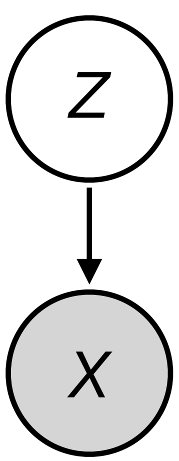
CS/DS 541: Class 18

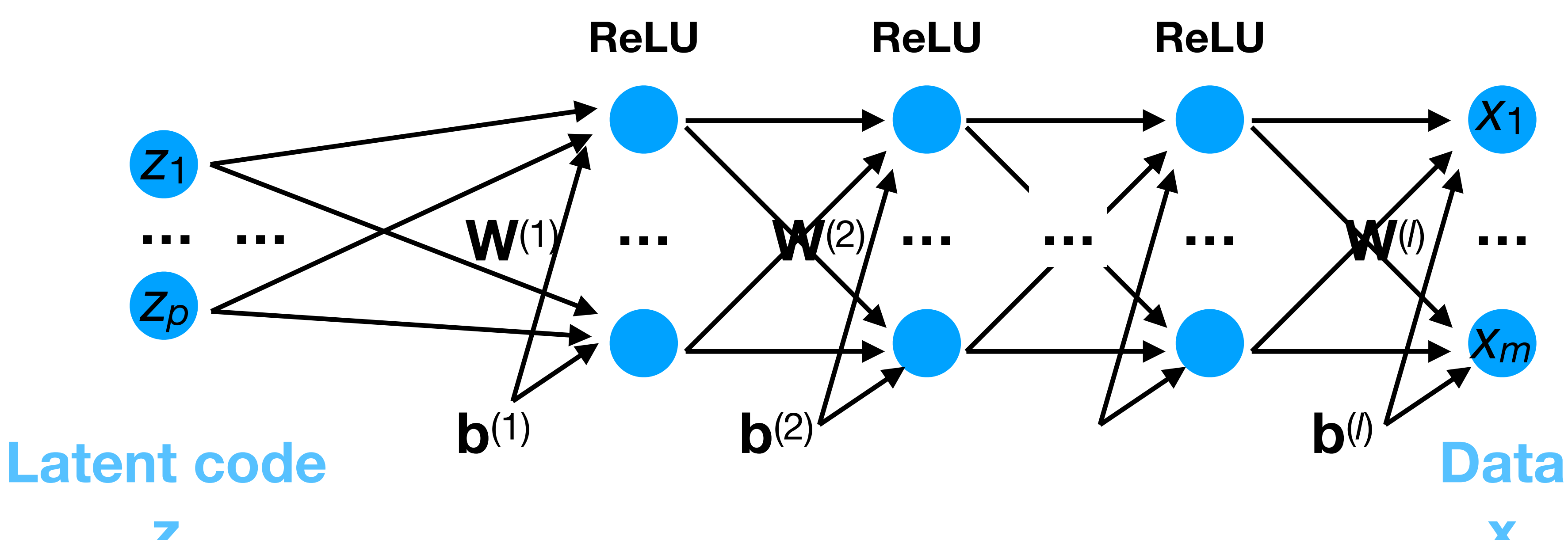
Jacob Whitehill

Variational Auto-encoders (VAE)

Latent variable models

- With DL, we can construct and train deeper and more powerful LVMs that are non-linear.
- The generated data can be much more realistic, e.g.:

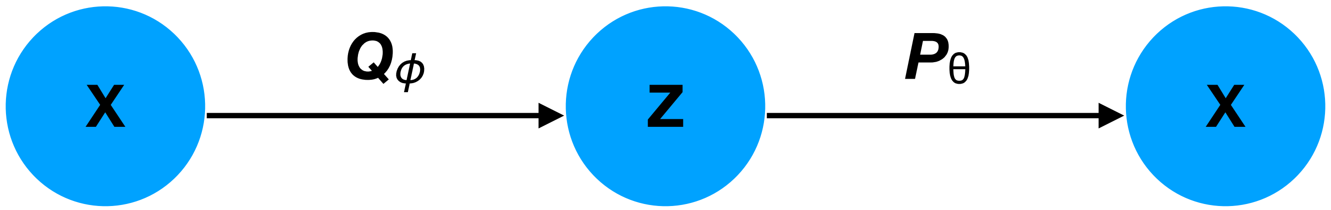




VAE architecture

- The VAE consists of an encoder Q_ϕ and decoder P_θ.
  - $Q(\mathbf{z} \mid \mathbf{x})$ outputs a probability distribution over **Z** given **X**.
  - $P(\mathbf{x} \mid \mathbf{z})$ outputs a probability distribution over **X** given **Z**.

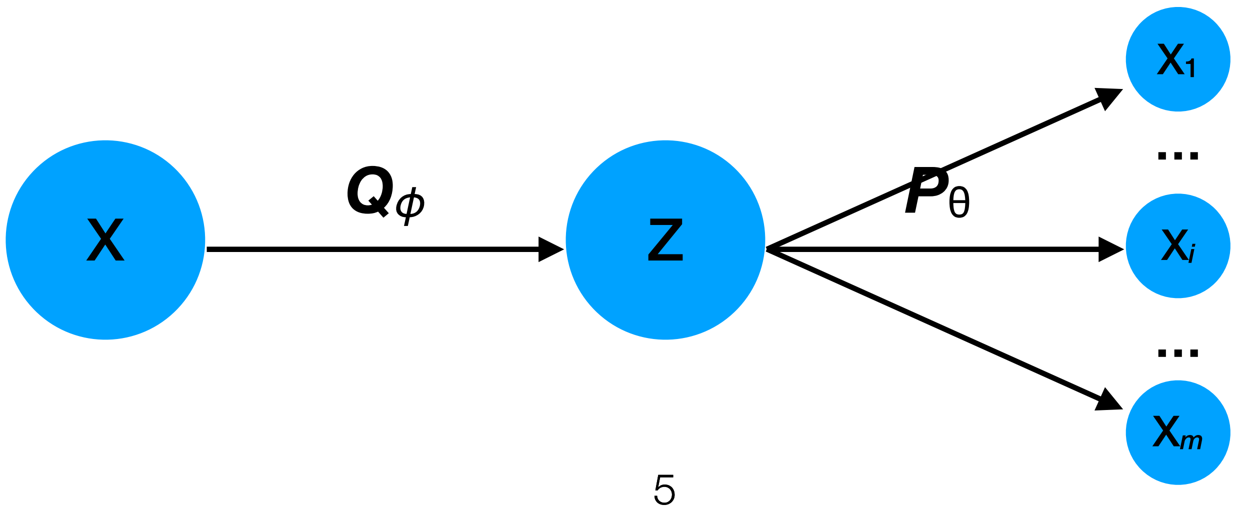
- We fix the probability distribution of the hidden state to be $P(\mathbf{z}) = \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I})$; this makes it easy to generate new data.



VAE architecture

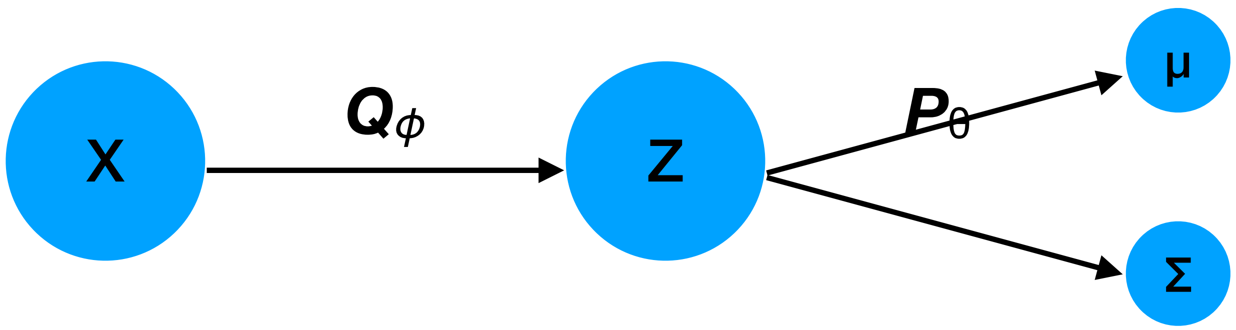
- To generate examples $\mathbf{x} \in [0,1]^m$ (like in Homework 7), $P_\theta(\mathbf{x} \mid \mathbf{z})$ can end with m logistic sigmoid functions.

- Each output i of $P_\theta(\mathbf{x} \mid \mathbf{z})$ represents the probability that the corresponding pixel i has value 1.



