Chenrui Xu Individual Final Report

I. Introduction

This project is expected to color black and white pictures. The application of the technology can be used widely, like restoring old images or color some comic pictures. Shared work is the final presentation, final report, literature review, and data preprocessing, determining which model to use.

II. Description of individual work

In this project, my work is to use conditional GANs as a solution to solve pixel to pixel problems. The mainly idea of the method is inputting three-channels gray image to an Unet model (which is the generator in this case) and outputting a three-channels color image, and the discriminator used patch GAN method.

III. Details

Preprocessing:

Note that OpenCV shows the image in BGR format, we read the image from google drive and use cv2.cvtColor to convert BGR to RGB. Then we divided the process into two parts. For the first part, we want to get the color image, so we turn the data into image format. We used Image.fromarray() to get the image-format RGB dataset. For the second part, we used cv2.cvtColor several times to finish the process from BGR-RGB-Gray (Gray in RGB format). Different from other black and white pictures, there are three channels instead of one (converting from Gray to RGB, three channels are all gray) so that it would keep the same channels with the RGB image. Also, the data was turned into image format. And at last, we transformed both of them into tensors.

Model

The model was inspired from (https://github.com/eriklindernoren/PyTorch-GAN/blob/master/implementations/pix2pix/models.py). The Generator model we used here is a Unet. Unet is an encoder-decoder with skip connections between mirrored layers in the encoder and decoder stacks. It can be divided into two parts. First, we use convolution neural networks to do down-sampling, then extract layer after layer of features, use this layer after layer of features, and then perform up-sampling, and finally get an image with each pixel corresponding to its type. The beginning input had three channels, then the numbers of filters were 64, 128, 256, 512, 512, 512, 512. That is the structure for the Unet down process. The up structure is from 512 back to 64 filters by using the ConvCompose2D function. Then the output of the model are images with 3 channels.

```python
class UNetDown(nn.Module):
    def __init__(self, in_size, out_size, normalize=True, dropout=0.0):
        super(UNetDown, self).__init__()
        layers = [nn.Conv2d(in_size, out_size, 4, 2, 1, bias=False)]
        if normalize:
            layers.append(nn.InstanceNorm2d(out_size))
        layers.append(nn.LeakyReLU(0.2))
        if dropout:
            layers.append(nn.Dropout(dropout))
        self.model = nn.Sequential(*layers)
    def forward(self, x):
        return self.model(x)
class UNetUp(nn.Module):
    def __init__(self, in_size, out_size, dropout=0.0):
        super(UNetUp, self).__init__()
        layers = [nn.ConvTranspose2d(in_size, out_size, 4, 2, 1, bias=False),nn.InstanceNorm2d(out_size),nn.ReLU(inplace=True
        if dropout:
            layers.append(nn.Dropout(dropout))
        self.model = nn.Sequential(*layers)
    def forward(self, x, skip_input):
        x = self.model(x)
        x = torch.cat((x, skip_input), 1)
        return x
```



```python
class GeneratorUNet(nn.Module):
    def __init__(self, in_channels=3, out_channels=3):
        super(GeneratorUNet, self).__init__()
        self.down1 = UNetDown(in_channels, 64, normalize=False)
        self.down2 = UNetDown(64, 128)
        self.down3 = UNetDown(128, 256)
        self.down4 = UNetDown(256, 512, dropout=0.5)
        self.down5 = UNetDown(512, 512, dropout=0.5)
        self.down6 = UNetDown(512, 512, dropout=0.5)
        self.down7 = UNetDown(512, 512, dropout=0.5)
        self.down8 = UNetDown(512, 512, normalize=False, dropout=0.5)
        self.up1 = UNetUp(512, 512, dropout=0.5)
        self.up2 = UNetUp(1024, 512, dropout=0.5)
        self.up3 = UNetUp(1024, 512, dropout=0.5)
        self.up4 = UNetUp(1024, 512, dropout=0.5)
        self.up5 = UNetUp(1024, 256)
        self.up6 = UNetUp(512, 128)
        self.up7 = UNetUp(256, 64)
        self.final = nn.Sequential(
            nn.Upsample(scale_factor=2),
            nn.ZeroPad2d((1, 0, 1, 0)),
            nn.Conv2d(128, out_channels, 4, padding=1),
            nn.Tanh(),)
```

The discriminator is Markovian discriminator which we can also call patch GAN. Patch GAN will get a predicted value of each patch. The discriminator will judge each patch instead of judging the whole area. Finally, we take the average value as the final output of it. The advantage of the method is that it is much faster than judging the whole image and it let the entire Pix2pix frame have no limitation on the image size, increasing the ability of the extension of the frame.

