Deep Learning project ENSAE 2020 Spring
Generative Adversarial Networks and Cycle-GAN

Jing TAN
ENSAE
jing.tan@ensae.fr

Yannick LY
ENSAE
yannick.ly@ensae.fr

Abstract

Generative adversarial networks have gained great popularity since it was proposed by Goodfellow in 2014. Other related researches into the mathematical basis and deep neural network have made the range of GAN application more and more extensive. The most remarkable applications of GAN are in the generation of image, music, videos, etc. Besides, the latest expansion in cross-domain image transformation attracts a lot of attention. In this report, we will firstly review some theoretical papers in GAN. Then we will present our implementation of DC GAN and Cycle-GAN with different data sets.

1 Bibliographical review

1.1 Generative Adversarial Networks

The Generative adversarial networks consist of two neural networks contest with each other in a game. It highlights the idea of adversarial training, i.e. guiding the generator with a discriminator. A generator learns to generate new data while the discriminator criticise generated data by comparing to the origins. The design of GAN can be summarized by the following structure.

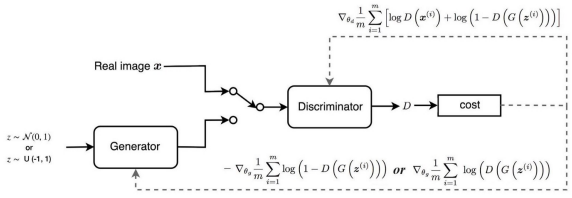


Figure 1: GAN structure
(Goodfellow et al., 2014)

However, training GAN is notorious for problems such as mode collapse, vanishing gradient, non-convergence and other problems caused by the unbalanced discriminator and generator. The improvement has been done in three main directions:

- **Completing the network structure design.** For instance, adding conditions and other information on the generator, using convolutional networks in the generators and discriminators and improving the deep neural network by joining attention, normalising weights, etc.
- **Changing the objective function.** In the original GAN model, the discriminator can be easily trained and optimised. However, the generator frequently encounters diminishing gradient problem when the generated image has distribution far away from the ground truth. Some propose alternative cost functions and add noise to the discriminator, others use Wasserstein distance to smooth the gradient which can also provide with a better training stability.
- **Improving optimisation method.** To illustrate, the Unrolled GAN uses the cost function calculated in different steps to perform the back-propagation for the generator and discriminator. This is similar to the LSTM while the look-ahead of costs avoid the generator from collapsing to local optimal.

1.2 Deep Convolutional GAN

A DC GAN is a GAN that uses a convolutional neural network as the discriminator, and a network composed of transposed convolutions as the generator. The following figure gives an example of the DC GAN generator. In this example, the generator is composed of four fractionally-strided convolutional layers and no fully connected or pooling layers are used.

The main modifications in the architecture compared to former works are:

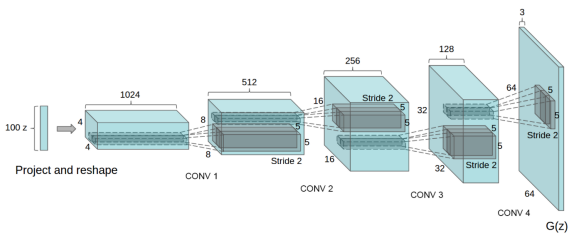


Figure 2: Example of DC GAN generator (Radford et al., 2015)

- All convolutional net. The pooling layers of the discriminator are replaced with strided convolutions and those of the generator are replaced by fractional-strided convolutions.

- Elimination of fully connected layers. The noise vector Z as input of the generator is fully connected but the result is reshaped into a 4-dimensional tensor and used as the start of the convolution stack.

- Batch normalisation. It helps improve training stability and prevent mode collapse. Thus it's applied directly to all layers except for the generator output layer and the discriminator input layer.

- ReLU activation is used in the generator except for the output layer which uses the Tanh function. Within the discriminator LeakyReLU are applied. LeakyReLU has a small slope for negative values. It fixes the "dying ReLU" problem caused by zero-slope.

It's demonstrated by the experiment of *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks* (Radford et al., 2015) that the DC GAN learns a hierarchy representation from object parts to scenes of images. It generates and learns features which are applicable for unsupervised learning tasks.

1.3 Wasserstein GAN

The major difference between the typical GAN and WGAN is the objective function. As mentioned in the section of GAN, the problem of gradient vanishing comes mostly from the cost function. Wasserstein GAN proposes an alternative cost function using Wasserstein distance.

Theorem 3. Let $\mathbb{P}_r$ be any distribution. Let $\mathbb{P}_\theta$ be the distribution of $g_\theta(Z)$ with Z a random variable with density p and g_θ a function satisfying assumption 1(limit of local Lipschitz constant). Then, there is a solution $f : X \to \mathbb{R}$ to the problem

$$\max_{\|f\|_L \leq 1} \mathbb{E}_{x \sim \mathbb{P}_r}[f(x)] - \mathbb{E}_{x \sim \mathbb{P}_\theta}[f(x)]$$

and we have

$$\nabla_\theta W(\mathbb{P}_r, \mathbb{P}_\theta) = \mathbb{E}_{z \sim p(z)}[\nabla_\theta f(g(z))]$$

when both terms are well-defined. Now it comes to optimising the function f where f is 1-Lipschitz. For the purpose of stability, we train the model weights w constrained on a compact space and update the critic f_w. The network design of WGAN is almost the same as GAN except that the critic does not have an output sigmoid function.

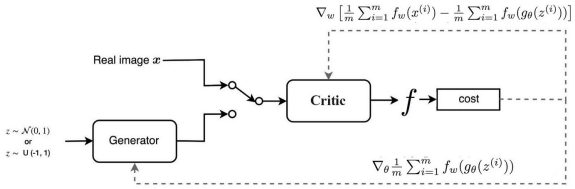


Figure 3: WGAN structure

The objective function proposed by WGAN has a smoother gradient no matter how far the distribution of real data is from that of the fake data. In other words, WGAN still learns when the generator doesn't work well.

