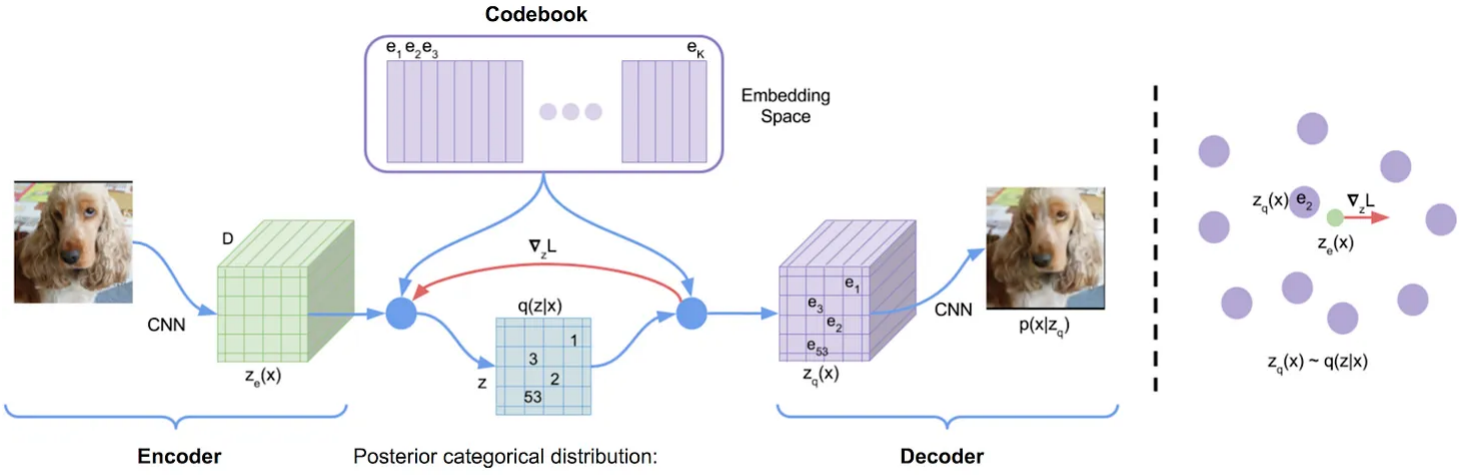
Understanding VQ-VAE (DALL-E Explained Pt. 1)

MACHINE LEARNING AT BERKELEY
9 FEB 2021

By Charlie Snell



$$q(\mathbf{z} = \mathbf{e}_k|\mathbf{x}) = \begin{cases} 1 & \text{if } k = \arg\min_i \|\mathbf{z}_e(\mathbf{x}) - \mathbf{e}_i\|_2 \\ 0 & \text{otherwise.} \end{cases}$$

Like everyone else in the ML community, we've been incredibly impressed by the results from OpenAI's DALL-E. This model is able to generate precise, high quality images from a text description. It can even produce creative renderings of objects that likely don't exist in the real world, like "an armchair in the shape of an avocado".

To celebrate the release of DALL-E, we're publishing a series of blog posts explaining the key components of this model.

This first blog post in the series will cover VQ-VAE, which is the component that allows DALL-E to generate such a diverse and high quality distribution of images.

In the next post we will cover transformers for multi-modal, autoregressive generation: the part of DALL-E that learns the correlations between language and images, resulting in such creative and accurate outputs.

Note: this blog post assumes that you have some knowledge of deep learning and Bayesian probability.

TEXT PROMPT

an illustration of a baby daikon radish in a tutu walking a dog

AI-GENERATED IMAGES

Edit prompt or view more images↓

TEXT PROMPT

an armchair in the shape of an avocado [...]

AI-GENERATED IMAGES

Edit prompt or view more images↓

The Basics

VQ-VAE stands for Vector Quantized Variational Autoencoder, that's a lot of big words, so let's first step back briefly and review the basics.

We will first define the concept of a latent space and then define autoencoders, lastly we will review the basics of variational autoencoders before getting into VQ-VAE *(Note: if you feel like*

you already understand this stuff, feel free to skip to the next section, where we dive into the details of VQ-VAE).

