CS/DS 552: Class 6

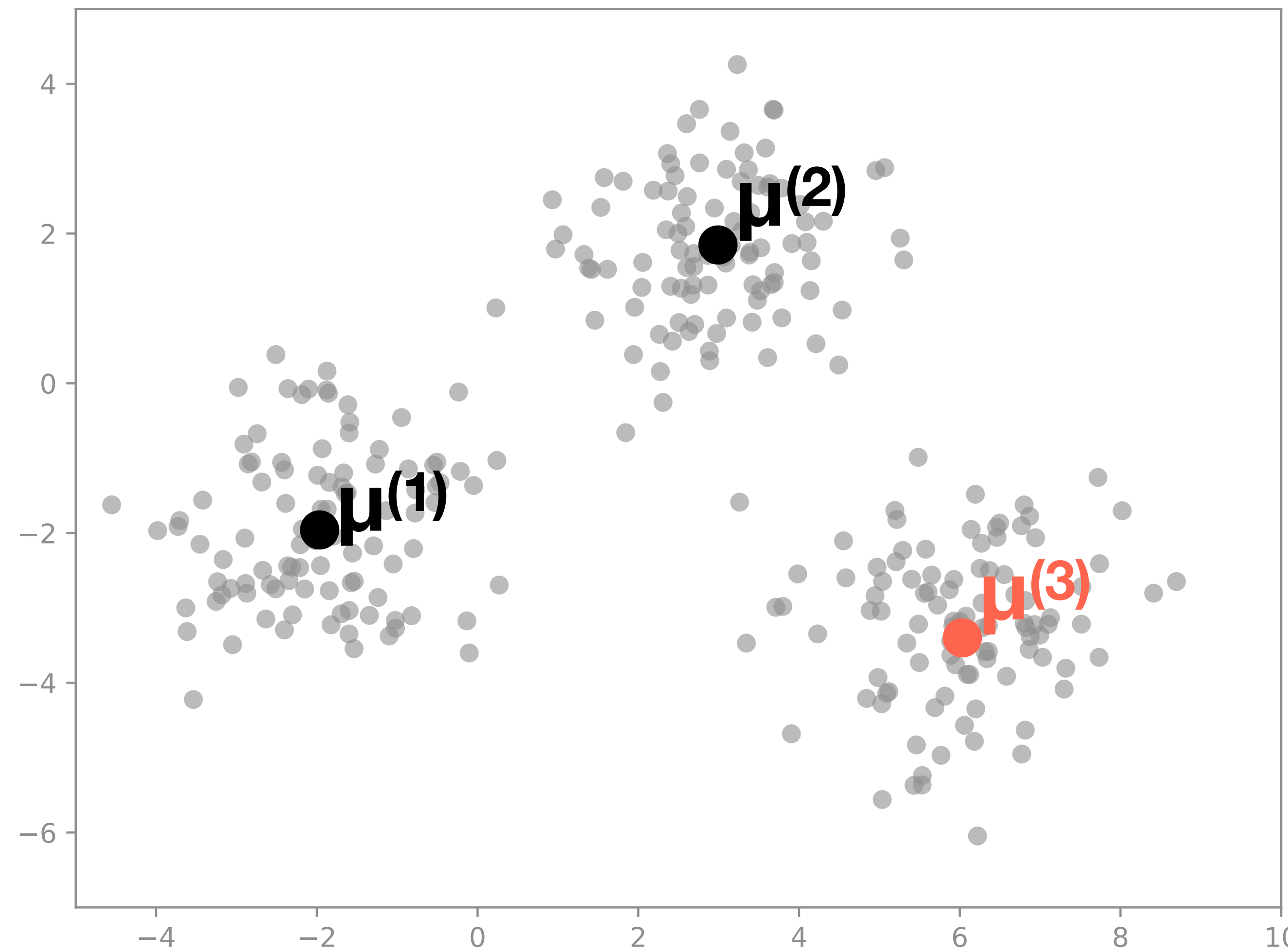
Jacob Whitehill

Vector Quantized (VQ) VAEs

Vector Quantization

- Consider the set of points $\{ \mathbf{x}^{(i)} \}$ shown to the right.

- We could quantize each $\mathbf{x}$ by mapping it to the closest vector from the set $\{ \boldsymbol{\mu}^{(1)}, \boldsymbol{\mu}^{(2)}, \boldsymbol{\mu}^{(3)} \}$.

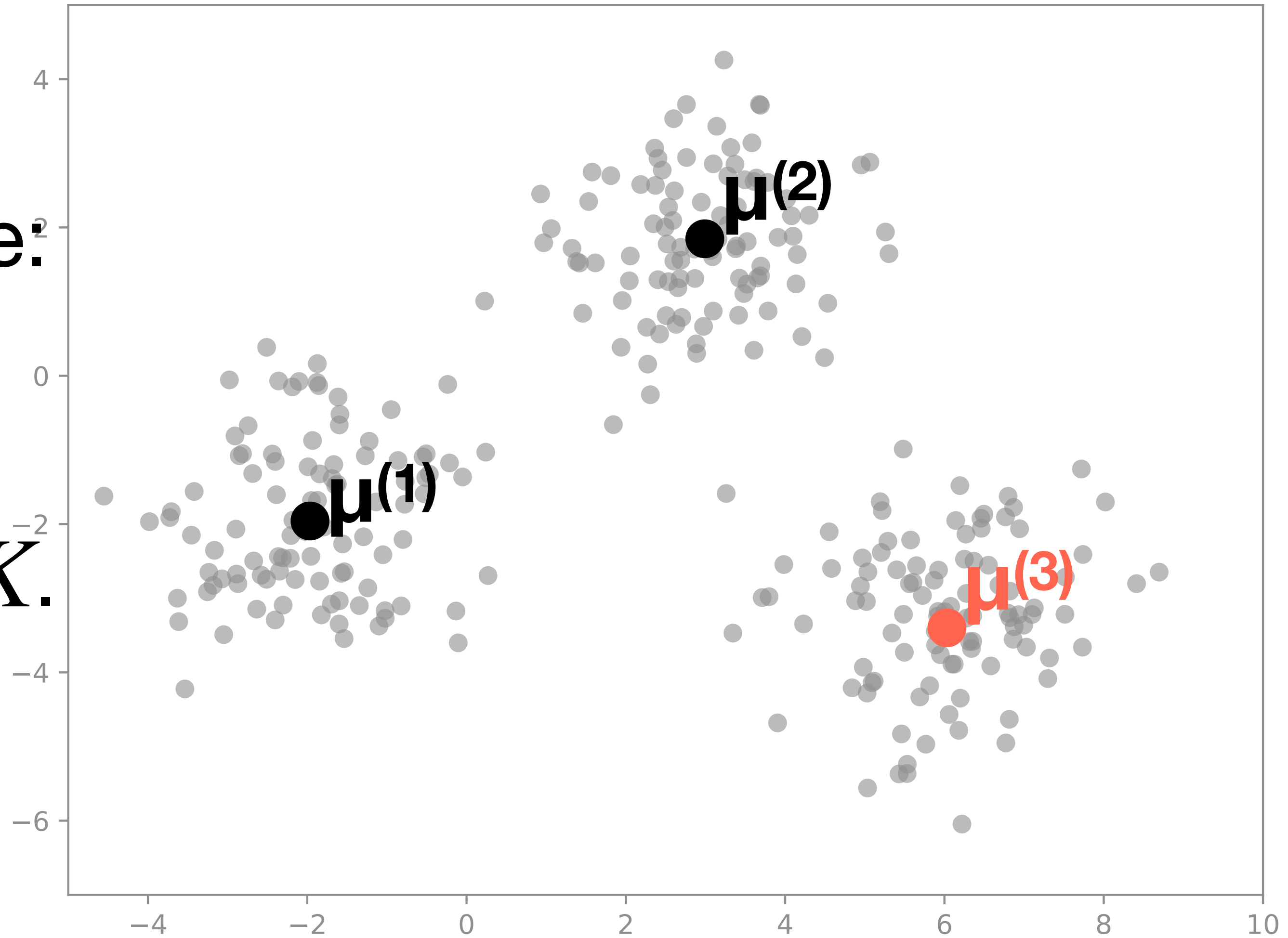


Vector Quantization

- We can (equivalently) consider this process to be:

  - A map from $\mathbb{R}^m$ to $\mathcal{S} \subset \mathbb{R}$, where $|\mathcal{S}| = K$.
    Produces a *vector*

  - A map from $\mathbb{R}^m$ to $\{1, \ldots, K\}$.
    Produces an integer/index



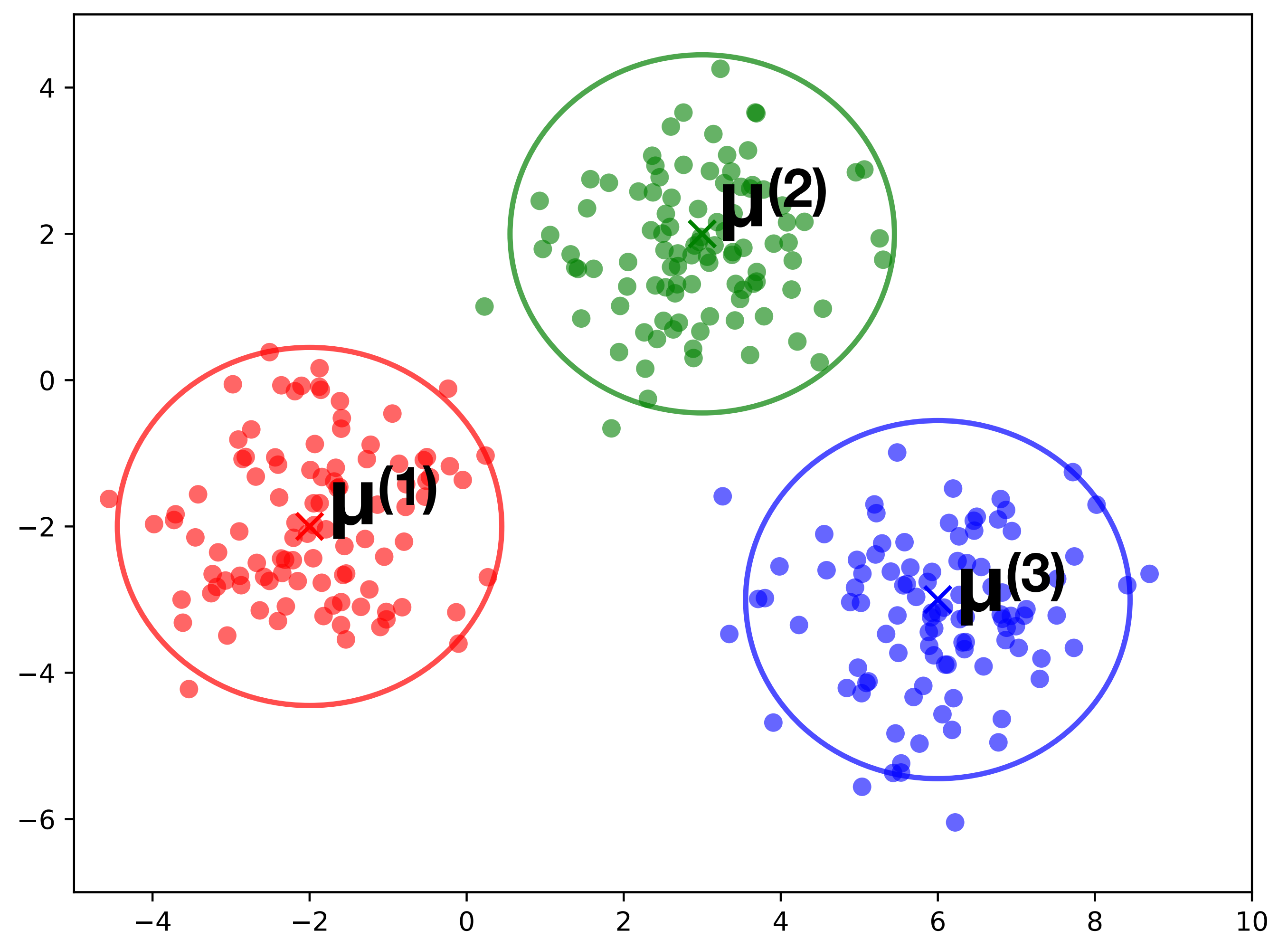
Vector Quantization

- The set of centroids $\{ \boldsymbol{\mu}^{(1)}, \boldsymbol{\mu}^{(2)}, \boldsymbol{\mu}^{(3)} \}$ is typically obtained with *K*-means or a GMM.

- *K*-Means objective:

$$\min_{\{\mu^{(k)}\},\{z^{(i)}\}} \sum_{i=1}^{n} \|\mathbf{x}^{(i)} - \mu^{(z^{(i)})}\|^2$$

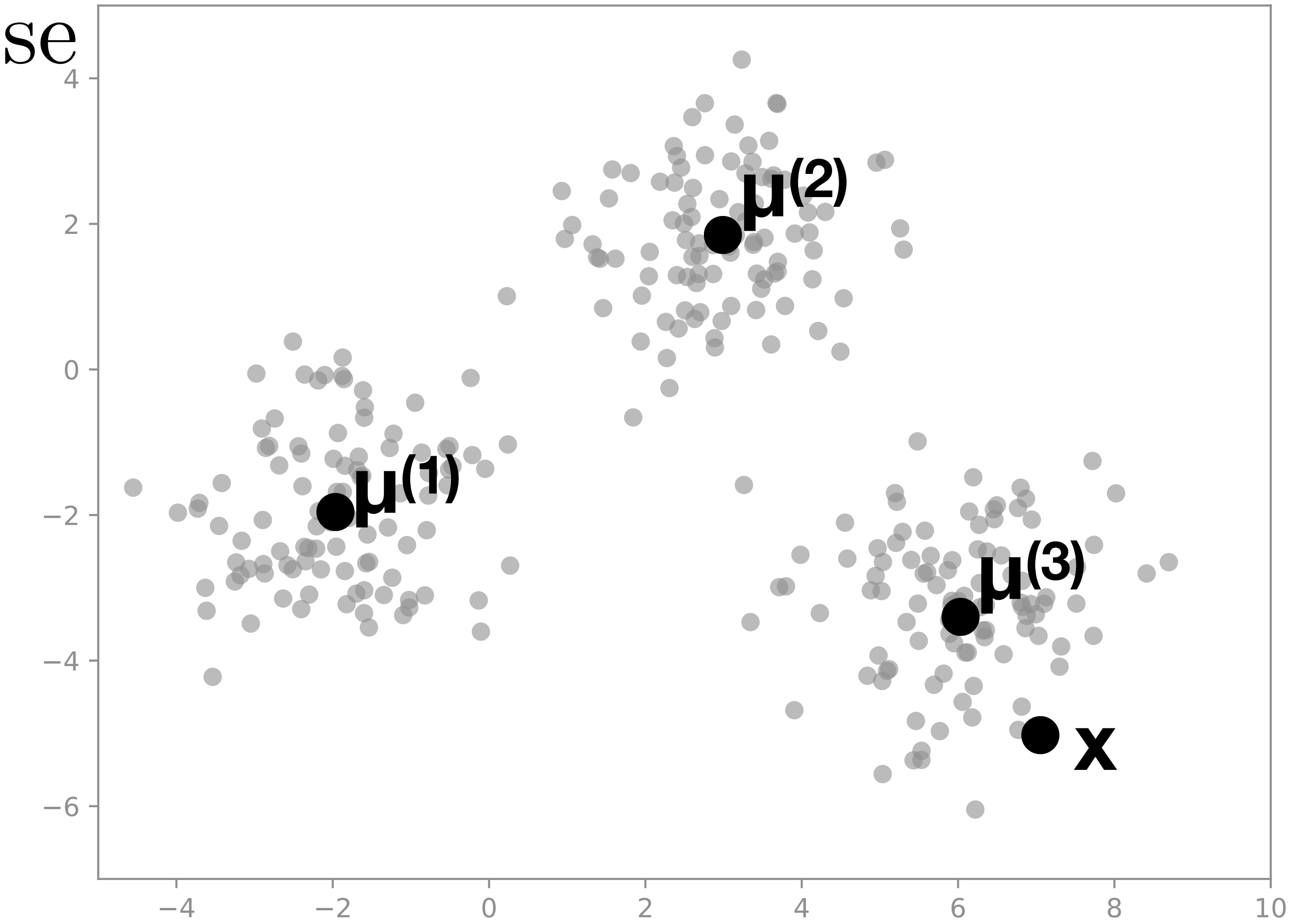
where $z^{(i)} \in \{1,\ldots,K\}$ indicates the cluster that $\mathbf{x}^{(i)}$ belongs to.

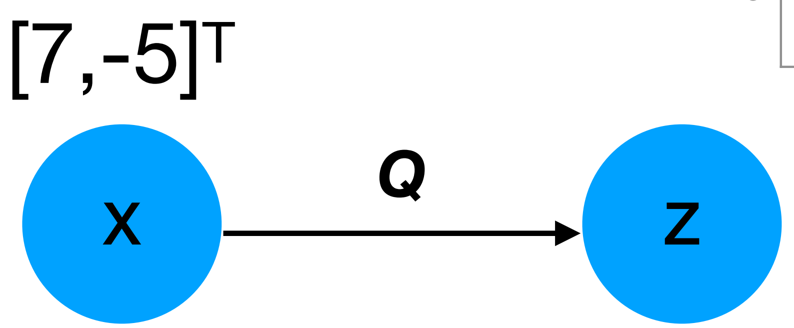


Shallow VQ VAE

- We can also construct a simple discrete "VAE":
  1. Q maps **x** deterministically to index of closest centroid:

$$Q(z \mid \mathbf{x}) = \begin{cases} 1 & \text{if } z = \arg\min_k \|\mathbf{x} - \mu^{(k)}\|^2 \\ 0 & \text{otherwise} \end{cases}$$

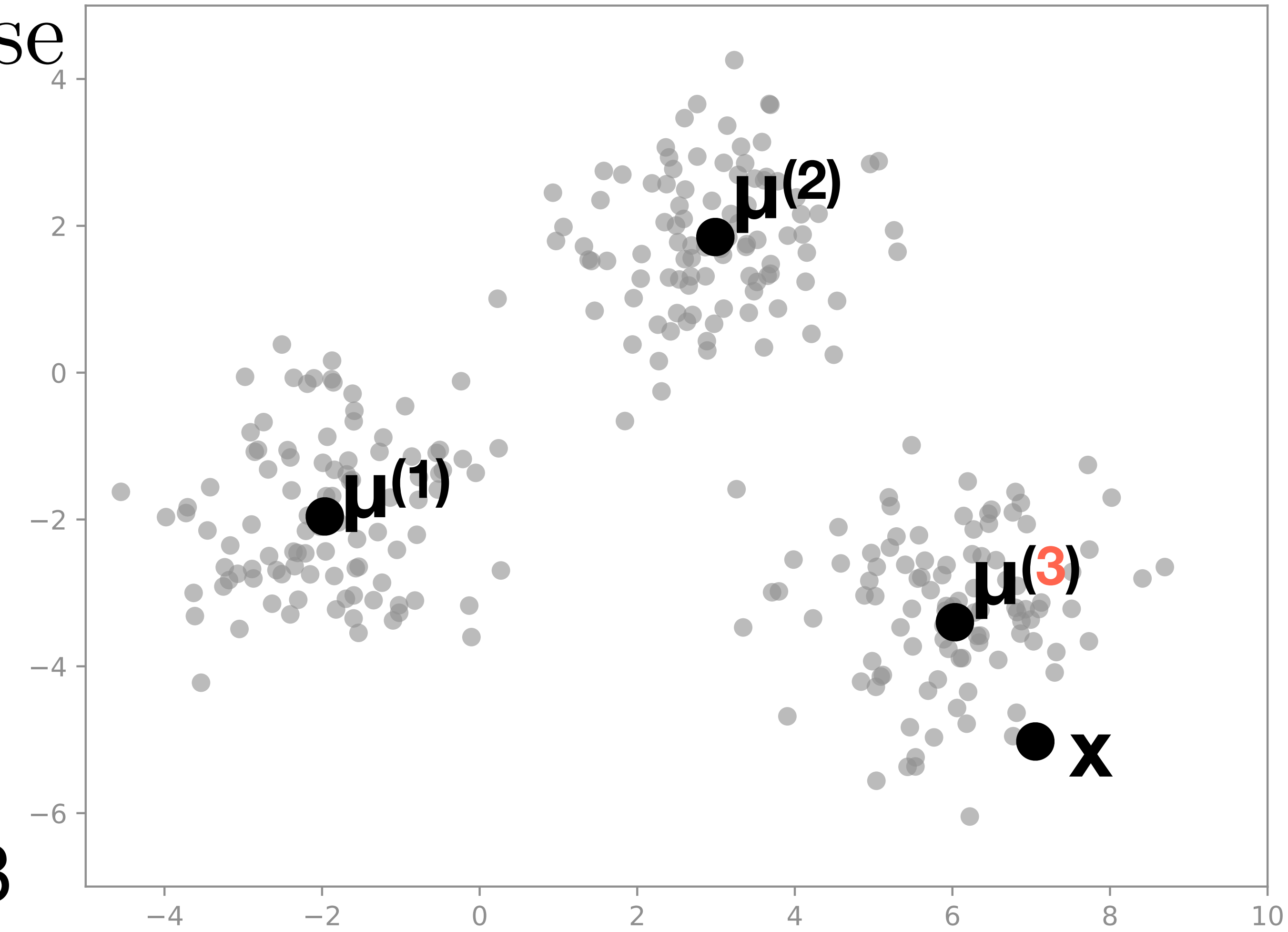


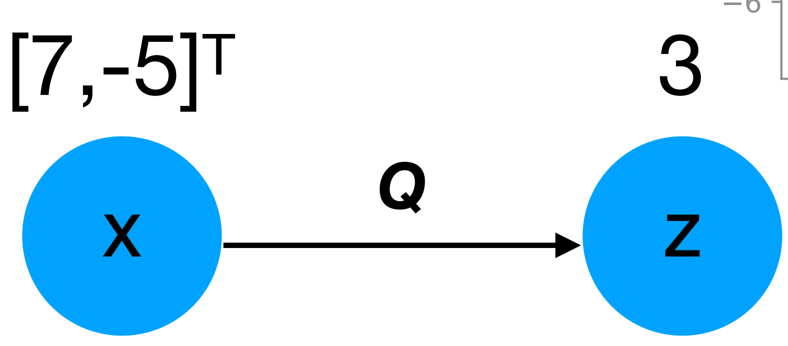


Shallow VQ VAE

- We can also construct a simple discrete “VAE”:
  1. *Q* maps **x** deterministically to index of closest centroid:

$$Q(z \mid \mathbf{x}) = \begin{cases} 1 & \text{if } z = \arg\min_k \|\mathbf{x} - \mu^{(k)}\|^2 \\ 0 & \text{otherwise} \end{cases}$$





