

Taeho Jo

---

# Deep Learning Foundations

# Deep Learning Foundations

Taeho Jo

# Deep Learning Foundations



Springer

Taeho Jo  
Alpha Lab AI  
Cheongju, Korea (Republic of)

ISBN 978-3-031-32878-7      ISBN 978-3-031-32879-4 (eBook)  
<https://doi.org/10.1007/978-3-031-32879-4>

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG  
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

# Preface

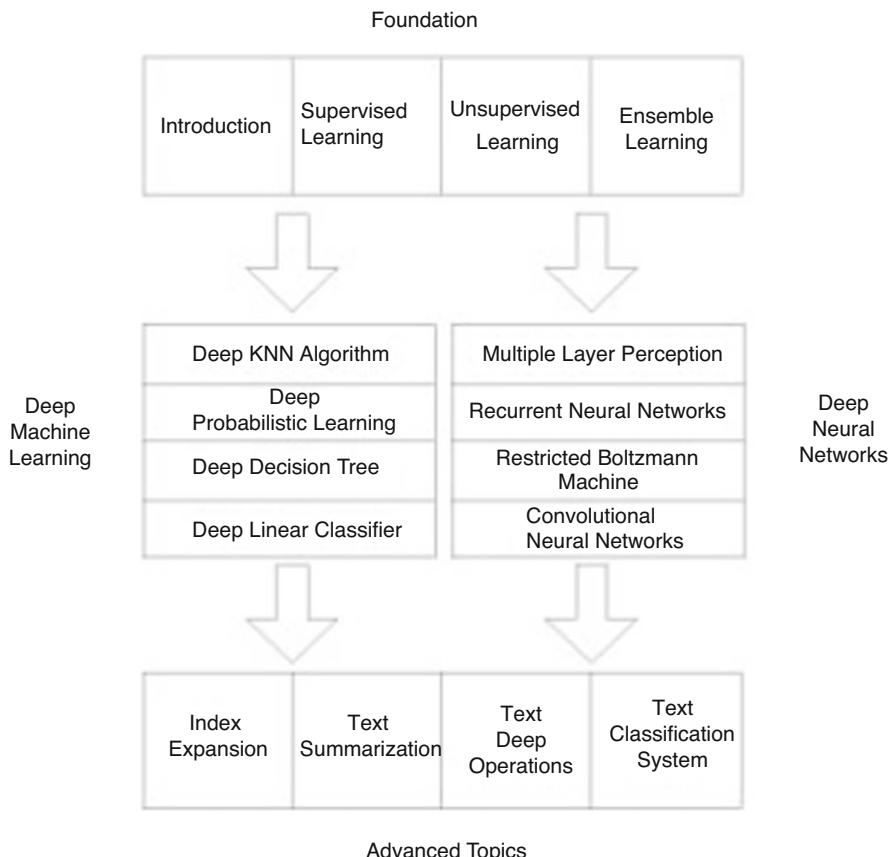
This book is concerned with the concepts, theories, and algorithms of deep learning. In Part I, we provide the fundamental knowledge about deep learning by exploring the traditional supervised and unsupervised learning algorithms and the alternative advanced machine learning type, called ensemble learning. In Parts II and III, we describe the modification of existing machine learning algorithms into deep versions and the deep neural networks, such as Multiple Layer Perceptron, Recurrent Neural Networks, Restricted Boltzmann Machine, and Convolutional Neural Networks, respectively. In Part IV, we cover the specialization of deep learning algorithms into the text mining tasks. Readers need the basic level of knowledge about linear algebra, vector calculus, and traditional machine learning algorithms for understanding the deep learning algorithms.

We mention some agenda which provide motivations for writing this book. The machine learning algorithms became very popular approaches to data mining tasks, because of their flexibility and adaptability, compared with the rule-based approaches. The deep learning algorithms are derived from the machine learning algorithms and become the advanced approaches to nonlinear problems. The deep learning algorithms are applied to tasks in various disciplines such as control system engineering, chemical engineering, and business administration and management. In this book, we describe the deep learning algorithms with the assumption that the input data is given as a numerical vector.

This book consists of the four parts: foundation, deep machine learning, deep neural networks, and textual deep learning. In the first part, we cover the swallow supervised learning, the swallow unsupervised learning, and the ensemble learning as background for understanding the deep learning. In the second part, we modify the four kinds of supervised machine learning algorithms, K-Nearest Neighbor, Probabilistic Learning, the Decision Tree, and the SVM (Support Vector Machine), into their deep learning versions. In the third part, we describe some typical deep learning algorithms: MLP (Multiple Layer Perceptron), Recurrent Neural Networks, Restricted Boltzmann Machine, and Convolutional Neural Networks. In the last part, we specialize some deep operations on numerical values into ones on textual data; the text summarization is used to extract some of each article, like the pooling.

This book is written, intended for the three groups of readers: students, professors, and researchers. Senior undergraduate and graduate students are able to study the contents of the deep learning by themselves with this book. For lecturers and professors in universities, this book may be used as the main material for providing lectures on the deep learning. This book is useful for researchers and system developers in industrial organizations for developing intelligent systems using the deep learning algorithms as tools. Therefore, this book is practical for people in both words: the academic world and the industrial one, to study the deep learning.

The chapters of this book are organized with the four parts as shown in Fig. 1. Part I is concerned with the introduction, the traditional supervised learning, the traditional unsupervised learning, and the ensemble learning as the foundation of the machine learning. Part II focuses on the modification of existing machine learning algorithms such as the KNN algorithm, the probabilistic learning algorithm, the



**Fig. 1** The organization of chapters in this book

decision tree, and the linear classifier, into their deep versions. Part III does on the popular deep neural networks: the Multiple Layer Perceptron, the Recurrent Neural Networks, the Restricted Boltzmann Machine, and the Convolutional Neural Networks. Part IV is concerned with the specialization of deep operations to textual data.

## Part I: Foundation

The first part of this book is concerned with the foundation for studying the deep learning algorithms. Because the deep learning is the area, which is expanded from the machine learning, we need to study the machine learning algorithms before doing the deep learning algorithms. There are four machine learning types: supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning, but we will cover the supervised learning and the unsupervised learning. The ensemble learning is viewed as the learning type where multiple machine learning algorithms are combined for solving their own demerits as the alternative advanced learning to the deep learning. This part is intended to review the supervised learning algorithms, the unsupervised ones, and the ensemble learning, for providing the foundation for understanding the deep learning.

Chapter 1 is concerned with the overview of deep learning. Before discussing the deep learning, let us discuss the swallow learning, which is opposite to the deep learning, and in the swallow learning, the output value is computed directly from the input value. The deep learning is characterized as the process of computing intermediate values between input values and output values, and the input encoding, the output decoding, the convolution, and the unsupervised layer between the input layer and the output layer become the components for implementing the deep learning. There are other advanced learning types than the deep learning: the kernel-based learning, the ensemble learning, and the semi-supervised learning. This chapter is intended to describe the deep learning conceptually for providing the introduction.

Chapter 2 is concerned with the supervised learning as a kind of swallow learning. The supervised learning refers to the leaning type where its parameters are optimized for minimizing the error between the desired output and the computed one. In the supervised learning process, the training examples each of which is labeled with its own target output are given, and the given learning algorithm is trained with them. Supervised learning algorithms are applied to classification and regression. This chapter is intended to review the supervised learning as a kind of swallow learning, before studying the deep learning.

Chapter 3 is concerned with the unsupervised learning as another kind of swallow learning. It refers to the learning types where its parameters are optimized for stabilizing distances of cluster members from their own cluster prototypes. In the process of the unsupervised learning, unlabeled training examples are prepared, a similarity metric or a distance metric between training examples is defined, and

cluster prototypes are optimized for maximizing their cohesions. Unsupervised learning algorithms are applied to clustering which is the process of segmenting a data item group into subgroups each of which contains similar ones. This chapter is intended to review the unsupervised learning before studying the deep learning.

Chapter 4 is concerned with the ensemble learning as an alternative advanced learning to the deep learning. It refers to the learning type for solving the problems such as classification, regression, and clustering, more reliably by combining multiple machine learning algorithms with each other. The typical schemes of ensemble learning are the voting which is the process of deciding the final answer by considering ones of multiple machine learning algorithms, the expert gate which is the process of nominating one among the machine learning algorithms as an expert which is suitable for solving the given input, and the cascading which is the process of deciding whether the current answer is adopted or the input is transferred to the next machine learning algorithm. We need to consider partitioning the training set which is viewed as a matrix with the two axes: the horizontal partition which partitions the training set into subsets and the vertical partition which partitions the attribute set. This chapter is intended to describe the ensemble learning as an advanced type of advanced learning.

## Part II: Deep Machine Learning

Part II is concerned with the modification of existing machine learning algorithms into the deep versions. We mention the typical supervised machine learning algorithms such as the KNN algorithm, the probabilistic learning, the decision tree, and the SVM (Support Vector Machine), as the targets which we modify into their deep versions. The machine learning algorithms are modified into the deep versions by attaching the input encoding and/or the output decoding as the basic schemes. As the advanced scheme, the supervised learning algorithms are modified into their unsupervised versions, and the original version and the unsupervised version are combined into the deep version, serially. This part is intended to study the modification of the existing machine learning algorithms into the deep versions.

Chapter 5 is concerned with the modification of the KNN algorithm into its deep versions. The KNN algorithm where a novice example is classified depending on labels of its nearest neighbors is the simplest supervised learning algorithm. As the basis schemes, the KNN algorithm is modified into their deep versions by attaching the input encoding and/or the output decoding. As the advanced schemes, the KNN algorithm is modified into an unsupervised version, and its unsupervised version and its original version are combined with each other into a deep version. This chapter is intended to describe the schemes of modifying the KNN algorithm into its deep versions.

Chapter 6 is concerned with the modification of the Bayes Classifier and the Naive Bayes into their deep version. In the probabilistic learning, we mentioned the three typical machine learning algorithms, the Bayes Classifier, the Naive Bayes,

and the Bayesian Learning, in [1]. As the basic schemes, the Bayes Classifier and the Naive Bayes are modified into their deep versions by attaching the input encoding and/or the output encoding. As the advanced schemes, they are modified into their unsupervised versions, and their original versions and their unsupervised versions are combined with each other into their stacked version. This chapter is intended to describe the schemes of modifying the two probabilistic learning algorithms into their deep versions.

Chapter 7 is concerned with the modification of the decision tree and its variants into their deep versions. The decision tree which is a typical symbolic machine learning algorithm, where it is able to trace its classification into logical rules, classifies a data item by following branches which correspond to its attribute values. As the basic schemes, a decision tree is modified into its deep versions by attaching the input encoding and/or the output decoding. As the advanced schemes, a decision tree is modified into its unsupervised version, and its original and modified versions are combined with each other into the stacked version. This chapter is intended to describe the schemes of modifying the decision tree into its deep versions.

Chapter 8 is concerned with the modification of the linear classifier and the SVM into their deep versions. The SVM is a linear classifier which defines its dual parallel hyperplanes with the maximal margin between them as the classification boundary. Even if the SVM is viewed as a deep learning algorithm, compared with the simple linear classifier, by itself, it may be modified into its further deep versions by attaching the input encoding and/or the output encoding. As the advanced schemes, the SVM is modified into its unsupervised version, and its original and modified versions are combined with each other into the stacked version. This chapter is intended to describe the schemes of modifying the SVM into its deep versions.

## Part III: Deep Neural Networks

Part III is concerned with the neural networks with their deep architectures. The neural networks are the kind of machine learning algorithms which simulate the nervous system. It is composed with neurons which are connected with each other, and each connection is called synaptic weight. The process of training the neural networks is to optimize weight values which are associated with their own connections among neurons, for minimizing the error on training examples. This part is intended to describe some neural networks, such as MLP (Multiple Layer Perceptron), the RNN (Recurrent Neural Networks), and the RBM (Restricted Boltzmann Machine).

Chapter 9 is concerned with the MLP as a typical deep neural networks model. The Perceptron which is basis for inventing the MLP was invented by Rosenblatt in 1950s. In the architecture of MLP, there are three layers: the input layer, the hidden layer, and the output layer. A layer is connected to its next layer with the feedforward direction, and the weights are updated in its learning process in the

backward direction. This chapter is intended to describe the MLP with respect to the architecture, the computation process, and the learning process.

Chapter 10 is concerned with the recurrent neural networks which are advanced from the MLP. In the previous chapter, we studied the MLP as a typical deep neural networks model with the feedforward connections between layers. In this chapter, we study the recurrent neural network with any feedback connection from a neuron as an advanced model. In the feedback connection, a previous output value is used as an input. This chapter is intended to describe the recurrent neural networks and the variants with respect to the connection and the learning process.

Chapter 11 is concerned with the RBM for restoring the input. The Hopfield Networks was intended as the associative memory for restoring the input vector, before inventing the RBM. There are two kinds of variables in the RBM: the visible variables which are given as input variables and the hidden variables which are given for restoring the input. The stacked version of multiple RBMs which is called Belief Networks is a kind of deep neural networks. This chapter is intended to describe the RBM, together with the stacked version with respect to its learning process.

Chapter 12 is concerned with the convolutional neural networks as another type of deep neural networks. We studied the MLP which is expanded into the convolutional neural networks in Chap. 9. The pooling layers and the convolution layers are added as the feature extraction part to the MLP. There are the two parts in the architecture of the convolutional neural networks: the feature extraction which is the alternative layers of the pooling and the learning part which is the Perceptron or the MLP. This section is intended to describe the convolutional neural networks model which is expanded from the MLP with respect to the architecture and the classification process.

## Part IV: Textual Deep Learning

Part IV is concerned with the specialization of deep learning algorithms for text mining tasks. In the previous parts, we studied the modification of the machine learning algorithms into their deep versions and the deep neural networks which are applied to generic tasks. In this part, we study the process of mapping the textual data into another textual data as hidden values, and the application of the process to the modification into textual deep learning algorithms. We mention the text summarization which is intended to map an initial text into its essential one as well as to summarize it. This part is intended to describe the specialization and the application of the deep learning to text mining tasks.

Chapter 13 is concerned with the index expansion which is necessary for implementing the textual deep learning. The index expansion refers to the process of adding more words which are relevant to ones in an input text. In the index expansion process, an input text is indexed into a list of words, their associated words are retrieved from external sources, and they are added to the list of words. There are three groups of words in indexing a text: the expansion group which

contains very important words demanding their associated ones, the inclusion group which contains medium ones which are included as index, and the removal group which contains trivial ones which should be excluded from the index. This chapter is intended to describe the text indexing process and the index expansion.

Chapter 14 is concerned with the text summarization as an instance of text mining. It refers to the process of extracting automatically essential parts as the summary. In the process of the text summarization, a text is partitioned into paragraphs, and important ones among them are selected as its summary. The text summarization is viewed as mapping a text into a hidden text in implementing the textual deep learning. This section is intended to describe the text summarization with the view of implementing the textual deep learning.

Chapter 15 is concerned with the textual deep operations as mapping a text into another text. We studied the numerical deep operations as mapping an input vector into another vector which is called a hidden vector. The textual deep operations are manipulations on a text which map it into another text called hidden text before encoding it into a numerical vector. The role of the textual deep operations is to preprocess a text in implementing the textual deep learning. This chapter is intended to describe the textual pooling and the textual convolution as the main textual deep operations.

Chapter 16 is concerned with the text classification system which is implemented with a deep learning algorithm. Text classification as the function of the system which is implemented in this chapter is the process of classifying a text into a topic or some topics. We adopt the deep learning algorithm which is modified from an existing machine learning algorithm by adding the convolutional layer as the approach to the text classification. We mention the two kinds of convolutional layer: the numerical convolutional layer which is applied after encoding a text and the textual convolutional layer which is applied before doing it. This section is intended to describe the text classification system and the adopted deep learning algorithms.

Cheongju, South Korea

Taeho Jo

# Contents

## Part I Foundation

<b>1</b>	<b>Introduction.....</b>	<b>3</b>
1.1	Definition of Deep Learning.....	3
1.2	Swallow Learning.....	4
1.2.1	Supervised Learning.....	5
1.2.2	Unsupervised Learning.....	6
1.2.3	Semi-supervised Learning.....	8
1.2.4	Reinforcement Learning.....	10
1.3	Deep Supervised Learning.....	11
1.3.1	Input Encoding.....	12
1.3.2	Output Encoding.....	13
1.3.3	Unsupervised Layer.....	15
1.3.4	Convolution.....	17
1.4	Advanced Learning Types.....	19
1.4.1	Ensemble Learning.....	19
1.4.2	Local Learning.....	21
1.4.3	Kernel-Based Learning.....	23
1.4.4	Incremental Learning.....	25
1.5	Summary and Further Discussions.....	26
References.....	27	
<b>2</b>	<b>Supervised Learning.....</b>	<b>29</b>
2.1	Introduction.....	29
2.2	Simple Supervised Learning Algorithms.....	31
2.2.1	Rule-Based Approach.....	31
2.2.2	Naive Retrieval.....	32
2.2.3	Data Similarity.....	34
2.2.4	One Nearest Neighbor.....	35
2.3	Neural Networks.....	37
2.3.1	Artificial Neuron.....	37
2.3.2	Activation Functions.....	39

2.3.3	Neural Connection.....	40
2.3.4	Perceptron.....	42
2.4	Advanced Supervised Learning Algorithms.....	45
2.4.1	Naive Bayes.....	45
2.4.2	Decision Tree.....	47
2.4.3	Random Forest.....	49
2.4.4	Support Vector Machine.....	51
2.5	Summary and Further Discussions.....	54
	References.....	55
<b>3</b>	<b>Unsupervised Learning.....</b>	<b>57</b>
3.1	Introduction.....	57
3.2	Simple Unsupervised Learning Algorithms.....	58
3.2.1	AHC Algorithm.....	59
3.2.2	Divisive Algorithm.....	60
3.2.3	Online Linear Clustering Algorithm.....	62
3.2.4	K Means Algorithm.....	65
3.3	Kohonen Networks.....	67
3.3.1	Initial Version.....	67
3.3.2	Learning Vector Quantization.....	69
3.3.3	Semi-supervised Model.....	70
3.3.4	Self-Organizing Map.....	72
3.4	EM Algorithm.....	73
3.4.1	Cluster Distributions.....	73
3.4.2	Notations.....	75
3.4.3	E-Step.....	76
3.4.4	M-Step.....	78
3.5	Summary and Further Discussions.....	80
	Reference.....	81
<b>4</b>	<b>Ensemble Learning.....</b>	<b>83</b>
4.1	Introduction.....	83
4.2	Partition.....	85
4.2.1	Training Set.....	85
4.2.2	Attribute Set.....	86
4.2.3	Array Partition.....	87
4.2.4	Partition Schemes.....	90
4.3	Supervised Combination Schemes.....	91
4.3.1	Voting.....	92
4.3.2	Expert Gate.....	94
4.3.3	Cascading.....	96
4.3.4	Cellular Learning.....	99
4.4	Multiple Viewed Learning.....	101
4.4.1	Views.....	101
4.4.2	Multiple Encodings.....	103
4.4.3	Multiple Viewed Supervised Learning.....	104

4.4.4	Multiple Viewed Unsupervised Learning.....	106
4.5	Summary and Further Discussions.....	108
<b>Part II Deep Machine Learning</b>		
<b>5</b>	<b>Deep KNN Algorithm.....</b>	113
5.1	Introduction.....	113
5.2	Swallow Version.....	114
5.2.1	KNN Algorithm.....	115
5.2.2	KNN Variants.....	116
5.2.3	Trainable KNN Algorithm.....	118
5.2.4	Radius Nearest Neighbor.....	120
5.3	Basic Deep Versions.....	122
5.3.1	Feature Reduction.....	122
5.3.2	Kernel-Based KNN Algorithm.....	124
5.3.3	Output Decoded KNN.....	125
5.3.4	Pooled KNN.....	127
5.4	Advanced Deep Versions.....	129
5.4.1	Unsupervised Layer.....	129
5.4.2	Unsupervised KNN.....	131
5.4.3	Stacked KNN.....	133
5.4.4	Convolutional KNN Algorithm.....	135
5.5	Summary and Further Discussions.....	136
Reference.....		137
<b>6</b>	<b>Deep Probabilistic Learning.....</b>	139
6.1	Introduction.....	139
6.2	Swallow Version.....	141
6.2.1	Normal Distribution.....	141
6.2.2	Bayes Classifier.....	144
6.2.3	Naive Bayes.....	146
6.2.4	Bayesian Networks.....	148
6.3	Basic Deep Versions.....	149
6.3.1	Kernel-Based Bayes Classifier.....	149
6.3.2	Pooling-Based Bayes Classifier.....	151
6.3.3	Output Decoded Naive Bayes.....	153
6.3.4	Pooled Naive Bayes.....	155
6.4	Advanced Deep Versions.....	156
6.4.1	Unsupervised Bayes Classifier.....	156
6.4.2	Unsupervised Naive Bayes.....	158
6.4.3	Stacked Bayes Classifier.....	159
6.4.4	Stacked Bayes Classifier.....	161
6.5	Summary and Further Discussions.....	163
Reference.....		164

<b>7 Deep Decision Tree.....</b>	165
7.1 Introduction.....	165
7.2 Swallow Version.....	166
7.2.1 Graphical View.....	167
7.2.2 Classification Process.....	168
7.2.3 Root Node Selection.....	171
7.2.4 Learning Process.....	173
7.3 Basic Deep Versions.....	174
7.3.1 Random Forest.....	176
7.3.2 Clustering-Based Multiple Decision Trees.....	177
7.3.3 Output-Decoded Decision Tree.....	179
7.3.4 Pooled Naive Bayes.....	181
7.4 Advanced Deep Versions.....	183
7.4.1 Unsupervised Decision Tree.....	183
7.4.2 Stacked Decision Tree.....	185
7.4.3 Unsupervised Random Forest.....	187
7.4.4 Stacked Random Forest.....	189
7.5 Summary and Further Discussions.....	191
Reference.....	192
<b>8 Deep Linear Classifier.....</b>	193
8.1 Introduction.....	193
8.2 Support Vector Machine.....	194
8.2.1 Linear Classifier.....	195
8.2.2 Kernel Function.....	196
8.2.3 SVM Classifier.....	198
8.2.4 Dual Constraints.....	201
8.3 Basic Deep Versions.....	204
8.3.1 SVM as Deep Learning Algorithm.....	204
8.3.2 Multiple Kernel-Based SVM.....	206
8.3.3 SVM for Multiple Classification.....	209
8.3.4 Pooled SVM.....	211
8.4 Advanced Deep Versions.....	213
8.4.1 Unsupervised Linear Classifier.....	213
8.4.2 Stacked Linear Classifier.....	215
8.4.3 Unsupervised SVM.....	217
8.4.4 Stacked SVM.....	219
8.5 Summary and Further Discussions.....	221
<b>Part III Deep Neural Networks</b>	
<b>9 Multiple Layer Perceptron.....</b>	225
9.1 Introduction.....	225
9.2 Perceptron.....	226
9.2.1 Architecture.....	226
9.2.2 Classification Process.....	228

9.2.3	Learning Process.....	230
9.2.4	Perceptron for Regression.....	232
9.3	Multiple Layer Perceptrons.....	233
9.3.1	Architecture.....	233
9.3.2	Input Layer.....	235
9.3.3	Hidden Layer.....	236
9.3.4	Output Layer.....	237
9.4	Learning Process.....	238
9.4.1	Weight Update Between Hidden Layer and Output Layer.....	239
9.4.2	Weight Update Between Input Layer and Hidden Layer.....	240
9.4.3	Entire Learning Process.....	242
9.4.4	Stochastic Gradient Descent.....	243
9.5	Summary and Further Discussions.....	245
	Reference.....	246
<b>10</b>	<b>Recurrent Neural Networks.....</b>	<b>247</b>
10.1	Introduction.....	247
10.2	Recurrent Architecture.....	248
10.2.1	Forward Connection.....	248
10.2.2	Recurrent Connection.....	250
10.2.3	Hybrid Architecture.....	252
10.2.4	Hidden Recurrency.....	255
10.3	Recurrent Neural Networks.....	256
10.3.1	Basic Recurrent Neural Networks.....	257
10.3.2	RNN Variants.....	260
10.3.3	LSTM (Long Short-Term Memory).....	263
10.3.4	LSTM Variants.....	265
10.4	Applications.....	267
10.4.1	Time Series Prediction.....	268
10.4.2	Sentimental Analysis.....	269
10.4.3	Entire Learning Process.....	271
10.4.4	Machine Translation.....	272
10.5	Summary and Further Discussions.....	274
	Reference.....	275
<b>11</b>	<b>Restricted Boltzmann Machine.....</b>	<b>277</b>
11.1	Introduction.....	277
11.2	Associative Memory.....	278
11.2.1	Input Restoration.....	278
11.2.2	Associative MLP.....	280
11.2.3	Hopfield Networks.....	282
11.2.4	Boltzmann Machine.....	283
11.3	Single RBM.....	286
11.3.1	Architecture.....	286
11.3.2	Input Layer.....	288

11.3.3	Learning Process.....	289
11.3.4	Classification Model.....	291
11.4	Stacked RBM.....	293
11.4.1	Multiple Stacked RBM.....	293
11.4.2	Input Encoding.....	295
11.4.3	Output Decoding.....	298
11.4.4	Evolutionary RBM.....	300
11.5	Summary and Further Discussions.....	302
<b>12</b>	<b>Convolutional Neural Networks.....</b>	<b>303</b>
12.1	Introduction.....	303
12.2	Pooling.....	304
12.2.1	Pooling Concepts.....	304
12.2.2	Pooling Types.....	306
12.2.3	Dimensionality Downsizing.....	307
12.2.4	Pooling for Ensemble Learning.....	309
12.3	Convolution.....	310
12.3.1	Tensor.....	311
12.3.2	Single-Channeled Convolution.....	313
12.3.3	Tensor Convolution.....	315
12.3.4	Convolution Variants.....	316
12.4	CNN Design.....	318
12.4.1	ReLU.....	318
12.4.2	Pooling + ReLU.....	320
12.4.3	Convolution + ReLU.....	322
12.4.4	Pooling + Convolution + ReLU.....	323
12.5	Summary and Further Discussions.....	325
	Reference.....	326

## Part IV Textual Deep Learning

<b>13</b>	<b>Index Expansion.....</b>	<b>329</b>
13.1	Introduction.....	329
13.2	Text Indexing.....	330
13.2.1	Tokenization.....	330
13.2.2	Pooling Types.....	332
13.2.3	Stop Word Removal.....	334
13.2.4	Additional Filtering.....	337
13.3	Semantic Similarity.....	339
13.3.1	Word Representation.....	339
13.3.2	Cosine Similarity.....	341
13.3.3	Euclidean Distances.....	342
13.3.4	Table Similarity.....	344
13.4	Expansion Schemes.....	345
13.4.1	Associated Words.....	345
13.4.2	Associated Text.....	346

13.4.3	Information Retrieval-Based Scheme.....	348
13.4.4	Index Optimization.....	350
13.5	Summary and Further Discussions.....	353
	References.....	354
<b>14</b>	<b>Text Summarization.....</b>	<b>355</b>
14.1	Introduction.....	355
14.2	Abstracting.....	356
14.2.1	Phrase-Based Abstracting.....	356
14.2.2	Keyword-Based Abstracting.....	358
14.2.3	Mapping Abstracting into Binary Classification.....	360
14.2.4	Machine Learning-Based Abstracting.....	361
14.3	Query-Based Text Summarization.....	363
14.3.1	Query.....	364
14.3.2	Word-Based Summarization.....	365
14.3.3	Sentence-Based Summarization.....	366
14.3.4	ML-Based Text Summarization.....	368
14.4	Multiple Text Summarization.....	370
14.4.1	Group Cohesion.....	370
14.4.2	Keyword-Based Summarization.....	372
14.4.3	Machine Learning-Based Text Summarization.....	373
14.4.4	Textual Cluster Prototype.....	375
14.5	Summary and Further Discussions.....	376
	References.....	377
<b>15</b>	<b>Textual Deep Operations.....</b>	<b>379</b>
15.1	Introduction.....	379
15.2	Numerical Deep Operations.....	380
15.2.1	Text Encoding.....	380
15.2.2	Convolution.....	382
15.2.3	Pooling.....	384
15.2.4	Virtual Examples.....	385
15.3	Textual Convolution.....	387
15.3.1	Raw Text Structure.....	387
15.3.2	Random Part Selection.....	389
15.3.3	Hierarchical Indexing.....	390
15.3.4	Temporal Topic Analysis.....	392
15.4	Textual Pooling.....	394
15.4.1	Text Partition.....	394
15.4.2	Sub-dimensional Down-sampling.....	395
15.4.3	Keyword Extraction.....	397
15.4.4	Text Summarization.....	398
15.5	Summary and Further Discussions.....	399
	References.....	400

<b>16</b>	<b>Text Classification System.....</b>	403
16.1	Introduction.....	403
16.2	System Architecture.....	404
16.2.1	Input Layer.....	404
16.2.2	Convolution Layer.....	406
16.2.3	Pooling Layer.....	408
16.2.4	Design.....	409
16.3	Text Classification Process.....	410
16.3.1	Convolutional KNN.....	411
16.3.2	Convolutional Naive Bayes.....	413
16.3.3	Restricted Boltzmann Machine.....	414
16.3.4	Convolutional Neural Networks.....	416
16.4	Learning Process.....	417
16.4.1	Convolutional KNN.....	417
16.4.2	Convolutional Naive Bayes.....	419
16.4.3	Restricted Boltzmann Machine.....	420
16.4.4	Convolutional Neural Networks.....	422
16.5	Summary and Further Discussions.....	423
	Reference.....	424
	<b>Index.....</b>	425

# **Part I**

## **Foundation**

This part is concerned with the foundation for studying the deep learning algorithms. Because the deep learning is the area, which is expanded from the machine learning, we need to study the machine learning algorithms before doing the deep learning algorithms. There are four machine learning types: supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning, but we will cover the supervised learning and the unsupervised learning. The ensemble learning is viewed as the learning type where multiple machine learning algorithms are combined for solving their own demerits as the alternative advanced learning to the deep learning. This part is intended to review the supervised learning algorithms, the unsupervised ones, and the ensemble learning, for providing the foundation for understanding the deep learning.

This part consists of the four chapters. In Chap. 1, we will introduce the deep learning for providing its basic concepts. In Chap. 2, we will review the supervised learning algorithms as swallow learning algorithms. In Chap. 3, we will study the unsupervised learning algorithms which are applied to the data clustering. In Chap. 4, we will cover the ensemble learning as an advanced learning which is alternative to the deep learning.

# Chapter 1

## Introduction



This chapter is concerned with the overview of deep learning. Before discussing the deep learning, let us discuss the swallow learning, which is opposite to the deep learning, and in the swallow learning, the output value is computed directly from the input value. The deep learning is characterized as the process of computing intermediate values between input values and output values, and the input encoding, the output decoding, the convolution, and the unsupervised layer between the input layer and the output layer become the components for implementing the deep learning. There are other advanced learning types than the deep learning: the kernel-based learning, the ensemble learning, and the semi-supervised learning. This chapter is intended to describe the deep learning conceptually for providing the introduction.

This chapter is organized into the five sections, and we describe the deep learning conceptually in Sect. 1.1. In Sect. 1.2, we review the swallow learning as the traditional learning type. In Sect. 1.3, we describe the components which are needed for implementing the deep learning. In Sect. 1.4, we describe some advanced learning types other than the deep learning. In Sect. 1.5, we summarize the entire contents of this chapter and discuss further on what is studied in this chapter.

### 1.1 Definition of Deep Learning

The deep learning is defined as the learning type which builds its classification capacity, its regression model, or its clustering capacity through multiple steps of learning. In the traditional learning which is called swallow learning, the connection between the input and the output is direct. One hidden layer or multiple hidden layers exist between the input layer and the output layer in the deep learning. The unsupervised learning between the input layer and the hidden layer and between hidden layers proceeds, and the supervised learning between the hidden layer and

the output layer proceeds. This section is intended to describe the deep learning conceptually for providing its introduction.

Let us mention the swallow learning which is opposite to the deep learning for providing its background. The swallow learning refers to the traditional machine learning type which computes the output values directly from the input values. The process of the swallow learning is to prepare the training examples each of which is labeled with its own output and to optimize the parameters which indicates the direct relations between the input and the output for minimizing the error. In the generalization of the swallow learning, the output values are computed from the input values without computing intermediate values which are called hidden values. The swallow learning is characterized by the direct computation of the output values from the input values.

Let us mention the deep learning frame as the guide for modifying the existing machine learning algorithms into their deep versions. We may nominate the MLP which computes hidden values as a deep learning algorithm. The existing machine learning algorithms are modified by adding the module for computing hidden values. The CNN (convolutional neural networks) is the addition of the convolution layer and the pooling layer to the MLP. The process of computing hidden values corresponds to the unsupervised learning.

Let us mention the representative learning before proposing the deep learning. In the classical learning, the representations of the training examples are optimized manually, or a separated module is put for optimizing the representations with a heuristic scheme. The representative learning is the process of optimizing the representations as well as the parameters. The process of finding essential features automatically for reducing the dimensionality is a typical example of representative learning. It becomes the prerequisite learning for proposing the deep learning.

Let us mention what is intended in this chapter. We need to realize that the traditional machine learning algorithms belong to the swallow learning, by reviewing them. We provide conceptually the frame of deep learning for understanding its outline. We explore other types of advanced machine learning paradigms as alternatives to the deep learning. This chapter is intended to provide the introduction to the deep learning by studying its outline and its background.

## 1.2 Swallow Learning

This section is concerned with the swallow learning which is opposite to the deep learning, and we cover the four types of machine learning. In Sect. 1.2.1, we describe the supervised learning which is applied to the classification and the regression. In Sect. 1.2.2, we describe the unsupervised learning which is applied to the clustering. In Sect. 1.2.3, we describe the semi-supervised learning where two sets of training examples, a labeled set and an unlabeled set, are prepared. In Sect. 1.2.4, we describe the reinforcement learning which is applied to the autonomous moving.

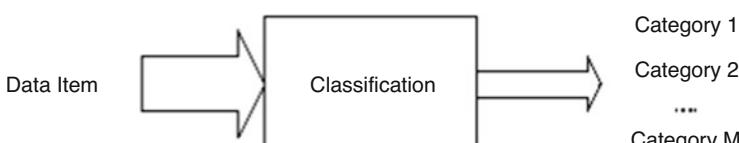
### 1.2.1 *Supervised Learning*

This section is concerned with the supervised learning as a machine learning type. Each of training examples is labeled with its own target output, and there are two kinds of output: target output which is initially assigned to each of training examples and computed output which is computed by the machine learning algorithm. The supervised learning is the learning type where the parameters are optimized for minimizing the error between the target output and the computed output. The supervised learning is applied to the classification which assigns a category to each item and the regression which estimates a continuous value to it. This section is intended to describe the supervised learning for providing the conceptual understanding of the supervised learning.

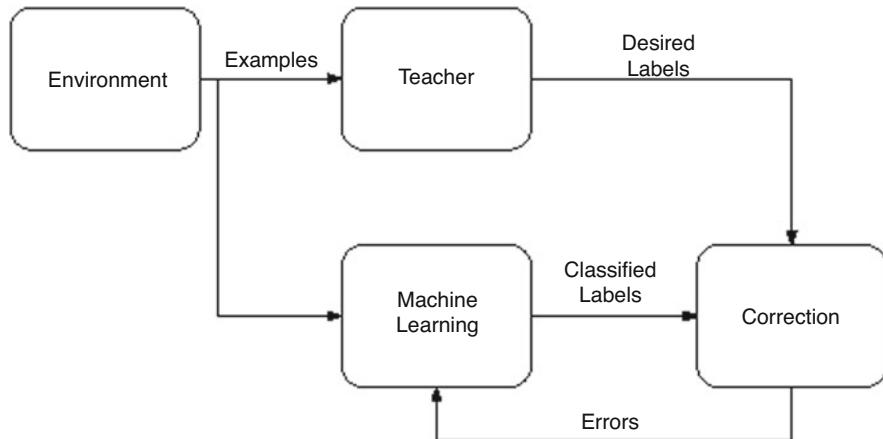
The data classification to which the supervised learning is applied is illustrated in Fig. 1.1. The given task is assumed as a multiple classification for explaining the supervised learning. The process of the data classification is to predefined categories into a list or a tree and classify a data item into one or some among them. The scope of the data classification is restricted to only hard classification where a data item is classified into only one category; the overlapping classification and the fuzzy classification are excluded in explaining the supervised learning. The regression as another task to which the supervised learning is applied is also excluded.

The frame of supervised learning is illustrated in Fig. 1.2. The training examples, each of which is labeled with its own output, are prepared; there are two kinds of labels for each training example: target label and computed label. The target label is generated by the teacher, and the computed label is generated by the machine learning algorithm. The parameters which are internal to the machine learning algorithm are optimized for matching the two kinds of labels, as many as possible. The fact that the target label is presented by the teacher is the reason of calling this machine learning paradigm supervised learning.

Let us mention some supervised learning algorithms which are covered in the next chapter. The KNN algorithm and its variants are the supervised learning algorithms which classify data items, depending on their similar training examples. The probabilistic learning algorithms, such as the Bayes classifier and the Naive Bayes, classify a novice item, depending on its likelihoods to the categories. The decision tree classifies a novice item following branches from the root node to a terminal node which is given as a label. The typical supervised learning algorithms are modified into their deep learning versions by attaching a module of computing hidden values.



**Fig. 1.1** Classification task



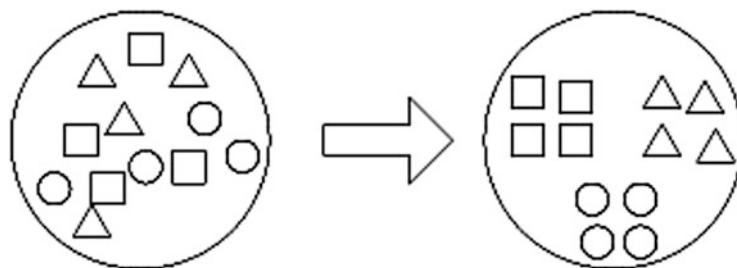
**Fig. 1.2** Supervised learning frame

Let us make some remarks on the supervised learning which is described as its frame. The data classification is the typical area to which the supervised learning is applied and the process of classifying a data item into one among the predefined categories. The supervised learning is the process of optimizing the internal parameters for minimizing the error between the target output and the computed output. The KNN algorithm, the Naive Bayes, the decision tree, and the linear classifier are typical supervised learning algorithms, and they are modified into the deep learning versions in Part II. The unsupervised learning will be considered as another type of machine learning in Chap. 3.

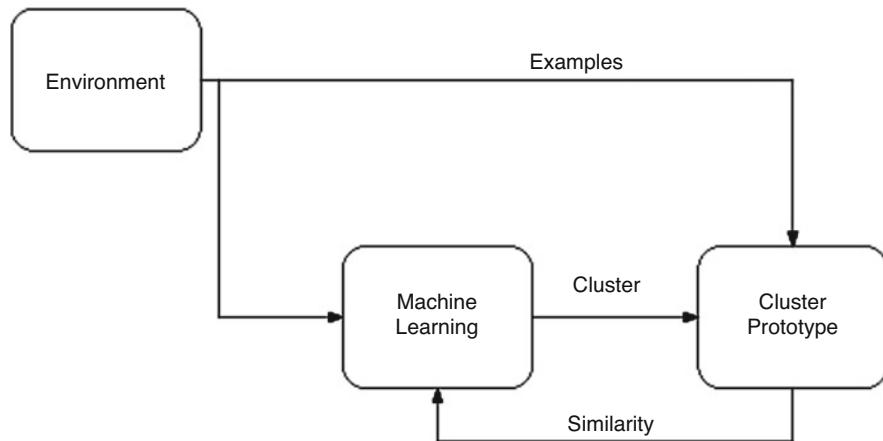
### 1.2.2 Unsupervised Learning

This section is concerned with the frame of unsupervised learning which is opposite to the supervised learning. In the previous section, we studied the frame of supervised learning for optimizing the parameters, depending on the target outputs which are presented by the teacher. The unsupervised learning is the machine learning paradigm which optimizes the parameters, depending on the similarities among the input vectors, and both the cohesion within each cluster and discriminations among clusters should be maximized as its direction. The data clustering is the process of segmenting a group of data items into subgroups each of which contains similar data items as the area to which the unsupervised learning is applied. This section is intended to describe the frame of unsupervised learning in the conceptual view.

The data clustering is visualized in Fig. 1.3. In the previous section, we studied the data classification to which the supervised learning is applied. The data clustering is the process of segmenting a group of data items into subgroups, each of which contains similar data items. The data clustering may be used as the technique



**Fig. 1.3** Clustering task



**Fig. 1.4** Unsupervised learning frame

of automating the category predefinition and the sample allocation. It is required to define the distances and the similarities among data items for implementing the data clustering.

The unsupervised learning frame is illustrated in Fig. 1.4. In this learning type, there is no teacher who provides the desired output; the fact characterizes the unsupervised learning. The parameters are usually given as the cluster prototypes and optimized as the unsupervised learning process. The similarity between an example and a cluster prototype is used as the optimization criteria, instead of the error between the desired output and the computed output. The unsupervised learning is applied for clustering data items.

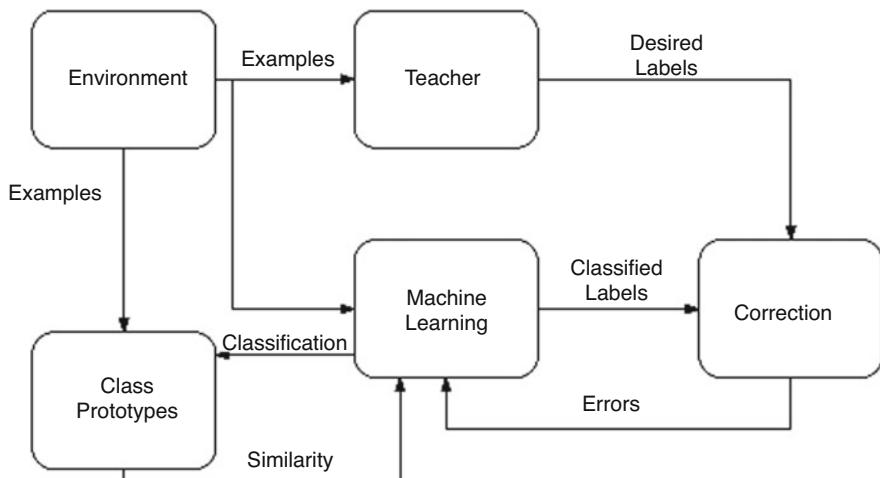
Let us mention some unsupervised learning algorithms as instances of supervised learning. The AHC algorithm, the divisive algorithm, and online linear clustering algorithm exist as simple clustering algorithms. The k means algorithm which clusters data items depending on the cluster mean vectors is the most popular unsupervised learning algorithm. The Kohonen Networks model is a typical neural networks model of unsupervised learning. The unsupervised learning algorithms will be studied in detail in Chap. 3.

Let us some remarks on the frame of unsupervised learning which is mentioned in this section. The data clustering to which the unsupervised learning is the process of segmenting a group of data items into subgroups depend on their similarities. In the unsupervised learning, unlabeled examples are prepared as the training examples, and the parameters which are given as the cluster prototypes are optimized for maximizing both the cohesion of each cluster and the discrimination among clusters. The k means algorithm and the Kohonen Networks are typical unsupervised learning algorithms. The semi-supervised learning which is the combination of the unsupervised learning and the supervised learning may be considered.

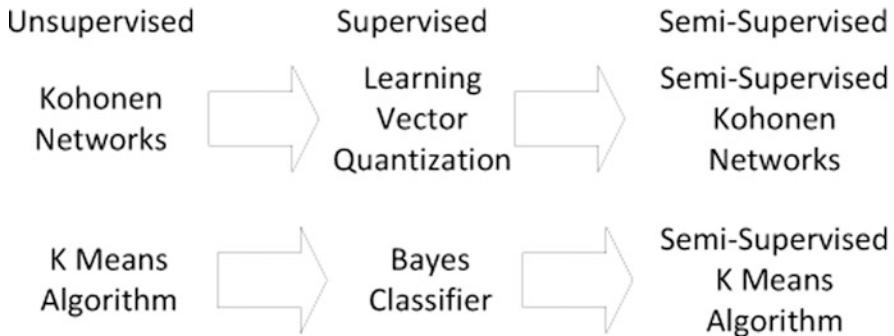
### 1.2.3 *Semi-supervised Learning*

This section is concerned with the third machine learning type which is called semi-supervised learning. It is intended to improve the generalization performance by utilizing unlabeled examples. Both the labeled examples and the unlabeled examples are prepared as the training examples, initial clusters are constructed by the labeled examples, and the initially unlabeled examples are labeled by clustering them. The supervised learning algorithm is trained with the originally labeled examples and the subsequently labeled examples for the tasks, such as classification and regression. This section is intended to describe the frame of the semi-supervised learning.

The frame of semi-supervised learning is illustrated in Fig. 1.5. The semi-supervised learning is viewed as the combination of the supervised learning and the unsupervised learning. The target label is presented from the labeled examples, and it depends on the similarities in processing the unlabeled examples. In the semi-



**Fig. 1.5** Semi-supervised learning frame



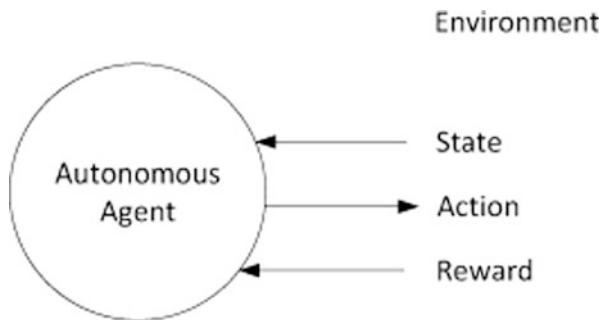
**Fig. 1.6** Transition of unsupervised learning into semi-supervised learning by means of supervised learning

supervised learning, the parameters are optimized for minimizing the error between the target output and the computed output and maximizing the cohesion in each cluster and the discrimination among the clusters. The supervised learning is the major part, in that the application area is the data classification or the regression.

The implementation of the semi-supervised learning algorithm by modifying an unsupervised learning algorithm is illustrated in Fig. 1.6. The k means algorithm and the Kohonen Networks are mentioned as typical unsupervised learning algorithms. They are modified into the supervised learning algorithms: Bayes classifier and learning vector quantization. They may be used as semi-supervised learning algorithms with the supervised learning to the labeled examples and the unsupervised learning to the unlabeled examples. Refer to [1] for getting the detail information about it.

We consider the case of implementing the semi-supervised learning by combining a supervised learning algorithm and an unsupervised learning algorithm. In the semi-supervised learning, both sets of training examples are prepared. The role of the unsupervised learning is to label the unlabeled example by clustering them, and the role of the supervised learning is to construct the classification or regression capacity by using both sets. The typical cases of implementing the semi-supervised learning are the combination of the k means algorithm and the KNN algorithm and the combination of the EM algorithm and the Naive Bayes [2]. The combination of two machine learning algorithms for implementing the semi-supervised learning should be distinguished from the ensemble learning which is covered in Chap. 4.

Let us make some remarks on the semi-supervised learning which is covered in this section. The semi-supervised learning is viewed as the combination of the supervised learning and the unsupervised learning. It is implemented by modifying an unsupervised learning algorithm into its semi-supervised version by means of its supervised version. Another scheme of implementing the semi-supervised learning is to combine a supervised learning algorithm and an unsupervised learning algorithm with each other. The semi-supervised learning is a supervised learning variant where unlabeled examples are added as additional training examples.



**Fig. 1.7** Autonomous agent and environment

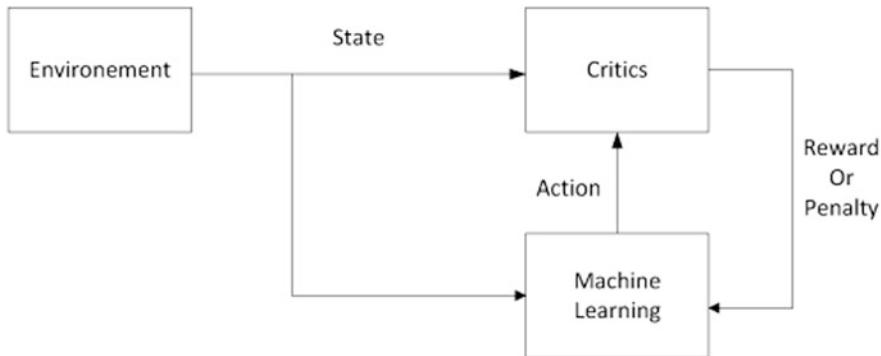
#### 1.2.4 Reinforcement Learning

This section is concerned with the reinforcement learning as the fourth machine learning type. In this learning type, the input is given as the state of environment, the output is given as an action, and the reward is given after taking the action. The reinforcement learning is the process of estimating rewards for each augmentation of a state and an action. It is required to define the states, the actions, and the relations of next state with the current state, and an action is required for implementing the reinforcement learning. This section is intended to describe the frame of the reinforcement learning.

The interaction between the autonomous agent which is the subject of the reinforcement learning and the environment is illustrated in Fig. 1.7. The agent receives a state from the environment; it recognizes the current state in the environment. The agent selects and takes an action as its output. The agent receives a reward or a penalty as the critics about the action. The reinforcement learning is aimed to maximize the reward to the action which is taken.

The frame of reinforcement learning is illustrated in Fig. 1.8. The teacher is replaced by the critics in the reinforcement learning. The state is given to both the critic and the machine learning algorithm, and the action is generated by the machine learning algorithm. A reward or a penalty is generated by the critic and is transferred to the machine learning algorithm as the feedback. The reinforcement learning is applied to the autonomous moving and the game play.

The table which is needed for implementing the reinforcement learning is illustrated in Fig. 1.9. The states from the environment, the actions from the autonomous agent, and delta rules, each of which is an association of the current state and an action with the next state, are defined. The goal states which users desire should be nominated among the defined states, and the rewards are initialized to the goal states with the complete value and the others with zero values. The reward is estimated for each augmentation of the current state and an action by the Q learning. The process of the reinforcement learning is to fill the table which is presented in Fig. 1.9 with the reward values.

**Fig. 1.8** Reinforcement learning frame

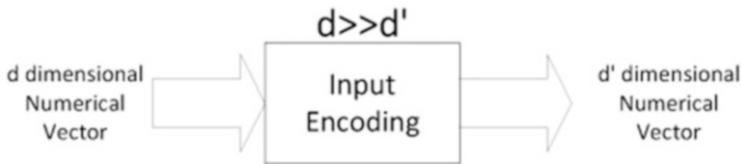
Current State	Action	Next State	Reward

**Fig. 1.9** Delta rule and Q table

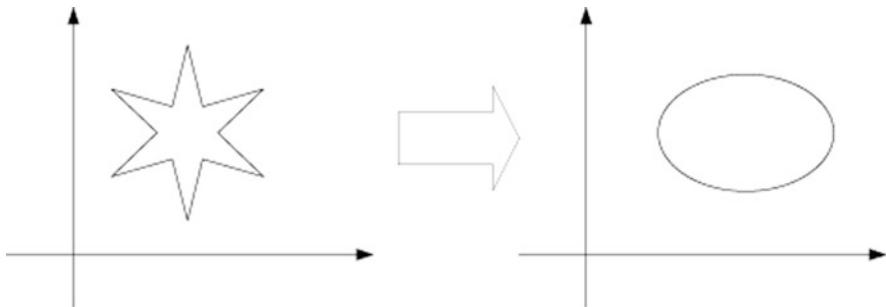
Let us make some remarks on the reinforcement learning which is described in this section. The autonomous agent which is the subject of the reinforcement learning receives the state from the environment and makes an action as its output. The teacher is replaced by the critics which provides a reward to the action in the frame of reinforcement learning. The reinforcement learning process is to initialize and estimate the reward value for each augmentation of a state and an action.

### 1.3 Deep Supervised Learning

This section is concerned with the components for satisfying the condition of deep supervised learning and organized with the four subsections. In Sect. 1.3.1, we mention the input encoding as the process of transforming an input into another input. In Sect. 1.3.2, we mention the output decoding as the process of generating the final output from another output. In Sect. 1.3.3, we mention the unsupervised layer which is followed by the supervised layer for implementing the deep learning. In Sect. 1.3.4, we mention the convolution as a linear transformation of a matrix given as the input into another matrix.



**Fig. 1.10** Dimension reduction



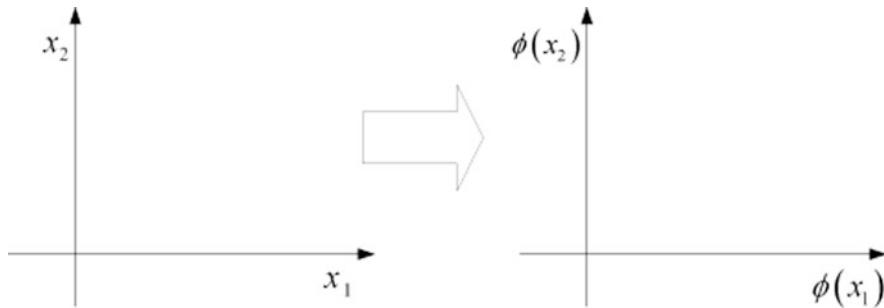
**Fig. 1.11** Outlier detection

### 1.3.1 *Input Encoding*

This section is concerned with the input encoding which is needed for implementing the deep learning. It refers to the process of mapping the input vector into another vector for computing easily the output vector. The dimension reduction, the outlier detection, and the feature mapping are mentioned as typical instances of input encoding. The input encoding is viewed as the process of computing a hidden vector from the input vector. This section is intended to describe the three instances of input encoding.

The process of reducing the input dimension is viewed as a module for implementing the deep learning as shown in Fig. 1.10. The dimension reduction is the process of mapping the input vector automatically into one with its smaller dimensionality. A typical scheme of reducing the dimensionality is to compute the correlations among attributes and remove ones which are correlated highly with others. The process of reducing the dimensionality is regarded as one of computing a hidden vector with its smaller dimensionality from the input vector. The dimension reduction plays the role of a component for implementing the deep learning.

The distributions over the data items before and after the outlier detection are illustrated in Fig. 1.11. The initial distribution over the data items is noisy, but it becomes smooth as the effect of the outlier detection. It refers to the process of detecting each data item as an outlier or not. The outlier detection is viewed as a binary classification of each data item into one of the two categories. The data items which are detected as outliers are removed from the training examples.



**Fig. 1.12** Feature mapping

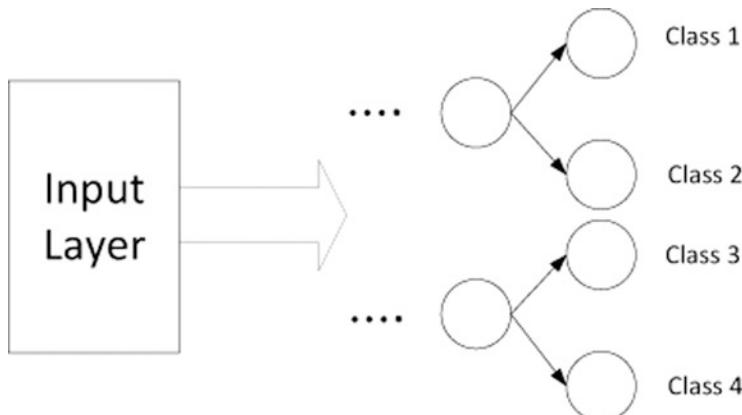
Mapping an input vector into one in another space is illustrated in Fig. 1.12. It is intended to transform the input space with its nonlinear separability into one with its linear separability. It is viewed as the process of computing a hidden vector from the input vector. The SVM which is covered in Chap. 8 is regarded as a deep learning algorithm by the feature mapping. The inner product of vectors which are mapped in another space is computed by a kernel function without mapping the original vectors.

Let us make some remarks on the input encodings which are used for implementing the deep learning. The dimension reduction is the process of encoding an input vector into one with its lower dimensionality. The outlier detection is the process of detecting an input vector as outlier or not. The feature mapping is the process of mapping an input vector into one in another space. The deep learning is implemented by attaching the input encoding module to the front of an existing machine learning algorithm.

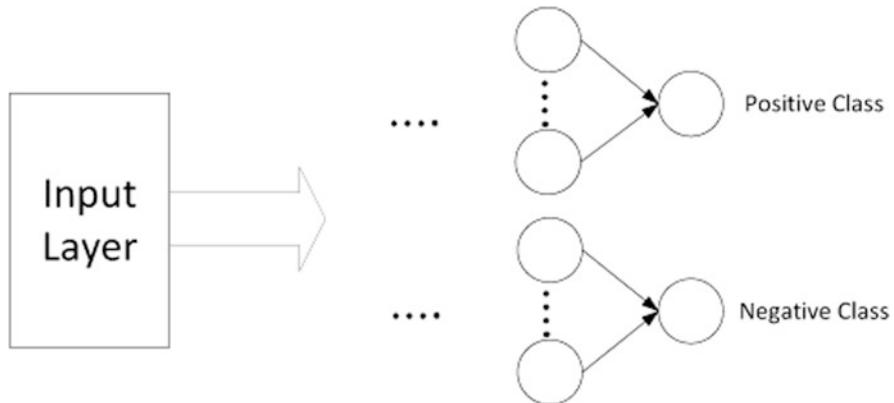
### 1.3.2 *Output Encoding*

This section is concerned with another module for implementing the deep learning, output decoding. In the previous section, we studied the input encoding which is the process of encrypting the input values into other values. The output decoding is the process of restoring the encrypted values into normal values as the output values, and the output specification, the output abstraction, and the output optimization are typical instances of output decoding. It is viewed as the process of computing the final output values from hidden values, in context of deep learning. This section is intended to describe the three instances of output decoding.

The output specification which is an output decoding instance is illustrated in Fig. 1.13. It is the process of classifying a data item which is labeled with a general category into its specific category. In Fig. 1.13, if the data item is classified into the upper category, it is classified into class 1 or class 2, and if it is classified into the



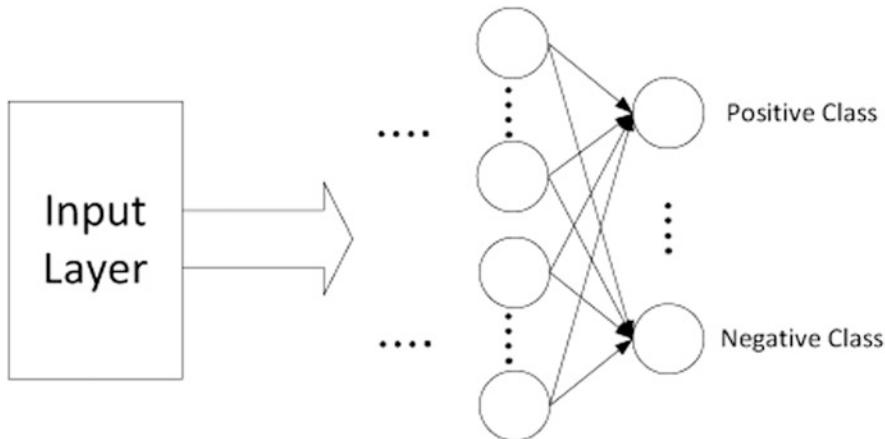
**Fig. 1.13** Output specification



**Fig. 1.14** Output abstracting

lower category, it is classified into class 3 or class 4. The output specification is viewed as the process of computing the output vector from the hidden vector with the lower dimensionality, in context of deep learning. This output decoding may be applied to the hierarchical classification.

The output abstraction as another output decoding instance is illustrated in Fig. 1.6. It is the process of classifying a data item which is labeled with a specific category into its more general category. In Fig. 1.6, if the data item is classified into one among the upper ones, it is classified into the positive class, and if it is classified into one among the lower ones, it is classified into the negative class. The output abstraction is viewed as the process of computing the output vector from a hidden vector with its higher dimensionality. We consider classifying a data item which is labeled with multiple specific categories into an abstract category (Fig. 1.14).



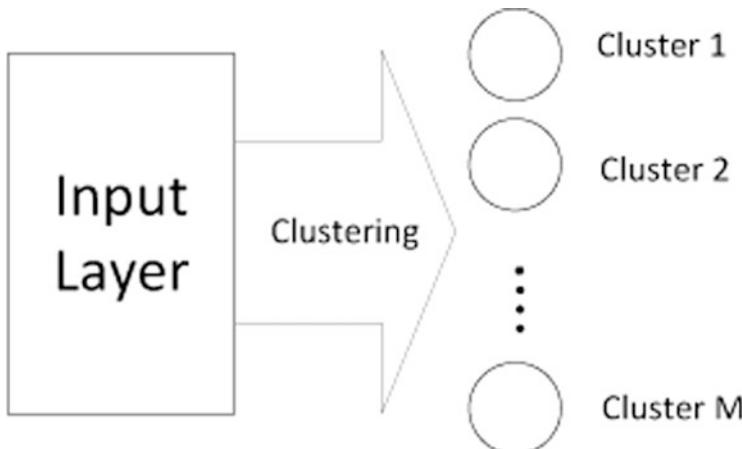
**Fig. 1.15** Output optimization

The third instance of output encoding called output optimization is illustrated in Fig. 1.15. The connection of the current layer with the previous layer is complete in the output optimization. All nodes in the previous layer are involved in classifying a data item into its final class. The output optimization is viewed as the process of computing the output vector from the hidden vector regardless of their dimensionality. In the output specification and the output abstraction, the connection from the previous layer is incomplete, whereas in the output optimization, the connection is complete.

Let us make some remarks on the output decoding which is a module for implementing the deep learning. The output specification is viewed as the process of computing the output vector from higher dimensionality from a hidden vector with its lower dimensionality. The output abstraction as the opposite to the output specification is the process of computing the output vector with its lower dimensionality from a hidden vector from its higher dimensionality. In the output optimization, it is assumed that the connection between the output layer and the hidden layer is complete. The process of computing the output vector from a hidden vector in the MLP is the case of the output optimization.

### 1.3.3 *Unsupervised Layer*

This section is concerned with the unsupervised layer which is attached for implementing the deep learning. In Sect. 1.2.2, we studied the unsupervised learning as the learning paradigm which is alternative to the supervised learning. A hidden vector is computed from the input vector by the unsupervised learning. If the swallow version of supervised learning is modified into the deep version, a



**Fig. 1.16** Clustering layer

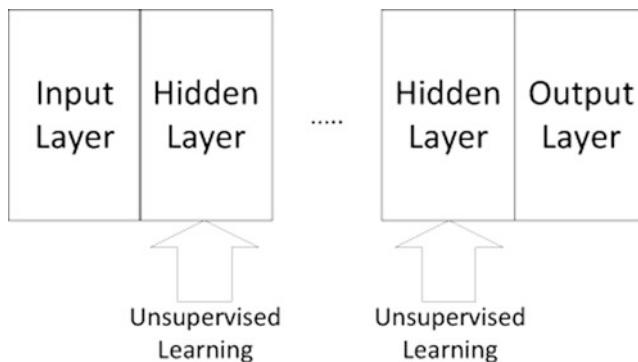
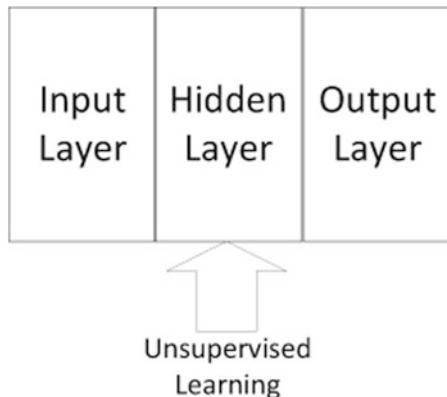
supervised learning algorithm should be modified into its own unsupervised version. This section is intended to describe the unsupervised layer for computing a hidden vector from the input vector.

The process of clustering data items is illustrated in Fig. 1.16. The data clustering is the process of segmenting a group of data items into subgroups each of which is called cluster and contains similar data items. The process of clustering data items is to define a similarity metric between data items and partition the group into subgroups, depending on their similarities. The role of clustering data items is to compute a hidden vector which consists of cluster memberships in the view of deep learning. The clustering module is used for implementing the deep learning.

The process of computing the hidden values in the unsupervised learning layer is illustrated in Fig. 1.17. The process of computing a hidden vector from the input vector is necessary for implementing the deep learning. The unsupervised learning is applied for computing a hidden vector as an alternative way to the input encoding and the output encoding. If existing machine learning algorithms are modified into their deep learning versions, we need to modify them into their unsupervised versions. We will study the application of the unsupervised learning to the computation of a hidden vector in Part II.

The application of the unsupervised learning to the implementation of the deep learning is illustrated in Fig. 1.18. The deep learning is composed with the input layer, multiple hidden layers, and the output layer. A hidden vector is computed from the input vector or one in the previous layer by the unsupervised learning. In each hidden layer, a single unsupervised learning algorithm is used for the fuzzy clustering, or multiple parallel unsupervised learning algorithms are used for multiple crisp clustering tasks. The fact that the hidden values are not provided as target values in the training set is the reason of applying the unsupervised learning to the computation of a hidden vector in the deep learning.

**Fig. 1.17** Application of unsupervised learning to hidden layer



**Fig. 1.18** Unsupervised learning in multiple hidden layers

Let us make on the unsupervised layer for computing the hidden vector. The unsupervised learning is applied to the data clustering which is the process of segmenting a group of data items into subgroups depending on their similarities. Because the target hidden vector is not available in deep learning, the unsupervised learning should be used for computing the hidden vector from the input vector. Each hidden layer is viewed as the unsupervised learning in implementing the deep learning. The process of computing the output vector from the hidden vector is regarded as the supervised learning in the supervised deep learning.

#### 1.3.4 Convolution

This section is concerned with the convolution which is needed for implementing the deep learning. The convolution is the operation on a vector or a matrix for filtering some components from it. In advance, filter vectors or filter matrices are



**Fig. 1.19** Convolutionary component

defined, and a source vector or a source matrix is mapped by summing the product of each element in the source and each element in the filter. A single source vector or matrix is mapped into multiple vectors or matrices in case of multiple filter vectors or matrices. This section is intended to describe the operation on a vector or a matrix, called convolution.

The frame of carrying out the convolution is presented in Fig. 1.19. It is assumed that the convolution is applied to the  $d$  dimensional vector. A  $f$  dimensional filter vector is prepared for carrying out the convolution; an arbitrary filter vector is decided with respect to its dimension and its elements. The output vector with its  $d - f + 1$  dimensions is filled with the values each of which is the summation of products of elements in the input vector and elements of in the filter vector. The  $f$  dimensional filter vector slides on the  $d$  dimensional vector during the computation of the output vector.

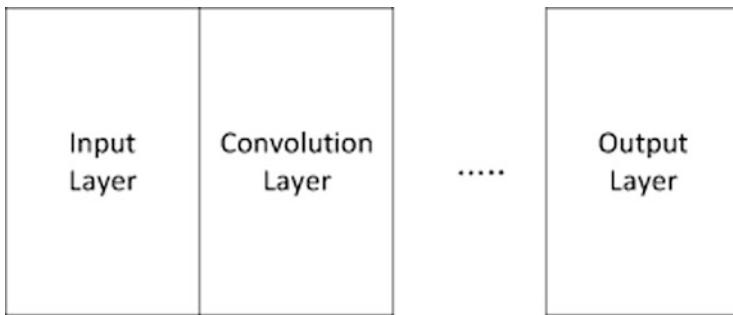
Let us mention the process of applying the convolution to a vector. Both the input vector and the filter vector are notated, respectively, by  $\mathbf{x} = [x_1 \ x_2 \ \dots \ x_d]$  and  $\mathbf{f} = [f_1 \ f_2 \ \dots \ f_f]$ . An element of the hidden vector,  $\mathbf{h}$ ,  $h_i$  is computed by Eq. (1.1):

$$h_i = \sum_{j=1}^f x_{i+j} f_j. \quad (1.1)$$

The hidden vector is a  $d - f + 1$  dimensional vector,  $\mathbf{h} = [h_1 \ h_2 \ \dots \ h_{d-f+1}]$ , as the results from the convolution. The filter vector is defined in advance, depending on the prior knowledge.

The application of the convolution layer to the implementation of the deep learning is illustrated in Fig. 1.20. The convolution is viewed as the process of computing a hidden vector, in context of the deep learning. The deep learning is implemented by adding the convolution layers between the input layer and the output layer. The CNN (convolutional neural networks) which is covered in Chap. 12 is a typical deep learning algorithm with the convolution layer. Multiple alternatives of the convolution layer and the pooling layer are used in the CNN.

Let us make some remarks on the convolution which is the component for implementing the deep learning. The convolution is the operation for filtering essential components from a vector by defining a filter vector. Each element in the mapped vector is computed by the filter vector which slides on the input vector and the part of the input vector. The convolution layer is used for modifying machine learning algorithms into their deep versions. We consider defining multiple filter vectors for performing the convolution with the multiple channels.



**Fig. 1.20** Convolutionary layer for deep learning

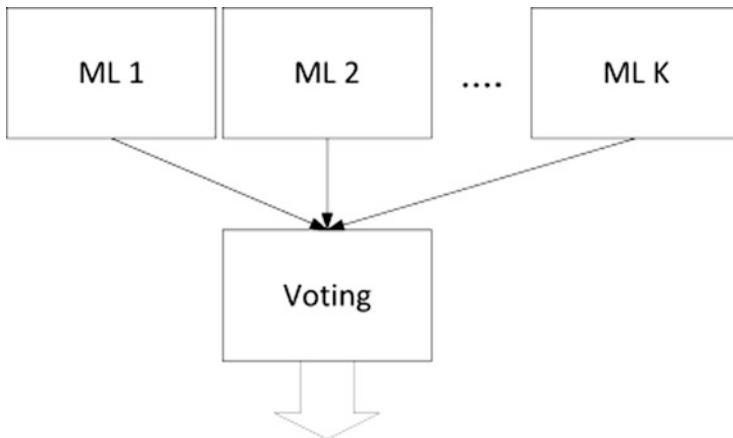
## 1.4 Advanced Learning Types

This section is concerned with the advanced learning types which are alternatives to the deep learning, and this section is organized with the four subsections. In Sect. 1.4.1, we mention the ensemble learning as the combination of multiple machine learning algorithms for improving the generalization performances. In Sect. 1.4.2, we mention the semi-supervised learning which utilizes unlabeled training examples, in addition to the labeled ones. In Sect. 1.4.3, we mention the kernel-based learning where input vectors are mapped into ones in another space. In Sect. 1.4.4, we mention the incremental learning which is the capacity of learning additional training examples.

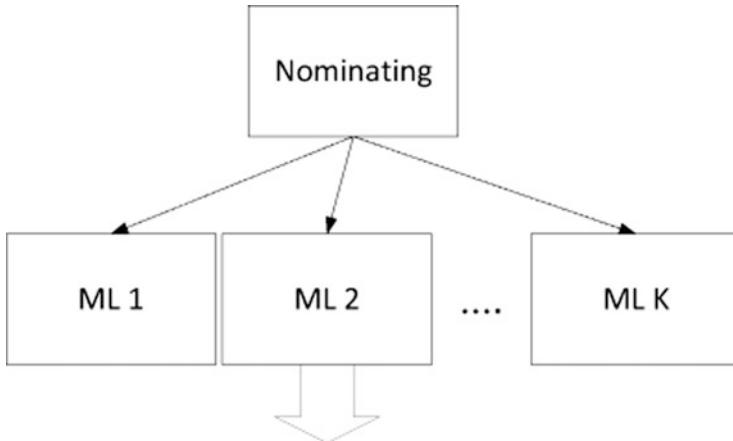
### 1.4.1 Ensemble Learning

This section is concerned with the ensemble learning as an advanced learning type. The ensemble learning is the scheme of involving multiple machine learning algorithms in the classification, the regression, or the clustering for improving their performances. In this section, we mention the three schemes of combining multiple machine learning algorithms with each other: voting, expert gate, and cascading. We may consider variants which are derived from each scheme and mixture of the three schemes. This section is intended to describe the three schemes of combining them.

The voting which is a scheme of combining multiple machine learning algorithms is illustrated in Fig. 1.21. It is to make final answer by voting answers from multiple machine learning algorithms. Each machine learning algorithm is trained with the set of training examples, and a novice item is classified by each machine learning algorithm. The answers which are made by them are collected, and the final answer is decided by voting them. In this ensemble learning scheme, multiple machine learning algorithms participate in making the final answer.



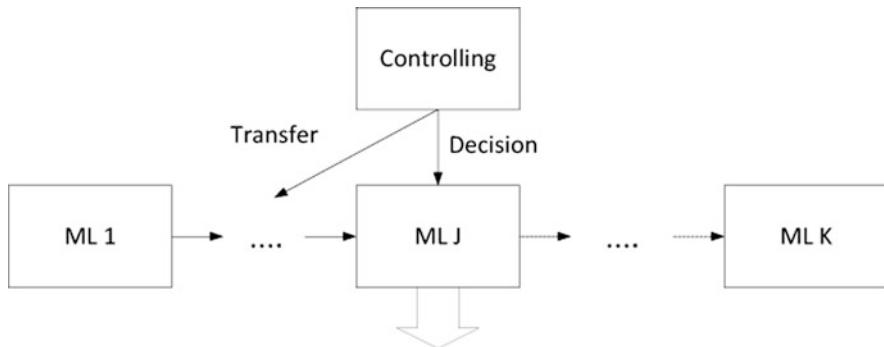
**Fig. 1.21** Voting of multiple machine learning algorithms



**Fig. 1.22** Expert gate of multiple machine learning algorithms

The expert gate as another scheme is illustrated in Fig. 1.22. In this scheme, the final answer is decided, depending on a single machine learning algorithm which is nominated. Each machine learning algorithm is trained with the training set, and particular machine learning is nominated to the novice input. It is classified by the nominated one, and the answer from the nominated is taken as the final one. Depending on the novice input, a different machine learning algorithm is nominated.

The cascading as one more scheme is illustrated in Fig. 1.23. In this scheme, the final answer is made by the answer which is made by the current machine learning algorithm depending on its certainty. Multiple machine learning algorithms are organized serially; the simplest one locates in the front, and the most advanced one locates in the tail in the organization. The answer from the current machine



**Fig. 1.23** Cascading of multiple machine learning algorithms

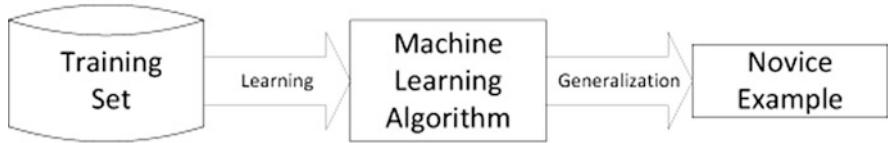
learning algorithm is taken, and whether the answer is the final answer, or the next machine learning algorithm is visited, is decided, depending on the certainty. In this scheme, some machine learning algorithms participate in making the final answer.

Let us make some remarks on the three schemes of combining multiple machine learning algorithms for implementing the ensemble learning. The voting is the combination scheme where the final decision is made by voting the answers from the machine learning algorithms. The expert gate is the combination scheme where the final answer is made by a particular machine learning algorithm which is nominated. The cascading is the combination scheme where the final answer is made by the current machine learning algorithm if its answer is certain. We will study the ensemble learning in detail in Chap. 4.

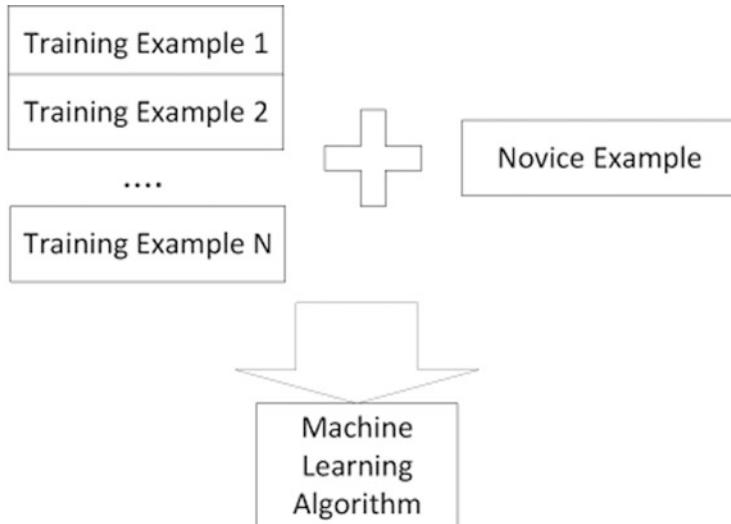
### 1.4.2 Local Learning

This section is concerned with another advanced learning type, called local learning. It is the learning type where the machine learning algorithm is trained with only training examples which are relevant to the novice input. The process of local learning is to retrieve some training examples which are relevant to the novice input and train the machine learning algorithm with them. The local learning is viewed as the lazy learning where the machine learning algorithm is not trained before the novice input is given. This section is intended to describe the local learning as an advanced learning type.

The global learning which is the traditional learning type is illustrated in Fig. 1.24. In the global learning, the training set is prepared, and the machine learning algorithm is trained with the entire set. A novice input is classified by the machine learning algorithm after learning all training examples. This learning type is adopted in the Perceptron, the MLP (multiple layer Perceptron), and the decision



**Fig. 1.24** Global learning

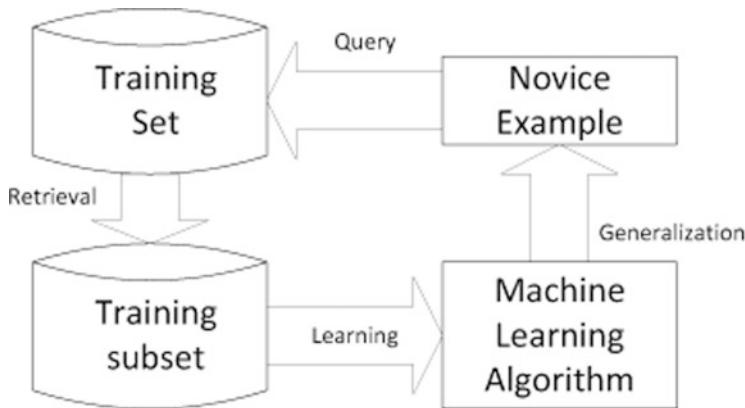


**Fig. 1.25** Instance-based learning

tree. The global learning is viewed as the eager learning which learns the training examples before its generalization.

The instance-based learning which is the basis for the local learning is illustrated in Fig. 1.25. Individual training examples are referred for classifying a novice item in this learning type. The training examples are never learned in advance, and they are accessed only in the generalization. This learning type is adopted in the KNN algorithm and the radius nearest neighbor algorithm. This learning type is viewed as the lazy learning which does not learn the training examples, until a novice example is presented.

The local learning which is opposite to the global learning is illustrated in Fig. 1.26. The local learning is the process of learning some training examples which are relevant to the novice input, instead of all training examples. In the local learning, the novice input is viewed as a query to the set of training examples, and its relevant training examples are retrieved. The machine learning algorithm learns the retrieved training examples and generalize the novice input. The local learning is implemented as the expert gate as the ensemble learning scheme for avoiding the lazy learning.



**Fig. 1.26** Local learning

Let us make some remarks on the instance-based learning and the local learning as the alternatives to the global learning. The global learning is the traditional learning type where all training examples are used for training the machine learning algorithm. The instance-based learning is the learning type where all training examples are referred for generalizing the novice input. The local learning is the learning type where some training examples which are relevant to the novice input are used for training the machine learning algorithm. The decision of scheme of computing the relevance of training examples to the novice input becomes the issue in the local learning.

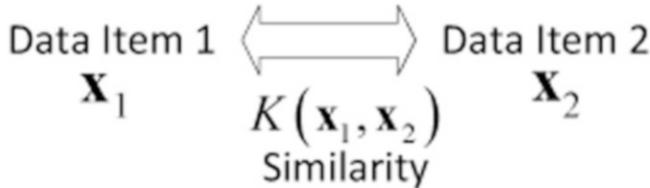
#### 1.4.3 Kernel-Based Learning

This section is concerned with one more advanced learning which is called kernel-based learning. It is the learning type where the kernel function is defined as the similarity between data items. Its idea is to improve separability between classes by mapping the input vectors in the original space into ones in another space. Through the kernel function, the similarity between two vectors in the mapped space is computed without mapping them directly. This section is intended to describe briefly the kernel-based learning as the advanced learning type.

The process of mapping the input vector into one in another space is illustrated in Fig. 1.27 as essence of the kernel-based learning. Its motivation is the fact that the overlapping between classes is a frequent case in the original space. If the input vector is mapped into another space, we get more chance of avoiding the overlapping and improving the separability. The original input vector and the mapped vector are notated, respectively, by  $\mathbf{x}$  and  $\Phi(\mathbf{x})$ . The inner product of the mapped vectors is given as a scalar value like that of the original vectors.



**Fig. 1.27** Feature mapping in kernel-based learning



**Fig. 1.28** Kernel function as similarity between mapped vectors

Let us mention the kernel function for computing the inner product between mapped vectors. The two vectors are notated by  $\mathbf{x}_1$  and  $\mathbf{x}_2$ , and their mapped vectors are notated by  $\Phi(\mathbf{x}_1)$  and  $\Phi(\mathbf{x}_2)$ . The inner product between the vectors,  $\Phi(\mathbf{x}_1)$  and  $\Phi(\mathbf{x}_2)$ , is expressed as the kernel function of the vectors,  $\mathbf{x}_1$  and  $\mathbf{x}_2$ , as shown in Eq. (1.2):

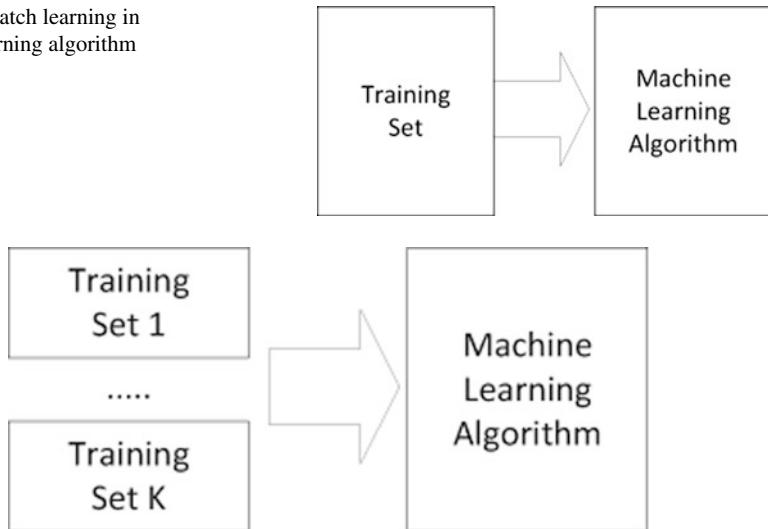
$$K(\mathbf{x}_1, \mathbf{x}_2) = \Phi(\mathbf{x}_1) \cdot \Phi(\mathbf{x}_2). \quad (1.2)$$

The basis of defining the kernel function of the two vectors is that the inner product of two vectors in any dimensionality is always given as a scalar value. The inner product between two mapped vectors is computed by the kernel function, avoiding mapping the input vectors individually.

The similarity between data items by the kernel function is illustrated in Fig. 1.30.  $\mathbf{x}_1$  and  $\mathbf{x}_2$  are the original vectors which represent data items, and  $\Phi(\mathbf{x}_1)$  and  $\Phi(\mathbf{x}_2)$  are vectors which are mapped from them. The inner product between  $\Phi(\mathbf{x}_1)$  and  $\Phi(\mathbf{x}_2)$  is expressed by the kernel function as the similarity between them. The kernel function is used for computing the similarity between data items in the SVM. It is not required to map the input vector into one in another space for computing the similarity between the mapped vector using the kernel function (Fig. 1.28).

Let us make some remarks on the kernel-based learning as a kind of advanced learning. A vector in a particular space is mapped into one in another space as the start of the kernel-based learning. The inner product between mapped vectors is computed using the kernel function, instead of mapping them explicitly. The inner product between mapped vector means the similarity between them. The kernel-based learning is applicable to any machine learning algorithm as well as the linear classifier.

**Fig. 1.29** Batch learning in machine learning algorithm



**Fig. 1.30** Interactive learning in machine learning algorithm

#### 1.4.4 Incremental Learning

This section is concerned with one more advanced learning which is called incremental learning. In studying machine learning algorithms, it is assumed that the generalization is continual once the machine learning algorithm is trained with the entire set of training examples. The idea of this incremental learning is to train continually the machine learning algorithm with additional training examples, whenever they become available. In the incremental learning, we expect the ability to learn additional training examples, keeping the generalization performance which is built by the previous examples. This section is intended to describe the incremental learning as an advanced learning type.

The batch learning which trains the machine learning algorithm with the entire set of training examples at a time is illustrated in Fig. 1.29. In the traditional machine learning algorithms, their learning style follow the batch learning. In the batch learning, the entire set of training examples is prepared, the machine learning algorithm is trained with the set, and novice input vectors which are given subsequently are generalized continually. Under the batch learning, it is assumed that all training examples are available, when the machine learning algorithm is trained. We need to utilize additional training examples which are given subsequently for improving the generalization performance.

The frame of increment learning is illustrated in Fig. 1.30. It is assumed that multiple sets of training examples are provided sequentially. The machine learning algorithm is trained with the first set of training examples, and the subsequent sets arrive at different times. The machine learning algorithm is trained incrementally to



**Fig. 1.31** Flow of data streams

each set. The goal of incremental learning is to avoid retraining entirely the previous sets.

The continual flow of data items which is called data stream is illustrated in Fig. 1.31. The case where training examples are provided interactively rather than at batch becomes the motivation for the incremental learning. The machine learning algorithm is trained continually with the training examples which are provided as a data stream. If more training examples are obtained, there is choice of waiting for additional training examples in the future or training the current machine learning algorithm with them instantly. We consider the case where the predefined categories are changed.

Let us make some remarks on the incremental learning which is mentioned as an advanced machine learning type. The machine learning algorithm is trained with the entire set of training examples in the batch learning. In the incremental learning, it is trained with an additional set of training examples interactively in the incremental learning. It is intended to utilize sets of training examples which is provided as a data stream in the reality. It is required to maintain the ability of classification or regression which is built by the previous sets of training examples for the incremental learning.

## 1.5 Summary and Further Discussions

Let us summarize what is studied in this chapter. There are four types of swallow learning: supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning. The frame of deep learning is to compute hidden values through the module, such as the input encoding, the output decoding, the pooling layer, and the unsupervised layer. The advanced learning types which is alternative to the deep learning are the ensemble learning, the local learning, the kernel-based learning, and the incremental learning. This section is intended to discuss further what is studied in this chapter.

Let us consider the ways of solving a nonlinear and complicated problem. Before inventing the deep learning, the advanced swallow learning was developed for solving it. The deep learning is the alternative way to the advanced swallow learning

where multiple layers are put between the input layer and the output layer. The big-sized decision tree are the results from constructing the decision tree from the training examples into its complicated form, whereas the MLP which is designed for solving a nonlinear problem by adding one more layer called hidden layer between the input layer and the output layer. By adopting the deep learning, rather than the advanced swallow learning, the better performance is expected.

The deep learning is viewed as the gradual processing with multiple steps. The swallow learning is regarded as the process of computing the output values from the input values, directly; it is viewed as the direct computation. In the deep learning, the output values are computed with multiple steps serially. In the ensemble learning as the alternative advanced machine learning, the output values are computed by multiple machine learning algorithms, parallelly. The deep learning is the trial of solving a complicated problem through the multiple stepped learning.

Let us mention the two dimensional array partition of the training set. It is viewed as a matrix where each row corresponds to a training example and each column corresponds to an attribute. The training is partitioned into subsets of training examples, and the attribute set is partitioned into subsets of attribute values. By crossing a subset of training examples and a subset of attributes, we obtain a subset of training examples with a subset of attributes. If the training set is partitioned into  $K$  subsets and the attribute set is partitioned into  $L$  subsets, the  $K \times L$  subsets are obtained.

The ensemble learning which is covered in Sect. 1.4.1 is expanded into the cellular learning. In the ensemble learning, a small number of machine learning algorithms is involved in improving the generalization performance. The idea of the cellular learning is to use a massive number of simple machine learning algorithms for solving nonlinear problems. Each machine learning algorithm is trained with its own subset which consists of a small number of training examples. The design of cellular learning is suitable to the parallel computing and the distributed computing.

## References

1. T. Jo, Machine Learning Foundation, Springer, 2021.
2. K. M. Nigam, K., A. K. McCallum, S. Thrun, and T. Mitchell, “Text classification from labeled and unlabeled documents using EM”, 103–134, Machine learning, 39(2), 2000.

# Chapter 2

## Supervised Learning



This chapter is concerned with the supervised learning as a kind of swallow learning. The supervised learning refers to the leaning type where its parameters are optimized for minimizing the error between the desired output and the computed one. In the supervised learning process, the training examples, each of which is labeled with its own target output, and the given learning algorithm are trained with them. Supervised learning algorithms are applied to classification and regression. This chapter is intended to review the supervised learning as a kind of swallow learning, before studying the deep learning.

This chapter is organized into the five sections, and in Sect. 2.1, we describe the supervised learning conceptually. In Sect. 2.2, we review some simple supervised learning algorithms. In Sect. 2.3, we describe the foundation of the neural networks and the simplest neural networks model, called Perceptron. In Sect. 2.4, we describe some typical supervised learning algorithms, such as the Naive Bayes, the decision tree, and the SVM (support vector machine). In Sect. 2.5, we summarize the entire contents of this chapter and discuss further on what is studied in this chapter.

### 2.1 Introduction

This section is concerned with the supervised learning in the conceptual view. All training examples are assumed to be labeled, and there are two kinds of output labels: the target labels which are initially assigned in gathering them and the computed labels into which they are classified. The supervised learning process is to optimize the parameter for minimizing the error which is the difference between the target labels and the computed labels. The supervised learning is applied to the classification which classified a data item into one or some among the predefined categories and the regression which estimates an output value or output values. This section is intended to describe the supervised learning for providing the introduction conceptually.

Let us mention the training set which is prepared for the supervised learning. In the supervised learning, the training set is viewed as a set of pairs of an input vector and an output vector as expressed in Eq. (2.1):

$$Tr = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \dots, (\mathbf{x}_N, \mathbf{y}_N)\} \quad (2.1)$$

In the multiple classification where the  $c$  categories are predefined, the input vector is given as a  $d$  dimensional numerical vector,  $\mathbf{x}_i = [x_{i1} \ x_{i2} \ \dots \ x_{id}]$ , and the output vector is given as a  $c$  dimensional numerical vector,  $\mathbf{y}_i = [y_{i1} \ y_{i2} \ \dots \ y_{id}]$ . There are two kinds of output vector in the supervised learning: the target output vector which is initially assigned to the input vector and the computed output vector which is computed by the machine learning algorithm during the training phase. The output vector is given as a scalar value in the univariate regression, while in the multivariate regression, it is given as a vector.

Let us mention the supervised learning frame in the conceptual view. The training examples each of which is labeled with its own output vector are prepared, and there are the two kinds of output: the target output vector and the computed output vector. The parameters which define the relation between the target output and the computed output are initialized at random, and the loss function which is proportional to the difference between the target output and the computed output is defined. The supervised learning process is to update the parameters for minimizing the loss function value gradually. The supervised learning algorithms are applied to the classification and the regression.

Let us mention some machine learning algorithms as instances of the supervised learning. The simplest supervised learning algorithm is the one nearest neighbor for classifying a data item by the label of its most similar training example. The neural networks are a kind of machine learning algorithm which optimizes the weights between the input layer and the output layer for minimizing the error on the training examples. The probabilistic learning is a kind of supervised learning which classifies a data item, depending on the likelihoods which are probabilities of the input vector-given categories. In this chapter, we will review some supervised learning algorithms as the basis for understanding the deep learning.

Let us mention what is intended in this chapter. We review some elementary supervised learning algorithms for warming up to study the supervised learning algorithms. We study the Perceptron which is the early neural networks model as the preparation for studying the MLP as the deep learning algorithm which is covered in Chap. 9. We review some main supervised learning algorithms, such as the Naive Bayes, the KNN algorithm, and the decision tree, as the targets which are modified into their deep versions. This chapter is intended to study the supervised machine learning algorithms as the swallow learning algorithms for studying the deep learning algorithms.

## 2.2 Simple Supervised Learning Algorithms

This section is concerned with the classification method before machine learning algorithms and simple machine learning algorithms. In Sect. 2.2.1, we mention the rule-based approach to the classification which was used before proposing the machine learning algorithm. In Sect. 2.2.2, we mention the naive retrieval as the start for studying the machine learning algorithm. In Sect. 2.2.3, we mention the similarity metric between two numerical vectors. In Sect. 2.2.4, we mention the one nearest neighbor as the most primitive machine learning algorithm.

### 2.2.1 Rule-Based Approach

This section is concerned with the rule-based approach to the data classification. This approach was used for classifying a data item before proposing the machine learning algorithm. In the rule-based approach, symbolic rules are defined manually and applied to the classification of each data item. Its high accuracy and its poor flexibility are results from applying the rule-based approach to the classification; when no symbolic rule is applicable, the novice item is not classified. This section is intended to describe the rule-based approach briefly as a scheme of classifying a data item.

A list of symbolic rules which are used for classifying a data item is illustrated in Fig. 2.1. In the logics, a proposition is a statement which is judged as true or false, and two propositions are connected by a condition. There are two parts in each symbolic rule: the conditional part which is the state followed by “if” and the causal part which indicates results from the conditional part. The symbolic rule which is used for classifying a data item is given as Eq. (2.2):

$$\text{if } a_i = v_i \text{ then } c_k \quad (2.2)$$

where  $a_i$  is an attribute,  $v_i$  is an attribute value, and  $c_k$  is a category. It is possible to make the conditional part with multiple propositions by connecting them by “or” and “and.”

The process of classifying a data item is illustrated as a pseudo code in Fig. 2.2. The list of symbolic rules and a novice example are arguments in the function. It

**Fig. 2.1** Classification rules

IF	$a_1 = v_1$	then	$c_1$
IF	$a_2 = v_2$	then	$c_2$
....			
IF	$a_4 = v_4$	then	$c_4$

**Fig. 2.2** Rule-based classification process

```

classifyByRuleList(List ruleList, Item example){
    for each value in example{
        for each rule in ruleList{
            if (rule.isMatch(value))
                return rule.getCategory();
        }
    }
    return 'unclassified';
}

```

searches for matching rule for each attribute value, and if it finds it, it returns the label. If there is no matching rule in exploring all attribute values, it returns the message, “unclassified.” The case of not classifying a data item by lack of matching rule is frequent.

Let us point out some issues in using the symbolic rules for classifying data items. The knowledge about the application area is required for defining symbolic rules manually. More symbolic rules should be added continually for improving the classification capacity. As symbolic rules are added continually, contradictions between rules happen with the higher probability. These issues in defining them manually become the motivation for replacing this approach by machine learning algorithms.

Let us make some remarks on the rule-based approach which is mentioned as the previous approach before using the machine learning algorithms. The symbolic rule which is applied to classify a data item consists of the conditional part which consists of an attribute value and the causal part which is given as a category. The classification algorithm searches for the symbolic rule which matches the attribute value in its conditional part in the process of classifying a data item. The poor flexibility becomes the motivation of replacing the rule-based approach by the machine learning algorithms. The symbolic rules are extracted from the training examples in the decision tree, implicitly.

### 2.2.2 Naive Retrieval

This section is concerned with the naive retrieval as the primitive classification scheme. This approach to the classification and the regression depends on individual training examples, as the initial instance-based learning. The naive retrieval is the scheme of classifying each data item by retrieving its exactly matching training example. Even if it is not practical, it becomes the start for understanding the instance-based learning algorithm, such as the KNN algorithm. This section is intended to describe the naive retrieve for warming up for studying the supervised learning algorithms.

Attribute 1	Attribute 2	....	Attribute d	Label

**Fig. 2.3** Training examples

```

Hashtable lookupTable; //Lookup Table given as a Hash Table
Category classifyExampleByLoopupTable(Item example){
    Category cc = lookupTable.findValue(example);
    return cc;
}

```

**Fig. 2.4** Naive retrieval

The training set is viewed as a table as shown in Fig. 2.3. The training set consists of the training examples as numerical vectors, and the given task is assumed as a classification. The training set is notated by a set of ordered pairs of input vectors and their target categories,  $Tr = \{(\mathbf{x}_1, c_1), (\mathbf{x}_2, c_2), \dots, (\mathbf{x}_N, c_N)\}$ , and each input vector,  $\mathbf{x}_i$ , is notated by  $\mathbf{x}_i = [x_{i,1} \ x_{i,2} \ \dots \ x_{i,d}]$ . The categories are predefined as a set,  $C = \{c_1, c_2, \dots, c_M\}$ , and the input vector,  $\mathbf{x}_i$ , is labeled by the category,  $c_i \in C$ . Each training example,  $(\mathbf{x}_i, c_i)$ , is viewed as a symbolic rule, if  $((a_1 = x_{i,1}) \wedge (a_2 = x_{i,2}) \wedge \dots \wedge (a_d = x_{i,d}))$  then  $c_i$ , in the naive retrieval.

The classification process in the naive retrieval is illustrated in Fig. 2.4. In advance, the training examples are prepared as a list. A novice item is given as the input and the algorithm searches for its exactly matching one among the training examples. The label of the novice item is decided by the label of the matching one. If there is no matching one, the novice item is rejected as an unclassified one.

Let us point out some issues in using the naive retrieval for classifying a data item. It is impossible to classify a data item with its noise. The case of mismatching between a novice item and all training examples is very frequent. It is possible to retrieve redundant training examples with their different labels as matching ones indicating the contradiction. The naive retrieve is interpreted into the rule-based classification which is mentioned in the previous section.

Let us make some remarks on the naive retrieval as the approach to the data classification. The training examples each of which is labeled with its own category are prepared. The classification process is to retrieve the training example which matches exactly with the novice input and decide the label of the novice input by the label of its matching one. The frequent rejection of a novice item as an unclassified one by its mismatching is an issue of the naive retrieval. It is improved into 1 nearest

neighbor where the similarities of a novice item with the training examples are computed.

### 2.2.3 Data Similarity

This section is concerned with the process of computing the similarity between data items. In the naive retrieval, we considered whether data items match with each other, or not. In this section, we measure how much data items match with each other, by defining a similarity metric between them. The inverse of Euclidean distance, the cosine similarity, and its variants are typical similarity metrics between data items. This section is intended to describe the similarity metric between data items.

The Euclidean distance is presented geometrically in Fig. 2.5. The Euclidean distance is one between two points in the Cartesian coordinate system as shown in Fig. 2.5. The Euclidean distance between two vectors,  $\mathbf{x}_1 = [x_{1,1}, x_{1,2}, \dots, x_{1,d}]$  and  $\mathbf{x}_2 = [x_{2,1}, x_{2,2}, \dots, x_{2,d}]$ , by Eq. (2.3):

$$ED(\mathbf{x}_1, \mathbf{x}_2) = \frac{1}{d} \sum_{i=1}^d \sqrt{(x_{1,i} - x_{2,i})^2}. \quad (2.3)$$

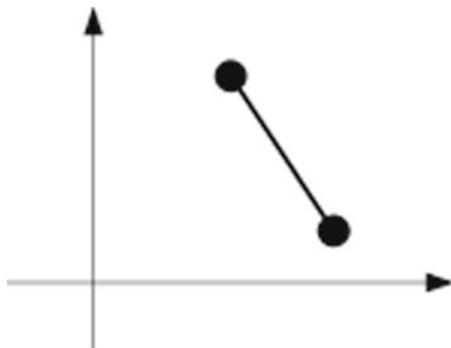
The similarity between the two vectors,  $\mathbf{x}_1$  and  $\mathbf{x}_2$ , is computed by reversing the distance as shown in Eq. (2.4):

$$sim(\mathbf{x}_1, \mathbf{x}_2) = \frac{1}{ED(\mathbf{x}_1, \mathbf{x}_2)}. \quad (2.4)$$

If the two vectors are the same to each other, the similarity between them is given as an infinite.

The cosine similarity is defined by Eq. (2.5),

**Fig. 2.5** Euclidean distance



$$sim(\mathbf{x}_1, \mathbf{x}_2) = \frac{\mathbf{x}_1 \cdot \mathbf{x}_2}{\|\mathbf{x}_1\| \cdot \|\mathbf{x}_2\|}. \quad (2.5)$$

The value of the cosine similarity is always given as a normalized value between zero and one, by  $\mathbf{x}_1 \cdot \mathbf{x}_2 \leq \|\mathbf{x}_1\| \cdot \|\mathbf{x}_2\|$ . The cosine similarity is defined as the more general form by Eq. (2.6):

$$sim(\mathbf{x}_1, \mathbf{x}_2) = \frac{\mathbf{x}_1 \cdot \mathbf{x}_2}{\|\mathbf{x}_1\|_p \cdot \|\mathbf{x}_2\|_p}, \quad (2.6)$$

where  $\|\mathbf{x}\|_p = \left( \sum_{i=1}^d x_i^p \right)^{\frac{1}{p}}$ .

Let us specify the cosine similarity which is mentioned above as the similarity between two numerical vectors. The cosine similarity is defined by Eq. (2.6) as the general form. When  $p = 2$ , it is defined as Eq. (2.5). When  $p = \infty$ , it is defined as Eq. (2.7):

$$sim(\mathbf{x}_1, \mathbf{x}_2) = \frac{\min(\mathbf{x}_1, \mathbf{x}_2)}{\max(\mathbf{x}_1, \mathbf{x}_2)}, \quad (2.7)$$

It was proposed that the similarities among the features should be considered in defining the cosine similarity in [2].

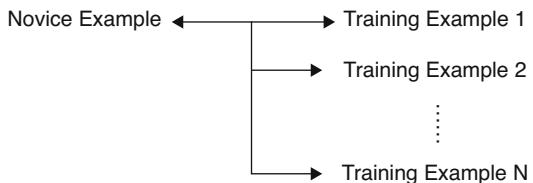
Let us make remarks on the cosine similarity which is used the most popular as a similarity metric between two numerical vectors. The Euclidean distance between two vectors is defined as the square root of the summation of one-to-one distance square between elements. The cosine similarity is the ratio of the inner product of two vectors to the product of two norms of two vectors. Some variants may be derived from the cosine similarity as similarity metrics. It is possible to consider similarities among attributes for defining a similarity metric between vectors [2].

## 2.2.4 One Nearest Neighbor

This section is concerned with the simplest supervised learning algorithm which is called one nearest neighbor. In the previous sections, we realized the limits of the naive retrieval as the approach to the data classification and defined the similarity metric between data items. The one nearest neighbor may be expanded into the KNN algorithm which decides the label of a novice item by voting ones of its nearest neighbors. This section is intended to describe the one nearest neighbor as a primitive machine learning algorithm.

The process of computing the similarities of a novice example with the training examples is illustrated in Fig. 2.6. A novice example is initially given as a classification target. Its similarities with the training examples are computed. The most similar training example is retrieved among them for decoding the category

**Fig. 2.6** Similarity computation in 1-NN



```
Category classifyBy1NN(item dataItem, List<trainingExample> trainingExampleList){
    int size = trainingExampleList.size();
    double maxSimilarity = 0;
    int maxIndex = 0;
    for(int i = 0; i < size; i++){
        Item trainExample = trainingExampleList.getElement(i);
        double similarity = dataItem.getSimilarity(trainExample);
        if(similarity > maxSimilarity){
            similarity = maxSimilarity;
            maxIndex = i;
        }
    }
    Item nearestNeighbor = trainingExampleList.getElement(maxIndex);
    return nearestNeighbor.getCategory();
}
```

**Fig. 2.7** Classification process by 1-NN

of the novice example as a nearest neighbor. The naive retrieval checks whether a novice example matches a training example, whereas the one nearest neighbor computes its similarities.

Let us mention the process of deciding the category of the novice example by a training example. The similarities of the novice example with the training examples are computed. The  $i$  training example is most similar as the novice example, and it is retrieved as its nearest neighbor. The category of the novice example is decided by taking the category of the  $i$ th training example. The advantage of the one nearest 204 neighbor over the naive retrieval is to avoid the rejection of a novice example.

The process of classifying a data item by the one nearest neighbor is illustrated as a pseudo code in Fig. 2.7. A data item and the list of training examples are given as the arguments. The similarities of a data item with the training examples are computed, and the training example with the maximal similarity is selected as the nearest neighbor. The category of the data item is decided by taking the category of the nearest neighbor. The linear complexity  $O(N)$  to the number of training examples,  $N$ , is taken for executing the one nearest neighbor as shown in Fig. 2.7.

Let us make some remarks on the one nearest neighbor which is covered in this section. The similarities of a novice item with the training examples are computed by the similarity metric which was mentioned in Sect. 2.2.3. The category of the novice item is decided by retrieving the most similar training example. The novice

item is classified with the two steps: computation of its similarities with the training examples and retrieval of the most similar training example. In the next section, the one nearest neighbor is expanded into the KNN algorithm.

## 2.3 Neural Networks

This section is concerned with the neural networks as the foundation for proposing the deep learning algorithms. In Sect. 2.3.1, we describe the artificial neuron as a computation unit in the artificial neural networks. In Sect. 2.3.2, we characterize the activation functions of the net input which is defined in an artificial neuron. In Sect. 2.3.3, we cover the connection of artificial neurons for building the neural networks. In Sect. 2.3.4, we study the Perceptron as the simplest neural networks.

### 2.3.1 Artificial Neuron

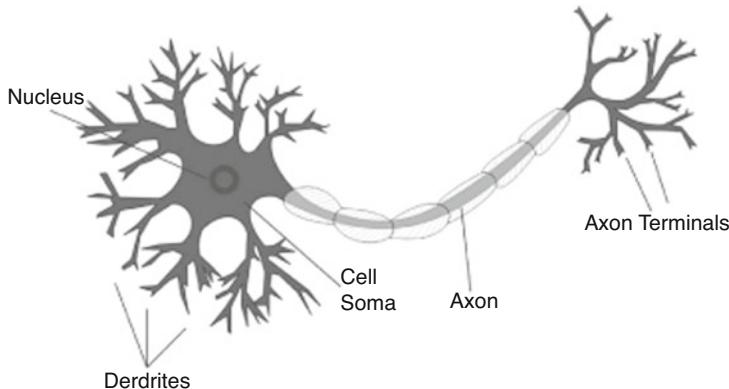
This section is concerned with the artificial neuron as a basic computation unit. The artificial neurons are connected as the neural networks which are studied in this section. The artificial neuron is a unit which receives multiple inputs and generate a single output. Multiple inputs are integrated into a single input by product of an input value and its own weight. This section is intended to describe the artificial neuron as a fundamental unit for studying the neural networks.

The artificial neuron which models the biological neuron is illustrated in Fig. 2.8. It is viewed as a computation unit with its input and its output. It receives multiple inputs with the discrimination among them and generates a single output. The output which is generated by an artificial neuron is transferred to another artificial neuron as one among its inputs. In the biological neural networks, the connection of neurons is irregular, whereas in the artificial neural networks, the connection is regular.

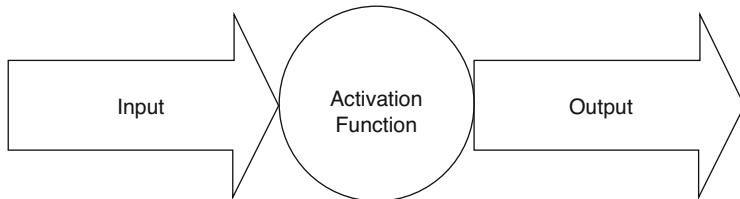
The artificial neuron which models the biological neuron is illustrated in Fig. 2.9. It is viewed as a computation unit with its input and its output. It receives multiple inputs with the discrimination among them and generates a single output. The output which is generated by an artificial neuron is transferred to another artificial neuron as one among its inputs. In the biological neural networks, the connection of neurons is irregular, whereas in the artificial neural networks, the connection is regular.

Computing the output value from multiple inputs in the artificial neuron is illustrated in Fig. 2.10. The inputs are notated by  $x_1, x_2, \dots, x_d$ , and the weights are notated by  $w_1, w_2, \dots, w_d$ . The net input is computed by summing the products of each input value and each weight, as shown in Eq. (2.8):

$$\text{Net\_Input} = \sum_{i=1}^d w_i x_i \quad (2.8)$$



**Fig. 2.8** Biological neuron from [https://subscription.packtpub.com/book/big\\_data\\_and\\_business\\_intelligence/9781787121393/5/ch05lv1sec38/the-biological-neuron](https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781787121393/5/ch05lv1sec38/the-biological-neuron)



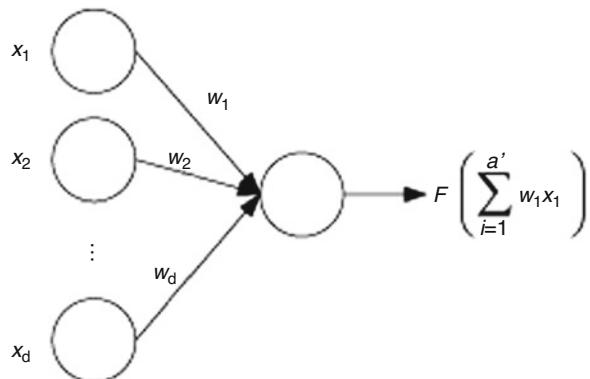
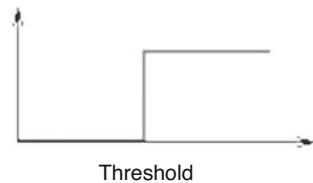
**Fig. 2.9** Single artificial neuron

The output is computed by applying an activation function to the net input as shown in Eq. (2.9):

$$y = F \left( \sum_{i=1}^d w_i x_i \right) \quad (2.9)$$

The activation functions which are applied to the net input will be mentioned in the next section.

Let us make some remarks on the artificial neuron which is mentioned as a computation unit in this section. The biological neuron receives the input signals from others through the dendrites and sends a single output signal through the axon. The biological neuron is modeled as the artificial neuron which receives multiple inputs from their discriminations and sends a single output. In Eqs. (2.8) and (2.9), the summation of products of inputs and weights is called net input, and the activation function is applied to the net input for generating the output. In the next section, we will study various activation functions which are applied to the net input.

**Fig. 2.10** Output equation**Fig. 2.11** Threshold function

### 2.3.2 Activation Functions

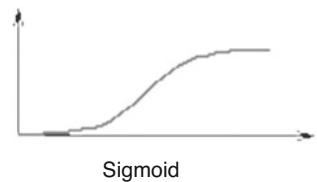
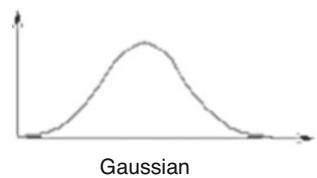
This section is concerned with the activation functions which are applied in the artificial neuron. In the previous section, we studied the artificial neuron as a computation unit which receives multiple inputs and sends a single output to another neuron. In this section, we study the activation functions which are applied to the net input within an artificial neuron. There is the case of sending the net input by itself as the output from the artificial neuron; the application which is applied to the net input is called linear function. This section is intended to describe some nonlinear activation functions which are applied to the net input.

The curve which indicates the threshold function is illustrated in Fig. 2.11. The value in the x-axis indicates a net input, and the value in the y-axis indicates the output value. The threshold function is expressed as Eq. (2.10):

$$y_j = \begin{cases} 1 & \text{if } \text{net} \geq 0 \\ 0 & \text{otherwise} \end{cases}. \quad (2.10)$$

the threshold and the constant are usually set respectively as zero and one. The output value of the threshold function in the y-axis is given as a binary value, 0 or 1. A real valued vector is mapped into a binary vector if the threshold function is applied to the net input as an activation function.

The curve of the sigmoid function is illustrated in Fig. 2.12. The domain of the sigmoid function is any value, and its range is a continuous value between zero and one. The function is expressed into Eq. (2.11):

**Fig. 2.12** Sigmoid function**Fig. 2.13** Gaussian function

$$y = \frac{1}{1 + e^{-x}} \quad (2.11)$$

The slope in the curve is stiff around zero. This function is adopted as the activation function in the MLP (multiple layer Perceptron).

The curve of the Gaussian function is illustrated in Fig. 2.13. The value in the x-axis is the norm of difference between the input vector,  $\mathbf{x}$ , and the center vector,  $\mathbf{c}$ . The Gaussian function is expressed by Eq. (2.12):

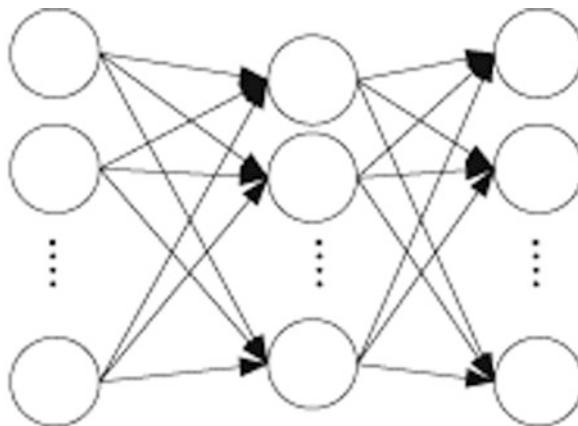
$$y = \exp\left(\frac{\|\mathbf{x} - \mathbf{c}\|}{2\sigma^2}\right) \quad (2.12)$$

where  $\sigma^2$  is the spread which is given as a parameter. When the input vector is consistent with the center, value in the function is maximal as shown in Eq. (2.12). The center vector,  $\mathbf{c}$ , and the spread,  $\sigma^2$ , are given as external parameters.