Latent Spaces

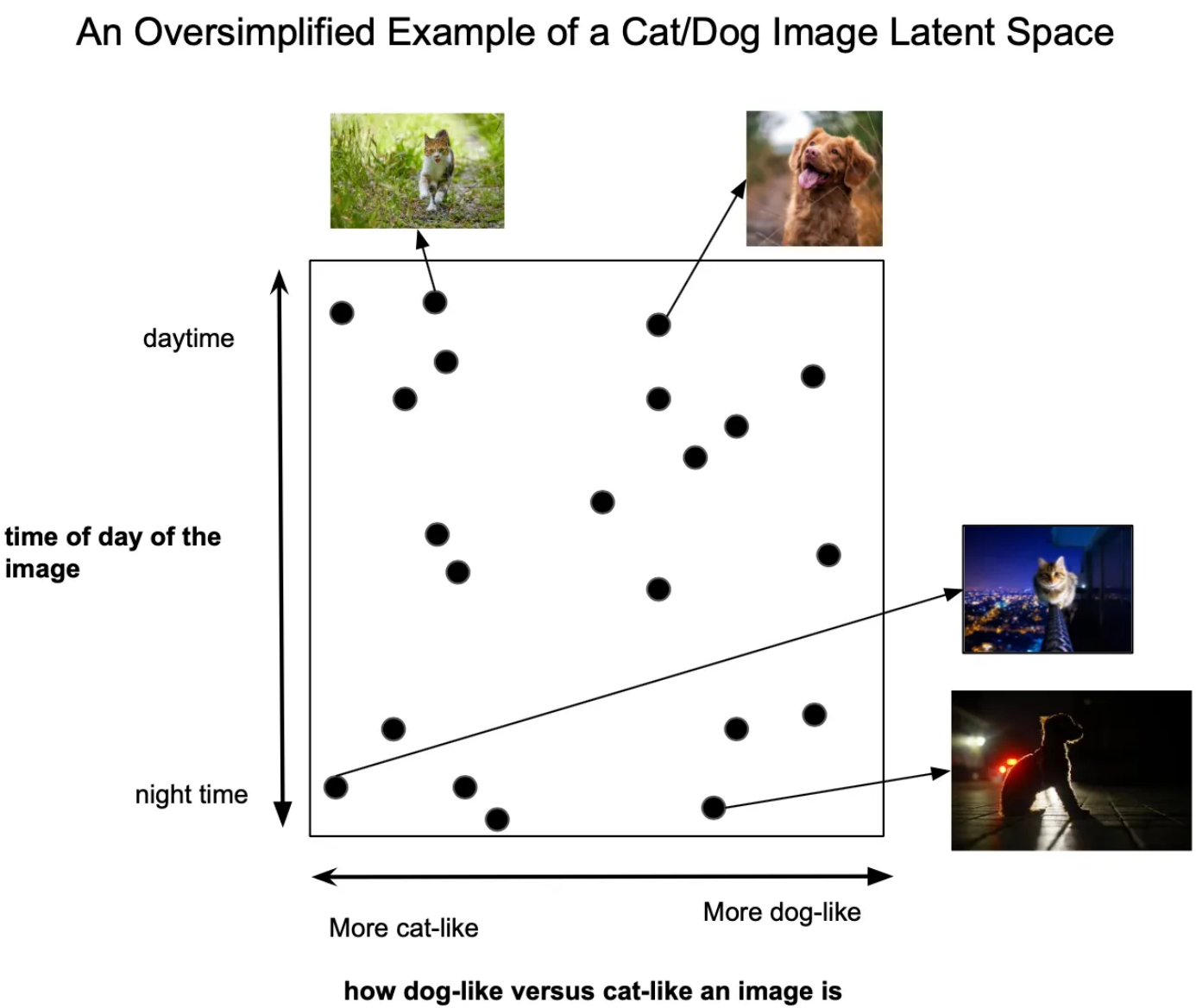
A latent space is some underlying "hidden" representation for a given distribution of raw data. To clearly understand this let's look at a simple example.

Imagine the raw data points are $x \in \mathbb{R}^n$, and they are noisily generated via a linear transform from some lower dimensional $z \in \mathbb{R}^m$ ($m<n$). Specifically x is generated as $x=Az+v$ where v is n-dimensional independent identically distributed gaussian noise and $A \in R^{n * m}$.

In general, we only get to see the raw data x; we do not get access to z, which is why it is referred to as a latent or "hidden" representation of the data. However, we would like to gain access to z because it is a more fundamental and compressed representation for the data. Additionally, this latent representation could be a more useful input to many other algorithms that need to use the data.

This specific setup, in which the raw data is a linear transformation of the latent space, is exactly what the classical unsupervised algorithm called "PCA" is designed for: PCA essentially tries to find the underlying z representation from the above example. This is great, but what if the latent representation had a more complicated, non-linear relationship with the raw data? For example, the figure below visualizes the kinds of high level information a more complex latent space could encode. In this case, PCA would fail to find the best latent representation. Instead we could use an autoencoder to find this more abstract latent space.

Note: A latent space does not need to be a continuous vector space, it could instead be a ***discrete*** *set of variables, which is the type of latent space that VQ-VAE tries to find. But before we get to that, let's first understand vanilla autoencoders.*



Autoencoders

An autoencoder is an unsupervised learning technique that uses neural networks to find non-linear latent representations for a given data distribution. The neural network consists of two parts: an **encoder** network, $z=f(x)$, and a **decoder** network, $\hat{x}=g(z)$.

