Generative Adversarial Networks (GANs)

By: Ismail Elezi
ismail.elezi@gmail.com

Supervised Learning vs Unsupervised Learning

Supervised Learning

Data: (x, y)
x is data, y is label

Goal: Learn a *function* to map x -> y

Examples: Classification, regression, object detection, semantic segmentation, image captioning, etc.

Supervised Learning vs Unsupervised Learning

Supervised Learning

Data: (x, y)
x is data, y is label

Goal: Learn a *function* to map x -> y

Examples: Classification, regression, object detection, semantic segmentation, image captioning, etc.

→ Cat

Classification

Supervised Learning vs Unsupervised Learning

Supervised Learning

Data: (x, y)
x is data, y is label

Goal: Learn a *function* to map x -> y

Examples: Classification, regression, object detection, semantic segmentation, image captioning, etc.

DOG, DOG, CAT

Object Detection

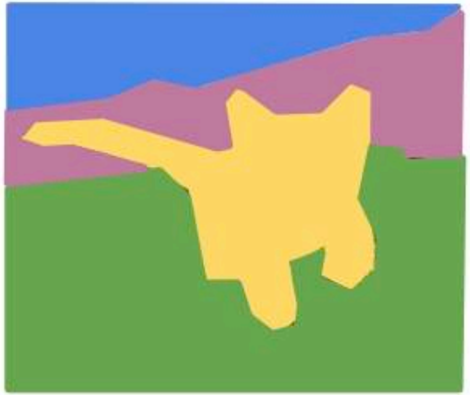
Supervised Learning vs Unsupervised Learning

Supervised Learning

Data: (x, y)
x is data, y is label

Goal: Learn a *function* to map x -> y

Examples: Classification, regression, object detection, semantic segmentation, image captioning, etc.



GRASS, CAT, TREE, SKY

Semantic Segmentation

Supervised Learning vs Unsupervised Learning

Unsupervised Learning

Data: x
Just data, no labels!

Goal: Learn some underlying hidden *structure* of the data

Examples: Clustering, dimensionality reduction, feature learning, density estimation, etc.

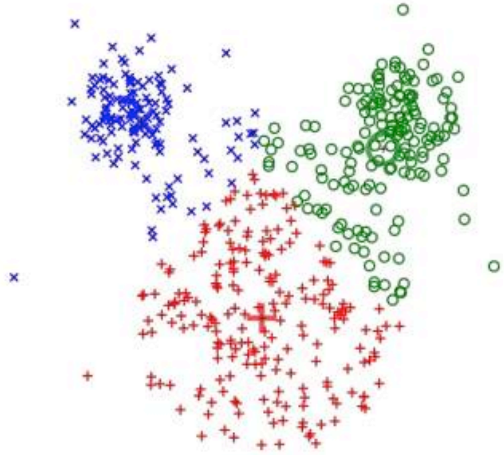
Supervised Learning vs Unsupervised Learning

Unsupervised Learning

Data: x
Just data, no labels!

Goal: Learn some underlying hidden *structure* of the data

Examples: Clustering, dimensionality reduction, feature learning, density estimation, etc.



K-means clustering

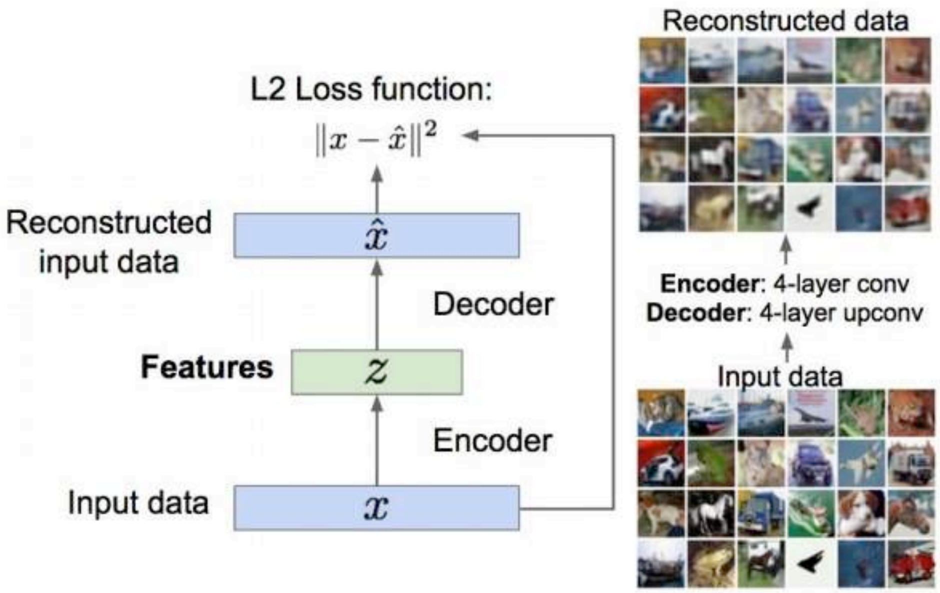
Supervised Learning vs Unsupervised Learning

Unsupervised Learning

Data: x
Just data, no labels!

Goal: Learn some underlying hidden *structure* of the data

Examples: Clustering, dimensionality reduction, feature learning, density estimation, etc.



Autoencoders
(Feature learning)

Supervised Learning vs Unsupervised Learning

Supervised Learning

Data: (x, y)
x is data, y is label

Goal: Learn a *function* to map x -> y

Examples: Classification, regression, object detection, semantic segmentation, image captioning, etc.

Unsupervised Learning

Training data is cheap

Data: x
Just data, no labels!

Holy grail: Solve unsupervised learning => understand structure of visual world

Goal: Learn some underlying hidden *structure* of the data

Examples: Clustering, dimensionality reduction, feature learning, density estimation, etc.

Generative Models

Given training data, generate new samples from same distribution

Training data ~ $p_{data}(x)$

Generated samples ~ $p_{model}(x)$

Want to learn $p_{model}(x)$ similar to $p_{data}(x)$

Addresses density estimation, a core problem in unsupervised learning

Several flavors:

- Explicit density estimation: explicitly define and solve for $p_{model}(x)$
- Implicit density estimation: learn model that can sample from $p_{model}(x)$ w/o explicitly defining it

Taxonomy of Generative Models

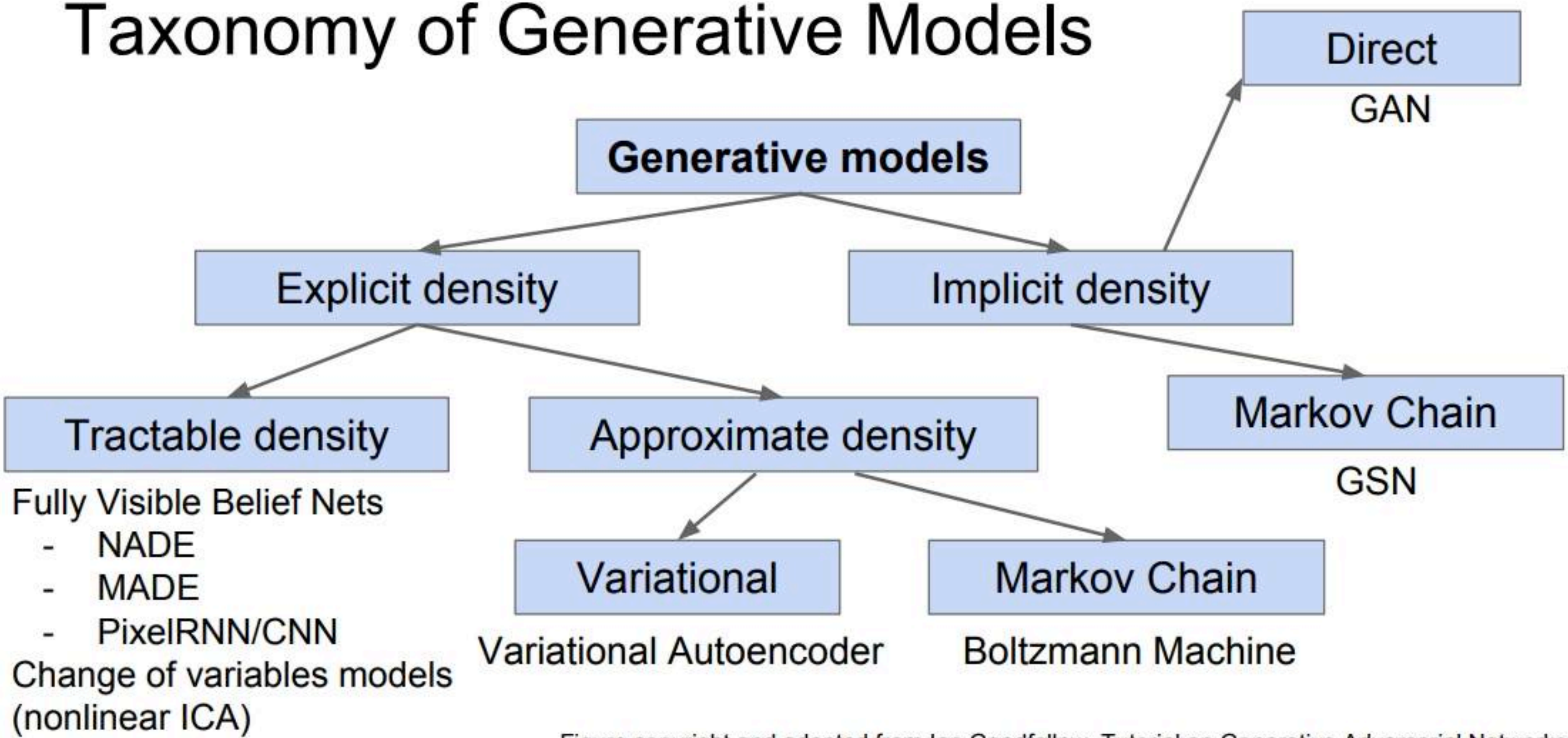


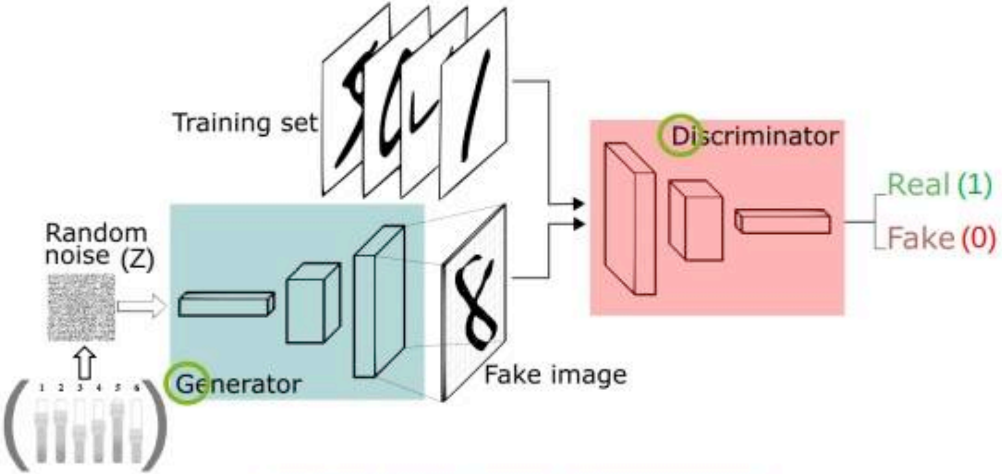
Figure copyright and adapted from Ian Goodfellow, Tutorial on Generative Adversarial Networks, 2017.

