Deep Residual Learning for Image Recognition

▼ Experiments

4.1 ImageNet Classification

▼ **실험 data set**

- **데이터셋**: ImageNet 2012 (1000 클래스)
- **데이터 구성**:
  - Train: 1.28M 이미지
  - Validation: 50K 이미지
  - Test: 100K 이미지
- **평가 지표**: Top-1, Top-5 error rate

▼ **Plain Networks**

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				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
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

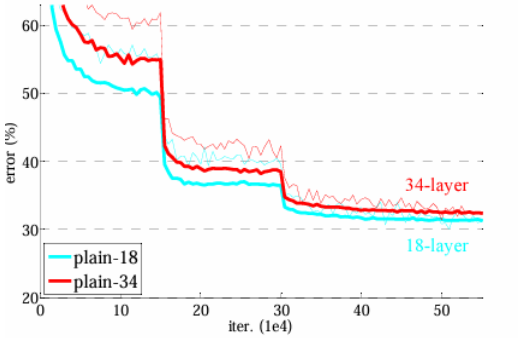
Table 1. Architectures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of blocks stacked. Down-sampling is performed by conv3_1, conv4_1, and conv5_1 with a stride of 2.

- FLOPs:
  CNN이나 ResNet 같은 모델이 한 번의 forward pass(입력 → 출력)를 수행할 때 필요한 **총 연산 횟수**

  표는 **깊이에 따른 레이어 구성 변화**를 보여줌
  - 깊이가 깊어질수록 각 블록 반복 횟수가 많아짐 → 더 많은 파라미터와 연산량

- conv2_x, conv3_x, conv4_x, conv5_x에서 채널 수가 64 → 128 → 256 → 512로 증가하면서 feature 표현력을 높임
- 이 구성은 **plain network** 설계 기준이며, ResNet에서는 여기에 **shortcut connection**만 추가됨

- **18-layer plain net과 34-layer plain net 비교.**



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

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.

- **표 해석:**
  - **X축 (iter. 1e4)** : 학습 반복 횟수(Iteration × 10,000 단위)
  - **Y축 (error %)** : 에러율(%) — 낮을수록 성능이 좋음
  - **굵은 선** : 검증(validation) 데이터셋의 에러율
  - **얇은 선** : 훈련(training) 데이터셋의 에러율
- **훈련/검증 에러 곡선을 비교한 결과:**
  - 34층 plain net은 학습 내내 **훈련 오차도 더 높음**.
  - 18층의 해공간(solution space)은 34층 해공간의 부분집합임에도 불구하고, 깊어진 모델이 오히려 학습 성능이

- **Top-1 error** : 모델이 예측한 1순위 클래스가 정답이 아닐 확률
- 34층 plain net의 검증에러가 18층 plain net보다 높음.

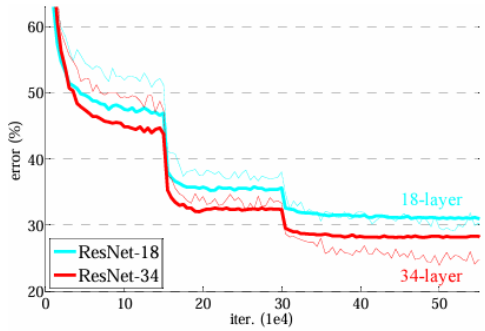
떨어짐 → **Degradation Problem** 발생.

- 이 현상은 **vanishing gradient** 때문이 아님.
  - 모든 plain network는 BN [16] 사용 → forward 신호의 분산이 0이 되지 않도록 보장.
  - backward 신호(gradient)도 BN 덕분에 정상적인 크기 유지.
  - 즉, forward/backward 신호가 사라지는 현상은 없음.
- 34층 plain net도 일정 수준의 정확도를 달성하지만,

  수렴 속도가 **지수적으로 느려져서** 학습 오차 감소가 느림.
- 결론: 깊은 plain net은 최적화 난이도가 커서 학습 효율이 떨어지며, 이 이유는 앞으로 더 연구가 필요→ Residual Learning 이 필요한 이유

▼ Residual Networks

- 실험 개요
  - 모델: 18-layer ResNet, 34-layer ResNet
  - 구조: plain net + 각 3×3 필터 쌍마다 shortcut connection (Fig. 3 Right)
  - 파라미터 수: plain net과 동일
  - Shortcut 설정:
    - 모든 shortcut: identity mapping
    - 차원 증가 시 zero-padding (Option A)



- 34층 ResNet은 훈련 에러가 훨씬 낮고, 검증 데이터에서도 잘 동작.

