28. Deep and Modular Neural Networks

Ke Chen

In this chapter, we focus on two important areas in neural computation, i. e., deep and modular neural networks, given the fact that both deep and modular neural networks are among the most powerful machine learning and pattern recognition techniques for complex AI problem solving. We begin by providing a general overview of deep and modular neural networks to describe the general motivation behind such neural architectures and fundamental requirements imposed by complex AI problems. Next, we describe background and motivation, methodologies, major building blocks, and the state-of-the-art hybrid learning strategy in context of deep neural architectures. Then, we describe background and motivation, taxonomy, and learning algorithms pertaining to various typical modular neural networks in a wide context. Furthermore, we also examine relevant issues and discuss open problems in deep and modular neural network research areas.

28.1 Overview

The human brain is a generic effective and efficient system that solves complex and difficult problems and generates the trait of intelligence and creation. Neural computation has been inspired by brain-related research in different disciplines, e.g., biology and neuroscience, on various levels ranging from a simple single-neuron to complex neuronal structure and organization [28.1]. Among many discoveries in brain-related sciences, two of the most important properties are modularity and hierarchy of neuronal organization in the human brain.

Neuroscientific research has revealed that the central nervous system (CNS) in the human brain is a distributed, massively parallel, and self-organizing modular system [28.1–3]. The CNS is composed of several regions such as the spinal cord, medulla oblongata, pons, midbrain, diencephalon, cerebellum, and the two cerebral hemispheres. Each such region forms a functional module and all regions are interconnected with other parts of the brain [28.1]. In particular, the cerebral cortex consists of several regions attributed to main perceptual and cognitive tasks, where modularity emerges in two different aspects: i. e., structural and functional modularity. Structural modularity is observable from the fact that there are sparse connections between different neuronal groups but neurons are often densely connected within a neuronal group, while functional modularity is evident from different response patterns produced by neural modules for different perceptual and cognitive tasks. Modularity evidence in the human brain strongly suggests that domain-specific modules are required by specific tasks and different modules can cooperate for high level, complex tasks, which primarily motivates the modular neural network (MNN) development in neural computation (NC) [28.4, 5].

Apart from modularity, the human brain also exhibits a functional and structural hierarchy given the fact

that information processing in the human brain is done in a hierarchical way. Previous studies [28.6, 7] suggested that there are different cortical visual areas that lead to hierarchical information representations to carry out highly complicated visual tasks, e.g., object recognition. In general, hierarchical information processing enables the human brain to accomplish complex perceptual and cognitive tasks in an effective and extremely efficient way, which mainly inspires the study of deep neural networks (DNNs) of multiple layers in NC.

In general, both DNNs and MNNs can be categorized into biologically plausible [28.8] and artificial models [28.9] in NC. The main difference between biologically plausible and artificial models lies their methodologies that a biologically plausible model often takes both structural and functional resemblance to its biological counterpart into account, while an artificial model simply works towards modeling the functionality of a biological system without considering those bio-mimetic factors. Due to the limited space, in this chapter we merely focus on artificial DNNs and MNNs. Readers interested in biologically plausible models are referred to the literature, e.g., [28.4], for useful information.

28.2 Deep Neural Networks

In this section, we overview main deep neural network (DNN) techniques with an emphasis on the latest progress. We first review background and motivation for DNN development. Then we describe major building blocks and relevant learning algorithms for constructing different DNNs. Next, we present a hybrid learning strategy in the context of NC. Finally, we examine relevant issues related to DNNs.

28.2.1 Background and Motivation

The study of NC dates back to the 1940s when McCullod and Pitts modeled a neuron mathematically. After that NC was an active area in AI studies until *Minsky* and *Papert* published their influential book, Perceptron [28.10], in 1969. In the book, they formally proved the limited capacities of the single-layer perceptron and further concluded that there is a slim chance to expand its capacities with its multi-layer version, which significantly slowed down NC research until the back-propagation (BP) algorithm was invented (or reinvented) to solve the learning problem in a multi-layer perceptron (MLP) [28.11].

In theory, the BP algorithm enables one to train an MLP of many hidden layers to form a powerful DNN. Such an attractive technique has aroused tremendous enthusiasm in applying DNNs in different fields [28.9]. Apart from a few exceptions, e.g., [28.12], researchers soon found that an MLP of more than two hidden layers often failed [28.13] due to the well-known fact that MLP learning involves an extremely difficult nonconvex optimization problem, and the gradient-based local search used in the BP algorithm easily gets stuck in an unwanted local minimum. As a result, most researchers gradually gave up deep architectures and devoted their attention to shallow learning architectures of theoretical justification, e.g., the formal but nonconstructive proof that an MLP of single hidden layer may be a universal function approximator [28.14] and a support vector machine (SVM) [28.15], instead. It has been shown that shallow architectures often work well with support of effective feature extraction techniques (but these are often handcrafted). However, recent theoretic justification suggests that learning models of insufficient depth have a fundamental weakness as they cannot efficiently represent the very complicated functions often required in complex AI tasks [28.16, 17].

To solve complex the non-convex optimization problem encountered in DNN learning, *Hinton* and his colleagues made a breakthrough by coming up with a hybrid learning strategy in 2006 [28.18]. The novel learning strategy combines unsupervised and supervised learning paradigms where a layer-wise greedy unsupervised learning is first used to construct an initial DNN with chosen building blocks (such an initial DNN alone can also be used for different purposes, e.g., unsupervised feature learning [28.19]), and supervised learning is then fulfilled based on the pre-trained DNN. Their seminal work led to an emerging machine learning (ML) area, *deep learning*. As a result, different building blocks and learning algorithms have been developed to construct various DNNs. Both theoretical justification and empirical evidence suggest that the hybrid learning strategy [28.18] greatly facilitates learning of DNNs [28.17].

Since 2006, DNNs trained with the hybrid learning strategy have been successfully applied in different and complex AI tasks, such as pattern recogni-

tion [28.20–23], various computer vision tasks [28.24–26], audio classification and speech information processing [28.27–31], information retrieval [28.32–34], natural language processing [28.35–37], and robotics [28.38]. Thus, DNNs have become one of the most promising ML and NC techniques to tackle challenging AI problems [28.39].

28.2.2 Building Blocks and Learning Algorithms

In general, a building block is composed of two parametric models, *encoder* and *decoder*, as illustrated in Fig. 28.1. An encoder transforms a raw input or a low-level representation $\boldsymbol{x}$ into a high-level and abstract representation $\boldsymbol{h}(\boldsymbol{x})$, while a decoder generates an output $\hat{\boldsymbol{x}}$, a reconstructed version of $\boldsymbol{x}$, from $\boldsymbol{h}(\boldsymbol{x})$. The learning building block is a self-supervised learning task that minimizes an elaborate *reconstruction cost function* to find appropriate parameters in encoder and decoder. Thus, the distinction between two building blocks of different types lies in their encoder and decoder mechanisms and reconstruction cost functions (as well as optimization algorithms used for parameter estimation). Below we describe different building blocks and their learning algorithms in terms of the generic architecture shown in Fig. 28.1.

Auto-Encoders

The auto-encoder [28.40] and its variants are simple building blocks used to build an MLP of many layers. It is carried out by an MLP of one hidden layer. As depicted in Fig. 28.2, the input and the hidden layers constitute an encoder to generate a M-dimensional representation $\boldsymbol{h}(\boldsymbol{x}) = (h_1(\boldsymbol{x}), \ldots, h_M(\boldsymbol{x}))^\mathsf{T}$ (hereinafter, we use the notation $\boldsymbol{h}(\boldsymbol{x}) = (h_m(\boldsymbol{x}))_{m=1}^M$ to indicate a vector–element relationship for simplifying the presentation) for a given input $\boldsymbol{x} = (x_n)_{n=1}^N$ in N-dimensional space

$$\boldsymbol{h}(\boldsymbol{x}) = f(W\boldsymbol{x} + \boldsymbol{b}_h) ,$$

where W is a connection weight matrix between the input and the hidden layers, $\boldsymbol{b}_h$ is the bias vector for all hidden neurons, and $f(\cdot)$ is a transfer function, e.g., the sigmoid function [28.9]. Let $f(\boldsymbol{u}) = (f(u_k))_{k=1}^K$ be a collective notation for output of all K neurons in a layer. Accordingly, the hidden and the output layers form a decoder that yields a reconstructed version $\hat{\boldsymbol{x}} = (\hat{x}_n)_{n=1}^N$

$$\hat{\boldsymbol{x}} = f(W^\mathsf{T}\boldsymbol{h}(\boldsymbol{x}) + \boldsymbol{b}_o) ,$$

where W^T is the transpose of the weight matrix W and $\boldsymbol{b}_o$ is the bias vector for all output neurons. Note that the auto-encoder can be viewed as a special case of auto-associator when the same weights are tied to be used in connections between different layers, which will be clearly seen in the learning algorithm later on. Doing so avoids an unwanted solution when an over-complete representation, i. e., $M > N$, is required [28.22].

Further studies [28.41] suggest that the auto-encoder is unlikely to lead to the discovery of a more useful representation than the input despite the fact that a representation should encode much of the information conveyed in the input whenever the auto-encoder produces a good reconstruction of its input. As a result, a variant named the denoising auto-encoder (DAE) was proposed to capture stable structures underlying the distribution of its observed input. The basic idea is as follows: instead of learning the auto-encoder from the intact input, the DAE will be trained to recover the

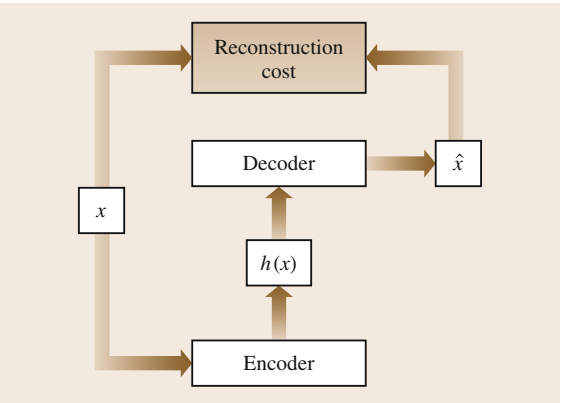

Fig. 28.1 Schematic diagram of a generic building block architecture

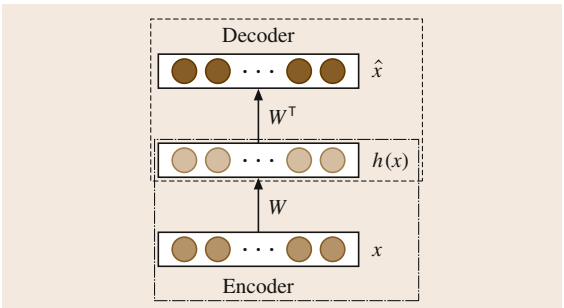

Fig. 28.2 Auto-encoder architecture

original input from its distorted version of partial destruction [28.41]. As is illustrated in Fig. 28.3, the DAE leads to a more useful representation $\boldsymbol{h}(\tilde{\boldsymbol{x}})$ by restoring the corrupted input $\tilde{\boldsymbol{x}}$ to a reconstructed version $\hat{\boldsymbol{x}}$ as close to the clean input $\boldsymbol{x}$ as possible. Thus, the encoder yields a representation as

$$\boldsymbol{h}(\tilde{\boldsymbol{x}}) = f(W\tilde{\boldsymbol{x}} + \boldsymbol{b}_h) ,$$

and the decoder produces a restored version $\hat{\boldsymbol{x}}$ via the representation $h(\tilde{\boldsymbol{x}})$

$$\hat{\boldsymbol{x}} = f(W^T\boldsymbol{h}(\tilde{\boldsymbol{x}}) + \boldsymbol{b}_o) .$$

To produce a corrupted input, we need to distort a clean input by corrupting it with appropriate noise. Depending on the attribute nature of input, there are three kinds of noise used in the corruption process: i. e., the isotropic *Gaussian noise*, $N(0, \sigma^2 I)$, *masking noise* (by setting some randomly chosen elements of $\boldsymbol{x}$ to zero) and *salt-and-pepper* noise (by flipping some randomly chosen elements' values of $\boldsymbol{x}$ to the maximum or the minimum of a given range). Normally, Gaussian noise is used for input of real or continuous values, while and masking and salt-and-pepper noise is applied to input of discrete values, e.g., pixel intensities of gray images. It is worth stating that the variance σ^2 in Gaussian noise and the number of randomly chosen elements in masking and salt-and-pepper noise are hyper-parameters that affect DAE learning. By corrupting a clean input with the chosen noise, we achieve an example, $(\tilde{\boldsymbol{x}}, \boldsymbol{x})$, for self-supervised learning.

Given a training set of T examples $\{(\boldsymbol{x}_t, \boldsymbol{x}_t)\}_{t=1}^T$ (auto-encoder) or $\{(\tilde{\boldsymbol{x}}_t, \boldsymbol{x}_t)\}_{t=1}^T$ (DAE) two reconstruction cost functions are commonly used for learning auto-encoders as follows

$$L(W, \boldsymbol{b}_h, \boldsymbol{b}_o) = \frac{1}{2T}\sum_{t=1}^{T}\sum_{n=1}^{N}(x_{tn} - \hat{x}_{tn})^2 , \qquad (28.1a)$$

$$L(W, \boldsymbol{b}_h, \boldsymbol{b}_o) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{n=1}^{N}(x_{tn}\log\hat{x}_{tn} + (1 - x_{tn})\log(1 - \hat{x}_{tn})) . \qquad (28.1b)$$

The cost function in (28.1a) is used for input of real or discrete values, while the cost function in (28.1b) is employed especially for input of binary values.