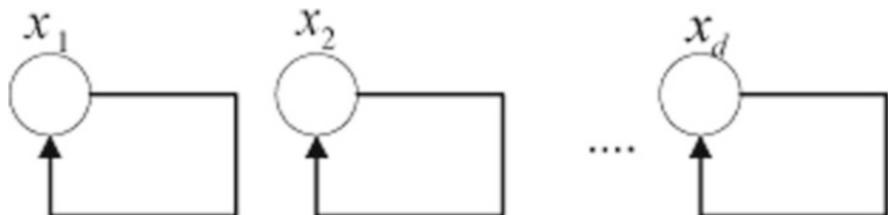
Let us make some remarks on some activation functions which are applied the net input in the artificial neuron. The output in the threshold function is given as a binary value, zero or one. The output in the sigmoid function is a continuous value between zero and one. In addition, the center vector and the spread are involved in the Gaussian function. The hyperbolic function whose output is a continuous value between -1 and 1 may be considered as an activation function.

### 2.3.3 Neural Connection

This section is concerned with the connection of artificial neurons as the basis for studying the neural networks. In Sect. 2.3.1, we studied the artificial neuron as a basic computation unit. Neurons are connected with the two ways: feedforward connection and recurrent connection. The output which is associated with its own



**Fig. 2.14** Feedforward connection



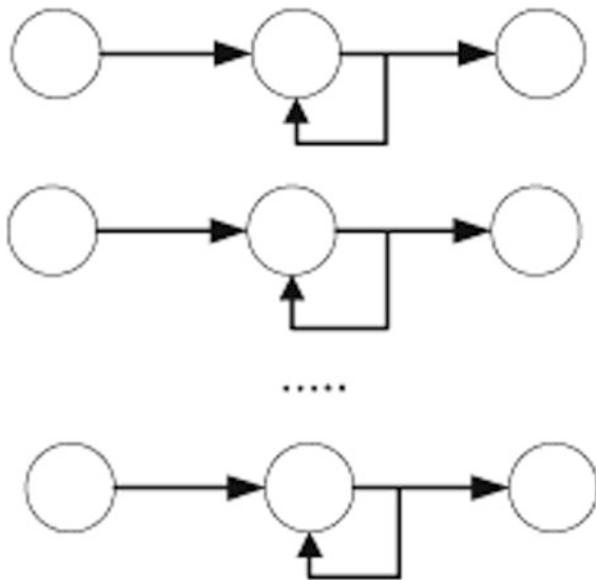
**Fig. 2.15** Recurrent connection

weight from the source neuron becomes the input to a destination neuron. This section is intended to describe the connection among neurons.

The feedforward connection among neurons is illustrated in Fig. 2.14. The three groups of neurons are given as the three layers. Each neuron in each layer is connected with all neurons in the next layer; the connection between layers is complete. This type of connection is adopted in implementing the Perceptron, the MLP (multiple layer Perceptron), and the Kohonen Networks. The Kohonen Networks will be studied as an unsupervised learning algorithm in Chap. 3.

The recurrent connection of each neuron is illustrated in Fig. 2.15. It is a circular connection from a neuron to itself. The previous output value of the neuron is given as the input by the feedback. This kind of connection is adopted for implementing the recurrent neural networks. This kind of connection is adopted for implementing the recurrent neural networks. We will study the recurrent neural networks as a deep learning algorithm in Chap. 10.

The hybrid connection among neurons is illustrated in Fig. 2.16. There are three groups of neurons; the connection from the left group to the right group is feedforward. The connection in the middle group is recurrent. Both the feedforward connection and the recurrent connection exist in this architecture of neurons. This



**Fig. 2.16** Hybrid connection

connection type is adopted for implementing the recurrent neural networks which is covered in Chap. 10.

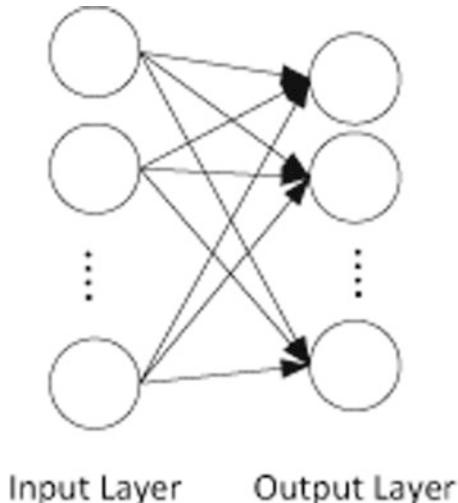
Let us make some remarks on the connection of neurons which is covered in this section. The feedforward connection is the linear connection from a neuron in a particular layer to neurons in its next layer. The recurrent connection which is used for implementing the recurrent neural networks which is covered in Chap. 10 is the circular connection from a neuron to itself. The hybrid connection is the mixture of the feedforward connection and the recurrent connection. The output of a neuron is the input of the next neuron in the connection among neurons.

### 2.3.4 Perceptron

This section is concerned with the simplest neural networks, called Perceptron. The neural networks were invented by Rosenblatt in the 1950s. Its architecture is composed with the input layer and the output layer, and the output value is computed by summing products of its input values and its weights. The weights which are given as a matrix between the input layer and the output layer are updated as the learning process. This section is intended to describe the Perceptron as a typical swallow learning algorithm.

The architecture of the Perceptron is illustrated in Fig. 2.17. It is composed with the input layer and the output layer. Each node in the input layer corresponds to

**Fig. 2.17** Perceptron architecture



an input value, and each node in the output layer corresponds to an output value. In applying the Perceptron to the multiple classification task, the number of input nodes is decided by the dimension of the input vector, and the number of output nodes is decided by the number of the predefined categories. The weights between the input layer and the output layer are involved as a matrix.

The direction of computing the output vector from the input layer to the output layer is illustrated in Fig. 2.18. The input vector as a  $d$  dimensional vector and the output vector as a  $c$  dimensional vector are notated, respectively, by  $\mathbf{x} = [x_1 \ x_2 \ \dots \ x_d]$  and  $\mathbf{y} = [y_1 \ y_2 \ \dots \ y_c]$ . The weights between the input layer and the output layer is expressed by  $c \times d$  matrix as shown in Eq. (2.13):

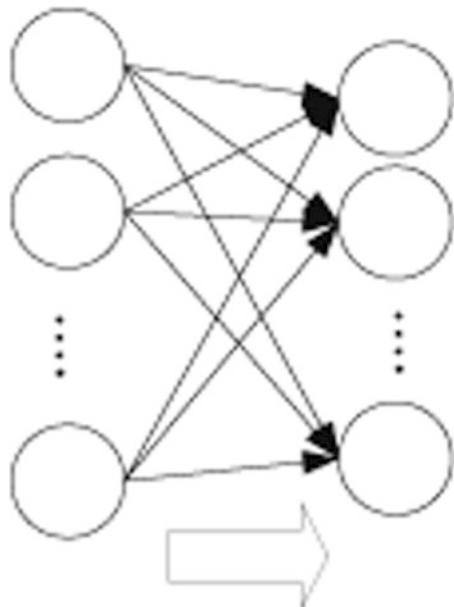
$$\mathbf{W} = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1d} \\ w_{21} & w_{22} & \dots & w_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ w_{c1} & w_{c2} & \dots & w_{cd} \end{pmatrix}. \quad (2.13)$$

Where, in the index of the weight,  $w_{ji}$ ,  $i$  is the index of an output node and  $j$  is the index of an input node. The element of the output vector,  $\hat{y}_j$ , is computed by Eq. (2.14):

$$\hat{y}_j = F \left( \sum_{i=1}^d w_{ji} x_i \right). \quad (2.14)$$

and the output vector,  $\hat{\mathbf{y}}$ , is expressed by the product of the weight matrix and the input vector as shown in Eq. (2.15):

**Fig. 2.18** Classification direction in perceptron



$$\hat{y} = F(\mathbf{W} \cdot \mathbf{x}). \quad (2.15)$$

The threshold function which is expressed in Eq. (2.10) is adopted as the activation function.

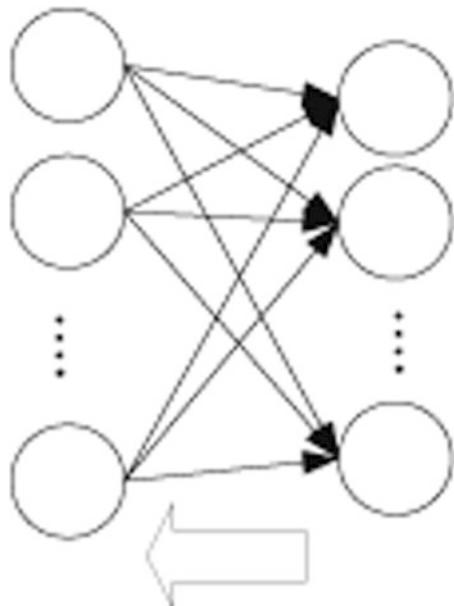
The direction of updating the weights from the input layer to the output layer is illustrated in Fig. 2.19. The training examples are prepared, the output value is computed by the above process for each training example, and it is assumed as a binary value which is the output of the threshold activation function. The weight between the input layer and the output layer are updated by Eq. (2.16):

$$w_{ji} \leftarrow w_{ji} + \eta x_i (y_j - \hat{y}_j), \quad (2.16)$$

where  $\eta$  is the learning rate which is given arbitrary between zero and one. There are two kinds of output values, target output,  $y_j$ , which is initially given to the input vector in the training example and computed output,  $\hat{y}_j$ , which is computed during the learning process, and if the target output is identical to the computed output, the weight is not updated. The learning process of the Perceptron is to iterate updating the weights by Eq. (2.16).

Let us make some remarks on the Perceptron as the early neural networks. There are two layers in the architecture: the input layer and the output layer. The output value is computed by applying the threshold function to the net input. The weights between the input layer and the output layer are updated on the difference between the target output and the computed output. The Perceptron is expanded into the MLP (multiple layer Perceptron) by adding one more layer called hidden layer.

**Fig. 2.19** Learning direction in perceptron

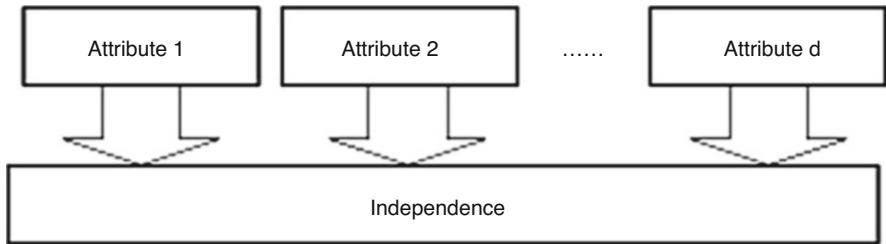


## 2.4 Advanced Supervised Learning Algorithms

This section is concerned with the supervised learning algorithms which are used popularly. In Sect. 2.4.1, we describe the Naive Bayes which assumes the independences among attributes. In Sect. 2.4.2, we describe the decision tree where symbolic logics may be traced for providing evidence. In Sect. 2.4.3, we describe the random forest as the expansion of the decision tree. In Sect. 2.4.4, we study the SVM (support vector machine) which defines dual parallel hyperplanes as its classification boundary.

### 2.4.1 Naive Bayes

This section is concerned with the Naive Bayes as the most popular probabilistic learning algorithm. The assumption underlying in this machine learning algorithm is that the attributes are independent of each other, even if they are dependent on each other in reality. In the Naive Bayes, the likelihoods of individual attribute values to each category are computed. The likelihood of a novice input vector is computed by product of individual attribute values, based on the assumption. This section is intended to describe the Naive Bayes as the most popular machine learning algorithm, briefly.



**Fig. 2.20** Attribute independence assumption

The relation among the attributes which is assumed in the Naive Bayes is illustrated in Fig. 2.20. The independence among them is assumed in computing the category likelihoods of the input vector. If the events,  $A$  and  $B$ , are independent of each other, the probability of both events,  $P(AB)$ , is computed by Eq. (2.17):

$$P(AB) = P(A)P(B). \quad (2.17)$$

If the  $d$  attributes are independent of each other, the probability,  $P(\mathbf{x})$ , is expressed as the product of probabilities of its elements by Eq. (2.18):

$$P(\mathbf{x}) = \prod_{i=1}^d dP(x_i). \quad (2.18)$$

The conditional probability of the event,  $A$ , given the event,  $B$ ,  $P(A|B)$  is the same to the probability of  $A$ , as shown in Eq. (2.19):

$$P(A) = P(A|B). \quad (2.19)$$

Let us mention the process of computing the likelihood of an attribute value to a category. The likelihood of a category is the conditional probability of entity given it. The attribute value and the category are notated, respectively, by  $x_i$  and  $c$ , and the likelihood of the attribute,  $x_i$ , to the category,  $c$ , is notated as the conditional probability,  $P(x_i|c)$ . The conditional probability,  $P(x_i|c)$ , is computed by Eq. (2.20):

$$P(x_i|c) = \frac{\#(x_i \wedge c)}{\#(c)}, \quad (2.20)$$

where  $\#(x_i \wedge c)$  is the number of training examples which belongs to the category,  $c$ , and have the attribute,  $x_i$ , and  $\#(c)$  is the number of training examples which belongs to the category,  $c$ . If the novice input is a  $d$  dimensional vector, the likelihoods of its elements are computed.

Let us mention the process of classifying a novice input vector by the Naive Bayes. By the above process, the likelihoods of the attributes,  $P(x_1|c)$ ,  $P(x_2|c), \dots, P(x_d|c)$ , to the category,  $c$ , are computed. The likelihood of the input vector,  $\mathbf{x}$ , is computed by the product of them, as shown in Eq. (2.21):

$$P(\mathbf{x}|c) = \prod_{i=1}^d P(x_i|c). \quad (2.21)$$

The likelihoods of the input vector to the predefined categories,  $P(\mathbf{x}|c_1)$ ,  $P(\mathbf{x}|c_2)$ ,  $\dots, P(\mathbf{x}|c_M)$ , are computed, and the input vector is classified into the category,  $c_{max}$  as shown in Eq. (2.22):

$$c_{max} = \operatorname{argmax}_{i=1}^M P(\mathbf{x}|c_i). \quad (2.22)$$

In the Naive Bayes, the novice item is classified depending on the maximum likelihood.

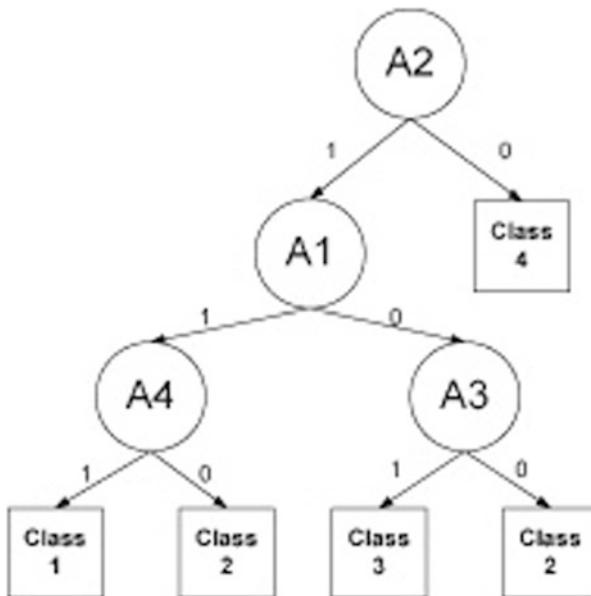
Let us make some remarks on the Naive Bayes which is a typical supervised learning algorithm. Even if the attributes are dependent on each other in the reality, the independence among them is assumed in this supervised learning algorithm. The likelihood of each attribute value to the category is computed as its conditional probability. The likelihood of the input vector to the category is computed by product of the likelihoods of its elements. We will study in detail this machine learning algorithm, together with its deep versions in Chap. 6.

### 2.4.2 Decision Tree

This section is concerned with the decision tree as another typical supervised learning algorithm. In the decision tree structure, the non-leaf nodes are attributes, the leaf nodes are categories, and the branches are attribute values. A data item is classified by following branches, each of which corresponds to its own attribute value from the root node to a leaf node. The learning process of decision tree is to construct the decision tree from the training examples. This section is intended to describe the decision tree briefly as a typical machine learning algorithm.

The decision tree structure is illustrated in Fig. 2.21. The leaf nodes indicate the categories, and the non-leaf nodes indicate the attributes. The branches from each non-leaf node are attribute values. The process of classifying a data item is to follow branches from the root node to a leaf node. The number of child nodes is identical to the number of values in the attribute which corresponds to the parent node.

The process of classifying a data item by the decision tree is visualized in Fig. 2.22. In the decision tree, a data item is classified in the top-down direction

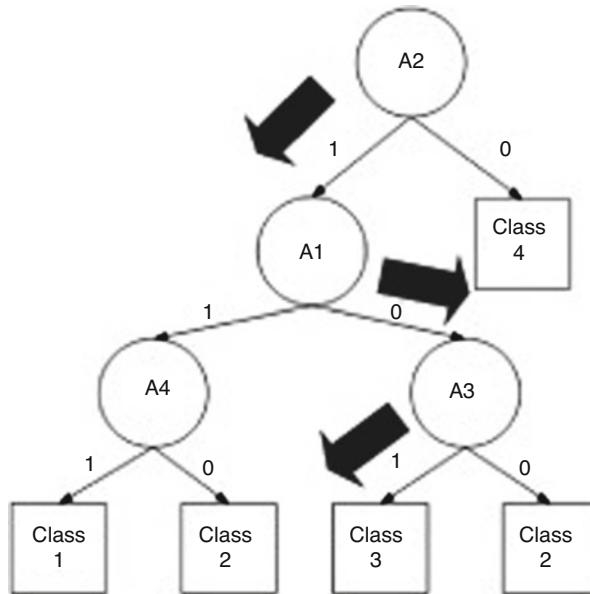
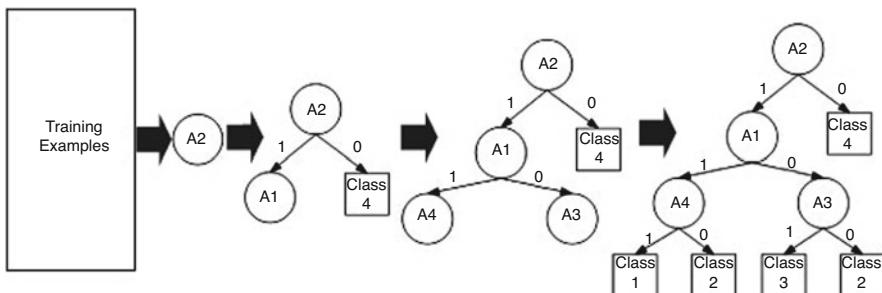


**Fig. 2.21** Structure of decision tree

from the root node to a leaf node. The branches from the root node to a leaf node is the list of attribute values for proving the classification. As shown in Fig. 2.22,  $A2 = 1$ ,  $A1 = 0$ , and  $A3 = 1$  become the evidence of classifying a data item into class 4. The merit of the decision tree is that it presents the evidence for classifying a data item.

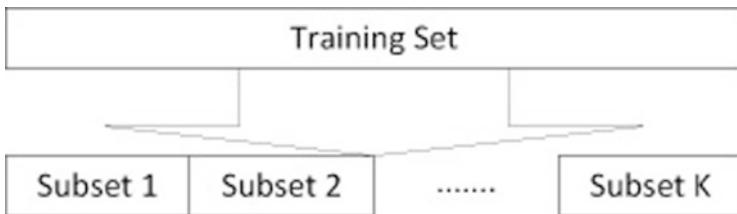
The process of constructing the decision tree from the training examples is illustrated in Fig. 2.23. The learning process in the decision tree is defined as the construction of the decision tree from the training examples. The attribute is selected as the root node as start of constructing the decision tree. The subset of training examples with the attribute which corresponds to the branch from the parent node is treated for selecting the attribute as the current node. If the remaining ones are labeled identically, the labeled category is decided as the leaf node.

Let us make some remarks on the decision tree as a typical supervised learning algorithm. In the decision tree structure, its non-leaf nodes are attributes, its leaf nodes are categories, and its branches are attribute values. In the decision tree, a data item is classified by following the branches from the root node to a leaf node. The decision tree is constructed from the training examples as its learning process. The decision of criteria for selecting an attribute as a node is an issue in training the decision tree.

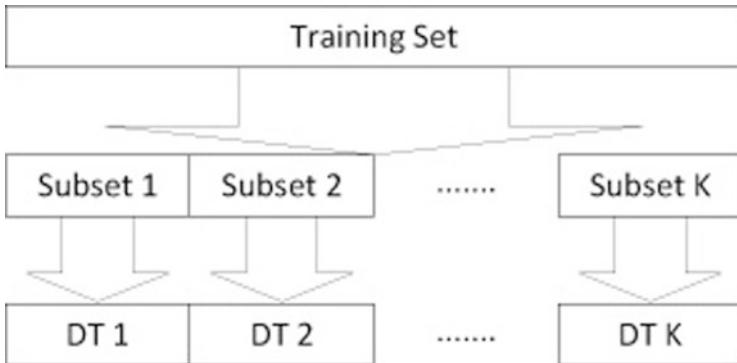
**Fig. 2.22** Classification direction in decision tree**Fig. 2.23** Learning direction in decision tree

### 2.4.3 Random Forest

This section is concerned with the random forest as a group of decision trees. In the previous section, we mentioned a single decision tree which is constructed from the entire set of training examples. The process of constructing the random forest is to partition the training set into subsets and construct each decision tree with its own subset. The ensemble learning scheme which is covered in Chap. 4 is applied to the decision trees in the random forest. This section is intended to describe the random forest as the expansion from the decision tree.



**Fig. 2.24** Random partition of training set

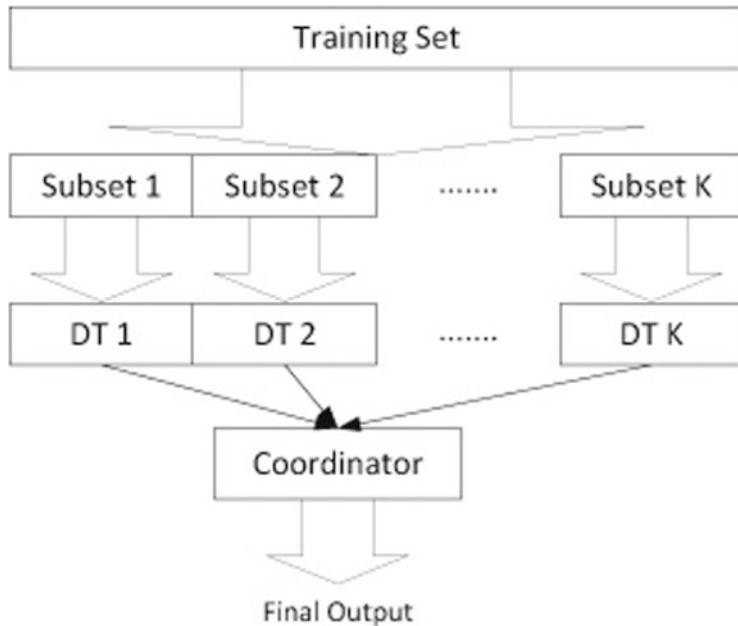


**Fig. 2.25** Training each decision tree with its own subset

The set of the training examples is partitioned into subsets as shown in Fig. 2.24. We need to partition the set for using simplified decision trees as a committee. The process of partitioning the set is to decide the number of subsets in advance and shuffle the examples in the set at random. The entire set which consists of shuffled examples is divided into subsets each of which has its identical number of examples. Each subset is used for constructing its own decision tree as a member of the random forest.

The set of the training examples is partitioned into subsets as shown in Fig. 2.25. We need to partition the set for using simplified decision trees as a committee. The process of partitioning the set is to decide the number of subsets in advance and shuffle the examples in the set at random. The entire set which consists of shuffled examples is divided into subsets each of which has its identical number of examples. Each subset is used for constructing its own decision tree as a member of the random forest.

The process of classifying a data item by the random forest is illustrated in Fig. 2.26. The training set is partitioned at random into subsets, and each decision tree is trained with its own subset. A novice input is classified by each decision tree, and its label is decided by the coordinator, finally. The ensemble learning scheme, such as voting and expert gate which will be covered in Chap. 4, is applied to the decision trees. The version of random forest is various depending on the scheme of partitioning the training set and the scheme of combining the decision trees.



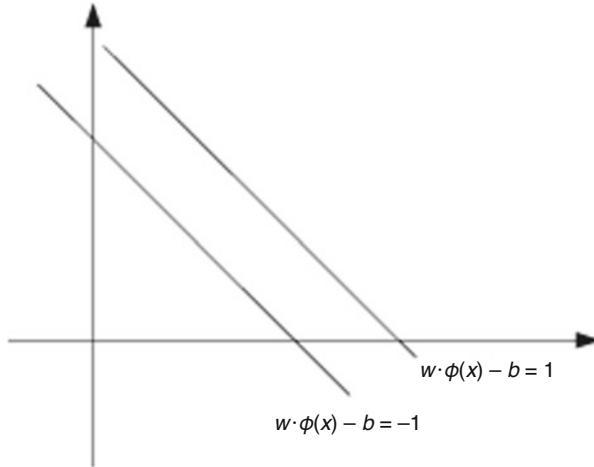
**Fig. 2.26** Ensemble learning of multiple decision trees in random forest

Let us make some remarks on the random forest which consists of multiple decision trees. The training set is partitioned into subsets at random. Each decision tree is constructed from its own subsets of training examples. The ensemble learning is applied to the group of decision trees in classifying a data item. The versions of decision tree are various depending on the criteria of selecting attrite, and we may choose the homogenous random forest which consists of identical versions or the heterogenous one which consists of various versions.

#### 2.4.4 Support Vector Machine

This section is concerned with the SVM as a typical supervised learning algorithm. The linear classifier which defines a hyperplane as the classification boundary is the basis for inventing the SVM. Its idea is to map vectors into ones in another space and define the dual parallel hyperplanes with the maximal margin. The learning process of SVM is to optimize the Lagrange multipliers, instead of weights. This section is intended to describe the SVM as a typical supervised learning algorithm.

The equations of the parallel dual hyperplanes are illustrated in Fig. 2.27. The weight vector and the input vector are notated, respectively, by  $w = [w_1 \ w_2 \ \dots \ w_d]$  and  $x = [x_1 \ x_2 \ \dots \ x_d]$ . The vector which mapped from the



**Fig. 2.27** Weight-based linear classification equation

vector,  $\mathbf{x}$ , is denoted by  $\Phi(\mathbf{x})$ , and the dual parallel hyperplanes are expressed by Eqs. (2.23) and (2.24):

$$\mathbf{w} \cdot \Phi(\mathbf{x}) - b = 1 \quad (2.23)$$

$$\mathbf{w} \cdot \Phi(\mathbf{x}) - b = -1. \quad (2.24)$$

The norm of weight vector which is expressed in Eq. (2.25) should be minimized for maximizing the margin between the dual hyperplanes:

$$\|\mathbf{w}\| = \left( \sum_{i=1}^d x_i^p \right)^p. \quad (2.25)$$

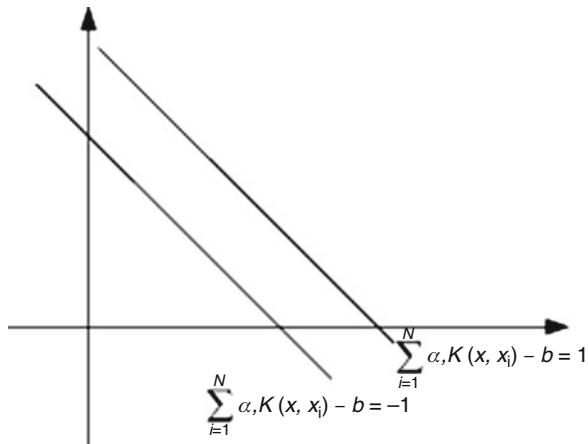
The weights are optimized for minimizing the misclassification rate and maximizing the margin.

The equations of the parallel dual hyperplanes based on the Lagrange multipliers are illustrated in Fig. 2.28. The weight vector is viewed as a linear combination of training examples; the weight vector in Eqs. (2.23) and (2.24) depends on the training examples. Equations (2.23) and (2.24) are transformed into Eqs. (2.26) and (2.27) based on Lagrange multipliers:

$$\sum_{i=1}^N \alpha_i K(\mathbf{x}_i, \mathbf{x}) - b = 1 \quad (2.26)$$

$$\sum_{i=1}^N \alpha_i K(\mathbf{x}_i, \mathbf{x}) - b = -1 \quad (2.27)$$

**Fig. 2.28** Lagrange multiplier-based linear classification equation



The term,  $K(\mathbf{x}_i, \mathbf{x})$ , in Eqs.(2.26) and (2.27), which is called kernel function indicates the inner product between two vectors,  $\Phi(\mathbf{x}_i)$  and  $\Phi(\mathbf{x})$ . The Lagrange multipliers,  $\alpha_1, \alpha_2, \dots, \alpha_N$ , as many as the training examples are optimized as the learning process.

Let us generalize the SVM equations. If the SVM is applied to a binary classification, the output which is given as -1 or 1 is computed by Eq.(2.28):

$$d_i = \begin{cases} 1 & \text{if } \sum_{j=1}^N \alpha_j K(\mathbf{x}_j, \mathbf{x}_i) - b \geq 1 \\ -1 & \text{if } \sum_{j=1}^N \alpha_j K(\mathbf{x}_j, \mathbf{x}_i) - b \leq -1 \end{cases}. \quad (2.28)$$

The above equation is transformed into Eq.(2.29):

$$f(\mathbf{x}) = \text{sign} \left( \sum_{i=1}^N \alpha_i K(\mathbf{x}_i, \mathbf{x}) - b \right) \quad (2.29)$$

If the SVM is applied to the regression, Eq.(2.29) should be modified into Eq.(2.30):

$$f(\mathbf{x}) = \sum_{i=1}^N \alpha_i K(\mathbf{x}_i, \mathbf{x}) - b \quad (2.30)$$

In this case, we may consider mapping the regression into multiple binary classifications.

Let us make some remarks on SVM as the typical supervised learning algorithm. The dual parallel hyperplanes in the mapped space by the weight-based equations.

They are transformed into the Lagrange multiplier-based one. The output value of SVM which is applied to the classification is given as -1 or 1. Because the input vector is explicitly mapped into one in another space, the SVM may be viewed as a deep learning algorithm.

## 2.5 Summary and Further Discussions

Let us summarize what is studied in this chapter. The study about the machine learning is traced to the Naive Retrieval, and we need to define the similarity metric between numerical vectors for setting the machine learning algorithms. The neural networks model is a kind of machine learning algorithms which simulate the nervous system, and an artificial neuron becomes a basic computation unit for generating an output with summing products of inputs and weights. The typical supervised learning algorithms are the decision tree, the random forest, the SVM, and so on. This section is intended to discuss further what is studied in this chapter.

The Gaussian distribution over vectors is characterized by its mean vector and its covariance matrix. In applying the probabilistic learning to the classification, the Gaussian distribution is characterized only by its mean vector, for simplicity. In considering both the mean vector and the covariance matrix, the likelihood of the input vector to a category is computed by Eq. (2.31).

$$f(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^d |\Sigma|}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu})\right) \quad (2.31)$$

Even if both are considered, the covariance matrix may be regarded as a diagonal matrix in assuming independences among the attributes. The complexity of considering the covariance matrix is quadratic to the dimension; if considering it, we need to reduce the number of features, as many as possible.

The MLP is the expansion of the Perceptron for solving its own limit. The Perceptron was invented by Rosenblatt in the 1950s and criticized by Papert and Minskey, in that it is not able to solve the XOR problem. However, the problem is solved by combining multiple Perceptrons hierarchically, and the MLP was invented in the 1980s by adding one more layer called hidden layer. The MLP becomes a very strong tool of the classification and the regression with the universal approximation theorem which means the possibility of approximating any nonlinear function. We will study the MLP as a deep learning algorithm in Chap. 9.

Let us consider the combination of the KNN algorithm and the Naive Bayes. In this combination, the KNN algorithm is in the major position, and the Naive Bayes is in the minor position. The Naive Bayes classifies each training example into nearest neighbor or non-nearest neighbor, and the KNN algorithm classified a data item by voting the labels of the training examples which are classified into nearest neighbor. The Naive Bayes is trained with another set of examples each of which consists of concatenations of two input vectors and its label indicating nearest neighbor or

non-nearest neighbor. In this combination, the KNN algorithm really classifies data items, and the Naive Bayes supports the role of the KNN algorithm.

Let us consider the combination of the two-machine learning algorithm where the Naive Bayes classifies data items as the master position. In the previous combination, the role of the Naive Bayes in this combination is to classify each training example into nearest neighbor or non-nearest neighbor. In this combination, the role of the KNN algorithm is to retrieve nearest neighbors, and the role of the Naive Bayes is to learn the nearest neighbor and classify a novice data item. The discriminated weights which are proportional to the similarities are assigned to the nearest neighbors for computing the likelihoods in the Naive Bayes. In this combination which is opposite to the above combination, the KNN algorithm is in the minor position, and the Naive Bayes is in the major position.

## References

1. T. Jo, “The Implementation of Dynamic Document Organization using Text Categorization and Text Clustering”, PhD Dissertation of University of Ottawa, 2006.
2. T. Jo, “Semantic Word Categorization using Feature Similarity based K Nearest Neighbor”, 67–78, Journal of Multimedia Information Systems, 2018.

# Chapter 3

## Unsupervised Learning



This chapter is concerned with the unsupervised learning as another kind of swallow learning. It refers to the learning types where its parameters are optimized for stabilizing distances of cluster members from their own cluster prototypes. In the process of the unsupervised learning, unlabeled training examples are prepared, a similarity metric or a distance metric between training examples is defined, and cluster prototypes are optimized for maximizing their cohesions. Unsupervised learning algorithms are applied to clustering which is the process of segmenting a data item group into subgroups each of which contains similar ones. This chapter is intended to review the unsupervised learning before studying the deep learning.

This chapter is organized into five sections, and in Sect. 3.1, we describe the unsupervised learning conceptually. In Sect. 3.2, we review some simple unsupervised learning algorithms, such as AHC (agglomerative hierarchical clustering) algorithm, divisive algorithm, and k means algorithm. In Sect. 3.3, we describe various versions of Kohonen Networks. In Sect. 3.4, we describe the EM (expectation maximization) algorithm, as an advanced unsupervised learning algorithm. In Sect. 3.5, we summarize the entire contents of this chapter and discuss further on what is studied in this chapter.

### 3.1 Introduction

This section is concerned with the unsupervised learning in the conceptual view. In the unsupervised learning, it is assumed that the training examples are prepared as unlabeled ones, and the similarity metric between examples is defined in advance. The unsupervised learning is the process which optimizes the parameters which are usually given as cluster prototypes for maximizing the cohesion in each cluster and the discriminations among clusters, depending on similarities of cluster prototypes with training examples. The unsupervised learning is applied to data clustering which partitions a group of entire data items into subgroups each of which contains

similar ones. This section is intended to describe the unsupervised learning for providing the introduction, conceptually.

Let us mention the training set which is prepared for the unsupervised learning. All of training examples in the set are assumed to be unlabeled, the training set is notated by  $Tr = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$ , and each element in the set is given as a  $d$  dimensional vector notated by  $\mathbf{x}_i = [x_{i1} \ x_{i2} \ \dots \ x_{id}]$ . It is required to define a similarity metric between two vectors,  $\mathbf{x}_i$  and  $\mathbf{x}_j$ , for implementing the unsupervised learning. The supervised learning depends on the difference between the target output vector and the computed one, whereas the unsupervised learning depends on the similarities among the training examples. In the semi-supervised learning, both the labeled training set and the unlabeled training set are used.

Let us mention the unsupervised learning frame. Unlabeled examples are prepared, and the similarity metric between two vectors are defined, in advance. The parameters which indicate cluster prototypes are initialized at random, and data items are arranged depending on their similarities with the parameters. They are updated for maximizing the cohesion of each cluster and the discrimination among clusters. The unsupervised learning is applied to the data clustering which is the process of segmenting a group of data items into subgroups, each of which contains similar ones.

Let us mention some machine learning algorithms as typical instances of unsupervised learning. Simple clustering algorithms are the AHC algorithm which clusters data items in the bottom-up direction and the divisive algorithm which clusters them in the top-down direction. The most popular clustering algorithm is the k means algorithm which iterates updating the mean vectors and arranging data items based on their similarities with the mean vectors. The EM algorithm clusters data items by estimating cluster membership values for each data item and optimizing the parameters for maximizing the cluster likelihoods. In this chapter, we will review some unsupervised learning algorithms for providing the background for understanding the deep learning.

Let us mention what is intended in this chapter. We review the simple clustering algorithms such as the AHC algorithm, the divisive clustering algorithm, and the k means algorithm for warming up to study the unsupervised learning. We study the various versions of the Kohonen Networks model which is initially designed as an unsupervised neural networks model. We study the EM algorithm as the most popular unsupervised learning algorithm. This chapter is intended to study the unsupervised learning as the other type of swallow learning.

## 3.2 Simple Unsupervised Learning Algorithms

This section is concerned with some simple clustering algorithms. In Sect. 3.2.1, we mention the AHC algorithm which clusters data items in the bottom-up direction. In Sect. 3.2.2, we mention the divisive algorithm which does them in the top-down direction. In Sect. 3.2.3, we mention the online linear clustering algorithm which

**Fig. 3.1** Singletons

focuses on the clustering speed. In Sect. 3.2.4, we mention the k means algorithm which is used most popularly for clustering data items.

### 3.2.1 AHC Algorithm

This section is concerned with the AHC algorithm as a simple clustering algorithm. The first step of using the clustering algorithm is to define a similarity metric between two vectors or two clusters. Data items are clustered by iterating computing similarities of all possible item pairs and merging the item pair with its highest similarity. The AHC algorithm is characterized as the process of clustering data items with the bottom-up direction; it begins with singletons as many as items. This section is intended to describe the AHC algorithm as a simple clustering algorithm.

The singletons which are initial clusters in using the AHC algorithm as many as data items are illustrated in Fig. 3.1. The bottom-up is the direction of clustering data items by the AHC algorithm. The clusters are initially constructed with the singletons as many as data items; a singleton is a cluster which contains a single data item. If multiple items are initially assigned to same cluster according to prior knowledge, the initial cluster may consist of multiple items. In proceeding the clustering, the similarity metric between items is defined for computing the similarity between clusters.

The process of computing the similarity between clusters is illustrated in Fig. 3.2. The two clusters are notated as sets of numerical vectors, by  $C_1 = \{\mathbf{x}_{1,1}, \mathbf{x}_{1,2}, \dots, \mathbf{x}_{1,|C_1|}\}$  and  $C_2 = \{\mathbf{x}_{2,1}, \mathbf{x}_{2,2}, \dots, \mathbf{x}_{2,|C_2|}\}$ . The mean vectors of the two clusters are computed by Eqs. (3.1) and (3.2):

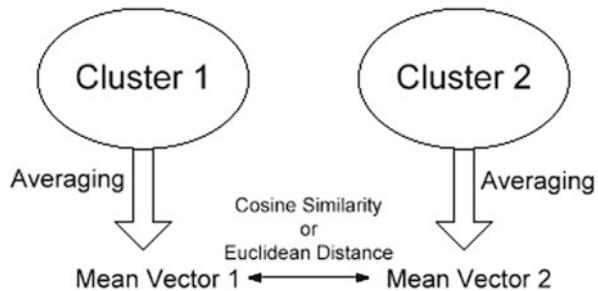
$$\boldsymbol{\mu}_1 = \frac{1}{|C_1|} \sum_{i=1}^{|C_1|} \mathbf{x}_{1,i} \quad (3.1)$$

$$\boldsymbol{\mu}_2 = \frac{1}{|C_2|} \sum_{i=1}^{|C_2|} \mathbf{x}_{2,i} \quad (3.2)$$

The similarity between the two clusters,  $C_1$  and  $C_2$ , is computed by Eq. (3.3):

$$sim(\boldsymbol{\mu}_1, \boldsymbol{\mu}_2) = \frac{\|\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2\|}{\|\boldsymbol{\mu}_1\| \cdot \|\boldsymbol{\mu}_2\|} \quad (3.3)$$

**Fig. 3.2** Similarity between clusters



Other similarity metrics between two numerical vectors may be applied for computing the similarity between two clusters.

The process of clustering data items by the AHC algorithm is illustrated as a pseudo code in Fig. 3.3. As the start of the AHC algorithm, singletons as many as items are created. All possible cluster pairs are generated, and the similarity is computed for each pair. The clusters in the pair with the highest similarity are merged into a cluster; computing the similarities and merging clusters are iterated, until reaching the desired number of clusters. The bottom-up is the direction of clustering data items by the AHC algorithm as shown in Fig. 3.3.

Let us make some remarks on the AHC algorithm as a simple clustering algorithm with the bottom-up direction. The singletons, each of which is a cluster with a single item, indicate the start of clustering data items by the AHC algorithm. The similarity between cluster mean vectors is the similarity between clusters in using the AHC algorithm. The process of clustering data items with the AHC algorithm is to iterate computing the similarity between clusters and merging clusters with the highest similarity into one. We may consider clustering data items in the opposite direction called the top-down direction.

### 3.2.2 *Divisive Algorithm*

This section is concerned with another simple clustering algorithm, called divisive algorithm. In the previous section, we studied the AHC algorithm which clusters data items in the bottom-up direction. The divisive algorithm clusters data items by starting with a single group of all data items and dividing a cluster recursively in the top-down direction. The division of a cluster into the similar group to the selected pivot and the remaining is the core operation of this clustering algorithm. This section is intended to describe the divisive algorithm with the opposite direction of the AHC algorithm.

The initial status of the divisive clustering algorithm is illustrated in Fig. 3.4. As shown in Fig. 3.1, the fact that there are singletons as many as data items is the start of AHC algorithm. A group of  $N$  is the start of the divisive algorithm which proceeds clustering data items in the top-down direction. Various versions of

**Fig. 3.3** Process of clustering data items by AHC algorithm

```

List clusterDataItemList(List itemList, int desiredClusterListSize){
    int itemSize = itemList.size();
    if(itemSize <= desiredClusterNumber)
        return null;
    List clusterList = new List();
    //Initialize Clusters
    for(int i = 0; i < itemSize; i++){
        Cluster cc = new Cluster();
        Item dataItem = itemList.getElement(i);
        cc.addDataItem(dataItem);
        clusterList.addElement(cc);
    }
    int clusterListSize = clusterList.size();

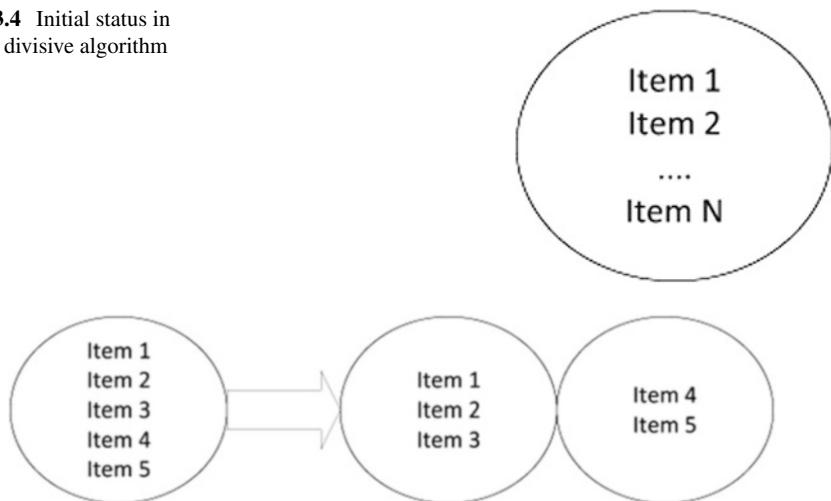
    //Cluster Data Items in the bottom up direction
    while(clusterList > desiredClusterListSize){
        double maxSimilarity = 0.0;
        int maxIndex1 = 0;
        int maxIndex2 = 0;
        for(int i = 0; i < clusterListSize; i++){
            Cluster c1 = clusterList.getElement[i];
            for(int j = 0; j < clusterListSize; j++){
                Cluster c2 = clusterList.getElement[j];
                double similarity = sim[c1,c2];
                if(similarity > maxSimilarity){
                    maxSimilarity = similarity;
                    maxIndex1 = i;
                    maxIndex2 = j;
                }
            }
            Cluster cmax1 = clusterList.getElement[maxIndex1];
            Cluster cmax2 = clusterList.getElement[maxIndex2];
            Cluster mergeCluster = cmax1.merge(cmax2);
            clusterList.deleteElement(cmax1);
            clusterList.deleteElement(cmax2);
            clusterList.addElement(mergeCluster);
        }
    }
}

```

divisive algorithm are derived, depending on the scheme of dividing a cluster into multiple clusters. When Figs. 3.4 and 3.1 are compared with each other, they present the opposite property of the AHC algorithm and the divisive algorithm.

The division of a cluster into two clusters is illustrated in Fig. 3.5. The cluster division is the process of dividing a cluster into two clusters by analyzing the similarities among its members. The cluster division process is to select an item at random as the pivot and arrange members depending on their similarities with the pivot. The results from the cluster division are the two clusters: the pivot similar group which consists of members which are similar as the pivot and the remaining group which consists of the others. The cluster division is the main operation for

**Fig. 3.4** Initial status in using divisive algorithm



**Fig. 3.5** Cluster division

implementing the divisive clustering algorithm which proceeds clustering data items in the top-down direction.

Let us mention the process of clustering data items in the divisive algorithm which is illustrated in Fig. 3.6. A single group of all data items is the start of clustering them with the divisive algorithm. A data item is selected at random as the pivot, and a cluster is divided into the two clusters: similar cluster and remaining cluster. Data items are clustered by dividing the remaining cluster into the similar cluster and the remaining cluster, recursively. Clustering data items proceeds in the top-down direction.

Let us make some remarks on the divisive algorithm as a simple clustering algorithm. The divisive algorithm starts with a single cluster of all data items. The main operation in the divisive clustering algorithm is to divide a cluster into two clusters: pivot similar cluster and remaining cluster. The data items are clustered by dividing a cluster into two cluster, recursively. The combination of the divisive clustering algorithm with the AHC algorithm is considered for proceeding clustering data items in both directions.

### 3.2.3 *Online Linear Clustering Algorithm*

This section is concerned with the third simple clustering algorithm called online linear clustering algorithm. In the previous sections, we studied the AHC algorithm which proceeds the data clustering with the bottom-up direction and the divisive algorithm which proceeds it with the top-down direction. This clustering algorithm

```

List clusterDataItemList(List itemList, int desiredClusterListSize, double similarityThreshold){
    int itemSize = itemList.size();
    List clusterList = new List();
    if(itemSize <= desiredClusterNumber)
        return null;
    List cluster = new List();
    //Initialize Clusters
    Cluster cc = new Cluster();
    for(int i = 0;i < itemSize; i++){
        Item dataItem = itemList.getElement(i);
        cc.addDataItem(dataItem);
    }
    clusterList.addElement(cc);
    //Cluster Data Items in the top down direction
    while(!cc.isEmpty()){
        int clusterSize = cc.size();
        int pivotIndex = random(0,clusterSize);
        Item pivotItem = cc.getElement(pivotIndex);
        cc.removeDataItem(pivotIndex);
        clusterSize = remainingCluster.size();
        Cluster pivotCluster = new Cluster();
        Cluster remainingCluster = new Cluster();
        pivotCluster.addElement(pivotItem);

        for(int i = 0; i < clusterSize;i++){
            Item dataItem = cc.getDataItem(i);
            double similarity = pivotItem.similarity(dataItem);
            if(similarity >= similarityThreshold)
                pivotCluster.addDataItem(dataItem);
            else
                remainingCluster.addDataItem(dataItem);
        }
        clusterList.addElement(pivotCluster);
        cc = remainingCluster;
    }
}

```

**Fig. 3.6** Process of clustering data items by divisive algorithm

**Fig. 3.7** Initial status in using online linear clustering algorithm



proceeds the data clustering from a single singleton with the almost linear complexity. The property of this clustering algorithm is that its clustering speed is very good, but the cluster quality is very poor. This section is intended to describe the online linear clustering algorithm with respect to its clustering process.

The initial status in using the online linear clustering algorithm is illustrated in Fig. 3.7. It is assumed that the data items which we try to cluster are given as a list. An empty cluster is created, and the first item is included in the cluster. The subsequent data items are processed sequentially with the start of a single singleton, which is presented in Fig. 3.7. The data item which is included initially in the cluster becomes the cluster prototype.

**Fig. 3.8** Process of clustering data items using online linear clustering algorithm

```

List clusterDataItemList(List itemList, double similarityThreshold){
    int itemSize = itemList.size();
    List clusterList = new List();
    //Initialize Clusters
    Cluster cc = new Cluster();
    Item dataItem = itemList.getElement(0);
    cc.addDataItem(dataItem);
    clusterList.addElement(cc);

    //Cluster Data Items
    for(int i = 1;i < itemSize;i++){
        Item dataItem = itemList.getElement(i);
        int clusterSize = clusterList.size();
        double maxSimilarity = 0.0;
        int maxClusterIndex = 0;
        for(int j = 0;j < clusterSize;j++){
            Cluster cc = clusterList.getElement(j);
            double similarity = dataItem.computeSimilarity(cc);
            if(similarity > maxSimilarity){
                maxSimilarity = similarity;
                maxClusterIndex = j;
            }
        }
        if(maxSimilarity >= similarityThreshold){
            Cluster maxCluster = clusterList.getElement(maxIndex);
            maxCluster.addElement(dataItem);
            clusterList.setElement(maxIndex,maxCluster);
        }
        else{
            Cluster newCluster = new Cluster();
            newCluster.addElement(dataItem);
            clusterList.addElement(newCluster);
        }
    }
}

```

The process of clustering data items by the online linear clustering algorithm is illustrated as the pseudo code in Fig. 3.8. A cluster is created, and the first item is included in it. For each item, its similarities with the existing clusters are computed, the maximal similarity is retrieved, and if the maximum similarity is greater than or equal to the threshold, it is included in the cluster corresponding to the maximal similarity. Otherwise, one more cluster is created, and the item is included in the new cluster; data items are clustered by iterating the process to all data items. If the number of clusters is small, it takes almost linear complexity for clustering data items.

Some variants may be derived from the online linear clustering algorithm. In the initial version, a data item which is initially jointed into a cluster is its cluster prototype, whereas in the first variant, the cluster mean vector is its cluster prototype. In the initial version, one cluster is created, and a data item joins into the cluster, whereas in the second variant, multiple clusters are created at a time, and multiple items join into their own clusters. In the initial version, the cluster prototypes are fixed to the data items which join initially to the clusters, whereas in the third

variant, the cluster prototypes are updated continually. The third variant is intended to improve the clustering quality by sacrificing the clustering speed.

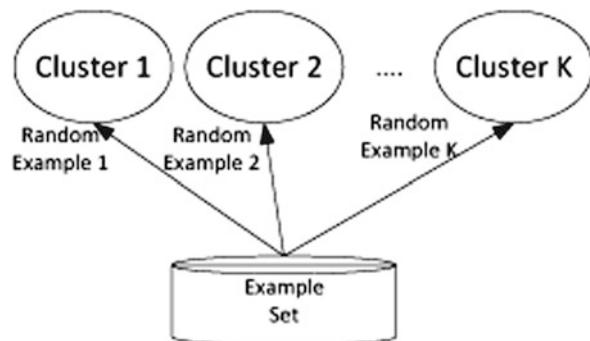
Let us make some remarks on the online linear clustering algorithm which is mentioned as a simple clustering algorithm. A single singleton which is a cluster with an item is the start of executing the online linear clustering algorithm. Each successive item chooses either of joining into an existing cluster or becoming the new cluster prototype. Some variants of online linear clustering algorithms are considered for improving the performance, sacrificing the speed. Because the cluster prototypes are not updated, the initial version was validated that its clustering speed is excellent, but its clustering performance is poor [1].

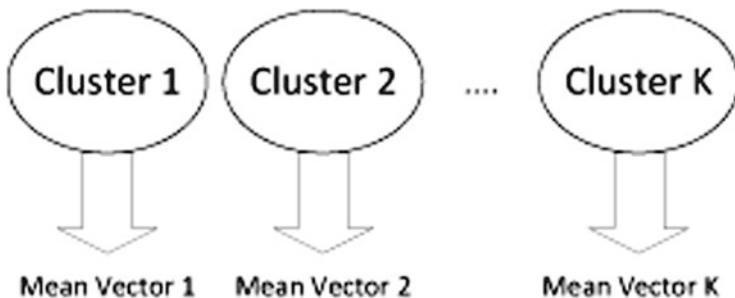
### 3.2.4 K Means Algorithm

This section is concerned with the k means algorithm as the most popular clustering algorithm. In the previous sections, we studied the simple clustering algorithms, the AHC algorithm, the divisive clustering algorithm, and the online linear clustering algorithm. As the preparation for applying the k means algorithm to the data clustering, the number of clusters should be decided in advance, and some data items should be selected at random as the initial cluster mean vectors. The algorithm iterates computing the cluster mean vectors and arranging the data items depending on their similarities with the cluster mean vectors. This section is intended to describe the k means algorithm as the most popular clustering algorithm.

The initial status in using the k means algorithm for clustering the data items is illustrated in Fig. 3.9. It is required to decide the number of clusters in advance for using the k means algorithm; it is assumed that the number of clusters is initially decided as  $k$ . The  $k$  clusters are created as empty ones, and  $k$  data items are selected at random. The  $k$  data items become the initial mean vectors, and the others are arranged into clusters, depending on their similarities with the mean vectors. The  $k$  singletons become the start of clustering the data items with the k means algorithm.

**Fig. 3.9** Mean vector initialization





**Fig. 3.10** Mean vector update

**Fig. 3.11** Fuzzy training set partition



The update of the mean cluster vectors is illustrated in Fig. 3.10. The mean vectors are initialized by selecting data items at random. For each data item, its similarities with the mean vectors are computed, and it is arranged into the cluster whose mean vector is most similar. After arranging the data items into clusters, the mean vectors are updated by computing them again. The data items are clustered by iterating computing the mean vectors and arranging data items.

The process of clustering the data items by the k means algorithm is illustrated in Fig. 3.11. The list of data items, the number of clusters, and the number of iterations is given as the arguments. The determined number of clusters are created, and their mean vectors are initialized by selecting data items at random. It iterates arranging data items into clusters, depending on their similarities with the mean vectors and updating the mean vectors by computing them again. The list of clusters is returned as the results.

Let us make some remarks on the k means algorithm which is covered in this section as the most popular clustering algorithm. It is required to decide the number of clusters in advance for using the k means algorithm. Whenever data items are arranged into their own clusters, the cluster mean vectors are updated. The data items are clustered by iterating updating the cluster mean vectors and arranging the data items. The k means algorithm is expanded into the EM algorithm where each cluster is assumed as a normal distribution which is characterized with its mean vector and its covariance matrix.

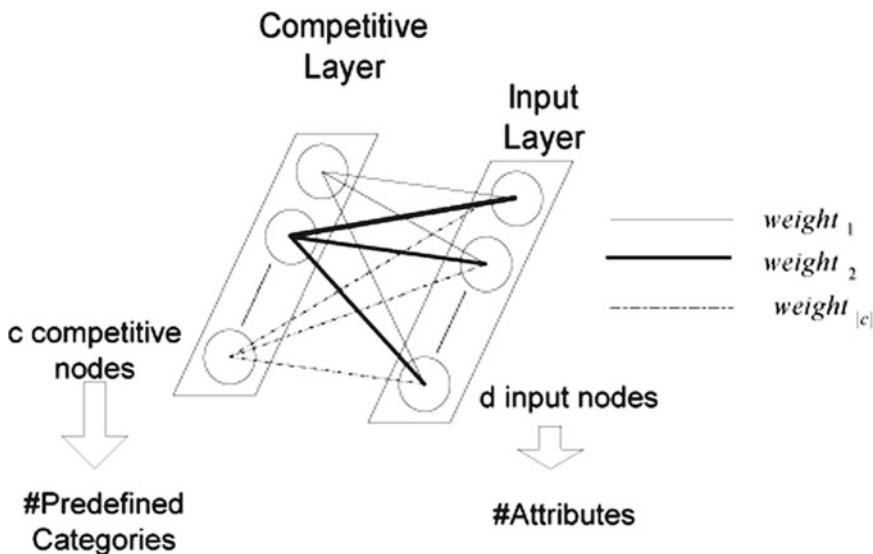
### 3.3 Kohonen Networks

This section is concerned with several versions of Kohonen Networks. In Sect. 3.3.1, we mention the initial version of Kohonen Networks which is designed as an unsupervised learning algorithm. In Sect. 3.3.2, we mention the LVQ (learning vector quantization) as a supervised learning algorithm. In Sect. 3.3.3, we modify the Kohonen Networks into the semi-supervised version. In Sect. 3.3.4, we expand the Kohonen Networks into the SOM (self-organizing map).

#### 3.3.1 Initial Version

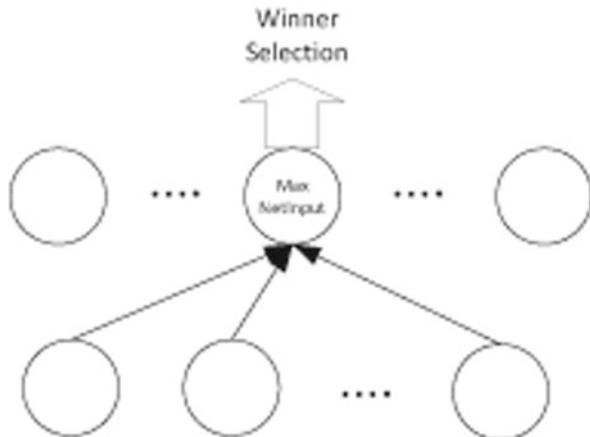
This section is concerned with the initial version of Kohonen Networks model as an unsupervised learning algorithm. In the architecture, there are two layers: the input layer and the competitive layer. The weight vectors are given as the cluster prototypes and updated as the learning process. The self-organizing map is the version which is expanded from the initial version and is intended to draw the similarity on the input pattern. This section is intended to describe the architecture and the learning process of the Kohonen Networks model.

The architecture of the Kohonen Networks is illustrated in Fig. 3.12. There are two layers in the architecture: the input layer which receives the input vector and the competitive layer which indicates the clusters. The weight vector which is connected



**Fig. 3.12** Kohonen Network architecture

**Fig. 3.13** Winner selection in Kohonen Networks



to the competitive node is given as a cluster prototype vector which corresponds to a competitive node. One among the competitive nodes whose weight vector is most similar as the input vector is decided as the winner. The number of input nodes is the dimension of the input vector, and the number of competitive nodes is the number of clusters.

The selection of the winner in the output layer of the Kohonen Networks is illustrated in Fig. 3.13. The Kohonen Networks are designed as the  $d$  input nodes in the input layer and the  $c$  output nodes in the output layer, and each input node and each output node are denoted, respectively, by  $x_i$  and  $y_j$ . The weight between the two nodes is denoted by  $w_{ji}$ , the net input is computed by Eq. (3.4), and the output node whose net input is maximal is selected as the winner:

$$net_j = \sum_{i=1}^d w_{ji} x_i \quad (3.4)$$

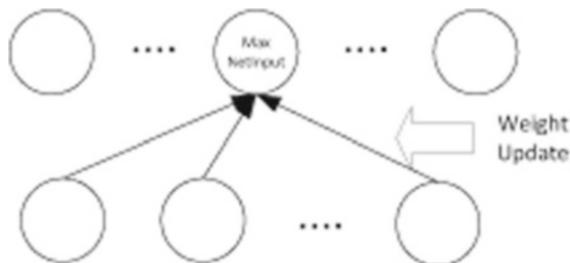
The output node value is zero or one, as expressed in Eq. (3.5):

$$y_j = \begin{cases} 1 & \text{if } \sum_{i=1}^d w_{ji} x_i = \max_{j=1}^c \sum_{i=1}^d w_{ji} x_i \\ 0 & \text{otherwise} \end{cases}. \quad (3.5)$$

Only one node has 1 as the winner, and the others have 0, in the output layer.

Updating the weights between the input layer and the output layer is illustrated in Fig. 3.14. For each output node, its net input is computed by Eq. (3.4), and the output node with its maximal net input is selected as the winner. The weights which are connected to the output node which is selected as the winner are updated by Eq. (3.6):

**Fig. 3.14** Weight update in Kohonen Networks



$$w_{ji} \leftarrow w_{ji} - \eta y_j (w_{ji} - x_i) \quad (3.6)$$

where  $\eta$  is the learning rate which is set arbitrary as an external parameter. The output node which is selected as the winner has 1 as its output value. The weights which are connected to each output node is viewed as a cluster prototype which is given as a  $d$  dimensional vector.

Let us make some remarks on the initial version of the Kohonen Networks which are initially designed for the unsupervised learning. In the architecture, the input layer receives the input vector, and each node in the output layer indicates a cluster. The net input of each output node is the inner product of the input vector and a weight vector, and the output node with its maximal net input is selected as the winner. The weights which are connected to the winner are updated closely to the input vector. The learning process of the Kohonen Networks is to iterate selecting an output node as the winner and update the weights which are connected to the winner.

### 3.3.2 Learning Vector Quantization

This section is concerned with the supervised version of Kohonen Networks which is called LVQ (learning vector quantization). In the previous section, we studied the initial version of Kohonen Networks as an unsupervised learning algorithm. In this section, we study the LVQ into which the Kohonen Networks are modified into as the supervised version. In the previous version, the output node with its maximal inner product of the input vector and the weight vector is selected as the winner, whereas in this version, the output which corresponds to the target label is selected as the winner. This section is intended to describe the LVQ as the supervised version of the Kohonen Networks.

Let us mention the transition of the unsupervised learning algorithm into the supervised learning algorithm. The supervised learning depends on the difference between the target output and the computed output, whereas the unsupervised learning depends on the similarities among the input vectors. The cluster prototypes are replaced by the predefined categories in the process of modifying the unsupervised learning algorithm into the supervised learning algorithm. For instance, the cluster

mean vector which is a cluster prototype is replaced by the mean vector which characterizes its own category, in case of modifying the k means algorithm into its supervised version, called Bayes classifier. Other unsupervised learning algorithm are modified into their supervised versions by doing so.

Let us mention the process of selecting an output node as the winner in the LVQ. The given problem is assumed as a hard classification, and the input vector and the output vector are, respectively, as  $d$  dimensional vector,  $\mathbf{x}_i = [x_{i1} \ x_{i2} \ \dots \ x_{id}]$ , and  $c$  dimensional vector,  $\mathbf{y} = [y_{i1} \ y_{i2} \ \dots \ y_{ic}]$ . In the architecture of LVQ, there are  $d$  nodes in the input layer and  $c$  nodes in the output layer. In the output vector, which is given as a binary vector, only one element which corresponds to the target category is given as one, and the others are given as zeros. The output node which corresponds to the target category is selected as the winner in the LVQ.

Let us mention the process of updating the weights as the learning process of LVQ. The output node which corresponds to the target category is selected as the winner in the LVQ. The process of updating the weights in the LVQ is expressed as Eq. (3.7):

$$w_{ji} \leftarrow w_{ji} - \eta y_j (w_{ji} - x_i) \quad (3.7)$$

Only one element in the  $c$  dimensional output vector is given as one, and the weights which are connected to the winner are updated for each training example. The direction of updating the weights in the LVQ is to minimize the error between the target output and the computed output.

Let us make some remarks on the LVQ which is the supervised version of Kohonen Networks. The unsupervised learning algorithm is modified into its supervised version by replacing the cluster prototypes by the category properties. In the LVQ, the output which corresponds to the target label is selected as the winner. The weights which are connected to the winner are updated into ones closer to the input vector. The process of updating weights based on the output node which is selected as the winner is called competitive learning.

### 3.3.3 *Semi-supervised Model*

This section is concerned with the semi-supervised version of the Kohonen Networks. In the previous sections, we studied both the unsupervised and the supervised versions of Kohonen Networks. In this section, we study the semi-supervised version of Kohonen Networks which covers both labeled and unlabeled training examples. To a labeled training example, the output node which corresponds to its target label is selected as the winner, whereas to an unlabeled training example, the output node from which the weight vector is closest to the input vector is selected as the winner. This section is intended to describe the semi-supervised version of Kohonen Networks, with respect to its computation of the output vector and its learning process.

**Fig. 3.15** Labeled and unlabeled training set

Labeled Set	Unlabeled Set
-------------	---------------

The training set which is prepared for the semi-supervised learning is illustrated in Fig. 3.15. The set of labeled examples and the set of unlabeled examples are prepared for the semi-supervised learning. The former and the latter are notated, respectively, by  $Tr_L = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{N_L}\}$  and  $Tr_U = \{\mathbf{x}_{N_L+1}, \mathbf{x}_{N_L+2}, \dots, \mathbf{x}_N\}$ . In reality, it is expansive to obtain labeled examples, whereas it is cheap to obtain unlabeled examples; unlabeled examples are usually dominant over labeled ones in the semi-supervised learning. It is intended to utilize unlabeled examples for improving the generalization performance.

Let us mention the selection of an output node as the winner in the semi-supervised version. The training set is prepared as the two sets: the set of labeled examples and the set of unlabeled examples. The output node value is computed by Eq. (3.5) to each unlabeled example. The output node value is computed by Eq. (3.8) to each labeled example:

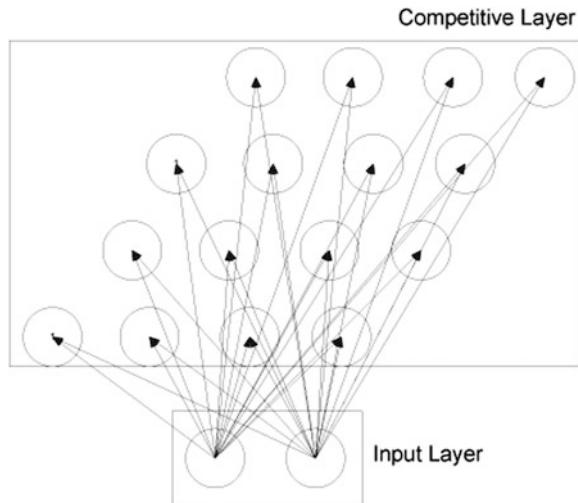
$$y_j = \begin{cases} 1 & \text{if } o_j = 1 \\ 0 & \text{otherwise} \end{cases}. \quad (3.8)$$

The process of selecting the winner is viewed as the mixture of the supervised version and the unsupervised version.

Let us mention the process of updating the weights in the semi-supervised version of Kohonen Networks. The weights which are connected to the output node which is selected as the winner by the above process are updated. There is no change in updating the weights as shown in Eq. (3.7). In each iteration, it is possible to select a different output node to the same unlabeled example, but a fixed output node is selected as the winner to the same labeled example. The role of the labeled examples is to guide learning the unlabeled examples.

Let us make one remark on the semi-supervised version of the Kohonen Networks. Both the set of labeled examples and the set of unlabeled examples are prepared for the semi-supervised learning. To each unlabeled example, an output node selected as the winner by adopting the style of the Kohonen Networks, and to each labeled example, an output node is selected by adopting the style of the LVQ. The value, 1.0, is assigned to the output node which is selected as the winner as its output value, and the weights are updated by Eq. (3.7). The semi-supervised version of the Kohonen Networks is viewed as the mixture of the initial version and the supervised version.

**Fig. 3.16** Architecture of self-organizing map



### 3.3.4 Self-Organizing Map

This section is concerned with a variant of the Kohonen Networks, called SOM (self-organizing map). In the previous versions, we studied the three versions of Kohonen Networks: the initial version, the supervised version, and the semi-supervised version. The SOM which is covered in this section is based on the Kohonen Networks and is intended to visualize the input vectors into the two-dimensional grid or the three-dimensional grid based on their similarities. The weights which are connected to nearest neighbors as well as the winner are updated in this version. This section is intended to describe the SOM which is intended to map the data items into the grid.

The architecture of the SOM is illustrated in Fig. 3.16. The number of nodes in the output layer which is called competitive layer is the number of clusters or the number of categories in the Kohonen Networks and the LVQ. An arbitrary massive number of nodes in the competitive layer is given in the architecture of SOM regardless of the number of clusters. The number of nodes in the input layer is the dimension of the input vector in SOM like the Kohonen Networks and the LVQ. The SOM is intended to visualize the input vector in the two-dimensional grid or the three-dimensional grid.

Let us mention the process of selecting output nodes as the winners in the SOM. It is assumed that the output nodes are given as a two-dimensional grid; each output node,  $y_{(j,k)}$ , is identified by a coordination in the grid,  $(j, k)$ ; and the weight vector which is connected to the output node,  $y_{(j,k)}$ , is notated by  $\mathbf{w}_{(j,k)}$ . The inner product of the input vector and the weight vector is computed as the net input of the output node,  $y_{(j,k)}$ ,  $\mathbf{w}_{(j,k)} \cdot \mathbf{x}$ . The output node whose net input is maximal is selected as the winner by Eq. (3.9):

$$y_{win} = \underset{j=1, k=1}{\operatorname{argmax}} \mathbf{w}_{(j,k)} \cdot \mathbf{x}. \quad (3.9)$$

The winner influences on its neighbors with respect to the weight updates.

Let us mention the process of updating the weights in the SOM. In the previous versions, the weights which are connected to the selected output node are updated, and the others are not updated. The function of the selected output node and the current output node which is called neighborhood function is introduced for defining the weight update rule. The rule of updating the weight in the SOM is expressed as Eq. (3.10):

$$\mathbf{w}_{(j,k)} \leftarrow \mathbf{w}_{(j,k)} + \eta \theta(y_{win}, y_{(j,k)}) (\mathbf{x} - \mathbf{w}_{(j,k)}) \quad (3.10)$$

The neighborhood function which replaces the output value is proportional to the similarity between the selected output and the current output node.

Let us make some remarks on the SOM which is the version of Kohonen Networks for visualizing the input vector into a two-dimensional grid or a three-dimensional one. A massive number of output nodes is arranged as a grid form regardless of the number of clusters in the output layer. The output node is selected as the winner by the inner product of the input vector and the output vector like the Kohonen Networks. The weights which are connected to not only the winner but also its neighbors are updated. The neural gas is derived from the SOM as its variant by replacing the neighborhood function by an exponential based one.

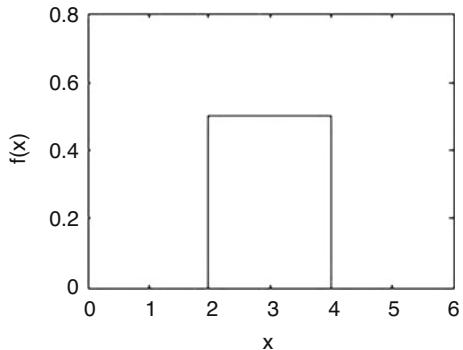
## 3.4 EM Algorithm

This section is concerned with the EM algorithm as an advanced clustering algorithm. In Sect. 3.4.1, we mention several probability distributions such as the normal distribution and the fuzzy distribution. In Sect. 3.4.2, we define the notations which are necessary for explaining the EM algorithm. In Sect. 3.4.3, we describe the E-step for estimating cluster memberships for each data item. In Sect. 3.4.4, we describe the M-step for updating clustering distributions based on cluster memberships.

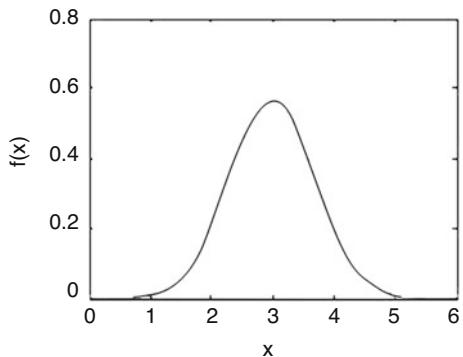
### 3.4.1 Cluster Distributions

This section is concerned with some distributions which may be defined to each cluster. It is required to define the probability distribution to each cluster for applying the EM algorithm to the data clustering. In this section, we mention the uniform distribution, the normal distribution, and the fuzzy distribution as the representative ones which are defined to each cluster. The probability distribution is characterized by its own parameters; the process of clustering data items by the

**Fig. 3.17** Uniform distribution



**Fig. 3.18** Normal distribution



EM algorithm is to optimize the parameters iteratively. This section is intended to describe the three representative probability distribution.

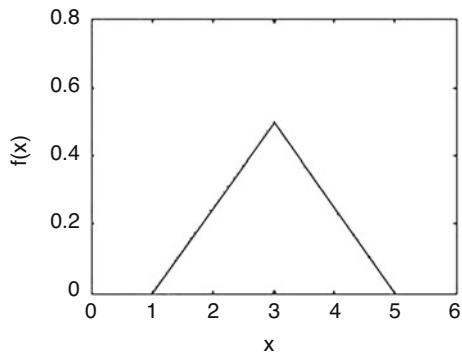
The uniform distribution is illustrated in Fig. 3.17. It is the continuous probability distribution where the probability is constant within the interval. The uniform distribution is characterized with the start point,  $a$ , and the end point,  $b$ , indicating the interval. The probability between  $a$  and  $b$  is  $\frac{1}{b-a}$ . As shown in Fig. 3.17, the rectangular shape is made in the uniform distribution.

The normal distribution on a value,  $x$ , is illustrated in Fig. 3.18. It is the continuous probability distribution with the bell shape which is characterized by its mean and its variance. The normal distribution which is presented in Fig. 3.18 is characterized by 3 as its mean and 1 as its variance. The equation for computing the probability of a particular value,  $x$ , in the normal distribution is expressed as Eq. (3.11):

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \quad (3.11)$$

This distribution is used most popularly for assuming cluster distribution in using the EM algorithm.

**Fig. 3.19** Fuzzy distribution:  
triangle distribution



The fuzzy distribution on the value,  $x$ , is illustrated in Fig. 3.19. It is the continuous probability distribution with the triangle shape, which is characterized by the start point,  $a$ , and the end point,  $b$ . In the fuzzy distribution, the probability is peak in the mid-point,  $\frac{a+b}{2}$ , and its value is  $\frac{2}{b-a}$ . In the example which is presented in Fig. 3.19, the fuzzy distribution is characterized by the start point, 1, and the end point, 5, and its probability is peak at 3 with the value, 0.5. The continuous probability distribution with its trapezoid is characterized by the four points as another kind of fuzzy distribution.

Let us make some remarks on some continuous probability distributions which are assumed to clusters in using the EM algorithm. The uniform distribution is the simplest distribution where the probability is identical within an interval. The normal distribution which is called Gaussian distribution is the bell-shaped one which is characterized by its mean and its variance. The fuzzy distribution is the triangle-shaped one which its probability is peak in the mid-point. In this section, we mentioned only some distributions as representative ones and consider other distributions over clusters.

### 3.4.2 Notations

This section is concerned with the notations for studying the EM algorithm. In the previous section, we studied some probability distributions which are defined for clusters. In this section, we define the notations which are involved in studying the EM algorithm. In the EM algorithm, the membership of each data item in a cluster is assumed as a fuzzy value. This section is intended to define and explain the involved notations.

Let us notate the list of clusters. The number of clusters is initially decided as  $M$  in using the EM algorithm. An individual cluster is notated by  $C_i$ , and it is viewed as a set of examples,  $C_i = \{\mathbf{x}_{i1}, \mathbf{x}_{i2}, \dots, \mathbf{x}_{i|C_i|}\}$ . The list of clusters is expressed as a set,  $C = \{C_1, C_2, \dots, C_M\}$ , and the union of the clusters is the entire set of unlabeled examples, as shown in Eq. (3.12):

$$Tr = \bigcup_{i=1}^M C_i \quad (3.12)$$

The list of exclusive clusters is the results from the hard clustering.

Let us notate the membership of a particular data item to a cluster. The clusters are notated above under assumption of their crisp sets. Each cluster is assumed as a fuzzy set, and the membership of the data item,  $\mathbf{x}$ , in the cluster,  $C_i$ , is notated by  $\mu_{C_i}(\mathbf{x})$ . The membership,  $\mu_{C_i}(\mathbf{x})$ , is given as a normalized value between zero and one. The  $M$  membership values,  $\mu_{C_1}(\mathbf{x}), \mu_{C_2}(\mathbf{x}), \dots, \mu_{C_M}(\mathbf{x})$ , are given to the data item,  $\mathbf{x}$ .

Let us notate the item-cluster membership matrix as the results from clustering data items by the EM algorithm. In the frame of matrix, each row is a data item, each column is a cluster, and each element is a cluster membership. The memberships,  $\mu_{C_1}(\mathbf{x}), \mu_{C_2}(\mathbf{x}), \dots, \mu_{C_M}(\mathbf{x})$ , are assigned to each item,  $\mathbf{x}_j$ . The  $N \times M$  matrix which is expressed as Eq.(3.13) is constructed as the item-cluster membership matrix:

$$\begin{pmatrix} \mu_{C_1}(\mathbf{x}_1) & \mu_{C_2}(\mathbf{x}_1) & \dots & \mu_{C_M}(\mathbf{x}_1) \\ \mu_{C_1}(\mathbf{x}_2) & \mu_{C_2}(\mathbf{x}_2) & \dots & \mu_{C_M}(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \mu_{C_1}(\mathbf{x}_N) & \mu_{C_2}(\mathbf{x}_N) & \dots & \mu_{C_M}(\mathbf{x}_N) \end{pmatrix}. \quad (3.13)$$

The item-cluster membership matrix is continually updated until its convergence as the clustering process of EM algorithm.

Let us make the notations which are involved in EM algorithm. A cluster is viewed as a set of data items, and the list of clusters is the results from clustering data items. In using the EM algorithm, the current clustering task is assumed as the fuzzy clustering, and the membership of each data item in a cluster is defined as a normalized value between zero and one. The item-cluster membership matrix is defined as the results from the fuzzy clustering, and the item-cluster membership matrix is updated as the process of clustering data items. The vector which consists of the  $M$  cluster memberships which are assigned to each data item is viewed as its hidden representation.

### 3.4.3 E-Step

This section is concerned with the E-step (estimation step) in the EM algorithm. The symbols which are involved in explaining the EM algorithm are notated in the previous section. In this section, we explain the process of estimating cluster memberships as the E-step using the notated symbols. The initial M-step for initializing the parameters of the probability distribution is required for proceeding

the E-step. This section is intended to describe the E-step as the process of estimating the cluster memberships for each data item.

Let us mention the initial M-step (initial maximization step) in the EM algorithm before explaining the E-step. A continuous probability distribution is assumed as a normal distribution for each cluster. The normal distribution is characterized with its mean vector,  $\mu$ , and its covariance matrix,  $\Sigma$ . As the initial M-step, the mean vectors are initialized at random or based on the prior knowledge, as  $\mu_1, \mu_2, \dots, \mu_M$ , to the  $M$  clusters, and the covariance matrix is fixed as the identity matrix. The M-step is viewed as the process of estimating the mean vectors for maximizing the likelihoods of the individual data items to the categories.

The cluster memberships which are assigned to a data item is illustrated in Fig. 3.20. It is assumed that a normal distribution is defined to each cluster with its only mean vector. Based on the normal distribution, the likelihood of the input vector,  $\mathbf{x}$ , to the cluster,  $C_i$ , is computed by Eq. (3.14):

$$f(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^d |\Sigma|}} \exp\left(-\frac{1}{2}(\mathbf{x} - \mu)^T \Sigma^{-1} (\mathbf{x} - \mu)\right) \quad (3.14)$$

The likelihood of the input vector,  $\mathbf{x}$ , to the cluster,  $C_i$ , is defined the cluster membership, as shown in Eq. (2.13):

$$\mu_{C_i}(\mathbf{x}) = P(\mathbf{x}|C_i). \quad (3.15)$$

The  $M$  cluster memberships,  $\mu_{C_1}(\mathbf{x}), \mu_{C_2}(\mathbf{x}), \dots, \mu_{C_M}(\mathbf{x})$ , are assigned to the data item,  $\mathbf{x}$ .

In the EM algorithm, the E-step is to estimate or update the item-cluster membership matrix. It was mentioned and notated in Sect. 3.4.2. The data items in the group as clustering targets are notated by  $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N$ , and for each data

	Cluster 1	Cluster 2	...	Cluster M
Item 1	Membership 11	Membership 12	...	Membership 1M
Item 2	Membership 21	Membership 22	...	Membership 2M
...	...	...	...	...
Item N	Membership N1	Membership N2	...	Membership NM

Fig. 3.20 Estimation of item-cluster memberships in E-step

item,  $\mathbf{x}_i$ , the  $M$  cluster memberships,  $\mu_{C_1}(\mathbf{x})$ ,  $\mu_{C_2}(\mathbf{x})$ ,  $\dots$ ,  $\mu_{C_M}(\mathbf{x})$ , are computed. As the item-cluster membership matrix, the  $N \times M$  matrix is constructed as shown in Eq. (3.13). The item-cluster membership matrix is the results from clustering data items by the EM algorithm.

Let us make some remarks on the E-step for estimating the cluster memberships of each data item. We need the initial M-step which defines the initial probability distributions to the clusters. The E-step is the process of estimating the cluster memberships for each data item based on the probability distributions. The goal of E-step is to construct the item-cluster membership matrix. We consider defining multiple probability distributions for each cluster or defining a single probability distribution to multiple clusters.

### 3.4.4 M-Step

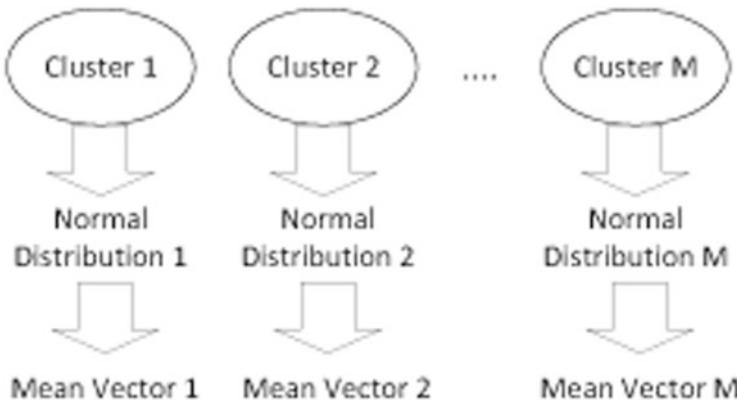
This section is concerned with the M-step which is the process of updating the parameters. In the previous section, we studied the E-step where the cluster memberships are estimated for each data item. The M-step is to update the parameters of the probability distributions which are defined over the clusters for maximizing the likelihoods of entire data items with the clusters. The data items are clustered alternatively by the E-step and the M-step in the EM algorithm. This section is intended to describe the M-step of EM algorithm.

The assumption of each cluster as a normal distribution is illustrated in Fig. 3.21. The normal distribution is characterized by its mean vector and its covariance matrix. The covariance matrix is assumed as the identity matrix, and the normal distribution is characterized by its only mean vector for reducing the computation complexity. The number of clusters is  $M$ , and each cluster corresponds to its own mean vector. The mean vectors are estimated from the clusters as the M-step in the EM algorithm.

Let us explain the process of computing the mean vectors as the M-step. The item cluster membership matrix which is the results of E-step is shown in Eq. (3.13). The mean vector which characterizes the normal distribution over the cluster,  $C_i$ , is estimated by Eq. (3.16):

$$\boldsymbol{\mu}_i = \frac{\sum_{j=1}^{|C_i|} \mu_{C_i}(\mathbf{x}_j) \mathbf{x}_j}{\sum_{j=1}^{|C_i|} \mu_{C_i}(\mathbf{x}_j)} \quad (3.16)$$

The column vector which corresponds to a cluster in Eq. (3.13) is involved in computing the mean vector. The estimated mean vectors are used for computing the cluster memberships for each vector in the E-step.



**Fig. 3.21** Mean vector estimation in M-step

Let us consider estimating the covariance matrix in the M-step. The diagonal elements which are attribute variances are computed by Eq. (3.17):

$$\sigma_{ik}^2 = \frac{\sum_{j=1}^{|C_i|} \mu_{C_i}(\mathbf{x}_j) (\mu_{jk} - x_{jk})^2}{\sum_{j=1}^{|C_i|} \mu_{C_i}(\mathbf{x}_j)}, \quad (3.17)$$

where  $i$  is the cluster index;  $j$  is the data item index in the cluster,  $i$ ; and  $k$  is the element index in the data item,  $j$ . The off-diagonal elements in the covariance between the  $k$ th element and the  $m$ th element are computed by Eq. (3.18):

$$\sigma_{ik}^2 = \frac{\sum_{j=1}^{|C_i|} \mu_{C_i}(\mathbf{x}_j) (\mu_{jk} - x_{jk})(\mu_{jm} - x_{jm})}{\sum_{j=1}^{|C_i|} \mu_{C_i}(\mathbf{x}_j)}, \quad (3.18)$$

where  $k$  and  $m$  are the element indexes in a data item. The covariance matrix which characterizes the normal distribution over the cluster,  $C_i$ , is constructed as Eq. (3.19):

$$\boldsymbol{\Sigma}_i \begin{pmatrix} \sigma_{i11}^2 & \sigma_{i12}^2 & \dots & \sigma_{i1d}^2 \\ \sigma_{i21}^2 & \sigma_{i22}^2 & \dots & \sigma_{i2d}^2 \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{id1}^2 & \sigma_{id2}^2 & \dots & \sigma_{idd}^2 \end{pmatrix}. \quad (3.19)$$

For simplicity, the process of computing the covariance matrix is omitted in the M-step.

Let us make some remarks on the M-step in the EM algorithm. It is the process of updating the parameters which characterize the continuous probability distributions which are defined to the clusters for maximizing the likelihoods of the data items to them. It is assumed that a normal distribution is defined to each cluster, and its mean vector is updated by averaging its members. The covariance matrices are updated as well as mean vectors for characterizing the normal distributions. We consider defining various probability distributions such as fuzzy distributions to the clusters.

### 3.5 Summary and Further Discussions

Let us summarize what is studied in this chapter. Simple approaches to the data clustering are the AHC algorithm, the k means algorithm, and the divisive algorithm. The Kohonen Networks are the unsupervised neural networks, and its variants are the LVQ, semi-supervised version, and SOM. The EM algorithm is the advanced frame of clustering data items with the E-step and the M-step, and its various versions exist depending on how to define the distribution for each cluster. This section is intended to discuss further what is studied in this chapter.

We may consider the combination of the two simple clustering algorithms, the AHC algorithm and the divisive algorithm. The AHC algorithm clusters data items in the bottom-up direction, whereas the divisive algorithm clusters data items in the top-down direction. The results from clustering data items by both are integrated by voting for deciding whether a pair of data items belongs to a same cluster or different clusters, as an independent combination. While clustering data items by the AHC algorithm, the divisive algorithm may be applied to clusters with their relatively weak cohesion, and while clustering data items by the divisive algorithm, the AHC algorithm may be applied to two clusters with their relative weak discriminations. Various schemes of combining both exist.

Let us mention a variant of Kohonen Networks, called neural gas, as an unsupervised neural model. Its role is to define a feature map to the input vectors, like the SOM. The difference of the neural gas from the SOM is to define the restraint to the distance by an exponential function. As the learning continues, both the learning rate and the restraint to the distance decrease. The shared point between the two neural models is that learning proceeds based on the difference between the weight vector and the input vector.

The semi-supervised learning was implemented as an approach to the text classification by combining the EM algorithm with the Naive Bayes. We prepare the two sets of training examples: one set consists of labeled examples, and the other consists of unlabeled examples. By using the EM algorithm, the initial clusters are constructed by the labeled examples, and the labels are assigned to the unlabeled examples by clustering them. A novice item is classified by using the Naive Bayes

which are trained with both sets of training examples. The semi-supervised learning is intended to utilize unlabeled examples which are obtained easily, for improving the generalization performance.

Let us consider the semi-supervised learning which is distinguished from both the supervised learning and the unsupervised learning. It was motivated by the real situation where it is easy to obtain unlabeled examples. It is intended to improve the generalization performance by utilizing unlabeled examples as well as labeled examples. The schemes of implementing it is to modify existing unsupervised learning algorithms and to combine a supervised learning algorithm with an unsupervised learning algorithm. The semi-supervised learning is applied to the classification and the regression like the supervised learning.

## Reference

1. T. Jo, “Semantic Word Categorization using Feature Similarity based K Nearest Neighbor”, 67–78, Journal of Multimedia Information Systems, 2018.

# Chapter 4

## Ensemble Learning



This chapter is concerned with the ensemble learning as an alternative advanced learning to the deep learning. It refers to the learning type for solving the problems such as classification, regression, and clustering, more reliably by combining multiple machine learning algorithms with each other. The typical schemes of ensemble learning are the voting which is the process of deciding the final answer by considering ones of multiple machine learning algorithms, the expert gate which is the process of nominating one among the machine learning algorithms as an expert which is suitable for solving the given input, and the cascading which is the process of deciding whether the current answer is adopted or the input is transferred to the next machine learning algorithm. We need to consider partitioning the training set which is viewed as a matrix with the two axes: the horizontal partition which partitions the training set into subsets and the vertical partition which partitions the attribute set. This chapter is intended to describe the ensemble learning as an advanced type of advanced learning.

This chapter is organized into the five sections, and in Sect. 4.1, we describe the ensemble learning conceptually. In Sect. 4.2, we describe the partition of the training set and the attribute set as the preparation for the ensemble learning. In Sect. 4.3, we describe the schemes of combining multiple machine learning algorithms. In Sect. 4.4, we describe the multiple viewed learning as the additional advanced learning type. In Sect. 4.5, we summarize the entire contents of this chapter and discuss further on what is studied in this chapter.

### 4.1 Introduction

This section is concerned with the ensemble learning in the conceptual view. The goal of ensemble learning is to overcome the limit of a machine learning algorithm with respect to its learning and its generalization. The ensemble learning involves multiple machine learning algorithms in the learning and the generalization for

improving the generalization performance. In implementing the ensemble learning, we choose either of training each machine learning algorithm with the entire set of training examples and training it with its own subset. This section is intended to describe the ensemble learning for providing its introduction conceptually.

Multiple machine learning algorithms are involved in implementing the ensemble learning. In the classical learning, only one machine learning algorithm is used; its demerits and its bias toward a particular output always exist in all kinds of machine learning algorithms. The ensemble learning is intended to overcome the limits in using a single machine learning algorithm by involving multiple machine learning algorithms. There are two kinds of ensemble learning: the homogenous ensemble learning which combines an identical kind of machine learning algorithms with their different external parameters and the heterogenous ensemble learning which combines different kinds of machine learning algorithms. The coordinator is needed as a separated module for controlling the combination of machine learning algorithms.

Let us mention the three schemes of combining multiple machine learning algorithms with each other for implementing the ensemble learning. The voting is the organization of multiple machine learning algorithms where the coordinator makes the final decision by collecting their answers. The expert gate is the organization where one is nominated among multiple machine learning algorithms for making the final decision. The cascading is the organization where the controller decides whether the answer from the current machine learning algorithm is adopted or the novice input is transferred to the next machine learning algorithm. Multiple machine learning algorithms may be combined by a hybrid scheme by mixing the three schemes.

The process of partitioning the training set into subsets is necessary before implementing the ensemble learning, depending on application areas, and we mention the three schemes of doing so. The exclusive partition is the process of partitioning it without any overlapping among subsets. The overlapping partition is the process of partitioning it, allowing overlapping between subsets. The fuzzy partition is the process of assigning membership values of subsets to each training example. We consider partitioning the training set by clustering training examples regardless of their labels.

Let us mention what is intended in this chapter. We study the schemes of partitioning the training set or the attribute set into subsets as the preparation for implementing the ensemble learning. We study the scheme of organizing multiple machine learning algorithms with each other for deciding the final answer. We study the multiple viewed learning as a variant which is derived from the ensemble learning with multiple views of observing the raw training examples. This chapter is intended to study the ensemble learning and its variant as another branch from the swallow learning.

## 4.2 Partition

This section is concerned with the partition of the training set and the attribute set as the preparation for the ensemble learning. In Sect. 4.2.1, we mention the partition of the training set into subsets. In Sect. 4.2.2, we mention the partition of the attribute set into subsets. In Sect. 4.2.3, we mention the two-dimensional array partition with viewing the training set as a matrix. In Sect. 4.2.4, we mention the schemes of partitioning the training set and/or the attribute set.

### 4.2.1 Training Set

This section is concerned with the training set as the target which is partitioned into subsets. We prepare the training examples, each of which is labeled with its own output. The training set is partitioned into subsets at random, and each machine learning algorithm is trained with its own subset in applying the ensemble learning. In an advanced way, the training set is partitioned by clustering data items. This section is intended to describe the partition of the training set into subsets for applying the ensemble learning.

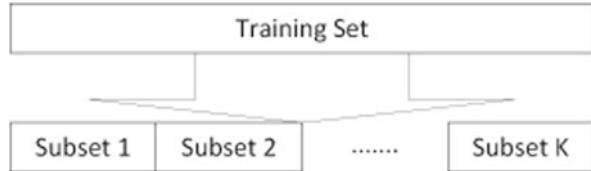
The training examples which are used for a machine learning algorithm is illustrated in Fig. 4.1. Each training example is given as a  $d$  dimensional numerical vector. The set which consists of the training examples, called training set, is notated by  $Tr = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$ . Each element in the vector which represents a training example is given as a continuous value, and each training example is notated by  $\mathbf{x}_i = [x_{i1} \ x_{i2} \ \dots \ x_{id}]$ . The dimension of numerical vector is assumed to fixed to  $d$ , and an attribute called feature corresponds to an element.

The partition of the training set into subsets is illustrated in Fig. 4.2. The training set,  $Tr$ , is portioned into subsets,  $Tr_1, Tr_2, \dots, Tr_K$ . Each subset,  $Tr_i$ , is defined by  $Tr_i = \{\mathbf{x}_{i,1}, \mathbf{x}_{i,2}, \dots, \mathbf{x}_{i,|Tr_i|}\}$ , and the relation of the subset,  $Tr_i$ , with the set,  $Tr$ , is  $Tr_i \subseteq Tr$ . The exclusive partition is expressed by  $\forall_{i,j} Tr_i \cap Tr_j = \emptyset$ , whereas the overlapping partition is expressed by  $\exists_{i,j} Tr_i \cap Tr_j \neq \emptyset$ . The training set is partitioned by selecting some training examples at random; this partition scheme is called random partition.

**Fig. 4.1** Training set

$$\begin{aligned}\mathbf{x}_1 &= [x_{11} \quad x_{12} \quad \dots \quad x_{1d}] \\ \mathbf{x}_2 &= [x_{21} \quad x_{22} \quad \dots \quad x_{2d}] \\ &\vdots \\ \mathbf{x}_N &= [x_{N1} \quad x_{N2} \quad \dots \quad x_{Nd}]\end{aligned}$$

**Fig. 4.2** Training set partition



**Fig. 4.3** Clustering-based partition

```
List partitionTrainingSet(List trainingSet,int partitionNumber){  
    List partitionList = clusterDataItemList(trainingSet,partitionNumber);  
    return partitionList;  
}
```

The partition of the training set by clustering the training examples is illustrated in Fig. 4.3. The examples in the training set,  $Tr$ , are clustering targets. The clusters,  $C_1, C_2, \dots, C_K$ , are generated by clustering the training examples regardless of their labels, depending on their dependences. Each cluster is a subset of the training set as expressed in Eq. (4.1):

$$C_i \rightarrow Tr_i \quad (4.1)$$

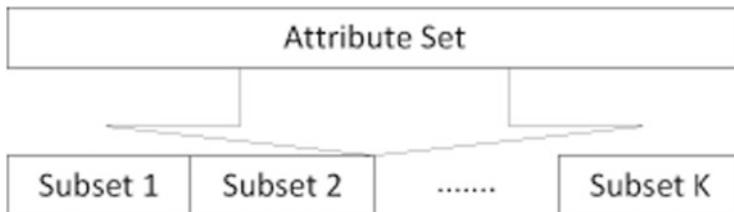
A tiny cluster which is a subset with a small number of training examples becomes an issue in this type of partition.

Let us make some remarks on the partition of the training set into subsets for implementing the ensemble learning. It is assumed that the training set consists of the  $N$  training examples, each of which is given as a  $d$  dimensional vector. The training set is portioned into subsets by selecting some training examples at random. It may be partitioned into subsets by clustering the training examples by an unsupervised learning algorithm. It is optional to partition the training set into subsets for implementing the ensemble learning.

#### 4.2.2 Attribute Set

This section is concerned with the partition of the attribute set into subsets. In the previous section, we studied the partition of the training set into subsets for implementing the ensemble learning. The training set is viewed as a set of attribute values, column by column. The set of attribute values is partitioned into subsets, each of which is called attribute subset, and we study the cross of a training subset and an attribute subset, in the subsequent section. This section is intended to describe the partition of the attribute set into subsets.

Let us mention the attribute set which is partitioned into subsets for implementing the ensemble learning. In the previous section, we studied the training set which consists of the training examples, and it is partitioned into subsets of training examples. The attribute set is notated by a set of attribute vectors, each of which has  $N$  attribute values,  $A = \{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_d\}$ , where  $\mathbf{a}_i = [a_{i,1} \ a_{i,2} \ \dots \ a_{i,N}]$  is a  $N$



**Fig. 4.4** Attribute set partition

**Fig. 4.5** Attribute clustering

```
list partitionAttributeSet(list attributeSet, int partitionNumber){  
    List partitionList = clusterAttributeList(attributeSet,partitionNumber);  
    return partitionList;  
}
```

dimensional numerical vector. The elements in the  $N$  dimensional attribute vector is interpreted as  $a_{i,j} = x_{j,i}$  and  $\mathbf{a}_i = [x_{i,1} \ x_{i,2} \ \dots \ x_{i,N}]$ . The training set which is covered in the previous section is viewed with the association of the attribute set as a matrix.

The partition of the attribute set into subsets is illustrated in Fig. 4.4. The attribute set,  $A$ , is partition into subsets,  $A_1, A_2, \dots, A_K$ , where  $A_i \subseteq A$ . Each subset,  $A_i$ , is defined as a set,  $A_i = \{\mathbf{a}_{i,1}, \mathbf{a}_{i,2}, \dots, \mathbf{a}_{i,|A_i|}\}$ . If the ensemble learning which consists of  $K$  machine learning algorithms is implemented, the machine learning algorithm,  $ML_i$ , is trained with the subset,  $A_i$ . The partition of the training set into subsets is called horizontal partition, whereas the partition of the attribute set into subsets is called vertical partition, in viewing the training set into a matrix.

The process of clustering attribute values which is called transposed clustering is illustrated in Fig. 4.5. The columns which correspond to the attribute values are extracted from the training set which is viewed as a matrix. Each attribute which is given as a  $N$  dimensional vector, and the  $d$  attribute values exist in training set. A group of the  $d$  attribute values is segmented into subgroups, each of which contains similar attribute values. Subsets of attribute values are obtained by clustering the  $d$  columns of the training set.

Let us make some remarks on the partition of the attribute set into subsets, called vertical partition. If each training example is assumed as a  $d$  dimensional vector, the  $d$  attributes are defined. The attribute set is partitioned into subsets, and each subset is used for representing the data items. A training set is viewed as a matrix, and column vectors are clustered, called transpose clustering. We consider crossing the training set partition and the attribute set partition as the two-dimensional partition.

#### 4.2.3 Array Partition

This section is concerned with the two-dimensional array partition of the training set. In the previous sections, we studied the training set partition and the attribute

**Fig. 4.6** View training set into matrix

	Attribute 1	Attribute 2	Attribute d	
Example 1	$x_{11}$	$x_{12}$	$\dots$	$x_{1d}$
Example 2	$x_{21}$	$x_{22}$	$\dots$	$x_{2d}$
Example N	$\dots$	$\dots$	$\dots$	$\dots$
	$x_{N1}$	$x_{N2}$	$\dots$	$x_{Nd}$

set partition as the preparation for implementing the ensemble learning. The training set is viewed as a matrix, and it is portioned in both the horizontal direction and the vertical direction into subsets. Each subset is viewed as the cross of the training subset and the attribute subset. This section is intended to describe the two-dimensional array partition of the training set.

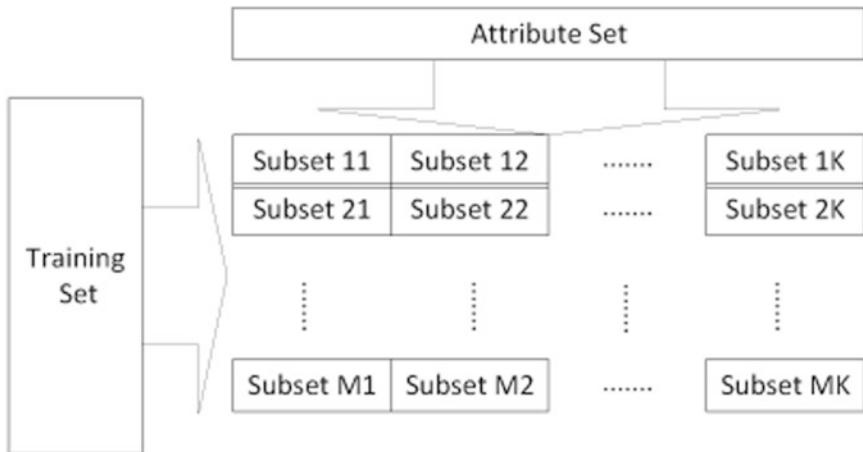
The training set is viewed as a matrix in Fig. 4.6. It is assumed that each training example is given as a  $d$  dimensional vector, and the size of the training set is  $N$ . In the matrix, each of row vectors,  $\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_N$ , is a  $d$  dimensional vector as a training example, and each of column vectors,  $\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_d$ , is a  $N$  dimensional vector as an attribute vector. Each row vector and each column vector correspond, respectively, to a training example and an attribute vector:  $\mathbf{r}_i = \mathbf{x}_i$  and  $\mathbf{c}_i = \mathbf{a}_i$ . The training set is expressed as a  $N \times d$  matrix.

The two-dimensional partition of the training set is illustrated in Fig. 4.7. The training set,  $Tr$ , is partitioned into subsets,  $Tr_1, Tr_2, \dots, Tr_M$ , and the attribute set,  $A$ , is portioned into subsets,  $A_1, A_2, \dots, A_K$ . The cross of  $Tr_i$  and  $A_j$  is notated by  $Tr_{ij}$  which is set of examples in the subset,  $Tr_i$ , with the attributes in the set,  $A_j$ . The grid of subsets is given as a matrix as expressed in Eq. (4.2):

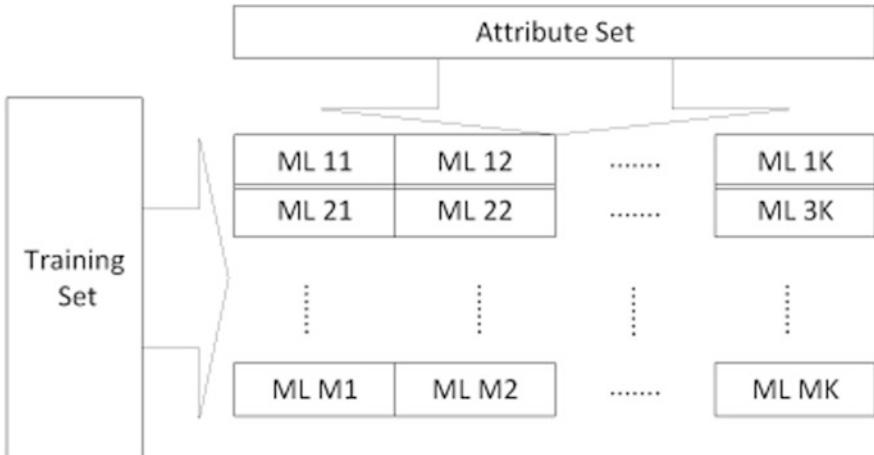
$$\begin{pmatrix} Tr_{11} & Tr_{12} & \dots & Tr_{1K} \\ Tr_{21} & Tr_{22} & \dots & Tr_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ Tr_{M1} & Tr_{M2} & \dots & Tr_{dK} \end{pmatrix}. \quad (4.2)$$

The subset,  $Tr_{ij}$ , is used for training the machine learning,  $ML_{ij}$ .

The cellular learning which is expanded from the ensemble learning is illustrated in Fig. 4.8. It involves a massive number of simple machine learning algorithms in learning the training examples and generalizing novice examples as a grid form. The process of cellular learning is to partition the training set into two-dimensional array subsets by the above process and to allocate each subset to machine learning algorithm for training it. In the generalization process, the relevancy of each machine learning algorithm to a novice input vector should be computed, and only relevant machine learning algorithms participate in doing the classification or the



**Fig. 4.7** Two-dimensional array partition



**Fig. 4.8** Two-dimensional cellular learning

regression. The idea of cellular learning is to solve the complicated problems using a massive number of simple machine learning algorithms.

Let us make some remarks on the two-dimensional array of the training set which is covered in this section. The training set which consists of  $N$  numerical vectors with their  $d$  dimensionality is viewed into a  $N \times d$ , matrix where its columns correspond to the attribute values, and its rows correspond to the training examples. The row set is partitioned into  $M$  subsets, and the column set is partitioned into  $K$  subsets, and a subset is defined by cross of a row subset and a column subset. In the cellular learning,  $M \times K$  machine learning algorithms are involved, and each subset

is allocated for train each machine learning algorithm. The two ensemble learning schemes, voting and expert gate, are applied to the cellular learning.

#### 4.2.4 Partition Schemes

This section is concerned with the schemes of partitioning a set into subsets. In the previous sections, we studied the direction of partitioning the training set into subsets. In this section, we consider various ways of partitioning a set into subsets. Even if it is assumed that the training set is partitioned before training a machine learning algorithm, the partition may be considered during training a machine learning algorithm. This section is intended to describe the typical schemes of partitioning a set into subsets.

The random partition of the training set into subsets as the first scheme is illustrated in Fig. 4.9. The number of subsets and their sizes are decided in advance. For each subset, a training example is selected at random and is included in the subset. The random selection with its restoration causes the overlapping partition, whereas one without restoration causes the exclusive partition. Each partition is used for training a machine learning algorithm in the ensemble learning.

The partition of the training set by clustering the training examples is illustrated in Fig. 4.10. The clustering is the process of segmenting a group of data items into subgroups each of which contains similar ones. The training examples are clustered by an unsupervised learning algorithm, depending on their similarities regardless of their labels. Several subgroups of data items are results from doing so and become subsets of training examples. Each cluster is used for training its corresponding machine learning algorithm.

The overlapping partition of the training set into subsets is illustrated in Fig. 4.11. We mentioned above the exclusive partition where no overlapping is allowed



**Fig. 4.9** Random training set partition

**Fig. 4.10** Clustering-based training set partition



**Fig. 4.11** Fuzzy training set partition



between subsets. The overlapping partition is one where any overlapping is allowed between subsets; any training example exists in the intersection of two subsets. The overlapping partition is expanded into the fuzzy partition where a membership of each training example is given as a continuous value between zero and one. In the overlapping partition, a membership is given as zero or one; in the fuzzy partition, it is given as a continuous value between zero and one.

Let us make some remarks on the schemes of partitioning a set into subsets. The random partition is to partition a set into subsets by selecting an element at random, as the simplest scheme. The more advanced partition scheme is the clustering-based partition where elements are clustered based on their similarities, and each cluster is a subset. There are the three partition types: exclusive partition, overlapping partition, and fuzzy partition. The boosting where the training set is partitioned into subsets by observing the training error in the set is the interactive partition scheme with the machine learning algorithm.

### 4.3 Supervised Combination Schemes

This section is concerned with the schemes of combining multiple machine learning algorithms for making the final output. In Sect. 4.3.1, we mention the voting

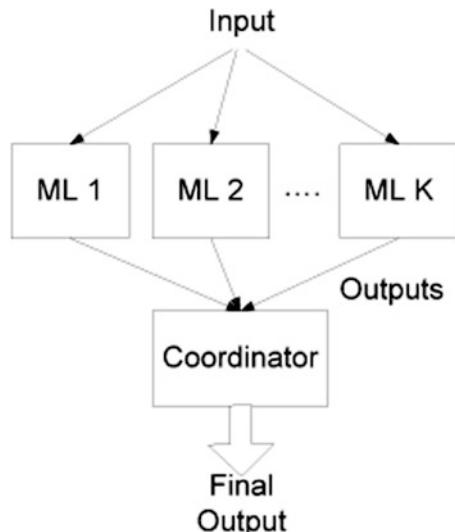
which makes the final output by collecting outputs of multiple machine learning algorithms. In Sect. 4.3.2, we mention the expert gate which nominates a particular machine learning algorithm for making the final output. In Sect. 4.3.3, we mention the cascading which organizes multiple machine learning algorithms, serially. In Sect. 4.3.3, we expand the ensemble learning into the cellular learning.

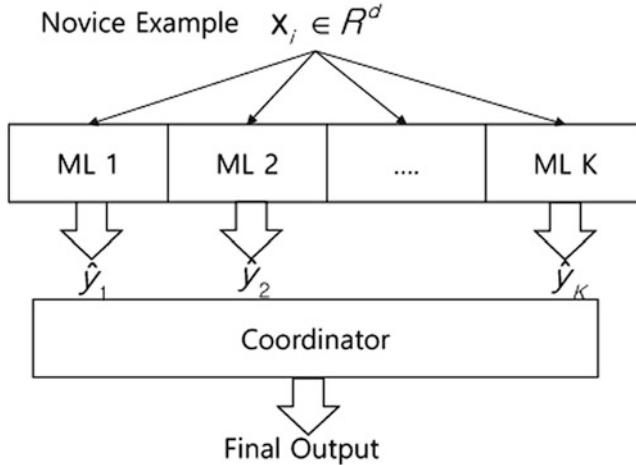
### 4.3.1 Voting

This section is concerned with an ensemble learning scheme, called voting. We studied the partition of the training set into subsets as the preparation for the ensemble learning in the previous sections and study the schemes of ensemble learning from this section. Multiple machine learning algorithms called committee members and the coordinator are involved in the voting. The role of the committee members is to learn the training examples and classify a novice data item, and the role of the coordinator is to decide the final answer. This section is intended to describe the voting as an ensemble learning scheme.

The organization of machine learning algorithms in the voting is illustrated in Fig. 4.12. The  $K$  machine learning algorithms and a single coordinator are involved in this organization. The role of each machine learning algorithm is to receive the input vector and generate their own output vector. The role of the coordinator is to receive the  $K$  output vector from the  $K$  machine learning algorithms and decide the final output. The  $K$  machine learning algorithms participate in the classification and the regression in this organization.

**Fig. 4.12** Machine learning organization in voting





**Fig. 4.13** Classification process in voting

The process of estimating the output value by the ensemble learning scheme called voting is illustrated in Fig. 4.13. The given task is assumed as a univariate regression for explaining it easily. The output values,  $\hat{y}_1, \hat{y}_2, \dots, \hat{y}_K$ , are estimated by the  $K$  machine learning algorithms. The final output,  $\hat{y}$ , is decided by averaging the output values by the coordinator, as shown in Eq. (4.3):

$$\hat{y} = \frac{1}{K} \sum_{i=1}^K \hat{y}_i. \quad (4.3)$$

The final of the novice input is decided by voting the categories which are decided by the  $K$  machine learning algorithms in the classification task.

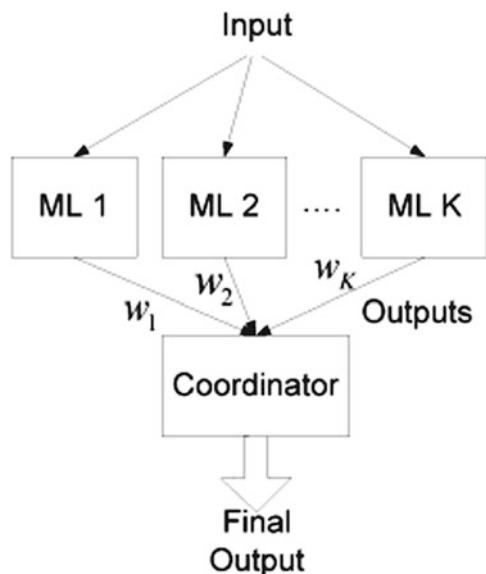
The weighted voting as a variant which is derived from the voting is illustrated in Fig. 4.14. If the  $K$  machine learning algorithms are given as the committee members, the identical weight are assigned to each member for voting the output values in the above voting. The performances of the committee members are variable, so they should be discriminated by assigning different weights to them in voting their answers. The output values are voted by Eq. (4.4):

$$\hat{y} = \frac{1}{K} \sum_{i=1}^K w_i \hat{y}_i. \quad (4.4)$$

where  $\sum_{i=1}^K w_i = 1.0$ . How to estimate the weights becomes the issue in this voting type.

Let us make some remarks on the voting as a scheme of combining multiple machine learning algorithms. Multiple machine learning algorithms which are

**Fig. 4.14** Machine learning organization in weighted voting



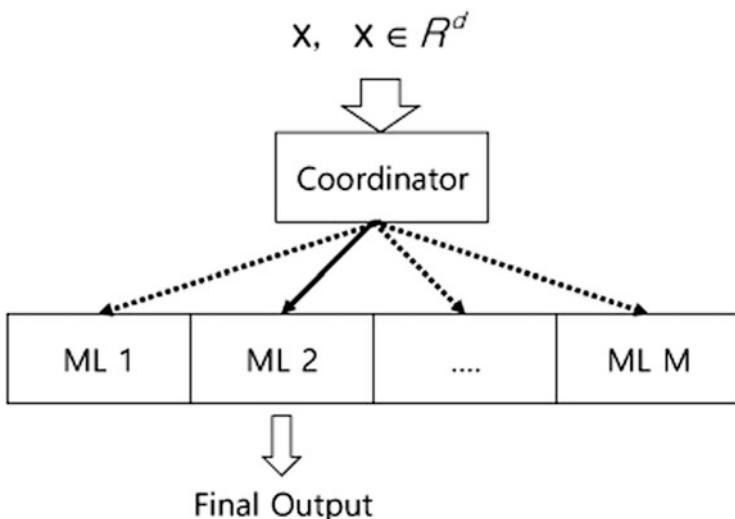
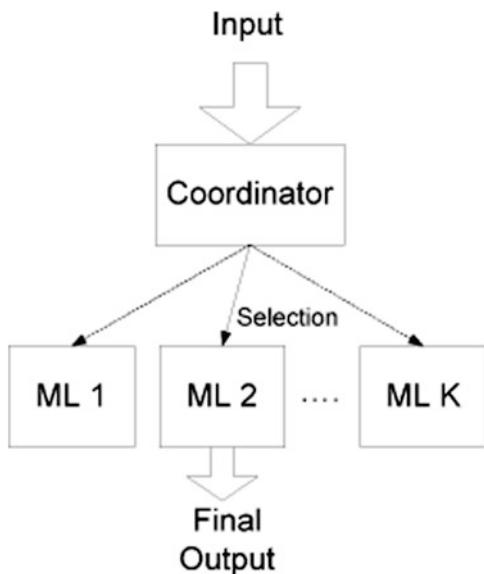
called committee members and the coordinator are involved in implementing the ensemble learning scheme. In the generalization process, the committee members generate their own answers, and the coordinator decides the final answer. The committee members may be discriminated by assigning different weights to them, in voting their answers. We may consider the meta learning which is used for deciding the final answer from the collected answers.

