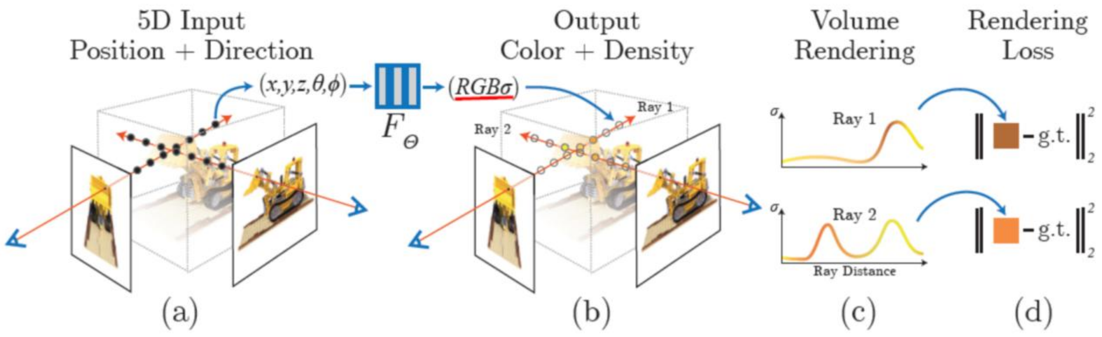
Final Project

Generating Photorealistic Pokemon Sprites' Images with DCGAN

Reporter: Jing Wang

1 Background

Recently, My interest is mainly 3-dimensional scene reconstruction methods using Neural Radiance Fields (NeRF). The Neural Radiance Fields (NeRF) paper was ostensibly published at ECCV 2020, at the end of August, by Mildenhall et al. For its simplicity and better photo-realistic rendering performance, growth of NeRF-style methods seems exponential. Up to now, less than 3 years after NeRF was proposed, there are more than 300 papers related to NeRFs. NeRF uses new deep learning and classical physics-based methods to get 3D scene representation. A NeRF model stores a volumetric scene representation as the weights of an MLP, trained on many images with a known camera pose. New views are rendered by integrating the density and color at regular intervals along each viewing ray[1].



Optimization of NeRF adopts minimizing distances between output and ground truth. However, studying shows, minimizing distances between output and ground truth does not guarantee realistic looking output, and the small distance and photorealism are at odds with each other[2].

Therefore, instead of distance minimization, deep generative models focus on distribution matching, i.e., matching the distribution of generated results to the distribution of training data. Among many types of generative models, Generative Adversarial Networks (GANs) have shown promising results for many computer graphics tasks.

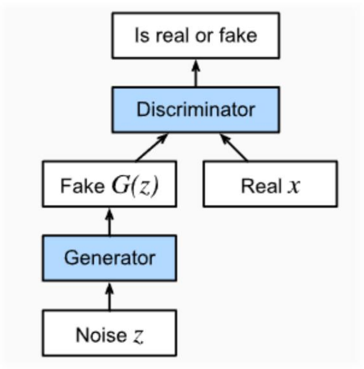
Most of NeRF-style papers use MLP models, very few use GAN models. Recently, I have learned some GAN papers related to NeRF, and I want to extend NeRF to GAN model.

In final project, I decide to implement the DCGAN model with Pytorch to generate photorealistic images before I do further research on GAN models applied to NeRF, which enable me to be more familiar with the practical coding part of GAN and be prepared for my further research.

2 Introduction

Discriminative learning in which we discriminate photos of different kinds is to use deep neural networks to learn mapping from data examples to labels. These processes are primarily based on backpropagation and dropout algorithms. But there is more to machine learning than just solving discriminative tasks. Given a large dataset without any labels, we want to learn a model that concisely captures the characteristics of this data and synthesizes data examples that resemble the distribution of the training data. For example, with photos of different faces, we want to generate new photorealistic images that looks like it might come from the same dataset. This kind of learning is called generative modeling.

In 2014, a breakthrough paper introduced Generative adversarial networks (GANs) came[3]. A GAN generator $\mathrm{G}: \mathrm{z} \rightarrow \hat{x}$ learns a mapping from a low-dimensional random vector z to an output image $\hat{x}$. Typically, the input vector is sampled from a multivariate Gaussian or Uniform distribution. The generator G is trained to produce outputs that cannot be distinguished from "real" images by an adversarially trained discriminator D. The discriminator is trained to detect synthetic images generated by the generator.



In this project, I apply models based on the deep convolutional GANs (DCGAN) introduced in Radford et al.[4] to generate photorealistic images.

