GAN

- Generative Adversarial Nets

Overview

- GAN intro
- Defining the neural networks in pytorch
- Computing a forward pass
- Training our GAN
- DCGAN

Some cool demos

2014

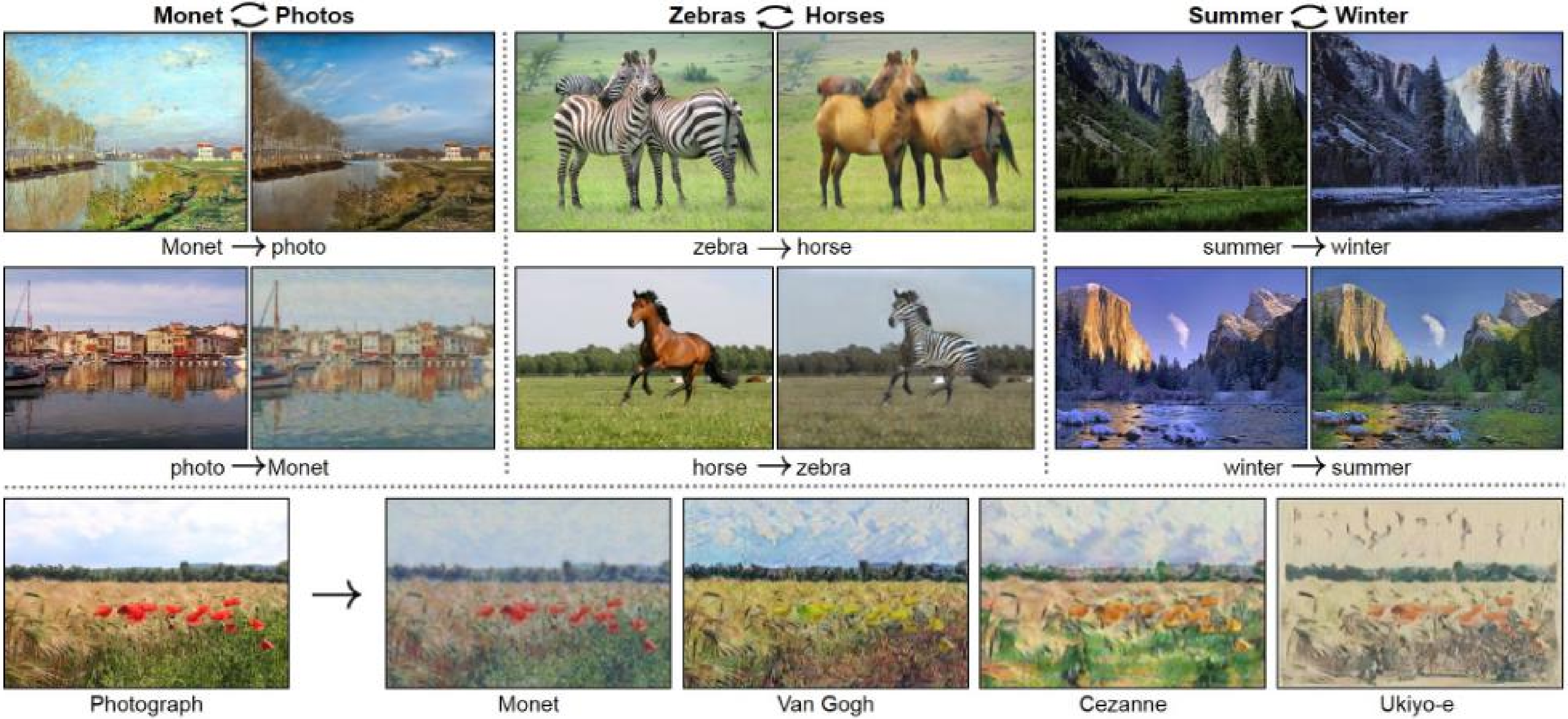
2015

2016

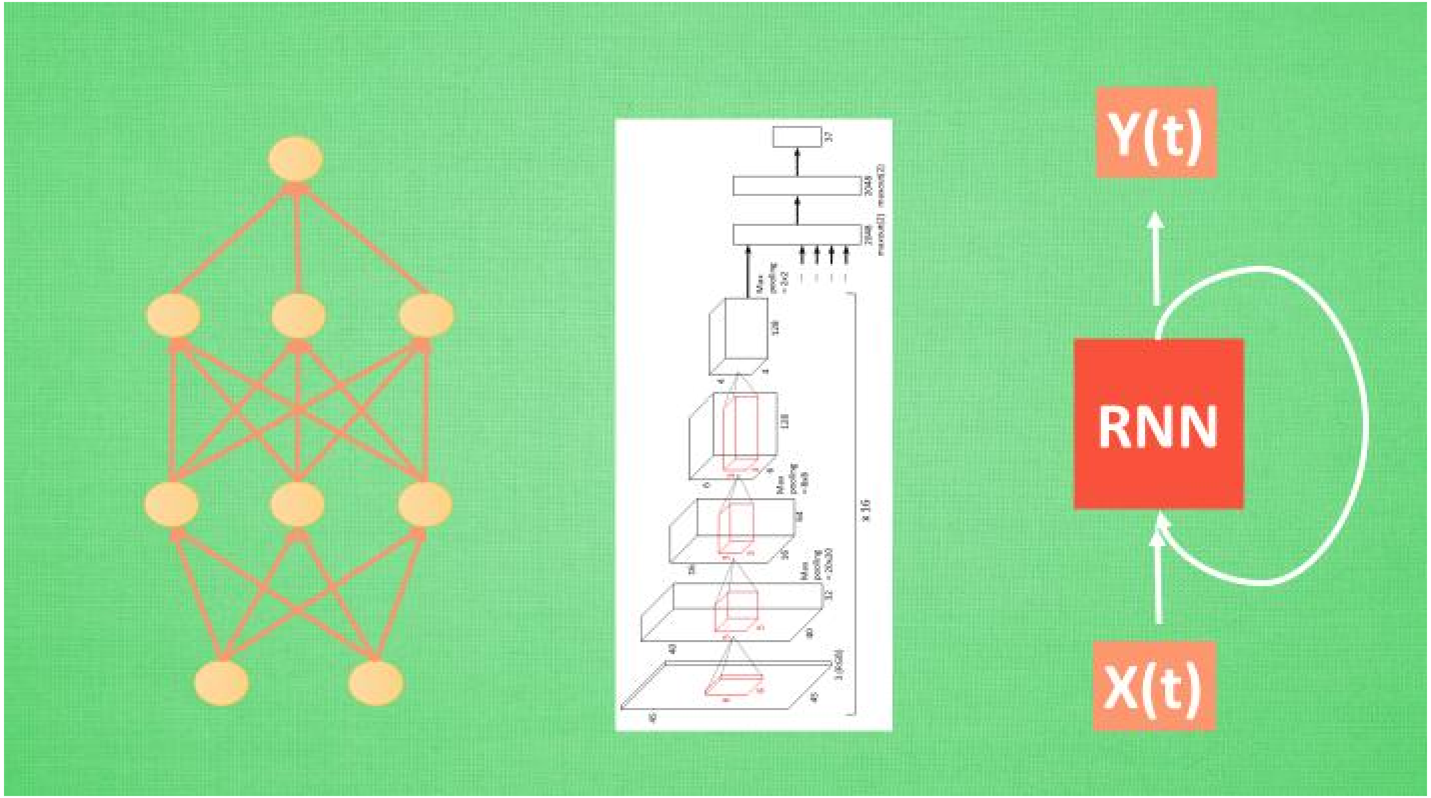
2017

2018

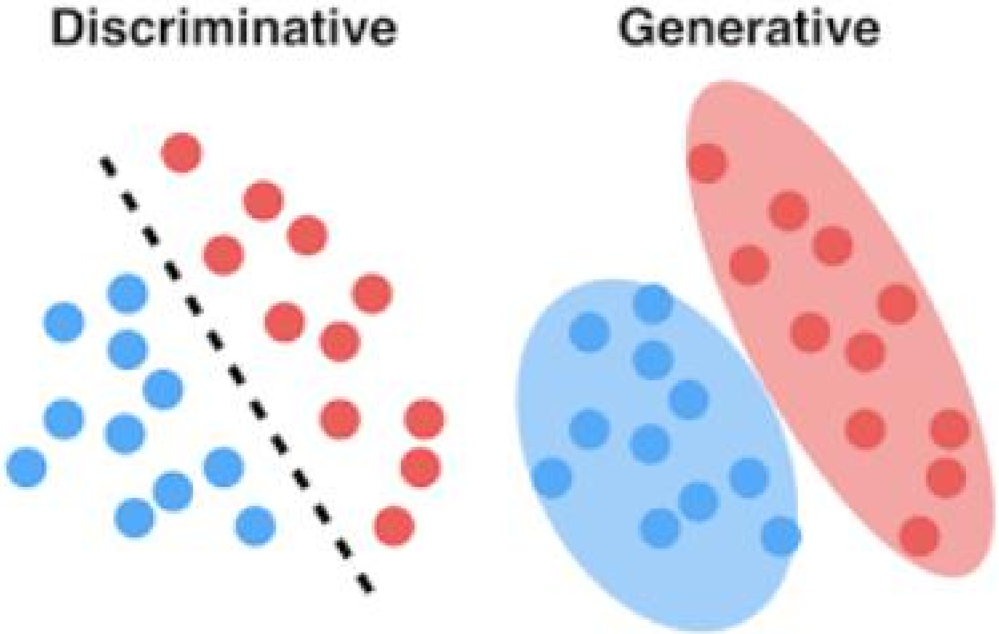
Some cool demos



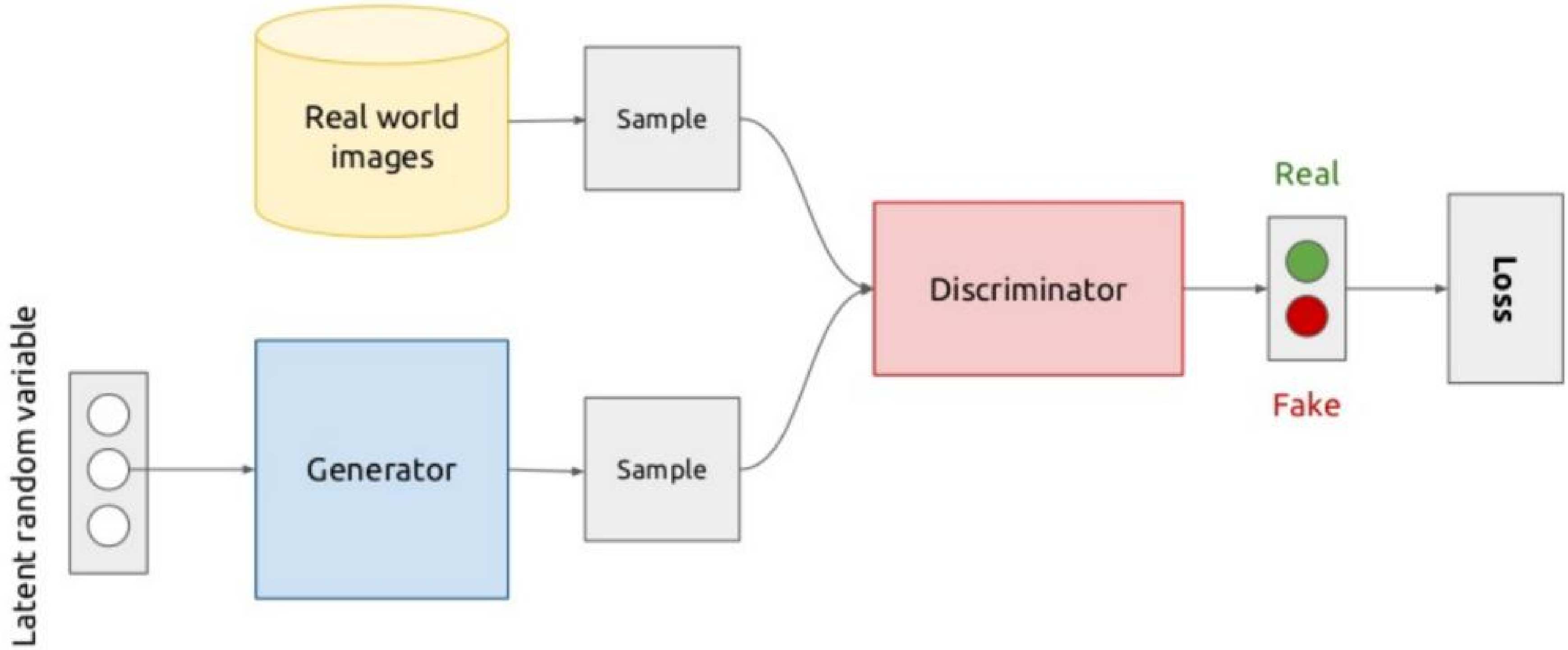
Discriminative vs Generative



Discriminative vs Generative



GAN first introduction



GAN first introduction

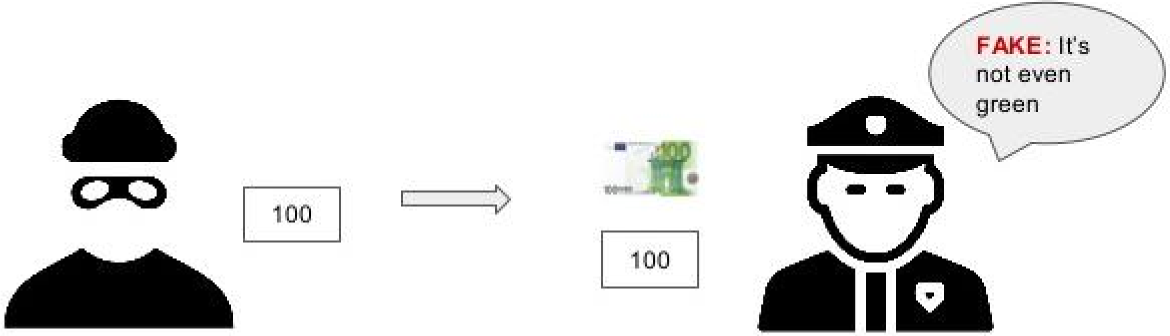
GANs are a class of unsupervised generative models which implicitly model the data density.

There are two "competing" neural networks:

- The Generator wants to learn to generate realistic images that are indistinguishable from the real data.
  - input: Gaussian noise random sample. output: a (higher dimensional) datapoint
- The Discriminator wants to tell the real & fake images apart.
  - input: datapoint/image, output: probability assigned to datapoint being real. Think binary classifier.

GAN first introduction

The typical analogy: the generator is like a counterfeiter trying to look like real, the discriminator is the police trying to tell counterfeits from the real work.



