Introduction

When evaluating the quality of synthetic data, it might be reasonable to speculate whether one could

- Train a NN to distinguish between real and synthetic data

We will call a NN designed for that purpose a *Discriminator*.

We will call the NN designed to *generate* synthetic data the *Generator*.

It's easy to train a weak Discriminator

- one that distinguishes between real data and noise (random data)

We can train a stronger Discriminator if we have access to higher quality (than noise) synthetic data.

The higher the quality of the synthetic data, the stronger the Discriminator.

But how do we construct a Generator that might be able to create synthetic data good enough to fool the Discriminator ?

Using the NN for the Discriminator

- given an input $\mathbf{x}$ created by the Generator
- we can compute the Gradient
    - of the logit (the Discriminator output indicating Real or Not Real)
    - with respect to $\mathbf{x}$
- the Generator can modify $\mathbf{x}$ using the Gradient in the direction that moves the logit toward "Real"

One can imagine an iterative process in which

- feedback from the Discriminator improves the Generator
- the resulting higher quality synthetic data from the Generator can be used to train a stronger Discriminator

This "adversarial" training is the basis for a *Generative Adversarial Network (GAN)*

Aside

The GAN (https://arxiv.org/pdf/1406.2661.pdf) was invented by Ian Goodfellow in one night, following a party at a bar (https://www.technologyreview.com/2018/02/21/145289/the-ganfather-the-man-whos-given-machines-the-gift-of-imagination/) !

Details

Notation summary

text	meaning
p_{data}	Distribution of real data
$\mathbf{x} \in p_{\text{data}}$	Real sample
p_{model}	Distribution of fake data
$\hat{\mathbf{x}}$	Fake sample
	$\hat{\mathbf{x}} \notin p_{\text{data}}$
	$\text{shape}(\hat{\mathbf{x}}) = \text{shape}(\mathbf{x})$
$\tilde{\mathbf{x}}$	Sample (real or fake)
	$\text{shape}(\tilde{\mathbf{x}}) = \text{shape}(\mathbf{x})$
D_{Θ_D}	Discriminator NN, parameterized by Θ_D
	Binary classifier: $\tilde{\mathbf{x}} \mapsto \{\text{Real}, \text{Fake}\}$
	$D_{\Theta_D}(\tilde{x}) \in \{\text{Real}, \text{Fake}\} \text{ for } \text{shape}(\tilde{\mathbf{x}}) = \text{shape}(\mathbf{x})$
$\mathbf{z}$	vector or randoms with distribution $p_{\mathbf{z}}$
G_{Θ_G}	Generator NN, parameterized by Θ_G
	$\mathbf{z} \mapsto \hat{\mathbf{x}}$
	$\text{shape}(G(\mathbf{z})) = \text{shape}(\mathbf{x})$
	$G(\mathbf{z}) \in p_{\text{model}}$

Our goal is to generate new *synthetic* examples.

Let

- $\mathbf{x}$ denote a *real* example
    - vector of length n
- p_{data} be the distribution of real examples
    - $\mathbf{x} \in p_{\text{data}}$