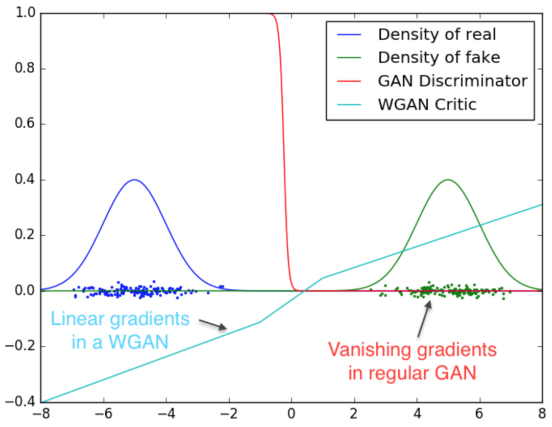


Figure 4: Gradients when learning to differentiate two Gaussians (Arjovsky et al., 2017)

The figure shows the gradients of original GAN

and WGAN. For GAN, the generator has large areas with diminishing or exploding gradients. For WGAN, the critic's gradient is smoother and it has a larger space of clean gradients.

Contrary to GAN, the loss function of WGAN measures how well the generator works as well as the image quality. However, M. Arjovsky, S. Chintala and L. Bottou tried to enforce the Lipschitz constraint with weight clipping, which means to clamp the weights to a fixed box. This method is simple but has introduced some problems to the training. Hence other methods such as WGAN with gradient penalty are investigated to fix this issue.

1.4 Cycle-Consistent Adversarial Networks

Cycle GAN was developed for image-to-image translation. It allows us to use un-paired training data from different domains X and Y. For example X can be the some photos and Y are Monet's paintings. We can extract features of Monet's work and transform a photo into Monet styled painting while the photos don't have to match the contents of Monet's paintings.

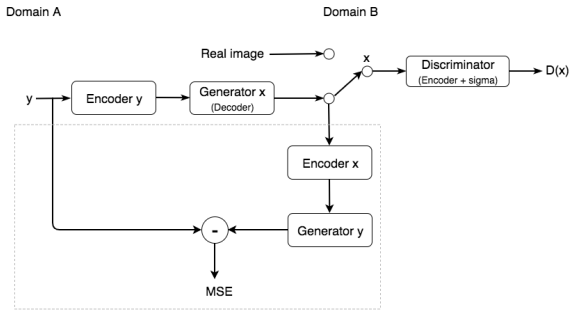


Figure 5: Cycle GAN structure

2 Problem Framing

The code of this project can be divided into two parts: in the first part, we implement a DC GAN. And we train the DC GAN to generate images from random noise. In the second part, we implement a Cycle GAN, which was designed for the task of image-to-image translation.

2.1 Implementation of DC GAN

To implement the DC-GAN, we need to specify three things: 1) the generator, 2) the discriminator, and 3) the training procedure. The discriminator in this DC-GAN is a convolutional neural network. The generator of the DC-GAN consists of a sequence of transpose convolutional layers that progressively upsample the input noise sample to generate a fake image. As mentioned in the previous section, there are some modifications on the networks compared to the original GAN for example, the full convolutional network, leaky ReLU, etc.

```
Discriminator(
  (model): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (3): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (9): InstanceNorm2d(512, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): ZeroPad2d(padding=(1, 0, 1, 0), value=0.0)
    (12): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), padding=(1, 1))
  )
)
```

Figure 6: DC GAN discriminator

Since a DC-GAN is simply a GAN with a specific type of generator and discriminator, we train it in exactly the same way as a standard GAN. The pseudo-code for the training procedure can be found in the appendix.

2.2 Implementation of Cycle GAN

We took the DC GAN discriminator and modified the generators so that they are more suitable for the image translation problem. There are one generator and one discriminator for each domain. It should be noted that we used the same structure for different domains but we can also use different generators and discriminators to deal with data from different domains.

We give un example of the generator of USPS style handwritten script from the MNIST data set. It contains 3 convolutions and 3 up-sampling convolutional layers. Instead of adding only one residual block, we add 9 residual blocks. The choice of number of residual blocks is an arbitrage between the training time of each epoch and training speed to the optimal state. When adding more residual blocks, the training is longer in each epoch but the generator converges faster and provides stable and high-quality images in several epochs. However, with only one residual block, each epoch takes less time but it converges slowly.

Similar to the implementation of DC GAN, the discriminator consists of several convolutional layers with leaky ReLU.

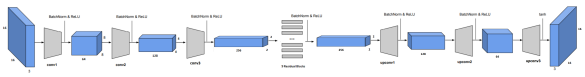


Figure 7: Cycle GAN generator for MNIST2USPS

Once we constructed the generators and discriminators we start to train our model. The figure below shows the training algorithm for the MNIST to USPS experiment. However, for the photo generation from paintings, it is helpful to introduce an additional loss, the identity loss (Taigman et al., 2016), which encourages the mapping to preserve color composition between the input and output. In particular, we regularize the generator to be near an identity mapping when real samples of the target domain are provided as the input to the generator: i.e., $L_{identity}(G, F) = \mathbb{E}_{y \sim p_{data}(y)}[\|G(y) - y\|_1] + E_{x \sim p_{data}(x)}[\|F(x) - x\|_1]$.

Algorithm 2 CycleGAN Training Loop Pseudocode

procedure TRAINCYCLEGAN

Draw a minibatch of samples $\{x^{(1)}, \ldots, x^{(m)}\}$ from domain X

Draw a minibatch of samples $\{y^{(1)}, \ldots, y^{(m)}\}$ from domain Y

Compute the discriminator loss on real images:

$$\mathcal{J}^{(D)}_{real} = \frac{1}{m}\sum_{i=1}^{m}(D_X(x^{(i)}) - 1)^2 + \frac{1}{n}\sum_{j=1}^{n}(D_Y(y^{(j)} - 1)^2$$

Compute the discriminator loss on fake images:

$$\mathcal{J}^{(D)}_{fake} = \frac{1}{m}\sum_{i=1}^{m}(D_Y(G_{X\to Y}(x^{(i)})))^2 + \frac{1}{n}\sum_{j=1}^{n}(D_X(G_{Y\to X}(y^{(j)})))^2$$

Update the discriminators

Compute the $Y \to X$ generator loss:

$$\mathcal{J}^{(G_{Y\to X})} = \frac{1}{n}\sum_{j=1}^{n}(D_X(G_{Y\to X}(y^{(j)})) - 1)^2 + \lambda_{cycle}\mathcal{J}^{(Y\to X\to Y)}_{cycle}$$

Compute the $X \to Y$ generator loss:

$$\mathcal{J}^{(G_{X\to Y})} = \frac{1}{m}\sum_{i=1}^{m}(D_Y(G_{X\to Y}(x^{(i)})) - 1)^2 + \lambda_{cycle}\mathcal{J}^{(X\to Y\to X)}_{cycle}$$

Update the generators

Figure 8: Cycle GAN algorithm

3 Experiments Protocol

3.1 Generating manga characters with DC GAN

Data

The input data to the generator is a vector of normalised random noise. The anime characters sketches are obtained from an open-source database and we cleaned the raw data and retained 23000 images. The cleaned data are available in our github repository.