GAN first introduction

The key novelty of GANs is to pass the error signal (gradients) from the discriminator to the generator: the generator neural network uses the information from the competing discriminator neural network to know how to produce more realistic output.

Define the neural networks in pytorch

```
import sys
print(sys.version) # python 3.7
import torch
import torch.nn as nn
import torchvision.datasets
import torchvision.transforms as transforms
import torch.nn.functional as F
import torchvision.utils as vutils
print(torch.__version__) # 1.4.0
```

Define the neural networks in pytorch

```
%matplotlib inline
import matplotlib.pyplot as plt

def show_imgs(x, new_fig=True):
    grid = vutils.make_grid(x.detach().cpu(), nrow=8, normalize=True, pad_value=0.3)
    grid = grid.transpose(0,2).transpose(0,1) # channels as last dimension
    if new_fig:
        plt.figure()
    plt.imshow(grid.numpy())
```

Defining the neural networks

Let's define a small 2-layer fully connected neural network (so one hidden layer) for the discriminator D:

```
class Discriminator(torch.nn.Module):
    def __init__(self, inp_dim=784):
        super(Discriminator, self).__init__()
        self.fc1 = nn.Linear(inp_dim, 128)
        self.nonlin1 = nn.LeakyReLU(0.2)
        self.fc2 = nn.Linear(128, 1)
    def forward(self, x):
        x = x.view(x.size(0), 784) # flatten (bs x 1 x 28 x 28) -> (bs x 784)
        h = self.nonlin1(self.fc1(x))
        out = self.fc2(h)
        out = torch.sigmoid(out)
        return out
```

Defining the neural networks

And a small 2-layer neural network for the generator G. G takes a 100-dimensional noise vector and generates an output of the size matching the data.

```
class Generator(nn.Module):
    def __init__(self, z_dim=100):
        super(Generator, self).__init__()
        self.fc1 = nn.Linear(z_dim, 128)
        self.nonlin1 = nn.LeakyReLU(0.2)
        self.fc2 = nn.Linear(128, 784)
    def forward(self, x):
        h = self.nonlin1(self.fc1(x))
        out = self.fc2(h)
        out = torch.tanh(out) # range [-1, 1]
        out = out.view(out.size(0), 1, 28, 28)# convert to image
        return out
```

Defining the neural networks

```
# instantiate a Generator and Discriminator according to their class definition.
D = Discriminator()
print(D)
G = Generator()
print(G)
```

Note that the dimensions of D input and G output were defined for MNIST data.

Testing the neural networks (forward pass)

```
# A small batch of 3 samples, all zeros.
samples = torch.randn(5, 1, 28, 28) # batch size x channels x width x height
# This is how to do a forward pass (calls the .forward() function under the hood)
D(samples)
```

Testing the neural networks (forward pass)

Things to try:

What happens if you change the number of samples in a batch?
What happens if you change the width/height of the input?
What are the weights of the discriminator? You can get an iterator over them with .parameters() and .named_parameters()

```
for name, p in D.named_parameters():
    print(name, p.shape)
```

Testing the neural networks (forward pass)

We will think of the concatentation of all these discriminator weights in one big vector as θD .
Similarly we name the concatentation of all the generator weights in one big vector θG .

```
for name, p in G.named_parameters():
    print(name, p.shape)

# A small batch of 2 samples, random noise.
z = torch.randn(2, 100)
# This is how to do a forward pass (calls the .forward() function under the hood)
x_gen = G(z)
x_gen.shape

z = torch.randn(2, 100)
show_imgs(G(z))
```