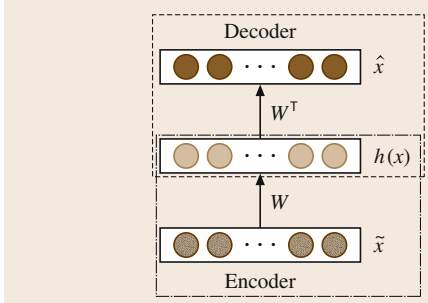

Fig. 28.3 Denoising auto-encoder architecture

To minimize reconstruction functions in (28.1), application of the stochastic gradient descent algorithm [28.12] leads to a generic learning algorithm for training the auto-encoder and its variant, summarized as follows:

Auto-Encoder Learning Algorithm. Given a training set of T examples, $\{(z_t, \boldsymbol{x}_t)\}_{t=1}^T$ where $z_t = \boldsymbol{x}_t$ for the auto-encoder or $z_t = \tilde{\boldsymbol{x}}_t$ for the DAE, and a transfer function, $f(\cdot)$, randomly initialize all parameters, $W, \boldsymbol{b}_h$ and $\boldsymbol{b}_o$, in auto-encoders and pre-set a learning rate ϵ. Furthermore, the training set is randomly divided into several batches of T_B examples, $\{(z_t, \boldsymbol{x}_t)\}_{t=1}^{T_B}$, and then parameters are updated based on each batch:

- Forward computation
  For the input z_t $(t = 1, \cdots, T_B)$, output of the hidden layer is

  $$\boldsymbol{h}(z_t) = f(\boldsymbol{u}_h(z_t)) , \quad \boldsymbol{u}_h(z_t) = Wz_t + \boldsymbol{b}_h .$$

  And output of the output layer is

  $$\hat{\boldsymbol{x}}_t = f(\boldsymbol{u}_o(z_t)) , \quad \boldsymbol{u}_o(z_t) = W^T\boldsymbol{h}(z_t) + \boldsymbol{b}_o .$$

- Backward gradient computation
  For the cost function in (28.1a),

  $$\frac{\partial L(W, \boldsymbol{b}_h, \boldsymbol{b}_o)}{\partial \boldsymbol{u}_o(z_t)} = \left((\hat{x}_{tn} - x_{tn})f'(u_{o,n}(z_t))\right)_{n=1}^N ,$$

  where $f'(\cdot)$ is the first-order derivative function of $f(\cdot)$. For the cost function in (28.1b),

  $$\frac{\partial L(W, \boldsymbol{b}_h, \boldsymbol{b}_o)}{\partial \boldsymbol{u}_o(z_t)} = \hat{\boldsymbol{x}}_t - \boldsymbol{x}_t .$$

  Then, the gradient with respect to $\boldsymbol{h}(z_t)$ is

  $$\frac{\partial L(W, \boldsymbol{b}_h, \boldsymbol{b}_o)}{\partial \boldsymbol{h}(z_t)} = W\frac{\partial L(W, \boldsymbol{b}_h, \boldsymbol{b}_o)}{\partial \boldsymbol{u}_o(z_t)} .$$

Applying the chain rule achieves the gradient with respect to $\boldsymbol{u}_h(z_t)$ as

$$\frac{\partial L(W,\boldsymbol{b}_h,\boldsymbol{b}_o)}{\partial \boldsymbol{u}_h(z_t)} = \left(f'(u_{h,m}(z_t))\frac{\partial L(W,\boldsymbol{b}_h,\boldsymbol{b}_o)}{\partial h_m(z_t)}\right)_{m=1}^{M} .$$

Gradients with respect to biases are

$$\frac{\partial L(W,\boldsymbol{b}_h,\boldsymbol{b}_o)}{\partial \boldsymbol{b}_o} = \frac{\partial L(W,\boldsymbol{b}_h,\boldsymbol{b}_o)}{\partial \boldsymbol{u}_o(z_t)} ,$$

and

$$\frac{\partial L(W,\boldsymbol{b}_h,\boldsymbol{b}_o)}{\partial \boldsymbol{b}_h} = \frac{\partial L(W,\boldsymbol{b}_h,\boldsymbol{b}_o)}{\partial \boldsymbol{u}_h(z_t)} .$$

- Parameter update
  Applying the gradient descent method and tied weights leads to update rules

$$W \leftarrow W - \frac{\epsilon}{T_B}\sum_{t=1}^{T_B}\left[\frac{\partial L(W,\boldsymbol{b}_h,\boldsymbol{b}_o)}{\partial \boldsymbol{u}_h(z_t)}[z_t]^{\mathsf{T}} + \boldsymbol{h}(z_t)\left(\frac{\partial L(W,\boldsymbol{b}_h,\boldsymbol{b}_o)}{\partial \boldsymbol{u}_o(z_t)}\right)^{\mathsf{T}}\right] ,$$

$$\boldsymbol{b}_o \leftarrow \boldsymbol{b}_o - \frac{\epsilon}{T_B}\sum_{t=1}^{T_B}\frac{\partial L(W,\boldsymbol{b}_h,\boldsymbol{b}_o)}{\partial \boldsymbol{u}_o(z_t)} ,$$

and

$$\boldsymbol{b}_h \leftarrow \boldsymbol{b}_h - \frac{\epsilon}{T_B}\sum_{t=1}^{T_B}\frac{\partial L(W,\boldsymbol{b}_h,\boldsymbol{b}_o)}{\partial \boldsymbol{u}_h(z_t)} .$$

The above three steps repeat for all batches, which leads to a training epoch. The learning algorithm runs iteratively until a termination condition is met (typically based on a cross-validation procedure [28.12]).

The Restricted Boltzmann Machine

Strictly speaking, the restricted Boltzmann machine (RBM) [28.42] is an energy-based generative model, a simplified version of the generic Boltzmann machine. As illustrated in Fig. 28.4, an RBM can be viewed as a probabilistic NN of two layers, i. e., *visible* and *hidden* layers, with bi-directional connections. Unlike the Boltzmann machine, there are no lateral connections among neurons in the same layer in an RBM. With the bottom-up connections from the visible to the hidden layer, RBM forms an encoder that yields a probabilistic representation $\boldsymbol{h} = (h_m)_{m=1}^{M}$ for input data $\boldsymbol{v} = (v_n)_{n=1}^{N}$

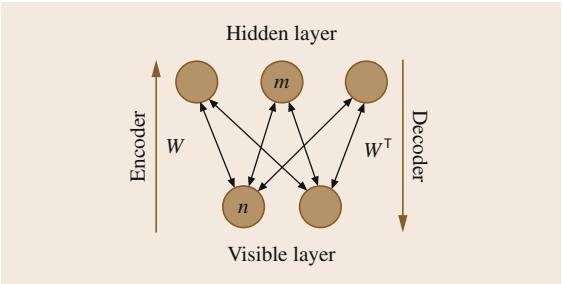

Fig. 28.4 Restricted Boltzmann machine (RBM) architecture

$$P(\boldsymbol{h}|\boldsymbol{v}) = \prod_{m=1}^{M} P(h_m|\boldsymbol{v}) ,$$

$$P(h_m|\boldsymbol{v}) = \phi\left(\sum_{n=1}^{N} W_{mn}v_n + b_{h,m}\right) , \qquad (28.2)$$

where W_{mn} is the connection weight between the visible neuron n and the hidden neuron m, and $b_{h,m}$ is the bias of the hidden neuron m. $\phi(u) = \frac{1}{1+e^{-u}}$ is the sigmoid transfer function. As h_m is assumed to take a binary value, i. e., $h_m \in \{0,1\}$, $P(h_m|\boldsymbol{v})$ is interpreted as the probability of $h_m = 1$. Accordingly, RBM performs a probabilistic decoder via the top-down connections from the hidden to the visible layer to reconstruct an input with the probability

$$P(\boldsymbol{v}|\boldsymbol{h}) = \prod_{m=1}^{M} P(v_n|\boldsymbol{h}) ,$$

$$P(v_n|\boldsymbol{h}) = \phi\left(\sum_{m=1}^{N} W_{nm}h_m + b_{v,n}\right) , \qquad (28.3)$$

where W_{nm} is the connection weight between the hidden neuron m and the visible neuron n, and $b_{v,n}$ is the bias of visible neuron n. Like connection weights in auto-encoders, bi-directional connection weights are tied, i. e., $W_{mn} = W_{nm}$, as shown in Fig. 28.4. By learning a parametric model of the data distribution $P(\boldsymbol{v})$ derived from the joint probability $P(\boldsymbol{v},\boldsymbol{h})$ for a given data set, RBM yields a probabilistic representation that tends to reconstruct any data subject to $P(\boldsymbol{v})$.

The joint probability $P(\boldsymbol{v}, \boldsymbol{h})$ is defined based on the following energy function for $v_n \in \{0, 1\}$

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\sum_{m=1}^{M}\sum_{n=1}^{N} W_{mn} h_m v_n - \sum_{m=1}^{M} h_m b_{h,m} - \sum_{n=1}^{N} v_n b_{v,m} \,. \tag{28.4}$$

As a result, the joint probability is subject to the Boltzmann distribution

$$P(\boldsymbol{v}, \boldsymbol{h}) = \frac{\mathrm{e}^{-E(\boldsymbol{v},\boldsymbol{h})}}{\sum_{\boldsymbol{v}}\sum_{\boldsymbol{h}} \mathrm{e}^{-E(\boldsymbol{v},\boldsymbol{h})}} \,. \tag{28.5}$$

Thus, we achieve the data probability by marginalizing the joint probability as follows

$$P(\boldsymbol{v}) = \sum_{\boldsymbol{h}} P(\boldsymbol{v}, \boldsymbol{h}) = \sum_{\boldsymbol{h}} P(\boldsymbol{v}|\boldsymbol{h}) P(\boldsymbol{h}) \,.$$

In order to achieve the most likely reconstruction, we need to maximize the log-likelihood of $P(\boldsymbol{v})$. Therefore, the reconstruction cost function of an RBM is its negative log-likelihood function

$$L(W, \boldsymbol{b}_h, \boldsymbol{b}_v) = -\log P(\boldsymbol{v}) = -\log \sum_{\boldsymbol{h}} P(\boldsymbol{v}|\boldsymbol{h}) P(\boldsymbol{h}) \,. \tag{28.6}$$

From (28.5) and (28.6), it is observed that the direct use of a gradient descent method for optimal parameters often leads to intractable computation due to the fact that the exponential number of possible hidden-layer configurations needs to be summed over in (28.5) and then used in (28.6). Fortunately, an approximation algorithm named *contrastive divergence (CD)* has been proposed to solve this problem [28.42]. The key ideas behind the CD algorithm are (i) using Gibbs sampling based on the conditional distributions in (28.2) and (28.3), and (ii) running only a few iterations of Gibbs sampling by treating the data $\boldsymbol{x}$ input to an RBM as the initial state, i. e., $\boldsymbol{v}^0 = \boldsymbol{x}$, of the Markov chain at the visible layer. Many studies have suggested that only the use of one iteration of the Markov chain in the CD algorithm works well for building up a deep belief network (DBN) in practice [28.17, 18, 22], and hence the algorithm is dubbed *CD-1* in this situation, a special case of the *CD-k* algorithm that executes k iterations of the Markov chain in the Gibbs sampling process.

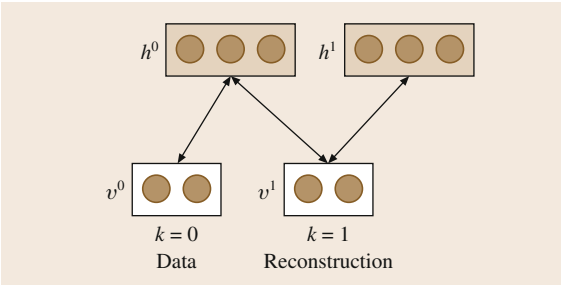

Fig. 28.5 Gibbs sampling process in the *CD-1* algorithm

Figure 28.5 illustrates a Gibbs sampling process used in the *CD-1* algorithm as follows

i) Estimating probabilities $P(h_m^0|\boldsymbol{v}^0)$, for $m = 1, \cdots, M$, with the encoder defined in (28.2) and then forming a realization of $\boldsymbol{h}^0$ by sampling with these probabilities.
ii) Applying the decoder defined in (28.3) to estimate probabilities $P(v_n^1|\boldsymbol{h}^0)$, for $n = 1, \cdots, N$, and then producing a reconstruction of $\boldsymbol{v}^1$ via sampling.
iii) With the reconstruction, estimating probabilities $P(h_m^1|\boldsymbol{v}^1)$, for $m = 1, \cdots, M$, with the encoder.

