Deep generative modeling: Implicit models

Jakub M. Tomczak
Deep Learning

VU VRIJE UNIVERSITEIT AMSTERDAM

TYPES OF GENERATIVE MODELS

Generative model

- **Autoregressive (*e.g.*, PixelCNN)**
- **Flow-based (e.g., RealNVP, GLOW)**
- **Latent variable models**
  - **Implicit models (*e.g.*, GANs)**
  - **Prescribed models (*e.g.*, VAE)**

GENERATIVE MODELS

	Training	Likelihood	Sampling	Compression
Autoregressive models (e.g., PixelCNN)	Stable	Exact	Slow	No
Flow-based models (e.g., RealNVP)	Stable	Exact	Fast/Slow	No
Implicit models (e.g., GANs)	Unstable	No	Fast	No
Prescribed models (e.g., VAEs)	Stable	Approximate	Fast	Yes

GENERATIVE MODELS

	Training	Likelihood	Sampling	Compression
Autoregressive models (e.g., PixelCNN)	Stable	Exact	Slow	No
Flow-based models (e.g., RealNVP)	Stable	Exact	Fast/Slow	No
Implicit models (e.g., GANs)	Unstable	No	Fast	No
Prescribed models (e.g., VAEs)	Stable	Approximate	Fast	Yes

DENSITY NETWORKS

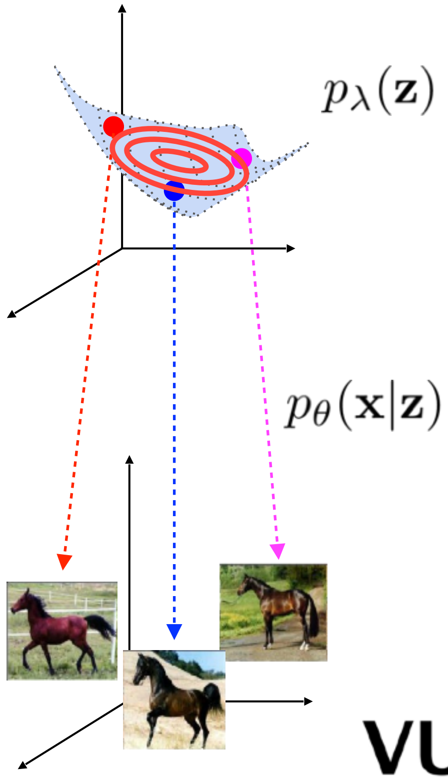
Generative process:

$$1.\ \mathbf{z} \sim p_{\lambda}(\mathbf{z})$$

$$2.\ \mathbf{x} \sim p_{\theta}(\mathbf{x} \mid \mathbf{z})$$

The log-likelihood function:

$$\log p(\mathbf{x}) = \log \int p_{\theta}(\mathbf{x} \mid \mathbf{z}) p_{\lambda}(\mathbf{z}) \mathrm{d}\mathbf{z}$$



Generative process:

$$1.\ \mathbf{z} \sim p_\lambda(\mathbf{z})$$

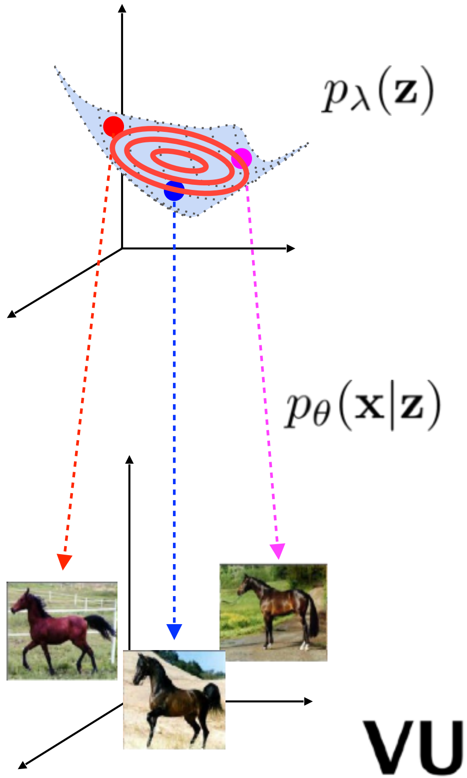
$$2.\ \mathbf{x} \sim p_\theta(\mathbf{x} \mid \mathbf{z})$$

The log-likelihood function:

$$\log p(\mathbf{x}) = \log \int p_\theta(\mathbf{x} \mid \mathbf{z}) p_\lambda(\mathbf{z}) \mathrm{d}\mathbf{z}$$

$$\approx \log \frac{1}{S} \sum_{s=1}^{S} \exp\left(\log p_\theta\left(\mathbf{x} \mid \mathbf{z}_s\right)\right)$$

It could be estimated by MC samples.



Generative process:

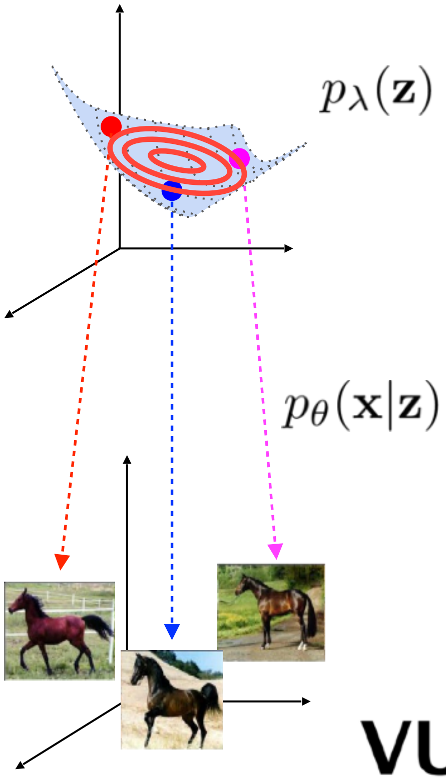
$$1.\ \mathbf{z} \sim p_{\lambda}(\mathbf{z})$$

$$2.\ \mathbf{x} \sim p_{\theta}(\mathbf{x} \mid \mathbf{z})$$

The log-likelihood function:

$$\log p(\mathbf{x}) = \log \int p_{\theta}(\mathbf{x} \mid \mathbf{z}) p_{\lambda}(\mathbf{z}) \mathrm{d}\mathbf{z}$$

