Image Colorization Using Conditional Generative Adversarial Networks

Phanitta Chomsinsap   Zhiwei Zhang

Abstract

Generative adversarial networks recently attract computer vision society with their effectiveness in generating realistic images. In this project, we use conditional GAN network to color greyscale images with high quality (2048x4096). Our results after 100 training epochs are able to color some landscape images with reasonable reality. Additionally, we were able to reproduce much higher resolution colored images (higher than 12MP) by stitching together smaller image patches.

1. Introduction

1.1. Conditional Generative Adversarial Networks

Generative adversarial network is designed as a zero-sum-game. There are two players in the game: the generator and the discriminator, and they are both neural networks. The generator wants to generate a probability distribution p_g that is the same as the data set p_x. In doing this, the generator builds a mapping function from a prior noise distribution $p_z(z)$ to data space as $G(z;\theta_g)$. On the contrary, the discriminator tries to label whether an input is real or generated. The value function of this zero-sum-game is

$$\min_G \max_D V(G,D) = \mathbf{E}_x \log D(x) + \mathbf{E}_z \log(1 - D(G(z))) \tag{1}$$

Conditional generative adversarial network is an extension model of GAN, where both the generator and the discriminator are conditioned on an extra information y. In the case of image colorization, this piece of information is the greyscale image which we would like to color. Thus the modified value function is

$$\min_G \max_D V(G,D) = \mathbf{E}_x \log D(x|y) + \mathbf{E}_z \log(1 - D(G(z|y))) \tag{2}$$

1.2. Design of Cost Function

Besides considering cGAN cost function alone, it is beneficial to include a commonly used cost function. Losses such as L_1 and L_2 norm are used to help enforce low-frequency correctness of the input. pix2pix utilizes L_1 norm rather than L_2 norm, as experiments showed that L_1 norm gives less blurry output images [1]. Thus the value function is a tradeoff between cGAN loss and L_1 loss,

$$\begin{aligned} &\min_G \max_D V(G,D) \\ &= \mathbf{E}_x \log D(x|y) + \mathbf{E}_z \log(1 - D(G(z|y))) \\ &+ \lambda \mathbf{E}_{x,y,z} ||y - G(x,y)||_1 \end{aligned} \tag{3}$$

2. cGAN Architecture

2.1. Generator Architecture

The generator receives a grayscale image as an input and returns a colored version of the same image (Fig. 1).[1] The generator network consists of a series of encoders and decoders implemented using stride 2 convolution for downsampling and transposed convolution for upsampling respectively. Downsampling is necessary for obtaining a high-level representation of the input image or large receptive field. Upsampling reverses this operation to output a high-resolution colored image. Each weight layer (either convolution or transposed convolution) is followed by a batch normalization and an activation function. Leaky ReLu is used for the encoder as an attempt to fix the "dying ReLu" problem because a leaky ReLu has a positive slope where $x < 0$ so that the gradient will not be saturated. ReLu activation function is used for decoder layers.

Skip connections known as "U-Nets" are added between mirrored layers in the encoder and decoder layers where the dimensions of the feature map are the same. Skip connections are necessary for such a deep network to prevent the problem of vanishing gradient, in which the values of the weight gradient become very small or close to zero in the beginning layers. Since backpropagation is merely an application of chain rule, if a gradient is less than zero, the product of that will become smaller, not to mention that when multiplied by a small learning rate (2×10^{-4} as our initial learning rate for Adam Optimizer) the total gradient value will become even smaller. Another function of the skip connections is to allow low-level information flow between the encoders and decoders such as the location of edges [1].

[1]The input grayscale image is of size 2048×4096x×3 instead of 2048×4096× because the input and target training image is concatenated together when fed into the system so their dimensions must match (see datasets for more information).

