Fundamental Generative AI Part 1: Variational Autoencoders (VAE)

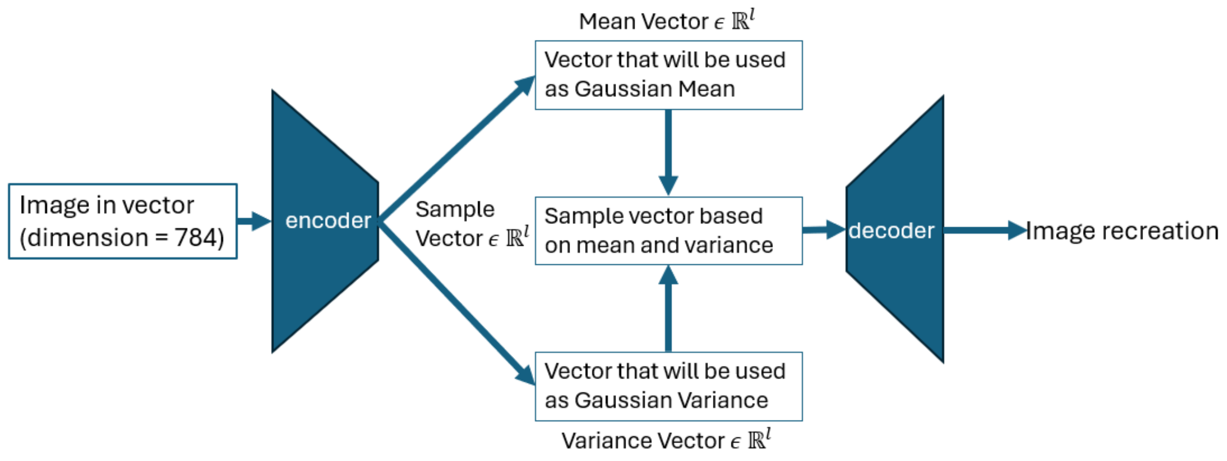
Introduction:

It's a bit of a misnomer to label "fundamental" and "Generative AI" together, as generative AI is itself an advanced field. However, in the upcoming series (the exact number of which I haven't decided yet), I'll explore the research that has laid the foundation for modern generative AI models like DALL-E, developed by OpenAI. In this paper, I'll focus on an article by Carl Doersch titled "Tutorial on Variational Autoencoders," which explains the concept of variational autoencoders—a key foundation for modern generative AI.

Concept of Variational Autoencoder:

Before diving into Variational Autoencoders (VAE), I'd like to lay out some foundational concepts. When I first encountered VAE during my Master's course, it took me some time to fully grasp the underlying motivations behind the model. Understanding the "why" of the problem is crucial, so I'll take a moment to explain.

VAE operates on the following principle. See the image below for a visual representation:


Mean Vector $\epsilon \mathbb{R}^l$
Vector that will be used as Gaussian Mean
Image in vector (dimension = 784)
encoder
Sample Vector $\epsilon \mathbb{R}^l$
Sample vector based on mean and variance
decoder
Image recreation
Vector that will be used as Gaussian Variance
Variance Vector $\epsilon \mathbb{R}^l$

For now, let's assume the encoders and decoders are already trained (I'll explain the training process later) and focus on understanding the concept behind the architecture.

First, the VAE takes in an original image and outputs another image that is similar to the original. It's not a text-to-image or sound-to-image generator. Instead, it treats the original image as a "meta-image" and generates an image similar to it.

Next, let's look at what the encoder does. It takes the original image, which is in vector form, and produces two vectors. The dimensions of these two vectors are a hyperparameter that can be specified by the user, known as the **latent dimension**, represented by l. These vectors are also referred to as **latent variables** in the latent space.

The two vectors correspond to the **mean** and **variance** of a Gaussian distribution. It's important not to confuse this with the mean and variance of the original input image, X. The encoder isn't calculating the mean or variance of the input; instead, it's defining a Gaussian space with these two vectors. For example, if the encoder produces a mean vector of [1.5, 1.0, 2.0] and a variance vector of [0.2, 0.4, 0.1], we can say the input has been transformed into a latent space with a mean of [1.5, 1.0, 2.0] and variance of [0.2, 0.4, 0.1].