### 4.3.2 Expert Gate

This section is concerned with the scheme of combining multiple machine learning algorithms which is called expert gate. In the previous section, we studied the voting which decides the final answer by collecting answers from multiple machine learning algorithms. In this section, we study the expert gate which nominates the expert to the novice input and decides the final answer by the answer from the nominated one. The  $K$  machine learning algorithms which are called experts and the coordinator are involved in this scheme. This section is intended to describe the expert gate as the scheme of combining multiple machine learning algorithms.

The machine learning algorithms and the coordinator which are involved in implementing the expert gate are illustrated in Fig. 4.15. In the voting which was covered in Sect. 4.3.1, the coordinator decides the final answer by voting the answers of the committee members. In the expert gate, the coordinator is replaced by the coordinator, and it nominates a particular machine learning algorithm. Multiple machine learning algorithms which are involved in the expert gate are called experts,

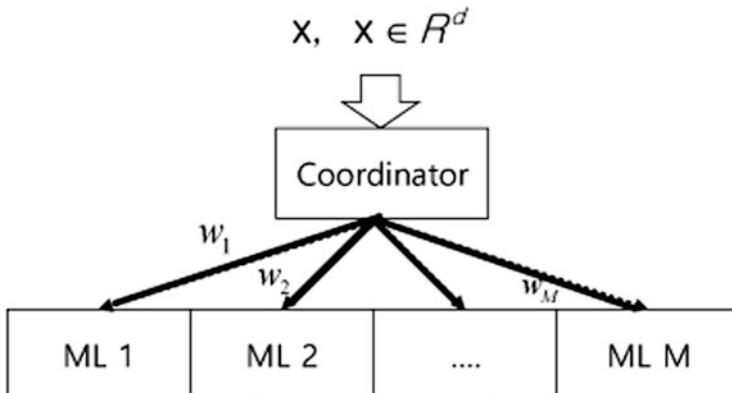
**Fig. 4.15** Machine learning organization in expert gate



**Fig. 4.16** Classification process in expert gate

and only one among them participates in making the final decision. Nomination of a particular machine learning algorithm means that the expert gate is opened.

The process of generalizing the input vector in the expert gate is illustrated in Fig. 4.16. A novice input is given as a  $d$  dimensional vector, and the coordinator takes it as its input. Among the  $M$  machine learning algorithms, one is nominated, and the input vector is transferred to it. The output is generated from the nominated



**Fig. 4.17** Machine learning organization in weighted expert gate

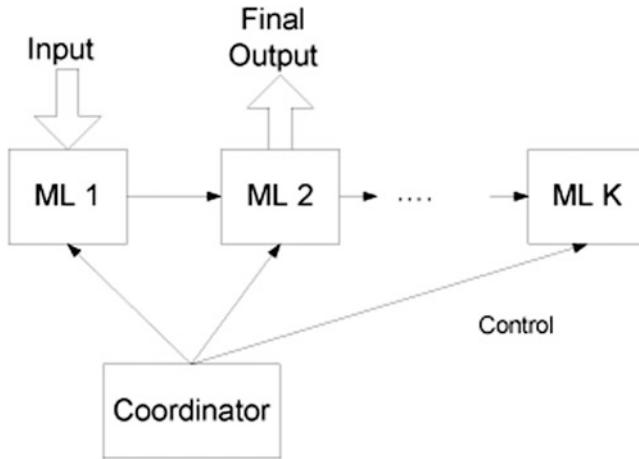
machine learning algorithm and decided as the final output. The final output is decided by the nominated machine learning algorithm as the expert in this ensemble learning scheme.

The weighted nomination of a machine learning algorithm is illustrated in Fig. 4.17. In the original version of the expert gate, only one machine learning algorithm is nominated for classifying a novice input. In this variant, a weight is assigned to each machine learning algorithm, rather than nominating one. In the version, which is illustrated in Fig. 4.16, the weight, 1.0, is assigned to only one machine learning algorithm. In the weighted expert gate, the weights which are assigned to machine learning algorithms are variable for each novice input vector, whereas in the weighted voting, the weights are constant for any novice input vector.

Let us make some remarks on the expert gate which is applied to the supervised learning. Multiple machine learning algorithms and the coordinator are involved in implementing this ensemble learning scheme. A particular machine learning algorithm is nominated as the expert by the coordinator, and the final answer is decided by the output of the nominated one. We may consider deciding the final answer by assigning different weights to the experts, voting their answers with the weights. The weighted nomination is viewed as the combination of the voting and the expert gate.

### 4.3.3 Cascading

This section is concerned with the third scheme of ensemble learning, called cascading. In the previous sections, we already studied the two ensemble learning schemes, voting and expert gate. In this section, we study one more scheme of ensemble learning, cascading, where multiple machine learning algorithms are combined serially. In the previous ensemble schemes, machine learning algorithms



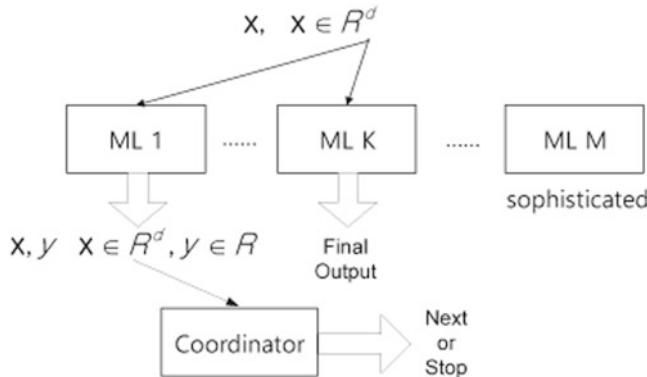
**Fig. 4.18** Machine learning organization in cascading

are independent of each other, whereas in this scheme, each machine learning algorithm depends on its previous one. This section is intended to describe the ensemble scheme, called cascading, which is different from the voting and the expert gate.

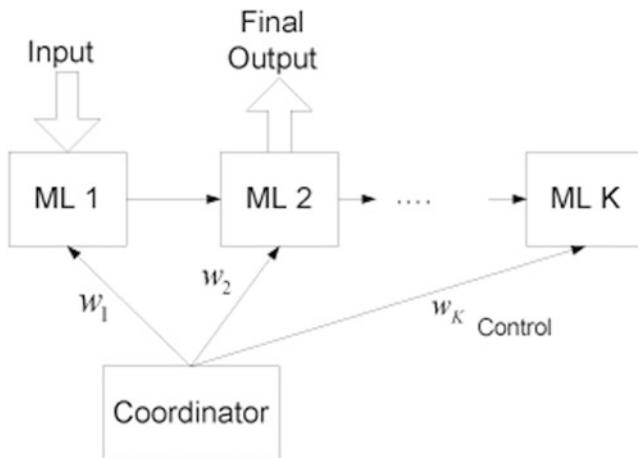
The organization of multiple machine learning algorithms and the coordinator for implementing the cascading is illustrated in Fig. 4.18. The  $K$  machine learning algorithms and a single coordinator are involved in implementing the ensemble learning scheme. The role of the coordinator is to decide whether to take the current answer as the final one or transfer the input vector to the next machine learning algorithm, depending on the certainty of the output value. The role of each machine learning algorithm is to learn the training examples and generalize a novice input. The  $K$  machine learning algorithms are organized serially; each machine learning algorithm depends on its previous one.

The process of making the final answer in the cascading is illustrated in Fig. 4.19. The  $M$  machine learning algorithms are arranged from the simplest one to the most advanced one serially, and the answer is taken from the first machine learning algorithm. The answer is taken from the current machine learning algorithm, and whether the answer is the final one or the next machine learning algorithm is visited is decided by the coordinator. If the answer is certain, the current answer is decided as the final one, and otherwise, the input vector is transferred to the next machine learning algorithm. Even if the number of machine learning algorithms which provide their answers is variable, only one machine learning algorithm participates in deciding the final answer.

The weighted cascading as a variant is illustrated in Fig. 4.20. In the initial version of cascading, only one machine learning algorithm participates in deciding the final answer. In this variant, previous machine learning algorithms are allowed to be involved in deciding the final answer, as well as the current one. After classifying



**Fig. 4.19** Classification process in cascading

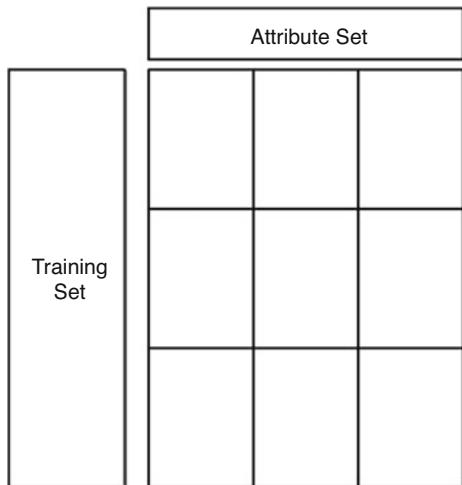


**Fig. 4.20** Machine learning organization in weighted cascading

a data item by the current machine learning algorithm, the weights are assigned to the previous ones and the current one by the coordinator. If identical weights are assigned to the previous machine learning algorithms, this variant is viewed as the mixture of the voting and the cascading.

Let us make some remarks on the ensemble learning scheme which is called cascading. Multiple machine learning algorithms and a coordinator are involved in implementing this ensemble learning scheme. The role of the coordinator is to decide whether the current answer is taken as the final one, or the input vector is transferred to the next machine learning algorithm, depending on the certainty of the current answer. We may consider answers of machine learning algorithms which are visited previously by assigning different weights to them. We consider

**Fig. 4.21** Two-dimensional array partition



the tournament of machine learning algorithms for deciding the final answer as one more ensemble learning scheme.

#### 4.3.4 Cellular Learning

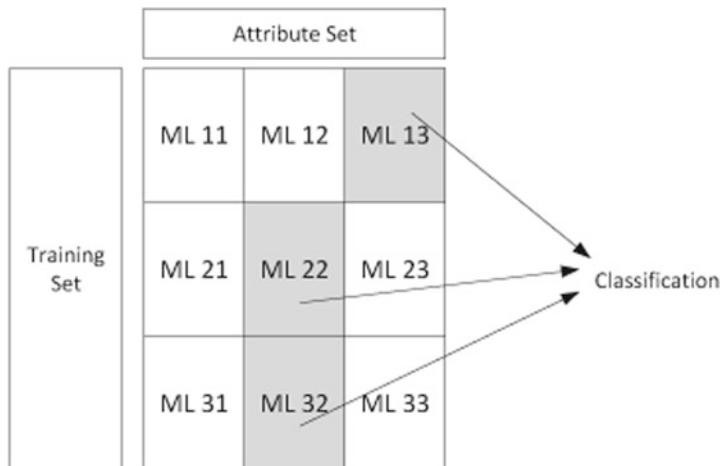
This section is concerned with the cellular learning which is expanded from the ensemble learning. In the previous section, we studied the three ensemble learning schemes, voting, expert gate, and cascading. In this section, the ensemble learning is expanded into the cellular learning, and the massive machine learning algorithms is involved as a grid. Both the training set and the attribute set are partitioned into crosses, called item-attribute subsets, and each of them is allocated to a machine learning algorithm. This section is intended to describe the cellular learning which is expanded from the ensemble learning.

The partition of the training set into subsets with the two axis is illustrated in Fig. 4.21. The training set is viewed as a matrix where each row is a training example and each column is a set of attribute values. The training set is partitioned into subsets of training examples horizontally and partitioned vertically into subsets of attributes. Each subset which is a cross of a training subset and an attribute subset consists of some training examples with their partial attributes and called item-attribute subset. The subset is used for training a machine learning algorithm.

The arrangement of machine learning algorithms for implementing the cellular learning is illustrated in Fig. 4.22. The training set is partitioned into item-attribute subsets, each of which consist of some training examples with some attributes by the above process. An item-attribute subset is allocated to each machine learning algorithm, and it is trained with its own subset. Each item-attribute subset

**Fig. 4.22** Two-dimensional array of machine learning algorithms

Attribute Set		
ML 11	ML 12	ML 13
ML 21	ML 22	ML 23
ML 31	ML 32	ML 33



**Fig. 4.23** Nomination of relevant machine learning algorithms for classification

corresponds to a machine learning algorithm. It is possible to train all machine learning algorithms the same with the parallel computing.

The involvement of machine learning algorithms in the cellular learning for classifying a data item is illustrated in Fig. 4.23. The machine learning algorithms are arranged into a grid, and each of them is trained with its own subset. A novice data item is given as the input, and the relevancy of each machine learning algorithm to the novice input is computed. Only relevant machine learning algorithms are involved in classifying the data item, and the ensemble learning is applied to the involved ones. Its advantage is that novice input vector is given with its incomplete attribute values.

Let us make some remarks on the cellular learning as the expansion of the ensemble learning. The partition which is called array partition is a cross of a subset of training examples and a subset of attributes. An item-attribute subset which consists of some training examples with some attributes is allocated to each machine learning algorithm. Some machine learning algorithms which are relevant to the novice item are nominated for classifying it. The cellular learning is suitable for the GPU environment as its advantage.

## 4.4 Multiple Viewed Learning

This section is concerned with the multiple viewed learning which observes the training examples with multiple views. In Sect. 4.4.1, we define the view conceptually. In Sect. 4.4.2, we specify the view into a process of encoding raw training examples into structured forms with its own scheme. In Sect. 4.4.3, we mention the supervised learning with multiple views. In Sect. 4.4.4, we mention the unsupervised learning with multiple views.

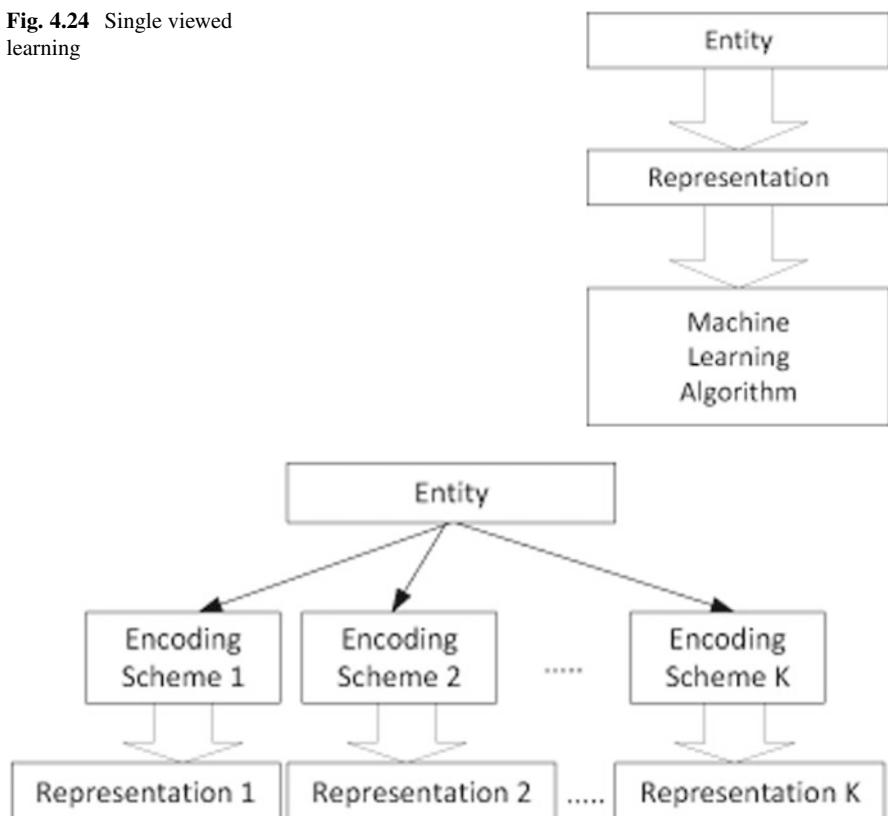
### 4.4.1 Views

This section is concerned with the concept of view in context of machine learning. The view is defined as the scheme of observing the training examples. The initial training examples are assumed to be given as raw data items, and more than one scheme of encoding them into structured ones may exist. Multiple sets of training examples which are represented differently in the raw data items may be prepared in the multiple viewed learning. This section is intended to describe the view which is needed for implementing the multiple viewed learning, conceptually.

The single viewed learning as the traditional style of machine learning is illustrated in Fig. 4.24. In the single viewed learning, only one scheme of encoding raw training examples into structured ones exists. The raw training examples are encoded into a single type of representations, and the machine learning algorithm is trained with a single set of the representations. In the generalization, a novice data item is encoded into only one type of representation, and the output is generated. We need to realize the existence of multiple schemes of encoding raw data items in applying the machine learning algorithm to application areas.

Encoding of a raw data into multiple representations is illustrated in Fig. 4.25. It is assumed that multiple schemes of encoding raw data items into their multiple representations are available. The set of training examples which are given as raw data is prepared, and multiple different  $K$  feature sets are defined. Each raw data is encoded into  $K$  numerical vectors with different feature sets. A view is the process of encoding raw data into numerical vectors in context of machine learning.

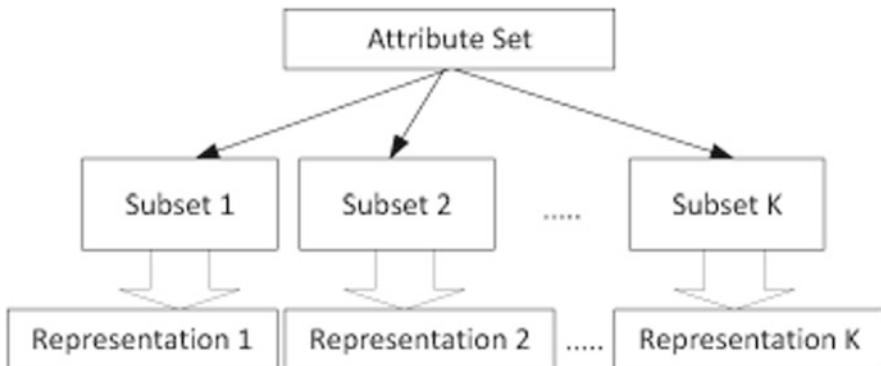
**Fig. 4.24** Single viewed learning



**Fig. 4.25** Multiple viewed learning with multiple encoding schemes

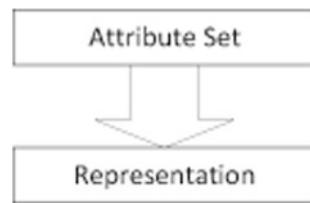
The partition of the attribute set into subsets for implementing the multiple viewed learning is illustrated in Fig. 4.26. The feature set is partitioned into subsets, if only one scheme of encoding raw data is available. In Fig. 4.26, the attribute set is partitioned into  $K$  subsets, and the raw data is encoded into  $K$  numerical vectors. The  $K$  sets of training examples are prepared for training a machine learning algorithm or machine learning algorithms. In this case, the view is a subset of attributes for encoding raw data into numerical vectors.

Let us make some remarks on the concept of view for implementing the multiple viewed learning. A single viewed learning is the traditional type of machine learning where training examples are observed with only one view. A scheme of encoding raw training examples into structured ones is assumed as a view. We define the representation of raw training examples into numerical vectors with a subset of attributes as a view. The essence of multiple viewed learning is to prepare multiple sets of represented examples to a single set of raw ones.



**Fig. 4.26** Multiple viewed learning with multiple attribute subsets

**Fig. 4.27** Single encoding

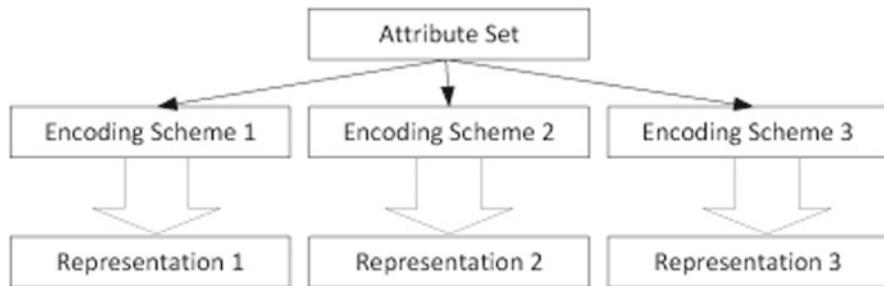


#### 4.4.2 *Multiple Encodings*

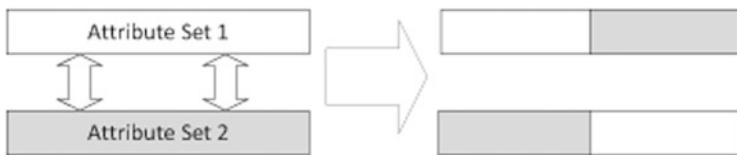
This section is concerned with encoding of a raw data item into its multiple representations. The features are selected from feature candidates in applying a machine learning algorithm to the classification. We may define multiple criteria of selecting features and prepare multiple different sets of features. A single raw data item is encoded into multiple numerical vectors with their different features. This section is intended to describe encoding of a raw data item into multiple numerical vectors.

A single encoding which is the process of mapping raw data into one kind of structured form is illustrated in Fig. 4.27. It is assumed that only a single set of attributes exists in the single encoding. A raw data item is encoded into only a representation as shown in Fig. 4.27; each data item is encoded into a numerical vector. Only a single set of training examples is prepared for training the machine learning algorithm. The single encoding is assumed in the traditional machine learning.

The multiple encoding system which is expanded from the single encoding system is illustrated in Fig. 4.28. Multiple feature sets are defined by various schemes of selecting features from the feature candidates. The feature set which consists of the selected features is partitioned to subsets. The three sets of represented training examples are prepared from a single set of raw training examples, in Fig. 4.28. In this case, a machine learning algorithm is trained with the three views.



**Fig. 4.28** Multiple encoding from single raw data



**Fig. 4.29** Attribute exchange for multiple encoding

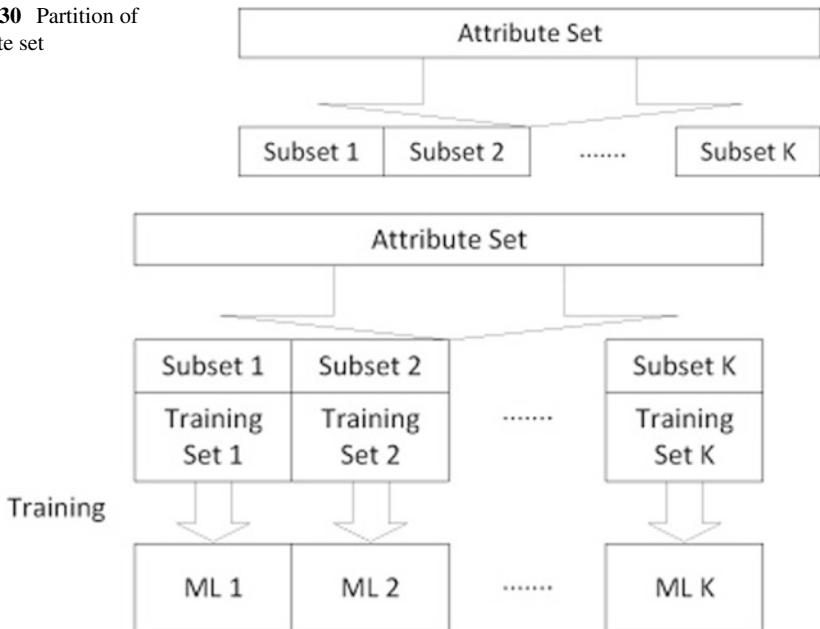
The exchange of attributes between two sets is illustrated in Fig. 4.29. The two sets of attributes are obtained by different schemes of selecting features from candidates. Two more sets of attributes are derived by exchanging some attributes between two sets. More different views are generated by this exchange in addition. The idea is from the crossover of two chromosomes in the genetic algorithm.

Let us make some remarks on the multiple encodings for implementing the multiple viewed learning. In the single viewed learning, which is the traditional learning type, it is assumed that a raw data item is encoded into a single representation. In the multiple viewed learning, a single raw data item is transformed into multiple representations, so multiple sets of representations are prepared. More views are derived by exchanging some attributes between two sets. If a single machine learning algorithm is used, we consider the integration of multiple representation into a single representation.

#### 4.4.3 *Multiple Viewed Supervised Learning*

This section is concerned with the implementation of multiple viewed learning to the supervised learning algorithms. Multiple sets of training examples as representations of raw data items are prepared by using multiple encoding schemes or partitioning the attribute set into multiple subsets. The multiple viewed learning is implemented as independent models where each machine learning algorithm is trained with its own set of training examples; each view corresponds to a machine learning algorithm. One among the ensemble learning schemes is applied to the

**Fig. 4.30** Partition of attribute set



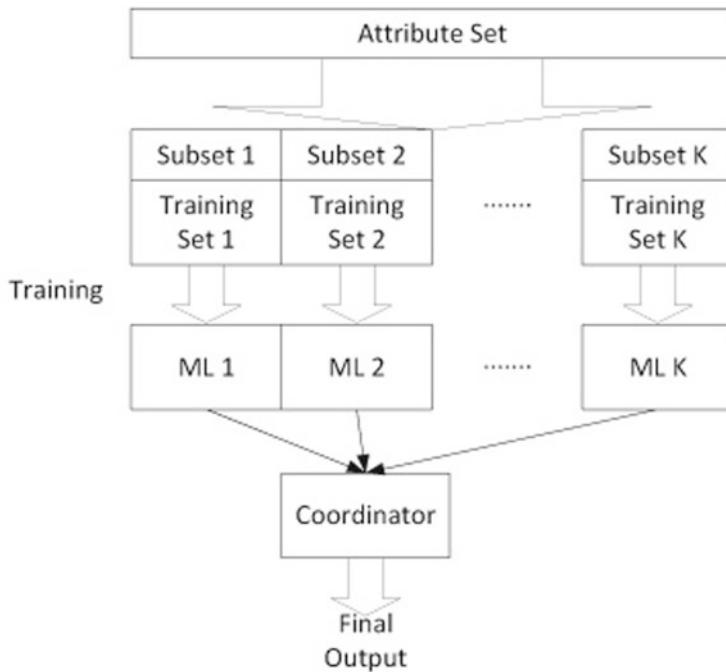
**Fig. 4.31** Training each machine learning algorithm with training examples with its own attribute subset

machine learning algorithms which are involved in the multiple viewed learning. This section is intended to describe the implementation of multiple viewed learning to the supervised learning.

The partition of the attribute set into subsets is illustrated in Fig. 4.30. The attribute set is the set of features which are selected from feature candidates by a particular scheme. The subset of features is regarded as a view, and multiple subsets of attributes are prepared for implementing the multiple viewed learning. The multiple schemes of selecting features from candidates and multiple sets of attributes are prepared, as the alternative way. The multiple sets of training examples are prepared based on the multiple subsets of attributes.

The implementation of the multiple viewed learning as independent models is illustrated in Fig. 4.31. The attribute set is partitioned into subsets, as mentioned above. We prepare multiple sets of training examples; each set of training examples corresponds to its own attribute subset. A machine learning algorithm is allocated independently to each set of training examples. The machine learning algorithms which are involved in implementing the multiple viewed learning are independent of each other.

The application of the ensemble learning to the implementation of multiple viewed learning is illustrated in Fig. 4.32. The attribute set is partitioned into subsets, and each machine learning algorithm is trained with numerical vectors which represent the training examples with its own attribute subset. A novice item



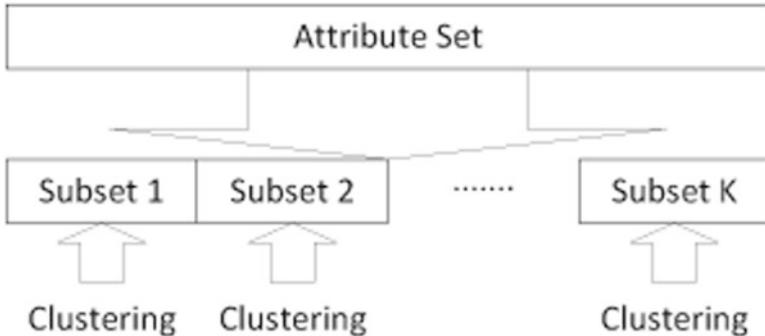
**Fig. 4.32** Final output from multiple machine learning algorithms in multiple viewed learning

which is given as a numerical vector with the full set of attributes is prepared and is transformed into multiple numerical vectors with their own attribute subsets. Each transformed one is classified by its own machine learning algorithm, and its label is finally decided by applying the ensemble learning scheme. The voting is recommended for implementing the multiple viewed learning.

Let us make some remarks on the implementation of multiple viewed learning on the supervised learning. The attribute set is partitioned into subsets exclusively; the subset of attributes is defined as a view. The multiple sets of training examples, each of which corresponds to an attribute subset, are prepared, and multiple machine learning algorithms are involved. We may consider the cross of an attribute subset and a training example subset as two-dimensional array partition.

#### 4.4.4 Multiple Viewed Unsupervised Learning

This section is concerned with the implementation of multiple viewed learning on the unsupervised learning. The multiple viewed learning was implemented on the supervised learning which is applied to the classification and the regression in the previous section. In this section, it is implemented on the unsupervised



**Fig. 4.33** Independent clustering of each attribute set

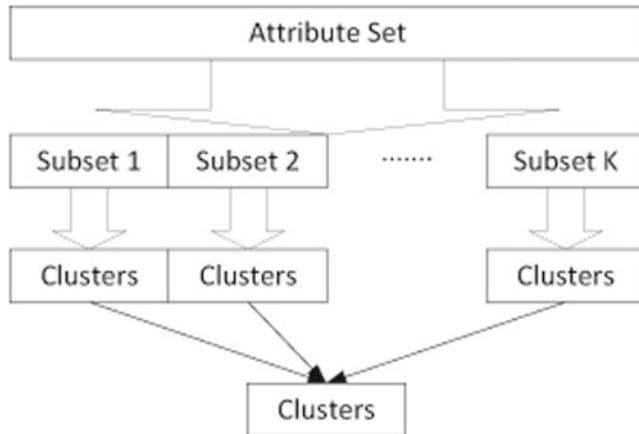
learning which is applied to the data clustering. It is assumed that each data item is represented into its multiple structured forms, and multiple groups of data items are built. This section is intended to describe the multiple viewed learning which is implemented on the unsupervised learning.

Let us consider the style of multiple views for implementing the multiple viewed learning. In the first style, the attribute set is partitioned into subsets, and each subset corresponds to a set of training examples. In the second style, various schemes of selecting features are defined, and a set of training examples corresponds to each feature scheme. In the third style, a set of training examples is defined with a single representation, and the set is partitioned into subsets. A single set of training examples is prepared, and different machine learning algorithms are trained with the same set.

The process of clustering data items independently in each subset is illustrated in Fig. 4.33. The attribute set is partitioned into subsets, and the multiple sets of training examples, each of which is a set of numerical vectors with its own attribute subset, are prepared. The numerical vectors with the attributes in the subset are clustered independently by a clustering algorithm. The results from clustering the data items are called single cluster group; the  $K$  clusters groups are results from clustering data items in multiple sets. We need to integrate the  $K$  cluster groups into a single cluster group.

The process of integrating the  $K$  cluster groups into a cluster group is illustrated in Fig. 4.34. The  $K$  cluster groups are constructed by clustering data items in each subset, independently. The voting is applied to the integration of multiple cluster groups into a cluster group. The opinion from each cluster algorithm is whether for each pair of data items, they belong to a same cluster or different clusters. Multiple cluster groups are integrated by voting the opinions of clustering algorithms.

Let us make some remarks on the implementation of multiple viewed learning on the unsupervised learning. The view is defined as a feature set of encoding a raw data item into a numerical vector, an attribute subset, or a training subset. When the independent model is adopted for implementing the multiple viewed learning, the



**Fig. 4.34** Integration of multiple clustering results

data items in each set which corresponds to its own view is clustered independently. Multiple cluster groups are integrated into a single cluster group by applying the ensemble learning scheme such as voting. We need to study schemes of integrating multiple cluster groups in addition to the data clustering.

## 4.5 Summary and Further Discussions

Let us summarize what is studied in this chapter. The training set is partitioned into subsets with one among various schemes. There are the three schemes of organizing multiple machine learning algorithms for implementing the ensemble learning, such as voting, expert gate, and cascading. The multiple viewed learning is the learning type where a machine learning algorithm or machine learning algorithms are trained with multiple schemes of observing the training examples. This section is intended to discuss further what is studied in this chapter.

Let us consider partitioning the training set by clustering training examples. They are gathered as an entire set, and the similarity metric between training examples should be defined. The training examples are clustered by a clustering algorithm regardless of their labels, depending on the similarities among them. Each cluster in the results from clustering the training example is a subset of the training examples. In adopting this style of partitioning the training set, the expert gate is used as the scheme of organizing machine learning algorithms.

Let us consider the cooperation of multiple machine learning algorithms for solving the given problem. The task is assumed to be a data classification, and the set of training examples is prepared. All of machine learning algorithms are trained with the entire set of training example, and their answers are made. The

coordinator makes the final answer by collecting their answers. The cooperation of machine learning algorithms is implemented differently, depending on whether train multiple machine learning algorithms with the entire training set or their own subsets and how to decide the final answer to the novice input.

Let us consider the division of multiple machine learning algorithms for solving the given problems. We prepare the set of training examples each of which is labeled with its own category. The set is partitioned as many as machine learning algorithms, and each of them is trained with its own subset. As an advanced way, the training examples are clustered regardless of their labels, depending on their similarities, and each machine learning algorithm is trained with its own cluster. The difference between the cooperation and the division is that in the cooperation, all of machine learning algorithms are trained with the entire set of training examples, whereas in the division, each machine learning algorithm is trained with its own subset.

Let us consider relation types of machine learning algorithms for implementing the ensemble learning. In the master-slave relation, one in the master position does its real task, and one in the slave position supports the task of the former. In the cooperation, machine learning algorithms are trained with the identical set of training examples, and all of them participate in making the output. In the division, machine learning algorithms are trained with their different sets of training examples, and one or some among them participate in making the output. In designing the ensemble learning, the relation should be separated from the combination schemes which are covered in Sect. 4.3.

## **Part II**

# **Deep Machine Learning**

This part is concerned with the modification of existing machine learning algorithms into the deep versions. We will mention the typical supervised machine learning algorithms such as the KNN algorithm, the probabilistic learning, the decision tree, and the SVM (support vector machine), as the targets which we modify into their deep versions. The machine learning algorithms are modified into the deep versions by attaching the input encoding and/or the output decoding as the basic schemes. As the advanced scheme, the supervised learning algorithms are modified into their unsupervised versions, and the original version and the unsupervised version are combined into the deep version, serially. This part is intended to study the modification of the existing machine learning algorithms into the deep versions.

This part consists of the four chapters. In Chap. 5, we will present both swallow and deep version of the KNN (K nearest neighbor) algorithm. In Chap. 6, we will modify the probabilistic learning algorithms such as the Bayes classifier and the Naive Bayes into the deep versions. In Chap. 7, we will consider the decision tree and the random forest as the targets which are modified into deep versions. In Chap. 8, we assert that the kernel-based learning is a kind of deep learning and the SVM will be modified additionally into its deep version.

# Chapter 5

## Deep KNN Algorithm



This chapter is concerned with the modification of the KNN algorithm into its deep versions. The KNN algorithm where a novice example is classified depending on labels of its nearest neighbors is the simplest supervised learning algorithm. As the basis schemes, the KNN algorithm is modified into their deep versions by attaching the input encoding and/or the output decoding. As the advanced schemes, the KNN algorithm is modified into an unsupervised version, and its unsupervised version and its original version are combined with each other into a deep version. This chapter is intended to describe the schemes of modifying the KNN algorithm into its deep versions.

This chapter is organized into the five sections, and in Sect. 5.1, we describe the deep KNN algorithm conceptually. In Sect. 5.2, we review the swallow version of KNN algorithm. In Sect. 5.3, we describe the deep version to which the input encoding and the output encoding are added. In Sect. 5.4, we describe the advanced deep version of KNN algorithm which is stacked by its unsupervised version. In Sect. 5.5, we summarize the entire contents of this chapter and discuss further on what is studied in this chapter.

### 5.1 Introduction

This section is concerned with the overview of the deep KNN algorithm. The KNN algorithm classifies a data item or estimates the output value by voting labels of selected nearest neighbors. The KNN algorithm is modified into its deep versions with the two ways: adding the input encoding or the output decoding and adding its unsupervised version as the previous layer. We consider modifying the KNN variants which discriminate nearest neighbors, attributes, or training examples and the radius nearest neighbor which selects training examples within the radius from a novice example as its nearest neighbors, as well as the KNN algorithm. This section

is intended to describe the deep version of KNN algorithm briefly for presenting the overview.

Let us mention the instance-based learning to which the KNN algorithm belongs. The instance-based learning refers to the machine learning paradigm which learns training examples depending on their distances from or their similarities as a novice example. The process of the instance-based learning is to prepare a novice example and involve some training examples which are most similar or least distant for deciding its output. The KNN algorithm, the radius nearest neighbor, and the local learning belong to the instance-based learning. The lazy learning which does not learn training examples until a novice example is given is usual in the instance-based learning.

Let us describe the deep version into which we modify the KNN algorithm, conceptually. The swallow version of KNN algorithm computes the similarities of a novice input vector with the training examples, directly. The KNN algorithm is modified into its deep version by adding the input encoding and the output encoding. The input encoding is the module for computing hidden values from the input values, and the output decoding is the module for computing output values from hidden values. The case where the input vector is mapped into one in another space is typical instances of input encoding.

We need to modify the KNN algorithm into its unsupervised version for implementing the deep KNN algorithm. For each cluster, its prototype is initialized at random, and its similar items are taken as its nearest neighbors. Each cluster has a fixed number of nearest neighbors to its prototype, they are utilized as the training examples, and data items are arranged by applying the KNN algorithm to clusters. For each cluster, by averaging members, its prototype is updated, and its nearest neighbors are taken. The learning process of the unsupervised KNN algorithm is to iterate updating the nearest neighbors and applying the KNN algorithm to the others.

Let us mention what is intended in this chapter. We review the initial version of the KNN algorithm and its variants as the swallow learning algorithms. We modify the KNN algorithm into its deep versions by attaching the input encoding or the output decoding. We modify the KNN algorithm into its unsupervised version and construct its stacked version by combining the supervised version and the unsupervised version as a more advanced deep learning algorithm. This section is intended to understand the modification of KNN algorithm into its deep version.

## 5.2 Swallow Version

This section is concerned with swallow version of the KNN algorithm before modifying it into its deep version. In Sect. 5.2.1, we mention the initial version of the KNN algorithm. In Sect. 5.2.2, we mention some variants which is derived from the initial version. In Sect. 5.2.3, we mention the version which learns the training

examples, in advance. In Sect. 5.2.4, we mention the radius nearest neighbor where a variable number of training examples may be selected.

### 5.2.1 KNN Algorithm

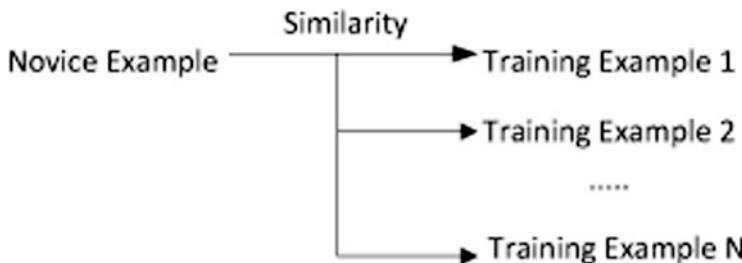
This section is concerned with the swallow version of KNN algorithm. We studied the 1 Nearest Neighbor where a novice example is classified into the label of its nearest neighbor, and it is expanded into the KNN algorithm in this section. The classification process of the KNN algorithm is to retrieve some training examples which are most similar as a novice item as the nearest neighbors and to decide the label of a novice item by voting ones of its nearest neighbors. The KNN algorithm is characterized as the instance-based learning which depends on individual training examples and the lazy learning which does not learn training examples in advance. This section is intended to briefly describe the swallow version of KNN algorithm as the basis for studying its deep version.

The similarity between a novice vector and each of the training examples is computed as shown in Fig. 5.1. A novice vector and a training example are notated by  $d$  dimensional vectors,  $\mathbf{x} = [x_1 \ x_2 \ \dots \ x_d]$  and  $\mathbf{x}_i = [x_{i1} \ x_{i2} \ \dots \ x_{id}]$ . The similarity between them is computed by the cosine similarity which is expressed as Eq. (5.1):

$$\text{sim}(\mathbf{x}, \mathbf{x}_i) = \frac{\|\mathbf{x} \cdot \mathbf{x}_i\|}{\|\mathbf{x}\| \cdot \|\mathbf{x}_i\|} \quad (5.1)$$

The inverse Euclidean distance which is expressed as Eq.(5.2) is used as an alternative one to the cosine similarity:

$$\text{sim}(\mathbf{x}, \mathbf{x}_i) = \frac{1}{\sqrt{\sum_{k=1}^d (x_k - x_{ik})^2}} \quad (5.2)$$



**Fig. 5.1** Computation of similarities with training examples

The training examples are ranked by their similarities as the novice vector, after computing the similarities.

The nearest neighbors are selected among training examples as the second step. The similarities of the novice item,  $\mathbf{x}$ , with the training examples,  $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N$ , are computed, and they are ranked by the descending order as  $sim(\mathbf{x}, \mathbf{x}_1) \geq sim(\mathbf{x}, \mathbf{x}_2) \geq \dots \geq sim(\mathbf{x}, \mathbf{x}_N)$ . The  $k$  training examples with the highest similarities are selected as  $sim(\mathbf{x}, \mathbf{x}_1) \geq sim(\mathbf{x}, \mathbf{x}_2) \geq \dots \geq sim(\mathbf{x}, \mathbf{x}_K)$ , where  $K \ll N$ . The training set is notated by a set,  $Tr = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$ , and the nearest neighbors of the novice item are expressed by function as shown in Eq.(5.3):

$$Near(\mathbf{x}, Tr) = \{\mathbf{x}_1^{near}, \mathbf{x}_2^{near}, \dots, \mathbf{x}_K^{near}\} \quad (5.3)$$

The nearest neighbors are involved in voting their labels.

The label of a novice item is decided by voting the labels of its nearest neighbors. The  $M$  categories,  $c_1, c_2, \dots, c_M$ , are predefined, and the number of nearest neighbors which are labeled with the category,  $c_i$ , is counted by Eq.(5.4):

$$Count(Near(\mathbf{x}, Tr), c_i). \quad (5.4)$$

The label is decided by the majority of nearest neighbor which is expressed by Eq.(5.5):

$$c_{max} = \operatorname{argmax}_{i=1}^M Count(Near(\mathbf{x}, Tr), c_i). \quad (5.5)$$

The label,  $c_{max}$ , is assigned to the novice item,  $\mathbf{x}$ . If there are more than one,  $c_{max}$ , one which is selected at random is assigned in the hard classification, or all of them are assigned in the soft classification.

Let us make some remarks on the swallow version of KNN algorithm which is described in this section. The similarity metric between two numerical vectors is defined for computing the similarities of a novice item with the training examples. They are ranked by their similarities with the novice input, and a fixed number of training examples is selected as nearest neighbors. The label of a novice input is decided by voting ones of the nearest neighbors. The KNN algorithm which is described in this section is the initial version, and some variants are derived from it.

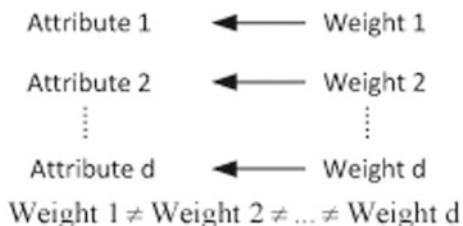
### 5.2.2 KNN Variants

This section is concerned with the KNN variants which are derived from the KNN algorithm. The KNN algorithm was studied as its swallow version in the previous section, and its variants are derived from it. In this section, we mention the three variants: discrimination among nearest neighbors, attributes, and training examples. The different weights are assigned to the nearest neighbors depending on their

**Fig. 5.2** Discrimination among nearest neighbors



**Fig. 5.3** Discrimination among attributes



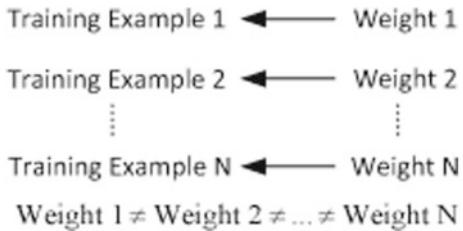
distances, the attributes depending on their correlations with the output values, and the training examples depending on their qualities. This section is intended to describe the three variants of KNN algorithm.

The assignment of weights to the nearest neighbors in the first variant is illustrated in Fig. 5.2. In the initial version of KNN algorithm, the labels of nearest neighbors are voted with their equal weights for deciding the label of the novice input. In this variant, the selected nearest neighbors are discriminated by assigning different weights to them for voting their labels. There are two ways of assigning weights to the nearest neighbors: the linear weights as the linear decrement and the exponential weights as the exponential decrement from the first ranked nearest neighbor to the last one. The labels of all training examples may be voted by assigning weights to them, depending on their distances.

The assignment of weights to the attributes in the second variant of KNN algorithm is illustrated in Fig. 5.3. The different weights are assigned to them for computing the similarity between a novice item and a training example. If the given problem is the classification, the correlation coefficients between the attribute values and the output values are used as the criteria for discriminating the attributes. The weights which are assigned to the attributes are proportional to the correlation coefficients. We will consider optimizing weights for minimizing the training error, by applying the KNN algorithm to the training examples.

The assignment of weights to the training examples in the third variant is illustrated in Fig. 5.4. The quality of individual training examples is variable in our reality. Lower weights should be assigned to training examples near the classification boundary, and higher weights are assigned to ones inside the classification boundary. The number of nearest neighbors which are labeled identically for each training example is set as the measure of its quality. This discrimination is used

**Fig. 5.4** Discrimination among training examples



for computing the similarities of training examples and voting the labels of nearest neighbors.

Let us make some remarks on the variants which are derived from the KNN algorithm. The first variant is derived by discriminating the nearest neighbors for voting their labels. The second variant is derived by discriminating attributes by their correlations with output for computing the similarities of a novice input with the training examples. The last variant is derived by discriminating the training examples by their qualities for voting the labels of the nearest neighbors and computing the similarities. The variants are expanded into the trainable KNN algorithm where optimize the weights based on the error on the training examples.

### 5.2.3 Trainable KNN Algorithm

This section is concerned with the trainable version of the KNN algorithm which can learn training examples in advance. The initial version of the KNN algorithm which is covered in Sect. 5.2.1, is known as a lazy learning algorithm which does not learn the training examples, in advance. In the version of the KNN algorithm, which is covered in this section, the weights which discriminate the nearest neighbors and the attributes are optimized before its generalization. The initial of the KNN algorithm belongs to the lazy learning, whereas the trainable version belongs to the eager learning. This section is intended to describe the trainable version of the KNN algorithm.

The process of updating the weights of nearest neighbors is illustrated in Fig. 5.5. The set of training examples, the number of nearest neighbors, and the weight list are given as the arguments. Each training example is classified, and if it is classified correctly, the weights are not updated. If it is classified incorrectly, the weights are updated for its correct classification; the weights of nearest neighbors which are labeled with the target category are increased, and the weights of the others are decreased. Updating the weights is iterated until there is no error in classifying the training examples.

The process of updating the attribute weights is illustrated in Fig. 5.6. The number of nearest neighbors is replaced by the dimension which is the number of attributes in the arguments. The weight is increased to the attribute whose value is high in the nearest neighbors with the target label and low to ones with one of the

```

List optimizeNearestNeighborWeightList(List trainingExampleList, int K, List weightList){
    int trainingExampleSize = trainingExampleList.size();
    for(int i = 0; i < trainingExampleSize; i++){
        Item trainingExample = trainingExampleList.getElement(i);
        List nearestNeighborList = trainingExample.getNearestNeighborList(trainingExampleList, K);
        Category targetCategory = trainingExample.getCategory();
        Category computedCategory = nearestNeighborList.vote();
        if(targetCategory != computedCategory)
            weightList.update();
    }
    return weightList;
}

```

**Fig. 5.5** Optimization of nearest neighbor weights

```

List optimizeNearestNeighborWeightList(List trainingExampleList, int dimension, List weightList){
    int trainingExampleSize = trainingExampleList.size();
    for(int i = 0; i < trainingExampleSize; i++){
        Item trainingExample = trainingExampleList.getElement(i);
        List nearestNeighborList = trainingExample.getNearestNeighborList(trainingExampleList, K);
        Category targetCategory = trainingExample.getCategory();
        Category computedCategory = nearestNeighborList.vote();
        if(targetCategory != computedCategory)
            weightList.update();
    }
    return weightList;
}

```

**Fig. 5.6** Optimization of attribute weights

remaining. The weight is decreased to the attribute whose value is opposite to the above case. The direction of updating weights is to retrieve more nearest neighbors with the target label and less nearest neighbors with one among the others.

The process of training the KNN algorithm is illustrated in Fig. 5.7. The process of updating weights of the nearest neighbors and the attributes is implemented above. The weights are optimized by iterating calling `optimizeNearstNeighborList` and `optimizeAttributeList`. The optimize weights of the nearest neighbors are used for voting their labels, and the optimized attribute weights are used for computing the similarities of a novice item with the training examples. The KNN algorithm which optimizes the weights belongs to the eager learning.

Let us make some remarks on the version of KNN algorithm which learns the training examples in advance. We mentioned the KNN variants which discriminate the nearest neighbors by their similarities and consider updating their weights. We consider updating the attribute weights in computing the similarities of novice item with the training examples. The weights of the nearest neighbors and the training examples are optimized for minimizing the training error. We consider optimizing the weights for minimizing error on the validation set, rather than the training set.

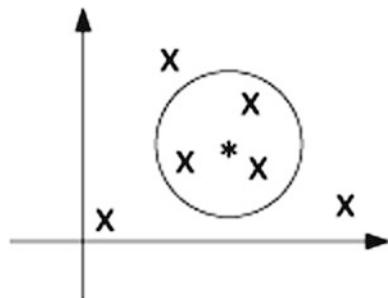
```

void trainKNN(List<List<Double>> trainingExampleList, int K, int dimension, int trainIteration){
    List<List<Double>> nearestNeighborWeightList = new List<List<Double>>(K);
    List<List<Double>> attributeWeightList = new List<List<Double>>(dimension);
    nearestNeighborWeightList.initialize();
    attributeWeightList.initialize();
    for(int i = 0;i < trainIteration;i++){
        nearestNeighborWeightList = optimizeNearestNeighborWeightList(trainingExampleList,K, nearestNeighborWeightList);
        attributeWeightList = optimizeAttributeWeightList(trainingExampleList,dimension, nearestNeighborWeightList);
    }
}

```

**Fig. 5.7** Learning process of trainable KNN algorithm

**Fig. 5.8** Threshold-based selection of nearest neighbors



### 5.2.4 Radius Nearest Neighbor

This section is concerned with the radius nearest neighbor as another instance-based learning. In the previous sections, we studied the KNN algorithm and its variants where the nearest neighbors are selected by ranking the training examples by their similarities with a novice input. In this instance-based algorithm, the nearest neighbors are selected based on the threshold which is given as an external parameter; any number of training examples with their higher similarities than the threshold is selected as the nearest neighbors. Because it is possible to select no nearest neighbor, various solutions to it should be considered. This section is intended to describe the radius nearest neighbor as the alternative instance-based learning to the KNN algorithm.

The scheme of selecting the nearest neighbors in the radius nearest neighbors is illustrated in Fig. 5.8. The point which is marked with '\*' is a novice example, and a point which is marked with "x" is a training example in Fig. 5.8. The hypersphere is defined with the novice example as its center, and its radius is given as the reverse of the similarity threshold. The training examples within the radius are selected as the nearest neighbors. The number of nearest neighbors which are selected in the radius nearest neighbor is variable, depending on the location of the novice example.

The process of classifying a novice item by the radius nearest neighbor is illustrated as a pseudo code in Fig. 5.9. A novice data item, the list of training examples, and the similarity threshold are given as the arguments. The training examples whose similarities are higher than the similarity threshold are included in the list of nearest neighbors. The category of a novice item is decided by voting

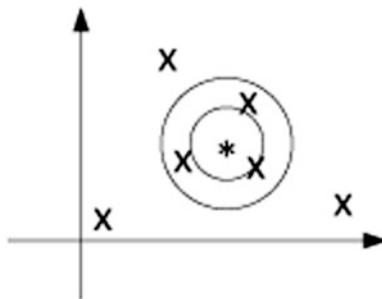
```

Category classifyByRNN(item dataItem, List trainingExampleList, double similarityThreshold){
    int trainSize = trainingExampleList.size();
    List nearestNeighborList = new List();
    for(int i = 0; i < trainSize; i++) {
        Item trainingExample = trainingExampleList.getElement(i);
        double similarity = dataItem.getSimilarity(trainingExample);
        if(similarity >= similarityThreshold)
            nearestNeighborList.addElement(trainingExample);
    }
    return nearestNeighborList.voteLabels();
}

```

**Fig. 5.9** Classification process by RNN

**Fig. 5.10** Concentrated RNN



the categories of the selected ones. The advantage of the radius nearest neighbor over the KNN algorithm is that it is not required to sort training examples by their similarities.

The variant of radius nearest neighbor, which is called concentrated RNN, is illustrated in Fig. 5.10. The variant is motivated by the possibility of a sparse number of nearest neighbors, depending on the input. Multiple similarity thresholds are used for obtaining enough nearest neighbors, instead of a single similarity threshold. Different weights are assigned to nearest neighbors in the core part and ones in the boundary part. The variant is intended to take the advantages from both the KNN algorithm and the radius nearest neighbor.

Let us make some remarks on the radius nearest neighbor as the alternative instance-based learning to the KNN algorithm. The threshold-based selection is adopted in getting the nearest neighbors for voting their labels. The training examples with their higher similarities than the threshold are selected as the nearest neighbors, and the novice item is classified by voting their labels. The concentrated radius nearest neighbor is considered for solving the problem of a sparse number of nearest neighbors, depending on the novice item, by using multiple thresholds. The advantage of the radius nearest neighbor over the KNN algorithm is to take only linear complexity to the number of training examples for getting the nearest neighbors.

### 5.3 Basic Deep Versions

This section is concerned with the deep versions of KNN algorithm by adding the input encoding or the output decoding. In Sect. 5.3.1, we describe the process of reducing the number of features. In Sect. 5.3.2, we describe the kernel-based version of KNN algorithm, together with the kernel function. In Sect. 5.3.3, we mention the deep version of KNN algorithm by adding the output decoding to the swallow version. In Sect. 5.3.4, we mention the version of KNN algorithm which is added by the pooling layer.

#### 5.3.1 Feature Reduction

This section is concerned with the process of reducing the dimension of the input vector which is called input encoding. Depending on the application area, many features are required for encoding a raw data item into a numerical vector. The feature reduction is the process of reducing the dimensionality of the input vector, automatically. The vector with the reduced dimension mapped by the feature reduction is viewed as a hidden vector which characterizes the deep learning. This section is intended to describe the process of reducing the number of features which is necessary for implementing the deep learning.

The correlations of the attribute values with the output values are illustrated in Fig. 5.11. The standard deviation of the input variable,  $x_i$ , is computed by Eq. (5.6):

$$s_{x_i} = \sqrt{\frac{1}{N} \sum_{k=1}^N x_{ik}^2 - \left( \frac{1}{N} \sum_{k=1}^N x_{ik} \right)^2} \quad (5.6)$$

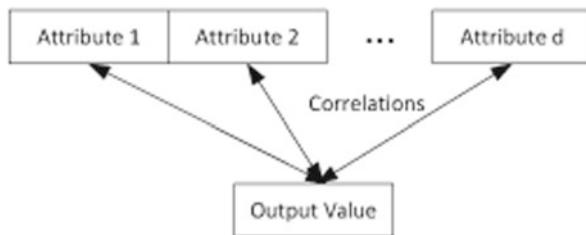
where  $N$  is the number of training examples and the standard deviation of the output variable,  $y$ , is computed by Eq. (5.7):

$$s_y = \sqrt{\frac{1}{N} \sum_{k=1}^N y_k^2 - \left( \frac{1}{N} \sum_{k=1}^N y_k \right)^2} \quad (5.7)$$

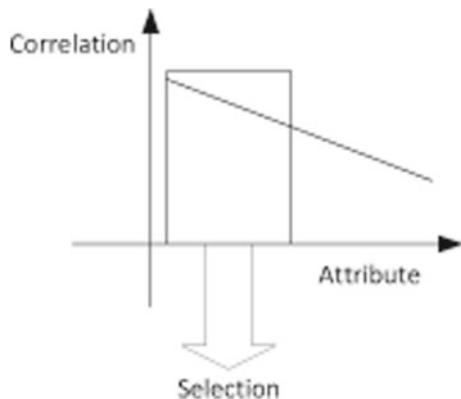
The covariance of the input variable,  $x_i$ , and the output variable,  $y$ , is computed by Eq. (5.8),

$$s_{x_i, y} = \sqrt{\frac{1}{N} \sum_{k=1}^N x_{ik} y_k - \frac{1}{N^2} \sum_{k=1}^N x_{ik} \sum_{k=1}^N y_k} \quad (5.8)$$

**Fig. 5.11** Attribute correlation with output values



**Fig. 5.12** Selection of attributes with higher correlations



The correlation coefficient between the input variable,  $x_i$ , and the output variable,  $y$ , is computed by Eq. (5.9):

$$r_{x_i, y} = \frac{s_{x_i, y}}{s_{x_i} s_y} \quad (5.9)$$

If the correlation coefficient is close to 1 or  $-1$ , the two variables are correlated strongly with each other, and if it is close to 0, the two variables are independent of each other.

The selection of some attributes by their correlations with the output values is illustrated in Fig. 5.12. The correlation coefficient with output variable is computed for each attribute by the above process. The attributes are ranked by the descending order of their correlation coefficients. The attributes with their higher correlation coefficients are selected. The input vector with the  $d$  dimensionality is encoded into one with its less dimensionality.

Let us mention some schemes of reducing the number of features for more efficient processing. The singular value decomposition is the dimension reduction scheme where the training set is viewed as a matrix, it is decomposed into the three components, and the number of features is reduced depending on the eigenvalues. In the principal component analysis, the covariance matrix is computed from the training set, its eigenvectors which correspond to the largest eigenvalues are selected, and the dimension is reduced by multiplying each training example by the

matrix which consists of the selected eigenvectors. In the independent component analysis, the dimension is reduced by multiplying each training example by the matrix which consists of cluster prototypes. Refer to [1] for studying the dimension reduction schemes in detail.

Let us make some remarks on the dimension reduction as a module for implementing the deep learning. The correlation coefficient with the output values is computed for each attribute. The attributes are ranked by their correlation coefficients, and some with highest correlation coefficients are selected among the attributes. The advanced schemes of selecting some features are the singular value decomposition, the principal component analysis, and the independent component analysis. The dimension reduction is viewed as the process of computing a hidden vector from the input vector by generating a lower dimensional vector.

### 5.3.2 *Kernel-Based KNN Algorithm*

This section is concerned with the kernel-based KNN algorithm which is a deep version of KNN algorithm. In the previous section, we studied the process of reducing the dimension with viewing it as the process of computing a hidden vector with its lower dimensionality from the input vector. In this section, we study the deep version of KNN algorithm where the kernel function is applied for computing the similarities of a novice item with the training examples. The kernel function of two input vectors indicates the inner product of their mapped vectors. This section is intended to describe the modified version of KNN algorithm, called kernel-based KNN algorithm.

Let us mention the kernel function which is applied to the KNN algorithm. The two vectors are denoted by  $\mathbf{x}_1$  and  $\mathbf{x}_2$ , and they are mapped into vector in another space,  $\Phi(\mathbf{x}_1)$  and  $\Phi(\mathbf{x}_2)$ . The kernel functions of the two vectors,  $\mathbf{x}_1$  and  $\mathbf{x}_2$ , are defined as the inner product of the vectors,  $\Phi(\mathbf{x}_1)$  and  $\Phi(\mathbf{x}_2)$ , as shown in Eq. (5.10):

$$K(\mathbf{x}_1, \mathbf{x}_2) = \Phi(\mathbf{x}_1) \cdot \Phi(\mathbf{x}_2) \quad (5.10)$$

The inner product of the two vector is absolutely given as a scalar value in any dimension. We will study in detail the kernel function in Chap. 8.

The kernel function which is mentioned above is utilized as the similarity metric. It indicates the inner product of mapped vectors as the similarity between them. The similarities of a novice input with the training examples are computed by the kernel function. They are ranked by the output values of the kernel function. In this version of the KNN algorithm, the similarity metric, such as the cosine similarity and the inverse Euclidean distance, is replaced by the kernel function.

Let us mention the process of classifying a novice input by this version of KNN algorithm. The input vector is implicitly transformed into one in another space as a hidden vector. The inner product of the hidden vector which is mapped from the input vector and one which is mapped from the training example is computed by the

kernel function. It is used as the similarity metric between a novice input vector and a training example. The process of classifying a data item in this version is the same to that in the initial version of KNN algorithm except using the kernel function as a similarity metric in computing the similarities of a novice vector with the training examples.

Let us make some remarks on the kernel-based KNN algorithm which is covered in this section. The kernel function of vectors is the inner product of their mapped vectors in another space. The kernel function is used for computing the similarities of a novice input with the training examples in this version of KNN algorithm. The nearest neighbors are selected from the training examples, depending on the kernel function in the process of classifying a novice item. The fact that the input vector is implicitly mapped into one in another space, as its hidden vector, is the reason of viewing this version of KNN algorithm as a deep learning algorithm.

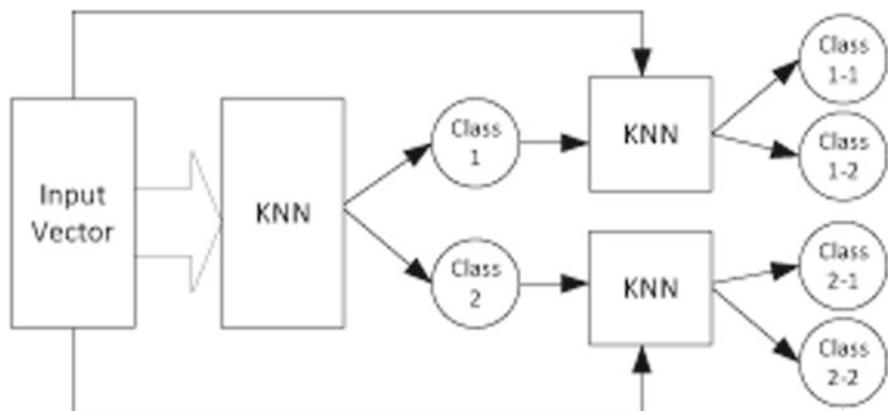
### 5.3.3 *Output Decoded KNN*

This section is concerned with another deep version of KNN called output decoded KNN. In the previous section, we studied the deep version of KNN with the attachment of the input encoding. In this section, the KNN algorithm is modified into its deep version by attaching the output decoding. It is viewed as the process of computing the output vector from a hidden vector, in context of deep learning. This section is intended to describe the deep version of KNN algorithm which is modified by attaching the output decoding.

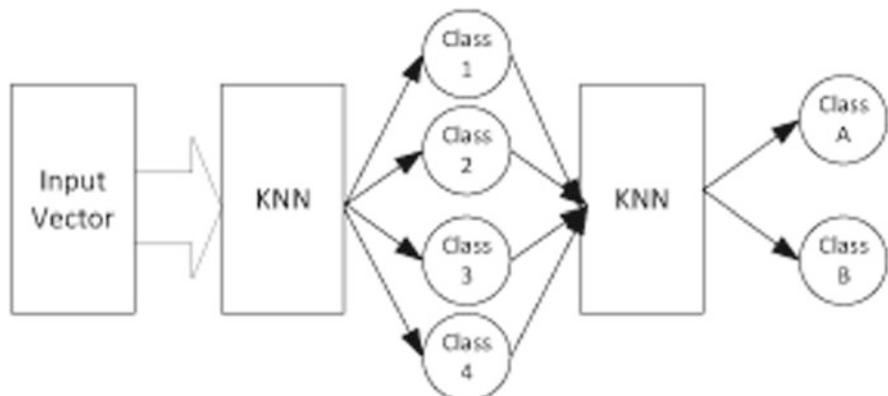
The application of the KNN algorithm to the implementation of the category specification is illustrated in Fig. 5.13. In the general level, there are two categories, class 1 and class 2, and two specific categories are allocated to each abstract category. In the left, the KNN algorithm classifies the input vector into class 1 or class 2. The top-right KNN algorithm receives the input vector which is classified into class 1 and classifies it into class 1–1 or class 1–2, and the bottom-right KNN algorithm receives the input vector which is classified into class 2 and classifies it into class 2–1 or class 2–2. This type of deep learning is viewed as implementing the hierarchical classification with the KNN algorithm.

The application of the KNN algorithms to the implementation of the category abstraction is illustrated in Fig. 5.14. It is assumed that the value of each category of the four specific categories is continuous as a categorical score. The process of estimating the categorical scores of the specific categories is viewed as the process of computing a four-dimensional hidden vector from the input vector. The four-dimensional hidden vector is classified into one of the two categories by the right KNN algorithm. The category abstraction is viewed as the process of computing the output vector with its lower dimensionality from a hidden vector with its higher dimensionality.

The addition of the outlier detection for modifying the KNN algorithm into its deep version is illustrated in Fig. 5.15. The outlier detection is the process of



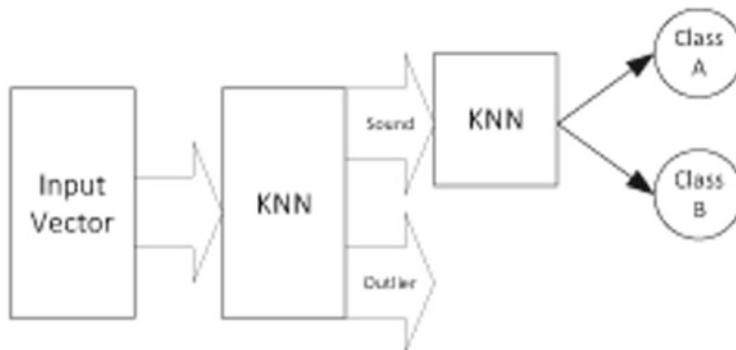
**Fig. 5.13** Category specification layer in KNN



**Fig. 5.14** Category abstraction layer in KNN

detecting a novice input as a outlier or not. A novice input is classified by the left KNN algorithm into an outlier or a sound item. The novice input which is classified into sound is classified into class A or class B by the right KNN algorithm. The novice input which is classified into outlier may be rejected in the classification task.

Let us make some remarks on the three deep versions of the KNN algorithm by attaching the output decoding. A data item is classified into a general category, augmented with the classified category, and classified into a specific category, if the category specification is attached to the KNN algorithm. In attaching the category abstraction, categorical scores are assigned to the data item as the fuzzy classification, and the hidden vector whose elements are categorical scores is classified into a general category. The outlier detection is the process of judging the current novice input as a strange vector, or not. The output decoding which is



**Fig. 5.15** Outlier detection layer in KNN

attached to the KNN algorithm for modifying it into the deep version focuses on the computation of the output vector from a hidden vector.

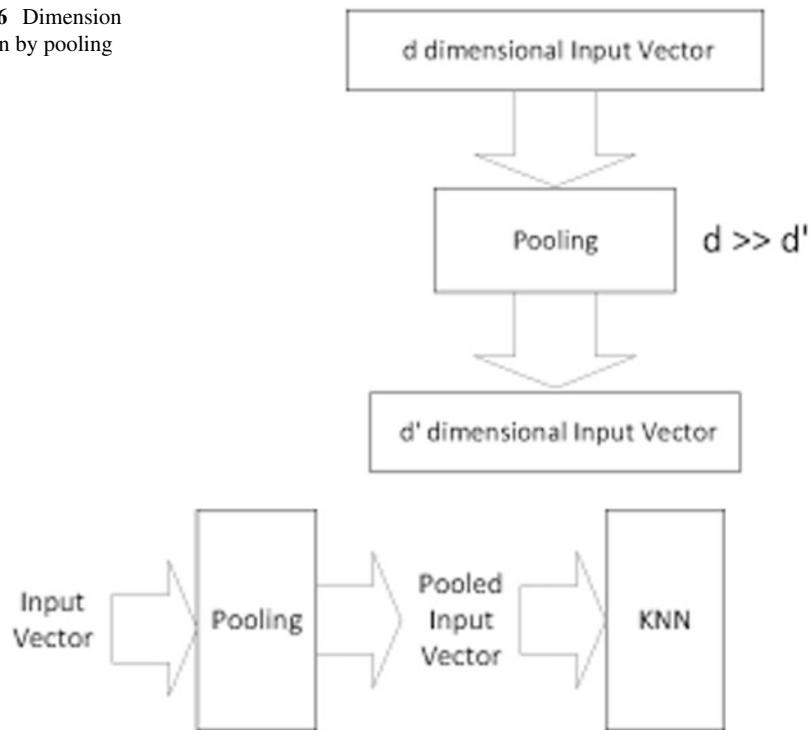
### 5.3.4 Pooled KNN

This section is concerned with the deep version of the KNN algorithm by attaching the pooling layer. In the previous section, we modified the KNN algorithm by attaching the output decoding such as the category specification and the category abstraction. In this section, we modify the KNN algorithm into the deep version by adding the pooling layer. In this version of the KNN algorithm, the similarities of one which is mapped from the input vector with ones which are mapped from the training examples are computed. This section is intended to describe the deep version of the KNN algorithm which is modified by attaching the pooling layer.

The role of pooling operation which is the dimension reduction is illustrated in Fig. 5.16. The pooling is the operation on a vector for collecting representative values from the window which slides on it. If the dimension of the input vector is  $d$ , and the size of the window is  $s$ , the dimension of the mapped vector is  $d - s + 1$  less than  $d$ . If the weighted average is adopted for implementing the pooling, the vector which consists of weights is involved as a filter vector, and the pooling is viewed as the convolution. A single input vector is mapped into multiple vectors by adopting multiple types of pooling.

The pooling layer which is arranged in front of the KNN algorithm is presented in Fig. 5.17. In this layer, it is assumed that the softmax pooling is adopted. The input vector is notated by  $\mathbf{x} = [x_1 \ x_2 \ \dots \ x_d]$ , and the size of sliding window is  $s$ . Each element of the hidden vector which is mapped from the input vector is computed by Eq. (5.11):

**Fig. 5.16** Dimension reduction by pooling



**Fig. 5.17** Pooling layer in KNN

$$h_i = \max_{j=1}^s x_{i+j-1}. \quad (5.11)$$

The hidden vector with its  $d - s + 1$  dimensions is denoted by  $\mathbf{h} = [h_1 \ h_2 \ \dots \ h_{d-s+1}]$ .

Let us mention the process of classifying a data item by this version of KNN algorithm. The training examples are prepared as a set,  $Tr = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$ , and a novice input vector is denoted by  $\mathbf{x}$ . The vectors,  $\mathbf{x}$  and  $\mathbf{x}_i$ , are mapped into  $\mathbf{h}$  and  $\mathbf{h}_i$  by the pooling for each training example. The similarity between the two mapped vectors is computed by Eq. (5.12):

$$\text{sim}(\mathbf{h}, \mathbf{h}_i) = \frac{\|\mathbf{h} \cdot \mathbf{h}_i\|}{\|\mathbf{h}\| \|\mathbf{h}_i\|} \quad (5.12)$$

The similarity,  $\text{sim}(\mathbf{x}, \mathbf{x}_i)$ , is replaced by the similarity,  $\text{sim}(\mathbf{h}, \mathbf{h}_i)$ , between mapped vectors in this version.

Let us make some remarks on the deep version of KNN algorithm which is modified by attaching the pooling layer. The pooling operation is used as a scheme of reducing the dimensionality. The pooling layer is viewed as the hidden

layer where a hidden vector is computed from the input vector. The similarity between vectors which are mapped from the novice input vector and a training example by the pooling is computed in this version of KNN algorithm. The deep KNN algorithm which is called convolutional KNN algorithm is implemented by attaching alternatively the pooling layer and the convolution layer.

## 5.4 Advanced Deep Versions

This section is concerned with the advanced deep versions of KNN algorithm by adding its unsupervised version. In Sect. 5.4.1, we introduce the unsupervised layers as the preparation for modifying the machine learning algorithm into its deep version. In Sect. 5.4.2, we modify the KNN algorithm into its unsupervised version. In Sect. 5.4.3, we describe the stacked KNN algorithm as a deep version. In Sect. 5.4.4, we describe the convolutional KNN algorithm where the convolution layer is added to the KNN algorithm.

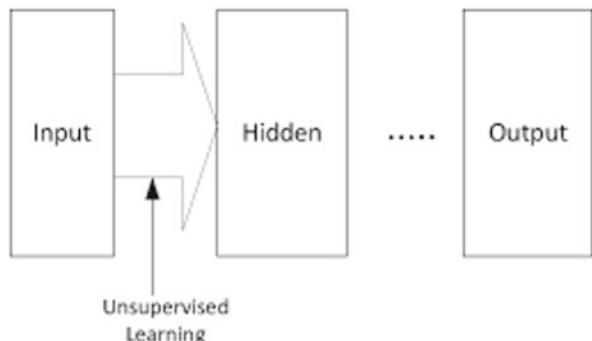
### 5.4.1 Unsupervised Layer

This section is concerned with the unsupervised layer which is needed for implementing the deep learning. The hidden values are computed from the input vector in the style of the unsupervised learning. A single unsupervised layer or multiple unsupervised layers are installed between the input layer and the output layer for implementing the deep learning. If the KNN algorithm is modified into its deep version, it should be modified into its unsupervised version. This section is intended to describe the install of the unsupervised learning as a layer into the existing supervised learning algorithm.

Let us review the unsupervised learning which was studied in Chap. 3 as the swallow learning. The unsupervised learning is the type of machine learning which searches for the optimized parameters, depending on similarities among unlabeled examples. The similarity metric between input vectors is defined, and the parameters which are usually given as cluster prototypes are optimized depending on the similarities among input vectors. In the k means algorithm, the mean vectors as parameters are optimized, and in the Kohonen Networks, the weight vectors are optimized. Even if the deep learning is aimed at the supervised learning, the unsupervised learning may be needed for implementing it.

Using the unsupervised learning for implementing the deep learning is illustrated in Fig. 5.18. Only final output values are presented in the training examples in the supervised learning; no desirable hidden value is available in implementing the deep learning. The unsupervised learning is applied for computing the hidden values from the input values. The unsupervised learning is applied between hidden layers, and the supervised learning is applied from the last hidden layer to the output layer.

**Fig. 5.18** Unsupervised learning layer



**Fig. 5.19** Deep learning structure with unsupervised learning layers

We need to modify the current supervised learning algorithm into its unsupervised version for implementing more advanced deep learning.

The structure of the deep learning algorithm is illustrated in Fig. 5.19. The deep learning has serially multiple hidden layers between the input layer and the output layer in the generic view. If multiple hidden layers are included, the first hidden values are computed from the input values, the hidden values in the medium layer are computed from ones in the previous layer, and the output values are computed from the last hidden values. The unsupervised learning is applied for computing the hidden values from the input values or the previous hidden values. The decision of the number of hidden layers and the scheme of computing hidden values in each layer is the issue in designing the deep learning.

Let us make some remarks on the application of the unsupervised learning for computing hidden values. The unsupervised learning is the learning type where parameters are optimized depending on the similarities among the input vectors. Because the target intermediate values are not available in the deep learning, the unsupervised learning is applied for computing the hidden values from the input values or one in the previous layer. It is assumed that there are multiple hidden layers between the input layer and the output layer as the deep learning structure, and the deep learning is implemented with the unsupervised learning from the input

layer to the last hidden layer. The supervised learning algorithm is modified into its unsupervised version for doing it into its deep version.

### 5.4.2 *Unsupervised KNN*

This section is concerned with the unsupervised version of KNN algorithm. In the previous section, we studied the role of the unsupervised learning for implementing the deep learning algorithm. In this section, we modify the KNN algorithm which is initially proposed as a supervised learning algorithm into its unsupervised version. The unsupervised version is used for computing hidden values in implementing its deep version. This section is needed to describe the unsupervised version of KNN algorithm as the approach to the data clustering.

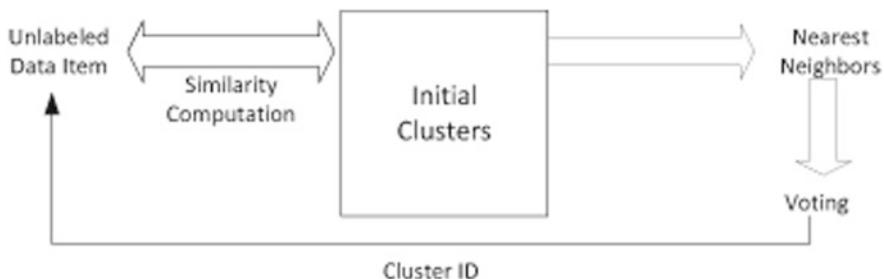
The cluster initialization for using the unsupervised KNN algorithm is illustrated in Fig. 5.20. In advance, the number of clusters is decided, and data items as many as clusters are selected at random as the initial cluster prototypes. For each cluster, the nearest neighbors of its initial cluster prototype are selected and included in the cluster as its members. Each cluster is composed with its initial cluster prototype and its nearest neighbors. The initial cluster prototypes and their nearest neighbors are used as the seed training examples.

The arrangement of a data item into a cluster by the unsupervised KNN algorithm is illustrated in Fig. 5.21. The initial clusters, each of which consists of its initial prototype and its nearest neighbors, are constructed by the above process. The data items which are members of the initial clusters are used as the seed examples; for each unlabeled data item, its similarities with the seed examples are computed. The nearest neighbors are selected from the seed examples, and the unlabeled data item is arranged by voting the clusters which the nearest neighbors belong to. The cluster is updated by rearranging data items after arranging all data items.

The process of clustering data items by the unsupervised KNN algorithm is illustrated in Fig. 5.22. The initial clusters are constructed as the seed examples, and the remaining is arranged by applying the KNN algorithm. The data items are rearranged into their own clusters by applying the KNN algorithm to the data items



**Fig. 5.20** Initialized clusters in unsupervised KNN



**Fig. 5.21** Arrangement of item in unsupervised KNN

```

void clusterDataItemList(List itemList, int clusterNumber, int K){
    List clusterList = itemList.getInitialClusterList(clusterNumber);
    for(int i = 0;i < clusterNumber;i++){
        Cluster cc = clusterList.getElement(i);
        Item clusterPrototype = cc.getPrototype();
        List nearestNeighborList = exampleList.getNearestNeighborList(clusterPrototype,K);
        cc.addItemList(nearestNeighborList);
        clusterList.update(i,cc);
    }
    int itemSize = itemList.size();
    repeat until convergence {
        for(int i = 0;i < itemSize;i++){
            Item indivItem = itemList.getElement(i);
            List nearestNeighborList = clusterList.getNearestNeighborList(indivItem,K);
            int clusterIndex = nearestNeighborList.vote();
            Cluster cc = clusterList.getElement(clusterIndex);
            cc.addItem(indivItem);
            clusterList.update(clusterIndex,cc);
        }
    }
}
  
```

**Fig. 5.22** Clustering process of unsupervised KNN

which were previously arranged. This arrangement of data items is iterated until the convergence. The results from clustering data items depend on the seed examples which are prepared in the initial stage.

Let us make some remarks on the unsupervised version of KNN algorithm for clustering data items. Clusters are initialized by deciding initial cluster prototypes at random and collecting their nearest neighbors. Data items are arranged into clusters depending on seed examples which are prepared in initializing the clusters. The clusters are continually updated by arranging data items into clusters with the KNN algorithm. It is possible to modify the radius nearest neighbor which is the alternative instance-based learning algorithm to the KNN algorithm into its unsupervised version.

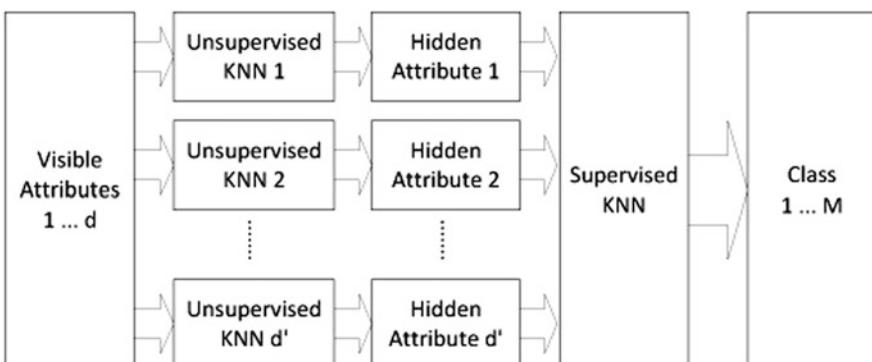
### 5.4.3 Stacked KNN

This section is concerned with the stacked KNN algorithm as a deep learning algorithm. In the previous section, we studied the unsupervised version of KNN algorithm which is applied to the data clustering. In this section, we combine the supervised version and the unsupervised version into the deep version of the KNN algorithm. The unsupervised versions are used for computing the hidden values, and the supervised version is used for computing the output values. This section is intended to describe the stacked KNN algorithm.

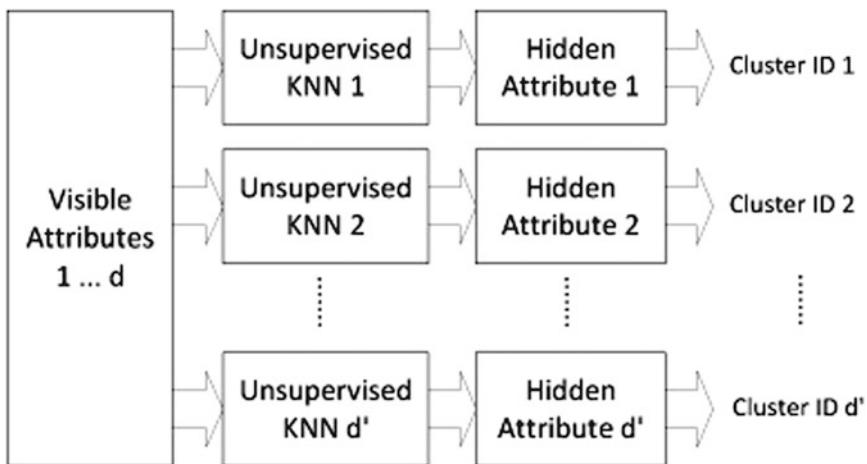
The stacked architecture of KNN algorithm for implementing the deep learning is illustrated in Fig. 5.23. The given task is assumed as the multiple classification which classifies each item into one among  $M$  classes. The  $d'$  unsupervised KNN algorithms are involved in computing the hidden vector with the  $d'$  dimensions from the input vector with the  $d$  dimensions. The role of the supervised KNN algorithm is to classify the hidden vector into one among the  $M$  classes. The hidden vector is computed by the unsupervised versions in this deep version.

The process of computing the hidden values from the input values by the unsupervised KNN is illustrated in Fig. 5.24. The clustering is assumed as the hard clustering where each data item is arranged into only one cluster. In this deep version, the  $d'$  unsupervised KNN algorithms are involved in computing the hidden values. Each element of the hidden vector is given as an integer which indicates a cluster identifier. The unsupervised KNN algorithms are discriminated by the number of clusters and the number of nearest neighbors.

The multiple classification part by the supervised KNN algorithm is illustrated in Fig. 5.25. The hidden vector is computed by the  $d'$  unsupervised KNN algorithms from the input vector. The similarities of the hidden vector which is mapped from the novice input vector with ones which are mapped from the training examples are computed. The nearest neighbors are selected, and their labels are voted for deciding

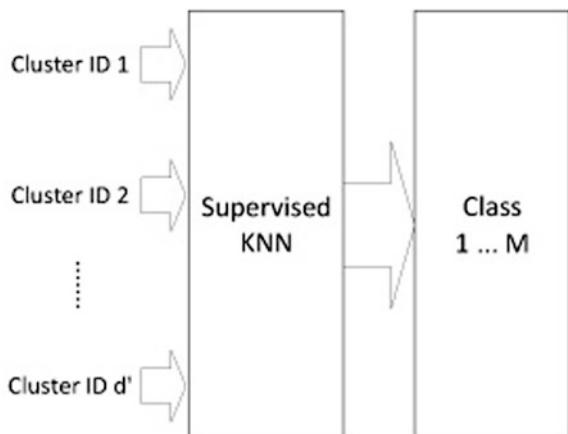


**Fig. 5.23** Architecture of deep KNN algorithm with unsupervised layer



**Fig. 5.24** Unsupervised KNN algorithms for computing hidden vector

**Fig. 5.25** Supervised KNN algorithm for classifying hidden vector



the label. In advance, the training examples are transformed into their hidden vector by the unsupervised KNN algorithms.

Let us make some remarks on the stacked version of KNN algorithm as a deep learning algorithm. In the architecture, multiple unsupervised KNN algorithms are involved in computing the hidden vector, and a single supervised KNN algorithm is involved in classifying the hidden vector. Each element of the hidden vector which is computed by the multiple unsupervised KNN algorithms is given as a cluster identifier. The hidden vector which is mapped from the novice input vector is classified by a single supervised KNN algorithm. The process of computing the hidden vector is the fuzzy clustering, and a single fuzzy unsupervised KNN algorithm may be involved.

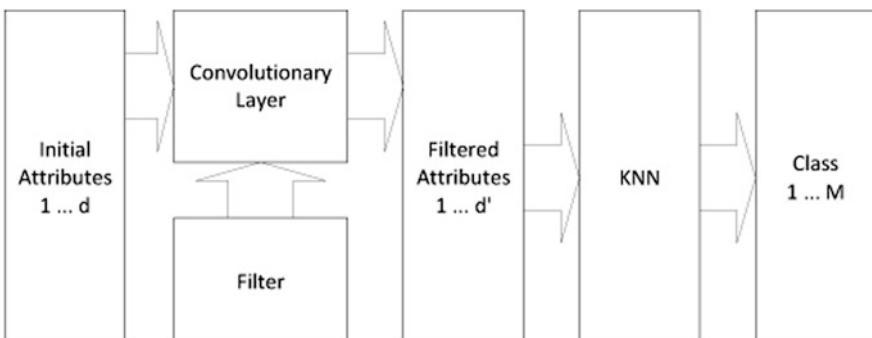
### 5.4.4 Convolutional KNN Algorithm

This section is concerned with the convolutional KNN algorithm as a deep learning algorithm. In the previous section, the deep learning was implemented on the KNN algorithm by combining the supervised version and the unsupervised versions with each other. In this section, the unsupervised KNN algorithms are replaced by the convolutional layer in the deep version of KNN algorithm. The hidden vector is computed from the input vector through the convolutional layer. This section is intended to describe the deep version of KNN algorithm which is called convolutional KNN algorithm.

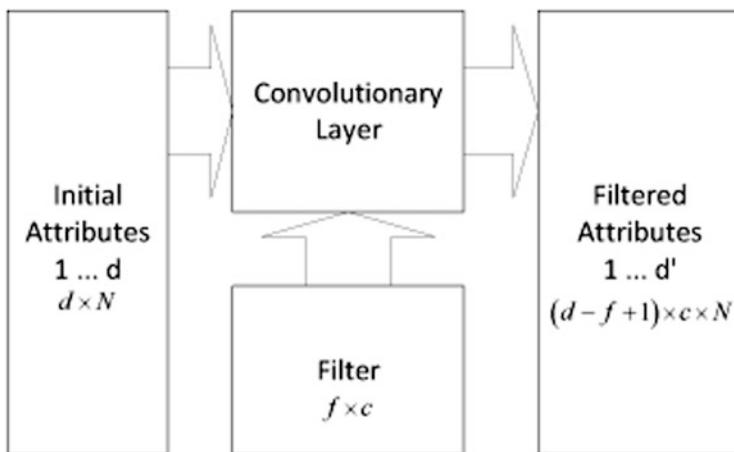
The architecture of this deep version of KNN algorithm is illustrated in Fig. 5.26. The input vector which is given as a  $d$  dimensional numerical vector is mapped into a  $d'$  dimensional hidden vector through the convolutional layer. The mapped hidden vector is classified into one among the  $M$  classes by the KNN algorithm. In the process of computing the hidden vector, multiple unsupervised KNN algorithms are replaced by the convolutional layer. The filter vector which is set arbitrary is involved in the convolutional layer.

Mapping the training examples into hidden vectors through the convolutional layer is illustrated in Fig. 5.27. The set of  $N$  training examples, each of which is given as a  $d$  dimensional vector, is initially prepared. The  $c$  filter vectors, each of which is given as a  $f$  dimensional numerical vectors are defined; the number of filter vectors,  $c$ , is the number of channels. The results from applying the convolution with the  $c$  filter vectors are  $N \times c$  numerical vectors with  $d - f + 1$  dimensions. The  $c$  numerical vectors which are mapped from each training example are viewed as a  $(d - f + 1) \times c$  matrix.

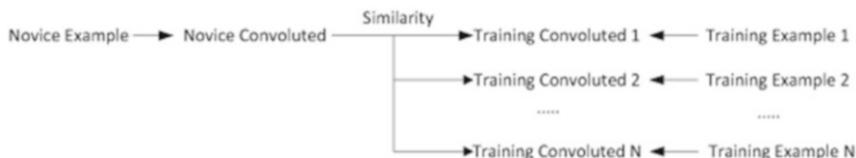
The process of classifying a novice data item by this deep version of KNN algorithm is illustrated in Fig. 5.28. A novice input vector and the training examples are mapped by the convolutional layer. The similarities of a mapped novice vector with the mapped training vectors are computed, and most similar ones are selected as nearest neighbors. The label of a novice input vector is decided voting the labels



**Fig. 5.26** Architecture of convolutionary KNN



**Fig. 5.27** Mapping of training examples by convolutionary layer



**Fig. 5.28** Similarity between novice example and training example in convolutionary KNN

of nearest neighbors. If multiple filter vectors are used in the convolutional layer, it takes much more time to compute the similarities.

Let us make some remarks on the convolutional KNN algorithm as its deep version. The hidden vector is computed from the input vector in the convolutional layer. A single filter vector or multiple filter vectors are involved in carrying out the convolution. The similarity between mapped vectors is computed rather than original vectors, in this version of KNN algorithm. Alternative multiple convolutional layers and pooling layers may be added for implementing the more advanced deep version.

## 5.5 Summary and Further Discussions

Let us summarize what is studied in this chapter. The KNN algorithm is the machine learning algorithm which classifies a data item or estimates an output value by voting labels of nearest neighbors among the training examples. The modified version of KNN algorithm by adding the input encoding or the output decoding are the kernel-based KNN algorithm, the output decoded KNN algorithm, and the

pooled KNN algorithm. The stacked KNN algorithm is the deep version of KNN algorithm by combining its supervised version and its unsupervised version with each other. This section is intended to discuss further what is studied in this chapter.

Let us consider the instance-based learning algorithm, which is alternative to the KNN algorithm, called radius nearest neighbor. In the KNN algorithm, a fixed number of nearest neighbors is selected. In this algorithm, the training examples whose similarities are more than or equal to the threshold are selected. The difference between the two algorithms is that the ranking selection is adopted in the KNN algorithm, and the threshold-based selection is adopted in the radius nearest neighbor. The limit of the radius nearest neighbor is that the possibility of getting no nearest neighbor to a novice input exist.

Let us consider the indirect neighbors in deriving variants from the KNN algorithm. The direct nearest neighbors are the training examples with higher similarities with a novice input, whereas the indirect nearest neighbors are ones from a particular nearest neighbor, instead of a novice input. The direct nearest neighbors are retrieved from the novice input, and the indirect nearest neighbors are retrieved from each direct nearest neighbor, additionally. The indirect nearest neighbors may participate in voting their labels with their relatively lower weights than ones of direct nearest neighbors. Variants may be derived by retrieving the indirect nearest neighbors from the KNN algorithm and the radius nearest neighbor.

Both the deep learning and the ensemble learning are applied to the KNN algorithm. It was modified into its deep versions in Sects. 5.3 and 5.4. We prepare the multiple deep versions of KNN algorithm and apply the ensemble learning to them. One or hybrid is selected among the three main schemes: voting, expert gate, and cascading. We choose either of the combination of identical deep versions and the combination of different deep versions.

Let us consider the KNN algorithm to which the text convolution is attached in applying it to the text classification. The convolution is the process of mapping an input vector or an input matrix into another by sliding a linear combination of elements of the filter vector or the filter matrix. A text is viewed as a list or a matrix of words, and some words are filtered in the text convolution. A word is replaced by its most similar word which is retrieved from external sources. The textual convolutional KNN is implemented by attaching the textual convolution as an additional layer.

## Reference

1. T. Jo, “Text Mining: Concepts, Implementation and Big Data Challenges”, Springer, 2019.

# Chapter 6

## Deep Probabilistic Learning



This chapter is concerned with the modification of the Bayes classifier and the Naive Bayes into their deep version. In the probabilistic learning, we mentioned the three typical machine learning algorithms, the Bayes classifier, the Naive Bayes, and the Bayesian learning in [1]. As the basic schemes, the Bayes classifier and the Naive Bayes are modified into their deep versions by attaching the input encoding and/or the output encoding. As the advanced schemes, they are modified into their unsupervised versions, and their original versions and their unsupervised versions are combined with each other into their stacked version. This chapter is intended to describe the schemes of modifying the two probabilistic learning algorithms into their deep versions.

This chapter is organized into five sections, and in Sect. 6.1, we describe the deep probabilistic learning conceptually. In Sect. 6.2, we review the swallow version of the three probabilistic learning algorithms, such as the Bayes classifier, the Naive Bayes, and the Bayesian learning. In Sect. 6.3, we describe the deep version of the Bayes classifier and the Naive Bayes to which the input encoding and the output encoding are added. In Sect. 6.4, we describe the advanced deep version of them which are stacked by their unsupervised versions. In Sect. 6.5, we summarize the entire contents of this chapter and discuss further on what is studied in this chapter.

### 6.1 Introduction

This section is concerned with the overview of the deep probabilistic learning. The probabilistic learning is a kind of machine learning algorithm, depending on category probabilities given a novice vector, defined from the training examples. The probabilistic learning algorithms such as the Bayes classifier and the Naive Bayes are modified into their deep versions with the two ways: adding the input encoding or the output encoding and adding the unsupervised version as the previous layer. The k means algorithm is the unsupervised version of the Bayes classifier; it will

be used for implementing the deep version. This section is intended to describe the deep version of the probabilistic learning for presenting the overview.

Let us mention the probability, to provide the background for studying what is covered in this chapter. The probability means the possibility degree with an event happens. The probability value is given as a continuous value between zero and one, as shown in Eq. (6.1):

$$0 \leq P(E) \leq 1 \quad (6.1)$$

where  $E$  is an event. The two events are  $A$  and  $B$ , and the conditional probability,  $P(A|B)$ , is the probability of the event,  $A$ , if the event,  $B$ , happens, and is expressed in Eq. (6.2):

$$P(A|B) = \frac{P(AB)}{P(B)} \quad (6.2)$$

Next, we will consider the relation between the two conditional probabilities,  $P(A|B)$  and  $P(B|A)$ .

Let us mention the Bayes rule which expresses the relation between the two conditional probabilities,  $P(A|B)$  and  $P(B|A)$ . They are expressed as Eq. (6.2) and (6.3):

$$P(B|A) = \frac{P(AB)}{P(A)} \quad (6.3)$$

The probability of the two events,  $A$  and  $B$ , is expressed as Eq. (6.4):

$$P(AB) = P(A|B)P(B) = P(B|A)P(A) \quad (6.4)$$

The relation between the two conditional probabilities is expressed as Eq. (6.5):

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \quad (6.5)$$

Equation (6.5) is called Bayes rule.

Let us mention the likelihood as the basis for classifying data items in the probabilistic learning which is covered in this chapter.  $P(c_i|\mathbf{x})$  is the probability of the category,  $c_i$ , given the input vector,  $\mathbf{x}$ , and it is expressed as Eq. (6.6):

$$P(c_i|\mathbf{x}) = \frac{P(\mathbf{x}|c_i)P(c_i)}{P(\mathbf{x})} \quad (6.6)$$

where  $P(c_i|\mathbf{x})$  is called posteriori,  $P(\mathbf{x}|c_i)$  is called likelihood, and  $P(c_i)$  is priori, by the Bayes rule which are mentioned above. We are interested in finding the maximum posteriori for classifying the data item,  $\mathbf{x}$ , so the term,  $P(\mathbf{x})$ , is ignored;

the posteriori is expressed as Eq. (6.7):

$$P(c_i|\mathbf{x}) \propto P(\mathbf{x}|c_i)P(c_i). \quad (6.7)$$

The priori is assumed as identical values over categories expressed as Eq. (6.8):

$$P(c_1) = P(c_2) = \dots = P(c_m) \quad (6.8)$$

so the posteriori is expressed as Eq. (6.9):

$$P(c_i|\mathbf{x}) \propto P(\mathbf{x}|c_i). \quad (6.9)$$

A data item is classified into a category, depending on the maximum likelihood.

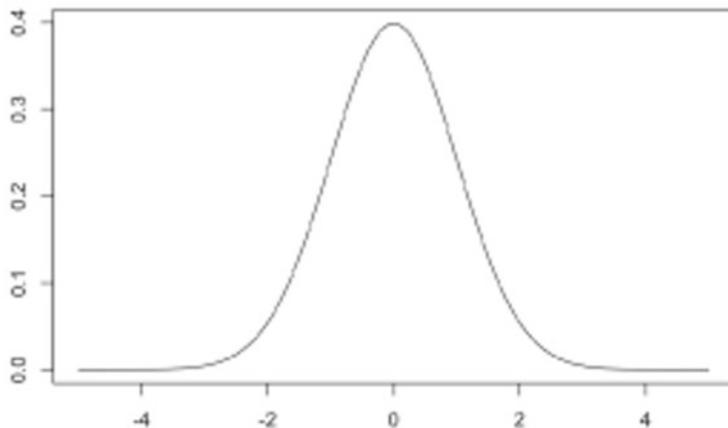
Let us mention what is intended in this chapter. We review the Bayes classifier and the Naive Bayes as the swallow learning algorithm. We modify them into their deep version by attaching the input encoding and the output encoding. We modify them into their unsupervised versions and construct their stacked versions by combining their original versions and their unsupervised versions as the more advanced deep learning algorithms. This chapter is intended to understand the process of modifying the Bayes classifier and the Naive Bayes into its deep versions.

## 6.2 Swallow Version

This section is concerned with the swallow version of the probabilistic learning algorithms before modifying them into their deep versions. In Sect. 6.2.1, we mention the normal distribution for providing the background for understanding the probabilistic learning. In Sect. 6.2.2, we mention the Bayes classifier where each category is assumed to be a normal distribution. In Sect. 6.2.3, we mention the Naive Bayes where attributes are assumed to be independent of each other. In Sect. 6.2.4, we mention the Bayesian networks which represent causal relations among attributes.

### 6.2.1 Normal Distribution

This section is concerned with the normal distribution as a continuous probability distribution. Each category is assumed to be a normal distribution most popularly by the central limit theorem. The normal distribution over vectors is characterized by its mean vector and its covariance matrix. It is used for computing the likelihoods of a vector given the categories. This section is intended to describe the normal distribution which is defined as a probability distribution for each category.



**Fig. 6.1** Scalar normal distribution

The normal distribution over scalar values is illustrated in Fig. 6.1. The distribution is characterized by its mean,  $\mu$ , and its variance,  $\sigma^2$ . In the normal distribution, the probability of  $x$  is computed by Eq. (6.10):

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \quad (6.10)$$

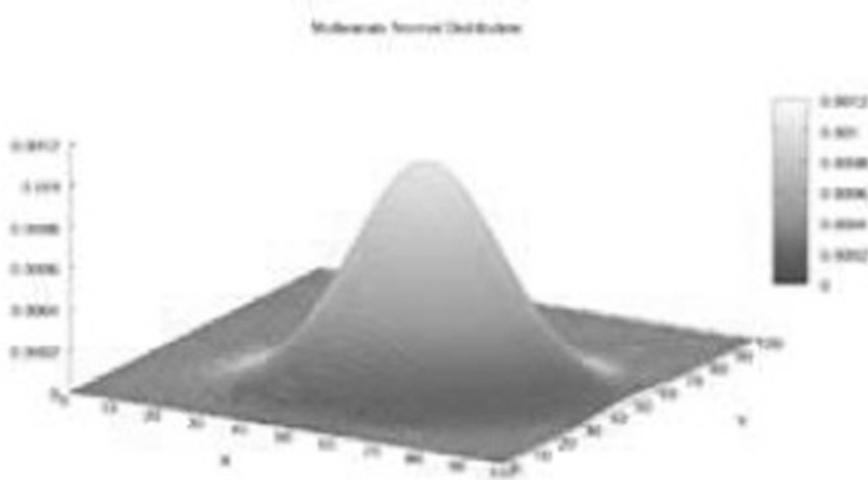
When  $x = \mu$ , the probability in the normal distribution is maximal, and if the variance,  $\sigma^2$ , is larger, the normal distribution becomes broader. When the variance,  $\sigma^2$ , is fixed, the probability in the normal distribution is dependent on the difference between the mean and the value.

The normal function over the two-dimensional vectors is illustrated in Fig. 6.2. The distribution is characterized by its mean vector,  $\boldsymbol{\mu}$ , and its covariance matrix,  $\boldsymbol{\Sigma}$ . The process of computing the probability of a particular vector,  $\mathbf{x}$ , is expressed by Eq. (6.11):

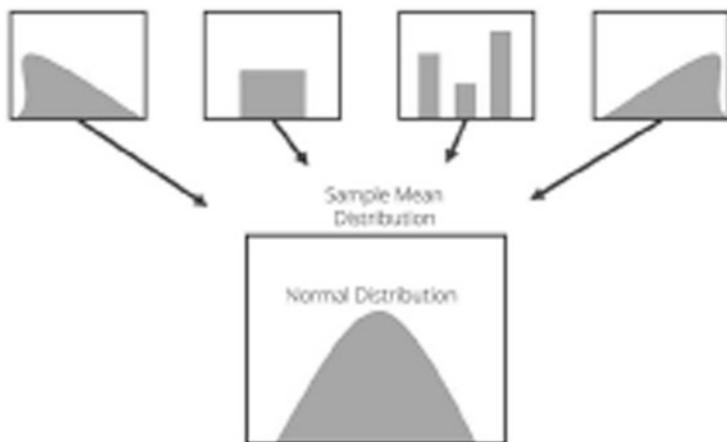
$$f(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^d |\boldsymbol{\Sigma}|}} \exp\left(-\frac{1}{2} (\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})\right) \quad (6.11)$$

If  $\boldsymbol{\mu} = \mathbf{x}$ , the probability in the normal distribution is maximal, and if the determinant of the covariance matrix is larger, the normal distribution becomes broader. For each category, the normal distribution is defined with a constant covariance matrix for simplicity, in computing the likelihood of an example.

The central limit theorem is visualized in Fig. 6.3. It becomes the reason of adopting the normal distribution for usually defining a continuous probability distribution. It states that the mixture of various probability distributions gets close to the normal distribution as shown in Fig. 6.3. If a random value is generated almost infinite



**Fig. 6.2** Vector Gaussian distribution from [https://en.wikipedia.org/wiki/Multivariate\\_normal\\_distribution](https://en.wikipedia.org/wiki/Multivariate_normal_distribution)



**Fig. 6.3** Central limit theorem from <https://morioh.com/p/9c7c90a3c8e3>

times, the distribution over the generated values follows the normal distribution. The normal distribution is usually defined for computing the likelihoods of the input vector to the categories in the probability learning.

Let us make some remarks on the normal distribution as the most popular continuous probability distribution. A scalar normal distribution is characterized with its mean and its variance in the statistical view. The scalar normal distribution may be expanded into the vector normal distribution, and it is characterized

with its mean vector and its covariance matrix. The central limit theorem is the reason of adopting the normal distribution among various continuous probability distributions. In the probabilistic learning, a single normal distribution to multiple categories or multiple normal distributions within a single category may be defined.

### 6.2.2 Bayes Classifier

This section is concerned with the Bayes classifier as a swallow learning algorithm. We studied the normal distribution which is characterized with its mean vector and its covariance matrix, and each category is defined as a normal distribution. In the Bayes classifier, the probability of a novice vector given a category, called likelihood, is computed and the novice input is classified into the category with its maximal likelihood. The learning process of the Bayes classifier is to define a normal distribution for each category, using the training examples. This section is intended to describe the Bayes classifier as its swallow version.

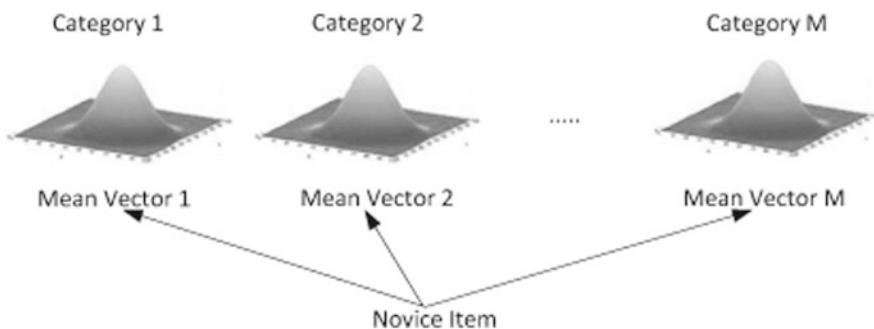
The process of defining a normal distribution for each category is illustrated in Fig. 6.4. In the Bayes classifier, we need to define a continuous probability distribution for each category. The normal distributions which are defined over the categories are characterized by their mean vectors and their covariance matrices. The  $M$  mean vectors and the  $M$  covariance matrices are obtained to the  $M$  predefined categories. Their parameters are used for computing the likelihoods of a novice input vector to the categories for classifying it.

The classification of a novice item by the Bayes classifier which considers only mean vectors is illustrated in Fig. 6.5. The covariance matrix is assumed to be fixed, and the likelihood of a novice item to the category is computed by Eq. (6.11). The differences of a novice input vector from the mean vectors are the important factors for deciding the probability values in considering the only mean vector. The category of a novice input vector,  $\mathbf{x}$ , is decided by its difference from the mean vectors among the categories,  $c_1, c_2, \dots, c_M$ , as shown in Eq. (6.12):

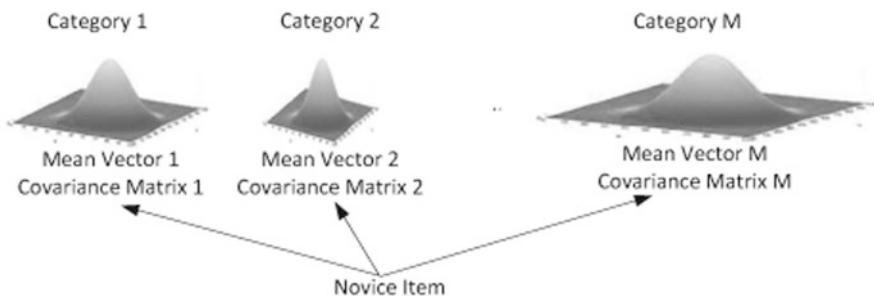
$$c_{\min} = \operatorname{argmin}_{i=1}^M \|\mathbf{x} - \boldsymbol{\mu}_i\| \quad (6.12)$$



**Fig. 6.4** Normal distributions over categories



**Fig. 6.5** Normal distributions with identical covariance matrices over categories



**Fig. 6.6** Normal distributions with different covariance matrices over categories

The determinants of the covariance matrices are variable depending on the categories in real classification tasks.

The process of classifying a novice item by the Bayes classifier, considering both the mean vectors and the covariance matrices, is illustrated in Fig. 6.6. The complexity of computing the determinant and the inverse matrix of the covariance matrix which are shown in Eq. (6.11) is very high. It is assumed that the covariance matrices are diagonal matrices in each of which only diagonal elements are non-zero values. The determinant is computed by multiplying the diagonal elements, and the inverse matrix is computed by replacing them by their multiplication reverses. The assumption of the covariance matrix as a diagonal matrix means that the attributes are independent of each other.

Let us make some remarks on the Bayes classifier as a simple probabilistic learning algorithm. Each category is assumed as a normal distribution which is characterized by its mean vector and its covariance matrix. If the only mean vector is considered, each data item is classified, depending on its similarity with the mean vector. If the mean vector and the covariance matrix are considered, the covariance matrix is assumed as a diagonal matrix for easy computation. In the Bayes classifier, we consider defining multiple categories as a single normal distribution or defining a single category as multiple normal distributions.

### 6.2.3 Naive Bayes

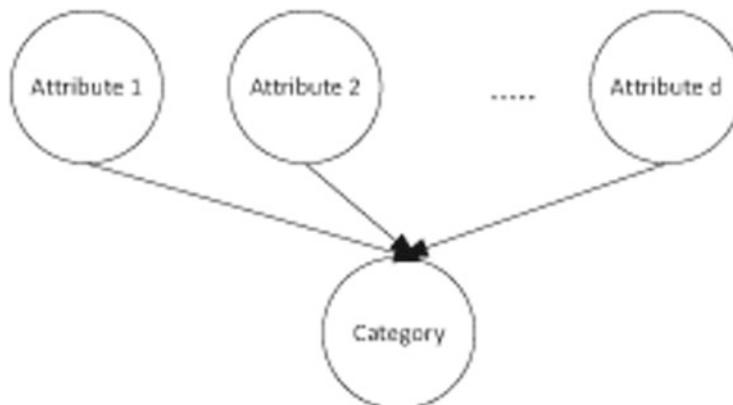
This section is concerned with the Naive Bayes as another probabilistic learning algorithm. In the Bayes classifier, the likelihood of the input vector to the category is computed by the probability of the input vector in the normal distribution which represents the category. In the Naive Bayes, the likelihood of an individual attribute value to the category is computed by the ratio of the number of training examples with the attribute value to the number of training examples in the category. The likelihood of the input vector is computed by the product of the likelihoods of its elements with the assumption of the independence among the attribute values. This section is intended to describe the Naive Bayes as the alternative to the Bayes classifier.

The Bayesian networks which represent the Naive Bayes are illustrated in Fig. 6.7 with the assumption of the independence of attributes. If the two events,  $A$  and  $B$ , are independent of each other, the probability of  $A$  and  $B$  is expressed as Eq. (6.13):

$$P(AB) = P(A)P(B). \quad (6.13)$$

The attributes are notated by  $A_1, A_2, \dots, A_d$ , the discrete values are assumed in each attribute, and each attribute is viewed as a set of values,  $A_i = \{v_{i1}, v_{i2}, \dots, v_{i|A_i|\}}$ . The input vector is notated by  $\mathbf{x} = [x_1 \ x_2 \ \dots \ x_d]$ , and the likelihood,  $P(\mathbf{x}|C)$ , is expressed by the product of the likelihoods of individual attribute values as shown in Eq. (6.14):

$$P(\mathbf{x}|C) = \prod_{i=1}^d P(x_i|C). \quad (6.14)$$



**Fig. 6.7** Bayesian networks in Naive Bayes

It is necessary to compute the likelihoods of individual values for classifying a data item.

Let us mention the process of computing the likelihoods of individual values from the training examples. The categories are predefined as  $c_1, c_2, \dots, c_M$ , and the input vector is given as a  $d$  dimensional vector,  $\mathbf{x} = [x_1 \ x_2 \ \dots \ x_d]$ . The probability,  $P(x_i|c_j)$ , is computed by Eq. (6.15):

$$P(x_i|c_j) = \frac{P(x_i \wedge c_j)}{P(c_j)} = \frac{\#(x_i \wedge c_j)}{\#(c_j)}, \quad (6.15)$$

where  $\#(x_i \wedge c_j)$  is the number of training examples with  $A_i = x_i$  within the category,  $c_j$ , and  $\#(c_j)$  is the number training examples within the category,  $c_j$ . The likelihoods,  $P(x_1|c_j), P(x_2|c_j), \dots, P(x_d|c_j)$ , are computed by Eq. (6.15) to the elements in the vector. The likelihoods are involved in classifying a data item into a category.

Let us mention the process of classifying a data item with the likelihoods of individual values. The likelihood of the input vector is computed by Eq. (6.14). The likelihoods of the input vector to the categories are computed as  $P(\mathbf{x}|c_1), P(\mathbf{x}|c_2), \dots, P(\mathbf{x}|c_M)$ . The category of the input vector is decided by the maximum likelihood which is expressed by Eq. (6.16):

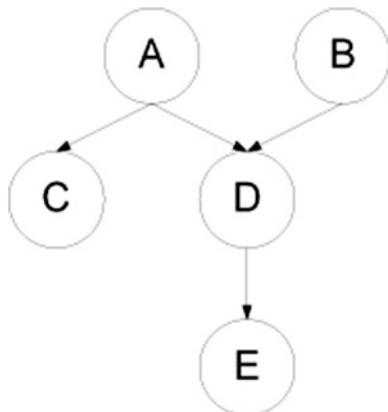
$$c_{\max} = \operatorname{argmax}_{j=1}^M P(\mathbf{x}|c_j). \quad (6.16)$$

Equation (6.14) may be adjusted in a variant of Naive Bayes into equation (6.17):

$$P(\mathbf{x}|c_j) = \frac{1}{d} \sum_{i=1}^d P(x_i|c_j). \quad (6.17)$$

Let us make some remarks on the Naive Bayes as a more advanced probabilistic learning. The assumption of the independence among the attributes characterizes the Naive Bayes. The likelihood of an individual value to a category is rate of the number of training examples which correspond to the attribute and the category to the rate of number of training examples which belong to the category. The likelihood of an input vector to the category is the product of likelihoods of individual values, and the input vector is classified into a category by its maximum likelihood. Some variants of Naive Bayes may be derived by proposing alternative schemes of computing individual value likelihoods and alternative schemes of computing the input vector likelihood.

