Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network

Ruchit Vithani (201701070)
Purvil Mehta (201701073)
Bhargey Mehta (201701074)
Kushal Shah (201701111)

Ref: Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network
Code : Code Link

Motivation and Context

- Image Super Resolution is a widespread problem and have several model with great accuracy and speed up including deep convolutional network.
- But the central problem of these models of lacking high frequency component or features remains the same.
- The image with the less MSE and high PSNR need not to be necessarily perceptually look good. This motivates to propose perceptual loss in addition to MSE loss to get the better features compared to previously generated fake HR images.

Example

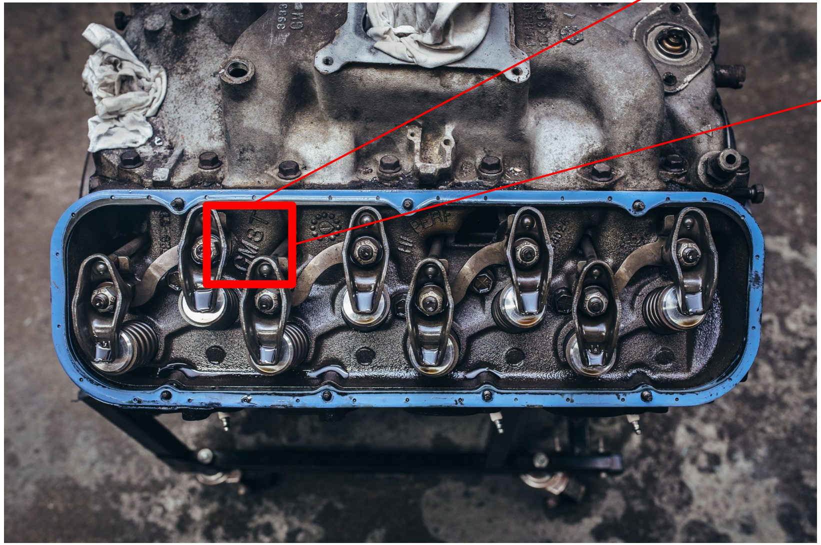


Image Source: DIV2K Dataset

Related Work

31st July, 2015

Image Super-Resolution Using Deep Convolutional Networks
Chao Dong, Chen Change Loy, Member, IEEE, Kaiming He, Member, IEEE, and Xiaoou Tang, Fellow, IEEE

15th October, 2015

Deep Networks for Image Super-Resolution with Sparse Prior
Zhaowen Wangyz Ding Liuy Jianchao Yangx Wei Hany Thomas Huangy

1st August, 2016

Accelerating the Super-Resolution Convolutional Neural Network
Chao Dong, Chen Change Loy, and Xiaoou Tang

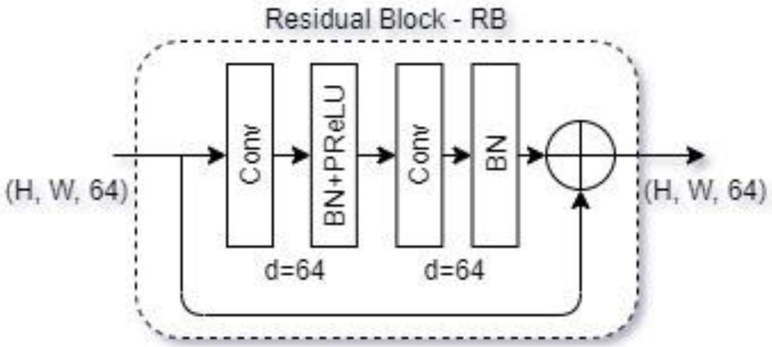
11th November, 2016

Deeply-Recursive Convolutional Network for Image Super-Resolution
Jiwon Kim, Jung Kwon Lee and Kyoung Mu Lee
Department of ECE, ASRI, Seoul National University, Korea

Enter: GAN - Bhargey Mehta (201701074)

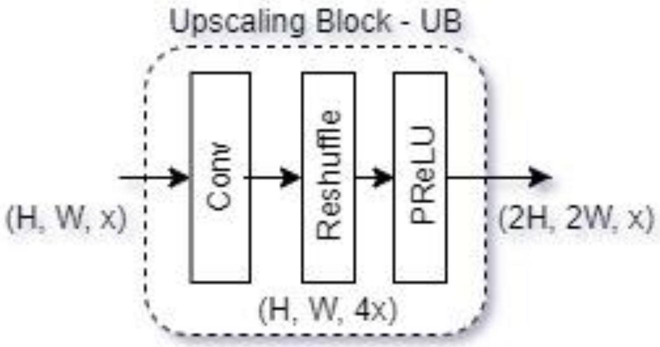
- GAN- imagine a forger and an authenticator teaching each other to be better
- Generative Network - take a seed $z^{(i)}$ and produce $G(z^{(i)})$ as close as possible to the original data distribution X
- Discriminative Network - identify that $G(z^{(i)})$ does NOT belong to the original data distribution X
- In SISR context:
  - $z^{(i)} \rightarrow$ Low Resolution Input Image
  - $G(z^{(i)}) \rightarrow$ Super Resolution Generated Image
  - $X \rightarrow$ High Resolution Original Image
- Aim: Generate super resolution images from input images which are as close as possible to the original high resolution images.

