EE-559 – Deep learning

10. Generative Adversarial Networks

François Fleuret
https://fleuret.org/dlc/

[version of: May 17, 2018]

Adversarial generative models

A different approach to learn high-dimension generative models are the **Generative Adversarial Networks** proposed by Goodfellow et al. (2014).

The idea behind GANs is to train two networks jointly:

- A **generator G** to map a Z following a [simple] fixed distribution to the desired "real" distribution, and
- a **discriminator D** to classify data points as "real" or "fake" (*i.e.* from **G**).

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

A bit more formally, let $\mathscr{X}$ be the signal space and D the latent space dimension.

- The **generator**

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

  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**

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

  is trained so that if it gets a sample as input, it predicts if it is genuine.

If $\mathbf{G}$ is fixed, to train $\mathbf{D}$ given a set of "real points"

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

we can generate

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

build a two-class data-set

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

and minimize the binary cross-entropy

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

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

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

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

Let's define

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

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

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

If we define the optimal discriminator for a given generator

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

our objective becomes

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

We have

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

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

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

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

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

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

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

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

computing

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

amounts to compute

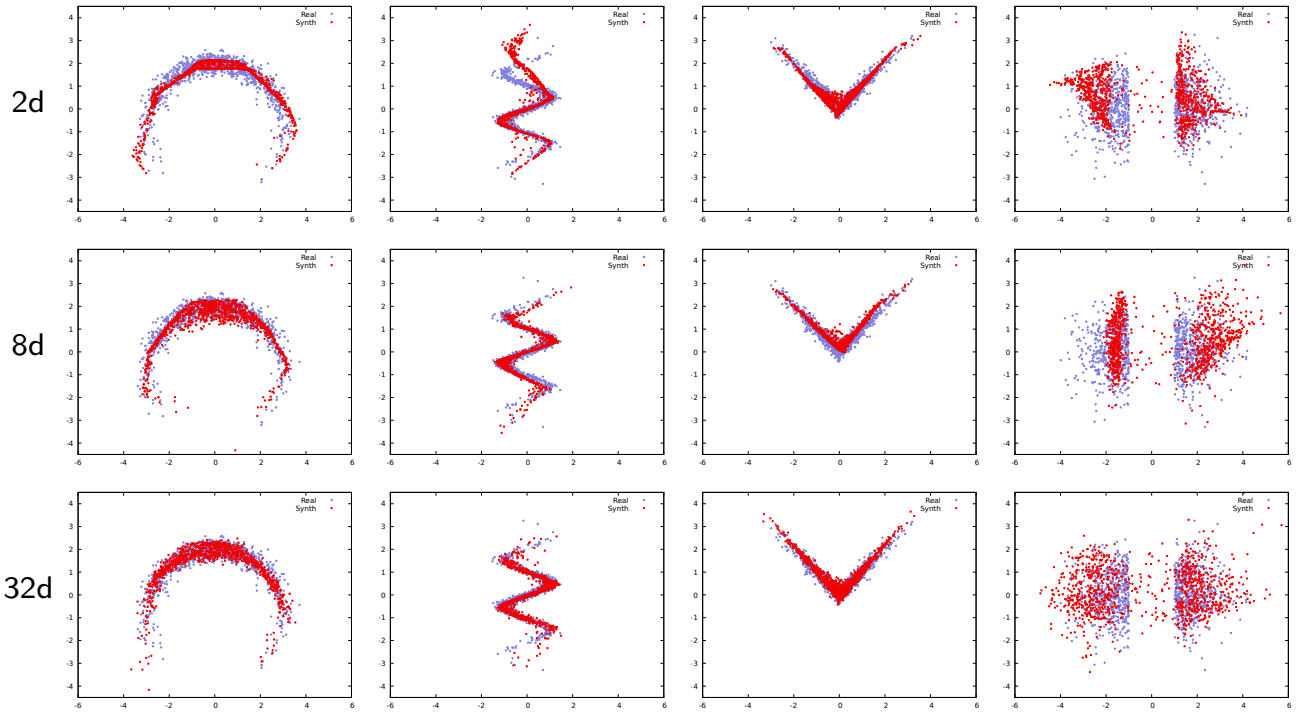
$$\mathbf{G}^* = \operatorname*{argmin}_{\mathbf{G}} \mathbb{D}_{JS}(\mu, \mu_{\mathbf{G}}),$$

where $\mathbb{D}_{JS}$ is a reasonable dissimilarity 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 our minimal example, we alternate gradient steps to improve $\mathbf{G}$ and $\mathbf{D}$.

```
z_dim, nb_hidden = 8, 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 = Variable(real_batch.new(real_batch.size(0), z_dim).normal_())

        fake_batch = model_G(z)
        real_batch = Variable(real_batch)

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

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

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.

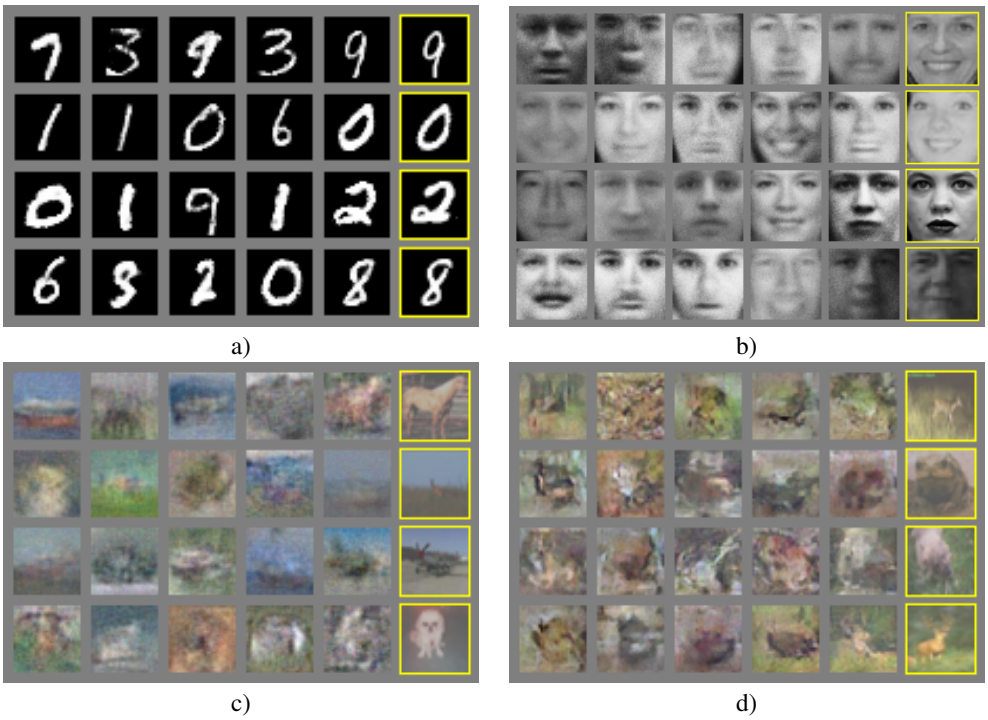


Figure 2: Visualization of samples from the model. Rightmost column shows the nearest training example of the neighboring sample, in order to demonstrate that the model has not memorized the training set. Samples are fair random draws, not cherry-picked. Unlike most other visualizations of deep generative 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)

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 and often boils down to a "beauty contest".

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.

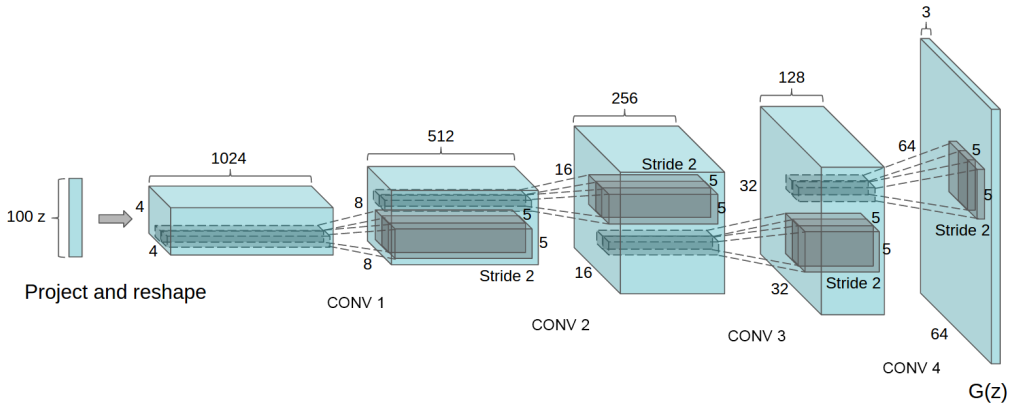


Figure 1: DCGAN generator used for LSUN scene modeling. A 100 dimensional uniform distribution Z is projected to a small spatial extent convolutional representation with many feature maps. A series of four fractionally-strided convolutions (in some recent papers, these are wrongly called deconvolutions) 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 _netG(nn.Module):
    def __init__(self, ngpu):
        super(_netG, 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
        )
```

