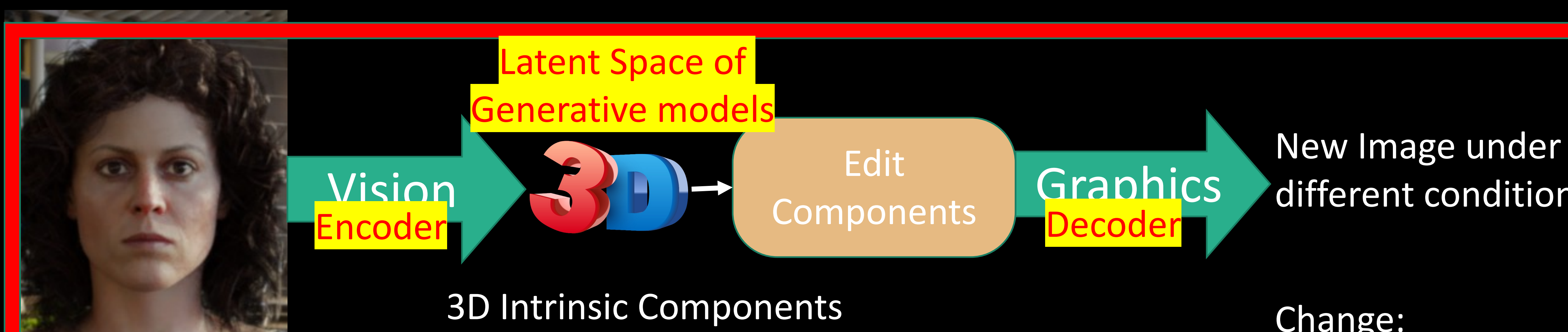
Lecture 5: Generative Adversarial Networks (GANs)

Respond at **PollEv.com/ronisen**

Text **RONISEN** to **22333** once to join, then text your message

Feel free to share your questions...

Next few lectures: Generative models for direct image based rendering.



Current Image

Implicit: Use a Neural Network (Conditional Generative networks)
Often, end-to-end.

Change:

- Viewpoint
- Lighting
- Reflectance
- Background
- Attributes
- Many others…

Taxonomy of Generative Models

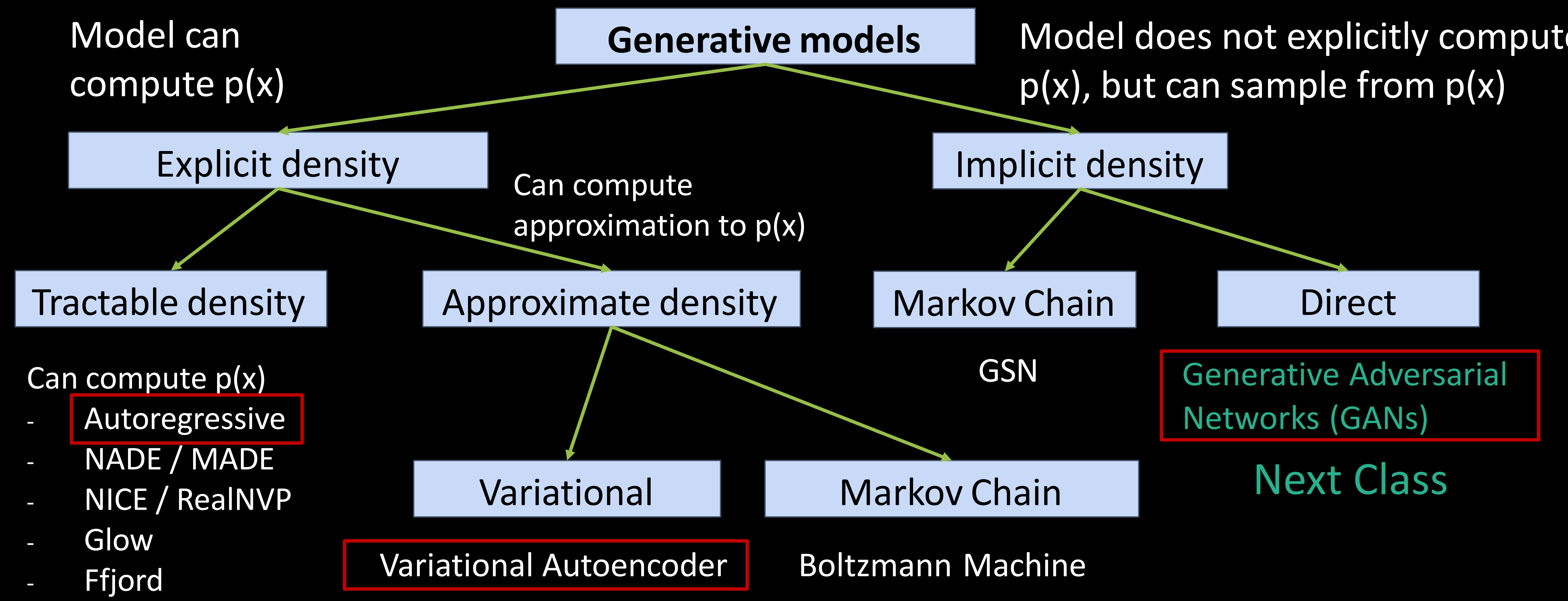
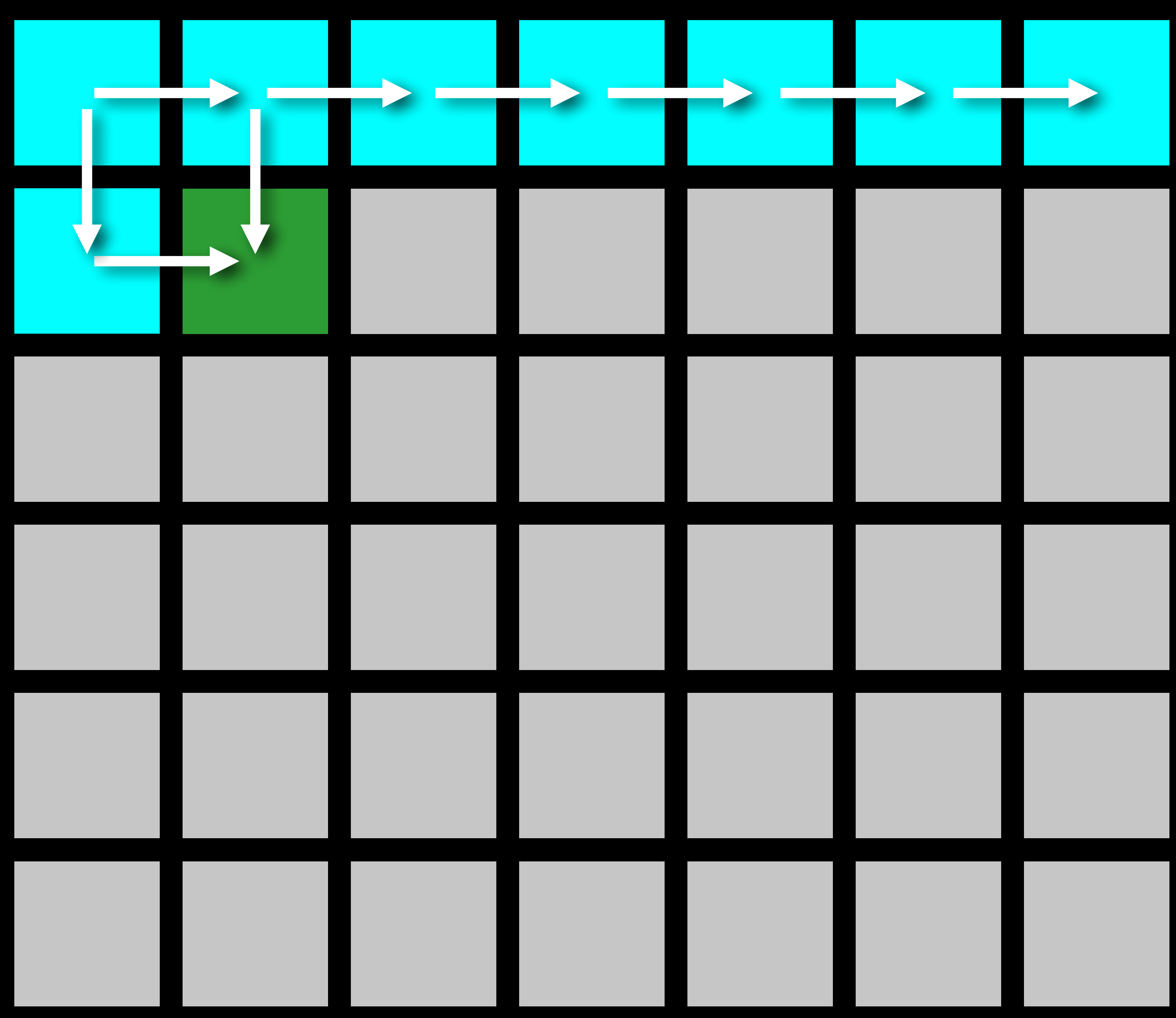


Figure adapted from Ian Goodfellow, Tutorial on Generative Adversarial Networks, 2017.

PixelRNN

Or

PixelCNN



Explicit Density: Autoregressive Models

Goal: Write down an explicit function for $p(x) = f(x, W)$

Assume x consists of multiple subparts:

$$x = (x_1, x_2, x_3, \ldots, x_T)$$

Break down probability using the chain rule:

$$\begin{aligned} p(x) &= p(x_1, x_2, x_3, \ldots, x_T) \\ &= p(x_1)p(x_2 \mid x_1)p(x_3 \mid x_1, x_2) \ldots \\ &= \prod_{t=1}^{T} p(x_t \mid x_1, \ldots, x_{t-1}) \end{aligned}$$

Probability of the next subpart given all the previous subparts

$p(x_1)$ $p(x_2)$ $p(x_3)$ $p(x_4)$

$h_1 \rightarrow h_2 \rightarrow h_3 \rightarrow h_4$

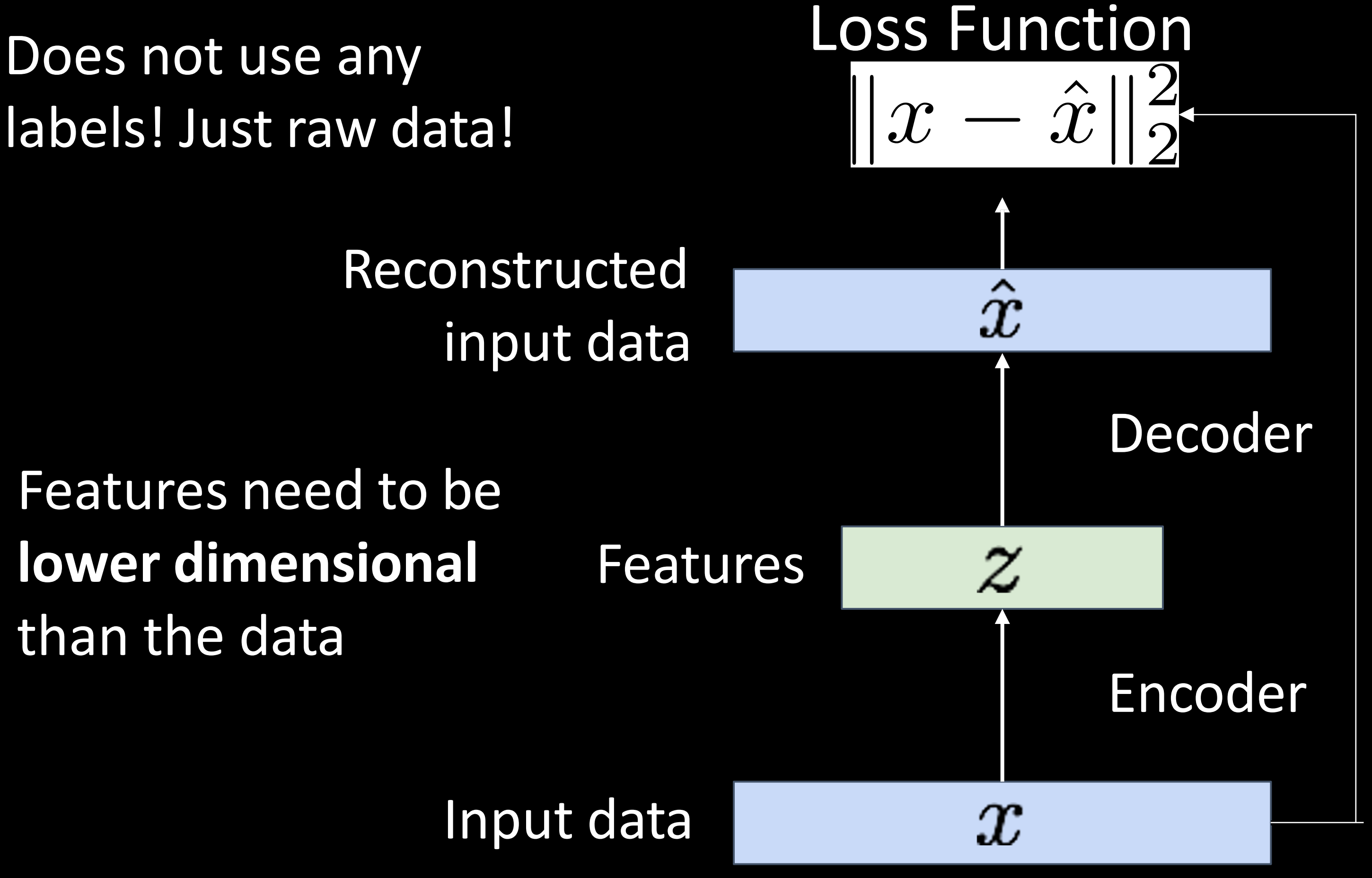
x_0 x_1 x_2 x_3

We've already seen this! Language modeling with an RNN!

