Paper Reproduction - "Image-to-Image Translation with Conditional Adversarial Networks"

CS4240 2019/20 Q3 Group 56: Yingfu Xu Liang Zeng Hao Liu

1. Introduction

Many problems in image processing, computer graphics and computer vision can be regarded as a problem that translate an input image to a new image with different style or format. Generative Adversarial Networks(GAN) is a of paramount method to implement the image translating. However, how effective image-conditional GANs can be as a general-purpose solition for image-to-image translation remains unclear. In this case, conditional GANs has been proposed in this paper [1] (https://arxiv.org/pdf/1611.07004.pdf) which is used to explore the effectness of image-condition GANs. And in this paper, it achives decent results on a wide variety of applications('Labels to Street Scene', 'Labels to Facade', 'BW to Color', 'Aerial to Map', 'Day to Night' and 'Edges to Photo'). For simplicity of expression, the method in [1] is called as "pix2pix" in the rest part of this report.

Inspired by the decent results that [1] achieves and many other applications in image translation, we would like to choose this paper as our reproduction project to gain a deeper insight. We worked on the following three criteria:

- New code variant (Rewrite existing code to be more efficient/readable)
- New data (Evaluating different datasets to obtain similar result)
- Hyperparameter check (Evaluating sensitivity to hyperparameters)

In this blog, we will walk through the following items:

- New Code Variant Description: Illustration of the significantly changed part of the existing open-sourced code.
- New Datasets: Description of the three new datasets we chose and the reason why we chose them.
- Results of Image Translation on New Datasets: The results of the pix2pix with default parameters on 3 new datasets.
- Comparison of Different Network Architectures and Learning Rates: The results of 2 generator network architectures and 3 learning rates on 2 new datasets.
- References
- Appendix: The simplified code in Jupyter Notebook.

2. New Code Variant Description

2.1 Code Variant from Open-Soursed Code

First, we ran and studied the sophisticated and organized orginal code (https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix) of pix2pix (https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix (https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix)). Because this project cleverly assembled both pix2pix and CycleGAN [2], it contains unrelated content with pix2pix, especially its sophisticated data loading and visualization part. Then, to speed up our studying progress, we turned to another open-source pix2pix implementation (https://github.com/mrzhu-cool/pix2pix-pytorch (https://github.com/mrzhu-cool/pix2pix-pytorch)). It is a simple implementation of pix2pix so easier to understand. Inspired by this simplified version, we combined some parts of it with some code from the original code to rewrite our code in both Jupyter Notebook (Appendix) and the multi-file Python project (https://github.com/YingfuXu/pix2pixCourseProject (https://github.com/YingfuXu/pix2pixCourseProject)). Note that the multi-file Python project (https://github.com/YingfuXu/pix2pixCourseProject) has more functions in data loading and result visualization to be trained on a remote server and performance analyzation.

Our code focuses on improvement in Readability and Efficiency.

- Readability: The orginal code (https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix) mixes "pix2pix" model and "CycleGAN" model. It is quite difficult for beginnners like us to follow its logic. For example, the `Resnet` architecture is redundant in the "pix2pix" model. In this case, inspired by the simple implementation (https://github.com/mrzhu-cool/pix2pix-pytorch), we picked up all "pix2pix" code and rewrite the code into modularity (dataloader, libraries, parameters, networks, train, test, plot)to have a clear logic.
- Efficiency: In the original code (https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix), the dataset format is troublesome since the dataset we found are usually separate pictures and the picture size is unique. In this case, much time was spent on changing picture size and combine pictures by `combine_A_and_B.py`. Inspired by the simple implementation (https://github.com/mrzhu-cool/pix2pix-pytorch), we change the data format as following. We do not need to change the image size and preprocess the dataset using `combine_A_and_B.py` as the original code (https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix).

```
1st class folder:       - facades
2nd class folder:    - test  - train
3rd class folder:       - a     - a
3rd class folder:       - b     - b
```

In the Appendix, the detailed explanation of the code is inllustrated.

2.2 Diffenrences between Code and Paper

Besides, we noticed that there are several differences between the original code (https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix) and the description in [1].

In [1], it says: “after the last layer, a convolution is applied to map to a 1-dimensional output, followed by a Sigmoid function.” But in the comments of the original code (https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix) it says: “Do not use sigmoid as the last layer of Discriminator. LSGAN needs no sigmoid.” Then it calls torch.nn.MSELoss() to calculate the least square loss.

For Loss GAN (the following Equation), CycleGAN [2] replaces the negative log likelihood objective by a least-squares loss. This loss is more stable during training and generates higher quality results. Besides, in [1], the authors pointed out that previous approaches have found it beneficial to mix the GAN objective with a more traditional loss, such as L2 distance.

$$\begin{aligned}\mathcal{L}_{\text{GAN}}\left(G, D_Y, X, Y\right) = & \mathbb{E}_{y \sim p_{\text{data}}(y)}\left[\log D_Y(y)\right] \\ & + \mathbb{E}_{x \sim p_{\text{data}}(x)}\left[\log(1 - D_Y(G(x))\right]\end{aligned}$$

In pix2pix, the 70 × 70 discriminator architecture is: C64-C128-C256-C512. We tried to find out how to set the size of the patch but never found the number 70 in code. Why it is called as 70 × 70 discriminator has been explained well in https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix/issues/39 (https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix/issues/39). Here we cite a paragraph to summarize it.

The difference between a PatchGAN and regular GAN discriminator is that rather the regular GAN maps from all the pixels in the image to a single scalar output, which signifies "real" or "fake", whereas the PatchGAN maps from all the pixels in the image to an NxN array of outputs X, where each X_ij signifies whether the patch ij in the image is real or fake. Which is patch ij in the input? Well, output X_ij is just a neuron in a convnet, and we can trace back its ***receptive field*** *to see which input pixels it is sensitive to. In the CycleGAN architecture, the receptive fields of the C64-C128-C256-C512 discriminator turn out to be 70x70 patches in the input image.*

3. New Datasets

pix2pix model has achieved impressive results in many image-to-image translation applications. In order to further explored the ability of pix2pix, we have designed 3 new application scenarios including the recovery of blurry images (deblurring), recovery RGB images from depth images, and Chinese calligraphy transformation. 3 new datasets were created for these applications respectively. In this section, the dataset creation process is introduced.

3.1 Blurry Image Dataset

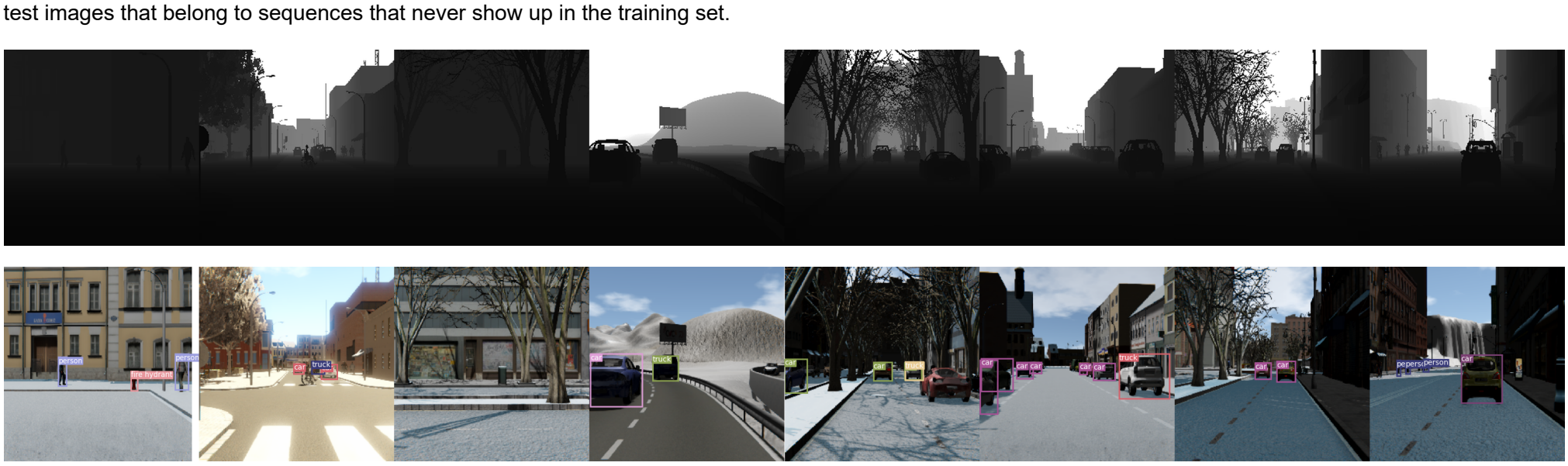
It is difficult to collect paired blurry and clear images so we had to take advantage of the existing image dataset. We applied multiple blurring to the images from PASCAL Visual Object Classes Challenge 2007, which consists of 2,501 training images, 2,510 validation images and 4,952 test images regarding 20 common objects. For each image in the original dataset, 9 blurry images were created by gaussian blur, motion blur, and disk blur with random filter size and intensity as the example below. The new images are paired with the original one. The original partition is maintained. Thus, the new dataset has 22509 training images and 22,590 test images.

