

Variational Autoencoder (VAE)

The "Vanilla" Autoencoder that we studied

- the Encoder maps example \mathbf{x}^{ip}
- to a single latent representations \mathbf{z}^{ip}
- such that the Decoder tries to "invert" \mathbf{z}^{ip} to recover an approximation of \mathbf{x}^{ip}

The hope was that a slight perturbation in $\backslash z^{ip}$

- would invert into an $\backslash x^{ip}_{synthetic}$
- that was a member of the valid class of examples

thus facilitating the use of the Decoder as a source of synthetic examples.

However, this is what we get

- if we invert latent representations
- that are near neighbors of $\backslash z^{\text{ip}}$

As you see

- even small changes to $\backslash z^{\text{ip}}$ result in corrupted (i.e., not members of the 10 classes of clothing items) examples

original + N (0, 0.0 * σ) original + N (0, 0.5 * σ) original + N (0, 1.0 * σ) original + N (0, 1.5 * σ) original + N (0, 2.0 * σ)



This motivates the *Variational Autoencoder (VAE)*

- an Encoder that maps example \mathbf{x}^{ip} to a *distribution* of latents
- such that samples from the distribution
- can be inverted by the Decoder
- into reasonable approximations of \mathbf{x}^{ip}

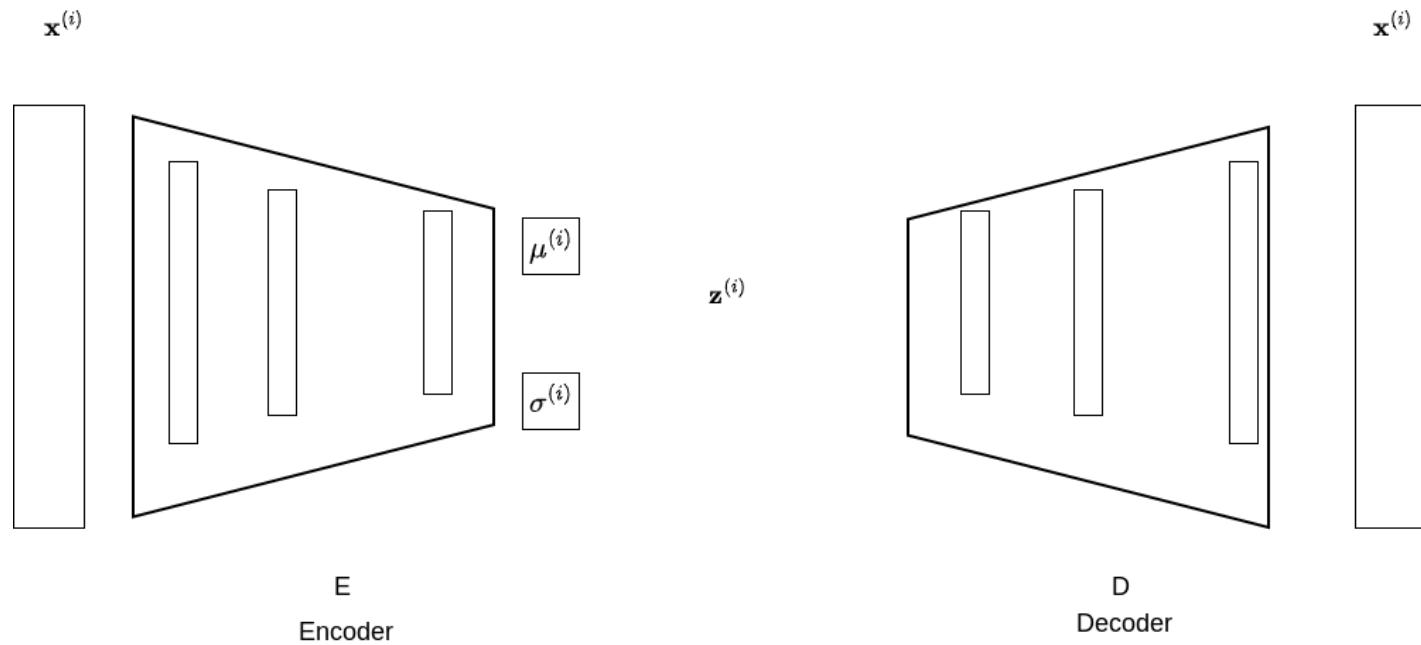
VAE: An engineering perspective

The Encoder part of a VAE, given input \mathbf{x}^{ip}

- Produces the parameters (e.g., $\mu^{\text{ip}}, \sigma^{\text{ip}}$) of a distributional form
- Draws a sample from the distribution as its output \mathbf{z}^{ip}

Thus, the latent representation \mathbf{z} of a given \mathbf{x} is a probability distribution
 $\mathbf{qrs}(\mathbf{z} | \mathbf{x})$

Variational Autoencoder (VAE)



Note

μ^{ip} and σ^{ip} are

- vectors
- computed values (and hence, functions of \mathbf{x}^{ip}) and **not** parameters
- so training learns a *function* from \mathbf{x}^{ip} to μ^{ip} and σ^{ip}

Although the Encoder

- creates a distribution of latents $\mathbf{q}(\mathbf{z} | \mathbf{x})$

it will be necessary to constrain this distribution

- to be an approximation of the "true" distribution $\mathbf{p}(\mathbf{z} | \mathbf{x})$ of latents

Thus, when training the VAE, the Loss Function will have two terms

- the Reconstruction Loss
 - enforcing that the reconstruction $\hat{\mathbf{x}}^{ip}$ is close to the input \mathbf{x}^{ip}
- the Divergence Constraint
 - ensuring that the constructed distribution $\mathbf{q}(\mathbf{z} | \mathbf{x})$ is close to true distribution $\mathbf{p}(\mathbf{z} | \mathbf{x})$
$$\text{KL}(\mathbf{q}(\mathbf{z} | \mathbf{x}) || \mathbf{p}(\mathbf{z} | \mathbf{x}))$$

This is not just straightforward engineering.

