

Autoencoders

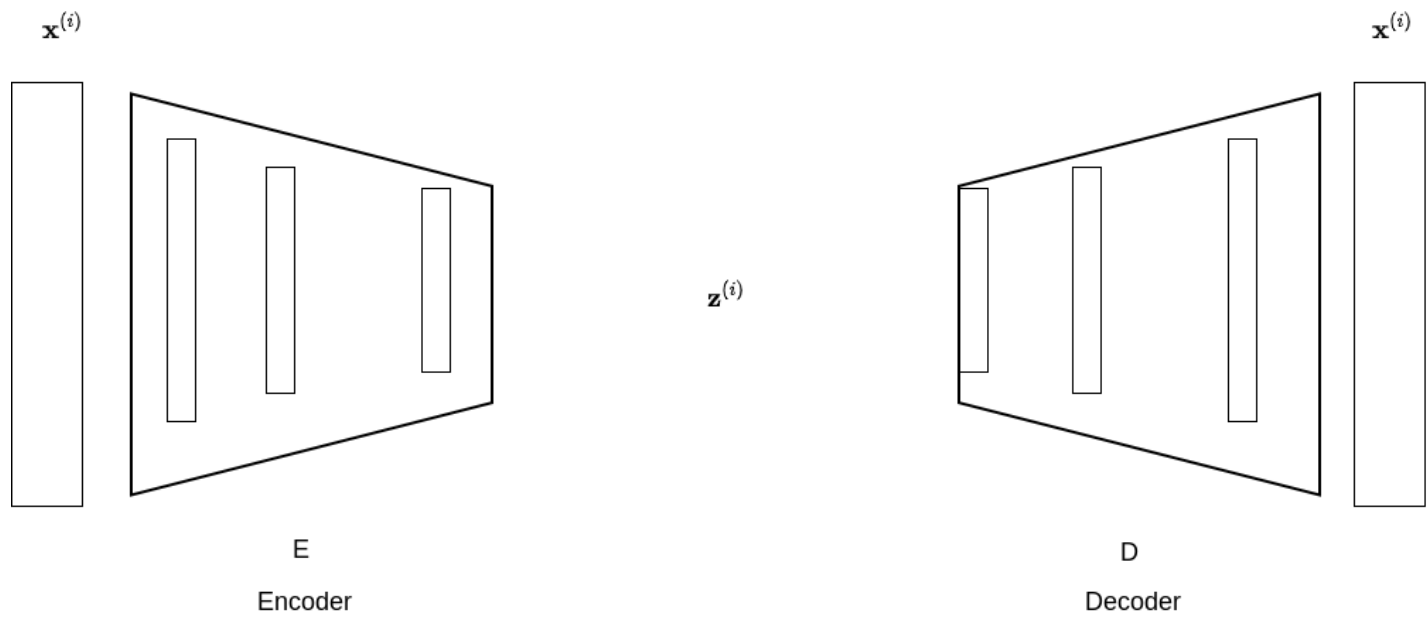
An *Autoencoder* (AE) is a Neural Network comprised of two parts:

- an *Encoder*, which takes the input \mathbf{x} and produces an intermediate ("latent") representation \mathbf{z} as output
- a *Decoder*, which takes \mathbf{z} as input and attempts to reproduce \mathbf{x} as output

Both the Encoder and Decoder are Neural Networks

- their weights are learned by training them in tandem
- on training set $\langle \mathbf{X}, \mathbf{y} \rangle = \langle \mathbf{X}, \mathbf{X} \rangle$

Autoencoder



A non-trivial Autoencoder (i.e., one in which the parts are not merely the Identity transformation)

- has latent representation \mathbf{z} of dimension less than input \mathbf{x}
- \mathbf{z} is a *bottle-neck*
- forcing dimensionality reduction, like PCA
- causing the inversion of the Decoder to be imperfect

Comparison of Autoencoders and PCA

Both the AE and PCA are methods to create representations of an input of length n via reduced dimensionality vectors of length $r \leq n$