3 Model & Algorithm

3.1 Training of GAN

The discriminator is a binary classifier to distinguish if the input is real (from real data) or fake (from the generator). Typically, the discriminator outputs a scalar for the corresponding input to which sigmoid function is usually applied to obtain the predicted probability. Assume the label y for the true data (x) is 1 and 0 for the fake data ($\hat{x}$). We train the discriminator to minimize the loss (to minimize $\mathrm{D}(\hat{x}) = \mathrm{P}(\mathrm{y} = 1|\hat{x})$ and to maximize $\mathrm{D}(x) = \mathrm{P}(\mathrm{y} = 1|x)$):

$$E_{x\sim P_{data}}[-logD(x)] + E_Z[-log(1 - D(G(z)))]$$

For the generator, it first usually draws parameters from a normal distribution and applies a function to generate. The goal of the generator is to fool the discriminator. We update the parameters of the generator to maximize the following loss:

$$E_Z[-log(1-D(G(z)))]$$

Actually, x and z contain a batch of data, which can be rewritten as $\begin{bmatrix} x^{(1)} \\ x^{(2)} \\ \dots \\ x^{(n)} \end{bmatrix}$ and $\begin{bmatrix} z^{(1)} \\ z^{(2)} \\ \dots \\ z^{(n)} \end{bmatrix}$, where n is the batch size.

Correspondingly, $\mathrm{D}(x) = \begin{bmatrix} \mathrm{P}(\mathrm{y}=1|x^{(1)}) \\ \mathrm{P}(\mathrm{y}=1|x^{(2)}) \\ \dots \\ \mathrm{P}(\mathrm{y}=1|x^{(n)}) \end{bmatrix}$, $\mathrm{G}(z) = \begin{bmatrix} \hat{x}^{(1)} \\ \hat{x}^{(2)} \\ \dots \\ \hat{x}^{(n)} \end{bmatrix}$

The GAN proposed by Goodfellow et al.[3] applied minibatch stochastic gradient descent (SGD) to train the generative adversarial nets. The gradient-based updates can use any standard gradient-based learning rule such as momentum.
So the training of GAN includes:
1. Sample a batch of noise samples z from a normal distribution and a batch of real data x
2. Fix the generator and train the discriminator with the batch of true and fake data by ascending its stochastic gradient for one step:

$$\nabla_{\theta_D} \frac{1}{n} \sum_{i=1}^{n} [logD(x^{(i)}) + log(1-D(G(z^{(i)})))]$$

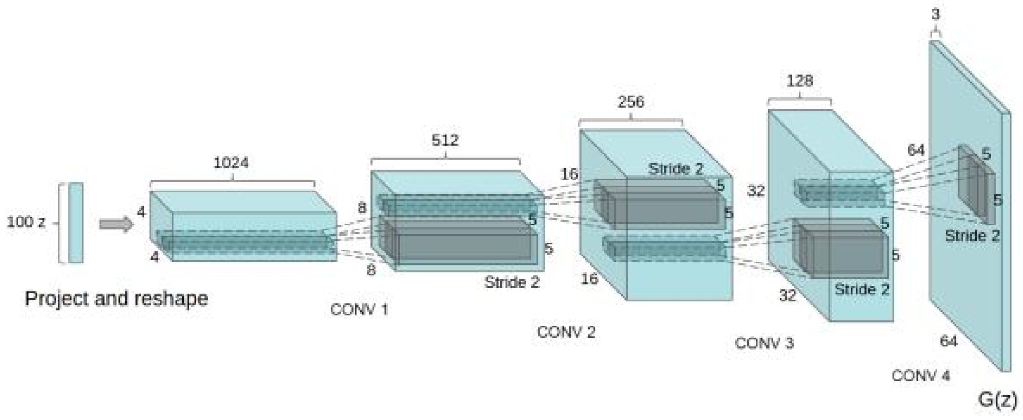
3. Fix the discriminator and update the generator by descending its stochastic gradient for one step:

$$\nabla_{\theta_G} \frac{1}{n} \sum_{i=1}^{n} [log(1-D(G(z^{(i)})))]$$

3.2 DCGAN Model

Approach proposed in Radford et al.[4] is to adopt three changes to CNN architectures. The first is to replace any pooling layers (spatial pooling functions such as maxpooling) in all convolutional net with strided convolutions (in discriminator) and transposed convolutions (in generator). In my project, the fully convolutional network that uses transposed convolution layer is introduced to enlarge input size in generator[5, 6].
Second is to remove fully connected layers on top of convolutional net and fully connected hidden layers for deeper architectures, because pooling such as global average pooling lowers convergence speed in spite of increasing model stability. The first layer of generator takes a normal noise distribution z (a length-d vector) as input, and the result is reshaped into a 4-dimensional tensor and propagated to the next layer. The generator in the project consists of four basic blocks (set of connected

layers) that increase input's both width and height and halve the channels at the same time to match the desired shape of real data (a RGB 64×64 image, the channel size is 3). And the last layer of discriminator is flattened and then fed into a single sigmoid output.



