파이썬으로 배우는 딥러닝(Deep Learning)

7회차 수업

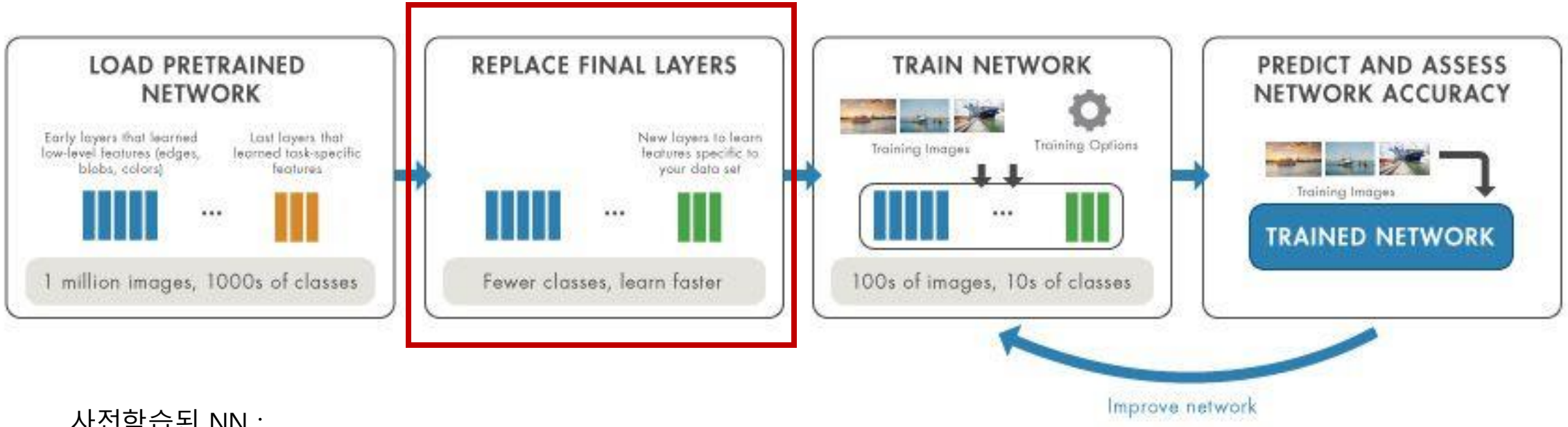
딥러닝을 더 빠르고 깊게 배우는 법: 전이학습과 ResNet

목 차

전이학습(Transfer Learning)

◆ 전이학습 의미 : **한 분야의 문제 해결을 위해 얻은 지식/정보를 다른 문제 해결에 사용**

◆ 딥러닝 분야의 전이학습 : 하나의 작업을 위해 학습된 신경망 모델을 유사한 다른 작업에 적용시키는 것을 의미함



사전학습된 NN :
- 1M개 이미지
- 1000개 클래스 분류

최종 Layer 교체

새로운 데이터/문제 학습
- 100개 이미지
- 10개 클래스 분류

예측 및 평가
NN 개선

https://kr.mathworks.com/discovery/transfer-learning.html

전이학습(Transfer Learning)

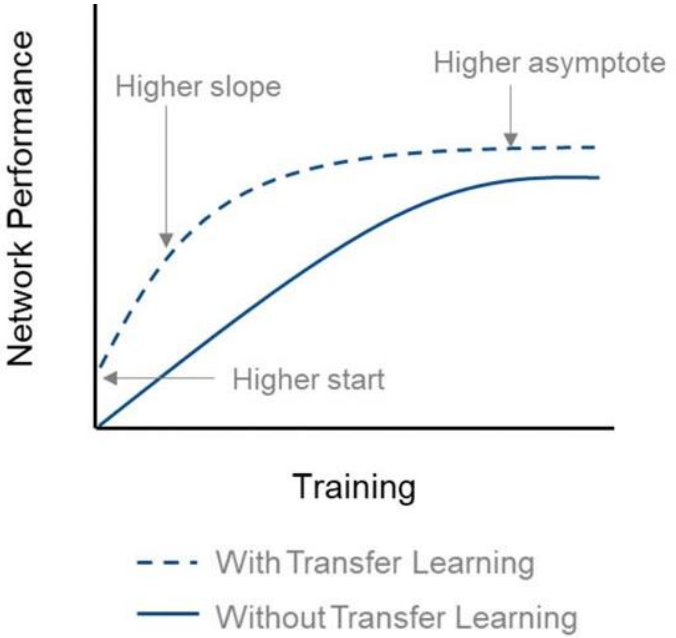
- CNN 모델은 **네트워크 깊이**에 따라 서로 **다른 종류의 feature들을 학습**함
- **전이학습**을 사용하여 새로운 데이터세트로 학습 → **깊은 계층의 파라미터를 조정** 가능
- 잘 훈련된 모델이 새로 구성하려는 모델과 유사한 경우, **짧은 시간 내에 높은 정확도** 달성 가능

