Codebook-Based Learning for Enhancing the Power of Generative Networks

Avinash Kak

Purdue University

Lecture Notes on Deep Learning by Avi Kak and Charles Bouman

Monday 5th May, 2025 10:35

Preamble

To TOC

For a given dataset of images, a codebook is a collection of vectors of real numbers that are learned in such a way that subsequently you can represent any image drawn from the distribution distribution that characterizes the dataset by a spatial arrangement of the codebook vectors.

You might ask: What exactly do we mean by "the spatial arrangement of the codebook vectors" and by "can be represented by" in the above statement.

In answer, think of a typical Encoder of an Encoder-Decoder architecture for a generative Variational Autoencoder. The Encoder could represent the $256 \times 256 \times 3$ input images by, say, $16 \times 16 \times 128$ at its output. With codebook learning in place, you would replace each of the 128-dimensional "pixels" at the output of the Encoder with its closest codebook vector of the same dimensionality.

You might ask: "But why?".

It has been shown that when you replace the pixels (meaning the C-dimensional vectors if C is dimensionality of the channel axis at the Encoder output), you obtain a superior "sequence-based representation" of an image that lends itself well to transformer based learning for downstream tasks.

Preamble (contd.)

This lecture will start with a gentle introduction to what exactly is meant by a codebook and codebook vectors in our context.

For a deeper understanding, because variational autoencoders have played a huge role in the development of codebook based methods, I will launch into a brief presentation on the topic of VAEs.

Subsequently, I will focus on VQ-VAE (also written as VQVAE) since that it was the VQ-VAE paper was the first to propose this approach to enhance the power of generative networks.

Finally, I'll talk about VQGAN that has taken the codebook based generative networks to the next level by adding transformer-based autoregressive modeling to the codebook vectors used as sequence elements.

Outline

Outline

Codebooks are Generally Used in Encoder-Decoder Architectures

- The idea of a codebook is to come up with a discrete vocabulary of embedding vectors — that remain fixed after they are learned — for representing images in Encoder-Decoder architectures.
- From a theoretical perspective at least, there exists a probability distribution that describes any given training dataset of images. You could think of the actual images in the dataset as samples drawn from that probability distribution. The codebook vectors, after they are learned, would represent all images drawn from that probability distribution.
- A direct analogy here would be with the embedding vectors that are learned for the individual tokens in a text processing system. After the embedding vectors are learned, for any subsequent neural-network processing of the text, you would use the embedding vectors as proxies for the actual tokens.

Codebooks for Encoder-Decoder Architectures (contd.)

- If you really think about what I have said in the last bullet on the previous slide, the analogy between codebook vectors for images and the embedding vectors for text has some gaping holes in it.
- Text, by definition, is a sequence of tokens and when each token is represented by the corresponding embedding vector, you end up with a sequence of embedding vectors. In that sense, the organization of the original units of information — meaning the tokens — does not change when you replace each token with its embedding vector.
- An image, on the other hand, is NOT fundamentally sequential. So what do I mean when I say let's represent an image by the learned codebook vectors?
- This is where we get into Encoder-Decoder architectures.

Codebooks for Encoder-Decoder Architectures (contd.)

- Continuing with the last bullet on the previous slide, in a typical such architecture, your input image would be of shape, say, $256 \times 256 \times 3$ and the output of the Encoder of shape, say, $16 \times 16 \times 128$. You can construe this transformation as saying that you are representing the input image with 256 128-dimensional vectors.
- If you use small-sized convo kernels (as is very likely to be case in practice) to transform 256×256 images into 16×16 arrays at the output of the Encoder, you can think of each element of the output array as representing a patch — of size 16×16 in this example — of the input image.
- You could re-arrange the elements in the 2D array at the Encoder output into a single "linearized" sequence by starting at the top row, following that with the next row, and then the next row, etc., until you end up with a 256 element sequence of vectors that represent the input image.

Codebooks for Encoder-Decoder Architectures (contd.)

- The idea of a codebook is to replace each vector in the above sequence with the closest codebook vector.
- Your first reaction to the above bullet is likely to be: **But why?**
- The answer to this question can be gleaned from the results presented by the authors who first formulated the concept of a codebook for improving the performance of VAEs (Variational Auto Encoder):

  `https://arxiv.org/pdf/1711.00937`

  and the authors who subsequently used the codebook to produce absolutely large and stunning images in a transformer-based generative framework:

  `https://arxiv.org/pdf/2012.09841`

- The first of these two papers is now known as the "VQVAE" paper the second as the "VQGAN" paper. I will formulate my answer to the question posed in the second bullet above in the context of the second paper.

Codebooks for Encoder-Decoder Architectures (contd.)

- In order to answer the question posed on the previous slide, let's go back to our example in which we have 256 vectors, each of dimension 128, at the output of the Encoder. We can join all the rows of the output array into a single 256-element sequence, with each element of the sequence being a 128-element embedding vector.
- The "linearized" sequence formed as described above is no different from a sequence of tokens in text, with each element in the sequence being represented by a 128-element embedding vector — except for the difference stated in the first bullet on the next slide.
- **While a text token, after all the learning has taken place, will always be represented by exactly the same embedding vector, for the case of images, a patch-based embedding vector will vary depending on the noise perturbations in the patch when we present different instances of what appear to be the same image at the input to the neural network.**

Codebooks for Encoder-Decoder Architectures (contd.)

- Such noise induced variations in the embedding vectors would obviously corrupt the calculation of the pairwise dot-products $Q \cdot K^T$ in (Q, K, V) based attention.
- In patch-based calculations for the case of images, when you replace the patch embedding vectors with their nearest codebook vectors, you are in essence protecting the the $Q \cdot K^T$ calculations from the ever-present noise (even when it is not visible to the naked eye) in images.
- **In the rest of this section**, I'll focus on how to theoretically represent the process of replacing each element of the array at the output of the Encoder with the embedding vectors drawn from the Codebook.
- The figure shown in the next slide is from the VQVAE paper. As mentioned previously, they were the first to come up with the idea of learning a codebook for discretized representations of embedding vectors.

Codebooks for Encoder-Decoder Architectures (contd.)

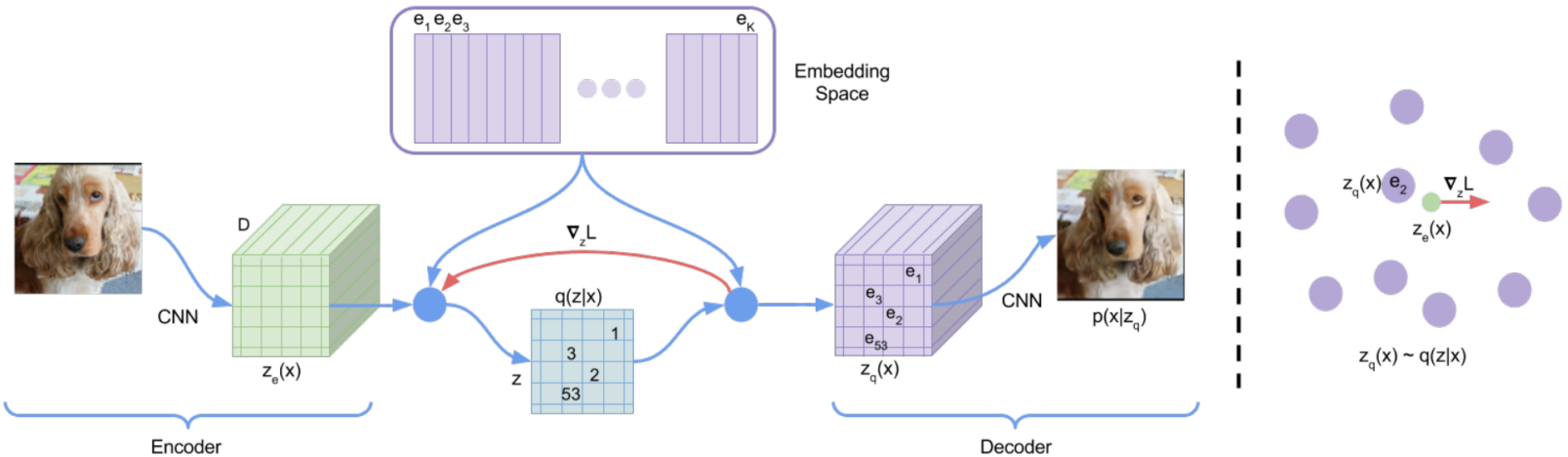


Figure: This figure is from the famous VQVAE paper `https://arxiv.org/pdf/1711.00937`

Codebooks for Encoder-Decoder Architectures (contd.)

- In the figure on the previous slide, the vectors $\mathbf{e}_i$ inside the box labeled "Embedding Space" are the codebook vectors that are learned during training. The symbol "CNN" at left stands for the convolutional Encoder whose output is represented by the vectors $z_\theta(x)$ where x is the input image and θ the learnable parameters in the Encoder.
- As for what exactly is denoted by z_θ, let's go back to the example in which the input images are of shape $256 \times 256 \times 3$ and the output of the Encoder of shape $16 \times 16 \times 128$. We represent each 256-element vector in the output 16×16 array by $z_\theta(x)$. What's more, we think of each 256-element vector as an embedding for that element in the output 16×16 array.
- You can think of the invisible face (from the left) of the left datacube in the figure as representing the 16×16 array of the Encoder, the horizontal axis of that cube can then be thought as representing the embedding dimension for the elements of the array output by the Encoder. In our example, the size of the embedding dimension is 128.

Codebooks for Encoder-Decoder Architectures (contd.)

- Our goal is to replace each $z_\theta(x)$ vector in the left datacube by its closest codebook vector $\mathbf{e}_i$, with the result being shown in the right datacube. The face of the right datacube you directly see has exactly the same shape as the output of the Encoder — it will be a 16×16 array for our example. However, along the embedding axis of the datacube, you'll see the codebook vectors e_i for different indexes i.

- The label $z_\theta(x)$ that was under the left datacube becomes $z_q(x)$ under the right datacube where the subscript q stands for "quantized" since the codebook vectors are quantized versions of the output of the Encoder.

- The integers 1, 3, 2, 53, etc., you see in the bottom square in the middle are the index values i for the codebook vectors e_i that replace each of the elements in the 16×16 array at the output of the Encoder.

Codebooks for Encoder-Decoder Architectures (contd.)

- With regard to the code vectors in the right datacube, we express their posterior distribution by $q(z|x)$ where x is the input image and the symbol q stands for "quantized". We can write the following equation for the posterior:

$$q\Big(z = k \;\Big|\; x\Big) \;=\; \begin{cases} 1 & \text{for } k \;=\; \text{argmin}_j \; ||z_{\text{enc}}(x) - e_j||_2 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

  where z_{enc} stands for the output of the Encoder for input image x and $||.||_2$ indicates the L_2 norm.

- Now we can express the codebook vectors that are in the right datacube in the figure by $z_q(x) = k$ where $k = \text{argmin}_j \; ||z_{\text{enc}}(x) - e_j||_2$.

- In the rest of this lecture, I'll start by introducing you to VAE (Variational Autoencoder). As you will see, that is foundational to understanding both the VQVAE and VQGAN architectures for codebook based learning.

Outline

The Latent Space for Generative Modeling

- Foundational to the idea of a VAE is existence of a simple Latent Space — simple in the sense that the structure of this space would be learned with the desirable goal of its being representable as a low-dimensional zero-mean, unit-variance Gaussian — that would capture the essence of your training dataset.
- In an Encoder-Decoder framework, while the Encoder network is learning the Latent Space, the Decoder is learning how to add complexity to the samples drawn from the Latent space to that the output of the Decoder looks like it came from the same distribution as the training images.
- VAEs allow you to do what's known as "**disentanglement learning**" that can be put to use to control what the images look like at the *output* of the VAE.

[**Disentanglement learning in this context means that the latent distribution learned from the training data represents the "essence" of what's in each input image and, after this "essence" has been learned, you can have the Decoder generate from it useful variants of the input images. The meaning of "essence" would be dictated by the application.**]

The Latent Space for Generative Modeling (contd.)

- The figure in the next slide illustrates succinctly the role played by the Latent Space in a VAE.

  [If this otherwise highly educational figure has any "defect" at all, it lies in using a perfect circle for the Latent Space. Visually, it creates the impression that a Latent Space is always a "perfect" zero-mean, unit-variance Gaussian. However, as mentioned in the note at the bottom of the previous slide, the Latent Space is a standard Gaussian only to the extent allowed by the minimization of the losses in the overall learning framework.]

- In the figure, the dataset distribution shown at the bottom is complex and represented by $p(x)$. The z-space in the figure is for the Latent Distribution, with the latent distribution represented by the priors $p(z)$ over the latent vectors z. I'll use $p_\theta(z|x)$ to denote the posteriors over the latent vectors where θ are the learnable parameters in the Generator (the same thing as the Decoder).

- The main goal of generative modeling with a VAE is to learn the latent distribution $p_\theta(z)$ and the associated Decoder so that we can generate images with the desired properties.

The Latent Space for Generative Modeling (contd.)

