Deep Learning and Practice: InfoGAN

0756079 陳冠聞

May 22, 2019

1 Introduction

GAN (Generative Adversarial Network) is the most popular generative model nowadays. Many researchs in recent years have found many variations of GANs. One important variation of GANs is **InfoGAN**, which learns the interpretable representation of latent code in unsupervised way.

In this lab, I will implement the InfoGAN in python using *PyTorch* package, and train on MNIST dataset to generate hand written digits. Futhermore, I will discuss the difference between traditional GAN and InfoGAN in both theory and implementation.

2 InfoGAN

2.1 Generative Adversarial Networks

Generative Adversarial Networks (GANs) are generative models proposed by Goodfellow et al. in 2014. GAN is consist of two two differentiable functions, represented by neural networks. One network is called generator, which tries to generate data that come from some probability distribution $p_G(x)$ to be as similar to real data distribution $p_{data}(x)$ as possible. Another network is called discriminator, which tries to classify given data are from real data distribution $p_{data}(x)$ or the fake data distribution $p_G(x)$. These two network act like two adversarial players in zero-sum game, and we train them by optimizing one network while the other fixed. The object function of training is

$$\min_G \max_D V_{GAN}(G, D), \text{ where} \tag{1}$$

$$V_{GAN}(G, D) = E_{x \sim p_{data}(x)}[log(D(x))] + E_{z \sim p_z(z)}[log(1 - D(G(z)))] \tag{2}$$

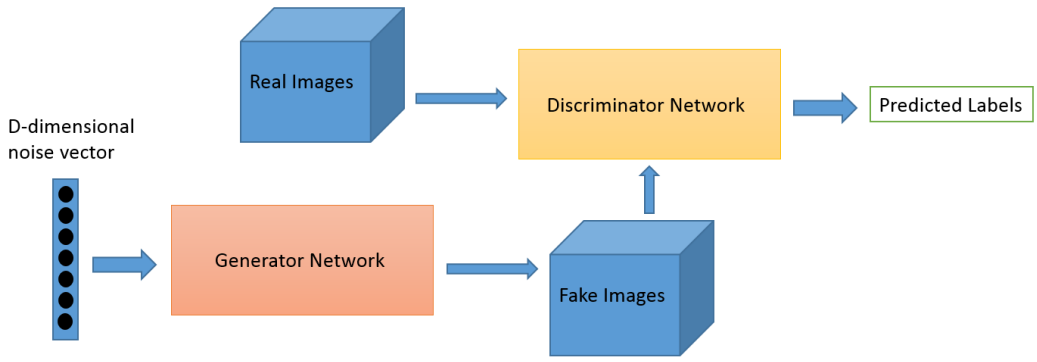


Figure 1: Generative Adversarial Networks

2.2 InfoGAN

Although GANs are capable of generating realistic data, but usually the feature vector z used to generate data is not interpretable. That is, given a hidden vector, we have no idea what the generated data will be like. The reason is that generated data x dose not have much information on the noise z from which x is generated because of havily entangled use of z.

InfoGAN try to solve the problem by maxmizing the mutual information $I(c|x)$ between the latent code c and the generated data x. The objection function become

$$\min_G \max_D V_{GAN}(G, D) - \lambda I(c|x = G(z, c))$$

where $I(c|x) = H(c) - H(c|x)$. The term of $I(c|x)$ is the difference between prior entropy $H(c)$ and the posterior entropy $H(c|x)$, which is the decrease of uncertainty when given the x. That is, the higher the mutual information is, the more confident about the c given the x. Unfortunately, evaluation of

$I(c|x)$ based on evaluation and sampling from the posterior $p(c|x)$, which is hard to get. Thus, InfoGAN uses **Variational Maximization** of mutual information by an approximate function $Q(c|x) = p(c|x)$, where the function Q is also represented by neural network, and shares the weights of Q with the discriminator to avoid increasing of network size.