Generative Adversarial Networks

Generative Adversarial Networks

Train 2 models simultaneously [1]

- G: Generator
  → learns to generate data
- D: Discriminator
  → learns $p(x \text{ not being generated})$



Sources: https://deeplearning4j.org/generative-adversarial-network; http://www.dpkingma.com/sgvb_mnist_demo/demo.html

→ Both differentiable functions D&G learn while competing
→ The latent space Z serves as a source of variation to generate different data points
→ Only D has access to real data

Credit: Thilo Stadelmann

Minimax Game on GANs

Generator network: try to fool the discriminator by generating real-looking images
Discriminator network: try to distinguish between real and fake images

Train jointly in **minimax game**

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Minimax Game on GANs

Generator network: try to fool the discriminator by generating real-looking images
Discriminator network: try to distinguish between real and fake images

Train jointly in **minimax game**

Minimax objective function:

Discriminator outputs likelihood in (0,1) of real image

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log \underbrace{D_{\theta_d}(x)}_{\text{Discriminator output for real data x}} + \mathbb{E}_{z \sim p(z)} \log(1 - \underbrace{D_{\theta_d}(G_{\theta_g}(z))}_{\text{Discriminator output for generated fake data G(z)}}) \right]$$

Minimax Game on GANs

Generator network: try to fool the discriminator by generating real-looking images
Discriminator network: try to distinguish between real and fake images

Train jointly in **minimax game**

Discriminator outputs likelihood in (0,1) of real image

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log \underbrace{D_{\theta_d}(x)}_{\text{Discriminator output for real data x}} + \mathbb{E}_{z \sim p(z)} \log(1 - \underbrace{D_{\theta_d}(G_{\theta_g}(z))}_{\text{Discriminator output for generated fake data G(z)}}) \right]$$

- Discriminator (θ_d) wants to **maximize objective** such that D(x) is close to 1 (real) and D(G(z)) is close to 0 (fake)
- Generator (θ_g) wants to **minimize objective** such that D(G(z)) is close to 1 (discriminator is fooled into thinking generated G(z) is real)

Minimax Game on GANs

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x\sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z\sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))\right]$$

Alternate between:

1. **Gradient ascent** on discriminator

$$\max_{\theta_d} \left[\mathbb{E}_{x\sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z\sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))\right]$$

2. **Gradient descent** on generator

$$\min_{\theta_g} \mathbb{E}_{z\sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

Minimax Game on GANs

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Alternate between:

1. **Gradient ascent** on discriminator

$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

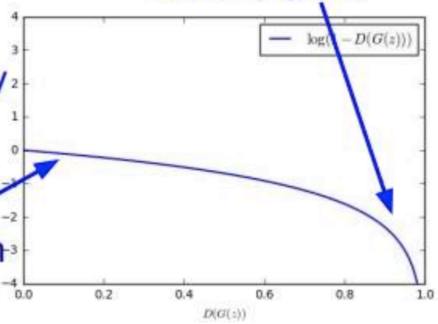
2. **Gradient descent** on generator

$$\min_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

In practice, optimizing this generator objective does not work well!

When sample is likely fake, want to learn from it to improve generator. But gradient in this region is relatively flat!

Gradient signal dominated by region where sample is already good



Alternative Cost Function

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Alternate between:

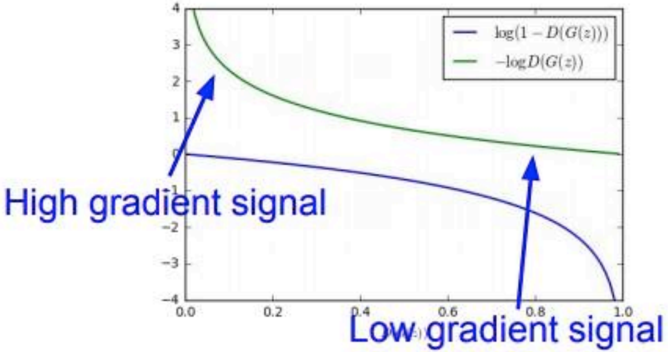
1. **Gradient ascent** on discriminator

$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

2. **Instead: Gradient ascent** on generator, different objective

$$\max_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(D_{\theta_d}(G_{\theta_g}(z)))$$

Instead of minimizing likelihood of discriminator being correct, now maximize likelihood of discriminator being wrong.
Same objective of fooling discriminator, but now higher gradient signal for bad samples => works much better! Standard in practice.



GAN Training Algorithm

for number of training iterations **do**

for k steps **do**

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

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

end for

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Update the generator by ascending its stochastic gradient (improved objective):

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log(D_{\theta_d}(G_{\theta_g}(z^{(i)})))$$

end for

Ian Goodfellow et al, Generative Adversarial Networks, NIPS 2014

```python
class Generator(nn.Module):
    def __init__(self, latent, img_shape):
        super(Generator, self).__init__()

        self.model = nn.Sequential(
            nn.Linear(latent, 128),
            nn.ReLU(),
            nn.Linear(128, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, 1024),
            nn.ReLU(),
            nn.Linear(1024, int(np.prod(img_shape))),
            nn.Tanh()
        )

    def forward(self, z, img_shape):
        img = self.model(z)
        img = img.view(img.size(0), *img_shape)
        return img
```

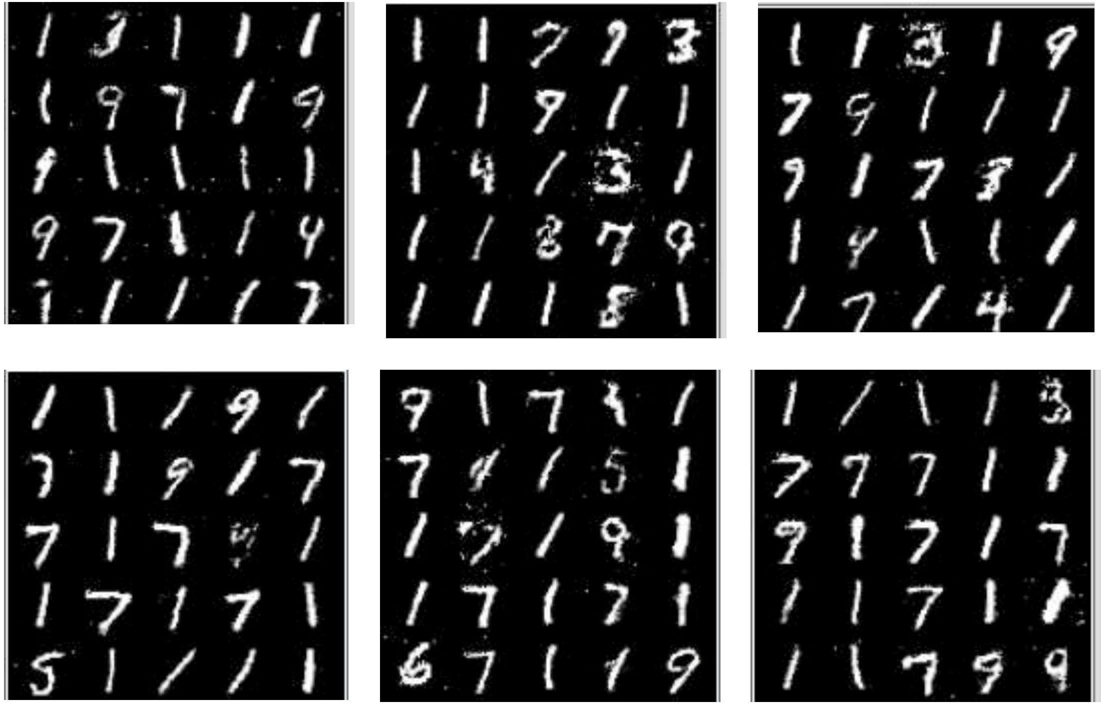
```python
class Discriminator(nn.Module):
    def __init__(self, img_shape):
        super(Discriminator, self).__init__()

        self.model = nn.Sequential(
            nn.Linear(int(np.prod(img_shape)), 512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, 1),
            nn.Sigmoid()
        )

    def forward(self, img):
        img_flat = img.view(img.size(0), -1)
        prob = self.model(img_flat)
        return prob
```

```python
class Generator(nn.Module):
    def __init__(self, latent, img_shape):
        super(Generator, self).__init__()

        self.model = nn.Sequential(
            nn.Linear(latent, 128),
            nn.ReLU(),
            nn.Linear(128, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, 1024),
            nn.ReLU(),
            nn.Linear(1024, int(np.prod(img_shape))),
            nn.Tanh()
        )

    def forward(self, z, img_shape):
        img = self.model(z)
        img = img.view(img.size(0), *img_shape)
        return img
```

```python
class Discriminator(nn.Module):
    def __init__(self, img_shape):
        super(Discriminator, self).__init__()

        self.model = nn.Sequential(
            nn.Linear(int(np.prod(img_shape)), 512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, 1),
            nn.Sigmoid()
        )

    def forward(self, img):
        img_flat = img.view(img.size(0), -1)
        prob = self.model(img_flat)
        return prob
```

```python
adversarial_loss = torch.nn.BCELoss()
generator = Generator(latent=opt.latent, img_shape=img_shape)
discriminator = Discriminator(img_shape=img_shape)
optimizer_G = torch.optim.Adam(generator.parameters(), lr=opt.learning_rate, betas=(opt.beta_1, opt.beta_2))
optimizer_D = torch.optim.Adam(discriminator.parameters(), lr=opt.learning_rate, betas=(opt.beta_1, opt.beta_2))

for epoch in range(opt.n_epochs):
    for i, (inputs, _) in enumerate(dataloader):
        inputs = inputs.to(device)

        # create the labels for the fake and real images
        real = torch.ones(inputs.size(0), requires_grad=False)
        fake = torch.zeros(inputs.size(0), requires_grad=False)
        real, fake = real.to(device), fake.to(device)

        # train the generator
        optimizer_G.zero_grad()
        z = torch.FloatTensor(np.random.normal(0, 1, (inputs.shape[0], opt.latent))).to(device)
        generated_images = generator(z, img_shape)

        # measure the generator loss and do backpropagation
        g_loss = adversarial_loss(discriminator(generated_images), real)
        g_loss.backward()
        optimizer_G.step()

        # train the discriminator
        optimizer_D.zero_grad()
        real_loss = adversarial_loss(discriminator(inputs), real)
        fake_loss = adversarial_loss(discriminator(generated_images.detach()), fake)
        d_loss = (real_loss + fake_loss) / 2

        d_loss.backward()
        optimizer_D.step()
```

Generating Digits



https://github.com/TheRevanchist/Generative_Adversarial_Networks/tree/master/gan

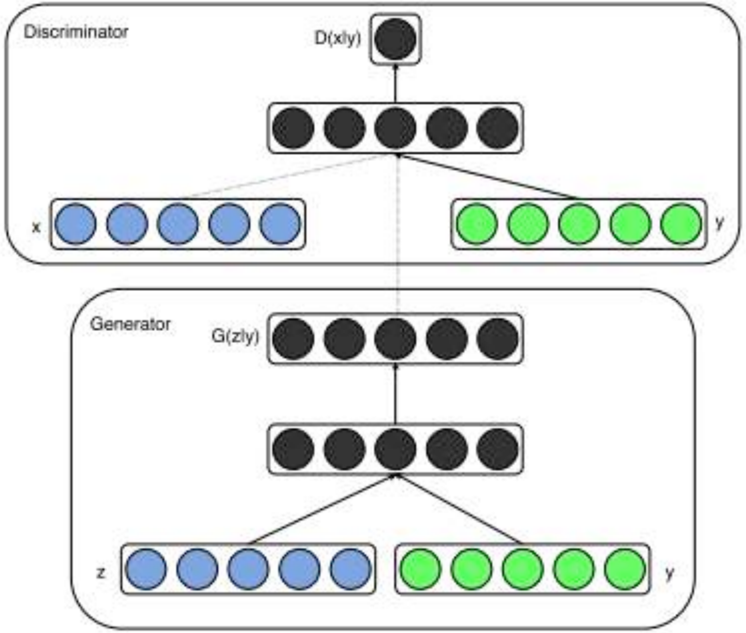
Conditional GANs

What if we want to generate only images of one particular class.

Idea: Give the labels of the classes (in one-hot format) to both the generator and discriminator.

For the generator concatenate the noise coming from latent space with the one hot vector. Similarly, the discriminator receives in input both the image and its label.

