Pix2pix Proof-of-concept (PyTorch)

Tian Yang 12/14/2017 - 12/20/2017

Thanks for giving me a chance to experiment such an interesting and challenging topic. This proof-of-concept project is aming to implement an image-to-image translation using conditional adversarial networks. The idea is from https://phillipi.github.io/pix2pix/ , and here I just try to transfer aerial photographies to flat maps. My timeline is:

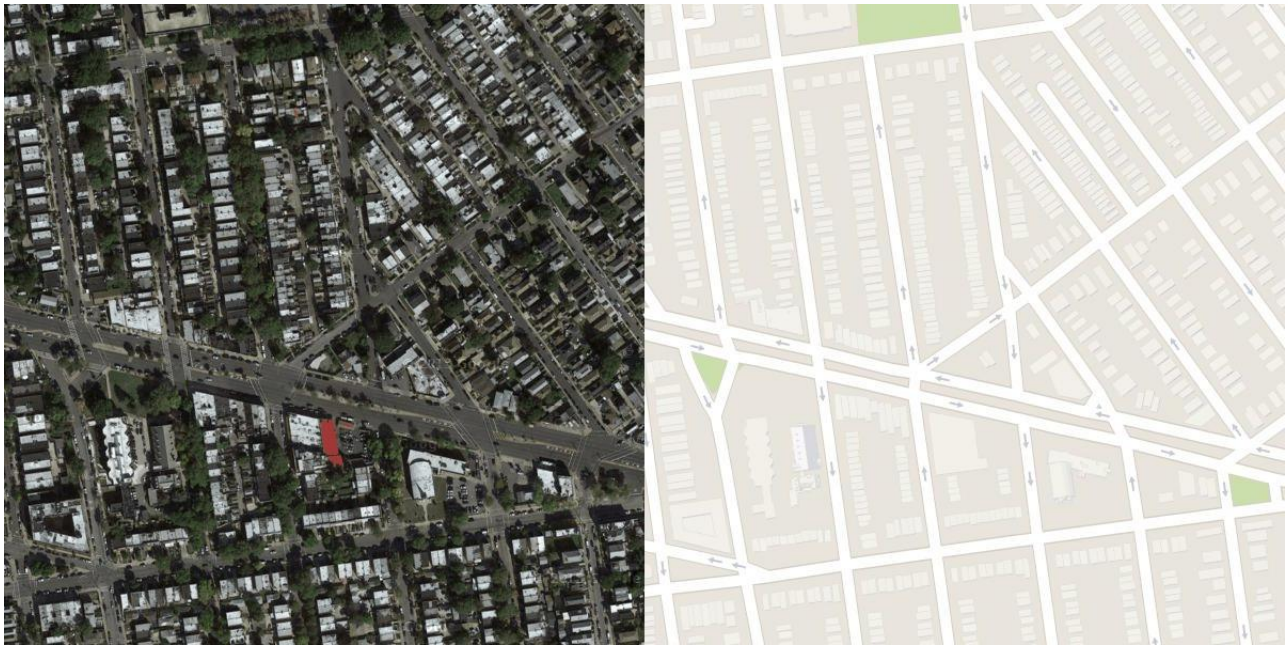
- 12/14 - 12/15: Study the paper, and understand the concept.
- 12/16 - 12/17: Find sample codes from Github, study those codes and networks, and modify them to meet specific need.
- 12/18 - 12/19: Test model, and tune the networks.
- 12/20: Document results, and write report.

This report is containing 4 parts:

- Dataset description
- Networks description
- Test results
- Possible improvements

Dataset Description

The data set contains 2196 figures in total, 1097 in training set and 1099 in validation set. Each figure has a dimension of 1200x600, with 3 channels. So actually this data set is labeled. Since my goal is to transfer aerial images to map images, the input data should be the left part, and target is the right part.