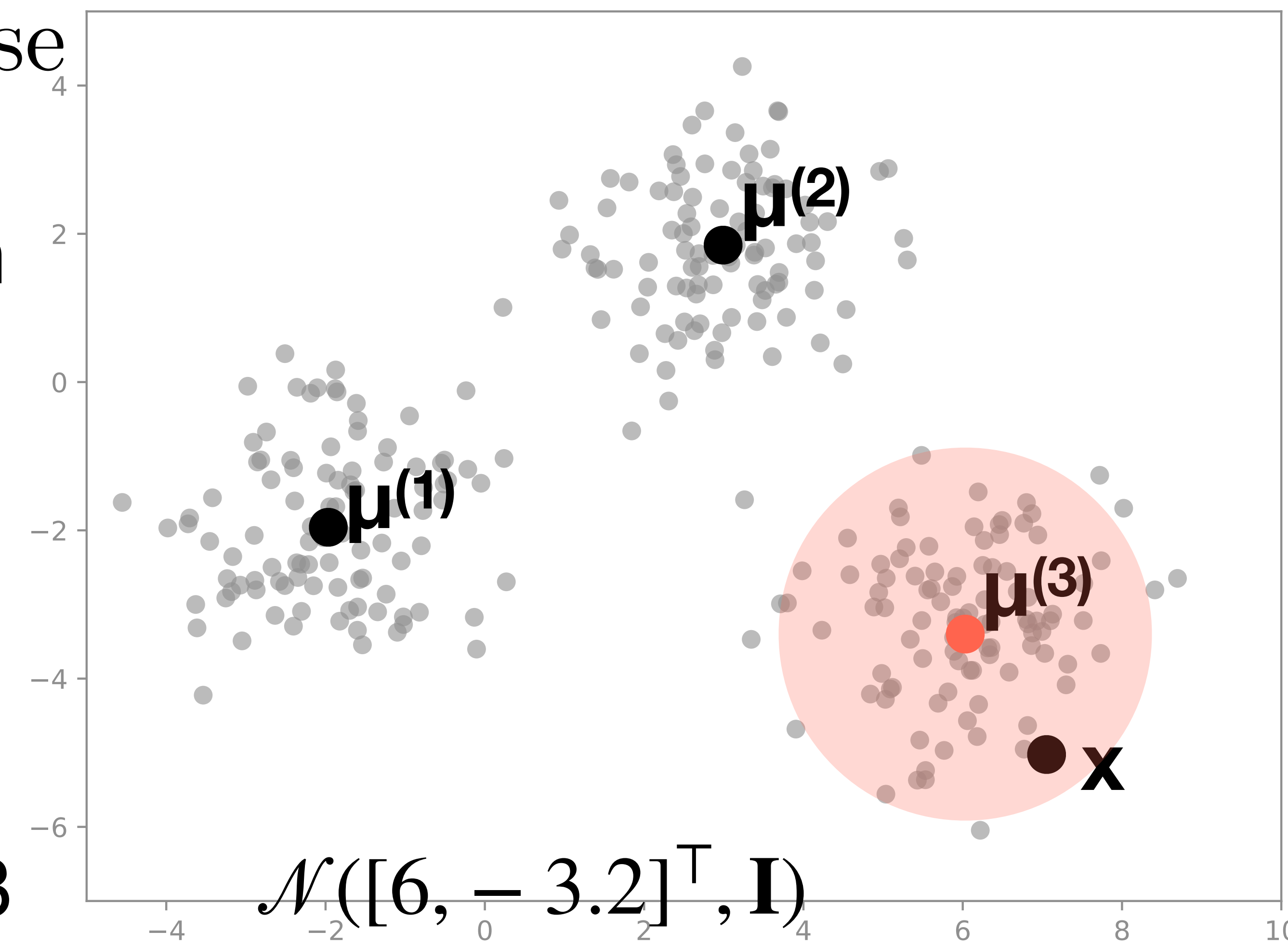
Shallow VQ VAE

- We can also construct a simple discrete "VAE":

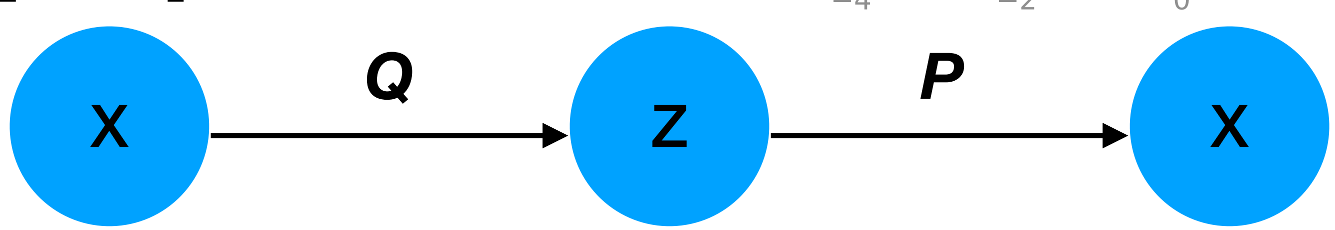
  1. *Q* maps **x** deterministically to index of closest centroid:

$$Q(z \mid \mathbf{x}) = \begin{cases} 1 & \text{if } z = \arg\min_k \|\mathbf{x} - \mu^{(k)}\|^2 \\ 0 & \text{otherwise} \end{cases}$$

  2. *P* maps z to a Gaussian centered at $\boldsymbol{\mu}^{(z)}$, i.e.,
  $P(\mathbf{x} \mid z) = \mathcal{N}(\mu^{(z)}, \mathbf{I})$



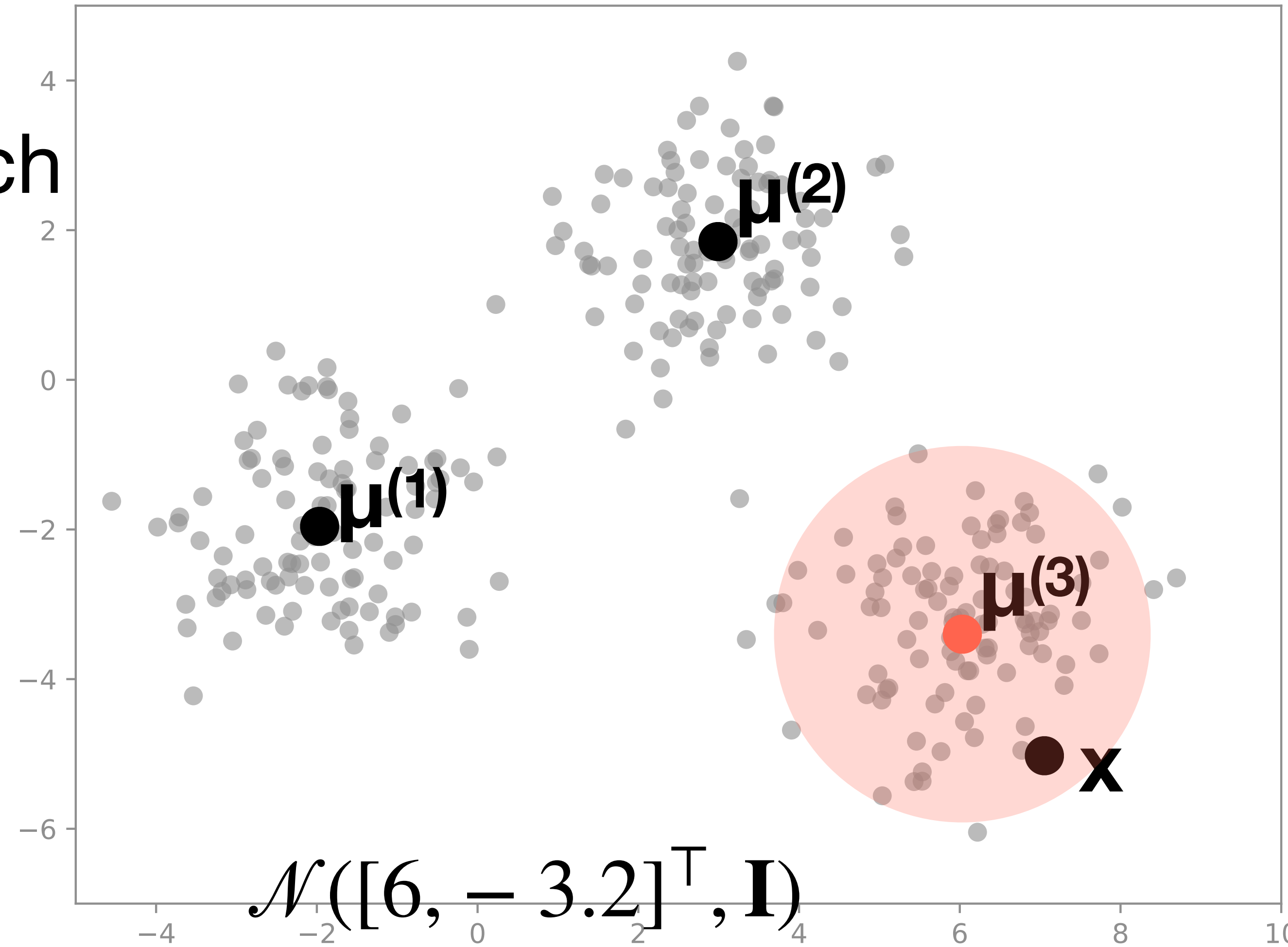
$[7,-5]^{\mathsf{T}}$ 3 $\mathcal{N}([6,-3.2]^{\mathsf{T}}, \mathbf{I})$

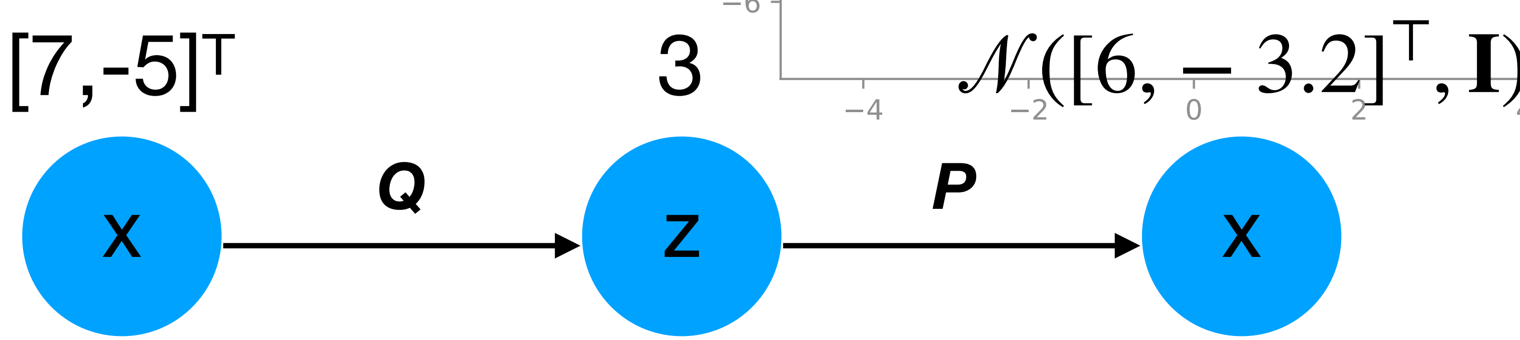


Shallow VQ VAE

- We can easily train this simple “VQ VAE” using MLE with a Gaussian Mixture Model (GMM).

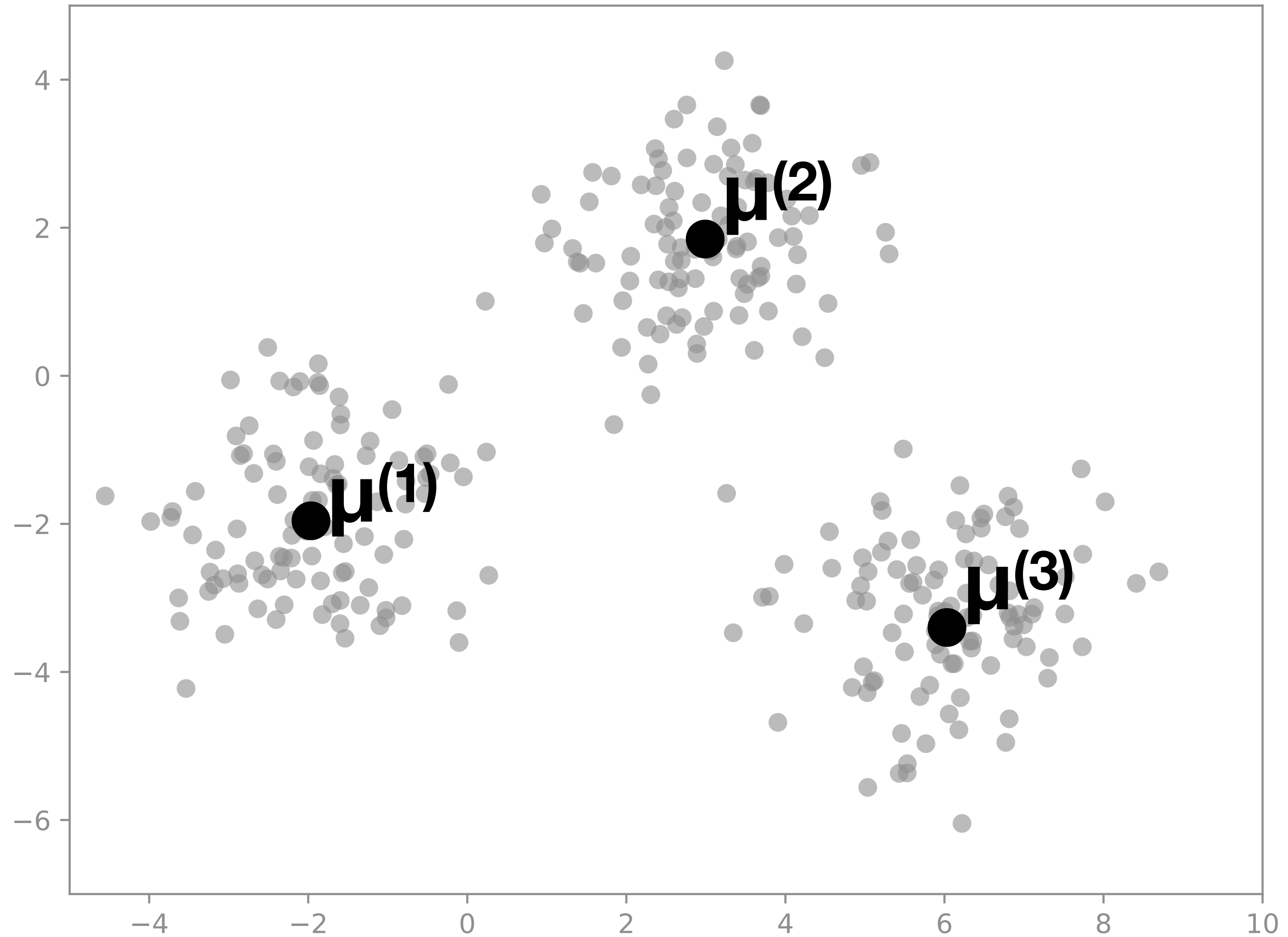
- In particular, we can enforce $P(z) = \mathscr{U}(1,\ldots,K)$, i.e., each centroids is equally likely.

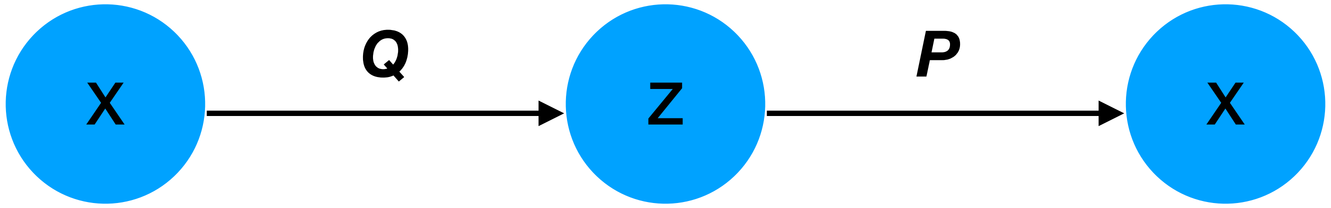




Generation

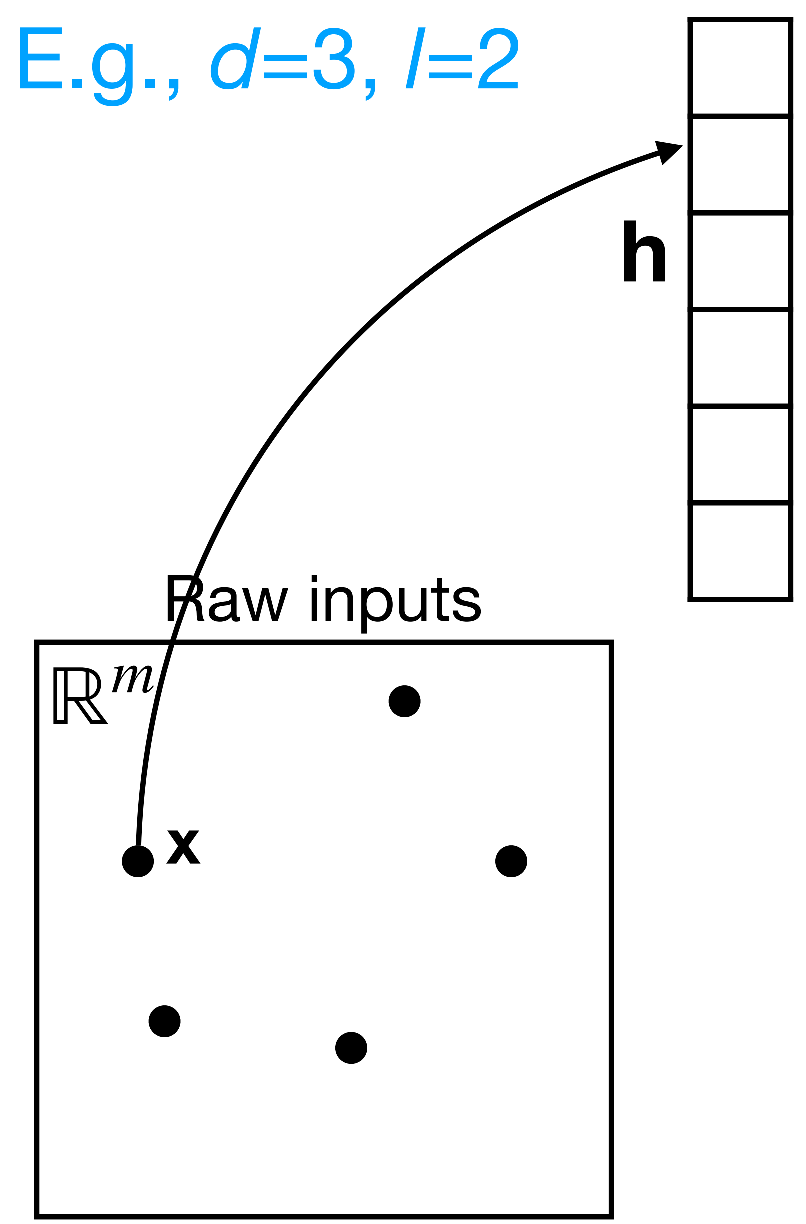
- After training, to generate new data we just:
    1. Sample $z \sim P(z) = \mathcal{U}(1,\ldots,K)$.
    2. Sample $\mathbf{x} \sim P(\mathbf{x} \mid z) = \mathcal{N}(\mu^{(z)}, \mathbf{I})$
- However, this model is very weak — we are just adding noise to centroids in the raw image space.





(Deep) VQ-VAEs

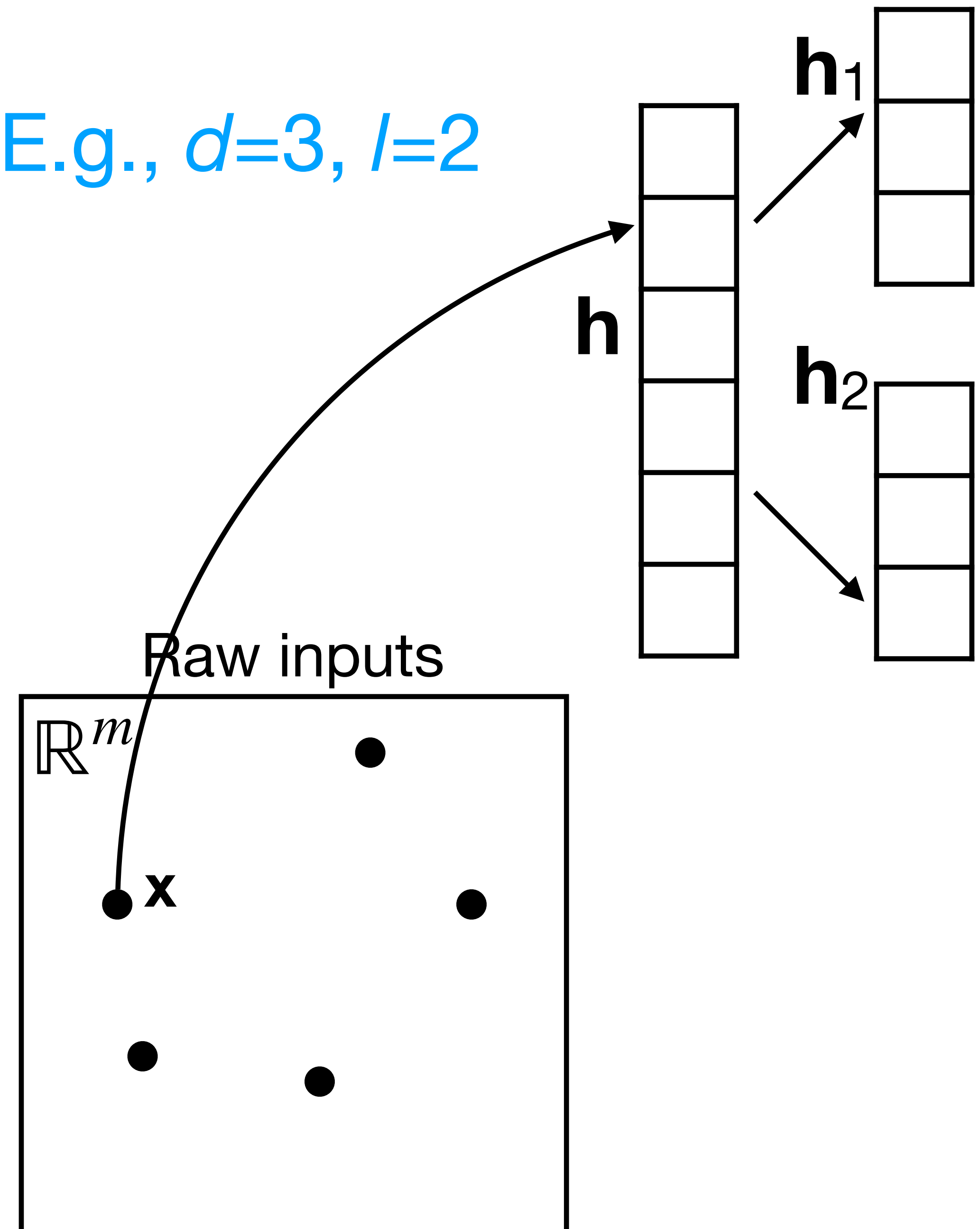
- We can generalize this idea into a deep VQ-VAE:
    1. Transform each $\mathbf{x} \in \mathbb{R}^m$ into a feature vector $\mathbf{h} \in \mathbb{R}^{l \times d}$.



