Autoencoders

Indro Spinelli

Fundamentals of Data Science

Adapted from Roger Grosse, Stephen Schott, Paul Quint, Ian Goodfellow and Geoffrey Hinton

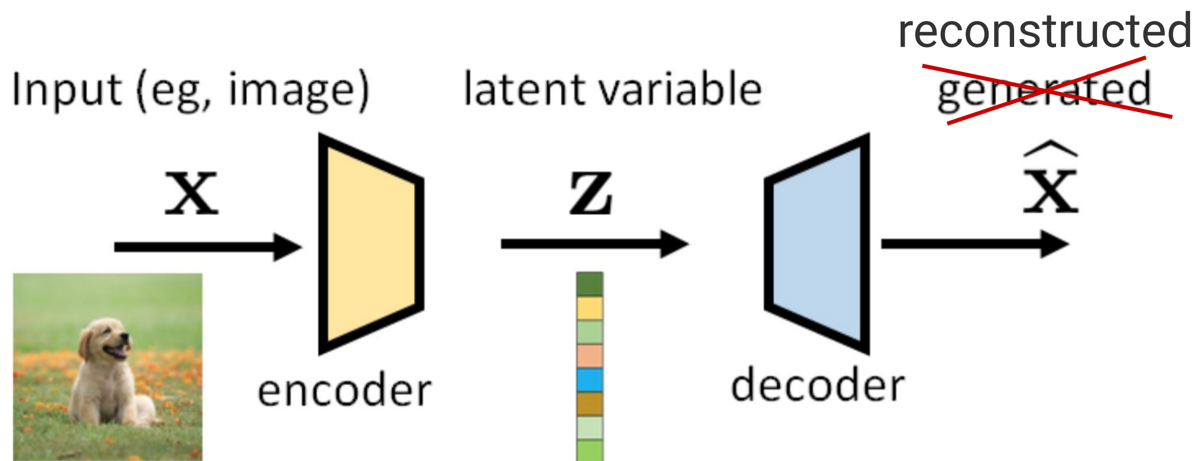
In short

An autoencoder is a type of feed-forward neural network designed to reconstruct its input x by predicting x^{hat}, where x^{hat} is an approximation of the original input.

The autoencoder incorporates a hidden layer with a significantly smaller dimensionality than the input. This bottleneck forces the network to learn a compressed, efficient representation of the data, capturing only the most essential features necessary for reconstruction.

This constraint ensures that the network does not simply memorize the input but instead learns meaningful patterns and structures in the data.

Autoencoder

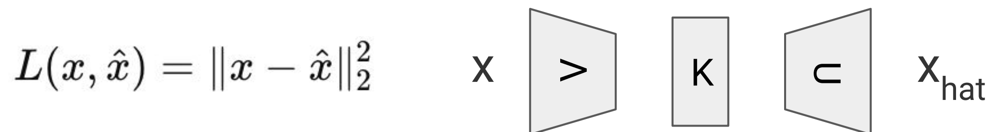


Why Use Autoencoders?

1. **Dimensionality Reduction for Visualization**
   Autoencoders can map high-dimensional data to a lower-dimensional space (e.g., 2D or 3D) to make it easier to visualize and interpret complex patterns.
2. **Data Compression**
   By learning efficient representations, autoencoders can reduce the file size of data while retaining key information, making them useful for tasks like image or audio compression.
3. **Feature Learning for Downstream Tasks**
   Autoencoders learn abstract and meaningful features from data in an unsupervised manner. These learned features can then be applied to supervised tasks, such as classification or regression, often improving performance.
4. **Utilizing Unlabeled Data**
   Autoencoders are particularly valuable when labeled data is scarce. Since they do not rely on labels, they can leverage large amounts of unlabeled data to extract useful patterns, making them a powerful tool in real-world scenarios where labeling is expensive or impractical.

