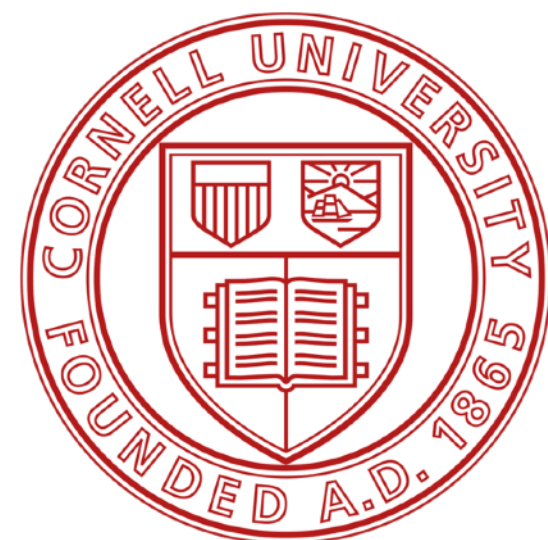


Lecture 5: Variational autoencoders

CS 5788: Introduction to Generative Models



Reminders

- PS1 due Feb. 10
- Make sure to enable GPU
- Questions?

Today

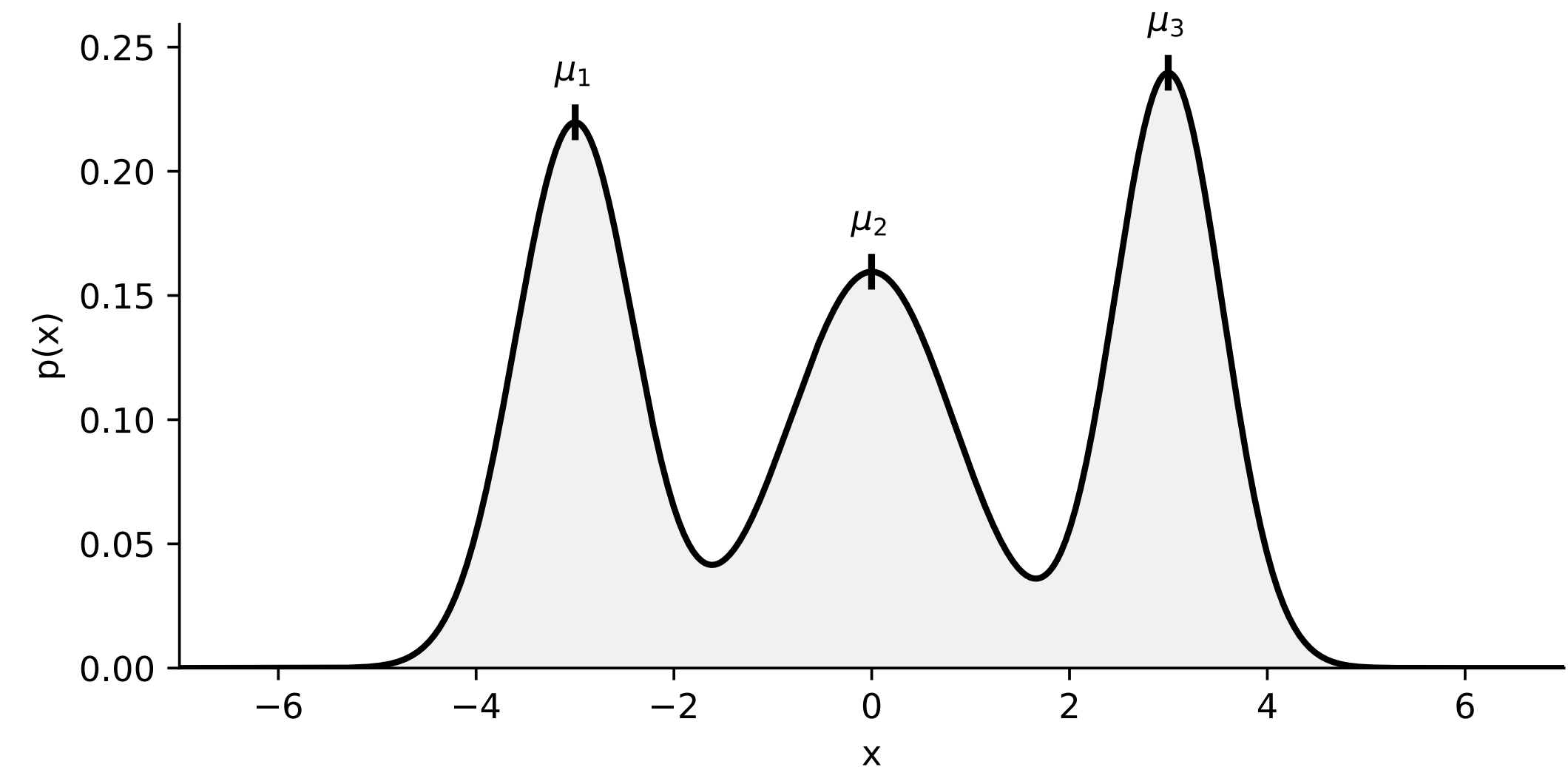
- Continue discussing Gaussian mixture models
- Variational autoencoders

Recall: Gaussian mixture model (GMM)

A useful type of mixture model that mixes multiple Gaussian distributions.

Density function:
$$p_{\theta}(\mathbf{x}) = \sum_{i=1}^K \pi_i \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$$

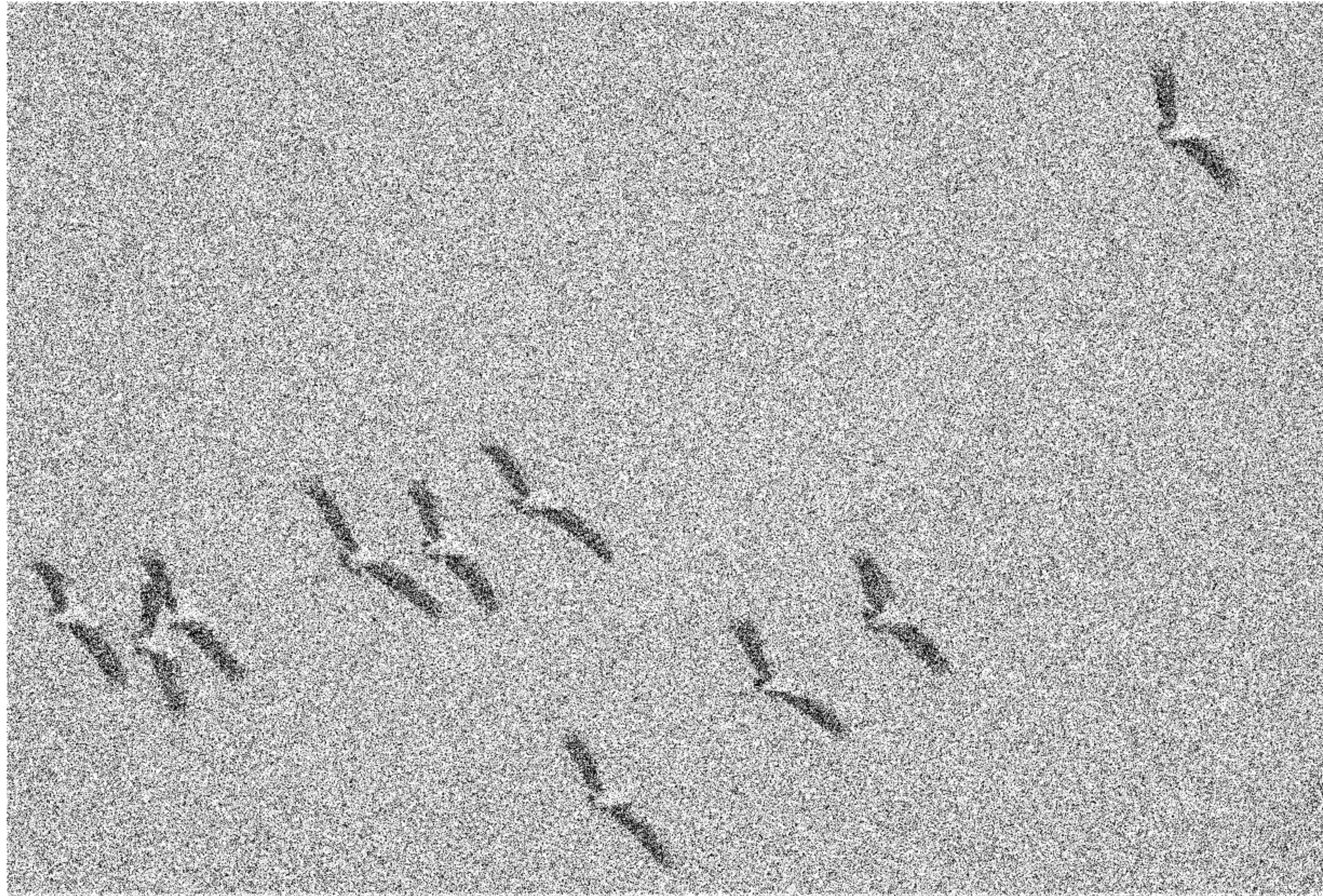
where π_i is the probability of choosing mixture component i , and $\mathcal{N}$ is the Gaussian pdf with mixture-specific mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$.



Recall: GMM as a latent variable model

- The variable z is the identity of the mixture component that generated the example. We call it a **latent variable** because we never directly observe it.
- Latent variables allow us to create more powerful generative models.
- They allow us to capture complex dependencies between variables.
- We'll see many other types of latent variable models in this course.

Image denoising

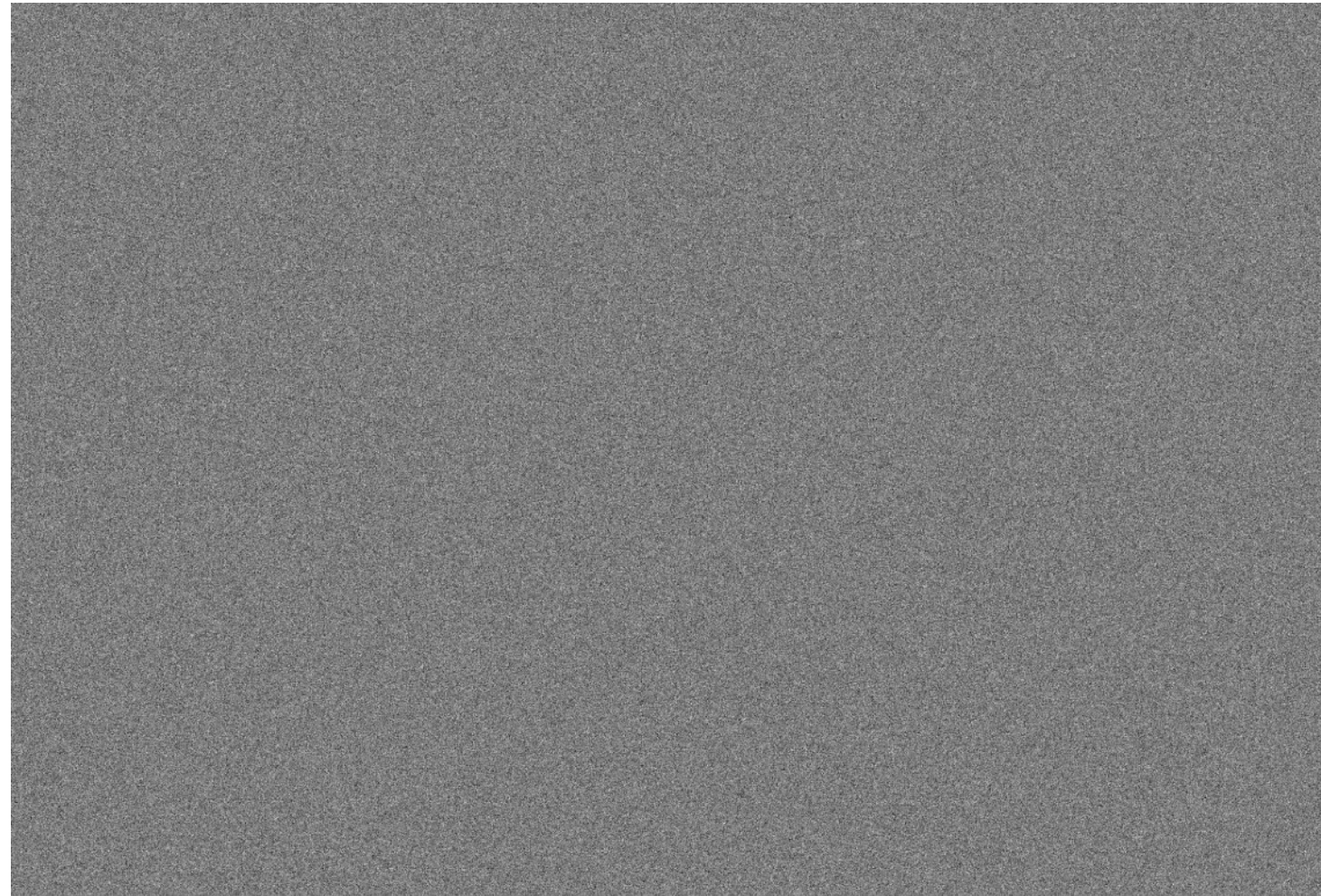


Application: image denoising

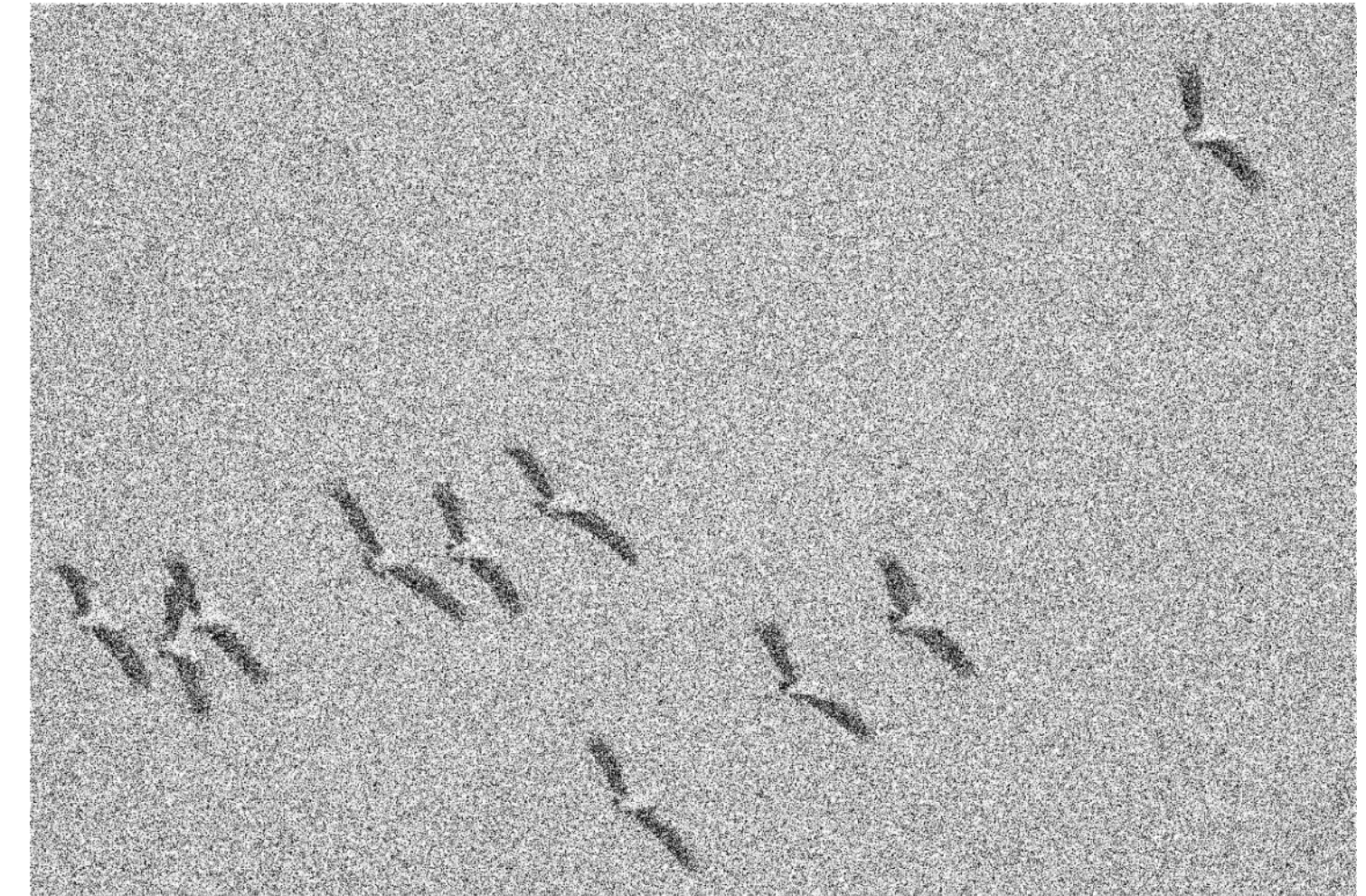
image



noise



noisy image



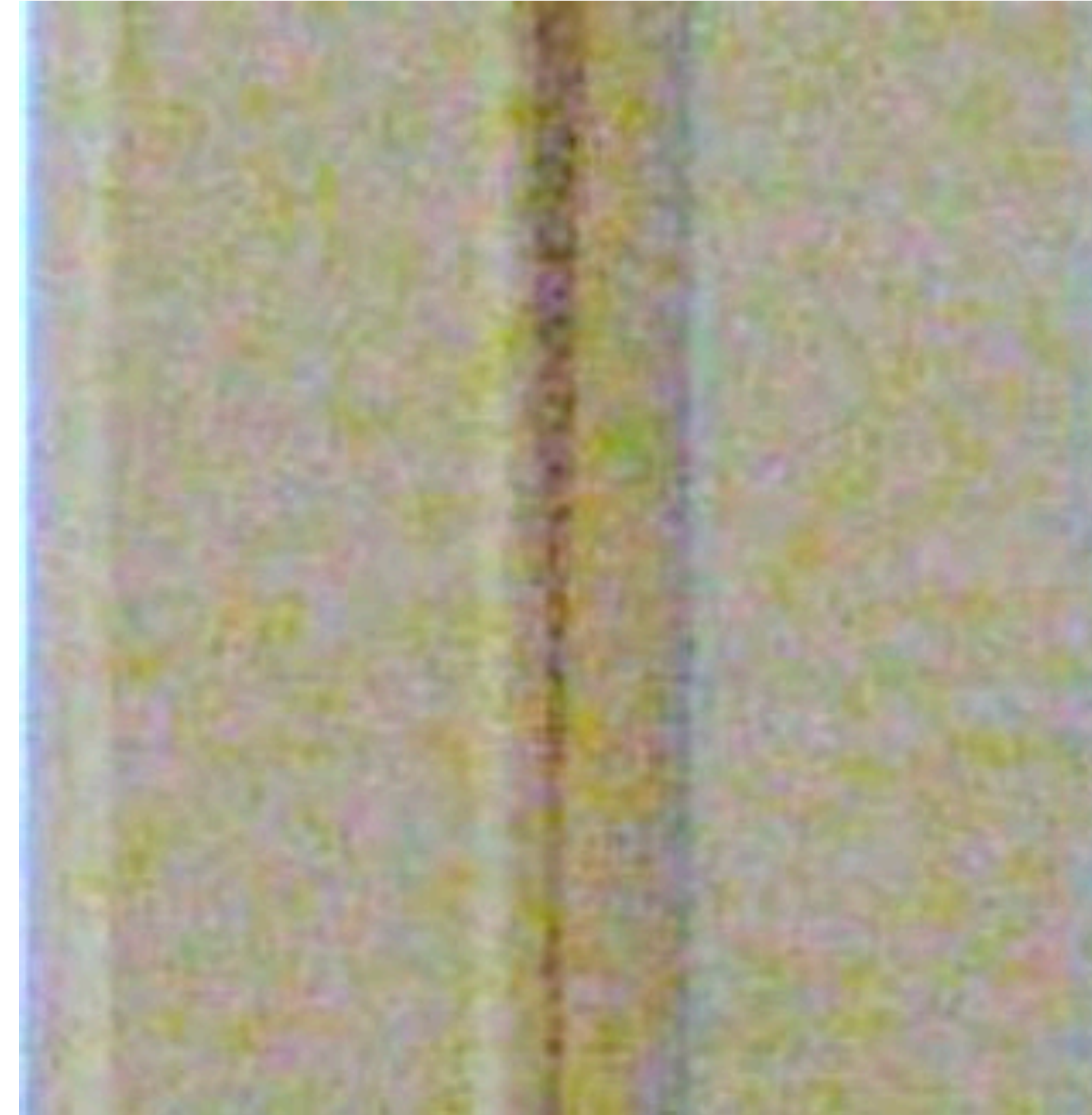
+

=



Goal: recover the original image

Image denoising problem



In practice: low light photography, “dead” pixels, etc.

