

3

Image Classification 2

강의 소개

이번 강의에서는 1강 Image Classification에 이어서 대표적인 CNN 모델들에 대해 배웁니다.

먼저 VGGNet과 비슷한 시기에 등장한 GoogLeNet을 시작으로, 지금도 많이 쓰이고 있는 ResNet에 공부하고 실습을 진행합니다.

이 외에도 추가적으로 몇가지 CNN 모델들에 대한 소개를 합니다.

끝으로 1강과 3강까지 다룬 4가지 모델 (AlexNet, VGGNet, GoogLeNet, ResNet)에 대하여 메모리 측면과 계산 효율 관점에서 비교 분석을 합니다.

Further Reading

- ResNet: <https://arxiv.org/pdf/1512.03385.pdf>

Problems with deeper layers

Going deeper with convolutions

The neural network is getting deeper and wider

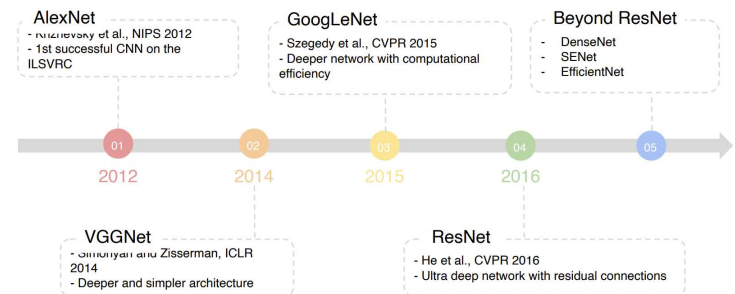
- Deeper networks learn more powerful features
 - ∴ Larger **receptive fields**
 - ∴ More **capacity and non-linearity**
- But, getting deeper and deeper always works better?

Hard to optimize

Deeper networks are harder to optimize

- Gradient vanishing / exploding
- Computationally complex
 - ∴ 깊게 쌓을수록 계산 복잡도 ↑
- Degradation problem
 - parameter 수 ↑라서 overfitting 문제로 예상했지만, 사실 degradation 문제

CNN architectures for image classification 2

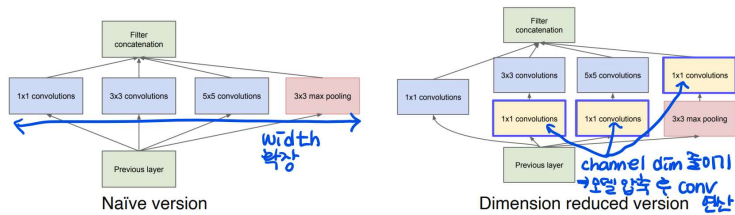


GoogLeNet

Inception module

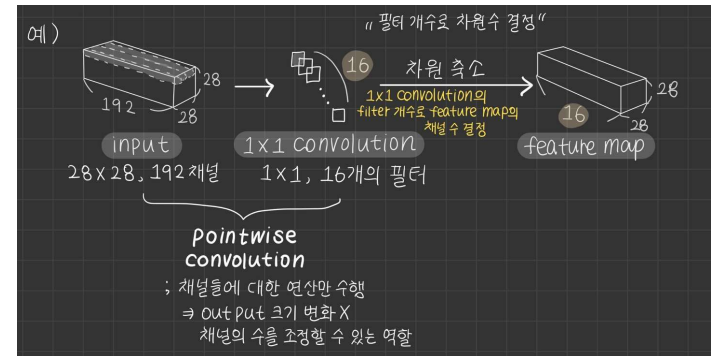
- Apply multiple filter operations on input activation from the previous layer
 - → 하나의 layer에 다양한 크기의 conv filter를 사용 ⇒ 여러 측면의 activation 관찰 (= width 확장)
 - 1x1, 3x3, 5x5 **convolution filters**

