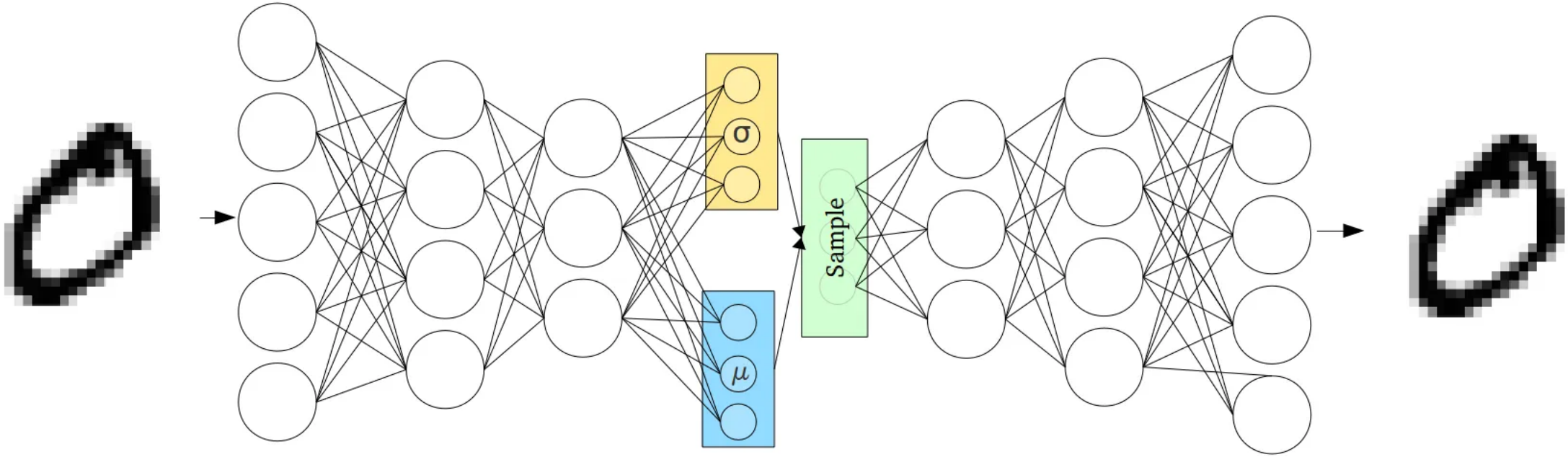


A Standard Variational Autoencoder

Intuitively Understanding Variational Autoencoders

And why they're so useful in creating your own generative text, art and even music

Irhum Shafkat · Follow

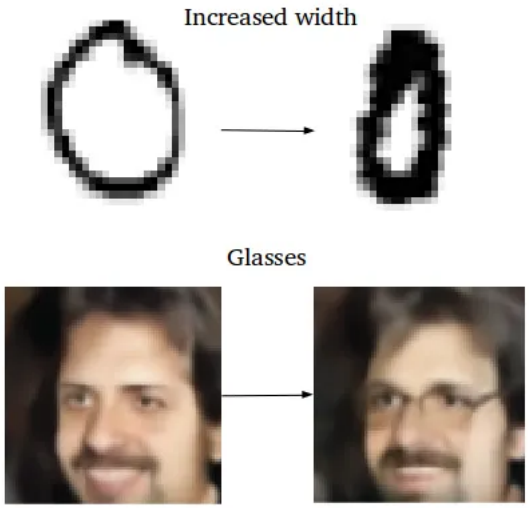
Published in Towards Data Science · 9 min read · Feb 4, 2018

In contrast to the more standard uses of neural networks as regressors or classifiers, Variational Autoencoders (VAEs) are powerful **generative** models,

now having applications as diverse as from generating fake human faces, to producing purely synthetic music.

This post will explore what a VAE is, the intuition behind why it works so well, and its uses as a powerful generative tool for all kinds of media.

But first, why VAEs?