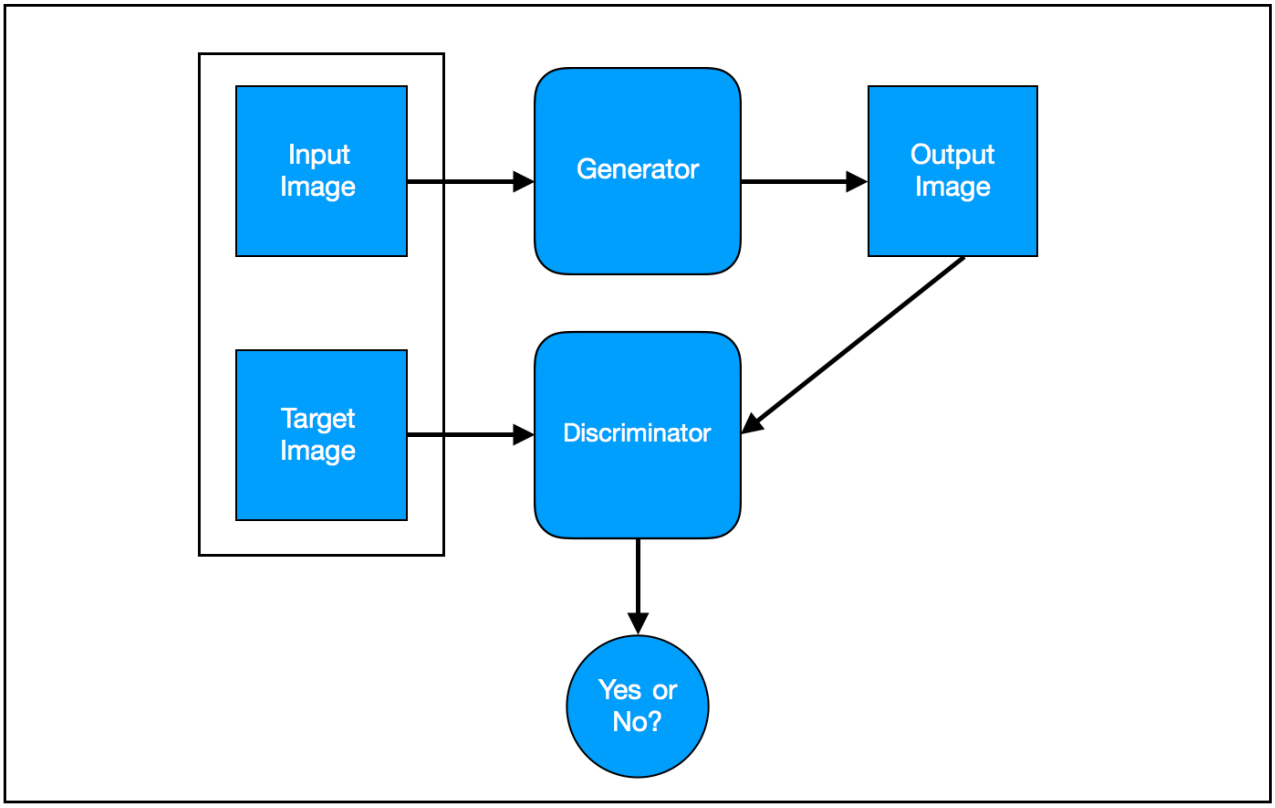


So the first thing to do is cutting these images into two parts, as inputs and targets. Then in order to speed up the training steep, I resize them to 256x256.

After preprocessing, I get 2196 groups of data, and each group has an input image and a target image, with a dimension of 256x256 respectively. Among which 1097 are training set, and 1099 are test set.

