Deep learning

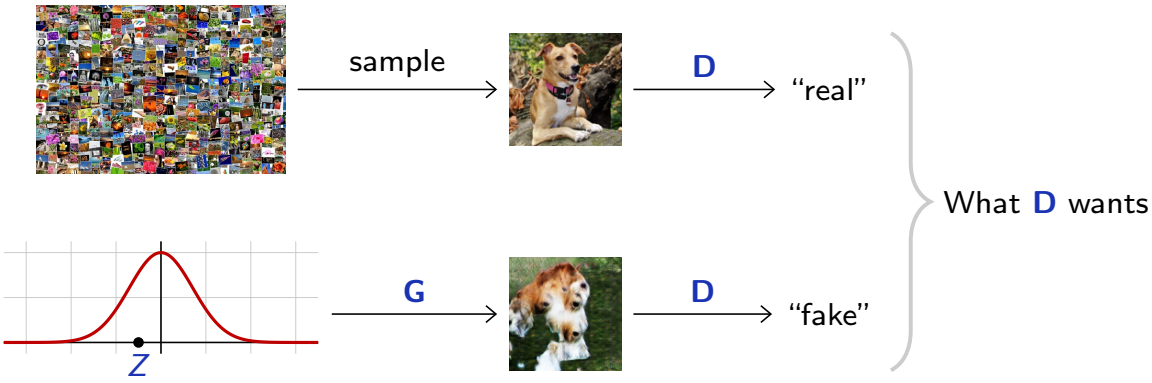
11.1. Generative Adversarial Networks

François Fleuret

https://fleuret.org/dlc/

A popular approach to learn high-dimension densities are the **Generative Adversarial Networks** proposed by Goodfellow et al. (2014), where two networks are trained jointly:

- A **discriminator** **D** to classify samples as "real" or "fake",
- a **generator** **G** to map a [simple] fixed distribution to samples that fool **D**.



The approach is **adversarial** since the two networks have antagonistic objectives.

Notes

The role of the discriminator is to detect if a sample is from the real world or was generated.
The role of the generator is to produce realistic samples: given some random noise following a fixed and simple distribution, it should produce samples which are realistic in the sense that they fool the discriminator.
So these two models have an opposite objective: the discriminator is optimized to minimize a standard classification loss, and the generator is optimized to maximize that loss.
A key point is that the generator maximize that loss *through the discriminator*. Hence the backward pass will propagate the gradient of the loss through the discriminator to the generator, and the generator will be constantly updated during training to remove any statistical structure that was picked up by the discriminator as specific to the synthetic samples.

Let $\mathcal{X}$ be the signal space, and D the latent space dimension.

- The **generator**

$$\mathbf{G} : \mathbb{R}^D \to \mathcal{X}$$

  is trained so that [ideally] if it gets a random normal-distributed Z as input, it produces a sample following the data distribution as output.

- The **discriminator**

$$\mathbf{D} : \mathcal{X} \to [0, 1]$$

  is trained so that if it gets a sample as input, it predicts if it comes from **G** or from the real data.

Notes

In what follows, the generator takes as input a random sample following a normal distribution, but another choice is possible.
The discriminator gets as input a sample in $\mathcal{X}$ and computes an estimate of the posterior probability for it to be "real."

Given a set of "real points"

$$x_n \sim \mu, \ \ n = 1, \ldots, N,$$

and if $\mathbf{G}$ is fixed, we can train $\mathbf{D}$ by generating

$$z_n \sim \mathcal{N}(0, I), \ \ n = 1, \ldots, N,$$

building a two-class data-set

$$\mathscr{D} = \Big\{ \underbrace{(x_1, 1), \ldots, (x_N, 1)}_{\text{real samples } \sim \mu}, \underbrace{(\mathbf{G}(z_1), 0), \ldots, (\mathbf{G}(z_N), 0)}_{\text{fake samples } \sim \mu_{\mathbf{G}}} \Big\},$$

where μ is the true data distribution, and $\mu_{\mathbf{G}}$ is the distribution of $\mathbf{G}(Z)$ with $Z \sim \mathcal{N}(0, I)$, and minimizing the binary cross-entropy