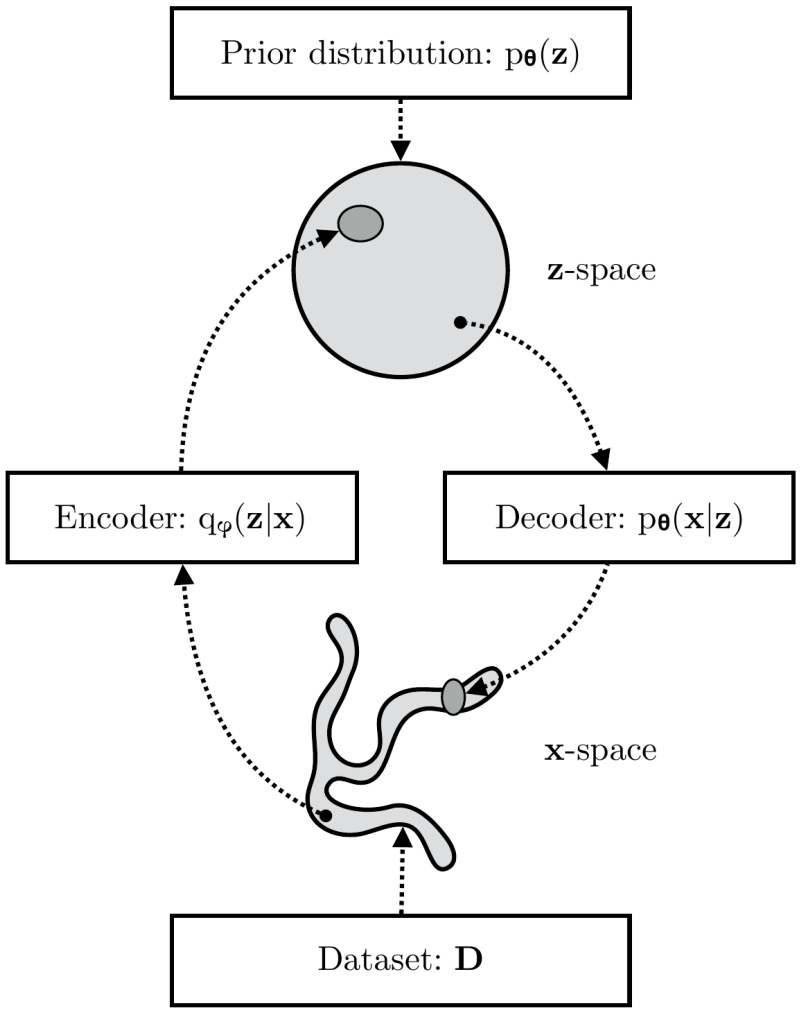


Figure: This figure is from "An Introduction to Variational Autoencoders" by D. P. Kingma and M. Welling, 2019.

The Latent Space for Generative Modeling (contd.)

- **For obvious reasons any learning of the Encoder-Decoder parameters for variational modeling must be self-supervised.** That is, it must not require any human-generated annotations. Think about it: What sort of annotations could a human possibly provide so that the overall framework would learn both the latent space and also how to generate data samples from the latent vectors.
- I'll now quickly review the Bayesian logic that is used for training a VAE. For a more detailed presentation of the logic, please see my tutorial at

  `https://engineering.purdue.edu/kak/Tutorials/OptimalSubspaces.pdf`

- The Bayesian logic uses the facts that the Generator is capable of directly learning $p_\theta(x|z)$ and, from that using Bayes' Rule, we can infer the conditional $p_\theta(z|x)$. And the fact that the Encoder can directly learn the conditional $q_\phi(z|x)$ and, from that using Bayes's Rule, we can infer the conditional $q_\phi(x|z)$.

Deriving the Loss Function for Training a VAE

- We should expect that the posterior $p_\theta(z|x)$, which we estimated using Bayes' Rule from the generative $p_\theta(x|z)$ in the Decoder, would not be too different from $q_\phi(z|x)$ as learned by the Encoder.

- In other words, our learnable parameters θ and ϕ should be such that the divergence between the inferred posterior distribution $p_\theta(z|x)$ and the Encoder-learned $q_\phi(z|x)$ should be as small as possible. As it turns out, we have a great criterion, KL-Divergence, to compare the two with the help of this formula:

$$d_{KL}\Big(q_\phi(z|x) \;||\; p_\theta(z|x)\Big) \;=\; -\int q_\phi(z|x)\cdot \log \frac{p_\theta(z|x)}{q_\phi(z|x)}\, dz \tag{2}$$

[See my Week 11 Lecture at Purdue's Deep Learning website for further info regarding KL-Divergence between two probability distributions.]

- Bayes' Rule tells us that $p_\theta(z|x) = \dfrac{p_\theta(x|z)p_\theta(z)}{p_\theta(x)}$. Substituting this in the expression for the KL-Divergence, we get

$$d_{KL}\Big(q_\phi(z|x) \;||\; p_\theta(z|x)\Big) \;=\; -\int q_\phi(z|x)\cdot \log \left(\frac{p_\theta(x|z)\cdot p_\theta(z)}{q_\phi(z|x)}\right) dz \;+\; \log p_\theta(x) \tag{3}$$

The Loss Function for Training a VAE

- Applying the condition that always $d_{KL} \geq 0$ to the formula shown at the bottom of the previous slide, **we arrive at a lower bound for the log-likelihood** $\log p_\theta(x)$:

$$\log p_\theta(x) \quad \geq \quad - d_{KL}\Big(q_\phi(z|x) \;||\; p_\theta(z)\Big) \quad + \quad \int q_\phi(z|x) \cdot \log p_\theta(x|z)\; dz \tag{4}$$

- The RHS again has the KL-Divergence in it, but this time time it is the divergence between the Encoder-learned $q_\phi(z|x)$ and the prior $p_\theta(z)$ for the encodings z.

- **This allows us to bring the following consideration into the learning of the parameters** (ϕ, θ)**:** Let's say we assume a standard normal (zero-mean, unit variance) distribution for the prior $p_\theta(z)$. The minimization of the KL-divergence shown above would ensure that the encodings z for the different inputs, x, are all "maximally" randomized in a similar fashion.

- The right-hand side of the inequality shown above has a name unto itself. It is called **ELBO (Evidence Lower BOund)**.

The Loss Function for Training a VAE (contd.)

- The second term on the RHS in Eq. (3) is in the form of a average — an expectation — for the conditional log-likelihood $\log p_\theta(x|z)$ of the images generated by the Decoder with respect to the Encoder-learned $q_\phi(z|x)$. Using the expectation operator $\mathbb{E}$, we can therefore say

$$\log p_\theta(x) \;\geq\; -\, d_{KL}\Big(q_\phi(z|x) \;||\; p_\theta(z)\Big) \;+\; \mathbb{E}_{\sim q_\phi(z|x)} \log p_\theta(x|z) \tag{5}$$

- This gives us a lower bound on the log-likelihood of the generated image x. **You can think of this as a viability score for the output x — that is, given the current values for the parameters (θ, ϕ) of the neural network, how likely is it (in a probabilistic sense) that the network would be able to generate that x from the hidden vector z that was learned from the input.**

- In an iterative learning framework, it would obviously be in our interest the update the values of the parameters (θ, ϕ) that increases the value of the lower bound shown above **in order to enhance the value of the log-likelihood** $\log p_\theta(x)$.

The Loss Function for Training a VAE (contd.)

- What's on the previous slide implies that, in terms of the language of neural-network learning, we should set the loss $\mathcal{L}oss$ to the negative of what you on the RHS in Eq. (4):

$$\mathcal{L}oss \;=\; d_{KL}\Big(q_\phi(z|x) \;||\; p_\theta(z)\Big) \;-\; \mathbb{E}_{\sim q_\phi(z|x)} \log p_\theta(x|z) \tag{6}$$

- In the rest of this section, I'll treat these two parts of the overall loss separately:

$$\text{loss\_kld} \;=\; d_{KL}\Big(q_\phi(z|x) \;||\; p_\theta(z)\Big) \tag{7}$$

$$\text{loss\_reconstruction} \;=\; \mathbb{E}_{\sim q_\phi(z|x)} \log p_\theta(x|z) \tag{8}$$

  As to why the second part of the loss is called "reconstruction loss" will be clear shortly.

- The expression for the KL-Divergence loss can be further simplified under the Gaussian assumption for the prior $p_\theta(z)$ and also for the Encoder-learned $q_\phi(z|x)$.

The Loss Function for Training a VAE (contd.)

- The Wikipedia page on "Multivariate Normal Distribution" presents the following formula for the KL-Divergence from the "true" $\mathcal{N}_q(\mu_q, \Sigma_q)$ for the Encoder to the learned $\mathcal{N}_p(\mu_p, \Sigma_p)$ for the Decoder, where d is the dimensionality of the z encodings and $|.|$ represents the determinants:

$$d_{KL}\Big(\mathcal{N}_q \;||\; \mathcal{N}_p\Big) \;=\; \frac{1}{2}\Big\{ tr\Big(\Sigma_q^{-1}\Sigma_p\Big) \;+\; (\mu_q - \mu_p)^T \Sigma_q^{-1}(\mu_q - \mu_p) \;-\; d \;+\; \log \frac{|\Sigma_q|}{|\Sigma_p|} \Big\} \tag{9}$$

- We now assume that both the Generator and the Encoder processes are isotropic:

$$\begin{aligned} \Sigma_p &= \sigma_p^2 \mathcal{I} \\ \Sigma_q &= \sigma_q^2 \mathcal{I} \end{aligned} \tag{10}$$

- With the Gaussian assumptions made above, the KL divergence can be expressed more simply as:

$$d_{KL}\Big(\mathcal{N}_q \;||\; \mathcal{N}_p\Big) \;=\; \frac{1}{2}\Big\{ \frac{\sigma_p^2}{\sigma_q^2} \;+\; \frac{(\mu_q - \mu_p)^2}{\sigma_q^2} \;-\; d \;+\; \log \frac{\sigma_q^2}{\sigma_p^2} \Big\} \tag{11}$$

The Loss Function for Training a VAE (contd.)

- Since the Wikipedia expression $d_{KL}\Big(\mathcal{N}_q \;||\; \mathcal{N}_p\Big)$ is the same as $d_{KL}\Big(q_\phi(z|x) \;||\; p_\theta(z)\Big)$ in our case, we can write the following form for the KL-divergence part of the loss:

$$d_{KL}\Big(q_\phi(z|x) \;||\; p_\theta(z)\Big) \;=\; \frac{1}{2}\Big\{\frac{\sigma_p^2}{\sigma_q^2} \;+\; \frac{(\mu_q - \mu_p)^2}{\sigma_q^2} \;-\; d \;+\; \log\frac{\sigma_q^2}{\sigma_p^2}\Big\} \tag{12}$$

- In the VAE literature, however, you are more likely to see the following formula that is derived from two *univariate* Gaussian distributions, that is, when $d = 1$ [**Substituting $d = 1$ in Eq. (12) yields the same result as you see below in Eq. (13)**]:

$$d_{KL}\Big(q_\phi(z|x) \;||\; p_\theta(z)\Big) \;=\; -\frac{1}{2} \;+\; \log\frac{\sigma_q}{\sigma_p} \;+\; \frac{\sigma_p^2 + (\mu_p - \mu_q)^2}{2\sigma_q^2} \tag{13}$$

- It is generally assumed that the prior for the z latent vectors is a standard normal. That is, $\mu_p = 0$ and $\sigma_p^2 = 1$. With these assumptions:

$$d_{KL}\Big(q_\phi(z|x) \;||\; p_\theta(z)\Big) \;=\; -\frac{1}{2} \;+\; \frac{1}{2}\log\sigma_q^2 \;+\; \frac{1 + \mu_q^2}{2\sigma_q^2} \tag{14}$$

The Loss Function for Training a VAE (contd.)

- **In the Pytorch implementations of VAE, the Encoder neural network ends in two linear layers, one for predicting $\log \sigma_q^2$ and the other for predicting μ_q.** The $\log \sigma_q^2$ predictor is represented by the variable `log_var` and the μ_q predictor by the variable `mu`. **In terms of these two variables, the following statement for computing the $d_{KL}\Big(q_\phi(z|x) \;||\; p_\theta(z)\Big)$ part of the loss shown in Eq. (7) has now become ubiquitous**:

```
loss_kld = -0.5 * torch.sum( 1 + log_var - mu.pow(2)  -  torch.exp(log_var) )
```

- I will now take up the computation of the reconstruction loss part of the overall loss as shown in Eq. (8):

$$\text{loss\_reconstruction} \;=\; \mathbb{E}_{\sim q_\phi(z|x)} \log p_\theta(x|z) \tag{15}$$

- To make the calculation of this part of the loss computationally feasible, we go back to the Gaussian assumption and assume:

$$\begin{aligned} p_\theta(x|z) \;&=\; \mathcal{N}(\mu_{x|z}, \Sigma_{x|z}) \\ &=\; \frac{1}{\sqrt{(2\pi)^d |\Sigma_{x|z}|}} e^{-\frac{1}{2}(x-\mu_{x|z})^T \Sigma_{x|z}^{-1}(x-\mu_{x|z})} \end{aligned} \tag{16}$$

where $|\Sigma|$ denotes the determinant of the covariance matrix.

The Loss Function for Training a VAE (contd.)

- If we further assume that the multivariate Gaussian is isotropic with a constant variance, we can write

$$p_\theta(x|z) = \frac{1}{\sqrt{(2\pi\sigma_{x|z})^d}} \cdot e^{-\frac{1}{2}\left(\frac{(x-\mu_{x|z})^T(x-\mu_{x|z})}{\sigma^d_{x|z}}\right)} \tag{17}$$

- Taking the logarithm of both sides, we get

$$\log p_\theta(x|z) = \log\left(\frac{1}{\sqrt{(2\pi\sigma_{x|z})^d}}\right) - \frac{(x-\mu_{x|z})^T(x-\mu_{x|z})}{2\sigma^d_{x|z}} \tag{18}$$

- Ignoring the constant terms, we can write for the loss expression in Eq. (15):

$$\mathbb{E}_{\sim q_\phi(z|x)} \log p_\theta(x|z) \approx - \mathbb{E}_{\sim q_\phi(z|x)} (x-\mu_{x|z})^T(x-\mu_{x|z}) \tag{19}$$

- The question that arises next is: What to use for $\mu_{x|z}$? To that end, we assume that, probabilistically speaking, the goal of variational autoencoding is to generate samples from the same distribution that describes the image that is at the input to the Encoder.

The Loss Function for Training a VAE (contd.)

- Continuing with the thought at the bottom of the previous slide, we assume that the distribution itself can be assumed to be normal around that input sample, it makes sense to use the input itself for $\mu_{x|z}$. So if we denote the input as x_{in}, we can write the following for the reconstruction loss:

$$\mathbb{E}_{\sim q_{\phi}(z|x)} \log p_{\theta}(x|z) \quad \approx \quad - \quad \mathbb{E}_{\sim q_{\phi}(z|x)} (x - x_{in})^T (x - x_{in}) \tag{20}$$

- In actual coding, the expectation would be implemented by averaging the square of the difference between the input and the output over the images in a batch. As you know, that's exactly what's accomplished by the Pytorch loss function `torch.nn.MSELoss`. **You are therefore likely to see the following sort of a statement in practically all VAE implementations**:

```
loss_recon = nn.MSELoss( reduction='sum' )( input_images, decoder_output_images )
```

[**Note the invocation of `reduction='sum'` for estimating the MSELoss. The theory only calls for averaging over the batch, and not summing the error magnitudes. The summing results in enhancing by a scale factor the reconstruction loss vis-a-vis the loss component related to the KL-Divergence in Eqs. (6), (7), and (8).**]

Outline

Implementing a VAE

- Now that we have the expressions for Loss, as represented by its KL and Reconstruction components in Eqs. (14) and (20), respectively, what sort of a neural architecture it would take to learn using those?
- Going forward in this explanation, I'll use x_{in} to denote a specific image at the input to the Encoder.
- The question that now arises is: **What exactly should we ask the Encoder to produce at its output?**
- Does it make sense to have the Encoder output a *specific* latent vector z for each input image x_{in}? **It does NOT!**
- The Encoder is supposed to help us learn the latent space for a given training dataset. In other words, the Encoder should eventually learn a probability distribution $q_\phi(z|x)$ that comes as close as possible to $\mathcal{N}(0, I)$. That is, the Encoder should seek to learn the two parameters of a normal distribution $\mathcal{N}(\mu, \sigma^2 I)$ with the goal that μ will be as close to zero as possible and σ^2 as close to 1 as possible.