Conditional Adversarial Networks

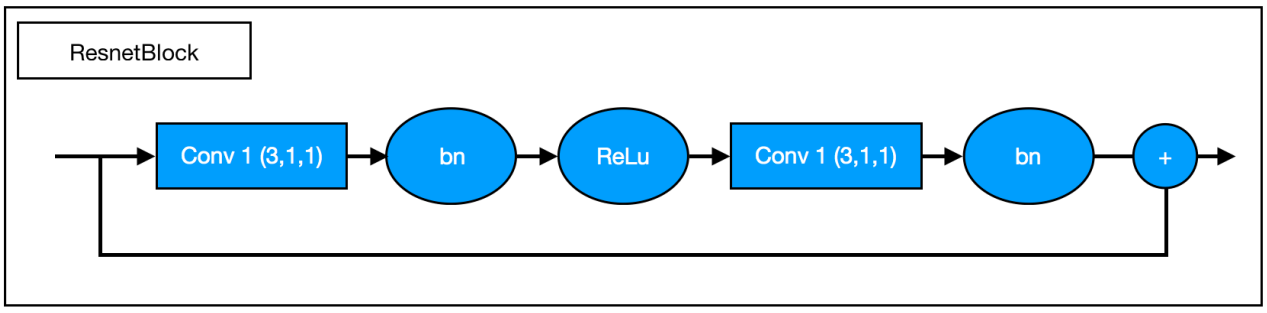
In the original paper, the authors use conditional adversarial networks, which is a special kind of generative adversarial network (GAN). There are two networks inside GAN. One called Generator, which is to generate images based on inputs, and the other one called Discriminator, which is to determine if the image generated by the Generator is real. So as conditional adversarial networks. Its architecture is like:

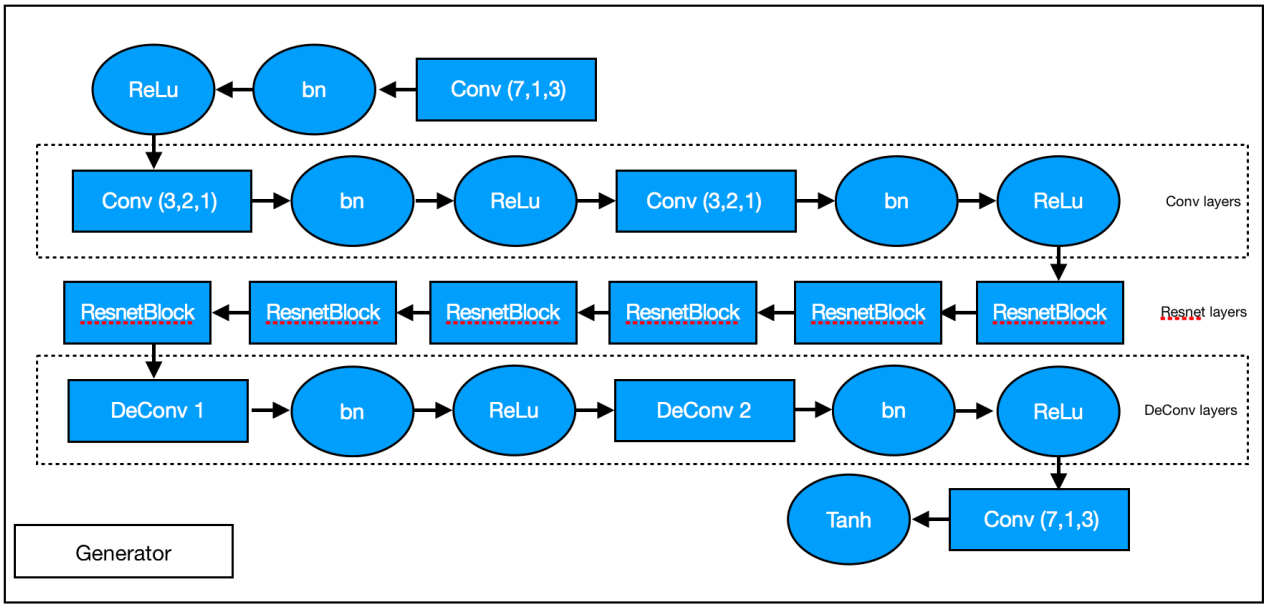


Basically, the goal of the Generator is to fool the Discriminator, and the goal of the Discriminator is not to be fooled by the Generator. So they are like adversaries to each other.

