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

1) Introduction

In this report I present the implementation and the results of using GAN framework to solve the super-resolution task. The main code presented at https://github.com/PacktPublishing/Generative-Adversarial-Networks-Projects/tree/master/Chapter05 implement the SRGAN in keras : the scope of this project is to implement whole network in Tensorflow and look how it works with different datasets. In the following sections first, I introduce SRGAN main concepts and ideas, then I explain implementation code in details and dataset used. Finally in the last section I will show training details and results.

2) SRGAN

Super-resolution task is one of the main test fields of neural networks, or convolutional neural network in particular: given as input a low-resolution images, we want to obtain as output its high resolution corresponding version. Classical Convolutional Neural Network setup achieves good results in terms of PSNR (peak-signal-to-noise-ratio) and SSIM (structural similarity), but obtained images tend to be afflicted by a constant blur since due to mean square error loss function these networks tend to erase high frequency image components. The resulting images so tend to be less realistic respect to original ones.
SRGAN approach try to solve this problems: the focus of the problem now is moved from obtaining smallest mean squared pixel error to producing output images with generator network able to fooling discriminator network and so trying to outmatch blur problem.
Reference code is based on paper *"Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial, C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, W. Shi"*

2.1) Architecture

The adopted adversarial structure is resumed in the image below. Generator network is composed by several convolution layers: the core structure is composed by B residual blocks, the concatenations of B standard block composed by Convolution + BatchNormalization + Prelu + Convolution + BatchNormalization + Prelu + Element-wise sum with block input image. This structure is resumed from SR Residual Network.
Discriminator network accept in input High Resolution images and has the job to determine if an image is a real high-resolution version or a reconstructed one.

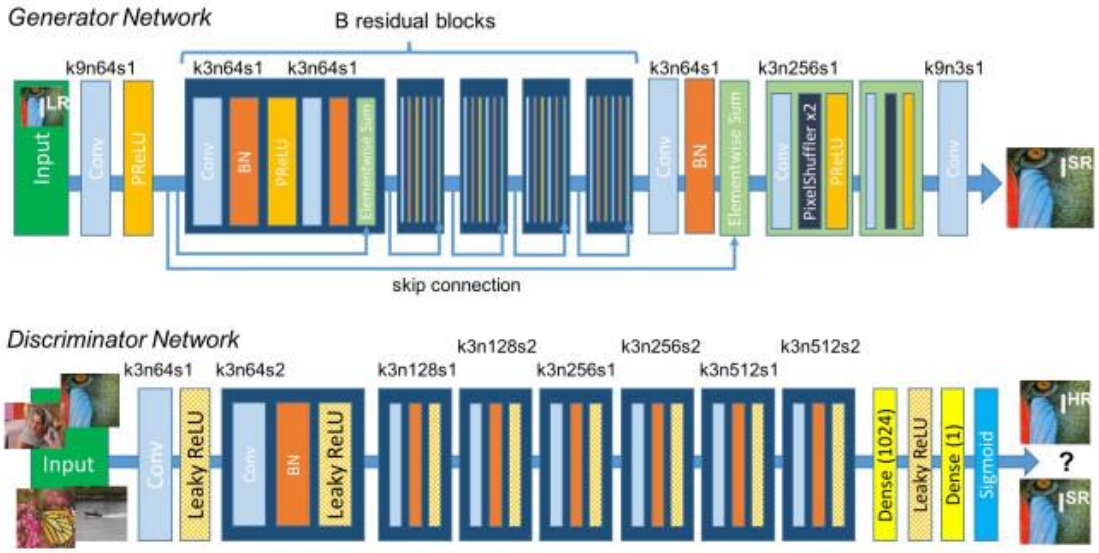

Fig.1 – Generator and discriminator network

2.2) Loss

The main particularity of this approach rely in loss function: more than optimize the network in order to decrease adversarial loss, in final loss is added an extra term called VGG loss. Loss function based on pixel mean square error obtain high PSNR index but also perceptually unsatisfying solutions with overly smooth textures due to cancelling high frequency content. VGG loss instead is closer to a perceptually similarity: obtained results have smaller PSNR and SSIM index but they are more realistic.