Implementing a VAE

- **What that implies is that, instead of outputting a specific z for a specific input x_{in}, the Encoder should be outputting updates to the parameters μ and σ^2.**
- **That then tells us how we should configure the output layer of the Encoder.**
- While having the Encoder output estimates for μ and σ^2 distribution parameters sounds doable enough, **do we know what to feed into the Decoder so that we can calculate the Loss at output of the Decoder?**
- In keeping with the fact that the Decoder is in reality a Generator that is supposed to transform the latent z vectors sampled from the latent space into images that bear a pre-specified relationship to the images in our dataset, in principle the input to the Decoder should be as depicted in the figure on the next slide.

Implementing a VAE (contd.)

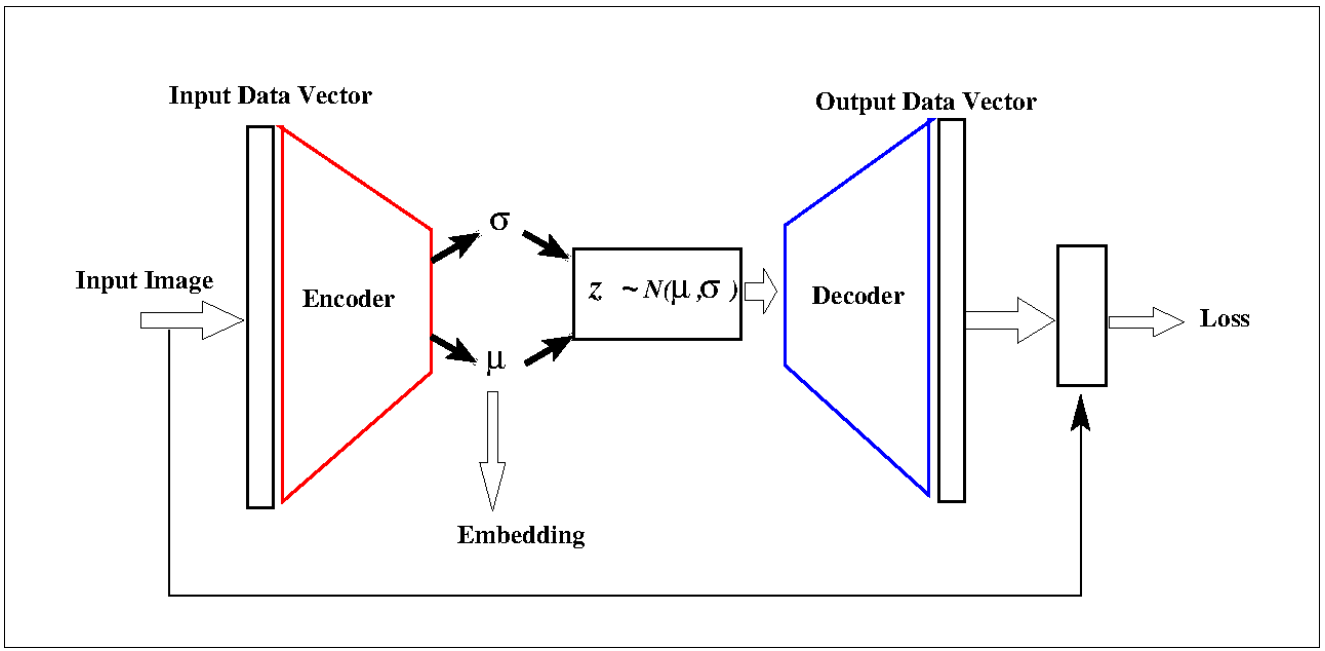


Figure: Variational Autoencoder — the main idea

Implementing a VAE (contd.)

- Unfortunately, drawing a random sample from a distribution is not a differentiable step — **differentiability being a fundamental requirement for any neural learning as explained in the next subsection**. Said another way, there is no way to backpropagate the loss through the computational step $z \sim \mathcal{N}(\mu, \sigma)$.
- This implementation difficulty is resolved by using the following formula to generate a latent vector z for feeding into the Decoder in each iteration of training:

$$z = \mu + \sigma * \epsilon \tag{21}$$

  where $\epsilon \sim \mathcal{N}(0, 1)$ as shown in the figure on the next slide.
- Note that the parameter ϵ in Eq. (21) is NOT meant to be a learnable parameter — it cannot be because of the reasons already stated. Drawing a sample from a distribution, as represented by $z \sim \mathcal{N}(\mu, \sigma)$ or $\epsilon \sim \mathcal{N}(0, 1)$, is **NOT** a differentiable step and therefore the loss cannot be backpropagated through it.

Implementing a VAE (contd.)

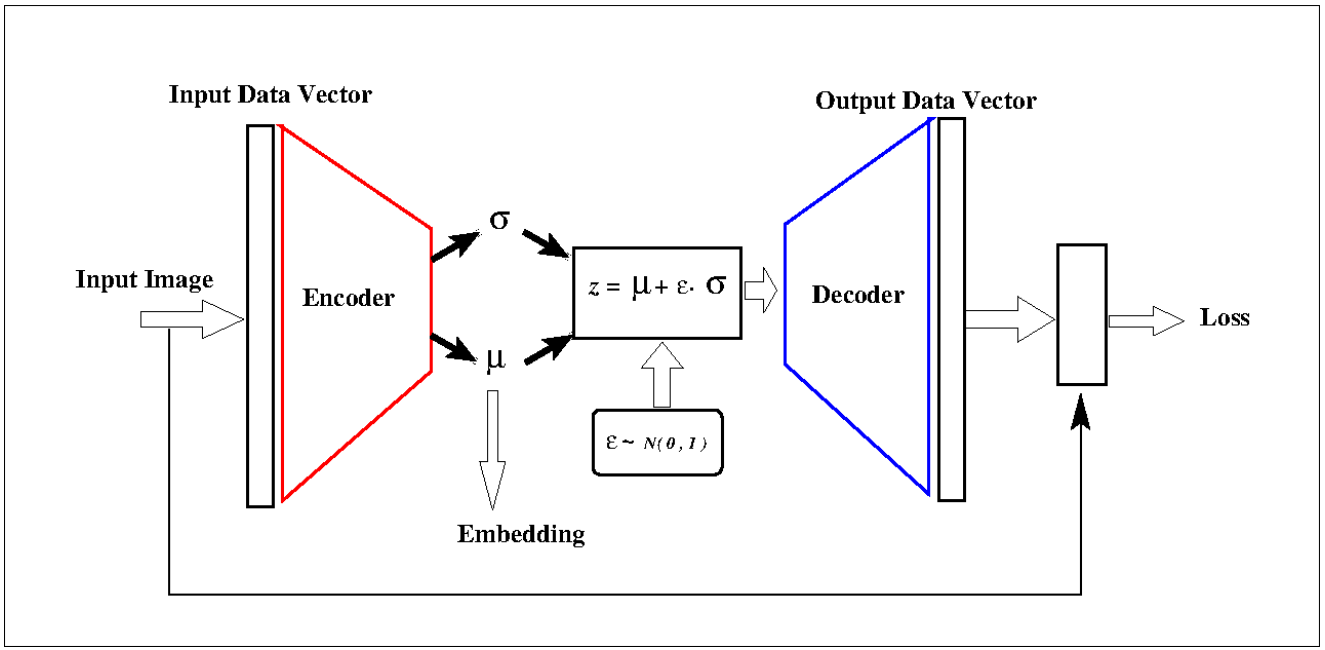


Figure: Variational Autoencoder — What's actually implemented

Implementing a VAE (contd.)

- However, the calculation of the latent vector z in Eq. (21) *is* differentiable — despite the fact that it includes a non-learnable parameter ϵ.
- Therefore, for any input x, the job of learning is to figure out the best values to use for (μ, σ) for that input. **The role of ϵ is to give a differentiable degree of freedom to the Decoder when it seeks to generate an output image from the latent vector z sampled from a normal distribution based on the parameters μ and σ^2 learned so far.**
- **The z-vector sampling shown in Eq. (21) is frequently referred to as the "reparameterization trick" and typically implemented by a function whose name is *reparameterize*().**

Outline

Autoencoder Class in DLStudio

- All autoencoding classes in DLStudio, including the class VAE, are derived from the parent class `Autoencoder`. So, in order to understand how I have implemented the `VAE` class, you must first come to terms with the `Autoencoder` class.
- The `Autoencoder` class does its job through its two inner classes, `EncoderForAutoenc` and `DecoderForAutoenc`, the first that serves as the encoder and the second as the decoder.
- An `Autoencoder` is the simplest of such classes in DLStudio because all it does is to feed the output of `EncoderForAutoenc` into the `DecoderForAutoenc`. The Encoder's job is to convert the input tensor, whose shape is (B, C_{in}, H, W), into an output tensor of shape (B, C_{out}, h, w) with $h \leq H$, $w \leq W$ and $C_{out} \geq C_{in}$.

Inner Class `EncoderForAutoenc` of the Autoencoder Class

- Experience has shown that in autoencoding, in general, you need to pack punch in the Encoder while the Decoder can be relatively simple.
- To that end, the Encoder class shown on the next slide gives you a constructor parameter called `num_repeats` that you can use to construct as deep a network as you want subject, obviously, to the GPU memory that is available.
- As to how `num_repeats` is used in the encoder depends on the value of `self.depth` that is calcuated in Line (A). Assuming square images, it is the ratio of the height of the input image to the height wanted at the output of the Encoder.
  [**Autoencodes typically reduce the image size by a power of 2. While the input images are likely to be of shape** $(64 \times 64 \times 3)$, $(128 \times 128 \times 3)$, $(256 \times 256 \times 3)$, **etc., the Encoder output is likely to be of shape** $(8 \times 8 \times 64)$, $(8 \times 8 \times 128)$, $(16 \times 16 \times 512)$, **etc.**]
- In the statements in Lines (B) through (G) on the next slide, for each downsampling of the input image, we repeat a `SkipBlock` structure a number of times equal to `num_repeats`.

Inner Class EncoderForAutoenc of Autoencoder

```
class EncoderForAutoenc(nn.Module):
    """
    The two main components of an Autoencoder are the encoder and the decoder. This is the encoder part of the
     Autoencoder.

    The parameter num_repeats is how many times you want to repeat in the Encoder the SkipBlock that has the same
    number of channels at the input and the output.

    Class Path:   DLStudio  ->  Autoencoder  ->  EncoderForAutoenc
    """
    def __init__(self, dl_studio, encoder_in_im_size, encoder_out_im_size, encoder_out_ch, num_repeats, skip_connections=True):
        super(DLStudio.Autoencoder.EncoderForAutoenc, self).__init__()
        downsampling_ratio =  encoder_in_im_size[0] // encoder_out_im_size[0]
        num_downsamples =  downsampling_ratio // 2
        assert( num_downsamples == 1 or num_downsamples == 2 or num_downsamples == 4 )
        self.depth = num_downsamples                                                                                    ## (A)
        self.encoder_out_im_size = encoder_out_im_size
        self.encoder_out_ch = encoder_out_ch
        self.conv_in = nn.Conv2d(3, 64, 3, padding=1)
        self.bn1DN  = nn.BatchNorm2d(64)
        self.bn2DN  = nn.BatchNorm2d(128)
        self.bn3DN  = nn.BatchNorm2d(256)
        self.skip_arr = nn.ModuleList()
        if self.depth == 1:
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockEncoder(64, 64, downsample=True, skip_connections=skip_connections))
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockEncoder(64, 128, downsample=False, skip_connections=skip_connections))
            for _ in range(num_repeats):                                                                                ## (B)
                self.skip_arr.append(DLStudio.Autoencoder.SkipBlockEncoder(128, 128, downsample=False, skip_connections=skip_connections))
        elif self.depth == 2:
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockEncoder(64, 64, downsample=True, skip_connections=skip_connections))
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockEncoder(64, 128, downsample=False, skip_connections=skip_connections))
            for _ in range(num_repeats):                                                                                ## (C)
                self.skip_arr.append(DLStudio.Autoencoder.SkipBlockEncoder(128, 128, downsample=False, skip_connections=skip_connections))
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockEncoder(128, 128, downsample=True, skip_connections=skip_connections))
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockEncoder(128, 256, downsample=False, skip_connections=skip_connections))
            for _ in range(num_repeats):                                                                                ## (D)
                self.skip_arr.append(DLStudio.Autoencoder.SkipBlockEncoder(256, 256, downsample=False, skip_connections=skip_connections))
        elif self.depth == 4:
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockEncoder(64, 64, downsample=True, skip_connections=skip_connections))
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockEncoder(64, 128, downsample=False, skip_connections=skip_connections))
            for _ in range(num_repeats):                                                                                ## (E)
                self.skip_arr.append(DLStudio.Autoencoder.SkipBlockEncoder(128, 128, downsample=False, skip_connections=skip_connections))
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockEncoder(128, 128, downsample=True, skip_connections=skip_connections))
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockEncoder(128, 256, downsample=False, skip_connections=skip_connections))
            for _ in range(num_repeats):                                                                                ## (F)
                self.skip_arr.append(DLStudio.Autoencoder.SkipBlockEncoder(256, 256, downsample=False, skip_connections=skip_connections))
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockEncoder(256, 256, downsample=True, skip_connections=skip_connections))
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockEncoder(256, 512, downsample=False, skip_connections=skip_connections))
            for _ in range(num_repeats):                                                                                ## (G)
                self.skip_arr.append(DLStudio.Autoencoder.SkipBlockEncoder(512, 512, downsample=False, skip_connections=skip_connections))
        self.skip128DN = DLStudio.Autoencoder.SkipBlockEncoder(128,128, skip_connections=skip_connections)

    def forward(self, x):
        x = nn.functional.relu(self.conv_in(x))
        for layer in self.skip_arr:
            x = layer(x)
        if (x.shape[2:] != self.encoder_out_im_size) or (x.shape[1] != self.encoder_out_ch):
            print("\n\nShape of x at output of Encoder: ", x.shape)
            sys.exit("\n\nThe Encoder part of the Autoencoder is misconfigured. Encoder output not according to specs\n\n")
        return x
```

