

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 attempts 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.

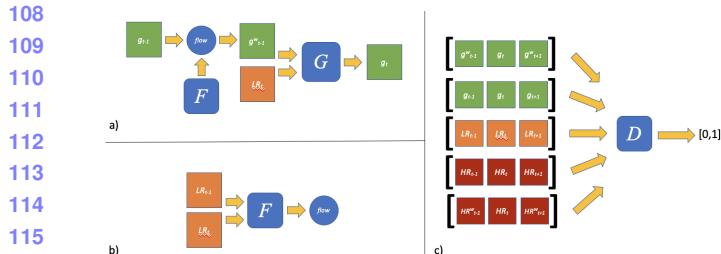


Figure 1. Overview of Super-Resolution TecoGAN. a) Generator, G takes low-resolution frame LR_t and warped, previously generated frame g_{t-1}^w to estimate the high-resolution of the current frame g_t . b) Optical flow that is used for warping images are estimated by motion-estimation network, F . c) Spatio-temporal 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 temporal coherence while maintaining 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.

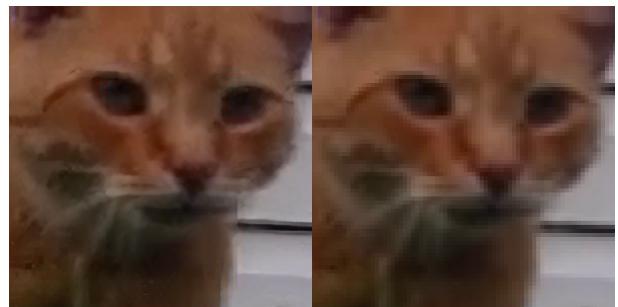


Figure 2. Super-Resolution with TecoGAN. Left: Frame generated by TecoGAN, Right: Original Frame

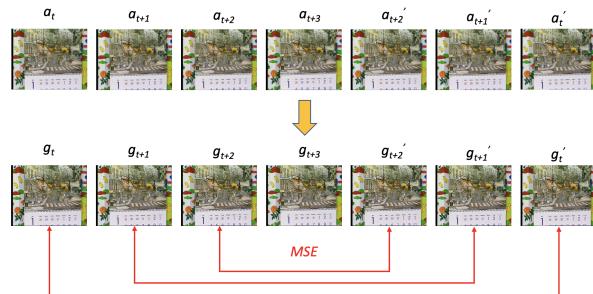


Figure 3. **Ping Pong Loss** is measures the generator’s ability to produce forward and backward motion. Input frames are mirrored (a_i and a_i' are identical) during pre-processing. Ping Pong Loss is simply the MSELoss between g_i and g_i' .

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 initial 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.

108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215

216

2.6. Performance

For video-super-resolution task, TecoGAN successfully 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.

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

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).

Model	Inference Time		
	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.

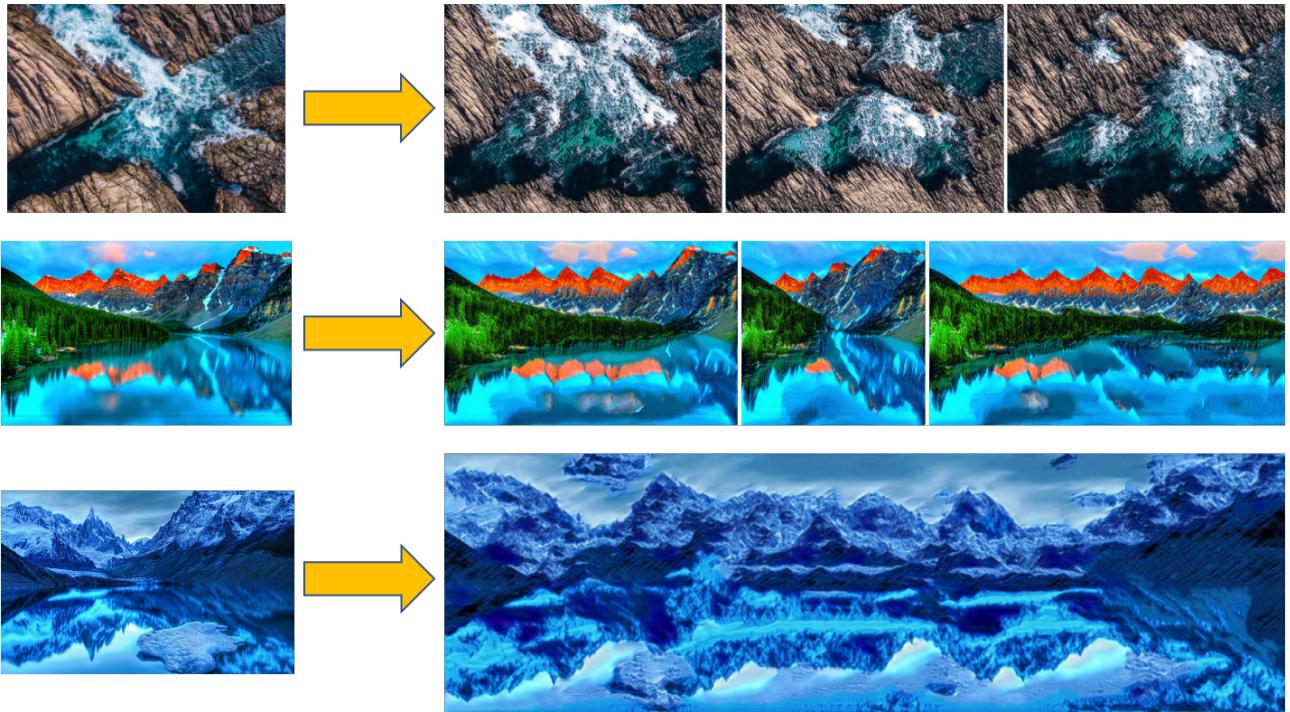


Figure 4. **Random Samples from SinGAN.** Images on the left are the images used for training. The images on the right shows the random samples from the trained model. Since generators use only convolutional layers, they can take variable-sized inputs and hence, produce variable-sized outputs.

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 n th 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 neu-

ron 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.

