Introduction

Autoencoders are a specific type of feedforward neural networks where the input is the same as the output. They compress the input into a lower-dimensional code and then reconstruct the output from this representation.

An autoencoder consists of three components: encoder, code and decoder.

Encoder: This is the part of the network that compresses the input into a latent-space representation. It can be represented by an encoding function $h = f(x)$.

Code: The code is a compact "summary" or "compression" of the input, also called the latent-space representation.

Decoder: This part aims to reconstruct the input from the latent space representation. It can be represented by a decoding function $r = g(h)$.

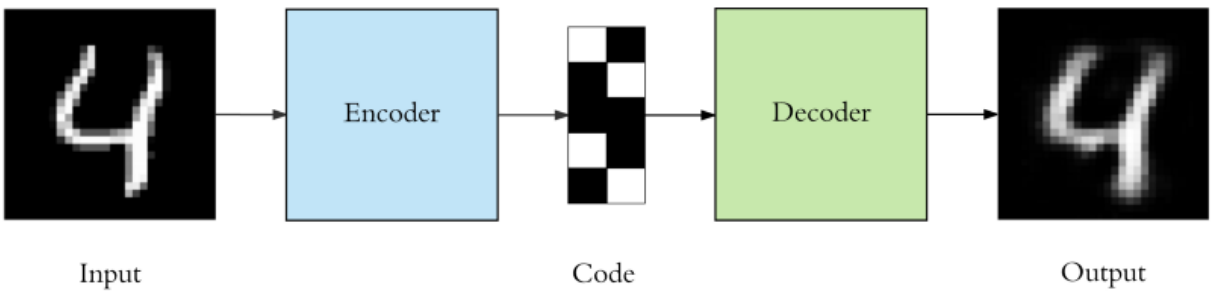

Figure 1: Process of Autoencoders

The autoencoder as a whole can thus be described by the function $g(f(x)) = r$ where you want r as close as the original input x.

Autoencoders are mainly a dimensionality reduction (or compression) algorithm with following important properties:

Data-specific: Autoencoders are only able to meaningfully compress data similar to what they have been trained on. Since they learn features specific for the given training data, they are different than a standard data compression algorithm like gzip. So we can't expect an autoencoder trained on handwritten digits to compress landscape photos.

Lossy: The output of the autoencoder will not be exactly the same as the input, it will be a close but degraded representation. If you want lossless compression they are not the way to go.

Unsupervised: To train an autoencoder we don't need to do anything fancy, just throw the raw input data at it. Autoencoders are considered an unsupervised learning technique since they don't need explicit labels to train on. But to be more precise they are self-supervised because they generate their own labels from the training data.

Architecture

Both the encoder and decoder are fully-connected feedforward neural networks, essentially the ANNs (Artificial Neural Networks). Code is a single layer of an ANN with the dimensionality of our choice. The number of nodes in the code layer (code size) is a hyperparameter that we set before training the autoencoder.

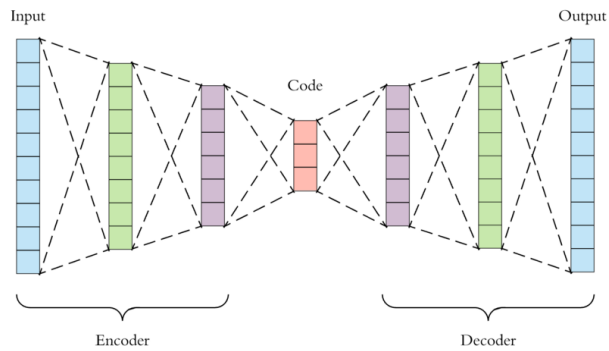

Figure 2: Visualization of Autoencoders

The above image shows the more detailed visualization of an autoencoder. First the input passes through the encoder, which is fully connected ANN, to produce the code. The decoder, which has the similar ANN structure, then produces the output only using the code. The goal is to get an output identical with the input. Note that the decoder architecture is the mirror image of the encoder. The only requirement is the dimensionality of the input and output needs to be the same.

