때는 층마다 패턴을 갖게
- 디아그램은 이미지의 일부만 표시, 전체는 28x28 (전체 이미지)
- down sampling: stride=2 conv
- global average pooling
- 1000 class fc + softmax

### 2. 34-layer plain ↘ : + 34 in weight layers

### 3. Residual Net

→ Shortcut connection 추가

↳ 이런 다른 패턴 (예: zero padding) (예: 2x2 stride) downsampling

- (A) 차원 증가로 zero-padding (예: 2x2 stride) ) 등장 stride=2
- (B) Projection: 1x1 Conv

= "Identity mapping"

= "Projection mapping"

## 3.4 Implementation

1. Image:  
 - Scale aug: [256, 480] 틱으로 등비 스케일  
 - 원본, 256x480 → 224x224 patches, 템플릿으로 정준화  
 - color aug

test: 흰 10 crop + (fully-conv)  
 $[224, 256, 384, 480, 640] \rightarrow scales$

### 2. Conv + Batch-Norm + Activation

3. Optimization: mini-batch 256 / SGD / lr = 0.1  $\xrightarrow{\text{warmup}}$  / max epochs: 60 x 10^4

weight-decay: SGD(model.parameters(), lr=0.1, momentum=0.9, weight\_decay=0.0001)

### 4. dropout X

# 4. Experiments

## 4.1 ImageNet Classification

### 1. Plain net VS Resnet

Plain net

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112x112					
conv2_x	56x56	[3x3, 64] x2	[3x3, 64] x3	[1x1, 64] 3x3, 64 x3	[1x1, 64] 3x3, 64 x3	[1x1, 64] 3x3, 64 x3
conv3_x	28x28	[3x3, 128] x2	[3x3, 128] x4	[1x1, 128] 3x3, 128 x4	[1x1, 128] 3x3, 128 x8	[1x1, 128] 3x3, 128 x8
conv4_x	14x14	[3x3, 256] x2	[3x3, 256] x6	[1x1, 256] 3x3, 256 x6	[1x1, 256] 3x3, 256 x23	[1x1, 256] 3x3, 256 x36
conv5_x	7x7	[3x3, 512] x2	[3x3, 512] x3	[1x1, 512] 3x3, 512 x3	[1x1, 512] 3x3, 512 x3	[1x1, 512] 3x3, 512 x3
	1x1					
	FLOPs	1.8x10 <sup>9</sup>	3.6x10 <sup>9</sup>	3.8x10 <sup>9</sup>	7.6x10 <sup>9</sup>	11.3x10 <sup>9</sup>

Table 1. Architectures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of blocks stacked. Down-sampling is performed by conv3\_1, conv4\_1, and conv5\_1 with a stride of 2.

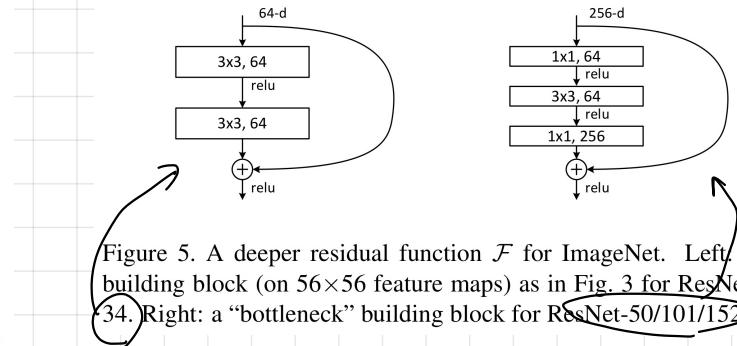


Figure 5. A deeper residual function  $\mathcal{F}$  for ImageNet. Left: a building block (on 56x56 feature maps) as in Fig. 3 for ResNet-34. Right: a “bottleneck” building block for ResNet-50/101/152

→ Red box: Stride = 2 or 3 downsampling (Identity shortcut)

2.

