CHAPTER

An Introduction to Neural Networks and Deep Learning

1

Heung-Il Suk
Korea University, Seoul, Republic of Korea

CHAPTER OUTLINE

1.1 INTRODUCTION

A brain or biological neural network is considered as the most well-organized system that processes information from different senses such as sight, hearing, touch, taste, and smell in an efficient and intelligent manner. One of the key mechanisms for

Deep Learning for Medical Image Analysis. DOI: 10.1016/B978-0-12-810408-8.00002-X

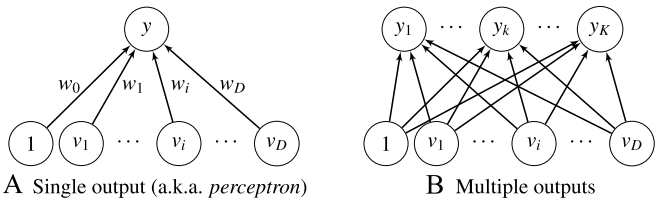

FIGURE 1.1

An architecture of a single-layer neural network.

information processing in a human brain is that the complicated high-level information is processed by means of the collaboration, i.e., connections (called synapses), of a large number of the structurally simple elements (called neurons). In machine learning, artificial neural networks are a family of models that mimic the structural elegance of the neural system and learn patterns inherent in observations.

1.2 FEED-FORWARD NEURAL NETWORKS

This section introduces neural networks that process information in a feed-forward manner. Throughout the chapter, matrices and vectors are denoted as boldface uppercase letters and boldface lowercase letters, respectively, and scalars are denoted as normal italic letters. For a transpose operator, a superscript $^\top$ is used.

1.2.1 PERCEPTRON

The simplest learnable artificial neural model, known as *perceptron* [1], is structured with input visible units $\{v_i\}_{i=1}^D$, trainable connection weights $\{w_i\}_{i=1}^D$, and a bias w_0, and an output unit y as shown in Fig. 1.1A. Since the perceptron model has a single layer of an output unit, not counting the input visible layer, it is also called a single-layer neural network. Given an observation[1] or datum $\mathbf{v} \in \mathbb{R}^D$, the value of the output unit y is obtained from an activation function $f(\cdot)$ by taking the weighted sum of the inputs as follows:

$$y(\mathbf{v}; \Theta) = f\left(\sum_{i=1}^{D} v_i w_i + w_0\right) = f\left(\mathbf{w}^\top \mathbf{v} + w_0\right) \tag{1.1}$$

where $\Theta = \{\mathbf{w}, w_0\}$ denotes a parameter set, $\mathbf{w} = [w_i]_{i=1}^D \in \mathbb{R}^D$ is a connection weight vector, and w_0 is a bias. Let us introduce a pre-activation variable z that is determined by the weighted sum of the inputs, i.e., $z = \mathbf{w}^\top \mathbf{v} + w_0$. As for the activation function $f(\cdot)$, a '*logistic sigmoid*' function, i.e., $\sigma(z) = \frac{1}{1+\exp[-z]}$, is commonly used for a binary classification task.

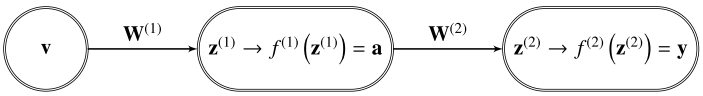

FIGURE 1.2

An architecture of a two-layer neural network in simplified representation.

Regarding a multi-output task, e.g., multi-class classification or multi-output regression, it is straightforward to extend the perceptron model by adding multiple output units $\{y_k\}_{k=1}^{K}$ (Fig. 1.1B), one for each class, with their respective connection weights $\{W_{ki}\}_{i=1,\ldots,D;k=1,\ldots,K}$ as follows:

$$y_k(\mathbf{v};\Theta) = f\left(\sum_{i=1}^{D} v_i W_{ki} + w_{ko}\right) = f\left(\mathbf{w}_k^{\top}\mathbf{v} + w_{k0}\right) \tag{1.2}$$

where $\Theta = \{\mathbf{W} \in \mathbb{R}^{K\times D}\}$, W_{ki} denotes a connection weight from v_i to y_k. As for the activation function, it is common to use a '*softmax*' function $s(z_k) = \frac{\exp(z_k)}{\sum_{l=1}^{K}\exp(z_l)}$ for multi-class classification, where the output values can be interpreted as probability.

1.2.2 MULTI-LAYER NEURAL NETWORK

One of the main limitations of the single-layer neural network comes from its linear separation for a classification task, despite the use of nonlinear activation function. This limitation can be circumvented by introducing a so-called '*hidden*' layer between the input layer and the output layer as shown in Fig. 1.2, where the double circles denote a vectorization of the units in the respective layers for simplification. Note that in Fig. 1.2, there can exist multiple units in each layer and units of the neighboring layers are fully connected to each other, but no connections in the same layer. For a two-layer neural network, which is also known as *multi-layer perceptron*, we can write its composition function as follows:

$$y_k(\mathbf{v};\Theta) = f^{(2)}\left(\sum_{j=1}^{M} W_{kj}^{(2)} f^{(1)}\left(\sum_{i=1}^{D} W_{ji}^{(1)} v_i\right)\right) \tag{1.3}$$

where the superscript denotes a layer index, M is the number of hidden units, and $\Theta = \{\mathbf{W}^{(1)} \in \mathbb{R}^{M\times D}, \mathbf{W}^{(2)} \in \mathbb{R}^{K\times M}\}$. Hereafter, the bias term is omitted for simplicity. It is possible to add a number of hidden layers $(L-1)$ and the corresponding estimation function is defined as

$$y_k = f^{(L)}\left(\sum_{l} W_{kl}^{(L)} f^{(L-1)}\left(\sum_{m} W_{lm} f^{(L-2)}\left(\cdots f^{(1)}\left(\sum_{i} W_{ji}^{(1)} x_i\right)\right)\right)\right). \tag{1.4}$$

While, in theory, it is possible to apply different types of activation functions for different layers or even different units, it is common to apply the same type of an activation function for the hidden layers in the literature. However, it should be a nonlinear function; otherwise, the function will be represented by a single-layer neural network with a weight matrix, equal to the resulting matrix of multiplying weight matrices of hidden layers. In convention, the activation function $f(\cdot)$ is commonly defined with a sigmoidal function such as a '*logistic sigmoid*' function, i.e., $\sigma(z) = \frac{1}{1+\exp[-z]}$, or a '*hyperbolic tangent*' function, i.e., $\tanh(z) = \frac{\exp[z]-\exp[-z]}{\exp[z]+\exp[-z]}$, due to their nonlinear and differentiable characteristics. Their difference lies in the range of output values squashed from real values to the range of $[0, 1]$ for logistic sigmoid function and $[-1, 1]$ for hyperbolic tangent function.

1.2.3 LEARNING IN FEED-FORWARD NEURAL NETWORKS

In terms of network learning, there are two fundamental problems, namely, network architecture learning and network parameters learning. While the network architecture learning still remains an open question,[2] there exists an efficient algorithm for network parameters learning as circumstantiated below.

The problem of learning parameters of a feed-forward neural network can be formulated as error function minimization. Given a training data set $\{\mathbf{x}_n, \mathbf{t}_n\}_{n=1}^N$, where $\mathbf{x}_n \in \mathbb{R}^D$ denotes an observation and $\mathbf{t}_n \in \{0, 1\}^K$ denotes a class indicator vector with one-of-K encoding, i.e., for a class k, only the kth element in a vector $\mathbf{t}_n$ is 1 and all the other elements are 0. For K-class classification, it is common to use a cross-entropy cost function defined as follows:

$$E(\Theta) = -\frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk} \ln y_{nk} \tag{1.5}$$

where t_{kn} denotes the kth element of the target vector $\mathbf{t}_n$ and y_{kn} is the kth element of the prediction vector $\mathbf{y}_n$ for $\mathbf{x}_n$. For a certain parameter set Θ, the prediction vector $\mathbf{y}_n$ from an L-layer neural network can be obtained by Eq. (1.4).