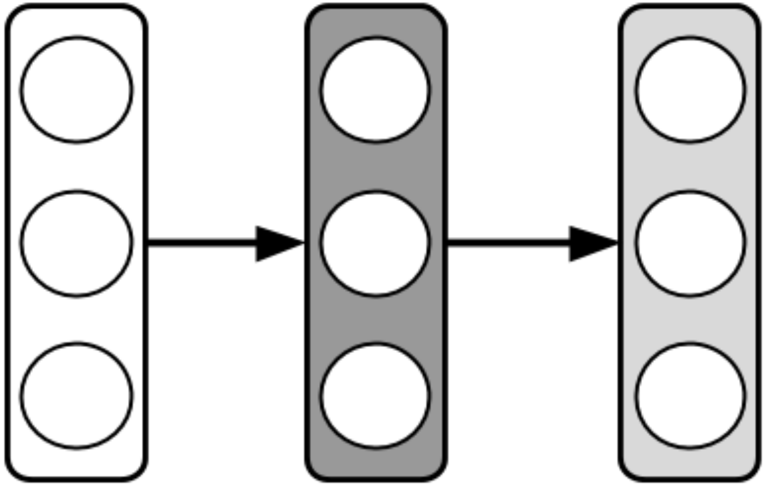
$$\approx \log \frac{1}{S} \sum_{s=1}^{S} \exp\left(\log p_{\theta}\left(\mathbf{x} \mid \mathbf{z}_s\right)\right)$$



It could be estimated by MC samples.

If we take standard Gaussian prior, we need to model p(xlz) only!

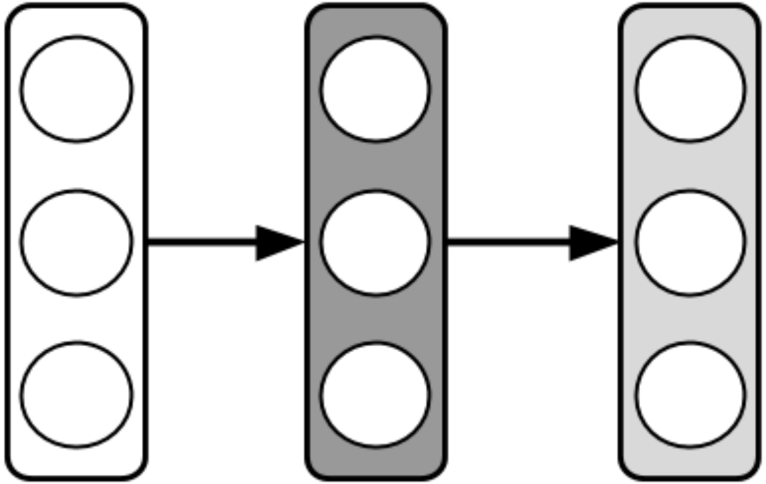
Let's consider a function *f* that transforms **z** to **x**.



$$\mathbf{z} \sim p_\lambda(\mathbf{z}) \qquad f(\mathbf{z}) \qquad \mathbf{x}$$

MacKay, D. J., & Gibbs, M. N. (1999). *Density networks*. Statistics and neural networks: advances at the interface. Oxford

Let's consider a function *f* that transforms **z** to **x**.

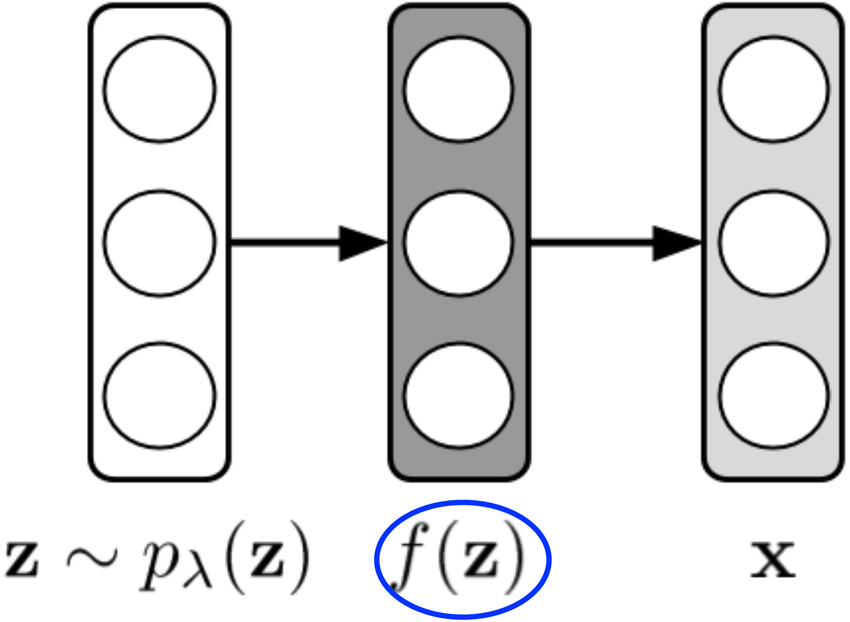


$\mathbf{z} \sim p_\lambda(\mathbf{z})$ $f(\mathbf{z})$ $\mathbf{x}$

It must be a powerful (=flexible) transformation!

MacKay, D. J., & Gibbs, M. N. (1999). *Density networks*. Statistics and neural networks: advances at the interface. Oxford

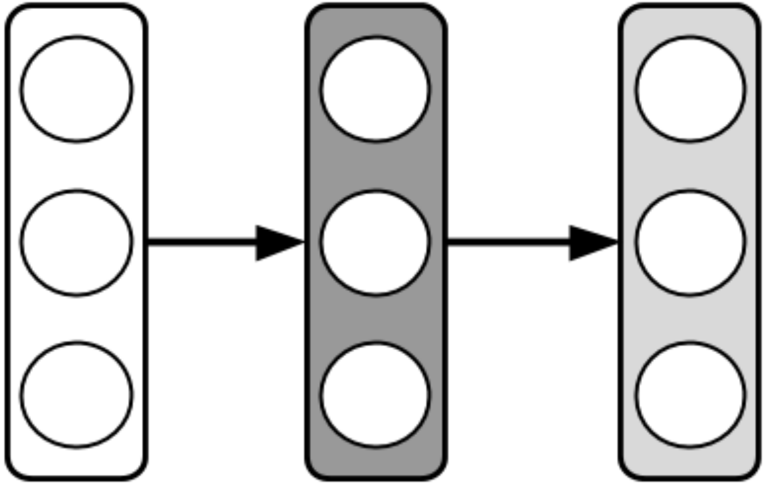
Let's consider a function *f* that transforms **z** to **x**.



It must be a powerful (=flexible) transformation!
NEURAL NETWORK 💪

MacKay, D. J., & Gibbs, M. N. (1999). *Density networks*. Statistics and neural networks: advances at the interface. Oxford

Let's consider a function *f* that transforms **z** to **x**.