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

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



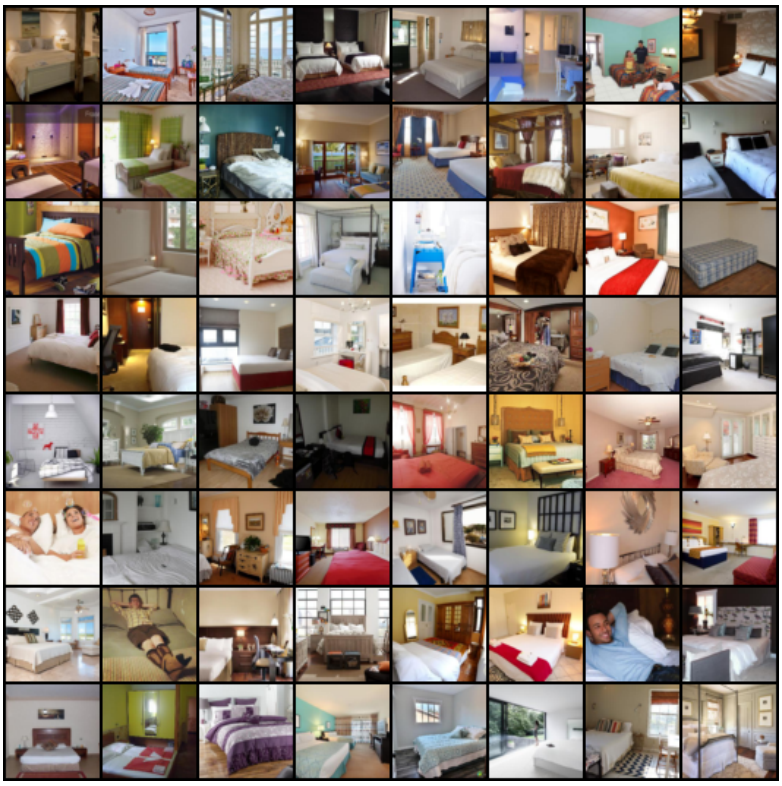
```
# custom weights initialization called on netG and netD
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        m.weight.data.normal_(0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        m.weight.data.normal_(1.0, 0.02)
        m.bias.data.fill_(0)


criterion = nn.BCELoss()

input = torch.FloatTensor(opt.batchSize, 3, opt.imageSize, opt.imageSize)
noise = torch.FloatTensor(opt.batchSize, nz, 1, 1)
fixed_noise = torch.FloatTensor(opt.batchSize, nz, 1, 1).normal_(0, 1)
label = torch.FloatTensor(opt.batchSize)
real_label = 1
fake_label = 0

fixed_noise = Variable(fixed_noise)

# 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))

for epoch in range(opt.niter):
    for i, data in enumerate(dataloader, 0):
```

```
# (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))

# train with real
netD.zero_grad()
real_cpu, _ = data
batch_size = real_cpu.size(0)
if opt.cuda:
    real_cpu = real_cpu.cuda()
input.resize_as_(real_cpu).copy_(real_cpu)
label.resize_(batch_size).fill_(real_label)
inputv = Variable(input)
labelv = Variable(label)

output = netD(inputv)
errD_real = criterion(output, labelv)
errD_real.backward()
D_x = output.data.mean()

# train with fake
noise.resize_(batch_size, nz, 1, 1).normal_(0, 1)
noisev = Variable(noise)
fake = netG(noisev)
labelv = Variable(label.fill_(fake_label))
output = netD(fake.detach())
errD_fake = criterion(output, labelv)
errD_fake.backward()
D_G_z1 = output.data.mean()
errD = errD_real + errD_fake
optimizerD.step()
```

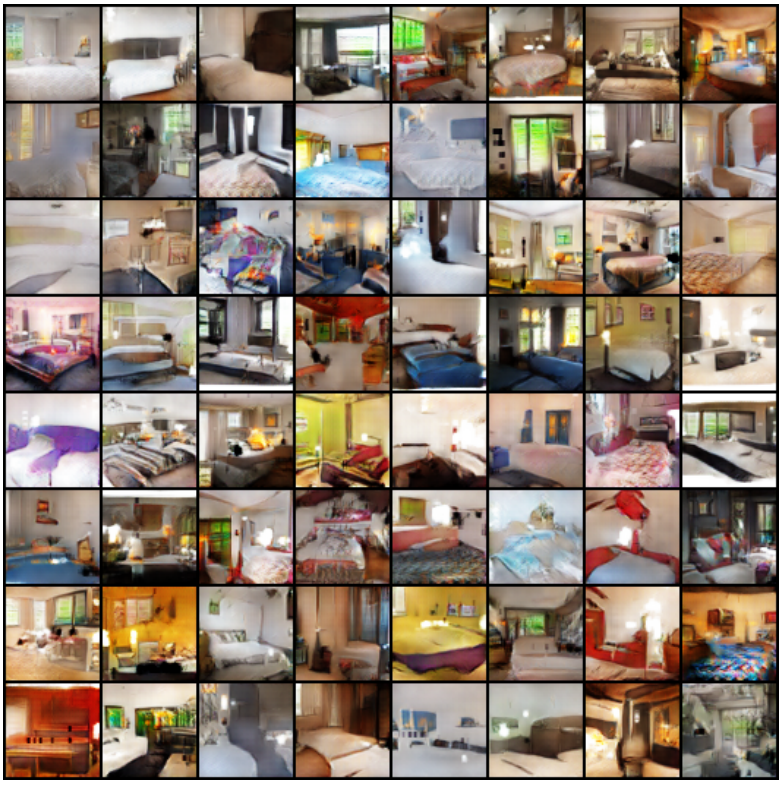


```
# (2) Update G network: maximize log(D(G(z)))

netG.zero_grad()
# fake labels are real for generator cost
labelv = Variable(label.fill_(real_label))
output = netD(fake)
errG = criterion(output, labelv)
errG.backward()
D_G_z2 = output.data.mean()
optimizerG.step()
```

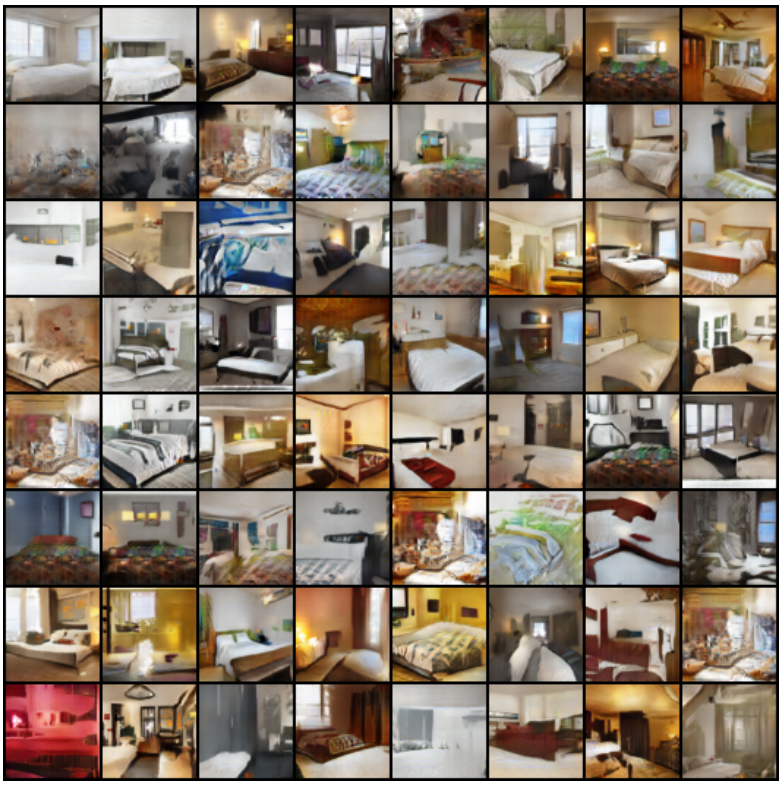
This update of **G** minimizes the loss with inverted labels instead of maximizing it for the correct ones, and hence implements the non-saturating loss.



Real images from LSUN's "bedroom" class.