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

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.

- plain net과 달리, 34층 ResNet이 18층 ResNet보다 성능이 **2.8% 더 좋음**.
- 더 깊어져도 성능이 떨어지지 않고 향상

- 즉, **Degradation Problem**이 해결됨

  → 깊어질수록 정확도 향상.

- 동일한 깊이(34층)에서 ResNet이 top-1 error를 **3.5% 감소**.
- 이는 훈련 에러가 줄어든 덕분이며, residual learning이 깊은 네트워크에서 효과적임을 검증.

- **Plain Net vs Residual Net**

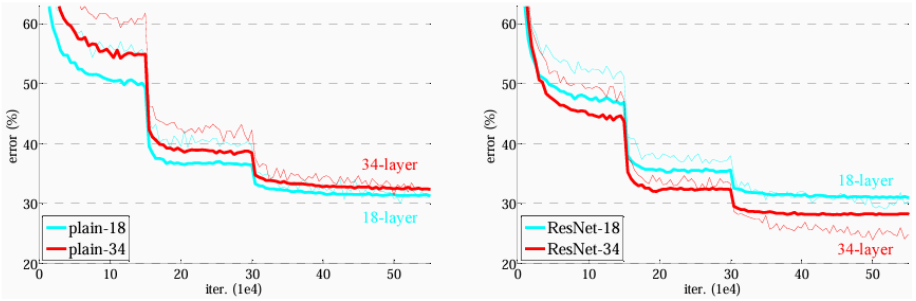


Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

- 18층 (얕은 network 에서 비교)
  - plain vs ResNet의 최종 정확도는 비슷.
  - 하지만 ResNet이 **더 빠르게 수렴** (Fig. 4 오른쪽 vs 왼쪽).
  - 깊지 않은 경우 SGD로도 좋은 해를 찾을 수 있지만, ResNet은 학습 초기에 최적화 속도를 높여줌.

▼ **Identity vs Projection Shortcuts**

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	**21.43**	**5.71**

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.

정의

- **Identity Shortcut**: 입력을 그대로 다음 레이어로 더하는 구조, 추가 파라미터 없음.
- **Projection Shortcut**: 1×1 convolution 등을 사용해 차원 맞추기, 추가 파라미터 있음

옵션 비교

- **A**: 차원 증가 시 zero-padding + 나머지는 identity (parameter-free)
- **B**: 차원 증가 시 projection + 나머지는 identity
- **C**: 모든 shortcut에 projection 적용

결과 해석

1. **Residual Learning 효과**
   - 모든 옵션(A/B/C)이 plain network 대비 오류율 크게 감소
2. **Projection Shortcut의 필요성**
   - C > B > A 순으로 약간의 성능 차이 있지만, 차이는 작음
   - Projection Shortcut은 필수 아님
3. **Identity Shortcut 중요성**

- 모델 복잡도 증가 없이 정확도 향상 가능

4. **깊이 증가 효과**
    - ResNet-50, 101, 152로 갈수록 Top-1, Top-5 오류율 모두 지속 감소

결론

- 차원 증가가 필요 없는 경우 → **Identity Shortcu**(A옵션)이 효율적
- 차원 증가 필요 시 → **Projection Shortcut** 사용(B 옵션) 권장
- 전체 Projection(C옵션)은 성능은 약간 향상되지만 계산량·모델 크기 증가 → 실용성 떨어짐