They are similar in *purpose* but different in *detail*

- PCA creates n vectors (of length n) called *components*
 - Each $\mathbf{x}^{(i)}$ of length n is represented as a linear combination of $r \leq n$ components
 - The reduced dimensionality representation is a vector of length $r \leq n$: the weights used in the linear combination
 - The components are common to all inputs $\mathbf{x}^{(i)}$
- Autoencoder
 - the reduced dimensionality representation is a vector of length $r \leq n$
 - the representation is unique to $\mathbf{x}^{(i)}$: not shared "components"

Our interest in Autoencoders

- Study Functional architecture
 - [TensorFlow Tutorial on Autoencoders](https://www.tensorflow.org/tutorials/generative/autoencoder)
(<https://www.tensorflow.org/tutorials/generative/autoencoder>)
- Generative
 - Create *synthetic* examples \mathbf{x}'
 - By sampling \mathbf{z}' from the space of latent representations
 - And inverting them

Generator

Uses

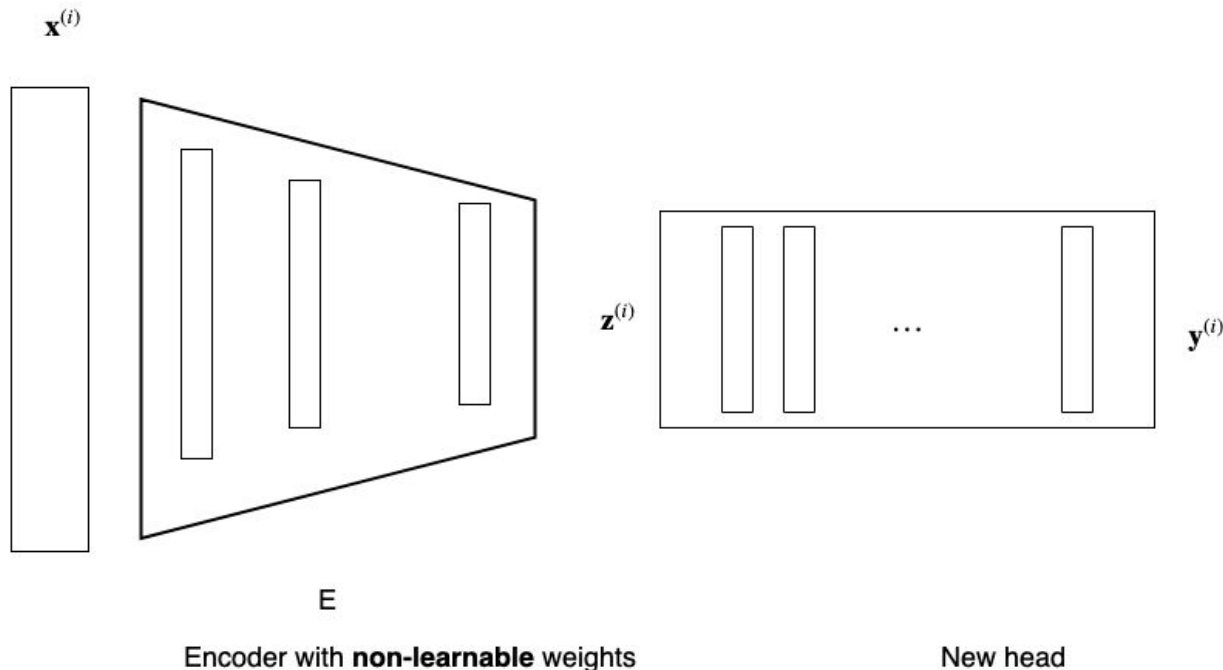
As an aside, we mention other use cases

Dimensionality reduction and Transfer learning

Once the Autoencoder has been trained, we can discard the Decoder

- Use the Encoder to create reduced dimension representations of large and high dimension inputs
 - Image search by replacing 3D megapixel images by shorter, 1D vectors
- Transfer to another task