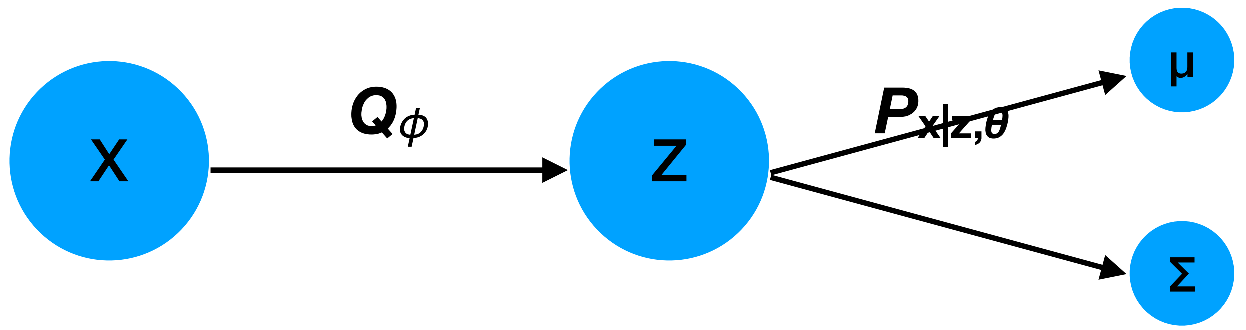
VAE architecture

- One might also construct $P_\theta(\mathbf{x} \mid \mathbf{z})$ so that it outputs the multivariate mean and covariance of a Normal distribution.



VAE architecture

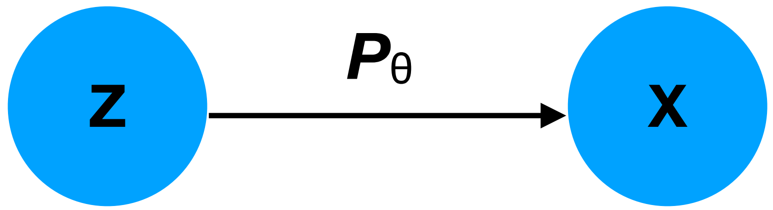
- To represent more clearly the distinction between the ground-truth **x** and the model's prediction, we could rename $P_\theta(\mathbf{x} \mid \mathbf{z})$ as $P_{x|z,\theta}(\mathbf{x})$, i.e., the likelihood of obtaining ground-truth **x** from the probability distribution estimated by the model given input **z**.



VAE architecture

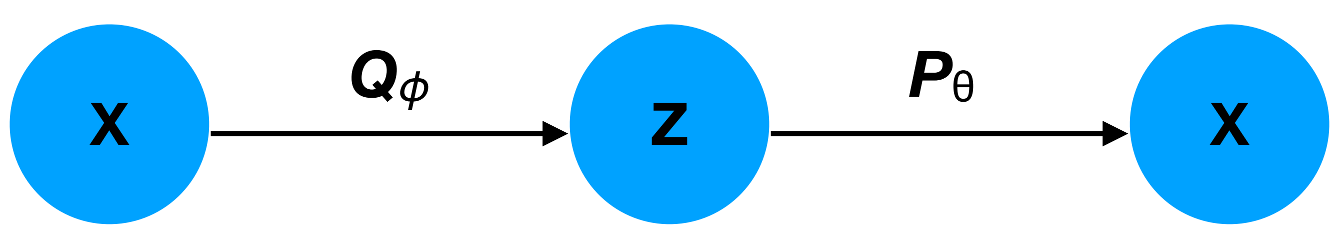
- Here is how we can generate data:

1. Sample $\mathbf{z} \sim P(\mathbf{z}) = \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I})$
2. Compute $P_\theta(\mathbf{x} \mid \mathbf{z})$
3. Sample $\mathbf{x} \sim P_\theta(\mathbf{x} \mid \mathbf{z})$



VAE architecture

- The parameters ϕ and θ are trained using maximum-likelihood estimation (MLE).
- We aim to maximize the likelihood of our observed training data, given P's parameters θ, i.e.: $P_\theta(\{\mathbf{x}^{(i)}\}_{i=1}^{n})$
- Using a variational approximation technique, we will also optimize Q's parameters ϕ along the way.



VAE: MLE derivation

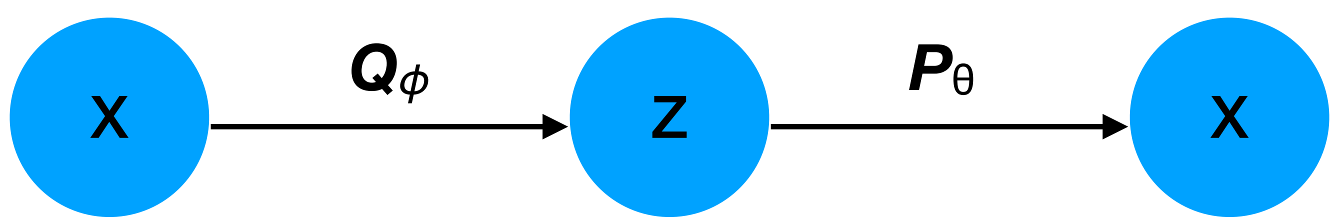
$$
\begin{aligned}
\log P(\mathbf{x}) &= \log \int_{\mathbf{z}} P(\mathbf{x}, \mathbf{z}) d\mathbf{z} \\
&= \log \int_{\mathbf{z}} P(\mathbf{x} \mid \mathbf{z}) P(\mathbf{z}) d\mathbf{z} \\
&= \log \int_{\mathbf{z}} Q(\mathbf{z} \mid \mathbf{x}) \frac{P(\mathbf{x} \mid \mathbf{z}) P(\mathbf{z})}{Q(\mathbf{z} \mid \mathbf{x})} d\mathbf{z} \\
&\geq \int_{\mathbf{z}} Q(\mathbf{z} \mid \mathbf{x}) \log \left(P(\mathbf{x} \mid \mathbf{z}) \frac{P(\mathbf{z})}{Q(\mathbf{z} \mid \mathbf{x})} \right) d\mathbf{z} \\
&= \int_{\mathbf{z}} Q(\mathbf{z} \mid \mathbf{x}) \log \frac{P(\mathbf{z})}{Q(\mathbf{z} \mid \mathbf{x})} d\mathbf{z} + \int_{\mathbf{z}} Q(\mathbf{z} \mid \mathbf{x}) \log P(\mathbf{x} \mid \mathbf{z}) d\mathbf{z} \\
&= -D_{\mathrm{KL}}(Q(\mathbf{z} \mid \mathbf{x}) \parallel P(\mathbf{z})) + \mathbb{E}_Q[\log P(\mathbf{x} \mid \mathbf{z})] \\
&= -D_{\mathrm{KL}}(Q_\phi(\mathbf{z} \mid \mathbf{x}) \parallel P(\mathbf{z})) + \mathbb{E}_{Q_\phi}[\log P_\theta(\mathbf{x} \mid \mathbf{z})]
\end{aligned}
$$

VAE: MLE derivation

- In other words: to maximize $P(\mathbf{x})$, we want to:
  - Minimize KL-divergence of hidden state w.r.t. standard normal distribution.

    and

  - Maximize the reconstruction probability.



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

Maximizing the lower bound

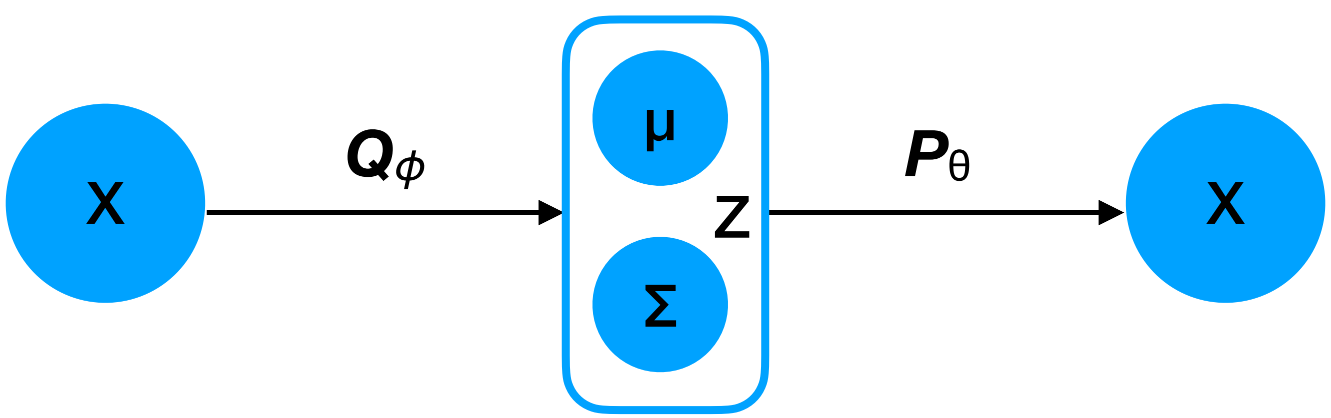
- How do we optimize this w.r.t. θ and ϕ?

