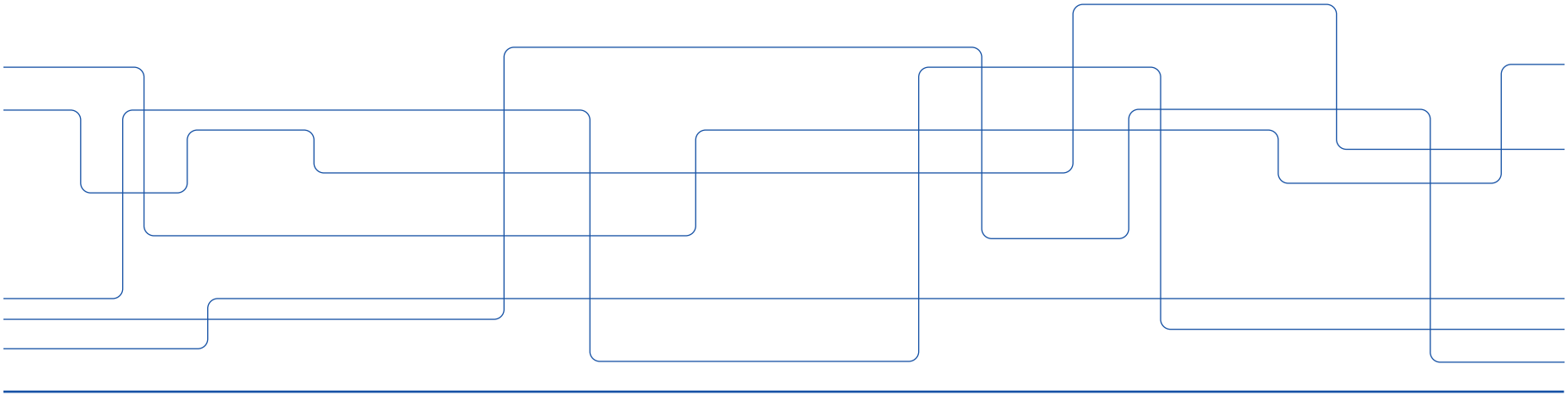
Variational Autoencoders:
From Theory to Implementation

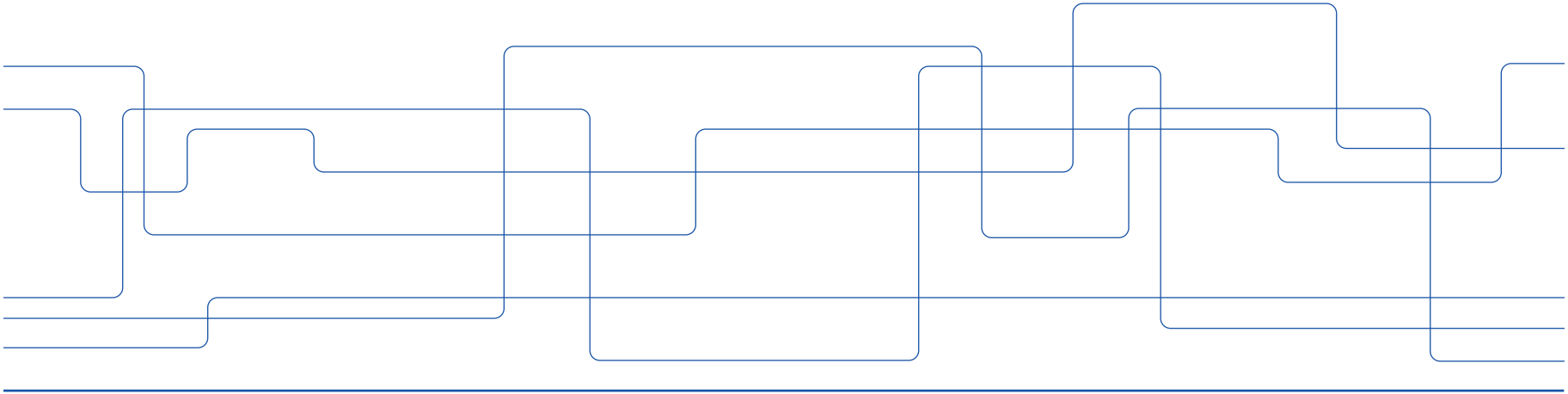
Petra Poklukar



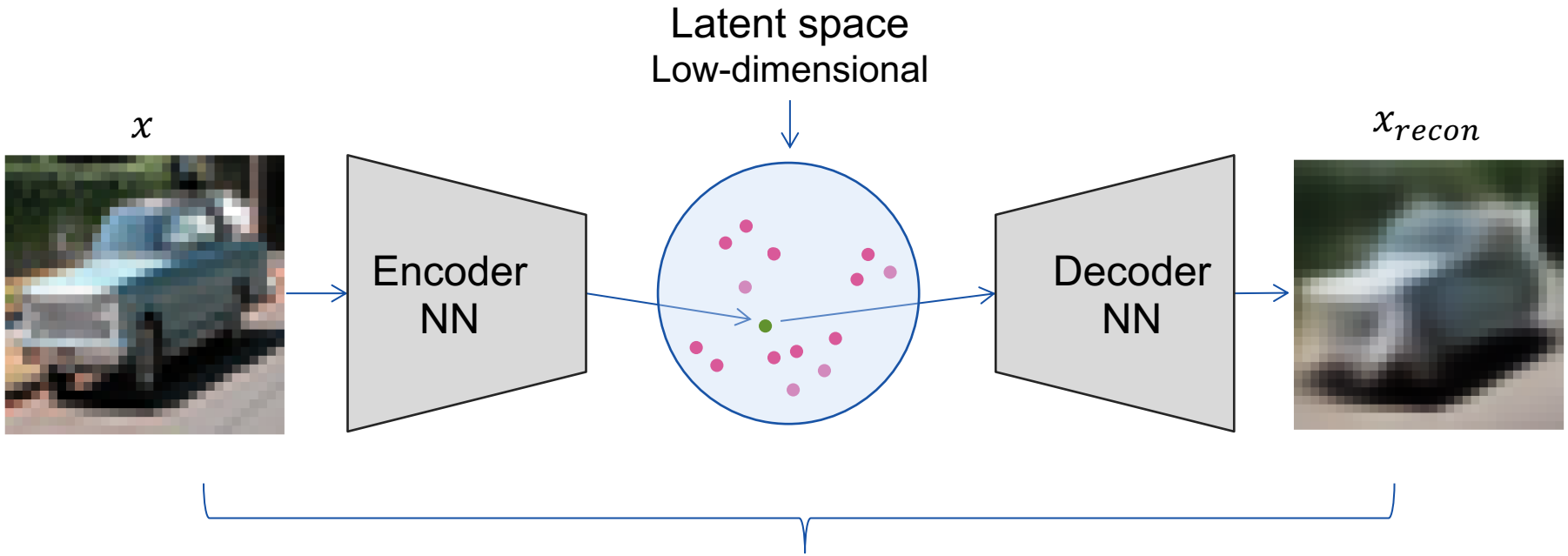
Outline

- Introduction & intuition
- Math, math and more math
- Implementation: tricks & pitfalls