$$|f(x) - f(y)| \leq C|x - y| \quad (2)$$

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

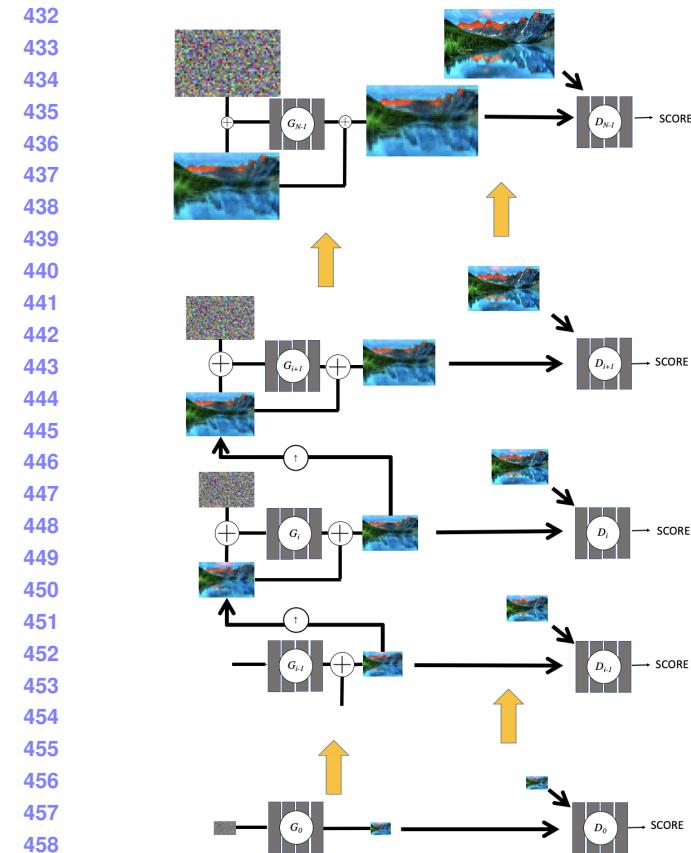


Figure 5. **Multi-Scale Architecture for SinGAN.** Multiple levels of scales of images are created, and for each level, a generator and a discriminator are trained. Training starts from the lowest scale, and every generator after that depends on all the previous generators that were trained in the lower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

```

469 for each epoch do {
470     for n times do {
471         sample real images, x
472         sample fake images, G(z)
473         critic_loss = D(G(z)) - D(x)
474         critic_loss.backprop
475         update critic
476         critic.clip(-c, c)
477     }
478     sample fake images, G(z)
479     gen_loss = -D(G(z))
480     gen_loss.backprop
481     update generator
482 }
```

However, parameter clipping regularizes the network and thus, limits its capacity. Wasserstein GAN with Gra-

dient 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-code for WGAN-GP training iterations.

```

486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539

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 $x_{i-1}^{\text{rec}\uparrow}$ 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 $x_{i-1}^{\text{rec}\uparrow}$, non-zero \tilde{z}_0 is chosen and fixed.

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

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

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

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.

540
 541
 542 $L_{critic} = \mathbb{E}[D_i(G_i(z_i, x_{i-1}^\uparrow)] - \mathbb{E}[D_i(a_i)]$
 543
 544 $L_{gp} = \lambda \cdot \mathbb{E}[(\nabla_{\hat{x}} D(\hat{x}) - 1)^2]$
 545
 546 $L_{adv} = -\mathbb{E}[D_i(G_i(z_i, x_{i-1}^\uparrow)]$
 547
 $L_{rec} = \alpha \cdot \mathbb{E}[(x_i^{rec} - a_i)^2]$

548 $\lambda = 0.1$ and $\alpha = 10$ were used for the implementation.
 549 Adam optimizer was used for both the generator and the
 550 critic with learning rate of 5e-4, $\beta_1 = 0.5$, $\beta_2 = 0.99$. 2000
 551 iterations and 3 nested iterations for both networks are done
 552 for training of each scale level. The learning is decreased
 553 by a factor of 0.1 after 1600 iterations [13].

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

558 Another factor that is extremely important in obtaining
 559 good results is to control the resolution of the lowest-scale
 560 image. When the resolution of the smallest image is too
 561 big, it becomes more difficult for the first generator to learn
 562 the statistics, which can affect the learning of every GAN
 563 after that. When the resolution of the smallest image is too
 564 small, the first generator may overfit the data, which leads
 565 to very small variations in the random samples being gen-
 566 erated. When the overfit happens, all samples will look the
 567 same as the target image. When preprocessing the image
 568 before the training, it is recommended to make the smallest
 569 image to have 100 to 200 pixels.
 570

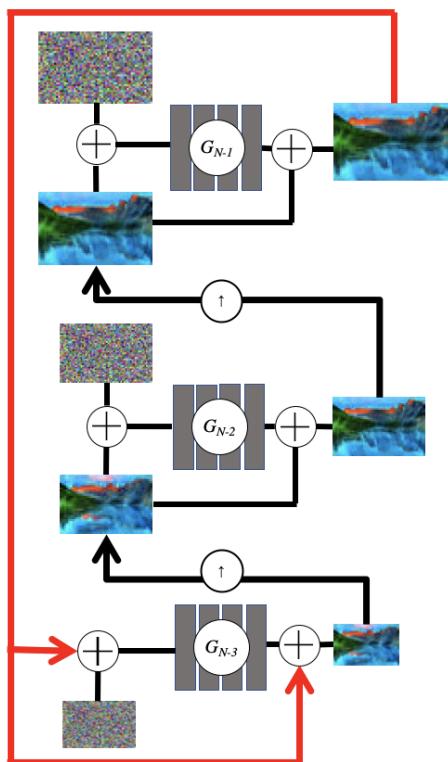
571 4.6. Implementation Result: Image Generation

572 As shown in Figure 4, the implementation was success-
 573 ful in generating images. SinGAN can generate variable-
 574 size of samples, since the generator only uses convolutional
 575 layers.
 576

577 Training with natural scene images are usually success-
 578 ful. However, images like human, faces that have strong
 579 spatio-correlation of features, often ends with either unreal-
 580 istic samples (due to underfitting) or less variation in images
 581 (due to overfitting), which both are uninteresting results.

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

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



619 572
 620 573
 621 574
 622 575
 623 576
 624 577
 625 578
 626 579
 627 580
 628 581
 629 582
 630 583
 631 584
 632 585
 633 586
 634 587
 635 588
 636 589
 637 590
 638 591
 639 592
 640 593

Image Before: Result without injection.



641 572
 642 573
 643 574
 644 575
 645 576
 646 577
 647 578
 648 579
 649 580
 650 581
 651 582
 652 583
 653 584
 654 585
 655 586
 656 587
 657 588
 658 589
 659 590
 660 591
 661 592
 662 593

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.

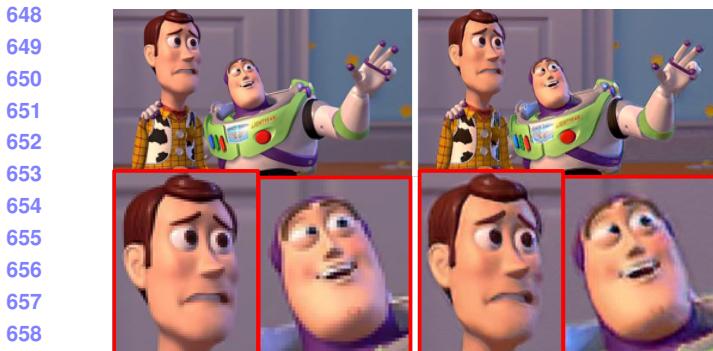


Figure 7. **Super-Resolution with SinGAN.** Left is the original low resolution image (250x140 upsampled to 500x280). Right is the generated super-resolution (500x280).



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. Ran-

dom 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.

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

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

- 756 Multi-View Images. *IEEE/CVF International Conference on* 810
757 *Computer Vision (ICCV)*, pages 2690-2698, 2019. 1, 3 811
758 [3] I. Gulrajani and F. Ahmed and M. Arjovsky and V. Dumoulin 812
759 and A. C. Courville. Improved Training of Wasserstein GANs. 813
760 *Advances in Neural Information Processing Systems*, vol.30, 814
761 2017. 4, 5 815
762 [4] J. Zhu, T. Park, P. Isola, and A. A. Efros. Unpaired Image- 816
763 to-Image Translation using Cycle-Consistent Adversarial 817
764 Networks. *IEEE International Conference on Computer Vision* 818
765 (*ICCV*), 2017. 2 819
766 [5] K. Simonyan, A. Zisserman. Very Deep Convolutional 820
767 Networks for Large-Scale Image Recognition. *International 821
768 Conference on Learning Representations*, 2015 1, 3 822
769 [6] M. Chu, Y. Xie, and J. Mayer, and L. Leal-Taixe and N. 823
770 Thurey. Learning Temporal Coherence via Self-Supervision 824
771 for GAN-based Video Generation (TecoGAN). *ACM Transactions 825
772 on Graphics (TOG)* vol.39(4) , 2020. 1 826
773 [7] M. S. M. Sajjadi, R. Vemulapalli, M. Brown. Frame-Recurrent 827
774 Video Super-Resolution. *IEEE Conference on Computer Vision* 828
775 and Pattern Recognition, pages 6626-6634, 2018. 1, 3 829
776 [8] M. S. M. Sajjadi, B. Schölkopf, and M. Hirsch. 2017. Enhancenet: 830
777 Single image super-resolution through automated 831
778 texture synthesis. In *Computer Vision (ICCV)*, *IEEE International 832
779 Conference on IEEE*, pages 4501–4510, 2017 3 833
780 [9] M. Arjovsky and S. Chintala and L. Bottou. Wasserstein GAN. 834
781 *34th International Conference on Machine Learning*, 2017. 4 835
782 [10] O. Ronneberger and P. Fischer and T. Brox. U-Net: 836
783 Convolutional Networks for Biomedical Image Segmentation. *Medical 837
784 Image Computing and Computer-Assisted Intervention (MIC- 838
785 CAI)*, vol.9351, pages 234-241, 2015. 3 839
786 [11] R. Zhang, P. Isola, A. A. Efros, E. Shechtman, O. Wang. The 840
787 Unreasonable Effectiveness of Deep Features as a Perceptual 841
788 Metric. *CVPR*, 2018 3 842
789 [12] T. R. Shaham and T. Dekel and T. Michaeli. SinGAN: 843
790 Learning a Generative Model From a Single Natural Image 844
791 *IEEE/CVF International Conference on Computer Vision* 845
792 (*ICCV*), pages 4569-4579, 2019. 1, 3 846
793 [13] T. R. Shaham and T. Dekel and T. Michaeli. SinGAN: Learning 847
794 a Generative Model from a Single Natural Image Supplementary 848
795 Material. *IEEE/CVF International Conference on* 849
796 *Computer Vision (ICCV)*, 2019. 6, 7 850
797 [14] Y. Jo, S. W. Oh, J. Kang, and S. J. Kim. Deep Video 851
798 Super-Resolution Network Using Dynamic Upsampling Filters 852
799 Without Explicit Motion Compensation. In *Proceedings of the* 853
800 *IEEE Conference on Computer Vision and Pattern Recognition*, 854
801 pages 3224–3232, 2018. 855
802
803 3 856
804
805
806
807
808
809