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

- The first term has closed-form differentiable solutions when $P(\mathbf{z})$ and $Q_\phi(\mathbf{z} \mid \mathbf{x})$ are Gaussian (see previous lecture).

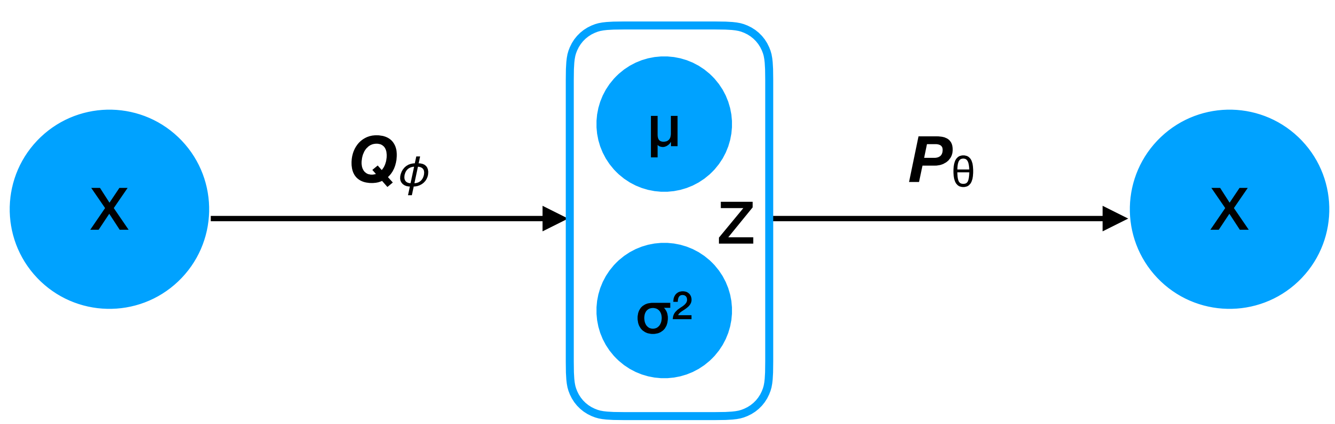
VAE architecture

- How do we force Q_ϕ to output a Gaussian distribution?
  - Given **x**, Q_ϕ needs to output:
    - Mean $\boldsymbol{\mu}$
    - Covariance matrix $\boldsymbol{\Sigma}$



VAE architecture

- As a simplification, Q_ϕ can output a diagonal covariance matrix parameterized by just a vector [$\sigma_1^2, \ldots, \sigma_p^2$].
- All in all, Q_ϕ outputs $2p$ entries, where the first p specify the mean and the second p specify the covariance.
- We must force positivity of [$\sigma_1^2, \ldots, \sigma_p^2$], e.g., by exponentiating or squaring the output of Q_ϕ.



Maximizing the lower bound

- How do we optimize this w.r.t. θ and ϕ?

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

- The second term is more problematic — we can try to estimate the expectation by sampling:

$$\mathbb{E}_{Q_\phi}[\log P_\theta(\mathbf{x} \mid \mathbf{z})] \approx \frac{1}{n} \sum_{i=1}^{n} \log P_\theta(\mathbf{x}^{(i)} \mid \mathbf{z}^{(i)})$$

$$\text{where } \mathbf{z}^{(i)} \sim Q_\phi(\mathbf{z} \mid \mathbf{x}^{(i)})$$

Maximizing the lower bound

- How do we optimize this w.r.t. θ and ϕ?

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

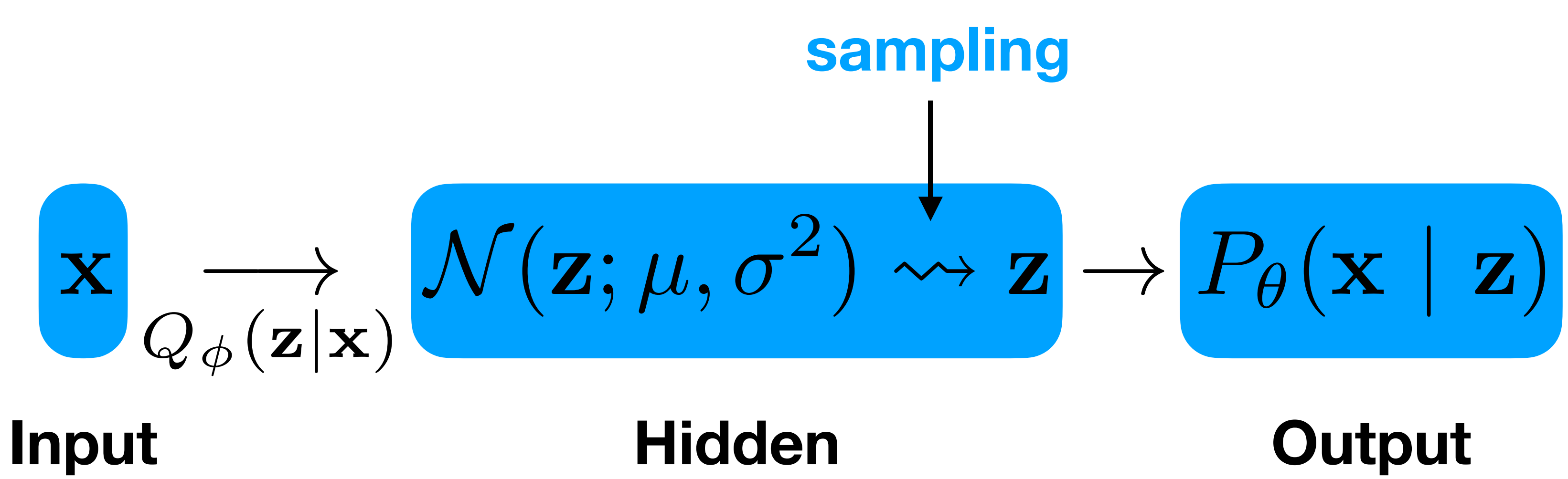
- But sampling a value from a probability distribution is a **non-differentiable operation** — we can no longer use back-propagation:

sampling

$$\underbrace{\mathbf{x}}_{\textbf{Input}} \rightarrow \underbrace{Q_\phi(\mathbf{z} \mid \mathbf{x}) \rightsquigarrow \mathbf{z}}_{\textbf{Hidden}} \rightarrow \underbrace{P_\theta(\mathbf{x} \mid \mathbf{z})}_{\textbf{Output}}$$

