Department of Mechanical Engineering
Indian Institute of Technology Tirupati

Analysis of GAN and its variants

Aakash

Department of Electrical Engineering
Indian Institute of Technology Tirupati

Generative models

Task at hand: generate new (fake) samples when training data is given

Real data

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

Fake data

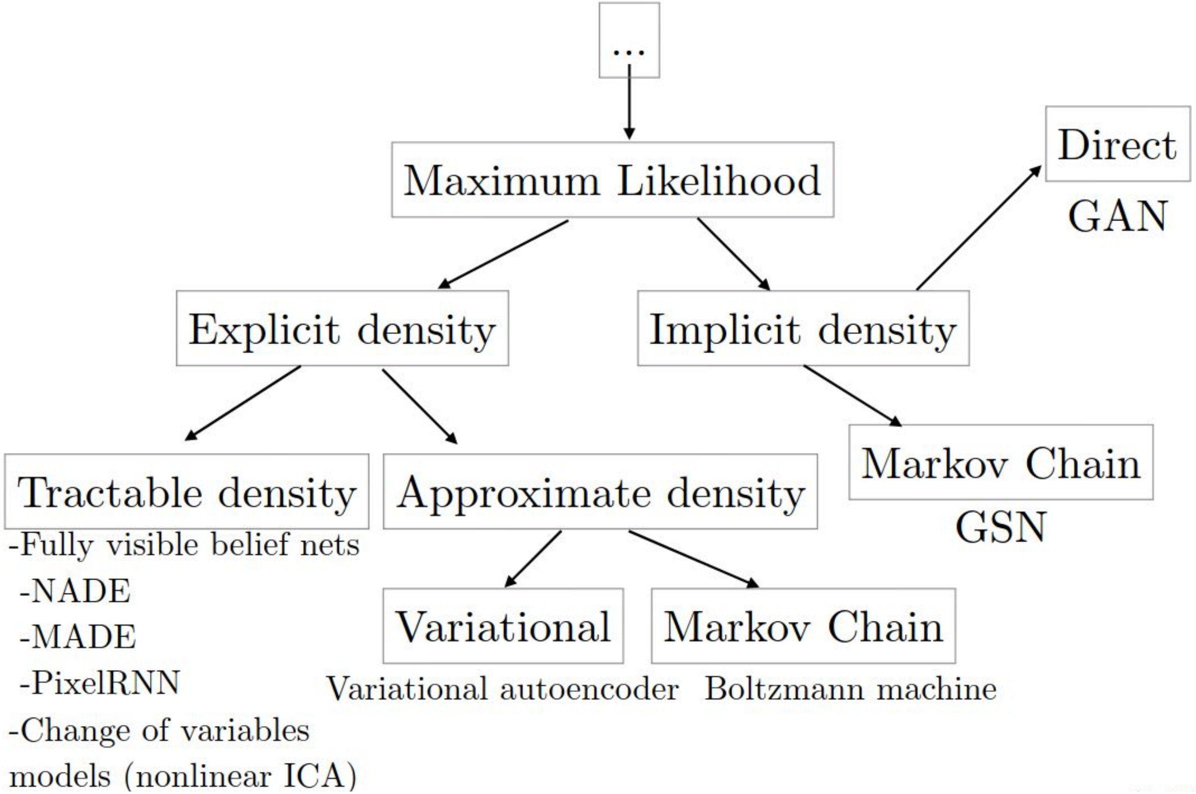
Generated samples ~ $p_{model}(x)$

We need to find model density similar to given data density. This can be done in two ways

- **Explicit density estimation**: explicitly define and solve for $p_{model}(x)$
- **Implicit density estimation**: learn model that can sample from $p_{model}(x)$ without explicitly defining it

