YOLOv4

YOLOv4: Optimal Speed and Accuracy of Object Detection

발표자 : 진아람

Index

Introduction

Improvement of YOLO Algorithm

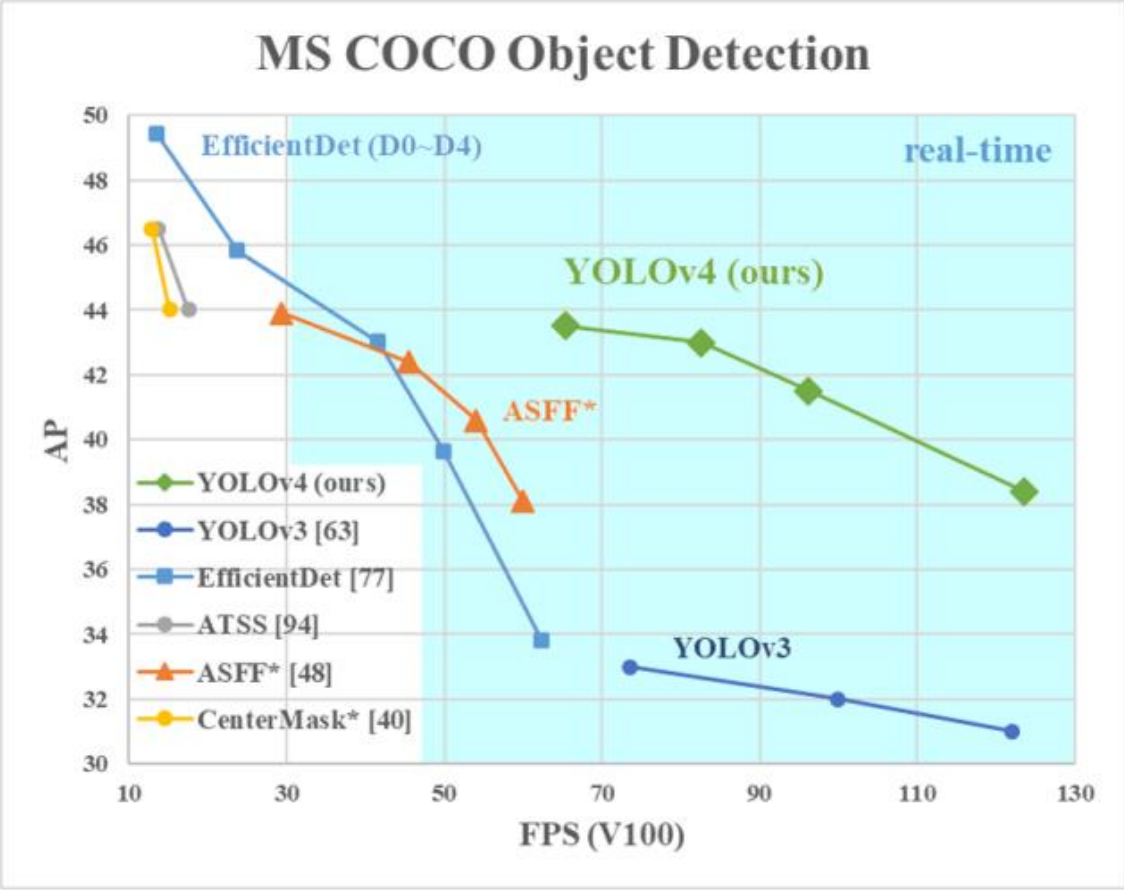
YOLOv1 → YOLO9000 → YOLOv3 → YOLOv4 → YOLOv5 → PP-YOLO

YOLO v1 : 가깝게 붙어 있는 물체를 잘 탐지 하지 못함
작은 물체를 잘 탐지하지 못함

YOLO v2 : 빠른 속도와 괜찮은 정확도를 가졌지만
작은 물체와 겹치는 물체들을 효과적으로 Localization 하지 못함

YOLO v3 : 빠른 속도와 괜찮은 정확도를 가졌지만
상대적으로 낮은 정확도(특히, 작은 object에 대해)

YOLOv4: Optimal Speed and Accuracy of Object Detection



✓ GOAL

누구나 실시간 고품질의

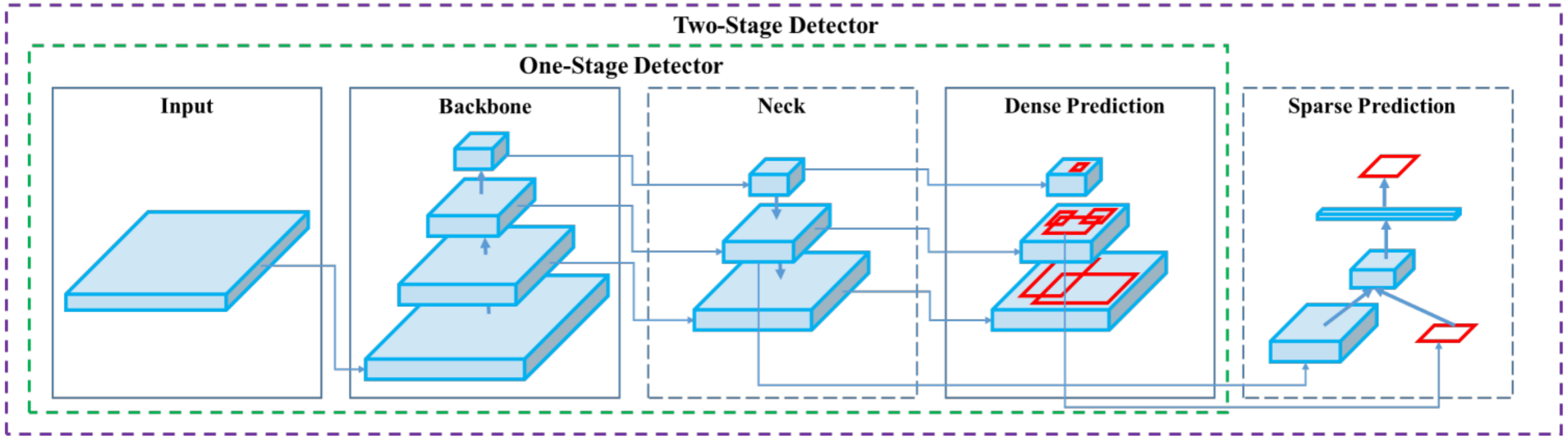
결과를 얻을 수 있도록,

빠른 속도로 동작하는

object detector를 고안

Related work

2.1. Object detection models



Input: { Image, Patches, Image Pyramid, … }

Backbone: { VGG16 [68], ResNet-50 [26], ResNeXt-101 [86], Darknet53 [63], … }

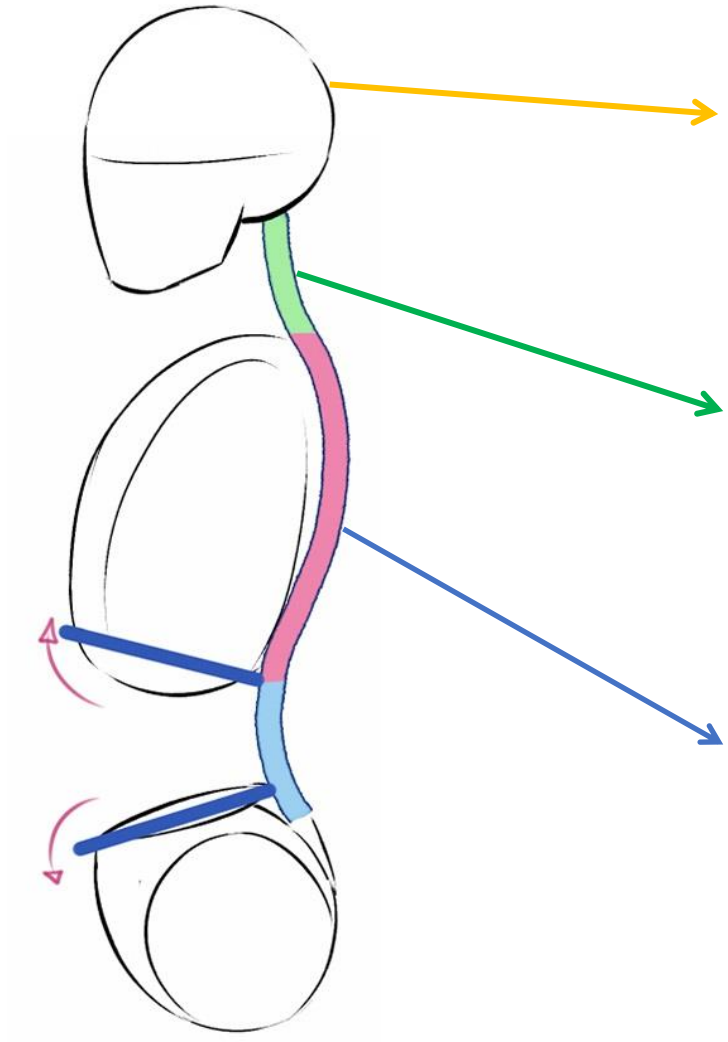
Neck: { FPN [44], PANet [49], Bi-FPN [77], … }

Head:

