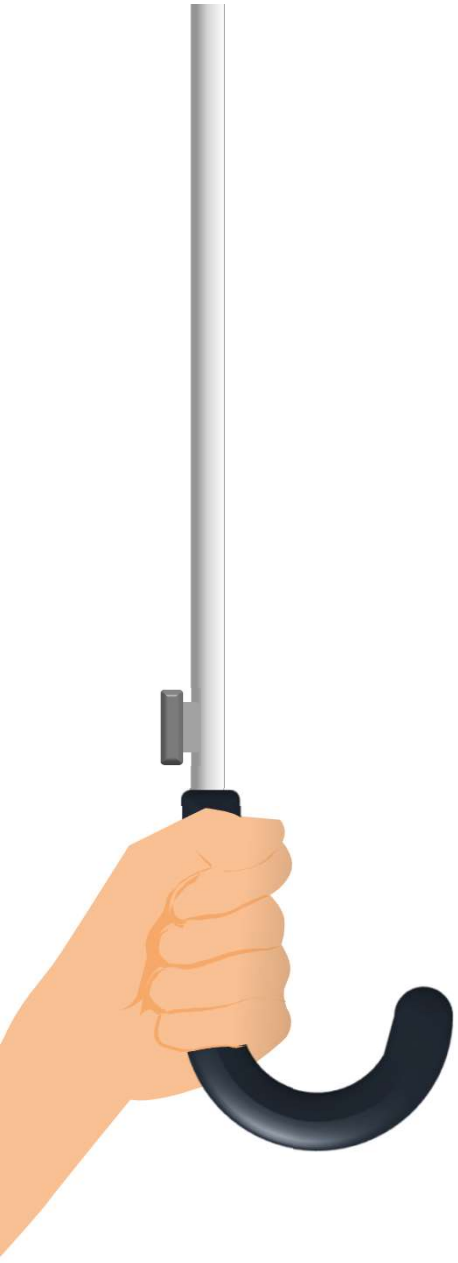
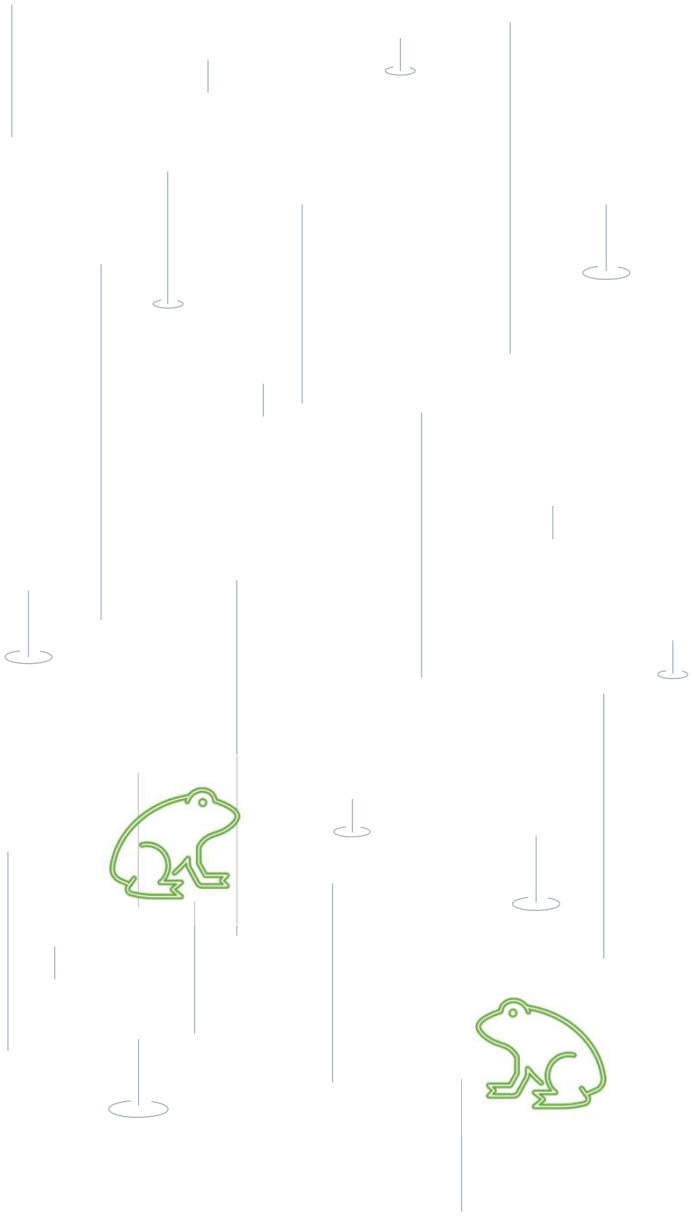
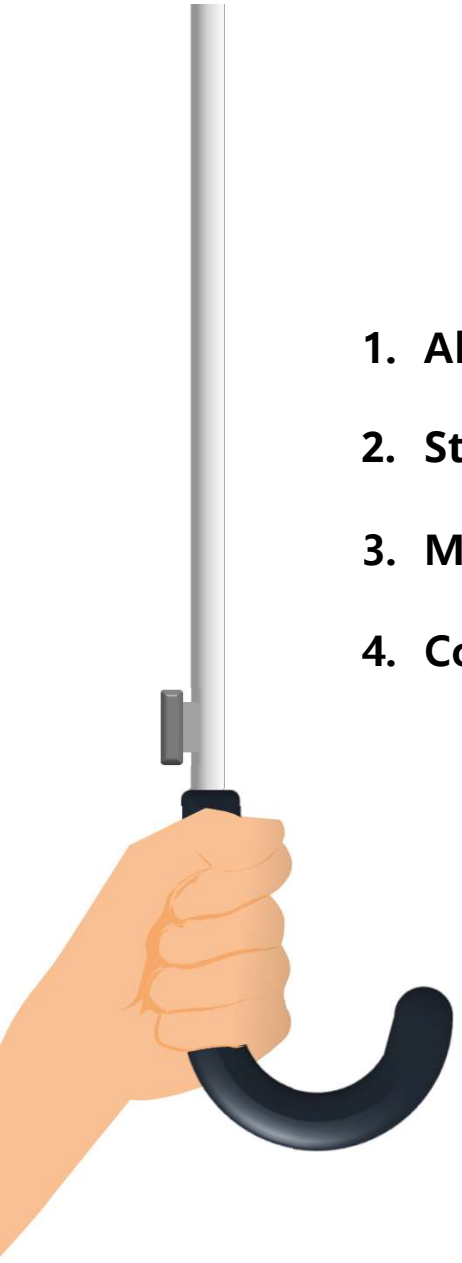