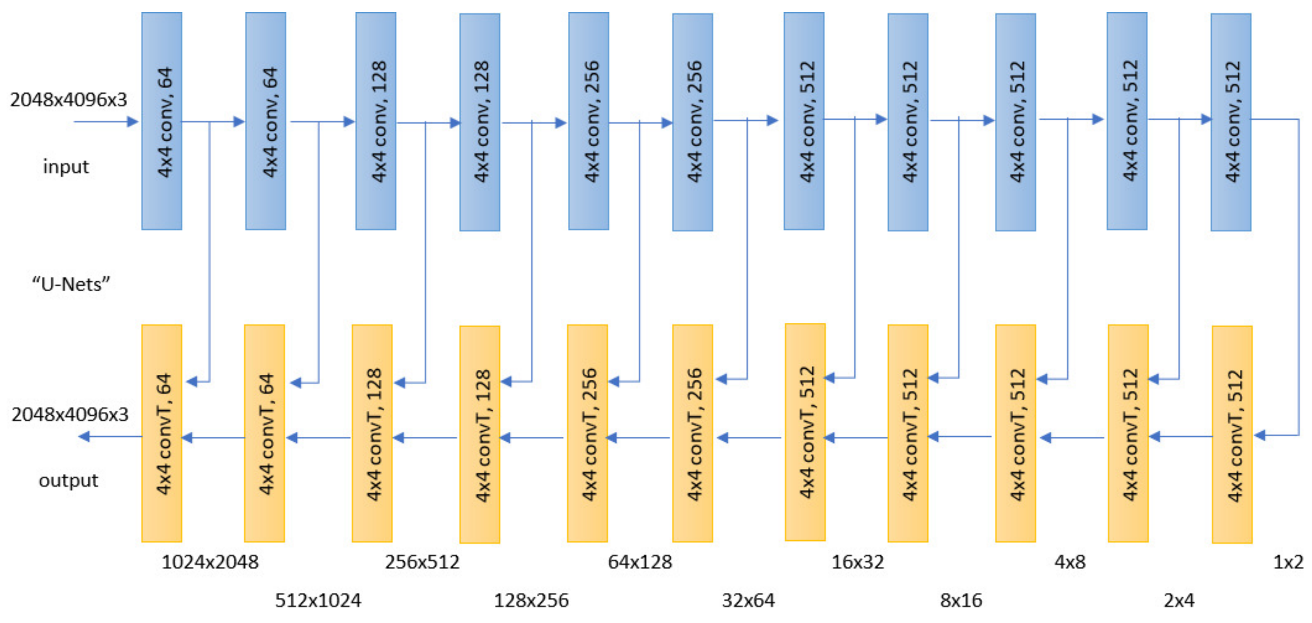


Figure 1: Generator Architecture

Width is defined to be the number of feature maps at each layer. The larger width architecture is inspired from VGG16 network that shows that classification accuracy can be improved when the number of feature maps increase with depth. VGG16 was a model used to classify the ImageNet dataset with 1000 different categories, and has 138M parameters. Similarly, starting with the width of 64, the size of the width is increased to 128, 256, and lastly 512 as the network goes deeper. The size of the width stops at 512 to prevent the total number of trainable parameters from going too high.

2.2. Discriminator Architecture

The discriminator takes two inputs: a grayscale image and a colored image (either from the groundtruth or a generator-generated image) (Fig. 2). The two input images are concatenated and then downsampled until M$\times$N dimension is reached, where M is 30 and N is 62 in our case. This process is known as PatchGAN, where each pixel in the output can be traced back to its receptive field (called patch) in the input image. In contrast to PatchGAN, a regular GAN discriminator consists of a single scalar output that can be mapped back to the entire input image. A regular GAN discriminator can be obtained by downsampling the feature map to 1x1 at the output. The benefit of having PatchGAN is that it allows us to penalize incorrect guesses at the scale of image patch instead of the entire image.