With the above Gibbing sampling procedure, the *CD-1* algorithm is summarized as follows:

Algorithm 28.1 RBM CD-1 Learning Algorithm

Given a training set of T instances, $\{\boldsymbol{x}_t\}_{t=1}^{T}$, randomly initialize all parameters, $W, \boldsymbol{b}_h$ and $\boldsymbol{b}_v$, in an RBM and pre-set a learning rate ϵ:

- Positive phase
  - Present an instance to the visible layer, i. e., $\boldsymbol{v}^0 = \boldsymbol{x}_t$.
  - Estimate probabilities with the encoder: $\hat{\boldsymbol{P}}(\boldsymbol{h}^0|\boldsymbol{v}^0) = (P(h_m^0|\boldsymbol{v}^0))_{m=1}^{M}$ by using (28.2).
- Negative phase
  - Form a realization of $\boldsymbol{h}^0$ by sampling with probabilities $\hat{\boldsymbol{P}}(\boldsymbol{h}^0|\boldsymbol{v}^0)$.
  - With the realization of $\boldsymbol{h}^0$, apply the decoder to estimate probabilities: $\hat{\boldsymbol{P}}(\boldsymbol{v}^1|\boldsymbol{h}^0) = (P(v_n^0|\boldsymbol{h}^0))_{n=1}^{N}$ by using (28.3), and then produce a reconstruction $\boldsymbol{v}^1$ via sampling based on $\hat{\boldsymbol{P}}(\boldsymbol{v}^1|\boldsymbol{h}^0)$.
  - With the encoder and the reconstruction, estimate probabilities: $\hat{\boldsymbol{P}}(\boldsymbol{h}^1|\boldsymbol{v}^1) = (P(h_m^1|\boldsymbol{v}^1))_{m=1}^{M}$ by using (28.2).
- Parameter update
  Based on Gibbs sampling results in the positive and

the negative phases, parameters are updated as follows:

$$W \leftarrow W + \epsilon \left(\hat{\boldsymbol{P}}(\boldsymbol{h}^0|\boldsymbol{v}^0)(\boldsymbol{v}^0)^\mathsf{T} - \hat{\boldsymbol{P}}(\boldsymbol{h}^1|\boldsymbol{v}^1)(\boldsymbol{v}^1)^\mathsf{T} \right),$$

$$\boldsymbol{b}_h \leftarrow \boldsymbol{b}_h + \epsilon \left(\hat{\boldsymbol{P}}(\boldsymbol{h}^0|\boldsymbol{v}^0) - \hat{\boldsymbol{P}}(\boldsymbol{h}^1|\boldsymbol{v}^1) \right),$$

and

$$\boldsymbol{b}_v \leftarrow \boldsymbol{b}_v + \epsilon \left(\boldsymbol{v}^0 - \boldsymbol{v}^1 \right).$$

The above three steps repeat for all instances in the given training set, which leads to a training epoch. The learning algorithm runs iteratively until it converges.

Predictive Sparse Decomposition

Predictive sparse decomposition (PSD) [28.43] is a building block obtained by combining sparse coding [28.44] and auto-encoder ideas. In a PSD building block, the encoder is specified by

$$\boldsymbol{h}(\boldsymbol{x}_t) = G \tanh(W_E \boldsymbol{x}_t + \boldsymbol{b}_h), \tag{28.7}$$

where W_E is the $M \times N$ connection matrix between input and hidden neurons in the encoder and $G = \text{diag}(g_{mm})$ is an $M \times M$ learnable diagonal gain matrix for an M-dimensional representation of an N-dimensional input, $\boldsymbol{x}_t, \boldsymbol{b}_h$ are biases of hidden neurons, and $\tanh(\cdot)$ is the hyperbolic tangent transfer function [28.9]. Accordingly, the decoder is implemented by a linear mapping used in the sparse coding [28.44]

$$\hat{\boldsymbol{x}}_t = W_D \boldsymbol{h}(\boldsymbol{x}_t), \tag{28.8}$$

where W_D is an $N \times M$ connection matrix between hidden and output neurons in the decoder, and each column of W_D always needs to be normalized to a unit vector to avoid trivial solutions [28.44].

Given a training set of T instances, $\{\boldsymbol{x}_t\}_{t=1}^T$, the PSD cost function is defined as

$$\begin{aligned} &L_{\text{PSD}}\left(G, W_E, W_D, \boldsymbol{b}_h; \boldsymbol{h}^*(\boldsymbol{x}_t)\right) \\ &= \sum_{t=1}^{\mathsf{T}} \|W_D \boldsymbol{h}^*(\boldsymbol{x}_t) - \boldsymbol{x}\|_2^2 + \alpha \|\boldsymbol{h}^*(\boldsymbol{x}_t)\|_1 \\ &\quad + \beta \|\boldsymbol{h}^*(\boldsymbol{x}_t) - \boldsymbol{h}(\boldsymbol{x}_t)\|_2^2, \end{aligned} \tag{28.9}$$

where $\boldsymbol{h}^*(\boldsymbol{x}_t)$ is the optimal sparse hidden representation of $\boldsymbol{x}_t$ while $\boldsymbol{h}(\boldsymbol{x}_t)$ is the output of the encoder in (28.7) based on the current parameter values. In (28.9), α and β are two hyper-parameters to control regularization strengths, and $\|\cdot\|_1$ and $\|\cdot\|_2$ are $\mathcal{L}_1$ and $\mathcal{L}_2$ norm, respectively. Intuitively, in the multi-objective cost function defined in (28.9), the first term specifies reconstruction errors, the second term refers to the magnitude of non-sparse representations, and the last term drives the encoder towards yielding the optimal representation.

For learning a PSD building block, the cost function in (28.9) needs to be optimized simultaneously with respect to the hidden representation and all the parameters. As a result, a learning algorithm of two alternate steps has been proposed to solve this problem [28.43] as follows:

Algorithm 28.2 PSD Learning algorithm

Given a training set of T instances $\{\boldsymbol{x}_t\}_{t=1}^T$ randomly initialize all the parameters, $W_E, W_D, G, \boldsymbol{b}_h$, and the optimal sparse representation $\{\boldsymbol{h}^*(\boldsymbol{x}_t)\}_{t=1}^T$ in a PSD building block and pre-set hyper-parameters α and β as well as learning rates ϵ_i $(i = 1, \cdots, 4)$:

- Optimal representation update
  In this step, the gradient descent method is applied to find the optimal sparse representation based on the current parameter values of the encoder and the decoder, which leads to the following update rule

$$\begin{aligned} \boldsymbol{h}^*(\boldsymbol{x}_t) \leftarrow \boldsymbol{h}^*(\boldsymbol{x}_t) - \epsilon_1 \big[&\alpha \text{sign}(\boldsymbol{h}^*(\boldsymbol{x}_t)) \\ &+ \beta(\boldsymbol{h}^*(\boldsymbol{x}_t^)) - \boldsymbol{h}(\boldsymbol{x}_t)) \\ &+ (W_D)^\mathsf{T}(W_D \boldsymbol{h}^*(\boldsymbol{x}_t) - \boldsymbol{x}_t) \big], \end{aligned}$$

  where $\text{sign}(\cdot)$ is the sign function; $\text{sign}(u) = \pm 1$ if $u \gtrless 0$ and $\text{sign}(u) = 0$ if $u = 0$.
- Parameter update
  In this step, $\boldsymbol{h}^*(\boldsymbol{x}_t)$ achieved in the above step is fixed. Then the gradient descent method is applied to the cost function (28.9) with respect to all encoder and decoder parameters, which results in the following update rules

$$W_E \leftarrow W_E - \epsilon_2 \boldsymbol{g}(\boldsymbol{x}_t)(\boldsymbol{x}_t)^\mathsf{T},$$

$$\boldsymbol{b}_h \leftarrow \boldsymbol{b}_h - \epsilon_3 \boldsymbol{g}(\boldsymbol{x}_t).$$

  Here $\boldsymbol{g}(\boldsymbol{x}_t)$ is obtained by

$$\begin{aligned} \boldsymbol{g}(\boldsymbol{x}_t) &= \big[g_{mm}^{-1}(g_{mm}^2 - h_m^2(\boldsymbol{x}_t)) \\ &\qquad (h_m^*(\boldsymbol{x}_t) - h_m(\boldsymbol{x}_t)) \big]_{m=1}^M. \end{aligned}$$

$$\begin{aligned} G \leftarrow G - \epsilon_2 \text{diag} \Bigg[&\left(h_m^*(\boldsymbol{x}_t) - h_m(\boldsymbol{x}_t) \right) \\ &\times \tanh\left(\sum_{n=1}^N [W_E]_{mn} x_{tn} + b_{v,m} \right) \Bigg], \end{aligned}$$

and

$$W_D \leftarrow W_D - \epsilon_4 \left[W_D \boldsymbol{h}^*(\boldsymbol{x}_t) - \boldsymbol{x}_t\right] \left[\boldsymbol{h}^*(\boldsymbol{x}_t)\right]^{\mathsf{T}} .$$

Normalize each column of W_D such that

$$\|[W_D]_{\cdot n}\|_2^2 = 1 \text{ for } n = 1, \cdots, N .$$

The above two steps repeat for all the instances in the given training set, which leads to a training epoch. The learning algorithm runs iteratively until it converges.

Other Building Blocks

While the auto-encoders and the RBM are building blocks widely used to construct DNNs, there are other building blocks that are either derived from existing building blocks for performance improvement or are developed with an alternative principle. Such building blocks include regularized auto-encoders and RBM variants. Due to the limited space, we briefly overview them below.

Recently, a number of auto-encoder variants have been developed by adding a regularization term to the standard reconstruction cost function in (28.1) and hence are dubbed regularized auto-encoders. The contrastive auto-encoder (CAE) is a typical regularized version of the auto-encoder with the introduction of the norm of the Jacobian matrix of the encoder evaluated at each training example $\boldsymbol{x}_t$ into the standard reconstruction cost function [28.45]

$$L_{\text{CAE}}(W, \boldsymbol{b}_h, \boldsymbol{b}_o) = L(W, \boldsymbol{b}_h, \boldsymbol{b}_o) + \alpha \sum_{t=1}^{\mathsf{T}} \|J(\boldsymbol{x}_t)\|_F^2 , \tag{28.10}$$

where α is a trade-off parameter to control the regularization strength and $\|J(\boldsymbol{x}_t)\|_F^2$ is the Frobenius norm of the Jacobian matrix of the encoder and is calculated as follows

$$\|J(\boldsymbol{x}_t)\|_F^2 = \sum_{m=1}^{M} \sum_{n=1}^{N} \left[f'(u_{h,m}(\boldsymbol{x}_t))\right]^2 W_{mn}^2 .$$

Here, $f'(\cdot)$ is the first-order derivative of a transfer function $f(\cdot)$, and $f'[u_{h,m}(\boldsymbol{x}_t)] = h_m(\boldsymbol{x}_t)[1 - h_m(\boldsymbol{x}_t)]$ when $f(\cdot)$ is the sigmoid function [28.9]. It is straightforward to apply the stochastic gradient method [28.12] to the CAE cost function in (28.10) to derive a learning algorithm used for training a CAE. Furthermore, an improved version of CAE was also proposed by penalizing additional higher order derivatives [28.46]. The sparse auto-encoder (SAE) is another class of regularized auto-encoders. The basic idea underlying SAEs is the introduction of a sparse regularization term working on either hidden neuron biases, e.g., [28.47], or their outputs, e.g., [28.48], into the standard reconstruction cost function. Different forms of sparsity penalties, e.g., $\mathcal{L}_1$ norm and student-t, are employed for regularization, and the learning algorithm is derived by applying the coordinate descent optimization procedure to a new reconstruction cost function [28.47, 48].

The RBM described above works only for an input of binary values. When an input has real values, a variant named Gaussian RBM (GRBM) [28.49], has been proposed with the following energy function

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\sum_{m=1}^{M} \sum_{n=1}^{N} W_{mn} h_m \frac{v_n}{\sigma_n} - \sum_{m=1}^{M} h_m b_{h,m} - \sum_{n=1}^{N} \frac{(v_n - b_{v,n})^2}{2\sigma_n^2} , \tag{28.11}$$

where σ_n is the standard deviation of the Gaussian noise for the visible neuron n. In the *CD* learning algorithm, the update rule for the hidden neurons remains the same except that each v_n is substituted by $\frac{v_n}{\sigma_n}$, and the update rule for all visible neurons needs to use reconstructions $\boldsymbol{v}$, produced by sampling from a Gaussian distribution with mean $\sigma_n \sum_{m=1}^{M} W_{nm} h_m + b_{v,n}$ and variance σ_n^2 for $n = 1, \cdots, N$. In addition, an improved GRBM was also proposed by introducing an alternative parameterization of the energy function in (28.11) and incorporating it into the *CD* algorithm [28.50]. Other RBM variants will be discussed later on, as they often play a different role from being used to construct a DNN.

28.2.3 Hybrid Learning Strategy

Based on the building blocks described in Sect. 28.2.2, we describe a systematic approach to establishing a feed-forward DNN for supervised and semi-supervised learning. This approach employs a hybrid learning strategy that combines unsupervised and supervised learning paradigms to overcome the optimization difficulty in training DNNs. The hybrid learning

strategy [28.18, 40] first applies layer-wise greedy unsupervised learning to set up a DNN and initialize parameters with input data only and then uses a global supervised learning algorithm with teachers' information to train all the parameters in the initialized DNN for a given task.

Layer-Wise Unsupervised Learning

In the hybrid learning strategy, unsupervised learning is a layer-wise greedy learning process that constructs a DNN with a chosen building block and initializes parameters in a layer-by-layer way.

Suppose we want to establish a DNN of K ($K>1$) hidden layers and denote output of hidden layer k as $\boldsymbol{h}_k(\boldsymbol{x})$ ($k=1,\cdots,K$) for a given input $\boldsymbol{x}$ and output of the output layer as $\boldsymbol{o}(\boldsymbol{x})$, respectively. To facilitate the presentation, we stipulate $\boldsymbol{h}_0(\boldsymbol{x})=\boldsymbol{x}$. Then, the generic layer-wise greedy learning procedure can be summarized as follows:

Algorithm 28.3 Layer-wise greedy learning procedure

Given a training set of T instances $\{\boldsymbol{x}_t\}_{t=1}^T$, randomly initialize its parameters in a chosen building block and pre-set all hyper-parameters required for learning such a building block:

- Train a building block for hidden layer k
  - Set the number of neurons required by hidden layer k to be the dimension of the hidden representation in the chosen building block.
  - Use the training data set $\{\boldsymbol{h}_{k-1}(\boldsymbol{x}_t)\}_{t=1}^T$ train the building block to achieve its optimal parameters.
- Construct a DNN up to hidden layer k
  With the trained building block in the above step, discard its decoder part, including all associated parameters, and stack its hidden layer on the existing DNN with connection weights of the encoder and biases of hidden neurons achieved in the above step (the input layer $\boldsymbol{h}_0(\boldsymbol{x})=\boldsymbol{x}$ is viewed as the starting architecture of a DNN).

