# ResNet(2015)

Deep Residual Learning for Image Recognition(2015)를 바탕으로 섬머리1, 섬머리2를 많이 참고했습니다.

원래는 페이퍼 한 줄, 한 줄의 의미를 곱씹어 보면서 봐야지만, 여러 섬머리를 참고하여 페이퍼의 중요한 부분만 해석하면서 코드 리뷰를 중요하게 봤습니다.

## ResNet의 가정 & Residual Learning

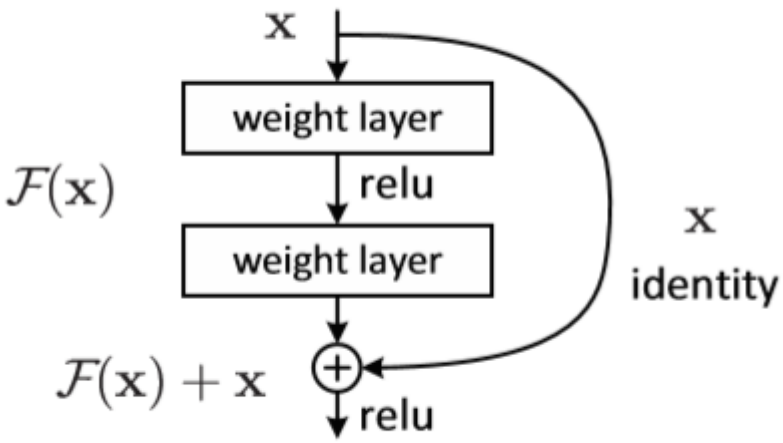



Figure 2. Residual learning: a building block.

1. H(x)를 몇개의 스택된 레이어(반드시 전체 네트워크는 아님)에 의해 매핑된 것으로 가정하고, x는 이 레이어의 첫 번째 인풋이다. (Let us consider H(x) as an underlying mapping to be fit by a few stacked layers (not necessarily the entire net), with x denoting the inputs to the first of these layers)

2. 가정은 멀티 비선형 레이어들이 복잡한 function^2에 근사할 수 있다고 가정하면, **residual function H(x) - x에 근사할 수 있다와 동일하다.**(If one hypothesizes that multiple nonlinear layers can asymptotically approximate complicated functions^2, then it is equivalent to hypothesize that they can asymptotically approximate the residual functions, (i.e., H(x)−x.))

3. 따라서 H(x)를 근사하기 위해 스택된 레이어에 기대하기 보다는, 이 레이어를 F(x) := H(x) - x에 근사하게 명시한다. 원래 함수는 F(x)+x이 된다.(So rather than expect stacked layers to approximate H(x), we explicitly let these layers approximate a residual function F(x) := H(x) − x. The original function thus becomes F(x)+x.)

4. residual mapping이 일반적인, unreferenced mapping보다 쉽게 optimize한다고 가정.(We hypothesize that it is easier to optimize the residual mapping than to optimize the original, unreferenced mapping.)

H(x) : 학습해야 할 target function이 F(x) := H(x) - x 처럼 output과 input의 차이인 F(x)을 학습하는 것이 residual representation의 주요 개념이다. 만약 H(x)가 identity function이라면, (H(x) = x) 이를 층을 쌓하서 학습 시키기 보다는 F(x)를 0으로 학습시키는 것이 더 용이함.

## Identity mapping by Shortcuts

1. F(x) + x 함수는 shorcut connections와 함께 순전파 계산으로 구현된다.(The formulation of F(x) + x can be realized by feed forward neural networks with “shortcut connections”)

$$y = F(x, \{W_i\}) + x$$
$$F = W_2\sigma(W_1 x)$$

sigma는 ReLu를 뜻함, 단 위의 식에서는 F와 x가 동일한 dimension이어야 한다.

2. 망이 깊어지면 vanishing gradient 문제 때문에 학습이 더 어려워지는 단점이 있는데 이를 degradation problem이라고 함.(The degradation problem suggests that the solvers might have difficulties in approximating identity mappings by multiple nonlinear layers)

