Final Report - Image Colorization with GAN

Group 4: Jialei Chen, Chenrui Xu, Ruijin Jia

Indroduction

The aim of the project is helping black and white pictures to get colors. This technology could be further applied in real life, like restoring old images. Moreover, we could also apply it into the 2D animation production industry that helps complete the work which needs the amount of the mutual coloring in the short time.

Dataset Description

Dataset used for this project:Flickr1024(https://yingqianwang.github.io/Flickr1024/). Flickr1024 dataset is a large-scale stereo image dataset consisting 1024 high-quality image pairs and covering diverse scenarios like animals, buildings, people, plants. The size of the dataset is 2.64GB. And to improve the model, we may introduce more pictures for training in the meanwhile.

Deep Learning Network and Training Algorithm

Generative Adversarial Networks, or GANs for short, are generative models. Using deep learning methods, GANs generate new data instances to resemble the training data. The application of GANs model includes creating new images, creating new versions of those images never existed before. G is for Generative, which means taking an input as a random noise signal and then outputs an image. A for Adversarial, which is the discriminator, assessing the images generated whether are similar to what it has been trained on. And N is for network.

We transformed photos through LAB method. The L channel contains information for the light sensitivity of a photo, and is equivalent to a black and white version. A and B are the color channels where A controls the green-red tradeoff and B controls the blue-yellow tradeoff. L channel is conducted as the input, and new A and B color channels are of the output.

Experimental setup

GAN based on L*a*b* color space

Input parameters:

- workers(number of worker threads for loading the data with DataLoader): 2
- batch size(batch size used in training): 2-32
- image size(the spatial size of the images used for training): 2
- nc(number of color channels in the input images): 3
- nz(length of latent vector): 128
- ndf(size of feature maps in discriminator): 64
- epochs(number of training epochs): 100-300
- learning rate: 0.0001
- beta1(hyperparameter for Adam optimizers):0.5
- ngpu(number of GPUs available): 1

Preprocessing:

Read the image information in RGB format by using OpenCV - cv2.imread, note that OpenCV shows the image in BGR format. And then resize the image to (256, 256). Due to the computation power, we don't try larger sizes. Then use OpenCV to transform the image from RGB format to L*a*b* format. After transforming to L*a*b* format, the dataset is ready for training.

Model:

The Generator is a network of encoder and decoder structure. Start with a convolutional layer of 32 filters, 64 filters and finally 128 filters. Then use 128 filters to 128 filters layers to get the network to learn more information inside the image. Finally use Convolutional transpose layer the turn the result into 2 sequences of data, which is a*b* channels. The discriminator is a classical CNN network. Pretrained model also works. We tried resnet 18 for discriminator as well.

Evaluation:

To evaluate the result, the Generator loss is the cross entropy loss of generated images plus MSE of ab channels. And the Discirmation loss is the average of the loss of generated images and the loss of real images.

Additional Method - Image Translation

After we realized the LAB method, we wanted to pursue another method. The method is called “Pix2Pix” GAN. In this case, we apply conditional GANs. Conditional adversarial networks is an approach for direct image-to-image transformation. In this part, we used a “U-Net”-based architecture for our generator, and for our discriminator we used a convolutional “PatchGAN” classifier, which only penalizes structure at the scale of image patches.

Input parameters:

- workers(number of worker threads for loading the data with DataLoader): 4
- batch size(batch size used in training): 16
- image height: 256
- image width: 256
- channels for the image is 3

- epochs(number of training epochs): 100
- learning rate: 0.0005
- b1(hyperparameter for Adam optimizers):0.5
- b2(hyperparameter for Adam optimizers):0.999
- checkpoint interval (will be used during the training to show some training situation)

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 method 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.cutColor again to finish the process from BGR-RGB-Gary. Different from other black and white pictures, there are three channels instead of one so that it would keep the same dimensions with the RGB image. Also, the data was turned into image format. And at last, we transformed both of them into tensors.