Architecture (Generator)

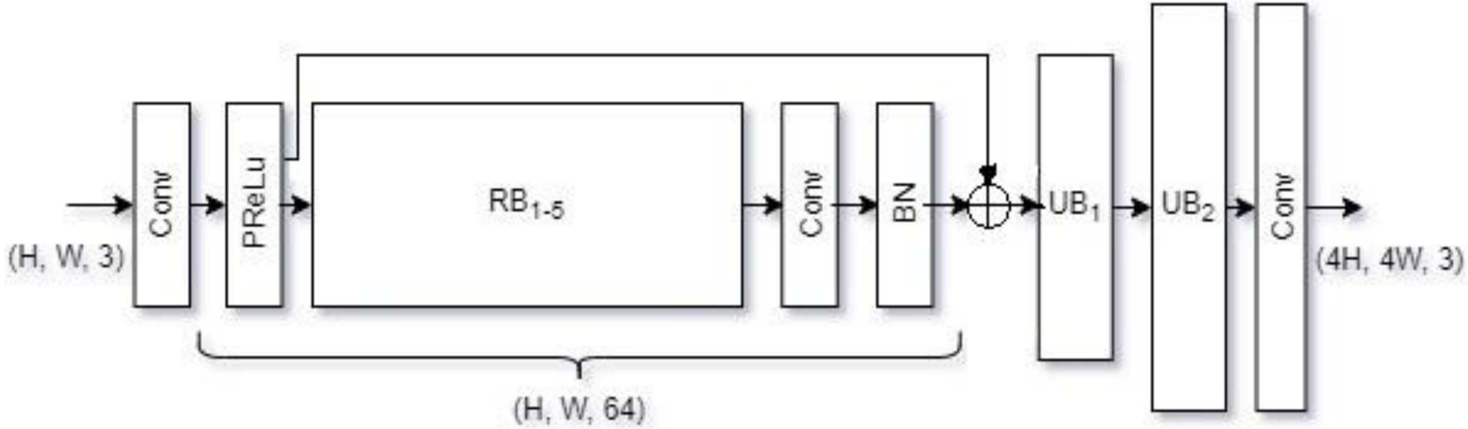



```python
class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(channels, channels,
                               kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(channels)
        self.prelu = nn.PReLU()
        self.conv2 = nn.Conv2d(channels, channels,
                               kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(channels)

    def forward(self, x):
        residual = self.conv1(x)
        residual = self.bn1(residual)
        residual = self.prelu(residual)
        residual = self.conv2(residual)
        residual = self.bn2(residual)

        return x.clone() + residual
```

Architecture (Generator)



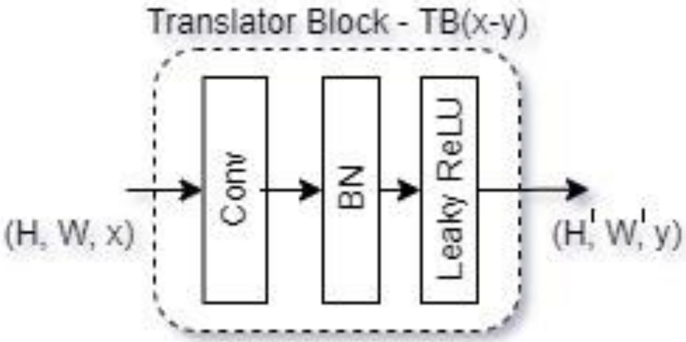

```python
class UpsampleBLock(nn.Module):
    def __init__(self, in_channels, up_scale):
        super(UpsampleBLock, self).__init__()
        self.conv = nn.Conv2d(in_channels, in_channels * up_scale ** 2,
                              kernel_size=3, padding=1)
        self.pixel_shuffle = nn.PixelShuffle(up_scale)
        self.prelu = nn.PReLU()

    def forward(self, x):
        x = self.conv(x)
        x = self.pixel_shuffle(x)
        x = self.prelu(x)
        return x
```

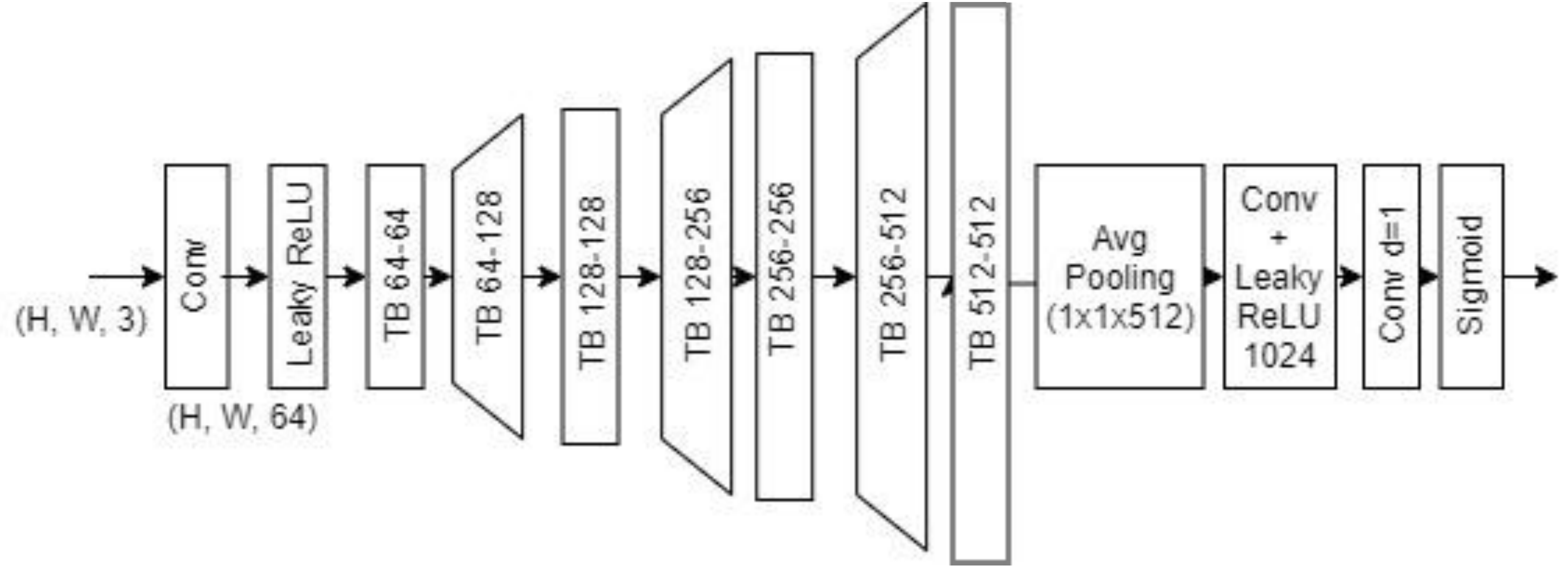



```python
class Generator(nn.Module):
    def __init__(self, scale_factor):
        upsample_block_num = int(math.log(scale_factor, 2))
        super(Generator, self).__init__()
        self.block1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=9, padding=4),
            nn.PReLU())
        self.block2 = ResidualBlock(64)
        self.block3 = ResidualBlock(64)
        self.block4 = ResidualBlock(64)
        self.block5 = ResidualBlock(64)
        self.block6 = ResidualBlock(64)
        self.block7 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64))
        block8 = [UpsampleBLock(64, 2) for _ in range(upsample_block_num)]
        block8.append(nn.Conv2d(64, 3, kernel_size=9, padding=4))
        self.block8 = nn.Sequential(*block8)
```

```python
    def forward(self, x):
        block1 = self.block1(x)
        block2 = self.block2(block1)
        block3 = self.block3(block2)
        block4 = self.block4(block3)
        block5 = self.block5(block4)
        block6 = self.block6(block5)
        block7 = self.block7(block6)
        block8 = self.block8(block1.clone() + block7)

        return block8
```