$\mathbf{z} \sim p_\lambda(\mathbf{z})$ $\quad f(\mathbf{z})$ $\quad \mathbf{x}$

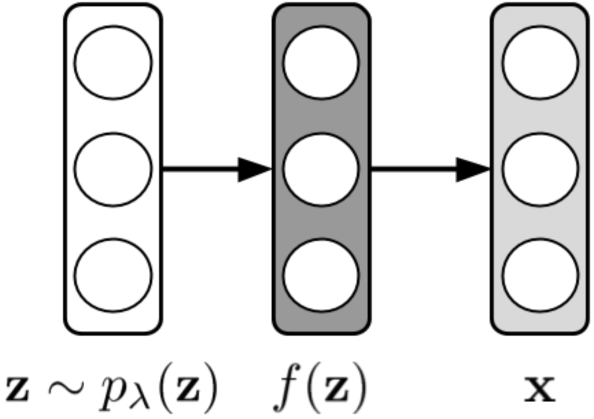
Neural network outputs parameters of a distribution, e.g., a mixture of Gaussians.

MacKay, D. J., & Gibbs, M. N. (1999). *Density networks*. Statistics and neural networks: advances at the interface. Oxford

The log-likelihood function:

$$\log p(\mathbf{x}) = \log \int p_\theta(\mathbf{x} \mid \mathbf{z}) p_\lambda(\mathbf{z}) \mathrm{d}\mathbf{z}$$

$$\approx \log \frac{1}{S} \sum_{s=1}^{S} \exp\left(\log p_\theta\left(\mathbf{x} \mid \mathbf{z}_s\right)\right)$$



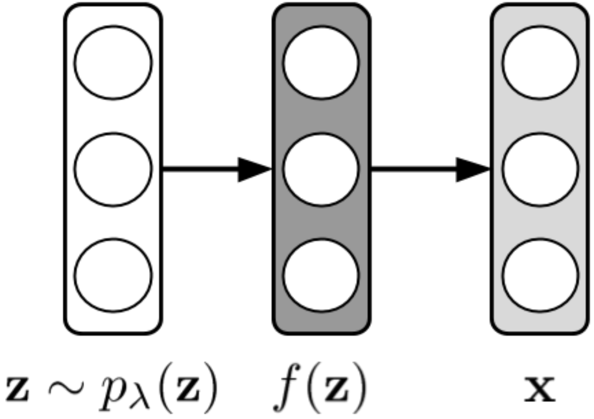
Training procedure:

1. Sample multiple **z**'s from the prior (e.g., standard Gaussian).

2. Apply log-sum-exp-trick, and apply backpropagation.

The log-likelihood function:

$$\log p(\mathbf{x}) = \log \int p_\theta(\mathbf{x} \mid \mathbf{z}) p_\lambda(\mathbf{z}) \mathrm{d}\mathbf{z}$$
$$\approx \log \frac{1}{S} \sum_{s=1}^{S} \exp \left(\log p_\theta \left(\mathbf{x} \mid \mathbf{z}_s \right) \right)$$



Training procedure:

1. Sample multiple **z**’s from the prior (e.g., standard Gaussian).

2. Apply log-sum-exp-trick, and apply backpropagation.

Drawback: It scales badly in high-dimensional cases…

Advantages

✓ Non-linear transformations.

✓ Allows to generate.

Disadvantages

- No analytical solutions.
- No exact likelihood.
- It requires a lot of samples from the prior.
- Fails in high-dim.
- It requires an explicit distribution (e.g., Gaussian).

Advantages

✓ Non-linear transformations.

✓ Allows to generate.

Disadvantages

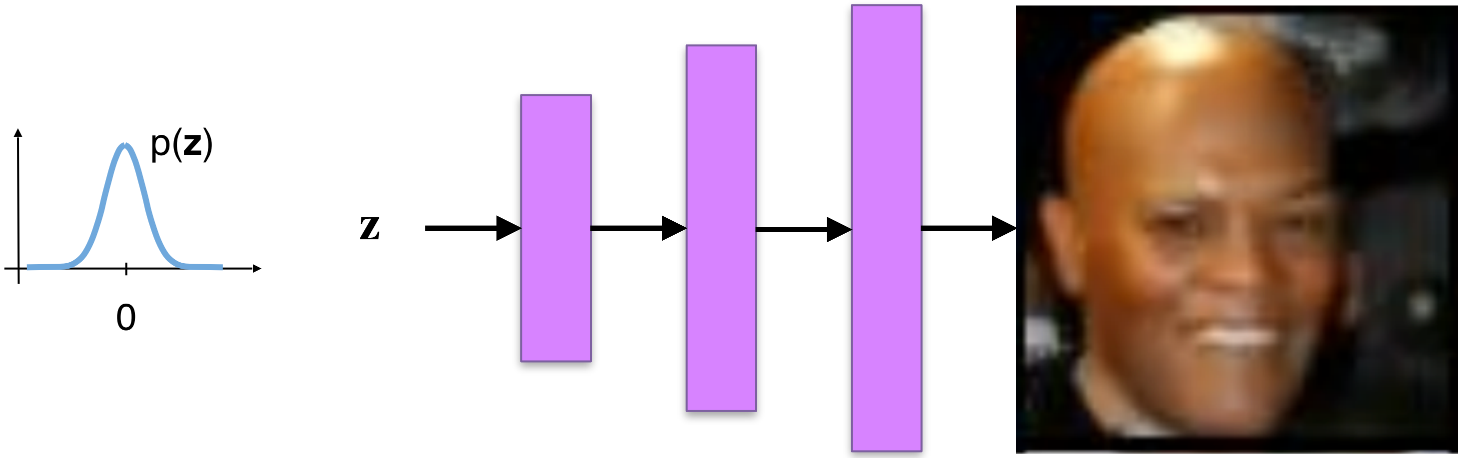
- No analytical solutions.
- No exact likelihood.
- It requires a lot of samples from the prior.
- Fails in high-dim.
- It requires an explicit distribution (e.g., Gaussian).

Can we do better?

