Generative Modelling in Image Processing Survey, Critique, and Implementation

Anonymous CVPR 2021 submission

Paper ID ****

Abstract

While deep discriminative models have contributed greatly in computer vision task such as image classification, deep generative models such as Variational Autoencoder and General Adversarial Networks have revolutionized the field of image processing by accomplishing challenging tasks such as super-resolution, image translation, and 3D reconstruction. The paper covers important details of 3 recent published work - TecoGAN, Pix2Vox, and SinGAN - that tackles image processing task as such using deep generative models, as well as implementation of SinGAN. The survey of these studies may suggests that field of image processing will likely to expand further and there are more yet-to-be-seen capabilities of the deep generative models.

1. Introduction

Over the past few years, deep generative models such as Variational Autoencoder (VAE), Generative Adversarial Networks (GAN) have become a popular method for image manipulation. GAN has especially gained popularity for its capability to dynamically change the objectives during training and its outstanding performance.

This paper will cover the works of 3 recently published papers that used deep generative models to tackle image processing tasks: TecoGAN[6], Pix2Vox[2], SinGAN[12]. TecoGAN attemps to reduce temporal artifacts in video generation, Pix2Vox addresses and tackles the issue in multi-view 3D-volume-reconstruction networks, and SinGAN attempts to train GAN with a single image.

Implementation of SinGAN is also included at the end of this paper. More focus will be put on the studies of SinGAN as implementation techniques and results will be discussed.

2. TecoGAN: Temporal Coherence GAN

2.1. Introduction to TecoGAN

Many works have been done on producing high-quality results for super-resolution and image translation of static images. However, performing such tasks on video frames, that have strong coherence in time, becomes more challenging since the coherence are often not taken into account during training.

Temporal Coherence Generative Adversarial Networks (TecoGAN) [6] is an unsupervised approach to generate videos while maintaining natural motion by learning the temporal coherence of the video frames. The authors of the paper suggested 2 applications of TecoGAN: Video super resolution and unpaired video translation, which details will be discussed in the later sections.

2.2. Learning Temporal Coherence

To enforce the learning of the temporal coherence, the paper proposed the use of spatio-temporal discriminator in addition to recurrent frame generator and a motion-estimating network that approximates the optical flow in the high-resolution image [7], which have been shown to be successful in video super-resolution task.

In addition to the conventional adversarial learning, the framework has 4 learning objectives.

1. $L_{content}$ (Content Loss): Loss in reconstruction of individual frames.

2. L_{ϕ} (Perceptual Loss): Temporal-perceptual loss of the generated frames. The loss is calculated by comparing the differences in the feature maps of triplets - 3 adjacent frames - from the generated frames and triplets from target frames. The feature maps are obtained from the pre-trained VGG-19 [5].

3. L_{warp} (Warp Loss): Loss in quality of motion estimated by motion-estimation network.

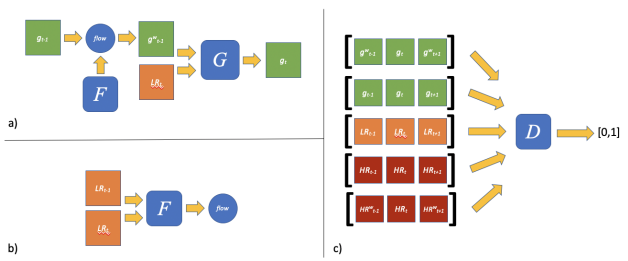

Figure 1. Overview of Super-Resolution TecoGAN. a) Generator, G takes low-resolution frame LR_t and warped, previously generated frame g^w_{t-1} to estimate the high-resolution of the current frame g^t. b) Optical flow that is used for warping images are estimated by motion-estimation network, F. c) Spatio-temoporal discriminator D takes set of triplets - 3 adjacent frames - in contrast to only 1 frame as the conventional discriminator does.

4. L_{PP} (Ping Pong Loss): Loss proposed by the authors of TecoGAN. It monitors the long-term temporal consistency in the generated videos. It evaluates the ability of the generator to produce forward and backward motion. Details are discussed in the later section.

2.3. Video Super Resolution

In super-resolution task, the networks are trained to generate a higher-resolution images given low-resolution ones. The high-resolution videos are generated using recurrent network. As shown in Figure 1a) The generator takes two inputs: low-resolution frame at time t and warped frame at time $t-1$ that was previously generated by the network itself.

Spatio-temporal discriminator, shown in Figure 1c) takes triplets in contrary to a conventional discriminator that takes only one frame at a time. There are two types of triplets; One is simply 3 adjacent frames. The other is the corresponding warped triplets. Warped triplets consists middle frame that is not warped and the other two warped to the middle frame. Warping requires optical flow, which is approximated by a motion-estimation network, shown in Figure 1b). Motion-estimation network takes encoder-decoder type structure.

2.4. Unpaired Video Translation

Image translation is a task to transfer a style of one image to another. CycleGAN [4] is able to accomplish this on static images with high spatio-details. TecoGAN tackles the same task, but with in consideration of termporal coherence while maintaing the spatio-details that CycleGAN is able to produce.

Authors take a similar approach to CycleGAN, which is training 2 sets of generator/discriminator: One set for translating A to B, the other for translating B to A.

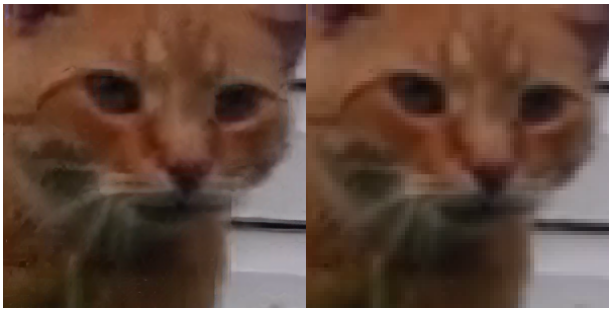

Figure 2. **Super-Resolution with TecoGAN**. Left: Frame generated by TecoGAN, Right: Original Frame

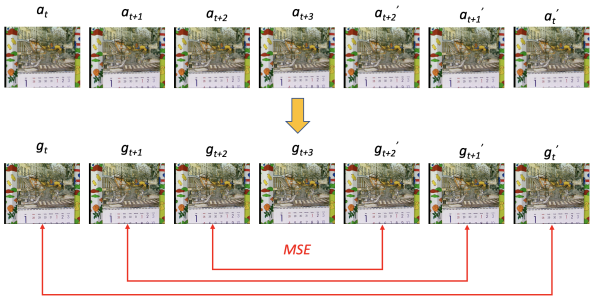

Figure 3. **Ping Pong Loss** is measures the generator's ability to produce forward and backward motion. Input frames are mirrored (a_i and $a_i^{'}$ are identical) during pre-processing. Ping Pong Loss is simply the MSELoss between g_i and $g_i^{'}$.

Similar to Video Super-Resolution task, discriminator takes non-warped and warped triplets. However, according to the claims made by the authors, it is important for the generators to learn reasonable spatio features first before learning the temporal features. To compensate, the discriminator can also take static triplets (same images concatenated 3 times). At the intial stage of training, discriminator takes static triplets instead of the 3 adjacent frames. Discriminator input is gradually transitioned to non-warped or warped non-static triplets.

2.5. Ping Pong Loss

Generator is able to learn the short-term temporal coherence from adversarial learning against the spatio-temporal discriminator. Ping Pong loss is an extension to the framework that adds robustness by self-supervising the long-term temporal consistency.

As shown in Figure 3, input frames are preprocessed to include backward motion. Ping Pong Loss is simply the mean-square-error between the corresponding pairs. The paper has claimed that incorporating Ping Pong Loss has reduced the temporal artifacts in the generated videos.