(Deep) VQ-VAEs

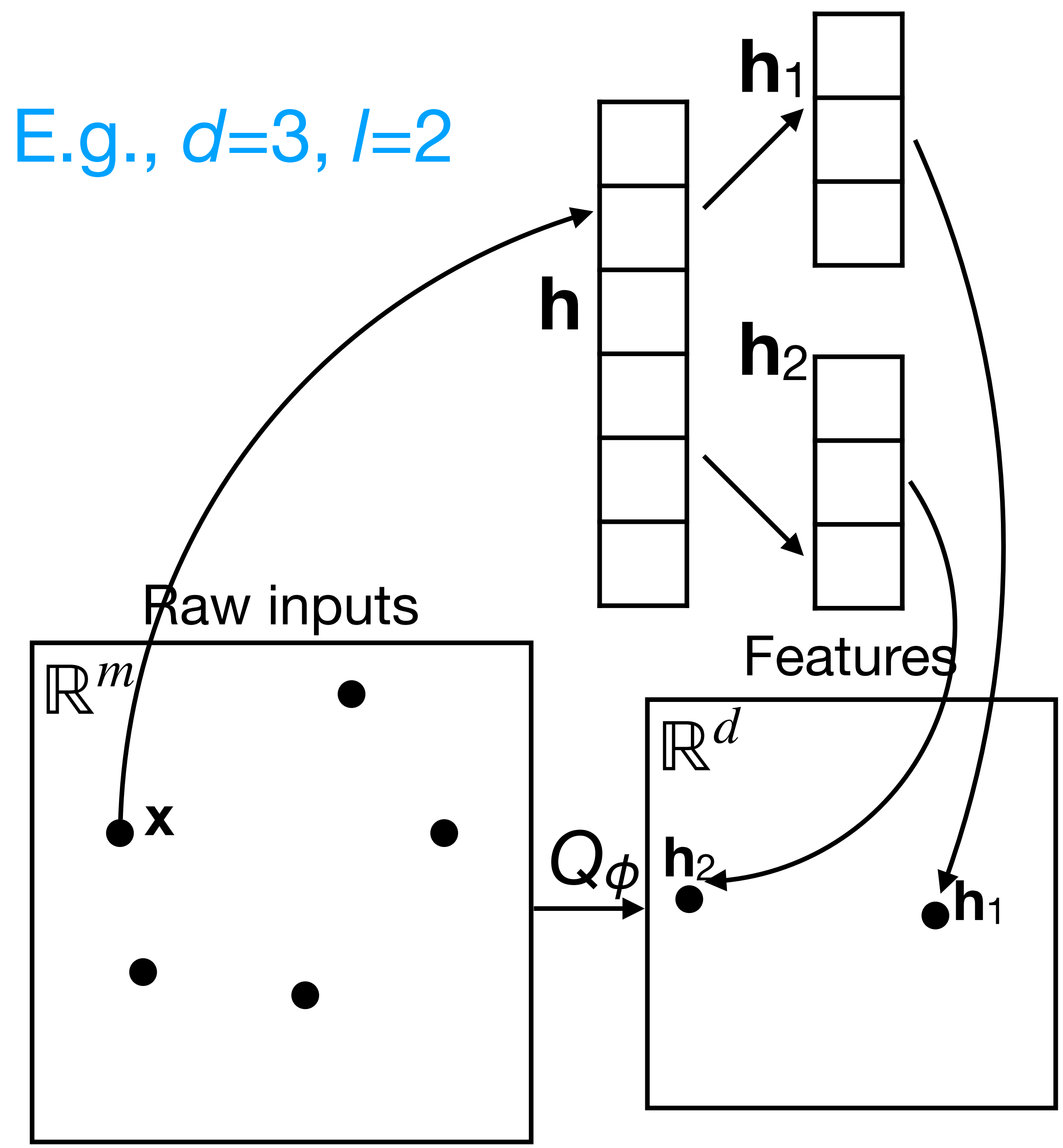
- We can generalize this idea into a deep VQ-VAE:
   1. Transform each $\mathbf{x} \in \mathbb{R}^m$ into a feature vector $\mathbf{h} \in \mathbb{R}^{l \times d}$.
   2. Split $\mathbf{h}$ into multiple (l) vectors $\mathbf{h}_1, \ldots, \mathbf{h}_l \in \mathbb{R}^d$.

E.g., d=3, l=2



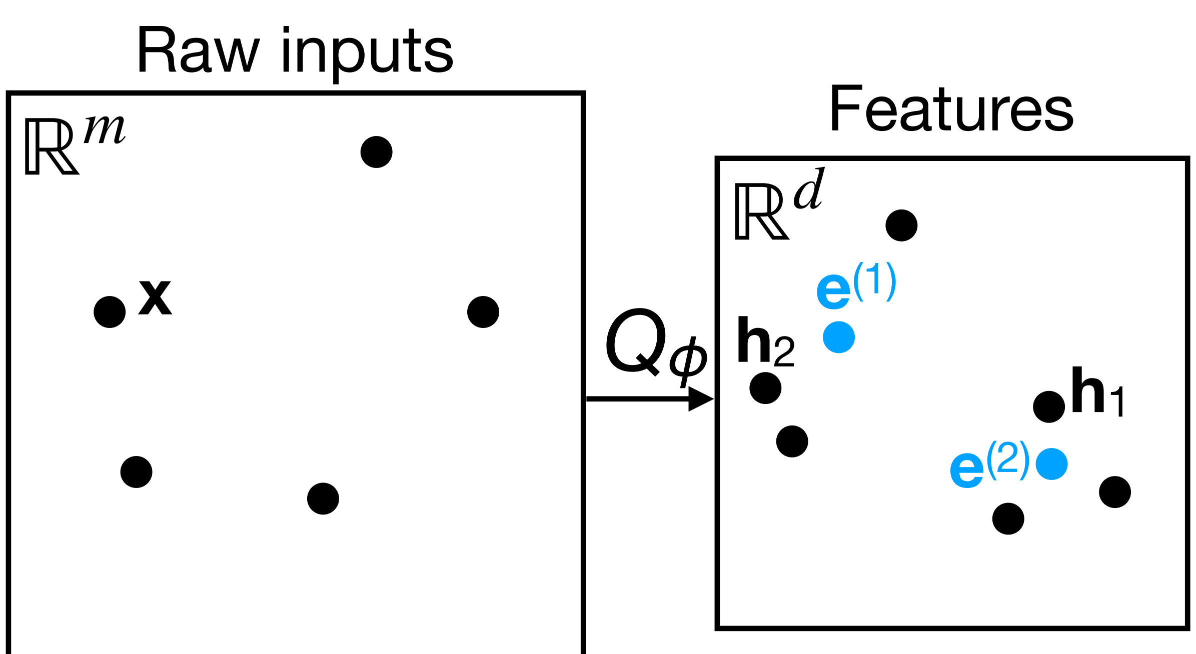
(Deep) VQ-VAEs

- We can generalize this idea into a deep VQ-VAE:
    1. Transform each $\mathbf{x} \in \mathbb{R}^m$ into a feature vector $\mathbf{h} \in \mathbb{R}^{l \times d}$.
    2. Split $\mathbf{h}$ into multiple (l) vectors $\mathbf{h}_1, \ldots, \mathbf{h}_l \in \mathbb{R}^d$.



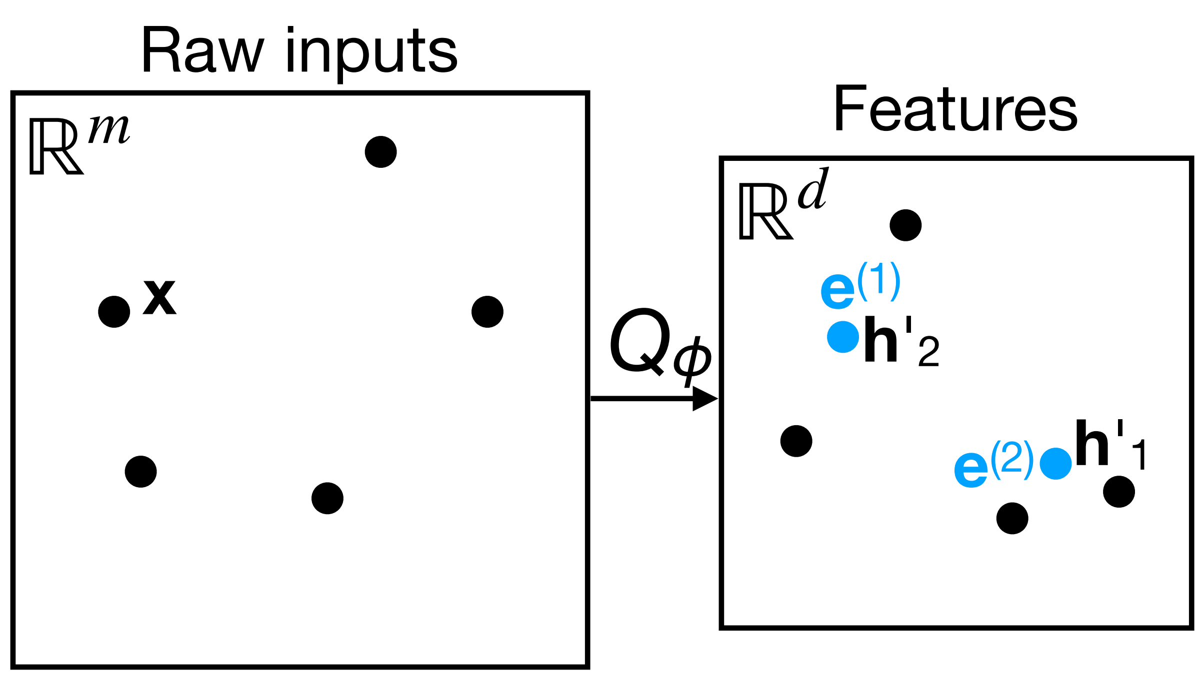
(Deep) VQ-VAEs

- We can generalize this idea into a deep VQ-VAE:
  1. Transform each $\mathbf{x} \in \mathbb{R}^m$ into a feature vector $\mathbf{h} \in \mathbb{R}^{l \times d}$.
  2. Split $\mathbf{h}$ into multiple (l) vectors $\mathbf{h}_1, \ldots, \mathbf{h}_l \in \mathbb{R}^d$.
  3. Using running estimates of K cluster centroids over $\{\mathbf{h}_j^{(i)}\}_{i,j}$,



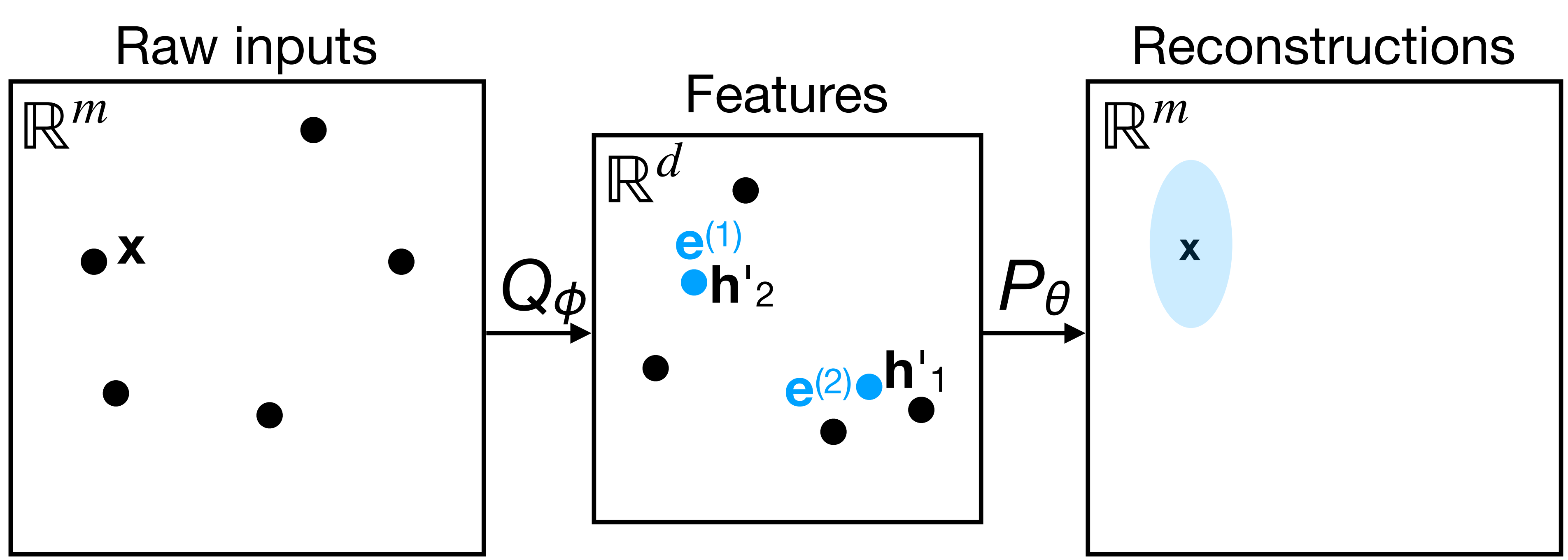
(Deep) VQ-VAEs

- We can generalize this idea into a deep VQ-VAE:
  1. Transform each $\mathbf{x} \in \mathbb{R}^m$ into a feature vector $\mathbf{h} \in \mathbb{R}^{l \times d}$.
  2. Split $\mathbf{h}$ into multiple (l) vectors $\mathbf{h}_1, \ldots, \mathbf{h}_l \in \mathbb{R}^d$.
  3. Using running estimates of K cluster centroids over $\{\mathbf{h}_j^{(i)}\}_{i,j}$, quantize each $\mathbf{h}_j$ into $\mathbf{h}'_j$ using the nearest centroid $\mathbf{e}^{(z)}$.



(Deep) VQ-VAEs

- We can generalize this idea into a deep VQ-VAE:
  1. Transform each $\mathbf{x} \in \mathbb{R}^m$ into a feature vector $\mathbf{h} \in \mathbb{R}^{l \times d}$.
  2. Split $\mathbf{h}$ into multiple (l) vectors $\mathbf{h}_1, \ldots, \mathbf{h}_l \in \mathbb{R}^d$.
  3. Using running estimates of K cluster centroids over $\{\mathbf{h}_j^{(i)}\}_{i,j}$, quantize each $\mathbf{h}_j$ into $\mathbf{h}'_j$ using the nearest centroid $\mathbf{e}^{(z)}$.
  4. Concatenate the $\mathbf{h}'_1, \ldots, \mathbf{h}'_l$ into $\mathbf{h}'$, and then transform $\mathbf{h}'$ into $P(\mathbf{x} \mid \mathbf{h}') = P(\mathbf{x} \mid \{z_j\})$.



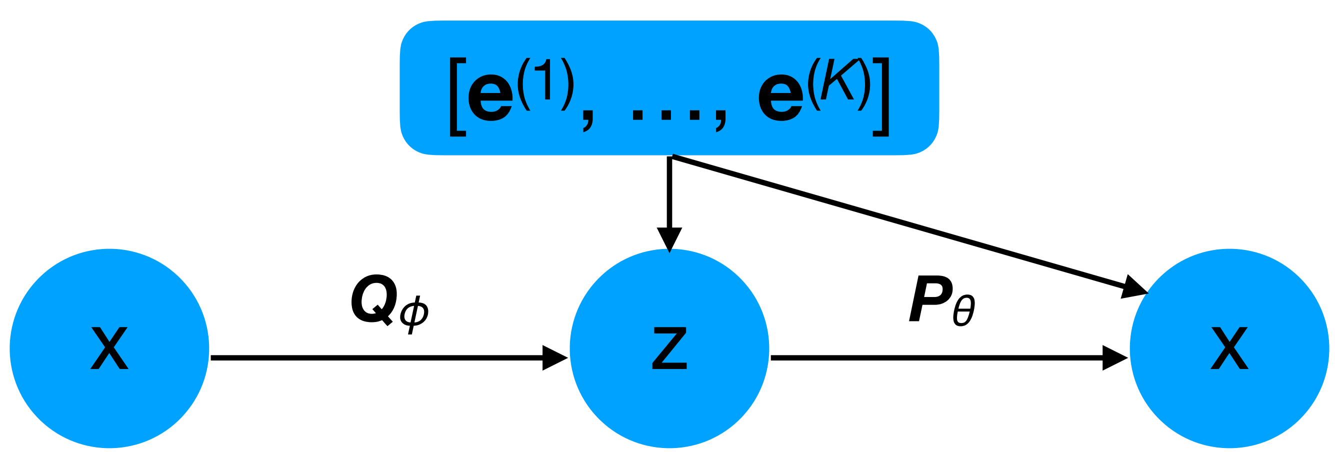
(Deep) VQ-VAEs

- The parameters that must be learned in this VQ-VAE include ϕ (encoder), θ (decoder), and $\mathbf{E}$=[$\mathbf{e}^{(1)}$, …, $\mathbf{e}^{(K)}$] (cluster centroids).

- To optimize these parameters, we will start with an MLE approach…

- Recall the ELBO for continuous VAEs:

$$-D_{\mathrm{KL}}(Q_\phi(z \mid \mathbf{x}) \mid P(z)) + \mathbb{E}_{Q_\phi}[\log P(\mathbf{x} \mid z)]$$



D_{KL} for VQ-VAEs

- For VQ-VAEs, we want $P(z) = \mathscr{U}(1, \ldots, K) = \dfrac{1}{K}\ \forall z$.

- We have a deterministic encoder:

$$Q(z \mid \mathbf{x}) = \begin{cases} 1 & \text{if } z = \arg\min_k \|\mathbf{x} - \mathbf{e}^{(k)}\|^2 \\ 0 & \text{otherwise} \end{cases}$$

D_{KL} for VQ-VAEs

- For VQ-VAEs, we want $P(z) = \mathscr{U}(1,\ldots,K) = \dfrac{1}{K}\ \forall z$.

- We have a deterministic encoder:

$$Q(z \mid \mathbf{x}) = \begin{cases} 1 & \text{if } z = \arg\min_k \|\mathbf{x} - \mathbf{e}^{(k)}\|^2 \\ 0 & \text{otherwise} \end{cases}$$

- By definition of KL divergence, we have:

$$D_{\mathrm{KL}}(Q_\phi(z \mid \mathbf{x}) \mid P(z)) = \sum_{z=1}^{K} Q(z \mid \mathbf{x}) \log \frac{Q(z \mid \mathbf{x})}{P(z)}$$

where 0 log 0 is defined to be 0.

D_{KL} for VQ-VAEs

- For VQ-VAEs, we want $P(z) = \mathscr{U}(1, \ldots, K) = \dfrac{1}{K}\ \forall z$.

- We have a deterministic encoder:

$$Q(z \mid \mathbf{x}) = \begin{cases} 1 & \text{if } z = \arg\min_k \|\mathbf{x} - \mathbf{e}^{(k)}\|^2 \\ 0 & \text{otherwise} \end{cases}$$

- By definition of KL divergence, we have:

$$\begin{aligned} D_{\text{KL}}(Q_\phi(z \mid \mathbf{x}) \mid P(z)) &= \sum_{z=1}^{K} Q(z \mid \mathbf{x}) \log \frac{Q(z \mid \mathbf{x})}{P(z)} \\ &= 1 \log \frac{1}{\frac{1}{K}} + \sum_{\ldots} 0 \log \frac{0}{\frac{1}{K}} \\ &= \log K \end{aligned}$$

D_{KL} for VQ-VAEs

- Since log K does not depend on any of the VQ-VAE's parameters (ϕ, θ, $\mathbf{E}$), it can be ignored from the ELBO.

- That just leaves:

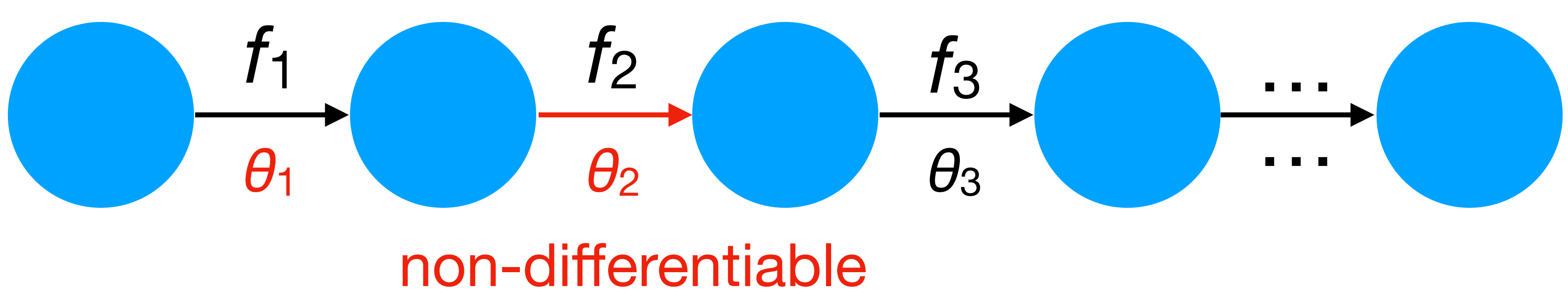
$$\cancel{-D_{\mathrm{KL}}(Q_\phi(z \mid \mathbf{x}) \mid P(z))} + \mathbb{E}_{Q_\phi}[\log P(\mathbf{x} \mid z)]$$

D_{KL} for VQ-VAEs

- In practice, this means that $Q(z \mid \mathbf{x})$ and $P(z)$ in VQ-VAEs tend to be very different from $\mathcal{U}(1,\ldots,K) = \frac{1}{K}\ \forall z$.
- Hence, sampling from $P(z)$ naively tends to produce very bad results.
- Instead, we may need to train an autoregressor on the latent space $P(\mathbf{z})$.

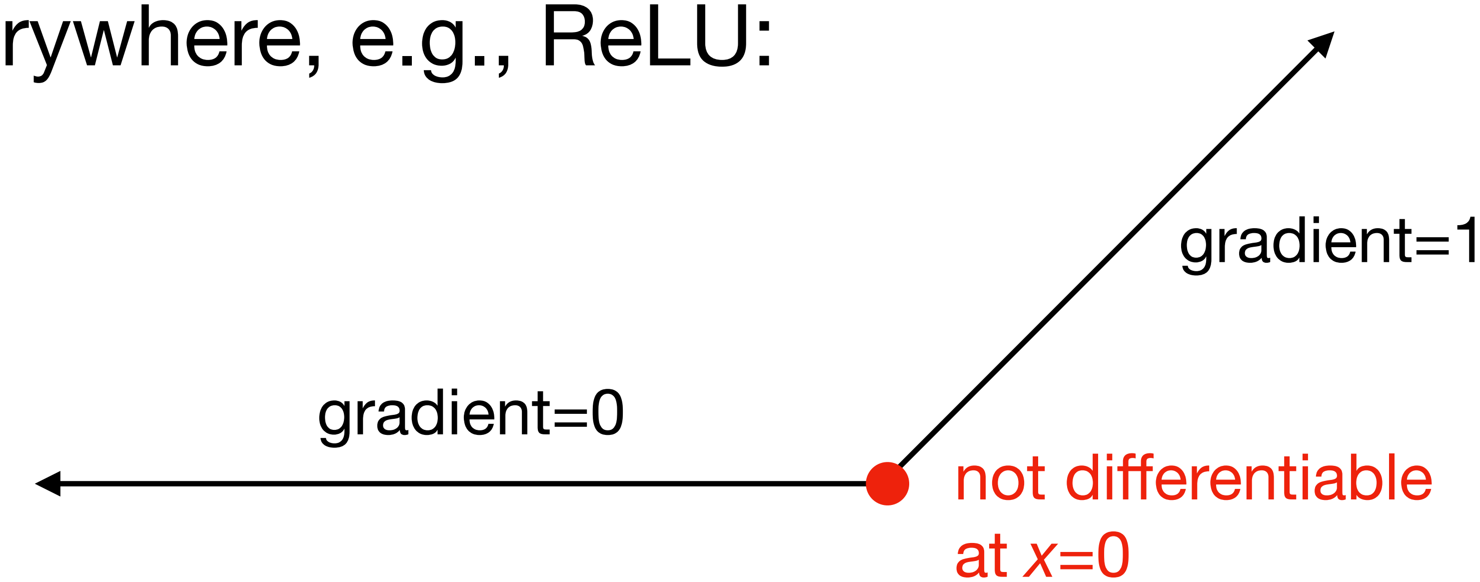
Non-differentiability

- In a computational graph (e.g., a NN), if any operation is non-differentiable, then we cannot use gradient descent to update any parameters in or before it.



Non-differentiability

- Contrary to classic optimization theory, modern NNs routinely use function that are not differentiable everywhere, e.g., ReLU:



- However, ReLU is differentiable *almost everywhere*, and where the derivative exists, it is “often” non-zero
=> learning can occur.

