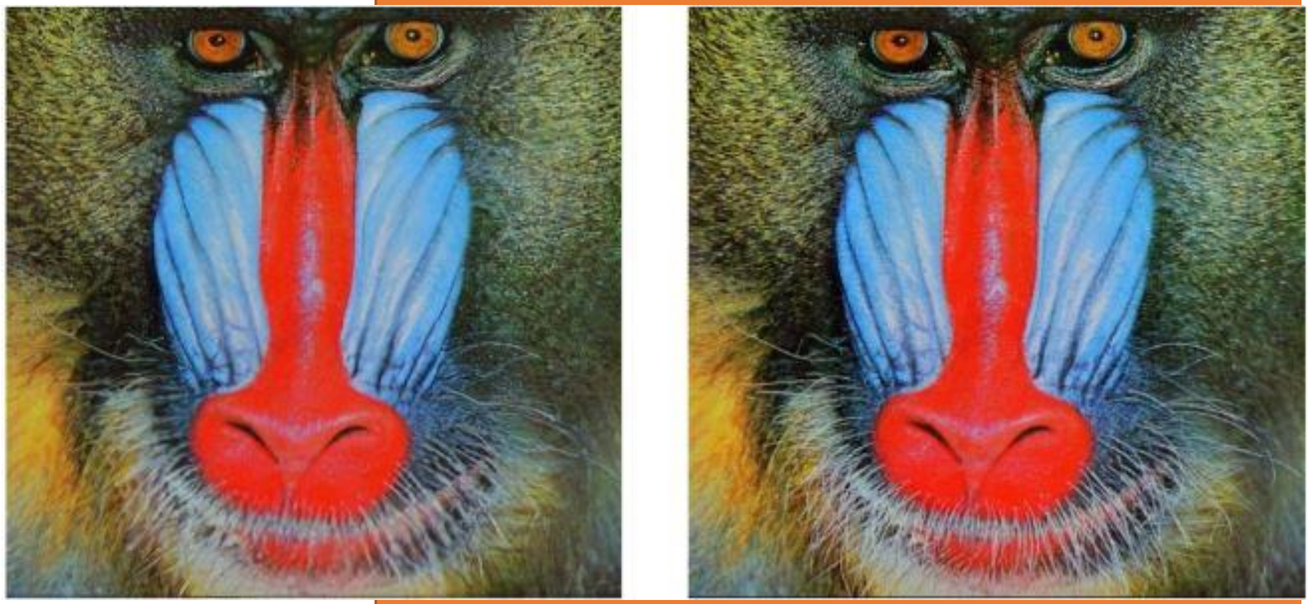
Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network



Vidyadhari Chinthala(50290085)

Disha Mehra (50288911)

Ravi Teja Sunkara (50292191)

Abstract

SRGAN, a generative adversarial network (GAN) for image super-resolution (SR) is an approach for improving imaging system. Recently, models based on convolutional neural networks, specifically designed for single image super-resolution, became a topic of research and shown great results. Models based on convolutional neural networks outperform other approaches in terms of image quality metrics such as peak signal to noise ratio (PSNR) and structural similarity (SSIM). The perceptual image quality of resulting super-resolved image is principally dependant on choice of a loss function, which is optimized during model training. Recent work is largely based on optimizing mean squared reconstruction error. Such loss does improve wieldy used image quality metrics. Nevertheless, these reconstruction metrics like PSNR and SSIM may not capture fine details in the image and give high scores to images with unsatisfying quality. SRGAN is a framework capable of inferring photo-realistic natural images for 4x4 upscaling factors. To achieve this, SRGAN uses a perceptual loss function, which consists of an adversarial loss and a content loss. The adversarial loss pushes the solution to the natural image manifold using a discriminator network that is trained to differentiate between the super-resolved images and original photo-realistic images. In addition, it uses a content loss motivated by perceptual similarity instead of similarity in pixel space. The deep residual network is able to recover photo-realistic textures from heavily downsampled images on public benchmarks. Extensive quality testing including metrics with a use of human opinion had shown that this approach generates more pleasant images. The goal of this project is to implement super-resolution model based on GAN and test quality of image reconstruction.

1. Introduction

The highly challenging task of estimating a high-resolution (HR) image from its low-resolution (LR) counterpart is referred to as super-resolution (SR). SR received substantial attention from within the computer vision research community and has a wide range of applications like satellite and aerial image analysis, medical image processing, compressed image/video enhancement etc.

The ill-posed nature of the underdetermined SR problem is particularly pronounced for high upscaling factors, for which texture detail in the reconstructed SR images is typically absent. There are various ways of enhancing image quality. One of the most commonly used technique is interpolation. This is easy to use but this leads to distorted image or reduces the visual quality of image. Most common interpolation methods produce blurry images, i.e. bi-cubic interpolation. More sophisticated methods exploit internal similarities of a given image or, use data-sets of low-resolution images and their high-resolution counterparts to effectively learn a mapping between them. Among Example-Based SR algorithms, the Sparse-Coding-Based method is one of the most popular.
Deep learning provides better solution to get optimized images. In recent years many methods have been proposed for Image Super Resolution. The optimization target of supervised SR algorithms is commonly the minimization of the mean squared error (MSE) between the recovered HR image and the ground truth. This is convenient as minimizing MSE also maximizes the peak signal-to-noise ratio (PSNR), which is a common measure used to evaluate and compare SR algorithms. However, the ability of MSE (and PSNR) to capture perceptually relevant differences, such as high texture detail, is very limited as they are defined based on pixel-wise image differences. The highest PSNR does not necessarily reflect perceptually better SR result.
Super-resolution generative network (SRGAN) employs a deep residual network (ResNet) with skip connections and diverges from MSE as the sole optimization target. SRGAN in this paper implements

a perceptual loss using high-level feature maps of the VGG network combined with discriminator that encourages solutions perceptually hard to distinguish from the HR reference images.

2. Adversarial Network Architecture

Following Goodfellow et al., we define a Discriminator network D_{θ_D}, which we optimize in an alternating manner along with G_{θ_G} to solve the adversarial min-max problem.

$$\min_{\theta_G} \max_{\theta_D} \mathbb{E}_{I^{HR} \sim p_{\text{train}}(I^{HR})}[\log D_{\theta_D}(I^{HR})] + \mathbb{E}_{I^{LR} \sim p_G(I^{LR})}[\log(1 - D_{\theta_D}(G_{\theta_G}(I^{LR}))] \tag{2}$$

The general idea behind this formulation is that it allows one to train a generative model G with the goal of fooling a differentiable discriminator D that is trained to distinguish super-resolved images from real images. With this approach our generator can learn to create solutions that are highly similar to real images and thus difficult to classify by D. This encourages perceptually superior solutions residing in the subspace, the manifold, of natural images. This is in contrast to SR solutions obtained by minimizing pixel-wise error measurements, such as the MSE.