Introduction

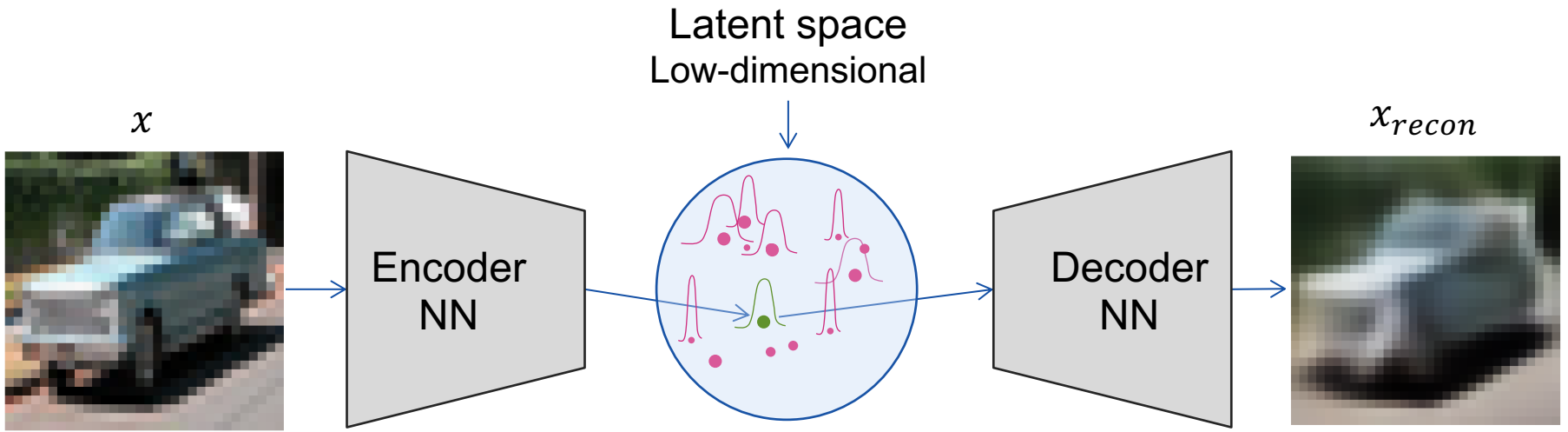


AE: Intuition



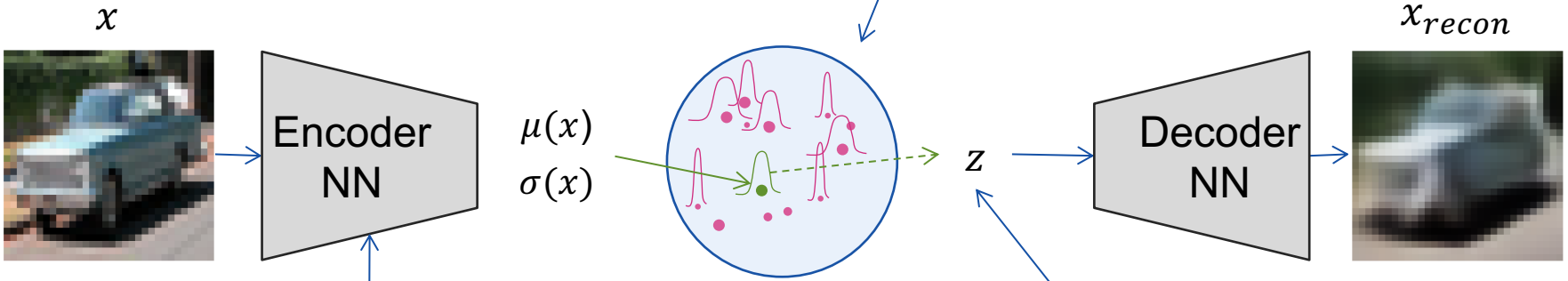
$$\mathcal{L}(x, x_{recon}) = min\|x - x_{recon}\|^2$$

VAE: Intuition



VAE: Intuition

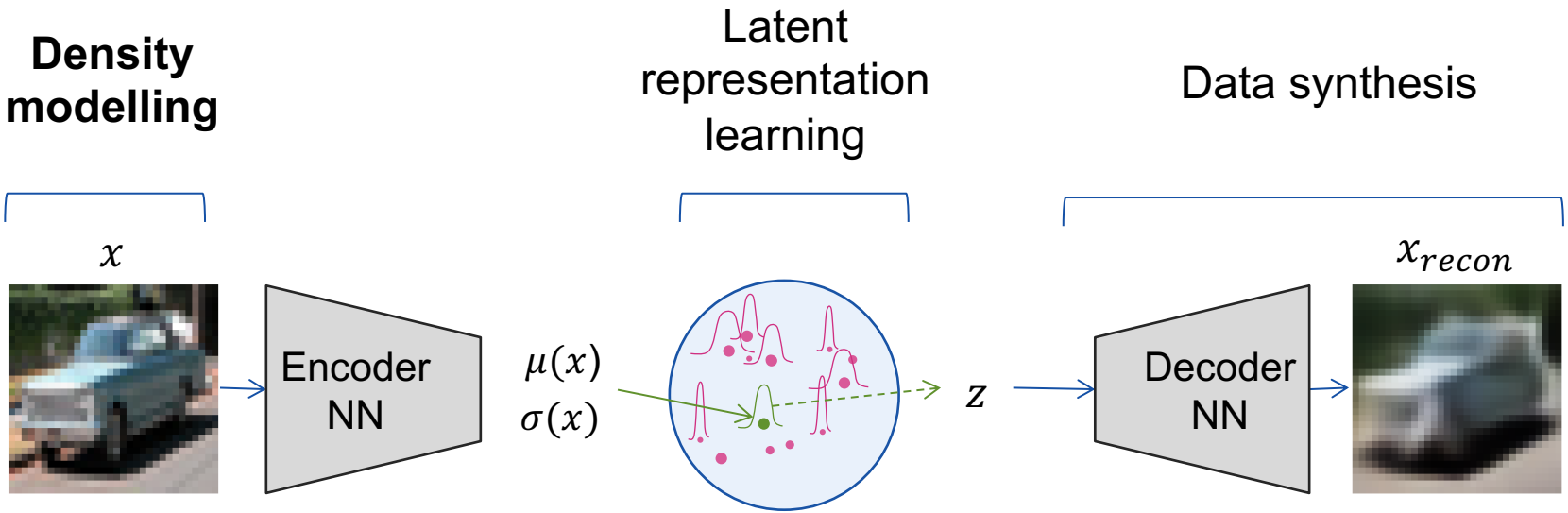
1. Fix a parametrization for each point: Gaussian $\mathrm{N}(\mu, \sigma)$



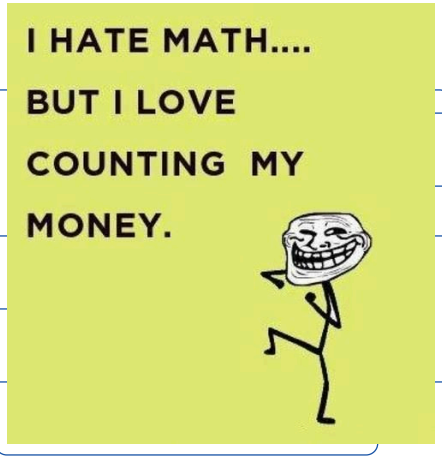
2. Encoder NN outputs the parameters of the Gaussian distribution: μ, σ

3. Sample $z \sim \mathrm{N}(\mu(x), \sigma(x))$

VAE: Use cases



Theory



The starting objective

	x observed random variables
Goal	**learn** the **true distribution** of the data $p_*(x)$
Problem	$p_*(x)$ is **intractable** (but we do have access to finitely many samples from p_*)
(Naive) Idea	**approximate** $p_*(\mathrm{x})$ (with a parametrized $p_{\boldsymbol{\theta}}(\mathrm{x})$)

How do we measure the quality of the approximation?