Autoencoder: Encoder + New head



De-noising Autoencoder

Using an AE for dimensionality reduction is similar to using PCA

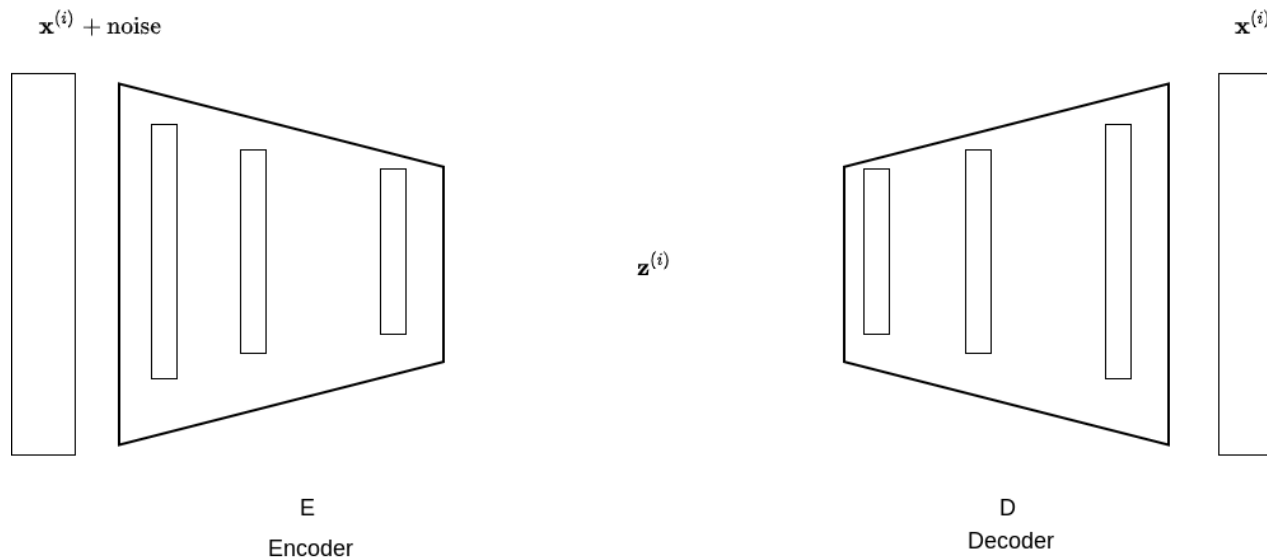
- **But** unlike PCA, there is no **explicit** "relative importance" associated with the retained dimensions

But we can *hope* that the information lost through the bottleneck process is less important.

A *De-noising Autoencoder* is an Autoencoder trained on a slightly corrupted "noisy" input

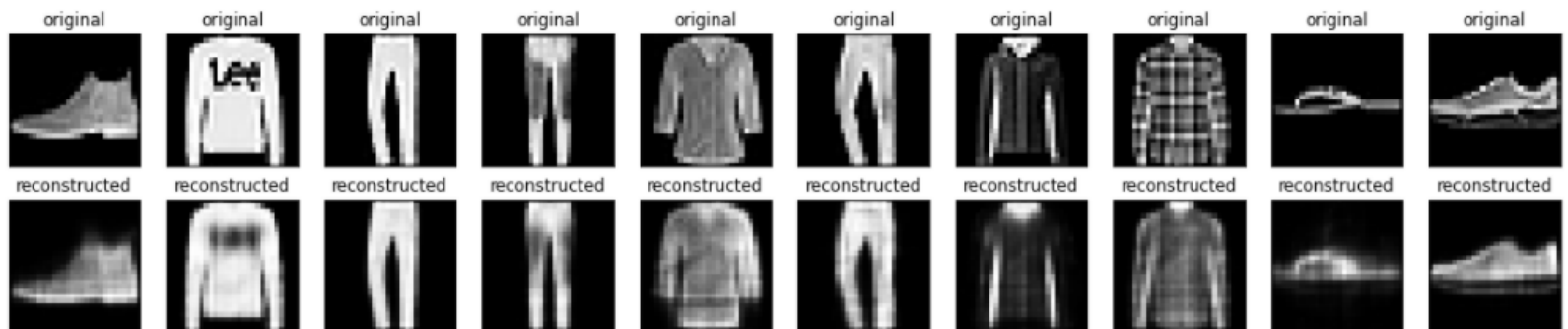
- $\langle \mathbf{X}, \mathbf{y} \rangle = \langle \mathbf{X} + \epsilon, \mathbf{X} \rangle$

Autoencoder: Denoising



De-noising may be useful as a pre-processing step for cleaning noisy data.