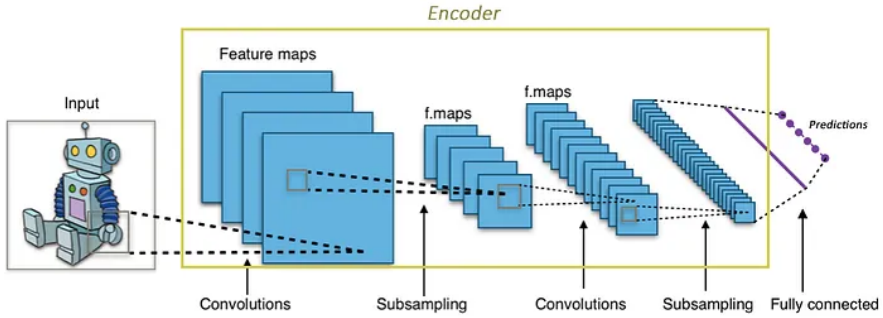
Exploring a specific variation of input data[1]

When using generative models, you could simply want to generate a random, new output, that looks similar to the training data, and you can certainly do that too with VAEs. But more often, you'd like to alter, or explore variations on *data you already have*, and not just in a random way either, but in a desired, *specific* direction. This is where VAEs work better than any other method currently available.

Decoding the standard autoencoder

An autoencoder network is actually a pair of two connected networks, an encoder and a decoder. An encoder network takes in an input, and converts it into a smaller, dense representation, which the decoder network can use to convert it back to the original input.

If you're unfamiliar with encoder networks, but familiar with Convolutional Neural Networks (CNNs), chances are, you already know what an encoder does.

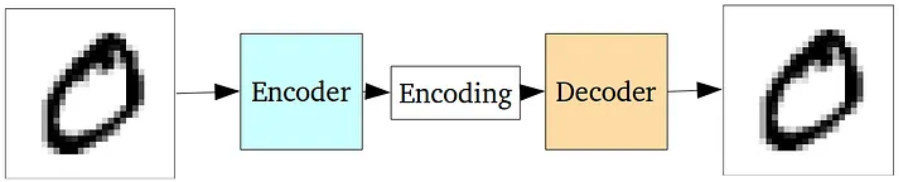


The encoder inside of a CNN[a]

The convolutional layers of any CNN take in a large image (eg. rank 3 tensor of size 299x299x3), and convert it to a much more compact, dense representation (eg. rank 1 tensor of size 1000). This dense representation is then used by the fully connected classifier network to classify the image.

The encoder is similar, it is simply is a network that takes in an input and produces a much smaller representation (the *encoding*), that contains enough information for the next part of the network to process it into the desired output format. Typically, the encoder is trained together with the other parts of the network, optimized via back-propagation, to produce encodings specifically useful for the task at hand. In CNNs, the 1000-dimensional encodings produced are such that they're specifically useful for classification.

Autoencoders take this idea, and slightly flip it on its head, by making the encoder generate encodings specifically useful for *reconstructing its own input*.



A standard Autoencoder

The entire network is usually trained as a whole. The loss function is usually either the mean-squared error or cross-entropy between the output and the input, known as the *reconstruction loss*, which penalizes the network for creating outputs different from the input.

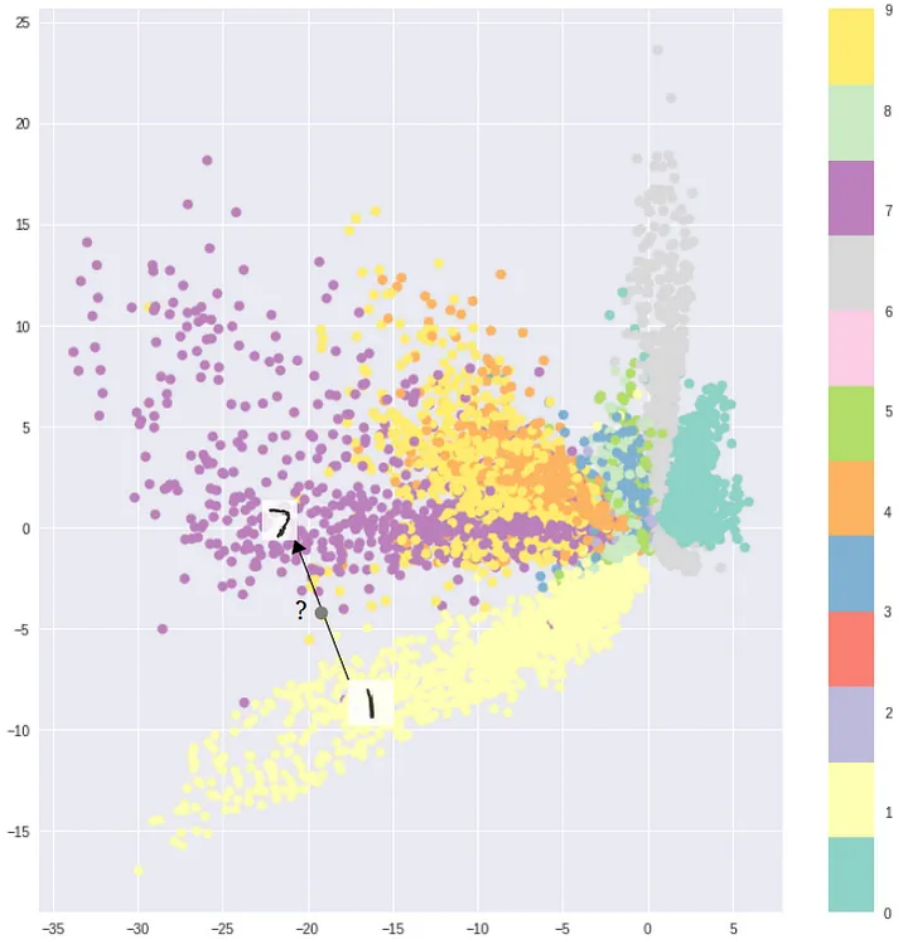
As the encoding (which is simply the output of the hidden layer in the middle) has far less units than the input, the encoder must choose to discard information. The encoder learns to preserve as much of the relevant information as possible in the limited encoding, and intelligently discard irrelevant parts. The decoder learns to take the encoding and properly reconstruct it into a full image. Together, they form an autoencoder.