Inner Class `DecoderForAutoenc` of Autoencoder

- As I mentioned earlier, of the Encoder and the Decoder, it is the former that needs to be more "powerful". That was the reason for endowing the Encoder constructor with the `num_repeats` parameter that you saw on the previous slide.
- The Decoder can be a bit more rudimentary as you see in the implementation code shown below.

```
class DecoderForAutoenc(nn.Module):

    def __init__(self, dl_studio, decoder_in_im_size, decoder_out_im_size, skip_connections=True):
        super(DLStudio.Autoencoder.DecoderForAutoenc, self).__init__()
        upsampling_ratio =  decoder_out_im_size[0] // decoder_in_im_size[0]
        num_upsamples =  upsampling_ratio // 2
        assert( num_upsamples == 1 or num_upsamples == 2 or num_upsamples == 4)
        self.depth = num_upsamples
        self.decoder_out_im_size = decoder_out_im_size
        self.conv_in = nn.Conv2d(3, 64, 3, padding=1)
        self.bn1DN  = nn.BatchNorm2d(64)
        self.bn2DN  = nn.BatchNorm2d(128)
        self.bn3DN  = nn.BatchNorm2d(256)
        self.skip_arr = nn.ModuleList()
        if self.depth == 1:
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockDecoder(128, 64, upsample=False, skip_connections=skip_connections))
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockDecoder(64, 64, upsample=True, skip_connections=skip_connections))
        elif self.depth == 2:
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockDecoder(256, 128, upsample=False, skip_connections=skip_connections))
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockDecoder(128, 128, upsample=True, skip_connections=skip_connections))
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockDecoder(128, 64, upsample=False, skip_connections=skip_connections))
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockDecoder(64, 64, upsample=True, skip_connections=skip_connections))
        elif self.depth == 4:
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockDecoder(512, 256, upsample=False, skip_connections=skip_connections))
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockDecoder(256, 256, upsample=True, skip_connections=skip_connections))
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockDecoder(256, 128, upsample=False, skip_connections=skip_connections))
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockDecoder(128, 128, upsample=True, skip_connections=skip_connections))
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockDecoder(128, 64, upsample=False, skip_connections=skip_connections))
            self.skip_arr.append(DLStudio.Autoencoder.SkipBlockDecoder(64, 64, upsample=True, skip_connections=skip_connections))
        self.bn1UP  = nn.BatchNorm2d(256)
        self.bn2UP  = nn.BatchNorm2d(128)
        self.bn3UP  = nn.BatchNorm2d(64)
        self.conv_out3 = nn.ConvTranspose2d(64,3, 3, stride=1,dilation=1,output_padding=0,padding=1)

    def forward(self, x):
        for layer in self.skip_arr:
            x = layer(x)
        x = self.conv_out3(x)
        if x.shape[2:] != self.decoder_out_im_size:
            print("\n\nShape of x at output of Decoder: ", x.shape)
            sys.exit("\n\nThe Decoder part of the Autoencoder is misconfigured. Output image not according to specs\n\n")
        return x
```

Some Results Obtained with the Autoencoder Class

I will now show some results obtained with the Autoencoder class on the `PurdueShapes5GAN` dataset of 20,000 images. What you below is a batch of 48 images drawn from this dataset.

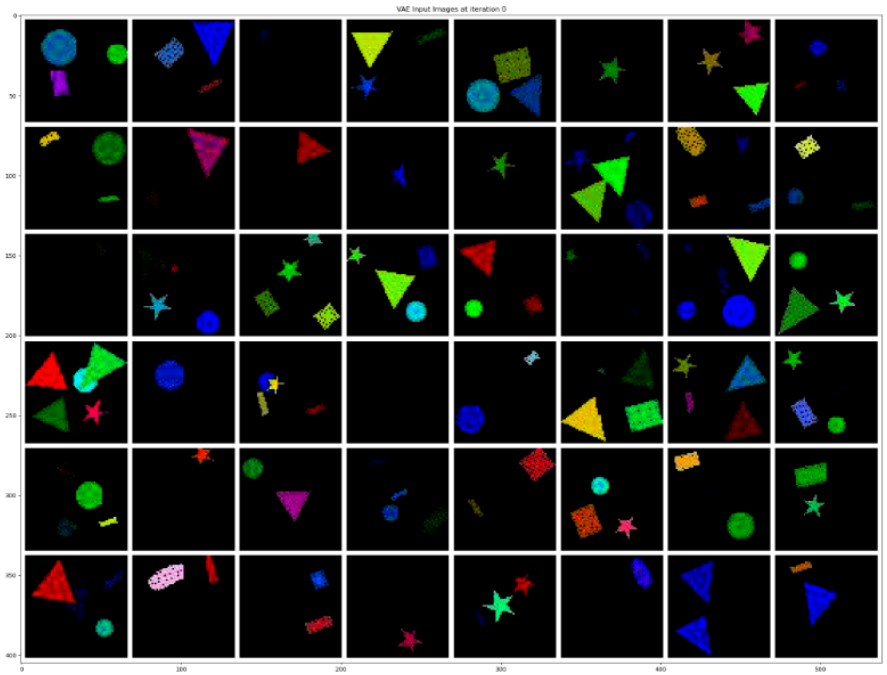
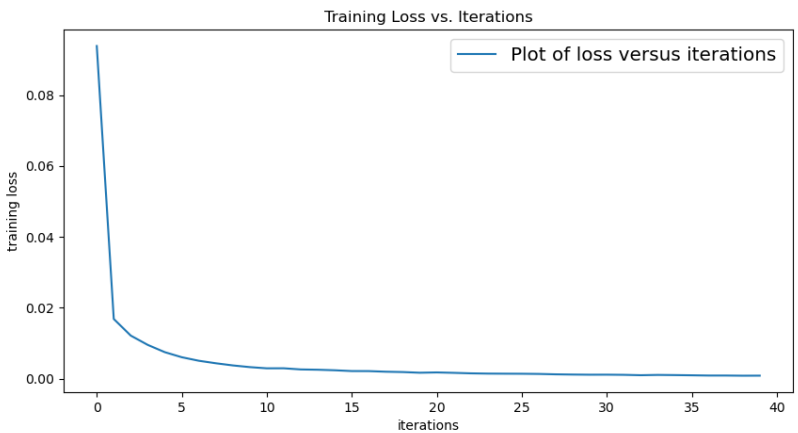


Figure: A batch of 48 input images used during the training of the Autoencoder network.

Results with the AutoEncoder Class (contd.)

- Shown below is training-loss-vs-iterations plot with the following parameters used for training (there was no hyperparameter tuning):

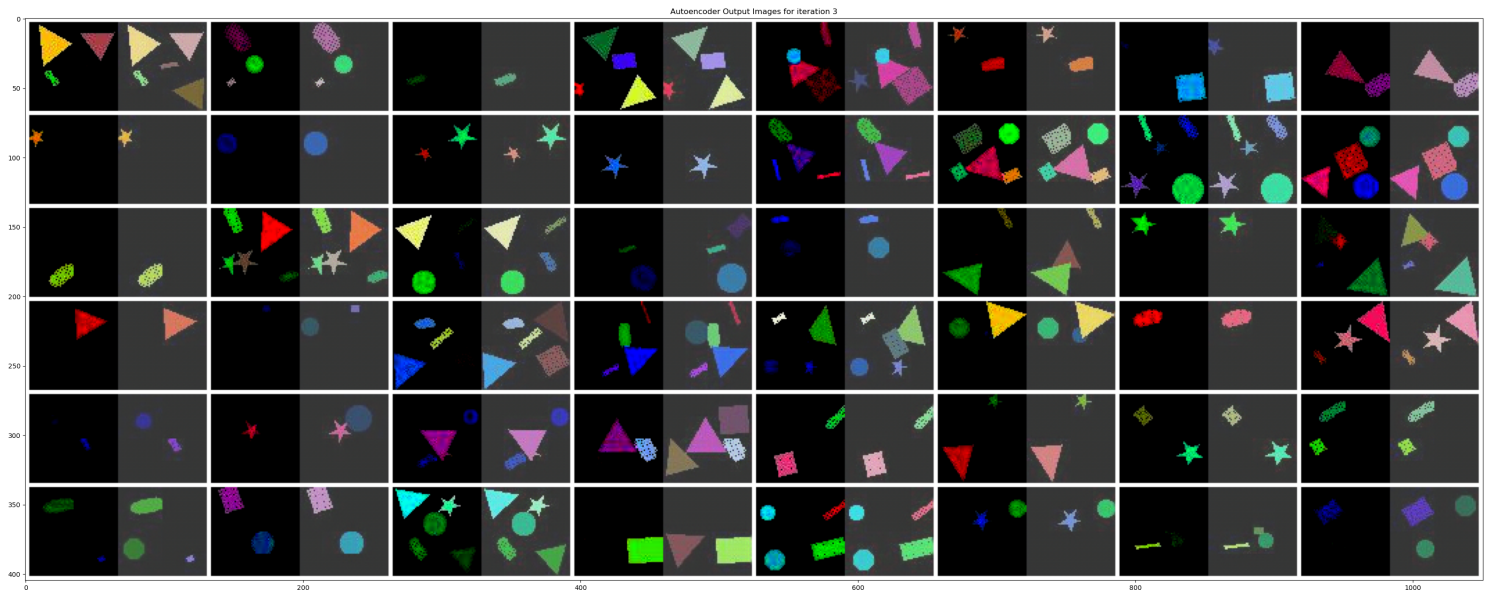
```
Input image size:  64x64
Encoder output size: 16x16
Encode output channels (the latent dimension): 256
Learning rate: 1e^-4
Number of epochs: 20
Total number of training iterations: 16,520
```



Figure

Some Results Obtained with the Autoencoder Class

- Shown below is the output of the Decoder for input images **NOT** seen during training.
- Inside each white-border delineated box, the left half shows the input image and the right half the corresponding output produced by the Decoder.



Outline

VAE in DLStudio

- As previously mentioned, the VAE class in DLStudio is derived from the parent class Autoencoder described in the previous section. As you would expect, a bulk of the heavy lifting in VAE is done through the functionality packed into the Autoencoder class.
- The VAE class focuses your mind on what exactly should be the output of an autoencoding encoder if the goal is to learn the latent space representation of the probability distribution that describes your training dataset. I previously posed this question on Slide 31.
- As I stated in Section 3, taking practical feasibility into account, we should assume that the latent space can be described simply by an isotropic normal distribution and we should try to learn its mean and variance subject to the condition that they be as close as possible to the mean and variance of the standard normal distribution (zero-mean and unit variance).

VAE in DLStudio (contd.)

- What I have said on the previous slide implies that our goal is **not** to learn how an input image x would map to a reduced-dimensionality vector z, but to learn the parameters μ and σ of the latent distribution.
- In the rest of this section, I'll provide a brief description of the VAE code presented on Slides 51-53.
- The two important components of the VAE class are the `VaeEncoder` and the `VaeDecoder`.
- About the class `VaeDecoder`, its job is to take the `mu` and `logvar` values produced by the Encoder and generate an output image that contains the information that the user wants to see there.
- About the class `VaeEncoder`, the most important thing to note is that this encoder outputs ONLY the mean and the log-variance of the Gaussian distribution that models the latent vectors.

VAE in DLStudio (contd.)

- Regarding the function `reparameterize(self, mu, logvar)`, note that `torch.randn` is sampling from an isotropic zero-mean unit-covariance Gaussian. The call `torch.randn_like` ensures that the returned tensor will have the same shape as the `std` tensor.

  [**In order to understand the shape of `std`, consider the case when the size of the pixel array at the Encoder output is 8×8, the embedding size 128, and the *batch_size* 48. In this case, you have 64 pixels at the output of the Encoder (before you go into the Linear layers for mu and logvar estimation). So the shape of both `logvar` and `std` is going to be [48, 8192] where 8192 is the product of the 64 pixels and the 128 channels at each pixel. Note that the shapes for all three of `mu`, `logvar`, and `std` are identical and, for our example, that shape is $[48, 8192]$.**]

- About the function `run_code_for_training_VAE( self, vae_net, loss_weighting, display_train_loss=False )` Note that the code for `set_dataloaders()` for the VAE class shows how the overall dataset of images is divided into training and testing subsets.

- The important thing to keep in mind about this function is the relative weighting of the reconstruction loss vis-a-vis the KL-divergence. For an "optimized" VAE implementation, finding the best value to use for this relative weighting of the two loss components would be a part of hyperparameter tuning of the network.

VAE in DLStudio (contd.)

- About the function `run_code_for_evaluating_VAE(self, vae_net, visualization_dir = ''vae_visualization_dir'' )` the main point here is to use the co-called "unseen images" for evaluating the performance of the VAE Encoder-Decoder network. If you look at the `set_dataloader()` function for the VAE class, you will see me setting aside a certain number of the available images for testing. These randomly chosen images play NO role in training.
- About the function `run_code_for_generating_images_from_noise_VAE()`, this function is for testing the functioning of just the Generator (which is the Decoder) in the VAE network. That is, after we have trained the VAE network, we disconnect the Encoder and ask the Decoder to sample the latent distribution for generating the images.
- Remember, the latent distribution is represented entirely by the final values learned for the mean (*mu*) and the log of the variance (*logvar*) that represent how close the training process was able to come to the ideal of zero-mean and unit-covariance isotropic distribution.

VAE in DLStudio (contd.)

- Since the job of this function `run_code_for_generating_images_from_noise_VAE()` is to sample the latent distribution actually learned, we must also supply it with the (mu,logvar) values learned during training.
- Finally, about the definition of `set_dataloaders()`, note the call to `random_split()` in the second statement for dividing the overall dataset of images into two DISJOINT parts, one for training and the other for testing.
- To continue the last bullet on the previous slide, since my evaluation of the VAE at this time is purely on the basis of the visual quality of the output of the Decoder, I have set aside only 200 randomly chosen images for testing. Ordinarily, through, you would want to split the dataset in the 70:30 or 80:20 ratio for training and testing.