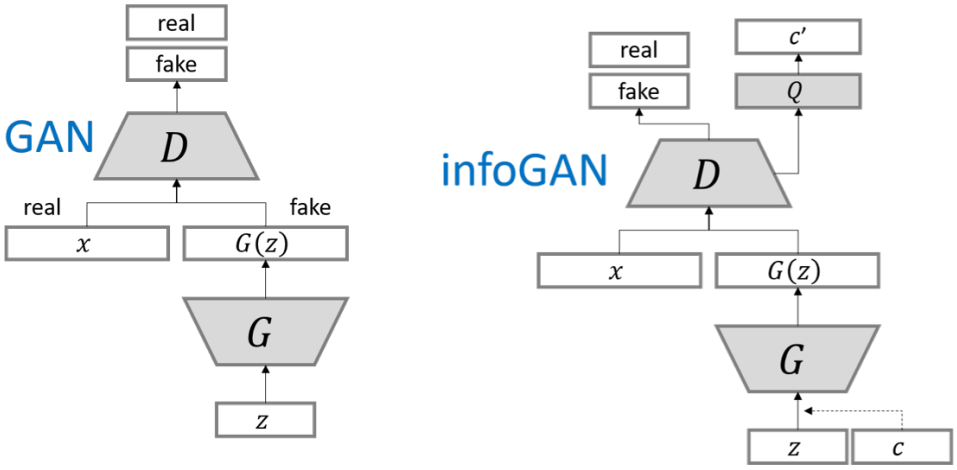


Figure 2: GAN v.s infoGAN

3 Implementaion

In this lab, I implement the InfoGAN by modifying from DCGAN (https://github.com/pytorch/examples/tree/master/dcgan), and I will explain the major difference between InfoGAN and DCGAN in following subsection. The detail code explaination can refer to Appendix.

3.1 Generator

Same as the DCGAN, the generator of InfoGAN is consist of multiple deconvolution layers to generate image from an latent vector. However, the latent vector in InfoGAN is consist of three part: the noise, the discrete code and the continuous code. And the meaning of these code will be discuss in Loss Function section.

```python
class Generator(nn.Module):
    def __init__(self, img_size=64, img_channels=1, latent_dim=52, cat_dim=10, cont_dim=2):
        super(Generator, self).__init__()

        def deconv_block(in_channels, out_channels, kernel_size=(4,4), stride=(1,1), padding=0):
            block = [
                nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride, padding, bias=False),
                nn.BatchNorm2d(out_channels),
                nn.ReLU(),
            ]
            return block

        self.deconv_blocks = nn.Sequential(
            *deconv_block(64 , 512, kernel_size=(4, 4), stride=(1, 1), padding=0),
            *deconv_block(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=1),
            *deconv_block(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=1),
            *deconv_block(128, 64 , kernel_size=(4, 4), stride=(2, 2), padding=1),
            nn.ConvTranspose2d( 64,   1, kernel_size=(4, 4), stride=(2, 2), padding=(1,1), bias=False),
            nn.Tanh(),
        )

    def forward(self, noise, cat_code, cont_code):
        batch_size = noise.size(0)
        gen_input = torch.cat((noise, cat_code, cont_code), -1).view(batch_size, -1, 1, 1)
        img = self.deconv_blocks(gen_input)
        return img
```

Figure 3: Code Snippet for Generator

```
Generator(
  (deconv_blocks): Sequential(
    (0): ConvTranspose2d(64, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU()
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU()
    (12): ConvTranspose2d(64, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
```

Figure 4: Architecture for Generator

3.2 Discriminator and Q

For the discriminator, it is composed of multiple convolution layers to classify an image is real or fake for both DCGAN and InfoGAN. The difference is that there is additional layer to reconstruct the latent code from the image in InfoGAN. In fact, the task of reconstrucion the latent code could be another network Q independent to discriminator, but we can reduce the model size by sharing the weights of Q with discriminator.

```
class Discriminator(nn.Module):
    def __init__(self, img_size=64, latent_dim=52, cat_dim=10, cont_dim=2):
        super(Discriminator, self).__init__()

        def discriminator_block(in_channels, out_channels, bn=True):
            block = [nn.Conv2d(in_channels, out_channels, 4, 2, 1, bias=False)]
            if bn:
                block.append(nn.BatchNorm2d(out_channels))
            block.append(nn.LeakyReLU(0.2, inplace=True))
            return block
        self.conv_blocks = nn.Sequential(
            *discriminator_block(1, 64, bn=False),
            *discriminator_block(64, 128),
            *discriminator_block(128, 256),
            *discriminator_block(256, 512),
        )
        # The height and width of downsampled image
        ds_size = img_size // 2 ** 4
        self.discriminator = nn.Sequential(
            nn.Conv2d(512, 1, kernel_size=(4,4), stride=(1, 1), bias=False),
            nn.Sigmoid()
        )
        self.Q = nn.Sequential(nn.Linear(512 * ds_size ** 2, cat_dim+cont_dim))

    def forward(self, img):
        batch_size = img.size(0)
        out = self.conv_blocks(img)
        validity = self.discriminator(out).view(batch_size, -1)
        out = out.view(out.shape[0], -1)
        code = self.Q(out)
        cat_code = code[:, :cat_dim]  # Discrete Code (Category)
        cont_code = code[:, cat_dim:] # Continuous Code
        return validity, cat_code, cont_code
```

Figure 5: Code Snippet for Discriminator

```
Discriminator(
  (conv_blocks): Sequential(
    (0): Conv2d(1, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace)
  )
  (discriminator): Sequential(
    (0): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): Sigmoid()
  )
  (Q): Sequential(
    (0): Linear(in_features=8192, out_features=12, bias=True)
  )
)
```

Figure 6: Code Snippet for Discriminator

3.3 Loss Function

Another difference between DCGAN and InfoGAN is that except the adversial loss, there is another loss term called information loss for InfoGAN.