얕은 layer

깊은 layer

← 일반적 특징 추출

구체적 특징 추출 →

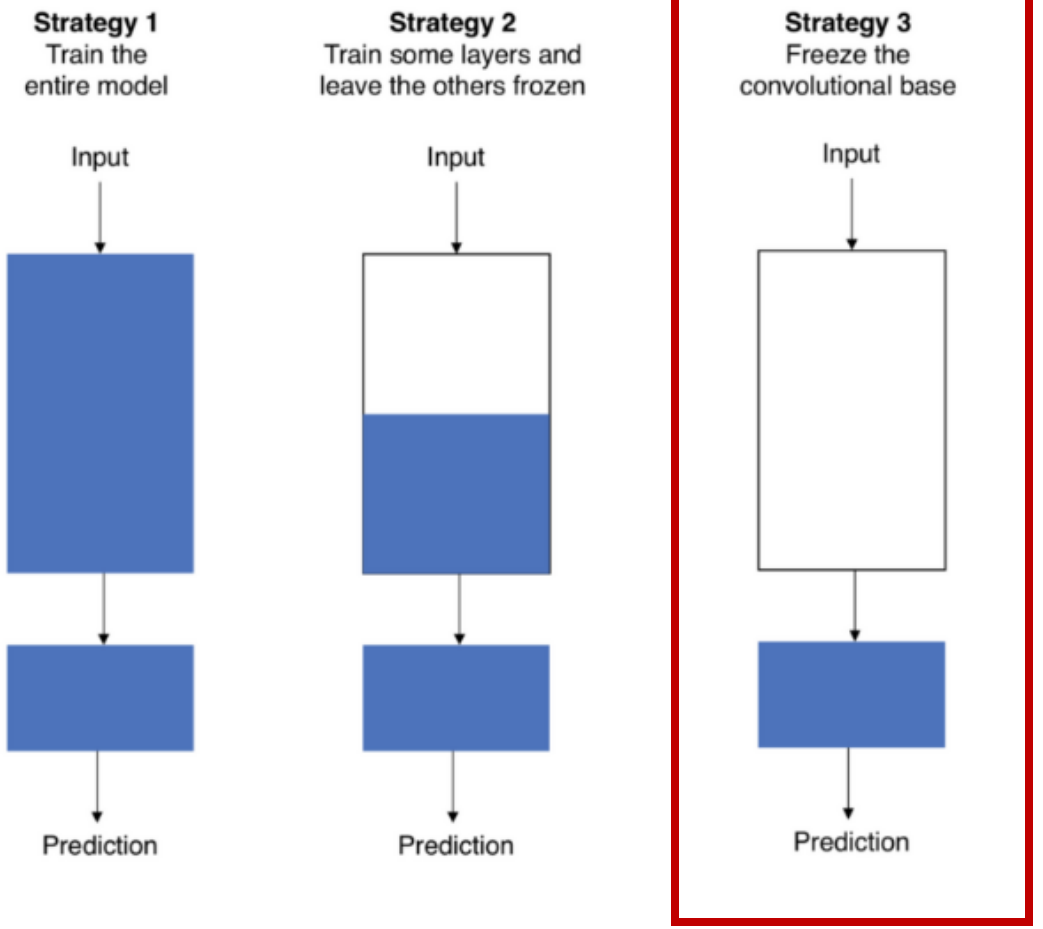


https://datahacker.rs/edge-detection/
https://kr.mathworks.com/discovery/transfer-learning.html

전이학습(Transfer Learning)

- Transfer Learning : 최종 Layer의 파라미터(가중치)를 조정(업데이트)
- Fine-Tunning : 모든 Layer의 파라미터(가중치)를 미세 조정 – 초기값은 사전 학습된 모델의 파라미터 사용

