Session 7 - Encoders and Variational Encoders

Due Sep 26 by 11:59pm **Points** 1,000 **Submitting** a text entry box or a website url
Available Sep 12 at 8am - Sep 26 at 11:59pm 15 days

This assignment was locked Sep 26 at 11:59pm.

Session 7 - Variational AutoEncoders

In a lot of problems we are trying to solve we are looking at a variety of data, it could be:

images

text

audio

etc

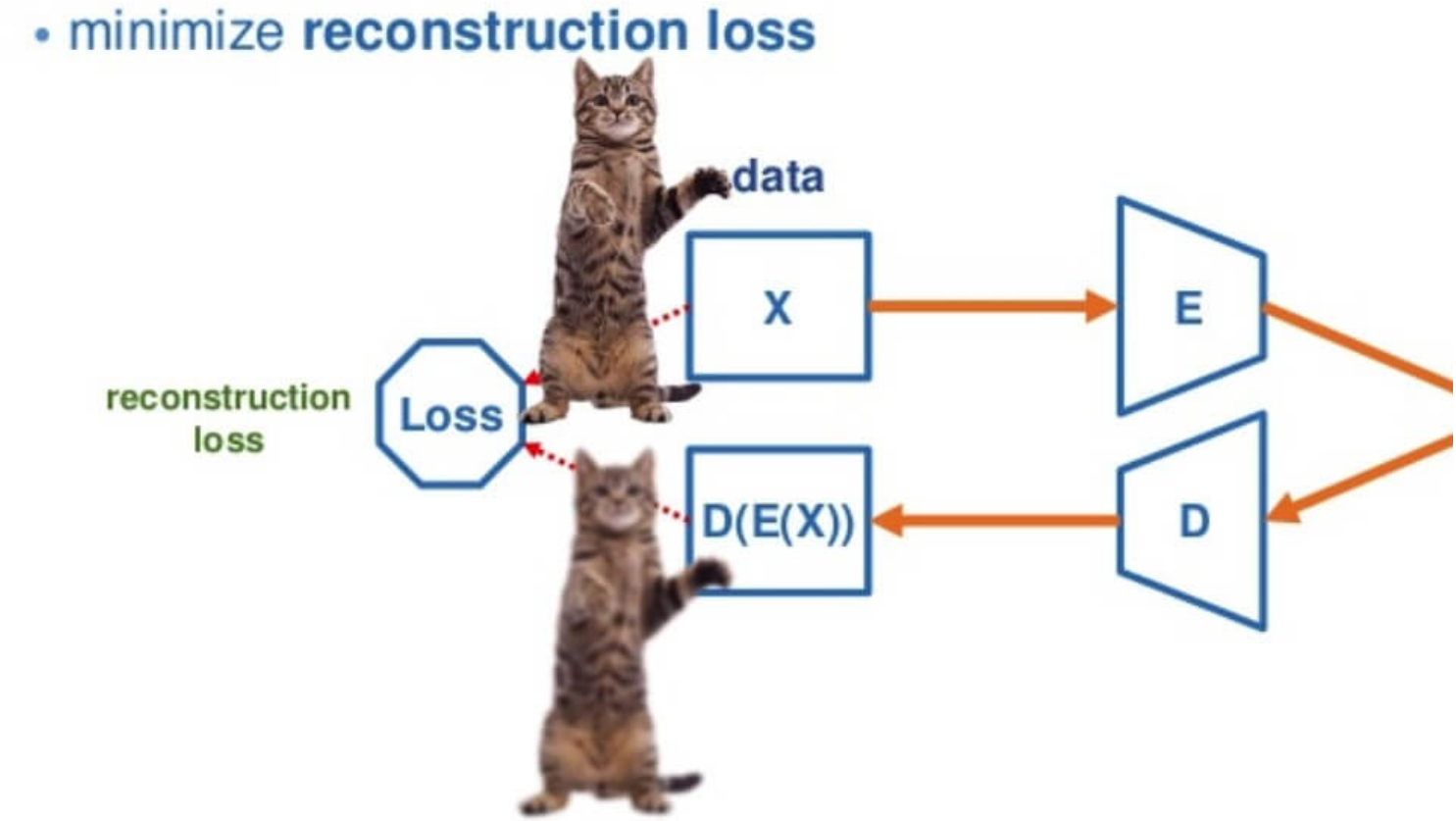
but the underlying process in which this data is created could be much simpler and of much lower dimension space than the actual data we are looking at.

$$z_{n+1} = z_n^2 + c$$

That is why in lot of techniques in machine learning we are trying to compress the dimensionality of the data into a smaller space.

Before we dive into Vatiational Autoencoders, let's see what AutoEncoders are.

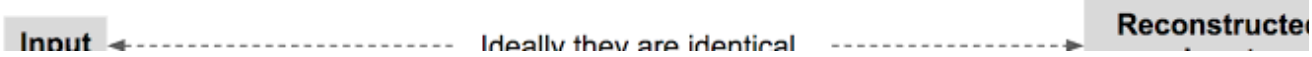
AUTOENCODERS

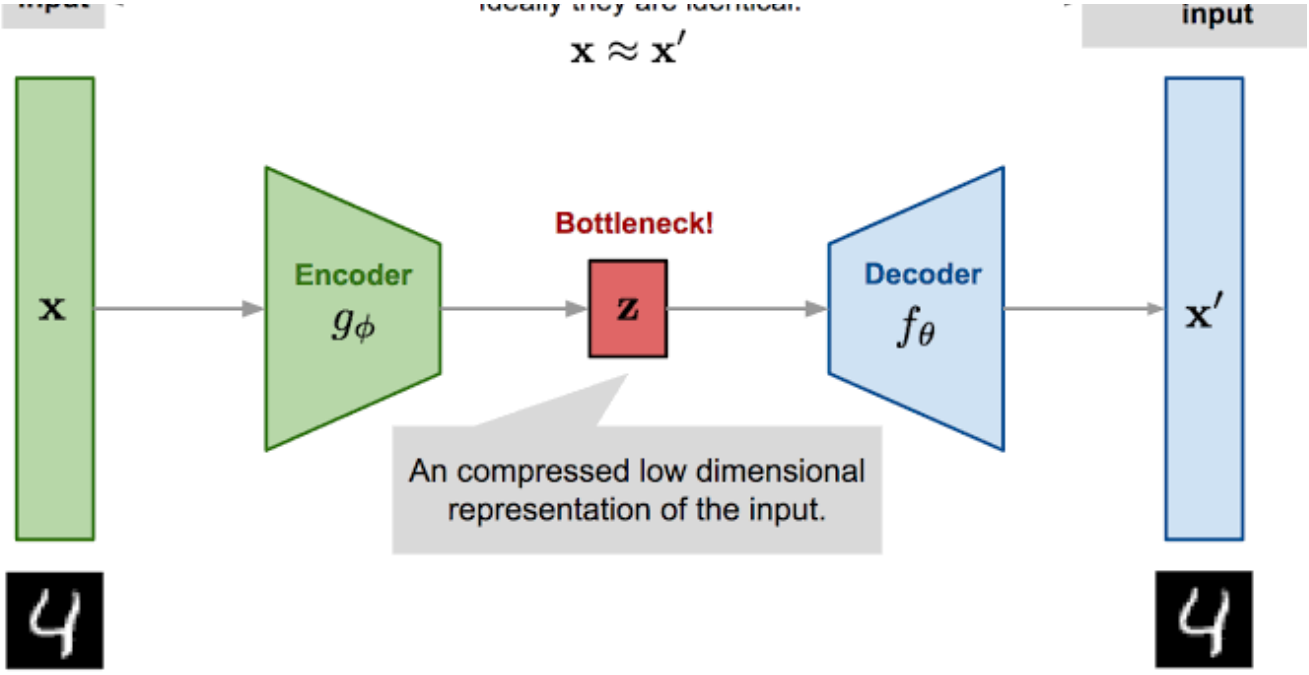


Autoencoder is a neural network designed to learn an identity function in an unsupervised way to reconstruct the original input while compressing the data in the process so as to discover a more efficient and compressed representation. The idea was originated in **the 1980s (https://en.wikipedia.org/wiki/Autoencoder)** , and later promoted by the seminal paper by **Hinton & Salakhutdinov, 2006 (https://pdfs.semanticscholar.org/c50d/ca78e97e335d362d6b991ae0e1448914e9a3.pdf)** .