3.3.1 Adversarial Loss

The object function for discriminator in GAN is

$$\max_G E_{x\sim p_{data}(x)}[log(D(x))] + E_{z\sim p_z(z)}[log(1 - D(G(z)))]$$

For Generator, since the D(x) is fixed, the object function I use is

$$\min_D E_{z\sim p_z(z)}[log(1 - D(G(z)))]$$

In this lab, I use binary cross entropy between vector of ones $\vec{1}$ and the output of discriminator for real images, as well as binary cross entropy between vecotr of zeros $\vec{0}$ and the output of discriminator for generated images to train the discriminator. On the other hand, I use binary cross entropy between $\vec{1}$ and the output of discriminator for generated images to train the generator.

3.3.2 Information Loss

InfoGAN tries to maximize the mutual information between latent code c and output image x. Thus, the loss of information is the reconsturction loss between the latent code c and the reconstructed code $\hat{c}$.

As mention earlier, the latent vector is consist of three parts: the noise z ($\in R^{52}$), the discrete code c_d ($\in R^{10}$) which is one hot vector for category and the continuous code c_c ($\in R^2$). I use the **cross entropy loss** between the real discrete code c_d and the reconstructed discrete code $\hat{c_d}$ for discrete code, and **mean square loss** between the real continuous code c_c and the reconstructed continuous code $\hat{c_c}$ for continuous code.

3.4 Fixed Latent Code

To see the change of generator over training epochs, I set an fixed latent code and use it to generate image every 500 batches.

```
zero_noise = Variable(FloatTensor(np.zeros((n_classes ** 2, latent_dim))))
fixed_noise = Variable(FloatTensor(np.random.normal(0, 1, (n_classes ** 2, latent_dim))))
fixed_cat_code = to_categorical(
    np.array([num for _ in range(n_classes) for num in range(n_classes)]), num_classes=n_classes
)
zero_cont_code = Variable(FloatTensor(np.zeros((n_classes ** 2, cont_dim))))

def sample_image(n_row, batches_done):
    """Saves a grid of generated digits ranging from 0 to n_classes"""
    # Fixed sample
    noise = Variable(FloatTensor(np.random.normal(0, 1, (n_row ** 2, latent_dim))))
    fixed_sample = generator(fixed_noise, fixed_cat_code, zero_cont_code)
    save_image(fixed_sample.data, "images/fixed/%d.png" % batches_done, nrow=n_row, normalize=True)

    # Varied c1 and c2
    zeros = np.zeros((n_row ** 2, 1))
    c_varied = np.repeat(np.linspace(-1, 1, n_row)[:, np.newaxis], n_row, 0)
    c1 = Variable(FloatTensor(np.concatenate((c_varied, zeros), -1)))
    c2 = Variable(FloatTensor(np.concatenate((zeros, c_varied), -1)))
    sample1 = generator(zero_noise, fixed_cat_code, c1)
    sample2 = generator(zero_noise, fixed_cat_code, c2)
    save_image(sample1.data, "images/varying_c1/%d.png" % batches_done, nrow=n_row, normalize=True)
    save_image(sample2.data, "images/varying_c2/%d.png" % batches_done, nrow=n_row, normalize=True)
```

Figure 7: Code Snippet for Fixed Latent Code

4 Result

In this lab, I read the images in MINST dataset and reshape it to (64, 64) and normalize the pixel value to [-1, 1]. I use the generator and discriminator mentioned above, and train them using Adam optimizer with learning rate 1e-3 for generator, as well as Adam optimizer with learning rate 2e-4 for discriminator, and the information loss coefficient λ is set to 1. The batch size is set to 128, and train 80 epochs in total. For the latent code, the noise is drawn from the standard normal distribution, the discrete code is a random one hot vector, and the continuous code is drawn from uniform distribution of [-1, 1].