The above steps are repeated for $k=1,\cdots,K$. Then, the output layer $\boldsymbol{o}(\boldsymbol{x})$ is stacked onto hidden layer K with randomly initialized connection weights so as to finalize the initial DNN construction and its parameter initialization.

Figure 28.6 illustrated two typical instances for constructing an initial DNN via the layer-wise greedy learning procedure described above. Figure 28.6a shows a schematic diagram of the layer-wise greedy learning process with the auto-encoder or its variants; to construct the hidden layer k, the output layer and its associated parameters W_k^{T} and $\boldsymbol{b}_{o,k}$ are removed and the remaining part is stacked onto hidden layer $k-1$, and W_o is a randomly initialized weight matrix for the connection between the hidden layer K and the output layer. When a DNN is constructed with the RBM or its variants, all backward connection weights in the decoder are abandoned after training and only the hidden layer with those forward connection weights and biases of hidden neurons are used to construct the DNN, as depicted in Fig. 28.6b.

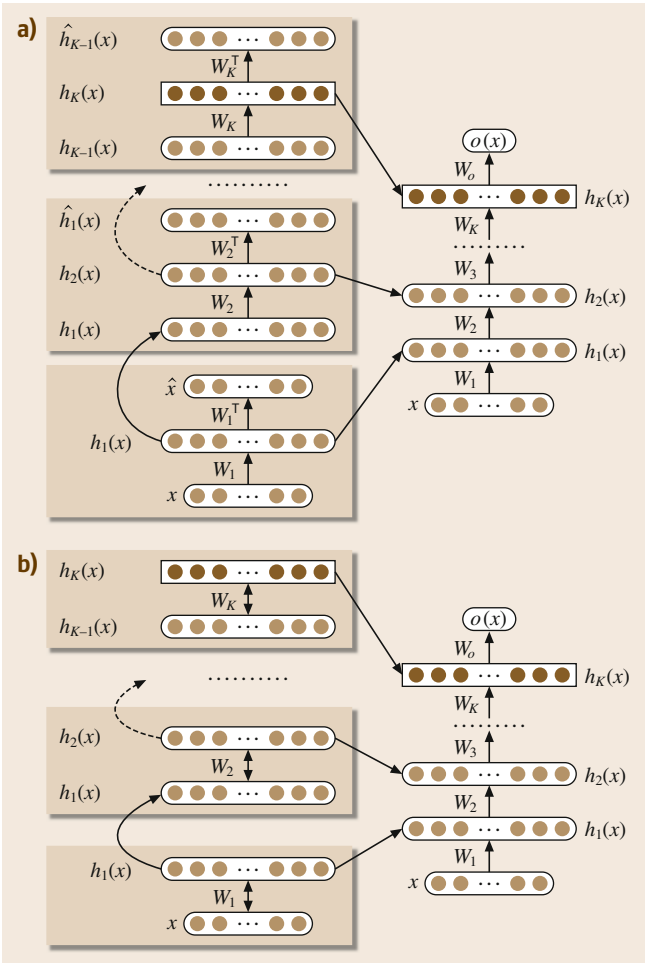

Fig. 28.6a,b Construction of a DNN with a building block via layer-wise greedy learning. **(a)** Auto-encoder or its variants. **(b)** RBM or its variants

Global Supervised Learning

Once a DNN is constructed and initialized based on the layer-wise greedy learning procedure, it is ready to be further trained in a supervision fashion for a classification or regression task. There are a variety of optimization methods for supervised learning, e.g., stochastic gradient descent and the second-order Levenberg–Marquadt methods [28.9, 12]. Also there are cost functions of different forms used for various supervised learning tasks and regularization towards improving the generalization of a DNN. Due to the limited space, we only review the stochastic gradient descent algorithm with a generic cost function for global supervised learning.

For a generic cost function $L(\Theta, \mathcal{D})$, where Θ is a collective notation of all parameters in a DNN (Fig. 28.6) and $\mathcal{D}$ is a training data set for a given supervised learning task, applying the stochastic gradient descent method [28.9, 12] to $L(\Theta, \mathcal{D})$ leads to the following learning algorithm for fine-tuning parameters.

Algorithm 28.4 Global supervised learning algorithm

Given a training set of T examples $\mathcal{D} = \{(\boldsymbol{x}_t, \boldsymbol{y}_t)\}_{t=1}^{T}$ pre-set a learning rate ϵ (and other hyper-parameters if required). Furthermore, the training set is randomly divided into many mini-batches of T_B examples $\{(\boldsymbol{x}_t, \boldsymbol{y}_t)\}_{t=1}^{T_B}$ and then parameters are updated based on each mini-batch. $\Theta = (\{W_k\}_{k=1}^{K+1}, \{\boldsymbol{b}_k\}_{k=1}^{K+1})$ are all parameters in a DNN, where W_k is the weight matrix for the connection between the hidden layers k and $k-1$, and $\boldsymbol{b}_k$ is biases of neurons in layer k (Fig. 28.6). Here, input and output layers are stipulated as layers 0 and $K+1$, respectively, i. e., $\boldsymbol{h}_0(\boldsymbol{x}_t) = \boldsymbol{x}_t$, $W_o = W_{K+1}$, $\boldsymbol{b}_o = \boldsymbol{b}_{K+1}$ and $\boldsymbol{o}(\boldsymbol{x}_t) = \boldsymbol{h}_{K+1}(\boldsymbol{x}_t)$:

- Forward computation
  Given the input $\boldsymbol{x}_t$, for $k = 1, \cdots, K+1$, the output of layer k is

$$\boldsymbol{h}_k(\boldsymbol{x}_t) = f(\boldsymbol{u}_k(\boldsymbol{x}_t)) , \quad \boldsymbol{u}_k(\boldsymbol{x}_t) = W_k \boldsymbol{h}_{k-1}(\boldsymbol{x}_t) + \boldsymbol{b}_k .$$

- Backward gradient computation
  Given a cost function on each mini-batch, $L_B(\Theta, \mathcal{D})$ calculate gradients at the output layer, i. e.,

$$\frac{\partial L_B(\Theta, \mathcal{D})}{\partial \boldsymbol{h}_{K+1}(\boldsymbol{x}_t)} = \frac{\partial L_B(\Theta, \mathcal{D})}{\partial \boldsymbol{o}(\boldsymbol{x}_t)} ,$$

$$\frac{\partial L_B(\Theta, \mathcal{D})}{\partial \boldsymbol{u}_{K+1}(\boldsymbol{x}_t)} = \left(\frac{\partial L_B(\Theta, \mathcal{D})}{\partial \boldsymbol{h}_{(K+1)j}(\boldsymbol{x}_t)} f'(u_{(K+1)j}(\boldsymbol{x}_t)) \right)_{j=1}^{|\boldsymbol{o}|} ,$$

where $f'(\cdot)$ is the first-order derivative of the transfer function $f(\cdot)$.
For all hidden layers, i. e., $k = K, \cdots, 1$, applying the chain rule leads to

$$\frac{\partial L_B(\Theta, \mathcal{D})}{\partial \boldsymbol{u}_k(\boldsymbol{x}_t)} = \left(\frac{\partial L_B(\Theta, \mathcal{D})}{\partial h_{kj}(\boldsymbol{x}_t)} f'(u_{kj}(\boldsymbol{x}_t)) \right)_{j=1}^{|\boldsymbol{h}_k|} ,$$

and

$$\frac{\partial L_B(\Theta, \mathcal{D})}{\partial \boldsymbol{h}_k(\boldsymbol{x}_t)} = \left(W_{k+1}^{(i)} \right)^{\mathsf{T}} \frac{\partial L_B(\Theta, \mathcal{D})}{\partial \boldsymbol{u}_{k+1}(\boldsymbol{x}_t)} .$$

For $k = K+1, \cdots, 1$, gradients with respect to biases of layer k are

$$\frac{\partial L_B(\Theta, \mathcal{D})}{\partial \boldsymbol{b}_k} = \frac{\partial L_B(\Theta, \mathcal{D})}{\partial \boldsymbol{u}_k(\boldsymbol{x}_t)} .$$

- Parameter update
  Applying the gradient descent method results in the following update rules:
  For $k = K+1, \cdots, 1$

$$W_k \leftarrow W_k - \frac{\epsilon}{T_B} \sum_{t=1}^{T_B} \frac{\partial L_B(\Theta, \mathcal{D})}{\partial \boldsymbol{u}_k(\boldsymbol{x}_t)} (\boldsymbol{h}_{k-1}(\boldsymbol{x}_t))^{\mathsf{T}} ,$$

$$\boldsymbol{b}_k \leftarrow \boldsymbol{b}_k - \frac{\epsilon}{T_B} \sum_{t=1}^{T_B} \frac{\partial L_B(\Theta, \mathcal{D})}{\partial \boldsymbol{u}_k(\boldsymbol{x}_t)} .$$

The above three steps repeat for all mini-batches, which leads to a training epoch. The learning algorithm runs iteratively until a termination condition is met (typically based on a cross-validation procedure [28.12]).

For the above learning algorithm, the BP algorithm [28.11] is a special case when the transfer function is the sigmoid function, i. e., $f(u) = \phi(u)$, and the cost function is the mean square error (MSE) function, i. e., for each mini-batch

$$L_B(\Theta, \mathcal{D}) = \frac{1}{2T_B} \sum_{t=1}^{T_B} \|\boldsymbol{o}(\boldsymbol{x}_t) - \boldsymbol{y}_t\|_2^2 .$$

Thus, we have

$$\phi'(u) = \phi(u)(1 - \phi(u)) ,$$

$$\frac{\partial L_B(\Theta, \mathcal{D})}{\partial \boldsymbol{o}(\boldsymbol{x}_t)} = \frac{1}{T_B} \sum_{t=1}^{T_B} (\boldsymbol{o}(\boldsymbol{x}_t) - \boldsymbol{y}_t) ,$$

and

$$\frac{\partial L_B(\Theta, \mathcal{D})}{\partial \boldsymbol{u}_{K+1}(\boldsymbol{x}_t)} = \left[\frac{1}{T_B} \sum_{t=1}^{T_B} (o_j(\boldsymbol{x}_t) - y_{tj}) o_j(\boldsymbol{x}_t)(1 - o_j(\boldsymbol{x}_t)) \right]_{j=1}^{|o|} .$$

28.2.4 Relevant Issues

In the literature, the hybrid learning strategy described in Sect. 28.2.3 is often called the semi-supervised learning strategy [28.17, 39]. Nevertheless, semi-supervised learning implies the situation that there are few labeled examples but many unlabeled instances in a training set. Indeed, such a strategy works well in a situation where both unlabeled and labeled data in a training set are used for layer-wise greedy learning, and only labeled data are used for fine-tuning in global supervised learning. However, other studies, e.g., [28.28, 29, 51], also show that this strategy can considerably improve the generalization of a DNN even though there are abundant labeled examples in a training data set. Hence we would rather name it hybrid learning. On the other hand, our review focuses on only primary supervised learning tasks in the context of NC. In a wider context, the unsupervised learning process itself develops a novel approach to automatic feature discovery/extraction via learning, which is an emerging ML area named representation learning [28.39]. In such a context, some DNNs can perform a generative model. For instance, the DBN [28.18] is a RBM-based DNN by retaining both forward and backward connections during layer-wise greedy learning. To be a generative model, the DBN needs an alternative learning algorithm, e.g., the wake–sleep algorithm [28.18], for global unsupervised learning. In general, the global unsupervised learning for a generative DNN is still a challenging problem.

While the hybrid learning strategy has been successfully applied to many complex AI tasks, in general, it is still not entirely clear why such a strategy works well empirically. A recent study attempted to provide some justification of the role played by layer-wise greedy learning for supervised learning [28.51]. The findings can be summarized as follows: such an unsupervised learning process brings about a regularization effect that initializes DNN parameters towards the basin of attraction corresponding to a *good* local minimum, which facilitates global supervised learning in terms of generalization [28.51]. In general, a deeper understanding of such a learning strategy will be required in the future. On the other hand, a successful story was recently reported [28.52] where no unsupervised pre-training was used in the DNN learning for a non-trivial task; which poses another open problem as to when and where such a learning strategy must be employed for training a DNN to yield a satisfactory generalization performance.

Recent studies also suggest that the use of artificially distorted or deformed training data and unsupervised *front-ends* can considerably improve the performance of DNNs regardless of the hybrid learning strategy. As DNN learning is of the data-driven nature, augmenting training data with known input deformation amounts to the use of more representative examples conveying intrinsic variations underlying a class of data in learning. For example, speech corrupted by some known channel noise and deformed images by using affine transformation and adding noise have significantly improved the DNN performance in various speech and visual information processing tasks [28.22, 28, 29, 51, 52]. On the other hand, the generic building blocks reviewed in Sect. 28.2.2 can be extended to be specialist *front-ends* by exploiting intrinsic data structures. For instance, the RBM has several variants, e.g., [28.53–55], to capture covariance and other statistical information underlying an image. After unsupervised learning, such *front-ends* generate powerful representations that greatly facilitate further DNN learning in visual information processing.