(Regular, non-variational) Autoencoders

Loss: L2 distance between input and reconstructed data.

Does not use any labels! Just raw data!



Features need to be **lower dimensional** than the data

Reconstructed data

Decoder:
4 tconv layers

Encoder:
4 conv layers

Input Data

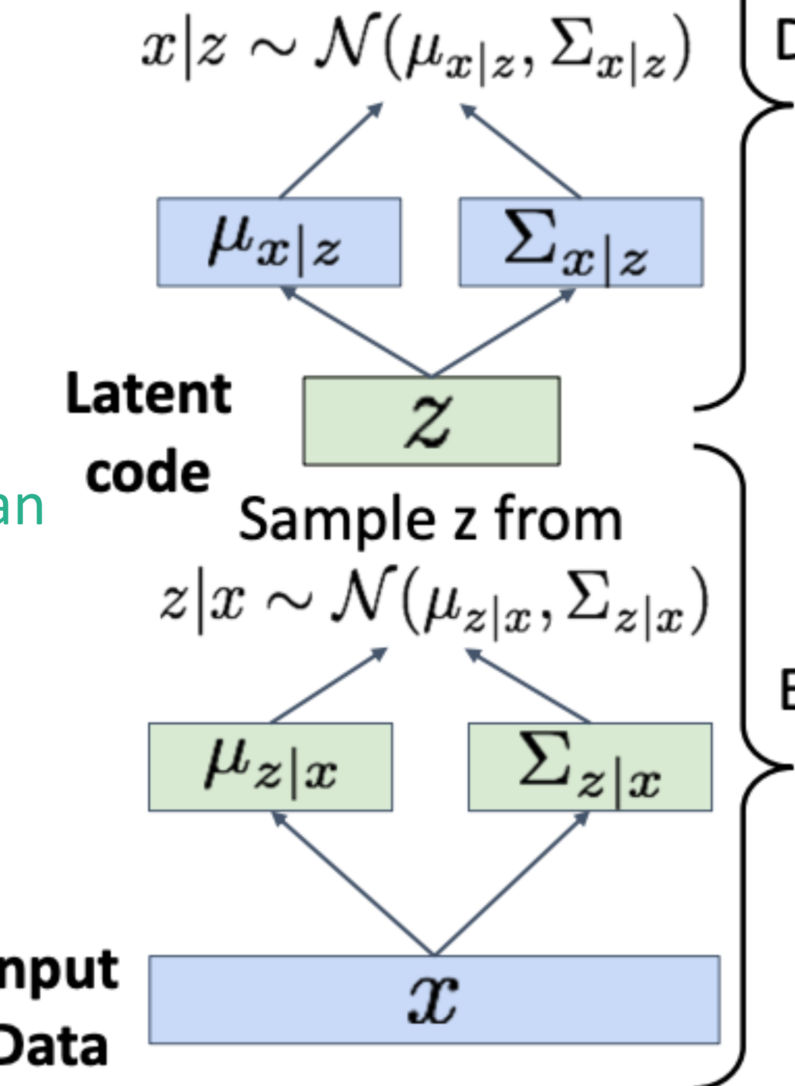
Not probabilistic: No way to sample new data from learned model

Variational Autoencoders

Train by maximizing the
variational lower bound

$$E_{z \sim q_\phi(z|x)}[\log p_\theta(x|z)] - D_{KL}\left(q_\phi(z|x), p(z)\right)$$

1. Run input data through **encoder** to get a distribution over latent codes Try to make z gaussian
2. **Encoder output should match the prior p(z)!**
3. Sample code z from encoder output
4. Run sampled code through **decoder** to get a distribution over data samples
5. **Original input data should be likely under the distribution output from (4)!**



Few Math recap: What is Expectation?

Definition:

$$\mathrm{E}[X] = \sum_x x p_X(x)$$

$$E_{z \sim q_\phi(z|x)}[\log p_\theta(x|z)]$$

- Sample z_i from the latent space (gaussian).
- Pass z_i through decoder to reconstruct image x'_i
- Loss can be computed as binary cross-entropy loss between the real images and the generated images = sum{x'_i * log (x_i)}

(Some implementation also use regular MSE loss).

GAN

Generative Adversarial Network

“The most interesting idea in the
last ten years in machine learning.”
— Yann LeCun (Facebook AI Research)

Generative Adversarial Nets

Ian J. Goodfellow, Jean Pouget-Abadie[*], Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair[†], Aaron Courville, Yoshua Bengio[‡]
Département d'informatique et de recherche opérationnelle
Université de Montréal
Montréal, QC H3C 3J7

Abstract

We propose a new framework for estimating generative models via an adversarial process, in which we simultaneously train two models: a generative model G that captures the data distribution, and a discriminative model D that estimates

Generative Adversarial Networks

Setup: Assume we have data x_i drawn from distribution $p_{data}(x)$. Want to sample from p_{data}.

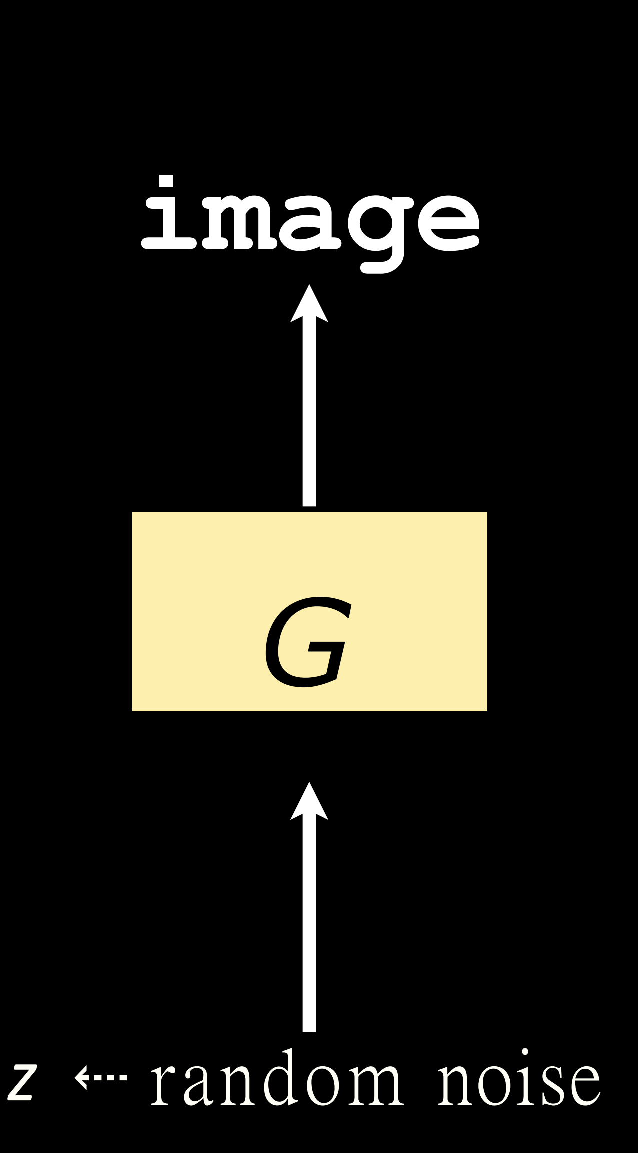
Idea: Introduce a latent variable z with simple prior p(z).

Sample $z \sim p(z)$ and pass to a **Generator Network** x = G(z)

Then x is a sample from the **Generator distribution** p_G. Want $p_G = p_{data}$!

Goodfellow et al, "Generative Adversarial Nets", NeurIPS 2014

Network takes a random input and produces a sample from the data distribution as output



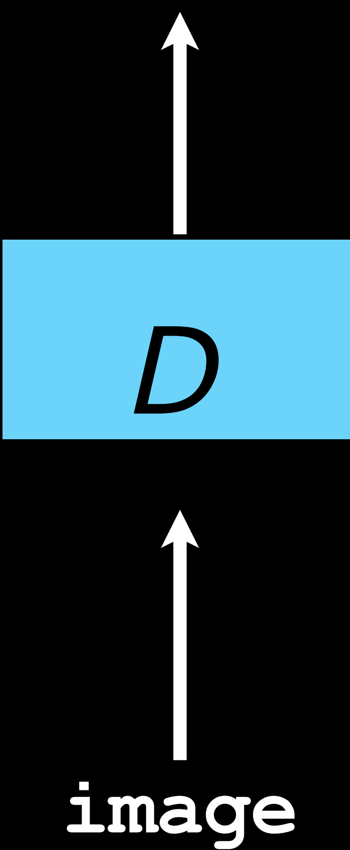
Similar as decoder for VAE.

Network classifies input as “real” or “fake”

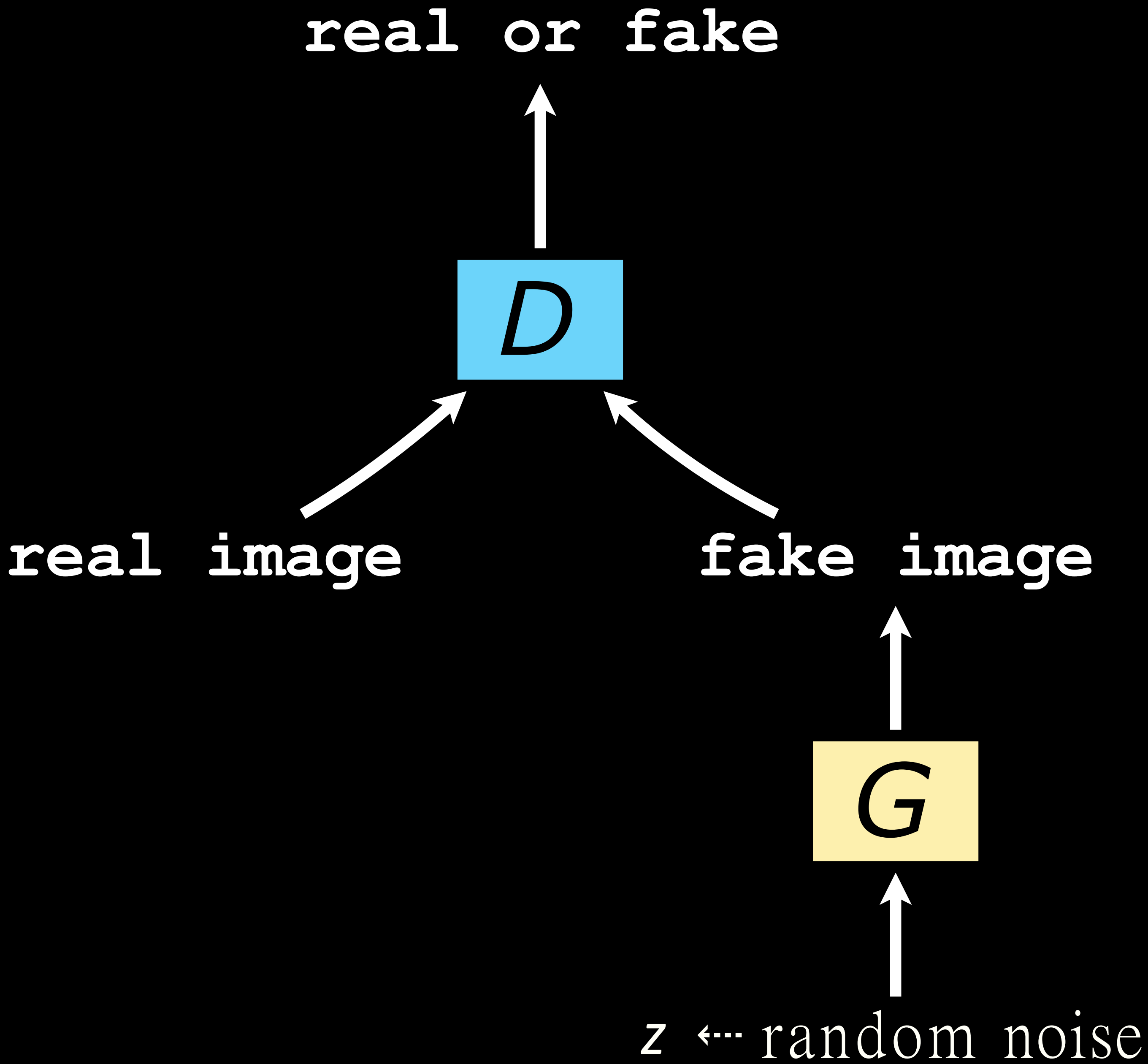
Network classifies input as “real” or “fake”