Architecture (Discriminator)



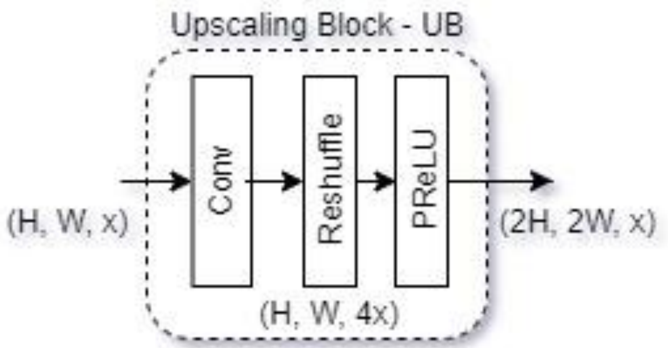


Architecture (Discriminator)

```python
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.net = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.LeakyReLU(0.2, inplace=False),

            nn.Conv2d(64, 64, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2, inplace=False),

            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=False),

            nn.Conv2d(128, 128, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=False),

            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=False),

            nn.Conv2d(256, 256, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=False),

            nn.Conv2d(256, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=False),

            nn.Conv2d(512, 512, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=False),

            nn.AdaptiveAvgPool2d(1),
            nn.Conv2d(512, 1024, kernel_size=1),
            nn.LeakyReLU(0.2, inplace=False),
            nn.Conv2d(1024, 1, kernel_size=1)
        )

    def forward(self, x):
        batch_size = x.size(0)
        x1 = self.net(x)
        x2 = x1.view(batch_size)
        x3 = torch.sigmoid(x2)

        return x3
```

Advocate 1

- A noteworthy feature of this architecture is that it utilises a learned upscaling layer instead of first using some interpolation for upscaling like in previous works.
- This reduces the computational cost since now the kernels operate on a smaller image instead of the image of final size.



Loss Functions - Purvil Mehta (201701073)

- As discussed previously, MSE is the pixel wise average of the square of difference between two images which generates the overly smooth images and has very less perceptual quality. Thus we proposed new loss function to overcome this fact.
- ***Perceptual loss:*** is defined as a weighted sum of Content loss and Adversarial loss.

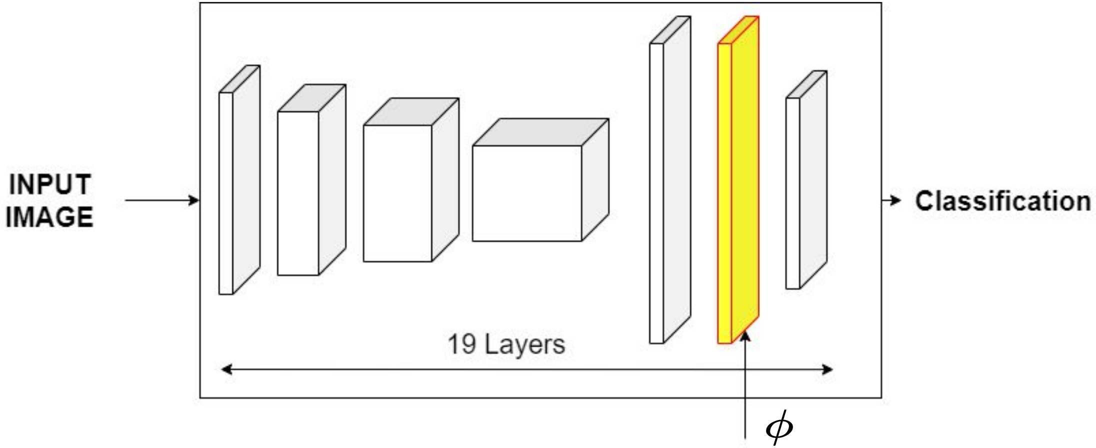
$$l^{SR} = \underbrace{l_X^{SR}}_{\text{Content Loss}} + \underbrace{10^{-3} l_{Gen}^{SR}}_{\text{Adversarial Loss}}$$