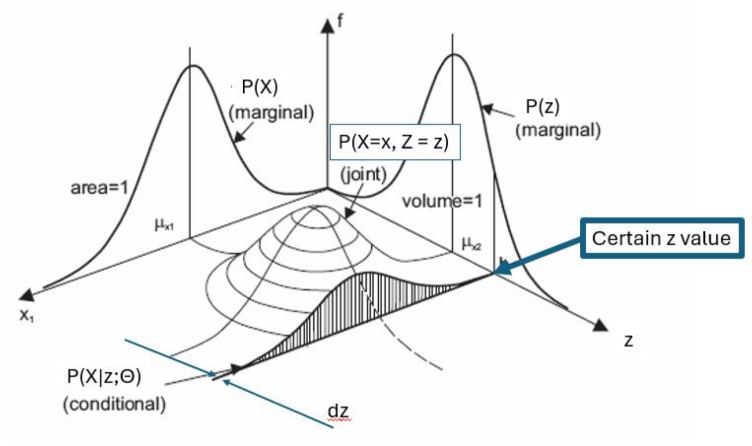
You might wonder why we're suddenly converting the image into two vectors. This is a key concept in generative AI. If we simply encoded the image into a vector and fed it directly into the decoder (e.g., [1.5, 1.0, 2.0]), we would always get the same output image—this would be a deterministic process. That's not generative AI. However, by defining both the mean and variance, and then **sampling** from this latent Gaussian space, we introduce variability. Even with the same input image, the encoder will produce different outputs for each sample, which, when fed into the decoder, will generate different images. This variability is what makes the model generative, as it introduces randomness in the output while keeping the mean and variance consistent.

Mathematical development of the Variation Autoencoder:

The fundamental equation: We can begin with the basic equation that defines **X** (the input image) and **z** (the Gaussian distribution function, not the sampled values themselves).

$$P(X) = \int P(X|z;\theta)P(z)dz. \qquad (1)$$

At this point, it may not seem like the encoder is connected to the function, but with a mathematical trick to generate the **z** distribution, the role of the encoder will become clear. Let's take a closer look at this equation. I'm explaining this for beginners in statistics (and honestly, I consider myself a beginner too), so I'll go through the details as well. See the image below:



With the image in mind, we can now explain some concepts from the paper. First, **P(X|z; Θ)** in simple terms means the following: a specific value of **z** is passed through a function with parameter **Θ**, which produces the 2-D distribution of **X** (the shaded part of the image). **P(z)** represents the probability distribution of **z**. So, **P(X|z; Θ) P(z)** simply gives the value of the **X** distribution for a specific **z** value, passed through the function with parameter **Θ**, multiplied by the **P(z)** value at that **z**. This result is then integrated along the **z**-axis. The outcome of this integration is called the **marginal distribution**, **P(X)**, which is shown as the Gaussian distribution in the top-left of the image.

There are a few important points to consider with this equation.

1. **P(X)** and **P(z)** are multi-dimensional distributions. This means that **P(X)** is not just a single Gaussian distribution, but a set of multiple Gaussian distributions, each one depending on several values of **P(z)**. For example, if **X** is an MNIST image with 28 x 28 pixels, then **P(X)** will actually consist of 784 Gaussian distributions (one for each pixel). Now, if we choose **z** to have 5 dimensions (we can choose the number of dimensions based on available computing power), we would perform the integration 5 times to find the marginal distribution for just **one pixel** of **P(X)**. This process needs to be repeated for every pixel in the image, as each pixel has its own distribution. Choosing fewer dimensions for **z** reduces the amount of information stored but makes the model more computationally efficient. On the other hand, choosing more dimensions (like 10) increases the information capacity but requires more computational resources.