As the figure shows, the generator includes four blocks which increase input's width and height from 1 to 32. Noise z with input channel size 100 is projected to the first layer which converts the noise into 1024 channels. And then, the channels are halved by series of four blocks respectively. At last, a transposed convolution layer generates the output matched to the desired 64×64 shape with channel size 3.

Third is Batch Normalization[7] that increases learning stability by normalizing the input to zero mean and unit variance, which can solve problems where initialization is poor, help gradient flow in deeper models and prevent the generator from collapsing all samples to a single point which is a common failure observed in GANs which also appears in the project. Batch normalization are applied to all layers in DCGAN except for the generator output layer and the discriminator input layer to avoid sample oscillation and model instability.
The ReLU activation[8] is applied to blocks mentioned above and the generator output layer uses the Tanh function. In the discriminator, the leaky rectified activation[9, 10] works well, especially for higher resolution modeling.

In conclusion, each block of the generator mentioned above contains a transposed convolution layer followed by the batch normalization and ReLU activation. The tanh activation function in the generator output layer is applied to project output values matched to the size of real data into the (-1,1) range. The discriminator is a normal convolutional network except that it uses a leaky ReLU as its activation function. The basic block of the discriminator is a convolution layer followed by a batch normalization and a leaky ReLU activation. And the last layer is a convolution layer with output channel 1 to obtain a prediction value (usually fed into a sigmoid function to obtain the predicted probability).

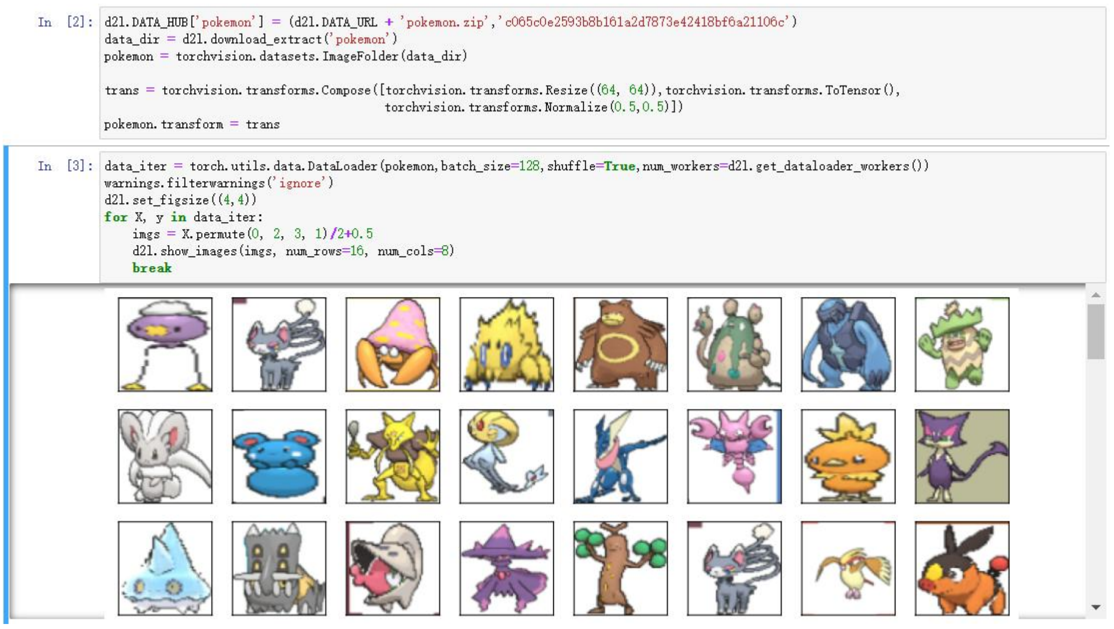
3.3 Training of DCGAN

In the project, the dataset is a collection of Pokemon sprites obtained from d2l[11], and no pre-processing was applied to training images besides resizing each image into 64×64 and scaling them to the range of the tanh activation function [-1,1]. All models are trained with SGD with a minibatch size of 128, and Adam optimizer[12] is used to accelerate training. The suggested learning rate and the momentum term $\beta1$, proposed in Radford et al.[4], is 0.0002 and 0.5 which helps stabilize training. All weights are initialized from a zero-centered normal distribution with standard deviation 0.02. In the leaky ReLU, the slope is set to 0.2 in all layers in the paper and the project.