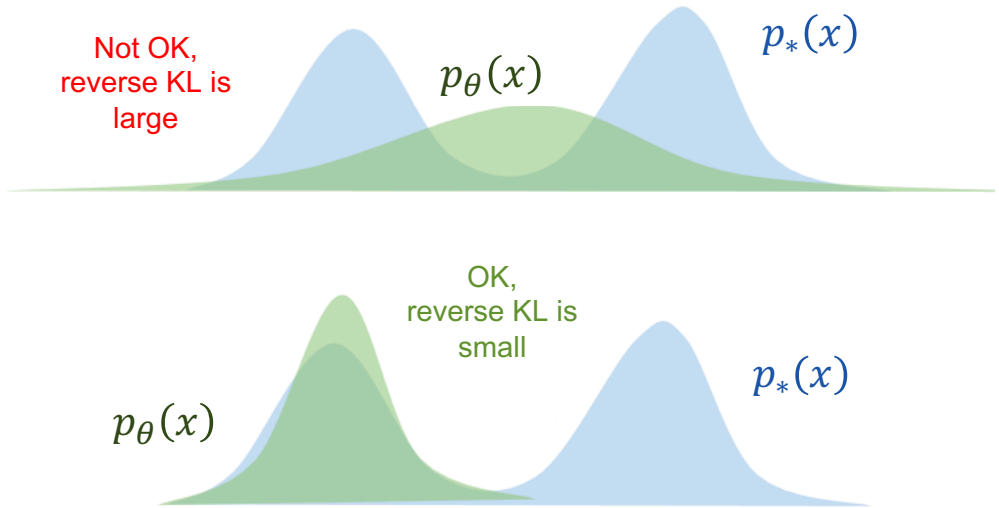
(Reverse) KL Divergence to the rescue:

$$D_{KL}\big(p_\theta(x) \parallel p_*(x)\big) = \int p_\theta(x) \log \frac{p_\theta(x)}{p_*(x)} \, dx$$

Same problem:
$p_*(x)$ is unknown

Introduce a **latent** variable z

Helpful with missing data
A way to leverage prior knowledge
Increasing expressive power



(Original image source: blog.evjang.com/2016/08/variational-bayes.html, modified here)

Latent variable z sneaking in

Let's instead consider the joint density $p(x, z)$:

$$p(x) = \int p(x, z)\, dz = \int p(x|z)p(z)\, dz$$

Problem:
- z is unobserved
- No analytical solution
- No efficient estimator

Convert it to an optimisation problem

Variational principle

Variational principle

Introduce a parametrised conditional probability distribution $q(z|x)$:

$$\ln p(x) = \ln p(x) \underbrace{\int q(z|x)dz}_{=1} = \int q(z|x) \ln p(x)\, dz$$

$$(\blacksquare)\ p(z|x) = \frac{p(x|z)p(z)}{p(x)}$$

$$\overset{(\blacksquare)}{=} \int q(z|x) \ln \frac{p(x|z)p(z)}{p(z|x)} dz = \int q(z|x) \ln \frac{p(x|z)p(z)q(z|x)}{p(z|x)q(z|x)} dz$$

$$= \int q(z|x) \ln \frac{q(z|x)}{p(z|x)}\, dz + \int q(z|x) \ln \frac{p(x,z)}{q(z|x)}\, dz$$

$$= D_{KL}\big(q(z|x) \parallel p(z|x)\big) + \int q(z|x) \ln \frac{p(x,z)}{q(z|x)}\, dz$$

Intractable but always ≥ 0

Tractable after some math

Variational principle

We get that

$$\ln p(x) = D_{KL}\big(q(z|x) \parallel p(z|x)\big) + \int q(z|x) \ln \frac{p(x,z)}{q(z|x)}\, dz$$

$$\geq \int q(z|x) \ln \frac{p(x,z)}{q(z|x)}\, dz$$

New objective: Maximise the variational lower bound $\mathcal{L}$

where $q(z|x)$ should approximate $p(z|x)$.

Keeping track of what we're doing

Goal	**learn** the **true distribution** of the data $p_*(x)$
Problem	$p_*(x)$ is **intractable** (but we do have access to finitely many samples from p_*)
Idea #1	**approximate** $p_*(\mathrm{x})$ with a parametrised $p_{\boldsymbol{\theta}}(\mathrm{x})$
Idea #2	Introduce a latent variable and leverage the joint density $\ln p(x) = \ln \int p(x,z)\,dz = \ln \int p(x\|z)p(z)\,dz$
Idea #3	Introduce $q(z\|x)$ and solve the optimization problem

Variational principle

We get that

$$\ln p(x) = D_{KL}\big(q(z|x) \parallel p(z|x)\big) + \int q(z|x) \ln \frac{p(x,z)}{q(z|x)}\, dz$$

$$\geq \int q(z|x) \ln \frac{p(x,z)}{q(z|x)}\, dz$$

New objective: Maximise the variational lower bound $\mathcal{L}$

where $q(z|x)$ should approximate $p(z|x)$.

Variational lower bound rewritten

$$\mathcal{L} = \int q(z|x) \ln \frac{p(x,z)}{q(z|x)} \, dz$$

$$= \int q(z|x) \ln \frac{p(x|z)p(z)}{q(z|x)} \, dz$$

$$= \int q(z|x) \ln \frac{p(z)}{q(z|x)} \, dz + \int q(z|x) \ln p(x|z) \, dz$$

$$= -\int q(z|x) \ln \frac{q(z|x)}{p(z)} \, dz + \int q(z|x) \ln p(x|z) \, dz$$

$$= -D_{KL}\big(q(z|x) \parallel p(z)\big) + \mathbb{E}_{q(z|x)}[\ln p(x|z)]$$

Evidence Lower Bound (ELBO), mostly known as the VAE training objective ☺

What we have done so far

$$\ln p(x) = D_{KL}\big(q(z|x) \parallel p(z|x)\big) + \int q(z|x) \ln \frac{p(x,z)}{q(z|x)}\, dz$$

Intractable but always ≥ 0

Variational lower bound $\mathcal{L}$

$$\geq \int q(z|x) \ln \frac{p(x,z)}{q(z|x)}\, dz$$

$$= -D_{KL}\big(q(z|x) \parallel p(z)\big) + \mathbb{E}_{q(z|x)}[\ln p(x|z)]$$

VAE training objective

We derived that

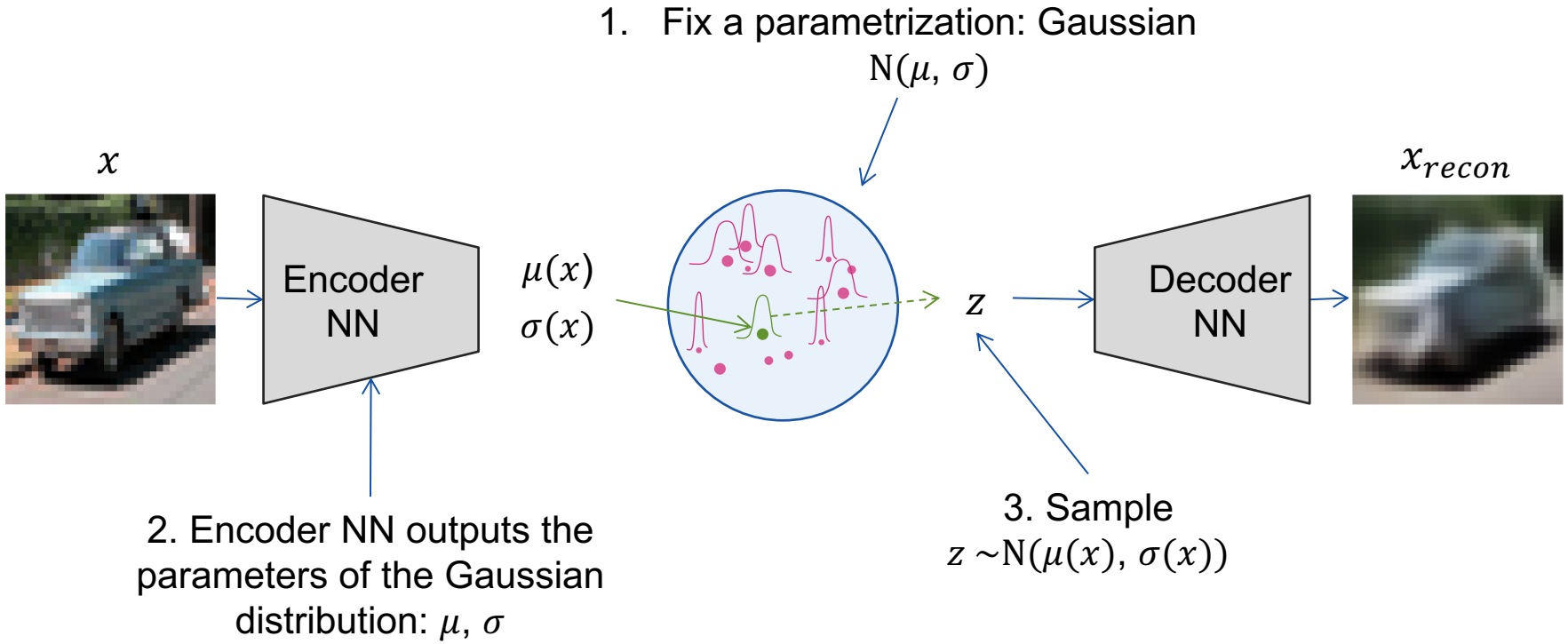
$$\ln p(x) \geq -D_{KL}\big(q(z|x) \parallel p(z)\big) + \mathbb{E}_{q(z|x)}[\ln p(x|z)]$$