▼ Deeper Bottleneck Architectures

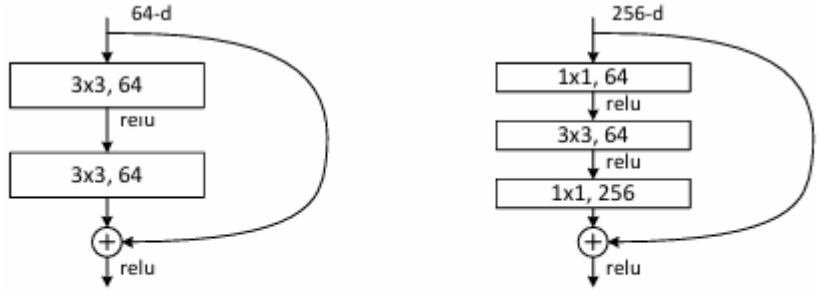


Figure 5. A deeper residual function $\mathcal{F}$ for ImageNet. Left: a building block (on 56×56 feature maps) as in Fig. 3 for ResNet-34. Right: a "bottleneck" building block for ResNet-50/101/152.

- **목표**: ResNet을 더 깊게 만들되, **연산량과 학습 시간**을 효율적으로 유지.
- **방법**: 기존 블록(2개의 3×3 convolution)을 **bottleneck 디자인**으로 변경.
    - Bottleneck: 3개의 계층 사용 (1×1 → 3×3 → 1×1 convolution).
    - 1×1 conv: 채널 수를 **감소**(reduce) 후, 다시 **복원**(restore).
    - 3×3 conv: 줄어든 채널 수로 연산, 효율성 향상
- **Identity Shortcuts(항등 지름길)**:
    - 추가 파라미터 없이 연결.
    - Bottleneck 구조에서 **투영(projection)**으로 바꾸면, **모델 크기와 연산량 2배** 증가.

    - 따라서 병목 디자인에서는 **Identity shortcut**이 **메모리·시간 복잡도 절감**에 핵심.
- **핵심**
    - Bottleneck 설계는 **더 깊은 ResNet**(50, 101, 152층)을 가능하게 함.
    - **연산 효율성**: 채널 수를 줄였다가 복원하는 방식으로 3×3 conv의 부담 완화.
    - **Identity shortcut**은 병목 구조의 효율성 유지에 매우 중요.

▼ 50-layer ResNet

- **50-layer ResNet**
    - 기존 34-layer ResNet의 **2-layer 블록**을 **bottleneck(3-layer) 블록**으로 교체.
    - 증가하는 차원에는 **Option B**(projection shortcut) 사용.
    - 연산량(FLOPs): **3.8B**.

▼ 101-layer and 152-layer ResNets

- **101-layer & 152-layer ResNet**
    - 더 많은 3-layer 블록을 쌓아 깊이를 확장.
    - 152-layer ResNet의 연산량이 **11.3B FLOPs**로, 오히려 VGG-16/19 (**15.3/19.6B**)보다 작음.
    - 깊어져도 **Degradation Problem이 발생하지 않음** → 깊이에 따른 정확도 향상만 얻음

▼ Comparisons with State-of-the-art Methods

method	top-1 err.	top-5 err.
VGG [41] (ILSVRC'14)	-	8.43†
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	**19.38**	**4.49**

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

- **지표 의미**
    - `top-1 err.` : 가장 높은 확률로 예측한 클래스가 정답이 아닌 비율.

- top-5 err. : 가장 높은 확률 5개 안에 정답이 없는 비율.

기존 모델 대비 우수한 성능

- ResNet 계열은 깊이가 깊어질수록 성능 향상(34 → 50 → 101 → 152).
- VGG, GoogLeNet 대비 ResNet이 훨씬 낮은 오류율
- Residual Learning이 단일 모델에서도 매우 강력함을 입증.
- 네트워크 깊이를 충분히 늘리면(degradation 문제 해결 전제) 성능이 계속 향상됨.

method	top-5 err. (**test**)
VGG [41] (ILSVRC'14)	7.32
GoogLeNet [44] (ILSVRC'14)	6.66
VGG [41] (v5)	6.8
PReLU-net [13]	4.94
BN-inception [16]	4.82
ResNet (ILSVRC'15)	**3.57**

Table 5. Error rates (%) of **ensembles**. The top-5 error is on the test set of ImageNet and reported by the test server.

