Introduction

We introduce an architecture with two Neural Network (NN) components for generating synthetic examples

- Discriminator
- Generator

The Discriminator

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*.

- a binary Classifier into $\{\text{Real}, \text{Not Real/Fake}\}$

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.

The Generator

The purpose of the Generator NN is to generate synthetic examples.

The goal is for these examples to be of such high quality

- that they "fool" the Discriminator
- into classifying them as "Real", not "Not Real"

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

This seems particularly hard

- an untrained Generator will produce nonsense outputs during the initial epochs of training
- which will be easily rejected as Not Real by the Discriminator

How can we adapt the weights of the Generator to learn to create higher-quality output ?

The answer:

- use the output of the Discriminator
- to provide feedback as to why it was classified as Not Real
- and use the feedback to improve the Generator

The Classifier NN for the Discriminator

- creates a logit (pre-probability): a proxy for the probability of the input being Real

This logit can be used to extract information for the Generator to improve !

Given a synthetic output $\mathbf{x}$ created by the Generator

- feed $\mathbf{x}$ to the Discriminator D
- compute $D(\mathbf{x})$: the probability (really the logit) assigned by D to $\mathbf{x}$ being Real

The Gradient

$$\frac{\partial\, D(\mathbf{x})}{\partial \Theta_G}$$

is the direction in which the weights Θ_G of the Generator needs to be incremented

- in order to increase the probability that the Discriminator classifies synthetic $\mathbf{x}$ as Real

We use Gradient Ascent (Gradient_ascent.ipynb#Use-Case:-Find-the-maximally-activating-input) on $D(\mathbf{x})$ to improve the probability that synthetic $\mathbf{x}$ will fool the Discriminator

A two player game between Generator and Discriminator

The feedback to the Generator allows it to improve the chances of creating a synthetic example of high enough quality to fool the Discriminator.

Thus, the Generator can iteratively use the Discriminator's feedback until it fools the Discriminator !

This is easy since the initial Discriminator was weak

- trained to distinguish between noise and Real examples

Still the quality of the synthetic examples

- although good enough to fool a weak Discriminator
- will still be low quality in absolute terms
    - marginally better than noise

The solution: make the Discriminator stronger !

Train it to distinguish between

- Real examples
- the synthetic examples created by the Generator that had previously fooled the Discriminator

The more powerful Discriminator can then be used

- to create a more powerful Generator

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

The GAN is a two-player game in which the Generator and Discriminator each try to dominate the other.

- the Generator wins when it fools the Discriminator
- the Discriminator wins when it can't be fooled

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
	$\hat{\mathbf{x}} \not\in p_{\text{data}}$
	$\text{shape}(\hat{\mathbf{x}}) = \text{shape}(\mathbf{x})$
$\tilde{\mathbf{x}}$	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
    - $\mathbf{x} \in p_{\text{data}}$