Dense Prediction: { RPN [64], YOLO [61, 62, 63], SSD [50], RetinaNet [45], FCOS [78], … }

Sparse Prediction: { Faster R-CNN [64], R-FCN [9], … }

- YOLOv4 consists of



Backbone - Neck - Head

Head: **YOLOv3**

Class와 Bounding box를 예측

Neck: **SPP , PAN**

서로 다른 scale의 feature map을 수집하는 layer

Backbone: **CSPDarknet53**

ImageNet을 이용하여 pre-trained된 backbone network

Bag of Freebies & Bag of Specials

2.2. Bag of Freebies

BoF

Bag of Freebies (pre-processing + training strategy)

Training Phase

- Call methods that only change the **training** strategy or only increase the **training cost** as “BoF”

✓ **훈련 전략만 바꾸거나 훈련 비용만 증가시켜, object detector의 정확도를 높이는 방법들을 의미**

- **Inference 시간을 늘리지 않으면서 더 높은 정확도를 얻을 수 있도록 하는 방법.**
- **Data Augmentation, Loss function, Regularization 등 학습에 관여하는 요소로, training cost를 증가시켜 정확도를 높이는 방법**

Ex) Data Augmentation, Regularization, loss function etc…

2.3. Bag of Specials

BoS

architecture related

Bag of Specials (plugin modules + post-processing)

Inference Phase

- Call methods that only increase the **inference cost** but can improve the accuracy as "BoS"

✓ **추론 비용을 약간만 증가시켜, object detection의 정확도를 크게 향상시키는 방법들을 의미**

- **Object Detection을 시행할 때, inference시 cost를 아주 조금 증가시키지만 정확도를 크게 향상시킬 수 있는 추가 모듈 혹은 후처리(modules + post-processing) 방법.**

Ex) activation function, normalization, enhancement of receptive field, Feature integration etc…

Methodology

3.1. Selection of architecture

➢ **Detector 성능 향상을 위한 고려 사항**

- **더 큰 크기의 network 입력 해상도 :** 작은 물체를 잘 탐지하기 위하여
- **More layers :** 큰 해상도 이미지를 다루기 위한 넓은 receptive field를 위하여
- **More parameters :** 한 장의 이미지에서 다양한 크기의 objects들을 검출하기 위해,

  모델은 더 많은 capacity가 필요

Table 1: Parameters of neural networks for image classification.

Backbone model	Input network resolution	Receptive field size	Parameters	Average size of layer output (WxHxC)	BFLOPs (512x512 network resolution)	FPS (GPU RTX 2070)
CSPResNext50	512x512	425x425	20.6 M	**1058 K**	31 (15.5 FMA)	62
CSPDarknet53	512x512	725x725	**27.6 M**	950 K	52 (26.0 FMA)	**66**
EfficientNet-B3 (ours)	512x512	**1311x1311**	12.0 M	668 K	11 (5.5 FMA)	26

3.1. Selection of architecture

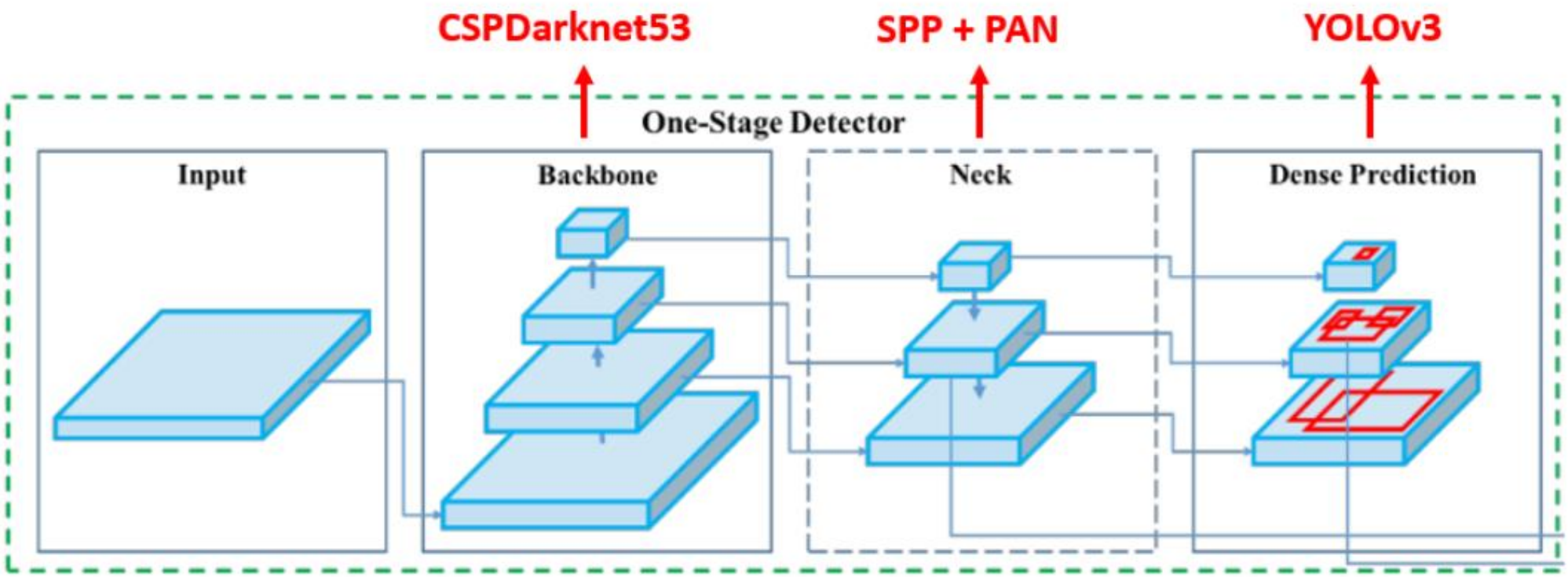
(1) 속도향상 :

- CSPDarknet53 사용(기존 Darknet)

(2) 정확도 향상 :

- Higher input resolution
- Higher Receptive field
- More Parameter : greater capacity of a model
- Detector 훈련 시 사용가능한 최신의 BoF 및 BoS 기법들 사용

3.1. Selection of architecture



YOLOv4 = CSPDarknet53 + SPP + PAN + YOLOv3

3.1. **Backbone –** CSP DarkNet 53

✓ **Mish activation사용**

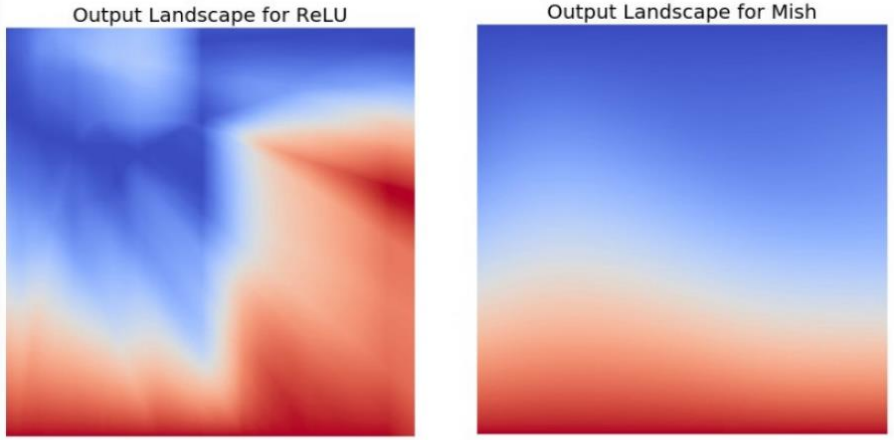
```
def relu(x):
    return max(0,x)

def swish(x) :
    return x * tf.nn.sigmoid(x)

def mish(x) :
    return x * tf.nn.tanh( tf.nn.softplus(x))
```

softplus 함수는 매끄럽게 만든 ReLU 함수 : log(exp(x) + 1)

Relu보다 Mish의 gradient 가 좀 더 스무스하게 나오는 것을 확인했다고 함



From the paper- a comparison of the output landscape from ReLU and Mish. The smooth gradients from Mish is a likely driver of it's outperformance.