Loading the data and computing forward pass

```
# let's download the Fashion MNIST data, if you do this locally and you downloaded before,
# you can change data paths to point to your existing files
# dataset = torchvision.datasets.MNIST(root='./MNISTdata', ...)
dataset = torchvision.datasets.FashionMNIST(root='./FashionMNIST/',
                    transform=transforms.Compose([transforms.ToTensor(),
                                                  transforms.Normalize((0.5,), (0.5,))]),
                    download=True)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=64, shuffle=True)
```

Loading the data and computing forward pass

Dataset and DataLoader are abstractions to help us iterate over the data in random order.

Let's look at a sample:

```
ix=149
x, _ = dataset[ix]
plt.matshow(x.squeeze().numpy(), cmap=plt.cm.gray)
plt.colorbar()
```

Loading the data and computing forward pass

Feed the image into the discriminator; the output will be the probability the (untrained) discriminator assigns to this sample being real.

```
# for one image:
Dscore = D(x)
Dscore

# How you can get a batch of images from the dataloader:
xbatch, _ = iter(dataloader).next() # 64 x 1 x 28 x 28: minibatch of 64 samples
xbatch.shape
D(xbatch) # 64x1 tensor: 64 predictions of probability of input being real.
D(xbatch).shape

show_imgs(xbatch)
```

The min-max game

We introduced and defined the generator G, the discriminator D, and the dataloader which will give us minibatches of real data.

The Generator and Discriminator have competing objectives, they are "adversaries".
The Discriminator wants to assign high probability to real images and low probability to generated (fake) images
The Generator wants its generated images to look real, so wants to modify its outputs to get high scores from the Discriminator
We will optimize both alternatingly, with SGD steps (as before): optimize θ_D the weights of $D(x,\theta_D)$, and θ_G the weights of $G(z,\theta_G)$.
Final goal of the whole min-max game is for the Generator to match the data distribution: $p_G(x) \approx p_{data}(x)$.

The min-max game

Now what are the objective functions for each of them? As mentioned in the introduction, the objective for the discriminator is to classify the real images as real, so $D(x)=1$, and the fake images as fake, so $D(G(z))=0$. This is a typical binary classification problem which calls for the binary cross-entropy (BCE) loss, which encourages exactly this solution.

For G we just try to minimize the same loss that D maximizes. See how G appears inside D? This shows how the output of the generator G is passed into the Discriminator to compute the loss.

The min-max game

This is the optimization problem:

$$\min_G \max_D V(D, G)$$

$$V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z))]$$

The min-max game

$$\min_G \max_D V(D, G)$$

$$V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z))]$$

We will do a single SGD step alternatingly to maximize D, then minimize G. In fact for G we use a modified (non-saturing) loss $-\log D(G(z))$. Different modifications of the loss and the relation to the distance between distributions p_{data} and p_G became a topic of research over the last years.

The min-max game

```
# Remember we have defined the discriminator and generator as:
D = Discriminator()
print(D)
G = Generator()
print(G)
# Now let's set up the optimizers
optimizerD = torch.optim.SGD(D.parameters(), lr=0.01)
optimizerG = torch.optim.SGD(G.parameters(), lr=0.01)

# and the BCE criterion which computes the loss above:
criterion = nn.BCELoss()
```

```
# STEP 1: Discriminator optimization step
x_real, _ = iter(dataloader).next()
lab_real = torch.ones(64, 1)
lab_fake = torch.zeros(64, 1)
optimizerD.zero_grad() # reset accumulated gradients from previous iteration

D_x = D(x_real)
lossD_real = criterion(D_x, lab_real)

z = torch.randn(64, 100) # random noise, 64 samples, z_dim=100
x_gen = G(z).detach()
D_G_z = D(x_gen)
lossD_fake = criterion(D_G_z, lab_fake)

lossD = lossD_real + lossD_fake
lossD.backward()
optimizerD.step()
```

The min-max game

Some things to think about / try out / investigate:

what are the mean probabilities for real and fake? print them and see how they change when executing the cell above a couple of times. Does this correspond to your expectation?
can you confirm how the use of the criterion maps to the objective stated above?
when calling backward, the derivative of the loss wrt what gets computed?
what does .detach() do? Are the Generator parameters' gradients computed?