Results

The discriminator's loss gradually declines. The generator has an increasing loss and it becomes more and more volatile. However the generator's loss doesn't represent the image quality.

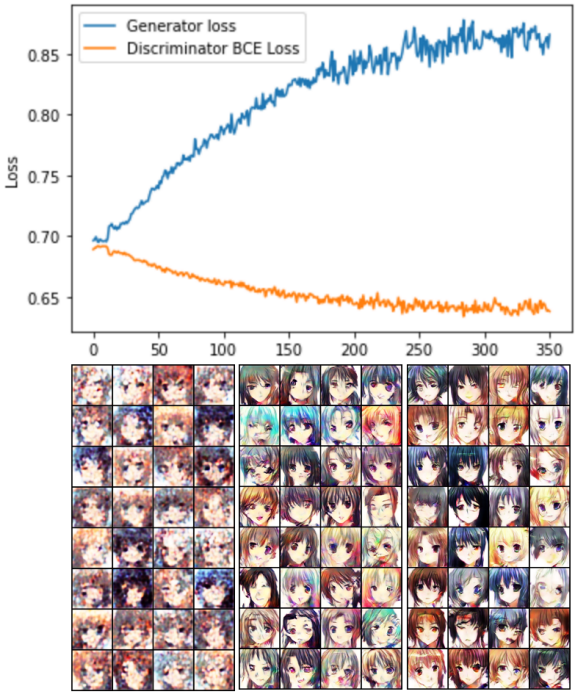


Figure 9: DC GAN Loss

3.2 Translating from MNIST digits to USPS digits with Cycle GAN

Data

We extract the MNIST dataset from torchvision datasets and we downloaded the USPS dataset from kaggle. The raw data are transformed into 16*16 pixel RGB images.

Results

After several epochs we obtain some results as follow. The training loss decreases fast and is optimal after 3 epochs. The generator's loss is stable and it converges rapidly into its optimal. What's more, we can see that the quality of images get gradually improved as training loss are minimised.

In each of the subfigure, we have four lines. Images in the first lines are from MNIST script, in the second lines are USPS style manuscript generated by Cycle GAN. In the third line, there are USPS digits and in the last line there are the MNIST style digits generated by Cycle GAN.

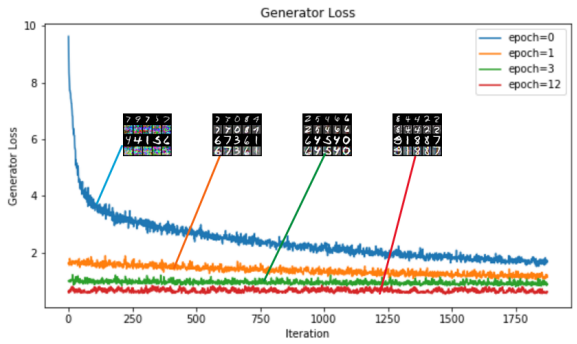
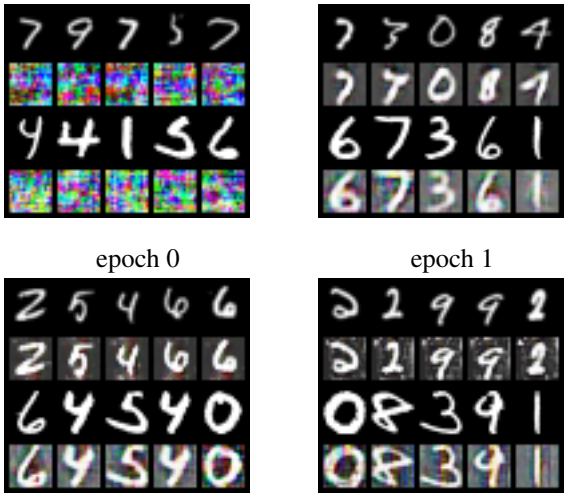


Figure 10: Loss of generator



epoch 0 epoch 1

epoch 3 epoch 13

Figure 11: Cycle GAN MNIST to USPS

3.3 Transforming between photos and Monet paintings with Cycle GAN

Data

We took data used in the paper of Cycle GAN from the author's personal page. The images are resized into 256*256 pixel RGB images and fed to our generator.

Results

In each of the subfigure, we have four lines. The first lines display Monet's original paintings, the second lines contain fake photos converted from Monet's paintings. In the third line, there are photos of landscapes and in the last line are the fake Monet style painting. The results are satisfying even though we didn't have time to run the training for more than 10 epochs.

We ran the code on a local machine with a GPU type NVIDIA RTX 2070. It takes around 4 hours to run 350 epochs for the DC GAN. For the MNIST2USPS, it takes around 5 hours to run 10 epochs and for Monet2Photo 10 hours to run 10 epochs.

epoch 1 epoch 3

epoch 5 epoch 7

Figure 12: Cycle GAN Monet to Photo

4 Discussion

GAN models have gained very much attention in recent years. Theories are enhanced by researchers and various architectures are developed by data engineers. While implementing the GAN strcuture, we have encountered some problems which are not always clarified in the lectures. For example, in the implementation of Cycle GAN should we separate the training of generators and discriminators? The generator works better when there are more than one residual block. This structure avoids vanishing gradient however increases the depth of the network. Shall we apply the Wasserstein distance in the Cycle GAN to deal with vanishing gradient and improve the model performance?

During the implementation, we found that GANs are normally domain-specified. For example, instead of using a normalised noise in the input of DC GAN, it will perform differently if ze had used an alternative noise. And in the Cycle GAN, we can't adopt transfer learning because the solutions are constrained in specific domains. In the future we might develop other interesting architectures which can be more universal.

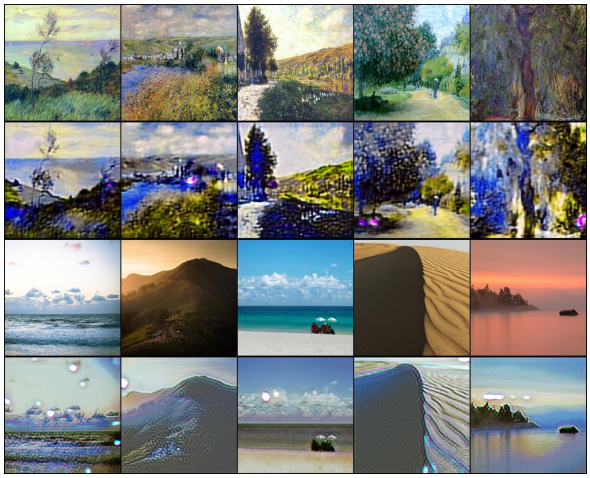


Figure 13: Failure with separately trained generator

References

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc.

