Generative Adversarial Nets

He Zean [1]

[1]Dept. of Computer Science and Engineering, SUSTech

Nov. 9, 2022

Outline

Outline

1. Introduction
   - Background
   - Motivation
2. Method
3. Experiments
4. Analysis
5. Code Implementation

Introduction: Overview

Introduction: Overview

Introduction: Overview

- Generative Adversarial Nets (GAN) was first introduced by Goodfellow *et al.* in 2014 [1].

Introduction: Overview

- Generative Adversarial Nets (GAN) was first introduced by Goodfellow *et al.* in 2014 [1].
- Popular Applications:

Introduction: Overview

- Generative Adversarial Nets (GAN) was first introduced by Goodfellow *et al.* in 2014 [1].
- Popular Applications:
  - Image Editing

Introduction: Overview

- Generative Adversarial Nets (GAN) was first introduced by Goodfellow *et al.* in 2014 [1].
- Popular Applications:
  - Image Editing
  - Data Generating

Introduction: Overview

- Generative Adversarial Nets (GAN) was first introduced by Goodfellow *et al.* in 2014 [1].
- Popular Applications:
  - Image Editing
  - Data Generating
  - Cross-Domain Image Translation [2]

Introduction: Overview

- Generative Adversarial Nets (GAN) was first introduced by Goodfellow *et al.* in 2014 [1].
- Popular Applications:
  - Image Editing
  - Data Generating
  - Cross-Domain Image Translation [2]

Retrieved from https://paperswithcode.com/method/gan.

Introduction: Background

- Deep undirected graphical models (ex. RBMs, DBMs)

- Deep undirected graphical models (ex. RBMs, DBMs)
  - The partition function & gradient are hard to calculate.

Introduction: Background

- Deep undirected graphical models (ex. RBMs, DBMs)
  - The partition function & gradient are hard to calculate.
  - Mixing is a problem for learning algorithms that rely on MCMC.

Introduction: Background

- Deep undirected graphical models (ex. RBMs, DBMs)
  - The partition function & gradient are hard to calculate.
  - Mixing is a problem for learning algorithms that rely on MCMC.
- Deep directed graphical models (ex. DBNs, hybrid)

Introduction: Background

- Deep undirected graphical models (ex. RBMs, DBMs)
  - The partition function & gradient are hard to calculate.
  - Mixing is a problem for learning algorithms that rely on MCMC.
- Deep directed graphical models (ex. DBNs, hybrid)
  - Computational difficulties: a fast approximate layer-wise training criterion exists.

Introduction: Background

- Deep undirected graphical models (ex. RBMs, DBMs)
  - The partition function & gradient are hard to calculate.
  - Mixing is a problem for learning algorithms that rely on MCMC.
- Deep directed graphical models (ex. DBNs, hybrid)
  - Computational difficulties: a fast approximate layer-wise training criterion exists.
- Generative autoencoders (ex. NCE, GSNs)

Introduction: Background

- Deep undirected graphical models (ex. RBMs, DBMs)
    - The partition function & gradient are hard to calculate.
    - Mixing is a problem for learning algorithms that rely on MCMC.
- Deep directed graphical models (ex. DBNs, hybrid)
    - Computational difficulties: a fast approximate layer-wise training criterion exists.
- Generative autoencoders (ex. NCE, GSNs)
    - Fixed noise distribution, learning slows dramatically if the distribution of even a small subset of data has been learnt.

Introduction: Background

- Deep undirected graphical models (ex. RBMs, DBMs)
  - The partition function & gradient are hard to calculate.
  - Mixing is a problem for learning algorithms that rely on MCMC.
- Deep directed graphical models (ex. DBNs, hybrid)
  - Computational difficulties: a fast approximate layer-wise training criterion exists.
- Generative autoencoders (ex. NCE, GSNs)
  - Fixed noise distribution, learning slows dramatically if the distribution of even a small subset of data has been learnt.
  - Performance problem: requires feedback loops, less able to leverage piecewise linear units.

Introduction: Motivation

- We want to construct a generative model that have lower computational costs, comparing with existing generative models.

Introduction: Motivation

- We want to construct a generative model that have lower computational costs, comparing with existing generative models.
- We want to sidestep difficulties such as intractable probabilistic computations and leveraging piecewise linear units.

Outline

Method

Method

Method

- **Generator**: input a random noise, the generator "convert" it to a fake image, and try to cheat the discriminator.