Linear Autoencoder

The simplest form of an autoencoder consists of a single hidden layer, uses linear activation functions, and optimizes a reconstruction loss based on the squared error. The loss function is defined as:



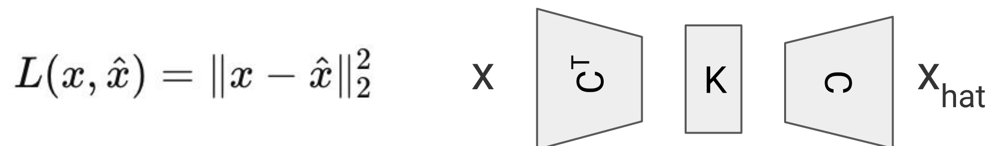
In this setup, the autoencoder computes the reconstruction as a linear transformation: $x_{hat} = UVx$

- U and V are weight matrices of the decoder and encoder, respectively.
- Vx represents the encoded representation (or compressed latent representation) of x.
- UVx reconstructs x from its encoded representation.

Because the activations are linear, the overall mapping $x_{hat} = UVx$ is also linear. This makes the network equivalent to **principal component analysis (PCA)**, where the hidden layer learns a lower-dimensional projection of the input that minimizes the reconstruction error in terms of squared distances.

Linear Autoencoder

The simplest form of an autoencoder consists of a single hidden layer, uses linear activation functions, and optimizes a reconstruction loss based on the squared error. The loss function is defined as:



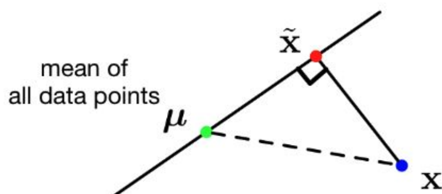
In this setup, the autoencoder computes the reconstruction as a linear transformation: $x_{hat} = CC^Tx$

- C and C^T are weight matrices of the decoder and encoder, respectively.
- C^Tx represents the encoded representation (or compressed latent representation) of x.
- CC^Tx reconstructs x from its encoded representation.

Because the activations are linear, the overall mapping is also linear. This makes the network equivalent to **principal component analysis (PCA)**, where the hidden layer learns a lower-dimensional projection of the input that minimizes the reconstruction error in terms of squared distances.

Pythagora

The autoencoder should learn to choose the subspace which minimizes the squared distance from the data to the projections. This is equivalent to the subspace which maximizes the variance of the projections.



By the Pythagorean Theorem,

$$\underbrace{\frac{1}{N}\sum_{i=1}^{N}\|\tilde{\mathbf{x}}^{(i)}-\boldsymbol{\mu}\|^2}_{\text{projected variance}}+\underbrace{\frac{1}{N}\sum_{i=1}^{N}\|\mathbf{x}^{(i)}-\tilde{\mathbf{x}}^{(i)}\|^2}_{\text{reconstruction error}}$$

$$=\underbrace{\frac{1}{N}\sum_{i=1}^{N}\|\mathbf{x}^{(i)}-\boldsymbol{\mu}\|^2}_{\text{constant}}$$

We do not solve by training but use the closed form solution of the PCA.

PCA on Eigenface

Eigenfaces are the principal components (or eigenvectors) of a large set of facial images.

These components capture the most important variations in facial structure across the dataset.

Each face can then be represented as a combination of these eigenfaces, reducing the dimensionality of the data.

How to compute them?

Collect a Dataset of Faces

- Start with a set of grayscale images of faces, typically resized to a fixed size (e.g., $m \times n$ pixels).
- Flatten each image into a vector, so an image of $m \times n$ pixels becomes a vector of length $m \cdot n$.

Center the Data

- Compute the mean face by averaging all the face vectors in the dataset.
- Subtract this mean face from each face vector, so the data is centered around zero.

Compute the Covariance Matrix

- The covariance matrix captures the relationships between the pixel values across the dataset.

Perform PCA

- Find the eigenvectors and eigenvalues of the covariance matrix.
- The eigenvectors are the **eigenfaces**, and the corresponding eigenvalues indicate the amount of variance each eigenface captures.
- Sort the eigenfaces by their eigenvalues, keeping only the top k eigenfaces that capture most of the variance.

