Chapter 12

Generative Adversarial Nets

12.1 Learning Objectives

Generative adversarial networks (GANs) [1] [2] was the most exiting innovation in machine learning in the past decade. GANs are generative models, which can create new data instances that resemble the training data. For example, GANs can generate images that look like photographs of human faces, but don't belong to any real person. The core idea of GANs is to pair a generator, which learns to produce the target output, with a discriminator, which learns to distinguish true data from the output of the generator. The generator tries to fool the discriminator by generating the output as much like the true data as possible, and the discriminator tries to keep from being fooled.

In this chapter, we will present the idea, algorithm and architecture of the basic GANs. To make it concrete, we will give a tutorial based on deep convolutional GANs (DCGAN) to generate fake face images. Finally, the variant versions of GANs and their applications will be presented. After finishing this chapter, one should be able to

- Understand the principle of GANs
- Implement a basic GAN in PyTorch
- Extend the tutorial to variations of GANs, such as conditional GAN, Cycle-GAN, etc.
- Be aware of various GANs and their applications

12.2 Basic Principles of GANs

12.2.1 Generative models and discriminative models

Before we discuss GANs, it is helpful to distinguish two type of models: generative and discriminative. Literally, generative models can generate new data instances while discriminative models discriminate between different kinds of data instances. For example, a generative model could generate new photos of human faces that look like real humans. A discriminative model could tell a human face from a cat face. Although a GAN consists of both type models, GANs belong to generative model in terms of their output and purpose.

From the probabilistic point of view, generative models capture the joint probability p(x,y), or just p(x) if there is no label. Discriminative models capture the conditional probability p(y|x). Thus, a generative model learns the distribution of the training data, and tells how likely a given example is. A discriminative model just tells how likely a label is to apply to a given instance x.

In general, generative models tackle a more difficult task than discriminative models. Discriminative models try to draw boundaries in the data space, while generative models try to model how data is placed throughout the space. For example, the following diagram shows discriminative and generative models of handwritten digits. The discriminative model tries to tell the difference between handwritten 0's and 1's by drawing a line in the data space. If it gets the line right, it can distinguish 0's from 1's without ever having to model exactly where the instances are placed in the data space on either side of the line. In contrast, the generative model tries to produce convincing 1's and 0's by generating digits that fall close to their real counterparts in the data space. It has to model the distribution throughout the data space.

12.2.2 Random variable mapping

As we will see later, a generative model is a mapping function that maps a simple random variable to a random variable with a complicated distribution patten.

Consider generating a random variable X with a certain complicated distribution $p_X(x)$ from a random variable Z with a simple distribution $p_Z(z)$ (e.g. uniform distribution). Let $F_Z(z)\ and\ F_X(x)$ be the cumulative distribution functions (CDF) for Z and X, respectively. Here suppose the two CDFs are known (but in practice $F_X(x)$ is usually unknown and needs to be learned). The mapping from Z to X can be found in this way: for an arbitrary value of z, we can locate the corresponding value of x by solving $F_Z(z) = F_X(x)$

For the simplicity and the purpose of illustration, lets consider an example shown in Fig.1. We assume that Z and X are single random variables, and Z has a uniform distribution between 0 and 1, and X is the normal distribution with mean=0 and standard deviation =1. The cumulative distribution function of X is $F_Z(z) = P[Z \leq z] = z$, for 0<z<1. If a map from Z to X, X=G(Z), exists, the cumulative distribution function of Z can be calculated by

$F_X(x) = P[X < x] = P[G(Z) < x]$

$= P[Z < G^{-1}(x)] = F_Z(G^{-1}(x)) = G^{-1}(x) \quad for \ \ 0 \leq G^{-1}(x) \leq 1$

Thus, in this case, the mapping function is $X = G(Z) = {F_X}^{-1}(Z)$

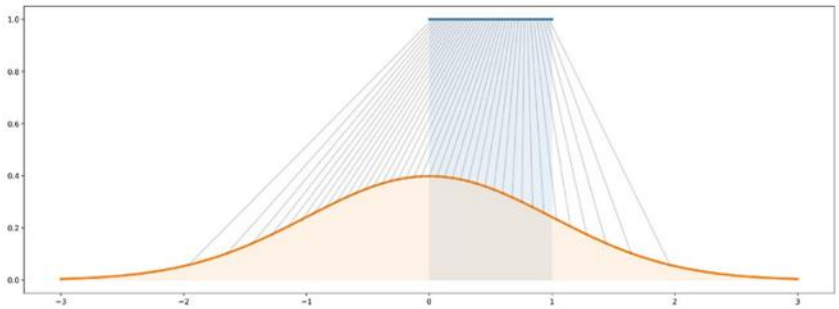


Fig.1, a function mapping uniform random variable to normal distribution [3]. The blue solid line (top) represents the uniform random variable Z, and the orange curve (below) represents the target normal distribution. The horizontal axis represents the normal random variable X. The gray lines show the mapping from Z to X. For example, the point Z=0.5 will map to X=0 because $F_Z(0.5) = F_X(0)$. The point Z=0.01 will map to ${F_X}^{-1}(0.01)$ whose value can be found by inversely looking-up the normal Z-table (area under the curve).

In fact, the generator in GANs plays a role in mapping a random vector to a data space with a very complicated probability distribution that cannot be explicitly represented but to be learned. For example, Fig.1 shows a generative network that maps a random vector to a 5x5 image with a "person" patten distribution. Of course, in real applications X has a much more complicated distribution. The goal of training the generative network is to maximize the probability for a good classifier (discriminator) to classify the output image as a "person".

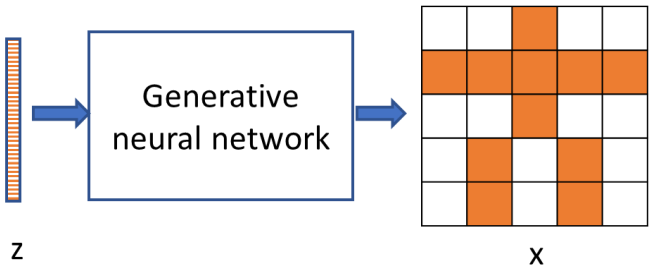


Fig.2 a generative network mapping a noise vector to a "person" image

12.2.3 Original idea of GANs

The original idea of generative adversarial nets (GANs) was proposed in [1]. A GAN consists of two basic components: generator and discriminator, shown in Fig.3. To learn the generator's distribution p_g over data x, the authors define a prior on input noise variable z with a distribution $p_Z(z)$, then represent a mapping to data space as $G(z;\ \theta_g)$, where G (called generator) is a differentiable function represented by a multilayer perceptron with parameters θ_g. They also define a second multiplayer perceptron (called discriminator) $D(x;\ \theta_d)$ that outputs a single scalar, denoted as *D(x)* which predicts the probability that x came from the data rather than p_g. They train D to maximize the probability of assigning the correct label to both training examples and samples from G, and simultaneously train G to minimize $log\left(1 - D\left(G(z)\right)\right)$. In other words, D and G play the following two-player minimax game with value function V(G,D):

$$\min_G \max_D \left\{V(D,G) = \mathrm{E}_{x\sim p_{data}(x)}[logD(x)] + \mathrm{E}_{z\sim p_Z(z)}\left[\log\left(1 - D\left(G(z)\right)\right)\right]\right\} \quad (1)$$

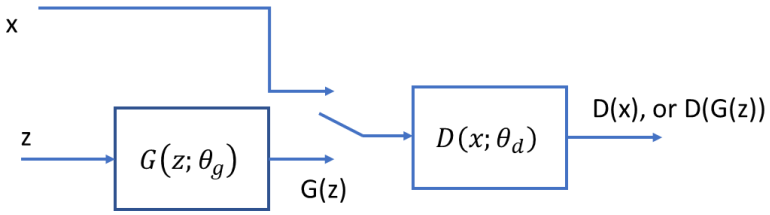


Fig.3 a basic architecture of GANs

A pedagogical explanation is illustrated in Fig. 4. The GANs are trained by simultaneously updating the discriminative distribution (D, blue dashed line) so that it discriminates between samples from the data generating distribution (black, dotted line) p_x from those of the generative distribution $p_g(G)$ (green, solid line). The lower horizontal line is the domain from which z is sampled, in this case uniformly. The horizontal line above is part of the domain of x. the upward

arrows show how the mapping $x = G(z)$ imposes the non-uniform distribution p_g on transformed samples. G contracts in regions of high density and expands in regions of low density of p_g.

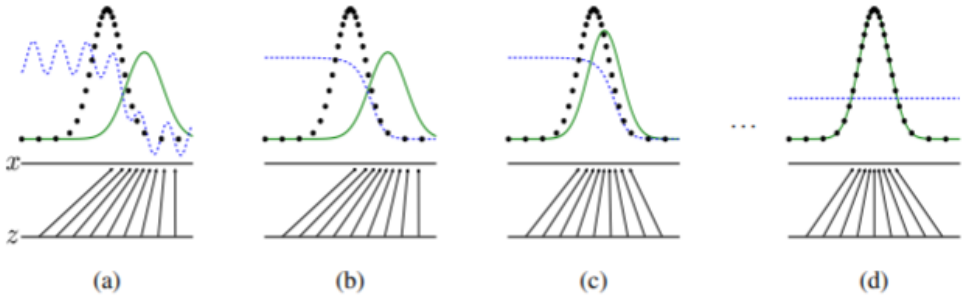


Fig. 4 an explanation of GANs. (a) consider an adversarial pair near convergence: p_g is similar to p_{data} and D is a partially accurate classifier. (b) in the inner loop of the algorithm D is trained to discriminate samples from data, converging to $D^*(x) = \frac{p_{data}(x)}{p_{data}(x)+p_g(x)}$. (c) after an update to G, gradient of D has guided G(z) to flow to regions that are more likely to be classified as data. (d) after several steps of training, if G and D have enough capacity, they will reach a point at which both cannot improve because $p_g = p_{data}$. The discriminator is unable to differentiate between the two distributions, i.e. $D(x) = 0.5$.

12.2.4 Training GANs

A general algorithm for training GANs was given in [1]. This algorithm serves as a basic guideline for us to develop a specific implementation for GANs. The algorithm can be described as below.

Algorithm Minibatch stochastic gradient descent training of GANs. The number of steps to apply to the discriminator, k, is a hyperparameter. We used k=1, the least expensive option, in our experiments

---

for number of training iterations **do**

(# part 1: update the discriminator)

for k steps **do**

- Sample minibatch of *m* noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prio $p_g(z)$.
- Sample minibatch of *m* examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{data}(x)$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[logD\left(x^{(i)}\right) + log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)\right]$$

end for

(# part 2: update the generator)

- Sample minibatch of *m* noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prio $p_g(z)$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)$$

end for

the gradient-based updates can use any standard gradient-based learning rule. [1] used momentum in their experiments.

Note that it may be very difficult to fully understand this theoretical section until a reader has completed the tutorial later in this chapter.

12.3 Structure of GANs

The previous section presents the basic principle of GANs. In this section, we will describe the details of GAN structure from a practical aspect.

12.3.1 Overall structure of GANs

A generative adversarial network (GAN) has two parts: 1) the generator learns to generate plausible data, shown in Fig.5. The generated instances become negative training examples for the discriminator; and 2) the discriminator learns to distinguish the generator's fake data from real data (i.e. positive examples). The discriminator penalizes the generator for producing implausible results.