While our review focuses on only fully connected feed-forward DNNs, there are alternative and more effective DNN architectures for specific tasks. Convolutional DNNs [28.12] make use of topological locality constraints underlying images to form more effective locally connected DNN architecture. Furthermore, various pooling techniques [28.56] used in convolutional DNNs facilitate learning invariant and robust features. With appropriate building blocks, e.g., the PSD reviewed in Sect. 28.2.2, convolutional DNNs work very well with the hybrid learning strategy [28.43, 57]. In addition, novel DNN architectures need to be developed by exploring the nature of a specific problem, e.g., a regularized Siamese DNN was recently developed for generic speaker-specific information extraction [28.28, 29]. As a result, novel DNN architecture development and model selection are among important DNN research topics.

Finally, theoretical justification of deep learning and the hybrid learning strategy, along with other developed recently techniques, e.g., parallel graphics processing unit (GPU) computing, enable researchers to develop

large-scale DNNs to tackle very complex real world problems. While some theoretic justification has been provided in the literature, e.g., [28.16, 17, 39], to show strengths in their potential capacity and efficient representational schemes of DNNs, more and more successful applications of DNNs, particulary working with the hybrid learning strategy, lend evidence to support the argument that DNNs are one of the most promising learning systems for dealing with complex and large-scale real world problems. For example, such evidence can be found from one of the latest developments in a DNN application to computer vision where it is demonstrated that applying a DNN of nine layers constructed with the SAE building block via layer-wise greedy learning results in the favorable performance in object recognition of over 22 000 categories [28.58].

28.3 Modular Neural Networks

In this section, we review main modular neural networks (MNN) and their learning algorithms with our own taxonomy. We first review background and motivation for MNN research and present our MNN taxonomy. Then we describe major MNN architectures and relevant learning algorithms. Finally, we examine relevant issues related to MNNs in a boarder context.

28.3.1 Background and Motivation

Soon after neural network (NN) research resurged in the middle of the 1980s, MNN studies emerged; they have become an important area in NC since then. There are a variety of motivations that inspire MNN researches, e.g., biological, psychological, computational, and implementation motivations [28.4, 5, 9]. Here, we only describe the background and motivation of MNN researches from learning and computational perspectives.

From the learning perspective, MNNs have several advantages over monolithic NNs. First of all, MNNs adopt an alternative methodology for learning, so that complex problem can be solved based an ensemble of simple NNs, which might avoid/alleviate the complex optimization problems encountered in monolithic NN learning without decreasing the learning capacity. Next, modularity enables MNNs to use a priori knowledge flexibly and facilitates knowledge integration and update in learning. To a great extent, MNNs are immune to temporal and spatial *cross-talk*, a problem faced by monolithic NNs during learning [28.9]. Finally, theoretical justification and abundant empirical evidence show that an MNN often yields a better generalization than its component networks [28.5, 59]. From the computational perspective, modularization in MNNs leads to more efficient and robust computation, given the fact that MNNs often do not suffer from a high coupling burden in a monolithic NN and hence tend to have a lower overall structural complexity in tackling the same problem [28.5]. This main computational merit makes MNNs scalable and extensible to large-scale MNN implementation.

There are two highly influential principles that are often used in artificial MNN development; i. e., *divide-and-conquer* and *diversity-promotion*. The divide-and-conquer principle refers to a generic methodology that tackles a complex and difficult problem by dividing it into several relatively simple and easy subproblems, whose solutions can be combined seamlessly to yield a final solution. On the other hand, theoretical justification [28.60, 61] and abundant empirical studies [28.62] suggest that apart from the condition that component networks need to reach some certain accuracy, the success of MNNs are largely attributed to diversity among them. Hence, the promotion of diversity in MNNs becomes critical in their design and development. To understand motivations and ideas underlying different MNNs, we believe that it is crucial to examine how two principles are applied in their development.

There are different taxonomies of MNNs [28.4, 5, 9]. In this chapter, we present an alternative taxonomy that highlights the interaction among component networks in an MNN during learning. As a result, there is a dichotomy between *tightly* and *loosely coupled models* in MNNs. In a tightly coupled MNN, all component networks are jointly trained in a dependent way by taking their interaction into account during a single learning stage, and hence all parameters of different networks (and combination mechanisms if there are any) need to be updated simultaneously by minimizing a cost function defined at the global level. In contrast, training of a loosely coupled MNN often undergoes multiple stages in a hierarchical or sequential way where learning undertaken in different stages may be either correlated or uncorrelated via different strategies. We believe that such a taxonomy facilitates not only un-

derstanding different MNNs especially from a learning perspective but also relating MNNs to generic ensemble learning in a broader context.

28.3.2 Tightly Coupled Models

There are two typical tightly coupled MNNs: the mixture of experts (MoE) [28.63, 64] and MNNs trained via negative correlation learning (NCL) [28.65].

Mixture of Experts

The MoE [28.63, 64] refers to a class of MNNs that dynamically partition input space to facilitate learning in a complex and non-stationary environment. By applying the divide-and-conquer principle, a soft-competition idea was proposed to develop the MoE architecture. That is, at every input data point, multiple expert networks compete to take on a given supervised learning task. Instead of winner-take-all, all expert networks may work together but the winner expert plays a more important role than the losers.

The MoE architecture is composed of N expert networks and a gating network, as illustrated in Fig. 28.7. The n-th expert network produces an output vector, $\boldsymbol{o}_n(\boldsymbol{x})$, for an input, $\boldsymbol{x}$. The gating network receives the vector $\boldsymbol{x}$ as input and produces N scalar outputs that form a partition of the input space at each point $\boldsymbol{x}$. For the input $\boldsymbol{x}$, the gating network outputs N linear combination coefficients used to verdict the importance of all expert networks for a given supervised learning task. The final output of MoE is a convex weighted sum of all the output yielded by N expert networks. Although NNs of different types can be used as expert networks, a class of generalized linear NNs are often employed where such an NN is linear with a single output non-

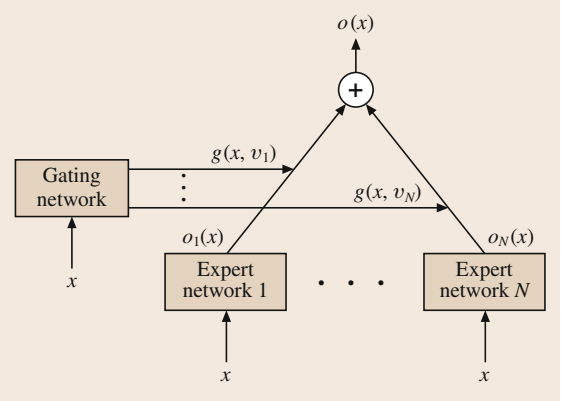

Fig. 28.7 Architecture of the mixture of experts

linearity [28.64]. As a result, output of the n-th expert network is a generalized linear function of the input $\boldsymbol{x}$

$$\boldsymbol{o}_n(\boldsymbol{x}) = f(W_n\boldsymbol{x}) ,$$

where W_n is a parameter matrix, a collective notation for both connection weights and biases, and $f(\cdot)$ is a nonlinear transfer function. The gating network is also a generalized linear model, and its n-th output $g(\boldsymbol{x}, \boldsymbol{v}_n)$ is the softmax function of $\boldsymbol{v}_n^{\mathrm{T}}\boldsymbol{x}$

$$g(\boldsymbol{x}, \boldsymbol{v}_n) = \frac{e^{\boldsymbol{v}_n^{\mathrm{T}}\boldsymbol{x}}}{\sum_{k=1}^{N} e^{\boldsymbol{v}_k^{\mathrm{T}}\boldsymbol{x}}} ,$$

where $\boldsymbol{v}_n$ is the n-th column of the parameter matrix V in the gating network and is responsible for the linear combination coefficient regarding the expert network n. The overall output of the MoE is the weighted sum resulted from the soft-competition at the point $\boldsymbol{x}$

$$\boldsymbol{o}(\boldsymbol{x}) = \sum_{n=1}^{N} g(\boldsymbol{x}, \boldsymbol{v}_n)\boldsymbol{o}_n(\boldsymbol{x}) .$$

There is a natural probabilistic interpretation of the MoE [28.64]. For a training example $(\boldsymbol{x}, \boldsymbol{y})$, the values of $\boldsymbol{g}(\boldsymbol{x}, V) = (g(\boldsymbol{x}, \boldsymbol{v}_n))_{n=1}^{N}$ are interpreted as the multinomial probabilities associated with the decision that terminates in a regressive process that maps $\boldsymbol{x}$ to $\boldsymbol{y}$. Once a decision has been made that leads to a choice of regressive process n, the output $\boldsymbol{y}$ is chosen from a probability distribution $P(\boldsymbol{y}|\boldsymbol{x}, W_n)$. Hence, the overall probability of generating $\boldsymbol{y}$ from $\boldsymbol{x}$ is the mixture of the probabilities of generating $\boldsymbol{y}$ from each component distribution and the mixing proportions are subject to a multinomial distribution

$$P(\boldsymbol{y}|\boldsymbol{x}, \Theta) = \sum_{n=1}^{N} g(\boldsymbol{x}, \boldsymbol{v}_n)P(\boldsymbol{y}|\boldsymbol{x}, W_n) , \qquad (28.12)$$

where Θ is a collective notation of all the parameters in the MoE, including both expert and gating network parameters. For different learning tasks, specific component distribution models are required. For example, the probabilistic component model should be a Gaussian distribution for a regression task, while a Bernoulli distribution and multinomial distributions are required for binary and multi-class classification tasks, respectively. In general, MoE is viewed as a conditional mixture model for supervised learning, a non-trivial extension of finite mixture model for unsupervised learning.

By means of the above probabilistic interpretation, learning in the MoE is treated as a maximum likelihood problem defined based on the model in (28.12). An expectation-maximization (EM) algorithm was proposed to update parameters in the MoE [28.64]. It is summarized as follows:

Algorithm 28.5 EM algorithm for MoE learning

Given a training set of T examples $\mathcal{D} = \{(\boldsymbol{x}_t, \boldsymbol{y}_t)\}_{t=1}^{\mathsf{T}}$ pre-set the number of expert networks, N, and randomly initialize all the parameters $\Theta = \{V, (W_n)_{n=1}^N\}$ in the MoE:

- E-step
  For each example, $(\boldsymbol{x}_t, \boldsymbol{y}_t)$ in $\mathcal{D}$, estimate posterior probabilities, $\boldsymbol{h}_t = (h_{nt})_{n=1}^N$, with the current parameter values, $\hat{V}$ and $\{\hat{W}_n\}_{n=1}^N$

$$h_{nt} = \frac{g(\boldsymbol{x}_t, \hat{\boldsymbol{v}}_n) P(\boldsymbol{y}_t | \boldsymbol{x}_t, \hat{W}_n)}{\sum_{k=1}^N g(\boldsymbol{x}_t, \hat{\boldsymbol{v}}_k) P(\boldsymbol{y}_t | \boldsymbol{x}_t, \hat{W}_k)} \,.$$

- M-step
  - For expert network n $(n = 1, \cdots, n)$, solve the maximization problems

$$\hat{W}_n = \arg\max_{W_n} \sum_{t=1}^{\mathsf{T}} h_{nt} \log P(\boldsymbol{y}_t | \boldsymbol{x}_t, W_n) \,,$$

    with all examples in $\mathcal{D}$ and posterior probabilities $\{\boldsymbol{h}_t\}_{t=1}^{\mathsf{T}}$ achieved in the E-step.
  - For the gating network, solve the maximization problem

$$\hat{V} = \arg\max_{V} \sum_{t=1}^{\mathsf{T}} \sum_{n=1}^{N} h_{nt} \log g(\boldsymbol{x}_t, \boldsymbol{v}_n) \,,$$

    with training examples, $\{(\boldsymbol{x}_t, \boldsymbol{h}_t)\}_{t=1}^{\mathsf{T}}$, derived from posterior probabilities $\{\boldsymbol{h}_t\}_{t=1}^{\mathsf{T}}$.

Repeat the E-step and the M-step alternately until the EM algorithm converges.

To solve optimization problems in the M-step, the iteratively re-weighted least squares (IRLS) algorithm was proposed [28.64]. Although the IRLS algorithm has the strength to solve maximum likelihood problems arising from MoE learning, it might result in some instable performance due to its incorrect assumption on multi-class classification [28.66]. As learning in the gating network is a multi-class classification task in essence, the problem always exists if the IRLS algorithm is used in the EM algorithm. Fortunately, improved algorithms were proposed to remedy this problem in the EM learning [28.66]. In summary, numerous MoE variants and extensions have been developed in the past 20 years [28.59], and the MoE architecture turns out to be one of the most successful MNNs.

Negative Correlation Learning

The NCL [28.65] is a learning algorithm to establish an MNN consisting of diverse neural networks (NNs) by promoting the diversity among component networks during learning. The NCL development was clearly inspired by the bias-variance analysis of generalization errors [28.60, 61]. As a result, the NCL encourages cooperation among component networks via interaction during learning to manage the bias-variance trade-off.

In the NCL, an unsupervised penalty term is introduced to the MSE cost function for each component NN so that the error diversity among component networks is explicitly managed via training towards negative correlation. Suppose that an NN ensemble $F(\boldsymbol{x}, \Theta)$ is established by simply taking the average of N neural networks $f(\boldsymbol{x}, W_n)$ $(n = 1, \cdots, N)$, where W_n denotes all the parameters in the n-th component network and $\Theta = \{W_n\}_{n=1}^N$. Given a training set $\mathcal{D} = \{(\boldsymbol{x}_t, \boldsymbol{y}_t)\}_{t=1}^{\mathsf{T}}$ the NCL cost function for the n-th component network [28.65] is defined as follows

$$L(\mathcal{D}, W_n) = \frac{1}{2T} \sum_{t=1}^{\mathsf{T}} \|f(\boldsymbol{x}_t, W_n) - \boldsymbol{y}_t\|_2^2 - \frac{\lambda}{2T} \sum_{t=1}^{\mathsf{T}} \|f(\boldsymbol{x}_t, W_n) - F(\boldsymbol{x}_t, \Theta)\|_2^2 \,, \tag{28.13}$$

where $F(\boldsymbol{x}_t, \Theta) = \frac{1}{N} \sum_{n=1}^N f(\boldsymbol{x}_t, W_n)$ and λ is a trade-off hyper-parameter. In (28.13), the first term is the MSE cost for network n and the second term refers to the negative correlation cost. By taking all component networks into account, minimizing the second term leads to maximum negative correlation among them. Therefore, λ needs to be set properly to control the penalty strength [28.65].