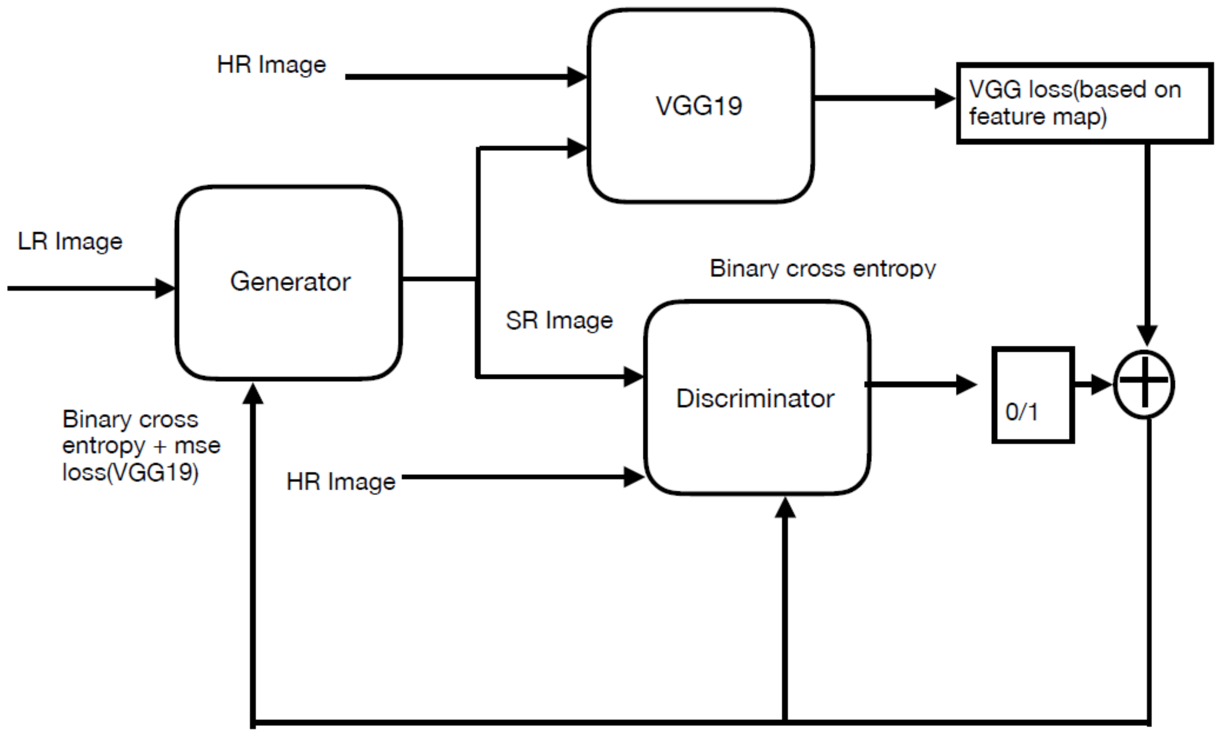


Fig: Block diagram of SRGAN

2.1 Training procedure steps

- We read the original image using cv2.imread() function and resize it to (224, 224) using bi-cubic interpolation to get High Resolution (HR) images.
- We process the HR (High Resolution) images to get down-sampled (by a factor of 4) LR (Low Resolution) images of shape (56, 56). Now we have both HR and LR images for training data set.
- The weights for VGG19 are loaded and the vgg_model.trainable is made false with loss function as mean squared error.
- Generator and discriminator models are built using the architectures explained in the following sections.
- Combined GAN model is constructed.
- Compile the combined GAN model & discriminator network with VGG + binary cross entropy and Binary Cross Entropy respectively.
- For each epoch datagen function is called which gives LR and HR images of specified batch size.
- We pass LR images through Generator, which generates an up-sampled image (by a factor of 4) resulting in Super Resolution (SR) images of shape (224, 224).
- Target label for HR images is set as 1 with some randomness and for SR images zero (0). Discriminator model with this as input is trained by setting discriminator.trainable = True
- Once the discriminator is trained the generator is trained by using the combined model and also by setting discriminator.trainable = False i.e. freezing the discriminator.
- The above procedure is repeated until the model converges.
- Once the model is trained, the weights are saved and these weights are further used to generate images from Validation dataset and to calculate PSNR values. This is performed on 3 different datasets.

<u>Block Diagram explanation of Generator and Discriminator:</u>

We train the Discriminator first and then the Generator since we freeze the weights of the discriminator only the losses for the generator are back propagated while knowing the loss of the discriminator.

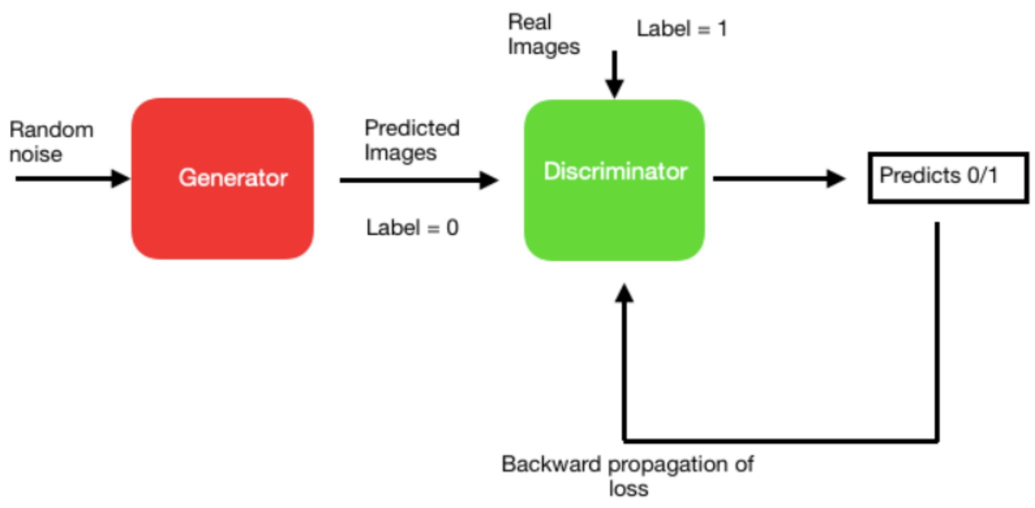


Figure 3a: Training just the Discriminator

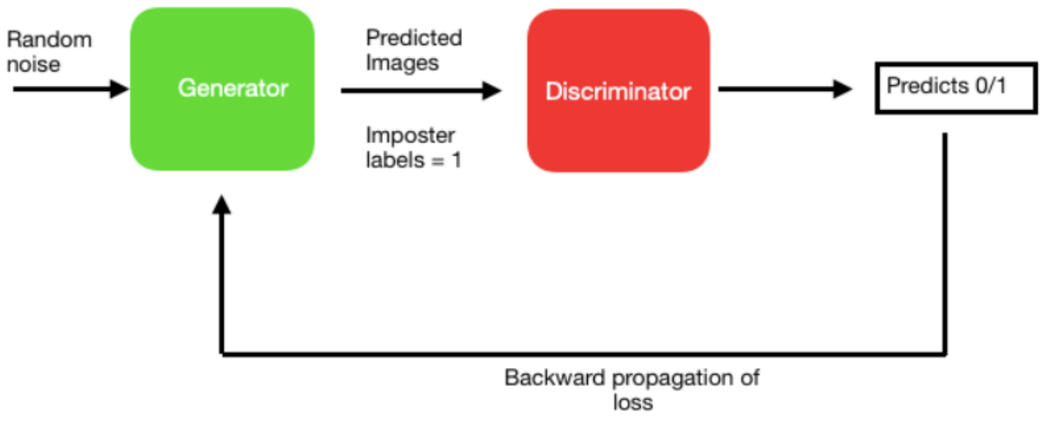
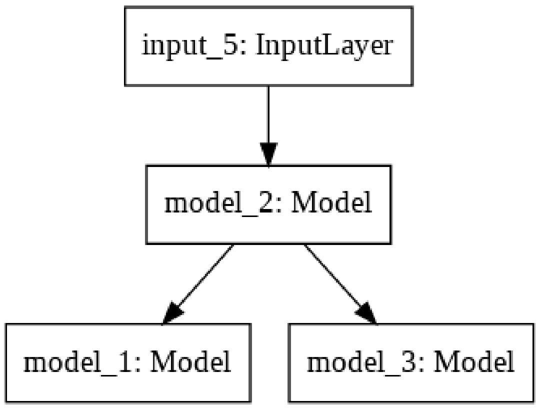


Figure 3c: Training just the Generator

GAN Model:



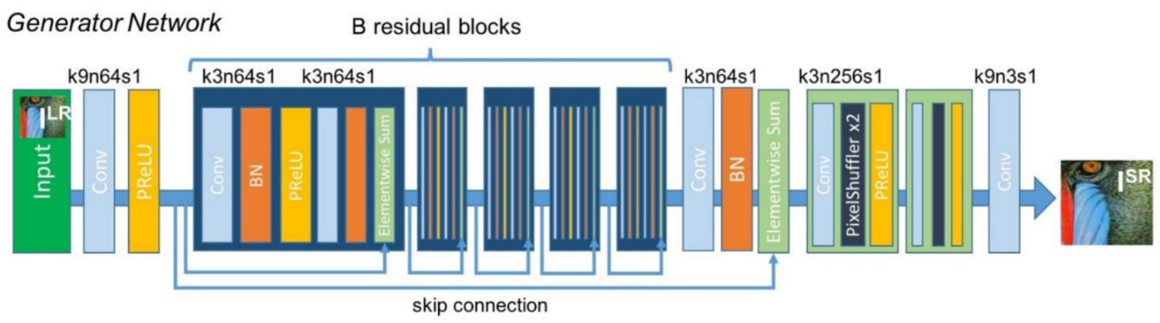
GAN Summary:

```
____________________________________________________________________________
Layer (type)                     Output Shape          Param #     Connected
to
============================================================================
input_5 (InputLayer)            (None, 56, 56, 3)     0

____________________________________________________________________________
_______________________
model_2 (Model)               (None, 224, 224, 3)  1554755     input_5[0][0]
___________________________________________________________________________
model_1 (Model)               (None, 56, 56, 256)  143667240   model_2[1][0]
__________________________________________________________________________
model_3 (Model)                (None, 1)           107455553   model_2[1][0]
============================================================================
Total params: 252,677,548
Trainable params: 1,550,531
Non-trainable params: 251,127,017

____________________________________________________________________________
```

2.2 Generator



The core of generator network consists of 16 residual blocks with identical layout. . Each block consists of two convolutional layers with small kernels of size 3x3 and 64 feature maps. Feature maps are followed by batch-normalization and ParametericReLu as the activation function. These are further followed by layers of PixelShufflers/Up sampling blocks with a scaling factor of two, which are inserted to accommodate various upscale ratios. The generator also implements skip connections similar to ResNet. A skip connection is present from input of each residual block to its output and also one more skip connection is present where the input before the first residual block is taken and is added to the output the last residual block.

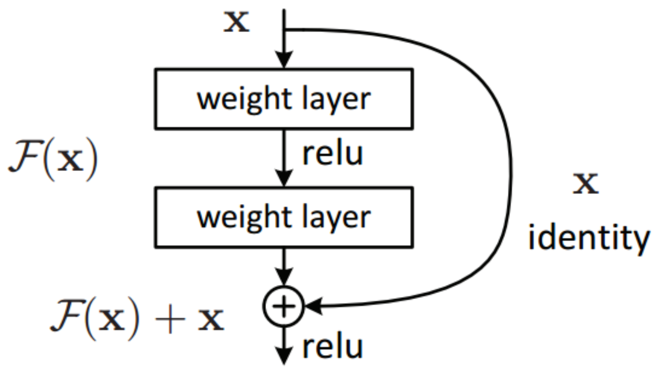
We are using PRelu in place of Relu or LeakyRelu. It introduces learn-able parameter that makes it possible to adaptively learn the negative part coefficient.