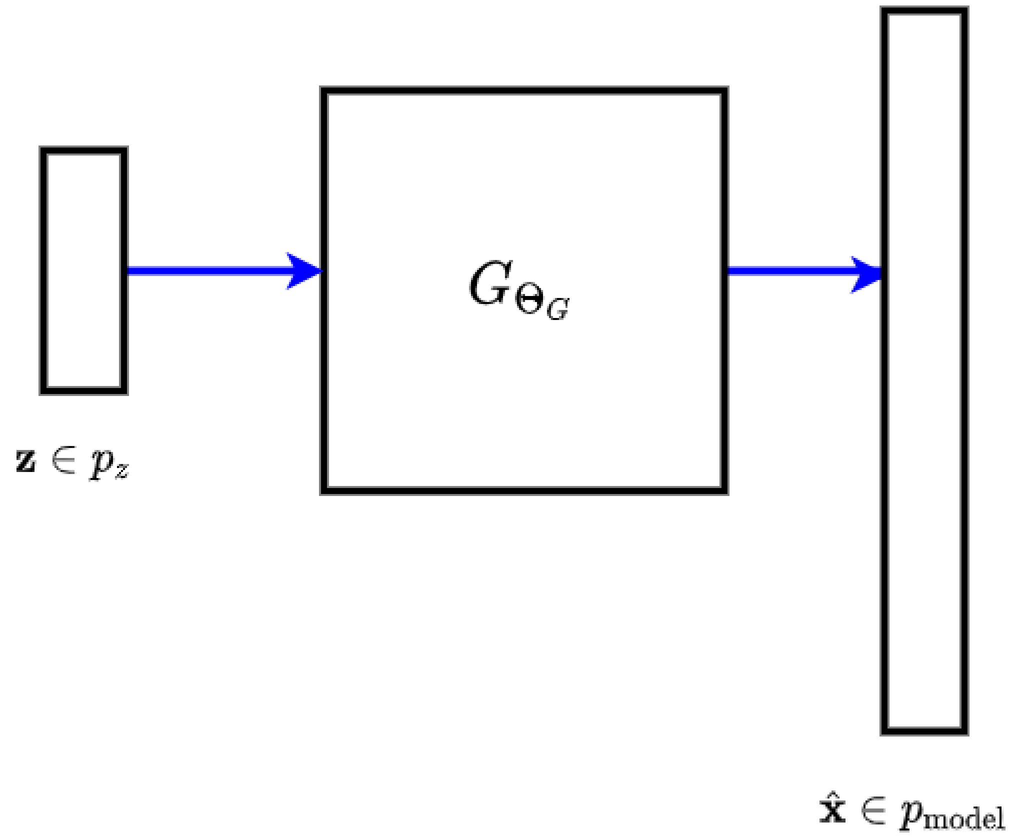
We will create a Neural Network called the *Generator*

Generator G_{Θ_G} (parameterized by Θ_G) will

- take a vector $\mathbf{z}$ of random numbers from distribution $p_{\mathbf{z}}$ as input
- and outputs $\hat{\mathbf{x}}$
- a *synthetic/fake* example
    - vector of length n

Let

- p_{model} be the distribution of fake examples


G_{Θ_G}
$\mathbf{z} \in p_z$
$\hat{\mathbf{x}} \in p_{\text{model}}$

The Generator will be paired with another Neural Network called the *Discriminator*.