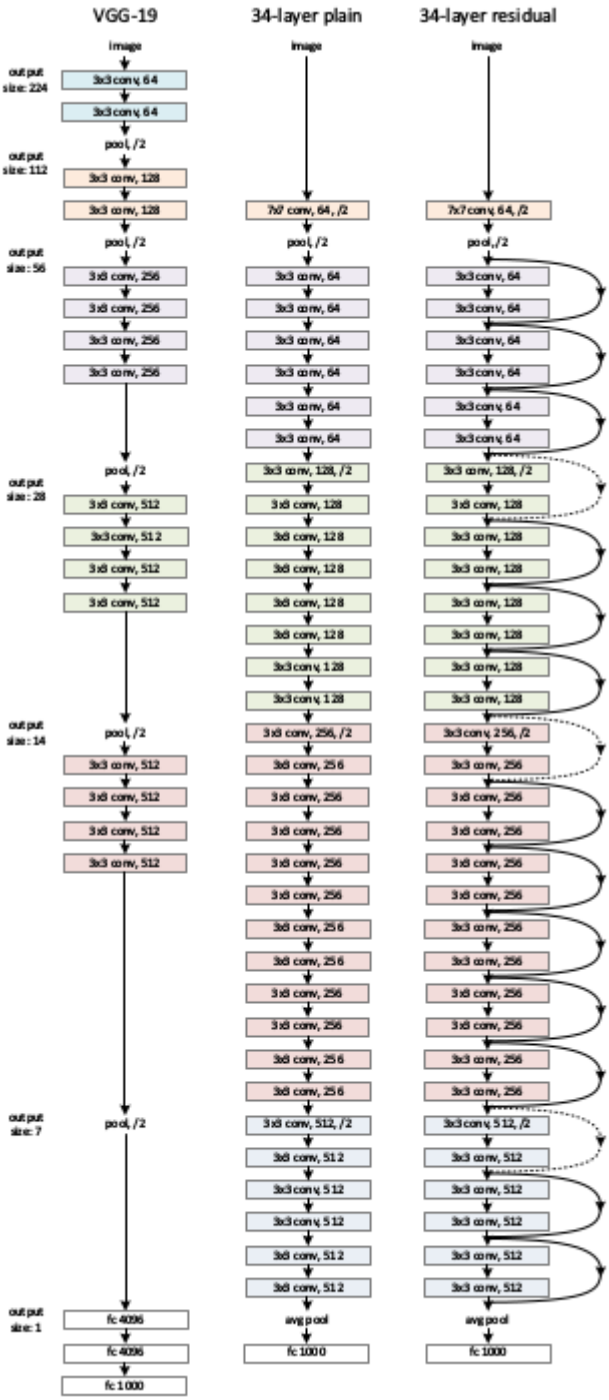
shortcut connections을 역전파 할 때, identity mapping(x)를 미분하면 적어도 1이상의 값이 나와 최소한의 기울기를 가지므로 학습이 잘 되지 않는 현상 최소화.

x가 그대로 shortcut을 통해 연결되므로 shortcut 연결을 제외한 연산의 증가가 거의 없다.

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

x와 F의 dimension이 동일하지 않다면 Linear Projection을 적용 (W)

## Model




맨 왼쪽이 일반적인 VGG-19, 가운데가 residual을 적용하지 않은 plain한 34층 넷, 오른쪽이 residual를 적용한 34층 넷.

1. 34-layer Plain

- plain network의 설계는 일반적인 VGG의 convoltion 필터를 많이 모방. (3x3)
- 각 layer들은 같은 크기의 feature map output size를 만들고, 동일한 number of filters를 가짐, feature map size가 반으로 줄어든 경우, layer당 complexity를 유지하기 위해, number of filters를 2배로 늘림.

2. 34-layer residual

- plain network에 base를 두고 shortcut connection을 추가한 ResNet.
- Dimension이 증가할 때 shortcut connections에서 점선 부분은 다음 2가지 방법을 처리

- Identity shortcut connection을 계속 실행하는 경우, dimension을 증가시키기 위해 나머지를 zero padding. (The shortcut still performs identity mapping, with extra zero entries padded for increasing dimensions.)
- Projection shortcut connection은 dimension을 맞추기 위하여 1x1 convolution을 사용(The projection shortcut in Eqn.(2) is used to match dimensions (done by 1×1 convolutions))

## Implementation

### 3.4. Implementation

Our implementation for ImageNet follows the practice in [21, 41]. The image is resized with its shorter side randomly sampled in $[256, 480]$ for scale augmentation [41]. A $224 \times 224$ crop is randomly sampled from an image or its horizontal flip, with the per-pixel mean subtracted [21]. The standard color augmentation in [21] is used. We adopt batch normalization (BN) [16] right after each convolution and before activation, following [16]. We initialize the weights as in [13] and train all plain/residual nets from scratch. We use SGD with a mini-batch size of 256. The learning rate starts from 0.1 and is divided by 10 when the error plateaus, and the models are trained for up to $60 \times 10^4$ iterations. We use a weight decay of 0.0001 and a momentum of 0.9. We do not use dropout [14], following the practice in [16].

In testing, for comparison studies we adopt the standard 10-crop testing [21]. For best results, we adopt the fully-convolutional form as in [41, 13], and average the scores at multiple scales (images are resized such that the shorter side is in $\{224, 256, 384, 480, 640\}$).

구현에 정의된 파라미터들은 논문에 나와있음.

| 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 | | $1.8\times10^9$ | $3.6\times10^9$ | $3.8\times10^9$ | $7.6\times10^9$ | $11.3\times10^9$ |

각 layer에 대한 모델의 파라미터, 어떤 필터를 적용했는지 상세하게 나와있어 구현 시 코드 리뷰하기 매우 좋았습니다.

## 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.

ResNet 50/101/152에서는 Bottleneck Architectures를 사용하여 layer를 더 깊게 쌓을 수 있었고, Dimension이 바뀌는 부분에 대해서는 Projection Shortcut을 사용하였다.