<u>Generator Summary of Parameters:</u>

```
Total params: 1,554,755
Trainable params: 1,550,531
Non-trainable params: 4,224
```

2.1.1 Residual Blocks

These are the building blocks of ResNet. In traditional neural networks, each layer feeds into the next layer. In a network with residual blocks, each layer feeds into the next layer and directly into the layers about 2–3 hops away.


x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Neural networks are universal function approximators and that the accuracy of the neural network increases with the number of layers. But there is a limit to the number of layers to improve the accuracy of neural network. If we have sufficiently deep neural networks, it may not be able to learn and improve accuracy after some layers because of the problems such as vanishing gradients and curse of dimensionality.

In addition, if we still keep increasing the number of layers, we will see that the accuracy will start to saturate at one point and eventually degrade. In addition, this is usually not caused due to overfitting. Therefore, it might seem that the shallower networks are learning better than their deeper counterparts and this is quite counter-intuitive. However, this is what is seen in practice and is popularly known as the degradation problem.

In degradation problem, we know that shallower networks perform better than the deeper counterparts that have few more layers added to them. So, why not skip these extra layers and at-least match the accuracy of the shallow sub networks. You can skip the training of few layers using skip-connections or residual connections. This is what we see in the image above. In fact, if you look closely, we can directly learn an identity function by relying on skip connections only. This is the exact reason why skip connections are also called as identity shortcut connections too.

Let us consider a neural network block, whose input is x and we would like to learn the true distribution H(x). Let us denote the difference (or the residual) between this as

$$R(x) = Output - Input = H(x) - x$$

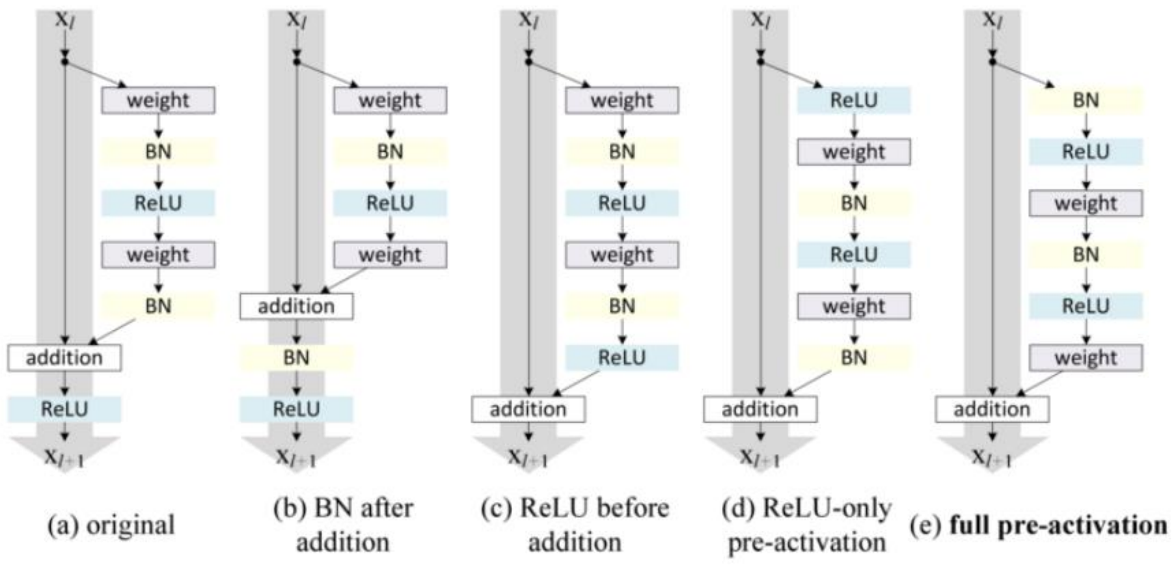
$$H(x) = R(x) + x$$

Our residual block is overall trying to learn the true output, H(x) and if you look closely at the image above, you will realize that since we have an identity connection coming due to x, the layers are actually trying to learn the residual, R(x). So to summarize, the layers in a traditional network are learning the true output (H(x)) whereas the layers in a residual network are learning the residual (R(x)). Hence, the name: Residual Block.

It has also been observed that it is easier to learn residual of output and input, rather than only the input. As an added advantage, our network can now learn identity function by simply setting residual as zero. To overcome the severity of the problem of vanishing gradient with increasing number of layers, we can clearly see that because of these skip connections, we can propagate larger gradients to initial layers and these layers could learn as fast as the final layers, giving us the ability to train deeper networks.

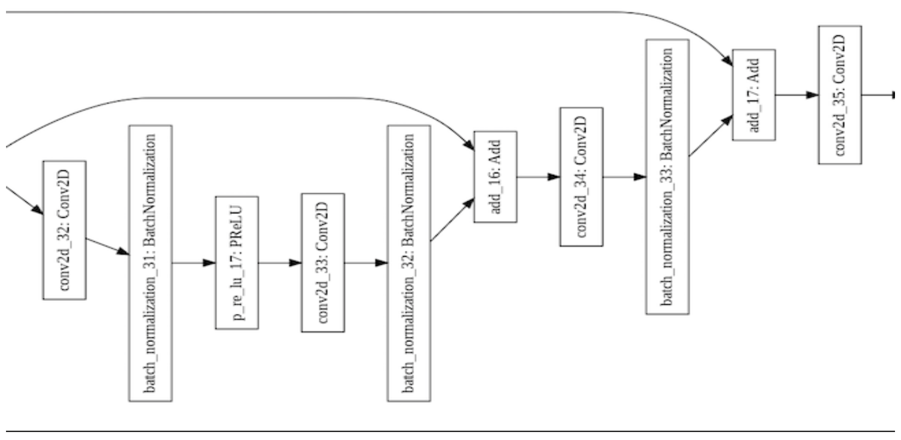
The image below shows how we can arrange the residual block and identity connections for the optimal gradient flow. It has been observed that pre-activations with batch normalizations give the best results in general (i.e. the right-most residual block in the image below gives most promising results).

While training ResNets, we either train the layers in residual blocks or skip the training for those layers using skip connections. So for different training data points, different parts of networks will be trained at different rates based on how the error flows backwards in the network. This can be thought of as training an ensemble of different models on the dataset and getting the best possible accuracy.



Types of Residual Block

Skipping training in some residual block layers can be looked from an optimistic point of view too. In general, we do not know the optimal number of layers (or residual blocks) required for a neural network which might depend on the complexity of the dataset. Instead of treating number of layers an important hyperparameter to tune, by adding skip connections to our network, we are allowing the network to skip training for the layers that are not useful and do not add value in overall accuracy. In a way, skip connections make our neural networks dynamic, so that it may optimally tune the number of layers during training.



conv2d_32: Conv2D
batch_normalization_31: BatchNormalization
p_re_lu_17: PReLU
conv2d_33: Conv2D
batch_normalization_32: BatchNormalization
add_16: Add
conv2d_34: Conv2D
batch_normalization_33: BatchNormalization
add_17: Add
conv2d_35: Conv2D

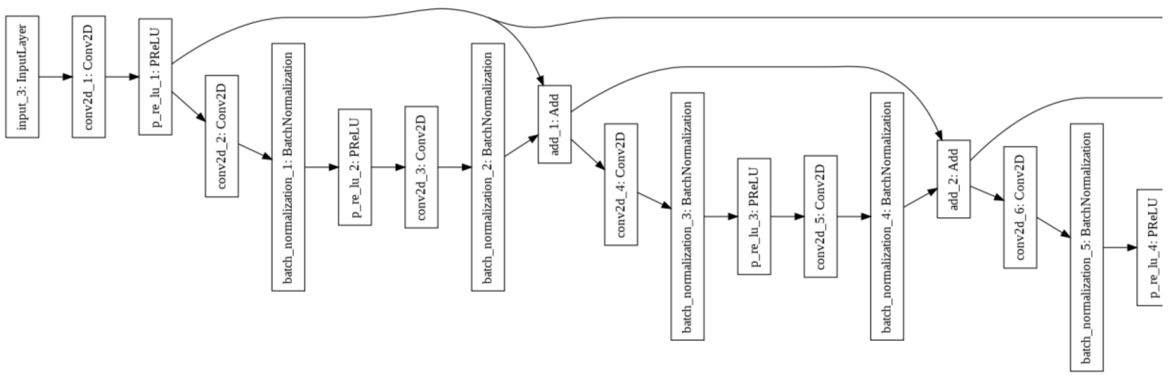

input_3: InputLayer
conv2d_1: Conv2D
p_re_lu_1: PReLU
conv2d_2: Conv2D
batch_normalization_1: BatchNormalization
p_re_lu_2: PReLU
conv2d_3: Conv2D
batch_normalization_2: BatchNormalization
add_1: Add
conv2d_4: Conv2D
batch_normalization_3: BatchNormalization
p_re_lu_3: PReLU
conv2d_5: Conv2D
batch_normalization_4: BatchNormalization
add_2: Add
conv2d_6: Conv2D
batch_normalization_5: BatchNormalization
p_re_lu_4: PReLU

Fig: Residual Blocks and skip connections in our implementation

2.1.2 Pixel Shuffler

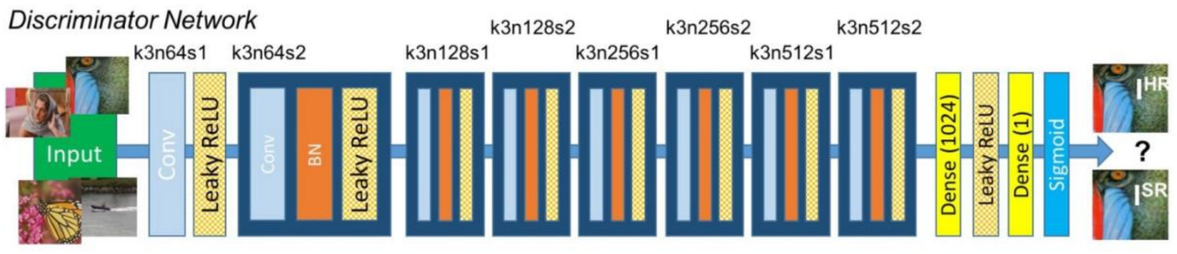
This is a feature map upscaling or upsampling. 2 sub-pixel CNN are used in generator. Pixel shuffler rearranges the elements of H x W x C. r^2 tensor to form rH x rW x C tensor.