Conditional GANs



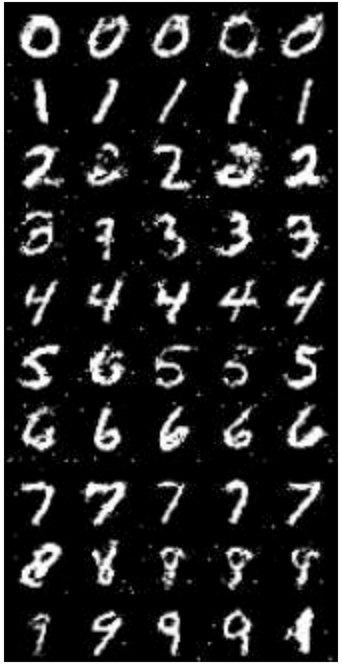
Mirza and Osindero, Conditional Generative Adversarial Networks, NIPS 2014

```python
class Generator(nn.Module):
    def __init__(self, latent, n_classes, img_shape):
        super(Generator, self).__init__()

        self.model = nn.Sequential(
            nn.Linear(latent + n_classes, 128),
            nn.ReLU(),
            nn.Linear(128, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, 1024),
            nn.ReLU(),
            nn.Linear(1024, int(np.prod(img_shape))),
            nn.Tanh()
        )

    def forward(self, z, labels, img_shape):
        image_and_label = torch.cat((z, labels), dim=1)
        img = self.model(image_and_label)
        img = img.view(img.size(0), *img_shape)
        return img
```

```python
class Discriminator(nn.Module):
    def __init__(self, n_classes, img_shape):
        super(Discriminator, self).__init__()

        self.model = nn.Sequential(
            nn.Linear(int(np.prod(img_shape) + n_classes), 512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, 1),
            nn.Sigmoid()
        )

    def forward(self, img, labels):
        img_flat = img.view(img.size(0), -1)
        image_and_label = torch.cat((img_flat, labels), dim=1)
        prob = self.model(image_and_label)
        return prob
```

```python
adversarial_loss = torch.nn.BCELoss()
generator = Generator(latent=opt.latent, n_classes=opt.n_classes, img_shape=img_shape)
discriminator = Discriminator(n_classes=opt.n_classes, img_shape=img_shape)
optimizer_G = torch.optim.Adam(generator.parameters(), lr=opt.learning_rate, betas=(opt.beta_1, opt.beta_2))
optimizer_D = torch.optim.Adam(discriminator.parameters(), lr=opt.learning_rate, betas=(opt.beta_1, opt.beta_2))

# start training
current_epoch = 0
for epoch in range(opt.n_epochs):
    for i, (inputs, labels) in enumerate(dataloader):
        inputs = inputs.to(device)
        labels = one_hot_embedding(labels, opt.n_classes).to(device)

        # create the labels for the fake and real images
        real = torch.ones(inputs.size(0), requires_grad=False)
        fake = torch.zeros(inputs.size(0), requires_grad=False)
        real, fake = real.to(device), fake.to(device)

        # train the generator
        optimizer_G.zero_grad()
        z = torch.FloatTensor(np.random.normal(0, 1, (inputs.shape[0], opt.latent))).to(device)
        generated_images = generator(z, labels, img_shape)

        # measure the generator loss and do backpropagation
        g_loss = adversarial_loss(discriminator(generated_images, labels), real)
        g_loss.backward()
        optimizer_G.step()

        # train the discriminator
        optimizer_D.zero_grad()
        real_loss = adversarial_loss(discriminator(inputs, labels), real)
        fake_loss = adversarial_loss(discriminator(generated_images.detach(), labels), fake)
        d_loss = (real_loss + fake_loss) / 2

        d_loss.backward()
        optimizer_D.step()
```

Generating Digits





https://github.com/TheRevanchist/Generative_Adversarial_Networks/tree/master/cgan

Any idea how to improve GANs?

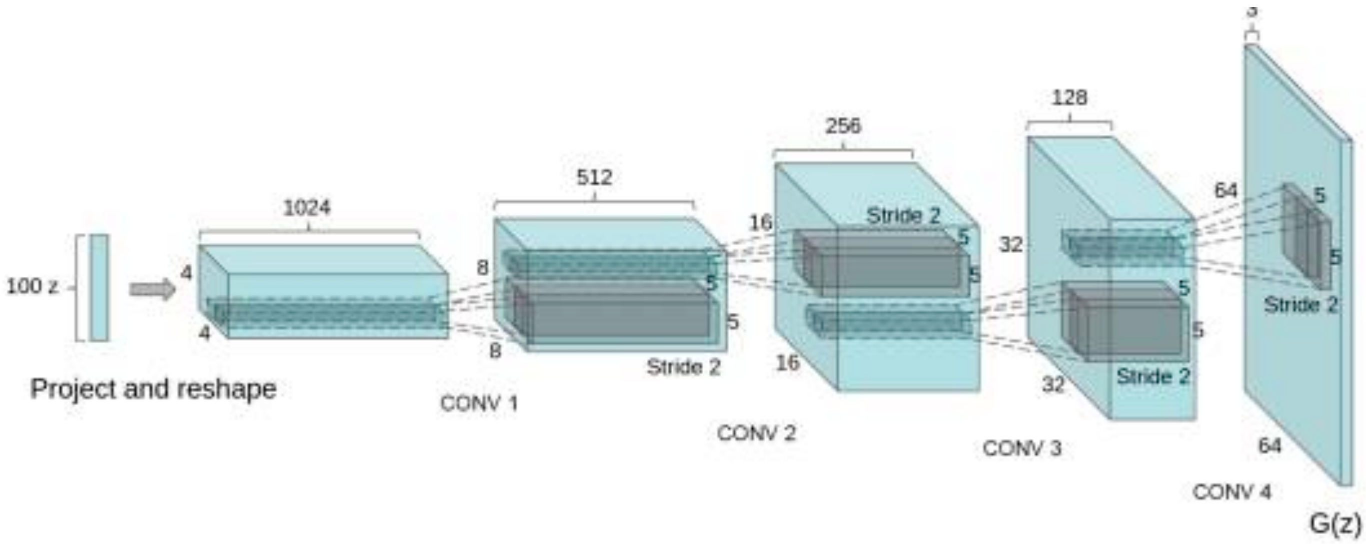
Deep Convolutional GANs (DCGAN)

Architecture guidelines for stable Deep Convolutional GANs

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batchnorm in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses Tanh.
- Use LeakyReLU activation in the discriminator for all layers.

Radford, Metz and Chintala, Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, ICLR 2016

Deep Convolutional GANs (DCGAN)



Radford, Metz and Chintala, Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, ICLR 2016

```python
class Generator(nn.Module):
    def __init__(self, latent, channels, num_filters):
        super(Generator, self).__init__()

        self.model = nn.Sequential(
            nn.ConvTranspose2d(latent, num_filters * 8, 4, 1, 0, bias=False),
            nn.ReLU(True),
            nn.BatchNorm2d(num_filters * 8),
            nn.ConvTranspose2d(num_filters * 8, num_filters * 4, 4, 2, 1, bias=False),
            nn.ReLU(True),
            nn.BatchNorm2d(num_filters * 4),
            nn.ConvTranspose2d(num_filters * 4, num_filters * 2, 4, 2, 1, bias=False),
            nn.ReLU(True),
            nn.BatchNorm2d(num_filters * 2),
            nn.ConvTranspose2d(num_filters * 2, num_filters, 4, 2, 1, bias=False),
            nn.ReLU(True),
            nn.BatchNorm2d(num_filters),
            nn.ConvTranspose2d(num_filters, channels, 4, 2, 1, bias=False),
            nn.Tanh()
        )

    def forward(self, z):
        img = self.model(z)
        return img
```

```python
class Generator(nn.Module):
    def __init__(self, latent, channels, num_filters):
        super(Generator, self).__init__()

        self.model = nn.Sequential(
            nn.ConvTranspose2d(latent, num_filters * 8, 4, 1, 0, bias=False),
            nn.ReLU(True),
            nn.BatchNorm2d(num_filters * 8),
            nn.ConvTranspose2d(num_filters * 8, num_filters * 4, 4, 2, 1, bias=False),
            nn.ReLU(True),
            nn.BatchNorm2d(num_filters * 4),
            nn.ConvTranspose2d(num_filters * 4, num_filters * 2, 4, 2, 1, bias=False),
            nn.ReLU(True),
            nn.BatchNorm2d(num_filters * 2),
            nn.ConvTranspose2d(num_filters * 2, num_filters, 4, 2, 1, bias=False),
            nn.ReLU(True),
            nn.BatchNorm2d(num_filters),
            nn.ConvTranspose2d(num_filters, channels, 4, 2, 1, bias=False),
            nn.Tanh()
        )

    def forward(self, z):
        img = self.model(z)
        return img
```

```python
class Discriminator(nn.Module):
    def __init__(self, channels, num_filters):
        super(Discriminator, self).__init__()

        self.model = nn.Sequential(
            nn.Conv2d(channels, num_filters, 4, 2, 1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.BatchNorm2d(num_filters),
            nn.Conv2d(num_filters, num_filters * 2, 4, 2, 1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.BatchNorm2d(num_filters * 2),
            nn.Conv2d(num_filters * 2, num_filters * 4, 4, 2, 1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.BatchNorm2d(num_filters * 4),
            nn.Conv2d(num_filters * 4, num_filters * 8, 4, 2, 1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.BatchNorm2d(num_filters * 8),
            nn.Conv2d(num_filters * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, img):
        prob = self.model(img)
        return prob.view(-1, 1).squeeze(1)
```

```python
class Generator(nn.Module):
    def __init__(self, latent, channels, num_filters):
        super(Generator, self).__init__()

        self.model = nn.Sequential(
            nn.ConvTranspose2d(latent, num_filters * 8, 4, 1, 0, bias=False),
            nn.ReLU(True),
            nn.BatchNorm2d(num_filters * 8),
            nn.ConvTranspose2d(num_filters * 8, num_filters * 4, 4, 2, 1, bias=False),
            nn.ReLU(True),
            nn.BatchNorm2d(num_filters * 4),
            nn.ConvTranspose2d(num_filters * 4, num_filters * 2, 4, 2, 1, bias=False),
            nn.ReLU(True),
            nn.BatchNorm2d(num_filters * 2),
            nn.ConvTranspose2d(num_filters * 2, num_filters, 4, 2, 1, bias=False),
            nn.ReLU(True),
            nn.BatchNorm2d(num_filters),
            nn.ConvTranspose2d(num_filters, channels, 4, 2, 1, bias=False),
            nn.Tanh()
        )

    def forward(self, z):
        img = self.model(z)
        return img
```

```python
class Discriminator(nn.Module):
    def __init__(self, channels, num_filters):
        super(Discriminator, self).__init__()

        self.model = nn.Sequential(
            nn.Conv2d(channels, num_filters, 4, 2, 1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.BatchNorm2d(num_filters),
            nn.Conv2d(num_filters, num_filters * 2, 4, 2, 1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.BatchNorm2d(num_filters * 2),
            nn.Conv2d(num_filters * 2, num_filters * 4, 4, 2, 1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.BatchNorm2d(num_filters * 4),
            nn.Conv2d(num_filters * 4, num_filters * 8, 4, 2, 1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.BatchNorm2d(num_filters * 8),
            nn.Conv2d(num_filters * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, img):
        prob = self.model(img)
        return prob.view(-1, 1).squeeze(1)
```

```python
# create the objects for loss function, two networks and for the two optimizers
if opt.loss == 'cross-entropy':
    adversarial_loss = torch.nn.BCELoss()
else:
    adversarial_loss = torch.nn.MSELoss()

generator = Generator(latent=opt.latent, channels=opt.channels, num_filters=opt.num_filters)
discriminator = Discriminator(channels=opt.channels, num_filters=opt.num_filters)
optimizer_G = torch.optim.Adam(generator.parameters(), lr=opt.learning_rate, betas=(opt.beta_1, opt.beta_2))
optimizer_D = torch.optim.Adam(discriminator.parameters(), lr=opt.learning_rate, betas=(opt.beta_1, opt.beta_2))

# put the nets on gpu
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
generator, discriminator = generator.to(device), discriminator.to(device)
generator.apply(weights_init)
discriminator.apply(weights_init)
```

```python
for epoch in range(opt.n_epochs):
    for i, (inputs, _) in enumerate(dataloader):
        inputs = inputs.to(device)

        # create the labels for the fake and real images
        real = torch.ones(inputs.size(0), requires_grad=False)
        fake = torch.zeros(inputs.size(0), requires_grad=False)
        real, fake = real.to(device), fake.to(device)

        # train the generator
        optimizer_G.zero_grad()
        z = torch.FloatTensor(np.random.normal(0, 1, (inputs.shape[0], opt.latent, 1, 1))).to(device)
        generated_images = generator(z)

        # measure the generator loss and do backpropagation
        g_loss = adversarial_loss(discriminator(generated_images), real)
        g_loss.backward()
        optimizer_G.step()

        # train the discriminator
        optimizer_D.zero_grad()
        real_loss = adversarial_loss(discriminator(inputs), real)
        fake_loss = adversarial_loss(discriminator(generated_images.detach()), fake)
        d_loss = (real_loss + fake_loss) / 2

        d_loss.backward()
        optimizer_D.step()
```

