Variational Autoencoder (VAE)

Like the plain Autoencoder that we have already encountered a *Variational Autoencoder (VAE)* is comprised of an Encoder and a Decoder

In both cases

- the Encoder produces a latent representation $\mathbf{z}^{(ip)}$ of its input $\mathbf{x}^{(ip)}$
- the Decoder attempts to reconstruct $\mathbf{x}^{(ip)}$ from $\mathbf{z}^{(ip)}$

However, the behavior of the Decoder is undefined when presented with a latent $\mathbf{z}$ that did not arise from a training example.

- We can only hope that the output is reasonable

As we saw, this has implications as to our ability to use the AE as a means of generating synthetic examples.

The Decoder part of a VAE is identical to that of the plain Autoencoder.

But the Encoder part of a VAE is different in an important way. Given input $\mathbf{x}$

- It creates a *distribution* for the the latent representation $\mathbf{z}$
- Rather than creating a unique $\mathbf{z}$

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

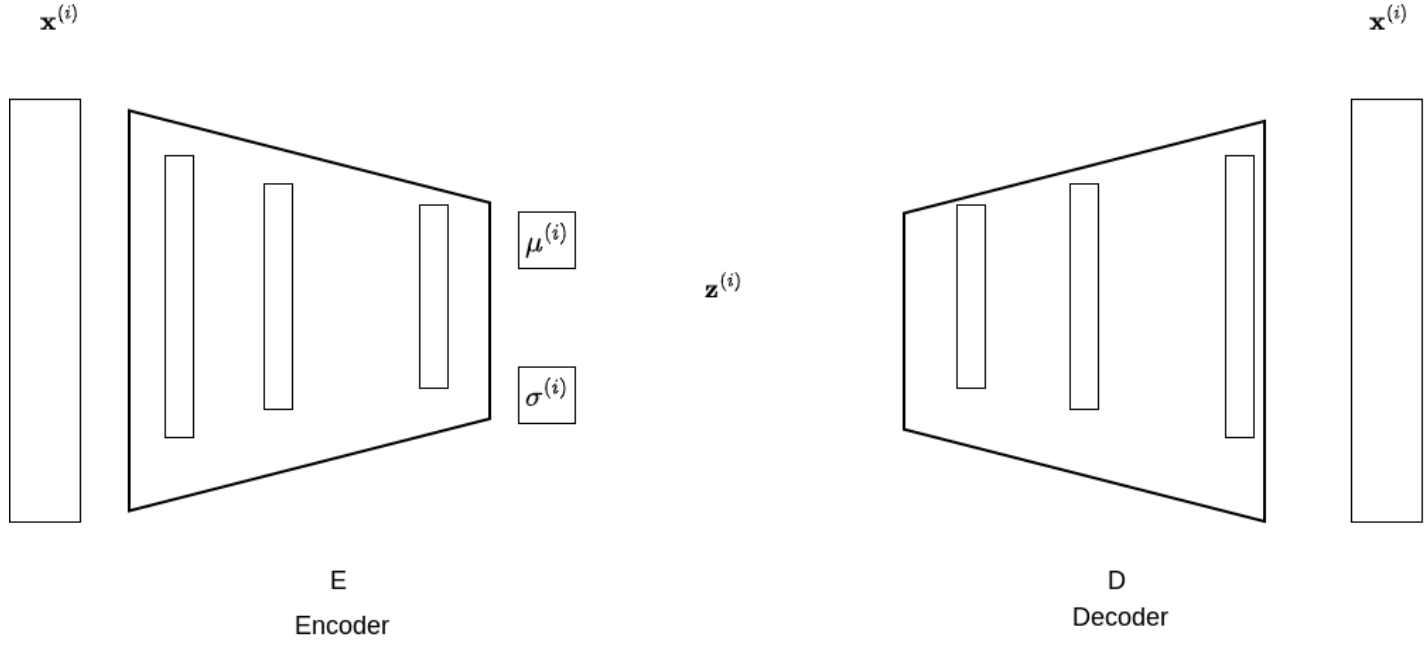
- Produces the parameters (e.g., $\mu^{(ip)}$, $\sigma^{(ip)}$) of a distributional form
- Draws a sample from the distribution as its output $\mathbf{z}^{(ip)}$

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

This may address one of the concerns we raised with using a plain Autoencoder in a generative manner

- that a slight perturbation of the latent $\mathbf{z}^{(ip)}$ obtained from input $\mathbf{x}^{(ip)}$
- might have the Decoder produce $\tilde{\mathbf{x}}^{(ip)}$ that is not similar to $\mathbf{x}^{(ip)}$

Variational Autoencoder (VAE)



Note

$\mu^{\backslash \text{ip}}$ and $\sigma^{\backslash \text{ip}}$ are

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

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) for a detailed presentation.

