Autoencoders

Olga Lyudchik

Non-member state summer student

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

Contents List

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.The schema of autoencoder is shown on the Figure 1.

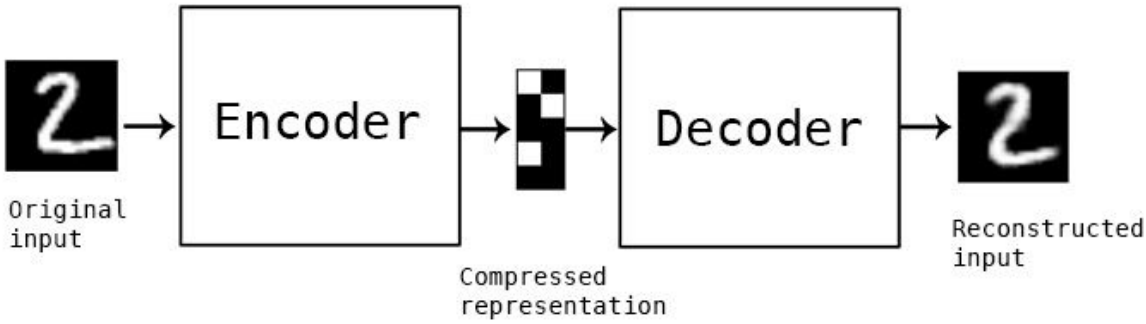


Figure 1. Autoencoder: schema

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.

One reason why autoencoders 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.

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 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. 64 dimensional), then use t-SNE for mapping the compressed data to a 2D plane.

Anomaly Detection using autoencoders

Automatically removing outliers from unlabeled data operates in an unsupervised mode. For this problem, methods in literature explicitly or implicitly make an assumption that inliers are located in dense areas while outliers are not. The dense areas can be estimated by statistical methods, neighbor-based methods, and reconstruction-based methods. For example, the methods in compute PCA projections on data, and those having large projection variances are determined as outliers.

MNIST Dataset

The problem of handwriting recognition is to interpret intelligible handwritten input auto-matically, which is of great interest in the pattern recognition research community because of its applicability to many fields towards more convenient input devices and more efficient data organization and processing. As one of the fundament problems in designing practical recog-nition systems, the recognition of handwritten digits is an active research field. Immediate applications of the digit recognition techniques include postal mail sorting, automatically address reading and mail routing, bank check processing, etc. As a benchmark for testing classification algorithms, the MNIST dataset has been widely used to design novel handwritten digit recognition systems. There are a great amountof studies based on MNIST dataset reported in the literature, suggesting many different methods. One of the major challenges in the recognition of handwritten digits is the within class variance, because people do not always write the same digit in exactly the same way. Many feature extraction approaches have been proposed trying to characterize the shape invariance within a class to improve the discrimination ability. Experiments have shown that by extracting direction features, local structure features or curvature features, the accuracy and efficiency of many classifiers could be improved significantly. In this report we train and test a set of classifiers on the MNIST database for pattern analysis in solving the handwritten digit recognition problem.

Data Description

MNIST is a simple computer vision dataset. It consists contains 60,000 digits ranging from 0 to 9 for training the digit recognition system, and another 10,000 digits as test data. Each digit is normalized and centeredin a gray-level image with size of 28x28. Some examples are shown in Figure 2.

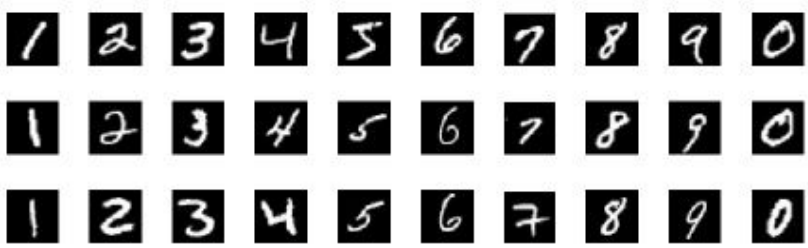


Figure 2. Examples of MNIST data set

Every MNIST data point, every image, can be thought of as an array of numbers describing how dark each pixel is. Since each image has 28 by 28 pixels, we get a 28x28 array. We can flatten each array into a 28∗28=784 dimensional vector. Each component of the vector is a value between zero and one describing the intensity of the pixel. Thus, we generally think of MNIST as being a collection of 784-dimensional vectors.

Not all vectors in this 784-dimensional space are MNIST digits. Typical points in this space are very different! To get a sense of what a typical point looks like, we can randomly pick a few points and examine them. In a random point – a random 28x28 image – each pixel is randomly black, white or some shade of gray. The result is that random points look like noise.

Images like MNIST digits are very rare. While the MNIST data points are embedded in 784-dimensional space, they live in a very small subspace. With some slightly harder arguments, we can see that they occupy a lower dimensional subspace.

