**AUTOENCODER**

[**udbhav.gupta@india.nec.com**](mailto:udbhav.gupta@india.nec.com)

**Member Technical Staff, NEC Corporation**

1. **INTRODUCTION**

An autoencoder is a neural network that is trained to attempt to copy its input to its output. Internally, it has a hidden layer h that describes a code used to represent the input. The network may be viewed as consisting of two parts: an encoder function h=f(x) and a decoder that produces a reconstruction r=g(h). This architecture is presented in ﬁgure 1.1. If an autoencoder succeeds in simply learning to set g(f(x)) =x everywhere, then it is not especially useful. Instead, autoencoders are designed to be unable to learn to copy perfectly. Usually they are restricted in ways that allow them to copy only approximately, and to copy only input that resembles the training data. Because the model is forced to prioritize which aspects of the input should be copied, it often learns useful properties of the data.

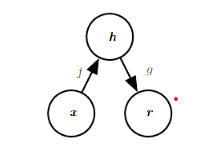


Fig 1.1: The general structure of an autoencoder, mapping an input x to an output (called reconstruction) r through an internal representation or code h. The autoencoder has two components: the encoder f (mapping x to h) and the decoder g (mapping h to r)

Autoencoders are mainly a dimensionality reduction (or compression) algorithm with a couple of 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**.**

1. **ARCHITECTURE**

Let’s explore the details of the encoder, code and decoder. Both the encoder and decoder are fully-connected feedforward neural networks, essentially the ANNs. 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.

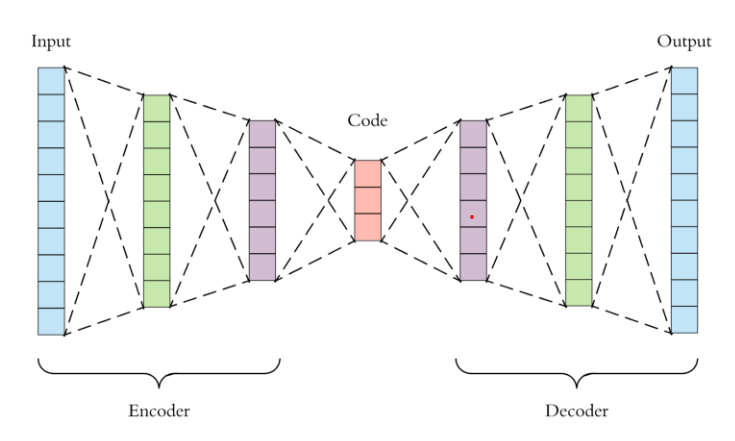
****

Fig 2.1: Flow chart of Autoencoder

This is a more detailed visualization of an autoencoder. First the input passes through the encoder, which is a 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. This is not a requirement but it’s typically the case. The only requirement is the dimensionality of the input and output needs to be the same. Anything in the middle can be played with.

There are 4 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.

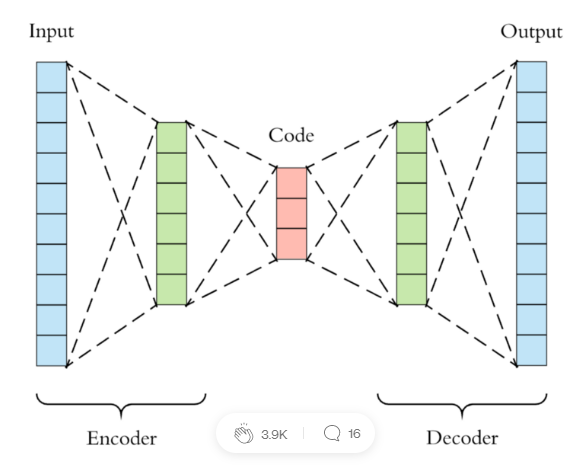
**Number of layers:** the autoencoder can be as deep as we like. In the figure above we have 2 layers in both the encoder and decoder, without considering the input and output.

**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. As noted above this is not necessary and we have total control over these parameters.

**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.

1. **Implementation**

Now let’s implement an autoencoder for the following architecture, 1 hidden layer in the encoder and decoder.



We will use the extremely popular MNIST dataset as input. It contains black-and-white images of handwritten digits.

The images are of size 28x28 and we use them as a vector of 784 numbers between [0, 1].

We will now implement the autoencoder with Keras. The hyperparameters are: 128 nodes in the hidden layer, code size is 32, and binary crossentropy is the loss function.



1. **Types of Autoencoders**
   1. **Undercomplete Autoencoders**

One way to obtain useful features from the autoencoder is to constrain h to have a smaller dimension than x. An autoencoder whose code dimension is less than the input dimension is called **undercomplete**. Learning an undercomplete representation forces the autoencoder to capture the most salient features of the training data.

The learning process is described simply as minimizing a loss function

L (x, g(f(x))),

Where L is a loss function penalizing g(f(x)) for being dissimilar from x, such as the mean squared error.