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

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

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

Fig.2 – (Top-Left) Vgg loss function. (Top-Right). Adversarial loss function (Down). Final loss combination of previous two

3)Implementation

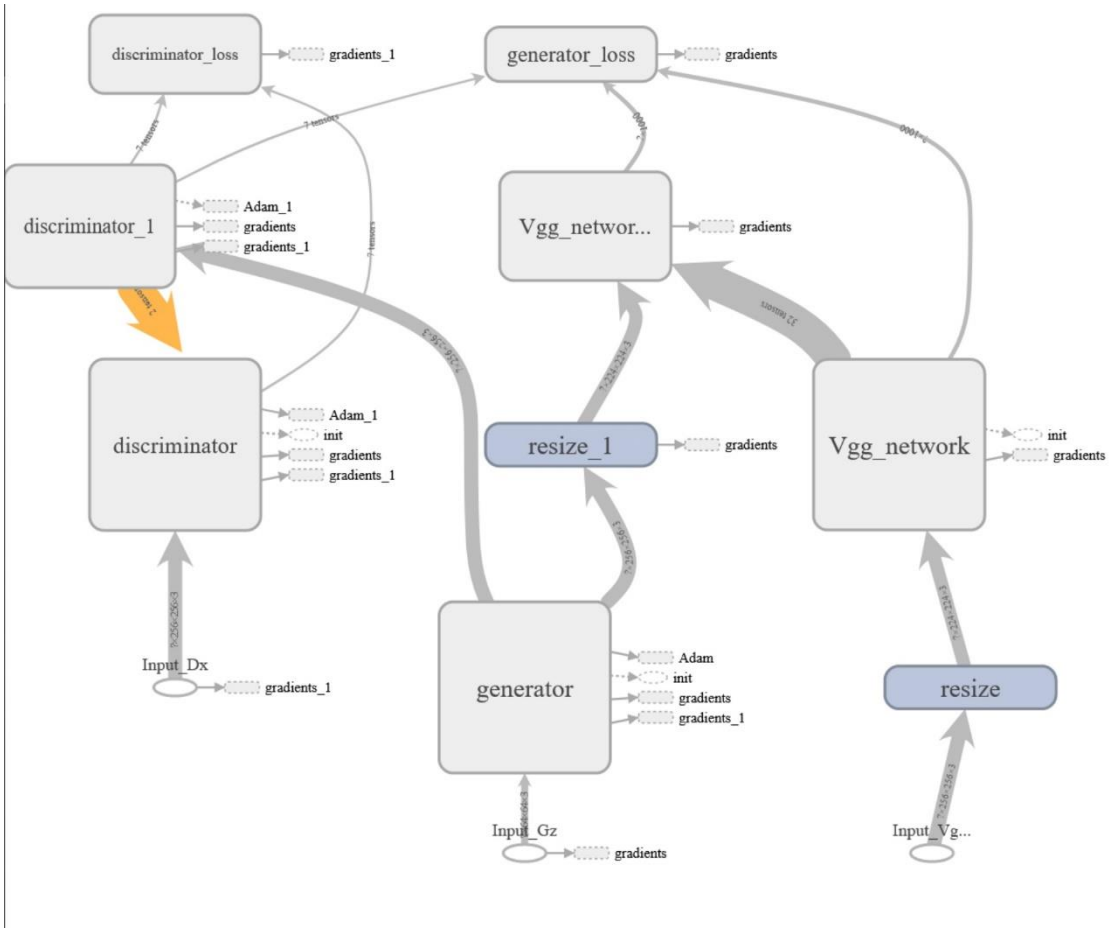

Fig.3 – Tensorflow computational graph. (2 tensor shared by discriminator are really 62, this is a tensorboard representation error)

In this section I will explain code structure. All project is implemented in TensorFlow.
The first thing to do is to build up principal networks: Vgg, generator and discriminator.

3.1) Vgg

Vgg network is a convolutional neural network created for classification and detection: it accepts in input image of size (224,224,3) and return as output a 1000-length vector of probabilities: each of them corresponding to a predefined class. The network is composed by 16 layers: 13 convolutional layer with 5 pooling stage and 3 final linear layers. The weights of the network have been downloaded and are used to complete Vgg setup: since we are not interested on tuning this network but we need only final vector for loss computation, Vgg weights are set to be not trainable. For each iteration it produces two output, one for the real high resolution image and one for the generated high resolution one and then this values are used to compute content loss. In the computational graph we can see the two Vgg network sharing weights.