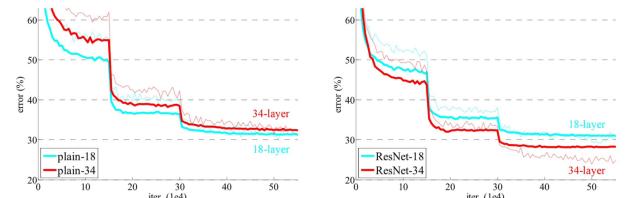


Figure 4. Training on ImageNet. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

→ 3번: Validation

→ Resnet: 더 빠른 수렴

→ 18 layers는 plain SGD 훈련과 같은 성능 (0)

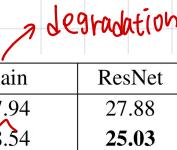


Table 2. Top-1 error (%), 10-crop testing) on ImageNet validation. Here the ResNets have no extra parameter compared to their plain counterparts. Fig. 4 shows the training procedures.

### 3. Identity mapping VS Projection mapping

→ A: zero Padding (parameter-free), identity  
 B: projection shortcut + identity mapping  
 C: all projection mapping

→ B은 zero padding → A(identity), B ( $A < B$  때는 T)

→ bottleneck은 모두

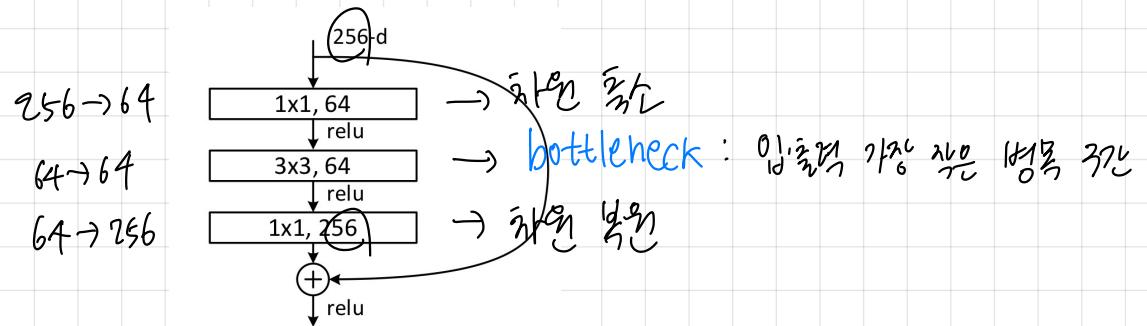
model	top-1 err.	top-5 err.
VGG-16 [41]	28.07	9.33
GoogLeNet [44]	-	9.15
PReLU-net [13]	24.27	7.38
plain-34	28.54	10.02
ResNet-34 A	25.03	7.76
ResNet-34 B	24.52	7.46
ResNet-34 C	24.19	7.40
ResNet-50	22.85	6.71
ResNet-101	21.75	6.05
ResNet-152	<b>21.43</b>	<b>5.71</b>

Table 3. Error rates (%), 10-crop testing) on ImageNet validation. VGG-16 is based on our test. ResNet-50/101/152 are of option B that only uses projections for increasing dimensions.

#### 4. Deeper Bottleneck Architectures

→ 50 / 101 / 152

→ 3 layers (Fig 5)



↳ 허드라인 짧은 Shortcut: Projection을 하면 연산량 줄어

→ 50: 34 + 2 → 3 뼈장 + B

(101/152: 32 뼈장 + 1 뼈장)

method	top-1 err.	top-5 err.
VGG [41] (ILSVRC'14)	-	8.43 <sup>†</sup>
GoogLeNet [44] (ILSVRC'14)	-	7.89
VGG [41] (v5)	24.4	7.1
PReLU-net [13]	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	<b>19.38</b>	<b>4.49</b>

↓ 2017년 모델

Table 4. Error rates (%) of single-model results on the ImageNet validation set (except <sup>†</sup> reported on the test set).

#### 4.2 CIFAR-10 and Analysis

(논문 핵심)

#### 4.3 Object Detection on PASCAL and MS COCO

(논문 핵심)

- resnet에서 제시한건 잔차매핑 즉  $H(x) = x$  가 되는 항등매핑을 위한  $H(x)$  레이어들을 직접 최적화하지 않고,  $F(x) = H(x) - x$  를 레이어로 해서 그걸 최적화 하도록 한다.
  - >  $H(x), F(x)$  모두 레이어는 맞고  $F(x)$  구현에서  $H(x)$  를 안 쓰는건 아님
  - > 코드에서 보듯이  $H(x), F(x)$  모두 레이어이고, 구현 자체가 크게 다르지 않음 (같은 모델 레이어 스택)
  - > 하지만 최적화 자체가 달라짐 ( $F(x)$  는 모델 스택 구조는  $H(x)$  와 같지만 마지막 relu 전의  $x$  를 더하고 relu 하는 작업에서 각 스택에서의 레이어들의 가중치를 최적화 해야하는건 동일하지만 최적화 방식이 달라짐)