```

864 SinGAN Implementation
865
866 See https://github.com/kristferseiya/SinGAN\_Implementation.git for Github Repository
867
868 models.py
869
870
871 import torch
872 from torch import nn
873 import torch.nn.functional as F
874 from math import ceil
875 from . import utils
876
877 class ConvBatchNormLeakyBlock(nn.Module):
878     def __init__(self, input_channel, output_channel, kernel_size=3, stride=1, padding=0):
879         super().__init__()
880         self.conv = nn.Conv2d(input_channel, output_channel, kernel_size, stride,
881                             padding=padding, bias=False)
882         self.bn = nn.BatchNorm2d(output_channel)
883         self.lrelu = nn.LeakyReLU(0.2)
884
885     def forward(self, x):
886         x = self.conv(x)
887         x = self.bn(x)
888         x = self.lrelu(x)
889         return x
890
891     """
892     a generator for SinGAN
893     takes two inputs (noise and generated image from previous generator)
894     z ----|
895         + ---- G -- + --- output
896     lr ---|-----|
897     """
898
899     class AddSkipGenerator(nn.Module):
900         """
901             constructor inputs:
902                 1. channels:
903                     a list of number of channels for each convolutional layer
904                     len(channels) must be equal to # of convolutional layer + 1
905                 2. kernels:
906                     a list of kernel-sizes for each convolutional layer
907                     len(kernels) must be equal to # of convolutional layer
908         """
909
910         def __init__(self, channels, kernels):
911             assert(len(channels) == (len(kernels) + 1))
912             super().__init__()
913             num_conv = len(channels) - 1
914             num_pad = sum(kernels) - len(kernels)
915
916             """
917                 input is padded in order to match the input and output size
918             """
919             self.pad = nn.ZeroPad2d((num_pad // 2, num_pad - num_pad // 2, \
920                                     num_pad // 2, num_pad - num_pad // 2))
921             self.convlist = nn.ModuleList()
922
923             if num_conv > 0:
924                 self.convlist.append(ConvBatchNormLeakyBlock(channels[0], channels[1],
925                                               kernel_size=kernels[0]))
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971

```

```

972      53     for i in range(1, num_conv - 1):
973      54         self.convlist.append(ConvBatchNormLeakyBlock(channels[i], channels[i+1],
974      55                         kernel_size=kernels[i]))
975      56     if num_conv > 1:
976      57         self.convlist.append(nn.Conv2d(channels[-2], channels[-1],
977      58                         kernels[-1], 1))
978      59
979      60     def forward(self, z, lr):
980      61         x = z + lr
981      62         x = self.pad(x)
982      63         for l in self.convlist:
983      64             x = l(x)
984      65         return x + lr
985      66
986      67 """
987      68     critic (used for WGAN/WGAN-GP) with only convolutional num_layers
988      69 """
989      70 class ConvCritic(nn.Module):
990      71     """
991      72     constructor inputs:
992      73     1. input_size:
993      74         a tuple or list of input tensor size
994      75     2. channels:
995      76         a list of number of channels for each convolutional layer
996      77         len(channels) must be equal to # of convolutional layer + 1
997      78     3. kernels:
998      79         a list of kernel-sizes for each convolutional layer
999      80         len(kernels) must be equal to # of convolutional layer
1000     81     4. strides:
1001     82         a list of strides for each convolutional layer
1002     83         len(strides) must be equal to # of convolutional layer
1003     84         if no input is given, 1-stride is done for every convolutional layer
1004     85     5. padding:
1005     86         if true, zero padding is done at the input to match the input size and output size
1006     87         (input size and output size will not match if strides are not 1s)
1007     88         by default, no padding is done (works better without padding)
1008     89 """
1009     90     def __init__(self, channels, kernels, strides=None, padding=False):
1010     91
1011     92         assert(len(channels)==(len(kernels)+1))
1012     93         if strides is None:
1013     94             strides = [1] * len(kernels)
1014     95         num_conv = len(channels) - 1
1015     96         if padding is True:
1016     97             num_pad = sum(kernels) - len(kernels)
1017     98         else:
1018     99             num_pad = 0
1019    100
1020    101         super().__init__()
1021    102         self.pad = nn.ZeroPad2d((num_pad//2,num_pad-num_pad//2, \
1022    103                         num_pad//2,num_pad-num_pad//2))
1023    104         self.convlist = nn.ModuleList()
1024    105
1025    106         if num_conv > 0:
1026    107             self.convlist.append(ConvBatchNormLeakyBlock(channels[0], channels[1],
1027    108                             kernel_size=kernels[0], stride=strides[0]))
1028    109         for i in range(1, num_conv - 1):
1029    110             self.convlist.append(ConvBatchNormLeakyBlock(channels[i], channels[i+1],
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079

```