De-noising autoencoder: noisy inputs, de-noised outputs



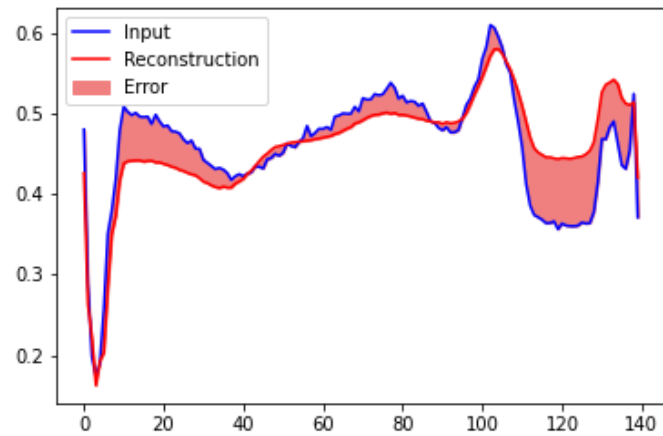
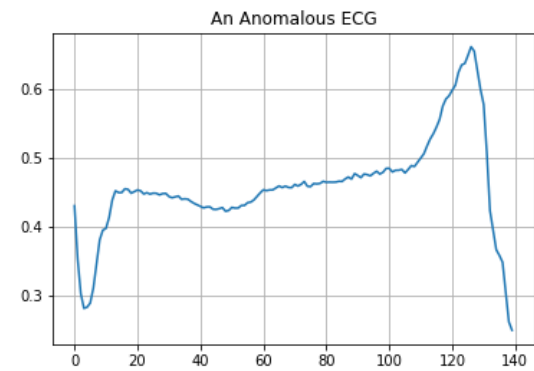
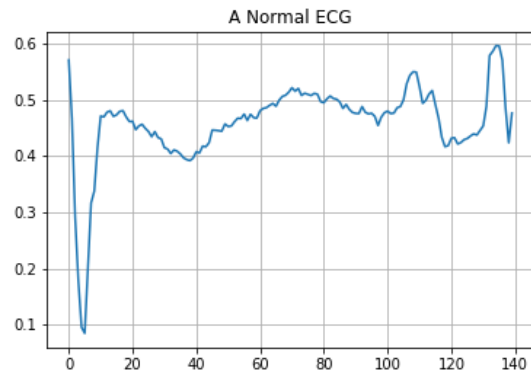
Autoencoder as Anomaly Detector

By forcing the input \mathbf{x} through a bottleneck, the reconstructed input hopefully has "less important" information stripped away.

We may choose to characterize this lost information as an *anomaly* if the magnitude of the reconstruction error is larger than some threshold.

- Error: noise to be removed
- Signal: something unusual to be flagged for attention
- Signal: a source of alpha
 - Reconstructed input is our "model"'s prediction
 - The noise is divergence from our model
 - trading opportunity?

Anomaly Detector



Details

Notation summary

term	dimension	meaning
\mathbf{x}	n	Input
$\tilde{\mathbf{x}}$	n	Output: reconstructed \mathbf{x}
\mathbf{z}	$n' \ll n$	Latent representation
E	$\mathbb{R}^n \rightarrow \mathbb{R}^{n'}$	Encoder
		$E(\mathbf{x}) = \mathbf{z}$
D	$\mathbb{R}^{n'} \rightarrow \mathbb{R}^n$	Decoder
		$\tilde{\mathbf{x}} = D(\mathbf{z})$
		$\tilde{\mathbf{x}} = D(E(\mathbf{x}))$
		$\tilde{\mathbf{x}} \approx \mathbf{x}$

Loss function

The obvious loss functions compare the original $\mathbf{x}^{(i)}$ and reconstructed $\tilde{\mathbf{x}}^{(i)}$ feature by feature:

Mean Squared Error (MSE)

$$\mathcal{L}^{(i)} = \sum_{j=1}^{|\mathbf{x}|} (\mathbf{x}_j^{(i)} - \tilde{\mathbf{x}}_j^{(i)})^2$$

