block b5

# block b5

## Autoencoders, Variational Autoencoders

# block b5

## Autoencoders, Variational Autoencoders

## Gene expression, scRNA-seq

# MCB128

## Final Project

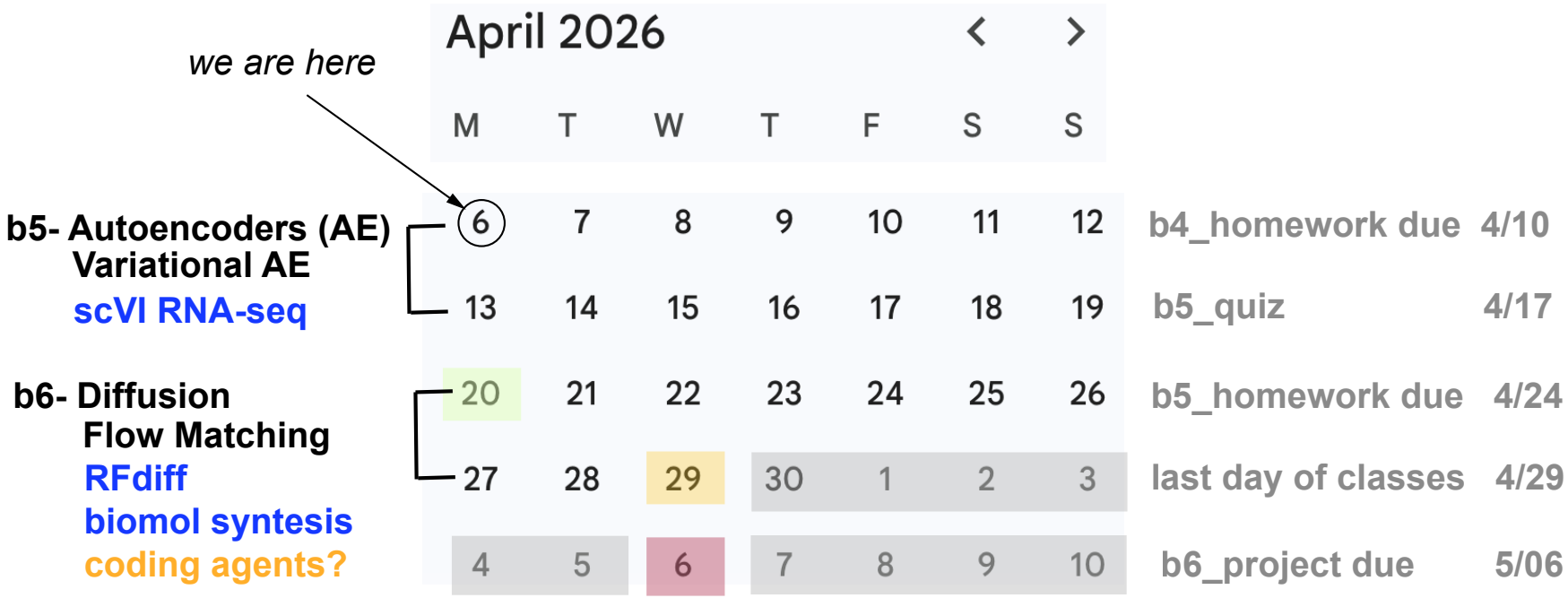
due May 6, 2026

---

**Instructions:**

- Project selection should be done by Monday 4/20, 2026
- Discuss project topics with the instructor and TFs as needed. In the end, you should send an email to the instructor (ccing the TFs) describing the project in order to obtain formal approval.
- This project counts as one homework (out of six total) towards 70% of your final grade.
- All tools are available to you...
- ...but the writing and reasoning should be in your own voice.
- You can discuss your project with everybody: TF, classmates, friends,...
- ...but projects are individual. No group projects.

---




4/20 **Project approval**

5/06 **Project deadline 11:59**

## 1. Topics

The topic should have some degree of originality, that is, not reproducing something identical provided with the course materials.

The type of projects is quite flexible:

(a) Theoretical: a follow up from class on a math aspect to expands on.

(b) Coding: Apply any of the methods described in class to some dataset of interest to you.

(c) Divulgation: A slide (or video) presentation of some paper or subject of interest, hopefully where you have reached some original understanding that you want to communicate to the world.

## 2. Development/Scope

(a) You need to take into account the limited amount of time provided. Don't do something too trivial, but do not embark on the equivalent of a PhD thesis.

(b) Depending on the nature of the project, it is ok to reach a certain point, and then describe what would have to be done next, given more time.

## 3. Format

The format is pretty flexible. As you do it, consider including some combination of the following items, some of which may apply to some types of projects but not to others

(a) Motivation that made you pick that project

(b) Approach that you are going to take

(c) Hypothesi(e)s that you plan to test

(d) Include at least one null hypothesis!

(e) Description of experiments performed

(f) Summary of results

(g) Interpretation

(h) Next to do (or not to do)

***Tell a story***

***be critical***

**Self-attention → Transformers → Language Models**

# Language Models:
# Encoders = Mask LMs
# Decoders = Autoregressive LMs

# AUTOENCODER

# AUTOENCODER

# =

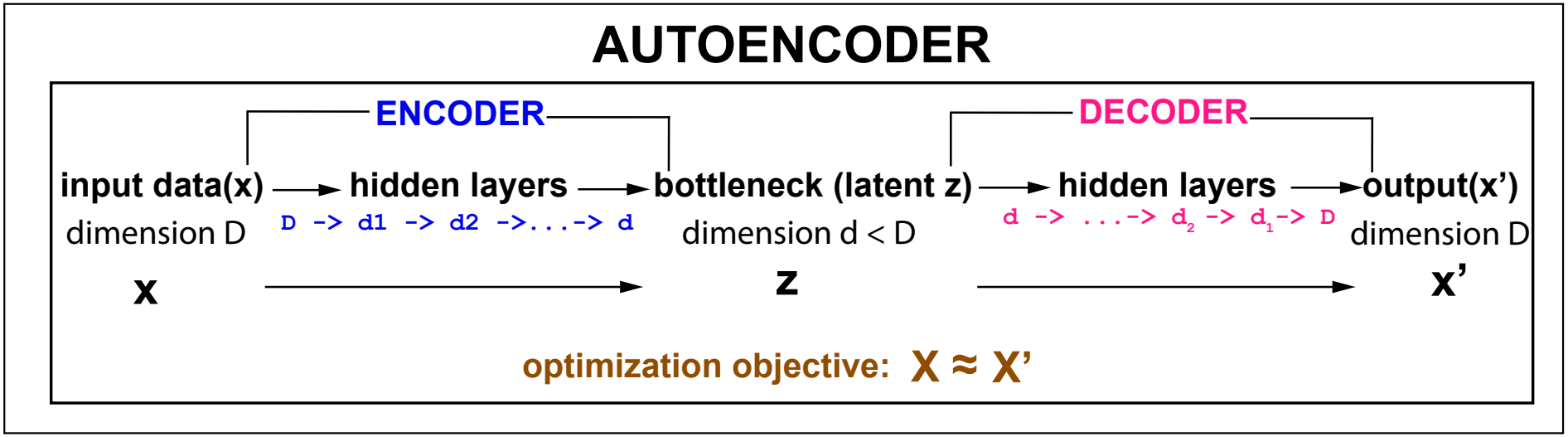
# COMPRESS REPRESENTION OF INPUT DATA

## Reducing the Dimensionality of Data with Neural Networks

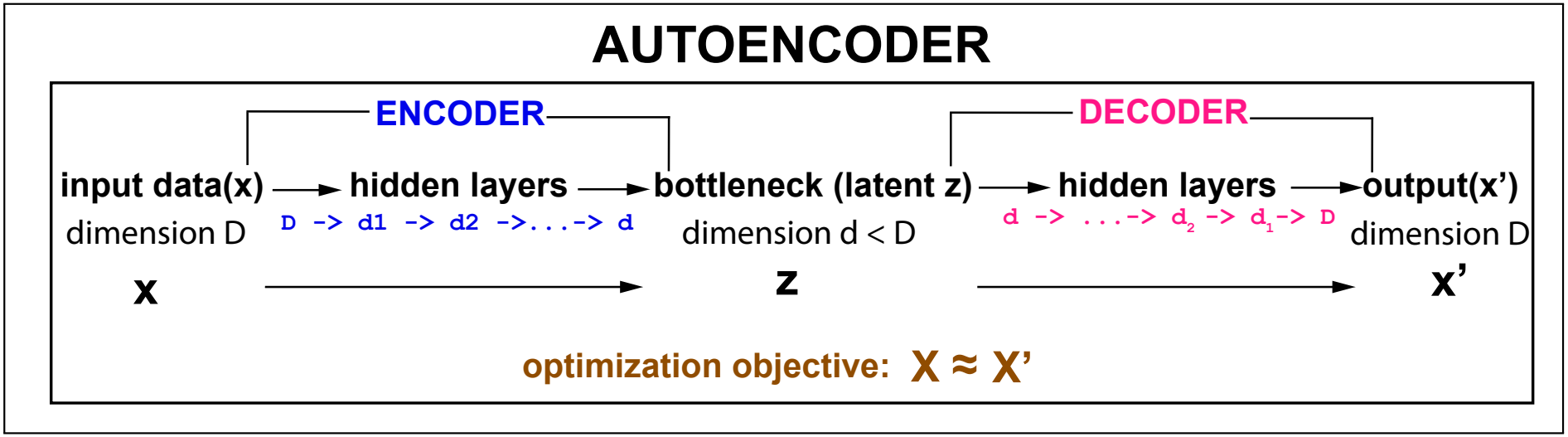
G. E. Hinton* and R. R. Salakhutdinov

Science 2006

High-dimensional data can be converted to low-dimensional codes by training a multilayer neural network with a small central layer to reconstruct high-dimensional input vectors. Gradient descent can be used for fine-tuning the weights in such "autoencoder" networks, but this works well only if the initial weights are close to a good solution. We describe an effective way of initializing the weights that allows deep autoencoder networks to learn low-dimensional codes that work much better than principal components analysis as a tool to reduce the dimensionality of data.


AUTOENCODER
ENCODER
DECODER
input data(x)
hidden layers
bottleneck (latent z)
hidden layers
output(x')
dimension D
D -> d1 -> d2 ->...-> d
dimension d < D
d -> ...-> d2 -> d1-> D
dimension D
x
z
x'
optimization objective: X ≈ X'

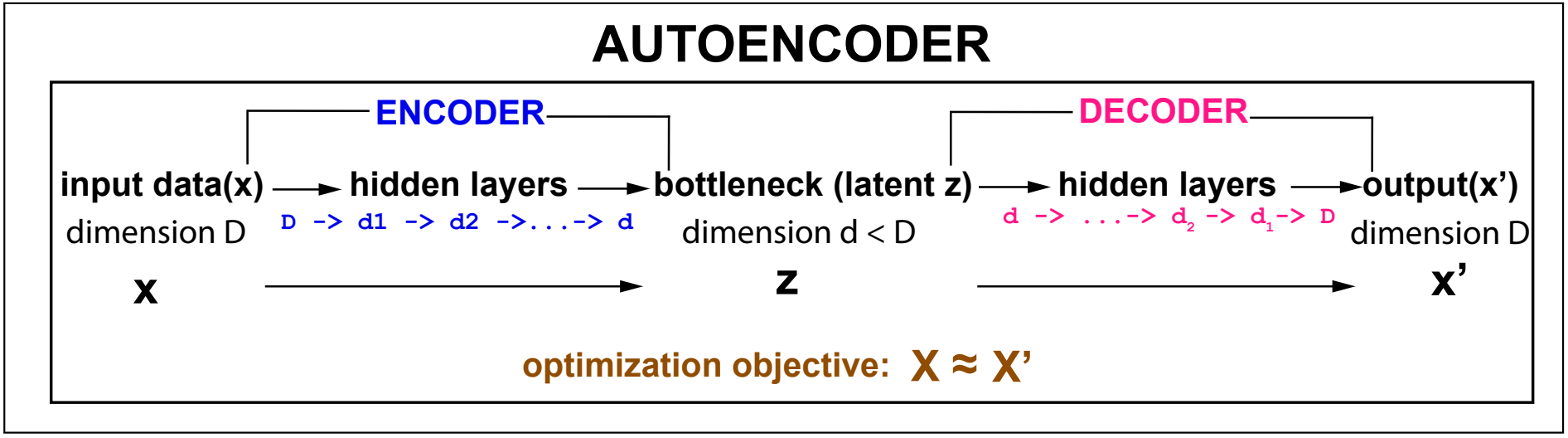
$$D \gg d$$




**Encoder** $D \rightarrow d$ (non-linear)

**Bottleneck** Latent variable z[d]

**Decoder** $d \rightarrow D$ (non-linear)



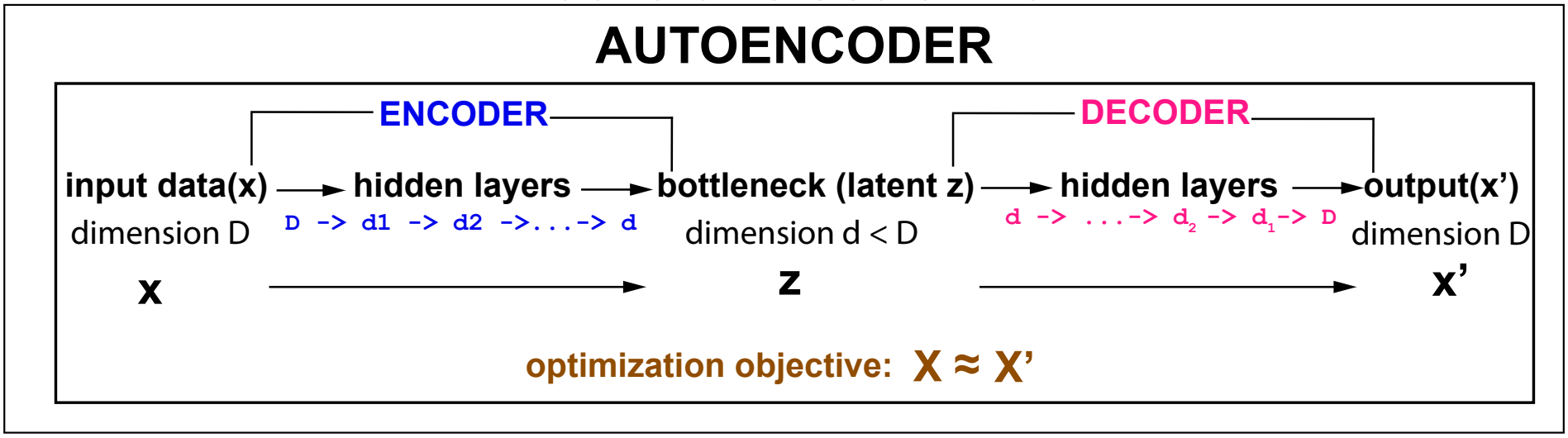

Autoencoder learning:

**Unsupervised**

The learning taks is simply to achive:

$$\text{output} = x'[D] \approx x[D]$$

# What is this usefull for?



What is this usefull for?

# Autoencoders for communication

*"A mathematical theory of communication", Shannon, 1948*

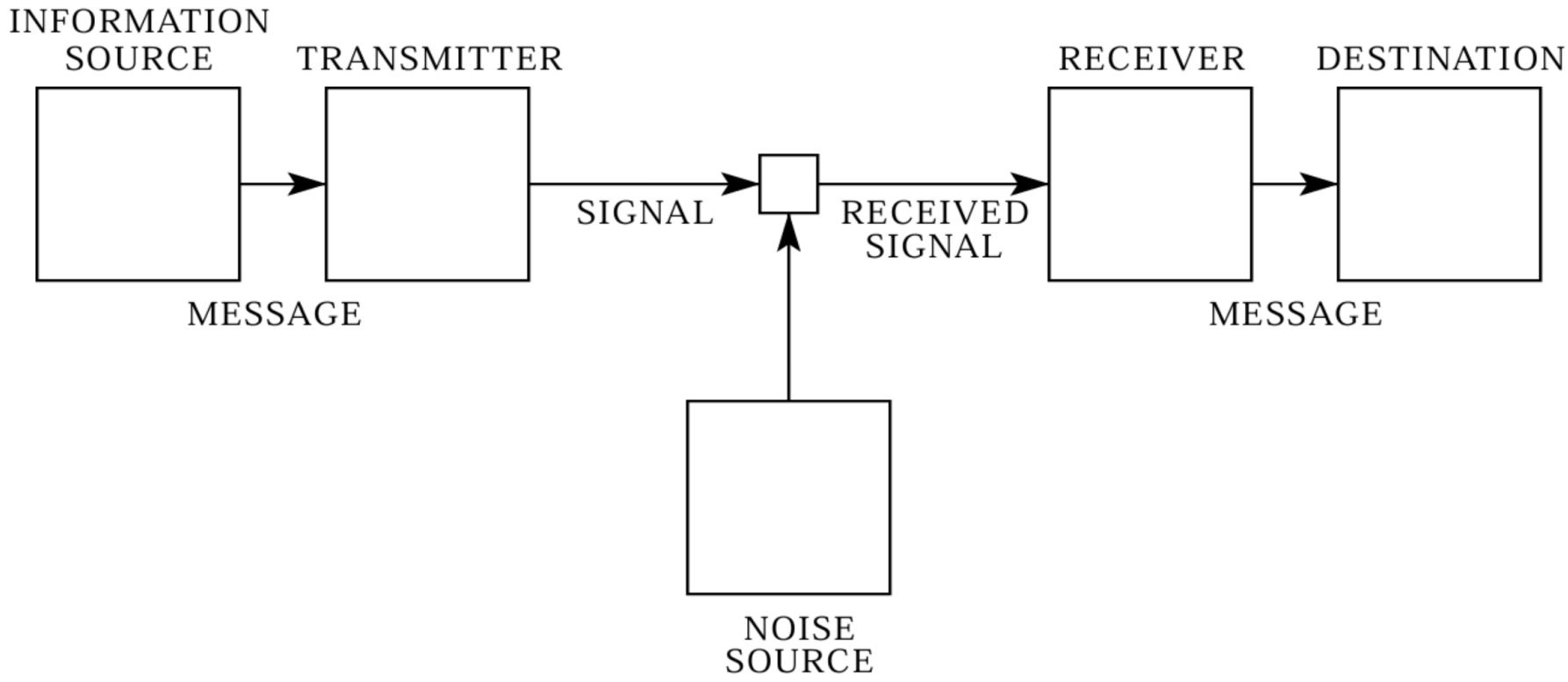



Fig. 1 — Schematic diagram of a general communication system.

# What is this usefull for?


$\{t_n\}_{n=1}^N$ → Learning algorithm → $\mathbf{w}$ → $\mathbf{w}$ → $\{\hat{t}_n\}_{n=1}^N$

$\{\mathbf{x}_n\}_{n=1}^N$ ↑ Learning algorithm

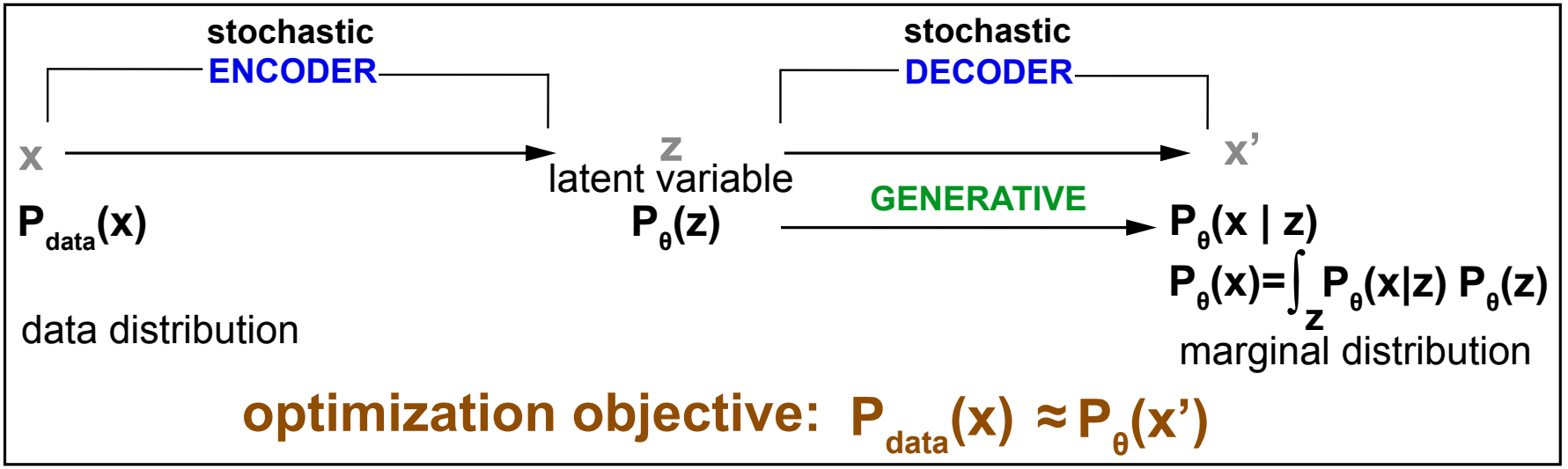
$\{\mathbf{x}_n\}_{n=1}^N$ ↑ $\mathbf{w}$