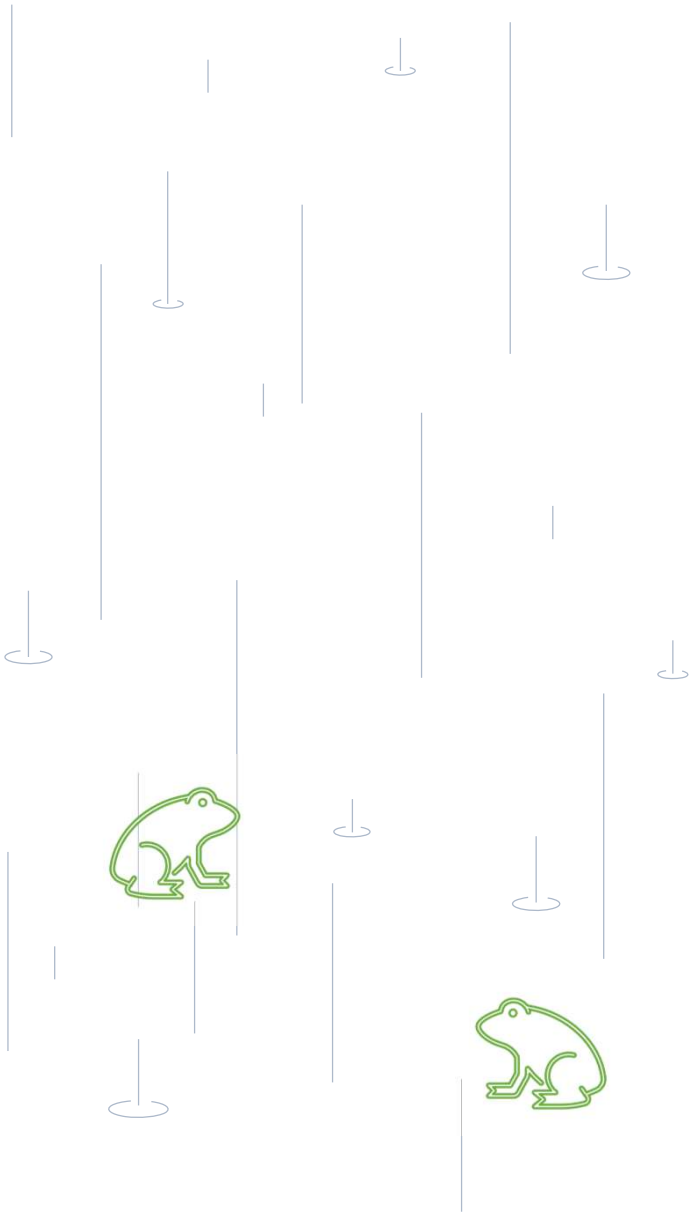
EDSR, MDSR

HB





1. About EDSR, MDSR
2. Structure of the EDSR model
3. MDSR, EDSR+, MDSR+
4. Code review



1. About EDSR,MDSR

HB

- EDSR : Enhanced Deep Super-Resolution

- 기존의 super-resolution은 deep convolutional neural network의 발전으로 인해 성능 상승

- 하지만 구조적인 optimality에서는 한계가 있다. (구조가 optimal 하지 않다.)

  1. reconstruction performance이 사소한 구조적인 변화에도 민감하게 반응한다. 같은 모델일지라도 초기화, 훈련 techniques에 따라 성능이 상이하다. -> **훈련의 안정성이 낮다.**

  2. 기존 SR 알고리즘은 서로 다른 scale 간의 상호 관계를 고려하고 활용하지 않는다. 때문에 다양한 scale의 sr을 처리하기 위해 독립적으로 훈련되어야 하는 **많은 네트워크를 필요로 한다.**

1. About EDSR,MDSR

HB

- VDSR

- 단일 네트워크에서 여러 scale의 SR을 처리할 수 있다.

- VDSR 모델을 여러 scale으로 훈련하면 성능이 크게 향상된다.

- 하지만 VDSR은 input으로 원본 이미지를 bicubic interpolate한 이미지가 필요
-> **계산 시간과 메모리 사용이 훨씬 더 많아진다.**

1. About EDSR,MDSR

HB

- SRResNet

- VDSR의 문제점인 시간과 메모리 문제를 좋은 성능으로 해결

- 단순히 ResNet(from He et al.) 구조를 수정없이 차용

- 하지만 original Resnet은 컴퓨터 비전의 higher-level 문제를 해결하기 위한 것

- 따라서 **SR 같은 low-level vision problem에는 optimal이 아니라 suboptimal일 수도 있다.**

1. About EDSR,MDSR

HB

- **EDSR은 SRResNet을 개선하여 문제점을 해결한 구조**

- 일반적인 Resnet 구조에서 불필요한 module 제거 -> 성능향상

- training 중 다른 scale의 훈련에서의 scale-independent 정보를 활용
    1. high-scale을 학습할 때 low-scale의 pretrained 모델을 활용(EDSR)
    2. 학습을 할 때 여러 scale간 파라미터를 공유하는 multi-scale model(MDSR)

  -> 네트워크 수 감소

- SR 모델에서 통용적으로 사용하던 MSE loss나 L2 loss 대신 L1 loss를 사용하여 약간의 성능 향상

- MDSR(Multi-scale Deep Super-Resolution) : 여러 단일 scale SR 모델과 비교하여 훨씬 적은 parameter를 사용하지만 EDSR과 유사한 성능(약간 낮음)

