BOSTON UNIVERSITY

Deep Learning for Data Science DS 542

Lecture 20
Adversarial Examples and
Generative Adversarial Networks

Slides originally by Thomas Gardos.
Images from Understanding Deep Learning unless otherwise cited.

Last Time:

- Contrastive Learning
  - Train useful encodings so similar inputs are similar and different inputs are different
- Vector Databases
  - Useful database for nearest neighbors and highest cosine similarity
- Retrieval Augmented Generation
  - Use encoding vectors to find related documents to improve context for generation

Today

- Adversarial inputs
- Generative adversarial networks (GANs)

Two Intriguing Properties of Neural Networks

1. The output of a hidden layer is better thought of as a vector space than individually important vectors.

- No privileged basis.
- Next layer just takes linear combinations anyway, so why expect one of the outputs to be special?

These are examples maximizing particular outputs.

(a) Unit sensitive to lower round stroke.

(b) Unit sensitive to upper round stroke, or lower straight stroke.

(c) Unit senstive to left, upper round stroke.

(d) Unit senstive to diagonal straight stroke.

Figure 1: An MNIST experiment. The figure shows images that maximize the activation of various units (maximum stimulation in the natural basis direction). Images within each row share semantic properties.

Intriguing properties of neural networks (2014)

Two Intriguing Properties of Neural Networks

1. The output of a hidden layer is better thought of as a vector space than individually important vectors.

- No privileged basis.
- Next layer just takes linear combinations anyway, so why expect one of the outputs to be special?

These are examples maximizing particular outputs.

(a) Unit sensitive to white flowers.

(b) Unit sensitive to postures.

(c) Unit senstive to round, spiky flowers.

(d) Unit senstive to round green or yellow objects.

Figure 3: Experiment performed on ImageNet. Images stimulating single unit most (maximum stimulation in natural basis direction). Images within each row share many semantic properties.

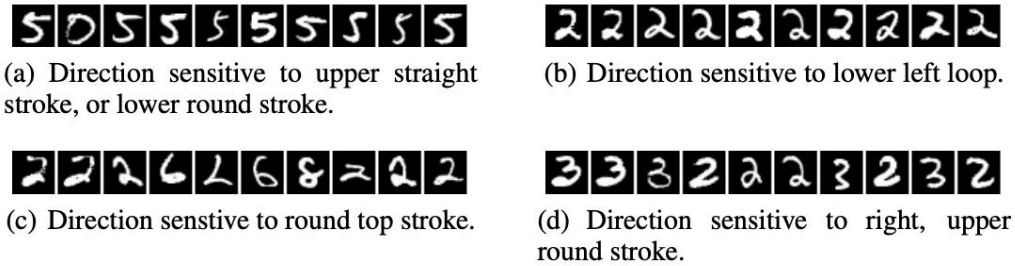
[Intriguing properties of neural networks](#) (2014)

Two Intriguing Properties of Neural Networks

1. The output of a hidden layer is better thought of as a vector space than individually important vectors.

- No privileged basis.
- Next layer just takes linear combinations anyway, so why expect one of the outputs to be special?

These are examples maximizing random directions.



(a) Direction sensitive to upper straight stroke, or lower round stroke.

(b) Direction sensitive to lower left loop.

(c) Direction senstive to round top stroke.

(d) Direction sensitive to right, upper round stroke.

Figure 2: An MNIST experiment. The figure shows images that maximize the activations in a random direction (maximum stimulation in a random basis). Images within each row share semantic properties.

Intriguing properties of neural networks (2014)

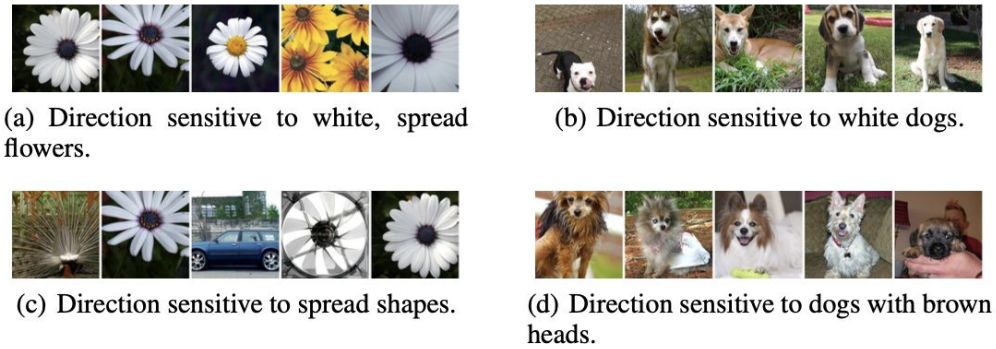
Two Intriguing Properties of Neural Networks

1. The output of a hidden layer is better thought of as a vector space than individually important vectors.

- No privileged basis.
- Next layer just takes linear combinations anyway, so why expect one of the outputs to be special?

[Intriguing properties of neural networks](#) (2014)

These are examples maximizing random directions.



(a) Direction sensitive to white, spread flowers.

(b) Direction sensitive to white dogs.

(c) Direction sensitive to spread shapes.

(d) Direction sensitive to dogs with brown heads.

Figure 4: Experiment performed on ImageNet. Images giving rise to maximum activations in a random direction (maximum stimulation in a random basis). Images within each row share many semantic properties.

Two Intriguing Properties of Neural Networks

2. It is easy to fool neural networks with small changes to the inputs.

- It was previously “obvious” that you could tweak inputs to change the outputs.
- It was a huge surprise that the tweaks could be so small!

[Intriguing properties of neural networks](#) (2014)

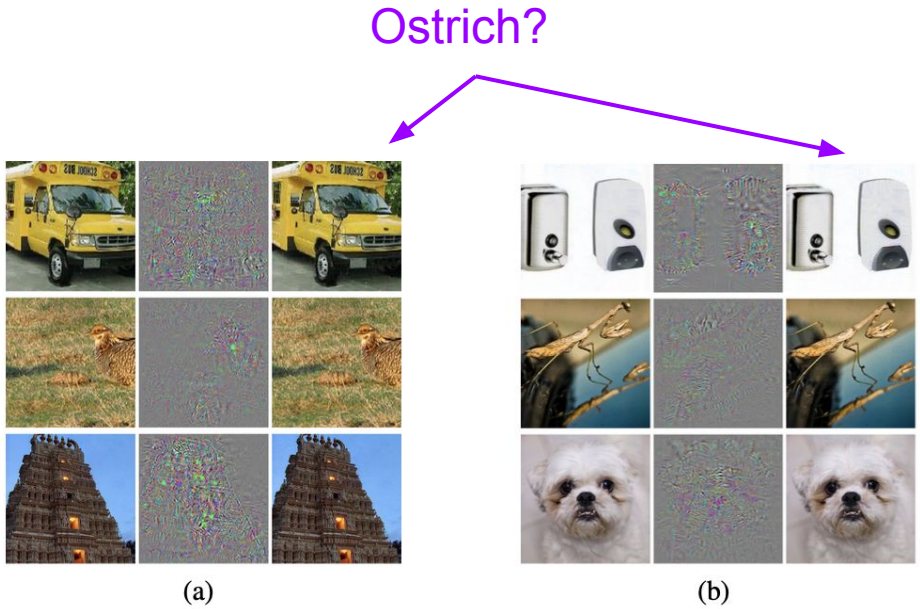


Figure 5: Adversarial examples generated for AlexNet [9].(Left) is a correctly predicted sample, (center) difference between correct image, and image predicted incorrectly magnified by 10x (values shifted by 128 and clamped), (right) adversarial example. All images in the right column are predicted to be an *“ostrich, Struthio camelus”*. Average distortion based on 64 examples is 0.006508. Plase refer to `http://goo.gl/huaGPb` for full resolution images. The examples are strictly randomly chosen. There is not any postselection involved.

Two Intriguing Properties of Neural Networks

2. It is easy to fool neural networks with small changes to the inputs.

- It was previously “obvious” that you could tweak inputs to change the outputs.
- It was a huge surprise that the tweaks could be so small!

(a) (b)

Figure 6: Adversarial examples for QuocNet [10]. A binary car classifier was trained on top of the last layer features without fine-tuning. The randomly chosen examples on the left are recognized correctly as cars, while the images in the middle are not recognized. The rightmost column is the magnified absolute value of the difference between the two images.

Intriguing properties of neural networks (2014)

Two Intriguing Properties of Neural Networks

2. It is easy to fool neural networks with small changes to the inputs.

- It was previously “obvious” that you could tweak inputs to change the outputs.
- It was a huge surprise that the tweaks could be so small!
- **And these tweaks work across models???**

Intriguing properties of neural networks (2014)

“In addition, the specific nature of these perturbations is not a random artifact of learning: the same perturbation can cause a different network, that was trained on a different subset of the dataset, to misclassify the same input.”