Details

Notation summary

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

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

- that the latent $\z^{\ip}$ is generated as a probability function of $\mu^{\ip}$ and $\sigma^{\ip}$ given input $\x^{\ip}$.

Instead let

- $\x$ denote a random variable representing an Input
    - the random variable is from the (unknown) distribution $\pr\x$
- $\z$ denote a random variable representing a Latent
    - the random variable is from the (unknown) distribution $\qr\z$

Because $\x$ and $\z$ are related, there is also

- a **joint distribution** of $\pr\x, \z$

from which we can define the marginal distributions

- $\qr\z|\x$: the marginal distribution of Latent, given an Input
- $\pr\x|\z$: the marginal distribution of Input, given a Latent

But there's a problem !

- The distribution $\pr\x$ of inputs $\x$ is *intractable*

We don't know what (or if it is even defined) the functional form of the distribution of $\x$ is.

- e.g., Who can say what the distribution of images is in the physical world ?

At best: we have an empirical distribution (our training dataset)

We will side-step the intractability issues by defining Neural Networks to learn an approximation.

- $\qrs\z|\x\Phi$: Neural Network, with weights Φ to approximate $\qr\z \mid \x$
    - The Encoder
- $\prs\x|\z\Theta$:Neural Network, with weights Θ, to approximate $\pr\x \mid \z$
    - The Decoder

These *learned* distributions (by training a model on the training dataset)

- are *approximations* of the true
- $\qr\z|\x$, $\pr\x|\z$

The mapping from Latent to reconstructed Input is not necessarily unique, thus we marginalize $\x$ over $\z$

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

Some obvious concerns about the integral

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

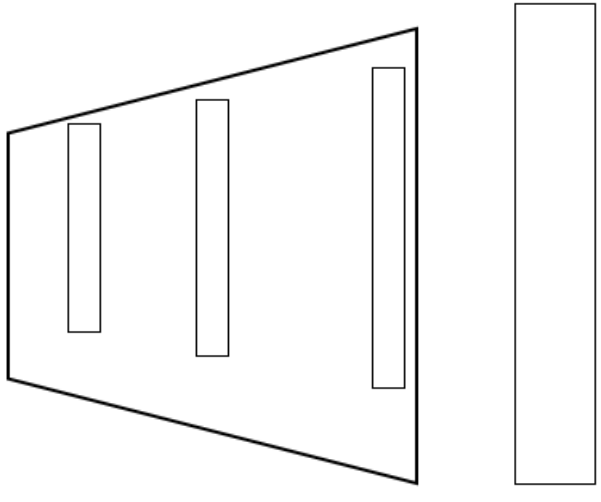
We can improve our sampling by considering only those choices of $\z$ that could generate $\x$ and re-write the objective as

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

Using the Decoder Neural Network $\backslash prs \backslash x | \backslash z \Theta$ to approximate $\backslash pr \backslash x | \backslash z$ gives rise to the following architecture

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

$\mathbf{z}^{(i)} \in q(\mathbf{z}|\mathbf{x})$



D
Decoder

We still can't train $\prs\z|\x\Theta$ because we don't know $\qr\z|\x$.