Let's choose:

$$p(z) = N(0, 1)$$
$$q(z|x) = N(\mu(x), \sigma(x))$$
$$p(x|z) = N(\mu(\mathrm{z}), \sigma(z))$$

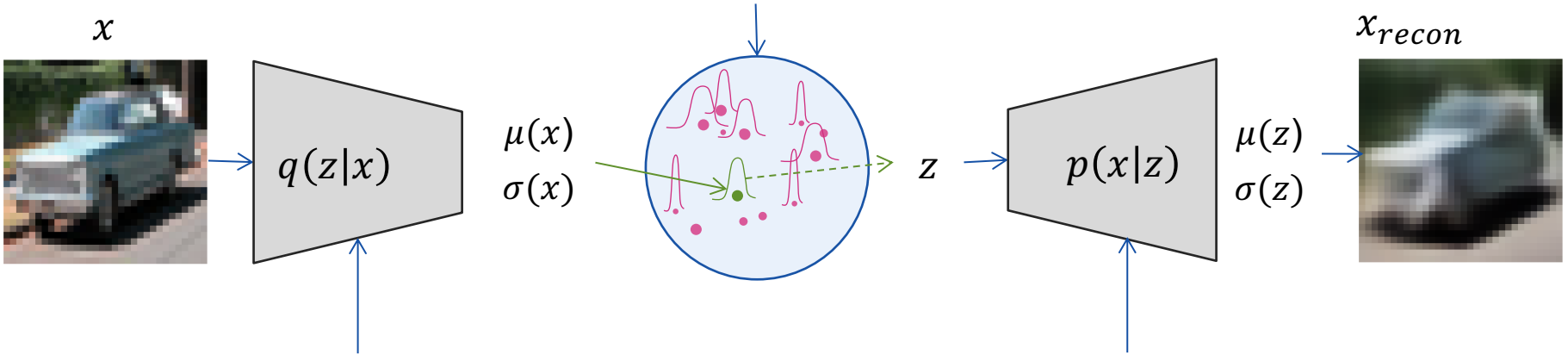
And use **two NNs to learn the parameters** $\mu(x), \sigma(x), \mu(z), \sigma(z)$!

Recall how we started



Everything falling together

Prior on the latent codes:
$p(z) = N(0, 1)$



Encoder NN outputs the parameters of the **approximate posterior distrubtion** $q(z|x)$ which we chose to parameterize as a Gaussian distribution

Decoder NN outputs the parameters of the **likelihood** $p(x|z)$ which we chose to parameterize as a Gaussian distribution

VAE training objective

We derived that

$$\ln p(x) \geq -D_{KL}\big(q(z|x) \parallel p(z)\big) + \mathbb{E}_{q(z|x)}[\ln p(x|z)]$$

Given finitely many observe samples $x_1, x_2, \ldots, x_N$ (training data):

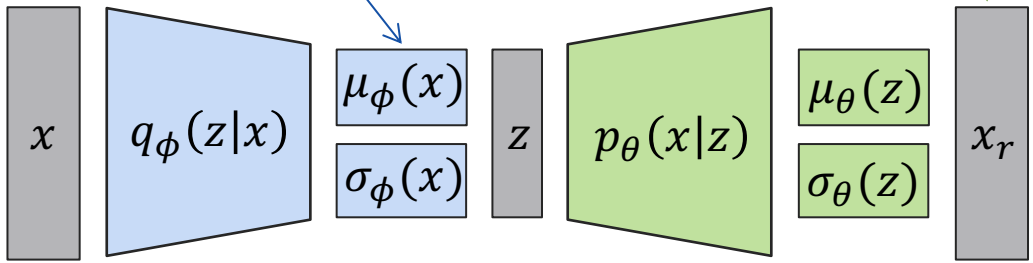
$$\ln p(x_i) \approx \max_i \left(-D_{KL}\left(q_\phi(z|x_i) \parallel p(z)\right) + \mathbb{E}_{q_\phi(z|x_i)}[\ln p_\theta(x_i|z)] \right)$$
$$= \min_i \left(D_{KL}\left(q_\phi(z|x_i) \parallel p(z)\right) - \mathbb{E}_{q_\phi(z|x_i)}[\ln p_\theta(x_i|z)] \right),$$

where θ and ϕ denote the parameters of the encoder NN and decoder NN, respectively.

VAE training objective

$$\min_i D_{KL}\left(q_\phi(z|x_i) \parallel p(z)\right) - \mathbb{E}_{q_\phi(z|x_i)}[\ln p_\theta(x_i|z)] \iff$$

$$\min_i D_{KL}\left(N(\mu_\phi(x), \sigma_\phi(x)) \parallel N(0,1)\right) - \mathbb{E}_{z\sim N(\mu_\phi(x),\sigma_\phi(x))}[\ln N(\mu_\theta(z), \sigma_\theta(z))]$$



VAE training objective: intuition

$$\min_i \; (D_{KL}\big(N(z|\mu_\phi(x), \sigma_\phi(x)) \parallel N(0,1)\big)$$

$$-\mathbb{E}_{z\sim N\left(\mu_\phi(x),\sigma_\phi(x)\right)}[\ln N\big(x|\mu_\theta(z), \sigma_\theta(z)\big)])$$

$$\approx \frac{(x - \mu_\theta(z))^2}{2\sigma_\theta(z)^2}$$

Hence the reconstruction!

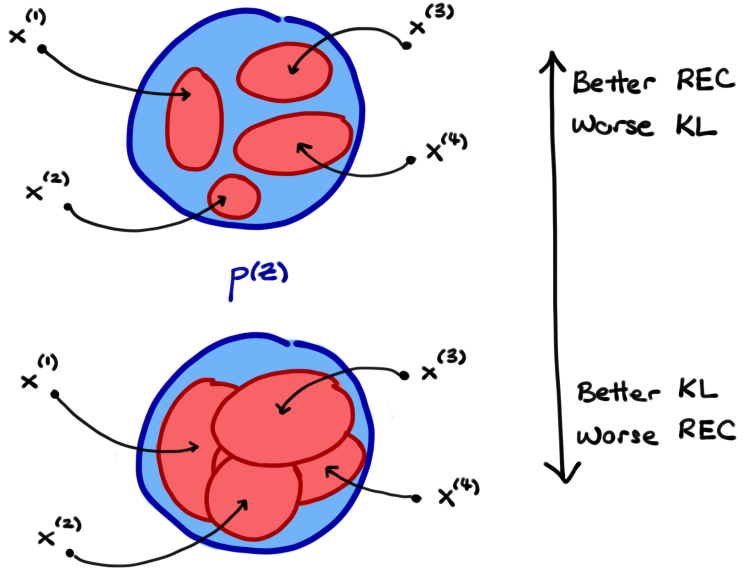
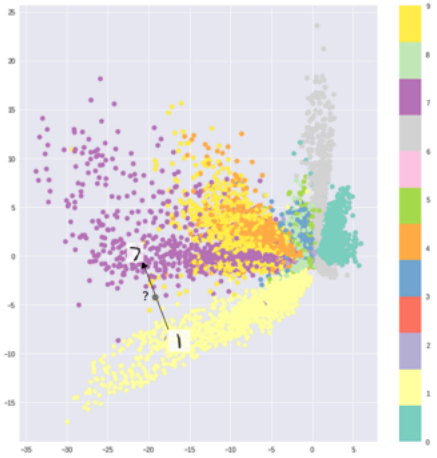