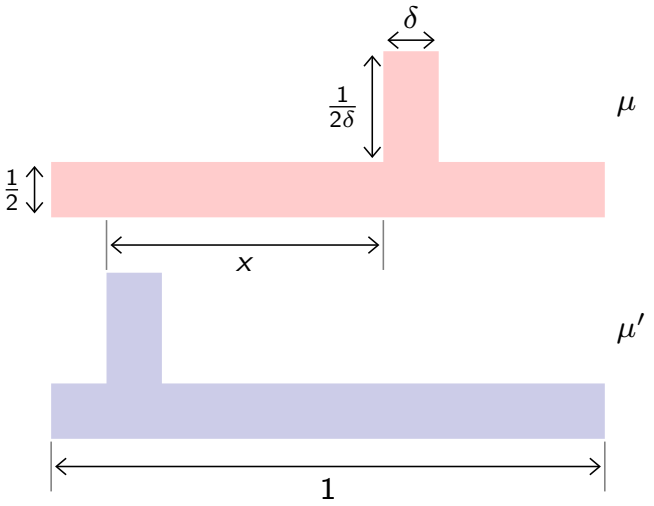
Fake images after 1 epoch (3M images)



Fake images after 20 epochs

Wasserstein GAN

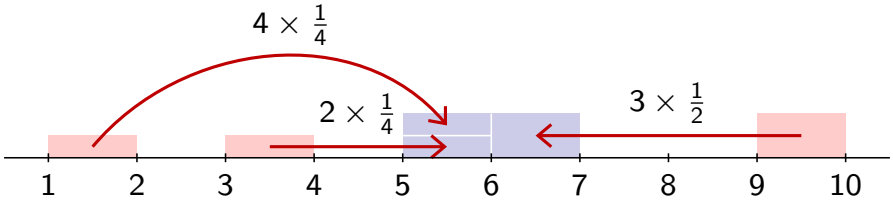
Arjovsky et al. (2017) point out that $\mathbb{D}_{JS}$ does not account [much] for the metric structure of the space.



$$\mathbb{D}_{JS}(\mu, \mu') = \min(\delta, |x|) \left(\frac{1}{\delta} \log \left(1 + \frac{1}{2\delta} \right) - \left(1 + \frac{1}{\delta} \right) \log \left(1 + \frac{1}{\delta} \right) \right)$$

Hence all $|x|$ greater than δ are seen the same.

An alternative choice is the "earth moving distance", which intuitively is the minimum mass displacement to transform one distribution into the other.



$$\mu = \frac{1}{4}\mathbf{1}_{[1,2]} + \frac{1}{4}\mathbf{1}_{[3,4]} + \frac{1}{2}\mathbf{1}_{[9,10]} \qquad \mu' = \frac{1}{2}\mathbf{1}_{[5,7]}$$

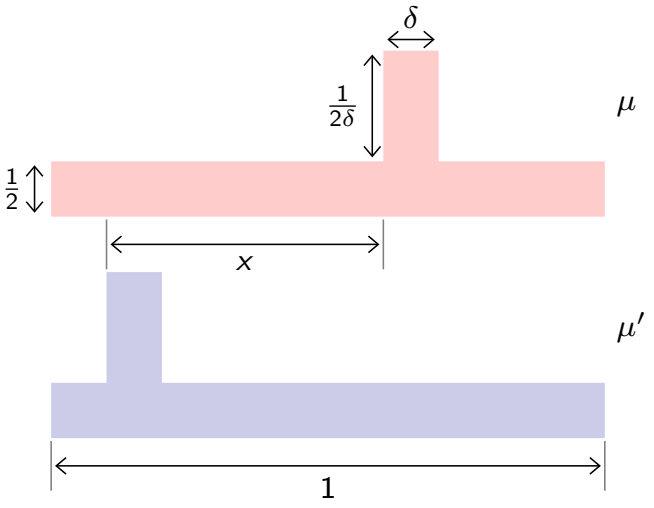
$$\mathbb{W}(\mu, \mu') = 4 \times \frac{1}{4} + 2 \times \frac{1}{4} + 3 \times \frac{1}{2} = 3$$

This distance is also known as the **Wasserstein** distance, defined as

$$\mathbb{W}(\mu, \mu') = \min_{q \in \Pi(\mu, \mu')} \mathbb{E}_{(X, X') \sim q} \Big[\|X - X'\| \Big],$$

where $\Pi(\mu, \mu')$ is the set of distributions over $\mathcal{X}^2$ whose marginals are μ and μ'.

Intuitively, it increases monotonically with the distance between modes



$$\mathbb{W}(\mu, \mu') = \frac{1}{2}|x|$$

So it would make a lot of sense to look for a generator matching the density for this metric, that is

$$\mathbf{G}^* = \operatorname*{argmin}_{\mathbf{G}} \mathbb{W}(\mu, \mu_{\mathbf{G}}).$$

Unfortunately, the definition of $\mathbb{W}$ does not provide an operational way of estimating it.

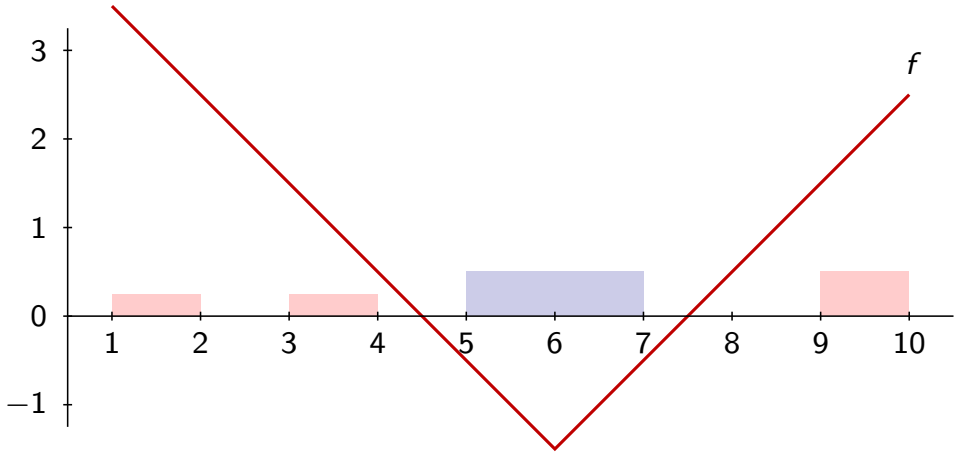
A duality theorem from Kantorovich and Rubinstein implies

$$\mathbb{W}(\mu, \mu') = \max_{\|f\|_L \leq 1} \mathbb{E}_{X \sim \mu}\Big[f(X)\Big] - \mathbb{E}_{X \sim \mu'}\Big[f(X)\Big]$$

where

$$\|f\|_L = \max_{x,x'} \frac{\|f(x) - f(x')\|}{\|x - x'\|}$$

is the Lipschitz seminorm.



$$\mu = \frac{1}{4}\mathbf{1}_{[1,2]} + \frac{1}{4}\mathbf{1}_{[3,4]} + \frac{1}{2}\mathbf{1}_{[9,10]} \qquad \mu' = \frac{1}{2}\mathbf{1}_{[5,7]}$$

$$\mathbb{W}(\mu,\mu') = \underbrace{\left(3\times\frac{1}{4}+1\times\frac{1}{4}+2\times\frac{1}{2}\right)}_{\mathbb{E}_{X\sim\mu}f(X)} - \underbrace{\left(-1\times\frac{1}{2}-1\times\frac{1}{2}\right)}_{\mathbb{E}_{X\sim\mu'}f(X)} = 3$$

Using this result, we are looking for a generator

$$\begin{aligned}\mathbf{G}^* &= \operatorname*{argmin}_{\mathbf{G}} \mathbb{W}(\mu,\mu_{\mathbf{G}})\\ &= \operatorname*{argmin}_{\mathbf{G}} \max_{\|\mathbf{D}\|_L\leq 1} \Big(\mathbb{E}_{X\sim\mu}\Big[\mathbf{D}(X)\Big] - \mathbb{E}_{X\sim\mu_{\mathbf{G}}}\Big[\mathbf{D}(X)\Big]\Big),\end{aligned}$$

where the max is now an optimized predictor.

This is very similar to the original GAN formulation, except that the value of $\mathbf{D}$ is not interpreted through a log-loss, and there is a strong regularization on $\mathbf{D}$.

The main issue in this formulation is to optimize the network $\mathbf{D}$ under a constraint on its Lipschitz seminorm

$$\|\mathbf{D}\|_L \leq 1.$$

Arjovsky et al. achieve this by clipping $\mathbf{D}$'s weights.