It consists of two networks:

- *Encoder* network: It translates the original high-dimension input into the latent low-dimensional code. The input size is larger than the output size.
- *Decoder* network: The decoder network recovers the data from the code, likely with larger and larger output layers.





The encoder network essentially accomplishes the **dimensionality reduction** (https://en.wikipedia.org/wiki/Dimensionality_reduction), just like how we would use Principal Component Analysis (PCA) or Matrix Factorization (MF) for. In addition, the autoencoder is explicitly optimized for the data reconstruction from the code. A good intermediate representation not only can capture latent variables but also benefits a full **decompression** (https://ai.googleblog.com/2016/09/image-compression-with-neural-networks.html) process.

There are various metrics to quantify the difference between two vectors, such as cross-entropy when the activation function is sigmoid, or as simple as MSE loss:

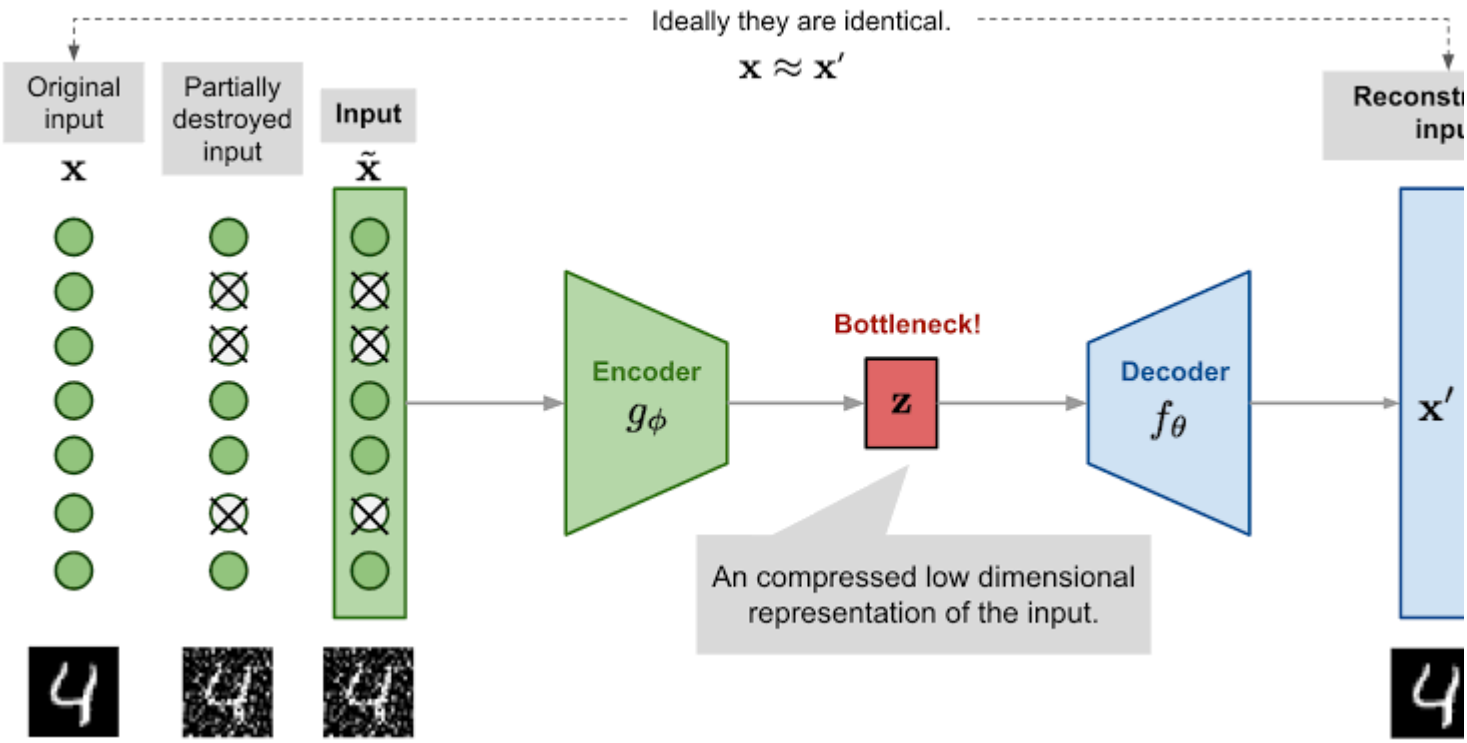
$$L_{AE}(\theta, \phi) = \frac{1}{n} \sum_{i=1}^{n} \left(x^i - f_\theta\left(g_\phi\left(x^i\right)\right)\right)^2$$

Latent Space Representation is also called **Bottleneck**.

Let's look at the **CODE** (https://colab.research.google.com/github/smartgeometry-ucl/dl4g/blob/master/autoencoder.ipynb) 1 and **CODE2** (https://github.com/udacity/deep-learning-v2-pytorch/blob/master/autoencoder/convolutional-autoencoder/Convolutional_Autoencoder_Solution.ipynb) and **CODE3** (https://github.com/udacity/deep-learning-v2-

pytorch/blob/master/autoencoder/convolutional-autoencoder/Upsampling_Solution.ipynb)

DENOISING AUTOENCODERS



(https://colab.research.google.com/github/smartgeometry-ucl/dl4g/blob/master/autoencoder.ipynb)

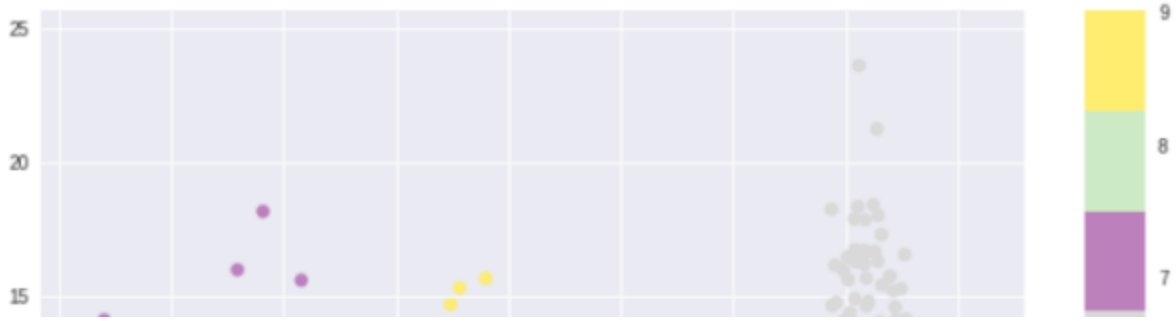
Let's look at a **Denoising AutoEncoder** (https://github.com/udacity/deep-learning-v2-pytorch/blob/master/autoencoder/denoising-autoencoder/Denoising_Autoencoder_Solution.ipynb)