2.1.3 Usampling

Instead of using Pixelshuffler, we can use upsampling or upscaling. There are various ways to do this. In the code, we used keras inbuilt functions.

```
model = UpSampling2D(size = 2)(model)
```

2.3 Discriminator



Discriminator is used to distinguish the HR images from SR images and back-propogate the GAN loss to train the discriminator and the generator. The discriminator takes the input from the generator. After a while, the generator learns to make better samples and begins to fool the discriminator. The discriminator also learns to detect these fake samples better, eventually, the generator becomes so good at creating samples that the Discriminator can't keep up and it leads to an equilibrium.

The architecture is same as above. We used LeakyRelu activation function (α=0.2) and avoid max-pooling throughout the network. The discriminator network is used to solve the maximization problem in equation 2. It consists of eight convolutional layers with an increasing number of 3x3 kernel filters, increasing by a factor of 2 from 64 to 512 kernels as in the VGG network. Strided convolutions are used to reduce the image resolution each time the number of features is doubled. The resulting 512 feature maps are followed by two dense layers and a final sigmoid activation function to obtain a probability for sample classification between SR and HR images.

Summary of our discriminator:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_4 (InputLayer)         (None, 224, 224, 3)       0         
_________________________________________________________________
conv2d_38 (Conv2D)           (None, 224, 224, 64)      1792      
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 224, 224, 64)      0         
_________________________________________________________________
conv2d_39 (Conv2D)           (None, 112, 112, 64)      36928     
_________________________________________________________________
batch_normalization_34 (Batc (None, 112, 112, 64)      256       
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 112, 112, 64)      0         
_________________________________________________________________
batch_normalization_35 (Batc (None, 112, 112, 64)      256       
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 112, 112, 64)      0         
_________________________________________________________________
conv2d_40 (Conv2D)           (None, 112, 112, 128)     73856     
_________________________________________________________________
batch_normalization_36 (Batc (None, 112, 112, 128)     512       
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 112, 112, 128)     0         
_________________________________________________________________
conv2d_41 (Conv2D)           (None, 56, 56, 128)       147584    
_________________________________________________________________
batch_normalization_37 (Batc (None, 56, 56, 128)       512       
_________________________________________________________________
leaky_re_lu_7 (LeakyReLU)    (None, 56, 56, 128)       0         
_________________________________________________________________
conv2d_42 (Conv2D)           (None, 56, 56, 256)       295168    
_________________________________________________________________
batch_normalization_38 (Batc (None, 56, 56, 256)       1024      
_________________________________________________________________
leaky_re_lu_8 (LeakyReLU)    (None, 56, 56, 256)       0         
_________________________________________________________________
conv2d_43 (Conv2D)           (None, 28, 28, 256)       590080    
_________________________________________________________________
batch_normalization_39 (Batc (None, 28, 28, 256)       1024      
_________________________________________________________________
leaky_re_lu_9 (LeakyReLU)    (None, 28, 28, 256)       0         
_________________________________________________________________
conv2d_44 (Conv2D)           (None, 28, 28, 512)       1180160   
_________________________________________________________________
batch_normalization_40 (Batc (None, 28, 28, 512)       2048      
_________________________________________________________________
leaky_re_lu_10 (LeakyReLU)   (None, 28, 28, 512)       0         
_________________________________________________________________
conv2d_45 (Conv2D)           (None, 14, 14, 512)       2359808   
_________________________________________________________________
batch_normalization_41 (Batc (None, 14, 14, 512)       2048      
_________________________________________________________________
leaky_re_lu_11 (LeakyReLU)   (None, 14, 14, 512)       0         
_________________________________________________________________
flatten_1 (Flatten)          (None, 100352)            0         
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              102761472 
_________________________________________________________________
```

```
leaky_re_lu_12 (LeakyReLU)    (None, 1024)              0
_________________________________________________________________
dense_2 (Dense)               (None, 1)                 1025
_________________________________________________________________
activation_2 (Activation)     (None, 1)                 0
=================================================================
Total params: 214,907,266
Trainable params: 107,451,713
Non-trainable params: 107,455,553
```

3. Loss

Loss is the most critical part for the performance of our generator network. We use perceptual loss.

3.1 Perceptual Loss Function

For calculating the Loss between generated image by the generator (SR) and the original High-resolution image (HR), the traditional loss used is pixel-wise loss function such as MSE, which compares the each pixel values between both the images and estimates the difference between them. However, this method fails to recover some of the lost high frequency details such as textures, nuance from the low resolution images.

Perceptual loss function compares the two images based on high-level representation from a pre trained convolution network. The pre-trained convolution network used in our case is VGG19 model. The definition of our perpetual loss function is

$$l^{SR} = \underbrace{\underbrace{l_X^{SR}}_{\text{content loss}} + \underbrace{10^{-3} l_{Gen}^{SR}}_{\text{adversarial loss}}}_{\text{perceptual loss (for VGG based content losses)}}$$

3.2 Content Loss

Based on the ideas of Gatys et al. Bruna et al. and Johnson et al. , the following loss functions built that is closer to perceptual similarity. This loss function was built based on the VGG 19 pre trained network. The following is the equation for content loss which consists of the euclidean distance between the feature map representation of the generated images $G_\theta(I^{HR})$ by the generator and the ground truth image I^{HR}.

$$l_{VGG/i.j}^{SR} = \frac{1}{W_{i,j}H_{i,j}} \sum_{x=1}^{W_{i,j}} \sum_{y=1}^{H_{i,j}} (\phi_{i,j}(I^{HR})_{x,y} - \phi_{i,j}(G_{\theta_G}(I^{LR}))_{x,y})^2$$

Here Wi,j and Hi,j describe the dimensions of the respective feature maps within the VGG network.

Following is the code where the VGG19 network is loaded with the weights parameter set to ‘imagenet’ and extracting the feature from the 9th layer of the VGG network. Making the contracted model.trainable= false is very important as we do not want to train the already trained VGG19 model. Compiling the VGG19 network with loss function as ‘mse’ gives the Euclidean distance between the feature map representation the go generated and the high-resolution image as stated above.

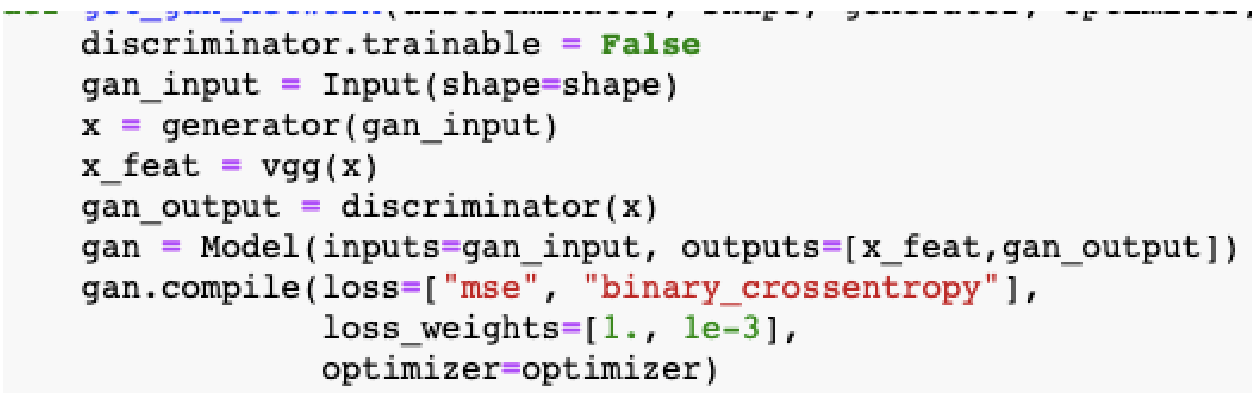
3.3 Adversarial Loss

In addition to the content we also add the adversarial loss that is used to fool the discriminator to favor our solutions resemble to the original high resolution images. The generative loss I^{SR} Gen is defined based on the probabilities of the discriminator $D_{\theta_D}\left(G_{\theta_G}(I^{LR})\right)$ over all training samples as:

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

Here, $D_{\theta_D}\left(G_{\theta_G}(I^{LR})\right)$ is the probability that the reconstructed image $G_{\theta_G}(I^{LR})$ is natural HR image.

As you can see in the below image, both the content loss from the VGG19 model and the adversarial loss (binary cross entropy) both are added to the GAN network where the generator network is trained