In fact: the architecture of the VAE was obtained from the math rather than vice-versa !

We provide a brief overview of the mathematics.

The interested reader is referred to a highly recommended [VAE tutorial](#)
[\(https://arxiv.org/pdf/1606.05908.pdf\)](https://arxiv.org/pdf/1606.05908.pdf) for a detailed presentation.

Details

Notation summary

term	dimension	meaning
$\backslash x$	n	Random variable for Input
$\tilde{\backslash x}$	n	Output: reconstructed $\backslash x$
$\backslash z$	$n' << n$	Random variable for Latent representation
E	$\mathbb{R}^n \rightarrow \mathbb{R}^{n'}$	Encoder
		$E(\backslash x) = \backslash z$
D	$\mathbb{R}^{n'} \rightarrow \mathbb{R}^n$	Decoder
		$\tilde{\backslash x} = D(\backslash z)$
		$\tilde{\backslash x} = D(E(\backslash x))$
		$\tilde{\backslash x} \approx \backslash x$
$\backslash pr\backslash x$	prob. distribution	prior distribution of Inputs intractable. Only have empirical.
$\backslash qr\backslash z$	prob. distribution	prior distribution of Latents
$\backslash qr\backslash z \backslash x$	prob. distribution	posterior marginal distribution of Latents given Input intractable
$\backslash qrs$ $\backslash z \backslash x\Phi$	Neural Network	NN to approximate $\backslash qr\backslash z \backslash x$
		Encoder
$\backslash prs$ $\backslash x \backslash z\Theta$	Neural Network	NN to approximate $\backslash pr\backslash x \backslash z$
		Decoder

VAE: Mathematical perspective

TL;DR

- The VAE has a very interesting **two part** Loss Function
 - Reconstruction Loss, as in the Vanilla AE
 - Divergence Loss
- The Reconstruction Loss is not sufficient
 - Issues of intractability arise
 - The Divergence Loss skirts intractability
 - By constraining the Encoder to produce a tractable distribution

Distributions of inputs, latents, outputs

Let's pretend that we don't already know the architecture of a VAE

- that the latent \mathbf{z}^{ip} is generated as a probability function of μ^{ip} and σ^{ip} given input \mathbf{x}^{ip} .

Instead let

- $\mathbf{x} \sim \text{pr}(\mathbf{x})$ denote
 - an example \mathbf{x} drawn from the distribution of examples $\text{pr}(\mathbf{x})$
- $\mathbf{z} \sim \text{qr}(\mathbf{z})$ denote
 - a latent drawn from the true distribution of latents $\text{qr}(\mathbf{z})$

We marginalize these distributions to

- show the dependence between example $\textcolor{red}{x}^{\text{ip}}$ and its latent representations

Let

- $\textcolor{red}{q}_{\theta}(z|x)$ denote the *conditional distribution* of Latent, given an Input
 - This is the distribution that is produced by the Encoder
- $\textcolor{red}{p}_{\phi}(x|z)$: the *conditional distribution* of Input, given a Latent
 - This is the distribution that is produced by the Decoder

We can go from the marginal back to the joint distribution by integration

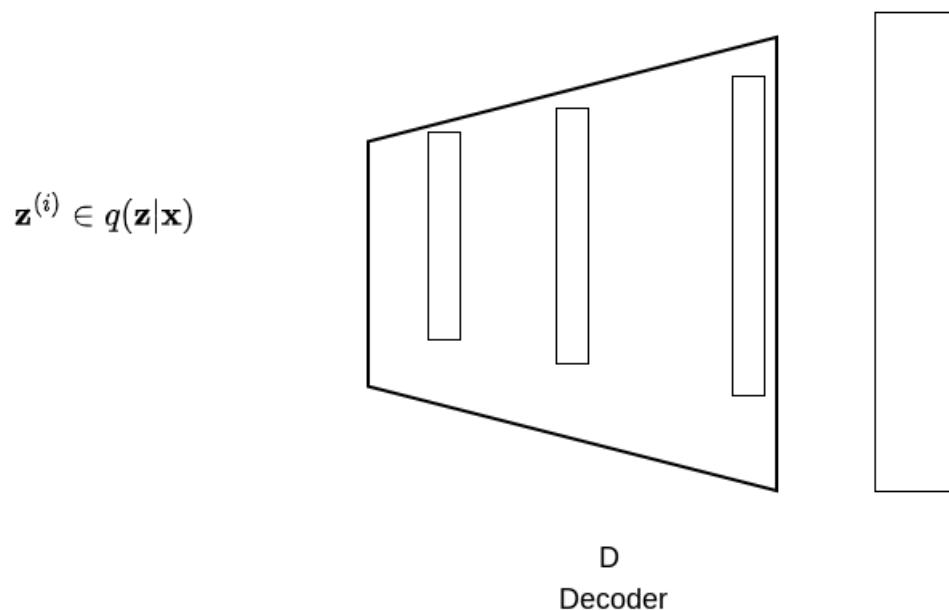
- $\backslash \text{pr} \backslash x | \backslash z$
 - over all $\backslash z$
- $$\in \backslash \text{qr} \backslash z$$

$$\backslash \text{pr} \backslash x = \int_{\backslash z \in \backslash \text{qr} \backslash z} \backslash \text{pr} \backslash x | \backslash z \backslash \text{qr} \backslash z$$