The Discriminator D_{Θ_D} (parameterized by Θ_D) is a binary Classifier

- takes a vector $\tilde{\mathbf{x}} \in p_{\text{data}} \cup p_{\text{model}}$

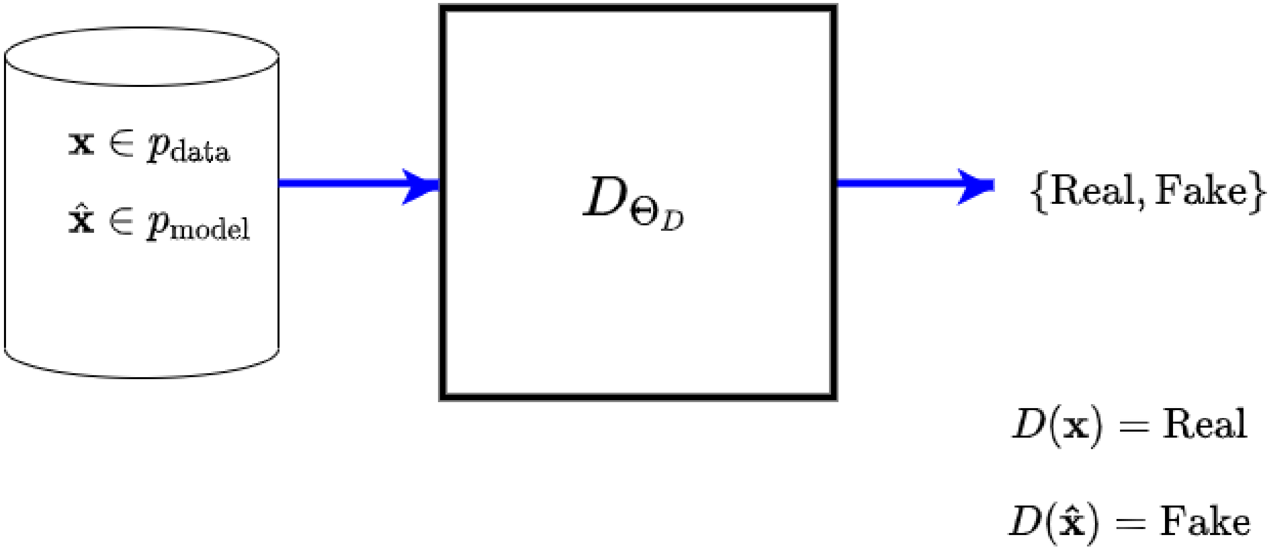
Goal of Discriminator

$$
\begin{array}{lll}
D(\tilde{\mathbf{x}}) & = & \text{Real} \quad \text{for } \tilde{\mathbf{x}} \in p_{\text{data}} \\
D(\tilde{\mathbf{x}}) & = & \text{Fake} \quad \text{for } \tilde{\mathbf{x}} \in p_{\text{model}}
\end{array}
$$

That is

- the Discriminator tries to distinguish between Real and Fake examples

GAN Discriminator



In contrast, the goal of the Generator

Goal of Generator

$$D(\hat{\mathbf{x}}) \quad = \quad \text{Real} \quad \text{for } \hat{\mathbf{x}} = G_{\Theta_G}(\mathbf{z}) \in p_{\text{model}}$$

That is

- the Generator tries to create fake examples that can fool the Discriminator into classifying as Real

How is this possible ?

We describe a training process (that updates Θ_G and Θ_D)

- That follows an *iterative* game
- Train the Discriminator to distinguish between
    - Real examples
    - and the Fake examples produced by the Generator on the prior iteration
- Train the Generator to produce examples better able to fool the updated Discriminator

Sounds reasonable, but how do we get the Generator to improve it's fakes ?

We will define loss functions

- $\mathcal{L}_G$ for the Generator
- $\mathcal{L}_D$ for the Discriminator

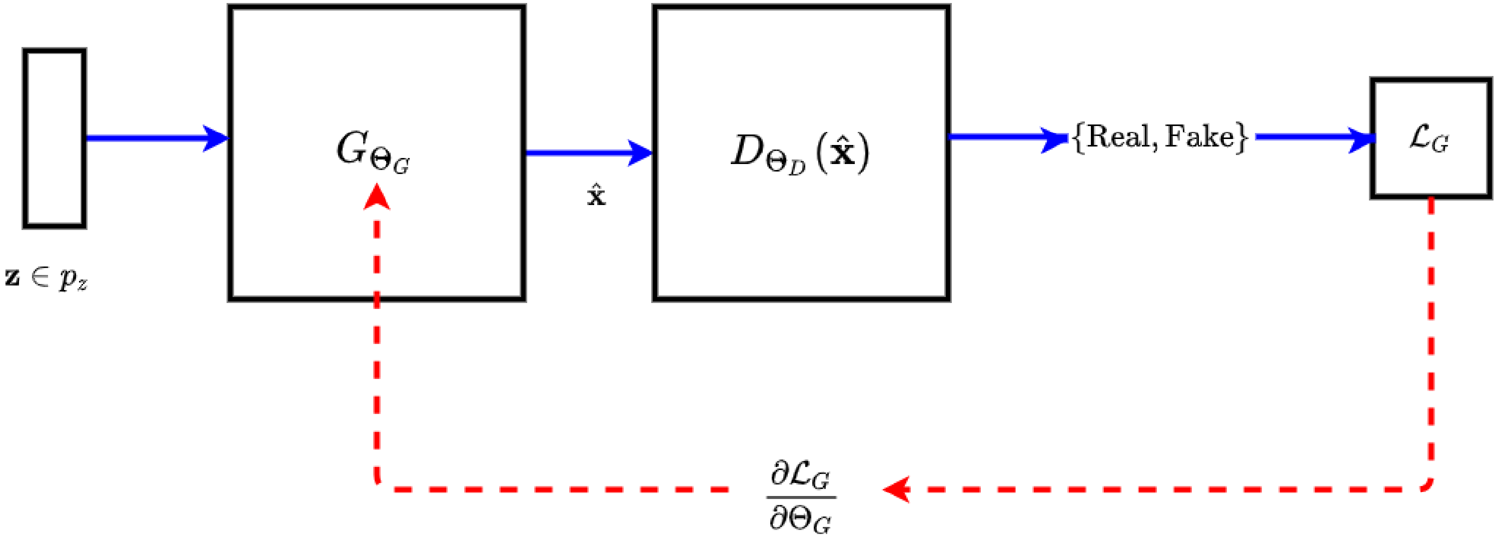
Then we can improve the Generator (parameterized by Θ_G) by Gradient Descent