The error function in Eq. (1.5) is highly nonlinear and non-convex. Thus, there is no analytic solution of the parameter set Θ that minimizes Eq. (1.5). Instead, we resort to a gradient descent algorithm by updating the parameters iteratively. To utilize a gradient descent algorithm, one requires a way to compute a gradient $\nabla E(\Theta)$ evaluated at the parameter set Θ.

For a feed-forward neural network, the gradient can be efficiently evaluated by means of error backpropagation [2]. The key idea of backpropagation algorithm is to propagate errors from the output layer back to the input layer by a chain rule. Specifically, in an L-layer neural network, the derivative of an error function E with respect to the parameters for lth layer, i.e., $\mathbf{W}^{(l)}$, can be estimated as follows:

$$\frac{\partial E}{\partial \mathbf{W}^{(l)}} = \frac{\partial E}{\partial \mathbf{a}^{(L)}}\frac{\partial \mathbf{a}^{(L)}}{\partial \mathbf{a}^{(L-1)}}\cdots\frac{\partial \mathbf{a}^{(l+2)}}{\partial \mathbf{a}^{(l+1)}}\frac{\partial \mathbf{a}^{(l+1)}}{\partial \mathbf{a}^{(l)}}\frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l)}}\frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{W}^{(l)}} \tag{1.6}$$

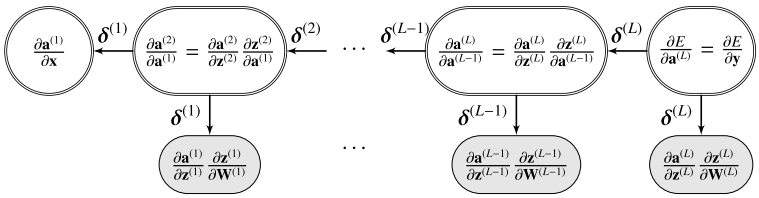

FIGURE 1.3

Graphical representation of a backpropagation algorithm in an L-layer network.

where $\mathbf{z}^{(l)}$ and $\mathbf{a}^{(l)}$ respectively denote the pre-activation vector and the activation vector of the layer l and $\mathbf{a}^{(L)} = \mathbf{y}$. Note that $\frac{\partial E}{\partial \mathbf{a}^{(L)}}$, or equivalently $\frac{\partial E}{\partial \mathbf{y}}$, corresponds to the error computed at the output layer. For the estimation of the gradient of an error function E with respect to the parameter $\mathbf{W}^{(l)}$, it utilizes the error propagated from the output layer through the chains in the form of $\frac{\partial \mathbf{a}^{(k+1)}}{\partial \mathbf{a}^{(k)}}$, $k = l, l+1, \ldots, L-1$, along with $\frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l)}} \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{W}^{(l)}}$. The fraction $\frac{\partial \mathbf{a}^{(k+1)}}{\partial \mathbf{a}^{(k)}}$ can be also computed in a similar way as

$$\frac{\partial \mathbf{a}^{(l+1)}}{\partial \mathbf{a}^{(l)}} = \frac{\partial \mathbf{a}^{(l+1)}}{\partial \mathbf{z}^{(l+1)}} \frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{a}^{(l)}} \tag{1.7}$$

$$= f'\left(\mathbf{z}^{(l)}\right)\left(\mathbf{W}^{(l+1)}\right)^{\top} \tag{1.8}$$

where $f'\left(\mathbf{z}^{(l)}\right)$ denotes a gradient of an activation function $f^{(l)}$ with respect to the pre-activation vector $\mathbf{z}^{(l)}$.

Fig. 1.3 illustrates a graphical representation of estimating the gradients of an error function with respect to parameters in an L-layer neural network based on Eq. (1.6) and Eq. (1.7). Basically, it depicts the error propagation mechanism in a backpropagation algorithm. Specifically, computations are performed at each node by passing error messages, $\boldsymbol{\delta}^{(l)}$, through arrows from tail (source) to head (destination), starting from the rightmost double circle node. When a double circle node of the layer l receives an error message $\boldsymbol{\delta}^{(l+1)}$ from the double circle node of the layer $l+1$, it first updates the error message $\boldsymbol{\delta}^{(l)}$ as

$$\boldsymbol{\delta}^{(l)} = \frac{\partial \mathbf{a}^{(l+1)}}{\partial \mathbf{z}^{(l+1)}} \odot \left\{ \frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{a}^{(l)}} \cdot \boldsymbol{\delta}^{(l+1)} \right\} \tag{1.9}$$

$$= f'\left(\mathbf{z}^{(l)}\right) \odot \left\{ \left(\mathbf{W}^{(l+1)}\right)^{\top} \cdot \boldsymbol{\delta}^{(l+1)} \right\} \tag{1.10}$$

where $\odot$ denotes an element-wise multiplication. After updating the error message, the double circle node of the layer l sends the error message to the double circle node of the layer $l-1$ and also the shaded node directly connected for computing $\frac{\partial E}{\partial \mathbf{W}^{(l)}}$. It should be noted that the double circle nodes can send their message to the neighboring nodes, once they receive an error message from its source node.

Once we obtain the gradient vector of all the layers, i.e., the shaded nodes at the bottom of the graph in Fig. 1.3, the parameter set $\mathbf{W} = \left[\mathbf{W}^{(1)} \dots \mathbf{W}^{(l)} \dots \mathbf{W}^{(L)}\right]$ is updated as follows:

$$\mathbf{W}^{(\tau+1)} = \mathbf{W}^{(\tau)} - \eta \nabla E\left(\mathbf{W}^{(\tau)}\right) \tag{1.11}$$

where $\nabla E\left(\mathbf{W}\right) = \left[\frac{\partial E}{\partial \mathbf{W}^{(1)}} \cdots \frac{\partial E}{\partial \mathbf{W}^{(l)}} \cdots \frac{\partial E}{\partial \mathbf{W}^{(L)}}\right]$ is obtained via backpropagation, η is a learning rate, and τ denotes an iteration index. The update process repeats until convergence or the predefined number of iterations is reached.

As for the parameter update in Eq. (1.11), there are two different approaches depending on the timing of parameter update, namely, batch gradient descent and stochastic gradient descent. The batch gradient descent updates the parameters based on the gradients ∇E evaluated over the whole training samples. Meanwhile, the stochastic gradient descent sequentially updates weight parameters by computing gradient on the basis of one sample at a time. When it comes to large scale learning such as deep learning, it is advocated to apply stochastic gradient descent [3]. As a trade-off between batch gradient and stochastic gradient, a mini-batch gradient descent method, which computes and updates the parameters on the basis of a small set of samples, is commonly used in the literature [4].

1.3 CONVOLUTIONAL NEURAL NETWORKS

In conventional multi-layer neural networks, the inputs are always in vector form. However, for (medical) images, the structural or configural information among neighboring pixels or voxels is another source of information. Hence, vectorization inevitably destroys such structural and configural information in images. A Convolutional Neural Network (CNN) that typically has convolutional layers interspersed with pooling (or sub-sampling) layers and then followed by fully connected layers as in a standard multi-layer neural network (Fig. 1.4) is designed to better utilize such spatial and configuration information by taking 2D or 3D images as input. Unlike the conventional multi-layer neural networks, a CNN exploits extensive weight-sharing to reduce the degrees of freedom of models. A pooling layer helps reduce computation time and gradually build up spatial and configural invariance.