2.6. Performance

For video-super-resolution task, TecoGAN succcessfully outperforms previous works such as ENet[8], FRVSR[7], DUF[14]. While the results of all of these frameworks are comparable, TecoGAN was consistently able to get lowest in Learned Perceptual Image Patch Similarity (LPIPS) [11] and the highest in user study among them.

TecoGAN was also successful in video-translation task and was able to reduce high frequency transition in the frames while maintaining the spatio-details that are comparable to the results of CycleGAN.

Authors also presented 2 new temporal metrics: tOF and tLP, shown in Equation 1.

$$\begin{aligned} tOF &= ||OF(b_{t-1}, b_t) - OF(g_{t-1}, g_t)||_1 \\ tLP &= ||LP(b_{t-1}, b_t) - LP(g_{t-1}, g_t)||_1 \end{aligned} \tag{1}$$

b refers the target frame, OF refers to optical flow, and LP refers to LPIPS metric. The motivation is based on the claim by the authors that pixel-wise change in perceptual change in time are crucial for evaluating the temporal coherence. This metric can be used in both super-resolution and video-translation tasks. Not surprisingly, TecoGAN outperformed all other frameworks that were brought up in the paper, in both tasks.

2.7. Critique

TecoGAN is able to perform video generation without creating noticeable temporal artifacts. Extensive engineering was involved in making the framework successful. It will also be interesting to see how the new proposed temporal metrics will come into play in future video processing tasks.

The authors also deserve many credits for extensive performance analysis with many of the previous works in super-resolution and image translation. The objective is clear throughout the paper and it is clear that the work was successful. However, since the training involves many objectives, it is possible that the result of the training may only satisfy few of criteria, so the training is susceptible to failure as it was stated by the authors. It would be interesting to see if there are more efficient way of constructing objective to learn both spatio-features and temporal coherence.

3. Pix2Vox: Context-Aware 3D Reconstruction

3.1. Introduction to Pix2Vox

Pix2Vox is a model that takes multiple 2D images and estimate the 3D volume in voxel representation. This can be useful in application such as Simultaneous Localization and Mapping (SLAM) since 3D reconstruction from 2D images can be very challenging, not to mention, especially when there are not a significant overlap in images.

Previous works in voxel reconstruction from 2D images involved Recurrent Neural Networks (RNN). The authors of Pix2Vox [2] replaces the RNN with context-aware fusion, motivated by the claim that RNN puts limitation in the framework due to long-term memory loss, and long inference time.

3.2. Architecture

Structure of Pix2Vox includes encoder, decoder, context-aware fusion, and refiner.

Pre-trained VGG16 [5] were used as encoder. For each 2D image, the encoder produces feature maps, which are processed by the decoder and produce a coarse 3D voxels. Multiple coarse 3D volumes are then fused together by context-aware fusion. In context-aware fusion, scores are assigned to each volume by convolutional layers that takes last two layers of the decoder as inputs. The scores are then normalized and the volumes are fused together by weighted sum. Finally, the fused volume is passed to UNet[10]-like refiner to correct the model further.

3.3. Performance

Pix2Vox was tested on ShapeNet by using Intersection-over-Union as a metric. Pix2Vox was able to outperform other networks for the most part.But most importantly, Pix2Vox magnificently improves in inference time (Pix2Vox-F is smaller version of Pix2Vox that does not include Refiner).

	Inference Time		
Model	3D-R2N2 [1]	Pix2Vox-F	Pix2Vox-A
1-view (ms)	73.35	9.25	9.90
2-view (ms)	108.11	12.05	13.69
3-view (ms)	112.36	23.26	26.31
4-view (ms)	117.64	52.63	55.56

3.4. Critique

The authors were able to both improve in accuracy, model size, and inference time. The findings in this paper really shows that RNN was inefficient for this task.

Pix2Vox was trained for low-resolution voxels, so it is hard to say that it is practical, yet. However, it may have build the foundation for future work in geometric deep learning.

4. SinGAN: Training GAN with a Single Image (with Implementation Details)

4.1. Introduction to SinGAN

Many of today's GANs require large datasets, which is typical for deep neural networks. SinGAN [12] is a proposed method to train a set of GAN with a single image.

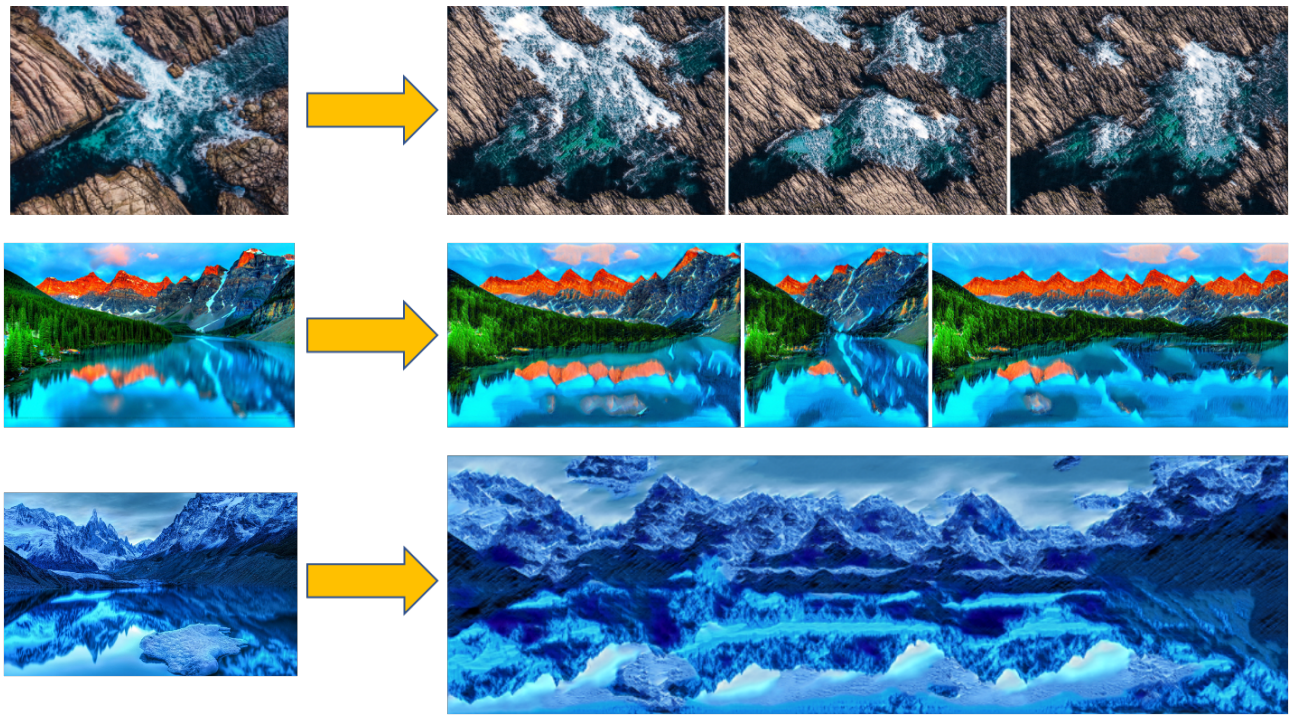

Figure 4. **Random Samples from SinGAN**.Images on the left are the images used for training. The images on the right shows the random samples from the trained model. Since generators use only convolutional layers, they can take variable-sized inputs and hence, produce variable-sized outputs.

When SinGAN is trained successfully, it is able to produce realistic and high-resolution variations of the image as shown in Figure 4. It can also be used to perform tasks such as super-resolution, texture transfer, and animations.