Model:

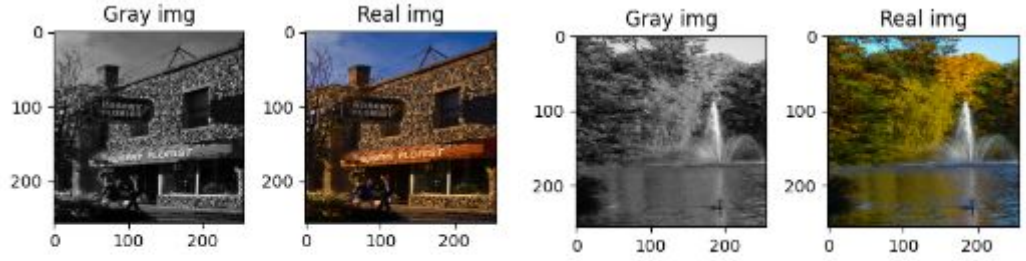
The Generator model we used here is a Unet. It can be divided into two parts. First, we use convolution neural networks to do down-sampling, then extract layer after layer of features, use this layer after layer of features, and then perform up-sampling, and finally get an image with each pixel corresponding to its type. The beginning input had three channels, then the number of filters are 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 ConvCompose function. Then the output of the model are images with 3 channels. The discriminator is Markovian discriminator which we can also call it patch GAN. By using this method, we only need to focus on some small areas instead of the whole image. We used it based on the CNN model.

Evaluation:

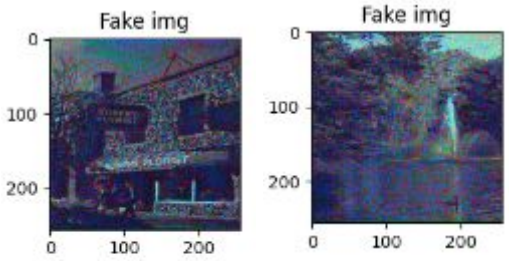
To evaluate the result, the Generator loss is the MSE loss of generated images and the loss for the pixelwise is the L1 loss. So the total generator loss is the MSE loss plus the pixlwise loss time 100 (which is referred as lambda pixel).And the Discirmation loss is the weighted sum of the real loss and fake loss. These two losses can be calculated in the same way with GAN loss.

Results

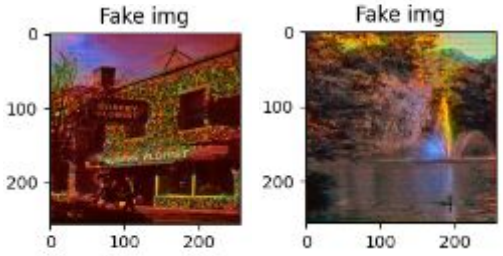
GAN based on L*a*b* color space:
The following figure shown that the result from 1 epohes to 300 epoches
The original photo is shown below, the left are the photo with only L* channel (input of the generator) and the right are the original photo



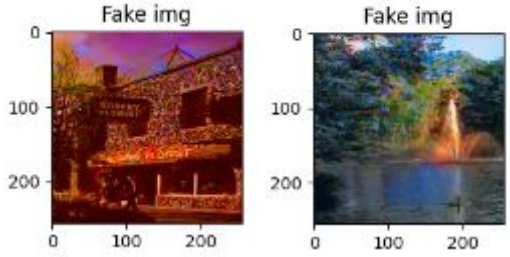
The result of 1 epochs training is shown below. Model learn a little bit of a*b* channel but there is still a lot of noisy data inside the image.



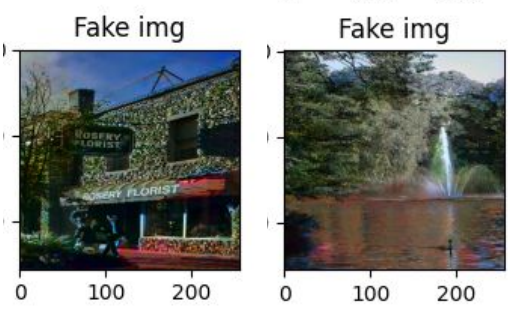
After 2 epochs training, the result is much better than the first epoch. There is still some noisy data in the image but not that much like the first epoch.