→ plain 은  $H(x)$  자체, residual 은  $H(x)$  을  $F(x) + x$  로 두기

#### ▼ 코드

```

import torch
import torch.nn as nn

def conv3×3(in_c, out_c, stride=1):
    return nn.Conv2d(in_c, out_c, 3, stride=stride, padding=1, bias=False)

def conv1×1(in_c, out_c, stride=1):
    return nn.Conv2d(in_c, out_c, 1, stride=stride, bias=False)

# 1) Plain: H(x)만 학습 (스킵 없음)
class PlainBlock(nn.Module):
    def __init__(self, in_c, out_c, stride=1):
        super().__init__()
        self.net = nn.Sequential(
            conv3×3(in_c, out_c, stride),
            nn.BatchNorm2d(out_c),
            nn.ReLU(inplace=True),
            conv3×3(out_c, out_c),
            nn.BatchNorm2d(out_c),
            nn.ReLU(inplace=True),

```

```

        )
        # 스킵이 없으니 in_c != out_c 여도 신경 X
    def forward(self, x):
        return self.net(x)           # y = G(x) (그냥 H(x))

# 2) Residual (post-activation, 원 논문 스타일)
class ResBlockPostAct(nn.Module):
    def __init__(self, in_c, out_c, stride=1):
        super().__init__()
        self.f = nn.Sequential(
            conv3x3(in_c, out_c, stride),
            nn.BatchNorm2d(out_c),
            nn.ReLU(inplace=True),
            conv3x3(out_c, out_c),
            nn.BatchNorm2d(out_c),      # 마지막에 ReLU 없음 (합 뒤에
ReLU)
    )
    self.skip = nn.Identity() if (in_c == out_c and stride == 1) \
        else nn.Sequential(conv1x1(in_c, out_c, stride),
                           nn.BatchNorm2d(out_c))
    self.act = nn.ReLU(inplace=True)

    # ★ 항등 쉬운 초기화: 마지막 BN의 gamma를 0으로 두면 F(x)≈0에
    서 시작
    nn.init.constant_(self.f[-1].weight, 0.0)

    def forward(self, x):
        y = self.skip(x) + self.f(x)      # y_pre = x + F(x)
        return self.act(y)               # post-activation

# 3) Residual (pre-activation, He et al. 2016 개정)
class ResBlockPreAct(nn.Module):
    def __init__(self, in_c, out_c, stride=1):
        super().__init__()
        self.bn1 = nn.BatchNorm2d(in_c)
        self.relu1 = nn.ReLU(inplace=True)
        self.conv1 = conv3x3(in_c, out_c, stride)
        self.bn2 = nn.BatchNorm2d(out_c)

```

```

self.relu2 = nn.ReLU(inplace=True)
self.conv2 = conv3×3(out_c, out_c)
self.skip = nn.Identity() if (in_c == out_c and stride == 1) \
    else nn.Sequential(conv1×1(in_c, out_c, stride))

# ★ 항등 용이 초기화
nn.init.constant_(self.bn2.weight, 0.0)

def forward(self, x):
    out = self.conv1(self.relu1(self.bn1(x)))
    out = self.conv2(self.relu2(self.bn2(out)))
    return self.skip(x) + out      # 합 뒤 활성화 없음 → F(x)=0이면
정확히 y=x