VAE in DLStudio (contd.)

```
class VAE (Autoencoder):

    def __init__(self, dl_studio, encoder_in_im_size, encoder_out_im_size, decoder_out_im_size, encoder_out_ch, num_repeats, path_saved_encoder, path_saved_decoder ):
        super(DLStudio.VAE, self).__init__( dl_studio, encoder_in_im_size, encoder_out_im_size, decoder_out_im_size, encoder_out_ch, num_repeats, path_saved_model=None )
        self.parent_encoder =  DLStudio.Autoencoder.EncoderForAutoenc(dl_studio, encoder_in_im_size, encoder_out_im_size, encoder_out_ch, num_repeats, skip_connections=True )
        self.parent_decoder =  DLStudio.Autoencoder.DecoderForAutoenc(dl_studio, encoder_out_im_size, decoder_out_im_size)
        self.vae_encoder =  DLStudio.VAE.VaeEncoder(self.parent_encoder, encoder_out_im_size, encoder_out_ch)
        self.vae_decoder =  DLStudio.VAE.VaeDecoder(self.parent_decoder, encoder_out_im_size, encoder_out_ch)
        self.encoder_out_im_size = self.encoder.encoder_out_im_size
        self.encoder_out_ch  =  self.encoder.encoder_out_ch
        self.path_saved_encoder = path_saved_encoder
        self.path_saved_decoder = path_saved_decoder


    class VaeEncoder(nn.Module):

        def __init__(self, parent_encoder, encoder_out_im_size, encoder_out_ch):
            super(DLStudio.VAE.VaeEncoder, self).__init__()
            self.parent_encoder = parent_encoder
            self.num_nodes = encoder_out_im_size[0] * encoder_out_im_size[1]  * encoder_out_ch
            self.latent_dim = encoder_out_ch
            self.mu_layer =  nn.Linear(self.num_nodes, self.latent_dim)
            self.log_var_layer =  nn.Linear(self.num_nodes, self.latent_dim)

        def forward(self, x):
           encoded = self.parent_encoder(x)
           mu  = self.mu_layer(encoded.view(-1, self.num_nodes))
           log_var = self.log_var_layer(encoded.view(-1, self.num_nodes))
           return mu, log_var


    class VaeDecoder(nn.Module):

        def __init__(self, parent_decoder, encoder_out_im_size, encoder_out_ch):
            super(DLStudio.VAE.VaeDecoder, self).__init__()
            self.parent_decoder = parent_decoder
            self.encoder_out_im_size = encoder_out_im_size
            self.num_nodes = encoder_out_im_size[0] * encoder_out_im_size[1]  * encoder_out_ch
            self.latent_dim = encoder_out_ch
            self.reparametrized_to_decoder_input = nn.Linear(self.latent_dim, self.num_nodes)

        def reparameterize(self, mu, logvar):
           std  =  torch.exp(0.5 * logvar)
           eps =  torch.randn_like( std )                    ## standard normal N(0;1)
           return mu + eps * std

        def forward(self, mu, logvar):
           z = self.reparameterize( mu, logvar )
           z = self.reparametrized_to_decoder_input(z)
           decoded = self.parent_decoder( z.view(-1, self.latent_dim, self.encoder_out_im_size[0], self.encoder_out_im_size[1]) )
           return decoded, mu, logvar


    def run_code_for_training_VAE( self, vae_net, loss_weighting, display_train_loss=False ):

        def loss_criterion(input_images, decoder_output_images, log_var, weighting):
            recon_loss = nn.MSELoss(reduction='sum')( input_images, decoder_output_images )   ## reconstruction loss
            KLD = -0.5 * torch.sum( 1 + log_var - mu.pow(2)  -  log_var.exp() )               ## KL Divergence
            KLD =  KLD * weighting
            return  recon_loss + KLD,  recon_loss, KLD

        vae_encoder = vae_net.vae_encoder.to(self.dl_studio.device)
        vae_decoder = vae_net.vae_decoder.to(self.dl_studio.device)
```

(Continued on the next slide .....)

VAE in DLStudio (contd.)

(..... continued from the previous slide)

```
accum_times = []
start_time = time.perf_counter()
print("")
batch_size = self.dl_studio.batch_size
print("\n\n batch_size: ", batch_size)
num_batches_in_data_source = len(self.train_dataloader)
total_num_updates = self.dl_studio.epochs * num_batches_in_data_source
print("\n\n number of batches in the dataset: ", num_batches_in_data_source)
optimizer1 = optim.Adam(vae_encoder.parameters(), lr=self.dl_studio.learning_rate)
optimizer2 = optim.Adam(vae_decoder.parameters(), lr=self.dl_studio.learning_rate)
mu = logvar = 0.0

total_training_loss_tally = []
recons_loss_tally = []
KL_divergence_tally = []

for epoch in range(self.dl_studio.epochs):
    print("")
    ##  The following are needed for calculating the avg values between displays:
    running_loss = running_recon_loss = running_kld_loss = 0.0
    for i, data in enumerate(self.train_dataloader):
        input_images, _ = data
        input_images = input_images.to(self.dl_studio.device)
        optimizer1.zero_grad()
        optimizer2.zero_grad()

        mu, logvar =  vae_encoder( input_images )
        ##  As required by VAE, the Decoder is only being supplied with the mean 'mu' and the log-variance 'logvar':
        decoder_out, _, _ =  vae_decoder( mu, logvar )
        loss, recon_loss, kld_loss  =  loss_criterion( input_images,  decoder_out, logvar, loss_weighting )

        loss.backward()
        optimizer1.step()
        optimizer2.step()
        running_loss += loss
        running_recon_loss += recon_loss
        running_kld_loss += kld_loss
        if i % 200 == 199:
            avg_loss = running_loss / float(200)
            avg_recon_loss = running_recon_loss / float(200)
            avg_kld_loss = running_kld_loss / float(200)

            total_training_loss_tally.append(avg_loss.item())
            recons_loss_tally.append(avg_recon_loss.item())
            KL_divergence_tally.append(avg_kld_loss.item())

            running_loss = running_recon_loss = running_kld_loss = 0.0
            current_time = time.perf_counter()
            time_elapsed = current_time-start_time
            print("[epoch:%2d/%2d  i:%4d  elapsed_time: %4d secs]     loss: %10.4f      recon_loss: %10.4f      kld_loss:  %10.4f " %
                                  (epoch+1, self.dl_studio.epochs, i+1,time_elapsed,avg_loss,avg_recon_loss,avg_kld_loss))
            accum_times.append(current_time-start_time)

print("\nFinished Training\n")
self.save_encoder_model( vae_encoder )
self.save_decoder_model( vae_decoder )

params_saved = { 'mean': mu, 'log_variance': logvar}
pickle.dump(params_saved, open('params_saved.p', 'wb'))
```

(Continued on the next slide .....)

VAE in DLStudio (contd.)

(..... continued from the previous slide)

```
def run_code_for_evaluating_VAE(self, vae_net, visualization_dir = "vae_visualization_dir" ):
    vae_encoder = vae_net.vae_encoder.eval()
    vae_decoder = vae_net.vae_decoder.eval()
    vae_encoder.load_state_dict(torch.load(self.path_saved_encoder))
    vae_decoder.load_state_dict(torch.load(self.path_saved_decoder))
    vae_encoder.to(self.dl_studio.device)
    vae_decoder.to(self.dl_studio.device)
    with torch.no_grad():
        for i, data in enumerate(self.test_dataloader):
            print("\n\n\n=========Showing results for test batch %d==============" % i)
            test_images, _ = data
            test_images = test_images.to(self.dl_studio.device)
            mu, logvar =  vae_encoder( test_images )
            ##  In the next statement, using mu and logvar, the Decoder first uses the "reparameterization trick"
            ##  to sample the latent distribution and to then feed it into the rest of the Decoder for image generation:
            decoder_out, _, _ =  vae_decoder( mu, logvar )
            decoder_out  =  ( decoder_out - decoder_out.min() ) / ( decoder_out.max() -  decoder_out.min() )
            together = torch.zeros( test_images.shape[0], test_images.shape[1], test_images.shape[2], 2 * test_images.shape[3], dtype=torch.float )
            together[:,:,:,0:test_images.shape[3]]  =  test_images
            together[:,:,:,test_images.shape[3]:]  =   decoder_out
            plt.figure(figsize=(40,20))
            plt.imshow(np.transpose(torchvision.utils.make_grid(together.cpu(), normalize=False, padding=3, pad_value=255).cpu(), (1,2,0)))
            plt.title("VAE Output Images for iteration %d" % i)
            plt.savefig(visualization_dir + "/vae_decoder_out_%s" % str(i) + ".png")
            plt.show()


def run_code_for_generating_images_from_noise_VAE(self, vae_net, visualization_dir = "vae_gen_visualization_dir" ):
    vae_decoder = vae_net.vae_decoder.eval()
    vae_decoder.load_state_dict(torch.load(self.path_saved_decoder))
    params_saved = pickle.load( open('params_saved.p', 'rb') )
    mu, logvar = params_saved['mean'], params_saved['log_variance']

    ##  The size of the batch axis for the mu and logvar tensors will corresponds to the number
    ##  of images in the last batch used for training.  If you want the purely generative process
    ##  in this script (which uses the VAE Decoder in a standalone mode) to produce a batchful of
    ##  images, you need to expand the previously learned mu and logvar tensors as shown below:
    if mu.shape[0] < self.dl_studio.batch_size:
        new_mu = torch.zeros( (self.dl_studio.batch_size, mu.shape[1]) ).float()
        new_mu[:mu.shape[0]]  =  mu
        new_mu[mu.shape[0]:] = mu[:(self.dl_studio.batch_size - mu.shape[0])]

        new_logvar = torch.zeros( (self.dl_studio.batch_size, logvar.shape[1]) ).float()
        new_logvar[:logvar.shape[0]]  = logvar
        new_logvar[logvar.shape[0]:] = logvar[:(self.dl_studio.batch_size - logvar.shape[0])]
    mu = new_mu.to(self.dl_studio.device)
    logvar = new_logvar.to(self.dl_studio.device)
    vae_decoder.to(self.dl_studio.device)
    sample_standard_normal_distribution = True
    sample_learned_normal_distribution =  False
    with torch.no_grad():
        for i in range(5):
            print("\n\n\n=========Showing results for test batch %d==============" % i)
            if sample_standard_normal_distribution:
                mu  =  torch.zeros_like(mu).float().to(self.dl_studio.device)
                logvar = torch.ones_like(logvar).float().to(self.dl_studio.device)
            elif sample_learned_normal_distribution:
                std = torch.exp(0.5 * logvar)
```

(Continued on the next slide .....)

VAE in DLStudio (contd.)

(..... continued from the previous slide)

```
            ##  In the next statement, using mu and logvar, the Decoder first uses the "reparameterization trick"
            ##  to sample the latent distribution and to then feed it into the rest of the Decoder for image generation:
            decoder_out, _, _ =  vae_decoder( mu, logvar )
            decoder_out  =  ( decoder_out - decoder_out.min() ) / ( decoder_out.max() -  decoder_out.min() )
            fake_input = torch.zeros_like(decoder_out).float().to(self.dl_studio.device)
            together = torch.zeros( fake_input.shape[0], fake_input.shape[1], fake_input.shape[2], 2 * fake_input.shape[3], dtype=torch.float )
            together[:,:,:,0:fake_input.shape[3]]  =  fake_input
            together[:,:,:,fake_input.shape[3]:]  =   decoder_out
            plt.figure(figsize=(40,20))
            plt.imshow(np.transpose(torchvision.utils.make_grid(together.cpu(), normalize=False, padding=3, pad_value=255).cpu(), (1,2,0)))
            plt.title("VAE Output Images for iteration %d" % i)
            plt.savefig(visualization_dir + "/vae_decoder_out_%s" % str(i) + ".png")
            plt.show()


def set_dataloader(self):
    dataset = torchvision.datasets.ImageFolder(root=self.dl_studio.dataroot,
                transform = tvt.Compose([
                                tvt.Resize(self.dl_studio.image_size),
                                tvt.CenterCrop(self.dl_studio.image_size),
                                tvt.ToTensor(),
                                tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                ]))

    dataset_train, dataset_test  =  torch.utils.data.random_split( dataset, lengths = [len(dataset) - 200, 200])
    self.train_dataloader = torch.utils.data.DataLoader(dataset_train, batch_size=self.dl_studio.batch_size, shuffle=True, num_workers=4)
    self.test_dataloader = torch.utils.data.DataLoader(dataset_test, batch_size=self.dl_studio.batch_size, shuffle=True, num_workers=4)
```

Outline

Results with VAE in DLStudio

- The results obtained with VAE that I am going to show in this section were obtained by training the VAE network on the same `PurdueShapes5GAN` dataset of 20,000 images that I used for the results obtained with the Autoencoder previously in this lecture.
- As you already know, each image in the dataset is of size 64×64. Each image in the `PurdueShapes5GAN` dataset contains a random number of up to five shapes: rectangle, triangle, disk, oval, and star. Each shape is located randomly in the image, oriented randomly, and assigned a random color. Since the orientation transformation is carried out without bilinear interpolation, it is possible for a shape to acquire holes in it.
- The figure on the next slide shows an example of batch of images when the batch size is set to 48 that is fed into the VAE network for the results in this section.

Results with VAE in DLStudio (contd.)