← Fine-tunning



Transfer Learning →

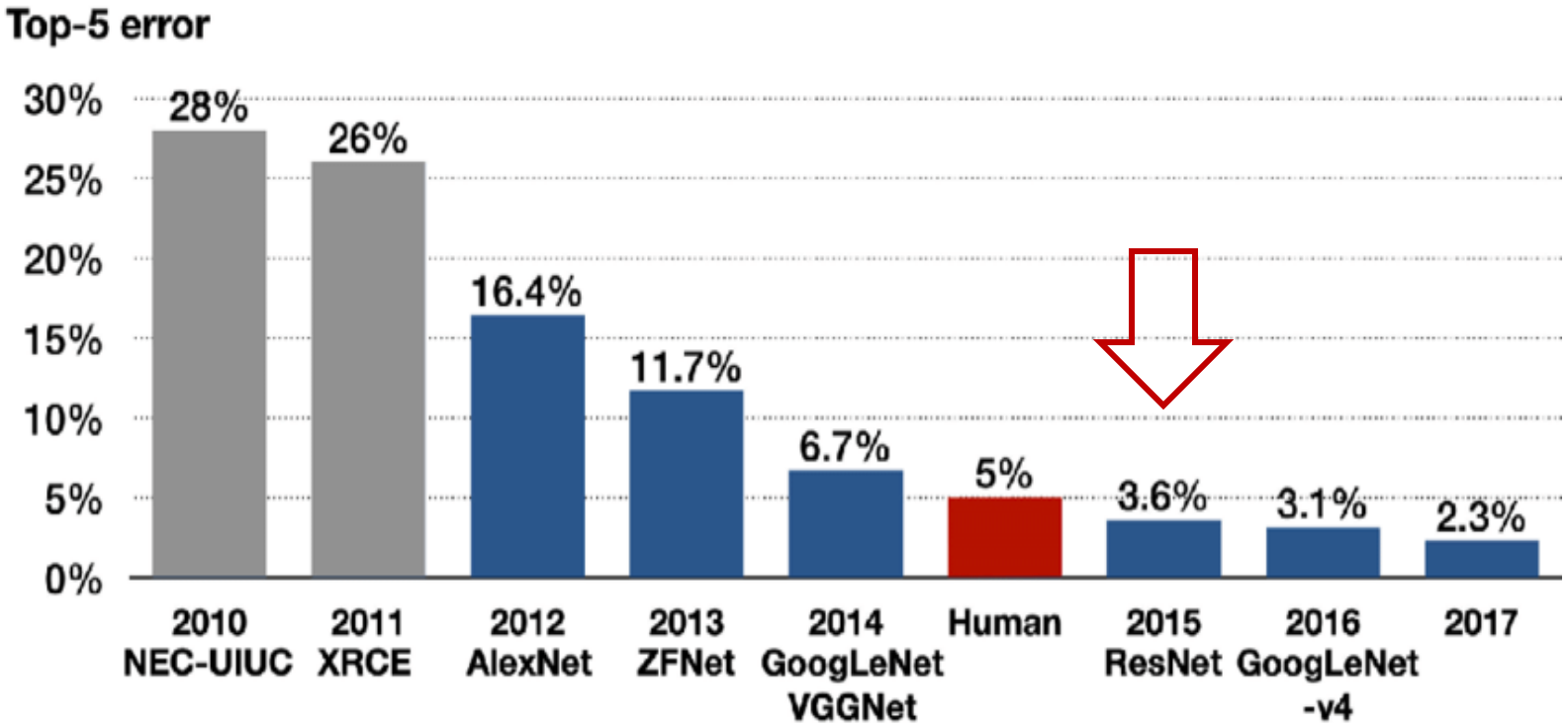
https://choice-life.tistory.com/40

목 차

사전 학습된 신경망 : ResNet

- **ILSVRC**(ImageNet Large Scale Visual Recognition Challenge) : 2010년에 시작된 ImageNet 인식 경진대회
- **ImageNet** : 1000가지 사물로 이루어진 100만개 이미지 데이터

ILSVRC 우승 모델 분류 에러율(%)



AlexNet :
- 깊은 신경망(8 layer) 사용.
- 인식 에러율을 16%로 낮춤

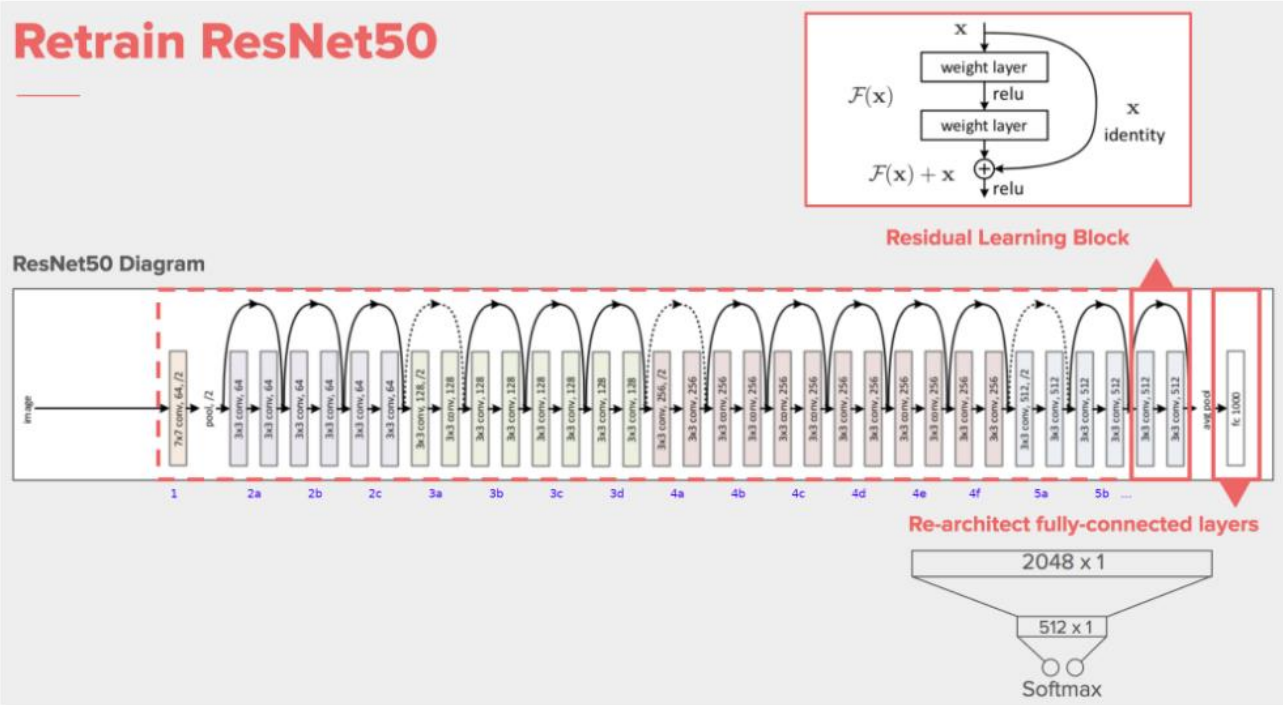
ResNet :
- 더 깊은 신경망(152 layer) 사용
- 인간의 인식 에러율보다 낮춤

