GAN及其变体用于手写数字图像生成

代码来自于 https://github.com/znxlwm/pytorch-generative-model-collections，缩放像素到[0,1]区间后，原始代码做了三通道归一化而mnist数据集是单通道图像，需改成单通道且均值方差必须为0.5，或者直接不做transforms.Normalize也可以。

一、实验环境配置

环境配置

torch	1.8.1+cu111
torchvision	0.9.1+cu111
numpy	1.17.3
Python	3.6
matplotlib	3.1.1

Dataset

- GANs are notoriously finicky with hyperparameters, and also require many training epochs. In order to make this assignment approachable without a GPU, we will be working on the MNIST dataset, which is 60,000 training and 10,000 test images. Each picture contains a centered image of white digit on black background (0 through 9). This was one of the first datasets used to train convolutional neural networks and it is fairly easy -- a standard CNN model can easily exceed 99% accuracy.

  To simplify our code here, we will use the PyTorch MNIST wrapper, which downloads and loads the MNIST dataset. The default parameters will take 5,000 of the training examples and place them into a validation dataset. The data will be saved into a folder called `data`.

二、理论方法

In 2014, Goodfellow et al. presented a method for training generative models called Generative Adversarial Networks (GANs for short). In a GAN, we build two different neural networks. Our first network is a traditional classification network, called the **discriminator**. We will train the discriminator to take images, and classify them as being real (belonging to the training set) or fake (not present in the training set). Our other network, called the **generator**, will take random noise as input and transform it using a neural network to produce images. The goal of the generator is to fool the discriminator into thinking the images it produced are real.

We can think of this back and forth process of the generator (G) trying to fool the discriminator (D), and the discriminator trying to correctly classify real vs. fake as a minimax game:

$$\underset{G}{\text{minimize}}\;\underset{D}{\text{maximize}}\;\mathbb{E}_{x\sim p_{\text{data}}}\left[\log D(x)\right]+\mathbb{E}_{z\sim p(z)}\left[\log(1-D(G(z)))\right]$$

where $z\sim p(z)$ are the random noise samples, $G(z)$ are the generated images using the neural network generator G, and D is the output of the discriminator, specifying the probability of an input being real. In

Goodfellow et al., they analyze this minimax game and show how it relates to minimizing the Jensen-Shannon divergence between the training data distribution and the generated samples from G.

To optimize this minimax game, we will aternate between taking gradient *descent* steps on the objective for G, and gradient *ascent* steps on the objective for D:

1. update the **generator** (G) to minimize the probability of the **discriminator making the correct choice**.
2. update the **discriminator** (D) to maximize the probability of the **discriminator making the correct choice**.

While these updates are useful for analysis, they do not perform well in practice. Instead, we will use a different objective when we update the generator: maximize the probability of the **discriminator making the incorrect choice**. This small change helps to allevaiate problems with the generator gradient vanishing when the discriminator is confident. This is the standard update used in most GAN papers, and was used in the original paper from Goodfellow et al..

In this assignment, we will alternate the following updates:

1. Update the generator (G) to maximize the probability of the discriminator making the incorrect choice on generated data:
$$\underset{G}{\text{maximize}}\; \mathbb{E}_{z \sim p(z)}\left[\log D(G(z))\right]$$
2. Update the discriminator (D), to maximize the probability of the discriminator making the correct choice on real and generated data:
$$\underset{D}{\text{maximize}}\; \mathbb{E}_{x \sim p_\text{data}}\left[\log D(x)\right] + \mathbb{E}_{z \sim p(z)}\left[\log(1 - D(G(z)))\right]$$

Generative Adversarial Networks (GANs)

Discriminator

Our first step is to build a discriminator. Fill in the architecture as part of the `nn.Sequential` constructor in the function below. All fully connected layers should include bias terms. The architecture is:

- Fully connected layer with input size 784 and output size 256
- LeakyReLU with alpha 0.01
- Fully connected layer with input_size 256 and output size 256
- LeakyReLU with alpha 0.01
- Fully connected layer with input size 256 and output size 1

Recall that the Leaky ReLU nonlinearity computes $f(x) = \max(\alpha x, x)$ for some fixed constant α; for the LeakyReLU nonlinearities in the architecture above we set $\alpha = 0.01$.