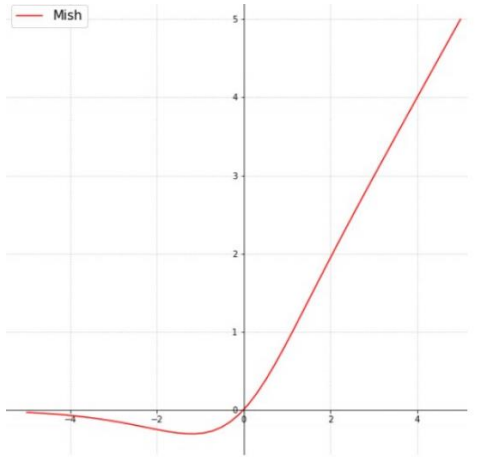


Figure 1. Mish Activation Function

- **Mish 사용**

→ 평균 정확도, 정점의 정확도 상승 확인

3.1. **Backbone –** CSP DarkNet 53

✓ **Mish activation사용**

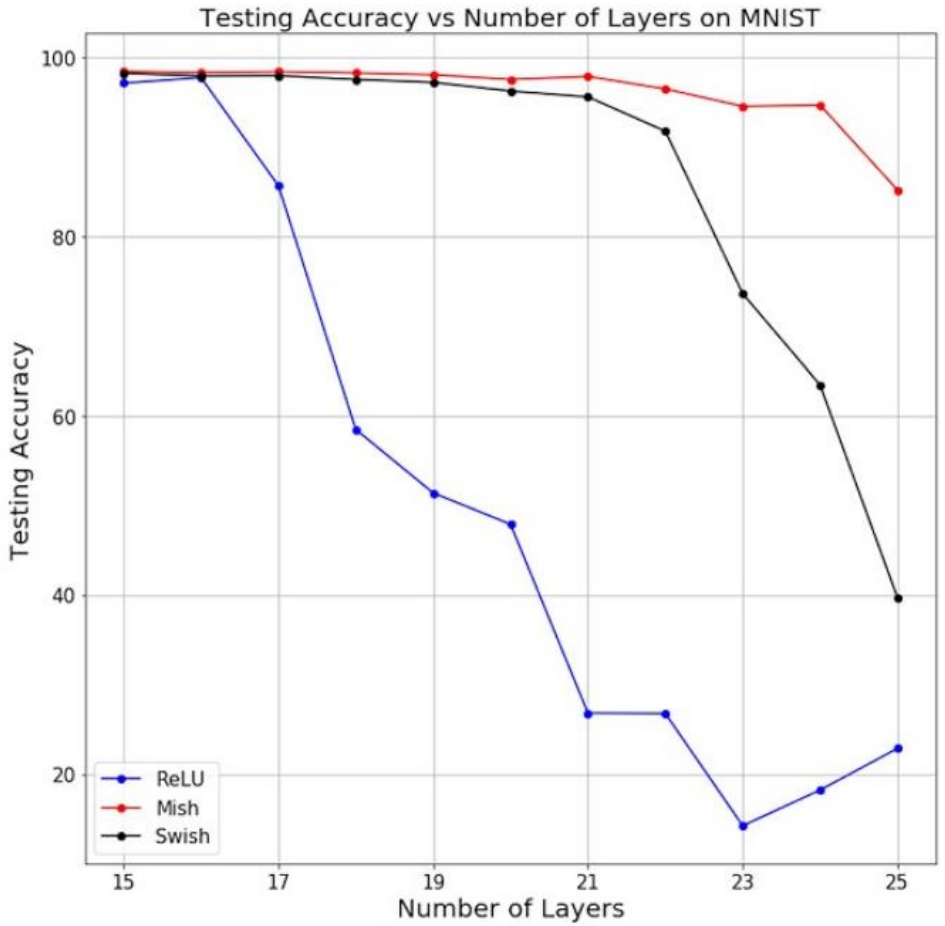
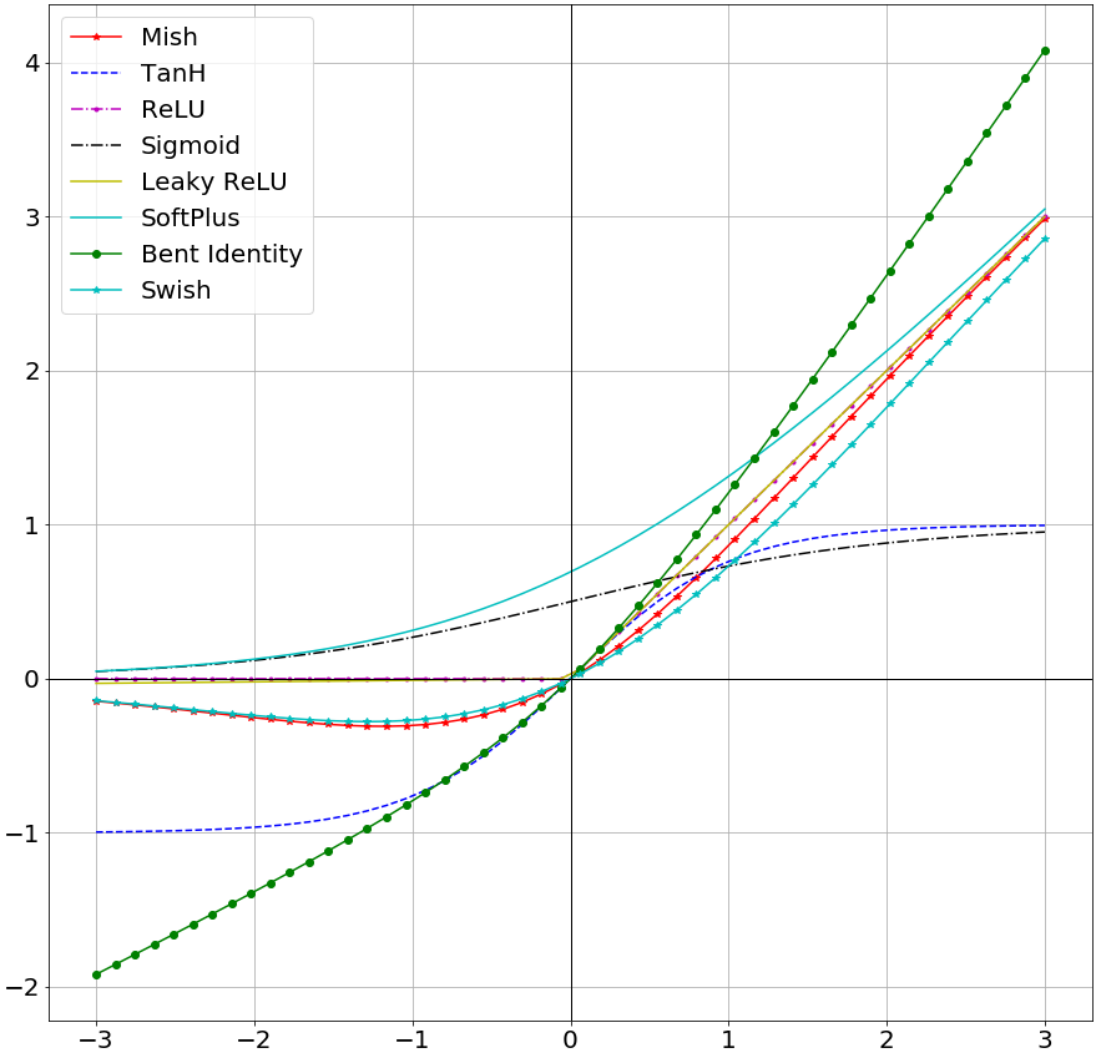


Figure 6. Testing Accuracy v/s Number of Layers on MNIST for Mish, Swish and ReLU.



3.1. **Backbone –** CSP DarkNet 53

✓ **CSP**(Cross-Stage-Partial-Connections) **Network**

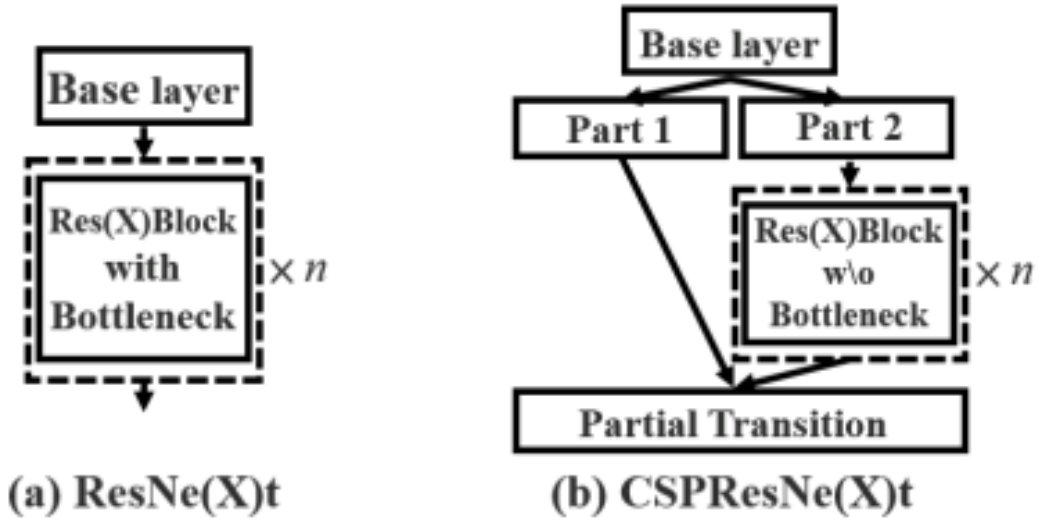


Figure 5: Applying CSPNet to ResNe(X)t.

- Feature Map의 **절반**만 Bottleneck통과
- **연산량** 감소
- **정확도** 상승

3.1. Selection of architecture

NECK은 다양한 크기의 feature map을 수집하기 위해 block을 추가하거나,
bottom-up path와 top-down path를 집계하는 방법을 사용.