$$\begin{aligned} \mathscr{L}(\mathbf{D}) &= -\frac{1}{2N} \left(\sum_{n=1}^{N} \log \mathbf{D}(x_n) + \sum_{n=1}^{N} \log(1 - \mathbf{D}(\mathbf{G}(z_n))) \right) \\ &= -\frac{1}{2} \left(\hat{\mathbb{E}}_{X \sim \mu} \Big[\log \mathbf{D}(X) \Big] + \hat{\mathbb{E}}_{X \sim \mu_{\mathbf{G}}} \Big[\log(1 - \mathbf{D}(X)) \Big] \right). \end{aligned}$$

The situation is slightly more complicated since we also want to optimize **G** to *maximize* **D**'s loss.

Goodfellow et al. (2014) provide an analysis of the resulting equilibrium of that strategy.

Let's define the loss of $\mathbf{G}$

$$\mathscr{L}_{\mathbf{G}}(\mathbf{D},\mathbf{G}) = \mathbb{E}_{X\sim\mu}\Big[\log \mathbf{D}(X)\Big] + \mathbb{E}_{X\sim\mu_{\mathbf{G}}}\Big[\log(1-\mathbf{D}(X))\Big]$$

which is high if $\mathbf{D}$ is doing a good job (low cross entropy), and low if $\mathbf{G}$ fools $\mathbf{D}$.

Our ultimate goal is a $\mathbf{G}^*$ that fools *any* $\mathbf{D}$, so

$$\mathbf{G}^* = \operatorname*{argmin}_{\mathbf{G}} \max_{\mathbf{D}} \mathscr{L}_{\mathbf{G}}(\mathbf{D},\mathbf{G}).$$

If we define the optimal discriminator for a given generator

$$\mathbf{D}^*_{\mathbf{G}} = \underset{\mathbf{D}}{\text{argmax}}\, \mathscr{L}_{\mathbf{G}}(\mathbf{D}, \mathbf{G}),$$

our objective becomes

$$\mathbf{G}^* = \underset{\mathbf{G}}{\text{argmin}}\, \mathscr{L}_{\mathbf{G}}(\mathbf{D}^*_{\mathbf{G}}, \mathbf{G}),$$

that is:

Find a $\mathbf{G}$ whose loss against its best adversary $\mathbf{D}^*_{\mathbf{G}}$ is low.

We have

$$\begin{aligned}\mathscr{L}_{\mathbf{G}}(\mathbf{D},\mathbf{G}) &= \mathbb{E}_{X\sim\mu}\Big[\log \mathbf{D}(X)\Big] + \mathbb{E}_{X\sim\mu_{\mathbf{G}}}\Big[\log(1-\mathbf{D}(X))\Big]\\ &= \int_x \mu(x)\log \mathbf{D}(x) + \mu_{\mathbf{G}}(x)\log(1-\mathbf{D}(x))dx.\end{aligned}$$

Since

$$\operatorname*{argmax}_d \ \mu(x)\log d + \mu_{\mathbf{G}}(x)\log(1-d) \quad = \quad \frac{\mu(x)}{\mu(x)+\mu_{\mathbf{G}}(x)},$$

and

$$\mathbf{D}^*_{\mathbf{G}} = \operatorname*{argmax}_{\mathbf{D}} \mathscr{L}_{\mathbf{G}}(\mathbf{D},\mathbf{G}),$$

if there is no regularization on $\mathbf{D}$, we get

$$\forall x,\ \mathbf{D}^*_{\mathbf{G}}(x) = \frac{\mu(x)}{\mu(x)+\mu_{\mathbf{G}}(x)}.$$