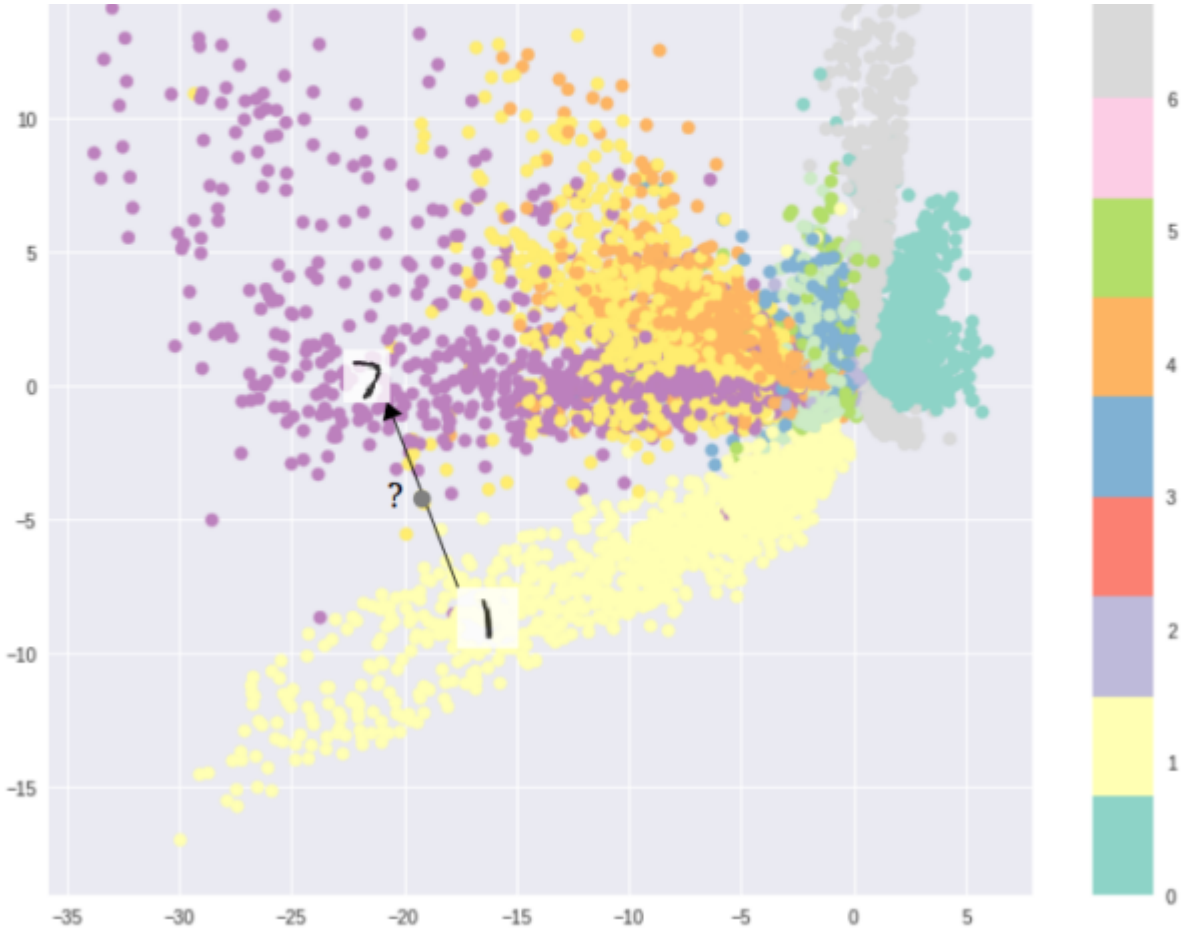
DENOISING AUTOENCODERS

The **shortcomings** (https://www.topbots.com/intuitively-understanding-variational-autoencoders/) with Standard Autoencoder

Standard autoencoders learn to generate compact representations and reconstruct their inputs well, but asides from a few applications like denoising autoencoders, they are fairly limited.

The fundamental problem with autoencoders, for generation, is that the latent space they convert their inputs to and where their encoded vectors lie, may not be continuous, or allow easy interpolation.





For example, training an autoencoder on the MNIST dataset, and visualizing the encodings from a 2D latent space reveals the formation of distinct clusters. This makes sense, as distinct encodings for each image type makes it far easier for the decoder to decode them. This is fine if you're just *replicating* the same images.

But when you're building a *generative* model, you **don't** want to prepare to *replicate* the same image you put in. You want to randomly sample from the latent space, or generate variations on an input image, from a continuous latent space.

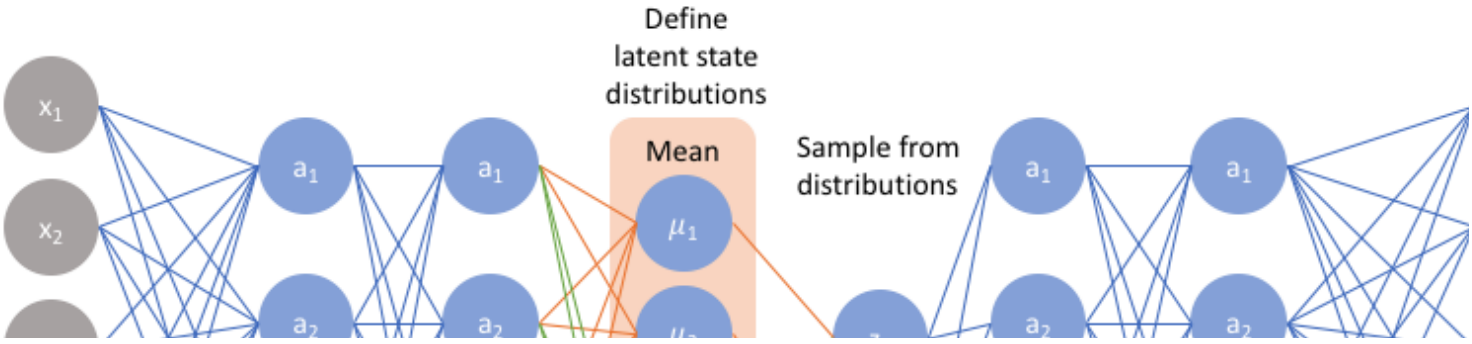
If the space has discontinuities (eg. gaps between clusters) and you sample/generate a variation from there, the decoder will simply generate an unrealistic output, because the decoder has *no idea* how to

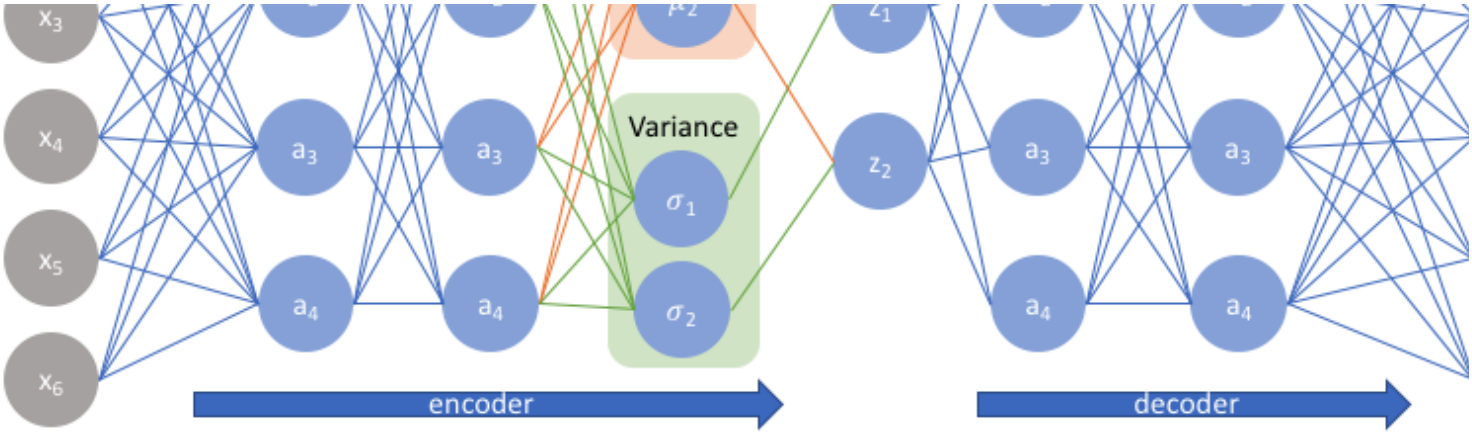
deal with that region of the latent space. During training, it *never saw* encoded vectors coming from that region of latent space.