When it comes to 50 epochs, there is no more noisy data in images. But the color is still not as good as real images.



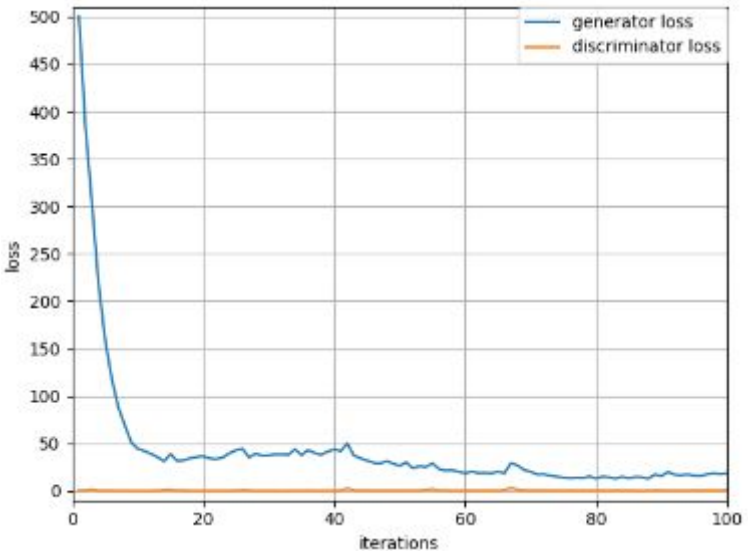
Here is the result for 100 epochs training, the color is very similar to the real image.



After 200 epochs training, the color of images are much better than previous. And even better than the original image.

The result of 300 epochs training is shown below, images are very close to the original one.

The loss function of L*a*b* GAN is shown below. The training processing is stable. When training after 10 epochs, the loss goes down to below 50 and then keeps decreasing.

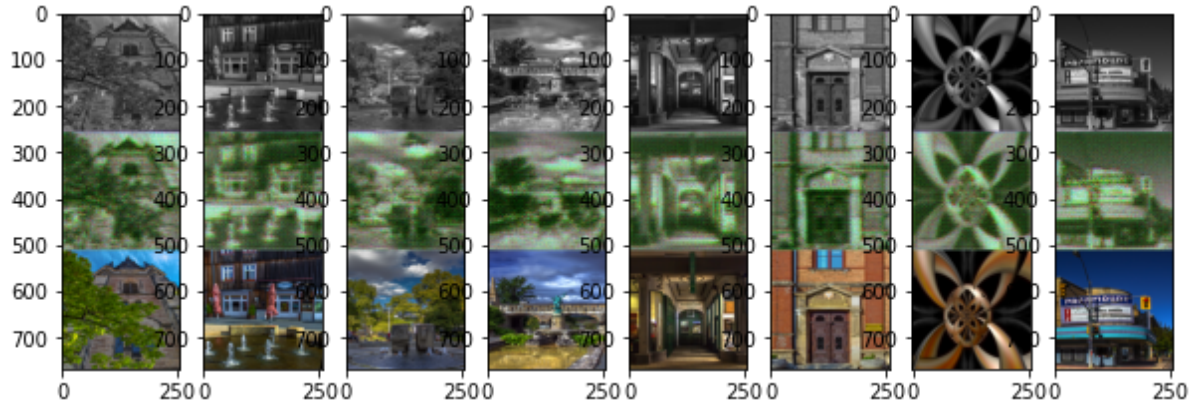


Pix2Pix GAN:

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

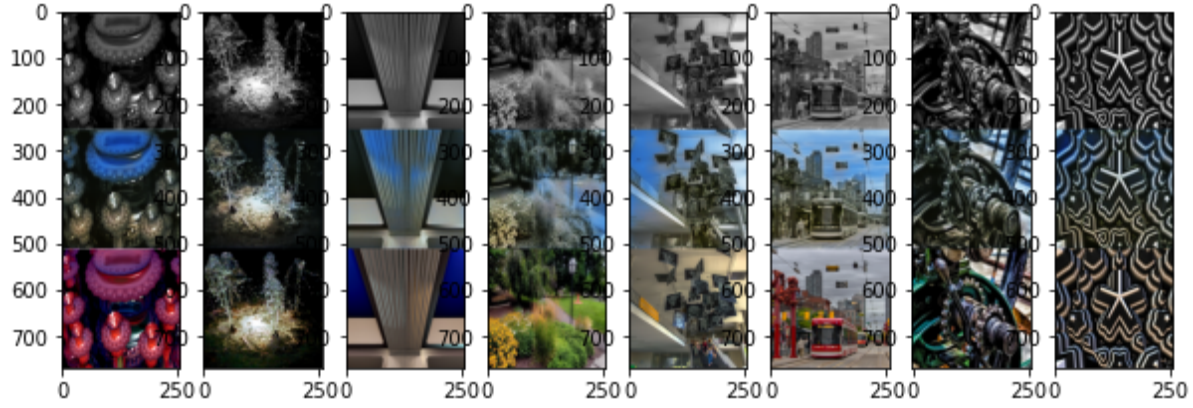
Results at the beginning of training.

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.



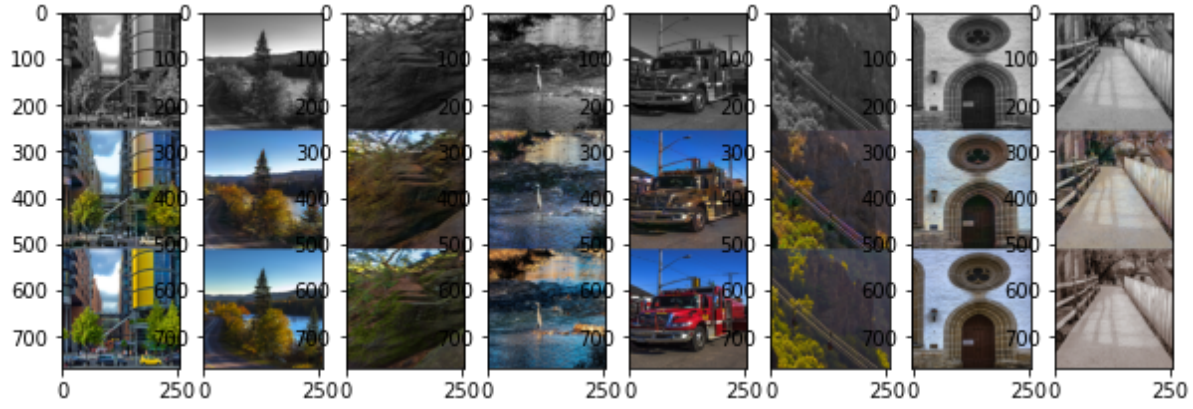
The results after training 5 times.

Things became better as we can see some blue pixels can be generated. However, excecpt sky, generator may color some area blue which are not supposed to be blue.



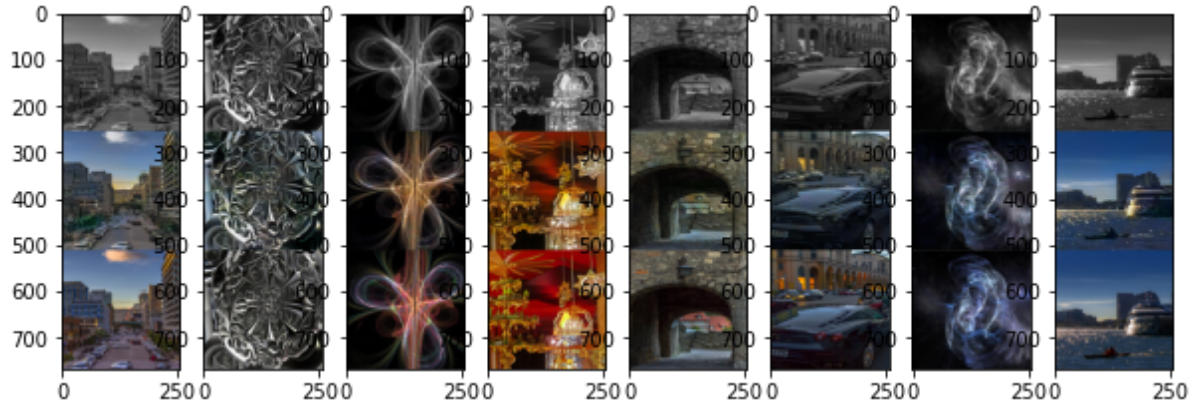
The results after training 40 times.