1.3.1 CONVOLUTION AND POOLING LAYER

The role of a convolution layer is to detect local features at different positions in the input feature maps with learnable kernels $k_{ij}^{(l)}$, i.e., connection weights between the feature map i at the layer $l-1$ and the feature map j at the layer l. Specifically, the units of the convolution layer l compute their activations $\mathbf{A}_j^{(l)}$ based only on a spatially contiguous subset of units in the feature maps $\mathbf{A}_i^{(l-1)}$ of the preceding layer

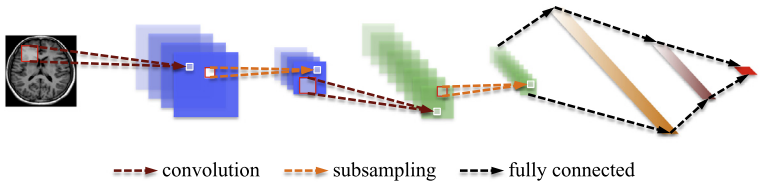

FIGURE 1.4

An architecture of a convolutional neural network.

$l-1$ by convolving the kernels $k_{ij}^{(l)}$ as follows:

$$\mathbf{A}_j^{(l)} = f\left(\sum_{i=1}^{M^{(l-1)}} \mathbf{A}_i^{(l-1)} * k_{ij}^{(l)} + b_j^{(l)}\right) \tag{1.12}$$

where $M^{(l-1)}$ denotes the number of feature maps in the layer $l-1$, $*$ denotes a convolution operator, $b_j^{(l)}$ is a bias parameter, and $f(\cdot)$ is a nonlinear activation function. Due to the local connectivity and weight sharing, we can greatly reduce the number of parameters compared to a fully connected neural network, and thus it is possible to avoid overfitting. Further, when the input image is shifted, the activation of the units in the feature maps are also shifted by the same amount, which allows a CNN to be equivariant to small shifts, as illustrated in Fig. 1.5. In the figure, when the pixel values in the input image are shifted by one-pixel right and one-pixel down, the outputs after convolution are also shifted by one-pixel right and one-pixel down.

A pooling layer follows a convolution layer to downsample the feature maps of the preceding convolution layer. Specifically, each feature map in a pooling layer is linked with a feature map in the convolution layer, and each unit in a feature map of the pooling layer is computed based on a subset of units in its receptive field. Similar to the convolution layer, the receptive field that finds a maximal value among the units in its receptive field is convolved with the convolution map but with a stride of the size of the receptive field so that the contiguous receptive fields are not overlapped commonly. The role of the pooling layer is to progressively reduce the spatial size of the feature maps, and thus reduce the number of parameters and computation involved in the network. Another important function of the pooling layer is for translation invariance over small spatial shifts in the input. In Fig. 1.5, while the bottom leftmost image is a translated version of the top leftmost image by one-pixel right and one-pixel down, their outputs after convolution and pooling operations are the same, especially for the units in green.

1.3.2 COMPUTING GRADIENTS

Assume that a convolution layer is followed by a pooling layer. In such a case, units in a feature map of a convolution layer l are connected to a single unit of the corre-

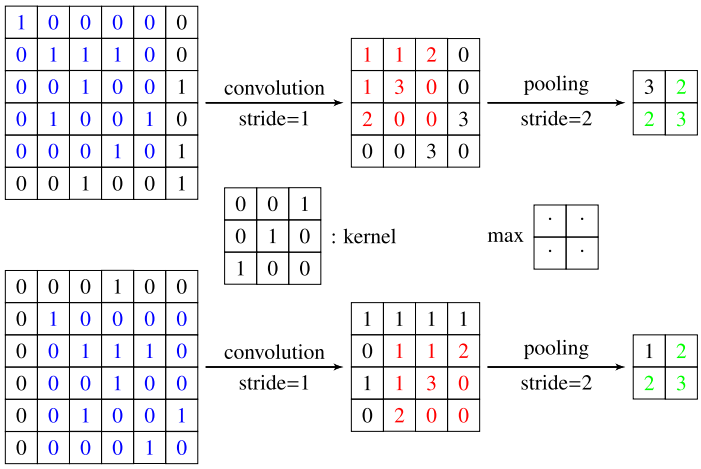

FIGURE 1.5

Illustration of translation invariance in convolution neural network. The bottom leftmost input is a translated version of the upper leftmost input image by one-pixel right and one-pixel down.

sponding feature map in the pooling layer $l + 1$. By up-sampling the feature maps of the pooling layer to recover the reduced size of maps, all we need to do is multiply with the derivative of activation function evaluated at the convolution layer's pre-activations $\mathbf{Z}_j^{(l)}$ as follows:

$$\Delta_j^{(l)} = f'\left(\mathbf{Z}_j^{(l)}\right) \odot \text{up}\left(\Delta_j^{(l+1)}\right) \tag{1.13}$$

where up$(\cdot)$ denotes an up-sampling operation.

For the case when a current layer, whether it is a pooling layer or a convolution layer, is followed by a convolution layer, we must figure out which patch in the current layer's feature map corresponds to a unit in the next layer's feature map. The kernel weights multiplying the connections between the input patch and the output unit are exactly the weights of the convolutional kernel. The gradients for the kernel weights are computed by the chain rule similar to backpropagation. However, since the same weights are now shared across many connections, we need to sum the gradients for a given weight over all the connections using the kernel weights as follows:

$$\frac{\partial E}{\partial k_{ij}^{(l)}} = \sum_{u,v} \Delta_{j;(u,v)}^{(l)} \mathbf{P}_{i;(u,v)}^{(l-1)} \tag{1.14}$$

where $\mathbf{P}^{(l-1)}_{i;(u,v)}$ denotes the patch in the ith feature map of the layer $l-1$, i.e., $\mathbf{A}^{(l-1)}_i$, which was multiplied by $k^{(l)}_{ij}$ during convolution to compute the element at (u, v) in the output feature map $\mathbf{A}^{(l)}_j$.

1.4 DEEP MODELS

Under a mild assumption on the activation function, a two-layer neural network with a finite number of hidden units can approximate any continuous function [5]. Thus, it is regarded as a universal approximator. However, it is also possible to approximate complex functions to the same accuracy using a '*deep*' architecture (i.e., multiple layers) with much fewer number of neurons in total. Hence, we can reduce the number of weight parameters, thus allowing us to train with a relatively small-sized dataset [6]. More importantly, compared to shallow architecture that needs a 'good' feature extractor, mostly designed by engineering skills or domain expert knowledge, deep models are good at discovering good features automatically in a hierarchical manner, i.e., fine-to-abstract [7].

1.4.1 VANISHING GRADIENT PROBLEM

While there have been earlier attempts to deploy deep neural networks, the lack of training samples and the limited computational power prohibited them from successful use for applications. Recently, thanks to the huge amount of data available online and the advances in hardware such as multi-core Central Processing Units (CPUs) and Graphics Processing Units (GPUs), those difficulties have been resolved.

However, so-called '*vanishing gradient*' problem has remained as one of the main challenges in deep learning. That is, for a deep network, the backpropagated error messages $\boldsymbol{\delta}^{(l)}$ in Fig. 1.3 become ineffective due to vanishing gradients after repeated multiplications. In this regard, the gradients tend to get smaller and smaller as they propagate backward through the hidden layers and units in the lower layers (close to the input layer) learn much more slowly than units in the higher layers (close to the output layer).