Blur		
Gaussian	size in $[3, 15]$	std in $[1, 5]$
Motion	length in $[8, 20]$	angle in $[0, 360]$
Disk	radius in $[2, 10]$	

3.2 Grayscale Depth Image Dataset

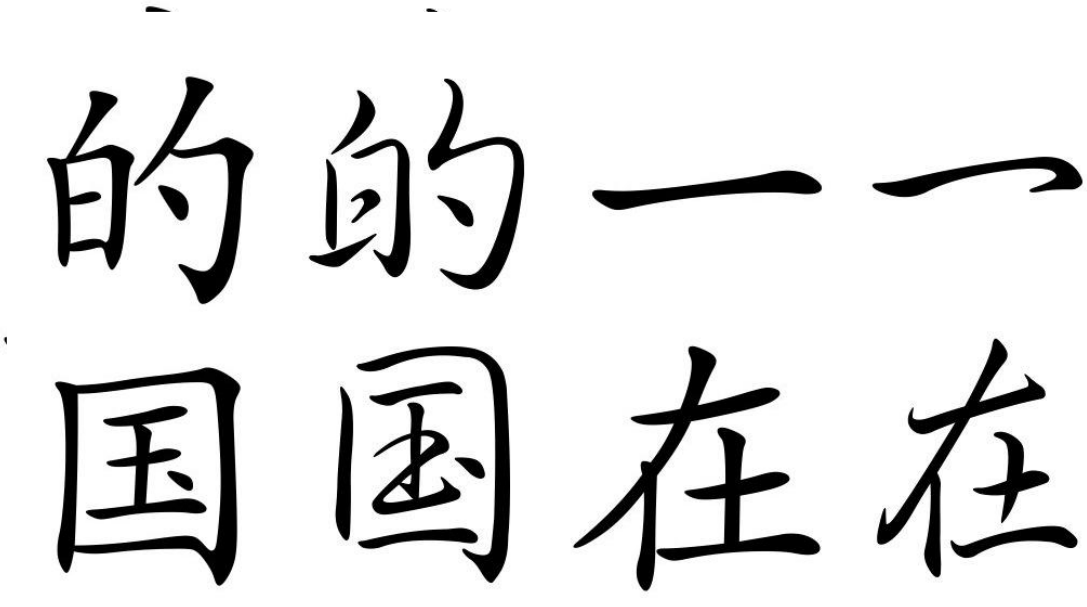
The depth grayscale images and RGB scene images are sampled from the SYNTHIA-AL (http://synthia-dataset.net/downloads/) dataset. The cGAN is trained to generate the RGB scene image from the input grayscale depth image. SYNTHIA is a dataset that has been generated with the purpose of aiding semantic segmentation and related scene understanding problems in the context of driving scenarios. SYNTHIA consists of a collection of photo-realistic frames rendered from a virtual city and comes with precise pixel-level semantic annotations. The accurate depth of frames is also provided by the dataset. Depth is encoded in the 3 channels using the following formula: Depth = 5000 x (R + G x 256 + B x 256 x 256) / (256 x 256 x 256 - 1). We process the RGB images where depth is encoded, to get grayscale images with 1 channel. The intensity of each pixel is determined by the depth in meters. So the maximum depth encoded in grayscale images is 255 meters.

We choose 3,400 image pairs as the training set and 1,000 pairs as the testing set. Note that all these mentioned 4,400 images belong to the same sequences in the dataset, which means the scenes in the test images also appear in the training images. To further show the ability of pix2pix on new scenes, we select 3,300 test images that belong to sequences that never show up in the training set.



3.3 Chinese Calligraphy Image Dataset

The Chinese calligraphy dataset is for training a model to transform characters of printing style to calligraphy. The images were generated from calligraphy fonts and those images of the same characters were made into pairs. The dataset consists of 300 paired images for training, 100 for validation, and 100 for test.


的的一一
国国在在

4. Results of Image Translation on New Datasets

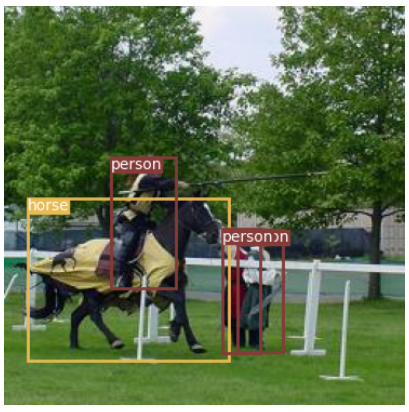
In this section, the generated images by pix2pix from the new datasets are shown. We list the images of different epochs in training to show how the generator network does a better and better job. Inspired by DeblurGAN [3] that uses GAN for motion deblurring, to quantitatively assess the performance of the generator, we utilize peak signal-to-noise ratio (PSNR) and structural similarity (SSIM). Both values are calculated by comparing the generated images and the target images. The target images are sharp images and RGB images respectively in deblurring and recovering RGB images from depth. As PSNR and SSIM measure the similarity between generated image and target image, this similarity is not very suitable for judging whether the image is 'real' or not. We further use YOLOv3 (https://github.com/eriklindernoren/PyTorch-YOLOv3) as an auxiliary way to tell whether the generated images are real enough for the state-of-the-art object detection algorithm.

4.1 Image Deblurring

In the following figures, the deblurring results of the default pix2pix project on a small test set are shown. This small test set for visualization (270 images) is made up of 9 different kinds of artificial blur of the same images. This small visualization set is also used for calculating the changing trend of PSNR and SSIM in different training epochs.

The size of our training set is 22,509. We trained the network using a single GTX1080ti GPU with 11GB VRAM, Core i7-7700 and 16GB RAM for 20 epochs. Each epoch took around 1,264 seconds. The default pix2pix uses U-Net [4] as the architecture of the generator network. The learning rate is 0.0002 and decay linearly to zero in 10 epochs after it holds 0.0002 for 10 epochs.

Original sharp image:



Blurry images (the first three images from left have gaussian blur, the three images in the middle have motion blur, the three images on the right have disk blur):

Results after 1 epoch of training:

Results after 5 epochs of training:

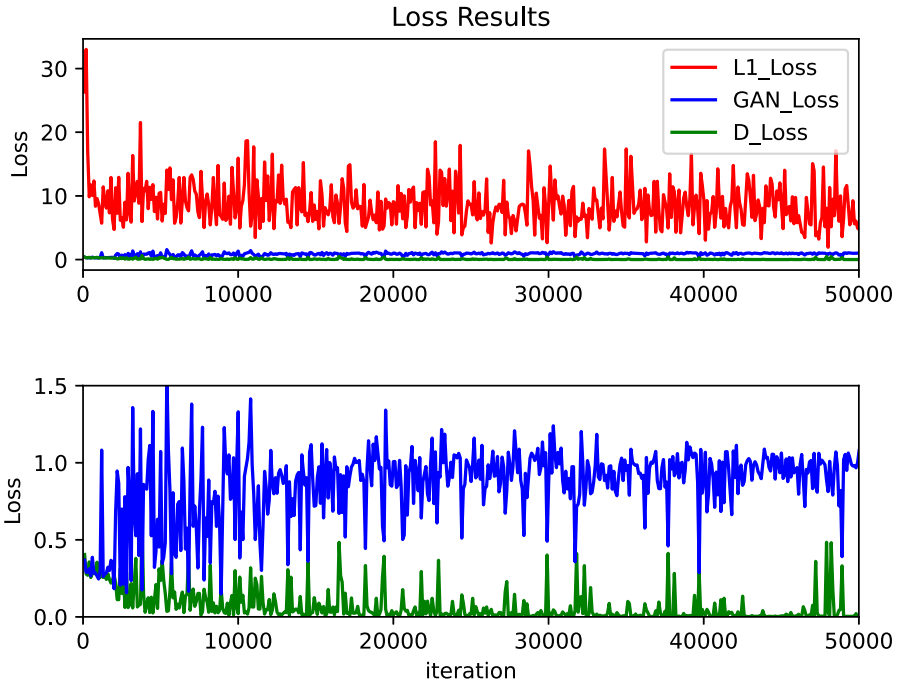
Results after 10 epochs of training:

Results after 15 epochs of training:

Results after 20 epochs of training:

From the above images, we noticed that the generated images look sharper and sharper. So we are sure that the network is doing what we want it to do. But especially for motion deblurring, the output looks more and more unrealistic with strange wrinkle-like texture. YOLOv3 detects fewer objects when the training finished. We guess the reason is that different types of blurry images have different distributions. But we train a single network for 3 different types of blur. This network reaches better results in Gaussian blur and disk blur than motion blur.

The following figure shows how the items of loss changes in the training. We show the first 50,000 iterations in 450,180. The trend of loss looks similar in later iterations.



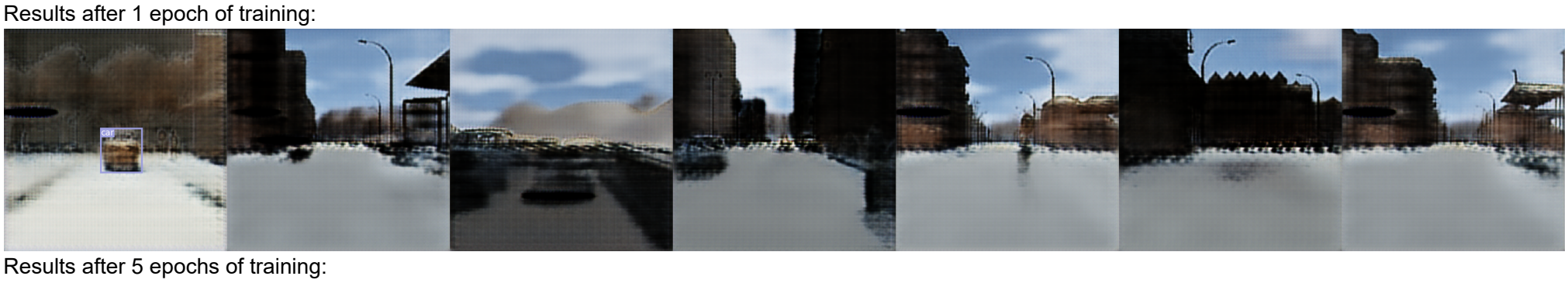
Here we notice very noisy loss curves. It is easy to understand why L1 loss goes down rapidly and fluctuates. Because the network was always feeding with different images. But the GAN loss keeps going up and down and does not show a converging trend. The orginal code (https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix) says: "For least square GAN objective, it is quite normal for the G and D losses to go up and down. It should be fine as long as they do not blow up." Further, [5] shows that in the more realistic case of distributions that are not absolutely continuous, unregularized GAN training is not always convergent.

4.2 Generating RGB Images from Graysacle Depth Images

In the following figures, the generated RGB images of different epochs of training are shown. These images are from a small test dataset (100 images) for visualization. The scene of the simulated city in each image is different from each other. These 7 kinds of scenes are included in the training set. This small visualization set is also used for calculating the changing trend of PSNR and SSIM in different training epochs.