- updating Θ_G by $-\frac{\partial \mathcal{L}_G}{\partial \Theta_G}$
- since Θ_G controls production of $\hat{\mathbf{x}}$, we modify Θ_G rather than $\hat{\mathbf{x}}$ directly

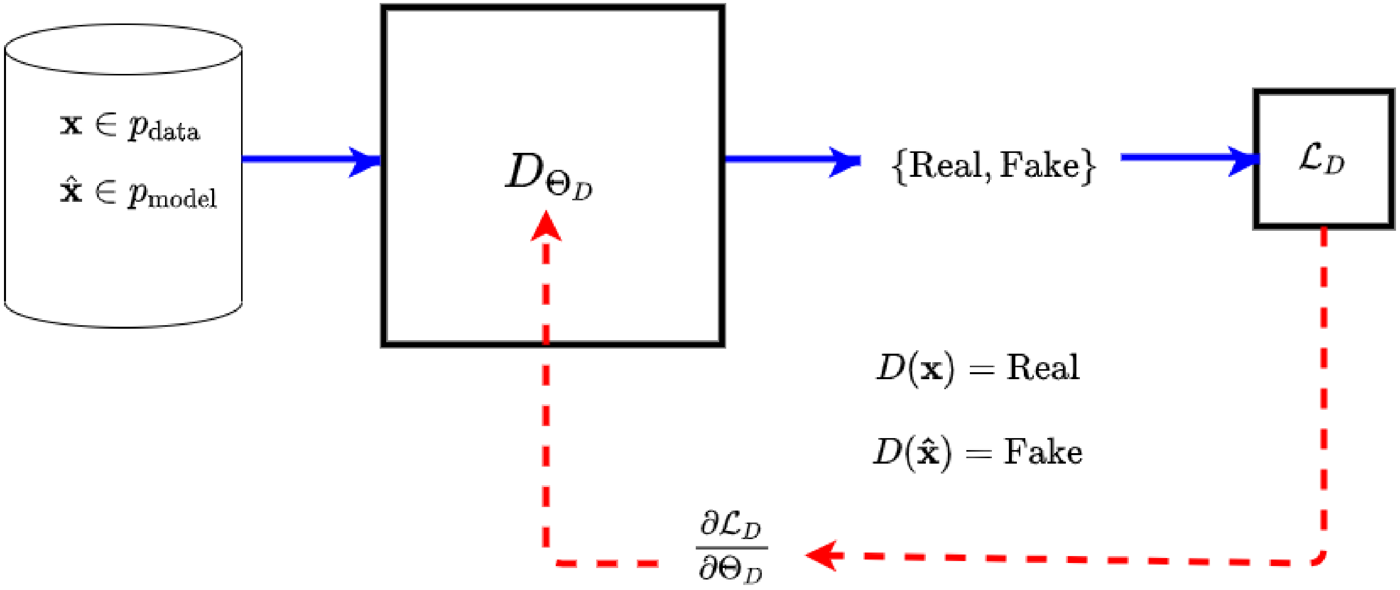
That is

- The Discriminator will indirectly give "hints" to the Generator as to why a fake example failed to fool