Operationally, this would result in a Decoder architecture

VAE derivation: 1

$$\tilde{\mathbf{x}}^{(i)} = \int_{\mathbf{z}} p_{\theta}(\mathbf{x}|\mathbf{z})q(\mathbf{z})$$



Some obvious concerns about the integral

- It may be very expensive to draw many samples of \mathbf{z} from $\mathbf{qr}(\mathbf{z})$ for each training example \mathbf{x}^{ip}
- There are many random choices of \mathbf{z} from $\mathbf{qr}(\mathbf{z})$ which can't reconstruct \mathbf{x}^{ip}
 - i.e., $\mathbf{pr}_{\mathbf{z}} \mathbf{x}^{ip} | \mathbf{z}' \Theta = 0$ for many \mathbf{z}'

We can improve our sampling by considering only those choices of \mathbf{z} that could generate \mathbf{x} and re-write the objective as

$$\mathbf{pr}_{\mathbf{x}} = \int_{\mathbf{z} \in \mathbf{qr}(\mathbf{z}) \setminus \mathbf{x}} \mathbf{pr}_{\mathbf{z}} \mathbf{x} | \mathbf{z} \mathbf{qr}(\mathbf{z})$$

Problem: Some distributions aren't known !

But there's a problem !

- The distribution $\Pr(\mathbf{x})$ of inputs \mathbf{x} is *intractable*
- We only have *empirical samples* from the distribution
 - i.e., the training examples
 - we don't know the functional form of the parent distribution

For instance: what is the distribution of images of Dogs ?

Similarly

- $\Pr(\mathbf{z}|\mathbf{x})$ is intractable
 - Recall: this is the intended output of the Encoder

Operationalizing the mathematically-derived architecture

We will side-step the intractability issues by defining Neural Networks to learn *approximations*

- $\text{qrs} \mid z \mid x \Phi$
 - a Neural Network (the Encoder)
 - with parameters Φ
 - that will use training examples
- to learn an approximation of true $\text{qr} \mid z \mid x$

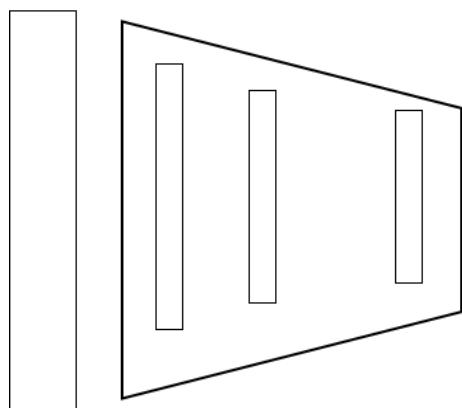
Similarly

- $\text{pr}_{\mathbf{x}|\mathbf{z}}(\theta)$
 - a Neural Network (the Decoder)
 - with parameters Θ
 - that will use training examples
- to learn an approximation of true $\text{pr}_{\mathbf{x}|\mathbf{z}}$

The resulting architecture:

VAE derivation: 2

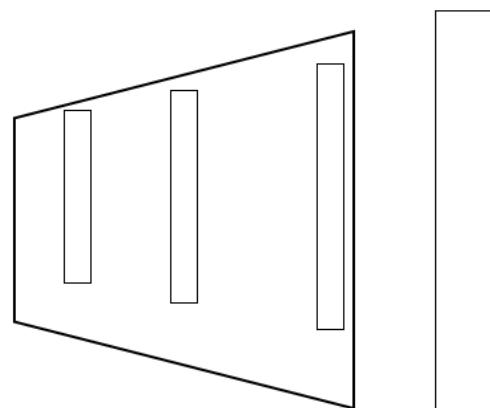
$$\mathbf{x}^{(i)}$$



$$\mathbf{z}^{(i)} = q_{\phi}(\mathbf{z}|\mathbf{x})$$

$$\text{KL}(q_{\phi}(\mathbf{z}|\mathbf{x}) \parallel q(\mathbf{z}|\mathbf{x}))$$

$$\tilde{\mathbf{x}}^{(i)} = \int_{\mathbf{z} \in q(\mathbf{z}|\mathbf{x})} p_{\theta}(\mathbf{x}|\mathbf{z})q(\mathbf{z})$$



E
Encoder

D
Decoder

Constraining the Encoder

In order to ensure that the Encoder's

- approximate distribution $\text{qrs}(\mathbf{z} | \mathbf{x})$
- is close to true distribution $\text{qr}(\mathbf{z} | \mathbf{x})$

we add the Divergence Constraint to the Loss Function

$$\text{KL}(\text{qrs}(\mathbf{z} | \mathbf{x}) || \text{qr}(\mathbf{z} | \mathbf{x}))$$

Note

The KL function is *not symmetric*

$$\text{KL}(\mathbf{q} \mathbf{r} \mathbf{s} \| \mathbf{z} \| \mathbf{x} \Phi) \neq \text{KL}(\mathbf{q} \mathbf{r} \| \mathbf{z} \| \mathbf{x} \| \mathbf{q} \mathbf{r} \mathbf{s} \| \mathbf{x} \Phi)$$

So the constraint, as written

- says that each approximation
- is close to the true value

It *does not say*

- that each true value
- has a close approximation

The Loss Function for training

We can train the Encoder/Decoder pair with the objective that the reconstructed $\tilde{\mathbf{x}}^{\text{ip}}$ approximates the true \mathbf{x}^{ip} from the training set, across all examples i .