The problem with standard autoencoders

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

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

Top highlight



Optimizing purely for reconstruction loss

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

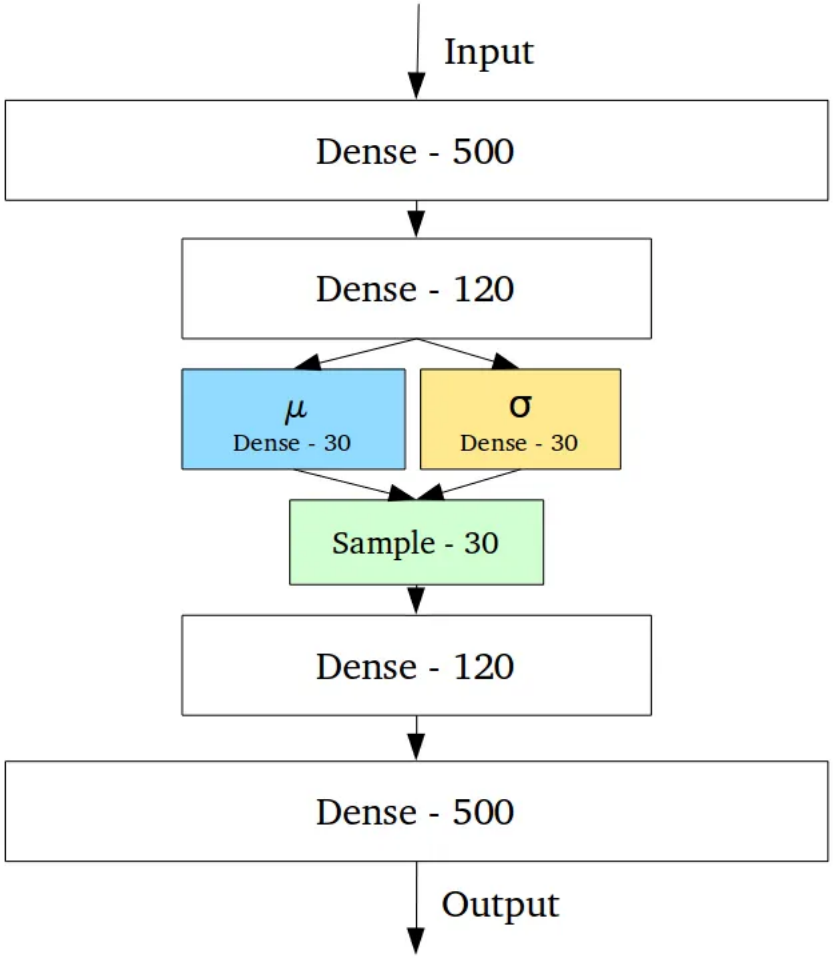
But when you're building a *generative* model, you **don't** want to prepare to *replicate* the same image you put in. You want to randomly sample from the latent space, or generate variations on an input image, from a continuous latent space.

If the space has discontinuities (eg. gaps between clusters) and you sample/generate a variation from there, the decoder will simply generate an unrealistic output, because the decoder has *no idea* how to deal with that region of the latent space. During training, it *never saw* encoded vectors coming from that region of latent space.