2. Structure of EDSR model

HB

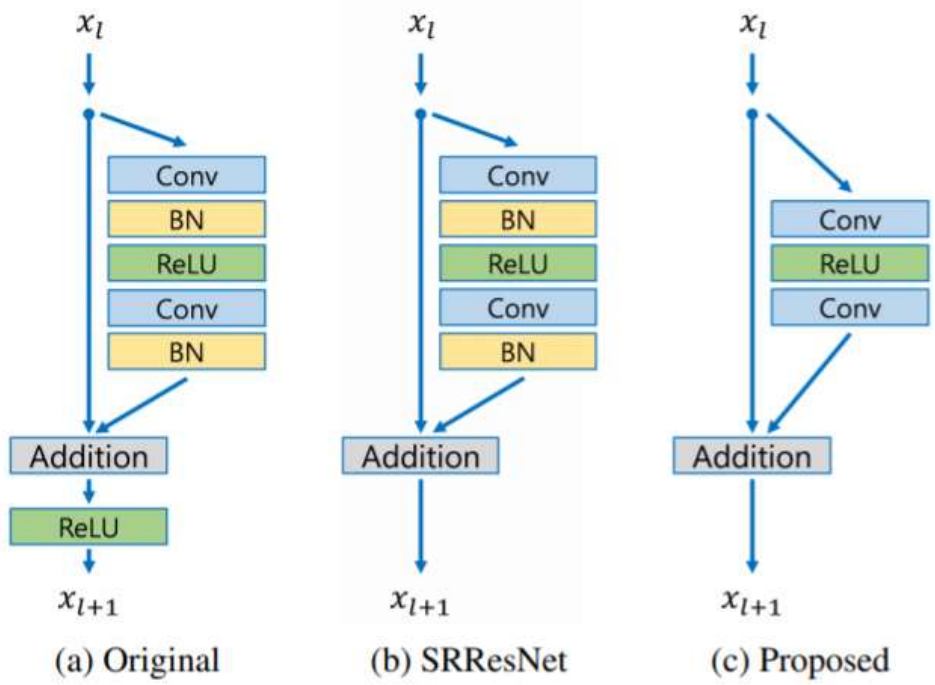


Figure 2: Comparison of residual blocks in original ResNet, SRResNet, and ours.

- Residual block의 Batch-Normalization(BN)은 feature들을 normalize
  -> range flexibility를 제거.

Normalize : 일정 범위안에 있는 값들을 한 값으로 일반화(get rid of range flexibility)
-> 분류, 검출 문제에는 좋은 성능
하지만 SR resolution에는 오히려 성능을 떨어뜨릴 수 있다.

2. Structure of EDSR model

HB

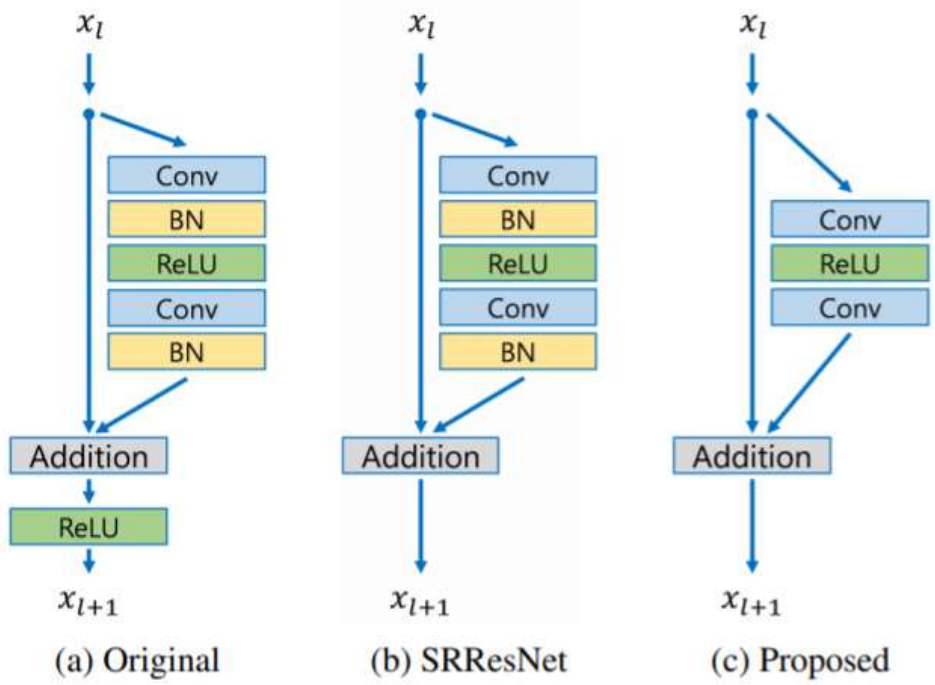


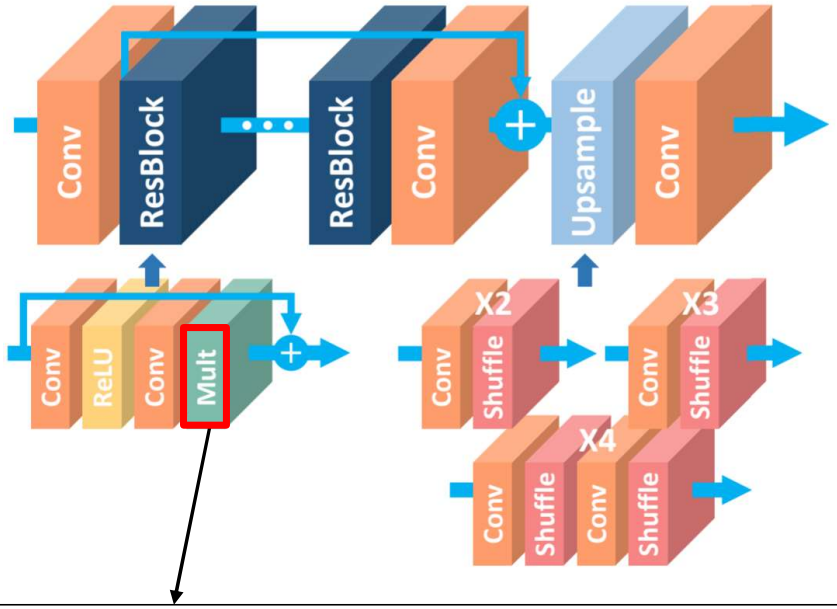
Figure 2: Comparison of residual blocks in original ResNet, SRResNet, and ours.