One way of stating this is as a Maximum Likelihood:

- Solve for the weights Φ, Θ
- That maximize the (log) Likelihood of the set of reconstructions $\tilde{\mathbf{X}}$ reproducing the training set \mathbf{X}

Since our practice is to minimize Loss (rather than maximize an objective function) we write our loss as (negative of log) likelihood

$$\text{loss} = -\log(\text{pr } \mathbf{X})$$

Minimizing loss is equivalent to maximizing likelihood.

Adding the KL divergence constraint to our Likelihood objective gives the loss function

$$\begin{aligned}\text{\textcolor{red}{loss}} &= -\log(\text{\textcolor{red}{p}}(\mathbf{x}|\Theta)) + \text{\textcolor{red}{KL}}(\text{\textcolor{red}{q}}(\mathbf{z}|\mathbf{x}, \Phi) || \text{\textcolor{red}{p}}(\mathbf{z}|\mathbf{x})) \\ &= \text{\textcolor{red}{loss}}_R + \text{\textcolor{red}{loss}}_D\end{aligned}$$

which now has two objectives

- Reconstruction loss $\text{\textcolor{red}{loss}}_R$: maximize the likelihood (by minimizing the negative likelihood)
- Divergence constraint $\text{\textcolor{red}{loss}}_D$: $\text{\textcolor{red}{q}}(\mathbf{z}|\mathbf{x}, \Phi)$ must be close to $\text{\textcolor{red}{p}}(\mathbf{z}|\mathbf{x})$

$$\text{\textcolor{red}{loss}}_R = -\log(p_\theta(\mathbf{x}))$$

$$\text{\textcolor{red}{loss}}_D = \text{\textcolor{red}{KL}}(\text{\textcolor{red}{q}}(\mathbf{z}|\mathbf{x}, \Phi) || \text{\textcolor{red}{p}}(\mathbf{z}|\mathbf{x}))$$

Problem: \loss contains an intractable term

Looking closely at the Divergence Constraint

$$\text{\loss}_D = \text{KL}(\text{\qrs}\text{\z}|\text{\x}\Phi \parallel \text{\qr}\text{\z}|\text{\x})$$

we note the presence of the term

$$\text{\qr}\text{\z}|\text{\x}$$

which we identified as intractable.

Hence

- the Divergence Constraint
- as written
- cannot be operationalized

We will show (in the next section: lots of algebra !) that the loss can be re-written as

$$\text{loss} = -\mathbb{E}_{z \sim q_{rs}(z|x)} (\log(p_{rs}(x|z|\Theta)) + \text{KL}(q_{rs}(z|x|\Phi) || q_r(z))$$

This is *almost* identical to our original express for `loss` except

- the intractable KL term

$$\text{KL}(q_{rs}(z|x|\Phi) || q_r(z|x))$$

becomes

$$\text{KL}(q_{rs}(z|x|\Phi) || q_r(z))$$

This is still contains intractable

$$\backslash qr \backslash z$$

We can make this tractable

- by making *an assumption* as to the functional form of $\backslash qr \backslash z$
 - e.g., Normal

Note

it is easier to make an assumption about

- *unconditional distribution* $\backslash qr \backslash z$

rather than

- *conditional distribution* $\backslash qr \backslash z | \backslash x$

since we don't have to know the joint relationship between example $\backslash x$ and its latent $\backslash z$

Choosing $\mathbf{q}_{\mathbf{r}}(\mathbf{z})$

So what distribution should we use for the prior $\mathbf{q}_{\mathbf{r}}(\mathbf{z})$?

- It should be differentiable, since we use Gradient Descent for optimization
- It should be tractable with a closed form (such as a normal)

A convenient (but not necessary) choice for $\mathbf{q}_{\mathbf{r}}(\mathbf{z})$ is normal

- If we choose $\mathbf{q}_{\mathbf{r}}(\mathbf{z})$ as normal, we can require $q_{\phi}(\mathbf{z} | \mathbf{x})$ to be normal too
- The KL divergence between two normals is an easy to compute function of their means and standard deviations.
 - the "easy to compute" part is the "convenience"
 - See [VAE tutorial \(<https://arxiv.org/pdf/1606.05908.pdf>\)](https://arxiv.org/pdf/1606.05908.pdf) Section 2.2

Re-parameterization trick

There is still one impediment to training.

It involves the random choice of $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ in

$$\text{loss}_R = \mathbb{E}_{z \sim \mathcal{N}(\mathbf{0}, \mathbf{I})} (\log(\mathbf{p}(x|z\Theta)))$$

This is not a problem in the forward pass.

But in the backward pass we need to compute

$$\frac{\partial \text{loss}_R}{\partial \Theta}$$

How do we back propagate through a random choice ?

The "reparameterization trick" redefines the random choice $\backslash z$ as

$$\begin{aligned} \mathbf{z} &= \mu_\theta(\backslash x) + \sigma_\theta(\backslash x) * \epsilon \\ \epsilon &\sim N(0, 1) \end{aligned}$$

That is

- Once we have defined $\backslash q \backslash z$ to be a Normal distribution
- We can re-write an element of the distribution
 - as the distribution's mean plus a random standardized Normal ϵ times the distribution's standard deviation

