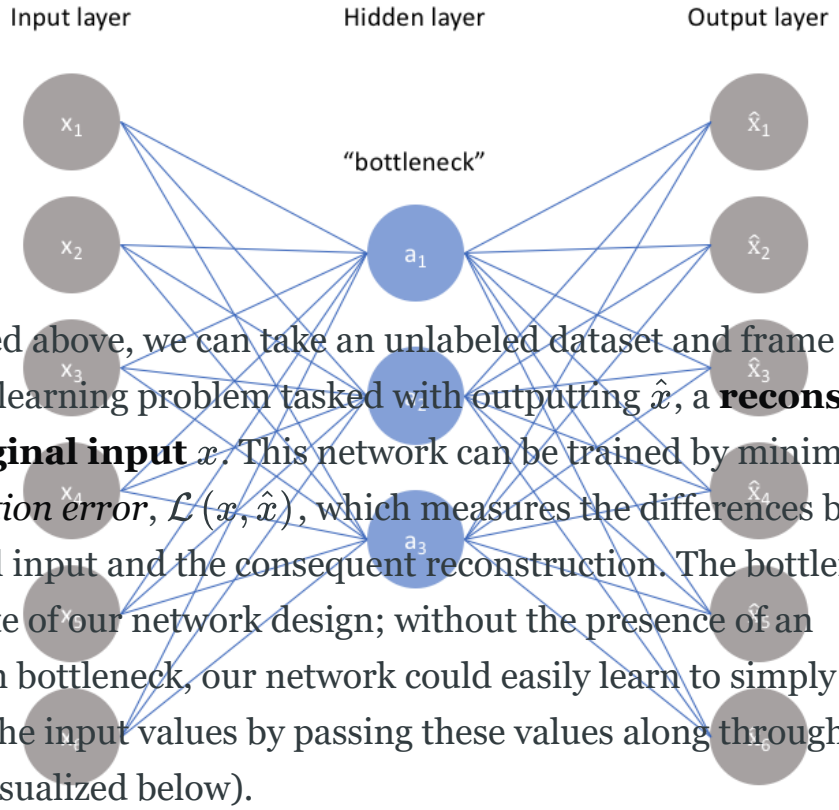
DATA SCIENCE

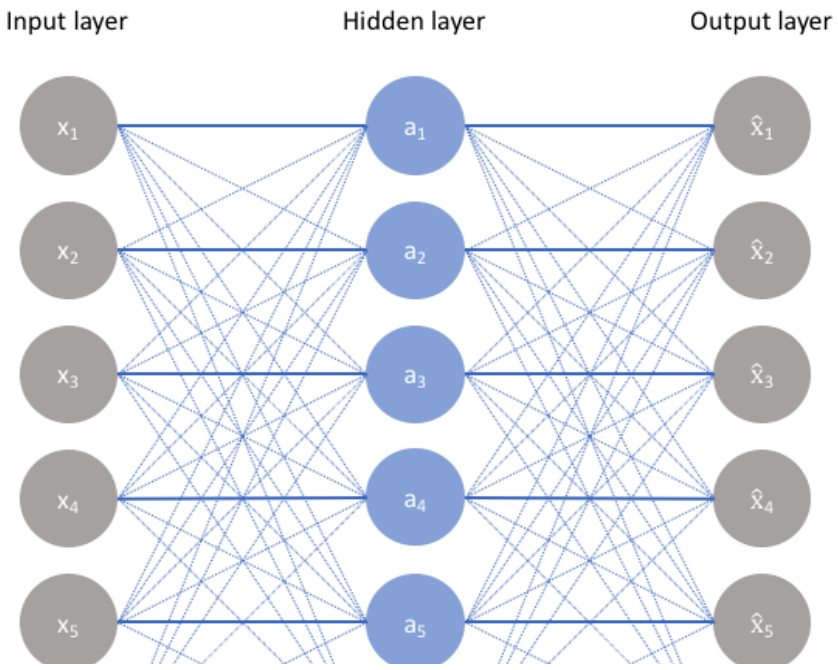
Introduction to autoencoders.