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



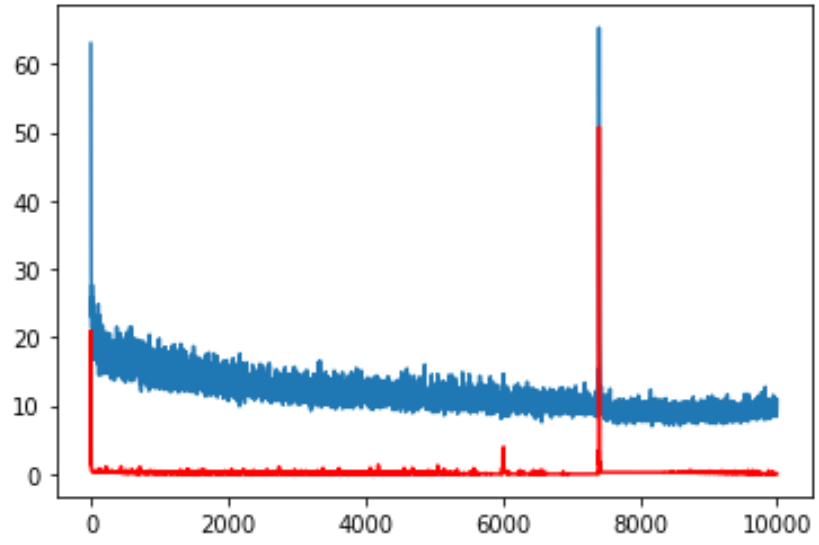
The results after training 100 times.

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 of P2P GAN: (Red is discriminator loss and blue is generator loss)

The overall trend of the loss is stable except for some special points.



Performance comparison

Result of L*a*b* GAN:

Result of Pix2Pix GAN:

Summary and Conclusions

In this study, we were able to automatically colorize grayscale images using GAN, to an acceptable visual degree. With the Flickr1024 dataset, the model was able to consistently produce better looking (qualitatively) images than real images. We obtained mixed results when colorizing grayscale images using the roman holiday screenshot. Mis-colorization was a frequent occurrence with images containing high levels of textured details. This leads us to believe that the model didn’t learn enough information about humans. In addition, this network was not as well-trained as due to the high resolution of images. We expect the

results will improve if the network is trained further. We would also need to seek a better quantitative metric to measure performance. This is because all evaluations of image quality were qualitative in our tests. Thus, having a new or existing quantitative metric such as peak signal-to-noise ratio (PSNR) and root mean square error (RMSE) will enable a much more robust process of quantifying performance.

References

Image colorization: https://arxiv.org/abs/1803.05400

Deep Convolutional GAN: https://arxiv.org/abs/1511.06434

Image to Image translation:

https://openaccess.thecvf.com/content_cvpr_2017/papers/Isola_Image-To-Image_Translation_With_CVPR_2017_paper.pdf

Appendix

Github link: https://github.com/olokojoh/Final--Project-Group4

GAN based on L*a*b* color space

Generator structure:

Encoder part:

```
self.main = nn.Sequential( # Sequential,
    nn.ReflectionPad2d((40, 40, 40, 40)),
    nn.Conv2d(1,32,(9, 9),(1, 1),(4, 4)),
    nn.BatchNorm2d(32),
    nn.ReLU(),
    nn.Conv2d(32,64,(3, 3),(2, 2),(1, 1)),
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.Conv2d(64,128,(3, 3),(2, 2),(1, 1)),
    nn.BatchNorm2d(128),
    nn.ReLU(),
```