Deep Convolutional GANs (DCGAN)

Radford, Metz and Chintala, Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, ICLR 2016

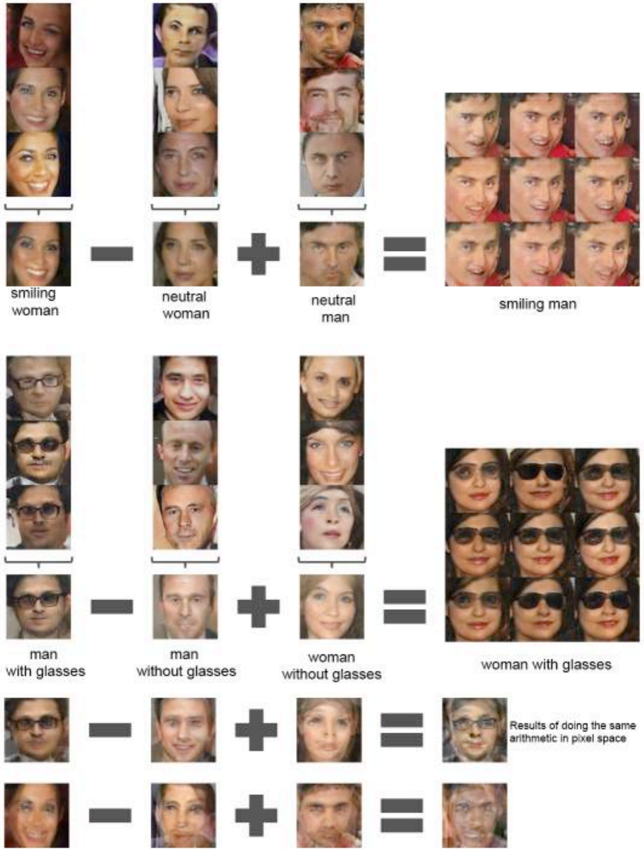


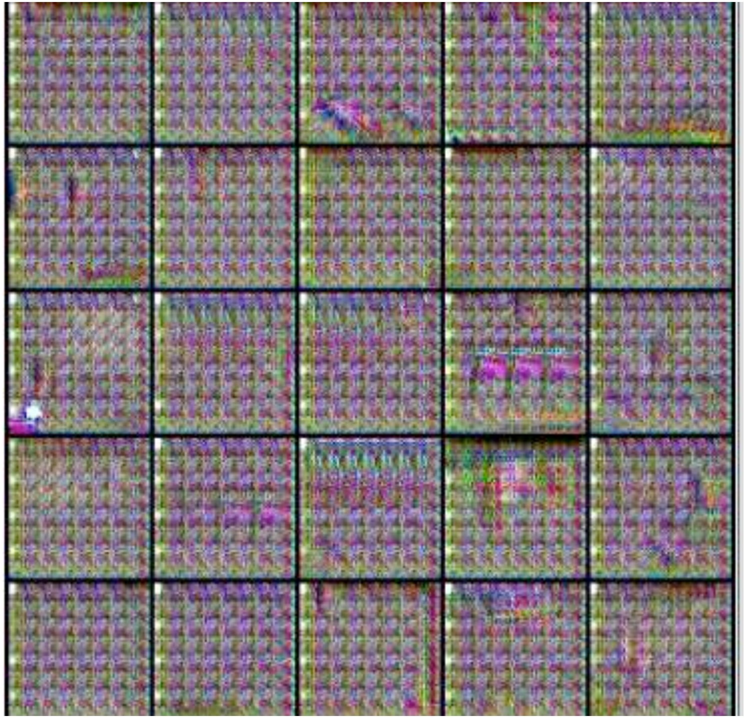
Figure 7: Vector arithmetic for visual concepts. For each column, the Z vectors of samples are averaged. Arithmetic was then performed on the mean vectors creating a new vector Y. The center sample on the right hand side is produce by feeding Y as input to the generator. To demonstrate the interpolation capabilities of the generator, uniform noise sampled with scale +-0.25 was added to Y to produce the 8 other samples. Applying arithmetic in the input space (bottom two examples) results in noisy overlap due to misalignment.

Figure 8: A "turn" vector was created from four averaged samples of faces looking left vs looking right. By adding interpolations along this axis to random samples we were able to reliably transform their pose.

Radford, Metz and Chintala, Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, ICLR 2016

However, During Training

Mode Collapse



https://github.com/TheRevanchist/Generative_Adversarial_Networks/tree/master/dcgan

Possible Fixes to Mode Collapse

- (Not scientific) Soft labeling, instead of giving to the discriminator labels 1/0, give to it 0.8/0.2
- (Definitely not scientific) Checkpoint the net, and every time mode collapse occurs, load the net from the previous checkpoint.
- (A bit more scientific) LSGAN, other types of cost functions.
- (Scientific) Wasserstein GAN
- (Even more scientific) Improved Wasserstein GAN, Dirac Gan etc

The GAN Zoo

GAN - Generative Adversarial Networks
3D-GAN - Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling
AC-GAN - Conditional Image Synthesis With Auxiliary Classifier GANs
AdaGAN - AdaGAN: Boosting Generative Models
AffGAN - Amortised MAP Inference for Image Super-resolution
AL-CGAN - Learning to Generate Images of Outdoor Scenes from Attributes and Semantic Layouts
ALI - Adversarially Learned Inference
AMGAN - Generative Adversarial Nets with Labeled Data by Activation Maximization
AnoGAN - Unsupervised Anomaly Detection with Generative Adversarial Networks to Guide Marker Discovery
ArtGAN - ArtGAN: Artwork Synthesis with Conditional Categorial GANs
b-GAN - b-GAN: Unified Framework of Generative Adversarial Networks
Bayesian GAN - Deep and Hierarchical Implicit Models
BEGAN - BEGAN: Boundary Equilibrium Generative Adversarial Networks
BiGAN - Adversarial Feature Learning
BS-GAN - Boundary-Seeking Generative Adversarial Networks
CGAN - Conditional Generative Adversarial Nets
CCGAN - Semi-Supervised Learning with Context-Conditional Generative Adversarial Networks
CatGAN - Unsupervised and Semi-supervised Learning with Categorical Generative Adversarial Networks
CoGAN - Coupled Generative Adversarial Networks
Context-RNN-GAN - Contextual RNN-GANs for Abstract Reasoning Diagram Generation
C-RNN-GAN - C-RNN-GAN: Continuous recurrent neural networks with adversarial training
CVAE-GAN - CVAE-GAN: Fine-Grained Image Generation through Asymmetric Training
CycleGAN - Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks
DTN—Unsupervised Cross-Domain Image Generation
DCGAN - Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks
DiscoGAN - Learning to Discover Cross-Domain Relations with Generative Adversarial Networks
DR-GAN - Disentangled Representation Learning GAN for Pose-Invariant Face Recognition
DualGAN - DualGAN: Unsupervised Dual Learning for Image-to-Image Translation
EBGAN - Energy-based Generative Adversarial Network
f-GAN - f-GAN: Training Generative Neural Samplers using Variational Divergence Minimization
FF-GAN - Towards Large-Pose Face Frontalization in the Wild
GAWWN - Learning What and Where to Draw
GoGAN - Gang of GANs: Generative Adversarial Networks with Maximum Margin Ranking
GP-GAN - GP-GAN: Towards Realistic High-Resolution Image Blending
IAN - Neural Photo Editing with Introspective Adversarial Networks
iGAN - Generative Visual Manipulation on the Natural Image Manifold
IcGAN - Invertible Conditional GANs for image editing
ID-CGAN - Image De-raining Using a Conditional Generative Adversarial Network
Improved GAN - Improved Techniques for Training GANs
InfoGAN - InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets
LAPGAN - Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks
LR-GAN - LR-GAN: Layered Recursive Generative Adversarial Networks for Image Generation

LSGAN - Least Squares Generative Adversarial Networks
LS-GAN - Loss-Sensitive Generative Adversarial Networks on Lipschitz Densities
MGAN - Precomputed Real-Time Texture Synthesis with Markovian Generative Adversarial Networks
MAGAN - MAGAN: Margin Adaptation for Generative Adversarial Networks
MAD-GAN - Multi-Agent Diverse Generative Adversarial Networks
MalGAN - Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN
MARTA-GAN - Deep Unsupervised Representation Learning for Remote Sensing Images
McGAN - McGan: Mean and Covariance Feature Matching GAN
MedGAN - Generating Multi-label Discrete Electronic Health Records using Generative Adversarial Networks
MIX+GAN - Generalization and Equilibrium in Generative Adversarial Nets (GANs)
MPM-GAN - Message Passing Multi-Agent GANs
MV-BiGAN - Multi-view Generative Adversarial Networks
pix2pix - Image-to-Image Translation with Conditional Adversarial Networks
PPGN - Plug & Play Generative Networks: Conditional Iterative Generation of Images in Latent Space
PrGAN - 3D Shape Induction from 2D Views of Multiple Objects
RenderGAN - RenderGAN: Generating Realistic Labeled Data
RTT-GAN - Recurrent Topic-Transition GAN for Visual Paragraph Generation
SGAN - Stacked Generative Adversarial Networks
SGAN - Texture Synthesis with Spatial Generative Adversarial Networks
SAD-GAN - SAD-GAN: Synthetic Autonomous Driving using Generative Adversarial Networks
SalGAN - SalGAN: Visual Saliency Prediction with Generative Adversarial Networks
SEGAN - SEGAN: Speech Enhancement Generative Adversarial Network
SeGAN - SeGAN: Segmenting and Generating the Invisible
SeqGAN - SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient
SketchGAN - Adversarial Training For Sketch Retrieval
SL-GAN - Semi-Latent GAN: Learning to generate and modify facial images from attributes
Softmax-GAN - Softmax GAN
SRGAN - Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network
S^2GAN - Generative Image Modeling using Style and Structure Adversarial Networks
SSL-GAN - Semi-Supervised Learning with Context-Conditional Generative Adversarial Networks
StackGAN - StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks
TGAN - Temporal Generative Adversarial Nets
TAC-GAN - TAC-GAN - Text Conditioned Auxiliary Classifier Generative Adversarial Network
TP-GAN - Beyond Face Rotation: Global and Local Perception GAN for Photorealistic and Identity Preserving Frontal View Synthesis
Triple-GAN - Triple Generative Adversarial Nets
Unrolled GAN - Unrolled Generative Adversarial Networks
VGAN - Generating Videos with Scene Dynamics
VGAN - Generative Adversarial Networks as Variational Training of Energy Based Models
VAE-GAN - Autoencoding beyond pixels using a learned similarity metric
VariGAN - Multi-View Image Generation from a Single-View
ViGAN - Image Generation and Editing with Variational Info Generative Adversarial Networks
WGAN - Wasserstein GAN
WGAN-GP - Improved Training of Wasserstein GANs
WaterGAN - WaterGAN: Unsupervised Generative Network to Enable Real-time Color Correction of Monocular Underwater Images

https://github.com/hindupuravinash/the-gan-zoo

Does it Really Matter?!

Are GANs Created Equal? A Large-Scale Study

Mario Lucic* Karol Kurach* Marcin Michalski Olivier Bousquet Sylvain Gelly
Google Brain

Abstract

Generative adversarial networks (GAN) are a powerful subclass of generative models. Despite a very rich research activity leading to numerous interesting GAN algorithms, it is still very hard to assess which algorithm(s) perform better than others. We conduct a neutral, multi-faceted large-scale empirical study on state-of-the art models and evaluation measures. We find that most models can reach similar scores with enough hyperparameter optimization and random restarts. This suggests that improvements can arise from a higher computational budget and tuning more than fundamental algorithmic changes. To overcome some limitations of the current metrics, we also propose several data sets on which precision and recall can be computed. Our experimental results suggest that future GAN research should be based on more systematic and objective evaluation procedures. Finally, we did not find evidence that any of the tested algorithms consistently outperforms the non-saturating GAN introduced in [9].