Fig.4 -Vgg16 architecture

3.2) Discriminator & Generator

Discriminator network must decide if an image was produced by the generator (fake) or it's an original high-resolution image: it produces a final probability, 0 if image is a fake or 1 if it's original. So, it's called two times for each iteration: to enable to share the weight among these two discriminators we set reuse scope for second call inside discriminator main function. Generator network starting from a low-resolution image has the job to return its high-resolution version upscaled of a factor 4. Residual blocks part is composed by a repetition of 16 residual blocks. Generator use a reuse variable flag in order to be executed with same weights in evaluation produced images parts; moreover it use a flag to be used for batch normalization layers : True during training and False for evaluation.

```
self.Dx, logits_dx = self.Discriminator(self.input_dx)
self.Gz = self.Generator()
self.Dg, logits_dg = self.Discriminator(self.Gz, reuse = True)
```

To better see parameters reusing, next image show in more details this concept: in two consecutive calls discriminator network pass its 62 parameters tensors which are main variables of the discriminator network and variables used by batch_normalization layers.

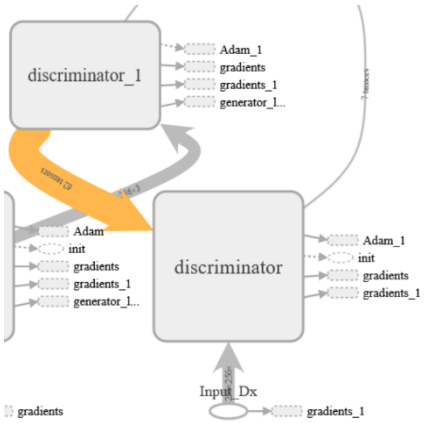

Fig.5 – Discriminator sharing variables detail

Then obtained values from both the networks are used in the loss computation section: for discriminator loss

```
with tf.name_scope('discriminator_loss'):

    d_loss_real = tf.reduce_mean(tf.losses.sigmoid_cross_entropy(tf.ones_like(logits_dx), logits_dx))
    d_loss_fake = tf.reduce_mean(tf.losses.sigmoid_cross_entropy(tf.zeros_like(logits_dg), logits_dg))

    self.d_loss = d_loss_real + d_loss_fake
```

And generator loss:

```
with tf.name_scope('generator_loss'):

    self.adv_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits = logits_dg, labels = tf.ones_like(logits_dg)))
    self.content_loss = tf.reduce_mean(tf.square(self.content_true - self.content_false))
    self.g_loss =  1e-3 * self.adv_loss +  self.content_loss
```

Since both discriminator and generator use batch normalization layers, I need to setup optimization phase. First, I divide trainable variables in discriminator's and generator's ones in order to after discriminator or generator loss update only respective variables. Then using control dependencies, I instantiate generator and discriminator optimizer using UPDATE_OPS function relative to batch normalization variables of each respective networks.

```
d_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope='discriminator')
g_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope='generator')
adam = tf.train.AdamOptimizer(beta1=0.97)

update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS, scope = 'generator')
with tf.control_dependencies(update_ops):
    self.train_optimizator_gen = adam.minimize(self.g_loss, var_list = g_vars )


update_ops_2 = tf.get_collection(tf.GraphKeys.UPDATE_OPS, scope = 'discriminator')
with tf.control_dependencies(update_ops_2):
    self.train_optimizator_dis = adam.minimize(self.d_loss, var_list = d_vars)
```

3.3) Dataset

Reference code use for training a dataset composed by persons' faces, *"img_align_celeba", available at* *https://drive.google.com/drive/folders/0B7EVK8r0v71pTUZsaXdaSnZBZzg*. For my experiments I used Places-205 database, from MIT Computer Science and Artificial Intelligence Laboratory, a collection of 41.000 images which belong to about 200 categories. The membership to difference categories however can be highlighted by Vgg components.
Moreover, I've used also Stanford Dog Dataset, http://vision.stanford.edu/aditya86/ImageNetDogs/, a collection of about 20k dog images divide in 200 races.