https://www.researchgate.net/figure/Algorithms-that-won-the-ImageNet-Large-Scale-Visual-Recognition-Challenge-ILSVRC-in_fig2_346091812

사전 학습된 신경망 : ResNet50

◆ **ResNet50** : **50개의 layer**로 구성된 CNN

◆ ImageNet 데이터세트의 이미지(100만개 이상의 이미지)에 대해 사전 학습된 모델



- 다양한 이미지를 대표하는 많은 특징을 학습하여 키보드, 마우스, 연필, 각종 동물 등 **1000여개의 사물로 분류** 가능
- **224 x 224 크기**의 이미지를 입력으로 사용함

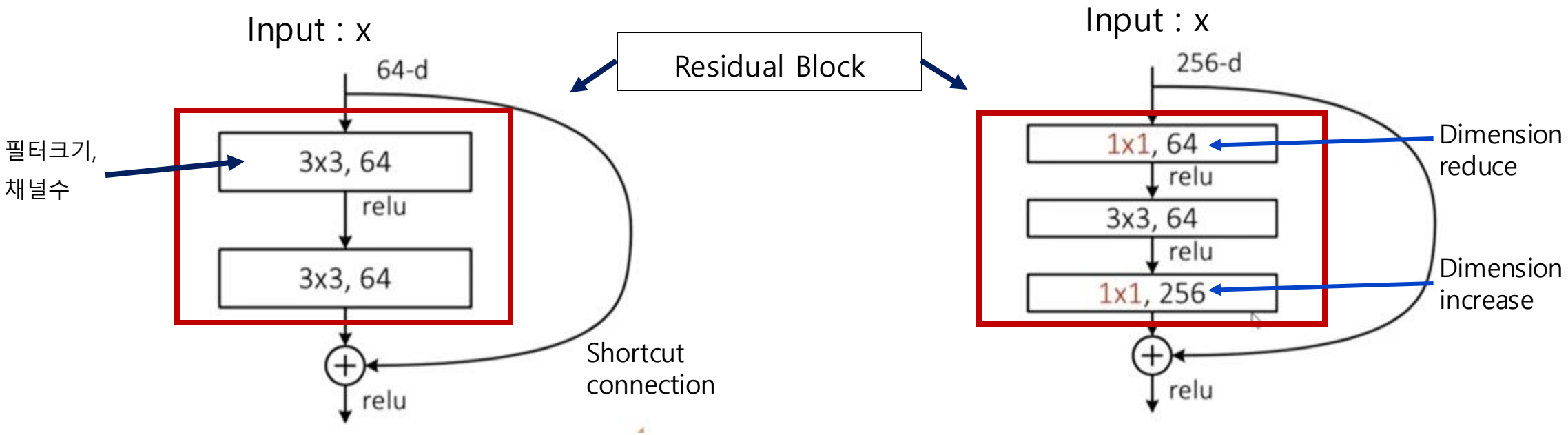
- 새로운 이미지 분류에 활용하기 위해 **Fully-Connected layer를 수정**해야 함
- **최종 출력**은 1000개에서 **2개로 수정**하여 전이학습에 사용 예정

https://stackoverflow.com/questions/54207410/how-to-split-resnet50-model-from-top-as-well-as-from-bottom#

ResNet – Residual Block, Bottleneck 구조

◆ **Vanishing Gradient 문제**를 피하기 위해 shortcut connection을 추가한 **Residual Block**을 사용

◆ 층을 깊게 쌓을 때 발생할 수 있는 성능저하(파라미터 수, 연산량 증가)를 막기 위해 **BottleNeck 구조**를 사용함(1x1 convolution을 사용하여 채널/차원 축소, 비선형성 증가, 과적합 방지).