**Fig. 6.8** General Bayesian networks



#### 6.2.4 Bayesian Networks

This section is concerned with the Bayesian networks as the advanced probabilistic learning algorithm. In the Naive Bayes, the complete independence among the attributes is assumed for computing the likelihoods of the input vector to the categories. In the Bayesian networks, the causal relations among the attributes are defined as a directed graph. The conditional probability table is defined in each node. This section is intended to describe the Bayesian networks as the most advanced learning algorithm.

The Bayesian networks is illustrated as an example in Fig. 6.8. The variables, A and B, are independent of each other. C depends on only A, D depends on A and B, and C and D are independent of each other. E depends on D, but independent of C. A and B are ancestors of E, but E is independent of A and B.

The probability table which is defined in a node of the Bayesian networks is illustrated as an example in Fig. 6.9. It is assumed that C depends on A, and both A and C are given as binary values. Because A is the initial variable,  $P(A = 0)$  and  $P(A = 1)$  are defined in A. The four conditional probabilities are defined in C. The probability table is constructed by the training examples as the learning process.

Let us consider the learning process in the Bayesian networks. The relations among the attributes are defined from the training examples; the causal relation between attributes or their independence is decided by the support and the confidence. The initial probabilities of the attribute values are computed to the starting attributes, and the conditional probabilities of causal attribute values given conditional attribute values are computed. The probability of a vector is computed by product of conditional probability and initial probability for classifying a novice item. Refer to [1] for studying in detail the Bayesian networks.

Let us make some remarks on the Bayesian networks as the advanced probabilistic learning. The Bayesian networks is viewed as a directed graph where each vertex is an attribute, and each edge is a causal relation. The probability table

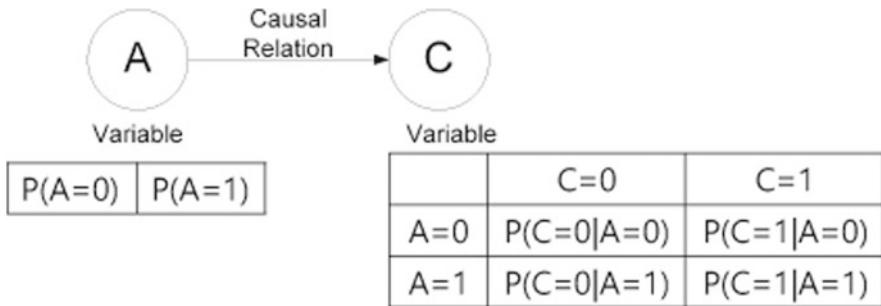


Fig. 6.9 Conditional probability table in Bayesian networks

which consists of the conditional probability values is defined in each vertex. The process of training the Bayesian networks with the training examples consists of the two steps: construction of causal relations among attributes and definition of a probability table for each attribute. The causal relation among hidden variables may be constructed in modifying the Bayesian networks into a deep version.

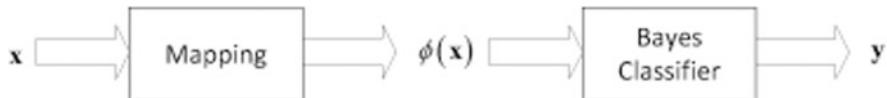
### 6.3 Basic Deep Versions

This section is concerned with the deep version of the probabilistic learning algorithms which are added by the input encoding or the output encoding. In Sect. 6.3.1, we mention the kernel-based version of Bayes classifier. In Sect. 6.3.2, we mention the Bayes classifier which is added by the pooling layer. In Sect. 6.3.3, we mention the version of Naive Bayes which is added by the output decoding. In Sect. 6.3.4, the polling layer is added to the Naive Bayes, as well as the Bayes classifier.

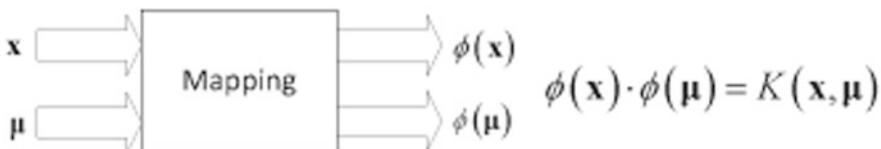
#### 6.3.1 Kernel-Based Bayes Classifier

This section is concerned with the kernel-based version of Bayes classifier. The kernel function is usually applied to two vectors for computing the inner product of their mapped vectors. In this version of Bayes classifier, the similarity between a mean vector and a novice vector is computed by the kernel function. The process of mapping an input vector into one in another space is viewed as the process of computing a hidden vector from an input vector. This section is intended to describe the version of Bayes classifier which applies the kernel function of an input vector and a mean vector.

The process of mapping the input vector into one in another space is viewed as the process of computing a hidden vector, as shown in Fig. 6.10. The original



**Fig. 6.10** Kernel-based Bayes classifier



**Fig. 6.11** Kernel function of novice vector and mean vector

input vector is noted by  $\mathbf{x}$ , and its mapped one is notated by  $\Phi(\mathbf{x})$ . The mapping,  $\mathbf{x} \rightarrow \Phi(\mathbf{x})$ , is regarded as the process of computing the hidden vector,  $\mathbf{h} = \Phi(\mathbf{x})$ , satisfying the definition of deep learning. The Bayes Classifier is applied to the mapped vector,  $\Phi(\mathbf{x})$ , instead of the input vector,  $\mathbf{x}$ , in this version. The mean vector of the category,  $c_i, \boldsymbol{\mu}_i$ , is mapped into the vector,  $\Phi(\boldsymbol{\mu}_i)$ .

The view of the inner product of the mean vector and the input vector into the kernel function of them is illustrated in Fig. 6.11. The novice input vector is classified in the Bayes classifier depending on its similarity with the mean vectors. Both the mean vector,  $\boldsymbol{\mu}_i$ , and the input vector,  $\mathbf{x}$ , are mapped, respectively, into the vectors in another space,  $\Phi(\boldsymbol{\mu}_i)$ , and  $\mathbf{x}$  is expressed by the kernel function as shown in Eq. (6.18):

$$\Phi(\mathbf{x}) \cdot \Phi(\boldsymbol{\mu}_i) = K(\mathbf{x}, \boldsymbol{\mu}_i). \quad (6.18)$$

The classification process in the kernel-based Bayes classifier depends on the kernel function of the input vector and the mean vector.

The process of classifying a data item by the kernel-based Bayes classifier is illustrated as a pseudo code in Fig. 6.12. A list of categories and a data item are given as the arguments. The inner product of the mapped input vector and the mapped mean vector is computed by the kernel function of the input vector and the mean vector by Eq. (6.18). The novice input vector is classified into the category whose kernel value is maximal. The hidden vectors are implicitly computed in this version of Bayes classifier.

Let us make some remarks on the kernel-based version of the Bayes classifier. The input vector is mapped into one in another space, in computing the inner product of mapped vectors by a kernel function. In this version of Bayes classifier, the similarity between the input vector and a mean vector is computed by applying the kernel function to them. A novice item is classified into the category where the output value of the kernel function of them is maximal. This version of the Bayes classifier is viewed as a deep learning algorithm, in that the kernel function is the inner product of the mapped vectors as hidden vectors.

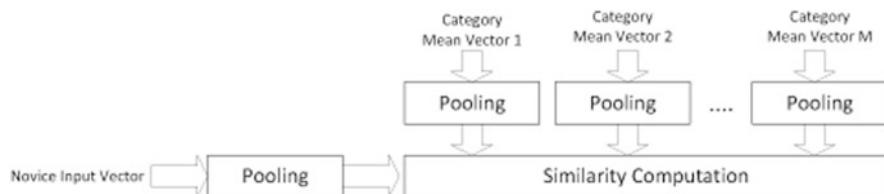
```

Category classifyByKernelBayesClassifier(item dataItem, List categoryList){
    int size = categoryList.size();
    double maxKernel = 0.0;
    int maxIndex = 0;

    for(int i = 0;i < size;i++){
        Category cc = categoryList.getElement(i);
        Item meanVector = cc.getMeanVector();
        double kernel = dataItem.getKernel(meanVector);
        if(kernel > maxKernel){
            kernel = maxKernel;
            maxIndex = i;
        }
    }
    return categoryList.getElement(maxIndex);
}

```

**Fig. 6.12** Classification process by kernel Bayes classifier

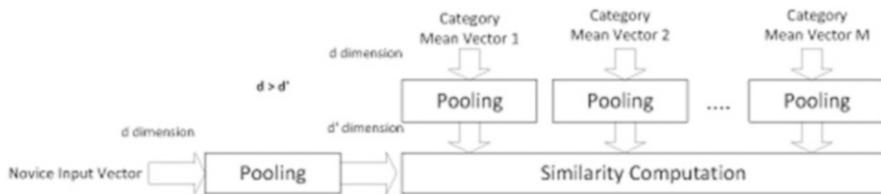


**Fig. 6.13** Pooled Bayes classifier architecture

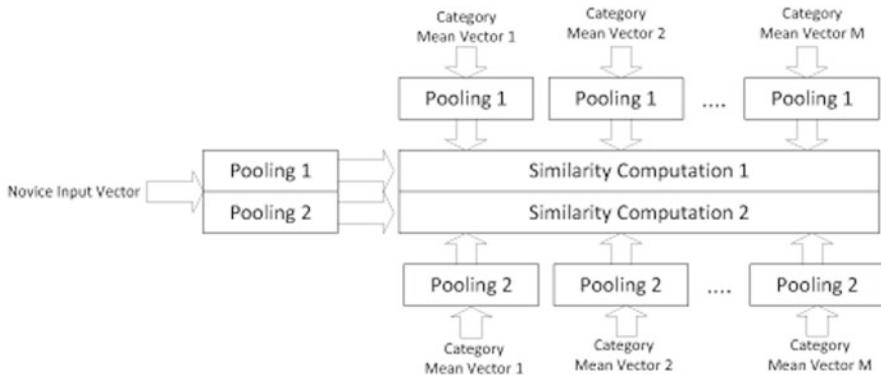
### 6.3.2 Pooling-Based Bayes Classifier

This section is concerned with another deep version of Bayes classifier, called pooling-based Bayes classifier. In the previous section, we studied the kernel-based Bayes classifier where the kernel function is applied to the input vector and the mean vector. In this section, we study the deep version of Bayes classifier where the pooling operation is attached in front of the Bayes classifier. The pooling which is an operation on a vector retrieves a representative value from the window which slides on the vector, for making another vector. This section is intended to describe the Bayes classifier to which the pooling layer is attached in its front.

The architecture of this version of Bayes classifier is illustrated in Fig. 6.13. The pooling is the operation for reducing the input dimension by retrieving a representative value from the window which slides on the input vector. The pooling is applied to the novice vector and the mean vectors which characterize their own categories. The similarities of the novice vector with the mean vectors are computed through the pooling in this version. The pooling for modifying the Bayes classifier into its deep version is the module of computing a hidden vector from the input vector.



**Fig. 6.14** Dimension reduction by pooled Bayes classifier



**Fig. 6.15** Multiple pooled Bayes classifier

The dimension reduction by the pooling is illustrated in Fig. 6.14. The pooling is mentioned as the operation which maps a vector into one with its lower dimensionality. If the dimension of the input vector is  $d$ , and the sliding window size is  $s$ , the dimension of the vector which is mapped by the pooling becomes  $d - s + 1$ . If the softmax pooling operation is used, the maximal among  $s$  values in the window is filled in  $d - s + 1$  elements. The  $d - s + 1$  dimensional vector which is mapped by the pooling from the  $d$  dimensional input vector becomes a hidden vector.

The version of Bayes classifier where multiple pooling operations are used is illustrated in Fig. 6.15. Various kinds of pooling operations such as the softmax pooling and the average pooling exist, and each of them is differentiated with a different window size. If two different pooling operation is used, two similarities of the novice input with a mean vector are computed. Using two different pooling operations is viewed as applying two Bayes classifiers to the classification task. We may consider using the pooling operation serially layer by layer.

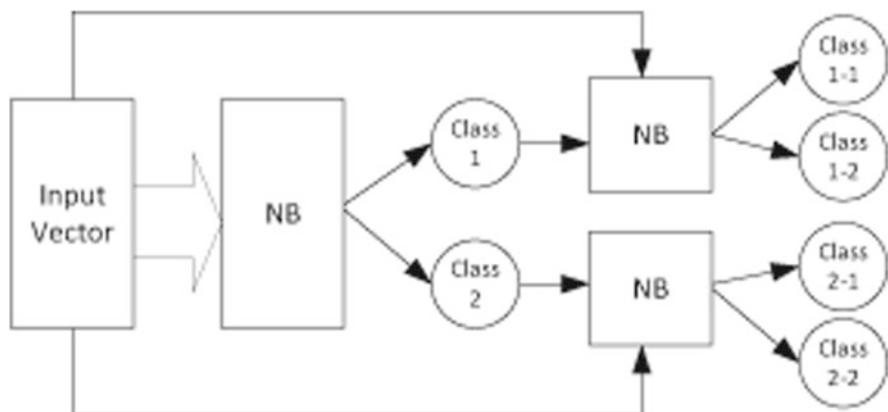
Let us make some remarks on the Bayes classifier with the pooling layer. In the architecture, the pooling layer is attached to the front of the Bayes classifier. The pooling generates the hidden vector with its lower dimensionality from the input vector as the operation for reducing the dimensionality. If the multiple pooling operations are used, different multiple hidden vectors are computed from a single input vector and a mean vector. In using multiple operations, the serial combination and the parallel combination may be chosen.

### 6.3.3 Output Decoded Naive Bayes

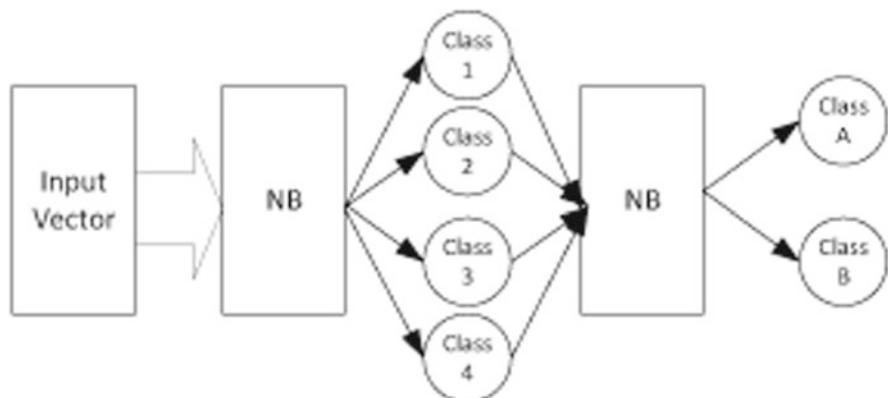
This section is concerned with the output decoded Naive Bayes as another deep version of Naive Bayes. In the previous sections, we modified the Bayes classifier which is an instance of the probabilistic learning into the deep version. In the current section and the next section, we modify the Naive Bayes which is another instance of the probabilistic learning into the deep versions. The output decoding which is viewed as the process of computing the output vector from a hidden vector is attached to the Naive Bayes. This section is intended to describe the deep version of the Naive Bayes which is modified by attaching the output decoding.

The application of the Naive Bayes to the implementation of the category specification is illustrated in Fig. 6.16. In the general level, there are the two categories, class 1 and class 2, and two specific categories are allocated to each abstract category. In the left, the Naive Bayes classifies the input vector into class 1 or class 2. The top-right Naive Bayes receives the input vector which is classified into class 1 and classifies it into class 1-1 or class 1-2, and the bottom-right Naive Bayes receives the input vector which is classified into class 2 and classifies it into class 2-1 or class 2-2. This type of deep learning is viewed as implementing the hierarchical classification with the Naive Bayes.

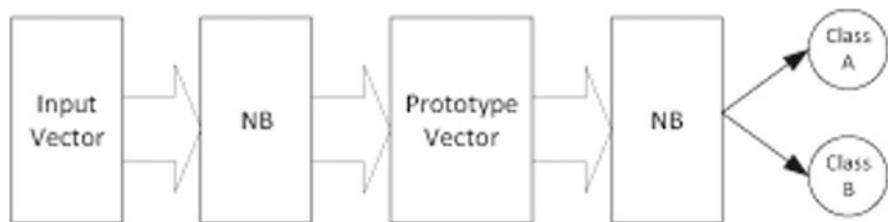
The modification of the Naive Bayes into its deep version by the category abstraction is illustrated in Fig. 6.17. In this case, it is assumed that the process of classifying an item into one among the four classes belongs to the fuzzy classification. The left Naive Bayes estimates the likelihoods of the four categories as their categorical scores. The right Naive Bayes classifies the four-dimensional vector which consists of the four likelihoods into one of the two categories. The four-dimensional vector is viewed as a hidden vector in this deep version of Naive Bayes.



**Fig. 6.16** Category specification layer in Naive Bayes



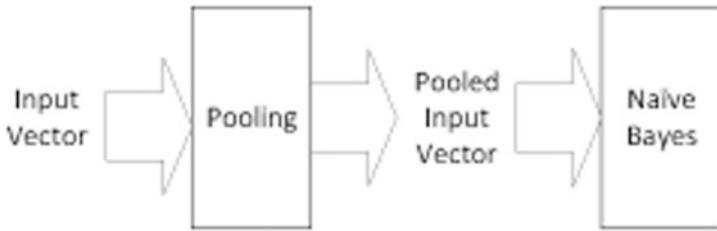
**Fig. 6.17** Category abstraction layer in Naive Bayes



**Fig. 6.18** Category prototyping layer in Naive Bayes

The modification of the Naive Bayes into its deep version by adding the category prototyping is illustrated in Fig. 6.18. The category prototyping is the process of computing a hidden vector which characterizes a category. The category prototype vector as a hidden vector is computed by the left Naive Bayes, and it is classified into class 1 or class 2 by the right Naive Bayes. The prototype vector is determined for each category by analyzing the training examples. It is possible that more than one prototype vector is available for each category.

Let us make some remarks on the modification of the Naive Bayes into its deep version by attaching the output decoding. The input vector which is augmented with a general category is a hidden vector which is classified directly in attaching the category specification to the Naive Bayes. In attaching the category abstraction to the Naive Bayes, the hidden vector is composed with the category scores of the specific categories. The category prototyping may be added for modifying the Naive Bayes into the deep version. We consider adding multiple categorical specifications and/or multiple categorical abstractions for implementing the deep learning algorithms.



**Fig. 6.19** Pooling layer in Naive Bayes

### 6.3.4 Pooled Naive Bayes

This section is concerned with the deep version of Naive Bayes with the pooling layer. In the previous section, the Naive Bayes is modified by attaching the output decoding such as the category specification and the category abstraction. In this section, it is modified by attaching the pooling layer. In this version, the likelihoods of the vector which is mapped from the input vector by the pooling layer to the categories are computed. This section is intended to describe the deep version of the Naive Bayes which is modified with the pooling layer.

The application of the pooling to the Naive Bayes is illustrated in Fig. 6.19. The pooling is the operation on a vector for generating representative values from the window which slides on it. The input vector which indicates a novice data item is mapped into another vector with its lower dimensionality by the pooling. The likelihoods of elements in the mapped vector to the categories are computed by the Naive Bayes. It is modified into its deep version by attaching the pooling in its front.

Let us mention the process of computing the likelihoods of the hidden vector to a category. The input vector,  $\mathbf{x}$ , is mapped into the hidden vector with its  $d - s + 1$  dimensions,  $\mathbf{h} = [h_1 \ h_2 \ \dots \ h_{d-s+1}]$ , by the pooling which is described in Sect. 5.3.4. The likelihood,  $P(h_i|c_j)$ , is computed by Eq. (6.19):

$$P(h_i|c_j) = \frac{P(h_i \wedge c_j)}{P(c_j)} = \frac{\#(h_i \wedge c_j)}{\#(c_j)}, \quad (6.19)$$

where  $\#(h_i \wedge c_j)$  is the number of hidden vectors mapped from training examples with  $A_i = h_i$  and  $\mathbf{x} \in c_j$  and  $\#(c_j)$  the number of hidden vectors mapped from training examples with  $\mathbf{x} \in c_j$ . The likelihoods,  $P(h_1|c_j)$ ,  $P(h_2|c_j)$ ,  $\dots$ ,  $P(h_{d-s+1}|c_j)$ , are computed by Eq. (6.19). It is required to map the training examples into hidden vectors for computing the likelihoods.

Let us mention the process of classifying a data item by this version of Naive Bayes. The input vector,  $\mathbf{x}$ , is mapped into the hidden vector,  $\mathbf{h}$ , and the likelihoods of the  $d - s + 1$  individual elements to the category,  $c_j$ , are computed. The likelihood of the hidden vector,  $\mathbf{h}$ , to the category,  $c_j$ , is computed by Eq. (6.20), following the independence among the elements.

$$P(\mathbf{h}|c_j) = \prod_{i=1}^{d-s+1} P(h_i|c_j). \quad (6.20)$$

The likelihoods of the hidden vector are computed to the predefined categories;  $P(\mathbf{h}|c_1), P(\mathbf{h}|c_2), \dots, P(\mathbf{h}|c_M)$  are computed; and the data item is classified into the category by Eq. (6.21):

$$c_{\max} = \operatorname{argmax}_{j=1}^M P(\mathbf{h}|c_j). \quad (6.21)$$

The likelihoods of the input vector are replaced by ones of the vector which is mapped by the pooling layer in this version of Naive Bayes.

Let us make some remarks on the deep version of Naive Bayes. It is modified into its deep version by attaching the pooling layer in front of it. The novice input vector is mapped into another vector by the pooling layer, and the likelihoods of the elements in the hidden vector to each category are computed. The likelihoods of the hidden vector to the predefined categories are computed, and the input vector is classified into the category whose likelihood is maximal. More advanced deep versions of Naive Bayes are implemented by attaching the pooling layer and the convolution layer, alternatively.

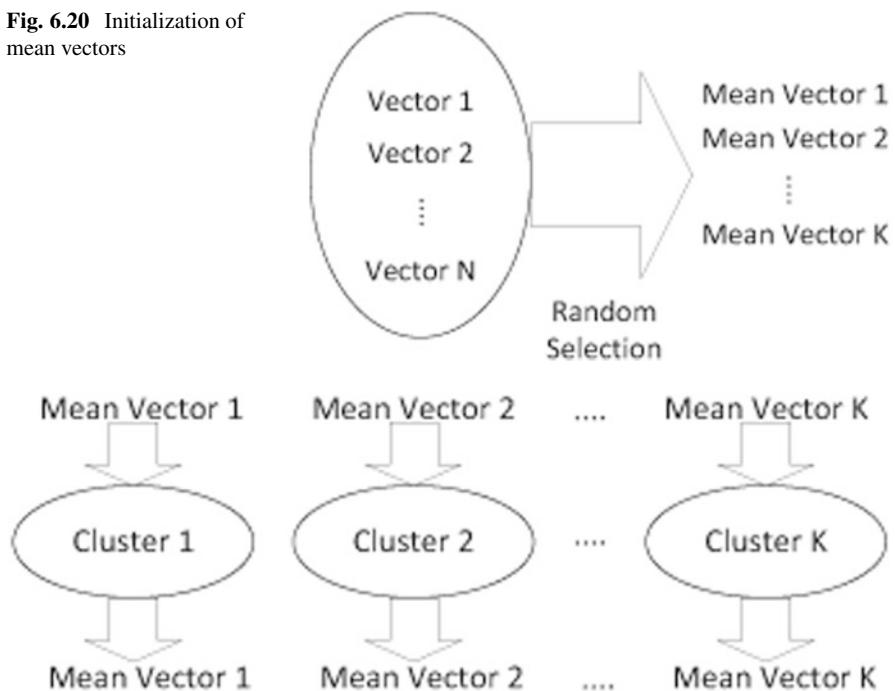
## 6.4 Advanced Deep Versions

This section is concerned with the advanced deep version of the Naive Bayes and the Bayes classifier by adding their unsupervised layer. In Sect. 6.4.1, we mention the k means algorithm as the unsupervised version of the Bayes classifier. In Sect. 6.4.2, we modify the Naive Bayes into its unsupervised version. In Sect. 6.4.3, we mention the deep version of Bayes classifier which is modified by adding the unsupervised layer. In Sect. 6.4.4, we add the unsupervised layer to the Naive Bayes as well as the Bayes classifier.

### 6.4.1 Unsupervised Bayes Classifier

This section is concerned with the unsupervised version of Bayes classifier. A data item is classified depending on its similarities with the mean vectors in the swallow version. If unlabeled examples are prepared, the normal distributions are defined at random. The normal distribution on vectors is characterized by its mean vector and its covariance matrix, but only mean vector will be considered for the simplicity. This section is intended to describe the unsupervised Bayes classifier for implementing the deep version of Bayes classifier.

**Fig. 6.20** Initialization of mean vectors

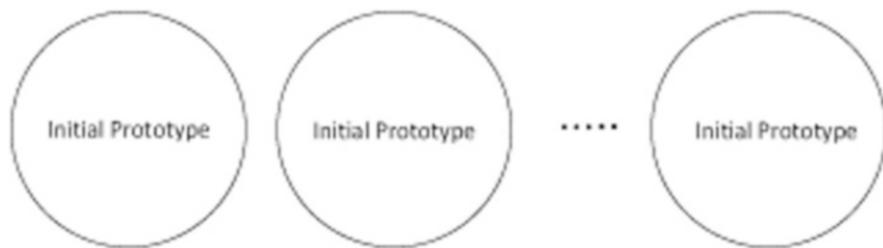


**Fig. 6.21** Update of mean vectors

The decision of the initial mean vectors by selecting some at random is illustrated in Fig. 6.20. The preparation of unlabeled examples is assumed modifying the supervised learning algorithm into its unsupervised version. The role of the initial mean vectors which are selected at random is the basis for clustering data items, called seed examples. The probability distributions over clusters are characterized by the selected mean vectors in the unsupervised Bayes classifier. From the unlabeled examples, the seed examples are prepared for modifying the supervised learning algorithm into its unsupervised version.

The process of updating the cluster mean vectors in the unsupervised version of Bayes classifier is illustrated in Fig. 6.21. The normal distributions over clusters are characterized by the initial mean vectors. Data items are arranged into clusters, depending on their similarities with the mean vectors. The normal distributions over the clusters are characterized again by computing the mean vectors and the covariance matrices. The covariance matrices are assumed as constant ones, so we consider only mean vector in characterizing the normal distributions.

Let us mention the process of clustering data items by the unsupervised version of Bayes classifier. The cluster distributions are initially characterized by the prior knowledge or random data items. The likelihoods of each data item to the clusters are computed, it is arranged into the cluster with its maximum likelihood, and the cluster distributions are characterized again after arranging data items. They are



**Fig. 6.22** Initialized clusters in unsupervised Naive Bayes

clustered by iterating arranging data items based on their likelihoods and updating the cluster distributions. The clustering process by the unsupervised Bayes classifier is similar as that by the k means algorithm which was studied in Chap. 3.

Let us make some remarks on the unsupervised version of Bayes classifier. The process of initializing the cluster distributions is to decide the number of clusters in advance and define the initial normal distributions as many as clusters. The data items are arranged depending on their probabilities in the cluster distributions, and their parameters are updated. The process of clustering data items with the unsupervised version is to iterate the arrangement of data items and the update of parameters. The process of clustering data items with it is viewed as the process of doing them with the k means algorithm which is covered in Chap. 3.

#### 6.4.2 Unsupervised Naive Bayes

This section is concerned with the unsupervised version of Naive Bayes. In the previous section, the Bayes classifier is modified into its unsupervised version. In this section, we modify the Naive Bayes as another probabilistic learning algorithm into its unsupervised version. It is required to decide the number of clusters in advance and select seed examples at random for using the unsupervised version. It is intended to describe the unsupervised version of Naive Bayes.

The initial clusters for using the unsupervised Naive Bayes are illustrated in Fig. 6.22. In advance, the number of clusters is decided. Data items as many as clusters are selected at random as the initial cluster prototypes. They are used for computing the likelihoods of each data item to the clusters. The likelihood of an individual attribute to each cluster is computed from the initial cluster prototypes.

Let us mention the process of computing the likelihood of an individual attribute value to the clusters. The prototype of the cluster,  $C_i$  is notated by a vector,  $\mathbf{c}_i = [c_{i1}, c_{i2}, \dots, c_{id}]$ , and it is assumed that a normal distribution is defined independently for each element. The likelihood of the attribute value,  $x_j$ , to the cluster,  $C_i$ , is  $P(x_j|C_i)$ , which is computed by Eq. (6.22):

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x_j - c_{ij})^2}{2\sigma^2}\right) \quad (6.22)$$

The variance is assumed as 1.0, and Eq. (6.22) is transformed into Eq. (6.23):

$$f(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(x_j - c_{ij})^2}{2}\right) \quad (6.23)$$

The difference from an attribute value and the corresponding element of the cluster prototype influences on the likelihood.

Let us mention the process of computing the likelihoods of a vector to the clusters. A particular data item in a group is given as a  $d$  dimensional vector,  $\mathbf{x} = [x_1 \ x_2 \ \dots \ x_d]$ . The likelihood of input vector,  $\mathbf{x}$ , to the cluster,  $C_i$ , is computed by Eq. (6.24):

$$P(\mathbf{x}|C_i) = \prod_{j=1}^d P(x_j|C_i). \quad (6.24)$$

The likelihoods are computed to all clusters as  $P(\mathbf{x}|C_1)$ ,  $P(\mathbf{x}|C_2)$ ,  $\dots$ ,  $P(\mathbf{x}|C_M)$ , and the data item is arranged into the cluster,  $C_{\max}$ , by Eq. (6.25):

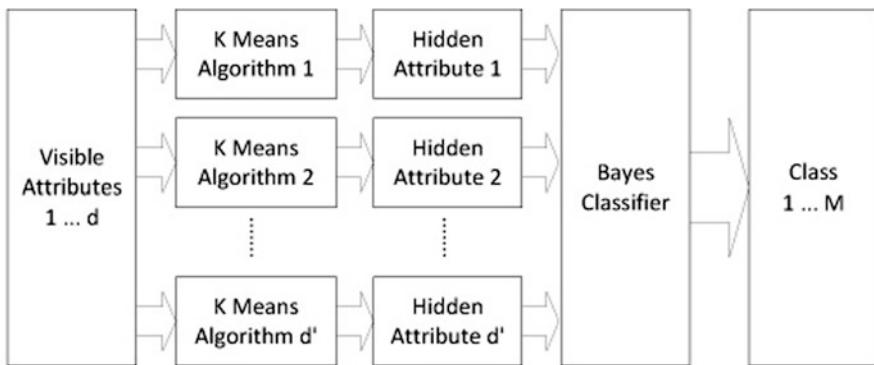
$$C_{\max} = \operatorname{argmax}_{i=1}^M P(\mathbf{x}|C_i). \quad (6.25)$$

The process of computing the likelihoods of individual attribute to the clusters is the difference from the supervised version.

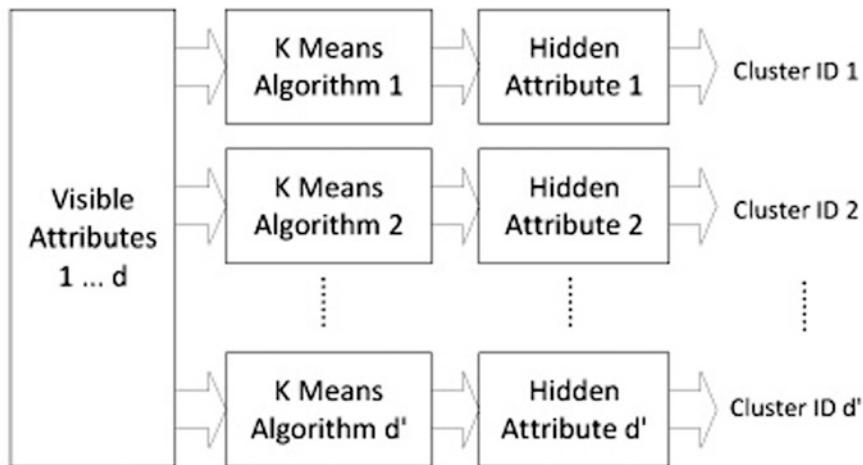
Let us make some remarks on the unsupervised version of Naive Bayes for clustering data items. The clusters are initialized with the decision of their initial prototypes at random. The likelihood of an attribute value to a cluster is computed by its independent normal distribution. The likelihood of vector is computed by product of ones of attribute values. We consider updating continually clusters by applying the supervised Naive Bayes to the data items until its convergence.

### 6.4.3 Stacked Bayes Classifier

This section is concerned with the stacked Bayes classifier as the deep learning algorithm. In Sect. 6.4.1, we studied the unsupervised version of Bayes classifier which is similar as the k means algorithm. In this section, we combine the unsupervised versions and the supervised version for implementing the deep learning. The unsupervised versions are used for computing the hidden values, and the supervised



**Fig. 6.23** Architecture of deep Bayes classifier with unsupervised layer



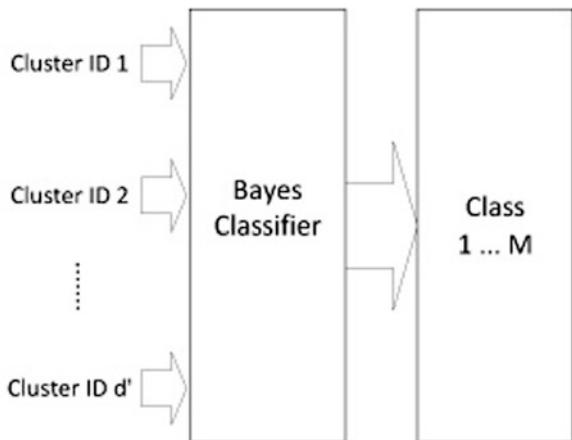
**Fig. 6.24** K means algorithms for computing hidden vector

version is used for computing the output values in the deep version. This section is intended to describe the deep learning algorithm based on the Bayes classifier.

The stacked architecture of Bayes classifier for implementing the deep learning is illustrated in Fig. 6.23. Because the k means algorithm is very similar as the unsupervised Bayes classifier, the k means algorithms are adopted for computing the hidden vector from the input vector. The probability distribution of hidden vector is defined for each category as a normal distribution. The hidden vector which is mapped from the input vector by the k means algorithms is classified into one among the  $M$  classes. If another distribution is defined, the unsupervised Bayes classifier may be different from the k means algorithm.

The unsupervised layer which involves the multiple k means algorithms is illustrated in Fig. 6.24. The unsupervised layer is composed with the  $d'$  k means algorithms for computing the hidden vector. Each element in the hidden vector

**Fig. 6.25** Bayes classifier for classifying hidden vector



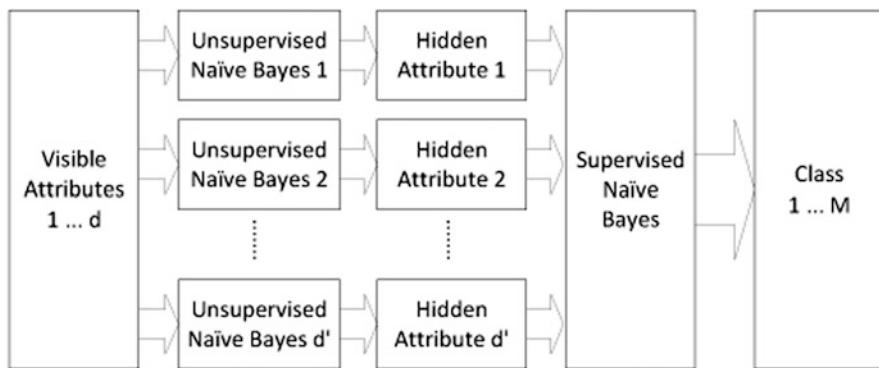
indicates a cluster identifier which is given as an integer. The hidden vector which is computed from the input vector consists of  $d'$  integers. The hidden vector is classified into one among the predefined categories.

The supervised layer which classifies the hidden vector is illustrated in Fig. 6.25. It is composed with only one Bayes classifier for carrying out the multiple classification. In the training phase, the training examples are transformed into hidden vector through the multiple k means algorithm, and a normal distribution is defined for each category by the hidden vectors. In the generalization phase, the novice input vector is transformed into the hidden vector, its similarities with the mean vectors are computed, and it is classified into the category whose mean vector is most similar. The Bayes classifier will be replaced by the Naive Bayes for dealing with the hidden vectors in the next section.

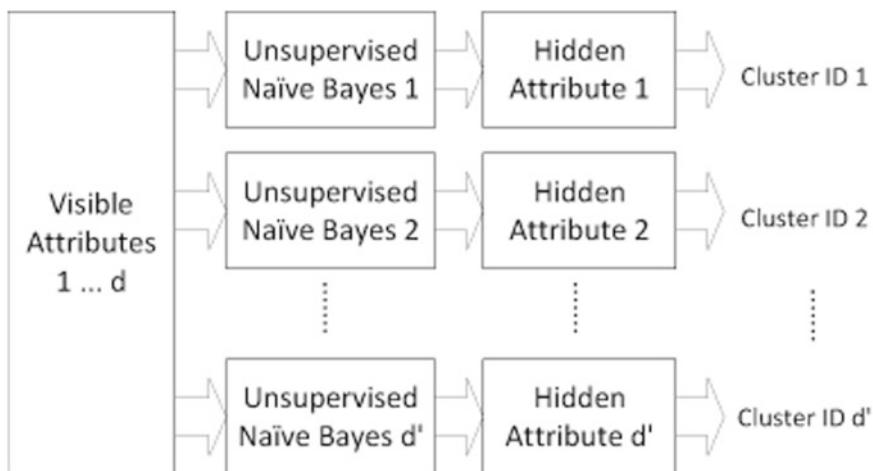
Let us make some remarks on the stacked version of Bayes classifier as a deep learning algorithm. In the architecture of this deep version, multiple k means algorithms are involved in computing the hidden vector, and a single Bayes classifier is involved in classifying it. Each element of the hidden vector which is computed by the multiple k means algorithms is given as a cluster identifier. The hidden vector which is mapped from the novice input vector is classified by the single Bayes classifier. The process of computing the hidden vector is the fuzzy clustering, and a single fuzzy k means algorithm is involved in the unsupervised layer.

#### 6.4.4 Stacked Bayes Classifier

This section is concerned with the stacked Bayes classifier as a deep learning algorithm. In Sect. 6.4.1, we studied the unsupervised version of Bayes classifier which is similar as the k means algorithm. In this section, the unsupervised versions and the supervised version are combined for implementing the deep learning on



**Fig. 6.26** Architecture of Naive Bayes with unsupervised layer



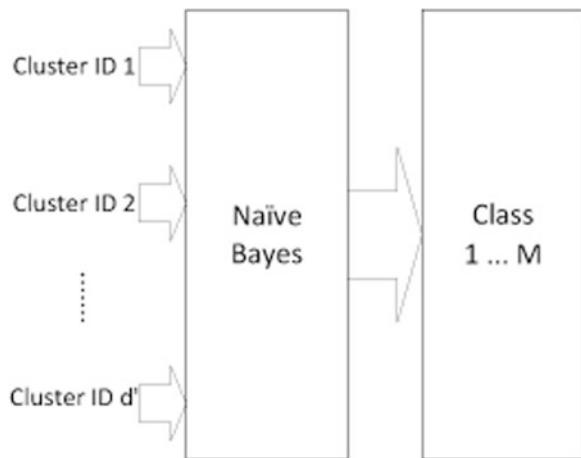
**Fig. 6.27** Unsupervised Naive Bayes for computing hidden vector

the Bayes classifier. The unsupervised versions are used for computing the hidden vector, and the supervised one is used for computing the output vector in this version. This section is intended to describe the deep learning algorithm which is based on the Bayes classifier.

The architecture of this deep version of Naive Bayes is illustrated in Fig. 6.26. The  $d'$  unsupervised Naive Bayes are involved in computing the hidden vector from the input vector. The hidden vector is classified by a single supervised Naive Bayes. One cluster is decided for each data item in applying an unsupervised Naive Bayes, assuming the hard clustering. The Bayes classifier is replaced by the Naive Bayes as another probabilistic learning in this deep learning algorithm.

The unsupervised layer in the deep version of Naive Bayes is illustrated in Fig. 6.27. The role of each unsupervised Naive Bayes is to compute an element

**Fig. 6.28** Naïve Bayes for classifying hidden vector



of the hidden vector. Each element is a cluster identifier into which the data item is arranged by an unsupervised Naïve Bayes. If the dimension of hidden vector is  $d'$ , the  $d'$  unsupervised Naïve Bayes models are involved in computing the hidden vector. In the unsupervised layer, the input vector with the  $d$  dimension is transformed into a hidden vector with the  $d'$  dimension.

The process of classifying a novice data item in the supervised layer is illustrated in Fig. 6.28. The hidden vector which consists of cluster identifiers is computed from the input vector in the unsupervised layer. The likelihoods of individual attributes to the categories are computed from the hidden vectors which are mapped from the training examples. The likelihood of the hidden vector is computed by product of ones of individual attributes for each category; it is classified into the category whose likelihood is maximal. The difference from the traditional version of Naïve Bayes is to compute the likelihoods of elements in the hidden vector.

Let us make some remarks on the stacked Naïve Bayes as a deep version. The stacked version is composed with multiple unsupervised versions and a single supervised version. The unsupervised versions as many as the dimension of hidden vector are involved in computing the hidden vector. The hidden vector is classified instead of the input vector by the supervised Naïve Bayes. We may consider replacing the Naïve Bayes by the Bayesian learning which defines the Bayesian networks for implementing more advanced deep learning algorithm.

## 6.5 Summary and Further Discussions

Let us summarize what is studied in this chapter. The typical machine learning algorithms which belong to the probabilistic learning are the Bayes classifier, the Naïve Bayes, and the Bayesian learning. The modified versions of the Bayes classifier and the Naïve Bayes by adding the input encoding and the output

decoding are the kernel-based Bayes classifier, the pooled Bayes classifier, the output decoded Naive Bayes, and the pooled Naive Bayes. The k means algorithm is the unsupervised version of the Bayes classifier, and the Naive Bayes and the Bayes classifier are stacked by combining their supervised version and their unsupervised version with each other. This section is intended to discuss further what is studied in this chapter.

We may consider the Fuzzy Naive Bayes in computing the likelihood of an individual attribute value to a category. In the crisp Naive Bayes, the number of training examples where their attribute values match exactly is counted for computing the likelihood. In this version, we define the fuzzy distribution for each attribute value and find the probability of an attribute value is found in each training example. In the crisp Naive Bayes, if the value does not match, the probability becomes zero, whereas in the fuzzy Naive Bayes, even if the value does not match, the probability is not zero. Zero probability of an attribute value is less frequent as results from using the fuzzy Naive Bayes.

Let us consider the kernel-based Naive Bayes as a variant. The kernel-based learning is viewed as kind of deep learning in that the input vector is mapped into one in another space which becomes potentially a hidden vector. The kernel function of two vectors is defined, and its output value to a training example and a novice vector is computed. The training examples whose kernel function values are greater than or equal to the threshold are counted for computing the likelihoods. The kernel-based learning is viewed as the implicit deep learning which is distinguished from the explicit deep learning where hidden values are computed explicitly.

Let us mention attaching the convolution layer to the Naive Bayes, called convolutional Naive Bayes. The convolution is the operation on vector for mapping a vector into another vector by filtering values, and a single filter vector or multiple filter vectors should be defined for performing the operation, in advance. In the process of classifying a data item by the convolutional Naive Bayes, the input vector is mapped into another vector by the convolution, and the likelihoods of the mapped vector to the categories are computed. If multiple filter vectors are defined, multiple mapped vectors are generated, and their likelihoods are averaged. The convolution is previously used for filtering a matrix which represents an image with a filter matrix.

Let us consider attaching the textual convolution to the Naive Bayes for implementing its deep version. The textual convolution is the operation on a raw text with viewing it as a list of words. A filter is defined as a list of words, called filter words, and most similar one among them is selected for each word in the raw text. Multiple texts are generated from a single raw text in using multiple lists of filter words. The textual convolution may be attached to the Naive Bayes, if it is applied to text classification.

## Reference

1. T. Jo, Machine Learning Foundation, Springer, 2021.

# Chapter 7

## Deep Decision Tree



This chapter is concerned with the modification of the decision tree and its variants into their deep versions. The decision tree which is a typical symbolic machine learning algorithm where it is able to trace its classification into logical rules, classifies a data item by following branches which correspond to its attribute values. As the basic schemes, a decision tree is modified into its deep versions by attaching the input encoding and/or the output decoding. As the advanced schemes, a decision tree is modified into its unsupervised version, and its original and modified versions are combined with each other into the stacked version. This chapter is intended to describe the schemes of modifying the decision tree into its deep versions.

This chapter is organized into the five sections, and in Sect. 7.1, we describe the deep decision tree conceptually. In Sect. 7.2, we review the swallow version of the decision tree. In Sect. 7.3, we describe the deep version of the decision tree and the random forest which the input encoding and the output decoding are added. In Sect. 7.4, we describe the advanced deep version of them which are stacked by their unsupervised version. In Sect. 7.5, we summarize the entire contents of this chapter and discuss further on what is studied in this chapter.

### 7.1 Introduction

This section is concerned with the overview of the deep decision tree. It is constructed by the training examples and classifies each novice item from its root node to its terminal node, which is given as a label, following the edge indicating an attribute value. The decision tree and the random forest are modified into the deep version by adding the input encoding and the output decoding. The decision tree is modified into its unsupervised version, and the deep version is constructed by combining its unsupervised version with its original version. This section is intended to describe the deep version of decision tree for providing the overview.

There are some assumptions in using the decision tree for classifying data items. It is assumed that there is no noise in the training examples; all of attribute values in the training examples are clean. The process of classifying data items is expressed as logical rules by tracing it from a terminal node to the root node. All training examples are trusted as reliable ones; the fact that some training examples are noisy or not reliable is overlooked in using the decision tree. However, the reason of preferring the decision tree to the neural networks which are very tolerable to noisy training examples is its transparency which can provide the logical rules as the evidence.

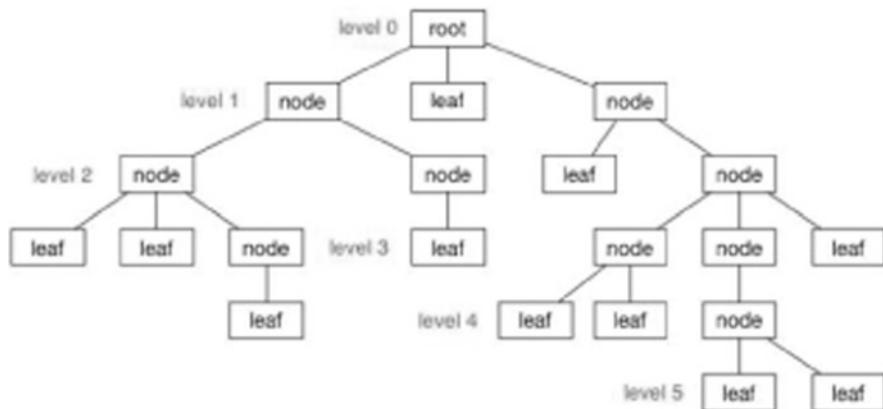
Let us mention the structure of decision tree which is used for the data classification. The root node and the nonterminal nodes are given as the attributes. A branch from a node to its child node corresponds to an attribute value. A set of training examples whose attribute value corresponds to the branch is covered in the child node. The terminal nodes in the decision tree are given as categories.

Let us mention the process of classifying a data item by the decision tree. In the root node, the attribute which corresponds to the root node is found, and the branch which corresponds to the attribute value is selected. If the current node is a nonterminal node, the attribute which corresponds to the current node in the decision tree is found, and the branch which corresponds to the attribute value is followed. If the current node is a terminal node, its label is returned as the classified one. In the decision tree, the classification proceeds from the root node to a terminal node, and symbolic classification rules are extracted as the evidence from a terminal node to the root node.

Let us mention what is intended in this chapter. We review the decision tree and the random forest as the swallow learning algorithms. We modify them into their deep versions by attaching the input encoding and the output encoding. We modify them into their unsupervised versions and construct the stacked versions by combining their supervised versions and their unsupervised versions as the more advanced deep learning algorithms. This section is intended to understand the process of modifying the decision tree and the random forest into their deep versions.

## 7.2 Swallow Version

This section is concerned with the swallow version of the decision tree before modifying it into its deep version. In Sect. 7.2.1, we present the graphical view of the decision tree. In Sect. 7.2.2, we mention the process of classifying a data item with the decision tree. In Sect. 7.2.3, we mention the process of selecting the root node as the start point of learning process. In Sect. 7.2.4, we mention the process of constructing the decision tree from the training examples as its learning process.



**Fig. 7.1** Tree structure from <https://usinuniverse.bitbucket.io/blog/treedatastructure.html>

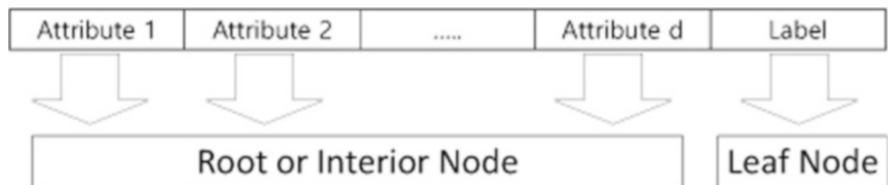
### 7.2.1 Graphical View

This section is concerned with the graphical view of the decision tree which is used for classifying a data item. The tree is the data structure which is given as a hierarchical form of nodes. In the decision tree, its nonterminal nodes are given as the attributes, the edges are given as the attribute values, and the terminal nodes are given as the categories. The decision tree is constructed from the training examples as its learning process. This section is intended to describe the decision tree with its graphical view.

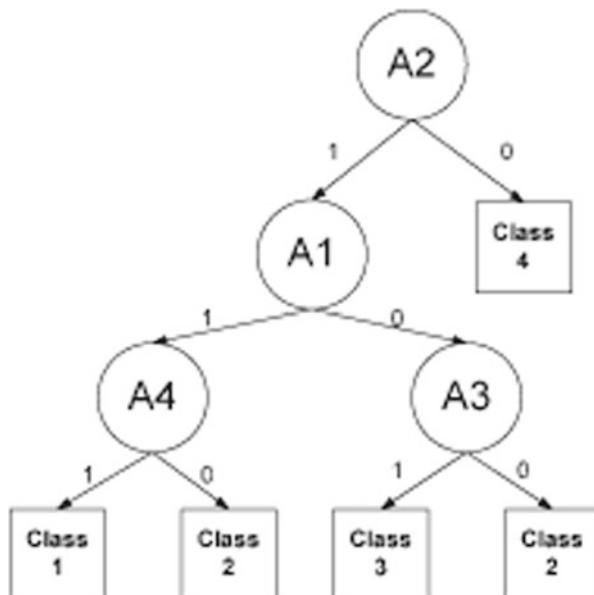
The relation of a numerical vector which represents an example with the decision tree is illustrated in Fig. 7.1. It is assumed that all the training examples are labeled, and the given task is the data classification. The non-leaf nodes in the decision tree indicate attributes or features. The leaf nodes indicate the categories. Both the learning process and the classification process of the decision tree begins with the root node.

The relation of a numerical vector which represents an example with the decision tree is illustrated in Fig. 7.2. It is assumed that all the training examples are labeled, and the given task is the data classification. The non-leaf nodes in the decision tree indicate attributes or features. The leaf nodes indicate the categories. Both the learning process and the classification process of the decision tree begin with the root node.

The structure of decision tree which is used for classifying a data item is illustrated in Fig. 7.3. The non-leaf nodes which are labeled with A1, A2, A3, and A4 are the attributes. The leaf nodes which are labeled with class 1, class 2, and class 3 are the predefined categories. The values of the four attributes are binary values; each edge indicates 0 or 1 as an attribute value. As the start of classifying an item by the decision tree, we need to check the value of the attribute, A2, as the label of the root node.



**Fig. 7.2** Nodes of decision tree



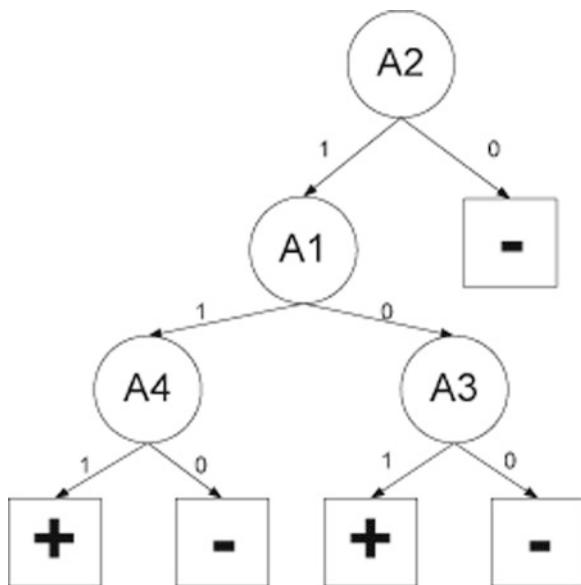
**Fig. 7.3** Example of decision tree

Let us make some remarks on the graphical view of the decision tree which is used for classifying data items. The tree is a nonlinear data structure which starts a single node, called root node, and is expanded into multiple ways. In the decision tree, a non-leaf node indicates an attribute, and a leaf node indicates a category. A data item is classified by the decision tree from its root node to its leaf node. The symbolic rules are extracted for providing evidence from a leaf node to the root node.

### 7.2.2 Classification Process

This section is concerned with the process of classifying a data item by the decision tree. In the previous section, we studied the graphical view of the decision tree with

**Fig. 7.4** Decision tree for binary classification



respect to the indications of leaf nodes, non-leaf nodes, and branches. In the decision tree, a data item is classified from the root node to a leaf node. The classification path is the list of branches which correspond to attribute values. This section is intended to describe the decision tree with respect to its classification process.

An example of the decision tree is presented for explaining the classification process in Fig. 7.4. The root node corresponds to the attribute A2 whose value is 0 or 1. In the classification process, if  $A2 = 1$ , the value of the attribute A1 is checked, and if  $A2 = 0$ , the item is classified into the negative class. If  $A1 = 1$ , the value of the attribute A3 is checked, and if  $A1 = 0$ , the values of the attribute A4 is checked. It reaches the terminal node which indicates the positive class or the negative class; the branches which indicate the attribute value are followed.

Let us explain the process of classifying a data item by the decision tree, the following one which is illustrated in Fig. 7.4. The binary input vector, [1 1 0 1] where  $A1 = 1$ ,  $A1 = 1$ ,  $A2 = 1$ ,  $A3 = 0$ , and  $A4 = 1$ , is given as the classification target. The root node is indicated by A2, and the left branch is selected to A1, because  $A2 = 1$ . Because  $A1 = 1$  and  $A4 = 1$ , the left child node of A1, A4 is selected, and the item is classified into the positive class. The path of classifying the item, [1 1 0 1], becomes  $A2 \rightarrow A1 \rightarrow A4 \rightarrow +$ .

The process of classifying a data item by the decision tree is implemented as a pseudo code in Fig. 7.5. In the function, `classifyByDecisionTree`, the root node is taken from the decision tree, and the function, `classifyByNode`, is called with the argument, the root node. In the function, `classifyByNode`, if the current node is a leaf node which indicates a category, it returns the label of the current node as the category. Otherwise, a list of branches is taken from the current node, it searches for the branch which matches the attribute value, and the function, `classifyByNode`, is

```

Category classifyByDecisionTree(item dataItem){
    Node rootNode = this.getRootNode();
    return this.classifyByNode(rootNode, dataItem);
}

Category classifyByNode(Node currentNode, item dataItem){
    if(currentNode.isTerminal()){
        Category cc = new Category[currentNode.getLabel()];
        return cc;
    }

    Feature ff = new Feature[currentNode.getLabel()]);
    List branchList = currentNode.getBranchList();
    int branchSize = branchList.size();
    for(int i = 0; i < branchSize;i++){
        Branch bb = branchList.getElement(i);
        if(bb.isMatch(dataItem)){
            Node childNode = currentNode.getChildNode(branch);
            return classifyByNode(childNode, dataItem);
        }
    }
    return "Not Classified";
}

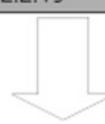
```

**Fig. 7.5** Classification algorithm by decision tree

called by replacing the current node by its child node, recursively. The classification process in the decision tree is implemented by calling the function, `classifyByNode`, recursively.

Let us make some remarks on the classification process in the decision tree which is covered in this section. The decision tree is interpreted into its leaf nodes as the categories, its non-leaf nodes as the attributes, and its branches as the attribute values. A data item is classified by the decision tree, from the root node to a leaf node, following the branches which correspond to its attribute values. The classification by the decision tree is implemented by calling recursively the function. The symbolic rules are extracted from the decision tree for tracing the classification process from a leaf node to the root node.

	Attribute 1	Attribute 2	Attribute 3	Attribute 4
Value 1	6:6:6:7	10:10:10:10	20:1:2:2	10: 5:5:5
Value 2	7:6:6:6	5:5:5:5	1:22:1:1	5:10:5:5
Value 3	6:7:6:6	5:5:5:5	1:1:23:0	5:5:10:5
Value 4	5:7:7:6	5:5:5:5	2:2:2:19	5:5:5:10



Root Node

**Fig. 7.6** Unbalanced for attribute selection

### 7.2.3 Root Node Selection

This section is concerned with the process of selecting the root node in the decision tree. In the previous section, we studied and implemented the classification process of the decision tree. In this section, the root node as the start of classifying a data item is decided from the training examples. The process of deciding the root node is applied to its descendants recursively as the learning process of the decision tree. This section is intended to describe the process of decoding the root node.

The distribution over attribute values is illustrated for providing the intuition of selecting an attribute as the root node in Fig. 7.6. In the given classification task, it is assumed that there are the four attributes, the four discrete values in each attribute, and the four categories. Each column is an attribute, each row is an attribute value, and each cell is the number of training examples with the attribute value in the four categories. Attribute 3 among the four attributes has the unbalanced distribution over the training examples in the four categories; attribute 3 is selected as the root node. The metric for measuring the attribute unbalance is needed for building the decision tree.

The process of selecting the root node from the attributes is illustrated as a pseudo code in Fig. 7.7. The attributes and the categories are notated respectively by  $A_1, A_2, \dots, A_d$  and  $c_1, c_2, \dots, c_M$ , and when each attribute is assumed to be discrete, it is viewed as a set of values,  $A_i = \{v_{i1}, v_{i2}, \dots, v_{i|A_i|\}$ .  $\#((A_i = v_{ik}) \wedge c_j)$  is the number of training examples with  $A_i = v_{ik}$  within the category,  $c_j$ ,  $\#(A_i = v_{ik})$  is the number of training examples with  $A_i = v_{ik}$ ,  $\#(c_j)$  is the number of training examples within the category,  $c_j$ ,  $p_{((A_i=v_{ik}) \wedge c_j)}$  is the portion of the category,  $c_j$  in the attribute value,  $A_i = v_{ik}$ , as expressed in Eq. (7.1):

$$p_{((A_i=v_{ik}) \wedge c_j)} = \frac{\#((A_i = v_{ik}) \wedge c_j)}{\#(A_i = v_{ik})}, \quad (7.1)$$

```

Node selectRootNode(List attributeList, List dataItemList, List categoryList){
    int dimension = attributeList.size();
    int categorySize = categoryList.size();
    double maxTotalPortion = 0.0;
    int rootIndex = 0;

    for(int i = 0; i < dimension; i++){
        Attribute aa = attributeList.getElement(i);
        List valueList = aa.getValueList(dataItemList);
        int valueSize = valueList.size();
        double totalPortion = 0.0;
        for(int j = 0; j < valueSize; j++){
            Value vv = valueList.getElement(j);
            double maxPortion = 0.0;
            for(int k = 0; k < categorySize; k++){
                Category cc = categoryList.getElement(k);
                double portion = cc.getPortion(dataItemList, vv);
                if(portion > maxPortion)
                    maxPortion = portion;
            }
            totalPortion = totalPortion + maxPortion;
        }
        if(totalPortion > maxTotalPortion){
            maxTotalPortion = totalPortion;
            rootIndex = i;
        }
    }
    Node rootNode = new Node(attributeList.getElement(rootIndex));
    return rootNode;
}

```

**Fig. 7.7** Root node selection by unbalanced index

$P((A_i=v_{ik}), c_{max})$  is the maximum portion of the category in the attribute value,  $A_i = v_{ik}$ , as expressed in Eq. (7.2):

$$P((A_i=v_{ik}), c_{max}) = \max_{j=1}^M \frac{\#((A_i = v_{ik}) \wedge c_j)}{\#(A_i = v_{ik})}, \quad (7.2)$$

the unbalanced index,  $p_{A_i}$ , is expressed as Eq. (7.3):

$$p_{A_i} = \sum_{k=1}^{|A_i|} P((A_i=v_{ik}), c_{max}). \quad (7.3)$$

For each attribute, its unbalanced index is computed by Eq. (7.3), and the attribute with its maximal index is selected as the root node, as shown in Eq. (7.4),

$$A_{\max} = \operatorname{argmax}_{i=1}^d p_{A_i}. \quad (7.4)$$

The root node is linked to its child node by calling this algorithm recursively with the attribute list except one which is selected as the root node and  $Tr_{A_i=v_k}$ , for each attribute value.

The unbalanced index may be replaced by the information gain in selecting an attribute as the current node identifier. Equation (7.2) is the portion of the category,  $c_j$ , in the attribute value,  $A_i = v_{ik}$ . The entropy is computed by Eq. (7.5), for each attribute:

$$E_{(A_i=v_{ik})} = - \sum_{j=1}^M p_{((A_i=v_{ik}) \wedge c_j)} \log_2 p_{((A_i=v_{ik}) \wedge c_j)} \quad (7.5)$$

The information gain is computed for each attribute by Eq. (7.6):

$$IG_{A_i} = \sum_{k=1}^{|A_i|} E_{(A_i=v_{ik})} \quad (7.6)$$

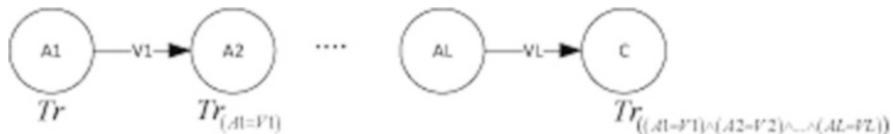
The attribute with the maximum information gain is selected as the node identifier.

Let us make some remarks on the process of selecting an attribute as the root node in the decision tree. By the intuition, the attribute with its maximal unbalance over the categories is selected. The dominance of a category is computed for each attribute for finding the unbalance index of the attribute. The unbalance index is replaced by the information gain. After deciding an attribute as the root node, a subset of training examples is allocated for selecting a child node for each attribute value.

#### 7.2.4 Learning Process

This section is concerned with the process of constructing the decision tree from the training examples. In the previous section, we studied the process of selecting an attribute as the root node. After deciding the root node, for each branch, the training examples correspond to it, and the child node is decided by the process of deciding the root node. If the remaining examples are labeled identically, the child node becomes a leaf node which indicates a category. This section is intended to describe the process of building the decision tree using the training examples.

The process of updating the training set following the path from the root node to a leaf node is illustrated in Fig. 7.8. In the decision tree, a non-leaf node is an attribute, and a branch is an attribute value. The entire set of the training examples are involved in the root node, A1, and the subset of training examples which satisfy



**Fig. 7.8** Training set update from root node to terminal node

the condition  $A_1 = V_1$  is involved in its child node,  $A_2$ . The training examples with  $(A_1 = V_1) \wedge (A_2 = V_2) \wedge \dots \wedge (A_L = V_L)$  remain in the leaf node,  $C$ . The set of training examples is updated from the root node to a leaf node.

The algorithm of selecting a node is called recursively with the arguments the attribute list and the example list, as shown in Fig. 7.9. If there is no attribute and no training example, it is terminated with nothing, and if the examples in the list are labeled identically with a label, it is terminated by creating the node which is identified with a label as a leaf node. The attribute is selected from the attribute list by the process of selecting the root node as a node identifier. For each value in the selected attribute, the attribute list and the example list are updated by removing the selected one and taking only examples which correspond to it. The node selection algorithm is called recursively with the updated attribute list and the updated example list.

The process of defining branches after selecting an attribute as the current node is illustrated in Fig. 7.10. The number of values is assumed to be finite in the selected attribute; it is required to discretize the attribute values for applying the decision tree. The attribute values in selecting the attribute,  $a_i$  as the current node as follows:  $a_i \leftarrow v_{i,1}, v_{i,2}, \dots, v_{i,|a_i|}$ , where  $|a_i|$  is the number of values of the attribute,  $a_i$ . The  $|a_i|$  branches are defined; each branch corresponds to an attribute value,  $v_{i,k}$ . The training set is partitioned into subsets, attribute value by attribute value in moving to the next level.

Let us make some remarks on the process of constructing the decision tree from the training examples. The training set is updated into the subsets which satisfy the attribute values from the top to the bottom. The unbalance index or the information gain is computed for each attribute in the current attribute set for selecting an attribute as the current node, and the attribute set is updated with the set excluding the selected one. The branch is constructed from the node value by value after selecting the attribute. The process of constructing the decision tree from the training examples is implemented in the recursive style.

### 7.3 Basic Deep Versions

This section is concerned with the deep version of decision tree, including the random forest. In Sect. 7.3.1, we describe the random forest which is viewed as an advanced version of decision tree. In Sect. 7.3.2, we mention the improved version

```

Node selectNode(Node currentNode, int branchIndex, List attributeList, List dataItemList, List categoryList ){
    int attributeSize = attributeList.size();
    int categorySize = categoryList.size();
    double maxTotalPortion = 0.0;
    int nodeIndex = 0;

    if(attributeList.isEmpty() || dataItemList.isEmpty())
        return;

    if(dataItemList.isIdenticallyLabeled()){
        Node node = new Node(dataItemList.getLabel());
        currentNode.setChildNode(branchIndex, node);
        return;
    }

    for(int i = 0;i < attributeSize;i++){
        Attribute aa = attributeList.getElement(i);
        List valueList = aa.getValueList(dataItemList);
        int valueSize = valueList.size();
        double totalPortion = 0.0;
        for(int j = 0;j < valueSize;j++){
            Value vv = valueList.getElement(j);
            double maxPortion = 0.0;
            for(int k = 0; k < categorySize;k++){
                Category cc = categoryList.getElement(k);
                double portion = cc.getPortion(dataItemList, vv);
                if(portion > maxPortion)
                    maxPortion = portion;
            }
            totalPortion = totalPortion + maxPortion;
        }
        if(totalPortion > maxTotalPortion){
            maxTotalPortion = totalPortion;
            nodeIndex = i;
        }
    }
    Attribute selectedAttribute = attributeList.getElement(nodeIndex);
    List selectedAttributeValueList = selectedAttribute.getValueList(dataItemList);
    int selectedAttributeValueSize = selectedAttributeValueList.size();
    Node node = new Node(selectedAttribute);
    attributeList.deleteElement(nodeIndex);
    currentNode.setChildNode(branchIndex, node);
    for(int i = 0;i < selectedAttributeValueSize;i++){
        Value attributeValue = selectedAttributeValueList.getElement(i);
        List updatedDataItemList = dataItemList.getElementList(attributeValue);
        selectNode(currentNode.getChildNode(branchIndex), i, attributeList, updatedDataItemList, categoryList);
    }
}

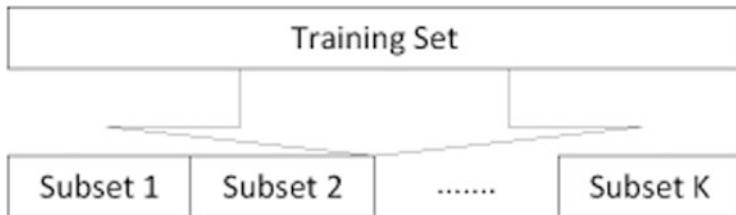
```

**Fig. 7.9** Node selection in recursive form

of random forest where the training set is partitioned by clustering the training examples. In Sect. 7.3.3, we mention the version of decision tree which is added by the output decoding. In Sect. 7.3.4, we add the pooling layer to the decision tree for modifying it into the deep version.



**Fig. 7.10** Node structure in decision tree



**Fig. 7.11** Training set partition for random forest

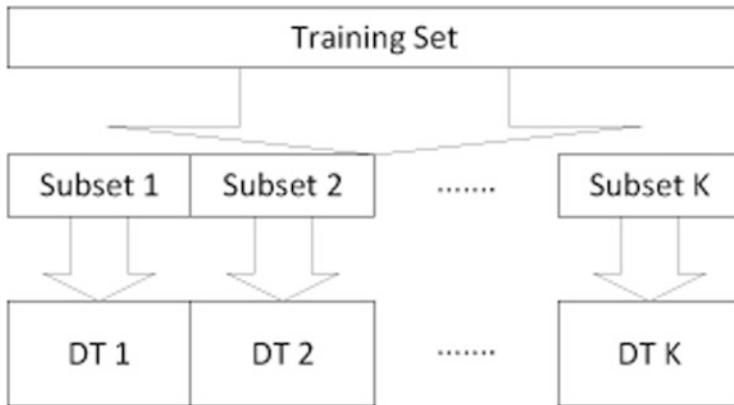
### 7.3.1 Random Forest

This section is concerned with the random forest as a group of decision trees. The training set is partitioned into subsets as the preparation for implementing the random forest. Each decision tree is trained with its own subset. Each data item is classified by applying an ensemble learning scheme to the decision trees. This section is intended to describe the random forest which is given as a group of decision trees.

The partition of the training set into subsets is illustrated in Fig. 7.11. We already studied the partition of the training set in detail in Sect. 4.2. The random partition is adopted as the scheme in the initial version of random forest. The subsets of the training examples are denoted by  $Tr_1, Tr_2, \dots, Tr_K$ . Each subset,  $Tr_i$  is allocated for constructing the decision tree,  $DT_i$ .

The construction of multiple decision trees from the subsets of training examples is illustrated in Fig. 7.12. The training set,  $Tr$ , is partitioned into  $K$  subsets,  $Tr_1, Tr_2, \dots, Tr_K$ . The decision tree,  $DT_i$ , is constructed from the training examples in the subset,  $Tr_i$ . The  $K$  decision trees,  $DT_1, DT_2, \dots, DT_K$ , are constructed as the random forest; finally, the random forest is composed with the  $K$  decision trees. The choice of the decision tree or the random forest is like the choice of a single decision tree with its big size or multiple decision trees with their small sizes.

Let us review the schemes which are applied to the combination of multiple decision trees in the random forest. The voting is the scheme of deciding the final answer by collecting answers from the multiple decision trees. The expert gate is the scheme of deciding the final answer by getting the answer from the nominated



**Fig. 7.12** Construction of independent decision trees from subsets

decision tree. The cascading is the scheme of deciding the final answer by the current decision tree where its answer is judged to be certain. We already studied the three schemes in detail in Chap. 4.

Let us make some remarks on the random forest as a combination of multiple decision trees. The training set is partitioned into subsets for implementing the random forest. A decision tree is constructed from each subset, and multiple decision trees are results from learning the training examples. In the process of classifying a data item, it is classified by the decision trees, and the final answer is decided by applying the voting. Let us consider various versions of random forest depending on the partition scheme and the combination scheme.

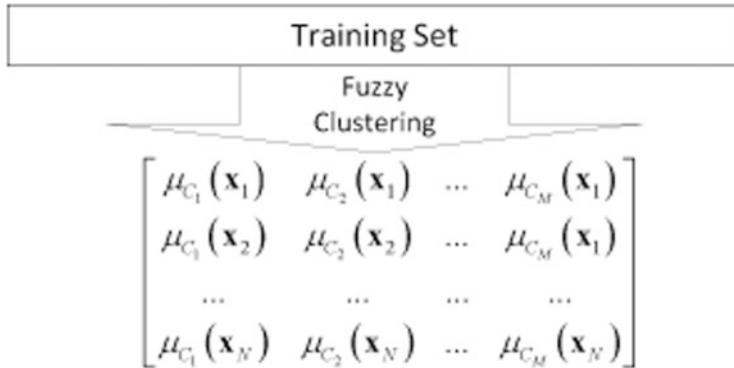
### 7.3.2 Clustering-Based Multiple Decision Trees

This section is concerned with the random forest which is trained with the clusters of training examples. In the previous section, we study the initial version of the random forest where the training set is partitioned into subsets by selecting the training examples at random. In this section, we study the upgraded version of the random forest where the training set is partitioned into subsets by clustering the training examples, depending on their similarities. An unsupervised learning algorithm, such as the k means algorithm, is involved for clustering the training examples. This section is intended to describe the advanced version of the random forest where the data clustering is added.

The partition of the training set into subsets by clustering the training examples is illustrated in Fig. 7.13. The data clustering is the process of segmenting a group of data items into subgroups, each of which is called cluster and contains similar data items. In this scheme of partitioning the training set, the training examples are



**Fig. 7.13** Training set clustering

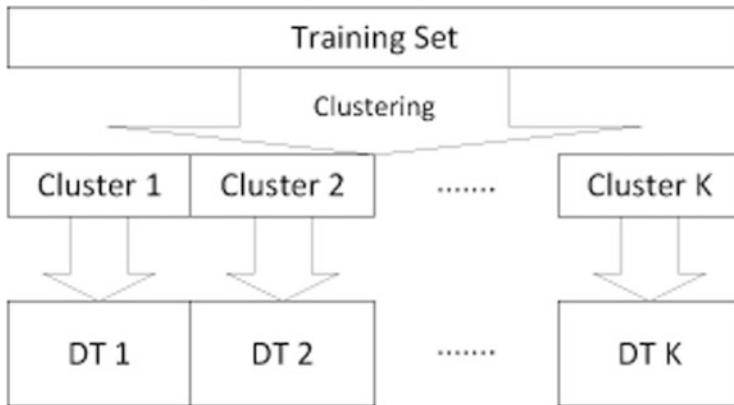


**Fig. 7.14** Fuzzy train set clustering

clustered into subgroups, each of which has similar training examples, and each subgroup becomes a subset. A clustering algorithm should be adopted separately from the supervised learning algorithm, and the training examples are clustered depending on their similarities regardless of their target labels. Each subset consists of similar training examples as the results from partitioning the training set.

The process of generating the item-cluster membership matrix by the fuzzy clustering is illustrated in Fig. 7.14. The fuzzy clustering is the process of assigning cluster memberships to each data item. The fuzzy clustering of the training examples is the process of assigning cluster membership values to each training example, and item-cluster membership matrix on the training examples is the result from doing so. The set of the entire training examples are allocated to each decision tree, and it learns the training examples with their discriminations. If the fuzzy clustering on the training examples is adopted, the decision tree may be bigger.

The construction of a decision tree with a cluster of training examples is illustrated in Fig. 7.15. The training examples are clustered by a clustering algorithm regardless of their labels. The  $K$  clusters, each of which contains similar training examples, are results from doing that, and each decision tree is constructed by ones in each cluster. Each decision tree corresponds to its own cluster; this version of random forest consists of the  $K$  decision trees which correspond to the clusters.



**Fig. 7.15** Cluster-based random forests

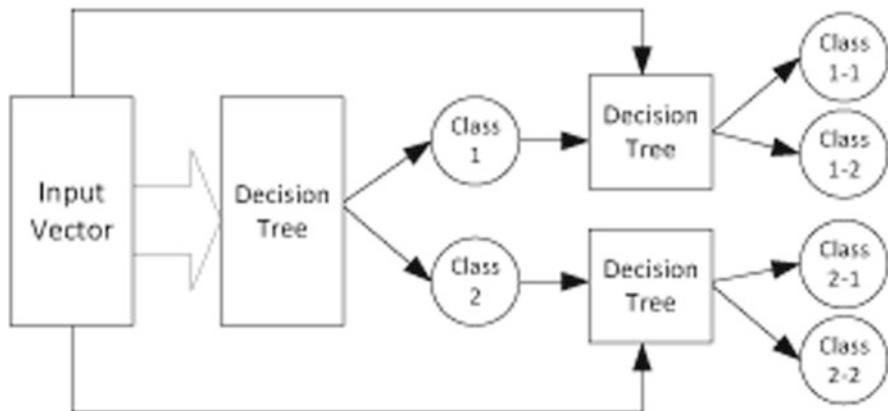
The expert gate is recommended rather than the voting for deciding the final answer from the  $K$  decision trees.

Let us make some remarks on the random forest which is built by clustering the training examples. They are clustered by a clustering algorithm regardless of their target labels, depending on their similarities. We consider the fuzzy clustering of the training examples which generates the item-cluster membership matrix, instead of clusters. A decision tree is constructed from each cluster of training examples, in this version of random forest. We consider partitioning the attribute set into subsets for building the random forest.

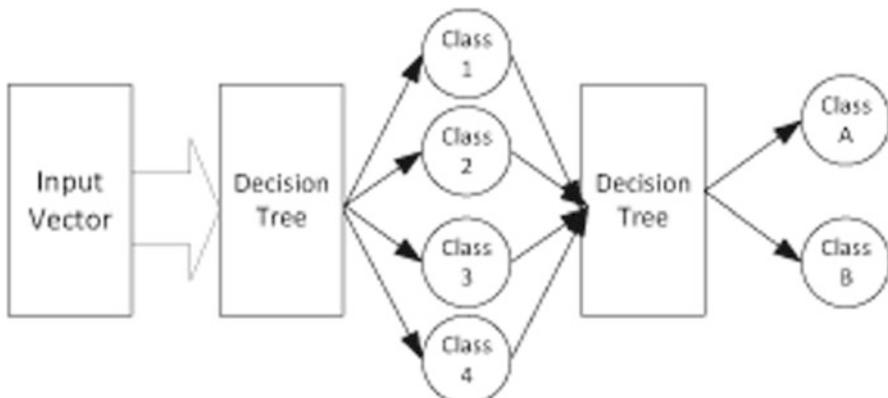
### 7.3.3 *Output-Decoded Decision Tree*

This section is concerned with the deep version of decision tree, called output-decoded decision tree. In the previous section, we studied the random forest as the ensemble learning which is applied to decision trees. In the current section and the next section, we modify the decision tree into its deep versions. The output decoding which is attached for modifying a machine learning algorithm into its deep version is regarded as the process of computing the output vector from a hidden vector. This section is intended to describe the output-decoded decision tree as a deep learning algorithm.

The application of the decision tree to the implementation of the category specification is illustrated in Fig. 7.16. In the general level, there are two categories, class 1 and class 2, and two specific categories are allocated to each abstract category. In the left, the decision tree classifies the input vector into class 1 or class 2. The top right decision tree receives the input vector which is classified into class 1 and classifies it into class 1–1 or class 1–2, and the bottom right decision tree



**Fig. 7.16** Category specification layer in decision tree

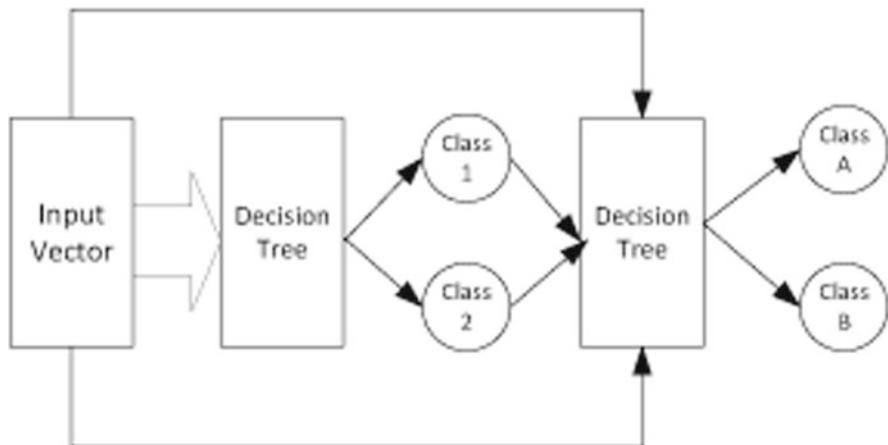


**Fig. 7.17** Category abstraction layer in decision tree

receives the input vector which is classified into class 2 and classifies it into class 2–1 or class 2–2. This type of deep learning is viewed as implementing the hierarchical classification with the decision tree.

The modification of the decision tree into its deep version by the category abstraction is illustrated in Fig. 7.17. It is assumed that the process of classifying an item by the left decision tree belongs to the fuzzy classification. The four categorical scores are estimated by the fuzzy decision tree; the four-dimensional hidden vector is computed from the input vector. The four-dimensional vector is classified into one of the two categories by the right decision tree. It is required to modify the decision tree into its fuzzy version for implementing this deep learning algorithm.

The implementation of the deep decision tree using the category optimization is illustrated in Fig. 7.18. The intermediate classes, class 1 and class 2, are defined, and the input vector is classified into one of the two classes by the left decision tree.



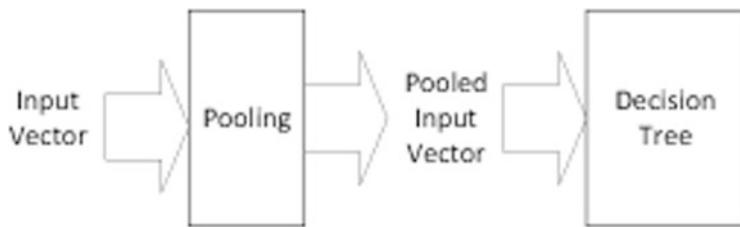
**Fig. 7.18** Category optimization layer in decision tree

The input which is augmented with its intermediate class is classified into class A or class B by the right decision tree. The left decision tree is viewed as an unsupervised decision tree; it is assumed that the correlation between the intermediate categories and the final categories is weak. The vector which indicates the categories scores of class 1 and class 2 is regarded as the additional elements to the input vector.

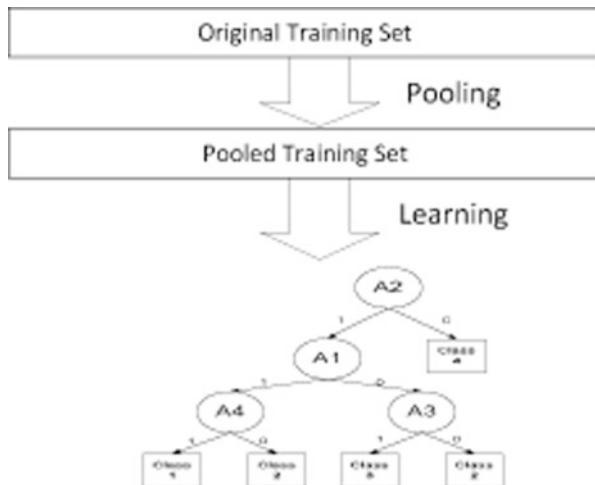
Let us make some remarks on the deep versions of the decision tree by attaching the output decoding. A hidden vector is computed by augmenting the input vector with its abstract class in adding the category specification. In adding the category specification, a hidden vector is computed by getting the categorical scores of the specific categories. In adding the categorical optimization, a hidden vector is computed by assigning categorical scores of intermediate categories to the input vector. The addition of the output decoding for modifying the machine learning algorithm into the deep version is viewed as the process of computing the output vector from a hidden vector.

### 7.3.4 Pooled Naive Bayes

This section is concerned with the deep version of the decision tree with the pooling layer. In the previous section, we modified the decision tree by attaching the output decoding, such as the category specification and the category abstraction. In this section, we modify the decision tree into the deep version by adding the pooling layer as the feature extraction. A more compact-sized decision tree is expected as the effect of reducing the dimension by the pooling layer. This section is intended to describe the deep version of the decision tree with the attachment of the pooling layer.



**Fig. 7.19** Pooling layer in decision tree



**Fig. 7.20** Learning in decision tree with pooling layer

The modification of the decision tree into its deep version by attaching the pooling layer is illustrated in Fig. 7.19. The input vector which indicates a data item is mapped into another vector through the pooling layer. If the dimension of the input vector is  $d$ , and the size of the window is  $s$ , the dimension of the mapped vector is  $d - s + 1$ . Mapping the input vector into another vector by the pooling is interpreted into the process of computing the hidden vector from the input vector. The hidden vector is classified directly in this version of decision tree.

The process of constructing the decision tree by the mapped training examples is illustrated in Fig. 7.20. The set of original training examples is denoted by  $Tr = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$ , and each training example which is a  $d$  dimensional vector is denoted by  $\mathbf{x}_i = [x_{i1} \ x_{i2} \ \dots \ x_{id}]$ . Each training example,  $\mathbf{x}_i$ , is mapped into a hidden vector,  $\mathbf{h}_i = [h_{i1} \ h_{i2} \ \dots \ h_{i(d-s+1)}]$  by the pooling with the sliding window with its  $s$  size. The set of hidden vectors which are mapped from the training examples by the pooling is denoted by  $Tr_h = \{\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_N\}$ . The decision tree is constructed from the set,  $Tr_h$ , instead of  $Tr$ , by the process which is described in Sect. 7.2.4.

Let us mention the process of classifying a data item by this version of decision tree. The decision tree is constructed from the set of vectors which are mapped from the training examples,  $Tr_h = \{\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_N\}$ . The novice vector,  $\mathbf{x}$ , is prepared and mapped into the hidden vector,  $\mathbf{h}$ , by the pooling layer. The mapped vector,  $\mathbf{h}$ , is classified by the process which is described in Sect. 7.2.2. The process of classifying the data item is identical to the swallow version of decision tree except applying the pooling.

Let us make some remarks on the deep version of decision tree. The pooling is the process of mapping the input vector into another vector with its lower dimensionality. The process of constructing the decision tree is to map the training examples into their own hidden vectors by the pooling layer and construct the decision tree by the hidden vectors. In this version of the decision tree, its non-leaf nodes are the attributes in the hidden vector, and its edges are elements of hidden vectors. We consider constructing the random forest by partitioning the set of hidden vectors.

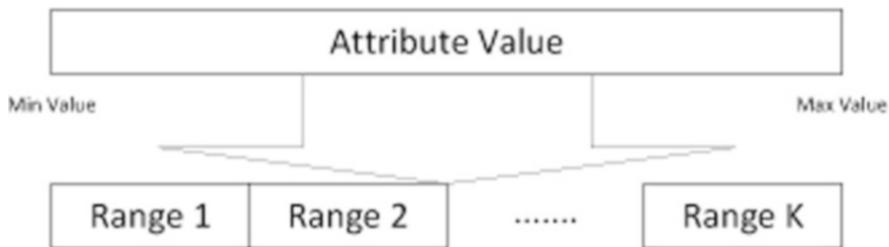
## 7.4 Advanced Deep Versions

This section is concerned with the advanced deep version of the decision tree and the random forest. In Sect. 7.4.1, we modify the decision tree into its unsupervised version. In Sect. 7.4.2, we mention the stacked decision tree into which the decision tree is modified. In Sect. 7.4.3, we modify the random forest into its unsupervised version. In Sect. 7.4.4, we add the unsupervised layer to the random forest, for modifying it into the deep version.

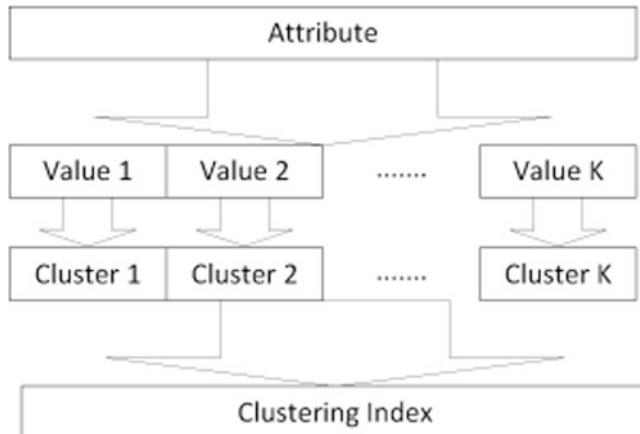
### 7.4.1 Unsupervised Decision Tree

This section is concerned with the unsupervised version of decision tree. It is initially designed as a supervised learning which treats labeled examples. In this section, we modify the decision tree into its unsupervised version which can treat unlabeled examples. In advance, the number clusters and the initial cluster prototypes should be decided. This section is intended to describe the unsupervised decision tree for clustering data items.

The discretization of a continuous attribute is illustrated in Fig. 7.21. In the branches of the decision tree which correspond to their own attribute values, each attribute is assumed to be discrete. In the process of discretizing a continuous attribute, the maximum value and the minimum value are selected, and the finite number of ranges is decided between them. A continuous attribute value is mapped into a discrete value which corresponds to the range within which its original value exists. An issue in discretizing continuous attributes is how to decide the number of ranges and the size of each range.



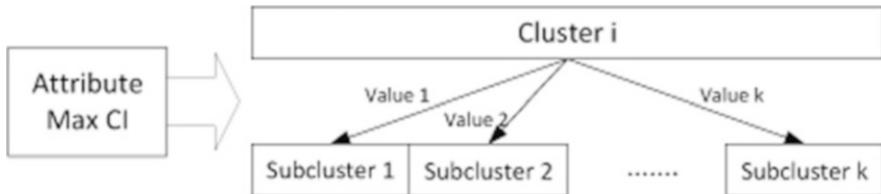
**Fig. 7.21** Attribute value discretization



**Fig. 7.22** Root node selection in unsupervised decision tree

The process of computing the cluster index from clusters for each attribute value is illustrated in Fig. 7.22. It is assumed that the number of values in each attribute is finite; continuous attributes are discretized by the above process. Unlabeled examples are clustered depending on their attribute values; each cluster is characterized by an attribute value. The clustering index which is explained in [1] is computed to the clusters; the attribute with its maximal clustering index is selected as the root node. The number of clusters is the number of values in the given attribute.

The non-leaf nodes and the leaf nodes in the unsupervised decision tree are illustrated in Fig. 7.23. The root node is selected by the above process, and the data items with the corresponding attribute value are given as a cluster. In the process of clustering the data items hierarchically, the attribute with its maximal clustering index is selected as a non-leaf node, and its child nodes become nested clusters. If there are too many clusters, some final clusters are merged into a cluster, depending on their similarities. The decision tree which is constructed by the unlabeled examples is used for classifying a novice data item into one among the clusters.



**Fig. 7.23** Hierarchical clustering in unsupervised decision tree

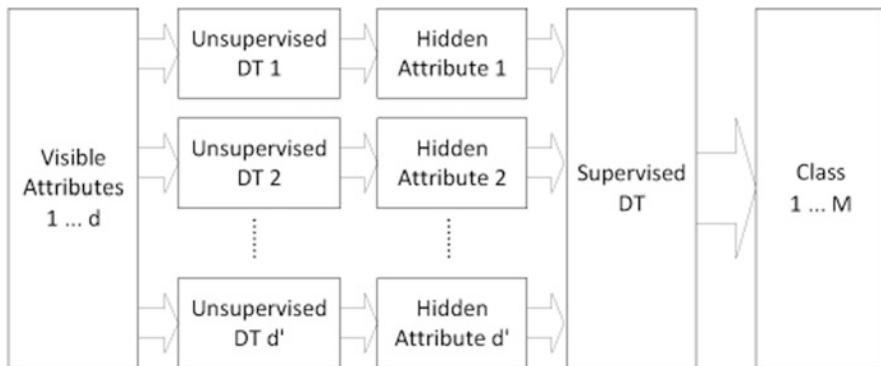
Let us make some remarks on the unsupervised version of decision tree. Each continuous attribute should be discretized by defining ranges. Data items are clustered by attribute values, clustering index is computed, and the attribute with maximal clustering index is selected as the root node. As the further clustering process, data items with the attribute value which correspond to a branch from the parent node are clustered by another attribute values, and subgroups which result from doing that are nested clusters. Different decision tree may be constructed from unlabeled examples, depending on the strategy of discretizing continuous attributes.

#### 7.4.2 Stacked Decision Tree

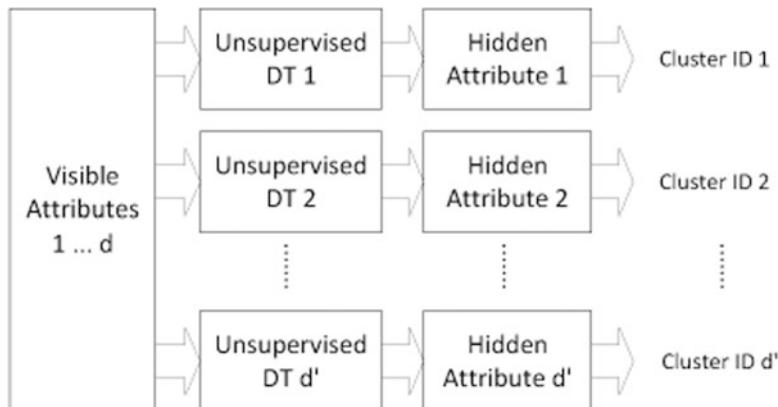
This section concerned with the stacked decision tree as the deep version of decision tree. In the previous section, we modified the decision tree into its unsupervised version which is applicable for clustering data items. In this section, the decision tree is modified into its deep version by combining its supervised version and its unsupervised version with each other. The role of the unsupervised version is to compute hidden values. This section is intended to describe the deep version of decision tree which consists of the supervised one and the unsupervised one.

The stacked decision tree as the deep version is illustrated in Fig. 7.24. It is assumed that the input vector is a  $d$  dimensional vector, and the hidden vector is a  $d'$  dimensional vector. An individual cluster identifier is an element of hidden vector; the  $d'$  unsupervised decision trees are involved in computing the hidden vector. The supervised decision tree in the right part is used for classifying the hidden vector into one among the  $M$  categories. The  $d'$  unsupervised decision trees are discriminated by the number of clusters and the initial cluster prototypes.

The process of computing a hidden value from the input values is illustrated in Fig. 7.25. The  $d'$  unsupervised decision trees are involved in computing the  $d'$  dimensional hidden vector. The  $d'$  unsupervised decision trees are discriminated with their different number of clusters and their different initial cluster prototypes. The identifier of the cluster into which the data item is arranged is an element of the hidden vector. The element is given as an integer value which is less than the number of clusters in case of starting from zero.



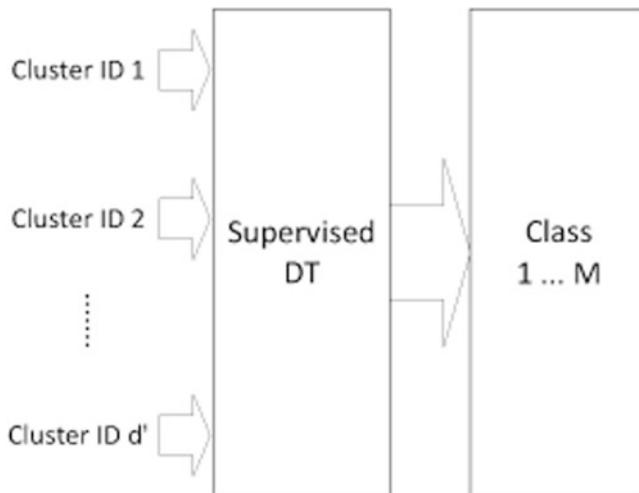
**Fig. 7.24** Architecture of deep decision tree with unsupervised layer



**Fig. 7.25** Hidden variable computation in deep decision tree

The process of computing the output value from the hidden values is illustrated in Fig. 7.26. The hidden values are computed by the  $d'$  unsupervised decision trees, and each hidden value is given as a cluster identifier. A single supervised decision tree is used for computing the final output values from the hidden values. The decision tree in the right position is constructed from the hidden vectors which are mapped from the input vector through multiple unsupervised decision trees. Because the output values are computed from the hidden values, this version of decision tree is viewed as a deep learning algorithm.

Let us make some remarks on the deep version of the decision tree as the combination of its supervised version and its unsupervised version. The multiple unsupervised decision trees and the single supervised one are involved in implementing the deep version. The hidden values are computed from the input values by getting its cluster identifiers, and each hidden value indicates a cluster identifier. The hidden vector which is computed by the multiple unsupervised decision trees is



**Fig. 7.26** Classification of hidden values in deep decision tree

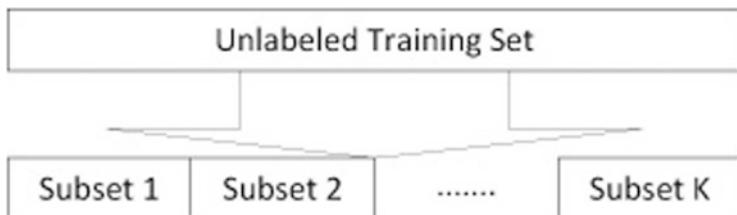
classified by a single supervised decision tree. The node in the decision tree which is constructed from the hidden vectors is an attribute of them.

### 7.4.3 Unsupervised Random Forest

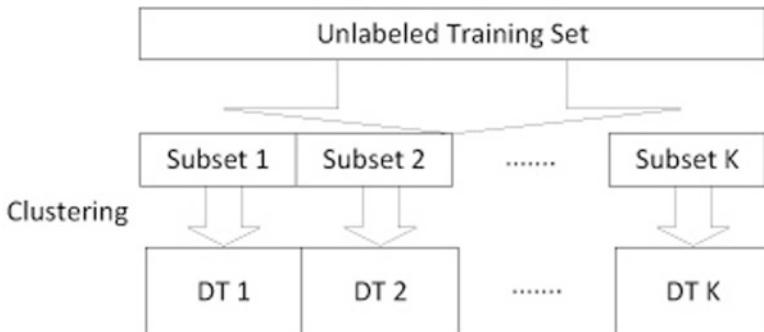
This section is concerned with the unsupervised version of the random forest. It is a group of decision trees, each of which is trained with its own subset of training examples. The set of data items is partitioned at random into subsets, and the data items in each subset are clustered by each unsupervised decision tree. The partition is assumed to be exclusive, and multiple cluster groups are integrated into a single cluster group by the meta-clustering. This section is intended to describe the unsupervised version of random forest as expansion from the unsupervised decision tree.

The random partition of the set of data items is illustrated in Fig. 7.27. The entire set of data items is used for clustering them by a single decision tree in Sect. 7.4.1. The number of unsupervised decision trees in the random forest is decided, and the set of data items is partitioned at random into the identically sized subsets as many as decision trees. An unsupervised decision tree is constructed by its own subset of data items; a group of smaller sized decision trees is the output from training the random forest. It is required to decide the number of clusters in advance for using the unsupervised random forest.

The set of unsupervised decision trees which are involved in clustering data items is illustrated in Fig. 7.28. The set of unlabeled examples is partitioned into subsets by the process which is mentioned above. Each subset of data items is allocated as



**Fig. 7.27** Partition of unlabeled training set in unsupervised random forest

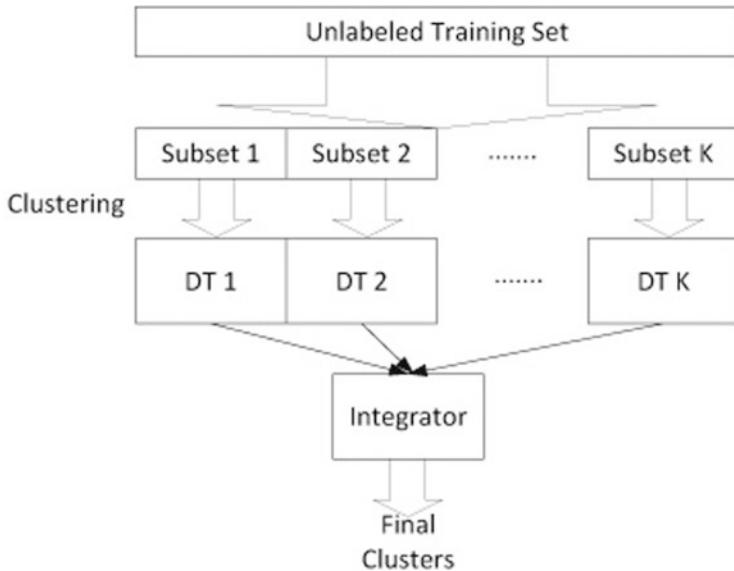


**Fig. 7.28** Training each unsupervised decision tree with its own subset

the clustering targets to each decision tree. Each decision tree is constructed by its own subsets, independently of others. The multiple cluster sets are collected from the decision trees which are constructed from their own subsets.

The process of clustering data items by the random forest is illustrated in Fig. 7.29. The set of unlabeled examples is partitioned into subsets, and each subset is allocated to each decision tree. The data items are clustered by each decision tree, and the cluster sets, each of which is the result from clustering data items in the subset by the unlabeled decision tree, are gathered. If the partition is assumed to be exclusive, the multiple cluster groups are integrated into a single cluster group by the meta-clustering. In the supervised random forest, the coordinator makes the final answer, whereas in the unsupervised random forest, the coordinator integrates multiple cluster groups.

Let us make some remarks on the unsupervised version of the random forest. The entire set of data items is partitioned into subsets. The data items in each subset are clustered by a decision tree; an unsupervised decision tree is constructed from the data items. Multiple cluster groups which are generated by multiple decision trees are integrated into a single cluster group by the meta-clustering. We consider the overlapping partition and scheme of integrating multiple overlapping clustering groups.



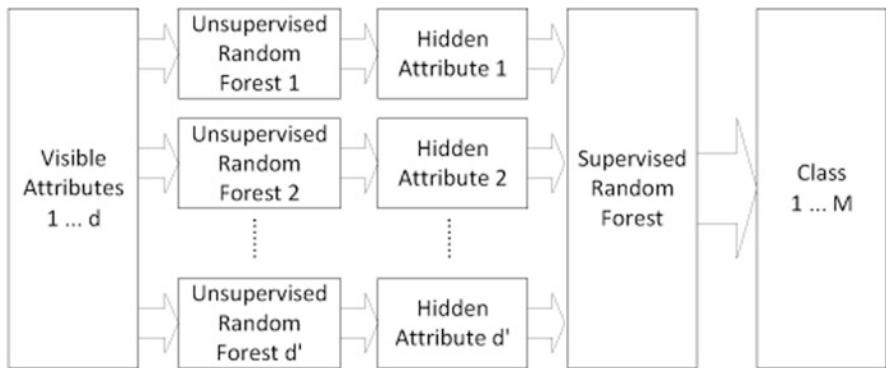
**Fig. 7.29** Integration of clustering results from decision trees in unsupervised random forest

#### 7.4.4 Stacked Random Forest

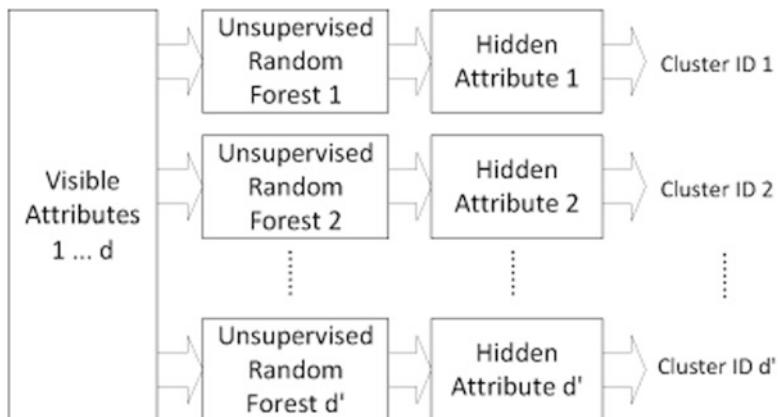
This section is concerned with the stacked random forest as the deep version. The random forest is viewed as a group of decision trees, each of which is trained with its own subset. A single decision tree is replaced by multiple decision trees as a random forest, in both the supervised layer and the unsupervised layer. Multiple unsupervised random forests are discriminated by different schemes of partitioning the training set and integrating the multiple clustering results. This section is intended to describe the deep learning algorithm which is called stacked random forest.

The deep version of random forest is illustrated in Fig. 7.30. The  $K \times d'$  decision trees are involved in the unsupervised layer if the random forest consist of  $K$  decision trees. The hidden vector which is computed in the unsupervised layer is classified into one among the  $M$  classes by the supervised random forest in the right part. The data item is arranged into only one cluster by an unsupervised random forest; this clustering type is assumed as the hard clustering. A massive number of decision trees are involved in implementing the deep learning algorithm; we need to consider implementing it by the parallel or distributed computing.

The unsupervised layer in the deep version of the random forest is illustrated in Fig. 7.31. The role of each unsupervised random forest is to compute an element in the hidden vector; the number of unsupervised random forest algorithms is the dimension of the hidden vector. Each element in the hidden vector is a cluster identifier into which the input vector is arranged. The clustering type is assumed



**Fig. 7.30** Architecture of deep random forest with unsupervised layer

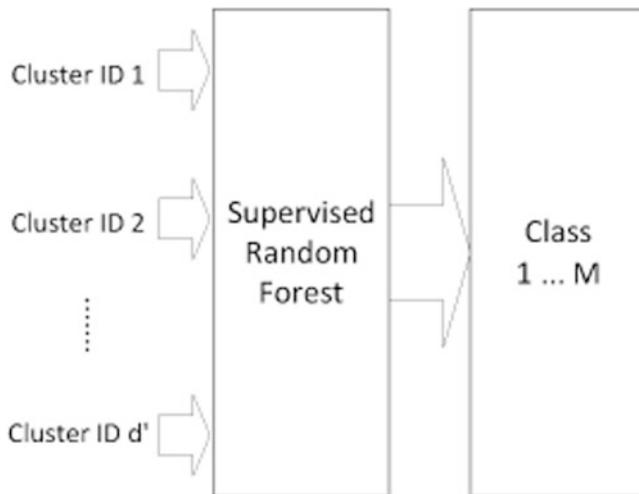


**Fig. 7.31** Unsupervised random forest for computing hidden values

as the hard clustering where each data item is arranged into only one cluster, in applying the unsupervised random forest. The hidden vector which is generated from the input vector through the unsupervised layer becomes the classification target.

The supervised layer where a novice data item is classified by the random forest is illustrated in Fig. 7.32. The hidden vector is computed from the input vector in the unsupervised layer which is mentioned above. Multiple decision trees are involved in a single random forest; the hidden vector is classified by a group of decision tree. Each decision tree in the random forest is trained with its own subset of hidden vectors which are mapped from training examples. The voting or the expert gate is chosen for deciding the final answer from the group of decision trees.

Let us make some remarks on the stacked random forest as a deep version. Multiple unsupervised versions and a single supervised version are involved in implementing the deep version. If it is assumed that the random forest consists of



**Fig. 7.32** Supervised random forest for classifying hidden values

$K$  decision trees and the dimension of hidden vector is  $d'$ ,  $K \times d'$  decision trees are involved in the unsupervised layer. The hidden vector with the  $d'$  dimension is classified by the supervised version of random forest. We consider implementing the convolution random forest by attaching alternatively the convolutional layers and the pooling layers.

## 7.5 Summary and Further Discussions

Let us summarize what is studied in this chapter. The decision tree and the random forest are the symbolic machine learning algorithms where symbolic rules are provided by trading the classification from a terminal node to the root node. The modified versions of decision tree by adding the input encoding or the output decoding are the output-decoded decision tree and the pooled decision tree. The decision trees are stacked by combining its supervised version and its unsupervised version into a deep version. This section is intended to discuss further that is studied in this chapter.

The pruning is to cut off some branches in the decision tree for improving the generalization performance. A big-sized decision tree is constructed for reducing the error on the training examples as much as possible. A set, called validation set, is separated from the training set, and while observing the error on the validation set, branches of decision tree are cut off. As the pruning continues, the training error increases, whereas the validation error decreases or increases. The effect of pruning the decision tree is to improve the generalization performance and to simplify the decision tree.

Let us consider applying the kernel function to derive the kernel-based decision tree. The kernel function is defined to two scalar values as well as two vectors, and it is applied to an attribute of training example and one corresponding to a branch. In the process of classifying a data item, edges whose kernel function values are maximal are selected from the root node to a terminal node. The training examples whose kernel function values are higher than threshold are retrieved in the process of learning the training examples for constructing the decision tree. The kernel-based decision tree is derived from the initial version of decision tree, by means of the fuzzy decision tree.

Let us consider the convolutional decision tree which is constructed from the mapped vectors. The training examples are initially prepared, a single filter vector is defined, and the mapped vectors are generated by applying the convolution to the training examples. The attributes are defined based on the mapped vectors, and the decision tree is constructed by them, in the learning process of the convolutional decision tree. In its generalization, a novice input vector is mapped into another vector by the convolution, and the mapped vector is classified following the edges from the root node to a terminal node. We consider defining multiple filter vectors as multiple channeled convolutions.

Let us consider attaching the convolution to the random forest as well as the decision tree. The training set is partitioned into subsets at random, and multiple decision trees each of which is trained with its own subset are constructed. In each decision tree in the random forest, the convolution is applied to each vector in its own subset. Instead of partitioning the training at random, differently mapped training sets are generated by applying different filter vectors to the training set, and each decision tree is trained with its own mapped training set. Various versions of convolutional random forest may be derived, depending on how to apply the convolution to a subset or the entire set, by a single filter or multiple filters.

## Reference

1. T. Jo, Text Mining: Concepts and Big Data Challenge, Springer, 2019.

# Chapter 8

## Deep Linear Classifier



This chapter is concerned with the modification of the linear classifier and the SVM into their deep versions. The SVM is a linear classifier which defines its dual parallel hyperplanes with the maximal margin between them as the classification boundary. Even if the SVM is viewed as a deep learning algorithm, compared with the simple linear classifier, by itself, it may be modified into its further deep versions by attaching the input encoding and/or the output encoding. As the advanced schemes, the SVM is modified into its unsupervised version, and its original and modified versions are combined with each other into the stacked version. This chapter is intended to describe the schemes of modifying the SVM into its deep versions.

This chapter is organized into the five sections, and in Sect. 8.1, we describe the deep SVM conceptually. In Sect. 8.2, we review the swallow version of SVM. In Sect. 8.3, viewing the SVM as a kind of deep learning, we mention its advanced versions. In Sect. 8.4, we describe the advanced deep version stacked by its unsupervised versions. In Sect. 8.5, we summarize the entire contents of this chapter and discuss further on what is studied in this chapter.

### 8.1 Introduction

This section is concerned with the overview of the deep SVM. The SVM is viewed as a deep learning algorithm, compared with the simple linear classifier, because the input vector is mapped into one in another space as the hidden vector. The SVM may be modified into its further deep version by adding the input encoding or the output decoding. The SVM is modified into its unsupervised version, and the deep version is implemented by combining its supervised version and its unsupervised version with each other. This section is intended to describe the SVM for providing its overview.

The linear classifier becomes the basis for the SVM. Before inventing the SVM, the linear classifier which defines a hyperplane as the classification boundary in the

vector space had been used. The linear separability is the case where all training examples are classified without any error by a single hyperplane. There are an infinite number of hyperplanes for classifying all training examples correctly in the linear separability. The hyperplane in the vector space is expressed as a linear combination of product of variables and weights.

The input vector is mapped into one in another space as the solution to the nonlinear separability in the SVM. The vector in the original space and one in another space are denoted by  $\mathbf{x}$  and  $\Phi(\mathbf{x})$ , respectively. The inner product of the vectors in another space,  $\Phi(\mathbf{x}_1)$  and  $\Phi(\mathbf{x}_2)$  is the kernel function of the vectors in the original space,  $\mathbf{x}_1$  and  $\mathbf{x}_2$ , as expressed in Eq. (8.1):

$$\Phi(\mathbf{x}_1) \cdot \Phi(\mathbf{x}_2) = K(\mathbf{x}_1, \mathbf{x}_2) \quad (8.1)$$

The inner product of the vectors in another space is computed through the kernel function without mapping individual vectors in the original space. It is required to define the kernel function for applying the SVM to the classification task.

In the SVM, dual parallel hyperplanes are defined as the classification boundary. In the simple linear classifier, a single hyperplane is defined; one is chosen at random among an infinite number of hyperplanes which classify the training examples completely in the linear separability. In the SVM, a single pair of parallel hyperplanes with their maximal margin is chosen as the most stable classification boundary. If the SVM is assumed as a binary classifier, one hyperplane is the boundary for the positive class, and the other is the boundary for the negative class. In principle, it is not allowed to put training examples between the dual hyperplanes.

Let us mention what is intended in this chapter. We review the linear classifier and the SVM as the swallow learning algorithms. Because the input vector is mapped into one in another space, the SVM is realized as a deep learning algorithm, but we propose its various further deep versions. We modify the SVM into its unsupervised version and construct the stacked version by combining its supervised version and its unsupervised version as the more advanced deep learning algorithm. This section is intended to understand the process of modifying the SVM into its deep version.

## 8.2 Support Vector Machine

This section is concerned with the original version of SVM. In Sect. 8.2.1, we mention the linear classifier as a simple machine learning algorithm. In Sect. 8.2.2, we mention the kernel function which indicates a similarity between vectors in the mapped space. In Sect. 8.2.3, we mention the equation and the classification process of SVM. In Sect. 8.2.4, we mention the constraints and the learning process.

### 8.2.1 Linear Classifier

This section is concerned with the linear classifier as a swallow learning algorithm. A line in the two-dimensional space and a plane in the three-dimensional space are defined in the linear classifier as the classification boundary. In the  $d$  dimensional space, a hyperplane is defined as the boundary for classifying a data item which is given as a  $d$  dimensional vector. If the given problem is characterized as the linear separability, there are an infinite number of hyperplanes for classifying the training examples without error. This section is intended to describe the linear classifier as a swallow learning algorithm.

The linear separability in the binary classification is illustrated in Fig. 8.1. The given task is assumed as a binary classification where each item is classified into one of the two categories, and each training example is represented into a two-dimensional vector. Each training example is labeled with the positive class or the negative class, and the training examples are plotted into the two-dimensional space, as shown in Fig. 8.1. The line becomes the boundary for classifying the training examples without any error in the linear separable classification. The classification boundary is expressed with a linear equation of the two variables.