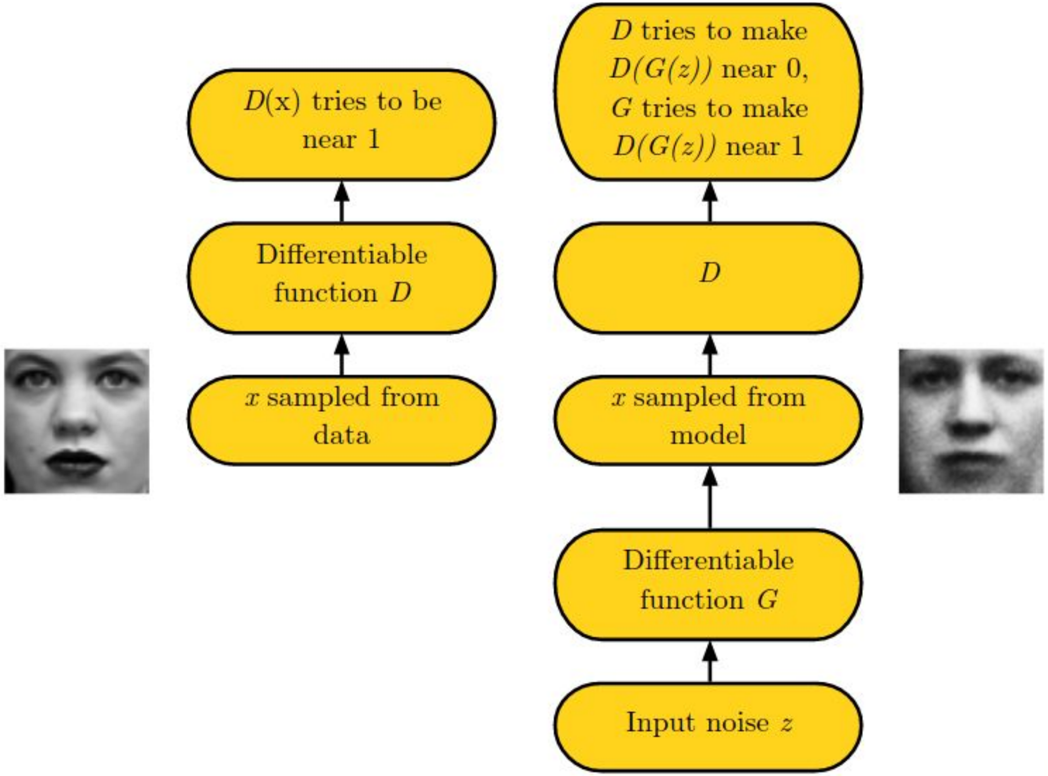
Modified - CS231n Stanford

Generative models



Ian Goodfellow (2016)

GAN: Adversarial Net Framework



Ian Goodfellow, NIPS 2016

GAN: Generative Adversarial Networks

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

Two player minimax game between generator (G) and discriminator (D)

- (D) tries to maximize the log-likelihood for the binary classification problem
  - data: real (1)
  - generated: fake (0)
- (G) tries to minimize the log-probability of its samples being classified as "fake" by the discriminator (D)

Goodfellow et al, 2014
Deep Supervised Learning, Spring 2020, Pieter Abbeel

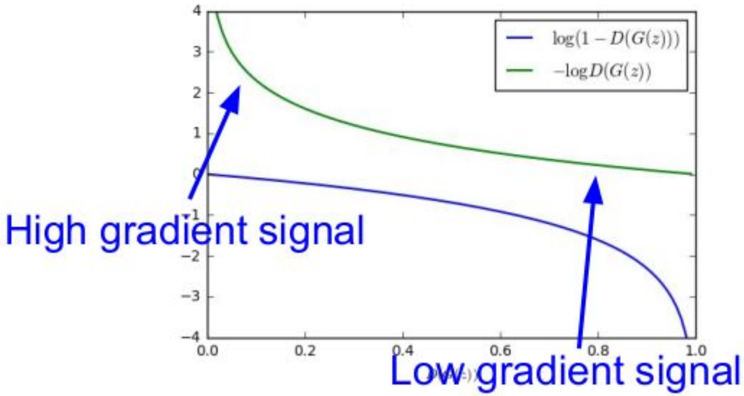
Training

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

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



CS231n Stanford

Training

1. **Gradient ascent** on discriminator

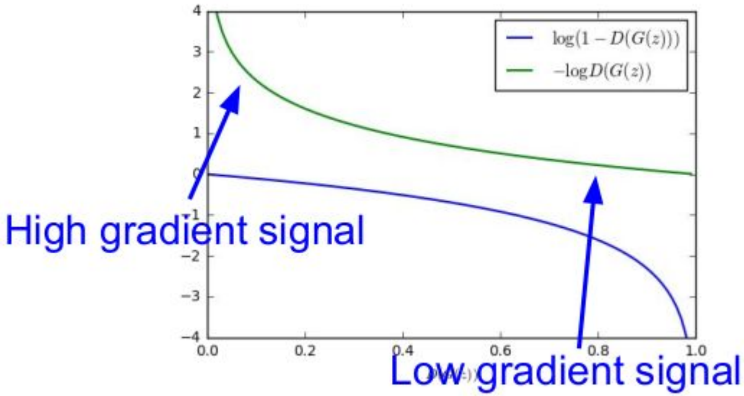
$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