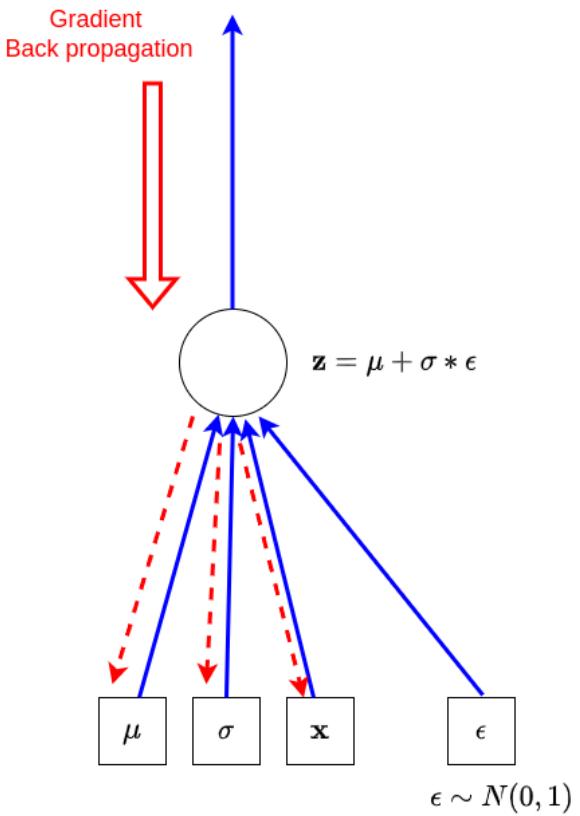
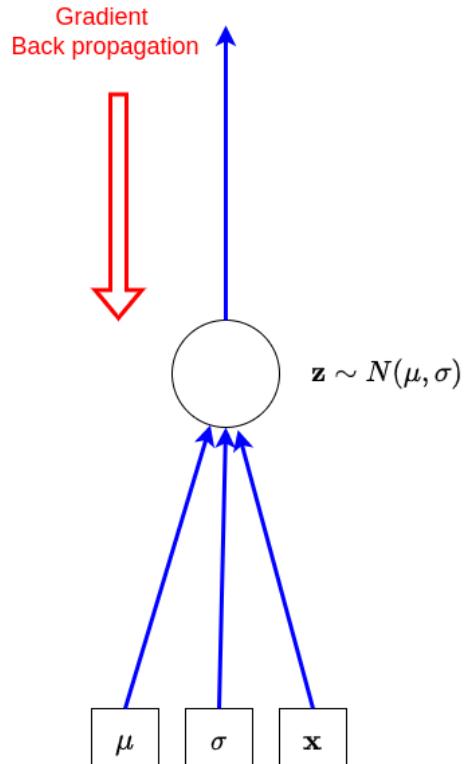
In this formulation, the random variable ϵ appears in a product term

- We can differentiate the product with respect to Θ
- ϵ can be treated as a constant in $\frac{\partial \epsilon}{\partial \Theta}$

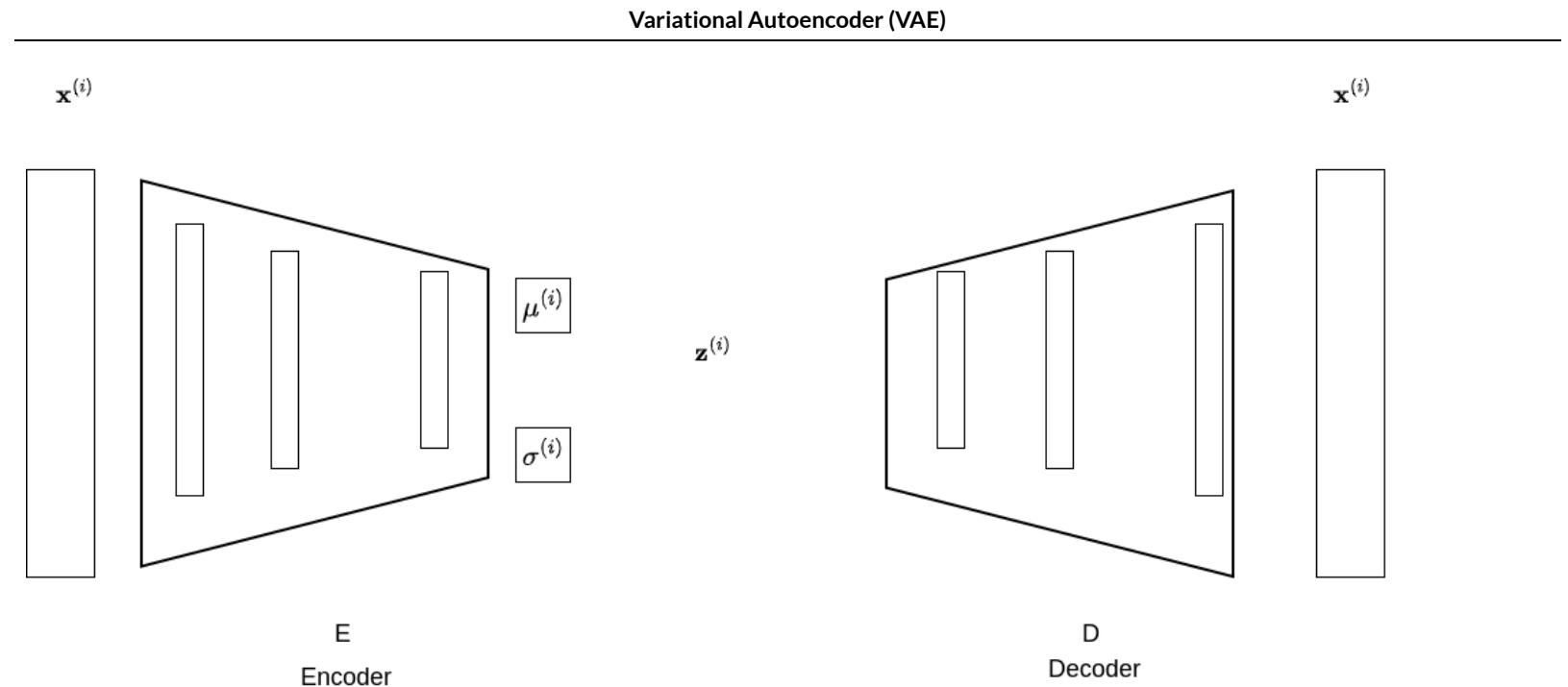
The Encoder design is now to produce (trainable parameters) $\mu_\Theta, \sigma_\Theta$