Image source: http://ruishu.io/2018/03/14/vae/

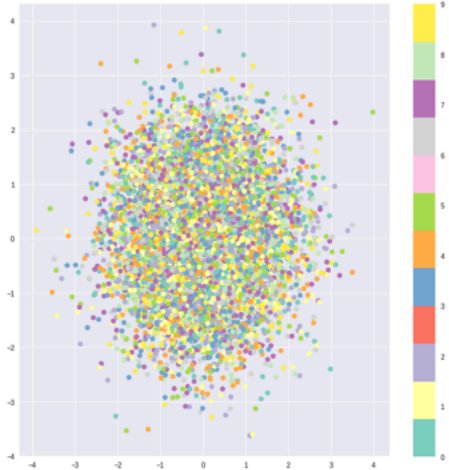
VAE training objective: intuition

$$\min_i D_{KL}\left(N(z|\mu_\phi(x), \sigma_\phi(x)) \parallel N(0,1)\right) - \mathbb{E}_{z\sim N(\mu_\phi(x),\sigma_\phi(x))}[\ln N(x|\mu_\theta(z), \sigma_\theta(z))]$$

Only reconstruction loss



Only KL divergence



Combination

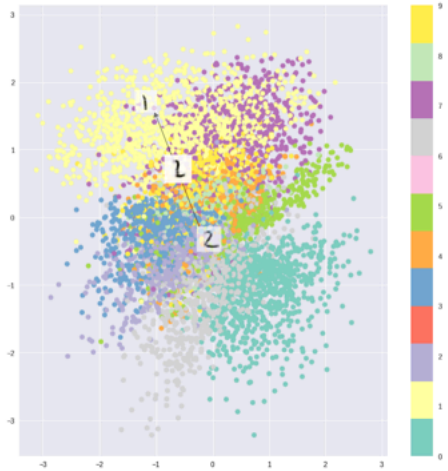


Image source: https://www.jeremyjordan.me/variational-autoencoders/

After the completed training…

… we can **evaluate** the log likelihood of **a new point** x_{new}:

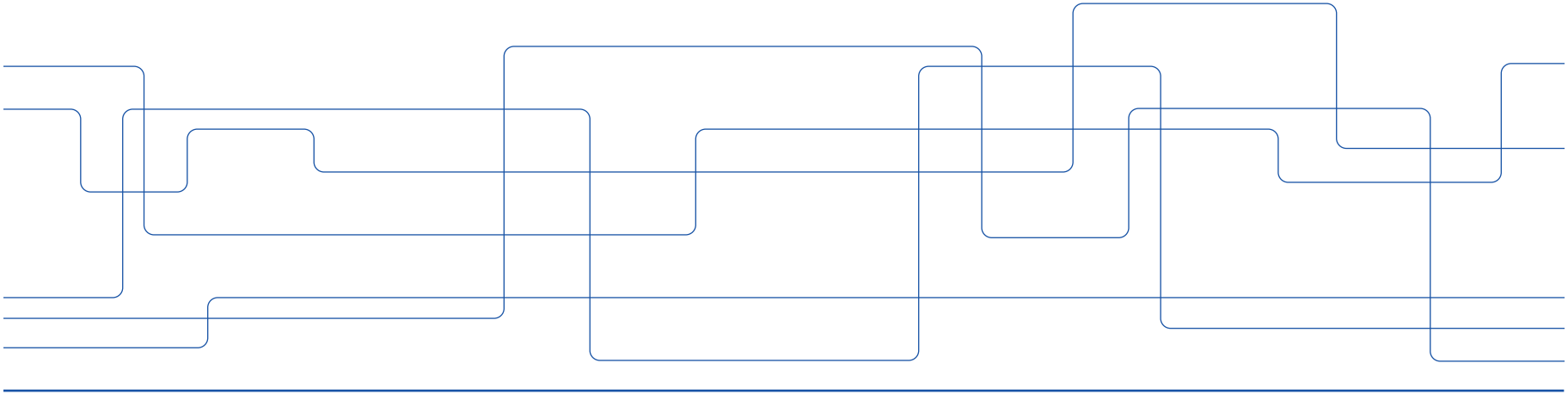
$$p(x_{new}) = \int p(x_{new}, z)\, dz = \int p(x_{new}|z)p(z)\, dz$$

$$= \mathbb{E}_{p(z)}[p(x_{new}|z)]$$

$$\approx \frac{1}{N}\sum_{j=1}^{N} p(x_{new}|z_j), \quad z \sim p(z)$$

$$= \frac{1}{N}\sum_{j=1}^{N} p(x_{new}|\, \mu_\theta(z_j), \sigma_\theta(z_j)), \quad z \sim p(z)$$

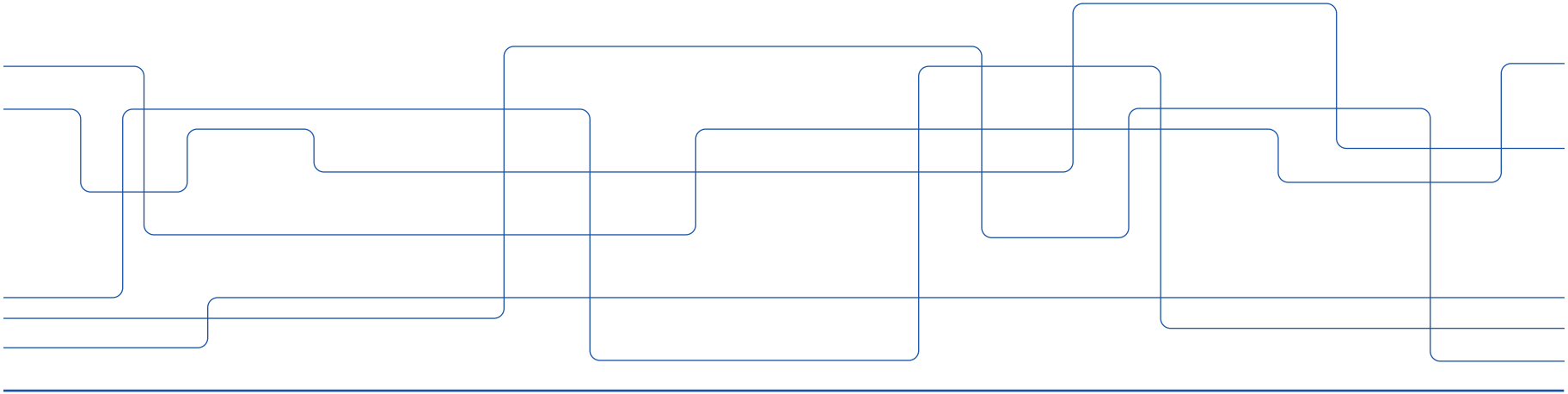
Implementation tricks



KTH ROYAL INSTITUTE
OF TECHNOLOGY

But first….

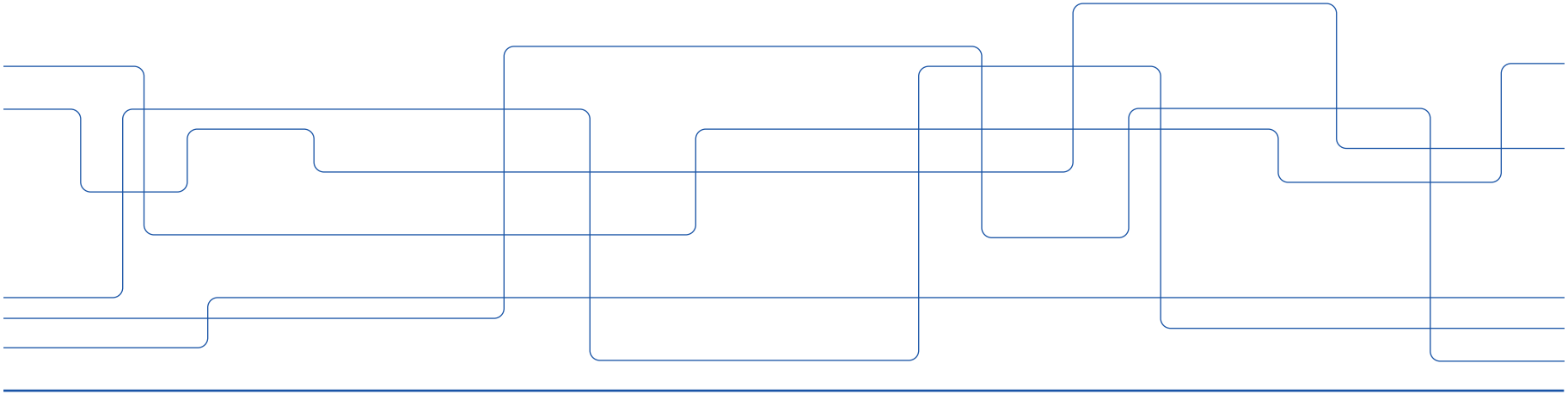
… STAND UP!