So, since

$$\forall x,\ \mathbf{D}^*_{\mathbf{G}}(x) = \frac{\mu(x)}{\mu(x) + \mu_{\mathbf{G}}(x)}.$$

we get

$$\begin{aligned}
\mathscr{L}_{\mathbf{G}}(\mathbf{D}^*_{\mathbf{G}}, \mathbf{G}) &= \mathbb{E}_{X\sim\mu}\Big[\log \mathbf{D}^*_{\mathbf{G}}(X)\Big] + \mathbb{E}_{X\sim\mu_{\mathbf{G}}}\Big[\log(1 - \mathbf{D}^*_{\mathbf{G}}(X))\Big] \\
&= \mathbb{E}_{X\sim\mu}\left[\log \frac{\mu(X)}{\mu(X) + \mu_{\mathbf{G}}(X)}\right] + \mathbb{E}_{X\sim\mu_{\mathbf{G}}}\left[\log \frac{\mu_{\mathbf{G}}(X)}{\mu(X) + \mu_{\mathbf{G}}(X)}\right] \\
&= \mathbb{D}_{\mathrm{KL}}\left(\mu \,\middle\|\, \frac{\mu + \mu_{\mathbf{G}}}{2}\right) + \mathbb{D}_{\mathrm{KL}}\left(\mu_{\mathbf{G}} \,\middle\|\, \frac{\mu + \mu_{\mathbf{G}}}{2}\right) - \log 4 \\
&= 2\,\mathbb{D}_{\mathrm{JS}}\left(\mu, \mu_{\mathbf{G}}\right) - \log 4
\end{aligned}$$

where $\mathbb{D}_{\mathrm{JS}}$ is the Jensen-Shannon Divergence, a standard similarity measure between distributions.

Notes

We show here that a low value of $\mathscr{L}_{\mathbf{G}}(\mathbf{D}^*_{\mathbf{G}}, \mathbf{G})$, which is the quantity that the generator aims at minimizing, corresponds to a low value of the Jensen-Shannon divergence between the true distribution and the one of the generated samples. Hence the generator is optimized to minimize this divergence, so to make the two distributions similar.

Intuitively, if a generator is good against *all* discriminators, it cannot have at any points x a probability density different from the true data density. If an x is more likely under μ than under $\mu_{\mathbf{G}}$, then a discriminator could do slightly better than random in predicting x to be real, and vice-versa.

To recap: if there is no capacity limitation for $\mathbf{D}$, and if we define

$$\mathscr{L}_{\mathbf{G}}(\mathbf{D},\mathbf{G}) = \mathbb{E}_{X\sim\mu}\Big[\log \mathbf{D}(X)\Big] + \mathbb{E}_{X\sim\mu_{\mathbf{G}}}\Big[\log(1-\mathbf{D}(X))\Big],$$

computing

$$\mathbf{G}^* = \operatorname*{argmin}_{\mathbf{G}} \max_{\mathbf{D}} \mathscr{L}_{\mathbf{G}}(\mathbf{D},\mathbf{G})$$

amounts to compute

$$\mathbf{G}^* = \operatorname*{argmin}_{\mathbf{G}} \mathbb{D}_{\mathrm{JS}}(\mu,\mu_{\mathbf{G}}),$$

where $\mathbb{D}_{\mathrm{JS}}$ is a reasonable similarity measure between distributions.

Although this derivation provides a nice formal framework, in practice $\mathbf{D}$ is not "fully" optimized to [come close to] $\mathbf{D}^*_{\mathbf{G}}$ when optimizing $\mathbf{G}$.

In the toy example that follows, we alternate gradient steps to improve $\mathbf{G}$ and $\mathbf{D}$.