“fake” inputs come from the generator

real or fake



Discriminator

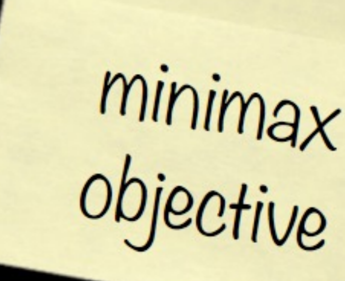

real or fake
D
real image
fake image
G
z ← random noise

minimax objective

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

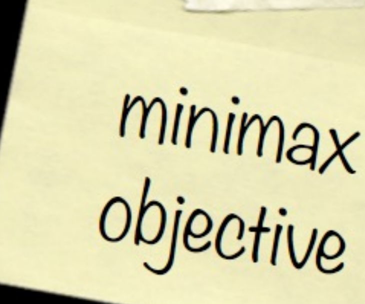
minimax objective

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



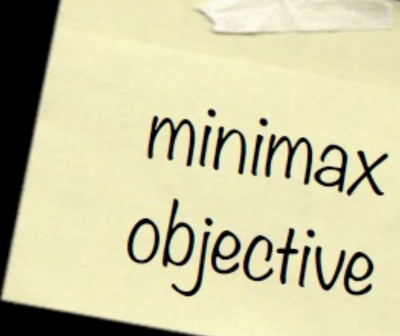
$$\mathbb{E}_{x \sim p_{\text{data}}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

cross entropy loss between real and fake images



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

loss for real images

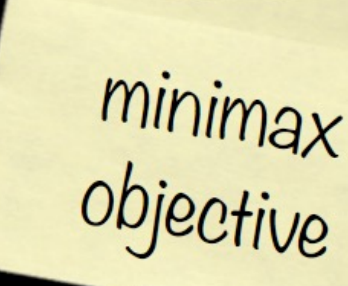


$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{\text{data}}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

loss for fake images

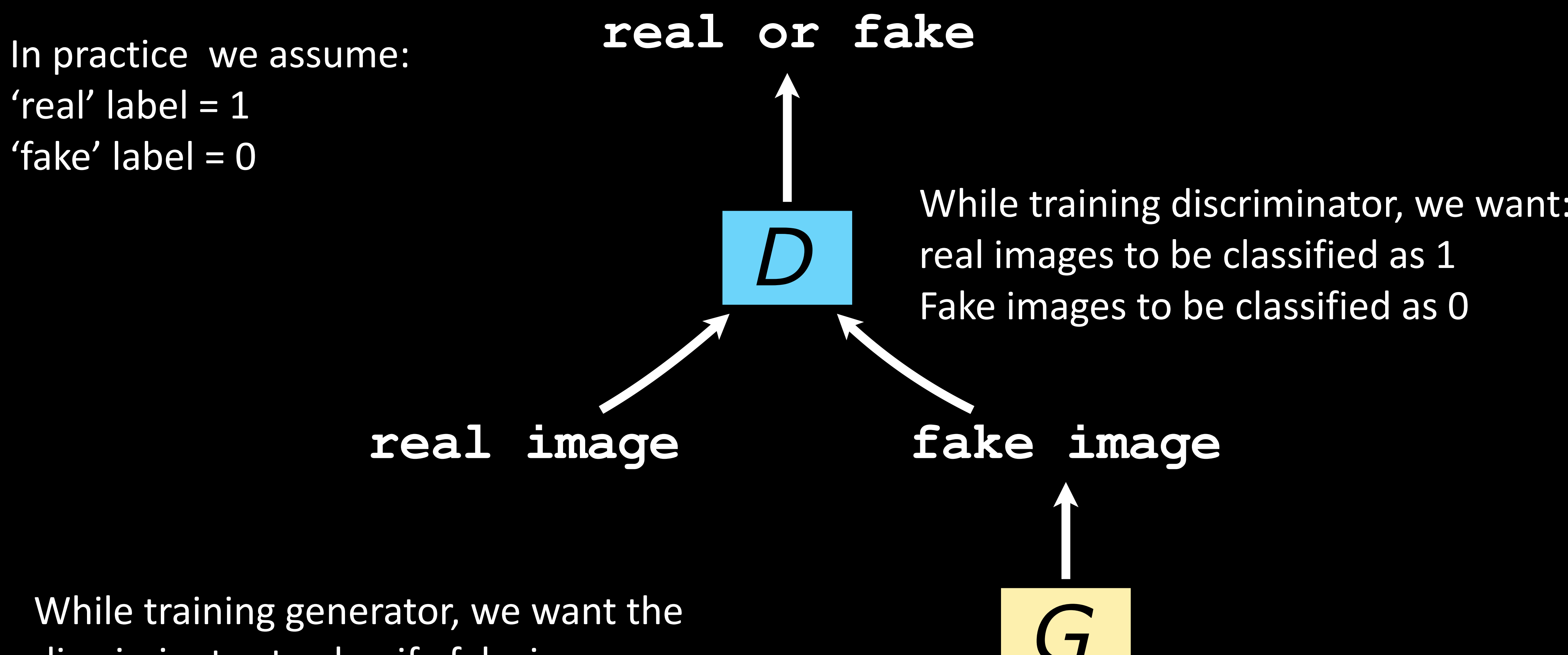
$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{\text{data}}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

discriminator wants to maximize objective

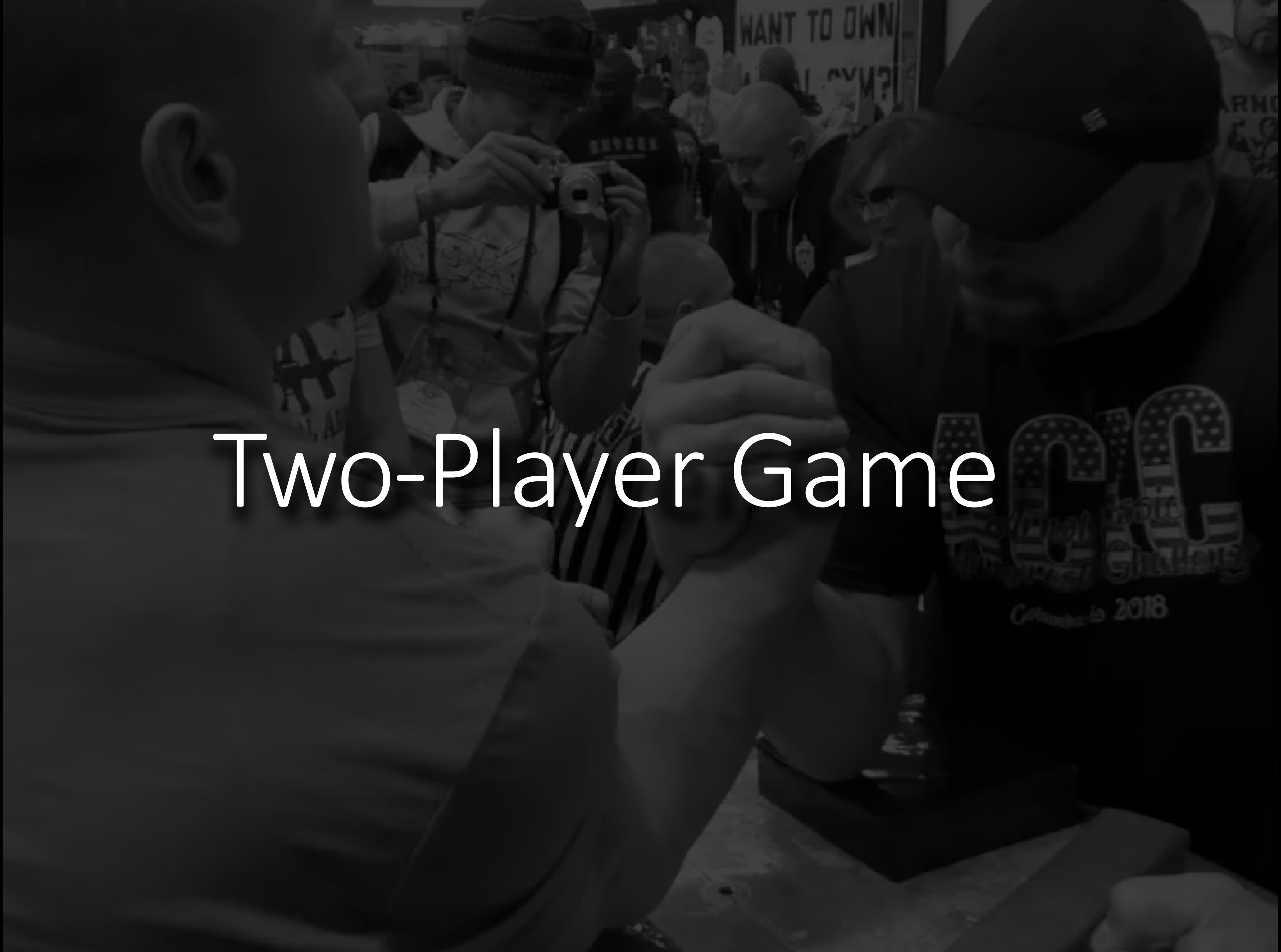


$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x\sim p_{\text{data}}} \log D_{\theta_d}(x) + \mathbb{E}_{z\sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))\right]$$

generator wants to minimize objective



While training generator, we want the discriminator to classify fake images as real (label=1).