JEREMY JORDAN
19 MAR 2018 • 10 MIN READ

Autoencoders are an unsupervised learning technique in which we leverage neural networks for the task of **representation learning**. Specifically, we'll design a neural network architecture such that we *impose a bottleneck in the network which forces a **compressed** knowledge representation of the original input*. If the input features were each independent of one another, this compression and subsequent reconstruction would be a very difficult task. However, if some sort of structure exists in the data (ie. correlations between input features), this structure can be learned and consequently leveraged when forcing the input through the network's bottleneck.



As visualized above, we can take an unlabeled dataset and frame it as a supervised learning problem tasked with outputting $\hat{x}$, a **reconstruction of the original input** x. This network can be trained by minimizing the *reconstruction error*, $\mathcal{L}(x, \hat{x})$, which measures the differences between our original input and the consequent reconstruction. The bottleneck is a key attribute of our network design; without the presence of an information bottleneck, our network could easily learn to simply memorize the input values by passing these values along through the network (visualized below).



full network, forcing a learned compression of the input data.

Note: In fact, if we were to construct a linear network (ie. without the use of nonlinear activation functions at each layer) we would observe a similar dimensionality reduction as observed in PCA. See Geoffrey Hinton's discussion of this here.

The ideal autoencoder model balances the following:

- Sensitive to the inputs enough to accurately build a reconstruction.
- Insensitive enough to the inputs that the model doesn't simply memorize or overfit the training data.

This trade-off forces the model to maintain only the variations in the data required to reconstruct the input without holding on to redundancies within the input. For most cases, this involves constructing a loss function where one term encourages our model to be sensitive to the inputs (ie. reconstruction loss $\mathcal{L}(x, \hat{x})$) and a second term discourages memorization/overfitting (ie. an added regularizer).

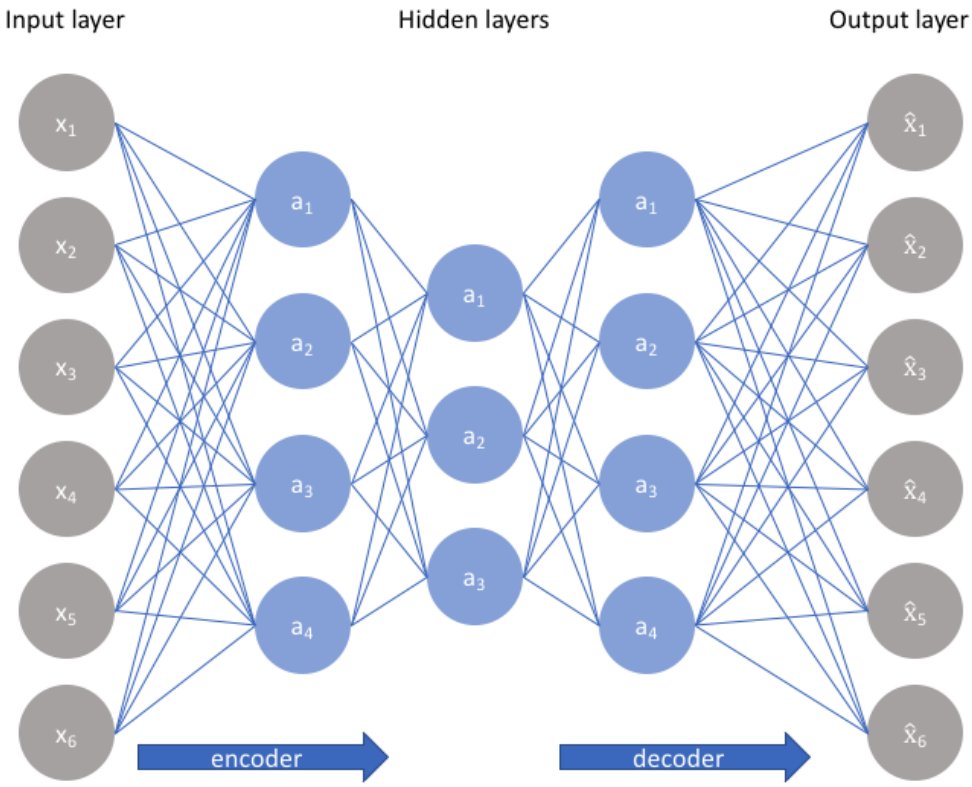
$$\mathcal{L}(x, \hat{x}) + regularizer$$