The two main benefits observed by Arjovsky et al. are

- A greater stability of the learning process, both in principle and in their experiments: they do not witness "mode collapse".
- A greater interpretability of the loss, which is a better indicator of the quality of the samples.

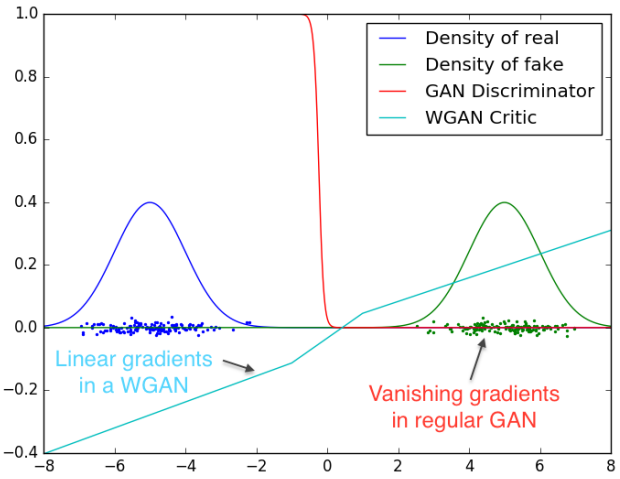


Figure 2: Optimal discriminator and critic when learning to differentiate two Gaussians. As we can see, the traditional GAN discriminator saturates and results in vanishing gradients. Our WGAN critic provides very clean gradients on all parts of the space.

(Arjovsky et al., 2017)

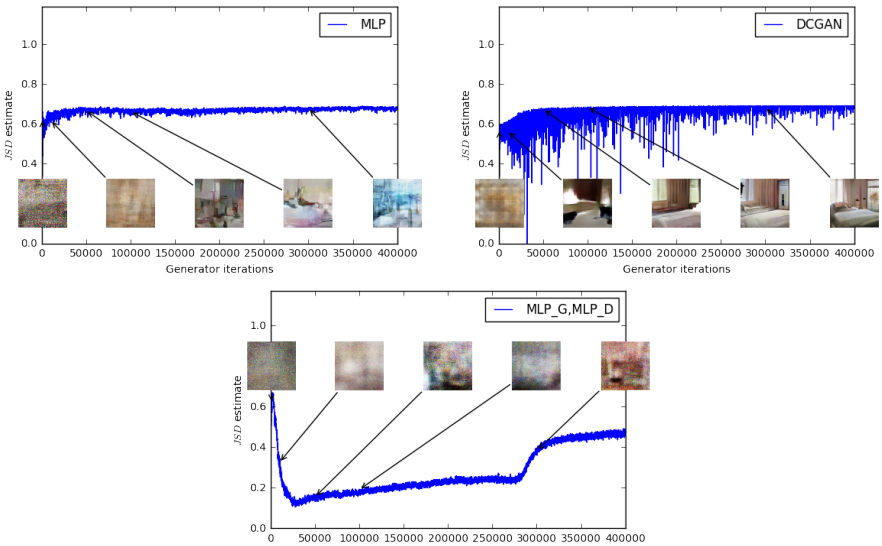


Figure 4: JS estimates for an MLP generator (upper left) and a DCGAN generator (upper right) trained with the standard GAN procedure. Both had a DCGAN discriminator. Both curves have increasing error. *Samples get better for the DCGAN but the JS estimate increases or stays constant, pointing towards no significant correlation between sample quality and loss. Bottom: MLP with both generator and discriminator. The curve goes up and down regardless of sample quality. All training curves were passed through the same median filter as in Figure 3.*

(Arjovsky et al., 2017)

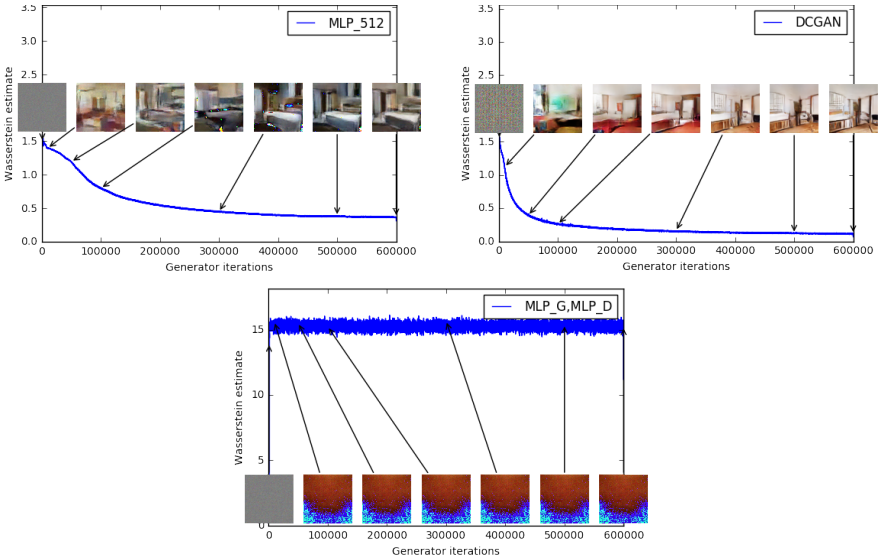


Figure 3: Training curves and samples at different stages of training. We can see a clear correlation between lower error and better sample quality. Upper left: the generator is an MLP with 4 hidden layers and 512 units at each layer. The loss decreases constistently as training progresses and sample quality increases. Upper right: the generator is a standard DCGAN. The loss decreases quickly and sample quality increases as well. In both upper plots the critic is a DCGAN without the sigmoid so losses can be subjected to comparison. Lower half: both the generator and the discriminator are MLPs with substantially high learning rates (so training failed). Loss is constant and samples are constant as well. The training curves were passed through a median filter for visualization purposes.

(Arjovsky et al., 2017)

However, as Arjovsky et al. wrote:

> "Weight clipping is a clearly terrible way to enforce a Lipschitz constraint. If the clipping parameter is large, then it can take a long time for any weights to reach their limit, thereby making it harder to train the critic till optimality. If the clipping is small, this can easily lead to vanishing gradients when the number of layers is big, or batch normalization is not used (such as in RNNs)."

(Arjovsky et al., 2017)

In some way, the resulting Wasserstein GAN (WGAN) trades the difficulty to train **G** for the difficulty to train **D**.

In practice, this weakness results in extremely long convergence time.

Gulrajani et al. (2017) proposed the **improved Wasserstein GAN** in which the constraint on the Lipschitz seminorm is replaced with a smooth penalty term.

They state that if

$$\mathbf{D}^* = \operatorname*{argmax}_{\|\mathbf{D}\|_L \leq 1} \Big(\mathbb{E}_{X \sim \mu} \Big[\mathbf{D}(X) \Big] - \mathbb{E}_{X \sim \mu_{\mathbf{G}}} \Big[\mathbf{D}(X) \Big] \Big)$$

then, with probability one under μ and $\mu_{\mathbf{G}}$

$$\|\nabla \mathbf{D}^*(X)\| = 1.$$

This implies that adding a regularization that pushes the gradient norm to one should not exclude [any of] the optimal discriminator[s].

So instead of looking for

$$\operatorname*{argmax}_{\|\mathbf{D}\|_L \leq 1} \ \mathbb{E}_{X \sim \mu} \Big[\mathbf{D}(X) \Big] - \mathbb{E}_{X \sim \mu_{\mathbf{G}}} \Big[\mathbf{D}(X) \Big],$$

Gulrajani et al. propose to solve

$$\operatorname*{argmax}_{\mathbf{D}} \ \mathbb{E}_{X \sim \mu} \Big[\mathbf{D}(X) \Big] - \mathbb{E}_{X \sim \mu_{\mathbf{G}}} \Big[\mathbf{D}(X) \Big] - \lambda \mathbb{E}_{X \sim \mu_p} \Big[\left(\|\nabla \mathbf{D}(X)\| - 1 \right)^2 \Big]$$

where μ_p is the distribution of a point B sampled uniformly between a real sample X and a fake sample $\mathbf{G}(Z)$, that is $B = UX + (1 - U)X'$ where $X \sim \mu$, $X' \sim \mu_{\mathbf{G}}$, and $U \sim \mathcal{U}[0, 1]$.

Note that this loss involves second-order derivatives.

Experiments show that this scheme is more stable than WGAN under many different conditions.

Conditional GAN

All the models we have seen so far model a density in high dimension and provide means to sample according to it, which is useful for synthesis only.

However, most of the practical applications require the ability to sample a **conditional distribution.** E.g.:

- Next frame prediction.
- "in-painting",
- segmentation,
- style transfer.