Loss Functions

- ***Content loss*** : Is generally MSE between two images. Here we considered also MSE between feature maps of VGG19 network at final layer.

$$l_{\mathrm{MSE}}^{\mathrm{SR}} = \frac{1}{r^2WH} \sum_{x=1}^{rW} \sum_{y=1}^{rH} (I_{x,y}^{HR} - G_{\theta_G}(I^{LR})_{x,y})^2$$

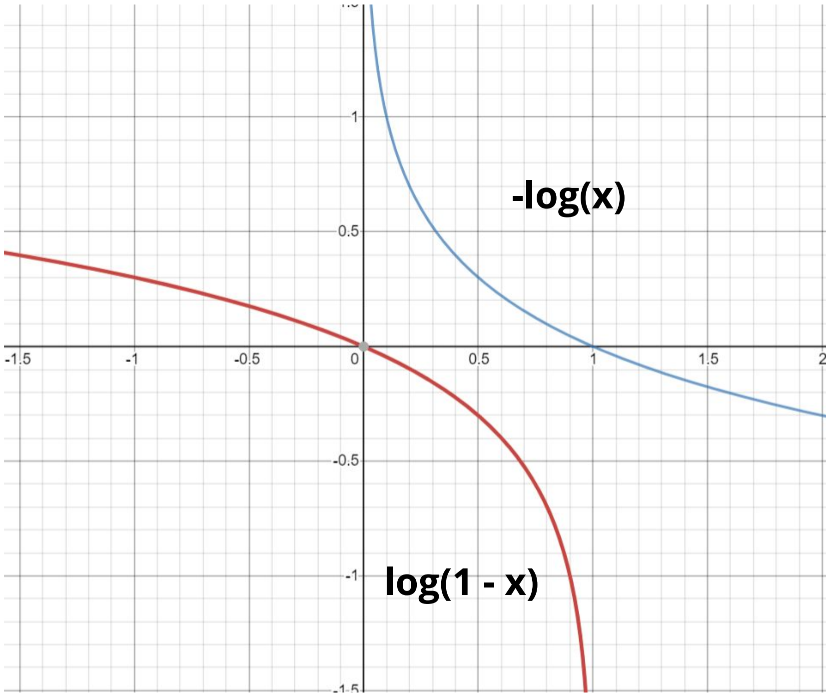
$$l_{\mathrm{VGG}}^{\mathrm{SR}} = \frac{1}{WH} \sum_{x=1}^{W} \sum_{y=1}^{H} (\phi(I^{HR})_{x,y} - \phi(G_{\theta_G}(I^{LR}))_{x,y})^2$$



Loss Functions

- ***Adversarial loss :*** Encourages G network to generate solutions that fool D network. This increases naturally appealing results.
- $D_{\theta_D}(G_{\theta_G}(I^{LR}))$ is a probability that the generated image $G_{\theta_G}(I^{LR})$ is a natural HR image.
- For better gradient behaviour, we minimize $-log(D_{\theta_D}(G_{\theta_G}(I^{LR})))$ instead of minimize $log(1 - D_{\theta_D}(G_{\theta_G}(I^{LR})))$

$$l_{Gen}^{SR} = \sum_{n=1}^{N} -log(D_{\theta_D}(G_{\theta_G}(I^{LR})))$$



How to code this?

Losses - Code

```
[ ]
class GeneratorLoss(nn.Module):
    def __init__(self):
        super(GeneratorLoss, self).__init__()
        vgg = vgg19(pretrained=True)
        loss_network = nn.Sequential(*list(vgg.features)[:34]).eval()
        for param in loss_network.parameters():
            param.requires_grad = False
        self.loss_network = loss_network
        self.mse_loss = nn.MSELoss()

    def forward(self, out_labels, out_images, target_images):
        # Adversarial Loss
        adversarial_loss = torch.mean(-torch.log(out_labels + 1e-6))
        # Perception Loss
        perception_loss = self.mse_loss(self.loss_network(out_images), self.loss_network(target_images))
        # Image Loss
        image_loss = self.mse_loss(out_images, target_images)

        return  image_loss + 0.001 * adversarial_loss + 0.006 * perception_loss

```

Fake HR (G(z))

Original HR (z)

D(G(z))

Training - Parameters - Ruchit Vithani (201701070)

- Dataset used : ***DIV2K dataset*** which contains 800 high definition training images and 100 high definition validation images.
- During training, we extract a random crop of size 96x96 from each image for training.
- Other hyperparameters :
  - Leaky Relu (α = 0.2)
  - Adam optimizer (β = 0.9, 0.99)
  - All our images have been scaled in the range of [0, 1]
  - Pretrained VGG19 model as feature extractor and for calculation of perceptual loss. We use features from final layer of VGG19
  - Epochs : 150
- We understood the pytorch implementation of the code and trained the G and D model to get the Super resolution images on the said dataset. We used pre trained VGG 19 model (last layer features) to train GAN model in google colab.

Training process

(1) Model initialisation

   (a) Initialise Generator network G

   (b) Initialise Discriminator network D

```
netG = Generator(UPSCALE_FACTOR)
print('# generator parameters:', sum(param.numel() for param in netG.parameters()))
netD = Discriminator()
print('# discriminator parameters:', sum(param.numel() for param in netD.parameters()))
```

Training process

(2) Dataloader loads mini batches of images from local directory.

```
class TrainDatasetFromFolder(Dataset):
    def __init__(self, dataset_dir, crop_size, upscale_factor):
        super(TrainDatasetFromFolder, self).__init__()
        self.image_filenames = [join(dataset_dir, x) for x in listdir(dataset_dir)]
        crop_size = calculate_valid_crop_size(crop_size, upscale_factor)
        self.hr_transform = train_hr_transform(crop_size)
        self.lr_transform = train_lr_transform(crop_size, upscale_factor)

    def __getitem__(self, index):
        hr_image = self.hr_transform(Image.open(self.image_filenames[index]))
        lr_image = self.lr_transform(hr_image)
        return lr_image, hr_image

    def __len__(self):
        return len(self.image_filenames)
```

From original HR image, returns a random square crop of size **`crop_size`**

Downsamples the HR patch, to obtain its corresponding LR patch

```
train_set = TrainDatasetFromFolder(train_path, crop_size=CROP_SIZE, upscale_factor=UPSCALE_FACTOR)
train_loader = DataLoader(dataset=train_set, num_workers=4, batch_size=16, shuffle=True)
```

Training process

(3) Update Discriminator network.

