Deep generative modeling: Implicit models

Jakub M. Tomczak
Deep Learning
TYPES OF GENERATIVE MODELS

Generative model

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

<table>
<thead>
<tr>
<th>Model Category</th>
<th>Training</th>
<th>Likelihood</th>
<th>Sampling</th>
<th>Compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autoregressive models (e.g., PixelCNN)</td>
<td>Stable</td>
<td>Exact</td>
<td>Slow</td>
<td>No</td>
</tr>
<tr>
<td>Flow-based models (e.g., RealNVP)</td>
<td>Stable</td>
<td>Exact</td>
<td>Fast/Slow</td>
<td>No</td>
</tr>
<tr>
<td>Implicit models (e.g., GANs)</td>
<td>Unstable</td>
<td>No</td>
<td>Fast</td>
<td>No</td>
</tr>
<tr>
<td>Prescribed models (e.g., VAEs)</td>
<td>Stable</td>
<td>Approximate</td>
<td>Fast</td>
<td>Yes</td>
</tr>
<tr>
<td>Generative Models</td>
<td>Training</td>
<td>Likelihood</td>
<td>Sampling</td>
<td>Compression</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>----------</td>
<td>------------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>Autoregressive models (e.g., PixelCNN)</td>
<td>Stable</td>
<td>Exact</td>
<td>Slow</td>
<td>No</td>
</tr>
<tr>
<td>Flow-based models (e.g., RealNVP)</td>
<td>Stable</td>
<td>Exact</td>
<td>Fast/Slow</td>
<td>No</td>
</tr>
<tr>
<td>Implicit models (e.g., GANs)</td>
<td>Unstable</td>
<td>No</td>
<td>Fast</td>
<td>No</td>
</tr>
<tr>
<td>Prescribed models (e.g., VAEs)</td>
<td>Stable</td>
<td>Approximate</td>
<td>Fast</td>
<td>Yes</td>
</tr>
</tbody>
</table>
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:

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

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

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

It could be estimated by MC samples.
Generative process:

1. $z \sim p_\lambda(z)$
2. $x \sim p_\theta(x \mid z)$

The log-likelihood function:

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

It could be estimated by MC samples. If we take standard Gaussian prior, we need to model $p(x \mid z)$ only!
Let’s consider a function $f$ that transforms $z$ to $x$.  

$z \sim p_\lambda(z) \quad f(z) \quad x$
Let’s consider a function $f$ that transforms $z$ to $x$. It must be a powerful (=flexible) transformation!

Let’s consider a function $f$ that transforms $z$ to $x$.

It must be a powerful (=flexible) transformation!

Let’s consider a function $f$ that transforms $z$ to $x$.

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

The log-likelihood function:

\[ \log p(x) = \log \int p_\theta(x \mid z)p_\lambda(z)dz \]

\[ \approx \log \frac{1}{S} \sum_{s=1}^{S} \exp \left( \log p_\theta(x \mid z_s) \right) \]

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:

\[
\log p(x) = \log \int p_\theta(x \mid z)p_\lambda(z)dz 
\approx \log \frac{1}{S} \sum_{s=1}^{S} \exp \left( \log p_\theta(x \mid z_s) \right) 
\]

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

\[ p(x \mid z) = \delta (x - f(z)) \]
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:

\[ p(x \mid z) = \delta(x - f(z)) \]

However, now we cannot use the likelihood-based approach, because \( \ln \delta(x - f(z)) \) 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:

\[ p(x | z) = \delta (x - f(z)) \]

However, now we cannot use the likelihood-based approach, because \( \ln \delta (x - f(z)) \) is ill-defined.

We need to use a different approach.
Let’s imagine two actors:
Let’s imagine two actors:

A fraud
Let’s imagine two actors:

A fraud

An art expert
Let’s imagine two actors:

A fraud

... and a real artist

An art expert
Let’s imagine two actors:

A fraud

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

... and a real artist

An art expert
BEFORE WE GO INTO MATH...

Let’s imagine two actors:

- A fraud
- ... and a real artist

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

The expert assesses a painting and gives her opinion.
Before we go into math...

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.

- A real artist

An art expert

- The expert assesses a painting and gives her opinion.
BEFORE WE GO INTO MATH...

Let’s imagine two actors:

A fraud

... and a real artist

An art expert

Hmmm... A FAKE!
Let’s imagine two actors:

- A fraud
- ... and a real artist

An art expert

Hmmm... PABLO!
BEFORE WE GO INTO MATH...

Let’s imagine two actors:

- A fraud
- ... and a real artist

An art expert

Hmmm... PABLO!
HOW WE CAN FORMULATE IT?

\[ z \sim p_\lambda(z) \quad G(z) \quad D(\cdot) \quad y \]

\[ X_{\text{real}} \]
HOW WE CAN FORMULATE IT?

Generator /fraud/

\[ z \sim p_\lambda(z) \]

\[ G(z) \]

\[ y \]

\[ X_{real} \]

\[ D(\cdot) \]
HOW WE CAN FORMULATE IT?

$z \sim p_\lambda(z)$  $G(z)$  $y$

$X_{\text{real}}$

$D(\cdot)$

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?
We can consider the following learning objective:

$$\min_G \max_D \mathbb{E}_{x \sim p_{\text{real}}} \left[ \log D(x) \right] + \mathbb{E}_{z \sim p(z)} \left[ \log(1 - D(G(z))) \right]$$
We can consider the following learning objective:

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

It resembles 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:

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

It resembles 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:

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

It resembles 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.
We can consider the following learning objective:

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

We want to minimize wrt. generator.

We can consider the following learning objective:

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

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 $G(z)$.

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

$$\min_G \max_D \mathbb{E}_{x \sim p_{\text{real}}} \left[ \log D(x) \right] + \mathbb{E}_{z \sim p(z)} \left[ \log (1 - D(G(z))) \right]$$
Generative process:

1. Sample \( z \).
2. Generate \( G(z) \).

The learning objective (adversarial loss):

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

Learning:

1. Generate fake images, and minimize wrt. \( G \).
2. Take real & fake images, and maximize wrt. \( D \).
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)
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
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
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

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

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

where the discriminator is a 1-Lipschitz function.
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**:

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

where the discriminator is a 1-Lipschitz function.

We need to **clip weights** of the discriminator: \texttt{clip(weights, -c, c)}
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:

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

where the discriminator is a 1-Lipschitz function.

We need to clip weights of the discriminator: \text{clip}(\text{weights, -c, c})

It stabilizes training, but other problems remain.
Thank you!