4.2. Multi-Scale Architecture

The successful training with a single image is accomplished by training a set of GANs in multiple scales as shown in Figure 5. Each GAN is trained with a different scale of target image. The first GAN is trained with the lowest scale, downsampled by r^{n-1} where r is some scale that is less than 1 and n is the number of different scale levels. After training the first GAN, the second GAN is trained with the target image downsampled by r^{n-2}. The process is repeated until successfully training the nth GAN with the original resolution.

The generator takes noise and interpolated, generated random sample from the generator from the previous scale. The noise and the sample are added prior to feed-forwarding to the network and the output is added with the sample, again. When training for the smallest scale, there would be no sample.

The discriminator's job is not to detect whether image is a generated sample but rather act as a critic that scores the realness of the image. The conventional discriminator is designed to evaluate the entire image with only 1 neuron at the output that describes the probability of the image being real/fake. Instead, the network validates the realness of the generated image based on the details that exists in the image. This is done by using only convolutional layers. Each neuron at the output layer will represent the score of realness of a patch. The size of the patch becomes smaller as the scale goes up, focusing more on the details on the image. The idea is that each generator learns to produce patches that are similar to the patches of the target image and adds details that the previous generators were not able to generate. In a way, patches may also be consider as a dataset.

4.3. Wasserstein GAN and Gradient Penalty

In order to make the training stable, Wasserstein GAN (WGAN) [9] with Gradient Penalty [3] was used. The idea of WGAN is to replace the discriminator, that determines the probability a given image is real/fake, with a critic that scores the realness/fakeness of an image. The critic follows 1-Lipschitz constraint in order to make the gradient smoother. This leads to stable learning of the generator.

$$|f(x) - f(y)| \leq C|x - y| \tag{2}$$

The property shown in Equation 2 is referred to as C-Lipschitz continuity, where C is a scalar constant. This constraint is enforced on the critic by performing parameter

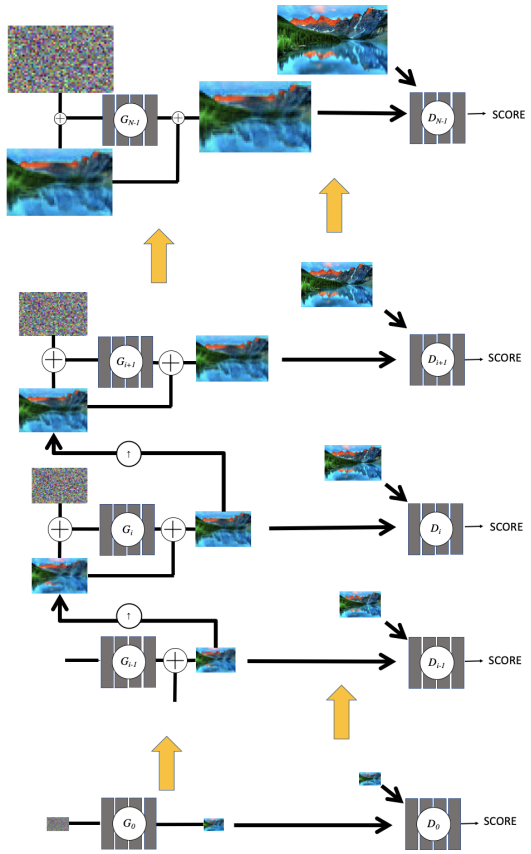

Figure 5. **Multi-Scale Architecture for SinGAN**. Multiple levels of scales of images are created, and for each level, a generator and a discriminator are trained. Training starts from the lowest scale, and every generator after that depends on all the previous generators that were trained in the lower scale.

clipping. After the parameter update, the critic's parameters are clipped in a predefined bounds. The following is a pseudo-code for WGAN training iterations.

```
for each epoch do {
  for n times do {
    sample real images, x
    sample fake images, G(z)
    critic_loss = D(G(z))-D(x)
    critic_loss.backprop
    update critic
    critic.clip(-c,c)
  }
  sample fake images, G(z)
  gen_loss = -D(G(z)
  gen_loss.backprop
  update generator
}
```

However, paramater clipping regularizes the network and thus, limits its capacity. Wasserstein GAN with Gradient Penalty (WGAN-GP) was suggested as an alternative [3]. WGAN-GP replaces the parameter clipping with a loss function that penalizes large gradient. The equation for gradient penalty is shown in Equation 5. Below are the pseudo-cdoe for WGAN-GP training iterations.

```
for each epoch do {
  for n times do {
    sample real images, x
    sample fake images, G(z)
    critic_loss = D(G(z))-D(x)
    gradient_penalty(D)
    critic_loss.backprop
    gradient_penalty.backprop
    update critic
  }
  sample fake images, G(z)
  gen_loss = -D(G(z)
  gen_loss.backprop
  update generator
}
```

4.4. Latent Space and Latent Mapping

Latent mapping is the concept of forcing a generator to be able to generate a image when given a specific input. The motivation is to enable SinGAN to generate the original target image. This can be managed by adding reconstruction loss of the generator when given a fixed input. Equation 3 describes the reconstruction scheme. $\tilde{z}_i$ is a fixed input that exists in latent space and $x_{i-1}^{rec\uparrow}$ is the interpolated reconstructed image from the previous generator. In the implementation, $\tilde{z}_i$ are set to 0 except when training the generator at the lowest scale, G_0. Since G_0 cannot take $x_{i-1}^{rec\uparrow}$, non-zero $\tilde{z}_0$ is chosen and fixed.

$$x_i^{rec} = G_i(\tilde{z}_i, x_{i-1}^{rec\uparrow}) \tag{3}$$

Latent variables are chosen from a normal distribution. The variance of the distribution is chosen by Equation 4.

$$\sigma_{Z_i} = \gamma^2 \mathbb{E}[(x_{i-1}^{rec\uparrow} - a_i)^2] \tag{4}$$

The more distorted the reconstructed image from the previous generator is, the higher variation of noise is expected to be needed by the generator to add details, claimed by the authors.

4.5. Implementation/Training Details

$r = 0.75$ was used to scale the images.

Both generator and critic does not have any fully connected layers, and only uses convolutional layers. Batch normalization and leaky ReLU were used for each convolutional layer except for the outer-most layer.

Equation 5 are the loss functions used for training SinGAN.

$$
\begin{aligned}
L_{critic} &= \mathbb{E}[D_i(G_i(z_i, x_{i-1}^{\uparrow})] - \mathbb{E}[D_i(a_i)] \\
L_{gp} &= \lambda \cdot \mathbb{E}[(\nabla_{\hat{x}} D(\hat{x}) - 1)^2] \\
L_{adv} &= -\mathbb{E}[D_i(G_i(z_i, x_{i-1}^{\uparrow})] \\
L_{rec} &= \alpha \cdot \mathbb{E}[(x_i^{rec} - a_i)^2]
\end{aligned} \tag{5}
$$

$\lambda = 0.1$ and $\alpha = 10$ were used for the implementation. Adam optimizer was used for both the generator and the critic with learning rate of 5e-4, β_1 = 0.5, β_2 = 0.99. 2000 iterations and 3 nested iterations for both networks are done for training of each scale level. The learning is decreased by a factor of 0.1 after 1600 iterations [13].

It is also important to note that $\gamma = 0.1$ were used scaling the variance of the latent space. When γ is set too high, the generators face difficulties in learning to produce good results.

Another factor that is extremely important in obtaining good results is to control the resolution of the lowest-scale image. When the resolution of the smallest image is too big, it becomes more difficult for the first generator to learn the statistics, which can affect the learning of every GAN after that. When the resolution of the smallest image is too small, the first generator may overfit the data, which leads to very small variations in the random samples being generated. When the overfit happens, all samples will look the same as the target image. When preprocessing the image before the training, it is recommended to make the smallest image to have 100 to 200 pixels.

4.6. Implementation Result: Image Generation

As shown in Figure 4, the implementation was successful in generating images. SinGAN can generate variable-size of samples, since the generator only uses convolutional layers.