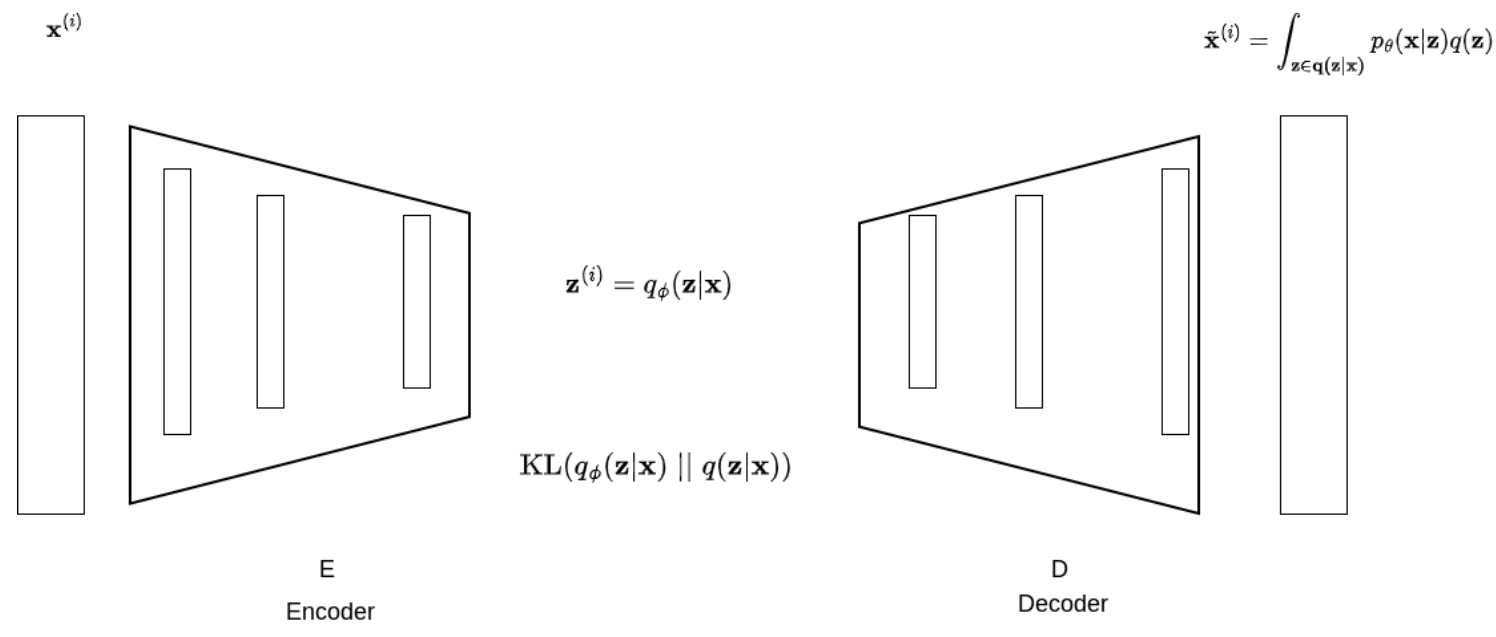
Let's use the Neural Network (Encoder) $\qrs\z|\x\Phi$ to approximate $\qr\z|\x$.

But we must **constrain the Encoder** to produce a distribution that approximates the true $\qr\z|\x$.

We do so by including this as a constraint (part of the Loss function used in training) using KL divergence as a measure of dissimilarity of two distributions

$$\KL(\qrs\z|\x\Phi \;||\; \qr\z|\x)$$

The resulting architecture:



We can train the Encoder/Decoder pair with the objective that the reconstructed $\tilde{\mathbf{x}}^{\mathrm{ip}}$ approximates the true $\mathbf{x}^{\mathrm{ip}}$ 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

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

Minimizing $\loss$ is equivalent to maximizing likelihood.

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

$$\begin{aligned} \loss &= -\log(\prs\x\Theta) + \KL(\qrs\z|\x\Phi \;||\; \qr\z|\x) \\ &= \loss_R + \loss_D \end{aligned}$$

which now has two objectives

- Reconstruction loss $\loss_R$: maximize the likelihood (by minimizing the negative likelihood)
- Divergence constraint $\loss_D$: $\qrs\z|\x\Phi$ must be close to $\qr\z|\x$

$$\begin{aligned} \loss_R &= -\log(p_\theta(\x)) \\ \loss_D &= \KL\left(\qrs\z|\x\Phi \;||\; \qr\z|\x\right) \end{aligned}$$

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

$$\loss = -\E_{\z\sim\qrs\z|\x\Phi}\left(\log(\prs\x|\z\Theta)\right) + \KL(\qrs\z|\x\Phi \;||\; \qr\z)$$

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

- Re-write
  $\log(p_\theta(\x)) = \E_{\z\sim\qrs\z|\x\Phi}\left(\log(\prs\x|\z\Theta)\right)$
- the KL term becomes
  $$\KL\left(\qrs\z|\x\Phi \;||\; \qr\z\right)$$
  rather than the original
  $$\KL\left(\qrs\z|\x\Phi \;||\; \qr\z|\x\right)$$

The purpose of re-writing: replace intractable $\qr\z|\x$ with a tractable $\qr\z$!

- So we can have a Loss function with which we can train !