VARIATIONAL AUTOENCODERS

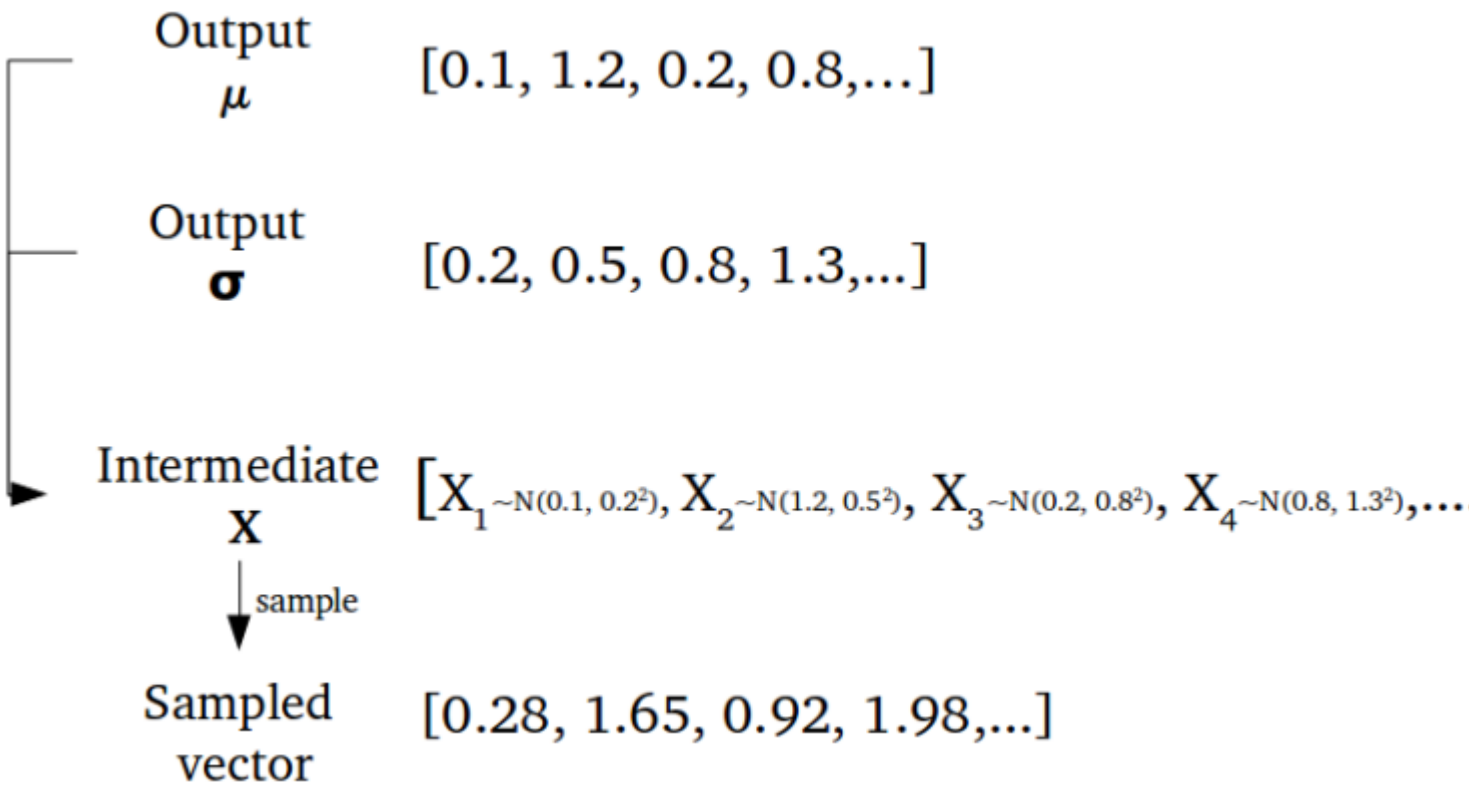
Variational Autoencoders (VAEs) have one fundamentally unique property that separates them from vanilla autoencoders, and it is this property that makes them so useful for generative modeling: their latent spaces are, *by design,* continuous, allowing easy random sampling and interpolation.

It achieves this by doing something that seems rather surprising at first: making its encoder not output an encoding vector of size *n,* rather, outputting two vectors of size n: a vector of means**, μ**, and another vector of standard deviations, **σ**.

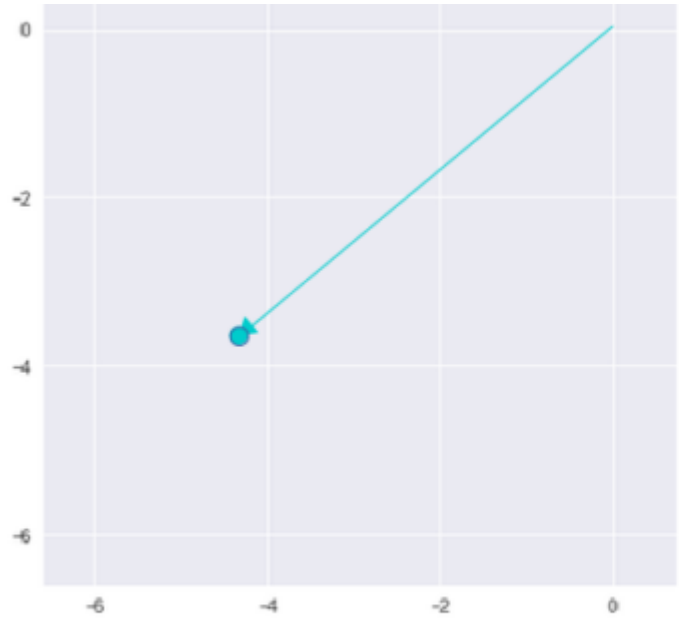




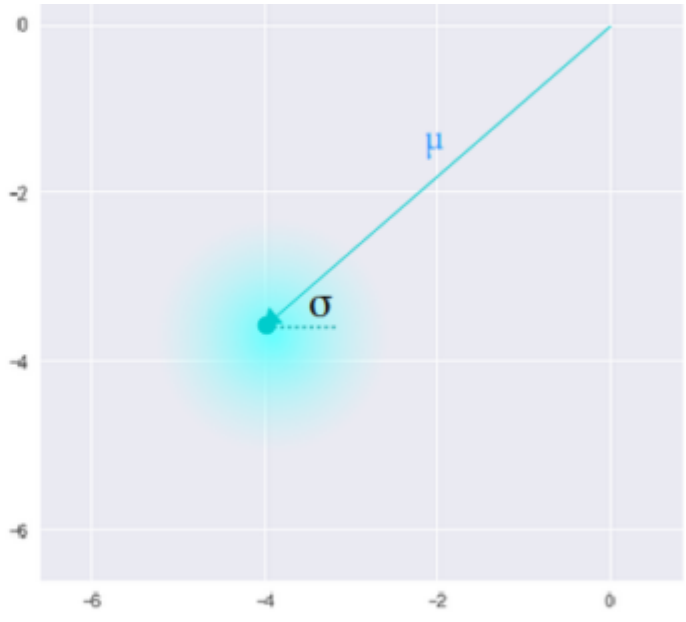
They form the parameters of a vector of random variables of length n, with the *i* th element of **μ** and **σ** being the mean and standard deviation of the *i* th random variable, **X** i, from which we sample, to obtain the sampled encoding which we pass onward to the decoder:



This stochastic generation means, that even for the same input, while the mean and standard deviations remain the same, the actual encoding will somewhat vary on every single pass simply due to sampling.