Training with natural scene images are usually successful. However, images like human, faces that have strong spatio-correlation of features, often ends with either unrealistic samples (due to underfitting) or less variation in images (due to overfitting), which both are uninteresting results.

One of the issues with multi-scale training is that training for a high-resolution image can take extremely long time and much more computational resources. When the image has high resolution, it will lead to more levels of scales. That means it will require more models and more time to train.

With SinGAN, there is a way to get around this problem and still obtain high-resolution images. Image does not have to be trained in the same resolution. First, downsample the image and trained for a low-resolution one instead of the original resolution. Once the training is done, the image is injected back to last or last few of the generators, referred

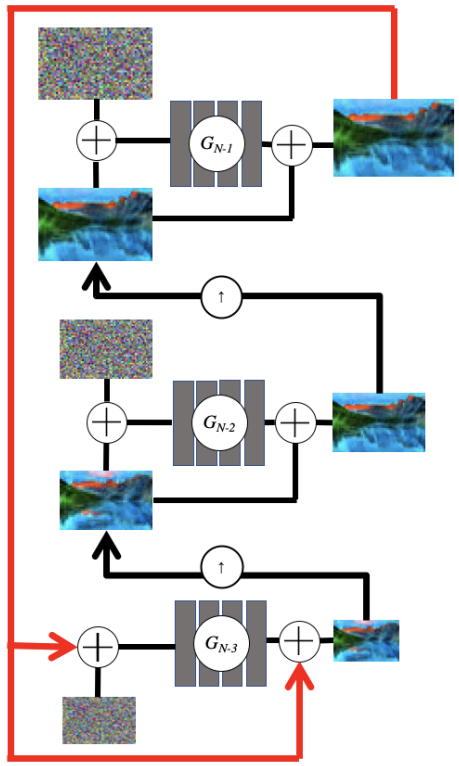

Image Before: Result without injection.

Image After: Result after 3 times of injection to the last 3 generators.

Figure 6. **Injection Method for SinGAN.** After training the multi-scale architecture, SinGAN is able to produce high-resolution images by feeding the output back to the last few generators. The model can be trained with image with resolution lower than the original image and exploit this method to produce higher-resolution images later to avoid spending enormous training time and computational resources.

Figure 7. **Super-Resolution with SinGAN**. Left is the original low resolution image (250x140 upsampled to 500x280). Right is the generated super-resolution (500x280).

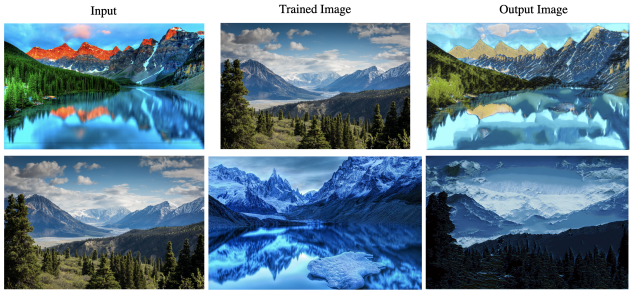

Figure 8. **Texture Transfer with SinGAN.** Texture can be transferred from one image to another by injecting an image to a trained SinGAN model. Left-most images are inputs, the middle column shows the images that the each SinGAN model was trained with, and the right-most images are the result after injection.

to as injection method as shown in Figure 6. The result will have more details.

4.7. Other Applications

4.7.1 Super-Resolution

Super-resolution can be achieved by the injection method shown in Figure 6. Take a low resolution image and train the model with it. After the training, the image can be upsampled and injected in the last or last few generators. An example is shown in Figure 7. However, the result is not as good as super-resolution networks that were trained on a dataset.

4.7.2 Texture Transfer

Texture transfer can be done by simply injecting an image to a trained SinGAN that was trained with a different image. Examples are shown in Figure 8.

4.7.3 Animation

After the training of SinGAN, short animation can be created by performing random walk in the latent space. Random walk is done only in z_0, the latent space of the first generator, while keeping all the latent variables for the other generators fixed (usually to 0). Equation 6 [13] shows the systematic way of performing random walk for this particular task.

$$z_0(t+1) = \alpha\tilde{z}_0 + (1-\alpha)(Z_0(t) + z_0^{diff}(t+1))$$
$$z_0^{diff}(t+1) = \beta(z_0(t) - z_0(t-1)) + (1-\beta)u_n(t) \quad (6)$$

α and β in Equation 6 are hyperparameters chosen prior random walk ($\alpha, \beta \in [0,1]$) . α controls how close $z_0(t)$ will remain around $\tilde{z}_0$, and β controls the rate of change in speed of walks (high α means walks will be closer to $\tilde{z}_0$ and higher β means speed of walks is more consistent).

4.8. Critique

With the initial purpose of generating images from a single image, not only the authors were able to achieve the goal but they were able to extend the idea to other applications.

Images like natural scenes that could still look realistic after deformed will most likely to yield great results with SinGAN. On the other hand, images that have strong spatio-correlations, such as human faces (requires eyes and mouth to be somewhat aligned in order to be recognized as a face), will result in less natural images or less variations in the images depending on how much the network is fitting to the image.

The results of this paper is absolutely brilliant. This approach may be extended to data augmentation or a more robust generative model that can yield great results with small datasets. The findings on this paper may defined the future of deep generative networks.

5. Conclusions and Final Remarks

Deep generative models have revolutionized the approach to image processing tasks. Although some parts may be magic, there were extensive amount engineering involved in all of these papers if looked closely in the details. It is also clear that computational power does not necessarily guarantee to solve the problem, but it is the matter of the approach. With proper engineering approach, more of unseen power of deep learning may be unlocked and expand the field of image processing even further.

References

[1] C. Choy and D. Xu, and G. JunYoung and K. Chen and S. Savarese. 3D-R2N2: A Unified Approach for Single and Multi-view 3D Object Reconstruction. *Proceedings of the European Conference on Computer Vision ECCV*, 2016 3

[2] H. Xie and H. Yao and X. Sun and S. Zhou and S. Zhang. Pix2Vox: Context-Aware 3D Reconstruction From Single and

Multi-View Images. *IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 2690-2698, 2019. 1, 3

[3] I. Gulrajani and F. Ahmed and M. Arjovsky and V. Dumoulin and A. C. Courville. Improved Training of Wasserstein GANs. *Advances in Neural Information Processing Systems, vol.30*, 2017. 4, 5

[4] J. Zhu, T. Park, P. Isola, and A. A. Efros. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. *IEEE International Conference on Computer Vision (ICCV)*, 2017. 2

[5] K. Simonyan, A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *International Conference on Learning Representations*, 2015 1, 3

[6] M. Chu, Y. Xie, and J. Mayer, and L. Leal-Taixe and N. Thuerey. Learning Temporal Coherence via Self-Supervision for GAN-based Video Generation (TecoGAN). *ACM Transactions on Graphics (TOG) vol.39(4)* , 2020. 1

[7] M. S. M. Sajjdi, R. Vemulapalli, M. Brown. Frame-Recurrent Video Super-Resolution. *IEEE Conference on Computer Vision and Pattern Recognition*, pages 6626-6634, 2018. 1, 3

[8] M. S. M. Sajjadi, B. Schölkopf, and M. Hirsch. 2017. Enhancenet: Single image super-resolution through automated texture synthesis. *In Computer Vision (ICCV), IEEE International Conference on. IEEE*, pages 4501–4510, 2017 3

[9] M. Arjovsky and S. Chintala and L. Bottou. Wasserstein GAN. *34th International Conference on Machine Learning*, 2017. 4

[10] O. Ronneberger and P.Fischer and T. Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. *Medical Image Computing and Computer-Assisted Intervention (MICCAI), vol.9351*, pages 234-241, 2015. 3

[11] R. Zhang, P. Isola, A. A. Efros, E. Shechtman, O. Wang. The Unreasonable Effectiveness of Deep Features as a Perceptual Metric. *CVPR*, 2018 3

[12] T. R. Shaham and T. Dekel and T. Michaeli. SinGAN: Learning a Generative Model From a Single Natural Image. *IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 4569-4579, 2019. 1, 3

[13] T. R. Shaham and T. Dekel and T. Michaeli. SinGAN: Learning a Generative Model from a Single Natural Image Supplementary Material. *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019. 6, 7

[14] Y. Jo, S. W. Oh, J. Kang, and S. J. Kim. Deep Video Super-Resolution Network Using Dynamic Upsampling Filters Without Explicit Motion Compensation. *In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3224–3232, 2018.