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

Advanced: Obtain $\loss$ by rewriting

$\KL(\qrs\z|\x\Phi \,||\, \qr\z|\x)$

Let's derive a simpler expression for $\loss$ by manipulating $\KL(\qrs\z|\x\Phi \,||\, \qr\z|\x)$:

The key step:

- Using Bayes Theorem to re-write
$$\log(\qr\z|\x)$$
as
$$\log(\pr\x|\z) + \log(\qr\z) - \log(\pr\x)$$
- This allows us do away with intractable conditional probability $\qr\z|\x$
- In favor of unconditional probability $\qr\z$

The LHS cannot be optimized via SGD (recall the tractability issue with $\qr\z|\x$).

But the RHS can be made tractable by

- Approximating $\pr\x|\z$ with $\prs\x|\z\Theta$
- Assuming that the distributions $\qr\z$ (and $\pr\x$) as Normal

Choosing $\qr\z$

So what distribution should we use for the prior $\qr\z$?

- 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 $\qr\z$ is normal

- If we choose $\qr\z$ as normal, we can require $q_\phi(\z|\x)$ 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) Section 2.2

Re-parameterization trick

There is still one impediment to training.

It involves the random choice of $\z \sim \qrs\z|\x\Phi$ in

$$\loss_R = \E_{z\sim\qrs\z|\x\Phi}(\log(\prs\x|\z\Theta))$$

This is not a problem in the forward pass.

But in the backward pass we need to compute

$$\frac{\partial \loss_R}{\partial \Theta}$$

How do we back propagte through a random choice ?

The "reparameterization trick" redefines the random choice z as

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

That is

- Once we have defined $\mathrm{qr}\mathrm{z}$ 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

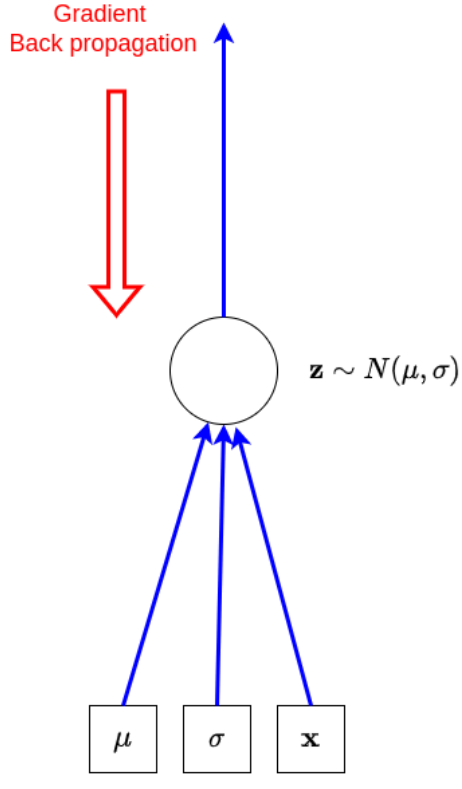
In this formulation, the random variable ϵ appears in a product term

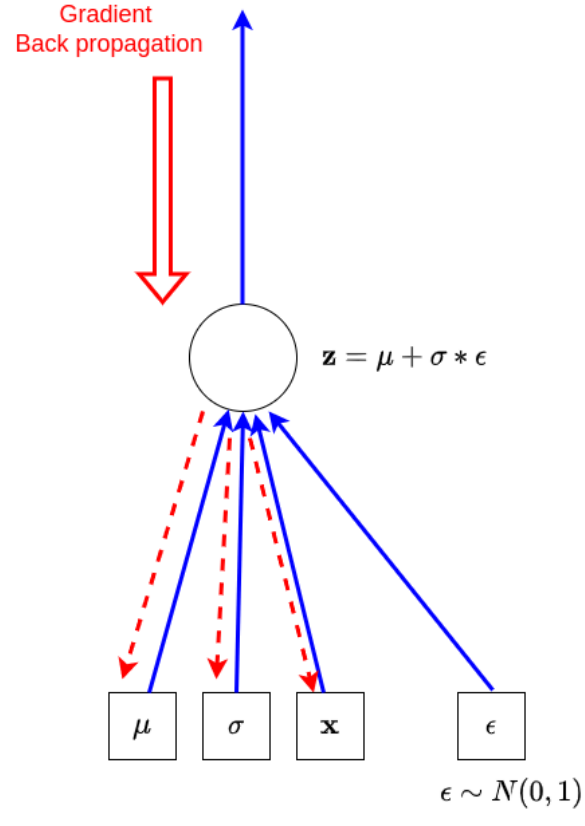
- We can differentiate the product with respect to Θ
- ϵ can be treated as a constant in $\frac{\partial \epsilon}{\partial \Theta}$