- 3x3 **pooling** operation
- **Concatenate** all filter outputs together **along the channel axis**
 - ▼ **channel-wise concatenate**
 - 각 feature map들의 결과는 그 크기를 하나로 통일되게 설계해야 Concat 연산이 가능해진다.
 - 참고 : [CNN 모델 탐구 \(3\) - GoogLeNet : 네이버 블로그 \(naver.com\)](#)
- The increased network size increases the use of **computational resources**
 - ∴ 하나의 layer에 여러 개의 filter 존재 → 계산 복잡도 ↑
 - Use **1x1 convolutions** → **bottleneck layer** ⇒ the number of **channels** ↓

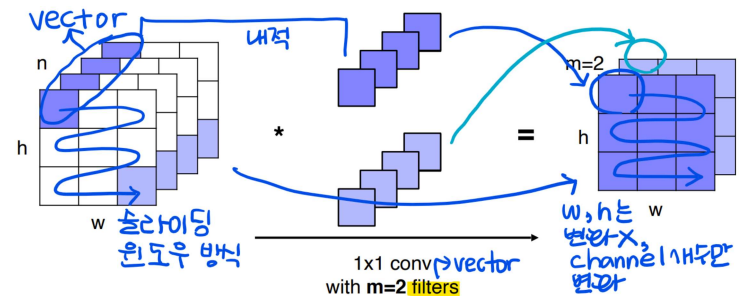


1x1 convolutions

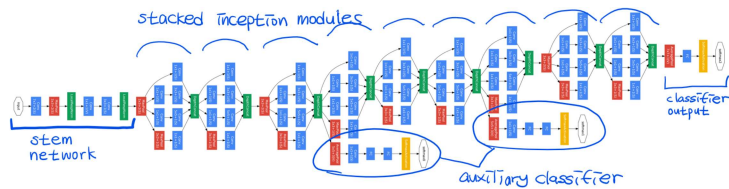
- ▼ **1x1 convolution**



- 1x1 convolution의 장점
 1. Channel 수 조절 → 1x1 convolution의 filter 개수 = feature map의 channel 수
 2. 연산량 감소(Efficient)
 3. 비선형성(Non-linearity)
- 참고 : [1x1 convolution이란, ∴ 대학원생이 쉽게 설명해보기 \(tistory.com\)](#)
- 그림 참고 : [\[DL\] 1x1 convolution은 무엇이고 왜 사용할까? | Sociological Imagination \(euneestella.github.io\)](#)

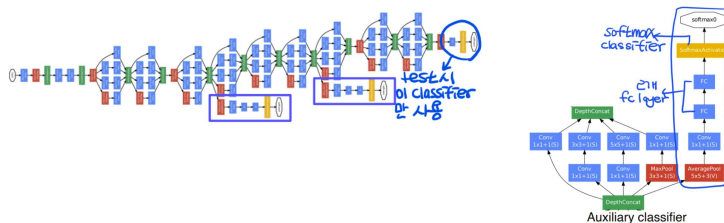


Overall architecture



- **stem network**
 - vanilla convolution networks
- **Stacked inception modules**
- **Auxiliary classifiers**
 - gradient를 주입 → gradient vanishing 문제 해결
- **Classifier output**
 - a single FC layer → 하나의 fc layer로 softmax score 산출

Auxiliary classifier



- lower layer에 additional gradient 주입 → vanishing gradient 문제 해결
- training에서만 사용 → test에서는 사용 X

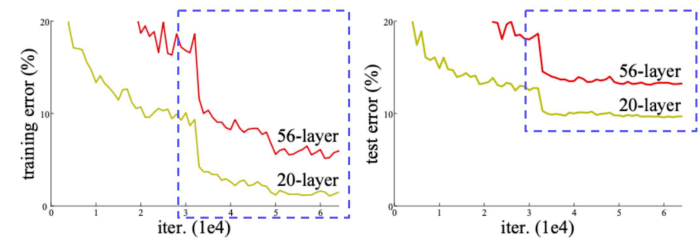
ResNet

Revolutions of depth

- building ultra-deeper than any other networks

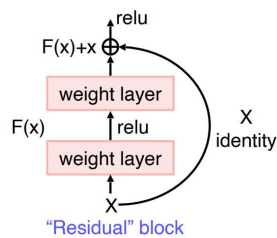
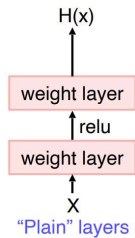
Degradation problem