1. **Ensemble에서도 압도적인 결과**
   - 깊이가 다른 ResNet 모델 6개를 조합(Ensemble)했더니 **Top-5 테스트 에러 3.57%**.
   - ILSVRC 2015 대회에서 1등 달성.
2. **의미**
   - ResNet 앙상블이 당시 모든 모델을 제치고 최저 오류율 달성.
   - 단일 모델에서도 강력했지만, 앙상블 시에는 SOTA를 크게 초월.
   - Residual 구조는 깊은 네트워크를 안정적으로 학습 가능하게 해주어, 앙상블에서도 강력한 성능을 발휘.

4.2 CIFAR-10 and Analysis

▼ CIFAR-10 실험

- CIFAR-10 실험 개요
  - 데이터: CIFAR-10 (50k train, 10k test, 10 classes)
  - 목표: SOTA 성능 X → **깊은 네트워크 학습 동작 분석**
  - 네트워크: Plain vs ResNet (동일한 depth/width/parameters)
- 네트워크 구조

    - **입력**: 32×32 (mean subtraction)
    - **구성**:
        1. 3×3 Conv
        2. Feature map 크기: {32, 16, 8}
        3. 각 크기마다 2n개의 Conv layer (필터 {16, 32, 64})
        4. 다운샘플링: stride=2 Conv
        5. Global Avg Pool → FC(10) → Softmax
    - **총 레이어 수**: 6n+2
    - **Shortcut**: 모든 경우 identity, 총 3n개
- 학습설정
    - Weight decay=0.0001, momentum=0.9
    - He init + BN, Dropout 없음
    - Batch=128, GPU 2개
    - LR: 0.1 시작 → 32k, 48k에서 /10 → 64k 종료
    - Data Aug: 4px padding + 32×32 crop + random flip

method			error (%)
Maxout [10]			9.38
NIN [25]			8.81
DSN [24]			8.22
	# layers	# params	
FitNet [35]	19	2.5M	8.39
Highway [42, 43]	19	2.3M	7.54 (7.72±0.16)
Highway [42, 43]	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

Table 6. Classification error on the **CIFAR-10** test set. All methods are with data augmentation. For ResNet-110, we run it 5 times and show "best (mean±std)" as in [43].

- 깊어질수록 ResNet 성능이 향상되지만, 너무 깊은 경우(1202층)에는 성능 저하 발생
- ResNet은 기존 모델보다 훨씬 적은 파라미터 수로 더 나은 성능 달성 가능

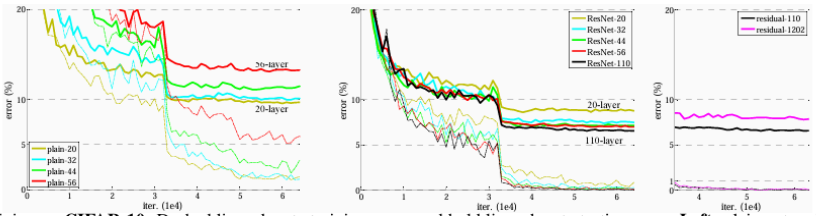


Figure 6. Training on **CIFAR-10**. Dashed lines denote training error, and bold lines denote testing error. **Left**: plain networks. The error of plain-110 is higher than 60% and not displayed. **Middle**: ResNets. **Right**: ResNets with 110 and 1202 layers.

- **Left**: Plain Net (20~56-layer) → 깊어질수록 train error와 test error 모두 증가 (Degradation)
- **Middle**: ResNet (20~110-layer) → 깊어질수록 성능 향상, 110-layer도 안정적 학습
- **Right**: 초깃값과 구조를 튜닝한 ResNet-110 vs ResNet-1202 → 둘 다 안정적이지만 1202-layer는 성능이 조금 떨어짐→ 과적합 & 최적화 난이도 증가