Lucic et al, Are GANs Created Equal? A Large-Scale Study, NIPS 2018

Sample Generation

Training Data
(CelebA)

Sample Generator
(Karras et al, 2017)

Goodfellow, CVPR tutorial, 2018

3.5 Years of Progress on Faces

(Brundage et al, 2018)

Goodfellow, CPVP tutorial, 2018

<2 Years of Progress on ImageNet

Odena et al
2016

Miyato et al
2017

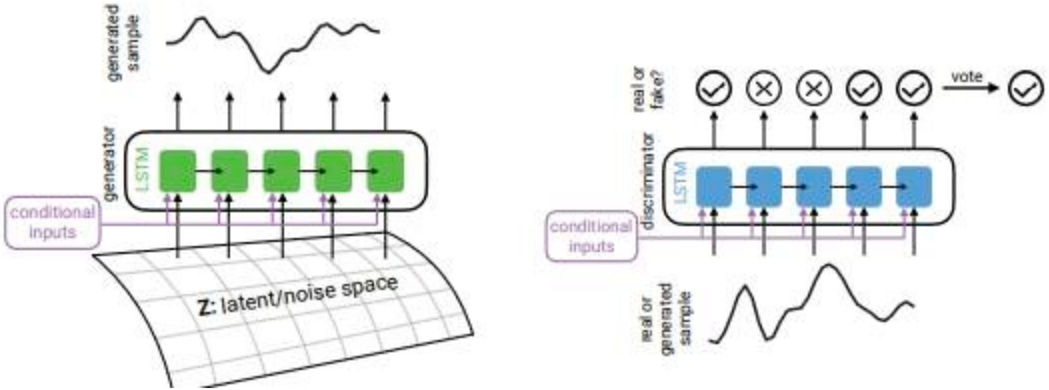
Zhang et al
2018

(Goodfellow 2018)

Goodfellow, CPVP tutorial, 2018

Goodfellow, CPVP tutorial, 2018

GANs for Time Series

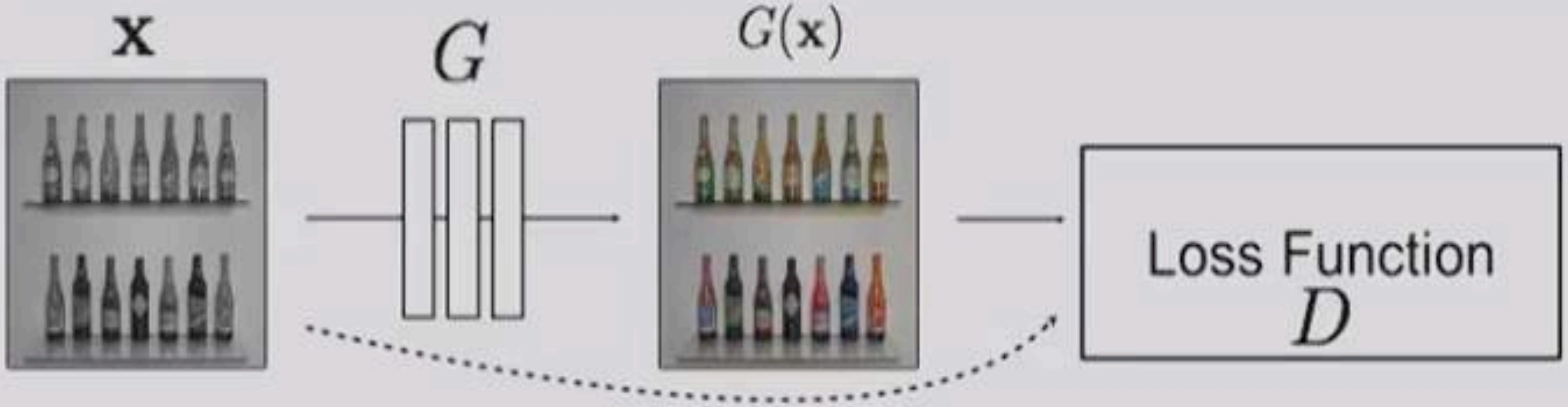


Hyland et al, Real-valued (medical) time series generation with recurrent conditional GANs, arXiv 2017

Reasons to dislike GANs

- They are a devil to train!
  - The discriminator nearly always wins
  - Sometimes, training longer makes it worse
  - Sometimes, more data doesn’t make it better
- Do they really generate a distribution?
- Generality penalty: for any given problem, application-tailored solutions might work better

Efros, ICCV tutorial, 2017

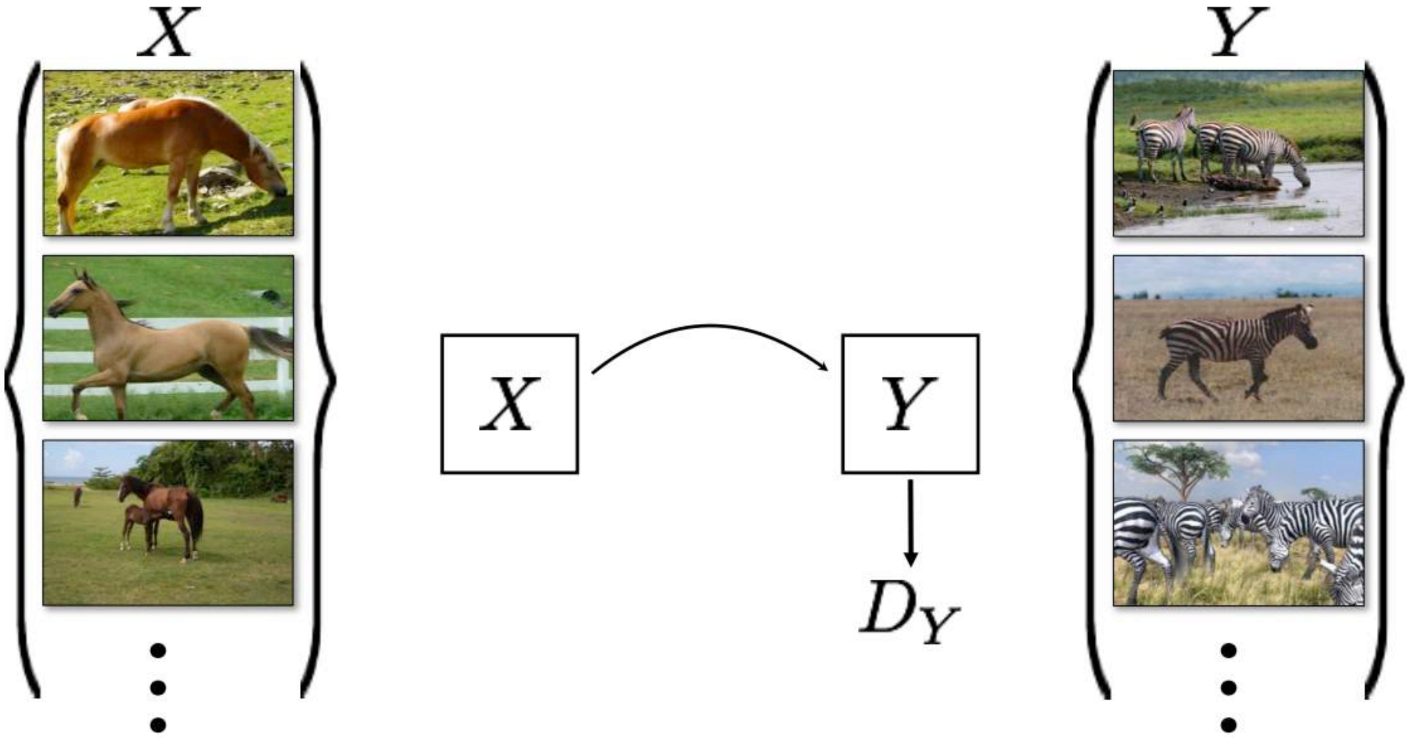


G’s perspective: **D** is a loss function.

Rather than being hand-designed, it is *learned*.

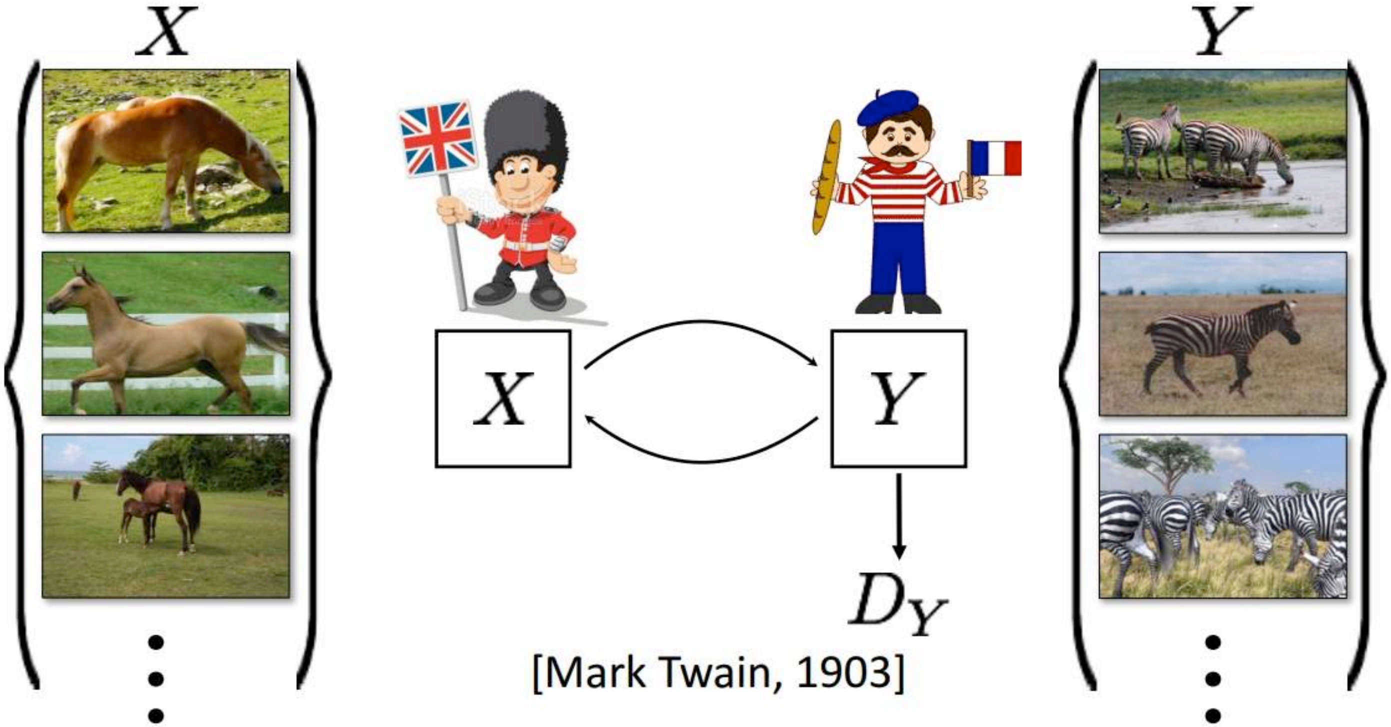
Efros, ICCV tutorial, 2017

Cycle-Consistent Adversarial Networks



[Zhu*, Park*, Isola, and Efros, ICCV 2017]

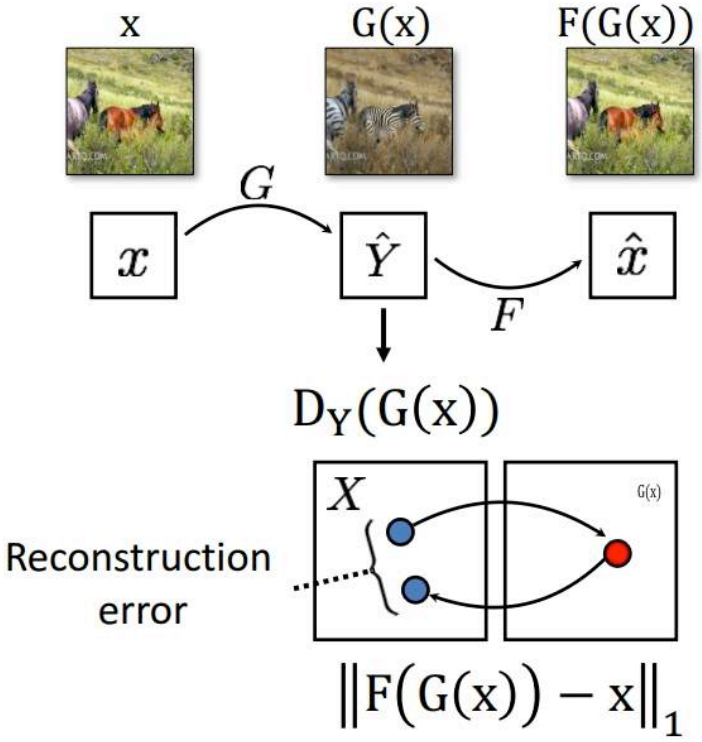
Cycle-Consistent Adversarial Networks



[Mark Twain, 1903]