- Batch-Normalization(BN)은 feature들을 normalize
  -> range flexibility를 제거.

- BN을 제거해 성능 향상 + 메모리 절감(SRResNet과 비교해서 40%나 절감)
  ->계산적인 자원 한계 때문에 불가하던 크기의 모델을 설계 가능

2. Structure of EDSR model

HB



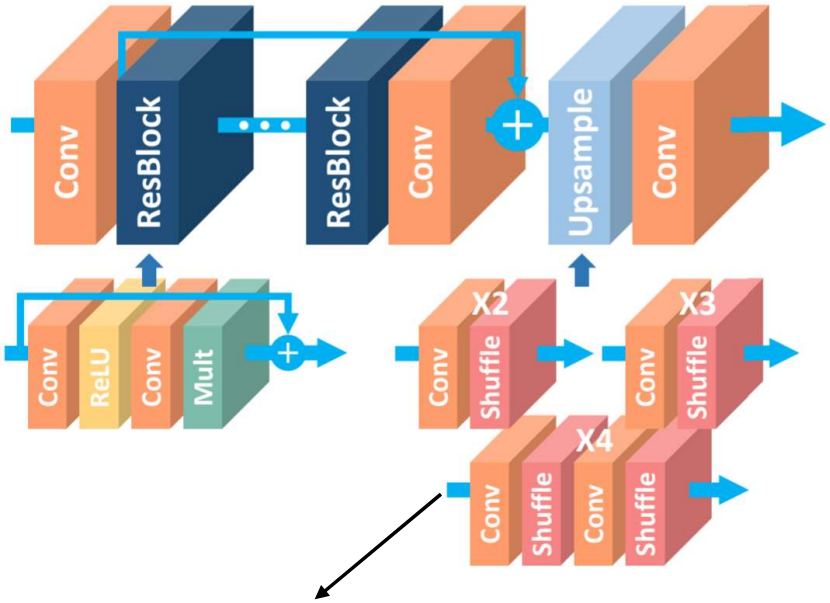
- 성능을 향상시키는 가장 단순한 방법은 모델의 크기를 증가시키는 것, 즉 parameter를 증가 시키는 것

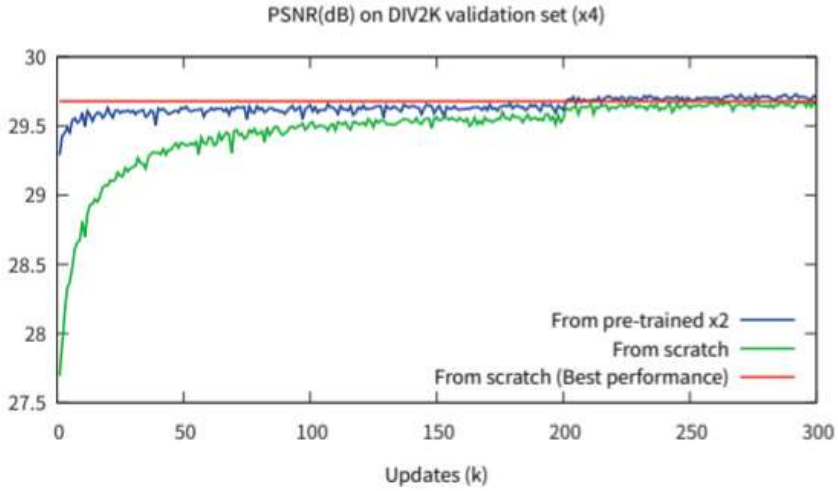
- CNN 구조는 O(BF) 메모리와 $O(BF^2)$의 파라미터 (B: depth F: width(the number of feature channel)
-> B보다 F를 증가시키는 것이 메모리를 덜 사용하면서 성능향상 가능)

- 하지만 일정 feature map의 개수가 일정 도달하면 학습이 불안정해짐(값들이 계속 더해지면서 variance가 증가) -> 따라서 residual scale factor(0.1)를 곱해줌

2. Structure of EDSR model

HB



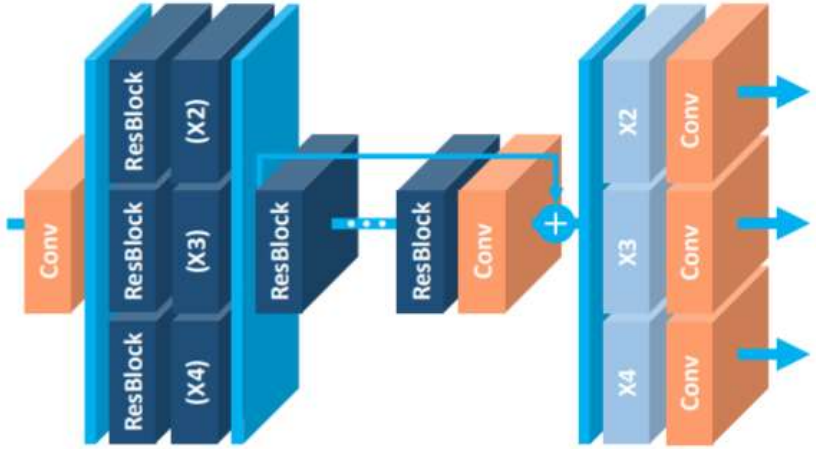


- upsampling factor x3, x4를 train할 때 pretrained x2 network의 파라미터로 initialize
-> 훈련 가속화, 성능 향상

- 오른쪽 그래프는 처음부터 train 했을 때와 x2 scale 모델의 pretrained 를 사용한 train의 성능을 비교한 것이다. Pretrained model을 사용했을 때가 성능도 더 좋고 비교적 빨리 최고점에 근접한다.