- And $\backslash z$ indirectly

Reparameterization trick



This gets us to the final picture of the VAE:



To train a VAE:

- pass input \mathbf{x}^{ip} through the Encoder, producing $\mu^{\text{ip}}, \sigma^{\text{ip}}$
 - use $\mu^{\text{ip}}, \sigma^{\text{ip}}$ to sample a latent representation \mathbf{z}^{ip} from the distribution
- pass the sampled \mathbf{z}^{ip} through the decoder, producing $D(\mathbf{z}^{\text{ip}})$
- measure the reconstruction error $\mathbf{x}^{\text{ip}} - D(\mathbf{z}^{\text{ip}})$, just as in a plain AE
- back propagate the error, updating all weights and μ, σ

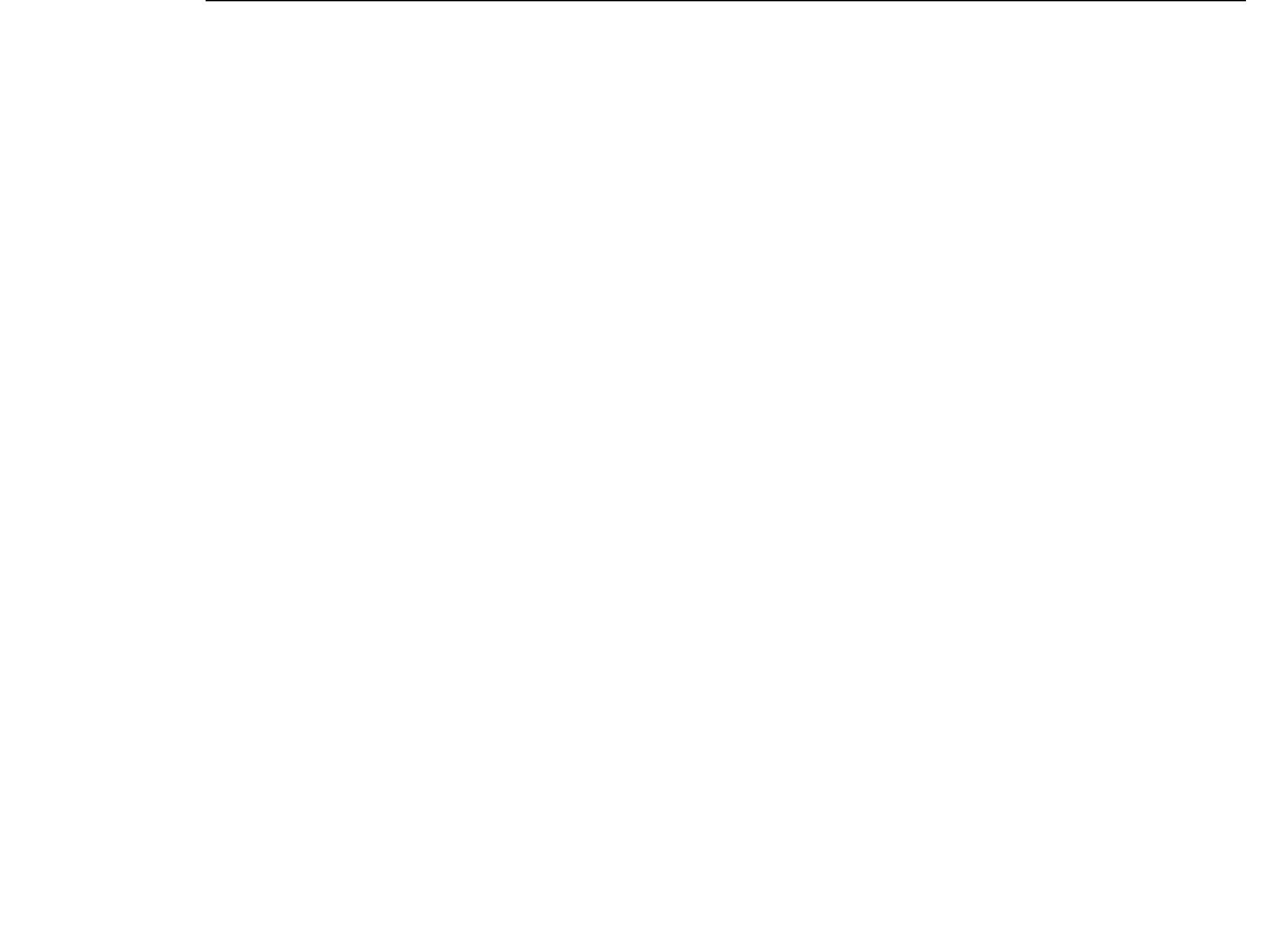
Each time that we encounter the same training example (e.g., in different epochs), we select another random element from the distribution.

Thus the VAE learns to represent the same example from multiple latents.