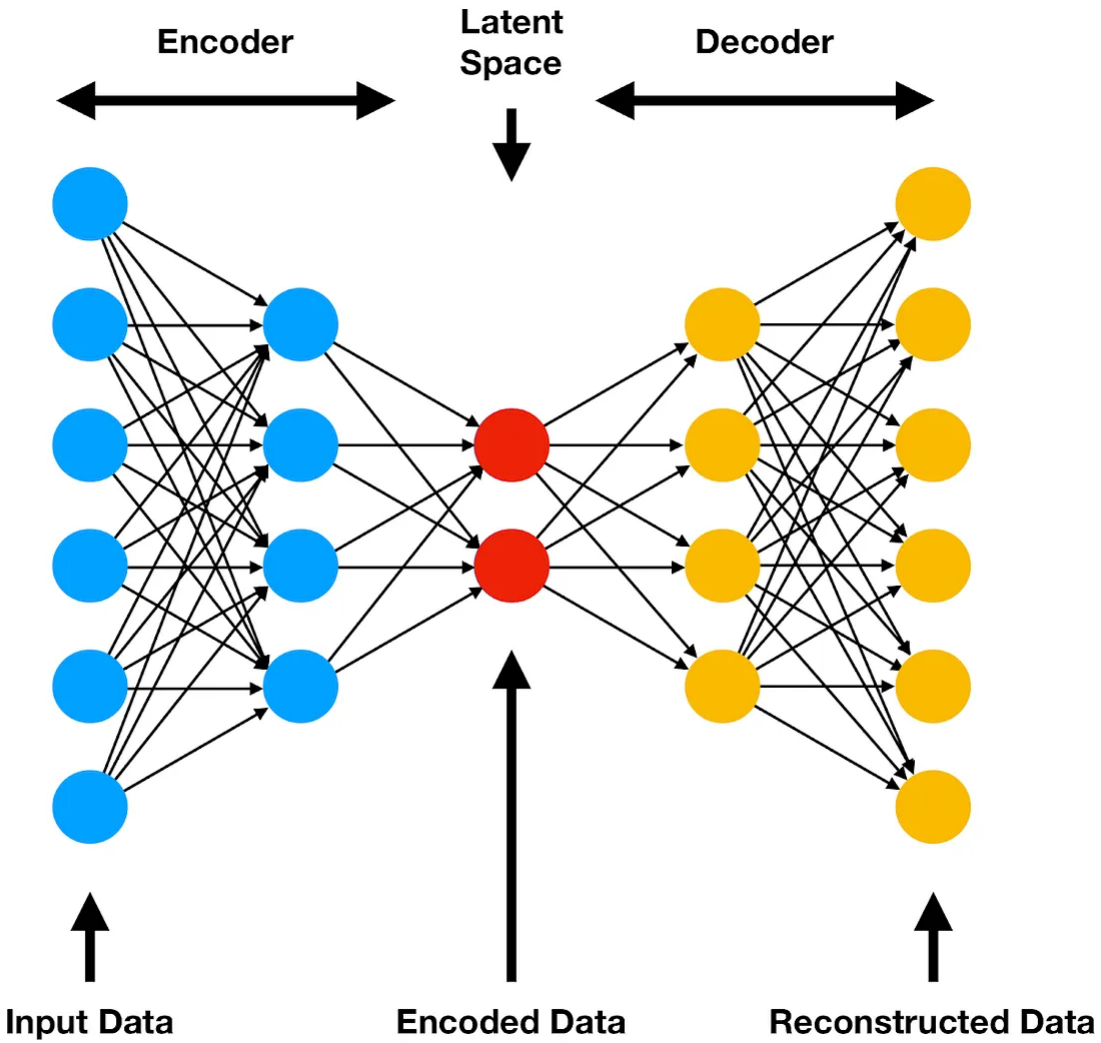
Here x is the input data, z is the latent vector representation, and $\hat{x}$ is the "reconstruction" of x from the latent space. $f(x)$ and $g(z)$ are both neural networks. Putting the two parts together, the entire model can be described by the relation $\hat{x}=g(f(x))$ (see the figure below).

Ideally, the decoder should be able to accurately reconstruct the raw data from the encoder's latent representation. If the model is able to learn such a reconstruction, then we can assume that our latent space represents the data well. To actualize this objective, we train the model with some sort of reconstruction loss between x and $\hat{x}$ (i.e.

$$||x - \hat{x}||_2^2 \(||x-\hat{x}||_2^2\)$$

or

$$log(p(x|z)) \(\log(p(x|z))\)$$



It's also important to note that z should always have a lower dimension than x. The whole point, after all, is to encode a compressed representation of the data, such that the algorithm is forced to find the most essential components of the raw data.

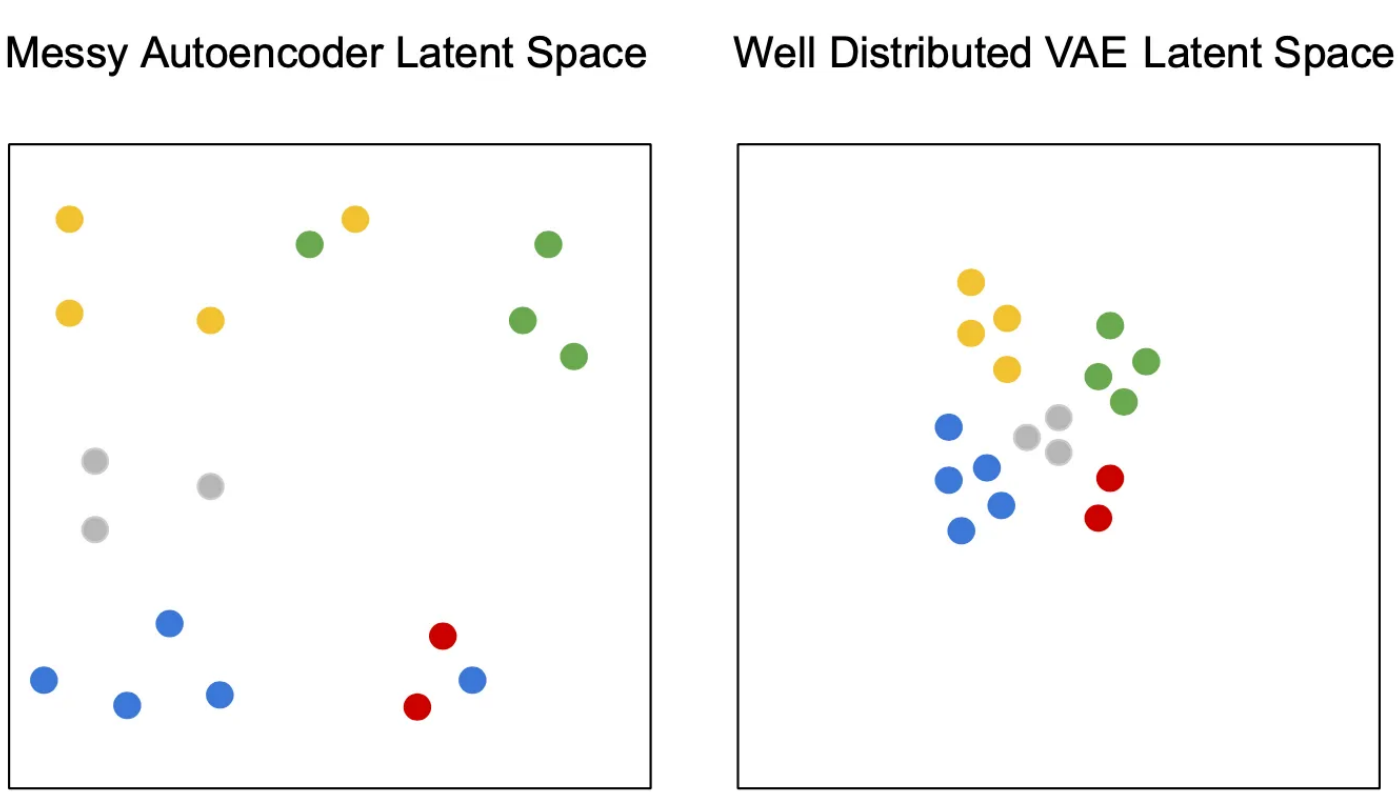
This compressed, latent part of the autoencoder is often also referred to as the **bottleneck** of the network because it is squeezing the data into a much smaller space.