```
    discriminator.trainable = False
    gan_input = Input(shape=shape)
    x = generator(gan_input)
    x_feat = vgg(x)
    gan_output = discriminator(x)
    gan = Model(inputs=gan_input, outputs=[x_feat,gan_output])
    gan.compile(loss=["mse", "binary_crossentropy"],
                loss_weights=[1., 1e-3],
                optimizer=optimizer)
```

by freezing the discriminator network.

To train the discriminator, the loss function uses the typical GAN discriminator loss.

$$\min_{\theta_G} \max_{\theta_D} \mathbb{E}_{I^{HR}\sim p_{\text{train}}(I^{HR})}[\log D_{\theta_D}(I^{HR})] + \mathbb{E}_{I^{LR}\sim p_G(I^{LR})}[\log(1 - D_{\theta_D}(G_{\theta_G}(I^{LR}))]$$

4. Data Sets

4.1 DIV2K

- A novel Diverse 2K resolution RGB image dataset for benchmarking super-resolution. The dataset has large diversity of contents.
- train data: consists of 800 high definition 2K resolution images. This original image is resized to 224 x 224 x 3 using bicubic interpolation over 4x4 pixel neighborhood, and this image is taken as High Resolution (HR) image to reduce the computation time. This HR image is further downsampled by scale of 4 using same process to obtain a low resolution (LR) image of 56 x 56 x 3.

- val data: consists of 100 high definition 2K resolution images. Similar to training data, HR and LR images are obtained from the original.

4.2 FLICKR2

- Consists of 2650 high definition 2K resolution RGB images collected on the Flickr website.

5. RESULTS

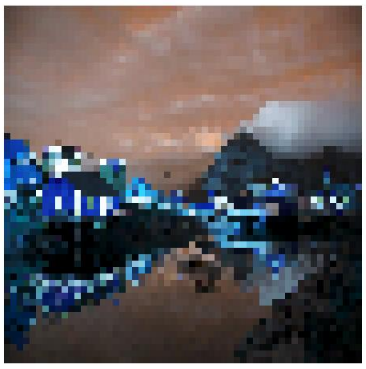
5.1 Qualitative performance measurement on DIV2K

After training the network for 12000 epochs with a batch size of 10 on google cloud with NVIDIA Tesla P100 with n1-highmem-8 (8 vCPUs, 52 GB memory), we investigated the capability of GAN to upscale a low resolution image to 4 times.