$P(\mathbf{w} \mid \{\mathbf{x}_n\})$

$P(\{\mathbf{x}_n\} \mid \mathbf{w})$

The latent variable plays the rol of a communication channel

# Variational Autoencoders (VAEs)




VAE are **Generative models**

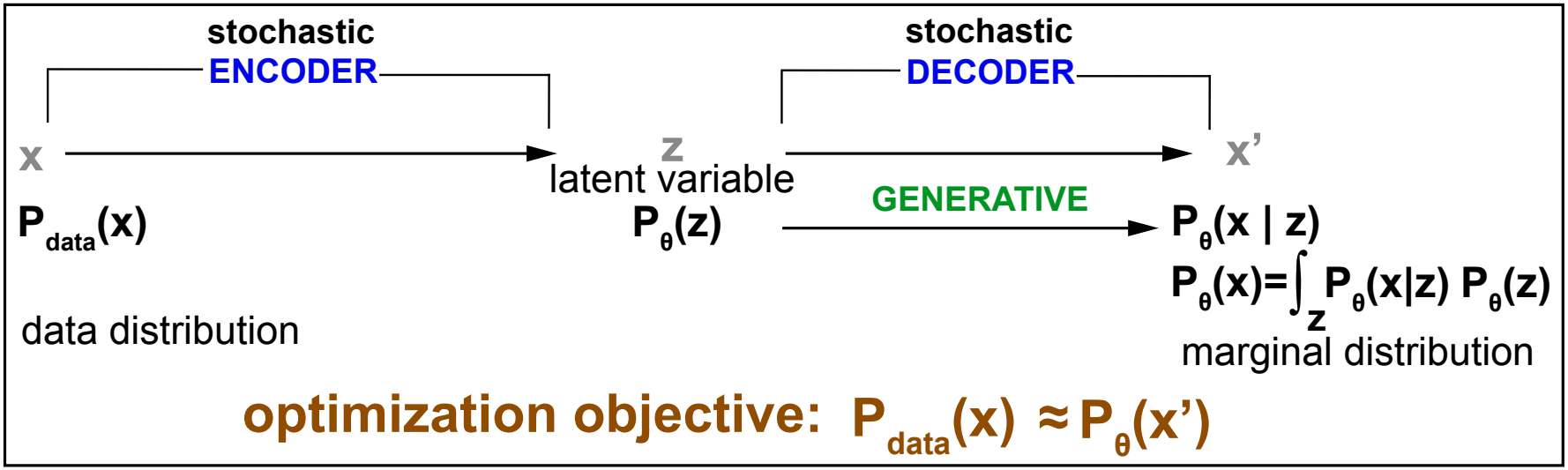
Introduces

$$P_\theta(xz) = P_\theta(x \mid z)P_\theta(z)$$

Such that

$$P_\theta(x) = \sum P_\theta(xz)$$

# Variational Autoencoders (VAEs)



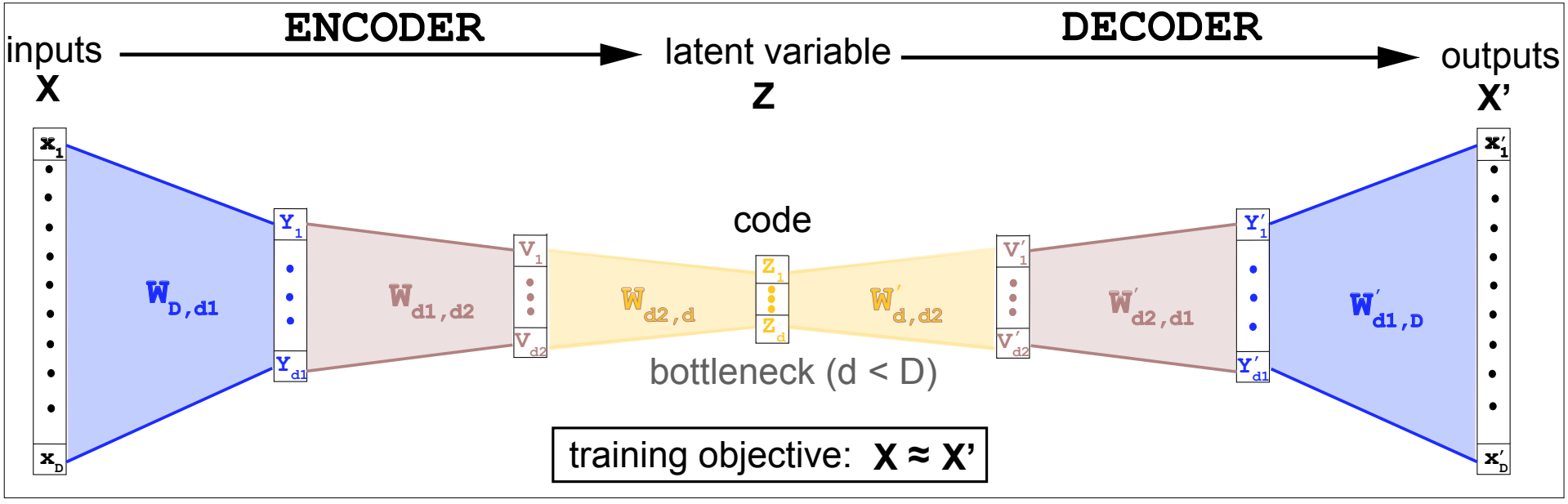

VAE are **Generative models**

And then we can sample the latent variable from

$$P_\theta(z \mid x) = \frac{P_\theta(xz)}{P_\theta(x)}$$

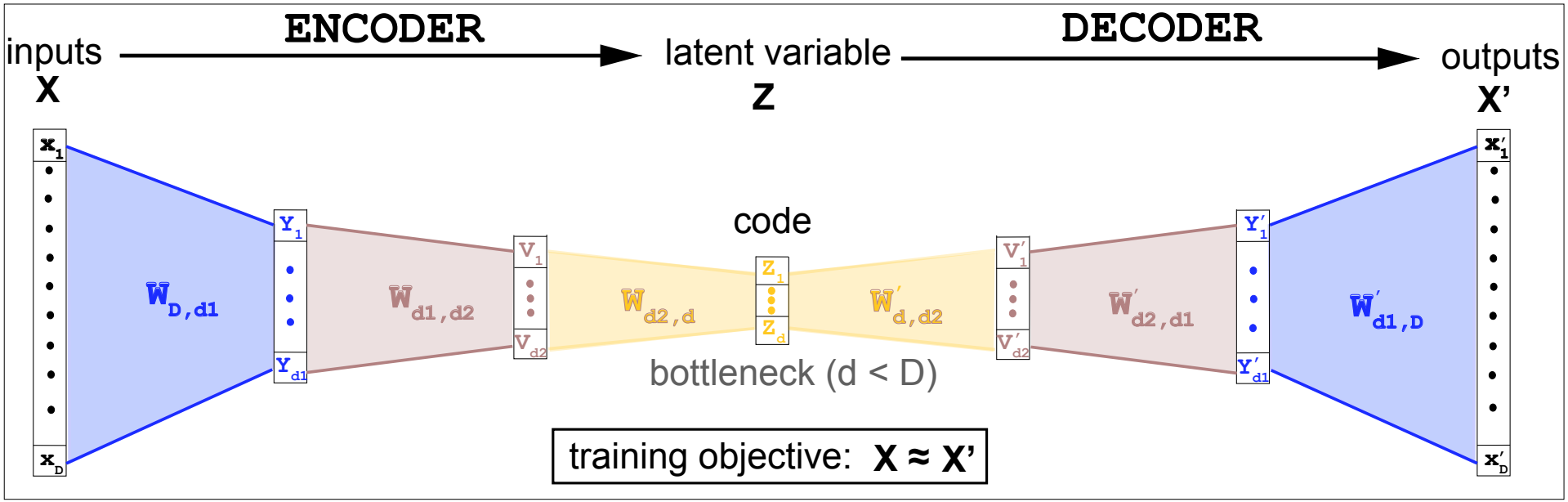
HARD!... ENCODER to the rescue!!!

## AUTOENCODER




**Encoder:** $y[d1] = RELU(x[D]@W^1[D, d1])$

**Bottleneck:** $z[d]$




## Decoder. Last layer activation:

1. INPUTs are arbitrary continuous $[x \in R^D]$: no activation

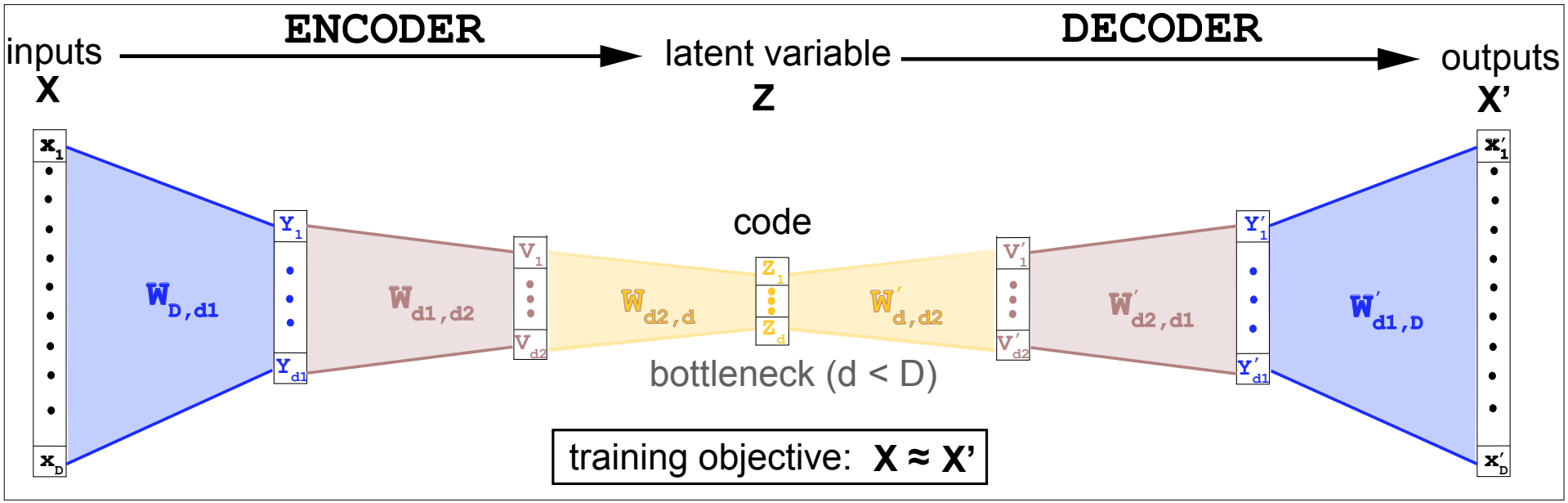
$$x' = \text{output} \in R^D$$

2. INPUTs are values $[x \in [0,1]]]$: sigmoidal activation

$$x'_i = \frac{1}{1 + e^{-\text{output}_i}} \in [0,1]$$

3. INPUTs is a categorical distribution $[x \in [0,1]$ and $\sum_i x_i = 1]$: softmax activation

$$x'_i = \frac{e^{-\text{output}_i}}{\sum_{i'}^{D} e^{-\text{output}_{i'}}}$$




## The AE Loss

1. INPUTs continuous[$x \in R^D$]: **MSE**

$$MSE(x, x') = \frac{1}{D} \sum_{i=1}^{D} (x_i - x'_i)^2$$

2. INPUTs categorical distribution[$x \in [0, 1]$ and $\sum_i x_i = 1$]: **CrossEntropy**

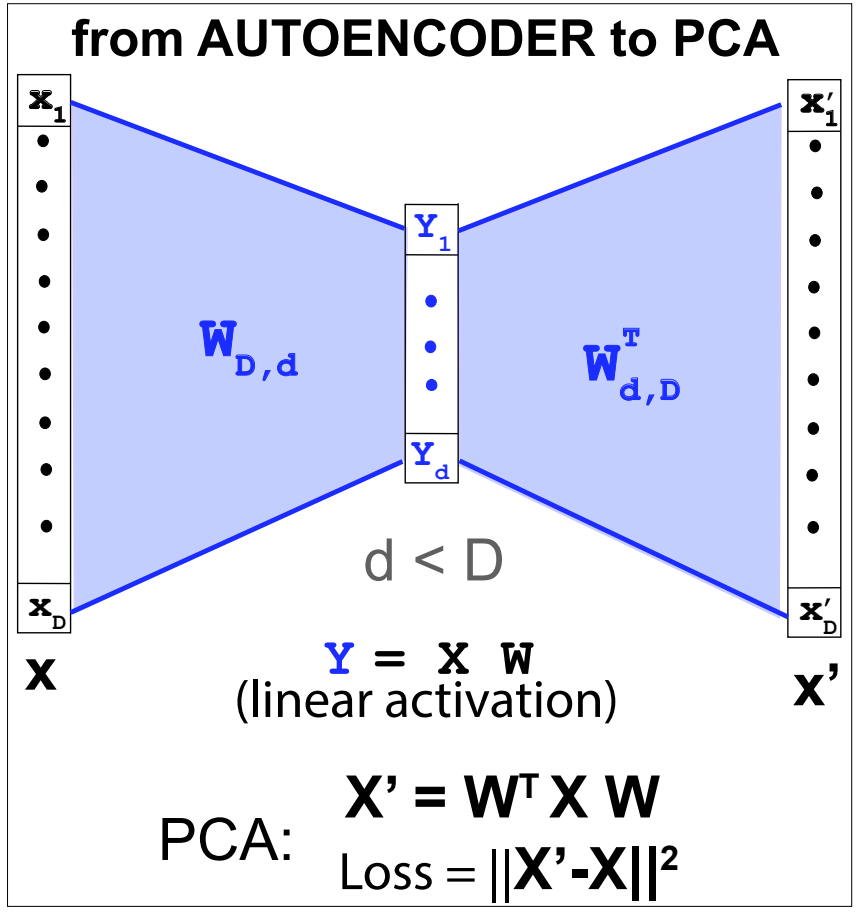
$$CrossEntropy(x, x') = -\frac{1}{D} \sum_{i=1}^{D} x_i \log(x'_i)$$

# Connection of AE to PCA

## Reducing the Dimensionality of Data with Neural Networks

G. E. Hinton* and R. R. Salakhutdinov Science 2006

High-dimensional data can be converted to low-dimensional codes by training a multilayer neural network with a small central layer to reconstruct high-dimensional input vectors. Gradient descent can be used for fine-tuning the weights in such "autoencoder" networks, but this works well only if the initial weights are close to a good solution. We describe an effective way of initializing the weights that allows deep autoencoder networks to learn low-dimensional codes that work much better than principal components analysis as a tool to reduce the dimensionality of data.



# GENEEX Autoencoder (National Cancer Institute)



## Gene Expression Autoencoder - P1B1 (GENEEX AUTOENCODER)



Builds a sparse autoencoder that can compress the expression profile of a sample of gene expression data into a low-dimensional vector.

### Description and Impact

#### USER COMMUNITY

Researchers interested in the following topics:

- Primary: Cancer biology data modeling
- Secondary: Machine learning, bioinformatics, computational biology

#### IMPACT

Offers an autoencoder to collapse high-dimensional expression profiles into low-dimensional vectors without significant loss of information.

#### HYPOTHESIS/OBJECTIVE

The objective was to build a sparse autoencoder that can compress high dimensional gene expression profiles into low dimensional vectors without much loss of information. Reducing the number of features can lessen processing time and overfitting of learning tasks.

# Gene Expression Profiles from single-cell sequencing (RNA-seq)


| | gene 1 | gene 2 | | | | gene G |
|---|---|---|---|---|---|---|
| cell 1 | | | | | | |
| cell 2 | | | | | | |
| | | | | | | |
| | | | counts[c,g] | | | |
| | | | | | | |
| cell C | | | | | | |

different libraries have different sequencing depths
some samples may have more RNA than others

$$\text{normalized\_counts}[c,g] = \frac{\text{counts}[c,g]}{\sum_{g'} \text{counts}[c,g']}$$

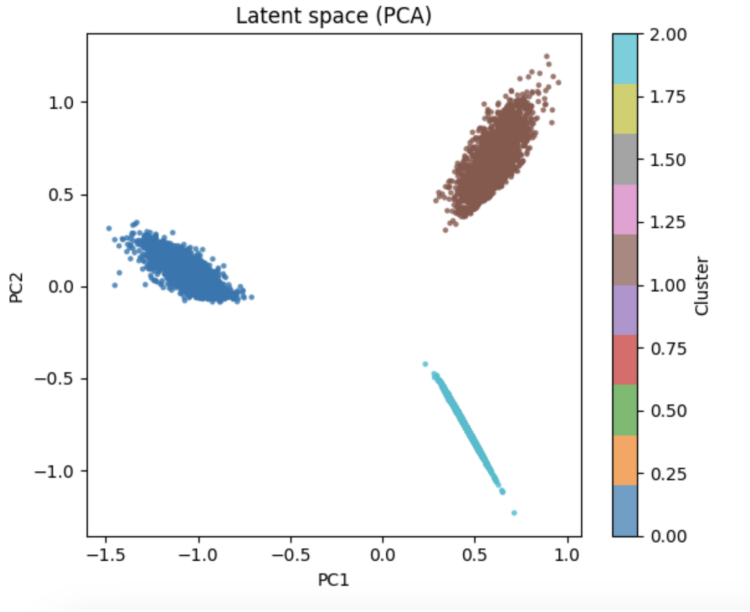
$$\text{log-normalized}[c,g] = \log\left(1 + \text{normalized\_counts}[c,g]\right)$$

# GENEEX AutoEncoder

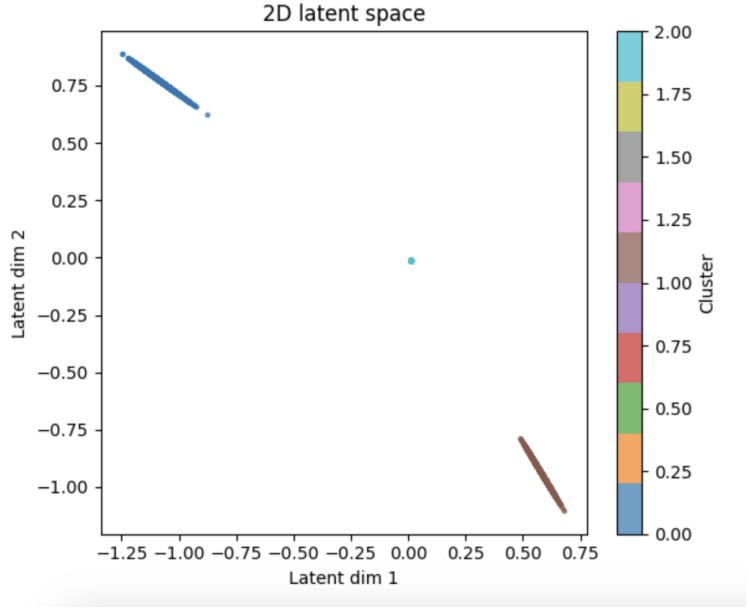
```python
class P1B1GeneExpressionAE(nn.Module):
    def __init__(
        self,
        input_dim,        # number of genes (features)
        h1=2048,
        h2=1024,
        h3=512,
        latent_dim=128,
        dropout=0.2,
    ):
        super().__init__()

        # ------- Encoder -------
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, h1),
            nn.ReLU(),
            nn.Dropout(dropout),

            nn.Linear(h1, h2),
            nn.ReLU(),
            nn.Dropout(dropout),

            nn.Linear(h2, h3),
            nn.ReLU(),
            nn.Dropout(dropout),

            nn.Linear(h3, latent_dim),
        )

        # ------- Decoder (mirrored) -------
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, h3),
            nn.ReLU(),
            nn.Dropout(dropout),

            nn.Linear(h3, h2),
            nn.ReLU(),
            nn.Dropout(dropout),

            nn.Linear(h2, h1),
            nn.ReLU(),
            nn.Dropout(dropout),

            nn.Linear(h1, input_dim)   # linear output for real-valued expression
        )

    def forward(self, x):
        z = self.encoder(x)
        x_hat = self.decoder(z)
        return x_hat, z
```

# GENEEX autoencoder

(a) Latent dimension 128




(b) Latent dimension 2




3 clusters of 2,000 cells each
each cell has 1,000 genes
each cluster over expresses 1/3 of the genes

# the (great) reference

Foundations and Trends® in Machine Learning

# An Introduction to Variational Autoencoders

---

**Suggested Citation:** Diederik P. Kingma and Max Welling (2019), "An Introduction to Variational Autoencoders", Foundations and Trends® in Machine Learning: Vol. xx, No. xx, pp 1–18. DOI: 10.1561/XXXXXXXXX.

**Diederik P. Kingma**
Google
durk@google.com

**Max Welling**
Universiteit van Amsterdam, Qualcomm
mwelling@qti.qualcomm.com

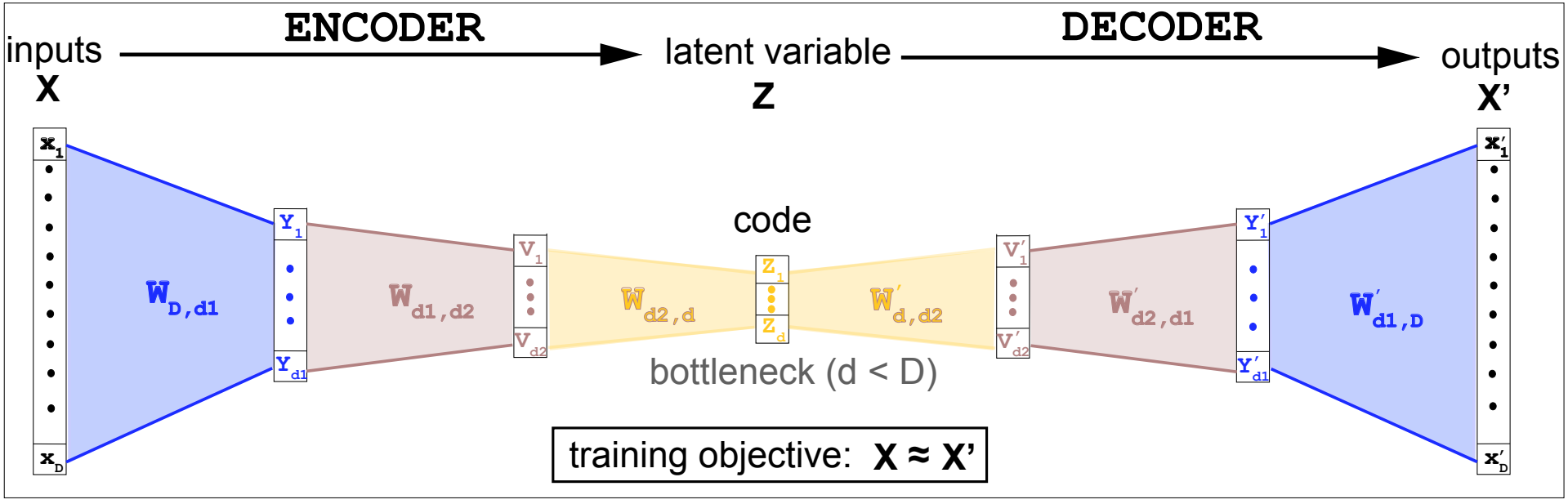
# Generative/Discriminative Modeling

## 1.1 Motivation

One major division in machine learning is generative versus discriminative modeling. While in discriminative modeling one aims to learn a predictor given the observations, in generative modeling one aims to solve the more general problem of learning a joint distribution over all the variables. A generative model simulates how the data is generated in the real world. "Modeling" is understood in almost every science as unveiling this generating process by hypothesizing theories and testing these theories through observations. For instance, when meteorologists model the weather they use highly complex partial differential equations to express the underlying physics of the weather. Or when an astronomer models the formation of galaxies s/he encodes in his/her equations of motion the physical laws under which stellar bodies interact. The same is true for biologists, chemists, economists and so on. Modeling in the sciences is in fact almost always generative modeling.

There are many reasons why generative modeling is attractive. First, we can express physical laws and constraints into the generative process while details that we don't know or care about, i.e. nuisance variables, are treated as noise. The resulting models are usually highly intuitive

## AUTOENCODER




**Encoder:** $y[d1] = f(x[D]@W^1[D, d1])$

**Bottleneck:** $z[d]$ with $d < D$ undercomplete AE

# Sparse Autoencoders

Article

https://doi.org/10.1038/s41592-025-02836-7

## InterPLM: discovering interpretable features in protein language models via sparse autoencoders

Received: 26 November 2024

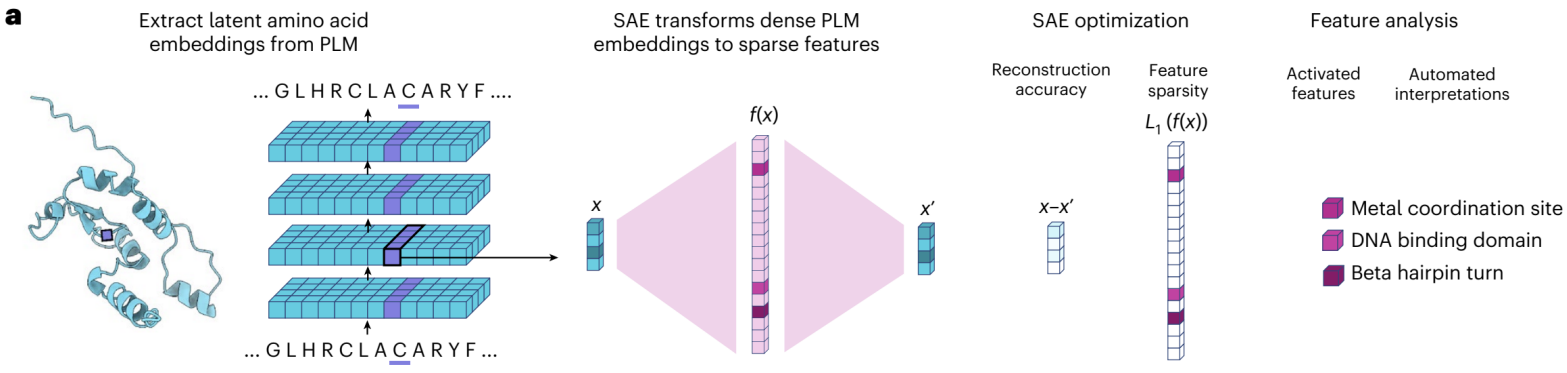
Accepted: 20 August 2025

Published online: 29 September 2025

Check for updates

Elana Simon & James Zou

Despite their success in protein modeling and design, the internal mechanisms of protein language models (PLMs) are poorly understood. Here we present a systematic framework to extract and analyze interpretable features from PLMs using sparse autoencoders. Training sparse autoencoders on ESM-2 embeddings, we identify thousands of interpretable features highlighting biological concepts including binding sites, structural motifs and functional domains. Individual neurons show