The output of the discriminator should have shape `[batch_size, 1]`, and contain real numbers corresponding to the scores that each of the `batch_size` inputs is a real image.

Generator

Now to build the generator network:

- Fully connected layer from noise_dim to 1024
- `ReLU`
- Fully connected layer with size 1024
- `ReLU`
- Fully connected layer with size 784
- `TanH` (to clip the image to be in the range of [-1,1])

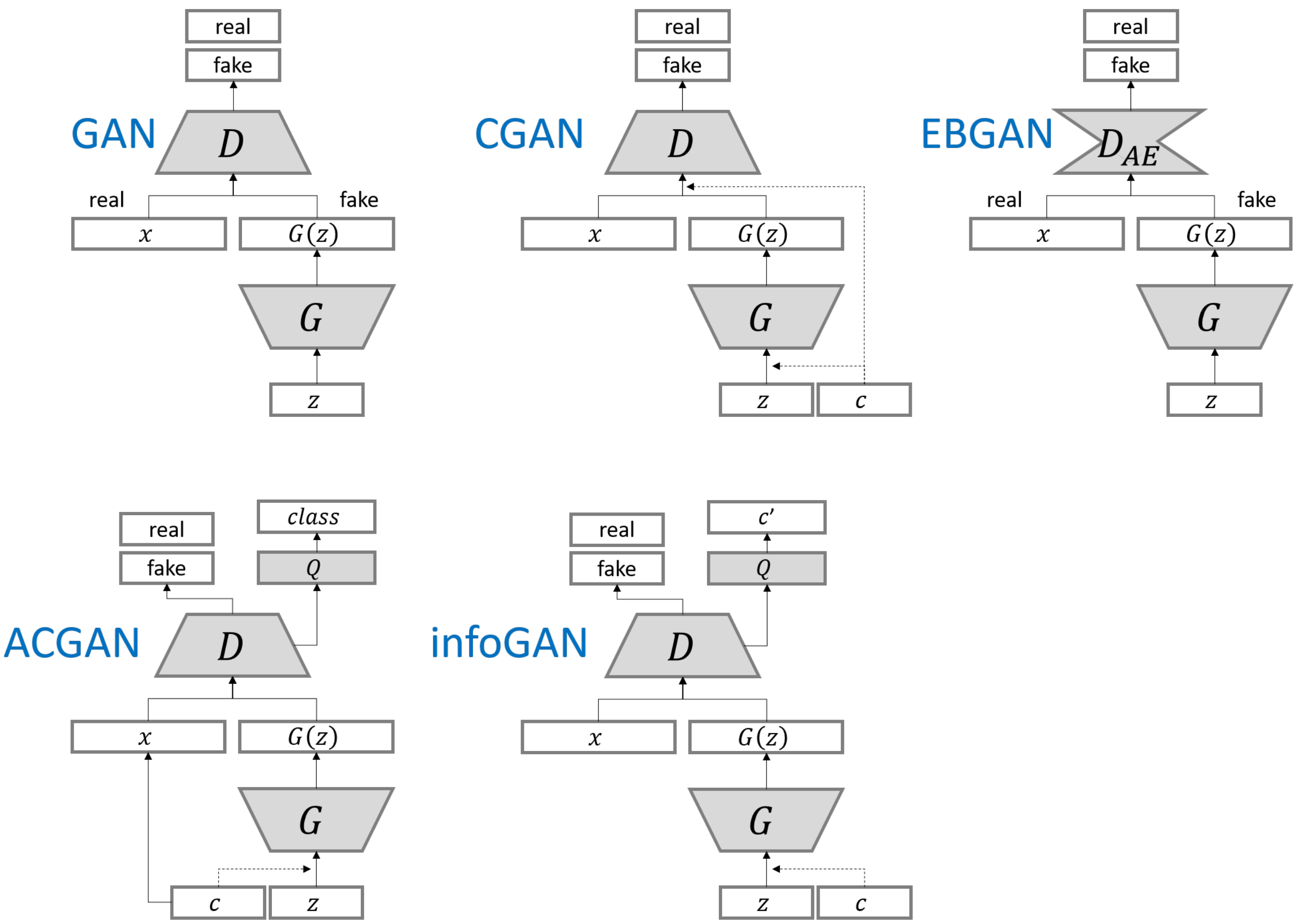
GAN Loss

Compute the generator and discriminator loss. The generator loss is:
$\ell_G = -\mathbb{E}_{z \sim p(z)}\left[\log D(G(z))\right]$
and the discriminator loss is:
$\ell_D = -\mathbb{E}_{x \sim p_\text{data}}\left[\log D(x)\right] - \mathbb{E}_{z \sim p(z)}\left[\log(1-D(G(z)))\right]$