2. The goal of developing this equation is to calculate **maximum likelihood**. Specifically, we want to maximize the value of **P(X)**. This doesn't mean we want **P(X)** to always be 1. Remember, **X** is the original image you started with. So, maximizing **P(X)** simply means we want the output to closely match the original image, which makes sense. You want to learn the function parameter **Θ** in such a way that the **z** is transformed back into the original image as accurately as possible. The goal is for the output to resemble the original image, but it should **not** be deterministic. If the output were deterministic, it

wouldn't be considered generative AI. Instead, you want the model to generate different outputs for different samples of **z**, so the decoder can produce the correct output based on each new **z** generated by the encoder.

Also, in the paper, **all distributions** are assumed to be Gaussian. If you're wondering how a Gaussian distribution can be used to encode complex images, the answer lies in the power of neural networks. Neural networks are capable of taking any distribution and transforming it into almost any desired output. This is because a neural network is essentially a very complex function that can model intricate patterns and relationships within the data, allowing it to represent highly complex distributions, including those needed for image generation.

Mathematical Tricks to Solve the Maximization (Kullback-Leibler + Bayesian):

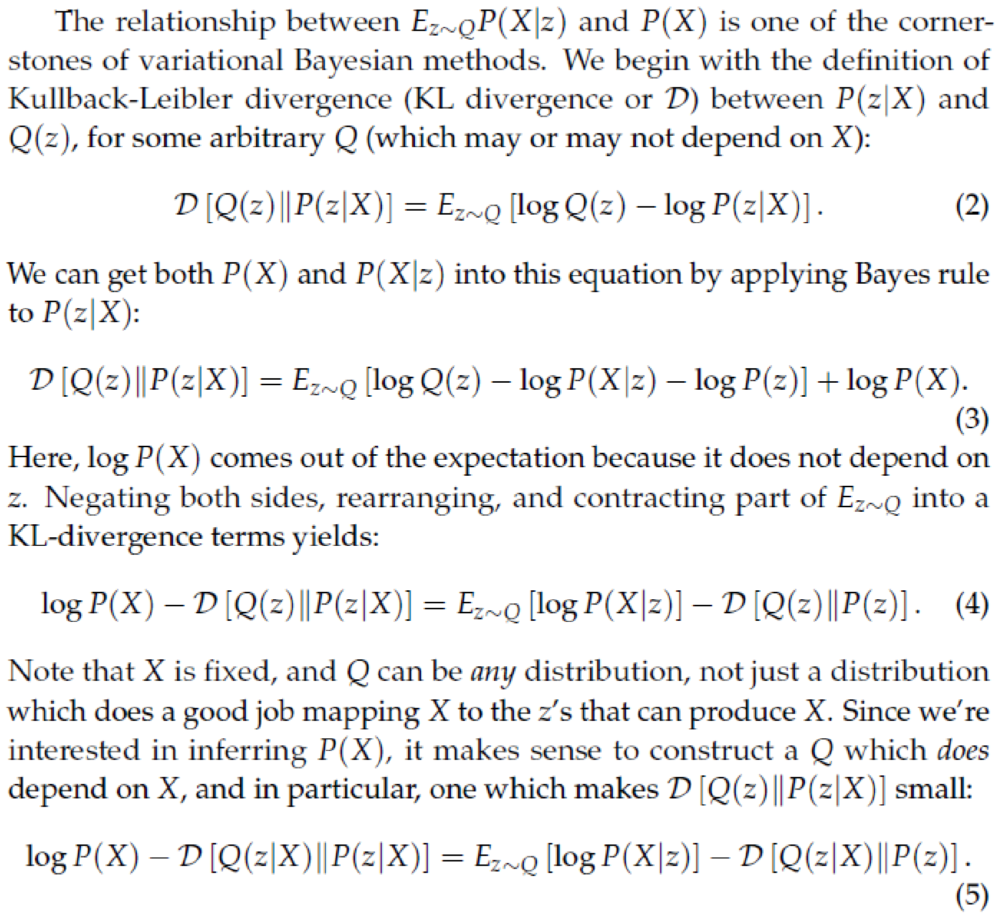
There are two issues with the current equation for maximizing **P(X)**. The first issue is that with the integral, it's computationally very difficult to maximize **P(X)**. The second issue is that the equation doesn't explicitly include the encoder part. It only shows how the encoded information, **z**, is transformed into **P(X)** to approximate the original input, but it doesn't explain how **z** is derived from the original image **X**. Fortunately, solving the first issue also helps solve the second one!