Output : $relu(F(x) + x)$

BasicBlock
(ResNet 18, 34)

Output : $relu(F(x) + x)$

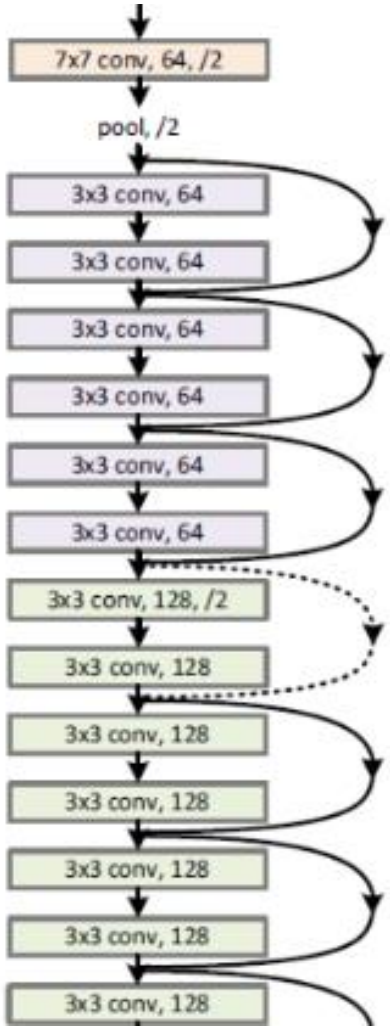
Bottleneck
(ResNet 50, 101, 152)

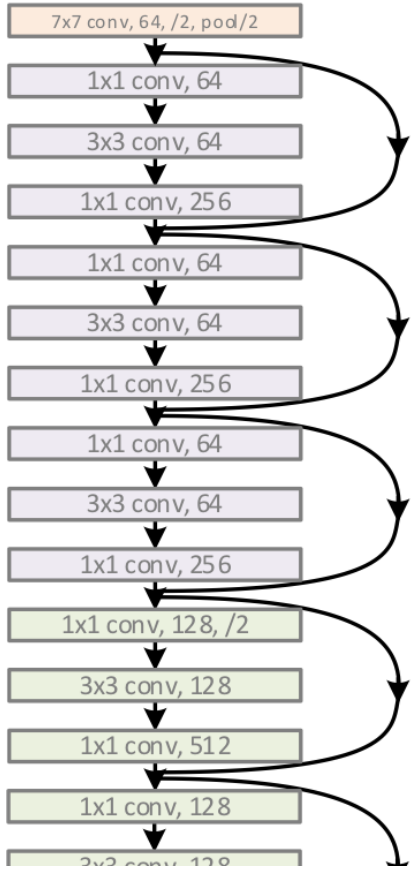
https://velog.io/@uonmf97/모두를-위한-딥러닝-시즌-2-ResNet

ResNet50 – Bottleneck 구조

◆ ResNet50은 ResNet34와 달리 bottleneck 구조가 적용됨.