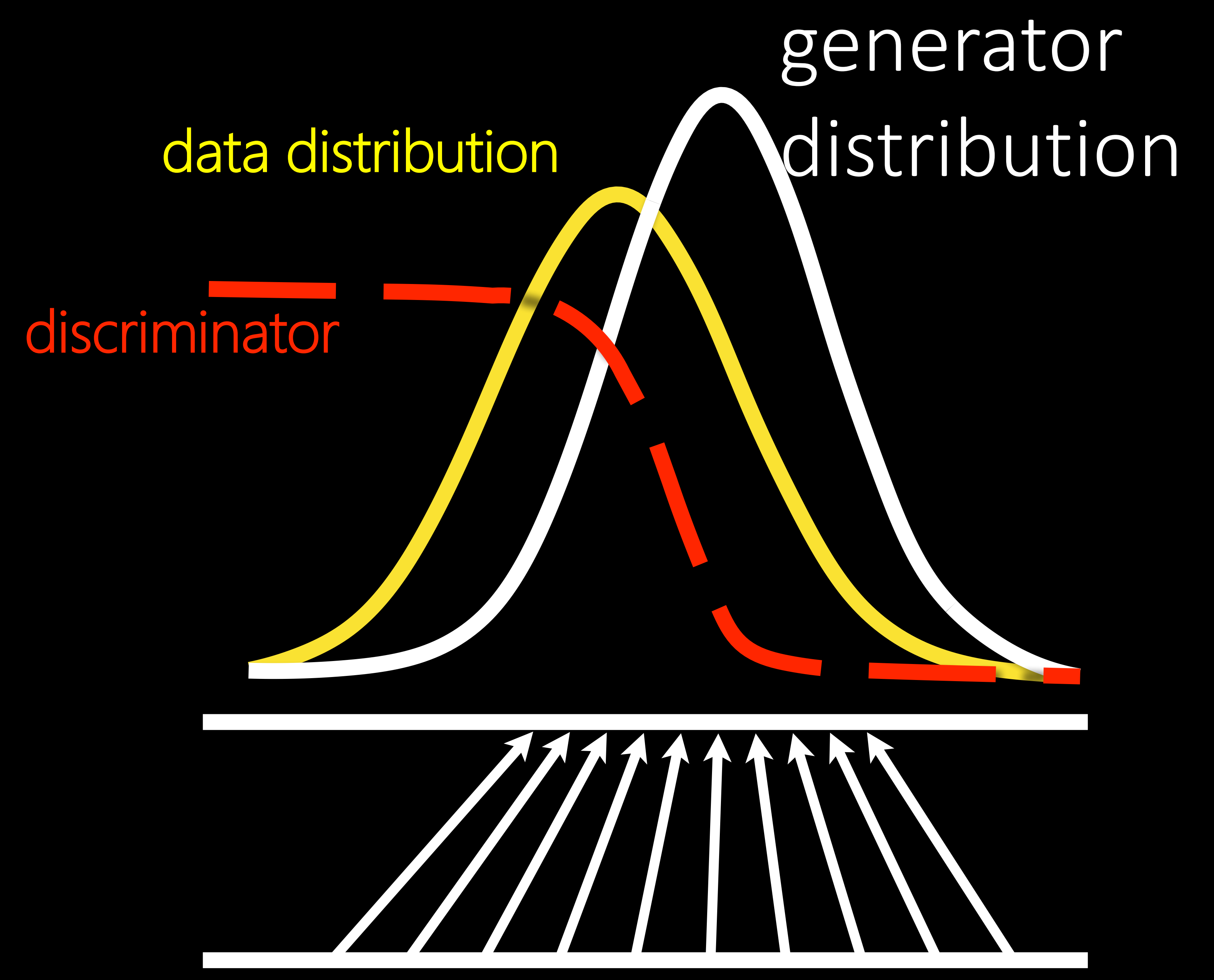


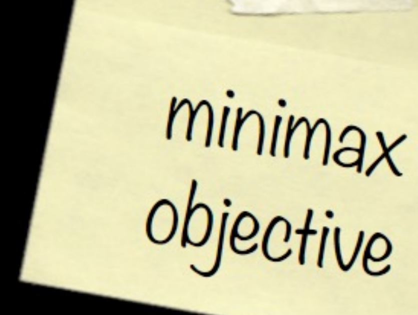
Two-Player Game

WANT TO OWN
ACIC
Columbus 2018

WANT TO OWN
Generator

WANT TO OWN
Discriminator
Generator


generator
distribution
data distribution
discriminator



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

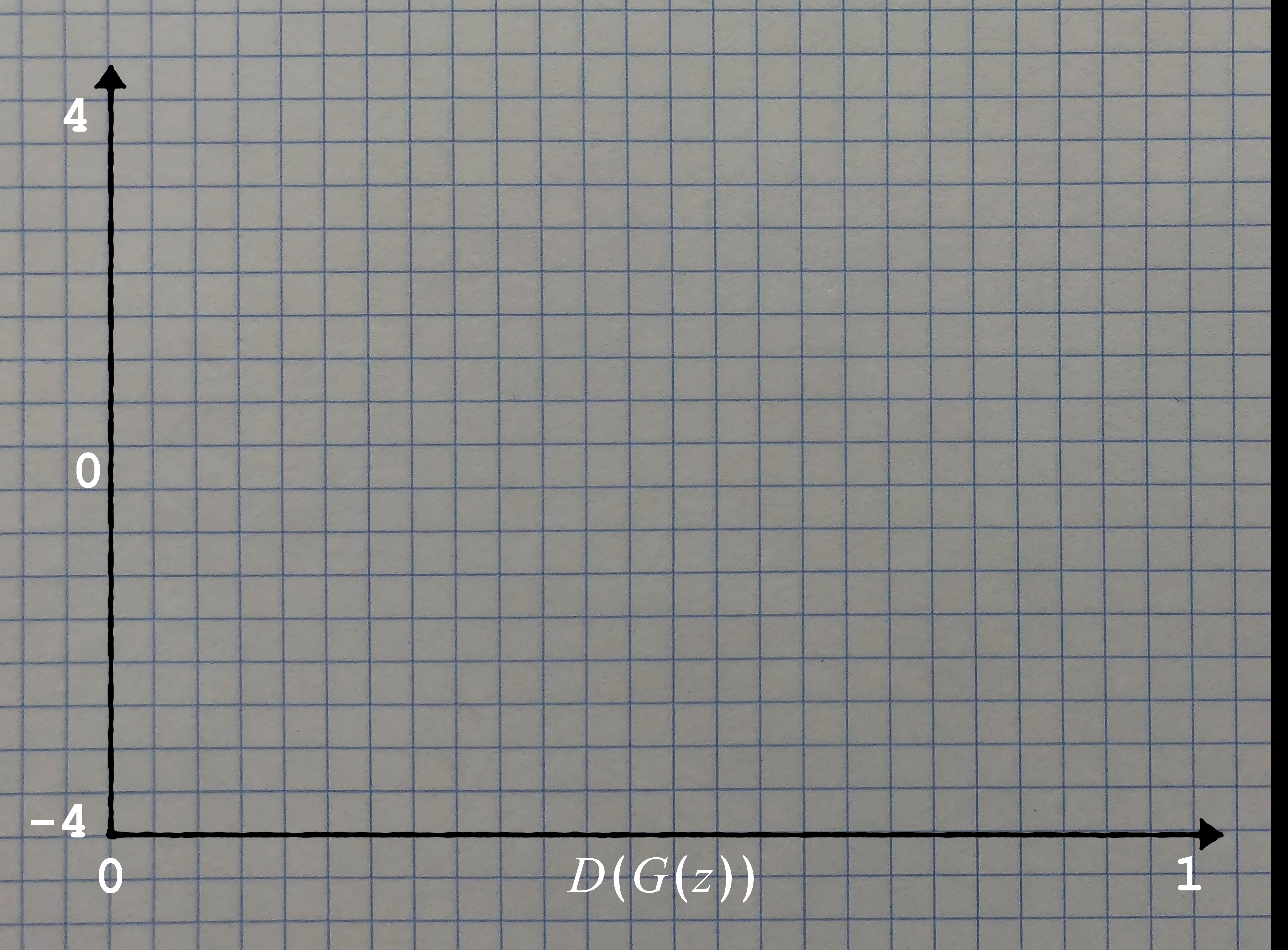
optimize by alternating between minimizing and maximizing respective sub-objectives.

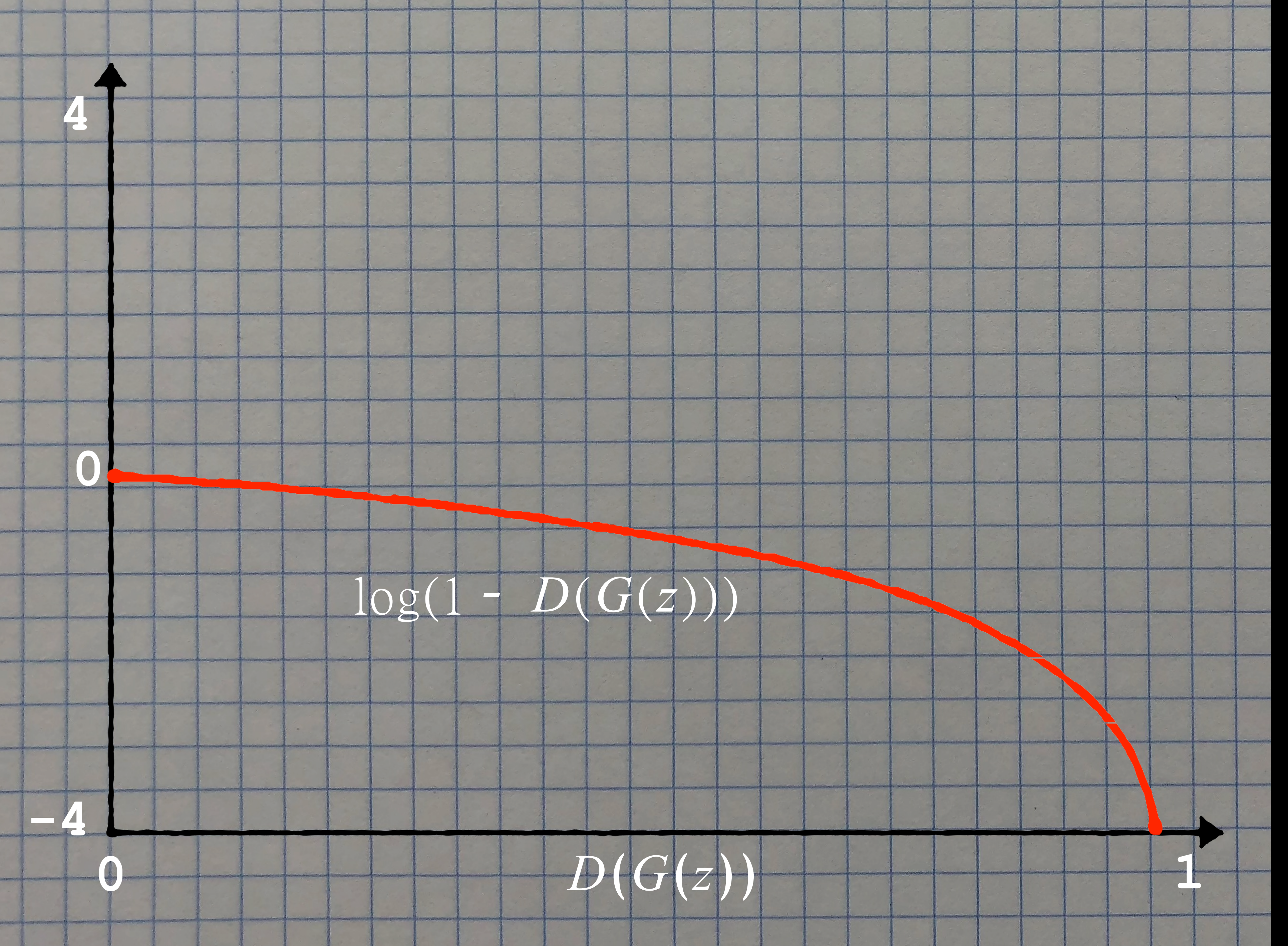
$$\max_{\theta_d} \left[\mathbb{E}_{x\sim p_{\text{data}}} \log D_{\theta_d}(x) + \mathbb{E}_{z\sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