The hyperplanes and their equations in the two-dimensional and the three-dimensional spaces are presented in Fig. 8.2. The hyperplane in the two dimension is a line which is expressed with Eq. (8.2):

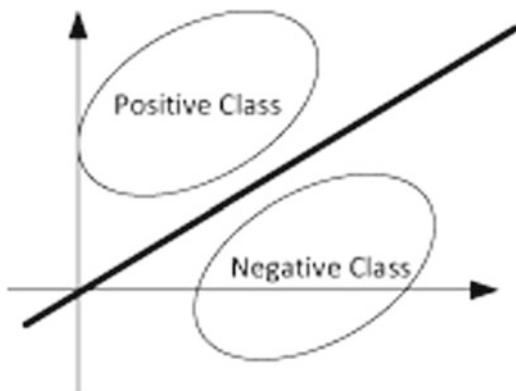
$$w_1x_1 + w_2x_2 = b \quad (8.2)$$

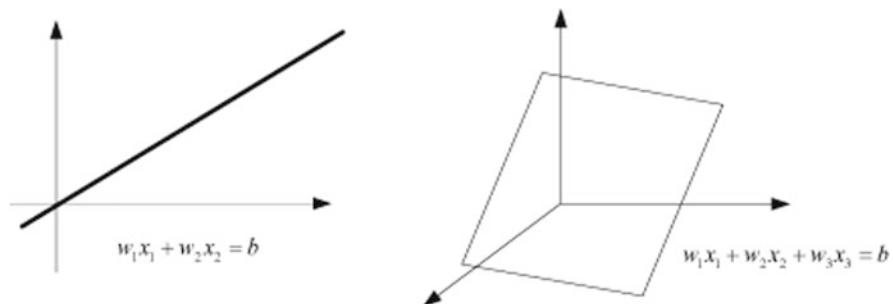
The hyperplane in the three dimensions is a plane which is expressed with Eq. (8.3):

$$w_1x_1 + w_2x_2 + w_3x_3 = b \quad (8.3)$$

The hyperplane in the  $d$  dimensional space is expressed with Eq. (8.4):

**Fig. 8.1** Linearly separable





**Fig. 8.2** Line equation and plane equation

$$w_1x_1 + w_2x_2 + \dots + w_dx_d - b \geq 0 \rightarrow \text{Positive Class}$$

$$w_1x_1 + w_2x_2 + \dots + w_dx_d - b < 0 \rightarrow \text{Negative Class}$$

**Fig. 8.3** Linear classification rule

$$w_1x_1 + w_2x_2 + \dots + w_dx_d = b \quad (8.4)$$

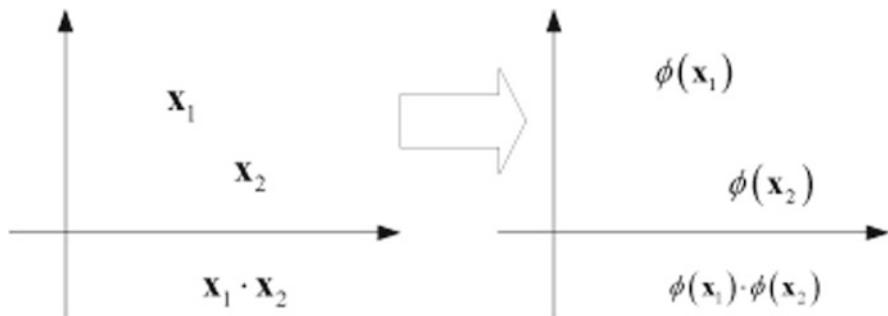
In Eq. (8.4),  $x_1, x_2, \dots, x_d$  are called input variables, and  $w_1, w_2, \dots, w_d$  are called weights or coefficients. The weight vector,  $\mathbf{w} = [w_1 \ w_2 \ \dots \ w_d]$ , indicates the direction which is perpendicular to the hyperplane.

The classification of a data item by the linear classifier is illustrated in Fig. 8.3. The hyperplane in the  $d$  dimension is expressed by Eq. (8.4). If  $w_1x_1 + w_2x_2 + \dots + w_dx_d \geq b$ , the data item is classified into the positive class, and otherwise, it is classified into the negative class. The learning process of the linear classifier is to search for the weight vector for classifying the training examples without error. If the linear separability keeps in the given space, there is an infinite number of weight vectors for satisfying the correct classification.

Let us make some remarks on the linear classifier as an approach to the data classification. The given task is assumed as a binary classification with its linear separability for explaining the linear classifier. The hyperplane which is the classification is expressed as a linear combination of variables and weights. If a value is greater than or equal to the bias, the data item is classified into the positive class, and otherwise it is classified into the negative class. The hyperplane which classifies the training examples without error is found by manipulating the weights.

### 8.2.2 Kernel Function

This section is concerned with the kernel function which is necessary for implementing the SVM. Its idea is to transform the nonlinear separability into linear



**Fig. 8.4** Mapping and inner product

**Fig. 8.5** Mathematical properties of kernel function

$$\begin{aligned} K(x, y) &= K_1(x, z) + K_2(x, z) \\ K(x, y) &= \alpha K_1(x, y) \\ K(x, y) &= K_1(x, y)K_2(x, y) \\ K(x, y) &= f(x)f(y) \\ K(x, y) &= K_3(\varphi(x), \varphi(y)) \\ K(x, y) &= x \otimes y \end{aligned}$$

separability by mapping an input vector into one in another space. The inner product of two vectors is always given as a scalar value in every dimensional space, and it is possible to compute the inner product of two vectors in the mapped space without mapping them by defining a kernel function. The kernel function of two vectors in the original space indicates the similarity between ones in another space. This section is intended to describe the kernel function with respect to mathematical definition.

The original vectors and the mapped vectors are illustrated in Fig. 8.4. The two original vectors are denoted by  $\mathbf{x}_1$  and  $\mathbf{x}_2$ , and the vectors which are mapped from  $\mathbf{x}_1$  and  $\mathbf{x}_2$  are denoted by  $\Phi(\mathbf{x}_1)$  and  $\Phi(\mathbf{x}_2)$ . The inner product of two vectors,  $\Phi(\mathbf{x}_1)$  and  $\Phi(\mathbf{x}_2)$ , is defined as a kernel function of the vectors,  $\mathbf{x}_1$  and  $\mathbf{x}_2$ , as shown in Eq. (8.1). Both the inner product of  $\mathbf{x}_1$  and  $\mathbf{x}_2$  and the inner product of  $\Phi(\mathbf{x}_1)$  and  $\Phi(\mathbf{x}_2)$  are given as scalar values. The inner product of two mapped vectors is computed without mapping the original vectors by the kernel function.

The mathematical properties of the kernel function are illustrated in Fig. 8.5. The addition of two kernel functions and the multiplication of a kernel function with a constant value are also kernel functions. The product of two kernel functions and product of two functions are kernel functions. The product of a vector, a matrix, and

$$\begin{bmatrix} K(\mathbf{x}_1, \mathbf{x}_1) & K(\mathbf{x}_1, \mathbf{x}_2) & \dots & K(\mathbf{x}_1, \mathbf{x}_N) \\ K(\mathbf{x}_2, \mathbf{x}_1) & K(\mathbf{x}_2, \mathbf{x}_2) & \dots & K(\mathbf{x}_2, \mathbf{x}_N) \\ \dots & \dots & \dots & \dots \\ K(\mathbf{x}_N, \mathbf{x}_1) & K(\mathbf{x}_N, \mathbf{x}_2) & \dots & K(\mathbf{x}_N, \mathbf{x}_N) \end{bmatrix}$$

**Fig. 8.6** Kernel matrix

another vector is viewed as a kernel function of two vectors. The output of kernel function of vectors is given as a scalar value.

The kernel matrix is illustrated in Fig. 8.6. The training set is assumed as a set of consists of  $N$  training examples,  $Tr = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$ . The kernel matrix is one where both each column and each row correspond to a training example,  $\mathbf{x}_i$ , and each element is the output of the kernel function of two corresponding training examples. The determinant of the kernel matrix is greater than or equal to zero; the matrix with this property is called semi-definite matrix. The kernel matrix is used for applying the kernel-based linear equation of SVM to the data classification.

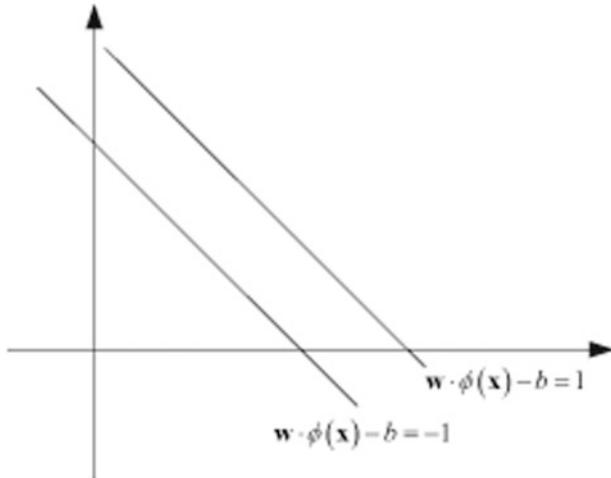
Let us make some remarks on the kernel function which is used as the main operation in the SVM. The output value of the kernel function of two vectors means the inner product of them. The mathematical properties of the kernel function are listed in Fig. 8.5. A kernel matrix consists of output values of kernel function of the training examples. The definition of the kernel function is intended to avoid mapping explicitly individual input vectors into ones in another space.

### 8.2.3 SVM Classifier

This section is concerned with the SVM classifier as a binary classifier. In Sect. 8.2.1, we studied the linear classifier which defines a single hyperplane as the classification boundary. The SVM classifier defines dual parallel hyperplanes as the maximal margin between them in the original space or the mapped space. The weight-based linear equation is transformed into the Lagrange multiplier linear equation for expressing the SVM classifier. This section is intended to describe the linear equation which expresses the SVM classifier.

The weight-based equations of dual hyperplanes are illustrated in Fig. 8.7. An original vector and a mapped one are notated by  $\mathbf{x}$  and  $\Phi(\mathbf{x})$ , respectively. The linear equation is defined as Eq. (8.5):

$$y = \mathbf{w} \cdot \Phi(\mathbf{x}) - b \quad (8.5)$$



**Fig. 8.7** Weight-based linear equation

If  $y \geq 1$ , the vector,  $\mathbf{x}$ , is classified into the positive class, and if  $y \leq -1$ , it is classified into the negative class. The equations of dual hyperplanes are defined as Eqs. (8.6) and (8.7):

$$\mathbf{w} \cdot \Phi(\mathbf{x}) - b = 1 \quad (8.6)$$

$$\mathbf{w} \cdot \Phi(\mathbf{x}) - b = -1 \quad (8.7)$$

The area in  $-1 < y < 1$  becomes the neutral zone between the dual hyperplanes.

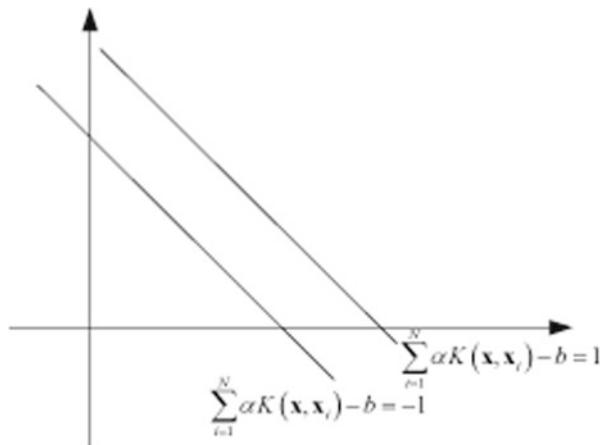
The equations of dual hyperplanes are illustrated in Fig. 8.8. Equation (8.5) is transformed into Eq. (8.8) based on the dependency of the weight vector on the training examples:

$$y = \sum_{i=1}^N \alpha_i \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}) - b \quad (8.8)$$

where  $\alpha_i$  is called Lagrange multiplier. Because the inner product of two mapped vectors is expressed as kernel function as shown in Eq. (8.1), Eq. (8.8) is transformed into Eq. (8.9):

$$y = \sum_{i=1}^N \alpha_i K(\mathbf{x}_i, \mathbf{x}) - b \quad (8.9)$$

The dual hyperplanes are defined as Eqs. (8.10) and (8.11):



**Fig. 8.8** Lagrange multiplier based linear equation

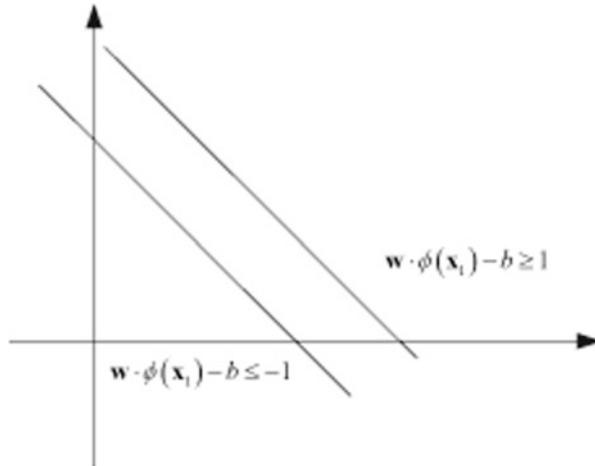
$$\sum_{i=1}^N \alpha_i K(\mathbf{x}_i, \mathbf{x}) - b = 1 \quad (8.10)$$

$$\sum_{i=1}^N \alpha_i K(\mathbf{x}_i, \mathbf{x}) - b = -1 \quad (8.11)$$

The learning process of SVM is to optimize the Lagrange multipliers.

The classification rule which is based on the SVM equations is illustrated in Fig. 8.9. The task is assumed as a binary classification, and the SVM is applied to the task. If  $\sum_{i=1}^N \alpha_i K(\mathbf{x}_i, \mathbf{x}) - b \geq 1$ , the novice item,  $\mathbf{x}$ , is classified into the positive class, and if  $\sum_{i=1}^N \alpha_i K(\mathbf{x}_i, \mathbf{x}) - b \leq -1$ ,  $\mathbf{x}$  is classified into the negative class. The training examples near the classification boundary are considered for defining the classification boundary; the nonzero Lagrange multipliers are assigned to them. The training examples which correspond to the nonzero Lagrange multipliers are called support vectors.

Let us make some remarks on the classification rule in the SVM which is covered in this section. The SVM classifier is expressed as the inner product of the weight vector and the mapped input vector. The weight vector-based equation is transformed into the sum of the inner products of a novice vector and a training example. The novice item is classified into the positive class, if the output is greater than or equal to 1, and it is classified into the negative class if the output is less than or equal to -1. When applying the SVM to the multiple classification, it is decomposed into binary classifications.



**Fig. 8.9** Linear classification equation

### 8.2.4 Dual Constraints

This section is concerned with the process of training the SVM classifier using the training examples. In the previous section, the weight-based linear equation is transformed into the Lagrange multiplier based one for expressing the SVM classifier. The learning task of SVM is expressed as the primal problem, and it is transformed into the dual problem. The SMO (sequential minimization optimization) algorithm is adopted for training the SVM classifier. This section is intended to describe the constraint and the learning process of the SVM classifier.

The primal problem is derived from the SVM equation as the constraints for optimizing the weights. We take Eq. (8.12) for the correct classification and Eq. (8.13) which should be minimized for the maximal margin:

$$d_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 \quad (8.12)$$

$$\min \Phi(\mathbf{w}) = \frac{1}{2} \mathbf{w}^T \mathbf{w}. \quad (8.13)$$

Equation (8.14) is derived for satisfying the constraints, Eqs. (8.12) and (8.13):

$$J(\mathbf{w}, b, \alpha) = \frac{1}{2} \mathbf{w}^T \mathbf{w} - \sum_{i=1}^N \alpha_i [d_i(\mathbf{w}^T \mathbf{x}_i + b) - 1]. \quad (8.14)$$

We derive the constraints which are expressed in Eqs. (8.15) and (8.16) respectively by the partial differentiations,  $\frac{\partial J(\mathbf{w}, b, \alpha)}{\partial \mathbf{w}} = 0$  and  $\frac{\partial J(\mathbf{w}, b, \alpha)}{\partial b} = 0$ ,

$$\mathbf{w} = \sum_{i=1}^N \alpha_i d_i \mathbf{x}_i \quad (8.15)$$

$$\sum_{i=1}^N \alpha_i d_i = 0. \quad (8.16)$$

The constraints of the dual problem are derived from ones of the primal problem.

Let us transform the constraints of the primal problem into ones of the dual problem. Equation (8.14) is transformed into Eq. (8.17) by Eq. (8.16):

$$J(\mathbf{w}, b, \alpha) = \frac{1}{2} \mathbf{w}^T \mathbf{w} - \sum_{i=1}^N \alpha_i d_i \mathbf{w}^T \mathbf{x}_i + \sum_{i=1}^N \alpha_i. \quad (8.17)$$

Equation (8.17) is transformed into Eq. (8.18) by Eq. (8.15):

$$Q(\alpha) = \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j d_i d_j \mathbf{x}_i^T \mathbf{x}_j. \quad (8.18)$$

Equation (8.18) should be maximized by the constraint which is expressed in Eq. (8.16). The weights and the bias are computed with the optimized Lagrange multipliers by Eqs. (8.19) and (8.20):

$$\mathbf{w}_o = \sum_{i=1}^N \alpha_{o,i} d_i \mathbf{x}_i \quad (8.19)$$

$$b_o = 1 - \mathbf{w}_o^T \mathbf{x}^{(S)}, \quad (8.20)$$

where  $\mathbf{x}^{(S)}$  is any training example in the training set.

The algorithm of optimizing the Lagrange multipliers is illustrated in Fig. 8.10. The Lagrange multipliers are updated until satisfying the KKT (Karush-Kuhn-Tucker) condition, as follows:

$$\alpha_i = 0 \Leftrightarrow d_i y_i > 1$$

$$0 \leq \alpha_i \leq C \Leftrightarrow d_i y_i = 1$$

$$\alpha_i = C \Leftrightarrow d_i y_i < 1$$

All the Lagrange multipliers are initialized as zero values, and the Lagrange multipliers are updated whenever the KKT condition is violated, as follows:

$\alpha_i = 0 \Leftrightarrow d_i y_i > 1$   
 $0 \leq \alpha_i \leq C \Leftrightarrow d_i y_i = 1$   
 $\alpha_i = C \Leftrightarrow d_i y_i < 1$

do until satify KKT condition   $\alpha_i = C \Leftrightarrow d_i y_i < 1$

```

for(int i = 0;i < N;i++)
     $\alpha_i = 0$  //initialize lagrange multipliers
for(int i = 0;i < N;i++)
    //for each training example
    if violate against KKT condition
        index1 = i;
        index2 = rand(0,N - 1) //except i
        updateLagrangeMultipliers(index1,index2)
        updateBais(index1,index2)
    
```

**Fig. 8.10** Sequential minimal optimization algorithm

```

updateLagrangeMultipliers(index1, index2)
if( $d_{index1} == d_{index2}$ )
     $L = \max(0, \alpha_{index2} + \alpha_{index1} - C)$ 
     $H = \max(0, \alpha_{index2} + \alpha_{index1})$ 
    if( $d_{index1} != d_{index2}$ )
         $L = \max(0, \alpha_{index2} - \alpha_{index1})$ 
         $H = \max(0, C + \alpha_{index2} - \alpha_{index1})$ 
     $\eta = K(\mathbf{x}_{index1}, \mathbf{x}_{index1}) + K(\mathbf{x}_{index2}, \mathbf{x}_{index2}) - 2K(\mathbf{x}_{index1}, \mathbf{x}_{index2})$ 
     $E_{index2} = y_{index2} - d_{index2}$ 
     $E_{index1} = y_{index1} - d_{index1}$ 
     $\alpha_{index2}^{new} = \alpha_{index2} + \frac{d_{index2}(E_{index1} - E_{index2})}{\eta}$ 
     $\alpha_{index1}^{new} = \alpha_{index1} + (d_{index1}d_{index2})(\alpha_{index2} - \alpha_{index2}^{new})$ 

```

The bias is updated as follows:

$$\begin{aligned}
& \text{updateBias}(index1, index2) \\
b_1 &= E_{index1} + d_{index1}(\alpha_{index1}^{new} - \alpha_{index1})K(\mathbf{x}_{index1}, \mathbf{x}_{index1}) \\
&\quad + d_{index2}(\alpha_{index2}^{new} - \alpha_{index2})K(\mathbf{x}_{index1}, \mathbf{x}_{index2}) \\
b_2 &= E_{index2} + d_{index1}(\alpha_{index1}^{new} - \alpha_{index1})K(\mathbf{x}_{index2}, \mathbf{x}_{index1}) \\
&\quad + d_{index2}(\alpha_{index2}^{new} - \alpha_{index2})K(\mathbf{x}_{index2}, \mathbf{x}_{index2}) \\
b^{new} &= \frac{1}{2}(b_1 + b_2)
\end{aligned}$$

$0 \leq \alpha_i \leq C$  is added to the constraints which are defined in (8.14), (8.15), and (8.16).

Let us make some remarks on the learning process of the SVM. The primal problem is defined by deriving the constraints for optimizing the Lagrange multipliers. The primal problem is converted into the dual problem by the above process. The SMO algorithm which is described above is used based on the dual problem for optimizing the Lagrange multipliers. The evolutionary computation is considered for optimizing them as the learning process of the SVM.

## 8.3 Basic Deep Versions

This section is concerned with the deep version of SVM. In Sect. 8.3.1, we state that the SVM is a deep linear classifier, because the input vector is mapped into one in another space. In Sect. 8.3.2, we describe the SVM with multiple kernel functions. In Sect. 8.3.3, we mention multiple SVMs which are applied to the multiple classification. In Sect. 8.3.4, we mention the SVM version which is added by the pooling layer.

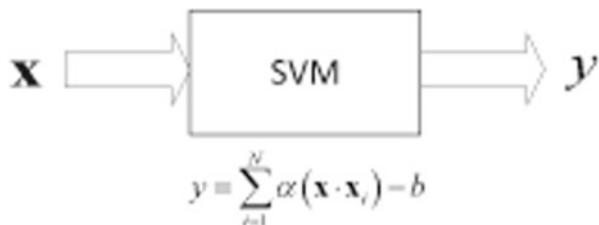
### 8.3.1 SVM as Deep Learning Algorithm

This section is concerned with viewing of the SVM into a deep learning algorithm. In the swallow learning, the output vector is computed directly from the input vector, whereas in the deep learning, it is computed from a hidden vector. The fact that the input vector is mapped into one in another space in the general version of SVM becomes the reason of regarding the SVM as a deep learning. The primitive version of SVM whose kernel function is the inner product of original vectors belongs to the swallow learning. This section is intended to describe the SVM with the view of deep learning.

Mapping the input vector into one in another space is illustrated in Fig. 8.11. The original vector and the mapped one are notated by  $\mathbf{x}$  and  $\Phi(\mathbf{x})$ , respectively.



**Fig. 8.11** Feature mapping in SVM



**Fig. 8.12** Primitive SVM: linear SVM

We already mentioned the mapping in Sect. 6.3.1, and it is viewed as the process of computing the hidden vector,  $\mathbf{h} = \Phi(\mathbf{x})$  from the input vector. If the training set,  $Tr = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$ , is prepared, and the training examples are mapped into  $\Phi(\mathbf{x}_1), \Phi(\mathbf{x}_2), \dots, \Phi(\mathbf{x}_N)$ . The fact that the process of mapping the input vector is regarded as the process of computing the hidden vector is the evidence of the fact that the SVM is a deep learning algorithm.

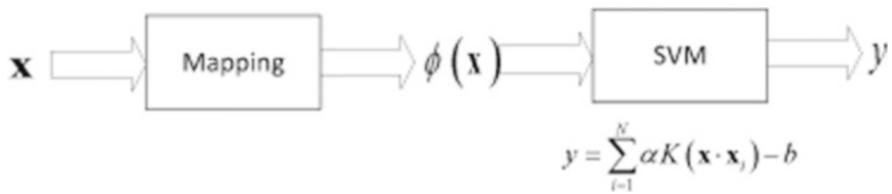
The primitive version of SVM is illustrated in Fig. 8.12. Even if the SVM is mentioned as a deep learning algorithm above, the primitive version whose kernel function is the inner product of the original vectors should be regarded as a swallow learning algorithm. The kernel function of a training example and a novice vector is expressed as Eq. (8.21):

$$K(\mathbf{x}_i, \mathbf{x}) = \mathbf{x}_i \cdot \mathbf{x} \quad (8.21)$$

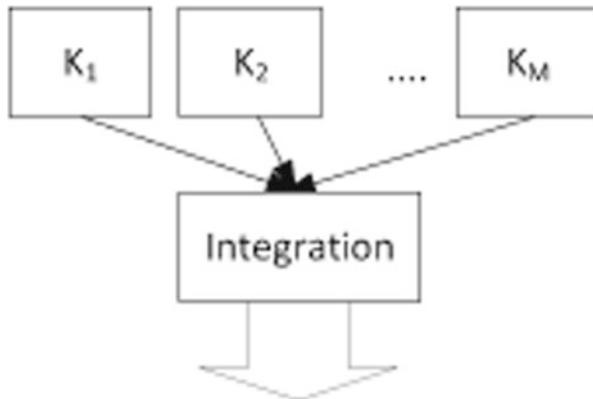
The vector is not mapped into one in another space in the primitive version; the input space is not changed. This SVM version is viewed as a swallow learning algorithm because the output is computed directly from the original input vector.

The kernel version of SVM is illustrated as a deep learning algorithm in Fig. 8.13. The primitive version is viewed as a swallow learning algorithm because a vector is not mapped into one in another space. The kernel function is the inner product of mapped vectors which is obtained without mapping vectors. The inner product of the mapped vectors is provided from the kernel function; the hidden vector is implicitly computed from the input vector. The SVM is viewed as a deep learning algorithm except its primitive version.

Let us make some remarks on the viewing of the SVM as a deep learning algorithm. The process of mapping a vector into one in another space is viewed as the process of computing a hidden vector from the input vector. The primitive SVM version where its kernel function is the inner product of two original vectors belongs



**Fig. 8.13** General SVM: SVM with feature mapping



**Fig. 8.14** Parallel combination of kernel functions

to the swallow learning, because the input vector is not mapped into one in another space. The general version of SVM is viewed as a deep learning algorithm. The SVM is modified into its further deep version by attaching multiple kernel functions.

### 8.3.2 *Multiple Kernel-Based SVM*

This section is concerned with another deep version of SVM, called multiple kernel-based SVM. In the previous section, it is asserted that the SVM is a deep learning algorithm by itself except its primitive version. In this section, we study the SVM with the combination of multiple kernel functions, as a more advanced deep version than the SVM with a single kernel function. There are typical types of kernel functions, such as linear kernel function, polynomial kernel function, Gaussian kernel function, and sigmoid kernel function, and it is possible to combine various kernel functions in this version of SVM. This section is intended to describe the SVM with the combination of multiple kernel functions as a deep learning algorithm.

The parallel combination of multiple kernel functions is illustrated in Fig. 8.14. The multiple kernel functions,  $K_1, K_2, \dots, K_M$ , are applied to the two vectors,



**Fig. 8.15** Serial combination of kernel functions

$\mathbf{x}_1$  and  $\mathbf{x}_2$ . The output value of the multiple kernel functions of the two vectors is decided by averaging their output, as shown in Eq. (8.22):

$$K_f(\mathbf{x}_1, \mathbf{x}_2) = \frac{1}{M} \sum_{i=1}^M K_i(\mathbf{x}_1, \mathbf{x}_2) \quad (8.22)$$

As a variant, the output may be decided by voting with the discriminations as shown in Eq. (8.23):

$$K_f(\mathbf{x}_1, \mathbf{x}_2) = \frac{1}{M} \sum_{i=1}^M w_i K_i(\mathbf{x}_1, \mathbf{x}_2) \quad (8.23)$$

where  $\sum_{i=1}^M w_i = 1.0$ . The kernel functions are independent of each other in this combination type.

The serial combination of multiple kernel functions is illustrated in Fig. 8.15. The  $d$  intermediate vectors,  $\theta_1, \theta_2, \dots, \theta_d$ , are prepared, and the input vectors are notated by  $\mathbf{x}_1^{(1)} = \mathbf{x}_1$  and  $\mathbf{x}_2^{(1)} = \mathbf{x}_2$ . The input vectors in the next generation are computed by Eqs. (8.24) and (8.25):

$$\mathbf{x}_1^{(2)} = [K_1(\mathbf{x}_1^{(1)}, \theta_1) \ K_1(\mathbf{x}_1^{(1)}, \theta_2) \ \dots \ K_1(\mathbf{x}_1^{(1)}, \theta_d)] \quad (8.24)$$

$$\mathbf{x}_2^{(2)} = [K_1(\mathbf{x}_2^{(1)}, \theta_1) \ K_1(\mathbf{x}_2^{(1)}, \theta_2) \ \dots \ K_1(\mathbf{x}_2^{(1)}, \theta_d)] \quad (8.25)$$

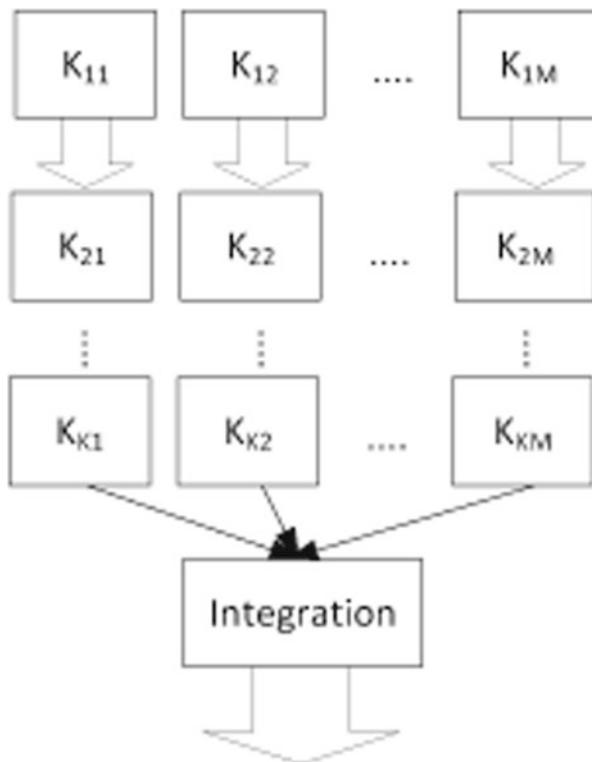
The relation between the vectors in the current generation and one in the previous generation is defined by Eqs. (8.26) and (8.27):

$$\mathbf{x}_1^{(i)} = [K_{i-1}(\mathbf{x}_1^{(i-1)}, \theta_1) \ K_{i-1}(\mathbf{x}_1^{(i-1)}, \theta_2) \ \dots \ K_{i-1}(\mathbf{x}_1^{(i-1)}, \theta_d)] \quad (8.26)$$

$$\mathbf{x}_2^{(i)} = [K_{i-1}(\mathbf{x}_2^{(i-1)}, \theta_1) \ K_{i-1}(\mathbf{x}_2^{(i-1)}, \theta_2) \ \dots \ K_{i-1}(\mathbf{x}_2^{(i-1)}, \theta_d)] \quad (8.27)$$

The final value is computed by Eq. (8.28):

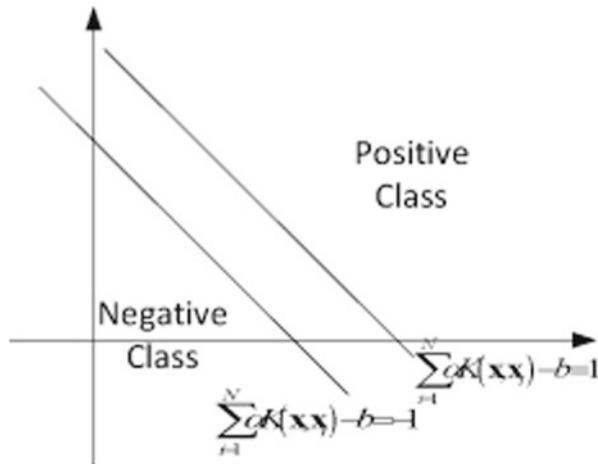
$$K_f(\mathbf{x}_1, \mathbf{x}_2) = K_M(\mathbf{x}_1^{(M-1)}, \mathbf{x}_2^{(M-1)}) \quad (8.28)$$



**Fig. 8.16** Parallel and serial combination of kernel functions

Both parallel and serial combinations of multiple kernel functions are illustrated in Fig. 8.16. The  $K \times M$  kernel functions are involved in computing the output value. It is computed from the kernel function,  $K_{1i}$ , to the kernel function,  $K_{Ki}$  in the serial direction by the process mentioned above. The final output value is computed by averaging the output values of the kernel functions,  $K_{K1}, K_{K2}, \dots, K_{KM}$ . The process of computing the output in both the serial combination and the parallel combination is described above.

Let us make some remarks on the SVM with multiple kernel functions. The final output value is computed by averaging the output values of kernel functions in the parallel combination of multiple kernel functions. In addition,  $d$  intermediate values are involved as separated vectors from two input vectors, in the serial combination of kernel functions. The output value may be computed with both the parallel combination and the serial combination of kernel functions. In the parallel combination, multiple hidden vectors are computed at same time in a single layer, whereas in the serial combination, the hidden vectors are computed with multiple steps.



**Fig. 8.17** Support vector machine: binary classifier

### 8.3.3 SVM for Multiple Classification

This section is concerned with the application of SVM to the multiple classification tasks. In the previous version, we studied the deep version of SVM as a binary classifier. The SVM is applied to the multiple classification as a real task, rather than the binary classification as a toy task. The multiple classification is decomposed into binary classifications, and a SVM is applied to each of them. This section is intended to describe the scheme of applying the SVM to the multiple classification.

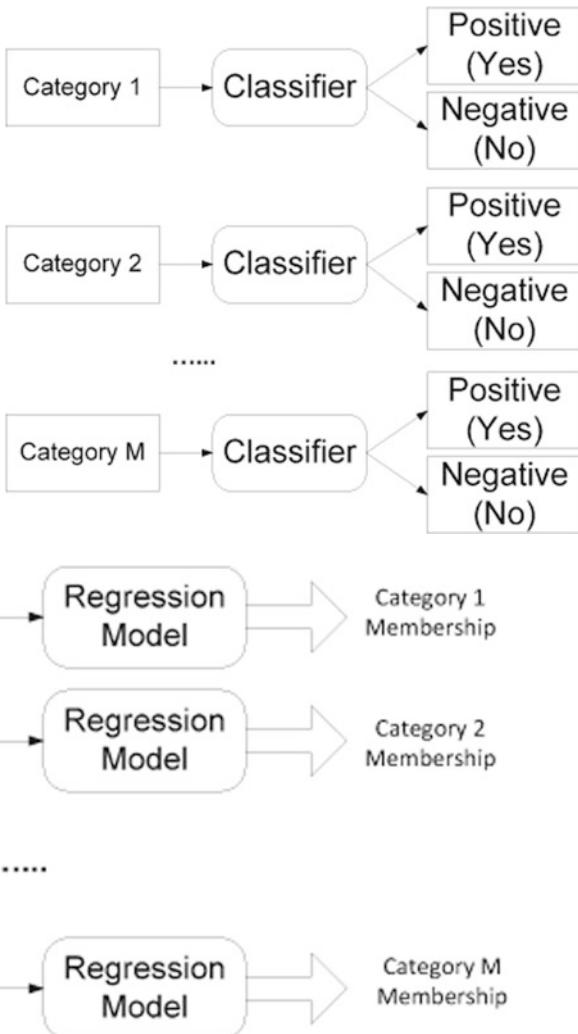
The SVM is illustrated as a binary classifier in Fig. 8.17. The parallel dual hyperplanes are constructed by learning the training examples. The area which corresponds to  $\sum_{i=1}^N \alpha_i K(\mathbf{x}_i, \mathbf{x}) - b \geq 1$  is defined as the positive class, and the

area which corresponds to  $\sum_{i=1}^N \alpha_i K(\mathbf{x}_i, \mathbf{x}) - b \leq -1$  is defined as the negative class.

The area which corresponds to  $-1 < \sum_{i=1}^N \alpha_i K(\mathbf{x}_i, \mathbf{x}) - b < 1$  is the neutral zone for classifying data items more stably. The SVM is initially designed as a binary classifier.

The decomposition of the multiple classification into binary classifications is illustrated in Fig. 8.18. The multiple classification is assumed as the overlapping classification where each item may be classified into more than one category. A binary classifier is allocated for each category, and an item is classified into one of the two categories, “yes” and “no.” Among the  $M$  categories, ones whose classifiers provide the category, “yes,” are assigned to the item. The SVM which is designed

**Fig. 8.18** Decomposition of multiple classification into binary classifications



**Fig. 8.19** Fuzzy multiple classification

as a binary classifier may be applied to the multiple classification by decomposing it into binary classifications.

The decomposition of the fuzzy multiple classification into univariate regressions is illustrated in Fig. 8.19. The fuzzy multiple classification is the process of estimating category membership values as continuous values. In this decomposition, a univariate regression model is allocated for each category, and the output value is estimated by the corresponding regression model as a continuous value between zero and one. The continuous output from Eq.(8.9) is normalized into a value

between zero and one in applying the SVM to the fuzzy multiple classification. In this application, Eq. (8.9) is given as the regression model.

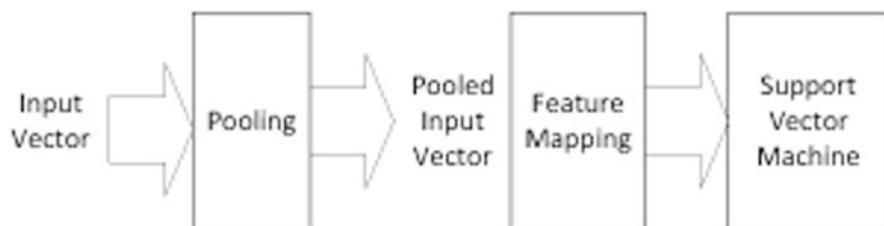
Let us make some remarks on the application of SVM to the multiple classification task. The SVM is initially designed as a binary classifier where the space is separated into two areas by the dual parallel hyperplanes. The multiple classification is decomposed into binary classification tasks in applying the SVM to it. Equation (8.9) is viewed as a univariate regression model in applying the SVM to the fuzzy multiple classification. We consider applying Eq. (8.9) to a regression task such as a nonlinear function approximation.

### 8.3.4 Pooled SVM

This section is concerned with the further deep version of SVM with the pooling layer. In Sect. 8.3.1, we mentioned the SVM as a deep learning algorithm, except its primitive version. In this section, we upgrade the SVM into its further deep version by adding the pooling layer. In this version, the input vector is mapped into the reduced sized vector by the pooling and mapped into a vector in another space as the two steps of computing a hidden vector. This section is intended to describe the further deep version of SVM by adding the pooling layer.

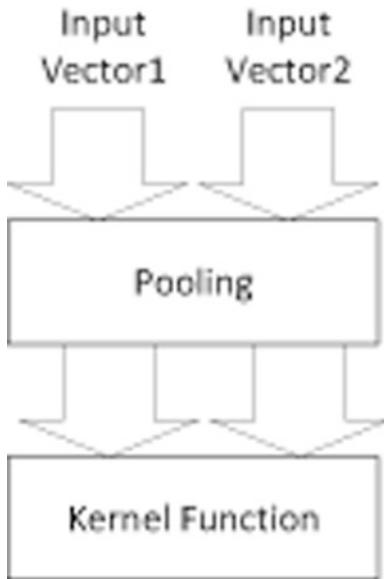
The attachment of the pooling layer in front of the SVM is illustrated in Fig. 8.20. The SVM was previously mentioned as a deep learning algorithm with the reason of mapping implicitly the input vector into one in another space. There are two steps of computing the hidden vector in this version of SVM: one applies the pooling to the input vector and the other maps the vector into one in another space. The process of applying the pooling to the vector is explicit, whereas the process of mapping vector into one in another space is implicit. The kernel function is applied to the vectors which are mapped by the pooling for computing the inner product of vectors in another space.

The application of the kernel function to the mapped vectors, instead of the input vectors, is illustrated in Fig. 8.21. The two original input vectors are denoted by  $\mathbf{x}_1$  and  $\mathbf{x}_2$ . The two vectors,  $\mathbf{x}_1$  and  $\mathbf{x}_2$ , are mapped into the two hidden vectors,  $\mathbf{h}_1$  and  $\mathbf{h}_2$ , by the pooling. The kernel function is applied as shown in Eq. (8.29):



**Fig. 8.20** Pooling and feature mapping in SVM

**Fig. 8.21** Application of kernel function to pooled vectors



$$K(\mathbf{h}_1, \mathbf{h}_2) = \Phi(\mathbf{h}_1) \cdot \Phi(\mathbf{h}_2). \quad (8.29)$$

One more mapping from the hidden vectors,  $\mathbf{h}_1$  and  $\mathbf{h}_2$ , into  $\Phi(\mathbf{h}_1)$  and  $\Phi(\mathbf{h}_2)$  is implicit by the kernel function.

Let us mention the process of classifying a data item by this version of SVM. The training examples in the set,  $Tr = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$ , are mapped into the set of mapped ones,  $Tr_h = \{\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_N\}$ . The equation of SVM is defined by learning vectors in  $Tr_h$  is defined as Eq. (8.30):

$$y = \sum_{i=1}^N \alpha_i K(\mathbf{h}, \mathbf{h}_i) + b \quad (8.30)$$

The input vector,  $\mathbf{x}$  is mapped into  $\mathbf{h}$ , and it is classified by Eq. (8.30). The process of mapping the original vector into another vector by the pooling is added.

Let us make some remarks on the deep version of SVM by attaching the pooling layer. In this version of SVM, the input vector is mapped into another vector by the pooling layer and mapped one more time implicitly into one in another space. The kernel function is applied to the hidden vectors which are mapped from the initial input vectors for computing the inner product of the finally mapped vectors. The original vectors are replaced by the hidden vectors in this version of SVM. The version of SVM which is called convolutional SVM is implemented by attaching the pooling layer and the convolution layer, alternatively.

## 8.4 Advanced Deep Versions

This section is concerned with the advanced deep version of the linear classifier and the SVM. In Sect. 8.4.1, we modify the linear classifier into its unsupervised version. In Sect. 8.4.2, we construct the deep version of linear classifier by adding the unsupervised layer. In Sect. 8.4.3, we modify the SVM into its unsupervised version. In Sect. 8.4.4, we construct the deep version of SVM by adding the unsupervised layer.

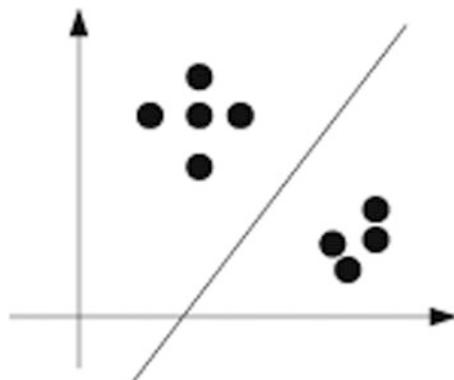
### 8.4.1 Unsupervised Linear Classifier

This section is concerned with the unsupervised version of linear classifier. Until now, the linear classifier has been used as a supervised learning algorithm which is applied to the classification and the regression. A single linear classifier may be applied to a binary clustering as its unsupervised version. Multiple linear classifiers are used for multiple clustering by corresponding each of them to a cluster. This section is intended to describe the unsupervised version of linear classifier.

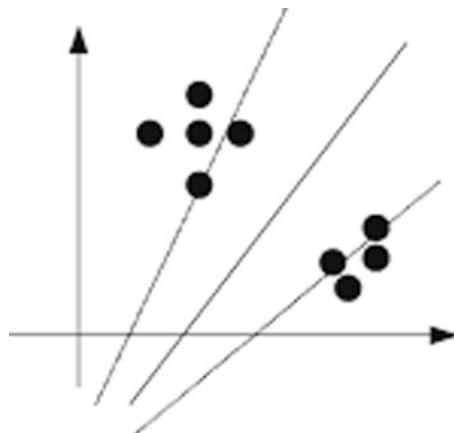
The linear boundary for defining two clusters is illustrated in Fig. 8.22. If the data items which are given as two-dimensional vectors are plotted in the two-dimensional space, the line which is the boundary between clusters is searched. The linear boundary is expressed as a linear equation in the  $d$  dimensional space,  $w_1x_1 + w_2x_2 + \dots + w_dx_d = c$ . The weight values,  $w_1, w_2, \dots, w_d$ , are optimized for minimizing the error between the target output and the computed output in the supervised linear classifier. In this version, the weights are optimized depending on the similarities among the input vectors.

The direction of optimizing the weights for the binary clustering is illustrated in Fig. 8.23. The unsupervised linear classifier is assumed for partitioning a group into two subgroups. The weights are initialized at random, and the data items are

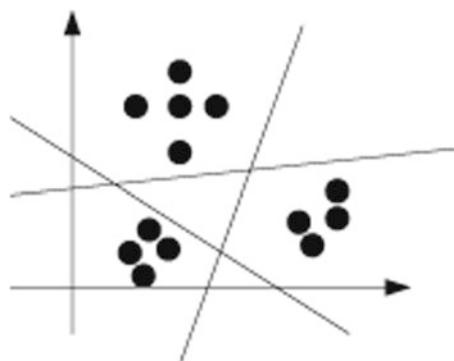
**Fig. 8.22** Linear boundary for binary clustering



**Fig. 8.23** Optimization of linear boundary



**Fig. 8.24** Multiple linear boundaries for multiple clustering



arranged by applying the linear classifier into one of the two clusters. The clustering index is computed to the two clusters, data items which cause the degrade of clustering index are selected, and the weights are updated based on the selected ones. The weights are optimized in the unsupervised linear classifier for maximizing the clustering index.

The application of the unsupervised learning classifiers to the multiple clustering is illustrated in Fig. 8.24. A single unsupervised linear classifier is used for implementing the binary clustering which partitions a group into two subgroups. In Fig. 8.24, the three linear classifiers are used for partitioning a group into the three subgroups. The role of each unsupervised linear classifier is to decide whether each data item belongs to the corresponding cluster or not. The process of applying the linear classifiers to the multiple clustering is viewed as the process of applying supervised linear classifiers to the multiple classification.

Let us make some remarks on the unsupervised version of linear classifier. The linear classifier constructs a linear boundary from the training examples. A single linear classifier is used for the binary clustering of data items which partitions a group into two subgroups. Multiple linear classifiers as many as clusters are applied

to the multiple clustering. In the subsequent sections, the unsupervised version will be applied for modifying the linear classifier into its deep versions.

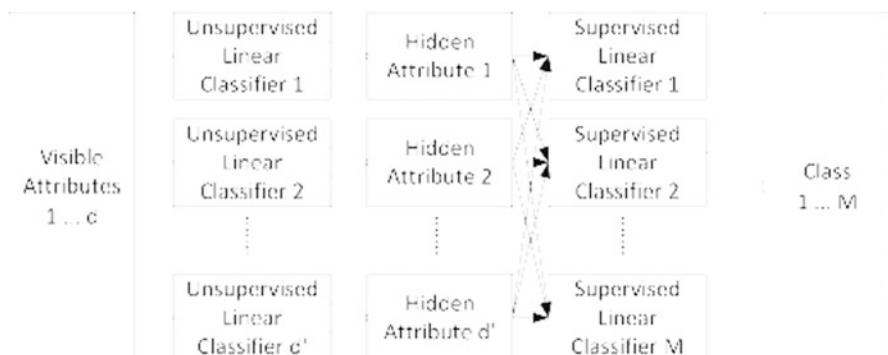
### 8.4.2 Stacked Linear Classifier

This section is concerned with the stacked linear classifier as the deep version of linear classifier. In the previous section, we modified the linear classifier into its unsupervised version. The unsupervised version is used for implementing the deep version of linear classifier in this section. The hidden vector which is computed by the unsupervised version consists of cluster identifiers as its elements. This section is intended to describe the deep version of linear classifier which combines its supervised version and its unsupervised with each other.

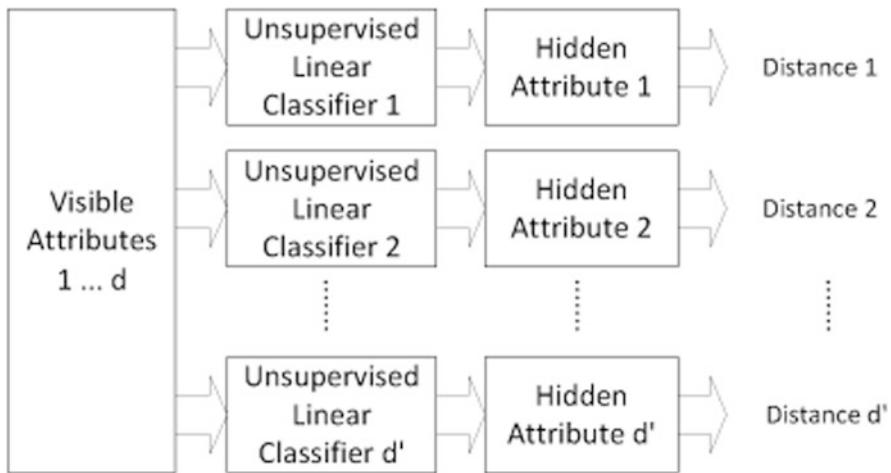
The deep architecture of the linear classifier is illustrated in Fig. 8.25. It is assumed that the input vector and the hidden vector are a  $d$  dimensional vector and a  $d'$  dimensional vector, respectively. The  $d'$  unsupervised learning classifiers are involved for computing the hidden vector. The role of  $M$  supervised linear classifiers in the right position is to classify the hidden vector into one among the  $M$  classes. Each element of the hidden vector indicates a cluster identifier.

The process of computing the hidden values from the input values is illustrated in Fig. 8.26. The  $d'$  unsupervised linear classifiers are involved in computing the hidden values. They are discriminated with their different subsets of training examples and their different initial linear boundaries. The distance from the linear boundary is used as a hidden value; if a data item locates above the linear boundary, the distance from it is given as a positive value, and otherwise, the value is given as a negative value. The element in the hidden vector is given as a continuous real value.

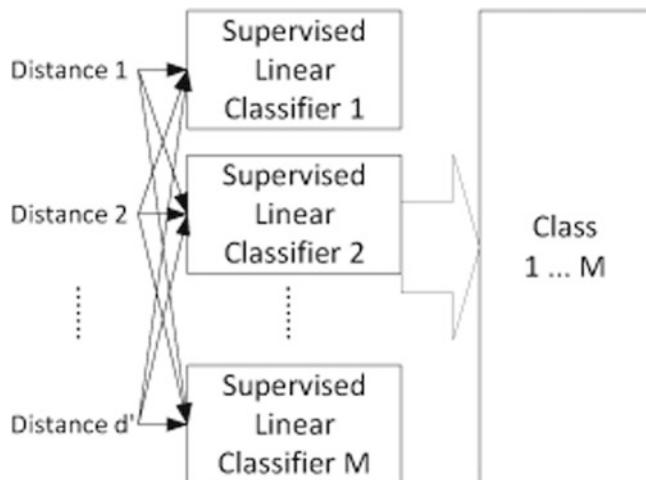
The process of computing the output values from the hidden values is illustrated in Fig. 8.27. The  $d''$  hidden values, each of which is a distance from the linear bound-



**Fig. 8.25** Architecture of deep linear classifier



**Fig. 8.26** Hidden variable computation in deep linear classifier



**Fig. 8.27** Classification of hidden values by deep linear classifier

ary, are computed. The  $M$  supervised linear classifiers are involved in computing the output values, and the given task is assumed as the multiple classification. The role of each linear classifier is to classify the hidden vector into whether it belongs to the corresponding category or not. This version of linear classifiers is viewed as a deep learning algorithm, in that the output values are computed from the hidden values.

Let us make some remarks on the stacked linear classifiers as the deep learning algorithm. The  $d'$  unsupervised linear classifiers and the  $M$  supervised learning classifiers are involved in implementing the deep learning algorithm. Each element in the hidden vector which is computed by the unsupervised linear classifiers

is a distance from its corresponding linear classification boundary. The multiple classification is decomposed into binary classifications, and output values are computed from the hidden values by the  $M$  supervised linear classifiers. In the subsequent sections, the deep learning algorithm will be upgraded by replacing the linear classifier by the SVM.

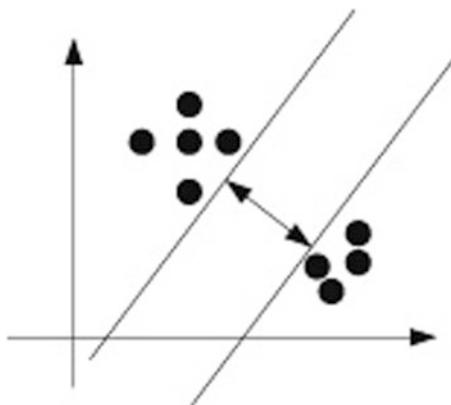
### 8.4.3 Unsupervised SVM

This section is concerned with the unsupervised version of SVM. In Sect. 8.4.1, we studied the unsupervised version of the linear classifier. In this section, we expand what is studied in Sect. 8.4.1 into the unsupervised SVM which defines dual hyperplanes in the mapped space. A single SVM as a binary classifier is for separating data items into two clusters; multiple SVMs are used for implementing the multiple clustering. This section is intended to describe the unsupervised version of SVM as an approach to the binary clustering.

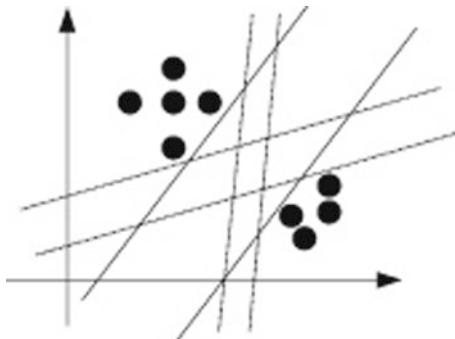
The dual linear classifiers which represent the unsupervised SVM are illustrated in Fig. 8.28. If it is assumed that the group of data items consists of two subgroups, there are an infinite number of linear boundaries for separating the group. If we define the two parallel linear boundaries, there is only one pair of parallel boundaries with the maximal margin. The maximal margin between the two parallel linear boundaries is the additional requirement for using the SVM for the binary clustering. The Lagrange multipliers should be optimized in the SVM, instead of the weights.

The process of optimizing the Lagrange multipliers of SVM is illustrated in Fig. 8.29. The weights are optimized in the binary clustering by a linear classifier. However, in using the SVM to the binary clustering, the Lagrange multipliers are optimized. The clustering index is computed to the two clusters, the data items which are harmful to the clustering are selected, and the Lagrange multipliers are

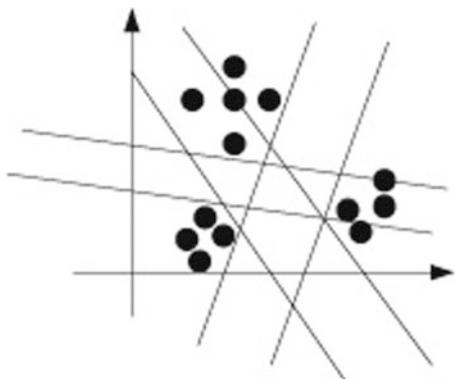
**Fig. 8.28** Dual hyperplanes in unsupervised SVM



**Fig. 8.29** Optimization of dual hyperplanes in unsupervised SVM



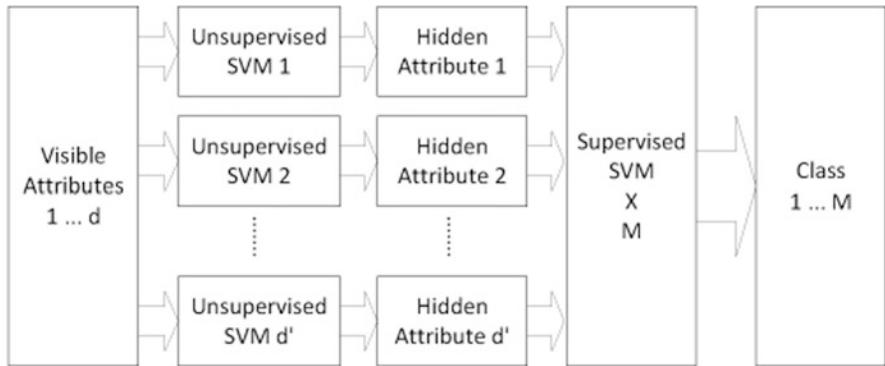
**Fig. 8.30** Decomposition of multiple clustering task into binary clustering tasks for applying unsupervised SVM



updated based on the selected data items. The data items which are selected are viewed as the misclassified training examples.

The application of the multiple SVMs to the multiple clustering task is illustrated in Fig. 8.30. A single SVM is viewed as a binary classifier in the supervised learning or an approach to the binary clustering in the unsupervised learning. It is decomposed into multiple binary clustering tasks as many as clusters, in applying the SVM to the multiple clustering task. The role of each SVM is to separate a single group into one which corresponds to the cluster and the remaining. The  $M$  unsupervised SVMs are involved in segmenting a group of data items into  $M$  clusters.

Let us make some remarks on the unsupervised version of SVM as the approach to the binary clustering. The dual parallel hyperplanes are constructed for separating a group into two subgroups in the unsupervised SVM. The Lagrange multipliers are optimized for searching for the dual parallel hyperplanes with the maximal margin. The multiple clustering task is decomposed into multiple binary clustering tasks, and the unsupervised SVM is applied to each of them. The slack variables are considered for improving the flexibility in constructing the dual parallel hyperplanes.



**Fig. 8.31** Architecture of deep SVM with unsupervised learning layer

#### 8.4.4 Stacked SVM

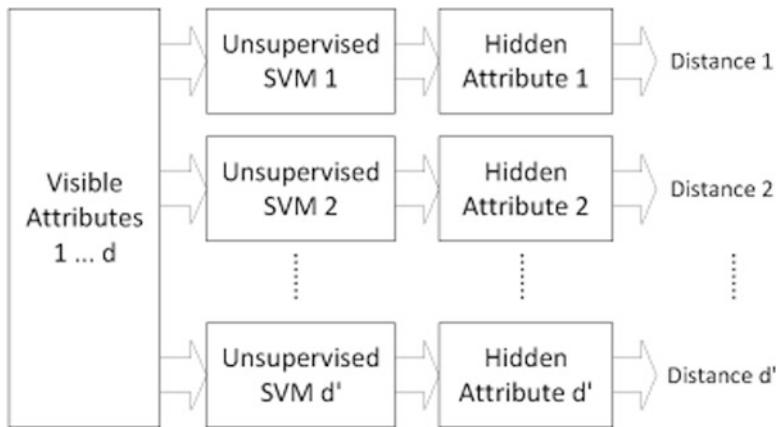
This section is concerned with the stacked SVM as a deep version of SVM. In Sect. 8.4.2, we studied the stacked linear classifier as a deep linear classifier. In this section, the linear classifier is replaced by the SVM which defines the dual parallel hyperplanes. In implementing the deep version, multiple unsupervised SVMs and a single supervised SVM are involved in implementing the deep version. This section is intended to describe the deep version of SVM which is called stacked SVMs.

The deep version of SVM which consists of its supervised version and its unsupervised versions is illustrated in Fig. 8.31. The  $d'$  unsupervised SVMs are involved in computing the hidden vector from the input vector. The role of each unsupervised SVM is to carry out the hard clustering for computing an element of the hidden vector. The hidden vector which is computed by the  $d'$  unsupervised SVMs is classified into one among the  $M$  classes by the  $M$  supervised SVMs. The multiple classification is decomposed into binary classification for applying the supervised SVMs.

The unsupervised layer in the deep version of SVM is illustrated in Fig. 8.32. The output value of the SVM to the training example,  $\mathbf{x}_i$ , is denoted by  $d_i$ , and the output value is expressed as Eq. (8.31):

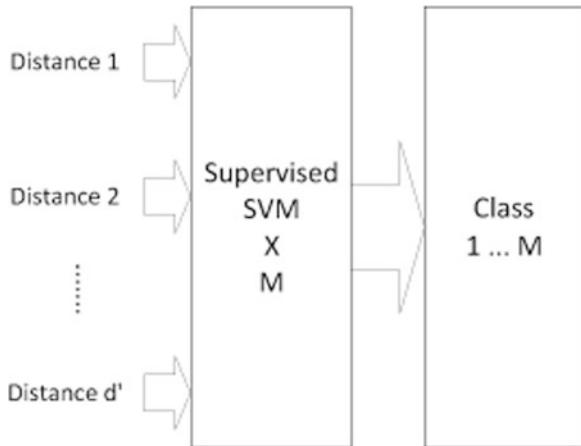
$$d_i = \begin{cases} 1 & \text{if } \sum_{j=1}^N \alpha_j K(\mathbf{x}_j, \mathbf{x}_i) - b \geq 1 \\ -1 & \text{if } \sum_{j=1}^N \alpha_j K(\mathbf{x}_j, \mathbf{x}_i) - b \leq -1 \end{cases}. \quad (8.31)$$

The distance which is an element of the hidden vector is computed by Eq. (8.32):



**Fig. 8.32** Unsupervised SVMs for computing hidden values

**Fig. 8.33** Supervised SVMs for multiple classification



$$h_i = \left( \sum_{j=1}^N \alpha_{ij} K(\mathbf{x}, \mathbf{x}_j) - b \right) - d, \quad (8.32)$$

where  $\alpha_{ij}$  is the Lagrange multiplier in the  $i$ th unsupervised SVM to the  $j$ th training example. The hidden vector with the  $d'$  dimension consists of distances which are computed by Eq. (8.32). The unsupervised SVM is applied to the hidden vector which is computed in the unsupervised layer.

The supervised layer which classifies a novice data item is illustrated in Fig. 8.33. The given task is assumed as the multiple classification which classifies a data item into one among the  $M$  categories, and the task is decomposed into  $M$  binary classification. The  $M$  SVMs, each of which is a binary classifier, are involved in carrying out the multiple classification. Each SVM classifies a novice data

item into belonging to the corresponding category or not. Each unsupervised SVM corresponds to an element in the hidden vector, and each supervised SVM corresponds to a category.

Let us make some remarks on the stacked SVM as a deep version. The version is composed of the unsupervised layer for computing the hidden vector and the supervised layer for classifying the hidden vector. Each element in the hidden vector is a distance from the parallel line which is computed by Eq. (8.32). If the task is the multiple classification, the SVMs as many as categories are involved in the supervised layer. If the dimension of hidden vector is less than that of input vector, the process of computing the hidden vector in the unsupervised layer is viewed as the dimension reduction.

## 8.5 Summary and Further Discussions

Let us summarize what is studied in this chapter. The linear classifier defines a single hyperplane as its classification boundary, whereas the SVM defines the dual parallel hyperplanes with their maximal margins, pursuing the highest stability. The SVM is mentioned as a deep learning algorithm by itself and expanded into the pooled SVM and the multiple kernel-based version. The linear classifier and the SVM are stacked by combining their supervised versions and their unsupervised versions with each other. This section is intended to discuss further what is studied in this chapter.

The kernel matrix is constructed by using the kernel function. It is a square matrix where each row and each column correspond to an individual training example. An element in the kernel matrix is an output value which is given as a scalar value and computed by the kernel function of corresponding vectors. The kernel matrix is characterized as a positively semi-definite one; all elements in the kernel matrix are greater than or equal to the zero. The similarities of all possible pairs of training examples in the mapped space are displayed in the kernel matrix.

Let us consider the multiple mapping of an input vector into one in another space. In the traditional SVM, it is assumed that an input vector is mapped only one time into one in another space. In training the SVM, it is possible to map an input vector, multiple times. The kernel function indicates the inner product value of two vectors in the mapped space; even if an input vector is mapped many times, single scalar value is still the output of the kernel function. The inner product of hidden vectors is viewed as the output value of the kernel function of two original input vectors.

Let us consider attaching the convolution layer to the linear classifier. It defines a single hyperplane as its classification boundary using the training set. Each training example is mapped into another vector, using the convolution, and a single hyperplane is constructed as the classification boundary in the mapped space. If multiple filter vectors are defined, multiple linear classifiers each of which corresponds to the convolution channel are constructed. The convolutional linear classifier will be expanded into the convolutional neural networks, and we study the machine learning algorithm in detail, in Chap. 12.

Let us consider attaching the convolution layer to the SVM. It constructs the dual parallel hyperplanes with their maximal margin from the training examples in the mapped space. In the convolutional SVM, the dual parallel hyperplanes are constructed on the space which are mapped by the convolution. If multiple convolution channels are used, multiple SVMs are constructed as a committee. We consider deriving the SVM version which processes a matrix or a tensor like the convolutional neural networks.

## **Part III**

# **Deep Neural Networks**

This part is concerned with the neural networks with their deep architectures. The neural networks are the kind of machine learning algorithms which simulate the nervous system. It is composed with neurons which are connected with each other, and each connection is called synaptic weight. The process of training the neural networks is to optimize weight values which are associated with their own connections among neurons, for minimizing the error on training examples. This part is intended to describe some neural networks, such as MLP (multiple layer Perceptron), the RNN (recurrent neural networks), and the RBM (restricted Boltzmann machine).

This part consists of the four chapters. In Chap. 9, we will study the MLP which is expanded from the Perceptron as a deep learning algorithm. In Chap. 10, we will study the RNN with the feedback connection from a neuron as a typical deep learning algorithm. In Chap. 11, we will study RBM which deals with visible variables and hidden variables. In Chap. 12, we will study the CNN (convolutional neural networks) as the MLP or the Perceptron which is modified into its deep version by adding convolutional and pooling layers.

# Chapter 9

## Multiple Layer Perceptron



This chapter is concerned with the MLP as a typical deep neural networks model. The Perceptron which is the basis for inventing the MLP was invented by Rosenblatt in the 1950s. In the architecture of MLP, there are three layers: the input layer, the hidden layer, and the output layer. A layer is connected to its next layer with the feedforward direction, and the weights are updated in its learning process in the backward direction. This chapter is intended to describe the MLP with respect to the architecture, the computation process, and the learning process.

This chapter is composed of five sections, and in Sect. 9.1, we overview what is studied in this chapter. In Sect. 9.2, we study the Perceptron which was invented before MLP in detail. In Sect. 9.3, we study the architecture and the computation process of MLP. In Sect. 9.4, we study the learning process of the MLP. In Sect. 9.5, we summarize the entire contents of this chapter and discuss further on what is studied in this chapter.

### 9.1 Introduction

This section is concerned with the overview of MLP as a kind of deep learning. The Perceptron was invented in 1950s by Rosenblatt as the basis for the MLP which is covered in this chapter. In the architecture of MLP, there are three layers, the input layer, the hidden layer, and the output layer. The output values are computed from the input layer to the output layer in the feedforward direction, and the weights are optimized as its learning process in the backward direction which is opposite to the feedforward one. This section is intended to overview the MLP as its introduction.

Let us mention the Perceptron as the basis for understanding the MLP. The Perceptron was invented by Rosenblatt in 1951. There are two layers in its architecture: the input layer and the output layer. Its learning process is to update the weights between the two layers. It was criticized by Papert and Minsky in the 1960s, because it has no ability to solve the XOR problem.

Let us mention the MLP which is mainly covered in this chapter as a deep learning algorithm. The neural networks were invented by Rumelhart in 1982, and its significance is to revive the research on neural networks from its dark age. There are the three layers in the architecture: the input layer, the hidden layer, and the output layer; one more layer is added between the input layer and the output layer. The neural networks are applicable to any kind of regression task, as well as the classification task. Its advantage is its excellent generalization performance by its higher tolerance to noises, but its demerit is too much time taken for training it.

Let us mention some MLP variants as the improved version. The typical MLP variant is the version where its weights are optimized by adding the momentum term to the gradient descent. If the MLP is applied to the time series prediction, it receives the window which slides on the time series data as its input; the version of MLP is called TDNN (time delay neural networks). The CNN (convolutional neural network) which is covered in Chap. 12 belongs to the MLP variant, in that the feature extraction part is added to the MLP. Other MLP variants may be derived by modifying the learning process and the architecture in the standard MLP.

Let us mention what is intended in this chapter. We review the Perceptron as the earlier neural networks model as the preparation for studying the MLP. This chapter provides the understanding of process of classifying a data item by the MLP. We understand the process of training the MLP with the training examples. This chapter is intended to understand the MLP as a deep learning algorithm.

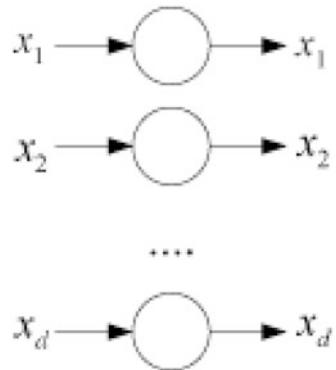
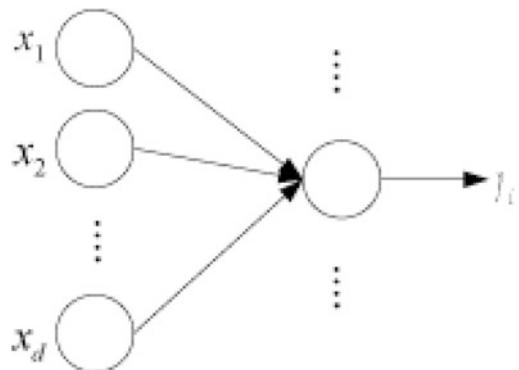
## 9.2 Perceptron

This section is concerned with the early feedforward neural networks, Perceptron. In Sect. 9.2.1, we describe the architecture. In Sect. 9.2.2, we describe the process of classifying a novice data item. In Sect. 9.2.3, we describe the process of updating the weights between the two layers as the learning process. In Sect. 9.2.4, we describe the application of the Perceptron to the regression task.

### 9.2.1 Architecture

This section is concerned with the architecture of the Perceptron. It was invented by Rosenblatt in the 1950s. There are two layers in the Perceptron: the input layer and the output layer. It was expanded into the MLP in 1982 by Rumelhart by adding the hidden layer. This section is intended to describe the two layers in the architecture.

The input layer of the Perceptron is illustrated in Fig. 9.1. The initial input vector is assumed as a  $d$  dimensional vector,  $\mathbf{x} = [x_1 \ x_2 \ \dots \ x_d]$ . The role of input layer is to transfer the input vector to the next layer by itself. The  $i$ th input node is expressed as Eq. (9.1):

**Fig. 9.1** Input layer**Fig. 9.2** Output layer

$$I_i = x_i \quad (9.1)$$

Because the values in the input nodes are the input values by themselves, the notation of the input node,  $I_i$ , is omitted; the input value is notated directly by  $x_i$ .

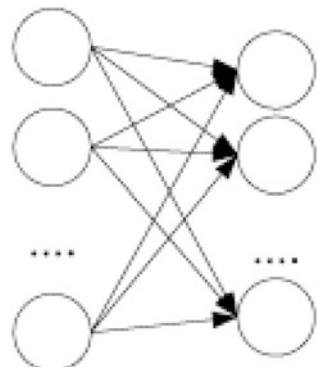
The output layer of the Perceptron is illustrated in Fig. 9.2. Each output node is connected with all the input nodes; it is assumed that the connection between the two layers is complete. The connection from the input node,  $x_i$ , to the output node,  $y_j$ , is associated with the weight,  $w_{ji}$ . If the number of input nodes is  $d$ , the net input is expressed by Eq. (9.2):

$$net_j = \sum_{i=1}^d w_{ji} x_i. \quad (9.2)$$

The function is applied for computing the output node value as shown in Eq. (9.3):

$$y_j = F(net_j). \quad (9.3)$$

**Fig. 9.3** Feedforward structure



The connection between the input layer and the output layer in the Perceptron, called feedforward connection, is illustrated in Fig. 9.3. The feedforward connection is the connection from one node to another different node for transferring the output value to another node. Each node in the input layer is connected with all nodes in the output layer; the complete feedforward connection is made between the two layers. The Perceptron and the MLP which are covered in this chapter are assumed as the complete feedforward connection between the two layers. The connection from one node to the same node is called recurrent connection and covered in the next chapter.

Let us make some remarks on the architecture of the Perceptron which consists of the input layer and the output layer. The role of the input layer is to transfer the input values to the output nodes. The output node value is computed by applying the activation function to the net input. The connection between the two layers is associated with its own weight and is called feedforward connection. The learning process of the Perceptron is to optimize the weights which are associated with the connections.

### 9.2.2 Classification Process

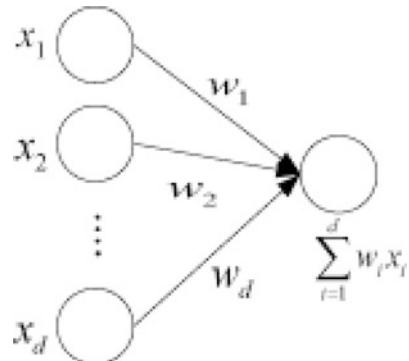
This section is concerned with the process of computing the output node values. In the previous section, we studied the architecture of Perceptron which consists of the two layers: the input layer and the output layer. In this section, we study the process of computing the output node values from the input values. In computing output values, it is assumed that the connection between the two layers is complete, and the weights which are associated with them are already optimized. This section is intended to describe the process of computing the output values as the generalization of Perceptron.

Let us mention the input vector which is given to the input layer as shown in Fig. 9.4. It is assumed that a raw data item is encoded into a  $d$  dimensional numerical vector. An input vector is notated by  $\mathbf{x} = [x_1 \ x_2 \ \dots \ x_d]$ , and each element

**Fig. 9.4** Input vector preparation



**Fig. 9.5** Net input computation



corresponds to its own node of the input layer. The number of input nodes is the dimension of the input vector which represents a raw data. The value of each input node is an element in the input vector.

The net input to the output node is illustrated in Fig. 9.5. To the  $d$  nodes in the input layer, the  $d$  dimensional vector,  $\mathbf{x} = [x_1 \ x_2 \ \dots \ x_d]$ , is given. All input nodes are connected to a single output node, and their connections are associated with the weights,  $w_1, w_2, \dots, w_d$ . The net input is computed by Eq. (9.4):

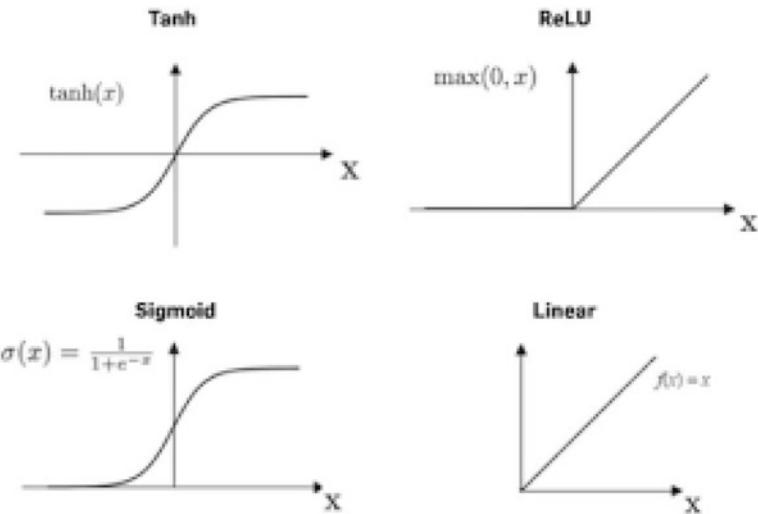
$$net = \sum_{i=1}^d w_i \cdot x_i \quad (9.4)$$

and if the weight is viewed as a vector,  $\mathbf{w}$ , Eq. (9.4) is transformed into the inner product of the input vector and the weight vector, as shown in Eq. (9.5):

$$net = \mathbf{w} \cdot \mathbf{x} \quad (9.5)$$

The net input is computed from the input nodes as the integration of multiple inputs into a single input.

The four representative activation functions are illustrated in Fig. 9.6. The net input which is the integrated input to the output node is computed by the above process. The output node value is computed by applying an activation function to the net input, as expressed in Eq. (9.6):



**Fig. 9.6** Activation functions from <https://docs.paperspace.com/machine-learning/wiki/activation-function>

$$y = F(\text{net}) = F \left( \sum_{i=1}^d w_i \cdot x_i \right). \quad (9.6)$$

If the linear function is adopted as the activation function, the net input becomes the output as shown in Eq. (9.7):

$$y = \text{net} = \sum_{i=1}^d w_i \cdot x_i. \quad (9.7)$$

The linear function is usually adopted in implementing the Perceptron.

Let us make some remarks on the process of computing the output values in the Perceptron. The input is provided as a  $d$  dimensional numerical vector for the Perceptron whose weights are optimized. The net input is the integration of the input values by the product of each of them and its own weight. The output value is computed by applying an activation function to the net input. We consider various versions of Perceptron, depending on its applied activation function.

### 9.2.3 Learning Process

This section is concerned with the learning process of Perceptron. In the previous section, we studied the process of computing the output values in the Perceptron,

under the assumption of the optimized weights. The learning process of the Perceptron is to optimize the weights between the input layer and the output layer. There are two kinds of output values, target output values which are initially given to the training examples and computed output values which are computed by the process which is described in the previous section, and the direction of optimizing weights is to minimize the error between the two kinds of output values. This section is intended to describe the process of optimizing the weights, as the learning process.

Before covering the learning process, we need to define the loss function which indicates the error between the target output and the computed output. The number of output nodes is  $c$ ; the output vector is assumed as a  $c$  dimensional vector. The target output vector and the computed output vector are notated by  $\mathbf{y} = [y_1 \ y_2 \ \dots \ y_c]$  and  $\hat{\mathbf{y}} = [\hat{y}_1 \ \hat{y}_2 \ \dots \ \hat{y}_c]$ , respectively. The loss function is defined as the difference between the two kinds of output vector as shown in Eq. (9.8):

$$L = \frac{1}{2} \sum_{i=1}^c (y_i - \hat{y}_i)^2. \quad (9.8)$$

The loss function which is expressed by Eq. (9.8) is the basis for defining the weight update rule.

Let us mention the rule of updating the weights between the input layer and the output layer. The weights between the two layers are defined as a matrix as shown in Eq. (9.9):

$$\mathbf{W} = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1d} \\ w_{21} & w_{22} & \dots & w_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ w_{c1} & w_{c2} & \dots & w_{cd} \end{pmatrix}. \quad (9.9)$$

The relation of the weight matrix,  $\mathbf{W}$ , with the loss function which is defined as Eq. (9.8) is considered for optimizing the weight vector. The weight update rule is defined by Eq. (9.10), based on the gradient descent:

$$\mathbf{W}(t+1) = \mathbf{W}(t) - \eta \frac{\partial L}{\partial \mathbf{W}}. \quad (9.10)$$

where  $\eta$  is the learning rate, which is given as the external parameter. The specific equation for optimizing the elements of the weight matrix gradually will be derived based on Eq. (9.10).

Let us derive a specific equation from Eq. (9.10) which represents the frame of updating the weights. The computed output,  $\hat{y}_i$ , is expressed by Eq. (9.11),

$$\hat{y}_i = \sum_{j=1}^d w_{ij} \cdot x_j. \quad (9.11)$$

The loss function is differentiated by the weights by the chain rule which is expressed as Eq. (9.12), into Eq. (9.13):

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial w_{ij}} \quad (9.12)$$

$$\frac{\partial L}{\partial w_{ij}} = -(y_i - \hat{y}_i)x_j. \quad (9.13)$$

Equation (9.14) for updating the weights is derived as follows:

$$w_{ij} \leftarrow w_{ij} + \eta(y_i - \hat{y}_i)x_j \quad (9.14)$$

The weights are updated depending on the difference between the target output and the computed output as shown in Eq. (9.14).

Let us make some remarks on the learning process of the Perceptron. The square of error between the target output and the computed output is defined as the loss function. The gradient descent between the weights and the error is set as the direction of optimizing the weights. Equation (2.12) is derived as the rule of updating the weights, based on the difference between the target output and the computed output. The possibility of falling into a local minimum is the demerit of optimizing the weights depending on the gradient descent.

#### 9.2.4 Perceptron for Regression

This section is concerned with the application of the Perceptron to the regression. In the previous section, it is assumed that the task to which the Perceptron is applied is the classification. In this section, the Perceptron is applied to the regression which estimates a continuous value from several factors. The difference between the classification and the regression is that the output value is continuous in the regression, whereas the output value is discrete in the classification. This section is intended to describe the application of the Perceptron to the regression.

Let us mention the regression to which the Perceptron is applied in the functional view. The regression is defined as the process of estimating a continuous value or values by analyzing more than one input value. A regression is modeled as an equation, and the process of doing so is called regression modeling. The nonlinear function approximation and the time series prediction are typical examples of regression. The traditional regression models were replaced by the neural networks or the deep learning algorithms.

Let us mention the direct application of the Perceptron to the regression task. It is assumed that the given task is a univariate regression where only one continuous value is estimated. In designing the Perceptron, the number of input nodes is the number of input factors, and the number of output nodes is one. The regression

mode which is defined by training the Perceptron is viewed as a linear combination of input values. The direction of training the Perceptron is to minimize the error on the training examples, rather than to eliminate it completely.

Let us mention the process of applying the Perceptron to the classification which is mapped from the regression. The task is mapped into a classification by discretizing the continuous output value; a discrete value indicates an identifier of a particular value interval. The number of output nodes is the number of value intervals which are defined by discretizing the output value. For each training example, its target continuous output value is replaced by a category which corresponds to its value interval. In applying the Perceptron to the regression, we choose the direct application to the regression or the indirect application to it by mapping it into a classification.

Let us make some remarks on the process of applying the Perceptron to the regression task. The regression is the process of estimating a continuous output value or values by analyzing more than one input value. If the Perceptron is applied directly to the regression, the regression model is viewed as a linear combination of the input values. The Perceptron is applied to the regression by mapping it into a classification. We may consider applying multiple Perceptrons to a nonlinear problem.

## 9.3 Multiple Layer Perceptrons

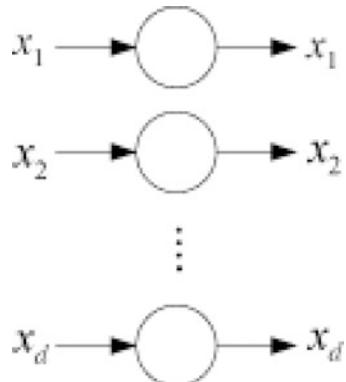
This section is concerned with the computation process in the MLP. In Sect. 9.3.1, we view the architecture of MLP with the three layers. In Sect. 9.3.2, we define the notations which are involved in understanding the MLP and describe the computation of input nodes. In Sect. 9.3.3, we describe the process of computing the hidden nodes. In Sect. 9.3.4, we describe the process of computing the output nodes.

### 9.3.1 *Architecture*

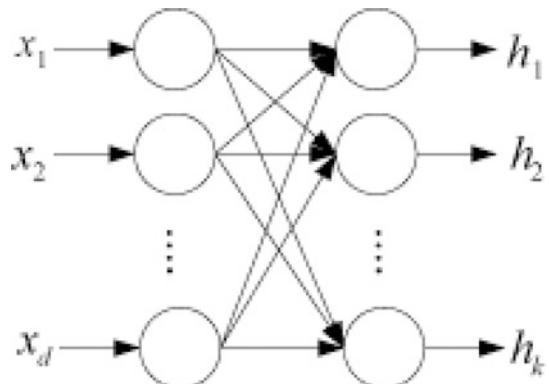
This section is concerned with the architecture of the MLP which is expanded from the Perceptron. In the previous section, we studied the Perceptron whose architecture consists of the input layer and the output layer. In the MLP, one more layer which is called hidden layer is added between the input layer and the output layer. The regression mode which is defined by training the MLP is viewed as a nonlinear model as the effect of adding the hidden layer. This section is intended to describe the MLP architecture which consists of the three layers.

The input layer in the MLP is illustrated in Fig. 9.7. The role of the input layer is the same to its role in the Perceptron. The input vector with the  $d$  dimensionality is

**Fig. 9.7** Input layer of multiple layer Perceptron



**Fig. 9.8** Hidden layer of multiple layer Perceptron



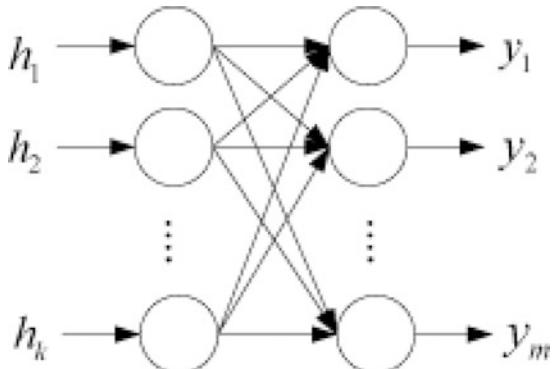
notated by  $\mathbf{x} = [x_1 \ x_2 \ \dots \ x_d]$ . The value of each input node is notated by  $x_i$ . The input values are used for computing the hidden values.

The hidden layer which is connected from the input layer is illustrated in Fig. 9.8. An arbitrary number of hidden nodes is determined, and the number is notated by  $k$ . The hidden vector with the  $k$  dimensionality is notated by  $\mathbf{h} = [h_1 \ h_2 \ \dots \ h_k]$ . The connection between the input layer and the hidden layer is complete; the number of weights becomes  $k \times d$ . The hidden node values are used for computing the output node values.

The complete connection between the hidden layer and the output layer is illustrated in Fig. 9.9. The architecture which is presented in Figure Prev is the left part of the MLP architecture, and the architecture which is presented in Fig. 9.9 is the right part of the MLP architecture. The target output vector and the computed vector are notated by  $\mathbf{y} = [y_1 \ y_2 \ \dots \ y_m]$  and  $\hat{\mathbf{y}} = [\hat{y}_1 \ \hat{y}_2 \ \dots \ \hat{y}_m]$ , respectively, and the  $m$  dimensional vectors. The connection between the hidden layer and the output layer is complete; the number of connections becomes. The total number of connections in the MLP with the three layers is  $(d \times k) + (k \times m)$ .

Let us make some remarks on the architecture of the MLP with the three layers. The role of input layer is to receive the input vector by itself. The number of

**Fig. 9.9** Output layer of multiple layer Perceptron



hidden nodes is set arbitrary, and their values are computed by the input values. The connection from the hidden layer to the output layer is complete; each output node value depends on the hidden node values. There are two ways of expanding the MLP further: increasing the number of hidden nodes and adding more hidden layers.

### 9.3.2 Input Layer

This section is concerned with the input layer of MLP. It is composed of the three layers: input layer, hidden layer, and output layer. The role of input layer is to receive the input vector. Each input node value is an element of the input vector. This section is intended to describe the input layer in the MLP with respect to the design.

In designing the MLP, let us consider the number of input nodes. The input layer corresponds to the input vector which represents a data item. The number of input nodes is the dimension of the input vector. If the dimension of input vector is  $d$ , the number of input nodes becomes  $d$ . The value of each input node is an element of the input vector.

Let us mention some notations which are involved in the input layer. The number of input nodes is  $d$  as mentioned above. A novice input vector is notated by  $\mathbf{x} = [x_1 \ x_2 \ \dots \ x_d]$ , and each input node is notated,  $x_i$ . The number of training examples is  $N$ , and the training set is notated by  $Tr = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$ . The notations are used for explaining the generalization and the learning of MLP in the subsequent sections.

Let us express the equation of the input node. The input layer is connected to input vector, directly. The node value is expressed by applying the activation function to the net input. In the input node, the activation function is given as the linear function, and the net input is the input value by itself as expressed in Eq. (9.15):

$$net_i = x_i \quad (9.15)$$

The input node value is given as  $x_i$ .

Let us make some remarks on the input layer of the MLP. The number of input nodes is the dimension of the input vector. The elements in the input vector correspond to the input nodes. Its value is computed by applying the linear function to the input value which is given as the net input. We may consider applying the sigmoid function or the hyperbolic function as an activation function to the input value.

### 9.3.3 Hidden Layer

This section is concerned with the hidden layer in the MLP. The hidden layer which distinguishes the MLP from the Perceptron is intended to solve nonlinear problems. It is assumed that the connection between the input layer and the hidden layer is complete, and each hidden node value is computed by applying an activation function to the net input of all input node values. The hidden vector which is transformed from the input vector is the basis for understanding the deep learning. This section is intended to describe the process of computing the hidden node values as a vector.

Let us mention the determination of the number of hidden nodes in designing the MLP. The number of hidden nodes is set arbitrary in the reality. The simplicity and the underfitting are caused by the small number of hidden nodes, whereas the ability to solve nonlinear problems and the overfitting are caused by the large number of hidden nodes. The validation set which is separated from the training set is used for optimizing the number of hidden nodes. Refer to [1] for studying the validation set in detail.

Let us mention some notations which are involved in the hidden layer. The number of hidden nodes is  $k$ ; an arbitrary number of hidden nodes is set. The hidden vector is notated by  $\mathbf{h} = [h_1 \ h_2 \ \dots \ h_k]$ , and each hidden node is notated by  $h_i$ . The hidden vector is interpreted into one which is mapped from the input vector with its different dimension. The hidden vector is classified directly, instead of the input vector, in the MLP.

Let us mention the process of computing the hidden values in the MLP. The weights between the input layer and the hidden layer are given as a matrix as shown in Eq. (9.16):

$$\mathbf{W}^1 = \begin{pmatrix} w_{11}^1 & w_{12}^1 & \dots & w_{1d}^1 \\ w_{21}^1 & w_{22}^1 & \dots & w_{2d}^1 \\ \vdots & \vdots & \ddots & \vdots \\ w_{k1}^1 & w_{k2}^1 & \dots & w_{kd}^1 \end{pmatrix}. \quad (9.16)$$

The hidden value,  $h_i$ , is computed by Eq. (9.17):

$$h_i = \text{sigmoid}\left(\sum_{j=1}^d w_{ji}x_j\right). \quad (9.17)$$

The computation of the hidden vector,  $\mathbf{h}$ , is expressed as Eq. (9.18):

$$\mathbf{h} = \text{sigmoid}(\mathbf{W}^{1T} \cdot \mathbf{x}). \quad (9.18)$$

The hidden vector is used for computing the output vector.

Let us make some remarks on the process of computing the hidden node values. The number of hidden nodes is set arbitrary in designing the MLP. The input node values are given as a  $d$  dimensional vector, and the hidden node values are given as a  $k$  dimensional vector. The weights between the hidden layer and the input layer are given as a  $d \times k$  matrix. The sigmoid function is adopted as the activation function in the MLP.

### 9.3.4 Output Layer

This section is concerned with the process of computing the output values in the MLP. It is assumed to have the three layers, the input layer, the hidden layer, and the output layer, and we studied the process of computing a hidden vector in the previous section. In this section, we study the process of computing the output vector from the hidden vector as the next step. On more kind of weights between the hidden layer and the output layer exists in addition to those between the input layer and the hidden layer. This section is intended to describe the process of computing the output vector with the hidden vector.

Let us consider the number of output nodes in designing the MLP. The number of input nodes and hidden nodes is set by the dimension of the input vector and an arbitrary number, respectively. If the MLP is applied to a classification task, the number of output nodes is set by the number of the predefined categories. If the given problem is a regression, the number of output nodes is set by the number of predicted variables. If the regression is univariate, the number of output node is one.

Let us define the notation which is involved in the output layer in the MLP. The number of hidden nodes and the hidden vector was previously notated by  $k$  and  $\mathbf{h} = [h_1 \ h_2 \ \dots \ h_k]$ , respectively. It is assumed that the MLP is applied to the classification task, and the number of output nodes is  $m$  as the number of predefined categories. The two kinds of output vectors, the target output vector and the computed output vector, are notated by  $\mathbf{y} = [y_1 \ y_2 \ \dots \ y_m]$  and  $\hat{\mathbf{y}} = [\hat{y}_1 \ \hat{y}_2 \ \dots \ \hat{y}_m]$ . We consider the weights between the hidden layer and the output layer for computing the output vector.

Let us mention the process of computing the output values in the MLP. The weights between the hidden layer and the output layer are given as a matrix as shown in Eq. (9.19):

$$\mathbf{W}^2 = \begin{pmatrix} w_{11}^2 & w_{12}^2 & \dots & w_{1k}^2 \\ w_{21}^2 & w_{22}^2 & \dots & w_{2k}^2 \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1}^2 & w_{m2}^2 & \dots & w_{mk}^2 \end{pmatrix}. \quad (9.19)$$

The output value,  $\hat{y}_i$ , is computed by Eq. (9.20):

$$\hat{y}_i = \text{sigmoid} \left( \sum_{j=1}^k w_{ij} \cdot h_j \right) \quad (9.20)$$

The computation of the output vector,  $\hat{\mathbf{y}}$ , is expressed as Eq. (9.21):

$$\hat{\mathbf{y}} = \text{sigmoid}(\mathbf{W}^{2T} \cdot \mathbf{x}). \quad (9.21)$$

If the MLP is applied to the regression, the sigmoid function is replaced by the hyperbolic function.

Let us make some remarks on the output layer in the MLP. If it is applied to the classification task, the number of output nodes is set by the number of predefined categories. There are two kinds of output vector: target output vector and computed output vector. One more kind of weights between the hidden layer and the output layer exists for computing the output vector. The fact that the two kinds of weights are optimized is the difference of the MLP from the Perceptron.

## 9.4 Learning Process

This section is concerned with the learning process of MLP called backpropagation. In Sect. 9.4.1, we describe the process of updating the weights between the output layer and the hidden layer. In Sect. 9.4.2, we describe the process of updating the weights between the hidden layer and the input layer. In Sect. 9.4.3, we describe the overall process of training the MLP with the training examples. In Sect. 9.4.4, we describe the advanced schemes of optimizing weights of MLP.

### 9.4.1 Weight Update Between Hidden Layer and Output Layer

This section is concerned with the process of updating the weights between the hidden layer and the output layer. The output values are computed in the forward direction as mentioned in Sect. 9.3. The weights are updated in the backward direction; the fact is the reason of calling the MLP backpropagation. We will derive the equation of updating the weights between the output layer and the hidden layer. This section is intended to describe the process of updating the weights between the output layer and the hidden layer.

The loss function is defined before setting the weight update rule. The direction of training the MLP is to minimize the error between the target output vector,  $\mathbf{y} = [y_1 \ y_2 \ \dots \ y_m]$ , and the computed output vector,  $\hat{\mathbf{y}} = [\hat{y}_1 \ \hat{y}_2 \ \dots \ \hat{y}_m]$ . The loss function which is based on the error is defined as Eq. (9.22):

$$L = \frac{1}{2} \sum_{i=1}^m (y_i - \hat{y}_i)^2 \quad (9.22)$$

We may consider the absolute error,  $\sum_{i=1}^m |y_i - \hat{y}_i|$  as another loss function, but it is not differentiable in defining the weight update rule by the gradient descent. The loss function which is defined in Eq. (9.22) is adopted for deriving the weight update rule.

Let us try to differentiate the loss function for deriving the weight update rule. The frame of updating the weight is expressed as Eq. (9.23):

$$w_{ji}^2 \leftarrow w_{ji}^2 - \eta \frac{\partial L}{\partial w_{ji}^2}. \quad (9.23)$$

The chain rule expressed to the last term in Eq. (9.23) is expressed as Eq. (9.24):

$$\frac{\partial L}{\partial w_{ji}^2} = \frac{\partial L}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial net_j^2} \frac{\partial net_j^2}{\partial w_{ji}^2} \quad (9.24)$$

If the sigmoid function is adopted as the activation function, each term in Eq. (9.24) is expressed as Eqs. (9.25), (9.26), and (9.27):

$$\frac{\partial L}{\partial \hat{y}_j} = (y_j - \hat{y}_j) \quad (9.25)$$

$$\frac{\partial \hat{y}_j}{\partial net_j^2} = \hat{y}_j(1 - \hat{y}_j) \quad (9.26)$$

$$\frac{\partial \text{net}_j^2}{\partial w_{ji}^2} = h_i \quad (9.27)$$

If the linear function is adopted, the second term in Eq.(9.24) is expressed as Eq.(9.28):

$$\frac{\partial \hat{y}_j}{\partial \text{net}_j^2} = 1 \quad (9.28)$$

Let us mention the rule of updating the weights between the hidden layer and the output layer. If the number of hidden nodes is  $k$ , and the number of output nodes is  $m$ , the weights are viewed as a  $m \times k$  matrix as shown in Eq.(9.19). The rule of weight,  $w_{ji}$ , is derived by substituting Eqs. (9.25), (9.26), and (9.27) to Eq. (9.23) into Eq. (9.29):

$$w_{ji} \leftarrow w_{ji} + \eta h_j \hat{y}_i (1 - \hat{y}_i) (y_i - \hat{y}_i) \quad (9.29)$$

If the sigmoid function is replaced by the hyperbolic function, the weight update rule becomes Eq. (9.30):

$$w_{ji} \leftarrow w_{ji} + \eta h_j (1 + \hat{y}_i) (1 - \hat{y}_i) (y_i - \hat{y}_i) \quad (9.30)$$

It is assumed that the activation function is a continuous function which is differentiable, in deriving the rule of updating the weights based on the gradient descent.

Let us make some remarks on the process of updating the weights between the hidden layer and the output layer. It is required to define the loss function for deriving the weight update rule. The gradient descent on the error surface on the weights is the basis for optimizing the weights. If the sigmoid function is adopted as the activation function, Eq. (9.29) is the weight update rule. The learning rate in Eqs. (9.29) and (9.30) should be controlled for updating weights; too high value is caused by the fluctuation, and too low value is caused by the very slow update.

#### 9.4.2 Weight Update Between Input Layer and Hidden Layer

This section is concerned with the process of updating the weights between the input layer and the hidden layer. In the previous section, we studied the process of updating the weights between the hidden layer and the output layer. In this section, we study the process of updating another kind of weights. The weights between the input layer and the hidden layer are influenced by those between the hidden layer and the output layer. This section is intended to describe the process of updating the weights between the input layer and the hidden layer.

Let us consider the spanning of weights between the input layer and the hidden layer. The weight which is connected from the  $i$ th input node and the  $j$ th hidden node is notated by  $w_{ji}^1$ . The hidden node,  $h_i$ , is connected to all output nodes,  $\hat{y}_1, \hat{y}_2, \dots, \hat{y}_m$ , and the weights which are connected from the hidden node,  $h_i$ , are  $w_{i1}^2, w_{i2}^2, \dots, w_{im}^2$ . The weight,  $w_{ji}^1$ , influences on the weights,  $w_{i1}^2, w_{i2}^2, \dots, w_{mi}^2$ ; they should be considered for updating the weight,  $w_{ji}^1$ . The weights,  $w_{i1}^2, w_{i2}^2, \dots, w_{mi}^2$ , are summed in the rule of updating the weight,  $w_{ji}^1$ .

Let us mention the chain rule for deriving the rule of updating the weights between the input layer and the hidden layer. The chain rule is expressed as Eq. (9.31):

$$\frac{\partial L}{\partial w_{ji}^1} = \sum_{k=1}^m \frac{\partial L}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial net_k^2} \frac{\partial net_k^2}{\partial h_i} \frac{\partial h_i}{\partial net_i^1} \frac{\partial net_i^1}{\partial w_{ji}^1} \quad (9.31)$$

If the sigmoid function is adopted as the activation function, the first three terms in Eq. (9.31) is expressed as Eqs. (9.32), (9.33), and (9.34):

$$\sum_{k=1}^m \frac{\partial L}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial net_k^2} = \sum_{k=1}^m (y_k - \hat{y}_k)^2 \quad (9.32)$$

$$\sum_{k=1}^m \frac{\partial \hat{y}_k}{\partial net_k^2} = \sum_{k=1}^m \hat{y}_k(1 - \hat{y}_k) \quad (9.33)$$

$$\sum_{k=1}^m \frac{\partial net_k^2}{\partial h_i} = \sum_{k=1}^m w_{ik}^2. \quad (9.34)$$

The last two terms in Eq. (9.31) are expressed as Eqs. (9.35) and (9.36):

$$\frac{\partial h_i}{\partial net_i^1} = h_i(1 - h_i) \quad (9.35)$$

$$\frac{\partial net_i^1}{\partial w_{ji}^1} = x_j \quad (9.36)$$

The weights which are connected from the hidden node,  $h_i$  to all output nodes,  $w_{i1}^2, w_{i2}^2, \dots, w_{im}^2$ , are considered for updating the weight,  $w_{ji}^1$ , as shown in Eq. (9.34).

The rule of updating the weights between the input layer and the hidden layer is derived. The frame of updating this kind of weight is Eq. (9.37):

$$w_{ji}^1 \leftarrow w_{ji}^1 - \eta \frac{\partial L}{\partial w_{ji}^1}. \quad (9.37)$$

If the sigmoid function is adopted in both layers, the hidden layer and the output layer, the rule of updating the weight,  $w_{ji}^1$ , is expressed as Eq. (9.38), based on the chain rule which is expressed as Eq. (9.31):

$$w_{ji}^1 \leftarrow w_{ji}^1 - \eta \sum_{k=1}^m x_j h_i (1 - h_i) w_{ik}^2 \hat{y}_k (1 - \hat{y}_k) (y_k - \hat{y}_k). \quad (9.38)$$

If the linear function is adopted in both layers, the rule of updating the weight is simplified into Eq. (9.39):

$$w_{ji}^1 \leftarrow w_{ji}^1 - \eta \sum_{k=1}^m x_j w_{ik}^2 (y_k - \hat{y}_k). \quad (9.39)$$

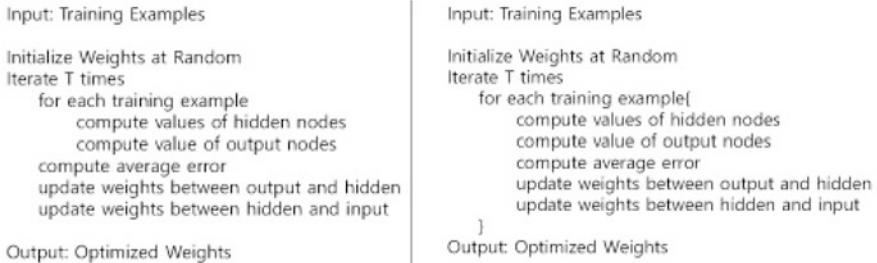
The weights between the input layer and the hidden layer are updated depending on the weights between the hidden layer and the output layer and the difference between the target output and the computed output.

Let us make some remarks on the process of updating weights between the input layer and the output layer. We need to consider the weights which are connected from the hidden node to all output nodes for updating the weight between the input node and the hidden node. The chain rule is applied to differentiate the loss function by the weight. The weight is updated depending on the summation of the difference between the target output and the computed output and the weights between the hidden layer and the output layer. The dynamic learning rate may be considered for training more efficiently the MLP.

### 9.4.3 Entire Learning Process

This section is concerned with the entire process of training the MLP with the training examples. We studied the rules of updating the weights in the MLP in Sects. 9.4.1 and 9.4.2. In this section, we study the entire process of training the MLP based on the weight update rules. The output vector is computed in the feedforward direction, and the weights are updated in the backward direction. This section is intended to describe the entire process of training the MLP with the training examples.

Let us mention the first step for training the MLP. The weights are initialized at random around zero values. The hidden values are computed by the input values, and the output values are computed by the computed hidden values by the processes which are described in Sects. 9.3.2 and 9.3.3. The difference between the target



**Fig. 9.10** Batch learning (left) and interactive learning (right) in MLP

output values and the computed ones on the training examples is called training error. The weights are updated for minimizing the training error.

Let us mention the process of updating the weights in the MLP. We studied the rule of updating the weights in the MLP in Sects. 9.4.1 and 9.4.2. The weights are updated in the backward direction; the weights between the hidden layer and the output layer are updated, and the weights between the input layer and the hidden layer are updated. The difference between the target output vector and the computed one and the weights in the subsequent layers are considered for updating the weights in the current layer. The learning process of MLP is to iterate updating the weights gradually.

The two learning patterns of MLP are illustrated in Fig. 9.10. The left one in Fig. 9.10 belongs to the batch learning, and the right one belongs to the interactive learning. In the batch learning, all training examples are presented, the average over errors on the training examples are computed, and the weights are updated by the rules which are expressed in Eqs. (9.29) and (9.38). In the interactive learning, the weights are updated for each training example. Their performances are comparable with each other and dependent on the application area.

Let us make some remarks on the entire learning process of MLP. The weights are initialized at random around zero, and the output values are computed by the initialized weights. The weights are updated in the backward direction from the output layer to the input layer. The learning process of MLP is to iterate computing the output values with the current weights in the forward direction and updating the weights in the backward direction. We consider some schemes for improving the speed of training the MLP.

#### 9.4.4 Stochastic Gradient Descent

This section is concerned with the stochastic gradient descent for improving the learning speed. In the previous sections, it is assumed that the learning in the rules of updating weights in the MLP is fixed. In this section, we consider the dynamic

learning rate which is variable during training the MLP for improving the learning speed. It is required to observe the amount of reducing the error over the training examples for defining the stochastic gradient descent. This section is intended to describe the modified rule of updating weights for improving the learning speed.

In the previous section, it is assumed that the learning rate is fixed initially between 0.1 and 0.9. The difference between the target output vector and the computed one is the basis for updating the weights. If the surface on the error is rapid, there is the fluctuation risk in fixing the learning rate. If the surface is almost flat, the MLP is trained very slowly. We need to change the learning speed dynamically, observing the change of training error.

Let us consider the change of the learning rate during the learning process, depending on the error change. The current error and the previous error are notated by  $e(t)$  and  $e(t - 1)$ , respectively, and the error change is expressed by Eq. (9.40):

$$\Delta e = e(t) - e(t - 1). \quad (9.40)$$

if  $\Delta e < 0$ , the learning rate,  $\eta$  should be increased, and if  $\Delta e > 0$ , it should be decreased. The frame of updating the weights which is expressed in Eqs. (9.23) and (9.37) is transformed into Eq. (9.41):

$$w_{ji} \leftarrow w_{ji} - \eta(1 - \Delta e \cdot \beta) \frac{\partial L}{\partial w_{ji}}, \quad (9.41)$$

where  $\beta$  is a constant between zero and one for smoothing the error change. This type of stochastic gradient descent is intended to speed up the learning process and prevent the fluctuation.

Let us mention another stochastic gradient descent which is called momentum term. The learning is controlled, depending on the error change, dynamically in the above stochastic descent. In this stochastic gradient descent, the rule of updating the weight change is defined as shown in Eq. (9.42):

$$\Delta w_{ji} \leftarrow \alpha \Delta w_{ji} - \eta \frac{\partial L}{\partial w_{ji}}, \quad (9.42)$$

where  $\alpha$  is the decay rate which is set between zero and 1.0. The weight update rule is expressed into Eq. (9.43):

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji} = w_{ji} - \eta \frac{\partial L}{\partial w_{ji}} + \alpha \Delta w_{ji}. \quad (9.43)$$

if  $\alpha = 0$ , this version is identical to the standard version of MLP.

Let us make some remarks on the stochastic gradient descent which is intended for improving the learning speed. In the version with a fixed learning rate, its limits are that the learning is very slow in the low learning rate and that the training error is not decreased but fluctuated in the high learning rate. The learning should be controlled during the learning process dynamically by observing the training error change. The weight change is also the basis for controlling the learning rate dynamically. We need to find the alternative way to the gradient descent for avoiding falling into a local minimum in the error space.

## 9.5 Summary and Further Discussions

This section is concerned with the summarization of what is studied in this chapter and further discussions on it. The Perceptron is studied as the basis for doing the MLP. The architecture and the output computation process of MLP are studied. The process of optimizing the weights as the learning process from the output layer to the input layer is studied. This section is intended to make the further discussions on what is studied in this chapter.

Multiple Perceptrons were combined before inventing the MLP. The inability to the XOR problem in the Perceptron was pointed out in the 1960s. The XOR problem may be solved by combining two Perceptrons under one more Perceptron as a hierarchical structure. The combination of multiple Perceptrons was mentioned as Madaline. This model becomes the basis for inventing the MLP by Rumelhart in the 1980s.

The activation function is applied to the net input for generating the output in each node. If the net input is set by itself as the output value, the activation function is the linear function. The linear function is usually adopted in implementing the Perceptron, whereas the sigmoid function is adopted in implementing the MLP. When the MLP is applied to the regression task, the hyperbolic function is adopted as the activation function. ReLU (Rectified Linear Unit) is used as the activation function in implementing the CNN (convolutional neural network) which is covered in Chap. 12.

We mention the TDNN (time delay neural networks) as a MLP variant. If the MLP is applied to the time series prediction, it becomes the TDNN. The input of TDNN has the elements in the window which slides on the time series. The results from applying the MLP to the time series were successful compared with the traditional statistical approaches. The TDNN is replaced by the recurrent neural networks which are covered in Chap. 10, as the approach to the time series prediction.

The MLP is expanded into the CNNs (convolutional neural networks) by attaching the deep operations. There are the two parts in the CNN: feature extraction and classification part. The convolution layer and the pooling layer are arranged alternatively in the feature extraction, and the MLP or the Perceptron is put in the classification part. The input data is mapped into one which is classified easily through the feature extraction. The CNN will be studied in detail in Chap. 12.

## Reference

1. T. Mitchell, Machine Learning, McFraw-Hill, 1996

# Chapter 10

## Recurrent Neural Networks



This chapter is concerned with the recurrent neural networks which are advanced from the MLP. In the previous chapter, we studied the MLP as a typical deep neural networks model with the feedforward connections between layers. In this chapter, we study the recurrent neural network with any feedback connection from a neuron as an advanced model. In the feedback connection, a previous output value is used as an input. This chapter is intended to describe the recurrent neural networks and the variants with respect to the connection and the learning process.

This chapter is composed of five sections, and in Sect. 10.1, we overview what is studied in this chapter. In Sect. 10.2, we study the recurrent architectures of nodes, together with the feedforward ones. In Sect. 10.3, we study models of recurrent neural networks. In Sect. 10.4, we explore their applications to real classification tasks. In Sect. 10.5, we discuss further what is studied in this chapter.

### 10.1 Introduction

This section is concerned with the overview of recurrent neural networks as another deep neural network. In the previous chapter, we studied the MLP as deep neural networks with the feedforward connections. The recurrent neural networks which are covered in this chapter have the feedback connection which connects one node to itself as a loop, and it is called recurrent connection. The previous value of a node is given as the input in the recurrent connection; the feedback of the output value becomes the input value. This section is intended to overview the recurrent neural networks as the introduction to this chapter.

Let us review the architecture of MLP which was covered in the previous chapter. There are the three layers in the architecture: the input layer, the hidden layer, and the output layer. The layers are connected in the feedforward direction from the input layer to the output layer. A node in the current layer is connected to another node in the next layer; there is no feedback connection from a node to itself in the

MLP. The MLP which was studied in the previous chapter does not belong to the recurrent neural networks.

Let us mention the feedback connection from a node to itself. The node values are assumed to be in different temporal points as a temporal sequence. The previous value in a node is used as the input for computing its current value. The current value of a node which is computed by applying the activation function to the net input is used as the input for computing its next value. Its previous value influences on computing the current value in the node with its feedback connection.

Let us mention the TDNN (time delay neural networks) as the previous approach to the time series prediction before proposing the recurrent neural networks. The TDNN is the MLP which is applied to the time series prediction and receives temporal values in a window which slides on the time series. The training examples are collected by sliding the window on the temporal sequence, and the MLP is trained with them. It receives the temporal values whose last value is the current measure as its input and predicts the future measure through its generalization. The results from applying the TDNN to the time series prediction before proposing the recurrent neural networks were successful.

Let us mention what is intended in this chapter. We understand the recurrent architecture as the foundation for studying the recurrent neural networks. We study the typical models: the basic recurrent neural networks and the LSTM (long short-term memory). We present the schemes of applying the recurrent neural networks to the real applications. This chapter is intended to study the recurrent neural networks as an advanced deep neural networks model.

## 10.2 Recurrent Architecture

This section is concerned with the recurrent connections among neurons. In Sect. 10.2.1, we review the feedforward connection which was covered in the previous chapter. In Sect. 10.2.2, we describe simple recurrent connections among neurons. In Sect. 10.2.3, we describe the hybrid connection among neurons of both kinds. In Sect. 10.2.4, we describe the hidden nodes with the recurrent connections.

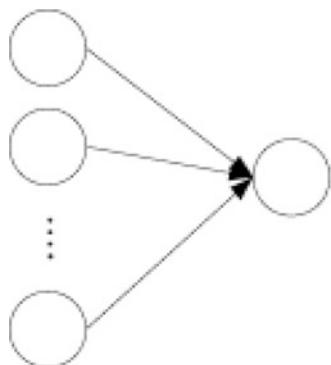
### 10.2.1 *Forward Connection*

This section is concerned with the feedforward connection in the artificial neural networks. It is the connection from one node in a layer to another node in the next layer. Each node in a layer is usually connected to all nodes in the next layer in the artificial neural networks. The neural networks with the only feedforward connection such as the Perceptron and the MLP which were covered in Chap. 9 are called feedforward neural networks. This section is intended to review

**Fig. 10.1** One-to-one connection



**Fig. 10.2** Many-to-one connection



the feedforward connection for providing the background for understanding the recurrent connection.

The one-to-one connection is illustrated as the simplest instance of the feedforward connection in Fig. 10.1. A single connection for one node to another node is associated with its own weight. The left node value is notated by  $x$ , the right node, the right node value is notated by  $y$ , and the weight which is associated with the connection between the nodes is notated by  $w$ . The two node values and the weight are related, as expressed by Eq. (10.1):

$$y = F(w \cdot x) \quad (10.1)$$

where  $F$  is an activation function. The node values and the weight are assumed to be real values.