Variational Autoencoders

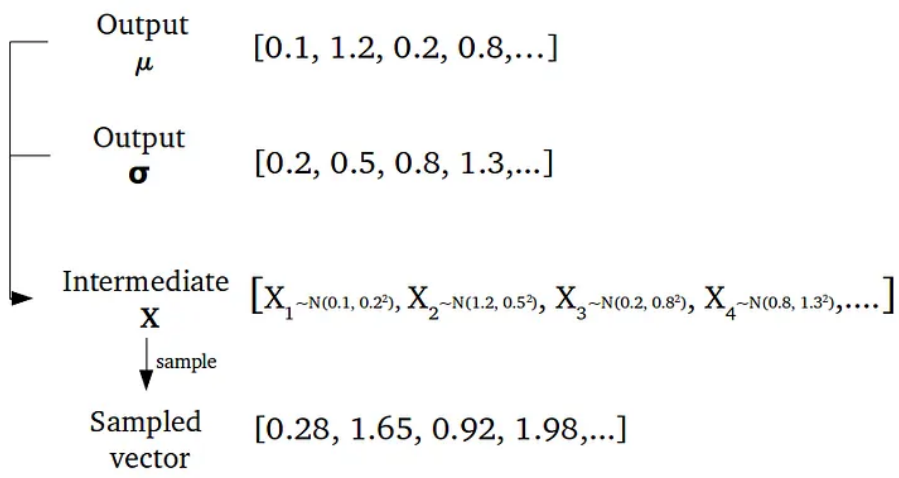
Variational Autoencoders (VAEs) have one fundamentally unique property that separates them from vanilla autoencoders, and it is this property that makes them so useful for generative modeling: their latent spaces are, *by design,* continuous, allowing easy random sampling and interpolation.

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



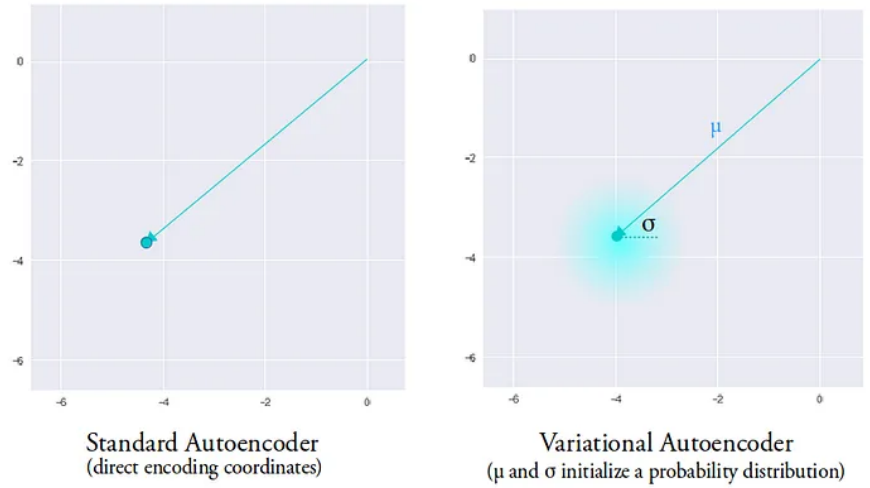
Variational Autoencoder

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



Stochastically generating encoding vectors

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



Standard Autoencoder
(direct encoding coordinates)

Variational Autoencoder
(μ and σ initialize a probability distribution)

Intuitively, the mean vector controls where the encoding of an input should be centered around, while the standard deviation controls the “area”, how much from the mean the encoding can vary. As encodings are generated at random from anywhere inside the “circle” (the distribution), the decoder learns that not only is a single point in latent space referring to a sample of

