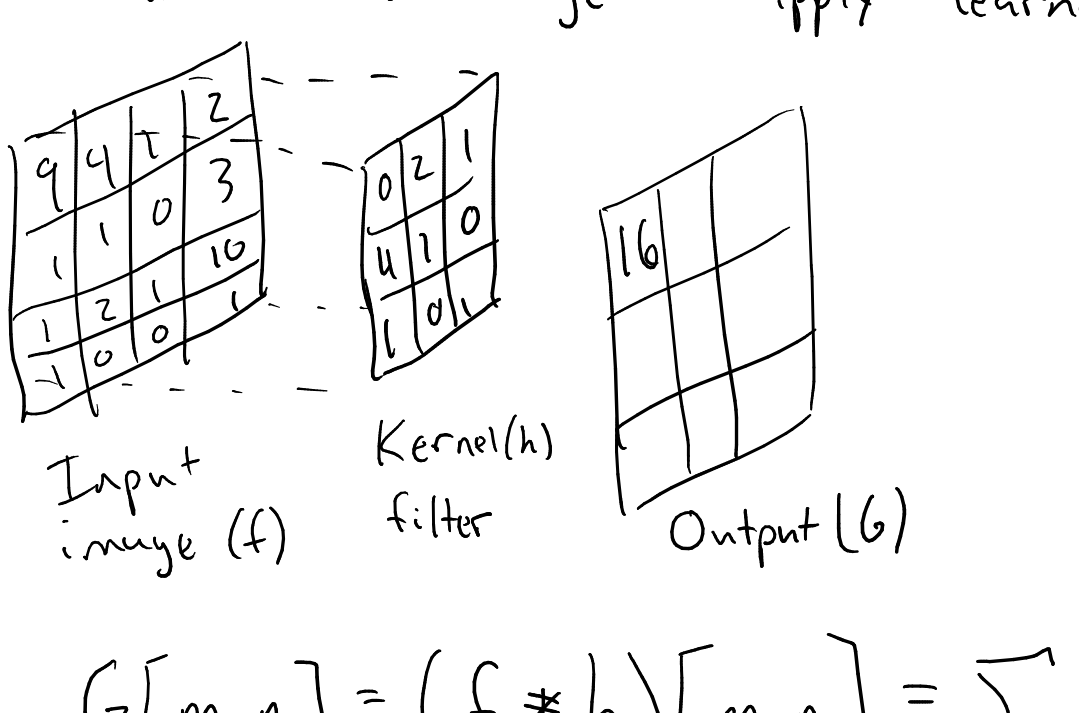
Convolutional NN

- Take 2D image + apply learnable kernel



$$G[m,n] = (f * h)[m,n] = \sum_j \sum_k h[j,k]\, f[m-j, n-k]$$

Question 1: What happens to image size when we apply one of these layers? Do you forsee any problems? If yes, how do you solve it?

Answer 1:
- Image size decreases (by the dim of $h-1$)
- Limited in how many we can apply
- Padding fixes this

Padding:

Maintain image size by adding $p = \frac{h-1}{2}$ pixels on each edge

Typically, new pixels are set to 0.

Strided convolutions:

Instead of stepping by 1 pixel, step by s pixels
Reduces overlap of pixels in convolution, reduces image size

$$n_{out} = \left\lfloor \frac{n_{in} + 2p - h}{s} + 1 \right\rfloor$$

(p ← padding, h ← filter size, s ↑ stride)

final output is similar to DNN but convolution instead of matrix mult.

$$z^{(l)} = w^{(l)} * x^{(l-1)} + b^{(l)}$$

(w ↑ kernel)

$$x^{(l)} = a(z^{(l)})$$

(a ↑ activation function)

