**Autoencoders**

**Olga Lyudchik**

**Non-member state summer student**

**Supervisor: Dr. Jean-Roch Valery VLIMANT (EP-UCM)**

# 

# Contents List

[What are autoencoders?](#h.gv0b7uvmwhd0)

[What are autoencoders good for?](#h.643ehtcujrwr)

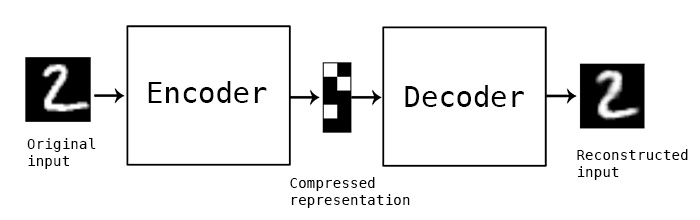
[What is the big deal with autoencoders?](#h.88uzqdim262)

[Variational autoencoder (VAE)](#h.o5dazhg3j157)

[Basic autoencoder](#h.14c9ltrw5c08)

# 

# What are autoencoders?



"Autoencoding" is a data compression algorithm where the compression and decompression functions are 1) data-specific, 2) lossy, and 3) learned automatically from examples rather than engineered by a human. Additionally, in almost all contexts where the term "autoencoder" is used, the compression and decompression functions are implemented with neural networks.

1) Autoencoders are data-specific, which means that they will only be able to compress data similar to what they have been trained on. An autoencoder trained on pictures of faces would do a rather poor job of compressing pictures of trees, because the features it would learn would be face-specific.

2) Autoencoders are lossy, which means that the decompressed outputs will be degraded compared to the original inputs. This differs from lossless arithmetic compression.

3) Autoencoders are learned automatically from data examples, which is a useful property: it means that it is easy to train specialized instances of the algorithm that will perform well on a specific type of input. It doesn't require any new engineering, just appropriate training data.

To build an autoencoder, you need three things:

1. an encoding function,
2. a decoding function,
3. a distance function between the amount of information loss between the compressed representation of your data and the decompressed representation (i.e. a "loss" function).

The encoder and decoder will be chosen to be parametric functions (typically neural networks), and to be differentiable with respect to the distance function, so the parameters of the encoding/decoding functions can be optimize to minimize the reconstruction loss, using Stochastic Gradient Descent.

# What are autoencoders good for?

Today two interesting practical applications of autoencoders are

1. data denoising,
2. dimensionality reduction for data visualization.

With appropriate dimensionality and sparsity constraints, autoencoders can learn data projections that are more interesting than PCA or other basic techniques.

For 2D visualization specifically, [t-SNE](https://en.wikipedia.org/wiki/T-distributed_stochastic_neighbor_embedding) is probably the best algorithm around, but it typically requires relatively low-dimensional data. So a good strategy for visualizing similarity relationships in high-dimensional data is to start by using an autoencoder to compress your data into a low-dimensional space (e.g. 32 dimensional), then use t-SNE for mapping the compressed data to a 2D plane.

# What is the big deal with autoencoders?

Otherwise, one reason why they have attracted so much research and attention is because they have long been thought to be a potential avenue for solving the problem of unsupervised learning, i.e. the learning of useful representations without the need for labels. Then again, autoencoders are not a true unsupervised learning technique (which would imply a different learning process altogether), they are a self-supervised technique, a specific instance of supervised learning where the targets are generated from the input data. In order to get self-supervised models to learn interesting features, you have to come up with an interesting synthetic target and loss function, and that's where problems arise: merely learning to reconstruct your input in minute detail might not be the right choice here. At this point there is significant evidence that focusing on the reconstruction of a picture at the pixel level, for instance, is not conductive to learning interesting, abstract features of the kind that label-supervized learning induces (where targets are fairly abstract concepts "invented" by humans such as "dog", "car"...). In fact, one may argue that the best features in this regard are those that are the worst at exact input reconstruction while achieving high performance on the main task that you are interested in (classification, localization, etc).

# Variational autoencoder (VAE)

Variational autoencoders are a slightly more modern and interesting take on autoencoding.

What is a variational autoencoder?

It's a type of autoencoder with added constraints on the encoded representations being learned. More precisely, it is an autoencoder that learns a [latent variable model](https://en.wikipedia.org/wiki/Latent_variable_model) for its input data. So instead of letting your neural network learn an arbitrary function, you are learning the parameters of a probability distribution modeling your data. If you sample points from this distribution, you can generate new input data samples: a VAE is a "generative model".

How does a variational autoencoder work?

First, an encoder network turns the input samples x into two parameters in a latent space, which we will note z\_mean and z\_log\_sigma. Then, we randomly sample similar points z from the latent normal distribution that is assumed to generate the data, via z = z\_mean + exp(z\_log\_sigma) \* epsilon, where epsilon is a random normal tensor. Finally, a decoder network maps these latent space points back to the original input data.

The parameters of the model are trained via two loss functions: a reconstruction loss forcing the decoded samples to match the initial inputs (just like in our previous autoencoders), and the KL divergence between the learned latent distribution and the prior distribution, acting as a regularization term. You could actually get rid of this latter term entirely, although it does help in learning well-formed latent spaces and reducing overfitting to the training data.

Because a VAE is a more complex example, we have made the code available on Github as [a standalone script](https://github.com/fchollet/keras/blob/master/examples/variational_autoencoder.py). Here we will review step by step how the model is created.

# Basic autoencoder

The traditional autoencoder is an artificial neural network that attempts to reproduce its

input, i.e., the target output is the input. More formally an autoencoder takes an input vector *x ∈ [0, 1]d* and maps it to a hidden representation *y ∈ [0, 1]d* through a deterministic mapping *y = hθ(x) = s(Wx + b)*, parameterized by *θ = {W, b}.*

*W* is a *d’ × d* weight matrix, *b* is a bias vector and *s* is the sigmoid3 activation function, *s(x) = 1/(1+e−x)*. The hidden representation y, sometimes called the *latent* representation, is then mapped back to a reconstructed vector *z ∈ [0, 1]d,* where *z = gθ’(y) = s(W’y + b’), with θ’ = {W’, b’}.*

The basic idea here is that the autoencoder is constructed in such a way that the mapping *x(i) -› y(i)* reveals essential structure in the input vector *x(i)* that is not otherwise obvious.

The parameters *θ* and *θ’* of the model are optimized to minimize the average reconstruction error as shown in the following equation:

Here *L* is a loss function such as the traditional squared error *L(x, z) = ||x – z|2|*.

**Advantages**

Defines a simple, tractable optimization objective that can be used to monitor progress.

**Disadvantages**

The fact that autoencoders are data-specific makes them generally impractical for real-world data compression problems: you can only use them on data that is similar to what they were trained on, and making them more general thus requires lots of training data