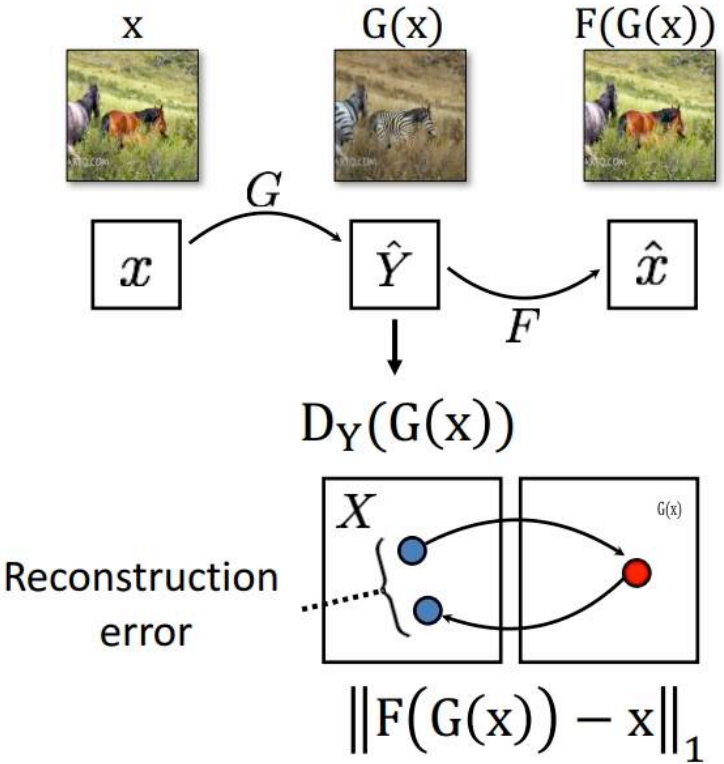
[Zhu*, Park*, Isola, and Efros, ICCV 2017]

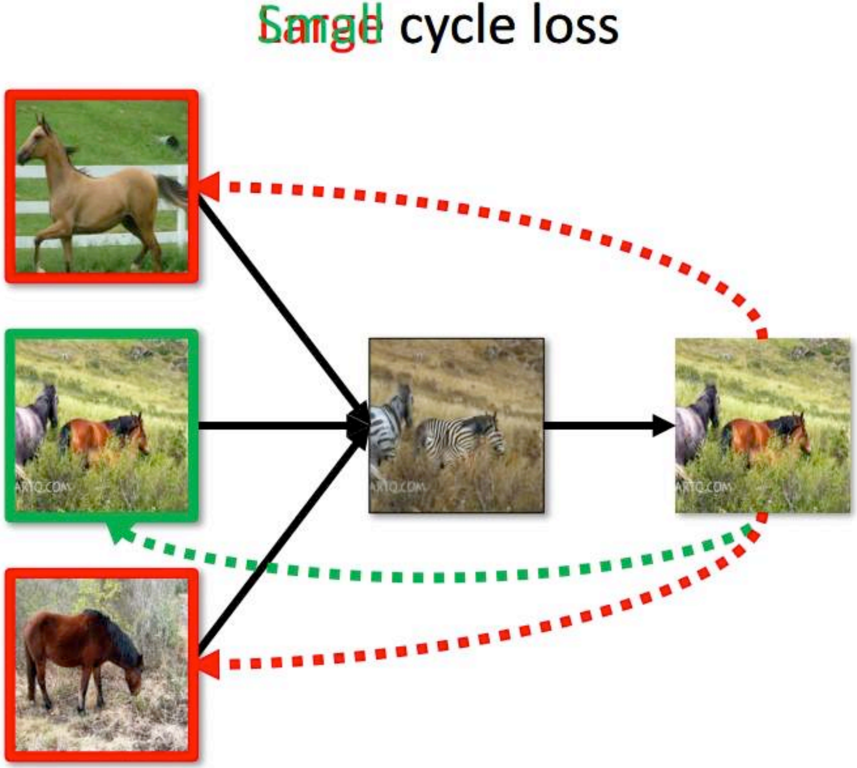
Cycle-Consistent Adversarial Networks



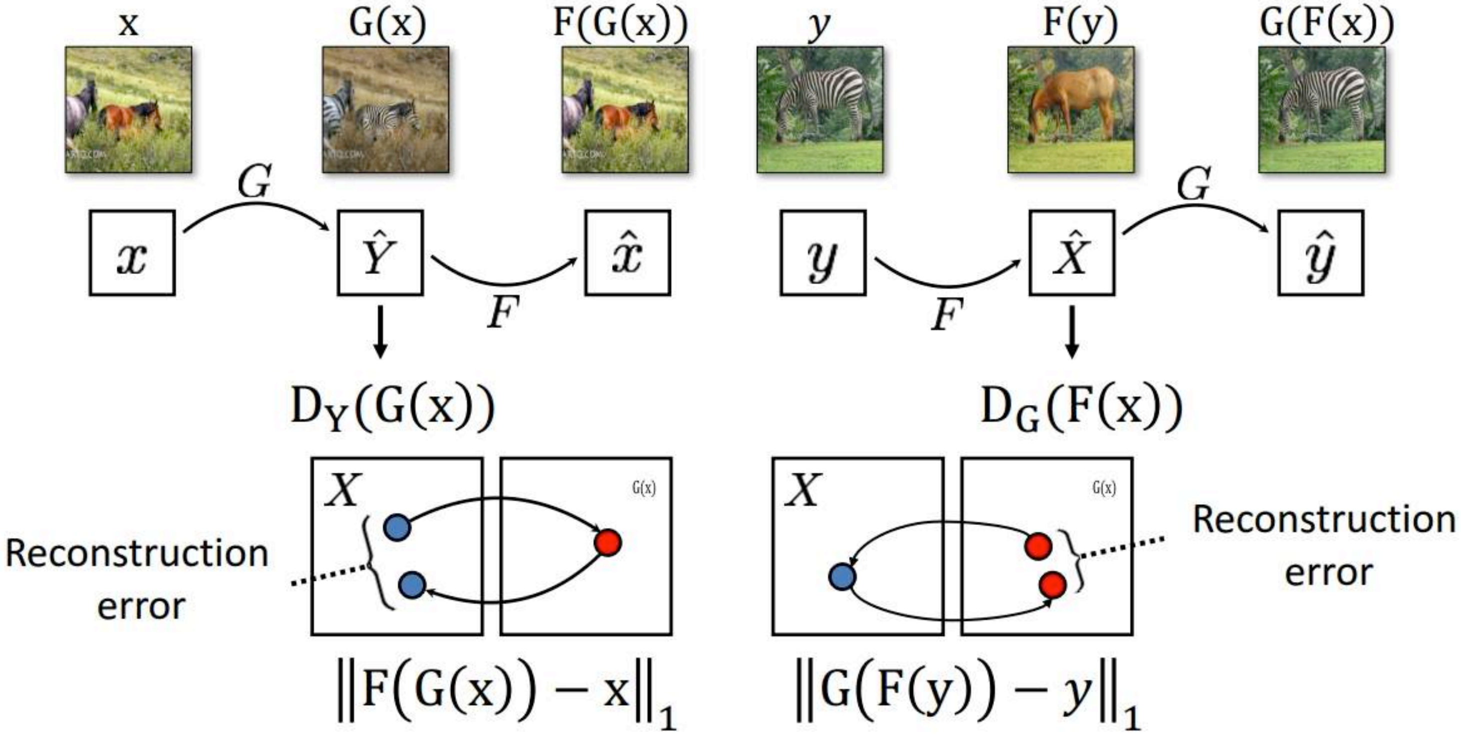
[Zhu*, Park*, Isola, and Efros, ICCV 2017]