3.4) Training

Once build up whole computational graph I can execute training phase. High-resolution and low-resolution shape are respective (256,256,3) and (64,64,3), so our upscale factor for this super-resolution problem is equal to 4. Batch size is set to 1 and input images are normalized in order to stay from -1 and 1: naturally for the output we do the opposite process to obtain real image. Following paper advices, first we pre-train generator's net using a mse loss for 5k iterations in order to avoid undesired local minima during training : trainings launched without this prevention have produced mismatched and wrongs images. For each step discriminator and generator loss iterations are set to k=1. Finally, all complete adversarial train was executed for 20k iterations. Moreover, I've tried also to use for training experiment a setup which use a combination of mse and vgg loss together which produce good results.

4) Results

In the following plots is show generator and discriminator loss

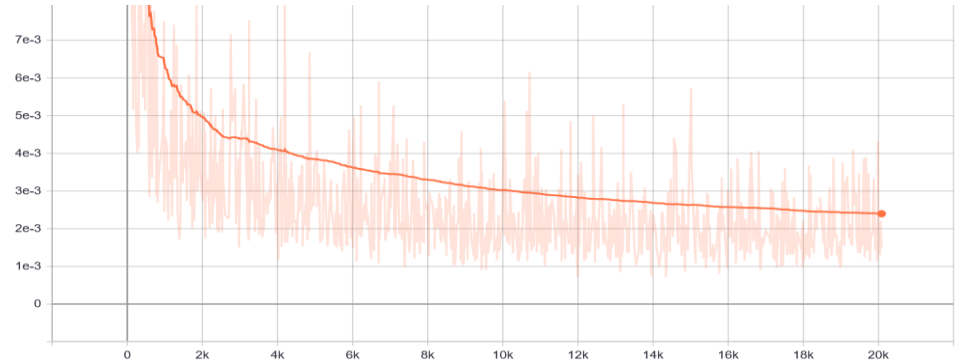

Fig.6 – Generator loss

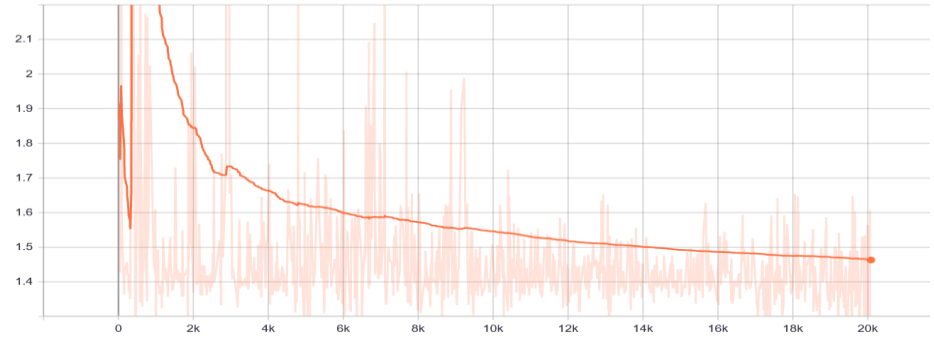

Fig.7 – Discriminator loss

We can see that probability that discriminator recognizes a correct image label stabilize around 50%, so it can't yet recognize if an image is fake or original.
Generator loss decrease over time as well as its component.