Alec Radford, Luke Metz, and Soumith Chintala. 2015. Unsupervised representation learning with deep convolutional generative adversarial networks. Cite arxiv:1511.06434Comment: Under review as a conference paper at ICLR 2016.

Yaniv Taigman, Adam Polyak, and Lior Wolf. 2016. Unsupervised cross-domain image generation. *CoRR*, abs/1611.02200.

Martin Arjovsky, Soumith Chintala, and Léon Bottou. 2017. Wasserstein generative adversarial networks. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 214–223, International Convention Centre, Sydney, Australia. PMLR.

DC GAN algorithm

Algorithm 1 GAN Training Loop Pseudocode

procedure TRAINGAN

Draw m training examples $\{x^{(1)}, \ldots, x^{(m)}\}$ from the data distribution p_{data}

Draw m noise samples $\{z^{(1)}, \ldots, z^{(m)}\}$ from the noise distribution p_z

Generate fake images from the noise: $G(z^{(i)})$ for $i \in \{1, \ldots . m\}$

Compute the (least-squares) discriminator loss:

$$J^{(D)} = \frac{1}{2m} \sum_{i=1}^{m} \left[\left(D(x^{(i)}) - 1 \right)^2 \right] + \frac{1}{2m} \sum_{i=1}^{m} \left[\left(D(G(z^{(i)})) \right)^2 \right]$$

Update the parameters of the discriminator

Draw m new noise samples $\{z^{(1)}, \ldots, z^{(m)}\}$ from the noise distribution p_z

Generate fake images from the noise: $G(z^{(i)})$ for $i \in \{1, \ldots . m\}$

Compute the (least-squares) generator loss:

$$J^{(G)} = \frac{1}{m} \sum_{i=1}^{m} \left[\left(D(G(z^{(i)})) - 1 \right)^2 \right]$$

Update the parameters of the generator

WGAN algorithm

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α, the learning rate. c, the clipping parameter. m, the batch size. n_{critic}, the number of iterations of the critic per generator iteration.

Require: : w_0, initial critic parameters. θ_0, initial generator's parameters.

while θ has not converged **do**

  for $t = 0, ..., n_{\text{critic}}$ **do**

    Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.

    Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.

    $g_w \leftarrow \nabla_w \left[\frac{1}{m} \sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m} \sum_{i=1}^{m} f_w(g_\theta(z^{(i)})) \right]$

    $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$

    $w \leftarrow \text{clip}(w, -c, c)$

  end for

  Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.

  $g_\theta \leftarrow -\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$

  $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$

end while

#CycleGAN applying to MNIST2USPS

In this notebook we are going to implement the CycleGAN architecture for the dataset MNIST to the dataset USPS.

0.1 Setup

[0]:
```
##############################
#  SETUP PYTHON ENVIRONMENT  #
##############################

!pip install torch torchvision
conda update pytorch torchvision
```

[1]:
```
!wget -c https://github.com/YannickLy/DeepLearning-Project-ENSAE-2020/blob/
 ↪master/Data/usps.h5?raw=true -O usps.h5
```

```
--2020-05-11 18:33:08--  https://github.com/YannickLy/DeepLearning-Project-
ENSAE-2020/blob/master/Data/usps.h5?raw=true
Resolving github.com (github.com)... 140.82.118.3
Connecting to github.com (github.com)|140.82.118.3|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://github.com/YannickLy/DeepLearning-Project-
ENSAE-2020/raw/master/Data/usps.h5 [following]
--2020-05-11 18:33:08--  https://github.com/YannickLy/DeepLearning-Project-
ENSAE-2020/raw/master/Data/usps.h5
Reusing existing connection to github.com:443.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/YannickLy/DeepLearning-Project-
ENSAE-2020/master/Data/usps.h5 [following]
--2020-05-11 18:33:08--
https://raw.githubusercontent.com/YannickLy/DeepLearning-Project-
ENSAE-2020/master/Data/usps.h5
Resolving raw.githubusercontent.com (raw.githubusercontent.com)...
151.101.0.133, 151.101.64.133, 151.101.128.133, ...
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|151.101.0.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2913522 (2.8M) [application/octet-stream]
Saving to: ‘usps.h5’

usps.h5             100%[===================>]   2.78M  --.-KB/s    in 0.1s

2020-05-11 18:33:09 (22.7 MB/s) - ‘usps.h5’ saved [2913522/2913522]
```

0.2 Helper functions

[0]:
```
from PIL import Image

import torch
import torchvision.transforms as transforms
from torch.utils.data import Dataset

def init_normal_weights(m):
    classname = m.__class__.__name__
    if classname.find("Conv") != -1:
        torch.nn.init.normal_(m.weight.data, 0.0, 0.02)
        if hasattr(m, "bias") and m.bias is not None:
            torch.nn.init.constant_(m.bias.data, 0.0)
    elif classname.find("BatchNorm2d") != -1:
        torch.nn.init.normal_(m.weight.data, 1.0, 0.02)
        torch.nn.init.constant_(m.bias.data, 0.0)

def print_models(G_XtoY, G_YtoX, D_X, D_Y):

    """ Prints model information for the generators and discriminators. """

    if G_YtoX:
      print("                 G_XtoY                 ")
      print("----------------------------------------")
      print(G_XtoY)
      print("----------------------------------------")

      print("                 G_YtoX                 ")
      print("----------------------------------------")
      print(G_YtoX)
      print("----------------------------------------")

      print("                  D_X                   ")
      print("----------------------------------------")
      print(D_X)
      print("----------------------------------------")

      print("                  D_Y                   ")
      print("----------------------------------------")
      print(D_Y)
      print("----------------------------------------")

class LambdaLR:

    """ LambdaLR or LambdaLearningRate
```

```
        Allow us to decrease the learning rate from a specific epoch␣
↪('decay_start_epoch').
        This accelerates learning and become and allows to be more precise for␣
↪larger epochs.
    """

    def __init__(self, n_epochs, offset, decay_start_epoch):
        self.n_epochs = n_epochs
        self.offset = offset
        self.decay_start_epoch = decay_start_epoch

    def step(self, epoch):
        return 1.0 - max(0, epoch + self.offset - self.decay_start_epoch) /␣
↪(self.n_epochs - self.decay_start_epoch)

class MergedDataset(Dataset):

    """ MergedDataSet
        Allow us to convert grayscale images into 3 channel RGB images.
        And merges two datasets of different styles into one dataset.
    """

    def __init__(self, dataset_X, dataset_Y, transforms_):
        self.transform = transforms.Compose(transforms_)
        self.dataset_X = dataset_X
        self.dataset_Y = dataset_Y

    def __to_rgb(self, image):
        trans = transforms.ToPILImage()
        pil_img = trans(image)
        rgb_image = Image.new("RGB", pil_img.size)
        rgb_image.paste(pil_img)
        return rgb_image

    def __getitem__(self, index):
        image_X = self.dataset_X[index % len(self.dataset_X)][0]
        image_Y = self.dataset_Y[index % len(self.dataset_Y)][0]
        # Convert grayscale images to rgb
        image_X = self.__to_rgb(image_X)
        image_Y = self.__to_rgb(image_Y)
        item_X = self.transform(image_X)
        item_Y = self.transform(image_Y)
        return {"X": item_X, "Y": item_Y}

    def __len__(self):
        return max(len(self.dataset_X), len(self.dataset_Y))
```