Evaluation

To evaluate the result, there are two parts of the total generator loss.

$$G^* = \arg\min_G \max_D \mathcal{L}_{cGAN}(G, D) + \lambda \mathcal{L}_{L1}(G).$$

The Generator loss is the MSE loss of generated images,

$$\begin{aligned}\mathcal{L}_{cGAN}(G, D) =& \mathbb{E}_{x,y}[\log D(x, y)] + \\ & \mathbb{E}_{x,z}[\log(1 - D(x, G(x, z))],\end{aligned}$$

and the loss for the pixelwise is the L1 loss.

$$\mathcal{L}_{L1}(G) = \mathbb{E}_{x,y,z}[\|y - G(x, z)\|_1].$$

The meaning for the L1 part is to constraint the difference between fake images and real images. Also, the reason we didn't use L2 distance is that after optimizing, the plot can be more blurry. So, the total generator loss is the MSE loss plus the pixel-wise loss time 100 (which is referred to as lambda pixel). In terms of the discriminator, it would get the whole loss of GAN as big as possible while in terms of the generator, the part

$$\mathbb{E}_{x,z}[\log(1 - D(G(x, z))]$$

should be as small as possible. So here is the final objective:

$$G^* = \arg\min_G \max_D \mathcal{L}_{cGAN}(G, D) + \lambda \mathcal{L}_{L1}(G).$$

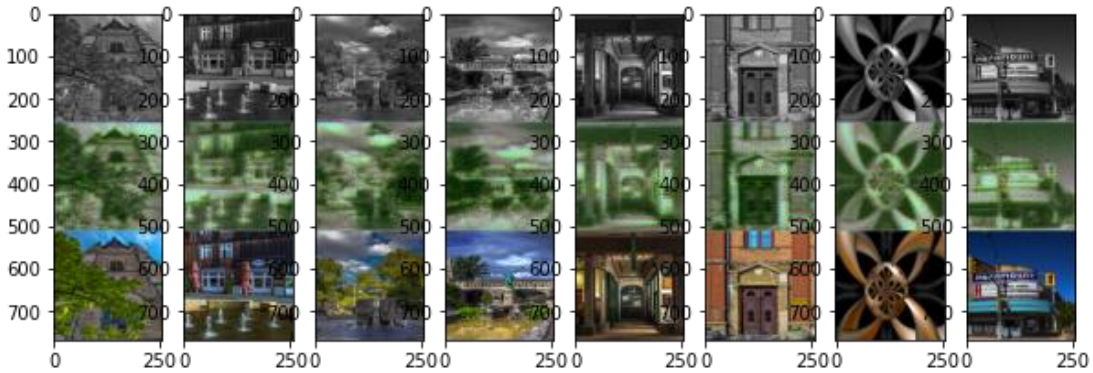
```
loss_G_plot=[]
loss_D_plot=[]
for epoch in range(epoch,n_epochs):
  for i, batch in enumerate(dataloader):
    # Model inputs
    real_A = Variable(batch["B"].type(Tensor))
    real_B = Variable(batch["A"].type(Tensor))
    # Adversarial ground truths
    valid = Variable(Tensor(np.ones((real_A.size(0), *patch))), requires_grad=False)
    fake = Variable(Tensor(np.zeros((real_A.size(0), *patch))), requires_grad=False)
    #Train Generators
    optimizer_G.zero_grad()
    # GAN loss
    fake_B = generator(real_A)
    pred_fake = discriminator(fake_B, real_A)
    loss_GAN = criterion_GAN(pred_fake, valid)
    # Pixel-wise loss
    loss_pixel = criterion_pixelwise(fake_B,real_B)
    # Total loss
    loss_G =lambda_pixel * loss_pixel+loss_GAN
    loss_G_plot.append(loss_G)
    loss_G.backward()
    optimizer_G.step()
```