For the NCL, all N cost functions specified in (28.13) need to be optimized together for parameter estimation. Based on the stochastic descent method, the generic NCL algorithm is summarized as follows:

Algorithm 28.6 Negative correlation learning algorithm

Given a training set of T examples $\mathcal{D} = \{(\boldsymbol{x}_t, \boldsymbol{y}_t)\}_{t=1}^{T}$ pre-set the number of component networks, N, and learning rate, ϵ, as well as randomly initialize all the parameters $\Theta = \{W_n\}_{n=1}^{N}$ in component networks:

- Output computation
  For each example, $(\boldsymbol{x}_t, \boldsymbol{y}_t)$ in $\mathcal{D}$, calculate output of each component network $f(\boldsymbol{x}_t, W_n)$ and that of the NN ensemble by

$$F(\boldsymbol{x}_t, \Theta) = \frac{1}{N}\sum_{n=1}^{N} f(\boldsymbol{x}_t, W_n) \,.$$

- Gradient computation
  For component network n $(n = 1, \cdots, N)$, calculate the gradient of the NCL cost function in (28.13) with respect to the parameters based on all training examples in $\mathcal{D}$

$$\frac{\partial L(\mathcal{D}, W_n)}{\partial W_n} = \frac{1}{T}\sum_{t=1}^{T}\left\{ \|f(\boldsymbol{x}_t, W_n) - \boldsymbol{y}_t\|_2 - \frac{\lambda(N-1)}{N}\|f(\boldsymbol{x}_t, W_n) - F(\boldsymbol{x}_t, \Theta)\|_2 \right\} \frac{\partial f(\boldsymbol{x}_t, W_n)}{\partial W_n} \,.$$

- Parameter update
  For component network n $(n = 1, \cdots, N)$, update the parameters

$$W_n \leftarrow W_n - \epsilon \frac{\partial L(\mathcal{D}, W_n)}{\partial W_n} \,.$$

Repeat the above three steps until a pre-specified termination condition is met.

While the NCL was originally proposed based on the MSE cost function, the NCL idea can be extended to other cost functions without difficulty. Hence, applying appropriate optimization techniques on alternative cost functions leads to NCL algorithms of different forms accordingly.

28.3.3 Loosely Coupled Models

In a loosely coupled model, component networks are trained independently or there is no direct interaction among them during learning. There are a variety of MNNs that can be classified as loosely coupled models. Below we review several typical loosely coupled MNNs.

Neural Network Ensemble

An neural network ensemble here refers to a committee machine where a number of NNs trained independently but their outputs are somehow combined to reach a consensus as a final solution. The development of NN ensembles is explicitly motivated by the diversity-promotion principle [28.67, 68].

Intuitively, errors made by component NNs can be corrected by taking their diversity or mutual complement into account. For example, three NNs, NN_i $(i = 1, 2, 3)$, trained on the same data set have different yet imperfect performance on test data. Given three test points, $\boldsymbol{x}_t$, $(t = 1, 2, 3)$, NN_1 yields the correct output for $\boldsymbol{x}_2$ and $\boldsymbol{x}_3$ but does not for $\boldsymbol{x}_1$, NN_2 yields the correct output for $\boldsymbol{x}_1$, and $\boldsymbol{x}_3$ but does not for $\boldsymbol{x}_2$ and NN_3 yields the correct output for $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ but does not for $\boldsymbol{x}_3$, respectively. In such circumstances, an error made by one NN can be corrected by other two NNs with a majority vote so that the ensemble can outperform any component NNs. Formally, there is a variety of theoretical justification [28.60, 61] for NN ensembles. For example, it has been proven for regression that the NN ensemble performance is never inferior to the average performance of all component NNs [28.60]. Furthermore, a theoretical bias-variance analysis [28.61] suggests that the promotion of diversity can improve the performance of NN ensembles provided that there is an adequate trade-off between bias and variance. In general, there are two non-trivial issues in constructing NN ensembles; i. e., creating diverse component NNs and ensembling strategies.

Depending on the nature of a given problem [28.5, 62], there are several methodologies for creating diverse component NNs. First of all, a NN learning process itself can be exploited. For instance, learning in a monolithic NN often needs to solve a complex non-convex optimization problem [28.9]. Hence, a local-search-based learning algorithm, e.g., BP [28.11], may end up with various solutions corresponding to local minima due to random initialization. In addition, model selection is required to find out an appropriate NN structure for a given problem. Such properties can be exploited to create component networks in a homogeneous NN ensemble [28.67]. Next, NNs of different types trained on the same data may also yield different performance and hence are candidates in a heterogeneous NN ensemble [28.5]. Finally, exploration/exploitation of input space and different representations is an alternative

methodology for creating different component NNs. Instead of training an NN on the input space, NNs can be trained on different input subspaces achieved by a partitioning method, e.g., random partitioning [28.69], which results in a subspace NN ensemble. Whenever raw data can be characterized by different representations, NNs trained on different feature sets would constitute a multi-view NN ensemble [28.70].

Ensembling strategies are required for different tasks. For regression, some optimal fusion rules have been developed for NN ensembles, e.g., [28.68], which are supported by theoretical justification, e.g., [28.60, 61]. For classification, ensembling strategies are more complicated but have been well-studied in a wider context, named combination of multiple classifiers. As is shown in Fig. 28.8, ensembling strategies are generally divided into two categories: learnable and non-learnable. Learnable strategies use a parametric model to learn an optimal fusion rule, while non-learnable strategies fulfil the combination by directly using the statistics of all competent network outputs along with simple measures. As depicted in Fig. 28.8, there are six main non-learnable fusion rules: sum, product, min, max, median, and majority vote; details of such non-learnable rules can be found in [28.71]. Below, we focus on the main learnable ensembling strategies in terms of classification.

In general, learnable ensembling strategies are viewed as an application of the stacked generalization principle [28.72]. In light of stacked generalization, all component NNs serve as level 0 generalizers, and a learnable ensembling strategy carried out by a combination mechanism would perform a level 1 generalizer working on the output space of all component NNs to improve the overall generalization. In this sense, such a combination mechanism is trained on a validation data set that is different from the training data set used in component NN learning. As is shown in Fig. 28.8, combination mechanisms have been developed from different perspectives, i. e., input-dependent and input-independent.

An input-dependent mechanism combines component NNs based on test data; i. e., given two inputs, $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$; there is the property: $\boldsymbol{c}(\boldsymbol{x}_1|\Theta) \neq \boldsymbol{c}(\boldsymbol{x}_2|\Theta)$ if $\boldsymbol{x}_1 \neq \boldsymbol{x}_2$, where $\boldsymbol{c}(\boldsymbol{x}|\Theta) = (c_n(\boldsymbol{x}|\Theta))_{n=1}^N$ is an input-dependent mechanism used to combine an ensemble of N component NNs and Θ collectively denotes all learnable parameters in this parametric model. As a result, output of an NN ensemble with the input-dependent ensembling strategy is of the following form

$$\boldsymbol{o}(\boldsymbol{x}) = \Omega\left(\boldsymbol{o}_1(\boldsymbol{x}), \cdots, \boldsymbol{o}_N(\boldsymbol{x}) \mid \boldsymbol{c}(\boldsymbol{x}|\Theta)\right),$$

where $\boldsymbol{o}_n(\boldsymbol{x})$ is output of the n-th component NN for $n = 1, \cdots, N$ and Ω indicates a method on how to apply $\boldsymbol{c}(\boldsymbol{x}|\Theta)$ to component NNs. For example, Ω may be a linear combination scheme such that

$$\boldsymbol{o}(\boldsymbol{x}) = \sum_{n=1}^{N} c_n(\boldsymbol{x}|\Theta)\boldsymbol{o}_n(\boldsymbol{x}) \,. \tag{28.14}$$

As listed in Fig. 28.8, soft-competition and associative switch are two commonly used input-dependent combination mechanisms. The soft-competition mechanism can be regarded as a special case of the MoE described earlier when all expert networks were trained on a data set independently in advance. In this case, the gating network plays the role of the combination mechanism

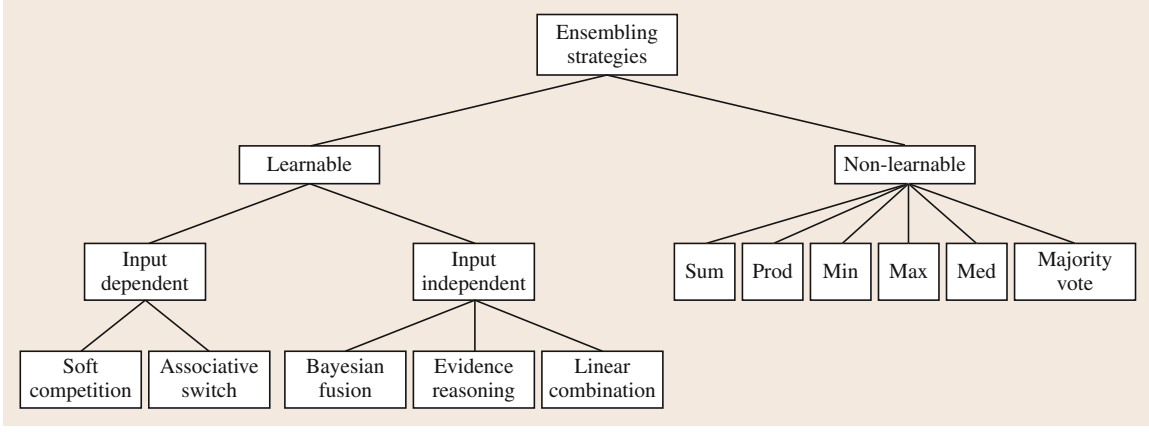

Fig. 28.8 A taxonomy of ensembling strategies

by deciding the importance of component NNs via soft-competition. Although various learning models may be used as such a gating network, a RBF-like (radial basis function) parametric model [28.73] trained on the EM algorithm has been widely used for this purpose. Unlike a soft-competition mechanism that produces the continuous-value weight vector $\boldsymbol{c}(\boldsymbol{x})$ used in (28.14), the associative switch [28.74] adopts a winner-take-all strategy, i. e., $\sum_{n=1}^{N} c_n(\boldsymbol{x}|\Theta) = 1$ and $c_n(\boldsymbol{x}|\Theta) \in \{0, 1\}$. Thus, an associative switch yields a specific code for a given input so that the output of the best performed component NN can be selected as the final output of the NN ensemble according to (28.14). The associative switch learning is a multi-class classification problem, and an MLP is often used to carry it out [28.74]. Although an input-dependent ensembling strategy is applicable to most NN ensembles, it is difficult to apply it to multi-view NN ensembles, since different representations need to be considered simultaneously in training a combination mechanism. Fortunately, such issues have been explored in a wider context on how to use different representations simultaneously for ML [28.70, 75–79] so that both soft-competition and associative switch mechanisms can be extended to multi-view NN ensembles.

In contrast, an input-independent mechanism combines component NNs based on the dependence of their outputs without considering input data directly. Given two inputs $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, and $\boldsymbol{x}_1 \neq \boldsymbol{x}_2$, the same $\boldsymbol{c}(\Theta)$ may be applied to outputs of component NNs, where $\boldsymbol{c}(\Theta) = (c_n(\Theta))_{n=1}^{N}$ is an input-independent combination mechanism used to combine an ensemble of N component NNs. Several input-independent mechanisms have been developed [28.62], which often fall into one of three categories, i. e., Bayesian fusion, evidence reasoning, and a linear combination scheme, as shown in Fig. 28.8. Bayesian fusion [28.80] refers to a class of combination schemes that use the information collected from errors made by component NNs on a validation set in order to find out the optimal output of the maximum a posteriori probability, $C^* = \arg\max_{1 \le l \le L} P(C_l|\boldsymbol{o}_1(\boldsymbol{x}), \cdots, \boldsymbol{o}_N(\boldsymbol{x}), \Theta)$, via Bayesian reasoning, where C_l is the label for the l-th class in a classification task of L classes, and Θ here encodes the information gathered, e.g., a confusion matrix achieved during learning [28.80]. Similarly, evidence reasoning mechanisms make use of alternative reasoning theories [28.80], e.g., the Dempster–Shafer theory, to yield the best output for NN ensembles via an evidence reasoning process that works on all outputs of component NNs in an ensemble. Finally, linear combination schemes of different forms are also popular as input-independent combination mechanisms [28.62]. For instance, the work presented in [28.68] exemplifies how to achieve optimal linear combination weights in a linear combination scheme.

Constructive Modularization Learning

Efforts have also been made towards constructive modularization learning for a given supervised learning task. In such work, the divide-and-conquer principle is explicitly applied in order to develop a constructive learning strategy for modularization. The basic idea behind such methods is to divide a difficult and complex problem into a number of subproblems that are easily solvable by NNs of proper capacities, matching the requirements of the subproblems, and then the solutions to subproblems are combined seamlessly to form a solution to the original problem. On the other hand, constructive modularization learning may alleviate the model selection problem encountered by a monolithic NN. As NNs of simple and even different architectures may be used to solve subproblems, empirical studies suggest that an MNN generated via constructive modularization learning is often insensitive to component NN architectures and hence is less likely to suffer from overall overfitting or underfitting [28.81]. Below we describe two constructive modularization learning strategies [28.81–83] for exemplification.