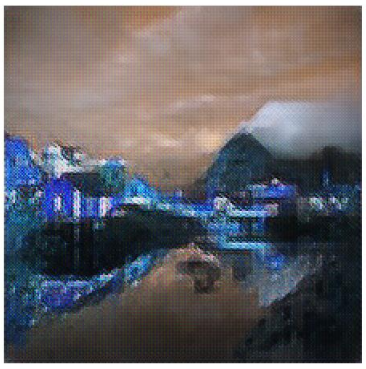
5.1.1 Using Pixel Shuffler

At 1000 epoch for training data:

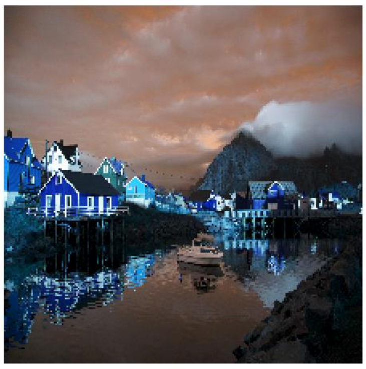
LR Image



SR Image

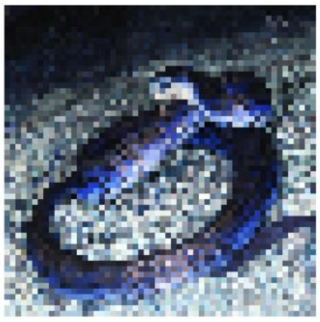


HR Image

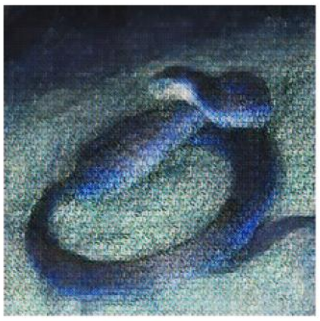


At 5200 epoch for training data:

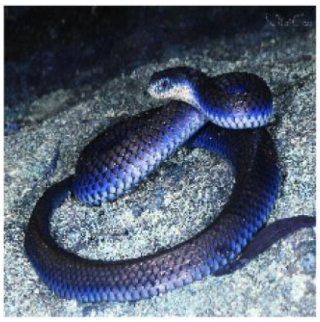
LR Image



SR Image

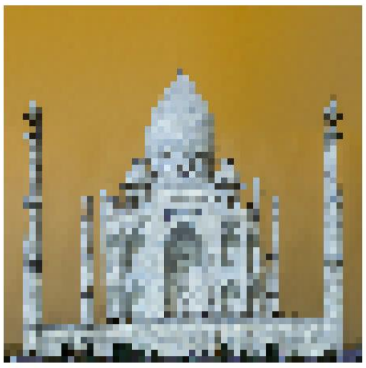
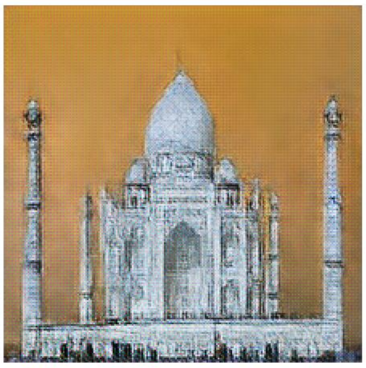
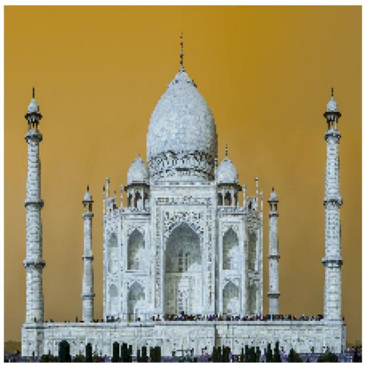


HR Image



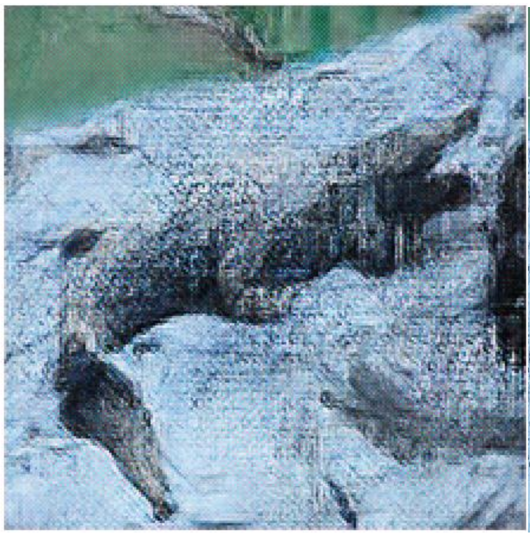
At 10,000 epoch for training data:

LR Image SR Image HR Image

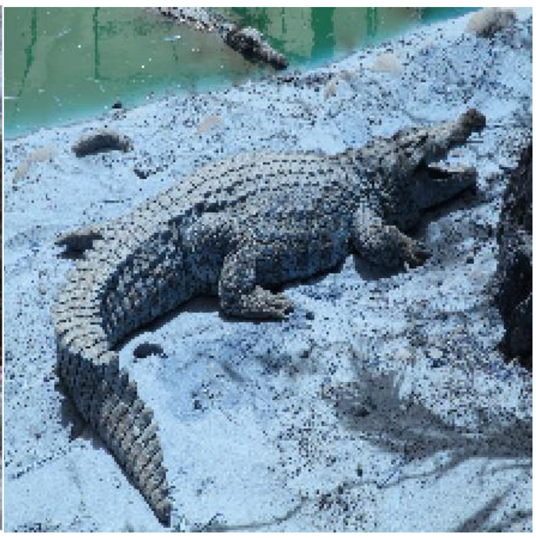




At 12000 epoch Generated images of Validation data:

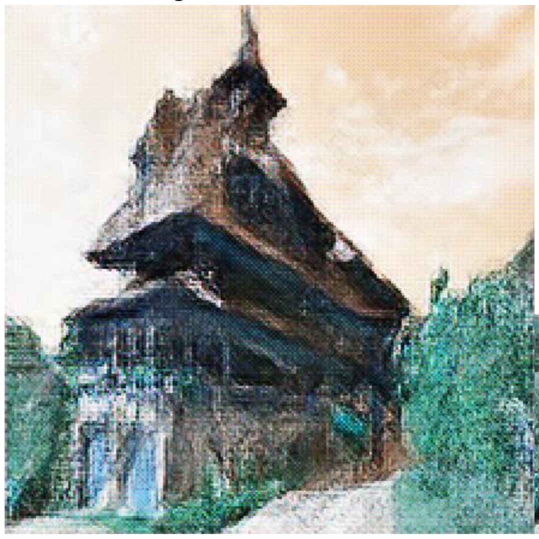
SR Image



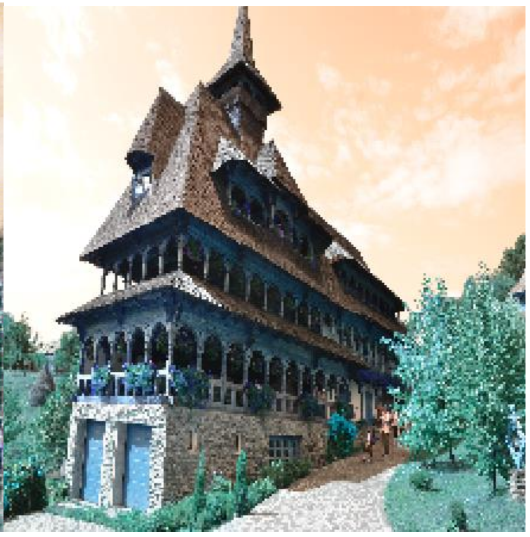
HR Image



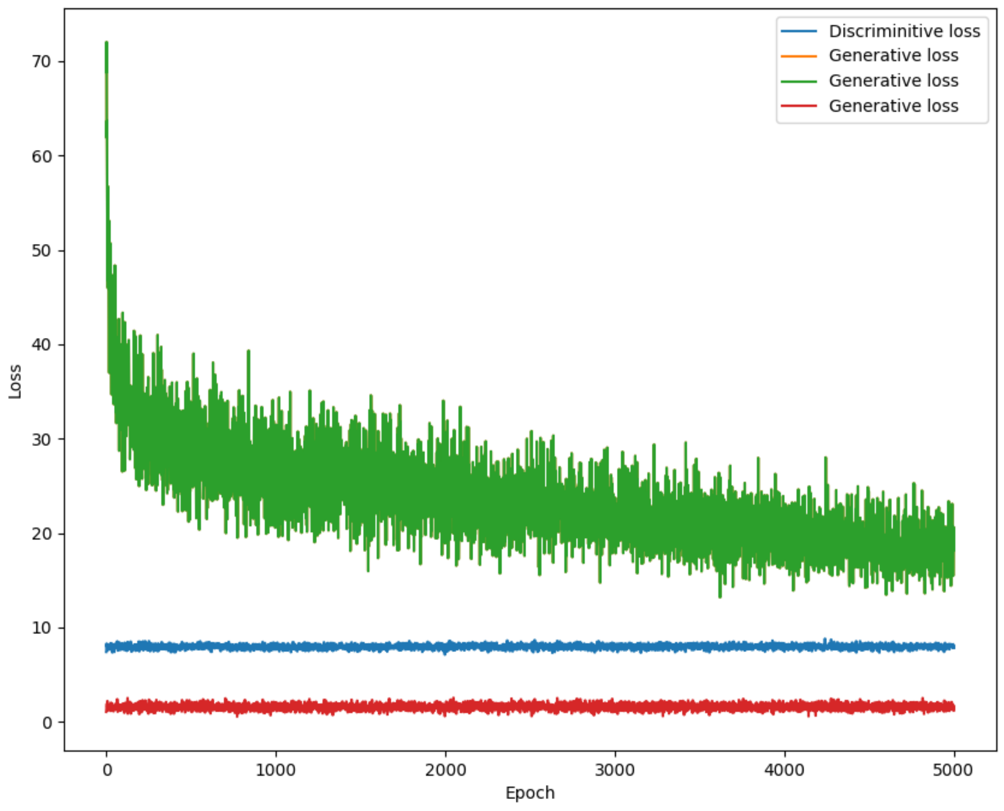
SR Image



HR Image



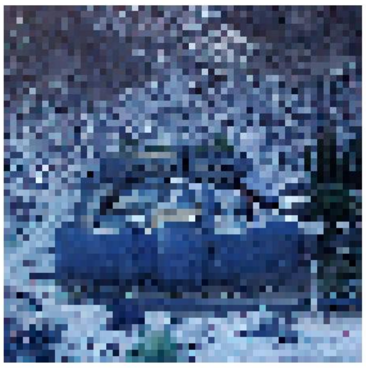
Loss Plot for training data



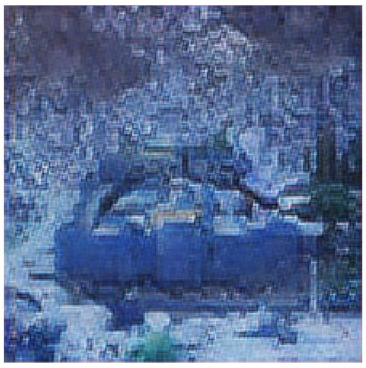
5.1.2 Using Upsampling

At 1000 epoch for training data:

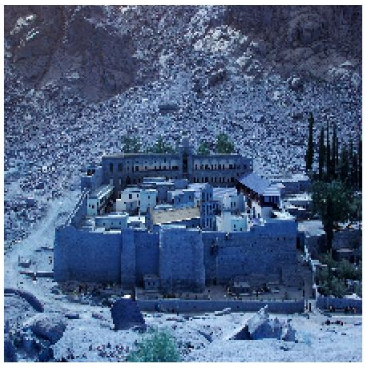
LR Image



SR Image

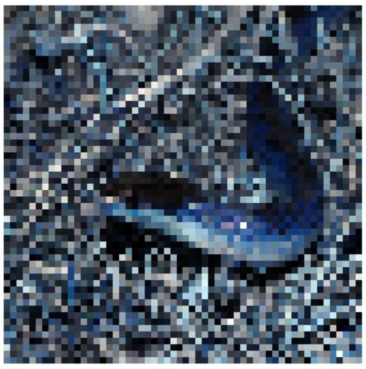


HR Image

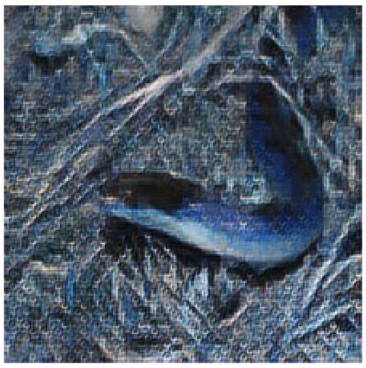


At 5000 epoch for training data:

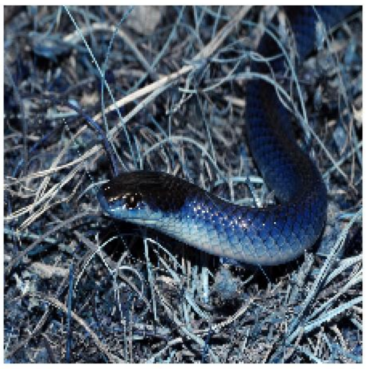
LR Image



SR Image

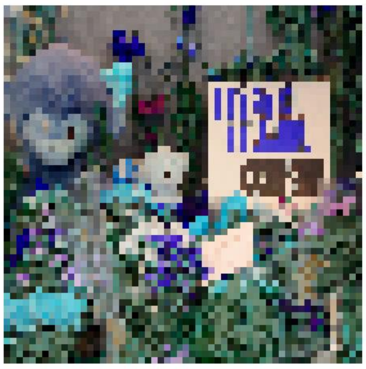


HR Image

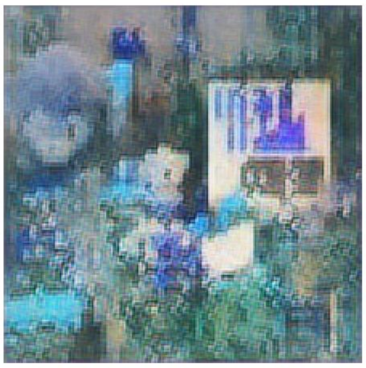


At 10,200 epoch for training data:

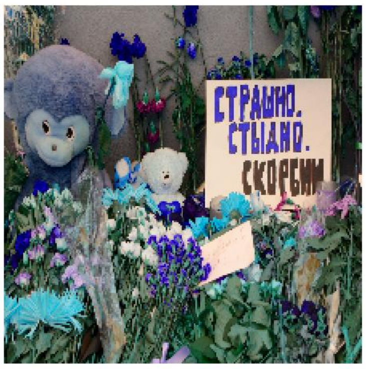
LR Image



SR Image

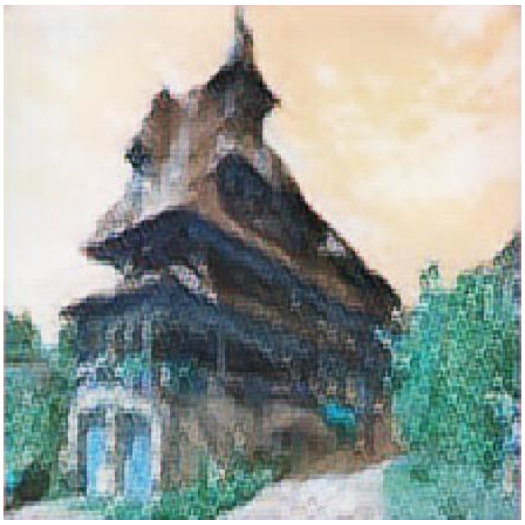


HR Image

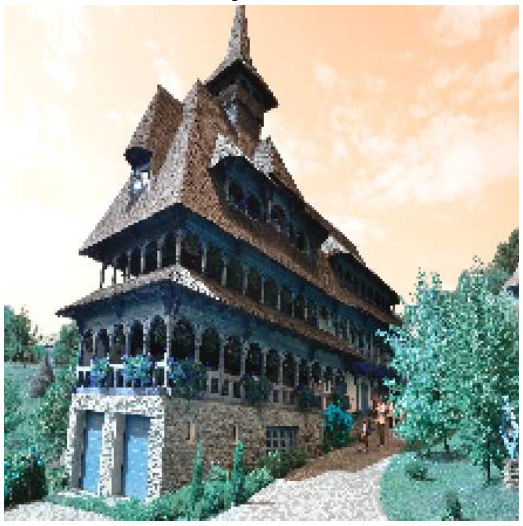


AT 12000 epoch Generated images of Validation data:

SR Image

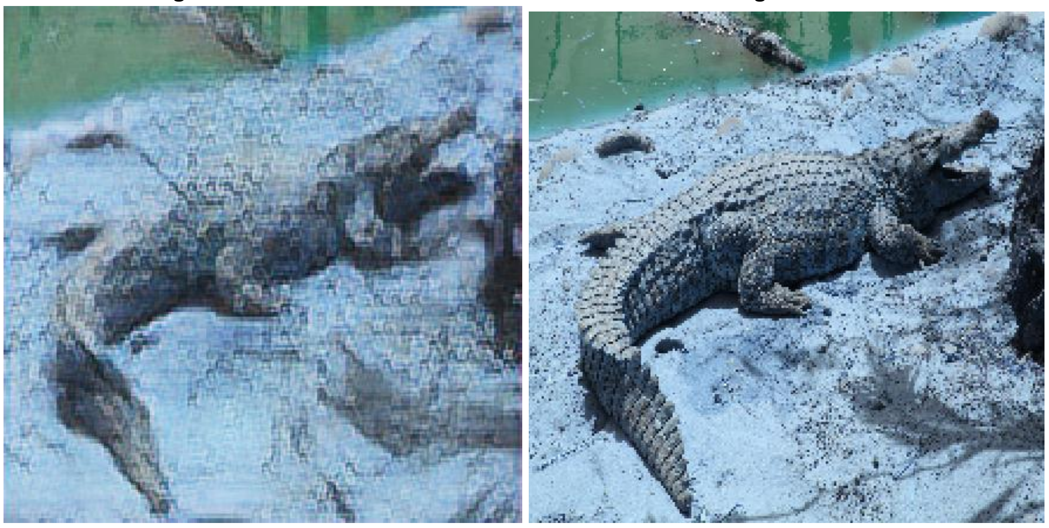