- depth of network $\uparrow \rightarrow$ accuracy (rapidly) \downarrow
 - not by **overfitting** but by **optimization**
 - overfitting으로 설명하려면, training error에서는 56-layer가 20-layer 보다는 낮았어야
 - optimization 문제 \rightarrow gradient vanishing / exploding 관련



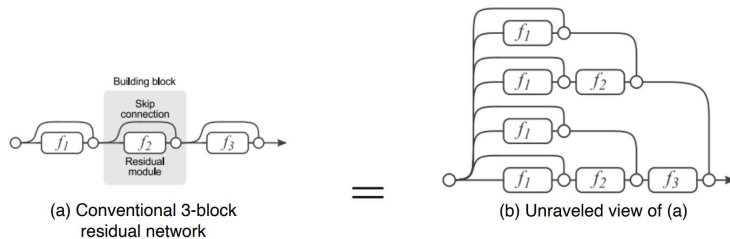
Hypothesis

- Plain layer
 - As the layers get **deeper** $\uparrow \rightarrow$ **hard** to learn good $H(x)$ directly (?)
- Residual block
 - **Target function**
 - $H(x) = F(x) + x \rightarrow$ Shortcut connection
 - **Residual function**
 - $F(x) = H(x) - x$

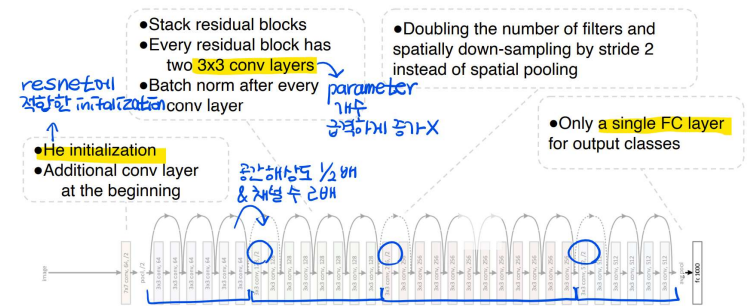


Analysis of residual connection

- gradient는 (plain layers에 비해) 비교적 shorter path를 이동 → gradient vanishing 문제 해결
- Residual network의 connection 경로는 $O(2^n)$ 의 경우의 수 존재 → block 하나가 추가될 때마다 경우의 수 2배씩 증가 ⇒ residual block n개일 때, connection 경우의 수 2^n



Overall architecture



stride 2로 spatially down sampling 하는 이유

- downsampling : 이미지의 크기를 줄이는 과정
- 참고 : [08. Downsampling \(tistory.com\)](https://tistory.com/08/Downsampling)

- Convolutional Neural Network에서 feature의 resolution을 줄일 때, **stride=2** 또는 **max/average pooling** 을 이용하여 resolution을 1/2로 줄이는 방법을 많이 사용합니다.
- **convolution layer** 를 이용하여 **stride = 2** 로 줄이면 학습 가능한 파라미터가 추가되므로 학습 가능한 방식으로 resolution을 줄이게 되나 그만큼 파라미터의 증가 및 연산량이 증가하게 됩니다.
 - feature를 뽑기 위한 Convolution Layer와 Downsampling을 위한 stride를 동시에 적용할 수 있습니다. 이 경우 같은 3 x 3 크기의 필터를 사용하더라도 stride가 적용되기 때문에 더 넓은 **receptive field**를 볼 수 있습니다.
- 반면 **pooling** 을 이용하여 resolution을 줄이게 되면 학습과 무관해지며 학습할 파라미터 없이 정해진 방식 (max, average)으로 resolution을 줄이게 되어 연산 및 학습량은 줄어들지만 **convolution with stride** 방식보다 성능이 좋지 못하다고 알려져 있습니다.
 - layer를 줄여서 gradient 전파에 초점을 두려고 할 때 pooling을 사용하는게 도움이 될 수 있습니다.