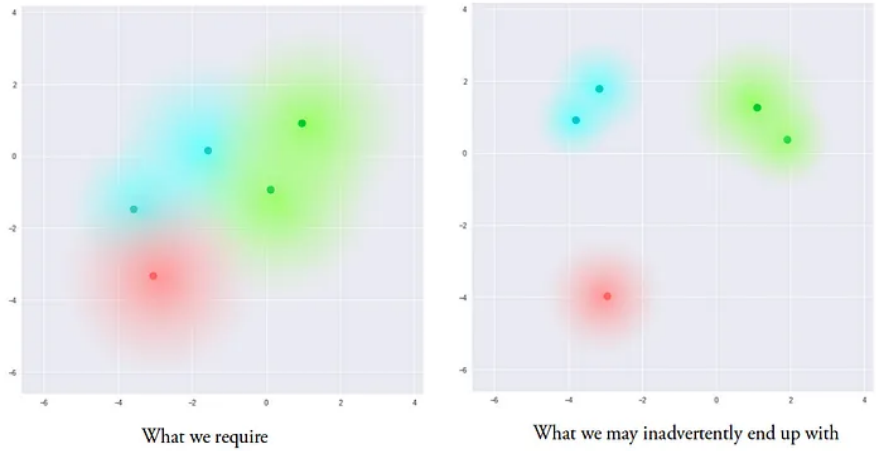
that class, but all nearby points refer to the same as well. This allows the decoder to not just decode single, specific encodings in the latent space (leaving the decodable latent space discontinuous), but ones that slightly vary too, as the decoder is exposed to a range of variations of the encoding of the same input during training. In code:

```python
# build your encoder upto here. It can simply be a series of dense layers, a convolution
# or even an LSTM decoder. Once made, flatten out the final layer of the encoder, call i

# we use Keras to build the graph

latent_size = 5
mean = Dense(latent_size)(hidden)

# we usually don't directly compute the stddev σ
# but the log of the stddev instead, which is log(σ)
# the reasoning is similar to why we use softmax, instead of directly outputting
# numbers in fixed range [0, 1], the network can output a wider range of numbers which w
log_stddev = Dense(latent_size)(hidden)

def sampler(mean, log_stddev):
    # we sample from the standard normal a matrix of batch_size * latent_size (taking in
    std_norm = K.random_normal(shape=(K.shape(mean)[0], latent_size), mean=0, stddev=1)
    # sampling from Z~N(μ, σ^2) is the same as sampling from μ + σX, X~N(0,1)
    return mean + K.exp(log_stddev) * std_norm

latent_vector = Lambda(sampler)([mean, log_stddev])
# pass latent_vector as input to decoder layers
```

variational-sampler.py hosted with ❤ by GitHub view raw

code for sampling mean and stddev

The model is now exposed to a certain degree of local variation by varying the encoding of one sample, resulting in smooth latent spaces on a local scale, that is, for similar samples. Ideally, we want overlap between samples that are not very similar too, in order to interpolate *between* classes. However, since there are *no limits* on what values vectors **μ** and **σ** can take on, the encoder can learn to generate very different **μ** for different classes, clustering them apart, and minimize **σ**, making sure the encodings themselves don't vary much for the same sample (that is, less uncertainty for the decoder). This allows the decoder to efficiently reconstruct the *training* data.



What we require

What we may inadvertently end up with

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

In order to force this, we introduce the Kullback–Leibler divergence (KL divergence[2]) into the loss function. The KL divergence between two probability distributions simply measures how much they *diverge* from each other. Minimizing the KL divergence here means optimizing the probability distribution parameters (**μ** and **σ**) to closely resemble that of the target distribution.

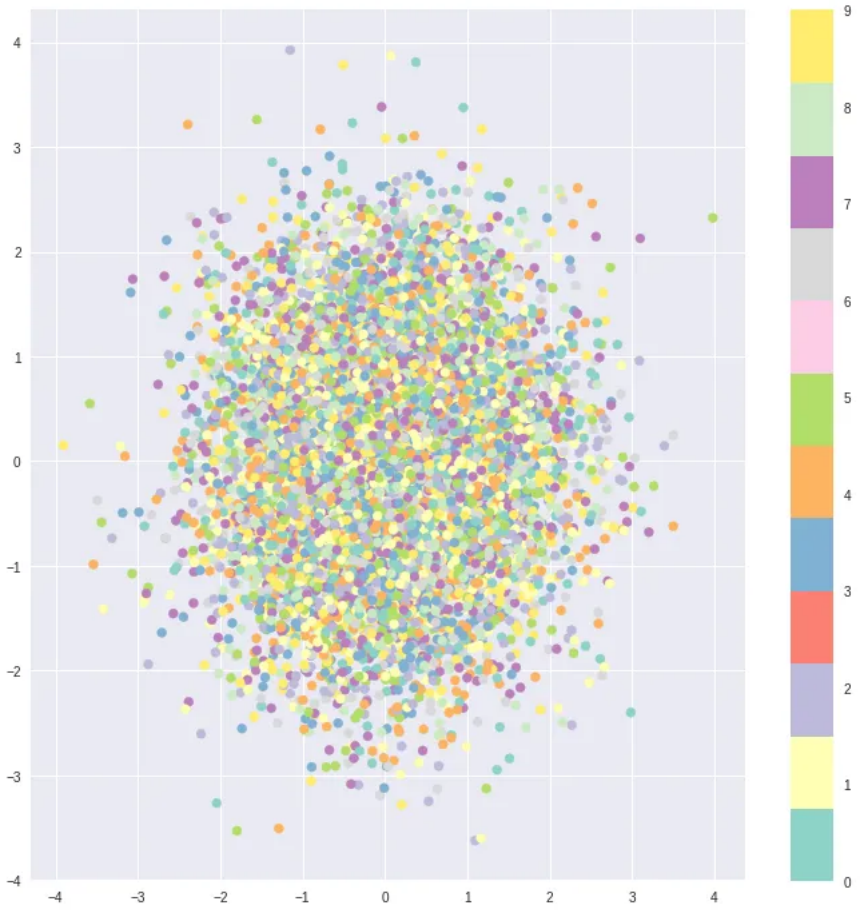
$$\sum_{i=1}^{n} \sigma_i^2 + \mu_i^2 - \log(\sigma_i) - 1$$

