CHAPTER 4

Autoencoder

LEARNING OBJECTIVES

After reading this chapter, you will be able to

- Understand the features and architecture of autoencoders.
- Understand the types of autoencoders and their architecture.
- Understand the use of autoencoders in the real world.

4.1 | INTRODUCTION

Deep learning architectures like Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN) are useful for supervised learning. In this chapter, we will introduce a deep learning architecture useful in unsupervised learning – an autoencoder. An autoencoder is one such unsupervised learning technique. It is useful for pre-training deep networks and for dimensionality reduction. Dimensionality reduction is an important step in data preprocessing to improve the process of machine learning. Given a dataset, dimensionality reduction reduces the dimensionality of the dataset. A dataset contains many features, but only some features are helpful to process the data further. Dimensionality reduction is the technique used to identify the required features.

This chapter focusses mainly on how autoencoders can be used for dimensionality reduction. Dimensionality reduction reduces the number of dimensions of the input data. Previously known dimensionality reduction techniques like Principal Component Analysis (PCA), Linear Discriminant Analysis (LDA) and Non-negative Matrix Factorization (NMF) try to find the feature space that best describes the data. The objective of the autoencoder is to find the best latent representation of the input data.

An autoencoder is a feedforward network that uses the backpropagation algorithm to learn weights. It has a simpler architecture when compared to the deep learning architectures seen so far. It is a two-layer architecture [remember that the input layer is not counted as a layer in the artificial neural network (ANN) terminology]. The autoencoder has an input layer, a hidden layer and an output layer. The only difference is that the output should be the same as the input. This differentiates autoencoders from the architectures seen so far. That is, if (1, 1, 0, 0, 0, 1) is given as input, the autoencoder neural network tries to output (1, 1, 0, 0, 0, 1). This validates the requirement that the number of input neurons should be the same as the number of output neurons.

The hidden layer captures the best representative features of the input. That is, the hidden layer stores the representation of the input. Let us assume that the number of hidden nodes are much less than the number of input nodes (we will see later that this is not always the case). This type of autoencoder in which the dimension of the hidden layer is less than the dimension of the input layer is called *undercomplete autoencoder*. The values of the hidden layer are viewed as a compressed version of the input. In this example, let the number of input and output neurons be 6. The hidden layer has a smaller number of neurons than the number of neurons in the input layer. Let us have 2 neurons in the hidden layer. The autoencoder takes 6 features and encodes it using just 2 features. These 2 features are enough to reconstruct the 6 features. That is, we have just performed dimensionality reduction. The dimension of the original data is 6. The dimension of the dataset is reduced from 6 to 2; this is termed as *dimensionality reduction*.

4.2 | FEATURES OF AUTOENCODER

Autoencoders exhibit the following features:

1. **Data Dependent:** Autoencoders are compression techniques where the model can be used only on data in which they have already been trained. For example, the model of an autoencoder used to compress house images cannot be used to compress human faces.
2. **Lossy Compression:** Reconstruction of the original data from the compressed representation would result in a degraded output. This is illustrated in Fig. 4.1.

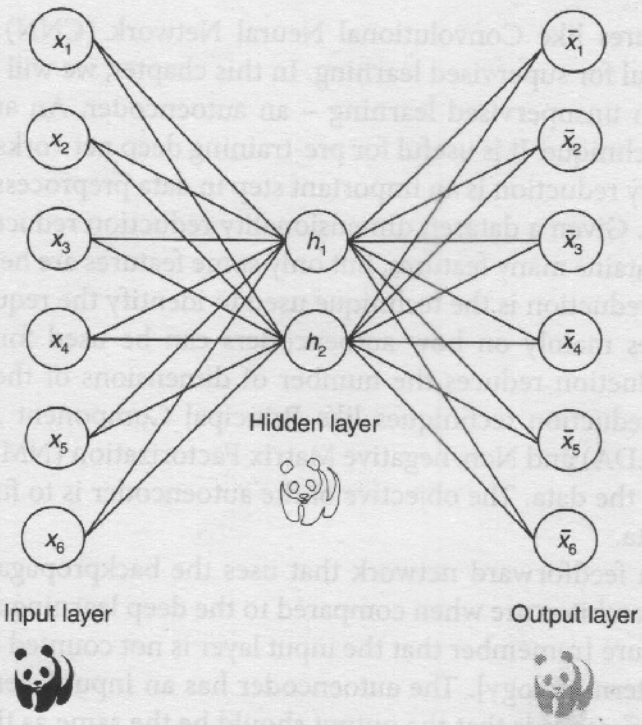

FIGURE 4.1 Lossy compression.

4.3 | TYPES OF AUTOENCODER

Let us learn about the following types of autoencoder:

1. Vanilla autoencoder.
2. Multilayer autoencoder.
3. Stacked autoencoder.
4. Deep autoencoder.
5. Denoising autoencoder.
6. Convolutional autoencoder.
7. Regularized autoencoder.

4.3.1 | Vanilla Autoencoder

Figure 4.2 is an illustration of a vanilla autoencoder. This is an ordinary autoencoder with no added features. It may be noted that the layers are fully connected.

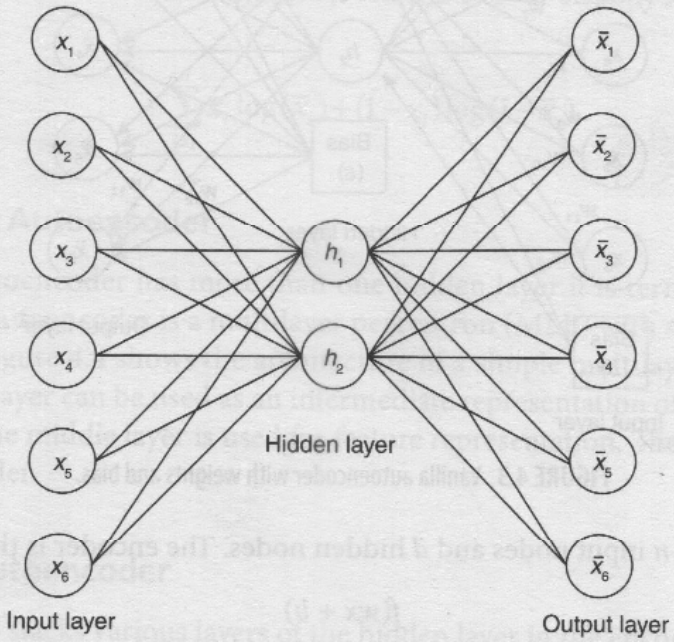

FIGURE 4.2 Vanilla autoencoder.

4.3.1.1 *Structure of Vanilla Autoencoder*

Autoencoders have three parts: the encoder, code and the decoder.

1. The **encoder** takes input from the input neurons. It then encodes the input and gives the output to the code. The encoder encompasses of the input layer and one or more hidden layers immediately after the input layer, finally resulting in the code. The encoder compresses the data.

2. The **code** is a hidden layer with reduced required number of nodes to represent the input.
3. The **decoder** is symmetric to the encoder. The final layer of the decoder is the output layer, where you get back the input data. It decodes the output of the code to produce a lossy reconstruction of the input and produces the final output.

The encoder takes (1, 1, 0, 0, 0, 1) as input and outputs (h_1, h_2). Code has the output (h_1, h_2). In an ideal case, the decoder takes (h_1, h_2) as input and outputs (1, 1, 0, 0, 0, 1). The vanilla autoencoder with weights and bias is shown in Fig. 4.3.