4 Baseline Implementation

The baseline method in the project is the DCGAN model discussed above, which is re-implemented with PyTorch (requirements: torch==1.12.0, torchvision==0.13.0, d2l==0.17.6). In the practical coding part, some code templates in Dive into deep learning[11] are referred to. The whole coding part is in the file DCGAN0.ipynb.

The followings are some important annotations in coding.



```
In [2]: d2l.DATA_HUB['pokemon'] = (d2l.DATA_URL + 'pokemon.zip','c065c0e2593b8b161a2d7873e42418bf6a21106c')
        data_dir = d2l.download_extract('pokemon')
        pokemon = torchvision.datasets.ImageFolder(data_dir)

        trans = torchvision.transforms.Compose([torchvision.transforms.Resize((64, 64)),torchvision.transforms.ToTensor(),
                                                torchvision.transforms.Normalize(0.5,0.5)])
        pokemon.transform = trans
```

```
In [3]: data_iter = torch.utils.data.DataLoader(pokemon,batch_size=128,shuffle=True,num_workers=d2l.get_dataloader_workers())
        warnings.filterwarnings('ignore')
        d2l.set_figsize((4,4))
        for X, y in data_iter:
            imgs = X.permute(0, 2, 3, 1)/2+0.5
            d2l.show_images(imgs, num_rows=16, num_cols=8)
            break
```

First download, extract and load the dataset which is a collection of Pokemon sprites obtained from d2l[11]. The Resize and ToTensor function are used to resize each image into 64×64 and project the pixel value into [0,1].
And normalize the data with 0.5 mean and 0.5 standard deviation to match the value range of the tanh function, which are mentioned above.

```
G = nn.Sequential(
    seq(nn.ConvTranspose2d(in_channels=100, out_channels=1024, kernel_size=4, stride=1, padding=0, bias=False),nn.BatchNorm2d(1024),
        nn.ReLU()),      # Output: (1024, 4, 4)
    seq(nn.ConvTranspose2d(in_channels=1024, out_channels=512,kernel_size=4, stride=2,padding=1,bias=False),nn.BatchNorm2d(512),
        nn.ReLU()),      # Output: (512, 8, 8)
    seq(nn.ConvTranspose2d(in_channels=512, out_channels=256,kernel_size=4, stride=2,padding=1,bias=False),nn.BatchNorm2d(256),
        nn.ReLU()),      # Output: (256, 16, 16)
    seq(nn.ConvTranspose2d(in_channels=256, out_channels=128,kernel_size=4, stride=2,padding=1,bias=False),nn.BatchNorm2d(128),
        nn.ReLU()),    # Output: (128, 32, 32)
    nn.ConvTranspose2d(in_channels=128, out_channels=3,kernel_size=4, stride=2, padding=1, bias=False),nn.Tanh())  # Output: (3, 64, 64)
```

In general, the transposed convolution increases the height and width of the input by s times for stride s, padding s/2 (an integer) and the height and width of the kernel 2s. If the transposed convolution layer with 4×4 kernel, 1×1 stride and 0 padding is used, with a input size 1×1, the output size is 4×4. In default, the transposed convolution layer with 4×4 kernel, 2×2 stride and 1×1 padding that is a basic block in the generator doubles the height and width of the input.
As discussed above, the generator includes four blocks which increase input's width and height from 1 to 32. Noise z with input channel size 100 is projected to the first layer which converts the noise into 1024 channels. And then, the channels are halved by series of four blocks respectively. At last, a transposed convolution layer generates the output matched to the desired 64×64 shape with channel size 3. The tanh activation is applied to project the output into (-1,1).

```
D=nn.Sequential(
    seq(nn.Conv2d(in_channels=3, out_channels=128, kernel_size=4, stride=2, padding=1, bias=False),nn.BatchNorm2d(128),
        nn.LeakyReLU(negative_slope=0.2, inplace=True)),      # Output: (128, 32, 32)
    seq(nn.Conv2d(in_channels=128, out_channels=256,kernel_size=4, stride=2,padding=1,bias=False),nn.BatchNorm2d(256),
        nn.LeakyReLU(negative_slope=0.2, inplace=True)),      # Output: (256, 16, 16)
    seq(nn.Conv2d(in_channels=256, out_channels=512,kernel_size=4, stride=2,padding=1,bias=False),nn.BatchNorm2d(512),
        nn.LeakyReLU(negative_slope=0.2, inplace=True)),      # Output: (512, 8, 8)
    seq(nn.Conv2d(in_channels=512, out_channels=1024,kernel_size=4, stride=2,padding=1,bias=False),nn.BatchNorm2d(1024),
        nn.LeakyReLU(negative_slope=0.2, inplace=True)),    # Output: (1024, 4, 4)
    nn.Conv2d(in_channels=1024, out_channels=1,kernel_size=4, stride=1, padding=0, bias=False))  # Output: (1,1,1)
```