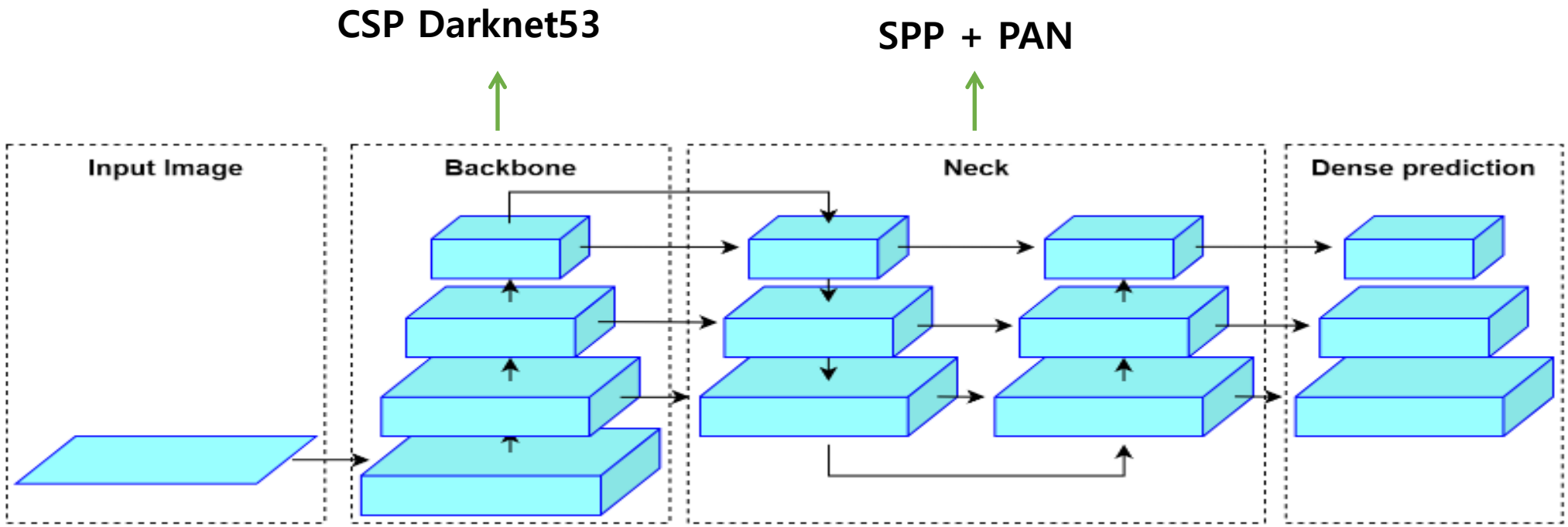
Feature map을 refinement(정제), reconfiguration(재구성)

YOLOv4 : SPP(Spatial Pyramid Pooling), PAN(Path Aggregation Network)

최근 개발된 detector들은 backbone과 head사이에 약간의 layer들을 삽입.
이 layers들은 보통 서로 다른 stage들로부터 온 feature maps들을 모으는데 사용.