Convert to DNN: Flatten the image

Generative networks

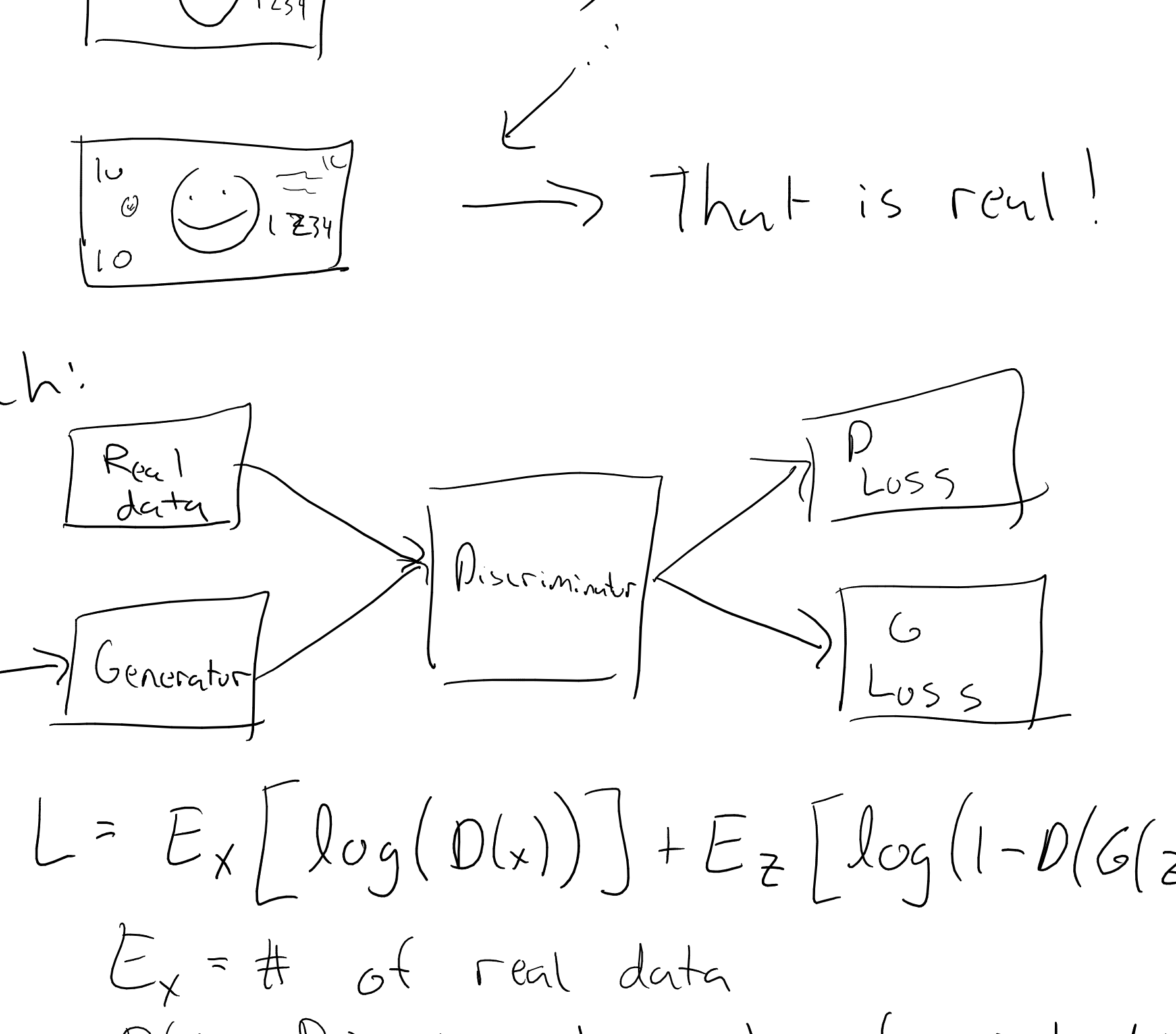
Question 2: In classifiers, we want a computer to learn key differences between classes. What might we want to learn for generation based on some input data?

Answer 2:
- Minimize $\|x - \hat{x}\|^2$
- Lower dimensional rep of input
- Learn inverse (easier if Unitary)
- Learn commonalities