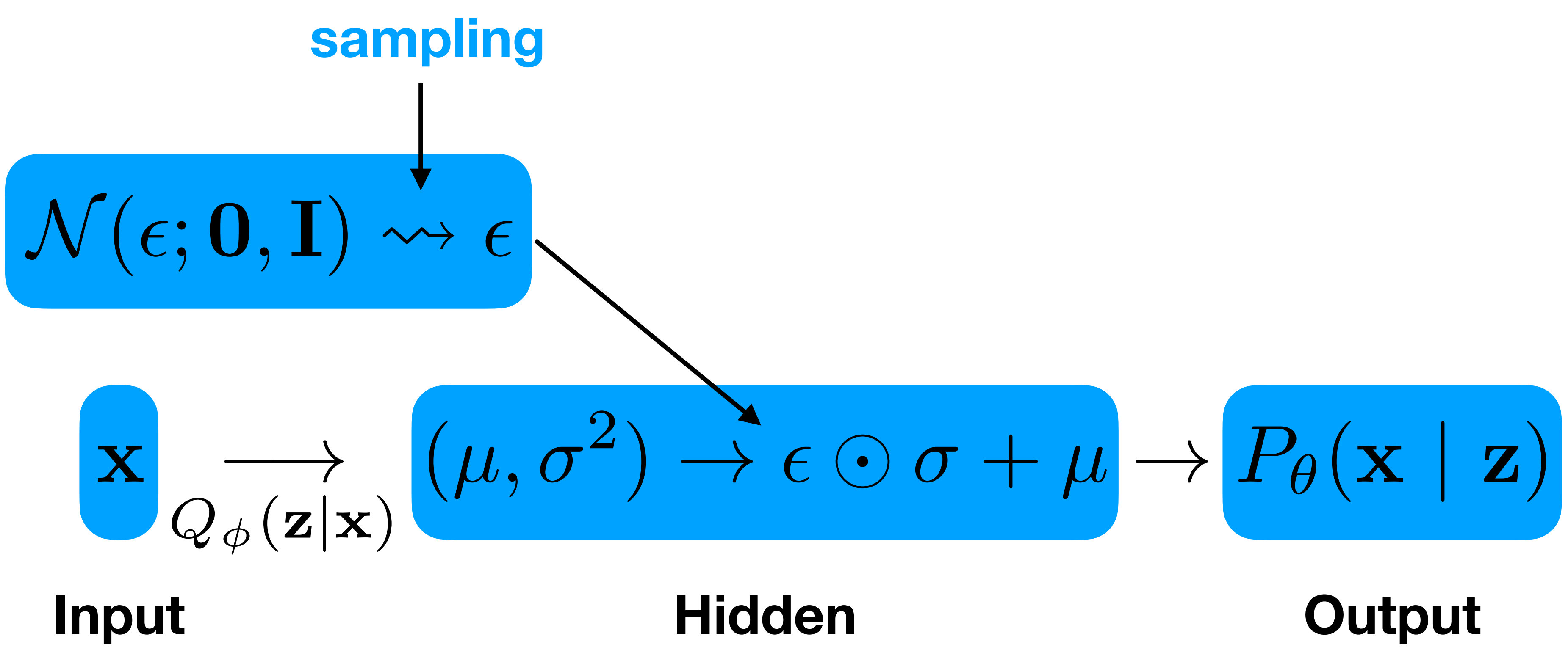
Reparameterization trick

- In the context of the VAE:
  - Instead of sampling $\mathbf{z} \sim Q_\phi(\mathbf{z} \mid \mathbf{x})$ within the computational graph, which would break back-propagation…



Reparameterization trick

- In the context of the VAE:
  - …we instead sample from outside the graph, multiply the result element-wise by vector σ, and add **μ**.



Training procedure

- Define networks Q_ϕ and decoder P_θ.
- Initialize parameters ϕ and θ.

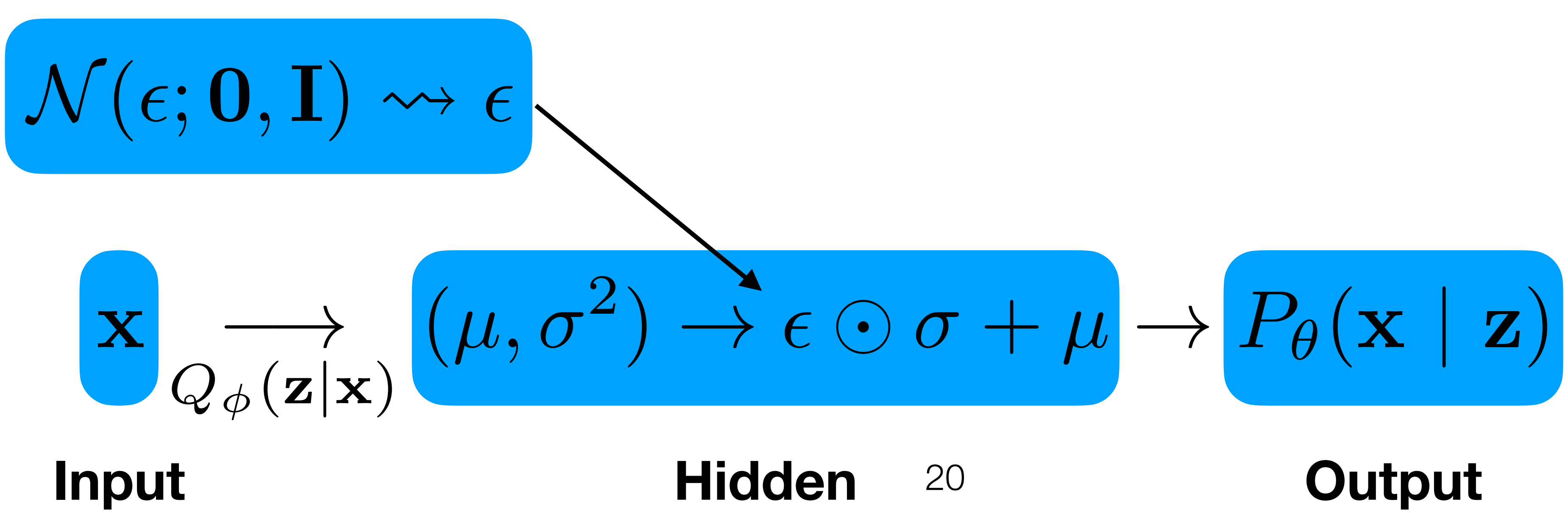
$$\mathcal{N}(\epsilon; \mathbf{0}, \mathbf{I}) \rightsquigarrow \epsilon$$

$$\mathbf{x} \xrightarrow[Q_\phi(\mathbf{z}|\mathbf{x})]{} (\mu, \sigma^2) \to \epsilon \odot \sigma + \mu \to P_\theta(\mathbf{x} \mid \mathbf{z})$$

Input **Hidden** **Output**

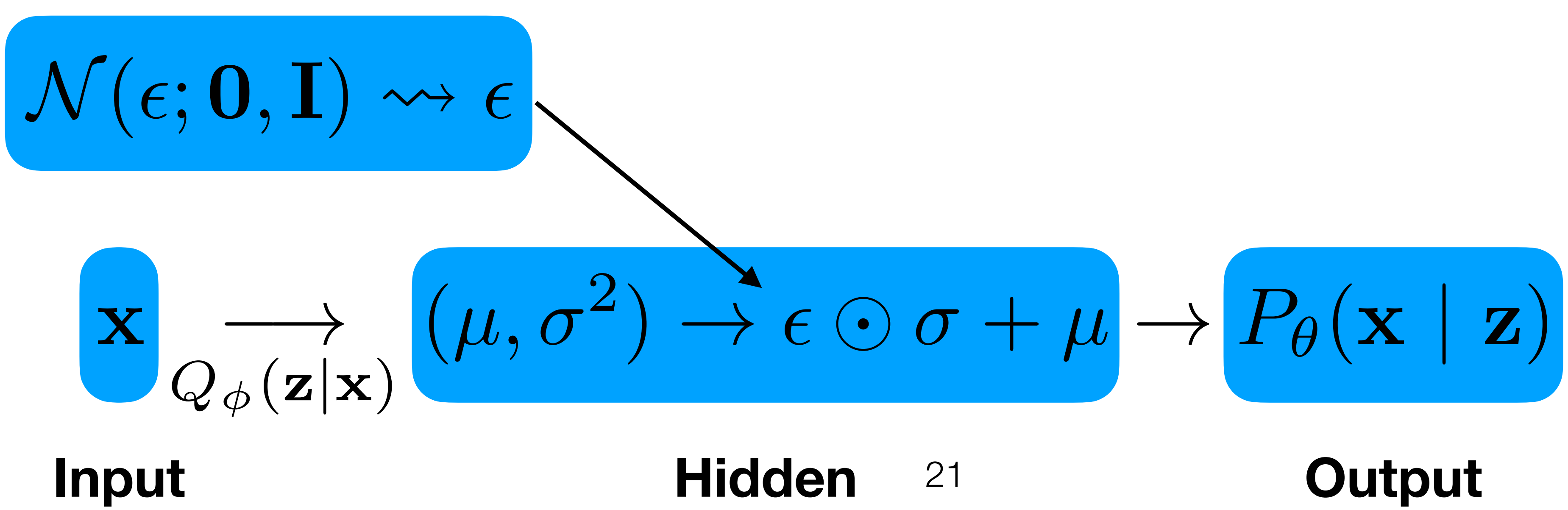
Training procedure

- Define networks Q_ϕ and decoder P_θ.
- Initialize parameters ϕ and θ.
- For each mini-batch:
  - Select $\tilde{n}$ examples: $\{\mathbf{x}^{(i)}\}_{i=1}^{\tilde{n}} \subset \mathbb{R}^m$



Training procedure

- Define networks Q_ϕ and decoder P_θ.
- Initialize parameters ϕ and θ.
- For each mini-batch:
  - Select $\tilde{n}$ examples: $\{\mathbf{x}^{(i)}\}_{i=1}^{\tilde{n}} \subset \mathbb{R}^m$
  - Sample $\tilde{n}$ noise vectors: $\{\epsilon^{(i)}\}_{i=1}^{\tilde{n}} \subset \mathbb{R}^p$