Cycle Consistency Loss





Cycle Consistency Loss



Collection Style Transfer

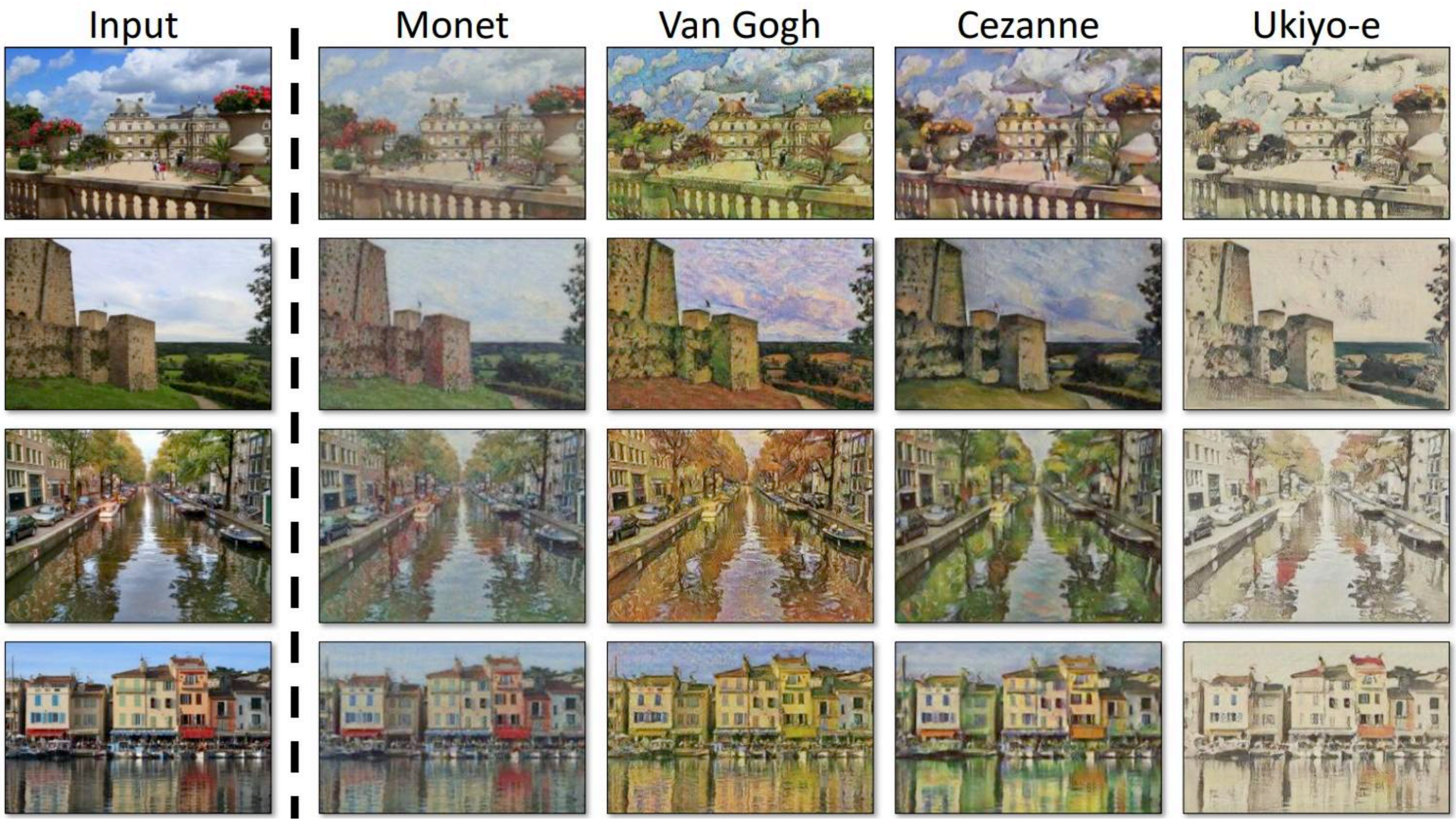
Photograph
@ Alexei Efros

Monet

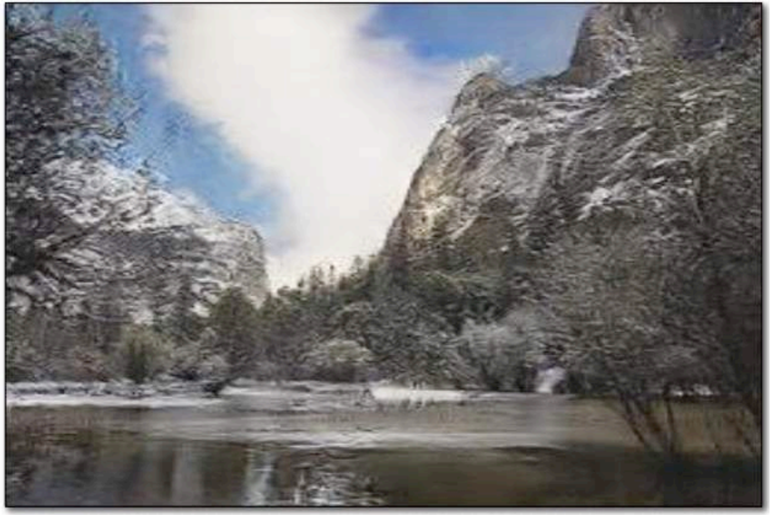
Van Gogh

Cezanne

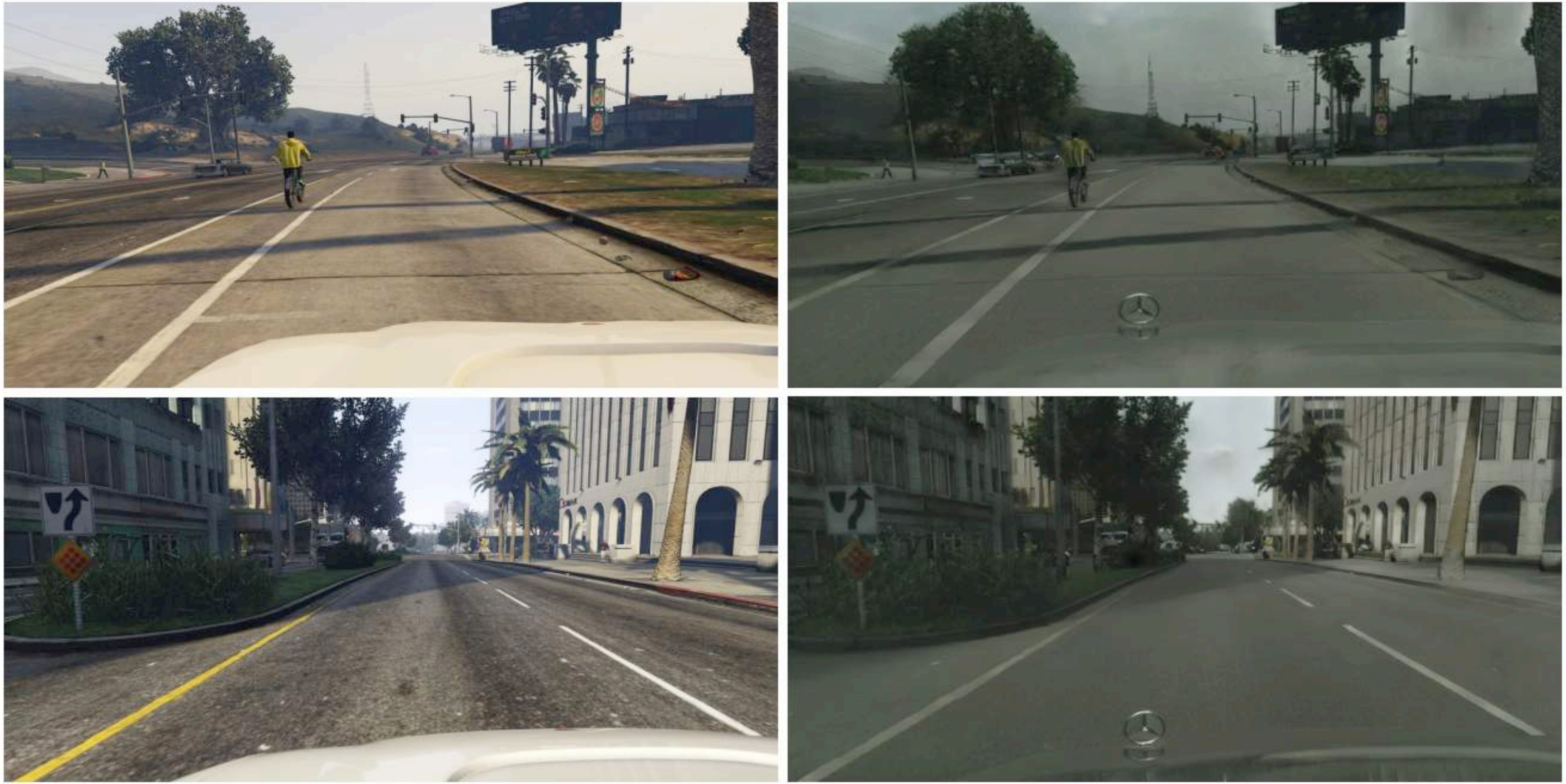
Ukiyo-e


Input
Monet
Van Gogh
Cezanne
Ukiyo-e

Monet's paintings → photos

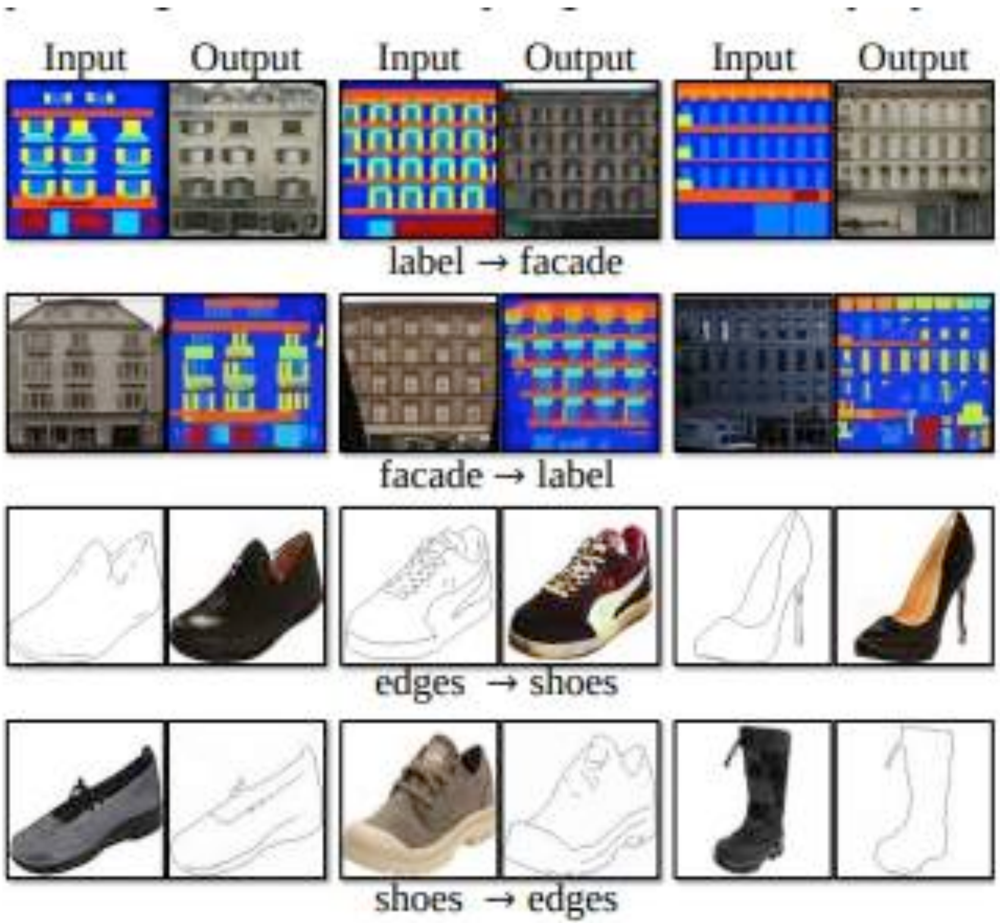



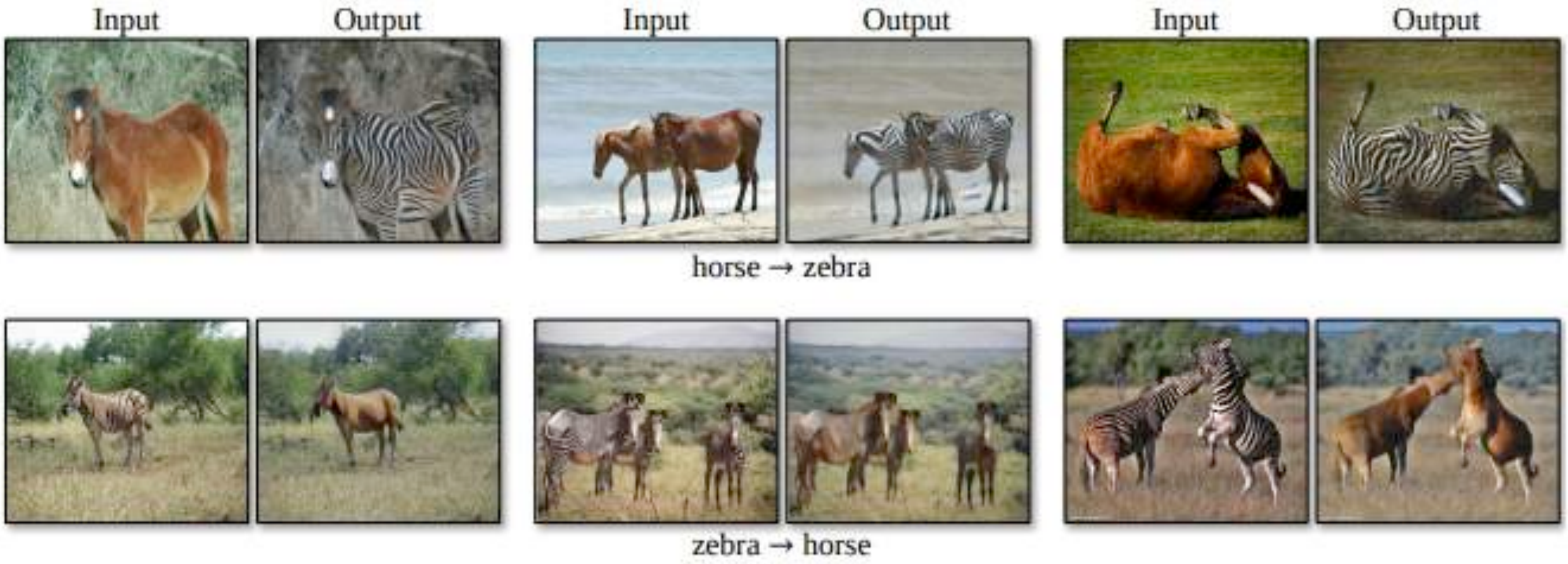
CG2Real: GTA5 → real streetview



Real2CG: real streetview → GTA

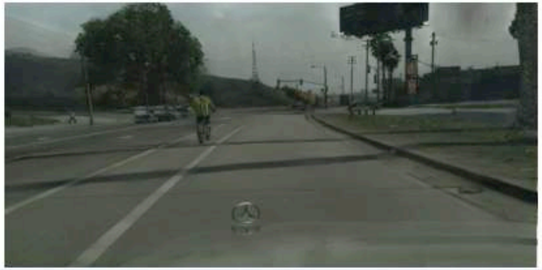
GTA5 images Segmentation labels


Input
Output
Input
Output
Input
Output
label → facade
facade → label
edges → shoes
shoes → edges

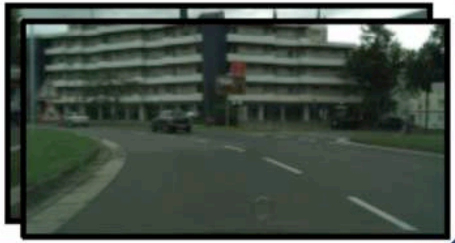

Input
Output
Input
Output
Input
Output
horse → zebra
zebra → horse

For much more look at: https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix

Domain Adaptation with CycleGAN



Train on CycleGAN data



Test on real images

	meanIOU	Per-pixel accuracy
Oracle (Train and test on Real)	60.3	93.1
Train on CG, test on Real	17.9	54.0
FCN in the wild [Previous STOA]	27.1	-
Train on CycleGAN, test on Real	**34.8**	**82.8**

My GAN-story


SOS
ABBA
arr. Andrew King
Soprano
Alto
Tenor
Bass
1. da da da da da
1. ah
2015


„AVE VERUM CORPUS“
MOTETTE
für 4 Singstimmen, 2 Violinen, Viola, Bass und Orgel
von
W. A. MOZART.
Köch. Verz. № 618.
Mozarts Werke.
Serie 3. № 31.
Adagio.
Violino I.
Violino II.
Viola.
Soprano.
Alto.
Tenore.
Basso.
Basso ed Organo.
W. A. M. 618.