2. **Gradient descent** on generator

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

$$\downarrow$$

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



CS231n Stanford

GAN: Bayes-Optimal Discriminator

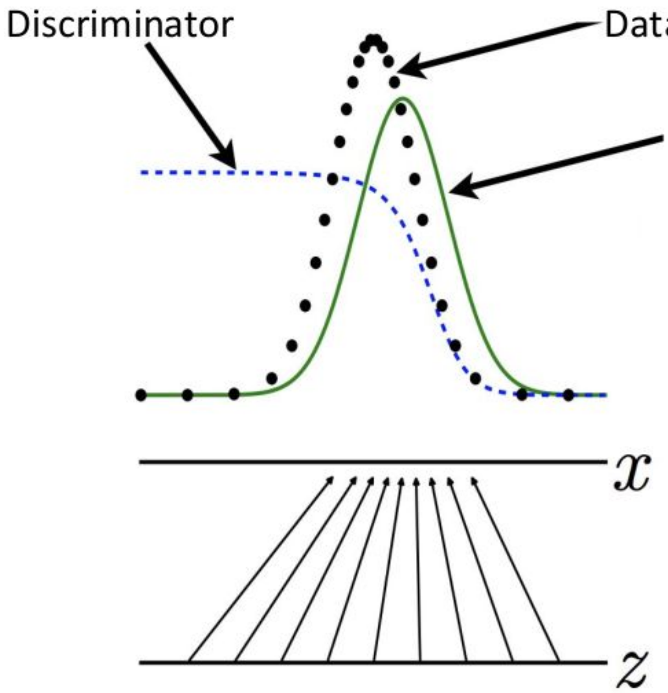
$$
\begin{aligned}
V(G, D) &= \mathbb{E}_{x \sim p_{\text{data}}} [\log D(x)] + \mathbb{E}_{z \sim p(z)} [\log(1 - D(G(z)))] \\
&= \int_x p_{\text{data}}(x) \log D(x) dx + \int_z p(z) \log(1 - D(G(z))) dz \\
&= \int_x p_{\text{data}}(x) \log D(x) dx + \int_x p_g(x) \log(1 - D(x)) dx \\
&= \int_x \left[p_{\text{data}}(x) \log D(x) + p_g(x) \log(1 - D(x)) \right] dx
\end{aligned}
$$

$$
\nabla_y [a \log y + b \log(1 - y)] = 0 \implies y^* = \frac{a}{a + b} \quad \forall \quad [a, b] \in \mathbb{R}^2 \backslash [0, 0]
$$

$$
\implies D^*(x) = \frac{p_{\text{data}}(x)}{(p_{\text{data}}(x) + p_g(x))}
$$

Goodfellow et al, 2014
Deep Supervised Learning, Spring 2020, Pieter Abbeel

GAN: Bayes-Optimal Discriminator



$$D(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_{\text{model}}(x)}$$

Goodfellow et al, 2014

GAN: Generator Objective under D*

$$
\begin{aligned}
V(G, D^*) &= \mathbb{E}_{x\sim p_{\text{data}}}[\log D^*(x)] + \mathbb{E}_{x\sim p_g}[\log(1 - D^*(x))] \\
&= \mathbb{E}_{x\sim p_{\text{data}}}\left[\log \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)}\right] + \mathbb{E}_{x\sim p_g}\left[\log \frac{p_g(x)}{p_{\text{data}}(x) + p_g(x)}\right] \\
&= -\log(4) + \underbrace{KL\left(p_{\text{data}} \middle\| \left(\frac{p_{\text{data}} + p_g}{2}\right)\right) + KL\left(p_{\text{g}} \middle\| \left(\frac{p_{\text{data}} + p_g}{2}\right)\right)}_{(\text{Jensen-Shannon Divergence (JSD) of } p_{\text{data}} \text{ and } p_g) \geq 0}
\end{aligned}
$$

$$
V(G^*, D^*) = -\log(4) \text{ when } p_g = p_{\text{data}}
$$

Goodfellow et al, 2014
Deep Supervised Learning, Spring 2020, Pieter Abbeel

GAN: Pseudocode

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 $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ 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:

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

  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 descending its stochastic gradient:

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

end for

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

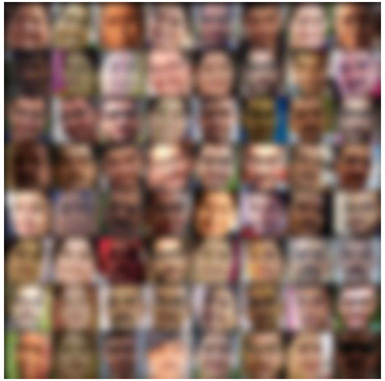
Goodfellow et al (2014)

Datasets: MNIST and Custom faces

भारतीय प्रौद्योगिकी संस्थान तिरुपति
TIRUPATI

MNIST

- The MNIST database consists of handwritten digits.
- The training set has 60,000 examples, and the test set has 10,000 examples.
- The MNIST dataset is a subset of NIST dataset. The digits have been normalized and centered.



→ **Image blurred for security purposes.**

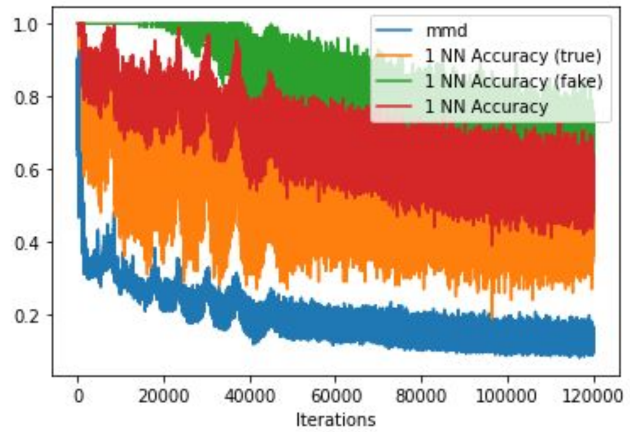
Custom face dataset

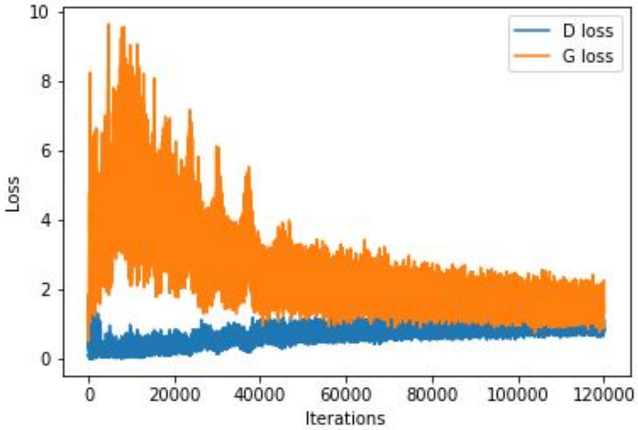
- The dataset consists of people's faces.
- It consists of 683 unlabeled face images of 5 persons (about 135 face images per person).
- The dataset has been created by running face detection on photos and center cropping them.
- The images are resized to 256x256 resolution.

GAN: Experiments and Results

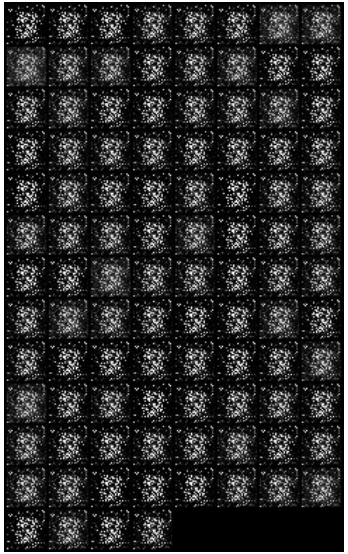
Discriminator
(0): Linear(in_features=784, out_features=256, bias=True)
(1): LeakyReLU(negative_slope=0.2)
(2): Linear(in_features=256, out_features=256, bias=True)
(3): LeakyReLU(negative_slope=0.2)
(4): Linear(in_features=256, out_features=1, bias=True)
(5): Sigmoid()
Generator
(0): Linear(in_features=64, out_features=256, bias=True)
(1): ReLU()
(2): Linear(in_features=256, out_features=256, bias=True)
(3): ReLU()
(4): Linear(in_features=256, out_features=784, bias=True)
(5): Tanh()