Method

- **Generator**: input a random noise, the generator "convert" it to a fake image, and try to cheat the discriminator.
- **Discriminator**: input an image (n-dim vector), output a label: the image comes from training data (real) or generator (fake). To be practical in training, we directly use the estimated possibility.

Method

- **Generator**: input a random noise, the generator "convert" it to a fake image, and try to cheat the discriminator.
- **Discriminator**: input an image (n-dim vector), output a label: the image comes from training data (real) or generator (fake). To be practical in training, we directly use the estimated possibility.

Retrieved from https://zhuanlan.zhihu.com/p/33752313.

Method: Annotations

Example: A multilayer perceptron - multilayer perceptron GAN

w.r.t.:

Example: A multilayer perceptron - multilayer perceptron GAN

$$G\left(\mathbf{z};\theta_g\right), \mathbf{z} \sim p_{\mathbf{z}}\left(\mathbf{z}\right)$$

w.r.t.:

Method: Annotations

Example: A multilayer perceptron - multilayer perceptron GAN

$$G\left(\mathbf{z};\theta_g\right), \mathbf{z} \sim p_{\mathbf{z}}\left(\mathbf{z}\right) \qquad D\left(\mathbf{x};\theta_d\right) = p\left(\mathbf{x} \in \text{data}\right)$$

w.r.t.:

Method: Annotations

Example: A multilayer perceptron - multilayer perceptron GAN

$$G\left(\mathbf{z};\theta_g\right),\mathbf{z}\sim p_{\mathbf{z}}\left(\mathbf{z}\right) \qquad D\left(\mathbf{x};\theta_d\right)=p\left(\mathbf{x}\in\text{data}\right)$$

w.r.t.:

- $G\left(\mathbf{z};\theta_g\right)$ is the function representation of generator (multilayer perceptron) with parameters θ_g, differentiable. $D\left(\mathbf{x};\theta_d\right)$ similars.

Method: Annotations

Example: A multilayer perceptron - multilayer perceptron GAN

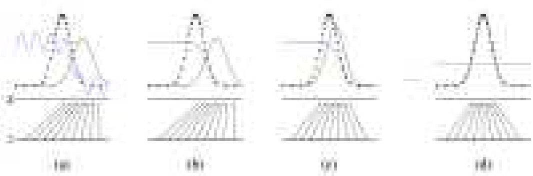
$$G\left(\mathbf{z};\theta_g\right), \mathbf{z} \sim p_{\mathbf{z}}\left(\mathbf{z}\right) \qquad D\left(\mathbf{x};\theta_d\right) = p\left(\mathbf{x} \in \text{data}\right)$$

w.r.t.:

- $G\left(\mathbf{z};\theta_g\right)$ is the function representation of generator (multilayer perceptron) with parameters θ_g, differentiable. $D\left(\mathbf{x};\theta_d\right)$ similars.
- $p_{\mathbf{z}}\left(\mathbf{z}\right)$ is the prior of input noise

POV: Minimax gaming

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))]$$



Method: Training Process

- Global Optimality of $p_g = p_{data}$

Method: Training Process

- Global Optimality of $p_g = p_{data}$
  - $C(G) = \log(4) + 2 \cdot JSD(p_{data} \| p_g)$

Method: Training Process

- $d\_loss = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [\log (1 - D(G(\mathbf{z})))]$, cross-entropy.

[1] Early in learning, when G is poor, rather than training G to minimize $\left[\log\left(1 - D\left(G\left(\mathbf{z}\right)\right)\right)\right]$, we can train G to maximize $\left[\log\left(D\left(G\left(\mathbf{z}\right)\right)\right)\right]$.

Method: Training Process

- $d\_loss = \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}\left[\log D\left(\mathbf{x}\right)\right] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})}\left[\log\left(1 - D\left(G\left(\mathbf{z}\right)\right)\right)\right]$, cross-entropy.
- $g\_loss = \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})}\left[\log\left(1 - D\left(G\left(\mathbf{z}\right)\right)\right)\right]$[1]

[1] Early in learning, when G is poor, rather than training G to minimize $\left[\log\left(1 - D\left(G\left(\mathbf{z}\right)\right)\right)\right]$, we can train G to maximize $\left[\log\left(D\left(G\left(\mathbf{z}\right)\right)\right)\right]$.