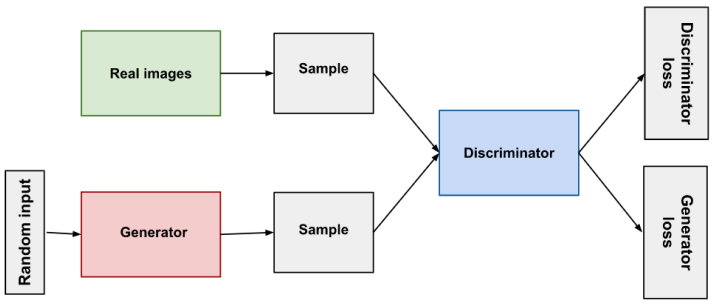


Fig.5 Structure of GANs

When training begins, the generator produces obviously fake data, and the discriminator quickly learns to tell that it's fake, As training progresses, the generator gets closer to producing output that can fool the discriminator. Finally, if generator training goes well, the discriminator gets worse at telling the difference between real and fake. It starts to classify fake data as real, and its accuracy decreases. Both the generator and the discriminator can be realized by neural networks. The generator output is connected directly to the discriminator input. Through backpropagation, the discriminator's classification provides a signal that the generator uses to update its weights.

A) Discriminator

The discriminator in a GAN is simply a classifier. It tries to distinguish real data from the data created by the generator. It could use any network architecture appropriate to the type of data it's classifying. The discriminator's training data comes from two sources: 1) Real data instances as positive examples, such as real pictures of people; and 2) Fake data instances created by the generator as negative examples.

The discriminator connects to two loss functions. During discriminator training, the discriminator ignores the generator loss and just uses the *discriminator loss*. We use the *generator loss* during generator training, as described in the next section. During discriminator training the

generator does not train. Its weights remain constant while it produces examples for the discriminator to train on. Thus the discriminator training process can be described in Fig.6 as:

- The discriminator classifies both real data and fake data from the generator.
- The discriminator loss penalizes the discriminator for misclassifying a real instance as fake or a fake instance as real.
- The discriminator updates its weights through backpropagation from the discriminator loss through the discriminator network.

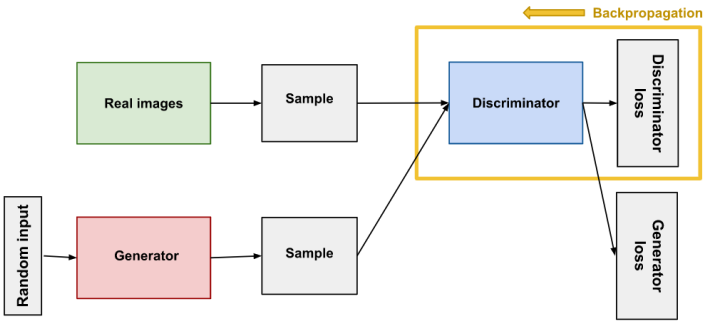


Fig.6 Discriminator training

B) Generator

The generator part of a GAN learns to create fake data by incorporating feedback from the discriminator. It learns to make the discriminator classify its output as real. Generator training requires tighter integration between the generator and the discriminator than discriminator training requires. The portion of the GAN that trains the generator includes: random input, generator network which transforms the random input into a data instance, discriminator network which classifies the generated data, discriminator output, generator loss which penalizes the generator for failing to fool the discriminator, shown in Fig.7.

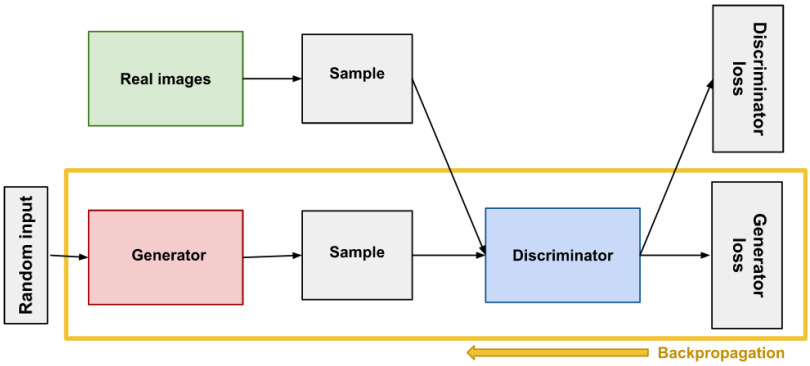


Fig.7 Generator training

In its most basic form, a GAN takes random noise as its input. The generator then transforms this noise into a meaningful output. By introducing noise, we can get the GAN to produce a wide variety of data, sampling from different places in the target distribution. Experiments suggest that the distribution of the noise doesn't matter much, so we can choose something that's easy to sample from, like a uniform distribution. Note that some GANs use non-random input to shape the output (See GAN Variations).

To train a neural net, we adjust the net's weights to reduce the error or loss of its output. In our GAN, however, the generator is not directly connected to the loss that we're trying to affect. The generator feeds into the discriminator net, and the discriminator produces the output we're trying to affect. The generator loss penalizes the generator for producing a sample that the discriminator network classifies as fake.

During generator training, we don't want the discriminator to change. So we train the generator with the following procedure:

- Sample random noise.
- Produce generator output from sampled random noise.
- Get discriminator "Real" or "Fake" classification for generator output.
- Calculate loss from discriminator classification.
- Backpropagate through both the discriminator and generator to obtain gradients.
- Use gradients to change only the generator weights.

C) Alternating two training processes

The generator and the discriminator have different training processes. GAN training proceeds in alternating periods: 1) The discriminator trains for one or more batches/epochs. 2) The generator trains for one or more batches/epochs. 3) Repeat steps 1) and 2) to continue to train the generator and discriminator networks. We keep the generator constant during the discriminator training phase. Similarly, we keep the discriminator constant during the generator training phase.

As the generator improves with training, the discriminator performance gets worse because the discriminator can't easily tell the difference between real and fake. If the generator succeeds perfectly, then the discriminator has a 50% accuracy. This progression poses a problem for convergence of the GAN as a whole: the discriminator feedback gets less meaningful over time. If the GAN continues training past the point when the discriminator is giving completely random feedback, then the generator starts to train on junk feedback, and its own quality may collapse.

D) Loss functions

1) Cross-entropy loss function

In the paper [1] that introduced GANs, the generator tries to minimize the following function while the discriminator tries to maximize it:

$$E_x[log(D(x))] + E_z\big[log\big(1 - D(G(z))\big)\big] \quad (2)$$

where

D(x) is the discriminator's estimate of the probability that real data instance x is real.
Ex is the expected value over all real data instances.
G(z) is the generator's output when given noise z.
D(G(z)) is the discriminator's estimate of the probability that a fake instance is real.
Ez is the expected value over all random inputs to the generator, equivalently, the expected value over all generated fake instances G(z).

The generator can't directly affect the log(D(x)) term in the function, so, for the generator, minimizing the loss is equivalent to minimizing log(1 - D(G(z))). The original GAN paper notes that the above minimax loss function can cause the GAN to get stuck in the early stages of GAN

training when the discriminator's job is very easy. The paper therefore suggests modifying the generator loss so that the generator tries to maximize log D(G(z)).

2) Wasserstein loss function

In Wasserstein GAN" or "WGAN" [4], the discriminator does not actually classify instances. For each instance it outputs a number. This number does not have to be less than one or greater than 0, so we can't use 0.5 as a threshold to decide whether an instance is real or fake. Discriminator training just tries to make the output bigger for real instances than for fake instances. Because it can't really discriminate between real and fake, the WGAN discriminator is actually called a "critic" instead of a "discriminator". This distinction has theoretical importance, but for practical purposes we can treat it as an acknowledgement that the inputs to the loss functions don't have to be probabilities. The loss functions themselves are deceptively simple:

Critic (discriminator) Loss: D(x) - D(G(z))

The discriminator tries to maximize this function. In other words, it tries to maximize the difference between its output on real instances and its output on fake instances.

Generator Loss: D(G(z))

The generator tries to maximize this function. In other words, it tries to maximize the discriminator's output for its fake instances.

In the above two loss functions,
D(x) is the critic's output for a real instance.
G(z) is the generator's output when given noise z.
D(G(z)) is the critic's output for a fake instance.
The output of critic D does not have to be between 1 and 0.
The formulas derive from the earth mover distance between the real and generated distributions.

The theoretical justification for the Wasserstein GAN (or WGAN) requires that the weights throughout the GAN be clipped so that they remain within a constrained range. Wasserstein GANs are less vulnerable to getting stuck than minimax-based GANs, and avoid problems with vanishing gradients. The earth mover distance also has the advantage of being a true metric: a measure of distance in a space of probability distributions. Cross-entropy is not a metric in this sense.

12.4 Tutorial: Generating faces by DCGAN using celeb-A dataset

(file: users/weido/GANs/dcgan_tutorial.ipynb)

A DCGAN (deep convolutional GAN) is a direct extension of the basic GAN described in [1]. It explicitly uses convolutional and convolutional-transpose layers in the discriminator and generator, respectively. This tutorial is adopted from [5], which is based on the work in [6]

12.4.1 Basic settings

Import packages

```
from __future__ import print_function
#%matplotlib inline
import argparse
import os
```

```
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML

# Set random seed for reproducibility
manualSeed = 999
#manualSeed = random.randint(1, 10000) # use if you want new results
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed)
```

define some hyperparameters

```
# Root directory for dataset
dataroot = "data/celeba"

# Number of workers for dataloader
workers = 2

# Batch size during training
batch_size = 128

# Spatial size of training images. All images will be resized to this
#   size using a transformer.
image_size = 64

# Number of channels in the training images. For color images this is 3
nc = 3

# Size of z latent vector (i.e. size of generator input)
nz = 100

# Size of feature maps in generator
ngf = 64

# Size of feature maps in discriminator
ndf = 64

# Number of training epochs
num_epochs = 5

# Learning rate for optimizers
lr = 0.0002

# Beta1 hyperparam for Adam optimizers
beta1 = 0.5

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 0
```

prepare data:

In this tutorial we will use the Celeb-A Faces dataset which can be downloaded at the linked site, or in Google Drive. The dataset will download as a file named img_align_celeba.zip. Once downloaded, create a directory named celeba and extract the zip file into that directory. Then, set the dataroot input for this notebook to the celeba directory you just created. The resulting directory structure should be:

```
/path/to/celeba
    -> img_align_celeba
        -> 188242.jpg
        -> 173822.jpg
        -> 284702.jpg
        -> 537394.jpg
           ...
```

This is an important step because we will be using the ImageFolder dataset class, which requires there to be subdirectories in the dataset's root folder. Now, we can create the dataset, create the dataloader, set the device to run on, and finally visualize some of the training data.

```
# We can use an image folder dataset the way we have it setup.
# Create the dataset
dataset = dset.ImageFolder(root=dataroot,
                           transform=transforms.Compose([
                               transforms.Resize(image_size),
                               transforms.CenterCrop(image_size),
                               transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                           ]))
# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=workers)

# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else "cpu")

# Plot some training images
real_batch = next(iter(dataloader))
plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Training Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64], padding=2,
normalize=True).cpu(),(1,2,0)))
```

12.4.2 Generator

The generator, G, is designed to map the latent space vector (z) to data space. Since our data are images, converting z to data space means ultimately creating an RGB image with the same size as the training images (i.e. 3x64x64). In practice, this is accomplished through a series of strided two-dimensional transposed convolution layers, each (except the output layer) paired with a 2D batch norm layer and a ReLU activation. The output of the generator is fed through a tanh function to return it to the input data range of [−1,1]. The architecture is shown in Fig.8.