State part:

```
nn.Sequential( # Sequential,
    LambdaMap(lambda x: x, # ConcatTable,
        nn.Sequential( # Sequential,
            nn.Conv2d(128,128,(3, 3)),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(128,128,(3, 3)),
            nn.BatchNorm2d(128),
            ),
        shave_block(2),
        ),
    LambdaReduce(lambda x,y: x+y), # CAddTable,
    ),
```

Decoder part:

```
nn.ConvTranspose2d(128,64,(3, 3),(2, 2),(1, 1),(1, 1)),
nn.BatchNorm2d(64),
nn.ReLU(),
nn.ConvTranspose2d(64,32,(3, 3),(2, 2),(1, 1),(1, 1)),
nn.BatchNorm2d(32),
nn.ReLU(),
nn.Conv2d(32,2,(9, 9),(1, 1),(4, 4)),
nn.Tanh(),
```

Discriminator structure: This is a CNN structure. To let the network get information as much as possible. I remove the pooling layer and use the stride of 2 to reduce the dimension of data.

```
class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is (nc) x 256 x 256
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf) x 128 x 128
            nn.Conv2d(ndf, ndf*2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf*2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf) x 64 x 64
            nn.Conv2d(ndf*2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*2) x 16 x 16
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*4) x 8 x 8
            nn.Conv2d(ndf * 8, ndf * 16, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 16),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*8) x 4 x 4
            nn.Conv2d(ndf * 16, ndf * 32, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 32),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*8) x 1 x 1
            nn.Conv2d(ndf * 32, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )
```

Training process:

```
for epoch in range(num_epochs):
    for i, (l, ab) in enumerate(train_dataloader):

      valid = Variable(torch.Tensor(l.size(0), 1).fill_(random.uniform(0.9,1)),
                       requires_grad=False).to(device)
      fake = Variable(torch.Tensor(l.size(0), 1).fill_(random.uniform(0,0.1)),
                       requires_grad=False).to(device)

      lvar = Variable(l).to(device)
      abvar = Variable(ab).to(device)
      real_imgs = torch.cat([lvar, abvar], dim=1)
      # print(lvar.shape)
      # break
      optimizerG.zero_grad()
      abgen = netG(lvar)
      # Generate a batch of images
      gen_imgs = torch.cat([lvar.detach(), abgen], dim=1)

      # Loss measures generator's ability to fool the discriminator
      g_loss_gan = criterion(netD(gen_imgs), valid)
      g_loss = g_loss_gan + pixel_loss_weights * torch.mean((abvar - abgen)**2)
      plt_g_loss.append(g_loss)

      if i % g_every == 0:
        g_loss.backward()
        optimizerG.step()

      optimizerD.zero_grad()
      # Measure discriminator's ability to classify real from generated samples
      real_loss = criterion(netD(real_imgs), valid)
      fake_loss = criterion(netD(gen_imgs.detach()), fake)
      d_loss = (real_loss + fake_loss) / 2
      plt_d_loss.append(d_loss)
      d_loss.backward()
      optimizerD.step()
```

Pix2Pix GAN

Unet function:

```
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
```


Model structure:

```
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(),)
```

Discriminator structure:

```python
class Discriminator(nn.Module):
    def __init__(self, in_channels=3):
        super(Discriminator, self).__init__()
        def discriminator_block(in_filters, out_filters, normalization=True):
            """Returns downsampling layers of each discriminator block"""
            layers = [nn.Conv2d(in_filters, out_filters, 4, stride=2, padding=1)]
            if normalization:
                layers.append(nn.InstanceNorm2d(out_filters))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            return layers
        self.model = nn.Sequential(
            *discriminator_block(in_channels * 2, 64, normalization=False),
            *discriminator_block(64, 128),
            *discriminator_block(128, 256),
            *discriminator_block(256, 512),
            nn.ZeroPad2d((1, 0, 1, 0)),
            nn.Conv2d(512, 1, 4, padding=1, bias=False))
    def forward(self, img_A, img_B):
        # Concatenate image and condition image by channels to produce input
        img_input = torch.cat((img_A, img_B), 1)
        return self.model(img_input)
```

Training Process:

```python
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()
```