I have tried two networks. The Discriminator part of these two networks are the same, both are using a 5-layer CNN. The difference is, in the Generator part, one is using typical CNN, and the other one is using Residual Network (Resnet). The result shows that the Resnet Generator has lower loss, but the typical CNN Generator needs less training time.

Here I just introduce the Resnet Generator since it has better performance. The architecture of the Resnet Generator is like:



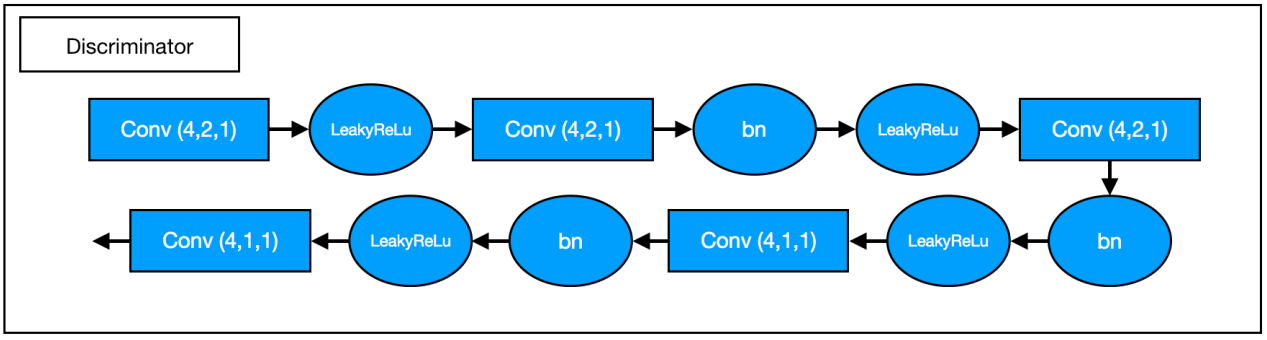


There are 1 beginning layer and 1 ending layer, both with kernel size = 7, stride = 1, and padding =3, which will not change the input dimension. And there are 2 typical convolutional layer and 2 deconvolutional layer to keep the dimension the same as beginning. So a 256x256 input image going through this Generator will produce a 256x256 output image. The code is as following:

```
class ResnetBlock(nn.Module):
    def __init__(self, dim):
        super(ResnetBlock, self).__init__()
        self.conv_block = self.build_conv_block(dim)

    def build_conv_block(self, dim):
        conv_block = []
        conv_block += [nn.Conv2d(dim, dim, kernel_size=3, padding=1),
                       nn.BatchNorm2d(dim),
                       nn.ReLU(True)]
        conv_block += [nn.Conv2d(dim, dim, kernel_size=3, padding=1),
                       nn.BatchNorm2d(dim)]
        return nn.Sequential(*conv_block)

    def forward(self, x):
        out = x + self.conv_block(x)
        return out

class ResnetGenerator(nn.Module):
    def __init__(self, input_nc, output_nc, ngf=64, n_blocks=6):
        assert (n_blocks >= 0)
        super(ResnetGenerator, self).__init__()
        self.input_nc = input_nc
        self.output_nc = output_nc
        self.ngf = ngf
```

```
        model = []
        model += [nn.Conv2d(input_nc, ngf, kernel_size=7, padding=3),
                  nn.BatchNorm2d(ngf),
                  nn.ReLU(True)]
        model += [nn.Conv2d(ngf, ngf * 2, kernel_size=3, stride=2, padding=1),
                  nn.BatchNorm2d(ngf * 2),
                  nn.ReLU(True)]
        model += [nn.Conv2d(ngf * 2, ngf * 4, kernel_size=3, stride=2,
padding=1),
                  nn.BatchNorm2d(ngf * 4),
                  nn.ReLU(True)]

        for i in range(n_blocks):
            model += [ResnetBlock(ngf * 4)]

        model += [nn.ConvTranspose2d(ngf * 4, ngf * 2, kernel_size=3,
stride=2, padding=1, output_padding=1),
                  nn.BatchNorm2d(ngf * 2),
                  nn.ReLU(True)]
        model += [nn.ConvTranspose2d(ngf * 2, ngf, kernel_size=3, stride=2,
padding=1, output_padding=1),
                  nn.BatchNorm2d(ngf),
                  nn.ReLU(True)]

        model += [nn.Conv2d(ngf, output_nc, kernel_size=7, padding=3)]
        model += [nn.Tanh()]

        self.model = nn.Sequential(*model)

    def forward(self, x):
        return self.model(x)
```