The min-max game

```
# STEP 2: Generator optimization step
# note how only one of the terms involves the Generator so this is the only one that
# matters for G.  reset accumulated gradients from previous iteration
optimizerG.zero_grad()

z = torch.randn(64, 100) # random noise, 64 samples, z_dim=100
D_G_z = D(G(z))
lossG = criterion(D_G_z, lab_real) # -log D(G(z))

lossG.backward()
optimizerG.step()

print(D_G_z.mean().item())
```

The min-max game

Again run this cell a couple of times. See how the generator increases its Discriminator score?

Some more things to ponder:

Do the Generator parameters now receive gradients? Why (compared to previous loop)?

From the definition of BCE loss confirm that this comes down to $-\log D(G(z))$

the full training loop

```
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
print('Device: ', device)
# Re-initialize D, G:
D = Discriminator().to(device)
G = Generator().to(device)
# Now let's set up the optimizers (Adam, better than SGD for this)
optimizerD = torch.optim.SGD(D.parameters(), lr=0.03)
optimizerG = torch.optim.SGD(G.parameters(), lr=0.03)
# optimizerD = torch.optim.Adam(D.parameters(), lr=0.0002)
# optimizerG = torch.optim.Adam(G.parameters(), lr=0.0002)
lab_real = torch.ones(64, 1, device=device)
lab_fake = torch.zeros(64, 1, device=device)
```

the full training loop

```
# for logging:
collect_x_gen = []
fixed_noise = torch.randn(64, 100, device=device)
fig = plt.figure() # keep updating this one
plt.ion()
```

the full training loop

```
for epoch in range(3): # 10 epochs
    for i, data in enumerate(dataloader, 0):
        # STEP 1: Discriminator optimization step
        x_real, _ = iter(dataloader).next()
        x_real = x_real.to(device)
        # reset accumulated gradients from previous iteration
        optimizerD.zero_grad()
```

the full training loop

```
D_x = D(x_real)
lossD_real = criterion(D_x, lab_real)

z = torch.randn(64, 100, device=device) # random noise, 64 samples, z_dim=100
x_gen = G(z).detach()
D_G_z = D(x_gen)
lossD_fake = criterion(D_G_z, lab_fake)

lossD = lossD_real + lossD_fake
lossD.backward()
optimizerD.step()
```

the full training loop

```
# STEP 2: Generator optimization step
# reset accumulated gradients from previous iteration
optimizerG.zero_grad()

z = torch.randn(64, 100, device=device) # random noise, 64 samples, z_dim=100
x_gen = G(z)
D_G_z = D(x_gen)
lossG = criterion(D_G_z, lab_real) # -log D(G(z))

lossG.backward()
optimizerG.step()
```

the full training loop

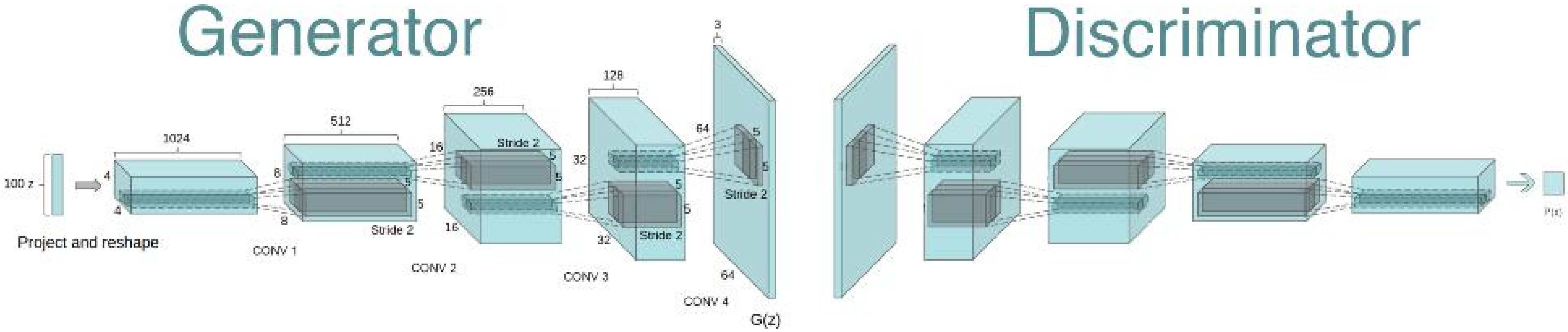
```
        if i % 100 == 0:
            x_gen = G(fixed_noise)
            show_imgs(x_gen, new_fig=False)
            fig.canvas.draw()
            print('e{}.i{}/{} last mb D(x)={:.4f} D(G(z))={:.4f}'.format(
                epoch, i, len(dataloader), D_x.mean().item(), D_G_z.mean().item()))
    # End of epoch
    x_gen = G(fixed_noise)
    collect_x_gen.append(x_gen.detach().clone())

for x_gen in collect_x_gen:
    show_imgs(x_gen)
```