Method: Training Process

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

  for k steps **do**

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

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

  end for

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

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Outline

Experiments

- Datasets

Model	MNIST	TFD
DBN [3]	138 ± 2	1909 ± 66
Stacked CAE [3]	121 ± 1.6	2110 ± 50
Deep GSN [6]	214 ± 1.1	1890 ± 29
Adversarial nets	225 ± 2	2057 ± 26

- Datasets
  - MNIST, TFD, CIFAR-10

Model	MNIST	TFD
DBN [3]	138 ± 2	1909 ± 66
Stacked CAE [3]	121 ± 1.6	$\mathbf{2110 \pm 50}$
Deep GSN [6]	214 ± 1.1	1890 ± 29
Adversarial nets	$\mathbf{225 \pm \boxed{2}}$	$\mathbf{\boxed{2057} \pm 26}$

Experiments

- Datasets
  - MNIST, TFD, CIFAR-10
- NNs' design will be mentioned in next section

Model	MNIST	TFD
DBN [3]	138 ± 2	1909 ± 66
Stacked CAE [3]	121 ± 1.6	$\mathbf{2110 \pm 50}$
Deep GSN [6]	214 ± 1.1	1890 ± 29
Adversarial nets	$\mathbf{225 \pm \boxed{2}}$	$\mathbf{\boxed{2057} \pm 26}$

Experiments

- Datasets
  - MNIST, TFD, CIFAR-10
- NNs' design will be mentioned in next section
- p_g estimation

Model	MNIST	TFD
DBN [3]	138 ± 2	1909 ± 66
Stacked CAE [3]	121 ± 1.6	$\mathbf{2110 \pm 50}$
Deep GSN [6]	214 ± 1.1	1890 ± 29
Adversarial nets	$\mathbf{225} \pm \boxed{2}$	$\boxed{2057} \pm 26$

Experiments

- Datasets
  - MNIST, TFD, CIFAR-10
- NNs' design will be mentioned in next section
- p_g estimation
  - Gaussian Parzen window

Model	MNIST	TFD
DBN [3]	138 ± 2	1909 ± 66
Stacked CAE [3]	121 ± 1.6	2110 ± 50
Deep GSN [6]	214 ± 1.1	1890 ± 29
Adversarial nets	$225 \pm \boxed{2}$	$\boxed{2057} \pm 26$

Experiments

- Datasets
  - MNIST, TFD, CIFAR-10
- NNs' design will be mentioned in next section
- p_g estimation
  - Gaussian Parzen window
  - Cross validation $\rightarrow \sigma$

Model	MNIST	TFD
DBN [3]	138 ± 2	1909 ± 66
Stacked CAE [3]	121 ± 1.6	$\mathbf{2110 \pm 50}$
Deep GSN [6]	214 ± 1.1	1890 ± 29
Adversarial nets	$\mathbf{225} \pm \boxed{2}$	$\boxed{\mathbf{2057}} \pm 26$

Outline

Analysis

- Advantages

Analysis

- Advantages
  - Less computational cost

Analysis

- Advantages
  - Less computational cost
  - Model can accept multiple types of functions (differentiable)

Analysis

- Advantages
  - Less computational cost
  - Model can accept multiple types of functions (differentiable)
  - Is able to represent very sharp, even degenerate distributions

Analysis

- Advantages
  - Less computational cost
  - Model can accept multiple types of functions (differentiable)
  - Is able to represent very sharp, even degenerate distributions
- Disadvantages

Analysis

- Advantages
  - Less computational cost
  - Model can accept multiple types of functions (differentiable)
  - Is able to represent very sharp, even degenerate distributions
- Disadvantages
  - Required synchronization between G and D (to avoid mode collapse)

Analysis

- Advantages
  - Less computational cost
  - Model can accept multiple types of functions (differentiable)
  - Is able to represent very sharp, even degenerate distributions
- Disadvantages
  - Required synchronization between G and D (to avoid mode collapse)
  - Highly sensitive to the hyperparameter selections

Outline

Input Format

```python
dataloader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "../../data/mnist",
        train=True,
        download=True,
        transform=transforms.Compose(
            [transforms.Resize(opt.img_size), transforms.ToTensor(), transforms.Normalize([0.5], [0.5])]
        ),
    ),
    batch_size=opt.batch_size,
    shuffle=True,
)
```

- MNIST dataset provides 60,000 samples, they are split into 938 $\times$ 64-sized mini-batch for each epoch