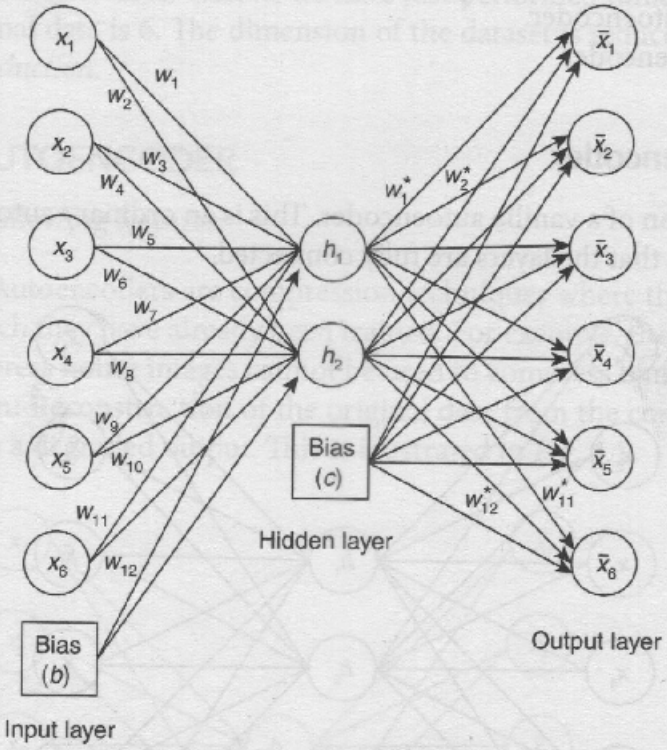

FIGURE 4.3 Vanilla autoencoder with weights and bias.

Let us assume there are n input nodes and d hidden nodes. The encoder is the function

$$f(wx + b)$$

where w is the real-valued vector $(w_1, w_2, \ldots, w_{12})$ and can be generalized as $w \in R^{d \times n}$; b is the real-valued vector (b_1, b_2) and can be generalized as $b \in R^d$; x is the input vector and can be generalized as $x \in R^n$.

Here $wx + b$ is a linear transformation followed by a non-linear transformation produced by the activation function f.

Similarly, the decoder is the function

$$g(w * \bar{x} + c)$$

where $w*$ is the real-valued vector $w_1*, w_2*, \ldots, w_{12}*$ and can be generalized as $w* \in R^{n \times d}$; c is the real-valued vector $(c_1, c_2 \ldots, c_6)$ and can be generalized as $c \in R^n$; y is the output vector and can be generalized as $\bar{x} \in R^n$.

Here $w * \bar{x} + c$ is a linear transformation followed by a non-linear transformation produced by the activation function g.

The choice of the activation function can differ depending on the input. For example, if the input values are binary then the activation function can be a logistic function.

4.3.1.2 Loss Function for Vanilla Autoencoder

The loss function is written in such a way that the output is the same as the input. The loss function used here is the mean squared error. For an autoencoder, it is the average of the squared error along all the input dimensions. If there are n input dimensions, then the mean squared error is

$$\min_{w,w*,B,C} \frac{1}{n} \sum_{i=1}^{n} (\bar{x}_i - x_i)^2$$

For a lossless compression, the loss function would be closer to 0. If the activation function is logistic, then the output would be in the range [0, 1]. Here, the cross-entropy loss function can be used, which is given by

$$-\sum_{i=1}^{n} x_i \log(\bar{x}_i) + (1 - x_i) \log(1 - \bar{x}_i)$$

4.3.2 | Multilayer Autoencoder

When the vanilla autoencoder has more than one hidden layer it is termed the multilayer autoencoder. Multilayer autoencoder is a multilayer perceptron (MLP) with symmetry in the encoder and decoder sides. Figure 4.4 shows the architecture of a simple multilayer autoencoder.

The value in any layer can be used as an intermediate representation of the features. But usually it is symmetric and the middle layer is used for feature representation. The loss function is like that of a vanilla autoencoder.

4.3.3 | Stacked Autoencoder

Stacked autoencoders stacks various layers of the hidden layer in the encoder and the decoder. The training does not involve training end-to-end as in multilayer perceptrons using the backpropagation algorithm. Rather, when there are multiple hidden layers, the first hidden layer (h^1) is trained and the parameters are identified using backpropagation. That is, the stacked autoencoder looks like a simple autoencoder as given in Fig. 4.5.

To train the second hidden layer (h^2), h^1 is the given in the input and output layers and h^2 is used as the code layer. To find h^3, we use h^2 and so on. This is shown in Fig. 4.6.

The size of the hidden layer continuously decreases, and each hidden layer is expected to learn an abstract feature. The final code layer can be given as input to a supervised learner to perform classification and regression. Figure 4.7 shows a binary classification.

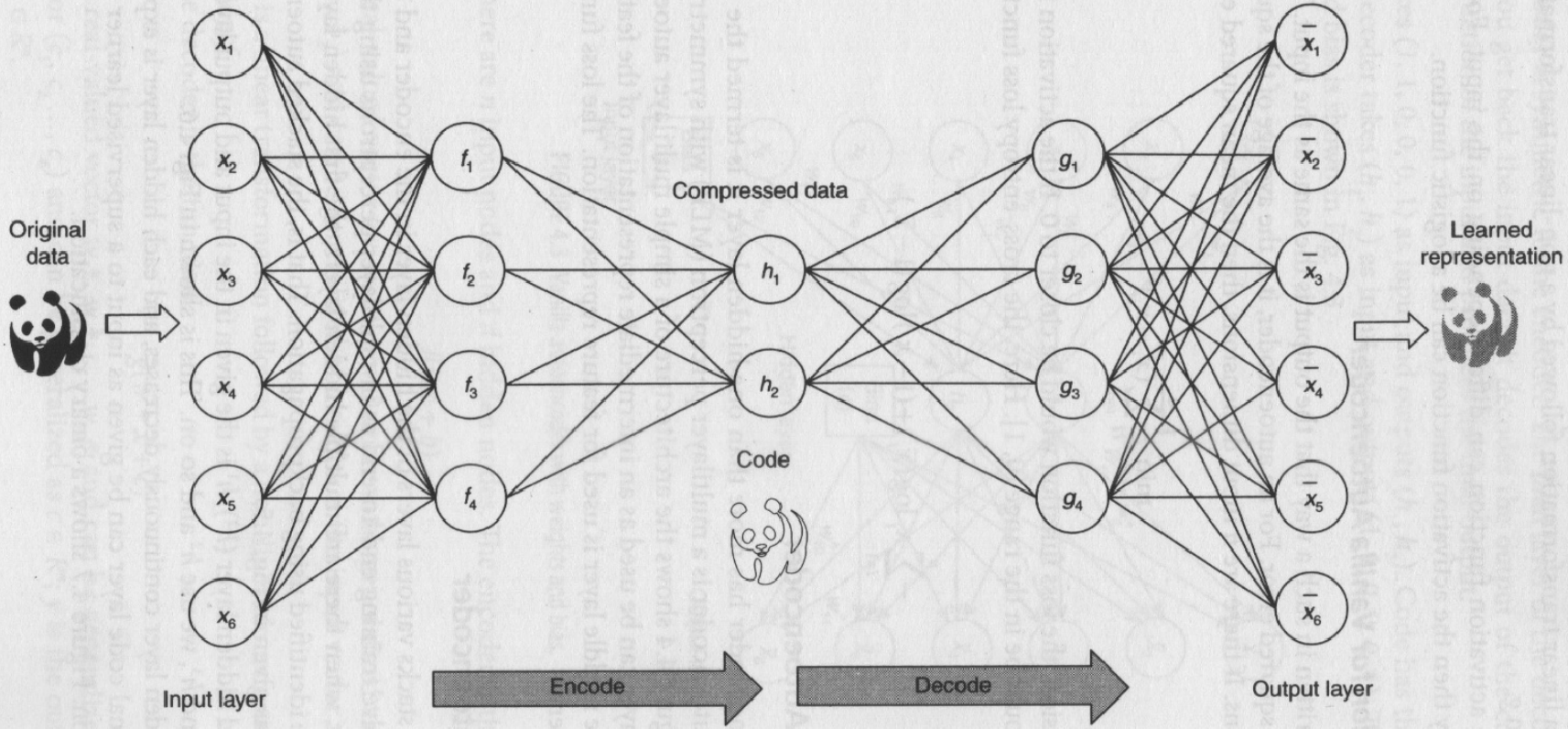

FIGURE 4.4 Simple multilayer autoencoder.