Takeaways

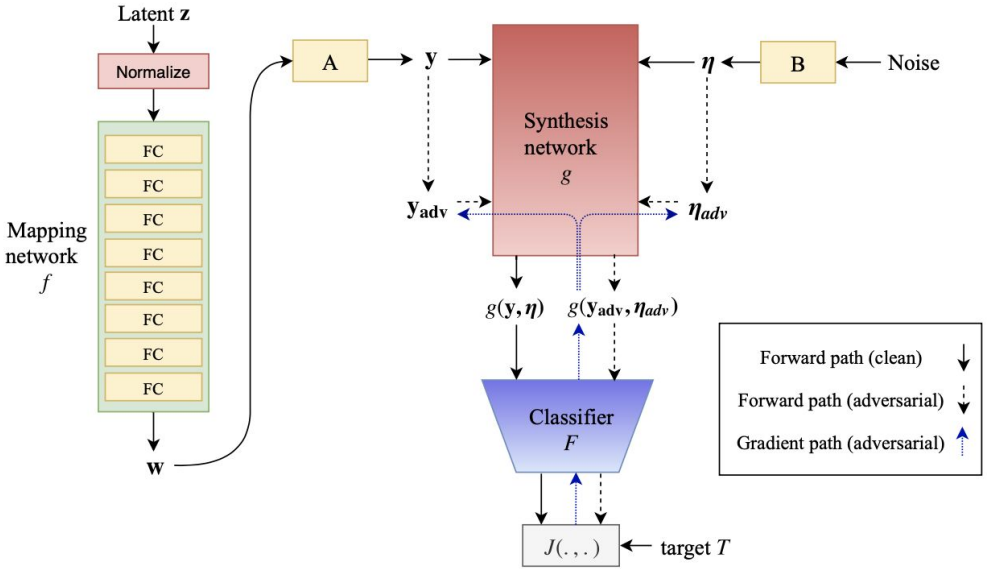
1. Neural network output space is not standardized to match our intuitions.
    a. “No privileged basis”
    b. Model ends up targeting vector spaces in the hidden layers?
2. Neural networks are surprisingly easy to fool with small input tweaks.
    a. These tweaks are usually imperceptible for images.
    b. They transfer across different data sets and models.
    c. Possibly related to common vector space representations?

Intriguing properties of neural networks (2014)

Adversarial Training

Idea: use adversarial examples to train more robust classifiers.

- Works in practice, at least for the kinds of adversarial examples tested.
  - But not necessarily for other types
  - Will see soon this often comes at the cost of accuracy for clean examples.



[Robustness and Generalization via Generative Adversarial Training] (2021)

What makes a feature useful?

- Features can be useful if they are correlated with a target.
- If the correlation is negative, flip the sign.

ρ-useful features: For a given distribution $\mathcal{D}$, we call a feature f *ρ-useful* ($\rho > 0$) if it is correlated with the true label in expectation, that is if

$$\mathbb{E}_{(x,y)\sim\mathcal{D}}[y \cdot f(x)] \geq \rho. \tag{1}$$

Adversarial Examples Are Not Bugs, They Are Features (2019)

What makes a feature robust?

- A feature is robust if it stays useful under adversarial permutations.

γ-robustly useful features: Suppose we have a ρ-useful feature f ($\rho_{\mathcal{D}}(f) > 0$). We refer to f as a *robust feature* (formally a γ-robustly useful feature for $\gamma > 0$) if, under adversarial perturbation (for some specified set of valid perturbations Δ), f remains γ-useful. Formally, if we have that

$$\mathbb{E}_{(x,y)\sim\mathcal{D}}\left[\inf_{\delta\in\Delta(x)} y \cdot f(x+\delta)\right] \geq \gamma. \tag{2}$$

Adversarial Examples Are Not Bugs, They Are Features (2019)

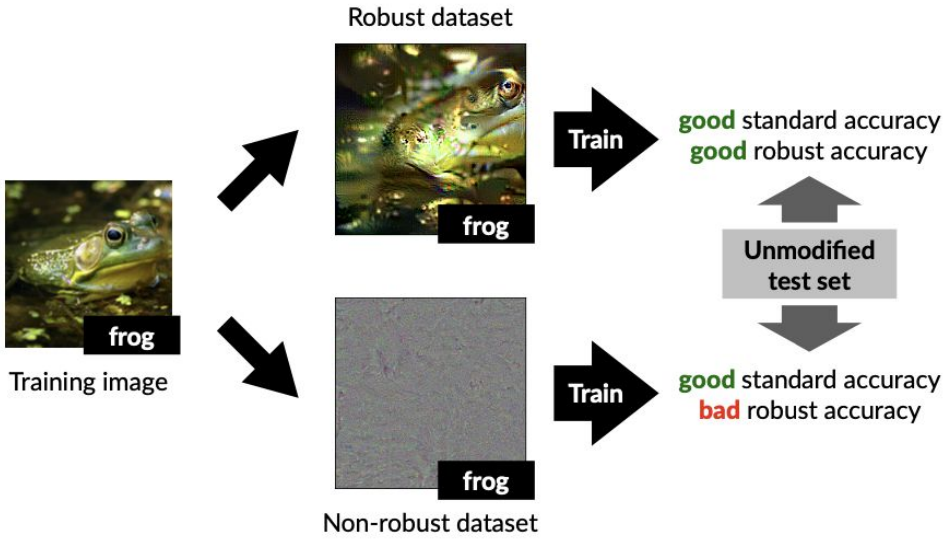
What about useful but non-robust features?

- Training will learn useful features even if they are not robust.

Useful, non-robust features: A *useful, non-robust feature* is a feature which is ρ-useful for some ρ bounded away from zero, but is not a γ-robust feature for any $\gamma \geq 0$. These features help with classification in the standard setting, but may hinder accuracy in the adversarial setting, as the correlation with the label can be flipped.

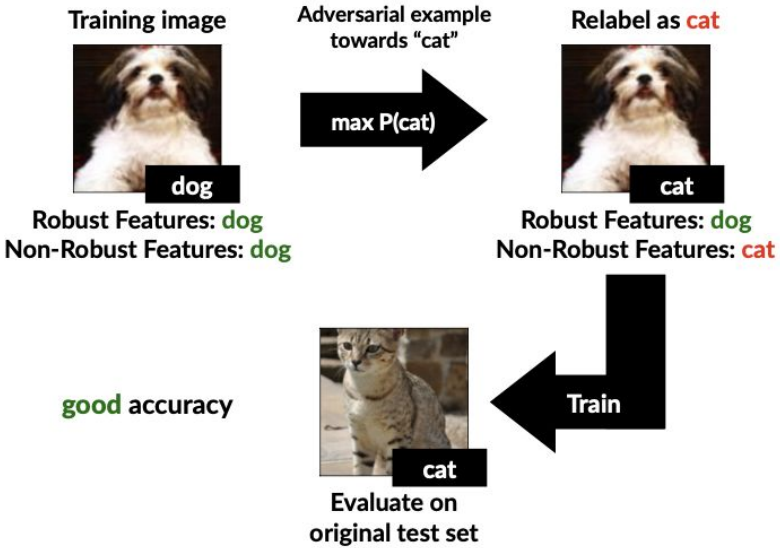
Adversarial Examples Are Not Bugs, They Are Features (2019)

Training Data can be Tweaked to be More or Less Robust



[Adversarial Examples Are Not Bugs, They Are Features](#) (2019)

Silly Training Tricks



[Adversarial Examples Are Not Bugs, They Are Features](#) (2019)

Takeaways

- Adversarial inputs take advantage of real features in the training data.
- Can try to remove these “non-robust” features, but may cost performance.

[Adversarial Examples Are Not Bugs, They Are Features](#) (2019)

Generative Adversarial Networks

TLDR: use adversarial examples idea to build a really good image generator.

Possibly an evolutionary dead end, but still interesting because

- Clever training procedure
- First really good image generation technique
- Helped identify a lot of possible pitfalls and sometimes techniques to fix them

Imagine…

A function f that maps samples from normally distributed noise to images.

Can we train a function g that distinguishes the output of f from a real image?

Imagine…

A function f that maps samples from normally distributed noise to images.

Can we train a function g that distinguishes the output of f from a real image?

- If f is randomly initialized but untrained, then g should be easy to train.

Imagine…

A function f that maps samples from normally distributed noise to images.