GAN Generator training



GAN Discriminator training



After enough rounds of the "game" we hope that the Generator and Discriminator battle to a stand-off

- the Generator produces realistic fakes
- the Discriminator has only a 50% chance of correctly labeling a fake as Fake

Loss functions

The goal of the generator can be stated as

- Creating p_{model} such that
- $p_{\text{model}} \approx p_{\text{data}}$

There are a number of ways to measure the dis-similarity of two distributions

- KL divergence
    - equivalent to Maximum Likelihood estimation
- Jensen Shannon Divergence (JSD)
- Earth Mover Distance (Wasserstein GAN)

The original paper choose the minimization of the KL divergence, so we illustrate with that measure.

To be concrete. let the Discriminator uses labels

- 1 for Real
- 0 for Fake

The Discriminator tries to maximize per example $\mathcal{L}_D$ (by minimizing the $-\mathcal{L}_D$)

$$-\mathcal{L}_D = \begin{cases} \log D(\tilde{\mathbf{x}}) & \text{when } \tilde{\mathbf{x}} \in p_{\text{data}} \\ 1 - \log D(\tilde{\mathbf{x}}) & \text{when } \tilde{\mathbf{x}} \in p_{\text{model}} \end{cases}$$

That is

- Classify real $\mathbf{x}$ as Real
- Classify fake $\hat{\mathbf{x}}$ as Fake

In training the Discriminator, we present it with batches of examples

- half real, half fake

The Discriminator tries to maximize (over the batch) the negative of the loss over the batch

$$
\begin{aligned}
\mathcal{L}_D &= -\left(\frac{1}{2}\mathbf{E}_{\mathbf{x^{(i)}} \in p_{\text{data}}} \log D(\mathbf{x^{(i)}}) + \frac{1}{2}\mathbf{E}_{z \in P_z} \log(1 - D(G(\mathbf{z})))\right) \\
&= -\left(\frac{1}{2}\mathbf{E}_{\mathbf{x^{(i)}} \in p_{\text{data}}} \log D(\mathbf{x^{(i)}}) + \frac{1}{2}\mathbf{E}_{\mathbf{x^{(i)}} \in p_{\text{model}}} \log\left(1 - D(\mathbf{x^{(i)}})\right)\right) \\
&= -\frac{1}{2} \sum_{\mathbf{x^{(i)}} \in p_{\text{data}}} p_{\text{data}}(\mathbf{x^{(i)}}) \log D(\mathbf{x^{(i)}}) - \frac{1}{2} \sum_{\mathbf{x^{(i)}} \in p_{\text{model}}} p_{\text{model}}(\mathbf{x^{(i)}}) \log(1 -
\end{aligned}
$$

You will recognize this term as Binary Cross Entropy (BCE)

- hence, you will see BCE used as the Loss Function in the code

$$\mathcal{L}_G = -\mathcal{L}_D \quad \text{Zero sum game}$$

The per-example Loss for the Generator is

$$\mathcal{L}_G = 1 - \log D(G(\mathbf{z}))$$

which is minimized when the fake example

$$D(G(\mathbf{z})) = 1$$

That is

- the Discriminator mis-classifies the fake example as Real

The Generator takes batches of $\mathbf{z}$ (and hence sees only fake examples, not an even mix of real and fake as does the Discriminator.