3

SinGAN Implementation

See https://github.com/kristerseiya/SinGAN_Implementation.git for Github Repository

models.py

```
import torch
from torch import nn
import torch.nn.functional as F
from math import ceil
from . import utils

class ConvBatchNormLeakyBlock(nn.Module):
    def __init__(self,input_channel,output_channel,kernel_size=3,stride=1,padding=0):
        super().__init__()
        self.conv = nn.Conv2d(input_channel,output_channel,kernel_size,stride,
                              padding=padding,bias=False)
        self.bn = nn.BatchNorm2d(output_channel)
        self.lrelu = nn.LeakyReLU(0.2)

    def forward(self,x):
        x = self.conv(x)
        x = self.bn(x)
        x = self.lrelu(x)
        return x
'''
a generator for SinGAN
takes two inputs (noise and generated image from previous generator)
z ----|
      + ---- G -- + --- output
lr ---|-----------|
'''
class AddSkipGenerator(nn.Module):
    '''
     constructor inputs:
     1. channels:
       a list of number of channels for each convolutional layer
       len(channels) must be equal to # of convolutional layer + 1
     2. kernels:
       a list of kernel-sizes for each convolutional layer
       len(kernels) must be equal to # of convolutional layer
    '''
    def __init__(self,channels,kernels):
        assert(len(channels)==(len(kernels)+1))
        super().__init__()
        num_conv = len(channels) - 1
        num_pad = sum(kernels) - len(kernels)

        '''
        input is padded in order to match the input and output size
        '''
        self.pad = nn.ZeroPad2d((num_pad//2,num_pad-num_pad//2, \
                                 num_pad//2,num_pad-num_pad//2))
        self.convlist = nn.ModuleList()

        if num_conv > 0:
            self.convlist.append(ConvBatchNormLeakyBlock(channels[0],channels[1],
                                 kernel_size=kernels[0]))
```

```
        for i in range(1, num_conv - 1):
            self.convlist.append(ConvBatchNormLeakyBlock(channels[i],channels[i+1],
                                 kernel_size=kernels[i]))
        if num_conv > 1:
            self.convlist.append(nn.Conv2d(channels[-2],channels[-1],
                                 kernels[-1],1))

    def forward(self,z,lr):
        x = z + lr
        x = self.pad(x)
        for l in self.convlist:
            x = l(x)
        return x + lr

'''
 critic (used for WGAN/WGAN-GP) with only convolutional num_layers
'''
class ConvCritic(nn.Module):
    '''
     constructor inputs:
     1. input_size:
       a tuple or list of input tensor size
     2. channels:
       a list of number of channels for each convolutional layer
       len(channels) must be equal to # of convolutional layer + 1
     3. kernels:
       a list of kernel-sizes for each convolutional layer
       len(kernels) must be equal to # of convolutional layer
     4. strides
       a list of strides for each convolutional layer
       len(strides) must be equal to # of convolutional layer
       if no input is given, 1-stride is done for every convolutional layer
     5. padding
       if true, zero padding is done at the input to match the input size and output size
       (input size and output size will not match if strides are not 1s)
       by default, no padding is done (works better without padding)
    '''
    def __init__(self,channels,kernels,strides=None,padding=False):

        assert(len(channels)==(len(kernels)+1))
        if strides is None:
            strides = [1] * len(kernels)
        num_conv = len(channels) - 1
        if padding is True:
            num_pad = sum(kernels) - len(kernels)
        else:
            num_pad = 0

        super().__init__()
        self.pad = nn.ZeroPad2d((num_pad//2,num_pad-num_pad//2, \
                                 num_pad//2,num_pad-num_pad//2))
        self.convlist = nn.ModuleList()

        if num_conv > 0:
            self.convlist.append(ConvBatchNormLeakyBlock(channels[0],channels[1],
                                                         kernel_size=kernels[0],stride=strides[0]))
        for i in range(1, num_conv - 1):
            self.convlist.append(ConvBatchNormLeakyBlock(channels[i],channels[i+1],
```

```
                                                kernel_size=kernels[i],stride=strides[i]))
        if num_conv > 1:
            self.convlist.append(nn.Conv2d(channels[-2],channels[-1],kernels[-1],strides[-1]))

    def forward(self,x):
        x = self.pad(x)
        for l in self.convlist:
            x = l(x)
        return x
'''
 discriminator for conventional GAN
'''
class Discriminator(nn.Module):
    '''
     constructor inputs:
     1. input_size:
       a tuple or list of input tensor size
     2. channels:
       a list of number of channels for each convolutional layer
       len(channels) must be equal to # of convolutional layer + 1
     3. kernels:
       a list of kernel-sizes for each convolutional layer
       len(kernels) must be equal to # of convolutional layer
     4. strides
       a list of strides for each convolutional layer
       len(strides) must be equal to # of convolutional layer
     5. n_dense_layer
       a number of dense layer at the end of the network
    '''
    def __init__(self,input_size,channels,kernels,strides,n_dense_layer=1):
        assert(len(channels)==(len(kernels)+1))
        super().__init__()
        num_conv = len(channels) - 1

        self.convlist = nn.ModuleList()

        if num_conv > 0:
            self.convlist.append(ConvBatchNormLeakyBlock(channels[0],channels[1],
                                                kernel_size=kernels[0],stride=strides[0]))
        for i in range(1, num_conv):
            self.convlist.append(ConvBatchNormLeakyBlock(channels[i],channels[i+1],
                                                kernel_size=kernels[i],stride=strides[i]))

        x = torch.randn(input_size)
        for l in self.convlist:
            x = l(x)
        self.num = x.nelement()
        self.linlist = nn.ModuleList()
        num = self.num
        for _ in range(n_dense_layer-1):
          self.linlist.append(nn.Linear(num,num//10))
          self.linlist.append(nn.LeakyReLU(0.2))
          num = num // 10
        if n_dense_layer > 0:
            self.linlist.append(nn.Linear(num,1))
        self.sgmd = nn.Sigmoid()

    def forward(self,x):
```

```
        for l in self.convlist:
            x = l(x)
        x = x.view(-1,self.num)
        for l in self.linlist:
            x = l(x)
        x = self.sgmd(x)
        return x

'''
 class used to store the trained generators, reconstruct, and sample random images
'''
class SinGAN():
    '''
     constructor inputs:
     1. scale
       scale factor for each downsampling
     2. trained_size:
       image size of image trained for. optional
     3. G
       list of generators, optional
     4. z_amp
       list of standard deviation of noises for each generator