We'll typically add a scaling parameter in front of the regularization term so that we can adjust the trade-off between the two objectives.

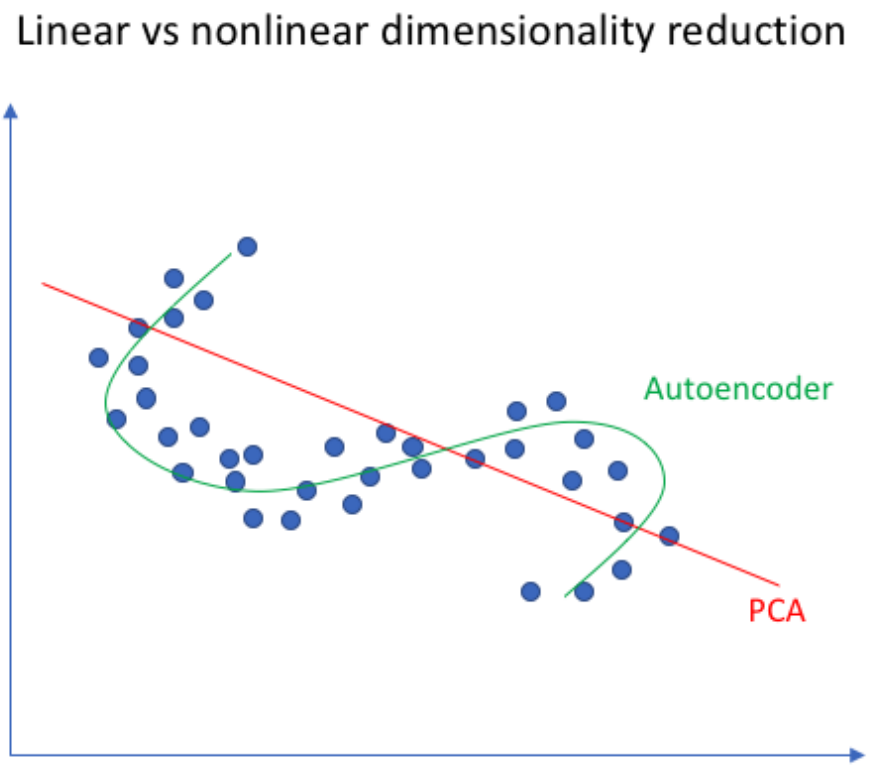
In this post, I'll discuss some of the standard autoencoder architectures for imposing these two constraints and tuning the trade-off; in a follow-up post I'll discuss variational autoencoders which builds on the concepts

Undercomplete autoencoder

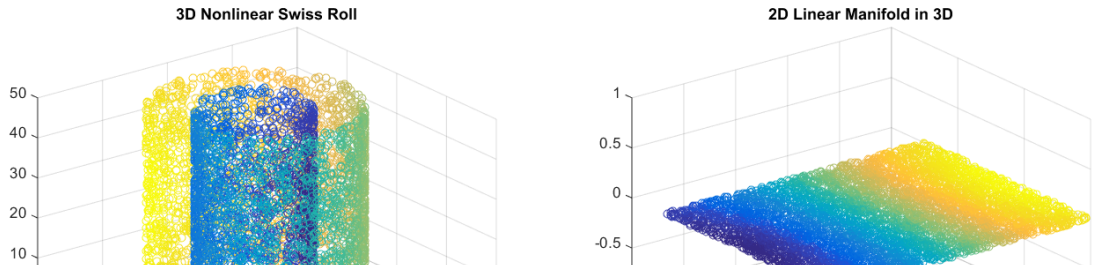
limiting the amount of information that can flow through the network. By penalizing the network according to the reconstruction error, our model can learn the most important attributes of the input data and how to best reconstruct the original input from an "encoded" state. Ideally, this encoding will **learn and describe latent attributes of the input data**.