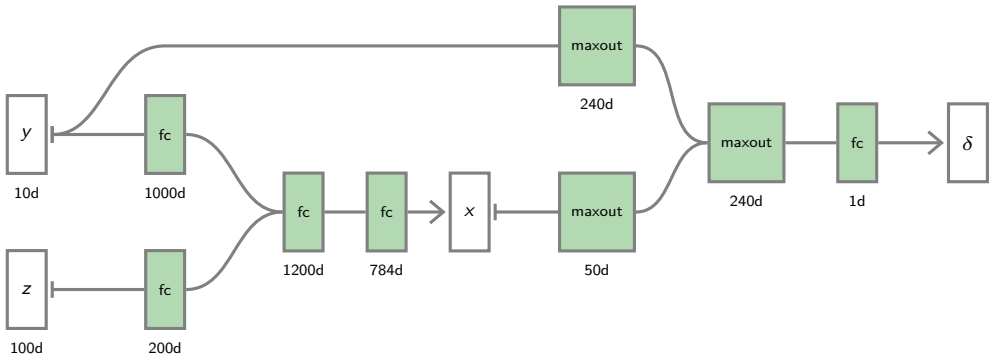
The Conditional GAN proposed by Mirza and Osindero (2014) consists of parameterizing both **G** and **D** by a conditioning quantity Y.

$$V(\mathbf{D},\mathbf{G}) = \mathbb{E}_{(X,Y)\sim\mu}\Big[\log \mathbf{D}(X,Y)\Big] + \mathbb{E}_{Z\sim\mathcal{N}(0,I),Y\sim\mu_Y}\Big[\log(1-\mathbf{D}(\mathbf{G}(Z,Y),Y))\Big],$$

To generate MNIST characters, with

$$Z \sim \mathcal{U}\left([0,1]^{100}\right),$$

and conditioned with the class y, encoded as a one-hot vector of dimension 10, they propose



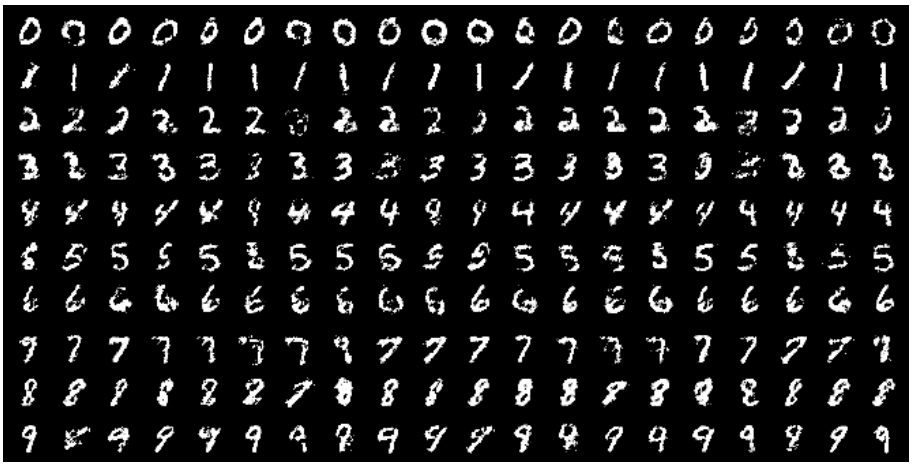


Figure 2: Generated MNIST digits, each row conditioned on one label

(Mirza and Osindero, 2014)

Image-to-Image translations

The main issue to generate realistic signal is that the value X to predict may remain non-deterministic given the conditioning quantity Y.

For a loss function such as MSE, the best fit is $E(X|Y = y)$ which can be pretty different from the MAP, or from any reasonable sample from $\mu_{X|Y=y}$.

In practice for images there are often remaining location indeterminacy that results into a blurry prediction.

Sampling according to $\mu_{X|Y=y}$ is the proper way to address the problem.

Isola et al. (2016) use conditional GANs to address this issue for the "translation" of images with pixel-to-pixel correspondence:

- edges to realistic photos,
- semantic segmentation,
- gray-scales to colors, etc.

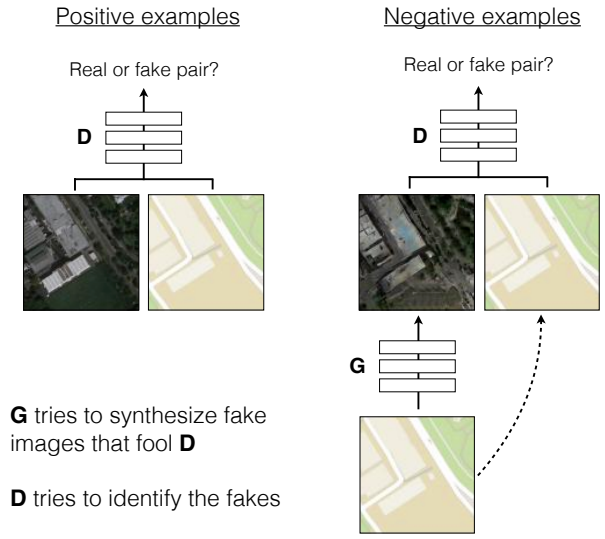


Figure 2: Training a conditional GAN to predict aerial photos from maps. The discriminator, D, learns to classify between real and synthesized pairs. The generator learns to fool the discriminator. Unlike an unconditional GAN, both the generator and discriminator observe an input image.

(Isola et al., 2016)

They define

$$V(\mathbf{D},\mathbf{G}) = \mathbb{E}_{(X,Y)\sim\mu}\Big[\log \mathbf{D}(Y,X)\Big] + \mathbb{E}_{Z\sim\mu_Z,X\sim\mu_X}\Big[\log(1-\mathbf{D}(\mathbf{G}(Z,X),X))\Big],$$

$$\mathscr{L}_{L^1}(\mathbf{G}) = \mathbb{E}_{(X,Y)\sim\mu,Z\sim\mathcal{N}(0,I)}\Big[\|Y-\mathbf{G}(Z,X)\|_1\Big],$$

and

$$\mathbf{G}^* = \operatorname*{argmin}_{\mathbf{G}}\max_{\mathbf{D}} \; V(\mathbf{D},\mathbf{G}) + \lambda\mathscr{L}_{L^1}(\mathbf{G}).$$

The term $\mathscr{L}_{L^1}$ pushes toward proper pixel-wise prediction, and V makes the generator prefer realistic images to better fitting pixel-wise.

Note that Isola et al. switch the meaning of X and Y wrt Mirza and Osindero. Here X is the conditioning quantity and Y the signal to generate.

For **G**, they start with Radford et al. (2015)'s DCGAN architecture and add skip connections from layer i to layer $D - i$ that concatenate channels.

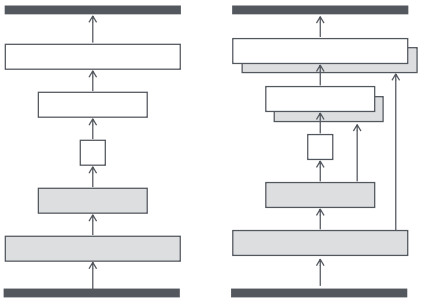


Figure 3: Two choices for the architecture of the generator. The "U-Net" [34] is an encoder-decoder with skip connections between mirrored layers in the encoder and decoder stacks.

(Isola et al., 2016)

Randomness Z is provided through dropout, and not as an additional input.

The discriminator **D** is a regular convnet which scores overlapping patches of size $N \times N$ and averages the scores for the final one.

This controls the network's complexity, while allowing to detect any inconsistency of the generated image (*e.g.* blurriness).