▼ Analysis of Layer Responses

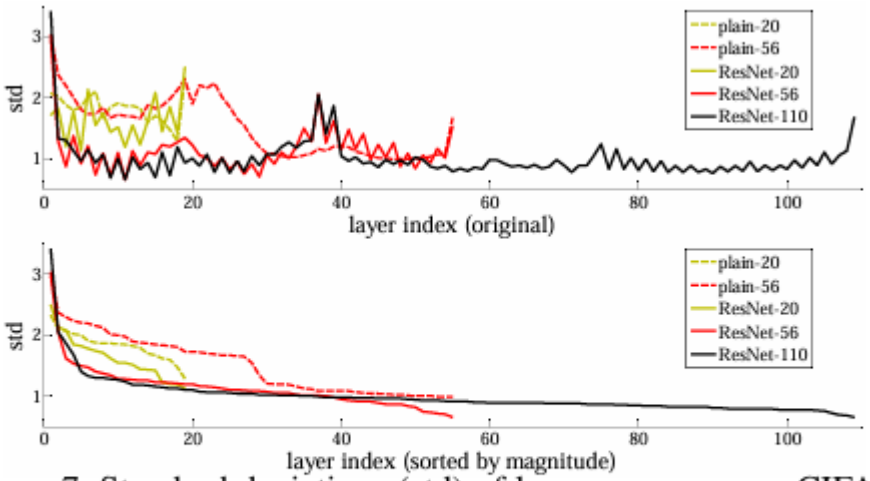
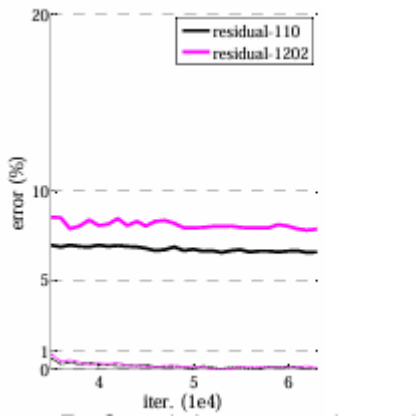


Figure 7. Standard deviations (std) of layer responses on CIFAR-10. The responses are the outputs of each 3×3 layer, after BN and before nonlinearity. **Top**: the layers are shown in their original order. **Bottom**: the responses are ranked in descending order.

- 그림 7은 레이어 반응(response)의 표준편차(std)를 보여준다.
- 여기서 반응이란 각 3×3 레이어의 출력값을 의미하며, **BN 이후**이지만 ReLU나 Add 연산 전의 값
- ResNet을 분석한 이유는 잔차 함수(residual function)의 "강도"를 보기 위함
- 그림 7 결과:
  - ResNet은 일반 네트워크보다 레이어 반응 크기가 작음 → 이는 residual function이 비잔차 함수보다 **0에 더 가까운 값**을 갖는 경향이 있음을 시사.
  - 더 깊은 ResNet(예: 110-layer)은 얕은 ResNet보다 반응 크기가 더 작음.
  - 레이어가 많아질수록 개별 레이어가 신호를 수정하는 정도가 줄어듦.

▼ Exploring Over 1000 layers



- 초심층 모델 1202-layer 네트워크 생성.
- **결과**:
  - 최적화 난이도 없음(no optimization difficulty) → 학습오차(training error) < 0.1%.
  - 그러나 테스트오차는 7.93%로 110-layer 네트워크(6.43%)보다 나쁨.
- **원인 분석**:
  - CIFAR-10 데이터셋 규모 대비 모델이 너무 커서(파라미터 19.4M) **오버피팅** 발생.
  - 강력한 정규화 기법(dropout, maxout 등)을 쓰지 않고, 깊고 얇은 구조로 설계했기 때문에 최적화 난이도는 낮았지만, 일반화 성능은 떨어짐
- **해결법**
  - 최고의 결과를 얻기 위해서는 **maxout** 이나 **dropout** 같은 강력한 정규화 기법 필요
  - 여기서는 연구 초점 흐리지 않기 위해 사용 x

4.3 Object Detection on PASCAL and MS COCO

▼ Object Detection on PASCAL and MS COCO

- **목적**: VGG-16 대비 ResNet-101의 객체 탐지 성능 비교
- **탐지 모델**: Faster R-CNN
- **주요 개선 포인트**:
  - 동일한 탐지 구현에서 네트워크만 교체
  - 성능 향상은 네트워크 구조 개선 덕분

training data	07+12	07++12
test data	VOC 07 test	VOC 12 test
VGG-16	73.2	70.4
ResNet-101	**76.4**	**73.8**

Table 7. Object detection mAP (%) on the PASCAL VOC 2007/2012 test sets using **baseline** Faster R-CNN. See also Table 10 and 11 for better results.