IMPLICIT DISTRIBUTIONS

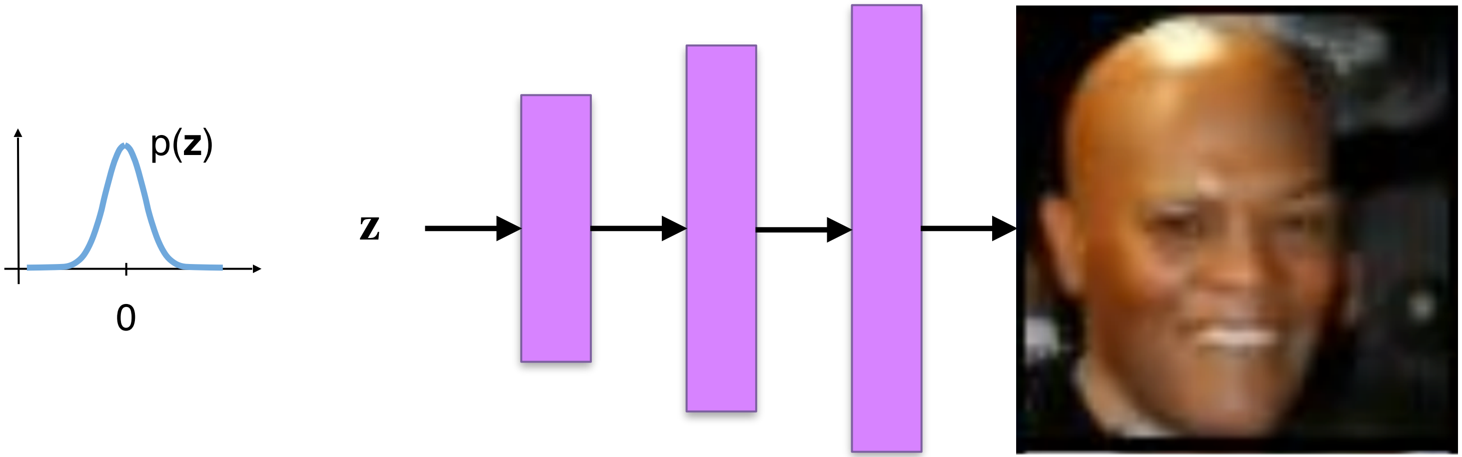
Let us look again at the Density Network model.

The idea is to inject noise to a neural network that serves as a **generator**:



Let us look again at the Density Network model.

The idea is to inject noise to a neural network that serves as a **generator**:



But now, we don't specify the distribution (e.g., MoG), but use a flexible transformation directly to return an image. This is now **implicit**.

THE IDEA BEHIND IMPLICIT DISTRIBUTIONS

We have a neural network (**generator**) that transforms noise into an image.

THE IDEA BEHIND IMPLICIT DISTRIBUTIONS

We have a neural network (**generator**) that transforms noise into an image.

It defines an **implicit distribution** (i.e., we do not assume any form of it), and it could be seen as Dirac's delta:

$$p(\mathbf{x}|\mathbf{z}) = \delta\left(\mathbf{x} - f(\mathbf{z})\right)$$

THE IDEA BEHIND IMPLICIT DISTRIBUTIONS

We have a neural network (**generator**) that transforms noise into an image.

It defines an **implicit distribution** (i.e., we do not assume any form of it), and it could be seen as Dirac's delta:

$$p(\mathbf{x}|\mathbf{z}) = \delta\left(\mathbf{x} - f(\mathbf{z})\right)$$

However, now we **cannot** use the likelihood-based approach, because $\ln \delta\left(\mathbf{x} - f(\mathbf{z})\right)$ is ill-defined.

We have a neural network (**generator**) that transforms noise into an image.

It defines an **implicit distribution** (i.e., we do not assume any form of it), and it could be seen as Dirac's delta:

$$p(\mathbf{x}|\mathbf{z}) = \delta\left(\mathbf{x} - f(\mathbf{z})\right)$$

However, now we **cannot** use the likelihood-based approach, because $\ln \delta\left(\mathbf{x} - f(\mathbf{z})\right)$ is ill-defined.

We need to use a different approach.

GENERATIVE ADVERSARIAL NETWORKS

BEFORE WE GO INTO MATH…

Let’s imagine two actors:

BEFORE WE GO INTO MATH…

Let's imagine two actors:

A fraud

Let’s imagine two actors:

A fraud

An art expert

BEFORE WE GO INTO MATH…

Let’s imagine two actors:

A fraud

… and a real artist

An art expert

BEFORE WE GO INTO MATH…

Let’s imagine two actors:

The fraud aims to copy the real artist and cheat the art expert.

A fraud

… and a real artist

An art expert

Let’s imagine two actors:

The fraud aims to copy the real artist and cheat the art expert.

A fraud

… and a real artist

An art expert

The expert assesses a painting and gives her opinion.

Let's imagine two actors:

A fraud

The fraud aims to copy the real artist and cheat the art expert.

The fraud learns and tries to fool the expert.

An art expert

The expert assesses a painting and gives her opinion.

… and a real artist

Let’s imagine two actors:

A fraud

Hmmm…
A FAKE!

An art expert

… and a real artist

BEFORE WE GO INTO MATH…

Let’s imagine two actors:

A fraud

An art expert

… and a real artist

Let's imagine two actors:

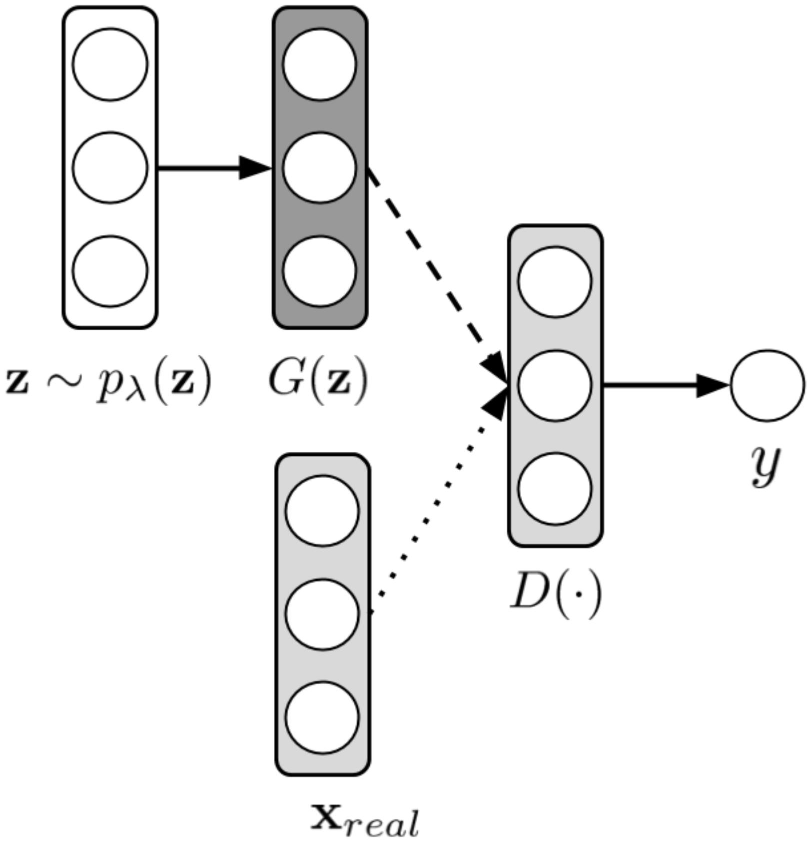
A fraud

Hmmm…
PABLO!