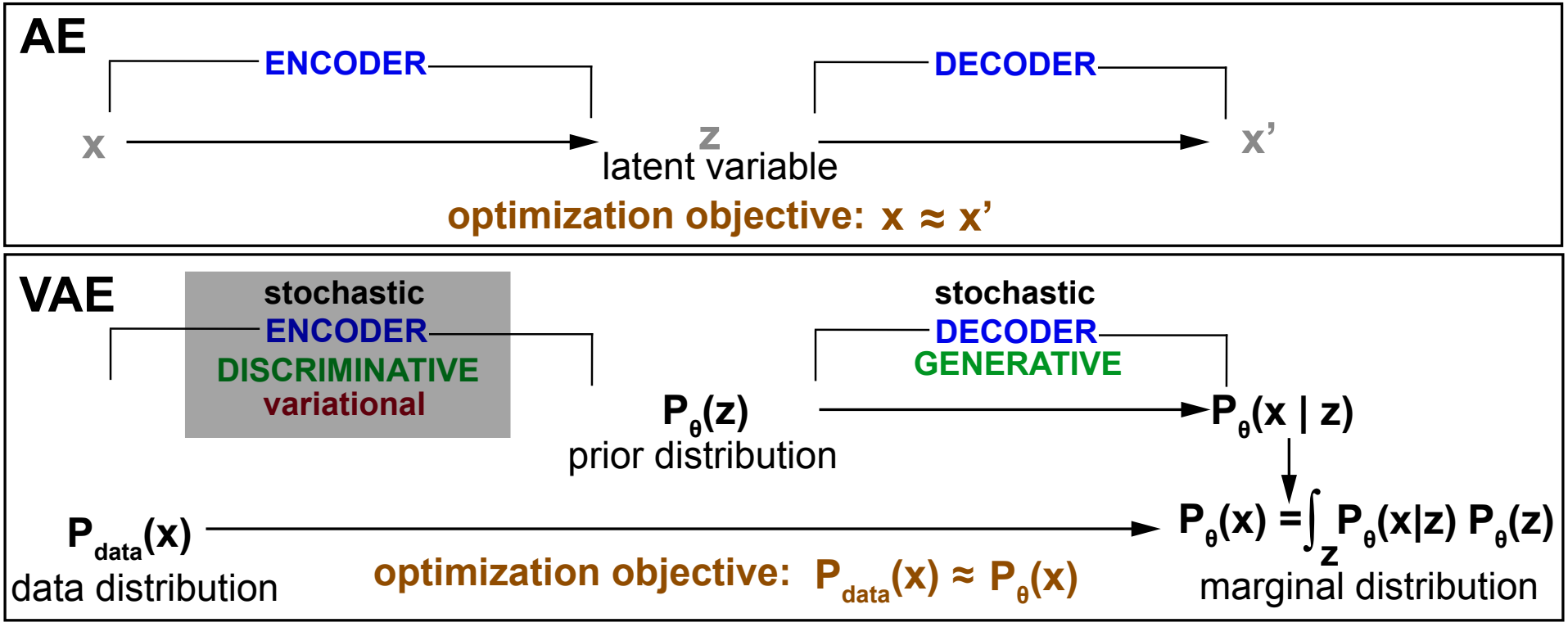



$$\sum_i (x_i - x_i')^2 + \sum_i |f(x)\_i|$$

**d >> D (d = 32xD)** **overcomplete AE**

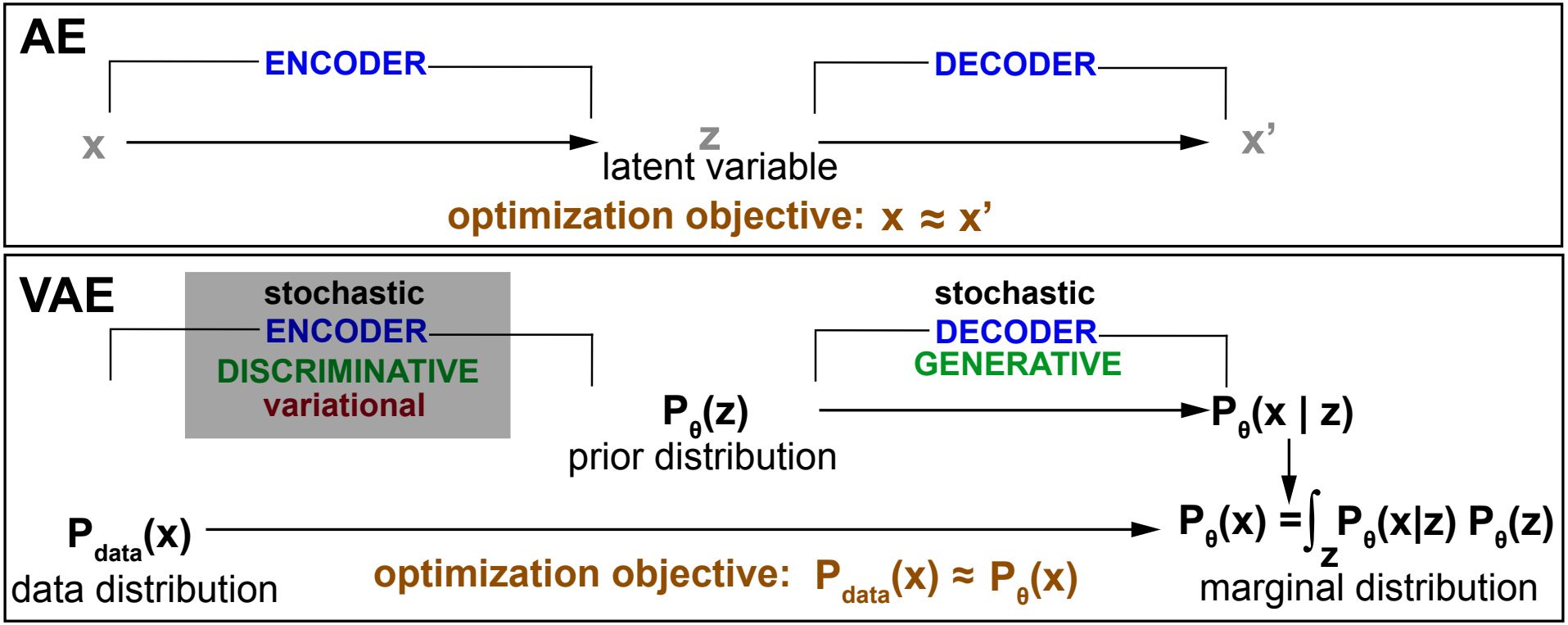
**sparcity by using a L1 reg loss on activations f()**

AutoEncoder (AE)




Variational AutoEncoder (VAE)

AutoEncoder (AE)

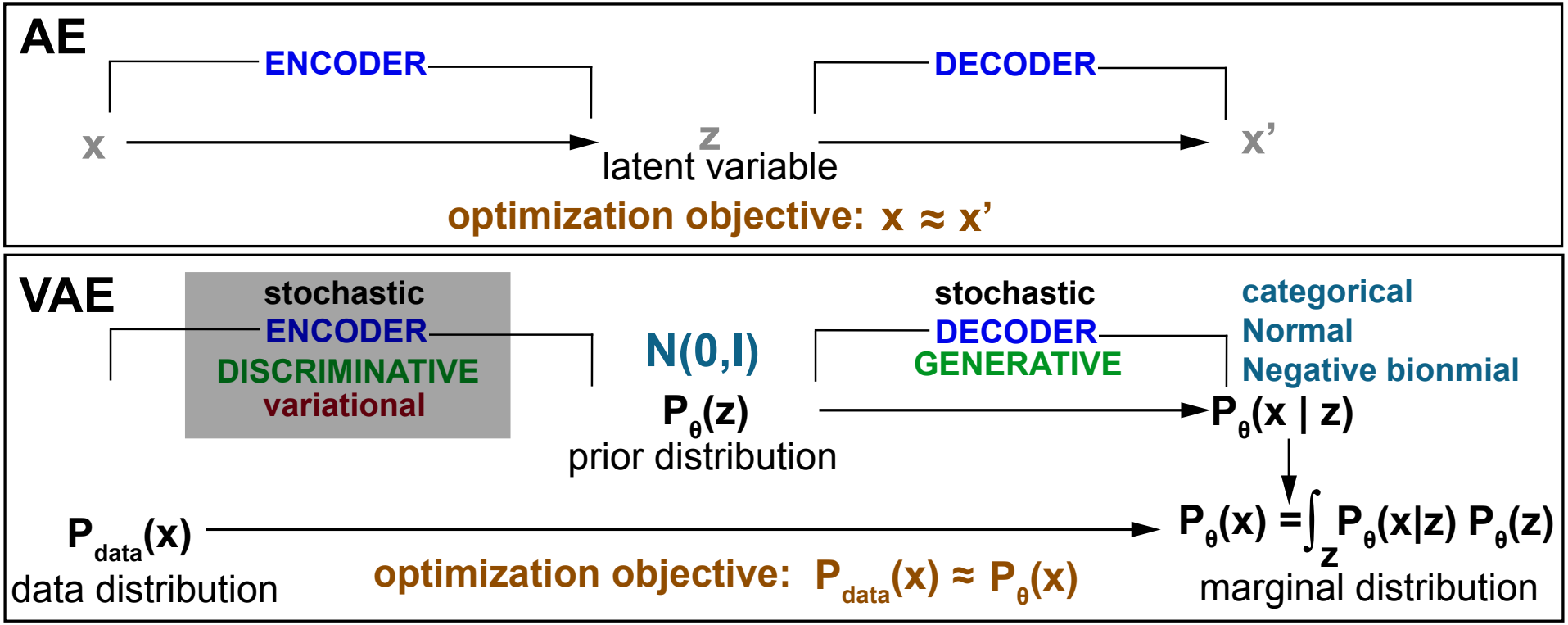



Variational AutoEncoder (VAE)

**The VAE DECODER is a generative model**

$$P_\theta(xz) = P_\theta(x \mid z)\, P_\theta(z)$$

AutoEncoder (AE)

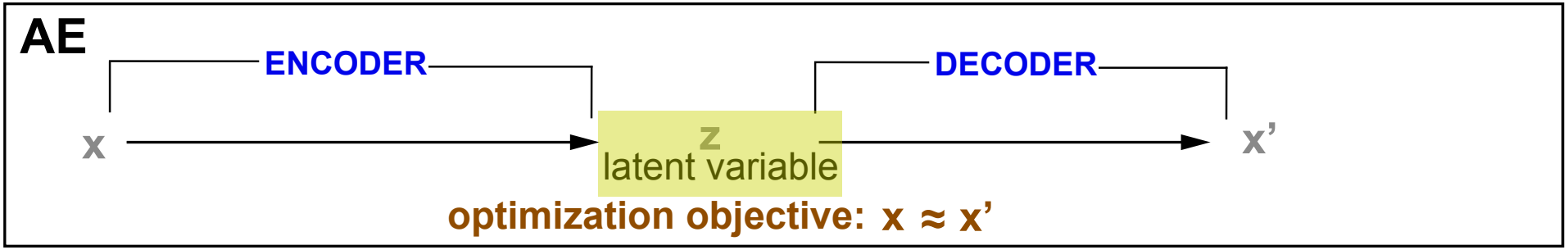


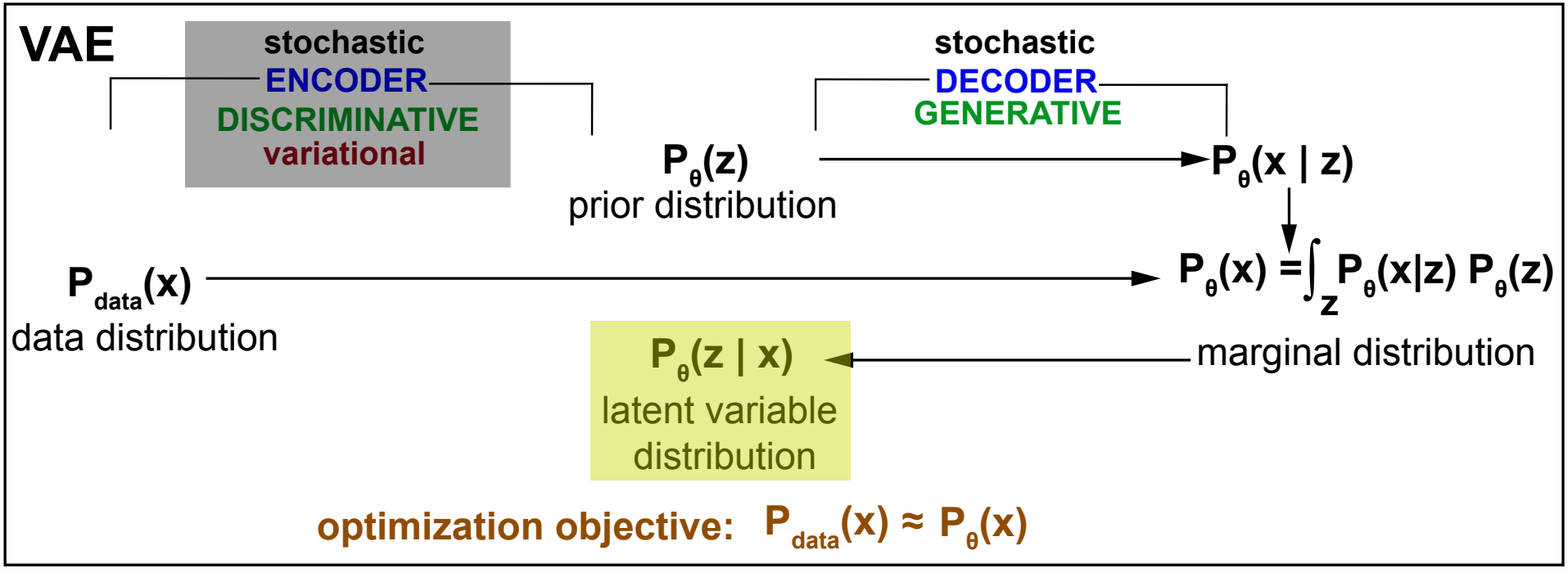

Variational AutoEncoder (VAE)

**The VAE DECODER is a generative model**

$$P_\theta(xz) = P_\theta(x \mid z)\, P_\theta(z)$$

AutoEncoder (AE)



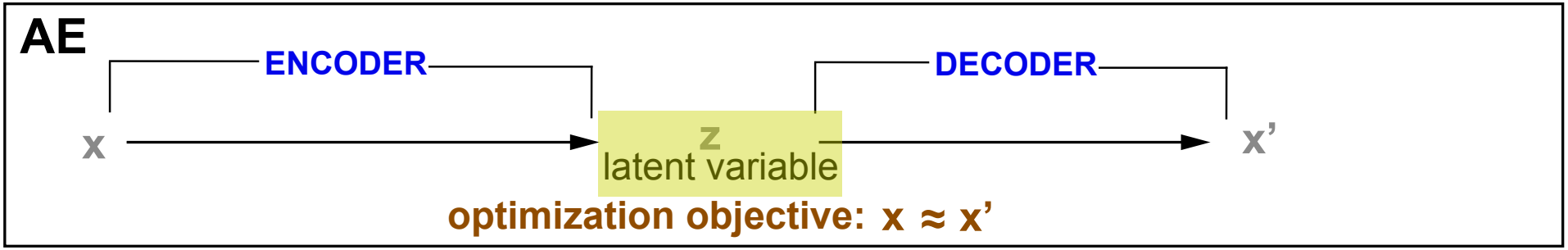



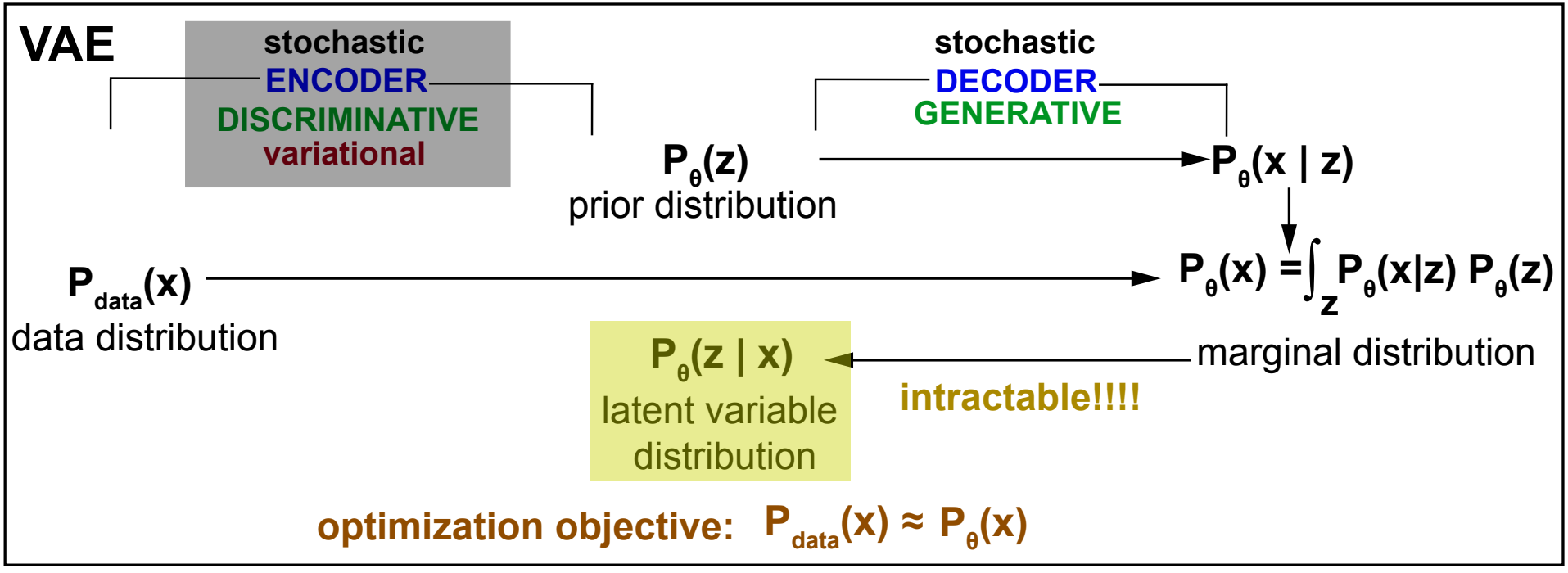

Variational AutoEncoder (VAE)

**The VAE DECODER is a generative model**

$$P_\theta(xz) = P_\theta(x \mid z)\, P_\theta(z)$$

AutoEncoder (AE)



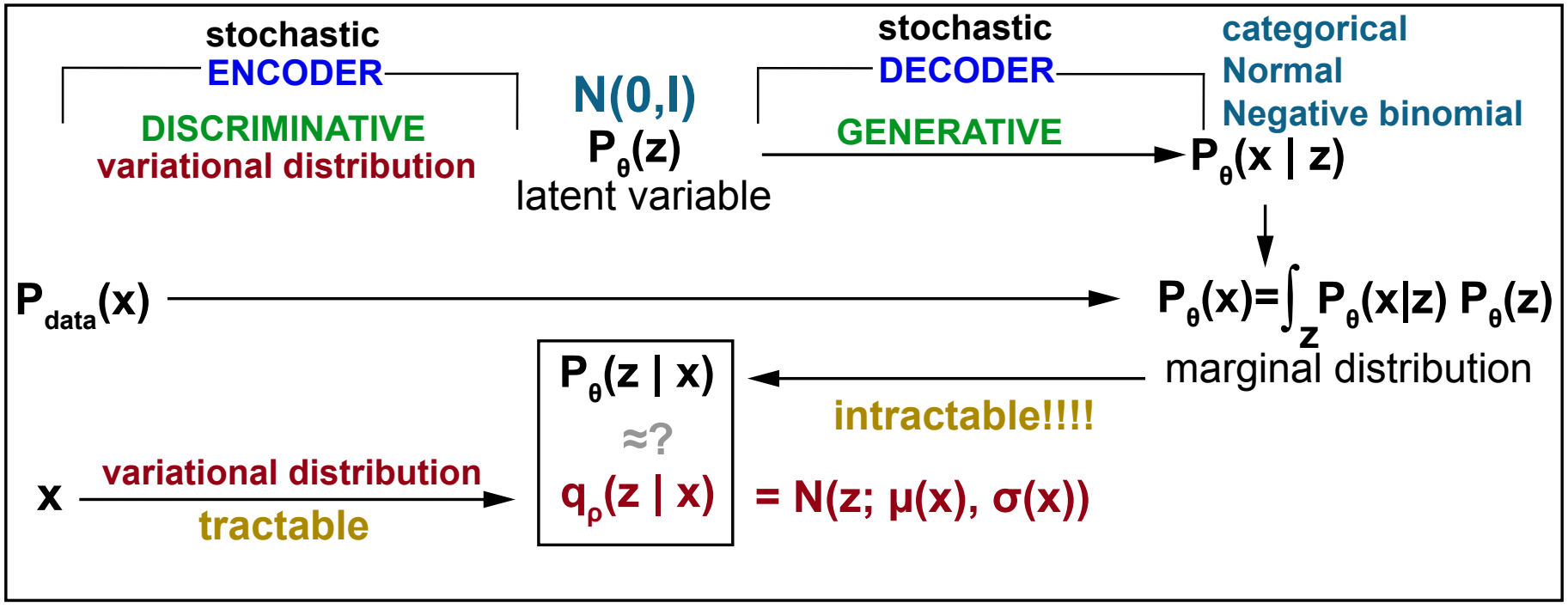




Variational AutoEncoder (VAE)

**The VAE DECODER is a generative model**

$$P_\theta(xz) = P_\theta(x \mid z)\, P_\theta(z)$$

# VARIATIONAL AUTOENCODER (VAE)

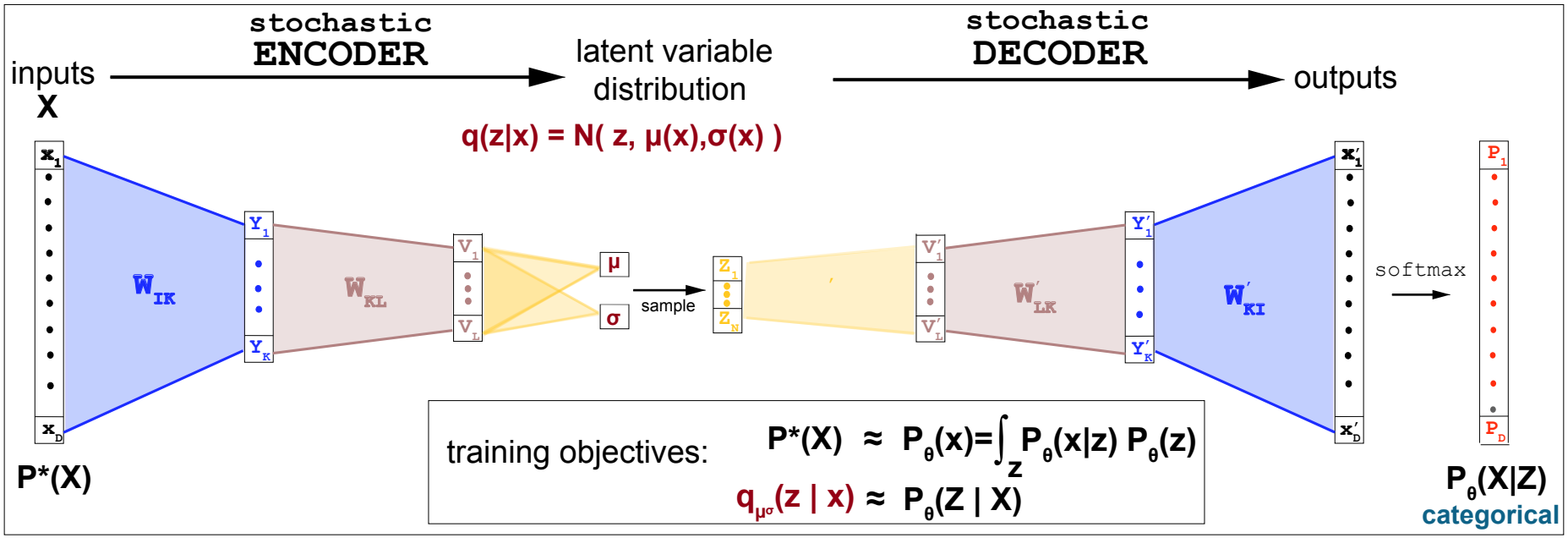



optimization objectives: $P_{data}(x) \approx P_\theta(x)$

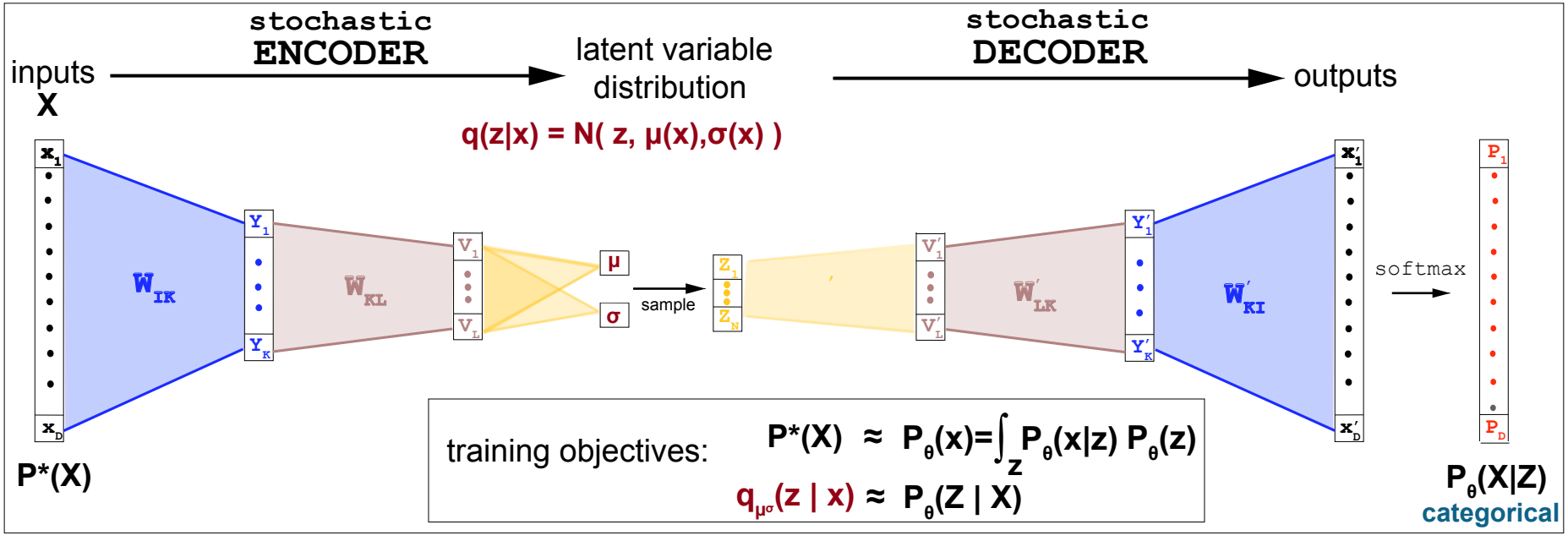
$P_\theta(z \mid x) \approx q_\rho(z \mid x)$