if X is fed into a convolution layer f to output Y=f(X) and a transposed convolution layer g with the same hyperparameters as f except for the number of output channels being the number of channels in X is created, then g(Y) will have the same shape as X. Therefore, the discriminator is a mirror of the generator, which uses a convolution layer with output channel 1 as the last layer to generate a prediction value.

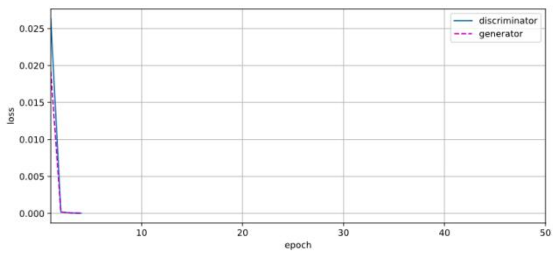
GPU is used to accelerate the computation. The project is on a single NVIDIA GeForce GTX 1650.

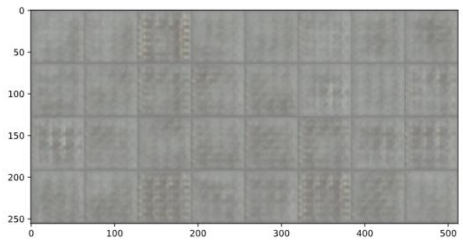
```
def train(D,G,data_iter,epochs,device=d2l.try_gpu()):
```

5 Results and Improvements

5.1 First Edition

In the file DCGAN0.ipynb, as the following figure shows, model collapsed.





The discriminator loss $E_{x\sim P_{data}}[-logD(x)] + E_Z[-log(1 - D(G(z)))]$ and the generator loss $E_Z[-log(1 - D(G(z)))]$ are near 0, which mean $D(x) \approx 1$ and $D(G(z)) \approx 0$.

Take a look at the generator's loss to be maximized:

$$E_Z[-log(1 - D(G(z)))]$$

It's possible that the discriminator is so perfect that $D(G(z)) \approx 0$, which means the loss is near 0 that results in too small gradients to improve the generator. So, the loss is changed to the following formula to be minimized:

$$E_Z[-log(D(G(z)))]$$

5.2 Second Edition

Some code snippets of DCGAN0.ipynb are changed.
To simplify the code, in function update_D, the sentence

```
l = (-1)*torch.log(torch.sigmoid(Y.reshape(Y.shape[0],-1)))+(-1)*torch.log(1-torch.sigmoid(Y1.reshape(Y1.shape[0],-1)))
loss_D=l.sum()/(batch*2)
```

is replaced by the following sentence where the 'loss' is 'nn.BCEWithLogitsLoss(reduction='sum')'.

```
loss_D = (loss(Y, ones.reshape(Y.shape)) +loss(Y1, zeros.reshape(Y1.shape))) / (2*batch)
```

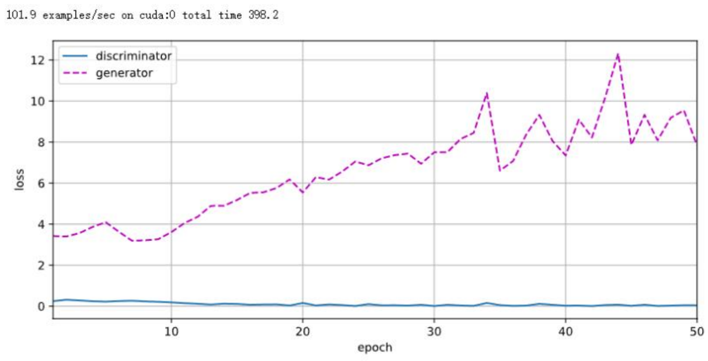
In function update_G, the loss below

```
l = (-1)*torch.log(1-torch.sigmoid(Y.reshape(Y.shape[0],-1)))
loss_G=l.sum()/batch
```