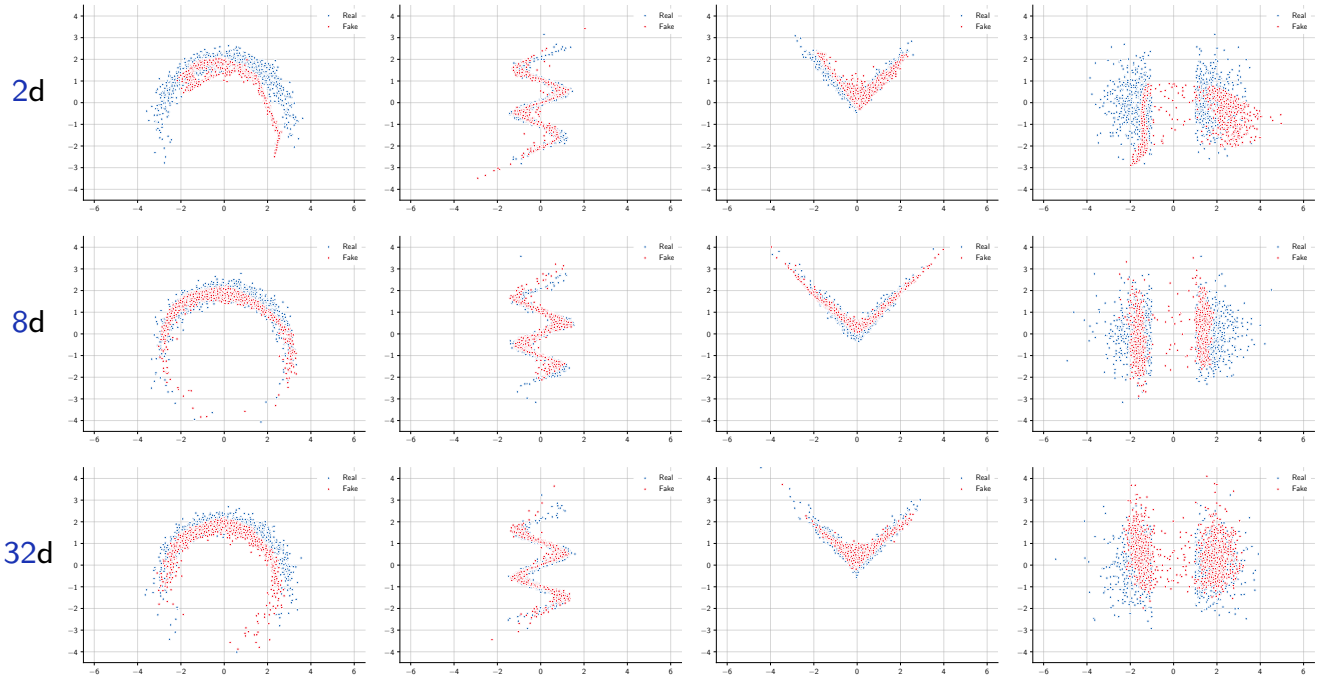
For our example, we take $D = 8$, and $\mathscr{X} = \mathbb{R}^2$.

```
z_dim = 8
nb_hidden = 100

model_G = nn.Sequential(nn.Linear(z_dim, nb_hidden),
                        nn.ReLU(),
                        nn.Linear(nb_hidden, 2))

model_D = nn.Sequential(nn.Linear(2, nb_hidden),
                        nn.ReLU(),
                        nn.Linear(nb_hidden, 1),
                        nn.Sigmoid())
```

```
batch_size, lr = 10, 1e-3

optimizer_G = optim.Adam(model_G.parameters(), lr = lr)
optimizer_D = optim.Adam(model_D.parameters(), lr = lr)

for e in range(nb_epochs):

    for t, real_batch in enumerate(real_samples.split(batch_size)):
        z = real_batch.new(real_batch.size(0), z_dim).normal_()
        fake_batch = model_G(z)

        D_scores_on_real = model_D(real_batch)
        D_scores_on_fake = model_D(fake_batch)

        if t%2 == 0:
            loss = (1 - D_scores_on_fake).log().mean()
            optimizer_G.zero_grad()
            loss.backward()
            optimizer_G.step()
        else:
            loss = - (1 - D_scores_on_fake).log().mean() \
                   - D_scores_on_real.log().mean()
            optimizer_D.zero_grad()
            loss.backward()
            optimizer_D.step()
```


Notes

For even batches, the loss is only computed on the fake samples to optimize the generator, and `D_scores_on_real` is not used. For odd batches, the loss is computed on all samples to optimize the discriminator.



Notes

Each row correspond to a dimension of the input to the generator, and each column to a synthetic 2d distribution. The blues points are sampled according to the real distribution, and the red points are generated with the trained generator. A synthetic point is generated by sampling from a Gaussian distribution, and passing it to the generator to produce a point in $\mathbb{R}^2$.
We can see that when the dimension of the space the generator operates on is higher, the resulting distribution is closer to the real one.
The last example (far right) is more difficult because there is a discontinuity in the target distribution.