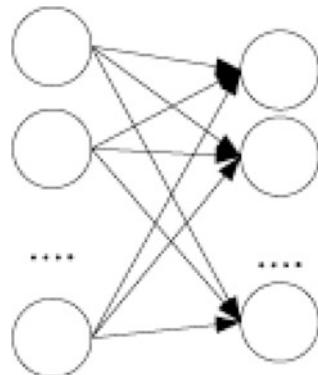
The one-to-many connection is illustrated as another instance of the feedforward connection in Fig. 10.2. Multiple nodes are connected to a single node, and each connection is associated with its own weight. The left values are notated by  $x_1, x_2, \dots, x_d$ , and the weights are notated by  $w_1, w_2, \dots, w_d$ . The right node value is computed by Eq. (10.2):

$$y = F \left( \sum_{i=1}^d w_i \cdot x_i \right). \quad (10.2)$$

The output node value depends on the  $d$  input node values in this instance of the feedforward connection.

The many-to-many connection is illustrated as the third instance of the feedforward connection in Fig. 10.3. The complete feedforward connection between two layers is one where each node connects in a layer with all nodes in the next layer.

**Fig. 10.3** Many-to-many connection



The left node values are denoted by  $x_1, x_2, \dots, x_d$ , the right node values are denoted by  $y_1, y_2, \dots, y_c$ , and the weight is denoted by  $w_{ji}$ , where  $i$  is the right node index and  $j$  is the left node index. The right node value is computed by Eq. (10.3):

$$y_j = F \left( \sum_{i=1}^d w_{ji} \cdot x_i \right). \quad (10.3)$$

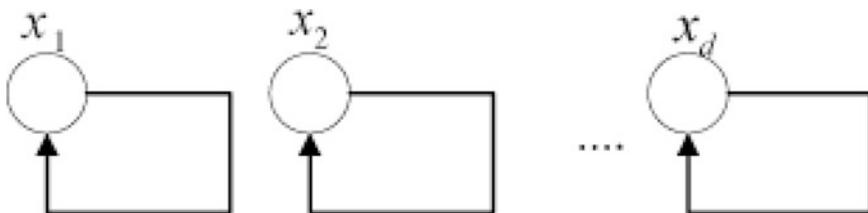
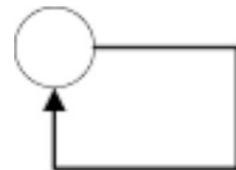
The reason that the left subscript is the destination node index is that the weight is updated with the backward direction, from the destination to the source.

Let us make some remarks on the feedforward connection which is reviewed in this section. The one-to-one connection is the connection from a node to another node as the simplest example. The one-to-many connection is the connection from multiple nodes to a node as another example. The many-to-many connection is the complete connection between two layers where each node in the current layer is connected to all nodes in the next layer. The feedforward connection is interpreted as the connection from one to different nodes; the neural networks which are studied in the previous chapter consist of only feedforward connections.

## 10.2.2 Recurrent Connection

This section is concerned with the recurrent connection which is needed for understanding the recurrent neural networks. In the previous section, we studied the feedforward connection which is one from a node to another node. In this section, we study the recurrent connection, which is one from a node to itself, where a current node value is used as the input for computing its next value. In this type of connection, we consider the temporal sequence which consists of node values in the sequential time. This section is intended to describe several cases of the recurrent connection which are used for implementing the recurrent networks.

**Fig. 10.4** Single self-connection



**Fig. 10.5** Multiple recurrent connections

The simplest example of the recurrent connection, called self-connection, is illustrated in Fig. 10.4. A single connection from a node to itself is assumed to be associated with the weight,  $w$ . The previous node value is denoted by  $x(t - 1)$ , and the current value is denoted by  $x(t)$ . The relation between the previous value and the current value is expressed by Eq. (10.4),

$$x(t) = w \cdot x(t - 1). \quad (10.4)$$

The previous value is used as the input for computing the current value.

Another example of the recurrent connection is illustrated in Fig. 10.5. There are  $d$  nodes, they are denoted by  $x_1, x_2, \dots, x_d$ , and each node is connected to itself. The recurrent connections are associated with the weights,  $w_1, w_2, \dots, w_d$ . The current values of the  $d$  nodes are computed by Eq. (10.5):

$$x_1(t) = w_1 \cdot x_1(t - 1), x_2(t) = w_2 \cdot x_2(t - 1), \dots, x_d(t) = w_d \cdot x_d(t - 1). \quad (10.5)$$

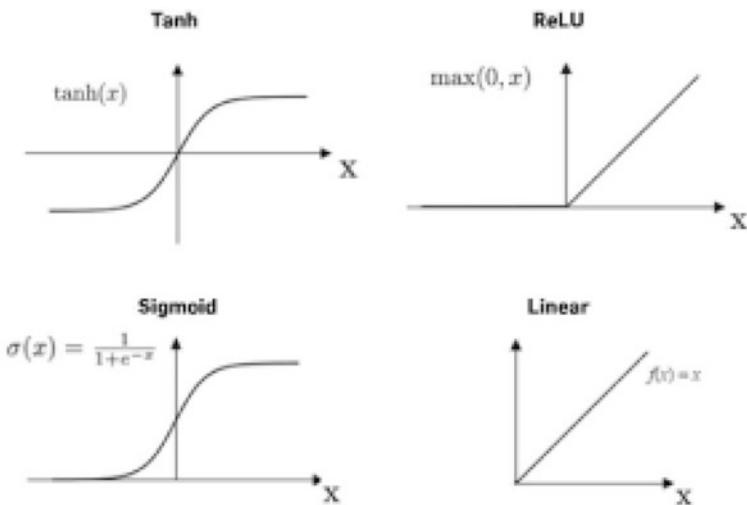
The current value of each node depends on its previous value in the current connection.

The four representative activation functions are illustrated in Fig. 10.6. They are applied to the recurrent connections as well as the feedforward connections. If an activation function is applied, the current value is computed by Eq. (10.6):

$$x_i(t) = F(w_i \cdot x(t - 1)). \quad (10.6)$$

Equation (10.5) for computing the current value is the case of adopting the linear function as the activation function. The sigmoid function or the tanh function is adopted in implementing the recurrent networks.

Let us make some remarks on the recurrent connection where a node is connected to itself. The current node value is computed by product of its previous value



**Fig. 10.6** Activation functions from <https://docs.paperspace.com/machine-learning/wiki/activation-function>

and its weight. A single recurrent connection is expanded into multiple recurrent connections from multiple nodes. The activation function is applied to the product of the previous values and the weights. We consider a circular connection between two nodes, associated with its dual weights.

### 10.2.3 Hybrid Architecture

This section is concerned with the architecture with the mixture of the feedforward connection and the recurrent connection. Both kinds of connections are studied separately in the previous sections. In this section, we study the mixture of both kinds of connections as the integrated architecture. The recurrent neural networks which are covered in this chapter have the architecture which consists of both kinds of connections. This section is intended to survey the three cases of the hybrid architecture.

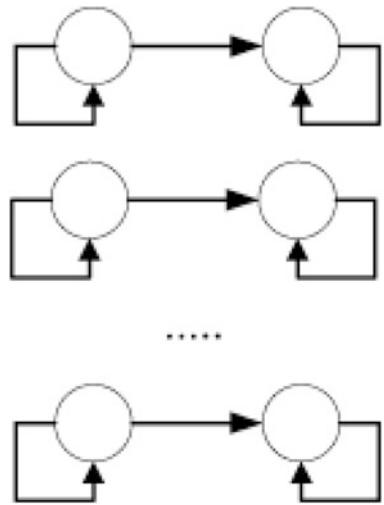
An example of the hybrid connection between two nodes is illustrated in Fig. 10.7. The values of the left node and the right node are denoted by  $x_1$  and  $x_2$ , respectively. The recurrent connection of the left node and the right node is associated with the weights,  $w_1$  and  $w_2$ , respectively, and the feedforward connection from the left node to the right node is associated with the weight,  $w_{21}$ . The value of  $x_1$  is computed by Eq. (10.7):

$$x_1(t) = F(w_1 \cdot x_1(t-1)) \quad (10.7)$$

**Fig. 10.7** Single feedforward and recurrent connection



**Fig. 10.8** Multiple feedforward and recurrent connection



the value of  $x_2$  is computed by Eq. (10.8):

$$x_2(t) = F(w_{21} \cdot x_1(t-1) + w_2 \cdot x_2(t-1)) \quad (10.8)$$

The value of  $x_2$  depends on both the current value of  $x_1$  and the previous value of  $x_2$ .

Another example of the hybrid connection among nodes is illustrated in Fig. 10.8. The left nodes and the right nodes are denoted by  $x_1, x_2, \dots, x_d$  and  $y_1, y_2, \dots, y_d$ , respectively. There are three kinds of weights in this connection: the recurrent connection of the left nodes which is denoted by  $w_{l,1}, w_{l,2}, \dots, w_{l,d}$ ; the recurrent connection of the right nodes which is denoted by  $w_{r,1}, w_{r,2}, \dots, w_{r,d}$ ; and the feedforward connection between the left nodes and the right nodes which is denoted by  $w_{rl,1}, w_{rl,2}, \dots, w_{rl,d}$ . The value of  $x_i$  is computed by Eq. (10.9):

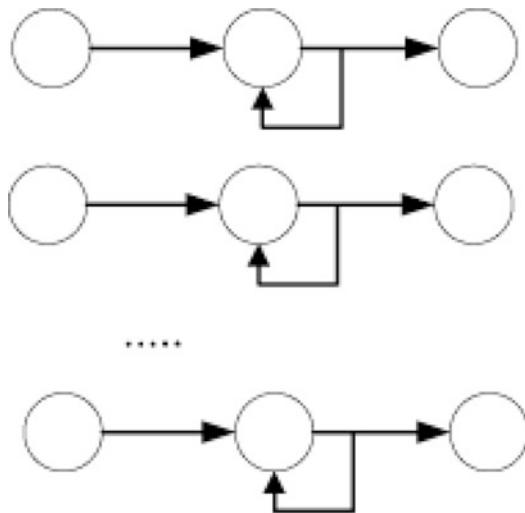
$$x_i(t) = F(w_{l,i} \cdot x_i(t-1)) \quad (10.9)$$

The value of  $y_i$  is computed by Eq. (10.10):

$$y_i(t) = F(w_{rl,i} \cdot x_i(t) + w_{r,i} \cdot y_i(t-1)) \quad (10.10)$$

The example which is presented in Fig. 10.8 is the expansion of the example which is presented in Fig. 10.7.

**Fig. 10.9** Multiple layers with Feedforward and recurrent connection



The more complicated example of the hybrid connection among nodes is illustrated in Fig. 10.9. The left nodes, the middle nodes, and the right nodes are denoted by  $x_1, x_2, \dots, x_d$ ,  $h_1, h_2, \dots, h_d$ , and  $y_1, y_2, \dots, y_d$ , respectively. There are three kinds of weights: the feedforward connections between the left layer and the middle layer which are associated with  $w_1^1, w_2^1, \dots, w_d^1$ ; the feedforward connections between the middle layer and the right layer which are associated with  $w_1^2, w_2^2, \dots, w_d^2$ ; and the recurrent connections of the middle nodes which are associated by  $w_1^r, w_2^r, \dots, w_d^r$ . The value of  $h_i$  is computed by Eq. (10.11):

$$h_i(t) = F(w_i^1 \cdot x_i + w_i^r \cdot h_i(t-1)). \quad (10.11)$$

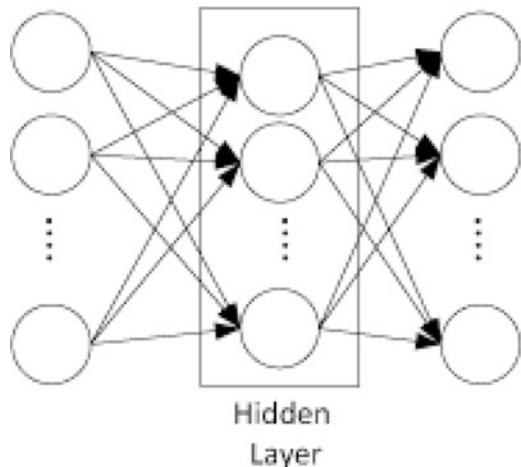
The value of  $y_i$  is computed by Eq. (10.12),

$$y_i(t) = F(w_i^2 \cdot h_i(t)). \quad (10.12)$$

Each hidden node has its recurrent connection in the recurrent networks which is covered in this chapter.

Let us make some remarks on the hybrid connections of node with the mixture of the feedforward connection and the recurrent connection. The connection from a node to another node and the recurrent connection of each node are viewed as the first case of hybrid connection. We study the second case of hybrid connection where the first cases of hybrid connection are given as independent modules. We study the third case where the recurrent connections of hidden nodes are added to the feedforward connection from the input layer to the output layer. The recurrent neural network which is studied in subsequent sections has more complicated hybrid connections.

**Fig. 10.10** Feedforward hidden layer



#### 10.2.4 Hidden Recurrency

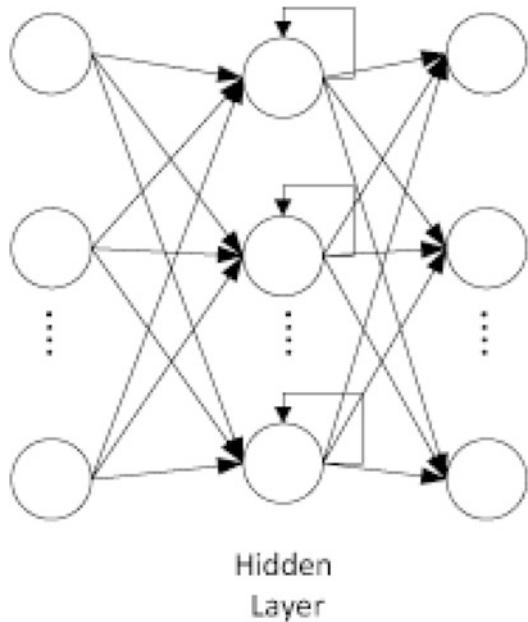
This section is concerned with the recurrent connections in the hidden layer. In the previous sections, we studied the feedforward connection, the recurrent connection, and the hybrid connection. In this section, we study the addition of the recurrent connection to the hidden layer in the MLP architecture. The current value of a hidden node depends on the input values and its previous value. This section is intended to describe the recurrent connection in the hidden layer as the architecture of the recurrent neural networks.

The hidden layer with the only feedforward connection is illustrated in Fig. 10.10. The architecture was already studied in Chap. 9. The feedforward connections between the input layer and the hidden layer and between the hidden layer and the output layer are complete. Each hidden node is connected from all input nodes and connected to all output nodes without their recurrent connections. The architecture which is presented in Fig. 10.10 is the basis for studying the recurrent networks in this chapter.

The hidden layer with both the feedforward connection and the recurrent connection is illustrated in Fig. 10.11. The architecture consists of the three layers, the input layer, the hidden layer, and the output layer, and each node in the hidden layer is connected from all input nodes and to all output nodes. Each hidden node is connected to itself, recurrently, as the additional part to the architecture in Fig. 10.10. The hidden node value depends on both the input node values and its previous value, and each recurrent connection is associated with the weight which is separated with ones of the feedforward connections. The architecture which is presented in Fig. 10.11 is one of the recurrent neural networks which are studied in the next section.

The multiple hidden layers with their own recurrent connections are illustrated in Fig. 10.12. It is assumed that the  $K$  hidden layers are available between the input

**Fig. 10.11** Recurrent hidden layer

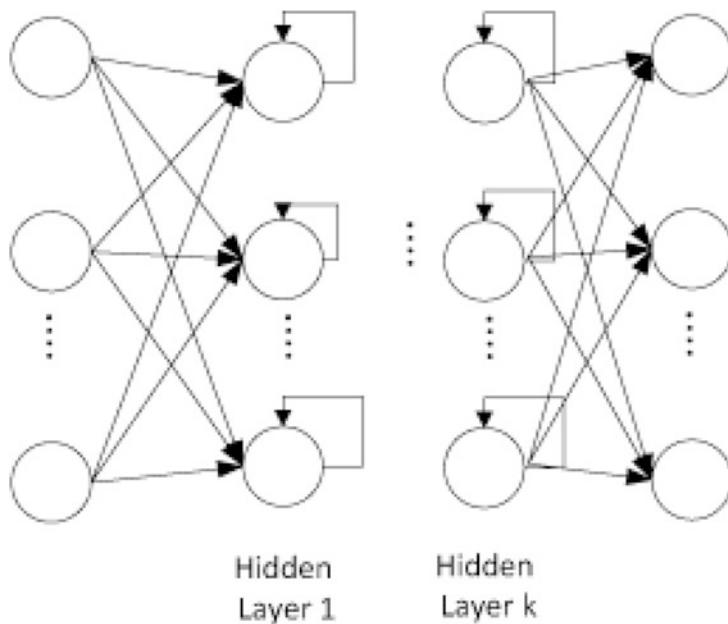


layer and the output layer. Each hidden node is connected to itself recurrently, connected from all nodes in the previous layer, and connected to all nodes in the next layer. The current value of each hidden node depends on its previous value and the values of all nodes in the previous layer. The architecture which is presented in Fig. 10.12 is one of the expanded RNN version.

Let us make some remarks on the recurrent connections in the hidden layer. If there is no recurrent connection, the architecture of the three layers is the same to that of the MLP which is covered in Chap. 9. The recurrent architecture with a hidden layer is expanded into one with multiple hidden layers. The recurrent connection, each of which is connected a node to itself, is installed in the hidden layer.

### 10.3 Recurrent Neural Networks

This section is concerned with some models of recurrent neural networks. In Sect. 10.3.1, we describe the basic model of recurrent neural networks. In Sect. 10.3.2, we present some variants which are derived from the basic model. In Sect. 10.3.3, we describe the typical model which is called LSTM (long short-term memory). In Sect. 10.3.4, we derive some variants from the LSTM.



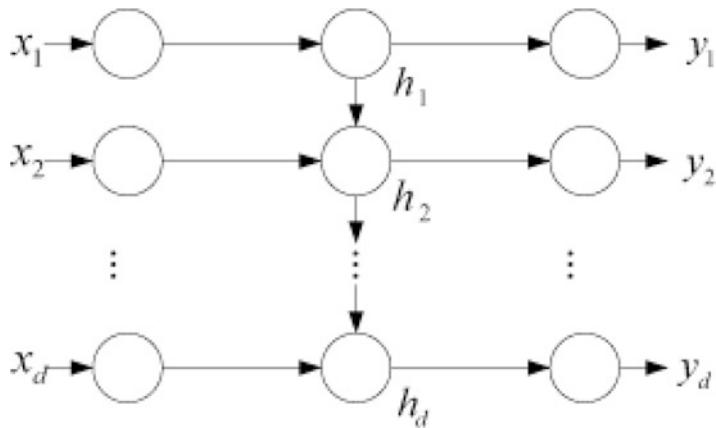
**Fig. 10.12** Multiple recurrent hidden layers

### 10.3.1 Basic Recurrent Neural Networks

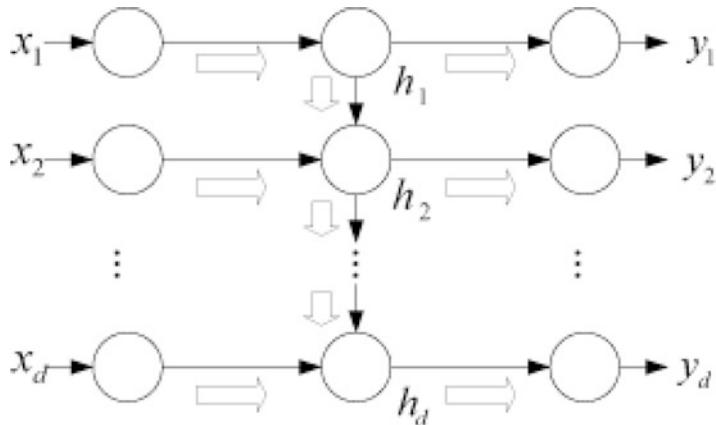
This section is concerned with the initial version of RNN (recurrent neural networks). In the previous section, we studied the recurrent connection and the feedforward connection as the preparation for studying the RNN. In the RNN architecture, there are three layers, the input layer, hidden layer, and output layer, and there are the recurrent connections of each node in the hidden layer. The difference of the RNN architecture from the MLP architecture is that the recurrent connections are added in the hidden layer. This section is intended to describe the basic RNN with respect to its architecture, its computation process, and its learning process.

The RNN architecture is illustrated in Fig. 10.13. There are three layers in the architecture: input layer, hidden layer, and output layer. The one-to-one connection from the input layer to the hidden layer and the one-to-one connection from the hidden layer to the output layer are the differences from the MLP architecture. The serial connection from the first hidden node to the last hidden node in the hidden layer characterizes the RNN. The node values are computed from the input layer to the output layer in the feedforward direction.

The direction of computing the output node values in the RNN is illustrated in Fig. 10.14. The weights are assumed as the optimized ones, and the input vector,  $\mathbf{x} = [x_1 \ x_2 \ \dots \ x_d]$ , is initially given. Each hidden node value is computed by Eq. (10.13):



**Fig. 10.13** Architecture of basic recurrent networks



**Fig. 10.14** Classification process of basic recurrent networks

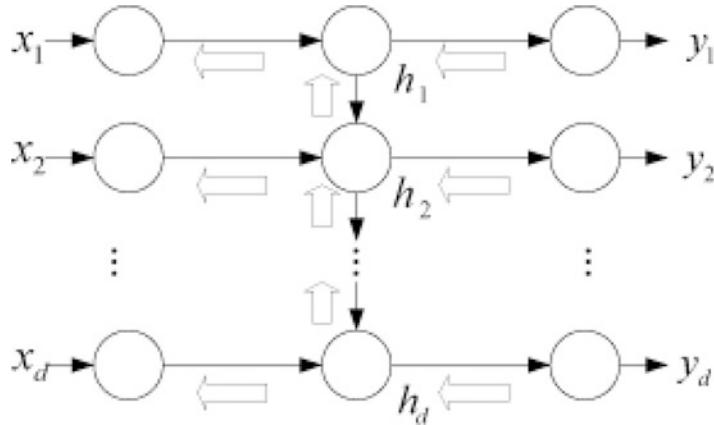
$$h_i = \tanh(w_{i-1}^h \cdot h_{i-1} + w_i^1 \cdot x_i) \quad (10.13)$$

where  $w_{i-1}^h$  is the weight from the hidden node,  $h_{i-1}$ , to the hidden node,  $h_i$ , and the first hidden node value is computed by Eq. (10.14):

$$h_1 = \tanh(w_1 \cdot x_i). \quad (10.14)$$

Each output node value is computed by Eq. (10.15), like the case in the MLP:

$$\hat{y}_i = w_i^2 \cdot h_i. \quad (10.15)$$



**Fig. 10.15** Learning process of basic recurrent networks

The existence of weights in the hidden layer is the difference of the RNN from the MLP.

The direction of optimizing the weights as the learning process of RNN is illustrated in Fig. 10.15. The weight between the hidden layer and the output layer,  $w_i^2$ , is updated by Eq. (10.16):

$$w_i^2 \leftarrow w_i^2 + \eta h_i (y_i - \hat{y}_i) \quad (10.16)$$

The weight between hidden nodes,  $w_i^h$ , is updated by Eq. (10.17):

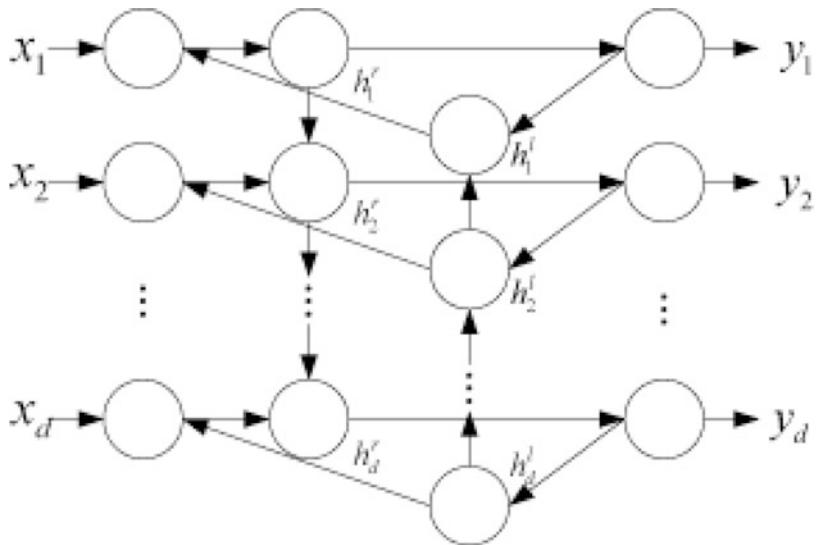
$$w_i^h \leftarrow w_i^h + \eta h_i \sum_{l=0}^{d-i-1} w_{d-l}^2 (y_{d-l} - \hat{y}_{d-l}) \prod_{k=i+1}^{d-l} (1 - h_k)(1 + h_k) w_k^h \quad (10.17)$$

The weight between the input layer and the hidden layer is updated by Eq. (10.18):

$$w_i^1 \leftarrow w_i^1 + \eta x_i \sum_{l=0}^{d-i} w_{d-l}^2 (y_{d-l} - \hat{y}_{d-l}) \prod_{k=i}^{d-l} (1 - h_k)(1 + h_k) w_k^h \quad (10.18)$$

The activation is assumed as the hyperbolic function in deriving the weight update equations.

Let us make some remarks on the basic version of the RNN. Like the MLP, there are three layers in the RNN, the input layer, the hidden layer, and the output layer, and the hidden nodes are connected serially with each other. The term, product of the previous hidden node and a weight, is added in computing a hidden node value. The weights between hidden nodes are updated as the additional part in the RNN. Its variants are derived by defining connections among layers differently, adopting different activation functions, or expanding the hidden layer.



**Fig. 10.16** Bidirectional recurrent networks

### 10.3.2 RNN Variants

This section is concerned with some variants of the RNN. In the previous section, we studied the initial version of the RNN, with respect to its generalization and its learning process. In this section, we derive some variants from the standard RNN. Various types of variants are derived by manipulating the connections between layers and adding more hidden layers. This section is intended to describe the three typical RNN variants.

The RNN variant with the dual hidden layers is illustrated in Fig. 10.16. One hidden layer is used for computing the output node values, and the other is used for computing the input node values. An output node value is computed by Eq. (10.19):

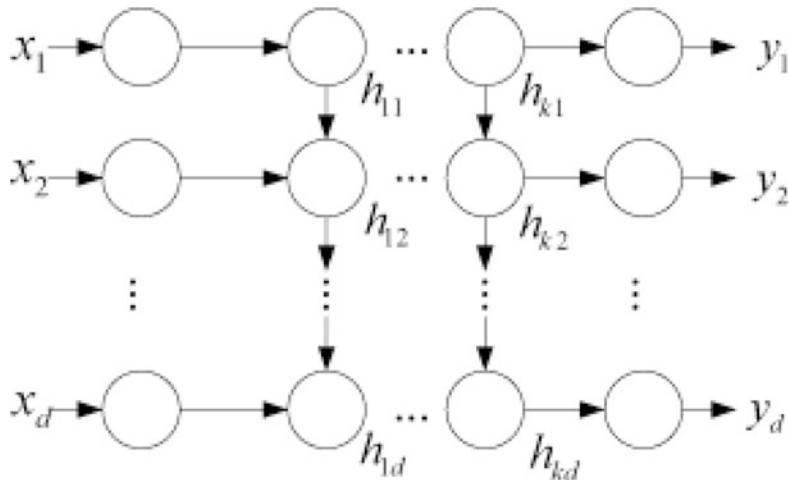
$$\hat{y}_i = \tanh(w_i^r \cdot h_i^r), \quad (10.19)$$

where  $w_i^r$  is the weight from the hidden node,  $h_i^r$ , and an input node value is computed by Eq. (10.20):

$$\hat{x}_i = \tanh(w_i^l \cdot h_i^l), \quad (10.20)$$

where  $w_i^l$  is the weight from the hidden node,  $h_i^l$ . The hidden node value,  $h_i^r$ , is computed by Eq. (10.21):

$$h_i^r = \tanh(w_i^{hr} \cdot h_{i-1}^r + w_i^{1r} \cdot x_i), \quad (10.21)$$



**Fig. 10.17** Recurrent networks with multiple hidden layers

where  $w_i^{hr}$  is the weight from the hidden node,  $h_{i-1}^r$  to the hidden node,  $h_i^r$ , and  $w_i^{1r}$  is the weight from the input node to the hidden node, and the hidden node value,  $h_i^l$ , is computed by Eq. (10.22):

$$h_i^l = \tanh(w_i^{hl} \cdot h_{i-1}^l + w_i^{1l} \cdot y_i), \quad (10.22)$$

where  $w_i^{hl}$  is the weight from the hidden node,  $h_{i-1}^l$ , to the hidden node,  $h_i^l$ , and  $w_i^{1l}$  is the weight from the input node to the hidden node. This variant which is called BRNN (bidirectional recurrent neural networks) is used for modeling both directions between temporal sequences.

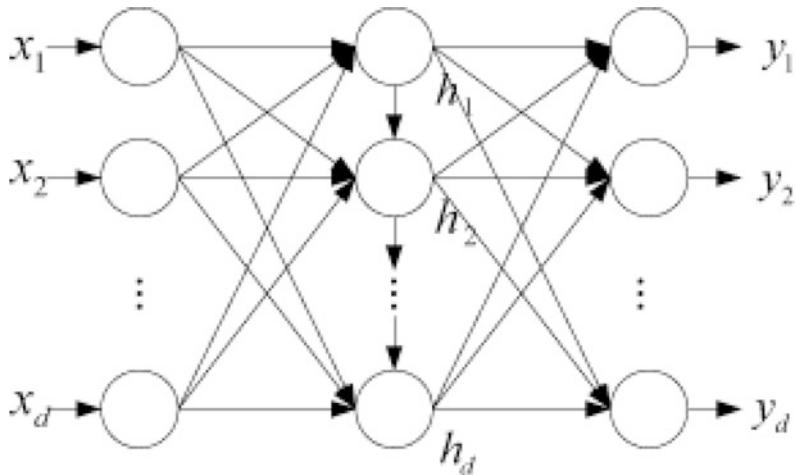
The RNN variant with multiple hidden layers is illustrated in Fig. 10.17. A hidden node value in the first hidden layer,  $h_{1i}$ , is computed by Eq. (10.23), depending on the input value and the previous hidden node value:

$$h_{1i} = \tanh(w_i^1 \cdot x_i + w_i^{1h} \cdot h_{1(i-1)}), \quad (10.23)$$

where  $w_i^{1h}$  is the weight from the hidden node,  $h_{1(i-1)}$ , to the hidden node,  $h_{1i}$ . The hidden node value in the middle hidden layer,  $h_{ci}$ , is computed by Eq. (10.24), depending on the hidden node value in the previous hidden layer and the previous hidden node in the current hidden layer:

$$h_{ci} = \tanh(w_i^{(c-1)h} \cdot h_{(c-1)i} + w_i^{ch} \cdot h_{c(i-1)}), \quad (10.24)$$

where  $w_i^{(c-1)h}$  is the weight from the hidden node,  $h_{(c-1)i}$ , to the hidden node,  $h_{ci}$ , and  $w_i^{ch}$  is the weight from the hidden node,  $h_{c(i-1)}$ , to the hidden node,  $h_{ci}$ . The



**Fig. 10.18** Recurrent networks with complete connections

output node value,  $\hat{y}_i$ , is computed by Eq. (10.25), depending on the hidden node value in the last hidden layer:

$$\hat{y}_i = \tanh(w_i^{kh} \cdot h_{ki}), \quad (10.25)$$

where  $w_i^{kh}$  is the weight from the hidden node  $h_{ki}$  in the last hidden layer to the output node. This variant is adopted for solving more nonlinear relation between the input vector and the output vector.

The RNN variant with the complete connections in its architecture is illustrated in Fig. 10.18. The complete connection between the input layer and the hidden layer and between the hidden layer and the output layer is the difference of this variant from the initial RNN version. Each hidden node value is computed by Eq. (10.26):

$$h_i = \tanh \left( w_{i-1}^h \cdot h_{i-1} + \sum_{j=1}^d w_i^1 \cdot x_j \right) \quad (10.26)$$

Each output node value is computed by Eq. (10.27):

$$\hat{y}_i = \tanh \left( \sum_{j=1}^d w_i^2 \cdot h_j \right) \quad (10.27)$$

The architecture of this RNN variant is the same to that of the MLP except the serial connection within the hidden layer.

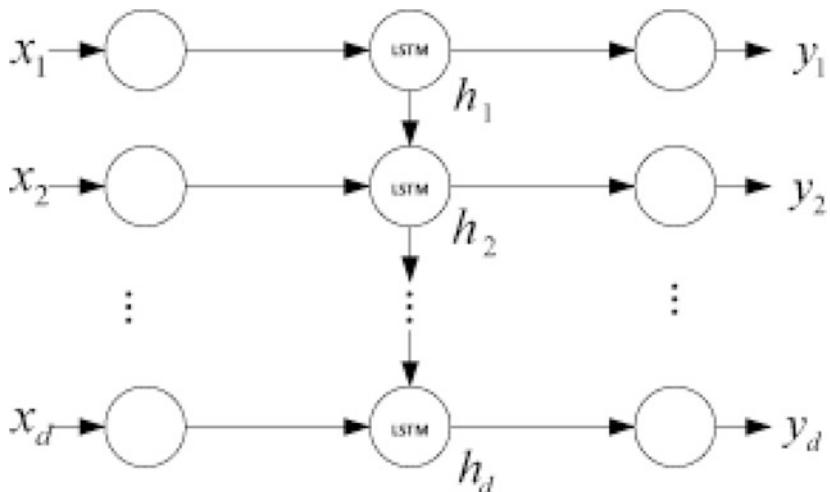
Let us make some remarks on the three RNN variants. The dual hidden layers lead the bidirection between the input layer and the output layer in the first variant.

Multiple hidden layers with their own serial connections are for solving nonlinear problems. One-to-one connection is replaced by the complete connection between the input layer and the hidden layer and between the hidden layer and the output layer. More variants are derived from the basic RNN version by manipulating the architecture and the activation function.

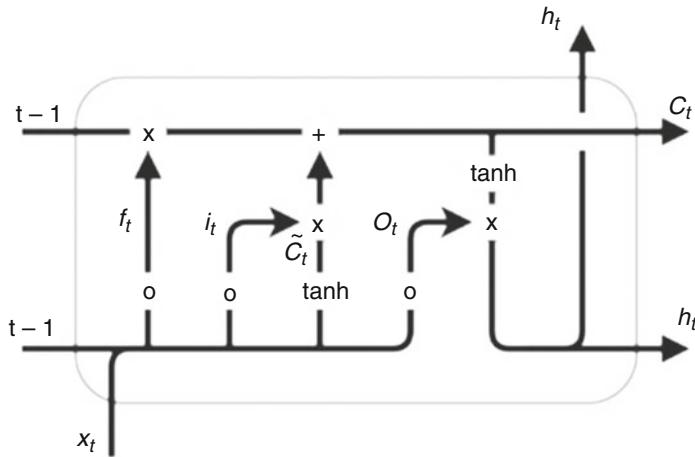
### 10.3.3 LSTM (Long Short-Term Memory)

This section is concerned with an instance of recurrent neural networks, called LSTM (long short-term memory). The limits of the RNN and its variants which are covered in the previous sections are its strong dependency on recent ones and weak dependency on past ones. This version of RNN, which is called LSTM, is designed for solving the limits by memorizing more past ones. The cell state and one more activation function, sigmoid, are involved additionally in the LSTM. This section is intended to describe the LSTM as the more advanced version of RNN.

The external architecture of the LSTM is illustrated in Fig. 10.19. The architecture of the basic RNN version which is covered in Sect. 10.3.2 is the basis of the architecture of the LSTM. The hidden node which consists of a single activation function, hyperbolic function, is replaced by the composition of multiple activation functions, hyperbolic function, and sigmoid function, in the LSTM. The existence of the cell state as an additional part is given as the conveyer through the entire hidden layer in the LSTM. Dual values of each hidden node exist in the LSTM: the hidden node value and the cell state value.



**Fig. 10.19** LSTM architecture



**Fig. 10.20** Inside hidden node in LSTM

The internal structure of the hidden node in the LSTM is illustrated in Fig. 10.20. The top horizontal arrow in the hidden node is the cell state which flows through the hidden layer, and the left vertical arrow is the forget gate which is expressed as Eq. (10.28):

$$F_i = \text{sigmoid}(w_i^{1F} \cdot x_i + w_i^{hF} \cdot h_{i-1}). \quad (10.28)$$

The two middle vertical arrows become the input gate which is expressed as Eq. (10.29):

$$I_i = \text{sigmoid}(w_i^{1I} \cdot x_i + w_i^{hI} \cdot h_{i-1}). \quad (10.29)$$

The right vertical arrow which involves the sigmoid function is the output gate which is expressed as Eq. (10.30) and used for computing the hidden node value:

$$O_i = \text{sigmoid}(w_i^{1O} \cdot x_i + w_i^{hO} \cdot h_{i-1}). \quad (10.30)$$

The forget gate and the input gate are used for updating the cell state.

Let us mention the process of computing the cell state and the hidden node value in the LSTM. The delta cell state is computed by Eq. (10.31):

$$\Delta C_i = \tanh(w_i^{1C} \cdot x_i + w_i^{hC} \cdot h_{i-1}). \quad (10.31)$$

The current cell state is computed by Eq. (10.32):

$$C_i = F_i \cdot C_{i-1} + I_i \cdot \Delta C_i \quad (10.32)$$

The hidden node value is computed by Eq. (10.33):

$$h_t = O_i \cdot \tanh(C_i). \quad (10.33)$$

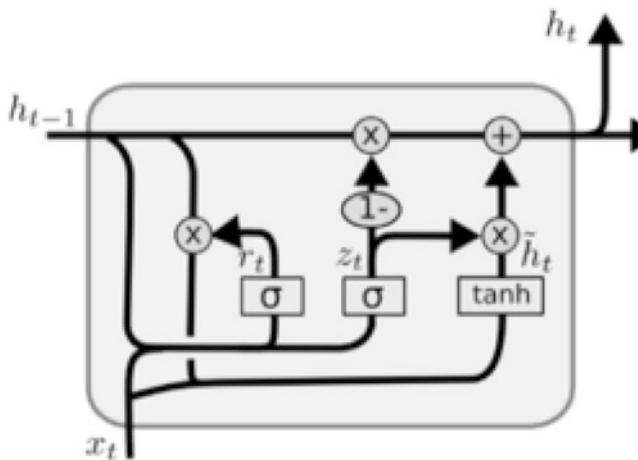
The hidden node value depends on the output gate and the cell state in the LSTM.

Let us make some remarks on the LSTM as the advanced version of the recurrent neural networks. Multiple activation functions are involved in each hidden node as the difference of the LSTM from the basic version of RNN. The three gates, the input gate, the forget gate, and the output gate, are included in each hidden node. A hidden node value is computed by the product of the output gate and the hyperbolic of the cell gate. The cell state and a previous hidden node value are used for computing both the cell state and the current hidden node value.

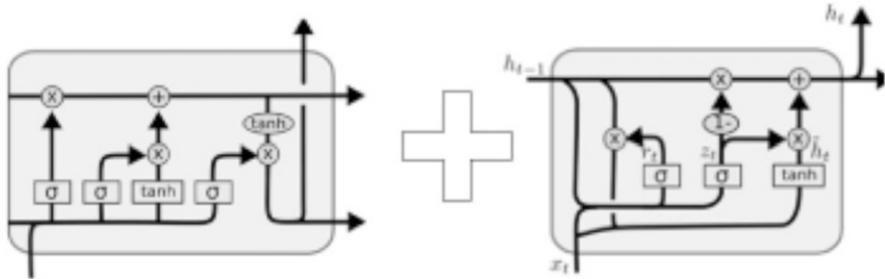
#### 10.3.4 LSTM Variants

This section is concerned with some variants which are derived from the LSTM. In the previous section, we studied the initial version of LSTM as a kind of recurrent neural networks. In this section, we derive some variants from the LSTM by modifying the circuits in each hidden node. The LSTM and its variants are intended to prevent the deterioration of previous hidden node values. This section is intended to describe some variants of LSTM.

Let us mention the GRU (Gate Recurrent Unit) which is illustrated in Fig. 10.21 as the first LSTM variant, briefly. In the LSTM, there are three gates, input gate, forget gate, and output gate, whereas in the GRU, there are two gates: reset gate and update gate. The reset gate and the update gate are computed by Eqs. (10.34) and (10.35), respectively,



**Fig. 10.21** Gate recurrent unit



**Fig. 10.22** Gate recurrent unit long short-term memory

$$r_i = \text{sigmoid}(w_i^{1R} \cdot x_i + w_i^{hR} \cdot h_{i-1}). \quad (10.34)$$

$$u_i = \text{sigmoid}(w_i^{1U} \cdot x_i + w_i^{hU} \cdot h_{i-1}). \quad (10.35)$$

The candidate value is computed by Eq. (10.36):

$$\tilde{h}_i = \tanh(w_i^{1C} \cdot r_i + u_i \cdot x_i), \quad (10.36)$$

a hidden value is computed by Eq. (10.37):

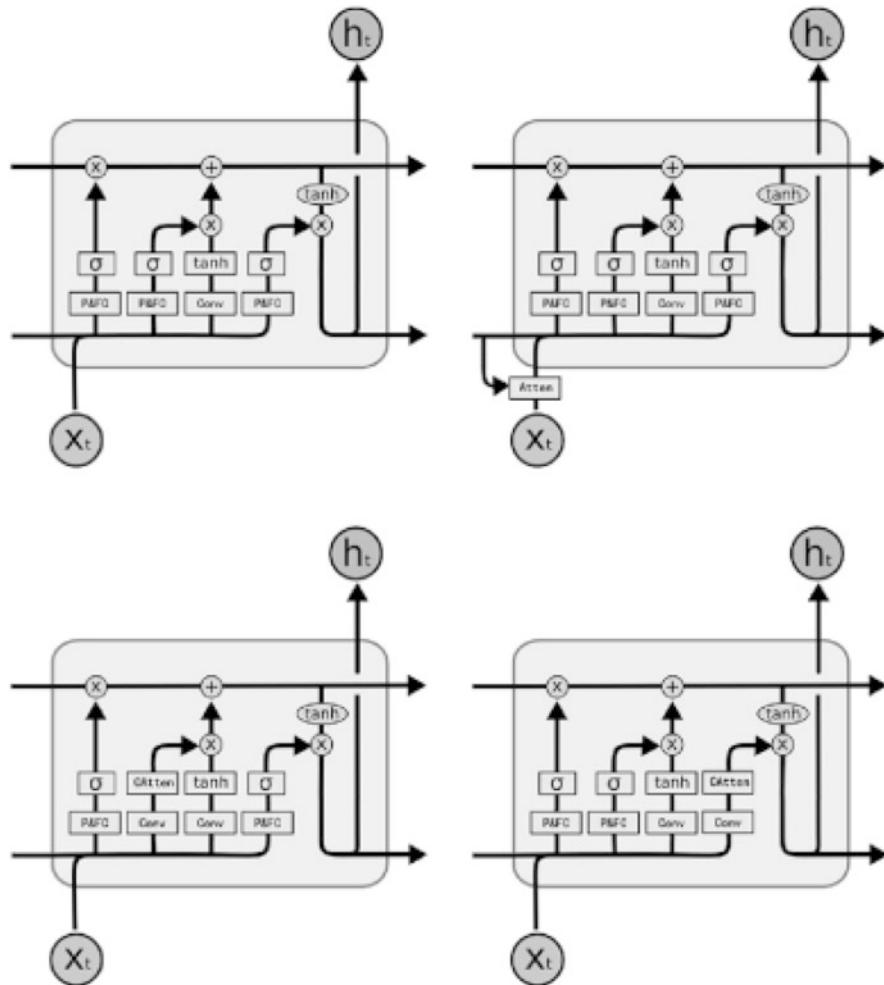
$$\tilde{h}_i = (1 - u_i)h_{i-1} + u_i \cdot \tilde{h}_i. \quad (10.37)$$

The GRU is viewed as a simplified version of LSTM.

As illustrated in Fig. 10.22, let us consider combining the LSTM and the GRU with each other. The difference between them is whether the cell state, as the alternative output,  $C_i$ , is involved or not. The scheme of combining them is to design the RNN with the dual hidden layer: one with LSTM and the other with GRU. Another scheme is to adopt the LSTM or GRU for each hidden node, alternatively. We may consider the compound unit with the LSTM and the GRU in each hidden node.

Two more variants of LSTM are illustrated in Fig. 10.23. The variants are initially proposed by Zhang et al. in 2018, as the approaches to the gesture recognition [1]. There are two outputs in each hidden node as the common part with the LSTM: the cell state and the hidden node value. The convolution is indicated by “conv” and the pooling is indicated by “P&FC” inside each hidden node. Refer to [1] for studying the variants in detail.

Let us make some remarks on the LSTM variants which are derived by modifying the circuits. The GRU has the two gates: reset gate and update gate. The combination of the LSTM and the GRU is considered in designing the recurrent neural networks. Inside each hidden node, the convolution and the pooling may be added. Other LSTM variants are derived by designing circuits inside each hidden node.

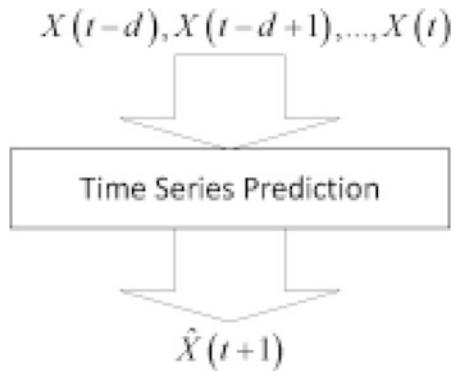


**Fig. 10.23** Four variants of convolutionary LSTM

## 10.4 Applications

This section is concerned with the areas to which the RNN is applied. In Sect. 10.4.1, we apply the RNN to the time series prediction. In Sect. 10.4.2, we apply it to the sentimental analysis. In Sect. 10.4.3, we apply it to the language modeling. In Sect. 10.4.4, we apply it to the machine translation.

**Fig. 10.24** Functional view of time series prediction



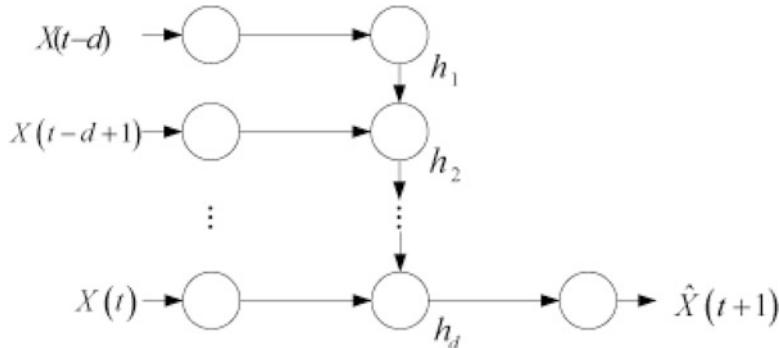
### 10.4.1 Time Series Prediction

This section is concerned with the application of RNN to the time series prediction. In Sect. 10.3, we studied entirely the learning and the generalization of RNN. In this section, we apply the RNN to the time series prediction. The RNN is designed into the number of input nodes following the window size and single output node. This section is intended to describe the process of applying the RMM to the time series prediction.

The time series prediction is illustrated with its functional view in Fig. 10.24. It is the process of predicting the future value or values by analyzing the past and current measures. In assuming the task as univariate time series prediction, the past values,  $X(t-d), \dots, X(t-1)$ , and the current value,  $X(t)$ , are given as the input, and the future value,  $\hat{X}(t+1)$ , is predicted as its output. The process of defining a mathematical equation or a formal process of predicting the future value is called time series modeling. The statistical models, such as AR (auto regression) and MA (moving average), were replaced by the neural networks, such as MLP and SVM, in the 1990s.

The architecture of RNN which is applied to the time series prediction is illustrated in Fig. 10.25. The past values and the current values,  $X(t-d), X(t-d+1), \dots, X(t)$ , correspond to the input nodes. Each hidden node,  $h_i$ , is connected with  $X(t-d+i+1)$  and  $h_{i-1}$ . The difference of this architecture from the regular one is the existence of only single output node. The output node which corresponds to  $\hat{X}(t+1)$  is connected with the previous hidden nodes and the current value,  $X(t)$ .

Let us mention the process of predicting the future value by the RNN. The past values and the current value,  $X(t-d), X(t-d+1), \dots, X(t)$ , are prepared as the input. The hidden values,  $h_1, h_2, \dots, h_d$ , are computed by the input values and the previous hidden values; the first hidden value,  $h_1$ , is computed by only input value,  $X(t-d)$ . The output value,  $\hat{X}(t+1)$ , is computed by the last hidden value,  $h_d$ . The past values influence on computing the last hidden value through the previous hidden values.



**Fig. 10.25** RNN architecture for time series prediction

Let us make some remarks on the application of the RNN to the time series prediction. The task is the process of predicting a future value by analyzing the past values and the current value. Each hidden node is connected with its own input measure and its previous hidden node, and each output node is connected with its own hidden node. Each hidden value is computed by its own input value and its previous hidden value, and the output value is computed by its own hidden value. Only one output node exists in the architecture of the RNN which is applied to the time series prediction.

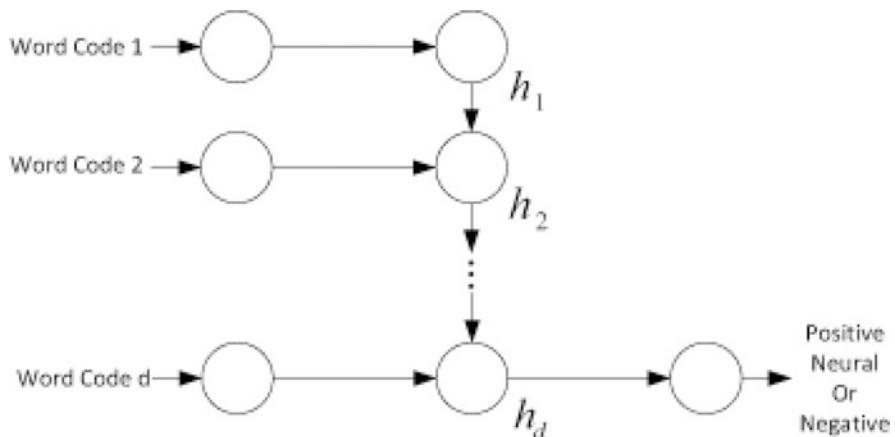
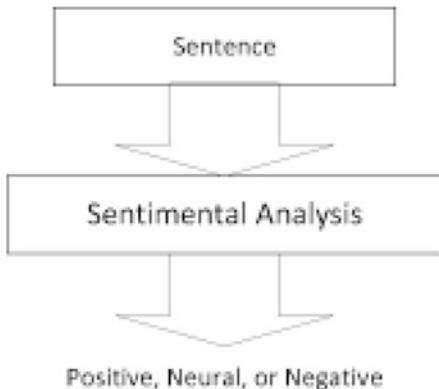
#### 10.4.2 Sentimental Analysis

This section is concerned with the application of RNN to the sentimental analysis. The task is the process of classifying a text or a sentence into negative, neutral, or positive. Each word corresponds to its own numerical value which is called word code, and the input values are given as word codes in a text or a sentence. The input nodes are connected to the hidden nodes as the one-to-one connection, and the output node is connected to only last hidden node. This section is intended to describe the RNN which is applied to the sentimental analysis.

The semantic analysis is illustrated with the functional view in Fig. 10.26. It is the process of judging whether a statement or a sentence is positive, neutral, or negative. A sentence is given as the input, and it is classified into positive, neutral, or negative. Even if both tasks, text classification and sentimental analysis, look similar, the text classification is the process of classifying a text with more than one paragraph into one among the predefined topics, and the sentimental analysis is the process of classifying a sentence or a short text into one of the three categories. The basic version of RNN is applied to this task.

The architecture of the RNN which is applied to the sentimental analysis is illustrated in Fig. 10.27. The raw input is assumed as a sentence which consists of

**Fig. 10.26** Functional view of sentimental analysis



**Fig. 10.27** RNN architecture for sentimental analysis

several words, and each of them is codified into its own numerical values. The input nodes correspond to the word codes which are given as numerical values, and each hidden node is connected from its own input node and its previous hidden node. The output node which is given as one of the three discrete values which indicates positive, neutral, and negative is connected from the last hidden node. The difference from the RNN which is applied to this task from one which is applied to time series prediction is that the output node is given as a discrete value.

Let us mention the process of performing the sentimental analysis by the RNN. A sentence is given as the input and is mapped into a temporal word sequence. Each word is codified into its own numerical value, and each hidden value is computed by its previous hidden value and its corresponding input value. A single output value is computed by its own hidden value; one among the three categories is decided, depending on a continuous output value. We need to discretize the continuous output value into one among the three values.

Let us make some remarks on the application of the RNN to the sentimental analysis. The sentimental analysis is the process of judging a statement as positive, neutral, or negative. A single output node is given, and its value is given as discrete one of the three cases, in applying the RNN to this task. A temporal word sequence is codified into numerical values, the hidden values are computed, and the output value is computed based on a single hidden value. As the alternative way, we consider a set of three output nodes, and the complete connection of the hidden nodes to the three output nodes.

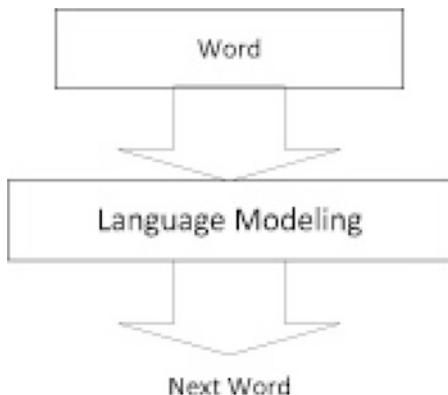
### 10.4.3 Entire Learning Process

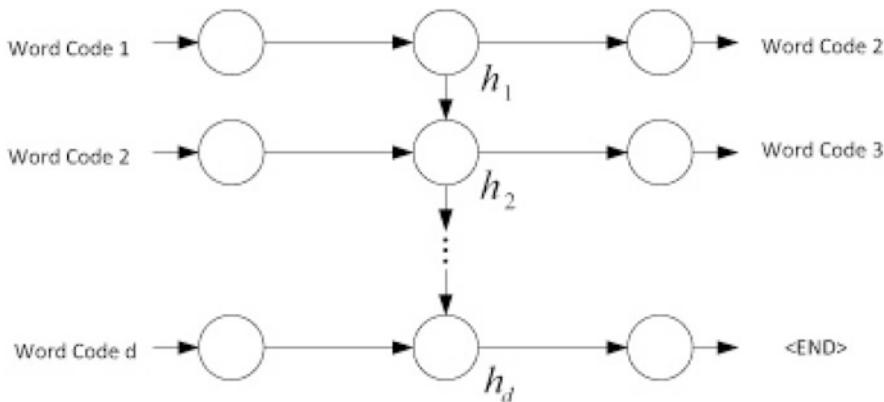
This section is concerned with the application of RNN to the language modeling. It is the process of predicting a word based on the previous words for constructing a sentence. The number of output nodes is identical to the number of the input nodes and the number of hidden nodes in the architecture. Each word is codified to its own numerical value, and the target value is the word which is put next to the word which is given in the input node. This section is intended to describe the process of applying the RNN to the language modeling.

The language modeling is illustrated with its functional view in Fig. 10.28. It is defined as the process of predicting a word based on its previous one or ones. The process of analyzing the correlation between the current word and the previous one is called language modeling. The language modeling is used for the speech recognition which maps a speech signal into a text. The RNN is applied to this task.

The architecture of RNN which is applied to the language modeling is illustrated in Fig. 10.29. The connection between the input layer and the hidden layer and the connection between hidden layer and the output layer are given as one-to-one type. Each input value is given as a numerical value which indicates a word as its code, and each output value is given as a numerical value which also indicates a word as

**Fig. 10.28** Functional view of language modeling





**Fig. 10.29** RNN architecture for language modeling

its code. The RNN is applied to the language modeling which predicts the next word to the previous words by means of the hidden values. It is assumed that the word sequence which is given as a temporal input value sequence is a sentence.

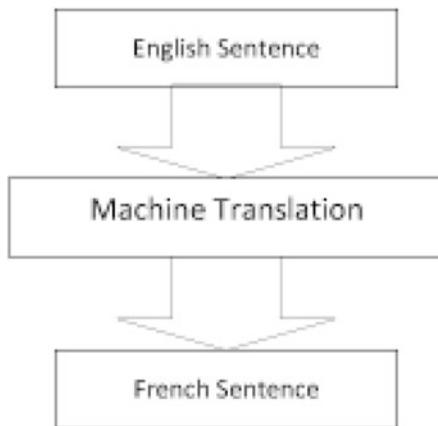
Let us mention the process of predicting the next word to the given word by the RNN. It is trained with sample sentences for optimizing the weights. An incomplete sentence or a word sequence is given as numerical values, and the output values are computed through the hidden values. The output values are decoded into corresponding words as the predicted words which follow the input words. The language modeling is used for presenting the most probable word which follows the incomplete sentence which is typed by a user.

Let us make some remarks on the application of RNN to the language modeling. It is the process of predicting a word based on its previous words. Each input node corresponds to an output node by means of a hidden node in the architecture. A word sequence is given as an incomplete sentence, and its next word to the word which corresponds to the last input node. The language modeling is expanded into the machine translation in applying the RNN.

#### 10.4.4 Machine Translation

This section is concerned with the application of RNN to the machine translation. It is the process of translating a sentence or a text in a particular natural language into one in another natural language by a program. The dual RNNs are used for the machine translation; one is for modeling the sentence in the original language, and the other is for modeling one in the translated language. A single sentence which is given as a temporal word sequence is translated into one in another language which

**Fig. 10.30** Functional view of machine translation



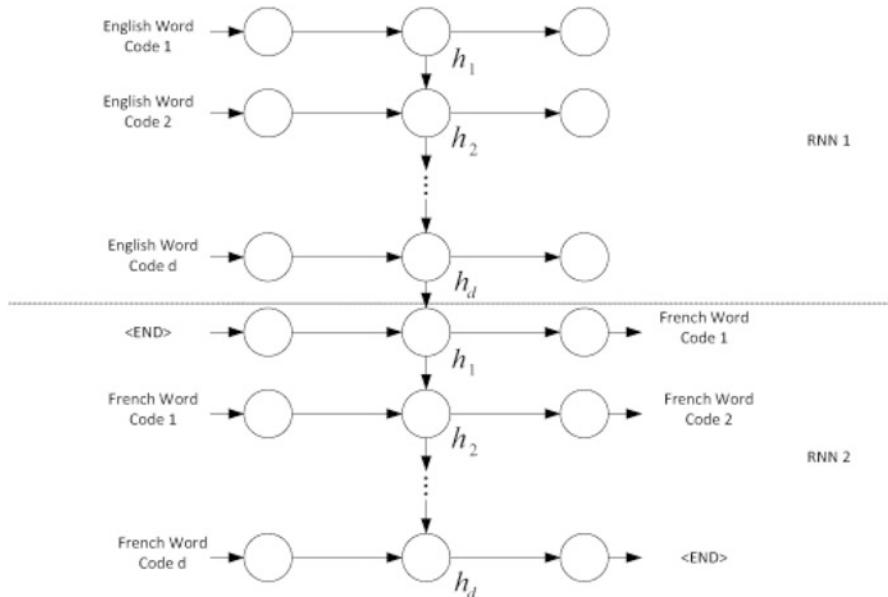
is also given as a temporal word sequence. This section is intended to describe the application of the dual RNNs to the machine translation.

The machine translation is illustrated with its functional view in Fig. 10.30. The machine translation is defined as the process of mapping a sentence which is written in a natural language into one which is written in another natural language. As an instance of machine translation, an English sentence is translated into a French sentence automatically by the system. The RNN is applied to this task. The pairs, each of which consists of an English sentence and a French sentence, are prepared as sample examples.

The architecture of dual RNNs which are applied to the machine translation is illustrated in Fig. 10.31. One is assigned to the language modeling in English, and the other is assigned to one in French. The training examples are the pairs, each of which consists of an English sentence and a French sentence. Each sentence is viewed into a temporal word sequence, and each word is codified into a numerical value in training the dual RNNs. A French sentence is generated from a English sentence by the trained RNNs.

Let us mention the process of translating an English sentence into a French sentence using the RNN. The English sentence is given as a word sequence, and each word is codified into its own numerical vector. The first French word is predicted based on the words in the English sentence. Subsequent French words are generated by the feedback of the predicted French words, together with the English words. The final output of this machine translation is a French sentence.

Let us make some remarks on the application of RNN to the machine translation. In this section, the machine translation is specified into the process of mapping an English sentence into a French sentence. The dual RNNs exist in applying so; one receives an English word sequence and the other receives a French word sequence in the training period. In the second RNN, a French word is predicted to the English word, and feedbacked as an input word, in the second RNN. We consider applying



**Fig. 10.31** RNN architecture for machine translation

the RNN to the translation of source code in a programming language into one in another programming language.

## 10.5 Summary and Further Discussions

This section is concerned with the summary and the further discussions on what is studied in this chapter. The basic connections of nodes such as the feedforward connection and the recurrent connection are studied as the background for understanding the recurrent networks. Some models of recurrent neural networks such as the basic model, the LSTM, and the variants are surveyed. The design of RNN in applying it to some tasks is mentioned. This section is intended to discuss further what is studied in this chapter.

Let us consider the combination of the recurrent neural networks and the MLP. In the basic version of the recurrent neural networks, the hidden nodes are connected with their feedback; the current value of a hidden node is computed by its previous value. Instead of all hidden units, feedback connections are allowed to some hidden nodes. We consider some hidden layers with their feedback connections and the others with only feedforward connections. In the future research, we consider designing more advanced recurrent neural networks, allowing some feedforward connections.

Let us mention the evolutionary computation as an optimization technique based on the natural evolution. There are four kinds of evolutionary computation: genetic algorithm, genetic programming, evolutionary strategy, and evolutionary programming; however, we mention only genetic algorithm as the most popular type. The process of encoding values into a bit string and decoding it into them is defined for applying the genetic algorithm to the optimization. In each iteration, offsprings are generated by making crossover of two-bit strings and mutating a bit string, evaluate fitness of the bit strings in the population, and select some bit strings based on their fitness. We may consider using the genetic algorithm for optimizing the weights of the recurrent neural networks.

Let us consider the evolutionary recurrent neural networks where the weights are optimized by the genetic algorithm. In the recurrent neural networks, the weights which are associated with the recurrent connections in the hidden layer are optimized as well as ones between layers. When the genetic algorithm is applied, the weights which are given as real values are encoded into binary values which are called bit strings. If the evolutionary strategy or the evolutionary programming is adopted, it does not require to encode the weights into a bit string. The merit of the evolutionary computation is to avoid the local minima, but it takes much more time for optimizing the weights.

Let us consider introducing the evolutionary computation for optimizing the weights in the LSTM. We mentioned above the application of the evolutionary computation to the weight optimization in the basic recurrent neural networks. The more complicated circuit inside each hidden node of LSTM is the difference from the basic recurrent neural networks. The evolutionary computation is applied to optimization of the four types of weights. We consider applying the evolutionary computation to optimize some weights and applying the gradient descent to optimize others.

## Reference

1. L. Zhang, G. Zhu, L. Mei, P. Shen, S. A. A. Shah, and M. Bennamoun, attention in convolutional LSTM for gesture recognition. *Advances in neural information processing systems*, 31, 2018.

# Chapter 11

## Restricted Boltzmann Machine



This chapter is concerned with the RBM for restoring the input. The Hopfield network was intended as the associative memory for restoring the input vector, before inventing the RBM. There are two kinds of variables in the RBM: the visible variables which are given as input variables and the hidden variables which are given for restoring the input. The stacked version of multiple RBMs which is called belief networks is a kind of deep neural networks. This chapter is intended to describe the RBM, together with the stacked version with respect to its learning process.

This chapter is composed of five sections, and we overview what is studied in this chapter in Sect. 11.1. In Sect. 11.2, we study the associative memory which is aimed for restoring the input. In Sect. 11.3, we study the basic version of RBM. In Sect. 11.4, the basic version is expanded into advanced versions. In Sect. 11.5, we discuss further what is studied in this chapter.

### 11.1 Introduction

This section is concerned with the overview of RBM as another deep neural network. The task of RBM is the associative memory which restores an incomplete data item into its complete one. There are two layers in the architecture, the visible layer and the hidden layer. The basic RBM may be expanded into a deep learning algorithm by stacking multiple RBMs. This section is intended to overview the RBM as the introduction to this chapter.

Let us explain the associative memory in the functional view. The associative memory is defined as the task which restores an incomplete input into its completed one. The input and the output are defined as identical dimensional vectors in this task. The associative memory is applied to the classification task by retrieving one among the training examples to the incomplete novice input. The Hopfield networks are designed as the traditional approach to the associative memory.

There are two types of variables in implementing the RBM: visible variables and hidden variables. The former is the input variables which should be restored, and the latter is the intermediate ones for restoring the input values. Each connection between a visible variable and a hidden variable is associated with its own weight. The learning process of RBM is to optimize the weights for making the computed visible values identical to the original ones. In applying the RBM to a real task, we are not interested in the hidden values but in restoring the input values.

There are two input types in applying the RBM. It is assumed that both input values are identical to each other by the weights which are optimized completely between the visible layer and the hidden layer. The input values which are given initially to the visible layer are called target inputs, and ones which are computed by product of the hidden values and the weights are called computed inputs. The direction of training the RBM is to minimize the error between the target input values and the computed ones. The target input values and the computed ones correspond to the incomplete input and complete input, respectively.

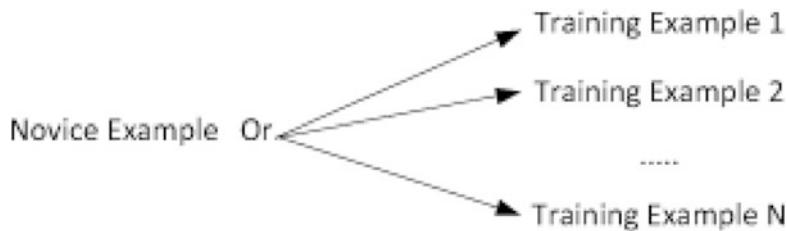
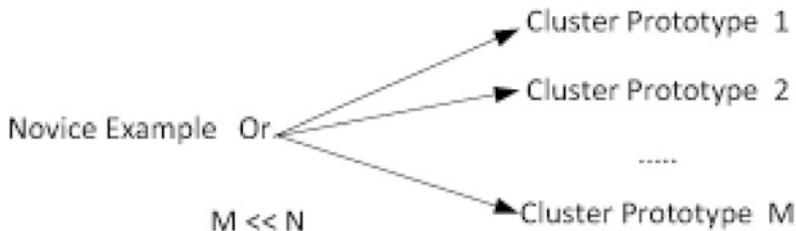
Let us mention what is intended in this chapter. We understand the associative memory as a task and some neural networks models for it. We understand the single RBM with respect to its architecture, its computation process, and its learning process. We understand the combination of multiple RBMs for implementing a deep learning algorithm. This chapter is intended to study the basic model and the advanced model of RBM.

## 11.2 Associative Memory

This section is concerned with the associative memory as the learning paradigm of RBM. In Sect. 11.2.1, we describe the input restoration as the basic concept of the associative memory. In Sect. 11.2.2, we study the associative MLP with its identical input and output. In Sect. 11.2.3, we study the early neural networks which follows the associative memory, called Hopfield neural networks. In Sect. 11.2.4, we study the Boltzmann machine as the previous model of the RBM.

### 11.2.1 Input Restoration

This section is concerned with the input restoration which is called associative memory. It is the process of restoring an incomplete input into its completed one. There are two kinds of inputs: the target input which is initially given and the computed input which is computed by the initial input. In the associative memory, the output is defined identically to the input; the dimension of the input vector is identical to that of the output vector. This section is intended to specify the task of associative memory in the functional view.

**Fig. 11.1** Individual input restoration**Fig. 11.2** Input clustering restoration

The restoration of the input values into one among the training examples is illustrated in Fig. 11.1. The  $N$  training examples are gathered in advanced like the supervised and the unsupervised learning. A novice example is notated by  $\mathbf{x}$ , and the training examples are notated by  $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N$ . A novice example,  $x$ , is restored into one among the training examples,  $\mathbf{x}_i$ , as expressed by Eq. (11.1):

$$\mathbf{x} \rightarrow \mathbf{x}_i \quad (11.1)$$

This kind of restoration is viewed as the process of retrieving a training example which is most relevant to a novice example.

The restoration of the input values into one among the cluster prototypes is illustrated in Fig. 11.2. The group of the  $N$  training examples is segmented into  $M$  clusters, depending on their similarities. The  $M$  cluster prototypes are notated by  $\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_M$ . The restoration of the input vector,  $\mathbf{x}$ , into a cluster prototype is expressed by Eq. (11.2):

$$\mathbf{x} \rightarrow \mathbf{c}_i \quad (11.2)$$

If there are too many training examples, we may consider the hierarchical restoration where an input vector is restored into one among training examples by means of the input cluster restoration.

The process of classifying a data item by the associative memory. The training examples which labeled with their own category are prepared. A novice vector is given as the input, and it is restored into one of the training examples. The label of a

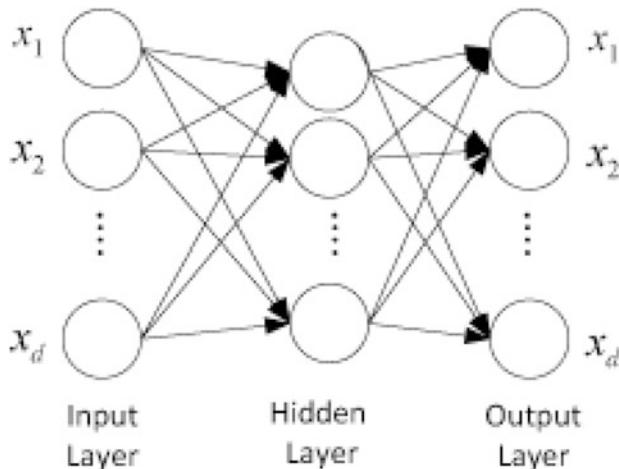
novice item is decided by the label of the training example into which it is restored. This classification type is viewed as the process of classifying it by the 1 nearest neighbor.

Let us make some remarks on the associative memory which is described in the functional view. The input vector is restored into one among the training examples. The training examples are clustered in advance, and the input vector may be restored into one among the cluster prototypes. A novice input vector is classified by restoring it into one among the labeled training examples, and its label is decided by the label of the restored training example. Before studying the RBM, we will study the Hopfield networks as the early neural networks which are designed for the associative memory.

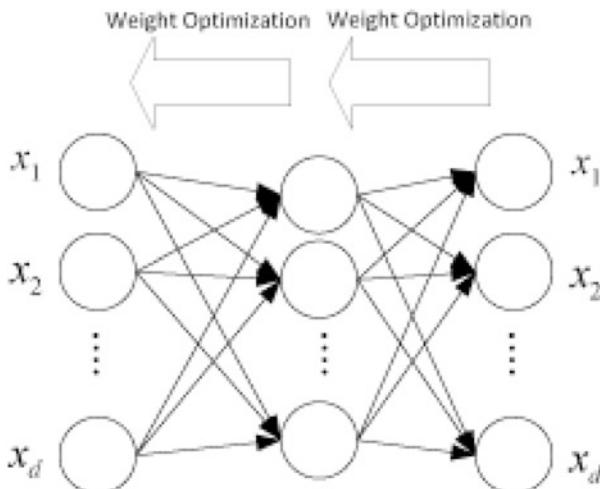
### 11.2.2 Associative MLP

This section is concerned with the MLP version, which is specialized for the associative memory, called associative MLP. In the previous section, we studied the associative memory which restores an incomplete input into its complete one in the functional view. The MLP is designed with the identical number of nodes in the input layer and the output layer, and an arbitrary number of nodes in the hidden layer. Each training example which is used for training the MLP is labeled with its input vector by itself. This section is intended to describe the MLP version for the associative memory.

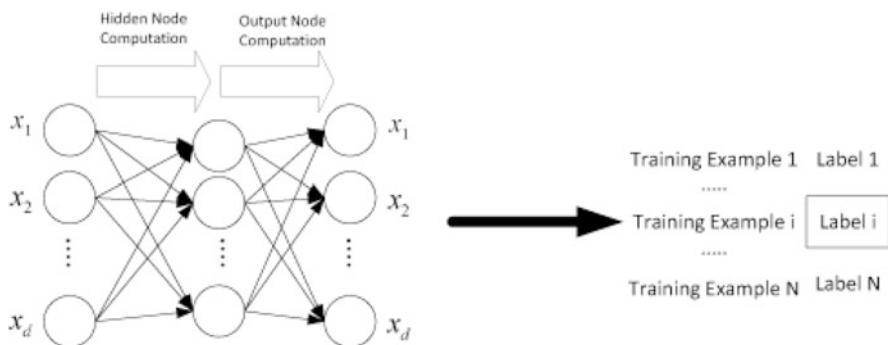
The architecture of the associative MLP is illustrated in Fig. 11.3. There are three layers in this MLP: the input layer, the hidden layer, and the output layer. The



**Fig. 11.3** Associative MLP architecture



**Fig. 11.4** Associative MLP learning



**Fig. 11.5** Associative MLP classification

input and the output are defined identically in the associative MLP: the number of input nodes is identical to the number of output nodes. The learning process of the associative MLP is to optimize the weights for restoring the incomplete input into its complete form. The number of hidden nodes is arbitrary like any version of MLP.

The direction of training the associative MLP is illustrated in Fig. 11.4. We already studied the process of training the MLP in Chap. 9. The two weight matrices are optimized as the learning process. If the number of input nodes and output nodes is  $d$ , and the number of hidden nodes is  $h$ , the weight matrix sizes are  $d \times h$  and  $h \times d$ . The direction of optimizing the weights is from the output layer to the input layer.

The process of classifying a data item using the associative MLP is illustrated in Fig. 11.5. The associative MLP is designed for performing the associative memory. A novice input vector is restored into one among the training examples, and the

label of a novice input vector is decided by one of the restored training examples. If it is restored not exactly into one among the training examples, the training example which is almost matching with the computed one is retrieved. The version which is covered in Chap. 9 is used as the classification model, rather than the associative MLP.

Let us make some remarks on the associative MLP which is the version for performing the associative memory. The number of input nodes is identical to the number of output nodes in designing the associative MLP. The weights between the input layer and the hidden layer and between the hidden layer and the output layer are optimized as the learning process of the associative MLP. It is used for restoring the input vector into one among the training examples. Other machine learning algorithms are modified into approaches to the associative memory by defining the output identically to the input.

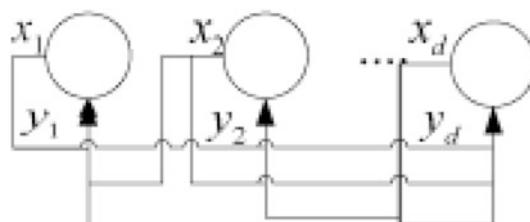
### 11.2.3 Hopfield Networks

This section is concerned with the Hopfield networks as the typical neural networks for the associative memory. In the previous section, we studied the MLP version which is specialized for the associative memory. In this section, we study the Hopfield networks with a single layer within which all nodes are connected with each other; each node in the layer is connected with itself and the others. Each node value is computed by applying an activation function to the product of its previous value and ones of the others. This section is intended to describe the Hopfield networks as the neural networks for the associative memory.

The architecture of Hopfield networks is illustrated in Fig. 11.6. Even if the Hopfield networks have a single layer, each node is associated with both the input value,  $x_i$ , and the output value,  $\hat{x}_i$ . Each node is connected with itself and the others; the connection of all nodes is complete in the architecture of the Hopfield networks. If the number of nodes is  $d$ , the number of connections in the Hopfield Networks is  $d^2$ . The architecture of Hopfield networks is viewed as one of the Perceptron where the number of both input nodes and output nodes is  $d$ .

Let us mention the computation process of the Hopfield networks. The architecture of Hopfield networks is interpreted into one of Perceptron where the input

**Fig. 11.6** Hopfield network architecture



vector and the output vector are identical to each other, and both the input vector and the output vector are assumed as binary vectors which consist of zero or one. The  $d$  nodes are denoted by  $x_1, x_2, \dots, x_d$ , and  $d \times d$  weights are viewed as a matrix which is expressed in Eq. (11.3):

$$\mathbf{W} = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1d} \\ w_{21} & w_{22} & \dots & w_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ w_{c1} & w_{c2} & \dots & w_{cd} \end{pmatrix}. \quad (11.3)$$

The output value which is given as a binary value is computed by Eq. (11.4):

$$\hat{x}_i = \begin{cases} 1 & \text{if } \sum_{j=1}^d w_{ji} \cdot x_j \geq 0.5 \\ 0 & \text{otherwise} \end{cases}. \quad (11.4)$$

The generalization of the Hopfield networks is to restore a novice input vector into one among the training examples.

Let us mention the learning process of Hopfield networks. Each training example is assumed as a binary vector which consists of zero and one, and each training example is labeled with itself. There are two kinds of input vector: the target input vector,  $\mathbf{x}_i = [x_{i1} \ x_{i2} \ \dots \ x_{id}]$ , and the computed input vector,  $\hat{\mathbf{x}}_i = [\hat{x}_{i1} \ \hat{x}_{i2} \ \dots \ \hat{x}_{id}]$ . The weight matrix is initialized at random. The rule of updating the weight,  $w_{ji}$ , is derived as Eq. (11.5):

$$w_{ji} \leftarrow w_{ji} + \eta x_i (x_j - \hat{x}_j) \quad (11.5)$$

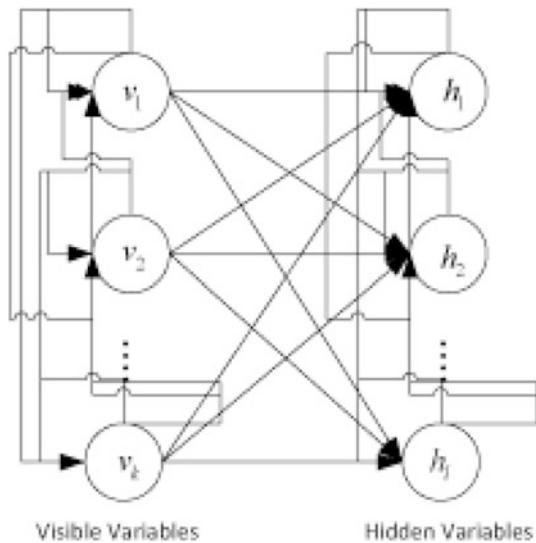
The difference of Hopfield networks from the Perceptron is to replace the output value by the input value.

Let us make some remarks on the Hopfield networks with respect to its computation and learning process. Each node is connected to itself and the others in the architecture of Hopfield networks. There are two kinds of input values: the target input which is one among the training examples and the computed input which is computed by Eq. (11.4). The weights are optimized for minimizing the error between the target input and the computed input. We consider upgrading the Hopfield networks into the version which treats with continuous input values.

### 11.2.4 Boltzmann Machine

This section is concerned with the Boltzmann machine which is the previous model to the RBM. As the start of studying the Boltzmann machine, we define the two

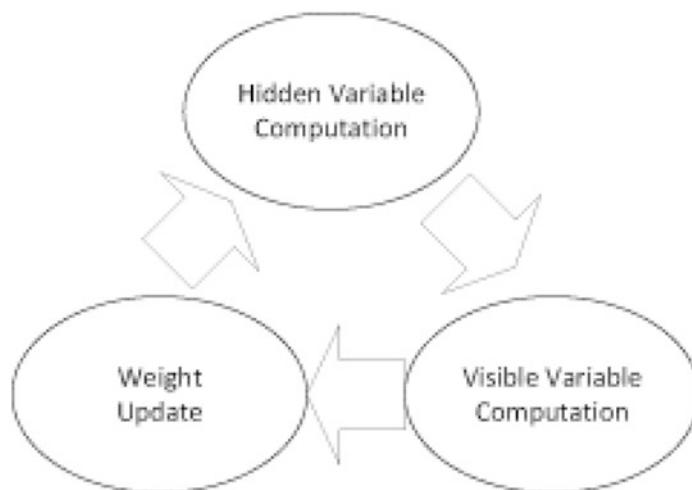
**Fig. 11.7** Boltzmann machine architecture



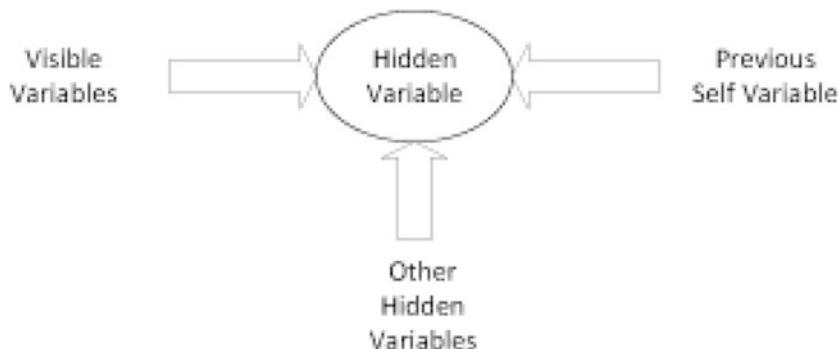
kinds of variables: visible variables and hidden variables. In the Boltzmann machine, the connection between the visible variables and the hidden variables is complete, and the connection within each of them is also complete. The Boltzmann machine evolves into the RBM by taking the only complete connection between the two kinds of variables. This section is intended to describe the Boltzmann machine as the precedent of the RBM.

The architecture of the Boltzmann machine is illustrated in Fig. 11.7. The architecture in the left part of Fig. 11.7, where each visible node is connected with itself and the others, is the same to the architecture of Hopfield networks. The architecture in the middle part of Fig. 11.7 with the full connection between the visible layer and the hidden layer is the same to the architecture of the Perceptron which was covered in Chap. 9. Each hidden node is connected in the right part in Fig. 11.7 like the Hopfield networks. The architecture of the Boltzmann machine is viewed as the integration of the two architectures of Hopfield networks and the architecture of the Perceptron.

The frame of training the Boltzmann machine is illustrated in Fig. 11.8. The weights which are associated with the connections and the hidden node values are initialized with random values or zeros. The values of visible values are given as the input, and the hidden node values are computed by product of visible values and weights and product of previous hidden values and weights. The visible values are computed by the product of the hidden values and weights and the product of previous visible values and weights, and the weights are updated for minimizing the error between the initial visible values and the computed visible values. The learning of the Boltzmann machine proceeds by circulating the three phases until the minimum error between the initial visible values and the computed ones.



**Fig. 11.8** Boltzmann machine learning cycle



**Fig. 11.9** Hidden variable computation in Boltzmann machine

The frame of computing the hidden values is illustrated in Fig. 11.9. Because it depends on the previous values, it is necessary to initialize the hidden values with zeros. Each hidden node is connected with the visible nodes and connected with the other hidden nodes and itself. The hidden node value is computed by addition of the product of visible values and weights and the product of previous hidden values and weights. The connection from each hidden node with other hidden nodes is removed in the RBM.

Let us make some remarks on the Boltzmann machine as another neural network for the associative memory. The connection between the visible layer and the hidden layer and the connection within the visible layer or the hidden layer are complete in the architecture. The learning process of the Boltzmann machine is to update the weights between the visible layer and the hidden layer for minimizing the error

between the computed visible values and the target visible values. Each hidden node value is dependent on the visible values, other hidden node values and its previous value. The demerit of the Boltzmann machine is to involve too many weights in computing the hidden node values and the visible node values.

### 11.3 Single RBM

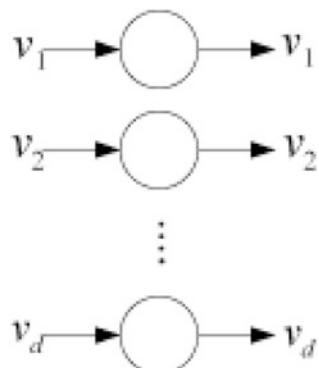
This section is concerned with the basic version of RBM. In Sect. 11.3.1, we study its architecture of single RBM. In Sect. 11.3.2, we study the process of computing the hidden values from the visible values. In Sect. 11.3.3, we study the process of training the RBM with the training examples. In Sect. 11.3.4, we modify the RBM as a classification model.

#### 11.3.1 Architecture

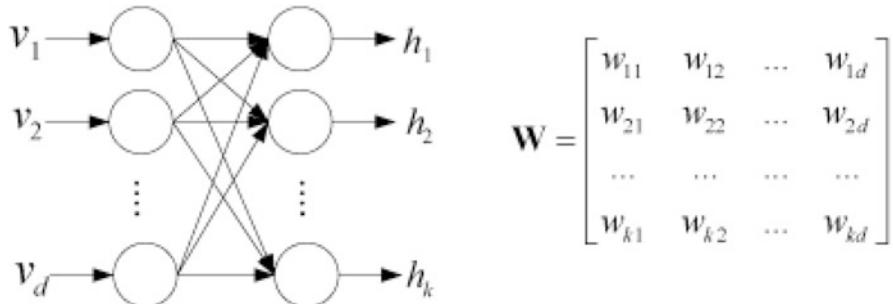
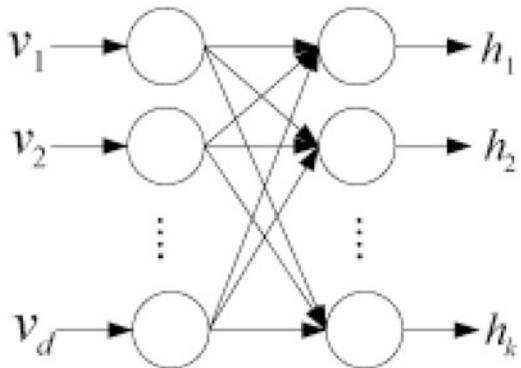
This section is concerned with the RBM architecture. In Sect. 11.2.4, we studied the architecture and the learning frame of Boltzmann machine. In this section, we study the RBM architecture where the connections among the hidden nodes and the connections among the visible nodes are removed. Even if the RBM looks similar as the Perceptron with respect to its architecture, the roles of the two layers are different in both neural networks. This section is intended to describe the RBM architecture with only connections between the two layers.

The visible layer in the RBM architecture is presented in Fig. 11.10. The visible values are denoted as a vector,  $\mathbf{v} = [v_1 \ v_2 \ \dots \ v_d]$ , with the  $d$  dimensionality. Each visible node is denoted by. The visible values are transferred to the hidden layer in the basic RBM. They are used for computing the hidden values.

**Fig. 11.10** Visible variables in basic RBM



**Fig. 11.11** Hidden variables in basic RBM



**Fig. 11.12** Weight between visible and hidden in basic RBM

The visible layer and the hidden layer are illustrated in Fig. 11.11 as the architecture of the RBM. The hidden values are noted by a vector with the  $k$  dimensionality,  $\mathbf{h} = [h_1 \ h_2 \ \dots \ h_d]$ . Each hidden node is noted by  $h_i$ , and its value is computed by the visible values. The bidirectional connection between the visible layer and the hidden layer is the difference of the RBM from the Perceptron. The hidden values are used for computing the visible values.

The RBM architecture and the weight matrix are illustrated in Fig. 11.12. The connection between the visible layer and the hidden layer is complete; there are  $k \times d$  connections between them. The weight,  $w_{ji}$ , means the connection from the  $i$ th visible node to the  $j$ th hidden node. The weights which are involved in the RBM are viewed as a matrix as shown in Eq. (11.6):

$$\mathbf{W} = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1d} \\ w_{21} & w_{22} & \dots & w_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k1} & w_{k2} & \dots & w_{kd} \end{pmatrix}. \quad (11.6)$$

The weights are used for computing the visible values as well as the hidden values.

Let us make some remarks on the RBM architecture which consists of the visible layer and the hidden layer. The input values are given as the visible values, and there are two kinds of visible values: target visible values and computed visible values. An arbitrary number of hidden nodes are determined, and each node value is computed by applying an activation function to the net input of the visible values. The weights between the two layers are bi-directional for computing both the hidden node values and the visible node values. Various versions of RBM are derived depending on the activation function which is adopted for computing both kinds of node values.

### 11.3.2 Input Layer

This section is concerned with the process of computing hidden node values. In the previous section, we studied the entire architecture of the RBM which consists of the visible layer and the hidden layer. In this section, we study the process of computing the hidden values by the visible values. The visible values are computed after computing the hidden values. This section is intended to describe the process of computing the hidden values by the visible values.

The visible vector is initially given as the input vector for the RBM. The two kinds of visible vectors are defined as the target visible vector and the computed visible vector. The target visible vector which is initially given as the input vector and the computed visible vector which is computed by applying an activation function to the product of the weight matrix and the hidden vector are denoted by  $\mathbf{v} = [v_1 \ v_2 \ \dots \ v_d]$  and  $\hat{\mathbf{v}} = [\hat{v}_1 \ \hat{v}_2 \ \dots \ \hat{v}_d]$ , respectively. The input vector which is augmented with the target output vector is given as the visible vector in applying the RBM to the classification. The direction of training the RBM is to minimize error between the two kinds of visible vector.

The weight matrix between the visible layer and the hidden layer is initialized after presenting the visible vector. The number of visible nodes is  $d$ , and the number of hidden nodes is  $k$ . The  $k \times d$  weight matrix which is presented in Eq. (11.6) is initialized with random values around zero. If the weights are initialized as zero values, and the linear function is adopted as the activation function, the hidden values are zeros. To avoid the situation, the sigmoid function is adopted for computing the hidden values.

Let us mention the process of computing the hidden values in the RBM. The visible vector is initially given, and the weight matrix between the visible layer and the hidden layer is given as Eq. (11.6). The hidden value,  $h_i$ , is computed by Eq. (11.7):

$$h_i = \sum_{j=1}^d w_{ji} \cdot v_j. \quad (11.7)$$

The equation of computing the hidden vector is viewed as the product of the weight matrix and the visible vector as shown in Eq. (11.8):

$$\mathbf{h} = \mathbf{W} \cdot \mathbf{v} \quad (11.8)$$

The hidden vector is used for computing the visible vector with the weight matrix.

Let us make some remarks on the process of computing the hidden node values. There are two kinds of visible vector in the RBM: target visible vector and computed visible vector. The weights between the visible layer and the hidden layer are initialized at random, and the sigmoid function is used as an activation function. The hidden vector is computed by product of the weight matrix and the visible vector. In the RBM, the partial connection within the visible layer or the hidden layer may be allowed.

### 11.3.3 Learning Process

This section is concerned with the process of training the RBM with the training examples. Unlabeled examples are prepared as the training examples; it is assumed that each training example is labeled with its identical input. There are two kinds of visible vectors: target visible vector which is initially given by a training example and computed visible vector which is computed by the product of the transpose of the weight matrix and the hidden vector. The direction of training the RBM is to minimize the error between two kinds of visible vectors. This section is intended to describe the learning process of RBM.

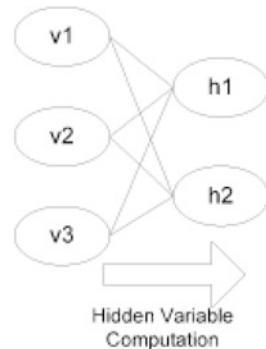
The computation of the hidden values from the visible values is illustrated in Fig. 11.13, as the first step of learning process. The weights between the input layer and the hidden layer are initialized at random, and the visible vector is initially given as the input vector,  $\mathbf{v}$ . The hidden vector,  $\mathbf{h}$ , is computed by the product of the weight matrix and the visible vector. The difference of the hidden vector from the output vector in the perceptron is to use it for computing the visible vector. The number of hidden nodes is set arbitrary; the number of hidden nodes is optimized for minimizing the training error.

The process of computing the visible vector by the hidden vector which are computed by the above process. The hidden vector is computed by the product of the weight matrix and the visible vector which is initially given. Each visible value,  $\hat{v}_i$ , is computed by Eq. (11.9):

$$\hat{v}_i = \sum_{j=1}^k w_{ji} \cdot h_j \quad (11.9)$$

The process of computing the visible vector,  $\hat{\mathbf{v}}$ , is viewed into the product of the weight matrix transpose and the hidden vector, as expressed in Eq. (11.10):

**Fig. 11.13** Hidden variable computation in RBM



$$\hat{\mathbf{v}} = \mathbf{W} \cdot \mathbf{h} \quad (11.10)$$

The direction of training the RBM is to minimize the error between the two vectors,  $\mathbf{v}$  and  $\hat{\mathbf{v}}$ .

Updating the weights between the visible layer and the hidden layer is illustrated in Fig. 11.14. The activation function and the loss function are assumed respective as the linear function and Equation (11.11):

$$E = \frac{1}{2}(\mathbf{v} - \hat{\mathbf{v}})^2. \quad (11.11)$$

The weight is updated by Eq. (11.12):

$$w_{ji} \leftarrow w_{ji} - \eta \frac{\partial E}{\partial w_{ji}}. \quad (11.12)$$

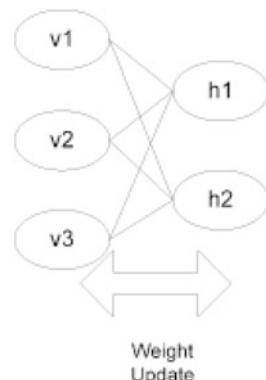
Equation (11.13) is derived from Eqs. (11.11) and (11.12) for updating the weight:

$$w_{ji} \leftarrow w_{ji} + \eta h_i(v_j - \hat{v}_j). \quad (11.13)$$

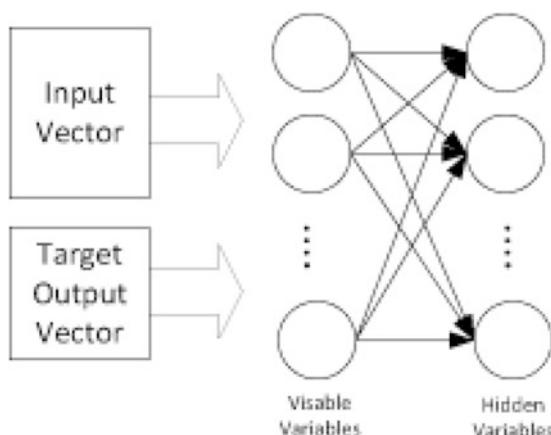
Updating the weights by Eq. (11.13) is repeated until their convergence.

Let us make some remarks on the process of training the RBM. The weights between the visible layer and the hidden layer are initialized at random, and the input vector is initially given as the visible vector. The hidden vector is computed by the visible vector, and the visible vector is computed again by the hidden vector. The weights are updated for minimizing the difference between the target visible vector and the computed visible vector. The process of training the RBM is viewed as the supervised learning where the input vector and target output vector are identical to each other.

**Fig. 11.14** Weight optimization in RBM



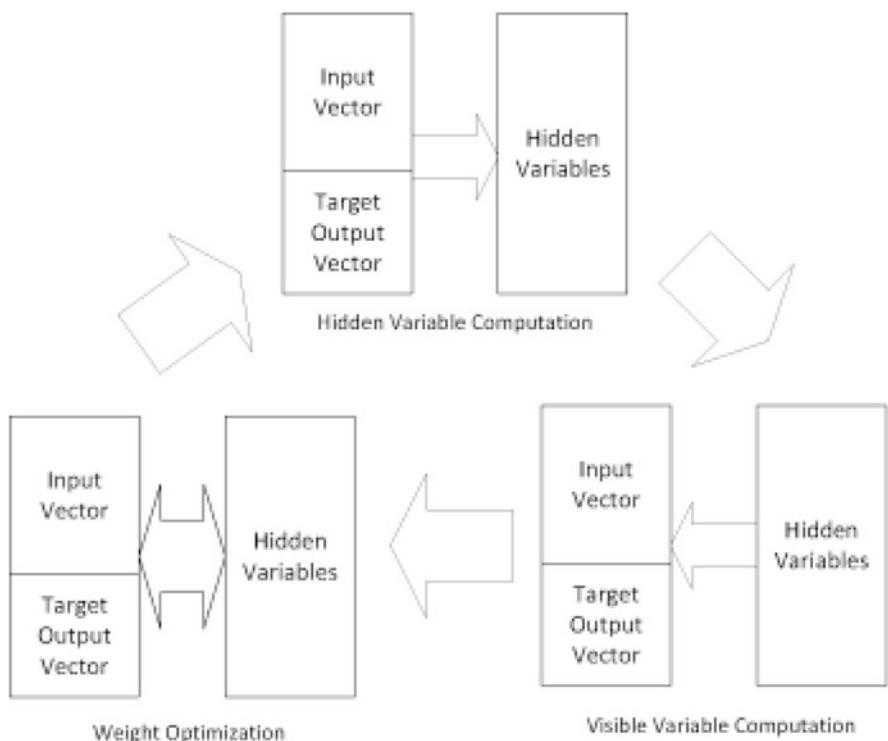
**Fig. 11.15** Architecture of RBM classification model



#### 11.3.4 Classification Model

This section is concerned with the application of the RBM to the classification task as a single model. The RBM is regarded as an associative memory model for restoring the input vector, until the previous section. In this study, the RBM is designed as a classifier; the input vector is augmented with its target output vector as the visible vector. The target visible vector is the augmentation of the input vector with the zero vector to an unlabeled novice input vector, and the augmented part in the computed visible vector is the label of the novice vector in the generalization process. This section is intended to describe the design of a single RBM for classifying a data item.

The architecture of RBM which is applied to the classification is illustrated in Fig. 11.15. If the RBM is applied to the associative memory, the input vector is assumed as an unlabeled item. If the RBM is applied to the classification, the input vector is augmented with its target output vector as the visible vector. If the input vector is a  $d$  dimensional vector, and the target output vector is a  $m$  dimensional



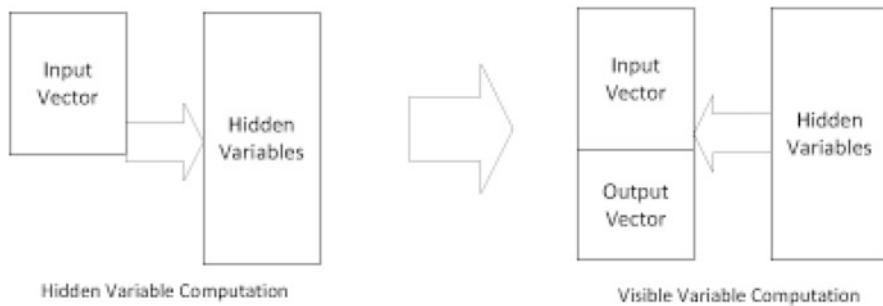
**Fig. 11.16** Learning process of RBM classification model

vector, the visible vector becomes a  $d + m$  dimensional vector. Both the input vector and zero output vector are given as the visible vector in the generalization.

The process of training the RBM which is designed for the classification task is illustrated in Fig. 11.16. The input vector which is augmented with its target vector is given as the visible vector for each training example. The hidden vector is computed from the visible vector, and the visible vector is computed with the hidden vector. The weights between the hidden layer and the visible layer are updated for decreasing the difference between the target visible vector and the computed one. The weights are optimized by iterating the cycle which is presented in Fig. 11.16.

The process of classifying a data item by the RBM as its generalization is illustrated in Fig. 11.17. The novice input is given as an only input vector, so it is augmented with zero output vector, as the visible vector. The hidden vector is computed by this visible vector, and the visible vector is computed again by the computed hidden vector. The augmented part in the computed visible vector is the answer to the novice input vector. The visible vector without the output vector is restored into one with it as the classification by the RBM.

Let us make some remarks on the RBM which is designed for classifying a data item. The visible vector is the augmentation of the input vector with its target output



**Fig. 11.17** Classification process in RBM

vector. The classification process is to compute the hidden vector by the visible vector which consists of the input vector and its zero vector, compute the visible vector again, and take the augmented part in the visible vector as the answer. The weights between the visible layer and the hidden layer are optimized for minimizing the error between the target visible vector and the computed visible vector. The RBM may be designed as a classifier by stacking serially multiple RBMs as the alternative way.

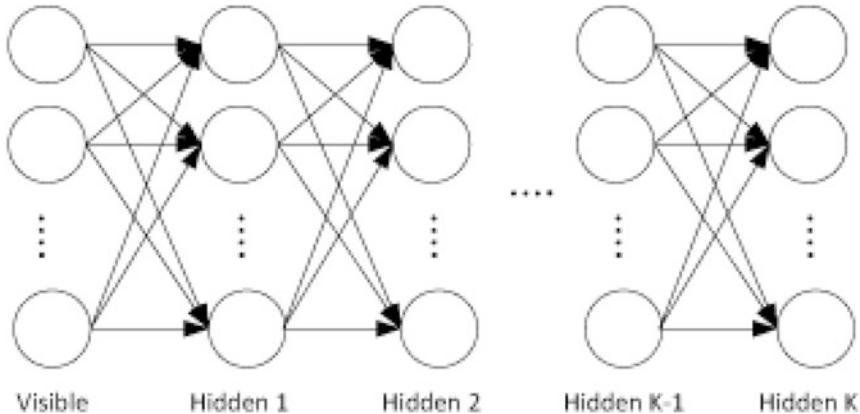
## 11.4 Stacked RBM

This section is concerned with the advanced models of RBM. In Sect. 11.4.1, we study the multiple stacked RBM which consists of multiple hidden layers. In Sect. 11.4.2, we study the application of multiple stacked RBM to the dimension reduction. In Sect. 11.4.3, we study the RBM which is applied to the output decoding. In Sect. 11.4.4, we mention the optimization of weights between the visible layers and the hidden layers by the genetic algorithm.

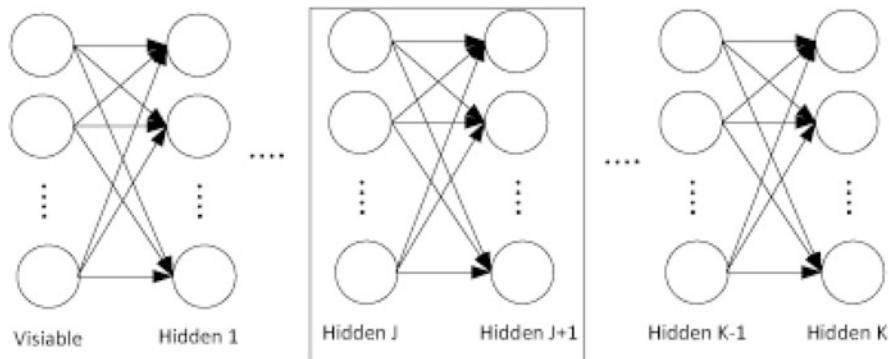
### 11.4.1 Multiple Stacked RBM

This section is concerned with the advanced version of RBM. In Sect. 11.3, we studied the single RBM with its only two layers: visible layer and hidden layer. From this section, the single RBM is expanded into multiple RBM. The relation between the hidden vector the visible vector is linear in the single RBM, whereas the relation between the last hidden vector and the visible vector is nonlinear in the multiple stacked RBM. This section is intended to describe the multiple stacked RBM as the advanced model.

The multiple stacked RBM is illustrated in Fig. 11.18. The single RBM which is studied in Sect. 11.3 is composed of the two layers: visible layer and hidden layer.



**Fig. 11.18** Architecture of multiple stacked RBMs

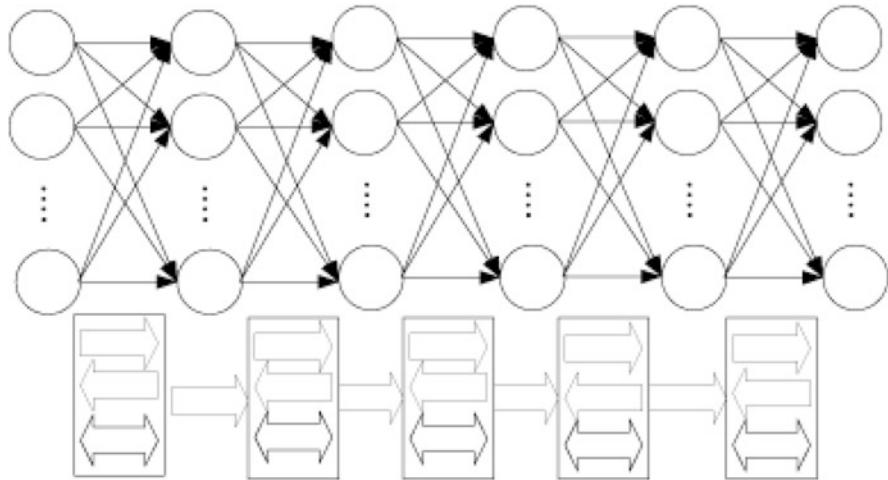


**Fig. 11.19** RBM  $J$  and RBM  $(J + 1)$

The multiple stacked RBM is composed of a visible layer and multiple hidden layers. The final hidden values are computed by means of intermediate hidden values in the forward direction, and the visible values are computed in the backward direction. The linear function is adopted as the activation function in the RBM for the simplicity.

In Fig. 11.19, we focus on the hidden layer,  $J$  and the hidden layer,  $J + 1$ , as the intermediate hidden layers. The most left layer is the visible layer and the second most left layer is the first hidden layer. The  $i$ th hidden node in the  $J + 1$  is denoted by  $h_i^{J+1}$ . Its value is computed by Eq. (11.14):

$$h_i^{J+1} = \sum_{r=1}^{|J|} w_{ir} h_r^j. \quad (11.14)$$



**Fig. 11.20** Learning process in multiple stacked RBMs

The hidden node value which is computed by Eq. (11.14) is used for computing the hidden node values in the next layer.

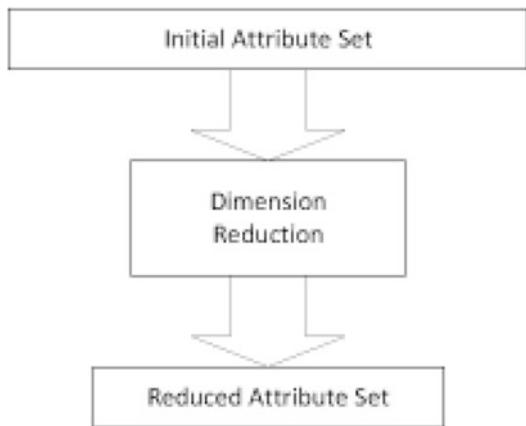
The direction of updating the weights in the multiple stacked RBM is illustrated in Fig. 11.20. The weights are initialized at random, and the hidden node values are computed in the forward direction, from the visible layer to the final hidden layer. The visible values are computed in the backward direction from the last hidden layer to the visible layer. The weights are updated between the visible layer and the first hidden layer; the weight updates are propagated from the visible layer to the last hidden layer. The process of deriving the rule of updating the weights is left as an exercise.

Let us make some remarks on the multiple stacked RBM as the advanced version. It is composed of a single visible layer and multiple hidden layers. The hidden node values in the current layer are computed by ones in the previous layer. The hidden node values in the last layer are computed in the forward direction, the visible values are computed in the backward direction, and the weights are updated in the forward direction. This version of RBM is used for solving the nonlinear relation in restoring the input values.

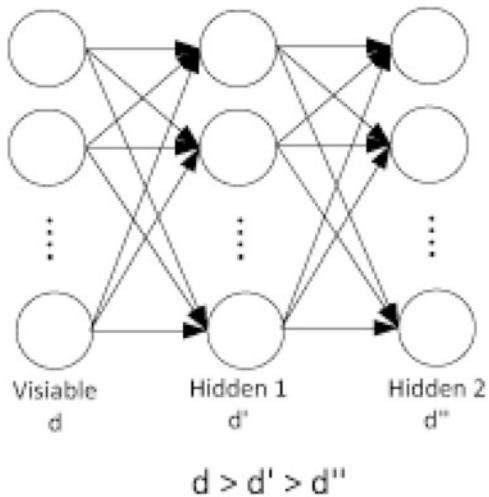
### 11.4.2 Input Encoding

This section is concerned with the version of RBM which is applied for the input encoding. It is the process of mapping the initial input vector into a vector with its different dimension. The initial input vector is set as a visible vector, and the dimension of hidden vector is set less than the dimension of visible vector. The fact

**Fig. 11.21** Dimension reduction



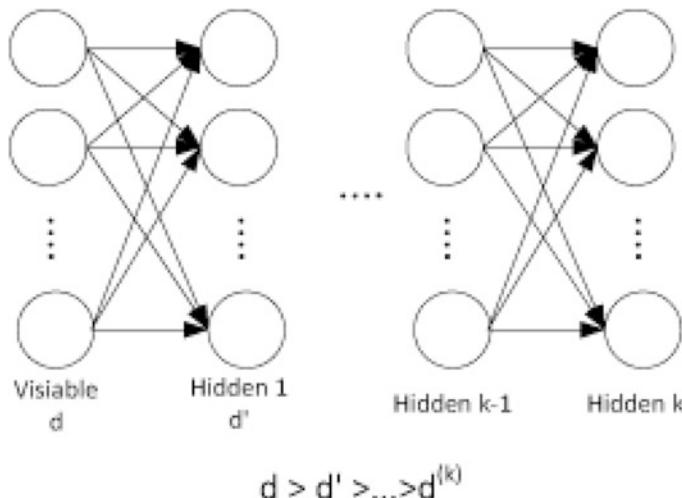
**Fig. 11.22** Dimension reduction by two RBMs



that the hidden values are taken as the output characterizes the RBM which is used for the input encoding. This section is intended to describe the use of RBM for the input encoding.

The dimensional reduction is illustrated with the functional view in Fig. 11.21. It is the process of mapping a numerical vector into another vector with its less dimensionality. The attributes are listed given as the input, and a list of new attributes with its smaller number is the output. The SVD (singular value decomposition), the PCA (principal component analysis), and the ICA (independent component analysis) are typical schemes of reducing the dimensionality. A single RBM or multiple RBMs are applied to the dimensional reduction.

The dual RBM for reducing the dimensionality is illustrated in Fig. 11.22. The initial input vector is set as the visible vector, and the weights between the visible layer and the first hidden layer are optimized. The hidden values in the first hidden

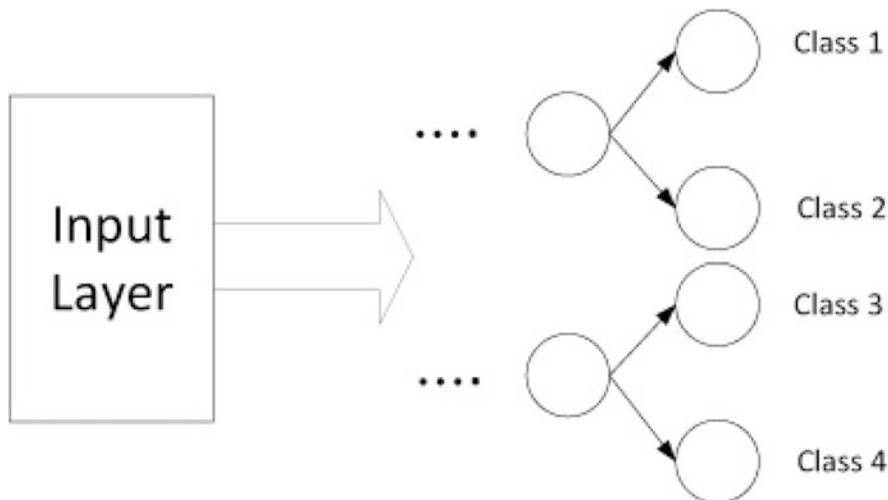


**Fig. 11.23** Dimension reduction by multiple RBMs

layer are given as the new visible values, and the weights between the first hidden layer and the second hidden layer are optimized. The hidden values in the second hidden layer are computed by means of ones in the first hidden layer from the input values in the generalization. The hidden vector in the second hidden layer is the vector with its reduced dimensionality which is mapped from the input vector.

The multiple RBMs for mapping the input vector into one with its lower dimensionality are illustrated in Fig. 11.23. The initial input vector is set as the visible vector, and it is encoded into the final hidden vector as the mapped one. In the feedforward direction, the hidden vector is computed layer by layer, and in the backward direction, the visible vector is computed from the last hidden layer to the input layer. The weights are updated in the feedforward direction, layer by layer. The final hidden vector is computed from the visible vector with the optimized weights as a vector with its reduced dimensionality in the generalization.

Let us make some remarks on the application of RBM for encoding the input vector into one with its lower dimensionality. The dimension reduction is the process of encoding the input vector into a vector with its lower dimensionality. The dual RBMs are applied to the dimensional reduction; the final hidden vector is the vector which is encoded from the input vector. In applying the multiple stacked RBMs to the dimension reduction, the final hidden vector is computed in the forward direction, the visible vector is computed in the backward direction, and the weights are updated in the forward direction. The pooling and the convolution are used for reducing the dimensionality.



**Fig. 11.24** Output specification in deep learning

### 11.4.3 Output Decoding

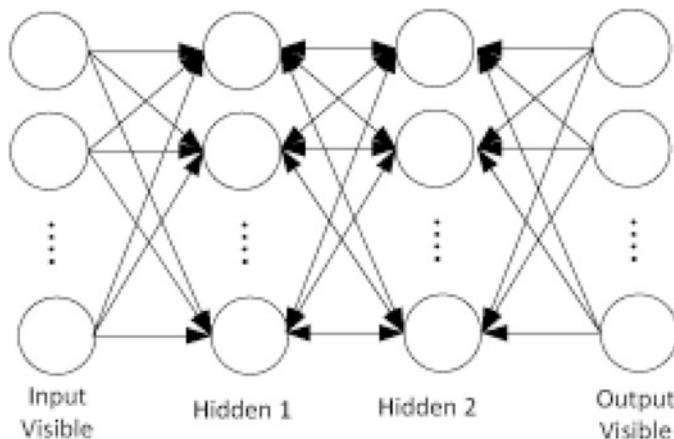
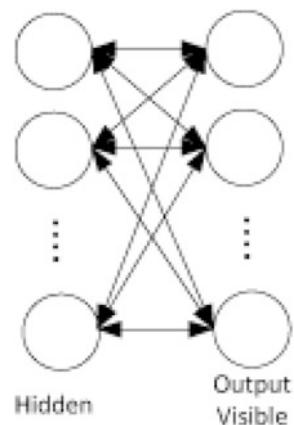
This section is concerned with the application of RBM to the output decoding. It is the process of mapping the encrypted code into its original code. The visible layer and the hidden layer are swapped with each other in applying a single RBM to the output decoding. The first layer and the last layer are visible layers in applying the multiple stacked RBM. This section is intended to describe the version of RBM which is applied to the output decoding.