The Encoder design is now to produce (trainable parameters) μ_Θ, σ_Θ

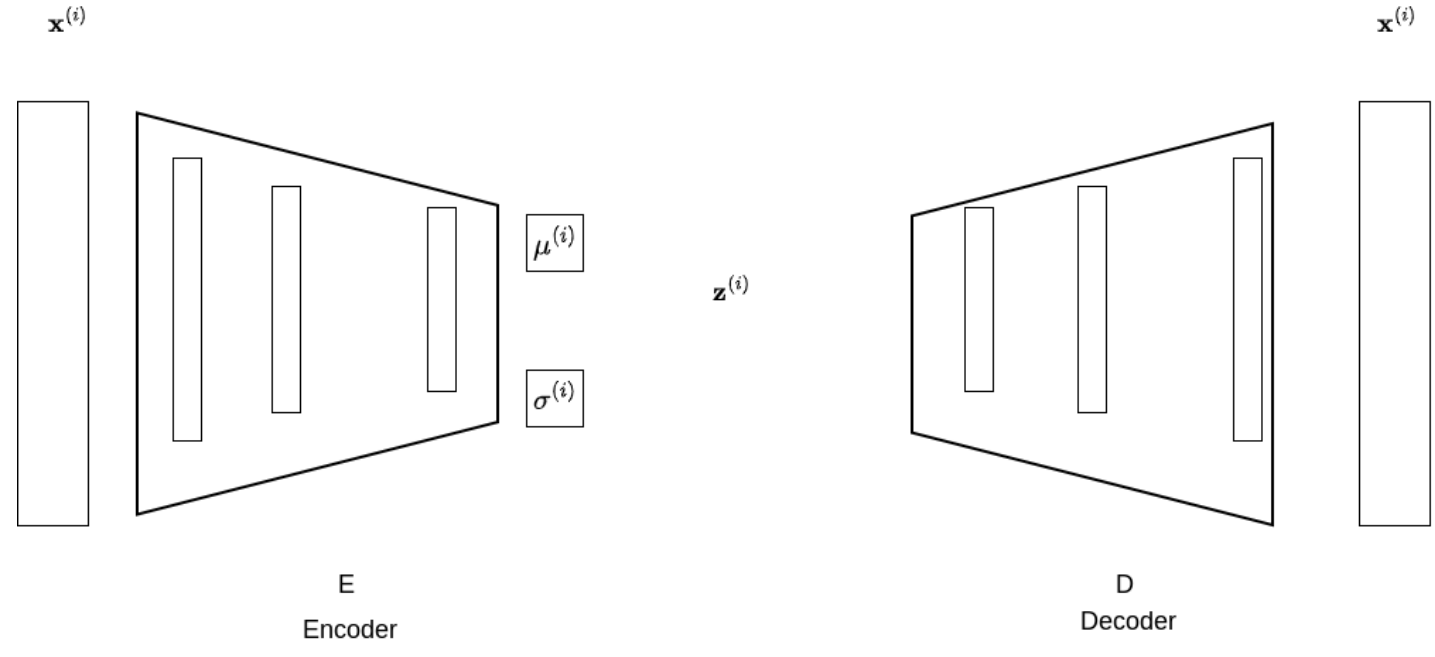
- And z indirectly

Reparameterization trick





This gets us to the final picture of the VAE:



To train a VAE:

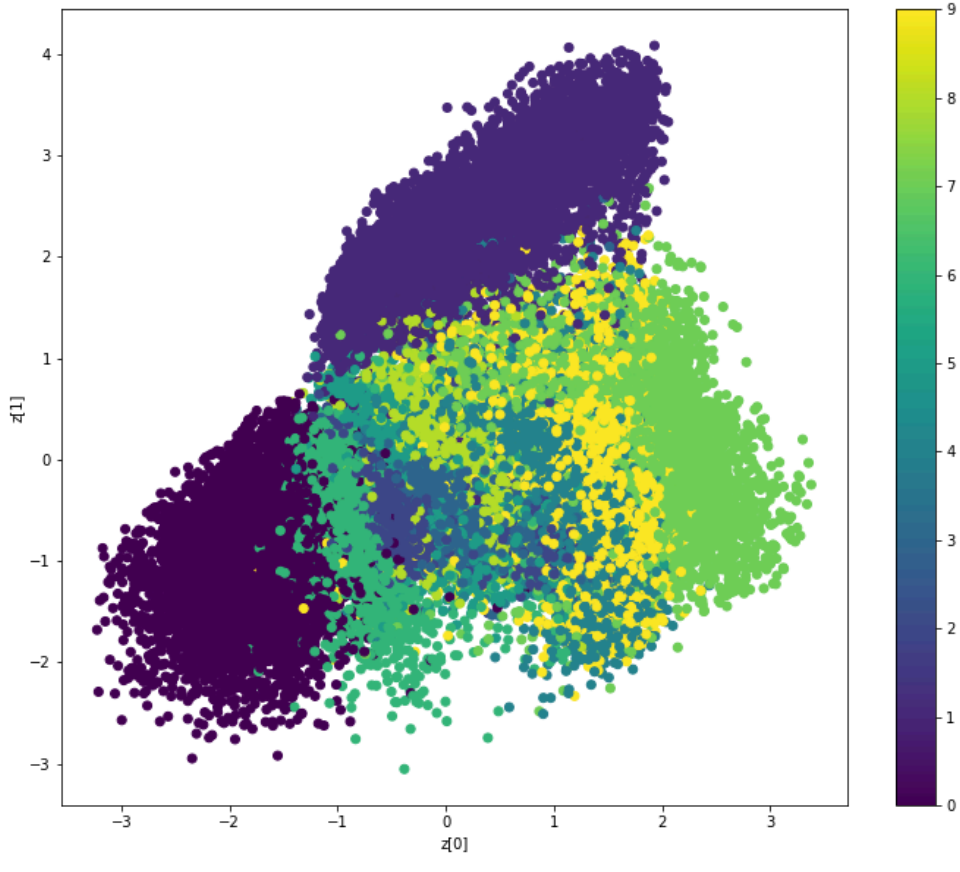
- pass input $\mathbf{x}^{(i)}$ through the Encoder, producing $\mu^{(i)}, \sigma^{(i)}$
  - use $\mu^{(i)}, \sigma^{(i)}$ to sample a latent representation $\mathbf{z}^{(i)}$ from the distribution
- pass the sampled $\mathbf{z}^{(i)}$ through the decoder, producing $D(\mathbf{z}^{(i)})$
- measure the reconstruction error $\mathbf{x}^{(i)} - D(\mathbf{z}^{(i)})$, 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.

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

MNIST examples: clustering of latent $\mathbf{z}$



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 verus Fashion MNIST
- The architecture of the Encoder and Decoder are different in the two models
    - The plain Autoencoder used extrememly simple architectures
        - Could the more complex architecture of the VAE Encoder/Decoder be the cause of tighter clustering ?

Certainly room for experimentation !

VAE in code

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="en
coder")
```

- 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, reconstruct
ion),
              axis=(1, 2),
          )
      )
      kl_loss = -0.5 * (1 + z_log_var - tf.square(z_mean) - tf.e
xp(z_log_var))
      kl_loss = tf.reduce_mean(tf.reduce_sum(kl_loss, axis=1))
      total_loss = reconstruction_loss + kl_loss
```

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 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

Encoder

input_10	input:	[(None, 28, 28, 1)]	[(None, 28, 28, 1)]

Using a VAE to produce synthetic examples

To give you an idea of the generative nature of the VAE we plot the *possible* set of outputs of the Decoder

- using arbitrary inputs to the Decoder

That is, we construct a plot using

- 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 $\mathbf{z}$
- that are not necessarily the output of the Encoder on any actual input $\mathbf{x}$

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

Note that the outputs

- are **not** instances of any examples
- There was no guarantee that a random $\backslash z$ would produce something that looked like a digit !

We may even be able to interpret the elements of $\backslash z$

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

ELBo (Evidence-based Lower Bound)

By re-writing the Loss, we removed the intractable term $\qr\z|\x$

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/) 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/), one can minimize the loss.

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 $\z$ and collapse to a vanilla VAE
- What would happen if we dropped $\loss_R$?
    - the encoding approximation $\qrs\z|\x\Phi$ would be close to the empirical $\pr\z|\x$ *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 !

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)

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	Decoder

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 z is also of length 2
    - In our grid illustration of generating synthetic examples, we vary each of the 2 components of 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

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

```
Done
```