```

1080          kernel_size=kernels[i], stride=strides[i]))           1134
1081      111
1082      112     if num_conv > 1:                                     1135
1083      113         self.convlist.append(nn.Conv2d(channels[-2], channels[-1], kernels[-1], strides[-1])) 1136
1084      114
1085      115     def forward(self, x):                                1137
1086      116         x = self.pad(x)                               1138
1087      117         for l in self.convlist:                         1139
1088      118             x = l(x)                                1140
1089      119         return x                                 1141
1090      120
1091      121     """                                             1142
1092      122         discriminator for conventional GAN       1143
1093      123
1094      124     class Discriminator(nn.Module):                   1144
1095      125         """
1096      126             constructor inputs:                  1145
1097      127                 1. input_size:                      1146
1098      128                     a tuple or list of input tensor size    1147
1099      129
1100      130                 2. channels:                        1148
1101      131                     a list of number of channels for each convolutional layer   1149
1102      132                     len(channels) must be equal to # of convolutional layer + 1 1150
1103      133
1104      134                 3. kernels:                          1151
1105      135                     a list of kernel-sizes for each convolutional layer        1152
1106      136                     len(kernels) must be equal to # of convolutional layer      1153
1107      137
1108      138                 4. strides:                           1154
1109      139                     a list of strides for each convolutional layer            1155
1110      140                     len(strides) must be equal to # of convolutional layer   1156
1111      141
1112      142     5. n_dense_layer:                      1157
1113      143                     a number of dense layer at the end of the network       1158
1114      144
1115      145
1116      146
1117      147     def __init__(self, input_size, channels, kernels, strides, n_dense_layer=1): 1159
1118      148         assert(len(channels)==(len(kernels)+1))                            1160
1119      149         super().__init__()
1120      150         num_conv = len(channels) - 1
1121      151
1122      152         self.convlist = nn.ModuleList()
1123      153
1124      154         if num_conv > 0:
1125      155             self.convlist.append(ConvBatchNormLeakyBlock(channels[0], channels[1], 1161
1126      156                     kernel_size=kernels[0], stride=strides[0]))
1127      157             for i in range(1, num_conv):
1128      158                 self.convlist.append(ConvBatchNormLeakyBlock(channels[i], channels[i+1], 1162
1129      159                     kernel_size=kernels[i], stride=strides[i]))
1130      160
1131      161         x = torch.randn(input_size)
1132      162         for l in self.convlist:
1133      163             x = l(x)
1134      164         self.num = x.nelement()
1135      165         self.linlist = nn.ModuleList()
1136      166         num = self.num
1137      167         for _ in range(n_dense_layer-1):
1138      168             self.linlist.append(nn.Linear(num, num//10))
1139      169             self.linlist.append(nn.LeakyReLU(0.2))
1140      170             num = num // 10
1141      171         if n_dense_layer > 0:
1142      172             self.linlist.append(nn.Linear(num, 1))
1143      173             self.sgmd = nn.Sigmoid()
1144      174
1145      175     def forward(self, x):

```

```

1188 169     for l in self.convlist:
1189 170         x = l(x)
1190 171         x = x.view(-1, self.num)
1191 172     for l in self.linlist:
1192 173         x = l(x)
1193 174     x = self.sgmd(x)
1194 175     return x
1195 176
1196 177 """
1197 178     class used to store the trained generators, reconstruct, and sample random images
1198 179 """
1199 180 class SinGAN():
1200 181     """
1201 182         constructor inputs:
1202 183             1. scale
1203 184                 scale factor for each downsampling
1204 185             2. trained_size:
1205 186                 image size of image trained for. optional
1206 187             3. G
1207 188                 list of generators, optional
1208 189             4. z_amp
1209 190                 list of standard deviation of noises for each generator
1210 191             5. Z
1211 192                 list of fixed noises for each generator for reconstruction
1212 193             6. recimg
1213 194                 list of reconstructed images for each generator
1214 195                 if G, z_amp, Z are given and this isn't,
1215 196                     it will automatically reconstruct the image for you
1216 197 """
1217 198     def __init__(self, scale, imgsize=None, G=None, z_amp=None, Z=None, recimg=None, device=None):
1218 199
1219 200         if imgsize is None:
1220 201             imgsize = []
1221 202         if G is None:
1222 203             G = []
1223 204         if z_amp is None:
1224 205             z_amp = []
1225 206         if Z is None:
1226 207             Z = []
1227 208         if recimg is None:
1228 209             recimg = []
1229 210
1230 211         if len(Z) != len(G) or \
1231 212             len(Z) != len(z_amp):
1232 213             raise Exception("G, z_amp, Z must be lists with same length")
1233 214
1234 215         self.n_scale = len(G)
1235 216
1236 217         if device is not None:
1237 218             for i in range(self.n_scale):
1238 219                 G[i] = G[i].to(device)
1239 220                 Z[i] = Z[i].to(device)
1240 221
1241 222         self.scale = scale
1242 223         self.G = G
1243 224         self.z_amp = z_amp
1244 225         self.Z = Z

```

```

1296    227     self.recimg = recimg          1350
1297    228     self.imgsize = imgsize        1351
1298    229     self.device = device         1352
1299    230
1300   231     with torch.no_grad():
1301   232         for i in range(len(recimg),self.n_scale):
1302   233             if i == 0:
1303   234                 new_recimg = self.G[0](Z[0],0.)
1304   235             else:
1305   236                 prev = utils.upsample(self.recimg[-1],1./self.scale)
1306   237                 new_recimg = self.G[i](Z[i],prev)
1307   238             self.recimg.append(new_recimg.detach())
1308   239
1309   240             if len(self.imgsize) < len(self.recimg):
1310   241                 self.imgsize = [(x.size(-2),x.size(-1)) for x in self.recimg]
1311   242
1312   243     """
1313   244     append(self, netG, z_amp, fixed_z):
1314   245         appends a generator, noise information
1315   246     1. netG
1316   247         a new generator trained at one scale above
1317   248         it is recommended to disable gradient calculation of network by requires_grad = False
1318   249     2. z_amp
1319   250         standard deviation of noise input for the generator
1320   251     3. fixed_z
1321   252         fixed noise for the generator for reconstruction
1322   253     """
1323   254     def append(self, netG, z_amp, fixed_z):
1324   255         self.G.append(netG)
1325   256         self.z_amp.append(z_amp)
1326   257         if type(fixed_z) == torch.Tensor:
1327   258             self.Z.append(fixed_z.detach())
1328   259         else:
1329   260             self.Z.append(fixed_z)
1330   261
1331   262     with torch.no_grad():
1332   263         if self.n_scale > 0:
1333   264             prev = utils.upsample(self.recimg[-1],1./self.scale)
1334   265             new_recimg = netG(fixed_z,prev)
1335   266         else:
1336   267             new_recimg = netG(fixed_z,0.)
1337   268
1338   269             self.recimg.append(new_recimg.detach())
1339   270             self.imgsize.append((new_recimg.size(-2),new_recimg.size(-1)))
1340   271             self.n_scale = self.n_scale + 1
1341   272
1342   273     return self
1343   274
1344   275     """
1345   276     reconstruct(self,scale_level=None)
1346   277         outputs a reconstructed image
1347   278     1. scale_level
1348   279         the scale level of reconstruction
1349   280         the smallest scale is 1, next scale up is 2, so forth
1350   281         if not given, it will output the reconstruction at the final scale
1351   282     """
1352   283     @torch.no_grad()
1353   284     def reconstruct(self,output_level=None):

```