The output specification as an instance of output decoding is illustrated in Fig. 11.24. The output specification is the process of mapping an abstract output value into its specific one. The predefined classes are defined as class 1, class 2, class 3, and class 4, and there are two nodes in the previous layer as shown in Fig. 11.24. The top node is connected to class 1 and class 2, and the bottom node is connected to class 3 and class 4. The two layers are viewed as a hierarchical classification system.

The single RBM which is applied to the output decoding is illustrated in Fig. 11.25. The left layer is given as the hidden layer and the right layer is given as the visible layer. The hidden vector is computed with the target output vector, and the output vector is computed with the computed hidden vector, and the weights are updated for minimizing the difference between the target output vector and the computed one in the learning process. In the generalization, the hidden vector is given as the input vector, and the output vector is computed by them. This version of RBM looks like the supervised learning, instead of the associative memory.

The compound of the input encoding and the output decoding with the multiple stacked RBM is illustrated in Fig. 11.26. In the first RBM, the visible values are

**Fig. 11.25** RBM architecture for output decoding



**Fig. 11.26** RBM architecture for input encoding and output decoding

input values, and the hidden values are ones which are encoded from the input values, and in the last RBM, the visible values are the output values and the hidden values are ones from which the output values are decoded. The weights in the first RBM and those in the last RBM are updated in the learning process. There are two ways of updating the weights in the intermediate RBMs; the weights are updated from the first RBM in the forward direction, and the weights are updated from the last RBM in the backward direction. This mode of the input encoding and the output encoding is used as a deep operation.

Let us make some remarks on the application of RBM to the output decoding. The output specification is the process of mapping the category in the abstract level into one in the specific level. In applying a single RBM to the output decoding, the hidden vector is one before being decoded, and the visible vector is the final output vector. The multiple stacked RBM is applied to the compound of the input encoding

and the output decoding. The existing machine learning algorithm is modified into its deep version by putting its multiple stacked versions in front of it.

### 11.4.4 Evolutionary RBM

This section is concerned with the evolutionary version of RBM. In the previous sections, the weights between the visible layer and the hidden layer are optimized by the gradient descent. It is replaced by the genetic algorithm which is an instance of evolutionary computation in optimizing the weights. The advantage of the genetic algorithm over the gradient descent is to avoid falling a local minimum in the error surface. This section is intended to describe the application of the genetic algorithm to the weight optimization in the RBM.

The fitness evaluation measure should be defined as the first step of applying the genetic algorithm to the weight optimization in the RBM. The direction for optimizing weights is to minimize the difference between the target visible vector and the computed visible vector. The fitness evaluation measure is defined as the reverse difference between the two kinds of visible vector, as expressed in Eq. (11.15):

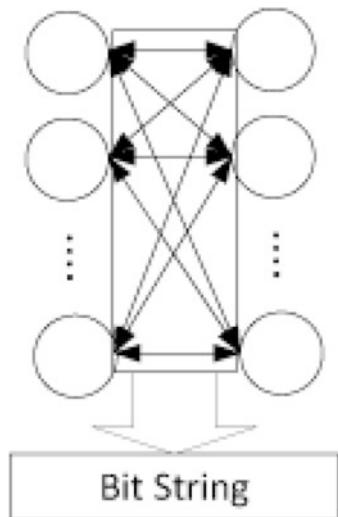
$$\text{fitness}(\mathbf{W}_i) = \frac{1}{\|\mathbf{v} - \hat{\mathbf{v}}_i\|}, \quad (11.15)$$

where  $\mathbf{W}_i$  is the weight matrix between the visible layer and the hidden layer and  $\hat{\mathbf{v}}_i$  is the computed visible vector in the current weight matrix,  $\mathbf{W}_i$ . Because it depends on the target visible vector and the current weight matrix, the hidden vector is not considered for defining the fitness evaluation measure. The weight matrix is encoded into a bit string in the next step of applying the genetic algorithm.

The process of encoding the weights between the visible layer and the hidden layer into a bit string is illustrated in Fig. 11.27. If the genetic algorithm is applied to the weight optimization, the weights are given as a matrix, and they are encoded into a bit string. Each weight value which is given as a decimal value is transformed into a binary value, and binary values which represent the weights are concatenated into a bit scheme, as a simple scheme. If the evolutionary strategy or the evolutionary programming is applied, the weight matrix is used by itself. The bit string which represents the weight matrix becomes a solution candidate in applying the genetic algorithm.

The process of optimizing the weights in the RBM by the genetic algorithm is illustrated in Fig. 11.28. The initial population is generated by randomizing bit strings. The offsprings are generated as additional candidates by applying the crossover and the mutation to existing solution candidates, their fitness values are evaluated based on the difference between the target visible vector and the computed visible vector, and the solution candidates with their higher fitness values are selected in the next population. The optimization process is to iterate the off-

**Fig. 11.27** Encoding weights into bit string



```

optimizeGA(List solutionList, int popSize, real survivalRatio, int generations){
    solutionList.initialize(popSize);
    for(int i = 0; i < generations; i++){
        additionalPopSize = popSize/survivalRatio - popSize;
        for(int i = 0; i < additionalPopSize;i++){
            randIndex1 = random(0, popSize-1);
            randIndex2 = random(0, popSize-1);
            solution1 = solutionList.getElement(randIndex1);
            solution2 = solutionList.getElement(randIndex2);
            offspring = crossover(solution1, solution2);
            solutionList.addElement(offspring);
        }
        solutionList.evalFitness();
        solutionList.rankFitness();
        solutionList.select(popSize);
    }
}

```

**Fig. 11.28** Process of optimizing weights by genetic algorithm

spring generation, the fitness evaluation, and the candidate selection. It is required to transform a bit string into a weight matrix for evaluating the fitness.

Let us make some remarks on the weight optimization in the RBM by the genetic algorithm. The difference between the target visible vector and the computed visible vector is the basis for defining the fitness evaluation metric. The weight matrix between the visible vector and the hidden vector is encoded into a bit string by applying the genetic algorithm. The weights are optimized by iterating the off-spring production, the fitness evaluation, and the candidate selection. In optimizing the weights, we consider other kinds of optimization techniques base on the swarm intelligence.

## 11.5 Summary and Further Discussions

This section is concerned with the summary and the further discussions on what is studied in this chapter. The associative memory is the process of restoring an incomplete data item into its complete one; the input and the output are defined as identical vectors in this task. The RBM is composed of the two layers: the visible layer and the hidden layer; the learning process is to optimize the weights between the two layers. The deep learning algorithm is implemented by stacking multiple RBMs. This section is intended to discuss further what is studied in this chapter.

Let us consider expanding the Hopfield networks which deals with binary values into one which deals with real values. In Sect. 11.2.3, it is assumed that the input vector is a binary vector. In the real valued Hopfield networks, each node value is given as a real value, and a real valued input vector is restored into its completed one. There are two ways of expanding the Hopfield networks into the real valued ones; each state is defined as a finite number of values and is defined as a continuous value between zero and one. We need to modify the process of updating the weights in the Hopfield networks for expanding it into the real valued version.

Let us consider the Boltzmann machine and the RBM as the two extreme cases. The Boltzmann machine is one where both full connections within hidden variables and visible variables and between them are available, whereas the RBM is one where only connection between hidden variables and visible variables is available. The partition connections within hidden variables and visible variables are allowed as a hybrid kind. We consider the full connection within either of hidden variables or visible variables as another hybrid kind. We need to determine which of visible variables and hidden variables are important for implementing the hybrid kind of Boltzmann machine.

Let us consider the parallel combination of multiple RBMs as the alternative way. In this chapter, multiple RBMs are serially combined for implementing restricted belief network. If multiple RBMs are parallelly combined, they are viewed as independent ones. The ensemble schemes such as voting and expert gate are applied to the parallel combination. It is used for dividing the input vector with its huge dimensionality into ones with their smaller dimensionality.

Let us consider introducing the fuzzy concepts for upgrading the RBM. It is designed for restoring incomplete values into complete values. If the fuzzy concept is introduced, the role of RBM is to restore fuzzy input values into crisp complete values. Its alternative role is to define a fuzzy distribution as a triangle or a trapezoid distribution from incomplete input values. If the fuzzy concepts are introduced to the neural networks, this hybrid model is called fuzzy neural networks.

# Chapter 12

## Convolutional Neural Networks



This chapter is concerned with the convolutional neural networks as another type of deep neural networks. We studied the MLP which is expanded into the convolutional neural networks in Chap. 9. The pooling layers and the convolution layers are added as the feature extraction part to the MLP. There are two parts in the architecture of the convolutional neural networks: the feature extraction which is the alternative layers of the pooling and the learning part which is the Perceptron or the MLP. This section is intended to describe the convolutional neural networks model which is expanded from the MLP with respect to the architecture and the classification process.

This chapter is composed of five sections, and we overview what is studied in this chapter in Sect. 12.1. In Sect. 12.2, we study the pooling operation as a main deep operation. In Sect. 12.3, we study the convolution as another main operation. In Sect. 12.4, we study the convolutional neural networks. In Sect. 12.5, we discuss further what is studied in this chapter.

### 12.1 Introduction

This section is concerned with the overview of CNNs (convolutional neural networks). In Chap. 9, we studied the MLP as a deep learning neural networks model. There are two parts in the CNN, the feature learning and the classification part which is covered by the MLP; the feature learning is added to MLP in the CNN. The feature learning is composed of the alternative layers of the pooling layer and the convolution layer. This section is intended to overview the CNN as the introduction to this chapter.

Because the MLP is a part of CNN, let us review the MLP which was covered in Chap. 9. There are three layers in the architecture of MLP: the input layer, the hidden layer, and the output layer; the MLP is expanded as a deep learning algorithm by adding more hidden layers. The output values are computed from the input layer by

means of hidden layers in the feedforward direction. The weights are updated from the output layer to the input layer in the backward direction. The learning process of CNN is identical to that of the MLP.

The pooling and the convolution are the main operations for implementing the CNN. The pooling is the process of extracting the representative value from each window which slides on the vector, and the convolution is the process of extracting product of input elements and filter elements from each window. Multiple pooling layers and multiple convolution layers are alternatively attached to MLP for implementing the CNN. If the input is given as a matrix, multiple matrices which are given as a tensor are generated by the convolution. In this study, it is assumed that the input is given as a vector.

In reality, there is no guarantee that the training examples and novice examples are clean. In using the machine learning algorithm, we need to consider the noise in the training examples. If there are too much noise in them, a novice input may be misclassified with the higher probability. In the CNN, more training examples are generated considering the noise through the convolution. The merit of CNN is the strong tolerance to noisy novice inputs.

Let us mention what is intended in this chapter. We understand the pooling and the convolution as the main deep operations. We design and implement the CNN by attaching both kinds of deep operations alternatively. This chapter is intended to study the two deep operations and construct the CNN by attaching the deep operations to the MLP.

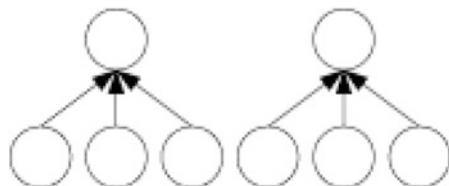
## 12.2 Pooling

This section is concerned with the pooling as an operation for implementing the convolutional neural networks. In Sect. 12.2.1, we study the basic concepts which are involved in understanding the pooling. In Sect. 12.2.2, we present the several types of pooling operation. In Sect. 12.2.3, the pooling is applied to downsizing the input. In Sect. 12.2.4, we present the applicability of the pooling to the ensemble learning.

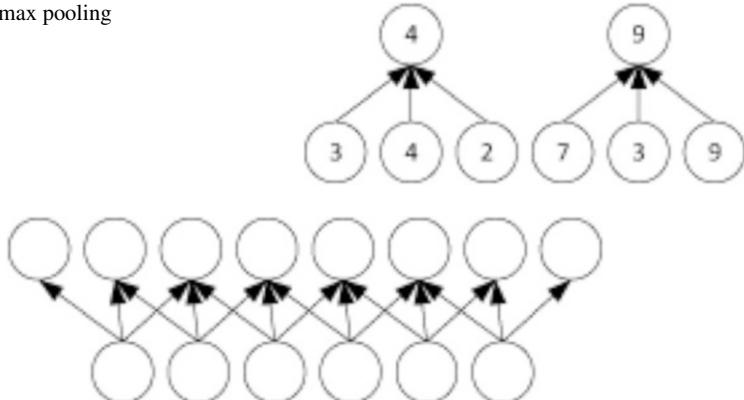
### 12.2.1 *Pooling Concepts*

This section is concerned with the concepts which are needed for understanding the pooling. It is viewed into the process of mapping a particular vector into another vector with their different dimensions. The pooling is the operation on a vector for extracting a representative value from each window which slides on it. The role of pooling is the dimension expansion as well as the dimension reduction. This section is intended to explain some concepts which are needed for understanding it.

**Fig. 12.1** Pooling for dimension reduction



**Fig. 12.2** Softmax pooling



**Fig. 12.3** Pooling for dimension expansion

The dimension reduction by the pooling operation is illustrated in Fig. 12.1. In Fig. 12.1, the dimension of the input vector is six, and the dimension of the output vector is only two. In the pooling operation, a representative value from the three-sized window. The input vector is decomposed into several subvectors, and a representative value is picked from each one. The window does not slide on the input vector, in this pooling operation.

The softmax is illustrated as a scheme of selecting a representative value from a window in the pooling in Fig. 12.2. It is the process of selecting the maximum value among several values in each window as the representative one. The six-dimensional vector is decomposed into the two subvectors, each of which has the three dimensionality. The maximum value is selected among three values in each subvector. The softmax is the popular scheme of selecting a representative value for carrying out the pooling operation.

The dimension expansion by the pooling operation is illustrated in Fig. 12.3. The pooling operation is usually used for reducing the dimension. If the dimension is expanded by the pooling operation, the nodes of both ends receive the first value and the last value of the input vector. Each inside node receives the representative value in its own window. The six-dimensional vector is expanded into the eight-dimensional one in this example.

Let us make some remarks on some concepts for understanding the pooling operation. It is used for reducing the dimension of the input vector. The softmax is to select the maximal value as the representative one among values in a particular

window, and it is adopted most frequently for implementing the pooling operation. The way of increasing the dimension by using the operation exists. Other schemes than the softmax exist, and we will study them in the next section.

### 12.2.2 Pooling Types

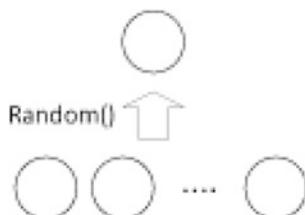
This section is concerned with the three types of pooling operation. In the previous section, we understood some concepts which are involved in the pooling operation. In this section, we mention the typical three types of pooling operation: random pooling, softmax pooling, and average pooling. The softmax pooling will be used subsequently for performing the pooling operation in the CNN. This section is intended to describe the typical three types of pooling operation.

The first type of pooling operation is illustrated in Fig. 12.4. The pooling is previously defined as the process of generating a representative value from a value set. The random pooling is to select a value at random from the set. The vector is filled with the random values which are selected from the window which slides on the input vector. Even the same input vector may be converted into a different vector for each trial in this pooling type.

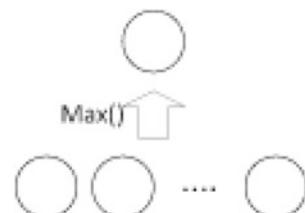
The second type of pooling operation which is called softmax pooling is illustrated in Fig. 12.5. This type of pooling operation is used most frequently in implementing the CNN. The maximal value is selected from the window which slides on an input vector or an input matrix. The output vector is filled with the maximal values which are selected from the sliding window. Because the maximal values are selected, the average on the elements of the output vector is larger.

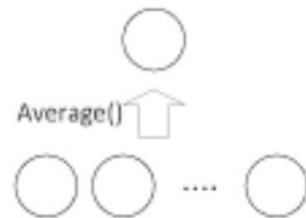
The third type of pooling operation which is called average pooling is illustrated in Fig. 12.6. If the multiple pooling layers are included in the convolutional neural

**Fig. 12.4** Random pooling



**Fig. 12.5** Softmax pooling



**Fig. 12.6** Average pooling

networks, this type is used secondly frequently. In this operation, the average over values in the sliding window is selected as the representative value. The output vector is filled with the averages, each of which is average over values in each window. The variance over values is decreased by the effect of this pooling operation.

Let us make some remarks on the three types of pooling operation. The random pooling is one where a value is selected at random from a sliding window. The softmax pooling is one where the maximal value is selected from it. The average pooling is one where average over values in a sliding window is set as its representative value. Multiple vectors or matrices are generated by using different types of pooling operations.

### 12.2.3 Dimensionality Downsizing

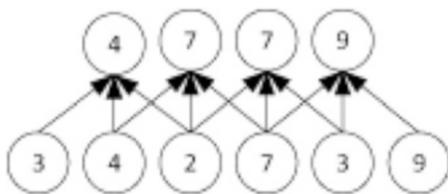
This section is concerned with the pooling operation for downsizing the dimension of the input vector. The pooling is defined as the process of selecting a representative value among values in the window. In this section, we present downsizing the input vector dimensionality as the effect of the pooling operation. The softmax is adopted as the scheme of selecting a representative value. This section is intended to describe the dimensionality reduction as the effect of the pooling operation.

The two structures of pooling operation are illustrated in Fig. 12.7. There are two ways of performing the pooling operation: exclusive way and sliding way. The  $k$  sized window slides on the input vector with  $N$  dimensions in the left part in Fig. 12.7; the  $N$  sized input is cut down into  $N - k + 1$  sided one as the effect of the pooling operation. In the right part of Fig. 12.7, the  $N$  sized input vector is divided into the exclusive  $k$  sized ones; the  $N$  sized input is cut down into  $N/k$  sized one. The former is adopted more frequently in implementing the CNN.

The pooling operation which slides on a vector is illustrated as a pseudo code in Fig. 12.8. The list of nodes which are associated with the values and the window size,  $k$ , are given as the arguments. For each element in the nodeList, the window, poolingList, is constructed as the start from the current element, the maximal value is selected in poolingList, and it is added to outputNodeList. The above process is iterated until the current value in nodeList minus  $k$ , and the ouputNodeList is returned as the final output. The size of outputNodeList is the size of nodeList minus  $k - 1$ .

**Fig. 12.7** Pooling structure**Fig. 12.8** Pooling process

```
List kSlidePooling(List nodeList, k){
    int nodeSize = nodeList.size();
    for(int i = 0; i < nodeSize - k; i++){
        List poolingList = new List();
        for(int j = 0; j < k; j++){
            Node nn = nodeList.getElement[i+j];
            poolingList.addElement(nn);
        }
        Node outputNode = poolingList.softmax();
        List outputNodeList.addElement(outputNode);
    }
    return outputNodeList;
}
```

**Fig. 12.9** Pooling example

An example of the pooling operation is illustrated in Fig. 12.9. In this example, the dimension of the input vector is six, and the window size is three. The softmax is adopted in this example; four is selected as the maximal value from the most left window which consists of 3, 4, and 2, as the first element of the output vector. The dimension of the output vector becomes five by  $4 = 6 - 3 + 1$ . The average over elements in the output vector is increased by the softmax pooling operation.

Let us make some remarks on the pooling as a scheme of reducing the dimension. There are two ways of performing the pooling operation: sliding way and exclusive way. If the sliding way is adopted, and the window size is  $k$ , and the dimension of input vector is  $d$ , the size of output vector which results from applying the pooling operation is  $d - k + 1$ . The maximal value is selected from each window in the softmax pooling. The sparsity of a numerical vector is cut down by mitigating the dominance of zero vectors as the effect of softmax pooling.



**Fig. 12.10** Ensemble learning frame

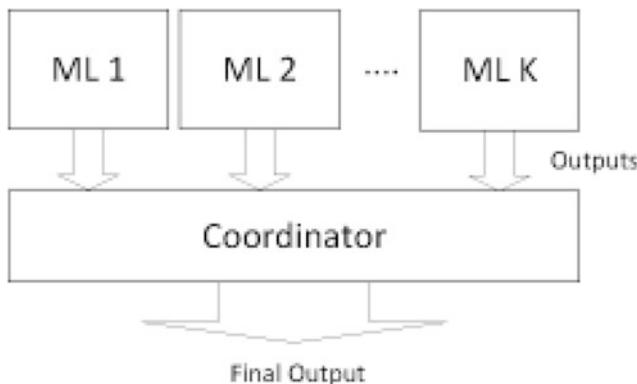
### 12.2.4 Pooling for Ensemble Learning

This section is concerned with the application of the pooling operation to the ensemble learning. In Chap. 4, we studied the ensemble learning as the learning paradigm where multiple machine learning algorithms are involved for the learning and the generalization. The pooling operation is used for voting the output values of committee members by adopting the average pooling. The voting is viewed as the process of applying the pooling operation to the  $m$  dimensional vector which consists of  $m$  output values of the committee members. This section is intended to describe the application of the pooling operation to the implementation of the ensemble learning.

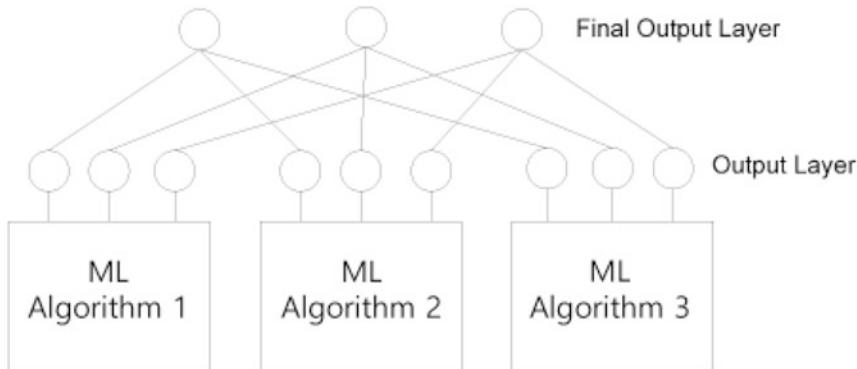
The frame of ensemble learning is illustrated in Fig. 12.10. The  $K$  machine learning algorithms are involved, and each of them is trained with the training examples. A novice input is given to the  $K$  machine learning algorithms, and the output is generated from each machine learning algorithm. The coordinator collects the answers from the  $K$  machine learning algorithms, and the final answer is decided by voting them. The pooling operation is applied to voting the answers of the  $K$  machine learning algorithms.

The voting for implementing the ensemble learning is illustrated in Fig. 12.11. It is the process of deciding the final answer by collecting the answers of the  $K$  machine learning algorithms. The output value is estimated to the novice input vector by each machine learning algorithm, and the output vectors of the  $K$  machine learning algorithms are collected by the coordinator. If the given task is a regression task, the coordinator decides the final output vector by averaging the answers. If it is a classification, the pooling operation may be applied to voting which is performed by the coordinator.

The application of the pooling operation to voting the classification results from the three classifiers is illustrated in Fig. 12.12. Each machine learning algorithm is trained for classifying an item into one of the three classes; the output vector is a three-dimensional binary vector. The voting pooling which selects a value by voting the values is adopted and applied to each element of the three vectors. If the output vector is a real vector which consists of continuous values, the average pooling is adopted for voting them. The pooling operation is applied with the spanning over multiple vectors in this case.



**Fig. 12.11** Ensemble learning scheme: voting



**Fig. 12.12** Application of pooling to voting

Let us make some remarks on the process of applying the pooling operation to the voting in the ensemble learning. It is the learning paradigm where multiple machine learning algorithms are involved in the learning and the generalization. The voting in the ensemble learning is to decide the final answer from multiple answers from the committee members. The voting is implemented by applying the pooling operation with spanning over the multiple output vectors. The pooling in the feature extraction part of the convolutional neural networks is to reduce the sparsity and the dimension of the input vector.

### 12.3 Convolution

This section is concerned with the convolution as a main operation for implementing the convolutional neural networks. In Sect. 12.3.1, we study the matrix which is

expanded from a vector as preparation of covering the convolution. In Sect. 12.3.2, we study the process of applying the convolution to the matrix. In Sect. 12.3.3, we study some variants which are derived from the convolution. In Sect. 12.3.4, we study the convolution, which is specialized for textual data, called textual convolution.

### 12.3.1 Tensor

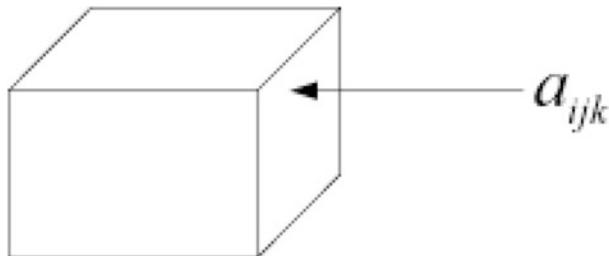
This section is concerned with the tensor which is expanded into the matrix. The vector is expanded into the matrix in studying the machine learning algorithm. The matrix is further expanded into the tensor where an element position is identified with three integer values. If the convolution is applied to a matrix with multiple filter matrices, a tensor is generated instead of a matrix. This section is intended to define a tensor, mathematically.

The tensor where each element is identified with the three indexes is illustrated in Fig. 12.13. A matrix is defined as Eq. (12.1); it is composed of  $m \times n$  elements:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}. \quad (12.1)$$

A tensor is defined as multiple matrices as Eq. (12.2):

$$\left[ \begin{pmatrix} a_{111} & a_{112} & \dots & a_{11n} \\ a_{121} & a_{122} & \dots & a_{12n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1m1} & a_{1m2} & \dots & a_{1mn} \end{pmatrix} \begin{pmatrix} a_{211} & a_{212} & \dots & a_{21n} \\ a_{221} & a_{222} & \dots & a_{22n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{2m1} & a_{2m2} & \dots & a_{2mn} \end{pmatrix} \dots \begin{pmatrix} a_{l11} & a_{l12} & \dots & a_{l1n} \\ a_{l21} & a_{l22} & \dots & a_{l2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{lm1} & a_{lm2} & \dots & a_{lmn} \end{pmatrix} \right]. \quad (12.2)$$



**Fig. 12.13** Three-dimensional tensor and its element

The tensor is viewed as a parallelepiped; a tensor is expressed as six matrices as shown in Eq.(12.3):

$$\begin{aligned} & \begin{pmatrix} a_{111} & a_{112} & \dots & a_{11n} \\ a_{121} & a_{122} & \dots & a_{12n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1m1} & a_{1m2} & \dots & a_{1mn} \end{pmatrix} \begin{pmatrix} a_{l11} & a_{l12} & \dots & a_{l1n} \\ a_{l21} & a_{l22} & \dots & a_{l2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{lm1} & a_{lm2} & \dots & a_{lmn} \end{pmatrix} \\ & \begin{pmatrix} a_{111} & a_{112} & \dots & a_{11n} \\ a_{211} & a_{212} & \dots & a_{21n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{l11} & a_{l12} & \dots & a_{l1n} \end{pmatrix} \begin{pmatrix} a_{1m1} & a_{1m2} & \dots & a_{1mn} \\ a_{2m1} & a_{2m2} & \dots & a_{2mn} \\ \vdots & \vdots & \ddots & \vdots \\ a_{lm1} & a_{lm2} & \dots & a_{lmn} \end{pmatrix}. \quad (12.3) \\ & \begin{pmatrix} a_{111} & a_{111} & \dots & a_{1m1} \\ a_{211} & a_{211} & \dots & a_{2m1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{l11} & a_{l11} & \dots & a_{lm1} \end{pmatrix} \begin{pmatrix} a_{11n} & a_{11n} & \dots & a_{1mn} \\ a_{21n} & a_{21n} & \dots & a_{2mn} \\ \vdots & \vdots & \ddots & \vdots \\ a_{l1n} & a_{l1n} & \dots & a_{lmn} \end{pmatrix} \end{aligned}$$

The tensor which is shown above is expressed as  $l \times m \times n$ .

Let us mention the basic operations on the tensors. The tensors **A** and **B** which are expressed in Eq.(12.3) are abbreviated into  $\mathbf{A} = [a]_{ijk}$  and  $\mathbf{B} = [b]_{ijk}$ . The addition and the minus of the two tensors, **A** and **B**, are defined into Eq.(12.4),

$$\mathbf{A} \pm \mathbf{B} = [a \pm b]_{ijk} \quad (12.4)$$

The multiplication of the tensor, **A**, by a scalar value,  $k$ , is defined into Eq.(12.5):

$$k\mathbf{A} = [ka]_{ijk} \quad (12.5)$$

The linear combination of tensors may be considered based on the operations which are defined above.

Let us consider the multiplication of the  $l \times m \times n$  with the  $l$  dimensional vector, the  $m$  dimensional vector, and the  $n$  dimensional vector. If the  $l \times m \times n$  tensor is multiplied with the  $n$  dimensional vector,  $\mathbf{b}_1 = [b_{11} \ b_{12} \ \dots \ b_{1n}]$ , the multiplication is expressed as Eq.(12.6):

$$\mathbf{Ab}_1 = \begin{pmatrix} \sum_{i=1}^n a_{11i} b_{1i} & \sum_{i=1}^n a_{12i} b_{1i} & \dots & \sum_{i=1}^n a_{1mi} b_{1i} \\ \sum_{i=1}^n a_{21i} b_{1i} & \sum_{i=1}^n a_{22i} b_{1i} & \dots & \sum_{i=1}^n a_{2mi} b_{1i} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^n a_{l1i} b_{1i} & \sum_{i=1}^n a_{l2i} b_{1i} & \dots & \sum_{i=1}^n a_{lmi} b_{1i} \end{pmatrix}. \quad (12.6)$$

If the  $l \times m \times n$  tensor is multiplied with the  $m$  dimensional vector,  $\mathbf{b}_2 = [b_{21} \ b_{22} \ \dots \ b_{2m}]$ , the  $l \times n$  matrix is generated as shown as Eq. (12.7):

$$\mathbf{Ab}_2 = \begin{pmatrix} \sum_{i=1}^m a_{1i1}b_{2i} & \sum_{i=1}^m a_{1i2}b_{2i} & \dots & \sum_{i=1}^m a_{1in}b_{2i} \\ \sum_{i=1}^m a_{2i1}b_{2i} & \sum_{i=1}^m a_{2i2}b_{2i} & \dots & \sum_{i=1}^m a_{2in}b_{2i} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^m a_{li1}b_{2i} & \sum_{i=1}^m a_{li2}b_{2i} & \dots & \sum_{i=1}^m a_{lin}b_{2i} \end{pmatrix}. \quad (12.7)$$

If the  $l \times m \times n$  tensor is multiplied with the  $l$  dimensional vector,  $\mathbf{b}_3 = [b_{31} \ b_{32} \ \dots \ b_{3l}]$ , the  $m \times n$  matrix is generated as shown as Eq. (12.8),

$$\mathbf{Ab}_3 = \begin{pmatrix} \sum_{i=1}^l a_{i11}b_{3i} & \sum_{i=1}^l a_{i12}b_{3i} & \dots & \sum_{i=1}^l a_{i1n}b_{3i} \\ \sum_{i=1}^l a_{i21}b_{3i} & \sum_{i=1}^l a_{i22}b_{3i} & \dots & \sum_{i=1}^l a_{i2n}b_{3i} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^l a_{im1}b_{3i} & \sum_{i=1}^l a_{im2}b_{3i} & \dots & \sum_{i=1}^l a_{imn}b_{3i} \end{pmatrix}. \quad (12.8)$$

If a tensor is multiplied with a matrix, a tensor is generated.

Let us make some remarks on the tensor which is expanded from a matrix. A tensor is represented into a set of matrices. The basic operations which are addition, deletion, and multiplication with a scalar value are applicable to the tensor as well as the matrix. A matrix is generated by multiplying a tensor with a vector. The multiplication of a tensor with another tensor will be considered in the next study.

### 12.3.2 Single-Channeled Convolution

This section is concerned with the convolution with a single filter. The convolution is intended to filter some pixels from an image; the matrix which represents directly an image is given as the input. The convolution is to map a matrix into another matrix by the filter matrix. If multiple filter matrices are used, tensor which is covered in Sect. 12.3.1 is generated by the operation on a matrix. This section is intended to describe the convolution on a matrix with a single filter matrix.

Let us mention the input matrix which is an operand of the deep operation, convolution. The CNN (convolutional neural network) is applied to the image classification, so the input data is assumed as a matrix [1]. The input matrix is assumed as a  $d \times d$  square matrix as shown in Eq. (12.9):

$$\mathbf{X} = \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1d} \\ x_{21} & x_{22} & \dots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ x_{d1} & x_{d2} & \dots & x_{dd} \end{pmatrix}. \quad (12.9)$$

The convolution is applied to the input matrix which is expressed in Eq. (12.9).

It is required to define the filter matrix in advance for applying the convolution. It is assumed that a single filter matrix is defined for carrying out the convolution. The filter matrix is given as Eq. (12.12):

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1k} \\ c_{21} & c_{22} & \dots & c_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ c_{k1} & c_{k2} & \dots & c_{kk} \end{pmatrix}. \quad (12.10)$$

The filter matrix is a smaller square matrix than the input matrix with  $k < d$ . In the next section, we will study the convolution with multiple filter matrices.

Let us mention the output matrix which is mapped from the input matrix by the convolution with a filter matrix. If the convolution is applied to the matrix  $\mathbf{X}$ , with the filter matrix,  $\mathbf{C}$ , the size of output matrix,  $\mathbf{Y}$  is  $(d - k + 1) \times (d - k + 1)$ . In the matrix,  $\mathbf{Y}$ , the element,  $y_{km}$ , is computed by Eq. (12.11):

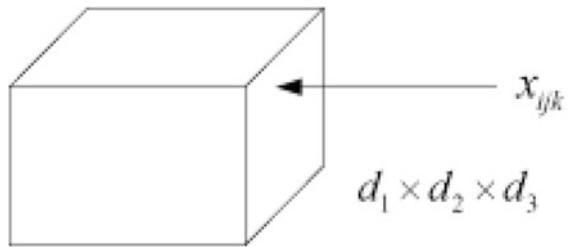
$$\mathbf{C} = y_{km} = \sum_{i=1}^h \sum_{j=1}^h c_{ij} x_{(i+k-1)(j+k-1)}. \quad (12.11)$$

The matrix,  $\mathbf{Y}$ , which is mapped by the convolution is expressed as Eq. (12.12):

$$\mathbf{Y} = \begin{pmatrix} \sum_{i=1}^h \sum_{j=1}^h c_{ij} x_{ij} & \sum_{i=1}^h \sum_{j=1}^h c_{ij} x_{i(j+1)} & \dots & \sum_{i=1}^h \sum_{j=1}^h c_{ij} x_{i(j+d-h)} \\ \sum_{i=1}^h \sum_{j=1}^h c_{ij} x_{(i+1)j} & \sum_{i=1}^h \sum_{j=1}^h c_{ij} x_{(i+1)(j+1)} & \dots & \sum_{i=1}^h \sum_{j=1}^h c_{ij} x_{(i+1)(j+d-h)} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^h \sum_{j=1}^h c_{ij} x_{(i+d-1)j} & \sum_{i=1}^h \sum_{j=1}^h c_{ij} x_{(i+d-1)(j+1)} & \dots & \sum_{i=1}^h \sum_{j=1}^h c_{ij} x_{(i+d-1)(j+d-h)} \end{pmatrix}. \quad (12.12)$$

We may consider the padding which fills the boundary of the input matrix with zero values for maintaining the matrix size.

Let us make some remarks on the convolution with a single filter matrix. A square matrix is assumed as the input, instead of a vector. In advance, a filter matrix is defined as the preparation for carrying out the convolution. The output matrix is computed by sliding the product of the filter matrix on the input matrix. This operation will be expanded into one with multiple filter matrices.

**Fig. 12.14** Input tensor

### 12.3.3 Tensor Convolution

This section is concerned with the convolution which is applied to a tensor. If the convolution is applied with a single filter matrix, the output is generated as a matrix from an input matrix. If the convolution is applied to it with a filter tensor, the output is generated as a tensor even from the input which is given as a matrix. If their sizes are identical, multiple filter matrices are viewed as a tensor; it is assumed that the input, the filter, and the output are tensors. This section is intended to describe the convolution on a tensor with multiple filter matrices which are given as a tensor.

The input tensor is illustrated in Fig. 12.14. If a data item is represented into multiple matrices or a single tensor, the input data is given as a tensor instead of a matrix. The input data is notated by a tensor as shown in Eq. (12.13):

$$\mathbf{C} = \mathbf{X} = [x]_{ijk}, 1 \leq i \leq d_1, 1 \leq j \leq d_2, 1 \leq k \leq d_3 \quad (12.13)$$

If multiple convolution layers are installed in the CNN, the output from the previous convolution layers which is given as a tensor becomes the input to the current convolution layer. The convolution with a filter tensor is applied to this input tensor.

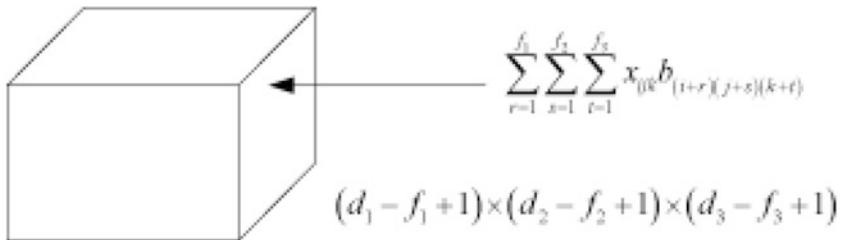
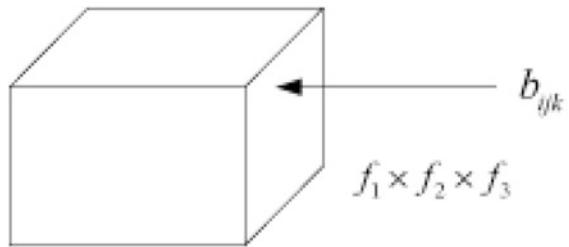
The filter tensor which consists of multiple filter matrices is illustrated in Fig. 12.15. The input data is assumed as a tensor for generalizing the convolution operation. The filter tensor is notated as shown in Eq. (12.14):

$$\mathbf{F} = \mathbf{X} = [b]_{ijk}, 1 \leq i \leq f_1, 1 \leq j \leq f_2, 1 \leq k \leq f_3 \quad (12.14)$$

The filter tensor with the size,  $f_1 \times f_2 \times f_3$ , is interpreted into  $f_3$  matrices with the size,  $f_1 \times f_2$ . The convolution with a single filter tensor is applied to a single input tensor.

The output tensor which is the result from applying the convolution to the input tensor is illustrated in Fig. 12.16. As mentioned above, the input tensor is prepared, and the filter tensor is defined for carrying out the convolution. The output tensor is expressed as Eq. (12.15):

$$\mathbf{Y} = \left[ \sum_{r=1}^{f_1} \sum_{s=1}^{f_2} \sum_{t=1}^{f_3} x_{ijk} b_{(i+r)(j+s)(k+t)} \right]_{ijk} \quad (12.15)$$

**Fig. 12.15** Filter tensor**Fig. 12.16** Output tensor

The size of the output tensor is expressed as Eq. (12.16):

$$(d_1 - f_1 + 1) \times (d_2 - f_2 + 1) \times (d_3 - f_3 + 1) \quad (12.16)$$

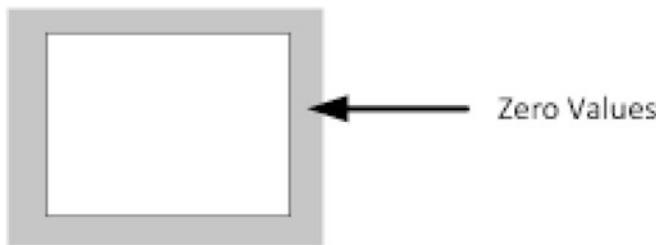
The size of tensor is reduced as shown in Eq. (12.16).

Let us make some remarks on the convolution which is applied to the tensor. The input data is assumed as a tensor. The filter is defined as a tensor for applying the convolution. A tensor with its smaller size than the input tensor is generated as the result. The convolution is applied to the input tensor with multiple tensors.

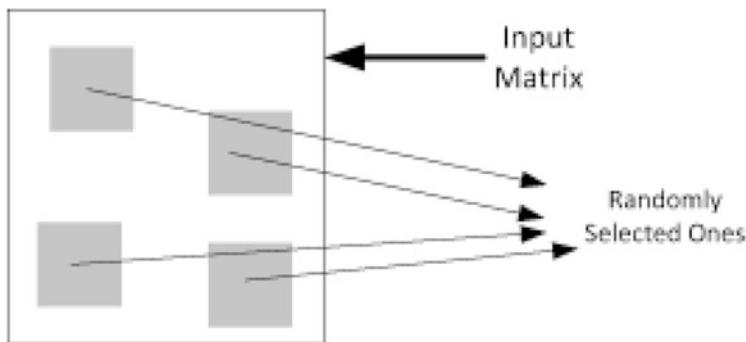
### 12.3.4 Convolution Variants

This section is concerned with some variants which are derived from the convolution. The standard convolution was studied in Sects. 12.3.2 and 12.3.3. In this section, we consider some variants in addition to the standard one. Some variants are derived by putting the padding, shifting multiple steps in applying the filter matrix, or selecting elements from the input matrix randomly. This section is intended to describe the three typical variants of the convolution.

The padding which fills zeros in the matrix boundary is illustrated in Fig. 12.17. The input data is given as a matrix. If the boundary is filled with zero values, the  $d \times d$  matrix is increased into the  $(d + 2) \times (d + 2)$  matrix. If the convolution is applied with the  $f \times f$  filter matrix by the padding, the size of output matrix is



**Fig. 12.17** Convolution with padding



**Fig. 12.18** Selective convolution

$(d - f + 3) \times (d - f + 3)$ . If the convolution is applied with multiple paddings, the matrix size is maintained or increased.

The stride is considered in addition for carrying out the convolution. The stride is set as one in the traditional convolution which is explained in the previous sections. The stride is the number of shifts of the filter vector or matrix in the input matrix for performing the convolution. The stride is denoted by  $s$ , and the size of output matrix is expressed as Eq. (12.17):

$$\frac{d - f}{s} + 1 \quad (12.17)$$

If the stride is set more than one, the size of output vector or matrix is reduced more.

The selective convolution as one more variant is illustrated in Fig. 12.18. The filter vector or matrix slides on the input vector or matrix in the conventional convolution. In this variant, the filter vector or matrix is put at random on the input vector or matrix, instead of sliding. If the convolution is applied with the  $f$  dimensional filter vector to the  $d$  dimensional vector, a random integer value is generated between 1 and  $d - f + 1$  for each trial. It is possible to visit the same area of the input matrix in this variant.

Let us make some remarks on some variants of the convolution operation. The padding is the process of filling the boundary of the input vector or matrix with zero

values. The stride is the number of shifting the filter vector or matrix in the input vector or matrix. The filter vector or matrix is put at random instead of sliding it in the input vector or matrix in performing the convolution. We consider using the filter vector or matrix at random among multiple ones.

## 12.4 CNN Design

This section is concerned with the various schemes of designing the convolutional neural networks. In Sect. 12.4.1, we study the ReLU (Rectified Linear Unit) as an activation function. In Sect. 12.4.2, we present a design of convolutional neural networks with the pooling and the ReLU. In Sect. 12.4.3, we present a design of it with the convolution and the ReLU. In Sect. 12.4.4, we present a design with the convolution, the pooling, and the ReLU.

### 12.4.1 ReLU

This section is concerned with ReLU (Rectified Linear Unit). We studied the pooling and the convolution in Sects. 12.2 and 12.3. The CNN may be designed as various versions for classifying data items. The ReLU is adopted as the activation function in the part of classifying the mapped vector or matrix, called classification part. This section is intended to describe the classification part which uses the ReLU as the activation function in the CNN.

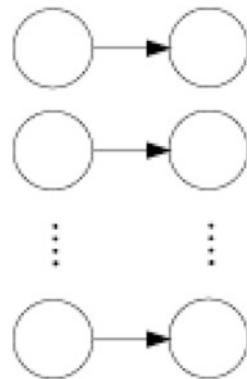
The one-to-one connection between the layers is illustrated in Fig. 12.19. It is only one connection from one node to its corresponding node in the next layer. The input node value and the output node value are denoted by  $x_i$  and  $\hat{x}_i$ , respectively. The weight is denoted by  $w_i$ , and the node value in the right layer is computed by Eq. (12.18):

$$\hat{x}_j = \begin{cases} w_i x_i & \text{if } net \geq 0 \\ 0 & \text{otherwise} \end{cases}. \quad (12.18)$$

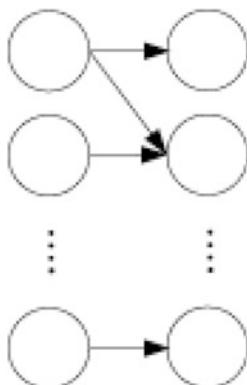
If the net input is more than or equal to zero, the ReLU is the same to the linear function.

The partial connection between layers is illustrated in Fig. 12.20. It is defined as the connection of each node to not all nodes in the next layer. The set of nodes in the previous layer which are connected to the node,  $\hat{x}_i$ , by  $N_i$ . The node value is computed by Eq. (12.19),

**Fig. 12.19** One-to-one connection in ReLU



**Fig. 12.20** Partial connection in ReLU



$$\hat{y}_j = \begin{cases} \sum_{r \in N_i} w_{ir} x_r & \text{if } \sum_{r \in N_i} w_{ir} x_r \geq 0 \\ 0 & \text{otherwise} \end{cases}. \quad (12.19)$$

The partial connection is caused by eliminating some weights in the full connection between layers.

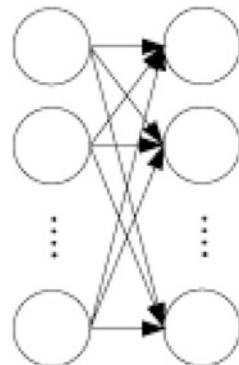
The full connection between the layers is illustrated in Fig. 12.21. The full connection is the case of connecting each node to all nodes in the next layer. If the number of nodes in the left layer is  $d$ , the node value is computed by Eq. (12.20):

$$\hat{y}_j = \begin{cases} \sum_{r=1}^d w_{ir} x_r & \text{if } \sum_{r=1}^d w_{ir} x_r \geq 0 \\ 0 & \text{otherwise} \end{cases}. \quad (12.20)$$

This full connection between the layers is adopted in the Perceptron and the MLP. It is assumed that the connection of nodes between layers is complete in the CNN.

Let us make some remarks on the activation function which is called Rectified Linear Unit. The output value is computed by Eq. (12.18) in the one-to-one

**Fig. 12.21** Complete connection in ReLU



connection. The partial connection is the connection of each node in the current layer to not all nodes in the next layer. The full connection which is the connection of each node to all nodes is adopted in the previous neural networks, such as MLP, RNN, and RBM. The ReLU is assumed as a linear function for deriving the weight update rule.

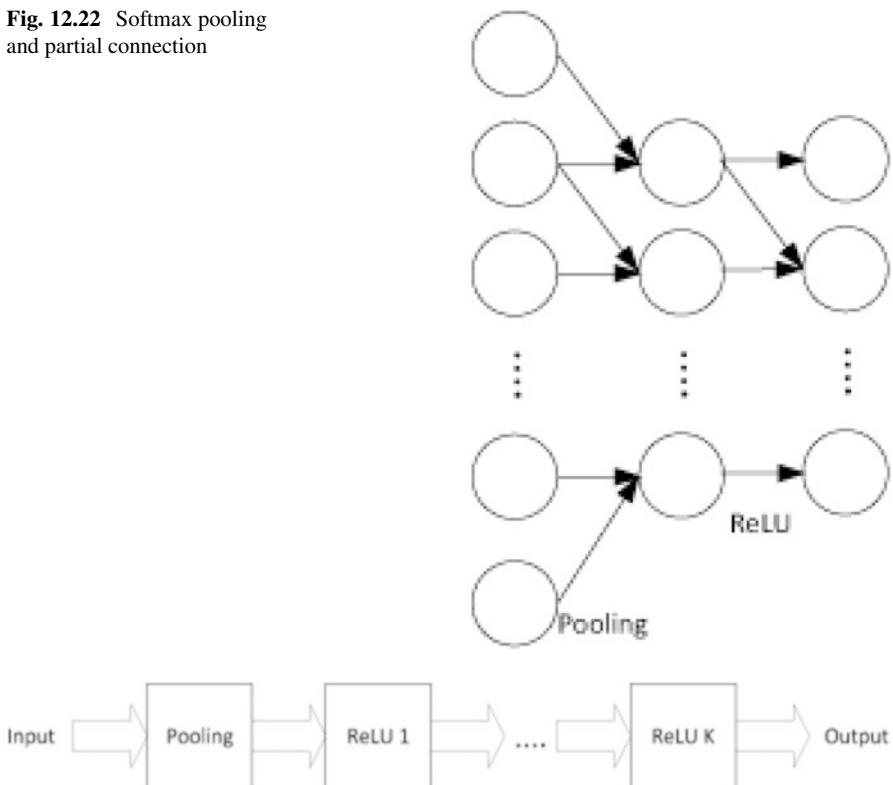
#### 12.4.2 Pooling + ReLU

This section is concerned with the simplest version CNN which consists of the pooling layer and the ReLU. In the previous section, the ReLU is an activation function with the linear function within the positive area. In this version of CNN, the role of pooling layer is the feature extraction, and the role of Perceptron with ReLU is the classification. In the Perceptron, it is assumed that the connection between the input layer and the output layer is partial. This section is intended to describe the design of CNN with the two components.

The simplest version of CNN which consists of the pooling layer and the ReLU with the partial connection is illustrated in Fig. 12.22. In the architecture, the pooling layer is the extraction part, and the ReLU is the classification part. The maximal value is selected from the window which slides on the input vector in the pooling layer. The vector which is mapped by the pooling layer is classified by the ReLU. The partial connection is viewed as the full connection where some connections are associated with zero values.

The second version of CNN which consists of a single pooling layer and multiple ReLU layers is illustrated in Fig. 12.23. The dimensionality is reduced by selecting representative values from each window in the pooling layer. The output values are computed by ones which are mapped by the pooling layer in the first ReLU. The output values in the current ReLU are transferred as the input values to the next ReLU, and the final output values are computed in the last ReLU. This version is

**Fig. 12.22** Softmax pooling and partial connection



**Fig. 12.23** Pooling and multiple ReLUs



**Fig. 12.24** Multiple poolings and multiple ReLUs

intended to solve nonlinear problems between the input values and the final output values.

The third version which consists of multiple pooling layers and multiple ReLUs is illustrated in Fig. 12.24. The difference of the version is the expansion of a single pooling layer into multiple pooling layers. The input vector is mapped into a vector with its reduced dimensionality through a series of pooling layers in the serial connection of pooling layers and ReLUs. The final output vector is computed by a series of ReLUs. We consider the parallel connection of multiple pooling layers and multiple ReLUs, later.

Let us make some remarks on the CNN with the pooling layer and the ReLU. The softmax pooling is adopted for implementing the pooling layer, and the partial connection is assumed in the ReLU. A single pooling layer is put, and multiple ReLUs are considered for solving nonlinear problems. It is possible to make the serial design and the parallel design of multiple pooling layers and multiple ReLUs. More versions of CNN with the pooling layer and the ReLU are derived by adopting another kind of pooling operation and another type of ReLU.

### ***12.4.3 Convolution + ReLU***

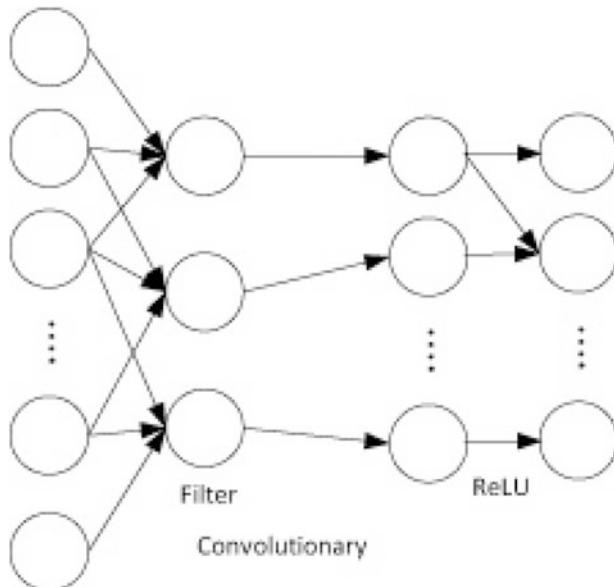
This section is concerned with another kind of CNN which consists of the convolution layer and the ReLU. In the previous section, we studied the version which consists of the pooling layer and the ReLU. The pooling layer is replaced by the convolutional layer. The CNN with a single convolutional layer and a single ReLU is expanded into one with multiple convolutional layers and multiple ReLUs. This section is intended to describe the CNN with the convolutional layer and the ReLU.

The CNN version which consists of the convolutional layer and the ReLU is illustrated in Fig. 12.25. The input vector is mapped into another vector by the convolutional layer. In the ReLU, the output vector is computed by the vector which is mapped through the convolutional layer. The weights in the ReLU are optimized by the mapped vectors, instead of the training examples. We consider applying the convolution to each training example with multiple filter vectors.

The version of CNN which consists of a single convolution layer and multiple ReLUs is illustrated in Fig. 12.26. As mentioned in Sect. 12.4.2, multiple ReLUs are intended to solve nonlinear classifications or regressions. The learning process is to map the training examples into other vectors by the convolutional layer and the update weights from the last ReLU to the first ReLU. The generalization process is to map a novice input vector into another vector by the convolutional layer and compute the output vector of the last ReLU. The role of the first ReLU and the intermediate ReLU is to compute the hidden vector.

The version of CNN with the multiple convolutional layers and multiple ReLUs is illustrated in Fig. 12.27. The convolutional layers are connected serially, and the ReLUs are also connected so, in the architecture. The input vector is mapped with multiple steps into a vector or vectors through the serial convolutional layers. The finally mapped vector or vectors are classified into one among the predefined categories by the multiple ReLUs. We may consider the parallel connection of convolutional layers and ReLUs as the alternative version.

Let us make some remarks on the CNN versions, each of which consists of the convolutional layer and the ReLU. The CNN with a single convolutional layer and a single ReLU is the simplest version. It is expanded into the version with multiple ReLUs for solving nonlinear problems. Multiple convolutional layers are connected serially for reducing the dimensionality and generating more training examples. If



**Fig. 12.25** Convolutionary and partial connection



**Fig. 12.26** Convolutionary layer and multiple ReLUs

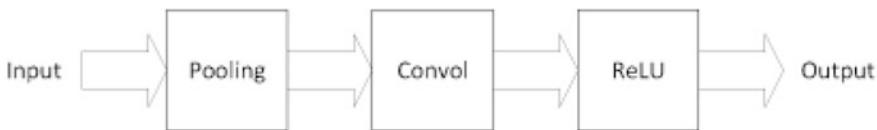


**Fig. 12.27** Multiple convolutionary layers + multiple ReLUs

the convolution is applied with multiple filter vectors, the number of mapped vectors is multiple to the number of training examples.

#### 12.4.4 Pooling + Convolution + ReLU

This section is concerned with the CNN which consists of the pooling layer, the convolutional layer, and the ReLU. In the previous section, in designing the CNN, we use either of the pooling layer or the convolutional layer. In this section, we use



**Fig. 12.28** Pooling + convolution + ReLU



**Fig. 12.29** Convolution + pooling + ReLU

both for designing it. There are both ways in CNN which consists of both layers and ReLU: the pooling after the convolution and the pooling before it. This section is intended to describe the design of CNN with both deep operations.

The simplest version of CNN which consists of the three components is illustrated in Fig. 12.28. The input vector is mapped into one with its reduced dimensionality by the pooling layer. If multiple filter vectors are defined, multiple vectors are mapped from a single vector by the convolutional layer. The final output vector is computed by the ReLU from the finally mapped vector. We may consider the parallel multiple ReLUs for processing multiple vectors which are generated from the convolutional layer.

Another version of CNN where the pooling layer and the convolutional layer are swapped with each other is illustrated in Fig. 12.29. In the previous version, the input vector is mapped into one with its lower dimensionality by the pooling layer, and multiple vectors are generated by the convolutional layer. In this version, multiple vectors are generated from the input vector by the convolutional layer, and the dimensionality is reduced by the pooling layer. We expect different results from the two versions; it is impossible to decide which of the two versions is better than the other. This version is expanded into one by putting multiple convolutional layers and multiple pooling layers.

The advanced version of CNN with multiple pooling layers and multiple convolution layers is illustrated in Fig. 12.30. The  $M$  convolutional layers and the  $N$  pooling layers are put in the extraction part; the  $M + N$  layers are totally put in the extraction part. The input vector is transformed into multiple vectors through the  $M$  convolutional layers, and they are transformed into ones with their reduced dimensionality by the  $N$  pooling layers. The vectors which are mapped finally by the  $M + N$  layers are classified by the ReLU. We may consider expanding a single ReLU into multiple ReLUs for solving the nonlinear classification.

Let us make some remarks on the CNN which consists of the pooling layer, the convolutional layer, and ReLU. In this version, the dimension is reduced by the pooling layer, multiple vectors are generated by the convolutional layer, and they are



**Fig. 12.30** Multiple convolutions + multiple poolings + ReLU

classified by ReLU. Another version is derived by swapping the pooling layer and the convolutional layer with each other. In the feature extraction part, consider both multiple pooling layers and multiple convolutional layers. The RNN with multiple pooling layers, multiple convolutional layers, and multiple ReLUs is implemented in the distributed and parallel computing environment.

## 12.5 Summary and Further Discussions

This section is concerned with the summary and further discussions on what is studied in this chapter. The pooling is the operation on a vector and a matrix for extracting representative value from each window which slides on it. The convolution is the operation for filtering values with defined filter vectors or matrices. The CNN is designed by attaching alternatively the two operation layers before the MLP or the Perceptron. This section is intended to discuss further what is studied in this chapter.

Let us consider the pooling which is specialized to textual data. In studying the pooling, it is assumed the input is a vector or a matrix. It is required to encode a text into a numerical vector for applying the CNN to text mining tasks. It is possible to modify the pooling operation into one which is applicable directly to textual data, called textual pooling. We study the textual pooling in detail in Chap. 15.

Let us consider the convolution which is specialized to textual data. It is required to define a filter vector or matrix for applying the operation to an input vector or matrix. This operation is modified into one which is applicable directly to textual data, called textual convolution. The query-based text summarization is the typical instance of textual convolution in that the query is given as the filter. The textual convolution will be studied in detail in Chap. 15.

Let us consider combining multiple CNNs. There are almost infinite number of CNN architectures, depending on how to arrange the pooling layers and the convolution layers, and how many layers we use. Multiple architectures of CNN are adopted for classifying a data item, and the ensemble learning is applied to them. Multiple CNNs with their different architectures are implemented as a parallel algorithm for processing data items more efficiently. The CNNs are discriminated with their different initial weights and their different activation functions in their classification part.

Let us consider the hierarchical combination of multiple CNNs. By the convolution with multiple filter vectors, more than one vector or matrix are generated from

a single vector or matrix. If multiple vectors or matrices are generated from the convolution layer, multiple pooling layers should be prepared as many as generated matrices and vectors. A convolution layer is put after a pooling layer, and multiple pooling layers are put for each convolution layer; the convolution layer may be expanded as a hierarchical structure. In designing the CNN, the convolution layer is constructed like a tree by using multiple filter vectors or matrices.

## Reference

1. W. Rawat and Z. Wang, “Deep convolutional neural networks for image classification: A comprehensive review.”, 2352–2449, Neural computation, 29, 2017.

## **Part IV**

# **Textual Deep Learning**

This part is concerned with the specialization of deep learning algorithms for text mining tasks. In the previous parts, we studied the modification of the machine learning algorithms into their deep versions and the deep neural networks which are applied to generic tasks. In this part, we study the process of mapping the textual data into another textual data as hidden values and the application of the process to the modification into textual deep learning algorithms. We mention the text summarization which is intended to map an initial text into its essential one as well as to summarize it. This part is intended to describe the specialization and the application of the deep learning to text mining tasks.

This part consists of the four chapters. In Chap. 13, we will study the index expansion which is necessary for implementing the textual deep learning. In Chap. 14, we will study the text summarization which is used for implementing the textual deep learning. In Chap. 15, we will do the textual operators which are based on the index expansion and the text summarization. In Chap. 16, we will study the design and the implementation of the text classification system, applying the textual deep learning algorithms.

# Chapter 13

## Index Expansion



This chapter is concerned with the index expansion which is necessary for implementing the textual deep learning. The index expansion refers to the process of adding more words which are relevant to ones in an input text. In the index expansion process, an input text is indexed into a list of words, their associated words are retrieved from external sources, and they are added to the list of words. There are three groups of words in indexing a text: the expansion group which contains very important words demanding their associated ones, the inclusion group which contains medium ones which are included as index, and the removal group which contains trivial ones which should be excluded from the index. This chapter is intended to describe the text indexing process and the index expansion.

This chapter is composed with the five sections, and in Sect. 13.1, we overview what is studied in this chapter. In Sect. 13.2, we study the process of indexing a text into a list of words. In Sect. 13.3, we study the schemes of computing the semantic similarity between words. In Sect. 13.4, we study the schemes of expanding the index. In Sect. 13.5, we discuss further what is studied in this chapter.

### 13.1 Introduction

This section is concerned with the overview of index expansion. It is defined as the process of adding more essential words to ones which are included in the given texts from external sources. The process of the index expansion is to index a text into a list of words, select important words among them, and add their associated words from external sources to the selected ones. It is necessary to remove unimportant words from the list for processing texts more efficiently. This section is intended to overview the index expansion as the introduction to this chapter.

The text indexing is defined as the process of converting a text into a list of words. A text which is given as an input is composed with sentences which are written in a natural language. A text is converted so with the basic three steps: tokenization,

stemming, and stop word removal. Depending on application area, we need further steps such as removal of unimportant words and addition of associate words. In addition, we consider the text length in indexing a text.

We need to consider the fact that text length is variable. The number of words which are results from indexing a text depends on the text length. It influences on the process of encoding a text into a numerical vector; a short text tends to be encoded into a sparse vector. Because a long text is indexed into many words, it has more chance to be retrieved much frequently in the information retrieval system. We need to segment a long text into subtexts, based on its contents.

Let us mention the index optimization which includes both index expansion and index filtering. If a short text is indexed, more relevant words should be added from external sources, and if a long text is indexed, words which are less relevant to its contents should be removed. For optimizing the index, more relevant words to important words should be added as their associative ones, and less relevant words should be removed. It is proposed that words in a text should be classified into one of the three categories: expansion, inclusion, and removal [4]. It is required to optimize the index for improving the information retrieval performance.

Let us mention what is intended in this chapter. We understand the process of indexing a text into a list of words as the background. We understand the semantic similarity between words and texts which is necessary for the index expansion. We understand some schemes of expanding index as the core part of this chapter. This chapter is intended to study the index expansion which is necessary for defining the textual deep operations.

## 13.2 Text Indexing

This section is concerned with the process of indexing a text into a list of words. In Sect. 13.2.1, we describe the tokenization as the first step of text indexing. In Sect. 13.2.2, we describe the stemming as the second step. In Sect. 13.2.3, we describe the stop word removal as the third step. In Sect. 13.2.4, we describe additional filtering as the additional step.

### 13.2.1 Tokenization

This section is concerned with the first step of indexing a text, called tokenization. It is assumed that the input text where words are delimited by the white space is written in English. The tokenization is the process of segmenting a text into words before stemming. As the additional tasks for the tokenization, special characters are removed, and capital characters are transformed into lower characters. This section is intended to describe the tokenization as the first step of text indexing.

**Fig. 13.1** Character processing

```
String treatCharacter(String fullText){
    String copiedText = new String();
    for(int i = 0; i < fullText.length(); i++){
        char ch = fullText.getChar(i);
        if(ch.isUpper())
            ch = ch.toLowerCase();
        if(!ch.isNumber() || !ch.isSpecial())
            copiedText.setChar(i, ch);
    }
    return copiedText;
}
```

**Fig. 13.2** String tokenization

```
String[] tokenizeString(String fullText){
    String tokenList[] = fullText.split("\r\n");
    return tokenList;
}
```

The character processing for the tokenization in the text is illustrated in Fig. 13.1. The initial text is given as a string which consists of characters. For each character, if it is an upper one, it is changed into its lower character, and if it is a number or a special character, it is removed. A text consists of lower characters with removal of numbers and special characters as the output. The preprocessed text is used as the input for the main tokenization step.

The process of segmenting a text into words is illustrated as a pseudo-code in Fig. 13.2. The text is preprocessed by changing upper cases into lower cases and removing numbers and special characters. The text which is preprocessed in the previous step is segmented into tokens by the white space, the carriage return, or punctuation mark. The output of this process is the list of tokens into which the text is segmented. We need to change each token into its root form in the next step.

The example of the text tokenization is presented in Fig. 13.3. The left part in the Fig. 13.3 is the input text, and the right part is the results from tokenizing the text. The upper characters are transformed into the lower characters in the preprocessing; the character, “T” is transformed into “t,” in this example. The text is partitioned into tokens by the white space; the list of tokens is presented in the right part of Fig. 13.3. The results from the tokenization are transferred into the next step, stemming, as the input.

Let us make some remarks on the tokenization as the first step of indexing a text into a list of words. The preprocess of the input text is to change upper cases into lower cases and remove numbers and special characters. The preprocessed text is partitioned into tokens by the white space as the main step of the tokenization. The text which is given as a string is converted into a list of tokens. The tokens are processed individually in subsequent steps; a list of strings is converted into another list.

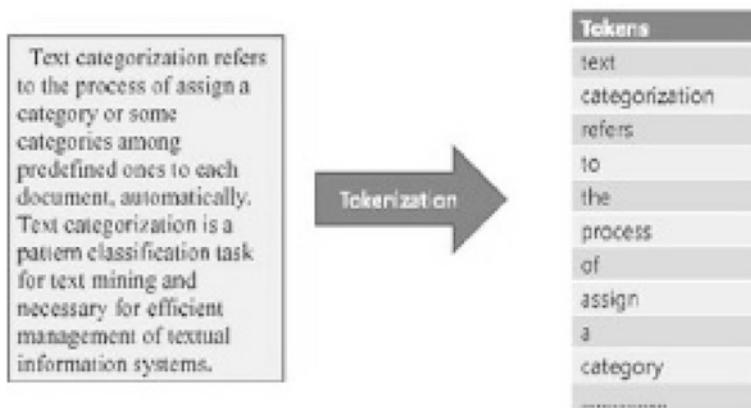


Fig. 13.3 Tokenization example

### 13.2.2 Pooling Types

This section is concerned with the stemming as the second step of text indexing. In the previous section, we studied the tokenization which is the process of segmenting a text into tokens. In this section, we study the stemming which is the process of converting each token into its grammatical root form. The cases of stemming are to change verbs which are transformed by their tenses into their root forms and to change plural forms of nouns into their singular forms. This section is intended to describe the stemming as the second step of indexing a text.

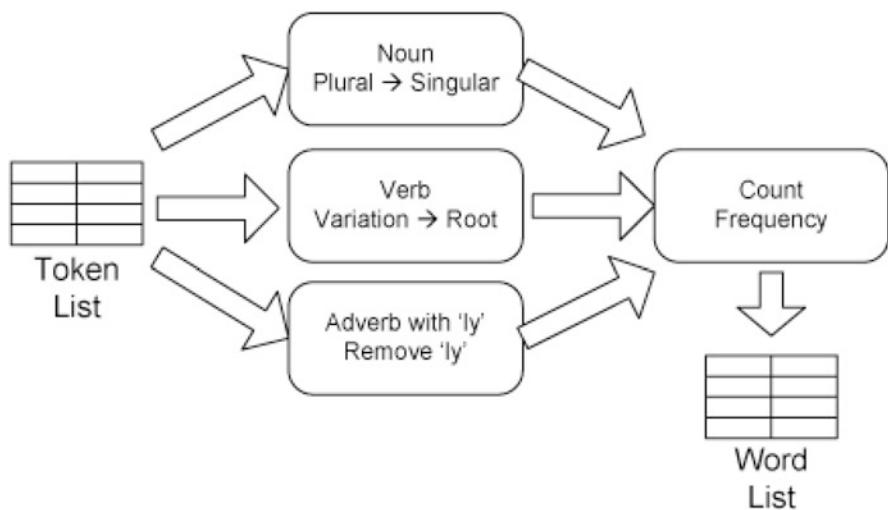
Some stemming rules for converting a word into its root form are illustrated in Fig. 13.4. If a word is a plural noun, it is converted into its singular form. If a word is a varied verb, it is converted into its root form. If a word is an adverb with its postfix, “ly,” it may be converted into the adjective by removing the postfix. The stemming is intended to make words with same meaning into ones with their identical spellings.

The process of stemming each word is illustrated in Fig. 13.5. A list of tokens is generated from the tokenization, and the stemming rules are loaded externally. If the current word is registered in the stemming rules, it is replaced by its associated one, and it is skipped, otherwise. The stemming is viewed as mapping the token list into the word list which consists of words in their root forms. In the more restricted stemming version, we may consider mapping a noun with the postfix “ation” into a verb.

Let us present the results from stemming the tokens as an example in Fig. 13.6. The stemming is the process of converting each token into its root form for preventing different treatments of words with their identical meanings. The nouns which are given as their plural forms, such as systems and techniques, are converted into ones in their singular forms, such as system and technique. The verbs which are given in their third personal singular form or passive form, such as has and

**Fig. 13.4** Stemming rules

Varied Form	Root Form
refers	refer
referred	refer
referring	refer
quick	quickly
done	do
did	do
does	do
children	child
remains	remain
has	have
had	have
having	have

**Fig. 13.5** Stemming process

progressed, are converted into verbs in their root forms such as have and progress. The process of stemming tokens is implemented by loading the stemming rules from a file in [8].

Text/ categorization/ refers→refer/ to/ the/ process/ of/ assign/ a/ category/ or/ some/ categories→category /among /predefined→predefine/ ones→one/ to/ each/ document/, automatically→automatic/. Text/ categorization/ is/ a/ pattern/ classification/ task/ for/ text/ mining/ and/ necessary/ for/ efficient/ management/ of/ textual/ information /systems→system/. In/ the/ academic/ world/, research/ on/ text/ categorization/ has→have/ been→be/ progressed→progress/ very/ much/, and/ we/ will/ survey/ it/ in/ next/ section/. In/ the/ industrial/ world/, text/ categorization/ systems→system / were→be/ already/ developed→develop/ as/ an/ independent/ system/ or/ a/ module/ for/ textual/ information/ systems→system/. /Although/ research/ and/ development/ on/ text/ categorization/ have/ been→be / progressed→progress/ like/ this/, /we/ need/ further→far/ research/ on/ it/ to/ improve/ techniques→technique/ and/ implementations→implementation/ of/ text/ categorization/.

**Fig. 13.6** Stemming example

Let us make some remarks on the stemming as the second step of indexing a text. The stemming rules are given as a table where each token is associated with its own root form. Each token is changed into its root form by retrieving its associate word if it is registered. Plural nouns are transformed into singular forms, and verbs in their varied forms are transformed into rooted ones. We may consider converting individual tokens into semantic word cluster identifier, rather than its root form, in the stemming.

### 13.2.3 Stop Word Removal

This section is concerned with the third step of text indexing which is called stop word removal. In the previous section, we studied the previous step of text indexing, called stemming. The stop word is the grammatical word which is irrelevant to text contents, such as conjunction and preposition, and we need to remove them for more efficiency. A list of stop words is constructed in advance, and if the current word is registered in the stop word list, it is removed. This section is intended to describe the stop words and the process of removing them.

The stop words are listed in Fig. 13.7. The stop word is defined as one which functions only grammatically in a text. Because the stop words are irrelevant to the text contents, we need to remove them for more efficiency. The prepositions, the conjunctions, and the special verbs belong to stop word. The stop word list is made as a file in advance.

**Fig. 13.7** Stop word list

all	am
an	and
any	are
as	at
be	because
been	before
being	below
between	both
but	by
could	did
do	does
doing	down
during	each

The process of removing stop words from a list of words is illustrated in Fig. 13.8. A list of words and the list of stop words are given as the arguments. For each word in the input list, if it is a stop word, it is skipped, and otherwise, it is added to the output list. If the current word is registered in the stop word list, it is removed from the list. The word list is downsized in indexing a text as the effect of the stop word removal.

The text where the stop words are removed is illustrated as an example in Fig. 13.9. Each word with its arrow is one which is transformed into its root form by the stemming process. The words which are marked as gray characters are stop words which will be removed from the text. The words which belong to the preposition are main stop words, and words, such as “the,” “a,” and “an,” belong to stop word. The portion of stop words in the text is about half of the entire text; the list of words is downsized by the 50% through the stop word removal.

**Fig. 13.8** Stop word removal process

```

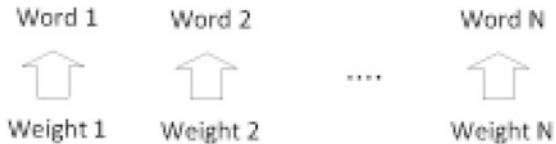
List removeStopwordList(List wordList, List stopWordList){
    int wordSize = wordList.size();
    int stopWordSize = stopWordList.size();
    List outputWordList = new List();
    for(int i = 0;i < wordSize; i++){
        Word word = wordList.getElement(i);
        boolean removeFlag = false;
        for(int j = 0;j < stopWordSize; j++){
            Word stopWord = stopWordList.getElement(j);
            if(word == stopWord)
                removeFlag = true;
        }
        if(!removeFlag)
            outputWordList.addElement(word);
    }
    return outputWordList;
}

```

Text/ categorization/ refers→refer/ to/ the/ process/ of/ assign/ a/ category/ or/ some/ categories→category /among /predefined→predefine/ ones→one/ to/ each/ document/, automatically→automatic/. Text/ categorization/ is/ a/ pattern/ classification/ task/ for/ text/ mining/ and/ necessary/ for/ efficient/ management/ of/ textual/ information /systems→system/. In/ the/ academic/ world/, research/ on/ text/ categorization/ has→have/ been→be/ progressed→progress/ very/ much/, and/ we/ will/ survey/ it/ in/ next/ section/. In/ the/ industrial/ world/, text/ categorization/ systems→system / were→be/ already/ developed→develop/ as/ an/ independent/ system/ or/ a/ module/ for/ textual/ information/ systems→system/. /Although/ research/ and/ development/ on/ text/ categorization/ have/ been→be / progressed→progress/ like/ this/, /we/ need/ further→far/ research/ on/ it/ to/ improve/ techniques→technique/ and/ implementations→implementation/ of/ text/ categorization/.

**Fig. 13.9** Stop word removal example

Let us make some remarks on the stop word removal as the third step of text indexing. The words which belong to the preposition and the conjunction belong to the stop word. The stop word removal is the process of removing one which registered in the stop word list. Half of the input list is removed by this process. There is no matter in swapping the stemming and the stop word removal in the steps of text indexing.

**Fig. 13.10** Word weighting

### 13.2.4 Additional Filtering

This section is concerned with the additional steps for indexing a text into words. In the previous section, we studied the basic three steps: tokenization, stemming, and stop word removal, and the output is a list of words. In this section, we study the assignment of weights to words, the additional removal of some words, and the addition of associative words from external sources. Depending on the application area, we need the additional steps for maximizing the performance of information retrieval. This section is intended to describe some additional steps which are needed optionally for indexing a text.

The process of weighting words as an additional step of text indexing is illustrated in Fig. 13.10. The need of discriminating the importance degrees of words which are extracted from indexing a text is the motivation for weighting the words. If a corpus is available, the weight is computed for each word by Eq. (13.1):

$$weight_i = \log_2 f_i + \log_2 \frac{F_i}{D}, \quad (13.1)$$

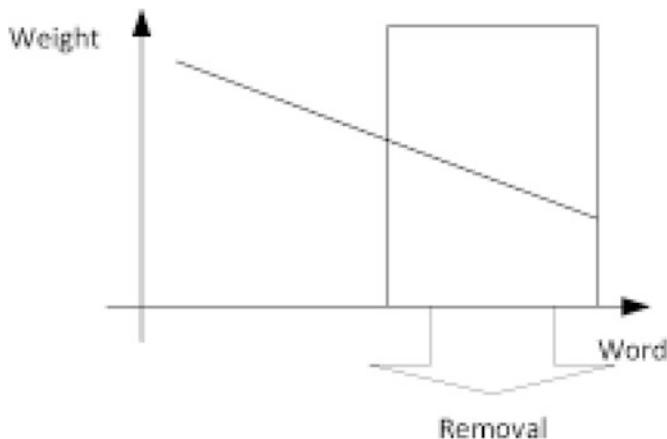
where  $f_i$  is the frequency of the word,  $word_i$  in the text;  $F_i$  is the total frequency of the word,  $word_i$ , in the corpus; and  $D$  is the number of texts in the corpus. If the corpus is not available, the weight is computed by Eq. (13.2):

$$weight_i = \frac{f_i}{|T|}, \quad (13.2)$$

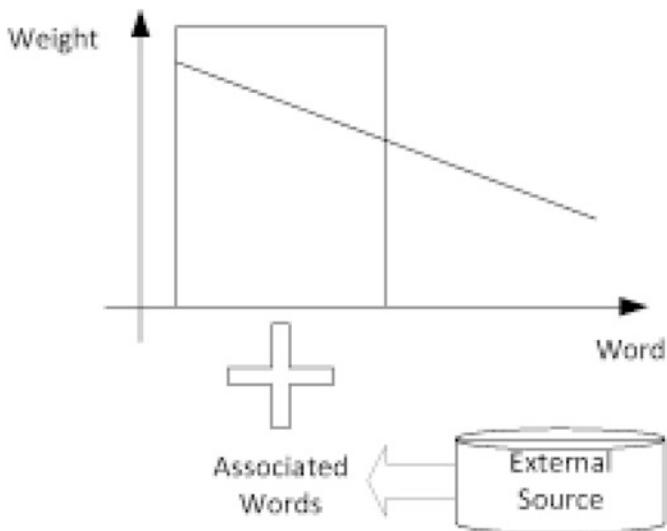
where  $|T|$  is the total number of words in the text. The word weights are used for the additional filtering of words by discriminating them.

The removal of additional words depending on their weights is illustrated in Fig. 13.11. The weights are computed for each word in the list which is built by the three basic steps. The words are ranked by their weights, and ones with their lower weights are removed. There are two schemes of removing the words: ranking selection which selects a fixed number of words and threshold selection which selects a variable number of words. The removal of additional words is intended to improve the efficiency by downsizing number of words in addition.

The index expansion by adding more words from external sources is illustrated in Fig. 13.12. Essential words are obtained from a text by removing ones with lower weights. Its semantically relevant words which are called associated words are retrieved by each selected word as a query. The semantical relevancy between



**Fig. 13.11** Index filtering



**Fig. 13.12** Index expansion

words is computed by their collocations in texts. The index expansion is intended to improve the precision and the recall in information retrieval systems.

Let us make some remarks on the additional steps in indexing a text into a list of words. After the three basic steps, a weight is assigned to each word for discriminating words. In addition, words with their lower weights are removed for more efficiency. Words with their lower weights are replaced by associated words to ones with their higher weights. The integration of the additional removal of words and addition of more associated words is called index optimization [4].

## 13.3 Semantic Similarity

This section is concerned with the computation of semantic similarity between words. In Sect. 13.3.1, we mention the process of encoding a word into a numerical vector. In Sect. 13.3.2, we describe the cosine similarity and its variants as similarity metrics between words. In Sect. 13.3.3, we describe the Euclidean distance as another similarity metric. In Sect. 13.3.4, we describe the similarity between tables, if words are encoded into tables.

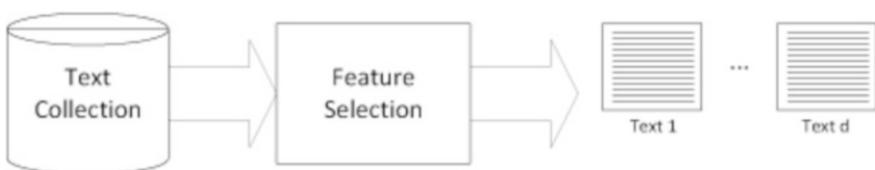
### 13.3.1 Word Representation

This section is concerned with the process of encoding a word into a numerical vector. In the previous section, we studied the process of a text into a list of words. In this section, we study the process of encoding a word into a numerical vector as the preparation for understanding the semantic similarity between two words. Texts are given as features for encoding a word so, and a feature value is given as the relationship between a word and a text. This section is intended to describe the steps which are involved in encoding words.

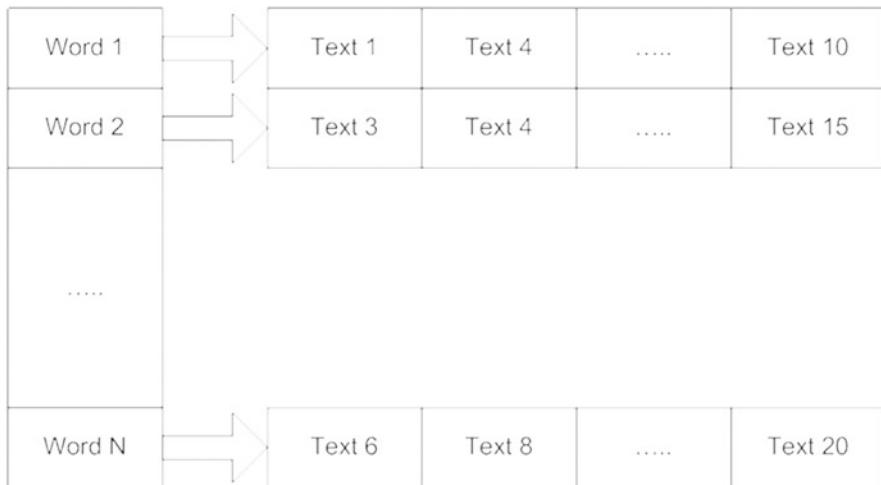
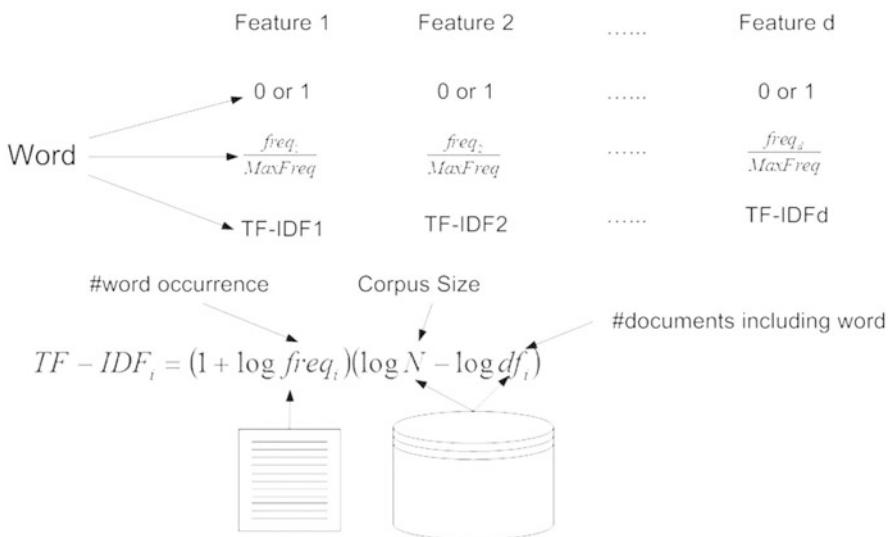
The process of extracting the feature for encoding a word into a numerical vector is shown in Fig. 13.13. A text collection is initially prepared; the texts in the collection are regarded as feature candidates. Sample words are prepared in advance, and the total frequencies of sample words are computed for each text. The texts with higher total frequencies are selected as features. The selected features are used for encoding a word into a numerical vector.

The inverted index which is the intermediate form for mapping a word into a numerical vector is shown in Fig. 13.14. The index is defined as a list of texts each of which is linked with words which are included. The inverted index is a list of words each of which is linked with texts which include itself. Each word is linked as a query to texts which are relevant to it. The inverted index may be constructed differently depending on the corpus domain.

The assignment of feature values to the attributes is illustrated in Fig. 13.15. A binary value which indicates the presence or the absence of a word in the text is used as a feature value. The relative frequency of a word in the text is a feature value.



**Fig. 13.13** Feature selection for word encoding

**Fig. 13.14** Inverted index for word encoding**Fig. 13.15** Feature value assignment for word encoding

The TF-IDF (term frequency-inverse document frequency) weight is computed by the equation, which is presented in Fig. 13.15, as a feature value. If a text is encoded into a numerical vector, words become features.

Let us make some remarks on the process of encoding a word into a numerical vector. Some texts are selected from a text collection as the features for mapping a word so. The inverted index where each word is linked with a list of texts which includes itself is constructed. A value which indicates a relationship between a word

and a text is assigned to each feature. Association words or words with similar meanings are used as features.

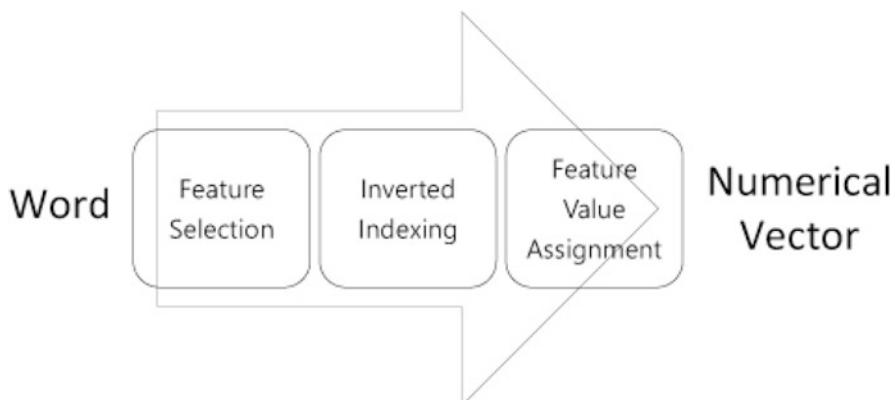
### 13.3.2 Cosine Similarity

This section is concerned with the cosine similarity as a semantic similarity between words. In the previous section, we studied the process of encoding a word into a numerical vector. If two words are encoded so, the cosine similarity between two vectors is computed as the semantic similarity between them. Some variants may be derived from the cosine similarity, and one among them is adopted in computing the similarity between words. This section is intended to describe the process of computing the cosine similarity between two vectors.

The process of encoding a word into a numerical vector is illustrated in Fig. 13.16. Some texts are selected from a corpus by a particular criterion. The inverted index where each word is linked to texts which include itself is constructed. Based on the inverted index, each feature value is computed as a relationship between a word and a text. The similarity between two vectors is one between words.

Let us mention the cosine similarity as the similarity metric between two numerical vectors. Two vectors are notated by  $\mathbf{x}_1 = [x_{11} \ x_{12} \ \dots \ x_{1d}]$  and  $\mathbf{x}_2 = [x_{21} \ x_{22} \ \dots \ x_{2d}]$ . The cosine similarity is defined by Eq. (13.3), as the general form:

$$\text{sim}(\mathbf{x}_1, \mathbf{x}_2) = \frac{\mathbf{x}_1 \cdot \mathbf{x}_2}{\|\mathbf{x}_1\|_p \cdot \|\mathbf{x}_2\|_p}. \quad (13.3)$$



**Fig. 13.16** Word encoding process

Because of  $\mathbf{x}_1 \cdot \mathbf{x}_2 \leq \|\mathbf{x}_1\|_p \cdot \|\mathbf{x}_2\|_p$ , the value of cosine similarity is always given as a normalized value between zero and one. In  $p = 2$ , the cosine similarity is expressed as Eq. (13.4):

$$\text{sim}(\mathbf{x}_1, \mathbf{x}_2) = \frac{\mathbf{x}_1 \cdot \mathbf{x}_2}{\|\mathbf{x}_1\| \cdot \|\mathbf{x}_2\|}, \quad (13.4)$$

where  $\|\mathbf{x}_1\| = \sqrt{\sum_{i=1}^d x_{1i}^2}$  and  $\|\mathbf{x}_2\| = \sqrt{\sum_{i=1}^d x_{2i}^2}$ .

Let us mention three cosine variants which is derived from Eq. (13.4). The first variant is defined as Eq. (13.5):

$$\text{sim}(\mathbf{x}_1, \mathbf{x}_2) = \frac{2\mathbf{x}_1 \cdot \mathbf{x}_2}{\|\mathbf{x}_1\| + \|\mathbf{x}_2\|}, \quad (13.5)$$

The second cosine similarity variant, called Jaccard similarity is defined as Eq. (13.6):

$$\text{sim}(\mathbf{x}_1, \mathbf{x}_2) = \frac{\sum_{i=1}^d \min(x_{1i}, x_{2i})}{\max(x_{1i}, x_{2i})}, \quad (13.6)$$

The last cosine similarity variant which is called Tanimoto similarity is defined as Eq. (13.7):

$$\text{sim}(\mathbf{x}_1, \mathbf{x}_2) = \frac{\mathbf{x}_1 \cdot \mathbf{x}_2}{\|\mathbf{x}_1\|^2 + \|\mathbf{x}_2\|^2 - \mathbf{x}_1 \cdot \mathbf{x}_2}, \quad (13.7)$$

The feature similarities may be considered in defining the similarity metric between two vectors [9].

Let us make some remarks on the similarity metric between words. A word is encoded into a numerical vector whose features are texts. The cosine similarity between two vectors is used as the semantic similarity between words. Some variants may be derived from the cosine similarity, and one among them may be adopted. We consider computing multiple similarity metrics and taking average over them.

### 13.3.3 Euclidean Distances

This section is concerned with another similarity metric which is called Euclidean distance. In the previous section, we studied the cosine similarity as the similarity metric between two vectors, and it is fragile to zero vectors. In this section, we study the Euclidean distance which is based on the difference between elements in two vectors, as another similarity metric. Its merit is the higher tolerance to sparse

vectors where zero values are dominant. This section is intended to describe the similarity metric between two vectors.

Let us mention the norm of a vector in the general form. The simplest version of the norm is defined by Eq. (13.8):

$$\|\mathbf{x}\| = \sqrt{\mathbf{x} \cdot \mathbf{x}} = \sqrt{\sum_{i=1}^d x_i^2}. \quad (13.8)$$

Equation (13.8) is expanded into the more general form as shown in Eq. (13.9):

$$\|\mathbf{x}\|_p = \left( \sum_{i=1}^d x_i^p \right)^{1/p}. \quad (13.9)$$

if  $p \rightarrow \infty$ ,  $\|\mathbf{x}\|_\infty = \max_{i=1}^d x_i$  and if  $p = 1$ ,  $\|\mathbf{x}\|_1 = \sum_{i=1}^d x_i$ .  $\|\mathbf{x}\|_1$  and  $\|\mathbf{x}\|_2$  are popular versions of the norm.

Let us mention a simple Euclidean distance between two numerical vectors. They are notated by  $\mathbf{x}_1 = [x_{11} \ x_{12} \ \dots \ x_{1d}]$  and  $\mathbf{x}_2 = [x_{21} \ x_{22} \ \dots \ x_{2d}]$ . The Euclidean distance is defined by Eq. (13.10):

$$ED(\mathbf{x}_1, \mathbf{x}_2) = \sqrt{\sum_{i=1}^d (x_{1i} - x_{2i})^2} \quad (13.10)$$

The similarity between two vectors is the reverse Euclidean distance. Its value is a non-normalized value.

Let us mention the general form of Euclidean distance. Equation (13.10) is used as the most popular version. The absolute version of Euclidean distance is expressed as Eq. (13.11):

$$ED(\mathbf{x}_1, \mathbf{x}_2) = \sum_{i=1}^d |x_{1i} - x_{2i}| \quad (13.11)$$

The general form of Euclidean distance is derived as expressed in Eq. (13.12):

$$ED(\mathbf{x}_1, \mathbf{x}_2) = \left( \sum_{i=1}^d (x_{1i} - x_{2i})^p \right)^{1/p} \quad (13.12)$$

The difference between two vectors is basis of the general version as shown in Eqs. (13.10), (13.11), and (13.12).

Let us make some remarks on the Euclidean distance between numerical vectors. The norm of a vector means its size; a vector is represented as a single scalar value. The most popular version of Euclidean distance is expressed as Eq. (13.10). We need to find the way of normalizing the Euclidean distance value.

### 13.3.4 Table Similarity

This section is concerned with the similarity metric between tables which represent words. In the previous section, it is assumed that a word is encoded into a numerical vector. It is recently proposed that a word should be encoded into a table as the alternative representation [5], and the similarity between table is considered in this section. Each entry consists of a text identifier and its weight which is a relationship between a word and a text. This section is intended to describe the similarity metric between tables which represent words.

Let us mention some previous works on encoding a text into tables. In 2007, Jo and Cho initially tried to encode a text into a table for carrying out the text classification [2]. In 2015, Jo proposed the similarity metric between tables which is given as a normalized value [3]. In 2018, Jo modified the KNN algorithm into its table-based version as an approach to the text classification [6]. This section covers the similarity metric between tables which is used in the above literatures.

Let us mention the process of converting a word into a table. A corpus which consists of texts is prepared, and texts which include the word are gathered from it. The weight of the word in the text is computed, and the table which represents a word consists of entries each of which consists of a text identifier and a weight. The table may be downsized by removing the entries with their lower weights for the efficient processing. The similarity between words is computed by representing them into tables.

Let us mention the process of computing the similarity between tables as the semantic similarity between words. Two words are encoded into tables by the above process. Shared text identifiers are extracted from the two tables, and its rate to the total entries is computed. The rate is the similarity between tables, and the similarity is given as a normalized value between zero and one. The shared text identifiers between two tables are collocations of the two words in texts.

Let us make some remarks on encoding words into tables for computing their similarities. The research on encoding words or texts into tables has been progressed. The table which represents a word is a set of entries each of which consists of a text identifier and a weight. The similarity between tables is computed based on common entries as the semantic similarity between words. Refer to [6] for getting the detail information about the similarity between tables.

## 13.4 Expansion Schemes

This section is concerned with schemes of expanding the index. In Sect. 13.4.1, we study associated words which are semantically relevant to words in the index. In Sect. 13.4.2, we study associated texts which are relevant to the input text. In Sect. 13.4.3, we study the scheme of expanding the index, depending on the information retrieval system. In Sect. 13.4.4, we study the index optimization with the functional view.

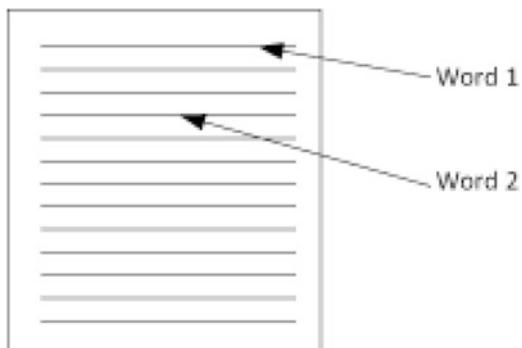
### 13.4.1 Associated Words

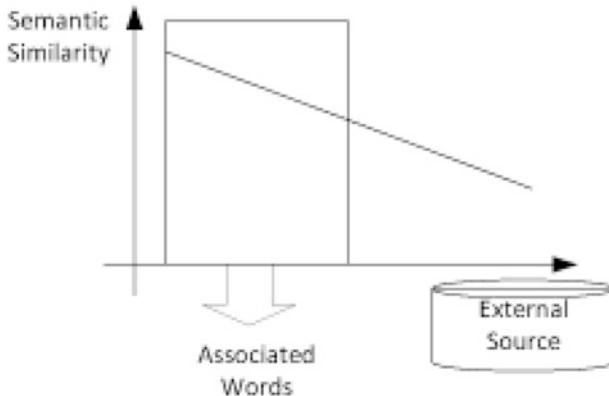
This section is concerned with associated words which is necessary for expanding the index. In Sect. 13.3, we studied the similarity metrics between words. Associated words are retrieved from a particular word, using a similarity metric. The collocation which is location of two words in a same text is the basis for retrieving associated words. This section is intended to describe the process of retrieving associated words from a word.

The collocation of two words is conceptually illustrated in Fig. 13.17. The collocation is the phenomena where two words appear in a same text. Word 1 and Word 2 appear in a same text as shown in Fig. 13.17. The large number of texts where two words collocate is viewed as the high semantic similarity between them. The collocation is used as the basis for computing the semantic similarity between words.

Let us compute the similarity between the two words,  $word_1$  and  $word_2$ , based on their collocations. The corpus which consists of texts is prepared in advance. The number of texts which include both words is  $\#(word_1, word_2)$ ; the number of texts which include the word,  $word_1$ , is  $\#(word_1)$ ; and the number of texts which include the word,  $word_2$  is  $\#(word_2)$ . The similarity between the two words is computed by Eq. (13.13):

**Fig. 13.17** Collocation of two words





**Fig. 13.18** Retrieval of associated words

$$sim(word_1, word_2) = \frac{2 \times \#(word_1, word_2)}{\#(word_1) + \#(word_2)}. \quad (13.13)$$

The similarity which is computed by Eq.(13.13) is given as a normalized value between zero and one.

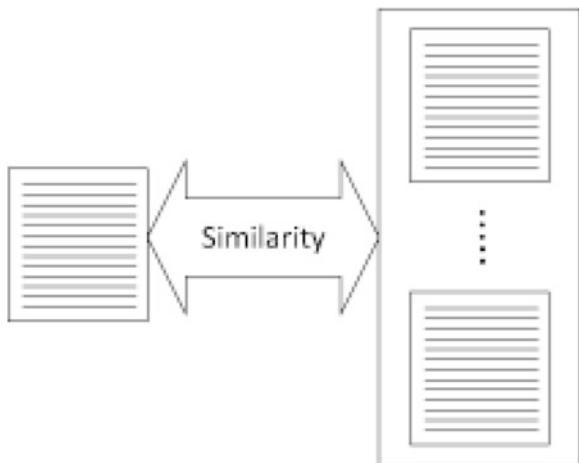
The process of retrieving associated words to the input word is illustrated in Fig. 13.18. It is assumed that texts in external sources are indexed into word lists. Their similarities with the input word are computed as their relevancy, and words are ranked by their relevancies. Some with their highest relevancies are selected and retrieved as the associated words. They are added to the word list which is indexed from the input text.

Let us make some remarks on the retrieval of associated words to a particular word. The collocation is the phenomena where two words locate in a same text. The semantic similarity between words is based on the rate of the number of texts with their collocations to the number of texts which include at least either of them. The words which are similar as the input word are retrieved from the external sources as its associated words. The associated words are used for expanding the index.

### 13.4.2 Associated Text

This section is concerned with associated texts for expanding the index. In the previous section, we studied the associated words which are added to the existing index as a scheme of expanding the index. In this section, we consider an associated text which is most similar as the current input text which is used for expanding the index. We review the process of computing the similarity between texts before retrieving the associated texts from external sources. This section is intended to describe the process of retrieving associated texts.

**Fig. 13.19** Text similarity computation

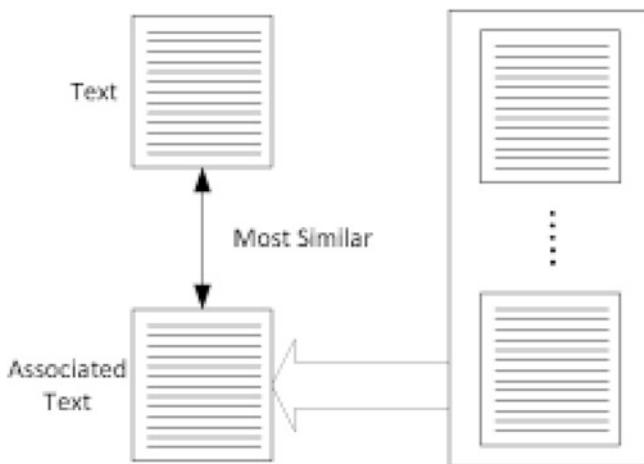


The process of measuring the similarity of a text with others is illustrated in Fig. 13.19. The first step of retrieving an associated text is to compute the similarity of the input text with others. The process of computing the similarity between texts is to encode them into numerical vector and compute the cosine similarity as the similarity between them. As the alternative way, two texts are indexed into the two word lists, and the rate of shared words is the basis of computing the similarity. Texts in external sources are ranked by their similarities with the input text.

The process of taking the most similar text as the associated text from external sources is illustrated in Fig. 13.20. The similarities of the input text with external text are computed by the above process. The external texts are ranked by their similarities, and one or some with their maximal similarities are taken as the associate text or texts. As the alternative way, the similarity threshold is decided, and the texts with their higher similarities than the threshold are taken. The associated texts are used for expanding the index.

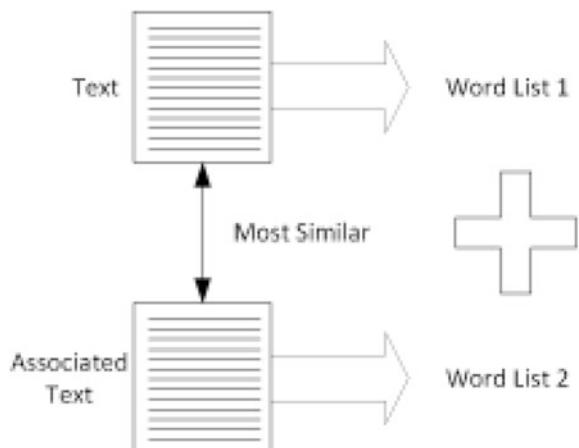
The process of expanding the index using the associated texts is illustrated in Fig. 13.21. A single text or multiple texts which are most similar as the input text are taken as the associated text or text. The input text and the associated texts are indexed into lists of words. The lists are integrated into a single list of words. We need to remove words with their lower importance by weighting them.

Let us make some remarks on the index expansion with associate texts. The similarity between texts is taken by encoding them into numerical vectors and computing the cosine similarity. The text which is most similar as the input text is retrieved from external sources as the associated text. The index is expanded by adding words from the associated text. We consider deciding the number of associated texts, depending on the input text length.



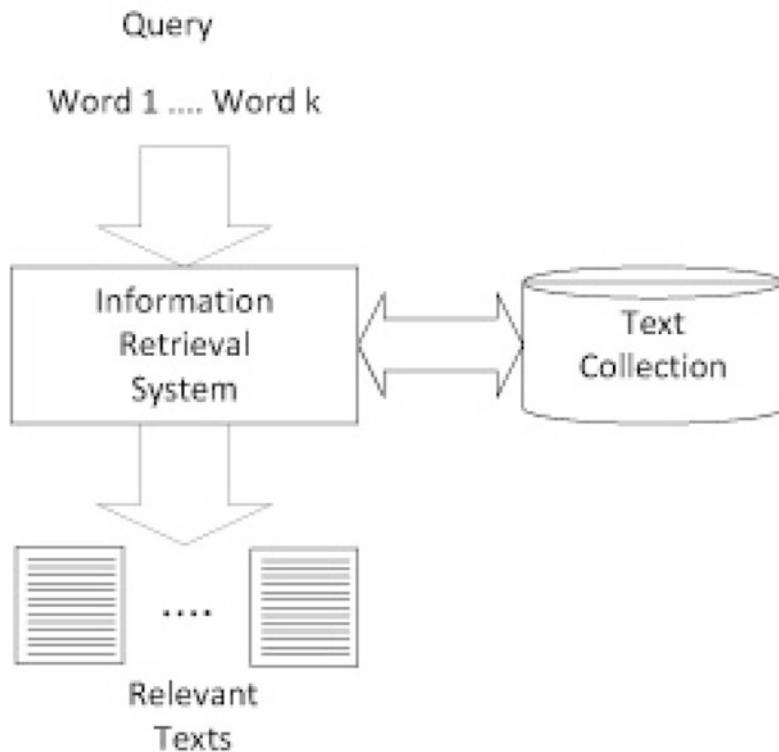
**Fig. 13.20** Retrieval of most similar text as associated text

**Fig. 13.21** Index expansion  
by associated text



### 13.4.3 Information Retrieval-Based Scheme

This section is concerned with the third scheme of expanding the index, called information retrieval-based scheme. In the previous sections, we mentioned the associated words and texts as means of expanding the index. The information retrieval which is mentioned in this section is the process of retrieving the texts which are relevant to the given query. The input text is used as the query to the information retrieval system in the index expansion. This section is intended to describe the scheme of expanding the index, called information retrieval-based scheme.

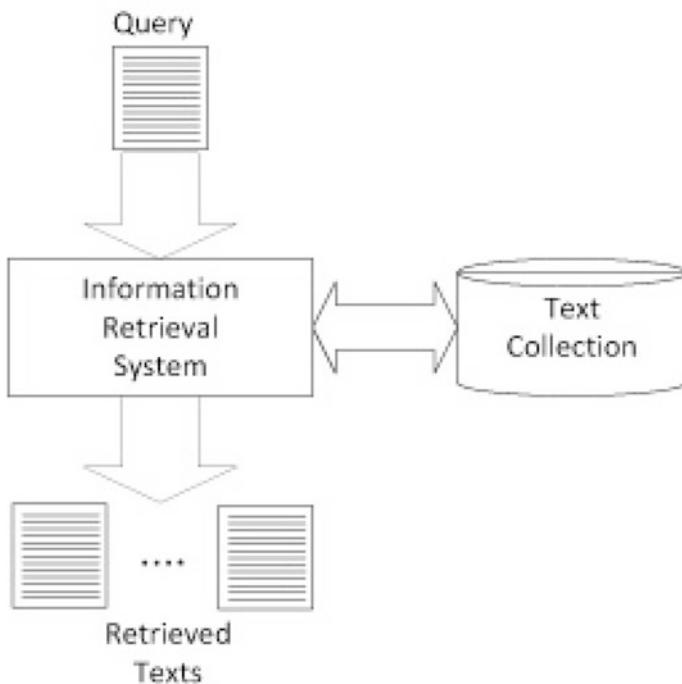


**Fig. 13.22** Information retrieval system

The information retrieval system is illustrated as a block diagram in Fig. 13.22. A query is given by a user as a list of words. The relevancies of the query to texts which are provided by external sources are computed. The texts which are relevant to the query are retrieved to the user. A text may be used as the query instead of a list of words.

The information retrieval system whose query is a text is illustrated in Fig. 13.23. The initial query is given as a text, and it is indexed into a list of words. The texts which are relevant to the words are retrieved from external sources. As the alternative way, the query text is encoded into a numerical vector, and its similarities with ones which represent external texts are computed as its relevancies. Some are selected among texts which are retrieved from the system.

The process of expanding the index by the retrieved texts is illustrated in Fig. 13.24. The texts are retrieved from the information retrieval system by the input text which is given as the query. Some are selected among the retrieved ones, and they are indexed into a list of words. Some words with their higher weights are selected among the words and added to the list of words which is indexed from the



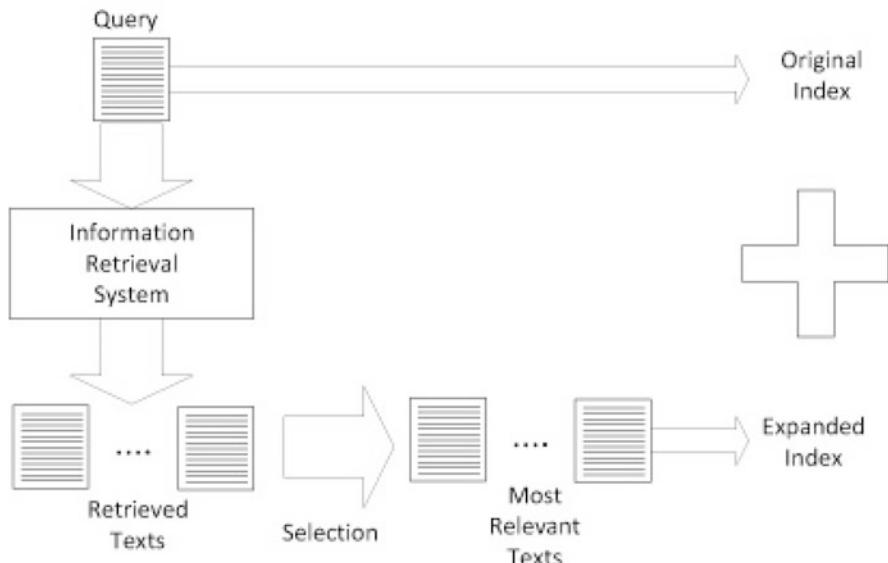
**Fig. 13.23** Retrieval of texts with text query

input text. Dependency on commercial information retrieval systems characterizes this scheme.

Let us make some remarks on the index expansion based on the information retrieval. It is the process of retrieving texts which are relevant to the query, as the basis for expanding index. If a text is given as the query, it is converted into a list of words, and the relevancies of texts in external sources are computed. Most relevant texts are retrieved, and they are indexed into a list of words, and some among them are added to the index of input text. If the retrieved texts are long, we may consider only a paragraph or a subtext from the most relevant texts.

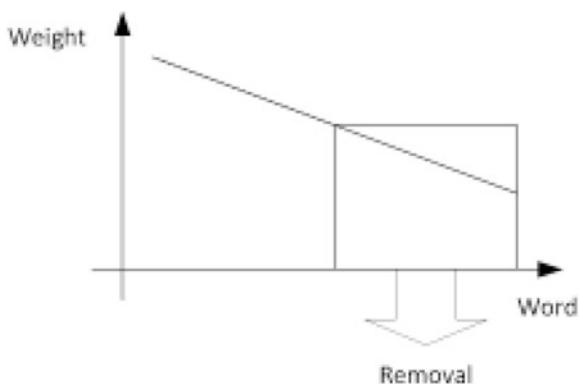
#### 13.4.4 Index Optimization

This section is concerned with the index optimization which covers both the index expansion and the index filtering. In the previous section, we mentioned the schemes of expanding the index with more relevant words. In this section, we consider optimizing the index with both the index expansion and the index filtering. The index optimization was interpreted into a binary classification which classifies a



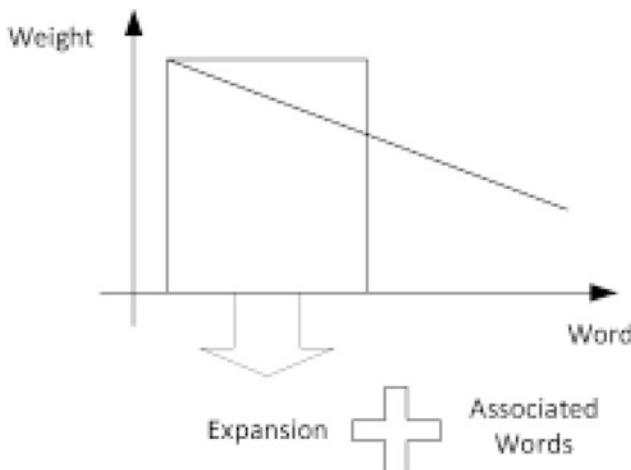
**Fig. 13.24** Index expansion by information retrieval system

**Fig. 13.25** Index filtering



word into expansion, inclusion, or removal in [4]. This section is intended to describe the index optimization which involves both.

The index filtering which is the process of removing some words from the index is illustrated in Fig. 13.25. It is the process of cutting down words with their lower weights from the index for improving the efficiency. The process of index filtering is to compute word weights and remove words with their lower weights. The index filtering is applied to a long text; the long text should be reduced for improving the efficiency. The index filtering is used for replacing unnecessary words by necessary ones.



**Fig. 13.26** Index expansion



**Fig. 13.27** Classification of word into its importance level

The index expansion which is applied to the text indexing is illustrated in Fig. 13.26. The index expansion is the process of adding more words from external sources as associated ones to important words. The weights which are computed by an equation are assigned to the words, and the associated words are retrieved and added to ones with their higher weights. If a short text is indexed, the index expansion is applied; we need to improve the reliability in processing texts. The index expansion is used to reinforce the important words by adding ones which are semantically similar as them.

The classification of each word into one of the three levels is illustrated in Fig. 13.27. We need to remove unimportant words and reinforce important words, whether a text is short or long. If a word is classified into expansion, its associated words should be retrieved from the external sources and added. If a word is classified into inclusion, it is included without adding its associated ones, and if a word is classified into removal, it should be removed. The research on the index optimization which is viewed as a classification has progressed [4].

Let us make some remarks on the index optimization which is the compound of the index filtering and the index expansion. The index filtering is the process of selecting and removing unimportant words in indexing a text. The index expansion

is the process of retrieving and adding associated words to the important words. Words may be classified into one of the three levels, expansion, inclusion, and removal for implementing the index optimization. The semantic text index is implemented by the word dictionary where each word is associated with its related ones.

## 13.5 Summary and Further Discussions

This section is concerned with the summary and the further discussions on what is studied in this chapter. A text is indexed into a list of words with the three steps: tokenization, stemming, and stop word removal. The semantic similarity between words is computed by encoding them into numerical vectors whose features are texts. The index may be expanded by retrieving associated words or texts. This section is intended to discuss further what is studied in this chapter.

Let us mention the POS (part of speech) tagging as a traditional task in the natural language processing. The POS tagging is defined as the process of tagging each word with its own grammatical function. It is viewed as a classification task where each word is classified into one among the grammatical categories. They are predefined as the eight word classes, and words each of which is labeled with one of the eight classes are prepared as the training example. The research on the application of machine learning algorithm to the POS tagging has progressed very much.

Let us consider the lexical similarity between words. It is the similarity between words or strings based on their characters. The lexical similarity is computed, depending on the number of shared character and their orders. The lexical similarity between texts was adopted for defining the string kernel for applying the SVM to the text classification in [1]. However, there is no correlation between the lexical similarity and the semantic similarity of two words in any natural language.