Denoising using a prior

By Bayes rule:

$$p(\mathbf{X}_{clean} | \mathbf{X}_{noisy}) = \frac{p(\mathbf{X}_{noisy} | \mathbf{X}_{clean})p(\mathbf{X}_{clean})}{p(\mathbf{X}_{noisy})}$$

We're solving for $\mathbf{X}_{clean}$ which is equivalent to *minimizing*:

$$L(\mathbf{X}_{clean}) = -\log p(\mathbf{X}_{noisy} | \mathbf{X}_{clean}) - \log p(\mathbf{X}_{clean})$$

What are good choices for these two terms?

Denoising using a GMM image prior

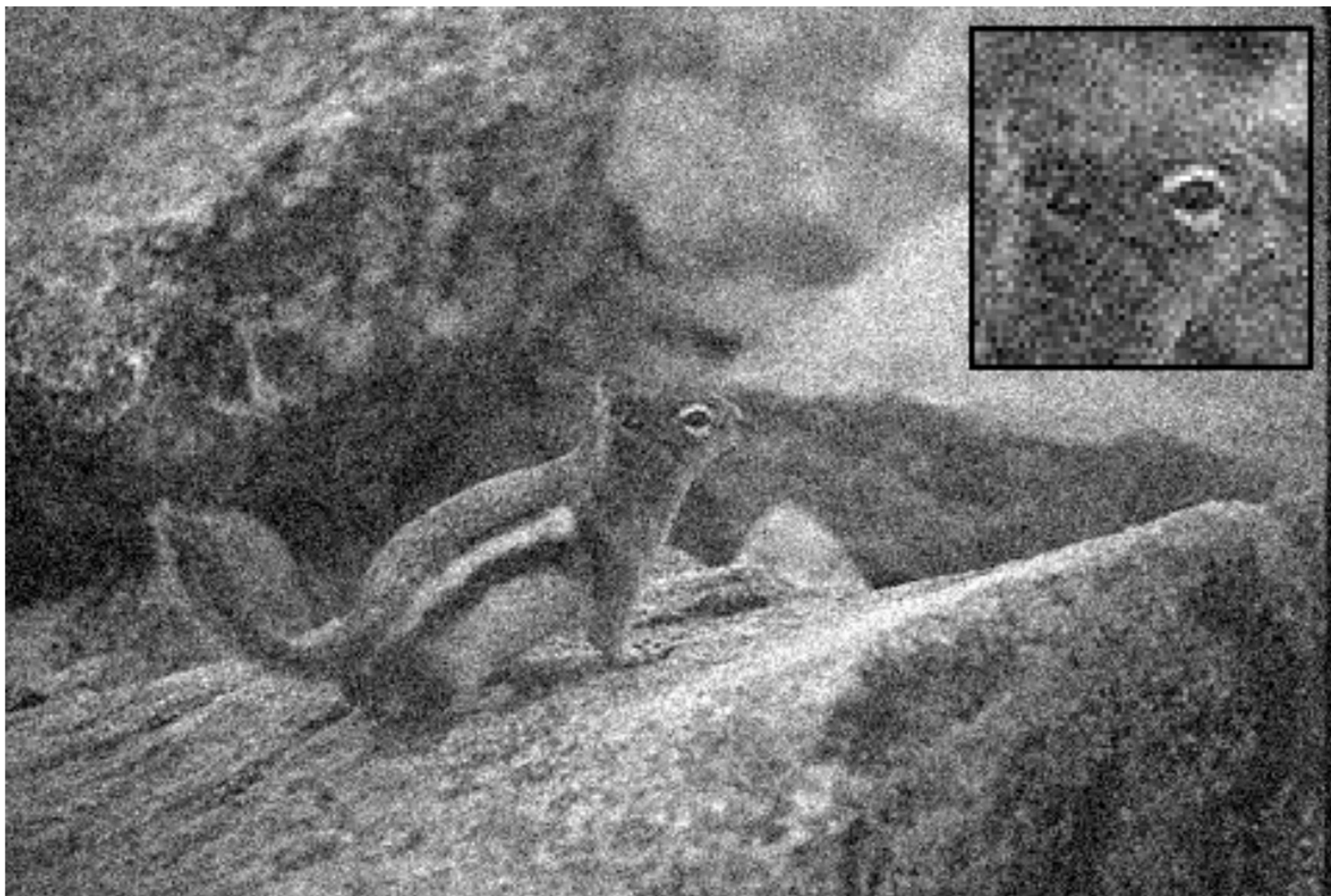
$$L(\hat{\mathbf{X}}) \approx \|\hat{\mathbf{X}} - \mathbf{X}_{noisy}\|^2 - \lambda \text{EPLL}(\hat{\mathbf{X}}),$$

- First term (the likelihood) equivalent to Gaussian centered on $\mathbf{X}_{noisy}$.
- Since we don't have a full image prior, let's use the "expected log patch likelihood": the average log likelihood our GMM assigns to each patch in the image:

$$\text{EPLL}(\mathbf{X}) = \frac{1}{|P(\mathbf{X})|} \sum_{\mathbf{x} \in P(\mathbf{X})} \log p_{\theta}(\mathbf{x})$$

How do we solve for $\hat{\mathbf{X}}$? Gradient descent!