Representation

Each face can be represented as a **weighted sum of eigenfaces**

$$\text{Face} \approx \text{Mean Face} + \sum_{i=1}^{k} w_i \cdot \text{Eigenface}_i$$

w_i are the weights (or coefficients) for each eigenface, found by projecting the original face onto the eigenfaces.

This transformation compresses the face into a smaller set of coefficients instead of storing the full image.

Face Recognition

Training

- Compute eigenfaces from a training set of face images.
- Project each face in the training set onto the eigenfaces to get a set of weights (feature vectors).

Recognition

- Project a new face onto the eigenfaces to obtain its weights.
- Compare these weights to those of known faces using a distance metric
- The closest match identifies the face.

Limitations

Lighting and Pose Sensitivity

Eigenfaces are sensitive to variations in lighting, pose, and facial expressions.

Data Dependency

The eigenfaces are computed from the training dataset, so their effectiveness depends on the diversity and quality of the training data.

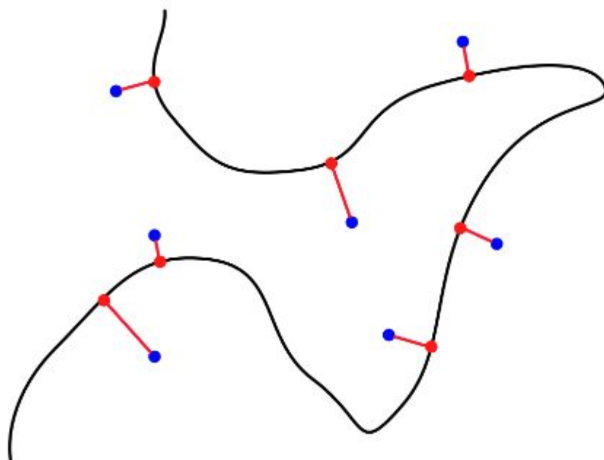
Nonlinearity

PCA is linear, so it struggles to capture complex, nonlinear facial variations.

Deep Autoencoders

Deep nonlinear autoencoders learn to project the data, not onto a subspace, but onto a nonlinear manifold.

This manifold is the image of the decoder.



Hidden Layers size

Undercomplete Autoencoders

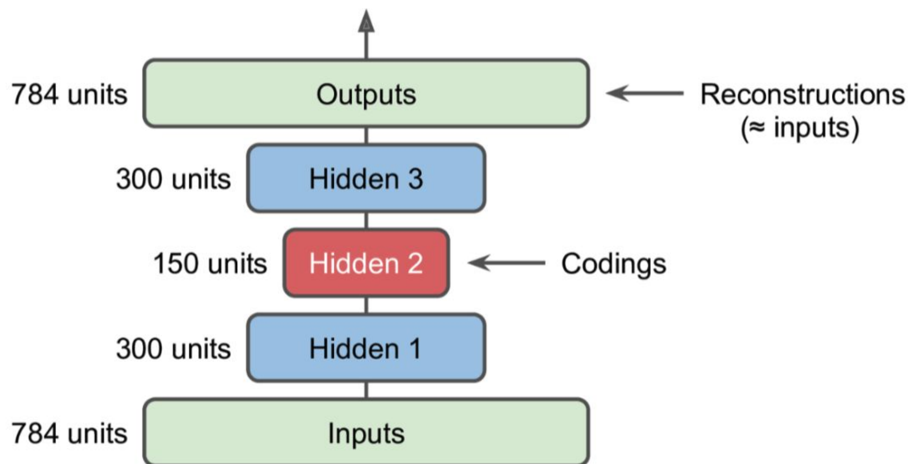
- The size of the hidden layer is smaller than the size of the input layer.
- **Key Feature**: The embedded space (latent representation) has a lower dimensionality than the input space.
- **Advantage**: Forces the model to learn compact, meaningful representations rather than simply memorizing the input data.
- **Limitation**: May lose some information if the input data is highly complex or high-dimensional.