Since the game is zero sum

$$= \mathcal{L}_G = -\mathcal{L}_D$$

and you will similarly see BCE as the Loss for the Generator

- except the "true" labels passed to BCE will be an array of "Real"
- as opposed to a mix of "Real" and "Fake" labels in the BCE of the Discriminator

So the iterative game seeks to solve a minimax problem

$$\min_G \max_D \left(\mathbb{E}_{\mathbf{x} \in p_{\text{data}}} \log D(\mathbf{x}) + \mathbb{E}_{\mathbf{z} \in p_z} (1 - \log D(G(\mathbf{z}))) \right)$$

- D tries to
  - make $D(\mathbf{x})$ big: correctly classify (with high probability) real $\mathbf{x}$
  - and $D(G(\mathbf{z}))$ small: correctly classify (with low probability) fake $G(\mathbf{z}))$
- G tries to
  - make $D(G(\mathbf{z}))$ high: fool D into a high probability for a fake

Note that the Generator improves

- by updating Θ_G
- so as to increase $D(G(\mathbf{z}))$
    - the mis-classification of the fake as Real

Optimal Discriminator Loss

Can minimize per example $\mathcal{L}_D$ wrt $D(\mathbf{x})$ by taking derivative and setting to 0

$$
\begin{aligned}
\frac{\partial \mathcal{L}_D}{\partial D(\mathbf{x})} &= -\frac{1}{2}\left(p_{\text{data}}(\mathbf{x}) * \frac{1}{\log_e 10}\frac{1}{D(\mathbf{x})} + p_{\text{model}}(\mathbf{x}) * \frac{1}{\log_e 10}\frac{1}{1-D(\mathbf{x})} * -1\right) \quad \text{Defi} \\
& \quad \text{Der} \\
&= -\frac{1}{2*\log_e 10}\frac{p_{\text{data}}(\mathbf{x})*(1-D(\mathbf{x}))-p_{\text{model}}(\mathbf{x})*D(\mathbf{x})}{D(\mathbf{x})*(1-D(\mathbf{x}))} \\
&= \frac{1}{c}\frac{p_{\text{data}}(\mathbf{x})-D(\mathbf{x})(p_{\text{model}}(\mathbf{x})+p_{\text{data}}(\mathbf{x}))}{D(\mathbf{x})*(1-D(\mathbf{x}))} \\
\frac{\partial \mathcal{L}_D}{\partial D(\mathbf{x})} &= 0 \quad \mapsto D^*(\mathbf{x}) = \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{model}}(\mathbf{x})+p_{\text{data}}(\mathbf{x})}
\end{aligned}
$$

So the optimal Discriminator succeeds with probability

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

The optimal Generator results in

$$p_{\text{model}}(\mathbf{x}) = p_{\text{data}}(\mathbf{x})$$

Thus, if the minimax optimization succeeds

$$D^*(\mathbf{x}) = \frac{1}{2}$$

Nothing better than a coin toss !

Training

We will train Generator G_{Θ_G} Discriminator D_{Θ_D} by turns

- creating sequence of updated parameters
  - $\Theta_{G,(1)} \ldots \Theta_{G,(T)}$
  - $\Theta_{D,(1)} \ldots \Theta_{D,(T)}$
- Trained *competitively*

Competitive training

Iteration t

- Train $D_{\Theta_{D,(t-1)}}$ on samples
  - $\tilde{\mathbf{x}} \in p_{\text{data}} \cup p_{\text{model},(t-1)}$
    - where $G_{\Theta_{G,(t-1)}}(\mathbf{z}) \in p_{\text{model},(t-1)}$
  - Update $\Theta_{D,(t-1)}$ to $\Theta_{D,(t)}$ via gradient $\frac{\partial \mathcal{L}_D}{\partial \Theta_{D,(t-1)}}$
    - D is a maximizer of $\int_{\mathbf{x} \in p_{\text{data}}} \log D(\mathbf{x}) + \int_{\mathbf{z} \in p_{\mathbf{z}}} \log (\, 1 - D(G(\mathbf{z}))\,)$
- Train $G_{\Theta_{G,(t-1)}}$ on random samples $\mathbf{z}$
  - Create samples $\hat{\mathbf{x}}_{(t)} \in G_{\Theta_{G,(t-1)}}(\mathbf{z}) \in p_{\text{model}}$
  - Have Discriminator $D_{\Theta_{D,(t)}}$ evaluate $D_{\Theta_{D,(t)}}(\hat{\mathbf{x}}_{(t)})$
  - Update $\Theta_{G,(t-1)}$ to $\Theta_{G,(t)}$ via gradient $\frac{\partial \mathcal{L}_G}{\partial \Theta_{G,(t-1)}}$
    - G is a minimizer of $\int_{\mathbf{z} \in p_{\mathbf{z}}} \log(\, 1 - D(G(\mathbf{z}))\,)$
      - i.e., want $D(G(\mathbf{z}))$ to be high
  - May update G multiple times per update of D

Training code for a simple GAN

Here (https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/dcgan_overriding_train_step.ipynb#scrollTo=A is the code for the training step of a simple GAN.