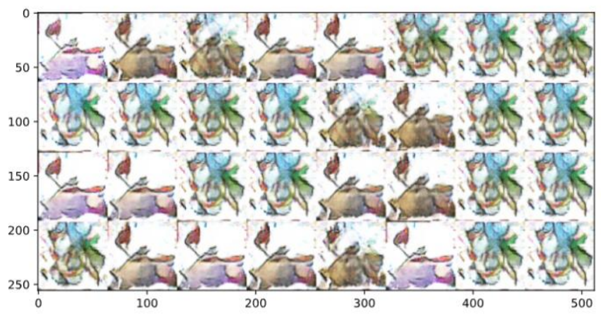
is replaced by the new generator loss mentioned above.

```
loss_G = loss(Y, ones.reshape(Y.shape))/batch   #(-1)*torch.log(torch.sigmoid(Y.reshape(Y.shape[0],-1))).sum()/batch
```

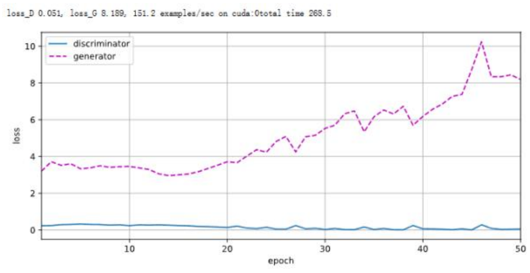
Unfortunately, the result was poor again and total time of 398.2 min was wasted.



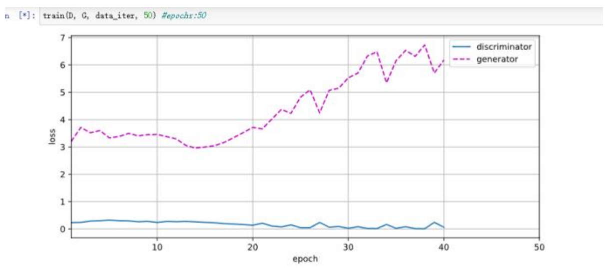
```
101.9 examples/sec on cuda:0 total time 398.2
```



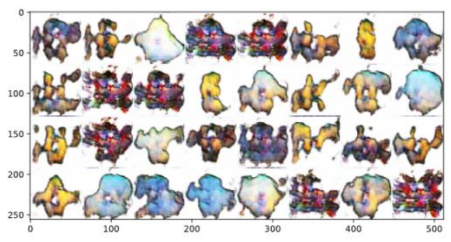
To improve the performance, minibatch stochastic gradients are altered. The gradients of discriminator and generator are not divided by '2*batch' and 'batch' respectively to accelerate gradient descent. The results are as follows.

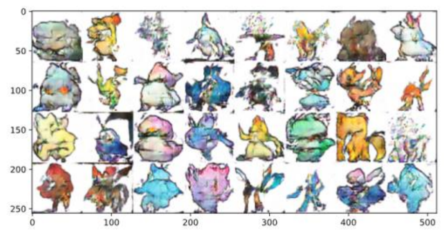


```
loss_D 0.051, loss_G 8.189, 151.2 examples/sec on cuda:0total time 268.5
```



```
In [*]: train(D, G, data_iter, 50) #epochs:50
```



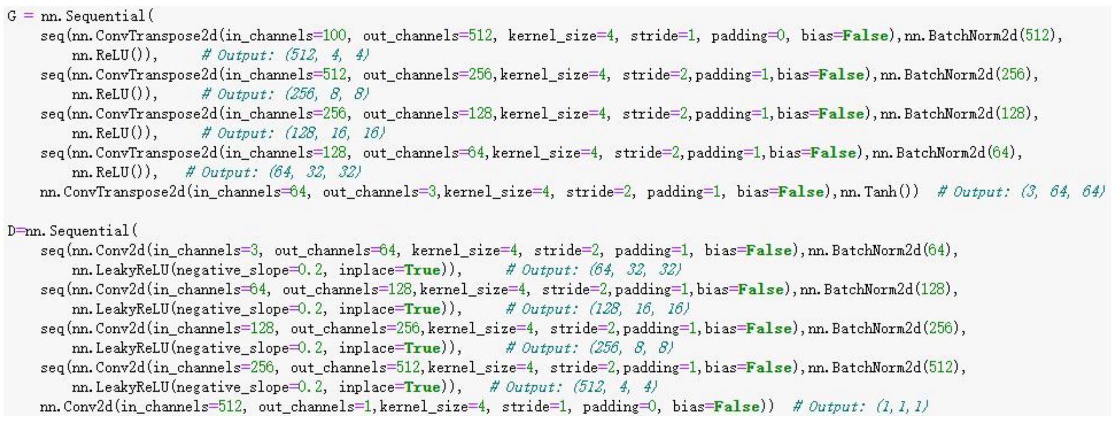


epoch 50, total time of 268.5, poor performance due to uncontrollably increasing generator loss