0.3 Architectures

[0]:
```
import torch.nn as nn

class ResnetBlock(nn.Module):

    """Defines the architecture of a ResidualBlock
       Note: We decided to choose the number of ResidualBlock we would like to␣
 ↪put in both generators
    """

    def __init__(self, in_features):
        super(ResnetBlock, self).__init__()

        self.block = nn.Sequential(
            nn.ReflectionPad2d(1),
            nn.Conv2d(in_features, in_features, 3),
            nn.InstanceNorm2d(in_features),
            nn.ReLU(inplace=True),
            nn.ReflectionPad2d(1),
            nn.Conv2d(in_features, in_features, 3),
            nn.InstanceNorm2d(in_features),
        )

    def forward(self, x):
        return x + self.block(x)

class Generator(nn.Module):

    """Defines the architecture of the generator network.
       Note: Both generators G_XtoY and G_YtoX have the same architecture in␣
 ↪this assignment.
    """

    def __init__(self, input_shape, num_residual_blocks = 1):
        super(Generator, self).__init__()

        channels, height, width = input_shape

        # Initial convolution block (first convolutional layer)
        out_features = 64
        model = [
            # Pads the input tensor using the reflection of the input boundary
            nn.ReflectionPad2d(channels),
            nn.Conv2d(channels, out_features, 7),
            # Applies Instance Normalization over a 4D input
            nn.InstanceNorm2d(out_features),
```

```
            nn.ReLU(inplace=True),
        ]
        in_features = out_features

        # Two convolution blocks for downsampling
        for _ in range(2):
            out_features *= 2
            model += [
                nn.Conv2d(in_features, out_features, 3, stride=2, padding=1),
                nn.InstanceNorm2d(out_features),
                nn.ReLU(inplace=True),
            ]
            in_features = out_features

        # Residual blocks
        for _ in range(num_residual_blocks):
            model += [ResnetBlock(out_features)]

        # Two convolution blocks for upsampling
        for _ in range(2):
            out_features //= 2
            model += [
                # Upsamples a given multi-channel data. Double the spatial space.
                nn.Upsample(scale_factor=2),
                nn.Conv2d(in_features, out_features, 3, stride=1, padding=1),
                nn.InstanceNorm2d(out_features),
                nn.ReLU(inplace=True),
            ]
            in_features = out_features

        # Output layer
        model += [nn.ReflectionPad2d(channels), nn.Conv2d(out_features,␣
↪channels, 7), nn.Tanh()]

        self.model = nn.Sequential(*model)

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

class Discriminator(nn.Module):

    """Defines the architecture of the discriminator network.
       Note: Both discriminators D_X and D_Y have the same architecture in this␣
↪assignment.
    """

    def __init__(self, input_shape):
```

```
        super(Discriminator, self).__init__()

        channels, height, width = input_shape

        # Calculate output shape of image discriminator
        self.output_shape = (1, height // 2 ** 4, width // 2 ** 4)

        def discriminator_block(in_filters, out_filters, normalize = True):
            layers = [nn.Conv2d(in_filters, out_filters, 4, stride = 2, padding␣
↪= 1)]
            if normalize:
                layers.append(nn.InstanceNorm2d(out_filters))
            layers.append(nn.LeakyReLU(0.2, inplace = True))
            return layers

        self.model = nn.Sequential(
            *discriminator_block(channels, 64, normalize = 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)
        )

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

0.4 CycleGAN

[4]:
```
import os
import sys
import numpy as np
import pandas as pd
pd.options.mode.chained_assignment = None
import itertools
import argparse
import h5py

import torch
import torchvision.transforms as transforms
from torchvision import datasets
from torchvision.utils import save_image, make_grid
from torch.utils import data
from torch.utils.data import DataLoader
from torch.autograd import Variable
from PIL import Image
```

```
##############################
#    MODEL CONFIGURATION     #
##############################

parser = argparse.ArgumentParser()
# data set
parser.add_argument("--dataset_name", type=str, default="MNIST2USPS", help="name␣
 ↪of the dataset")
parser.add_argument("--img_height", type=int, default=16, help="size of image␣
 ↪height")
parser.add_argument("--img_width", type=int, default=16, help="size of image␣
 ↪width")
parser.add_argument("--channels", type=int, default=3, help="number of image␣
 ↪channels")
# cycleGAN parameters
parser.add_argument("--n_residual_blocks", type=int, default=9, help="number of␣
 ↪residual blocks in generator")
parser.add_argument("--lambda_cyc", type=float, default=10.0, help="cycle loss␣
 ↪weight")
parser.add_argument("--lambda_id", type=float, default=5.0, help="identity loss␣
 ↪weight")
parser.add_argument("--lr", type=float, default=0.0002, help="adam: learning␣
 ↪rate")
parser.add_argument("--b1", type=float, default=0.3, help="adam: decay of first␣
 ↪order momentum of gradient")
parser.add_argument("--b2", type=float, default=0.999, help="adam: decay of␣
 ↪first order momentum of gradient")
parser.add_argument("--decay_epoch", type=int, default=10, help="epoch from␣
 ↪which to start lr decay")
# training parameters
parser.add_argument("--cuda", type=bool, default=False, help="change to GPU␣
 ↪mode")
parser.add_argument("--n_epochs", type=int, default=50, help = "number of epochs␣
 ↪of training")
parser.add_argument("--batch_size", type=int, default=32, help="size of the␣
 ↪batches")
parser.add_argument("--n_cpu", type=int, default=0, help="number of cpu threads␣
 ↪to use during batch generation") # 0 by default in windows because␣
 ↪multiprocessing doesn't work
# saving parameters
parser.add_argument("--epoch", type=int, default=0, help = "epoch to start␣
 ↪training from")
parser.add_argument("--sample_interval", type=int, default=1000, help="interval␣
 ↪between saving generator outputs")
```