4.1 Loss Curve

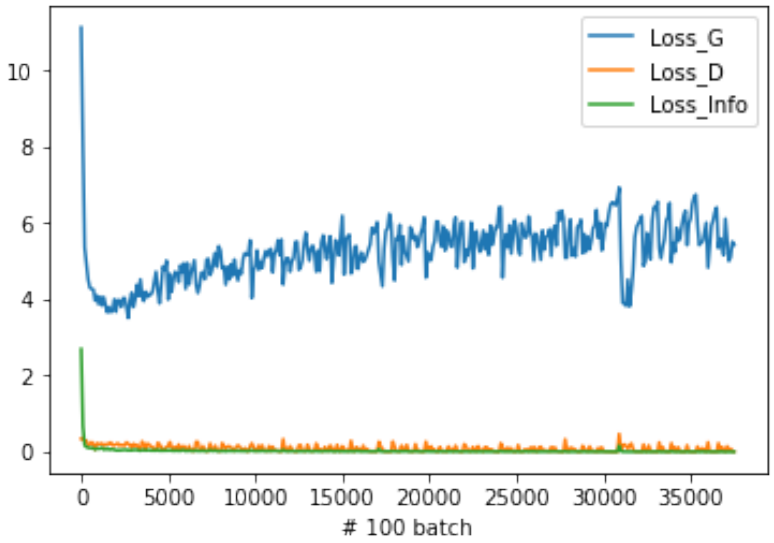


Figure 8: Loss Curve

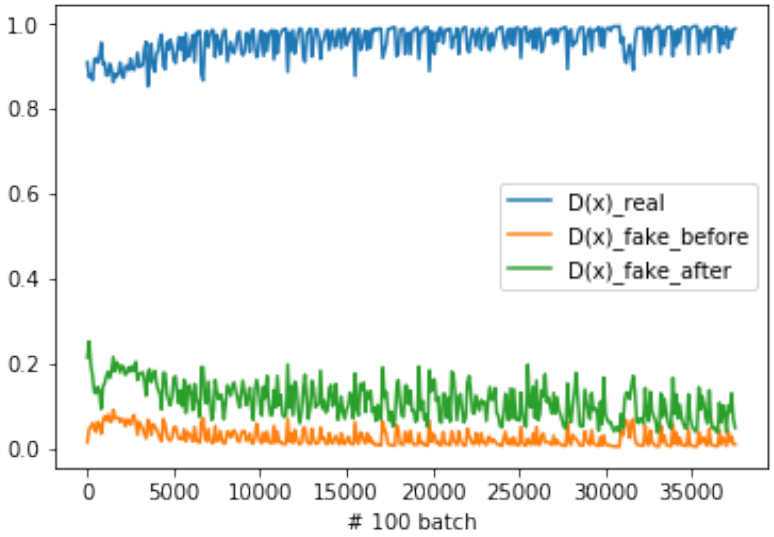


Figure 9: Output probability of discriminator

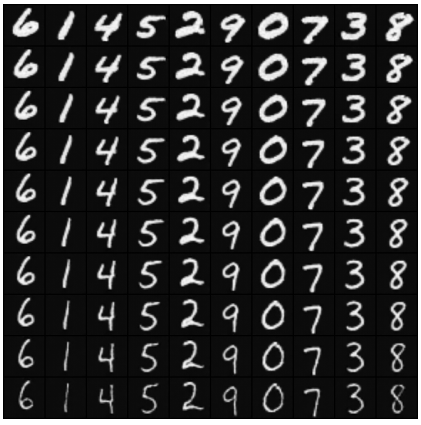
From the Figure 8, we can see that the information loss drop very quickly, and converge in very early stage. And from the Figure 9, we can see that the output of generated image improve after generator update, which means that the generator indeed learn something. More interpretation of these two figures are in the discussion section.

4.2 Generated Image

To see how well the InfoGAN learn the latent representation, I generate the image by setting different latent code. In following three figures, I all use the same discrete code for the same column, and use different discrete code for different column. For (a), the continuous code are set to zero, and sample random noise. For (b) and (c), the noise are set to zero, and vary first and second dimension of continuous code from -1 (first row) to +1 (last row) respectively, the other dimension of continuous code are set to zero.

(a) Random Noise

(b) vary $c1$ from -1 to 1 (Rotation)



(c) vary $c2$ from -1 to 1 (Thickness)

Figure 10: Generated Image

5 Discussion

5.1 Interpretation of the loss curve

From the Figure 8, we can see that the term of information loss decrease very quickly, which means that the model can learn to reconstruct latent code in very early stage. Another evidence is that we can easily distinguish different latent code from very begining iterations (especially the discrete code) in Fiugre 11, while the generated digits are not so realistic yet.

(a) after 0 iterations (b) after 2000 iterations (c) after 4000 iterations (d) after 6000 iterations

Figure 11: Generated Image at Different Iteration

On the other hand, I was very confused with the result of Figure 9 at the beginning. I suspected that the generator not learn well, because the discriminator can easily distinguish between real image and fake image, but the generated images seem to be realistic. Then I do some research on the Internet and found out that the losses of GAN are very non-intuitive, because in fact that generator and discriminator are competing against each other. Thus, although the generator can not totally fool the discriminator, is still learn to generate images with better quality since the discriminator becomes better gradually.