People have lots of theories about what sort of lower dimensional structure MNIST, and similar data, have. One popular theory among machine learning researchers is the manifold hypothesis: MNIST is a low dimensional manifold, sweeping and curving through its high-dimensional embedding space. Another hypothesis, more associated with topological data analysis, is that data like MNIST consists of blobs with tentacle-like protrusions sticking out into the surrounding space.

t-Distributed Stochastic Neighbor Embedding

The *t-distributed stochastic neighbor embedding (t-SNE)* is a nonlinear dimensionality reduction technique that is particularly well-suited for embedding high-dimensional data into a space of two or three dimensions, which can then be visualized in a scatter plot.

The t-SNE algorithm comprises two main stages. First, t-SNE constructs a probability distribution over pairs of high-dimensional objects in such a way that similar objects have a high probability of being picked, whilst dissimilar points have an extremely small probability of being picked. Second, t-SNE defines a similar probability distribution over the points in the

low-dimensional map, and it minimizes the Kullback–Leibler divergence between the two distributions with respect to the locations of the points in the map. Note that whilst the original algorithm uses the Euclidean distance between objects as the base of its similarity metric, this should be changed as appropriate.

Lets see in details, Given a set of N high-dimentional objects $x_1, \ldots, x_N$, t-SNE first computes probabilities p_{ij} that are proportional to the similarity of objects x_i and x_j, as follows:

$$p_{j|i} = \frac{\exp(-\|\mathbf{x}_i - \mathbf{x}_j\|^2 / 2\sigma_i^2)}{\sum_{k \neq i} \exp(-\|\mathbf{x}_i - \mathbf{x}_k\|^2 / 2\sigma_i^2)},$$

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}$$

The bandwidth of the Gaussian kernels σ_i is set in such a way that the perplexity of the conditional distribution equals a predefined perplexity using a binary search. As a result, the bandwidth is adapted to the density of the data: smaller values of σ_i are used in denser parts of the data space.

t-SNE aims to learn a d-dimentional map $y_1, \ldots, y_N$ *(with* $y_i \in R^d$*)* that reflects the similarities p_{ij} as well as possible. To the end, it measures similarities q_{ij} between two points in the map y_i and y_j using a very similar approach. Specifically, q_{ij} is defined as*:*

$$q_{ij} = \frac{(1 + \|\mathbf{y}_i - \mathbf{y}_j\|^2)^{-1}}{\sum_{k \neq l} (1 + \|\mathbf{y}_k - \mathbf{y}_l\|^2)^{-1}}$$

Herein a heavy-tailed Student-t distribution (with one-degree of freedom, which is the same as a Cauchy distribution) is used to measure similarities between low-dimensional points in order to allow dissimilar objects to be modeled far apart in the map. The locations of the points y_i in the map are determined by minimizing the (non-symmetric) Kullback–Leibler divergence of the distribution Q from the distribution P that is:

$$KL(P||Q) = \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

The minimization of the Kullback–Leibler divergence with respect to the points y_i is performed using gradient descent. The result of this optimization is a map that reflects the similarities between the high-dimensional inputs well.

t-SNE is extremely popular in the deep learning community. Unfortunately, t-SNE's cost function involves some non-trivial mathematical machinery and requires some significant effort to understand. But, roughly, what t-SNE tries to optimize for is preserving the topology of the data. For every point, it constructs a notion of which other points are it's 'neighbors,' trying to make all points have the same number of neighbors. Then it tries to embed them so that those points all have the same number of neighbors.

In some ways, t-SNE is a lot like the graph based visualization. But instead of just having points be neighbors (if there's an edge) or not neighbors (if there isn't an edge), t-SNE has a continuous spectrum of having points be neighbors to different extents. t-SNE is often very successful at revealing clusters and subclusters in data.

t-SNE does an impressive job finding clusters and subclusters in the data, but is prone to getting stuck in local minima. For example, in the following image shown at Figure 3 we can see two

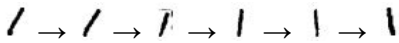
clusters of zeros (red) that fail to come together because a cluster of sixes (blue) get stuck between them.

Figure 3. Autoencoder stuck at local minima

A number of tricks can help us avoid these bad local minima. Firstly, using more data helps a lot. Using the full 50,000 MNIST points works a lot better.

Well done t-SNE plots reveal many interesting features of MNIST. It's not just the classes that t-SNE finds. Let's look more closely at the ones.

The ones cluster is stretched horizontally. As we look at digits from left to right, we see a consistent pattern.



They move from forward leaning ones, like  , into straighter like , and finally to slightly backwards leaning ones, like  . It seems that in MNIST, the primary factor of variation in the ones is tilting. This is likely because MNIST normalizes digits in a number of ways, centering and scaling them. After that, the easiest way to be "far apart" is to rotate and not overlap very much. Similar structure can be observed in other classes.