Image denoising with a GMM



Input image

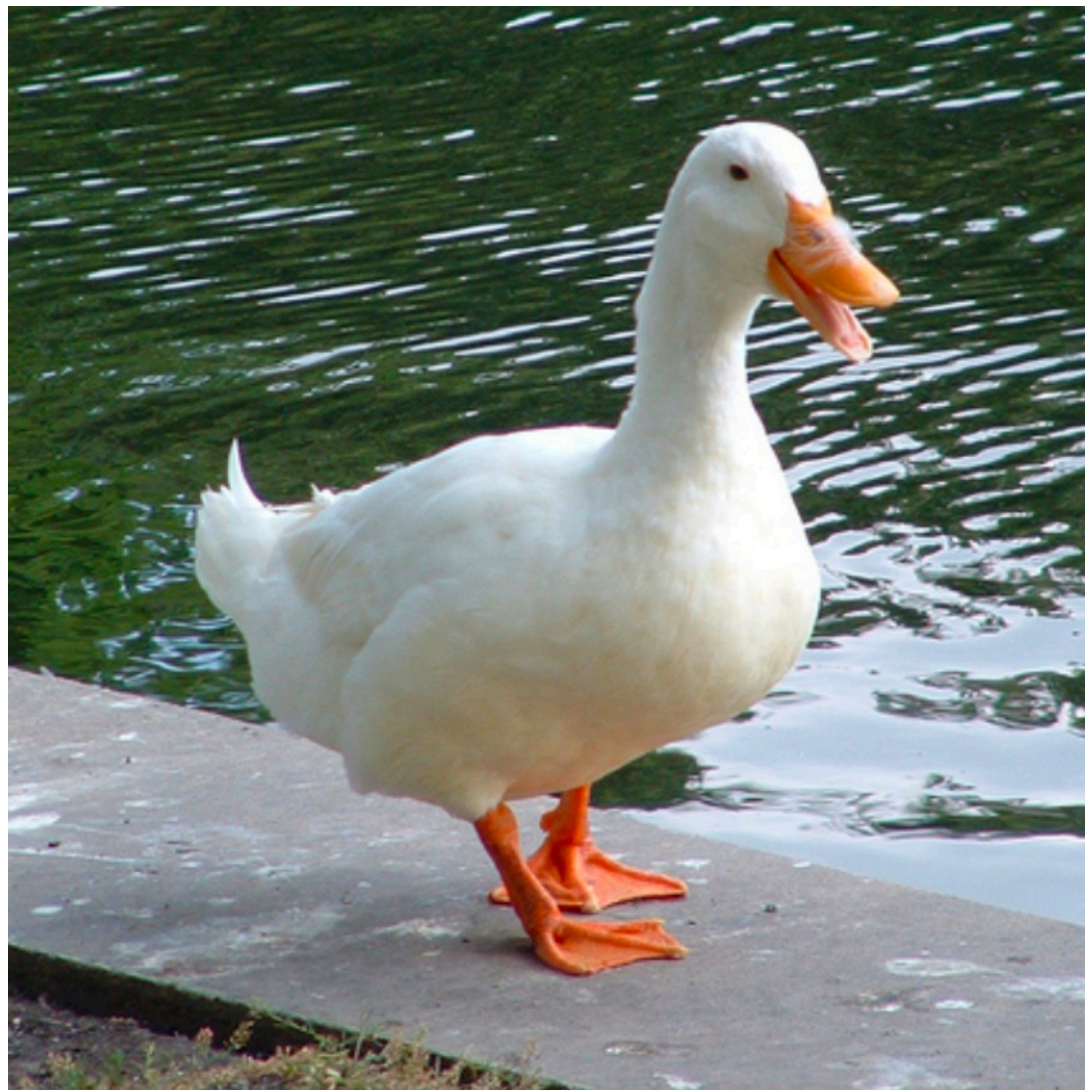


Denoised image

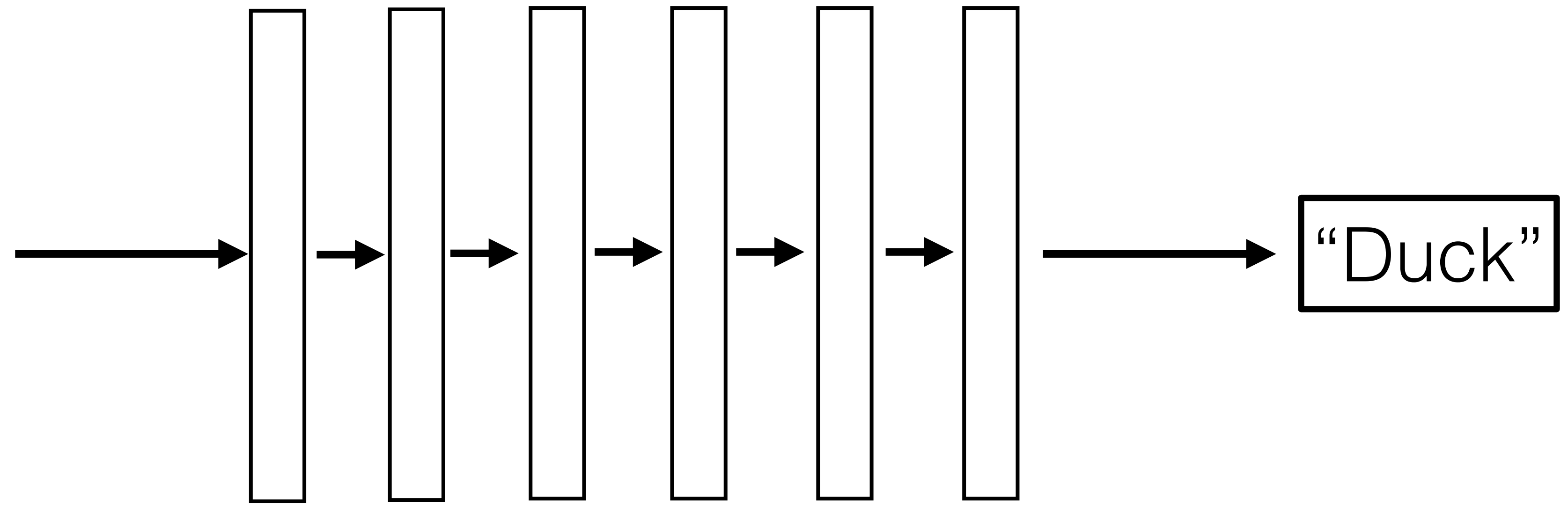
Today

- Continue discussing Gaussian mixture models
- **Variational autoencoders**

Image classification



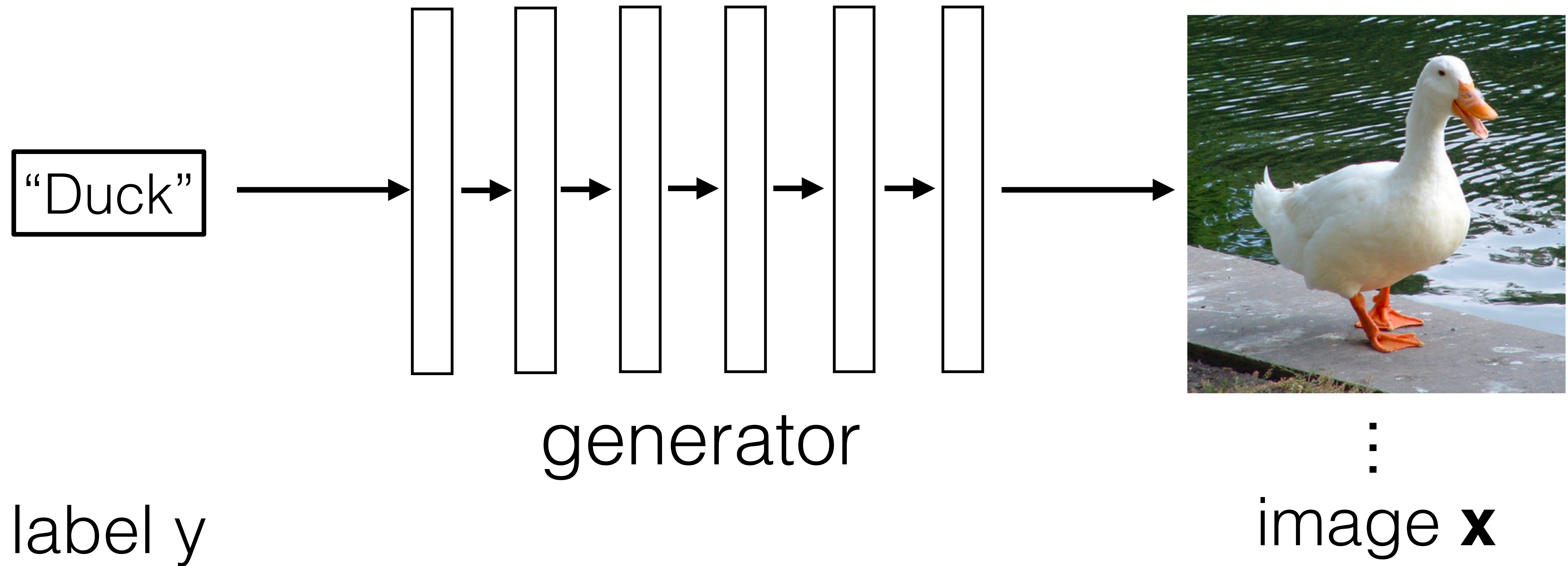
⋮
image $\mathbf{x}$



classifier

label y

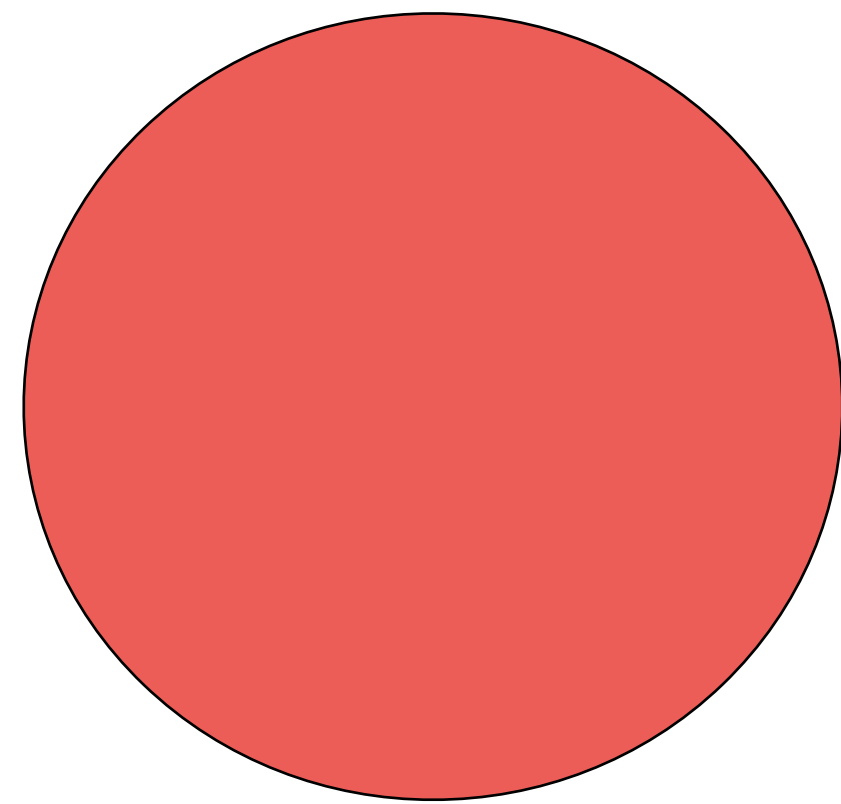
Image synthesis



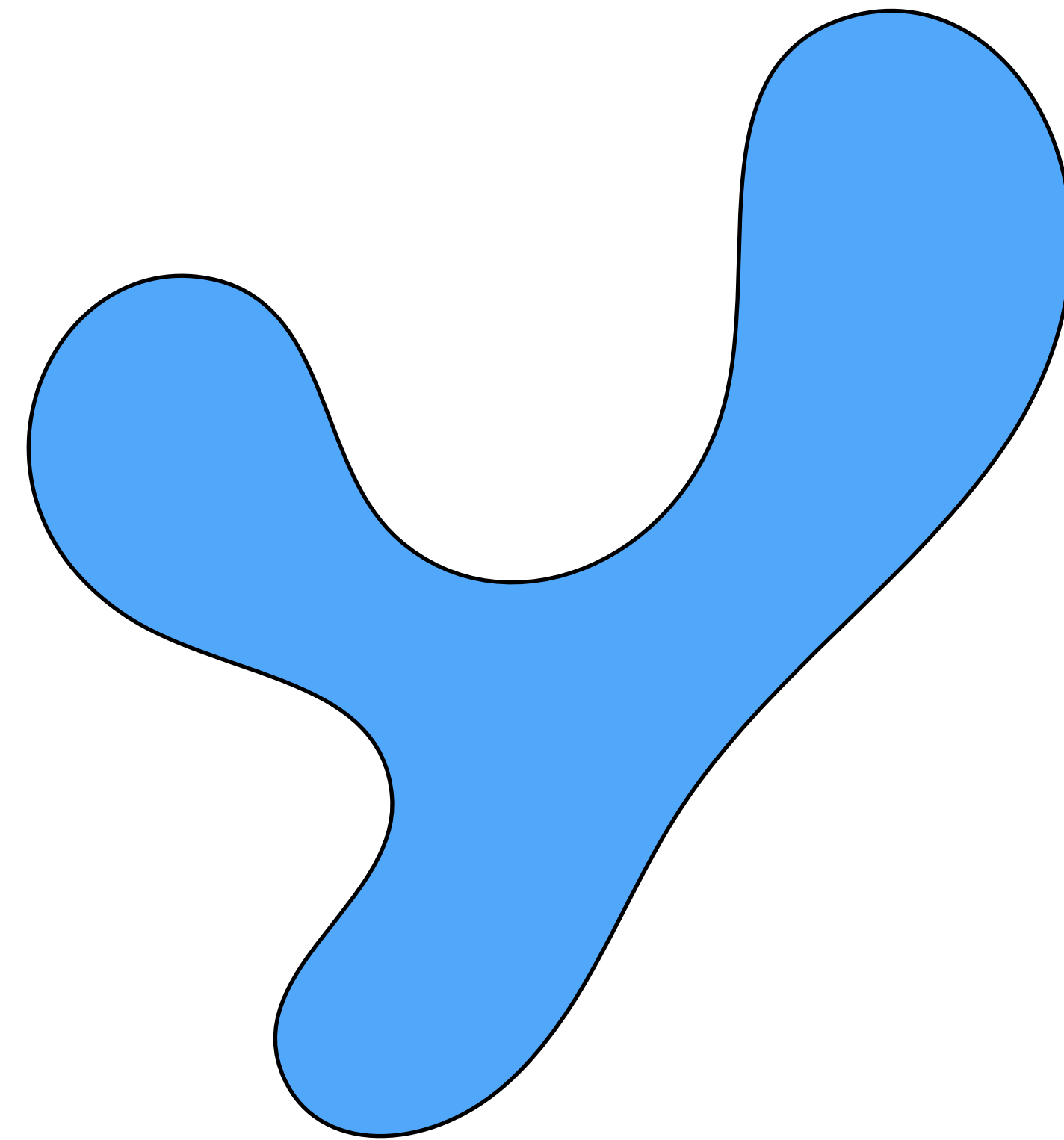
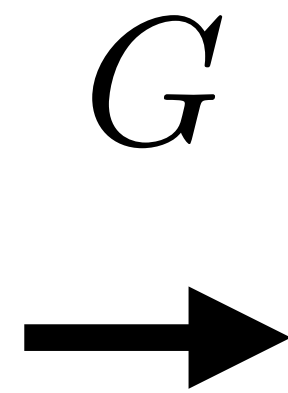
Neural networks as distribution transformers

Source distribution

Target distribution

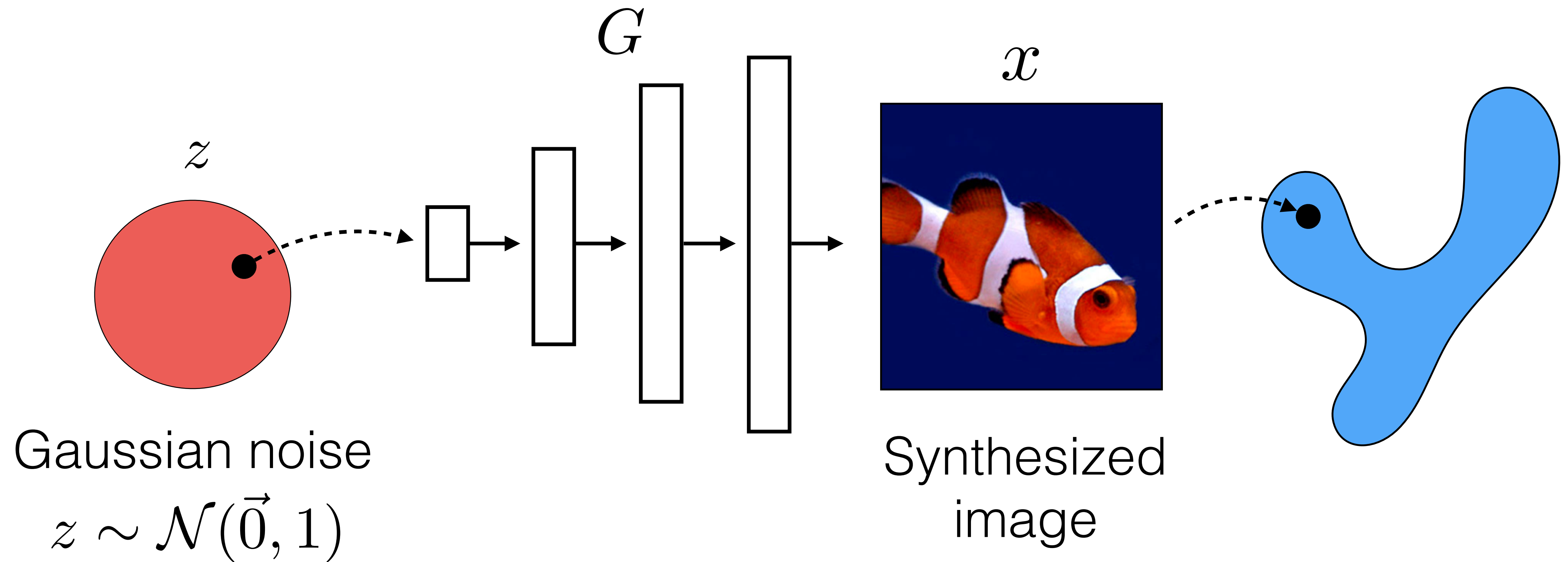


$p(z)$

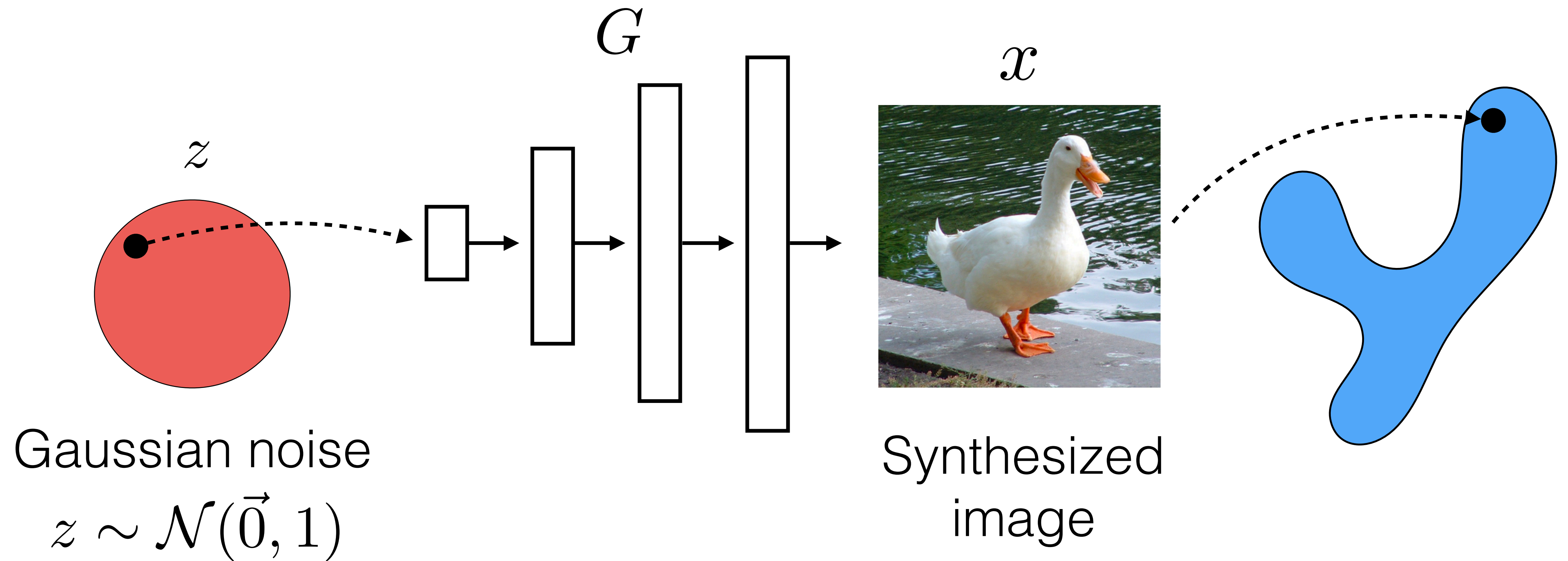


$p(x)$

Neural networks as distribution transformers



Neural networks as distribution transformers



How can we learn this model?

Maximum likelihood learning:

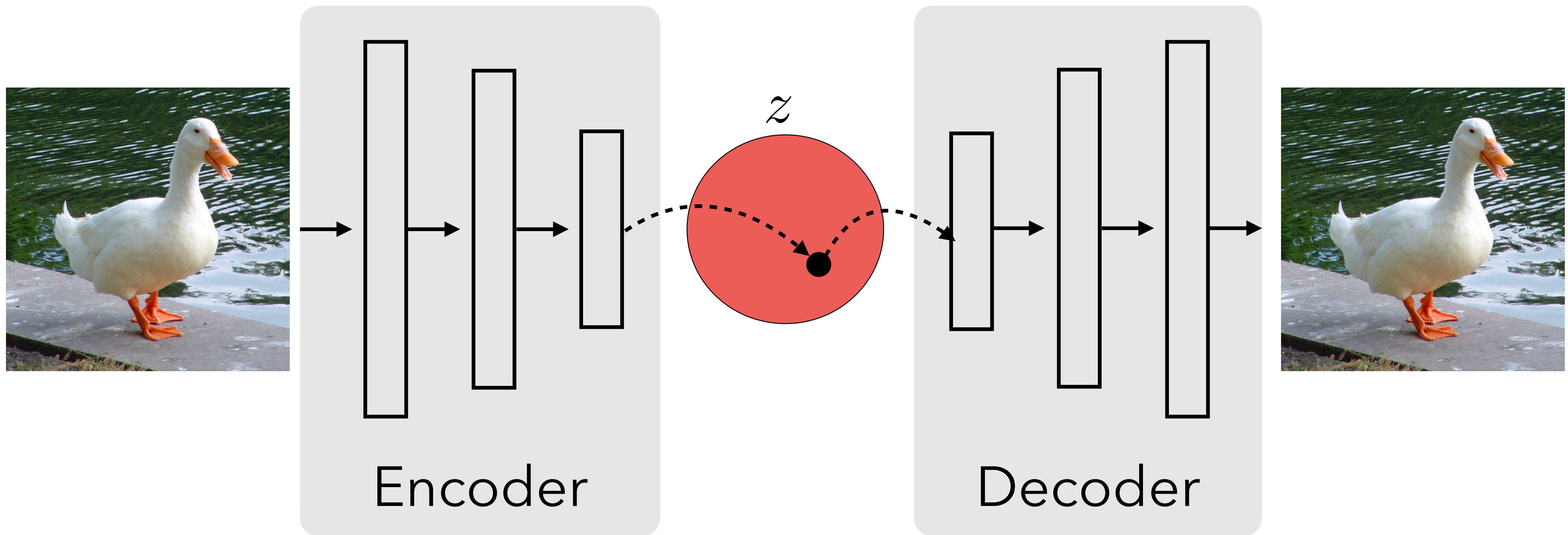
$$\operatorname{argmax}_{\theta} \sum_{i=1}^N \log(p_{\theta}(\mathbf{x}_i))$$

integrate over all of the $\mathbf{z}$ vectors that can produce $\mathbf{x}_i$!

$$= \operatorname{argmax}_{\theta} \sum_{i=1}^N \log \left(\int_{\mathbf{z}} p_{\theta}(\mathbf{x}_i, \mathbf{z}) d\mathbf{z} \right)$$

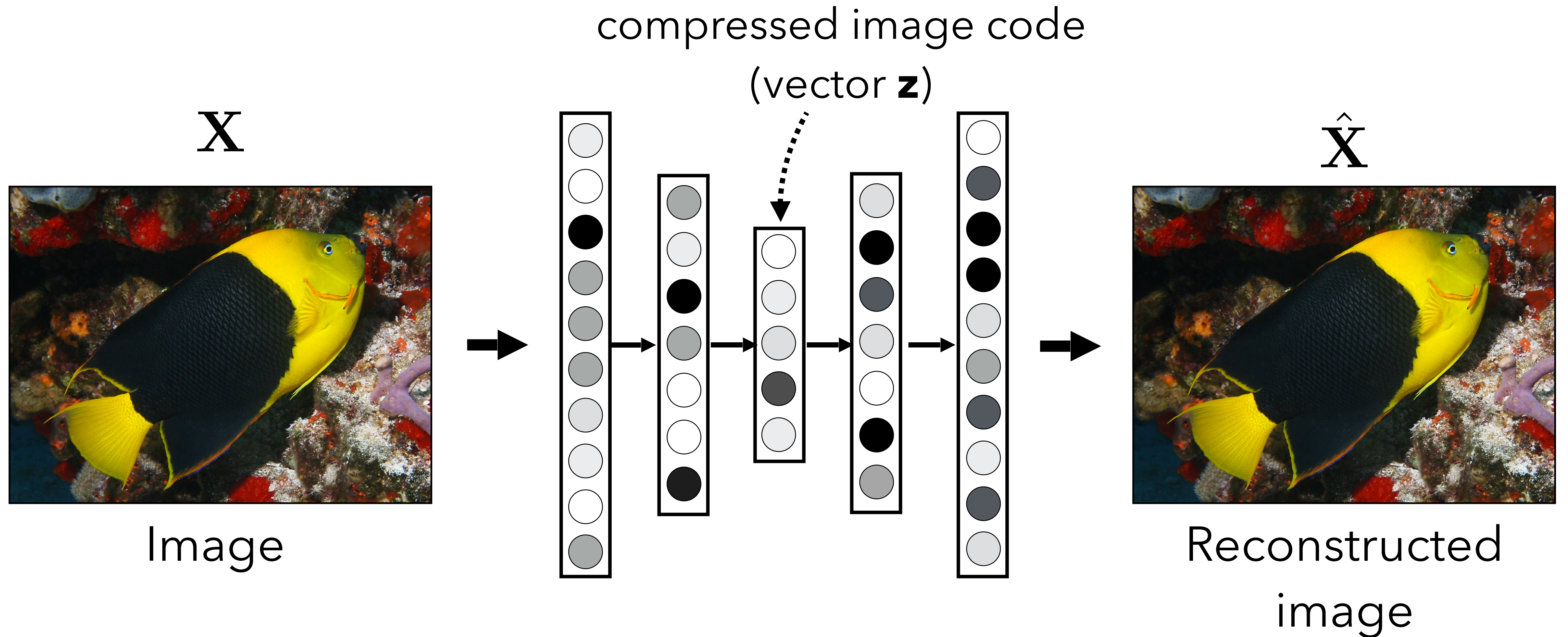
This integral is expensive to compute!

Encoder-decoder models



Assign each image a vector $\mathbf{z}$ that can be decoded to reconstruct the image.