In more realistic settings, the fake samples may be initially so "unrealistic" that the response of **D** saturates. That causes the loss for **G**

$$\hat{\mathbb{E}}_{X \sim \mu_{\mathbf{G}}} \Big[\log(1 - \mathbf{D}(X)) \Big]$$

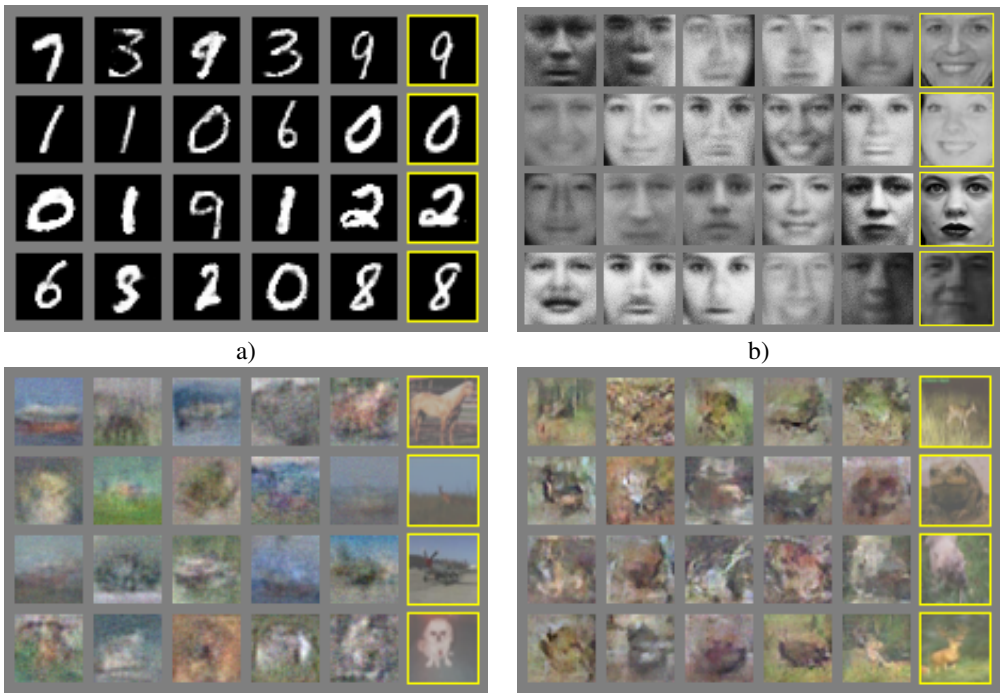
to be far in the exponential tail of **D**'s sigmoid, and have zero gradient since $\log(1 + \epsilon) \simeq \epsilon$ does not correct it in any way.

Goodfellow et al. suggest to replace this term with a **non-saturating** cost

$$-\hat{\mathbb{E}}_{X \sim \mu_{\mathbf{G}}} \Big[\log(\mathbf{D}(X)) \Big]$$

so that the log fixes **D**'s exponential behavior. The resulting optimization problem has the same optima as the original one.

The loss for **D** remains unchanged.



a) b) c) d)

Figure 2: Visualization of samples from the model. Rightmost column shows the nearest training example of the neighboring sample, in order to demonstrate that the model has not memorized the training set. Samples are fair random draws, not cherry-picked. Unlike most other visualizations of deep generative models, these images show actual samples from the model distributions, not conditional means given samples of hidden units. Moreover, these samples are uncorrelated because the sampling process does not depend on Markov chain mixing. a) MNIST b) TFD c) CIFAR-10 (fully connected model) d) CIFAR-10 (convolutional discriminator and "deconvolutional" generator)

(Goodfellow et al., 2014)

Deep Convolutional GAN

"We also encountered difficulties attempting to scale GANs using CNN architectures commonly used in the supervised literature. However, after extensive model exploration we identified a family of architectures that resulted in stable training across a range of datasets and allowed for training higher resolution and deeper generative models."