The Discriminator is much easier to understand. The architecture is like:



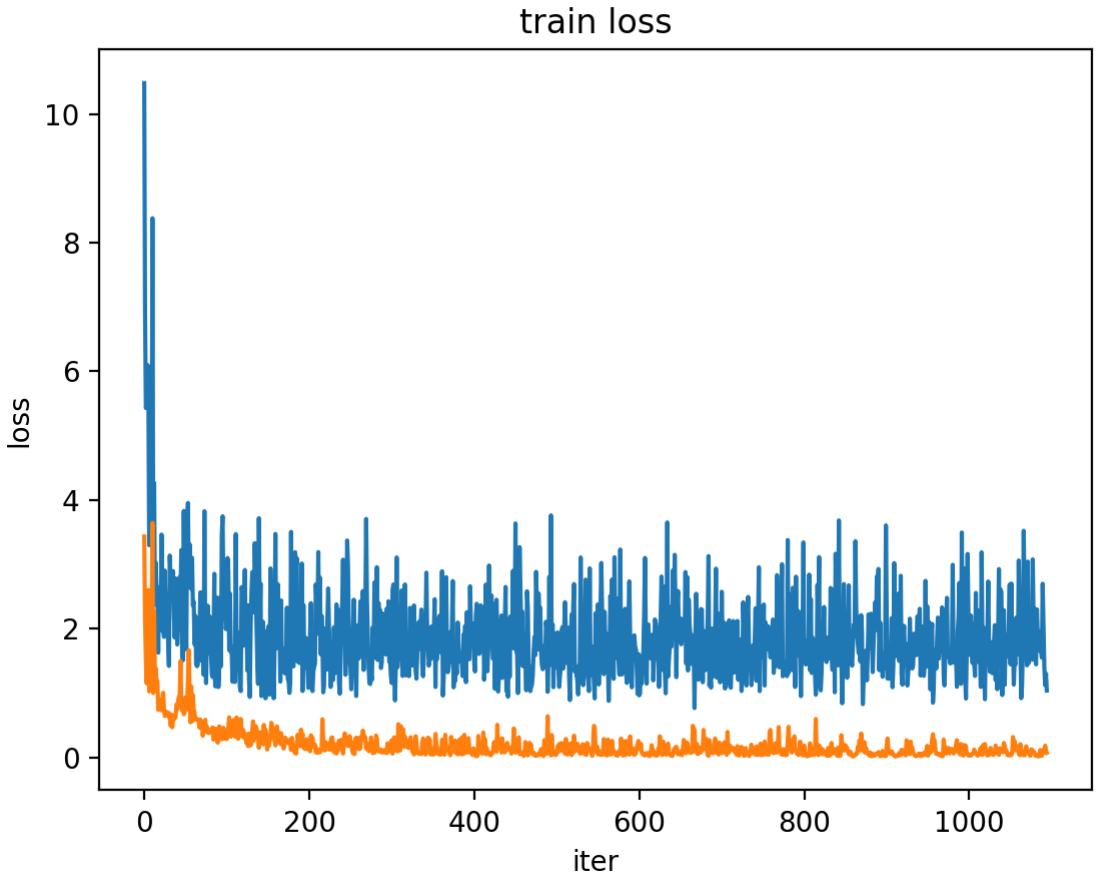
And the code:

```
class Discriminator(nn.Module):
    def __init__(self, input_nc, ndf=64):
        super(Discriminator, self).__init__()

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

        model += [nn.Conv2d(ndf, ndf * 2, kernel_size=4, stride=2, padding=1),
                  nn.BatchNorm2d(ndf * 2),
                  nn.LeakyReLU(0.2, True)]
        model += [nn.Conv2d(ndf * 2, ndf * 4, kernel_size=4, stride=2,
padding=1),
                  nn.BatchNorm2d(ndf * 4),
                  nn.LeakyReLU(0.2, True)]
        model += [nn.Conv2d(ndf * 4, ndf * 8, kernel_size=4, stride=1,
padding=1),
                  nn.BatchNorm2d(ndf * 8),
                  nn.LeakyReLU(0.2, True)]

        model += [nn.Conv2d(ndf * 8, 1, kernel_size=4, stride=1, padding=1)]

        self.model = nn.Sequential(*model)

    def forward(self, x):
        return self.model(x)
```