Training procedure

- Define networks Q_ϕ and decoder P_θ.
- Initialize parameters ϕ and θ.
- For each mini-batch:
    - Select $\tilde{n}$ examples: $\{\mathbf{x}^{(i)}\}_{i=1}^{\tilde{n}} \subset \mathbb{R}^m$
    - Sample $\tilde{n}$ noise vectors: $\{\epsilon^{(i)}\}_{i=1}^{\tilde{n}} \subset \mathbb{R}^p$
    - Compute: $(\mu^{(i)}, {\sigma^{(i)}}^2) = Q_\phi(\mathbf{z}^{(i)} \mid \mathbf{x}^{(i)}), \;\; \forall i$

$$\mathcal{N}(\epsilon; \mathbf{0}, \mathbf{I}) \rightsquigarrow \epsilon$$

$$\mathbf{x} \xrightarrow[Q_\phi(\mathbf{z}|\mathbf{x})]{} (\mu, \sigma^2) \to \epsilon \odot \sigma + \mu \to P_\theta(\mathbf{x} \mid \mathbf{z})$$

Input **Hidden** **Output**

Training procedure

- Define networks Q_ϕ and decoder P_θ.
- Initialize parameters ϕ and θ.
- For each mini-batch:
  - Select $\tilde{n}$ examples: $\{\mathbf{x}^{(i)}\}_{i=1}^{\tilde{n}} \subset \mathbb{R}^m$
  - Sample $\tilde{n}$ noise vectors: $\{\epsilon^{(i)}\}_{i=1}^{\tilde{n}} \subset \mathbb{R}^p$
  - Compute: $(\mu^{(i)}, {\sigma^{(i)}}^2) = Q_\phi(\mathbf{z}^{(i)} \mid \mathbf{x}^{(i)}), \ \forall i$
  - Compute: $\mathbf{z}^{(i)} = \epsilon^{(i)} \odot \sigma^{(i)} + \mu^{(i)}$

$$\mathbf{x} \xrightarrow[Q_\phi(\mathbf{z}|\mathbf{x})]{} (\mu, \sigma^2) \rightarrow \epsilon \odot \sigma + \mu \rightarrow P_\theta(\mathbf{x} \mid \mathbf{z})$$

Input **Hidden** **Output**

Training procedure

- Define networks Q_ϕ and decoder P_θ.
- Initialize parameters ϕ and θ.
- For each mini-batch:
  - Select $\tilde{n}$ examples: $\{\mathbf{x}^{(i)}\}_{i=1}^{\tilde{n}} \subset \mathbb{R}^m$
  - Sample $\tilde{n}$ noise vectors: $\{\epsilon^{(i)}\}_{i=1}^{\tilde{n}} \subset \mathbb{R}^p$
  - Compute: $(\mu^{(i)}, {\sigma^{(i)}}^2) = Q_\phi(\mathbf{z}^{(i)} \mid \mathbf{x}^{(i)}), \ \forall i$
  - Compute: $\mathbf{z}^{(i)} = \epsilon^{(i)} \odot \sigma^{(i)} + \mu^{(i)}$
  - Compute likelihood:
    $$\log P_\theta(\mathbf{x}^{(i)} \mid \mathbf{z}^{(i)}) - D_{\mathrm{KL}}(Q_\phi(\mathbf{z}^{(i)};\ \mathbf{x}^{(i)}) \parallel \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I}))$$

Training procedure

- Define networks Q_ϕ and decoder P_θ.
- Initialize parameters ϕ and θ.
- For each mini-batch:
  - Select $\tilde{n}$ examples: $\{\mathbf{x}^{(i)}\}_{i=1}^{\tilde{n}} \subset \mathbb{R}^m$
  - Sample $\tilde{n}$ noise vectors: $\{\epsilon^{(i)}\}_{i=1}^{\tilde{n}} \subset \mathbb{R}^p$
  - Compute: $(\mu^{(i)}, {\sigma^{(i)}}^2) = Q_\phi(\mathbf{z}^{(i)} \mid \mathbf{x}^{(i)}), \ \forall i$
  - Compute: $\mathbf{z}^{(i)} = \epsilon^{(i)} \odot \sigma^{(i)} + \mu^{(i)}$
  - Compute likelihood:

$$\log P_\theta(\mathbf{x}^{(i)} \mid \mathbf{z}^{(i)}) - D_{\mathrm{KL}}(Q_\phi(\mathbf{z}^{(i)};\ \mathbf{x}^{(i)}) \parallel \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I}))$$

Note: this is an approximation of the expectation with just 1 sample.

Training procedure

- Define networks Q_ϕ and decoder P_θ.
- Initialize parameters ϕ and θ.
- For each mini-batch:
  - Select $\tilde{n}$ examples: $\{\mathbf{x}^{(i)}\}_{i=1}^{\tilde{n}} \subset \mathbb{R}^m$
  - Sample $\tilde{n}$ noise vectors: $\{\epsilon^{(i)}\}_{i=1}^{\tilde{n}} \subset \mathbb{R}^p$
  - Compute: $(\mu^{(i)}, {\sigma^{(i)}}^2) = Q_\phi(\mathbf{z}^{(i)} \mid \mathbf{x}^{(i)}), \ \forall i$
  - Compute: $\mathbf{z}^{(i)} = \epsilon^{(i)} \odot \sigma^{(i)} + \mu^{(i)}$
  - Compute likelihood:
    $$\log P_\theta(\mathbf{x}^{(i)} \mid \mathbf{z}^{(i)}) - D_{\mathrm{KL}}(Q_\phi(\mathbf{z}^{(i)};\ \mathbf{x}^{(i)}) \parallel \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I}))$$
  - Update ϕ and θ using back-propagation.

Optimization objective

- Note that, for the VAE, we want to maximize a likelihood instead of minimizing a loss.
- The likelihood consists of two components:
  - The reconstruction probability is computed w.r.t. each dimension of $\mathbf{x}^{(i)} \in \mathbb{R}^m$ and then summed, e.g.:

$\mathbf{x}^{(i)}$ = `[0.1,  0.8,  0.7,  0.23, 0.5, …]` // Ground-truth
$\hat{\mathbf{x}}^{(i)}$ = `[0.08, 0.83, 0.58, 0.21, 0.42, …]` // $P_\theta(\mathbf{x}^{(i)} \mid \mathbf{z}^{(i)})$

$$\log P_\theta(\mathbf{x}^{(i)} \mid \mathbf{z}^{(i)}) - D_{\mathrm{KL}}(Q_\phi(\mathbf{z}^{(i)};\ \mathbf{x}^{(i)}) \parallel \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I}))$$

Optimization objective

- Note that, for the VAE, we want to maximize a likelihood instead of minimizing a loss.

- The likelihood consists of two components:

  - The reconstruction probability is computed w.r.t. each dimension of $\mathbf{x}^{(i)} \in \mathbb{R}^m$ and then summed, e.g.:

$\mathbf{x}^{(i)}$ = `[0.1,  0.8,  0.7,  0.23, 0.5, …]` // Ground-truth
$\hat{\mathbf{x}}^{(i)}$ = `[0.08, 0.83, 0.58, 0.21, 0.42, …]` // $P_\theta(\mathbf{x}^{(i)} \mid \mathbf{z}^{(i)})$

Log-like.

$$\log P_\theta(\mathbf{x}^{(i)} \mid \mathbf{z}^{(i)}) - D_{\mathrm{KL}}(Q_\phi(\mathbf{z}^{(i)};\ \mathbf{x}^{(i)}) \parallel \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I}))$$

Optimization objective

- Note that, for the VAE, we want to maximize a likelihood instead of minimizing a loss.

- The likelihood consists of two components:

    - The reconstruction probability is computed w.r.t. each dimension of $\mathbf{x}^{(i)} \in \mathbb{R}^m$ and then summed, e.g.:

$\mathbf{x}^{(i)}$ = `[0.1,  0.8,  0.7,  0.23, 0.5, …]` // Ground-truth
$\hat{\mathbf{x}}^{(i)}$ = `[0.08, 0.83, 0.58, 0.21, 0.42, …]` // $P_\theta(\mathbf{x}^{(i)} \mid \mathbf{z}^{(i)})$

Log-like.

$$\boxed{\log P_\theta(\mathbf{x}^{(i)} \mid \mathbf{z}^{(i)})} - D_{\mathrm{KL}}(Q_\phi(\mathbf{z}^{(i)};\ \mathbf{x}^{(i)}) \parallel \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I}))$$

Optimization objective

- Note that, for the VAE, we want to maximize a likelihood instead of minimizing a loss.