Binary Cross Entropy

For the special case where *each* original feature is in the range $[0, 1]$ (e.g., an image)

$$\mathcal{L}^{(i)} = \sum_{j=1}^{|\mathbf{x}|} \left(\mathbf{x}_j^{(i)} \log(\tilde{\mathbf{x}}_j^{(i)}) + (1 - \mathbf{x}_j^{(i)}) \log(1 - \tilde{\mathbf{x}}_j^{(i)}) \right)$$

Generative Limitations

We propose to create synthetic examples \mathbf{x}' by sampling \mathbf{z} .

Although the synthetic \mathbf{x}' created by this inversion seems appealing, there are some issues

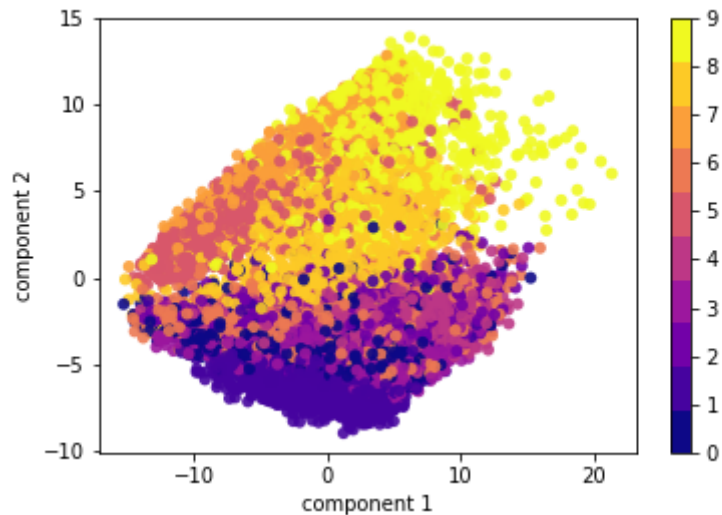
- Assuming we need labeled examples $\langle \mathbf{x}, \mathbf{y} \rangle$
 - we have no control as to the class \mathbf{y}' of the synthetic \mathbf{x}'
- Our method of sampling \mathbf{z} is not dependent on the distribution of \mathbf{z}
 - In general, the distribution is unknown
 - In particular, the sample may not be representative of any known (e.g., training) true example
 - Even if we obtain \mathbf{z} by slight modification of a particular $\mathbf{x}^{(i)}$

$$\mathbf{z} = E(\mathbf{x}^{(i)}) + \epsilon$$

there is no guarantee as to the label or fidelity of $\mathbf{x}' = D(\mathbf{z})$