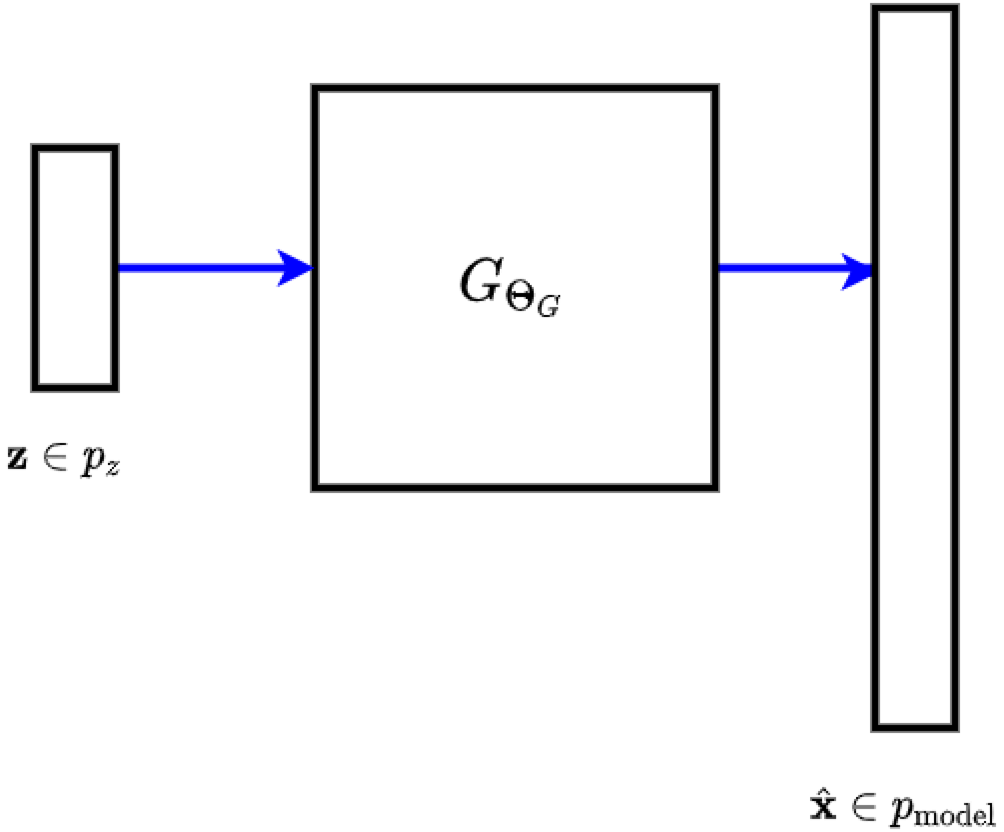
We will create a Neural Network called the *Generator*

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 $\tilde{\mathbf{x}} \in p_{\text{data}} \cup p_{\text{model}}$

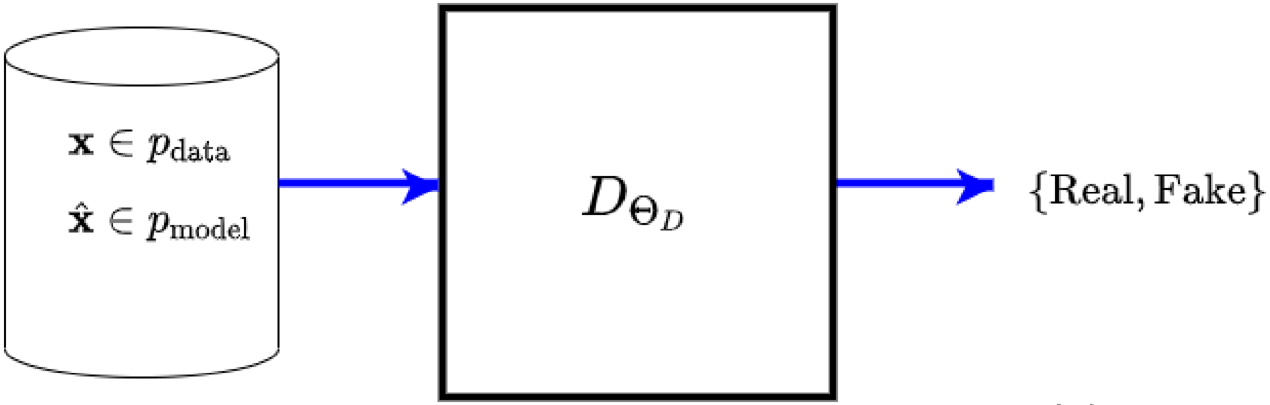
Goal of Discriminator

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

That is

- the Discriminator tries to distinguish between Real and Fake examples

GAN Discriminator



$$D(\mathbf{x}) = \text{Real}$$

$$D(\hat{\mathbf{x}}) = \text{Fake}$$

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 ?