Idea #1: Autoencoder



[e.g., Hinton & Salakhutdinov, Science 2006]

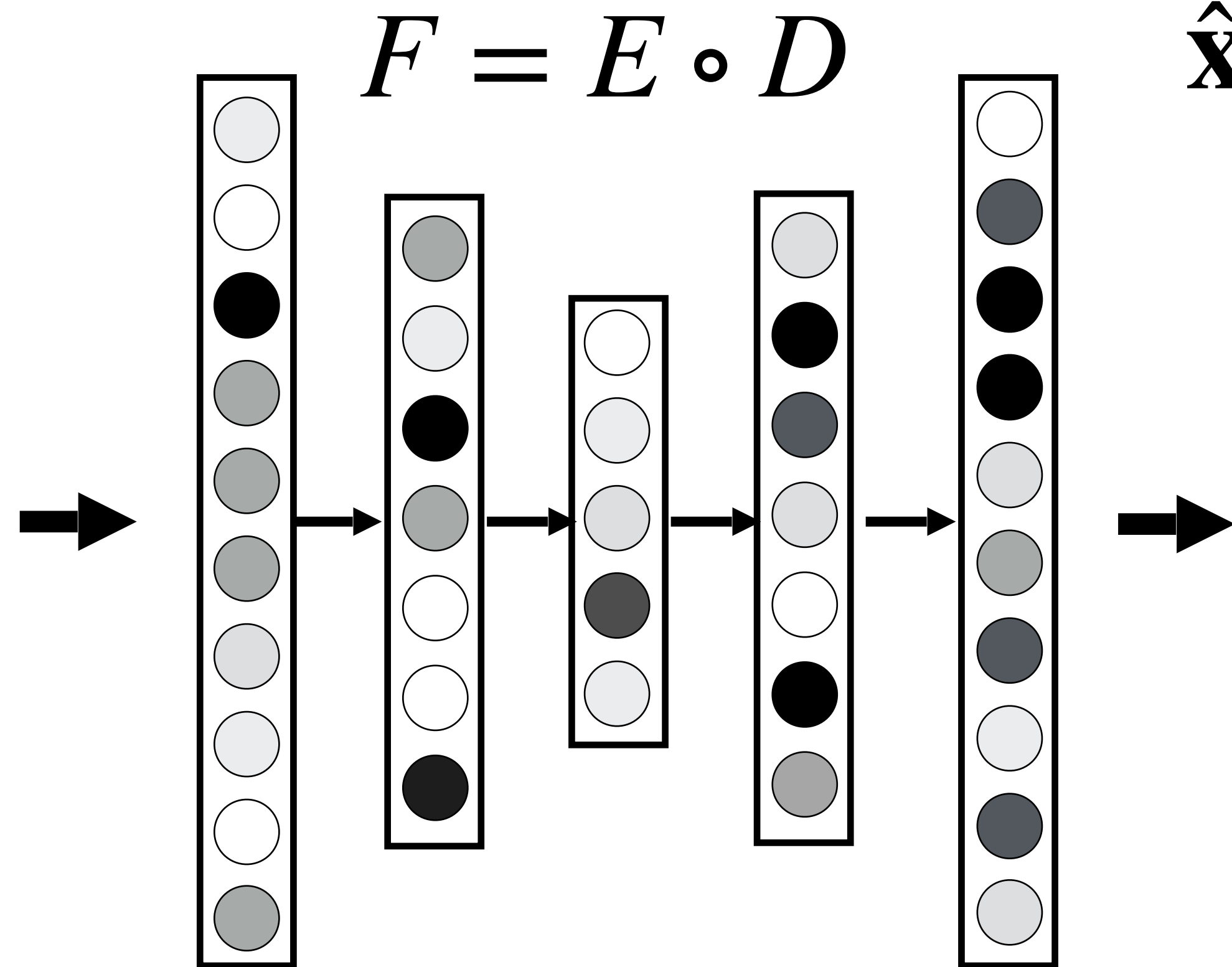
Adapted from Isola, Freeman, Torralba

Autoencoder

$\mathbf{x}$



Image



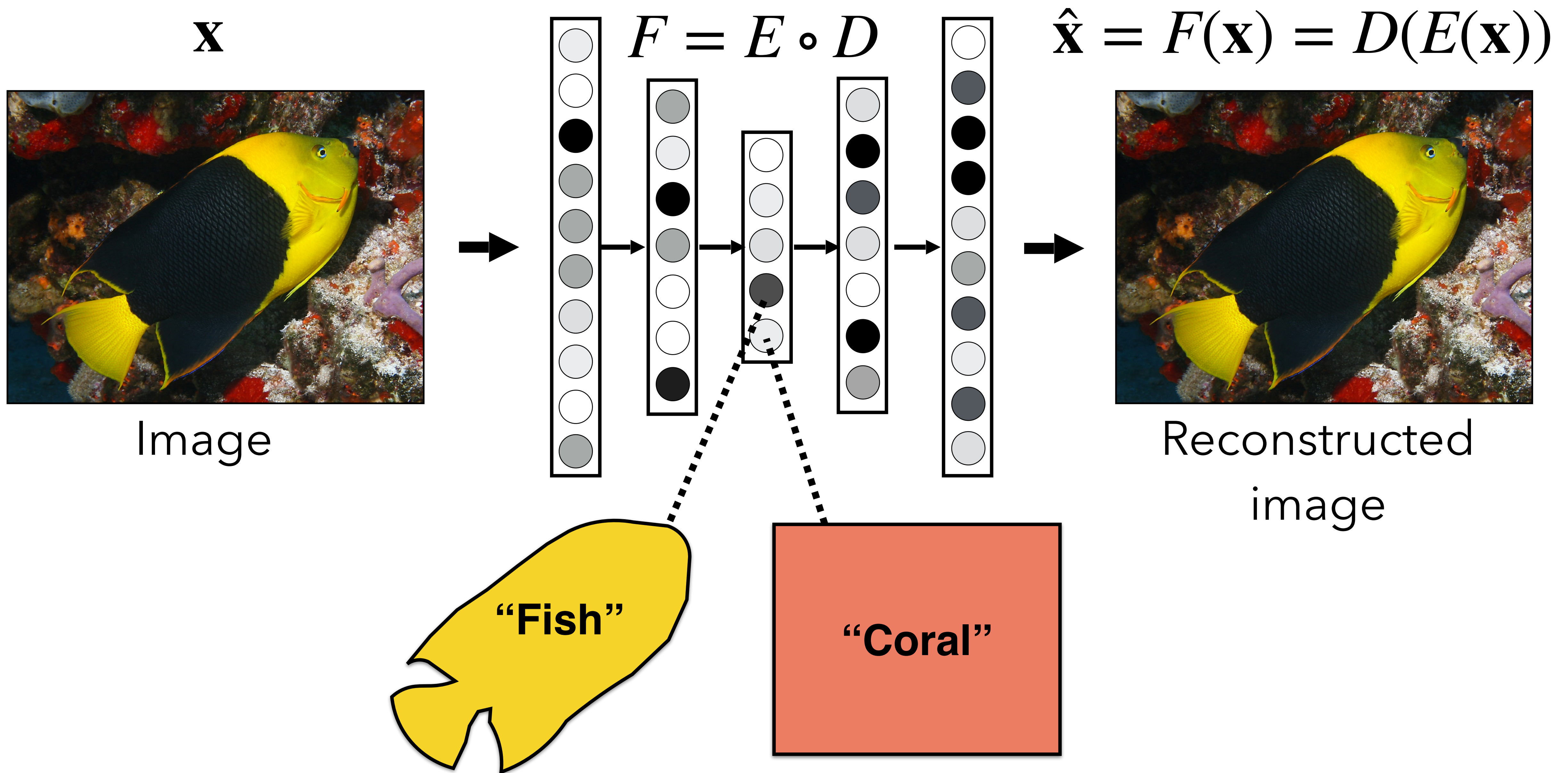
$$\hat{\mathbf{x}} = F(\mathbf{x}) = D(E(\mathbf{x}))$$



Reconstructed
image

Loss:

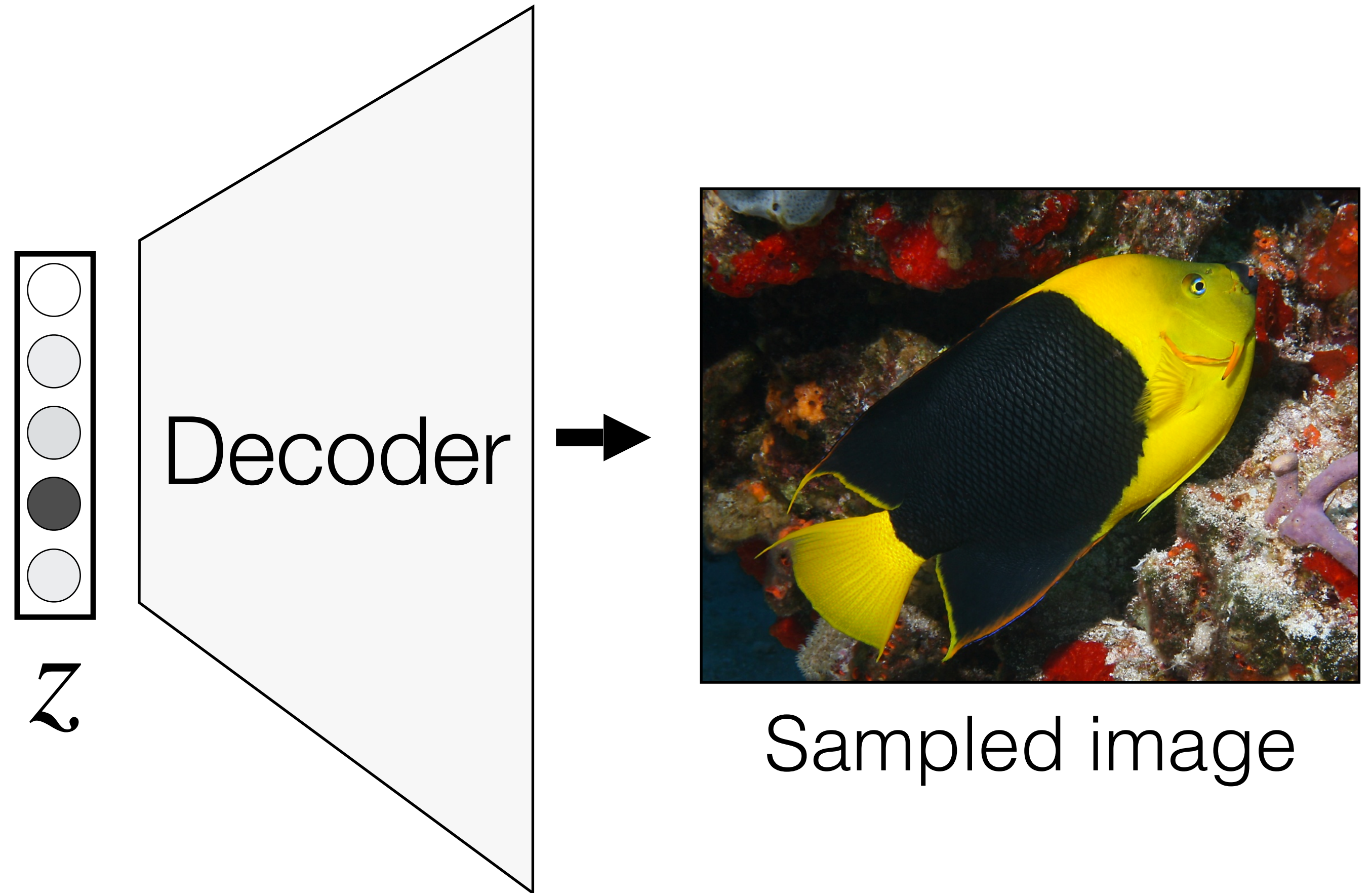
$$\mathcal{L}_{\theta} = \|\mathbf{x} - \hat{\mathbf{x}}\|^2$$



Also hope that it learns a useful code that captures important things through compression.

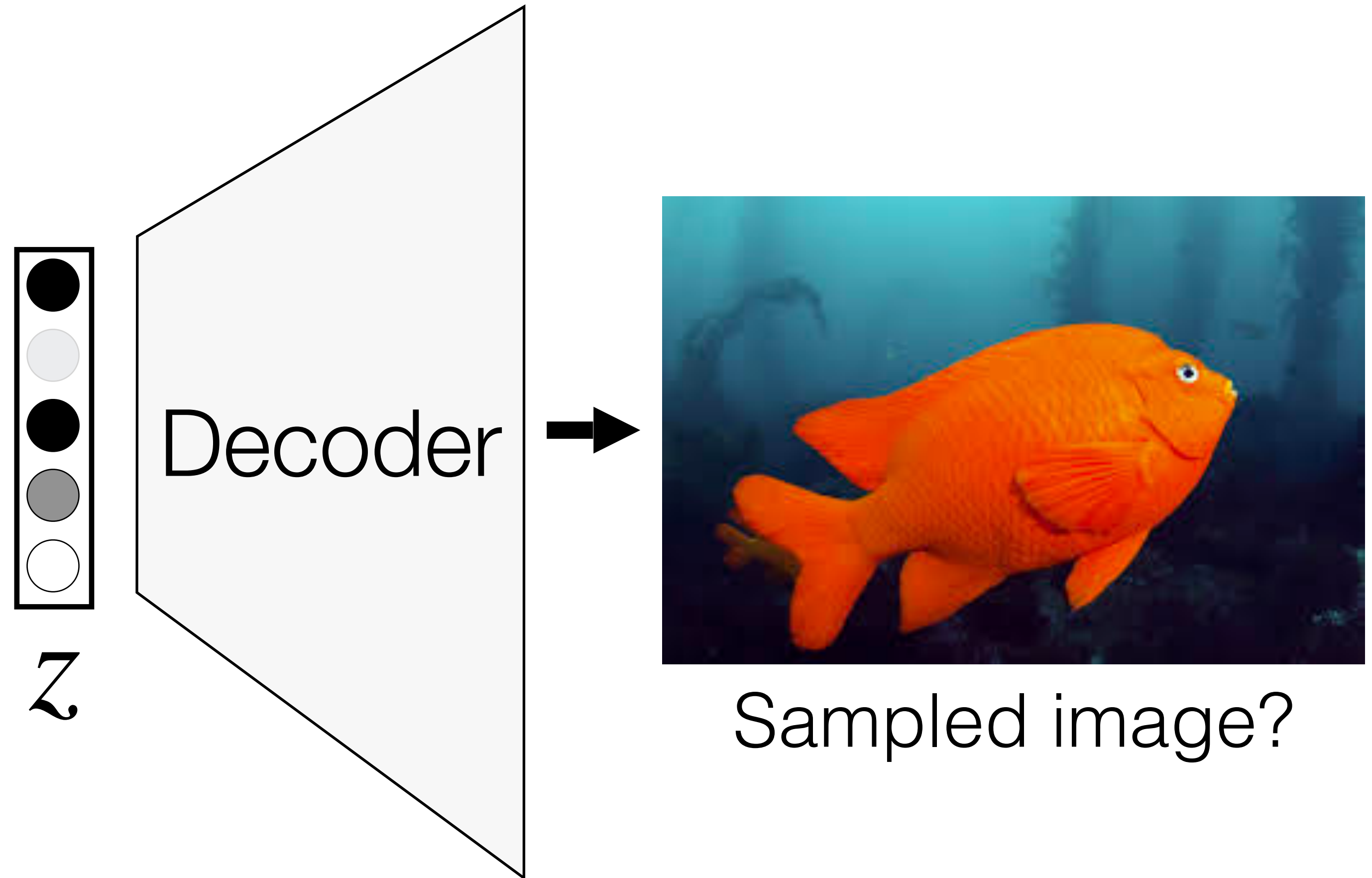
Autoencoders as generative models?

What if we just plug in
random z vectors?



Autoencoders as generative models?

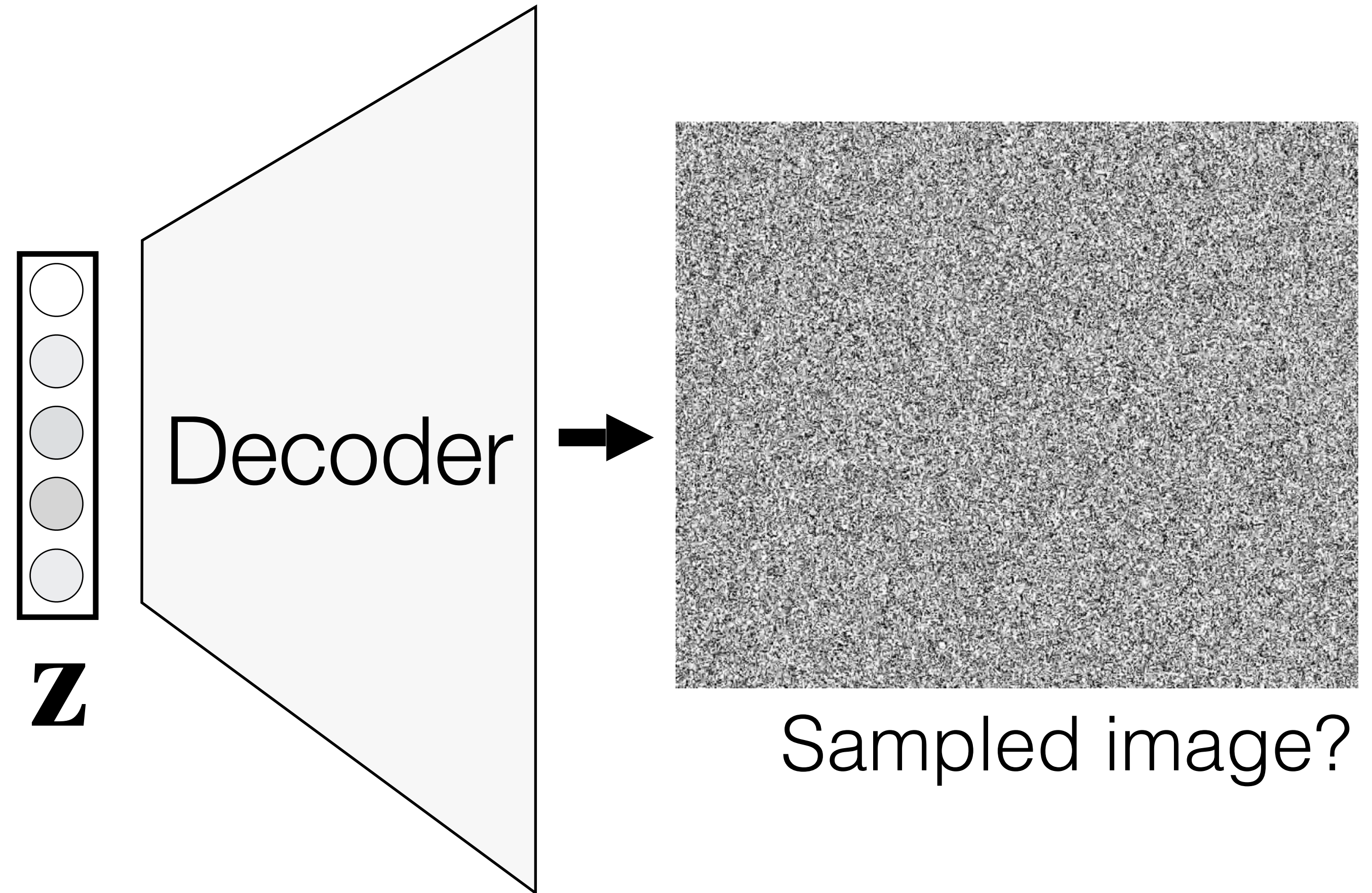
What if we just plug in
random z vectors?



Autoencoders as generative models?