The size of training set is 3,300. We trained the network using a single GTX1080ti GPU with 11GB VRAM, Core i7-7700 and 16GB RAM for 160 epochs. Each epoch took around 440 seconds. The default pix2pix uses U-Net [4] as the architecture of the generator network. The learning rate is 0.0002 and decay linearly to zero in 80 epochs after it holds 0.0002 for 80 epochs.

Results after 1 epoch of training:



Results after 5 epochs of training:

Results after 10 epochs of training:

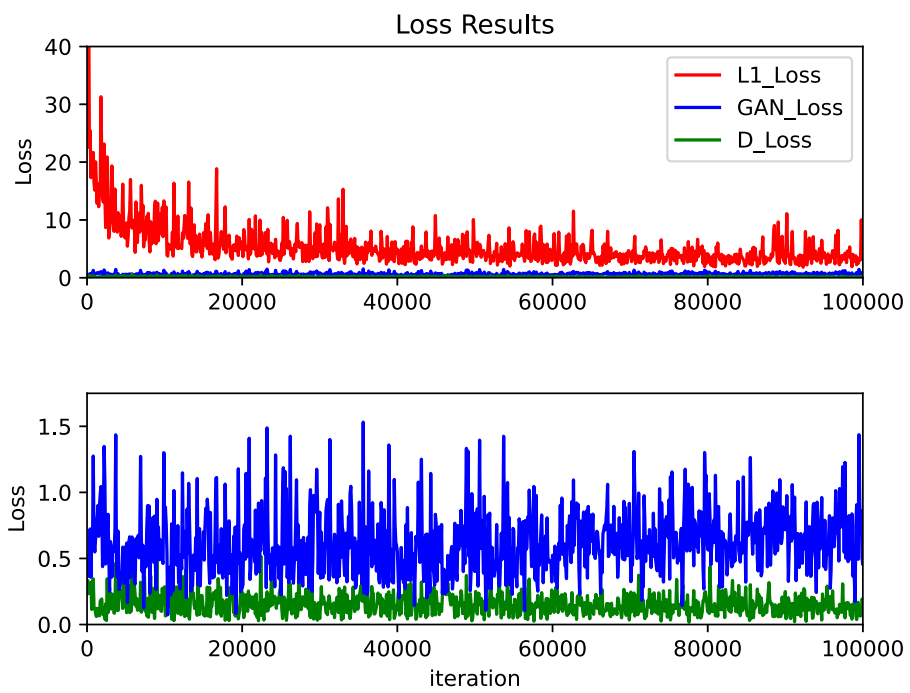
Results after 15 epochs of training:

Results after 20 epochs of training:

Results after 160 epochs of training:

From the above images, we can see that the generated images have more and more details as the training goes on. Actually, from the grayscale depth image, the network can only get the shape of the contour of a tree, or a car. The reason why the network gives the right color and texture for objects is that the network has seen these objects before in the training set. What will happen if the network is used to generate something it has never seen before, like a tree without leaf? We will discuss it in the next section "Comparison on Different Network Architectures and Learning Rate".

The following figure shows how the items of loss changes in the training. We show the first 100,000 iterations in 528,000. The trend of loss looks similar in later iterations.



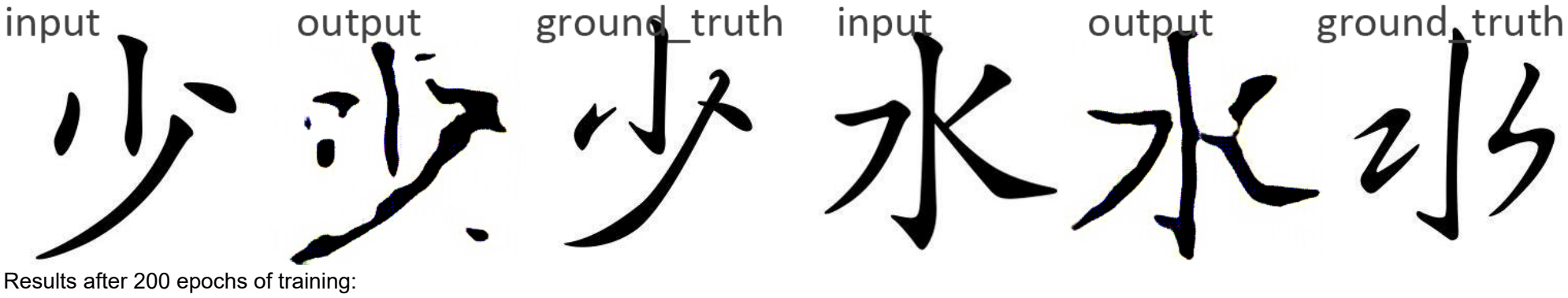
More results are shown in the "Comparison on Different Network Architectures and Learning Rate" part.

4.3 Chinese Calligraphy

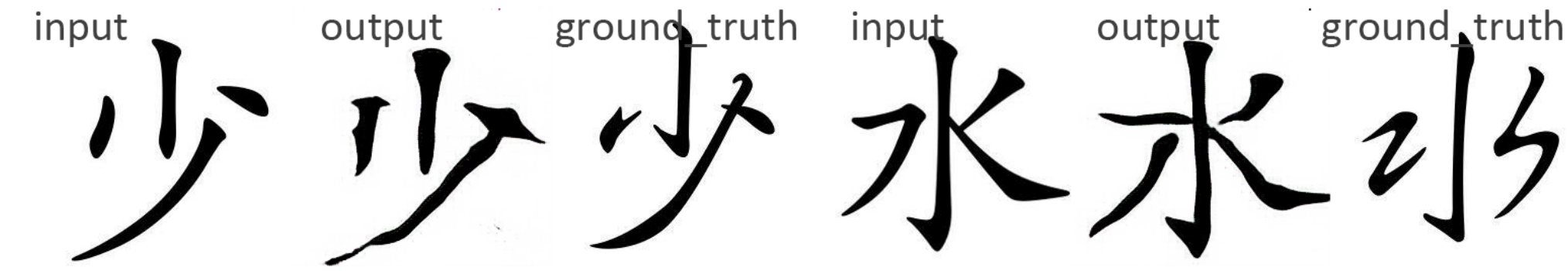
This pix2pix model used resnet_9blocks as the generator architecture. The learning rate is 0.0002 and decay linearly to zero in 200 epochs after it holds 0.0002 for 300 epochs.

In the following figures, the generated images at different epochs of training are shown.

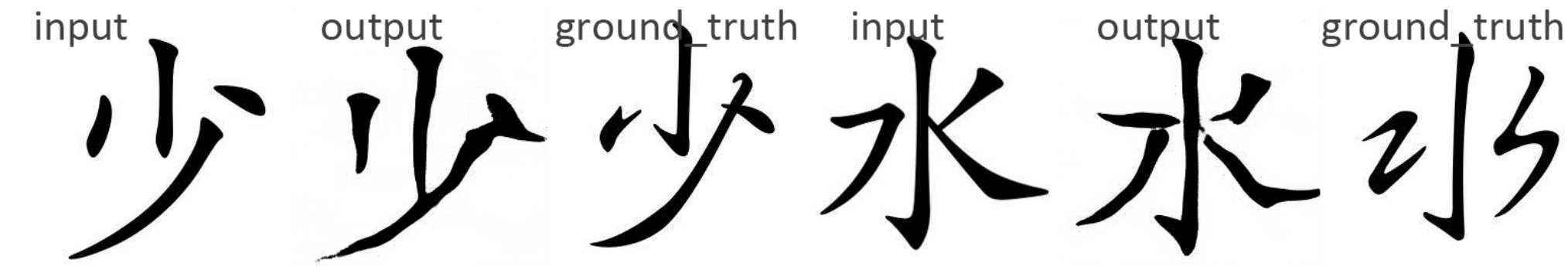
Results after 100 epochs of training:



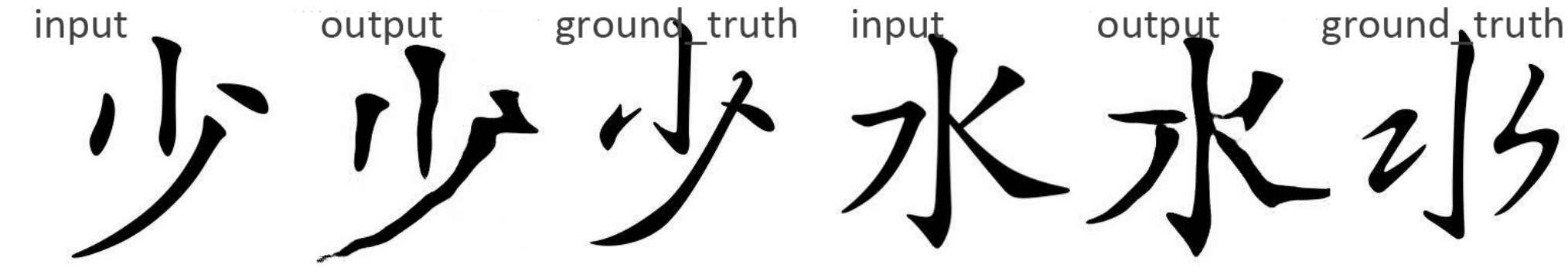
Results after 200 epochs of training:



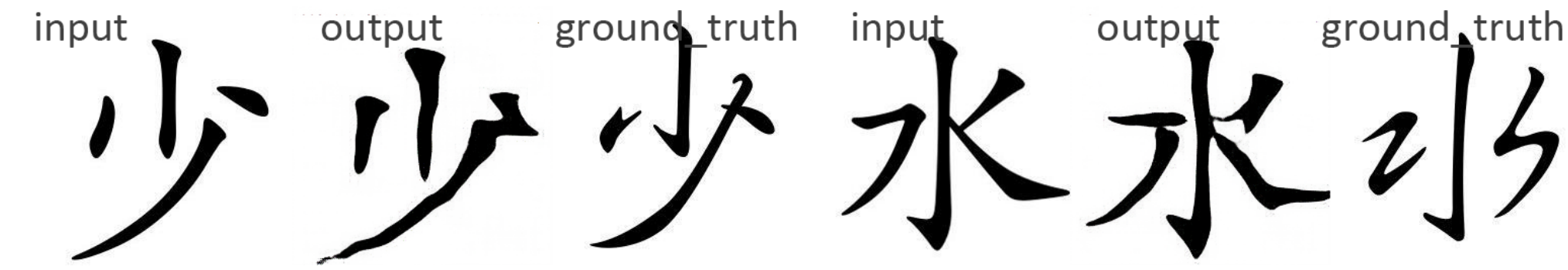
Results after 300 epochs of training:



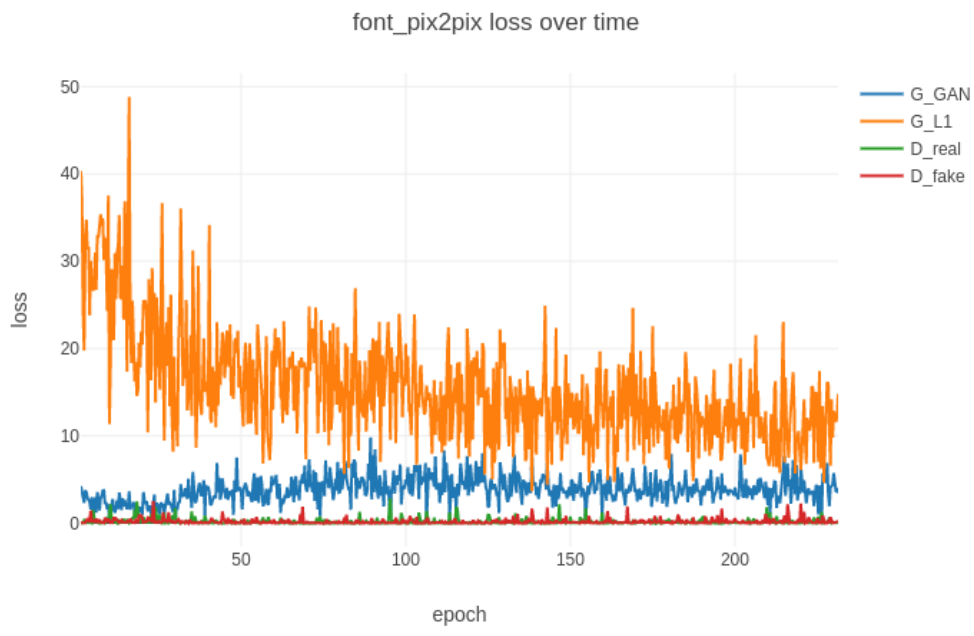
Results after 400 epochs of training:



Results after 500 epochs of training:



The following figure shows how the items of loss changes in the training. The results is not very satisfactory. They still largely keep the shape of the input. It is possible that the dataset is too small or the training is not enough. But it is more likely because the discriminator uses 70x70 patch as receptive fields. In this application, the information is quite sparse in images so it fails to discriminate real and fake images well only based on local information. However, we still can find that it can more or less learn to reconsture the characters, connect the specific strokes and make the strokes more flexuous.



5. Comparison on Different Network Architectures and Learning Rate

In order to explore whether we can get better results by different settings of pix2pix, we tried another generator network architecture ResNet [6] (the default one is U-Net). ResNet is the generator network architecture of CycleGAN [2]. It has 9 residual blocks for 256 × 256 or higher-resolution training images. Compared with the U-Net, ResNet has no skip connection but it is deeper, benefited from the residual blocks. The detailed network architecture of U-Net and ResNet has been described in the Appendix part of [1] and [2] respectively. This variance is tried in image deblurring and generating RGB Images from Graysacle Depth Images.

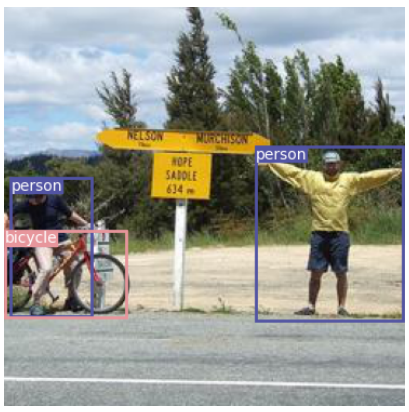
We also check the effect of the learning rate. Pix2pix default using learning rate (LR) = 0.0002. And it starts to decay to zero linearly after half of the training epochs have been finished. We explored to increase LR to 0.0005 and cancel the LR decay. This variance is tried in image deblurring.

For simplicity of expression, we call the model with the default setting as "U-Net LR=0.0002". We call the model without learning rate decay as "U-Net LR=0.0002 hold". We call the model with a higher learning rate and linear decay as "U-Net LR=0.0005". We call the model with ResNet generator and linear learning rate decay as "ResNet LR=0.0002".

5.1 Image Deblurring

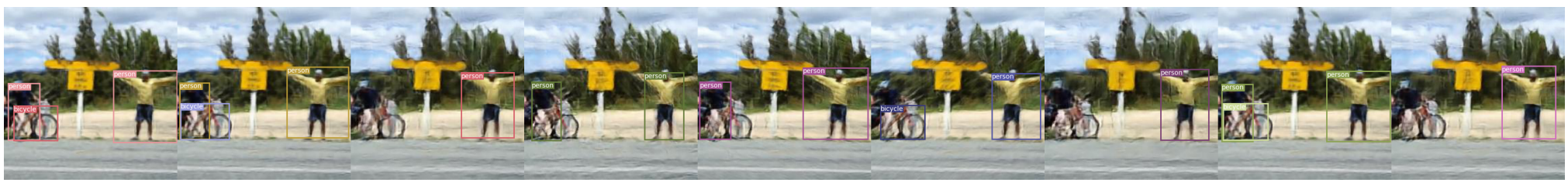
In the following figures, the deblurring results of 4 different models on 9 images from the small test set for visualization are shown.

Shape image:



Blurry images (the first three images from left have gaussian blur, the three images in the middle have motion blur, the three images on the right have disk blur):

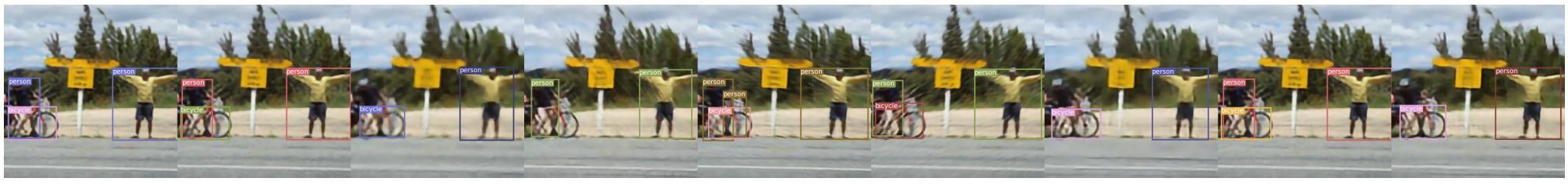
Results of U-Net LR=0.0002:



Results of U-Net LR=0.0002 hold:

Results of U-Net LR=0.0005:

Results of ResNet LR=0.0002:

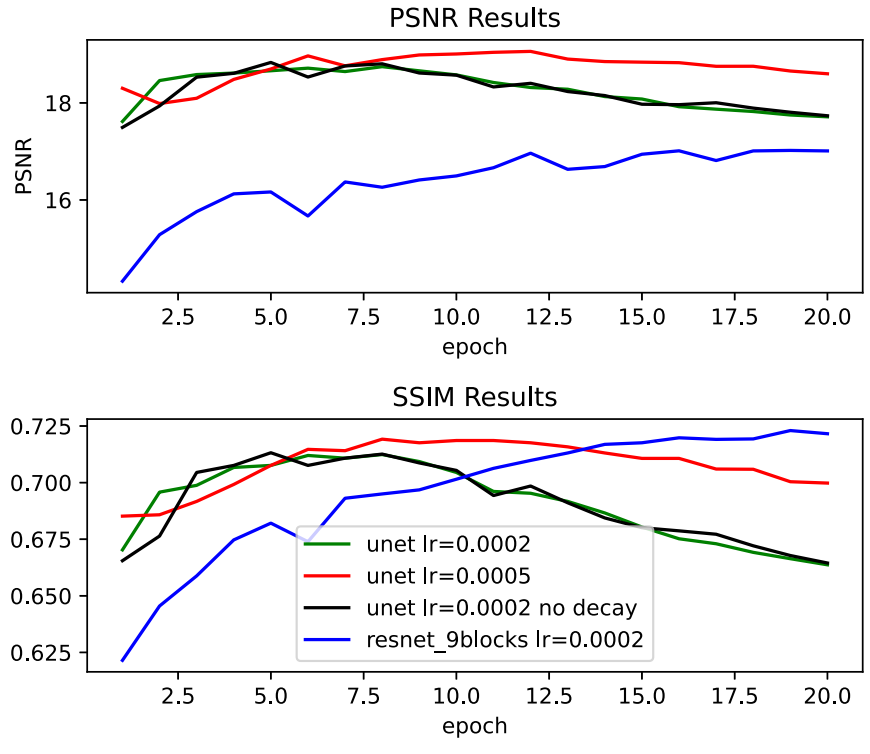


The following figure shows how the average PSNR and SSIM of the generated images of 4 different models on the small test set (270 images) for visualization change with epoch.

The 3 models with U-Net seem to learn very fast, but the performance decreased a little bit afterward. The decay in learning rate has little effect when LR=0.0002.

Bigger learning rate (0.0005) gets better results. We are not sure about the reason. We trained the models on a dataset for 20 epochs with more than 20 thousand images. We think the training is enough. But the phenomenon that a bigger learning rate has better results seems to imply that the network needs more training to perform better.

The deeper ResNet has more parameters and took longer time to reach similar performance with the U-Net. But the deeper ResNet seems to have better performance after finishing the training. We can also notice this from the images shown above.