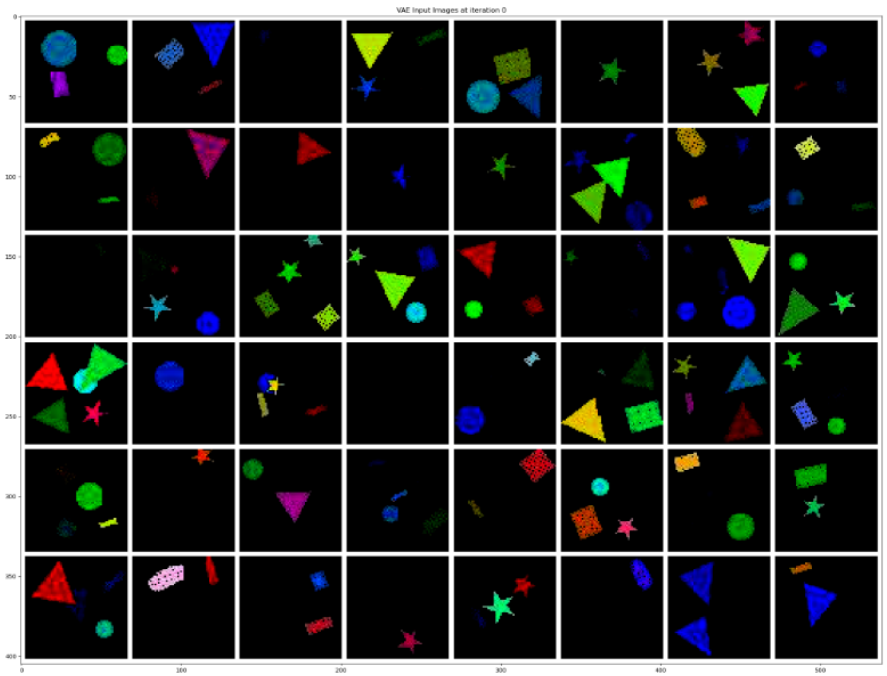
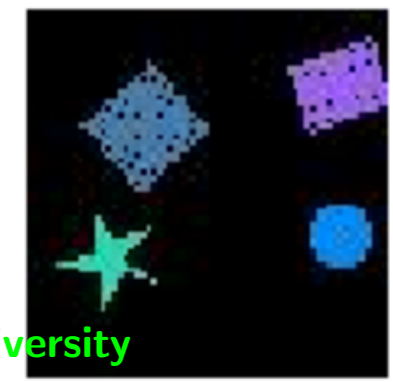


Figure: An example batch of 48 input images used during the training of the VAE network.

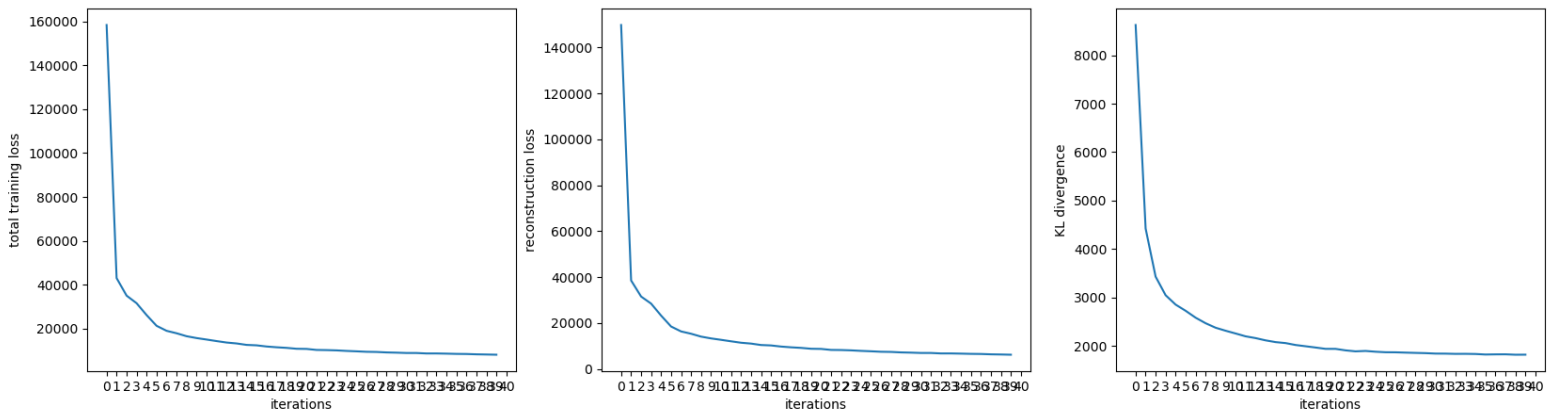
Results with VAE in DLStudio (contd.)

- Note that the images in the previous slide are not as simple as they may seem. Shown in the next figure are enlarged versions of two of the images like those in the two batches shown. In addition to the sharp shape boundaries, you can also small holes inside the shapes. The holes that you see inside the shapes were caused by intentionally suppressing bilinear interpolation as the shapes were randomly reoriented.
- So the challenge for the data modeler would be its ability to not only reproduce the shapes while preserving the sharp edges, but also to incorporate the tiny holes inside the shapes, **and do so with the probabilities that reflect the training data.**



Results with VAE in DLStudio (contd.)

- The results I show next were obtained by executing the following script in the `Examples` directory of the DLStudio platform: `run_vae.py`
- Shown in the figure below are the losses recorded during the training of the VAE network over 20 epochs. As you know from the discussion in Section 2, we have two different kinds of losses: the reconstruction loss and the KL-divergence loss, the former shown by the plot in the middle and the latter by the plot on the right. The plot on the left is the total loss. The horizontal axis is for the iterations over 20 epochs of training.



Results with VAE in DLStudio (contd.)

- **It is well known that the loss decreasing with training iterations is a necessary but not a sufficient condition that any useful learning is actually taking place and that you are not inadvertently overfitting to the training data.**
- Therefore, after the training is finished, you must test the VAE by feeding previously unseen at the input to the Encoder and then evaluate the quality of the Decoder output for such images.
- The next figure is an example of the result obtained when a batch of previously unseen images is presented as input to the Encoder. As the caption says, what you are looking at a composite of the input images and the outputs produced by the Decoder. Each input/output pair is shown in a white-border delimited box, with its left-half being the input image and the right-half the Decoder output.

Results with VAE in DLStudio (contd.)

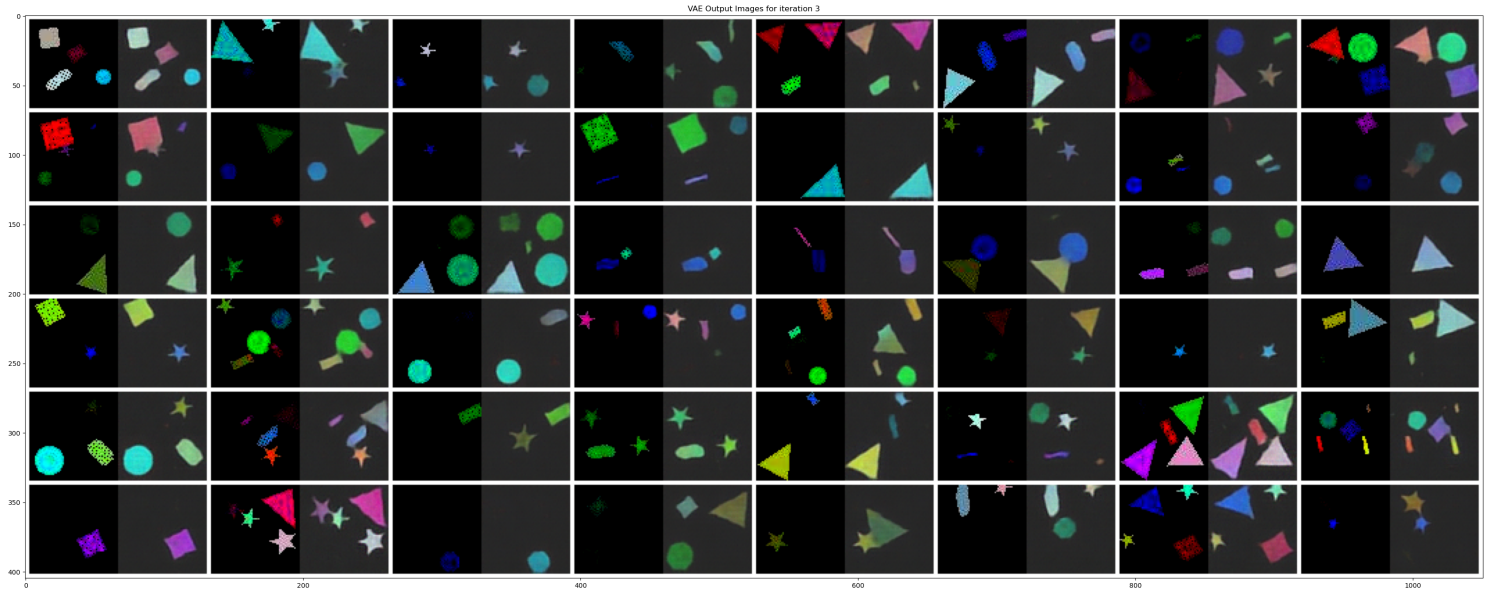


Figure: **Images at the output of the VAE Decoder during testing with previously unseen images at the input to the Encoder. Inside each box delineated by white lines, the left-half shows the image at the input to the Encoder and the right-half shows the corresponding image at the output of the Decoder.**

Outline

VQVAE — VAE with Codebook Learning

- VQVAE stands for "Vector Quantized Variational Auto Encoder", which is also frequently represented by the acronym VQ-VAE. The concept of VQ-VAE was formulated in the 2018 paper "Neural Discrete Representation Learning" by van den Oord, Vinyals, and Kavukcuoglu as mentioned in the Preamble.

- VQVAE is an important architecture in deep learning because it teaches us about what has come to be known as "**Codebook Learning**" for creating discrete representations for images. The Codebook learning consists of learning a fixed number of learned embedding vectors. Subsequently, in an overall Encoder-Decoder architectures, you replace each element at the output of the Encoder with the closest embedding vector in the Codebook. The dimensionality you associate with a pixel at the output of the Encoder is the number of channels at that point.

VQVAE — VAE with Codebook Learning (contd.)

- **You can think of the learned Codebook vectors as the quantized versions of what the Encoder presents at its output.**
- The next slide presents the same figure that I described in great detail in Slides 12-15. But now let's examine the forward and backpropagation of information in this network.
- It is easy to visualize the forward flow of information in this diagram, notwithstanding the presence of the step in which we replace each embedding vector in the left datacube with its closest approximation from the collection of codebook vector to result in the right datacube. The left datacube, marked $z_{enc}(x)$, represents the output of the Encoder and the right datacube, marked $z_q(x)$, represents the input to the Decoder.
- **While forward propagation is easy to visualize, how does one backpropagate the gradients of the loss across the forward-prop step in which you replace the embedding vectors at the output of the Encoder with vectors from the codebook?**

VQVAE — VAE with Codebook Learning

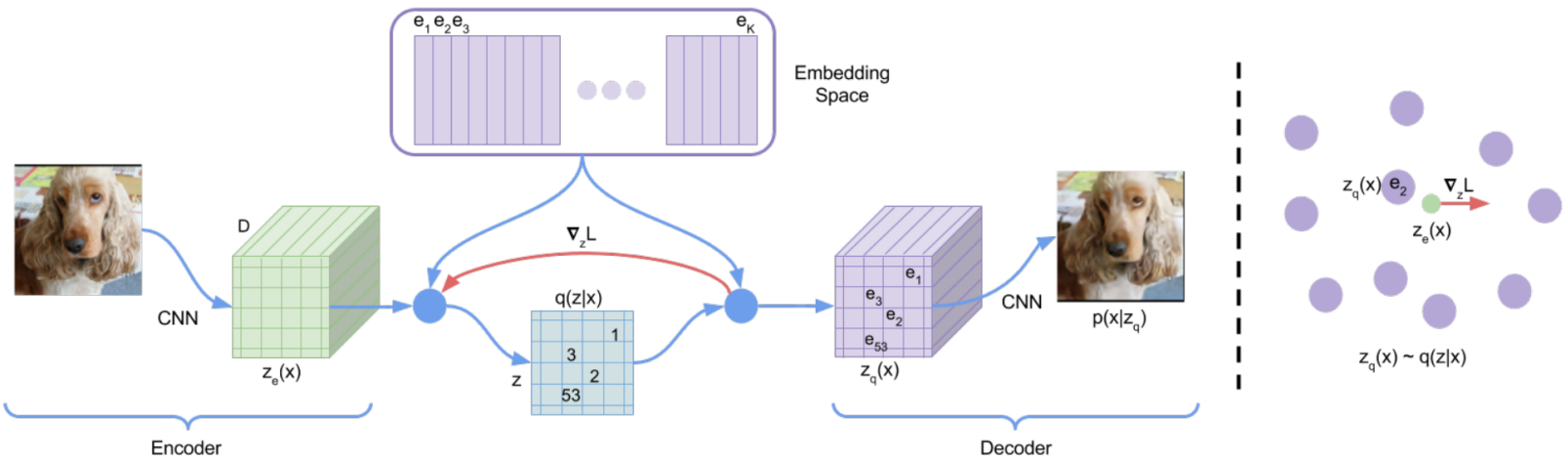


Figure: This figure is from the famous VQVAE paper `https://arxiv.org/pdf/1711.00937`

VQVAE — VAE with Codebook Learning (contd.)

- To add to the question at the bottom of Slide 64, **How exactly do we formulate the loss for VQVAE learning?**
- To respond to the question posed at the bottom of Slide 64 and the question posed above, authors van den Oord et al. have proposed the following the three-part loss function for training a VQVAE in which x in the image at the input to the Encoder and $\hat{x}$ the image at the output of the Decoder, $z_{enc}(x)$ the sequence of output embeddings produced by the Encoder and e the sequence of the closest codebook vectors chosen as replacements for the output emeddings:

$$\mathcal{L} \;=\; ||x - \hat{x}||^2 \;+\; \Big|\Big|sg[z_{enc}(x)] - e\Big|\Big|_2^2 \;+\; \beta\Big|\Big|z_{enc}(x) - sg[e]\Big|\Big|_2^2 \tag{22}$$

- Note the use of the $sg[]$ operator in the second and the third terms. It is known as the *stop-gradient* operator that is needed to cope with the problem created by our wanting to backpropagate the gradients of loss over the operation in which we replace the embedding-vectors at the output of the Encoder with the closest codebook vectors.

VQVAE — VAE with Codebook Learning (contd.)

- The first term shown in Eq. (22) is the reconstruction loss as in a regular VAE (See Slides 24 and 29) which optimizes the learnable parameters in both the Encoder and the Decoder.
- The second and the third terms in the expression for loss on the previous slide can be interpreted as the quantization loss for the second term and the commitment loss for the third term.
- The second term is the quantization loss since the gradients will only be calculated for the codebook vectors and, therefore, the updates will only be made to the codebook vectors in order to move them closer to the encoder output vectors.
- The third term is the commitment loss since the calculated gradients will only be there the Encoder output vectors and any updates to those vectors will move them closer to the closest codebook vectors.
- There remains the question of how to implement the stop-gradient operator in your code. I will address that in the next section.