The partitioning-based strategy [28.81, 82] performs the constructive modularization learning by applying the divide-and-conquer principle explicitly. For a given supervised learning task, the strategy consists of two learning stages: *dividing* and *conquering*. In the dividing stage, it first recursively partitions the input space into overlapping subspaces, which facilitates dealing with various uncertainties, by taking into supervision information into account until the nature of each subproblem defined in generated subspaces

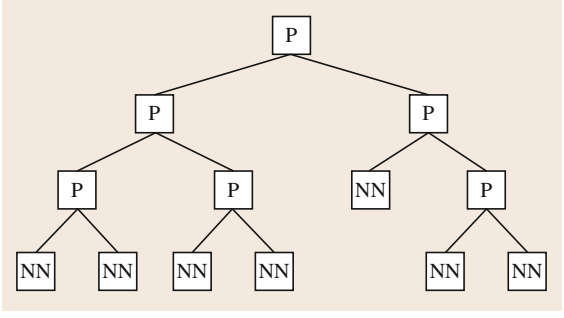

Fig. 28.9 A *self-generated* tree-structured MNN

matches the capacity of one pre-selected NN. In the conquering stage, an NN works on a given input subspace to complete the corresponding learning subtask. As a result, a tree-structured MNN is *self-generated*, where a learnable partitioning mechanism P, is situated at intermediate levels and NNs works at leaves of the tree, as illustrated in Fig. 28.9. To enable the partition-based constructive modularization learning, two generic algorithms have been proposed, i. e., growing and credit-assignment algorithms [28.81, 82] as summarized below.

Algorithm 28.7 Growing algorithm

Given a training set $\mathcal{D}$, set $\mathcal{X} \leftarrow \mathcal{D}$. Randomly initialize parameters in all component NNs in a given repository and pre-set hyper-parameters in a learnable partitioning mechanism and compatibility criteria, respectively:

- Compatibility test
  For a training (sub)set $\mathcal{X}$, apply the compatibility criteria to $\mathcal{X}$ to examine whether the learning task defined on $\mathcal{X}$ matches the capacity of a component NN in the repository.
- Partitioning space
  If none in the repository can solve the problem defined on $\mathcal{X}$, then train the partitioning mechanism on the current $\mathcal{X}$ to partition it into two overlapped $\mathcal{X}_l$ and $\mathcal{X}_r$. Set $\mathcal{X} \leftarrow \mathcal{X}_l$, then go to the *compatibility test* step. Set $\mathcal{X} \leftarrow \mathcal{X}_r$, then go to the *compatibility test* step.
  Otherwise, go to the *subproblem solving* step.
- Subproblem solving
  Train this NN on $\mathcal{X}$ with an appropriate learning algorithm. The trained NN resides at the current leaf node.

The growing algorithm expands a tree-structured MNNs until learning problems defined on all partitioned subspaces are solvable with NNs in the repository.

For a given test data point, output of such a tree-structured MNN may depend on several component NNs at the leaves of the tree since the input space is partitioned into overlapping subspaces. A credit-assignment algorithm [28.81, 82] has been developed to weight the importance of component NNs contributed to the overall output, which is summarized as follows:

Algorithm 28.8 Credit-assignment algorithm

$P(\boldsymbol{x})$ is a trained partitioning mechanisms that resides at a nonterminal node and partitions the current input (sub)space into two subspaces with an overlapping defined by $-\tau \le P(\boldsymbol{x}) \le \tau (\tau > 0)$. $C_L(\cdot)$, and $C_R(\cdot)$ are two credit assignment functions for two subspaces, respectively. For a test data point $\tilde{\boldsymbol{x}}$:

- Initialization
  Set $\alpha(\tilde{\boldsymbol{x}}) \leftarrow 1$ and `Pointer` $\leftarrow$ `Root`.
- Credit assignment
  As a recursive credit propagation process to assign credits to all the component NNs at leaf nodes that $\tilde{\boldsymbol{x}}$ can reach, $CR[\alpha(\tilde{\boldsymbol{x}}), \texttt{Pointer}]$ consists of three steps:
  - If `Pointer` points to a leaf node, then output $\alpha(\tilde{\boldsymbol{x}})$ and stop.
  - If $P(\tilde{\boldsymbol{x}}) \le \tau$, $\alpha(\tilde{\boldsymbol{x}}) \leftarrow \alpha(\tilde{\boldsymbol{x}}) \times C_L(P(\tilde{\boldsymbol{x}}))$ and invoke $CR[\alpha(\tilde{\boldsymbol{x}}), \texttt{Pointer.Leftchild}]$.
  - If $P(\tilde{\boldsymbol{x}}) \ge -\tau$, $\alpha(\tilde{\boldsymbol{x}}) \leftarrow \alpha(\tilde{\boldsymbol{x}}) \times C_R(P(\tilde{\boldsymbol{x}}))$ and invoke $CR[\alpha(\tilde{\boldsymbol{x}}), \texttt{Pointer.Rightchild}]$.

  Thus, the output of a *self-generated* MNN is

$$\boldsymbol{o}(\tilde{\boldsymbol{x}}) = \sum_{n \in \mathcal{N}} \alpha_n(\tilde{\boldsymbol{x}}) \times \boldsymbol{o}_n(\tilde{\boldsymbol{x}}) \,,$$

where $\mathcal{N}$ denotes all the component NNs that $\tilde{\boldsymbol{x}}$ can reach, and $\alpha_n(\tilde{\boldsymbol{x}})$ and $\boldsymbol{o}_n(\tilde{\boldsymbol{x}})$ are the credit assigned and the output of the n-th component NN in $\mathcal{N}$ for $\tilde{\boldsymbol{x}}$, respectively.

To implement such a strategy, hyper-planes placed with heuristics [28.81] or linear classifiers trained with the Fisher discriminative analysis [28.82] were first used as the partition mechanism and NNs such as MLP or RBF can be employed to solve subproblems. Accordingly, two piece-wise linear credit assignment functions [28.81, 82] were designed for the hyper-plane partitioning mechanism, so that $C_L(x) + C_R(x) = 1$. Heuristic compatibility criteria were developed by considering learning errors and efficiency [28.81, 82]. By using the same constructive learning algorithms described above, an alternative implementation was also proposed by using the self-organization map as a partitioning mechanism and SVMs were used for subproblem solving [28.84]. Empirical studies suggest that the partitioning-based strategy leads to favorable results in various supervised learning tasks despite different implementations [28.81, 82, 84].

By applying the divide-and-conquer principle, *task decomposition* [28.83] is yet another constructive modularization learning strategy for classification. Unlike the partitioning-based strategy, the task decomposition strategy converts a multi-class classification task into

a number of binary classification subtasks in a brute-force way and each binary classification subtask is expected to be fulfilled by a simple NN. If a subtask is too *difficult* to carry out by a given NN, the subtask is allowed to be further decomposed into simpler binary classification subtasks. For a multi-class classification task of M categories, the task decomposition strategy first exhaustively decomposes it into $\frac{1}{2}M(M-1)$ different primary binary subtasks where each subtask merely concerns classification between two different classes without taking remaining $M-2$ classes into account, which differs from the commonly used one-against-rest decomposition method. In general, the original multi-class classification task may be decomposed into more binary subtasks if some primary subtasks are too *difficult*. Once the decomposition is completed, all the subtasks are undertaken by pre-selected simple NNs, e.g., MLP of one hidden layer, in parallel. For a final solution to the original problem, three non-learnable operations, *min*, *max*, and *inv*, were proposed to combine individual binary classification results achieved by all the component NNs. By applying three operations properly, all the component NNs are integrated together to form a min-max MNN [28.83].

28.3.4 Relevant Issues

In general, studies of MNNs closely relate to several areas in different disciplines, e.g., ML and statistics. We here examine several important issues related to MNNs in a wider context.

As described above, a tightly coupled MNN leads to an optimal solution to a given supervised learning problem. The MoE is rooted in the finite mixture model (FMM) studied in probability and statistics and becomes a non-trivial extension to conditional models where each expert is a parametric conditional probabilistic model and the mixture coefficients also depend on input [28.64]. While the MoE has been well studied for 20 years [28.59] in different disciplines, there still exist some open problems in general, e.g., model selection, global optimal solution, and convergence of its learning algorithms for arbitrary component models. Different from the FMM, the product of experts (PoE) [28.42] was also proposed to combine a number of experts (parametric probabilistic models) by taking their product and normalizing the result into account. The PoE has been argued to have some advantages over the MoE [28.42] but has so far merely been developed in the context of unsupervised learning. As a result, extending the PoE to conditional models for supervised learning would be a non-trivial topic in tightly coupled MNN studies. On the other hand, the NCL [28.65] directly applies the bias-variance analysis [28.60, 61] to construction of an MNN. This implies that MNNs could be also built up via alternative loss functions that properly promote diversities among component MNNs during learning.

Almost all existing NN ensemble methods are now included in ensemble learning [28.85], which is an important area in ML, or the multiple classifier system [28.62] in the context of pattern recognition. In statistical ensemble learning, generic frameworks, e.g., boosting [28.86] and bootstrapping [28.87], were developed to construct ensemble learners where any learning models including NNs may be used as component learners. Hence, most of common issues raised for ensemble learning are applicable to NN ensembles. Nevertheless, ensemble learning researches suggest that behaviors of component learners may considerably affect the stability and overall performance of ensemble learning. As exemplified in [28.88], properties of different NN ensembles are worth investigating from both theoretical and application perspectives.

While constructive modularization learning provides an alternative way of model selection, it is generally a less developed area in MNNs, and existing methods are subject to limitation due to a lack of theoretical justification and underpinning techniques. For example, a critical issue in the partitioning-based strategy [28.81, 82] is how to measure the nature of a subproblem to decide if any further partitioning is required and the appropriateness of a pre-selected NN to a subproblem in terms of its capacity. In previous studies [28.81, 82], a number of heuristic and simple criteria were proposed based on learning errors and efficiency. Although such heuristic criteria work practically, there is no theoretical justification. As a result, more sophisticated compatibility criteria need to be developed for such a constructive learning strategy based on the latest ML development, e.g., manifold and adaptive kernel learning. Fortunately, the partitioning-based strategy has inspired the latest developments in ML [28.89]. In general, constructive modularization learning is still a non-trivial topic in MNN research.

Finally, it is worth stating that our MNN review here only focuses on supervised learning due to the limited space. Most MNNs described above may be extended to other learning paradigms, e.g., semi-supervised and unsupervised learning. More details on such topics are available in the literature, e.g., [28.90, 91].

28.4 Concluding Remarks

In this chapter, we have reviewed two important areas, DNNs and MNNs, in NC. While we have presented several sophisticated techniques that are ready for applications, we have discussed several challenging problems in both deep and modular neural network research as well. Apart from other non-trivial issues discussed in the chapter, it is worth emphasizing that it is still an open problem to develop large-scale DNNs and MNNs and integrate them for modeling highly intelligent behaviors, although some progress has been made recently [28.58]. In a wider context, DNNs and MNNs are closely related to two active areas, deep learning and ensemble learning, in ML. We anticipate that motivation and methodologies from different perspectives will mutually benefit each other and lead to effective solutions to common challenging problems in the NC and ML communities.

References

28.1 E.R. Kandel, J.H. Schwartz, T.M. Jessell: *Principle of Neural Science*, 4th edn. (McGraw-Hill, New York 2000)

28.2 G.M. Edelman: *Neural Darwinism: Theory of Neural Group Selection* (Basic Books, New York 1987)

28.3 J.A. Fodor: *The Modularity of Mind* (MIT Press, Cambridge 1983)

28.4 F. Azam: Biologically inspired modular neural networks, Ph.D. Thesis (School of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Blacksburg 2000)

28.5 G. Auda, M. Kamel: Modular neural networks: A survey, Int. J. Neural Syst. **9**(2), 129–151 (1999)

28.6 D.C. Van Essen, C.H. Anderson, D.J. Fellman: Information processing in the primate visual system, Science **255**, 419–423 (1992)

28.7 J.H. Kaas: Why does the brain have so many visual areas?, J. Cogn. Neurosci. **1**(2), 121–135 (1989)

28.8 G. Bugmann: Biologically plausible neural computation, Biosystems **40**(1), 11–19 (1997)

28.9 S. Haykin: *Neural Networks and Learning Machines*, 3rd edn. (Prentice Hall, New York 2009)

28.10 M. Minsky, S. Papert: *Perceptrons* (MIT Press, Cambridge 1969)

28.11 D.E. Rumelhurt, G.E. Hinton, R.J. Williams: Learning internal representations by error propagation, Nature **323**, 533–536 (1986)

28.12 Y. LeCun, L. Bottou, Y. Bengio, P. Haffner: Gradient based learning applied to document recognition, Proc. IEEE **86**(9), 2278–2324 (1998)

28.13 G. Tesauro: Practical issues in temporal difference learning, Mach. Learn. **8**(2), 257–277 (1992)

28.14 G. Cybenko: Approximations by superpositions of sigmoidal functions, Math. Control Signals Syst. **2**(4), 302–314 (1989)

28.15 N. Cristianini, J. Shawe-Taylor: *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods* (Cambridge University Press, Cambridge 2000)

28.16 Y. Bengio, Y. LeCun: Scaling learning algorithms towards AI. In: *Large-Scale Kernel Machines*, ed. by L. Bottou, O. Chapelle, D. DeCoste, J. Weston (MIT Press, Cambridge 2006), Chap. 14