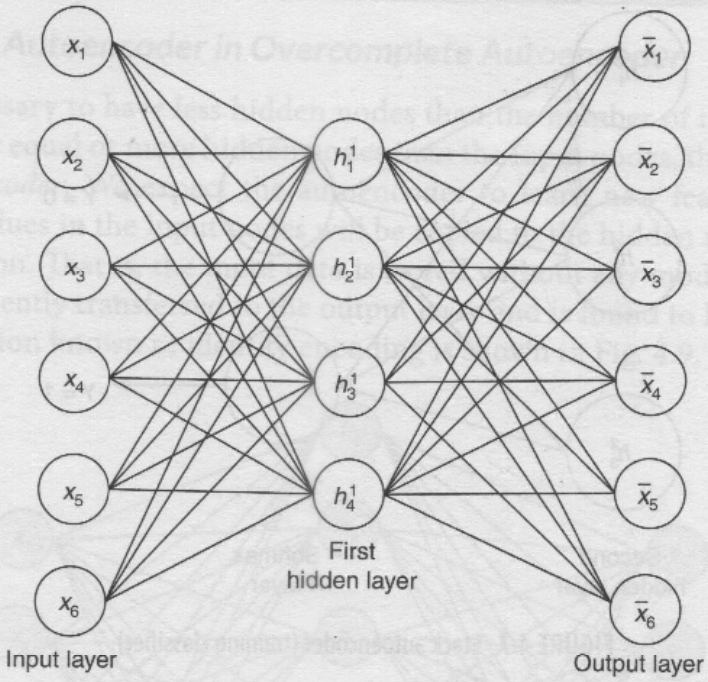

FIGURE 4.5 Stack autoencoder (training first hidden layer).

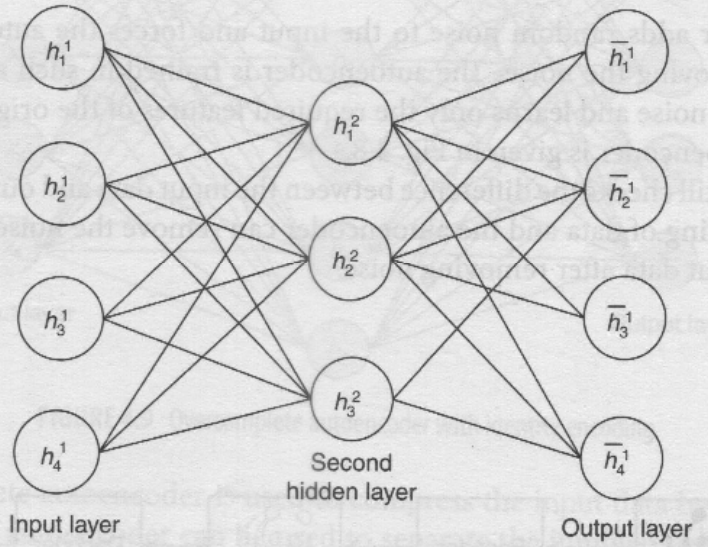

FIGURE 4.6 Stack autoencoder (training second hidden layer).

4.3.4 | Deep Autoencoder

A deep autoencoder comprises of two symmetric deep belief networks. Deep belief networks have been explained in Chapter 5. The layers are restricted Boltzmann machines because they are the building blocks of deep belief networks. The encoder is a deep belief network with four or five hidden layers. The decoder has the same number of layers as the encoder but it is a mirror image as given in Fig. 4.4.

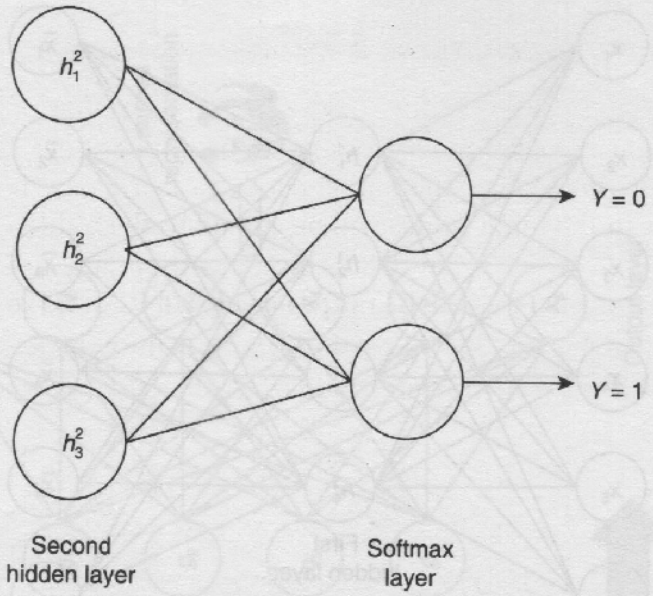

FIGURE 4.7 Stack autoencoder (training classifier).

4.3.5 | Denoising Autoencoder

Denoising autoencoder adds random noise to the input and forces the autoencoder to learn the original data after removing the noise. The autoencoder is trained in such a way that it identifies the noise, removes the noise and learns only the required features of the original data. The general architecture of the autoencoder is given in Fig. 4.8.

The loss function still checks the difference between the input data and output data. This ensures that there is no overfitting of data and the autoencoder can remove the noise and learn the important features of the input data after removing noise.

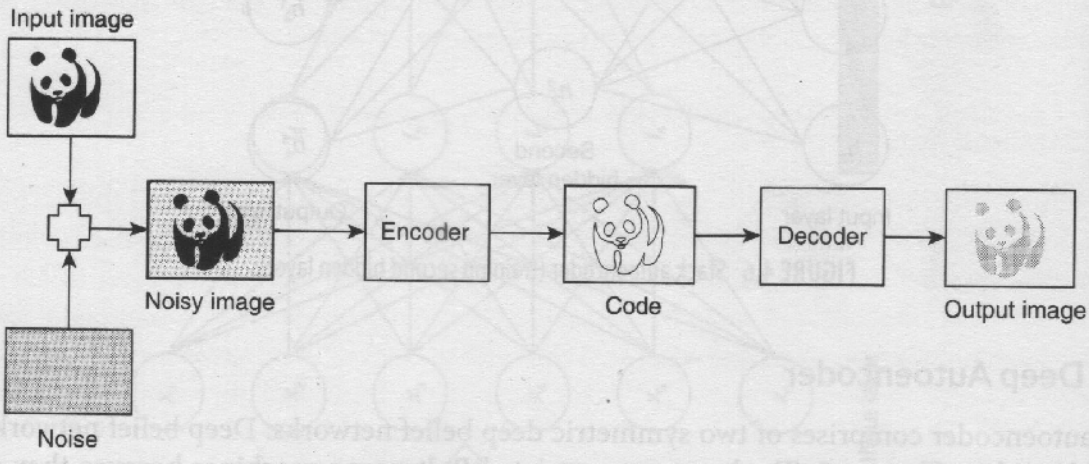

FIGURE 4.8 Denoising autoencoder.