```

1404    285      if self.G == []:
1405    286          return None
1406    287      if output_level is None:
1407    288          return self.recimg[-1]
1408    289      else:
1409    290          return self.recimg[output_level]
1410   291
1411   292      """
1412   293          sample(self,n_sample=1,scale_level=None)
1413   294              generates random samples
1414   295      1. n_sample
1415   296          a number of random samples
1416   297      2. scale_level
1417   298          the scale level of samples
1418   299      """
1419   300      @torch.no_grad()
1420   301      def sample(self,input_size=None, output_level=-1, n_sample=1):
1421   302          if self.G == []:
1422   303              return None
1423   304          if output_level < 0:
1424   305              output_level = self.n_scale + output_level
1425   306
1426   307          if input_size is not None:
1427   308              z = self.z_amp[0] * torch.randn(n_sample,self.Z[0].size(1),input_size[0],
1428   309                                  input_size[1],device=self.device)
1429   310          else:
1430   311              z = self.z_amp[0] * torch.randn(n_sample,self.Z[0].size(1),self.Z[0].size(2),
1431   312                                  self.Z[0].size(3),device=self.device)
1432   313          sample = self.G[0](z,0.)
1433   314          for i in range(1,output_level+1):
1434   315              sample = utils.upsample(sample, 1./self.scale)
1435   316              z = self.z_amp[i] * torch.randn_like(sample)
1436   317              sample = self.G[i](z,sample)
1437   318          return sample
1438   319
1439   320      """
1440   321          inject(self,x,n_sample=1,insert_level=None,scale_level=None)
1441   322      1. n_sample
1442   323          a number of random samples
1443   324      2. insert_level
1444   325          specifies where to inject the image
1445   326      3. scale_level
1446   327          the scale level of the output image
1447   328      """
1448   329      @torch.no_grad()
1449   330      def inject(self,x,inject_level=-1,output_level=-1,n_sample=1):
1450   331          if self.G == []:
1451   332              return None
1452   333          if n_sample != 1:
1453   334              x = torch.cat(n_sample*[x],0)
1454   335          if inject_level < 0:
1455   336              inject_level = self.n_scale + inject_level
1456   337          if output_level < 0:
1457   338              output_level = self.n_scale + output_level
1458   339
1459   340          z = self.z_amp[inject_level] * torch.randn_like(x)
1460   341          x = self.G[inject_level](z,x)
1461          for i in range(inject_level+1,output_level+1):

```

```

1512      343             x = utils.upsample(x, 1./self.scale)
1513      344             z = self.z_amp[i] * torch.randn_like(x)
1514      345             x = self.G[i](z,x)
1515      346         return x
1516      347
1517 348     def to(self,device):
1518 349         self.device = device
1519 350         for i in range(self.n_scale):
1520 351             self.G[i] = self.G[i].to(device)
1521 352             self.recimg[i] = self.recimg[i].to(device)
1522 353             if type(self.Z[i]) is torch.Tensor:
1523 354                 self.Z[i] = self.Z[i].to(device)
1524 355         return self
1525 356
1526 357     def walk(self,n,alpha,beta):
1527 358         z_t1 = 0.
1528 359         z_t = self.Z[0]
1529 360
1530 361         if n > 0:
1531 362             xs = self.G[0](z_t,0.)
1532 363             for j in range(1,self.n_scale):
1533 364                 xs = utils.upsample(xs,1./self.scale)
1534 365                 xs = self.G[j](0.,xs)
1535 366             for i in range(1,n):
1536 367                 z_diff = beta * (z_t - z_t1) + (1 - beta) * torch.randn_like(z_t)
1537 368                 z_t1 = z_t
1538 369                 z_t = alpha * self.Z[0] + (1 - alpha) * (z_t + z_diff)
1539 370                 x = self.G[0](z_t,0.)
1540 371                 for j in range(1,self.n_scale):
1541 372                     x = utils.upsample(x,1./self.scale)
1542 373                     x = self.G[j](0.,x)
1543 374                 xs = torch.cat([xs,x],0)
1544 375         return xs
1545 376
1546 377 """
1547 378     loading and saving SinGAN
1548 379 """
1549 380     def save_singan(singan,path):
1550 381         torch.save({'n_scale': singan.n_scale, \
1551 382             'scale': singan.scale, \
1552 383             'trained_size': singan.imgsize, \
1553 384             'models': singan.G, \
1554 385             'noise_amp': singan.z_amp, \
1555 386             'fixed_noise': singan.Z, \
1556 387             'reconstructed_images': singan.recimg \
1557 388             },path)
1558 389         return
1559 390
1560 391     def load_singan(path):
1561 392         load = torch.load(path)
1562 393         singan = SinGAN(load['scale'], load['trained_size'], load['models'], \
1563 394             load['noise_amp'], load['fixed_noise'],load['reconstructed_images'])
1564 395         return singan
1565 1 | from PIL import Image

```

```

1620  2 import torch
1621  3 import torchvision
1622  4 from torchvision import transforms
1623  5 from torch import nn
1624  6 import torch.nn.functional as F
1625  7 import matplotlib.pyplot as plt
1626  8 import numpy as np
1627  9 import random
1628 10 from math import ceil, floor
1629 11 import imageio
1630 12
1631 13 """
1632 14     loades image given a path to image
1633 15 """
1634 16 def load_image(path):
1635 17     img = Image.open(path)
1636 18     return img
1637 19
1638 20 """
1639 21     creates a list of images with different scales
1640 22 """
1641 23 def create_scaled_images(img,scale,min_len,max_len,match_min=True):
1642 24     scaled_imgs = []
1643 25     if type(img) == list:
1644 26         width, height = img[0].size
1645 27     else:
1646 28         width, height = img.size
1647 29
1648 30     if match_min == True:
1649 31         if width <= height:
1650 32             new_width = (int)(min_len)
1651 33             new_height = (int)(min_len / width * height)
1652 34         else:
1653 35             new_height = (int)(min_len)
1654 36             new_width = (int)(min_len / height * width)
1655 37     while new_height <= max_len and new_width <= max_len:
1656 38         if type(img) == list:
1657 39             new_imgs = []
1658 40             for i in img:
1659 41                 new_img = i.resize((new_width,new_height))
1660 42                 new_imgs.append(new_img)
1661 43             scaled_imgs.append(new_imgs)
1662 44         else:
1663 45             new_img = img.resize((new_width,new_height))
1664 46             scaled_imgs.append(new_img)
1665 47             new_width, new_height = floor(new_width / scale), floor(new_height / scale)
1666 48     else:
1667 49         if width <= height:
1668 50             new_height = max_len
1669 51             new_width = (int)(max_len / height * width)
1670 52         else:
1671 53             new_width = max_len
1672 54             new_height = (int)(max_len / width * height)
1673 55     while new_width >= min_len and new_height >= min_len:
1674 56         if type(img) == list:
1675 57             new_imgs = []
1676 58             for i in img:
1677 59                 new_img = i.resize((new_width,new_height))

```