Optimizing our loss

Make a function that returns an `optim.Adam` optimizer for the given model with a 1e-3 learning rate, beta1=0.55, beta2=0.999. You'll use this to construct optimizers for the generators and discriminators for the rest of the notebook.

Variants of GAN structure



LOSS Lists

Name	*Paper Link*	*Value Function*
GAN	https://arxiv.org/abs/1406.2661	$L_D^{GAN} = E\left[\log\left(D(x)\right)\right] + E\left[\log\left(1 - D(G(z))\right)\right]$ $L_G^{GAN} = E\left[\log\left(D(G(z))\right)\right]$
LSGAN	https://arxiv.org/abs/1611.04076	$L_D^{LSGAN} = E[(D(x) - 1)^2] + E[D(G(z))^2]$ $L_G^{LSGAN} = E[(D(G(z)) - 1)^2]$
WGAN	https://arxiv.org/abs/1701.07875	$L_D^{WGAN} = E[D(x)] - E[D(G(z))]$ $L_G^{WGAN} = E[D(G(z))]$ $W_D \leftarrow clip\_by\_value(W_D, -0.01, 0.01)$
WGAN_GP	https://arxiv.org/abs/1704.00028	$L_D^{WGAN\_GP} = L_D^{WGAN} + \lambda E[(\lvert \nabla D(\alpha x - (1 - \alpha G(z)))\rvert - 1)^2]$ $L_G^{WGAN\_GP} = L_G^{WGAN}$
DRAGAN	https://arxiv.org/abs/1705.07215	$L_D^{DRAGAN} = L_D^{GAN} + \lambda E[(\lvert \nabla D(\alpha x - (1 - \alpha x_p))\rvert - 1)^2]$ $L_G^{DRAGAN} = L_G^{GAN}$
CGAN	https://arxiv.org/abs/1411.1784	$L_D^{CGAN} = E\left[\log\left(D(x, c)\right)\right] + E\left[\log\left(1 - D(G(z), c)\right)\right]$ $L_G^{CGAN} = E\left[\log\left(D(G(z), c)\right)\right]$
ACGAN	[https://arxiv.org/abs/1610.09585	$L_{D,Q}^{ACGAN} = L_D^{GAN} + E[P(class = c \mid x)] + E[P(class = c \mid G(z))]$ $L_G^{ACGAN} = L_G^{GAN} + E[P(class = c \mid G(z))]$
EBGAN	https://arxiv.org/abs/1609.03126	$L_D^{EBGAN} = D_{AE}(x) + \max(0, m - D_{AE}(G(z)))$ $L_G^{EBGAN} = D_{AE}(G(z)) + \lambda \cdot PT$
BEGAN	https://arxiv.org/abs/1703.10717	$L_D^{BEGAN} = D_{AE}(x) - k_t D_{AE}(G(z))$ $L_G^{BEGAN} = D_{AE}(G(z))$ $k_{t+1} = k_t + \lambda(\gamma D_{AE}(x) - D_{AE}(G(z)))$