Because neural networks are capable of learning nonlinear relationships, this can be thought of as a more powerful (nonlinear) generalization of PCA. Whereas PCA attempts to discover a lower dimensional hyperplane which describes the original data, autoencoders are capable of learning nonlinear manifolds (a manifold is defined in *simple* terms as a



For higher dimensional data, autoencoders are capable of learning a complex representation of the data (manifold) which can be used to describe observations in a lower dimensionality and correspondingly decoded into the original input space.



-20 -20 0 0

Image credit

An undercomplete autoencoder has no explicit regularization term - we simply train our model according to the reconstruction loss. Thus, our only way to ensure that the model isn't memorizing the input data is the ensure that we've sufficiently restricted the number of nodes in the hidden layer(s).

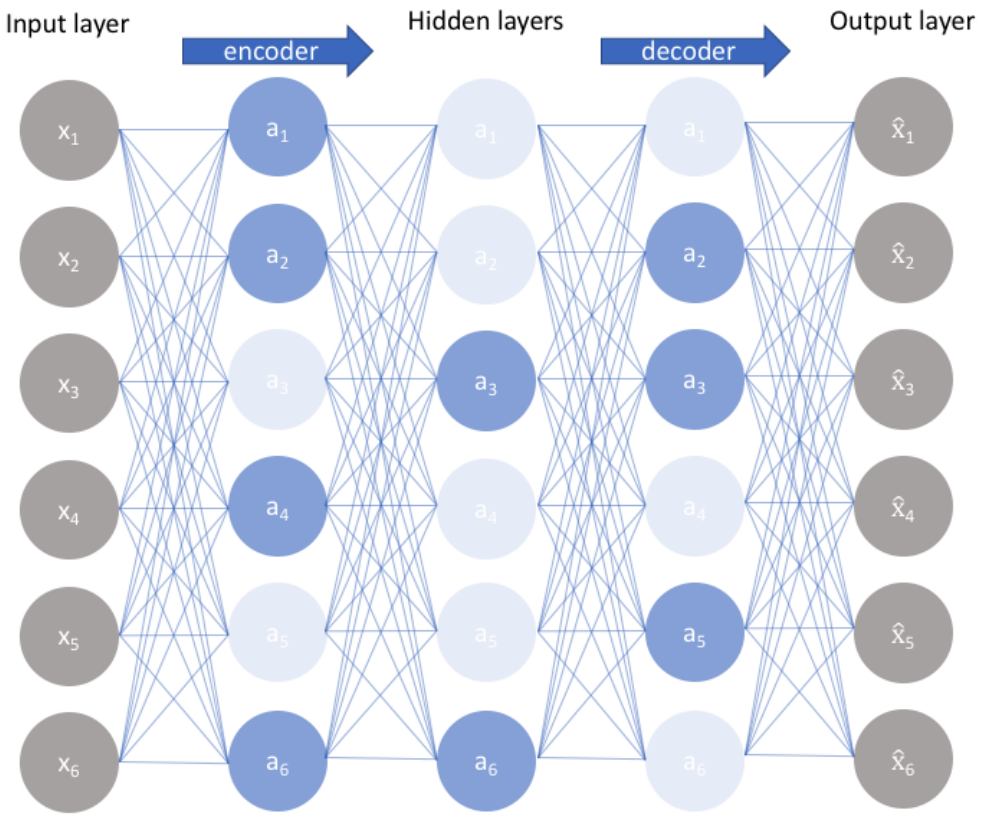
For deep autoencoders, we must also be aware of the *capacity* of our encoder and decoder models. Even if the "bottleneck layer" is only one hidden node, it's still possible for our model to memorize the training data provided that the encoder and decoder models have sufficient capability to learn some arbitrary function which can map the data to an index.