It is all through the Loss functions that we will define:

- $\mathcal{L}_G$ for the Generator
- $\mathcal{L}_D$ for the Discriminator
    - technically: the Discriminator is *maximizing*: -1 * $\mathcal{L}_D$

Aside

Yet another illustration of the premise of the Advanced Neural Networks course

- creating a **loss function** that expresses the semantics of your task
- is the key to solving the task
- as opposed to the architecture of the NN

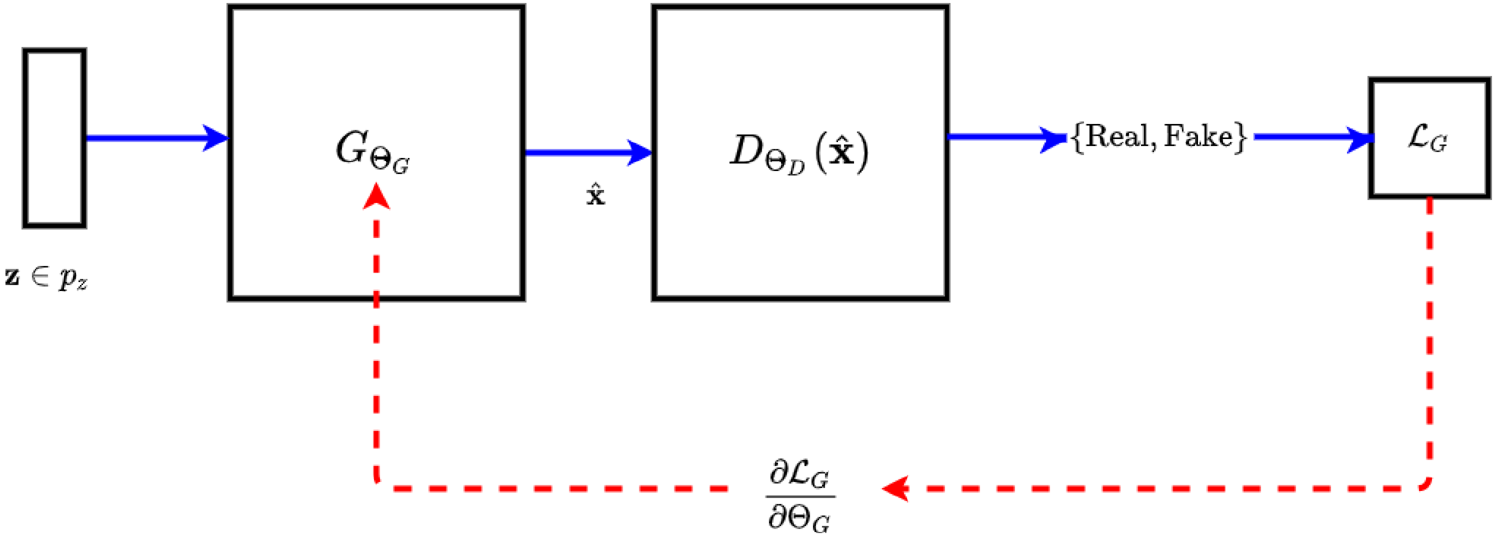
As we will see, the Loss function for the Generator will