Input Format

```
dataloader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "../../data/mnist",
        train=True,
        download=True,
        transform=transforms.Compose(
            [transforms.Resize(opt.img_size), transforms.ToTensor(), transforms.Normalize([0.5], [0.5])]
        ),
    ),
    batch_size=opt.batch_size,
    shuffle=True,
)
```

- MNIST dataset provides 60,000 samples, they are split into 938 $\times$ 64-sized mini-batch for each epoch
- Each sample is a 28 $\times$ 28 pixel, 1 channel (grayscale) image

Input Format

```
dataloader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "../../data/mnist",
        train=True,
        download=True,
        transform=transforms.Compose(
            [transforms.Resize(opt.img_size), transforms.ToTensor(), transforms.Normalize([0.5], [0.5])]
        ),
    ),
    batch_size=opt.batch_size,
    shuffle=True,
)
```

- MNIST dataset provides 60,000 samples, they are split into 938 $\times$ 64-sized mini-batch for each epoch
- Each sample is a 28 $\times$ 28 pixel, 1 channel (grayscale) image
- ToTensor() + Normalize([0.5], [0.5]): each pixel's value is mapped to the range of [-1, 1]

Input Format (Cont.)

```
# Sample noise as generator input
z = Variable(Tensor(np.random.normal(0, 1, (imgs.shape[0], opt.latent_dim))))
```

- For each mini-batch, it generates 64 (batch size) 100-dim (latent dimension) Gaussian noises ($\mu = 0, \sigma = 1$) as noise prior $p_g\left(\mathbf{z}\right)$

Misc.

- Optimizer: Adam

Misc.

- Optimizer: Adam
- Adversarial loss: Binary Classification Error Loss

- Optimizer: Adam
- Adversarial loss: Binary Classification Error Loss
- 

Output Format

```python
print(
    "[Epoch %d/%d] [Batch %d/%d] [D loss: %f] [G loss: %f]"
    % (epoch, opt.n_epochs, i, len(dataloader), d_loss.item(), g_loss.item())
)

batches_done = epoch * len(dataloader) + i
if batches_done % opt.sample_interval == 0:
    save_image(gen_imgs.data[:25], "images/%d.png" % batches_done, nrow=5, normalize=True)
```

- For each mini-batch, it uses BCELoss to calculate d_loss (loss for discriminator) and g_loss (loss for generator)

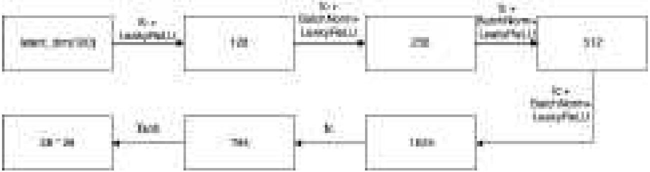
Output Format

```python
print(
    "[Epoch %d/%d] [Batch %d/%d] [D loss: %f] [G loss: %f]"
    % (epoch, opt.n_epochs, i, len(dataloader), d_loss.item(), g_loss.item())
)

batches_done = epoch * len(dataloader) + i
if batches_done % opt.sample_interval == 0:
    save_image(gen_imgs.data[:25], "images/%d.png" % batches_done, nrow=5, normalize=True)
```

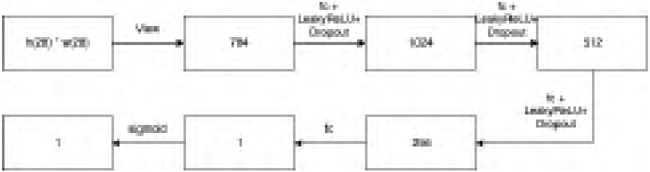
- For each mini-batch, it uses BCELoss to calculate d_loss (loss for discriminator) and g_loss (loss for generator)
- For every 400 batches (sample interval), it saves the first 25 generated images (28×28 pixel, 1 channel grayscale)