Deep convolutional GAN

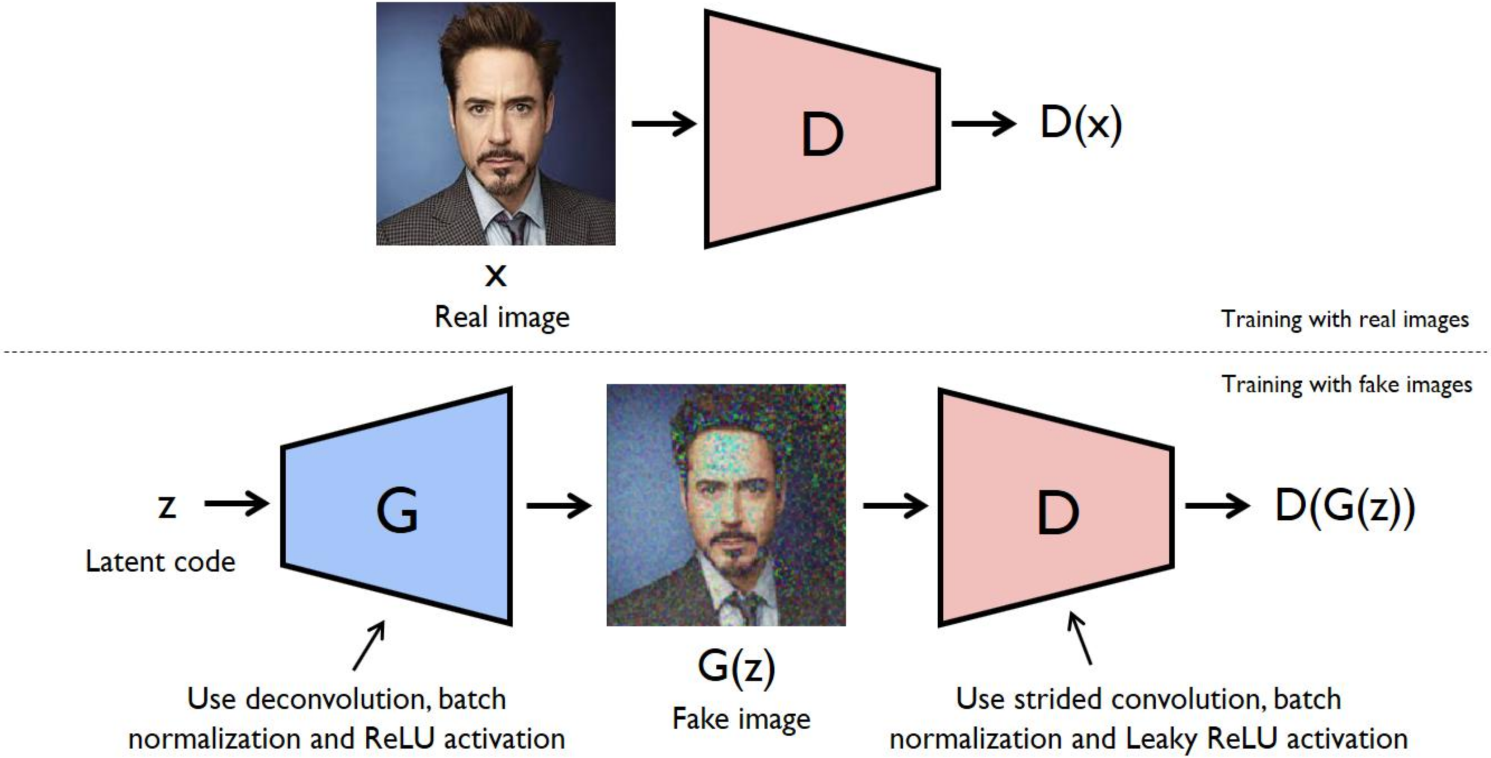
The DCGAN is one of the early models that demonstrated how to build a GAN model that learns by itself and generates meaningful images.