Regarding the vanishing gradient, Glorot and Bengio [8] pointed out that the use of logistic sigmoid activation function caused the derivatives of units with respect to connection weight parameters to saturate near 0 early in training, and hence substantially slowed down learning speed. As a potential remedy to this problem, they suggested alternative activation functions such as a hyperbolic tangent function and a soft sign function with ways to initialize weights [8]. In the meantime, Hinton and Salakhutdinov devised a greedy layer-wise pretraining technique [9] that made a breakthrough of sorts to the vanishing gradient problem. From a learning perspective, in conventional network training, we randomly initialize the connection weights and then learn the whole weight parameters jointly by means of gradient-descent methods along with backpropagation algorithm for gradients computation in a super-

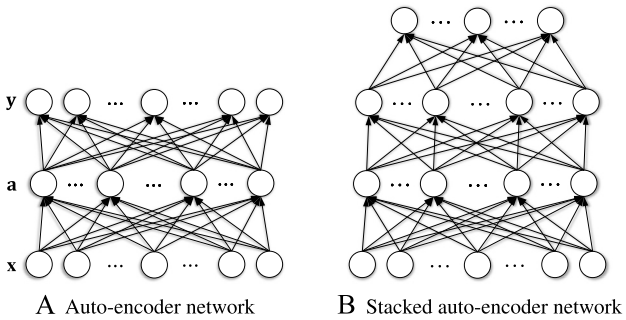

FIGURE 1.6

A graphical illustration of an auto-encoder and a stacked auto-encoder.

vised manner. The idea of pretraining is that instead of directly adjusting the whole weight parameters from random initial values, it is beneficial to train the connection weights of the hidden layers in an unsupervised and layer-wise fashion first, and then fine-tune the connection weights jointly. This pretraining technique will be described with further details below in a stacked auto-encoder.

1.4.2 DEEP NEURAL NETWORKS

Here, we introduce a deep neural network that constructs a deep architecture by taking auto-encoders as building blocks for a hierarchical feature representation.

1.4.2.1 Auto-Encoder

An auto-encoder, also called as auto-associator, is a special type of a two-layer neural network, composed of an input layer, a hidden layer, and an output layer. The input layer is fully connected to the hidden layer, which is further fully connected to the output layer as illustrated in Fig. 1.6A. The aim of an auto-encoder is to learn a latent or compressed representation of an input, by minimizing the reconstruction error between the input and the reconstructed values from the learned representation.

Let D_H and D_I denote, respectively, the number of hidden units and the number of input units in a neural network. For an input $\mathbf{x} \in \mathbb{R}^{D_I}$, an auto-encoder maps it to a latent representation $\mathbf{a} \in \mathbb{R}^{D_H}$ through a linear mapping and then a nonlinear transformation with a nonlinear activation function f as follows:

$$\mathbf{a} = f(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \tag{1.15}$$

where $\mathbf{W}^{(1)} \in \mathbb{R}^{D_H \times D_I}$ is an encoding weight matrix and $\mathbf{b}^{(1)} \in \mathbb{R}^{D_H}$ is a bias vector. The representation $\mathbf{a}$ of the hidden layer is then mapped back to a vector $\mathbf{y} \in \mathbb{R}^{D_I}$, which approximately reconstructs the input vector $\mathbf{x}$ by another mapping as follows:

$$\mathbf{y} = \mathbf{W}^{(2)}\mathbf{a} + \mathbf{b}^{(2)} \approx \mathbf{x} \tag{1.16}$$

where $\mathbf{W}^{(2)} \in \mathbb{R}^{D_I \times D_H}$ and $\mathbf{b}^{(2)} \in \mathbb{R}^{D_I}$ are a decoding weight matrix and a bias vector, respectively. Structurally, the number of input units and the number of output units are determined by the dimension of an input vector. Meanwhile, the number of hidden units can be determined based on the nature of the data. If the number of hidden units is less than the dimension of the input data, then the auto-encoder can be used for dimensionality reduction. However, it is worth noting that to obtain complicated nonlinear relations among input features, it is possible to allow the number of hidden units to be even larger than the input dimension, from which we can still find an interesting structure by imposing a sparsity constraint [10,11].

From a learning perspective, the goal of an auto-encoder is to minimize the reconstruction error between the input $\mathbf{x}$ and the output $\mathbf{y}$ with respect to the parameters. Given a training set $\{\mathbf{X}, \mathbf{Y}\} = \{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^N$, let $E(\mathbf{X}, \mathbf{Y}) = \frac{1}{2}\sum_{i=1}^N \|\mathbf{x}_i - \mathbf{y}_i\|_2^2$ denote a reconstruction error over training samples. To encourage sparseness of the hidden units, it is common to use Kullback–Leibler (KL) divergence to measure the difference between the average activation $\hat{\rho}_j$ of the jth hidden unit over the training samples and the target average activation ρ defined as [12]:

$$\mathrm{KL}(\rho \| \hat{\rho}_j) = \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j}. \tag{1.17}$$

Then our objective function can be written as:

$$E(\mathbf{X}, \mathbf{Z}) + \gamma \sum_{j}^{D_H} \mathrm{KL}(\rho \| \hat{\rho}_j) \tag{1.18}$$

where γ denotes a sparsity control parameter. With the introduction of the KL divergence, the error function is penalized by a large average activation of a hidden unit over the training samples by setting ρ to be small. This penalization drives the activation of many hidden units to be equal or close to zero by making sparse connections between layers.

1.4.2.2 Stacked Auto-Encoder

Note that the outputs of units in the hidden layer become the latent representation of the input vector. However, due to its simple shallow structural characteristic, the representational power of a single-layer auto-encoder is known to be very limited. But when stacking with multiple auto-encoders by taking the activation values of hidden units of an auto-encoder as the input to the following upper auto-encoder, and thus building an SAE, it is possible to greatly improve the representational power [13]. Thanks to the hierarchical structure, one of the most important characteristics of the SAE is to learn or discover highly nonlinear and complicated patterns such as the relations among input features. When an input vector is presented to an SAE, the different layers of the network represent different levels of information. That is, the lower the layer in the network, the simpler the patterns that are learned; the higher the layer, the more complicated or abstract patterns inherent in the input feature vector.

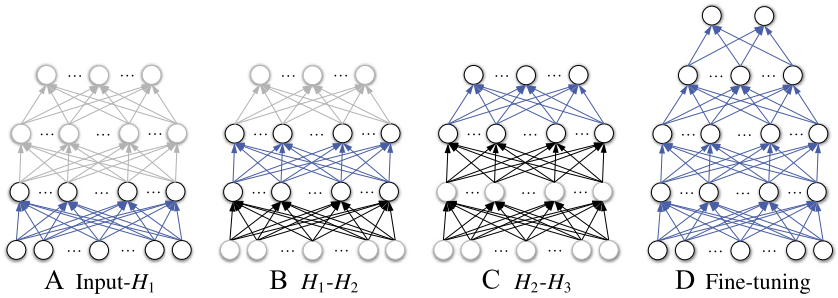

FIGURE 1.7

Greedy layer-wise pretraining and fine-tuning of the whole network. (H_i denotes the ith hidden layer in the network.)