三、实验结果展示

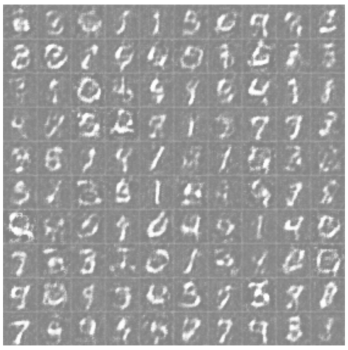
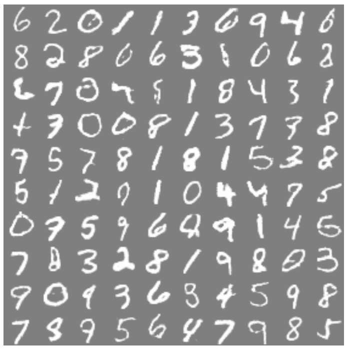
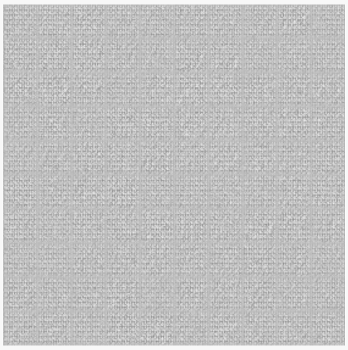
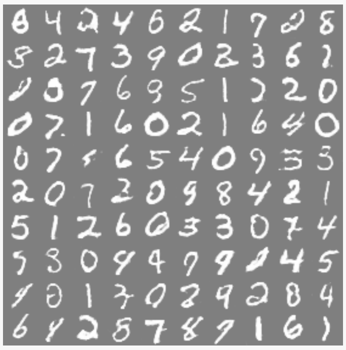
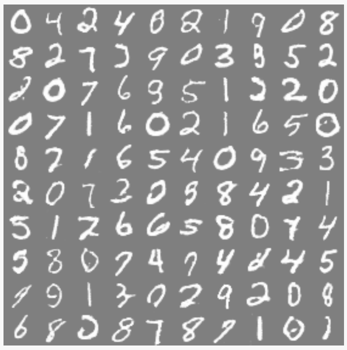
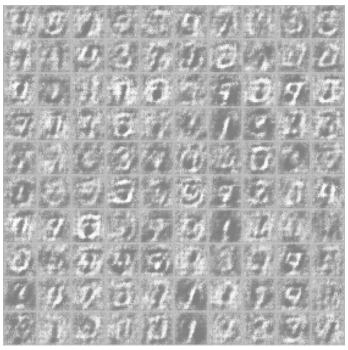
Results for mnist

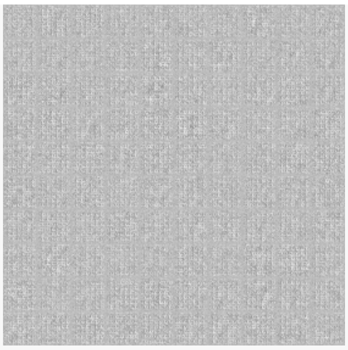
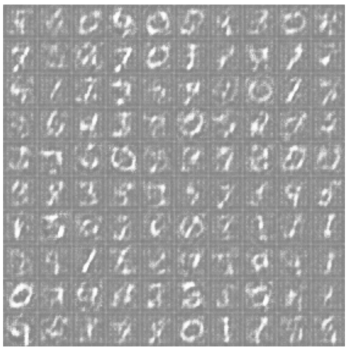
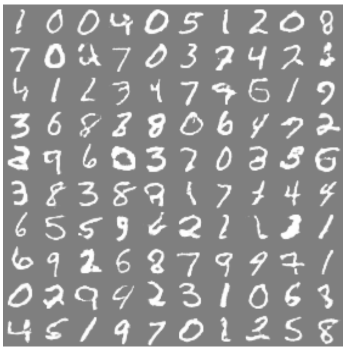
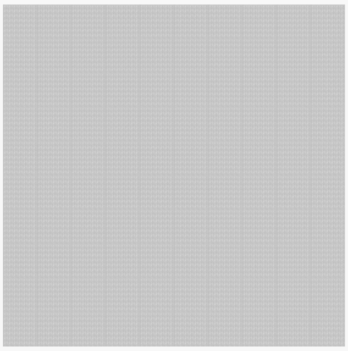
The following results can be reproduced with command:

```
python main.py --dataset mnist --gan_type <TYPE> --epoch 50 --batch_size 64
```