```
parser.add_argument("--checkpoint_interval", type=int, default=1, help="interval␣
 ↪between saving model checkpoints")
opt = parser.parse_args("")

# GPU option
cuda = torch.cuda.is_available()
if cuda:
    print('Models moved to GPU.')
    opt.cuda = True

# Create sample image, checkpoint and losses directories
os.makedirs("images", exist_ok = True)
os.makedirs("saved_models", exist_ok = True)
os.makedirs("losses_models", exist_ok = True)

# Image transformations
transforms_ = [
    transforms.Resize(int(opt.img_height), Image.BICUBIC),
    # Add some noise into the data
    transforms.RandomCrop((opt.img_height, opt.img_width)),
    transforms.ToTensor(),
]

# MNIST datasets
MNIST_trainset = datasets.MNIST('MNIST',
                                train = True,
                                download = True,
                                transform = transforms.Compose([transforms.
 ↪Resize(opt.img_height), transforms.ToTensor()]))
MNIST_testset = datasets.MNIST('MNIST',
                               train = False,
                               download = True,
                               transform = transforms.Compose([transforms.
 ↪Resize(opt.img_height), transforms.ToTensor()]))

# USPS dataset
with h5py.File('usps.h5', 'r') as hf:
    train = hf.get('train')
    X_train = train.get('data')[:]
    test = hf.get('test')
    X_test = test.get('data')[:]

# The data were in an 1D numpy array of grayscale and normalised between [0,1]
# We transform the data into the same dimensions as for the mnist.
X_train = X_train.reshape(-1, 1, 16, 16)
X_test = X_test.reshape(-1, 1, 16, 16)
```

```
USPS_trainset = data.TensorDataset(torch.Tensor(X_train))
USPS_testset = data.TensorDataset(torch.Tensor(X_test))

# Train dataloader
train_dataloader = DataLoader(
    MergedDataset(MNIST_trainset, USPS_trainset, transforms_ = transforms_),
    batch_size = opt.batch_size,
    shuffle = True,
    num_workers = opt.n_cpu,
)

# Test dataloader
test_dataloader = DataLoader(
    MergedDataset(MNIST_testset, USPS_testset, transforms_ = transforms_),
    batch_size = 5,
    shuffle = True,
    num_workers = opt.n_cpu,
)

##############################
#    MODEL INITIALIZATION    #
##############################

def create_model(opt):

    """ Builds the generators and discriminators. """

    input_shape = (opt.channels, opt.img_height, opt.img_width)

    # Initialize generator and discriminator
    G_XtoY = Generator(input_shape, opt.n_residual_blocks)
    G_YtoX = Generator(input_shape, opt.n_residual_blocks)
    D_X = Discriminator(input_shape)
    D_Y = Discriminator(input_shape)

    print_models(G_XtoY, G_YtoX, D_X, D_Y)

    if opt.cuda:
        G_XtoY = G_XtoY.cuda()
        G_YtoX = G_YtoX.cuda()
        D_X = D_X.cuda()
        D_Y = D_Y.cuda()

    if opt.epoch != 0:
        # Load pretrained models
        G_XtoY.load_state_dict(torch.load("saved_models/G_XtoY_%d.pth" % opt.
 ↪epoch))
```

```
        G_YtoX.load_state_dict(torch.load("saved_models/G_YtoX_%d.pth" % opt.
↪epoch))
        D_X.load_state_dict(torch.load("saved_models/D_X_%d.pth" % opt.epoch))
        D_Y.load_state_dict(torch.load("saved_models/D_Y_%d.pth" % opt.epoch))
    else:
        # Initialize weights
        G_XtoY.apply(init_normal_weights)
        G_YtoX.apply(init_normal_weights)
        D_X.apply(init_normal_weights)
        D_Y.apply(init_normal_weights)

    return G_XtoY, G_YtoX, D_X, D_Y

##############################
#       MODEL TRAINING       #
##############################

def training_loop(train_dataloader, test_dataloader, opt):

    """ Runs the training loop.
        * Saves checkpoint every opt.checkpoint_interval iterations
        * Saves generated samples every opt.sample_interval iterations
    """

    # Create generators and discriminators
    G_XtoY, G_YtoX, D_X, D_Y = create_model(opt)

    # Losses
    loss_GAN = torch.nn.MSELoss()
    loss_cycle = torch.nn.L1Loss()
    loss_identity = torch.nn.L1Loss()

    # Optimizers
    optimizer_G = torch.optim.Adam(itertools.chain(G_XtoY.parameters(), G_YtoX.
↪parameters()), lr = opt.lr, betas = (opt.b1, opt.b2))
    optimizer_D_X = torch.optim.Adam(D_X.parameters(), lr = opt.lr, betas = (opt.
↪b1, opt.b2))
    optimizer_D_Y = torch.optim.Adam(D_Y.parameters(), lr = opt.lr, betas = (opt.
↪b1, opt.b2))

    # Learning rate update schedulers
    LambdaLR_schedular_G = torch.optim.lr_scheduler.LambdaLR(optimizer_G,␣
↪lr_lambda = LambdaLR(opt.n_epochs, opt.epoch, opt.decay_epoch).step)
    LambdaLR_scheduler_D_X = torch.optim.lr_scheduler.LambdaLR(optimizer_D_X,␣
↪lr_lambda = LambdaLR(opt.n_epochs, opt.epoch, opt.decay_epoch).step)
```

```
    LambdaLR_scheduler_D_Y = torch.optim.lr_scheduler.LambdaLR(optimizer_D_Y,␣
↪lr_lambda = LambdaLR(opt.n_epochs, opt.epoch, opt.decay_epoch).step)