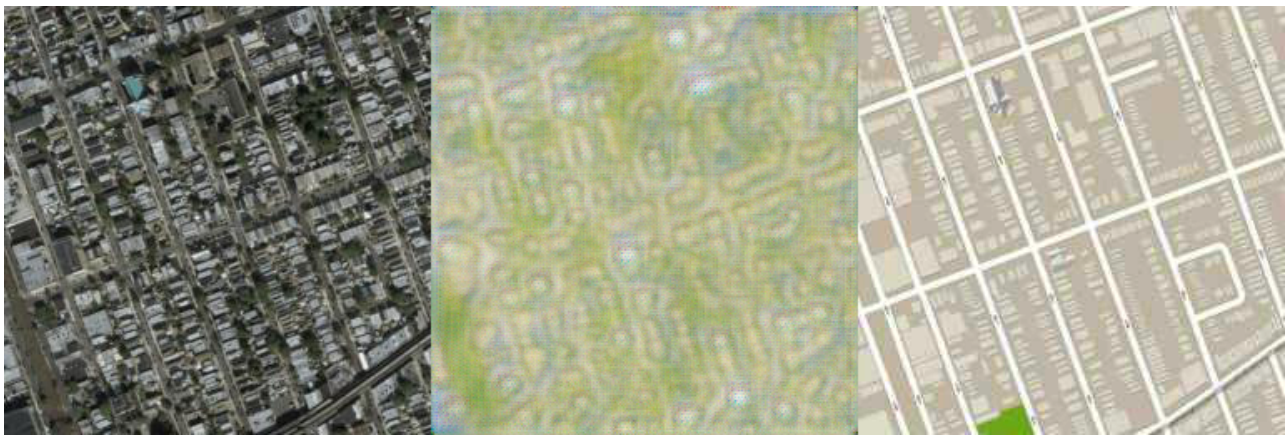
Results

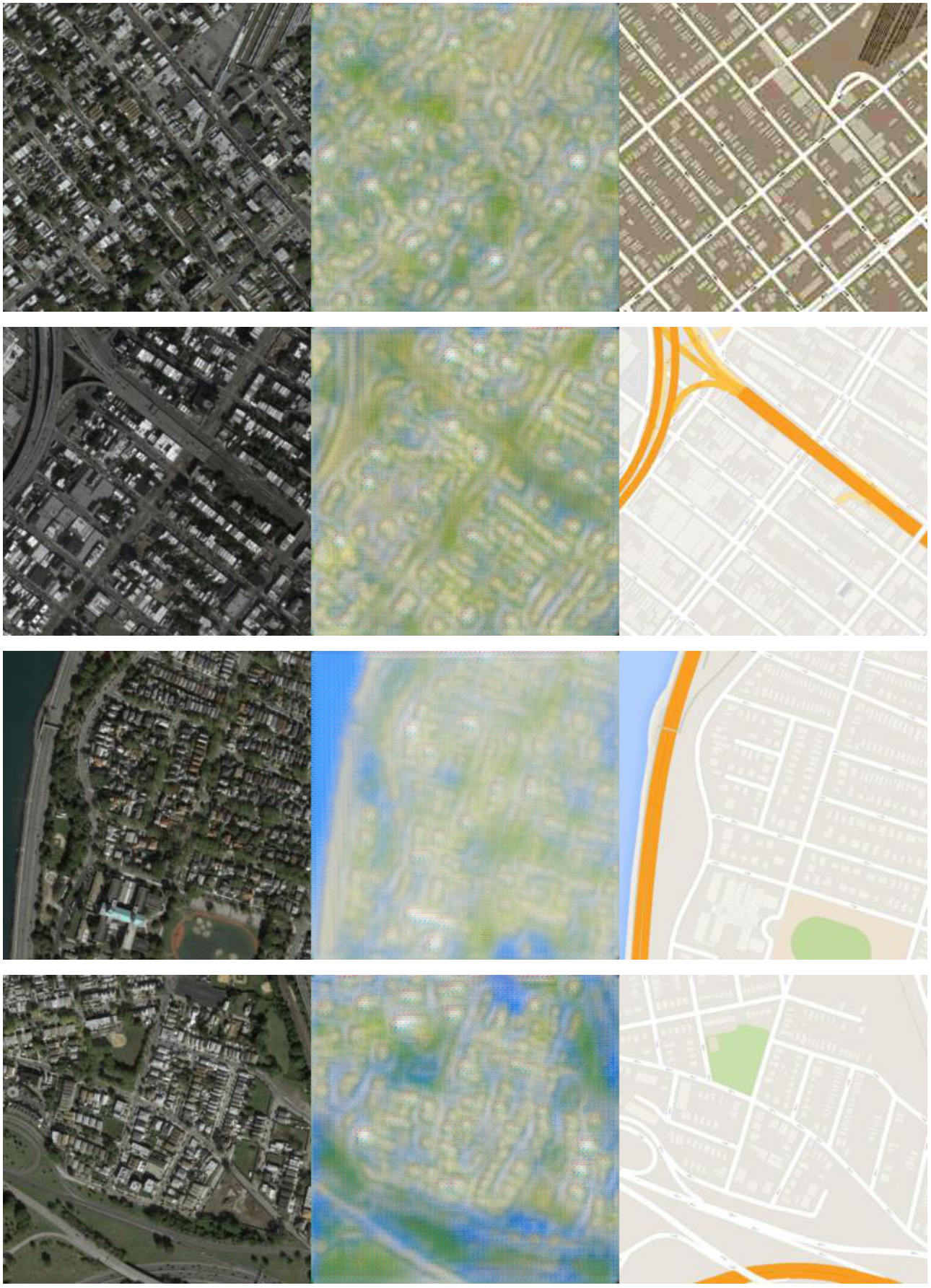
Epoch: 1 Blue line: Loss of Generator

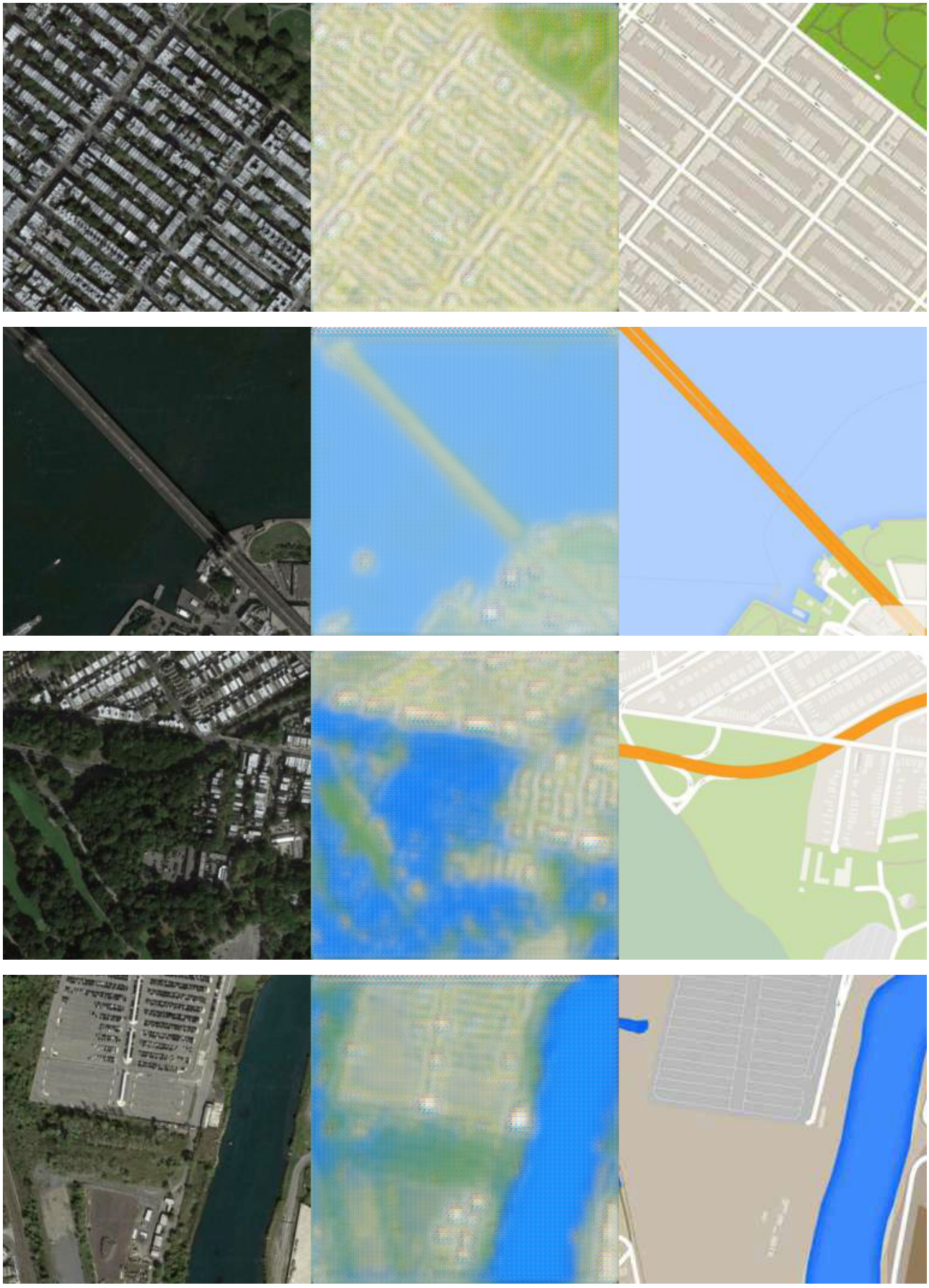
Training time: 3h Orange line: Loss of Discriminator



Some results in test set: (1st col is input, 2nd col is output, 3rd col is target)







Summaries:

1. It could be seen that when buildings are too crowd, the transformation is not good. The Generator would misclassify the spaces between buildings as roads, but in target images those buildings are organized as blocks. But it do recognize the direction of roads correctly.
2. It is good to see that the model could distinguish water and ground. It could even mark out the bridge over the sea. But sometimes it would mess up water and grassland.

Since the network is only trainned for 1 epoch, it is pretty good to get these resonable results.

Possible Improvements

- If using GPU, it would be much faster. Since I am using Macbook Pro, it is not supporting cuda, which would slow down the trianing process.
- Increasing training epoch. It could be the easiest way to get improvement. Because of the time issue, I don't have time to train enough epoch.
- Changing the padding method may influences the results, but I have not test it. In the original paper, the authors use padding before conv layers. But here I just use normal padding inside conv layers.
- Adding more Resnet blocks might improve the results. Here I use 6 Resnet blocks, and the result is much better than typical CNN (without residual terms). If adding up the number of Resnet blocks, the results might be even better.

References

https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix

https://github.com/togheppi/pix2pix

https://hardikbansal.github.io/CycleGANBlog/