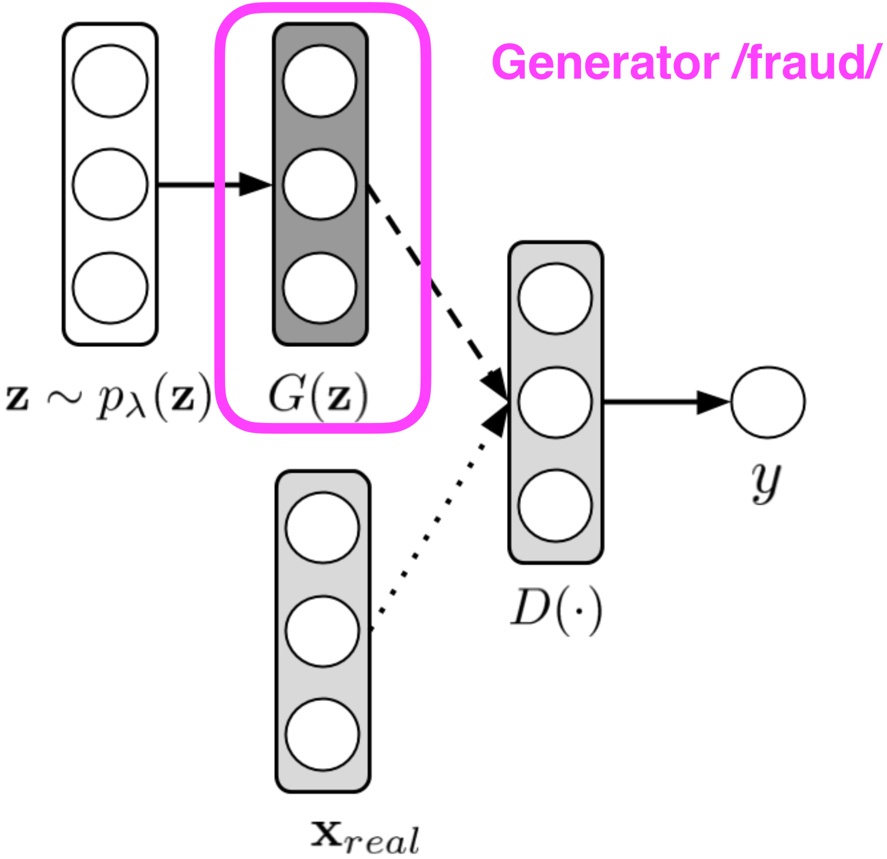
An art expert

… and a real artist

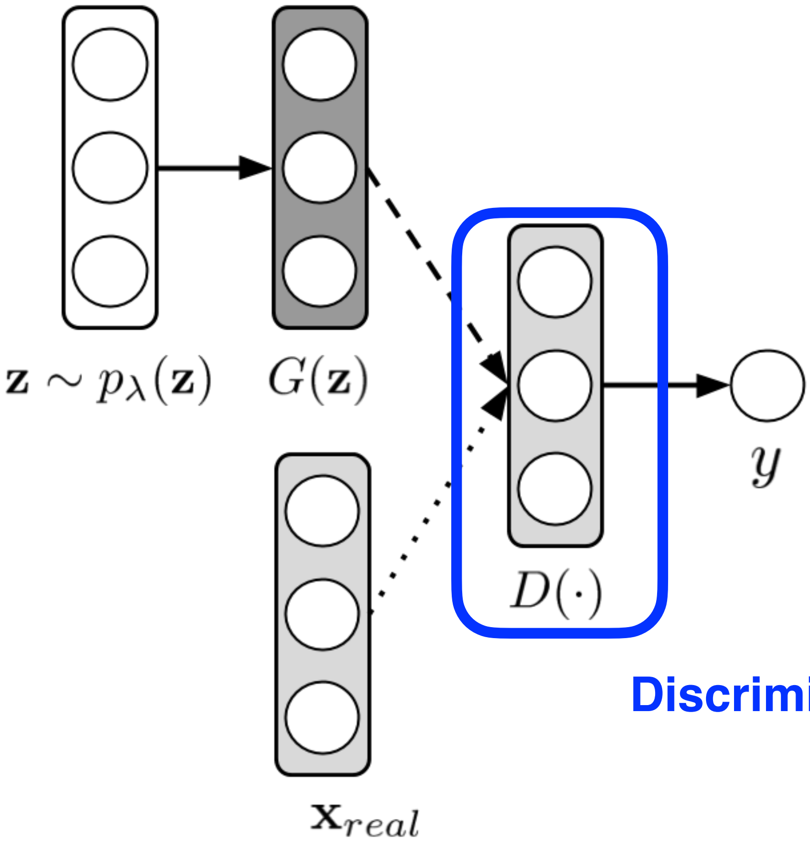
HOW WE CAN FORMULATE IT?



HOW WE CAN FORMULATE IT?



HOW WE CAN FORMULATE IT?