With regard to training parameters of the weight matrices and the biases in an SAE, a straightforward way is to apply backpropagation with the gradient-based optimization technique starting from random initialization by regarding the SAE as a conventional multi-layer neural network. Unfortunately, it is generally known that deep networks trained in that manner perform worse than networks with a shallow architecture, suffering from falling into a poor local optimum [11]. To circumvent this problem, it is good to consider a greedy layer-wise learning [9]. The key idea in a greedy layer-wise learning is to train one layer at a time by maximizing the variational lower bound. That is, we first train the first hidden layer with the training data as input, and then train the second hidden layer with the outputs from the first hidden layer as input, and so on. That is, the representation of the lth hidden layer is used as input for the $(l+1)$th hidden layer. This greedy layer-wise learning is performed as '*pretraining*' (Figs. 1.7A–1.7C). The important feature of pretraining is that it is conducted in an unsupervised manner with a standard backpropagation algorithm [14].

When it comes to a classification problem, we stack another output layer on top of the SAE (Fig. 1.7D) with an appropriate activation function. This top output layer is used to represent the class-label of an input sample. Then by taking the pretrained connection weights as the initial parameters for the hidden units and randomly initializing the connection weights between the top hidden layer and the output layer, it is possible to train the whole parameters jointly in a supervised manner by gradient descent with a backpropagation algorithm. Note that the initialization of the parameters via pretraining helps the supervised optimization, called '*fine-tuning*', reduce the risk of falling into local poor optima [9,11].

1.4.3 DEEP GENERATIVE MODELS

Unlike the deep neural networks, deep generative models draw samples from the trained model, which thus allows checking qualitatively what has been learned by these models, for instance, by visualizing images that the model has learned. Here,

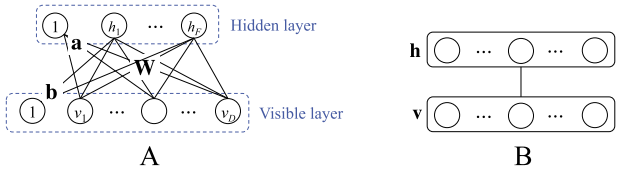

FIGURE 1.8

An architecture of a restricted Boltzmann machine (A) and its simplified illustration (B).

we introduce two deep generative models, namely, Deep Belief Network (DBN) and Deep Boltzmann Machine (DBM), which use restricted Boltzmann machines (RBMs) as basic building blocks for their construction.

1.4.3.1 Restricted Boltzmann Machine

An RBM is a two-layer undirected graphical model with visible and hidden units in each layer (Fig. 1.8). It assumes a symmetric connectivity $\mathbf{W}$ between the visible layer and the hidden layer, but no connections within the layers, and each layer has a bias term, $\mathbf{a}$ and $\mathbf{b}$, respectively. In Fig. 1.8, the units of the visible layer $\mathbf{v} \in \mathbb{R}^D$ correspond to observations while the units of the hidden layer $\mathbf{h} \in \mathbb{R}^F$ models the structures or dependencies over visible units, where D and F denote the numbers of visible and hidden units, respectively. Note that because of the symmetry of the weight matrix $\mathbf{W}$, it is possible to reconstruct the input observations from the hidden representations. Therefore, an RBM is naturally regarded as an auto-encoder [9] and this favorable characteristic is used in RBM parameters learning [9].

In RBM, a joint probability of $(\mathbf{v}, \mathbf{h})$ is given by

$$P(\mathbf{v}, \mathbf{h}; \Theta) = \frac{1}{Z(\Theta)} \exp\left[-E\left(\mathbf{v}, \mathbf{h}; \Theta\right)\right] \tag{1.19}$$

where $\Theta = \left\{\mathbf{W} \in \mathbb{R}^{D \times F}, \mathbf{a} \in \mathbb{R}^D, \mathbf{b} \in \mathbb{R}^F\right\}$, $E\left(\mathbf{v}, \mathbf{h}; \Theta\right)$ is an energy function, and $Z(\Theta)$ is a partition function that can be obtained by summing over all possible pairs of $\mathbf{v}$ and $\mathbf{h}$. For the sake of simplicity, by assuming binary visible and hidden units, which are the commonly studied case, the energy function $E(\mathbf{v}, \mathbf{h}; \Theta)$ is defined as

$$\begin{aligned} E\left(\mathbf{v}, \mathbf{h}; \Theta\right) &= -\mathbf{h}^\top \mathbf{W} \mathbf{v} - \mathbf{a}^\top \mathbf{v} - \mathbf{b}^\top \mathbf{h} \\ &= -\sum_{i=1}^{D} \sum_{j=1}^{F} v_i W_{ij} h_j - \sum_{i=1}^{D} a_i v_i - \sum_{j=1}^{F} b_j h_j. \end{aligned} \tag{1.20}$$

The conditional distribution of the hidden units given the visible units and also the conditional distribution of the visible units given the hidden units are respectively

computed as

$$P(h_j = 1|\mathbf{v};\Theta) = \sigma\left(b_j + \sum_{i=1}^{D} W_{ij} v_i\right), \tag{1.21}$$

$$P(v_i = 1|\mathbf{h};\Theta) = \sigma\left(a_i + \sum_{j=1}^{F} W_{ij} h_j\right) \tag{1.22}$$

where $\sigma(\cdot)$ is a logistic sigmoid function. Due to the unobservable hidden units, the objective function is defined as the marginal distribution of the visible units as

$$P(\mathbf{v};\Theta) = \frac{1}{Z(\Theta)} \sum_{\mathbf{h}} \exp(-E(\mathbf{v},\mathbf{h};\Theta)). \tag{1.23}$$

In medical image analysis, the observations can be real-valued voxel intensities $\mathbf{v} \in \mathbb{R}^D$. For this case, it is common to use a Gaussian RBM [9], in which the energy function is given by

$$E(\mathbf{v},\mathbf{h};\Theta) = \sum_{i=1}^{D} \frac{(v_i - a_i)^2}{2s_i^2} - \sum_{i=1}^{D}\sum_{j=1}^{F} \frac{v_i}{s_i} W_{ij} h_j - \sum_{j=1}^{F} b_j h_j \tag{1.24}$$

where s_i denotes a standard deviation of the ith visible unit and $\Theta = \{\mathbf{W}, \mathbf{a}, \mathbf{b}, \mathbf{s} \in \mathbb{R}^D\}$. This variation leads to the following conditional distribution of visible units given the binary hidden units

$$p(v_i|\mathbf{h};\Theta) = \frac{1}{\sqrt{2\pi}s_i} \exp\left(-\frac{\left(v_i - a_i - \sum_{j=1}^{F} h_j W_{ij}\right)^2}{2s_i^2}\right). \tag{1.25}$$

The RBM parameters are usually trained using a contrastive divergence algorithm [15] that maximizes the log-likelihood of observations. When taking the derivative of the log-likelihood with respect to parameters Θ, we obtain

$$\frac{\partial}{\partial\Theta} \ln P(\mathbf{v};\Theta) = \mathbb{E}_{P\left(\tilde{\mathbf{v}},\tilde{\mathbf{h}}\right)}\left[\frac{\partial}{\partial\Theta} E(\tilde{\mathbf{h}},\tilde{\mathbf{v}})\right] - \mathbb{E}_{P(\mathbf{h}|\mathbf{v})}\left[\frac{\partial}{\partial\Theta} E(\mathbf{h},\mathbf{v})\right]. \tag{1.26}$$