gradient ascent on discriminator

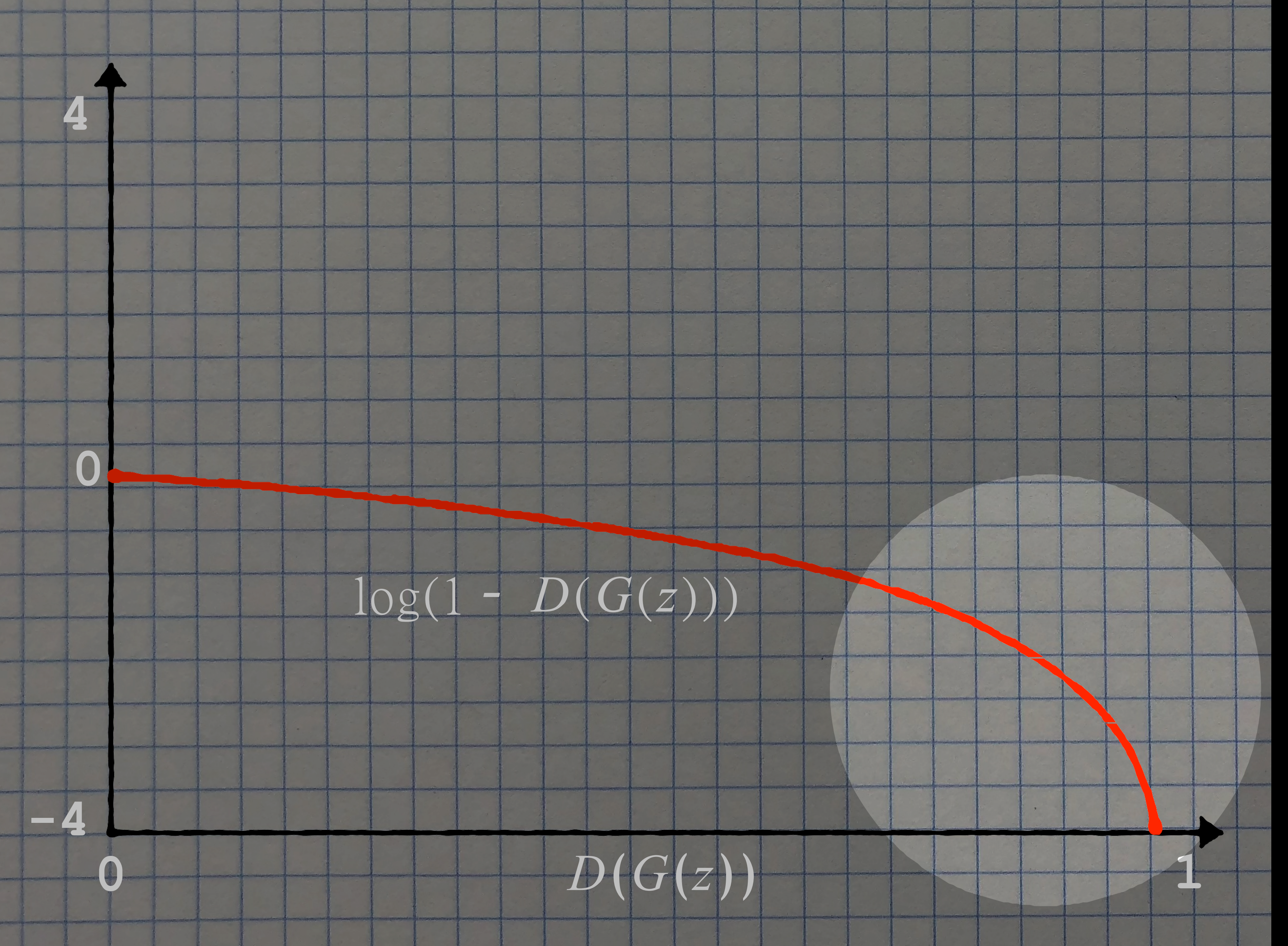
$$\min_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