- 참고 : [Stride와 Pooling의 비교 - gaussian37](#)

▼ He initialization

- He 초기화(He Initialization)는 **ReLU**를 활성화 함수로 사용할 때 추천되는 초기화 방법입니다.
- 참고 : [가중치 초기화 \(Weight Initialization\) - Data Science \(yngie-c.github.io\)](#)

▼ Batch normalization 하는 이유

- 배치 정규화는 초기 가중치 설정 문제와 비슷하게 **가중치 소멸 문제 (Gradient Vanishing) 또는 가중치 폭발 문제(Gradient Exploding)**를 해결하기 위한 접근 방법 중 하나이다.
- Batch normalization 하는 이유
 - 학습 속도가 개선된다 (학습률을 높게 설정할 수 있기 때문)
 - 가중치 초기값 선택의 의존성이 적어진다 (학습을 할 때마다 출력값을 정규화하기 때문)
 - 과적합(overfitting) 위험을 줄일 수 있다 (드롭아웃 같은 기법 대체 가능)
 - Gradient Vanishing 문제 해결
- 참고 : [문과생도 이해하는 딥러닝 \(10\) - 배치 정규화 \(tistory.com\)](#)

PyTorch code for ResNet

```
return _resnet('resnet18', BasicBlock, [2, 2, 2, 2], pretrained, progress, **kwargs)
# BasicBlock : resnet18의 기본 residual block
# [2, 2, 2, 2] : 각 layer에 몇 개의 residual block이 존재하는지
```

```
def _forward_impl(self, x: Tensor):
    # conv1
    x = self.conv1(x)
    x = self.bn1(x)
    x = self.relu(x)
    x = self.maxpool(x)
```

```
# residual block 쌓기
x = self.layer1(x) # self.layer1 = self._make_layer(block=2, 64, layers[0])
x = self.layer2(x) # self.layer2 = self._make_layer(block=2, 128, layers[1],
x = self.layer3(x) # self.layer3 = self._make_layer(block=2, 256, layers[2],
x = self.layer4(x) # self.layer4 = self._make_layer(block=2, 512, layers[3],

# fc layer
x = self.avgpool(x) # self.avgpool = nn.AdaptiveAvgPool2d((1, 1)) # 벡터화
x = torch.flatten(x, 1) # self.fc = nn.Linear(512 * block.expansion, num_classes)
x = self.fc(x)

return x
```

```
def _make_layer(self, block: Type[Union[BasicBlock, Bottleneck]], planes: int,
                blocks: int, stride: int=1, dilate: bool=False) -> nn.Sequential:
    norm_layer = self._norm_layer
    downsample = None
    previous_dilation = self.dilation
    if dilate:
        self.dilation *= stride
        stride = 1
    if stride != 1 or self.inplanes != planes * block.expansion:
        downsample = nn.Sequential(
            conv1x1(self.inplanes, planes * block.expansion, stride),
            norm_layer(planes * block.expansion),
        )
    layers = []
    layers.append(block(self.inplanes, planes, stride, downsample, self.groups,
                        self.base_width, previous_dilation, norm_layer))
    self.inplanes = planes * block.expansion
    for _ in range(1, blocks):
        layers.append(block(self.inplanes, planes, groups=self.groups,
                            base_width=self.base_width, dilation=self.dilation,
                            norm_layer=norm_layer))
    return nn.Sequential(*layers)
```


▼ dilate=replace_stride_with_dilation ?

Beyond ResNet

DenseNet

- 각 layer의 모든 output은 channel axis를 기준으로 **concatenate** → channel 개수 늘어남 (cf. ResNet은 **summation**)
 - alleviate **vanishing gradient problem**
 - strengthen **feature propagation**
 - encourage **the reuse of features**

SeNet

- Attention across channels
- Recalibrates channel-wise responses by modeling interdependencies between channels
- Squeeze and excitation operations
 - **Squeeze**
 - capturing **distributions of channel-wise** responses by **global average pooling**
 - ▼  Squeeze
 - 각 채널별 가중치를 계산하기 위해서는 우선, 각 채널을 1차원으로 만들어야 합니다. 예를 들어, 3채널이 있으면 [0.6, 0.1, 0.7]로 표기를 해야 가중치를 나타낼 수 있습니다. **Squeeze**는 각 채널을 1차원으로 만드는 역할(압축)을 합니다.