There are four hyperparameters that we need to set before training an autoencoder:

Code size: number of nodes in the middle layer. Smaller size results in more compression.

Number of layers: the autoencoder can be as deep as we like. In the figure above we have two layers in both the encoder and decoder, without considering the input and output.

Number of nodes per layer: the autoencoder architecture we're working on is called a stacked autoencoder since the layers are stacked one after another. Usually stacked autoencoders look like a "sandwitch". The number of nodes per layer decreases with each subsequent layer of the encoder, and increases back in the decoder. Also the decoder is symmetric to the encoder in terms of layer structure.

Loss function: We either use mean squared error (MSE) or binary crossentropy. If the input values are in the range [0, 1] then we typically use crossentropy, otherwise we use the mean squared error.

What are autoencoders used for?

Today, data denoising and dimensionality reduction for data visualization are considered as two main interesting practical applications of autoencoders. With appropriate dimensionality and sparsity constraints, autoencoders can learn data projections that are more interesting than PCA or other basic techniques.

Autoencoders are learned automatically from data examples. It means that it is easy to train specialized instances of the algorithm that will perform well on a specific type of input and that it does not require any new engineering, only the appropriate training data.

Autoencoders are trained to preserve as much information as possible when an input is run through the encoder and then the decoder, but are also trained to make the new representation have various nice properties. Different kinds of autoencoders aim to achieve different kinds of properties.

Various techniques exist to prevent autoencoders from learning the identity function and to improve their ability to capture important information and learn richer representations:

Denoising Autoencoders

Autoencoders are Neural Networks which are commonly used for feature selection and extraction. However, when there are more nodes in the hidden layer than there are inputs, the Network is risking to learn the "Identity Function", also called "Null Function", meaning that the output equals the input, marking the Autoencoder useless.

Denoising Autoencoders solve this problem by corrupting the data on purpose by randomly turning some of the input values to zero. In general, the percentage of input nodes which are being set to zero is about 50%. While other sources suggest a lower count, such as 30%. It depends on the amount of data and input nodes you have.

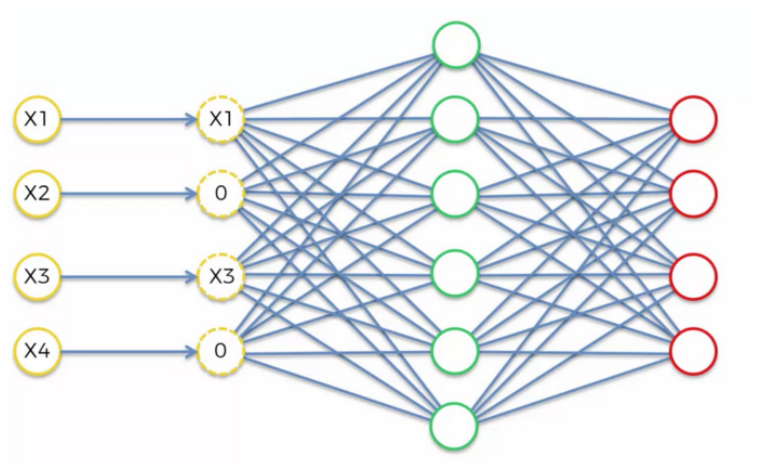

Figure 3: Architecture of Denoising Autoencoders

When calculating the Loss function, it is important to compare the output values with the original input, not with the corrupted input. That way, the risk of learning the identity function instead of extracting features is eliminated.

Apart from Denoising Autoencoders there are different other types of Autoencoders they are Variational AutoEncoder (VAE), Sparse AutoEncoder, Sparse AutoEncoder, Stacked AutoEnoder and Deep Autoencoders. Now we will discuss Deep Autoencoders in detail.

Deep Autoencoders

A deep autoencoder is composed of two, symmetrical deep-belief networks that typically have four or five shallow layers representing the encoding half of the net, and second set of four or five layers that make up the decoding half.