1. Converting the integral to the expectation using **Kullback-Leibler (KL) divergence**: It's clear that sweeping across all possible combinations of **z** to maximize **P(X)** is practically impossible. However, in practice, most values of **z** will make **P(X|z)** equal to 0, contributing nothing to **P(X)** – remember that **P(X)** is the integral of **P(X|z)**. To address this, we introduce an encoder that takes **X** as input and produces a specific set of **z** values. We'll represent this encoder as **Q(z|X)**. The idea is that the **z** values produced by **Q(z|X)** form a much smaller subspace of the full **z** space, which allows us to compute more efficiently:

   $$E_{z \sim Q} P(X|z)$$

   Let me explain this term a bit more. This is fundamentally different from **P(X|z)**. It's an **EXPECTATION** (denoted by **E**) of **P(X|z)**, where **z** is given by **Q(z|X)**. The **EXPECTATION** term is necessary because the **z** generated by **Q(z|X)** is only a subset of the full **z** space. So, calculating **P(X|z)** using this subset of **z** from **Q(z|X)** is not the same as calculating **P(X|z)** from the entire **z** space.
   Now, you might wonder, if **E(P(X|z))**, where **z** is given by **Q(z|X)**, is so different from **P(X|z)**, how is this useful? The answer is that we can relate the **EXPECTATION** term to **P(X|z)** using **Kullback-Leibler divergence**, which we'll refer to as **D** from now on.

2. Using Bayesian Statistics to relate P(X|z) to the expectation value: See the equation development below form the original except:



The relationship between $E_{z\sim Q}P(X|z)$ and $P(X)$ is one of the cornerstones of variational Bayesian methods. We begin with the definition of Kullback-Leibler divergence (KL divergence or $\mathcal{D}$) between $P(z|X)$ and $Q(z)$, for some arbitrary Q (which may or may not depend on X):

$$\mathcal{D}\left[Q(z)\|P(z|X)\right] = E_{z\sim Q}\left[\log Q(z) - \log P(z|X)\right]. \tag{2}$$

We can get both $P(X)$ and $P(X|z)$ into this equation by applying Bayes rule to $P(z|X)$:

$$\mathcal{D}\left[Q(z)\|P(z|X)\right] = E_{z\sim Q}\left[\log Q(z) - \log P(X|z) - \log P(z)\right] + \log P(X). \tag{3}$$

Here, $\log P(X)$ comes out of the expectation because it does not depend on z. Negating both sides, rearranging, and contracting part of $E_{z\sim Q}$ into a KL-divergence terms yields:

$$\log P(X) - \mathcal{D}\left[Q(z)\|P(z|X)\right] = E_{z\sim Q}\left[\log P(X|z)\right] - \mathcal{D}\left[Q(z)\|P(z)\right]. \tag{4}$$

Note that X is fixed, and Q can be *any* distribution, not just a distribution which does a good job mapping X to the z's that can produce X. Since we're interested in inferring $P(X)$, it makes sense to construct a Q which *does* depend on X, and in particular, one which makes $\mathcal{D}\left[Q(z)\|P(z|X)\right]$ small:

$$\log P(X) - \mathcal{D}\left[Q(z|X)\|P(z|X)\right] = E_{z\sim Q}\left[\log P(X|z)\right] - \mathcal{D}\left[Q(z|X)\|P(z)\right]. \tag{5}$$

The authors use an ingenious approach from Bayesian statistics to create a maximization function that bypasses the entire integral issue by introducing **Kullback-Leibler divergence** from equation 5:

$$\mathcal{D}\left[Q(z|X)\|P(z|X)\right]$$

I think this part needs a bit more explanation. Let's first look at **P(z|X)** in the context of **Kullback-Leibler divergence**. The idea is as follows: we have a decoder **P** that can generate images similar to **X** from **z**. If we know **P(z|X)**, which is the distribution of **z** given **X** from the decoder, we can be confident that such a **z** will help generate a high **P(X)** — this is similar to an inverse function. If **P(X|z)** is true, then **z** from **P(z|X)** will maximize **P(X|z)**.