$$z_c = \mathbf{F}_{sq}(\mathbf{u}_c) = \frac{1}{H \times W} \sum_{i=1}^H \sum_{j=1}^W u_c(i, j).$$

- Squeeze는 conv 연산을 통해 생성된 피쳐맵을 입력으로 받습니다. $H \times W \times C$ 크기의 피쳐맵을 **global average pooling 연산**을 통해 $(1 \times 1 \times C)$ 로 압축합니다. 피쳐맵의 한 채널에 해당하는 픽셀 값을 모두 더한 다음에, $H \times W$ 로 나누어 $1 \times 1 \times 1$ 로 압축합니다. 피쳐맵은 C 개의 채널을 갖고 있으므로 다 연결하면 $(1 \times 1 \times C)$ 가 됩니다.

- 생성된 $(1 \times 1 \times C)$ 벡터는 Excitation으로 전달됩니다.
- 참고 : [논문 읽기] [SENet\(2018\) 리뷰](#), [Squeeze-and-Excitation Networks \(tistory.com\)](#)

Excitation

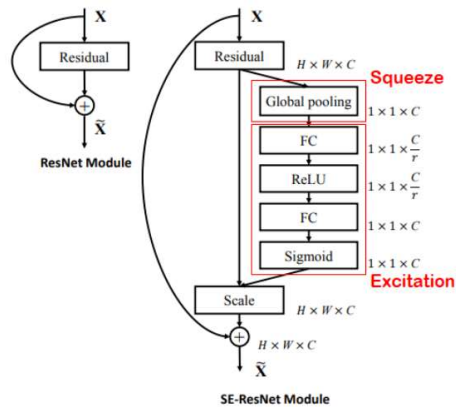
- gating channels by **channel-wise attention weights** obtained by a **FC layer**

Excitation

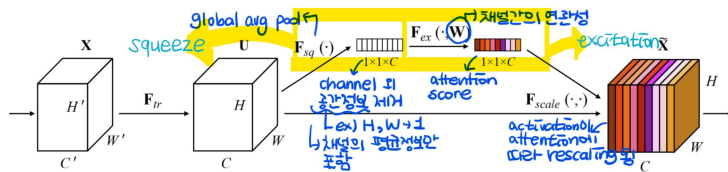
- **Excitation**은 Squeeze에서 생성된 $(1 \times 1 \times C)$ 벡터를 정규화하여 가중치를 부여하는 역할을 합니다.
- Excitation은 **FC1 - ReLU - FC2 - Sigmoid**로 구성됩니다. FC1에 $(1 \times 1 \times C)$ 벡터가 입력되어, C 채널을 C/r 개 채널로 축소합니다. r 은 하이퍼파라미터입니다.
- 연산량 제한과 일반화 효과 때문에 bottleneck 구조를 선택했다고 하네요. C/r 개 채널로 축소되어 $(1 \times 1 \times C/r)$ 가 된 벡터는 ReLU로 전달되고, FC2를 통과합니다. FC2는 채널 수를 다시 C 로 되돌립니다. 그리고 Sigmoid를 거쳐서 $[0 \sim 1]$ 범위의 값을 지니게 됩니다. 마지막으로, 피쳐맵과 곱해서 피쳐맵의 채널에 가중치를 가합니다.
- 참고 : [논문 읽기] [SENet\(2018\) 리뷰](#), [Squeeze-and-Excitation Networks \(tistory.com\)](#)
- **depth + or connection 방법 변화** → 현재 주어진 activation 간의 관계를 명확히 ⇒ channel 간의 관계 modeling + attention weight 파악

그림 설명

- SB Block(Squeeze(압축) + Excitation(재조정))을 통해 **채널별 가중치**를 계산하고 **피쳐맵에 곱해지는 모습**을 나타냅니다. 색으로 표현된 가중치가 피쳐맵과 곱해서 피쳐맵의 색도 바뀌었네요.