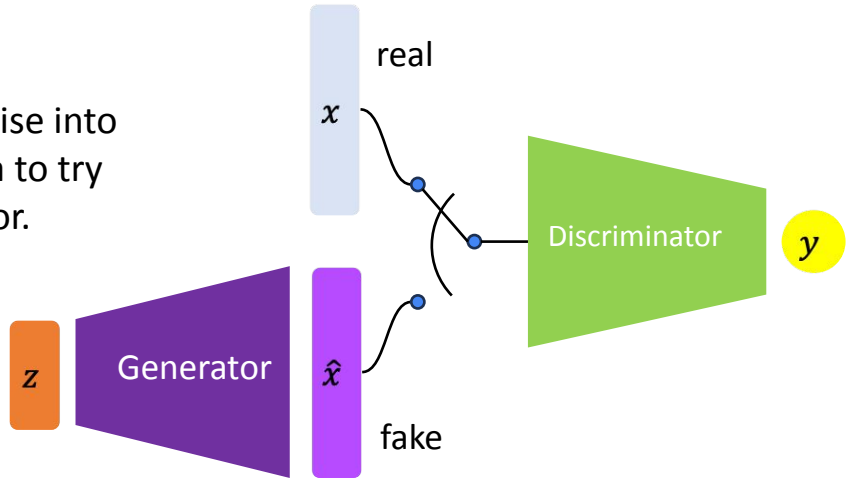
Can we train a function g that distinguishes the output of f from a real image?

- If f is randomly initialized but untrained, then g should be easy to train.
- What if we then train f to fool g?

Generative Adversarial Networks

Train a generative model to try to fool a “discriminator” model.

The **generator** turns noise into an imitation of the data to try to trick the discriminator.

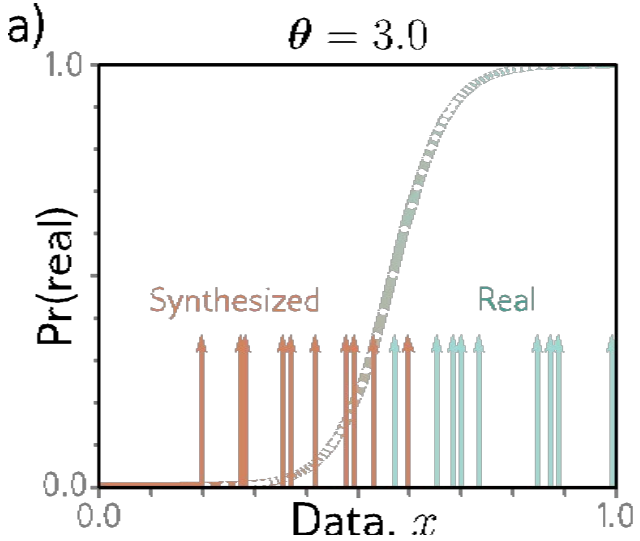


The **discriminator** tries to identify real data from fakes created by the generator.

fake
real

GAN example

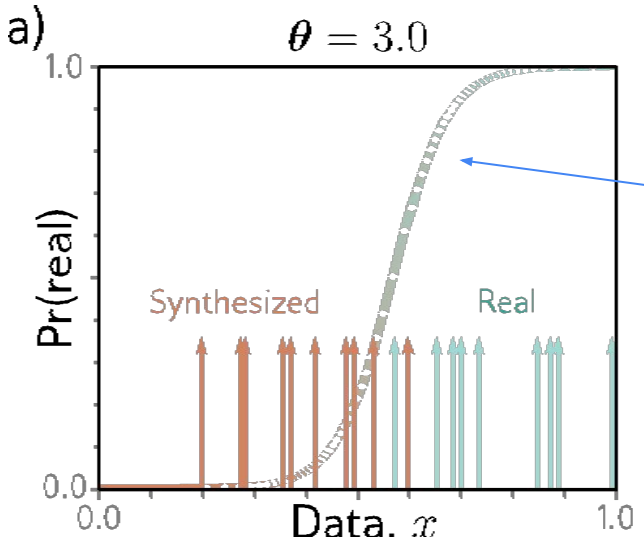
$$x_j^* = g[z_j, \theta] = z_j + \theta$$



- We take examples from a real distribution (e.g. shifted standard gaussian)
- We generate synthesized samples, z_j, from a standard gaussian and shift by θ.
- Train a classifier on the data

GAN example

$$x_j^* = g[z_j, \theta] = z_j + \theta$$



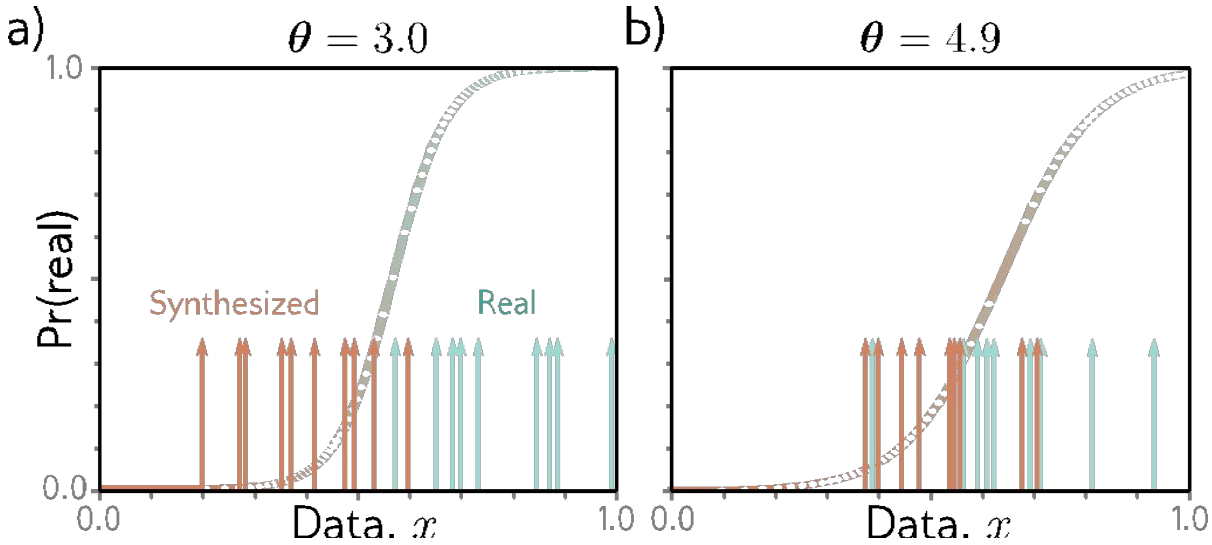
$f[\bullet, \phi]$

Discriminator

- Train the discriminator
- using logistic regression parameterized by ϕ
- as a binary classifier on the data
- e.g. $\begin{cases} \text{real if } f[\cdot] \geq .5 \\ \text{fake if } f[\cdot] < .5 \end{cases}$

GAN example

$$x_j^* = \mathrm{g}[z_j, \theta] = z_j + \theta$$

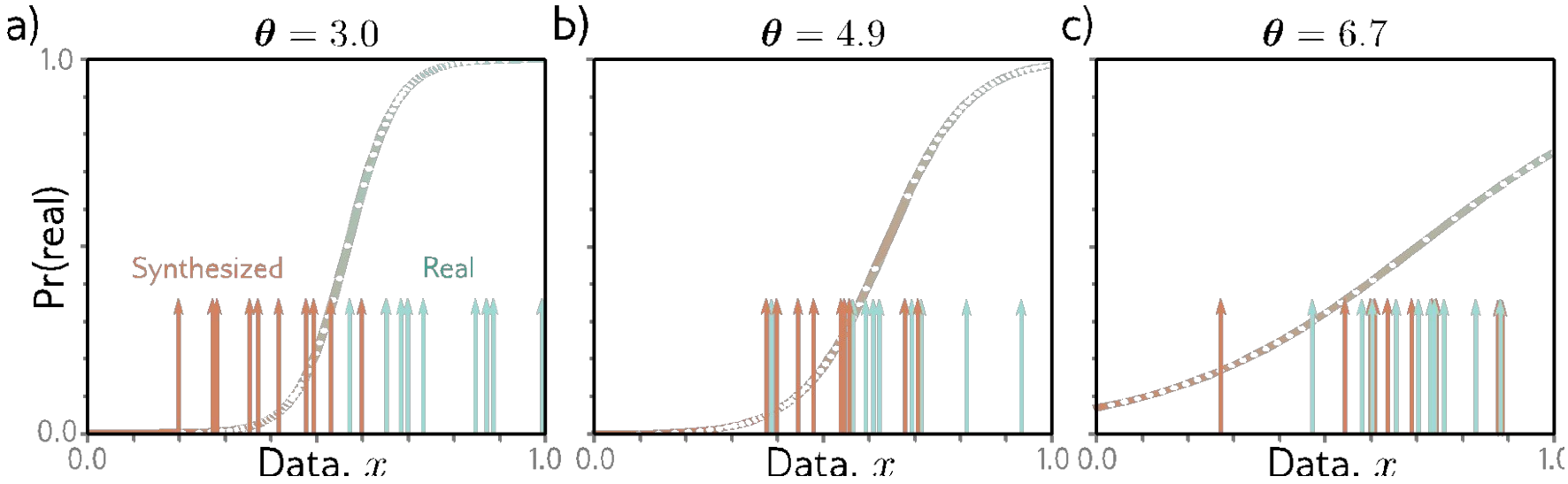


- Train the **generator** to update θ in order to *increase* the loss on the discriminator
- Then train the **discriminator** to *decrease* the loss