Fig.8 – Adversarial Loss

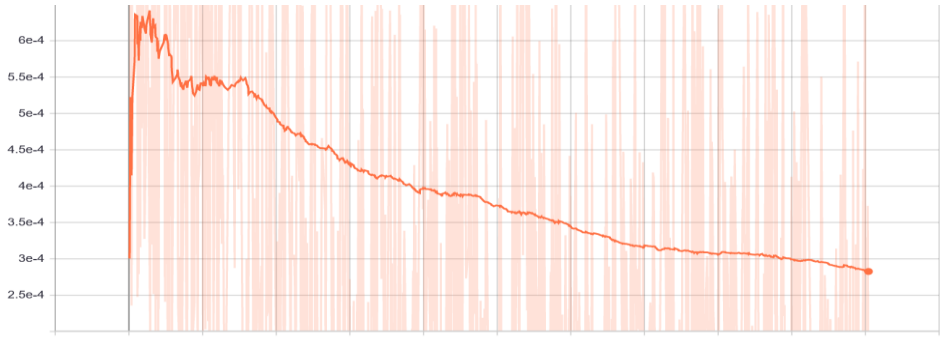

Fig.9 – Vgg content loss

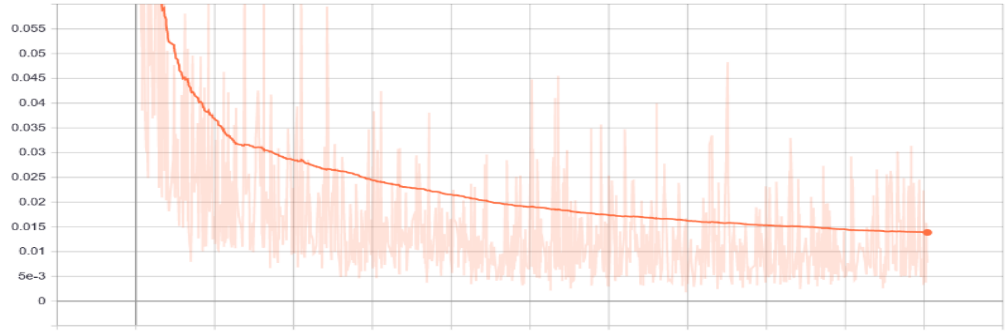

Fig.10 – Mean-Square error loss used for training together vgg content loss

Images generated are presented below:

Fig.11.1-4 – 11.1 (Up high resolution images, down reconstructed images) Resulting image after pre-train phase using only mean-square loss function. 11.2-4 Final obtained images starting from low resolution one with an upscaling factor equal to 4.

Mse-based images loose high-frequency components of original images but it's very quickly to reconstruct the image. Vgg resulting images demonstrate its best advantage: the network reproduces as its best surfaces and edges. Undesired results occur when it must reproduce images with small details like writings, as we can see in the picture below.

Fig. 12 – Real and reconstructed image from place dataset

Then I show also results obtained from dogs' dataset.

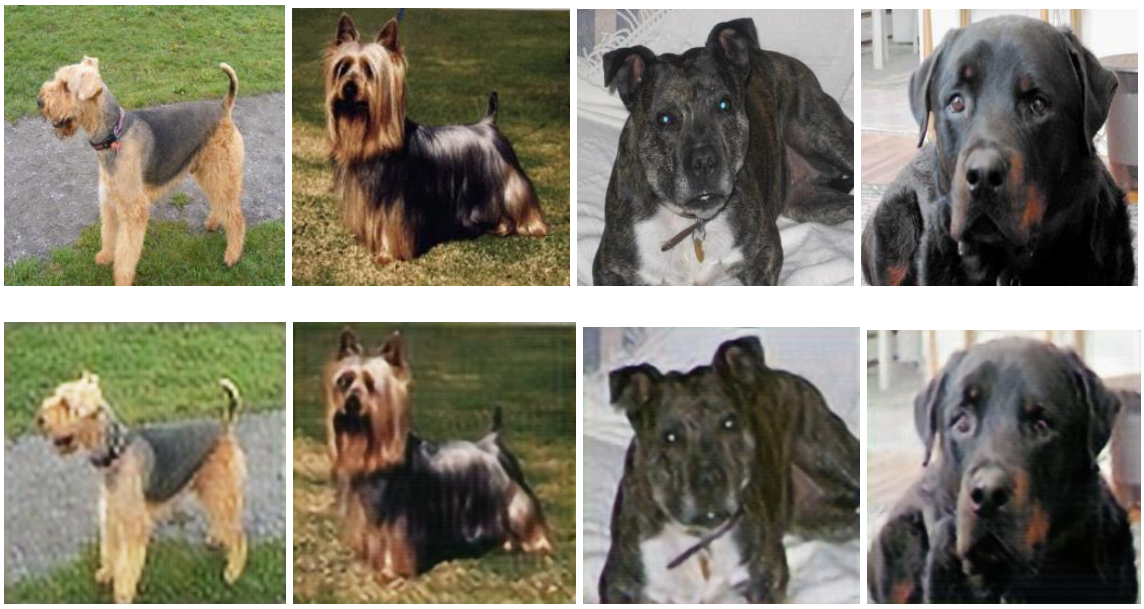

Fig. 13 – 1 (Up high resolution images, down reconstructed images) Reconstructed images from dog dataset