gradient descent on generator

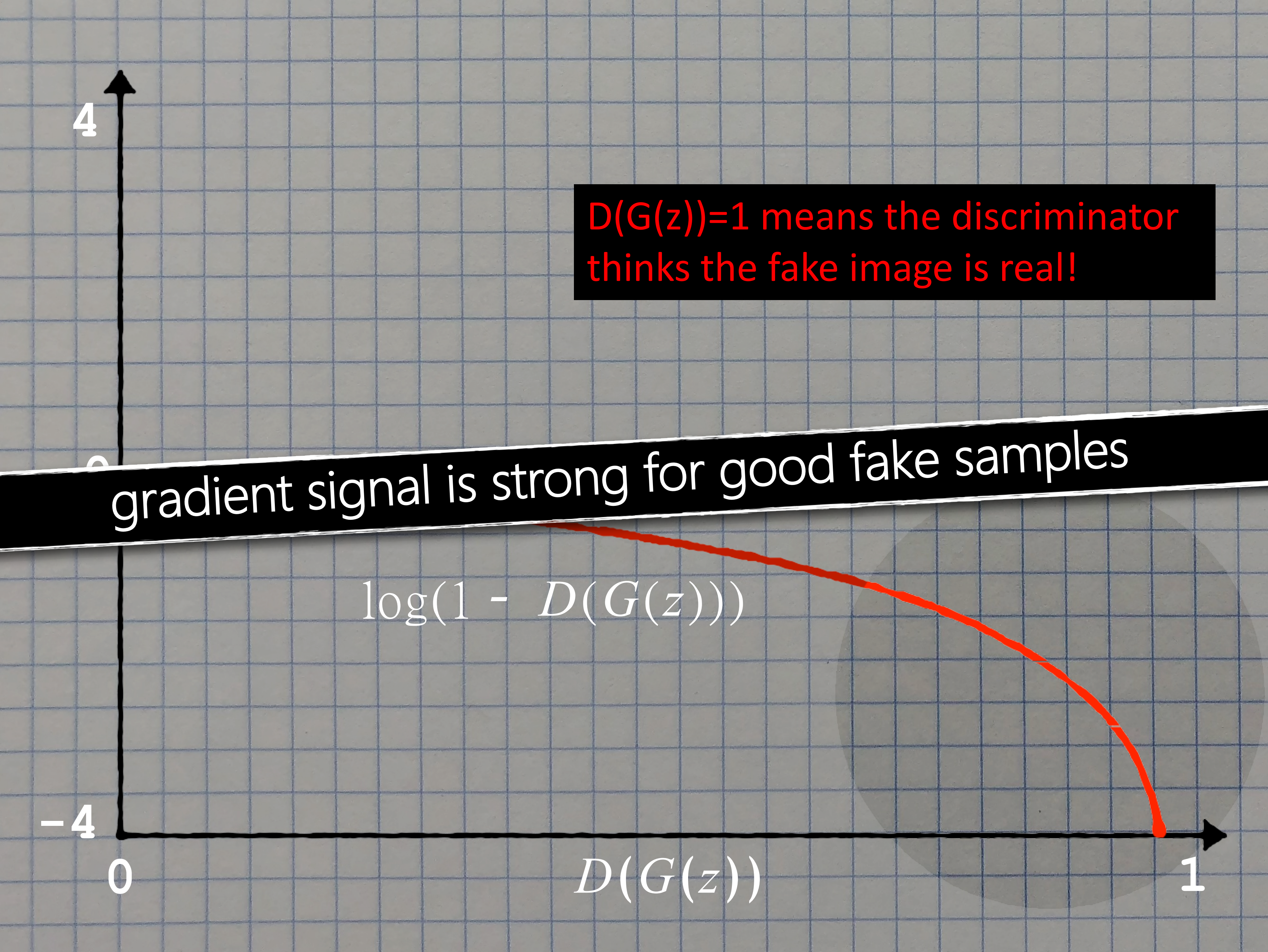

4
0
-4
0
$D(G(z))$
1

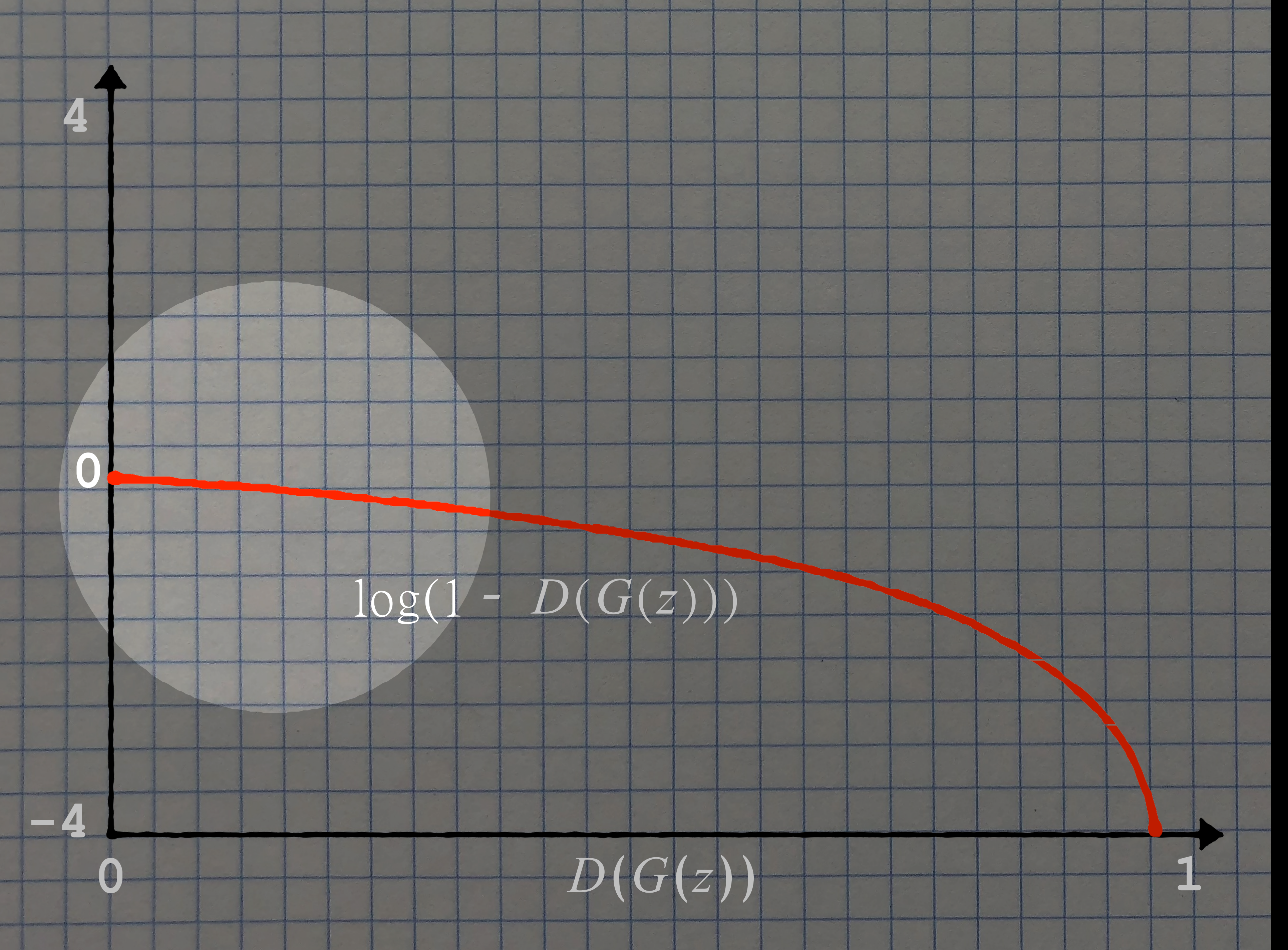


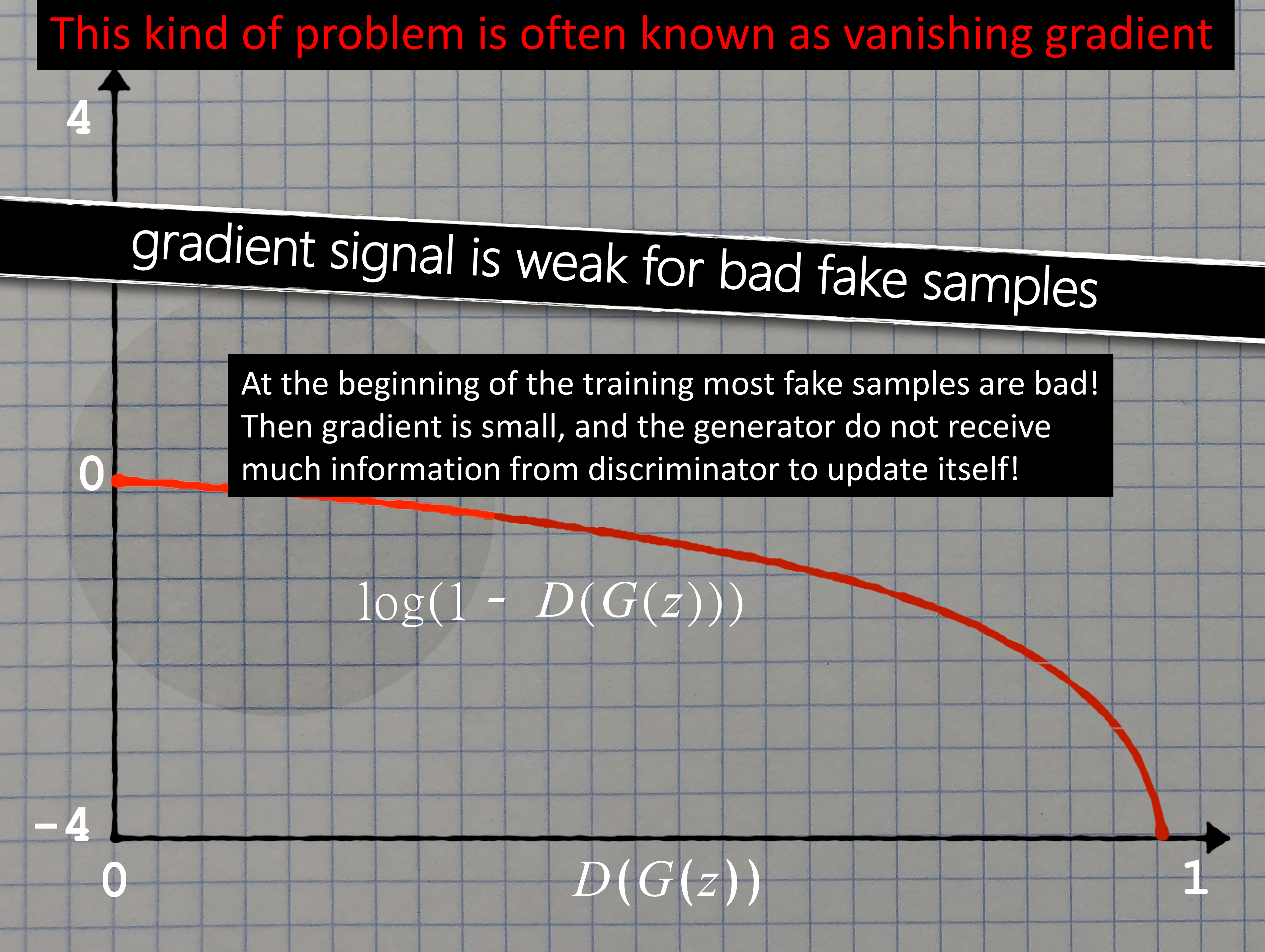
4
0
-4
0
1
log(1 - D(G(z)))
D(G(z))



4
0
-4
0
1
log(1 - D(G(z)))
D(G(z))


4
D(G(z))=1 means the discriminator thinks the fake image is real!
gradient signal is strong for good fake samples
log(1 - D(G(z)))
-4
0
D(G(z))
1

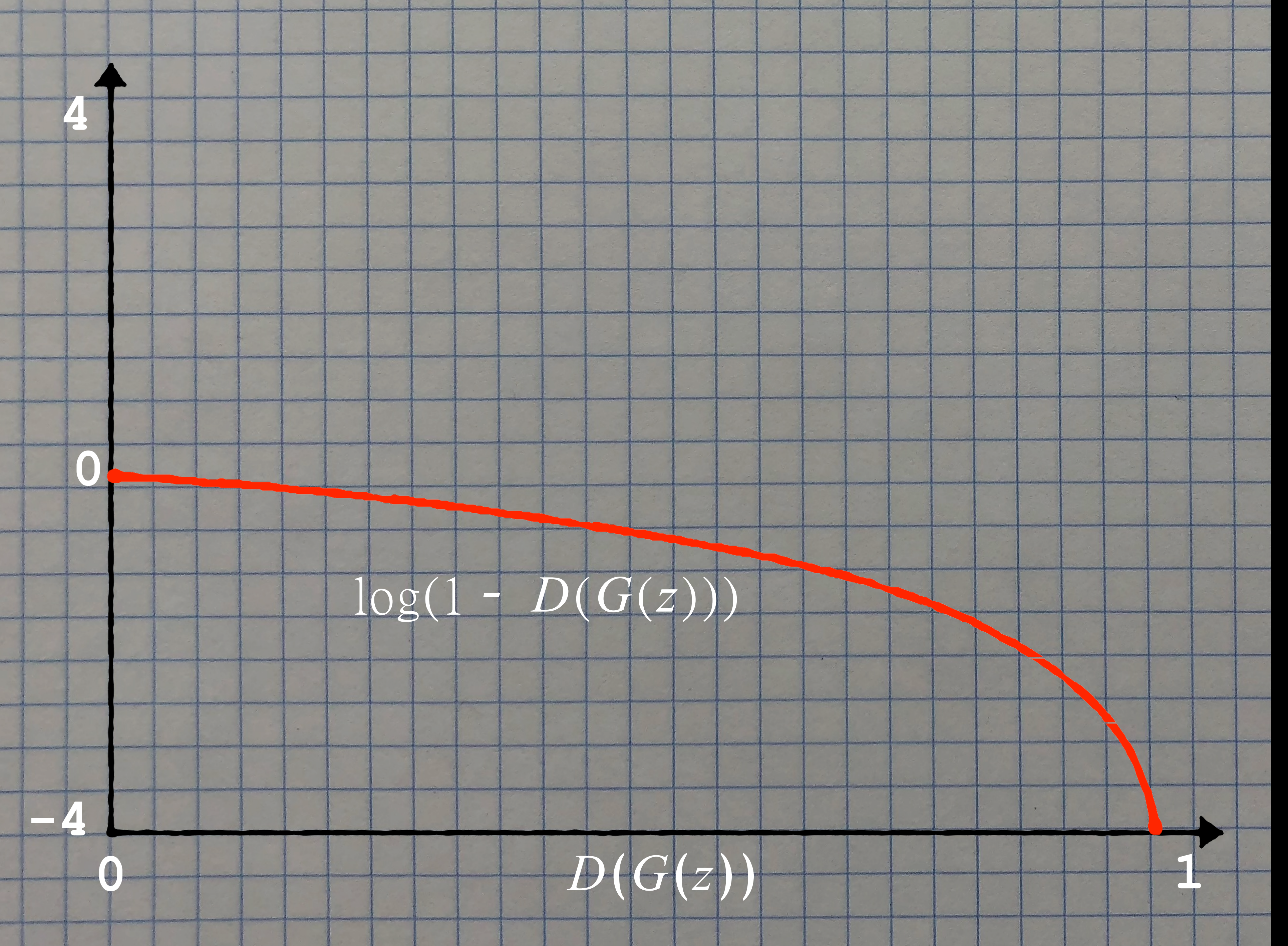

4
0
-4
log(1 - D(G(z)))
0
D(G(z))
1


This kind of problem is often known as vanishing gradient
4
gradient signal is weak for bad fake samples
At the beginning of the training most fake samples are bad!
Then gradient is small, and the generator do not receive
much information from discriminator to update itself!
0
log(1 - D(G(z)))
-4
0
D(G(z))
1

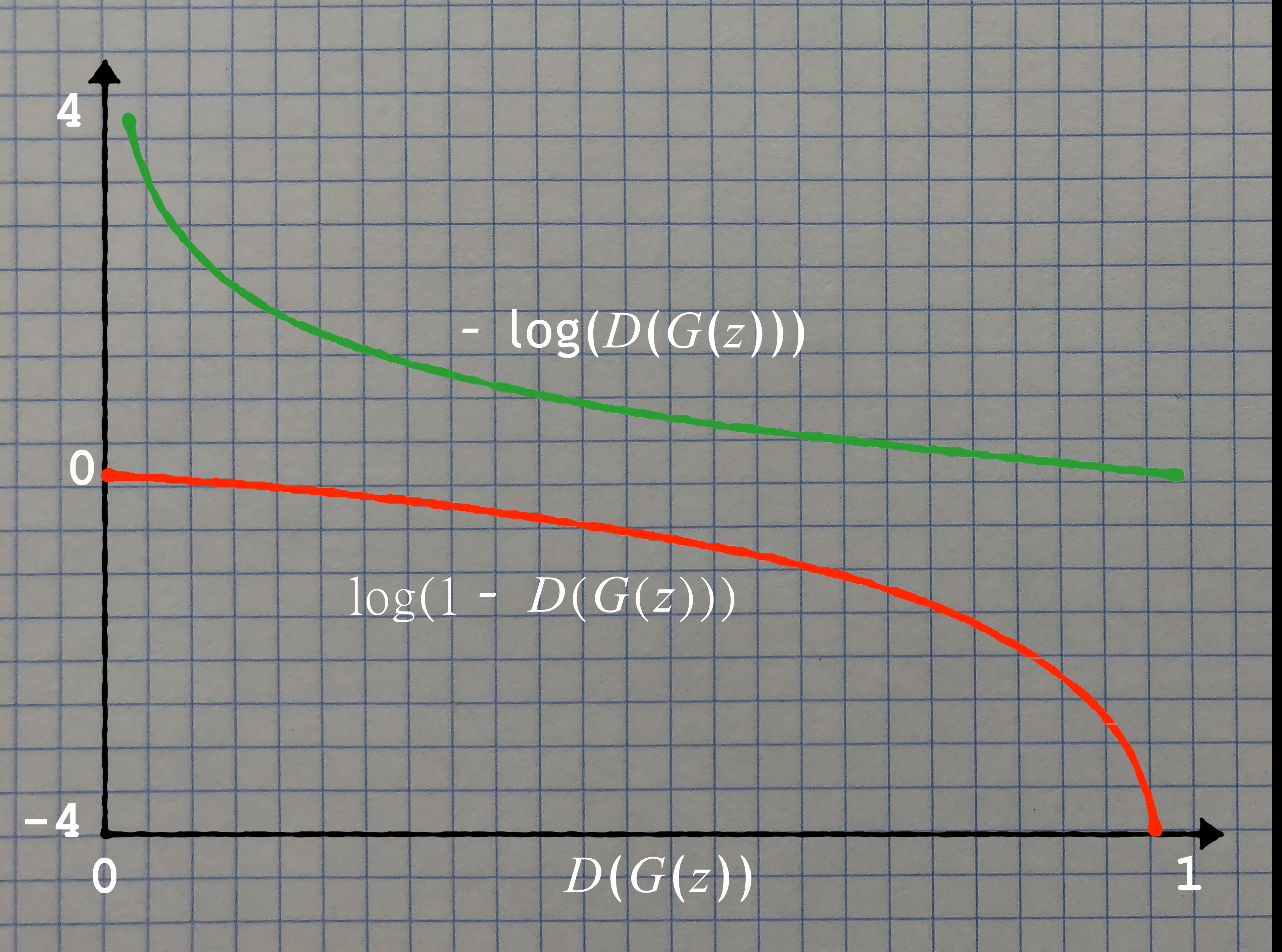
$$\min_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