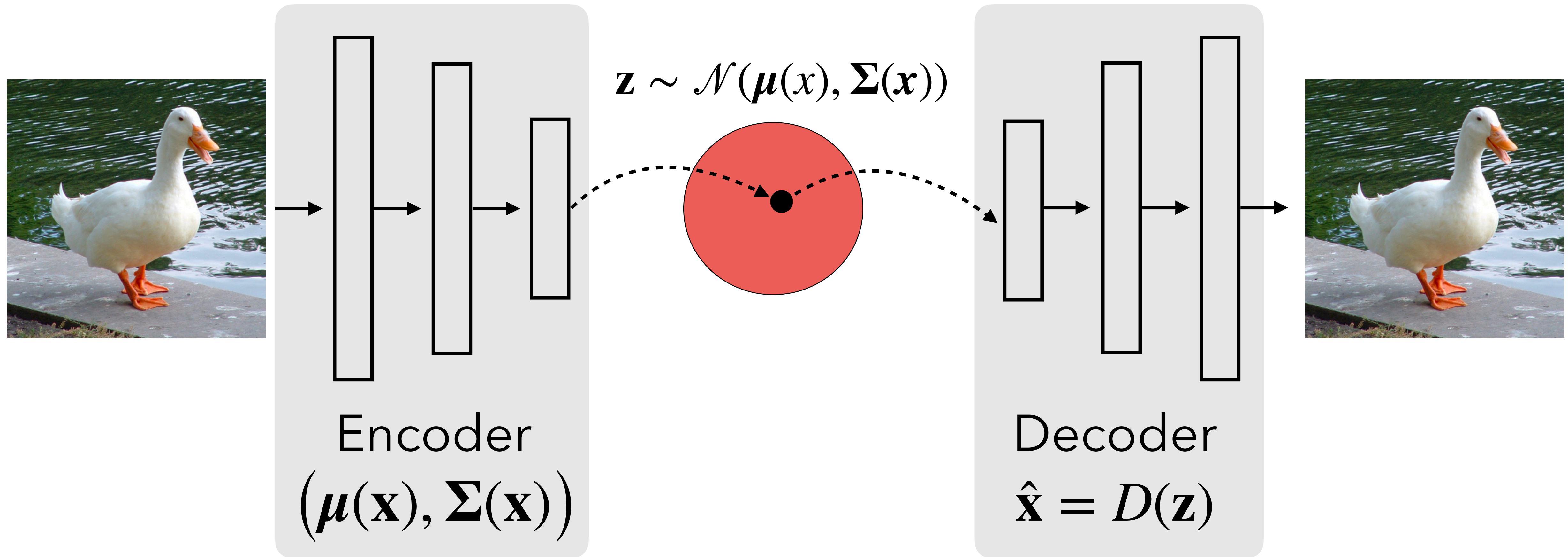
No guarantee that the latent space $\mathbf{z}$:

- Covers the space of examples.
- Follows a particular source distribution (e.g., Gaussian).



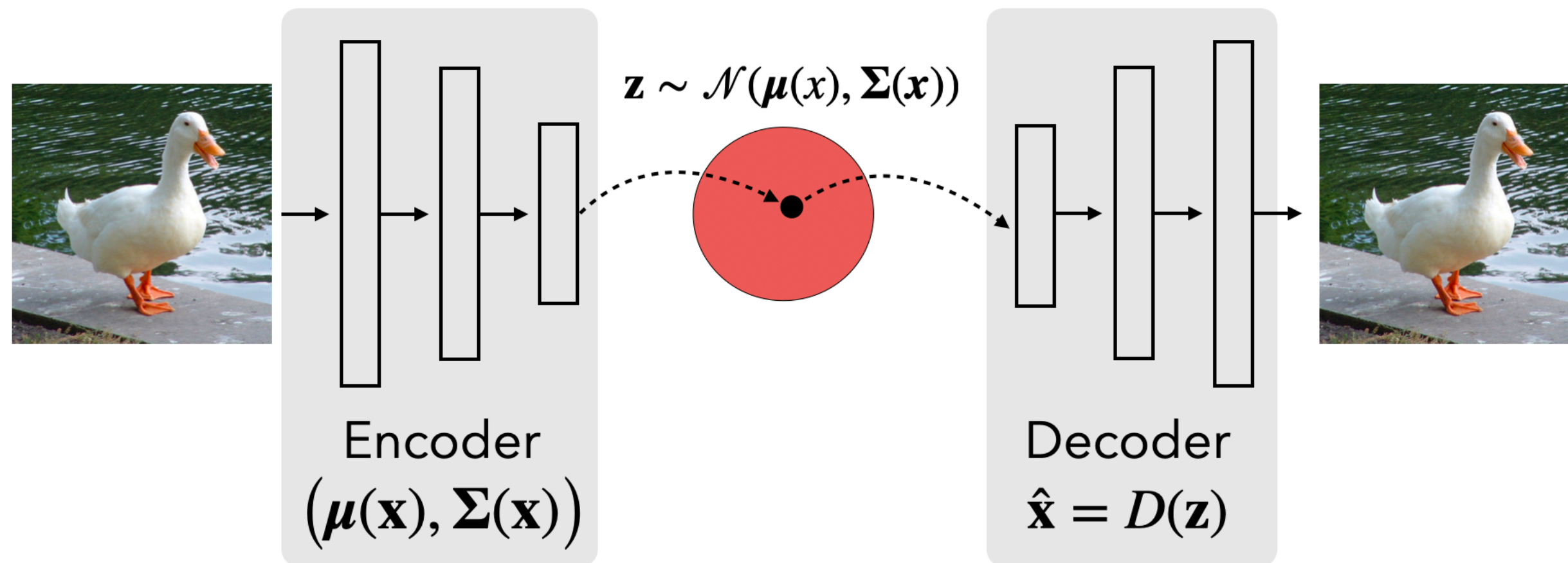
What if we force $\mathbf{z}$ to be from the source distribution?

Variational autoencoder



- Encoder net defines a distribution over $\mathbf{z}$: $q(\mathbf{z} \mid \mathbf{x}) = \mathcal{N}(\boldsymbol{\mu}(\mathbf{x}), \boldsymbol{\Sigma}(\mathbf{x}))$.
- Decoder net defines a distribution over $\mathbf{x}$: $p(\mathbf{x} \mid \mathbf{z}) = \mathcal{N}(\hat{\mathbf{X}}, I)$.

Variational autoencoder



Reconstruction loss:

$$\mathcal{L}_R = \|\mathbf{x} - \hat{\mathbf{x}}\|^2$$

Make $\mathbf{z}$ match source distribution

$$\mathcal{L}_P = D_{KL} (q(\mathbf{z} | \mathbf{x}) \parallel \mathcal{N}(0, I))$$

Why does this work?

Recall: maximum likelihood

$$\operatorname{argmax}_{\theta} \sum_{i=1}^N \log(p_{\theta}(\mathbf{x}_i))$$

$$= \operatorname{argmax}_{\theta} \sum_{i=1}^N \log \left(\int_{\mathbf{z}} p_{\theta}(\mathbf{x}_i, \mathbf{z}) d\mathbf{z} \right)$$

(we'll simplify notation a bit)

Evidence lower bound

$$\log p(\mathbf{x}) = \log \int p(\mathbf{x}, \mathbf{z}) d\mathbf{z}$$

the encoder, i.e., latent given image

$$= \log \int q(\mathbf{z} | \mathbf{x}) \frac{p(\mathbf{x}, \mathbf{z})}{q(\mathbf{z} | \mathbf{x})} d\mathbf{z}$$

Evidence lower bound

$$\begin{aligned}\log p(\mathbf{x}) &= \log \int p(\mathbf{x}, \mathbf{z}) d\mathbf{z} \\ &= \log \int q(\mathbf{z} \mid \mathbf{x}) \frac{p(\mathbf{x}, \mathbf{z})}{q(\mathbf{z} \mid \mathbf{x})} d\mathbf{z} \\ &\geq \int q(\mathbf{z} \mid \mathbf{x}) \log \frac{p(\mathbf{x}, \mathbf{z})}{q(\mathbf{z} \mid \mathbf{x})} d\mathbf{z}\end{aligned}$$

Jensen's inequality:

Since log is concave,

$$\log(\alpha x + (1 - \alpha)y) \geq \alpha \log(x) + (1 - \alpha)\log(y)$$

for $\alpha \in [0, 1]$.

Evidence lower bound

$$\log p(\mathbf{x}) = \log \int p(\mathbf{x}, \mathbf{z}) d\mathbf{z}$$

$$= \log \int q(\mathbf{z} | \mathbf{x}) \frac{p(\mathbf{x}, \mathbf{z})}{q(\mathbf{z} | \mathbf{x})} d\mathbf{z}$$

$$\geq \int q(\mathbf{z} | \mathbf{x}) \log \frac{p(\mathbf{x}, \mathbf{z})}{q(\mathbf{z} | \mathbf{x})} d\mathbf{z}$$

the decoder source distribution, e.g., $\mathcal{N}(\mathbf{0}, I)$

$$= \int q(\mathbf{z} | \mathbf{x}) \log \frac{p(\mathbf{x} | \mathbf{z}) p(\mathbf{z})}{q(\mathbf{z} | \mathbf{x})} d\mathbf{z}$$

Evidence lower bound

$$\begin{aligned}\log p(\mathbf{x}) &= \log \int p(\mathbf{x}, \mathbf{z}) d\mathbf{z} \\&= \log \int q(\mathbf{z} \mid \mathbf{x}) \frac{p(\mathbf{x}, \mathbf{z})}{q(\mathbf{z} \mid \mathbf{x})} d\mathbf{z} \\&\geq \int q(\mathbf{z} \mid \mathbf{x}) \log \frac{p(\mathbf{x}, \mathbf{z})}{q(\mathbf{z} \mid \mathbf{x})} d\mathbf{z} \\&= \int q(\mathbf{z} \mid \mathbf{x}) \log \frac{p(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z})}{q(\mathbf{z} \mid \mathbf{x})} d\mathbf{z} \\&= \int q(\mathbf{z} \mid \mathbf{x}) \log p(\mathbf{x} \mid \mathbf{z}) d\mathbf{z} - \int q(\mathbf{z} \mid \mathbf{x}) \log \frac{q(\mathbf{z} \mid \mathbf{x})}{p(\mathbf{z})} d\mathbf{z}\end{aligned}$$

Evidence lower bound

$$\begin{aligned}\log p(\mathbf{x}) &= \log \int p(\mathbf{x}, \mathbf{z}) d\mathbf{z} \\ &\geq \int q(\mathbf{z} | \mathbf{x}) \log p(\mathbf{x} | \mathbf{z}) d\mathbf{z} - \int q(\mathbf{z} | \mathbf{x}) \log \frac{q(\mathbf{z} | \mathbf{x})}{p(\mathbf{z})} d\mathbf{z} \\ &= \mathbb{E}_{q(\mathbf{z}|\mathbf{x})} [\log p(\mathbf{x} | \mathbf{z})] - D_{\text{KL}}(q(\mathbf{z} | \mathbf{x}) || p(\mathbf{z}))\end{aligned}$$

This is called the **evidence lower bound (ELBO)** or the **variational lower bound**.

The “variational” term comes from the fact that bounds like these were optimized using variational calculus in other lines of work (whereas here, we’ll use gradient descent).

Evidence lower bound

$$\begin{aligned}\log p(\mathbf{x}) &= \log \int p(\mathbf{x}, \mathbf{z}) d\mathbf{z} \\ &\geq \int q(\mathbf{z} | \mathbf{x}) \log p(\mathbf{x} | \mathbf{z}) d\mathbf{z} - \int q(\mathbf{z} | \mathbf{x}) \log \frac{q(\mathbf{z} | \mathbf{x})}{p(\mathbf{z})} d\mathbf{z} \\ &= \mathbb{E}_{q(\mathbf{z}|\mathbf{x})} [\log p(\mathbf{x} | \mathbf{z})] - D_{\text{KL}}(q(\mathbf{z} | \mathbf{x}) \parallel p(\mathbf{z}))\end{aligned}$$

Reconstruction loss:

$$\mathcal{L}_R = \|\mathbf{x} - \hat{\mathbf{x}}\|^2 \quad (\text{up to scale factor})$$

Equivalent to log likelihood of $\mathcal{N}(\hat{\mathbf{x}}, I)$: i.e.,
how likely is $\mathbf{x}$ under a Gaussian with mean $\hat{\mathbf{x}}$?

KL divergence loss:

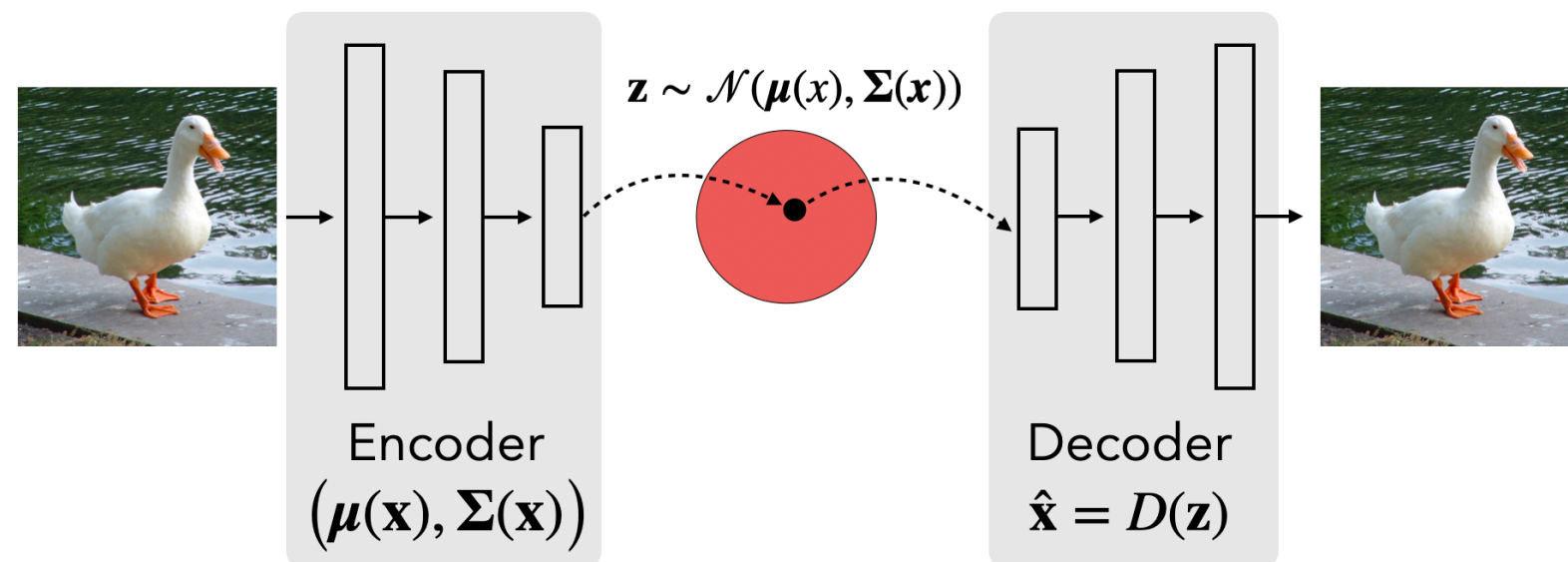
$$\mathcal{L}_P = D_{KL} (q(\mathbf{z} | \mathbf{x}) \parallel \mathcal{N}(0, I))$$

Make $\mathbf{z}$ to match our desired source distribution.

(when we negate this and minimize, rather than maximize)

A closer look

$$\begin{aligned}\log p(\mathbf{x}) &= \log \int p(\mathbf{x}, \mathbf{z}) d\mathbf{z} \\ &\geq \mathbb{E}_{q(\mathbf{z}|\mathbf{x})} [\log p(\mathbf{x} | \mathbf{z})] - D_{\text{KL}}(q(\mathbf{z} | \mathbf{x}) \parallel p(\mathbf{z}))\end{aligned}$$

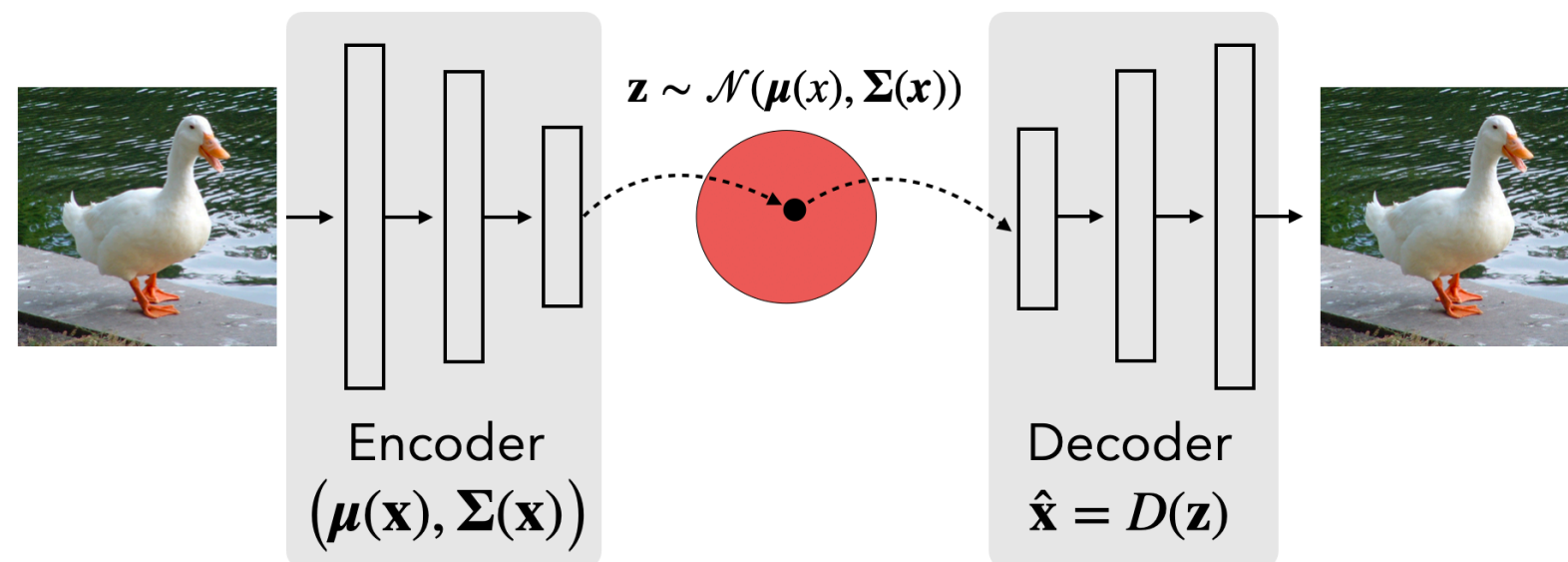


This term describes an *autoencoder*! It says:

1. Choose a random example $\mathbf{x}$ from the dataset.
2. Sample $\mathbf{z}$ given $\mathbf{x}$ using the encoder, $q(\mathbf{z} | \mathbf{x})$
3. Decode $\hat{\mathbf{x}}$ and compare it to $\mathbf{x}$, which corresponds to computing $\log p(\mathbf{x} | \mathbf{z})$

A closer look

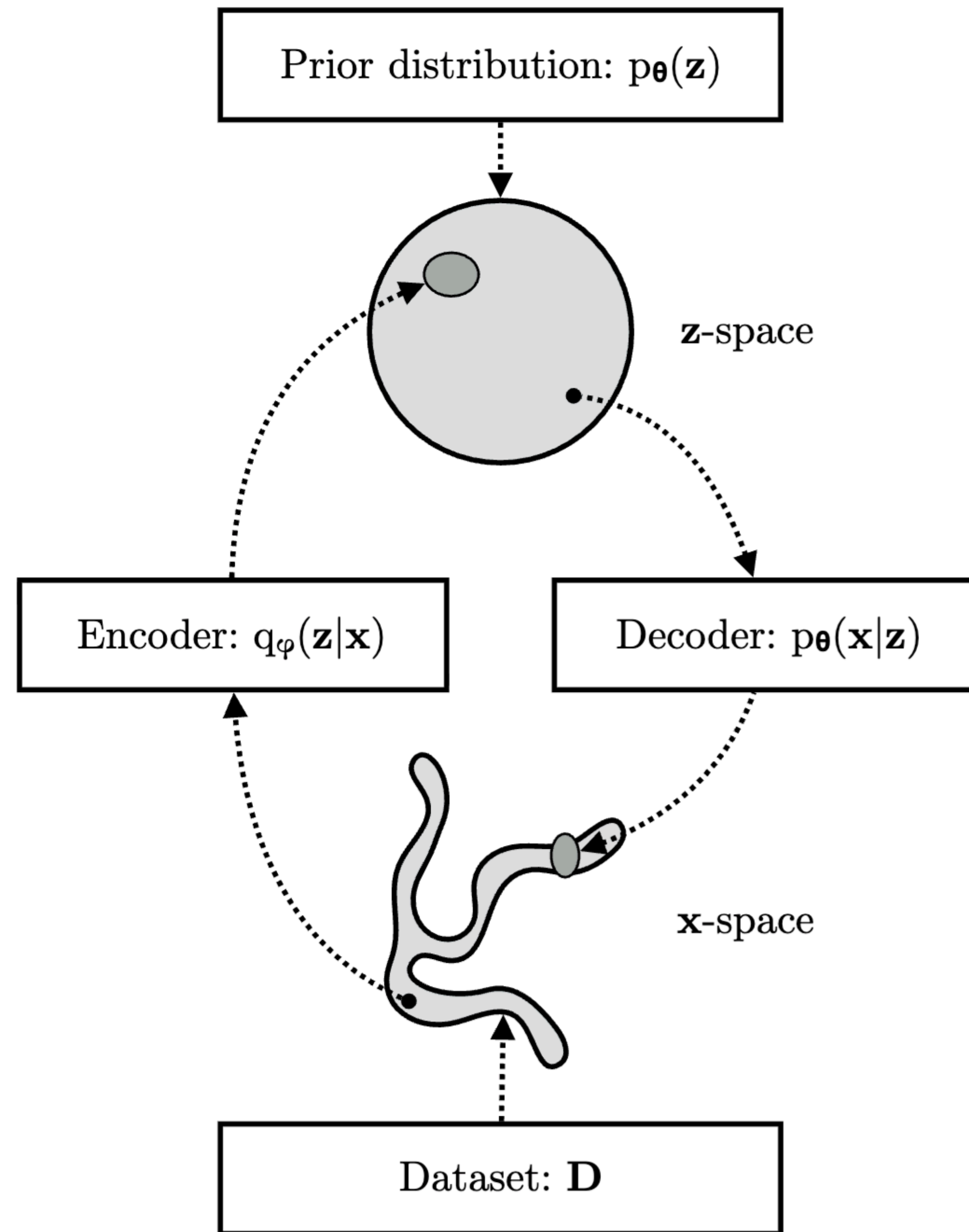
$$\begin{aligned}\log p(\mathbf{x}) &= \log \int p(\mathbf{x}, \mathbf{z}) d\mathbf{z} \\ &\geq \mathbb{E}_{q(\mathbf{z}|\mathbf{x})} [\log p(\mathbf{x} | \mathbf{z})] - D_{\text{KL}}(q(\mathbf{z} | \mathbf{x}) || p(\mathbf{z}))\end{aligned}$$



KL divergence. Assuming that $p(\mathbf{z})$ is Gaussian:

1. How far is the latent space that our encoder produces from a Gaussian?
2. Analytical solution if $p(\mathbf{z})$ is Gaussian (very similar to PS1!).

Recap:



Pseudocode

```
1 for x in loader: # x: minibatch of images
2     # Encode/decode the image
3     (mu, sigma) = encoder.forward(x) # params for  $q(\mathbf{z} | \mathbf{x})$ 
4     z = sample_gaussian(mu, sigma)   # sample from  $q(\mathbf{z} | \mathbf{x})$ 
5     x_recons = decoder.forward(z)     # reconstruct input image
6
7     # Compute losses
8     L_recons = mean((x - x_recons)**2.) # compute  $\log p(\mathbf{x} | \mathbf{z})$ 
9     L_kl = KL_divergence_loss(mu, sigma) # compute  $D_{KL}(\mathbf{z} || \mathcal{N}(0, I))$ 
10    loss = L_recons + L_kl
11
12    # Perform an SGD update
13    loss.backward()
```

Pseudocode

How do we backprop through this?

```
1 for x in loader: # x: minibatch of images
2     # Encode/decode the image
3     (mu, sigma) = encoder.forward(x) # params for  $q(\mathbf{z} | \mathbf{x})$ 
4     z = sample_gaussian(mu, sigma) # sample from  $q(\mathbf{z} | \mathbf{x})$ 
5     x_recons = decoder.forward(z) # reconstruct input image
6
7     # Compute losses
8     L_recons = mean((x - x_recons)**2.) # compute  $\log p(\mathbf{x} | \mathbf{z})$ 
9     L_kl = KL_divergence_loss(mu, sigma) # compute  $D_{KL}(\mathbf{z} || \mathcal{N}(0, I))$ 
10    loss = L_recons + L_kl
11
12    # Perform an SGD update
13    loss.backward()
```

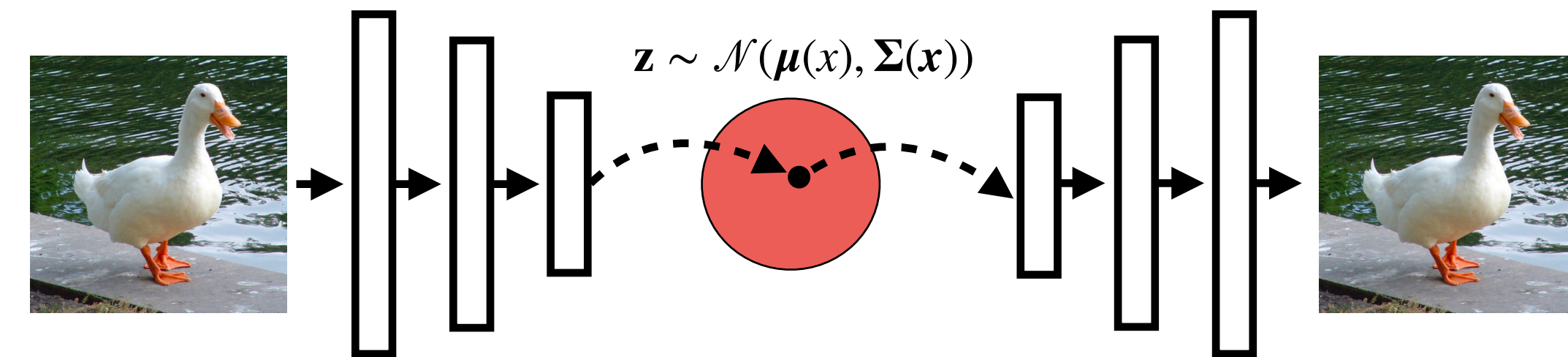

Reparameterization trick

- Sampling process can be written as:

$$\mathbf{z} = g(\boldsymbol{\epsilon}, \mathbf{x})$$

if we are given a sample $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, I)$.

- For example: $z_i = \sigma_i \epsilon_i + \mu_i$
for $(\boldsymbol{\sigma}, \boldsymbol{\mu})$ output by the network,
assuming diagonal covariance for
network $\boldsymbol{\Sigma}(\mathbf{x}) = \text{diag}(\sigma_1, \sigma_2, \dots, \sigma_d)$.
- We can sample $\boldsymbol{\epsilon}$ in each SGD batch
and use it to compute $g(\boldsymbol{\epsilon}, \mathbf{x})$.
- Easy to backprop through this.

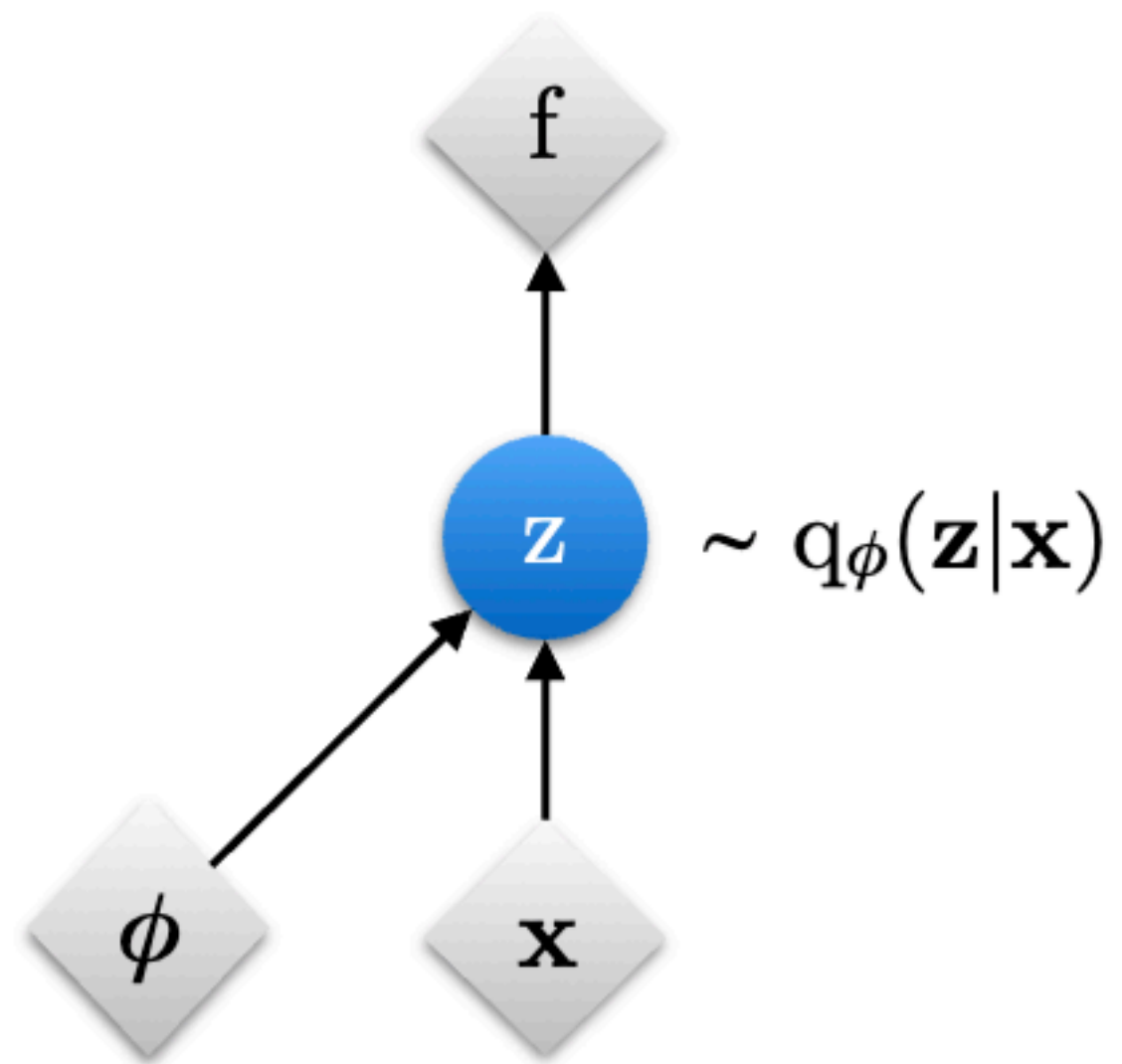


- Can use other estimators (like REINFORCE, which we will discuss later), but they often have higher variance.

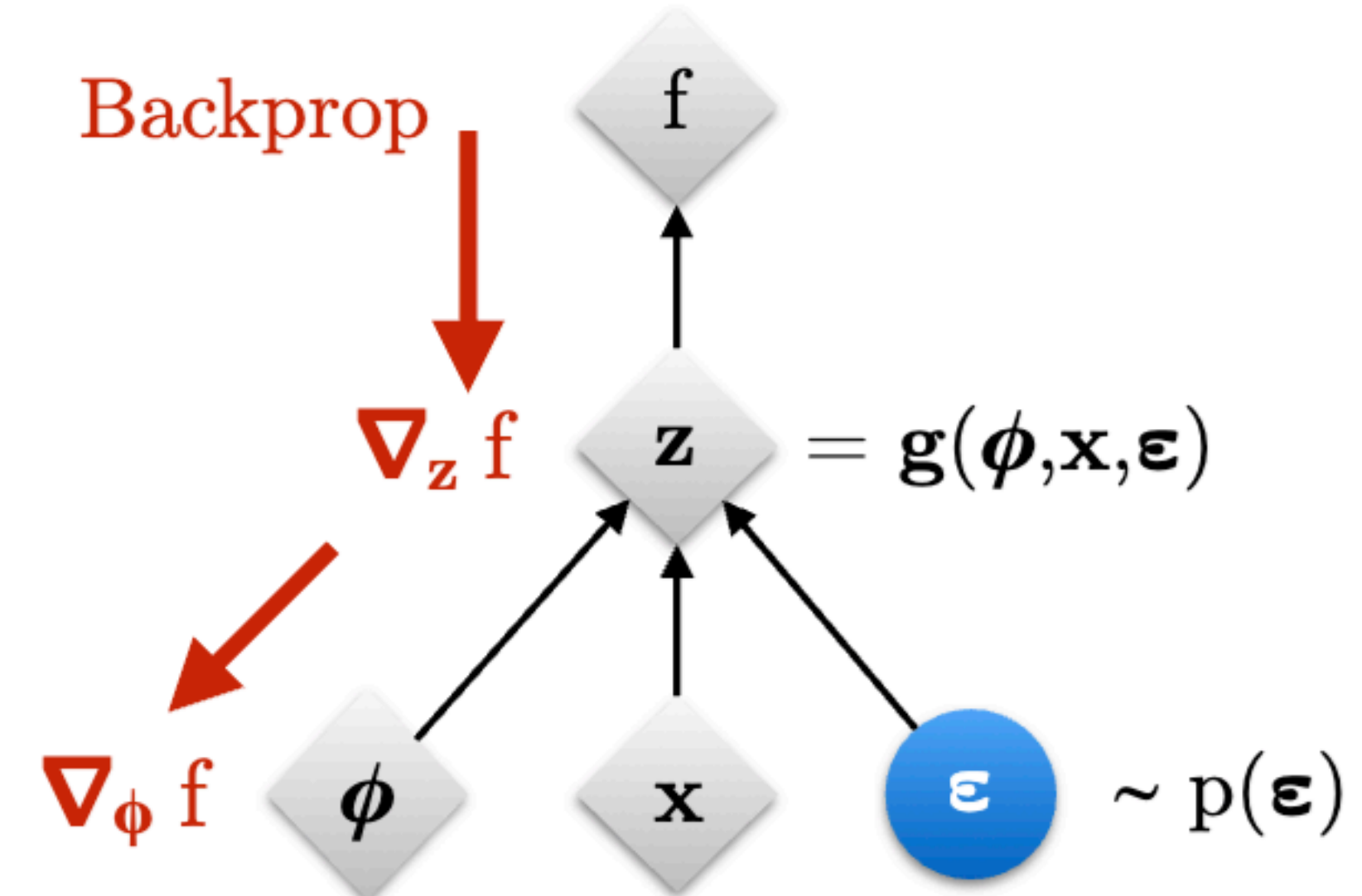
Reparameterization trick

```
1 for x in loader: # x: minibatch of images
2     # Encode/decode the image
3     (mu, sigma) = encoder.forward(x) # params for  $q(\mathbf{z} | \mathbf{x})$ 
4
5     # sample from  $q(\mathbf{z} | \mathbf{x})$ 
6     epsilon = random_normal(len(mu))
7     z = mu + sigma * epsilon
8
9     x_recons = decoder.forward(z) # reconstruct input image
10
11     # Compute losses
12     L_recons = mean((x - x_recons)**2.) # compute  $\log p(\mathbf{x} | \mathbf{z})$ 
13     L_kl = KL_divergence_loss(mu, sigma) # compute  $D_{KL}(\mathbf{z} || \mathcal{N}(0, I))$ 
14     loss = L_recons + L_kl
15
16     # Perform an SGD update
17     loss.backward()
```

Original form



Reparameterized form



: Deterministic node



: Random node



: Evaluation of f



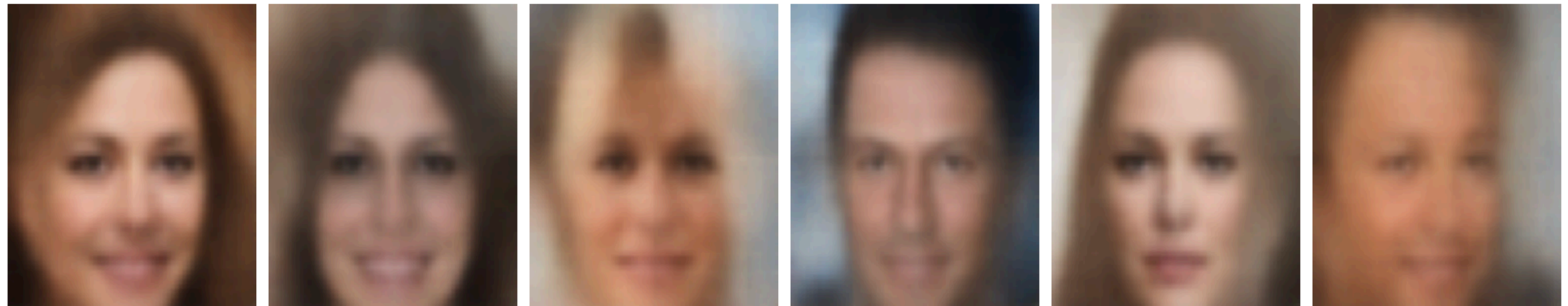
: Differentiation of f

Generating faces with a VAE

Autoencoder



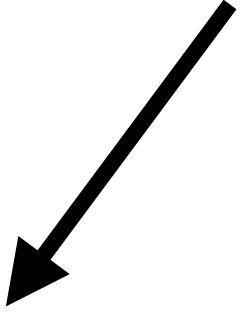
VAE



Why is it blurry? Many $\mathbf{x}$ for each $\mathbf{z}$.