```

1728      new_imgs.append(new_img)
1729      scaled_imgs.append(new_imgs)
1730  else:
1731      new_img = img.resize((new_width,new_height))
1732      scaled_imgs.append(new_img)
1733      new_width, new_height = ceil(new_width * scale), ceil(new_height * scale)
1734      scaled_imgs = scaled_imgs[::-1]
1735
1736  return scaled_imgs
1737
1738 """
1739 copy image to tensor
1740 """
1741 def convert_image2tensor(img,transform=None,device=None):
1742     if transform==None:
1743         transform = transforms.Compose([transforms.ToTensor(), \
1744                                         transforms.Normalize([0.5,0.5,0.5],[0.5,0.5,0.5])])
1745     tensor = transform(img)
1746     tensor = tensor.unsqueeze(0)
1747     if device != None:
1748         tensor = tensor.to(device)
1749     return tensor
1750
1751 """
1752 copy a list of images to a list of tensors
1753 """
1754 def convert_images2tensors(imgs,transform=None, device=None):
1755     if transform==None:
1756         transform = transforms.Compose([transforms.ToTensor(), \
1757                                         transforms.Normalize([0.5,0.5,0.5],[0.5,0.5,0.5])])
1758     transformed_imgs = []
1759     for img in imgs:
1760         if type(img) == list:
1761             tensor = []
1762             for i in img:
1763                 t = transform(i)
1764                 tensor.append(t)
1765             tensor = torch.stack(tensor,0)
1766         else:
1767             tensor = transform(img)
1768             tensor = tensor.unsqueeze(0)
1769             if device != None:
1770                 tensor = tensor.to(device)
1771             transformed_imgs.append(tensor)
1772     return transformed_imgs
1773
1774 """
1775 plot tensors as images
1776 """
1777 def show_tensor_image(tensor, row_n=1, use_matplot=True):
1778     tensor = tensor.detach().cpu()
1779     tensor = torchvision.utils.make_grid(tensor, row_n)
1780     img = tensor.numpy()
1781     img = img.transpose([1,2,0])
1782     img = (img + 1) / 2
1783     img = np.clip(img,0.,1.)
1784     if use_matplot is True:
1785         plt.imshow(img)

```

```

1836    118     plt.subplots_adjust(left=0.05, right=0.95, top=0.95, bottom=0.05)
1837    119     plt.show()
1838    120 else:
1839    121     img = img * 255
1840    122     img = img.astype(np.uint8)
1841    123     x = Image.fromarray(img)
1842    124     x.show()
1843    125
1844    126
1845    127 """
1846    128     parameter initialization
1847    129 """
1848    130 def xavier_uniform_weight_init(layer):
1849    131     if type(layer) == nn.Conv2d:
1850    132         torch.nn.init.xavier_uniform_(layer.weight)
1851    133         if layer.bias is not None:
1852    134             layer.bias.data.fill_(0.1)
1853    135     return
1854    136
1855    137 def xavier_normal_weight_init(layer):
1856    138     if type(layer) == nn.Conv2d:
1857    139         torch.nn.init.xavier_normal_(layer.weight)
1858    140         if layer.bias is not None:
1859    141             layer.bias.data.fill_(0.1)
1860    142     return
1861    143
1862    144 """
1863    145     turn off gradient calculation
1864    146 """
1865    147 def disable_grad(net):
1866    148     for p in net.parameters():
1867    149         p.requires_grad = False
1868    150     return
1869    151
1870    152 """
1871    153     turn on gradient calculation
1872    154 """
1873    155 def enable_grad(net):
1874    156     for p in net.parameters():
1875    157         p.requires_grad = True
1876    158     return
1877    159
1878    160 def downsample(tensor, r):
1879    161     h, w = tensor.size(-2), tensor.size(-1)
1880    162     nh, nw = ceil(h*r), ceil(w*r)
1881    163     tensor = F.interpolate(tensor, (nh, nw))
1882    164     return tensor
1883    165
1884    166 def upsample(tensor, r):
1885    167     h, w = tensor.size(-2), tensor.size(-1)
1886    168     nh, nw = floor(h*r), floor(w*r)
1887    169     tensor = F.interpolate(tensor, (nh, nw))
1888    170     return tensor
1889    171
1890    172 def save_tensor_image(tensor, path, row_n=2):
1891    173     tensor = tensor.detach().cpu()
1892    174     tensor = torchvision.utils.make_grid(tensor, row_n)
1893    175     arr = tensor.numpy()
1894    176
1895    177
1896    178
1897    179
1898    180
1899    181
1900    182
1901    183
1902    184
1903    185
1904    186
1905    187
1906    188
1907    189
1908    190
1909    191
1910    192
1911    193
1912    194
1913    195
1914    196
1915    197
1916    198
1917    199
1918    200
1919    201
1920    202
1921    203
1922    204
1923    205
1924    206
1925    207
1926    208
1927    209
1928    210
1929    211
1930    212
1931    213
1932    214
1933    215
1934    216
1935    217
1936    218
1937    219
1938    220
1939    221
1940    222
1941    223
1942    224
1943    225

```

```

1944 176     arr = arr.transpose([1,2,0])
1945 177     arr = (arr + 1) / 2
1946 178     arr = np.clip(arr,0.,1.)
1947 179     arr = arr * 255
1948 180     arr = arr.astype(np.uint8)
1949 181     img = Image.fromarray(arr)
1950 182     img.save(path)
1951 183     return
1952 184
1953 185 def save_tensor_gif(tensor, path):
1954 186     tensor = tensor.detach().cpu()
1955 187     images = tensor.numpy()
1956 188     images = images.transpose([0,2,3,1])
1957 189     images = (images + 1) / 2
1958 190     images = np.clip(images, 0., 1.)
1959 191     images = images * 255
1960 192     images = images.astype(np.uint8)
1961 193     imageio.mimsave(path, images)
1962 194     return
1963
1964     train.py
1965
1966 1 import torch
1967 2 from torch import nn
1968 3 import torch.nn.functional as F
1969 4 import matplotlib.pyplot as plt
1970 5 from . import utils
1971 6 import numpy as np
1972 7
1973 8 """
1974 9     calculates gradient penalty loss for WGAN-GP critic
1975 10 """
1976 11 def GradientPenaltyLoss(netD, real, fake):
1977 12     real = real.expand(fake.size())
1978 13     alpha = torch.rand(fake.size(0), 1, 1, 1, device=fake.device)
1979 14     alpha = alpha.expand(fake.size())
1980 15     interpolates = alpha * real + (1-alpha) * fake
1981 16     interpolates = torch.autograd.Variable(interpolates, requires_grad=True)
1982 17     Dout_interpolates = netD(interpolates)
1983 18     gradients = torch.autograd.grad(outputs=Dout_interpolates, inputs=interpolates, \
1984 19                     grad_outputs=torch.ones_like(Dout_interpolates), \
1985 20                     create_graph=True, retain_graph=True, only_inputs=True)[0]
1986 21
1987 22     grad_penalty = ((gradients.norm(2, dim=1)-1)**2).mean()
1988 23
1989 24     return grad_penalty
1990 25
1991 26 """
1992 27 trains GAN for one scale of SinGAN
1993 28 1. img
1994 29     target image
1995 30 2. netG
1996 31     generator to train
1997 32 3. netG_optim
1998 33     optimizer for generator
1999 34 4. netD
2000 35     discriminator or critic

```