Overcomplete Autoencoders

- The size of the hidden layer is much larger than the size of the input layer.
- **Key Feature**: Provides a high-capacity latent space, which can potentially capture richer representations of the data.
- **Challenge**: Without proper constraints, the model may overfit by simply copying the input to the output.
- **Solution**: Regularization techniques such as sparsity constraints (e.g., L1 regularization) or other penalties are applied to prevent overfitting and encourage meaningful representations.

Stacked Autoencoders

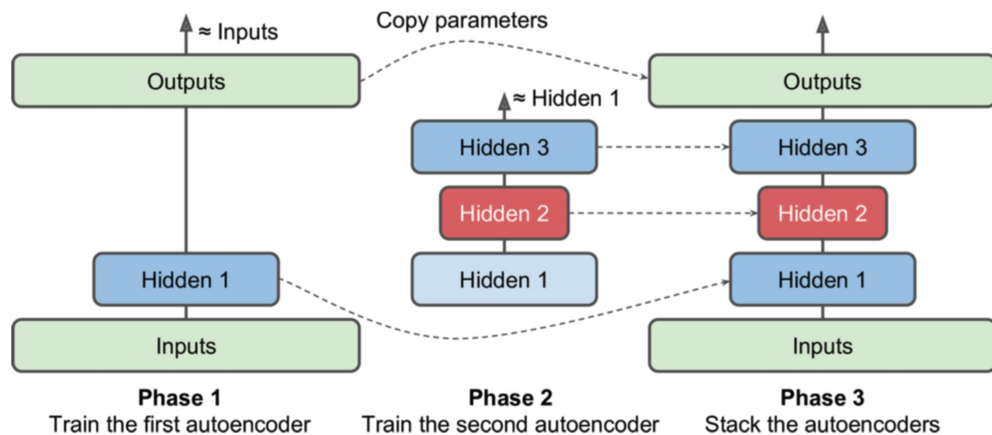
A stacked autoencoder consists of **multiple encoding layers** and **multiple decoding layers**. Introduces hierarchical representations, enabling it to capture increasingly abstract features of the input data.



Simplified Training

1. **Train Layer H_1**: Start with a single hidden layer autoencoder to reconstruct the input.
2. **Train Layer H_2**: Use H_1 's output as training data to train the next autoencoder.
3. **Stack Layers**: Combine H_2 with the first autoencoder and repeat the process for additional layers.

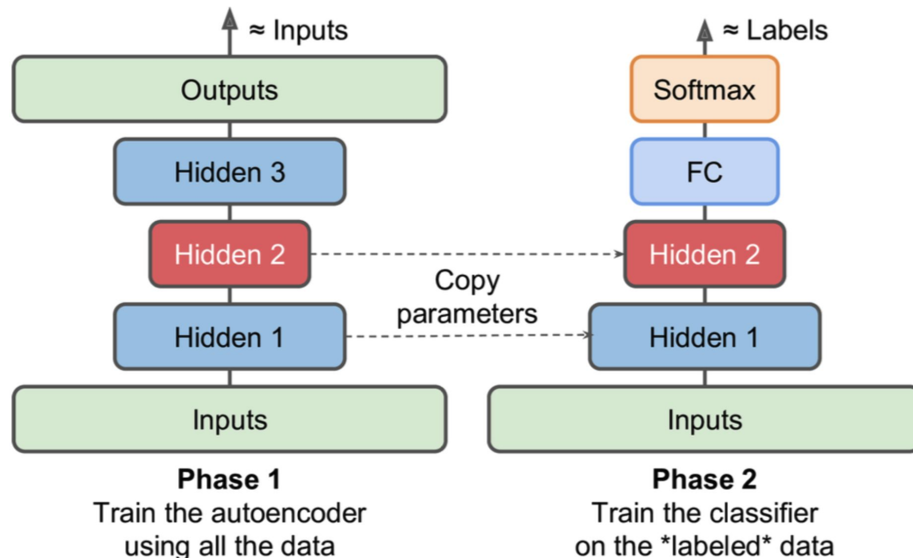
This layer-wise training simplifies optimization and ensures meaningful feature learning at each step.