(a) Compute `D(x) (real p)`. (label = 1)

(b) Compute `D(G(z)) (fake p)`. (label = 0)

(c) Maximize following function.

```
d_loss = log(D(x)) + log(1-D(G(z)))
```

(d) Update the parameters of D, keeping G same.

```
############################
# (1) Update D network: maximize log(D(x)) + log(1-D(G(z)))
###########################
netD.zero_grad()

real_img = torch.Tensor(target)
if torch.cuda.is_available():
    real_img = real_img.cuda()
z = torch.Tensor(data)
if torch.cuda.is_available():
    z = z.cuda()
fake_img = netG(z)

real_out = netD(real_img)
labels = torch.ones(real_out.shape).cuda()
errD_real = bce_loss(real_out, labels)
errD_real.backward()
D_x = real_out.mean().item()


fake_out = netD(fake_img.detach())
labels = torch.zeros(fake_out.shape).cuda()
errD_fake = bce_loss(fake_out, labels)
errD_fake.backward()
D_G_z = fake_out.mean().item()

d_loss = errD_real + errD_fake
optimizerD.step()
```

Training process

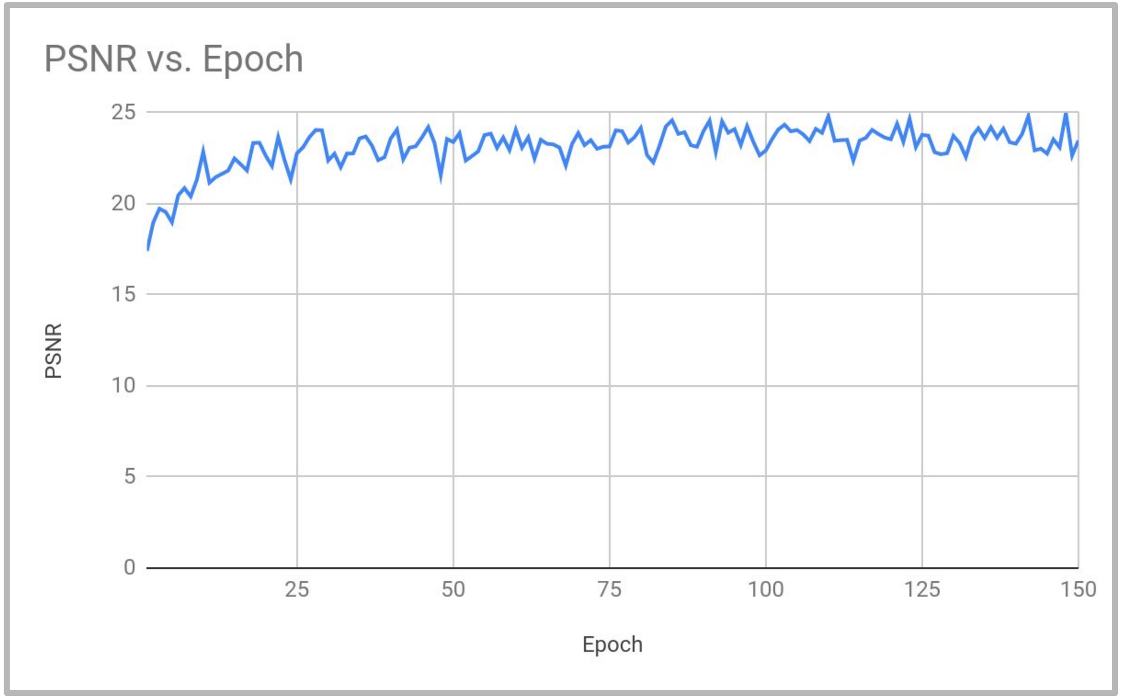
(4) Update Generator network.

```
############################
# (2) Update G network: minimize -log(D(G(z))) + Perception Loss + Image Loss
###########################
netG.zero_grad()
labels = torch.ones(real_out.shape, dtype=torch.long)
fake_out = netD(fake_img)
D_G_z = fake_out.mean().item()
g_loss = generator_criterion(fake_out, fake_img, real_img)
g_loss.backward()
optimizerG.step()
```