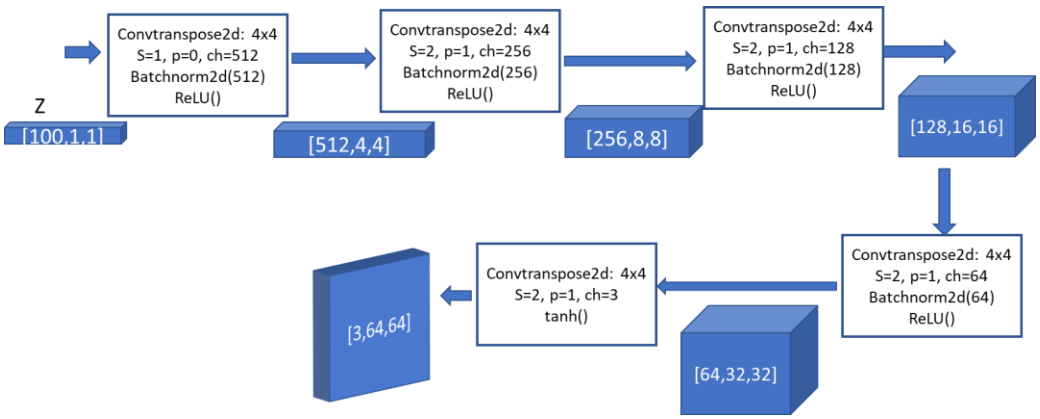


Fig.8 architecture of generator

```
# Generator Code

class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. (nc) x 64 x 64
        )

    def forward(self, input):
        return self.main(input)
```

12.4.3 Discriminator

The discriminator, D, is a binary classification network that takes an image as input and outputs a scalar probability that the input image is real (as opposed to fake). Here, D takes a 3x64x64 input image, processes it through a series of Conv2d, BatchNorm2d, and LeakyReLU layers, and outputs the final probability through a Sigmoid activation function. This architecture can be extended with more layers if necessary for the problem, but there is significance to the use of the strided convolution, BatchNorm (except for input and output layers), and LeakyReLUs. The DCGAN paper [6] mentions it is a good practice to use strided convolution rather than pooling to downsample because it lets the network learn its own pooling function. Also batch norm and leaky relu functions promote healthy gradient flow which is critical for the learning process of both G and D. The architecture is shown in Fig.9.

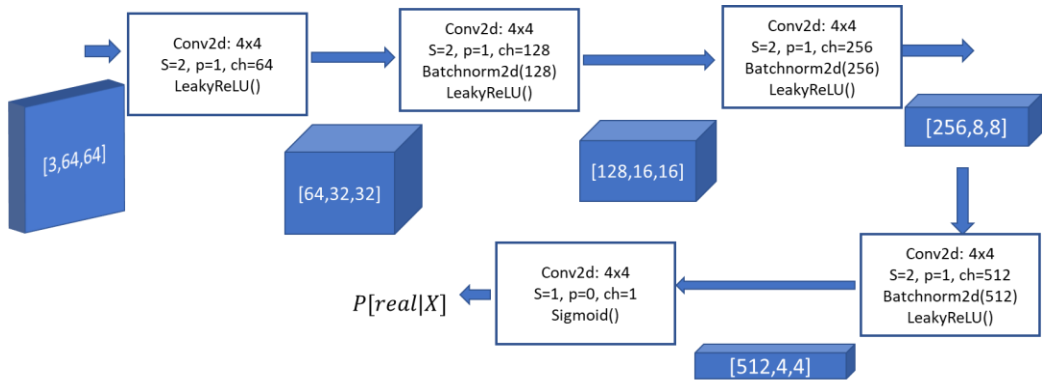


Fig.9 architecture of discriminator

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

    def forward(self, input):
        return self.main(input)
```

12.4.4 Loss functions and optimizers

With D and G setup, we can specify how they learn through the loss functions and optimizers. We will use the Binary Cross Entropy loss (BCELoss) function which is defined in PyTorch as:

$$\ell(x, y) = (l_1, \dots, l_N)^T$$

where

N is the batch size,

$l_n = -(y_n log x_n + (1 - y_n) \log(1 - x_n))$ is the loss for the nth prediction x_n,

y_n is the target (or label 0 or 1).

Notice how this function provides the calculation of both log components in the objective function (i.e. log(D(x)) and log(1−D(G(z)))). We can specify what part of the BCE equation to use with the y input. This is accomplished in the training loop which is coming up soon, but it is important to understand how we can choose which component we wish to calculate just by changing y (i.e. ground truth labels).

Next, we define our real label as 1 and the fake label as 0. These labels will be used when calculating the losses of D and G, and this is also the convention used in the original GAN paper. Finally, we set up two separate optimizers, one for D and one for G. As specified in the DCGAN paper, both are Adam optimizers with learning rate 0.0002 and Beta1 = 0.5. For keeping track of the generator's learning progression, we will generate a fixed batch of latent vectors that are drawn from a Gaussian distribution (i.e. fixed_noise) . In the training loop, we will periodically input this fixed_noise into G, and over the iterations we will see images form out of the noise.

```
# Initialize BCELoss function
criterion = nn.BCELoss()

# Create batch of latent vectors that we will use to visualize
#  the progression of the generator
fixed_noise = torch.randn(64, nz, 1, 1, device=device)

# Establish convention for real and fake labels during training
real_label = 1.
fake_label = 0.

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))
```

12.4.5 Training

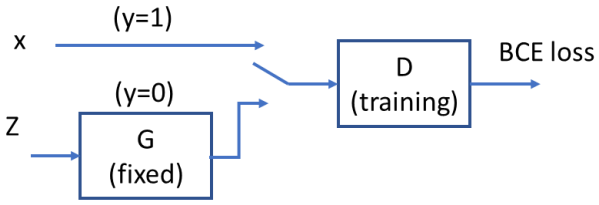
Finally, now that we have all of the parts of the GAN framework defined, we can train it. Be mindful that training GANs is somewhat of an art form, as incorrect hyperparameter settings lead to mode collapse with little explanation of what went wrong. Here, we will closely follow Algorithm from Goodfellow's paper (presented in Section 12.2.4), while abiding by some of the best practices shown in ganhacks [7]. Namely, we will "construct mini-batches for real and fake" images separately, and also adjust G's objective function to maximize logD(G(z)). Training is split up into two main parts. Part 1 updates the Discriminator and Part 2 updates the Generator.

Part 1 - Train the Discriminator

The goal of training the discriminator is to maximize the probability of correctly classifying a given input as real with label y=1 or fake with y=0. In terms of Goodfellow, we wish to "update the discriminator by ascending its stochastic gradient". Practically, we want to minimize the negative of (log(D(x))+log(1−D(G(z)))). In this tutorial, we implement it by minimizing the binary cross entropy loss function

$$BCEloss = -\left[ylog\left(D(x)\right) + (1-y)\log\left(1 - D\left(G(z)\right)\right)\right] \quad (3)$$

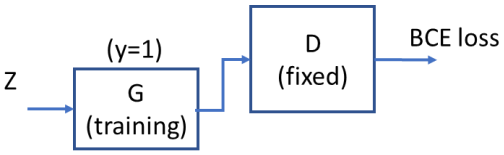
Due to the separate mini-batch suggestion from ganhacks [7], we will calculate this in two steps. First, we will construct a batch of real samples (i.e. y=1) from the training set, forward pass through D, calculate the BCEloss (i.e. -log(D(x))), then calculate the gradients through a backward pass. Secondly, we will construct a batch of fake samples G(z) (i.e. y=0) with the current generator, forward pass this batch through D, calculate the BCEloss (i.e., -(log(1−D(G(z))))), and accumulate the gradients with a backward pass. Now, with the gradients accumulated from both the all-real and all-fake batches, we call a step of the Discriminator's optimizer to update the parameters of discriminator.



$$BCEloss = -(ylog\left(D(x)\right) + (1-y)\log\left(1 - D(G(z))\right))$$

Part 2 - Train the Generator

We train the Generator by minimizing log(1−D(G(z))) to generate better fakes. As mentioned, this was shown by Goodfellow to not provide sufficient gradients, especially early in the learning process. As a fix, we instead wish to maximize log(D(G(z))), equivalently to minimize -log(D(G(z))). In the code we accomplish this by: classifying the Generator output from Part 1 with the Discriminator, computing G's loss using real labels (i.e. y=1) as ground truth, computing G's gradients in a backward pass, and finally updating G's parameters with an optimizer step. It may seem counter-intuitive to use the real labels for the loss function, but this allows us to use the log(x) part of the BCELoss (rather than the log(1−x) part) which is exactly what we want.



$$BCEloss = -\log(D(G(z)))$$

Finally, we will do some statistic reporting and at the end of each epoch we will push our fixed_noise batch through the generator to visually track the progress of G's training. The training statistics reported are:

Loss_D - discriminator loss calculated as the sum of losses for the all real and all fake batches - (log(D(x))+log(D(G(z)))).

Loss_G - generator loss calculated as -log(D(G(z)))

D(x) - the average output (across the batch) of the discriminator for the all real batch. This should start close to 1 then theoretically converge to 0.5 when G gets better.

D(G(z)) - average discriminator outputs for the all fake batch. The first number is before D is updated and the second number is after D is updated. These numbers should start near 0 and converge to 0.5 as G gets better.

Note that to prevent the CPU from overheating, we let the CPU take a 5-minute break every 50 mini-batches training. So, the total training time is longer. If a GPU is available, the 5-minute break is not necessary and the training process will be much faster.