     5. Z
       list of fixed noises for each generator for reconstruction
     6. recimg
       list of reconstructed images for each generator
       if G, z_amp, Z are given and this isn't,
       it will automatically reconstruct the image for you
    '''
    def __init__(self, scale, imgsize=None, G=None, z_amp=None, Z=None, recimg=None, device=None):

        if imgsize is None:
            imgsize = []
        if G is None:
            G = []
        if z_amp is None:
            z_amp = []
        if Z is None:
            Z = []
        if recimg is None:
            recimg = []

        if len(Z) != len(G) or \
           len(Z) != len(z_amp):
            raise Exception("G, z_amp, Z must be lists with same length")


        self.n_scale = len(G)

        if device is not None:
            for i in range(self.n_scale):
                G[i] = G[i].to(device)
                Z[i] = Z[i].to(device)

        self.scale = scale
        self.G = G
        self.z_amp = z_amp
        self.Z = Z
```

```
        self.recimg = recimg
        self.imgsize = imgsize
        self.device = device

        with torch.no_grad():
            for i in range(len(recimg),self.n_scale):
                if i == 0:
                    new_recimg = self.G[0](Z[0],0.)
                else:
                    prev = utils.upsample(self.recimg[-1],1./self.scale)
                    new_recimg = self.G[i](Z[i],prev)
                self.recimg.append(new_recimg.detach())

        if len(self.imgsize) < len(self.recimg):
            self.imgsize = [(x.size(-2),x.size(-1)) for x in self.recimg]

    '''
     append(self, netG, z_amp, fixed_z):
       appends a generator, noise information
     1. netG
       a new generator trained at one scale above
       it is recommended to disable gradient calculation of network by requires_grad = False
     2. z_amp
       standard deviation of noise input for the generator
     3. fixed_z
       fixed noise for the generator for reconstruction
    '''
    def append(self, netG, z_amp, fixed_z):
        self.G.append(netG)
        self.z_amp.append(z_amp)
        if type(fixed_z) == torch.Tensor:
            self.Z.append(fixed_z.detach())
        else:
            self.Z.append(fixed_z)

        with torch.no_grad():
            if self.n_scale > 0:
                prev = utils.upsample(self.recimg[-1],1./self.scale)
                new_recimg = netG(fixed_z,prev)
            else:
                new_recimg = netG(fixed_z,0.)

        self.recimg.append(new_recimg.detach())
        self.imgsize.append((new_recimg.size(-2),new_recimg.size(-1)))
        self.n_scale = self.n_scale + 1

        return self

    '''
     reconstruct(self,scale_level=None)
       outputs a reconstructed image
     1. scale_level
       the scale level of reconstruction
       the smallest scale is 1, next scale up is 2, so forth
       if not given, it will output the reconstruction at the final scale
    '''
    @torch.no_grad()
    def reconstruct(self,output_level=None):
```

```
        if self.G == []:
            return None
        if output_level is None:
            return self.recimg[-1]
        else:
            return self.recimg[output_level]

    '''
     sample(self,n_sample=1,scale_level=None)
       generates random samples
     1. n_sample
       a number of random samples
     2. scale_level
       the scale level of samples
    '''
    @torch.no_grad()
    def sample(self,input_size=None, output_level=-1, n_sample=1):
        if self.G == []:
            return None
        if output_level < 0:
            output_level = self.n_scale + output_level

        if input_size is not None:
            z = self.z_amp[0] * torch.randn(n_sample,self.Z[0].size(1),input_size[0],
                                            input_size[1],device=self.device)
        else:
            z = self.z_amp[0] * torch.randn(n_sample,self.Z[0].size(1),self.Z[0].size(2),
                                            self.Z[0].size(3),device=self.device)
        sample = self.G[0](z,0.)
        for i in range(1,output_level+1):
            sample = utils.upsample(sample, 1./self.scale)
            z = self.z_amp[i] * torch.randn_like(sample)
            sample = self.G[i](z,sample)
        return sample

    '''
     inject(self,x,n_sample=1,insert_level=None,scale_level=None)
     1. n_sample
       a number of random samples
     2. insert_level
       specifies where to inject the image
     3. scale_level
       the scale level of the output image
    '''
    @torch.no_grad()
    def inject(self,x,inject_level=-1,output_level=-1,n_sample=1):
        if self.G == []:
            return None
        if n_sample != 1:
            x = torch.cat(n_sample*[x],0)
        if inject_level < 0:
            inject_level = self.n_scale + inject_level
        if output_level < 0:
            output_level = self.n_scale + output_level

        z = self.z_amp[inject_level] * torch.randn_like(x)
        x = self.G[inject_level](z,x)
        for i in range(inject_level+1,output_level+1):
```

```
            x = utils.upsample(x, 1./self.scale)
            z = self.z_amp[i] * torch.randn_like(x)
            x = self.G[i](z,x)
        return x

    def to(self,device):
        self.device = device
        for i in range(self.n_scale):
            self.G[i] = self.G[i].to(device)
            self.recimg[i] = self.recimg[i].to(device)
            if type(self.Z[i]) is torch.Tensor:
                self.Z[i] = self.Z[i].to(device)
        return self

    def walk(self,n,alpha,beta):
        z_t1 = 0.
        z_t = self.Z[0]

        if n > 0:
            xs = self.G[0](z_t,0.)
            for j in range(1,self.n_scale):
                xs = utils.upsample(xs,1./self.scale)
                xs = self.G[j](0.,xs)
        for i in range(1,n):
            z_diff = beta * (z_t - z_t1) + (1 - beta) * torch.rand_like(z_t)
            z_t1 = z_t
            z_t = alpha * self.Z[0]  + (1 - alpha) * (z_t + z_diff)
            x = self.G[0](z_t,0.)
            for j in range(1,self.n_scale):
                x = utils.upsample(x,1./self.scale)
                x = self.G[j](0.,x)
            xs = torch.cat([xs,x],0)
        return xs

'''
 loading and saving SinGAN
'''
def save_singan(singan,path):
    torch.save({'n_scale': singan.n_scale, \
                'scale': singan.scale, \
                'trained_size': singan.imgsize, \
                'models': singan.G, \
                'noise_amp': singan.z_amp, \
                'fixed_noise': singan.Z, \
                'reconstructed_images': singan.recimg \
                },path)
    return

def load_singan(path):
    load = torch.load(path)
    singan = SinGAN(load['scale'], load['trained_size'], load['models'], \
                    load['noise_amp'], load['fixed_noise'],load['reconstructed_images'])
    return singan
```