ResNet34



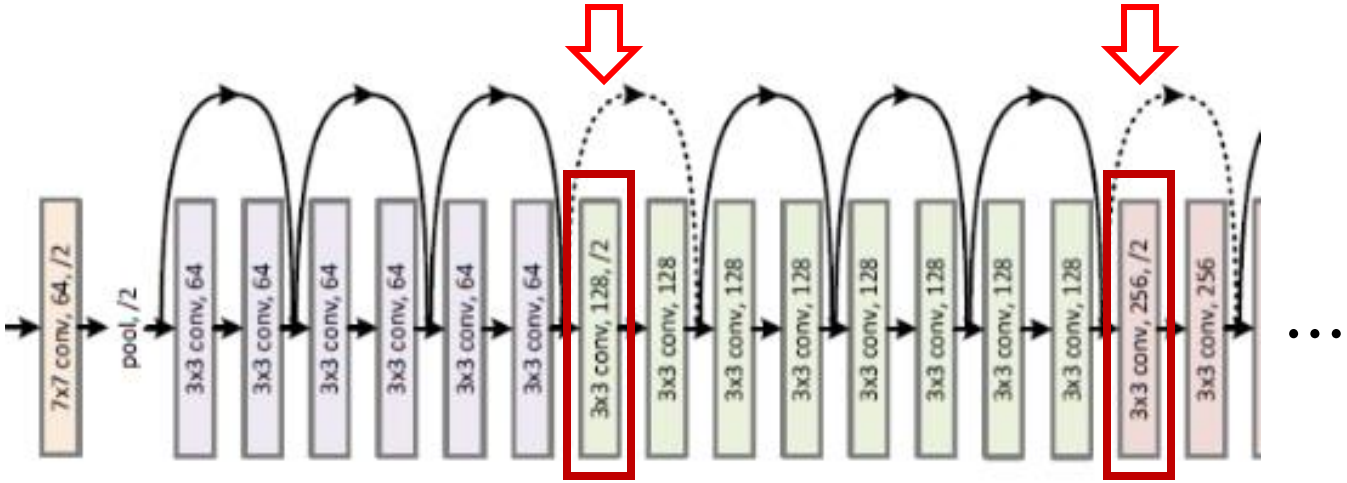


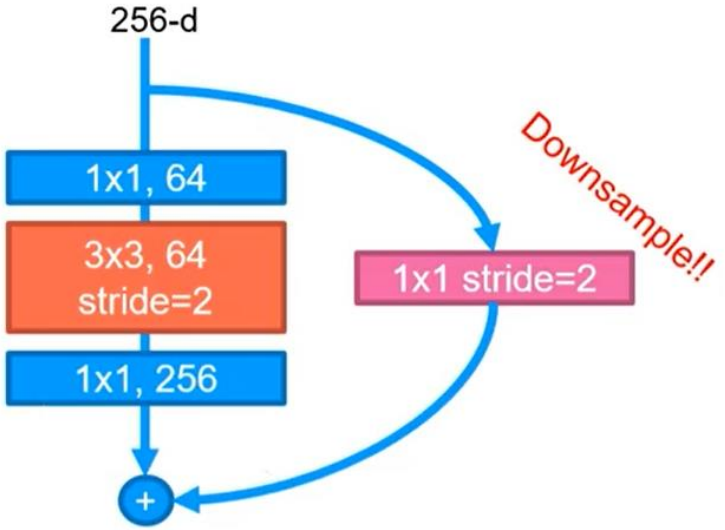
ResNet50

ResNet – Downsampling

◆ BasicBlock이나 Bottleneck 출력을 결정할때 $F(x)$ 와 x 의 텐서 사이즈가 다를때 Downsampling을 사용함

- Feature map 크기 변화 지점의 shortcut은 ResNet 그림에서 점선으로 표시
  - pooling에 해당하는 부분
  - **채널이 64→128로 128→256으로 변경될 때 feature map 크기가 줄어듦**
- ResNet50에서는 단순하게 처리하기 위해 stride 2인 1x1 convolution 사용

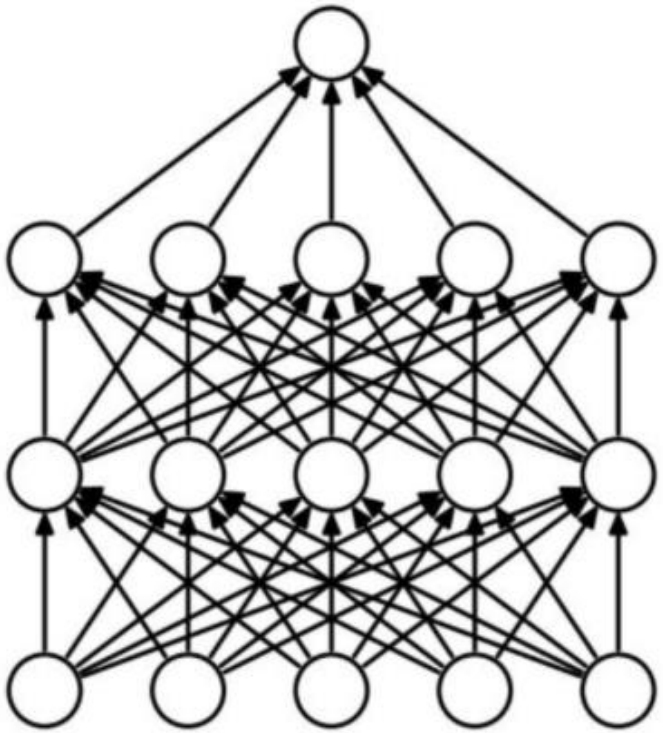




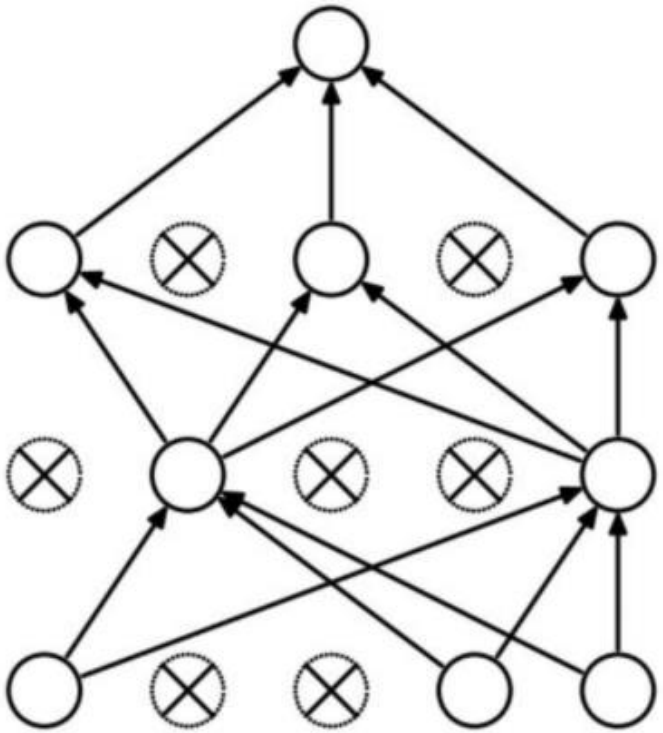
https://computistics.tistory.com/3 https://velog.io/@uonmf97/모두를-위한-딥러닝-시즌-2-ResNet