**Encoder/Decoder Optimization is done all at once!**

## VARIATIONAL AUTOENCODER (VAE)



## VARIATIONAL AUTOENCODER (VAE)



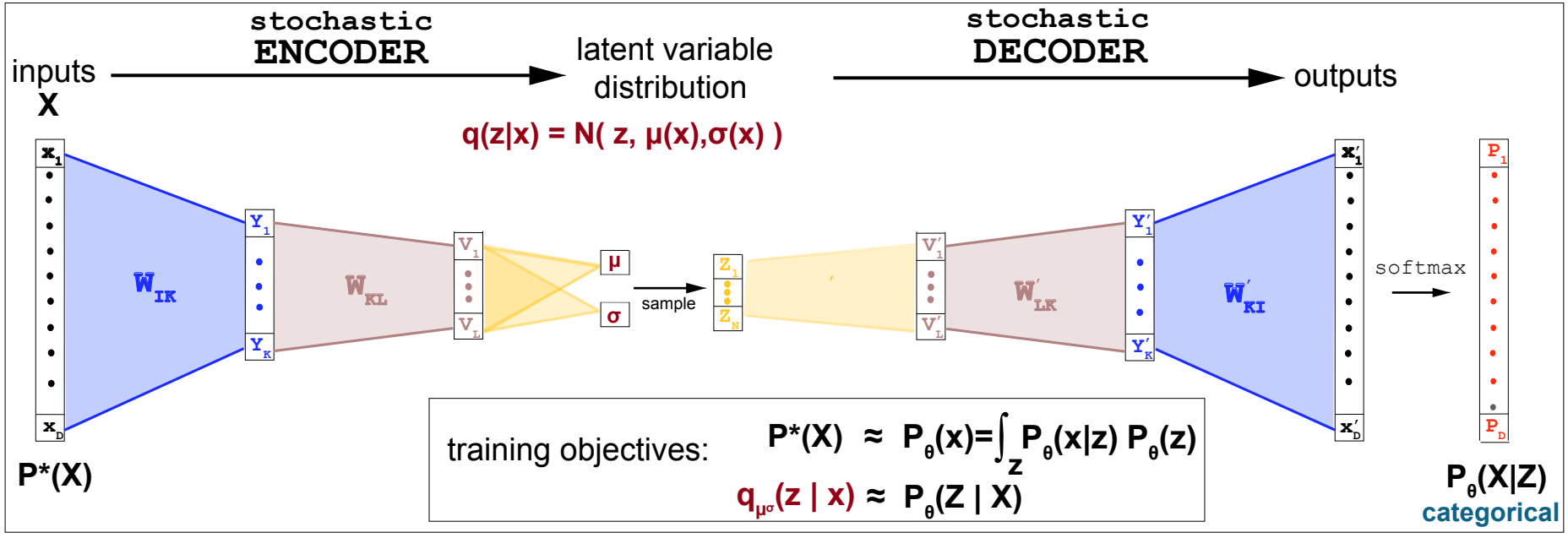

VAE are **Deep Learning Variable Models (DLVM)**

DLVMs use neural networks architectures to train parameters

The **VAE decoder** uses NNs to train $\theta$ in $P_\theta(x \mid z)$

The **VAE encoder** uses NNs to train $\mu, \sigma$ in $q(z \mid x) = N(\mu(x), \sigma(x))$

**VARIATIONAL AUTOENCODER (VAE)**




## The two optimization objectives in a VAE are

- $P_{data}(x) \approx P_\theta(x) = \int_z P_\theta(x \mid z) P_\theta(z)$
  the generative model's marginal fits the data
- $q_{\mu,\sigma}(z \mid x) = N(z; \mu(x), \sigma(x)) \approx P_\theta(z \mid x) = \frac{P_\theta(x|z) P_\theta(z)}{P_\theta(x)}$
  the inference model fits the generative model's posterior

# The **Important Quantity** in Unsupervised Learning

$$\min_{\theta} D_{KL}(P_{data}||P_{\theta}(x)) = \min_{\theta} \int_{x} P_{data}(x) \log \frac{P_{data}(x)}{P_{\theta}(x)}$$

$$= \max_{\theta} \int_{x \sim P_{data}} \log P_{\theta}(x)$$

$$\boxed{\max_{\theta} \int_{x \sim P_{data}} \log P_{\theta}(x)}$$

$$\max_\theta \int_{x \sim P_{data}} \log P_\theta(x)$$

# Why do we need a "variational inference" (VI) approach to optimize?

$$\max_\theta \int_{x \sim P_{data}} \log P_\theta(x)$$

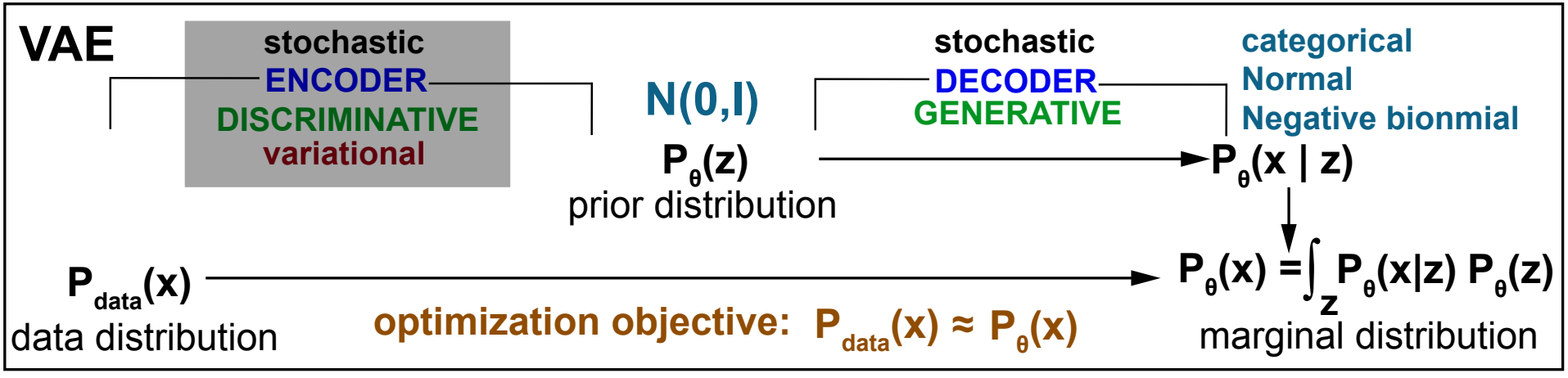
## Why do we need a "variational inference" (VI) approach to optimize?

Because we don't know $P_\theta(x)$ directly

$$\max_\theta \int_{x \sim P_{data}} \log P_\theta(x)$$

# Why do we need a "variational inference" (VI) approach to optimize?

Because we don't know $P_\theta(x)$ directly




We only know $P_\theta(xz)$ $[P_\theta(x \mid z)\, P_\theta(z)]$

# Variational Inference

$$
\begin{aligned}
\log P_\theta(x) &= \log \left[ \int_z P_\theta(x\, z)\, dz \right] \\
&= \log \left[ \int_z \frac{q_\varphi(z \mid x)}{q_\varphi(z \mid x)} P_\theta(x\, z)\, dz \right] \\
&\geq \int_z q_\varphi(z \mid x) \log \left[ \frac{P_\theta(x\, z)}{q_\varphi(z \mid x)} \right] dz \qquad \text{Jensen's inequality} \\
&= E_{q_\varphi(z|x)} \log \left[ \frac{P_\theta(x\, z)}{q_\varphi(z \mid x)} \right] \\
&= \mathsf{ELBO}(\varphi, \theta, x)
\end{aligned}
$$

$$\mathsf{ELBO} = \text{Evidence Lower Bound}$$

# the ELBO is the Loss

$$ELBO_{\varphi\theta}(x) = E_{q_\varphi(z|x)}\left[\frac{P_\theta(x\,z)}{q_\varphi(z \mid x)}\right]$$
$$= E_{q_\varphi(z|x)}\left[\log P_\theta(x,z) - \log q_\varphi(z \mid x)\right].$$

$$\log P_\theta(x) \geq \mathsf{ELBO}(\varphi,\theta,x)$$

Proxy to

$$\max_\theta E_{x\sim P_{data}} \log P_\theta(x)$$

**We maximize the ELBO**

$$\boxed{\max_{\varphi,\theta} E_{x\sim P_{data}} \mathsf{ELBO}(\varphi,\theta,x)}$$

# The **Variational Inference** approach

## Alternative derivation

$$
\begin{aligned}
\log P_\theta(x) &= E_{q_\varphi(z|x)}\left[\log P_\theta(x)\right] \\
&= E_{q_\varphi(z|x)}\left[\log \frac{P_\theta(x,z)}{P_\theta(z \mid x)}\right] \\
&= E_{q_\varphi(z|x)}\left[\log \frac{P_\theta(x,z)}{q_\varphi(z \mid x)} \frac{q_\varphi(z \mid x)}{P_\theta(z \mid x)}\right] \\
&= E_{q_\varphi(z|x)}\left[\log \frac{P_\theta(x,z)}{q_\varphi(z \mid x)}\right] + E_{q_\varphi(z|x)}\left[\log \frac{q_\varphi(z \mid x)}{P_\theta(z \mid x)}\right]
\end{aligned}
$$

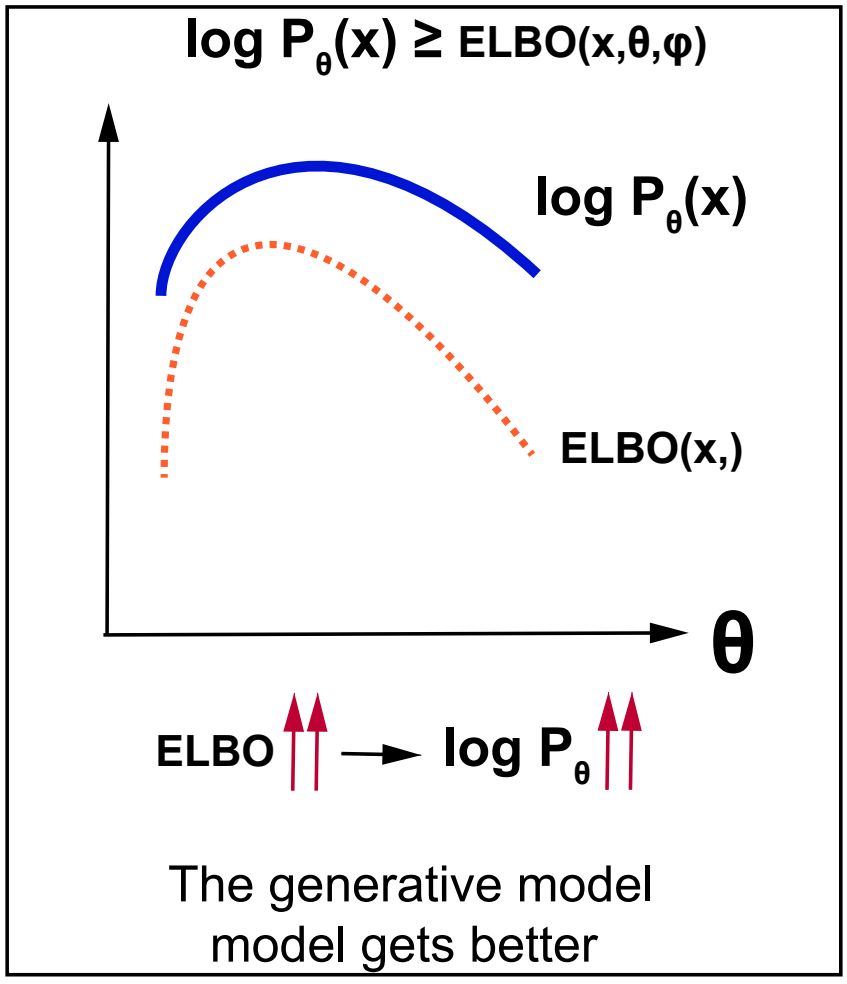
Because the second term

$$
E_{q_\varphi(z|x)}\left[\log \frac{q_\varphi(z|x)}{P_\theta(z|x)}\right] = D_{KL}(q_\varphi(z \mid x)||P_\theta(z \mid x)) \geq 0,
$$

then

$$
\boxed{\log P_\theta(x) \geq ELBO_{\varphi\theta}(x) = E_{q_\varphi(z|x)}\left[\log \frac{P_\theta(x,z)}{q_\varphi(z|x)}\right]}
$$