3. MDSR

HB



MDSR

- VDSR이 한 것처럼 scale 간 상관 관계를 활용 -> 파라미터 공유

- 모든 scale의 training을 한번에 진행.

- 시작 부분에 scale별 전처리 모듈(5x5 kernels로 이루어진 resblock 2개)

- scale-specific reconstruction을 위한 끝 부분에 scale별 upsampling module

-> EDSR보다 파라미터가 적다 (EDSR의 약 3분의 2). 성능 비슷(EDSR보다 약간 낮다.)

3. EDSR+, MDSR+

HB

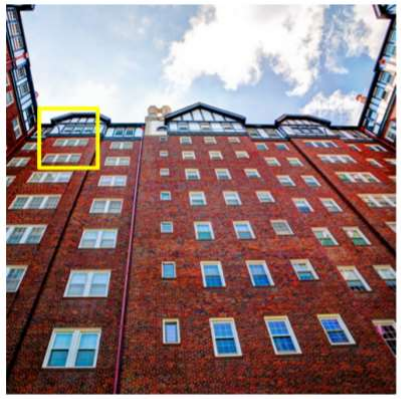
Scale	SRResNet (L2 loss)	SRResNet (L1 loss)	Our baseline (Single-scale)	Our baseline (Multi-scale)	EDSR (Ours)	MDSR (Ours)	**EDSR+ (Ours)**	**MDSR+ (Ours)**
×2	34.40 / 0.9662	34.44 / 0.9665	34.55 / 0.9671	34.60 / 0.9673	35.03 / 0.9695	34.96 / 0.9692	35.12 / 0.9699	35.05 / 0.9696
×3	30.82 / 0.9288	30.85 / 0.9292	30.90 / 0.9298	30.91 / 0.9298	31.26 / 0.9340	31.25 / 0.9338	31.39 / 0.9351	31.36 / 0.9346
×4	28.92 / 0.8960	28.92 / 0.8961	28.94 / 0.8963	28.95 / 0.8962	29.25 / 0.9017	29.26 / 0.9016	29.38 / 0.9032	29.36 / 0.9029

$$I_n^{SR} = \frac{1}{8}\sum_{i=1}^{8} \tilde{I}_{n,i}^{SR}$$

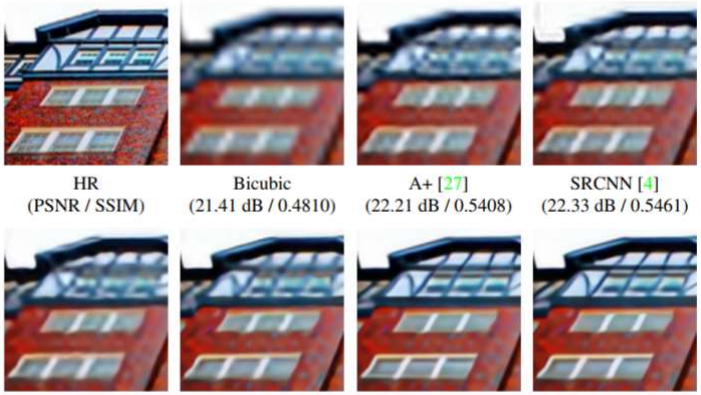
- 테스트하는 동안 입력 이미지를 7개의 이미지를 data augmentation을 통해 생성하여 총 8개의 이미지를 생성하여 8개의 SR이미지를 생성
- 8개의 SR이미지를 원래의 이미지로 변환하여 이의 평균의 값을 통해 결과를 도출
- 이는 추가적인 별도의 모델을 필요하지 않고 추가적인 성능(약간 증가)을 얻을 수 있는 방법
- 위 표는 각 모델의 성능을 나타낸 것 (PSNR/SSIM)

Result

HB



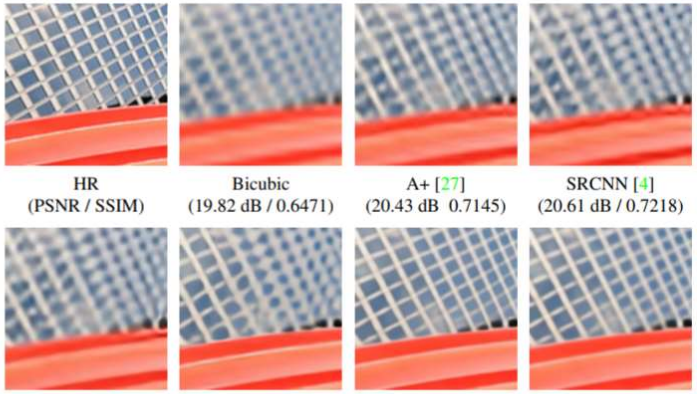
img034 from Urban100 [10]



HR (PSNR / SSIM) | Bicubic (21.41 dB / 0.4810) | A+ [27] (22.21 dB / 0.5408) | SRCNN [4] (22.33 dB / 0.5461)

VDSR [11] (22.62 dB / 0.5657) | SRResNet [14] (23.14 dB / 0.5891) | **EDSR+ (Ours)** (23.48 dB / 0.6048) | **MDSR+ (Ours)** (23.46 dB / 0.6039)

img062 from Urban100 [10]



HR (PSNR / SSIM) | Bicubic (19.82 dB / 0.6471) | A+ [27] (20.43 dB 0.7145) | SRCNN [4] (20.61 dB / 0.7218)

VDSR [11] (20.75 dB / 0.7504) | SRResNet [14] (21.70 dB / 0.8054) | **EDSR+ (Ours)** (22.70 dB / 0.8537) | **MDSR+ (Ours)** (22.66 dB / 0.8508)

Result

HB

Origin (318x180)

EDSR x4

Resize x4

Result (1272x720)

PSNR: 33.17

PSNR: 32.01

Result

HB

Origin (318x180)