For VAEs, the KL loss is equivalent to the *sum* of all the KL divergences between the *component* $Xi \sim N(\mu i, \sigma i^2)$ in **X**, and the standard normal[3]. It's minimized when $\mu i = 0$, $\sigma i = 1$.

Intuitively, this loss encourages the encoder to distribute all encodings (for all types of inputs, eg. all MNIST numbers), evenly around the center of the

latent space. If it tries to “cheat” by clustering them apart into specific regions, away from the origin, it will be penalized.

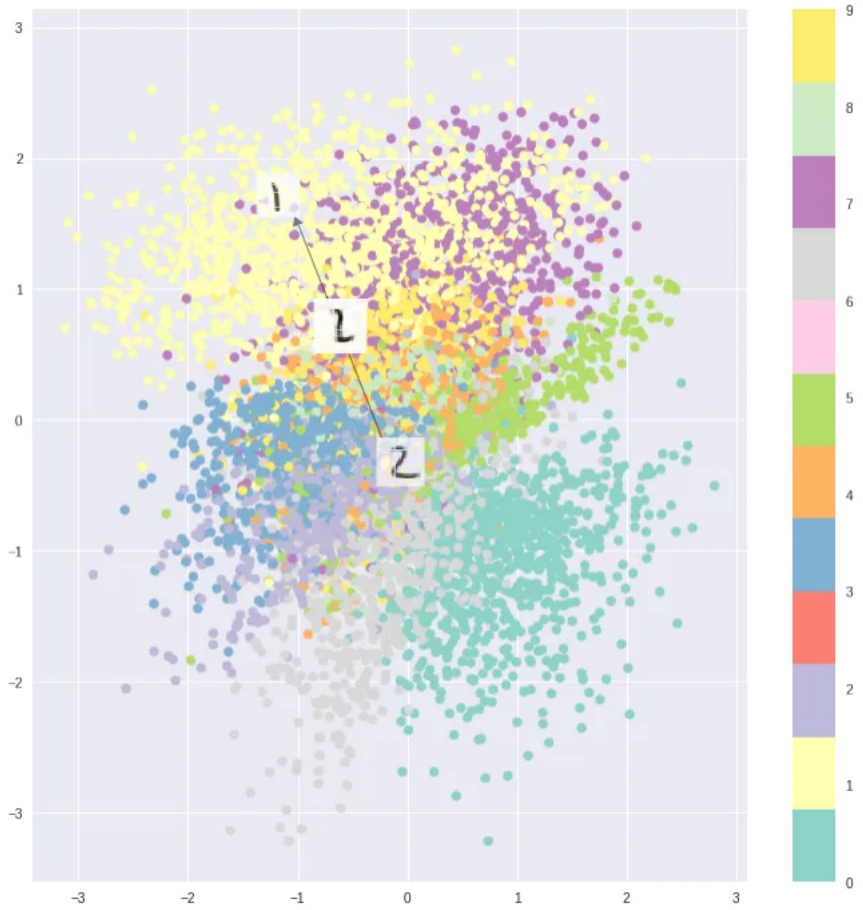
Now, using purely KL loss results in a latent space results in encodings densely placed randomly, near the center of the latent space, with little regard for similarity among nearby encodings. The decoder finds it impossible to decode anything meaningful from this space, simply because there really isn’t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



Optimizing using pure KL divergence loss

Optimizing the two together, however, results in the generation of a latent space which maintains the similarity of nearby encodings on the *local scale*

via clustering, yet *globally,* is very densely packed near the latent space origin (compare the axes with the original).