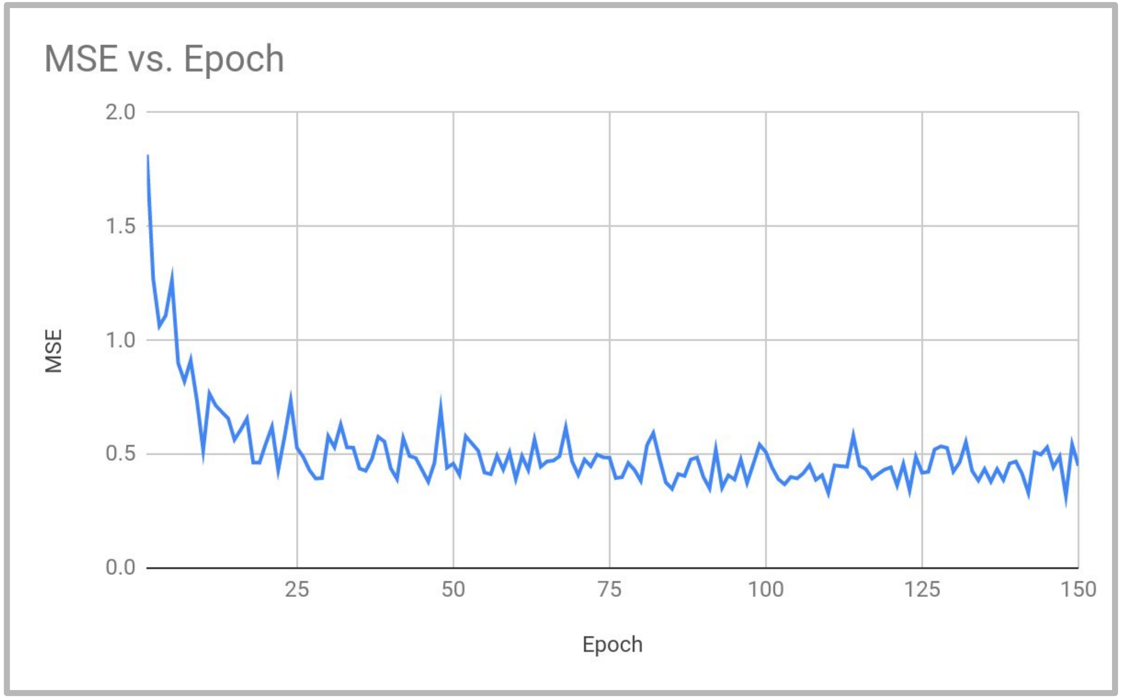
(a) Minimize perceptual loss

We repeat this training process for all mini-batches we obtain from dataloader, and for each epoch.