```
# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
# For each epoch
for epoch in range(num_epochs):
    # For each batch in the dataloader
    for i, data in enumerate(dataloader, 0):

        ############################
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
        ###########################
        ## Train with all-real batch
        netD.zero_grad()
        # Format batch
        real_cpu = data[0].to(device)
        b_size = real_cpu.size(0)
        label = torch.full((b_size,), real_label, dtype=torch.float, device=device)
        # Forward pass real batch through D
        output = netD(real_cpu).view(-1)
        # Calculate loss on all-real batch
        errD_real = criterion(output, label)
        # Calculate gradients for D in backward pass
        errD_real.backward()
        D_x = output.mean().item()

        ## Train with all-fake batch
        # Generate batch of latent vectors
        noise = torch.randn(b_size, nz, 1, 1, device=device)
        # Generate fake image batch with G
        fake = netG(noise)
        label.fill_(fake_label)
        # Classify all fake batch with D
        output = netD(fake.detach()).view(-1)
        # Calculate D's loss on the all-fake batch
        errD_fake = criterion(output, label)
        # Calculate the gradients for this batch
        errD_fake.backward()
```

```
        D_G_z1 = output.mean().item()
        # Add the gradients from the all-real and all-fake batches
        errD = errD_real + errD_fake
        # Update D
        optimizerD.step()

        ############################
        # (2) Update G network: maximize log(D(G(z)))
        ###########################
        netG.zero_grad()
        label.fill_(real_label)  # fake labels are real for generator cost
        # Since we just updated D,perform another forward pass of all-fake batch through D
        output = netD(fake).view(-1)
        # Calculate G's loss based on this output
        errG = criterion(output, label)
        # Calculate gradients for G
        errG.backward()
        D_G_z2 = output.mean().item()
        # Update G
        optimizerG.step()

        # Output training stats
        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.4f\tD(G(z)): %.4f / %.4f'
                  % (epoch, num_epochs, i, len(dataloader),
                     errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))
            print('taking a break for 5 minutes')  # CPU take a rest
            time.sleep(300)
        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())

        # Check how the generator is doing by saving G's output on fixed_noise
        if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i == len(dataloader)-1)):
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2, normalize=True))

        iters += 1
```

```
Starting Training Loop...
[0/1][0/1583]      Loss_D: 0.1976     Loss_G: 3.3648  D(x): 0.9384  D(G(z)): 0.1122 / 0.0503
taking a break for 5 minutes
[0/1][50/1583]     Loss_D: 0.6748     Loss_G: 4.0421  D(x): 0.7592  D(G(z)): 0.2695 / 0.0268
taking a break for 5 minutes
[0/1][100/1583]    Loss_D: 0.4434     Loss_G: 5.1498  D(x): 0.9148   D(G(z)): 0.2673 / 0.0105
taking a break for 5 minutes
[0/1][150/1583]    Loss_D: 0.4131     Loss_G: 4.3651  D(x): 0.8716   D(G(z)): 0.1876 / 0.0295
taking a break for 5 minutes
[0/1][200/1583]    Loss_D: 0.5046     Loss_G: 4.8253  D(x): 0.7675   D(G(z)): 0.1245 / 0.0134
taking a break for 5 minutes
[0/1][250/1583]    Loss_D: 0.4082     Loss_G: 3.6981  D(x): 0.7877   D(G(z)): 0.1075 / 0.0380
taking a break for 5 minutes
[0/1][300/1583]    Loss_D: 0.3060     Loss_G: 5.9304  D(x): 0.9174   D(G(z)): 0.1757 / 0.0040
taking a break for 5 minutes
[0/1][350/1583]    Loss_D: 0.3892     Loss_G: 5.7550  D(x): 0.8887   D(G(z)): 0.1855 / 0.0056
taking a break for 5 minutes
[0/1][400/1583]    Loss_D: 1.0067     Loss_G: 6.5052  D(x): 0.8518   D(G(z)): 0.4989 / 0.0025
taking a break for 5 minutes
[0/1][450/1583]    Loss_D: 0.4848     Loss_G: 3.9813  D(x): 0.7815   D(G(z)): 0.1467 / 0.0311
taking a break for 5 minutes
[0/1][500/1583]    Loss_D: 0.5387     Loss_G: 4.9735  D(x): 0.6706   D(G(z)): 0.0156 / 0.0139
taking a break for 5 minutes
```

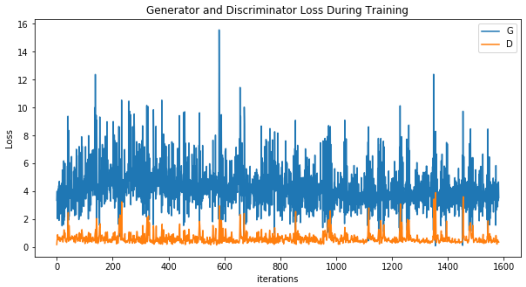
```
[0/1][550/1583]    Loss_D: 0.3011     Loss_G: 4.8817  D(x): 0.8146   D(G(z)): 0.0486 / 0.0144
taking a break for 5 minutes
[0/1][600/1583]    Loss_D: 0.5776     Loss_G: 5.0378  D(x): 0.8858   D(G(z)): 0.3219 / 0.0120
taking a break for 5 minutes
[0/1][650/1583]    Loss_D: 0.3496     Loss_G: 4.5710  D(x): 0.8871   D(G(z)): 0.1673 / 0.0194
taking a break for 5 minutes
[0/1][700/1583]    Loss_D: 0.5622     Loss_G: 4.9007  D(x): 0.8343   D(G(z)): 0.2398 / 0.0158
taking a break for 5 minutes
[0/1][750/1583]    Loss_D: 0.5222     Loss_G: 5.1435  D(x): 0.6746   D(G(z)): 0.0111 / 0.0164
taking a break for 5 minutes
[0/1][800/1583]    Loss_D: 0.7444     Loss_G: 2.2189  D(x): 0.6812   D(G(z)): 0.1837 / 0.1672
taking a break for 5 minutes
[0/1][850/1583]    Loss_D: 0.4962     Loss_G: 3.9515  D(x): 0.7276   D(G(z)): 0.0311 / 0.0482
taking a break for 5 minutes
[0/1][900/1583]    Loss_D: 0.6815     Loss_G: 2.7870  D(x): 0.6108   D(G(z)): 0.0291 / 0.0928
taking a break for 5 minutes
[0/1][950/1583]    Loss_D: 0.3756     Loss_G: 4.3074  D(x): 0.8685   D(G(z)): 0.1633 / 0.0218
taking a break for 5 minutes
[0/1][1000/1583]   Loss_D: 0.6879     Loss_G: 4.2626  D(x): 0.8134   D(G(z)): 0.3038 / 0.0303
taking a break for 5 minutes
[0/1][1050/1583]   Loss_D: 0.3484     Loss_G: 3.0436  D(x): 0.8039   D(G(z)): 0.0716 / 0.0660
taking a break for 5 minutes
[0/1][1100/1583]   Loss_D: 0.3156     Loss_G: 3.2511  D(x): 0.8260   D(G(z)): 0.0798 / 0.0634
taking a break for 5 minutes
[0/1][1150/1583]   Loss_D: 0.7224     Loss_G: 1.8951  D(x): 0.5986   D(G(z)): 0.0577 / 0.2239
taking a break for 5 minutes
[0/1][1200/1583]   Loss_D: 0.3060     Loss_G: 4.2853  D(x): 0.8422   D(G(z)): 0.0907 / 0.0254
taking a break for 5 minutes
[0/1][1250/1583]   Loss_D: 0.5452     Loss_G: 3.4728  D(x): 0.8104   D(G(z)): 0.2310 / 0.0490
taking a break for 5 minutes
[0/1][1300/1583]   Loss_D: 0.2874     Loss_G: 3.7413  D(x): 0.8834   D(G(z)): 0.1242 / 0.0388
taking a break for 5 minutes
[0/1][1350/1583]   Loss_D: 3.4598     Loss_G: 3.7833  D(x): 0.0903   D(G(z)): 0.0002 / 0.0579
taking a break for 5 minutes
[0/1][1400/1583]   Loss_D: 0.4932     Loss_G: 4.2762  D(x): 0.9340   D(G(z)): 0.2918 / 0.0334
taking a break for 5 minutes
[0/1][1450/1583]   Loss_D: 0.4163     Loss_G: 4.8419  D(x): 0.9020   D(G(z)): 0.2332 / 0.0136
taking a break for 5 minutes
[0/1][1500/1583]   Loss_D: 0.6055     Loss_G: 2.7326  D(x): 0.6361   D(G(z)): 0.0296 / 0.0950
taking a break for 5 minutes
[0/1][1550/1583]   Loss_D: 0.9763     Loss_G: 4.4269  D(x): 0.9779   D(G(z)): 0.5443 / 0.0252
taking a break for 5 minutes
```

12.4.5 Results

Here, we will look at the results from three different aspects. First, we will see how D and G's losses changed during training. Second, we will visualize G's output on the fixed_noise batch for every epoch. And third, we will look at a batch of real data next to a batch of fake data from G.

Loss versus training iteration

```
plt.figure(figsize=(10,5))
plt.title("Generator and Discriminator Loss During Training")
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



Visualization of G's progression

Remember how we saved the generator's output on the fixed_noise batch after every epoch of training. Now, we can visualize the training progression of G with an animation. Press the play button to start the animation.

```
#%%capture
fig = plt.figure(figsize=(8,8))
plt.axis("off")
ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for i in
img_list]
ani = animation.ArtistAnimation(fig, ims, interval=1000,
repeat_delay=1000, blit=True)

HTML(ani.to_jshtml())
```

Real Images vs. Fake Images

Finally, let's take a look at some real images and fake images side by side.

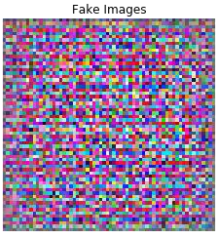
```
# Grab a batch of real images from the dataloader
real_batch = next(iter(dataloader))

# Plot the real images
plt.figure(figsize=(15,15))
plt.subplot(1,2,1)
plt.axis("off")
plt.title("Real Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64],
padding=5, normalize=True).cpu(),(1,2,0)))

# Plot the fake images from the last epoch
plt.subplot(1,2,2)
plt.axis("off")
plt.title("Fake Images")
plt.imshow(np.transpose(img_list[-1],(1,2,0)))
plt.show()
```

Save the trained model

data_path = ''
torch.save(netG.state_dict(), data_path + 'parameters_netG.pt')
torch.save(netD.state_dict(), data_path + 'parameters_netD.pt')

instantiate G1 with initial parameters

```
G1=Generator(ngpu).to(device)
fake = G1(fixed_noise).detach().cpu()
plt.axis("off")
plt.title("Fake Images")
plt.imshow(np.transpose(fake[0]*0.5+0.5,(1,2,0)))
plt.show()
```



Load the saved model parameters

```
G1.load_state_dict(torch.load(data_path + 'parameters_netG.pt'))
fake = G1(fixed_noise).detach().cpu()
plt.axis("off")
plt.title("Fake Images")
plt.imshow(np.transpose(fake[0]*0.5+0.5,(1,2,0)))
plt.show()
```

12.4.6 Generating handwritten digits using MNIST training set

The above tutorial can be adopted to generate handwritten digits using training set MNIST. We make the following modification.

1) Architectures of G and D: double the number of channels for each hidden layer, as shown in Fig. 10.
2) Image channel nc=3 (color) is changed to 1 (black/white).
3) The input bias for all hidden layers is set to True (i.e. Default). This is not required but preferred. There is no significant impact on the result.
4) We use GPU to speed up the training process through Google Colab.

The following codes show the changes. Also, we need to change the G and D module class definitions for doubling the channels. The results are shown in Fig.11 and Fig.12.