## Result

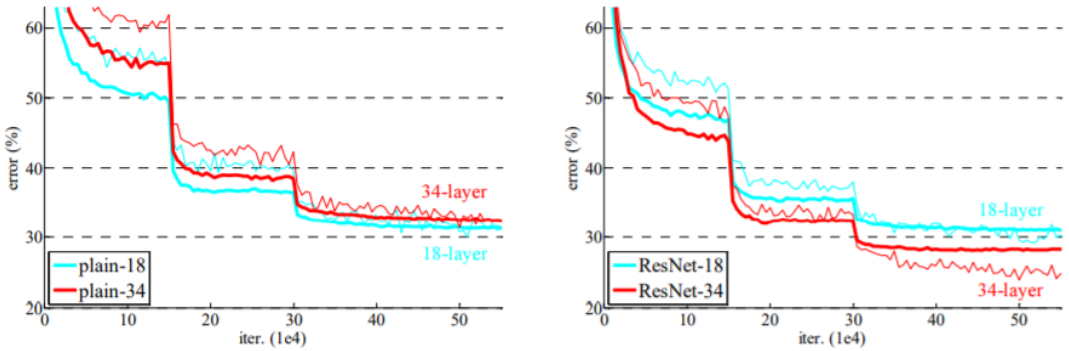



Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

일반적으로 plain보다 Resnet이 깊은 층에서도error가 더 낮은 것을 볼 수 있다.

| | 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.

더불어 같은 layer 갯수라도 ResNet이 더 좋은 성능을 낼 수 있음.

하지만 극단적으로 깊은 층(1202)에 대해서는 다시 error가 상승하는 것이 보임.

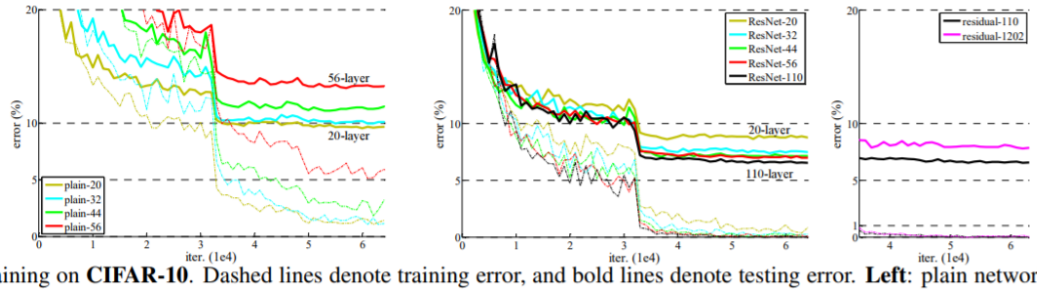



Figure 6. Training on **CIFAR-10**. Dashed lines denote training error, and bold lines denote testing error. **Left**: plain networks. The error of plain-110 is higher than 60% and not displayed. **Middle**: ResNets. **Right**: ResNets with 110 and 1202 layers.

## Code Review with Pytorch

https://gist.github.com/graykode/12025382299bc4912d1d8f5bcb68a71d

일반적으로 논문에 나왔던 ResNet-34를 보면 다음과 같이 구현

```
def ResNet34():
    # 사실상 다 더해보면 16이지만 16 * 2 + 2(인풋7x7, avg pool까지) = 34
    return ResNet(BasicBlock, [3,4,6,3])
```

`[3,4,6,3]` 은 Fig 5에서 나왔던 왼쪽 블록을 뜻함

| 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 | | $1.8\times10^9$ | $3.6\times10^9$ | $3.8\times10^9$ | $7.6\times10^9$ | $11.3\times10^9$ |

따라서 `BasicBlock` 은 다음과 같이 구현

```
 # BasicBlock는 _make_layer에서 2층으로 된 구조
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)

        # shortcut을 만든다.
        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != self.expansion*planes:
            # 페이퍼 ResNet34 fig3의 dot line 케이스
            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) # shorcut 더해줌
        out = F.relu(out)
```

```
        return out
```

```
if stride != 1 or in_planes != self.expansion*planes:
    # 페이퍼 ResNet34 fig3의 dot line 케이스
    self.shortcut = nn.Sequential(
        nn.Conv2d(in_planes, self.expansion*planes, kernel_size=1, stride=stride,
bias=False),
        nn.BatchNorm2d(self.expansion*planes)
    )
```

이 부분이 34-layer residual에서 말한 dot-line이다. (dimension이 바뀌는 경우)

일반 적인 ResNet 모델은 다음과 같음

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

        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)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1]*(num_blocks-1)
        print(strides)
        layers = []
        for stride in strides:
            print(self.in_planes, planes, stride)
            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
```

구글 코랩 : https://colab.research.google.com/drive/15rWBnnHB8qyut23LDp2rZ2xW_1uOZryf

## 요약...

VGG(2014)이 19개, GoogleNet(2014)이 22개층을 쌓았을때, ResNet(2015)에서 ResNet의 페이퍼의 연도와 더 깊은 층을 쌓았다는 점에서 혁신적인 모델, 2015년 ILSVRC에서 오류율 3.6%로 1등