Variational Autoencoders (VAE)

(NOTE: I will not go into all the mathematical details of variational autoencoders in this blog post because I'd need to devote an entire blog post to VAEs to really do them justice. Luckily enough, many such blog posts already exist, and I can personally recommend this excellent blog post by Jaan Altosaar if you want a more detailed treatment of the topic.)

Now that we understand the essentials of autoencoders, let's look at what makes a Variational Autoencoder different. The key difference comes from the structure imposed on the latent space of VAEs. Ideally we would want our latent space to lump semantically similar data points next to each other and to place semantically dissimilar points far apart. Preferably the majority of the data distribution should make up a fairly compact volume in latent space and not extend to infinity.

The main issue with vanilla autoencoders is that the learned latents do not necessarily have either of these properties! The model can learn any latent space it wants, so it often ends up memorizing individual data points by placing them in their own far out pockets of latent space. The figure below visualizes these issues with autoencoder latent spaces and compares them to that of VAEs.



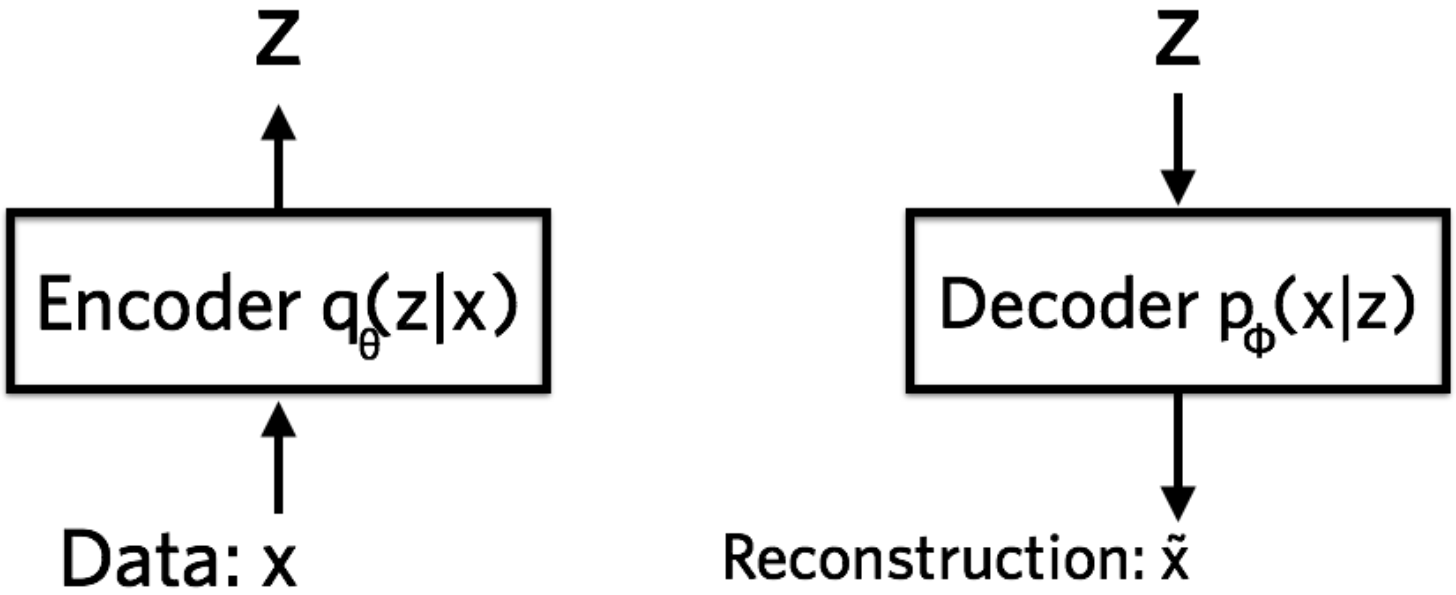
Variational autoencoders overcome this problem by enforcing a probabilistic prior on the latent space. To formalize this, consider our latent space *z* as a random variable. First let's enforce a **prior** *p(z)* on our latents, in most VAEs this is typically just a standard gaussian distribution N(0,1). Given a raw datapoint *x*, we also define a **posterior** for the latent space as *p(z|x)*.

Our end goal is to compute this posterior for the data, which we can express using bayes rule as

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)}$$ \(p(z|x)=\frac{p(x|z)p(z)}{p(x)}\)

Unfortunately, because we are working with continuous variables, *p(x)* is a computationally intractable quantity. In order to make things computable, we restrict our approximation of the posterior to a specific family of distributions: independent gaussians. Call this approximated

distribution *q(z/x)*. Now our goal becomes to minimize the KL divergence between the true posterior and this approximated form.