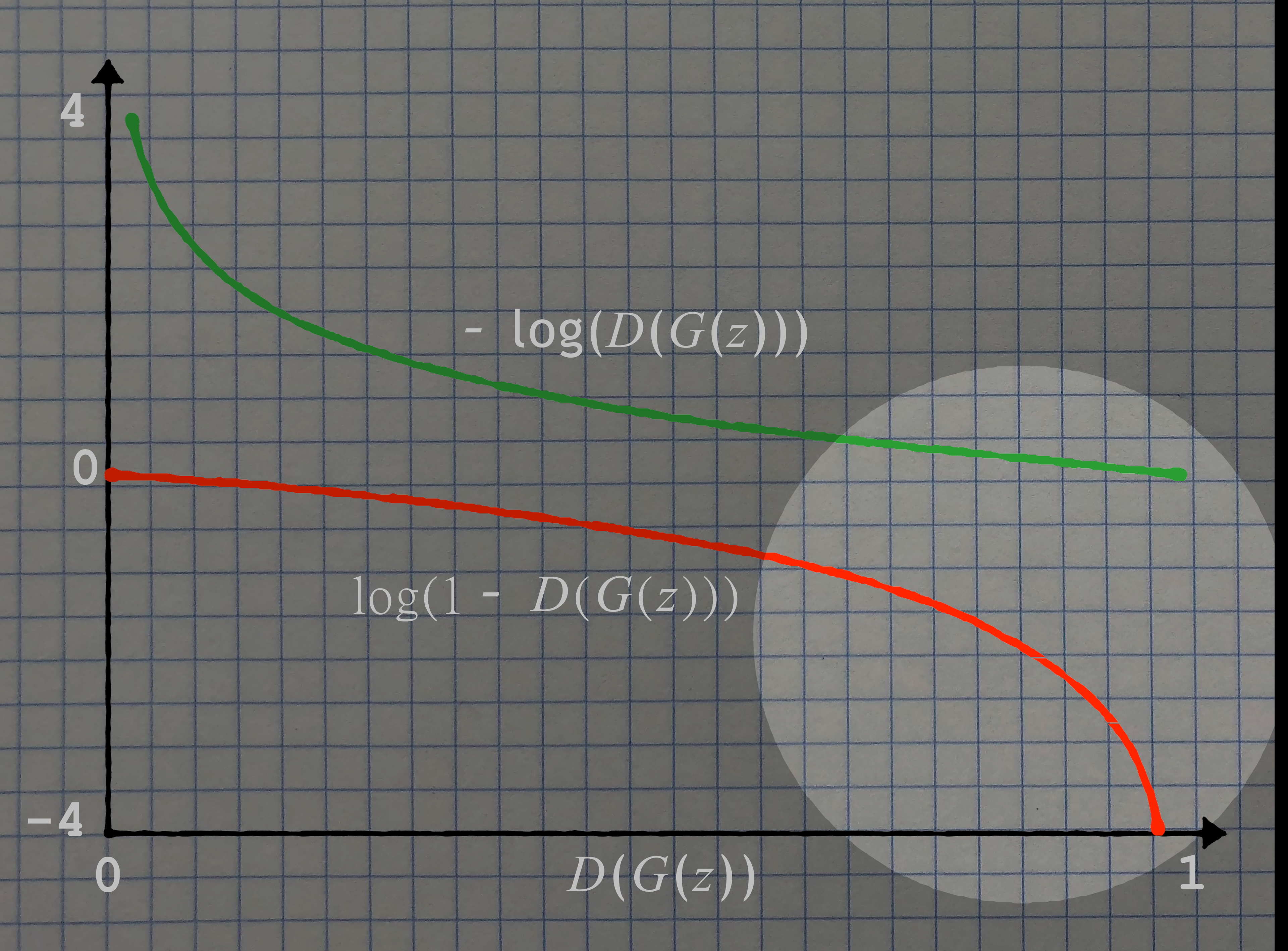
replace with

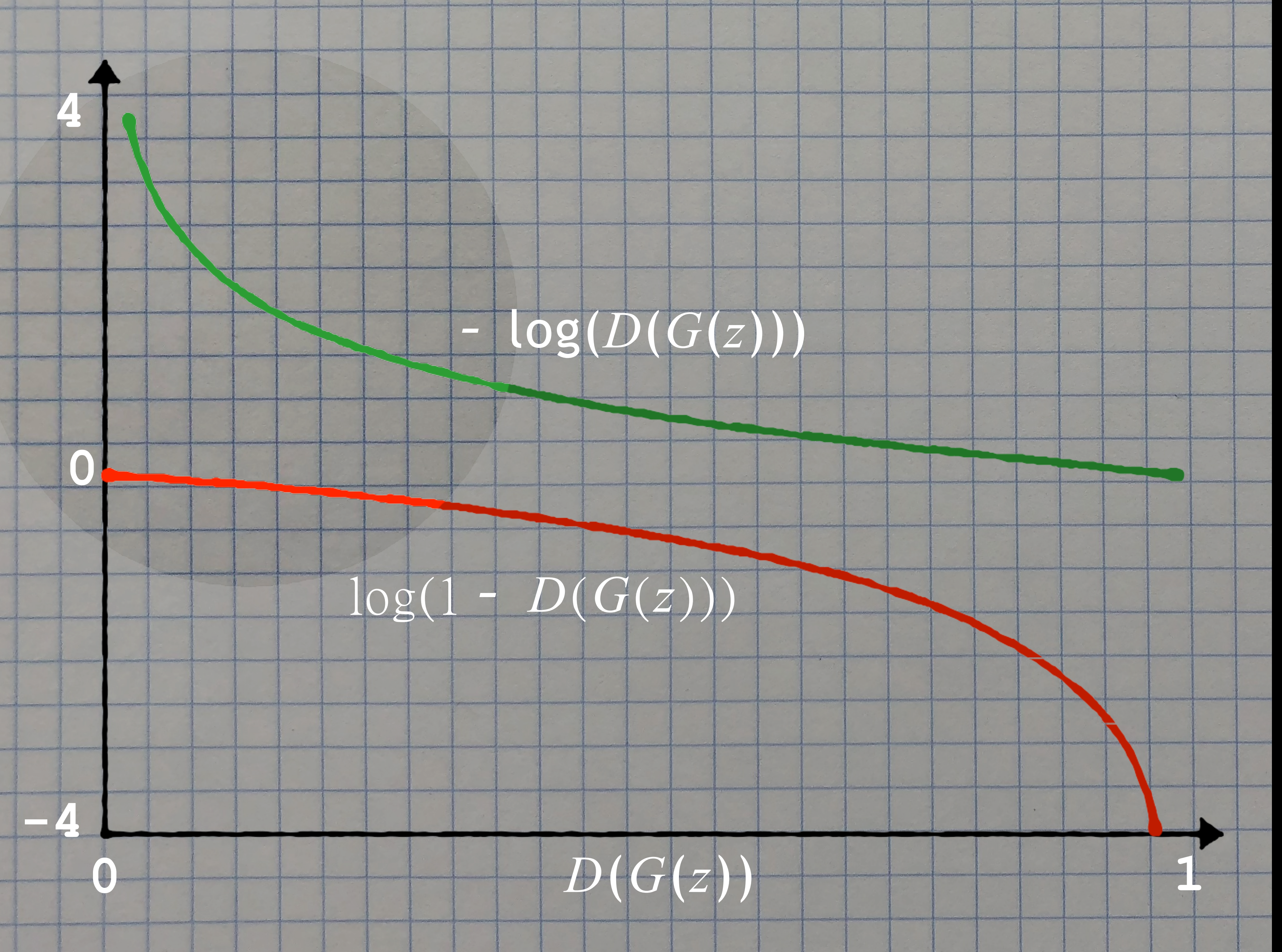
$$\max_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(D_{\theta_d}(G_{\theta_g}(z)))$$

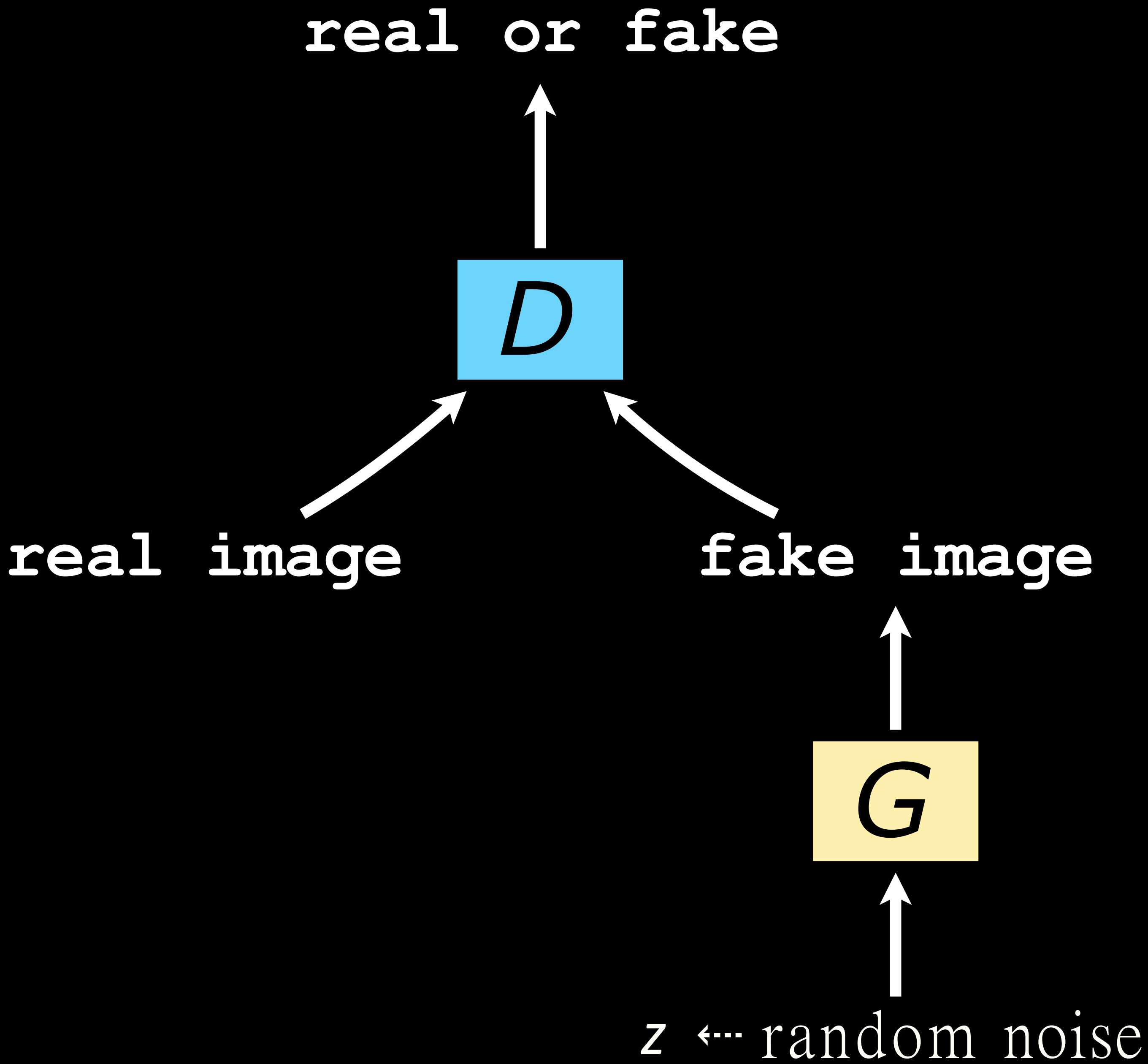


4
0
-4
0
1
log(1 - D(G(z)))
D(G(z))


4
0
-4
0
1
- log(D(G(z)))
log(1 - D(G(z)))
D(G(z))


4
0
-4
0
1
- log(D(G(z)))
log(1 - D(G(z)))
D(G(z))


4
0
-4
0
1
- log(D(G(z)))
log(1 - D(G(z)))
D(G(z))


real or fake
D
real image
fake image
G
z ⇠ random noise

GAN training

```
for number of training iterations do

   for k steps do
      sample m noise samples from noise prior
      sample m real examples from dataset
      update the discriminator by gradient ascent
   end for

   sample m noise samples from noise prior
   update generator by stochastic gradient ascent

end for
```

```
for number of training iterations do

   for k steps do
      sample m noise samples from noise prior
      sample m real examples from dataset
      update the discriminator by gradient ascent
   end for

   sample m noise samples from noise prior
   update generator by stochastic gradient ascent

end for
```

```
for number of training iterations do

    for k steps do
        sample m noise samples from noise prior
        sample m real examples from dataset
        update the discriminator by gradient ascent
    end for

    sample m noise samples from noise prior
    update generator by stochastic gradient ascent

end for
```

```
for number of training iterations do
   for k steps do
      sample m noise samples from noise prior
      sample m real examples from dataset
      update the discriminator by gradient ascent
   end for
   update generator by stochastic gra...
end for
```

update discriminator

```
for number of training iterations do

   for k steps do
      sample m noise samples from noise prior
      sample m real examples from dataset
      update the discriminator by gradient ascent
   end for

   sample m noise samples from noise prior
   update generator by stochastic gradient ascent

end for
```

```
for number of training iterations do

   for k steps do
      sample m noise samples from noise prior
      sample m real examples from dataset
      update the discriminator by gradient ascent
   end for

   sample m noise samples from noise prior
   update generator by stochastic gradient ascent

end for
```

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

```
for k steps do
    sample m noise samples from noise prior
```

```
    update the discriminator by gradient ascent
```

```
for number of training iterations do

   for k steps do
      sample m noise samples from noise prior
      sample m real examples from dataset
      update the discriminator by gradient ascent
   end for

   sample m noise samples from noise prior
   update generator by stochastic gradient ascent

end for
```

```
for number of training iterations do

   for k steps do
      sample m noise samples from noise prior
      sample m real examples from dataset
      update the discriminator by gradient ascent
   end for

   sample m noise samples from noise prior
   update generator by stochastic gradient ascent
```

update generator using modified objective

```
for number of training iterations do

    for k steps do
        sample m noise samples from noise prior
        sample m real examples from dataset
        update the discriminator by gradient ascent
    end for

    sample m noise samples from noise prior
    update generator by stochastic gradient ascent

end for
```