ResNet – Dropout

◆ 뉴런이라 부르는 노드를 무작위로 껐다 켰다를 반복하는 것을 Dropout이라 함.



(a) Standard Neural Net



(b) After applying dropout.

- Overfitting 방지
- 학습 시간 단축
- 매번 다른 형태의 노드로 학습하기 때문에 여러 형태의 네트워크들을 통해서 앙상블 효과 얻음

https://wegonnamakeit.tistory.com/46

ResNet50 – Layers

◆ 총 50개의 layer를 가지고 있으며, 각 conv에 있는 행렬(**[ ]**)이 residual block 임.

◆ ResNet50은 layer마다 다른 residual block 형태가 반복되어 학습됨.

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	[3×3, 64 3×3, 64] ×2	[3×3, 64 3×3, 64] ×3	[1×1, 64 3×3, 64 1×1, 256] ×3	[1×1, 64 3×3, 64 1×1, 256] ×3	[1×1, 64 3×3, 64 1×1, 256] ×3
conv3_x	28×28	[3×3, 128 3×3, 128] ×2	[3×3, 128 3×3, 128] ×4	[1×1, 128 3×3, 128 1×1, 512] ×4	[1×1, 128 3×3, 128 1×1, 512] ×4	[1×1, 128 3×3, 128 1×1, 512] ×8
conv4_x	14×14	[3×3, 256 3×3, 256] ×2	[3×3, 256 3×3, 256] ×6	[1×1, 256 3×3, 256 1×1, 1024] ×6	[1×1, 256 3×3, 256 1×1, 1024] ×23	[1×1, 256 3×3, 256 1×1, 1024] ×36
conv5_x	7×7	[3×3, 512 3×3, 512] ×2	[3×3, 512 3×3, 512] ×3	[1×1, 512 3×3, 512 1×1, 2048] ×3	[1×1, 512 3×3, 512 1×1, 2048] ×3	[1×1, 512 3×3, 512 1×1, 2048] ×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

ResNet50 – Residual Block 구성

1. conv1x1 -> [1x1, 64]

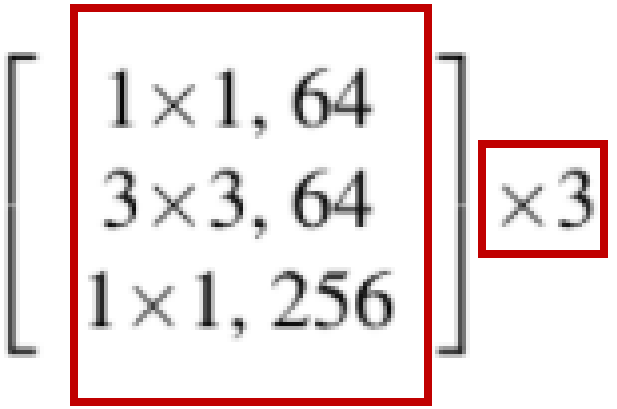
- Convolution layer
- kernel(filter) size : 1 x 1, kernel(filter) 개수는 64이다.

2. conv3x3 -> [3x3, 64]

- Convolution layer
- kernel(filter) size : 3 x 3, kernel(filter) 개수는 64이다.

3. conv1x1 -> [1x1, 256]

- Convolution layer
- kernel(filter) size : 1 x 1, kernel(filter) 개수는 256이다.



ResNet50 적용 방법

◆ 암석 이미지 식별 실습을 위해 **Python의 PyTorch**를 이용함.

◆ PyTorch의 라이브러리 중 하나인 **TorchVision**에서 ResNet50 등 사전 학습된 이미지 인식 모델 제공.