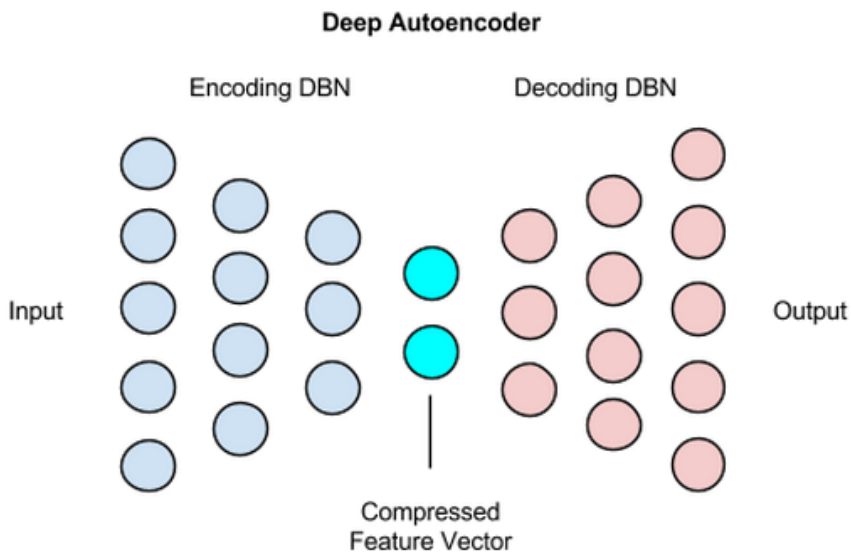

Figure 4: Deep Autoencoder's Structure

Deep autoencoders use MNIST dataset and can also be used for other types of datasets with real-valued data, on which you would use Gaussian rectified transformations for the RBMs (Restricted Boltzmann Machines) instead.

Encoding Input Data

784 (input) —->1000 —->500 —->250 —->100 —>30

Let's, say, the input fed to the network is 784 pixels (the square of the 28x28 pixel images in the MNIST dataset), then the first layer of the deep autoencoder should have 1000 parameters; i.e. slightly larger, because having more parameters than input is a good way to overfit a neural network.

In above example, the layers will be 1000, 500, 250, 100 nodes wide, respectively, until the end, where the net produces a vector 30 numbers long. This 30-number vector is the last layer of the first half of the deep autoencoder, the pretraining half, and it is the product of a normal RBM, rather than an classification output layer such as Softmax or logistic regression, as you would normally see at the end of a deep-belief network.

Decoding Representation

Those 30 numbers are an encoded version of the 28x28 pixel image. The second half of a deep autoencoder actually learns how to decode the condensed vector, which becomes the input as it makes its way back.

The decoding half of a deep autoencoder is a feed-forward net with layers 100, 250, 500 and 1000 nodes wide, respectively. Layer weights are initialized randomly.

784 (output) <—- 1000 <—- 500 <—- 250 <—- 30

The decoding half of a deep autoencoder is the part that learns to reconstruct the image. It does so with a second feed-forward net which also conducts back propagation. The back propagation happens through reconstruction entropy.

Many different variants of the general autoencoder architecture exist with the goal of ensuring that the compressed representation represents meaningful attributes of the original data input; typically the biggest challenge when working with autoencoders is getting your model to actually learn a meaningful and generalizable latent space representation.

Because autoencoders learn how to compress the data based on attributes (ie. correlations between the input feature vector) discovered from data during training, these models are typically only capable of reconstructing data similar to the class of observations of which the model observed during training.

Applications of autoencoders include:

- Anomaly detection
- Data denoising (ex. images, audio)
- Image inpainting
- Information retrieval

References:

For more details on Autoencoders can refer below:

`https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf`
`https://becominghuman.ai/understanding-autoencoders-unsupervised-learning-technique-82fb3fbaec2`