Given the fact that we'd like our model to discover latent attributes within our data, it's important to ensure that the autoencoder model is not simply learning an efficient way to memorize the training data. Similar to supervised learning problems, we can employ various forms of regularization to the network in order to encourage good generalization properties; these techniques are discussed below.

Sparse autoencoders

Sparse autoencoders offer us an alternative method for introducing an information bottleneck without *requiring* a reduction in the number of nodes at our hidden layers. Rather, we'll construct our loss function such that we penalize *activations* within a layer. For any given observation, we'll encourage our network to learn an encoding and decoding which only relies on activating a small number of neurons. It's worth noting that this is a different approach towards regularization, as we normally regularize the *weights* of a network, not the activations.

node corresponds with the level of activation. It's important to note that the individual nodes of a trained model which activate are *data-dependent*, different inputs will result in activations of different nodes



One result of this fact is that **we allow our network to sensitize individual hidden layer nodes toward specific attributes of the input data**. Whereas an undercomplete autoencoder will use the entire network for every observation, a sparse autoencoder will be forced to selectively activate regions of the network depending on the input data. As a result, we've limited the network's capacity to memorize the input data without limiting the networks capability to extract features from the data. This allows us to consider the latent state representation and regularization of the network *separately*, such that we can choose a latent state representation (ie. encoding dimensionality) in accordance with what makes sense given the context of the data while imposing

There are two main ways by which we can impose this sparsity constraint; both involve measuring the hidden layer activations for each training

batch and adding some term to the loss function in order to penalize excessive activations. There terms are:

- **L1 Regularization**: We can add a term to our loss function that penalizes the absolute value of the vector of activations a in layer h for observation i, scaled by a tuning parameter λ.

$$\mathcal{L}\left(x, \hat{x}\right) + \lambda \sum_{i} \left| a_i^{(h)} \right|$$

- **KL-Divergence**: In essence, KL-divergence is a measure of the difference between two probability distributions. We can define a sparsity parameter ρ which denotes the average activation of a neuron over a collection of samples. This expectation can be calculated as $\hat{\rho}_j = \frac{1}{m} \sum_{i} \left[a_i^{(h)}\left(x\right) \right]$ where the subscript j denotes the specific neuron in layer h, summing the activations for m training observations denoted individually as x. In essence, by constraining the average activation of a neuron over a collection of samples we're encouraging neurons to only fire for a subset of the observations. We can describe ρ as a Bernoulli random variable distribution such that we can leverage the KL divergence (expanded below) to compare the ideal distribution ρ to the observed distributions over all hidden layer nodes $\hat{\rho}$.

$$\mathcal{L}\left(x, \hat{x}\right) + \sum_{j} KL\left(\rho || \hat{\rho}_j\right)$$

Note: A Bernoulli distribution is "the probability distribution of a

establishing the probability a neuron will fire.

$\sum_{j=1}^{l^{(h)}} \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j}$. This loss term is visualized below for an ideal distribution of $\rho = 0.2$, corresponding with the minimum (zero) penalty at this point.

Denoising autoencoders

So far I've discussed the concept of training a neural network where the input and outputs are identical and our model is tasked with reproducing the input as closely as possible while passing through some sort of information bottleneck. Recall that I mentioned we'd like our autoencoder to be sensitive enough to recreate the original observation but insensitive enough to the training data such that the model learns a generalizable encoding and decoding. Another approach towards developing a generalizable model is to slightly corrupt the input data but still maintain the uncorrupted data as our target output.

With this approach, **our model isn't able to simply develop a mapping which memorizes the training data because our input and target output are no longer the same**. Rather, the model learns a vector field for mapping the input data towards a lower-dimensional manifold (recall from my earlier graphic that a manifold describes the high density region where the input data concentrates); if this manifold accurately describes the natural data, we've effectively "canceled out" the

Image credit