Now, we want to match **Q(z|X)** to **P(z|X)**, which means minimizing the **Kullback-Leibler divergence**. This implies that if **Q(z|X)** is close to **P(z|X)**, then the latent variable **z** sampled from **Q(z|X)** will correspond to values that the decoder **P(X|z)** can use to faithfully reconstruct the original input **X**.

In other words, by maximizing the equation (5), we are maximizing **P(X)**, which is the goal of the variational autoencoder. Additionally, we introduce an encoder **Q(z|X)** that generates **z** so that **z** is no longer a mysterious Gaussian sample. Instead, it represents the mean and variance of a Gaussian distribution generated from the original input **X**.

Mathematical Tricks to Solve the Maximization (optimizing equation 5):

Let us take a look at equation 5, which is the final derivation that we want to maximize.

$$\log P(X) - \mathcal{D}\left[Q(z|X) \| P(z|X)\right] = E_{z \sim Q}\left[\log P(X|z)\right] - \mathcal{D}\left[Q(z|X) \| P(z)\right]. \tag{5}$$

Let us take a look at the right side of the equation, which we need to maximize. First, we have:

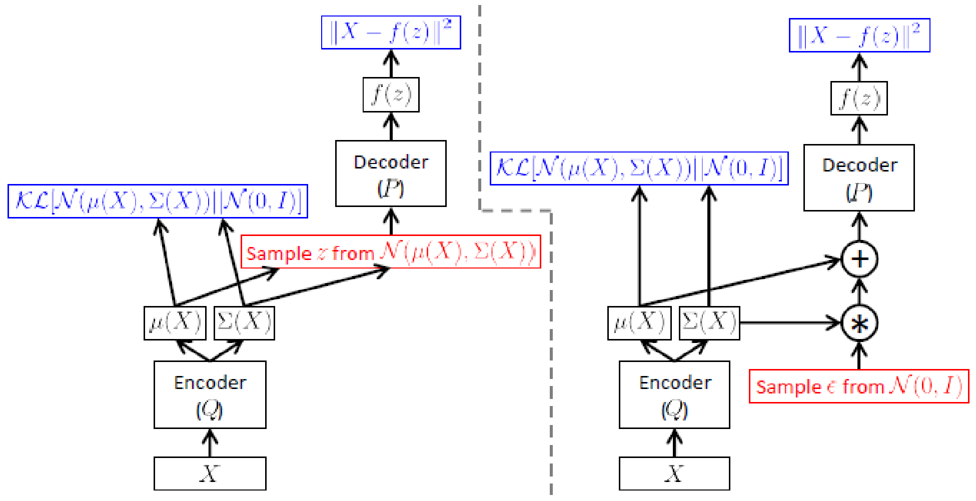
$$E_{z \sim Q}\left[\log P(X|z)\right]$$

To get the expectation value, we could pass many samples of **z** through the decoder, but this would be too computationally expensive. Instead, we use **stochastic gradient descent**, which means we sample **z** to approximate **E[log(P(X|z))]**. However, keep in mind that **z** is derived from **X** through the encoder. So, if we have a dataset **D**, where we sample **X** as input, we end up optimizing the following equation:

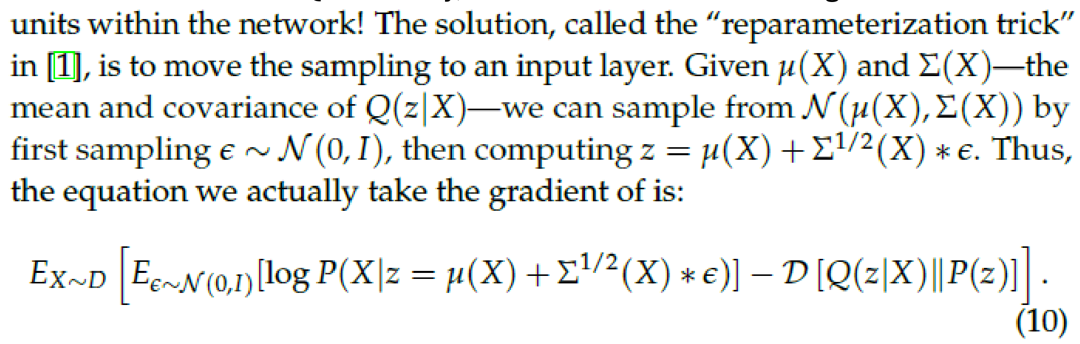
$$\begin{aligned} E_{X \sim D}\left[\log P(X) - \mathcal{D}\left[Q(z|X) \| P(z|X)\right]\right] = \\ E_{X \sim D}\left[E_{z \sim Q}\left[\log P(X|z)\right] - \mathcal{D}\left[Q(z|X) \| P(z)\right]\right]. \end{aligned} \tag{8}$$

Don't be overwhelmed by the symbols. This equation is not much different from equation (5), except that it shows how we use **stochastic gradient descent** and sample **X** from the dataset **D**. This introduces the expectation value, indicated by **E(X~D)**. So, as long as we sample **X** from the dataset **D** (which we need to do for training), we satisfy **E(X~D)**.

Now, we also need to handle **E(z~Q)**. This is important because sampling **z** from **Q(z|X)** can interfere with the gradient flow, making it difficult for the model to learn. Take a look at the image below for further explanation:


$\|X - f(z)\|^2$
$f(z)$
Decoder (P)
$\mathcal{KL}[\mathcal{N}(\mu(X), \Sigma(X))\|\mathcal{N}(0, I)]$
Sample z from $\mathcal{N}(\mu(X), \Sigma(X))$
$\mu(X)$
$\Sigma(X)$
Encoder (Q)
X
$\|X - f(z)\|^2$
$f(z)$
Decoder (P)
$\mathcal{KL}[\mathcal{N}(\mu(X), \Sigma(X))\|\mathcal{N}(0, I)]$
$+$
$*$
$\mu(X)$
$\Sigma(X)$
Sample ϵ from $\mathcal{N}(0, I)$
Encoder (Q)
X

Since you can't perform backpropagation directly when sampling **z** from **Q(z|X)**, we use a technique called the **reparameterization trick**, as shown in the image above. Instead of sampling directly from **Q**, we sample from a Gaussian distribution and simulate sampling from **Q**. Formally, this looks like the following:



units within the network! The solution, called the "reparameterization trick" in [1], is to move the sampling to an input layer. Given $\mu(X)$ and $\Sigma(X)$—the mean and covariance of $Q(z|X)$—we can sample from $\mathcal{N}(\mu(X), \Sigma(X))$ by first sampling $\epsilon \sim \mathcal{N}(0, I)$, then computing $z = \mu(X) + \Sigma^{1/2}(X) * \epsilon$. Thus, the equation we actually take the gradient of is:

$$E_{X\sim D}\left[E_{\epsilon\sim\mathcal{N}(0,I)}[\log P(X|z = \mu(X) + \Sigma^{1/2}(X) * \epsilon)] - \mathcal{D}\left[Q(z|X)\|P(z)\right]\right]. \tag{10}$$