- **표 7**: 비교적 단순한 데이터셋(PASCAL VOC)에서도 ResNet이 VGG 대비 향상됨을 검증

metric	mAP@.5	mAP@[.5, .95]
VGG-16	41.5	21.2
ResNet-101	**48.4**	**27.2**

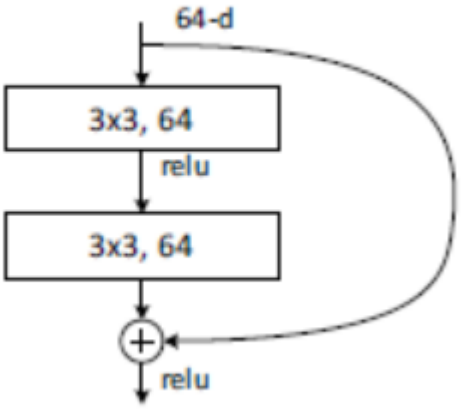
Table 8. Object detection mAP (%) on the COCO validation set using **baseline** Faster R-CNN. See also Table 9 for better results.

- **표 8**: 더 어려운 데이터셋(COCO)에서도 성능 우위를 보여 ResNet의 범용성 입증

Code Review

1. **ResNet BasicBlock**
    - ResNet18과 ResNet34에 사용되는 Building Block
    - in_planes와 planes, stride라는 변수를 받아서 생성

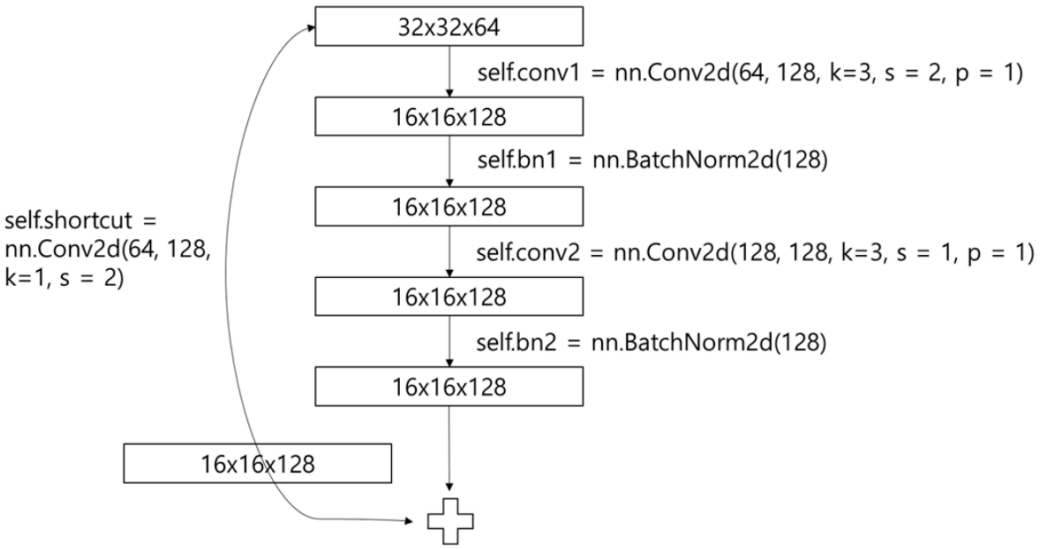
    1. 기본 구조: 3x3 conv - BN - 3×3 conv - BN - shortcut