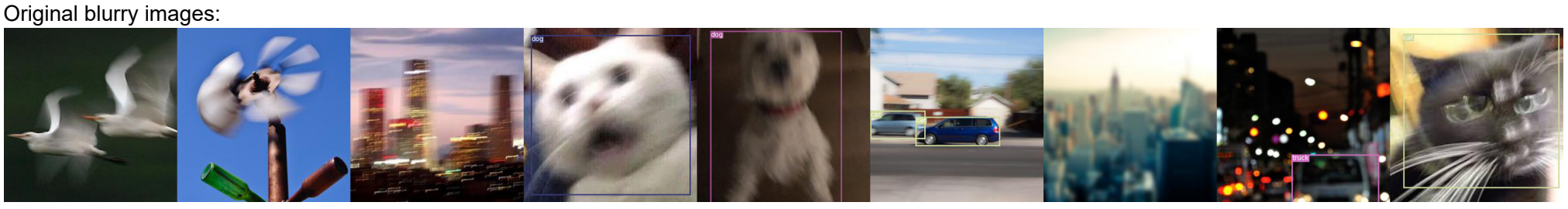
The following table shows the average PSNR and SSIM of different models on. From the PSNR data, we notice that the performances of the 4 models are similar. As for the SSIM data, the model with a higher learning rate and the model with ResNet have higher values.

	$\mathrm{UNet, LR} = 0.0002$	$\mathrm{UNet, LR} = 0.0002$ hold	$\mathrm{UNet, LR} = 0.0005$	$\mathrm{ResNet, LR} = 0.0002$
Avg. PSNR	17.8870	17.8683	**18.7120**	17.2981
Avg. SSIM	0.6661	0.6652	0.7047	**0.7289**

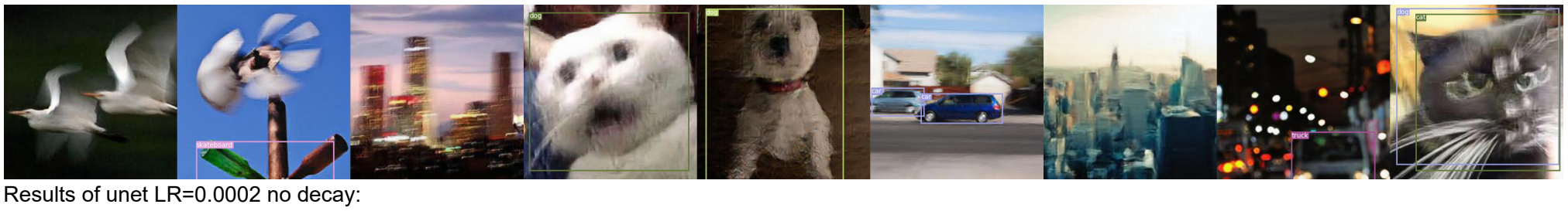
In the following figures, the deblurring results of 4 different models on a new small test set are shown. This test set is made up of originally blurry images found on the internet. This new originally blurry dataset can be partly blurry so they are different from the images in the training set, which are artificially blurred in the whole image.

From the results, we can find that the networks gave slightly better results than the blurry images, but not as good as the artificially blurred images in the test set, especially when there is big motion blur. In general, the model with ResNet has the best performance among the 4 models.

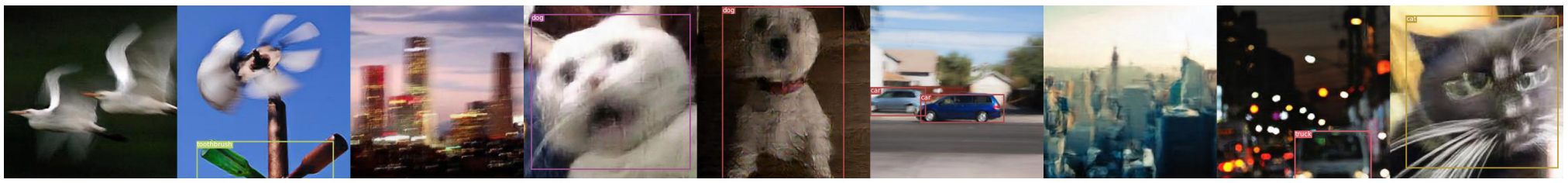
Original blurry images:



Results of unet LR=0.0002:

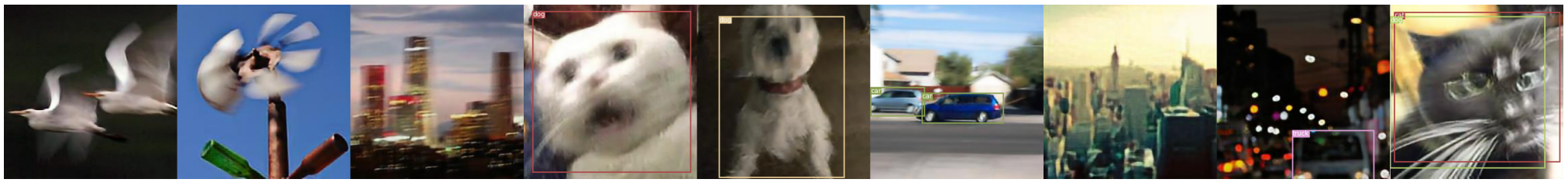


Results of unet LR=0.0002 no decay:



Results of unet LR=0.0005:

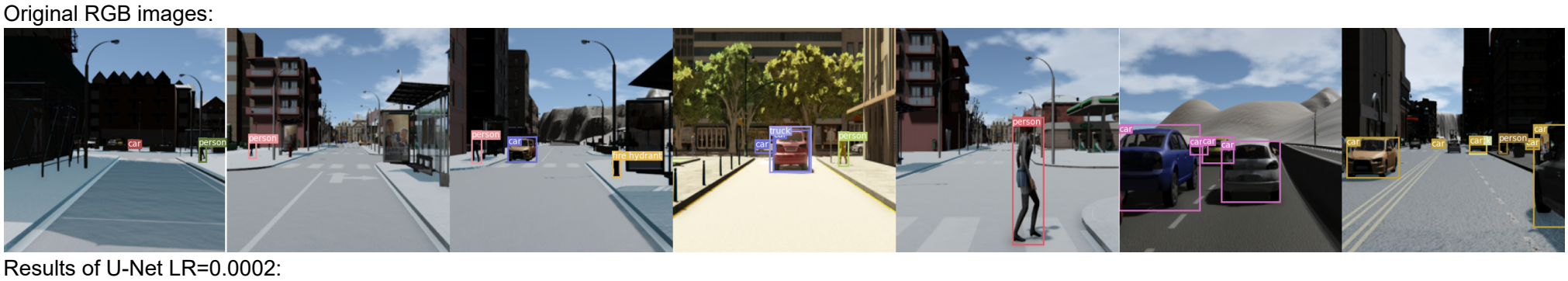
Results of resnet LR=0.0002:



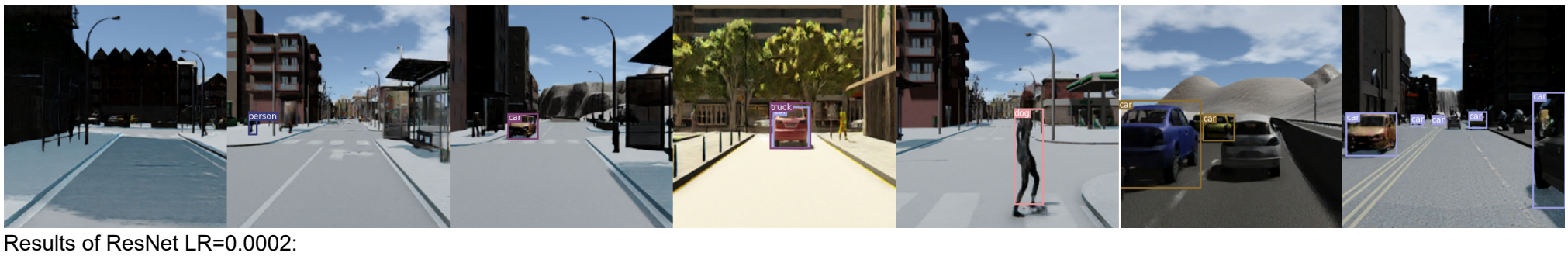
5.2 Generating RGB Images from Graysacle Depth Images

In the following figures, the generated RGB images from input graysacle depth images of 2 different models on a small test set are shown. This test set is selected from the visualization dataset of depth dataset. Each image in this set has a unique scene in the simulatied city. These 7 kinds of scene are also included in the traning set.

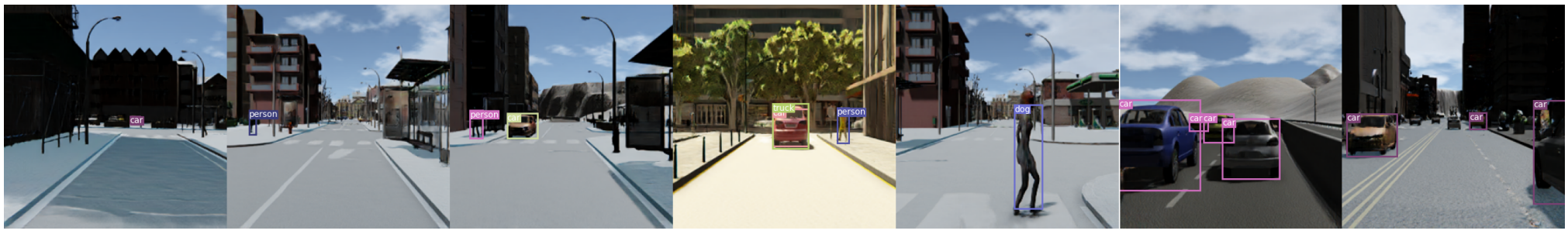
Original RGB images:



Results of U-Net LR=0.0002:



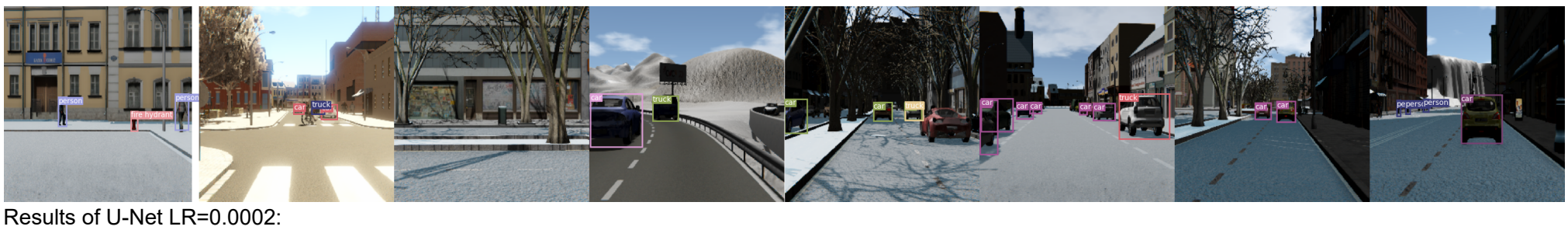
Results of ResNet LR=0.0002:



In the following figures, the generated RGB images of 2 different models on a new small test set are shown. This test set is selected from the SYNTHIA-AL (http://synthia-dataset.net/downloads/) dataset. Each image in this set has a unique scene in the simulatied city. These 7 kinds of scene are not included in the traning set.