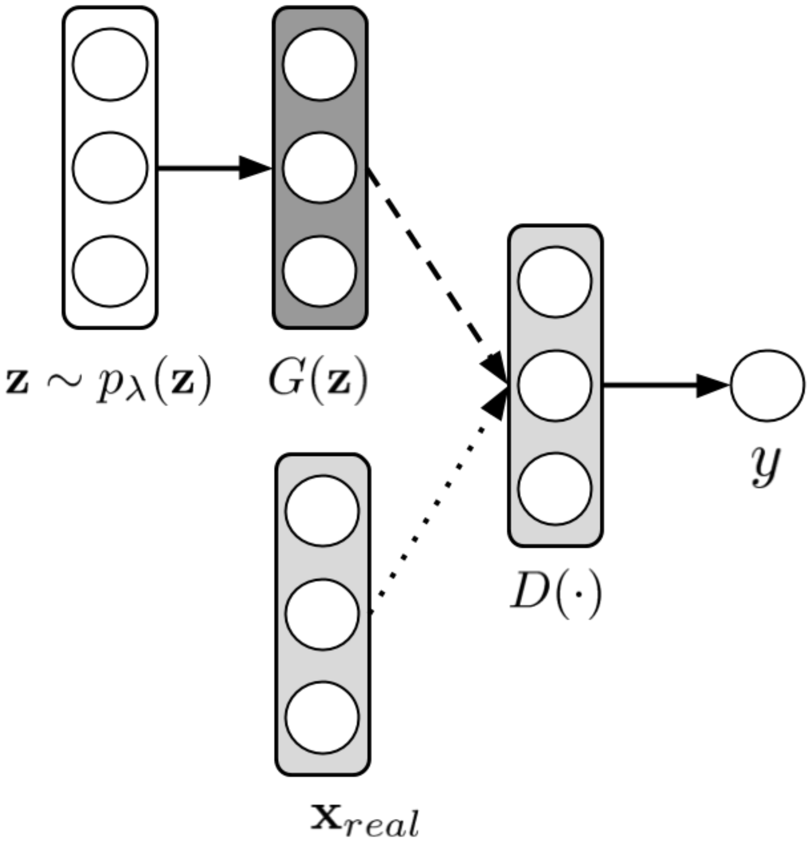
Discriminator /expert/

HOW WE CAN FORMULATE IT?

1. Sample **z**.

2. Generate *G*(**z**).

3. Discriminate whether given image is real or fake.

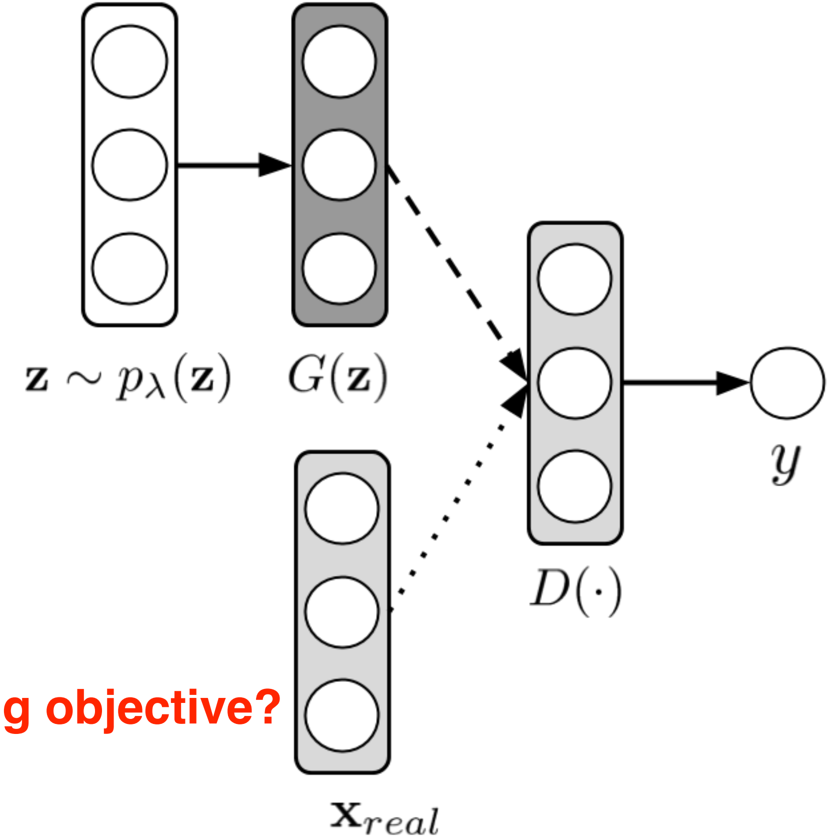


HOW WE CAN FORMULATE IT?

1. Sample **z**.

2. Generate *G*(**z**).

3. Discriminate whether given image is real or fake.



What about the learning objective?

ADVERSARIAL LOSS

We can consider the following learning objective:

$$\min_G \max_D \mathbb{E}_{\mathbf{x} \sim p_{real}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))]$$

Goodfellow, I., et al. (2014). *Generative adversarial nets*. In NIPS

ADVERSARIAL LOSS

We can consider the following learning objective:

$$\min_G \max_D \mathbb{E}_{\mathbf{x}\sim p_{real}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))]$$

It resemblances the logarithm of the Bernoulli distribution:

$$y \log p(y = 1) + (1 - y)\log(1 - p(y = 1))$$

Goodfellow, I., et al. (2014). *Generative adversarial nets*. In NIPS

ADVERSARIAL LOSS

We can consider the following learning objective:

$$\min_G \max_D \mathbb{E}_{\mathbf{x}\sim p_{real}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))]$$

It resemblances the logarithm of the Bernoulli distribution:

$$y \log p(y=1) + (1-y)\log(1 - p(y=1))$$

Goodfellow, I., et al. (2014). *Generative adversarial nets*. In NIPS

We can consider the following learning objective:

$$\min_G \max_D \mathbb{E}_{\mathbf{x}\sim p_{real}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))]$$

It resemblances the logarithm of the Bernoulli distribution:

$$y \log p(y=1) + (1-y)\log(1 - p(y=1))$$

Therefore, the discriminator network should end with a sigmoid function to mimic probability.

Goodfellow, I., et al. (2014). *Generative adversarial nets*. In NIPS

We can consider the following learning objective:

$$\min_G \max_D \mathbb{E}_{\mathbf{x}\sim p_{real}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))]$$

We want to minimize wrt. generator.

Goodfellow, I., et al. (2014). *Generative adversarial nets*. In NIPS

ADVERSARIAL LOSS

We can consider the following learning objective:

$$\min_G \max_D \mathbb{E}_{\mathbf{x} \sim p_{real}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))]$$

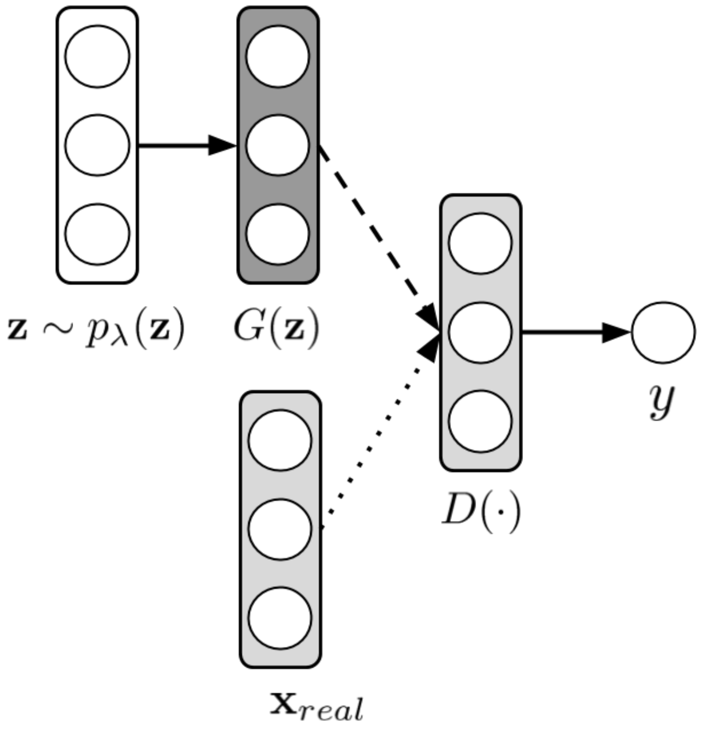
We want to maximize wrt. discriminator.