In the equation, the two expected values of $\mathbb{E}_{P(\tilde{\mathbf{v}},\tilde{\mathbf{h}})}\left[\mathbf{v}\mathbf{h}^\top\right]$ and $\mathbb{E}_{P(\mathbf{h}|\mathbf{v})}\left[\mathbf{v}\mathbf{h}^\top\right]$ are approximated by means of a Gibbs sampling technique [16] and their difference provides the direction of updating parameters in (stochastic) gradient descent as follows:

$$\Delta\mathbf{W} = \alpha\left(E_{P\left(\tilde{\mathbf{v}},\tilde{\mathbf{h}}\right)}\left[\mathbf{v}\mathbf{h}^\top\right] - E_{P(\mathbf{h}|\mathbf{v})}\left[\mathbf{v}\mathbf{h}^\top\right]\right), \tag{1.27}$$

$$\Delta\mathbf{a} = \alpha\left(E_{P\left(\tilde{\mathbf{v}},\tilde{\mathbf{h}}\right)}[\mathbf{h}] - E_{P(\mathbf{h}|\mathbf{v})}[\mathbf{h}]\right), \tag{1.28}$$

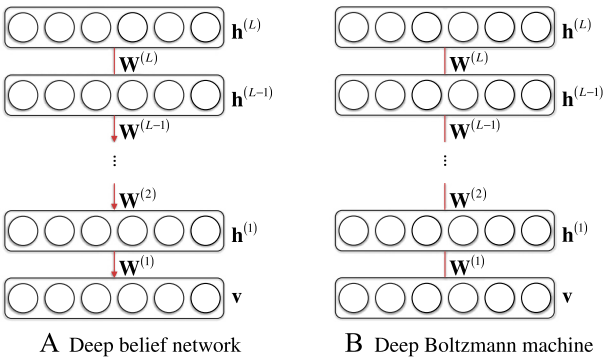

FIGURE 1.9

Graphical illustrations of two different deep generative models with L hidden layers.

$$\Delta \mathbf{b} \quad = \quad \alpha \left(E_{P\left(\tilde{\mathbf{v}}, \tilde{\mathbf{h}}\right)} [\mathbf{v}] - E_{P(\mathbf{h}|\mathbf{v})} [\mathbf{v}] \right) \tag{1.29}$$

where α denotes the learning rate.

1.4.3.2 Deep Belief Network

Since an RBM is a kind of an auto-encoder, it is straightforward to stack multiple RBMs for deep architecture construction, similar to SAE, which results in a single probabilistic model called a Deep Belief Network (DBN). That is, a DBN has one visible layer $\mathbf{v}$ and a series of hidden layers $\mathbf{h}^{(1)}, \ldots, \mathbf{h}^{(L)}$ as shown in Fig. 1.9A. Between any two consecutive layers, let $\left\{\Theta^{(l)}\right\}_{l=1}^{L}$ denote the corresponding RBM parameters. Note that while the top two layers still form an undirected generative model, i.e., RBM, the lower layers form directed generative models. Hence, the joint distribution of the observed units $\mathbf{v}$ and the L hidden layers $\mathbf{h}^{(l)}$ $(l = 1, \ldots, L)$ in DBN is given as

$$P\left(\mathbf{v}, \mathbf{h}^{(1)}, \ldots, \mathbf{h}^{(L)}\right) = \left(\prod_{l=0}^{L-2} P(\mathbf{h}^{(l)}|\mathbf{h}^{(l+1)}) \right) P\left(\mathbf{h}^{(L-1)}, \mathbf{h}^{(L)}\right) \tag{1.30}$$

where $\mathbf{h}^{(0)} = \mathbf{v}$, $P(\mathbf{h}^{(l)}|\mathbf{h}^{(l+1)})$ corresponds to a conditional distribution for the units of the layer l given the units of the layer $l+1$, and $P\left(\mathbf{h}^{(L-1)}, \mathbf{h}^{(L)}\right)$ denotes the joint distribution of the units in the layers $L-1$ and L.

As for the parameter learning, the pretraining scheme described in Section 1.4.2.2 can also be applied as follows:

1. Train the first layer as an RBM with $\mathbf{v} = \mathbf{h}^{(0)}$.
2. Use the first layer to obtain a representation of the input that will be used as observation for the second layer, i.e., either the mean activations of $P(\mathbf{h}^{(1)} = 1|\mathbf{h}^{(0)})$ or samples drawn from $P(\mathbf{h}^{(1)}|\mathbf{h}^{(0)})$.

3. Train the second layer as an RBM, taking the transformed data (samples or mean activations) as training examples (for the visible layer of the RBM).
4. Iterate 2. and 3. for the desired number of layers, each time propagating upward either samples or mean activations.

This greedy layer-wise training of the DBN can be justified as increasing a variational lower bound on the log-likelihood of the data [9]. After the greedy layer-wise procedure is completed, it is possible to perform generative fine-tuning using the wake-sleep algorithm [17]. But in practice, no further procedure is made to train the whole DBN jointly. In order to use a DBN in classification, a trained DBN can also be directly used to initialize a deep neural network with the trained weights and biases. Then the deep neural network can be fine-tuned by means of backpropagation and (stochastic) gradient descent.

1.4.3.3 Deep Boltzmann Machine

A DBM is also structured by stacking multiple RBMs in a hierarchical manner. However, unlike DBN, all the layers in DBM still form an undirected generative model after stacking RBMs as illustrated in Fig. 1.9B. Thus, for the hidden layer l, its probability distribution is conditioned by its two neighboring layers $l+1$ and $l-1$.

Let us consider a three-layer DBM, i.e., $L = 2$ in Fig. 1.9B. The energy of the state $(\mathbf{v}, \mathbf{h}^{(1)}, \mathbf{h}^{(2)})$ in the DBM is given by

$$E(\mathbf{v}, \mathbf{h}^{(1)}, \mathbf{h}^{(2)}; \Theta) = -\mathbf{v}^\top \mathbf{W}^{(1)} \mathbf{h}^{(1)} - (\mathbf{h}^{(1)})^\top \mathbf{W}^{(2)} \mathbf{h}^{(2)} \tag{1.31}$$

where $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$ are symmetric connections of $(\mathbf{v}, \mathbf{h}^{(1)})$ and $(\mathbf{h}^{(1)}, \mathbf{h}^{(2)})$, respectively, and $\Theta = \{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}\}$. Given the values of the units in the neighboring layer(s), the probability of the binary visible or binary hidden units being set to 1 is computed as

$$P\left(h_j^{(1)} = 1 | \mathbf{v}, \mathbf{h}^{(2)}\right) = \sigma\left(\sum_i W_{ij}^{(1)} v_i + \sum_k W_{jk}^{(2)} h_k^{(2)}\right), \tag{1.32}$$

$$P\left(h_k^{(2)} = 1 | \mathbf{h}^{(1)}\right) = \sigma\left(\sum_j W_{jk}^{(2)} h_j^{(1)}\right), \tag{1.33}$$

$$P\left(v_i = 1 | \mathbf{h}^{(1)}\right) = \sigma\left(\sum_j W_{ij}^{(1)} h_j^{(1)}\right). \tag{1.34}$$

Note that in the computation of the conditional probability of the hidden units $\mathbf{h}^{(1)}$, we incorporate both the lower visible layer $\mathbf{v}$ and the upper hidden layer $\mathbf{h}^{(2)}$, and this makes DBM differentiated from DBN and also more robust to noisy observations [18,19].

For a classification task, it is possible to use DBM by replacing an RBM at the top hidden layer with a discriminative RBM [20], which can also be applied for DBN.