Exercise

- Consider the following functions:

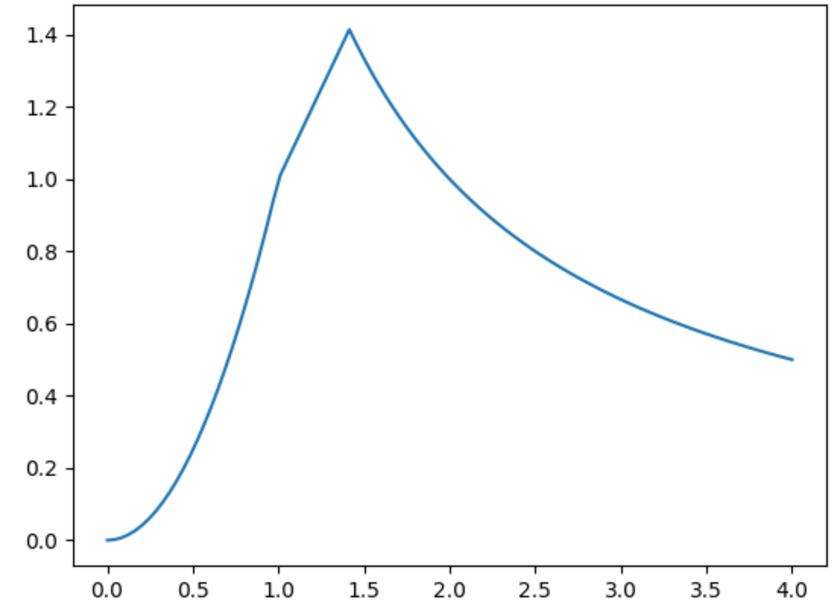
  1. $f(x) = \min\{x, x^2, 2/x\}$

  2. $g(x) = \arg\min\{x, x^2, 2/x\}$

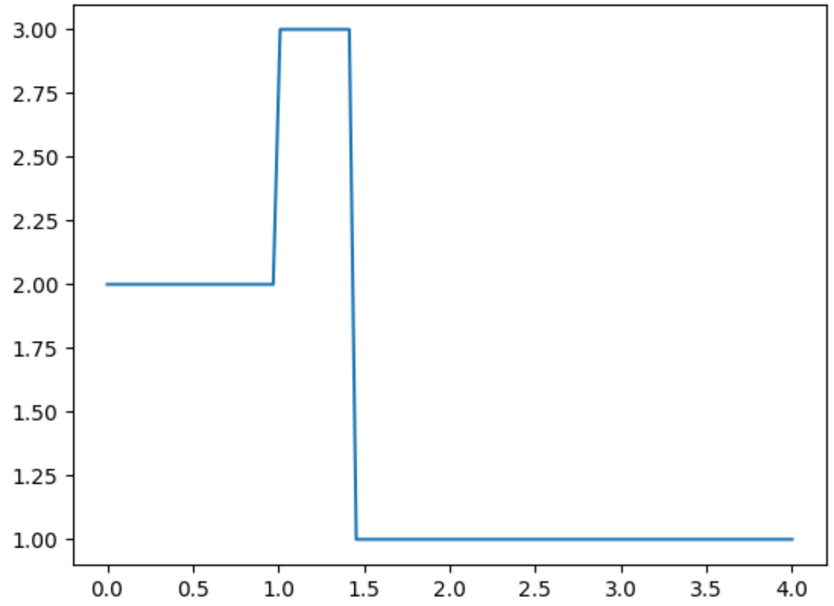
- Could you use (1), (2), or both (1)&(2) within a NN, or would they “break” backpropagation?

Solution

- *f*(*x*) is differentiable almost everywhere and has non-zero gradient where it is defined => we can learn!



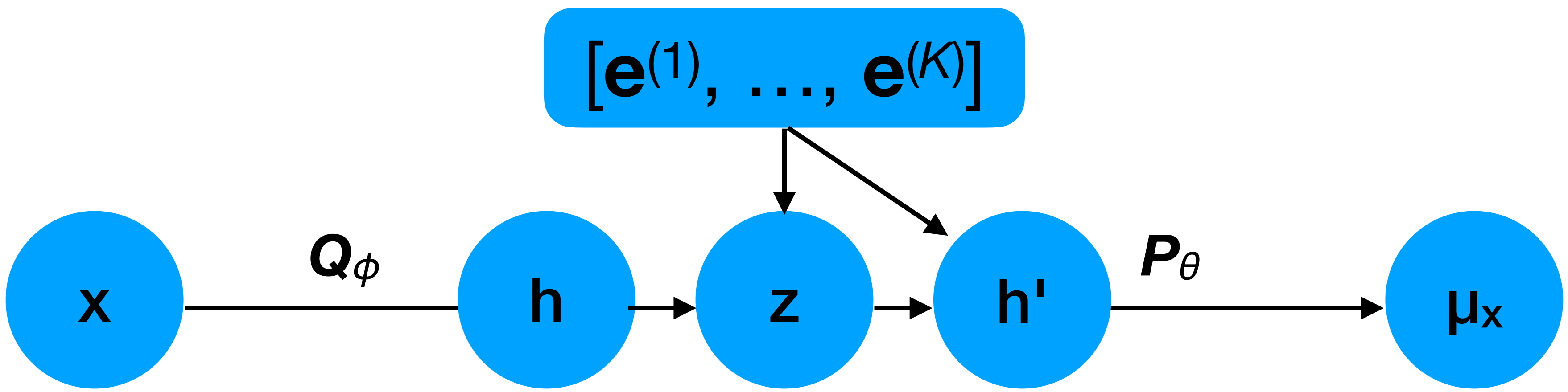
- *g*(*x*) has a gradient that is either 0 or not defined => bad!



Reconstruction probability

$$-D_{\mathrm{KL}}(Q_\phi(z \mid \mathbf{x}) \mid P(z)) + \mathbb{E}_{Q_\phi}[\log P(\mathbf{x} \mid z)]$$

- Like with a VAE, we could try to estimate the second term in the ELBO by sampling K=1 samples, i.e.:
  1. Sample z from $Q(z \mid \mathbf{x})$.
  2. Compute log $P(\mathbf{x} \mid z)$, e.g., $-||\mathbf{x} - \mu_{\mathbf{x}}||^2$



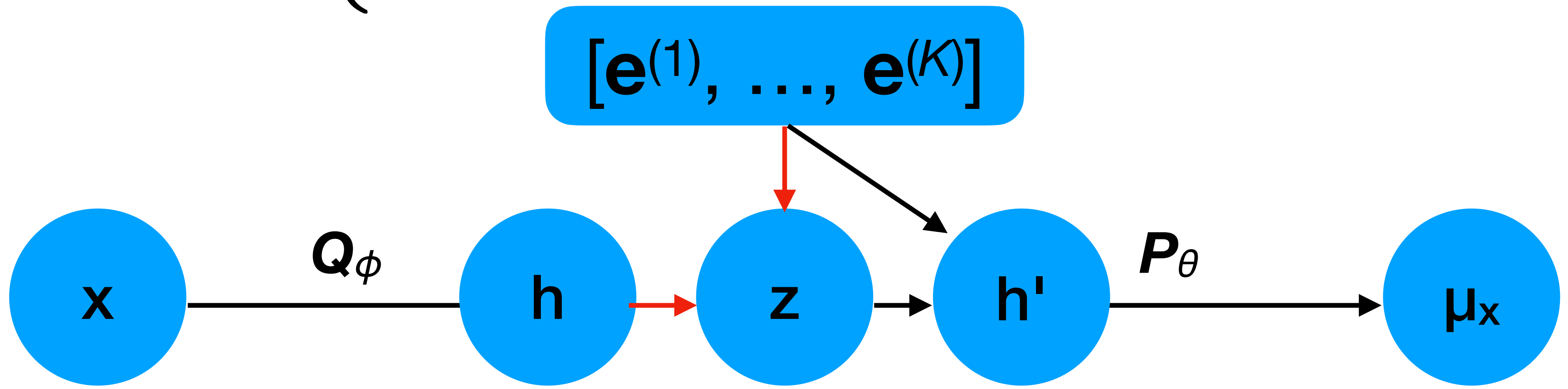
Reconstruction probability

$$-D_{\mathrm{KL}}(Q_\phi(z \mid \mathbf{x}) \mid P(z)) + \boxed{\mathbb{E}_{Q_\phi}[\log P(\mathbf{x} \mid z)]}$$

- Like with a VAE, we could try to estimate the second term in the ELBO by sampling *K*=1 samples, i.e.:
  1. Sample z from $Q(z \mid \mathbf{x})$.
  2. Compute log $P(\mathbf{x} \mid z)$, e.g., $-\|\mathbf{x} - \mu_{\mathbf{x}}\|^2$

- Like with continuous VAE, sampling is non-differentiable:

$$Q(z \mid \mathbf{x}) = \begin{cases} 1 & \text{if } z = \arg\min_k \|\mathbf{x} - \mathbf{e}^{(k)}\|^2 \\ 0 & \text{otherwise} \end{cases}$$



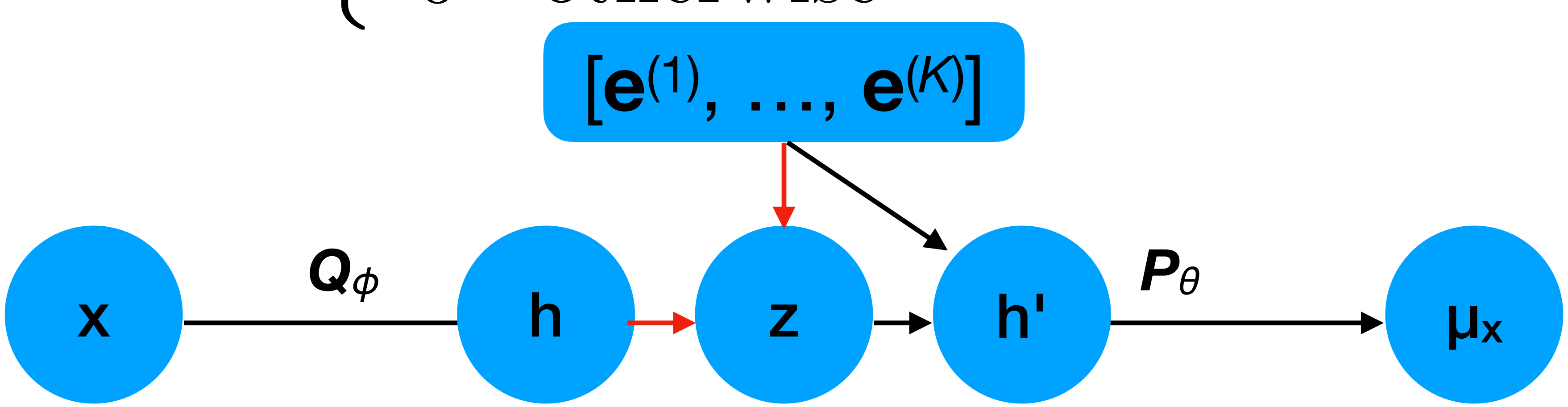
Reconstruction probability

$$-D_{\mathrm{KL}}(Q_\phi(z \mid \mathbf{x}) \mid P(z)) + \mathbb{E}_{Q_\phi}[\log P(\mathbf{x} \mid z)]$$

- Like with a VAE, we could try to estimate the second term in the ELBO by sampling K=1 samples, i.e.:
    1. Sample z from $Q(z \mid \mathbf{x})$.
    2. Compute $\log P(\mathbf{x} \mid z)$, e.g., $-\|\mathbf{x} - \mu_{\mathbf{x}}\|^2$

- Unlike continuous VAE, there is no simple reparam. trick*:

$$Q(z \mid \mathbf{x}) = \begin{cases} 1 & \text{if } z = \arg\min_k \|\mathbf{x} - \mathbf{e}^{(k)}\|^2 \\ 0 & \text{otherwise} \end{cases}$$



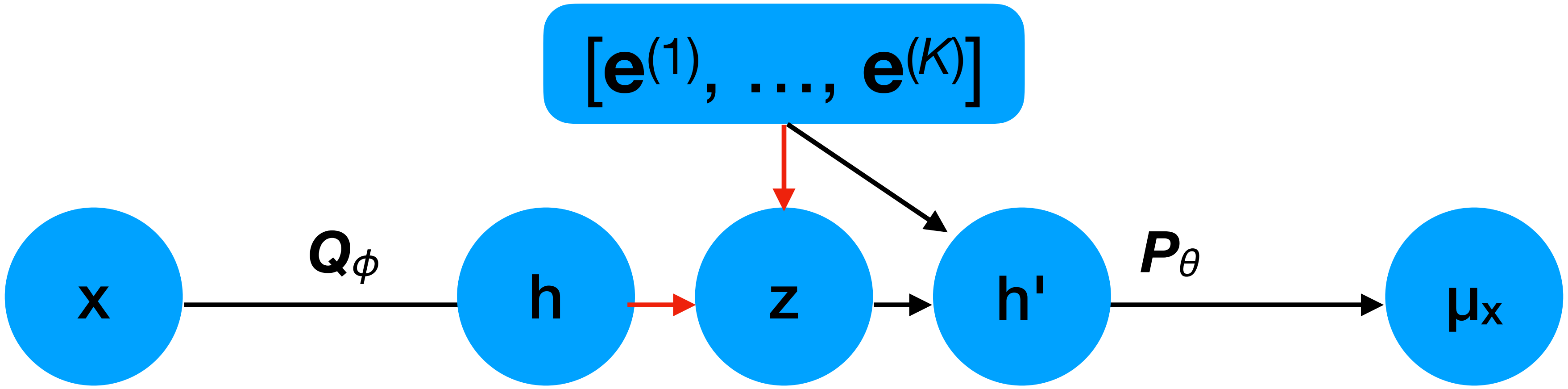
*The Gumbel softmax is sometimes used but is more complicated and not clearly better than the straight-through gradient estimator.

Reconstruction probability

$$-D_{\mathrm{KL}}(Q_\phi(z \mid \mathbf{x}) \mid P(z)) + \mathbb{E}_{Q_\phi}[\log P(\mathbf{x} \mid z)]$$

- The concrete issue is that the path through the graph to ϕ is blocked due to non-differentiability:

$$\frac{\partial \log P(\mathbf{x} \mid z)}{\partial \mathbf{h}'} \frac{\partial \mathbf{h}'}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \phi}$$



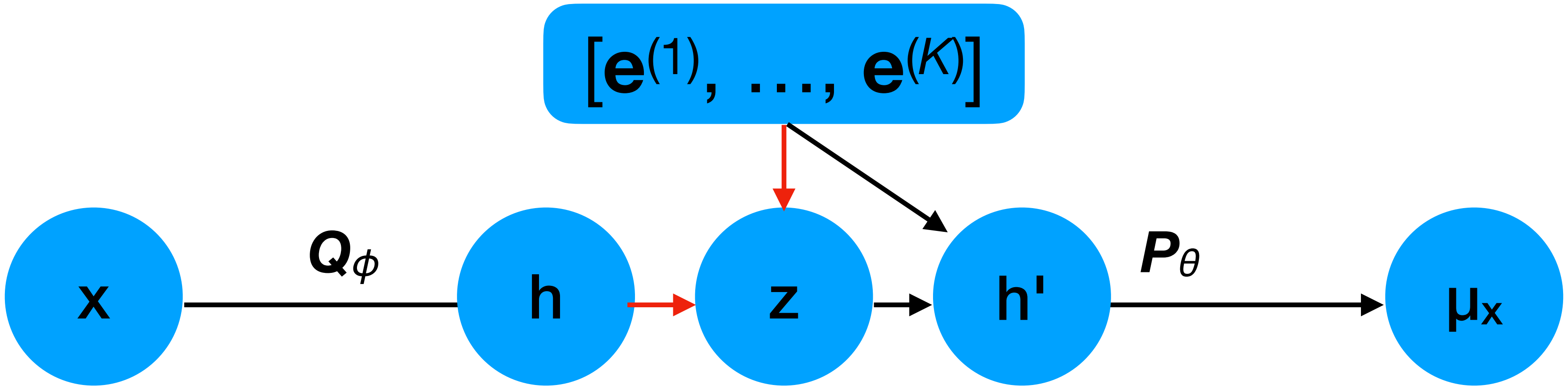
VQVAE: Loss function

- For this reason, we have to “give up” on a first-principles approach to maximizing $\mathbb{E}_{Q_\phi}[P(\mathbf{x} \mid z)]$ for VQ-VAEs.

- Instead, we hand-design a loss function with the goals:

   1. Encourage each $\mathbf{h}^{(i)'}$ to give a high $\log P(\mathbf{x} \,|\, z)$.

$$f_{\text{loss}}(\phi, \theta, \mathbf{E}) = \|\mathbf{x} - \mu_{\mathbf{x}}\|^2$$



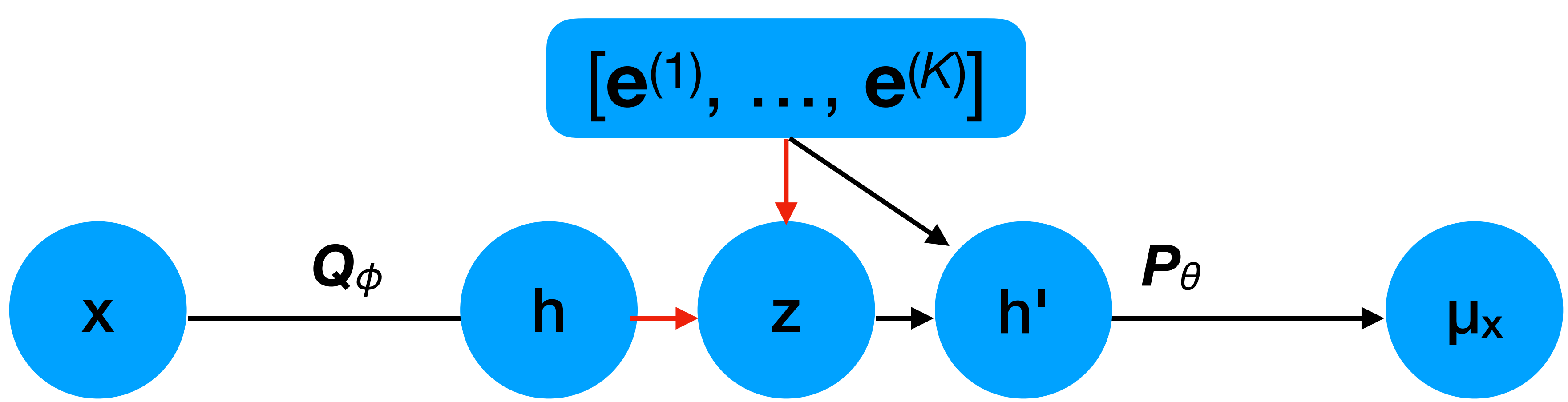
VQVAE: Loss function

- For this reason, we have to "give up" on a first-principles approach to maximizing $\mathbb{E}_{Q_\phi}[P(\mathbf{x} \mid z)]$ for VQ-VAEs.

- Instead, we hand-design a loss function with the goals:

  1. Encourage each $\mathbf{h}^{(i)'}$ to give a high $\log P(\mathbf{x} \,|\, z)$.
  2. Encourage the $\mathbf{e}^{(1)}, \ldots, \mathbf{e}^{(K)}$ to be the centroids of K clusters among the $\{\mathbf{h}_j^{(i)}\}_{i,j}$. Pick centroids to represent the data.

$$f_{\text{loss}}(\phi, \theta, \mathbf{E}) = \|\mathbf{x} - \mu_{\mathbf{x}}\|^2 + \|\mathbf{h} - \mathbf{e}^{(z)}\|^2$$



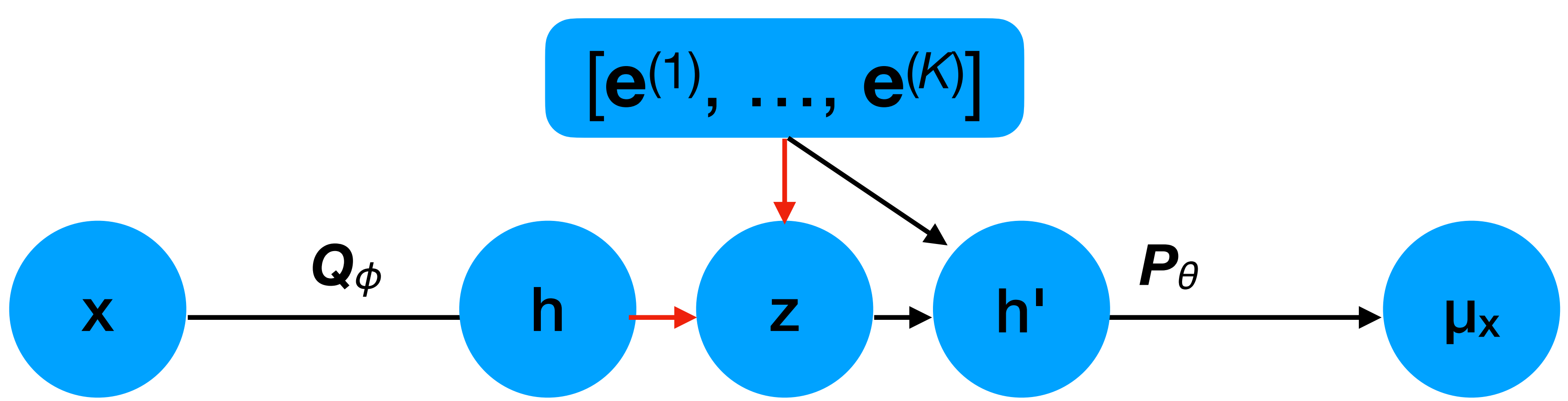
VQVAE: Loss function

- For this reason, we have to "give up" on a first-principles approach to maximizing $\mathbb{E}_{Q_\phi}[P(\mathbf{x} \mid z)]$ for VQ-VAEs.

- Instead, we hand-design a loss function with the goals:
  1. Encourage each $\mathbf{h}^{(i)'}$ to give a high $\log P(\mathbf{x}\,|\,z)$.
  2. Encourage the $\mathbf{e}^{(1)}, \ldots, \mathbf{e}^{(K)}$ to be the centroids of K clusters among the $\{\mathbf{h}_j^{(i)}\}_{i,j}$.
  3. Encourage each $\mathbf{h}^{(i)}$ to be "close" to its nearest centroid $\mathbf{e}^{(z)}$.