Implementation tricks

- Objective function
- Encoder
- Decoder
  - Variance of the decoder
- Design choices
  - Architecture
  - Latent space dimension
  - Modelling decoder variance

Objective function

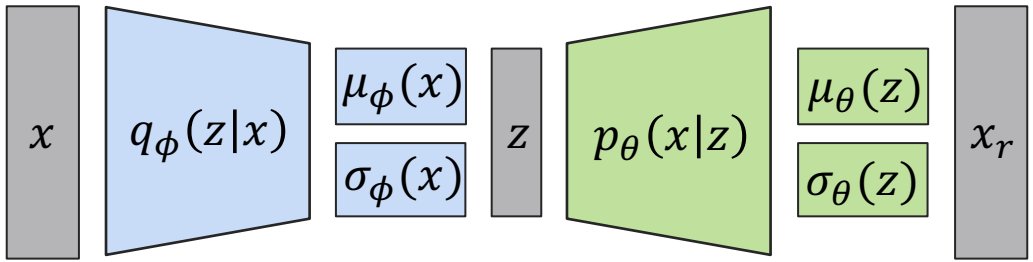


VAE training objective

Optimize

$$\min_{\theta,\phi} D_{KL}\left(q_\phi(z|x_i) \parallel p(z)\right) - \mathbb{E}_{q_\phi(z|x_i)}[\ln p_\theta(x_i|z)]$$

using stochastic gradient descent (SGD).



KL divergence has a closed form

For two Gaussian distributions

$$p(z) = N(0, 1)$$
$$q_\phi(z|x) = N(\mu_\phi(x), \sigma_\phi(x))$$

the KL divergence term has a closed form:

$$D_{KL}\big(N(\mu_\phi(x), \sigma_\phi(x)) \parallel N(0, 1)\big) = -\frac{1}{2} \sum_{j=1}^{latent\ dim} (1 + \ln \sigma_\phi(x)_j^2 - \mu_\phi(x)_j^2 - \sigma_\phi(x)_j^2) \quad (1)$$

(1) See the full derivation Auto-Encoding Variational Bayes, Appendix B

Computing the reconstruction error involves sampling ☹

$$\min_{\theta,\phi} D_{KL}\left(q_\phi(z|x) \parallel p(z)\right) - \mathbb{E}_{q_\phi(z|x)}[\ln p_\theta(x|z)]$$

Problem: we cannot directly backpropagate gradients through the random variable z:

$$\nabla_\phi \, \mathbb{E}_{q_\phi(z|x)} \left[\ln p_\theta(x,z)\right] = ?$$

Reparametrization trick

Express the latent random variable z

$$z \sim q_\phi(z|x) = N\big(\mu_\phi(x), \sigma_\phi(x)\big)$$

as a differentiable transformation of another random variable ϵ

$$z \sim T_\phi(x, \epsilon) = \mu_\phi(x) + \sigma_\phi(x) \odot \epsilon$$
$$\epsilon \sim p(\epsilon) = N(0, 1)$$

where $\odot$ denotes the element-wise product.

Reparametrization trick

Given the change of variable

$$z \sim T_\phi(x, \epsilon) = \mu_\phi(x) + \sigma_\phi(x) \odot \epsilon$$

we can write

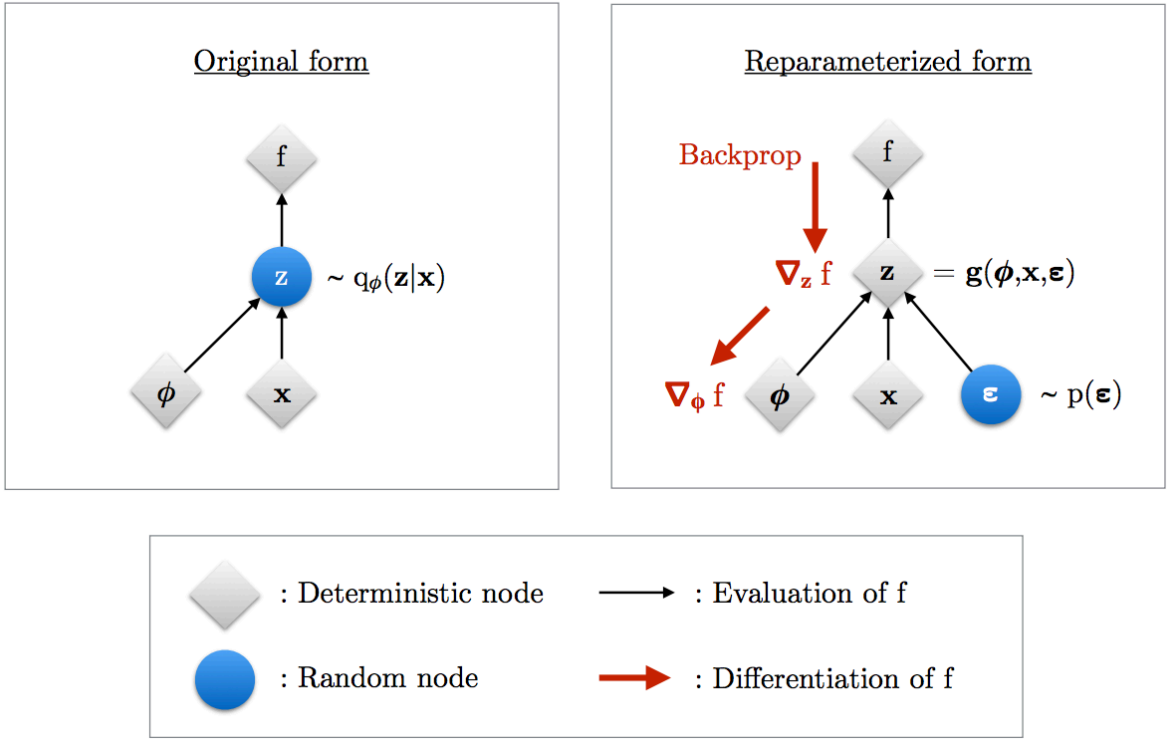
$$\mathbb{E}_{q_\phi(z|x)}[\ln p_\theta(x|z)] = \mathbb{E}_{p(\epsilon)}[\ln p_\theta(x|z)]$$

so that the gradients are commutative:

$$\nabla_\phi \mathbb{E}_{q_\phi(z|x)}[\ln p_\theta(x|z)] = \nabla_\phi \mathbb{E}_{p(\epsilon)}[\ln p_\theta(x|z)]$$