```

2052 36 5. netD_optim
2053 37   optimizer for discriminator or critic
2054 38 6. singan
2055 39   SinGAN object that includes generator at lower scale
2056 40 7. num_epoch
2057 41   number of iteration
2058 42 8. mode
2059 43   specifies type of GAN
2060 44   option: 'wgangp', 'wgan', 'gan'
2061 45   gan type is 'wgangp' by default
2062 46 9. netG_lrscheduler
2063 47   learning rate scheduler for the generator, optional
2064 48 10. netD_lrscheduler
2065 49   learning rate scheduler for the critic, optional
2066 50 11. use_zero
2067 51   if True, use zero tensor for reconstruction
2068 52   however, zero will not be used when training GAN at the lowest scale
2069 53 12. batch_size
2070 54   a number of samples generated for each parameter update
2071 55 13. recloss
2072 56   a function used for calculation of reconstruction loss, nn.MSELoss is used by default
2073 57 14. recloss_scale
2074 58   scalar multiplier for reconstruction loss, 10 is used by default
2075 59 15. gp_scale
2076 60   scalar multiplier for gradient penalty loss, 0.1 is used by default
2077 61 16. clip_range
2078 62   clipping range for critic's parameter when training for WGAN
2079 63   (no clipping is done for WGAN-GP)
2080 64   0.01 by default
2081 65 17. z_std_scale
2082 66   scalar multiplier for input noise's standard deviation
2083 67 18. netG_iter
2084 68   a number of consecutive parameter updates for generator, 3 by default
2085 69 19. netD_iter
2086 70   a number of consecutive parameter updates for critic, 3 by default
2087 71 20. freq
2088 72   frequency of plotting random samples
2089 73 21. figsize
2090 74   size of the plot for generated samples
2091 75 """
2092 76 def train_singan_onescale(img, \
2093 77     netG,netG_optim, \
2094 78     netD,netD_optim, \
2095 79     singan,num_epoch, \
2096 80     mode='wgangp', \
2097 81     netG_lrscheduler=None, netD_lrscheduler=None, \
2098 82     use_zero=True,batch_size=1, \
2099 83     recloss_fun=None,recloss_scale=10,gp_scale=0.1,clip_range=0.01, \
2100 84     z_std_scale=0.1, \
2101 85     netG_iter=3,netD_iter=3, \
2102 86     log_freq=0, plot_freq=0, figsize=(15,15)) :
2103 87
2104 88     imgsize = (img.size(-2),img.size(-1))
2105 89
2106 90     if singan.n_scale == 0:
2107 91         z_std = z_std_scale
2108 92         fixed_z = z_std * torch.randn_like(img)
2109 93     else:

```

```

2160    94     prev_rec = singan.reconstruct()
2161    95     prev_rec = F.interpolate(prev_rec, imgsize)
2162    96     z_std = z_std_scale * torch.sqrt(F.mse_loss(prev_rec, img)).item()
2163    97     if use_zero:
2164    98         fixed_z = 0.
2165    99     else:
2166   100         fixed_z = z_std * torch.randn_like(img)
2167   101
2168   102     if recloss_fun is None:
2169   103         ReconstructionLoss = nn.MSELoss()
2170   104     else:
2171   105         ReconstructionLoss = recloss_fun
2172   106
2173   107     meta_data = np.zeros((num_epoch, (netG_iter+netD_iter)*2))
2174   108
2175   109     netG.train()
2176   110     netD.train()
2177   111
2178   112     for epoch in range(1, num_epoch+1):
2179   113
2180   114         utils.enable_grad(netD)
2181   115         utils.disable_grad(netG)
2182   116
2183   117         for i in range(netD_iter):
2184   118             """
2185   119             generate image
2186   120             """
2187   121             z = z_std * torch.randn(batch_size, img.size(1),
2188   122                             img.size(2), img.size(3), device=img.device)
2189   123             if singan.n_scale == 0:
2190   124                 Gout = netG(z, 0.)
2191   125             else:
2192   126                 base = singan.sample(n_sample=batch_size)
2193   127                 base = F.interpolate(base, imgsize)
2194   128                 Gout = netG(z, base)
2195   129
2196   130             """
2197   131             train critic
2198   132             """
2199   133             netD_optim.zero_grad()
2200   134             Dout_real = netD(img)
2201   135             Dout_fake = netD(Gout.detach())
2202   136             if mode == 'gan':
2203   137                 # calculate
2204   138                 real_loss = F.binary_cross_entropy(Dout_real, torch.ones_like(Dout_real))
2205   139                 real_loss.backward()
2206   140                 fake_loss = F.binary_cross_entropy(Dout_fake, torch.zeros_like(Dout_fake))
2207   141                 fake_loss.backward()
2208   142                 D_loss_total = real_loss.item() + fake_loss.item()
2209   143                 wdistance = 0.0
2210   144             elif mode == 'wgan':
2211   145                 """
2212   146                 calculate wasserstein distance
2213   147                 """
2214   148                 wdistance_loss = Dout_fake.mean() - Dout_real.mean()
2215   149                 wdistance_loss.backward()
2216   150                 D_loss_total = wdistance_loss.item()
2217   151

```

```

2268      152
2269      153     wdistance = wdistance_loss.item()
2270      154     elif mode == 'wgangp':
2271      155     """
2272      156         calculate wasserstein distance and gradient penalty
2273      157     """
2274      158     wdistance_loss = Dout_fake.mean() - Dout_real.mean()
2275      159     wdistance_loss.backward()
2276      160     grad_penalty_loss = GradientPenaltyLoss(netD, img, Gout) * gp_scale
2277      161     grad_penalty_loss.backward()
2278      162     D_loss_total = wdistance_loss.item() + grad_penalty_loss.item()
2279      163     wdistance = wdistance_loss.item()
2280      164
2281      165     netD_optim.step()
2282      166     meta_data[epoch-1,i] = D_loss_total
2283      167     meta_data[epoch-1,i+netD_iter] = wdistance
2284      168
2285      169     if mode == 'wgan':
2286      170         for p in netD.parameters():
2287      171             p.data.clamp_(-clip_range, clip_range)
2288      172
2289      173     utils.disable_grad(netD)
2290      174     utils.enable_grad(netG)
2291      175
2292      176     for j in range(netG_iter):
2293      177
2294      178     """
2295      179         generate image
2296      180     """
2297      181     if singan.n_scale == 0:
2298      182         Gout = netG(z, 0.)
2299      183     else:
2300      184         base = singan.sample(n_sample=batch_size)
2301      185         base = F.interpolate(base, imgsize)
2302      186         Gout = netG(z, base)
2303      187
2304      188     """
2305      189         train generator
2306      190     """
2307      191     netG_optim.zero_grad()
2308      192     Dout_fake = netD(Gout)
2309      193     if mode == 'gan':
2310      194         adv_loss = F.binary_cross_entropy(Dout_fake, torch.ones_like(Dout_fake))
2311      195         adv_loss.backward()
2312      196     elif mode == 'wgan' or mode == 'wgangp':
2313      197         adv_loss = - Dout_fake.mean()
2314      198         adv_loss.backward()
2315      199     if singan.n_scale == 0:
2316      200         rec = netG(fixed_z, 0.)
2317      201     else:
2318      202         rec = netG(fixed_z, prev_rec)
2319      203     rec_loss = ReconstructionLoss(rec, img) * recloss_scale
2320      204     rec_loss.backward()
2321      205     G_loss_total = adv_loss.item() + rec_loss.item()
2322      206
2323      207     netG_optim.step()
2324      208
2325      209     meta_data[epoch-1, netD_iter*2+j] = G_loss_total

```