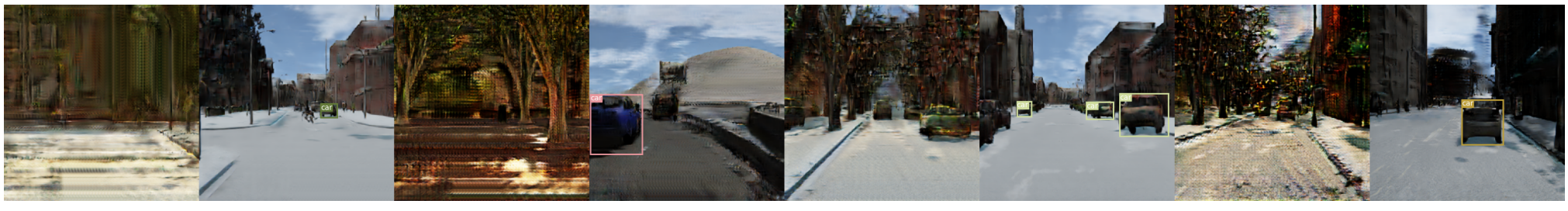
Input grayscale depth images:

Original RGB images:



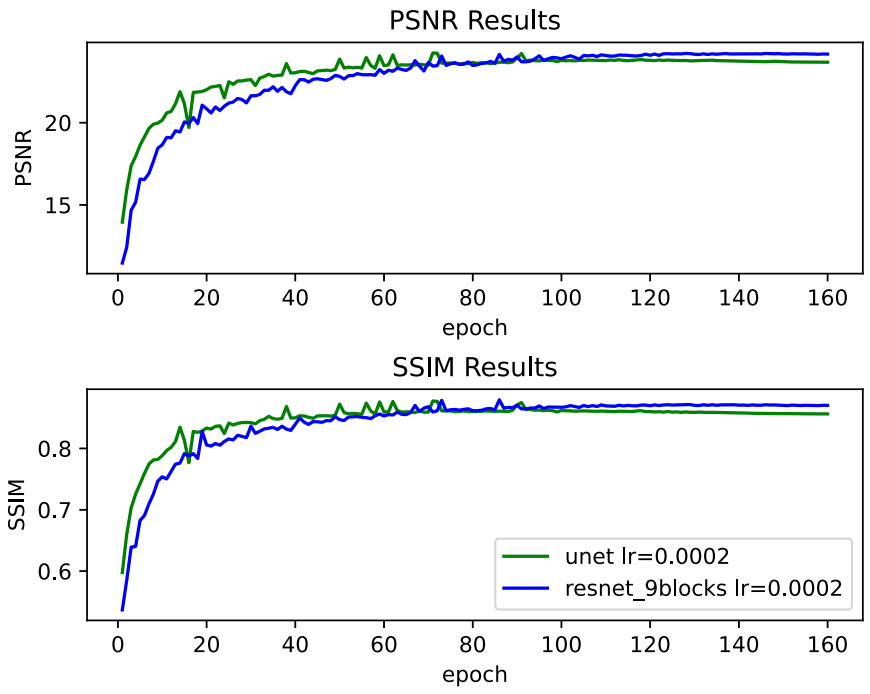
Results of U-Net LR=0.0002:

Results of ResNet LR=0.0002:



In the results shown above, the networks performed much better in the scenes they were trained on. It is obvious that the models actually use what they learned from the training set to generate images and cannot "imagine" correctly in new scenes. For example, because most trees are with leaves in the training set, so the network generated trees with leaves in testing. In the unfamiliar scenes, the networks failed to generate good details inside the contour of vehicles and buildings either. If we feed the network with enough all kinds of images in training, the network should get better performance.

The following figure shows how the average PSNR and SSIM of the generated images of 2 different models in the visualization set changes with epoch.



The following table shows the average PSNR and SSIM of 2 different models on the test set. The deeper ResNet generator has better performance in both old and new test sets.

	UNet, Old	UNet, New	ResNet, Old	ResNet, New
Avg. PSNR	24.3231	6.3220	**24.3985**	**7.0870**
Avg. SSIM	0.8675	0.3045	**0.8703**	**0.3350**

References

[1] Isola, Phillip, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. "Image-to-image translation with conditional adversarial networks." In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 1125-1134. 2017.

[2] Zhu, Jun-Yan, Taesung Park, Phillip Isola, and Alexei A. Efros. "Unpaired image-to-image translation using cycle-consistent adversarial networks." In Proceedings of the IEEE international conference on computer vision, pp. 2223-2232. 2017.

[3] Kupyn, Orest, Volodymyr Budzan, Mykola Mykhailych, Dmytro Mishkin, and Jiří Matas. "Deblurgan: Blind motion deblurring using conditional adversarial networks." In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 8183-8192. 2018.

[4] Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. "U-net: Convolutional networks for biomedical image segmentation." In International Conference on Medical image computing and computer-assisted intervention, pp. 234-241. Springer, Cham, 2015.

[5] Mescheder, Lars, Andreas Geiger, and Sebastian Nowozin. "Which training methods for GANs do actually converge?." arXiv preprint arXiv:1801.04406 (2018).

[6] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770-778. 2016.

Appendix

In the below, we show the rewritten code in Jupyter Notebook. The multi-file Python project (https://github.com/YingfuXu/pix2pixCourseProject) can be found here: https://github.com/YingfuXu/pix2pixCourseProject (https://github.com/YingfuXu/pix2pixCourseProject)