HR Image

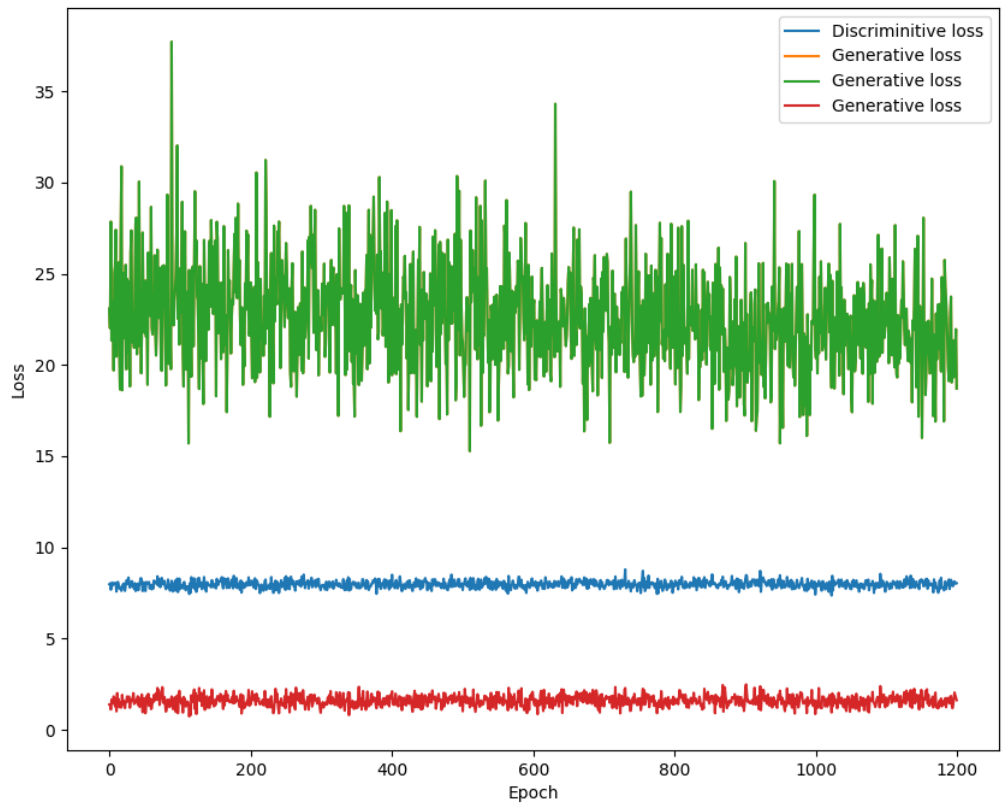


SR Image

HR Image



Loss Plot for training data:

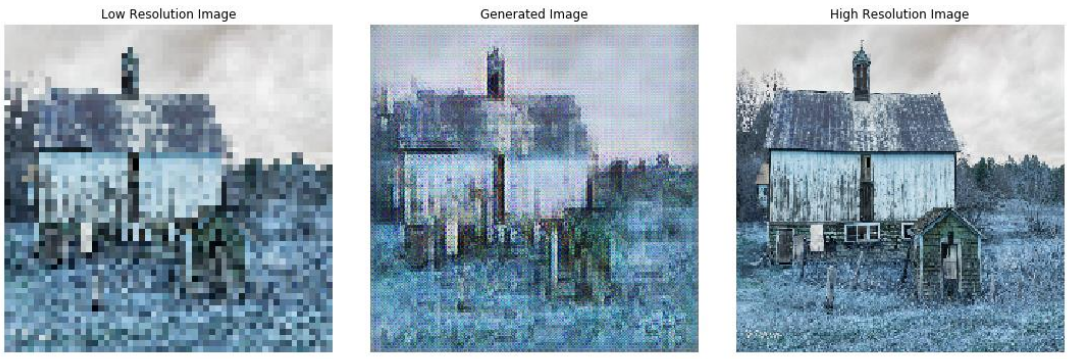


5.2 Qualitative performance measurement on FLICKR2K

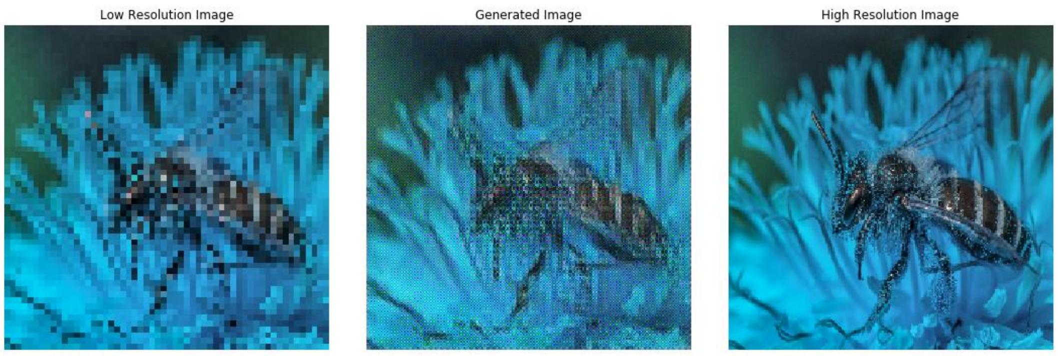
After training the network for 5000 epochs with a batch size of 10 on google cloud with NVIDIA Tesla P100 with n1-highmem-8 (8 vCPUs, 52 GB memory), we investigated the capability of GAN to upscale a low resolution image to 4 times.

5.2.1 Using Pixel Shuffler

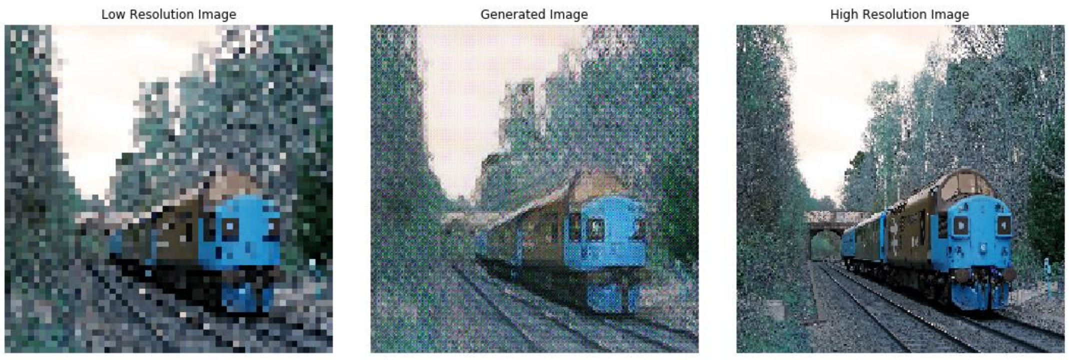
At 100 epoch for training data:



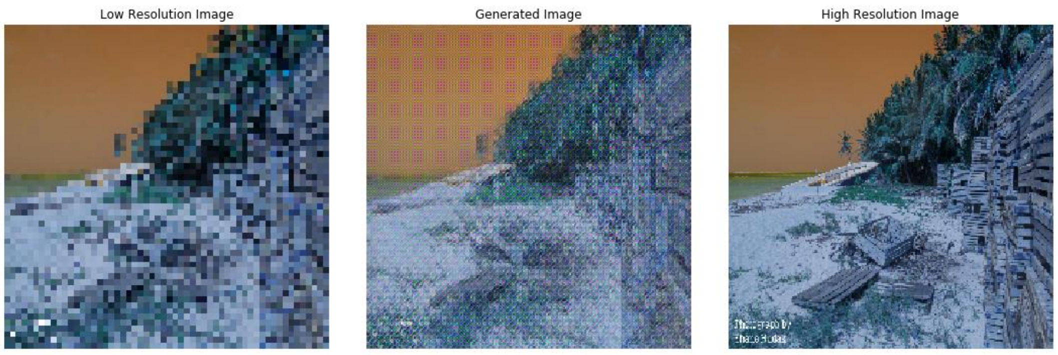
At 1000 epoch for training data:



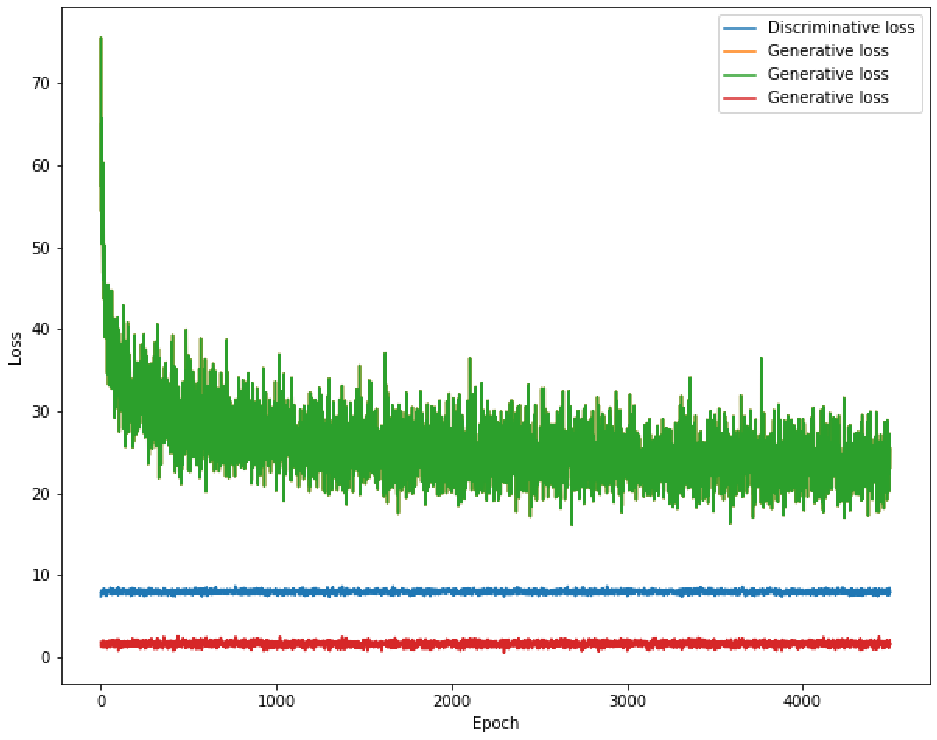
At 2500 epoch for training data:



At 4500 epoch of training data:



Loss Plot for training data:



5.3 Quantitative Comparison

We merged both DIV2K and FLICKR2K datasets to create a combined dataset and termed it as DF2K dataset.

Peak singal-to-noise ratio (PSNR) was used to do the quantitative comparison of performance of our SRGAN on 3 different datasets (DIV2K, FLICKR2K, DF2K). The results are tabulated below.

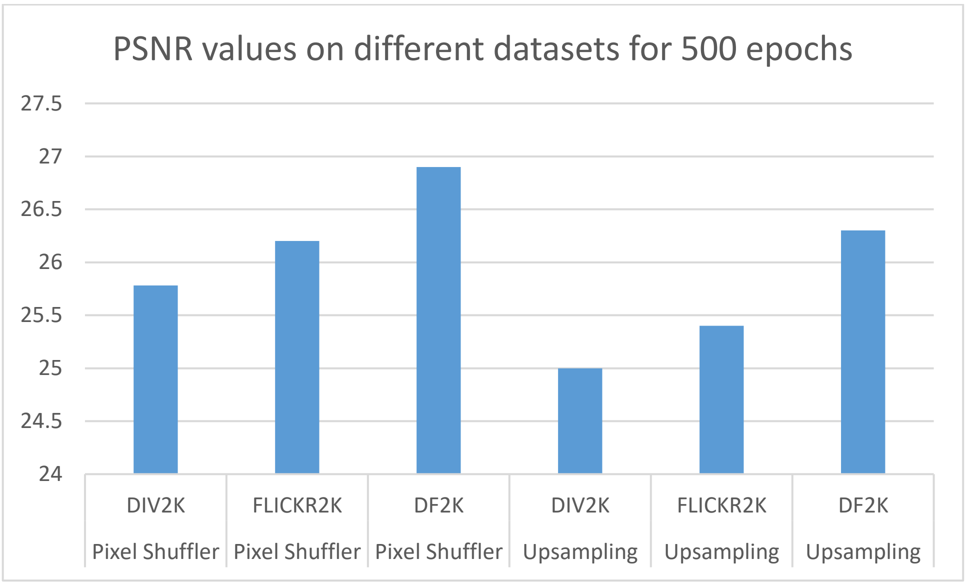
Formula used to calculate PSNR:

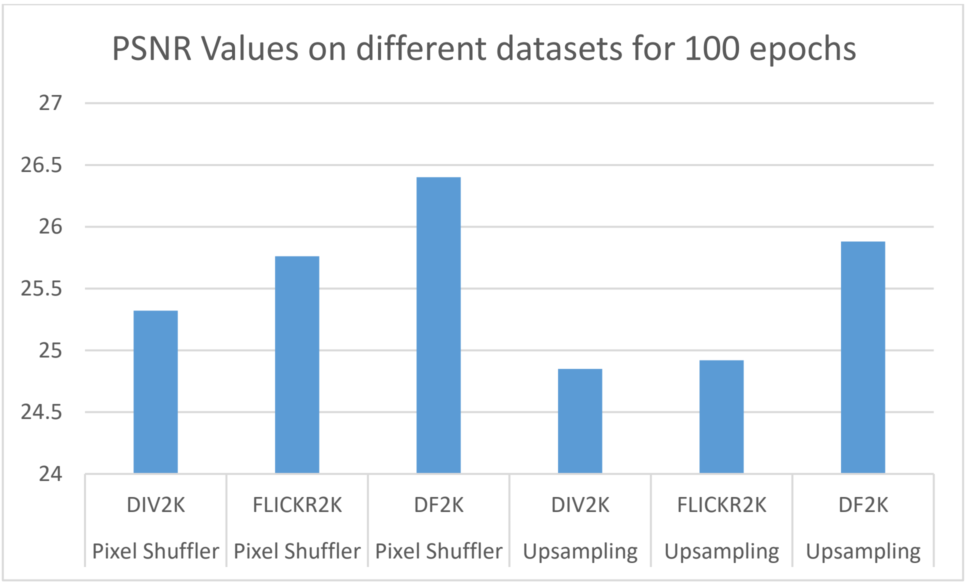
The PSNR (in dB) is defined as:

$$\begin{aligned} PSNR &= 10 \cdot \log_{10}\left(\frac{MAX_I^2}{MSE}\right) \\ &= 20 \cdot \log_{10}\left(\frac{MAX_I}{\sqrt{MSE}}\right) \\ &= 20 \cdot \log_{10}(MAX_I) - 10 \cdot \log_{10}(MSE) \end{aligned}$$

Method	Data	Epoch	PSNR
Pixel Shuffler	DIV2K	500	25.78 dB
Upsampling	DIV2K	500	25 dB
Pixel Shuffler	FLICKR2K	500	26.2 dB
Upsampling	FLICKR2K	500	25.4 dB
Pixel Shuffler	DF2K	500	26.9 dB
Upsampling	DF2K	500	26.3 dB

Method	Data	Epoch	PSNR