# One ELBO Loss does it all
# Two for One



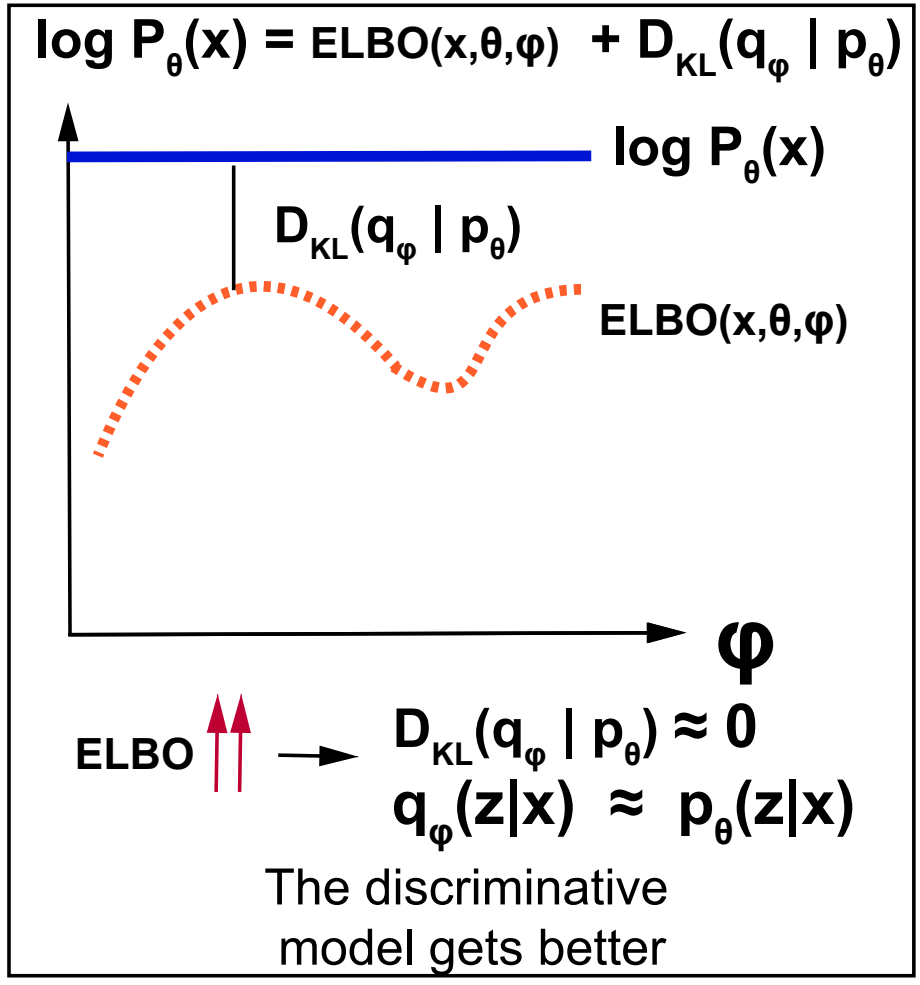



Figure 2.1 Kingma & Welling, 2019

**Variational Autoencoder**

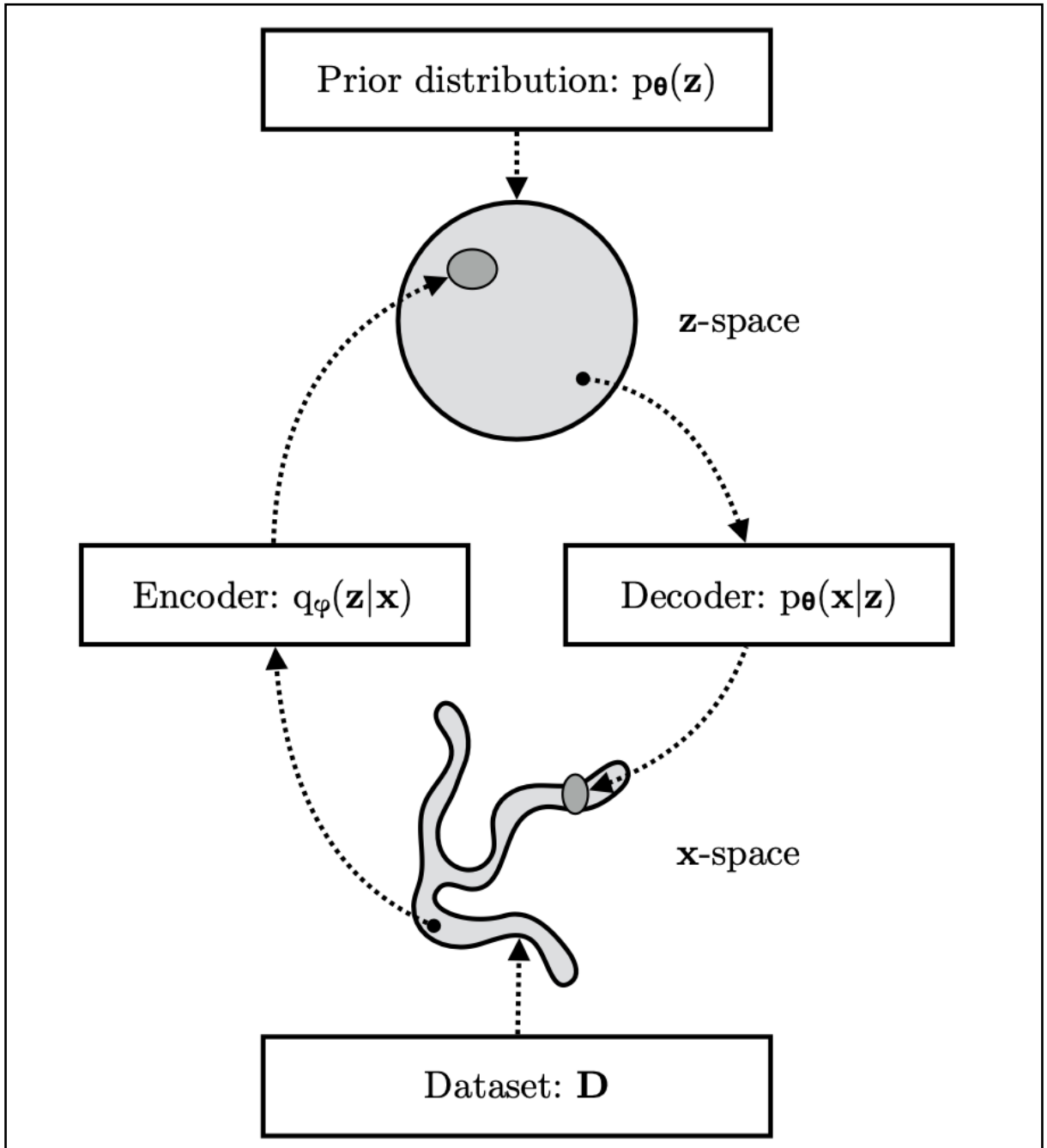


Figure 2.1 Kingma & Welling, 2019

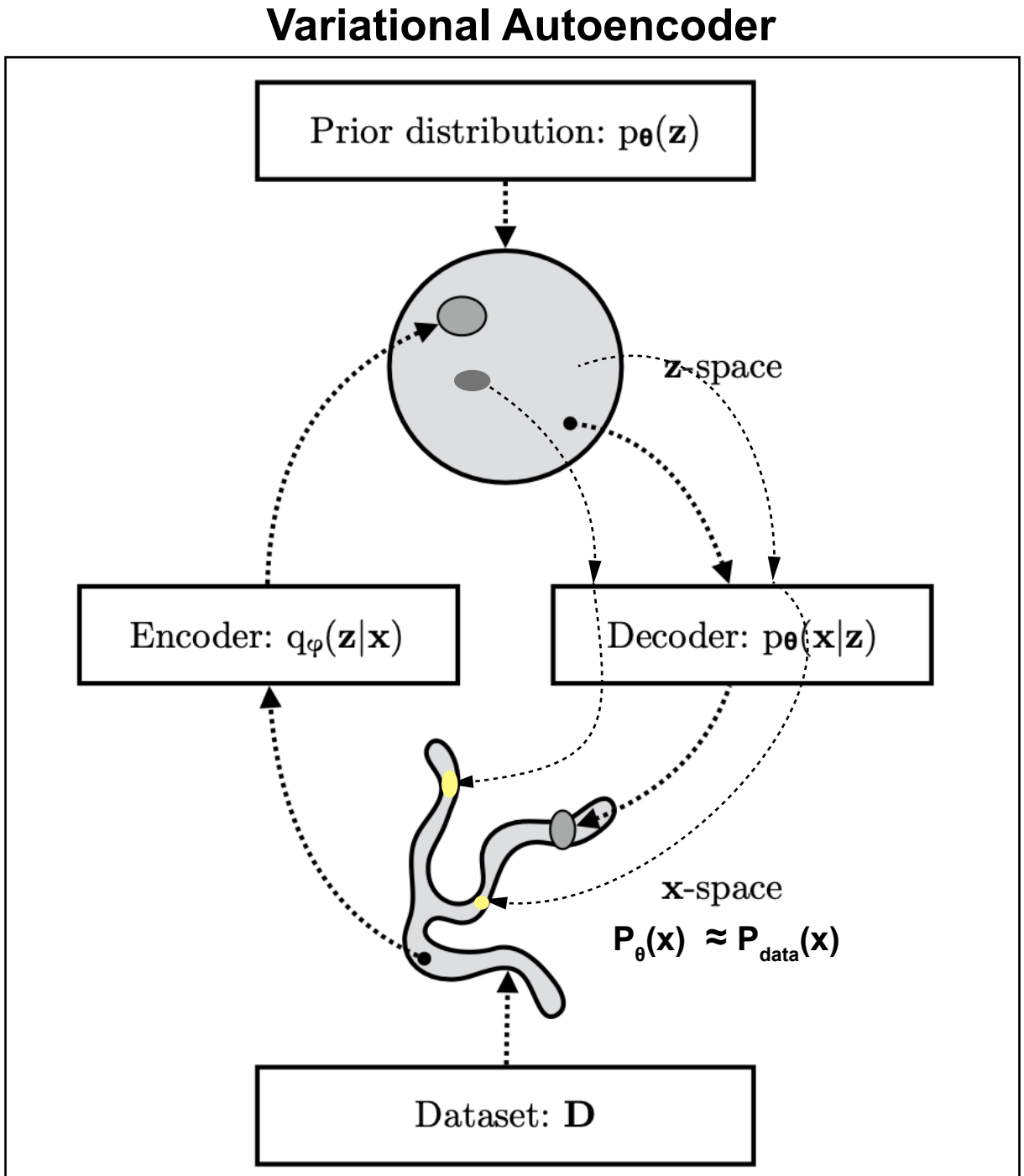
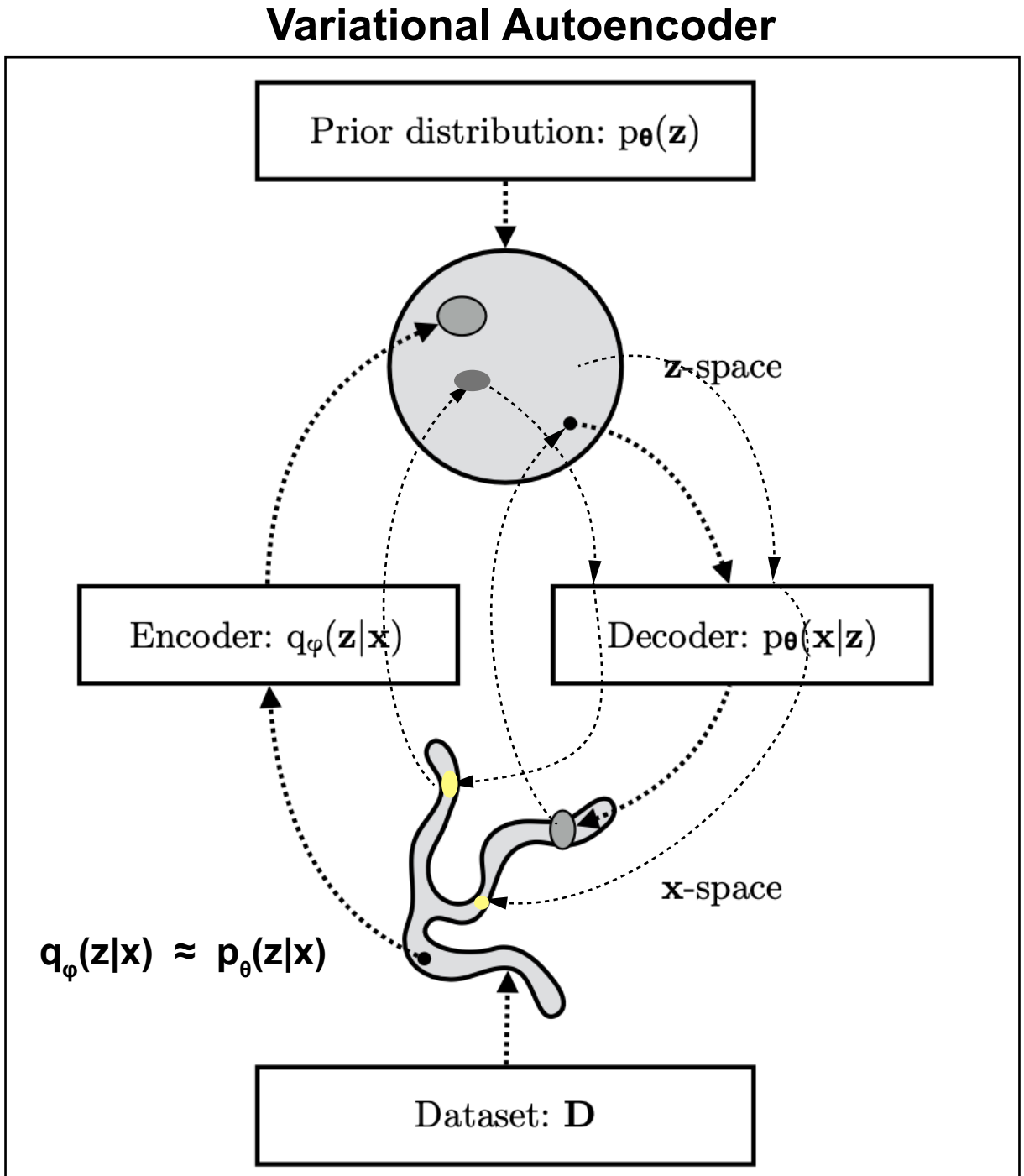
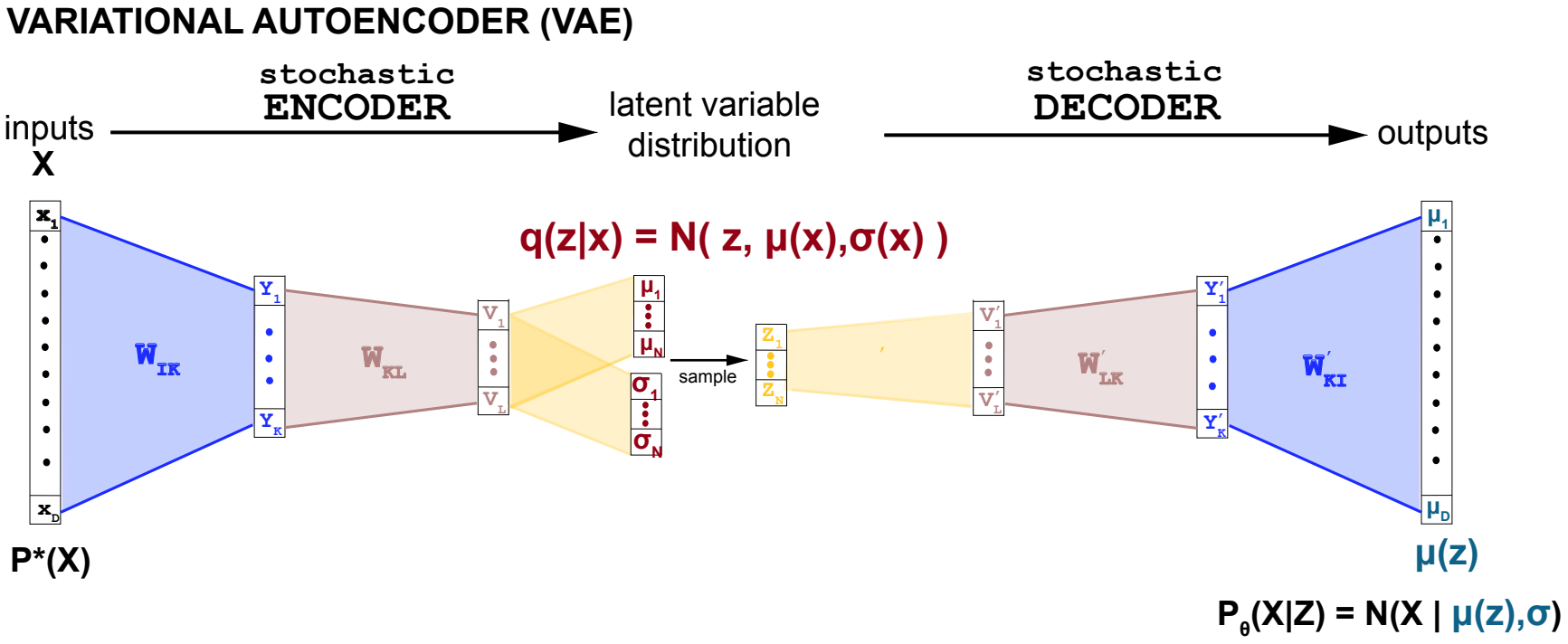


Figure 2.1 Kingma & Welling, 2019

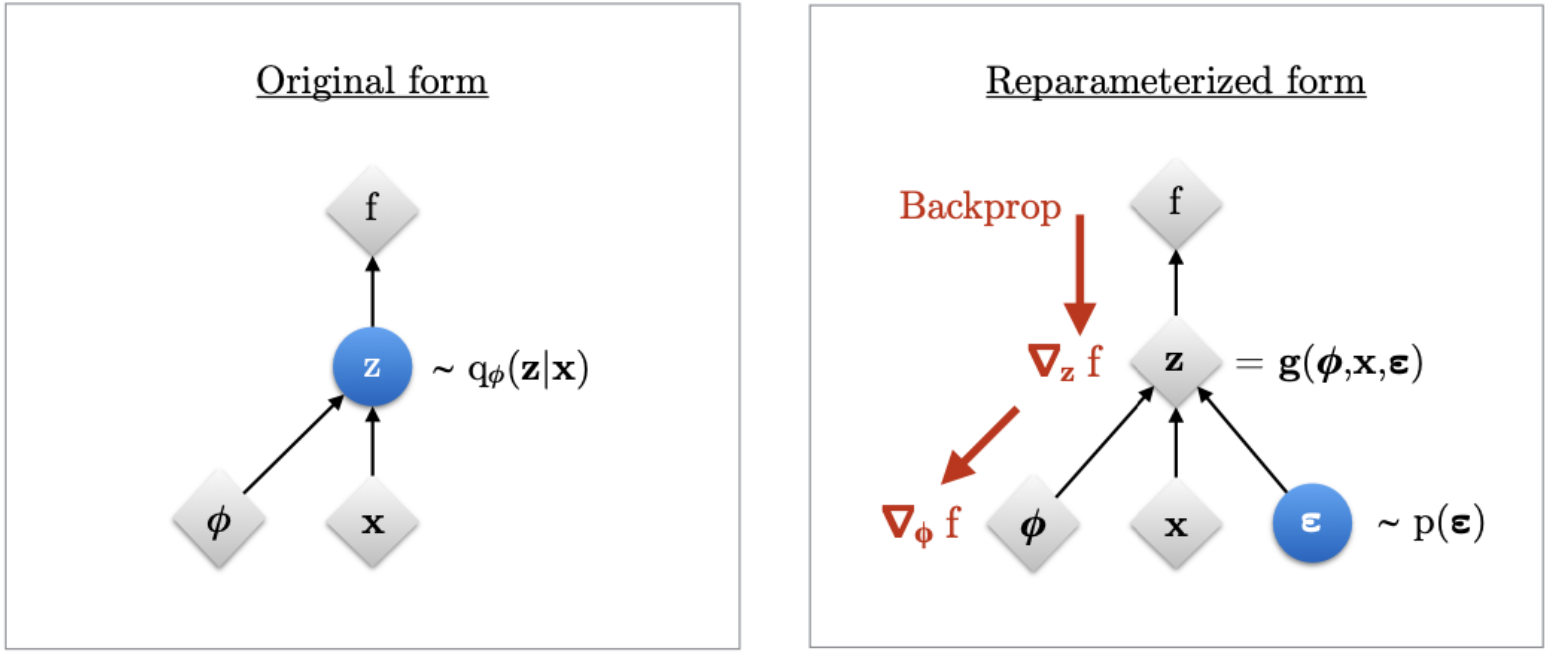


# the reparameterization trick

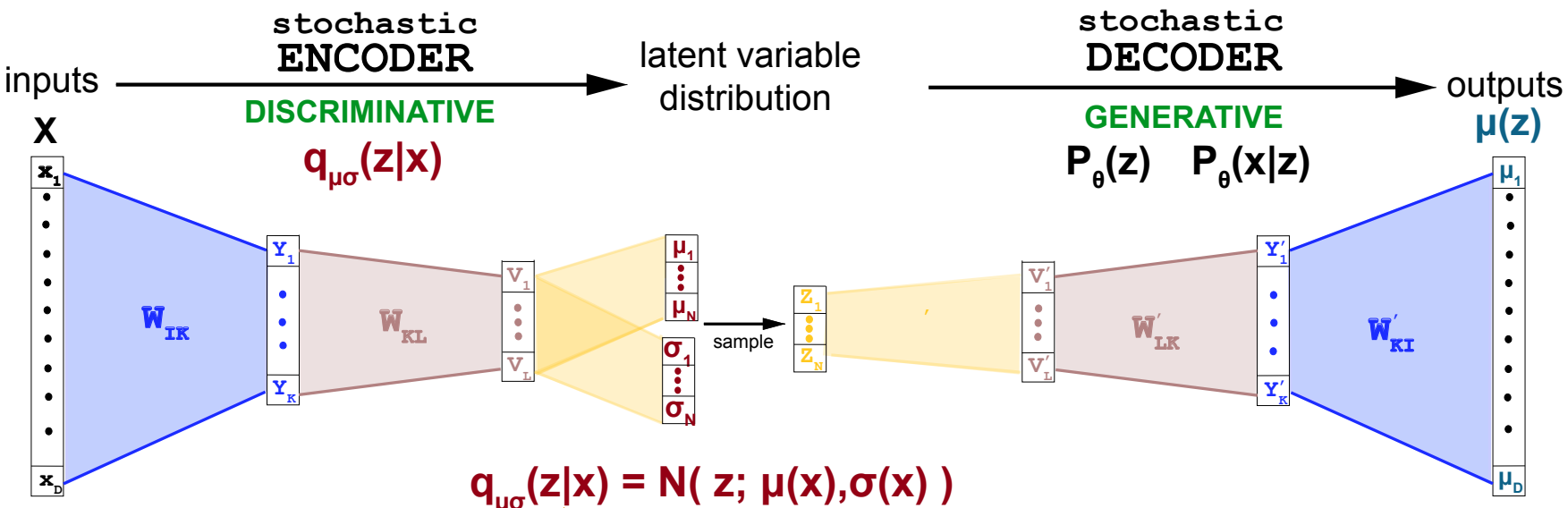



In PyTorch, a **stochastic node** samples from a probability distribution.
stochastic nodes cannot be directly backpropagated
we need a **deterministic node**

# The reparameterization trick



# VARIATIONAL AUTOENCODER (VAE)




$$q_{\mu\sigma}(z|x) = N(z;\ \mu(x),\sigma(x))$$

$$\approx$$

$$P_\theta(z|x) = \frac{P_\theta(x|z)\,P_\theta(z)}{P_\theta(x)}$$

$$P_\theta(x|z) = N(x \mid \mu(z),\sigma)$$

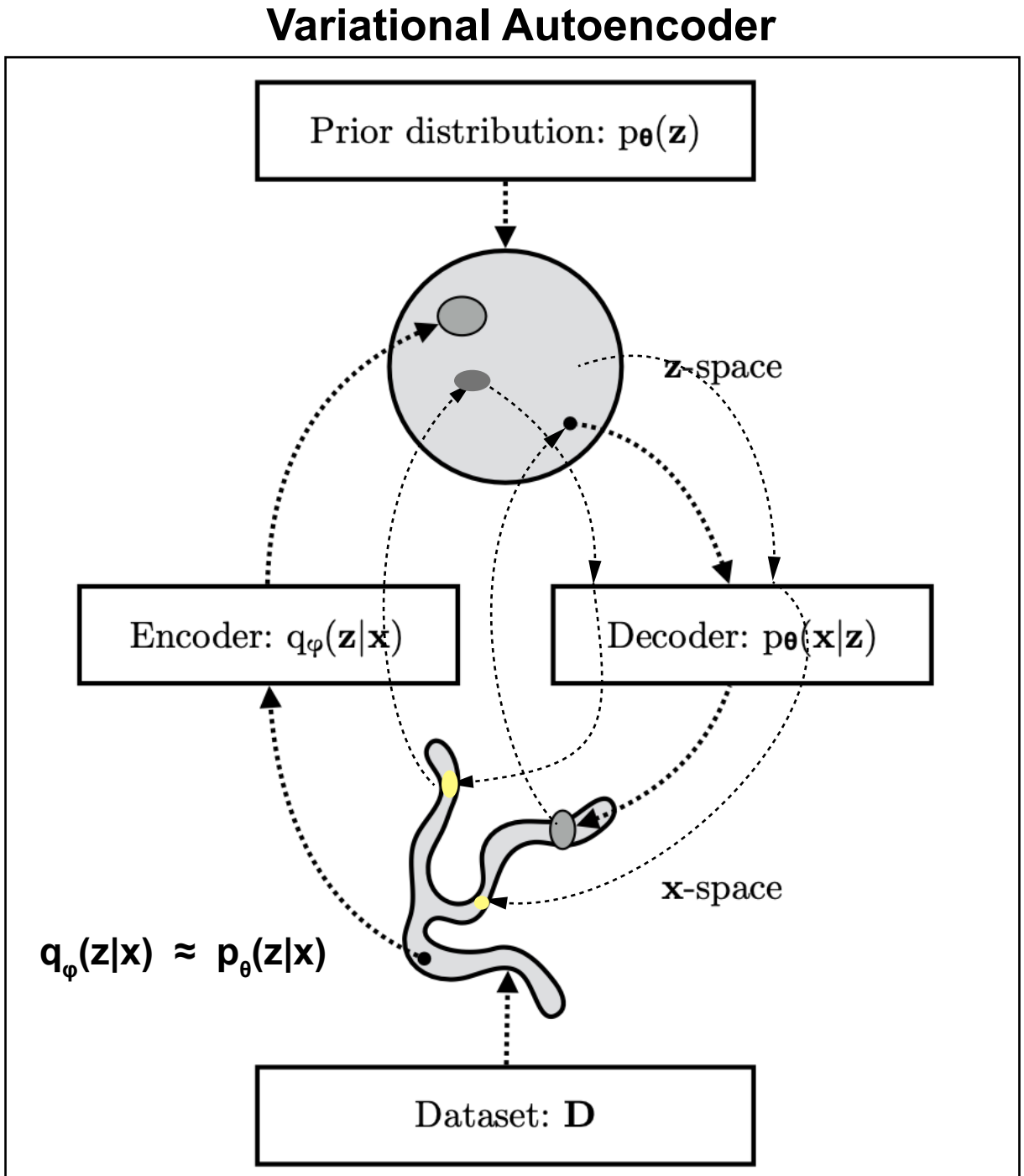
$$P_{data}(X) \quad \approx \quad P_\theta(x)=\int_z P_\theta(x|z)\,P_\theta(z)$$

**optimization objectives:**

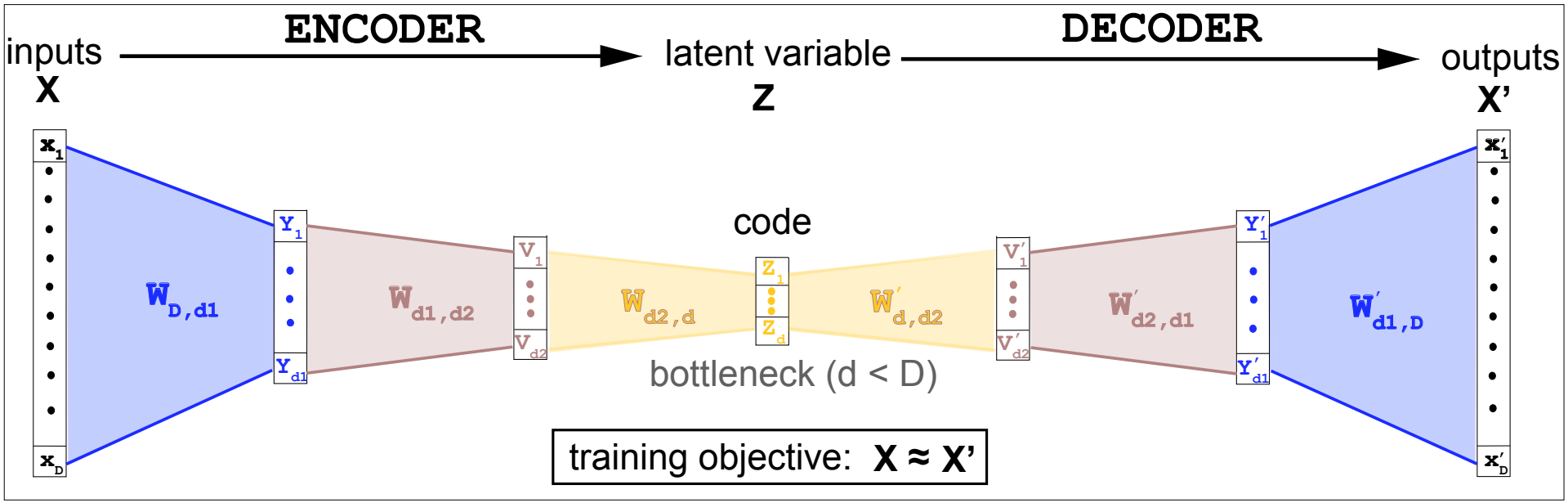
$$P_{data}(x) \approx P_\theta(x)$$

$$P_\theta(z \mid x) \approx q_{\mu\sigma}(z \mid x)$$

Figure 2.1 Kingma & Welling, 2019



## AUTOENCODER



# AutoEncoder Forward loop

```python
# The model
class GeneExprAE(nn.Module):
    def __init__(self, input_dim, h1=512, h2=256, latent_dim=32):
        super().__init__()
        # ----- Encoder: input -> h1 -> h2 -> latent(z) -----
        self.enc_fc1 = nn.Linear(input_dim, h1)
        self.enc_fc2 = nn.Linear(h1, h2)
        self.enc_latent = nn.Linear(h2, latent_dim)

        # ----- Decoder: latent -> h2 -> h1 -> input -----
        self.dec_fc1 = nn.Linear(latent_dim, h2)
        self.dec_fc2 = nn.Linear(h2, h1)
        self.dec_out = nn.Linear(h1, input_dim)

    def encode(self, x):
        h = F.relu(self.enc_fc1(x))   # 1st hidden
        h = F.relu(self.enc_fc2(h))   # 2nd hidden
        z = self.enc_latent(h)        # bottleneck
        return z

    def decode(self, z):
        h = F.relu(self.dec_fc1(z))   # 1st decoder hidden
        h = F.relu(self.dec_fc2(h))   # 2nd decoder hidden
        x_ae = self.dec_out(h)        # linear output for real-valued expression
        return x_ae

    def forward(self, x):
        z = self.encode(x)
        x_ae = self.decode(z)
        return x_ae, z
```

```python
# VARIATIONAL AUTOENCODER
#
class GeneExprVAE(nn.Module):
    def __init__(self, input_dim, h1=512, h2=256, latent_dim=32):
        super().__init__()
        # ----- Encoder: input -> h1 -> h2 -> (mu, logvar) -----
        self.fc1       = nn.Linear(input_dim, h1)
        self.fc2       = nn.Linear(h1, h2)
        self.fc_mu     = nn.Linear(h2, latent_dim)
        self.fc_logvar = nn.Linear(h2, latent_dim)

        # ----- Decoder: latent -> h2 -> h1 -> input -----
        self.fc3    = nn.Linear(latent_dim, h2)
        self.fc4    = nn.Linear(h2, h1)
        self.fc_out = nn.Linear(h1, input_dim)

    def encode(self, x):
        h = F.relu(self.fc1(x))    # 1st hidden layer
        h = F.relu(self.fc2(h))    # 2nd hidden layer
        mu = self.fc_mu(h)         # bottleneck mean
        logvar = self.fc_logvar(h) # bottleneck log-variance
        return mu, logvar

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std) # generate epsilon from N(0,1)
        return mu + eps * std

    def decode(self, z):
        h = F.relu(self.fc3(z))    # 1st decoder hidden
        h = F.relu(self.fc4(h))    # 2nd decoder hidden
        x_out = self.fc_out(h)     # reconstruction (linear for real-valued expression)
        return x_out

    def forward(self, x):
        mu, logvar = self.encode(x)
        z = self.reparameterize(mu, logvar)
        x_out = self.decode(z)
        return x_out, mu, logvar
```

***instead of z directly***

***instead of z sampled from q(z|x) using the reparameterization trick***

$$\max_\theta \int_{x \sim P_{data}} \log P_\theta(x)$$

$$\begin{aligned}
\log P_\theta(x) &= E_{q_\varphi(z|x)} \left[\log P_\theta(x)\right] \\
&= E_{q_\varphi(z|x)} \left[\log \frac{P_\theta(x,z)}{P_\theta(z \mid x)}\right] \\
&= E_{q_\varphi(z|x)} \left[\log \frac{P_\theta(x,z)}{q_\varphi(z \mid x)} \frac{q_\varphi(z \mid x)}{P_\theta(z \mid x)}\right] \\
&= E_{q_\varphi(z|x)} \left[\log \frac{P_\theta(x,z)}{q_\varphi(z \mid x)}\right] + E_{q_\varphi(z|x)} \left[\log \frac{q_\varphi(z \mid x)}{P_\theta(z \mid x)}\right]
\end{aligned}$$

$\text{ELBO}(\varphi, \theta, x)$

$D_{KL}(q_\varphi(\ \mid x) || P_\theta(\ \mid x)) \geq 0$

$$\log P_\theta(x) \geq ELBO_{\varphi\theta}(x) = E_{q_\varphi(z|x)} \left[\log \frac{P_\theta(x,z)}{q_\varphi(z|x)}\right]$$