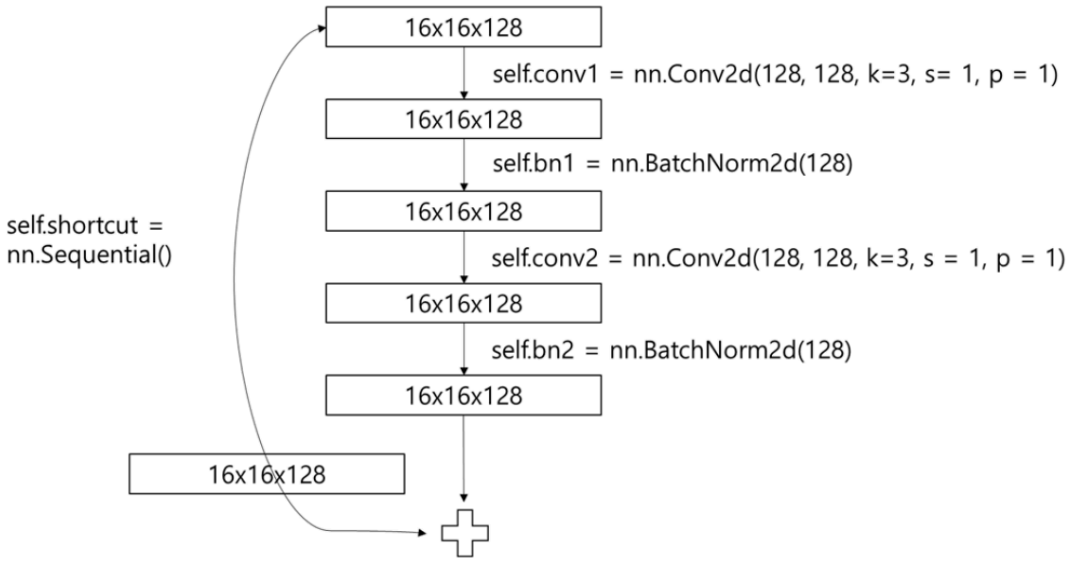
case1) stride가 1이 아니거나 in_planes != self.expansion*planes인 경우

- (즉, input으로 들어온 차원과 output에 해당하는 차원이 다른 경우)에 해당
- input인 32x32×64이었으나, layer가 거치면서 16x16×128가 되었으므로 input을 그대로 더해줄 수 없게 됨

따라서, 1x1 conv를 이용해서 channel 수를 맞춰주면서 동시에 stride = 2를 적용하여 가로, 세로 길이도 /2 해서 16x16×128를 맞춰주게 되면 shortcut을 더해줄 수 있게 된다



case2) shape 동일한 경우




```
class BasicBlock(nn.Module):
    expansion = 1
    def __init__(self, in_planes, planes, stride=1):
    super(BasicBlock, self).__init__()
    self.conv1 = nn.Conv2d(
        in_planes, planes, kernel_size=3, stride=stride, padding=1, bias=False)
    self.bn1 = nn.BatchNorm2d(planes)
    self.conv2 = nn.Conv2d(planes, planes, kernel_size=3,
                           stride=1, padding=1, bias=False)
    self.bn2 = nn.BatchNorm2d(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 = self.bn2(self.conv2(out))
```

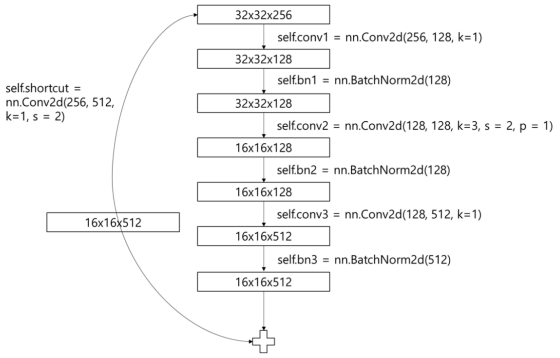
```
    out += self.shortcut(x)
    out = F.relu(out)
```

2. **Bottleneck**

- ResNet의 layer가 50층 이상 넘어갈 경우, 연산량을 줄여주기 위해

기존의 3x3 conv를 두 개 쌓아서 만든 layer를 사용하는 것이 아닌, 1x1 conv - 3×3 conv - 1×1 conv를 이용하는 방식