Similar to the encoder layer in the generator, each convolutional layer in the discriminator is followed by a batch norm and a leaky ReLu, except for the last layer where a sigmoid is used to map a prediction or "guess" to either 0 or 1, which signifies an unknown image as "fake" or "real" respectively.

3. Experiment

The tensorflow code for pix2pix based on Isola et al. [1] was taken from GitHub [2], and was modified accordingly to meet our project's purpose. The implementation for image stitching was done entirely in Matlab.

3.1. Dataset

It is well-known that the discriminator and the generator should be trained simultaneously. For this project, we trained our cGAN using 459 pairs of greycale and colored images of mixed landscape (ocean, lake, mountain, rock, desert, forest, Fig. 3) with 2048$\times$4096 resolution (Copyright Zhiwei Zhang). 20 test images were taken from different online sources to ensure the diversity of the set.

3.2. Results: Non-stitched Images

In non-stitching image colorization, we fed the network with 2048$\times$4096 images, and the output is also 2048$\times$4096. Figure 4 gives some outputs of the cGAN. The images look realistic. Figure 5 gives an example of a failure case where the output of cGAN looks unrealistic for a human inspector. In this failure case, the cGAN was confused between ocean and sand.

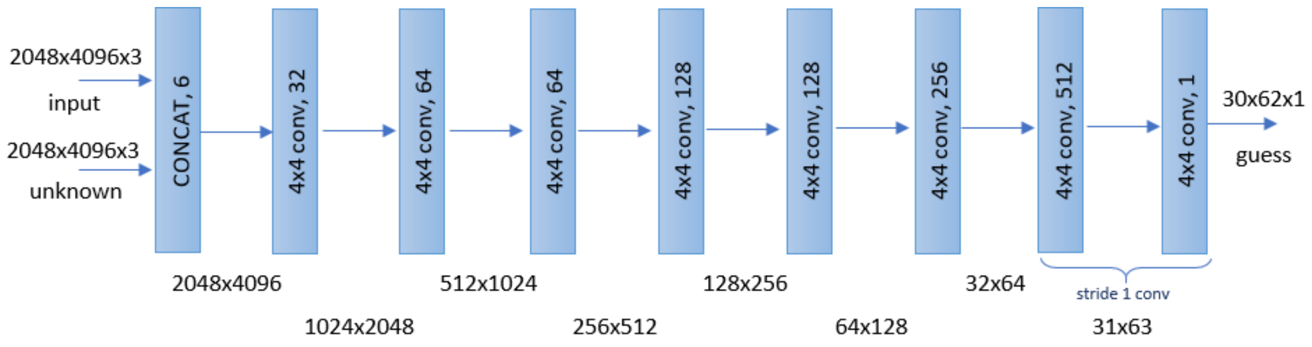


Figure 2: Discriminator Architecture

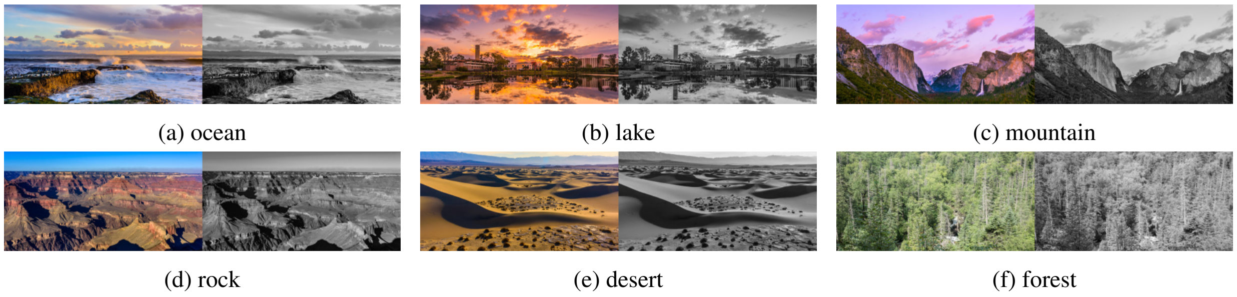


Figure 3: Examples of Training Images

3.3. Results: Stitched Images

In nowadays digital photography, the size of images are large, and image sizes are different. Thus in order to apply our results to larger images, we apply the strategy of stitching small image patches together to form larger images. For overlapping regions, we average all patches. Figure 6 gives results of the stitched images with 256 and 32 stitching strides. These images are 2048×5120. The results show that larger stride has more obvious transition between patches as compared with smaller stride.