- The likelihood consists of two components:

  - The reconstruction probability is computed w.r.t. each dimension of $\mathbf{x}^{(i)} \in \mathbb{R}^m$ and then summed, e.g.:

$\mathbf{x}^{(i)}$ = `[0.1,  0.8,  0.7,  0.23, 0.5, …]` // Ground-truth
$\hat{\mathbf{x}}^{(i)}$ = `[0.08, 0.83, 0.58, 0.21, 0.42, …]` // $P_\theta(\mathbf{x}^{(i)} \mid \mathbf{z}^{(i)})$
Log-like.

$$\boxed{\log P_\theta(\mathbf{x}^{(i)} \mid \mathbf{z}^{(i)})} - D_{\mathrm{KL}}(Q_\phi(\mathbf{z}^{(i)};\ \mathbf{x}^{(i)}) \parallel \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I}))$$

Optimization objective

- Note that, for the VAE, we want to maximize a likelihood instead of minimizing a loss.

- The likelihood consists of two components:

    - The reconstruction probability is computed w.r.t. each dimension of $\mathbf{x}^{(i)} \in \mathbb{R}^m$ and then summed, e.g.:

      $\mathbf{x}^{(i)}$ = `[0.1,  0.8,  0.7,  0.23, 0.5, …]` // Ground-truth
      $\hat{\mathbf{x}}^{(i)}$ = `[0.08, 0.83, 0.58, 0.21, 0.42, …]` // $P_\theta(\mathbf{x}^{(i)} \mid \mathbf{z}^{(i)})$

      ...

$$\boxed{\log P_\theta(\mathbf{x}^{(i)} \mid \mathbf{z}^{(i)})} - D_{\mathrm{KL}}(Q_\phi(\mathbf{z}^{(i)};\ \mathbf{x}^{(i)}) \parallel \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I}))$$

Optimization objective

- Note that, for the VAE, we want to maximize a likelihood instead of minimizing a loss.

- The likelihood consists of two components:

    - The reconstruction probability is computed w.r.t. each dimension of $\mathbf{x}^{(i)} \in \mathbb{R}^m$ and then summed, e.g.:

    - In practice, you can use a binary cross-entropy loss in either TensorFlow or PyTorch.

$$\boxed{\log P_\theta(\mathbf{x}^{(i)} \mid \mathbf{z}^{(i)})} - D_{\mathrm{KL}}(Q_\phi(\mathbf{z}^{(i)};\ \mathbf{x}^{(i)}) \parallel \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I}))$$

Optimization objective

- Note that, for the VAE, we want to maximize a likelihood instead of minimizing a loss.

- The likelihood consists of two components:

    - The KL-divergence is differentiable w.r.t. μ and σ, which in turn are differentiable w.r.t. ϕ in Q.

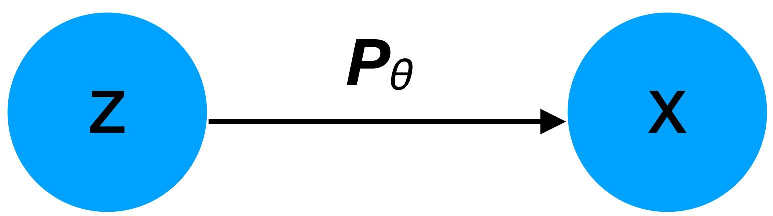
$$\log P_\theta(\mathbf{x}^{(i)} \mid \mathbf{z}^{(i)}) - \boxed{D_{\mathrm{KL}}(Q_\phi(\mathbf{z}^{(i)};\ \mathbf{x}^{(i)}) \parallel \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I}))}$$

Generative Adversarial Networks (GANs)

Generative models

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

- With LVMs, each vector **x** ~ P_θ(**x** | **z**) is assumed to be computed from an underlying hidden state vector **z**.

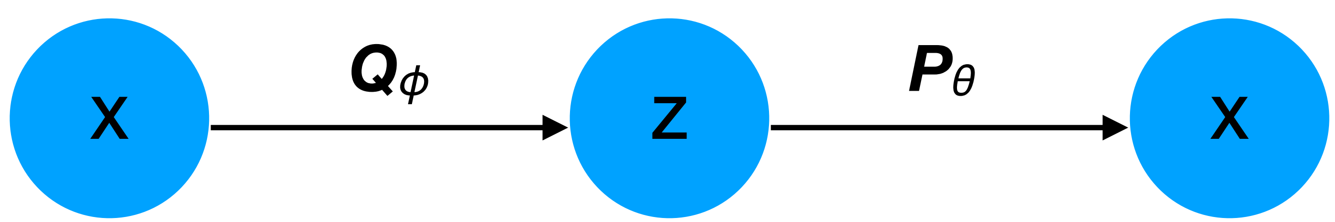


Generative models

- PCA is an example of a shallow LVM.
- VAEs are an example of a deep LVM.

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

$$\arg\max_{\theta,\phi} \log P_\theta(\mathbf{x}) = \arg\max_{\theta,\phi} \log \int_{\mathbf{z}} P_\theta(\mathbf{x}, \mathbf{z}) d\mathbf{z}$$



Generative models

- PCA is an example of a shallow LVM.
- VAEs are an example of a deep LVM.

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

$$\arg\max_{\theta,\phi} \log P_\theta(\mathbf{x}) = \arg\max_{\theta,\phi} \boxed{\log \int_{\mathbf{z}} P_\theta(\mathbf{x},\mathbf{z}) d\mathbf{z}}$$

Implicitly depends on Q_ϕ.

$$x \xrightarrow{Q_\phi} z \xrightarrow{P_\theta} x$$

Generative models

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

$$\arg\max_{\theta,\phi} \log P_\theta(\mathbf{x}) = \arg\max_{\theta,\phi} \log \int_{\mathbf{z}} P_\theta(\mathbf{x},\mathbf{z})d\mathbf{z}$$

Implicitly depends on Q_ϕ.

$$x \xrightarrow{Q_\phi} z \xrightarrow{P_\theta} x$$

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.
- In particular, the most prominent method (as of 2020) for generating novel data **x** is 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.

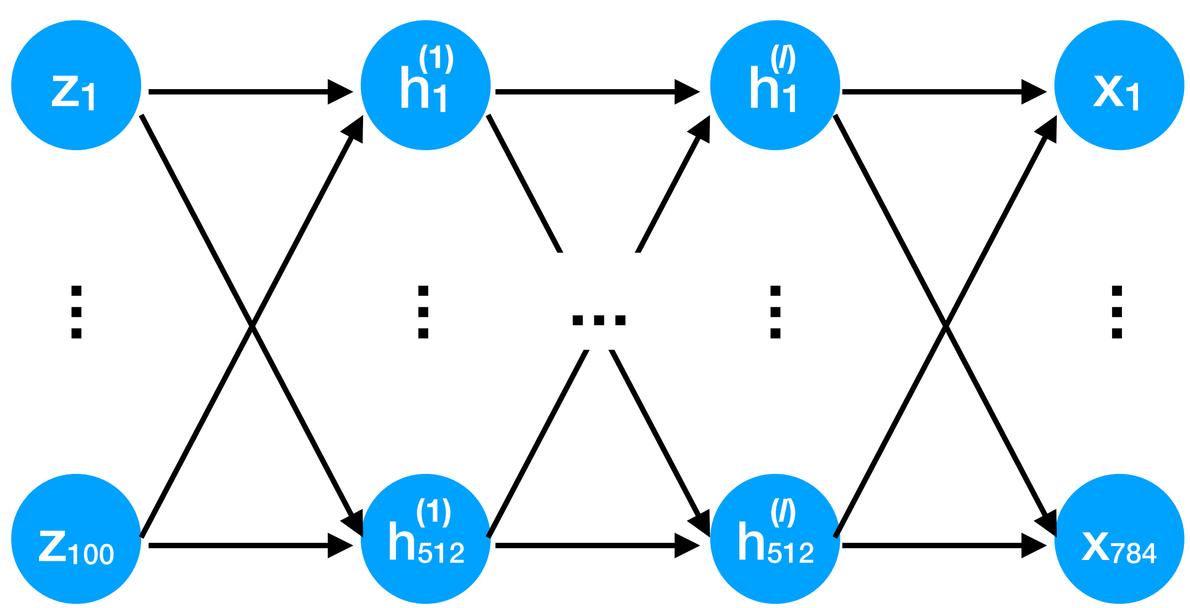
- Let $P_{\text{data}}(\mathbf{x})$ be the ground-truth data distribution.

- **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”.