I'll spare the mathematical details, but it turns out that this objective can be computed by minimizing the loss function

$$-E_{z\sim q(z|x)}[log(p(x|z))] + KL(q(z|x)||p(z))\(-E_{z \sim q(z|x)}[log(p(x|z))]+KL(q(z|x) || p(z))\)$$

In this case *q(z/x)* is approximated by the encoder and *p(x/z)* is represented by the decoder.

The VAE loss actually has a nice intuitive interpretation, the first term is essentially the reconstruction loss, and the second term represents a regularization of the posterior. The posterior is being pulled towards the prior by the KL divergence, essentially regularizing the latent space towards the gaussian prior. This has the effect of keeping the latent distribution compactly distributed around 0.

Given sufficiently expressive neural networks, the VAE latent space can fit complex data distributions very neatly. Generally you can expect to see smooth transitions between the different types of data points in the latent space. For example, if a VAE is trained on MNIST, you could expect a cluster for the 6's and a separate cluster for the 5's. If you go between the two clusters, you should get a digit that looks like a weird mix of a 5 and a 6.

Basics Summary

- Latent spaces are compressed representations of data, which often emphasize the most important and semantically interesting features of the data in their representation.
- A learned latent representation can be useful for many downstream algorithms.
- Autoencoders are a method for finding latent spaces that are complicated non-linear functions of the data.
- Variational autoencoders add a prior to the autoencoder latent space. This gives the learned latent space some very nice properties (i.e. we can smoothly interpolate the data distribution through the latents).

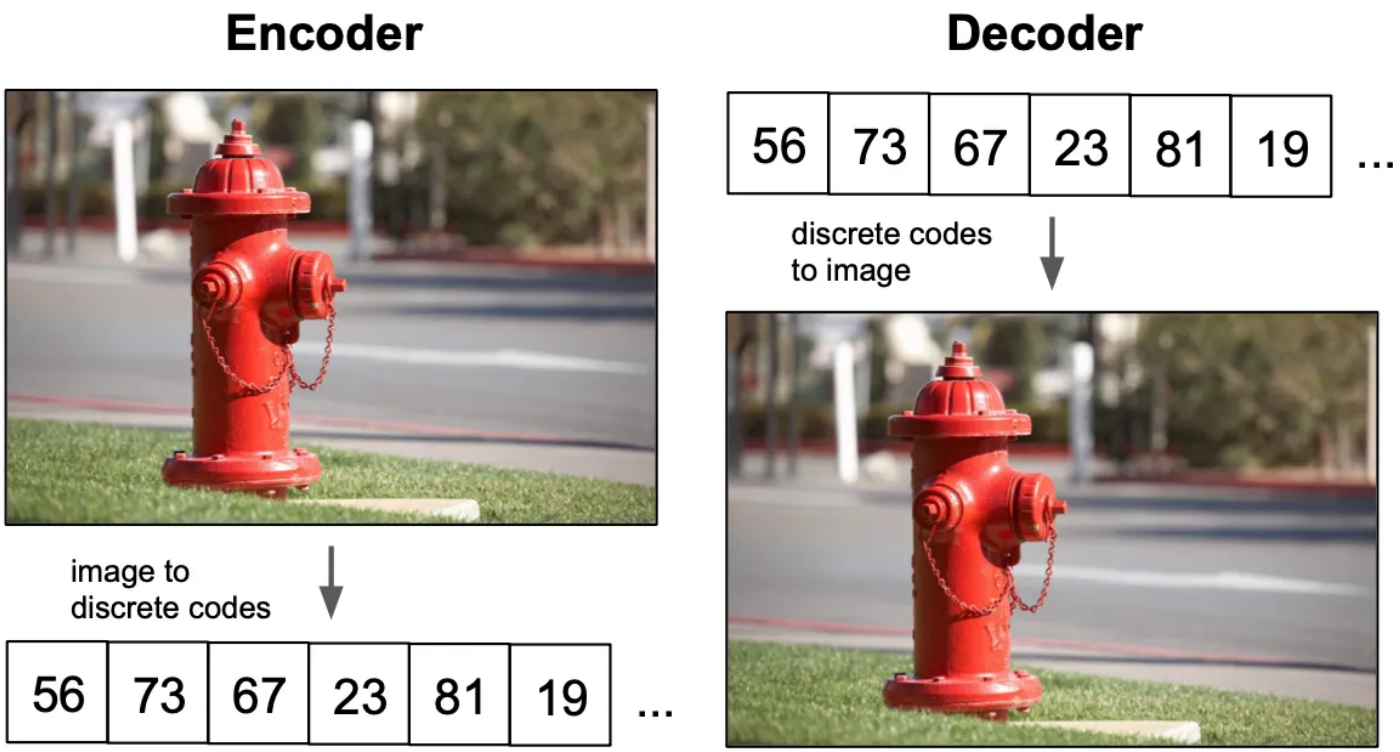
Discrete Spaces

Now that we have a handle on the fundamentals of autoencoders, we can discuss what exactly a VQ-VAE is. The fundamental difference between a VAE and a VQ-VAE is that VAE learns a continuous latent representation, whereas VQ-VAE learns a discrete latent representation.

So far we have seen how continuous vector spaces can be used to represent the latents in an autoencoder. But latents do not necessarily need to be continuous vectors, it really just needs to be some numerical representation for the data. And one such, potentially desirable, alternative to a vector space is a discrete representation.

In general, a lot of the data we encounter in the real world favors a discrete representation. For example, human speech is well represented by discrete phonemes and language. Additionally, images contain discrete objects with some discrete set of qualifiers. You could imagine having one discrete variable for the type of object, one for its color, one for its size, one for its orientation, one for its shape, one for its texture, one for the background color, one for the background texture, etc…

In addition to the representation, there are a number of algorithms (like transformers) that are designed to work on discrete data, so we would like to have a discrete representation of the data for these algorithms to use.