4.3.5.1 Denoising Autoencoder in Overcomplete Autoencoder

It is not always necessary to have less hidden nodes than the number of input nodes in the code layer. When there are equal or more hidden nodes than the input nodes, the autoencoder is called *overcomplete autoencoder*. We expect the autoencoder to learn new features. But what might happen is that the values in the input nodes will be copied to the hidden nodes without learning any useful information. That is, the input data is stored without any modification in the hidden nodes and is subsequently transferred to the output layer and is found to have learnt the identity function. This condition known as identity encoding is shown in Fig. 4.9.

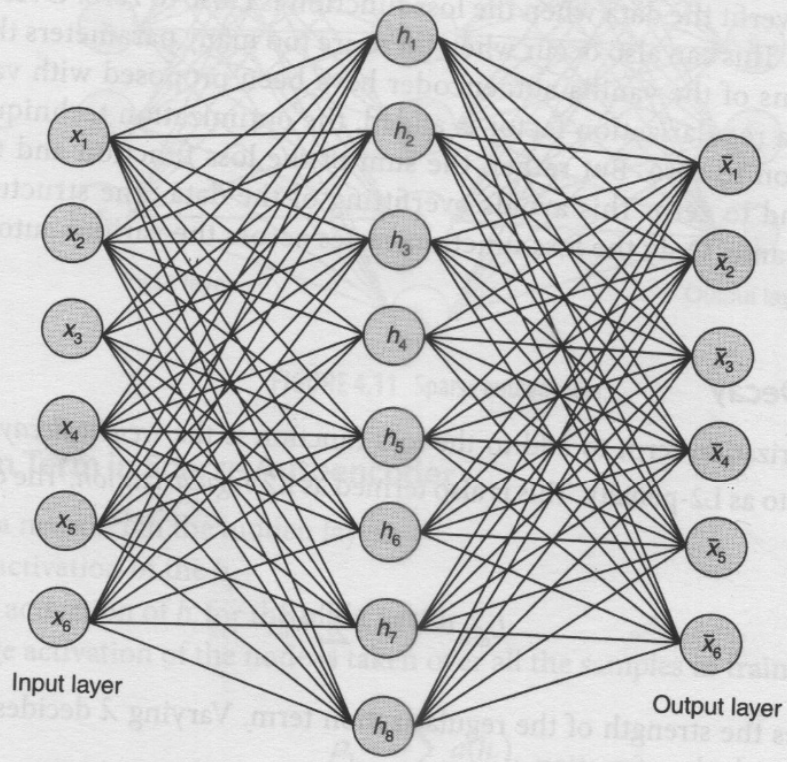

FIGURE 4.9 Overcomplete autoencoder with identity encoding.

Just as an undercomplete autoencoder is used to compress the input data by extracting useful features, an overcomplete autoencoder can be used to separate the jumbled features in an input data. Identity encoding can be avoided using denoising autoencoders.

4.3.6 | Convolutional Autoencoder

As the name suggests, a convolutional autoencoder combines the architecture of CNN and the vanilla autoencoder. While CNN create supervised machine learning models, autoencoders create unsupervised machine learning models. The encoder is a combination of the convolution layers and pooling layers as in a normal CNN. The pooling layers perform downsampling, while the decoder performs deconvolution and upsampling. The final layer gives the output which is expected to be the same as the input. A simple architecture is shown in Fig. 4.10.

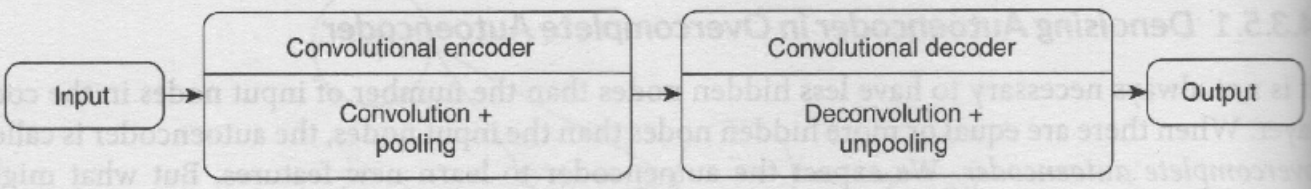

FIGURE 4.10 Convolutional autoencoder.

4.3.7 | Regularization in Autoencoder (Regularized Autoencoder)

Autoencoders can overfit the data when the loss function is close to zero. Overfitting can lead to poor generalization. This can also occur when there are too many parameters that fit the training data. Many variations of the vanilla autoencoder have been proposed with various regularization factors. When a regularization factor is added, the optimization technique does not try to take the loss function to zero. But rather, the sum of the loss function and the regularization term is made to tend to zero. This avoids overfitting of the data. The structure of the autoencoder remains the same. Only the loss function varies across the various autoencoders that use regularization.

4.3.7.1 *Weight Decay*

The simplest regularization term to add to the loss function is the *weight decay*, which is given by $\sum_{ij} w_{ij}^2$ and referred to as L2-penalty. This is also termed as *L2-regularization*. The overall loss function now becomes

$$L_{\text{new}} = L + \lambda \sum_{ij} w_{ij}^2$$

where λ determines the strength of the regularization term. Varying λ decides how much importance can be given to the loss function.

4.3.7.2 *Sparse Autoencoder*

As the name suggests, sparse autoencoders introduce sparsity into the autoencoders for the purpose of regularization. This method is also a type of regularization because we add a penalty term to the loss function to ensure that most of the nodes are inactive. A node is said to be inactive when it produces an output closer to zero. The sparsity constraint ensures that only a fraction of the input nodes is active at a time. The rest of the nodes are zero. The nodes should be active only when they must learn an important feature.

Overfitting occurs due to the numerous weight parameters that can take any value to learn the training data. Restricting the parameters helps avoid overfitting. The regularization term added in a sparse autoencoder is based on the Kullback–Leibler divergence (KL divergence). This is shown in Fig. 4.11, where the shaded nodes are inactive.

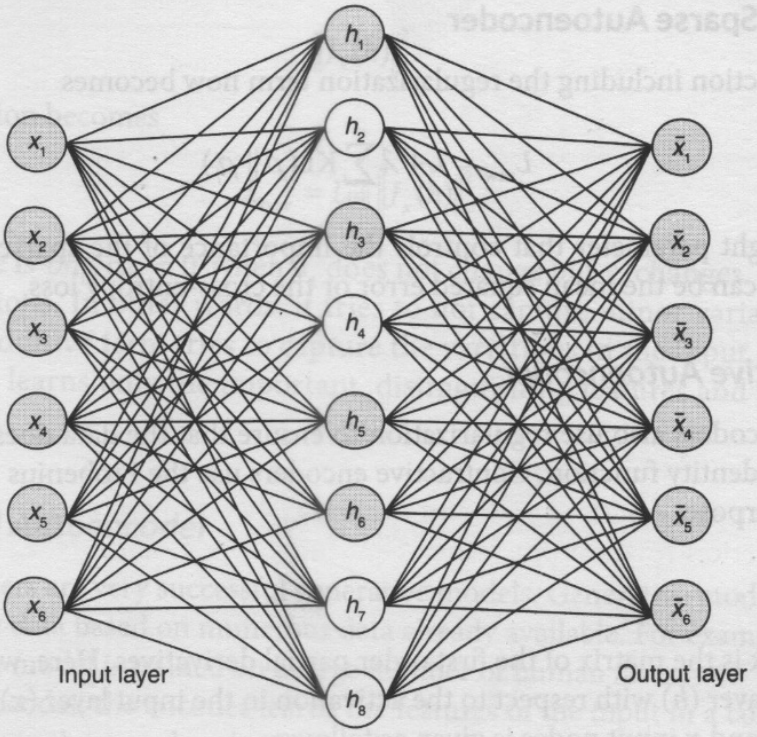

FIGURE 4.11 Sparse autoencoder.