epoch 40, better performance, the time is much less than 398.2

5.3 Third Edition

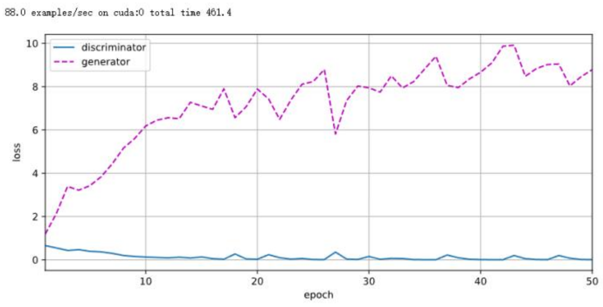
The learning rate and batch size are increased to 0.005 and 256 respectively to accelerate training. In Adam, β1 remains the same to stabilize training.



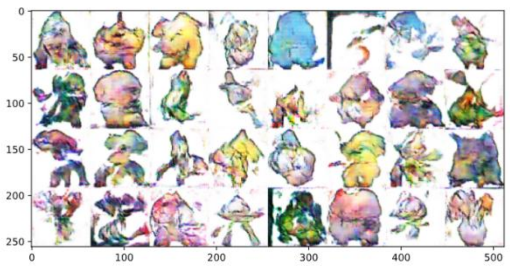
```
G = nn.Sequential(
    seq(nn.ConvTranspose2d(in_channels=100, out_channels=512, kernel_size=4, stride=1, padding=0, bias=False),nn.BatchNorm2d(512),
        nn.ReLU()),      # Output: (512, 4, 4)
    seq(nn.ConvTranspose2d(in_channels=512, out_channels=256,kernel_size=4, stride=2,padding=1,bias=False),nn.BatchNorm2d(256),
        nn.ReLU()),      # Output: (256, 8, 8)
    seq(nn.ConvTranspose2d(in_channels=256, out_channels=128,kernel_size=4, stride=2,padding=1,bias=False),nn.BatchNorm2d(128),
        nn.ReLU()),      # Output: (128, 16, 16)
    seq(nn.ConvTranspose2d(in_channels=128, out_channels=64,kernel_size=4, stride=2,padding=1,bias=False),nn.BatchNorm2d(64),
        nn.ReLU()),    # Output: (64, 32, 32)
    nn.ConvTranspose2d(in_channels=64, out_channels=3,kernel_size=4, stride=2, padding=1, bias=False),nn.Tanh())  # Output: (3, 64, 64)

D=nn.Sequential(
    seq(nn.Conv2d(in_channels=3, out_channels=64, kernel_size=4, stride=2, padding=1, bias=False),nn.BatchNorm2d(64),
        nn.LeakyReLU(negative_slope=0.2, inplace=True)),      # Output: (64, 32, 32)
    seq(nn.Conv2d(in_channels=64, out_channels=128,kernel_size=4, stride=2,padding=1,bias=False),nn.BatchNorm2d(128),
        nn.LeakyReLU(negative_slope=0.2, inplace=True)),      # Output: (128, 16, 16)
    seq(nn.Conv2d(in_channels=128, out_channels=256,kernel_size=4, stride=2,padding=1,bias=False),nn.BatchNorm2d(256),
        nn.LeakyReLU(negative_slope=0.2, inplace=True)),      # Output: (256, 8, 8)
    seq(nn.Conv2d(in_channels=256, out_channels=512,kernel_size=4, stride=2,padding=1,bias=False),nn.BatchNorm2d(512),
        nn.LeakyReLU(negative_slope=0.2, inplace=True)),    # Output: (512, 4, 4)
    nn.Conv2d(in_channels=512, out_channels=1,kernel_size=4, stride=1, padding=0, bias=False))  # Output: (1,1,1)
```

As the figure shows, the original output channel size of each block in generator and discriminator are halved.