Clearly discrete latent representations can be useful, but how do we learn such a representation? This might seem very challenging at first because in general discrete things do not play too

nicely with deep learning. Luckily enough, VQ-VAE manages to make deep learning work for the task with just a few tweaks to the vanilla autoencoder paradigm.

Quantizing Autoencoders

VQ-VAE extends the standard autoencoder by adding a discrete **codebook** component to the network. The codebook is basically a list of vectors associated with a corresponding index. It is used to quantize the bottleneck of the autoencoder; the output of the encoder network is compared to all the vectors in the codebook, and the codebook vector closest in euclidean distance is fed to the decoder.

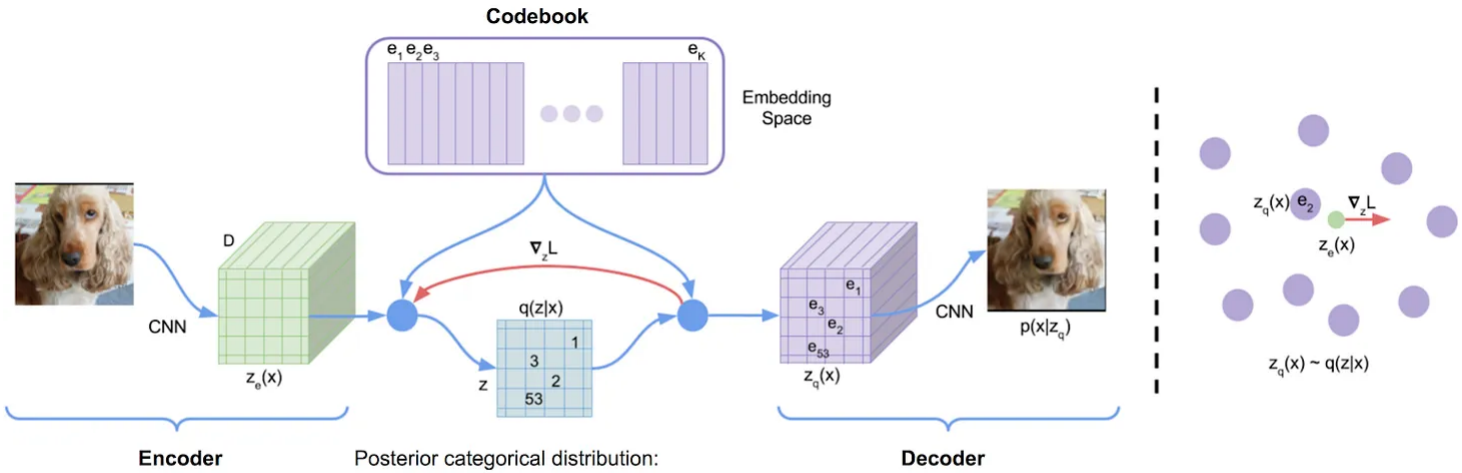
Mathematically this is written as

$$z_q(x) = \text{argmin}_i ||z_e(x) - e_i||_2 \(z_q(x)=\text{argmin}_i ||z_e(x)-e_i||_2\)$$

where $z_e(x)$ is the encoder vector for some raw input x, e_i is the ith codebook vector, and $z_q(x)$ is the resulting quantized vector that is passed as input to the decoder.

This argmin operation is a bit concerning, since it is non-differentiable with respect to the encoder. But in practice everything seems to work fine if you just pass the decoder gradient directly through this operation to the encoder (i.e. set its gradient to 1 wrt the encoder and the quantized codebook vector; and to 0 wrt all other codebook vectors)

The decoder is then tasked with reconstructing the input from this quantized vector as in the standard autoencoder formulation.



$$q(\mathbf{z} = \mathbf{e}_k|\mathbf{x}) = \begin{cases} 1 & \text{if } k = \arg\min_i \|\mathbf{z}_e(\mathbf{x}) - \mathbf{e}_i\|_2 \\ 0 & \text{otherwise.} \end{cases}$$

Producing Multiple Codes

You might be puzzled by the fact that there is such a restricted set of vectors that can be fed to the decoder (just the set of codebook vectors). How could one ever expect to generate the huge quantity and diversity of possible images when the decoder can only accept the set of codebook vectors as input? We would need a unique discrete value for each training point to be able to reconstruct all the data. And if this is actually the case, then wouldn't the model effectively be memorizing the data by mapping each training point to a different discrete code?

This would be a very valid concern if the encoder were to output just one vector, but in actual VQ-VAEs the encoder usually produces a series of vectors. For instance, with images, the encoder might output a 32x32 grid of vectors, each of these are quantized and then the entire grid is fed to the decoder. Or similarly, for processing audio, the encoder could output a long 1-d sequence of vectors. All vectors are quantized to the same codebook, so the number of discrete values doesn't change, but by outputting multiple codes we are able to exponentially increase the number of datapoints our decoder can construct.