Regularization Term in Sparse Autoencoder

Let h_l represent a neuron l in the hidden layer.
Let $a(h_l)$ be the activation of the h_l.
Let $a(h_l)_{x_i}$ be the activation of h_l for the input vector x_i.
Then, the average activation of the node is taken over all the samples in training set and is given by

$$\rho_l = \frac{1}{n}\sum_{i=1}^{n} a(h_l)_{x_i}$$

where n is the number of samples.
Sparse autoencoders make the average activation of the node tend to a value closer to zero. This value near zero is given by the sparsity parameter ρ. If $\rho = 0.03$, then the sparse autoencoder tries to make $\rho_l = \rho$.

To ensure this, the regularization term is chosen in such a way that ρ_l does not deviate much from ρ. The KL divergence is used for this purpose. Based on the KL divergence, the regularization term added to the loss function is given by

$$\sum_{j=1}^{m} \mathrm{KL}(\rho \,\|\, \rho_j) = \rho \log \frac{\rho}{\rho_j} + (1-\rho)\log \frac{1-\rho}{1-\rho_j}$$

where m is the number of neurons in the hidden layer.

Loss Function of Sparse Autoencoder

The overall loss function including the regularization term now becomes

$$L_{new} = L + \lambda \sum_{j=1}^{m} \mathrm{KL}(\rho \| \rho_j)$$

where λ is the weight parameter that controls the importance of the sparse regularization term. Here, loss function can be the mean squared error or the cross-entropy loss.

4.3.7.3 Contractive Autoencoder

Contractive autoencoders also use regularization to ensure that the data does not overfit nor does the code learn the identity function. Contractive encoders use the Frobenius norm of the Jacobian function for this purpose.

Jacobian Matrix

The Jacobian matrix is the matrix of the first order partial derivatives. Here, we measure the activation in the hidden layer (h) with respect to the activation in the input layer (x). The Jacobian matrix for k hidden nodes and n input nodes is given as follows:

$$J_x(h) = \begin{pmatrix} \frac{\partial h_1}{\partial x_1} & \frac{\partial h_1}{\partial x_2} & \cdots & \frac{\partial h_1}{\partial x_n} \\ \frac{\partial h_2}{\partial x_1} & \frac{\partial h_2}{\partial x_2} & \cdots & \frac{\partial h_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial h_k}{\partial x_1} & \frac{\partial h_k}{\partial x_2} & \cdots & \frac{\partial h_k}{\partial x_n} \end{pmatrix}$$

The first column is the partial derivative of every neuron in the hidden layer with respect to the first neuron in the input layer. The first column shows how much the neuron in the hidden layer reacts to neuron x_1. The second column is the partial derivative of every neuron in the hidden layer with respect to the second neuron in the input layer and so on.

Frobenius Norm

The Frobenius norm or Euclidean norm of a matrix (M) of order $n \times m$ is the square root of the sum of the squares of the elements of the matrix. The Frobenious norm of matrix M is given by

$$\|M\|_F = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{m} |m_{ij}|^2}$$

The regularization term used in contractive autoencoders is the square of the Frobenious norm of the Jacobian matrix $J_x(h)$. It is given by

$$\|J_x(h)\|_F^2$$

The overall loss function becomes

$$L_{new} = L + \|J_x(h)\|_F^2$$

The partial derivative is $\partial h_1 / \partial x_1 = 0$ when h_1 does not change but x_1 changes. That is, it tries not to capture the variations. In other words, it tries to not capture minor variations in the input data. Also, the first additive term tries to capture the variations in the input data. This ensures that the autoencoder learns only the important, distinguishing features and ignores the minor feature variations.

4.3.7.4 Variational Autoencoder

Variational autoencoders are very successful generative models. Generative models are models that generate or create new data based on numerous data already available. For example, a new face can be generated when the model is trained on a large number of human faces.

In a vanilla autoencoder, the encoder learns the features of the input in a compressed form. In variational autoencoders, the encoder also learns the features of the input, but outputs a vector of means and standard deviations. If c is a sample in the unit Gaussian distribution (a unit Gaussian distribution has mean 0 and standard deviation 1), then $\sigma x + \mu$ is a sample with mean μ and standard deviation σ. The decoder takes as input a sample from the vector of means and standard deviations. This helps the decoder to generate an output in the category of the input data, but different from the actual input data. This is shown in the following example.

Let

Output vector (μ) = [0.2, 0.9, 0.3, 0.7, ...]
Output vector (σ) = [0.3, 0.4, 1, 1.3, ...]

Then the samples generated will be

$$[X_{1\sim N(0.2,0.3^2)}, X_{2\sim N(0.9,0.4^2)}, X_{3\sim N(0.3,1^2)}, X_{4\sim N(0.7,1.3^2)}, \ldots]$$

From this, a sample of encoding vectors are stochastically generated. The architecture of the variational autoencoder is represented in Fig. 4.12.

Loss Function of Variational Autoencoder

The loss function is a combination of reconstruction loss and latent loss. The reconstruction loss is the usual mean squared error. The latent loss is the KL divergence and it measures how closely the latent variables match the unit Gaussian distribution ($N(0,1)$). The loss function can be written as

$$L_{new} = -\frac{1}{2}\sum_{j=1}^{J}\left[1+\log(\sigma_j^2)-\mu_j^2+\sigma_j^2\right] + E\left|\sum_{i=1}^{D}\left[x_i \log y_i + (1-x_i)\cdot\log(1-y_i)\right]\right|$$

where J is the size of the standard deviation and mean vectors.