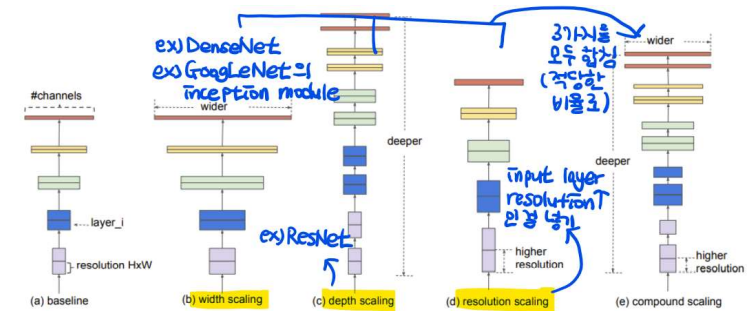


- 참고 : [논문 읽기] SENet(2018) 리뷰, Squeeze-and-Excitation Networks (tistory.com).



EfficientNet

- deep + wide + high resolution network in an efficient way
- 적은 FLOP으로도 성능 좋음



FLOP

- 딥러닝에서의 FLOPS는 단위 시간이 아닌 절대적인 연산량 (곱하기, 더하기 등)의 횟수를 지칭합니다.

- 참고 : FLOPS (FLoating point OPerationS) - 플롭스 (tistory.com).

width scaling VS. depth scaling VS. resolution scaling

- 채널의 개수를 늘리는 width scaling
- 모델의 layer를 바꾸는 depth scaling
- Input 이미지의 해상도(= 이미지 크기)를 바꾸는 resolution scaling
- 참고 : 논문 리뷰 EfficientNet : 스케일링으로 모델의 성능 높이기 :: DOOWN (tistory.com).

compound scaling

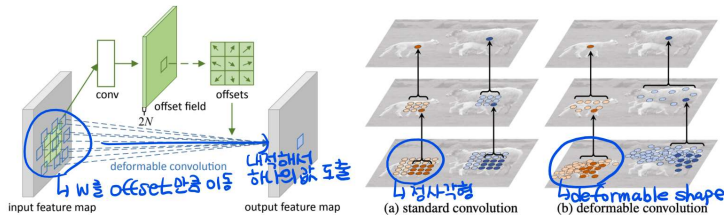
- 저자들은 아래의 조건식으로 모델의 성능을 향상시킬 수 있는 최적의 Width, Depth, Resolution scaling이 가능하다고 제안합니다.

$$\begin{aligned}
 \text{depth: } d &= \alpha^\phi \\
 \text{width: } w &= \beta^\phi \\
 \text{resolution: } r &= \gamma^\phi \\
 \text{s.t. } \alpha \cdot \beta^2 \cdot \gamma^2 &\approx 2 \\
 \alpha \geq 1, \beta \geq 1, \gamma &\geq 1
 \end{aligned}$$

- 위 식에서 α, β, γ 를 설정한 뒤 ϕ 를 컴퓨팅 리소스에 맞추어 늘려주면 뛰어난 성능의 모델을 얻을 수 있습니다.
- 참고 : [논문 리뷰] [EfficientNet: Rethinking Model Scaling For Convolutional Neural Networks \(tistory.com\)](https://tistory.com).

Deformable convolution

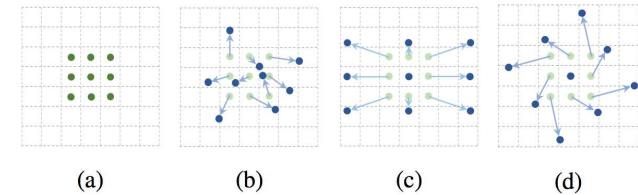
- 2D spatial offset prediction for irregular convolution
- Irregular grid sampling with 2D spatial offsets
- Implemented by **standard CNN** and grid sampling with **2D offsets**