Proxy to

$\max_\theta E_{x \sim P_{data}} \log P_\theta(x)$

**We maximize the ELBO**

$\max_{\varphi,\theta} E_{x \sim P_{data}} \text{ELBO}(\varphi, \theta, x)$

Proxy to

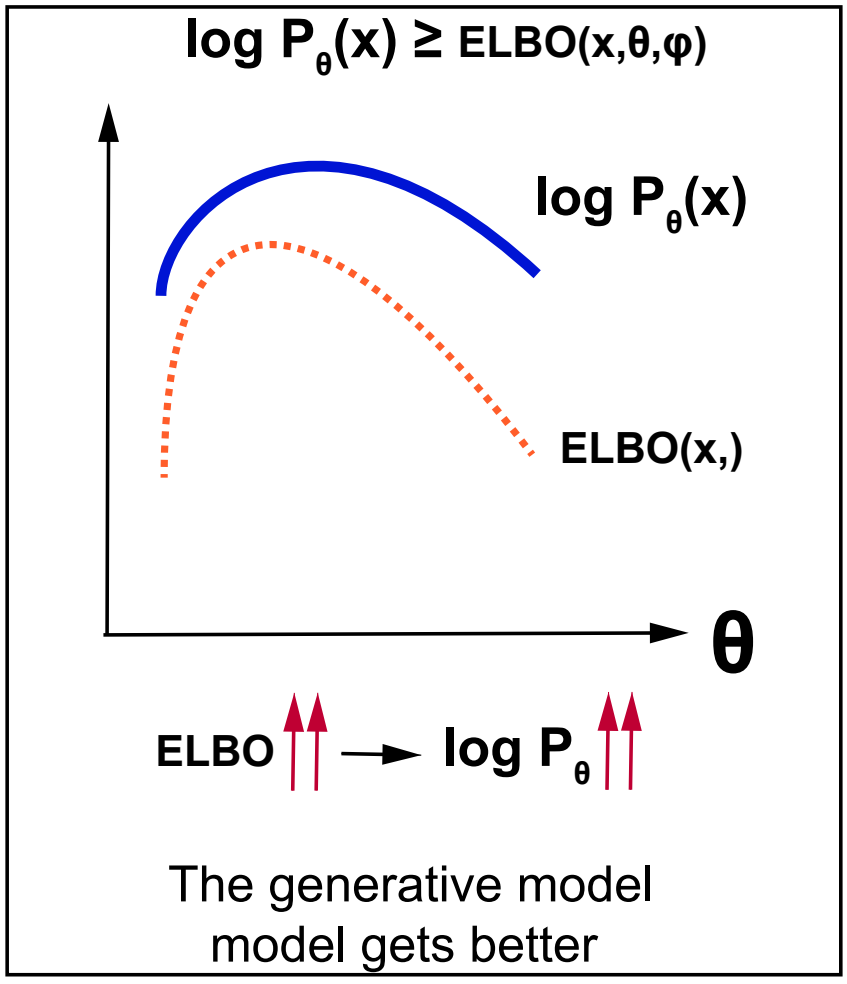
$$\max_\theta E_{x \sim P_{data}} \log P_\theta(x)$$

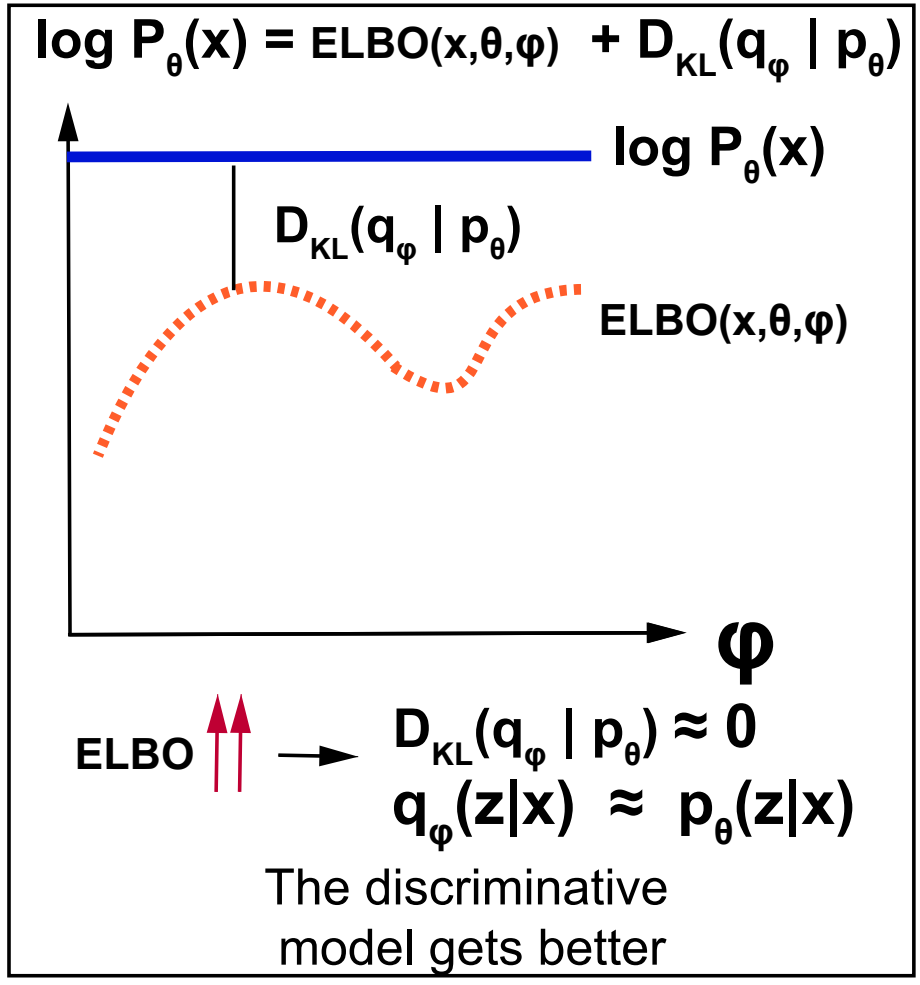
**We maximize the ELBO**

$$\max_{\varphi,\theta} E_{x \sim P_{data}} \text{ELBO}(\varphi, \theta, x)$$

$$ELBO_{\varphi\theta}(x) = E_{q_\varphi(z|x)} \left[ \frac{P_\theta(x\, z)}{q_\varphi(z \mid x)} \right]$$

# One ELBO Loss does it all
# Two for One



# VAE Loss

$$
\begin{aligned}
ELBO_{\varphi\theta}(x) &= E_{q_\varphi(z|x)}\left[\log P_\theta(x,z) - \log q_\varphi(z \mid x)\right] \\
&= E_{q_\phi(z|x)} \log \frac{P_\theta(x,z)}{q_\varphi(z \mid x)} \\
&= E_{q_\phi(z|x)} \log \frac{P_\theta(x,z)}{P_\theta(z)} \frac{P_\theta(z)}{q_\varphi(z \mid x)} \\
&= E_{q_\phi(z|x)} \log \frac{P_\theta(x,z)}{P_\theta(z)} + E_{q_\phi(z|x)} \log \frac{P_\theta(z)}{q_\varphi(z \mid x)} \\
&= E_{q_\phi(z|x)} \log P_\theta(x \mid z) - D_{KL}(q_\varphi(z \mid x) || P_\theta(z))
\end{aligned}
$$

$$
\begin{aligned}
VAE\_LOSS_{\varphi\theta}(x) &= -ELBO_{\varphi\theta}(x) \\
&= -E_{q_\phi(z|x)} \log P_\theta(x \mid z) + D_{KL}(q_\varphi(z \mid x) || P_\theta(z))
\end{aligned}
$$

z conditional / z prior

# VAE Loss

$$
\begin{aligned}
ELBO_{\varphi\theta}(x) &= E_{q_\varphi(z|x)} \left[\log P_\theta(x,z) - \log q_\varphi(z \mid x)\right] \\
&= E_{q_\phi(z|x)} \log \frac{P_\theta(x,z)}{q_\varphi(z \mid x)} \\
&= E_{q_\phi(z|x)} \log \frac{P_\theta(x,z)}{P_\theta(z)} \frac{P_\theta(z)}{q_\varphi(z \mid x)} \\
&= E_{q_\phi(z|x)} \log \frac{P_\theta(x,z)}{P_\theta(z)} + E_{q_\phi(z|x)} \log \frac{P_\theta(z)}{q_\varphi(z \mid x)} \\
&= E_{q_\phi(z|x)} \log P_\theta(x \mid z) - D_{KL}(q_\varphi(z \mid x) || P_\theta(z))
\end{aligned}
$$

$$
\begin{aligned}
VAE\_LOSS_{\varphi\theta}(x) &= -ELBO_{\varphi\theta}(x) \\
&= -E_{q_\phi(z|x)} \log P_\theta(x \mid z) + D_{KL}(q_\varphi(z \mid x) || P_\theta(z))
\end{aligned}
$$

# VAE Loss

$$VAE\_LOSS_{\varphi\theta}(x) = -E_{q_\phi(z|x)} \log P_\theta(x \mid z) + D_{KL}(q_\varphi(z \mid x)||P_\theta(z))$$

Because $q_\varphi(z \mid x)$ and $P_\theta(z)$ are both Normal

$$D_{KL}(N(\mu_\varphi(x), \sigma^2(x)||N(0,1)) = -\frac{1}{2}\sum_{i=1}^{d}(1 + \log \sigma_i^2 - \mu_i^2 - \sigma_i^2)$$

$$VAE\_LOSS_{\varphi\theta}(x) = -E_{q_\phi(z|x)} \log P_\theta(x \mid z) - \frac{1}{2}\sum_{i=1}^{d}(1 + \log \sigma_i^2 - \mu_i^2 - \sigma_i^2)$$

# VAE Loss

$$VAE\_LOSS_{\varphi\theta}(x) = -E_{q_\phi(z|x)} \log P_\theta(x \mid z) + D_{KL}(q_\varphi(z \mid x)||P_\theta(z))$$

Because $q_\varphi(z \mid x)$ and $P_\theta(z)$ are both Normal

$$D_{KL}(N(\mu_\varphi(x), \sigma^2(x)||N(0,1)) = -\frac{1}{2}\sum_{i=1}^{d}(1 + \log \sigma_i^2 - \mu_i^2 - \sigma_i^2)$$

$$VAE\_LOSS_{\varphi\theta}(x) = -E_{q_\phi(z|x)} \log P_\theta(x \mid z) - \frac{1}{2}\sum_{i=1}^{d}(1 + \log \sigma_i^2 - \mu_i^2 - \sigma_i^2)$$

# VAE Loss

$$VAE\_LOSS_{\varphi\theta}(x) = -E_{q_\phi(z|x)} \log P_\theta(x \mid z) + D_{KL}(q_\varphi(\cdot \mid x)||P_\theta(\cdot))$$

If this generative model is Normal,

$$\log P_\theta(x \mid z) = \frac{-1}{2}||x - \mu_\theta(z)||^2 + constant.$$

Thus, **minimizing** the

$$-E_{q_\varphi(z|x)} \log P_\theta(x \mid z)$$

is equivalent to **minimizing** the mean square error

$$VAE\_LOSS_{\varphi\theta}(x) = E_{q_\varphi(z|x)}||x - \mu_\theta(z)||^2 - \frac{1}{2}\sum_{i=1}^{d}(1 + \log \sigma_i^2 - \mu_i^2 - \sigma_i^2)$$

# VAE Loss

$$VAE\_LOSS_{\varphi\theta}(x) = -E_{q_\phi(z|x)} \log P_\theta(x \mid z) + D_{KL}(q_\varphi(\cdot \mid x)||P_\theta(\cdot))$$

If this generative model is Normal,

$$\log P_\theta(x \mid z) = \frac{-1}{2}||x - \mu_\theta(z)||^2 + constant.$$

Thus, **minimizing** the

$$-E_{q_\varphi(z|x)} \log P_\theta(x \mid z)$$

is equivalent to **minimizing** the mean square error

$$VAE\_LOSS_{\varphi\theta}(x) = E_{q_\varphi(z|x)}||x - \mu_\theta(z)||^2 - \frac{1}{2}\sum_{i=1}^{d}(1 + \log \sigma_i^2 - \mu_i^2 - \sigma_i^2)$$

# VAE Loss

```python
from torch.utils.data import DataLoader, TensorDataset

# VAE loss ----
def vae_loss(x_out, x, mu, logvar):
    # reconstruction (MSE)
    recon = F.mse_loss(x_out, x, reduction='mean')
    # KL divergence
    kl = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return recon + kl

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model_vae = GeneExprVAE(input_dim=G, latent_dim=32).to(device)
optimizer = torch.optim.Adam(model_vae.parameters(), lr=1e-3)

n_epochs = 100
for epoch in range(1, n_epochs + 1):
    model_vae.train()
    total_loss = 0.0

    for (batch,) in loader:
        x_in = batch.to(device)

        optimizer.zero_grad()
        x_out, mu, logvar = model_vae(x_in)
        loss = vae_loss(x_out, x_in, mu, logvar)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    print(f"Epoch {epoch:02d} | Loss: {total_loss / len(loader.dataset):.8f}")
```

**If VAE generative P(x|z) is Normal this term is equivalent to the AE loss**

# scVI: VAE for single-cell transcriptomics

nature methods

ARTICLES

https://doi.org/10.1038/s41592-018-0229-2

Dec 2018

## Deep generative modeling for single-cell transcriptomics

Romain Lopez[1], Jeffrey Regier[1], Michael B. Cole[2], Michael I. Jordan[1,3] and Nir Yosef[1,4,5*]

**Single-cell transcriptome measurements can reveal unexplored biological diversity, but they suffer from technical noise and bias that must be modeled to account for the resulting uncertainty in downstream analyses. Here we introduce single-cell variational inference (scVI), a ready-to-use scalable framework for the probabilistic representation and analysis of gene expression in single cells (https://github.com/YosefLab/scVI). scVI uses stochastic optimization and deep neural networks to aggregate information across similar cells and genes and to approximate the distributions that underlie observed expression values, while accounting for batch effects and limited sensitivity. We used scVI for a range of fundamental analysis tasks including batch correction, visualization, clustering, and differential expression, and achieved high accuracy for each task.**

# Gene Expression Profiles from single-cell sequencing (RNA-seq)


| | gene 1 | gene 2 | | | | gene G |
|---|---|---|---|---|---|---|
| cell 1 | | | | | | |
| cell 2 | | | | | | |
| | | | | | | |
| | | | counts[c,g] | | | |
| | | | | | | |
| cell C | | | | | | |

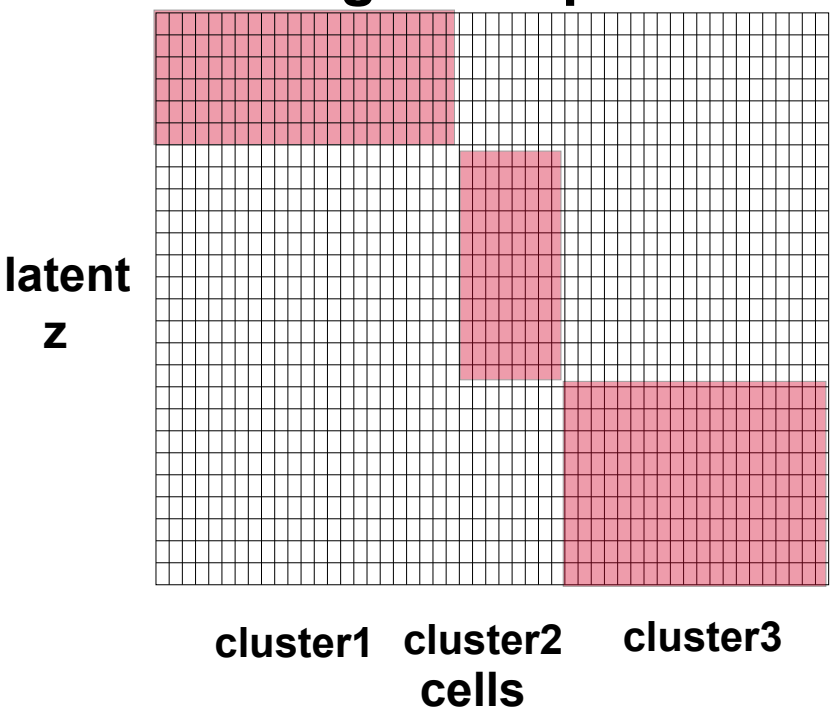
different libraries have different sequencing depths
some samples may have more RNA than others

$$\text{normalized\_counts}[c,g] = \frac{\text{counts}[c,g]}{\sum_{g'} \text{counts}[c,g']}$$

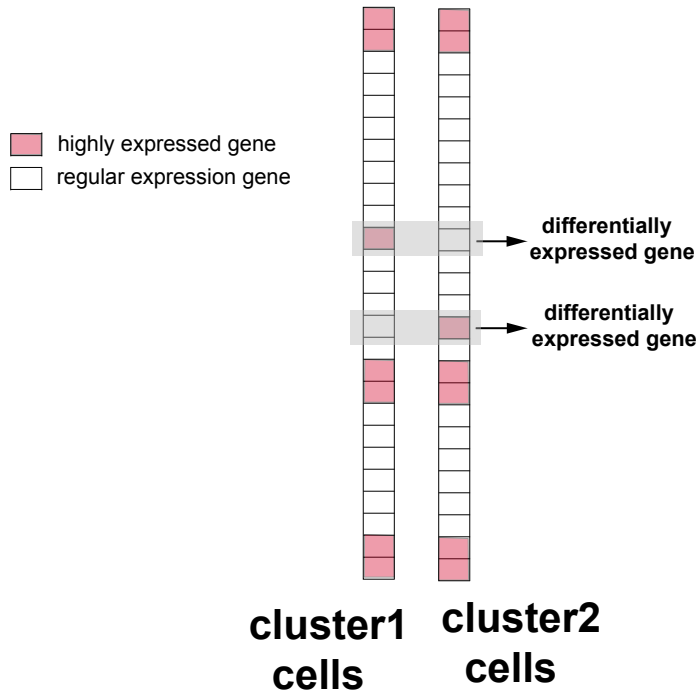
$$\text{log-normalized}[c,g] = \log\left(1 + \text{normalized\_counts}[c,g]\right)$$

# Single-cell transcriptomics

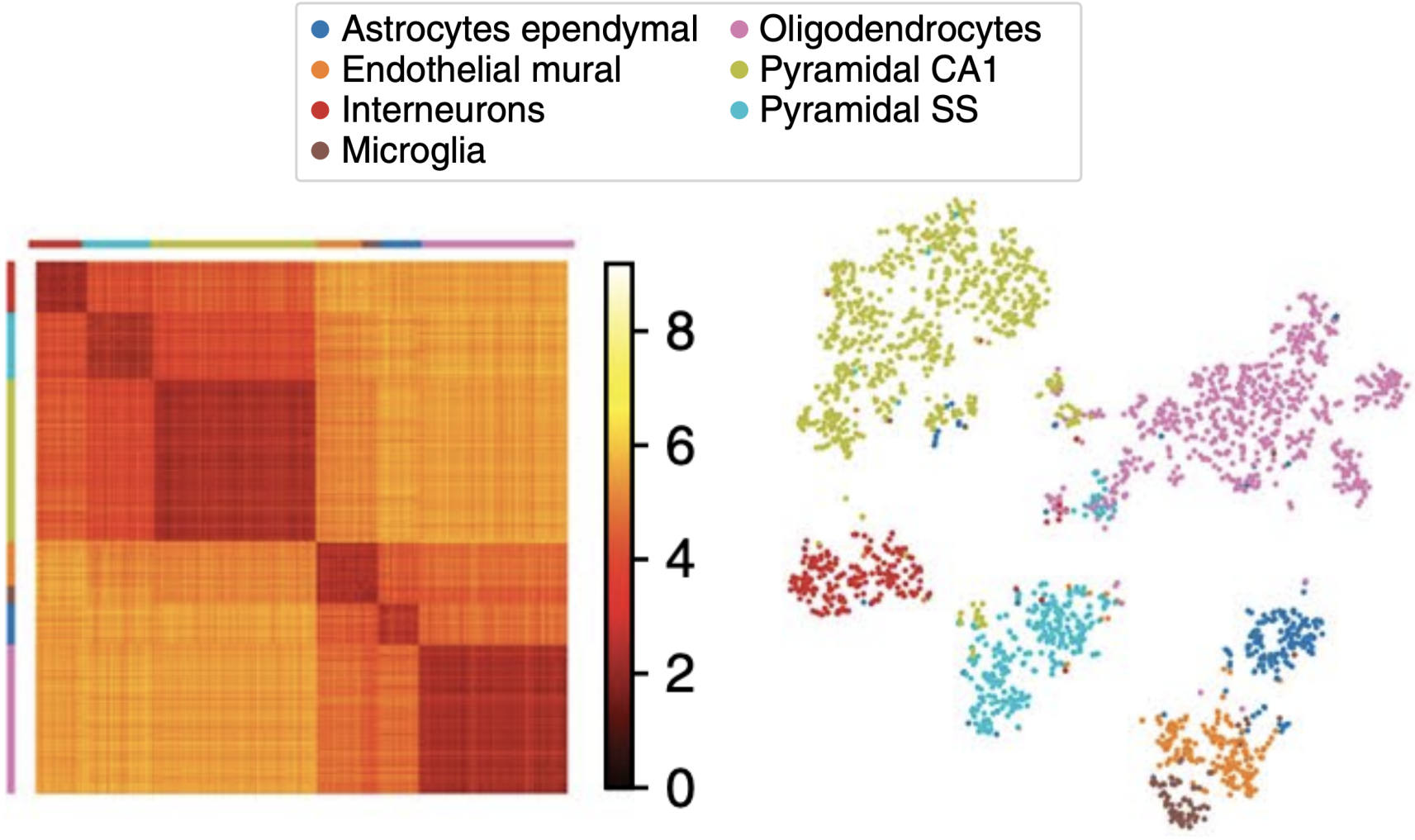
## clusters of cells with similar gene expression