3.1. Neck – SPP(Spatial Pyramid Pooling), PAN(Path Aggregation Network)

- **Additional blocks : SPP**
- **Path-aggregation blocks : PANet**



3.1. Additional Blocks : SPP(Spatial Pyramid Pooling)

- CSPDarknet53에 additional blocks으로 SPP block을 추가

✓ convolution layer를 통해 추출된 feature map을

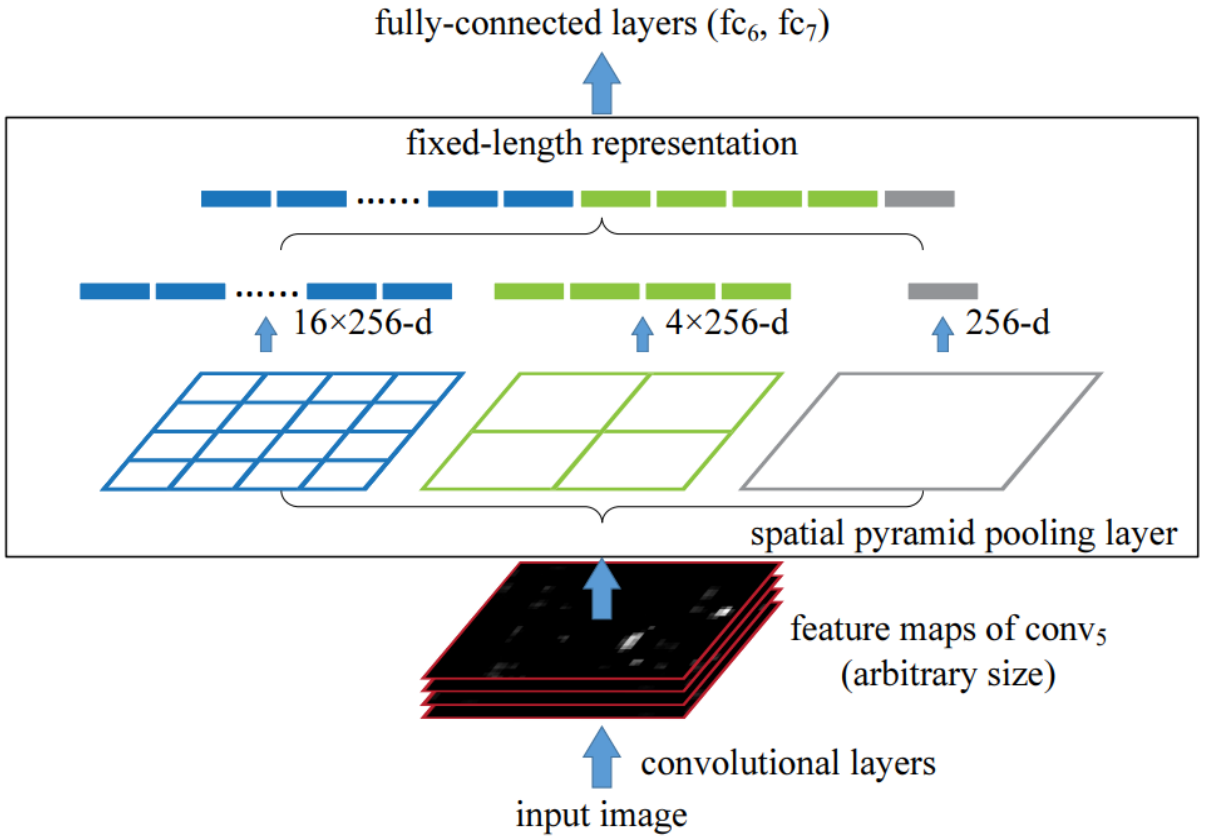
n개의 피라미드를 이용하여

고정된 길이의 feature representation을 생성

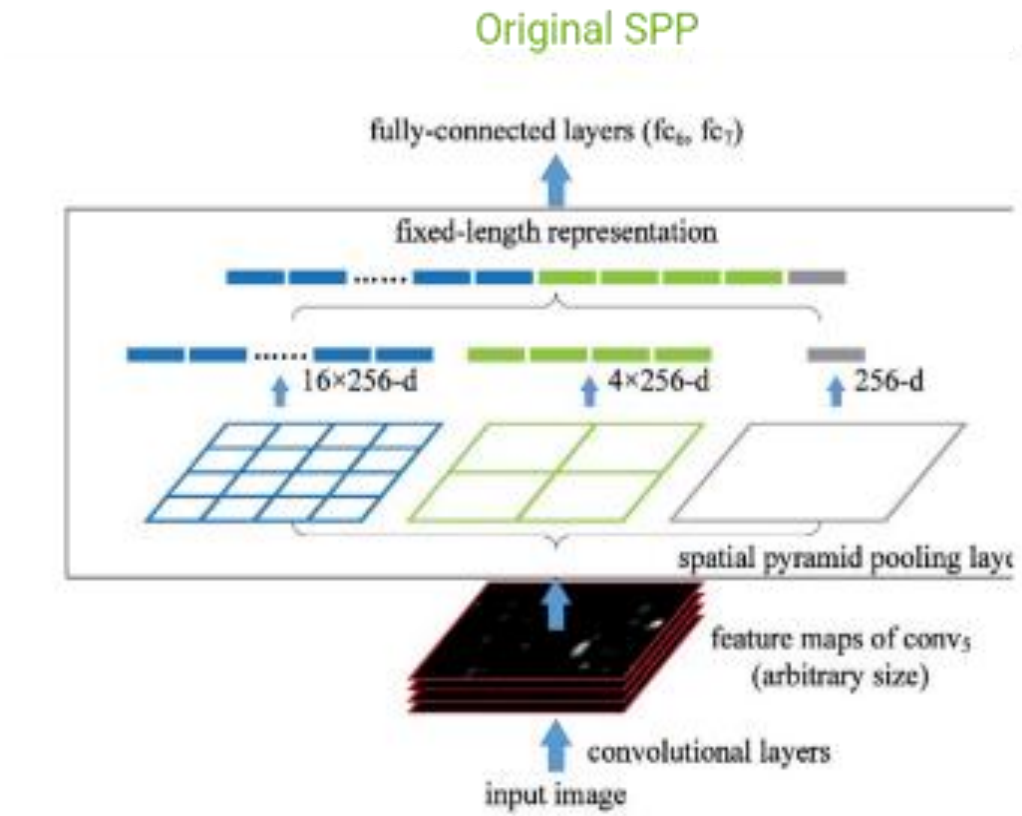
✓ FC layer가 포함된 CNN model들은

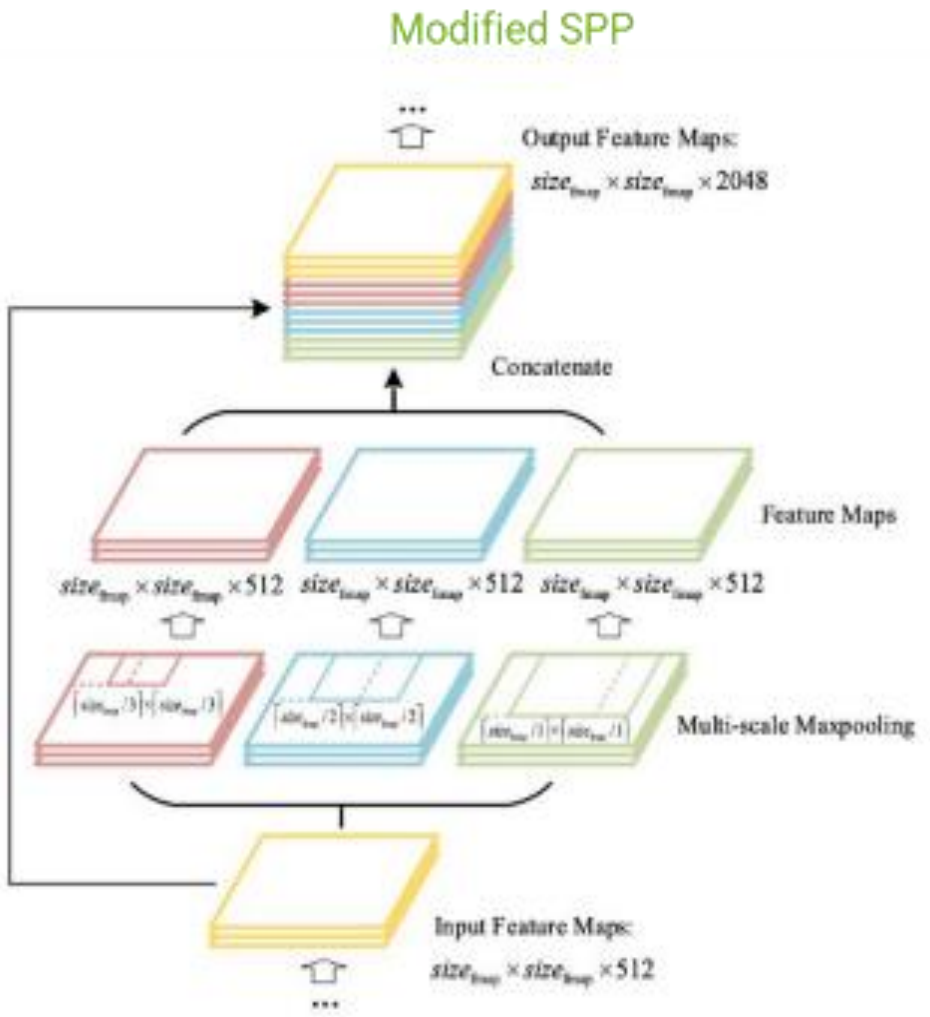
특정 차원의 입력 이미지만 허용하나,

SPP는 서로 다른 크기의 입력 이미지를 허용 가능



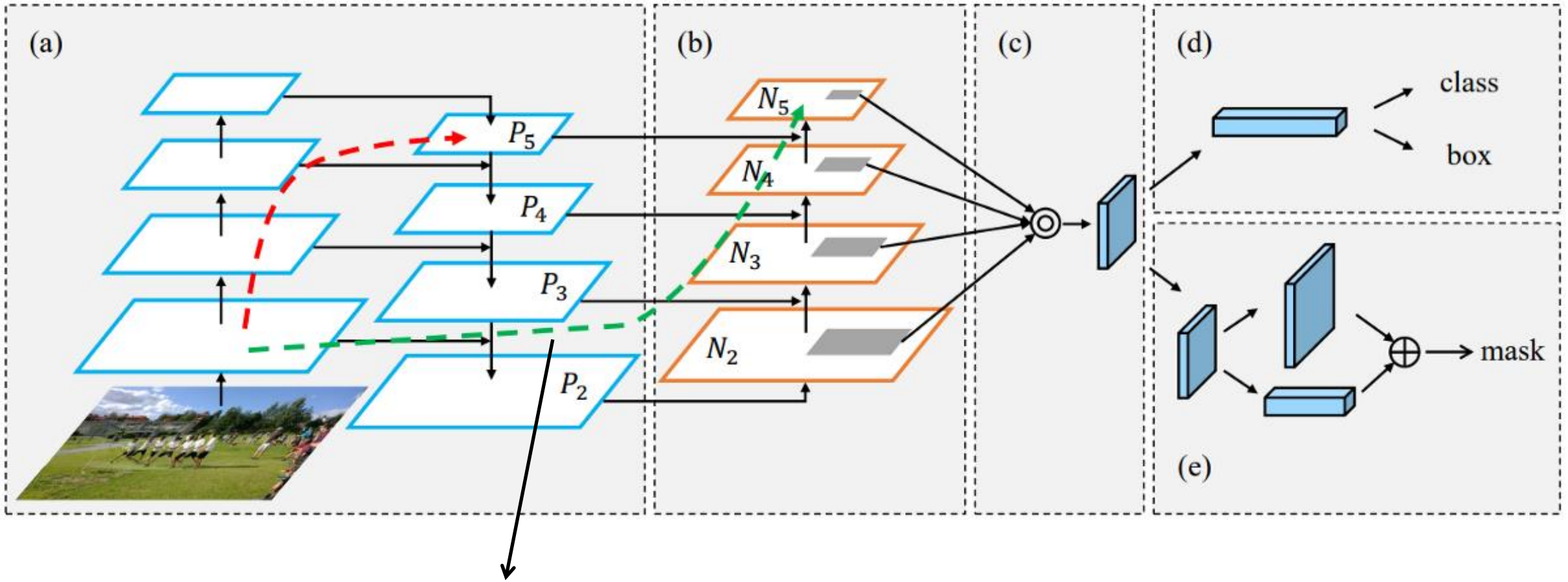
3.1. Additional Blocks : SPP(Spatial Pyramid Pooling)





3.1. Path-aggregation blocks : PAN(Path Aggregation Network)

FPN의 한 종류



상위 계층에서 미세한 지역화된 정보를 사용가능

3.4. YOLOv4 Architecture

CSPResNet

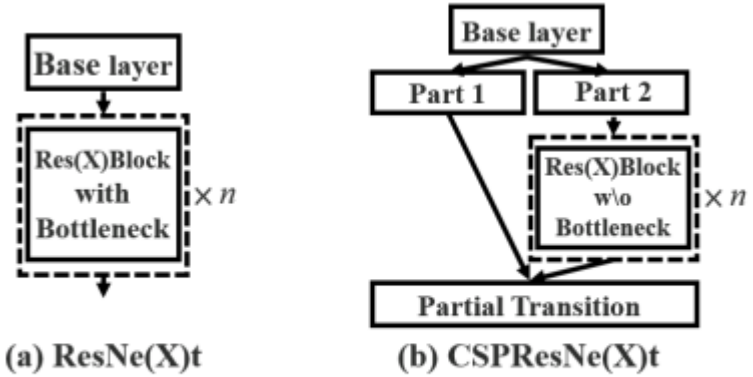
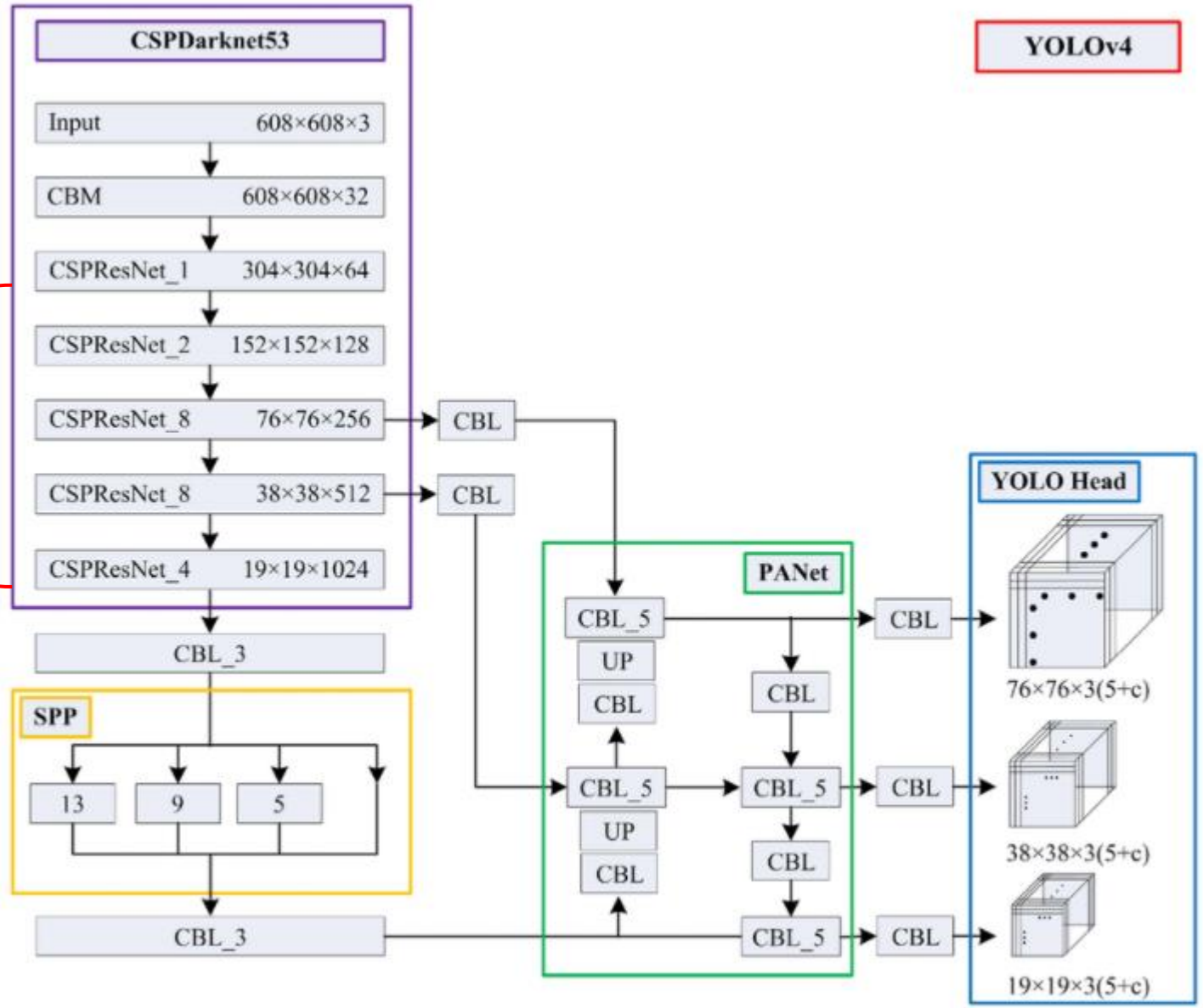