5) References

[1] *"Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial, C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, W. Shi", Twitter*

6) Appendix

Other reconstructed images obtained from the network

Original SRGAN

Fig. 14 – Reconstructed images from place dataset

Original SRGAN

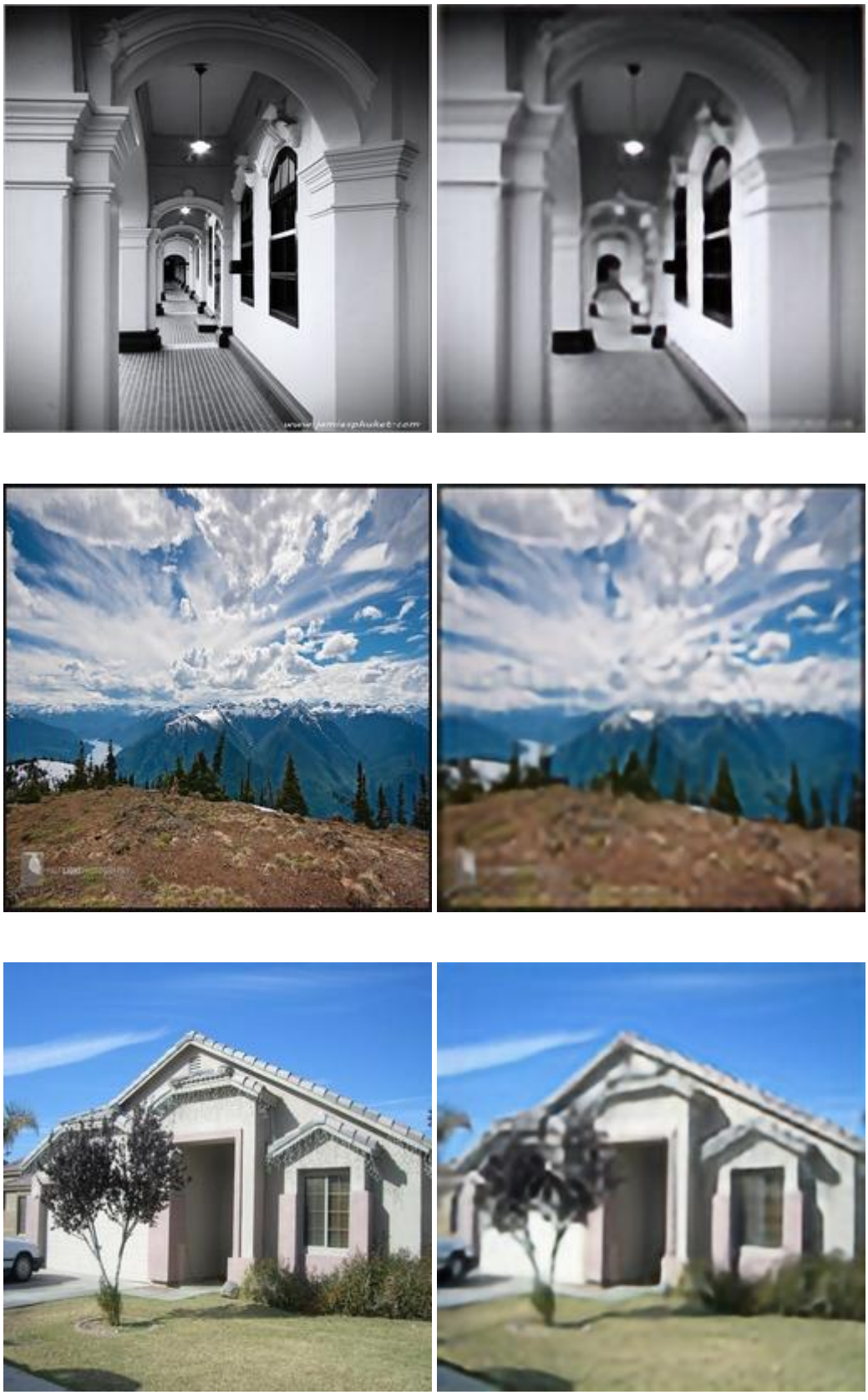

Fig. 15 – Reconstructed images from place dataset

Original SRGAN

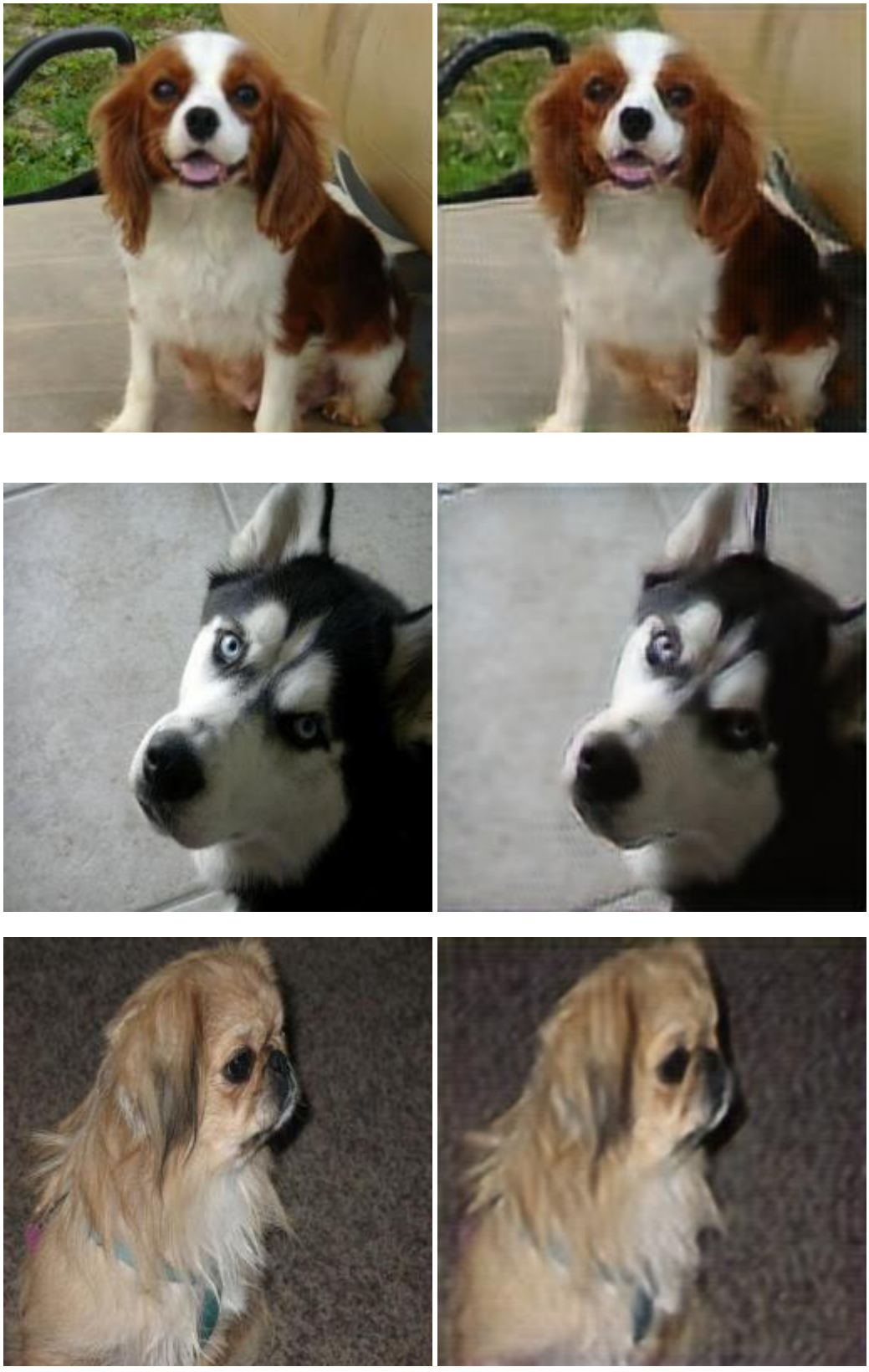

Fig. 16 – Reconstructed images from dog dataset