Discourage { $\mathbf{h}^{(i)}$ } from growing too large.

$$f_{\text{loss}}(\phi, \theta, \mathbf{E}) = \|\mathbf{x} - \mu_{\mathbf{x}}\|^2 + \|\mathbf{h} - \mathbf{e}^{(z)}\|^2$$

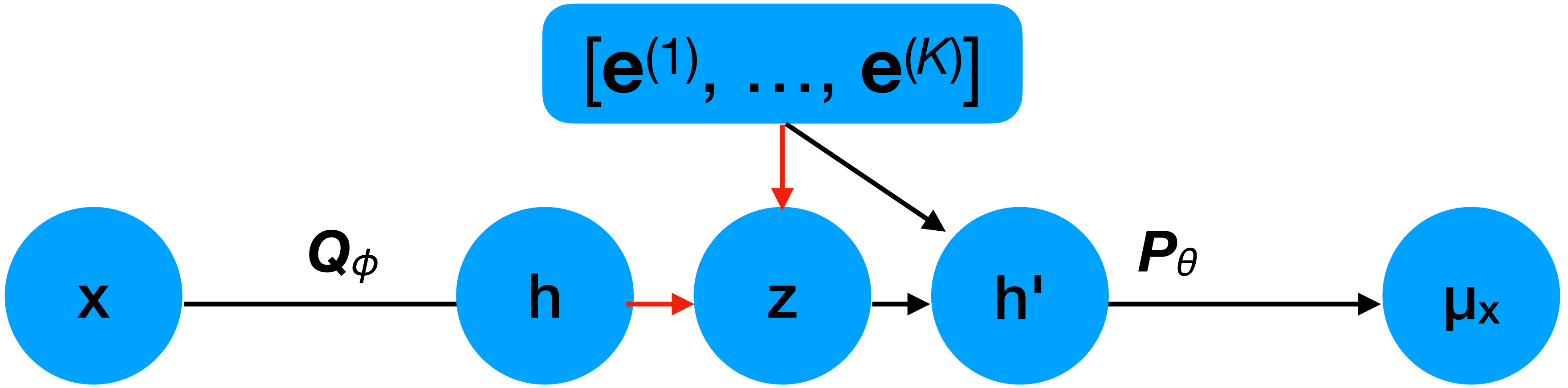


VQVAE: Loss function

- We can express the gradient w.r.t. ϕ as a product-of-Jacobians:

$$\frac{\partial \log P(\mathbf{x} \mid z)}{\partial \mathbf{h}'} \frac{\partial \mathbf{h}'}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \phi}$$

$$f_{\text{loss}}(\phi, \theta, \mathbf{E}) = \|\mathbf{x} - \mu_{\mathbf{x}}\|^2 + \|\mathbf{h} - \mathbf{e}^{(z)}\|^2$$



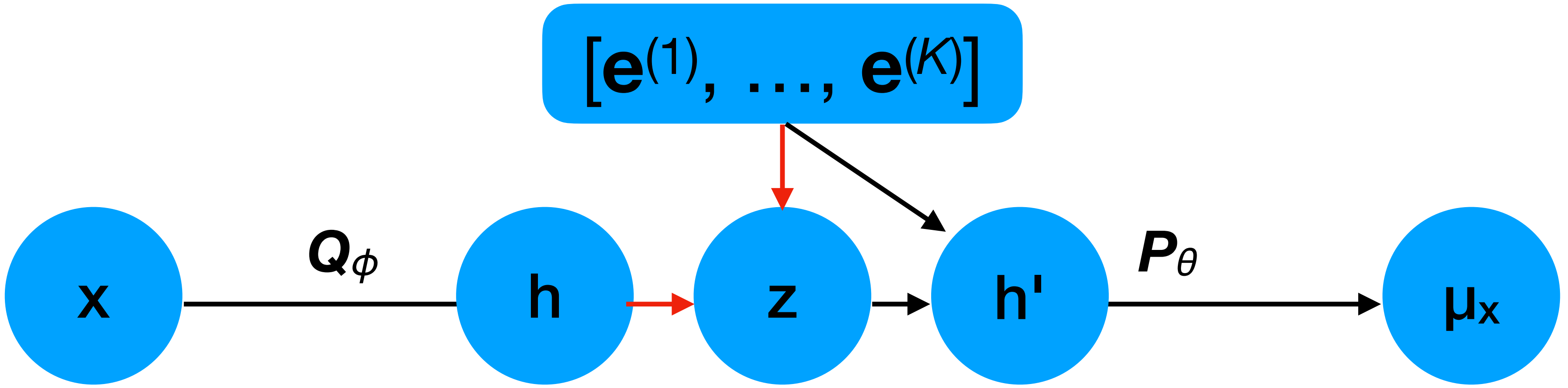
VQVAE: Loss function

- We can express the gradient w.r.t. ϕ as a product-of-Jacobians:

$$\frac{\partial \log P(\mathbf{x} \mid z)}{\partial \mathbf{h}'} \frac{\partial \mathbf{h}'}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \phi}$$

How a change in **h**' affects reconstruction error

$$f_{\text{loss}}(\phi, \theta, \mathbf{E}) = \|\mathbf{x} - \mu_{\mathbf{x}}\|^2 + \|\mathbf{h} - \mathbf{e}^{(z)}\|^2$$



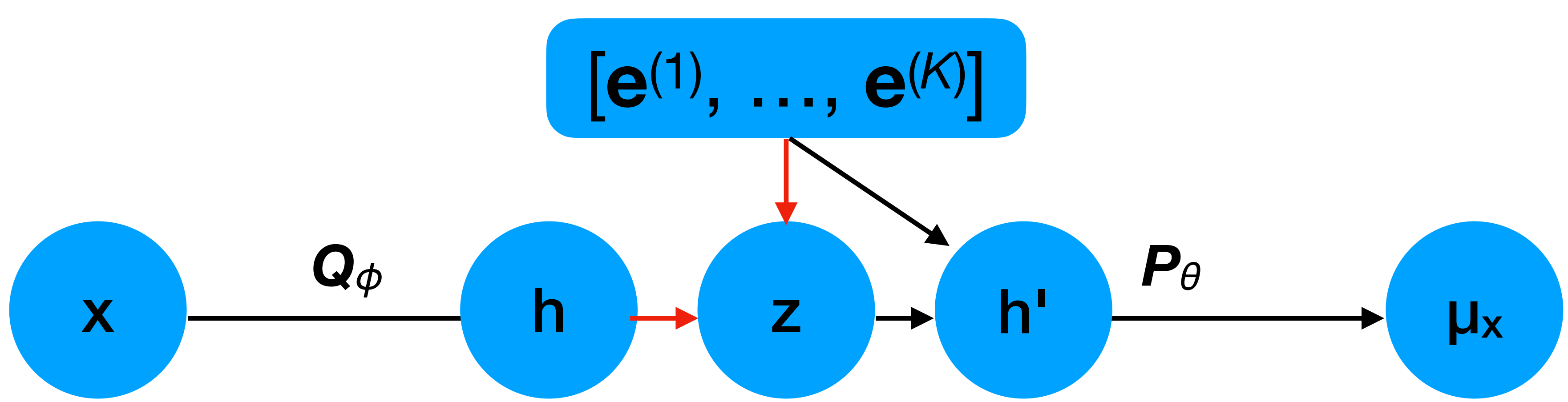
VQVAE: Loss function

- We can express the gradient w.r.t. ϕ as a product-of-Jacobians:

$$\frac{\partial \log P(\mathbf{x} \mid z)}{\partial \mathbf{h}'} \frac{\partial \mathbf{h}'}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \phi}$$

How a change in **h**
affects **h**'

$$f_{\text{loss}}(\phi, \theta, \mathbf{E}) = \|\mathbf{x} - \mu_{\mathbf{x}}\|^2 + \|\mathbf{h} - \mathbf{e}^{(z)}\|^2$$



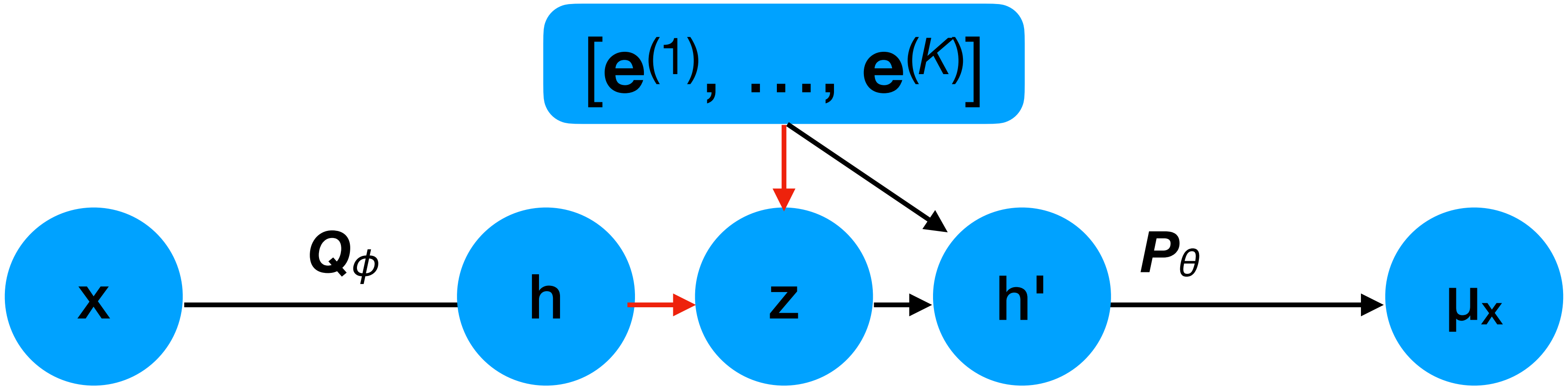
VQVAE: Loss function

- We can express the gradient w.r.t. ϕ as a product-of-Jacobians:

$$\frac{\partial \log P(\mathbf{x} \mid z)}{\partial \mathbf{h}'} \frac{\partial \mathbf{h}'}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \phi}$$

How the parameters of Q affect **h**

$$f_{\text{loss}}(\phi, \theta, \mathbf{E}) = \|\mathbf{x} - \mu_{\mathbf{x}}\|^2 + \|\mathbf{h} - \mathbf{e}^{(z)}\|^2$$

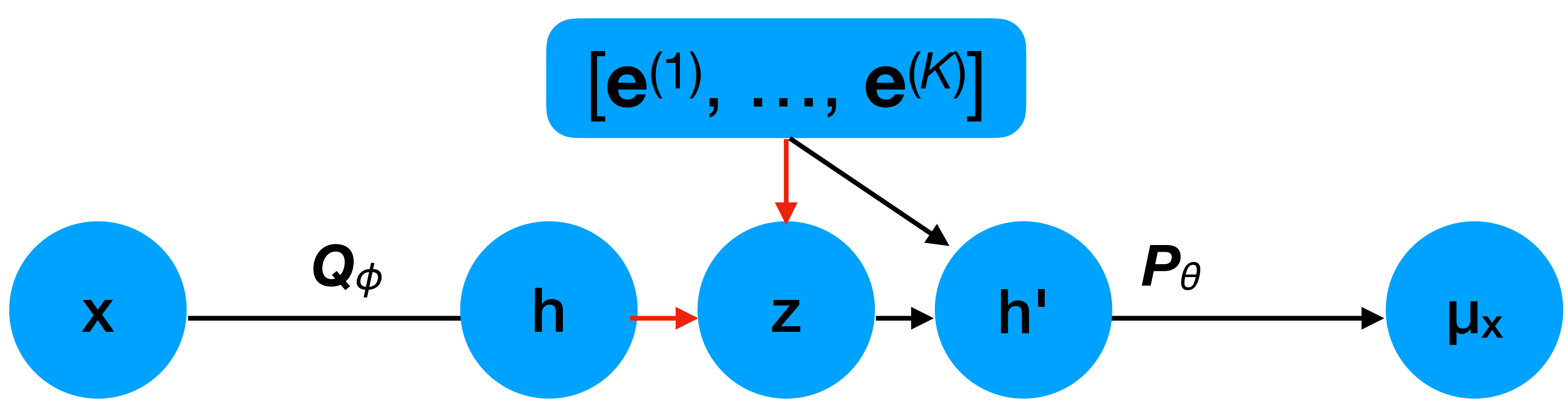


VQVAE: Loss function

- But what about the non-differentiable map from **h** to **h**'?

$$\frac{\partial \log P(\mathbf{x} \mid z)}{\partial \mathbf{h}'} \frac{\partial \mathbf{h}'}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \phi}$$

$$f_{\text{loss}}(\phi, \theta, \mathbf{E}) = \|\mathbf{x} - \mu_{\mathbf{x}}\|^2 + \|\mathbf{h} - \mathbf{e}^{(z)}\|^2$$



VQVAE: Loss function

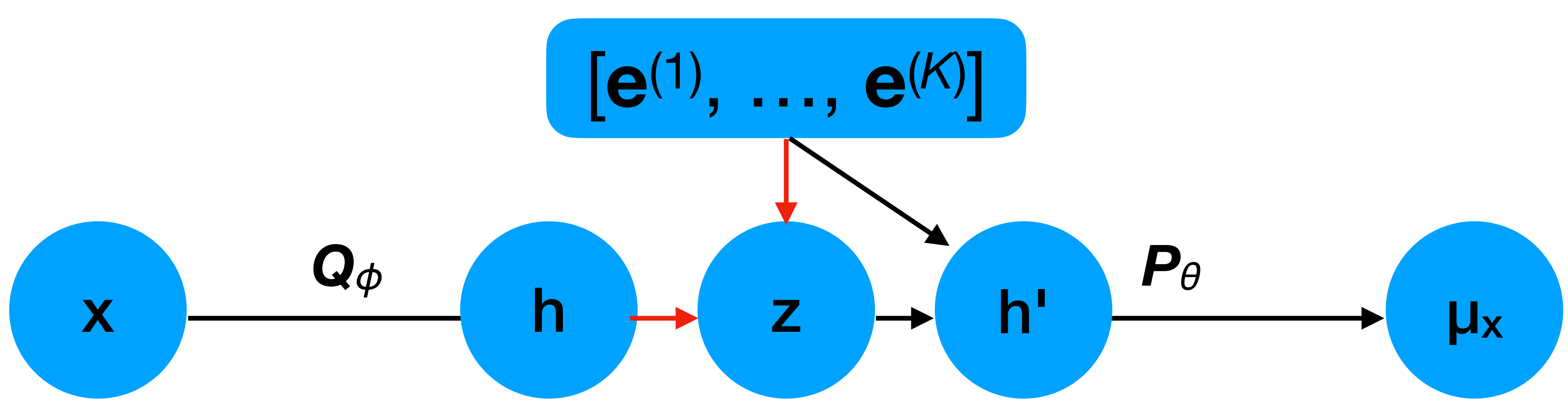
- But what about the non-differentiable map from **h** to **h**'?

$$\frac{\partial \log P(\mathbf{x} \mid z)}{\partial \mathbf{h}'} \mathbf{I} \frac{\partial \mathbf{h}}{\partial \phi}$$

- **Straight-through gradient estimator**: we just ignore it! I.e., we set it to **I**.

- Why is this reasonable? My interpretation: a small change in **h** *does* pull **h**' toward h (though scaled by inverse cluster size).

$$f_{\text{loss}}(\phi, \theta, \mathbf{E}) = \|\mathbf{x} - \mu_{\mathbf{x}}\|^2 + \|\mathbf{h} - \mathbf{e}^{(z)}\|^2$$



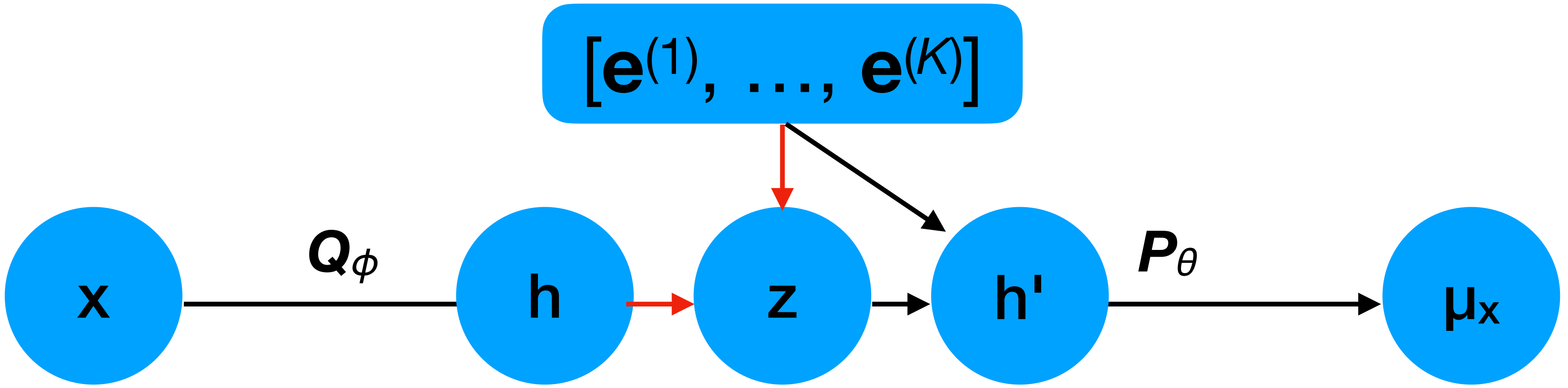
VQVAE: Loss function

- Note that the second loss term has non-zero gradient w.r.t. both **h** (the embedding of example **x**) and $\mathbf{e}^{(z)}$ (**x**'s assigned centroid):

$$\frac{\partial}{\partial \mathbf{h}} \|\mathbf{h} - \mathbf{e}^{(z)}\|^2 = 2\|\mathbf{h} - \mathbf{e}^{(z)}\|$$

$$\frac{\partial}{\partial \mathbf{e}^{(z)}} \|\mathbf{h} - \mathbf{e}^{(z)}\|^2 = 2\|\mathbf{h} - \mathbf{e}^{(z)}\|$$

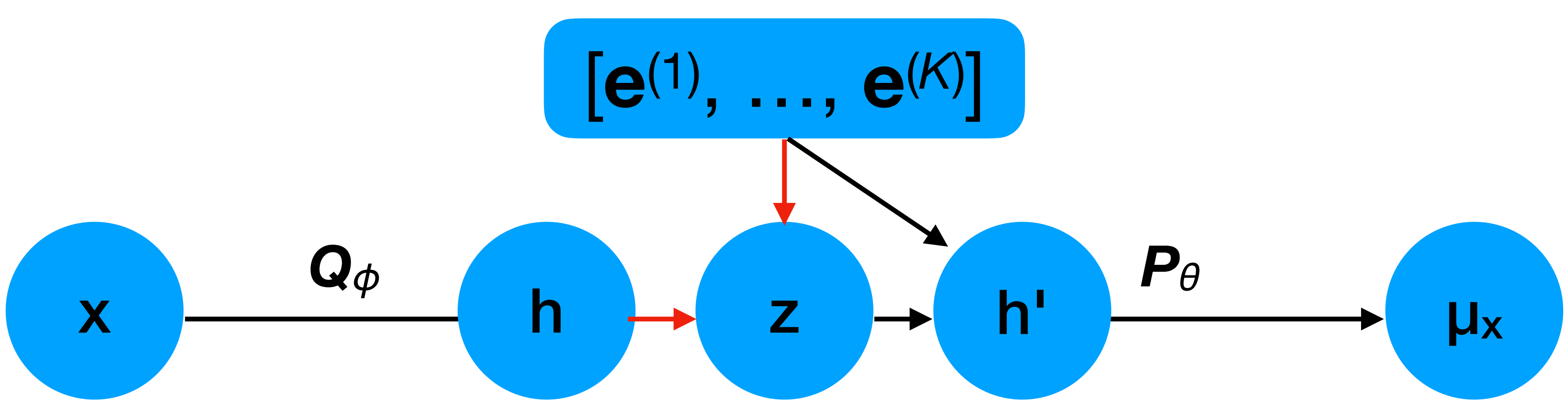
$$f_{\text{loss}}(\phi, \theta, \mathbf{E}) = \|\mathbf{x} - \mu_{\mathbf{x}}\|^2 + \boxed{\|\mathbf{h} - \mathbf{e}^{(z)}\|^2}$$



VQVAE: Loss function

- However, to get more flexibility in how we calculate and use these gradients, we typically split this term into 2 terms:
- Here, **sg** stands for "stop gradient" and prevents gradient for the variable in () from being calculated.

$$f_{\text{loss}}(\phi, \theta, \mathbf{E}) = \|\mathbf{x} - \mu_{\mathbf{x}}\|^2 + \boxed{\|\mathbf{h} - \text{sg}(\mathbf{e}^{(z)})\|^2} + \boxed{\|\text{sg}(\mathbf{h}) - \mathbf{e}^{(z)}\|^2}$$



VQVAE: Loss function

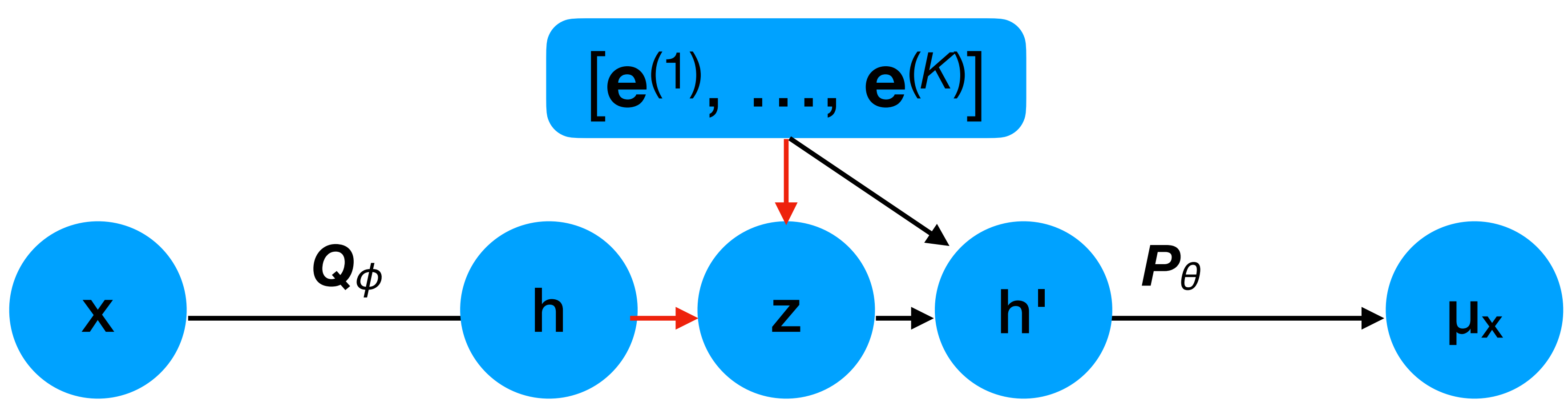
- Summing over both of the new terms, we still obtain the gradient expressions below.

- But now we can weight each loss term and do fancier things as well (e.g., exponential moving average).

$$\frac{\partial}{\partial \mathbf{h}} \|\mathbf{h} - \mathbf{e}^{(z)}\|^2 = 2\|\mathbf{h} - \mathbf{e}^{(z)}\|$$

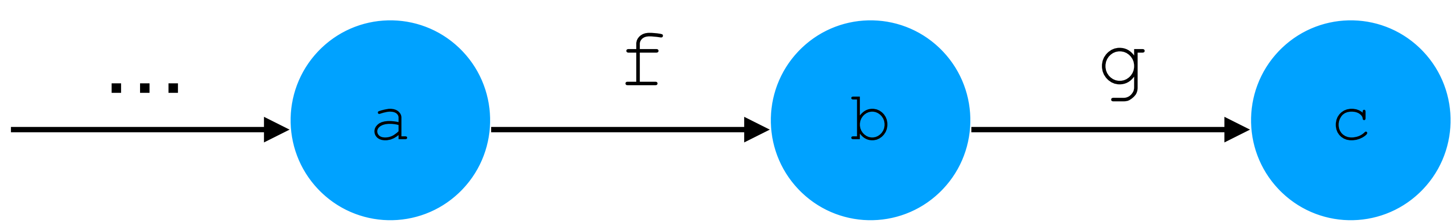
$$\frac{\partial}{\partial \mathbf{e}^{(z)}} \|\mathbf{h} - \mathbf{e}^{(z)}\|^2 = 2\|\mathbf{h} - \mathbf{e}^{(z)}\|$$

$$f_{\text{loss}}(\phi, \theta, \mathbf{E}) = \|\mathbf{x} - \mu_{\mathbf{x}}\|^2 + \boxed{\|\mathbf{h} - \text{sg}(\mathbf{e}^{(z)})\|^2} + \boxed{\|\text{sg}(\mathbf{h}) - \mathbf{e}^{(z)}\|^2}$$



Implementing VQ-VAE in code

- PyTorch's autograd keeps track of the *value* as well as the *derivative* of each node in the graph.
- We can implement a sg operation using the `.detach()` method.