Figure 5: Applying CSPNet to ResNe(X)t.



Block diagram of YOLOv4 object detection. The small modules included are CBM: Convolution + Batch Normalization + Mish; CBL: Convolution + Batch Normalization + Leaky ReLU; and UP: Upsampling

➢ 최종적으로 선택된 기법들

1. Backbone: CSPDarknet53

2. Neck:

- additional blocks: SPP

- path-aggregation blocks: PAN

3. **Head**: YOLOv3(anchor-based)

3.3. Additional improvement

- Data augmentation : Mosaic, Self-Adversarial Training(SAT)
- Modified Existing Methods :

  Modified SAM, Modified PAN, Cross mini-Batch Normalization(CmBN)

3.3. Data augmentation : Mosaic

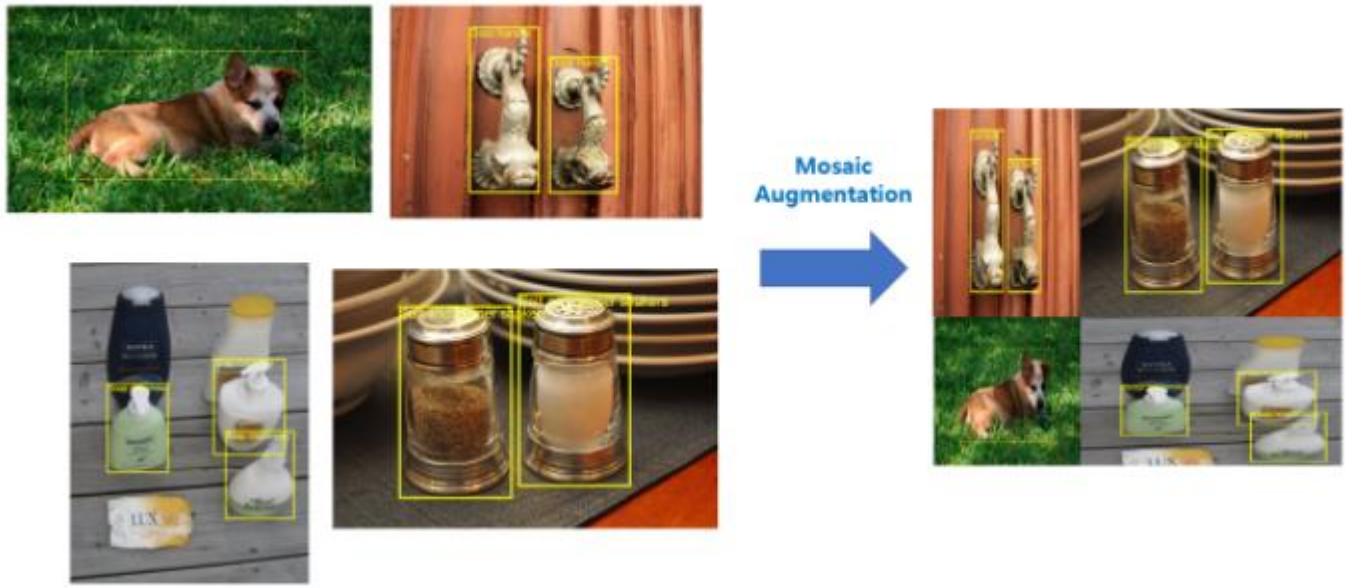
Mosaic

Figure 3: Mosaic represents a new method of data augmentation.

새로운 data augmentation 기법

- ✓ 4개의 학습 이미지를 혼합
- ✓ 이미지를 섞음으로써 하나의 객체로 인식
- ✓ Mini Batch에 대한 필요성을 줄임.

3.3. Data augmentation : Mosaic



4개의 image

+

Bounding Box

→ 512 X 512 image로 합침 → 하나의 input으로 Batch size X 4의 효과를 얻을 수 있음

→ 작은 물체가 늘어남 → 작은 물체에 대한 성능 UP

3.3. Modified Existing Methods : Modified SAM, Modified PAN

Modified SAM(Spatial Attention Module), Modified PAN(Path Aggregation Network)

- SAM을 spatial-wise attention에서 point-wise attention으로 변경
- PAN의 shortcut connection을 concatenation으로 교체

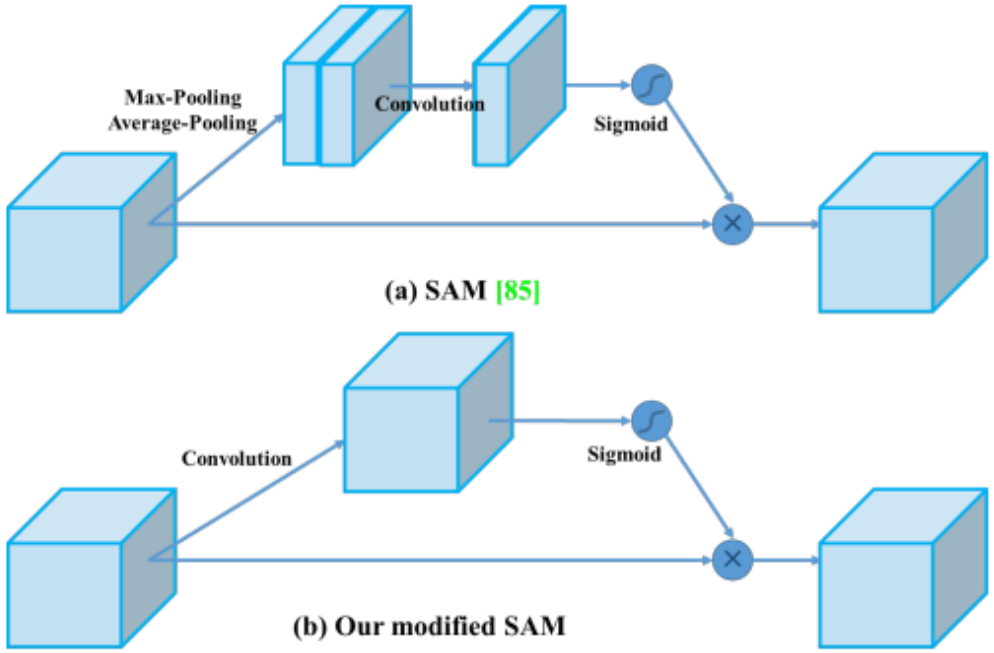


Figure 5: Modified SAM.

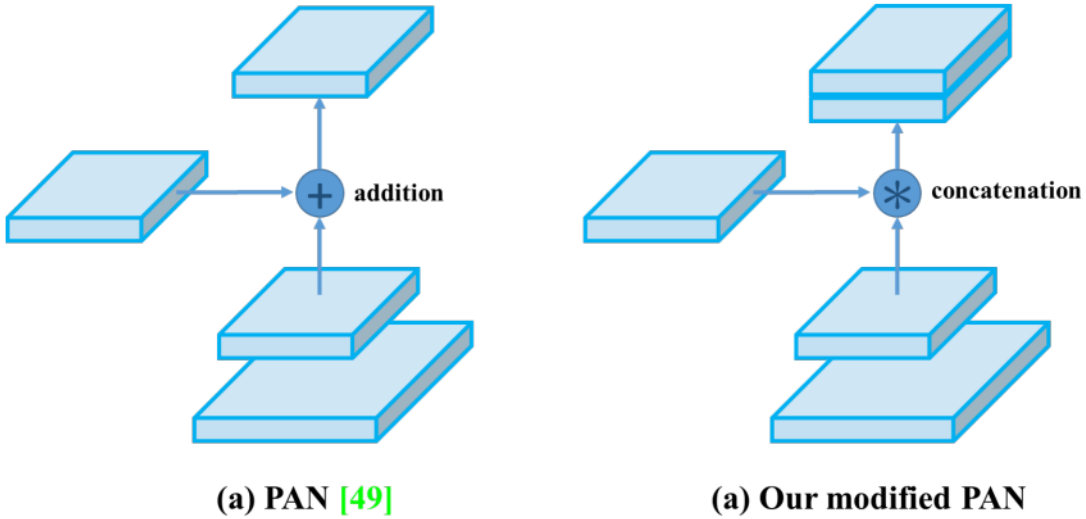


Figure 6: Modified PAN.