That is, the top hidden layer is now connected to both the lower hidden layer and an additional label layer, which indicates the label of the input $\mathbf{v}$. In this way, a DBM can be trained to discover hierarchical and discriminative feature representations by integrating the process of discovering features of inputs with their use in classification [20]. With the inclusion of the additional label layer, the energy of the state $(\mathbf{v}, \mathbf{h}^{(1)}, \mathbf{h}^{(2)}, \mathbf{o})$ in the discriminative DBM is given by

$$E(\mathbf{v}, \mathbf{h}^{(1)}, \mathbf{h}^{(2)}, \mathbf{o}; \Theta) = -\mathbf{v}^\top \mathbf{W}^{(1)} \mathbf{h}^{(1)} - (\mathbf{h}^{(1)})^\top \mathbf{W}^{(2)} \mathbf{h}^{(2)} - (\mathbf{h}^{(2)})^\top \mathbf{U}\mathbf{o} \quad (1.35)$$

where $\mathbf{U}$ and $\mathbf{o} \in \{0, 1\}^C$ denote a connectivity between the top hidden layer and the label layer and a class-label indicator vector, respectively, C is the number of classes, and $\Theta = \{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \mathbf{U}\}$. The probability of an observation $(\mathbf{v}, \mathbf{o})$ is computed by

$$P(\mathbf{v}, \mathbf{o}; \Theta) = \frac{1}{Z(\Theta)} \sum_{\mathbf{h}^{(1)}, \mathbf{h}^{(2)}} \exp\left(-E(\mathbf{v}, \mathbf{h}^{(1)}, \mathbf{h}^{(2)}, \mathbf{o}; \Theta)\right). \quad (1.36)$$

The conditional probability of the top hidden units being set to 1 is given by

$$P\left(h_k^{(2)} = 1 | \mathbf{h}^{(1)}, \mathbf{o}\right) = \sigma\left(\sum_j W_{jk}^{(2)} h_j^{(1)} + \sum_l U_{lk} o_l\right). \quad (1.37)$$

For the label layer, it uses a softmax function

$$P\left(o_l = 1 | \mathbf{h}^{(2)}\right) = \frac{\exp\left[\sum_k U_{lk} h_k^{(2)}\right]}{\sum_{l'} \exp\left[\sum_k U_{l'k} h_k^{(2)}\right]}. \quad (1.38)$$

In this way, the hidden units capture class-predictive information about the input vector.

In order to learn the parameters $\Theta = \{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \mathbf{U}\}$, we maximize the log-likelihood of the observed data $(\mathbf{v}, \mathbf{o})$. The derivative of the log-likelihood of the observed data with respect to the model parameters takes the following simple form:

$$\frac{\partial}{\partial \mathbf{W}^{(l)}} \ln P(\mathbf{v}, \mathbf{o}; \Theta) = \mathbb{E}_{\text{data}}\left[\mathbf{h}^{(l-1)} (\mathbf{h}^{(l)})^\top\right] - \mathbb{E}_{\text{model}}\left[\mathbf{h}^{(l-1)} (\mathbf{h}^{(l)})^\top\right], \quad (1.39)$$

$$\frac{\partial}{\partial \mathbf{U}} \ln P(\mathbf{v}, \mathbf{o}; \Theta) = \mathbb{E}_{\text{data}}\left[\mathbf{h}^{(2)} \mathbf{o}^\top\right] - \mathbb{E}_{\text{model}}\left\langle \mathbf{h}^{(2)} \mathbf{o}^\top\right] \quad (1.40)$$

where $\mathbb{E}_{\text{data}}[\cdot]$ denotes the data-dependent statistics obtained by sampling the model conditioned on the visible units $\mathbf{v}$ ($\equiv \mathbf{h}^{(0)}$) and the label units $\mathbf{o}$ clamped to the observation and the corresponding label, respectively, and $\mathbb{E}_{\text{model}}[\cdot]$ denotes the data-independent statistics obtained by sampling from the model. When the model approximates the data distribution well, it can be reached for the equilibrium of data-dependent and data-independent statistics.

In parameter learning, a gradient-based optimization strategy can be used. In Eq. (1.39) and Eq. (1.40), it is necessary to compute the data-dependent and the data-independent statistics. First, because of the two-way dependency in DBM, it is not tractable for the data-dependent statistics. Fortunately, variational mean-field approximation works well for estimating the data-dependent statistics. For the details of computing the data-dependent statistics, please refer to [21]. Similar to DBN, it can be applied for a greedy layer-wise pretraining strategy to provide a good initial configuration of the parameters, which helps the learning procedure converge much faster than random initialization. However, since the DBM integrates both bottom-up and top-down information, the first and last RBMs in the network need modification by using weights twice as big as in one direction. Then, it is performed for iterative alternation of variational mean-field approximation to estimate the posterior probabilities of hidden units and stochastic approximation to update model parameters.

1.5 TRICKS FOR BETTER LEARNING

Earlier, LeCun et al. presented that by transforming data to have an identity covariance and a zero mean, i.e., data whitening, the network training could converge faster [3,22]. Besides such a simple trick, recent studies have devised other nice tricks to better train deep models.

1.5.1 RECTIFIED LINEAR UNIT (RELU)

The gradient of the logistic sigmoid function and the hyperbolic tangent function vanishes as the value of the respective inputs increases or decreases, which is known as one of the sources to cause the vanishing gradient problem. In this regard, Nair and Hinton suggested to use a Rectified Linear function, $f(a) = \max(0, a)$, for hidden Units (ReLU) and validated its usefulness to improve training time by resolving the vanishing gradient problem. However, mathematically, the ReLU has two problems: (i) it is non-differentiable at $a = 0$, and thus not valid to be used along with a gradient-based method; (ii) it is unbounded on the positive side, and thus can be a potential problem to cause overfitting. Nonetheless, as for the first problem, since it is highly unlikely that the input to any hidden unit will be at exactly $a = 0$ at any time, in practice, the gradient of the ReLU at $a = 0$ is set either 0 or 1. Regarding the unboundedness, the application of a regularization technique is helpful to limit the magnitude of weights, thus circumventing the overfitting issue. Glorot et al. showed that deep neural networks could be trained efficiently by using ReLU and ℓ_2-regularization [23].

1.5.2 DROPOUT

A dropout is a simple but helpful technique to train a deep network with a relatively small dataset. The idea of dropout is to randomly deactivate a fraction of the units,

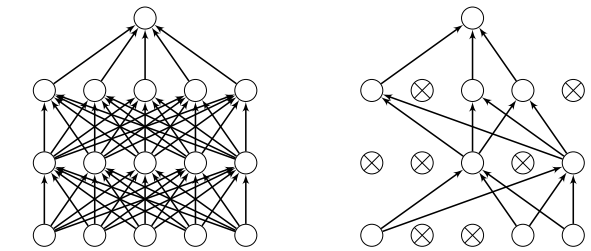

FIGURE 1.10

Comparison between a standard deep neural network and the same network with dropout application. The circles with a cross symbol inside denote deactivated units.

e.g., 50%, in a network on each training iteration (Fig. 1.10B). This helps prevent complex co-adaptations among units, i.e., undesirable dependence on the presence of particular other units [24]. By preventing complex co-adaptations with dropout, it naturally helps avoid overfitting, and thus makes the trained model better generalized. The other noteworthy effect of dropout is to provide a way of combining exponentially many different network architectures efficiently. The random and temporal removal of units in training results in different network architectures, and thus at each iteration, it can be thought to train different networks but their connection weights are shared. In testing, all units in the network should be on, i.e., no dropout, but the weights are halved to maintain the same output range.