	MMD	1 NN Accuracy	1 NN Accuracy (real)	1 NN Accuracy (fake)
GAN	0.1964±0.0064	0.6835±0.0122	0.5346±0.0129	0.8324±0.0176

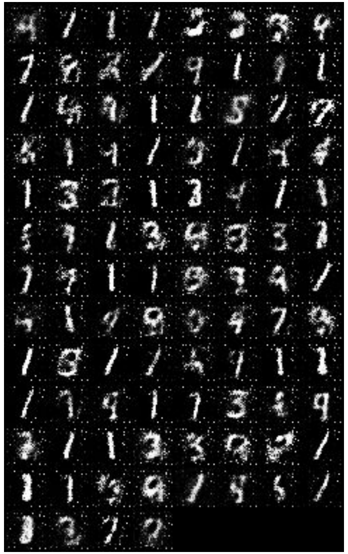




GAN: Experiments and Results



Epoch 1



Epoch 50

Epoch 100

Epoch 150

Epoch 200

WGAN: Wasserstein GAN

- Uses Wasserstein or Earth Mover's Distance instead of using JS Divergence
  - Smoother representation of distance between two distributions located in low dimensional manifolds

$$W(p_r, p_g) = \inf_{\gamma \sim \Pi(p_r, p_g)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|]$$

- Proposed Kantorovich-Rubinstein duality as dual of Wasserstein distance
- Maintains K-Lipschitz continuity using **weight clipping**: Slow convergence!!

Standard GAN

$$\boxed{\min_G \max_D \mathbb{E}_{x\sim P_r}[\log D(x)] + \mathbb{E}_{\tilde{x}\sim P_g}[\log(1 - D(\tilde{x}))]}$$

$\downarrow$

Wasserstein GAN

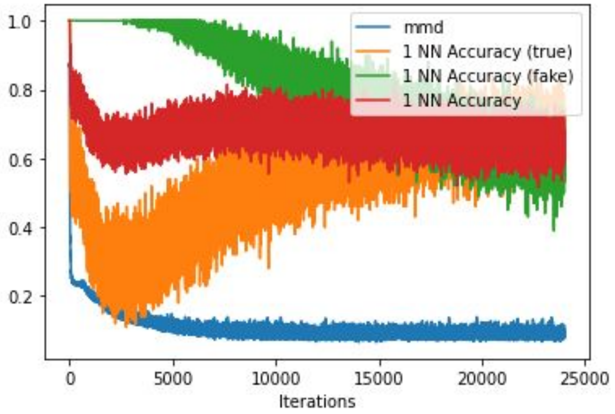
$$\boxed{\min_G \max_{D\in\mathcal{D}} \mathbb{E}_{x\sim P_r}[D(x)] - \mathbb{E}_{\tilde{x}\sim P_g}[D(\tilde{x})]}$$

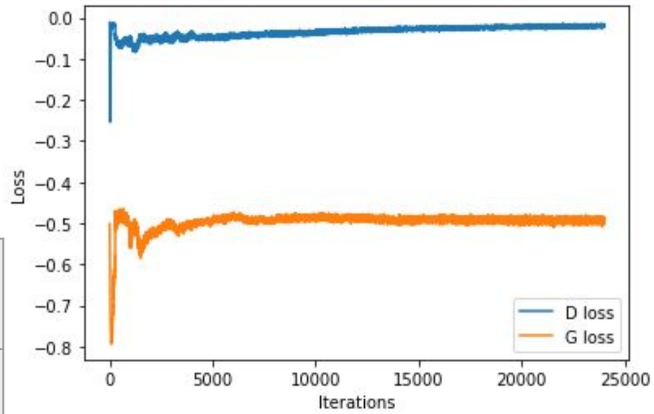
Arjovsky et al (2017)