3.3. Modified Existing Methods : CmBN(Cross Mini-Batch Normalization)

- **BN** : batch size가 작을 경우, examples들에 대한 정확한 statistic estimation이 어려우므로 효율성이 저하
- **CBN** : estimation quality의 향상을 위해, 이전 iteration들의 통계를 함께 활용
- **CmBN :** CBN의 수정된 버전으로서, 단일 batch 내에서 mini-batches 사이에 대한 통계를 수집

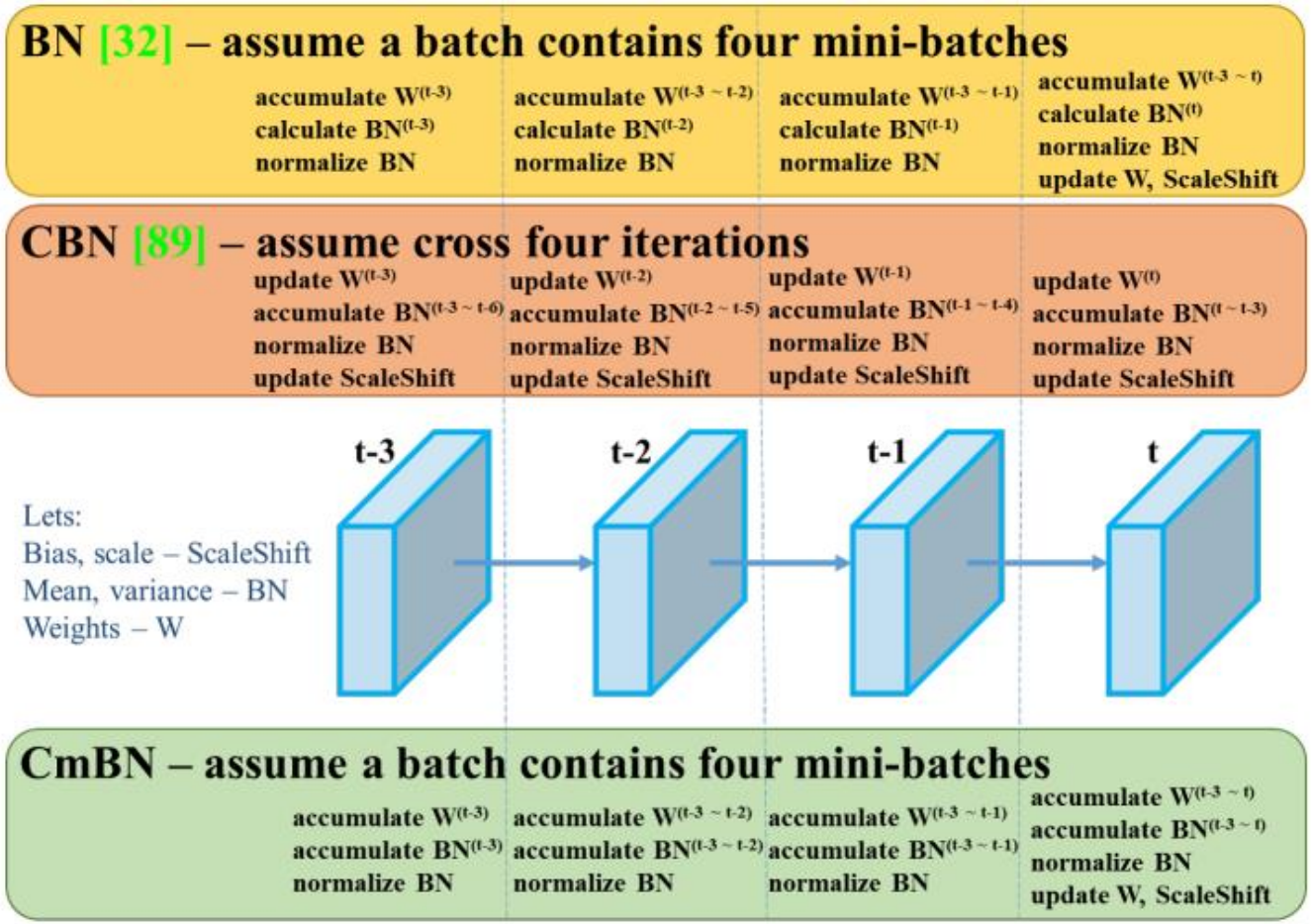


Figure 4: Cross mini-Batch Normalization.

Experiments

4. Experiments

classifier training 성능 향상시키는 feature들

- CutMix and Mosaic for data augmentation
- Class label smoothing
- Mish activation

Table 2: Influence of BoF and Mish on the CSPResNeXt-50 classifier accuracy.

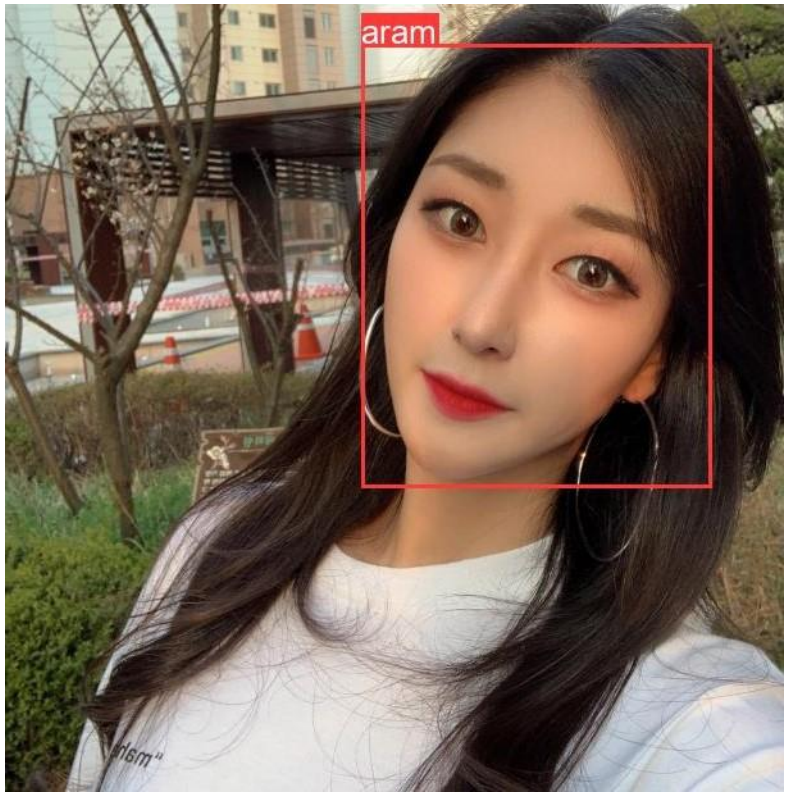
MixUp	CutMix	Mosaic	Bluring	Label Smoothing	Swish	Mish	Top-1	Top-5
							77.9%	94.0%
✓							77.2%	**94.0%**
	✓						**78.0%**	**94.3%**
		✓					**78.1%**	**94.5%**
			✓				77.5%	93.8%
				✓			**78.1%**	**94.4%**
					✓		64.5%	86.0%
						✓	**78.9%**	**94.5%**
	✓	✓		✓			**78.5%**	**94.8%**
	✓	✓		✓		✓	**79.8%**	**95.2%**

Table 3: Influence of BoF and Mish on the CSPDarknet-53 classifier accuracy.