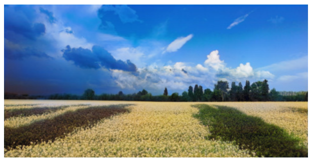
4. Conclusion

In this project, we use cGAN to color greyscale images. We use 459 pairs of 2048×4096 images as a training set, and 20 images as a testing set. Results showed that this cGAN is able to color greyscale images with reasonable reality. To further expand its application, we implemented stitch-and-average strategy to color images with larger sizes.

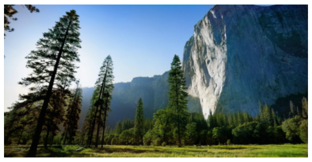
References

[1] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. *arXiv preprint*, 2017. 1, 2

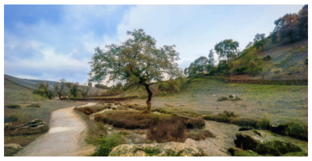
[2] Github pix2pix-tensorflow. `https://github.com/affinelayer/pix2pix-tensorflow`. Accessed: 2018-06-11. 2

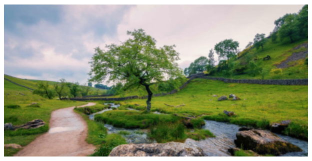

(a) output

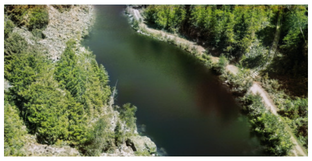
(b) target


(c) output

(d) target


(e) output


(f) target


(g) output

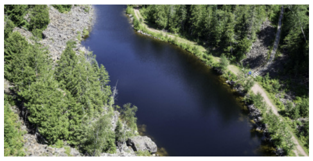
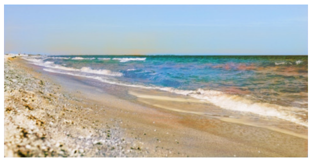

(h) target

Figure 4: Successful Cases of Non-stitched Images


(a) output

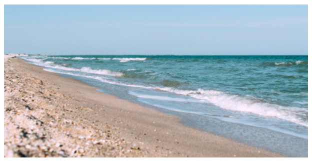
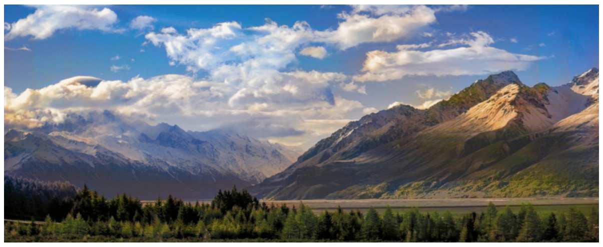

(b) target

Figure 5: Failure Case of Non-stitched Images


(a) output with stride 256

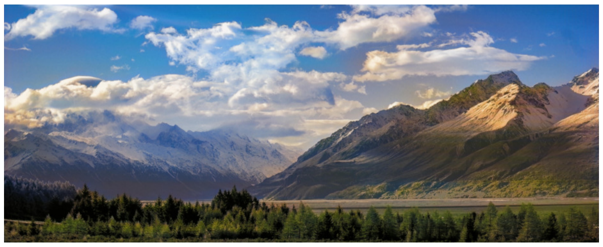

(b) output with stride 32

Figure 6: Stitched Images with Different Stride