Standard Autoencoder
(direct encoding coordinates)

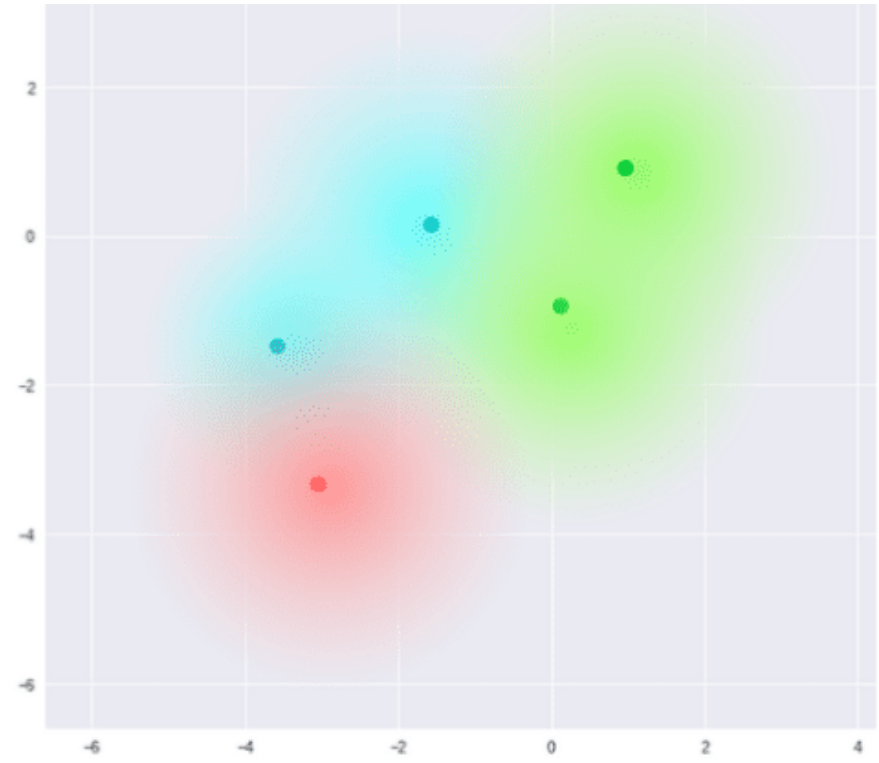


Variational Autoencoder
(μ and σ initialize a probability distribution)

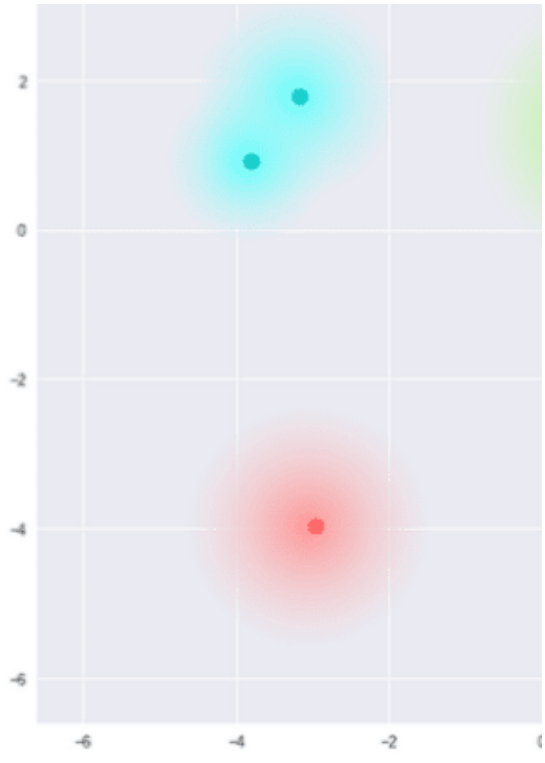
Intuitively, the mean vector controls where the encoding of an input should be centered around, while the standard deviation controls the “area”, how much from the mean the encoding can vary. As encodings are generated at random from anywhere inside the “circle” (the distribution), the decoder learns that not only is a single point in latent space referring to a sample of that class, but all nearby points refer to the same as well. This allows the decoder to not just decode single, specific encodings in the latent space (leaving the decodable latent space discontinuous), but ones that slightly vary too, as the decoder is exposed to a range of variations of the encoding of the same input during training

The model is now exposed to a certain degree of local variation by varying the encoding of one sample, resulting in smooth latent spaces on a local scale, that is, for similar samples. Ideally, we want overlap between samples that are not very similar too, in order to interpolate *between* classes. However, since there are *no limits* on what values vectors **μ** and **σ** can take on, the encoder can learn to generate very

different **μ** for different classes, clustering them apart, and minimize **σ**, making sure the encodings themselves don’t vary much for the same sample (that is, less uncertainty for the decoder). This allows the decoder to efficiently reconstruct the *training* data.



What we require



What we may inadverter

What we ideally want are encodings, *all* of which are as close as possible to each other while still being distinct, allowing smooth interpolation, and enabling the construction of *new* samples.

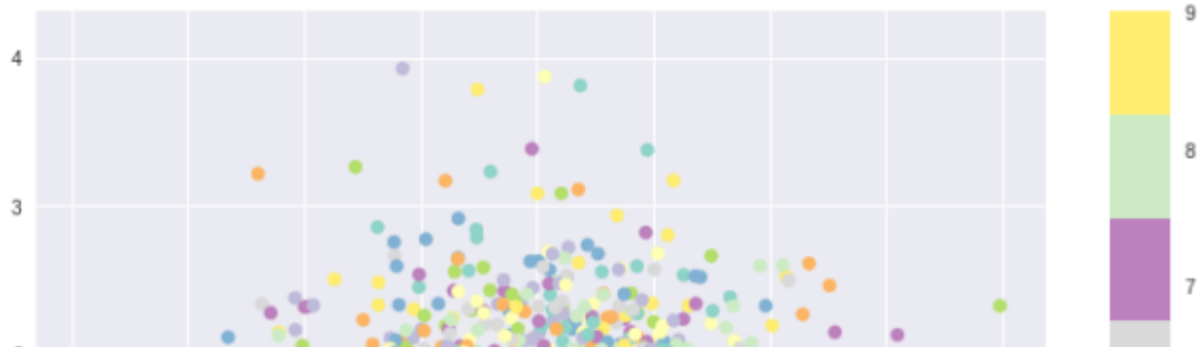
In order to force this, we introduce the Kullback–Leibler divergence (KL divergence [2] (https://www.countbayesie.com/blog/2017/5/9/kullback-leibler-divergence-explained)) into the loss function. The KL divergence between two probability distributions simply measures how much they *diverge* from each other. Minimizing the KL divergence here means optimizing the probability distribution parameters **(μ** and **σ)** to closely resemble that of the target distribution.