IV. Results

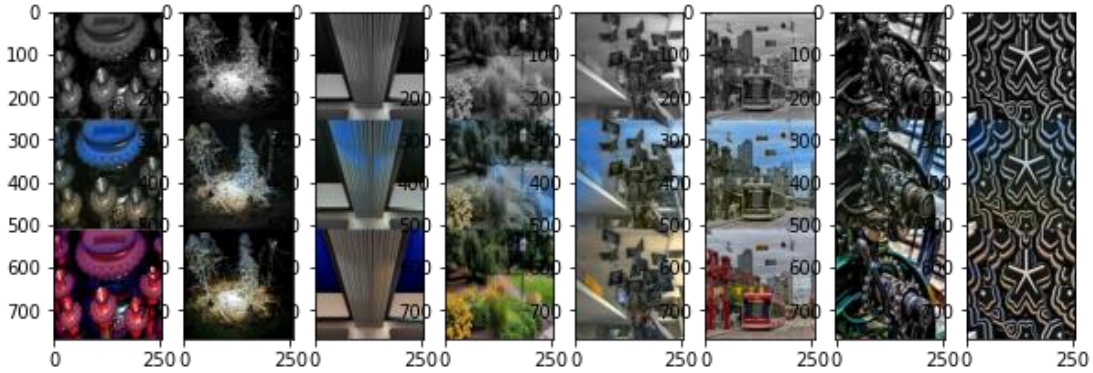
For each picture, the first row are the images we input, which are 3-channels gray images. The second ones are images that the generator made. And the third row are real images.

Results at the beginning of training.

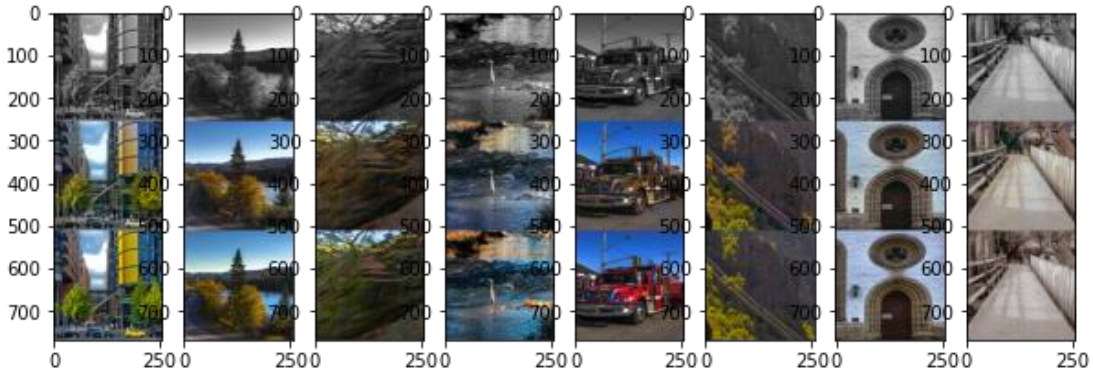
From the beginning, it is not hard to find that there is a lot of noise in those images while it can be seen as the overlap of noisy data plus gray images.



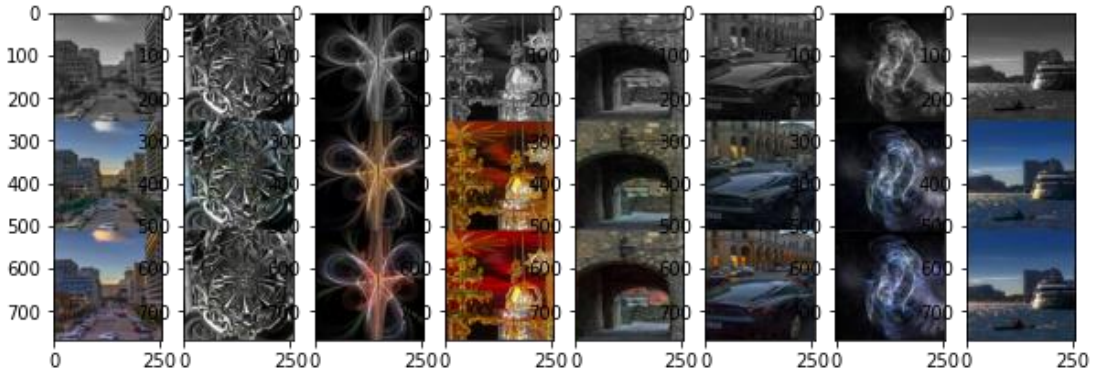
The results after training 5 times.
Things became better as we can see some blue pixels can be generated. However, except for the sky, generators may color some areas blue which are not supposed to be blue.



The results after training 40 times.
Some pictures are good enough (for some landscape images as they only have yellow, green, blue), but for some rare colors like red, the generator can hardly color their areas rightly.