## differentially expressed genes



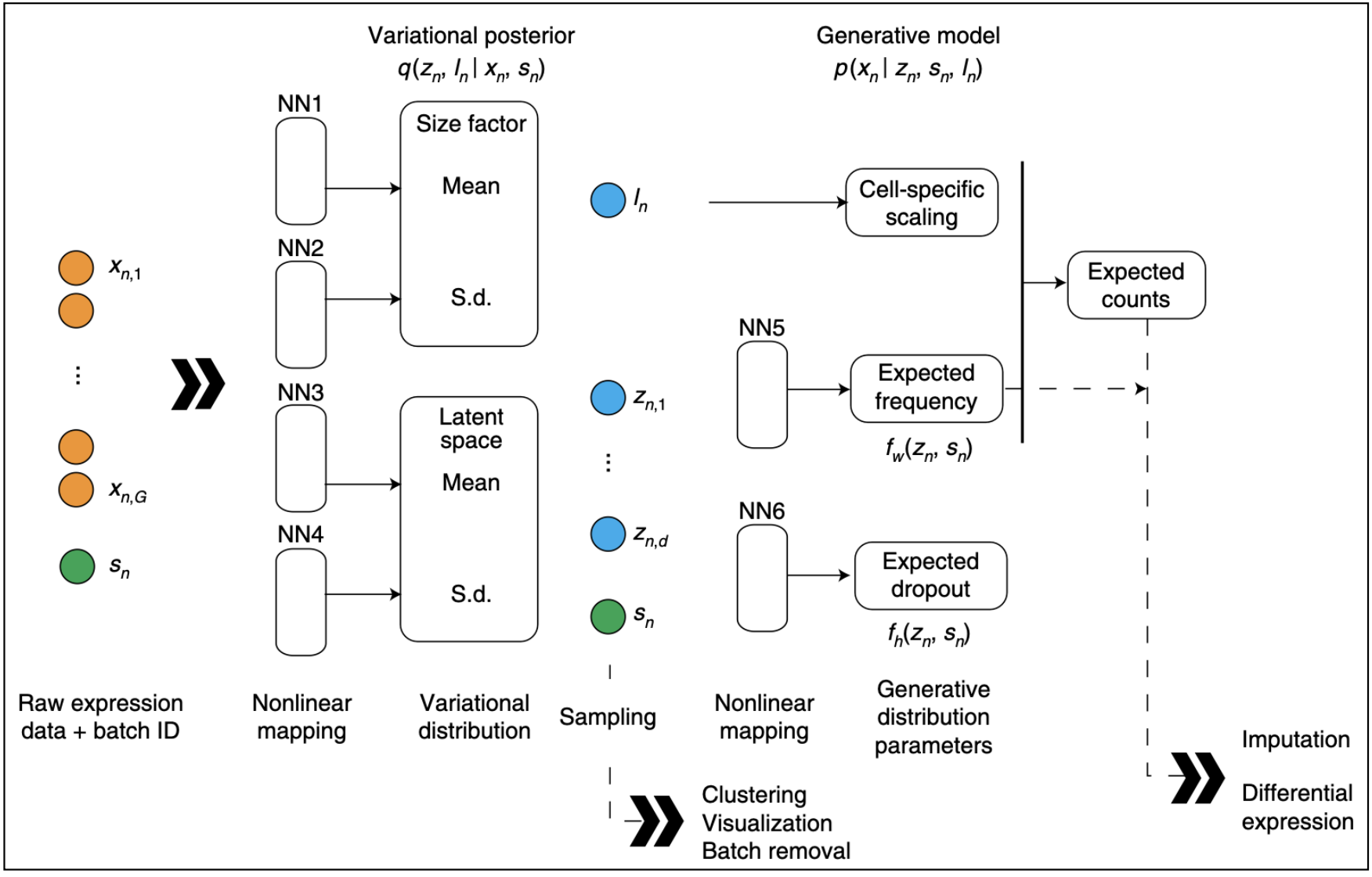
# Single-cell transcriptomics
## cortex



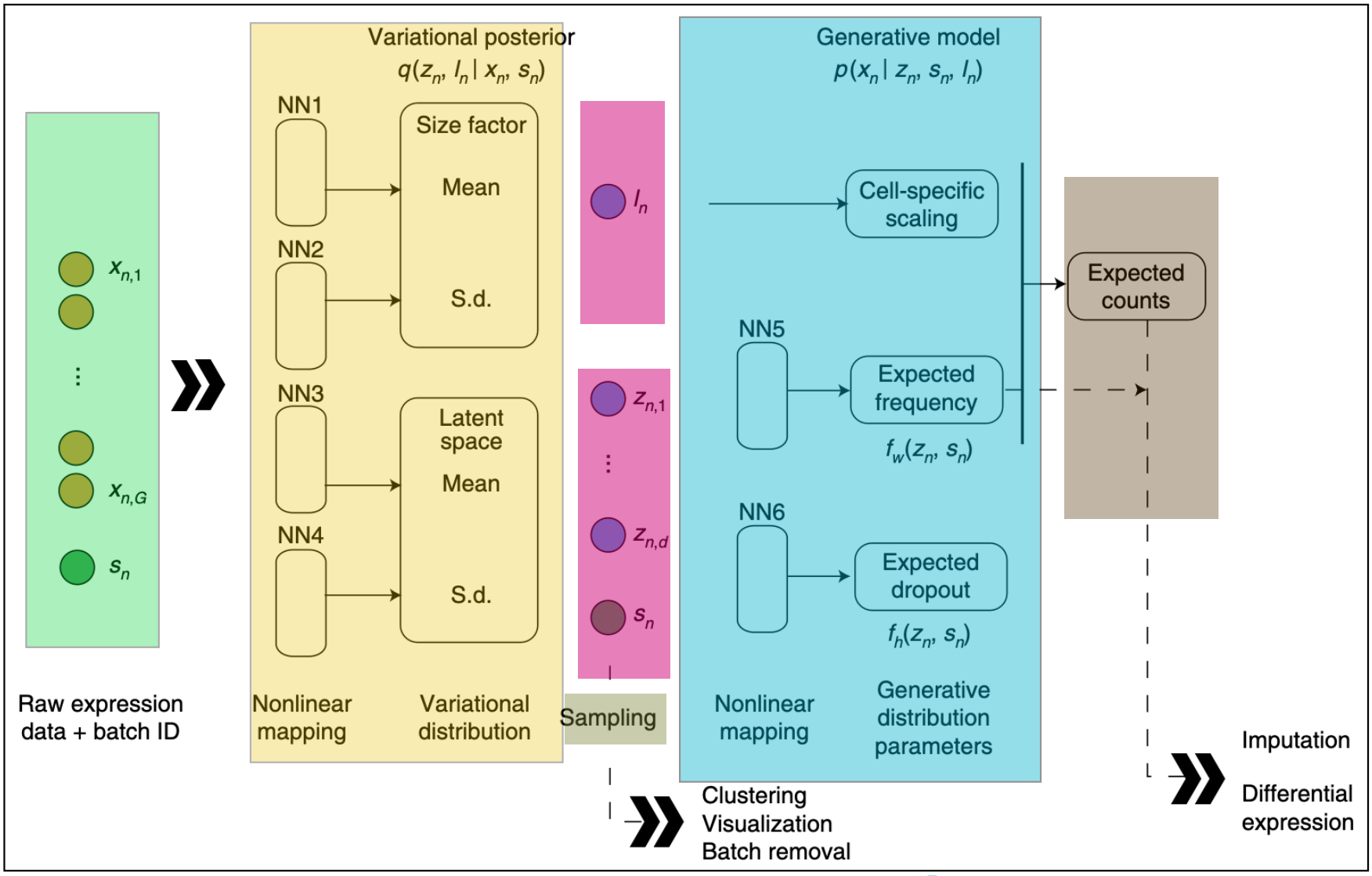

**latent variable distance matrix**

**latent variable 2D projection**

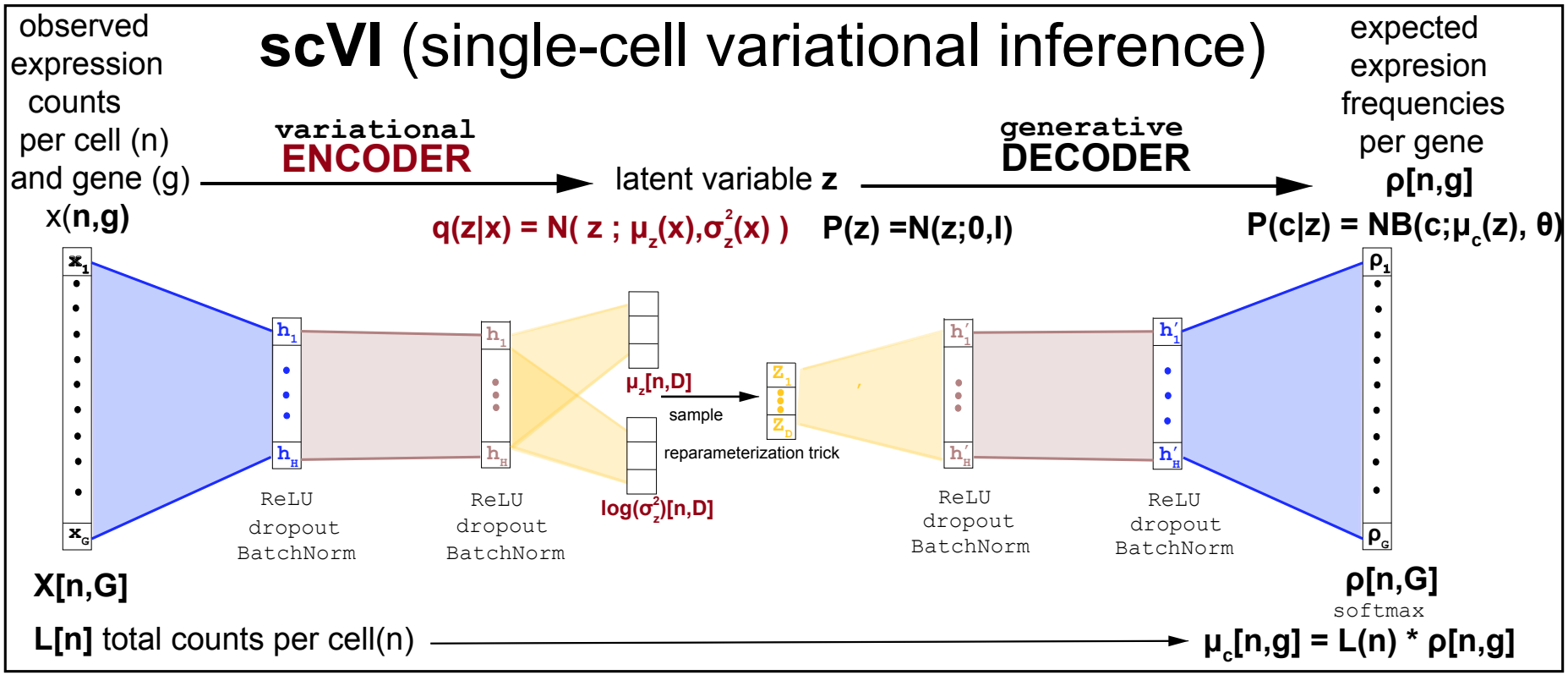
# scVI method


Variational posterior
$q(z_n, l_n | x_n, s_n)$
Generative model
$p(x_n | z_n, s_n, l_n)$
NN1
Size factor
Mean
NN2
S.d.
NN3
Latent space
Mean
NN4
S.d.
$x_{n,1}$
$x_{n,G}$
$s_n$
$l_n$
$z_{n,1}$
$z_{n,d}$
$s_n$
Cell-specific scaling
NN5
Expected frequency
$f_w(z_n, s_n)$
NN6
Expected dropout
$f_h(z_n, s_n)$
Expected counts
Raw expression data + batch ID
Nonlinear mapping
Variational distribution
Sampling
Nonlinear mapping
Generative distribution parameters
Clustering
Visualization
Batch removal
Imputation
Differential expression

# scVI method




**inputs (obs counts gene/cell)** **variational encoder** **latent variables** **generative encoder** **outputs (exp counts gene/cell)**


scVI (single-cell variational inference)
observed expression counts per cell (n) and gene (g) x(n,g)
variational ENCODER
latent variable z
generative DECODER
expected expresion frequencies per gene ρ[n,g]
q(z|x) = N( z ; μz(x),σ2z(x) )
P(z) =N(z;0,I)
P(c|z) = NB(c;μc(z), θ)
ReLU dropout BatchNorm
ReLU dropout BatchNorm
μz[n,D]
sample
reparameterization trick
log(σ2z)[n,D]
ReLU dropout BatchNorm
ReLU dropout BatchNorm
X[n,G]
ρ[n,G]
softmax
L[n] total counts per cell(n)
μc[n,g] = L(n) * ρ[n,g]

# scVI Inputs

**counts[n,g]** — **RNA expresion counts per gene (g) and cell (n)**

| | gene1 | gene2 | gene3 | | | | | | | | | | | | | | | | | | | | | gene25 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cell_1 | 0. | 0. | 0. | 0. | 19. | 10. | 15. | 0. | 0. | 0. | 18. | 16. | 0. | 0. | 16. | 0. | 0. | 0. | 0. | 0. | 0. | 0. | 0. | 0. |
| cell_2 | 19. | 13. | 0. | 0. | 0. | 14. | 8. | 11. | 14. | 18. | 0. | 0. | 11. | 0. | 16. | 0. | 26. | 14. | 12. | 20. | 14. | 0. | 9. | 0. | 18. |
| cell_3 | 0. | 0. | 0. | 0. | 0. | 18. | 9. | 0. | 16. | 0. | 17. | 0. | 0. | 0. | 21. | 13. | 0. | 0. | 19. | 0. | 0. | 0. | 0. | 0. | 0. |
| cell_4 | 14. | 0. | 0. | 0. | 16. | 0. | 0. | 0. | 0. | 0. | 0. | 15. | 0. | 0. | 0. | 0. | 0. | 0. | 0. | 14. | 0. | 0. | 0. | 0. | 19. |
| cell_5 | 0. | 0. | 0. | 12. | 18. | 10. | 0. | 15. | 0. | 14. | 0. | 0. | 18. | 12. | 0. | 0. | 0. | 0. | 20. | 17. | 13. | 0. | 0. | 0. | 0. |
| cell_6 | 0. | 0. | 0. | 0. | 0. | 11. | 0. | 0. | 0. | 0. | 10. | 0. | 0. | 14. | 0. | 0. | 0. | 19. | 0. | 0. | 16. | 0. | 0. | 0. | 0. |

**inputs = expression level per cell for all genes**

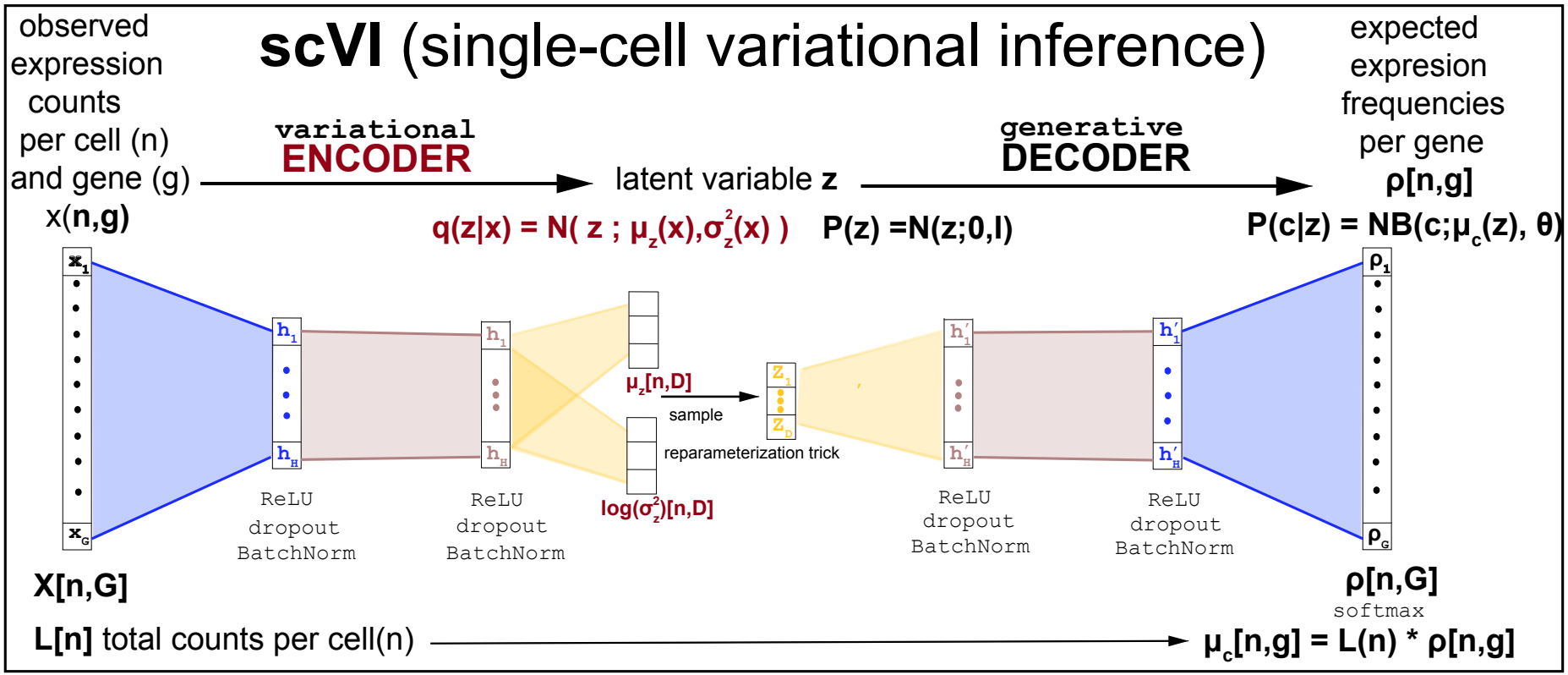
$$x[n,g] = \text{normalized\_counts}[n,g] = \frac{\text{counts}[n,g]}{\sum_{g'} \text{counts}[n,g']}$$

OR

$$x[n,g] = \text{log-normalized}[n,g] = \log\left(1 + \text{normalized\_counts}[n,g]\right) >= 0$$

**inputs = cell-specific scaling factor (one per cell)**

$$L[n] = \sum_{g} \log\left(1 + \text{counts}[n,g]\right)$$


scVI (single-cell variational inference)
observed expression counts per cell (n) and gene (g) x(n,g)
variational ENCODER
latent variable z
generative DECODER
expected expresion frequencies per gene ρ[n,g]
q(z|x) = N( z ; μz(x),σ2z(x) )
P(z) =N(z;0,I)
P(c|z) = NB(c;μc(z), θ)
ReLU dropout BatchNorm
ReLU dropout BatchNorm
μz[n,D]
sample
reparameterization trick
log(σ2z)[n,D]
ReLU dropout BatchNorm
ReLU dropout BatchNorm
X[n,G]
ρ[n,G]
softmax
L[n] total counts per cell(n)
μc[n,g] = L(n) * ρ[n,g]

# scVI probability distributions

(Encoder) Posterior of latent variable (standard)

$$q(z \mid x) = N(z;\, \mu_z(x),\, \sigma_z^2(x))$$

(Decoder) Prior for latent variable (standard)

$$P_\theta(z) = N(z;\, 0, I)$$

(Decoder)

$$P_\theta(x \mid z) = ?$$

$$P_\theta(\text{counts} \mid z)?$$

cannot be a categorical distribution
cannot be a Normal distribution

$$\boxed{\text{counts}[g] \geq 0}$$

can take arbitrary values $\rightarrow$ Not categorical variable
is a discrete variable
semi-definite positive

# Negative binomial distribution

$$P(c \mid z) = NB(c \mid \mu, \theta) = \frac{\Gamma(c+\theta)}{\Gamma(c+1)\Gamma(\theta)} \left(\frac{\mu}{\mu+\theta}\right)^{c} \left(\frac{\theta}{\theta+\mu}\right)^{\theta}$$

for $\mu = \mu_x(z)[g]$, $\theta = \theta[g]$, and $c = c[g]$.

**parameters:**

**mean**

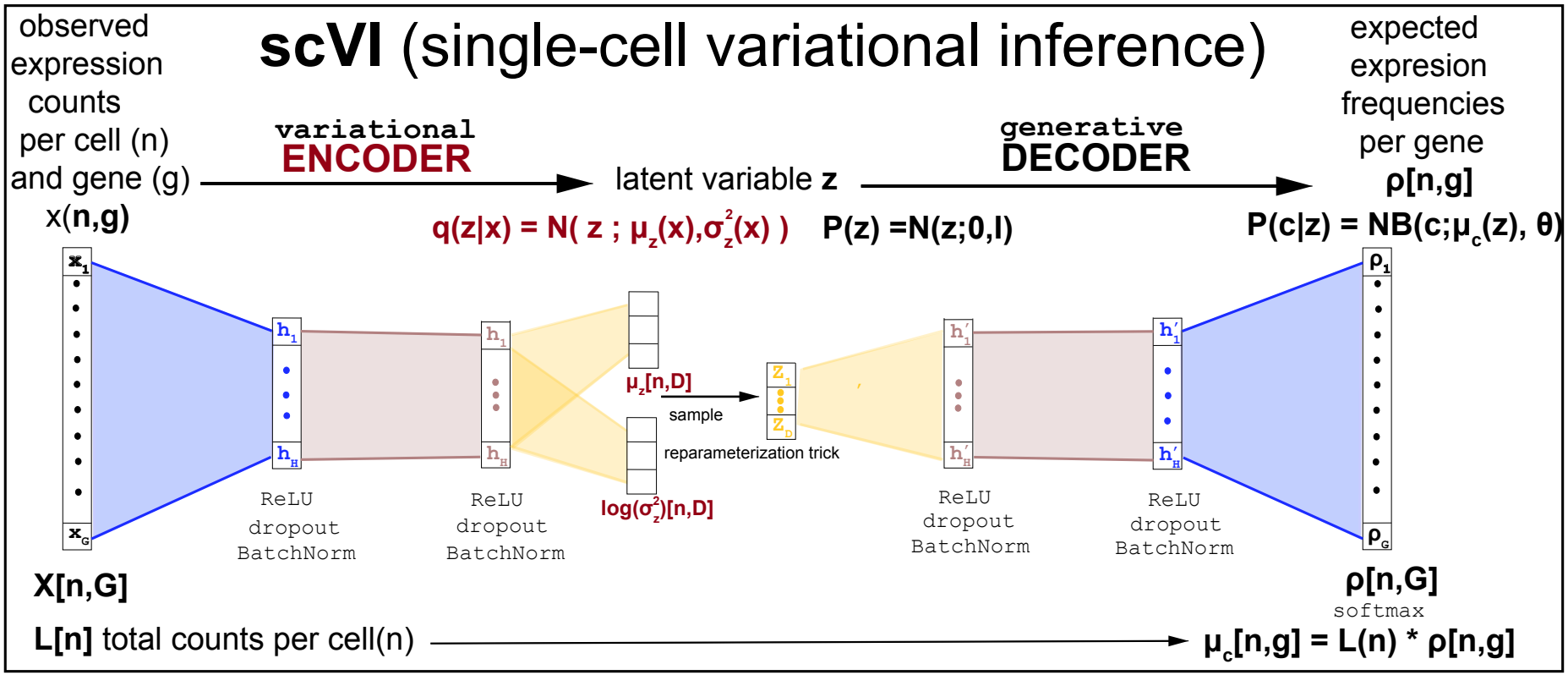
$\mu_x(z)[G]$

trained by the decoder network

**dispersion**

$\theta[G]$

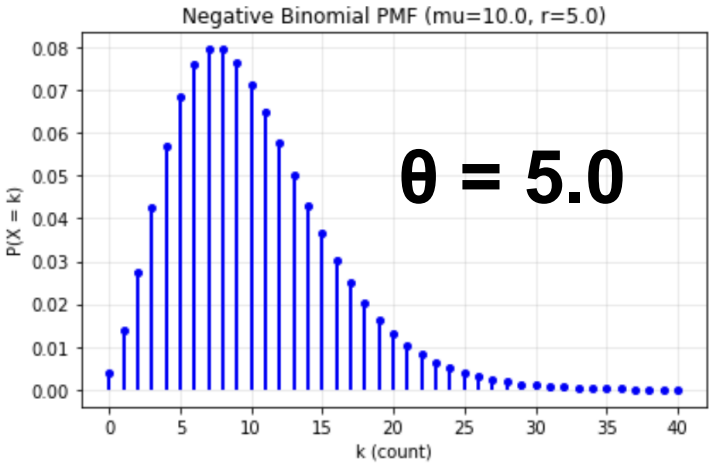
trained independently of the latent variable.


scVI (single-cell variational inference)
observed expression counts per cell (n) and gene (g) x(n,g)
variational ENCODER
latent variable z
generative DECODER
expected expresion frequencies per gene ρ[n,g]
q(z|x) = N( z ; μz(x),σ2z(x) )
P(z) =N(z;0,I)
P(c|z) = NB(c;μc(z), θ)
ReLU dropout BatchNorm
ReLU dropout BatchNorm
μz[n,D]
sample
reparameterization trick
log(σ2z)[n,D]
ReLU dropout BatchNorm
ReLU dropout BatchNorm
X[n,G]
ρ[n,G]
softmax
L[n] total counts per cell(n)
μc[n,g] = L(n) * ρ[n,g]

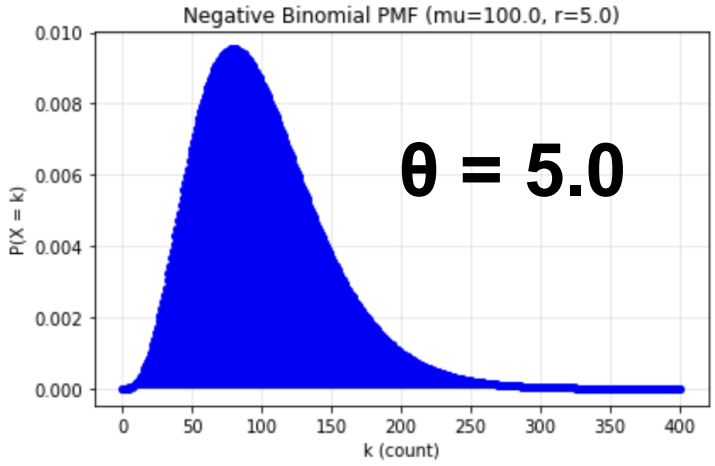
# Why NB?