To illustrate, we

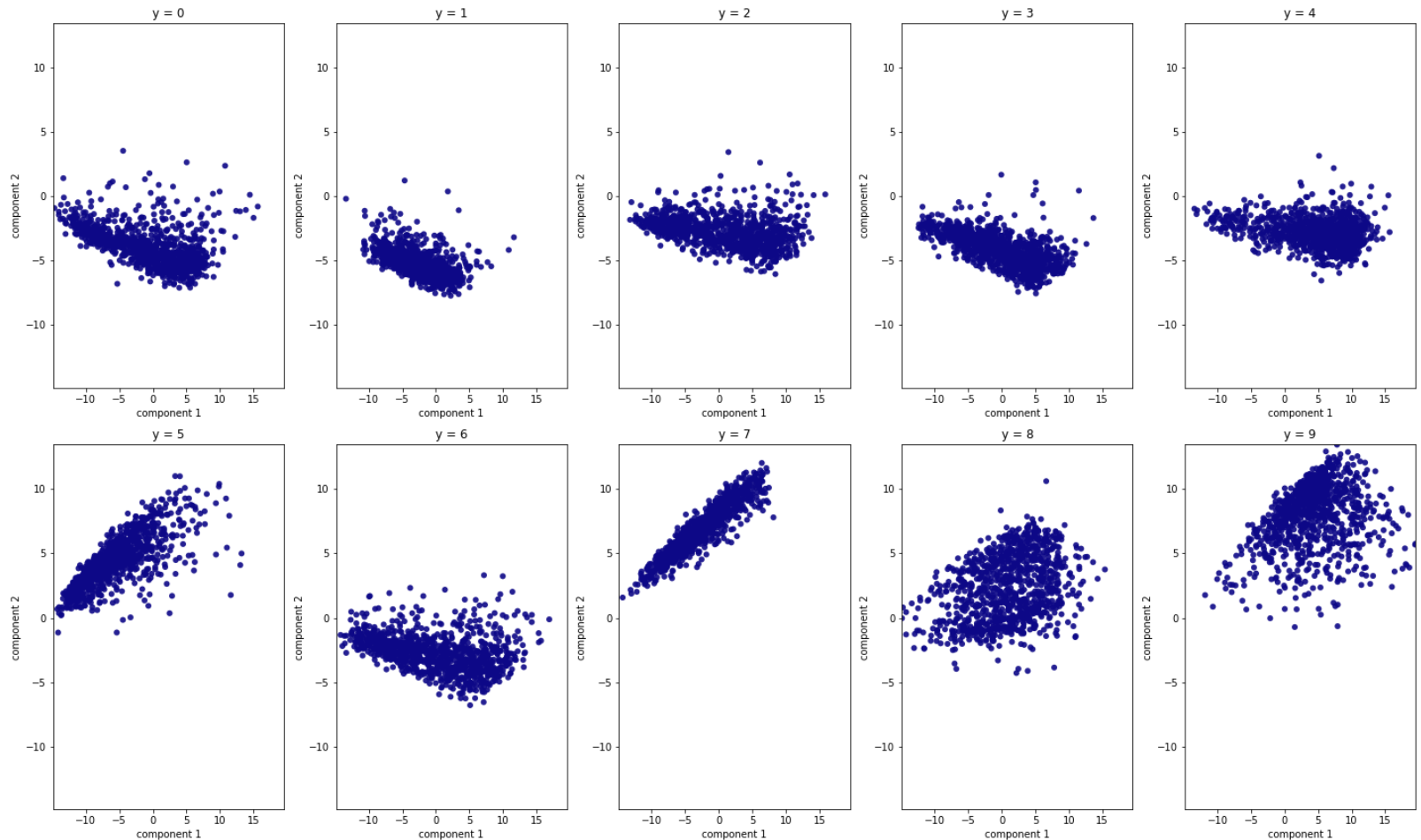
- create an [autoencoder \(autoencoder.ipynb\)](#) for MNIST fashion
 - 10 classes
 - Latent representation are vectors of length 64
- obtain the latent representations for a set of test inputs
- create a scatter plot of the latents
 - using PCA to project the high dimensionality latents to 2D



As you can see

- the latents are not uniformly distributed
- latents of particular classes (each class depicted with a unique color) form clusters

We can illustrate the latter point via a separate plot of the latents for each class



Thus, sampling latents uniformly will not necessarily find a latent "in the neighborhood"

- of any of the classes
- of any particular class

We can emphasize the latter point.

Let's explore the neighborhood around a the latent representation of a single input

- add random normal noise with varying increments of standard deviation

We might expect to obtain images similar to the original.

As you can see from the above, even moving in a small radius from the latent of the original does not guarantee a realistic decoded output.

- So we can't generate a synthetic example of a particular class by a small perturbation of the latent from a genuine image of the class

Next, we conduct an experiment in interpolating between the latents associated with 2 inputs.

- interpolate between the latents and decode
- first plot: 0% second "end" image; 100% first image
- last plot: 100% second image; 0% first image

As you can see from the intermediate outputs

- not all latents correspond to recognizable classes

Thus, we see issues associated with generating synthetic examples by simple-minded sampling of the latent space.

Experiments with Autoencoders

The plots in this notebook were generated by this [notebook](#) ([Autoencoder_practice.ipynb](#)).

- derived from the [TensorFlow tutorial on Autoencoders](#) (<https://www.tensorflow.org/tutorials/generative/autoencoder>).
- illustrates Latent representation, Denoising, Anomaly Detection
- (secondary objective: study the code)

In [3]: `print("Done")`

Done