By incorporating this reparameterization trick, we can get rid of E(z~Q) as well and we only need to compute the gradient/loss of the equation below:

$$\log P(X|z) - \mathcal{D}\left[Q(z|X)\|P(z)\right]. \tag{9}$$

Q is the encoder that takes **X** and computes the mean and variance of the Gaussian distribution. But what about **P(z)**? The good news is that we only specified that **P(z)** should be a Gaussian distribution, but we didn't define which exact Gaussian distribution it should be. This means we have the flexibility to choose a distribution that makes the calculations easier. Since the general equation for **Kullback-Leibler divergence** looks like this:

$$\mathcal{D}[\mathcal{N}(\mu_0, \Sigma_0) \| \mathcal{N}(\mu_1, \Sigma_1)] = \frac{1}{2}\left(\mathrm{tr}\left(\Sigma_1^{-1}\Sigma_0\right) + (\mu_1 - \mu_0)^\top \Sigma_1^{-1}(\mu_1 - \mu_0) - k + \log\left(\frac{\det\Sigma_1}{\det\Sigma_0}\right)\right) \tag{6}$$

where k is the dimensionality of the distribution. In our case, this simplifies

However, if we take P(z) to have mean of 0 and variance of 1, then it simplifies to the following:

$$\mathcal{D}[\mathcal{N}(\mu(X), \Sigma(X)) \| \mathcal{N}(0, I)] = \frac{1}{2}\left(\mathrm{tr}\left(\Sigma(X)\right) + (\mu(X))^\top (\mu(X)) - k - \log\det\left(\Sigma(X)\right)\right). \tag{7}$$

During inference:

Setting **P(z)** to have a mean of 0 and variance of 1 offers an additional advantage during the inference stage. Remember, we want to **MINIMIZE** the **Kullback-Leibler** distance between **Q(z|X)** and **P(z)**. We assume that if these two distributions are the same, the encoder will map **X** into a **z** that the decoder can then use to reconstruct the original input. You might recall this idea: *"If we know **P(z|X)**, the distribution of **z** from **X** via the decoder, we can trust that this **z** will help produce a high **P(X)**, similar to an inverse function."*

So, by minimizing the **Kullback-Leibler** divergence, we try to make **Q(z|X)** and **P(z)** as similar as possible. This means that no matter the input to the encoder, the output will always be a Gaussian with mean 0 and variance 1. You might wonder, then, what role the encoder plays during training. The answer is that, during training, the decoder needs to perform stochastic gradient descent when **P(z)** and **Q(z|X)** are not identical. But after training, we assume they are close enough. So, during inference, we can simply feed samples from a standard Gaussian (with 0 mean and 1 variance) into the decoder!

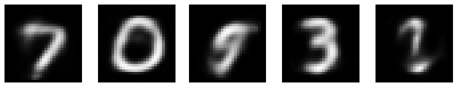
If you want to generate a specific class based on metadata, you can input the metadata (e.g., a sample from a class) and obtain the corresponding **z** values from the encoder. This improves the reconstruction process, and it's known as a **conditional VAE**. This approach is useful for reconstruction tasks going forward.

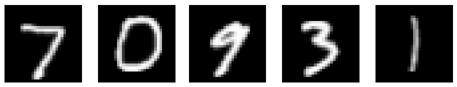
Results and insight into various hyperparameters:

Now it's time to experiment with the hyperparameters! If we fix the neural network (since the network itself isn't considered a hyperparameter) and the training parameters, we're left with two main hyperparameters: **beta**, which controls how much focus you want to put on the latent space representation, and **k**, the number of dimensions in the latent space.

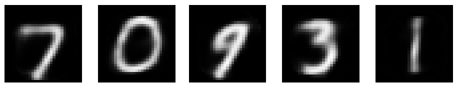
Intuitively, increasing **beta** should shift the focus more onto the latent space, which may lead to worse reconstruction but better generalization. On the other hand, increasing **k** will require more computational power but should improve reconstruction quality.

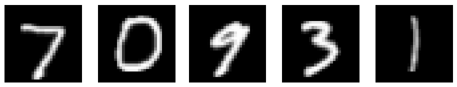
Let's start by looking at **beta = 2.5e-2** and **k = 16**. This means we are slightly suppressing the latent space generalization by a factor of **2.5e-2** and embedding the latent space into a **16-dimensional** space.