```

2. 순서가 깊은 모델들이 성능이 안 좋아서  $H(x)$  를 직접 최적화 하도록 해봤는데, 그게 degradation 을 발생시켜서  $F(x)$  라는 잔차 최적화를 만들기로 했다.
3.  $H(x)$  를 직접 최적화 하는 항등 매핑이라고 했을때, 입력과 출력이 같아진다는 말은 이미 혹은 이전 레이어 피처맵의 그 각 요소와 현재 레이어와 relu 를 거쳐 나온 피처맵의 각 요소가 같다는 말인가?  
-> 맞/ 차원 다르면 subsampling 으로 차원 맞춰줌 (element-wise 로 같음)
4. 항등매핑의 장점은 뭐임? 단순히  $x$  더해지는거 아님? (단순히  $x$  가 요소별로 더해지는 거 아님?) 역전파 시 뭐가 대체 달라진다는거임?  
→ 쓸모없는 블록을 0으로 만들고 입력 그대로를 유지 (쓸모없는 블록이라면  $F$  를 0으로)  
→ 모든 층이 항등이 목표라는게 아니라 필요없어 보이는 블록을 막연하게 제거하지 않고 항등 매핑으로 해보려고 했는데, 그게  $H(x)$  를 직접 최적화 하려고 하니까 깊은 층에서는 너무 어려운 문제가 된다 그래서  $F(x)$  를 이용함 (plain 에서 필요없는 블록의 항등 매핑은 그 블록의 최종 결과  $H(x)$  가  $x$  런 같아지도록 학습함 / residual 에서는 그 블록 결과를  $F(x) + x$  로 둬서  $F(x)$  라는걸 0으로 만들게 함)  
→ plain (이전 모델들): 단순히 블록 많이 쌓음 → degradation 문제  
→ resnet
  - plain 에서 단순 블록 쌓은것에서 만약 필요없는 층이 있다면 그냥 얇은 층 복사 + 항등 매핑 층(필요없는 층은 아무일 안하고 입력이랑 같게 하려고) 하면 되는데 (이론적으로는 그러한데,)

- 근데 이게  $H \approx x$  를 직접하기에는 그 많은 항등 매핑을 위한 가중치를 직접 찾기 어려움  $\rightarrow F(x)$  도입

$\rightarrow F(x) + x$  로  $x$  를 단순히 element-wise 로 더하는거에서 loss gradient 하는건데 어째서 더 최적화가 잘되는거지?

- 단순 덧셈의 의미가 아니라 계산 그래프에 항등 경로라는게 생김
  - 역전파 할 때, 적어도  $I$  가 남아있으니까 기울기 소실이 안 일어남
  - $J$  를 작게 시작하도록 설계하면 폭발할 일도 없음
  - 항등 경로라는건 연산 노드 '+' 여서 가중치도 없음

#### ▼ plain

**Plain(스킵 없음)**

- 순전파:  $y = H(x)$
- 역전파(체인룰):  $\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y} J_H(x)$

```
bash
x → [ H ] → y
|
역전파: | dL/dx = dL/dy · J_H
▼
```

여기서  $J_H(x) = H$ 의 야코비안(미분 행렬).

**Plain**

- $z_1 = H_0(x), z_2 = H_1(z_1), \dots, z_L = H_{L-1}(z_{L-1})$
- 역전파:

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial z_L} \prod_{\ell=0}^{L-1} J_{H_\ell}(z_\ell)$$

$\rightarrow$  야코비안들의 곱이 누적(소실/폭발 위험).

#### ▼ residual

### Residual(스킬 있음)

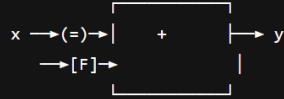
- 순전파:  $y = x + F(x)$
- 역전파:

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y} \underbrace{\frac{\partial y}{\partial x}}_{I+J_F(x)} = \frac{\partial \mathcal{L}}{\partial y} (I + J_F(x))$$

(+ 연산의 미분이 항등  $I$  라서 이렇게 됩니다.)

bash

☞ 코드 복사



역전파 at '+' : 기울기가 두 입력으로 그대로 복제(도함수=I)

결과:  $dL/dx = dL/dy \cdot (I + J_F)$

핵심: 덧셈 노드(+)의 도함수 = I 이므로, 역전파 식에 항등항 I 가 항상 끼어듭니다.

### Residual

- $z_1 = x + F_0(x), z_2 = z_1 + F_1(z_1), \dots$
- 역전파:

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial z_L} \prod_{\ell=0}^{L-1} (I + J_{F_\ell}(z_\ell))$$

→ 각 항이  $I$  를 포함.  $J_{F_\ell}$ 를 작게 시작하면(예: 마지막 BN의  $\gamma = 0$ )  
 $I + J_{F_\ell} \approx I$  이라 곱 전체가 1 근처에 머무름(기울기 보존).