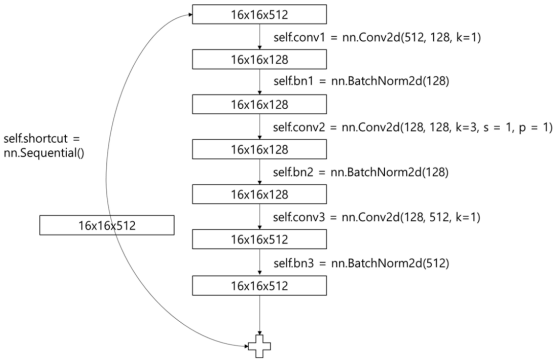
1. if 문에 해당하는 경우



1. 1×1 conv을 이용해서 차원을 128차원으로 줄임
2. 3×3 conv 연산을 진행
3. 1×1 conv 연산을 통해서 다시 512차원으로 늘리기

1×1 conv을 통해서 차원을 맞춰주면서 동시에 stride = 2를 적용해 가로, 세로 사이즈를 1/2 해준다

2. stride 가 1인 경우



- channel의 수만 중간에 한번 변경

→ 3×3 conv를 적용할 때 더 적은 차원의 데이터만 적용하도록 만들어서 연산량을 줄이기 위함

- nn.Sequential()을 이용해서 shortcut을 진행

```
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
```

3. **ResNet Class**

```
class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=10):
        super(ResNet, self).__init__()
        self.in_planes = 64
        #3×3 conv - BN을 적용
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3,
                               stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.layer1 = self._make_layer(block, 64, num_blocks[0], stride=1)
        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)
        self.linear = nn.Linear(512*block.expansion, num_classes)
    #_make_layer 함수를 이용해서 block이 포함된 layer들을 만들어준다
    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1]*(num_blocks-1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_planes, planes, stride))
            self.in_planes = planes * block.expansion
        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
```

4. **Train & Test**

```
def train(epoch):
    print('\nEpoch: %d' % epoch)
    net.train()
    train_loss = 0
    correct = 0
    total = 0
    batch_count = 0
    for batch_idx, (inputs, targets) in enumerate(trainloader):
    #batch size 단위로 받은 이미지를 net에 투입해서 예측 결과인 outputs을 만든다
        inputs, targets = inputs.to(device), targets.to(device)
        optimizer.zero_grad()
        outputs = net(inputs)
```

```
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
        # target & loss계산

        train_loss += loss.item() # 누적해서 loss 더함
        _, predicted = outputs.max(1)
        total += targets.size(0)
        correct += predicted.eq(targets).sum().item()
        # predicted 와 target값 같을 때 1 아닐때 0 맞힌 개수 세기
        batch_count += 1

    print("Train Loss : {:.3f} | Train Acc: {:.3f}".format(train_loss / batch_coun
t, 100.*correct/total))
    final_loss = train_loss / batch_count
    final_acc = 100.*correct / total
    return final_loss, final_acc
```

```
def test(epoch):
    global best_acc
    net.eval()
    test_loss = 0
    correct = 0
    total = 0
    batch_count = 0
    with torch.no_grad():
        for batch_idx, (inputs, targets) in enumerate(testloader):
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = net(inputs)
            loss = criterion(outputs, targets)

            test_loss += loss.item()
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()
            batch_count += 1
```

```
    print("Test Loss : {:.3f} | Test Acc: {:.3f}".format(test_loss / batch_count,
100.*correct/total))

    # Save checkpoint.
    acc = 100.*correct/total
    if acc > best_acc:
        print('Saving..')
        state = {
            'net': net.state_dict(),
            'acc': acc,
            'epoch': epoch,
        }
        if not os.path.isdir(os.path.join(saved_loc, 'checkpoint')):
            os.mkdir(os.path.join(saved_loc, 'checkpoint'))
        torch.save(state, os.path.join(saved_loc, 'checkpoint/ckpt.pth'))
        best_acc = acc

    final_loss = test_loss / batch_count
    return final_loss, acc
```