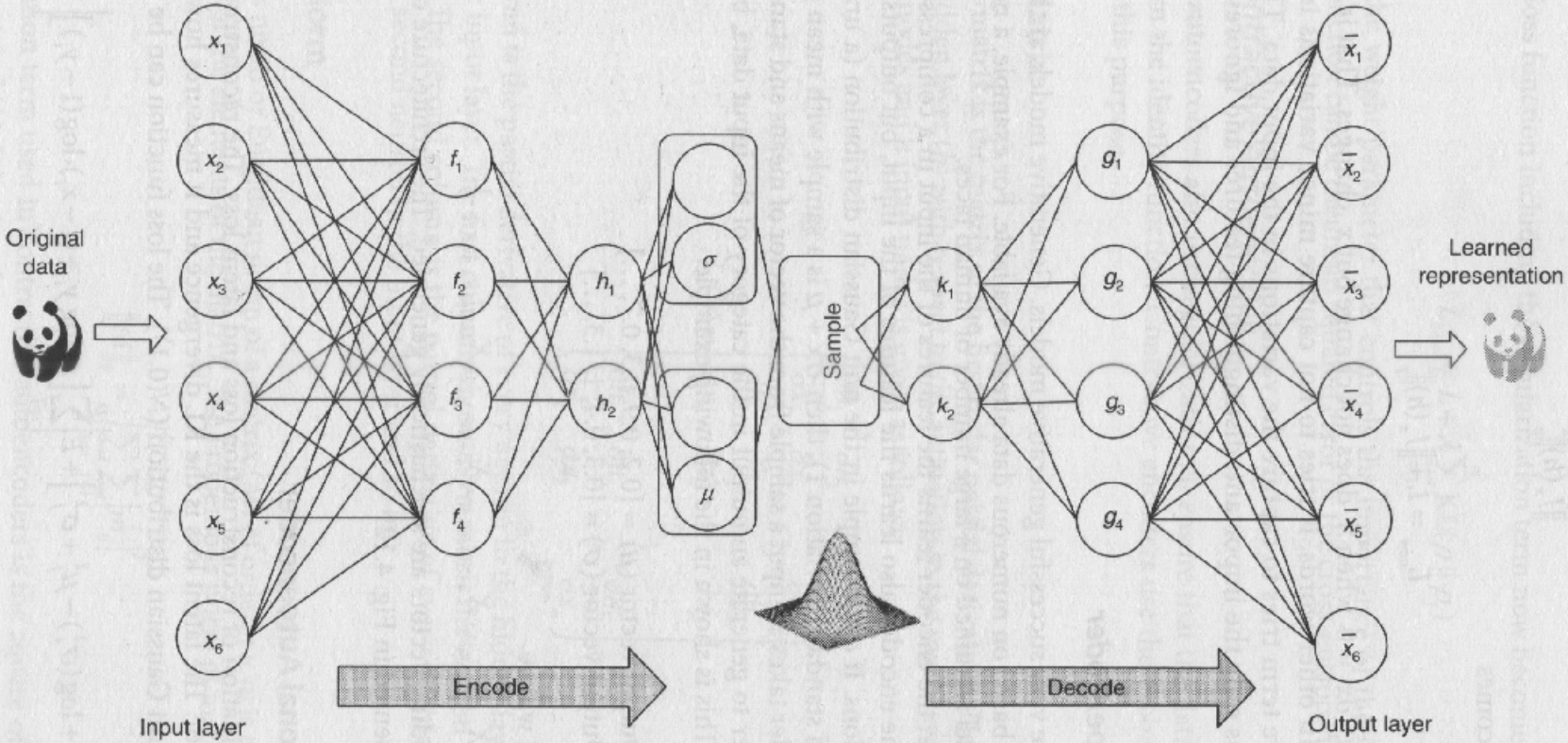

FIGURE 4.12 Variational autoencoders.

4.1 EXAMPLE Real-World

1. **Dimensionality Reduction using Autoencoders in MNIST Dataset**
As a real-world example, let us consider the MNIST dataset. The MNIST dataset is a common dataset used for classification and image recognition tasks. It has images of handwritten digits. It has 60,000 examples in the training set and 10,000 examples in the test set. Each digit is a 28 × 28 image. The input to an autoencoder is 28 × 28 = 784 neurons. The images can be compressed using autoencoders. The compression depends on the number of hidden neurons. If the number of hidden neurons is 50, the image can be reconstructed using just 50 features instead of 784 features. There have been cases where the authors have reconstructed the MNIST dataset using just 25 hidden neurons. The original input and the output of the autoencoder are given in Fig. 4.13.

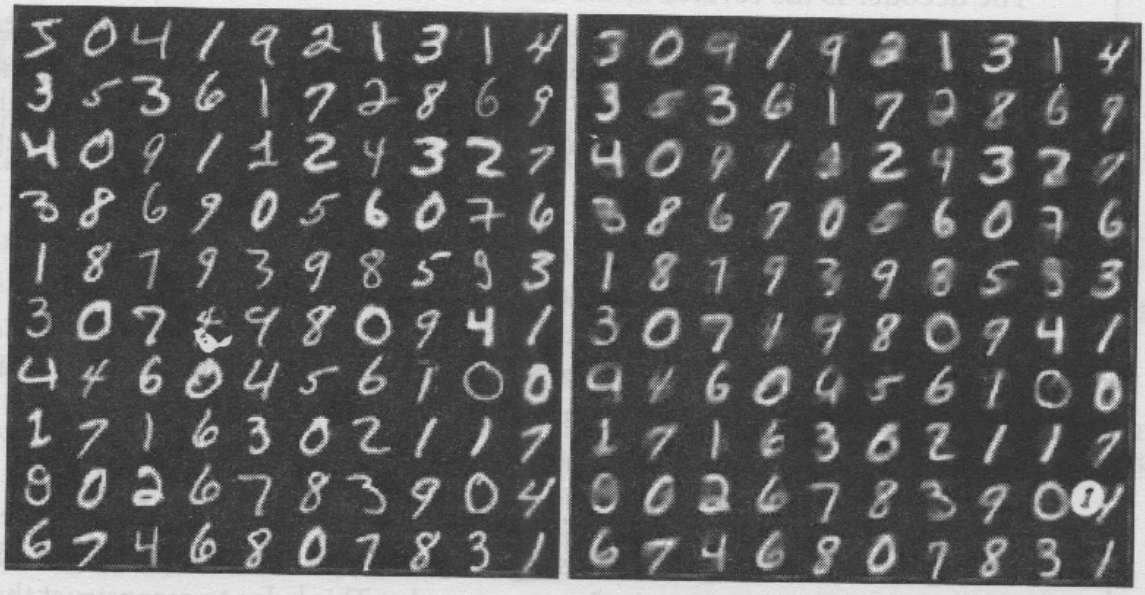

FIGURE 4.13 MNIST dataset reconstructed using autoencoders.

2. **Image Retrieval using Autoencoders**
Content-based image retrieval (CBIR) is a type of image retrieval system. When a query image is given, based on the amount of similarity in the visual contents, images are retrieved. Convolution autoencoder is used to encode the query image into the latent space representation. The images from the search dataset are also encoded to the latent space representation. Subsequently, the encoded query is compared with encoded images from the search dataset using the nearest neighbor algorithm that uses the Euclidean Distance. The images having the closest distance will be displayed.

4.1 EXAMPLE (Continued)

The dataset used is the MNIST dataset and images with similar digits and handwriting style are retrieved.

Architecture

Two encoders E1 and E2 take 28 × 28 grayscale images as input and the architecture consists of four convolutional layers with 3 × 3 kernels, followed by ReLU activations. The outputs of the last convolutional layers are 4 × 4 × 256 tensors that are mapped to a 16-dimensional latent representations z1 and z2 using a single fully-connected layer with ReLU activation. The final layer is a fully-connected layer with linear activation behind E2. Batch normalization is employed for all the convolutional layers and not for the fully-connected layers. The decoder is the reverse of the encoders. It starts with a fully-connected layer to increase the dimensionality of the latent representation to 4096 dimensions and reshape to form a 4 × 4 × 256 tensor. This tensor is mapped back to the input image space using four layers of transposed convolutions. Batch normalization and ReLU activations are employed in all the layers excluding the output layer. The output layer uses hyperbolic tangent activation.

3. **Reconstruction of Video Frames using LSTM Autoencoders**

Sequence prediction problems are problems in which the input is a sequence of data and the output is to predict the next data in the sequence. The data can be text, image or sound. Nitish Srivastava, *et al.* took videos as input data and used LSTM autoencoders to predict the next frame in a video sequence.

Architecture

The architecture comprises of an encoder LSTM and a decoder LSTM. The encoder LSTM reads a frame sequence. This is given as input to a decoder LSTM. The decoder predicts the target sequence in the reverse order. This helps to reconstruct the input. It is also possible to remove the decoder part and use the architecture to predict a new target sequence.

SUMMARY

This chapter introduced autoencoders and its features. The various types of autoencoders were dealt in detail and the differences were highlighted. Readers should have a clarity on the different autoencoder architectures and must be able to use the appropriate autoencoder for their application.

REVIEW QUESTIONS

1. How does the autoencoder differ from the other deep learning architectures?
2. How are the weights updated in stacked autoencoders?
3. How does adding noise help to overcome overfitting in autoencoders?
4. When does an autoencoder learn the identity function?
5. What are generative models? Why are variational autoencoders said to be generative models?