The first result below without changes in minibatch stochastic gradients mentioned above:



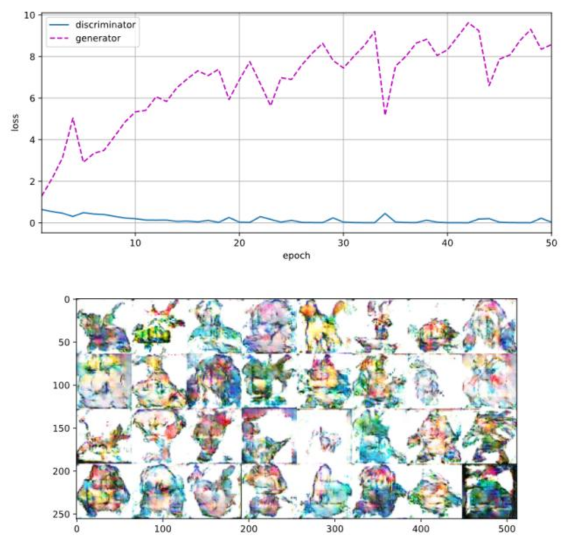
```
88.0 examples/sec on cuda:0 total time 461.4
```



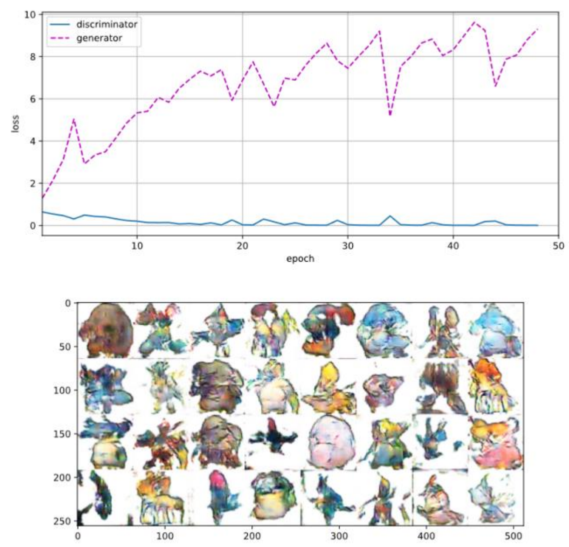
The total time of 461.4 is too long with the same epoch 50, while the performance is not good.

Applying the gradient changes, the results are as follows.

```
loss_D 0.031, loss_G 8.574, 440.3 examples/sec on cuda:0total time 92.2
```



Epoch 50, the total time is reduced sharply, but the generating figures are vague



The performance is better between epoch 40 and 50, with reduced total time.

6 Further Thinking

In the optimal case, $D(x) \approx D(G(z)) \approx 0.5$, while the generator loss and the discriminator loss are both near $\log 2$. In the figures above, uncontrollably increasing generator losses always appear, which demonstrate that results of the training are a perfect discriminator and a low-quality generator.

Therefore, further attention should be paid to the loss function of generator to ‘fight against’ the discriminator instead of succumbing to it.

References

[1] MILDENHALL B, SRINIVASAN P, TANCIK M, et al. Nerf: Representing scenes as neural radiance fields for view synthesis; proceedings of the European conference on computer vision, F, 2020 [C].

[2] BLAU Y, MICHAELI T. The perception-distortion tradeoff; proceedings of the Proceedings of the IEEE conference on computer vision and pattern recognition, F, 2018 [C].

[3] GOODFELLOW I J, POUGET-ABADIE J, MIRZA M, et al. Generative Adversarial Nets [J]. stat, 2014, 1050: 10.

[4] RADFORD A, METZ L, CHINTALA S. Unsupervised representation learning with deep convolutional generative adversarial networks [J]. arXiv preprint arXiv:151106434, 2015.

[5] DUMOULIN V, VISIN F. A guide to convolution arithmetic for deep learning [J]. arXiv preprint arXiv:160307285, 2016.

[6] LONG J, SHELHAMER E, DARRELL T. Fully convolutional networks for semantic segmentation; proceedings of the Proceedings of the IEEE conference on computer vision and pattern recognition, F, 2015 [C].

[7] IOFFE S, SZEGEDY C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift [J]. arXiv preprint arXiv:150203167, 2015.

[8] NAIR V, HINTON G E. Rectified linear units improve restricted boltzmann machines; proceedings of the Icml, F, 2010 [C].

[9] MAAS A L, HANNUN A Y, NG A Y. Rectifier nonlinearities improve neural network acoustic models; proceedings of the Proc icml, F, 2013 [C]. Atlanta, Georgia, USA.

[10] XU B, WANG N, CHEN T, et al. Empirical evaluation of rectified activations in convolutional network [J]. arXiv preprint arXiv:150500853, 2015.

[11] ZHANG A, LIPTON Z C, LI M, et al. Dive into deep learning [J]. arXiv preprint arXiv:210611342, 2021.

[12] KINGMA D P, BA J. Adam: A method for stochastic optimization [J]. arXiv preprint arXiv:14126980, 2014.