EDSR x4

Resize x4

Result (1272x720)

PSNR: 33.60

PSNR: 32.59

Result

HB

Origin (318x180)

EDSR x4

Resize x4

Result (1272x720)

PSNR: 36.60

PSNR: 35.31

Result

HB

Origin (318x180)

EDSR x4

Resize x4

Result (1272x720)

PSNR: 34.07

PSNR: 32.88

Result

HB

Origin (318x180)

EDSR x4

Resize x4

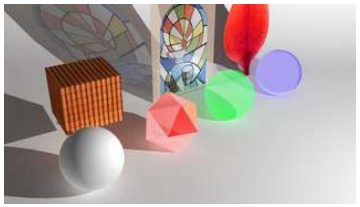
Result (1272x720)

PSNR: 33.82

PSNR: 32.70

Result

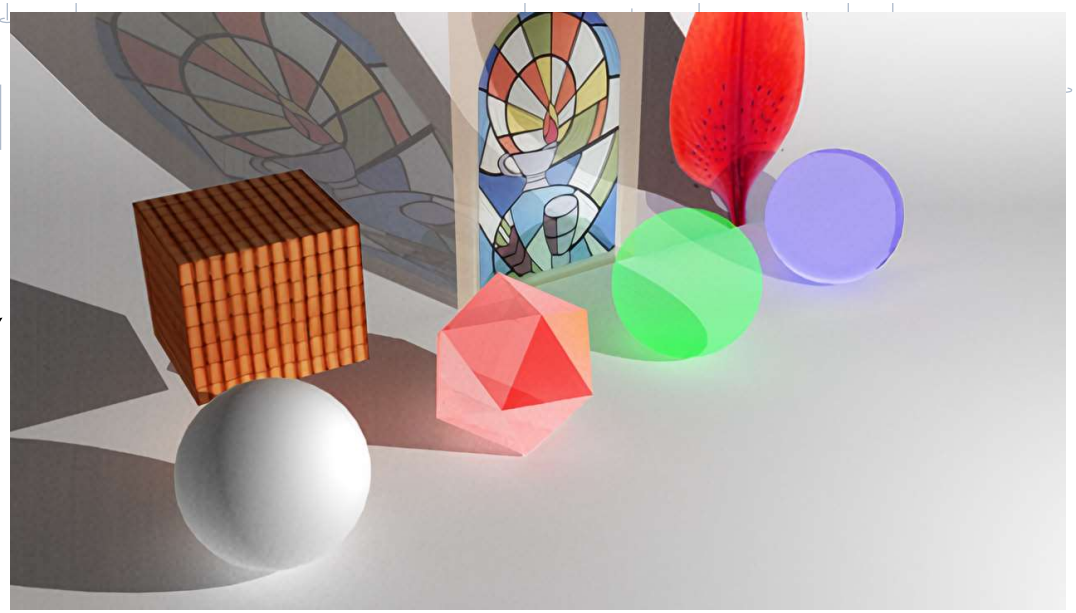
HB



Origin (318x180)

EDSR x4

Resize x4



Result (1272x720)

PSNR: 35.57

PSNR: 34.59

Code review

Main

```
import torch

import utility
import data
import model
import loss
from option import args
from trainer import Trainer

torch.manual_seed(args.seed)
checkpoint = utility.checkpoint(args)

def main():
    global model
    if args.data_test == ['video']:
        from videotester import VideoTester
        model = model.Model(args, checkpoint)
        t = VideoTester(args, model, checkpoint)
        t.test()
    else:
        if checkpoint.ok:
            loader = data.Data(args)
            _model = model.Model(args, checkpoint)
            _loss = loss.Loss(args, checkpoint) if not args.test_only else None
            t = Trainer(args, loader, _model, _loss, checkpoint)
            while not t.terminate():
                t.train()
                t.test()

            checkpoint.done()

if __name__ == '__main__':
    main()
```

Data.DIV2K

```python
import os
from data import srdata

class DIV2K(srdata.SRData):
    def __init__(self, args, name='DIV2K', train=True, benchmark=False):
        data_range = [r.split('-') for r in args.data_range.split('/')]
        if train:
            data_range = data_range[0]
        else:
            if args.test_only and len(data_range) == 1:
                data_range = data_range[0]
            else:
                data_range = data_range[1]

        self.begin, self.end = list(map(lambda x: int(x), data_range))
        super(DIV2K, self).__init__(
            args, name=name, train=train, benchmark=benchmark
        )

    def _scan(self):
        names_hr, names_lr = super(DIV2K, self)._scan()
        names_hr = names_hr[self.begin - 1:self.end]
        names_lr = [n[self.begin - 1:self.end] for n in names_lr]

        return names_hr, names_lr

    def _set_filesystem(self, dir_data):
        super(DIV2K, self)._set_filesystem(dir_data)
        self.dir_hr = os.path.join(self.apath, 'DIV2K_train_HR')
        self.dir_lr = os.path.join(self.apath, 'DIV2K_train_LR_bicubic')
        if self.input_large: self.dir_lr += 'L'
```

Model.edsr

```
class EDSR(nn.Module):
    def __init__(self, args, conv=common.default_conv):
        super(EDSR, self).__init__()

        n_resblocks = args.n_resblocks
        n_feats = args.n_feats
        kernel_size = 3
        scale = args.scale[0]
        act = nn.ReLU(True)
        url_name = 'r{}f{}x{}'.format(n_resblocks, n_feats, scale)
        if url_name in url:
            self.url = url[url_name]
        else:
            self.url = None
        self.sub_mean = common.MeanShift(args.rgb_range)
        self.add_mean = common.MeanShift(args.rgb_range, sign=1)

        # define head module
        m_head = [conv(args.n_colors, n_feats, kernel_size)]

        # define body module
        m_body = [
            common.ResBlock(
                conv, n_feats, kernel_size, act=act, res_scale=args.res_scale
            ) for _ in range(n_resblocks)
        ]
        m_body.append(conv(n_feats, n_feats, kernel_size))

        # define tail module
        m_tail = [
            common.Upsampler(conv, scale, n_feats, act=False),
            conv(n_feats, args.n_colors, kernel_size)
        ]

        self.head = nn.Sequential(*m_head)
        self.body = nn.Sequential(*m_body)
        self.tail = nn.Sequential(*m_tail)
```