ASSIGNMENT PROBLEMS

1. What is the effect on the regularization term λ as the value of λ changes? Show its effect on using $L2$ regularization when $\lambda = 1$, $\lambda = 10$ and $\lambda = 100$.
2. Write a program using Python to generate images of faces using variational autoencoders.
3. Write a program using Python to compress MNIST images using autoencoders. Find the minimum number of features required for best possible decompression of data.

REFERENCES

1. Domingos, Pedro. 2015. *The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World.* Penguin
2. Dorta, G., Vicente, S., Agapito, L., Campbell, N. D. F., and Simpson, I. 2018. Training VAEs under structured residuals. [arXiv:1804.01050 (stat.ML)]
3. Liou, Cheng-Yuan; Cheng, Wei-Chen; Liou, Jiun-Wei; Liou, Daw-Ran. 2014. Autoencoder for words. *Neurocomputing.* 139: 84–96
4. Makhzani, Alireza and Frey, Brendan. 2013. k-sparse autoencoder. [arXiv:1312.5663]
5. Vincent, Pascal; Larochelle, Hugo; Lajoie, Isabelle; Bengio, Yoshua; Manzagol, Pierre-Antoine. 2010. Stacked denoising autoencoders: learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research.* 11: 3371–3408

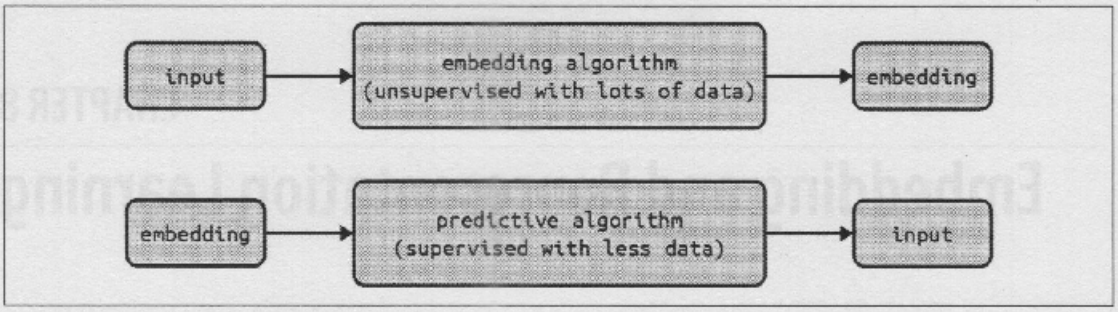

Figure 8-1. Using embeddings to automate feature selection in the face of scarce labeled data

In the next section, we'll introduce *principal component analysis* (PCA), a classic method for dimensionality reduction. In subsequent sections, we'll explore more powerful neural methods for learning compressive embeddings.

Principal Component Analysis

The basic concept behind PCA is to find a set of axes that communicates the most information about our dataset. More specifically, if we have d-dimensional data, we'd like to find a new set of $m < d$ dimensions that conserves as much valuable information from the original dataset as possible. For simplicity, let's choose $d = 2, m = 1$. Assuming that variance corresponds to information, we can perform this transformation through an iterative process. First, we find a unit vector along which the dataset has maximum variance. Because this direction contains the most information, we select this direction as our first axis. Then from the set of vectors orthogonal to this first choice, we pick a new unit vector along which the dataset has maximum variance. This is our second axis.

We continue this process until we have found a total of d new vectors that represent new axes. We project our data onto this new set of axes. We then decide a good value for m and toss out all but the first m axes (the principal components, which store the most information). The result is shown in Figure 8-2.

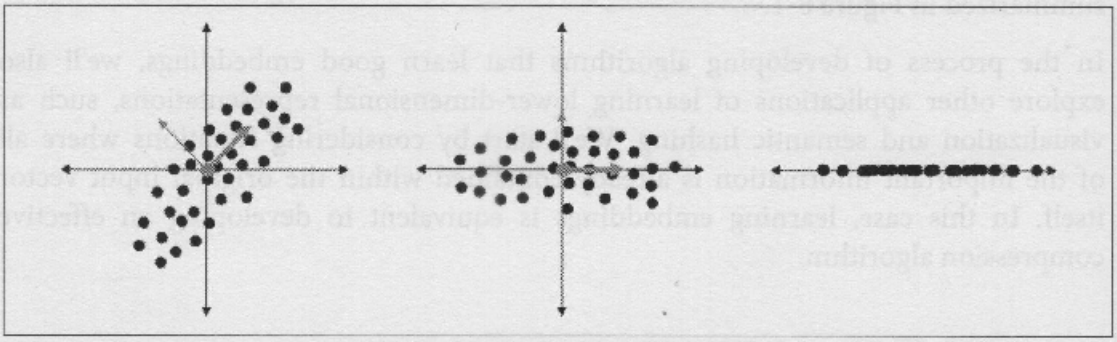

Figure 8-2. An illustration of PCA for dimensionality reduction to capture the dimension with the most information (as proxied by variance)

For the mathematically inclined, we can view this operation as a projection onto the vector space spanned by the top m eigenvectors of the dataset's correlation matrix, which is equivalent to the dataset's covariance matrix when the dataset has been z-score normalized (zero-mean and unit-variance per input dimension). Let us represent the dataset as a matrix $\mathbf{X}$ with dimensions $n \times d$ (i.e., n inputs of d dimensions). We'd like to create an embedding matrix $\mathbf{T}$ with dimensions $n \times m$. We can compute the matrix using the relationship $\mathbf{T} = \mathbf{X}$, where each column of $\mathbf{W}$ corresponds to an eigenvector of the matrix $\frac{1}{n}\mathbf{X}^T\mathbf{X}$. Those with linear algebra background or core data science experience may be seeing a striking parallel between PCA and the singular value decomposition (SVD), which we cover in more depth in "Theory: PCA and SVD" on page 187.

While PCA has been used for decades for dimensionality reduction, it spectacularly fails to capture important relationships that are piecewise linear or nonlinear. Take, for instance, the example illustrated in Figure 8-3.

The example shows data points selected at random from two concentric circles. We hope that PCA will transform this dataset so that we can pick a single new axis that allows us to easily separate the dots. Unfortunately for us, there is no linear direction that contains more information here than another (we have equal variance in all directions). Instead, as human beings, we notice that information is being encoded in a nonlinear way, in terms of how far points are from the origin. With this information in mind, we notice that the polar transformation (expressing points as their distance from the origin, as the new horizontal axis, and their angle bearing from the original x-axis, as the new vertical axis) does just the trick.

Figure 8-3 highlights the shortcomings of an approach like PCA in capturing important relationships in complex datasets. Because most of the datasets we are likely to encounter in the wild (images, text, etc.) are characterized by nonlinear relationships, we must develop a theory that will perform nonlinear dimensionality reduction. Deep learning practitioners have closed this gap using neural models, which we'll cover in the next section.