WGAN: Experiments and Results

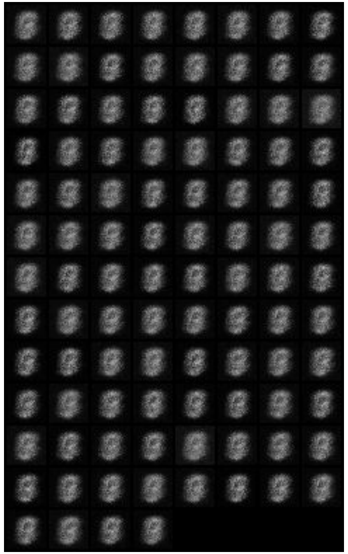
<u>Discriminator</u>
(0): Linear(in_features=784, out_features=256, bias=True)
(1): LeakyReLU(negative_slope=0.2)
(2): Linear(in_features=256, out_features=256, bias=True)
(3): LeakyReLU(negative_slope=0.2)
(4): Linear(in_features=256, out_features=1, bias=True)
(5): Sigmoid()
<u>Generator</u>
(0): Linear(in_features=64, out_features=256, bias=True)
(1): ReLU()
(2): Linear(in_features=256, out_features=256, bias=True)
(3): ReLU()
(4): Linear(in_features=256, out_features=784, bias=True)
(5): Tanh()

	MMD	1 NN Accuracy	1 NN Accuracy (real)	1 NN Accuracy (fake)
WGAN	0.1079±0.0016	0.6788±0.0019	0.5409±0.0198	0.8167±0.0209





WGAN: Experiments and Results



Epoch 1

Epoch 50

Epoch 100

Epoch 150

Epoch 200

DCGAN: Experiments and Results

Generator

(0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)

(1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)

(2): ReLU(inplace=True)

(3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)

(4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)

(5): ReLU(inplace=True)

(6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)

(7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)

(8): ReLU(inplace=True)

(9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)

(10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)

(11): ReLU(inplace=True)

(12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)

(13): Tanh()

Discriminator

(0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)

(1): LeakyReLU(negative_slope=0.2, inplace=True)

(2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)

(3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)

(4): LeakyReLU(negative_slope=0.2, inplace=True)

(5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)

(6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)

(7): LeakyReLU(negative_slope=0.2, inplace=True)

(8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)

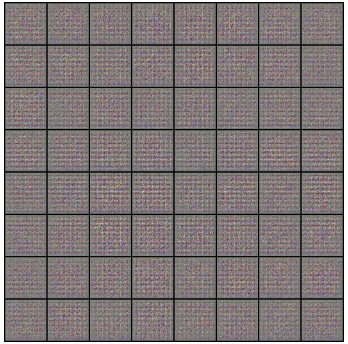
(9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)

(10): LeakyReLU(negative_slope=0.2, inplace=True)

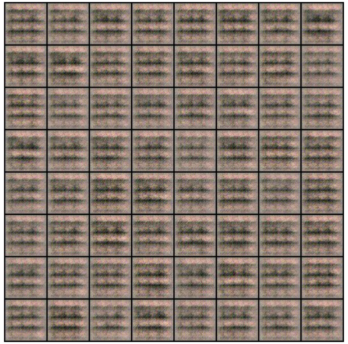
(11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)

(12): Sigmoid()