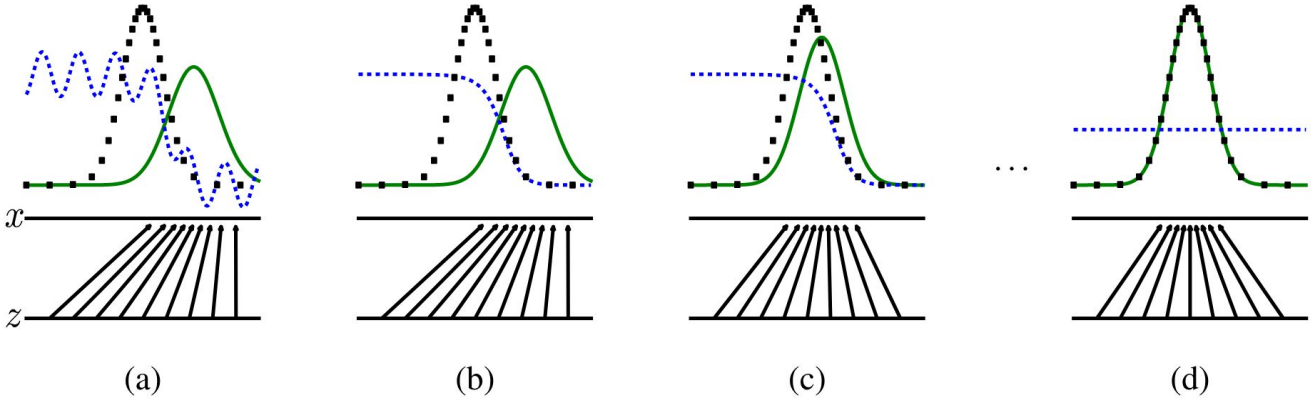
GAN example

$$x_j^* = g[z_j, \theta] = z_j + \theta$$

- Keep repeating till the discriminator does no better than random chance



Trained to completion



- z: uniform latent variable
- x: samples according to a (green solid) generative distribution
- black dotted curve: real data distribution
- blue dashed curve: discriminator

4.1 Global Optimality of $p_g = p_{\text{data}}$

We first consider the optimal discriminator D for any given generator G.

Proposition 1. *For G fixed, the optimal discriminator D is*

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

I. Goodfellow *et al.*, "Generative Adversarial Nets," 2014

GAN cost function

Discriminator uses standard cross entropy loss (see Section 5.4 – binary classification loss): :

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[\sum_i -(1-y_i) \log\Big[1 - \operatorname{sig}[\mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}]]\Big] - y_i \log\Big[\operatorname{sig}[\mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}]]\Big] \right]$$

GAN cost function

Discriminator uses standard cross entropy loss (see Section 5.4 – binary classification loss):

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[\sum_i -(1-y_i) \log\Big[1 - \operatorname{sig}[\mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}]]\Big] - y_i \log\Big[\operatorname{sig}[\mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}]]\Big] \right]$$

Generated samples, $\mathbf{x}_i^*,\ y_i = 0$, and for real examples, $\mathbf{x}_i,\ y_i = 1$:

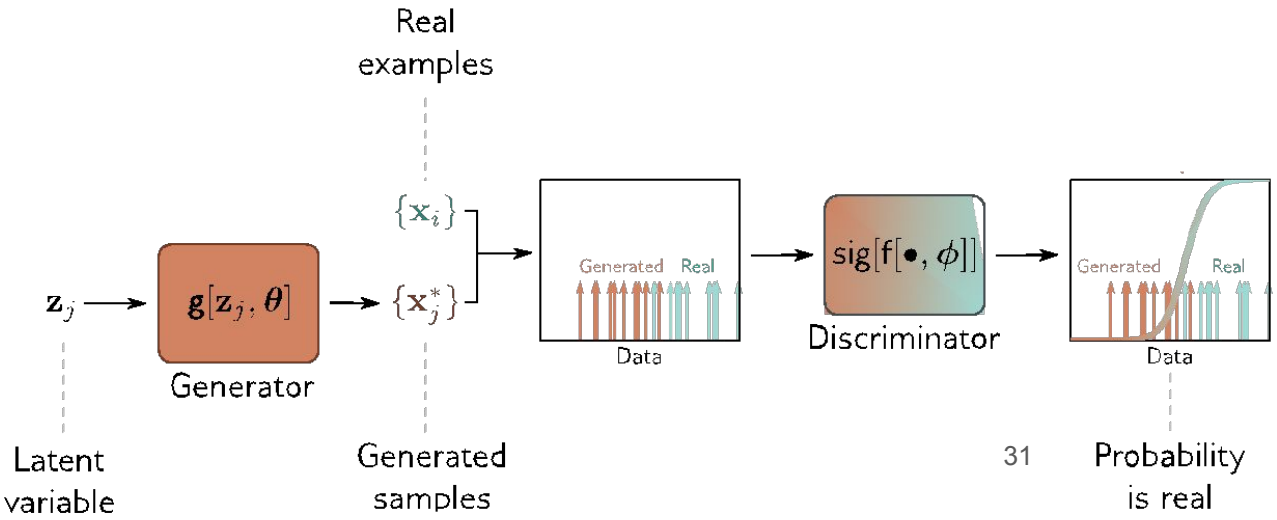
$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[\sum_j -\log\Big[1 - \operatorname{sig}[\mathrm{f}[\mathbf{x}_j^*, \boldsymbol{\phi}]]\Big] - \sum_i \log\Big[\operatorname{sig}[\mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}]]\Big] \right]$$

These are *generated* samples so $y_j = 0$

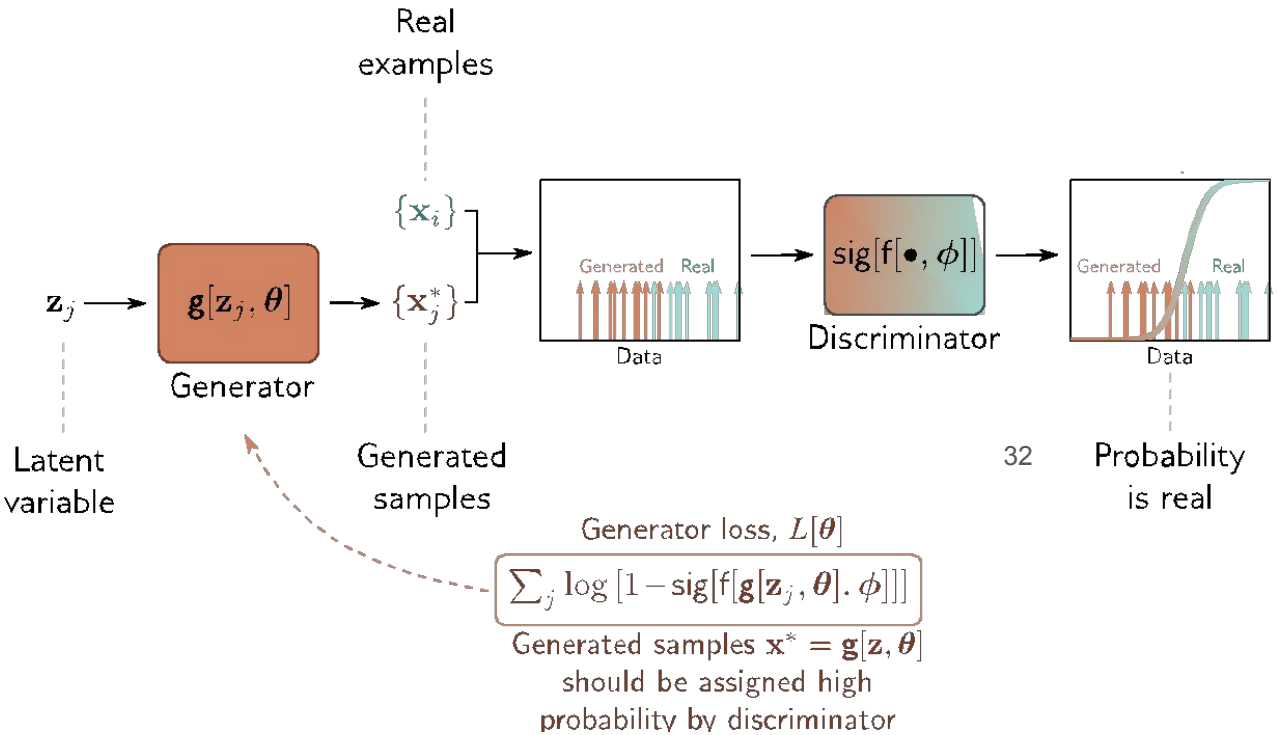
These are *real* samples so $y_i = 1$

We can separate into two summations that separately index over the generated samples and the real samples.