random generation

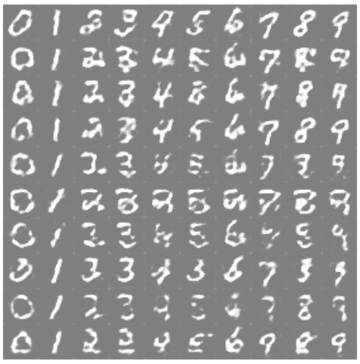
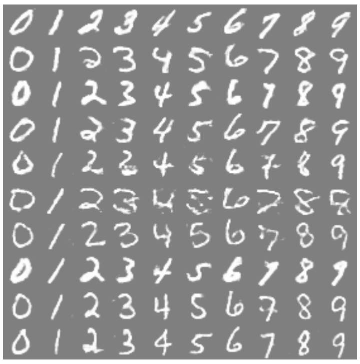
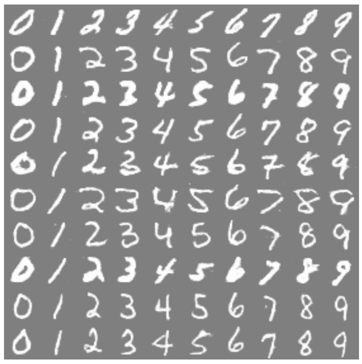
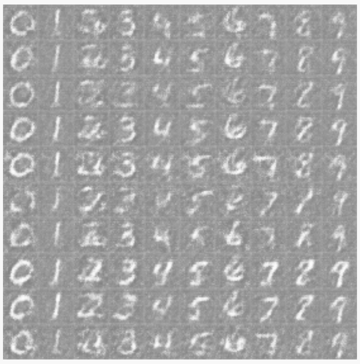
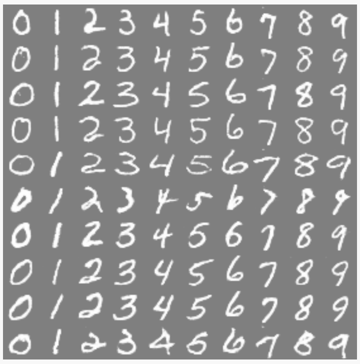
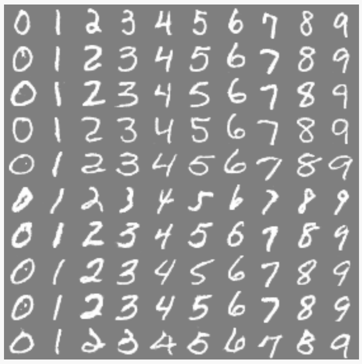
All results are generated from the random noise vector.

Name	***Epoch 1***	***Epoch 15***	***Epoch 30***
GAN			
LSGAN			
WGAN			

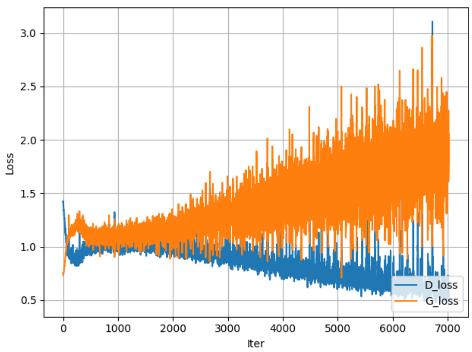
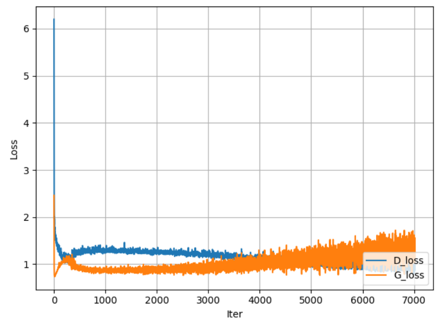
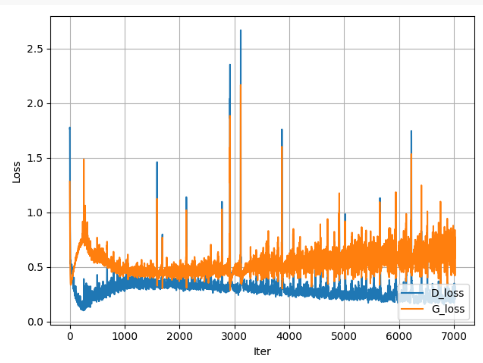
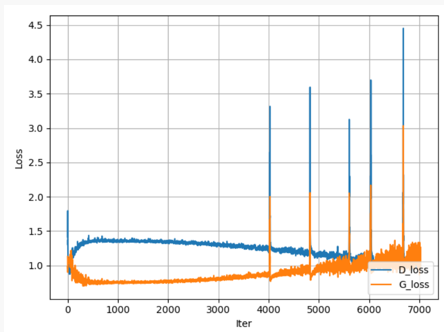
WGAN_GP			
DRAGAN			
EBGAN			
BEGAN			