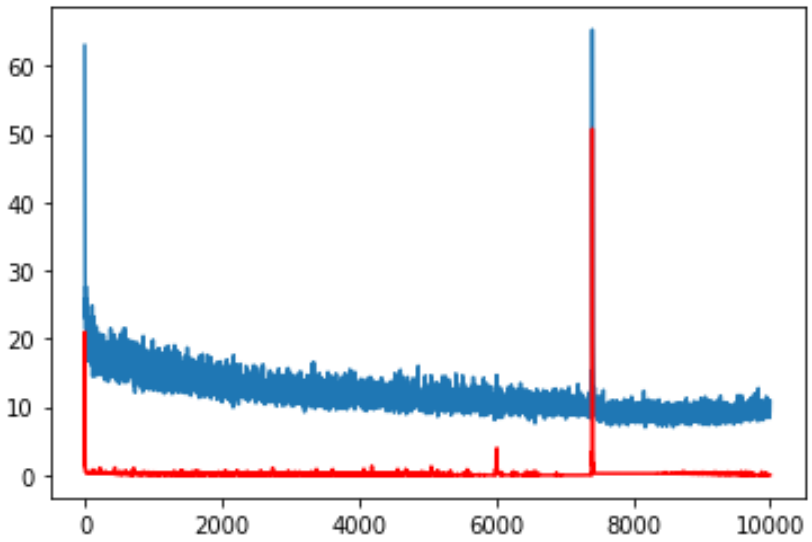


The results after training 100 times.
All fake images are good as the most rare color red can be colored well. And it is hard to distinguish the difference between real and fake images for the training set.

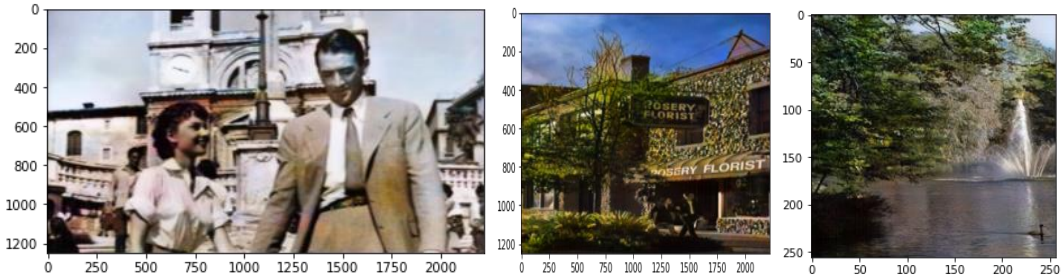


The loss function: (Red is discriminator loss and blue is generator loss).

The overall trend of the loss is stable except for some special points. The special point means it is a picture that has colors that haven't appeared before so the loss can be very large. The overall trend is great as after several times, the loss decreased sharply and trends to be stable afterward.



Result of Pix2Pix (Test):



Roman holiday screenshot is an image with people, so the result is not as good as landscape images, but overall, there are some colors colored, like clothes and the building. In terms of landscapes, the result can be great, although some colors were wrong, overall, there is only a small difference between real and fake one.

V. Summary and conclusions

Pixel to pixel method overall is very great as we can see during the training process, it is hard for us to tell the difference between fake and true images after 80 epochs (totally 100). For the test part, in terms of landscape images, the generator can get pictures well while people pictures didn't show great results. The reason is that there are only several people images in the training dataset. For extension, I can train more categories of images so that the model can color more general images. Also, I only trained the model 100 epochs with 100 iterations. If given a longer time, the result can be better if I use more time and a better GPU. If the method is used to color an old movie, some same topic movie should be trained instead of some regular pictures.

What I learned in this project is how to pursue a GAN model in real life, also some detailed methods like Unet, patch GAN, some data processing methods. Also, it helped me solidify my PyTorch knowledge.

VI. Code

Original code around 380 rows. Regardless of space lines and some explanation lines, there are around 330 rows of code. I kept the model part and loss calculation part originally (around 120). So, the percentage should be around 1-120/(330+50)=69%.

VII. Reference

Image to Image translation:
https://openaccess.thecvf.com/content_cvpr_2017/papers/Isola_Image-To-Image_Translation_With_CVPR_2017_paper.pdf
Code: https://github.com/eriklindernoren/PyTorch-GAN/blob/master/implementations/pix2pix/models.py