Optimizing using both reconstruction loss and KL divergence loss

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

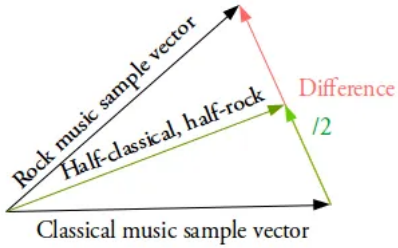
```python
def vae_loss(input_img, output):
    # compute the average MSE error, then scale it up i.e. simply sum on all axes
    reconstruction_loss = K.sum(K.square(output-input_img))
    # compute the KL loss
    kl_loss = -0.5 * K.sum(1 + log_stddev - K.square(mean) - K.square(K.exp(log_stddev)),
    # return the average loss over all images in batch
    total_loss = K.mean(reconstruction_loss + kl_loss)
    return total_loss
```

variationalloss.py hosted with ❤ by GitHub view raw

The final loss function

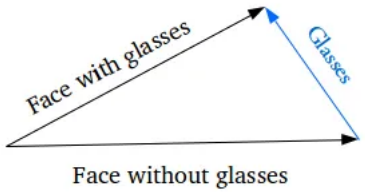
Vector arithmetic

So how do we actually produce these smooth interpolations we speak of? From here on out, it's simple vector arithmetic in the latent space.



Interpolating between samples

For example, if you wish to generate a new sample halfway between two samples, just find the difference between their mean (μ) vectors, and add half the difference to the original, and then simply decode it.



Adding new features to samples

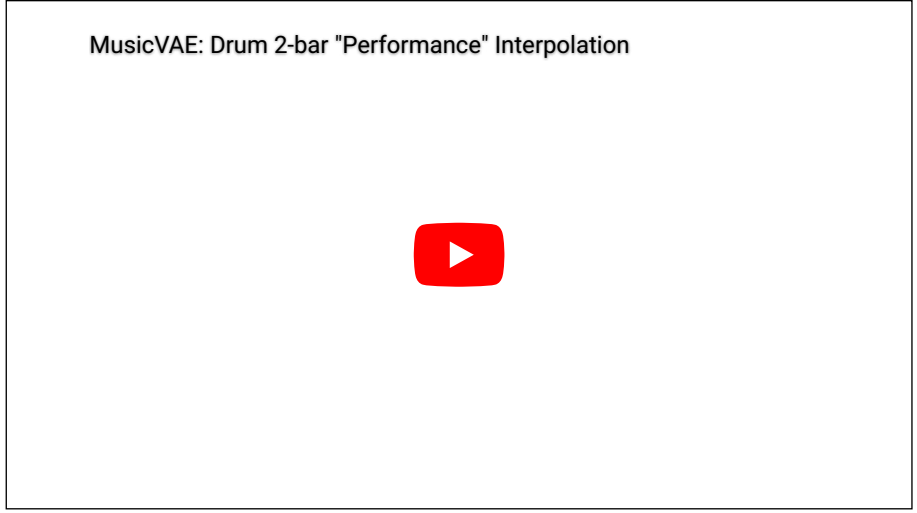
What about generating *specific features*, such as generating glasses on a face? Find two samples, one with glasses, one without, obtain their encoded vectors from the encoder, and save the difference. Add this new "glasses" vector to any other face image, and decode it.

Where to from here?

There are plenty of further improvements that can be made over the variational autoencoder. You could indeed, replace the standard fully-connected dense encoder-decoder with a convolutional-deconvolutional encoder-decoder pair, such as this project[4], to produce great synthetic human face photos.

Generating celebrity-lookalike photos

You could even train an autoencoder using LSTM encoder-decoder pairs (using a modified version of the seq2seq architecture) for *sequential, discrete* data (something not possible with methods such as GANs), to produce synthetic text, or even interpolate between MIDI samples such as Google Brain's Magenta's MusicVAE[5]:



VAEs work with remarkably diverse types of data, sequential or non-sequential, continuous or discrete, even labelled or completely unlabelled, making them highly powerful generative tools. I hope you now understand how VAEs work, and that you will be able to use them on your own generative endeavors as well.