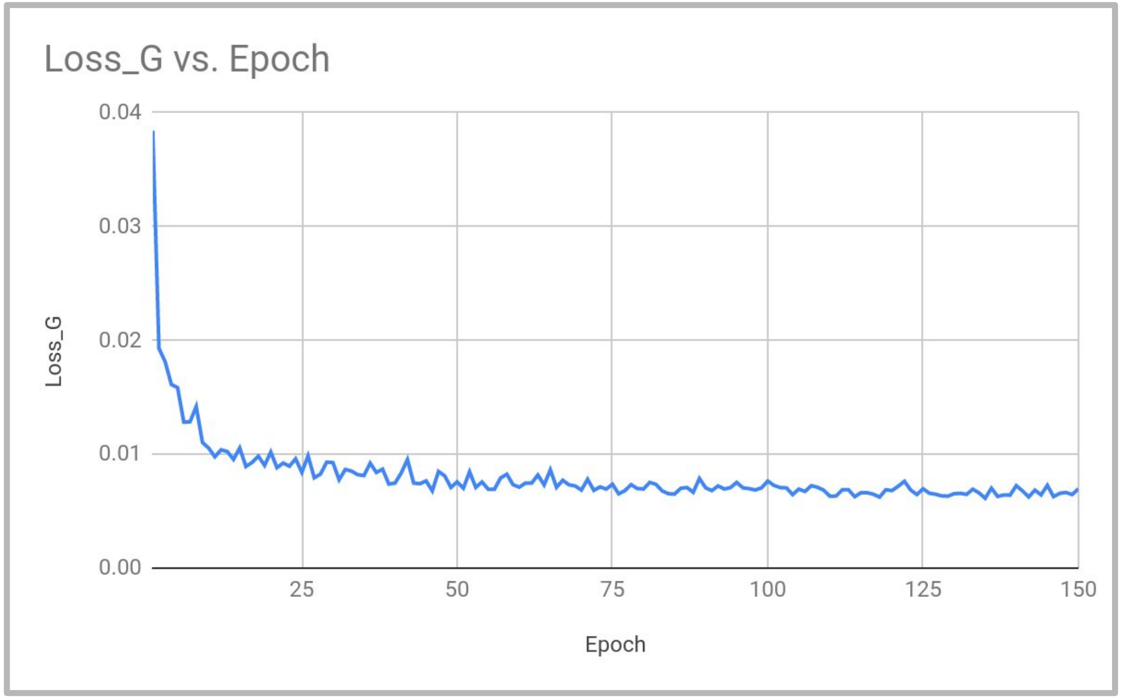
PSNR vs. Epochs on validation set



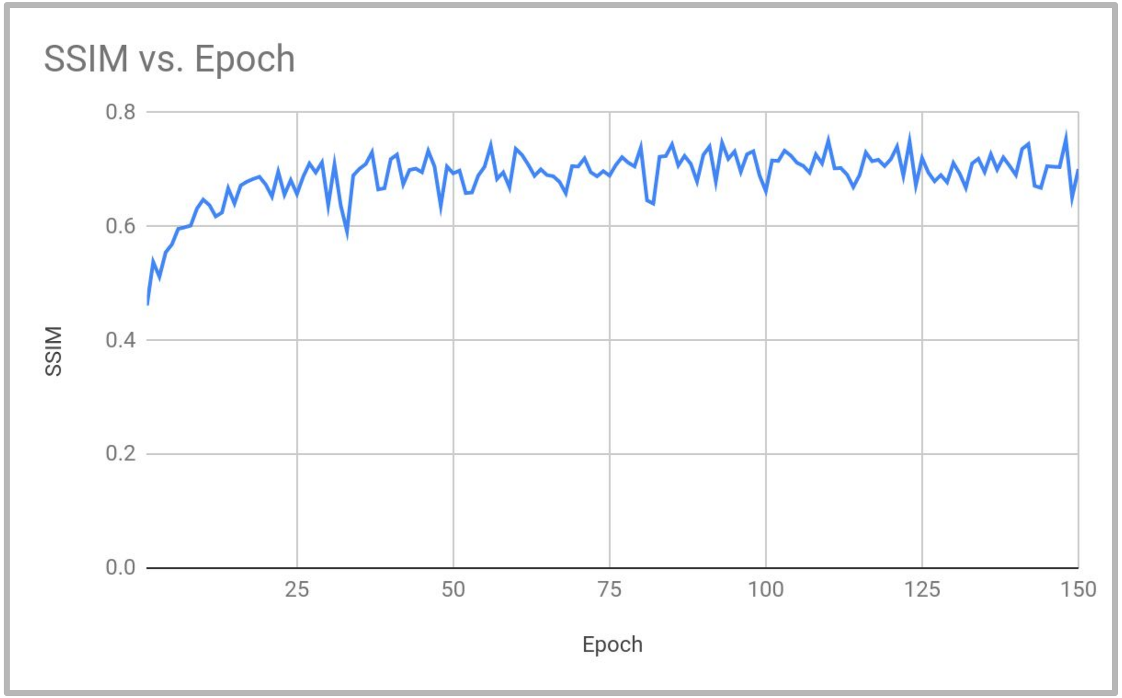
MSE of validation set images



Generator loss vs. Epochs



SSIM scores of validation set images



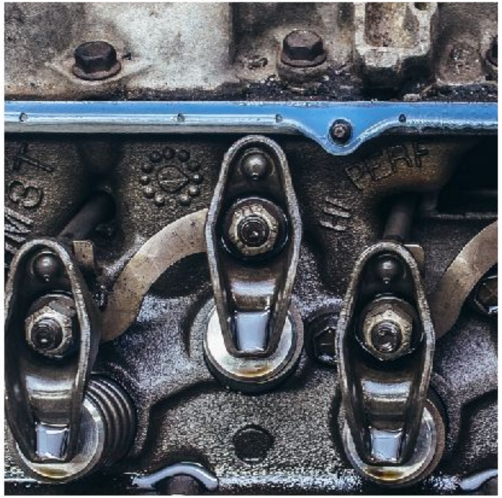
Training Results - 1

Original Image

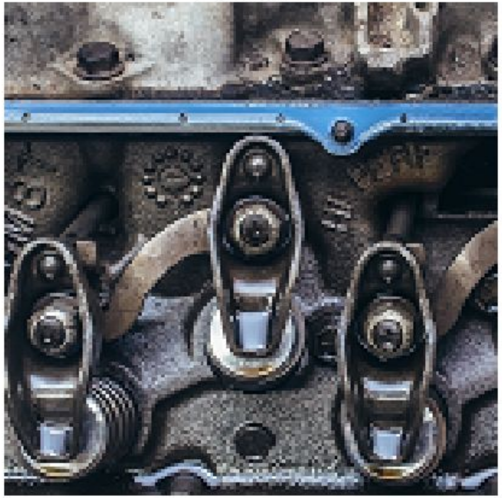
SRGAN MSE: 0.0054852

Training Results - 2

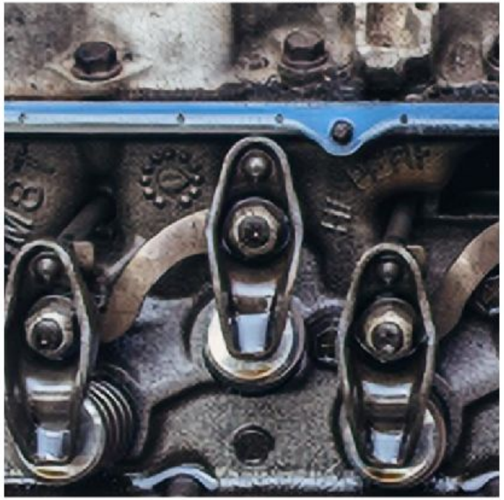
Original Image



LR Image

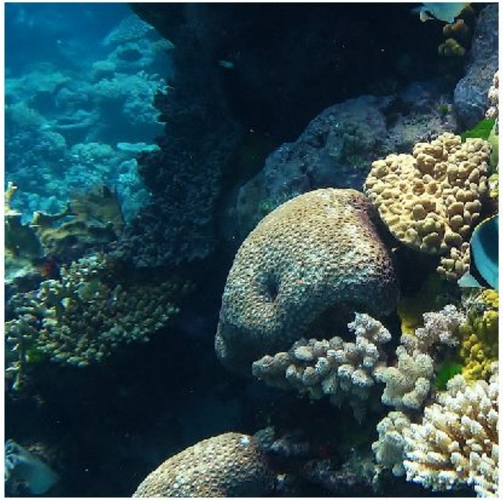


SRGAN MSE: 0.011705

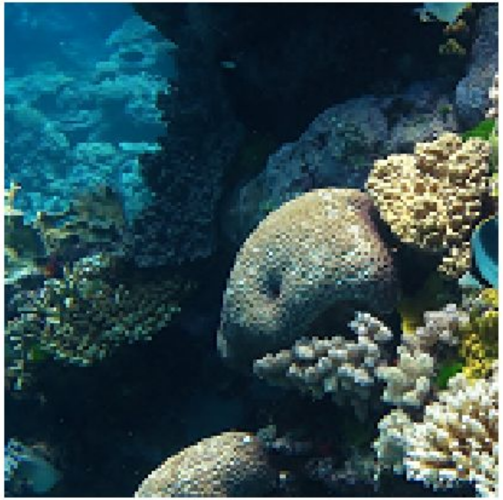


Training Results - 3

Original Image



LR Image



SRGAN MSE: 0.01252

Advocate 2

- For all HR patches, LR patches are nearly similar. When we feed all pairs to CNN, with MSE loss, our solution when minimising MSE, arrives at the solution which is pixel-wise average of all possible solution. Thus, this solution is smoother.
- In contrast to this, GAN drives the reconstruction towards the natural image manifold producing perceptually more convincing solutions. This is true due to the fact the generator samples the image from the distribution it estimates for super-resolution image.