utils.py

```
from PIL import Image
```

```
import torch
import torchvision
from torchvision import transforms
from torch import nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
import numpy as np
import random
from math import ceil, floor
import imageio

'''
 loades image given a path to image
'''
def load_image(path):
    img = Image.open(path)
    return img

'''
 creates a list of images with different scales
'''
def create_scaled_images(img,scale,min_len,max_len,match_min=True):
    scaled_imgs = []
    if type(img) == list:
        width, height = img[0].size
    else:
        width, height = img.size

    if match_min == True:
        if width <= height:
            new_width = (int)(min_len)
            new_height = (int)(min_len / width * height)
        else:
            new_height = (int)(min_len)
            new_width = (int)(min_len / height * width)
        while new_height <= max_len and new_width <= max_len:
            if type(img) == list:
                new_imgs = []
                for i in img:
                    new_img = i.resize((new_width,new_height))
                    new_imgs.append(new_img)
                scaled_imgs.append(new_imgs)
            else:
                new_img = img.resize((new_width,new_height))
                scaled_imgs.append(new_img)
            new_width, new_height = floor(new_width / scale), floor(new_height / scale)
    else:
        if width <= height:
            new_height = max_len
            new_width = (int)(max_len / height * width)
        else:
            new_width = max_len
            new_height = (int)(max_len / width * height)
        while new_width >= min_len and new_height >= min_len:
            if type(img) == list:
                new_imgs = []
                for i in img:
                    new_img = i.resize((new_width,new_height))
```

```
                    new_imgs.append(new_img)
                scaled_imgs.append(new_imgs)
            else:
                new_img = img.resize((new_width,new_height))
                scaled_imgs.append(new_img)
            new_width, new_height = ceil(new_width * scale), ceil(new_height * scale)
        scaled_imgs = scaled_imgs[::-1]

    return scaled_imgs

'''
 copy image to tensor
'''
def convert_image2tensor(img,transform=None,device=None):
    if transform==None:
        transform = transforms.Compose([transforms.ToTensor(), \
                        transforms.Normalize([0.5,0.5,0.5],[0.5,0.5,0.5])])
    tensor = transform(img)
    tensor = tensor.unsqueeze(0)
    if device != None:
        tensor = tensor.to(device)
    return tensor

'''
 copy a list of images to a list of tensors
'''
def convert_images2tensors(imgs,transform=None, device=None):
    if transform==None:
        transform = transforms.Compose([transforms.ToTensor(), \
                        transforms.Normalize([0.5,0.5,0.5],[0.5,0.5,0.5])])
    transformed_imgs = []
    for img in imgs:
        if type(img) == list:
            tensor = []
            for i in img:
                t = transform(i)
                tensor.append(t)
            tensor = torch.stack(tensor,0)
        else:
            tensor = transform(img)
            tensor = tensor.unsqueeze(0)
        if device != None:
            tensor = tensor.to(device)
        transformed_imgs.append(tensor)
    return transformed_imgs

'''
 plot tensors as images
'''
def show_tensor_image(tensor,row_n=1,use_matplot=True):
    tensor = tensor.detach().cpu()
    tensor = torchvision.utils.make_grid(tensor,row_n)
    img = tensor.numpy()
    img = img.transpose([1,2,0])
    img = (img + 1) / 2
    img = np.clip(img,0.,1.)
    if use_matplot is True:
        plt.imshow(img)
```

```
        plt.subplots_adjust(left=0.05, right=0.95, top=0.95, bottom=0.05)
        plt.show()
    else:
        img = img * 255
        img = img.astype(np.uint8)
        x = Image.fromarray(img)
        x.show()


'''
 parameter initialization
'''
def xavier_uniform_weight_init(layer):
    if type(layer) == nn.Conv2d:
        torch.nn.init.xavier_uniform_(layer.weight)
        if layer.bias is not None:
            layer.bias.data.fill_(0.1)
    return

def xavier_normal_weight_init(layer):
    if type(layer) == nn.Conv2d:
        torch.nn.init.xavier_normal_(layer.weight)
        if layer.bias is not None:
            layer.bias.data.fill_(0.1)
    return

'''
 turn off gradient calculation
'''
def disable_grad(net):
    for p in net.parameters():
        p.requires_grad = False
    return

'''
 turn on gradient calculation
'''
def enable_grad(net):
    for p in net.parameters():
        p.requires_grad = True
    return

def downsample(tensor,r):
    h, w = tensor.size(-2), tensor.size(-1)
    nh , nw = ceil(h*r), ceil(w*r)
    tensor = F.interpolate(tensor,(nh,nw))
    return tensor

def upsample(tensor, r):
    h, w = tensor.size(-2), tensor.size(-1)
    nh , nw = floor(h*r), floor(w*r)
    tensor = F.interpolate(tensor,(nh,nw))
    return tensor

def save_tensor_image(tensor, path, row_n=2):
    tensor = tensor.detach().cpu()
    tensor = torchvision.utils.make_grid(tensor,row_n)
    arr = tensor.numpy()
```

```
    arr = arr.transpose([1,2,0])
    arr = (arr + 1) / 2
    arr = np.clip(arr,0.,1.)
    arr = arr * 255
    arr = arr.astype(np.uint8)
    img = Image.fromarray(arr)
    img.save(path)
    return

def save_tensor_gif(tensor, path):
    tensor = tensor.detach().cpu()
    images = tensor.numpy()
    images = images.transpose([0,2,3,1])
    images = (images + 1) / 2
    images = np.clip(images, 0., 1.)
    images = images * 255
    images = images.astype(np.uint8)
    imageio.mimsave(path,images)
    return
```