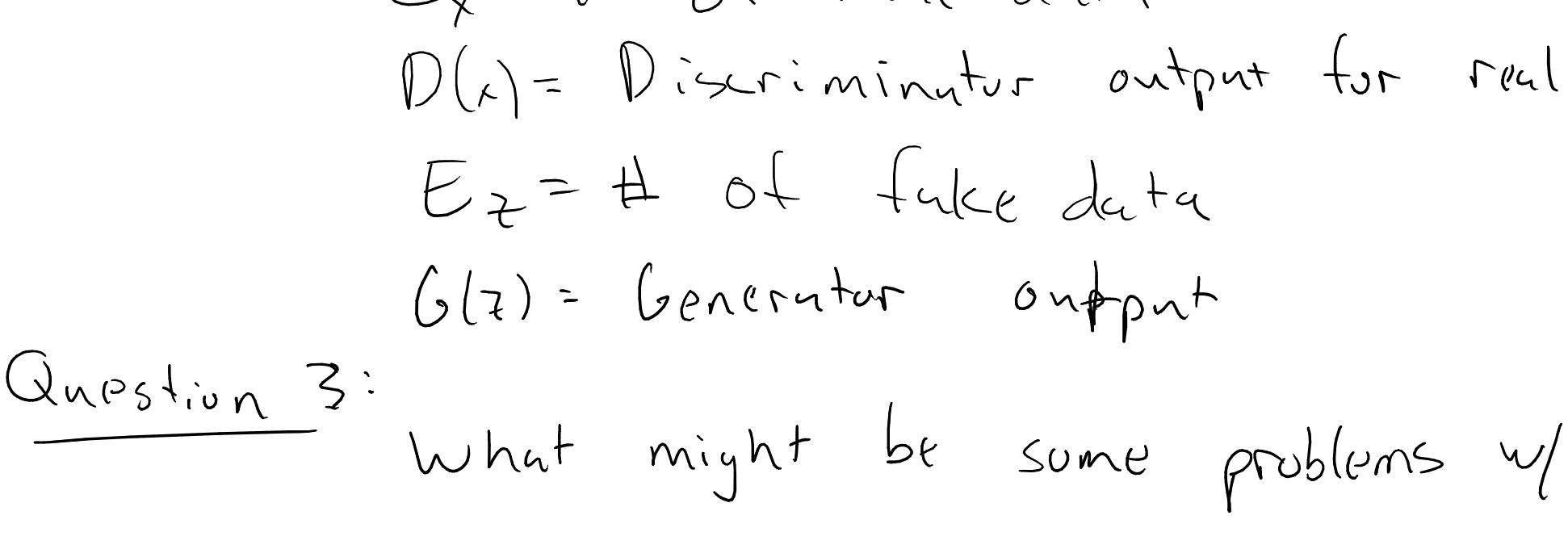
Main goal: Is to learn the PDF (CDF)

Generative Adversarial Networks (GANs)

1. Generator (forger)
2. Discriminator (detective)



Arch:



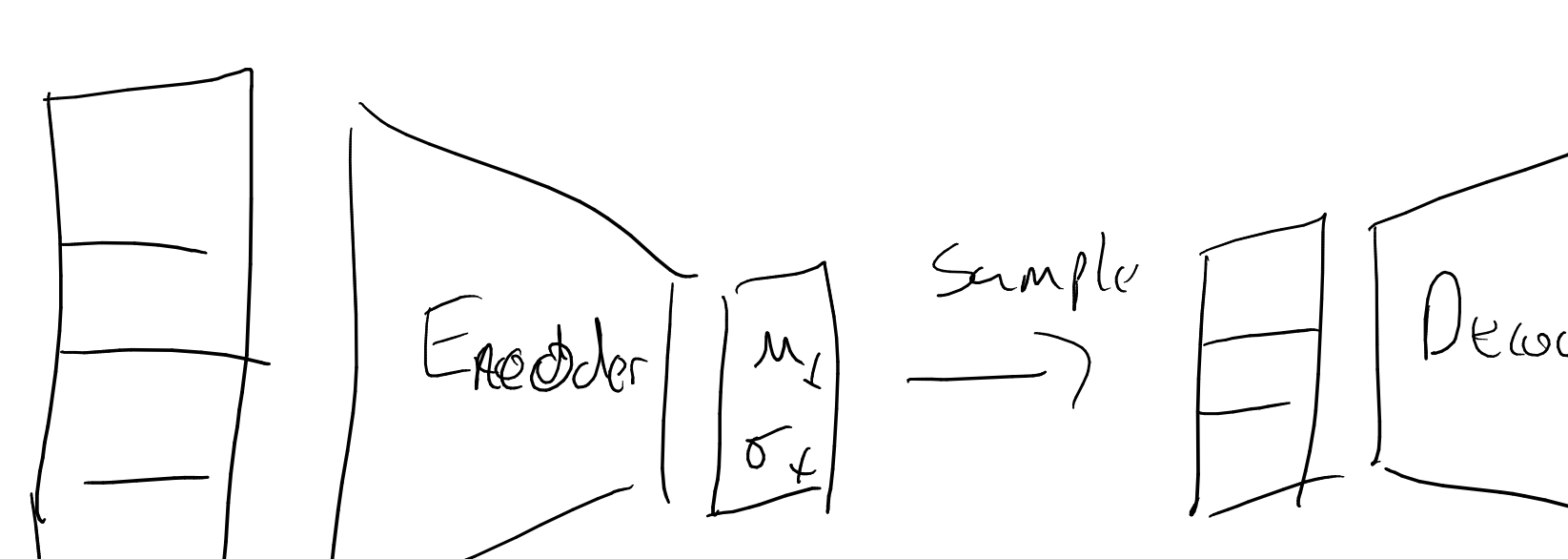
$$L = E_x[\log(D(x))] + E_z[\log(1 - D(G(z)))]$$

E_x = # of real data
$D(x)$ = Discriminator output for real data
E_z = # of fake data
$G(z)$ = Generator output

Question 3: What might be some problems w/GANs?