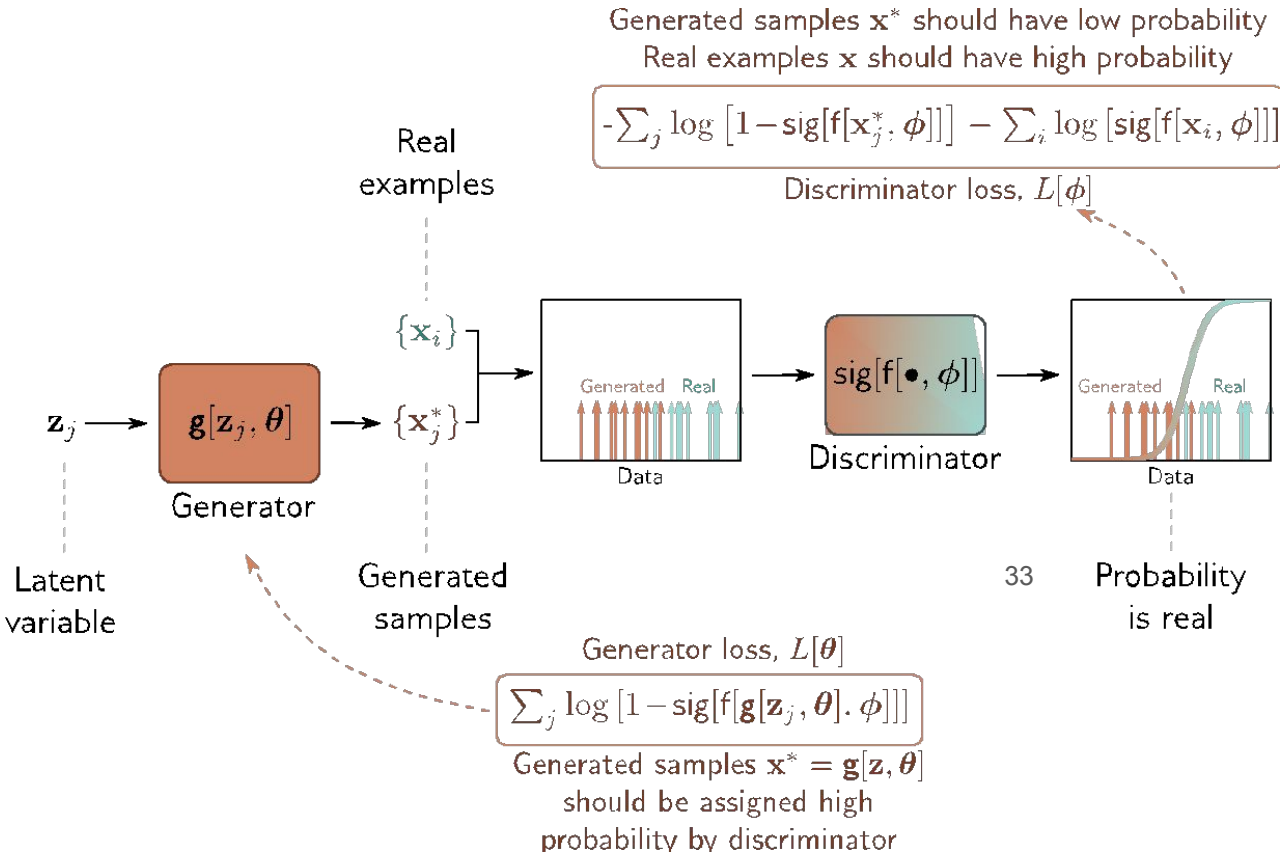
GAN loss function



GAN loss function



GAN loss function



GAN cost function

Discriminator uses standard cross entropy loss:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\text{argmin}} \left[\sum_i -(1-y_i) \log\Big[1 - \text{sig}[\text{f}[\mathbf{x}_i, \boldsymbol{\phi}]]\Big] - y_i \log\Big[\text{sig}[\text{f}[\mathbf{x}_i, \boldsymbol{\phi}]]\Big] \right]$$

Discriminator: generated samples, y = 0, real examples, y = 1:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\text{argmin}} \left[\sum_j -\log\Big[1 - \text{sig}[\text{f}[\mathbf{x}^*_j, \boldsymbol{\phi}]]\Big] - \sum_i \log\Big[\text{sig}[\text{f}[\mathbf{x}_i, \boldsymbol{\phi}]]\Big] \right]$$

Generator loss: make generated samples more likely under discriminator (i.e. make discriminator loss larger)

$$\hat{\boldsymbol{\phi}}, \hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\text{argmax}} \left[\underset{\boldsymbol{\phi}}{\text{argmin}} \left[\sum_j -\log\Big[1 - \text{sig}[\text{f}[\mathbf{g}[\mathbf{z}_j, \boldsymbol{\theta}], \boldsymbol{\phi}]]\Big] - \sum_i \log\Big[\text{sig}[\text{f}[\mathbf{x}_i, \boldsymbol{\phi}]]\Big] \right] \right]$$

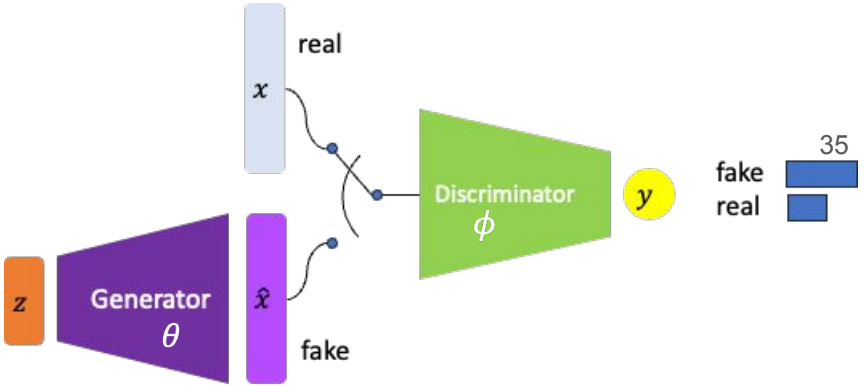
substituted the generator function for the generated sample

GAN Cost function

$$\hat{\phi}, \hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \left[\underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[\sum_j -\log\left[1 - \operatorname{sig}[\mathrm{f}[\mathbf{g}[\mathbf{z}_j, \boldsymbol{\theta}], \boldsymbol{\phi}]]\right] - \sum_i \log\left[\operatorname{sig}[\mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}]]\right] \right] \right]$$

The **discriminator** parameters, ϕ, are manipulated to *minimize* the loss function

The **generator** parameters, θ, are manipulated to *maximize* the loss function.



GAN Cost function

$$\hat{\phi}, \hat{\theta} = \underset{\theta}{\operatorname{argmax}} \left[\underset{\phi}{\operatorname{argmin}} \left[\sum_j -\log\left[1 - \operatorname{sig}[\mathrm{f}[\mathbf{g}[\mathbf{z}_j, \theta], \phi]]\right] - \sum_i \log\left[\operatorname{sig}[\mathrm{f}[\mathbf{x}_i, \phi]]\right] \right] \right]$$

The **discriminator** parameters, ϕ, are manipulated to *minimize* the loss function

The **generator** parameters, θ, are manipulated to *maximize* the loss function.

Can divide into two parts:

discriminator loss: $$L[\phi] = \sum_j -\log\left[1 - \operatorname{sig}[\mathrm{f}[\mathbf{g}[\mathbf{z}_j, \theta], \phi]]\right] - \sum_i \log\left[\operatorname{sig}[\mathrm{f}[\mathbf{x}_i, \phi]]\right]$$

negated **generator** loss: $$L[\theta] = \sum_j \log\left[1 - \operatorname{sig}[\mathrm{f}[\mathbf{g}[\mathbf{z}_j, \theta], \phi]]\right]$$

The 2nd term is constant w.r.t. θ (gradient $\partial\mathcal{L}/\partial\theta = 0$) so we can drop it)

Generator does not need access to real examples.