(Radford et al., 2015)

Radford et al. converged to the following rules:

- Replace pooling layers with strided convolutions in **D** and strided transposed convolutions in **G**,
- use batchnorm in both **D** and **G**,
- remove fully connected hidden layers,
- use ReLU in **G** except for the output, which uses Tanh,
- use LeakyReLU activation in **D** for all layers.

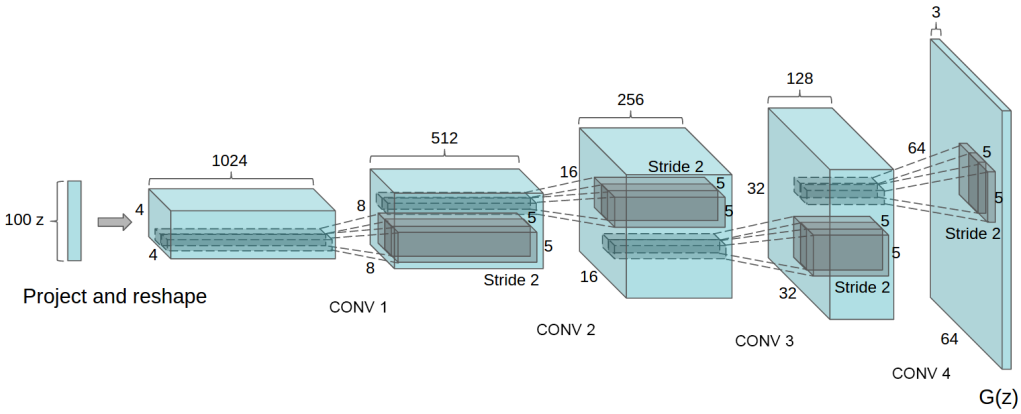


Figure 1: DCGAN generator used for LSUN scene modeling. A 100 dimensional uniform distribution Z is projected to a small spatial extent convolutional representation with many feature maps. A series of four fractionally-strided convolutions (in some recent papers, these are wrongly called deconvolutions) then convert this high level representation into a 64×64 pixel image. Notably, no fully connected or pooling layers are used.

(Radford et al., 2015)

We can have a look at the reference implementation provided in

https://github.com/pytorch/examples.git

```
# default nz = 100, ngf = 64

class Generator(nn.Module):
    def __init__(self, ngpu):
        super().__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
        )
```

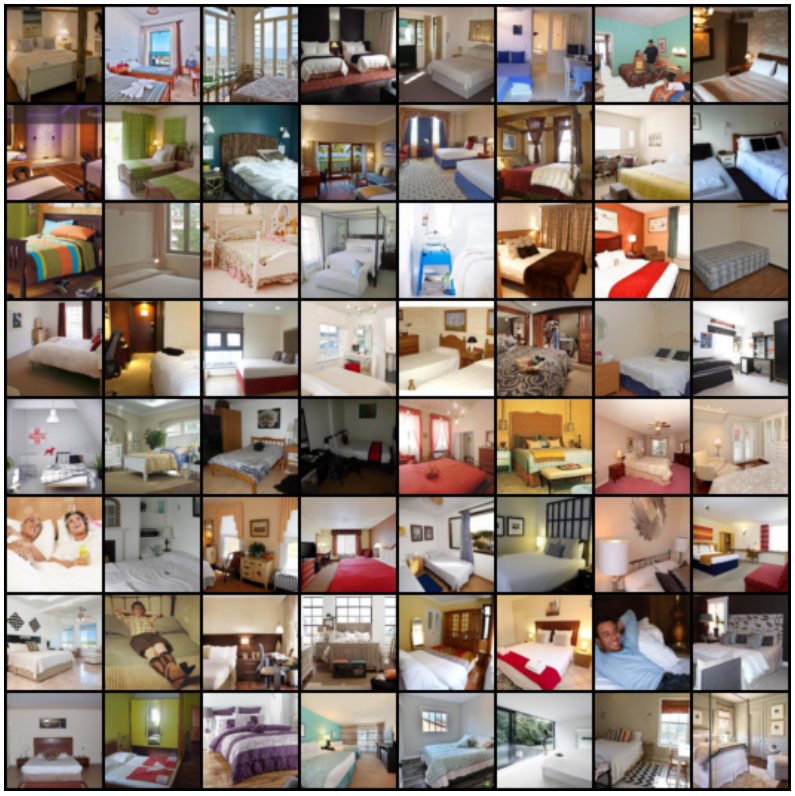
```
# default nz = 100, ndf = 64

class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super().__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()
        )
```