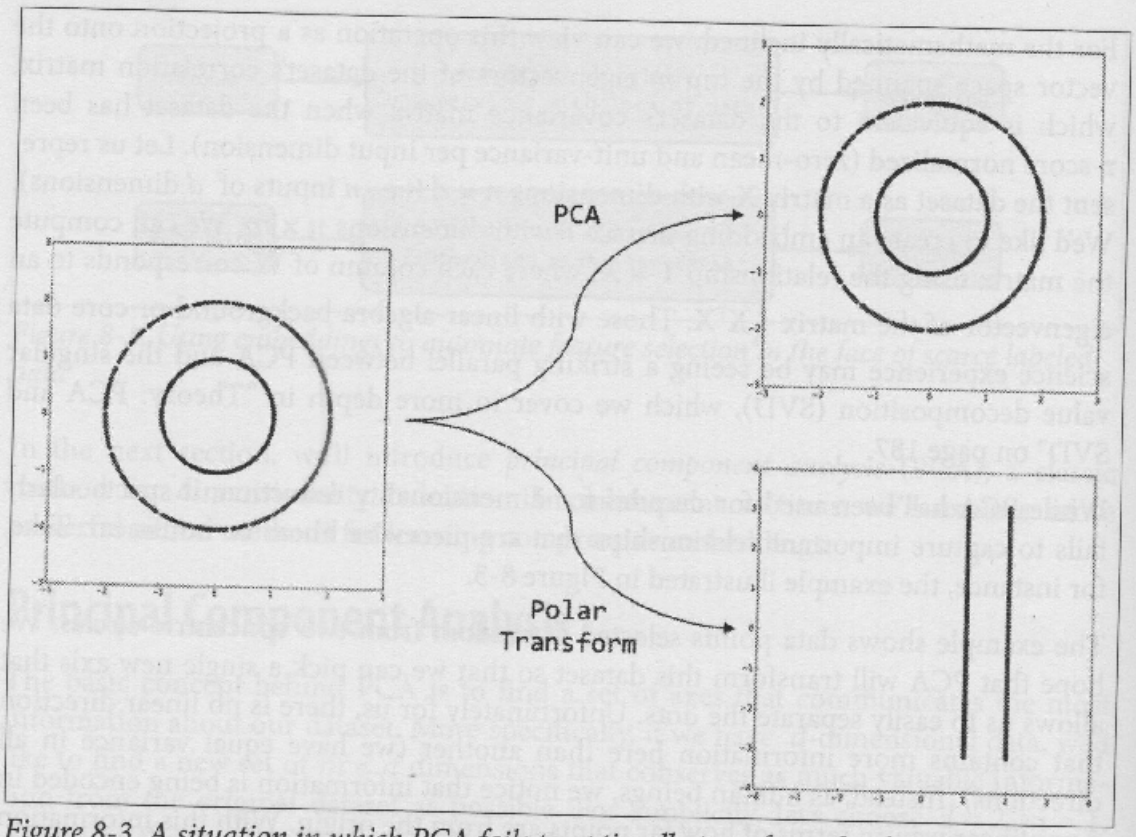

Figure 8-3. A situation in which PCA fails to optimally transform the data for dimensionality reduction

Motivating the Autoencoder Architecture

When we talked about feed-forward networks, we discussed how each layer learned progressively more relevant representations of the input. In fact, in Chapter 7, we took the output of the final convolutional layer and used that as a lower-dimensional representation of the input image. Putting aside the fact that we want to generate these low-dimensional representations in an unsupervised fashion, there are fundamental problems with these approaches in general. Specifically, while the selected layer does contain information from the input, the network has been trained to pay attention to the aspects of the input that are critical to solving the task at hand. As a result, there's a significant amount of information loss with respect to elements of the input that may be important for other classification tasks, but potentially less important than the one immediately at hand.

However, the fundamental intuition here still applies. We define a new network architecture that we call the *autoencoder*. We first take the input and compress it into a low-dimensional vector. This part of the network is called the *encoder* because it is responsible for producing the low-dimensional embedding or *code*. The second part of the network, instead of mapping the embedding to an arbitrary label as we would

Theory: PCA and SVD

Those who have taken any form of applied linear algebra are probably familiar with the SVD, one of the most important matrix factorizations in all of linear algebra. For those uninitiated with the SVD, I will first explain the key concepts behind it (assuming some prior linear algebra knowledge) before jumping into its relationship with PCA.

The SVD states that any matrix M with dimension m by n can be factorized into the form $U\Sigma V^\intercal$ (where $\intercal$ represents the transpose operation) with U of dimension m by m, Σ of dimension m by n, and V of dimension n by n. The matrices U and V are both orthogonal matrices. Orthogonal matrices are square matrices made up of orthonormal column vectors. An important fact about orthogonal matrices is that their transposes are also orthogonal matrices, so $V^\intercal$ in the decomposition is still orthogonal. In addition, the transpose of an orthogonal matrix is its inverse, so we have $U^\intercal U = UU^\intercal = I_m$ and $V^\intercal V = VV^\intercal = I_n$. Σ is a rectangular diagonal matrix with only nonnegative entries along its diagonal, which are termed the *singular values* of the M. Although the SVD itself is not unique, the singular values of a matrix are. If we inspect the product Σx, where x is any random vector, we note that Σ simply acts as a scaling factor for each dimension of x due to Σ being a diagonal matrix (and when rectangular diagonal, either adds dimensions with value 0 when tall or removes dimensions when wide).

Another important and potentially more nonobvious property of orthogonal matrices is that they preserve the length, or L2 norm, of any vector they are multiplied by (this is left as an exercise for you). Orthogonal matrices can change only a vector's orientation, and thus, we characterize the action of an orthogonal matrix upon a vector as a rotation. We call norms such as the L2 norm *rotationally invariant* for this reason. One famous example of an orthogonal matrix you're already familiar with is the identity matrix I—this matrix maps any vector to itself, so we can think of it as a rotation of 0.

To understand the SVD more intuitively, let's imagine the matrix-vector product Mx decomposed as $U\Sigma V^\intercal x$. Based on our discussion so far, we can see that the action of the matrix M upon x can be decomposed into a rotation, followed by a scaling, followed by another rotation.

Now that we have an intuitive understanding of SVD, let's connect it back to the PCA algorithm presented in the main text. Let's assume we have a data matrix X, which is of dimension d by n, where d represents the number of features per datapoint and n represents the number of datapoints. Again, we assume for simplicity that the rows of X have been z-score normalized. The PCA algorithm can be reduced to taking the eigendecomposition of the correlation matrix $\frac{1}{n}XX^\intercal$, which we will

represent as $PDP^\top$. The matrix of eigenvalues D is a diagonal matrix, while the matrix P is a matrix of the corresponding eigenvectors as columns. Generally, the eigendecomposition of a matrix looks like PDP^{-1}, but here the correlation matrix is symmetric so the eigenvectors are orthogonal (we leave this as an exercise for you). Thus, we can represent P as an orthogonal matrix once the eigenvectors are normalized to unit length, at which point the inverse and transpose are equal.

Instead of working with the correlation matrix, let's instead work with the data matrix first and then move to the correlation matrix. We first represent X as $U\Sigma V^\top$. Now, we express the correlation matrix in terms of components of the SVD:

$$\frac{1}{n}XX^\top = \frac{1}{n}U\Sigma V^\top V\Sigma^\top U^\top$$
$$= \frac{1}{n}U\Sigma^2 U^\top$$

We already see the obvious parallels to the correlation matrix's eigendecomposition: U is orthogonal, and the square of the singular value matrix is also a diagonal matrix (this matrix is just the diagonal matrix of the squares of all the singular values). One can show that U's columns, also termed the *left singular vectors*, are the eigenvectors of the correlation matrix. Imagine multiplying $\frac{1}{n}U\Sigma^2U^\top$ by Ue_i, where e_i is a vector of all zeroes except for a one at the index corresponding to any single column (Ue_i is *the ith column of U*). *In practice, it is actually ideal to use the SVD of the data matrix rather than take the eigendecomposition of the correlation matrix due to precision issues when calculating the correlation matrix, which is a product of two potentially* very large matrices.

Summary

In this chapter, we explored various methods in representation learning. We learned about how we can perform effective dimensionality reduction using autoencoders. We also learned about denoising and sparsity, which augment autoencoders with useful properties. After discussing autoencoders, we shifted our attention to representation learning when context of an input is more informative than the input itself. We learned how to generate embeddings for English words using the Skip-Gram model, which will prove useful as we explore deep learning models for understanding language. In the next chapter, we will build on this tangent to analyze language and other sequences using deep learning.