DCGAN



Generator
Discriminator
100 z
Project and reshape
1024
4
4
512
8
8
Stride 2
5
5
CONV 1
256
16
16
Stride 2
CONV 2
128
32
32
Stride 2
CONV 3
3
64
64
Stride 2
CONV 4
G(z)

DCGAN



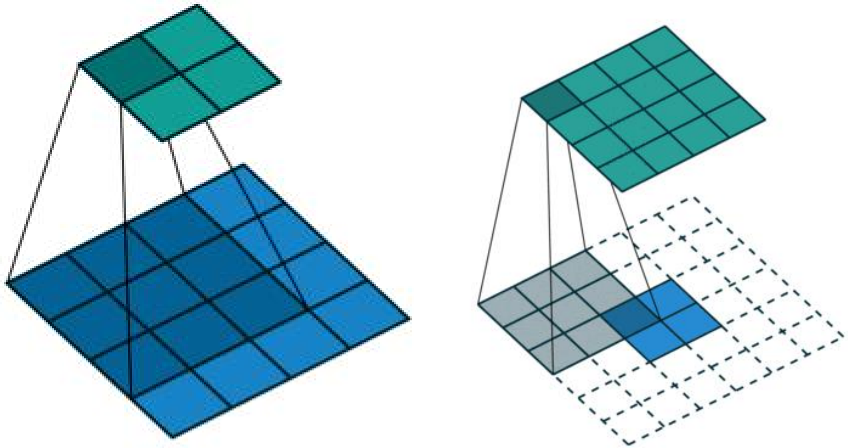
D(x)
D
x
Real image
Training with real images
Training with fake images
z
Latent code
G
D
D(G(z))
G(z)
Fake image
Use deconvolution, batch normalization and ReLU activation
Use strided convolution, batch normalization and Leaky ReLU activation

generator network

The generator network takes a random vector of fixed dimension as input, and applies a set of transposed convolutions, batch normalization, and ReLu activation to it, and generatesan image of the required size.

Transposed convolutions

Transposed convolutions are also called fractionally strided convolutions. They work in the opposite way to how convolution works. Intuitively, they try to calculate how the input vector can be mapped to higher dimensions.

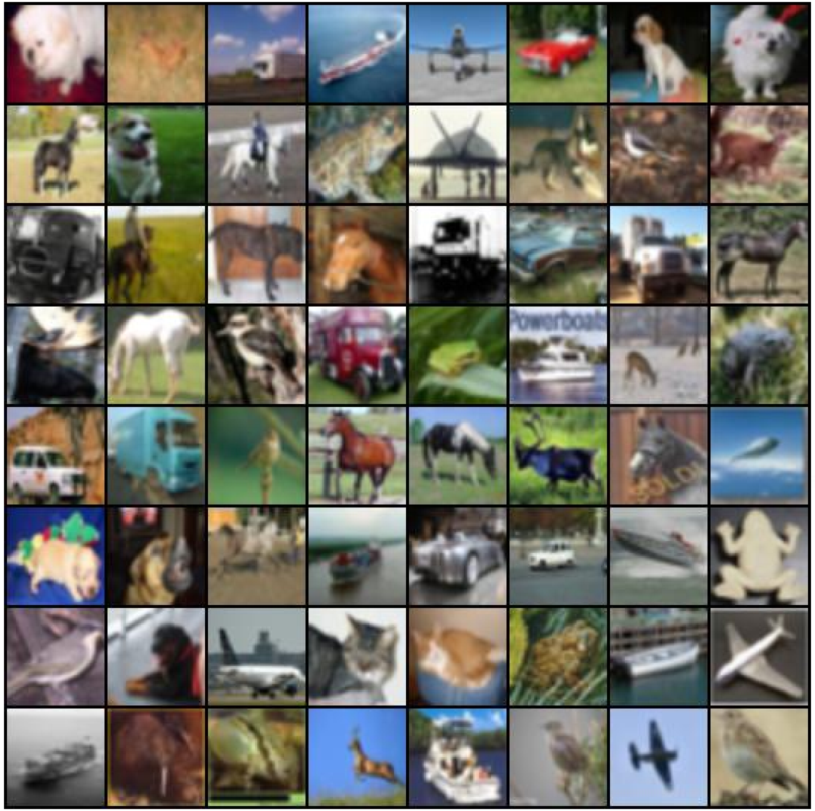


```
self.main = nn.Sequential(
    # input is Z, going into a convolution
    nn.ConvTranspose2d(     nz, ngf * 8, 4, 1, 0, bias=False),
    nn.BatchNorm2d(ngf * 8),
    nn.ReLU(True),
    # state size. (ngf*8) x 4 x 4
    nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
    nn.BatchNorm2d(ngf * 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
)
```

discriminator network

discriminator network uses leaky ReLU is an attempt to fix the dying ReLU problem. Instead of the function returning zero when the input is negative, leaky ReLU will output a very small number like 0.001. In the paper, it is shown that using leaky ReLU improves the efficiency of the discriminator.

```
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()
)
```


Powerboats