GAN Training Flow Pseudo Python

```
for c_gan_iter in range(n_gan_iters):  # GAN Iterations

    # Run generator to produce synthesized data
    x_syn = generator(z, theta)

    # Update/train the discriminator
    phi = update_discriminator(x_real, x_syn, n_iter_discrim, phi)

    # Update/train the generator
    theta = update_generator(z, theta, n_iter_gen, phi)
```

Convergence Proof Theory

4.2 Convergence of Algorithm 1

Proposition 2. *If G and D have enough capacity, and at each step of Algorithm 1, the discriminator is allowed to reach its optimum given G, and p_g is updated so as to improve the criterion*

$$\mathbb{E}_{\boldsymbol{x}\sim p_{data}}[\log D_G^*(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{x}\sim p_g}[\log(1 - D_G^*(\boldsymbol{x}))]$$

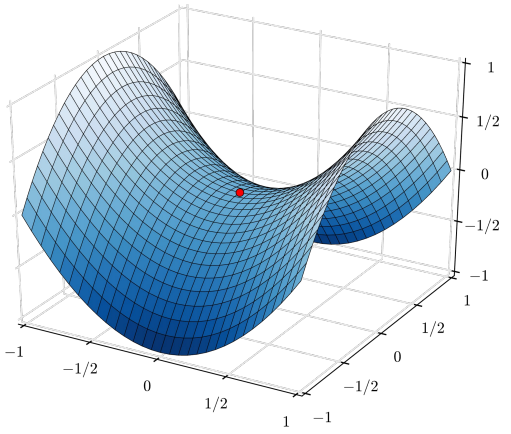
then p_g converges to p_{data}

Generative Adversarial Nets (2014)

Convergence Practice

$$\hat{\phi}, \hat{\theta} = \underset{\theta}{\operatorname{argmax}} \left[\underset{\phi}{\operatorname{argmin}} \left[\sum_j -\log\left[1 - \operatorname{sig}[\mathrm{f}[\mathbf{g}[\mathbf{z}_j, \boldsymbol{\theta}], \boldsymbol{\phi}]]\right] - \sum_i \log\left[\operatorname{sig}[\mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}]]\right] \right] \right]$$

- The solution is the *Nash equilibrium*
- It lays at a saddle point
- Is inherently unstable

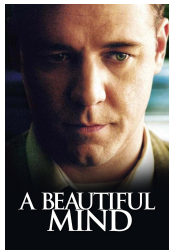


One dimension is discriminator parameter. Other is generator parameter.

Nash equilibrium

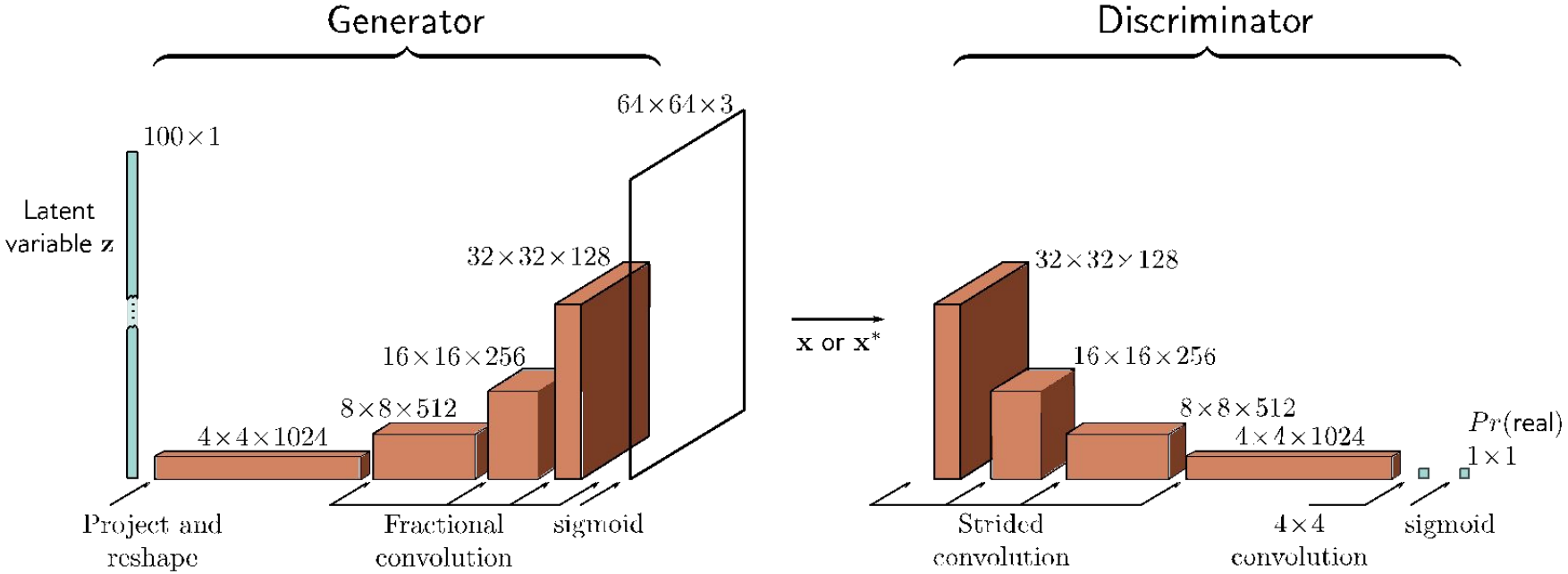
In game theory, the Nash equilibrium, named after the mathematician John Nash, is the most common way to define the solution of a non-cooperative game involving two or more players.

...each player is assumed to know the equilibrium strategies of the other players, and no one has anything to gain by changing only one's own strategy. Wikipedia



Deep Convolutional (DC) GAN

- Early GAN specialized in image generation



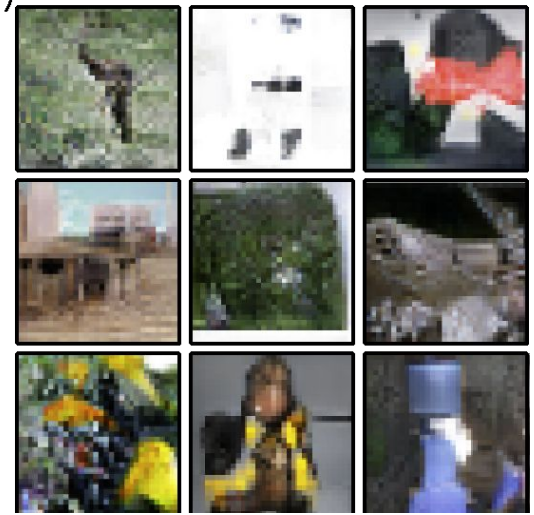
Radford et al, “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.”, 2015

DC GAN Results

a)

Trained on a faces dataset.

b)



Trained on ImageNet dataset.

c)

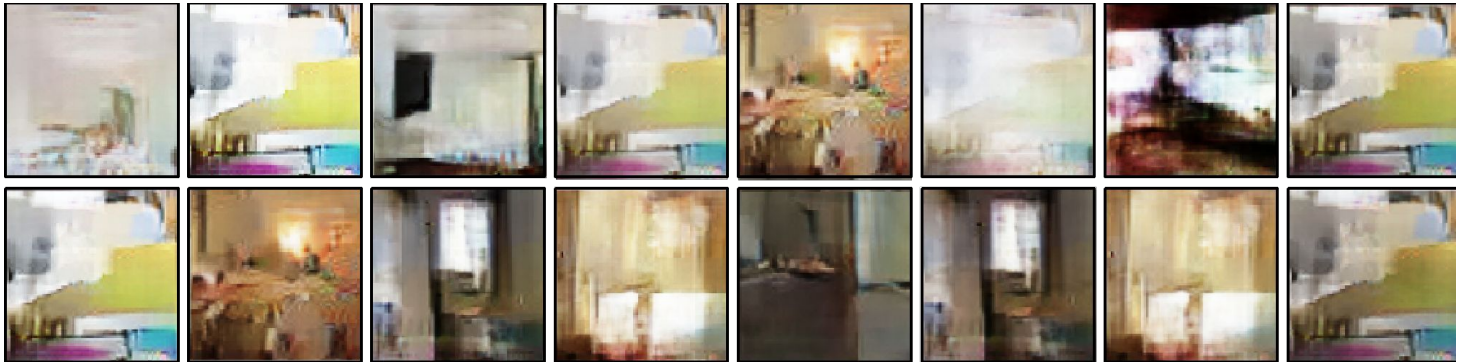
Trained on LSUN dataset.

The LSUN classification dataset contains 10 scene categories, such as dining room, bedroom, chicken, outdoor church, and so on.

Common Failures with GANs

Mode Dropping: Only represent a subset of the training distribution.