Pixel Shuffler	DIV2K	100	25.32 dB
Upsampling	DIV2K	100	24.85 dB
Pixel Shuffler	FLICKR2K	100	25.76 dB
Upsampling	FLICKR2K	100	24.92 dB
Pixel Shuffler	DF2K	100	26.4 dB
Upsampling	DF2K	100	25.88 dB





5.4 Quantitative Observations

- The combined dataset (DF2K) increases the PSNR performance by a large margin.
- The pixel shuffler has better PSNR values when compared to Upsampling method.
- Flickr2K performs better than DIV2K.
- More number of epochs result in better PSNR values.

5.5 Qualitative Observations

- As the number of epochs increase, the quality of the image increases
- On Training data: The images generated by Pixel Shuffler have better quality when compared images generated using an upsampler
- On Validation data: The images generated by Pixel Shuffler have better quality when compared images generated using an upsampler
- Based on quality of images, the FLICKR2K dataset generates better images compared to DIV2K images. The possible explanation is, FLICKR2K dataset has more images around 2650 images compared 800 images in DIV2K dataset
- It was observed that as the number of epochs increases with specified batch size the generator loss decreases which indicates that the generator is able to learn the distribution close to the original distribution of the dataset.
- As can be seen from the loss plot graph, though the generator loss is getting reduced there is not much difference in the discriminator loss. This is because the generator is still learning and hasn’t reached the level where it can fool the discriminator to identify the generated images as the actual one. Once the generator loss decrease to the level where it will generate the images similar to the actual image the discriminator loss will increase.