◆ TorchVision은 dataset, models, transforms 등 다양한 하위 라이브러리를 제공함.

torchvision.datasets

- MNIST
- ImageFolder
- DatasetFolder
- Imagenet-12
- PhotoTour
- Cityscapes
- KMNIST
- EMNIST
- FakeData
- COCO
- LSUN
- CIFAR
- STL10
- …

torchvision.models

- Alexnet
- VGG
- **ResNet**
- SqueezeNet
- DenseNet
- Inception v3
- GoogLeNet
- ShuffleNet v2
- MobileNetV2
- MobileNetV3
- ResNeXt
- Wide ResNet
- MNASNet
- …

torchvision.transforms

- Transforms on PIL Image
- Transforms on torch.*Tensor
- Conversion Transfoorms
- Generic Transform
- Functional Transforms
- …

ResNet50 적용 방법

- ResNet50 : torchvision.models의 ResNet에서 모델 제공

- ResNet (18, 34, 50, 101, 152) 사용 가능

- pretrained=True로 하이퍼파라미터 설정 → 사전 학습된 ResNet50 모델 사용.
  Pre-trained 모델을 사용하면 사전 학습된 가중치, 편향 등의 파라미터를 사용함.

```
model = models.ResNet50(pretrained=True)
```

- Input size가 다른 경우 ResNet50의 적용 : 이미지의 변환 필요
  ResNet50 모델의 입력은 224 x 224 이므로 torchvision.transforms.Compose()를 사용하여 변환

```
from torchvision import transforms
transform = transforms.Compose( [ transforms.RandomResizedCrop(224),
                                  transforms.Resize(224),
                                  transforms.ToTensor()
                                ] )
```

ResNet50 살펴보기 – 첫 convolution/pooling

◆ ResNet50의 layer 블록 별로 구성을 살펴본다.

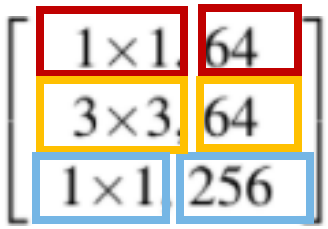
layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112			7×7, 64, stride 2		
				3×3 max pool, stride 2		

1. **kernel(filter) size** = 7x7, **kernel(filter) 개수** = 64, **stride** = 2
2. **Convolution**(Conv2d) → **Batch Normalization**(BatchNorm2d) → **Relu 함수**(ReLU) 순서로 진행.
3. **Max Pooling**(MaxPool2d) : 3x3 kernel에서 최댓값 선택, **stride** = 2 간격으로 움직임

print(model)

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
```

ResNet50 살펴보기 – layer1



→ 총 9개 layer

- Layer1에는 위의 bottleneck 3개로 구성
- 각 bottleneck마다 conv1 → bn1 → conv2 → bn2 → conv3 → bn3 → ReLU의 순서로 구성
- Convolution → Batch normalization → 활성화 함수(ReLU)
- Downsample : $f(x) + x$의 residual를 구현할 경우 $f(x)$와 x의 데이터 사이즈가 다른 경우 사용

```
(layer1): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (downsample): Sequential(
      (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (2): Bottleneck(
    (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
)
```

ResNet50 살펴보기 – layer2

$$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 4$$

-> 12개 layer

- Layer2에는 위의 bottleneck 구조를 4번 수행.
- Downsample : stride = (2, 2)를 사용

```
(layer2): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (2): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (3): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
)
```

ResNet50 살펴보기 – layer3

```
(layer3): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (downsample): Sequential(
      (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (2): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (3): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
```

```
  (4): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (5): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
)
```

$$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 6$$

-> 18개 layer

- Layer3에는 위의 bottleneck 구조 6개로 구성

ResNet50 살펴보기 – layer4

$$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$$ -> 9개 layer

- Layer4에는 $\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}$로 구성된 bottleneck 3개로 구성
- 여기까지 통과한 후의 피처맵 개수는 2048임.

```
(layer4): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (downsample): Sequential(
      (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (2): Bottleneck(
    (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
)
```

ResNet50 살펴보기 – Fully-Connected layer / **수정할 곳**

◆ **Fully Connected Layer** : 한 layer의 모든 뉴런이 그 다음 layer의 모든 뉴런과 연결된 구조를 의미함.
1차원 배열의 형태로 평탄화된 행렬을 통해 이미지를 분류.

→ Average Pooling(AdaptiveAvgPool2d)을 통해 output 사이즈를 (1, 1)로 평탄화 후, 활성화 함수로 Relu 적용.
Dropout 적용 후, 마지막 활성화 함수로 LogSoftmax 함수 사용

average pool, 1000-d fc, softmax

```
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=2048, out_features=1000, bias=True)
)
```

```
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Sequential(
    (0): Linear(in_features=2048, out_features=512, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.2, inplace=False)
    (3): Linear(in_features=512, out_features=2, bias=True)
    (4): LogSoftmax(dim=1)
  )
)
```

수정사항

(0) : 2048의 입력 → 512의 출력을 내는 layer 추가

(1) : Relu 함수 적용(활성화 함수)

(2) : Dropout 적용(과적합 방지, 성능 향상)

(3) : 최종 출력 feature 변경 : 2개

(4) : LogSoftmax 함수 사용 (성능 향상)