Outline

VQVAE in DLStudio

- In DLStudio, like the VAE class, the VQVAE class is also derived from the parent class Autoencoder. Bulk of the computing in VQVAE is done through the functionality packed into the Autoencoder class that was presented earlier in this lecture.

- In what follows, I'll first show some code from the function run_code_for_training_VQVAE(). This is just to indicate how the image data flows through the overall network so that you'll better understand at what point the different parts of the overall Loss shown on Slide 66 are calculated.

- In the code shown on Slide 71, we grab a batch of images in Line (D) and feed them into the Encoder in Line (E), the output of which is fed into the Vector Quantizer in Line (G). Finally, the output of the Vector Quantizer is fed into the Decoder in Line (H).

VQVAE in DLStudio

- The call to Vector Quantizer in Line (G) also returns the a quantity named `vq_loss`, which is a sum of the second and the third terms for the equation for Loss on Slide 66.
- Just from the calls in the code shown on the next slide, it is clear that the VectorQuantizer network shown on Slide 75 has the responsibility of implementing all of the core notions related to the learning of the codebook vectors and the calculation of the loss `vq_loss`.
- The `recon_loss` that is calculated in Line (I) is the reconstruction loss, which is the first term in the overall loss equation on Slide 66.
- The code shown on the next slide also calculates the "perplexity" metric for the codebook. Perplexity is 2 to the power of the Entropy associated with the code vectors in the codebook.

  [Say, we have 256 vectors in the codebook. If all the codebook vectors are used equally often, their entropy would be 8 and the perplexity would be 256. When the perplexity is less than the number of codebook vectors, the codebook vectors are not being utilized are not being utilized as effectively as they could be.]

VQVAE in DLStudio (contd.)

```
def run_code_for_training_VQVAE( self, vqvae, display_train_loss=False ):

    vqvae_encoder = vqvae.vqvae_encoder.to(self.dl_studio.device)
    vqvae_vector_quantizer =  vqvae.vector_quantizer.to(self.dl_studio.device)
    vqvae_decoder = vqvae.vqvae_decoder.to(self.dl_studio.device)
    pre_vq_conv   =  self.pre_vq_conv.to(self.dl_studio.device)
    accum_times = []
    start_time = time.perf_counter()
    print("")
    batch_size = self.dl_studio.batch_size
    print("\n\n batch_size: ", batch_size)
    num_batches_in_data_source = len(self.train_dataloader)
    total_num_updates = self.dl_studio.epochs * num_batches_in_data_source
    print("\n\n number of batches in the dataset: ", num_batches_in_data_source)
    optimizer1 = optim.Adam(vqvae_encoder.parameters(), lr=self.dl_studio.learning_rate)                          ## (A)
    optimizer2 = optim.Adam(vqvae_decoder.parameters(), lr=self.dl_studio.learning_rate)                          ## (B)
    optimizer3 = optim.Adam(vqvae_vector_quantizer.parameters(), lr=self.dl_studio.learning_rate)                 ## (C)
    training_loss_tally = []
    perplexity_tally = []
    data_variance = 0.0
    for epoch in range(self.dl_studio.epochs):
        print("")
        running_loss = 0.0
        running_perplexity = 0.0
        for i, data in enumerate(self.train_dataloader):
            input_images, _ = data                                                                                ## (D)
            input_images = input_images.to(self.dl_studio.device)
            optimizer1.zero_grad()
            optimizer2.zero_grad()
            optimizer3.zero_grad()
            z = vqvae_encoder(input_images)                                                                       ## (E)
            z = pre_vq_conv(z)                                                                                    ## (F)
            vq_loss, quantized, perplexity, _ =  vqvae_vector_quantizer(z)                                        ## (G)
            decoder_out = vqvae_decoder(quantized)                                                                ## (H)
            recon_loss = nn.MSELoss(reduction='sum')( input_images, decoder_out )                                 ## (I)
            loss = recon_loss + vq_loss
            loss.backward()
            optimizer1.step()
            optimizer2.step()
            optimizer3.step()
            running_loss += loss
            running_perplexity += perplexity
            if i % 200 == 199:
                avg_loss = running_loss / float(200)
                avg_perplexity = running_perplexity / float(200)
                training_loss_tally.append(avg_loss.item())
                perplexity_tally.append(avg_perplexity.item())
                running_loss = 0.0
                running_perplexity = 0.0
                current_time = time.perf_counter()
                time_elapsed = current_time-start_time
                print("[epoch:%2d/%2d  i:%4d  elapsed_time: %4d secs]   loss: %10.6f         perplexity: %10.6f " %
                                                  (epoch+1, self.dl_studio.epochs, i+1, time_elapsed, avg_loss, avg_perplexity))
                accum_times.append(current_time-start_time)
    print("\nFinished Training\n")
```

VQVAE in DLStudio

- Slide 75 shows an implementation of the `VectorQuantizer` class that is in charge of calculating the second the third loss terms in the formulation of the overall loss on Slide 66.
- Let's first to over the initialization of an instance of this class. Notice how the codebook is declared in Line (C) and its weights initialized in Line (D). The number of embeddings (at this point synonymous with codebook vectors) to use the their dimensionality is set by the user. However, note you cannot violate the constraint that the channel dimension at the output of the Encoder must equal the dimensionality of the codebook vectors. The declaration in Line (E) is for the β coefficient in the expression for Loss in Slide 66.
- Regarding the code in `forward()`, we feed the output of the Encoder into the `VectorQuantizer` network. When the `batch_size` is, say, 48, the `image_size` is 64×64, the Encoder `enc_output_size` 8×8 and the number of output channels 128, the shape of the `input` tensor in Line (F) will be $(48, 128, 8, 8)$.

VQVAE in DLStudio

- We prefer to think of the output of the Encoder as a flattened 1D sequence of embedding vectors, with each vector being of the specified embedding size. Flattening consists of joining one row after another in the 8×8 array at the output of the Encoder. As shown in Line (H), we can get this flattening effect by supplying "-1" as the first argument to `tensor.view()` provided we first reshape the input so that the channel axis is the last, as accomplished in Line (F) itself.
- Now we are ready to calculate the square of the Euclidean distance between each vector that is in the output layer of the Encoder and every codebook vector, as shown in Line (I). The formula used there is the same as at the bottom of Slide 45 of my Week 9 lecture on Metric Learning.
- In Line (J), we collect the integer indices of the codebook vectors, each on the basis of being the closest to the corresponding Encoder output vector. These are the integers you see as 1, 3, 2, 53, etc., in the right datacube in Slide 57.

VQVAE in DLStudio

- In Line (K) we now initialize a zero tensor in which we will store the replacement codebook vectors for the Encoder output vectors. Line (L) creates a one-hot vector representation for each codebook vector index that was chosen. The matrix multiplication in Line (M) then yields a sequence of the codebook vectors that are replacements for the encoder output embeddings in the sequence created by flattening in Line (H).
- Now we are ready to calculate the quantization loss and the commitment loss defined on Slide 66. The question here is how do we implement the stop-gradient operator $sg[]$ defined there. As shown by the statements in Lines (N) and (O), we can use the *detach*() provided by PyTorch for that. Calling *detach*() on a tensor is the implementation of $sg[]$ on that tensor. The call to *detach*() when invoked on a tensor returns a copy of that tensor from the computational graph. So no gradients would be calculated for the tensor returned by *detach*().

VQVAE in DLStudio (contd.)

```
class VectorQuantizer(nn.Module):
    """
    This class is from:   https://github.com/zalandoresearch/pytorch-vq-vae
    """

    def __init__(self, num_embeddings, embedding_dim, commitment_cost):
        super(DLStudio.VQVAE.VectorQuantizer, self).__init__()
        self._embedding_dim = embedding_dim                                                                    ## (A)
        self._num_embeddings = num_embeddings                                                                  ## (B)

        self._embedding = nn.Embedding(self._num_embeddings, self._embedding_dim)                              ## (C)
        self._embedding.weight.data.uniform_(-1/self._num_embeddings, 1/self._num_embeddings)                  ## (D)
        ##  The \beta term for the Commitment Cost in Slide 66:
        self._commitment_cost = commitment_cost                                                                ## (E)

    def forward(self, inputs):                                    ##  "inputs"  is the output of Encoder
        # convert inputs from BCHW -> BHWC
        inputs = inputs.permute(0, 2, 3, 1).contiguous()                                                       ## (F)
        ## needed later for shape restoration with unflattening:
        input_shape = inputs.shape                                                                             ## (G)
        # Flatten input
        flat_input = inputs.view(-1, self._embedding_dim)                                                      ## (H)
        # Calculate distances between the input embedding vector and each of the codebook vectors
        distances = (torch.sum(flat_input**2, dim=1, keepdim=True) + torch.sum(self._embedding.weight**2, dim=1)
                                                - 2 * torch.matmul(flat_input, self._embedding.weight.t()))    ## (I)
        # Codebook vector index values:
        encoding_indices = torch.argmin(distances, dim=1).unsqueeze(1)                                         ## (J)
        encodings = torch.zeros(encoding_indices.shape[0], self._num_embeddings, device=inputs.device)         ## (K)
        encodings.scatter_(1, encoding_indices, 1)                                                             ## (L)
        # Quantize and unflatten
        quantized = torch.matmul(encodings, self._embedding.weight).view(input_shape)                          ## (M)
        ##  Commitment Loss (Slide 66):
        e_latent_loss = F.mse_loss(quantized.detach(), inputs)                                                 ## (N)
        ##  Quantization Loss (Slide 66):
        q_latent_loss = F.mse_loss(quantized, inputs.detach())                                                 ## (O)
        ##  See __init__() for the commitment_cost
        loss = q_latent_loss + self._commitment_cost * e_latent_loss                                           ## (P)

        quantized = inputs + (quantized - inputs).detach()                                                     ## (Q)
        avg_probs = torch.mean(encodings, dim=0)                                                               ## (R)
        perplexity = torch.exp(-torch.sum(avg_probs * torch.log(avg_probs + 1e-10)))                           ## (S)

        # convert quantized from BHWC -> BCHW
        return loss, quantized.permute(0, 3, 1, 2).contiguous(), perplexity, encodings
```

Outline

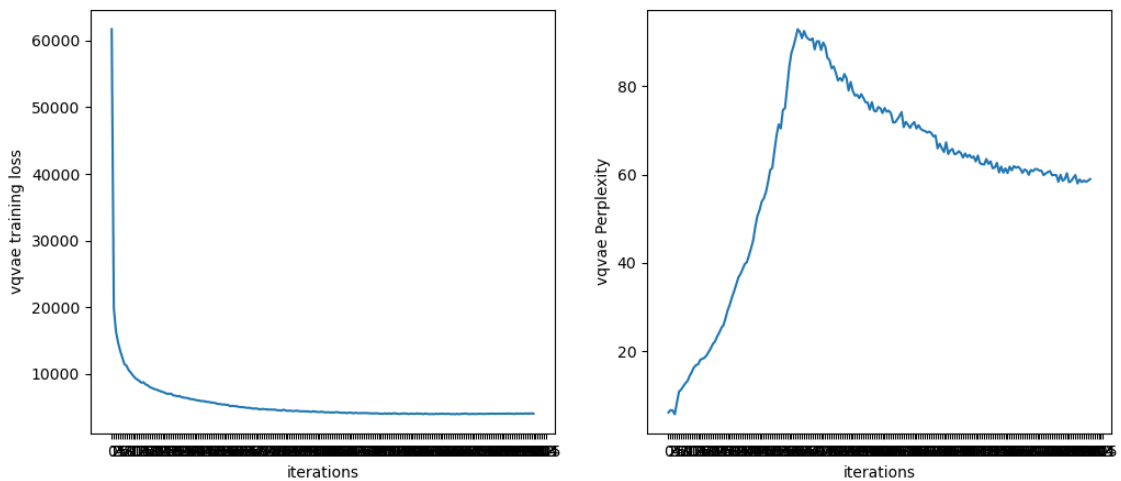
Results with VQVAE in DLStudio

- I will now show results obtained with the VQVAE class in DLStudio using the same `PurdueShapes5GAN` dataset of 20,000 images that I used earlier for the results with the Autoencoder and VAE classes.
- **The results shown are for the case when previously unseen images were presented at the input to the Encoder in VQVAE.**
- The parameters for the training routine are as shown below. **Note that I have not carried out any hyperparameter tuning for this exercise.** The parameters used are the same as for the previous results with Autoencoder and VAE except for the number of epochs that was 100 and for the new parameters "**Number of codebook vectors**" and "**Commitment cost**".

```
Input image size:  64x64
Encoder output size: 16x16
Encoder output channels (the latent dimension): 256
Number of codebook vectors: 512
Commitment Cost: 0.25
Learning rate: 1e^-4
Number of epochs: 100
Total number of training iterations: 82,600
```

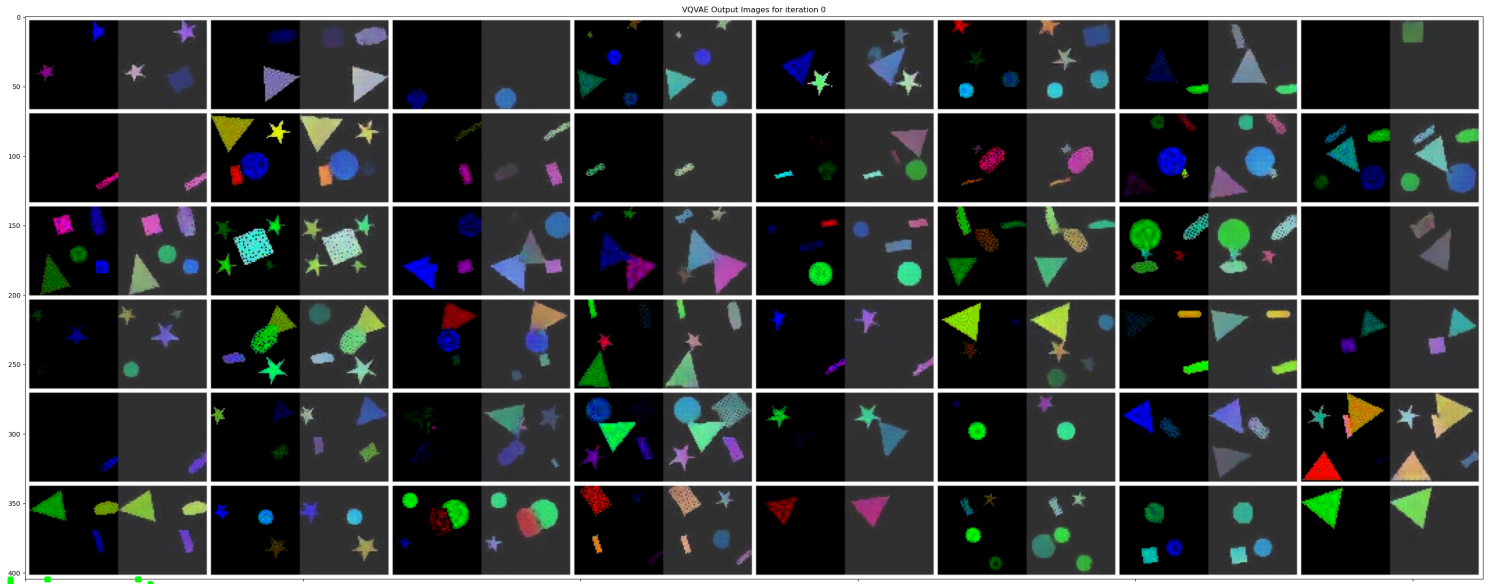
VQVAE Results: Training Losses and Perplexity

- Shown below are the training-loss-vs-iterations plot and a plot of perplexity-vs-iterations.
- The calculation of perplexity is shown in Line (S) on Slide 75. As mentioned in Slide 70, perplexity is defined as 2 to the power of the entropy associated with the codebook vectors. **Roughly speaking, it translates into how many of the codebook vectors are actually being used on the average**.