In [1]:
```
'''**********************************************6.1 Libraries**********************************************'''
from __future__ import print_function
import os
from os import listdir
from os.path import join
from math import log10
import torch
import torch.nn as nn
import torch.optim as optim
from torch.nn import init
from torch.optim import lr_scheduler
from torch.utils.data import DataLoader
import torch.backends.cudnn as cudnn
import functools
from matplotlib import pyplot as plt
import torch.utils.data as data
import numpy as np
from PIL import Image
import torchvision.transforms as transforms
import random



'''**********************************************6.2 Parameters**********************************************'''
# We can adjust the parameters directly here.
# change the dataset in the current directory
dataset = 'facades'

batch_size = 1
test_batch_size=1
# direction of the dataset
direction='b2a'
# number of channels
input_nc=3
output_nc=3
# the number of filters in the first convolution layer
ngf=64
ndf=64

epoch_count=1
niter=100
niter_decay=100

lr=0.0002
lr_policy='lambda'
lr_decay_iters=30
beta1=0.5

threads=0
seed=123
lamb=10

torch.manual_seed(seed)
# this is the cpu version
# device = torch.device("cpu")

# if using the gpu, open the following code
torch.cuda.manual_seed(seed)
device = torch.device("cuda: 0")



'''**********************************************6.3 Dataset Loading and Image Preprocessing**********************'''

# Adjust if it is a image file
def is_image_file(filename):
    return any(filename.endswith(extension) for extension in [".png", ".jpg", ".jpeg"])
# Loading images and resizing
def load_img(filepath):
    img = Image.open(filepath).convert('RGB')
    img = img.resize((256, 256), Image.BICUBIC)
    return img
# Saving images
def save_img(image_tensor, filename):
    image_numpy = image_tensor.float().numpy()
    image_numpy = (np.transpose(image_numpy, (1, 2, 0)) + 1) / 2.0 * 255.0
    image_numpy = image_numpy.clip(0, 255)
    image_numpy = image_numpy.astype(np.uint8)
    image_pil = Image.fromarray(image_numpy)
    image_pil.save(filename)
    print("Image saved as {}".format(filename))


# Inherit the data.Dataset and create a new dataset for getting each item easily
class DatasetFromFolder(data.Dataset):

    def __init__(self, image_dir, direction):
        super(DatasetFromFolder, self).__init__()
```

```
        self.direction = direction
        self.a_path = join(image_dir, "a")
        self.b_path = join(image_dir, "b")
        self.image_filenames = [x for x in listdir(self.a_path) if is_image_file(x)]

        transform_list = [transforms.ToTensor(),
                          transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]
        self.transform = transforms.Compose(transform_list)

    def __getitem__(self, index):
        # covert into RGB picture, do they cover the original pictures?
        # Answer: this is just converting the original pictures
        # join : combine each paths in the list
        # Preprocessing of the pictures
        a = Image.open(join(self.a_path, self.image_filenames[index])).convert('RGB')
        b = Image.open(join(self.b_path, self.image_filenames[index])).convert('RGB')
        # Resize
        a = a.resize((286, 286), Image.BICUBIC)
        b = b.resize((286, 286), Image.BICUBIC)
        # To tensor
        a = transforms.ToTensor()(a)
        b = transforms.ToTensor()(b)
        # add a offset to the picture
        w_offset = random.randint(0, max(0, 286 - 256 - 1))
        h_offset = random.randint(0, max(0, 286 - 256 - 1))

        a = a[:, h_offset:h_offset + 256, w_offset:w_offset + 256]
        b = b[:, h_offset:h_offset + 256, w_offset:w_offset + 256]
        # Normalize
        a = transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))(a)
        b = transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))(b)


        if random.random() < 0.5:
            idx = [i for i in range(a.size(2) - 1, -1, -1)]
            idx = torch.LongTensor(idx)
            a = a.index_select(2, idx)
            b = b.index_select(2, idx)

        if self.direction == "a2b":
            return a, b
        else:
            return b, a

    def __len__(self):
        return len(self.image_filenames)



'''**********************************************6.4 Network Components**********************************************'''

class Inconv(nn.Module):
    def __init__(self, in_ch, out_ch, norm_layer, use_bias):
        super(Inconv, self).__init__()
        self.inconv = nn.Sequential(
            nn.ReflectionPad2d(3),
            nn.Conv2d(in_ch, out_ch, kernel_size=7, padding=0,
                      bias=use_bias),
            norm_layer(out_ch),
            nn.ReLU(True)
        )

    def forward(self, x):
        x = self.inconv(x)
        return x

class Down(nn.Module):
    def __init__(self, in_ch, out_ch, norm_layer, use_bias):
        super(Down, self).__init__()
        self.down = nn.Sequential(
            nn.Conv2d(in_ch, out_ch, kernel_size=3,
                      stride=2, padding=1, bias=use_bias),
            norm_layer(out_ch),
            nn.ReLU(True)
        )

    def forward(self, x):
        x = self.down(x)
        return x

class Up(nn.Module):
    def __init__(self, in_ch, out_ch, norm_layer, use_bias):
        super(Up, self).__init__()
        self.up = nn.Sequential(
            nn.ConvTranspose2d(in_ch, out_ch,
                               kernel_size=3, stride=2,
                               padding=1, output_padding=1,
                               bias=use_bias),
```

```
            norm_layer(out_ch),
            nn.ReLU(True)
        )

    def forward(self, x):
        x = self.up(x)
        return x


class Outconv(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(Outconv, self).__init__()
        self.outconv = nn.Sequential(
            nn.ReflectionPad2d(3),
            nn.Conv2d(in_ch, out_ch, kernel_size=7, padding=0),
            nn.Tanh()
        )

    def forward(self, x):
        x = self.outconv(x)
        return x

def init_net(net, init_type='normal', init_gain=0.02, gpu_id='cuda:0'):

    with torch.no_grad():
        net.to(gpu_id)
        init_weights(net, init_type, gain=init_gain)
    return net


'''##############################################6.5 Gernerator Network##############################################'''
class UnetSkipConnectionBlock(nn.Module):
    """Defines the Unet submodule with skip connection.
        X -------------------identity----------------------
        |-- downsampling -- |submodule| -- upsampling --|
    """

    def __init__(self, outer_nc, inner_nc, input_nc=None,
                 submodule=None, outermost=False, innermost=False, norm_layer=nn.BatchNorm2d, use_dropout=False):
        """Construct a Unet submodule with skip connections.

        Parameters:
            outer_nc (int) -- the number of filters in the outer conv layer
            inner_nc (int) -- the number of filters in the inner conv layer
            input_nc (int) -- the number of channels in input images/features
            submodule (UnetSkipConnectionBlock) -- previously defined submodules
            outermost (bool)    -- if this module is the outermost module
            innermost (bool)    -- if this module is the innermost module
            norm_layer          -- normalization layer
            use_dropout (bool)  -- if use dropout layers.
        """
        super(UnetSkipConnectionBlock, self).__init__()
        self.outermost = outermost
        if type(norm_layer) == functools.partial:
            use_bias = norm_layer.func == nn.InstanceNorm2d
        else:
            use_bias = norm_layer == nn.InstanceNorm2d
        if input_nc is None:
            input_nc = outer_nc
        downconv = nn.Conv2d(input_nc, inner_nc, kernel_size=4,
                             stride=2, padding=1, bias=use_bias)
        downrelu = nn.LeakyReLU(0.2, True)
        downnorm = norm_layer(inner_nc)
        uprelu = nn.ReLU(True)
        upnorm = norm_layer(outer_nc)

        if outermost:
            upconv = nn.ConvTranspose2d(inner_nc * 2, outer_nc,
                                        kernel_size=4, stride=2,
                                        padding=1)
            down = [downconv]
            up = [uprelu, upconv, nn.Tanh()]
            model = down + [submodule] + up
        elif innermost:
            upconv = nn.ConvTranspose2d(inner_nc, outer_nc,
                                        kernel_size=4, stride=2,
                                        padding=1, bias=use_bias)
            down = [downrelu, downconv]
            up = [uprelu, upconv, upnorm]
            model = down + up
        else:
            upconv = nn.ConvTranspose2d(inner_nc * 2, outer_nc,
                                        kernel_size=4, stride=2,
                                        padding=1, bias=use_bias)
            down = [downrelu, downconv, downnorm]
            up = [uprelu, upconv, upnorm]
```

```
            if use_dropout:
                model = down + [submodule] + up + [nn.Dropout(0.5)]
            else:
                model = down + [submodule] + up

        self.model = nn.Sequential(*model)

    def forward(self, x):
        if self.outermost:
            return self.model(x)
        else:   # add skip connections
            return torch.cat([x, self.model(x)], 1)



class UnetGenerator(nn.Module):
    """Create a Unet-based generator"""

    def __init__(self, input_nc, output_nc, num_downs, ngf=64, norm_layer=nn.BatchNorm2d, use_dropout=False):
        """Construct a Unet generator
        Parameters:
            input_nc (int)  -- the number of channels in input images
            output_nc (int) -- the number of channels in output images
            num_downs (int) -- the number of downsamplings in UNet. For example, # if |num_downs| == 7,
                               image of size 128x128 will become of size 1x1 # at the bottleneck
            ngf (int)       -- the number of filters in the last conv layer
            norm_layer      -- normalization layer

        We construct the U-Net from the innermost layer to the outermost layer.
        It is a recursive process.
        """
        super(UnetGenerator, self).__init__()
        # construct unet structure
        unet_block = UnetSkipConnectionBlock(ngf * 8, ngf * 8, input_nc=None, submodule=None,
                                             norm_layer=norm_layer, innermost=True)  # add the innermost layer
        for i in range(num_downs - 5):          # add intermediate layers with ngf * 8 filters
            unet_block = UnetSkipConnectionBlock(ngf * 8, ngf * 8, input_nc=None, submodule=unet_block,
                                                 norm_layer=norm_layer, use_dropout=use_dropout)
        # gradually reduce the number of filters from ngf * 8 to ngf
        unet_block = UnetSkipConnectionBlock(ngf * 4, ngf * 8, input_nc=None, submodule=unet_block,
                                             norm_layer=norm_layer)
        unet_block = UnetSkipConnectionBlock(ngf * 2, ngf * 4, input_nc=None, submodule=unet_block,
                                             norm_layer=norm_layer)
        unet_block = UnetSkipConnectionBlock(ngf, ngf * 2, input_nc=None, submodule=unet_block,
                                             norm_layer=norm_layer)
        self.model = UnetSkipConnectionBlock(output_nc, ngf, input_nc=input_nc, submodule=unet_block,
                                             outermost=True, norm_layer=norm_layer)  # add the outermost layer

    def forward(self, input):
        """Standard forward"""
        return self.model(input)


def define_G(input_nc, output_nc, ngf, norm='batch', use_dropout=False, init_type='normal', init_gain=0.02, gpu_id='cuda:0'):
    net = None
    norm_layer = get_norm_layer(norm_type=norm)