Goodfellow, I., et al. (2014). *Generative adversarial nets*. In NIPS

Generative process:

1. Sample **z**.

2. Generate *G*(**z**).



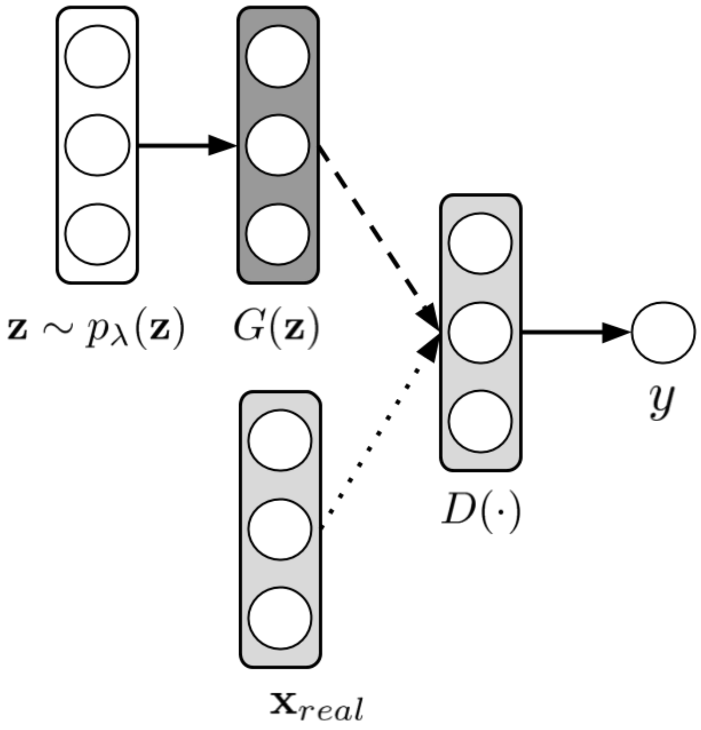
Generative process:

1. Sample **z**.

2. Generate $G(\mathbf{z})$.

The learning objective (**adversarial loss**):

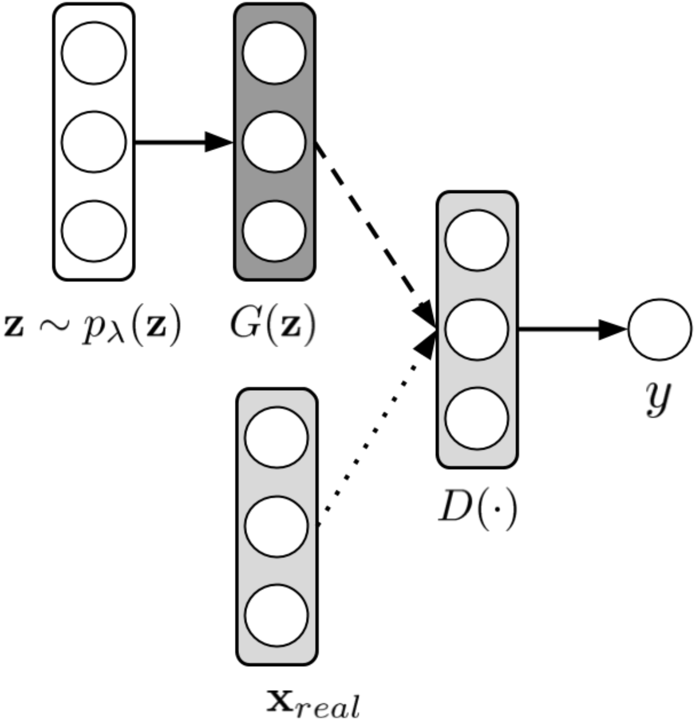
$$\min_G \max_D \mathbb{E}_{\mathbf{x}\sim p_{real}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))]$$



Generative process:

1. Sample **z**.

2. Generate *G*(**z**).



The learning objective (**adversarial loss**):

$$\min_G \max_D \mathbb{E}_{\mathbf{x}\sim p_{real}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))]$$

Learning:

1. Generate fake images, and minimize wrt. *G*.

2. Take real & fake images, and maximize wrt. *D*.

GANS

```
import torch.nn as nn

class GAN(nn.Module):
    def __init__(self, D, M):
        super(GAN, self).__init__()
        self.D = D
        self.M = M

        self.gen1 = nn.Linear(in_features= self.M, out_features=300)
        self.gen2 = nn.Linear(in_features=300, out_features= self.D)

        self.dis1 = nn.Linear(in_features= self.D, out_features=300)
        self.dis2 = nn.Linear(in_features=300, out_features=1)
```

GANS

```
def generate(self, N):
    z = torch.randn(size=(N, self.D))
    x_gen = self.gen1(z)
    x_gen = nn.functional.relu(x_gen)
    x_gen = self.gen2(x_gen)
    return x_gen

def discriminate(self, x):
    y = self.dis1(x)
    y = nn.functional.relu(y)
    y = self.dis2(y)
    y = torch.sigmoid(y)
    return y
```