Mode Collapse: Extreme case where the generator mostly ignores the latent variable and collapses all samples to a few points



GAN Performance and Distribution Distance

$$D_{JS}\left[Pr(\mathbf{x}^*) \,||\, Pr(\mathbf{x})\right] = \underbrace{\frac{1}{2}D_{KL}\left[Pr(\mathbf{x}^*)\left|\right|\frac{Pr(\mathbf{x}^*)+Pr(\mathbf{x})}{2}\right]}_{\text{quality}} + \underbrace{\frac{1}{2}D_{KL}\left[Pr(\mathbf{x})\left|\right|\frac{Pr(\mathbf{x}^*)+Pr(\mathbf{x})}{2}\right]}_{\text{coverage}}$$

Summary of lengthy analysis in §15.2.1 “Analysis of GAN loss function”

Can be rewritten in terms of dissimilarities between *generated* and *real* probability distributions.

Two important takeaways:

Quality: Generated samples need to occur where real samples are

Coverage: Where there is concentrations of real samples, there should be good representation from generated samples

We can conclude that:

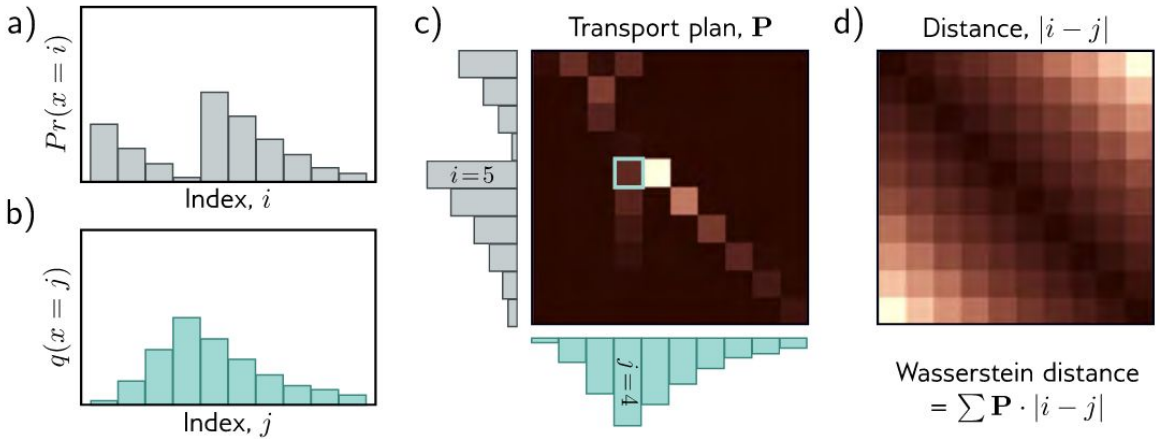
(i) the GAN loss can be interpreted in terms of distances between probability distributions and that

(ii) the gradient of this distance becomes zero when the generated samples are too easy to distinguish from the real examples.

We need a distance metric with better properties.

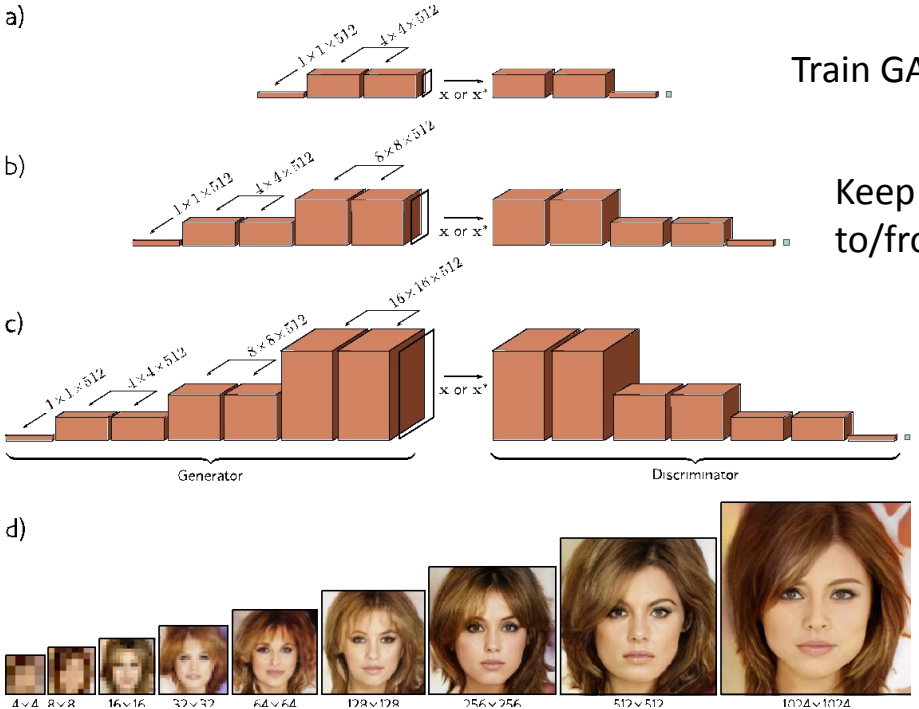
Wasserstein Distance (for continuous distributions)
Earth Mover's Distance (for discrete probabilities)

- The quantity of work required to transport the probability mass from one distribution to create the other.
- Use linear programming to find an optimal "transport plan" that minimizes $\Sigma\, \mathbf{P} \cdot |i - j|$



See 15.2.4 Wasserstein distance for discrete distributions

Trick 1: Progressive growing



Train GAN to generate and discriminate 4x4 images

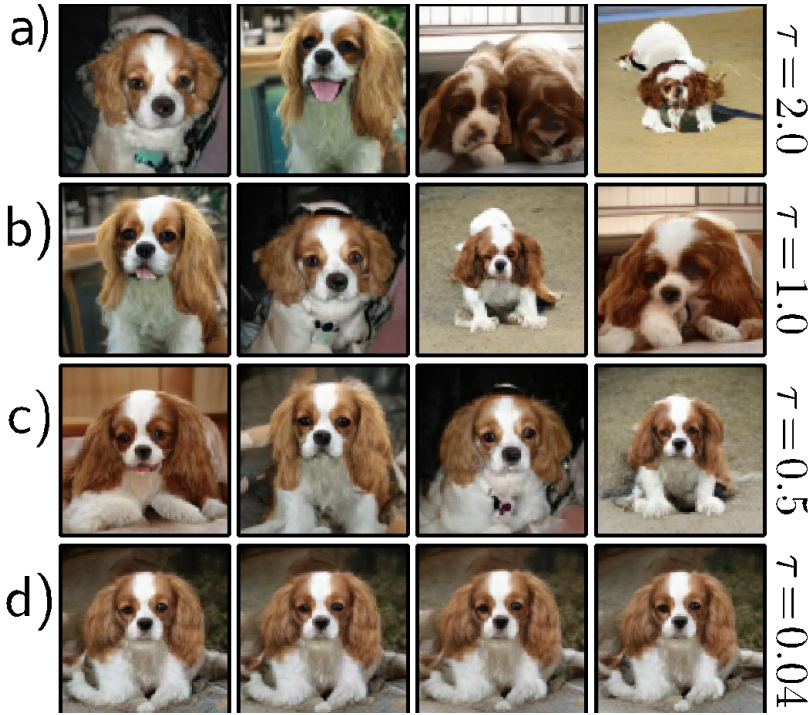
Keep weights from step (a), add layers to get to/from 8x8 images and continue training GAN

Add layers to get to 16x16 and continue to train.

Repeat above steps to get to high resolution.

Wolf (2021), Kerras (2018)

Trick 2: Truncation



- Only choose random values of latent variables that are less than a threshold τ distance from the mean of the latent variables.
- Reduces variation but improves quality

Interpolation

Well-behaved latent space: Every latent variable z should correspond to a plausible data example x and smooth changes in z should correspond to smooth changes in x.

StyleGAN (2019-2021)

GAN development focused on quality.

- Integrated styles into generation process.
- Analysis using more visually oriented evaluation (Fréchet inception distance)
- Later models systematically worked on known artifacts in the generation process.