Model.edsr

```
n_resblocks = args.n_resblocks
n_feats = args.n_feats
kernel_size = 3
scale = args.scale[0]
act = nn.ReLU(True)
url_name = 'r{}f{}x{}'.format(n_resblocks, n_feats, scale)
if url_name in url:
    self.url = url[url_name]
else:
    self.url = None
self.sub_mean = common.MeanShift(args.rgb_range)
self.add_mean = common.MeanShift(args.rgb_range, sign=1)
```

- resblock의 수와 feature map의 수를 입력 받는다.
- 커널은 size가 3인 커널을 사용한다.
- Scale은 사용자가 입력한 scale으로 sr 후에 입력한 scale만큼 upsampling 한다.

Mdel.edsr

```
n_resblocks = args.n_resblocks
n_feats = args.n_feats
kernel_size = 3
scale = args.scale[0]
act = nn.ReLU(True)
url_name = 'r{}f{}x{}'.format(n_resblocks, n_feats, scale)
if url_name in url:
    self.url = url[url_name]
else:
    self.url = None
self.sub_mean = common.MeanShift(args.rgb_range)
self.add_mean = common.MeanShift(args.rgb_range, sign=1)
```

- 입력한 argment를 문자열으로 만들어 미리 정의된 모델중에 있는지 확인.
- Sub_mean과 add_mean은 bias로 사용한다고 함(저자)

```
url = {
    'r16f64x2': 'https://cv.snu.ac.kr/research/EDSR/models/edsr_baseline_x2-1bc95232.pt',
    'r16f64x3': 'https://cv.snu.ac.kr/research/EDSR/models/edsr_baseline_x3-abf2a44e.pt',
    'r16f64x4': 'https://cv.snu.ac.kr/research/EDSR/models/edsr_baseline_x4-6b446fab.pt',
    'r32f256x2': 'https://cv.snu.ac.kr/research/EDSR/models/edsr_x2-0edfb8a3.pt',
    'r32f256x3': 'https://cv.snu.ac.kr/research/EDSR/models/edsr_x3-ea3ef2c6.pt',
    'r32f256x4': 'https://cv.snu.ac.kr/research/EDSR/models/edsr_x4-4f62e9ef.pt'
}
```

Common.MeanShift

```
class MeanShift(nn.Conv2d):
    def __init__(
        self, rgb_range,
        rgb_mean=(0.4488, 0.4371, 0.4040), rgb_std=(1.0, 1.0, 1.0), sign=-1):

        super(MeanShift, self).__init__(3, 3, kernel_size=1)
        std = torch.Tensor(rgb_std)
        self.weight.data = torch.eye(3).view(3, 3, 1, 1) / std.view(3, 1, 1, 1)
        self.bias.data = sign * rgb_range * torch.Tensor(rgb_mean) / std
        for p in self.parameters():
            p.requires_grad = False
```

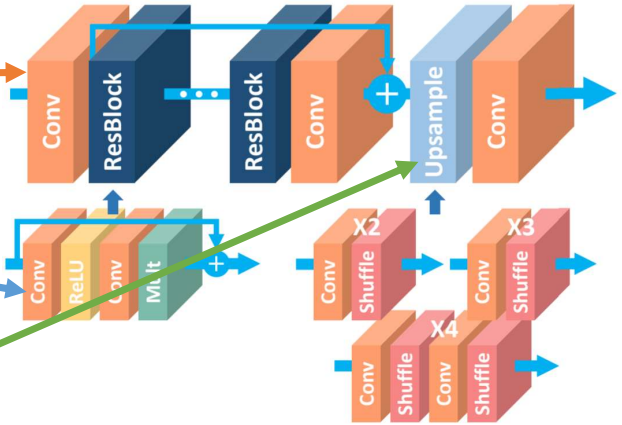
- 논문에서는 그저 preprocessing 하는 것이라고만 언급.
- 위의 (0.4488,0.4371,0.4040)은 본 논문에서 사용한 DIV2K 데이터셋의 rgb 평균
- 입력 이미지의 rgb 각 채널의 평균과 분산을 구한다. Sign은 부호를 의미(더할 건지 뺄 건지)
- Meanshift를 통해 sub_mean 값과 add_mean 값을 생성한다.
- Sub_mean은 입력 이미지에서 입력 이미지의 rgb 평균값을 뺀 값이라고 함.
- Rgb 값을 뺌으로서 shadow feature(주변 요소)을 제거하는 것이라고 함.
- 모델 연산 진행 후에 add_mean으로 뺀 값을 다시 더해준다.

Model.edsr

```python
# define head module
m_head = [conv(args.n_colors, n_feats, kernel_size)]
```

```python
# define body module
m_body = [
    common.ResBlock(
        conv, n_feats, kernel_size, act=act, res_scale=args.res_scale
    ) for _ in range(n_resblocks)
]
m_body.append(conv(n_feats, n_feats, kernel_size))
```

```python
# define tail module
m_tail = [
    common.Upsampler(conv, scale, n_feats, act=False),
    conv(n_feats, args.n_colors, kernel_size)
]
```



- 모델의 head 부분인 convolution layer, body 부분인 ResBlock, tail 부분의 upsampler
- Activation는 ReLU를 사용(옵션으로 설정 가능).
- Res_scale은 residual scaling으로 적절한 값(0.1)을 곱해주어 모델이 깊어질수록 훈련이 불안정한 것을 개선해준다.
- Upsampler는 입력한 scale 대로 upsampling 해준다.