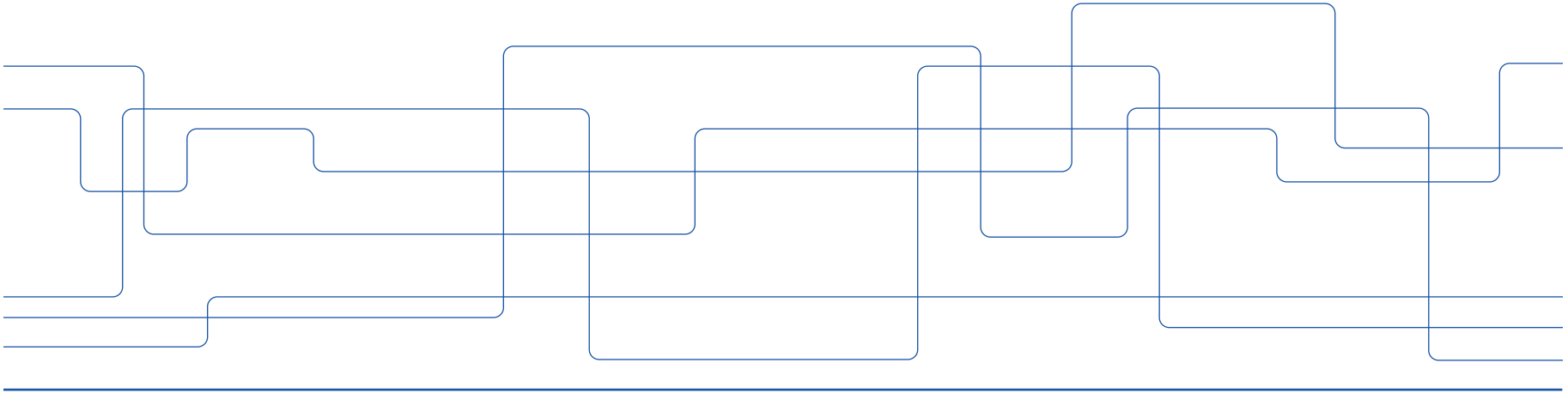
$$= \mathbb{E}_{p(\epsilon)}[\nabla_\phi \ln p_\theta(x|z)]$$

Reparametrization trick



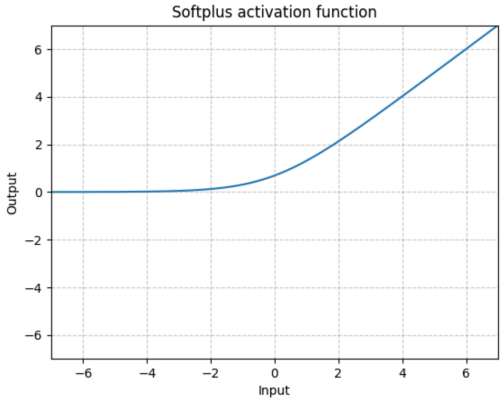
(Image source: https://arxiv.org/pdf/1906.02691.pdf, Section 2.4)

Encoder NN



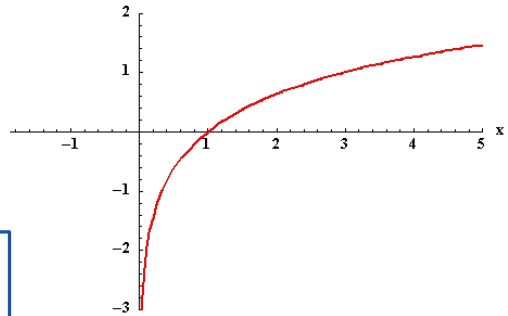
Encoder: model the variance directly

```
class Encoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.q = nn.Sequential(
                        nn.Conv2d(3, 8, 5, stride=2, padding=1),
                        nn.BatchNorm2d(8),
                        nn.ReLU(),
                        nn.Dropout(p=0.2),
                        ...
                        )

        self.enc_mean = nn.Linear(1024, 128)
        self.enc_var = nn.Sequential(
                nn.Linear(1024, 128),
                nn.Softplus()
                )

    def forward(self, x):
        qx = self.q(x)
        mean = self.enc_mean(qx)
        var = self.enc_var(qx) + 1e-5
        return mean, var

    def sample(self, mean, var):
        if self.training:
            std = torch.sqrt(var)
            eps = torch.empty(std.size(), device=self.device).normal_()
            return eps.mul(std).add(mean)
        else:
            return mean
```

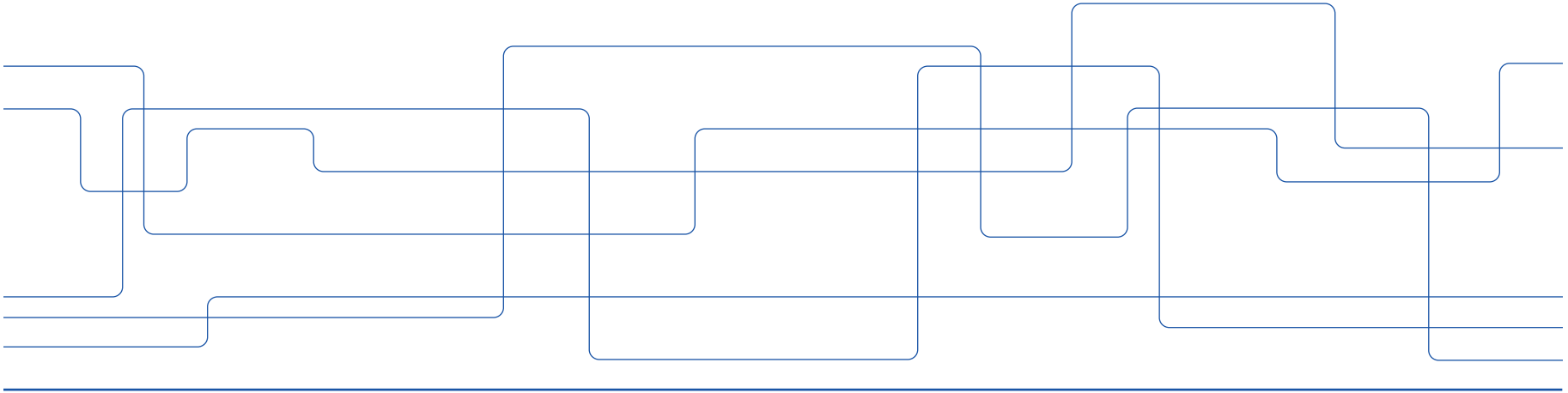


Encoder: model the log variance instead

```
class Encoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.q = nn.Sequential(
                        nn.Conv2d(3, 8, 5, stride=2, padding=1),
                        nn.BatchNorm2d(8),
                        nn.ReLU(),
                        nn.Dropout(p=0.2),
                        ...)

        self.enc_mean = nn.Linear(1024, 128)
        self.enc_logvar = nn.Linear(1024, 128)

    def forward(self, x):
        qx = self.q(x)
        mean = self.enc_mean(qx)
        logvar = self.enc_logvar(qx)
        return mean, logvar

    def sample(self, mean, logvar):
        if self.training:
            std = torch.exp(0.5*logvar)
            eps = torch.empty(std.size(), device=self.device).normal_()
            return eps.mul(std).add(mean)
        else:
            return mean
```



Decoder NN



Decoder: learn the likelihood variance

```
class Decoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.p = nn.Sequential(
                nn.ConvTranspose2d(64, 32, 3, stride=2, padding=1),
                nn.BatchNorm2d(32),
                nn.ReLU(),
                ...
                )

        self.dec_mean = nn.Sequential(
                nn.Conv2d(3, 8, 3, stride=1, padding=1),
                nn.Sigmoid()
                )
        self.dec_logvar = nn.Sequential(
                nn.Conv2d(3, 8, 3, stride=1, padding=1),
                )

    def forward(self, z):
        pz = self.p(z)
        mean = self.dec_mean(pz)
        logvar = self.dec_logvar(pz)
        return mean, logvar

    def recon_loss(self, x, dec_mu, dec_logvar):
        HALF_LOG_TWO_PI = 0.91893
        dec_var = torch.exp(dec_logvar)
        batch_rec = torch.sum(
                HALF_LOG_TWO_PI + 0.5 * dec_logvar + 0.5 * ((x - dec_mu) / dec_var) ** 2,
                dim=(1, 2, 3)) # batch_size
        batch_rec = torch.mean(batch_rec)
```

Depends on your data preprocessing!!!