Descriptions



Generator NN Structure

Descriptions

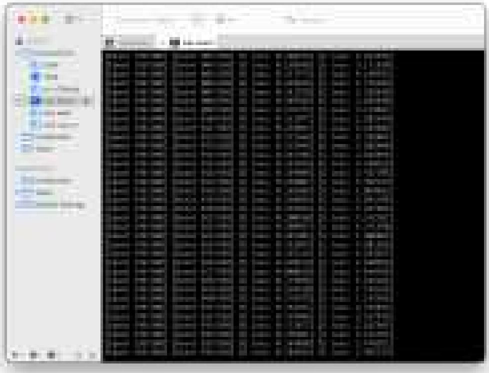


Discriminator NN Structure

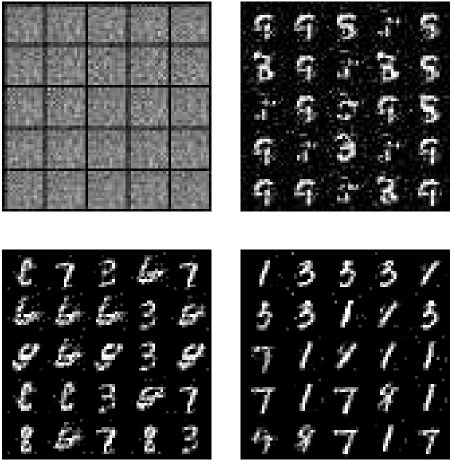
Running Result

Ran on server with GeForce RTX 2080 Ti

200 epochs, batch size = 64, learning rate = 0.0002

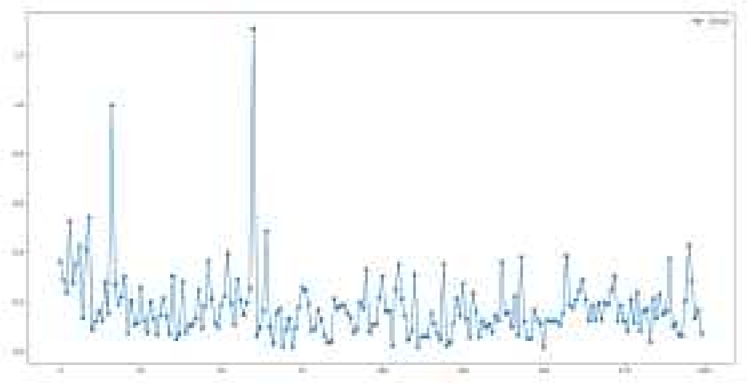


Running Result

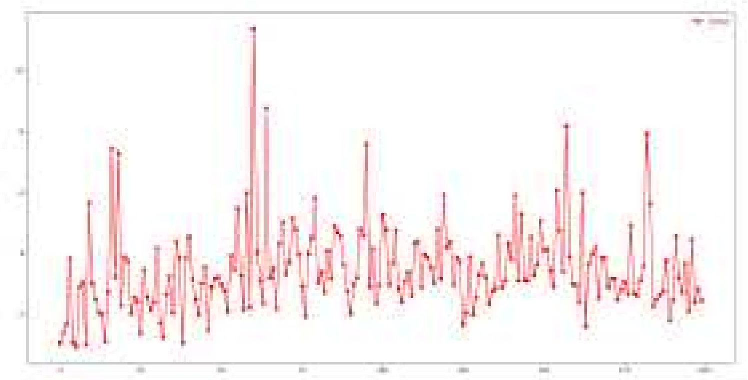


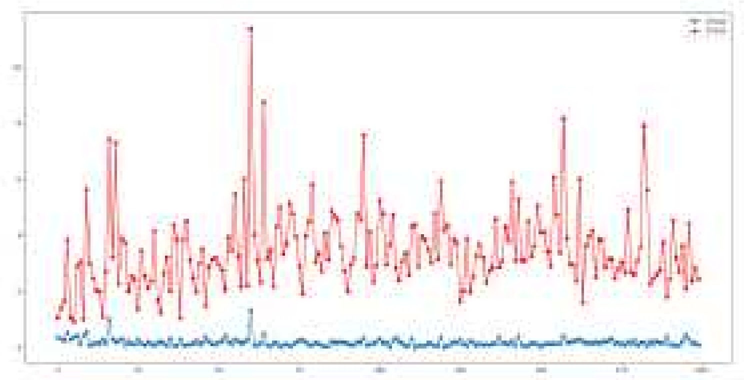
Generated Images for MNIST (patch #0, #46800, #93600, and # 180000)

Running Result



Running Result





References I

[1] Ian J. Goodfellow **andothers**. "Generative Adversarial Nets". **in***Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*: **byeditor**Zoubin Ghahramani **andothers**. 2014, **pages** 2672–2680.

[2] Ming-Yu Liu, Thomas M. Breuel **and** Jan Kautz. "Unsupervised Image-to-Image Translation Networks". **in***Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*: **byeditor**Isabelle Guyon **andothers**. 2017, **pages** 700–708.