Common.ResBlock

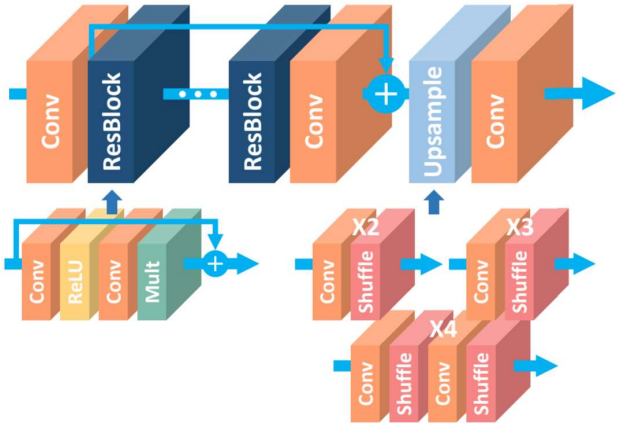
```
class ResBlock(nn.Module):
    def __init__(
        self, conv, n_feats, kernel_size,
        bias=True, bn=False, act=nn.ReLU(True), res_scale=1):

        super(ResBlock, self).__init__()
        m = []
        for i in range(2):
            m.append(conv(n_feats, n_feats, kernel_size, bias=bias))
            if bn:
                m.append(nn.BatchNorm2d(n_feats))
            if i == 0:
                m.append(act)

        self.body = nn.Sequential(*m)
        self.res_scale = res_scale

    def forward(self, x):
        res = self.body(x).mul(self.res_scale)
        res += x

        return res
```



- ResBlock을 정의
- Conv – ReLU(activation) – Conv – Mult(res_scale을 곱해줌) 으로 구성

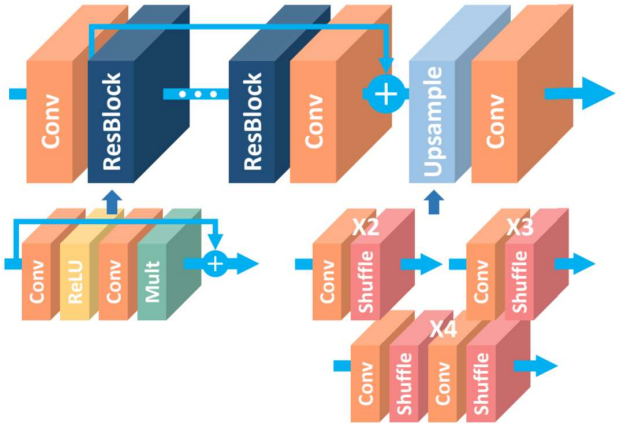
Common.Upsampler

```
class Upsampler(nn.Sequential):
    def __init__(self, conv, scale, n_feats, bn=False, act=False, bias=True):

        m = []
        if (scale & (scale - 1)) == 0:    # Is scale = 2^n?
            for _ in range(int(math.log(scale, 2))):
                m.append(conv(n_feats, 4 * n_feats, 3, bias))
                m.append(nn.PixelShuffle(2))
                if bn:
                    m.append(nn.BatchNorm2d(n_feats))
                if act == 'relu':
                    m.append(nn.ReLU(True))
                elif act == 'prelu':
                    m.append(nn.PReLU(n_feats))

        elif scale == 3:
            m.append(conv(n_feats, 9 * n_feats, 3, bias))
            m.append(nn.PixelShuffle(3))
            if bn:
                m.append(nn.BatchNorm2d(n_feats))
            if act == 'relu':
                m.append(nn.ReLU(True))
            elif act == 'prelu':
                m.append(nn.PReLU(n_feats))
        else:
            raise NotImplementedError

        super(Upsampler, self).__init__(*m)
```



- Upsampler을 정의
- 연산이 끝난 후 모델의 마지막 부분(tail)에서 결과 이미지의 크기를 키워준다.(Upsample)
- Scale이 2의 power 이라면 scale이 2인 upsampler을 이용한다.

Loss.init

```
class Loss(nn.modules.loss._Loss):
    def __init__(self, args, ckp):
        super(Loss, self).__init__()
        print('Preparing loss function:')

        self.n_GPUs = args.n_GPUs
        self.loss = []
        self.loss_module = nn.ModuleList()
        for loss in args.loss.split('+'):
            weight, loss_type = loss.split('*')
            if loss_type == 'MSE':
                loss_function = nn.MSELoss()
            elif loss_type == 'L1':
                loss_function = nn.L1Loss()
            elif loss_type.find('VGG') >= 0:
                module = import_module('loss.vgg')
                loss_function = getattr(module, 'VGG')(
                    loss_type[3:],
                    rgb_range=args.rgb_range
                )
            elif loss_type.find('GAN') >= 0:
                module = import_module('loss.adversarial')
                loss_function = getattr(module, 'Adversarial')(
                    args,
                    loss_type
                )
```

- Loss는 입력한 args에 따라 다양한 loss 사용 가능
- 하지만 논문에서 설명한대로 L1 loss를 사용하여 성능 개선을 하였으므로 기본적으로 L1 loss를 사용한다.

Trainer.init

```
class Trainer():
    def __init__(self, args, loader, my_model, my_loss, ckp):
        self.args = args
        self.scale = args.scale

        self.ckp = ckp
        self.loader_train = loader.loader_train
        self.loader_test = loader.loader_test
        self.model = my_model
        self.loss = my_loss
        self.optimizer = utility.make_optimizer(args, self.model)

        if self.args.load != '':
            self.optimizer.load(ckp.dir, epoch=len(ckp.log))

        self.error_last = 1e8
```

- Args를 읽어 Train, Test에 사용할 data_loader, model, loss, optimizer을 설정

Trainer.init

```
class Trainer():
    def __init__(self, args, loader, my_model, my_loss, ckp):
        self.args = args
        self.scale = args.scale

        self.ckp = ckp
        self.loader_train = loader.loader_train
        self.loader_test = loader.loader_test
        self.model = my_model
        self.loss = my_loss
        self.optimizer = utility.make_optimizer(args, self.model)

        if self.args.load != '':
            self.optimizer.load(ckp.dir, epoch=len(ckp.log))

        self.error_last = 1e8
```

- Args를 읽어 Train, Test에 사용할 data_loader, model, loss, optimizer을 설정