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log(D_{\theta_d}(G_{\theta_g}(z^{(i)})))$$

GAN training

```
for number of training iterations do

   for k steps do
      sample m noise samples from noise prior
      sample m real examples from dataset
      update the discriminator by gradient ascent
   end for

   sample m noise samples from noise prior
   update generator by stochastic gradient ascent

end for
```

Generative Adversarial Networks

PYTORCH

GAN training

```
z_dim, data_dim = 8, 2
hidden_dim = 100
batch_size = 10
lr = 1e-3

G_model = nn.Sequential(nn.Linear(z_dim, hidden_dim),
                        nn.ReLU(),
                        nn.Linear(hidden_dim, data_dim))

D_model = nn.Sequential(nn.Linear(data_dim, hidden_dim),
                        nn.ReLU(),
                        nn.Linear(hidden_dim, 1),
                        nn.Sigmoid())

G_optimizer =  optim.Adam(G_model.parameters(),
                         lr = lr)
D_optimizer =  optim.Adam(D_model.parameters(),
                         lr = lr)
```

```
z_dim, data_dim = 8, 2
hidden_dim = 100
batch_size = 10
lr = 1e-3

G_model = nn.Sequential(nn.Linear(z_dim, hidden_dim),
                        nn.ReLU(),
                        nn.Linear(hidden_dim, data_dim))

D_model = nn.Sequential(nn.Linear(data_dim, hidden_dim),
                        nn.ReLU(),
                        nn.Linear(hidden_dim, 1),
                        nn.Sigmoid())

G_optimizer =  optim.Adam(G_model.parameters(),
                          lr = lr)
D_optimizer =  optim.Adam(D_model.parameters(),
                          lr = lr)
```

```
z_dim, data_dim = 8, 2
hidden_dim = 100
batch_size = 10
lr = 1e-3

G_model = nn.Sequential(nn.Linear(z_dim, hidden_dim),
                        nn.ReLU(),
                        nn.Linear(hidden_dim, data_dim))

D_model = nn.Sequential(nn.Linear(data_dim, hidden_dim),
                        nn.ReLU(),
                        nn.Linear(hidden_dim, 1),
                        nn.Sigmoid())

G_optimizer =  optim.Adam(G_model.parameters(),
                          lr = lr)
D_optimizer =  optim.Adam(D_model.parameters(),
                          lr = lr)
```

```
z_dim, data_dim = 8, 2
hidden_dim = 100
batch_size = 10
lr = 1e-3

G_model = nn.Sequential(nn.Linear(z_dim, hidden_dim),
                        nn.ReLU(),
                        nn.Linear(hidden_dim, data_dim))

D_model = nn.Sequential(nn.Linear(data_dim, hidden_dim),
                        nn.ReLU(),
                        nn.Linear(hidden_dim, 1),
                        nn.Sigmoid())

G_optimizer =  optim.Adam(G_model.parameters(),
                          lr = lr)
D_optimizer =  optim.Adam(D_model.parameters(),
                          lr = lr)
```

GAN training

```
z_dim, data_dim = 8, 2
hidden_dim = 100
batch_size = 10
lr = 1e-3

G_model = nn.Sequential(nn.Linear(z_dim, hidden_dim),
                        nn.ReLU(),
                        nn.Linear(hidden_dim, data_dim))

D_model = nn.Sequential(nn.Linear(data_dim, hidden_dim),
                        nn.ReLU(),
                        nn.Linear(hidden_dim, 1),
                        nn.Sigmoid())

G_optimizer =  optim.Adam(G_model.parameters(),
                          lr = lr)
D_optimizer =  optim.Adam(D_model.parameters(),
                          lr = lr)
```

```
                              lr = lr)
```

```
for iters in range(epochs_num):

    for t, real_batch in \
      enumerate(real_samples.split(batch_size)):

        z = real_batch.new_empty((real_batch.size(0),
                                  z_dim)).normal_()
        fake_batch = G_model(z)

        real_D_scores = D_model(real_batch)
        fake_D_scores = D_model(fake_batch)

        if t%2 == 0:
            loss = -fake_D_scores.log().mean()
            G_optimizer.zero_grad()
            loss.backward()
            G_optimizer.step()
        else:
            loss = (- (1 - fake_D_scores).log().mean()
                    - real_D_scores.log().mean())
            D_optimizer.zero_grad()
```

```
for t, real_batch in \
  enumerate(real_samples.split(batch_size)):

    z = real_batch.new_empty((real_batch.size(0),
                              z_dim)).normal_()
    fake_batch = G_model(z)

    real_D_scores = D_model(real_batch)
    fake_D_scores = D_model(fake_batch)

    if t%2 == 0:
        loss = -fake_D_scores.log().mean()
        G_optimizer.zero_grad()
        loss.backward()
        G_optimizer.step()
    else:
        loss = (- (1 - fake_D_scores).log().mean()
                - real_D_scores.log().mean())
        D_optimizer.zero_grad()
        loss.backward()
        D_optimizer.step()
```

```python
for t, real_batch in \
  enumerate(real_samples.split(batch_size)):

    z = real_batch.new_empty((real_batch.size(0),
                              z_dim)).normal_()
    fake_batch = G_model(z)

    real_D_scores = D_model(real_batch)
    fake_D_scores = D_model(fake_batch)

    if t%2 == 0:
       loss = -fake_D_scores.log().mean()
       G_optimizer.zero_grad()
       loss.backward()
       G_optimizer.step()
    else:
       loss = (- (1 - fake_D_scores).log().mean()
               - real_D_scores.log().mean())
       D_optimizer.zero_grad()
       loss.backward()
       D_optimizer.step()
```

```
for t, real_batch in \
  enumerate(real_samples.split(batch_size)):

   z = real_batch.new_empty((real_batch.size(0),
                             z_dim)).normal_()
   fake_batch = G_model(z)
                 _model(real_batch)
   fake_D_scores = D_model(fake_batch)

   if t%2 == 0:
      loss = -fake_D_scores.log().mean()
      G_optimizer.zero_grad()
      loss.backward()
      G_optimizer.step()
   else:
      loss = (- (1 - fake_D_scores).log().mean()
              - real_D_scores.log().mean())
      D_optimizer.zero_grad()
      loss.backward()
      D_optimizer.step()
```

generate random latent vectors

```python
for t, real_batch in \
  enumerate(real_samples.split(batch_size)):

    z = real_batch.new_empty((real_batch.size(0),
                              z_dim)).normal_()
    fake_batch = G_model(z)

    real_D_scores = D_model(real_batch)
    fake_D_scores = D_model(fake_batch)

    if t%2 == 0:
       loss = -fake_D_scores.log().mean()
       G_optimizer.zero_grad()
       loss.backward()
       G_optimizer.step()
    else:
       loss = (- (1 - fake_D_scores).log().mean()
               - real_D_scores.log().mean())
       D_optimizer.zero_grad()
       loss.backward()
       D_optimizer.step()
```

```
for t, real_batch in \
  enumerate(real_samples.split(batch_size)):

    z = real_batch.new_empty((real_batch.size(0),
                              z_dim)).normal_()
    fake_batch = G_model(z)

    real_D_scores = D_model(real_batch)
    fake_D_scores = D_model(fake_batch)

    if t%2 == 0:
       loss = -fake_D_scores.log().mean()
       G_optimizer.zero_grad()
       loss.backward()
       G_optimizer.step()
    else:
       loss = (- (1 - fake_D_scores).log().mean()
               - real_D_scores.log().mean())
       D_optimizer.zero_grad()
       loss.backward()
       D_optimizer.step()
```

```
real_D_scores = D_model(real_batch)
fake_D_scores = D_model(fake_batch)
```

```
if t%2 == 0:
   loss = -fake_D_scores.log().mean()
   G_optimizer.zero_grad()
   loss.backward()
   G_optimizer.step()
else:
   loss = (- (1 - fake_D_scores).log().mean()
           - real_D_scores.log().mean())
   D_optimizer.zero_grad()
   loss.backward()
   D_optimizer.step()
```

```
real_D_scores = D_model(real_batch)
fake_D_scores = D_model(fake_batch)

if t%2 == 0:
   loss = -fake_D_scores.log().mean()
   G_optimizer.zero_grad()
   loss.backward()
   G_optimizer.step()
else:
               - real_D_scores.log().mean())
   D_optimizer.zero_grad()
   loss.backward()
   D_optimizer.step()
```

update generator using modified objective

```
real_D_scores = D_model(real_batch)
fake_D_scores = D_model(fake_batch)

if t%2 == 0:
   loss = -fake_D_scores.log().mean()
   G_optimizer.zero_grad()
   loss.backward()
   G_optimizer.step()
else:
   loss = (- (1 - fake_D_scores).log().mean()
           - real_D_scores.log().mean())
   D_optimizer.zero_grad()
   loss.backward()
   D_optimizer.step()
```

update discriminator

GAN training

```
z_dim, data_dim = 8, 2
hidden_dim = 100
batch_size = 10
lr = 1e-3

G_model = nn.Sequential(nn.Linear(z_dim, hidden_dim),
                        nn.ReLU(),
                        nn.Linear(hidden_dim, data_dim))

D_model = nn.Sequential(nn.Linear(data_dim, hidden_dim),
                        nn.ReLU(),
                        nn.Linear(hidden_dim, 1),
                        nn.Sigmoid())

G_optimizer = optim.Adam(G_model.parameters(),
                         lr = lr)
D_optimizer = optim.Adam(D_model.parameters(),
                         lr = lr)

for iters in range(epochs_num):

   for t, real_batch in \
     enumerate(real_samples.split(batch_size)):

      z = real_batch.new_empty((real_batch.size(0),
                                 z_dim)).normal_()
      fake_batch = G_model(z)

      real_D_scores = D_model(real_batch)
      fake_D_scores = D_model(fake_batch)

      if t%2 == 0:
         loss = -fake_D_scores.log().mean()
         G_optimizer.zero_grad()
         loss.backward()
         G_optimizer.step()
      else:
         loss = (- (1 - fake_D_scores).log().mean()
                 - real_D_scores.log().mean())
         D_optimizer.zero_grad()
         loss.backward()
         D_optimizer.step()
```

How is the quality of generated images assessed?

Two simple properties for evaluation metric:

- Fidelity: We want our GAN to generate *high* quality images.
- Diversity: Our GAN should generate images that are inherent in the training dataset.

Feature Distance:

- Use a pre-trained image classification model (neural network).
- Pass an image through the model and use the activation of intermediate layers as features.
- Calculate any distance metric (L2/L1) between the features of generated image and GT real image.
- LPIPS metric (Learned Perceptual Image Patch Similarity).

But often, we do not have the GT image to compare with.

What do we do?

FID (Frechet Inception Distance)

Frechet Distance between two univariate gaussian distribution

$$d(X, Y) = (\mu_X - \mu_Y)^2 + (\sigma_X - \sigma_Y)^2$$

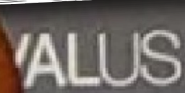
Frechet Distance between two multi-variate gaussian distribution

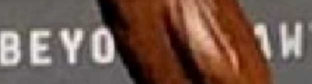
$$FID = ||\mu_X - \mu_Y||^2 - Tr(\sum_X + \sum_Y - 2 \quad \sum_X \sum_Y)$$

Frechet Inception Distance (FID), X and Y are features of Inception V3 classification model for real and fake images respectively.

Note: The loss is between set of real and fake images, not individual real and fake image!

results typically assessed visually





Important Deadlines

- 590: Assignment 2 announced, due Sept 8.
- 590/790: Paper presentation/review schedule announced
- 790: Deadline to register your project group, Sept 1! 1 points deducted per late day!
- 790: Project Proposal presentation is due Sept 20!

Slide Credits

- EECS 6322 Deep Learning for Computer Vision, Kosta Derpanis (York University)
- EECS 498 Deep Learning for Computer Vision, Justin Johnson (U. Michigan)
- Many amazing research papers!