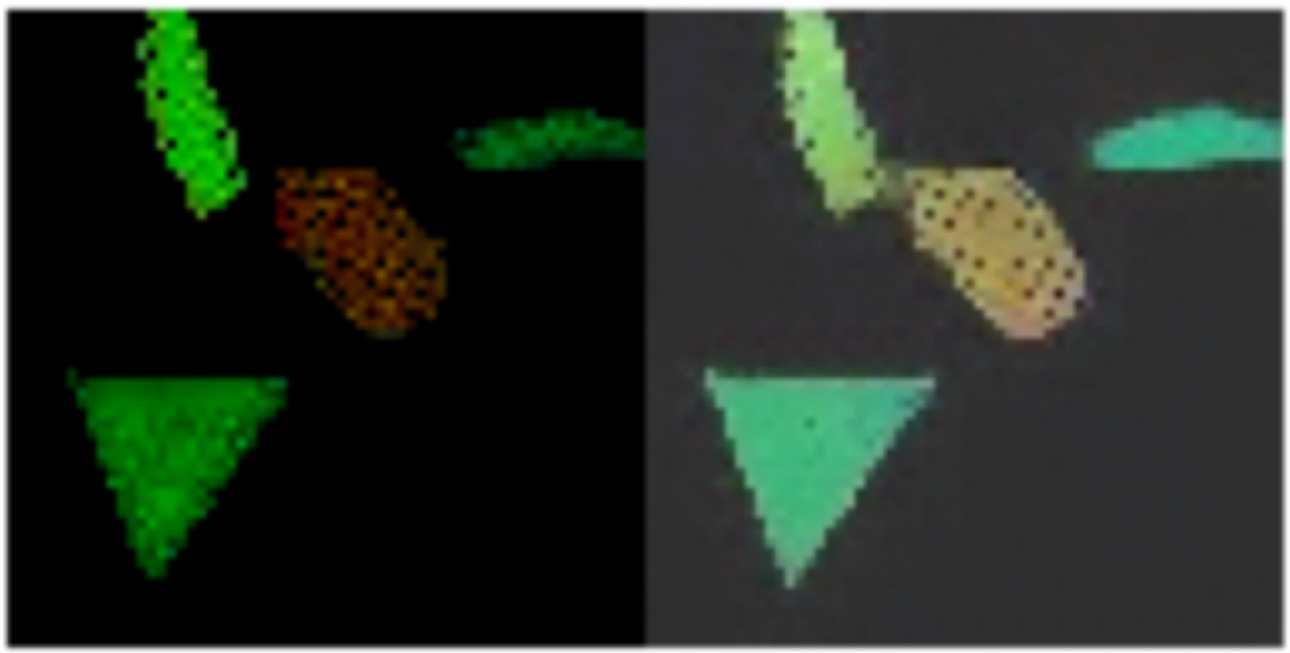
VQVAE Results: Decoder Output on Unseen Inputs (contd.)

- Shown below is an example of a batch of images at the output of the VQVAE Decoder.
- As with similar depictions for Autoencoder and VAE, inside each cell, the left-half shows the input to the Encoder and the right-half the output of the Decoder. In order to get a sense of the precision achieved with codebook based learning, see the next slide.



VQVAE Results: Decoder Output on Unseen Inputs (contd.)

- Shown below is an exploded view of one of the cells in the output batch shown on the previous slide.
- Notice that the very fine detail in the form of the randomly located tiny holes inside the patterns. The Decoder is able to reconstruct them now. That would not be the case if you exploded in a similar manner the output of the VAE network.



Figure

Outline

VQGAN — Codebook Learning and Autoregressive Modeling

- VQGAN made two significant advances over VQVAE: For one, **it placed the Encoder-Decoder network of the VQVAE inside a GAN based framework in which the Encoder-Decoder pair is treated as the Generator in a GAN**. **It is this part of the overall framework that is referred to as the VQGAN.** The rest of the framework is about autoregressive modeling of the images using the learned codebook vectors.

- What I have described above would be along the same lines as the material presented in Section 10 entitled "Using Adversarial Learning for Non-Generative Tasks" of my Week 11 lecture.

- The second significant advance made in VQGAN is to show that the quality of the embedding vectors produced by the GAN part of VQGAN and the codebook vectors learned is so high that you can use them for generating large images of high quality — **larger in size than any that had been generated prior to this paper**.

VQGAN — Codebook Learning and Autoregressive Modeling (contd.)

- Just as what is accomplished with a tokenizer for text processing, VQGAN gives you a token vocabulary of a fixed size for representing images. In that sense, the learned codebook vectors are the tokens for the case of images. As is the case with text processing tokens, the VQGAN tokens have integer indices associated with them and, for each integer index, an embedding vector.
- The next slide shows the synthesized images from the VQGAN paper.

VQGAN — Some Example Results by the Original Authors

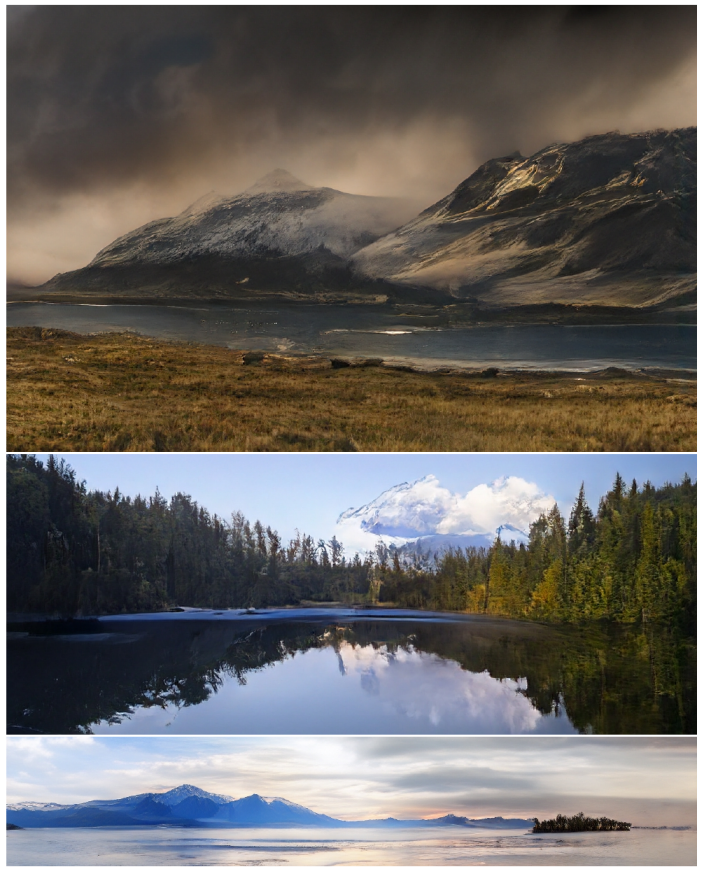


Figure: Images synthesized by VQGAN. The top image is of size 1280 × 832, the middle of size 1024 × 416 and the bottom of size 1280 × 240. This figure is from the VQGAN paper `https://arxiv.org/pdf/2012.09841`

VQGAN — Codebook Learning and Autoregressive Modeling

- The next figure explains the "architecture" of VQGAN. You again have an Encoder-Decoder structure, as in VQVAE, except for the difference that it serves as the Generator inside a GAN.

- As is always the case with a GAN, the Discriminator's job is to become an expert at recognizing the images in the training dataset and, at the same time, to disbelieve to the maximum extent possible the output of the Generator.

- What you see inside the large box labeled "VQGAN" is very much like the architecture of VQVAE. The output of the Encoder would be a small-sized array with a large channel dimension. As for VQVAE, you think of each element of the output array along with its channel axis an an embedding vector. Subsequently, you would replace each embedding vector at the output of the Encoder with the closest codebook vector.

VQGAN — Codebook Learning and Autoregressive Modeling (contd.)

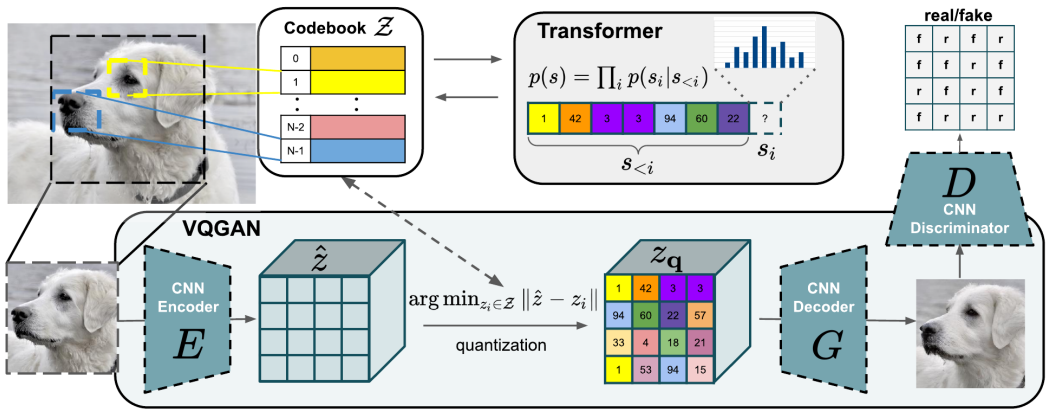


Figure: From the VQGAN paper `https://arxiv.org/pdf/2012.09841`

VQGAN — Codebook Learning and Autoregressive Modeling (contd.)

- We represent the codebook vectors by $\mathcal{Z} = \{z_k\}_{k=1}^{K}$ with each vector $z_i \in \mathcal{R}^{n_z}$, where n_z is the dimensionality of the codebook vectors.

- If x represent the input image fed into the Encoder and x' the image at the output of the Decoder, for the loss for the operations inside the VQGAN box in the figure on the previous slide, we write the same expression as for the VQVAE loss on Slide 66:

$$\mathcal{L}_{VQ}(\mathcal{E}, \mathcal{G}, \mathcal{Z}) \;=\; ||x - x'||_2^2 \;+\; \Big|\Big|sg[z_{enc}(x)] - e\Big|\Big|_2^2 \;+\; \beta\Big|\Big|z_{enc}(x) - sg[e]\Big|\Big|_2^2 \tag{23}$$

  where the symbol $\mathcal{E}$ stands for the Encoder, $\mathcal{G}$ for the Decoder, and $\mathcal{Z}$ for the codebook defined in the first bullet above.

- To the above loss, we now add the Adversarial Loss defined by (See Section 10 of my Week 11 lecture):

$$\mathcal{L}_{GAN}(\{\mathcal{E}, \mathcal{G}, \mathcal{Z}\}, D) \;=\; \Big[\log D(x) \;+\; \log(1 - D(\hat{x}))\Big] \tag{24}$$

VQGAN — Codebook Learning and Autoregressive Modeling (contd.)

- Combining the two losses shown on the previous slide, we write the following expression for the complete objective function for training the VQGAN:

$$\mathcal{L}_{vqgan} = \min_{\mathcal{E},\mathcal{G},\mathcal{Z}} \max_{\mathcal{D}} \Big[\mathcal{L}_{VQ}(\mathcal{E},\mathcal{G},\mathcal{Z}) + \lambda \mathcal{L}_{GAN}(\{\mathcal{E},\mathcal{G},\mathcal{Z}\},D) \Big] \tag{25}$$

with

$$\lambda = \frac{\nabla_{G_L}[\mathcal{L}_{rec}]}{\nabla_{G_L}[\mathcal{L}_{GAN}] + \delta} \tag{26}$$

- The symbol ∇_{G_L} means the partial of its argument with respect to the last layer of the Decoder. $\mathcal{L}_{rec}$ stands for the reconstruction loss as given by the first term in Eq. (23) and δ is set to a small number like 10^{-6} for numerical safety.

VQGAN — Codebook Learning and Autoregressive Modeling (contd.)

- VQGAN is trained two separate stages. First you train just the VQGAN part of the overall network (shown in the big box labeled "VQGAN" in the figure on Slide 86). This is based on the min-max optimization criterion defined in Eq. (25).
- In the second stage, you train the transformer based autoregressive modeler. Here you treat the integer index labels associated with the code vectors as the tokens in a sentence. **You want the transformer to predict the next integer index while using for the ground truth the integer index values that characterize the codebook vectors chosen for the input image**.
- The goal here is very much like in, say, language translation. As with the prediction of the next token in the target language for the case of language translation, we now want to predict the integer index for the next codebook vector to be used.

VQGAN — Codebook Learning and Autoregressive Modeling (contd.)

- **It is important to realize that this autoregressive modeling over the codebook vector index values is done entirely in the Latent Space**, which is the space spanned by the embeddings at the output of the Encoder and also by the learned codebook vectors.