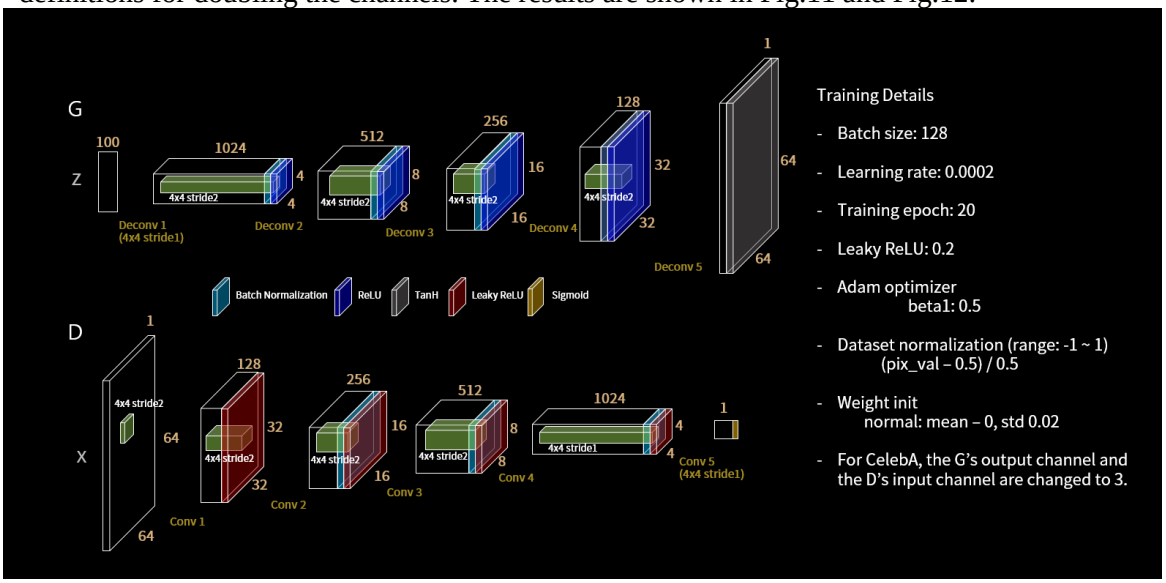


Fig. 10 GAN architecture for MNIST and training parameters (from [8])

(Google colab: mnist_dcgan.ipynb)

```
from __future__ import print_function
#%matplotlib inline
import argparse
import os
import random
import time
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML
import torchvision
import torchvision.transforms as transforms
```

```
# Set random seed for reproducibility
manualSeed = 999
#manualSeed = random.randint(1, 10000) # use if you want new results
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed)


#from google.colab import drive
#drive.mount('/content/drive')

# Root directory for dataset
dataroot = "./data"       #/content/data

# Number of workers for dataloader
workers = 2

# Batch size during training
batch_size = 128

# Spatial size of training images. All images will be resized to this
#   size using a transformer.
image_size = 64

# Number of channels in the training images. For color images this is 3
nc = 1   # 3 for color

# Size of z latent vector (i.e. size of generator input)
nz = 100

# Size of feature maps in generator
ngf = 64

# Size of feature maps in discriminator
ndf = 64

# Number of training epochs
num_epochs = 20

# Learning rate for optimizers
lr = 0.0002

# Beta1 hyperparam for Adam optimizers
beta1 = 0.5

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1

# We can use an image folder dataset the way we have it setup.
# Create the dataset
transform=transforms.Compose([
                               transforms.Resize(image_size),
                               transforms.ToTensor(),
                               transforms.Normalize((0.5,), (0.5,))])
dataset = torchvision.datasets.MNIST(root=dataroot, train=True,
                                     download=True, transform=transform)
# temparorly download the MNIST training set to /content/dataroot/ if run by
google colab

# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=workers)
```

```
# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else
"cpu")

device

classes = ('0', '1', '2', '3',
           '4', '5', '6', '7', '8', '9')
examples = enumerate(dataloader)
batch_idx, (example_data, example_targets) = next(examples)
example_data.shape


import matplotlib.pyplot as plt

# Plot some training images
real_batch = next(iter(dataloader))
plt.figure(figsize=(16,16))
plt.axis("off")
plt.title("Training Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64],
padding=2, normalize=True).cpu(),(1,2,0)))

# custom weights initialization called on netG and netD
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)  #
nn.init.normal_(m.weight.data, 1.0, 0.02) original
        nn.init.constant_(m.bias.data, 0)

# Generator Code

class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d( nz, ngf * 16, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 16),
            nn.ReLU(True),
            # state size. (ngf*16=1024) x 4 x 4
            nn.ConvTranspose2d(ngf * 16, ngf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8=512) x 8 x 8
            nn.ConvTranspose2d( ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4=256) x 16 x 16
            nn.ConvTranspose2d( ngf * 4, ngf*2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf*2),
            nn.ReLU(True),
            # state size. (ngf*2=128) x 32 x 32
            nn.ConvTranspose2d( ngf*2, nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. (nc) x 64 x 64
        )

    def forward(self, input):
```

```
        return self.main(input)

# Create the generator
netG = Generator(ngpu).to(device)

# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
    netG = nn.DataParallel(netG, list(range(ngpu)))

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.2.
netG.apply(weights_init)

# Print the model
print(netG)

class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is (nc) x 64 x 64
            nn.Conv2d(nc, ndf*2, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*2=128) x 32 x 32
            nn.Conv2d(ndf*2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*4=256) x 16 x 16
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*8=512) x 8 x 8
            nn.Conv2d(ndf * 8, ndf * 16, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 16),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*16=1024) x 4 x 4
            nn.Conv2d(ndf * 16, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)

# Create the Discriminator
netD = Discriminator(ngpu).to(device)

# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
    netD = nn.DataParallel(netD, list(range(ngpu)))

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.2.
netD.apply(weights_init)

# Print the model
print(netD)

# Initialize BCELoss function
criterion = nn.BCELoss()

# Create batch of latent vectors that we will use to visualize
#  the progression of the generator
```

```
fixed_noise = torch.randn(64, nz, 1, 1, device=device)

# Establish convention for real and fake labels during training
real_label = 1.
fake_label = 0.

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))

# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
# For each epoch
for epoch in range(num_epochs):
    # For each batch in the dataloader
    for i, data in enumerate(dataloader, 0):

        ############################
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
        ###########################
        ## Train with all-real batch
        netD.zero_grad()
        # Format batch
        real_cpu = data[0].to(device)
        b_size = real_cpu.size(0)
        label = torch.full((b_size,), real_label, dtype=torch.float,
device=device)
        # Forward pass real batch through D
        output = netD(real_cpu).view(-1)
        # Calculate loss on all-real batch
        errD_real = criterion(output, label)
        # Calculate gradients for D in backward pass
        errD_real.backward()
        D_x = output.mean().item()

        ## Train with all-fake batch
        # Generate batch of latent vectors
        noise = torch.randn(b_size, nz, 1, 1, device=device)
        # Generate fake image batch with G
        fake = netG(noise)
        label.fill_(fake_label)
        # Classify all fake batch with D
        output = netD(fake.detach()).view(-1)
        # Calculate D's loss on the all-fake batch
        errD_fake = criterion(output, label)
        # Calculate the gradients for this batch
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        # Add the gradients from the all-real and all-fake batches
        errD = errD_real + errD_fake
        # Update D
        optimizerD.step()

        ############################
        # (2) Update G network: maximize log(D(G(z)))
        ###########################
```

```
        netG.zero_grad()
        label.fill_(real_label)  # fake labels are real for generator cost
        # Since we just updated D, perform another forward pass of all-fake
batch through D
        output = netD(fake).view(-1)
        # use the fake in (1), another option: generate new fake using new z
        # but no big difference.
        # Calculate G's loss based on this output
        errG = criterion(output, label)
        # Calculate gradients for G
        errG.backward()
        D_G_z2 = output.mean().item()
        # Update G
        optimizerG.step()

        # Output training stats
        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x):
%.4f\tD(G(z)): %.4f / %.4f'
                  % (epoch, num_epochs, i, len(dataloader),
                     errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))
            #print('taking a break for 5 minutes')
            #time.sleep(300)

        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())

        # Check how the generator is doing by saving G's output on fixed_noise
        if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i ==
len(dataloader)-1)):
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2, normalize=True))

        iters += 1

plt.figure(figsize=(10,5))
plt.title("Generator and Discriminator Loss During Training")
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()

#%%capture
fig = plt.figure(figsize=(8,8))
plt.axis("off")
ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for i in img_list]
ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000,
blit=True)

HTML(ani.to_jshtml())

# real images vs fake image

# Grab a batch of real images from the dataloader
real_batch = next(iter(dataloader))

# Plot the real images
plt.figure(figsize=(15,15))
plt.subplot(1,2,1)
```

```
plt.axis("off")
plt.title("Real Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64],
padding=5, normalize=True).cpu(),(1,2,0)))

# Plot the fake images from the last epoch
plt.subplot(1,2,2)
plt.axis("off")
plt.title("Fake Images")
plt.imshow(np.transpose(img_list[-1],(1,2,0)))
plt.show()
```

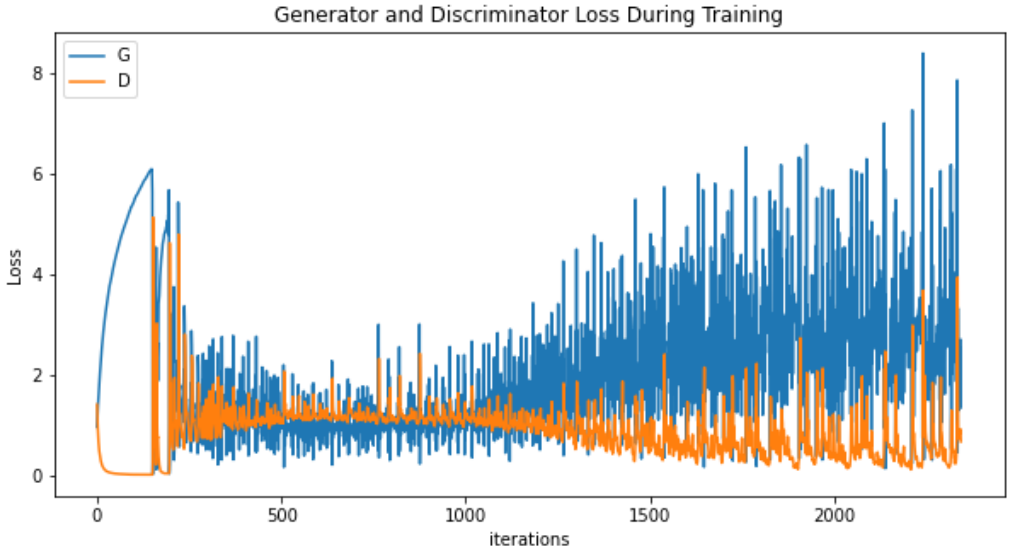


Fig. 11 Loss plots for D and G for 5 epochs (469 batches per epoch)

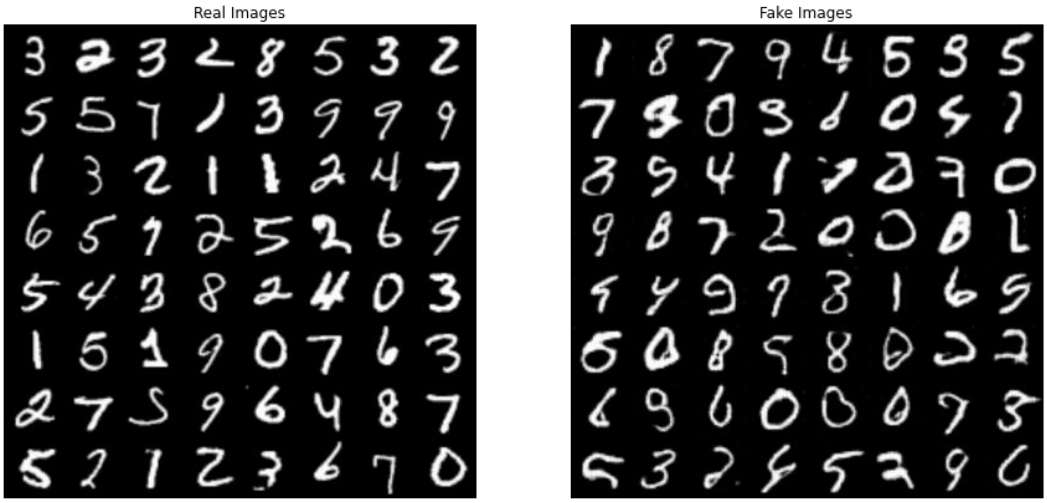


Fig.12 Real images versus fake images

12.5 Conditional Generative Adversarial Nets

In the previous section, if we train a DCGAN to generate new MNIST images, there is no control over which specific digits will be produced by the Generator. There is no mechanism for how to request a particular digit from the Generator. This problem can be addressed by a variation of GAN called Conditional GAN (CGAN) [9].