    Tensor = torch.Tensor
    if opt.cuda:
        loss_GAN.cuda()
        loss_cycle.cuda()
        loss_identity.cuda()
        Tensor = torch.cuda.FloatTensor

    def sample_images(batches_done):
        """Saves a generated sample from the test set"""
        imgs = next(iter(test_dataloader))
        G_XtoY.eval()
        G_YtoX.eval()
        real_X = Variable(imgs["X"].type(Tensor))
        fake_Y = G_XtoY(real_X)
        real_Y = Variable(imgs["Y"].type(Tensor))
        fake_X = G_YtoX(real_Y)
        # Arange images along x-axis
        real_X = make_grid(real_X, nrow = 5, normalize = True)
        real_Y = make_grid(real_Y, nrow = 5, normalize = True)
        fake_X = make_grid(fake_X, nrow = 5, normalize = True)
        fake_Y = make_grid(fake_Y, nrow = 5, normalize = True)
        # Arange images along y-axis
        image_grid = torch.cat((real_X, fake_Y, real_Y, fake_X), 1)
        save_image(image_grid, "images/%s.png" % batches_done, normalize = False)

    losses_models_G = pd.DataFrame(np.zeros((len(train_dataloader), opt.n_epochs␣
↪- opt.epoch + 1)))
    losses_models_D = pd.DataFrame(np.zeros((len(train_dataloader), opt.n_epochs␣
↪- opt.epoch + 1)))
    losses_models_G.columns = range(opt.epoch, opt.n_epochs+1)
    losses_models_D.columns = range(opt.epoch, opt.n_epochs+1)
    # Training
    for epoch in range(opt.epoch, opt.n_epochs):
        for i, batch in enumerate(train_dataloader):

            # Set model input
            real_X = Variable(batch["X"].type(Tensor))
            real_Y = Variable(batch["Y"].type(Tensor))

            # Adversarial ground truths
            valid = Variable(Tensor(np.ones((real_X.size(0), *D_X.
↪output_shape))), requires_grad = False)
            fake = Variable(Tensor(np.zeros((real_X.size(0), *D_X.
↪output_shape))), requires_grad = False)
```

```
        # -----------------------
        #  Train Discriminator X
        # -----------------------

        optimizer_D_X.zero_grad()

        # Real loss
        loss_real_D_X = loss_GAN(D_X(real_X), valid)

        # Fake loss
        fake_X = G_YtoX(real_Y)
        loss_fake_D_X = loss_GAN(D_X(fake_X), fake)

        # Total loss
        loss_D_X = (loss_real_D_X + loss_fake_D_X) / 2

        loss_D_X.backward()
        optimizer_D_X.step()

        # -----------------------
        #  Train Discriminator Y
        # -----------------------

        optimizer_D_Y.zero_grad()

        # Real loss
        loss_real_D_Y = loss_GAN(D_Y(real_Y), valid)

        # Fake loss
        fake_Y = G_XtoY(real_X)
        loss_fake_D_Y = loss_GAN(D_Y(fake_Y), fake)

        # Total loss
        loss_D_Y = (loss_real_D_Y + loss_fake_D_Y) / 2

        loss_D_Y.backward()
        optimizer_D_Y.step()

        loss_D = (loss_D_X + loss_D_Y) / 2
        losses_models_D[epoch][i] = loss_D

        # ------------------------------
        #  Train Generators XtoY and YtoX
        # ------------------------------

        G_XtoY.train()
```

```
            G_YtoX.train()

            optimizer_G.zero_grad()

            # GAN loss
            fake_Y = G_XtoY(real_X)
            loss_GAN_XtoY = loss_GAN(D_Y(fake_Y), valid)
            fake_X = G_YtoX(real_Y)
            loss_GAN_YtoX = loss_GAN(D_X(fake_X), valid)
            loss_GAN_G = (loss_GAN_XtoY + loss_GAN_YtoX) / 2

            # Cycle loss
            recov_X = G_YtoX(fake_Y)
            loss_cycle_X = loss_cycle(recov_X, real_X)
            recov_Y = G_XtoY(fake_X)
            loss_cycle_Y = loss_cycle(recov_Y, real_Y)
            loss_cycle_G = (loss_cycle_X + loss_cycle_Y) / 2

            # Identity loss
            loss_identity_X = loss_identity(G_YtoX(real_X), real_X)
            loss_identity_Y = loss_identity(G_XtoY(real_Y), real_Y)
            loss_identity_G = (loss_identity_X + loss_identity_Y) / 2

            # Total loss
            loss_G = loss_GAN_G + opt.lambda_cyc * loss_cycle_G + opt.lambda_id␣
↪* loss_identity_G

            loss_G.backward()
            optimizer_G.step()
            losses_models_G[epoch][i] = loss_G

            # --------------
            #  Log Progress
            # --------------

            batches_done = epoch * len(train_dataloader) + i

            # Print log
            sys.stdout.write(
                "\r[Epoch %d/%d] [Batch %d/%d] [D loss: %f] [G loss: %f]"
                % (
                    epoch,
                    opt.n_epochs,
                    i,
                    len(train_dataloader),
                    loss_D.item(),
                    loss_G.item()
```

```
                )
            )

            # Save sample image at interval
            if batches_done % opt.sample_interval == 0:
                sample_images(batches_done)

        # Update learning rates
        LambdaLR_schedular_G.step()
        LambdaLR_scheduler_D_X.step()
        LambdaLR_scheduler_D_Y.step()

        # Save discriminators and generators lossses at each epoch
        losses_models_G.to_pickle("losses_models/losses_models_G_%d" % epoch)
        losses_models_D.to_pickle("losses_models/losses_models_D_%d" % epoch)

        # Save model at checkpoints
        if opt.checkpoint_interval != -1 and epoch % opt.checkpoint_interval ==␣
↪0:
            torch.save(G_XtoY.state_dict(), "saved_models/G_XtoY_%d.pth" % epoch)
            torch.save(G_YtoX.state_dict(), "saved_models/G_YtoX_%d.pth" % epoch)
            torch.save(D_X.state_dict(), "saved_models/D_X_%d.pth" % epoch)
            torch.save(D_Y.state_dict(), "saved_models/D_Y_%d.pth" % epoch)
```

```
Models moved to GPU.
```

[5]:
```
training_loop(train_dataloader, test_dataloader, opt)
```

```
                G_XtoY
----------------------------------------
Generator(
  (model): Sequential(
    (0): ReflectionPad2d((3, 3, 3, 3))
    (1): Conv2d(3, 64, kernel_size=(7, 7), stride=(1, 1))
    (2): InstanceNorm2d(64, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
    (3): ReLU(inplace=True)
    (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (5): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (8): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
    (9): ReLU(inplace=True)
    (10): ResnetBlock(
      (block): Sequential(
```