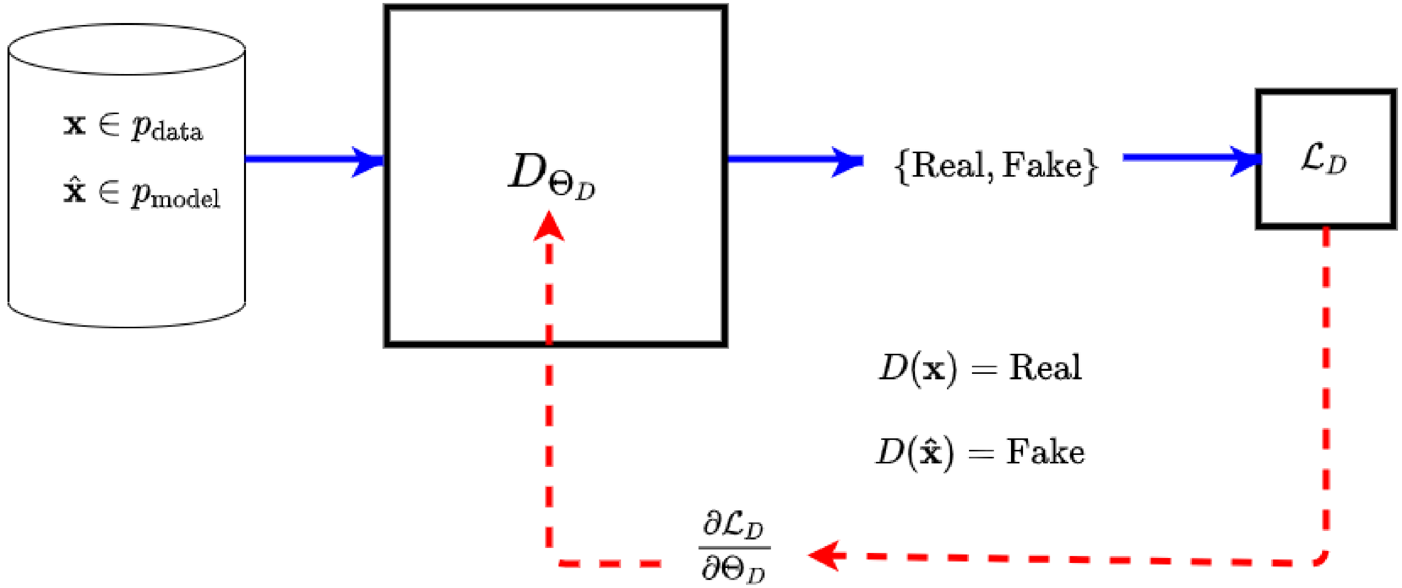
- be based on how confidently the Discriminator detected the fake
- updating Θ_G
    - by $- \frac{\partial \mathcal{L}_G}{\partial \Theta_G}$
    - to reduce loss
- means we make the Discriminator less confident that this example is fake

In essence

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

GAN Generator training




$\mathbf{x} \in p_{\text{data}}$
$\hat{\mathbf{x}} \in p_{\text{model}}$
D_{Θ_D}
$\{\text{Real}, \text{Fake}\}$
$\mathcal{L}_D$
$D(\mathbf{x}) = \text{Real}$
$D(\hat{\mathbf{x}}) = \text{Fake}$
$\frac{\partial \mathcal{L}_D}{\partial \Theta_D}$

After enough rounds of the iterative "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 Not Real

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 synthetic $\hat{\mathbf{x}}$ as Not Real

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

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

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

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

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 synthetic example as Real

This is the "hint" from the Discriminator

- that enables the Generator to improve the quality of its synthetic examples

The Generator takes batches of $\mathbf{z}$ (and hence sees only syntheic examples, not an even mix of real and synthetic 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 "Not Real" labels in the BCE of the Discriminator

Note

- all the examples passed by the Generator to the Discriminator are synthetic
- so the "true labels" passed to the Loss Function
    - are not accurate
    - but are meant to achieve the goal of fooling the Discriminator into classifying examples as "Real"

That is: the objective of the Generator is to produce examples that the Discriminator labels as "Real".

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 the derivative and setting to 0

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

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 $\int_{\mathbf{x} \in p_{\text{data}}} \log D(\mathbf{x}) + \int_{\mathbf{z} \in p_{\mathbf{z}}} \log \left(1 - D(G(\mathbf{z})) \right)$
- 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: Highlights

The key to implementing a GAN

- is a custom training step
- that trains *both* the Discriminator and the Generator

Here is the detailed custom training step (http://keras.io/examples/generative/dcgan_overriding_train_step/#override-trainstep)

We break down this step in our module on Advanced Keras (Keras_Advanced.ipynb#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)
- Also, see this paper, section 3 (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 y
$$\frac{\partial(-x*y)}{\partial y} = -x$$
$$y \rightarrow (y + x) \qquad \text{update y by negative of gradient}$$

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)

In [2]:

```python
print("Done")
```

```
Done
```