12.5.1 Principle of conditional generative adversarial nets

Generative adversarial nets can be extended to a conditional model if both the generator and discriminator are conditioned on some extra information y. y could be any kind of auxiliary information, such as class labels or data from other modalities. We can perform the conditioning by feeding y into the both the discriminator and generator as additional input layer. For example, in order for the generator to generate a particular digit image, we could add an additional input layer with the value of one-hot-encoded image label.

In the generator the prior input noise $p_Z(z)$, and y are combined in joint hidden representation, and the adversarial training framework allows for considerable flexibility in how this hidden representation is composed. In the discriminator x and y are presented as inputs and to a discriminative function. Fig. 13 shows the generic architecture of conditional GANs. The objective function of a two-player minimax game would be as

$$\min_G \max_D V(D,G) = \mathrm{E}_{x\sim p_{data}(x)}[logD(x|y)] + \mathrm{E}_{z\sim p_z(z)}\left[log\left(1 - D\big(G(z|y)\big)\right)\right] \quad (4)$$

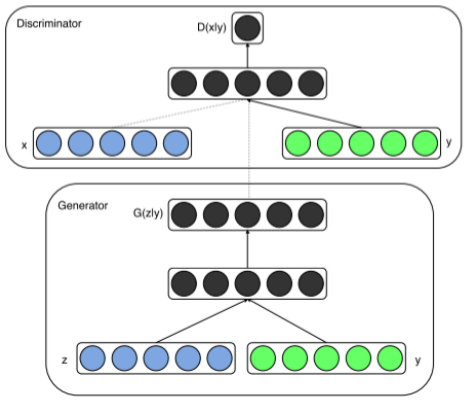


Fig. 13 Conditional adversarial nets (from [9])

12.5.2 A tutorial of conditional GAN on MNIST

To demonstrate how a conditional GAN can be implemented, we present a tutorial of a conditional GAN on MNIST dataset, which can generate a digit image specified by its label $y \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

The overall architecture of the conditional GAN is shown in Fig.14. The input of discriminator (D) consists of an image (real or fake) and its corresponding label image. In our case the size of image x is 32x32. The label image is encoded in one-hot-feature image format 10x32x32. For example, if the label y=3, the third feature map [3,:,:] of the label image is all-ones while all other

feature maps are all-zeros. The generator (G) receives a noise vector z and one-hot label (y) vector as the input, and delivers a fake image for the class y, G(z|y). It is also necessary to encode the label in the one-hot-feature image format to form a fake training example. The details of G and D are illustrated by Fig.15. How the input x or z is combined with the label y is flexible in general. In our case, we choose this way: the first hidden layers for z and y label in the generator are separate, and their outputs are concatenated (or merged) as the input of the next layer. A similar way is applied to discriminator.

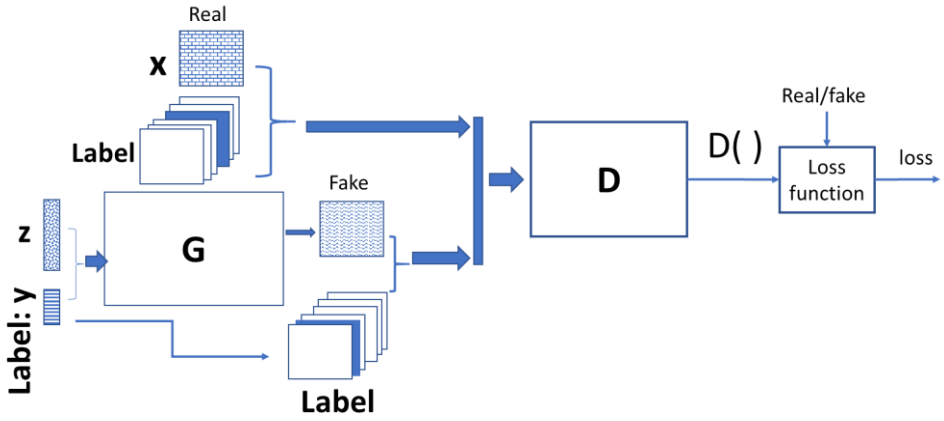


Fig.14 overall architecture of conditional GAN for MNIST

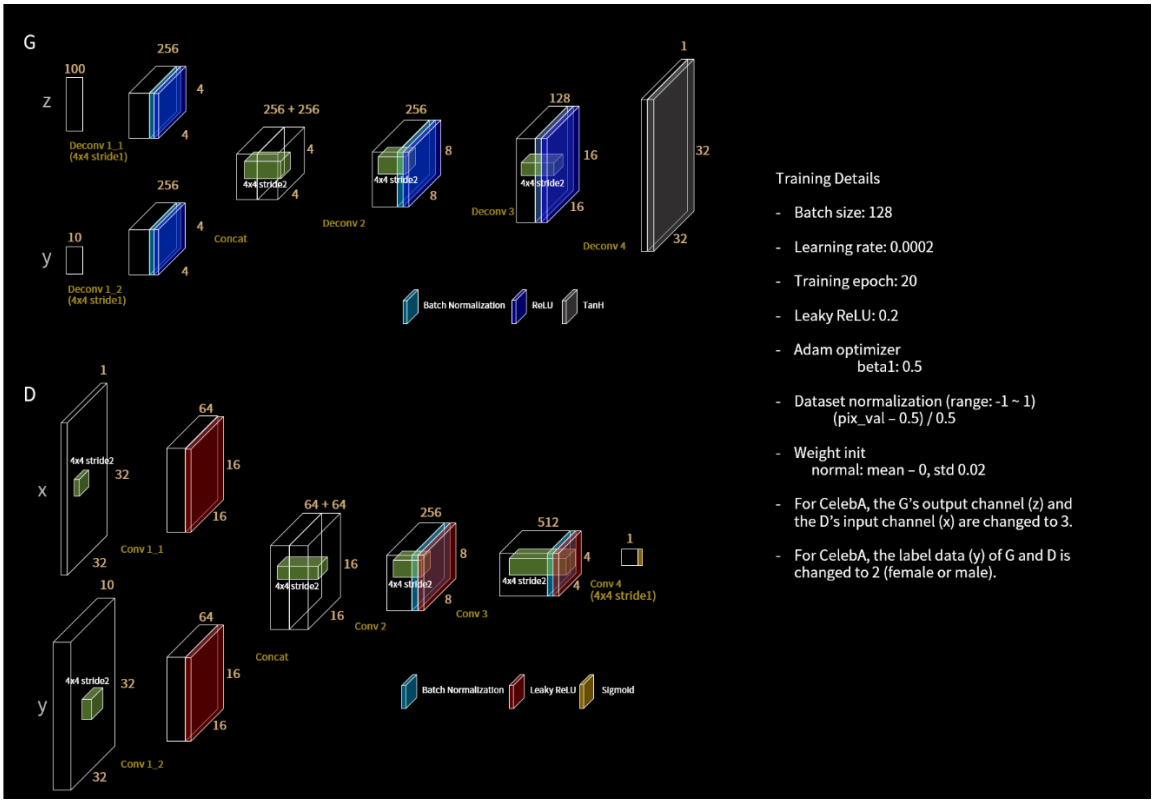


Fig. 15 Generator and discriminator for conditional GAN on MNIST (from [8])

Training process is almost the same as the one we used to train basic GANs in Section 12.4, except that some efforts are needed to generator label one-hot-feature image. The results are shown in Fig.16 and Fig.17.