Using a VAE to produce synthetic examples

We can examine how the latent representations produced by the VAE form clusters:

MNIST examples: clustering of latent $\backslash z$



In comparing the clusterings produced by the VAE against our previous example of a plain Autoencoder be aware

- The two models are displaying results on different data: MNIST digits versus Fashion MNIST
- The architecture of the Encoder and Decoder are different in the two models
 - The plain Autoencoder used extremely simple architectures
 - Could the more complex architecture of the VAE Encoder/Decoder be the cause of tighter clustering ?

Certainly room for experimentation !

Can we interpret the elements of the latent vector ?

The latent vector can be thought of as a

- reduced dimensionality
- representation of \mathbf{x}^{ip}

where each element of \mathbf{z}^{ip} is in

- a new basis space
- of "concepts"
- rather than syntactic features

This is similar to the ideas we expressed

- when trying to interpret Principal Components

We will attempt an interpretation by

- examining *the neighborhood* around the latent \mathbf{z}^{ip} of example \mathbf{x}^{ip}
- by perturbing each element of the latent vector

In our illustration

- latent vectors \mathbf{z} that are implemented as a vectors of length 2
 - as are μ and σ

We vary the first and second element of a vector

- and obtain the Decoder output on each combination of first and second element

So the results are *possible* outputs of the Decoder

- based on latent vectors $\backslash z$
- that are not necessarily the output of the Encoder on any actual input $\backslash x$

Synthetic MNIST examples from a VAE:
Code implements latent $\backslash z$ as vector with length 2
vary the 2 elements of the latent vector

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Can we interpret $\backslash z_0$ and $\backslash z_1$ by inspecting how the Decoder output changes by varying each ?

- $\backslash z_0$: controls slant?
 - See the bottom row of 0's
- $\backslash z_1$: controls "verticality"?
 - See right-most column

Conditional VAE

The VAE would seem to offer a solution to the problem of creating synthetic data.

But there is a problem

- an *unlabeled* example is created
- we have no way of knowing the label
- nor do we have a way of *controlling* the output so as to be an example with a specified label

We can modify the VAE so as to *conditionally* generate an example with a specified label.

- [Conditional VAE \(Cond_VAE_Generative.ipynb\)](#)

Advanced details

Obtain loss by rewriting

$$\text{KL}(\text{qrs}\text{z}|\text{x}\Phi \parallel \text{qr}\text{z}|\text{x})$$

Let's derive a simpler expression for loss by manipulating

$$\text{KL}(\text{qrs}\text{z}|\text{x}\Phi \parallel \text{qr}\text{z}|\text{x}):$$

The key step:

- Using Bayes Theorem to re-write

$$\log(\textcolor{red}{\mathbf{q}(\mathbf{r}|\mathbf{z})})$$

as

$$\log(\textcolor{red}{\mathbf{p}(\mathbf{r}|\mathbf{x})}) + \log(\textcolor{red}{\mathbf{q}(\mathbf{z})}) - \log(\textcolor{red}{\mathbf{p}(\mathbf{x})})$$

- This allows us do away with intractable conditional probability $\textcolor{red}{\mathbf{q}(\mathbf{r}|\mathbf{z})}$
- In favor of unconditional probability $\textcolor{red}{\mathbf{q}(\mathbf{z})}$

The LHS cannot be optimized via SGD (recall the tractability issue with $\textcolor{red}{\mathbf{q}(\mathbf{r}|\mathbf{z})}$).

But the RHS can be made tractable by

- Approximating $\textcolor{red}{\mathbf{p}(\mathbf{r}|\mathbf{x})}$ with $\textcolor{red}{\mathbf{p}(\mathbf{r}|\mathbf{x}, \Theta)}$
- Assuming that the distributions $\textcolor{red}{\mathbf{q}(\mathbf{z})}$ (and $\textcolor{red}{\mathbf{p}(\mathbf{x})}$) as Normal

Code

Here is a diagram of the code for the Encoder side of the VAE.

Note that the sub-networks that compute

- μ (box labeled z_mean)
- σ (box labeled z_log_var)

have outputs of length 2 so latent vector $\backslash z$ has two elements.

We will use latents of length 2 in an example below showing some possible outputs of the Decoder.

Notice that

- the inputs to the sub-network computing μ and σ
- are identical

VAE: Components (Encoder)

Encoder

input_10	input:	[None 28 28 1] [None 28 28 1]
----------	--------	---------------------------------

VAE code highlights

Here (<https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/vae.ipynb#scrollTo=HPRXyb06O2y8>) is an implementation of the VAE.

Highlights

- `encoder` samples from the distribution

```
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
z = Sampling()([z_mean, z_log_var])
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var, z], name="encoder")
```

- custom `train_step`
 - computes two-part loss: Reconstruction Loss + KL Loss

```
def train_step(self, data):  
  
    with tf.GradientTape() as tape:  
        z_mean, z_log_var, z = self.encoder(data)  
        reconstruction = self.decoder(z)  
        reconstruction_loss = tf.reduce_mean(  
            tf.reduce_sum(  
                keras.losses.binary_crossentropy(data, reconstruction),  
                axis=(1, 2),  
            )  
        )  
        kl_loss = -0.5 * (1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var))  
        kl_loss = tf.reduce_mean(tf.reduce_sum(kl_loss, axis=1))  
        total_loss = reconstruction_loss + kl_loss
```