```
        (0): ReflectionPad2d((1, 1, 1, 1))
        (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
        (3): ReLU(inplace=True)
        (4): ReflectionPad2d((1, 1, 1, 1))
        (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      )
    )
    (11): ResnetBlock(
      (block): Sequential(
        (0): ReflectionPad2d((1, 1, 1, 1))
        (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
        (3): ReLU(inplace=True)
        (4): ReflectionPad2d((1, 1, 1, 1))
        (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      )
    )
    (12): ResnetBlock(
      (block): Sequential(
        (0): ReflectionPad2d((1, 1, 1, 1))
        (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
        (3): ReLU(inplace=True)
        (4): ReflectionPad2d((1, 1, 1, 1))
        (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      )
    )
    (13): ResnetBlock(
      (block): Sequential(
        (0): ReflectionPad2d((1, 1, 1, 1))
        (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
        (3): ReLU(inplace=True)
        (4): ReflectionPad2d((1, 1, 1, 1))
        (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
```

```
      )
    )
    (14): ResnetBlock(
      (block): Sequential(
        (0): ReflectionPad2d((1, 1, 1, 1))
        (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
        (3): ReLU(inplace=True)
        (4): ReflectionPad2d((1, 1, 1, 1))
        (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      )
    )
    (15): ResnetBlock(
      (block): Sequential(
        (0): ReflectionPad2d((1, 1, 1, 1))
        (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
        (3): ReLU(inplace=True)
        (4): ReflectionPad2d((1, 1, 1, 1))
        (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      )
    )
    (16): ResnetBlock(
      (block): Sequential(
        (0): ReflectionPad2d((1, 1, 1, 1))
        (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
        (3): ReLU(inplace=True)
        (4): ReflectionPad2d((1, 1, 1, 1))
        (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      )
    )
    (17): ResnetBlock(
      (block): Sequential(
        (0): ReflectionPad2d((1, 1, 1, 1))
        (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
        (3): ReLU(inplace=True)
```

```
        (4): ReflectionPad2d((1, 1, 1, 1))
        (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      )
    )
    (18): ResnetBlock(
      (block): Sequential(
        (0): ReflectionPad2d((1, 1, 1, 1))
        (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
        (3): ReLU(inplace=True)
        (4): ReflectionPad2d((1, 1, 1, 1))
        (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      )
    )
    (19): Upsample(scale_factor=2.0, mode=nearest)
    (20): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (21): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
    (22): ReLU(inplace=True)
    (23): Upsample(scale_factor=2.0, mode=nearest)
    (24): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): InstanceNorm2d(64, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
    (26): ReLU(inplace=True)
    (27): ReflectionPad2d((3, 3, 3, 3))
    (28): Conv2d(64, 3, kernel_size=(7, 7), stride=(1, 1))
    (29): Tanh()
  )
)
---------------------------------------
                G_YtoX
---------------------------------------
Generator(
  (model): Sequential(
    (0): ReflectionPad2d((3, 3, 3, 3))
    (1): Conv2d(3, 64, kernel_size=(7, 7), stride=(1, 1))
    (2): InstanceNorm2d(64, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
    (3): ReLU(inplace=True)
    (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (5): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
    (6): ReLU(inplace=True)
```

```
    (7): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (8): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
    (9): ReLU(inplace=True)
    (10): ResnetBlock(
      (block): Sequential(
        (0): ReflectionPad2d((1, 1, 1, 1))
        (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
        (3): ReLU(inplace=True)
        (4): ReflectionPad2d((1, 1, 1, 1))
        (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      )
    )
    (11): ResnetBlock(
      (block): Sequential(
        (0): ReflectionPad2d((1, 1, 1, 1))
        (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
        (3): ReLU(inplace=True)
        (4): ReflectionPad2d((1, 1, 1, 1))
        (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      )
    )
    (12): ResnetBlock(
      (block): Sequential(
        (0): ReflectionPad2d((1, 1, 1, 1))
        (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
        (3): ReLU(inplace=True)
        (4): ReflectionPad2d((1, 1, 1, 1))
        (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      )
    )
    (13): ResnetBlock(
      (block): Sequential(
        (0): ReflectionPad2d((1, 1, 1, 1))
        (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
```

```
track_running_stats=False)
        (3): ReLU(inplace=True)
        (4): ReflectionPad2d((1, 1, 1, 1))
        (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      )
    )
    (14): ResnetBlock(
      (block): Sequential(
        (0): ReflectionPad2d((1, 1, 1, 1))
        (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
        (3): ReLU(inplace=True)
        (4): ReflectionPad2d((1, 1, 1, 1))
        (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      )
    )
    (15): ResnetBlock(
      (block): Sequential(
        (0): ReflectionPad2d((1, 1, 1, 1))
        (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
        (3): ReLU(inplace=True)
        (4): ReflectionPad2d((1, 1, 1, 1))
        (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      )
    )
    (16): ResnetBlock(
      (block): Sequential(
        (0): ReflectionPad2d((1, 1, 1, 1))
        (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
        (3): ReLU(inplace=True)
        (4): ReflectionPad2d((1, 1, 1, 1))
        (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      )
    )
    (17): ResnetBlock(
```

```
      (block): Sequential(
        (0): ReflectionPad2d((1, 1, 1, 1))
        (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
        (3): ReLU(inplace=True)
        (4): ReflectionPad2d((1, 1, 1, 1))
        (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      )
    )
    (18): ResnetBlock(
      (block): Sequential(
        (0): ReflectionPad2d((1, 1, 1, 1))
        (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
        (3): ReLU(inplace=True)
        (4): ReflectionPad2d((1, 1, 1, 1))
        (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      )
    )
    (19): Upsample(scale_factor=2.0, mode=nearest)
    (20): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (21): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
    (22): ReLU(inplace=True)
    (23): Upsample(scale_factor=2.0, mode=nearest)
    (24): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): InstanceNorm2d(64, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
    (26): ReLU(inplace=True)
    (27): ReflectionPad2d((3, 3, 3, 3))
    (28): Conv2d(64, 3, kernel_size=(7, 7), stride=(1, 1))
    (29): Tanh()
  )
)
----------------------------------------
                D_X
----------------------------------------
Discriminator(
  (model): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
```

```
    (3): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (9): InstanceNorm2d(512, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): ZeroPad2d(padding=(1, 0, 1, 0), value=0.0)
    (12): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), padding=(1, 1))
  )
)
--------------------------------------
                  D_Y
--------------------------------------
Discriminator(
  (model): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (3): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (9): InstanceNorm2d(512, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): ZeroPad2d(padding=(1, 0, 1, 0), value=0.0)
    (12): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), padding=(1, 1))
  )
)
--------------------------------------
[Epoch 13/50] [Batch 587/1875] [D loss: 0.250000] [G loss: 0.621749]
```