train.py

```
import torch
from torch import nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
from . import utils
import numpy as np

'''
 calculates gradient penalty loss for WGAN-GP critic
'''
def GradientPenaltyLoss(netD,real,fake):
    real = real.expand(fake.size())
    alpha = torch.rand(fake.size(0),1,1,1,device=fake.device)
    alpha = alpha.expand(fake.size())
    interpolates = alpha * real + (1-alpha) * fake
    interpolates = torch.autograd.Variable(interpolates,requires_grad=True)
    Dout_interpolates = netD(interpolates)
    gradients = torch.autograd.grad(outputs=Dout_interpolates, inputs=interpolates, \
                                    grad_outputs=torch.ones_like(Dout_interpolates), \
                                    create_graph=True,retain_graph=True,only_inputs=True)[0]


    grad_penalty = ((gradients.norm(2,dim=1)-1)**2).mean()

    return grad_penalty

'''
trains GAN for one scale of SinGAN
1. img
  target image
2. netG
  generator to train
3. netG_optim
  optimizer for generator
4. netD
  discriminator or critic
```

```
5. netD_optim
  optimizer for discriminator or critic
6. singan
  SinGAN object that includes generator at lower scale
7. num_epoch
  number of iteration
8. mode
  specifies type of GAN
  option: 'wgangp', 'wgan', 'gan'
  gan type is 'wgangp' by default
9. netG_lrscheduler
  learning rate scheduler for the generator, optional
10. netD_lrscheduler
  learning rate scheduler for the critic, optional
11. use_zero
  if True, use zero tensor for reconstruction
  however, zero will not be used when training GAN at the lowest scale
12. batch_size
  a number of samples generated for each parameter update
13. recloss
  a function used for calculation of reconstruction loss, nn.MSELoss is used by default
14. recloss_scale
  scalar multiplier for reconstruction loss, 10 is used by default
15. gp_scale
  scalar multiplier for gradient penalty loss, 0.1 is used by default
16. clip_range
  clipping range for critic's parameter when training for WGAN
  (no clipping is done for WGAN-GP)
  0.01 by default
17. z_std_scale
  scalar multiplier for input noise's standard deviation
18. netG_iter
  a number of consecutive parameter updates for generator, 3 by default
19. netD_iter
  a number of consecutive parameter updates for critic, 3 by default
20. freq
  frequency of plotting random samples
21. figsize
  size of the plot for generated samples
'''
def train_singan_onescale(img, \
                          netG,netG_optim, \
                          netD,netD_optim, \
                          singan,num_epoch, \
                          mode='wgangp', \
                          netG_lrscheduler=None, netD_lrscheduler=None, \
                          use_zero=True,batch_size=1, \
                          recloss_fun=None,recloss_scale=10,gp_scale=0.1,clip_range=0.01, \
                          z_std_scale=0.1, \
                          netG_iter=3,netD_iter=3, \
                          log_freq=0, plot_freq=0, figsize=(15,15)):

    imgsize = (img.size(-2),img.size(-1))

    if  singan.n_scale == 0:
        z_std = z_std_scale
        fixed_z = z_std * torch.randn_like(img)
    else:
```

```
        prev_rec = singan.reconstruct()
        prev_rec = F.interpolate(prev_rec,imgsize)
        z_std = z_std_scale * torch.sqrt(F.mse_loss(prev_rec,img)).item()
        if use_zero:
            fixed_z = 0.
        else:
            fixed_z = z_std * torch.randn_like(img)

    if recloss_fun is None:
        ReconstructionLoss = nn.MSELoss()
    else:
        ReconstructionLoss = recloss_fun

    meta_data = np.zeros((num_epoch,(netG_iter+netD_iter)*2))

    netG.train()
    netD.train()

    for epoch in range(1,num_epoch+1):

        utils.enable_grad(netD)
        utils.disable_grad(netG)

        for i in range(netD_iter):

            '''
             generate image
            '''
            z = z_std * torch.randn(batch_size,img.size(1),
                           img.size(2),img.size(3),device=img.device)
            if singan.n_scale == 0:
                Gout = netG(z,0.)
            else:
                base = singan.sample(n_sample=batch_size)
                base = F.interpolate(base,imgsize)
                Gout = netG(z,base)

            '''
             train critic
            '''
            netD_optim.zero_grad()
            Dout_real = netD(img)
            Dout_fake = netD(Gout.detach())
            if mode == 'gan':
                # calculate
                real_loss = F.binary_cross_entropy(Dout_real,torch.ones_like(Dout_real))
                real_loss.backward()
                fake_loss = F.binary_cross_entropy(Dout_fake,torch.zeros_like(Dout_fake))
                fake_loss.backward()
                D_loss_total = real_loss.item() + fake_loss.item()
                wdistance = 0.0
            elif mode == 'wgan':
                '''
                 calculate wasserstein distance
                '''
                wdistance_loss = Dout_fake.mean() - Dout_real.mean()
                wdistance_loss.backward()
                D_loss_total = wdistance_loss.item()
```

```
                wdistance = wdistance_loss.item()
            elif mode == 'wgangp':
                '''
                 calculate wasserstein distance and gradient penalty
                '''
                wdistance_loss = Dout_fake.mean() - Dout_real.mean()
                wdistance_loss.backward()
                grad_penalty_loss = GradientPenaltyLoss(netD,img,Gout) * gp_scale
                grad_penalty_loss.backward()
                D_loss_total = wdistance_loss.item() + grad_penalty_loss.item()
                wdistance = wdistance_loss.item()

            netD_optim.step()
            meta_data[epoch-1,i] = D_loss_total
            meta_data[epoch-1,i+netD_iter] = wdistance

            if mode == 'wgan':
                for p in netD.parameters():
                    p.data.clamp_(-clip_range,clip_range)


        utils.disable_grad(netD)
        utils.enable_grad(netG)

        for j in range(netG_iter):

            '''
             generate image
            '''
            if singan.n_scale == 0:
                Gout = netG(z,0.)
            else:
                base = singan.sample(n_sample=batch_size)
                base = F.interpolate(base,imgsize)
                Gout = netG(z,base)

            '''
             train generator
            '''
            netG_optim.zero_grad()
            Dout_fake = netD(Gout)
            if mode == 'gan':
                adv_loss = F.binary_cross_entropy(Dout_fake,torch.ones_like(Dout_fake))
                adv_loss.backward()
            elif mode == 'wgan' or mode == 'wgangp':
                adv_loss = - Dout_fake.mean()
                adv_loss.backward()
            if singan.n_scale == 0:
                rec = netG(fixed_z,0.)
            else:
                rec = netG(fixed_z,prev_rec)
            rec_loss = ReconstructionLoss(rec,img) * recloss_scale
            rec_loss.backward()
            G_loss_total = adv_loss.item() + rec_loss.item()

            netG_optim.step()

            meta_data[epoch-1,netD_iter*2+j] = G_loss_total
```

```
            meta_data[epoch-1,netD_iter*2+netG_iter+j] = rec_loss

        if netD_lrscheduler is not None:
            netD_lrscheduler.step()
        if netG_lrscheduler is not None:
            netG_lrscheduler.step()

        if (log_freq != 0) and (epoch % log_freq == 0):
            print("[{:d}/{:d}]".format(epoch,num_epoch))
            print("   generator loss   : {:.3f}".format(G_loss_total))
            print(" reconstruction loss: {:.3f}".format(rec_loss))
            if mode == 'gan':
                print(" discriminator loss : {:.3f}".format(D_loss_total))
            elif mode == 'wgan' or mode == 'wgangp':
                print("     critic loss    : {:.3f}".format(D_loss_total))
                print("wasserstein distance: {:.3f}".format(wdistance))

        if (plot_freq != 0) and (epoch % plot_freq == 0):
            netG.eval()
            with torch.no_grad():
                '''
                 display sample from generator
                '''
                if singan.n_scale == 0:
                    z = z_std * torch.randn(7,img.size(1),img.size(2),
                                            img.size(3),device=img.device)
                    sample = netG(z,0.)
                    rec = netG(fixed_z,0.)
                else:
                    z = z_std * torch.randn(7,img.size(1),img.size(2),
                                            img.size(3),device=img.device)
                    base = singan.sample(n_sample=7)
                    base = F.interpolate(base,imgsize)
                    sample = netG(z,base)
                    rec = netG(fixed_z,prev_rec)

                sample = torch.cat([sample,rec],0)

            plt.figure(figsize=figsize)
            utils.show_tensor_image(sample,4)
            netG.train()

    return z_std, fixed_z, meta_data


def train(r, scaled_img_list, device=None):

    singan = models.SinGAN(r)

    n_scale = len(scaled_img_list)
    tensor_list = utils.convert_images2tensors(scaled_img_list)
    nc = 8


    for i in range(num_scales):

        img = tensor_list[i]
        netG = models.AddSkipGenerator([3,nc,nc,nc,3],[3,3,3,3])
```

```
        if device is not None:
            netG = netG.to(device)
        netG.apply(xavier_uniform_weight_init)
        netD = models.ConvCritic([3,nc,nc,nc,1],[3,3,3,3])
        if device is not None:
            netD = netD.to(device)
        netD.apply(xavier_uniform_weight_init)
        netG_optim = torch.optim.Adam(netG.parameters(),lr=5e-4,betas=(0.5,0.999))
        netD_optim = torch.optim.Adam(netD.parameters(),lr=5e-4,betas=(0.5,0.999))
        netG_lrscheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer=netG_optim,
                                                                milestones=[1600],gamma=0.1)
        netD_lrscheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer=netD_optim,
                                                                milestones=[1600],gamma=0.1)

        print("Training Scale {}".format(i+1))

        z_std, fixed_z, _ = train_singan_onescale(img, \
                                                  netG,netG_optim, \
                                                  netD,netD_optim, \
                                                  netG_chain,2000, \
                                                  mode='wgangp', \
                                                  netG_lrscheduler=netG_lrscheduler,
                                                  netD_lrscheduler=netD_lrscheduler, \
                                                  use_zero=True,batch_size=1, \
                                                  recloss_fun=recloss_fun,
                                                  recloss_scale=10,gp_scale=0.1,
                                                  clip_range=0.01, \
                                                  z_std_scale=0.1, \
                                                  netG_iter=3,netD_iter=3, \
                                                  log_freq=50, plot_freq=0,
                                                  figsize=(15,15))

        netG.eval()
        utils.disable_grad(netG)
        singan.append(netG,z_std,fixed_z)

        nc = (int) (num_channel / r)

    return singan
```

main.py

```
import torch
from . import models, utils, train
import argparse

img = loadImage("SinGAN_Implementation/3-nature-wallpaper-mountain.jpg")
print(img.size)
plt.imshow(img)
plt.show()


parser = argparse.ArgumentParser()
parser.add_argument('--image', required=True, help="input image")
parser.add_argument('--r', default=0.75, help="scale of downsampling")
parser.add_argument('--min_len', default=12, help="minimum height or width of image")
parser.add_argument('--max_len', default=250, help="maximum height or width of image")
```

```
parser.add_argument('--match_min', default=True, help="match the minimum length")

arg = parser.parse_args()

img = load_image(args.images)
scaled_img_list = utils.create_scaled_images(img,arg.r,arg.min_len,arg.max_len,
                                             match_min=arg.match_min)

print("{} scales created".format(len(scaled_img_list)))

if torch.cuda.is_available() is True:
    print("Using CUDA")
    device = torch.device('cuda:0')
else:
    device = None

print("training starts...")
singan = train.train(arg.r, scaled_img_list, device=device)

cpu = torch.device('cpu')
singan = singan.to(cpu)
models.save_singan(singan,'singan.pth')
```