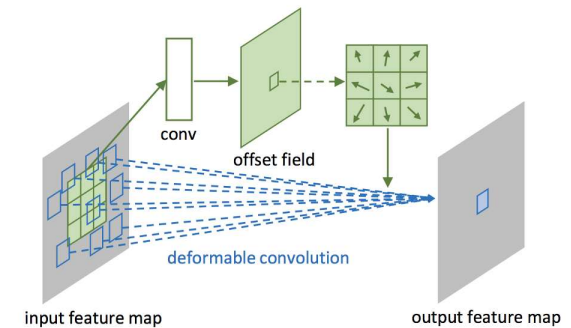
Deformable convolution

- 여러 연산(convolution, pooling, RoI pooling 등)이 **기하학적으로 일정한 패턴**을 가정하고 있기 때문에 **복잡한 transformation에 유연하게 대처하기 어렵다**는 것입니다. 저자들은 그 예로 CNN layer에서 사용하는 **receptive field**의 크기가 항상 같고, object detection에 사용하는 feature를 얻기 위해 사람의 작업이 필요한 점 등을 들고 있습니다.

- Deformable Convolution**은 아래 그림처럼 convolution에서 사용하는 **sampling grid에 2D offset을 더한다**는 아이디어에서 출발합니다.



- 그림 (a)의 초록색 점이 일반적인 convolution의 sampling grid입니다. 여기에 offset을 더해(초록색 화살표) (b)(c)(d)의 푸른색 점들처럼 다양한 패턴으로 변형시켜 사용할 수 있습니다.



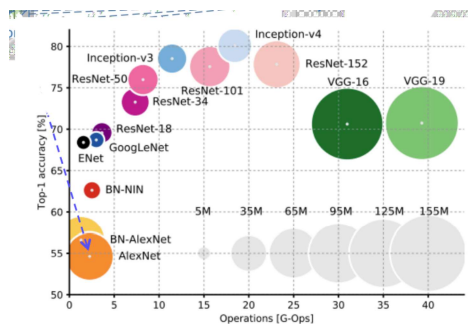
- 그림에서 보는 것처럼 deformable convolution에는 일반적인 convolution layer 말고 하나의 convolution layer가 더 있습니다. 그림에서 **conv** 라는 이름이 붙은 이 초록색 layer가 **각 입력의 2D offset을 학습하기 위한 것**입니다. 여기서 **offset**은 integer 값이 아니라 fractional number이기 때문에 0.5 같은 **소수 값이 가능하며, 실제 계산은 linear interpolation (2D이므로 bilinear interpolation)으로 이뤄 집니다.**

- Training 과정에서, output feature를 만드는 convolution kernel과 offset을 정하는 convolution kernel을 동시에 학습할 수 있습니다.
- 참고 : [Deformable Convolutional Networks · Pull Requests to Tomorrow \(jamiekang.github.io\)](https://github.com/jamiekang/Deformable-Convolutional-Networks).

- Moderate efficiency (depending on the model)
 - inception (ex. GoogLeNet) 계열에 비해 모델이 크고 느림

Summary of image classification

Summary of image classification



AlexNet

- simple CNN architecture
- Simple computation, but heavy memory size
- Low accuracy

VGGNet

- simple with `3x3 convolutions`
- Highest memory, the heaviest computation → 속도 느림

GoogLeNet

- `inception module` and `auxiliary classifier`

ResNet

- deeper layers with `residual blocks`

CNN backbones

- AlexNet, VGG, ResNet에 비해 `GoogLeNet` is the most efficient CNN model
 - **complicated** to use
- `VGGNet` and `ResNet` are typically used as a backbone model for many tasks
 - **simple** `3x3 conv layers`

Reference

2. CNN architectures for image classification 2

- Szegedy et al., Going Deeper with Convolution, CVPR 2015
- He et al., Deep Residual Learning for Image Recognition, CVPR 2015
- Veit et al., Residual Networks Behave Like Ensembles of Relatively Shallow Networks, NIPS 2016
- Huang et al., Densely Connected Convolutional Networks, CVPR 2017
- Hu et al., Squeeze-and-Excitation Networks, CVPR 2018
- Tan and Le, EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks, ICML 2019
- Dai et al., Deformable Convolutional Networks, ICCV 2017

3. Summary of image classification

- Canziani et al., An Analysis of Deep Neural Network Models for Practical Applications, CVPR 2016