1.5.3 BATCH NORMALIZATION

Ioffe and Szegedy [25] observed that the change in the distribution of network activations due to the change in network parameters during training, which they defined as *internal covariate shift*, causes longer training time. To tackle this issue, they introduced a batch normalization technique by performing normalization for each mini-batch and backpropagating the gradients through the normalization parameters (i.e., scale and shift). Specifically, for each unit in a layer l, their value is normalized as follows:

$$\hat{a}_k^{(l)} = \frac{a_k^{(l)} - \mathbb{E}[a_k^{(l)}]}{\sqrt{\text{Var}[a_k^{(l)}]}} \tag{1.41}$$

where k denotes an index of units in the layer l. A pair of learnable parameters $\gamma_k^{(l)}$ and $\beta_k^{(l)}$ are then introduced to scale and shift the normalized values to restore the representation power of the network as follows:

$$y_k^{(l)} = \gamma_k^{(l)} \hat{x}_k^{(l)} + \beta_k^{(l)}. \tag{1.42}$$

The scaled and shifted values are then fed into the following layer as input. In their experiments, it was shown that the batch normalization could greatly shorten the training time and reduce the need for dropout.

1.6 OPEN-SOURCE TOOLS FOR DEEP LEARNING

With the great successes of deep learning methods in different fields, the leading groups in deep learning have publicized their source codes, tools, or even their deep models trained for some applications. Thanks to their great efforts, it is easy for those who are not familiar with deep models to build their own system or methods. Here, we listed the most widely used tools for deep learning along with their features.

- Caffe[3] was originally developed by the Berkeley Vision and Learning Center (BVLC) at the University of California, Berkeley, and has been being updated by the group and community contributors. Of all the open-source tools for deep learning, it has been the most widely used. It's framework is a BSD-licensed C++ library with Python and MATLAB bindings for training and constructing various deep models [26].
- Theano[4] is a Python library, introduced to the machine learning community [27] and originated in 2008 at the Montreal Institute for Learning Algorithms at the University of Montreal. It has nice properties of tight integration with NumPy, transparent use of a GPU, efficient symbolic differentiation, high speed and stability optimizations, dynamic C code generation, and extensive unit-testing and self-verification.
- Torch[5] is a computing framework with wide support for machine learning algorithms. Similarly as other tools, it also allows to use GPUs and to build neural networks and train it with efficient optimization techniques. However, it is dependent on the programming language Lua.
- MatConvNet[6] is a MATLAB library, primarily designed for implementing CNNs, but also possibly to deploy other deep neural networks [28]. It is simple and efficient to use and also provide many pre-trained models proposed in the literature for image classification, segmentation, etc.
- Besides the tools above, Google and Microsoft recently opened their libraries of Tensorflow[7] and CNTK,[8] respectively.

REFERENCES

1. F. Rosenblatt, The perceptron: a probabilistic model for information storage and organization in the brain, Psychol. Rev. (1958) 65–386.
2. D.E. Rumelhart, G.E. Hinton, R.J. Williams, Learning representations by back-propagating errors, Nature 323 (6088) (1986) 533–536.

3. Y. LeCun, L. Bottou, G.B. Orr, K.R. Müller, Efficient BackProp, in: Neural Networks: Tricks of the Trade, Springer, Berlin, Heidelberg, 1998, pp. 9–50.
4. M. Li, T. Zhang, Y. Chen, A.J. Smola, Efficient mini-batch training for stochastic optimization, in: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, 2014, pp. 661–670.
5. K. Hornik, Approximation capabilities of multilayer feedforward networks, Neural Netw. 4 (2) (1991) 251–257.
6. G. Schwarz, Estimating the dimension of a model, Ann. Stat. 6 (2) (1978) 461–464.
7. Y. Bengio, Learning deep architectures for AI, Found. Trends Mach. Learn. 2 (1) (2009) 1–127.
8. X. Glorot, Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, in: Y.W. Teh, D.M. Titterington (Eds.), Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, vol. 9, 2010, pp. 249–256.
9. G.E. Hinton, R.R. Salakhutdinov, Reducing the dimensionality of data with neural networks, Science 313 (5786) (2006) 504–507.
10. H. Lee, C. Ekanadham, A. Ng, Sparse deep belief net model for visual area v2, in: J. Platt, D. Koller, Y. Singer, S. Roweis (Eds.), Advances in Neural Information Processing Systems, vol. 20, MIT Press, Cambridge, MA, 2008, pp. 873–880.
11. H. Larochelle, Y. Bengio, J. Louradour, P. Lamblin, Exploring strategies for training deep neural networks, J. Mach. Learn. Res. 10 (2009) 1–40.
12. H.-C. Shin, M.R. Orton, D.J. Collins, S.J. Doran, M.O. Leach, Stacked autoencoders for unsupervised feature learning and multiple organ detection in a pilot study using 4D patient data, IEEE Trans. Pattern Anal. Mach. Intell. 35 (8) (2013) 1930–1943.
13. Y. Bengio, P. Lamblin, D. Popovici, H. Larochelle, Greedy layer-wise training of deep networks, in: B. Schölkopf, J. Platt, T. Hoffman (Eds.), Advances in Neural Information Processing Systems, vol. 19, MIT Press, Cambridge, MA, 2007, pp. 153–160.
14. C.M. Bishop, Neural Networks for Pattern Recognition, Oxford University Press, Inc., New York, NY, USA, 1995.
15. G.E. Hinton, Training products of experts by minimizing contrastive divergence, Neural Comput. 14 (8) (2000) 1771–1800.
16. C.M. Bishop, Pattern Recognition and Machine Learning (Information Science and Statistics), Springer-Verlag, New York, 2006.
17. G. Hinton, P. Dayan, B. Frey, R. Neal, The wake–sleep algorithm for unsupervised neural networks, Science 268 (5214) (1995) 1158–1161.
18. R. Salakhutdinov, G.E. Hinton, Deep Boltzmann machines, in: Proceedings of the International Conference on Artificial Intelligence and Statistics, 2009, pp. 448–455.
19. N. Srivastava, R. Salakhutdinov, Multimodal learning with deep Boltzmann machines, in: Advances in Neural Information Processing Systems, vol. 25, 2012, pp. 2231–2239.
20. H. Larochelle, Y. Bengio, Classification using discriminative restricted Boltzmann machines, in: Proceedings of the 25th International Conference on Machine Learning, 2008, pp. 536–543.
21. R. Salakhutdinov, G. Hinton, An efficient learning procedure for deep Boltzmann machines, Neural Comput. 24 (8) (2012) 1967–2006.
22. S. Wiesler, H. Ney, A convergence analysis of log-linear training, in: J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, K. Weinberger (Eds.), Advances in Neural Information Processing Systems, vol. 24, 2011, pp. 657–665.

23. X. Glorot, A. Bordes, Y. Bengio, Deep sparse rectifier neural networks, in: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, 2011, pp. 315–323.
24. N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: a simple way to prevent neural networks from overfitting, J. Mach. Learn. Res. 15 (2014) 1929–1958.
25. S. Ioffe, C. Szegedy, Batch normalization: accelerating deep network training by reducing internal covariate shift, in: Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6–11 July 2015, 2015, pp. 448–456.
26. Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, T. Darrell, Caffe: convolutional architecture for fast feature embedding, in: Proceedings of the 22nd ACM International Conference on Multimedia, 2014, pp. 675–678.
27. J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, Y. Bengio, Theano: a CPU and GPU math expression compiler, in: Proceedings of the Python for Scientific Computing Conference, 2010.
28. A. Vedaldi, K. Lenc, MatConvNet – convolutional neural networks for Matlab.

NOTES

1. Each unit in the visible layer takes a scalar value as input. Thus, the observation should be represented in a vector form with elements of raw voxel intensities or features, for instance.
2. It is mostly designed empirically.
3. https://github.com/BVLC/caffe.
4. http://deeplearning.net/software/theano.
5. http://torch.ch.
6. http://www.vlfeat.org/matconvnet.
7. https://www.tensorflow.org.
8. https://github.com/Microsoft/CNTK.