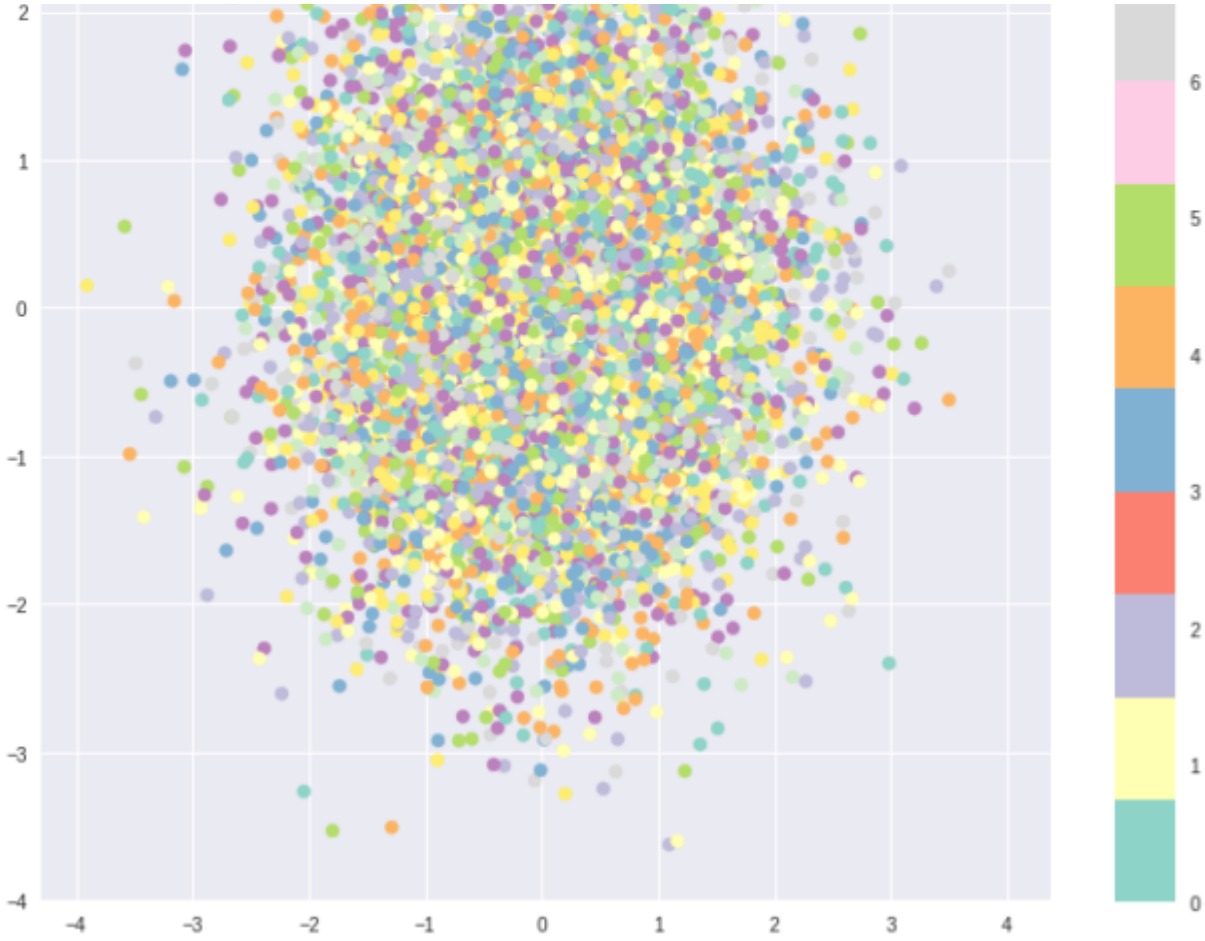
$$\sum_{i=1}^{n} \sigma_i^2 + \mu_i^2 - \log(\sigma_i) - 1$$

For VAEs, the KL loss is equivalent to the *sum* of all the KL divergences between the *component* $X_i \sim N(\mu_i, \sigma_i^2)$ in **X**, and the standard normal [3] (https://stats.stackexchange.com/questions/7440/kl-divergence-between-two-univariate-gaussians) . It's minimized when $\mu_i = 0$, $\sigma_i = 1$.

Intuitively, this loss encourages the encoder to distribute all encodings (for all types of inputs, eg. all MNIST numbers), evenly around the center of the latent space. If it tries to "cheat" by clustering them apart into specific regions, away from the origin, it will be penalized.

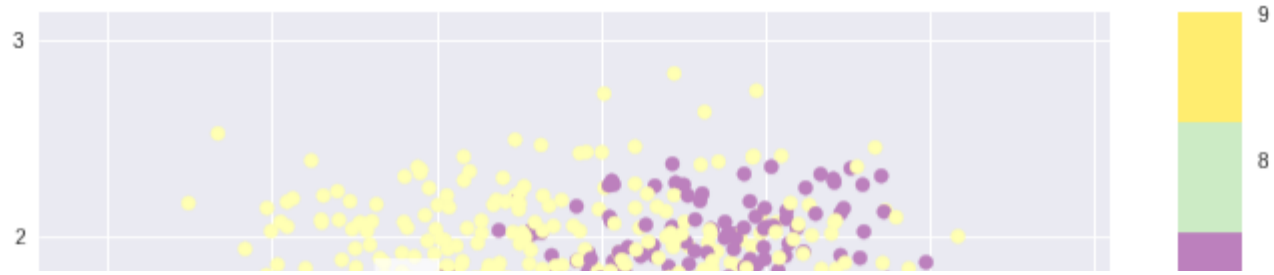
Now, using purely KL loss results in a latent space results in encodings densely placed randomly, near the center of the latent space, with little regard for similarity among nearby encodings. The decoder finds it impossible to decode anything meaningful from this space, simply because there really isn't any meaning.