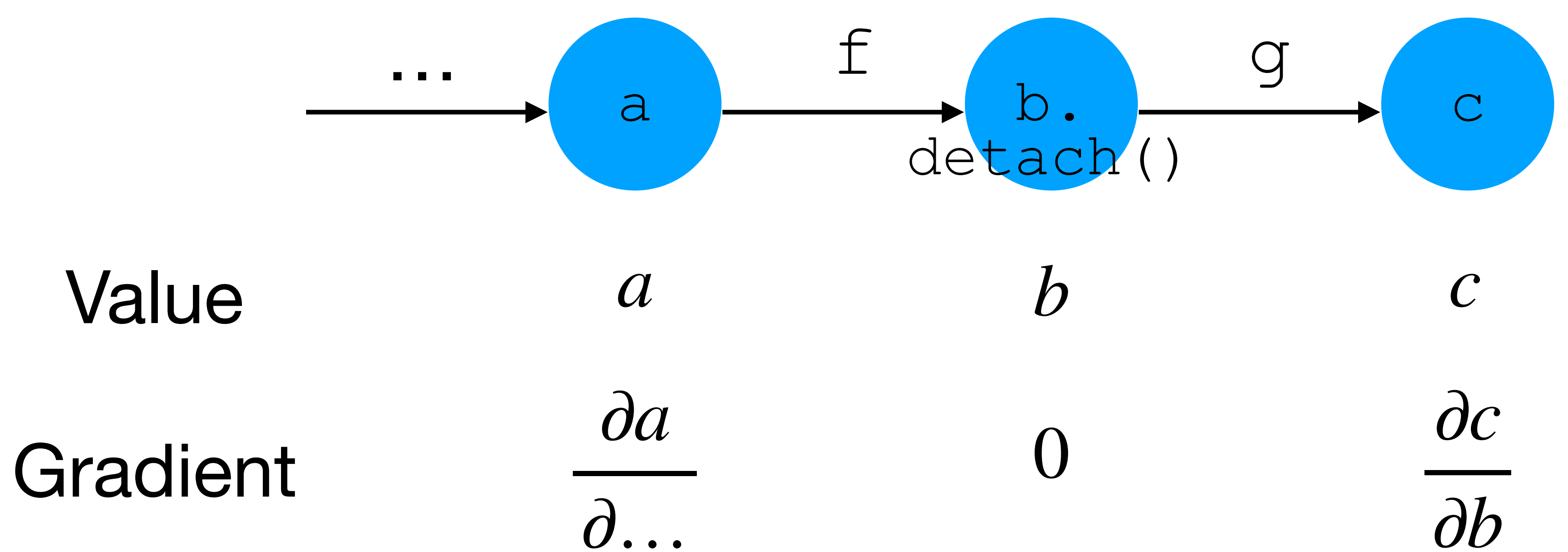
Value	a	b	c
Gradient	$\frac{\partial a}{\partial \ldots}$	$\frac{\partial b}{\partial a}$	$\frac{\partial c}{\partial b}$

Implementing VQ-VAE in code

- PyTorch's autograd keeps track of the *value* as well as the *derivative* of each node in the graph.

- We can implement a sg operation using the `.detach()` method.

- For a node `b`, the call `b.detach()` preserves `b`'s *value* but zeros its *derivative*.

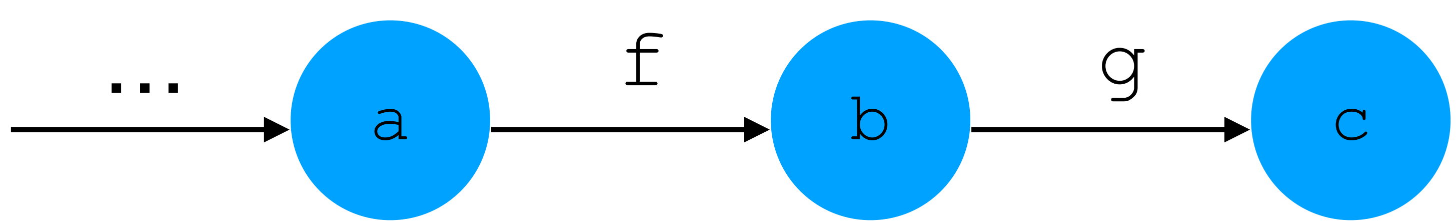


Implementing VQ-VAE in code

- Consider:

```
b = f(a)
c = g(b)
```

- Here, we can back-prop from `c` to `a` with $\dfrac{\partial c}{\partial a} = \dfrac{\partial c}{\partial b}\dfrac{\partial b}{\partial a}$.



Value	a	b	c
Gradient	$\dfrac{\partial a}{\partial \ldots}$	$\dfrac{\partial b}{\partial a}$	$\dfrac{\partial c}{\partial b}$

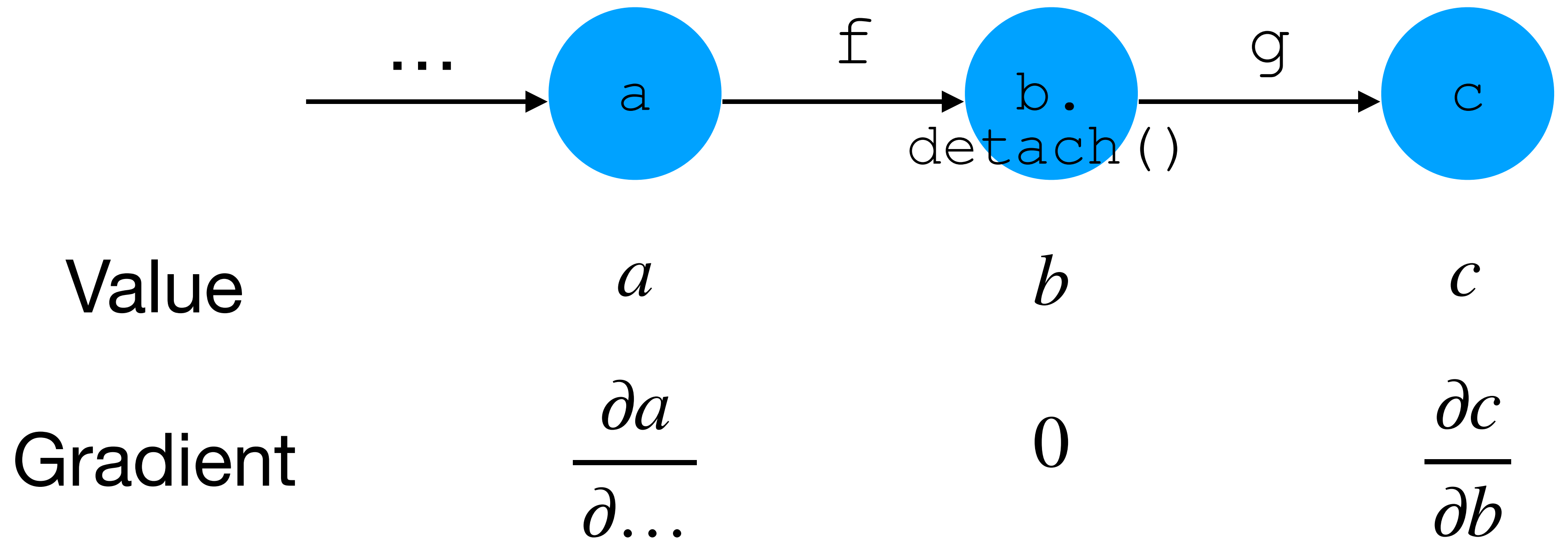
Implementing VQ-VAE in code

- But what if we call `b.detach()`:

```
b = f(a)
c = g(b.detach())
```

- We can still forward-prop from `a` to `c`.

- However, since $\frac{\partial b}{\partial a} = 0$ (according to autograd), we cannot back-prop through `f` to compute $\frac{\partial c}{\partial a}$.



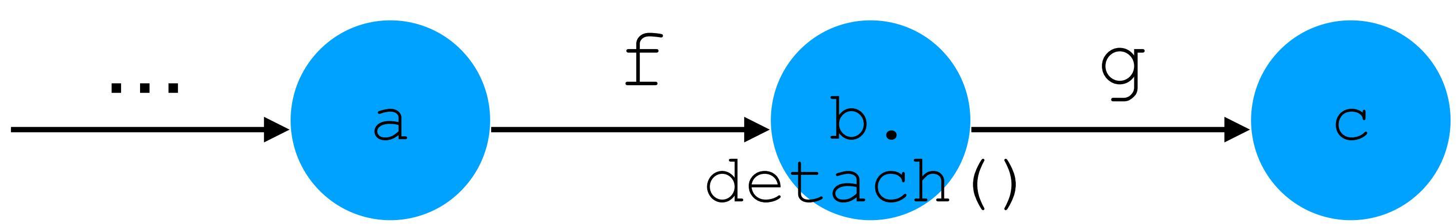
Implementing VQ-VAE in code

- But what if we call `b.detach()`:

```
b = f(a)
c = g(b.detach())
```

- We can still forward-prop from `a` to `c`.

- Hence, any parameters in `f` (or earlier in the graph) will *not* get updated in SGD.



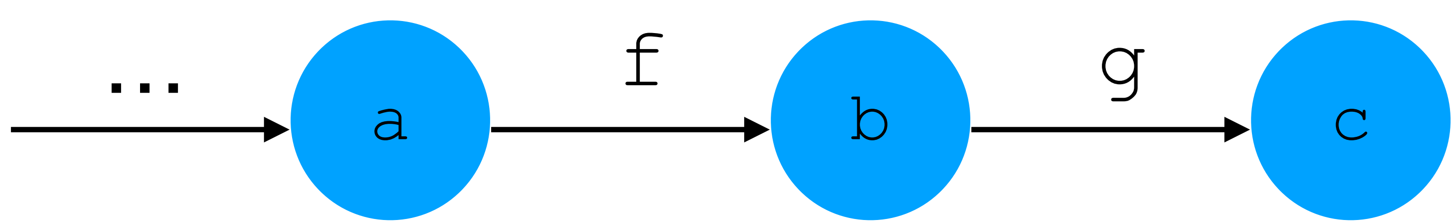
Value	a	b	c
Gradient	$\frac{\partial a}{\partial \ldots}$	0	$\frac{\partial c}{\partial b}$

Implementing VQ-VAE in code

- If `f` is non-differentiable and we want to“skip over” its Jacobian $\frac{\partial b}{\partial a}$, we can write:

```
b = f(a)
c = g(b.detach() + a - a.detach())
```

- Expression `a` will generate both its value (`a`) and its derivative ($\frac{\partial a}{\partial \ldots}$).



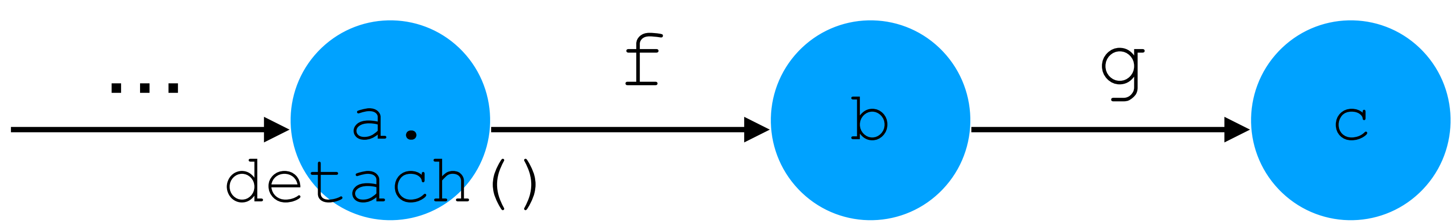
Value	a	b	c
Gradient	$\frac{\partial a}{\partial \ldots}$	$\frac{\partial b}{\partial a}$	$\frac{\partial c}{\partial b}$

Implementing VQ-VAE in code

- If `f` is non-differentiable and we want to“skip over” its Jacobian $\frac{\partial b}{\partial a}$, we can write:

```
b = f(a)
c = g(b.detach() + a - a.detach())
```

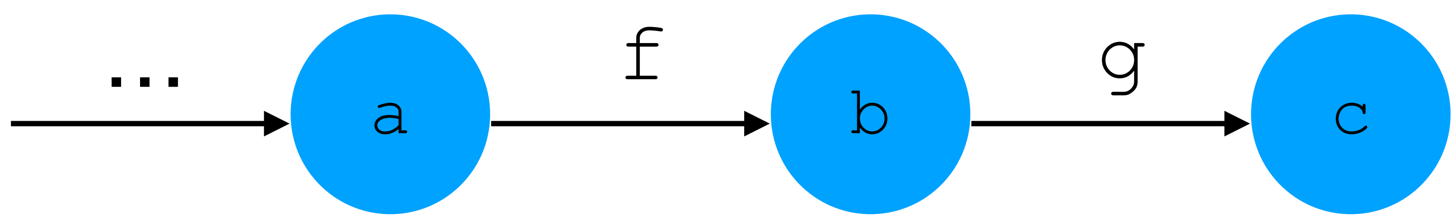
- Expression `a.detach()` will generate both its value (`a`) but 0 derivative.



Value	a	b	c
Gradient	0	$\frac{\partial b}{\partial a}$	$\frac{\partial c}{\partial b}$

Implementing VQ-VAE in code

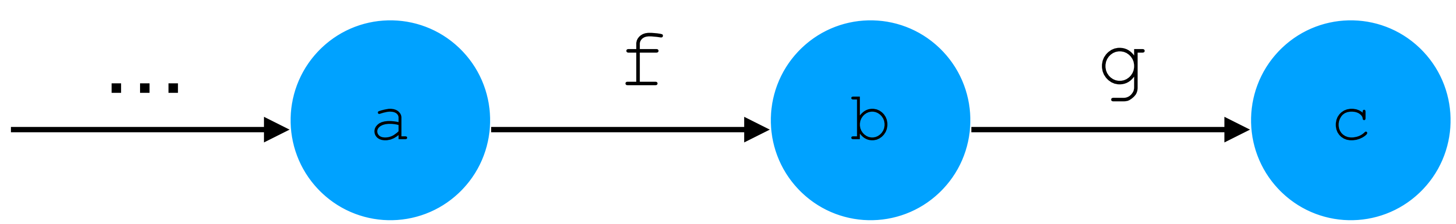
- `b.detach() + a - a.detach()`:
  - Value: `b + a - a = b`
  - Derivative: $0 + \dfrac{\partial a}{\partial \ldots} - 0 = \dfrac{\partial a}{\partial \ldots}$



Value	a	b	c
Gradient	$\dfrac{\partial a}{\partial \ldots}$	$\dfrac{\partial b}{\partial a}$	$\dfrac{\partial c}{\partial b}$

Implementing VQ-VAE in code

- `g(b.detach() + a - a.detach())`:
    - Value: `g(b)`
    - Derivative: $\frac{\partial c}{\partial b} \frac{\partial a}{\partial \ldots}$



Value	a	b	c
Gradient	$\frac{\partial a}{\partial \ldots}$	$\frac{\partial b}{\partial a}$	$\frac{\partial c}{\partial b}$

Implementing VQ-VAE in code

- Hence, to compute the reconstruction loss term:

$$f_{\text{loss}}(\phi, \theta, \mathbf{E}) = \boxed{\|\mathbf{x} - \mu_{\mathbf{x}}\|^2} + \|\mathbf{h} - \text{sg}(\mathbf{e}^{(z)})\|^2 + \|\text{sg}(\mathbf{h}) - \mathbf{e}^{(z)}\|^2$$

with the straight-through gradient estimator, we can write:

```
h = encoder(x)
hprime = quantize(h)
mu_x = decoder(hprime.detach() + h - h.detach())
```

Implementing VQ-VAE in code

- For the other two loss terms,

$$f_{\text{loss}}(\phi, \theta, \mathbf{E}) = \|\mathbf{x} - \mu_{\mathbf{x}}\|^2 + \|\mathbf{h} - \text{sg}(\mathbf{e}^{(z)})\|^2 + \|\text{sg}(\mathbf{h}) - \mathbf{e}^{(z)}\|^2$$

  we can write:

```
mse(h, hprime.detach())
```

  and:

```
mse(h.detach(), hprime)
```

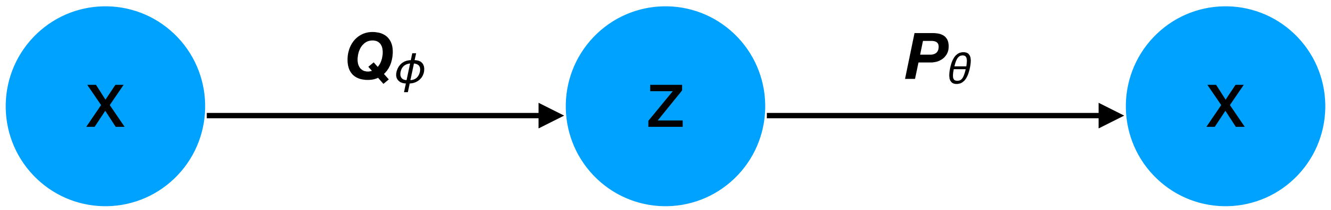
Generative Adversarial Networks (GANs)

Generative models

- So far, the generative models we have discussed were latent variable models (LVMs).

- In these cases, we can train the model as the combination of an encoder *Q* and decoder *P* with a **single optimization objective** of maximizing the ELBO:

$$-D_{\mathrm{KL}}(Q_\phi(z \mid \mathbf{x}) \mid P(z)) + \mathbb{E}_{Q_\phi}[\log P(\mathbf{x} \mid z)]$$



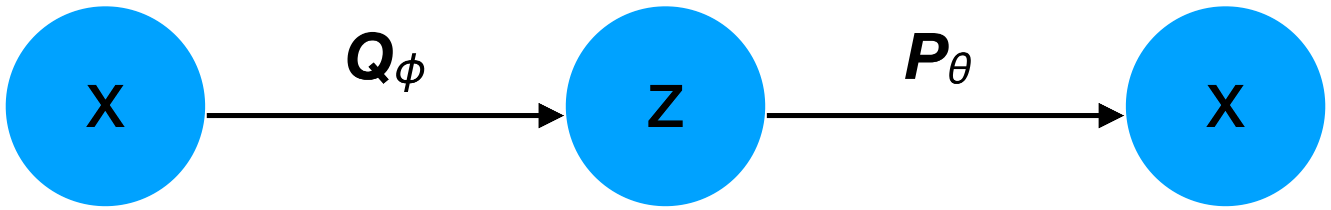
Generative models

- So far, the generative models we have discussed were latent variable models (LVMs).

- In these cases, we can train the model as the combination of an encoder *Q* and decoder *P* with a **single optimization objective** of maximizing the ELBO:

$$-D_{\mathrm{KL}}(Q_\phi(z \mid \mathbf{x}) \mid P(z)) + \mathbb{E}_{Q_\phi}[\log P(\mathbf{x} \mid z)]$$

- In other words, the encoder and decoder are working **cooperatively** to maximize the data log-likelihood.



Generative Adverarial Networks (GANs)

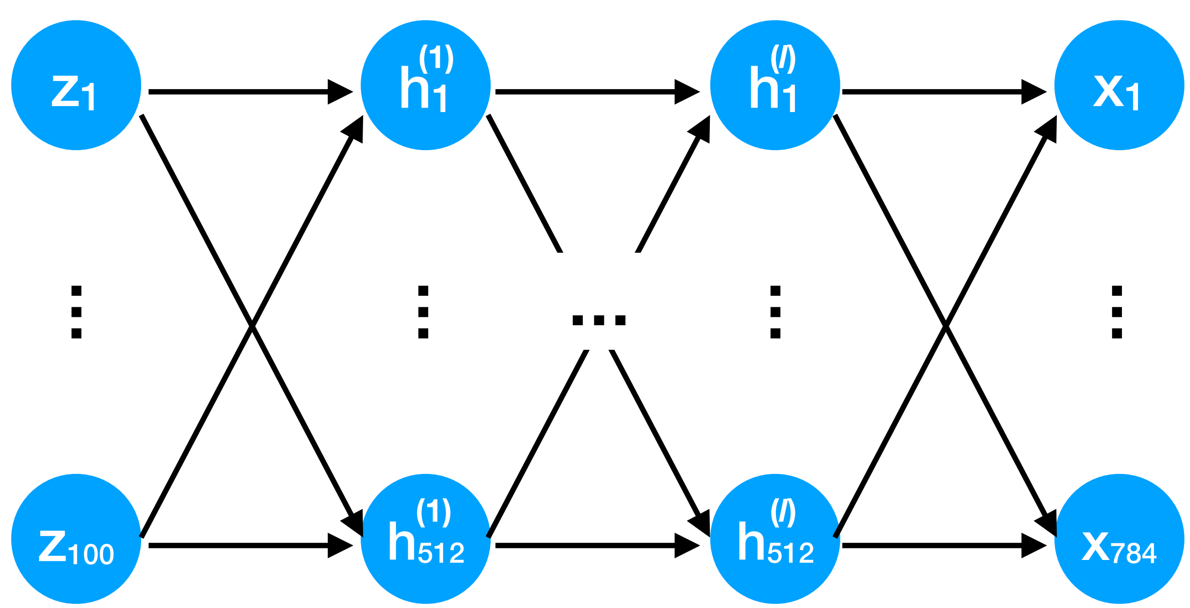
- However, another entire class of deep learning methods is based on training two networks that **compete against each other** in a zero-sum game.
- This idea is the basis for the **Generative Adversarial Network** (**GAN**; Goodfellow et al. 2014).

Generative Adverarial Networks (GANs)

- Like VAEs, GANs consist of two components, but their semantics are different.
- Let $P_{\text{data}}(\mathbf{x})$ be the ground-truth data distribution.
- **Generator *G***: given a noise vector **z** from an easy-to-sample distribution (e.g., Gaussian, uniform), generate a vector **x** that looks like it came from $P_{\text{data}}(\mathbf{x})$.
- **Discriminator *D***: given a vector **x**, decide if it is real ($\hat{y}$=1) or fake ($\hat{y}$=0). *D* acts as a “forgery detector”.