```

2376    210     meta_data[epoch-1,netD_iter*2+netG_iter+j] = rec_loss      2430
2377    211
2378    212     if netD_lrscheduler is not None:                          2431
2379    213         netD_lrscheduler.step()                                2432
2380    214     if netG_lrscheduler is not None:                          2433
2381    215         netG_lrscheduler.step()                                2434
2382    216
2383    217     if (log_freq != 0) and (epoch % log_freq == 0):           2435
2384    218         print("[{:d}/{:d}]\n".format(epoch,num_epoch))        2436
2385    219         print(" generator loss : {:.3f}\n".format(G_loss_total)) 2437
2386    220         print(" reconstruction loss: {:.3f}\n".format(rec_loss)) 2438
2387    221         if mode == 'gan':
2388    222             print(" discriminator loss : {:.3f}\n".format(D_loss_total)) 2439
2389    223         elif mode == 'wgan' or mode == 'wgangp':
2390    224             print(" critic loss : {:.3f}\n".format(D_loss_total)) 2440
2391    225             print("wasserstein distance: {:.3f}\n".format(wdistance)) 2441
2392    226
2393    227     if (plot_freq != 0) and (epoch % plot_freq == 0):           2442
2394    228         netG.eval()
2395    229         with torch.no_grad():
2396    230             '''
2397    231                 display sample from generator
2398    232             '''
2399    233             if singan.n_scale == 0:
2400    234                 z = z_std * torch.randn(7,img.size(1),img.size(2),      2443
2401    235                                     img.size(3),device=img.device)
2402    236                 sample = netG(z,0.)
2403    237                 rec = netG(fixed_z,0.)
2404    238             else:
2405    239                 z = z_std * torch.randn(7,img.size(1),img.size(2),      2444
2406    240                                     img.size(3),device=img.device)
2407    241                 base = singan.sample(n_sample=7)
2408    242                 base = F.interpolate(base,imgsize)
2409    243                 sample = netG(z,base)
2410    244                 rec = netG(fixed_z,prev_rec)
2411    245
2412    246                 sample = torch.cat([sample,rec],0)
2413    247
2414    248                 plt.figure(figsize=figsize)
2415    249                 utils.show_tensor_image(sample,4)
2416    250                 netG.train()
2417    251
2418    252             return z_std, fixed_z, meta_data
2419    253
2420    254
2421    255 def train(r, scaled_img_list, device=None):
2422    256
2423    257     singan = models.SinGAN(r)
2424    258
2425    259     n_scale = len(scaled_img_list)
2426    260     tensor_list = utils.convert_images2tensors(scaled_img_list)
2427    261     nc = 8
2428    262
2429    263
2430    264     for i in range(num_scales):
2431    265
2432    266         img = tensor_list[i]
2433    267         netG = models.AddSkipGenerator([3,nc,nc,nc,3],[3,3,3,3])
```

```

2484    268     if device is not None:
2485    269         netG = netG.to(device)
2486    270     netG.apply(xavier_uniform_weight_init)
2487    271     netD = models.ConvCritic([3,nc,nc,nc,1],[3,3,3,3])
2488    272     if device is not None:
2489    273         netD = netD.to(device)
2490    274     netD.apply(xavier_uniform_weight_init)
2491    275     netG_optim = torch.optim.Adam(netG.parameters(),lr=5e-4,betas=(0.5,0.999))
2492    276     netD_optim = torch.optim.Adam(netD.parameters(),lr=5e-4,betas=(0.5,0.999))
2493    277     netG_lrscheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer=netG_optim,
2494    278                                     milestones=[1600],gamma=0.1)
2495    279     netD_lrscheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer=netD_optim,
2496    280                                     milestones=[1600],gamma=0.1)
2497    281
2498    282     print("Training Scale {}".format(i+1))
2499    283
2500    284     z_std, fixed_z, _ = train_singan_onescale(img, \
2501    285             netG,netG_optim, \
2502    286             netD,netD_optim, \
2503    287             netG_chain,2000, \
2504    288             mode='wgangp', \
2505    289             netG_lrscheduler=netG_lrscheduler, \
2506    290             netD_lrscheduler=netD_lrscheduler, \
2507    291             use_zero=True,batch_size=1, \
2508    292             recloss_fun=recloss_fun, \
2509    293             recloss_scale=10, gp_scale=0.1, \
2510    294             clip_range=0.01, \
2511    295             z_std_scale=0.1, \
2512    296             netG_iter=3,netD_iter=3, \
2513    297             log_freq=50, plot_freq=0, \
2514    298             figsize=(15,15))
2515    299
2516    300     netG.eval()
2517    301     utils.disable_grad(netG)
2518    302     singan.append(netG,z_std,fixed_z)
2519    303
2520    304     nc = (int) (num_channel / r)
2521
2522    305     return singan
2523
2524
2525    1 import torch
2526    2 from . import models, utils, train
2527    3 import argparse
2528    4
2529    5 img = loadImage("SinGAN_Implementation/3-nature-wallpaper-mountain.jpg")
2530    6 print(img.size)
2531    7 plt.imshow(img)
2532    8 plt.show()
2533    9
2534   10
2535   11 parser = argparse.ArgumentParser()
2536   12 parser.add_argument('--image', required=True, help="input image")
2537   13 parser.add_argument('--r', default=0.75, help="scale of downsampling")
2538   14 parser.add_argument('--min_len', default=12, help="minimum height or width of image")
2539   15 parser.add_argument('--max_len', default=250, help="maximum height or width of image")

```

```
2592 16 parser.add_argument('--match_min', default=True, help="match the minimum length") 2646
2593 17 arg = parser.parse_args() 2647
2594 18 img = load_image(args.images) 2648
2595 19 scaled_img_list = utils.create_scaled_images(img,arg.r,arg.min_len,arg.max_len, 2649
2596 20 match_min=arg.match_min) 2650
2597 21 print("{} scales created".format(len(scaled_img_list))) 2651
2598 22 2652
2599 23 2653
2600 24 if torch.cuda.is_available() is True: 2654
2601 25     print("Using CUDA") 2655
2602 26     device = torch.device('cuda:0') 2656
2603 27 else: 2657
2604 28     device = None 2658
2605 29 2659
2606 30 print("training starts...") 2660
2607 31 singan = train.train(arg.r, scaled_img_list, device=device) 2661
2608 32 2662
2609 33 2663
2610 34 cpu = torch.device('cpu') 2664
2611 35 singan = singan.to(cpu) 2665
2612 36 models.save_singan(singan,'singan.pth') 2666
2613 37 2667
2614 38 2668
2615 39 2669
2616 40 2670
2617 41 2671
2618 42 2672
2619 43 2673
2620 44 2674
2621 45 2675
2622 46 2676
2623 47 2677
2624 48 2678
2625 49 2679
2626 50 2680
2627 51 2681
2628 52 2682
2629 53 2683
2630 54 2684
2631 55 2685
2632 56 2686
2633 57 2687
2634 58 2688
2635 59 2689
2636 60 2690
2637 61 2691
2638 62 2692
2639 63 2693
2640 64 2694
2641 65 2695
2642 66 2696
2643 67 2697
2644 68 2698
2645 69 2699
```