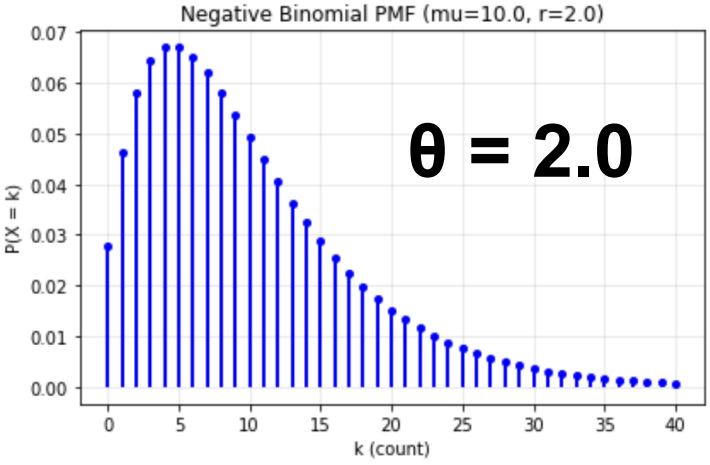
## Negative binomial

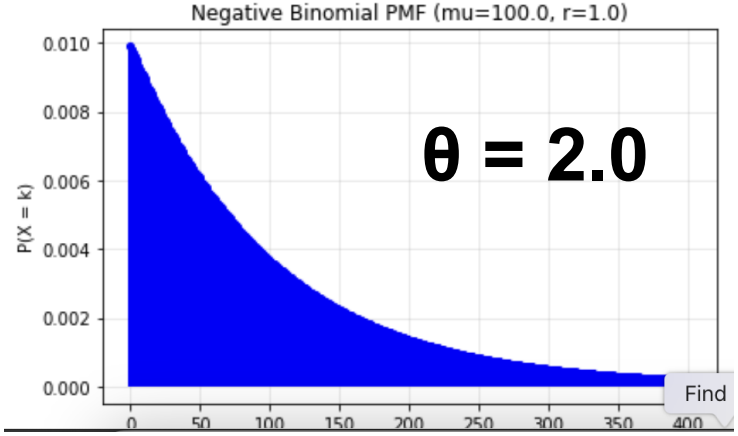
### μ = 10




### μ = 100



# scVI decoder trains

The mean proportion of transcrips expressed per cell

$$0 \leq \rho[n, g] \leq 1$$

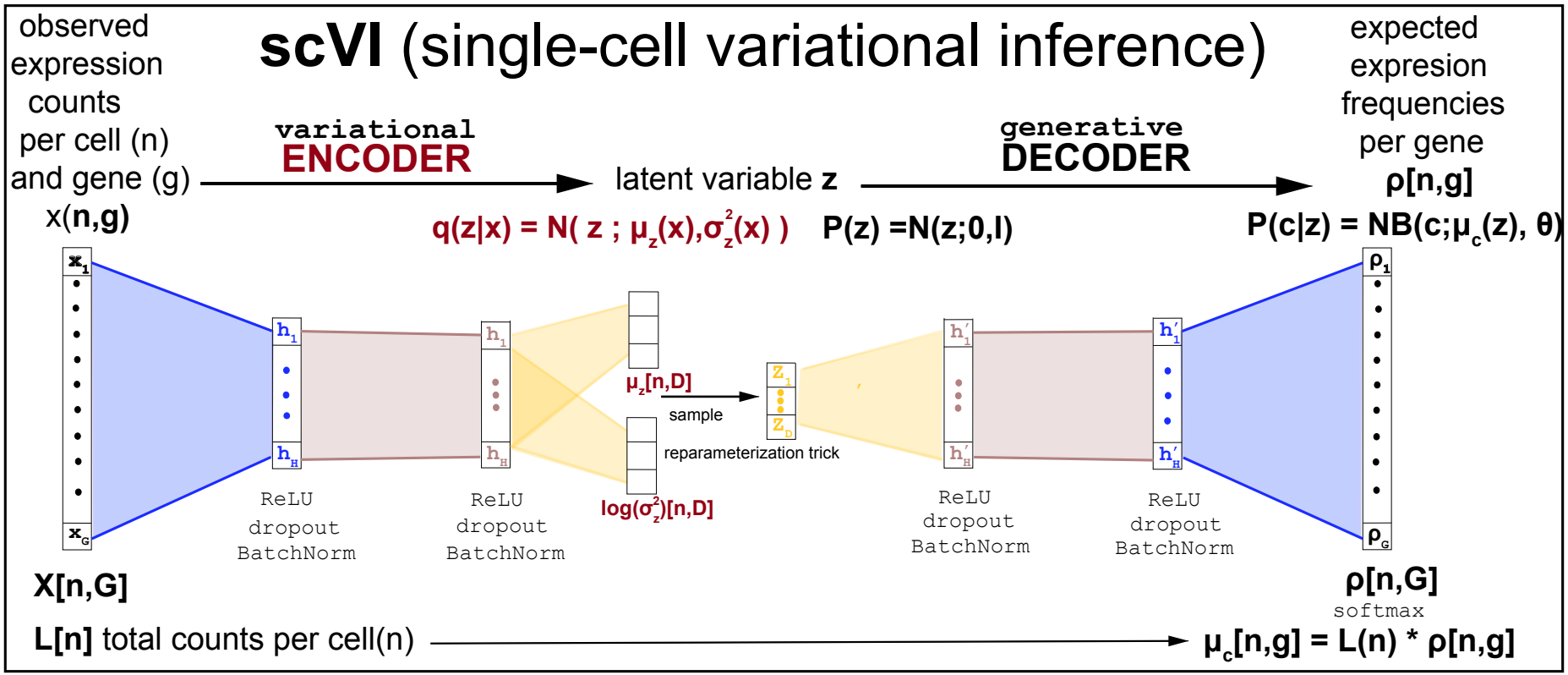
such that

$$\sum_{g=1}^{G} \rho[n, g] = 1 \quad \text{for all } n.$$

The NB mean is

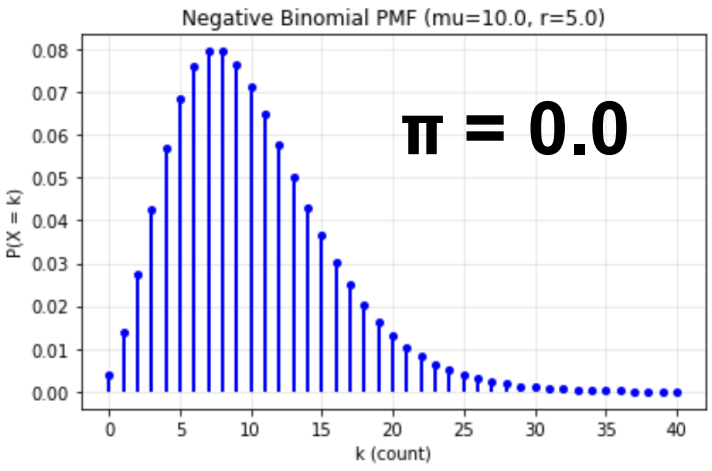
$$\mu_x[n, g] = \rho[n, g] \times l[n]$$

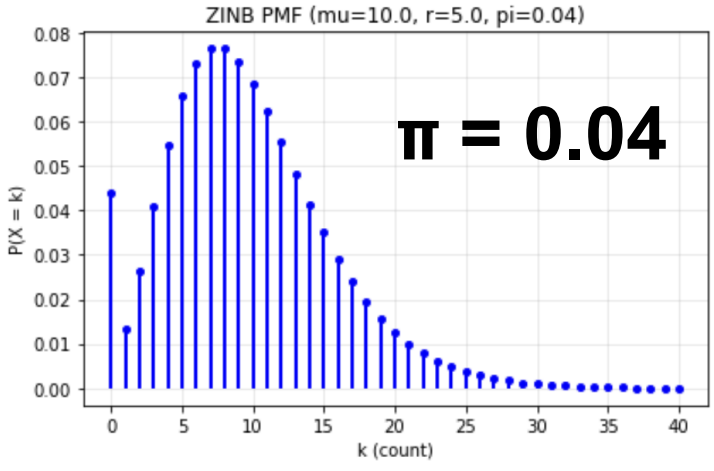
where $l[n]$ is the cell-specific scaling factor


scVI (single-cell variational inference)
observed expression counts per cell (n) and gene (g) x(n,g)
variational ENCODER
latent variable z
generative DECODER
expected expresion frequencies per gene ρ[n,g]
q(z|x) = N( z ; μz(x),σ2z(x) )
P(z) =N(z;0,I)
P(c|z) = NB(c;μc(z), θ)
ReLU dropout BatchNorm
ReLU dropout BatchNorm
μz[n,D]
sample
reparameterization trick
log(σ2z)[n,D]
ReLU dropout BatchNorm
ReLU dropout BatchNorm
X[n,G]
ρ[n,G]
softmax
L[n] total counts per cell(n)
μc[n,g] = L(n) * ρ[n,g]

# Zero-Inflated Negative binomial

## μ = 10 θ = 5.0




For $c = 0$

$$P_{ZINB}(c = 0) = \pi + (1 - \pi) \times P_{NB}(0)$$

For $c > 0$

$$P_{ZINB}(c) = (1 - \pi) \times P_{NB}(c)$$

# Zero-Inflated Negative Bionmial (ZINB)

## Negative Binomial

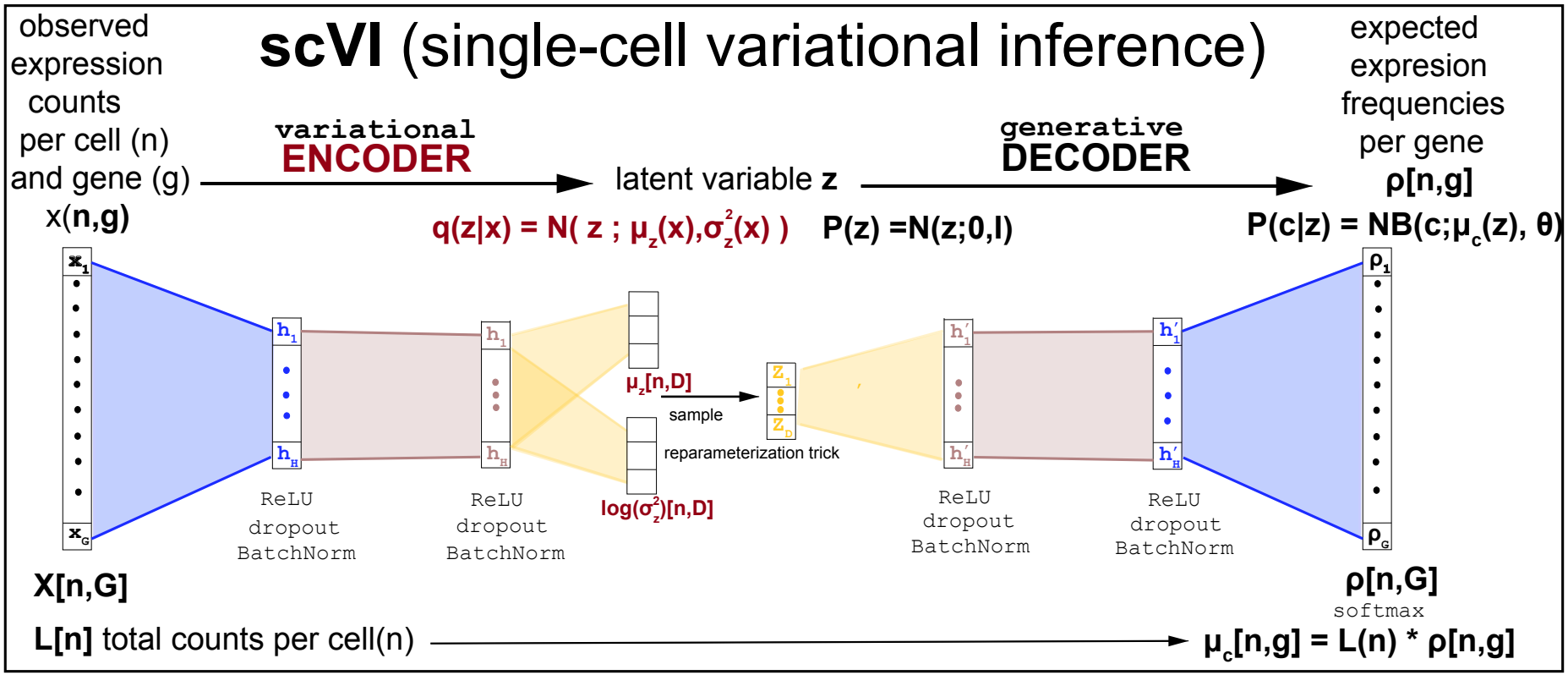
r = 500 (number of failures to stop)
p = 0.03 (probability of success)

| | gene1 | gene2 | gene3 | | | gene7 | log(counts per cell) |
|---|---|---|---|---|---|---|---|
| cell_1 | [15., | 23., | 16., | 16., | 14., | 18., | 16.] [4.7707] |
| cell_2 | [20., | 18., | 27., | 19., | 16., | 21., | 11.] [4.8828] |
| cell_3 | [11., | 16., | 17., | 9., | 7., | 15., | 16.] [4.5109] |
| cell_4 | [17., | 4., | 18., | 20., | 15., | 13., | 24.] [4.7095] |
| cell_5 | [14., | 18., | 13., | 19., | 17., | 17., | 16.] [4.7362] |
| cell_6 | [21., | 16., | 18., | 18., | 11., | 13., | 15.] [4.7185] |
| cell_7 | [15., | 19., | 13., | 13., | 15., | 11., | 17.] [4.6347] |
| cell_8 | [17., | 22., | 14., | 16., | 20., | 15., | 15.] [4.7791] |
| cell_9 | [13., | 20., | 18., | 15., | 19., | 13., | 17.] [4.7449] |
| cell_10 | [13., | 15., | 14., | 18., | 18., | 19., | 17.] [4.7362] |

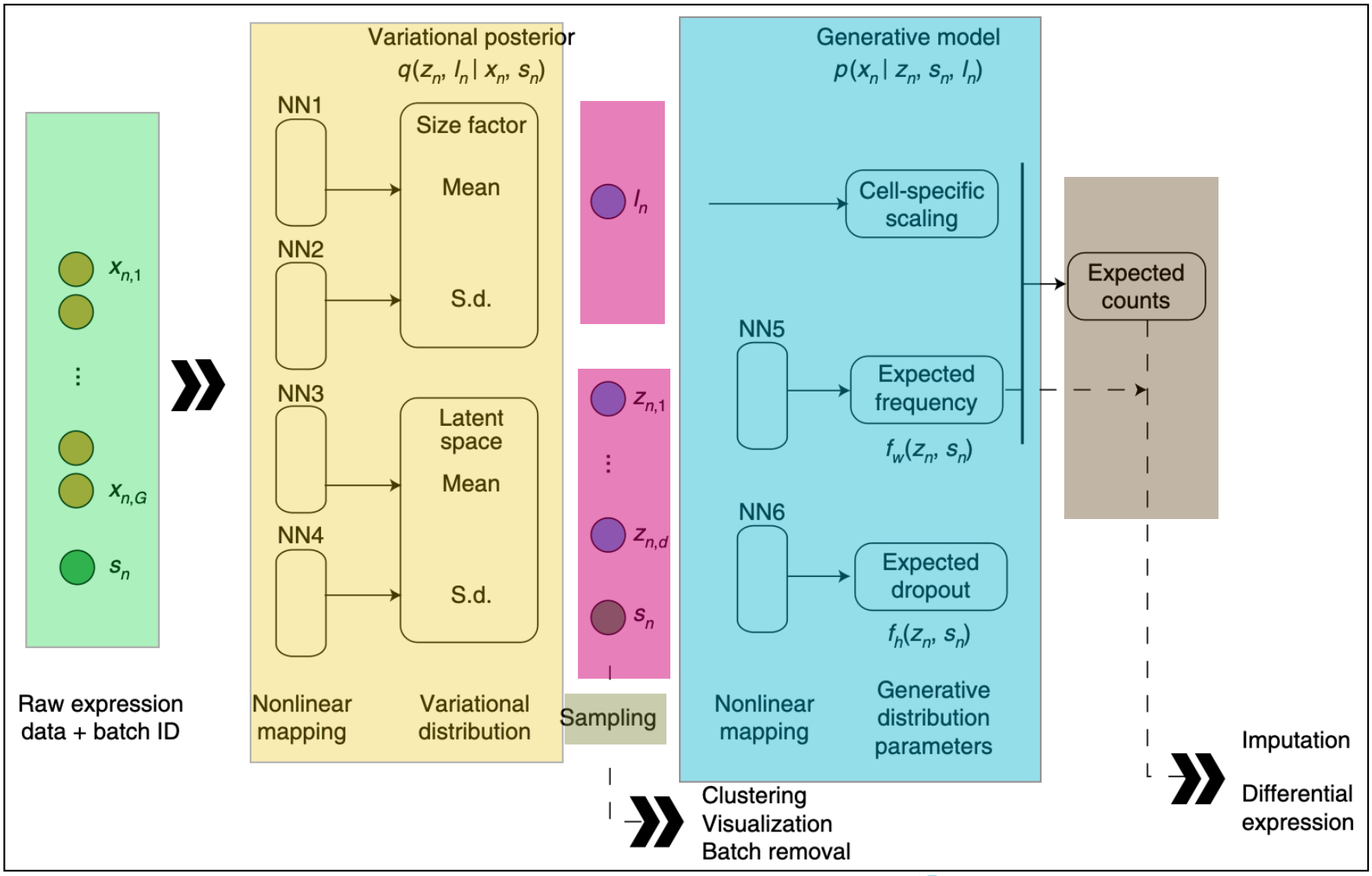
## Zero-Inflated Negative Binomial

r = 500 (number of failures to stop)
p = 0.03 (probability of success)
π = 0.7 (zero-inflation probability)

| | gene1 | gene2 | gene3 | | | gene7 | log(counts per cell) |
|---|---|---|---|---|---|---|---|
| cell_1 | [ 0., | 14., | 0., | 0., | 0., | 0., | 0.] [ 2.6391] |
| cell_2 | [ 0., | 0., | 0., | 0., | 0., | 0., | 17.] [ 2.8332] |
| cell_3 | [20., | 15., | 16., | 0., | 18., | 7., | 0.] [ 4.3307] |
| cell_4 | [ 0., | 11., | 0., | 0., | 0., | 0., | 0.] [ 2.3979] |
| cell_5 | [19., | 0., | 0., | 0., | 0., | 0., | 0.] [ 2.9444] |
| cell_6 | [ 0., | 0., | 0., | 11., | 0., | 0., | 0.] [ 2.3979] |
| cell_7 | [16., | 0., | 0., | 0., | 17., | 21., | 0.] [ 3.9890] |
| cell_8 | [ 0., | 0., | 0., | 0., | 0., | 0., | 15.] [ 2.7081] |
| cell_9 | [ 0., | 0., | 0., | 0., | 0., | 0., | 24.] [ 3.1781] |
| cell_10 | [ 0., | 0., | 0., | 0., | 0., | 0., | 0.] [-18.4207] |


scVI (single-cell variational inference)
observed expression counts per cell (n) and gene (g) x(n,g)
variational ENCODER
latent variable z
generative DECODER
expected expresion frequencies per gene ρ[n,g]
q(z|x) = N( z ; μz(x),σ2z(x) )
P(z) =N(z;0,I)
P(c|z) = NB(c;μc(z), θ)
ReLU dropout BatchNorm
ReLU dropout BatchNorm
μz[n,D]
sample
reparameterization trick
log(σ2z)[n,D]
ReLU dropout BatchNorm
ReLU dropout BatchNorm
X[n,G]
ρ[n,G]
softmax
L[n] total counts per cell(n)
μc[n,g] = L(n) * ρ[n,g]

# scVI method




**inputs (obs counts gene/cell)** **variational encoder** **latent variables** **generative encoder** **outputs (exp counts gene/cell)**

# scVI Forward loop

```python
class SimpleScVIVAE(nn.Module):
    def __init__(
        self,
        n_genes,
        n_hidden=128,
        n_latent=10,
    ):
        super().__init__()
        self.n_genes = n_genes
        self.n_latent = n_latent

        # ----- Encoder q(z | x) -----
        self.encoder = nn.Sequential(
            nn.Linear(n_genes, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, n_hidden),
            nn.ReLU(),
        )
        self.z_mu = nn.Linear(n_hidden, n_latent)
        self.z_logvar = nn.Linear(n_hidden, n_latent)

        # ----- Decoder p(x | z, l) -----
        # First map z to hidden
        self.decoder = nn.Sequential(
            nn.Linear(n_latent, n_hidden),
            nn.ReLU(),
        )
        # Gene-specific mean (log scale) before adding library size
        self.px_scale = nn.Linear(n_hidden, n_genes)
        # Gene-specific dispersion (log-theta, one per gene, not a function of z
        self.px_r = nn.Parameter(torch.randn(n_genes))
```

```python
def encode(self, x):
    h = self.encoder(x)
    mu = self.z_mu(h)
    logvar = self.z_logvar(h)
    return mu, logvar
```

$\mu_z(x), \log\sigma^2_z(x)$

```python
def reparameterize(self, mu, logvar):
    std = torch.exp(0.5 * logvar)
    eps = torch.randn_like(std)
    return mu + eps * std
```

$z \sim N(\mu_z(x), \log\sigma^2_z(x))$

```python
def decode(self, z, library):
    """
    z       : [batch, n_latent]
    library : [batch, 1] log-library size (log total counts)
    """
    h = self.decoder(z)
    # log mean proportion per gene
    px_scale_logit = self.px_scale(h)  # [batch, n_genes]
    # softmax to get proportions that sum to 1
    px_scale = F.softmax(px_scale_logit, dim=-1)
```

$\rho[n,g]$

```python
    # mean = library_size * proportion
    # library is log(total counts), so use exp
    library_exp = torch.exp(library)  # [batch, 1]
    mu = library_exp * px_scale  # [batch, n_genes]
```

$L(n)$

$\mu_c[n,g] = L(n)*\rho[n,g]$

```python
    # inverse dispersion (theta) per gene
    theta = torch.exp(self.px_r)  # [n_genes]
    return mu, theta
```

$\theta(g)$

```python
def forward(self, x, library):
    """
    x       : counts [batch, n_genes]
    library : log library size [batch, 1]
    """
    # Encoder
    mu_z, logvar_z = self.encode(x)
    z = self.reparameterize(mu_z, logvar_z)

    # Decoder
    mu_x, theta = self.decode(z, library)

    return mu_x, theta, mu_z, logvar_z, z
```

$\mu_z(x), \log\sigma^2_z(x)$

$z$

$\mu_c[n,g], \theta(g)$

# scVI Loss

$$VAE\_LOSS_{\varphi\theta}(x) = -E_{q_\phi(z|x)} \log P_\theta(x \mid z) - \tfrac{1}{2} \sum_{i=1}^{d} (1 + \log \sigma_i^2 - \mu_i^2 - \sigma_i^2)$$

```python
## LOSS = ELBO = reconstruction + KL)
#
def scvi_loss(x, mu_x, theta, mu_z, logvar_z):
    # Reconstruction term (sum over genes, then mean over cells)
    recon = log_nb_positive(x, mu_x, theta).sum(dim=1)  # [batch]
    recon_loss = -recon.mean()

    # KL(q(z|x) || N(0, I)) per cell
    kl_div = -0.5 * torch.sum(1 + logvar_z - mu_z.pow(2) - logvar_z.exp(), dim=1)
    kl_loss = kl_div.mean()

    loss = recon_loss + kl_loss
    return loss, recon_loss, kl_loss
```

```python
def log_nb_positive(x, mu, theta, eps=1e-8):
    """
    Negative binomial log-likelihood (per entry).

    x    : observed counts
    mu   : mean
    theta: inverse dispersion (>0), broadcastable to x
    """
    if torch.any(theta <= 0):
        raise ValueError("theta must be > 0")

    log_theta_mu = torch.log(theta + mu + eps)
    res = (
        theta * (torch.log(theta + eps) - log_theta_mu)
        + x * (torch.log(mu + eps) - log_theta_mu)
        + torch.lgamma(x + theta)
        - torch.lgamma(theta)
        - torch.lgamma(x + 1)
    )
    return res
```

$$P(c \mid z) = \frac{\Gamma(c+\theta)}{\Gamma(c+1)\Gamma(\theta)} \left(\frac{\mu}{\mu+\theta}\right)^{c} \left(\frac{\theta}{\theta+\mu}\right)^{\theta}$$

# scVI Training loop

```python
# Training loop
from torch.utils.data import DataLoader, TensorDataset

# X: counts [N, G], torch.float32 (or float32-cast of ints)
# libsize: log library size [N, 1]
X = zinb_counts
libsize = torch.log(X.sum(dim=1, keepdim=True) + 1e-8)

dataset = TensorDataset(X, libsize)
loader = DataLoader(dataset, batch_size=128, shuffle=True)

n_genes = X.shape[1]
model = SimpleScVIVAE(n_genes=n_genes, n_hidden=128, n_latent=10)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

for epoch in range(1, 151):
    model.train()
    total_loss = 0.0
    for batch_x, batch_lib in loader:
        batch_x = batch_x.to(device)
        batch_lib = batch_lib.to(device)

        optimizer.zero_grad()
        mu_x, theta, mu_z, logvar_z, z = model(batch_x, batch_lib)
        loss, recon_loss, kl_loss = scvi_loss(batch_x, mu_x, theta, mu_z, logvar_z)
        loss.backward()
        optimizer.step()

        total_loss += loss.item() * batch_x.size(0)

    print(f"Epoch {epoch:03d} | Loss: {total_loss / len(dataset):.3f}")
```

# cell clustering

```
# Latent variables
model.eval()
with torch.inference_mode():
    X_dev = X.to(device)
    lib_dev = libsize.to(device)
    mu_z, logvar_z = model.encode(X_dev)
    Z = mu_z.cpu()  # use posterior mean as embedding (like scVI)
```

# Differential gene expression

Two cells $n_a$ and $n_b$,

**Hypothesis 1**

$$H_1(g) =\mid \rho(n_a, g) - \rho(n_b, g) \mid > k$$

**Hypothesis 2**

$$H_2(g) =\mid \rho(n_a, g) - \rho(n_b, g) \mid \leq k$$

We can test those hypotheses as follows

- Given the counts $x_a^g = x(n_a, g)$ and $x_b^g = x(n_b, g)$, sample latent variables from the variational distribution
  $q(z \mid x_a^g) \longrightarrow (z_a^g)^{(s)}$ S samples
  $q(z \mid x_b^g) \longrightarrow (z_b^g)^{(s)}$ S samples
- With those sampled latent variables, using the decoder, calculate normalized expressions for the gene
  $(\rho_a^g)^{(s)}$
  $(\rho_b^g)^{(s)}$
- Then the ratio of the two hypotheses can be calculated as
  $$\frac{P(H_1)}{P(H_2)} = \frac{\sum_{s_a, s_b} C^{(s_a s_b)}(|\rho_a^g - \rho_b^g| > K)}{\sum_{s_a, s_b} C^{(s_a s_b)}(|\rho_a^g - \rho_b^g| \leq K)}$$

If $\frac{P(H_1)}{P(H_2)} > 2$

twice as likely that $g$ is differentially expressed