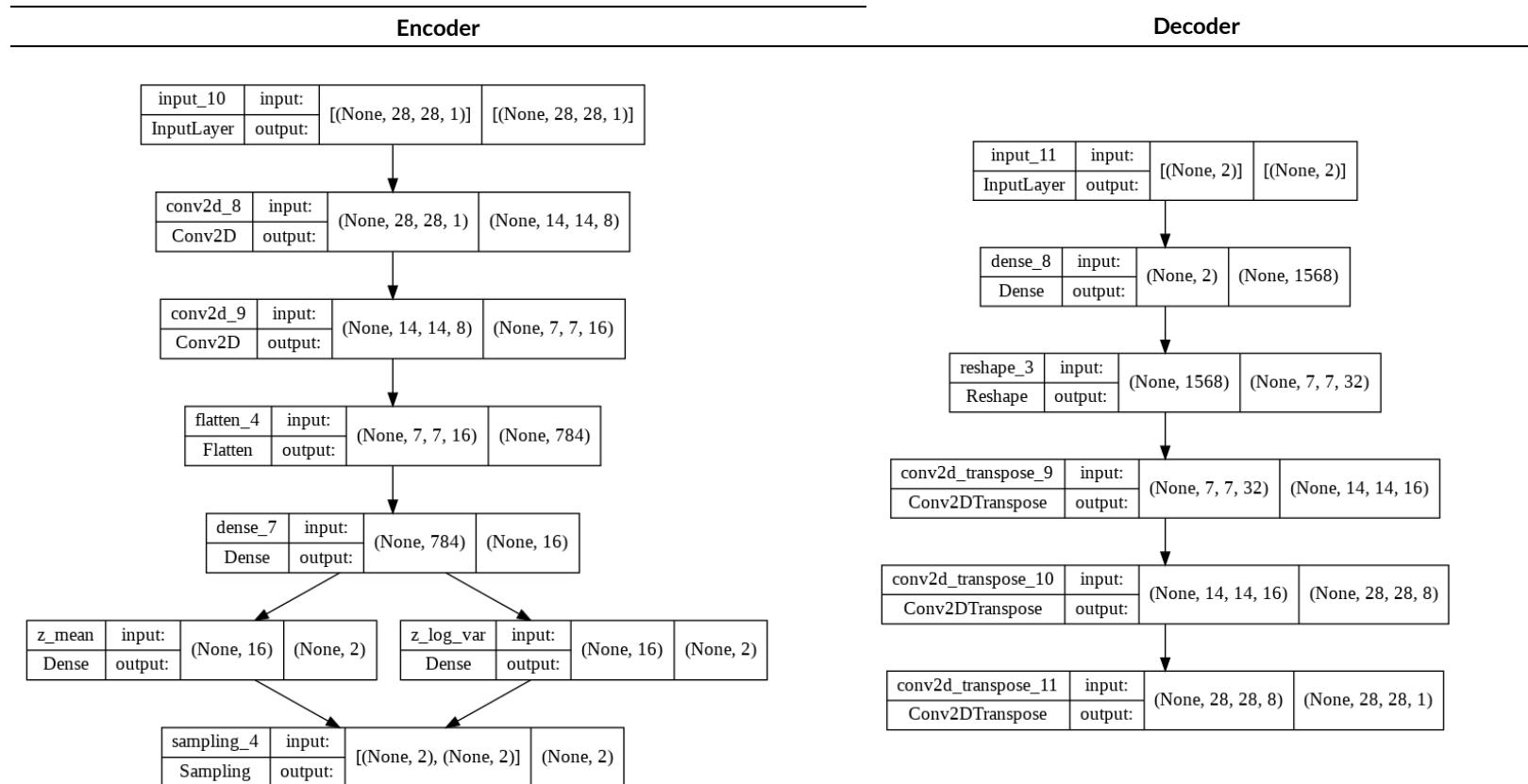
Code: VAE, CVAE

We can learn much more about the properties and use of a VAE through examples

A secondary objective is to look at the code which involves some advanced features of Keras.

The architecture of the VAE will be more complex compared to the vanilla Autoencoder.

VAE: Components



Encoder

- Note the two branches to nodes `z_mean` and `z_log_var`
 - The output of their common parent is used to generate two separate values (μ and σ)
 - μ and σ are both vectors of length 2
 - Thus, the sampled $\backslash z$ is also of length 2
 - In our grid illustration of generating synthetic examples, we vary each of the 2 components of $\backslash z$
 - Latent is *much* shorter than in the plain VAE
 - does the random nature of sampling facilitate shorter representation ?

Let's explore the code through this [notebook \(VAE_code.ipynb\)](#).

- VAE code
- CVAE code

Loss function: discussion

Let's examine the role of loss_R and loss_D in the loss function loss .

- What would happen if we dropped loss_D ?
 - We would wind up with a deterministic \mathbf{z} and collapse to a vanilla VAE
- What would happen if we dropped loss_R ?
 - the encoding approximation $\mathbf{q} \approx \mathbf{p} \mid \mathbf{x}$ would be close to the empirical $\mathbf{p} \approx \mathbf{q} \mid \mathbf{x}$ in distribution
 - but two variables with the same distribution are not necessarily the same?
 - e.g., get a distribution p by flipping a coin
 - let distribution q be a relabelling of p by changing Heads to Tails and vice-versa
 - p and q are equal in distribution but clearly different !

ELBo (Evidence-based Lower Bound)

By re-writing the Loss, we removed the intractable term $\log p(x|z)$

It turns out that even this may not be necessary.

For the truly interested reader:

- The derivation uses a method known as *Variational Inference*. See this [blog](https://mbernste.github.io/posts/variational_inference/) (https://mbernste.github.io/posts/variational_inference/) for a summary.
- One can show that loss loss is equal to -1 times the *ELBo* (Evidence Based Lower Bound)

So if one knows how to maximize the [ELBo](https://mbernste.github.io/posts/elbo/) (<https://mbernste.github.io/posts/elbo/>), one can minimize the loss.

In [2]: `print("Done")`

Done