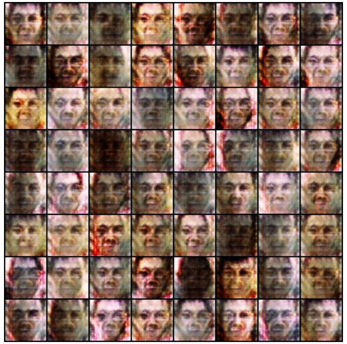
DCGAN: Experiments and Results



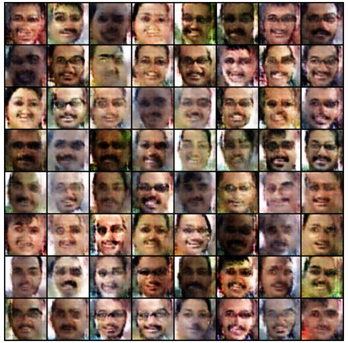
Epoch 1



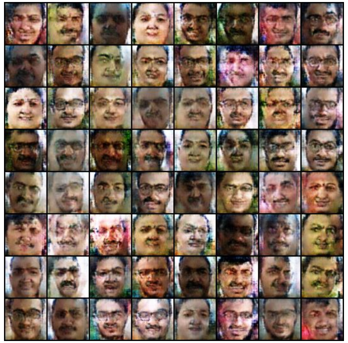
Epoch 50



Epoch 100

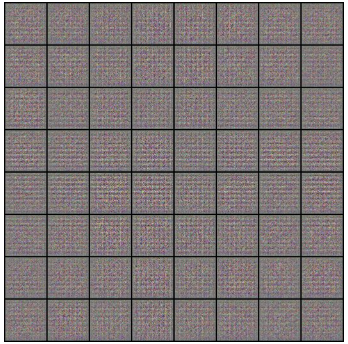


Epoch 150

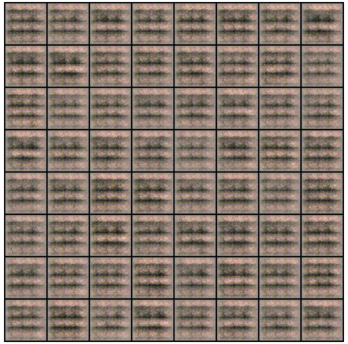


Epoch 200

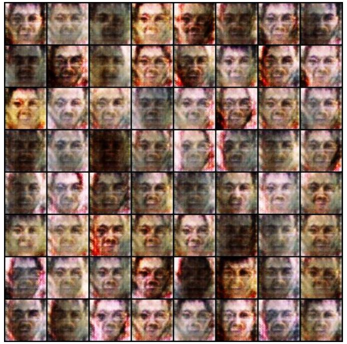
DCGAN: Experiments and Results



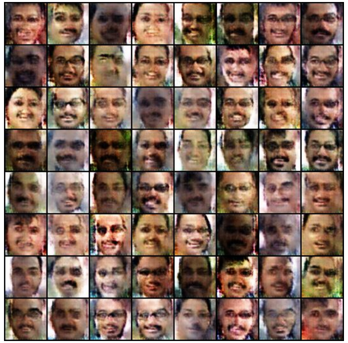
Epoch 1



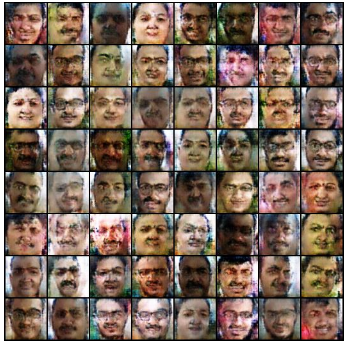
Epoch 50



Epoch 100



Epoch 150



Epoch 200

'Bindi'

Moustache

The network seems to have learned the spatial placement of facial features!

Conclusions

- GANs can generate realistic looking images by using two player non-cooperative minimax game. It has many applications such as Image super resolution, Image to Image translation (pix-to-pix), Next video frame prediction.
- Training GANs is very difficult: mode collapse, hard to achieve Nash equilibrium, vanishing gradient problem, lack of proper evaluation metric.
- Latent inference (sampling $z \sim \mathcal{P}(z)$) is inherently not possible for original GANs.
- WGAN attempts to stabilise the GAN training but weight clipping still causes instabilities.
- Gradient penalty was introduced in WGANs to further stabilise the training

Code release: https://github.com/nimRobotics/GANs (will be made public by 12 noon, May 11)

References

- Goodfellow, Ian, et al. "Generative adversarial nets." Advances in neural information processing systems. 2014.
- Arjovsky, Martin, Soumith Chintala, and Léon Bottou. "Wasserstein gan." arXiv preprint arXiv:1701.07875 (2017).
- Radford, Alec, Luke Metz, and Soumith Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks." arXiv preprint arXiv:1511.06434 (2015).
- Pieter Abbeel, Peter Chen, Jonathan Ho, Aravind Srinivas, Alex Li, Wilson Yan, Deep Unsupervised Learning (Spring 2020), UC Berkeley
- Fei-Fei Li CS231n Lecture slides, Stanford University
- Paszke, Adam, et al. "PyTorch: An imperative style, high-performance deep learning library." Advances in Neural Information Processing Systems. 2019.

Thank You!