Weighting the terms

measures importance of KL

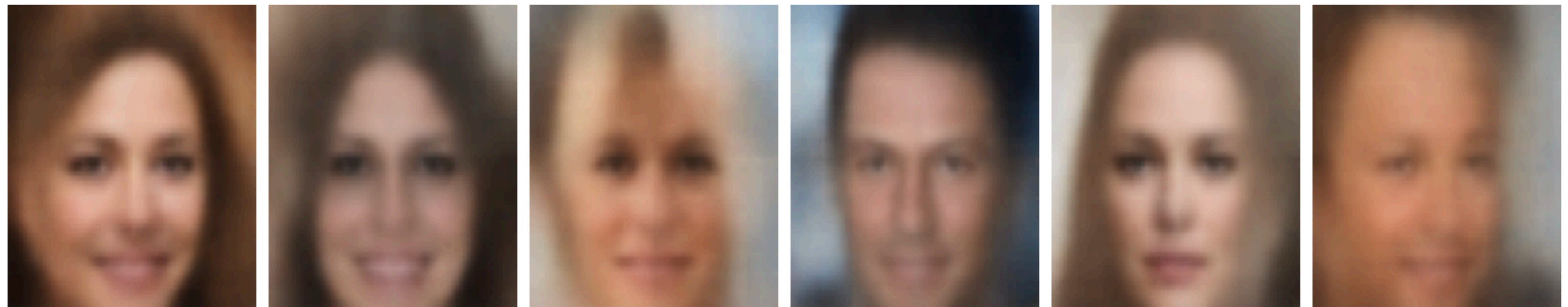

$$\mathcal{L}_\beta = -\mathbb{E}_{q(\mathbf{z}|\mathbf{x})} [\log p(\mathbf{x} | \mathbf{z})] + \beta D_{\text{KL}}(q(\mathbf{z} | \mathbf{x}) || p(\mathbf{z}))$$

What happens when we change β ?

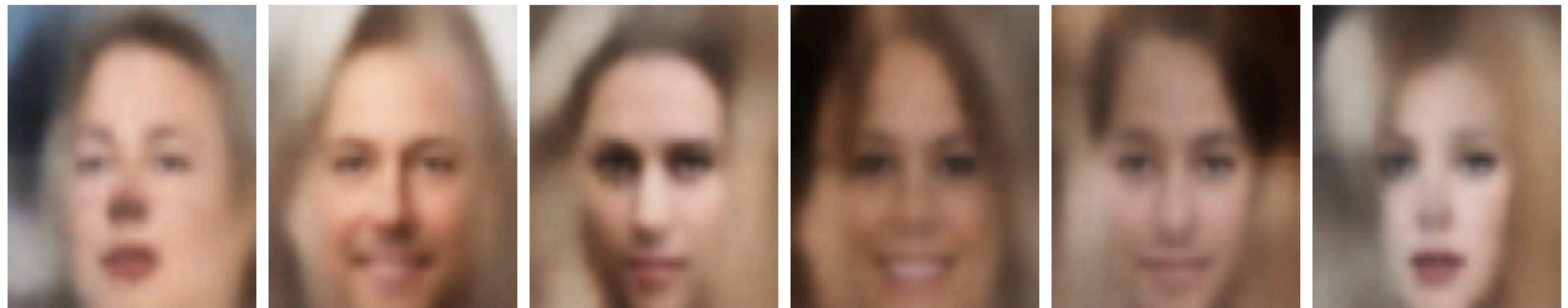
Autoencoder



VAE
 $\beta = 1$



VAE
 $\beta = 0.5$



Exploring the latent space

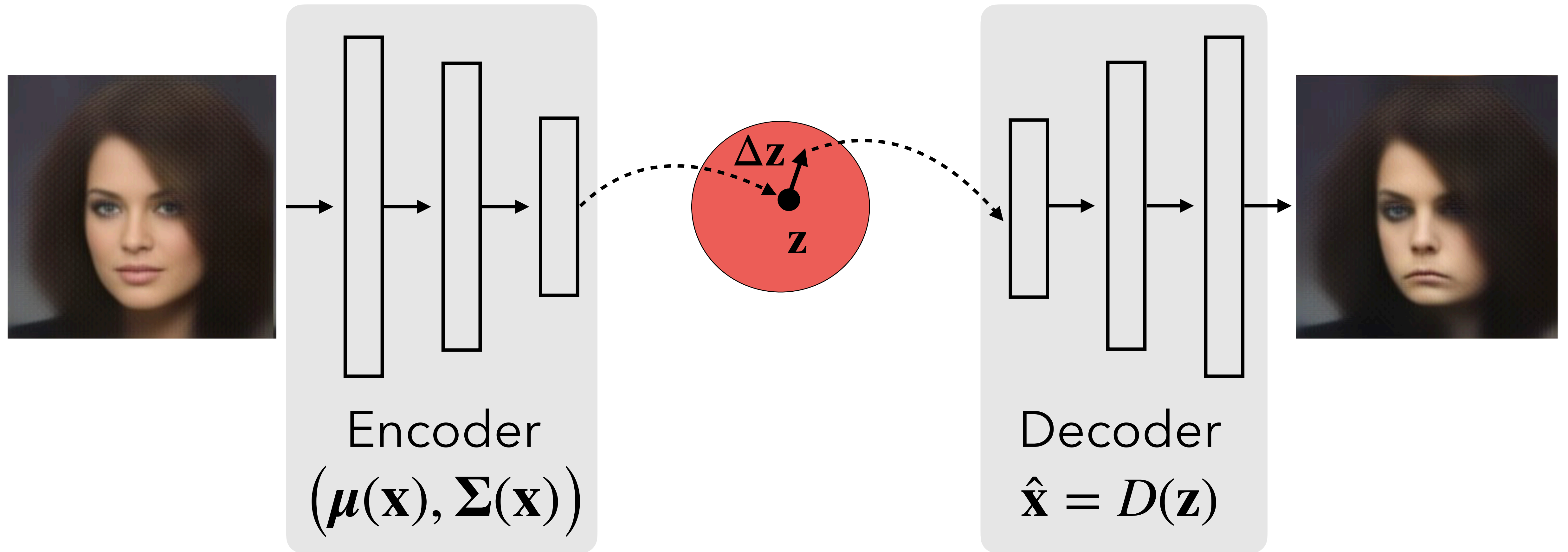
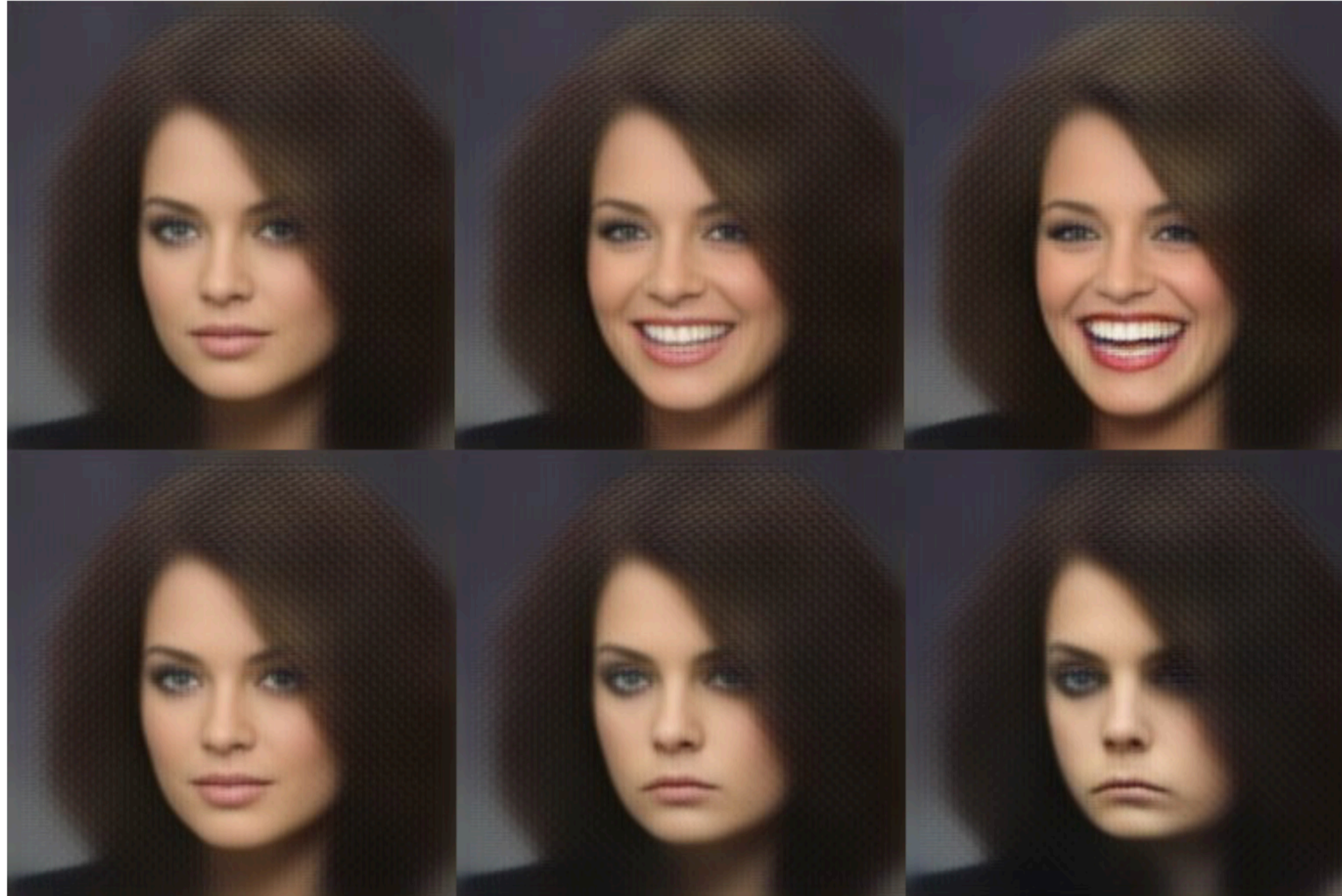


Image resynthesis

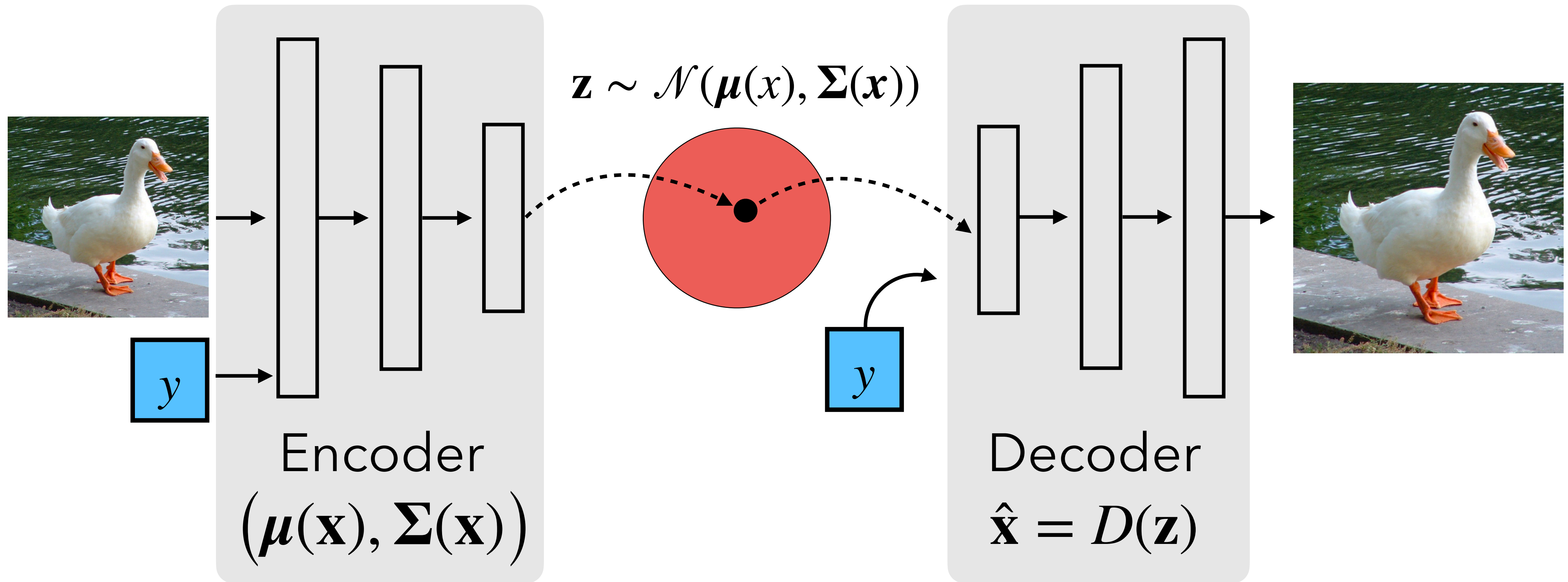


What does the latent code represent?



Manipulating one component of $\mathbf{z}$.

Class-conditional VAE



Sampling with fixed class y

2 2 2 2 2 2 2 2 2 2	3 3 3 3 3 3 3 3 3 3	4 4 4 4 4 4 4 4 4 4
2 2 2 2 2 2 2 2 2 2	3 3 3 3 3 3 3 3 3 3	4 4 4 4 4 4 4 4 4 4
2 2 2 2 2 2 2 2 2 2	3 3 3 3 3 3 3 3 3 3	4 4 4 4 4 4 4 4 4 4
2 2 2 2 2 2 2 2 2 2	3 3 3 3 3 3 3 3 3 3	4 4 4 4 4 4 4 4 4 4
2 2 2 2 2 2 2 2 2 2	3 3 3 3 3 3 3 3 3 3	4 4 4 4 4 4 4 4 4 4
2 2 2 2 2 2 2 2 2 2	3 3 3 3 3 3 3 3 3 3	4 4 4 4 4 4 4 4 4 4
2 2 2 2 2 2 2 2 2 2	3 3 3 3 3 3 3 3 3 3	4 4 4 4 4 4 4 4 4 4
2 2 2 2 2 2 2 2 2 2	3 3 3 3 3 3 3 3 3 3	4 4 4 4 4 4 4 4 4 4
2 2 2 2 2 2 2 2 2 2	3 3 3 3 3 3 3 3 3 3	4 4 4 4 4 4 4 4 4 4
2 2 2 2 2 2 2 2 2 2	3 3 3 3 3 3 3 3 3 3	4 4 4 4 4 4 4 4 4 4

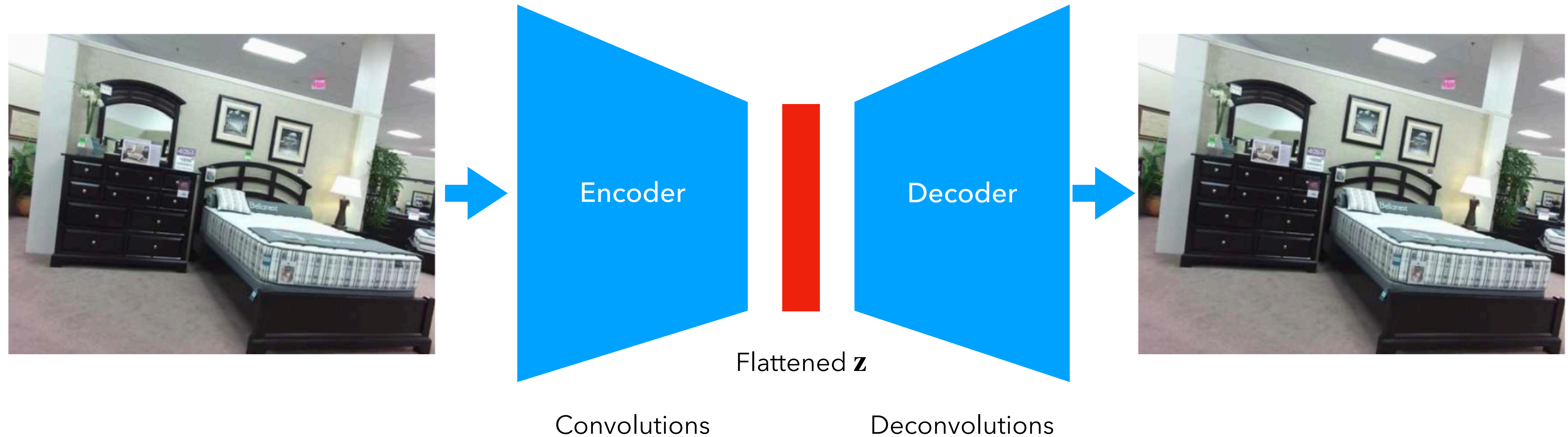
Sampling with fixed latent $\mathbf{z}$

4	0	1	2	3	4	5	6	7	8	9
9	0	1	2	3	4	5	6	7	8	9
5	0	1	2	3	4	5	6	7	8	9
4	0	1	2	3	4	5	6	7	8	9
2	0	1	2	3	4	5	6	7	8	9
7	0	1	2	3	4	5	6	7	8	9
5	0	1	2	3	4	5	6	7	8	9
1	0	1	2	3	4	5	6	7	8	9
7	0	1	2	3	4	5	6	7	8	9
1	0	1	2	3	4	5	6	7	8	9