(google colab: mnist_cdcgan.ipynb)

```
from __future__ import print_function
#%matplotlib inline
import argparse
import os
import random
import time
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML
import torchvision
import torchvision.transforms as transforms
import torch.nn.functional as F
from torch.autograd import Variable


# Set random seed for reproducibility
manualSeed = 999
#manualSeed = random.randint(1, 10000) # use if you want new results
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed)


#from google.colab import drive
#drive.mount('/content/drive')

# Root directory for dataset
dataroot = "./data"       #/content/data

# Number of workers for dataloader
workers = 2

# Batch size during training
batch_size = 128

# Spatial size of training images. All images will be resized to this
#   size using a transformer.
image_size = 32 #64

# Number of channels in the training images. For color images this is 3
nc = 1   # 3 for color
```

```
# Size of z latent vector (i.e. size of generator input)
nz = 100

# Size of feature maps in generator
ngf = 128  # 64

# Size of feature maps in discriminator
ndf = 128  #64

# Number of training epochs
num_epochs = 20

# Learning rate for optimizers
lr = 0.0002

# Beta1 hyperparam for Adam optimizers
beta1 = 0.5

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1

# We can use an image folder dataset the way we have it setup.
# Create the dataset
transform=transforms.Compose([
                               transforms.Resize(image_size),
                               transforms.ToTensor(),
                               transforms.Normalize((0.5,), (0.5,))])
dataset = torchvision.datasets.MNIST(root=dataroot, train=True,
                                     download=True, transform=transform)
# temparorly download the MNIST training set to /content/dataroot/ if run by
google colab

# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=workers)

# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else
"cpu")

device

classes = ('0', '1', '2', '3',
           '4', '5', '6', '7', '8', '9')
examples = enumerate(dataloader)
batch_idx, (example_data, example_targets) = next(examples)
example_data.shape


import matplotlib.pyplot as plt

# Plot some training images
real_batch = next(iter(dataloader))
plt.figure(figsize=(16,16))
plt.axis("off")
plt.title("Training Images")
```

```
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64],
padding=2, normalize=True).cpu(),(1,2,0)))

# custom weights initialization called on netG and netD
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)  #
nn.init.normal_(m.weight.data, 1.0, 0.02) original
        nn.init.constant_(m.bias.data, 0)

# Generator Code

class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.deconv1_1 = nn.ConvTranspose2d(100,ngf*2, 4, 1, 0)
        self.deconv1_1_bn = nn.BatchNorm2d(ngf*2)
        self.deconv1_2 = nn.ConvTranspose2d(10, ngf*2, 4, 1, 0)
        self.deconv1_2_bn = nn.BatchNorm2d(ngf*2)
        self.deconv2 = nn.ConvTranspose2d(ngf*4, ngf*2, 4, 2, 1)
        self.deconv2_bn = nn.BatchNorm2d(ngf*2)
        self.deconv3 = nn.ConvTranspose2d(ngf*2, ngf, 4, 2, 1)
        self.deconv3_bn = nn.BatchNorm2d(ngf)
        self.deconv4 = nn.ConvTranspose2d(ngf, 1, 4, 2, 1)

    def forward(self, input, label):
        x = F.relu(self.deconv1_1_bn(self.deconv1_1(input)))
        y = F.relu(self.deconv1_2_bn(self.deconv1_2(label)))
        x = torch.cat([x,y],1)
        x = F.relu(self.deconv2_bn(self.deconv2(x)))
        x = F.relu(self.deconv3_bn(self.deconv3(x)))
        x = torch.tanh(self.deconv4(x))

        return x

# Create the generator
netG = Generator(ngpu).to(device)

# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
    netG = nn.DataParallel(netG, list(range(ngpu)))

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.2.
netG.apply(weights_init)

# Print the model
print(netG)

class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.conv1_1 = nn.Conv2d(1,int(ndf/2), 4, 2, 1)
```

```
        self.conv1_2 = nn.Conv2d(10, int(ndf/2), 4, 2, 1)
        self.conv2 = nn.Conv2d(ndf, ndf*2, 4, 2, 1)
        self.conv2_bn = nn.BatchNorm2d(ndf*2)
        self.conv3 = nn.Conv2d(ndf*2, ndf*4, 4, 2, 1)
        self.conv3_bn = nn.BatchNorm2d(ndf*4)
        self.conv4 = nn.Conv2d(ndf*4, 1, 4, 1, 0)


    def forward(self, input, label):
        x = F.leaky_relu(self.conv1_1(input), 0.2)
        y = F.leaky_relu(self.conv1_2(label), 0.2)
        x = torch.cat([x,y], 1)
        x = F.leaky_relu(self.conv2_bn(self.conv2(x)), 0.2)
        x = F.leaky_relu(self.conv3_bn(self.conv3(x)), 0.2)
        x = torch.sigmoid(self.conv4(x))
        return x

# Create the Discriminator
netD = Discriminator(ngpu).to(device)

# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
    netD = nn.DataParallel(netD, list(range(ngpu)))

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.2.
netD.apply(weights_init)

# Print the model
print(netD)

temp_z_ = torch.randn(10, 100)
fixed_z_ = temp_z_
fixed_y_ = torch.zeros(10,1)

for i in range(9):
    fixed_z_ = torch.cat([fixed_z_, temp_z_], 0)
    temp = torch.ones(10, 1) + i
    fixed_y_ = torch.cat([fixed_y_, temp], 0)

fixed_z_=fixed_z_.view(-1,100,1,1)

fixed_y_label_ = torch.zeros(100,10)
fixed_y_label_.scatter_(1, fixed_y_.type(torch.LongTensor), 1)
fixed_y_label_ = fixed_y_label_.view(-1, 10, 1, 1)

fixed_z_, fixed_y_label_ = Variable(fixed_z_.cuda()),
Variable(fixed_y_label_.cuda())
#fixed_z_, fixed_y_label_ = Variable(fixed_z_.cuda(), volatile=True),
Variable(fixed_y_label_.cuda(), volatile=True)

# Initialize BCELoss function
criterion = nn.BCELoss()

# Create batch of latent vectors that we will use to visualize
#  the progression of the generator
fixed_noise = torch.randn(64, nz, 1, 1, device=device)
```

```
# Establish convention for real and fake labels during training
real_label = 1.
fake_label = 0.

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))

# label preprocess
onehot = torch.zeros(10, 10)
onehot = onehot.scatter_(1, torch.LongTensor([0, 1, 2, 3, 4, 5, 6, 7, 8,
9]).view(10,1), 1).view(10, 10, 1, 1)
fill = torch.zeros([10, 10, image_size, image_size])
for i in range(10):
    fill[i, i, :, :] = 1
#onehot [10,10,1,1]
#fill [10,10,32,32]

# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
# For each epoch
for epoch in range(num_epochs):
    # For each batch in the dataloader
    y_real_ = torch.ones(batch_size)
    y_fake_ = torch.zeros(batch_size)
    y_real_, y_fake_ = Variable(y_real_.cuda()), Variable(y_fake_.cuda())
    # y_real_=[96]: [1,1,1,1,...,1]
    #
    for i, data in enumerate(dataloader, 0):

        ############################
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
        ###########################
        ## Train with all-real batch
        netD.zero_grad()
        # Format batch
        real_cpu = data[0].to(device)
        b_size = real_cpu.size(0)
        label = torch.full((b_size,), real_label, dtype=torch.float,
device=device)
        if b_size != batch_size:
            y_real_ = torch.ones(b_size)
            y_fake_ = torch.zeros(b_size)
            y_real_, y_fake_ = Variable(y_real_.cuda()),
Variable(y_fake_.cuda())
        # y_real_ is [1,1,1...,1]: [b_size]
        # y_fake_ is [0,0,0,...,0]: [b_size]
        y_fill_ = fill[data[1]]  # y_fill_ : [b_size, 10, 32, 32]
        real_cpu, y_fill_ = Variable(real_cpu.cuda()), Variable(y_fill_.cuda())
        #real_cpu: [b_size, 1, 32,32]
        # Forward pass real batch through D
```

```
        output = netD(real_cpu, y_fill_).view(-1)         # output: [b_size]
        # Calculate loss on all-real batch
        errD_real = criterion(output, y_real_)
        # Calculate gradients for D in backward pass
        errD_real.backward()
        D_x = output.mean().item()

        ## Train with all-fake batch
        # Generate batch of latent vectors
        z_ = torch.randn((b_size, 100)).view(-1, 100, 1, 1)
        # z_ is [b_size, 100, 1, 1]
        y_ = (torch.rand(b_size, 1) * 10).type(torch.LongTensor).squeeze()
        # y_ is [b_size]
        y_label_ = onehot[y_]    #: [b_size, 10, 1, 1]
        y_fill_ = fill[y_]  #: [b_size, 10, 32, 32]
        z_, y_label_, y_fill_ = Variable(z_.cuda()), Variable(y_label_.cuda()),
Variable(y_fill_.cuda())


        #noise = torch.randn(b_size, nz, 1, 1, device=device)
        # Generate fake image batch with G
        fake = netG(z_, y_label_)
        #label.fill_(fake_label)
        # Classify all fake batch with D
        output = netD(fake.detach(), y_fill_).view(-1)
        # Calculate D's loss on the all-fake batch
        errD_fake = criterion(output, y_fake_)
        # Calculate the gradients for this batch
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        # Add the gradients from the all-real and all-fake batches
        errD = errD_real + errD_fake
        # Update D
        optimizerD.step()

        ############################
        # (2) Update G network: maximize log(D(G(z)))
        ###########################
        netG.zero_grad()
        #label.fill_(real_label)  # fake labels are real for generator cost
        # Since we just updated D, perform another forward pass of all-fake
batch through D
        z_ = torch.randn((b_size, 100)).view(-1, 100, 1, 1)
        y_ = (torch.rand(b_size, 1) * 10).type(torch.LongTensor).squeeze()
        y_label_ = onehot[y_]
        y_fill_ = fill[y_]
        z_, y_label_, y_fill_ = Variable(z_.cuda()), Variable(y_label_.cuda()),
Variable(y_fill_.cuda())
        fake=netG(z_, y_label_)
        output = netD(fake, y_fill_).view(-1)
        # Calculate G's loss based on this output
        errG = criterion(output, y_real_)
        # Calculate gradients for G
        errG.backward()
        D_G_z2 = output.mean().item()
        # Update G
        optimizerG.step()
```

```
        # Output training stats
        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x):
%.4f\tD(G(z)): %.4f / %.4f'
                  % (epoch, num_epochs, i, len(dataloader),
                     errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))
            #print('taking a break for 5 minutes')
            #time.sleep(300)

        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())

        # Check how the generator is doing by saving G's output on fixed_noise
        if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i ==
len(dataloader)-1)):
            with torch.no_grad():
                fake = netG(fixed_z_, fixed_y_label_).detach().cpu()
            img_list.append(vutils.make_grid(fake, nrow=10, padding=2,
normalize=True))

        iters += 1

y_label_.size()

plt.figure(figsize=(10,5))
plt.title("Generator and Discriminator Loss During Training")
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()

#%%capture
fig = plt.figure(figsize=(8,8))
plt.axis("off")
ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for i in img_list]
ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000,
blit=True)

HTML(ani.to_jshtml())

img_list[0].size()

# real images vs fake image

# Grab a batch of real images from the dataloader
real_batch = next(iter(dataloader))

# Plot the real images
plt.figure(figsize=(15,15))
plt.subplot(1,2,1)
plt.axis("off")
plt.title("Real Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64],
padding=5, normalize=True).cpu(),(1,2,0)))
```

```
# Plot the fake images from the last epoch
plt.subplot(1,2,2)
plt.axis("off")
plt.title("Fake Images")
plt.imshow(np.transpose(img_list[-1],(1,2,0)))
plt.show()
```

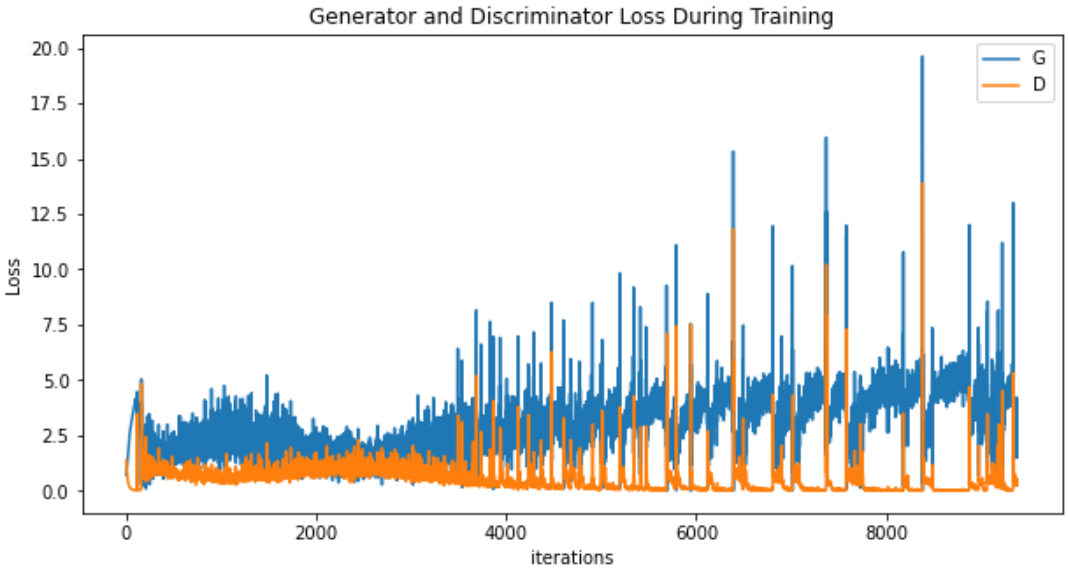


Fig.16 loss plots of G and D

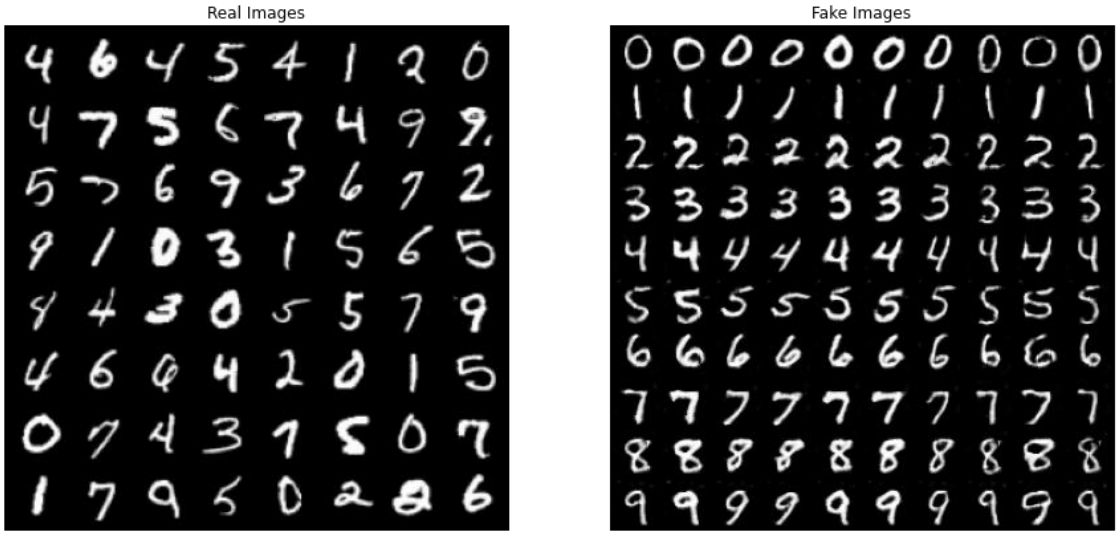


Fig. 17 visual comparison between real images and fake images

12.6 GAN Variantions

Researchers continue to find improved GAN techniques and new uses for GANs. Here's a sampling of GAN variations to give you a sense of the possibilities.