GANS

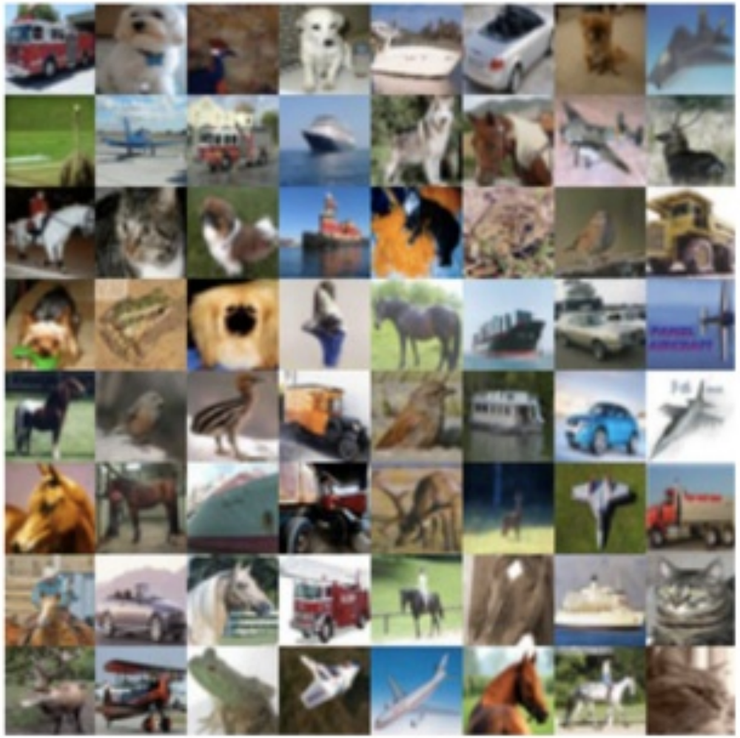
```
def gen_loss(self, d_gen):
    return torch.log(1. - d_gen)

def dis_loss(self, d_real, d_gen):
    # We maximize wrt. the discriminator, but optimizers minimize!
    # We need to include the negative sign!
    return -(torch.log(d_real) + torch.log(1. - d_gen))

def forward(self, x_real):
    x_gen = self.generate(N=x_real.shape[0])
    d_real = self.discriminate(x_real)
    d_gen = self.discriminate(x_gen)
    return d_real, d_gen
```

GANS

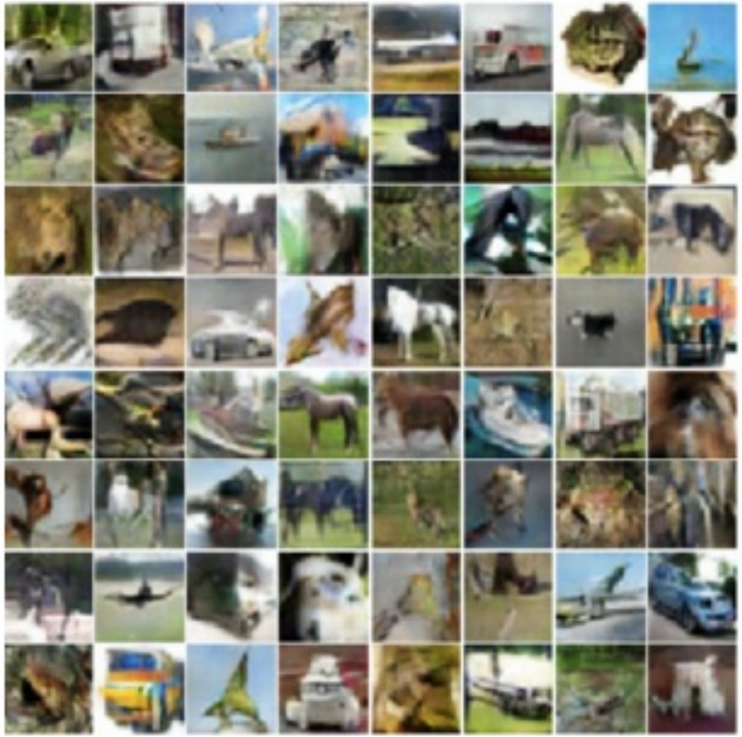
```
def gen_loss(self, d_gen):
    return torch.log(1. - d_gem)

def dis_loss(self, d_real, d_gen):
    # We maximize wrt. the discriminator, but optimizers minimize!
    # We need to include the negative sign!
    return -(torch.log(d_real) + torch.log(1. - d_gen))

def forward(self, x_real):
    x_gen = self.generate(N=x_real.shape[0])
    d_real = self.discriminate(x_real)
    d_gen = self.discriminate(x_gen)
    return d_real, d_gen
```

We can use two optimizers, one for d_real & d_gen, and one for d_gen.

GENERATIONS



Training Data



Samples

Salimans, T., et al. (2016). Improved techniques for training GANs. NeurIPS

GANS

Advantages

✓ Non-linear transformations.

✓ Allows to generate.

✓ Learnable loss.

✓ Allows implicit models.

✓ Works in high-dim.

Disadvantages

- No exact likelihood.
- **Unstable training**.
- *Missing mode problem* (i.e., it doesn't cover the whole space).
- No clear way for quantitative assessment.

Before, the motivation for the adversarial loss was the Bernoulli distribution. But we can use other ideas.

WASSERSTEIN GAN

Before, the motivation for the adversarial loss was the Bernoulli distribution. But we can use other ideas.

For instance, we can use the **earth-mover distance**:

$$\min_G \max_{D \in \mathscr{W}} \mathbb{E}_{\mathbf{x} \sim p_{real}}[D(\mathbf{x})] - \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})}[D(G(\mathbf{z}))]$$

where the discriminator is a 1-Lipschitz function.

Arjovsky, M., Chintala, S., & Bottou, L. (2017). Wasserstein GAN. arXiv preprint arXiv:1701.07875.

WASSERSTEIN GAN

Before, the motivation for the adversarial loss was the Bernoulli distribution. But we can use other ideas.

For instance, we can use the **earth-mover distance**:

$$\min_G \max_{D \in \mathscr{W}} \mathbb{E}_{\mathbf{x} \sim p_{real}}[D(\mathbf{x})] - \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})}[D(G(\mathbf{z}))]$$

where the discriminator is a 1-Lipschitz function.

We need to **clip weights** of the discriminator: `clip(weights, -c, c)`

Arjovsky, M., Chintala, S., & Bottou, L. (2017). Wasserstein GAN. arXiv preprint arXiv:1701.07875.

WASSERSTEIN GAN

Before, the motivation for the adversarial loss was the Bernoulli distribution. But we can use other ideas.

For instance, we can use the **earth-mover distance**:

$$\min_G \max_{D \in \mathscr{W}} \mathbb{E}_{\mathbf{x} \sim p_{real}}[D(\mathbf{x})] - \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})}[D(G(\mathbf{z}))]$$

where the discriminator is a 1-Lipschitz function.

We need to **clip weights** of the discriminator: `clip(weights, -c, c)`

It stabilizes training, but **other problems remain**.

Arjovsky, M., Chintala, S., & Bottou, L. (2017). Wasserstein GAN. arXiv preprint arXiv:1701.07875.

Thank you!

VU VRIJE UNIVERSITEIT AMSTERDAM