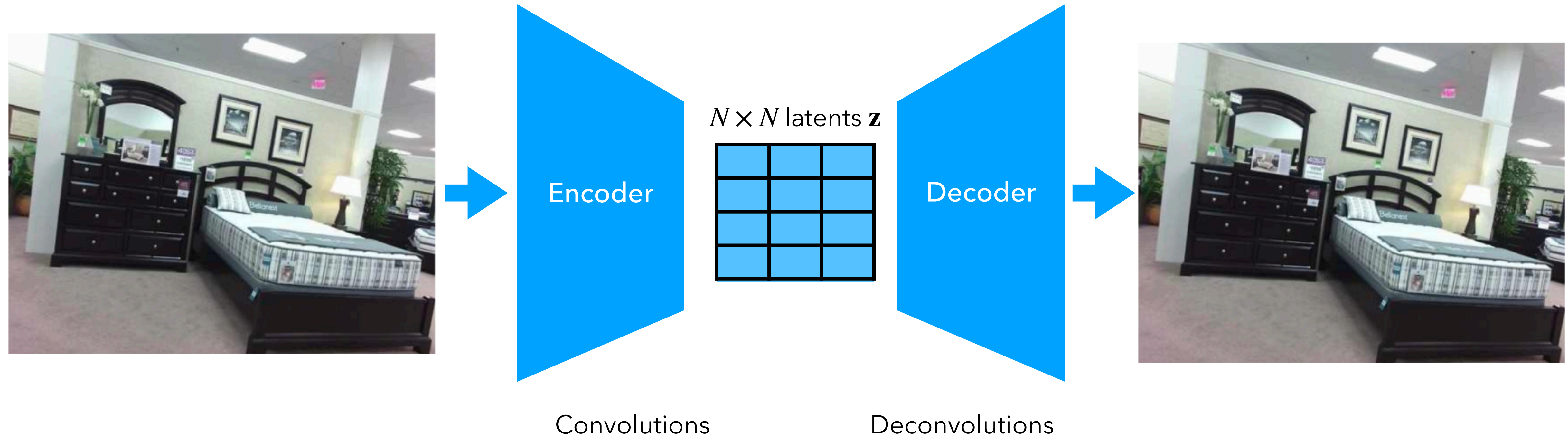


Architectures for VAEs

Recall: encoder-decoder models



Convolutional VAEs

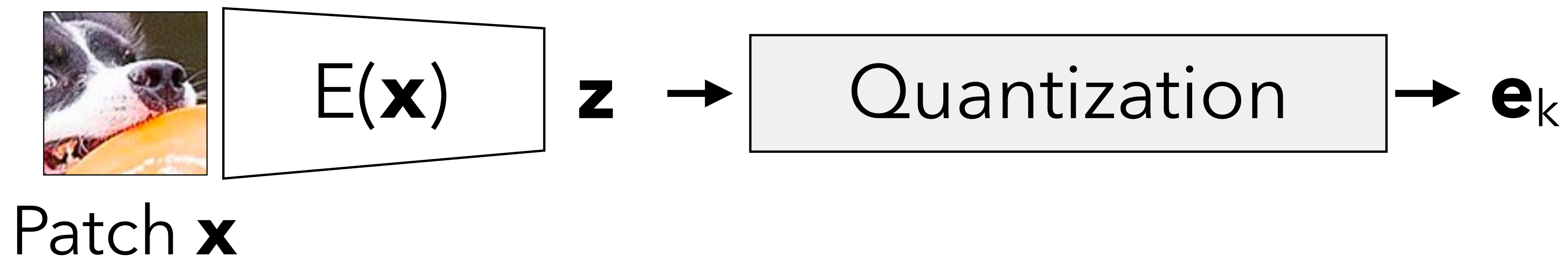
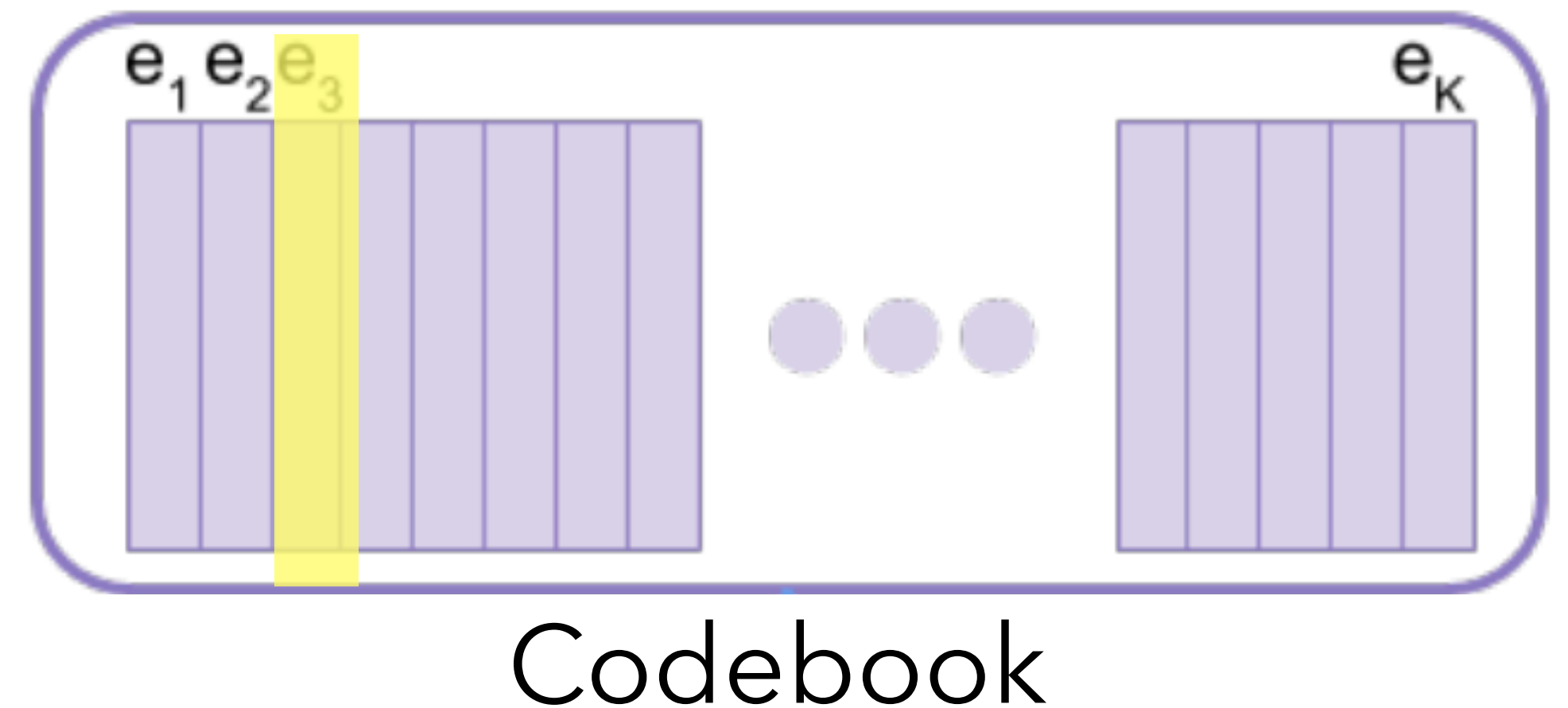


Vector-quantized VAE

- Learn a VAE with a discrete code.
- In practice, often an $N \times N$ grid
- This is a useful low-dimensional representation for *other* generative models.
- It is particularly useful for autoregressive models (which we'll discuss in more depth later), since they usually work best with discrete structures.

Vector quantization

Predict a real-valued vector, then “snap” it to a nearest neighbor from a *codebook*.



$$\text{Quantize}(E(\mathbf{x})) = \mathbf{e}_k \quad \text{where } k = \arg \min_j ||E(\mathbf{x}) - \mathbf{e}_j||$$

Vector-quantized variational autoencoder (VQ-VAE)

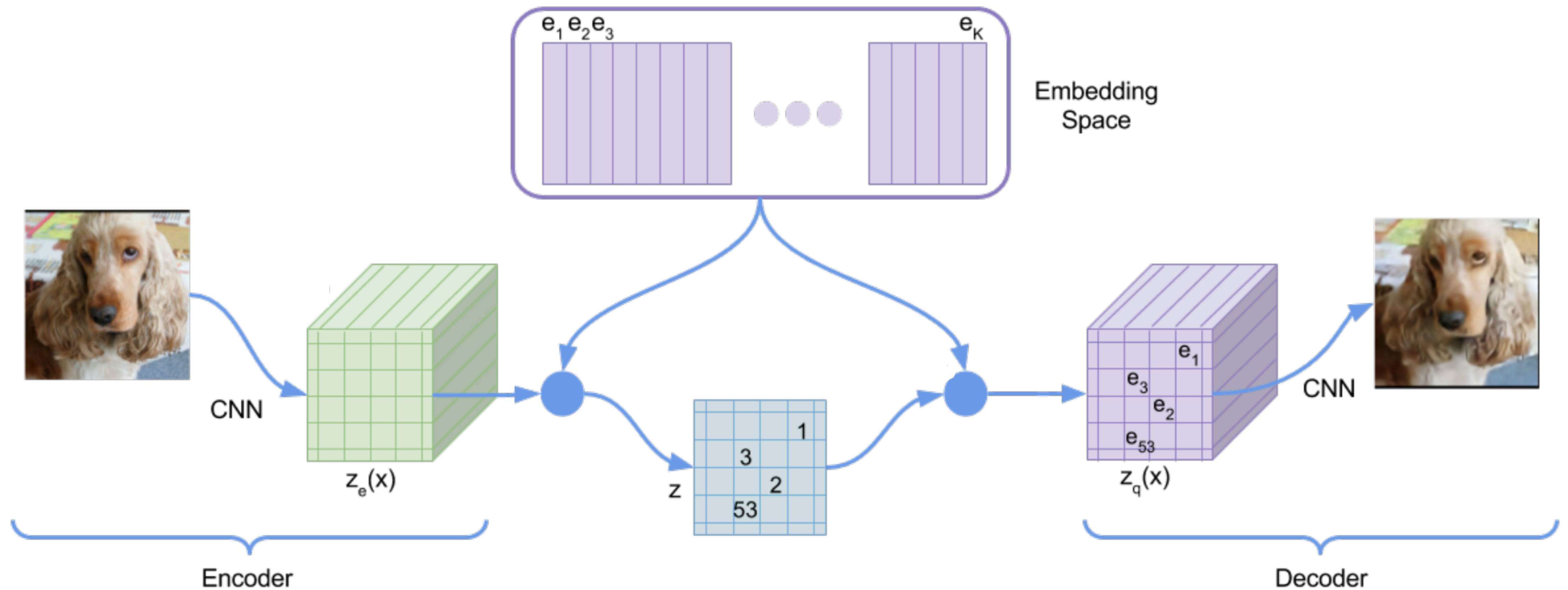


Figure source: [van den Oord et al., "VQ-VAE", 2017]

Learning the VQ-VAE

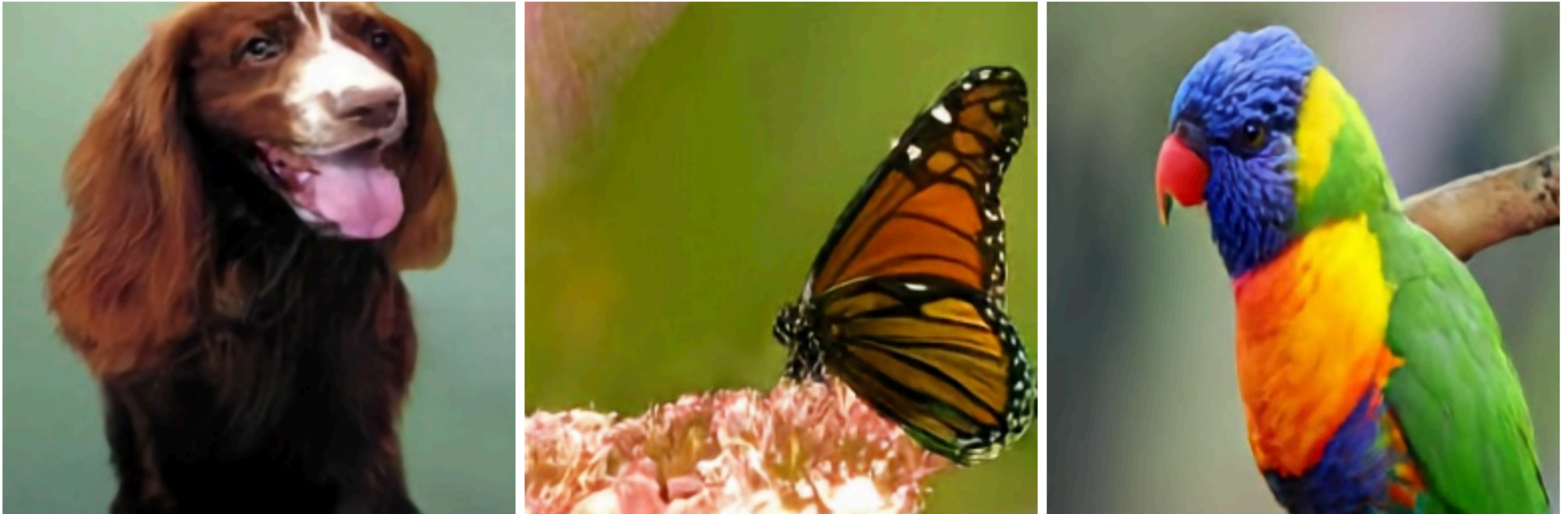
Minimize the following (heuristically motivated) loss:

$$\mathcal{L}(\mathbf{x}, D(\mathbf{e})) = ||\mathbf{x} - D(\mathbf{e})||_2^2 + ||sg[E(\mathbf{x})] - \mathbf{e}||_2^2 + \beta ||sg[\mathbf{e}] - E(\mathbf{x})||_2^2$$

where:

- $\mathbf{e}$ is the quantized code for $\mathbf{x}$
- $E(\mathbf{x})$ and $D(\mathbf{x})$ are the encoded and decoded images
- $sg[\mathbf{e}]$ is "stop gradient" a.k.a. "detach"

VQ-VAE + autoregressive model for image generation



Generated images (256×256)

Other architectures: hierarchical VAEs

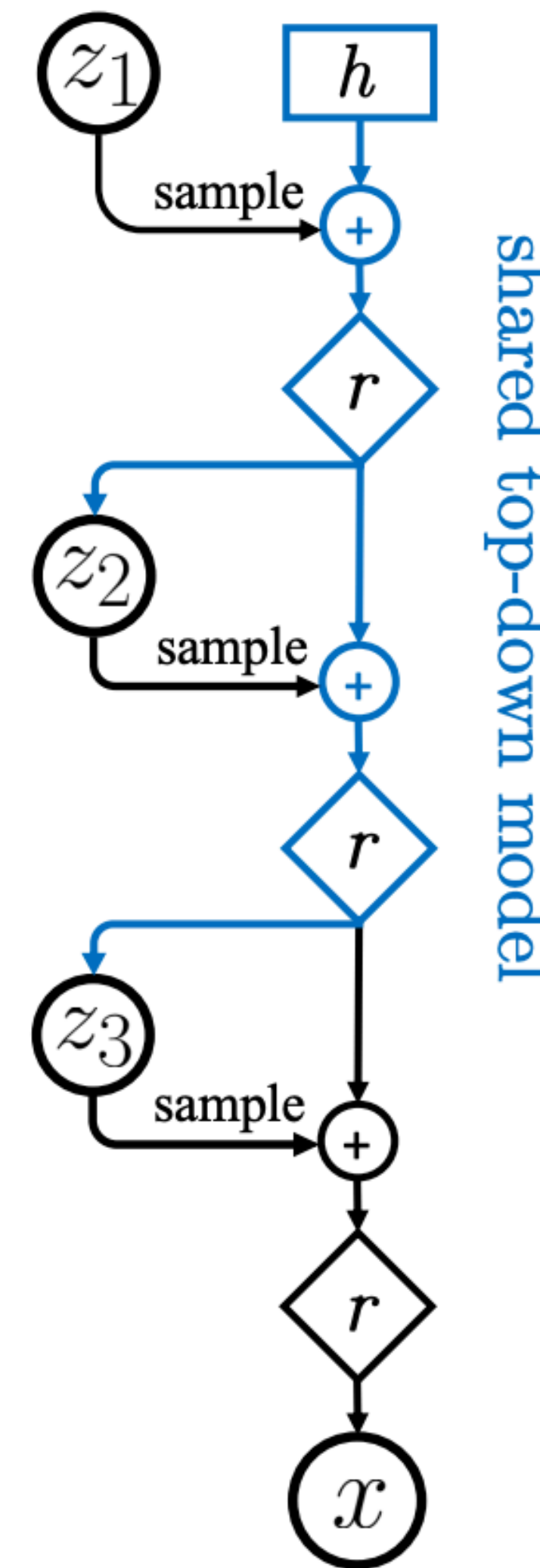
Partition $\mathbf{z}$ into L groups:

$$p(\mathbf{z}) = \prod_l p(\mathbf{z}_l | \mathbf{z}_{<l})$$

Encoders:

$$q(\mathbf{z}_{<l} | \mathbf{x}) := \prod_{i=1}^{l-1} q(\mathbf{z}_i | \mathbf{x}, \mathbf{z}_{<i})$$

$\mathbf{z}$ variables arranged spatially and multiscale (2x size each level)



Source: [Vahadt & Kautz, 2020]

Qualitative results



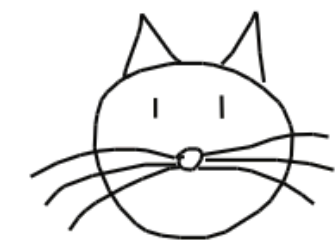
(d) CelebA HQ ($t = 0.6$)

(e) FFHQ ($t = 0.5$)

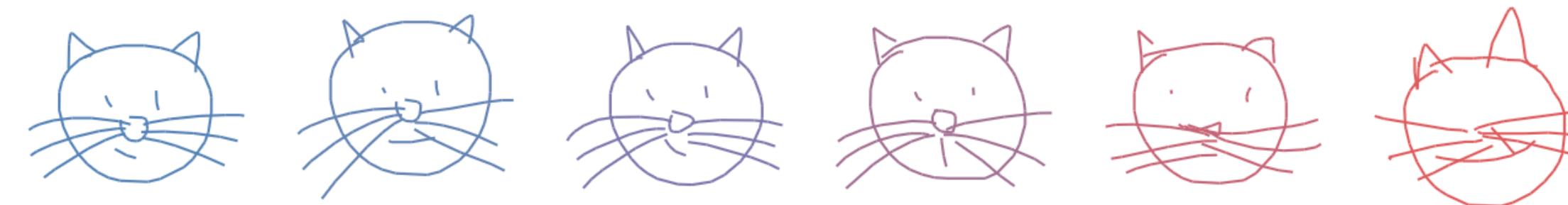
Source: [Vahadt & Kautz, 2020]

Other architectures: recurrent net

Human Input



Reconstructions



RNN for sketch re-generation, predicting (x, y) points. Source: [Ha & Eck, "Sketch-RNN", 2018]

Next class: normalizing flows