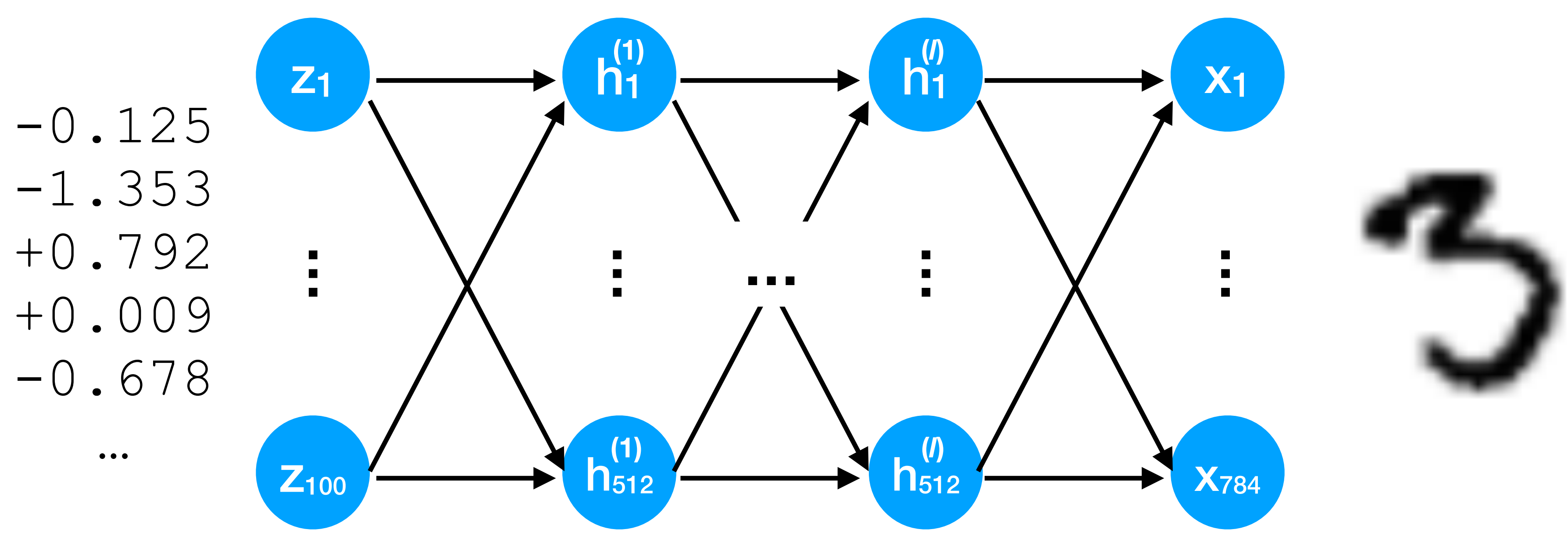
Generator *G*

- Example *G* with *l* hidden layers that generates an MNIST image (28x28=784) **x** from a 100-dim noise vector **z:**



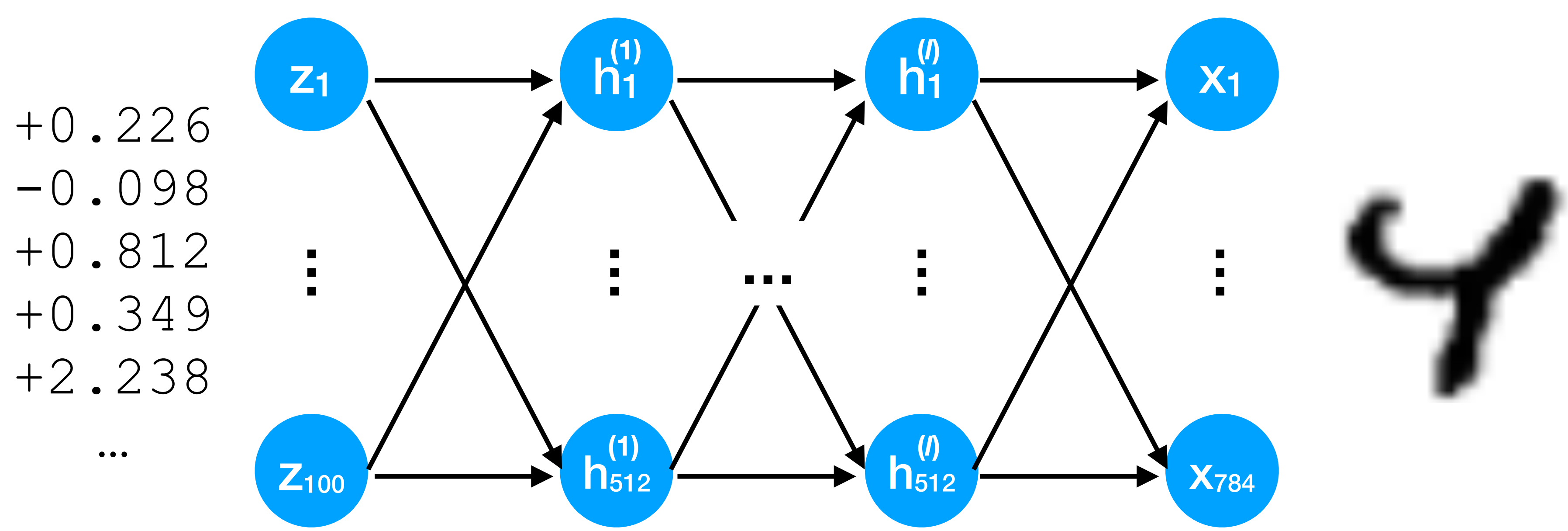
Generator G

- By feeding different noise vectors **z**, we obtain different **x**.
- Implicitly, **z** encodes the different dimensions of variability of $P_{\text{data}}(\mathbf{x})$ (though they may not be intuitive, independent, or **disentangled**).



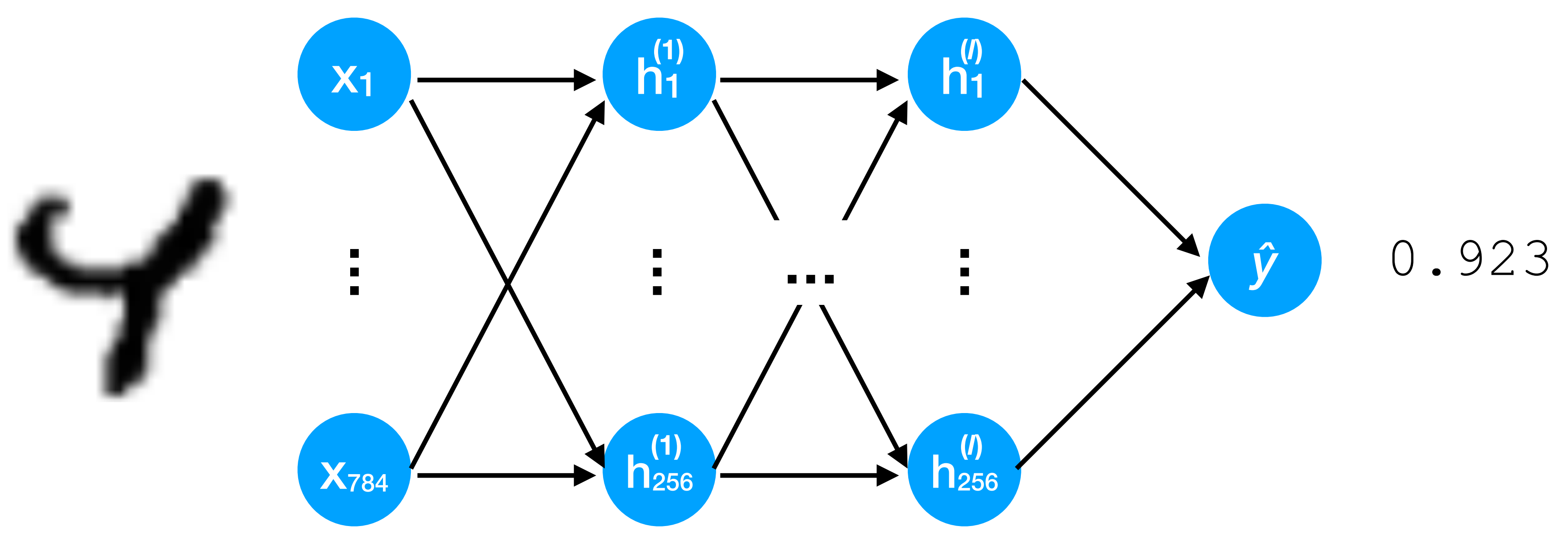
Generator G

- By feeding different noise vectors **z**, we obtain different **x**.
- Implicitly, **z** encodes the different dimensions of variability of $P_{\text{data}}(\mathbf{x})$ (though they may not be intuitive, independent, or **disentangled**).



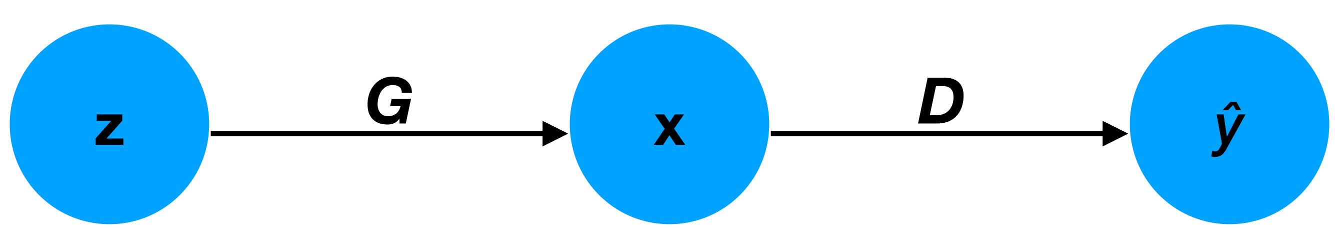
Discriminator *D*

- Example *D* with *l* hidden layers that estimates $\hat{y} \in (0,1)$ that expresses probability that the input **x** is real:



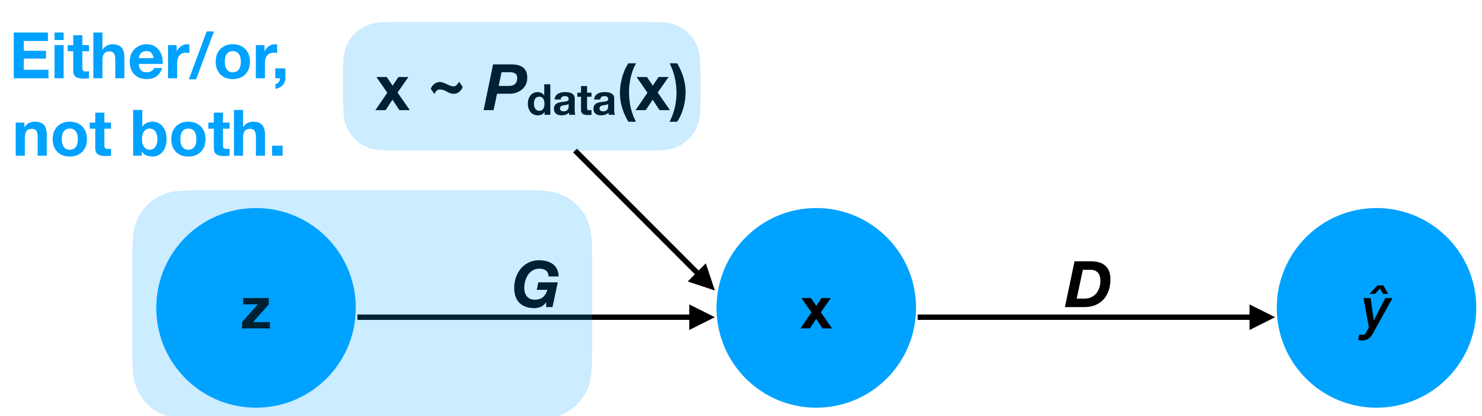
Generative Adverarial Networks (GANs)

- Like VAEs, GANs are trained such that one component “feeds” to the other.



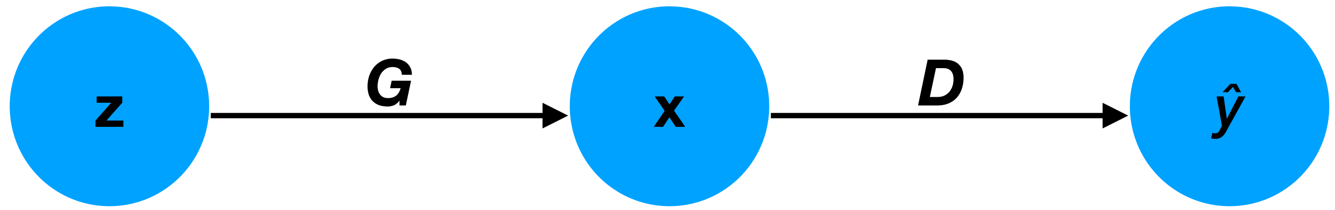
Generative Adverarial Networks (GANs)

- Like VAEs, GANs are trained such that one component "feeds" to the other.
- In contrast to VAEs, the discriminator *D* is sometimes given a "fake" data vector **x** (generated by *G*), and sometimes given a "real" vector x sampled from the training set (which approximates $P_{\text{data}}(\mathbf{x})$).



Generative Adverarial Networks (GANs)

- Each network has its own parameters:
  - G has parameters θ_G.
  - D has parameters θ_D.

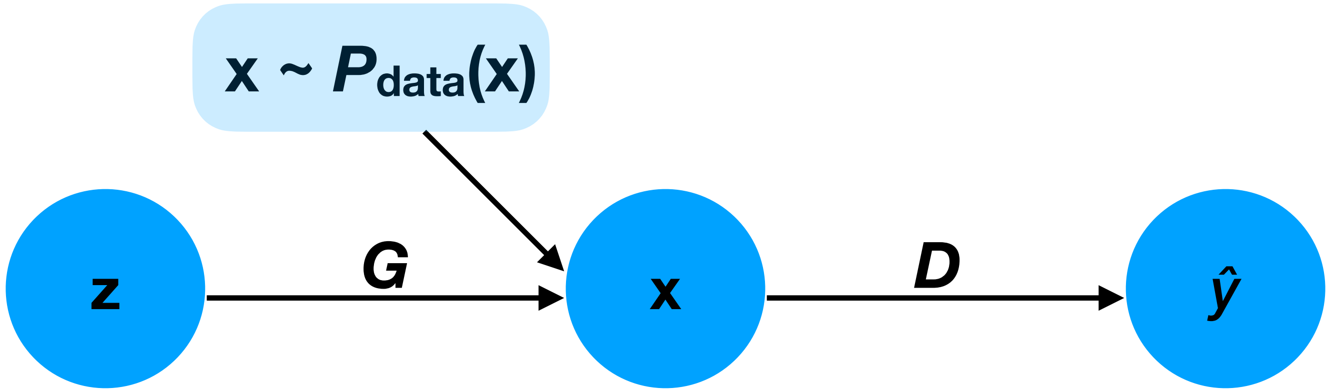


Generative Adverarial Networks (GANs)

- We can define the following loss on how well *D* can discriminate fake from real data:

$$f_{\text{acc}}(\theta_G, \theta_D) = \mathbb{E}_{\mathbf{x}\sim P_{\text{data}}(\mathbf{x})}[\log D_{\theta_D}(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim P(\mathbf{z})}[\log(1 - D_{\theta_D}(G_{\theta_G}(\mathbf{z})))]$$

Log-likelihood that *D* recognizes real data as real.

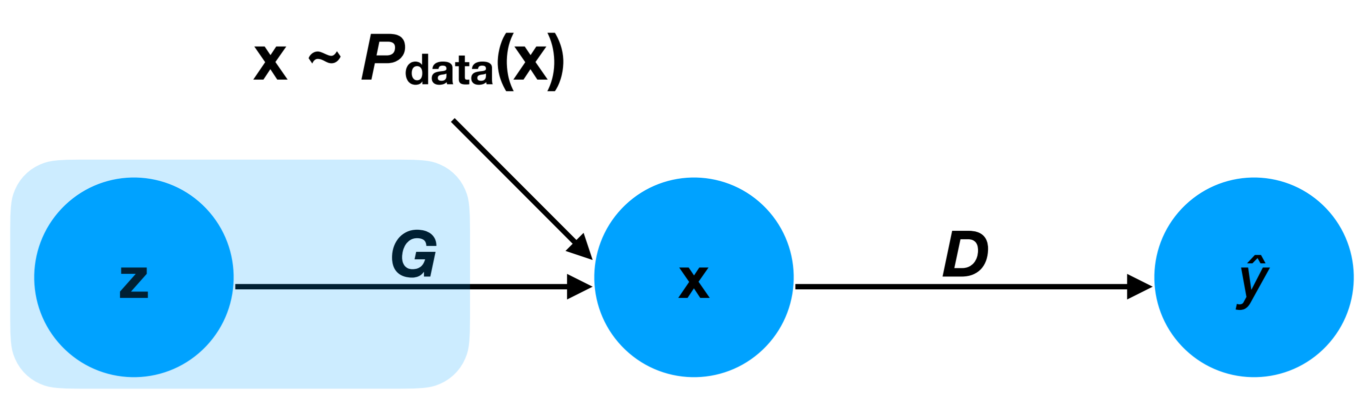


Generative Adverarial Networks (GANs)

- We can define the following loss on how well D can discriminate fake from real data:

$$f_{\text{acc}}(\theta_G, \theta_D) = \mathbb{E}_{\mathbf{x} \sim P_{\text{data}}(\mathbf{x})}[\log D_{\theta_D}(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim P(\mathbf{z})}[\log(1 - D_{\theta_D}(G_{\theta_G}(\mathbf{z})))]$$

Log-likelihood that *D* recognizes fake data as fake.



Generative Adverarial Networks (GANs)

- The goal of *D* is to *maximize* f_{acc}, whereas the goal of *G* is to *minimize* f_{acc}.

$$f_{\text{acc}}(\theta_G, \theta_D) = \mathbb{E}_{\mathbf{x} \sim P_{\text{data}}(\mathbf{x})}[\log D_{\theta_D}(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim P(\mathbf{z})}[\log(1 - D_{\theta_D}(G_{\theta_G}(\mathbf{z})))]$$

- This two-player game will reach an equilibrium if we find:

$$\min_{\theta_G} \max_{\theta_D} f_{\text{acc}}(\theta_G, \theta_D)$$

- In particular, this solution corresponds to *D* having 50% accuracy at detecting forgeries, and *G* generating fake **x** according to $P_{\text{data}}(\mathbf{x})$.

Training GANs

$$f_{\text{acc}}(\theta_G, \theta_D) = \mathbb{E}_{\mathbf{x} \sim P_{\text{data}}(\mathbf{x})}[\log D_{\theta_D}(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim P(\mathbf{z})}[\log(1 - D_{\theta_D}(G_{\theta_G}(\mathbf{z})))]$$

- In practice, we train *D* and *G* *iteratively*:
  - Freeze *G*, and perform SGD on *D* for *k* iterations to increase f_{acc}.

Training GANs

$$f_{\text{acc}}(\theta_G, \theta_D) = \mathbb{E}_{\mathbf{x}\sim P_{\text{data}}(\mathbf{x})}[\log D_{\theta_D}(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim P(\mathbf{z})}[\log(1 - D_{\theta_D}(G_{\theta_G}(\mathbf{z})))]$$

- In practice, we train *D* and *G* *iteratively*:
  - Freeze *G*, and perform SGD on *D* for *k* iterations to increase f_{acc}.

Improve *D*'s forgery detection accuracy for a fixed distribution of fake data.

Training GANs

$$f_{\text{acc}}(\theta_G, \theta_D) = \mathbb{E}_{\mathbf{x} \sim P_{\text{data}}(\mathbf{x})}[\log D_{\theta_D}(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim P(\mathbf{z})}[\log(1 - D_{\theta_D}(G_{\theta_G}(\mathbf{z})))]$$

- In practice, we train *D* and *G* *iteratively*:
  - Freeze *G*, and perform SGD on *D* for *k* iterations to increase f_{acc}.
  - Freeze *D*, and perform SGD on *G* for *l* iterations to decrease f_{acc}.

Training GANs

$$f_{\rm acc}(\theta_G, \theta_D) = \mathbb{E}_{\mathbf{x} \sim P_{\rm data}(\mathbf{x})}[\log D_{\theta_D}(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim P(\mathbf{z})}[\log(1 - D_{\theta_D}(G_{\theta_G}(\mathbf{z})))]$$

- In practice, we train *D* and *G* *iteratively*:
  - Freeze *G*, and perform SGD on *D* for *k* iterations to increase $f_{\rm acc}$.
  - Freeze *D*, and perform SGD on *G* for *l* iterations to decrease $f_{\rm acc}$.

Improve *G* for a fixed forgery detector *D*.

Difficulty in training

- GANs are renowned for being difficult to train:

  1. How to choose k, l? More hyperparameters to optimize.

Difficulty in training

- GANs are renowned for being difficult to train:

  1. How to choose *k*, *l*? More hyperparameters to optimize.

  2. We will probably never reach the equilibrium where *G* exactly produces $P_{\text{data}}(\mathbf{x})$ and *D*'s accuracy is 0.5.
     - What kind of “training curve” for *D*, *G* should we expect?
     - If *D* gets too good too fast, then *G* may never have a chance to improve.

Difficulty in training

- GANs are renowned for being difficult to train:

  3.**Mode collapse** — *G* generates realistic data but only for a *subset* of the domain of $P_{\text{data}}(\mathbf{x})$, e.g.:

https://machinelearningmastery.com/practical-guide-to-gan-failure-modes/

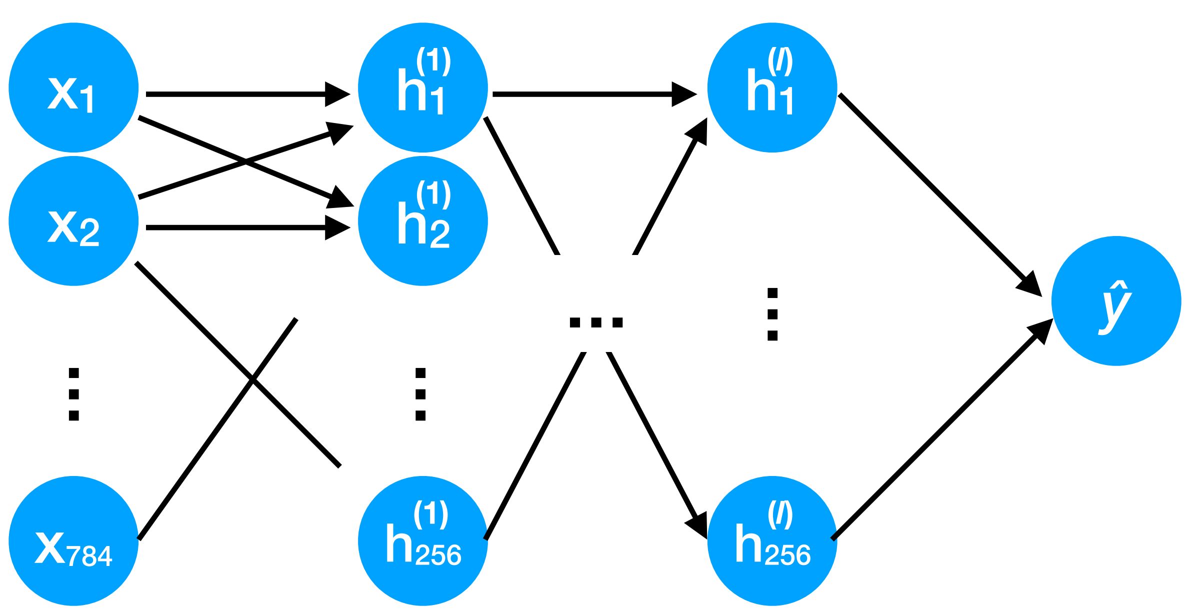
Difficulty in training

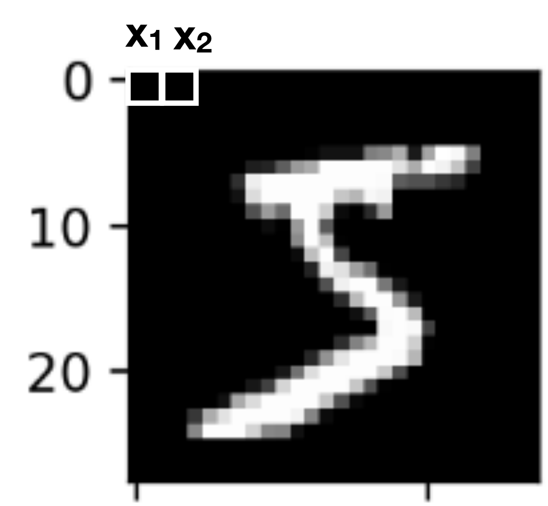
- GANs are renowned for being difficult to train:

  3. **Neuron co-adaptation** — training gets stuck because multiple NN pathways rely on each other too much.

     - Consider an MNIST image near the borders: what property do pixels x_1, x_2 have?

       - $x_1 = x_2 = 0$?