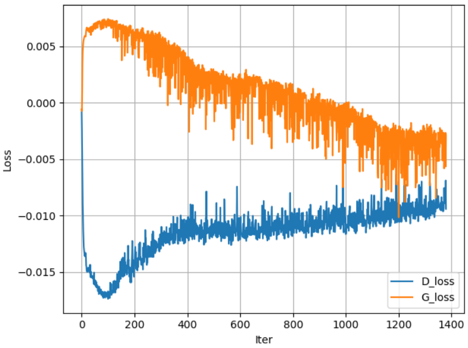
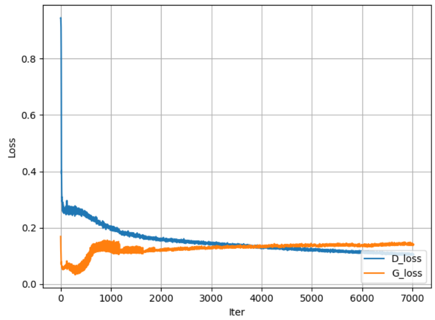
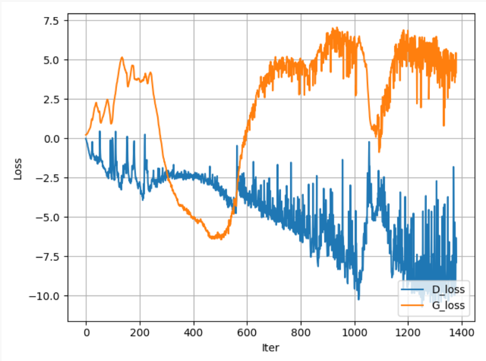
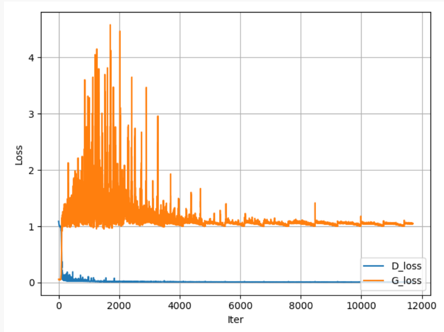
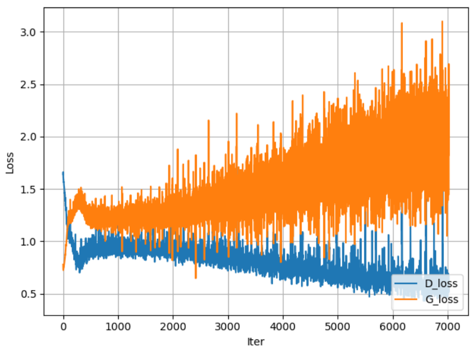
Conditional generation

Each row has the same noise vector and each column has the same label condition.

Name	Epoch 1	Epoch 15	Epoch 30
CGAN			
ACGAN			

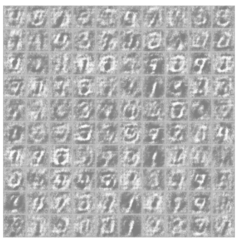
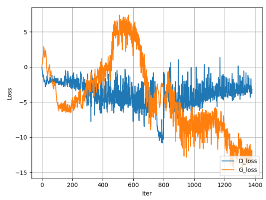
Loss plot

Name	Loss	Name	Loss
GAN		ACGAN	
LSGAN		CGAN	

WGAN		BEGAN	
WGAN_GP		EBGAN	
DRAGAN			

WGAN_GP和EBGAN没有达到代码demo中的效果，进一步分析原因。

在30个EPOCH内，可以看出只有WGAN_GP和EBGAN的discriminator和generator的loss快速重合后，迅速分开。观察应该是对于mnist数据集这两个的discriminator太强了，以至于让generator学不到什么东西。对学习率调整，扩大十倍后WGAN_GP达到预期效果，EBGAN没训练出来，调整了很多参数还是不行。

Name	***Epoch 1***	***Epoch 15***	***Epoch 30***	***loss***
WGAN_GP				
EBGAN				