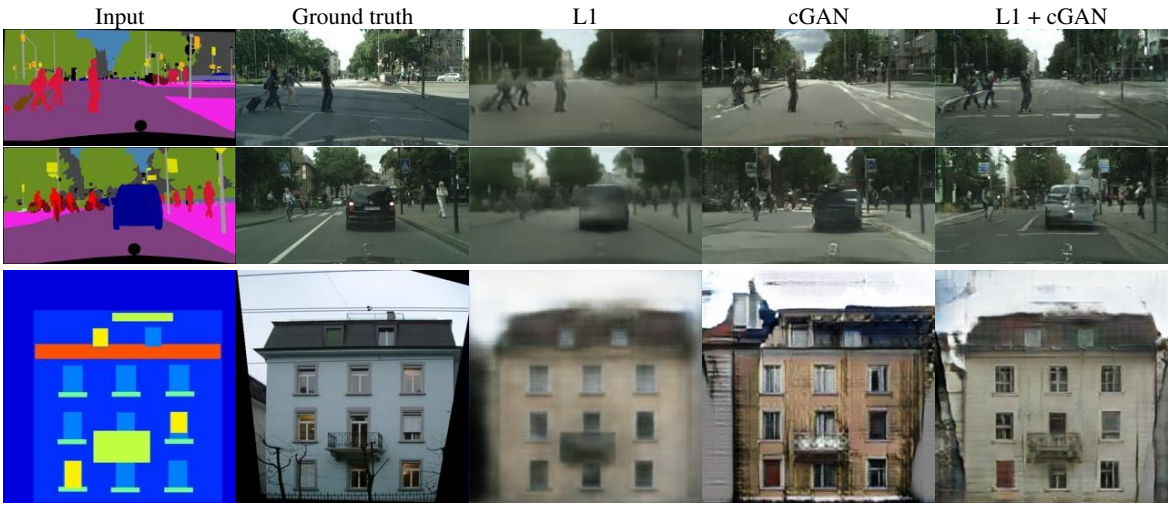


Figure 4: Different losses induce different quality of results. Each column shows results trained under a different loss. Please see `https://phillipi.github.io/pix2pix/` for additional examples.

(Isola et al., 2016)

Figure 6: Patch size variations. Uncertainty in the output manifests itself differently for different loss functions. Uncertain regions become blurry and desaturated under L1. The 1x1 PixelGAN encourages greater color diversity but has no effect on spatial statistics. The 16x16 PatchGAN creates locally sharp results, but also leads to tiling artifacts beyond the scale it can observe. The 70x70 PatchGAN forces outputs that are sharp, even if incorrect, in both the spatial and spectral (coforfulness) dimensions. The full 256x256 ImageGAN produces results that are visually similar to the 70x70 PatchGAN, but somewhat lower quality according to our FCN-score metric (Table 2). Please see `https://phillipi.github.io/pix2pix/` for additional examples.

(Isola et al., 2016)

Figure 8: Example results on Google Maps at 512x512 resolution (model was trained on images at 256x256 resolution, and run convolutionally on the larger images at test time). Contrast adjusted for clarity.

(Isola et al., 2016)

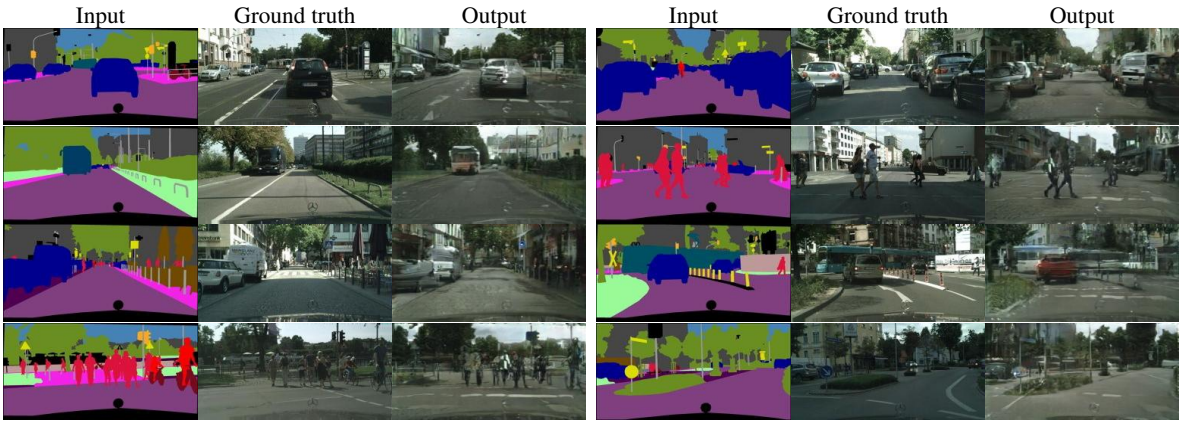


Figure 11: Example results of our method on Cityscapes labels→photo, compared to ground truth.

(Isola et al., 2016)

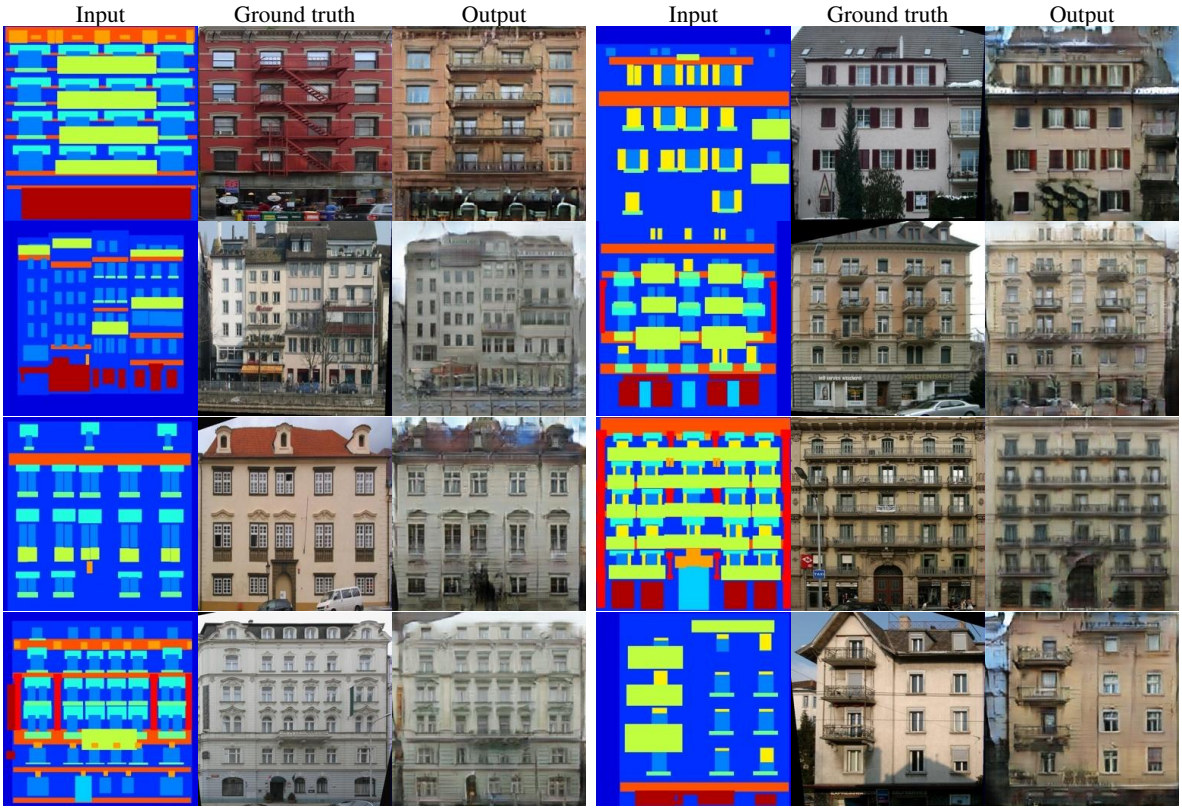


Figure 12: Example results of our method on facades labels→photo, compared to ground truth

(Isola et al., 2016)

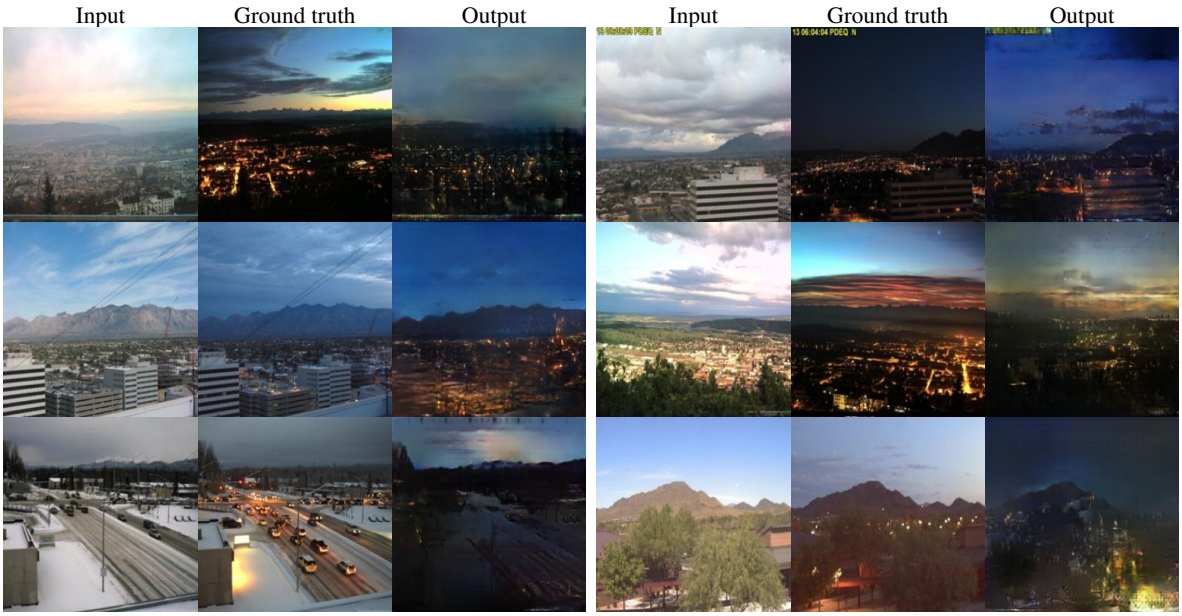


Figure 13: Example results of our method on day→night, compared to ground truth.