For instance, imagine we are working with images, we have a codebook of size 512, and our encoder outputs a 32x32 grid of vectors. In this case, our decoder can output $512^{32 * 32} = 2^{9216}$ distinct images! Just to make it clear how enormous this number is, I have computed it in python in the figure below... it is truly an unfathomably large number. There is absolutely no way anyone could ever visualize every possible output from this model. The size of the discrete space really is no longer a problem here.

```
>>> 2**9216
19608304352668943221565390150839095508718758603821183350857270552532778258365389731320360283413868778900635905644638661218272958143533989172469170582635293991946517116704191729729501018887561528561979061199679711207419657828835997045123149330953138221382551826519295780148184830257628047540965035233610795226453345405219910376637917515019627445104233002103505877419229870321413438916495306644844403314648330162975599838641983346566196928395633881358885152586414880905288190861933085649543437392668396385315386148827675870309205530918987024321892964715479723356659563955917692507818555791102527436188657152984768015015658043976521716974644725768202868002615722161552077691074830207914237787332385516151134154538815535640496908262998162372524787088445668571483525100729325306043625053615319754206645337535840178375118554980348759184208884074025277017561382997989517664989597201036493533942611858781362041975130491088788596943191686787350245720205239132623344250852954199432990526563834984074418657506792139563985304386089991197401786118306670964413214091507872404478948072032706538424972974342475514696669662363769670317522504301321595314030186526060470483755984918351931338079855506119859919057156315324416253715694506455416576875443964618044437350001615176392070825473356061907044289359249724517004354937124618746826852641814395422822859391213290094187426173186094868146860706106553136564619410036690971415080725176074233012326456669338931707273995797901766284405882787335569141996562436460195267647275147149205610463929384078839689426103532822156526714375623697059664372632794012557279683693447972246065202562591657030121882456014169969325909542093386657044847559644017983857580367228839444750225410532011400775939182327993840869025967095413647307769315594858391481658152666544922121171823727490607401746158585304788755383045205191437451775169458678544199478704738789501636360142632818276724048129802203207935437567380783964186435163926788534617028326061117919367604484844921356916923523247449265424511105098478836814747526992386751806425252184207489128728792824943689655074807800515853112660139242197274266229061952680327438328671815163784513886971727602729888921523795732674186288508167924341719459948962807382366847562516814695305800945127915287319494658919711863800069357890734737261070540370708492011266113226697383353438037641361127454887451974254619866043536891164185748638999677095990281908623078798766977635579970492589382848507468687622721860884553653265717180761954414100115661084883983062285289285346636378604214730100480657010819951184685497866500762087892065897068998925731115031743362283408922762884308028995976800393727184116644770851546262263121276281051781132606813765311962734045305226205027320910150499078954883386617195179985788372478254401447804530942017536
```

Of course the model could still memorize the training data, but with the right inductive biases baked into the encoder (i.e. using a conv-net for images) and the right structure for the latent codes (i.e. a 32x32 grid for images), the model should learn a discrete space that represents the data well.

Learning the Codebook

Just like the encoder and decoder networks, the codebook vectors are learned via gradient descent. Ideally our encoder will output vectors that are generally close to one of the learned codebook vectors. There is essentially a bi-directional problem here: learning codebook vectors that align to the encoder outputs and learning encoder outputs that align to a codebook vector.

Both of these problems are solved by adding terms to the loss function. The entire VQ-VAE loss function is

$\log(p(x|q(x))) + ||\text{sg}[z_e(x)] - e||_2^2 + \beta||z_e(x) - \text{sg}[e]||_2^2$\(\text{log}(p(x|q(x)))+||\text{sg}[z_e(x)]-e||_2^2+\beta||z_e(x)-\text{sg}[e]||^2_2\)

Here we use the same notation from the previous section, and sg[x] stands for “stop gradient”, it prevents the gradient from flowing through that part of the equation.

The first term is just the standard reconstruction loss. The second term is the codebook alignment loss, whose goal is to get the chosen codebook vector as close to the encoder output as possible. There is a stop gradient operator on the encoder output because this term is only intended to update the codebook. And the third term is similar to the second, but it puts the stop gradient on the codebook vector instead because it is meant to solve the inverse problem of getting the encoder output to commit as much as possible to its closest codebook vector. This term is called the codebook commitment loss, and its importance to the overall loss is scaled by the tunable hyperparameter β. Of course these last two terms are averaged over each quantized vector output from the model, if there is more than one.

This loss function basically completes our description of the VQ-VAE.

With this, we can fully train a VQ-VAE capable of reconstructing a diverse set of images that are almost imperceptibly different from the original images as seen in the figure below. We can also train VQ-VAEs to reconstruct other modalities like audio or video.

Figure 2: Left: ImageNet 128x128x3 images, right: reconstructions from a VQ-VAE with a 32x32x1 latent space, with K=512.

Interpreting VQ-VAE in the Language of VAEs

Now that we know how VQ-VAEs work, let's try to understand them in the context of the probabilistic formalism that VAEs provide. Recall that VAEs enforce a pre-defined prior on the latent space $p(z)$, and the encoder is tasked with approximating the posterior distribution of the latents $p(z|x)$. Lastly, the decoder approximates the reconstruction from the latent space $p(x|z)$.

During training, VQ-VAE assumes a uniform prior over all the latent codes, so all latents are considered equally likely. Additionally the posterior is deterministic in that $p(z|x)=1$ if $z=z_q(x)$ and 0 otherwise. Lastly, the decoder distribution $p(x|z)$ is unchanged and learned as usual. Using these definitions, it can easily be shown that the KL divergence term from the VAE loss becomes a constant and therefore can be dropped for VQ-VAEs, leaving just the reconstruction term. The VQ-VAE reconstruction loss is therefore consistent with VAE formalism.

Learning the Prior

Once a VQ-VAE is fully trained, we can abandon the uniform prior imposed at training time and learn a new, updated prior $p(z)$ over the latents. If we learn a prior that accurately represents the distribution of discrete codes, we will be able to generate new data from the distribution by sampling from this prior and feeding the samples to the decoder (see the figure below for original VQ-VAE samples).

Training the prior also has compression benefits. If the distribution of latents is non-uniform, then the bits representing the sequence of latents can be further compressed by applying standard huffman or arithmetic coding to the prior distribution.