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

criterion = nn.BCELoss()

fixed_noise = torch.randn(opt.batchSize, nz, 1, 1, device=device)
real_label = 1
fake_label = 0

# setup optimizer
optimizerD = optim.Adam(netD.parameters(), lr=opt.lr, betas=(opt.beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=opt.lr, betas=(opt.beta1, 0.999))
```

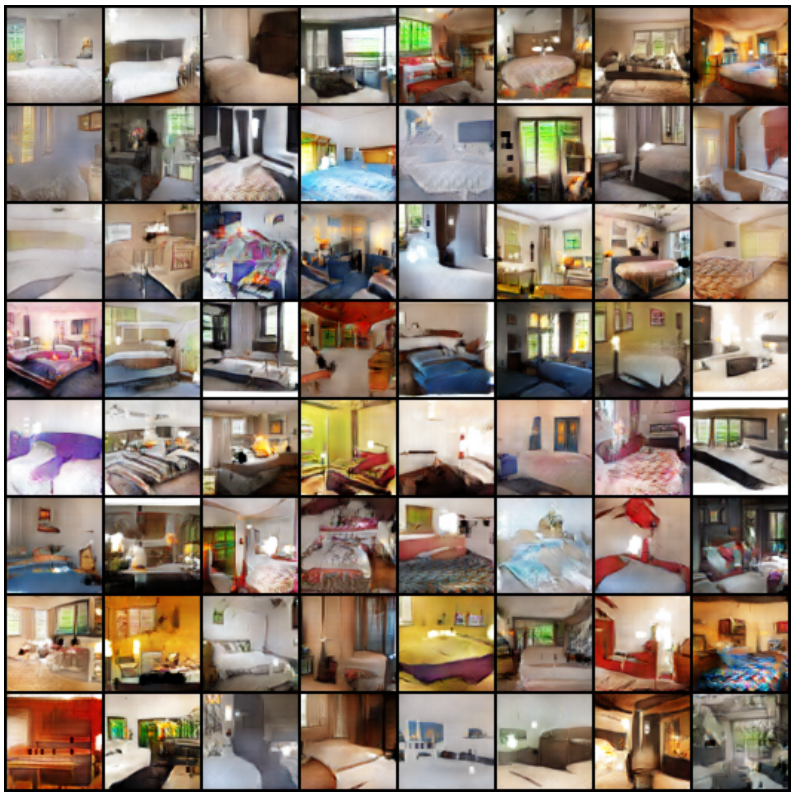
```
############################
# (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
###########################
# train with real
netD.zero_grad()
real_cpu = data[0].to(device)
batch_size = real_cpu.size(0)
label = torch.full((batch_size,), real_label, device=device)

output = netD(real_cpu)
errD_real = criterion(output, label)
errD_real.backward()
D_x = output.mean().item()