Issues

Although the description of GAN training as an adversarial game is appealing, actually getting training to find a stable equilibrium is difficult in practice.

Vanishing Gradient

Early in training, the Discriminator has the advantage

- it has been trained to distinguish real input from noise
- the parameters of the Generator are uninitialized
    - Generator needs feedback from Discriminator in order to learn direction for improvement

What happens if the Discriminator is "too good" ?

- $D(\hat{\mathbf{x}}) = 0$ for all $\hat{\mathbf{x}} \in p_{\text{model}}$

With absolute certainty that every $\hat{\mathbf{x}}$ from the Generator is Fake, the gradient is zero (or near zero)

- Generator can't learn (weight updates near zero)

So we don't want the Discriminator to be too good, too early in training.

Mode Collapse

We condition the Generator on random $\mathbf{z}$ so that it will produce diverse $\hat{\mathbf{x}}$.

Sometimes, the Generator is only able to create a single (or small number) $\hat{\mathbf{x}}'$ that is good enough to fool the Discriminator.

In this case: the Generator may learn to ignore input $\mathbf{z}$ and *only* produce $\hat{\mathbf{x}}'$.

Hard to achieve equilibrium

The optimal solution is the Nash equilibrium of the minimax problem

$$\min_{G} \max_{D} \left(\mathbb{E}_{\mathbf{x} \in p_{\text{data}}} \log D(\mathbf{x}) + \mathbb{E}_{\mathbf{z} \in p_z} (1 - \log D(G(\mathbf{z})) \right)$$

However: the objective of Neural Network training is minimization of a Loss.

There is no guarantee that Gradient Descent will always converge to the Nash equilibrium

- [See this paper, section 3 (https://arxiv.org/pdf/1412.6515.pdf)](https://arxiv.org/pdf/1412.6515.pdf)
- [Also, see this paper, section 3 (https://arxiv.org/pdf/1606.03498.pdf)](https://arxiv.org/pdf/1606.03498.pdf)

The gradients are partials with respect to the denominator, *holding everything else constant*.

But everything is *not* constant: the Generator and Discriminator are each modifying their weights.

- So the weight update of the Generator may not result in improvement if the simultaneous weight update of the Discriminator moves in the opposite direction.

An often cited example 2 player game illustrates the point

- Player 1 seeks to minimize product $x * y$ by manipulating x
$$\frac{\partial x * y}{\partial x} = y$$
$$x \rightarrow (x - y) \quad \text{update x by negative of gradient}$$
- Player 2 seeks to minimize product $-x * y$ by manipulating x
$$\frac{\partial (-x * y)}{\partial y} = -x$$
$$y \rightarrow (y + x) \qquad \text{update y by negative of gradient}$$

If x, y have opposite signs, then the update causes them to either both increase or both decreases.

- one can show by experiment that each update causes x, y to oscillate in increasing magnitude.

Code

- GAN on Colab (https://keras.io/examples/generative/dcgan_overriding_train_step/)
- Wasserstein GAN with Gradient Penalty (https://keras.io/examples/generative/wgan_gp/#create-the-wgangp-model)

References

- Goodfellow (https://arxiv.org/pdf/1406.2661.pdf)
- Huszar (https://arxiv.org/pdf/1511.05101.pdf)
- Wasserstein GAN paper (https://arxiv.org/pdf/1701.07875.pdf)

Good blog, submitted as paper

- Weng blog (https://arxiv.org/pdf/1904.08994.pdf)