(Isola et al., 2016)

Figure 14: Example results of our method on automatically detected edges→handbags, compared to ground truth.

(Isola et al., 2016)

Figure 16: Example results of the edges→photo models applied to human-drawn sketches from [10]. Note that the models were trained on automatically detected edges, but generalize to human drawings

(Isola et al., 2016)

The main drawback of this technique is that it requires pairs of samples with pixel-to-pixel correspondence.

In many cases, one has at its disposal examples from two densities and wants to translate a sample from the first ("images of apples") into a sample likely under the second ("images of oranges").

We consider X r.v. on $\mathscr{X}$ a sample from the first data-set, and Y r.v. on $\mathscr{Y}$ a sample for the second data-set. Zhu et al. (2017) propose to train at the same time two mappings

$$\begin{aligned}\mathbf{G} &: \mathscr{X} \to \mathscr{Y}\\ \mathbf{F} &: \mathscr{Y} \to \mathscr{X}\end{aligned}$$

such that

$$\begin{aligned}\mathbf{G}(X) &\sim \mu_Y,\\ \mathbf{G} \circ \mathbf{F}(X) &\simeq X.\end{aligned}$$

Where the matching in density is characterized with a discriminator $\mathbf{D}_Y$ and the reconstruction with the L^1 loss. They also do this both ways symmetrically.

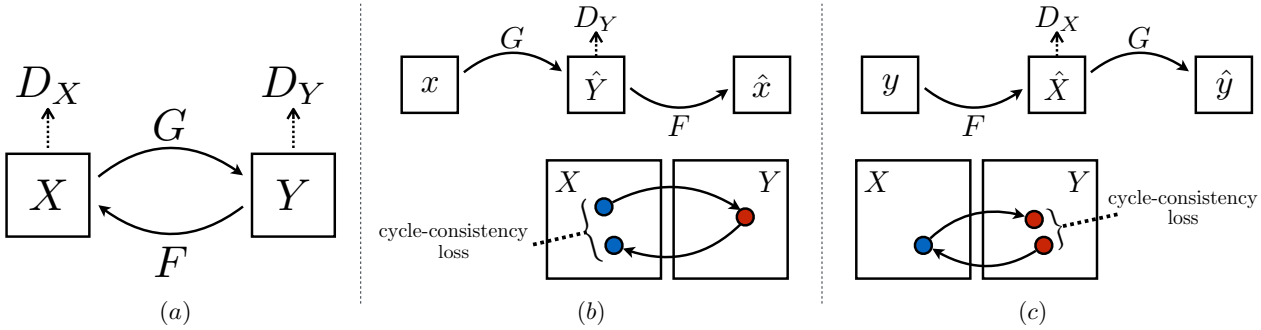


Figure 3: (a) Our model contains two mapping functions $G : X \rightarrow Y$ and $F : Y \rightarrow X$, and associated adversarial discriminators D_Y and D_X. D_Y encourages G to translate X into outputs indistinguishable from domain Y, and vice versa for D_X and F. To further regularize the mappings, we introduce two *cycle consistency losses* that capture the intuition that if we translate from one domain to the other and back again we should arrive at where we started: (b) forward cycle-consistency loss: $x \rightarrow G(x) \rightarrow F(G(x)) \approx x$, and (c) backward cycle-consistency loss: $y \rightarrow F(y) \rightarrow G(F(y)) \approx y$

(Zhu et al., 2017)

The generator is from Johnson et al. (2016), an updated version of the one from Radford et al. (2015)'s DCGAN.

The loss optimized alternatively is

$$\begin{aligned} V^*(\mathbf{G}, \mathbf{F}, \mathbf{D}_X, \mathbf{D}_Y) = & V(\mathbf{G}, \mathbf{D}_Y, X, Y) + V(\mathbf{F}, \mathbf{D}_X, Y, X) \\ & + \lambda \left(\mathbb{E}\Big[\|\mathbf{F}(\mathbf{G}(X)) - X\|_1 \Big] + \mathbb{E}\Big[\|\mathbf{G}(\mathbf{F}(Y)) - Y\|_1 \Big] \right) \end{aligned}$$

where V is a quadratic loss, instead of the usual log (Mao et al., 2016)

$$V(\mathbf{G}, \mathbf{D}_Y, X, Y) = \mathbb{E}\Big[(\mathbf{D}_Y(Y) - 1)^2 \Big] + \mathbb{E}\Big[\mathbf{D}_Y(\mathbf{G}(X))^2 \Big].$$

As always, there are plenty of specific technical details in the models and the training, *e.g.* using an history of generated images (Shrivastava et al., 2016).

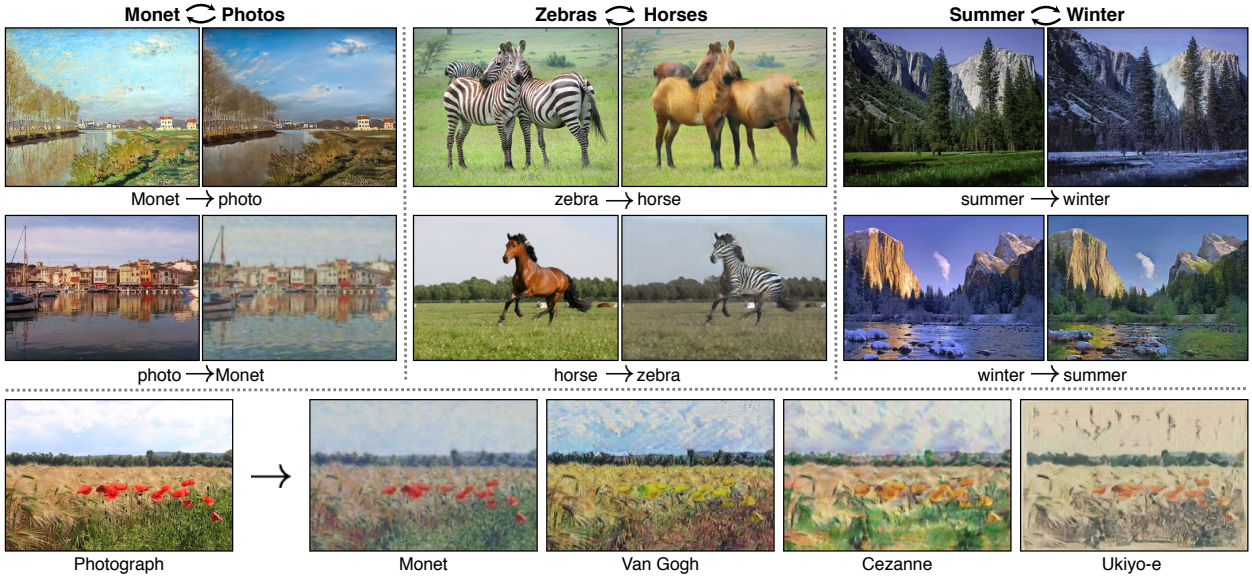
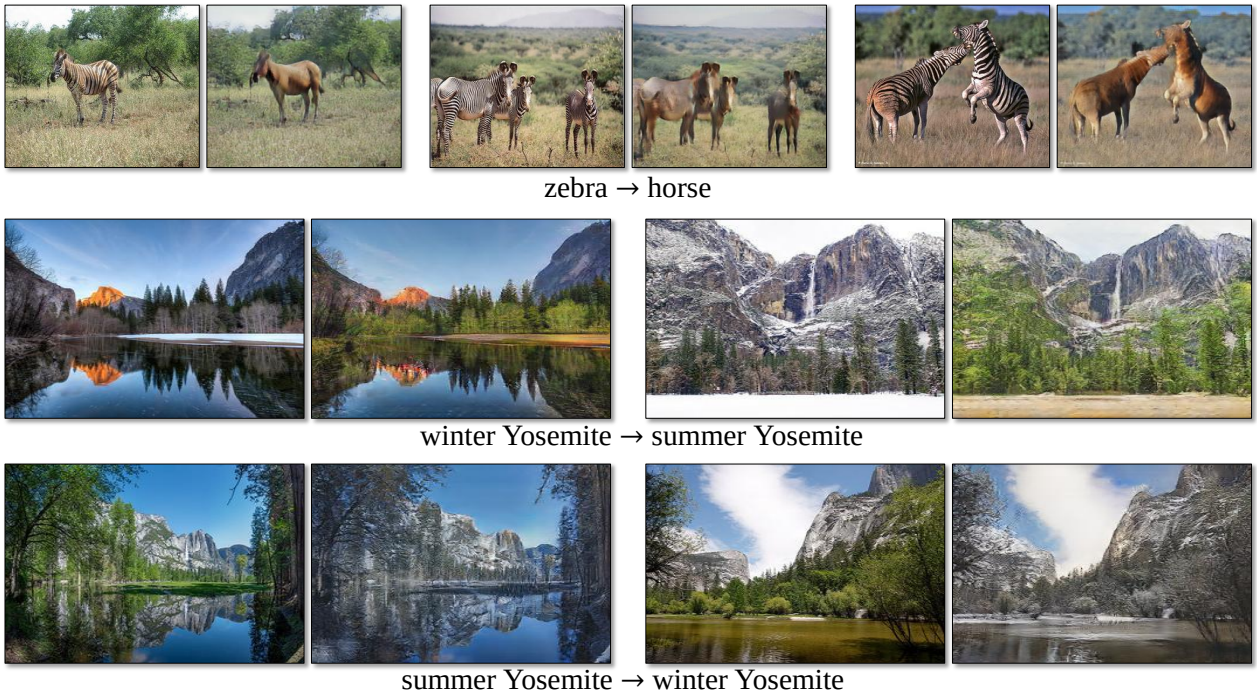