Problems

1) Our images are 2000 x 2000. At 700 (+ delta) by 700 (+delta) images, even a VOLTA V100 runs out of memory

Problems

1) Our images are 2000 x 2000. At 700 by 700 images, even a VOLTA V100 runs out of memory

    - Solution 1: train in patches, generate large images.

Problems

1) Our images are 2000 x 2000. At 700 by 700 images, even a VOLTA V100 runs out of memory

    - Solution 1: train in patches, generate large images. It doesn’t work.

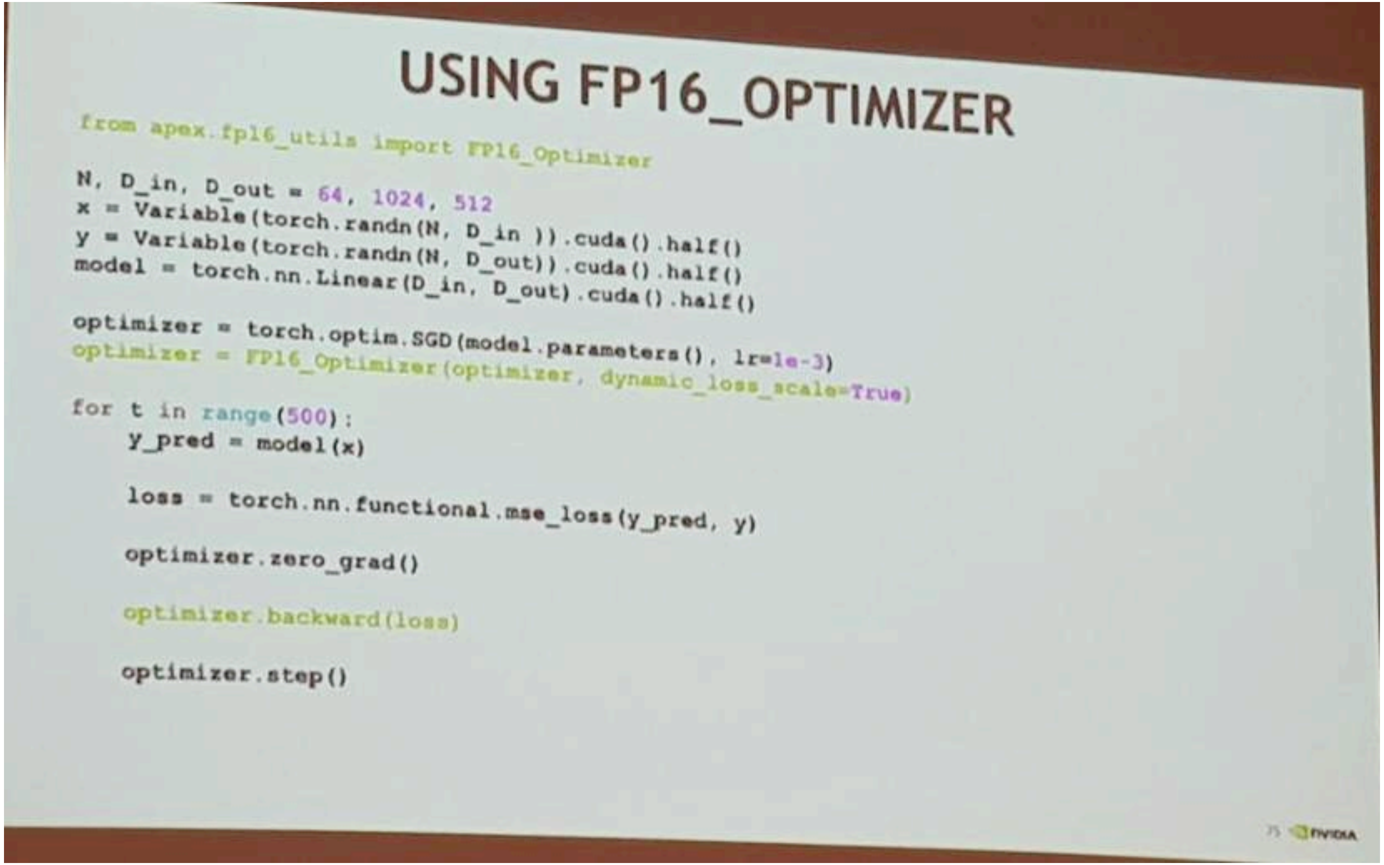
Problems

1) Our images are 2000 x 2000. At 700 by 700 images, even a VOLTA V100 runs out of memory
   - Solution 1: train in patches, generate large images. It doesn’t work.
   - Solution 2: make the nets more efficient. Train on float16 (NVIDIA Apex) and use gradient checkpointing.

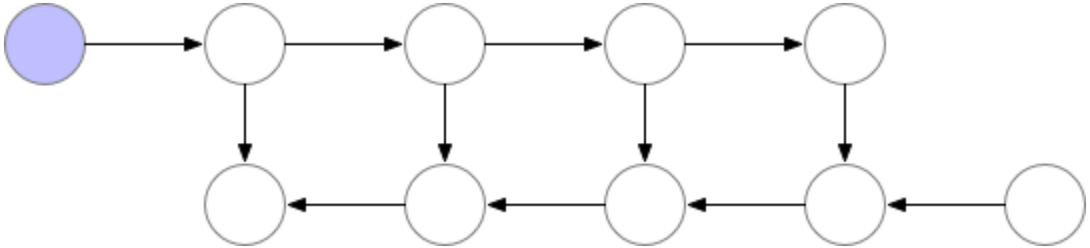
Problems

1) Our images are 2000 x 2000. At 700 by 700 images, even a VOLTA V100 runs out of memory
   - Solution 1: train in patches, generate large images. It doesn't work.
   - Solution 2: make the nets more efficient. Train on float16 (NVIDIA Apex) and use gradient checkpointing. It works.

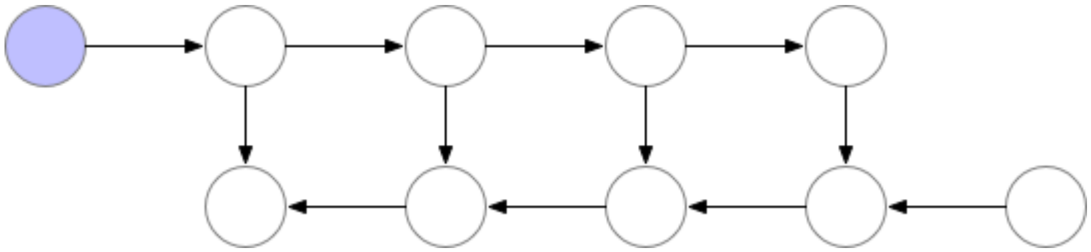
Digression: Half precision training



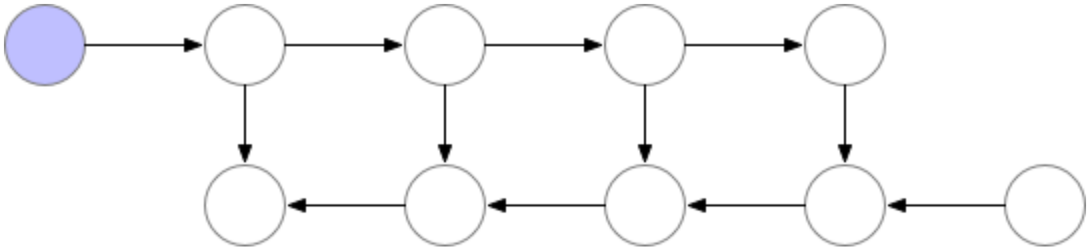
Digression: Gradient Checkpointing



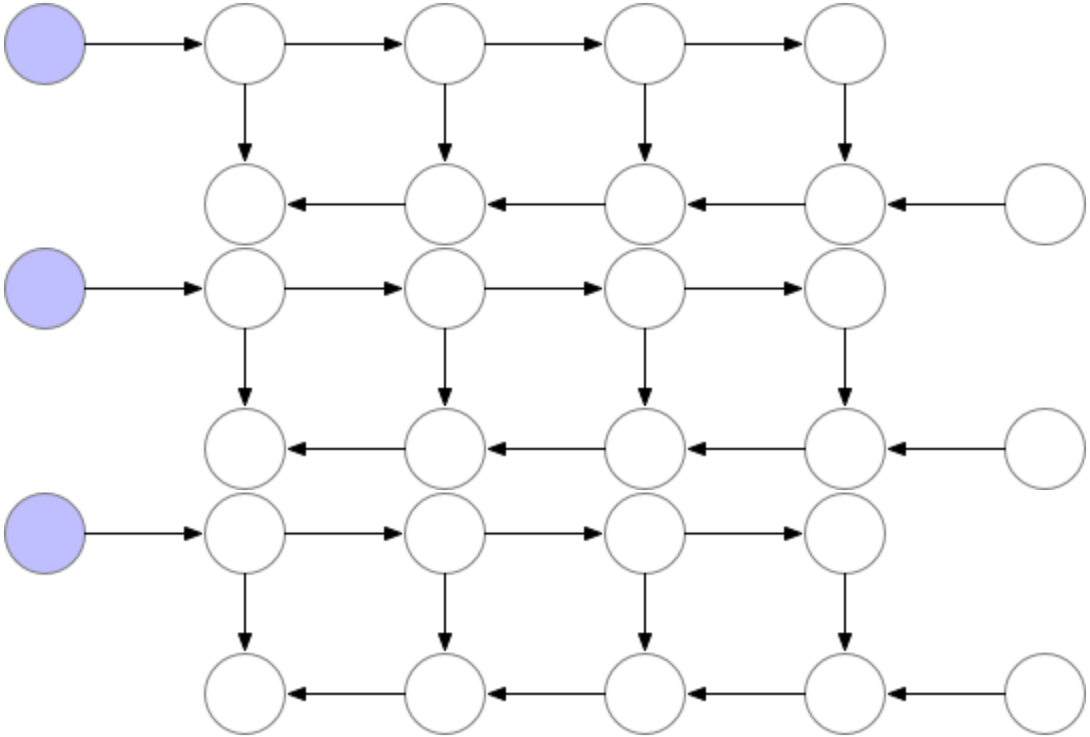
Digression: Gradient Checkpointing



Digression: Gradient Checkpointing



Digression: Gradient Checkpointing



https://github.com/TheRevanchist/pytorch-CycleGAN-and-pix2pix

Problems

1) Our images are 2000 x 2000. At 700 by 700 images, even a VOLTA V100 runs out of memory
   - Solution 1: train in patches, generate large images. It doesn't work.
   - Solution 2: make the nets more efficient. Train on float16 (NVIDIA Apex) and use gradient checkpointing. It works.
2) Bigger images, less likely that we will be able to generate meaningful images (mode collapse)

Problems

1) Our images are 2000 x 2000. At 700 by 700 images, even a VOLTA V100 runs out of memory
   - Solution 1: train in patches, generate large images. It doesn’t work.
   - Solution 2: make the nets more efficient. Train on float16 (NVIDIA Apex) and use gradient checkpointing. It works.
2) Bigger images, less likely that we will be able to generate meaningful images (mode collapse)
   - Solution 1: more careful training and hyperparameter optimization.
   - Solution 2: different loss functions, maybe Wasserstein GANs (or the improved version of it), researchy stuff.
   - Solution 3: progressive training and/or BigGan-inspired approach.

Thank You!

Christie's
@ChristiesInc

#AuctionUpdate The first AI artwork to be sold in a major auction achieves $432,500 after a bidding battle on the phones and via ChristiesLive bit.ly/2PVN2ly

2,362 4:22 PM - Oct 25, 2018

1,571 people are talking about this