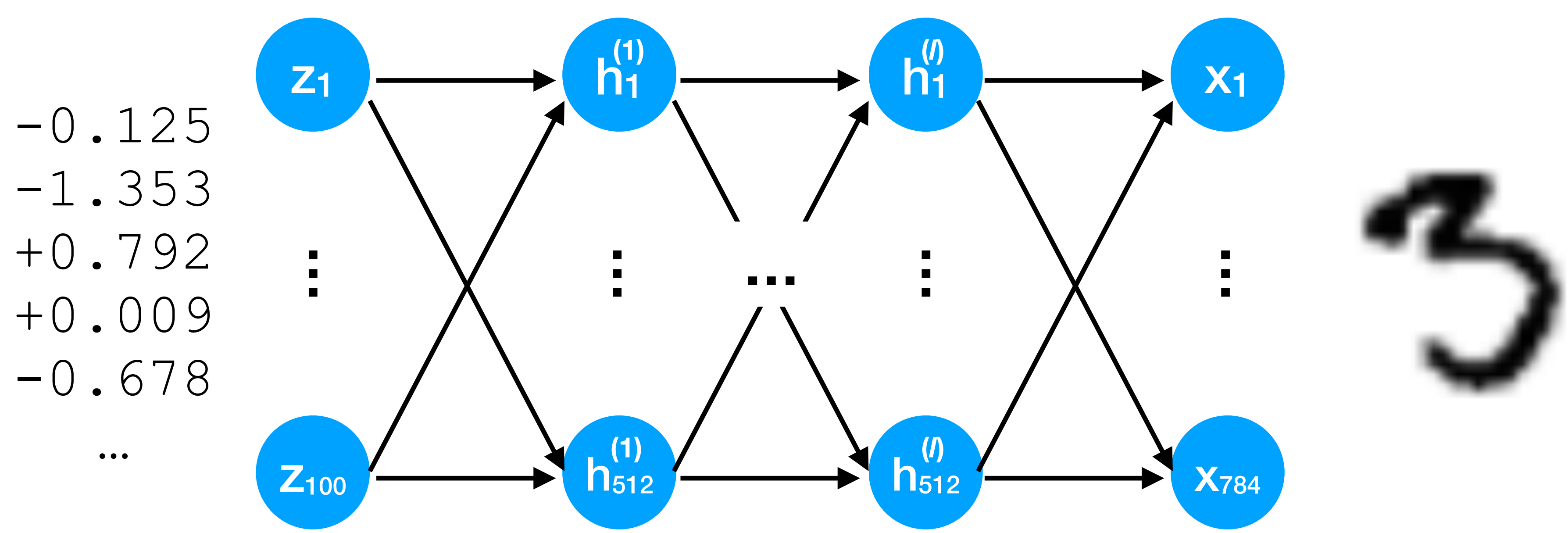
Generator *G*

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



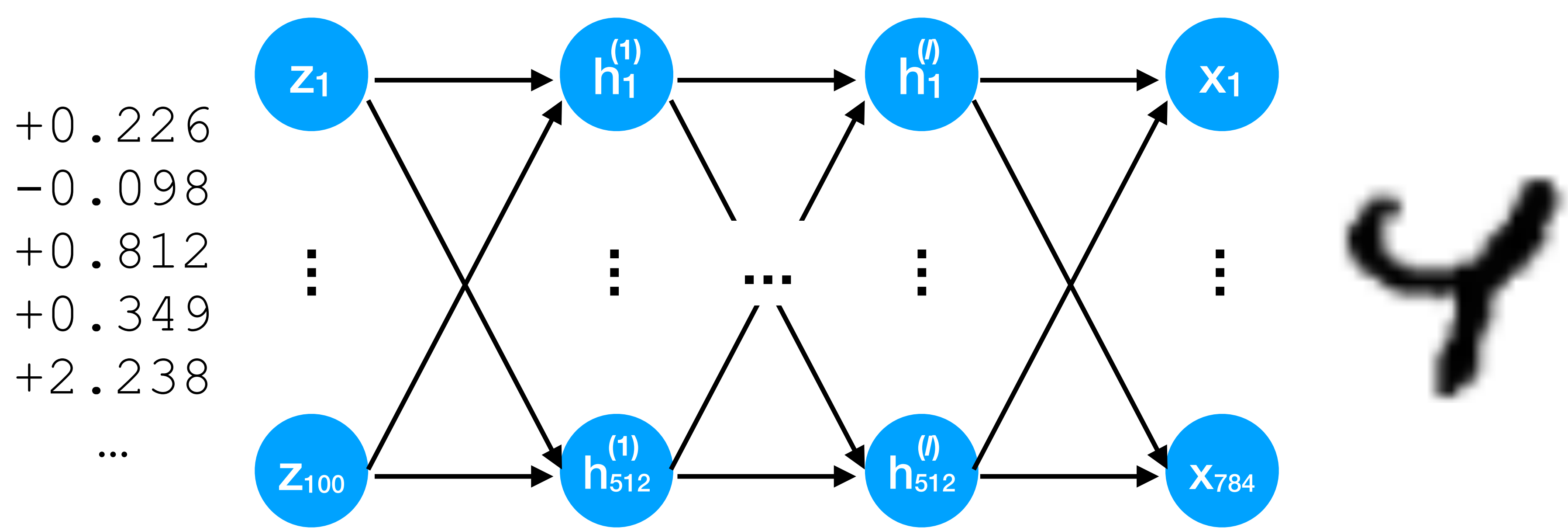
Generator *G*

- 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**).



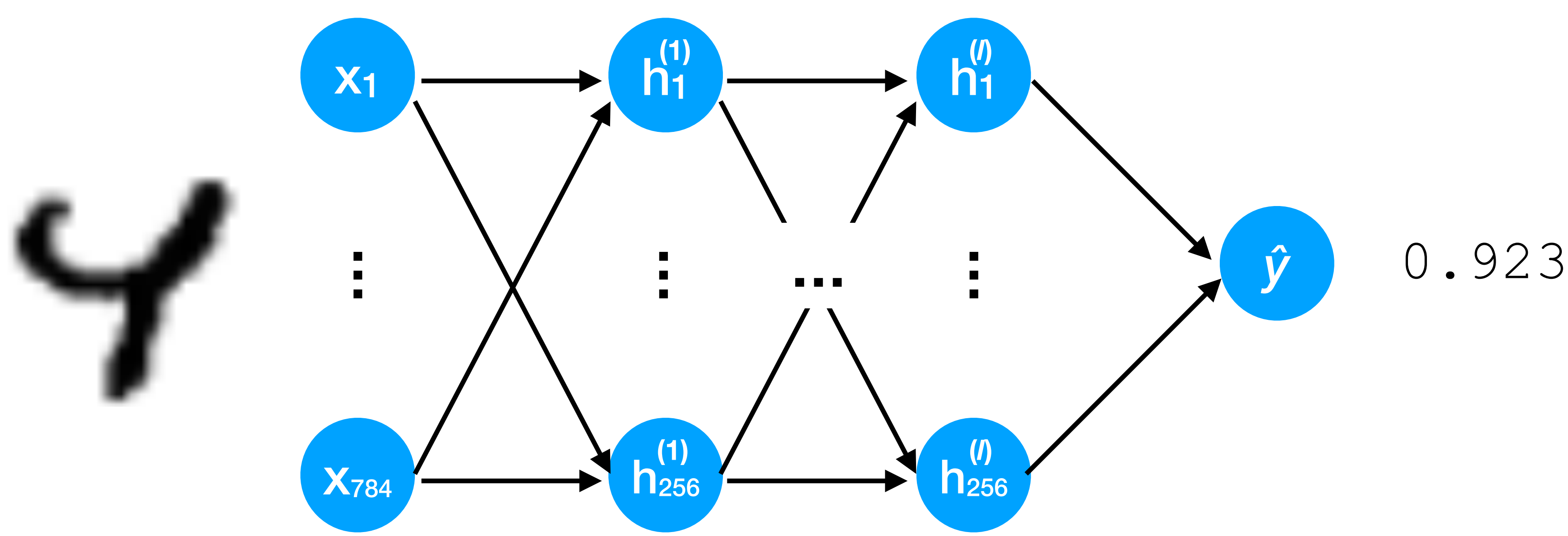
Generator G

- 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**).