$$\ln p_\theta(x|z) = \ln \frac{1}{\sqrt{2\pi\sigma_\theta(z)^2}} e^{-\frac{(x-\mu_\theta(z))^2}{2\sigma_\theta(z)^2}}$$

Decoder: fix the likelihood variance to 1

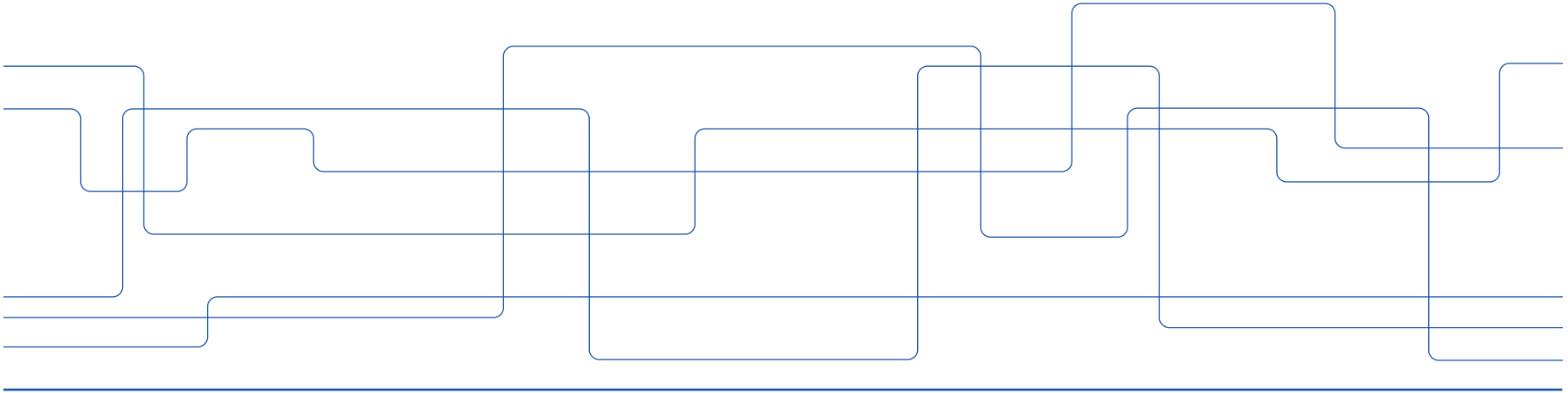
```
class Decoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.p = nn.Sequential(
                nn.ConvTranspose2d(64, 32, 3, stride=2, padding=1),
                nn.BatchNorm2d(32),
                nn.ReLU(),
                ...
                )

        self.dec_mean = nn.Sequential(
                nn.Conv2d(3, 8, 3, stride=1, padding=1),
                nn.Sigmoid()
                )

    def forward(self, z):
        pz = self.p(z)
        mean = self.dec_mean(pz)
        return mean

    def recon_loss(self, x, dec_mean):
        mse_loss = nn.MSELoss(reduction='none')
        batch_rec = torch.sum(mse_loss(x, dec_mean),
                              dim=(1, 2, 3)) # batch_size
        batch_rec = torch.mean(batch_rec)
        return batch_rec
```

$$\ln p_\theta(x|z) = \ln \frac{1}{\sqrt{2\pi}} e^{-\frac{(x-\mu_\theta(z))^2}{2}}$$
$$\approx (x - \mu_\theta(z))^2$$

Since constants don't matter for optimisation ⇒ again the reconstruction effect!

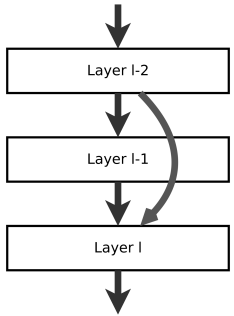
KTH ROYAL INSTITUTE
OF TECHNOLOGY

Design choices
(according to my experience)



Choosing the architecture

- Multi layer perceptron (nn.Linear):
  - simple data potentially without spatial-temporal dependencies (MNIST, simple robot trajectories, …)
- Convolutional layers (nn.Conv, nn.TransposeConv):
  - 2d: images (MNIST, CIFAR-10, SVHN, …)
  - 1d: time-series
- Extra: residual layers
  - Beneficial for more complex datasets (CIFAR-10, ImageNet, …)

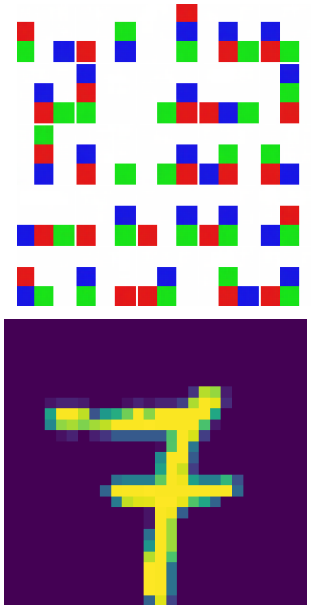


Choosing the latent dimension

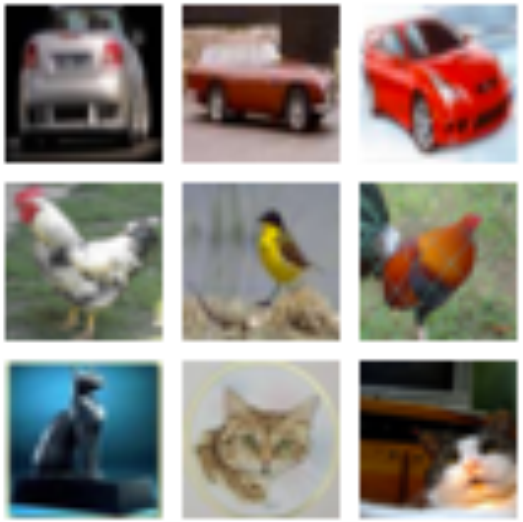
- For simple data (e.g. MNIST):
  - Up to 20, start with say 10.
  - 2-dimensional latent space is practical for visualisation.
- For complex data (e.g. CIFAR-10, ImageNet):
  - More than 60, start with say 64.
  - I used 128 when training ResNet on CIFAR-10.

Choosing the decoder variance

- Fix it to 1 if you are dealing with simple data (e.g. MNIST)



- Learn it if you are dealing with more complex data (e.g. CIFAR-10)



KL annealing schedule

Introduce a hyperparameter β

$$\min_{\theta,\phi} \beta D_{KL}\left(q_\phi(z|x) \parallel p(z)\right) - \mathbb{E}_{q_\phi(z|x)}\left[\ln p_\theta(x|z)\right]$$

And gradually increase it from 0 to n (the higher the n the better disentanglement of the latent space).

Benefits:

- Better reconstructions
- Preventing posterior collapse (i.e. KL term == 0)

Check out β-VAE for more details: https://openreview.net/references/pdf?id=Sy2fzU9gl

What I left out

- Variants of VAEs
  - Disentagled representations: FactorVAE, β-VAE
  - Discrete latent space: VQ-VAE
  - Adversarial: AAE, GAN-VAE hybrids
  - LadderVAEs, Lossy VAE, CAE, TD-VAE, 2StageVAE, …

Summary

- The true goal: **learn the true distribution** of the data $p_*(x)$
- Derivation of ELBO
- **Encoder and decoder NNs return the parameters** of the approximate posterior $q_\phi(z|x)$ and likelihood $p_\theta(x|z)$, respectively.
- Implementation:
  - Reparametrization trick for propagating the gradients
  - Variance of the decoder

Literature

- Papers:
  - Auto-encoding Variational Bayes [Kingma&Welling, 2014]
  - Stochastic Backpropagation and Approximate Inference in Deep Generative Models [Rezende&Wiestra, 2014]
  - β-VAE: Learning Basic Visual Concepts With a Constrained Variational Framework [Higgins et al., 2017]
  - An Introduction to Variational Autoencoders [Kigma&Welling, 2019]
- Blogs:
  - Rui Shu, Density Estimation: Variational Autoencoders
  - Jeremy Jordan, Variational Autoencoders
  - Lilian Weng, From Autoencoder to Beta-VAE

Finally the end!