12.6.1 Progressive GANs

In a progressive GAN [10], the generator's first layers produce very low resolution images, and subsequent layers add details. This technique allows the GAN to train more quickly than comparable non-progressive GANs, and produces higher resolution images.

12.6.2 Conditional GANs

Conditional GANs [9] train on a labeled data set and let you specify the label for each generated instance, described in Section 12.5. For example, an unconditional MNIST GAN would produce random digits, while a conditional MNIST GAN would let you specify which digit the GAN should generate. Instead of modeling the joint probability P(X, Y), conditional GANs model the conditional probability P(X | Y).

12.6.3 Image-to-Image Translation

Image-to-Image translation GANs take an image as input and map it to a generated output image with different properties. For example, we can train an image-to-image GAN to take sketches of handbags and turn them into photorealistic images of handbags [11].

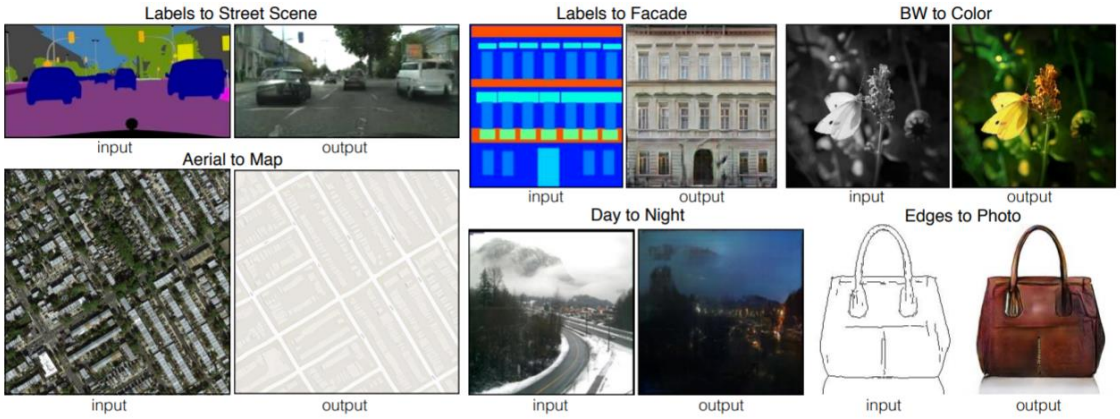


Fig. 10 Image-to-image translation (from [11])

12.6.4 CycleGAN

CycleGANs learn to transform images from one set into images that could plausibly belong to another set. For example, a CycleGAN [12] produced the righthand image below when given the lefthand image as input. It took an image of a horse and turned it into an image of a zebra. An image of a horse running, and a second image that's identical in all respects except that the horse is a zebra. The training data for the CycleGAN is simply two sets of images (in this case, a set of horse images and a set of zebra images). The system requires no labels or pairwise correspondences between images.

Fig.11 Image-to-image translation by CycleGAN [12]

12.6.5 Super-resolution

Super-resolution GANs increase the resolution of images, adding detail where necessary to fill in blurry areas. For example, the blurry middle image below is a down-sampled version of the original image on the left. Given the blurry image, a GAN produced the sharper image on the right [13]. The GAN-generated image looks very similar to the original image, but if you look closely at the headband you'll see that the GAN didn't reproduce the starburst pattern from the original. Instead, it made up its own plausible pattern to replace the pattern erased by the down-sampling.

Original Blurred Restored with GAN

Fig.12 A painting of a girl wearing an elaborate headdress. The headband of the headdress is knit in a complex pattern. A blurry version of the painting of a girl wearing an elaborate headdress. A sharp, clear painting of a girl wearing an elaborate headdress. This painting is almost identical to the first image in this table, but some of the details of the patterns on her headdress and clothing are subtly different.

12.6.6 Face Inpainting

GANs have been used for the semantic image inpainting task. In the inpainting task, chunks of an image are blacked out, and the system tries to fill in the missing chunks. Yeh et al, 2017 used a GAN to outperform other techniques for inpainting images of faces [14].

Fig. 13 Face inpainting [14]

12.6.7 Text-to-Image Synthesis

Text-to-image GANs take text as input and produce images that are plausible and described by the text [15]. For example, the flower image below was produced by feeding a text description to a GAN. Note that in this system the GAN can only produce images from a small set of classes.

Fig.14 Text-to-Image synthesis (from [15])

12.6.8 Text-to-Speech

Not all GANs produce images. For example, researchers have also used GANs to produce synthesized speech from text input [16].

12.7 Common Challenges of GANs

GANs have a number of common failure modes. All of these common problems are areas of active research. While none of these problems have been completely solved, we'll mention some things that people have tried.

12.7.1 Vanishing gradients

Research has suggested that if the discriminator is too good, then generator training can fail due to vanishing gradients. In effect, an optimal discriminator doesn't provide enough information for the generator to make progress. The Wasserstein loss is designed to prevent vanishing gradients even when we train the discriminator to optimality.

12.7.2 Mode collapse

Usually we want our GAN to produce a wide variety of outputs. We want, for example, a different face for every random input to the face generator. However, if a generator produces an especially plausible output, the generator may learn to produce only that output. In fact, the generator is always trying to find the one output that seems most plausible to the discriminator.

If the generator starts producing the same output (or a small set of outputs) over and over again, the discriminator's best strategy is to learn to always reject that output. But if the next generation of discriminator gets stuck in a local minimum and doesn't find the best strategy, then it's too easy for the next generator iteration to find the most plausible output for the current discriminator.

Each iteration of generator over-optimizes for a particular discriminator, and the discriminator never manages to learn its way out of the trap. As a result the generators rotate through a small set of output types. This form of GAN failure is called mode collapse.

The following approaches try to force the generator to broaden its scope by preventing it from optimizing for a single fixed discriminator:

- Wasserstein loss: The Wasserstein loss alleviates mode collapse by letting you train the discriminator to optimality without worrying about vanishing gradients. If the discriminator doesn't get stuck in local minima, it learns to reject the outputs that the generator stabilizes on. So the generator has to try something new.
- Unrolled GANs: Unrolled GANs use a generator loss function that incorporates not only the current discriminator's classifications, but also the outputs of future discriminator versions. So the generator can't over-optimize for a single discriminator.

12.7.3 GAN evaluation

Generally, deep learning models are trained until the convergence of the cost function. However, GAN exploits the balance between the generator and the discriminator for training. Thus, one of the problems with using adversarial networks to make a fair comparison is to evaluate the strengths and weaknesses of various models. To the best of the researchers' knowledge, no consensus has been reached so far on the estimation of relative or absolute quality and developed cost function training. A comprehensive review of various evaluation methods can be found in [17].

Summary

This chapter presents the principle of basic generative adversarial nets from a perspective of a minmax two-player game. The generator tries to generate a fake example to fool the discriminator while the discriminator tries to distinguish the fake example from the real examples. At the end of the training process, the generator and the discriminator reach an equilibrium: the discriminator can hardly distinguish the fake examples from the real examples because the generator does a great job.

To train a GAN, it is essential to represent the loss functions for the generator and the discriminator, respectively. For the generator, the loss function measures the penalty for the discriminator mistakenly classifying a generated fake example as real example. The loss function of the discriminator is a measure of penalty for misclassifying both fake examples and real examples. The key to the implementation of GANs in our tutorials is how to apply binary cross entropy loss function `nn.BCELoss()`. During the training loop, the parameters of generator and discriminator are updated alternatively within an iteration.

Conditional GANs can generate images under a specific constraint (or condition). A tutorial is given to demonstrate a conditional GAN that generates an image for a particular digit. Finally, we briefly highlight the variations of GANs for different applications and the challenges of GANs.

References

[1] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Networks. In NIPS, 2014.

[2] Tero Karras, Timo Aila, Samuli Laine, Jaakko Lehtinen,Progressive Growing of GANs for Improved Quality, Stability, and Variation, ICLR 2018.

[3] Understanding Generative Adversarial Networks (GANs),Building, step by step, the reasoning that leads to GANs. Joseph Rocca https://towardsdatascience.com/understanding-generative-adversarial-networks-gans-cd6e4651a29

[4] Martin Arjovsky, Soumith Chintala, and L´eon Bottou1, Wasserstein GAN, 2017.

[5] https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

[6] Radford et. al. , Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, 2016.

[7] https://github.com/soumith/ganhacks

[8] https://github.com/znxlwm/pytorch-MNIST-CelebA-GAN-DCGAN

[9] Mehdi Mirza, Simon Osindero, Conditional Generative Adversarial Nets, 2014.

[10] Tero Karras, Timo Aila, Samuli Laine, Jaakko Lehtinen, Progressive Growing of GANs for Improved Quality, Stability, and Variation, 2017.

[11] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, Alexei A. Efros, Image-to-Image Translation with Conditional Adversarial Networks, 2016.

[12] Jun-Yan Zhu, Taesung Park, Phillip Isola Alexei A. Efros, Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks, 2017.

[13] Christian Ledig, Lucas Theis, Ferenc Huszar, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, Wenzhe Shi, Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network, 2017.

[14] Raymond A. Yeh, Chen Chen, Teck Yian Lim, Alexander G. Schwing, Mark Hasegawa-Johnson, Minh N. Do, Semantic Image Inpainting with Deep Generative Models, 2017.

[15] Han Zhang, Tao Xu, Hongsheng Li, Shaoting Zhang, Xiaogang Wang, Xiaolei Huang, Dimitris Metaxas, StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks, 2016.

[16] Shan Yang, Lei Xie, Xiao Chen, Xiaoyan Lou, Xuan Zhu, Dongyan Huang, Haizhou Li, STATISTICAL PARAMETRIC SPEECH SYNTHESIS USING GENERATIVE ADVERSARIAL NETWORKS UNDER A MULTI-TASK LEARNING FRAMEWORK, 2017.

[17] A. Borji, ‘Pros and cons of gan evaluation measures’, Comput. Vis. Image Underst., vol. 179, pp. 41–65, 2019.

Exercises