Answer 3:
- Too good discriminator (Vanishing gradient problem)
  EMD can fix this
- Failure to converge:
  Fix by adding noise to D input or penalizing D weights
- Need tons of data
- Mode collapse: Discriminator gets stuck in local min, G only generates the same output for all inputs
  Fixed with EMD

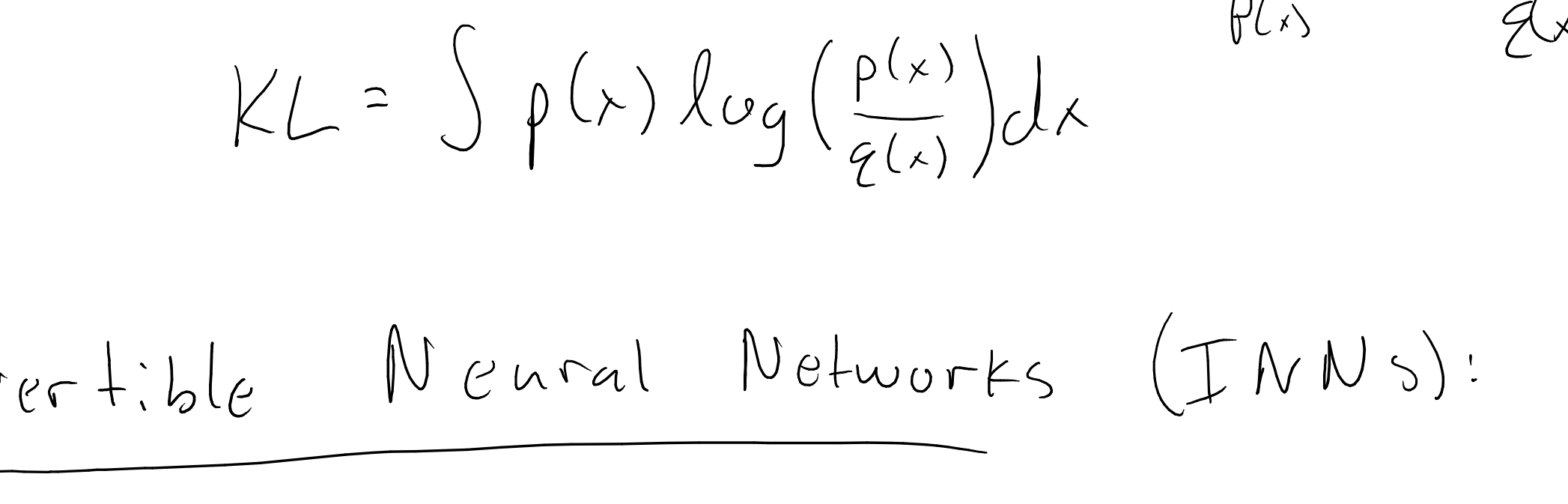
Variational Autoencoders (VAEs):



Autoencoder

X $\quad z = e(x)$ $\quad \hat{x} = d(z)$

$$\text{Loss} = \|x - \hat{x}\|^2 = \|x - d(e(x))\|^2$$



X $\quad N(\mu_x, \sigma_x)$ $\quad z \sim N(\mu_x, \sigma_x)$ $\quad \hat{x} = d(z)$

$$\text{Loss} = \|x - d(z)\|^2 - KL[\underbrace{N(\mu_x, \sigma_x)}_{p(x)}, \underbrace{N(0,1)}_{q(x)}]$$

$$KL = \int p(x) \log\left(\frac{p(x)}{q(x)}\right) dx$$

Invertible Neural Networks (INNs):