* 1. **Regularized Autoencoders**

Undercomplete autoencoders, with code dimension less than the input dimension, can learn the most salient features of the data distribution. We have seen that these autoencoders fail to learn anything useful if the encoder and decoder are given too much capacity. A similar problem occurs if the hidden code is allowed to have dimension equal to the input, and in the overcomplete case in which the hidden code has dimension greater than the input. In these cases, even a linear encoder and a linear decoder can learn to copy the input to the output without learning anything useful about the data distribution. Ideally, one could train any architecture of autoencoder successfully, choosing the code dimension and the capacity of the encoder and decoder based on the complexity of distribution to be modeled. Regularized autoencoders provide the ability to do so. Rather than limiting the model capacity by keeping the encoder and decoder shallow and the code size small, regularized autoencoders use a loss function that encourages the model to have other properties besides the ability to copy its input to its output. These other properties include sparsity of the representation, smallness of the derivative of the representation, and robustness to noise or to missing inputs. A regularized autoencoder can be nonlinear and overcomplete but still learn something useful about the data distribution, even if the model capacity is great enough to learn a trivial identity function.

* 1. **Sparse Autoencoders**

A sparse autoencoder is simply an autoencoder whose training criterion involves a sparsity penalty Ω(h) on the code layer h, in addition to the reconstruction error:

L (x, g(f(x))) + Ω(h)

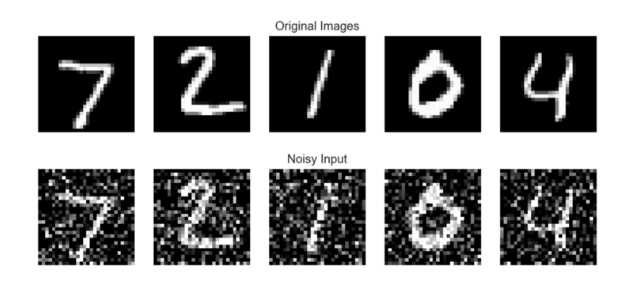
Where g(h) is the decoder output, and typically we have h=f(x), the encoder output. Sparse autoencoders are typically used to learn features for another task, such as classiﬁcation. An autoencoder that has been regularized to be sparse must respond to unique statistical features of the dataset it has been trained on, rather than simply acting as an identity function. In this way, training to perform the copying task with a sparsity penalty can yield a model that has learned useful features as a by product. We can think of the penalty Ω(h) simply as a regularizer term added to a feedforward network whose primary task is to copy the input to the output (unsupervised learning objective) and possibly also perform some supervised task (with a supervised learning objective) that depends on these sparse features.

It’s pretty easy to add regularization in Keras, we need to add just one parameter called **activity\_regularizer** by specifying the regularization strength. This is typically a value in the range [0.001, 0.000001]. Here we chose 10e-6

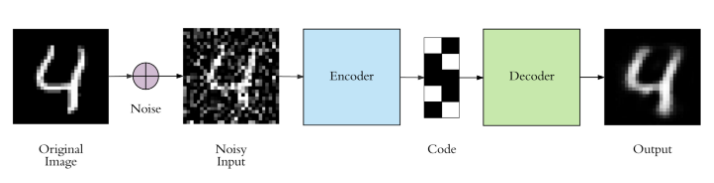
code = Dense(code\_size, activation='relu', activity\_regularizer=l1(10e-6))(input\_img)

* 1. **Denoising Autoencoders**

Keeping the code layer small forced our autoencoder to learn an intelligent representation of the data. There is another way to force the autoencoder to learn useful features, which is adding random noise to its inputs and making it recover the original noise-free data. This way the autoencoder can’t simply copy the input to its output because the input also contains random noise. We are asking it to subtract the noise and produce the underlying meaningful data. This is called a denoising autoencoder.



The top row contains the original images. We add random Gaussian noise to them and the noisy data becomes the input to the autoencoder. The autoencoder doesn’t see the original image at all. But then we expect the autoencoder to regenerate the noise-free original image.



There is only one small difference between the implementation of denoising autoencoder and the regular one. The architecture doesn’t change at all, only the fit function. We trained the regular autoencoder as follows:

autoencoder.fit(x\_train, x\_train)

Denoising autoencoder is trained as:

autoencoder.fit(x\_train\_noisy, x\_train)

Simple as that, everything else is exactly the same. The input to the autoencoder is the noisy image, and the expected target is the original noise free one.

1. **Application/Usage**

They have 3 common use cases though:

**Data denoising:** we have seen an example of this on images.

**Dimensionality reduction:** visualizing high-dimensional data is challenging. t-SNE is the most commonly used method but struggles with large number of dimensions (typically above 32). So autoencoders are used as a preprocessing step to reduce the dimensionality, and this compressed representation is used by t-SNE to visualize the data in 2D space.

**Variational Autoencoders (VAE):** this is a more modern and complex use-case of autoencoders and we will cover them in another article. But as a quick summary, VAE learns the parameters of the probability distribution modeling the input data, instead of learning an arbitrary function in the case of vanilla autoencoders. By sampling points from this distribution we can also use the VAE as a generative model.

1. **Conclusion**

Autoencoders are a very useful dimensionality reduction technique. They are very popular as a teaching material in introductory deep learning courses, most likely due to their simplicity.

1. **Reference**

* <https://www.deeplearningbook.org>
* <https://arxiv.org/abs/2003.05991>