#     net = ResnetGenerator(input_nc, output_nc, ngf, norm_layer=norm_layer, use_dropout=use_dropout, n_blocks=9)
    net = UnetGenerator(input_nc, output_nc, 8, ngf, norm_layer=norm_layer, use_dropout=use_dropout)
    return init_net(net, init_type, init_gain, gpu_id)


'''#####################################################6.6 Discriminator Network#########################################################'''
class NLayerDiscriminator(nn.Module):
    def __init__(self, input_nc, ndf=64, n_layers=3, norm_layer=nn.BatchNorm2d, use_sigmoid=False):
        super(NLayerDiscriminator, self).__init__()
        if type(norm_layer) == functools.partial:
            use_bias = norm_layer.func == nn.InstanceNorm2d
        else:
            use_bias = norm_layer == nn.InstanceNorm2d

        kw = 4
        padw = 1
        sequence = [
            nn.Conv2d(input_nc, ndf, kernel_size=kw, stride=2, padding=padw),
            nn.LeakyReLU(0.2, True)
        ]

        nf_mult = 1
        nf_mult_prev = 1
        for n in range(1, n_layers):
            nf_mult_prev = nf_mult
            nf_mult = min(2**n, 8)
            sequence += [
                nn.Conv2d(ndf * nf_mult_prev, ndf * nf_mult,
                          kernel_size=kw, stride=2, padding=padw, bias=use_bias),
                norm_layer(ndf * nf_mult),
                nn.LeakyReLU(0.2, True)
```

```
            ]

        nf_mult_prev = nf_mult
        nf_mult = min(2**n_layers, 8)
        sequence += [
            nn.Conv2d(ndf * nf_mult_prev, ndf * nf_mult,
                      kernel_size=kw, stride=1, padding=padw, bias=use_bias),
            norm_layer(ndf * nf_mult),
            nn.LeakyReLU(0.2, True)
        ]

        sequence += [nn.Conv2d(ndf * nf_mult, 1, kernel_size=kw, stride=1, padding=padw)]

        if use_sigmoid:
            sequence += [nn.Sigmoid()]

        self.model = nn.Sequential(*sequence)

    def forward(self, input):
        return self.model(input)


class PixelDiscriminator(nn.Module):
    def __init__(self, input_nc, ndf=64, norm_layer=nn.BatchNorm2d, use_sigmoid=False):
        super(PixelDiscriminator, self).__init__()
        if type(norm_layer) == functools.partial:
            use_bias = norm_layer.func == nn.InstanceNorm2d
        else:
            use_bias = norm_layer == nn.InstanceNorm2d

        self.net = [
            nn.Conv2d(input_nc, ndf, kernel_size=1, stride=1, padding=0),
            nn.LeakyReLU(0.2, True),
            nn.Conv2d(ndf, ndf * 2, kernel_size=1, stride=1, padding=0, bias=use_bias),
            norm_layer(ndf * 2),
            nn.LeakyReLU(0.2, True),
            nn.Conv2d(ndf * 2, 1, kernel_size=1, stride=1, padding=0, bias=use_bias)]

        if use_sigmoid:
            self.net.append(nn.Sigmoid())

        self.net = nn.Sequential(*self.net)

    def forward(self, input):
        return self.net(input)

def define_D(input_nc, ndf, netD,
             n_layers_D=3, norm='batch', use_sigmoid=False, init_type='normal', init_gain=0.02, gpu_id='cuda:0'):
    net = None
    norm_layer = get_norm_layer(norm_type=norm)

    if netD == 'basic':
        net = NLayerDiscriminator(input_nc, ndf, n_layers=3, norm_layer=norm_layer, use_sigmoid=use_sigmoid)
    elif netD == 'n_layers':
        net = NLayerDiscriminator(input_nc, ndf, n_layers_D, norm_layer=norm_layer, use_sigmoid=use_sigmoid)
    elif netD == 'pixel':
        net = PixelDiscriminator(input_nc, ndf, norm_layer=norm_layer, use_sigmoid=use_sigmoid)
    else:
        raise NotImplementedError('Discriminator model name [%s] is not recognized' % net)

    return init_net(net, init_type, init_gain, gpu_id)


'''*********************************************6.7 Initialization, Loss Function, Optimization*************************'''
def init_weights(net, init_type='normal', gain=0.02):
    def init_func(m):
        classname = m.__class__.__name__
        if hasattr(m, 'weight') and (classname.find('Conv') != -1 or classname.find('Linear') != -1):
            if init_type == 'normal':
                init.normal_(m.weight.data, 0.0, gain)
            elif init_type == 'xavier':
                init.xavier_normal_(m.weight.data, gain=gain)
            elif init_type == 'kaiming':
                init.kaiming_normal_(m.weight.data, a=0, mode='fan_in')
            elif init_type == 'orthogonal':
                init.orthogonal_(m.weight.data, gain=gain)
            else:
                raise NotImplementedError('initialization method [%s] is not implemented' % init_type)
            if hasattr(m, 'bias') and m.bias is not None:
                init.constant_(m.bias.data, 0.0)
        elif classname.find('BatchNorm2d') != -1:
            init.normal_(m.weight.data, 1.0, gain)
            init.constant_(m.bias.data, 0.0)

    print('initialize network with %s' % init_type)
    net.apply(init_func)
```

```
def get_norm_layer(norm_type='instance'):
    if norm_type == 'batch':
        norm_layer = functools.partial(nn.BatchNorm2d, affine=True)
    elif norm_type == 'instance':
        norm_layer = functools.partial(nn.InstanceNorm2d, affine=False, track_running_stats=False)
    elif norm_type == 'switchable':
        norm_layer = SwitchNorm2d
    elif norm_type == 'none':
        norm_layer = None
    else:
        raise NotImplementedError('normalization layer [%s] is not found' % norm_type)
    return norm_layer


class GANLoss(nn.Module):
    def __init__(self, use_lsgan=True, target_real_label=1.0, target_fake_label=0.0):
        super(GANLoss, self).__init__()
        self.register_buffer('real_label', torch.tensor(target_real_label))
        self.register_buffer('fake_label', torch.tensor(target_fake_label))
        if use_lsgan:
            self.loss = nn.MSELoss()
        else:
            # binary cross entropy
            self.loss = nn.BCELoss()

    def get_target_tensor(self, input, target_is_real):
        if target_is_real:
            target_tensor = self.real_label
        else:
            target_tensor = self.fake_label
        return target_tensor.expand_as(input)

    def __call__(self, input, target_is_real):
        target_tensor = self.get_target_tensor(input, target_is_real)
        return self.loss(input, target_tensor)

def update_learning_rate(scheduler, optimizer):
    scheduler.step()
    lr = optimizer.param_groups[0]['lr']
    print('learning rate = %.7f' % lr)

# learning rate decay
def get_scheduler(optimizer):
    if lr_policy == 'lambda':
        def lambda_rule(epoch):
            lr_l = 1.0 - max(0, epoch + epoch_count - niter) / float(niter_decay + 1)
            return lr_l
        scheduler = lr_scheduler.LambdaLR(optimizer, lr_lambda=lambda_rule)
    elif lr_policy == 'step':
        scheduler = lr_scheduler.StepLR(optimizer, step_size=lr_decay_iters, gamma=0.1)
    elif lr_policy == 'plateau':
        scheduler = lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.2, threshold=0.01, patience=5)
    elif lr_policy == 'cosine':
        scheduler = lr_scheduler.CosineAnnealingLR(optimizer, T_max=opt.niter, eta_min=0)
    else:
        return NotImplementedError('learning rate policy [%s] is not implemented', opt.lr_policy)
    return scheduler


'''**************************************************6.8 Train**************************************************'''
dataroot1 = "datasets/facades/train"
dataroot2 = "datasets/facades/test"
# As for windows, delete num_works parameter
training_data_loader = DataLoader(dataset=DatasetFromFolder(dataroot1, direction), num_workers=threads, batch_size=batch_size, shuf
fle=True)
testing_data_loader = DataLoader(dataset=DatasetFromFolder(dataroot2, direction), num_workers=threads, batch_size=test_batch_size)


print('===> Building models')

#loading the generator and discriminator
net_g = define_G(input_nc, output_nc, ngf, 'batch', False, 'normal', 0.02, gpu_id=device)
net_d = define_D(input_nc + output_nc, ndf, 'basic', gpu_id=device)

#set loss fn
criterionGAN = GANLoss().to(device)
criterionL1 = nn.L1Loss().to(device)
criterionMSE = nn.MSELoss().to(device)

#setup optimizer
optimizer_g = optim.Adam(net_g.parameters(), lr=lr, betas=(beta1, 0.999))
optimizer_d = optim.Adam(net_d.parameters(), lr=lr, betas=(beta1, 0.999))

#set the learning rate adjust policy
net_g_scheduler = get_scheduler(optimizer_g)
```

```
net_d_scheduler = get_scheduler(optimizer_d)

#training process
for epoch in range(epoch_count, niter + niter_decay + 1):

    for iteration, batch in enumerate(training_data_loader, 1):
        # forward
        real_a, real_b = batch[0].to(device), batch[1].to(device)
        fake_b = net_g(real_a)

        ######################
        # (1) Update D network
        ######################

        optimizer_d.zero_grad()

        # D train with fake
        fake_ab = torch.cat((real_a, fake_b), 1)
        pred_fake = net_d.forward(fake_ab.detach())
        loss_d_fake = criterionGAN(pred_fake, False)

        # D train with real
        real_ab = torch.cat((real_a, real_b), 1)
        pred_real = net_d.forward(real_ab)
        loss_d_real = criterionGAN(pred_real, True)

        # Combined D loss
        loss_d = (loss_d_fake + loss_d_real) * 0.5

        loss_d.backward()

        optimizer_d.step()

        ######################
        # (2) Update G network
        ######################

        optimizer_g.zero_grad()

        # First, G(A) should fake the discriminator
        fake_ab = torch.cat((real_a, fake_b), 1)
        pred_fake = net_d.forward(fake_ab)
        loss_g_gan = criterionGAN(pred_fake, True)

        # Second, G(A) = B
        loss_g_l1 = criterionL1(fake_b, real_b) * lamb

        loss_g = loss_g_gan + loss_g_l1

        loss_g.backward()

        optimizer_g.step()

        print("===> Epoch[{}]({}/{}): Loss_D: {:.4f} Loss_G: {:.4f}".format(
            epoch, iteration, len(training_data_loader), loss_d.item(), loss_g.item()))

    update_learning_rate(net_g_scheduler, optimizer_g)
    update_learning_rate(net_d_scheduler, optimizer_d)

    # test
    avg_psnr = 0
    for batch in testing_data_loader:
        input, target = batch[0].to(device), batch[1].to(device)

        prediction = net_g(input)
        mse = criterionMSE(prediction, target)
        psnr = 10 * log10(1 / mse.item())
        avg_psnr += psnr
    print("===> Avg. PSNR: {:.4f} dB".format(avg_psnr / len(testing_data_loader)))

    #checkpoint
    if epoch % 50 == 0:
        if not os.path.exists("checkpoint"):
            os.mkdir("checkpoint")
        if not os.path.exists(os.path.join("checkpoint", dataset)):
            os.mkdir(os.path.join("checkpoint", dataset))
        net_g_model_out_path = "checkpoint/{}/netG_model_epoch_{}.pth".format(dataset, epoch)
        net_d_model_out_path = "checkpoint/{}/netD_model_epoch_{}.pth".format(dataset, epoch)
        torch.save(net_g, net_g_model_out_path)
        torch.save(net_d, net_d_model_out_path)
        print("Checkpoint saved to {}".format("checkpoint" + dataset))


'''**************************************************6.9 Test**************************************************'''
nepochs = 150
model_path = "checkpoint/{}/netG_model_epoch_{}.pth".format(dataset, nepochs)

net_g = torch.load(model_path).to(device)
```

```
if direction == "a2b":
    image_dir = "datasets/{}/test/a/".format(dataset)
else:
    image_dir = "datasets/{}/test/b/".format(dataset)

image_filenames = [x for x in os.listdir(image_dir) if is_image_file(x)]

transform_list = [transforms.ToTensor(),
                  transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]

transform = transforms.Compose(transform_list)

for image_name in image_filenames:
    img = load_img(image_dir + image_name)
    img = transform(img)
    input = img.unsqueeze(0).to(device)
    out = net_g(input)
    out_img = out.detach().squeeze(0).cpu()

    if not os.path.exists(os.path.join("result", dataset)):
        os.makedirs(os.path.join("result", dataset))
    save_img(out_img, "result/{}/{}".format(dataset, image_name))
```