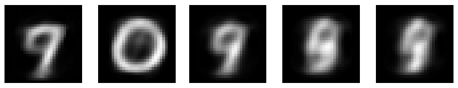
We can see that the given images have been reconstructed pretty well. Now, let us see what happens if we decrease the beta value to 2.5e-6.

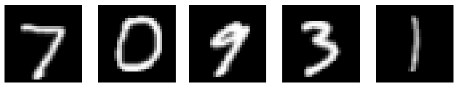




As you can see, when we decrease the **beta** value, the generated image starts to resemble the "meta" image we provided. You might wonder, why not always use a low **beta** value if it seems to reconstruct the meta-image so well? The issue is that with a low **beta**, we're essentially just encoding the image and decoding it without fully leveraging the latent space. Remember, the whole point of a variational autoencoder is to minimize the **Kullback-Leibler** divergence between the encoder and decoder, so that the model has a latent space capable of generalizing the samples.

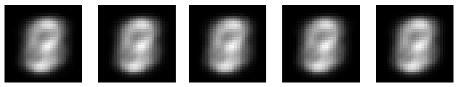
Now, let's see what happens when we reduce **k**. After all, one of our goals is to reduce the dimensionality so we can represent the samples in a smaller space. This example uses **k = 3** and **beta = 2.5e-2**.

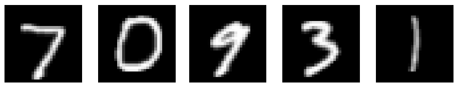




Uh, oh…as we expected, with the dimension reduction, we can no longer discern the difference between a few digits. It started to confuse 7 with 9. Also 3 and 1 has been morphed into "meta" data of latent space.

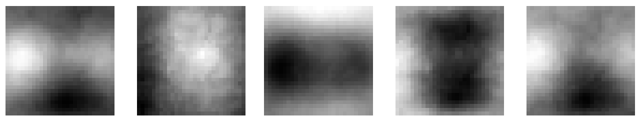
Let us finally take a look at what happens if beta = 1. This means we are taking the equal importance training of the reconstruction and the latent space generalization.





This does not look like much of anything, but actually, this is very significant. We can see that the generated images are mostly the same. This is because we gave much more significance to the generalization instead of reconstruction. Those images represent the latent space images, or meta images themselves. You can see that those images could represent anything. 0, 1, 9, 3….

Now, let us try this on a more complex dataset, CIFAR10. I put k = 16, and beta = 1e-4 since CIFAR10 is much more complex and needs to have more weighting on reconstruction.



Well, this is a big failure. We can see that the generated image vaguely captures the lighting aspect, but it fails to create the sharp and crisp image of CIFAR10. What happens if we increase the beta value to 5e-1?

As we expected, the images again converged to meta-images.

Conclusion:

The **Variational Autoencoder (VAE)** is a powerful generative model that combines deep learning and probabilistic modeling to learn complex data distributions. It consists of two main components: an **encoder**, which maps input data to a latent space, and a **decoder**, which reconstructs the data from the latent variables. The VAE introduces a probabilistic twist to the standard autoencoder architecture by assuming the latent variables follow a certain distribution (typically Gaussian). This allows the model to generate new data by sampling from the latent space, making it a powerful tool for generative tasks.

One of the key innovations of VAE is the use of **variational inference** to approximate the true posterior distribution of the latent variables. This is done by minimizing the **evidence lower bound (ELBO)**, which balances between reconstruction accuracy and the divergence between the approximate posterior and the prior distribution. The VAE has been successful in generating high-quality images, such as with MNIST, and has been extended to more complex tasks, including speech synthesis and protein structure prediction.

Overall, VAE offers a flexible and interpretable framework for generative modeling. Its ability to learn a smooth and continuous latent space makes it an important tool in deep learning and probabilistic modeling, paving the way for more advanced models like VQ-VAE and conditional VAEs.