MixUp	CutMix	Mosaic	Bluring	Label Smoothing	Swish	Mish	Top-1	Top-5
							77.2%	93.6%
	✓	✓		✓			**77.8%**	**94.4%**
	✓	✓		✓		✓	**78.7%**	**94.8%**

기술 구현


aram


aram


aram

Source Code

*Detector는 **백본**(Backbone)과 **헤드**(Head)로 구성*

Backbone : 이미지로부터 **Feature map을 추출**하는 부분(ex. CSP-Darknet)
Head : Feature map의 **location**작업을 수행하는 부분

```yaml
# YOLOv5 v6.0 backbone
backbone:
  # [from, number, module, args]
  [[-1, 1, Conv, [64, 6, 2, 2]],  # 0-P1/2
   [-1, 1, Conv, [128, 3, 2]],  # 1-P2/4
   [-1, 3, C3, [128]],
   [-1, 1, Conv, [256, 3, 2]],  # 3-P3/8
   [-1, 6, C3, [256]],
   [-1, 1, Conv, [512, 3, 2]],  # 5-P4/16
   [-1, 9, C3, [512]],
   [-1, 1, Conv, [1024, 3, 2]],  # 7-P5/32
   [-1, 3, C3, [1024]],
   [-1, 1, SPPF, [1024, 5]],  # 9
  ]
```

```yaml
# YOLOv5 v6.0 head
head:
  [[-1, 1, Conv, [512, 1, 1]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [[-1, 6], 1, Concat, [1]],  # cat backbone P4
   [-1, 3, C3, [512, False]],  # 13

   [-1, 1, Conv, [256, 1, 1]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [[-1, 4], 1, Concat, [1]],  # cat backbone P3
   [-1, 3, C3, [256, False]],  # 17 (P3/8-small)

   [-1, 1, Conv, [256, 3, 2]],
   [[-1, 14], 1, Concat, [1]],  # cat head P4
   [-1, 3, C3, [512, False]],  # 20 (P4/16-medium)

   [-1, 1, Conv, [512, 3, 2]],
   [[-1, 10], 1, Concat, [1]],  # cat head P5
   [-1, 3, C3, [1024, False]],  # 23 (P5/32-large)

   [[17, 20, 23], 1, Detect, [nc, anchors]], # Detect(P3, P4, P5)
  ]
```

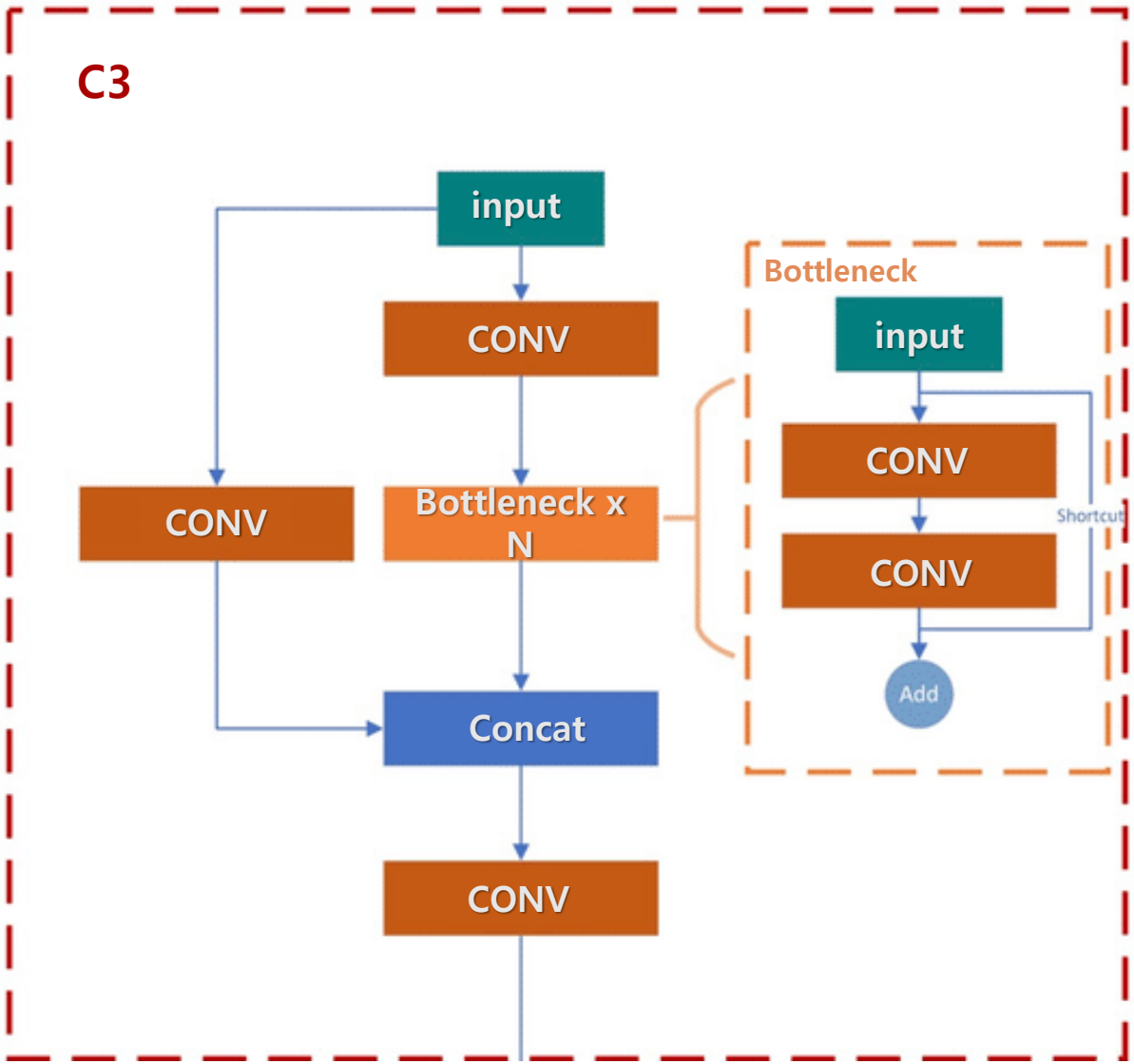
- BottleneckCSP

```python
class BottleneckCSP(nn.Module):
    def __init__(self, c1, c2, n=1, shortcut=True, g=1, e=0.5):  # ch_in, ch_out, number, shortcut, groups, expansion
        super().__init__()
        c_ = int(c2 * e)  # hidden channels
        self.cv1 = Conv(c1, c_, 1, 1)
        self.cv2 = nn.Conv2d(c1, c_, 1, 1, bias=False)
        self.cv3 = nn.Conv2d(c_, c_, 1, 1, bias=False)
        self.cv4 = Conv(2 * c_, c2, 1, 1)
        self.bn = nn.BatchNorm2d(2 * c_)  # applied to cat(cv2, cv3)
        self.act = nn.SiLU()
        self.m = nn.Sequential(*(Bottleneck(c_, c_, shortcut, g, e=1.0) for _ in range(n)))

    def forward(self, x):
        y1 = self.cv3(self.m(self.cv1(x)))
        y2 = self.cv2(x)
        return self.cv4(self.act(self.bn(torch.cat((y1, y2), 1))))
```

Feature map을 정제(Refinement), 재구성(Reconfiguration) 하는 부분

C3 module



- BottleneckCSP와 유사한 구조
- 절반만 Bottleneck 통과
- 간단한 구조 사용

- **Neck은 Backbone과 Head를 연결하는 부분**

```python
class C3(nn.Module):
    # CSP Bottleneck with 3 convolutions
    def __init__(self, c1, c2, n=1, shortcut=True, g=1, e=0.5):  # ch_in, ch_out, number, shortcut, groups, expansion
        super().__init__()
        c_ = int(c2 * e)  # hidden channels
        self.cv1 = Conv(c1, c_, 1, 1)
        self.cv2 = Conv(c1, c_, 1, 1)
        self.cv3 = Conv(2 * c_, c2, 1)  # act=FReLU(c2)
        self.m = nn.Sequential(*(Bottleneck(c_, c_, shortcut, g, e=1.0) for _ in range(n)))
        # self.m = nn.Sequential(*[CrossConv(c_, c_, 3, 1, g, 1.0, shortcut) for _ in range(n)])

    def forward(self, x):
        return self.cv3(torch.cat((self.m(self.cv1(x)), self.cv2(x)), dim=1))
```

```python
class Bottleneck(nn.Module):
    # Standard bottleneck
    def __init__(self, c1, c2, shortcut=True, g=1, e=0.5):  # ch_in, ch_out, shortcut, groups, expansion
        super().__init__()
        c_ = int(c2 * e)  # hidden channels
        self.cv1 = Conv(c1, c_, 1, 1)
        self.cv2 = Conv(c_, c2, 3, 1, g=g)
        self.add = shortcut and c1 == c2

    def forward(self, x):
        return x + self.cv2(self.cv1(x)) if self.add else self.cv2(self.cv1(x))
```

Zero_grad()

```
import torch as T
import torch,optimizer as optim

optimizer = optim.Adam(lr=lr, params=self.parameters())

...

optimizer.zero_grad()
loss.backward()
optimizer.step()
```

미니배치 + 루프

epoch

전체 데이터 셋을 반복하는 횟수

전체 데이터셋으로 forwardpropagation(순전파)와 backwardpropagation(역전파)가 완료되면 1번의 epoch가 진행되었다고 보면된다.

반복적인 학습을 통해 높은 정확도의 모델을 만들 수 있다.

설정한 epoch 값이 너무 작다면 underfitting, 너무 크다면 overfitting이 발생할 확률이 높아짐

batch size

forward와 backward에서 **한번에 학습할 데이터의 수**

메모리의 한계와 속도 저하 때문에 한 번의 epoch에서 모든 데이터를 한꺼번에 집어넣을 수 없음

iteration

한 epoch 에서 batch를 학습하는 횟수

전체 데이터에 대한 오차 총합으로 propagation을 수행하면 weight가 한 번에 크게 변할 수 있기 때문에 gradient decent처럼 조금씩 이동해 최적화 할 수 있도록 한다.

> 데이터가 100개, batch size가 10 이면, 1 Epoch = 10 iterations
>
> data size = batch size * iterations

Q & A