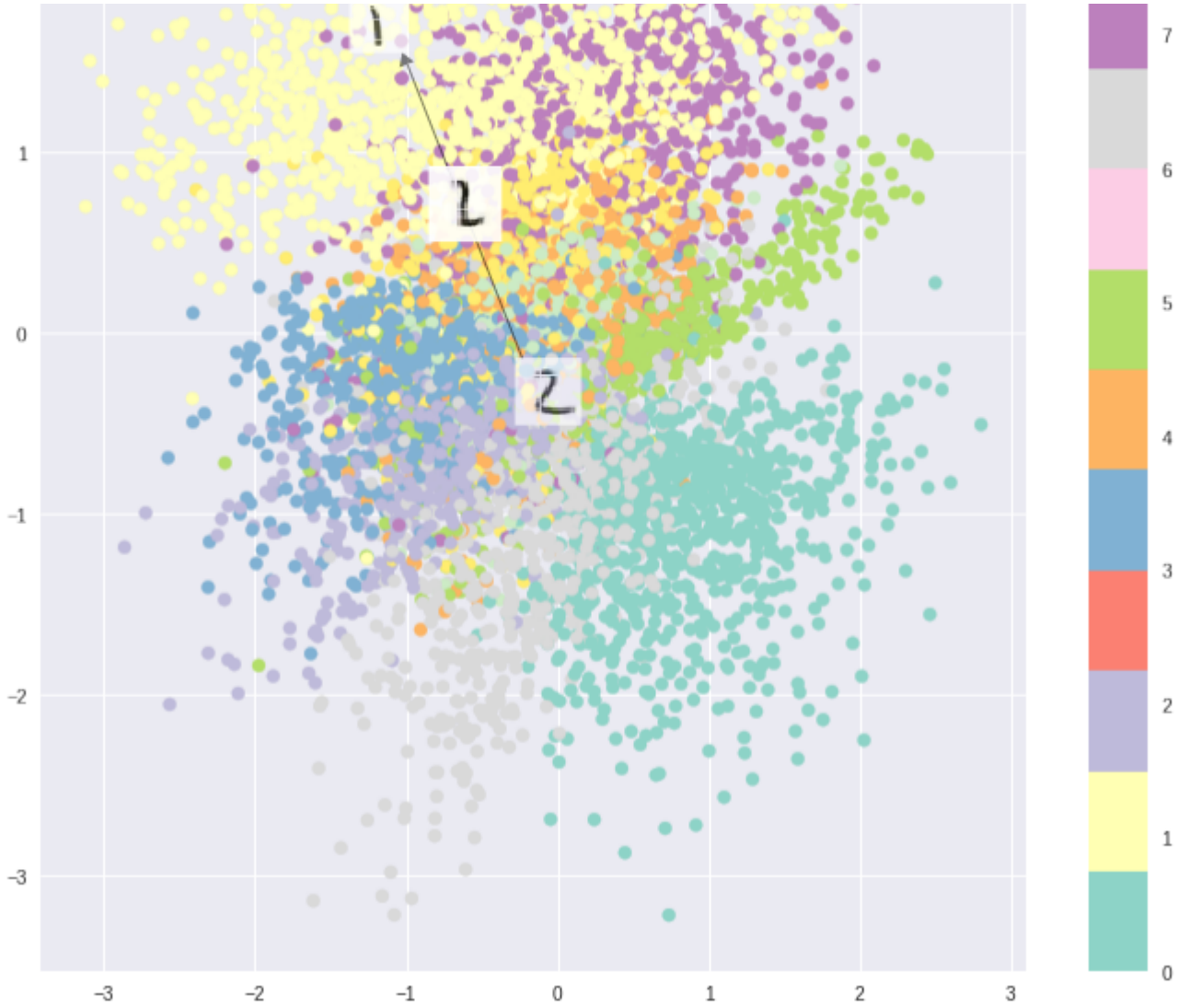




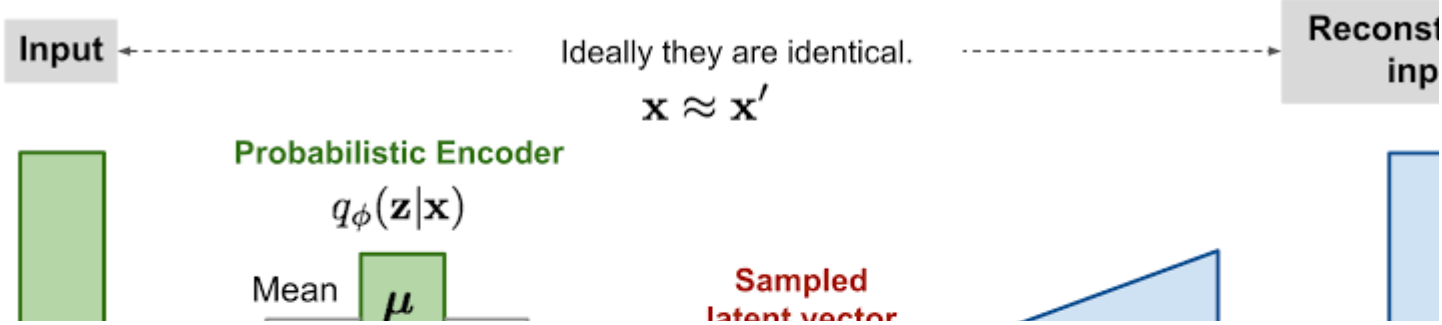
Optimizing the two together, however, results in the generation of a latent space that maintains the similarity of nearby encodings on the *local scale* via clustering, yet *globally,* is very densely packed near the latent space origin (compare the axes with the original).

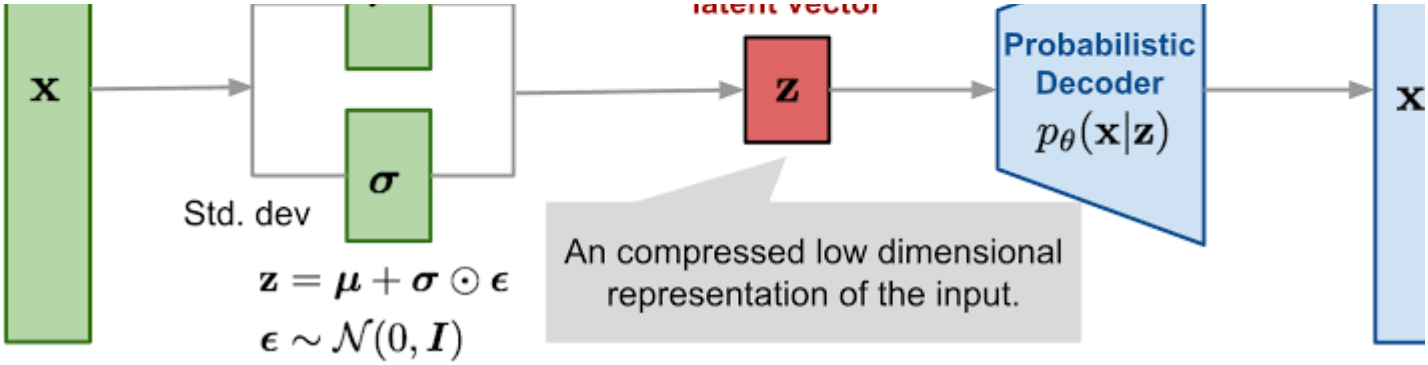
$$\mathcal{L}(\theta, \phi; \mathbf{x}, \mathbf{z}) = \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] - D_{KL}\big(q_\phi(\mathbf{z}|\mathbf{x})$$