A Style-Based Generator Architecture for Generative Adversarial Networks (2019)

Analyzing and Improving the Image Quality of StyleGAN (2019)

Alias-Free Generative Adversarial Networks (2021)

StyleGAN

Considered a very specific notion of style…

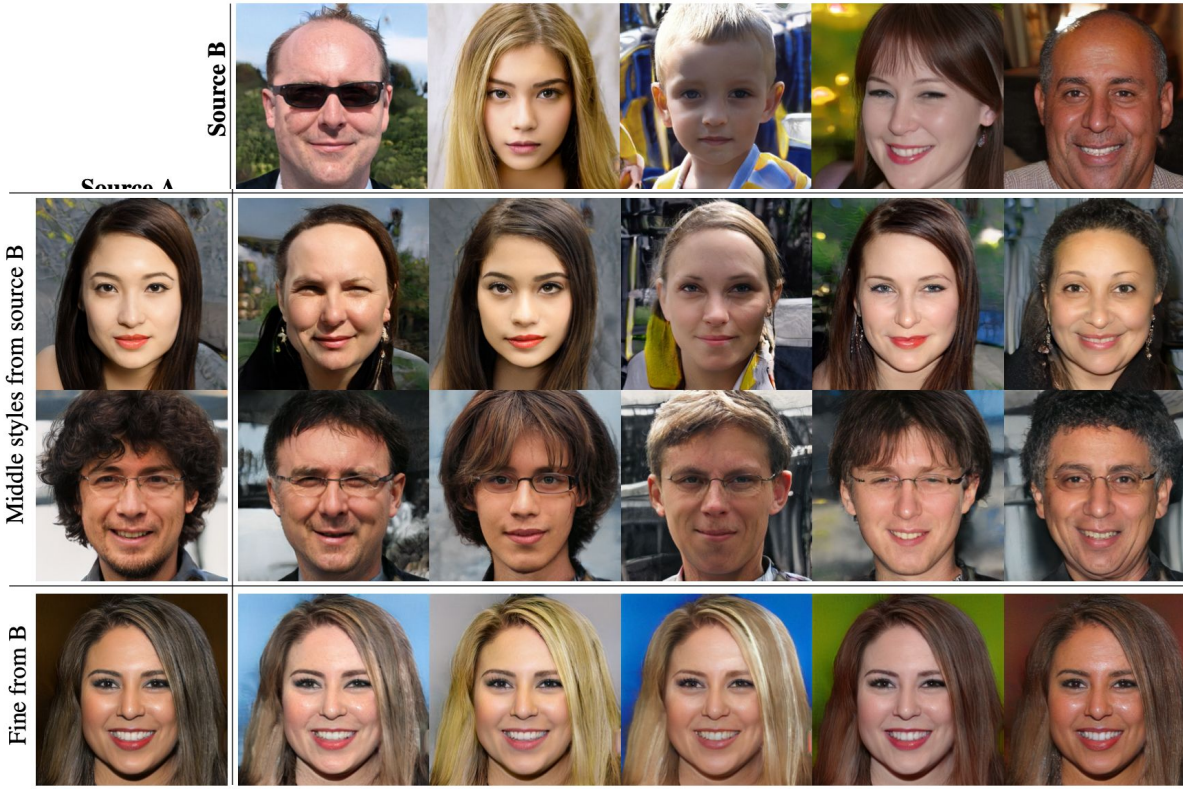
- Coarse vs medium vs fine styles
- Able to separate and remix them separately

[A Style-Based Generator Architecture for Generative Adversarial Networks](#) (2019)

StyleGAN

Considered a very specific notion of style…

- Coarse vs medium vs fine styles
- Able to separate and remix them separately



[A Style-Based Generator Architecture for Generative Adversarial Networks] (2019)

This Person Does Not Exist

Images made with StyleGAN (still v1?)

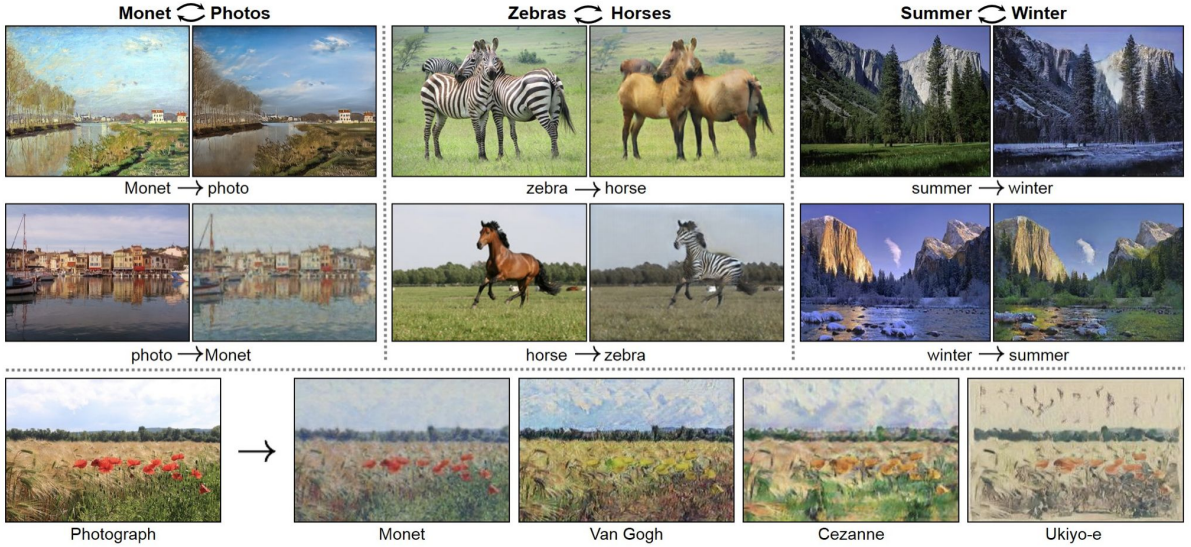
https://thispersondoesnotexist.com/

CycleGAN

Goal:

- Generate G: X → Y such that distribution of G(X) is indistinguishable from distribution of Y
  - But underconstrained.
  - What if input is ignored?
- Also generate F: Y → X similarly and train F(G(x))=x and G(F(y))=y

[Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks](#) (2017)



Challenge: very few style transfer pairs for training. Also very few Monet samples.

Inverting the Generator

Given the generator of a GAN and an image, can we find its latent?

- This is pretty straightforward with gradient descent.
  - More reliable than training inverse?
- Why?
  - Because the latents tend to be useful.
  - Similar latent ~ similar image ~ similar semantics.

[Inverting The Generator Of A Generative Adversarial Network](#) (2016)

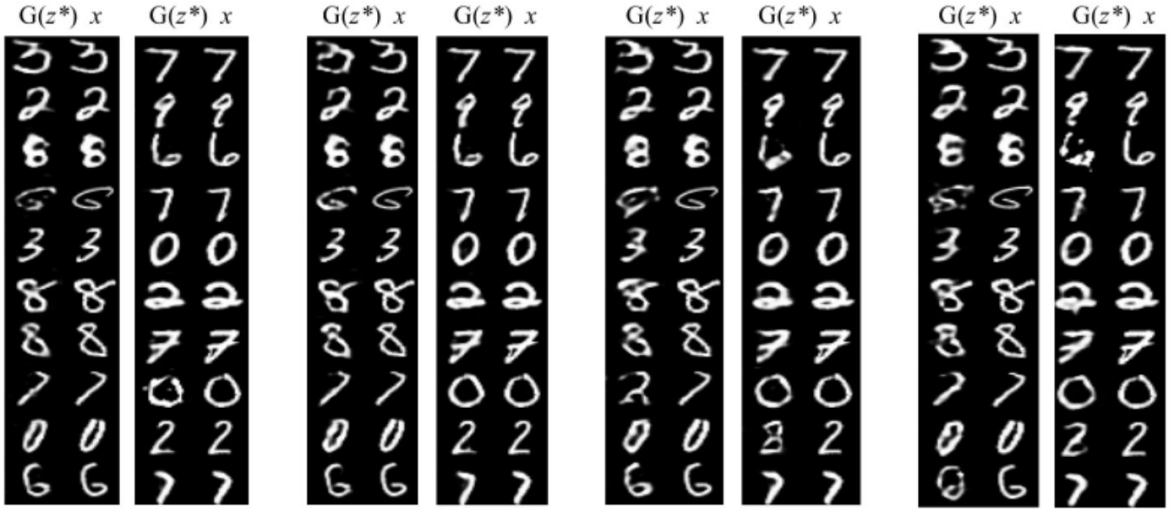


Figure 2: **Reconstructions for MNIST**: inverting a generator trained using a uniform prior (a-b) and a normal prior (c-d). The original image, x is on the right, while the inverted image is on the left $G(z^*)$.

Use this to check if an image came out of a particular generator?

Takeaways

- Generative adversarial networks are surprisingly effective at image generation.
  - If you can get them to converge.
  - If there is no mode collapse.
- This was one of the feet in the door to get generative image models working.
  - Identified many challenges for later kinds of models to avoid.
  - Still active research here.

Unsupervised Learning (preview)

What kind of models do we build when we don’t have explicit targets?

- Contrastive learning? (sometimes has labels but not explicit targets)
- Generative adversarial networks

Coming up:

- Normalizing flows and variational auto-encoders
- Diffusion models
- Neural fields

Feedback?