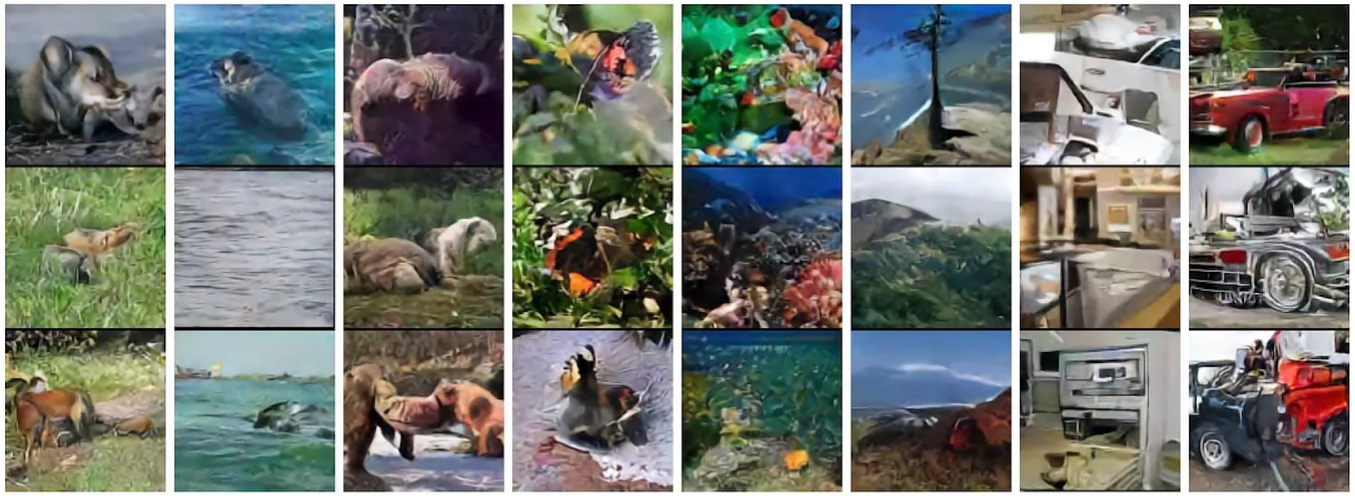


Figure 3: Samples (128x128) from a VQ-VAE with a PixelCNN prior trained on ImageNet images. From left to right: kit fox, gray whale, brown bear, admiral (butterfly), coral reef, alp, microwave, pickup.

Assuming the encoder outputs a sequence of latent codes for each datapoint, we can use any autoregressive model we want (i.e. RNN or transformer) to train the prior. The autoregressive factorization is, given all previous latent codes in the sequence, predict the next one. This breaks the latent distribution down as

$p(z) = p(z_1)p(z_2|z_1)p(z_3|z_1, z_2)p(z_4|z_1, z_2, z_3)\ldots\backslash$
(p(z)=p(z_1)p(z_2|z_1)p(z_3|z_1,z_2)p(z_4|z_1,z_2,z_3)...\)

where z_i is the ith latent in the sequence.

If we are working with a 1-d signal like audio, casting the problem into an autoregressive form is very simple: just predict the next latent in the 1-d sequence encoded from the audio. This approach assumes that the encoder is configured such that later latents in the sequence are representative of later sections of audio (this property should be expected if using convolutions to encode). For images we can cast the problem similarly, by first rolling out the 32x32 grid of

latents into a 1-d sequence, such that the sequence goes from top left to bottom right, we can then apply autoregressive learning to this sequence.

State Of the Art

This paradigm of training a VQ-VAE and then learning an updated prior, is the exact formula that OpenAI has successfully used in several of their groundbreaking recent releases. OpenAI's jukebox, a model that is able to generate original songs from raw audio, trains a VQ-VAE on audio and then uses a transformer trained on the latents to generate new audio samples. Their recently released text to image model, DALL-E, involves a transformer that takes as input both the embeddings from text and the latent codes from a VQ-VAE trained on images. Of course, some of these works are using a few additional tricks that are outside the scope of this blog post, like hierarchical VQ-VAEs and different relaxations for making the VQ-VAE bottleneck differentiable.

a cube with the texture of cloud. a cube
a macro photograph of an orchid. a macro
painting of san francisco's mission dolores. a
a photo of an owl at sunrise in pop art style. a
a cross-section view of a strawberry. a cro
on a table. a stack of chips sitting on a
a large blue block. a small red block on a
a photo of a museum in cuba. a ph
a photo of a computer from the 70s.
image of a blue pineapple. a neon

VQ-VAEs can represent diverse, complex data distributions better than pretty much any other algorithm out currently. And since these models play so nicely with transformers, the generative possibilities can be scaled almost arbitrarily given a large enough compute budget (unfortunately, for state of the art results, this is a budget that very few individuals or even organizations can afford). For these reasons, I expect VQ-VAEs to remain a popular component in the deep learning ecosystem for quite a while.

The next blog post in this DALL-E Explained series, will more closely examine transformers for autoregressive modeling on both single-modal and multi-modal data.

Edit: part 2 is out now ! Check it out here.

3 years ago · 1 like · Machine Learning at Berkeley

References

- VQ-VAE paper: https://arxiv.org/pdf/1711.00937.pdf
- OpenAI DALL-E blog: https://openai.com/blog/dall-e/
- OpenAI Jukebox blog: https://openai.com/blog/jukebox/
- Comp Three Inc. Autoencoders blog: https://www.compthree.com/blog/autoencoder/
- Jaan Altosaar VAE blog: https://jaan.io/what-is-variational-autoencoder-vae-tutorial/

5 Likes

1 Comment

Write a comment...

Donghun Kim Donghun's Substack Apr 14, 2023

What a wonderful article for VQ-VAE. I appreciate the authors of this fantastic explanation.

There might be a typo in the first loss term in VQ-VAE. text{log}(p(x|q(x))) -> text{log}(p(x|z_q(x)))

LIKE REPLY SHARE