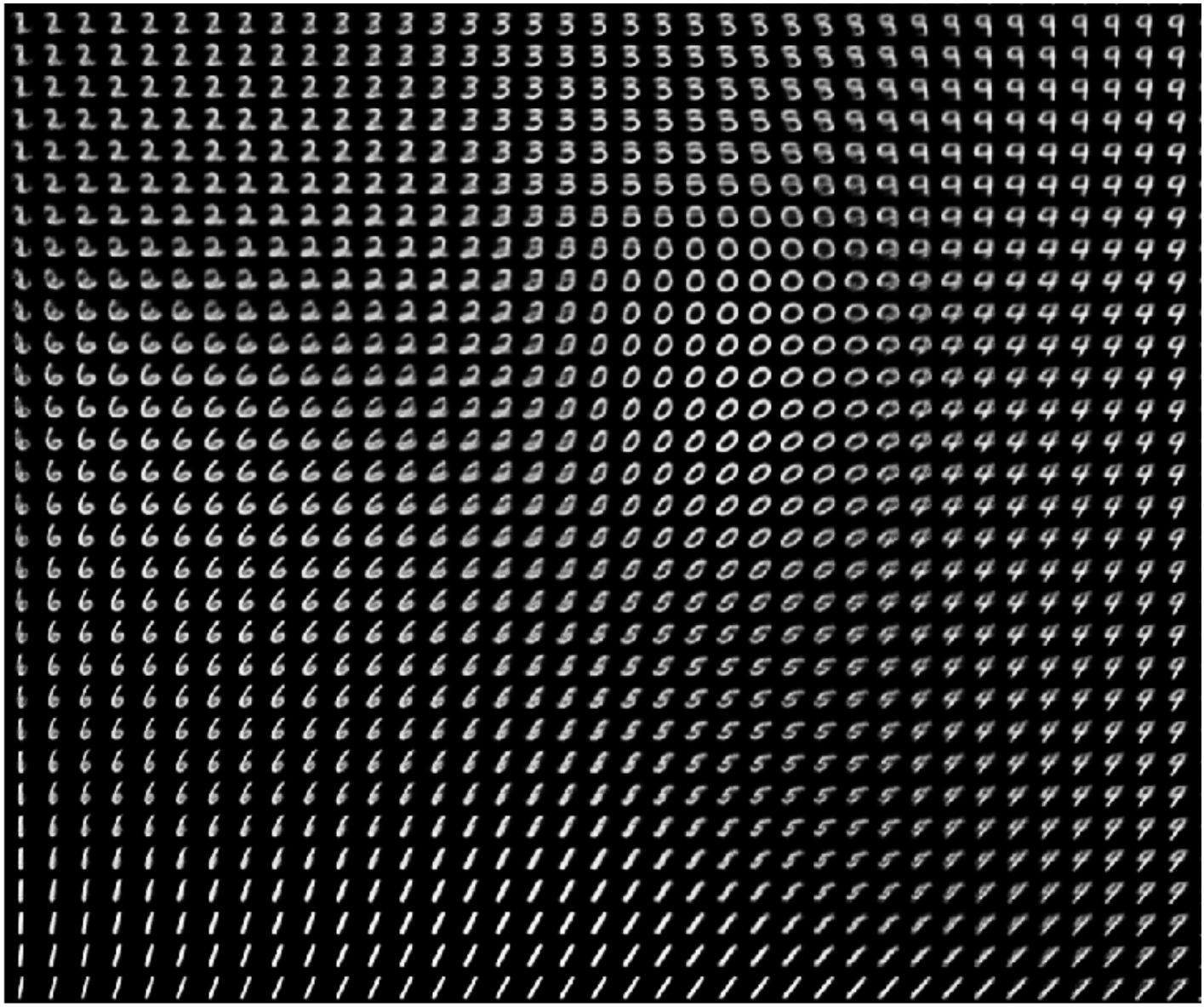


Intuitively, this is the equilibrium reached by the *cluster-forming* nature of the reconstruction loss, and the *dense packing* nature of the KL loss, forming distinct clusters the decoder can decode. This is great, as it means when randomly generating, if you sample a vector from the same prior distribution of the encoded vectors, $N(\mathbf{0}, \mathbf{I})$, the decoder will successfully decode it. And if you're interpolating, there are no sudden gaps between clusters, but a *smooth mix of features* a decoder can understand.





$$\mathcal{L}(\theta, \phi; \mathbf{x}, \mathbf{z}) = \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] - D_{KL}\big(q_\phi(\mathbf{z}|\mathbf{x}) \parallel p(\mathbf{z})\big)$$



Let's look at some **code** **(https://vxlabs.com/2017/12/08/variational-autoencoder-in-pytorch-commented-and-annotated/)** .

SOME MATH

- $p(x)$ - Probability
- $p(x|y)$ - Conditional Probability
- $E[\cdot]$ - Expectation

- KL Divergence

$p(x)$- Probability

Defines the probability of random variable/incident occurring

$p(x|y)$ - Conditional Probability

Defines the probability of random variable/incident x provided y has happened.

$$p(x \mid y) = \frac{p(x|y)p(y)}{p(x)}$$

When a die is tossed once, what is the probability of getting 3?

In tossing a fair die, what is the probability that 3 has occurred conditioned on the toss value being odd?

The theorem of Total Probability:

Let y1, y2, y3, ... yN be a set of **mutually exclusive events** & event x is the union of N mutually exclusive events, then

$$p(x) = \sum_{i=1}^{n} p(x \mid y_i)\, p(y_i)$$

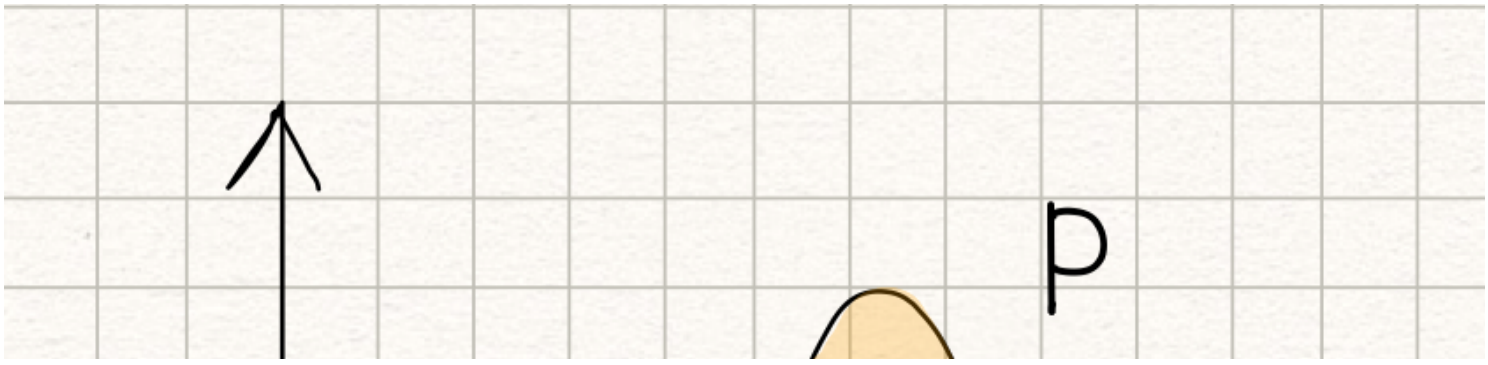
$E[\cdot]$- The expectation of random variable

$$E(x) = \sum_{i=1}^{k} x_i \, p(x = x_i)$$

Let x represent the outcome of a fair die, what is the E(x)?

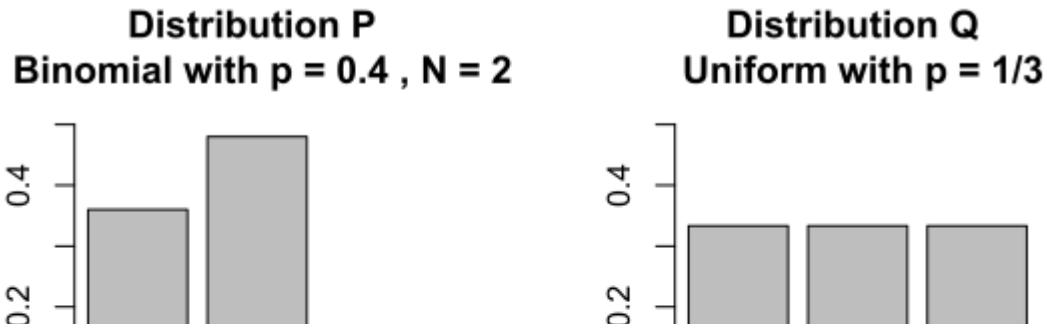
$E(x)$= 1*1/6 + 2*1/6 + 3*1/6 + 4*1/6 + 5*1/6 + 6*1/6 = 3.5

KULLBACK-LIEBLER DIVERGENCE



It is the measure of how one probability distribution is different from the second. For the discrete probability distribution p & q, the KL Divergence between them is defined as

$$D_{KL}\left(p \parallel q\right) = \sum_{x} p\left(x\right) \log\left(\frac{p(x)}{q(x)}\right)$$



$$
\begin{aligned}
D_{\mathrm{KL}}(P \parallel Q) &= \sum_{x \in \mathcal{X}} P(x) \ln\left(\frac{P(x)}{Q(x)}\right) \\
&= 0.36 \ln\left(\frac{0.36}{0.333}\right) + 0.48 \ln\left(\frac{0.48}{0.333}\right) + 0.16 \ln\left(\frac{0.16}{0.333}\right) \\
&= 0.0852996
\end{aligned}
$$

$$
\begin{aligned}
D_{\mathrm{KL}}(Q \parallel P) &= \sum_{x \in \mathcal{X}} Q(x) \ln\left(\frac{Q(x)}{P(x)}\right) \\
&= 0.333 \ln\left(\frac{0.333}{0.36}\right) + 0.333 \ln\left(\frac{0.333}{0.48}\right) + 0.333 \ln\left(\frac{0.333}{0.16}\right) \\
&= 0.097455
\end{aligned}
$$

Variational Autoencoders

ASSIGNMENT

Solve the last problem using VAEs.

VIDEO

EVA4P2S7