28.17 Y. Bengio: Learning deep architectures for AI, Found. Trends Mach. Learn. **2**(1), 1–127 (2009)

28.18 G.E. Hinton, S. Osindero, Y. Teh: A fast learning algorithm for deep belief nets, Neural Comput. **18**(9), 1527–1554 (2006)

28.19 Y. Bengio: Deep learning of representations for unsupervised and transfer learning, JMLR: Workshop Conf. Proc., Vol. 7 (2011) pp. 1–20

28.20 H. Larochelle, D. Erhan, A. Courville, J. Bergstra, Y. Bengio: An empirical evaluation of deep architectures on problems with many factors of variation, Proc. Int. Conf. Mach. Learn. (ICML) (2007) pp. 473–480

28.21 R. Salakhutdinov, G.E. Hinton: Learning a nonlinear embedding by preserving class neighbourhood structure, Proc. Int. Conf. Artif. Intell. Stat. (AISTATS) (2007)

28.22 H. Larochelle, Y. Bengio, J. Louradour, P. Lamblin: Exploring strategies for training deep neural networks, J. Mach. Learn. Res. **10**(1), 1–40 (2009)

28.23 W.K. Wong, M. Sun: Deep learning regularized Fisher mappings, IEEE Trans. Neural Netw. **22**(10), 1668–1675 (2011)

28.24 S. Osindero, G.E. Hinton: Modeling image patches with a directed hierarchy of Markov random field, Adv. Neural Inf. Process. Syst. (NIPS) (2007) pp. 1121–1128

28.25 I. Levner: Data driven object segmentation, Ph.D. Thesis (Department of Computer Science, University of Alberta, Edmonton 2008)

28.26 H. Mobahi, R. Collobert, J. Weston: Deep learning from temporal coherence in video, Proc. Int. Conf. Mach. Learn. (ICML) (2009) pp. 737–744

28.27 H. Lee, Y. Largman, P. Pham, A. Ng: Unsupervised feature learning for audio classification using convolutional deep belief networks, Adv. Neural Inf. Process. Syst. (NIPS) (2009)

28.28 K. Chen, A. Salman: Learning speaker-specific characteristics with a deep neural architec-

ture, IEEE Trans. Neural Netw. **22**(11), 1744–1756 (2011)
28.29 K. Chen, A. Salman: Extracting speaker-specific information with a regularized Siamese deep network, Adv. Neural Inf. Process. Syst. (NIPS) (2011)
28.30 A. Mohamed, G.E. Dahl, G.E. Hinton: Acoustic modeling using deep belief networks, IEEE Trans. Audio Speech Lang. Process. **20**(1), 14–22 (2012)
28.31 G.E. Dahl, D. Yu, L. Deng, A. Acero: Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition, IEEE Trans. Audio Speech Lang. Process. **20**(1), 30–42 (2012)
28.32 R. Salakhutdinov, G.E. Hinton: Semantic hashing, Proc. SIGIR Workshop Inf. Retr. Appl. Graph. Model. (2007)
28.33 M. Ranzato, M. Szummer: Semi-supervised learning of compact document representations with deep networks, Proc. Int. Conf. Mach. Learn. (ICML) (2008)
28.34 A. Torralba, R. Fergus, Y. Weiss: Small codes and large databases for recognition, Proc. Int. Conf. Comput. Vis. Pattern Recogn. (CVPR) (2008) pp. 1–8
28.35 R. Collobert, J. Weston: A unified architecture for natural language processing: Deep neural networks with multitask learning, Proc. Int. Conf. Mach. Learn. (ICML) (2008)
28.36 A. Mnih, G.E. Hinton: A scalable hierarchical distributed language model, Adv. Neural Inf. Process. Syst. (NIPS) (2008)
28.37 J. Weston, F. Ratle, R. Collobert: Deep learning via semi-supervised embedding, Proc. Int. Conf. Mach. Learn. (ICML) (2008)
28.38 R. Hadsell, A. Erkan, P. Sermanet, M. Scoffier, U. Muller, Y. LeCun: Deep belief net learning in a long-range vision system for autonomous off-road driving, Proc. Intell. Robots Syst. (IROS) (2008) pp. 628–633
28.39 Y. Bengio, A. Courville, P. Vincent: Representation learning: A review and new perspectives, IEEE Trans. Pattern Anal. Mach. Intell. **35**(8), 1798–1827 (2013)
28.40 Y. Bengio, P. Lamblin, D. Popovici, H. Larochelle: Greedy layer-wise training of deep networks, Adv. Neural Inf. Process. Syst. (NIPS) (2006)
28.41 P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, P.A. Manzagol: Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion, J. Mach. Learn. Res. **11**, 3371–3408 (2010)
28.42 G.E. Hinton: Training products of experts by minimizing contrastive divergence, Neural Comput. **14**(10), 1771–1800 (2002)
28.43 K. Kavukcuoglu, M. Ranzato, Y. LeCun: Fast inference in sparse coding algorithms with applications to object recognition. CoRR, arXiv:1010.3467 (2010)
28.44 B.A. Olshausen, D.J. Field: Sparse coding with an overcomplete basis set: A strategy employed by V1?, Vis. Res. **37**, 3311–3325 (1997)
28.45 S. Rifai, P. Vincent, X. Muller, X. Glorot, Y. Bengio: Contracting auto-encoders: Explicit invariance during feature extraction, Proc. Int. Conf. Mach. Learn. (ICML) (2011)
28.46 S. Rifai, G. Mesnil, P. Vincent, X. Muller, Y. Bengio, Y. Dauphin, X. Glorot: Higher order contractive auto-encoder, Proc. Eur. Conf. Mach. Learn. (ECML) (2011)
28.47 M. Ranzato, C. Poultney, S. Chopra, Y. LeCun: Efficient learning of sparse representations with an energy based model, Adv. Neural Inf. Process. Syst. (NIPS) (2006)
28.48 M. Ranzato, Y. Boureau, Y. LeCun: Sparse feature learning for deep belief networks, Adv. Neural Inf. Process. Syst. (NIPS) (2007)
28.49 G.E. Hinton, R. Salakhutdinov: Reducing the dimensionality of data with neural networks, Science **313**, 504–507 (2006)
28.50 K. Cho, A. Ilin, T. Raiko: Improved learning of Gaussian-Bernoulli restricted Boltzmann machines, Proc. Int. Conf. Artif. Neural Netw. (ICANN) (2011)
28.51 D. Erhan, Y. Bengio, A. Courville, P.A. Manzagol, P. Vincent, S. Bengio: Why does unsupervised pre-training help deep learning?, J. Mach. Learn. Res. **11**, 625–660 (2010)
28.52 D.C. Ciresan, U. Meier, L.M. Gambardella, J. Schmidhuber: Deep big simple neural nets for handwritten digit recognition, Neural Comput. **22**(1), 1–14 (2010)
28.53 M. Ranzato, A. Krizhevsky, G.E. Hinton: Factored 3-way restricted Boltzmann machines for modeling natural images, Proc. Int. Conf. Artif. Intell. Stat. (AISTATS) (2010) pp. 621–628
28.54 M. Ranzato, V. Mnih, G.E. Hinton: Generating more realistic images using gated MRF's, Adv. Neural Inf. Process. Syst. (NIPS) (2010)
28.55 A. Courville, J. Bergstra, Y. Bengio: Unsupervised models of images by spike-and-slab RBMs, Proc. Int. Conf. Mach. Learn. (ICML) (2011)
28.56 H. Lee, R. Grosse, R. Ranganath, A.Y. Ng: Unsupervised learning of hierarchical representations with convolutional deep belief networks, Commun. ACM **54**(10), 95–103 (2011)
28.57 D. Hau, K. Chen: Exploring hierarchical speech representations with a deep convolutional neural network, Proc. U.K. Workshop Comput. Intell. (UKCI) (2011)
28.58 Q. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G.S. Corrado, J. Dean, A.Y. Ng: Building high-level features using large scale unsupervised learning, Proc. Int. Conf. Mach. Learn. (ICML) (2012)
28.59 S.E. Yuksel, J.N. Wilson, P.D. Gader: Twenty years of mixture of experts, IEEE Trans. Neural Netw. Learn. Syst. **23**(8), 1177–1193 (2012)
28.60 A. Krogh, J. Vedelsby: Neural network ensembles, cross validation, and active learning, Adv. Neural Inf. Process. Syst. (NIPS) (1995)

28.61 N. Ueda, R. Nakano: Generalization error of ensemble estimators, Proc. Int. Conf. Neural Netw. (ICNN) (1996) pp. 90–95

28.62 L.I. Kuncheva: *Combining Pattern Classifiers* (Wiley-Interscience, Hoboken 2004)

28.63 R.A. Jacobs, M.I. Jordan, S. Nowlan, G.E. Hinton: Adaptive mixture of local experts, Neural Comput. **3**(1), 79–87 (1991)

28.64 M.I. Jordan, R.A. Jacobs: Hierarchical mixture of experts and the EM algorithm, Neural Comput. **6**(2), 181–214 (1994)

28.65 Y. Liu, X. Yao: Simultaneous training of negatively correlated neural networks in an ensemble, IEEE Trans. Syst. Man Cybern. B **29**(6), 716–725 (1999)

28.66 K. Chen, L. Xu, H.S. Chi: Improved learning algorithms for mixture of experts in multi-class classification, Neural Netw. **12**(9), 1229–1252 (1999)

28.67 L.K. Hansen, P. Salamon: Neural network ensembles, IEEE Trans. Pattern Anal. Mach. Intell. **12**(10), 993–1001 (1990)

28.68 M.P. Perrone, L.N. Cooper: Ensemble methods for hybrid neural networks. In: *Artificial Neural Networks for Speech and Vision*, ed. by R.J. Mammone (Chapman-Hall, New York 1993) pp. 126–142

28.69 T.K. Ho: The random subspace method for constructing decision forests, IEEE Trans. Pattern Anal. Mach. Intell. **20**(8), 823–844 (1998)

28.70 K. Chen, L. Wang, H.S. Chi: Methods of combining multiple classifiers with different feature sets and their applications to text-independent speaker identification, Int. J. Pattern Recogn. Artif. Intell. **11**(3), 417–445 (1997)

28.71 J. Kittler, M. Hatef, R.P.W. Duin, J. Matas: On combining classifiers, IEEE Trans. Pattern Anal. Mach. Intell. **20**(3), 226–239 (1998)

28.72 D.H. Wolpert: Stacked generalization, Neural Netw. **2**(3), 241–259 (1992)

28.73 L. Xu, M.I. Jordan, G.E. Hinton: An alternative model for mixtures of experts, Adv. Neural Inf. Process. Syst. (NIPS) (1995)

28.74 L. Xu, A. Krzyzak, C.Y. Suen: Associative switch for combining multiple classifiers, J. Artif. Neural Netw. **1**(1), 77–100 (1994)

28.75 K. Chen: A connectionist method for pattern classification with diverse feature sets, Pattern Recogn. Lett. **19**(7), 545–558 (1998)

28.76 K. Chen, H.S. Chi: A method of combining multiple probabilistic classifiers through soft competition on different feature sets, Neurocomputing **20**(1–3), 227–252 (1998)

28.77 K. Chen: On the use of different representations for speaker modeling, IEEE Trans. Syst. Man Cybern. C **35**(3), 328–346 (2005)

28.78 Y. Yang, K. Chen: Temporal data clustering via weighted clustering ensemble with different representations, IEEE Trans. Knowl. Data Eng. **23**(2), 307–320 (2011)

28.79 Y. Yang, K. Chen: Time series clustering via RPCL ensemble networks with different representations, IEEE Trans. Syst. Man Cybern. C **41**(2), 190–199 (2011)

28.80 L. Xu, A. Krzyzak, C.Y. Suen: Methods of combining multiple classifiers and their applications to handwriting recognition, IEEE Trans. Syst. Man Cybern. **22**(3), 418–435 (1992)

28.81 K. Chen, X. Yu, H.S. Chi: Combining linear discriminant functions with neural networks for supervised learning, Neural Comput. Appl. **6**(1), 19–41 (1997)

28.82 K. Chen, L.P. Yang, X. Yu, H.S. Chi: A self-generating modular neural network architecture for supervised learning, Neurocomputing **16**(1), 33–48 (1997)

28.83 B.L. Lu, M. Ito: Task decomposition and module combination based on class relations: A modular neural network for pattern classification, IEEE Trans. Neural Netw. Learn. Syst. **10**(5), 1244–1256 (1999)

28.84 L. Cao: Support vector machines experts for time series forecasting, Neurocomputing **51**(3), 321–339 (2003)

28.85 T.G. Dietterich: Ensemble learning. In: *Handbook of Brain Theory and Neural Networks*, ed. by M.A. Arbib (MIT Press, Cambridge 2002) pp. 405–408

28.86 Y. Freund, R.E. Schapire: Experiments with a new boosting algorithm, Proc. Int. Conf. Mach. Learn. (ICML) (1996) pp. 148–156

28.87 L. Breiman: Bagging predictors, Mach. Learn. **24**(2), 123–140 (1996)

28.88 H. Schwenk, Y. Bengio: Boosting neural networks, Neural Comput. **12**(8), 1869–1887 (2000)

28.89 J. Wang, V. Saligrama: Local supervised learning through space partitioning, Adv. Neural Inf. Process. Syst. (NIPS) (2012)

28.90 X.J. Zhu: *Semi-supervised learning literature survey, Technical Report, School of Computer Science* (University of Wisconsin, Madison 2008)

28.91 J. Ghosh, A. Acharya: Cluster ensembles, WIREs Data Min. Knowl. Discov. **1**(2), 305–315 (2011)