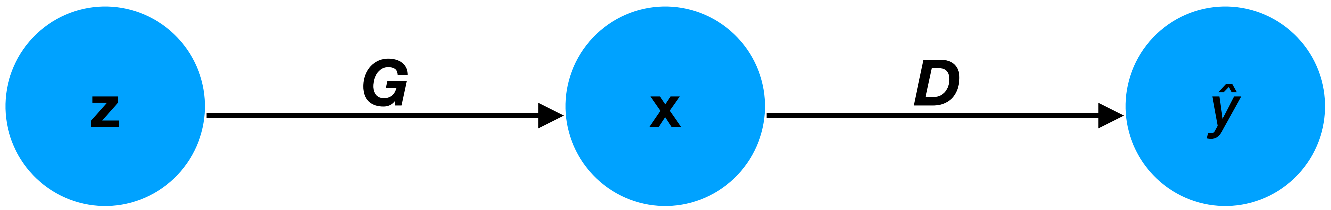
Discriminator *D*

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



Generative Adverarial Networks (GANs)

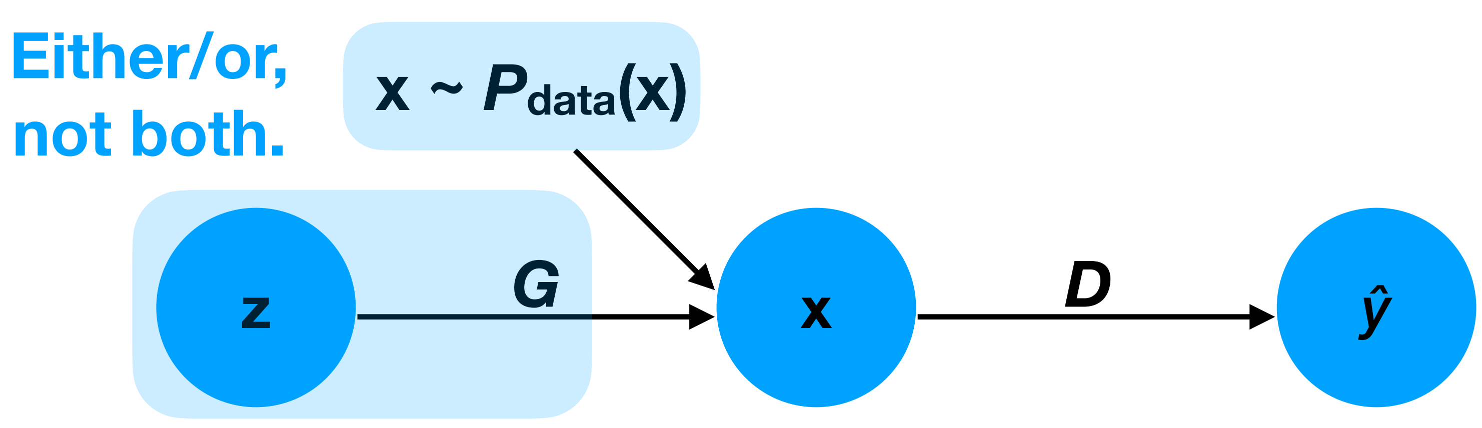
- Like VAEs, GANs are trained such that one component "feeds" to the other.



Generative Adverarial Networks (GANs)

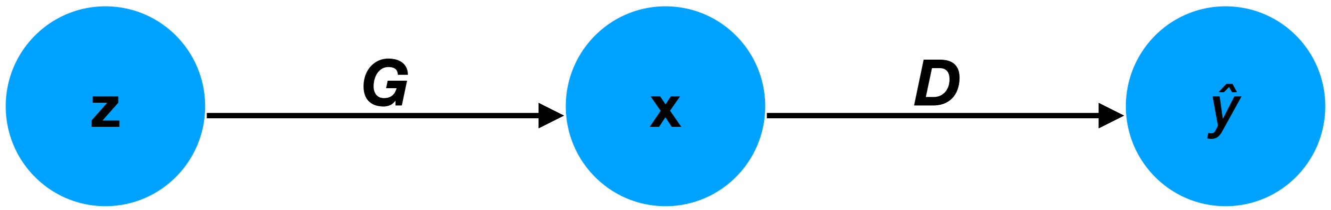
- Like VAEs, GANs are trained such that one component "feeds" to the other.

- In contrast to VAEs, the discriminator D is sometimes given a "fake" data vector $\mathbf{x}$ (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.

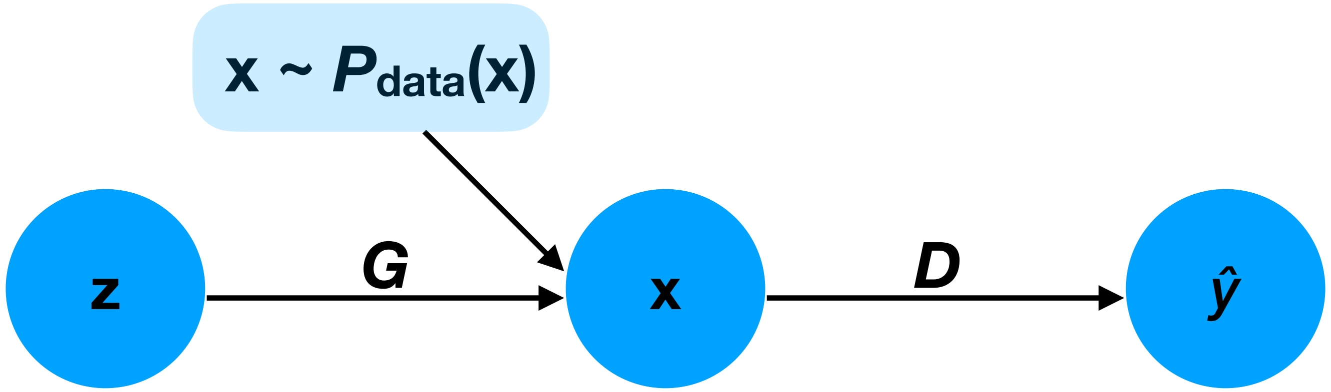


Generative Adverarial Networks (GANs)

- 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.

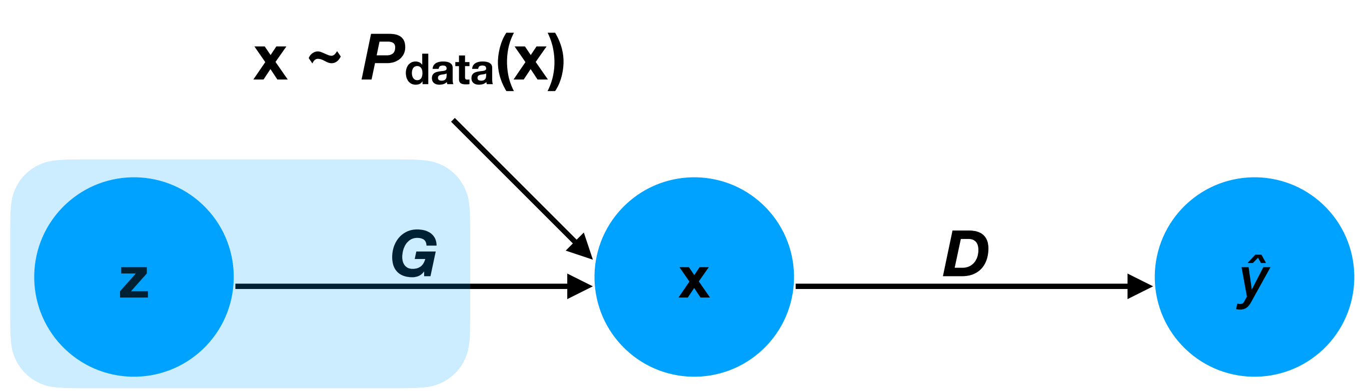


Generative Adverarial Networks (GANs)

- 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

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

- 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

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

- 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

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

- 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}.

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

Training GANs

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

  for k steps **do**

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Goodfellow et al. 2014

Exercises

https://cs230.stanford.edu/files/cs230exam_fall19_soln.pdf

Exercise 1

(2 points) The backpropagated gradient through a tanh non-linearity is always smaller or equal in magnitude than the upstream gradient. (Recall: if z = tanh(x) then $\frac{\partial z}{\partial x} = 1 - z^2$)

(i) True

(ii) False

Exercise 2

(2 points) Consider a trained logistic regression. Its weight vector is W and its test accuracy on a given data set is A. Assuming there is no bias, dividing W by 2 won't change the test accuracy.

(i) True

(ii) False

Exercise 3

(2 points) You're solving a binary classification task. The final two layers in your network are a ReLU activation followed by a sigmoid activation. What will happen?

Exercise 4

Consider a model trying to learn an encoding of some input $x \in \mathbb{R}$. The goal is to encode the input x using $z = w_1 x \in \mathbb{R}$, then accurately reconstruct the original x from the encoded representation using $\hat{x} = w_2 z \in \mathbb{R}$. Here, $(w_1, w_2) \in \mathbb{R} \times \mathbb{R}$. The model is trained with the squared reconstruction error:

$$L(W) = \frac{1}{n} \sum_{i=1}^{n} (x^{(i)} - w_2 w_1 x^{(i)})^2$$

(2 points) What is the set of solutions for w_1 and w_2 which makes loss zero?

(3 points) Does the loss have a saddle point? Where?