If you found this article useful and believe others would too, leave a clap! If you'd like to stay connected, you'll find me on Twitter here.

Notes

[1]Latent Constraints: Conditional Generation from Unconditional Generative Models

[2]Kullback-Leibler Divergence Explained

[3]KL divergence between two univariate Gaussians

[4]Deep Feature Consistent Variational Autoencoder

[5]Hierarchical Variational Autoencoders for Music

Further reading:

- Variational Autoencoders
- Tutorial — What is a variational autoencoder?
- Tutorial on Variational Autoencoders

Implementations:

- Building Autoencoders in Keras
- Convolutional-deconvolutional autoencoder in Keras

Machine Learning | Deep Learning | Artificial Intelligence | Art | Design

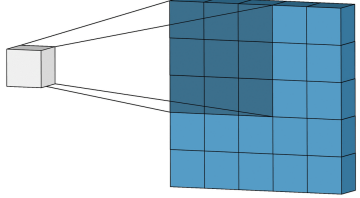
Written by Irhum Shafkat

2K Followers · Writer for Towards Data Science

Curious programmer, tinkers around in Python and deep learning.

Follow

More from Irhum Shafkat and Towards Data Science



Irhum Shafkat in Towards Data Science

Intuitively Understanding Convolutions for Deep Learning

Exploring the strong visual hierarchies that makes them work

15 min read · Jun 2, 2018

12.2K 31

Iulia Brezeanu in Towards Data Science

How to Cut RAG Costs by 80% Using Prompt Compression

Accelerating Inference With Prompt Compression

✦ · 11 min read · Jan 4

1K 6

Siavash Yasini in Towards Data Science

How to Write Memory-Efficient Classes in Python

Three tricks to prevent your data project from memory overflow

✦ · 7 min read · Jan 13

1.1K 10

Irhum Shafkat

New Post: Understanding Graph Neural Networks

I've been a Medium user for quite a while now, but two things have bothered me as of late:

1 min read · Apr 29, 2020

151 1

See all from Irhum Shafkat

See all from Towards Data Science

Recommended from Medium

Will Badr in Towards Data Science

Uncovering Anomalies with Variational Autoencoders (VAE): ...

An example use case of using Variational Autoencoders (VAE) to detect anomalies in ...

· 9 min read · Jan 17, 2023

120 1

Tiya Vaj

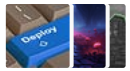
Variational Autoencoders

Variational Autoencoders (VAEs) are valuable in tasks that involve combining data,...

2 min read · Dec 26, 2023

1

Lists



Predictive Modeling w/ Python

20 stories · 818 saves

Natural Language Processing

1112 stories · 588 saves

AI Regulation

6 stories · 282 saves

Practical Guides to Machine Learning

10 stories · 949 saves

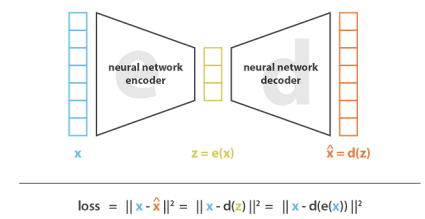
Harshita Sharma in Accredian

Building Intuition: Variational Autoencoders (VAEs)

How Variational Autoencoders are able to generate new data

7 min read · Sep 25, 2023

14



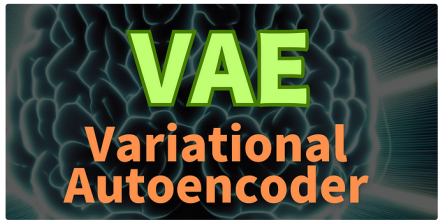
Etorezone

Understanding the Differences Between AutoEncoder (AE) and...

Are you familiar with the nuances between AutoEncoder (AE) and Variational...

2 min read · Oct 2, 2023

11



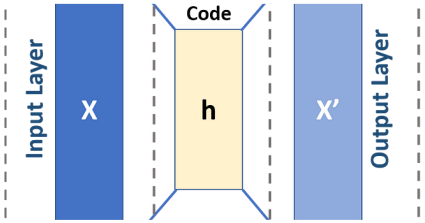
Naoki

VAE: Variational Auto-Encoder (2013)

Understanding the Auto-Encoding Variational Bayes Paper

· 29 min read · Aug 23, 2023

46



Skann.ai

What are Convolutional Variational Auto-Encoders (CVAE)

Introduction

6 min read · Sep 21, 2023

See more recommendations