Figure 1: Given any two unordered image collections X and Y, our algorithm learns to automatically "translate" an image from one into the other and vice versa: *(left)* Monet paintings and landscape photos from Flickr; *(center)* zebras and horses from ImageNet; *(right)* summer and winter Yosemite photos from Flickr. Example application *(bottom)*: using a collection of paintings of famous artists, our method learns to render natural photographs into the respective styles.

(Zhu et al., 2017)



(Zhu et al., 2017)

apple → orange

orange → apple

(Zhu et al., 2017)

While GANs are often used for their [theoretical] ability to model a distribution fully and accurately, generating consistent samples is enough for image-to-image translation.

In particular, this application does not suffer much from mode collapse, as long as the generated images "look nice".

The key aspect of the GAN here is the "perceptual loss" that the discriminator implements, more than the theoretical convergence to the true distribution.

Model persistence and checkpoints in PyTorch

Saving and loading models is key to use models trained previously.

It also allows to implement **checkpoints** which keep track of the state during training and allow to either restart after an expected interruption, or modulate meta-parameters manually.

The underlying operation is **serialization**, that is the transcription of an arbitrary object into a sequence of bytes saved on disk.

The main PyTorch methods for serializing are `torch.save(obj, filename)` and `torch.load(filename)`.

```
>>> x = 34
>>> torch.save(x, 'x.pth')
>>> y = torch.load('x.pth')
>>> y
34
```

```
>>> z = { 'a': torch.LongTensor(2, 3).random_(10),
...       'b': nn.Linear(10, 20) }
>>> torch.save(z, 'z.pth')
>>> w = torch.load('z.pth')
>>> w
{'a':
 2  2  3
 8  9  8
[torch.LongTensor of size 2x3]
, 'b': Linear(in_features=10, out_features=20)}
```


One can save directly a full model like this, including arbitrary fields

```
>>> x = nn.Sequential(nn.Linear(3, 10), nn.ReLU(), nn.Linear(10, 1))
>>> x.blah = 14
>>> torch.save(x, 'model.pth')
>>>
>>> z = torch.load('model.pth')
>>> z(Variable(Tensor(2, 3).normal_()))
Variable containing:
 0.2408
 0.0929
[torch.FloatTensor of size 2x1]

>>> z.blah
14
```

Saving a full model with `torch.save()` bounds the saved quantities to the specific class implementation, and may break after changes in the code.

The suggested policy is to save the **state dictionary** alone, as provided by `Module.state_dict()`, which encompasses `Parameters` and **buffers** such as batchnorm running estimates, etc.

Additionally

- Tensors are saved with their locations (CPU, or GPU), and will be loaded in the same configuration,
- in your `Module`s, buffers have to be identified with `register_buffer`,
- loaded models are in train mode by default,
- optimizers have a state too (momentum, Adam).

A checkpoint is a persistent object that keeps the global state of the training: model and optimizer. In the following example (1) we load it when we start if it exists, and (2) we save it at every epoch.

```
criterion = nn.CrossEntropyLoss()

nb_epochs_finished = 0
model = Net()
optimizer = torch.optim.SGD(model.parameters(), lr = lr)

checkpoint_name = 'checkpoint.pth'

try:
    checkpoint = torch.load(checkpoint_name)
    nb_epochs_finished = checkpoint['nb_epochs_finished']
    model.load_state_dict(checkpoint['model_state'])
    optimizer.load_state_dict(checkpoint['optimizer_state'])
    print('Checkpoint loaded with {:d} epochs finished.'.format(nb_epochs_finished))

except FileNotFoundError:
    print('Starting from scratch.')

except:
    print('Error when loading the checkpoint.')
    exit(1)

if torch.cuda.is_available():
    torch.backends.cudnn.benchmark = True
    model.cuda()
    criterion.cuda()
```

```
for k in range(nb_epochs_finished, nb_epochs):
    acc_loss = 0
    for b in range(0, train_input.size(0), batch_size):
        output = model(train_input.narrow(0, b, batch_size))
        loss = criterion(output, train_target.narrow(0, b, batch_size))
        acc_loss += loss.data[0]
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(k, acc_loss)

    checkpoint = {
        'nb_epochs_finished': k + 1,
        'model_state': model.state_dict(),
        'optimizer_state': optimizer.state_dict()
    }
    torch.save(checkpoint, checkpoint_name)
```


If we `killall python` during training

```
fleuret@elk:/tmp ./tinywithcheckpoint.py
Starting from scratch.
0 155.7866949379677
1 34.80593343087821
2 23.501393611499225
Terminated
```

and re-start

```
fleuret@elk:/tmp ./tinywithcheckpoint.py
Checkpoint loaded with 3 epochs finished.
3 17.466753122906084
4 13.512543070963147
5 10.474066113200024
6 8.01903374180074
7 6.152274705537366
8 4.789176231754482
9 3.5722521024140406
test_error 0.97% (97/10000)
```

Since a model is saved with information about the CPU/GPUs where each `Storage` is located there may be issues if the model is loaded on a different hardware configuration.

For instance, if we save a model located on a GPU:

```
>>> import torch
>>> from torch import nn
>>> x = nn.Linear(10, 4)
>>> x.cuda()
Linear(in_features=10, out_features=4, bias=True)
>>> torch.save(x, 'x.pth')
```

And load it on a machine without GPU:

```
>>> import torch
>>> from torch import nn
>>> x = torch.load('x.pth')
THCudaCheck FAIL file=torch/csrc/cuda/Module.cpp ...
```

This can be fixed by specifying at load time how to relocate storages:

```
>>> x = torch.load('x.pth', map_location = lambda storage, loc: storage)
```


References

M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein gan. *CoRR*, abs/1701.07875, 2017.

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.

I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville. Improved training of wasserstein gans. *CoRR*, abs/1704.00028, 2017.

P. Isola, J. Zhu, T. Zhou, and A. A. Efros. Image-to-image translation with conditional adversarial networks. *CoRR*, abs/1611.07004, 2016.

J. Johnson, A. Alahi, and L. Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. In *European Conference on Computer Vision (ECCV)*, 2016.

X. Mao, Q. Li, H. Xie, R. Lau, Z. Wang, and S. Smolley. Least squares generative adversarial networks. *CoRR*, abs/1611.04076, 2016.

M. Mirza and S. Osindero. Conditional generative adversarial nets. *CoRR*, abs/1411.1784, 2014.

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

A. Shrivastava, T. Pfister, O. Tuzel, J. Susskind, W. Wang, and R. Webb. Learning from simulated and unsupervised images through adversarial training. *CoRR*, abs/1612.07828, 2016.

J. Zhu, T. Park, P. Isola, and A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. *CoRR*, abs/1703.10593, 2017.