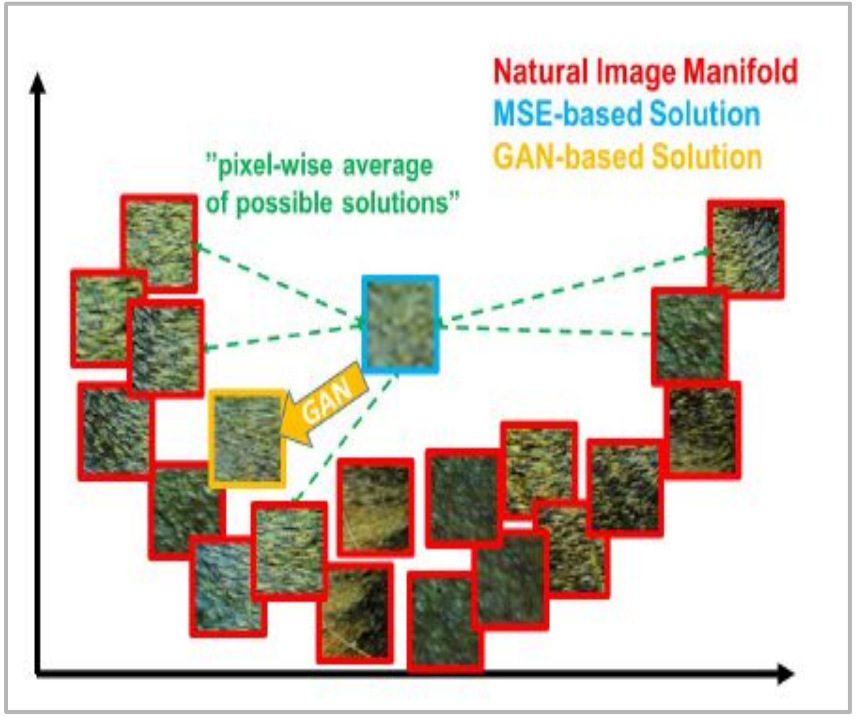


Figure illustration of patches from the natural image manifold (red) and super-resolved patches obtained withMSE (blue) and GAN (orange).

Image Source: Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network

Conclusions - Kushal Shah (201701111)

- The paper supports the use of deeper network architecture as they allow the mappings of very high complexity.
- The difficulties in training these deep networks are resolved by skip-connections and batch-normalisation.
- A major highlight of the paper is novel perceptual loss that makes the super-resolved image visually attractive.
- By experiments and mean-opinion-score, the writers confirm that images super-resolved by SRGAN are visually more similar to original HR images.

Devil's Advocate

Assertion:

- The model tries to hallucinate the details.

Reasons:

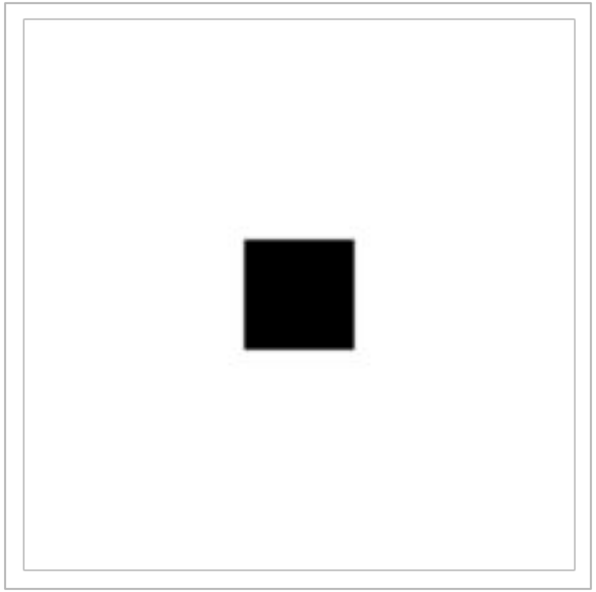
- The generative network is trained using the perceptual loss.
- The perceptual loss pushes the network to model the mappings such that every smooth pattern is mapped to high-frequency, textured patterns.
- This essentially means that we are creating the artificial details which might have no connection with ground reality.
- Such artificial fill-up of the patterns make the model unsuitable for surveillance and medical applications.

Example:

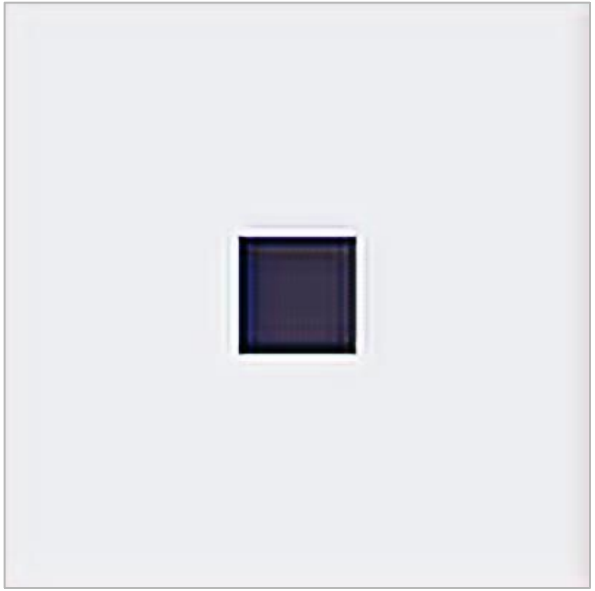
Bicubic Interpolation

SR GAN

Example:



Original High Resolution



SRGAN

Thank You