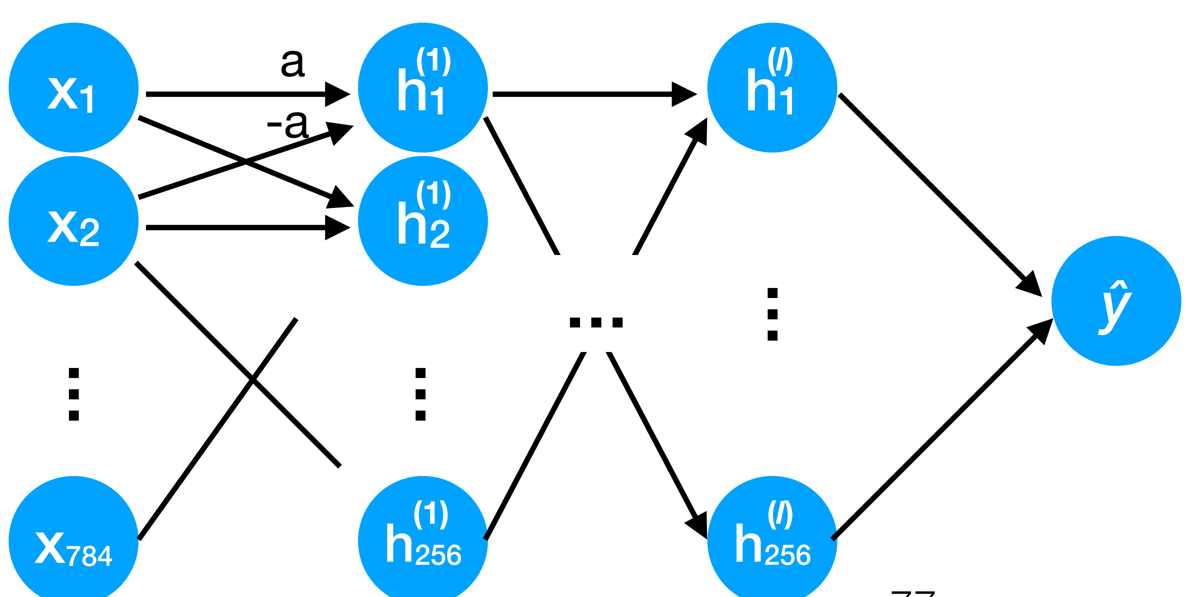
https://machinelearningmastery.com/practical-guide-to-gan-failure-modes/

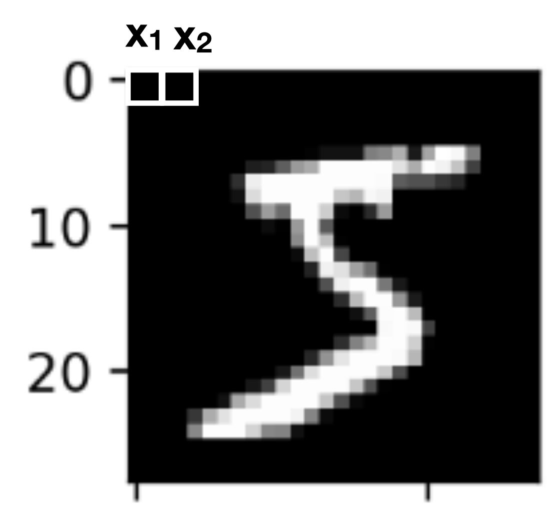
Difficulty in training

- GANs are renowned for being difficult to train:

  3.**Neuron co-adaptation** — training gets stuck because multiple NN pathways rely on each other too much.

  - Consider an MNIST image near the borders: what property do pixels x_1, x_2 have?
    - $ax_1 - ax_2 = 0$?





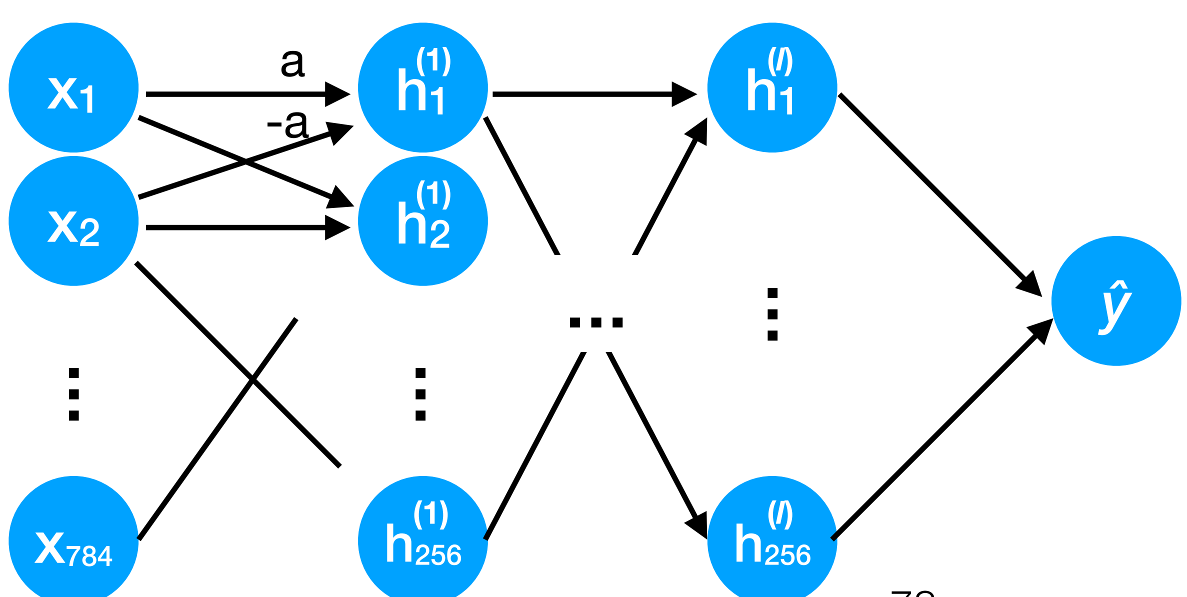
https://machinelearningmastery.com/practical-guide-to-gan-failure-modes/

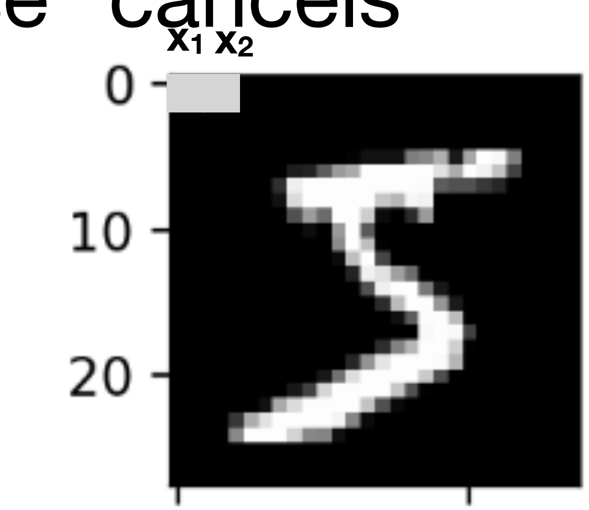
Difficulty in training

- GANs are renowned for being difficult to train:

  3. **Neuron co-adaptation** — training gets stuck because multiple NN pathways rely on each other too much.

     - In the latter case, *D* gives feedback to *G* that images are "ok" as long the background noise "cancels" itself, e.g.:





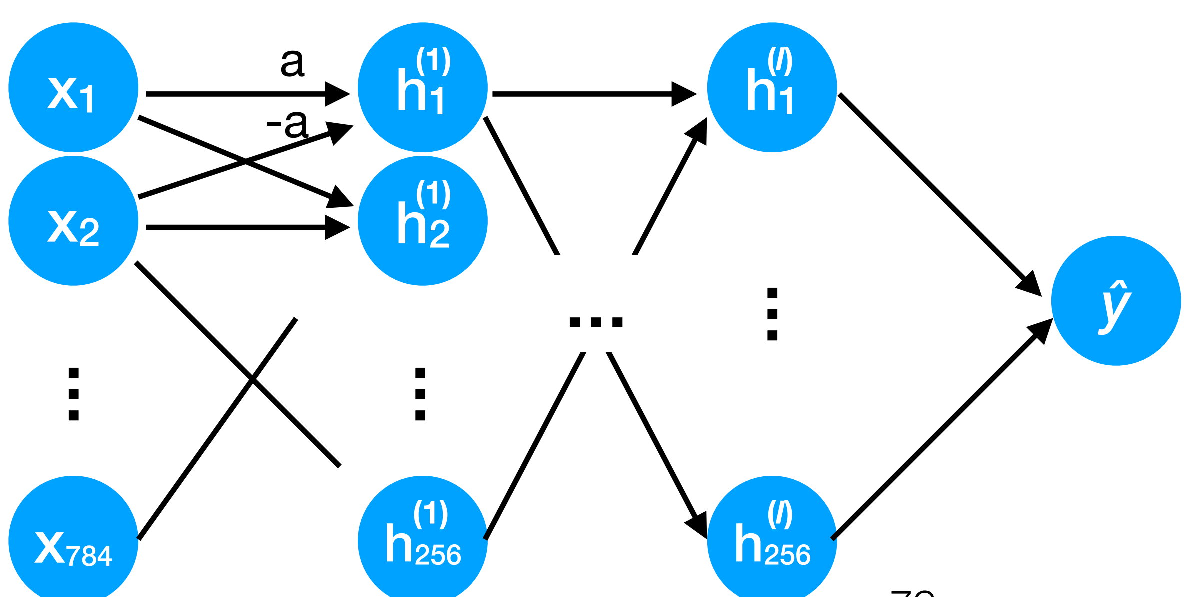
https://machinelearningmastery.com/practical-guide-to-gan-failure-modes/

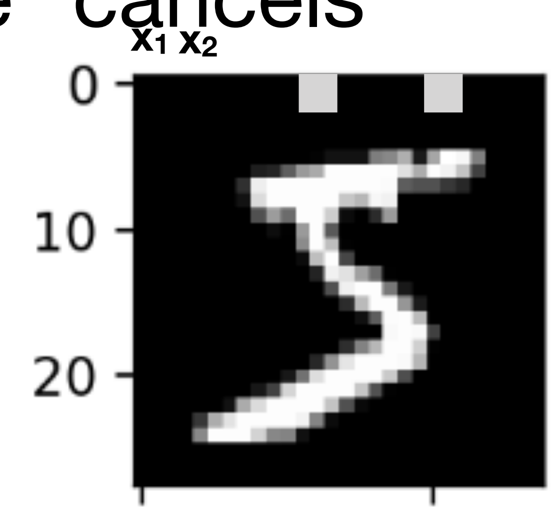
Difficulty in training

- GANs are renowned for being difficult to train:

  3. **Neuron co-adaptation** — training gets stuck because multiple NN pathways rely on each other too much.

     - In the latter case, *D* gives feedback to *G* that images are “ok" as long the background noise “cancels” itself, e.g.:



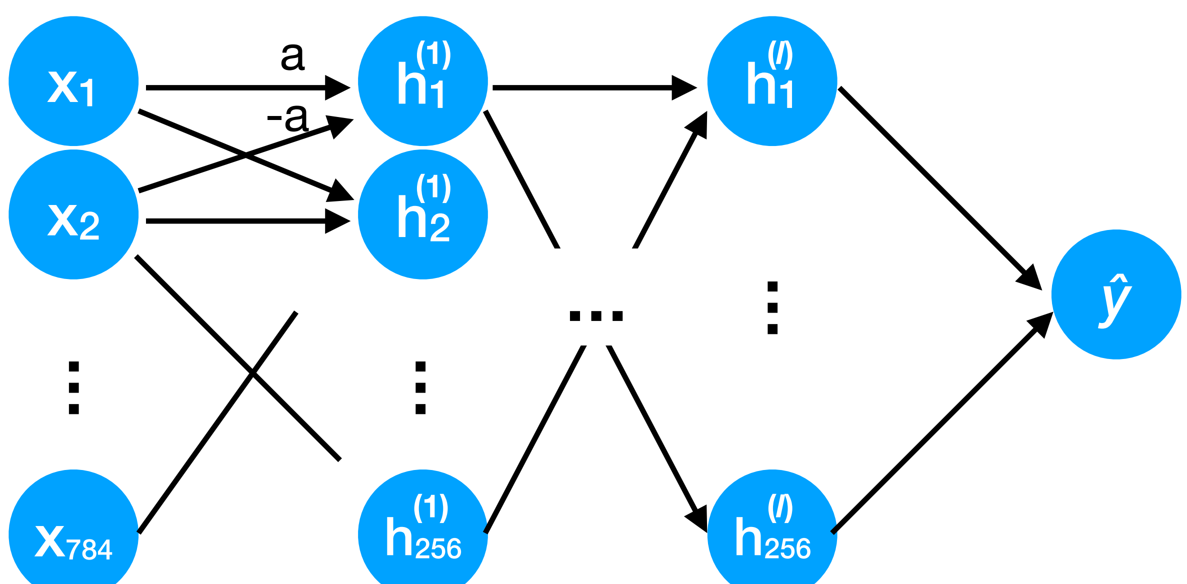


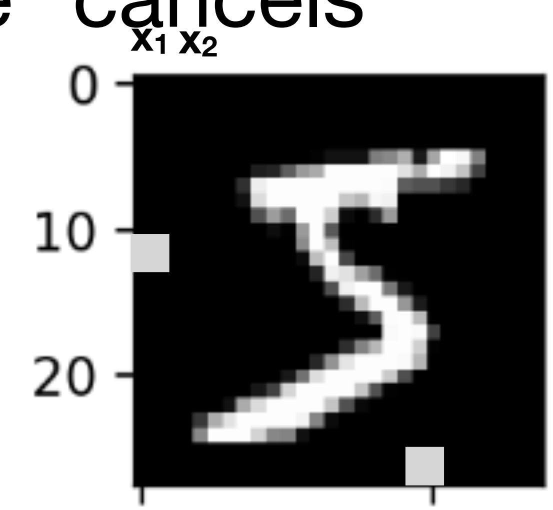
https://machinelearningmastery.com/practical-guide-to-gan-failure-modes/

Difficulty in training

- GANs are renowned for being difficult to train:

  3. **Neuron co-adaptation** — training gets stuck because multiple NN pathways rely on each other too much.
     - In the latter case, *D* gives feedback to *G* that images are “ok" as long the background noise “cancels” itself, e.g.:



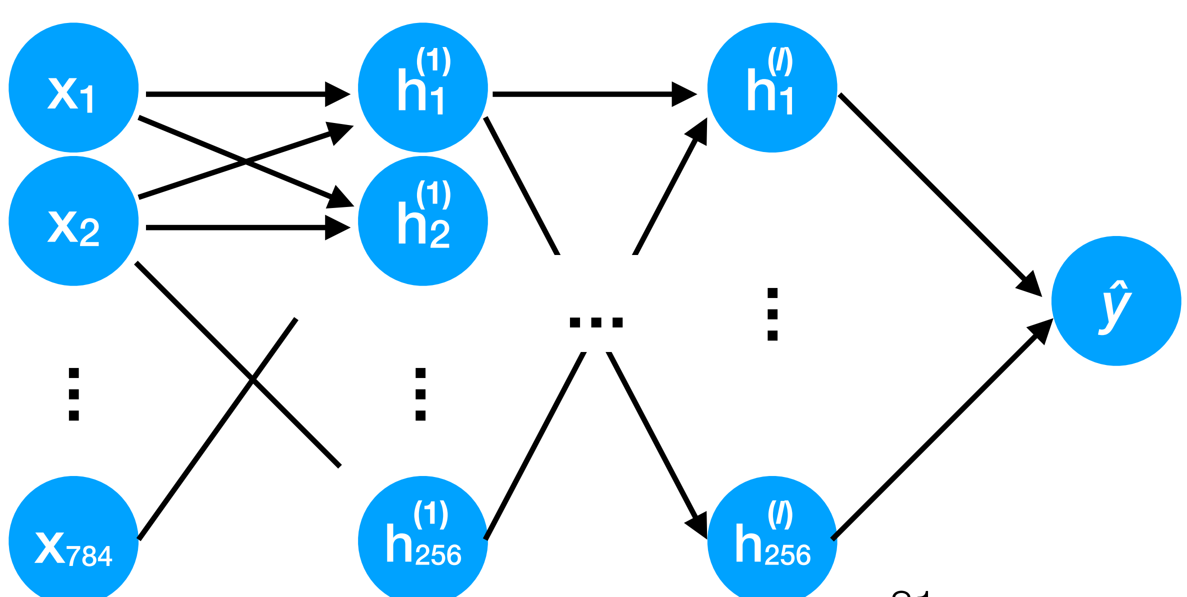


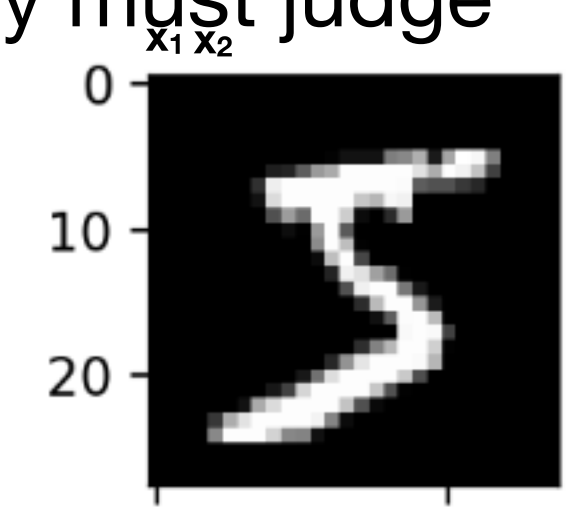
Difficulty in training

- GANs are renowned for being difficult to train:

  3. **Neuron co-adaptation** — training gets stuck because multiple NN pathways rely on each other too much.

     - To prevent this from occurring, we can use dropout on the input layer **x**, so that each pathway must judge independently if the image is a fake.





https://machinelearningmastery.com/practical-guide-to-gan-failure-modes/

Closer look at f_{acc}

- Consider the loss term for fake data:

$$f_{\text{acc}}(\theta_G, \theta_D) = \mathbb{E}_{\mathbf{x} \sim P_{\text{data}}(\mathbf{x})}[\log D_{\theta_D}(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim P(\mathbf{z})}[\log(1 - D_{\theta_D}(G_{\theta_G}(\mathbf{z})))]$$

- What happens early during training, when *G* is not very good (but *D* typically is fairly good)?

$$\nabla_{\theta_G} \log(1 - D(G(\mathbf{z}))) = -\frac{1}{1 - D(G(\mathbf{z}))} \frac{\partial D}{\partial G} \frac{\partial G}{\partial \theta_G}$$

Closer look at f_{acc}

- Consider the loss term for fake data:

$$f_{\text{acc}}(\theta_G, \theta_D) = \mathbb{E}_{\mathbf{x}\sim P_{\text{data}}(\mathbf{x})}[\log D_{\theta_D}(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim P(\mathbf{z})}[\log(1 - D_{\theta_D}(G_{\theta_G}(\mathbf{z})))]$$

- What happens early during training, when *G* is not very good (but *D* typically is fairly good)?

- *D* will output ~0.

$$\nabla_{\theta_G} \log(1 - D(G(\mathbf{z}))) = -\frac{1}{1 - D(G(\mathbf{z}))} \frac{\partial D}{\partial G} \frac{\partial G}{\partial \theta_G}$$

Closer look at f_{acc}

- Consider the loss term for fake data:

$$f_{\text{acc}}(\theta_G, \theta_D) = \mathbb{E}_{\mathbf{x} \sim P_{\text{data}}(\mathbf{x})}[\log D_{\theta_D}(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim P(\mathbf{z})}[\log(1 - D_{\theta_D}(G_{\theta_G}(\mathbf{z})))]$$

- What happens early during training, when *G* is not very good (but *D* typically is fairly good)?

- *D* will output ~0.

$$\begin{aligned} \nabla_{\theta_G} \log(1 - D(G(\mathbf{z}))) &= -\frac{1}{1 - D(G(\mathbf{z}))} \frac{\partial D}{\partial G} \frac{\partial G}{\partial \theta_G} \\ &= -\frac{1}{1} \sigma'(v) \frac{\partial v}{\partial G} \frac{\partial G}{\partial \theta_G} \end{aligned}$$

Here we assume *D* uses a logistic sigmoid *σ* as its output layer, whose input is *v*.

Closer look at f_{acc}

- Consider the loss term for fake data:

$$f_{\text{acc}}(\theta_G, \theta_D) = \mathbb{E}_{\mathbf{x} \sim P_{\text{data}}(\mathbf{x})}[\log D_{\theta_D}(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim P(\mathbf{z})}[\log(1 - D_{\theta_D}(G_{\theta_G}(\mathbf{z})))]$$

- What happens early during training, when *G* is not very good (but *D* typically is fairly good)?

- *D* will output ~0.

$$\begin{aligned} \nabla_{\theta_G} \log(1 - D(G(\mathbf{z}))) &= -\frac{1}{1 - D(G(\mathbf{z}))} \frac{\partial D}{\partial G} \frac{\partial G}{\partial \theta_G} \\ &= -\frac{1}{1} \sigma'(v) \frac{\partial v}{\partial G} \frac{\partial G}{\partial \theta_G} \\ &\approx -1 \times 0 \times \frac{\partial v}{\partial G} \frac{\partial G}{\partial \theta_G} \end{aligned}$$

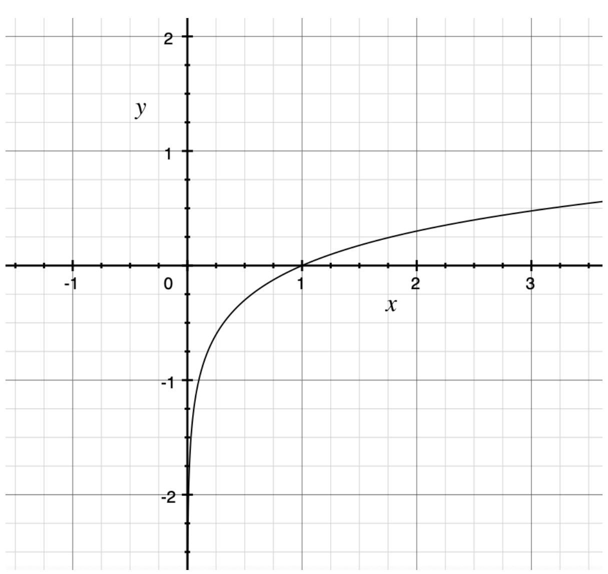
Closer look at f_{acc}

- To accelerate training early on, we can instead use a different loss term for the fake data that yields the same desired behavior but trains faster.

$$f_{\text{acc}}(\theta_G, \theta_D) = \mathbb{E}_{\mathbf{x} \sim P_{\text{data}}(\mathbf{x})}[\log D_{\theta_D}(\mathbf{x})] + \boxed{\mathbb{E}_{\mathbf{z} \sim P(\mathbf{z})}[-\log(D_{\theta_D}(G_{\theta_G}(\mathbf{z})))]}$$

New loss term

- The reason is that the gradient of -log(*q*) for q≈0 is very large, whereas the gradient of log(1-*q*)≈1.



$$\begin{aligned}
\nabla_{\theta_G} - \log D(G(\mathbf{z})) &= -\frac{1}{D(G(\mathbf{z}))} \frac{\partial D}{\partial G} \frac{\partial G}{\partial \theta_G} \\
&= -\frac{1}{\text{small}} \sigma'(v) \frac{\partial v}{\partial G} \frac{\partial G}{\partial \theta_G} \\
&\approx -\frac{1}{\text{small}} \times \text{small} \times \frac{\partial v}{\partial G} \frac{\partial G}{\partial \theta_G}
\end{aligned}$$

Conditional GANs

- One variant is a **conditional GAN:**
  - *G* also accepts a parameter vector **f** (e.g., 1-hot encoding of MNIST class) that specifies what *kind* of data to generate.
  - *D* also accepts **f** to help discriminate a particular kind of real from fake data.