Layers as Lego Pieces

You can pre-train the network using unlabeled data to learn meaningful features. Then, fine-tune the dense layer using labeled data for the specific task. Often called Semi-Supervised Learning.

If the data domain or the task changes it is called Transfer Learning.



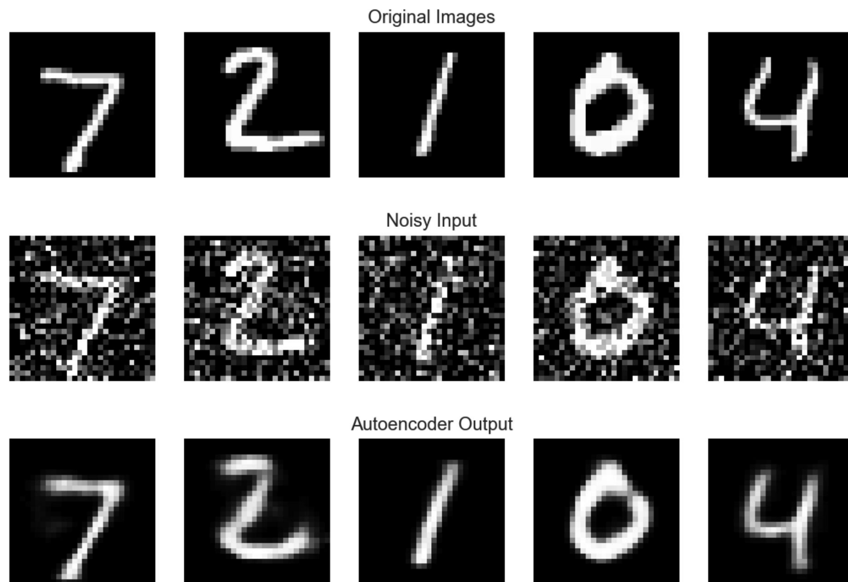
Denoising Autoencoders

A **denoising autoencoder** (DAE) is a variation of the standard autoencoder designed to learn robust, noise-resistant representations by training the model to reconstruct clean data from noisy inputs.

During training, the input data x is deliberately noised to create a corrupted version. This noise could be gaussian noise, salt and pepper, dropout on the input.

The autoencoder tries to map the noisy input back to the original, clean version x by learning to extract useful features from the noisy input.

By training on noisy data, denoising autoencoders learn to ignore irrelevant features or noise, focusing on the core structure of the data.



Sparse Autoencoders

A **sparse autoencoder** is an overcomplete autoencoder where the hidden layer has more units than the input, but we enforce **sparsity**—i.e., most hidden units should have zero activation.

$$\tilde{J}(x) = J(x, g(f(x))) + \alpha\Omega(h)$$

$J(x, g(f(x)))$ is the reconstruction error.

$\Omega(h)$ is the **sparsity penalty** function (e.g., L1 norm: $\sum_i |h_i|$)

α controls the penalty strength.

KL Divergence for Sparsity

Another approach is to penalize the average activation of the hidden units h_i across a mini-batch to match a user-specified target sparsity value p. The target sparsity p represents the proportion of neurons that should be "active" (non-zero). This can be achieved using the **Kullback-Leibler (KL) divergence** between the average activation q and the target sparsity p:

$$\mathrm{KL}(p, q) = p \log \frac{p}{q} + (1 - p) \log \frac{1 - p}{1 - q}$$

$q = \frac{1}{N} \sum_{i=1}^{N} h_i$ is the average activation of hidden units over the mini-batch.

p is the target sparsity value, such as 0.1 (10% of hidden units should be active).

Sparse Autoencoders Find Highly Interpretable Features in Language Models

Hoagy Cunningham[*12], **Aidan Ewart**[*13], **Logan Riggs**[*1], **Robert Huben**, **Lee Sharkey**[4]
[1]EleutherAI, [2]MATS, [3]Bristol AI Safety Centre, [4]Apollo Research
{hoagycunningham, aidanprattewart, logansmith5}@gmail.com