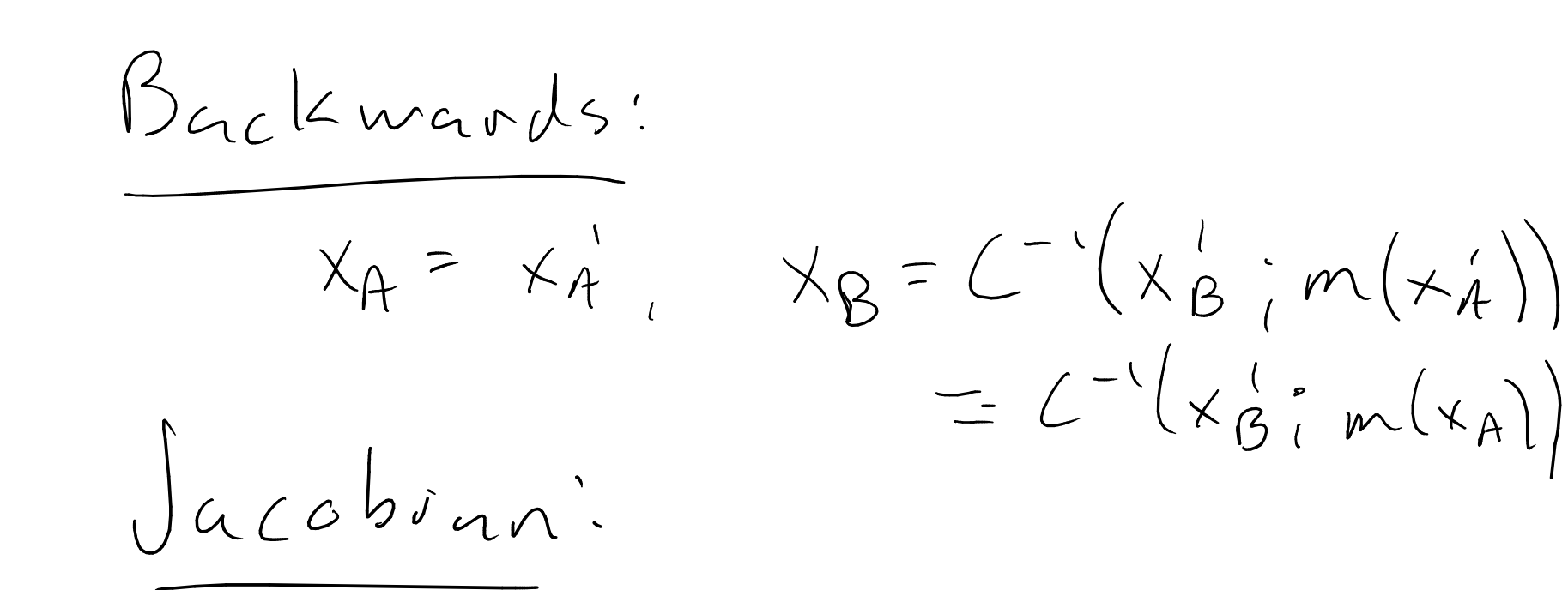
Goal: Transform simple distribution $q_0(x)$ into a "good" approximation of the data distribution $p(y)$
- Find a variable transformation from $x \to y$

Question 4: What are some requirements on such a network? (Hint: Jacobian takes $O(n^3)$ operations for n-dimensional transformations)

Answer 4:
- # of inputs = # of outputs
- $\det|J| \neq 0$
- Invertible
- We don't want a Jacobian of a NN

Masked Autoregressive Flows
Inverse Autoregressive Flows] Not covered

Normalizing Flows (Coupling layers)



Forward:

$x_A' = x_A, \quad A \in [1, d]$

$x_B' = C(x_B; m(x_A)), \quad B \in [d+1, D]$

Backwards:

$x_A = x_A', \quad x_B = C^{-1}(x_B'; m(x_A'))$
$\qquad = C^{-1}(x_B'; m(x_A))$

Jacobian:

$$\left|\frac{\partial c(\vec{x})}{\partial \vec{x}}\right|^{-1} = \left|\begin{pmatrix} I & \vec{0} \\ \frac{\partial C}{\partial m}\frac{\partial m}{\partial x_A} & \frac{\partial C}{\partial x_B} \end{pmatrix}\right|^{-1} = \left|\frac{\partial C(\vec{x}_B, m(\vec{x}_A))}{\partial \vec{x}_B}\right|^{-1}$$

Coupling Layer:

Piecewise Linear: PDF given by (K bin of width w)

$$q_i(t) = \begin{cases} Q_{i1}/w & t < w \\ \vdots & \\ Q_{iK}/w & 1 - w \le t < 1 \end{cases}$$

CDF: $C(x_i^B; Q) = \alpha Q_{ib} + \sum_{k=1}^{b-1} Q_{ik}$

b in the bin x_i^B falls into

$$\alpha = \frac{x_i^B - (b-1)w}{w}$$

Inverse: $x_i^B(C_i; Q) = \frac{w(C_i - \sum_{k=1}^{b-1} Q_{ik})}{Q_{ib}} + (b-1)w$

Jacobian: $\left|\frac{\partial C}{\partial x_B}\right| = \prod_i Q_{ib}/w$

i-flow