# train with fake
noise = torch.randn(batch_size, nz, 1, 1, device=device)
fake = netG(noise)
label.fill_(fake_label)
output = netD(fake.detach())
errD_fake = criterion(output, label)
errD_fake.backward()
D_G_z1 = output.mean().item()
errD = errD_real + errD_fake
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
output = netD(fake)
errG = criterion(output, label)
errG.backward()
D_G_z2 = output.mean().item()
optimizerG.step()
```

Note that this update implements the $-\log(D(G(z)))$ trick.



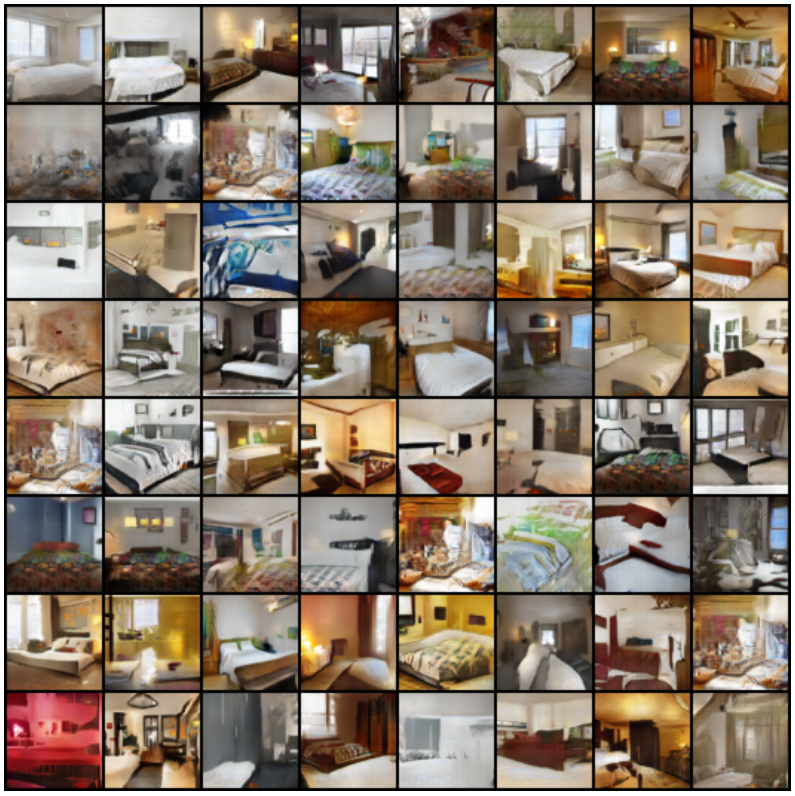
Real images from LSUN's "bedroom" class.



Fake images after 1 epoch (3M images)

Notes

The images shown here were generated with the reference code and with the Z kept unchanged across epochs: at each epoch, for visualization purposes, the generator takes as input the very same random values.
What is interesting to notice is that the overall semantic of the images is kept: location of windows, main colors, etc.



Fake images after 20 epochs

Training a standard GAN often results in two pathological behaviors:

- Oscillations without convergence. Contrary to standard loss minimization, we have no guarantee here that it will actually decrease.
- The infamous "mode collapse", when **G** models very well a small sub-population, concentrating on a few modes.

Additionally, performance is hard to assess. Two standard measures are the Inception Score (Salimans et al., 2016) and the Fréchet Inception Distance (Heusel et al., 2017), but assessment is often a "beauty contest".

Notes

The Inception Score checks that when generated images are classified by an inception model (Szegedy et al., 2015) the estimated posterior distribution of classes is similar to the real class distribution, which in particular penalizes a missing class.
The Fréchet Inception Distance looks at the distributions of the features in one of the feature maps of the inception model, for the real and synthetic samples, and estimate their similarity under a Gaussian model.

(Brock et al., 2018)

(Brock et al., 2018)

Notes

To make sure that the generator did not learn by heart the training set, Brock et al. compare a generated sample (top left) with "its closest ones" in the training set.

(Karras et al., 2018)

References

A. Brock, J. Donahue, and K. Simonyan. **Large scale GAN training for high fidelity natural image synthesis**. CoRR, abs/1809.11096, 2018.

I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. **Generative adversarial networks**. CoRR, abs/1406.2661, 2014.

M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter. **GANs trained by a two time-scale update rule converge to a local nash equilibrium**. CoRR, abs/1706.08500, 2017.

T. Karras, S. Laine, and T. Aila. **A style-based generator architecture for generative adversarial networks**. CoRR, abs/1812.04948, 2018.

A. Radford, L. Metz, and S. Chintala. **Unsupervised representation learning with deep convolutional generative adversarial networks**. CoRR, abs/1511.06434, 2015.

T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, X. Chen, and X. Chen. **Improved techniques for training GANs**. In Neural Information Processing Systems (NIPS), pages 2234–2242, 2016.

C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. **Rethinking the inception architecture for computer vision**. CoRR, abs/1512.00567, 2015.