Let us consider some points in indexing a long text. Many words tend to be generated in doing so; it becomes the cause of retrieving long texts frequently in information retrieval system. Because a long text has too many irrelevant parts, it is inefficient for users. It is necessary to segment a long text into subtexts based on its contents. The research on the text segmentation has progressed for solving the problem in [7].

The text indexing which is covered in this chapter may be interpreted as a textual pooling. It is the process of converting a text into a list of words. The textual deep operation is defined as the process of generating another textual data from an input text. In the text indexing, a list of words is generated from a raw text as another type of textual data. Mentioning the text indexing in this chapter is intended to use the process for implementing the textual deep operation.

## References

1. H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, and C. Watkins, “Text Classification with String Kernels, pp419–444, Journal of Machine Learning Research, Vol 2, No 2, 2002.
2. T. Jo and D. Cho, “Index Based Approach for Text Categorization”, 127–132, International Journal of Mathematics and Computers in Simulation, 2, 2007.
3. T. Jo, “Normalized Table Matching Algorithm as Approach to Text Categorization”, 839–849, Soft Computing, 19, 4, 2015.
4. T. Jo, “Graph based KNN for Optimizing Index of News Articles”, 53–62, Journal of Multimedia Information System, 3, 2016.
5. T. Jo, “Keyword Extraction in News Articles using Table based K Nearest Neighbors”, The Proceedings of 25th International Conference on Computational Science & Computational Intelligence, 2018.
6. T. Jo, “Modification into Table based K Nearest Neighbor for News Article Classification”, 49–50, The Proceedings of 1st International Conference on Advanced Engineering and ICT-Convergence, 2018.
7. T. Jo, “Content based Segmentation of News Articles using Feature Similarity based K Nearest Neighbor”, 61–64, The Proceedings of 19st International Conference on Information and Knowledge Engineering, 2019.
8. T. Jo, “Text Mining: Concepts and Big Data Challenge”, Springer, 2019.
9. T. Jo, “Content based Segmentation of News Articles using Feature Similarity based K Nearest Neighbor”, 61–64, The Proceedings of 19st International Conference on Information and Knowledge Engineering, 2019.

# Chapter 14

## Text Summarization



This chapter is concerned with the text summarization as an instance of text mining. It refers to the process of extracting automatically essential parts as the summary. In the process of the text summarization, a text is partitioned into paragraphs, and important ones among them are selected as its summary. The text summarization is viewed as mapping a text into a hidden text in implementing the textual deep learning. This section is intended to describe the text summarization with the view of implementing the textual deep learning.

This chapter is composed with the five sections, and in Sect. 14.1, we overview what is studied in this chapter. In Sect. 14.2, we study the process of abstracting a text generally. In Sect. 14.3, we study process of summarizing a text biased to a query. In Sect. 14.4, we study the process of summarizing multiple texts. In Sect. 14.5, we discuss further what is studied in this chapter.

### 14.1 Introduction

The text summarization refers to the process of extracting essential part from a text automatically. The manual text summarization is the process of rewriting its content into a brief form; the automatic summarization and the manual summarization should be defined differently. The process of text summarization is to partition a text into sentences or paragraphs and select important sentences or paragraphs as the summary. If applying a machine learning (ML) algorithm to this task, the text summarization may be viewed into a binary classification which classifies each sentence or each paragraph into summary or non-summary. This section is intended to overview the text summarization as the introduction to this chapter.

Let us mention the manual summarization of a text which is performed by human being. The manual text summarization is defined as the process of rewriting the entire contents into its brief form. The process of manual text summarization is to scan an entire text and describe its content into a shorter text. A text is summarized

differently depending on a subject; the manual summarization is characterized by summarizing it subjectively. It costs very much time for summarizing texts manually; it motivates the automatic text summarization.

The automatic text summarization is what is covered in this chapter. It is defined as the process of extracting the essential part from a text as its summary. The automatic text summarization process is to partition a text into sentences or paragraphs and extract essential ones among them as its summary. The text summarization is viewed into a binary classification, to apply a machine learning algorithm to the task. In this study, we treat the text summarization as an instance of textual deep operation.

The text summarization which is studied in this chapter is intended to present the textual deep operation. The dimension of numerical vector which represents a data item is reduced by selecting a represent value from each window which slides on the input vector in the pooling operation. The text summarization is viewed as the process of selecting the representative part from a raw text; this task looks like the pooling operation. The text summarization is expanded into the summarization of textual window which slide on a raw text which consists of paragraphs. In the next chapter, we will study using the text summarization for implementing the textual deep operation.

Let us mention what is intended in this chapter. We understand the process of abstracting a text as the unbiased summarization. We understand the query-based summarization as the biased one to the query. We understand the summarization of multiple texts as another type of text summarization. This chapter is intended to study the three types of text summarization for implementing the textual deep operations.

## 14.2 Abstracting

This section is concerned with the schemes of abstracting a full text. In Sect. 14.2.1, we describe the phrase-based abstracting as the simplest scheme and point out its limits. In Sect. 14.2.2, we describe the keyword-based abstraction as the second scheme of abstracting a full text. In Sect. 14.2.3, we interpret the abstracting into a binary classification. In Sect. 14.2.4, we apply a machine learning algorithm to the text abstracting.

### 14.2.1 *Phrase-Based Abstracting*

This section is concerned with a scheme of abstracting a text, depending on relevant phrases, called phrase-based abstracting. The abstracting is the process of extracting the summary without any bias to the information need. The process of the phrase-based abstracting is to decide some phrases which indicate the summary, in advance, and select paragraphs which include one among them as the summary. The limit

**Fig. 14.1** Key phrases

In summary, ...  
 Summarizing ...  
 The summary of this article ....  
 Telling the summary, ....  
 In conclusion, ...  
 Concluding ....  
 Finally, ...  
 .....

**Fig. 14.2** Process of summarizing a text based on key phrases

```
String summarizeFullText(String fullText, List phraseList){
    String paragraphList[] = fullText.split("\n");
    int paragraphSize = paragraphList.size();
    int phraseSize = phraseList.size();
    for(int i = 0; i < paragraphSize; i++){
        for(int j = 0; j < phraseSize; j++){
            String phrase = phraseList.getElement(j);
            if(paragraphList[i].contains(phrase))
                return paragraphList[i];
        }
    }
    return "No Summary";
}
```

of this scheme is that if the text does not include any of phrases, no paragraph is extracted. This section is intended to describe the first scheme of abstracting a text, called phrase-based abstracting.

The key phrases for deciding the summary are presented in Fig. 14.1. The phrase, “in summary” is usually given as start of the last paragraph, so the last paragraph tends to be extracted as the summary. The sentence which follows the phrase is given as the summary of the current article. The phrases, such as “concluding,” “in conclusion,” and “finally,” are given as the start of the last paragraph. If it depends on the key phrases, the last paragraph is selected as the summary with higher probability.

The process of summarizing a text by the key phrases is illustrated as a pseudocode in Fig. 14.2. The list of phrases which indicate an essential paragraph is constructed manually in advance. The input text is segmented into paragraphs by the carriage return, and for each paragraph, it checks whether it includes one among the key phrases at least. If any of the key phrases is included in the paragraph, it is added to the list, and the list of paragraphs is returned as the summary. If there is no paragraph which includes any key phrase, it extracts no summary.

Let us point out some limits from this scheme of summarizing a text. It is very tedious to add key phrases as a list, manually. Because no paragraph with one among the key phrases happens frequently, the summary is missed very frequently. There is no guarantee that a paragraph with one among the key phrases is summary.

The scheme is used together with another scheme which is covered in subsequent sections, for extracting summary more reliably.

Let us make some remarks on the key phrase-based method of the text summarization. In this scheme, a list of key phrases which indicate the summary is constructed manually. The paragraph which includes one among the key phrases is extracted as the summary. If no paragraph has one among the predefined key phrases, the summary is not extracted from the text. If so, we may consider the partial matching of key phrases.

### **14.2.2 Keyword-Based Abstracting**

This section is concerned with the second scheme of summarizing a text, called keyword-based abstracting. In the previous scheme, the paragraph which includes any key phrase is extracted as the summary. In this scheme, if an article is initially associated with its keywords, some paragraphs are selected as the summary, depending on the distribution over keywords. We may develop techniques of extracting keywords automatically from a text. This section is intended to describe the abstraction scheme, depending on keywords.

An article which is associated with its keywords is illustrated in Fig. 14.3. In this scheme of summarizing a text, it is assumed that the author writes an article, tagging its keywords. A typical example of this article is an abstract of a research paper which is associated with its keywords. An article and its keywords are constructed as an XML document. Various techniques of extracting keywords automatically have been developed to process articles which are not tagged with keywords.

The process of summarizing a text based on the keywords is illustrated as a pseudocode in Fig. 14.4. The input text is partitioned into paragraphs, and a list of keywords is provided together with the input text. The total frequency of the keywords is counted for each paragraph. The paragraph with its maximal total frequency is returned as the summary of the input text. In this summarization scheme, it is assumed that the input text is associated with its own keywords.

The process of summarizing a text based on the keyword weights is illustrated as a pseudocode in Fig. 14.5. In the previous scheme, the process of summarizing a text depends on the total frequency of the keywords in each paragraph. Each

**Fig. 14.3** Text + keywords



Keyword 1, Keyword 2, ..., Keyword k

```

String summarizeFullText(String fullText, List keywordList){
    String paragraphList[] = fullText.split("\n");
    int paragraphSize = paragraphList.size();
    int keywordSize = keywordList.size();
    int maxFrequency = 0;
    int maxIndex = 0;
    for(int i = 0; i < paragraphSize; i++){
        int totalFrequency = 0;
        for(int j = 0; j < keywordSize; j++){
            String keyword = keywordList.getElement(j);
            totalFrequency = totalFrequency + paragraphList[i].countFrequency(keyword);
        }
        if(maxFrequency < totalFrequency){
            maxFrequency = totalFrequency;
            maxIndex = i;
        }
    }
    return paragraphList.getElement(maxIndex);
}

```

**Fig. 14.4** Keyword counted summarization process

```

String summarizeFullText(String fullText, List keywordList){
    String paragraphList[] = fullText.split("\n");
    int paragraphSize = paragraphList.size();
    int keywordSize = keywordList.size();
    double maxWeight = 0.0;
    int maxIndex = 0;
    for(int i = 0; i < paragraphSize; i++){
        double totalWeight = 0.0;
        for(int j = 0; j < keywordSize; j++){
            String keyword = keywordList.getElement(j);
            totalWeight = totalWeight + paragraphList[i].computeWeight(keyword);
        }
        if(maxWeight < totalWeight){
            maxWeight = totalWeight;
            maxIndex = i;
        }
    }
    return paragraphList.getElement(maxIndex);
}

```

**Fig. 14.5** Keyword-weighted summarization process

keyword is initially associated with its weights, and the total weight of keywords is computed in each paragraph. The paragraph with its maximal total weight is returned as the summary of the input text. The difference from the above scheme is to assign initially weights to the keywords.

Let us make some remarks on this scheme of summarizing a text. In this scheme, the initial association of a text with its own keywords is assumed. The paragraph with the maximal total frequency of the keywords is selected as the summary.

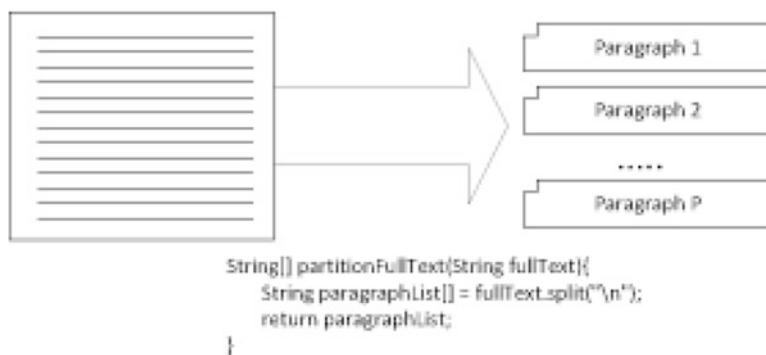
In a variant, the paragraph with the maximal total weight is selected. A machine learning algorithm is applied to the text summarization by encoding a paragraph into a numerical vector whose features are keywords.

### 14.2.3 Mapping Abstracting into Binary Classification

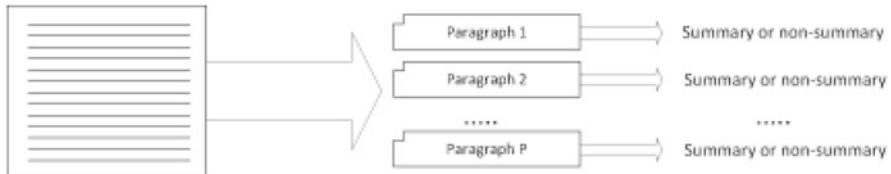
This section is concerned with the task of abstracting a text in the view of binary classification. Abstracting which is studied in the two previous sections is the judgment whether each paragraph or each sentence is a summary or a non-summary. In this section, the abstracting is viewed as the binary classification of each paragraph into summary or non-summary. A machine learning algorithm is applicable to the abstracting which is defined as a binary classification. This section is intended to describe the interpretation of abstracting into the classification of a paragraph into summary or non-summary.

The partition of a text into paragraphs is illustrated in Fig. 14.6, together with the pseudocode. It is assumed that a text has more than one paragraph, and the carriage return is the boundary between paragraphs. A text is partitioned by splitting it by the carriage return into paragraphs. If a text is partitioned into sentences, the period becomes the boundary between them. The paragraphs which are returned by the process of text partition become the classification targets.

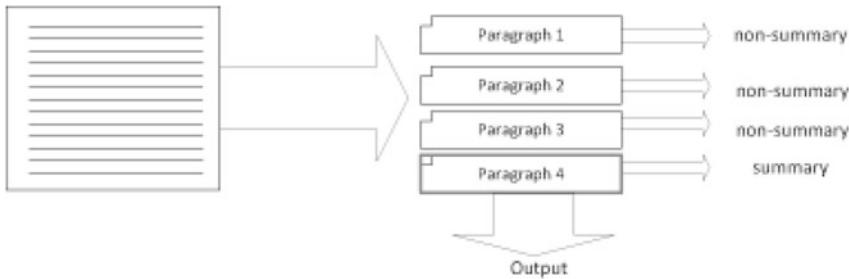
The binary classification of paragraphs into summary or non-summary is presented in Fig. 14.7. A text is partitioned into a list of paragraphs by the above process. Each paragraph is classified into summary or non-summary; the text summarization is interpreted into a binary classification. The paragraphs which are classified into summary are extracted as the summary. We need to encode each paragraph into a numerical vector for applying a machine learning algorithm to the task.



**Fig. 14.6** Partition of text into paragraphs



**Fig. 14.7** Classification of each paragraph into summary or non-summary



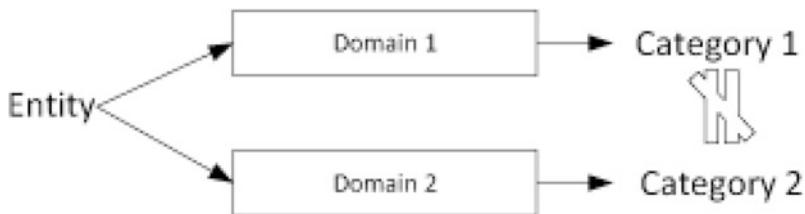
**Fig. 14.8** Selection of paragraph as summary

The view of the text summarization into a binary classification is presented as an example in Fig. 14.8. A text is partitioned into the four paragraphs by the carriage return. Three paragraphs are classified into non-summary, and the last paragraph is classified into summary. The last paragraph is extracted as the summary in this example. If no paragraph is classified into summary, nothing is extracted; this is the demerit of this scheme.

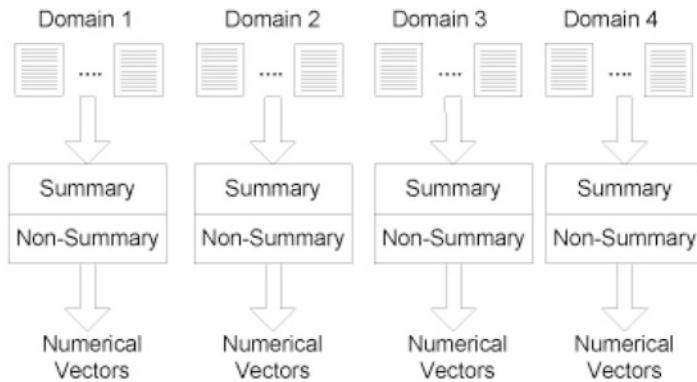
Let us make some remarks on the view of text summarization into a binary classification. A text which is given as the input is partitioned into paragraphs by the carriage return, each paragraph is classified into summary or non-summary, and ones which are classified into summary are extracted. A machine learning algorithm is applied to the binary classification which judges each paragraph into one of two categories. A domain should be considered for classifying each paragraph more correctly.

#### 14.2.4 Machine Learning-Based Abstracting

This section is concerned with the application of a machine learning algorithm to the text summarization. It is interpreted into a binary classification of paragraphs in the previous section. In this section, we apply a machine learning algorithm to the binary classification. The text domain should be considered for classifying a paragraph into summary or non-summary. This section is intended to describe the process of applying a machine learning algorithm to the text summarization.



**Fig. 14.9** Domain-dependent classification

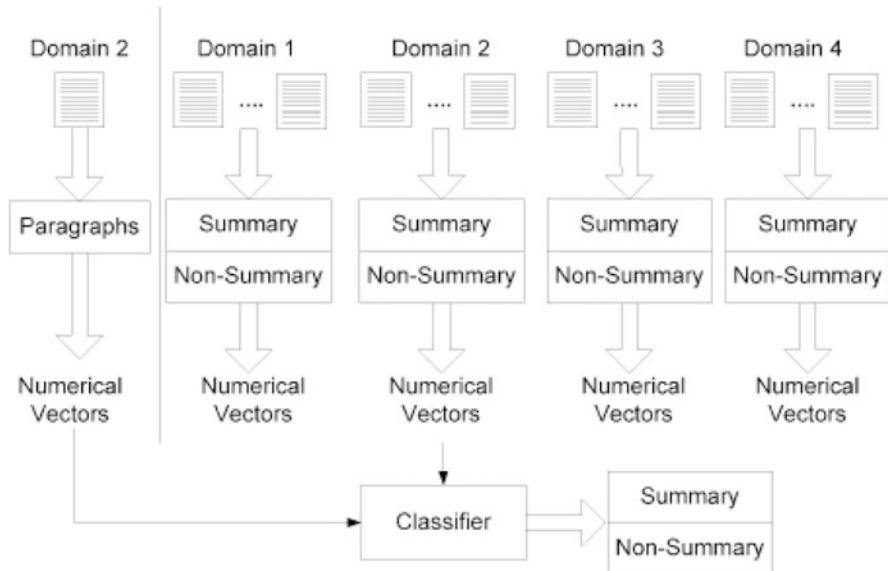


**Fig. 14.10** Sample scheme for text summarization

The motivation of introducing the domain-dependent classification is illustrated in Fig. 14.9. The domain-dependent classification was mentioned in viewing the keyword extraction, the index optimization, the text summarization, and the text segmentation into binary classifications [2–4, 6]. Even same entity may be classified differently, depending on the domain. The domain is needed as a tag for applying a machine learning algorithm to the text summarization. The text summarization may be connected to the text classification, to automate tagging a text with its domain.

The process of summarizing a text by a machine learning algorithm is illustrated in Fig. 14.10. It is assumed that the input text is initially tagged with its own domain, and a classifier which corresponds to the domain is nominated. The input text is partitioned into paragraphs, and each paragraph is classified into summary or non-summary. The paragraphs which are classified into summary are extracted as the final output. We consider connecting the text summarization with the text classification, to automate tagging a text with its own domain.

The process of summarizing a text by a machine learning algorithm is illustrated in Fig. 14.11. It is assumed that the input text is initially tagged with its own domain, and a classifier which corresponds to the domain is nominated. The input text is partitioned into paragraphs, and each paragraph is classified into summary or non-summary. The paragraphs which are classified into summary are extracted



**Fig. 14.11** Paragraph classification in text summarization

as the final output. We consider connecting the text summarization with the text classification, to automate tagging a text with its own domain.

Let us make some remarks on the process of applying a machine learning algorithm to the text summarization. It should be considered that even a same paragraph may be classified differently depending on the domain, in the text summarization which belongs to the domain-dependent classification. As the preparation for training the machine learning algorithm, the paragraphs which are labeled with summary and non-summary are gathered. After training the machine learning algorithm, each paragraph is classified into summary or non-summary, and ones which labeled with summary are extracted. We need to consider that each sentence is classified into summary or non-summary, instead of a paragraph.

## 14.3 Query-Based Text Summarization

This section is concerned with the query-based summarization which is different from the abstracting. In Sect. 14.3.1, we study the query by itself which is mentioned in information retrieval area. In Sect. 14.3.2, we study the text summarization which is based on the similarity between a word and a query. In Sect. 14.3.3, we study the text summarization which is based on the similarity between a query and a sentence. In Sect. 14.3.4, we mention the application of a machine learning algorithm to the query-based text summarization.

word 1; word 2;...word k

**Fig. 14.12** Word-based query

$$(word\ 1 \wedge word\ 2) \vee (word\ 3 \wedge word\ 4)$$

**Fig. 14.13** Query in logic expression

**Fig. 14.14** Representation of query into numerical vector

Word 1	Word 2	...	Word d
0 or 1	0 or 1	...	0 or 1

### 14.3.1 Query

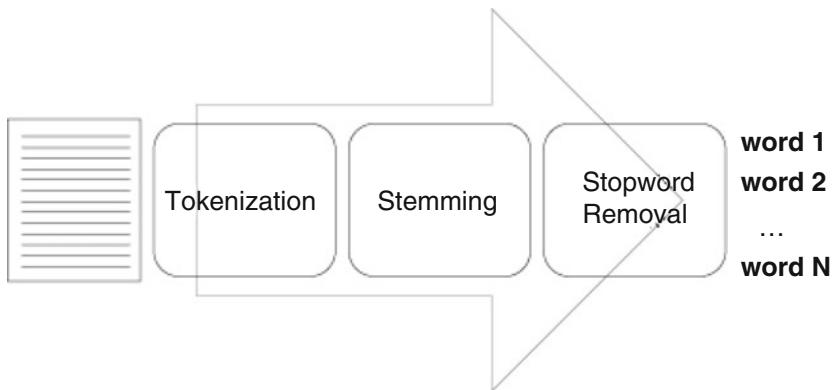
This section is concerned with the query which is given as the input in the information retrieval system. In the previous sections, we studied the schemes of summarizing a text into its abstract without the query. From this section, we consider summarizing a text with the bias toward the query. The process of extracting the text parts which are relevant to the query is called query-based text summarization. This section is intended to describe the query as the preparation for studying the query-based text summarization.

A list of words is illustrated in Fig. 14.12 as a query. Several words are given as the most common query type in the information retrieval system. The words are provided with the semicolon as the boundary between words. The texts which include all of words are retrieved as the highest ranked texts. It is possible to use this query type for summarizing a text.

A Boolean expression of words is illustrated as another query type in Fig. 14.13. The typical three operations,  $\wedge$  (and),  $\vee$  (or), and  $\neg$  (not), are Boolean operations for expressing a Boolean equation. A query is expressed by a Boolean expression where the Boolean operations are applied to words which are the operands. The query which is presented in Fig. 14.13 means the retrieval of texts which include both word1 and word2, or both word3 and word4. This query type is used by advanced users of the information retrieval system.

Representing a query into a numerical vector is illustrated in Fig. 14.14. A query is assumed as a list of words, and the words which are selected from external sources are defined as the features. A query which consists of words is encoded into a binary vector whose element indicates the presence or the absence of each feature in the query. The numerical vector which represents a query is used for computing the relevancy between the query and the text. We need to encode the texts in the collection as well as the query into numerical vectors.

Let us make some remarks on the query which expresses information needs of users. A list of words is the first query type. A logical expression of words which consists of logical operators and operands is the advanced query type. If a query is



**Fig. 14.15** Text indexing process

expressed as a list of words, the list is encoded into a numerical vector for computing its relevancy. We consider automating the query specification by adding associative words.

### 14.3.2 Word-Based Summarization

This section is concerned with the first scheme of summarizing a text biased to the query. The query which is studied in the previous section is the formal expression of the information need which is given as a list of words or a Boolean expression of them. In this scheme, a query is given as a list of words, and the paragraphs which are most relevant to the query are extracted as the summary. This scheme of the query-based summarization is viewed as the process of retrieving relevant paragraphs from a text. This section is intended to describe this scheme of summarizing a text based on the query.

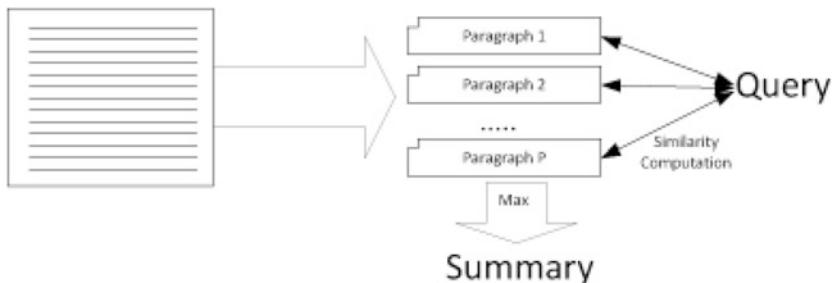
The process of indexing a text into a list of words is illustrated in Fig. 14.15. A text is segmented into tokens by the white space. Each token is transformed into its root form by the grammatical rules. Stop words which are grammatical words which are irrelevant to the contents are removed. The process is described in detail in Sect. 13.2.

The process of computing the similarity between the query and a word is illustrated in Fig. 14.16. The query is assumed as a single word. Both the query and a word are encoded into numerical vectors by the process which is described in Sect. 13.3.1. The similarity between them which represent the query and a word is computed by the process which is described in Sect. 13.3.2. The similarity metric between words is used for extracting the summary from a text.

The process of extracting the summary by the word-based scheme is illustrated in Fig. 14.17. The input text is partitioned into paragraphs, and each paragraph



**Fig. 14.16** Assignment of similarities with query to words



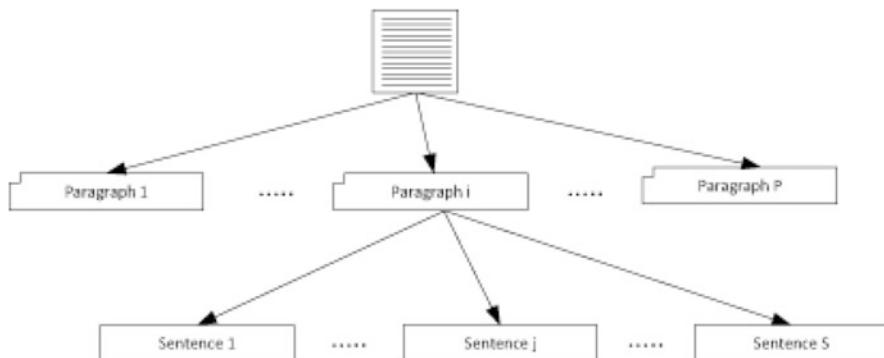
**Fig. 14.17** Word similarity-based summary extraction

is indexed into a list of words. The relevancy of each paragraph to the query is computed based on the similarity metric between words which is mentioned above. The paragraph with its maximal similarity with the query is extracted as the summary. This type of query-based text summarization is viewed as the process of retrieving the most relevant paragraph to the query from a text.

Let us make some remarks on this scheme of summarizing a text based on a query. Each paragraph is indexed into a list of words with the basic three steps: tokenization, stemming, and stop word removal. Its similarity with the query is computed for each paragraph. The paragraph with its maximal similarity is selected as the summary. We may consider word weights as well as the semantic similarity for extracting the summary.

### 14.3.3 Sentence-Based Summarization

This section is concerned with another scheme of summarizing a text based on the query. A text is partitioned into paragraphs, and the most relevant paragraph is retrieved as the summary in the previous scheme. In this scheme, a text is partitioned into sentences, and multiple sentences with their highest relevancy to the query are retrieved as the summary. In the previous scheme, a paragraph is extracted as the summary, whereas in this scheme, several sentences are extracted. This section is intended to describe another scheme of query-based text summarization.



**Fig. 14.18** Hierarchical text partition

Sentence	Relevancy

Query

**Fig. 14.19** Assignment of relevancies with query to sentences

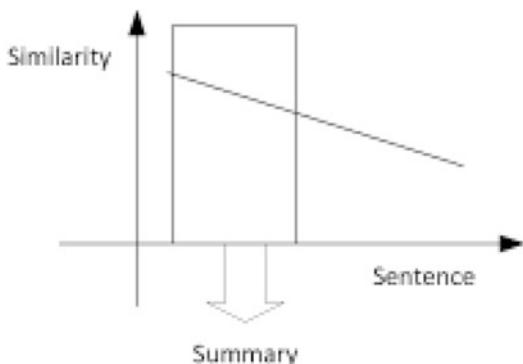
The hierarchical partition of a text is illustrated in Fig. 14.18. A text is partitioned into paragraphs by the carriage return. Each paragraph is partitioned into sentences by the punctuation mark. The colon and the semicolon may become the boundary between sentences, depending on the partition policy. We consider partitioning a compound sentence or a complex sentence into clauses.

The table which consists of sentences and their relevancies to the query is illustrated in Fig. 14.19. A text is partitioned into paragraphs and sentences by the above process. A sentence is indexed into a list of words, and the total relevancy of words is computed for each sentence. The relevancy of a word to the query is the semantic similarity with the query; the total relevancy is divided by the sentence length which is measured by the number of words or characters. We already studied the semantic similarity between words in Sect. 13.3.

The process of selecting sentences with their highest weights as the summary is illustrated in Fig. 14.20. Each sentence is weighted by indexing it and computing its semantic similarity with the query. The sentences are sorted by their weights, and a fixed number of sentences are selected as the summary. We may consider selecting sentences with their higher weights than the threshold as the alternative way. Some sentences which are selected by their relevancies to the query are the summary of the text.

Let us make some remarks on the scheme of summarizing a text based on the query. A text is partitioned into sentences by the punctuation mark. The relevancies

**Fig. 14.20** Selection of relevant sentences



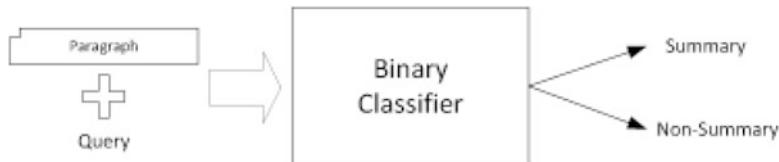
of sentences to the query are computed as their weights. The sentences with their higher weights are selected as the summary. This scheme of the query-based text summarization is viewed as the process of retrieving sentences which are relevant to the query from the text.

#### 14.3.4 ML-Based Text Summarization

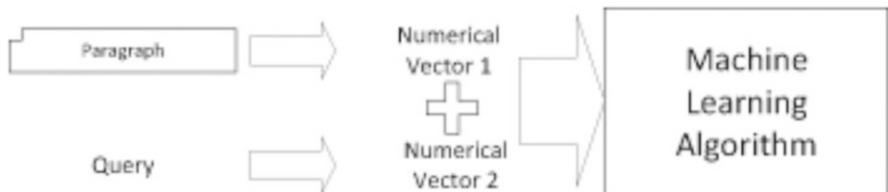
This section is concerned with the application of a machine learning algorithm to the query-based text summarization. The text summarization is viewed as a binary classification, and the application of a machine learning algorithm was mentioned in Sects. 14.2.3 and 14.2.4. In this section, we consider applying a machine learning algorithm to the query-based text summarization with viewing the task into a binary classification. The augmentation of a paragraph with the query is encoded into a numerical vector. This section is intended to describe the process of applying a machine learning algorithm to the query-based text summarization.

Mapping the text summarization into a binary classification is illustrated in Fig. 14.21. In Sect. 14.2.3, we previously mentioned the interpretation of the text summarization into a binary classification for applying a machine learning algorithm. The difference from the case which is covered in Sect. 14.2.3 is to augment a paragraph with the query as an entity which is classified. It is required to encode the query as well as a paragraph into a numerical vector for applying a machine learning algorithm to the query-based text summarization. We consider encoding the query into a vector and augmenting it with one which represents a paragraph.

The application of a machine learning algorithm to the query-based text summarization is illustrated in Fig. 14.22. A text and a query are given as the input. The text is partitioned into paragraphs, and the augmentation of each paragraph with the query is encoded into a numerical vector. The numerical vector which represents



**Fig. 14.21** View of query-based text summarization into binary classification



**Fig. 14.22** Application of machine learning algorithm to query-based text summarization



**Fig. 14.23** Query-based text summarization system

the augmentation is classified into summary or non-summary. We need to sample augmentations which are labeled with summary or non-summary.

The query-based text summarization system is illustrated in Fig. 14.23. The machine learning algorithm is trained with sample augmentations, each of which consists of a paragraph and a query. The text and the query are given as the input, and the text is partitioned into paragraphs. Each augmentation of a paragraph and the query is classified into summary or non-summary, and ones which are classified into summary are extracted. We need to consider the query and the domain at same time for implementing this kind of text summarization system.

Let us make some remarks on the application of a machine learning algorithm to the query-based text summarization. This kind of text summarization is interpreted into a binary classification like the case of abstraction. Both a paragraph and the query are encoded into a numerical vector. The numerical vector which represents a paragraph and the query is classified into summary or non-summary. From this task, we derive the interest-based text summarization which summarizes a text considering user's interests.

## 14.4 Multiple Text Summarization

This section is concerned with the process of summarizing multiple texts. In Sect. 14.4.1, we study the cohesion measure of texts in a group. In Sect. 14.4.2, we study the process of extracting the summary from a text, using its keywords. In Sect. 14.4.3, we apply the machine learning algorithm to the task. In Sect. 14.4.4, we present the process of prototyping a text cluster with its summary, as an instance of multiple text summarization.

### 14.4.1 Group Cohesion

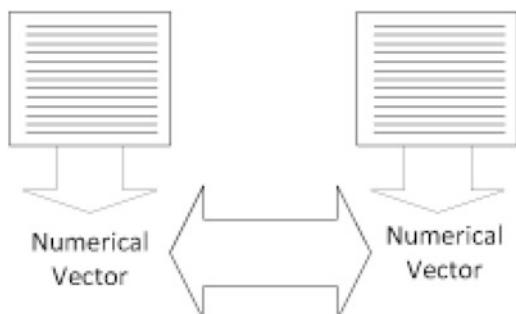
This section is concerned with the cohesion of a text group. In the previous section, we studied the process of summarizing a single text. From this section, the single text summarization is expanded into the multiple text summarization which is the process of summarizing multiple texts. We consider the content-based similarities among texts in a group for summarizing them. This section is intended to describe the cohesion as the similarity metric among texts.

The process of computing the similarity between texts is illustrated in Fig. 14.24. Texts are encoded into numerical vectors, and the process will be studied in the next chapter. The similarity metric, such as cosine similarity and its variant, is computed between two vectors. The similarity metric indicates the similarity between texts. It may be expanded into the similarity of multiple texts.

The intra-cluster similarity is conceptually illustrated in Fig. 14.25. It is the average over similarities of all possible pairs of data items in the given group. If the group is given as the set,  $G = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{|G|}\}$ , the intra-cluster similarity is computed by Eq. (14.1):

$$\text{intra-sim}(G) = \frac{2}{|G|(|G| - 1)} \sum_{i=1}^{|G|} \sum_{j=i+1}^{|G|} \text{sim}(\mathbf{x}_i, \mathbf{x}_j). \quad (14.1)$$

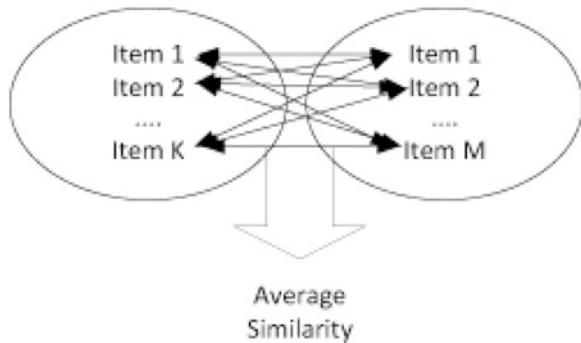
**Fig. 14.24** Similarity between two texts



**Fig. 14.25** Intra-cluster similarity within a cluster



**Fig. 14.26** Inter-cluster similarity between two clusters



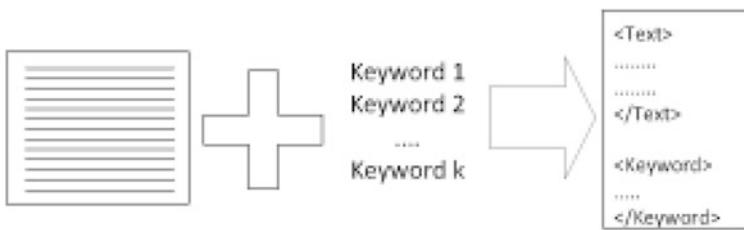
If the similarity between individual items is a normalized value, the intra-cluster similarity is given as a normalized value. The intra-cluster similarity indicates the group cohesion.

The concept of the inter-cluster similarity is illustrated in Fig. 14.26. The inter-cluster similarity is the average over similarities of items that belong to different groups. The two groups are notated by sets,  $G_1 = \{\mathbf{x}_{11}, \mathbf{x}_{12}, \dots, \mathbf{x}_{1|G_1|}\}$  and  $G_2 = \{\mathbf{x}_{21}, \mathbf{x}_{22}, \dots, \mathbf{x}_{2|G_2|}\}$ . The inter-cluster similarity is computed by Eq. (14.2):

$$\text{inter-sim}(G_1, G_2) = \frac{1}{|G_1| \times |G_2|} \sum_{i=1}^{|G_1|} \sum_{j=1}^{|G_2|} \text{sim}(\mathbf{x}_i, \mathbf{x}_j). \quad (14.2)$$

The inter-cluster similarity indicates the similarity between two groups; higher inter-cluster similarity means poor discrimination between two clusters.

Let us make some remarks on the text group cohesion. The similarity between texts is computed by encoding them into numerical vectors and applying the cosine similarity to them. The intra-cluster similarity is the average over the similarities among items within a group. The inter-cluster similarity is the average over similarities of all possible pairs of items from two different groups. The two measures are considered for performing the multiple text summarization.



**Fig. 14.27** Association of text with its keywords



**Fig. 14.28** Keyword extraction process

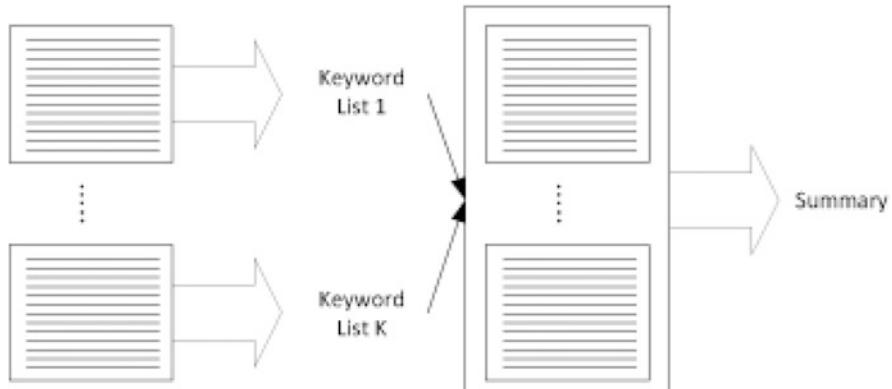
#### 14.4.2 *Keyword-Based Summarization*

This section is concerned with the first scheme of summarizing multiple texts. We studied the characterization of a text group with the two measures: intra-cluster similarity and inter-cluster similarity. In this section, we start to study the scheme of summarizing multiple texts based on keywords. We consider both cases: one is that each text is initially associated with its own keywords, and the other is that keywords are automatically extracted from a text. This section is intended to describe the scheme of summarizing multiple texts based on their keywords.

The text which is initially associated with its keywords is illustrated in Fig. 14.27. It is assumed that texts are associated with their own keywords in adopting this scheme of summarizing multiple texts; keywords are initially provided by authors. A typical text format which is associated with keywords is the XML document which consists of text and keywords as its tags. The XML document is separated into a text and keywords in implementing the scheme of summarizing multiple texts. It is tedious to assign keywords to each text manually.

The process of extracting keywords from a text is illustrated in Fig. 14.28. Because it is very tedious to prepare texts which are initially associated with keywords, we need the automatic process of extracting keywords from a text. The process of doing so is to index a text into a list of words and select some among them as keywords. In [4], the keyword extraction is viewed as the binary classification which each word is classified into keyword or non-keyword. The keywords are extracted from a text, and it is associated with them as an XML document.

The process of summarizing multiple texts based on their keywords is illustrated in Fig. 14.29. It is assumed that each text in the group is associated with its own keywords; each text is XML document which consists of its article and its keywords.



**Fig. 14.29** Keyword-based multiple text summarization

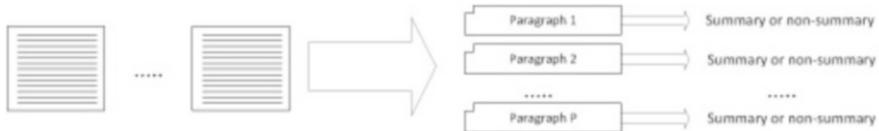
The keywords are extracted from each text, and multiple lists of keywords are integrated into a single list by the union of them. The paragraphs which are most relevant to the keywords are retrieved as the summary of multiple texts. If a text is initially not associated with their keywords, its keywords are extracted by the above process, automatically.

Let us make some remarks on the keyword-based summarization of multiple texts. They are assumed as XML documents, each of which consists of its article and its keywords. If it is very tedious to associate manually each text with its keyword, we need the technique of extracting automatically keywords from it. The process of summarizing multiple texts is to select paragraphs which include most keywords as the summary. We need to discriminate keywords with their weights for extracting summary more reliably.

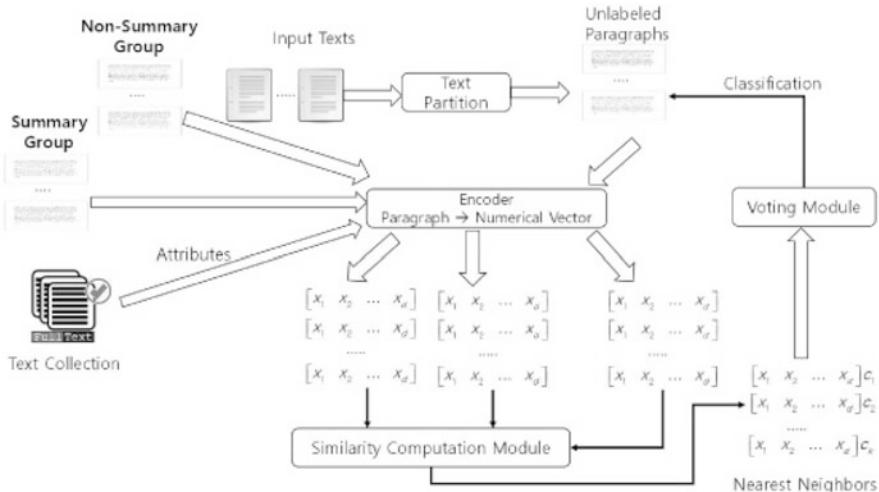
#### 14.4.3 Machine Learning-Based Text Summarization

This section is concerned with the application of a machine learning algorithm to the multiple text summarization. In the previous sections, we studied the heuristic schemes of summarizing multiple texts. In this section, we apply a machine learning algorithm to the multiple text summarization with the interpretation of it into the classification task. In applying a machine learning algorithm, the input is given as multiple texts, they are partitioned into paragraphs, and each paragraph is classified into summary or non-summary. This section is intended to describe the process of applying a machine learning algorithm to the multiple text summarization.

Let us mention the process of gathering sample paragraphs which is called sampling. The multiple text summarization as well as the single text summarization is interpreted as a binary classification. The text collection with a domain is partitioned into paragraphs, and individual paragraphs are labeled with summary



**Fig. 14.30** Mapping multiple text summarization into binary classification

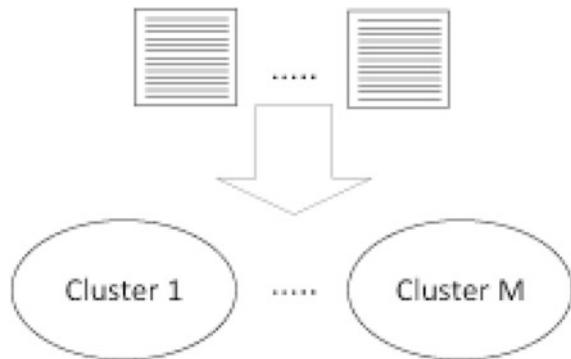


**Fig. 14.31** Architecture of multiple text summarization system

or non-summary manually. The features are defined as words, and the sample paragraphs are encoded into numerical vectors. They are used for training the machine learning algorithm as the approach to the text summarization.

Mapping the multiple text summarization into a binary classification is illustrated in Fig. 14.30. Multiple texts are given as the initial input. Paragraphs are generated by partitioning the multiple texts into paragraphs, and each of them is classified into summary or non-summary. The paragraphs which are classified into summary are extracted from the system like the case of the single text summarization. The process of mapping the multiple text summarization into a binary classification is same to the case of the single text summarization, except the fact that multiple texts are given.

The multiple text summarization system is illustrated in Fig. 14.31. The features are extracted from the text collection, and sample paragraphs which are labeled with summary or non-summary are gathered. Multiple texts which are given as the input are transformed into paragraphs, and each of them is encoded into a numerical vector. Each paragraph is classified into summary or non-summary, and ones which are classified into summary are extracted as the output. In the multiple text summarization system, the KNN was adopted, and the system was proposed in [3].

**Fig. 14.32** Text clustering

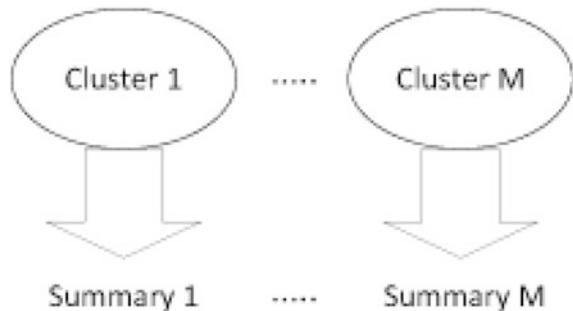
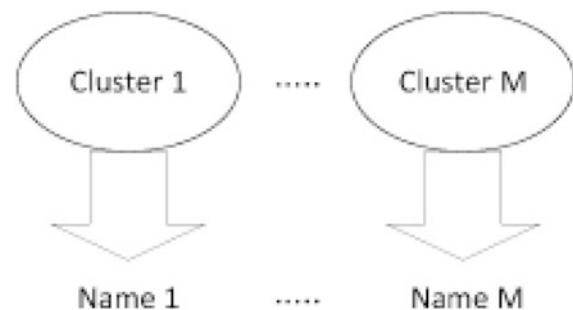
Let us make some remarks on the application of a machine learning algorithm to the multiple text summarization. It is viewed as a binary classification, and sample paragraphs which are labeled with summary or non-summary are gathered in each domain. The binary classification which is mapped from the multiple text summarization is to classify each paragraph into summary or non-summary. The multiple texts are partitioned into a list of paragraphs, and the paragraphs which are classified into summary are extracted. We consider the cohesion of the multiple texts in summarizing them.

#### ***14.4.4 Textual Cluster Prototype***

This section is concerned with the application of the multiple text summarization to maintain the text clusters. In the previous section, we mentioned the schemes of summarizing multiple texts. In this section, we apply the multiple text summarization for generating text cluster scripts. We consider cluster naming which assigns a title to each text cluster as well as its script. This section is intended to describe the text clustering, cluster summarization, and cluster naming.

The text clustering is illustrated in Fig. 14.32 with its functional view. It is the process of segmenting a group of texts into subgroups by their semantic similarities. It is required to define a similarity metric between texts for implementing the task. Unsupervised learning algorithms, such as the k-means algorithm, the AHC algorithm, and the Kohonen Networks, are used as the approaches. We consider making the results from clustering texts into ones where the browsing is possible.

The application of multiple text summarization to the summarization of text clusters is illustrated in Fig. 14.33. Texts are clustered, depending on their content-based similarities by the above process. Texts in each cluster are summarized by adopting a scheme among what is described in the previous sections. The summary of each cluster is the script which briefly explains its contents. The script which is given as the summary provides the guide for browsing text clusters.

**Fig. 14.33** Cluster naming**Fig. 14.34** Cluster script

The process of naming symbolically each text cluster is illustrated in Fig. 14.34. We mentioned the process of scripting each text cluster using the multiple text summarization, above. The cluster naming is the process of assigning less than three words to each cluster as its title. The cluster scripting is for providing a brief explanation about a text cluster, whereas the cluster naming is for guiding the access to a text by the browsing. Some principles of naming text clusters are defined in [1].

Let us make some remarks on the application of multiple text summarization to the maintenance of text clusters. The text clustering is the process of segmenting a text group into subgroups based on the content-based similarities among texts. The cluster summarization is the process of summarizing texts in a cluster into its script. The cluster naming is the process of assigning a title symbolically to each cluster with less than three words. We consider using both the text clustering and the cluster maintenance for generating sample texts automatically for the text classification.

## 14.5 Summary and Further Discussions

This section is concerned with the summary and the further discussions on what is studied in this chapter. The process of extracting the summary from a text in the generic view is called abstracting. The summary which is biased to an information need is retrieved with a query which expresses it. Multiple texts rather than a single text may be summarized for extracting a script automatically from a text group. This section is intended to discuss further on what is studied in this chapter.

The text partition is mentioned as the process of partitioning a text into its parts. The text indexing is an instance of text partition because words are parts of a text. When a text is partitioned into sentences, periods or question marks are set as the boundaries. When it is partitioned into paragraphs, the carriage return is set as the boundary. When a text partition into sentences or paragraphs, it is assumed that the author writes it correctly.

Let us mention the information retrieval as the area which is related with the text summarization. The information retrieval is defined as the area for retrieving relevant textual data, satisfying the information needs of users. The process of information retrieval is to give some words as the query which expresses the information need, compute the relevancy of the query with texts in the collection, and retrieve highly relevant texts as the results. The text summary is used for computing its relevancy to the query more efficiently. The text summarization is mentioned frequently in the area of information retrieval.

Let us mention the process of scripting text clusters using the text summarization. A text cluster script is a brief explanation about a text cluster contents. From texts in a cluster, the summary is extracted as a script about them. The script is used for accessing a particular text by browsing. We need to modify the techniques of multiple text summarization which were covered in Sect. 14.4, for specializing them for scripting text clusters.

The text segmentation is mentioned as another text mining task which is different from the text summarization. It is the process of partitioning a text into subtexts based on its contents. Its process is to generate paragraph pairs by sliding a two sized window on paragraphs and put boundary between paragraphs in each pair which contents change. The text segmentation is viewed as a binary classification which classified a paragraph pair into continuance and boundary [5]. The subtexts as results from the text segmentation are treated as independent texts.

## References

1. T. Jo, “The Implementation of Dynamic Document Organization using Text Categorization and Text Clustering”, PhD Dissertation of University of Ottawa, 2006.
2. T. Jo, “Graph based KNN for Optimizing Index of News Articles”, 53–62, Journal of Multimedia Information System, 3, 2016.
3. T. Jo, “Automatic Text Summarization using String Vector based K Nearest Neighbor”, 6005–6016, Journal of Intelligent and Fuzzy Systems, 35, 2018.
4. T. Jo, “Keyword Extraction in News Articles using Table based K Nearest Neighbors”, The Proceedings of 25th International Conference on Computational Science & Computational Intelligence, 2018.
5. T. Jo, “Content based Segmentation of News Articles using Feature Similarity based K Nearest Neighbor”, 61–64, The Proceedings of 19st International Conference on Information and Knowledge Engineering, 2019.
6. T. Jo, “Content based Segmentation of News Articles using Feature Similarity based K Nearest Neighbor”, 61–64, The Proceedings of 19st International Conference on Information and Knowledge Engineering, 2019.

# Chapter 15

## Textual Deep Operations



This chapter is concerned with the textual deep operations as mapping a text into another text. We studied the numerical deep operations as mapping an input vector into another vector which is called a hidden vector. The textual deep operations are manipulations on a text which map it into another text called hidden text before encoding it into a numerical vector. The role of the textual deep operations is to preprocess a text in implementing the textual deep learning. This chapter is intended to describe the textual pooling and the textual convolution as the main textual deep operations.

This chapter is composed with the five sections, and we overview what is studied in this chapter in Sect. 15.1. In Sect. 15.2, we review the operations on numerical vectors for implementing the deep learning. In Sect. 15.3, we study the textual convolutions as deep operations on textual data. In Sect. 15.4, we study the textual pooling operations. In Sect. 15.5, we discuss further what is studied in this chapter.

### 15.1 Introduction

This section is concerned with the overview of textual deep operations. The pooling and the convolution were studied as the typical deep operations on vectors or matrices. In this chapter, we study the deep operations which are applicable to textual data instead of vectors or matrices. The deep operations are used for implementing the deep learning algorithms which are specialized for processing texts. This section is intended to overview the textual deep operations as the introduction to this chapter.

Let us review the deep operations which are applied to numerical vectors. The pooling is the process of selecting a representative value for each window which slides on the vector. The convolution is the process of filtering values by the product of elements in the input vector and ones in the filter vector. In the pooling operation, a single vector or a single matrix is generated as the output in the pooling

operation, whereas in the convolution, multiple vectors or multiple matrices are generated as the output, if using multiple filter vectors or matrices. The numerical deep operations, the convolution, and the pooling, are the basis for deriving the textual deep operations.

Let us mention the textual deep operations which are applicable directly to the textual data. The input data is assumed as a text, instead of a vector or a matrix. Another text is generated as the results from applying the text deep operation to the text. The text summarization which was covered in Chap. 14 is a typical instance of textual deep operation, in that a text part is generated as another text. The textual deep operations are used for implementing the deep learning algorithms which are specialized for the text classification.

The textual deep learning algorithms are implemented using the textual deep operation. It is the process of generating another text or texts from the input. The textual deep operations are added to a machine learning algorithm before the module of encoding a text into a numerical vector. The deep operations on numerical vectors may be added between the encoding module and the classification module for implementing the deep learning algorithm, further. The textual deep learning is viewed as the combination of textual deep operation with the machine learning.

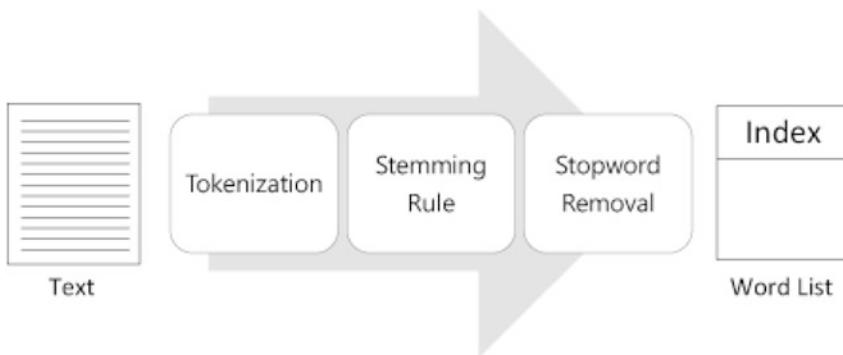
Let us mention what is intended in this chapter. We review the numerical deep operations which are used for modifying the existing machine learning algorithms into deep versions. We study the textual convolution operation which is necessary for implementing the textual deep learning algorithm. We study the textual pooling operation as another deep operation. This chapter is intended to study the textual deep operations which are used for implementing a textual deep learning algorithm as an approach to the text classification.

## 15.2 Numerical Deep Operations

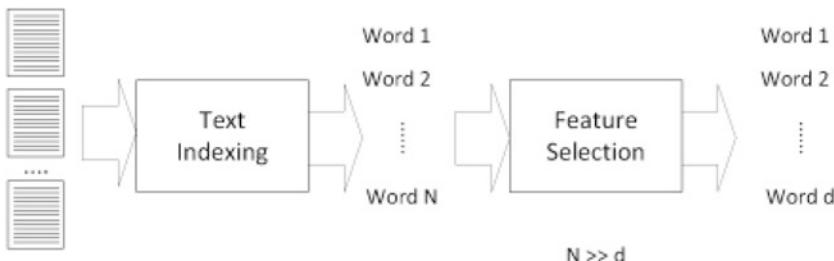
This section is concerned with the convolution and the pooling which are applied to numerical vectors. In Sect. 15.2.1, we study the process of encoding a text into a numerical vector. In Sect. 15.2.2, we review the convolution which was covered in Chap. 12. In Sect. 15.2.3, we review the pooling, which was also covered in Chap. 12, as another operation. In Sect. 15.2.4, we consider generating virtual examples from the training examples.

### 15.2.1 Text Encoding

This section is concerned with encoding a text into a numerical vector. The numerical deep operations, such as pooling and convolution, are applicable after doing it. Feature candidates, given as words, are extracted by indexing a text collection, and features are defined as the attributes, by selecting some words among



**Fig. 15.1** Indexing process



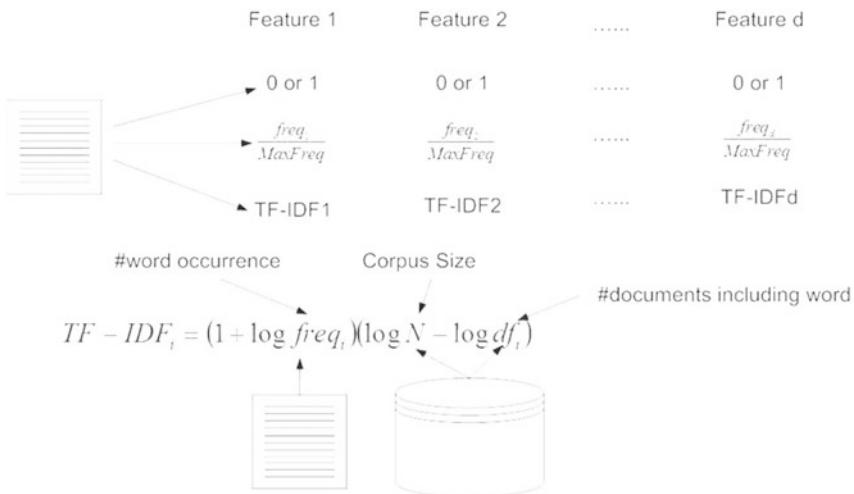
**Fig. 15.2** Feature selection

feature candidates. The feature frequency in the text or the feature weight, which is computed by an equation, becomes a feature value in encoding so. This section is intended to describe the process of encoding a text into a numerical vector.

The entire process of indexing a text is illustrated in Fig. 15.1. A text is segmented into words by the white space in the tokenization as the first text indexing step. Each word is transformed into its root form in the stemming as the second step. The stop words are removed in the third step. The text indexing is accomplished with the three basic steps.

The process of selecting some words as the features is illustrated in Fig. 15.2. The entire text collection is indexed into a list of words by the above process. The words which are indexed from the text collection are given as feature candidates, and some are selected among them as the features. The criteria for selecting the features are the number of texts which include each of them and the weight of each word in the collection. The number of words which are selected as the features becomes the dimensionality of the vector which represents a text.

The process of assigning values to features in encoding a text into a numerical vector is illustrated in Fig. 15.3. The feature value is a binary value; zero indicates the absences of the word in the text, and one indicates the presence. The frequency of the word in the text is used as a feature value; the vector which represents a text



**Fig. 15.3** Feature value assignment

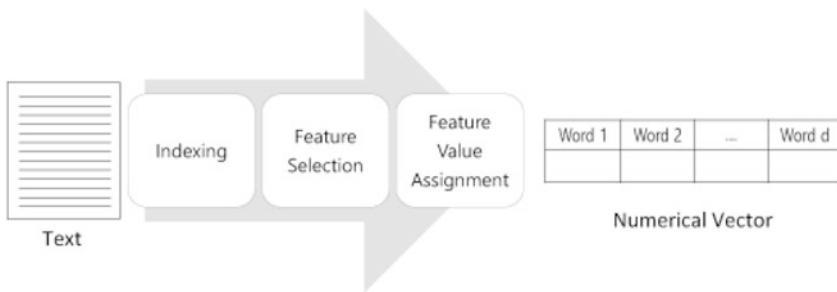
consists of zero values and positive integers. The word weight is computed by the equation, which is presented in Fig. 15.3, and the feature is given as a real value. This step accomplishes encoding a text into a numerical vector.

Let us make some remarks on the process of encoding a text into a numerical vector. The texts in the corpus are indexed into a list of words as the feature candidates. Some are selected as the features among the words; the number of the selected features is the dimension of the numerical vector which represents a text. A binary value, a frequency, or a weight is given as a feature value. We consider encoding a text into other types of structured forms as alternatives to the numerical vector.

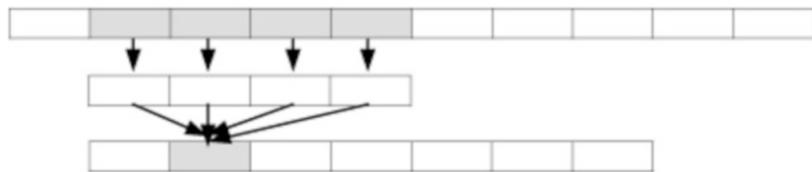
### 15.2.2 Convolution

This section is concerned with the convolution as the deep operation on numerical vectors. We studied the operation in detail in Chap. 12. In this section, we review this deep operation for providing the background for understanding the textual convolution. A filter vector or matrix which is defined arbitrary is involved in performing the operation. This section is intended to describe the convolution on numerical vectors, briefly.

The process of representing a text into a numerical vector is illustrated in Fig. 15.4. In the previous section, we studied the process in detail, but this process is mentioned for reviewing it briefly. A text is encoded into a numerical vector with the three steps: text indexing, feature selection, and feature value assignment. The deep operation such as the convolution is applied to the numerical vector which represents



**Fig. 15.4** Text-encoding process



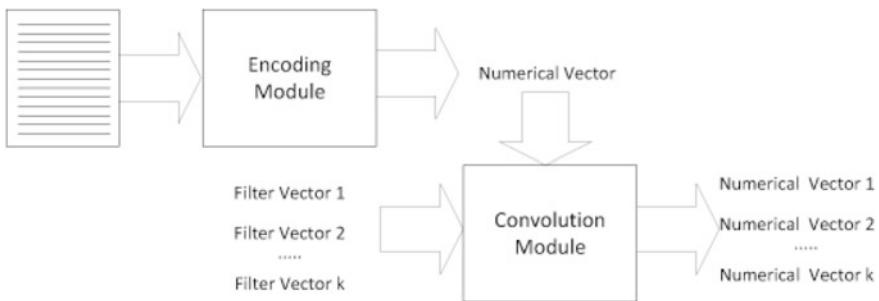
**Fig. 15.5** Convolution process

a text. The textual deep operation which is covered in Sect. 15.3 is applicable to a raw text before encoding it.

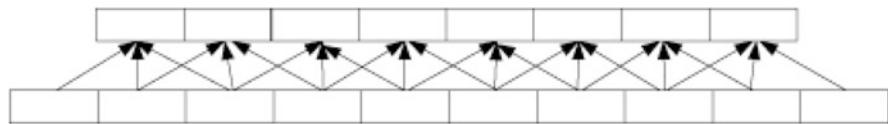
The frame of applying the convolution to a vector is illustrated in Fig. 15.5. The window size is set as four, and the four elements of the input vector are involved in the convolution. The four-dimensional filter vector is prepared, and each element in the output vector is computed by the inner product of the window which slides on the input vector and the filter vector. The dimension of input vector is ten, and the dimension of output vector becomes  $7 = 10 - 4 + 1$ , in this example. The convolution is also applicable to a matrix as well as a vector.

The convolution module for implementing the deep learning algorithm is illustrated in Fig. 15.6. A text is encoded into a numerical vector by the encoding module. Multiple filter vectors,  $k$  filter vectors, are defined in this operation, and the  $k$  output vectors are computed from a single input vector. The  $k$  numerical vectors are transferred as the input vector to the next layer or the classification part. If multiple input vectors are classified, the voting may be applied.

Let us make some remarks on the numerical convolution which is applied for processing a text. It is required to encode a text into a numerical vector by the process which is described in the previous section for applying the numerical convolution. The dimension of a vector is decreased by the size of a filter vector minus one, through the convolution operation. If multiple filter vectors are involved, multiple output vectors are generated by the convolution. If multiple filter vectors are used, multiple output vectors are viewed as a matrix.



**Fig. 15.6** Convolution module



**Fig. 15.7** One-step pooling

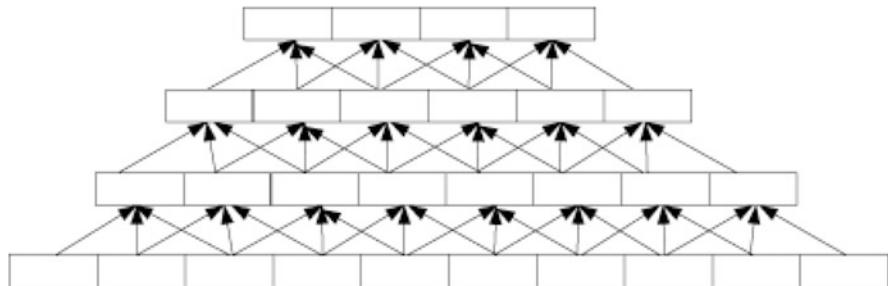
### 15.2.3 Pooling

This section is concerned with the pooling as another deep operation. In the previous section, we studied the convolution as a main deep operation on numerical vectors. In this section, we study the pooling on a numerical vector with the assumption of encoding a text into a numerical vector. The pooling is used together with the convolution for implementing deep learning algorithms as the approaches to the text classification. This section is intended to describe the pooling as a deep operation.

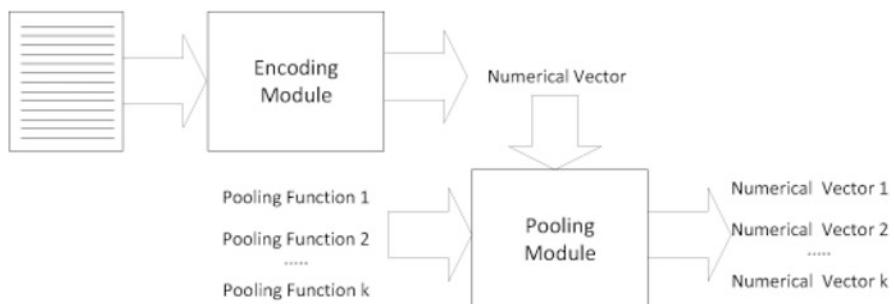
The single step pooling is illustrated in Fig. 15.7. The number of input values is ten, and the size of window which slides on the input values is three. Each output value is the representative which is selected from the three sized window. The number of output values becomes  $8 = 10 - 3 + 1$ . The ten-dimensional vector is reduced into the eight-dimensional vector by the pooling operation on the three sized window.

The multiple step pooling is illustrated in Fig. 15.8. The ten-dimensional vector is downsized into the seven-dimensional vector by the pooling operation which is presented in Fig. 15.7. The ten-dimensional vector is downsized into the four-dimensional vector by applying the pooling operation three times with the three sized window. The ten dimensions are reduced to the eight dimensions by  $8 = 10 - 3 + 1$ ; the eight dimensions are reduced to the six dimensions, by  $6 = 8 - 3 + 1$ ; and the six dimensions are reduced to the four dimensions by  $4 = 6 - 3 + 1$ . The dimensionality is cut down as the effect of the accumulative pooling operation.

The architecture of the pooling module for processing a text is illustrated in Fig. 15.9. A text is represented into a numerical vector by the encoding module. Multiple pooling operations are defined, and an output vector is generated from each



**Fig. 15.8** Multiple-step pooling



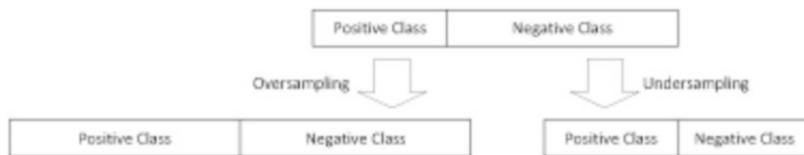
**Fig. 15.9** Pooling module

pooling operation. If the  $k$  pooling functions are defined, the  $k$  output vectors are generated from this module. The dimension of the vector is reduced by the pooling operation.

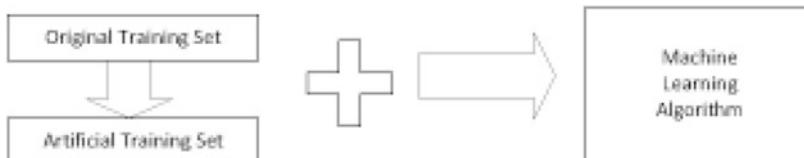
Let us make some remarks on the pooling operation on a numerical vector. If the window size is  $k$  and the pooling operation is performed one time, the dimension is reduced by  $k - 1$ . If the pooling operation is formed  $m$  times, the dimension is reduced by  $m(k - 1)$ . The pooling module is installed after the encoding module where a text is represented into a numerical vector. We consider the pooling operation on a text and the encoding module which follows the pooling operation.

#### 15.2.4 Virtual Examples

This section is concerned with the manipulation on training examples. In the previous sections, we reviewed the convolution and the pooling as the manipulations on the training examples. In this section, we study the process of generating additional training examples which are called virtual examples as another manipulation on the training examples. If the convolution is applied with multiple filter vectors or matrices, multiple vectors or matrices are generated from a vector or a matrix. This



**Fig. 15.10** Simple resampling schemes: oversampling and undersampling



**Fig. 15.11** Virtual example generation

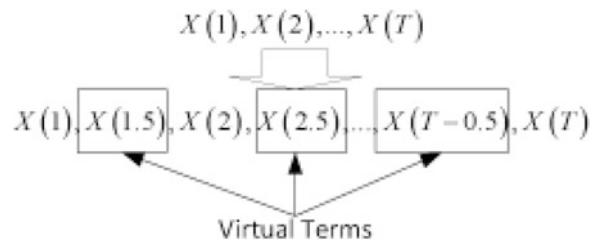
section is intended to describe the manipulations other than the convolution and the pooling.

The application of the resampling to the training examples is illustrated in Fig. 15.10. The resampling is the process of balancing the category portions in the training examples [3]. The predefined categories are assumed as the positive class and the negative class, and the negative class is dominant over the positive class as shown in Fig. 15.10. The oversampling is the process of increasing the size of weak categories by adding more training examples, and the undersampling is the process of decreasing the size of dominant categories by removing some training examples. The resampling is applied to the case of unbalanced category portions in the training example.

The frame of generating virtual examples for training a machine learning algorithm is illustrated in Fig. 15.11. A set of training examples is prepared, and this kind of training examples is called actual examples. By manipulating the actual examples, additional training examples which are called virtual examples are generated. Both the actual examples and the virtual examples are used for training a machine learning algorithm. The scheme of generating virtual examples was proposed in 1996 by Cho et al. [1].

The generation of midterms in the time series prediction is illustrated in Fig. 15.12. It was initially proposed that the neural networks should be applied to the time series prediction with generating the midterms by Jo in 2010 [4], and this scheme was applied to multivariate time series predictions in 2013 [5]. The midterms are estimated by defining interpolation equations, and the training examples are generated by sliding a window on the time series which includes the midterms. The number of training examples is increased almost two times, and the generalization performance is expected to be improved by the increased number of training examples. The effect of this scheme was successful in applying the neural networks to the time series prediction in various domains.

**Fig. 15.12** Virtual term generation in time series



Let us make some remarks on the manipulation of the training examples for improving the generalization performance. The resampling is the process of balancing the category portions by removing some training examples or adding more training examples. The training examples which are artificially generated by manipulating the training examples are called virtual examples. It was proposed that midterms should be estimated in the time series prediction for improving the performance of the neural networks which are applied to the time series prediction. The convolution is intended for generating more training examples as well as downsizing of the dimension, as a deep learning operation.

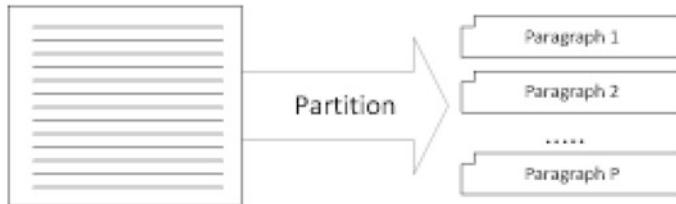
## 15.3 Textual Convolution

This section is concerned with the textual convolution as a specialized convolution for textual data. In Sect. 15.3.1, we study the structure of raw text. In Sect. 15.3.2, we study the selection of part at random from text as an instance of textual convolution. In Sect. 15.3.3, we mention the process of indexing a text into a hierarchical structure of words. In Sect. 15.3.4, we study the temporal topic analysis which tags topics to each paragraph.

### 15.3.1 Raw Text Structure

This section is concerned with the structural view of a raw text. In the previous section, we studied the deep operations, pooling, and convolution, which are applied to a numerical vector. From this section, we consider the deep operations which are applicable to textual data directly, called textual deep operations. A text is viewed as a list of paragraphs or words, simply; it is viewed as a hierarchical structure. This section is intended to describe a raw text with its hierarchical view.

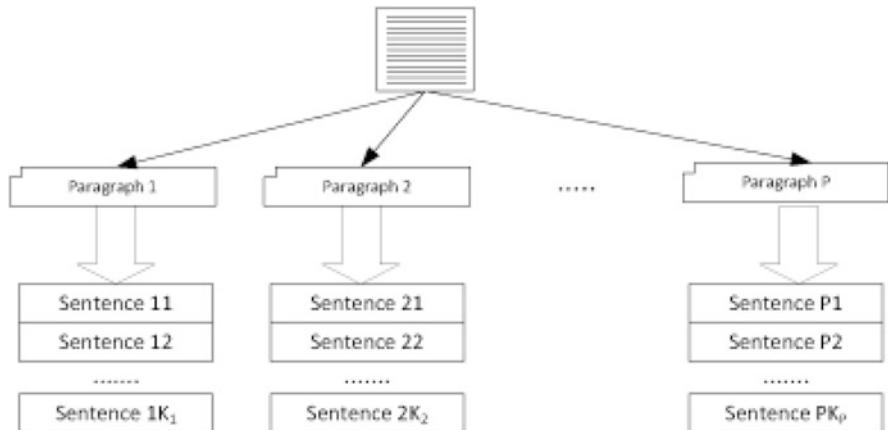
The partition of a paragraph is sentences is illustrated in Fig. 15.13. A text is partitioned into paragraphs as shown in Figure Prev. A paragraph is segmented by the period into sentences. A paragraph is expressed as a set of sentences. A sentence is viewed as a combination of words based on the grammatical rules.



**Fig. 15.13** Text: paragraph list



**Fig. 15.14** Paragraph: sentence list

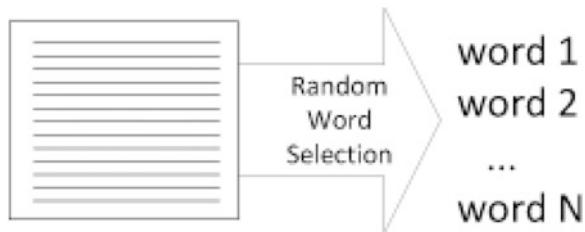


**Fig. 15.15** Hierarchical text structure

The partition of a paragraph into sentences is illustrated in Fig. 15.14. A text is partitioned into paragraphs as shown in Fig. 15.13. A paragraph is segmented by the period into sentences. A paragraph is expressed as a set of sentences. A sentence is viewed as a combination of words based on the grammatical rules.

The hierarchical structure of the raw text is illustrated in Fig. 15.15. The raw text is partitioned into paragraphs by the carriage return; the root node is given as the entire text and the nodes in the next level are given as the paragraphs. Each paragraph is partitioned into sentences by the period; the child nodes are given as the sentences. Each sentence is partitioned into words by the white space, further. Compound words should be considered in segmenting a sentence into words.

**Fig. 15.16** Random word selection



Let us make some remarks on the structure of raw text. Because a text is partitioned into paragraph, it is viewed as a set of paragraphs. Each paragraph is partitioned into sentences further; a paragraph is viewed as a set of sentences. The hierarchical structure which represents a text is constructed. We may consider representing a raw text into a graph where each node is a word, and each edge is a contextual relation between words.

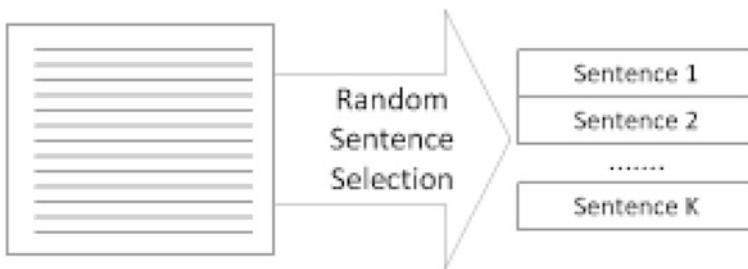
### 15.3.2 Random Part Selection

This section is concerned with a text convolution scheme, called random part selection. In the previous section, a raw text is viewed as a hierarchical structure which consists of paragraphs and sentences. In this operation, a text is indexed into words or is portioned into paragraph, and some words or some sentences are selected at random. A word list or a sentence list is encoded into a numerical vector as the results from applying the operation to the text. This section is intended to describe this text convolution instance.

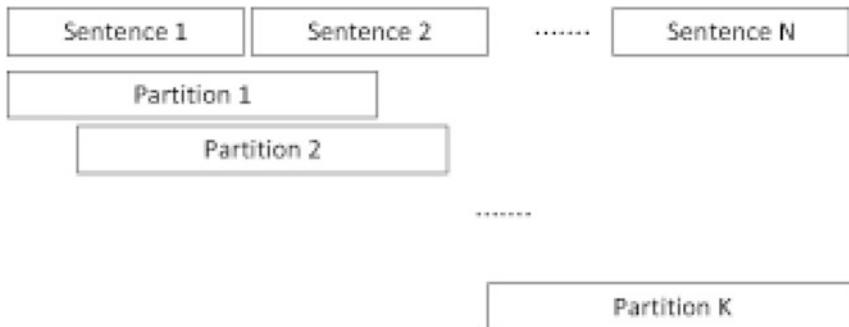
The random selection of words from a text is illustrated in Fig. 15.16. A text is indexed into a list of words by the process which is described in Sect. 13.1. Some are selected at random among them. The selected words are listed as results from applying the textual convolution to the text. The results for every trial of applying the textual convolution are different from each other.

The process of selecting sentences at random from a text is illustrated in Fig. 15.17. A text is segmented into sentences by the period which indicates the sentence boundary. Some sentences are selected at random among them in this type of convolution. The text which consists of the selected sentences is one which is mapped from a raw text. The mapped text is encoded into a numerical vector for applying a machine learning algorithm to a text mining task.

The process of generating the partitions by sliding a window on the text is illustrated in Fig. 15.18. A text is given as the input, and the window size is determined as the number of words. The partitions over sentences are generated by sliding the window. One which is mapped from the input text is a list of partitions. The partition list is encoded into a numerical vector.



**Fig. 15.17** Random sentence selection



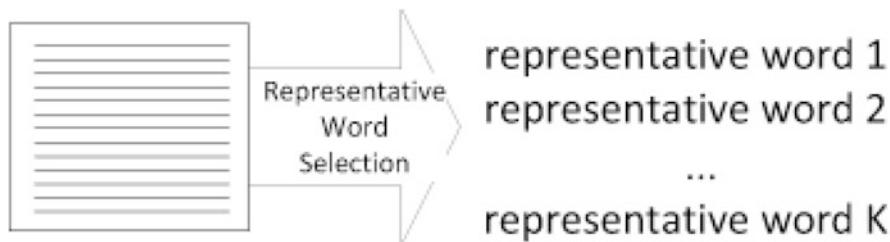
**Fig. 15.18** Window sliding-based selection

Let us make some remarks on this instance of textual convolution. A text is indexed into a list of words, and some are selected at random among them. A text is segmented into sentences, and some are selected at random. The partitions are generated by sliding a fixed-sized window on the text. We need to consider the filtering frame which slides on the text for selecting the text partitions.

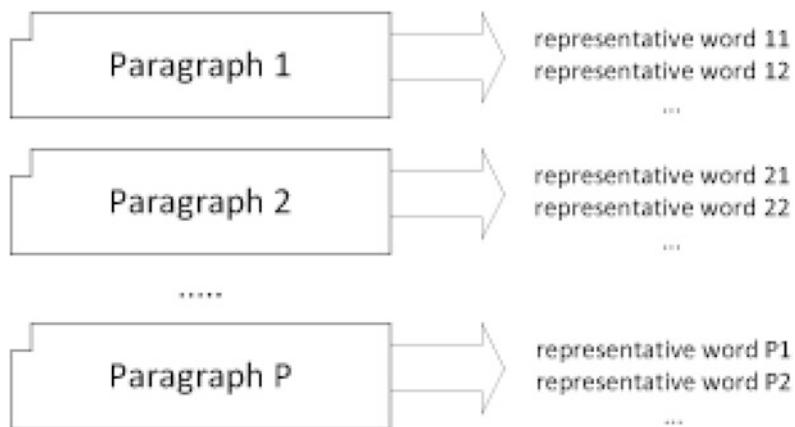
### 15.3.3 Hierarchical Indexing

This section is concerned with the process of indexing a text hierarchically. The text indexing which is studied in Chap. 13 is the process of mapping a text into a flat list of words. It is expanded into the hierarchical text indexing which maps a text into a hierarchical structure of words. The tree structure where each node is a group of representative words is the results from indexing a text hierarchically. This section is intended to describe the process of mapping a text into a tree structure.

Index of the entire text as the root is illustrated in Fig. 15.19. It consists of more than one paragraph and is given as the input. It is indexed into a list of words with the three steps: tokenization, stemming, and stop word removal. The words which are



**Fig. 15.19** Representative word selection in entire text



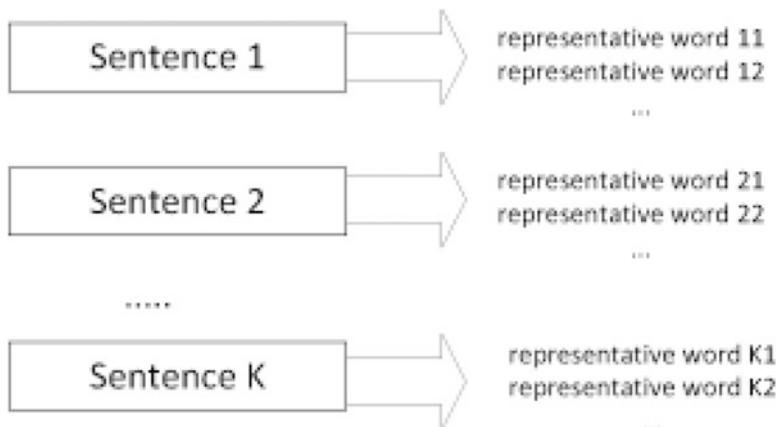
**Fig. 15.20** Representative word selection in each paragraph

extracted by the process are weights, and ones with their lower weights are removed further. A list of selected words is given as the root node in the hierarchical indexing.

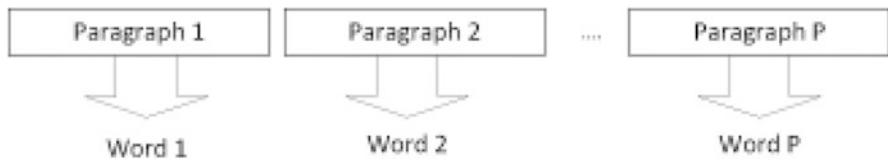
Indexing paragraphs as the nodes in the second level is illustrated in Fig. 15.20. The entire text is partitioned into a list of paragraphs. Each paragraph is indexed into a list of words with the basic three steps. All words should be taken in a short paragraph consists of only two sentences, but further filtering is needed in a long paragraph. Each node in the second level is composed with several words.

Indexes of the sentences as the nodes in the third level are illustrated in Fig. 15.21. Each paragraph is partitioned into sentences by the punctuation mark. Each sentence is indexed into a list of words with the basic three steps. Because a sentence is short in general, the index expansion is needed. The additional filtering is needed to a long sentence, or it may be partitioned into subsentences by its colon or semicolon, further.

Let us make some remarks on the hierarchical index of a text. It is indexed into a list of words, and some of them are filtered in addition in the root node. A text is partitioned into paragraphs, and each paragraph is indexed in the second level. In the third level, each paragraph is partitioned into sentences, and each sentence is indexed. The words in the root node become keywords in the current text.



**Fig. 15.21** Representative word selection in each sentence



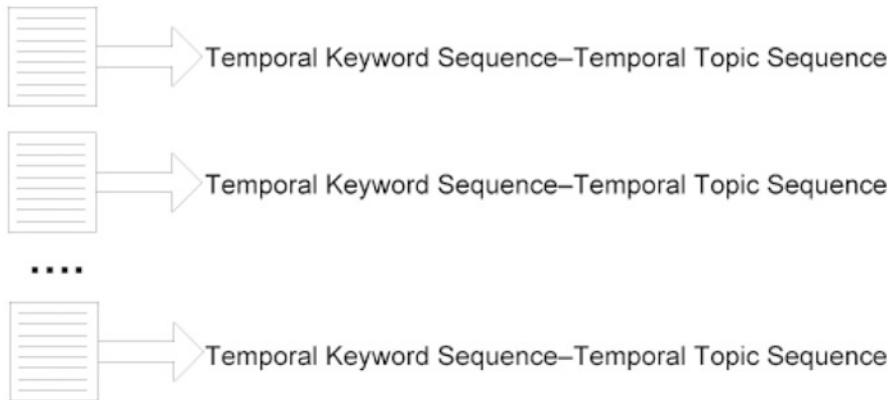
**Fig. 15.22** Mapping paragraph sequence into word sequence

### 15.3.4 Temporal Topic Analysis

This section is concerned with the textual topic analysis as a kind of text convolution. The textual topic analysis was mentioned in [7], and the HMM (Hidden Markov Model) is applied to the task. A text is given as the input, it is encoded into a word sequence as an observation sequence, and the topic sequence with its maximal probability is found as the state sequence. A text is mapped into a temporal sequence of topics, and it is used as the classification target. This section is intended to describe the temporal topic analysis and the application of HMM to it.

Mapping the temporal paragraphs in a text into a word sequence is illustrated in Fig. 15.22. A text is partitioned into paragraphs, and they are arranged into a temporal sequence. Each paragraph is indexed into words, and the most important word is selected among them as the representative word. The words which are extracted from the paragraphs are arranged into a word sequence. The word sequence is used as the observation sequence for this task.

The training examples each of which consists of an observation sequence and a state sequence is illustrated in Fig. 15.23. The parameters of HMM which is applied to the text topic analysis are the probabilities of initial states, the conditional probabilities of state transitions, and the conditional probabilities of states given observations. The probabilities of initial states are computed by dividing the number



**Fig. 15.23** Training sequences for parameter estimation

$$\begin{aligned}
 & \text{Topic}_{11} \text{Topic}_{12} \dots \text{Topic}_{1P} \xrightarrow{\quad} P(\text{Topic\_Sequence}_1) \\
 & \text{Topic}_{21} \text{Topic}_{22} \dots \text{Topic}_{2P} \xrightarrow{\quad} P(\text{Topic\_Sequence}_2) \\
 & \quad \cdots \quad \cdots \\
 & \text{Topic}_{N1} \text{Topic}_{N2} \dots \text{Topic}_{NP} \xrightarrow{\quad} P(\text{Topic\_Sequence}_N) \\
 & \qquad \qquad \qquad \underset{i=1}{\operatorname{argmax}} P(\text{Topic\_Sequence}_i)
 \end{aligned}$$

**Fig. 15.24** Finding topic sequence from word sequence

of initial states of state sequences by the total number of training examples. Both conditional probabilities are computed by dividing the corresponding frequencies by the total number of training examples. Refer to [7] for getting the detail process.

The process of finding a topic sequence from a word sequence is illustrated in Fig. 15.24. The parameters which are involved in the HMM are estimated from the training examples by the above process. All possible topic sequences are generated, and their probabilities are computed. The topic sequence with its maximal probability is selected as the output. Its detail computation process is provided by [7].

Let us make some remarks on the application of HMM to the textual topic analysis. A text is mapped into a temporal word sequence as an observation sequence, and it is mapped into a temporal topic sequence by the HMM. The parameters which are involved in the HMM are estimated by the training examples, each of which consists of an observation sequence and a state sequence. The probabilities of all possible topic sequences given the observation sequence are computed, and the topic sequence with the maximal probability is selected as the answer. In this task, we may consider mapping a text into multiple word sequences.

## 15.4 Textual Pooling

This section is concerned with the pooling which is specialized for textual data. In Sect. 15.4.1, we mention the partition of a text into sentences or paragraphs. In Sect. 15.4.2, we study the process of downsizing the dimension using the pooling. In Sect. 15.4.3, we mention the process of extracting keywords automatically from a text. In Sect. 15.4.4, we review the text summarization as a kind of textual pooling.

### 15.4.1 Text Partition

This section is concerned with the partition of a text which is needed for implementing the pooling operation. In Sect. 15.3, we studied the textual convolutions which performed directly to a raw text. The essence of the pooling operation is to select representative value from each entity partition. The partition of a text into paragraphs, sentences, or words is necessary for performing the pooling operation. This section is intended to describe the partition of a text into them.

The process of indexing a text into a list of words is illustrated in Fig. 15.25. A word is a minimum semantic unit in the text. The steps of indexing a text are tokenization, stemming, and stop word removal. The process of doing so is viewed as the partition of the text into smallest semantic units. The pooling operation is applied to a list of words by sliding a window.

The partition of a text into sentences is illustrated in Fig. 15.26. If the text is assumed to almost formal, it is segmented depending on the period. The text is expressed into a list of sentences; it is viewed as a set of sentences. A representative word is extracted from each sentence; a list of words is the results from applying this kind of textual pooling. A sentence may be selected from the window which slides on the sentence list, as the alternative way.

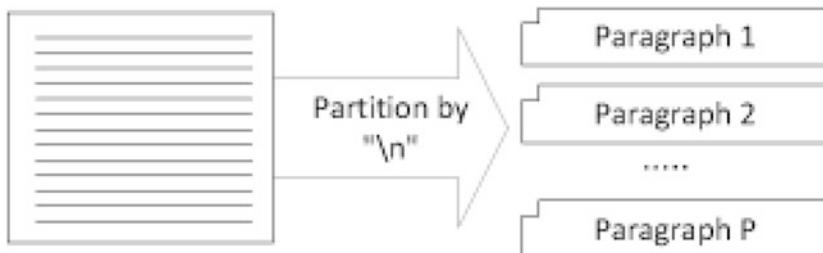
Partitioning a text into paragraphs is illustrated in Fig. 15.27. A text is segmented depending on the carriage return. A text is expressed into a list of paragraphs; it is viewed as a set of paragraphs. A representative word is extracted from each paragraph; a list of words each of which represents its own paragraph is the results



**Fig. 15.25** Partition of text into words by text indexing



**Fig. 15.26** Partition of text into sentences by period mark



**Fig. 15.27** Partition of text into paragraphs by carriage

from applying this kind of textual pooling. We consider selecting a representative sentence from each paragraph as another textual pooling.

Let us make some remarks on the text partition for implementing the textual pooling. A text is partitioned into words by indexing it with the three basic steps. It is partitioned into sentences by the period mark which indicates the sentence end. It is partitioned into paragraphs by the carriage return. Representative partition is selected from the window which slides on the temporal partition list.

#### 15.4.2 Sub-dimensional Down-sampling

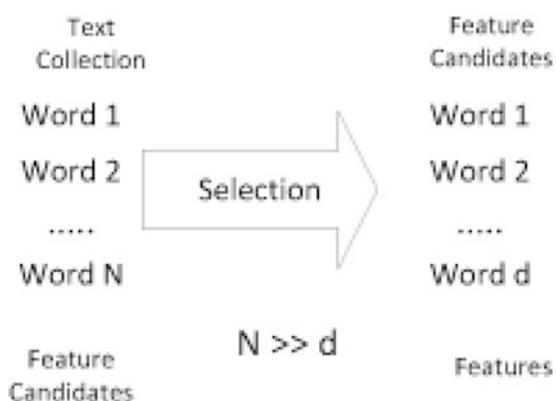
This section is concerned with the sub-dimensional down-sampling. It is the process of reducing the dimension using the pooling operation. A text is encoded into a numerical vector, and the dimension is downsized by the pooling operation. The dimensionality of a numerical vector which represents a text is usually huge in spite of selecting some features. This section is intended to describe the process of reducing the dimension of numerical vector.

The process of indexing a text into a list of words as the feature extraction is illustrated in Fig. 15.28. The corpus which consists of texts is prepared, and they are concatenated into a single text. The integrated text is indexed into a list of words; the words which are generated from indexing the integrated text are feature candidates.

**Fig. 15.28** Extraction of feature candidates from text collection



**Fig. 15.29** Selection of features from feature candidates

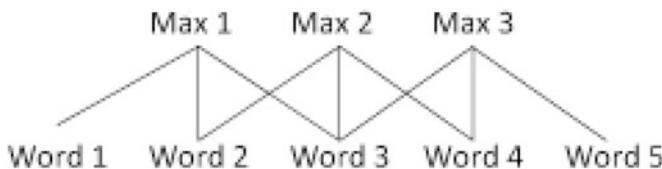


The number of feature candidates is usually tens of thousands. We need to select only some among them.

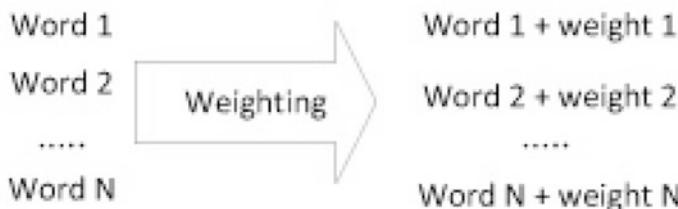
The process of selecting some among the feature candidates as the features is illustrated in Fig. 15.29. The process which is presented in Fig. 15.29 is called feature selection, and this process is needed for improving the efficiency. The selection criteria are defined in advance, and some which satisfying the criteria are selected as features. The number of feature candidates is usually several ten-thousands, and the number of selected features is usually several hundreds, according to [2]. A text is encoded into a vector with its several hundreds dimensionality; the huge dimensionality is the issue.

The pooling to the word list is illustrated in Fig. 15.30. Words are extracted as the features by indexing the integrated text, and some are selected among them. The importance of each feature or its correlation with the output from the sample texts is computed. The word with its maximal importance or correlation is selected from the window which slides on the word list. This kind of pooling is for selecting attributes, rather than attribute values.

Let us make some remarks on the sub-dimensional down-sampling as an instance of textual pooling operation. Feature candidates are extracted by indexing the corpus into a list of words. Some are selected among them as the features. They are selected by applying the pooling operation to the window which slides on



**Fig. 15.30** Dimension reduction by pooling



**Fig. 15.31** Word weighting for keyword extraction

the words, additionally. We consider applying multiple pooling operations to the window without the step of feature selection.

### 15.4.3 Keyword Extraction

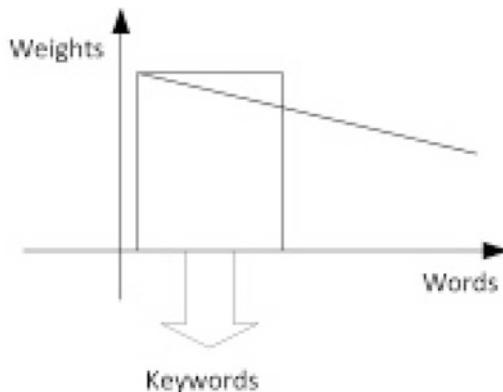
This section is concerned with the keyword extraction as an instance of text pooling. The keyword extraction is the process of selecting and extracting important words which represent the entire contents. It is viewed as an instance of textual pooling, based on the fact the pooling is viewed as the process of selecting the representative value from each window. We consider extracting keywords from a paragraph or a sentence instead of the entire text. This section is intended to describe the process of extracting keywords as an instance of textual pooling.

The text indexing is mentioned as the first step of the keyword extraction. It is the process of extracting a list of words from a text. The tokenization, the stemming, and the stop word removal are the basic steps of text indexing. It is viewed as the segmentation of a text into words. The keyword extraction is the process of selecting some among the words which are indexed from the text.

The process of weighting the words is illustrated in Fig. 15.31. The words are extracted by indexing a text with the three basic steps. The weight is computed for each equation with the equation which was mentioned in Sect. 13.2.4. If the corpus is not available, the frequency or the relative frequency is used for weighting each word. We consider other equations for weighting a word.

The process of selecting some words among the weighted ones is illustrated in Fig. 15.32. A text is indexed into a list of words, and the weights are assigned to

**Fig. 15.32** Selection of highly weighted words as keywords



them. The words are ranked by their weights, and some with their higher weights are selected among them. The alternative way of selecting some words is to select ones with their higher weights than the threshold. The keyword extraction is viewed as a binary classification in [6].

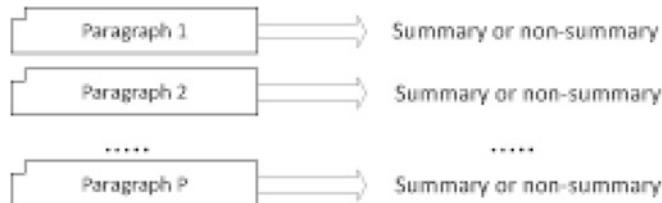
Let us make some remarks on the keyword extraction which is an instance of textual pooling operation. A text is indexed into a list of words by the basic three stops. The weights which indicate importance degrees in the text are assigned to the words. Some with their higher weights are selected among them as the keywords. They are encoded into a numerical vector, viewing a list of keywords as a mapped text.

#### 15.4.4 Text Summarization

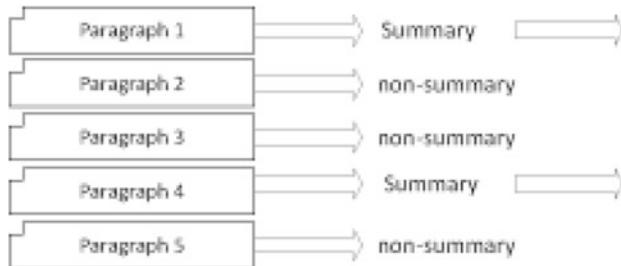
This section is concerned with the text summarization which is an instance of textual pooling. In Chap. 14, we studied the text summarization in detail. In this section, we review the text summarization with the view of textual pooling. The summary is generated by selecting important paragraphs and is encoded into a numerical vector. This section is intended to describe the text summarization which is mapped into a binary classification.

Let us review the text partition as the first step of text summarization. It is the process of partitioning a text into sentences or paragraphs. The boundary between paragraphs is assumed as the carriage return, and the boundary between sentences is the period. We may consider the hierarchical partition of a text whose root is the entire text. It is assumed that a text is partitioned into paragraphs for implementing the text summarization.

The classification of each paragraph into summary or non-summary is illustrated in Fig. 15.33. A text is partitioned into paragraphs by the above process, and each paragraph is encoded into a numerical vector. The machine learning algorithm is



**Fig. 15.33** Classification of each paragraph into summary or non-summary



**Fig. 15.34** Extraction of paragraphs which are classified into summary

trained with the sample paragraphs which are initially labeled with one of the two categories. Each vector which represents a paragraph is classified into summary or non-summary. The paragraphs which are classified into summary are selected as the summary.

The extraction of paragraphs which are classified into summary is illustrated in Fig. 15.34. The text summarization is mapped as a binary classification, and paragraphs are encoded into numerical vectors. The paragraphs which are classified into summary are selected and extracted as the text summary. When the text summarization is viewed into a textual pooling, the summary is encoded into a numerical vector for the text classification. Because a query plays the role of a filter vector, the query-based text summarization is viewed as a textual convolution.

Let us make some remarks on the text summarization as an instance of textual pooling. A text is partitioned into paragraphs by the carriage return. Each paragraph is classified into summary or non-summary. The paragraphs which are classified into summary are extracted as the summary. The text summarization may be connected with other text mining tasks for getting their synergy effect.

## 15.5 Summary and Further Discussions

This section is concerned with the summary and the further discussions on what is studied in this chapter. The deep operations on numerical vectors are applied by encoding texts into numerical vectors. The text convolution is the process of filter

text parts such as words, sentences, and paragraphs, from a text. The text pooling is the process of selecting representative parts; the keyword extraction and the text summarization are typical examples of textual pooling. This section is intended to discuss further what is studied in this chapter.

Let us consider encoding a text into multiple representations. In using multiple criteria for selecting features, it is possible to define multiple different sets of features. A text is encoded into multiple numerical vectors whose features are different from each other. If a raw data is encoded into multiple representations, a single machine learning algorithm is applied by concatenating the multiple representations into a single representation, or multiple machine learning algorithms are applied independently. The type of machine learning which is based on the multiple representations of raw data is called multiple viewed learning.

Let us consider indexing a text into multiple lists of words. The text indexing is viewed as the process of converting a text into a set of words. Different lists of words are expected in adding the additional steps to the basic steps. If a text collection is indexed into multiple lists, it is possible select different sets of features for encoding a text into a numerical vector. Indexing a text into multiple lists is the cause of encoding a text into multiple representations.

Let us consider combining the textual convolution and the textual pooling with each other in implementing the textual deep learning algorithms. In this chapter, we mentioned attaching of either of the textual convolution or the textual pooling to the machine learning algorithm for implementing a deep learning algorithm. Both are attached before encoding a text into a numerical vector. In addition, both the numerical convolution and the numerical pooling are put between the encoding module and the classification module. Multiple convolution layers and multiple pooling layers are arranged alternatively.

Let us mention the implementation of textual deep learning algorithm as an approach to the text classification. The CNN which was covered in Chap. 12 was specialized for the image classification, in that the convolution is based on the image filtering. In this chapter, we define the deep operations, the textual convolution, and the textual pooling, as the basis for implementing the textual deep learning algorithm. It is adopted for implementing the text classification system. We will study the textual deep learning algorithms as approaches to the text classification in next chapter.

## References

1. S. Cho, M. Jang, and S. Chang, “Virtual Sample Generation using a Population of Network”, 83–89, Neural Processing Letters, 5, 1996.
2. F. Sebastiani, “Machine learning in automated text categorization”, 1–47, ACM Computing Survey, 2002.
3. T. Jo and N. Japkowicz, “Class Imbalances versus Small Disjuncts”, 40–49, ACM SIGKDD Exploration, 6, 2004.

4. T. Jo, “The Effect of Mid-Term Estimation on Back Propagation for Time Series Prediction”, 1237–1250, Neural Computing and Applications, 19, 2010.
5. T. Jo, “VTG Schemes for using Back Propagation for Multivariate Time Series Prediction”, 2692–2702, Applied Soft Computing, 13, 2013.
6. T. Jo, “Keyword Extraction in News Articles using Table based K Nearest Neighbors”, The Proceedings of 25th International Conference on Computational Science & Computational Intelligence, 2018.
7. T. Jo, Machine Learning Foundation, Springer, 2021.

# Chapter 16

## Text Classification System



This chapter is concerned with the text classification system which is implemented with a deep learning algorithm. Text classification as the function of the system which is implemented in this chapter is the process of classifying a text into a topic or some topics. We adopt the deep learning algorithm which is modified from an existing machine learning algorithm by adding the convolutional layer as the approach to the text classification. We mention the two kinds of convolutional layer: the numerical convolutional layer which is applied after encoding a text and the textual convolutional layer which is applied before doing it. This section is intended to describe the text classification system and the adopted deep learning algorithms.

This chapter is composed with the five sections, and in Sect. 16.1, we overview what is studied in this chapter. In Sect. 16.2, we study the architecture of the text classification system. In Sect. 16.3, we study the textual deep learning algorithms which are approaches to the text classification. In Sect. 16.4, we study the learning process of textual deep learning algorithms. In Sect. 16.5, we discuss further what is studied in this chapter.

### 16.1 Introduction

This section is concerned with the overview of convolutional text classifier. In the previous chapter, we studied the deep operations which are specialized on textual data. We modify the existing machine learning algorithms by attaching the deep operations. In this chapter, the text classification system is designed by adopting the deep learning algorithms which are modified by the textual deep operation. This section is intended to overview the text deep operations as the introduction to this chapter.

Let us review the CNN which was covered in Chap. 12. There are two parts in the CNN: the feature extraction and the classification. The feature extraction part is composed alternatively with convolution layers and pooling layers. The

classification part is the Perceptron or the Multiple Layer Perceptron (MLP) by itself. Two layers are designed arbitrary in the feature extraction part.

The textual deep operations are alternatively added to the machine learning algorithms. In Chap. 15, we studied the textual convolution and the textual pooling as the deep operations which are specialized for texts. The textual convolution and the textual pooling are attached to the machine learning algorithm for implementing the textual deep learning algorithm. The numerical deep operations may be replaced by the textual deep operations for applying the CNN to the text classification. The textual deep learning algorithm is designed, considering the instances of textual deep operations, their orders, and the machine learning algorithm.

The textual deep learning algorithm may be implemented by attaching the textual deep operations to the machine learning algorithm. Before the part where a text is encoded into a numerical vector, the textual deep operations such as the text convolution and the text pooling are arranged alternatively. The process of classifying a text by the textual deep learning is to convert a text into an optimized text through the textual deep operations, encode it into a numerical vector, and classify it by the given machine learning algorithm. The deep operations on a numerical vector may be added between the encoding part and the classification part; a numerical vector may be converted into another vector in addition. The textual deep learning algorithm is adopted for implementing the text classification system as the approach.

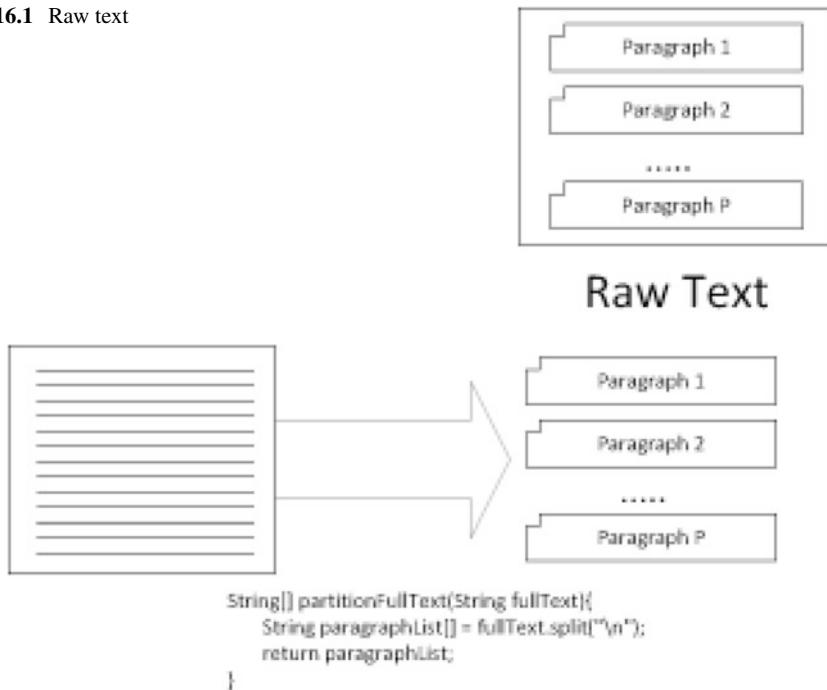
Let us mention what is intended in this chapter. We understand the architecture of textual deep learning algorithm as the frame of implementing the text classification system. We understand the process of classifying a text by the textual deep learning algorithms. We understand the process of training the text deep learning algorithms with sample texts. This chapter is intended to study the textual deep learning algorithms with respect to their architectures, their classification process, and their learning process.

## 16.2 System Architecture

This section is concerned with the frame of applying the deep learning algorithms to the text classification. In Sect. 16.2.1, we study the input layer for preprocessing a text. In Sect. 16.2.2, we study the convolution layer. In Sect. 16.2.3, we study the pooling layer. In Sect. 16.2.4, we design the text classification system which adopts the deep learning algorithm.

### 16.2.1 Input Layer

This section is concerned with the initial layer in the textual deep learning algorithm. This kind of deep learning algorithm is specialized for processing textual data.

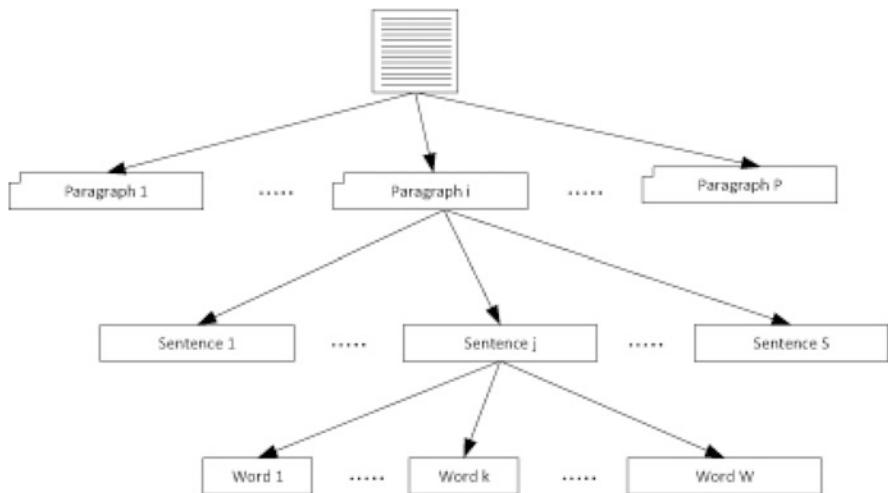
**Fig. 16.1** Raw text**Fig. 16.2** Text partition

The initial layer in the text deep learning algorithm is given as a text. In the input layer, the text is partitioned into paragraphs, or a text is expressed as a hierarchical structure of words. This section is intended to describe the text processing in the input layer.

A raw text is presented in Fig. 16.1. It is assumed that the text is written in English for helping us to understand it. The raw text is composed with more than one paragraph; it consists of  $p$  paragraphs in this example. The raw text is modeled by an ordered set of paragraphs,  $T = \langle P_1, P_2, \dots, P_p \rangle$ . It is viewed as a set of one paragraph, at least.

The partition of a text into a list of paragraphs is illustrated in Fig. 16.2, together with a pseudo code. A raw text is viewed as an ordered set of more than one paragraph, and it is assumed that the text is written correctly by authors. The carriage return is defined as the boundary between paragraphs, and the text is segmented into paragraphs by the carriage return. The list of paragraphs is the output from the text partition. It is necessary for carrying the textual deep operation as well as the text summarization.

The process of representing a text into a hierarchical form is illustrated in Fig. 16.3. The root node in the hierarchical form is given as the entire text. The text is partitioned into paragraphs, and each node is viewed as a paragraph in the next level. In the further level, a paragraph is partitioned into sentences, and each sentence is



**Fig. 16.3** Hierarchical text structure

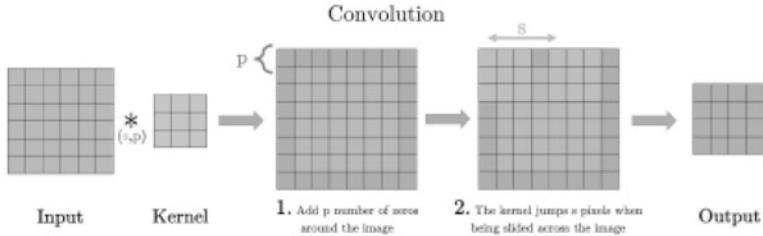
partitioned into words. The hierarchical form is given as the input, instead of a single text, in some application areas.

Let us make some remarks on the view of the raw text as the operation of the textual deep operations. A raw text is defined as an article which consists of more than one paragraph which is written in a natural language. A text is partitioned into paragraph by the carriage return, and a paragraph is partitioned into sentences by the punctuation mark. A text is expressed into a hierarchical form where the root node is the entire text and the nodes in the next level are paragraphs. The textual deep operation is applied to a raw text or its hierarchical form.

### 16.2.2 Convolution Layer

This section is concerned with the convolution layer for implementing the deep learning. A raw text is viewed in the three directions: a raw text, a list of paragraphs, and a hierarchical form. In this section, we study the convolution layer as the process of designing the deep learning algorithm. This convolution layer is used for modifying existing machine learning algorithms into deep learning versions. This section is intended to describe the convolution layer which is needed for implementing the deep learning algorithm.

The architecture of the convolution layer is illustrated in Fig. 16.4 as its frame. The input is assumed to be a matrix which represents an image, and a filter matrix which is called kernel is involved in this layer. The image boundary is augmented with zero values, called padding, and a filter matrix slides on the augmented matrix. Another matrix is constructed by the product of the input matrix and the filter matrix.



**Fig. 16.4** Convolution layer architecture from <https://towardsdatascience.com/what-is-transposed-convolutional-layer-40e5e6e31c11>

$$\begin{bmatrix} x_{11} & x_{12} & \dots & x_{1d} \\ x_{21} & x_{22} & \dots & x_{2d} \\ \dots & \dots & \dots & \dots \\ x_{d1} & x_{d2} & \dots & x_{dd} \end{bmatrix} \cdot \begin{bmatrix} c_{11} & c_{12} & \dots & c_{1h} \\ c_{21} & c_{22} & \dots & c_{2h} \\ \dots & \dots & \dots & \dots \\ c_{h1} & c_{h2} & \dots & c_{hh} \end{bmatrix} \rightarrow \begin{bmatrix} \sum_{j=1}^h c_{j1} x_{j1} & \sum_{j=1}^h c_{j1} x_{j+1} & \dots & \sum_{j=1}^h c_{j1} x_{(j+d-h)} \\ \sum_{i=1}^h \sum_{j=1}^h c_{ij} x_{(i+1)j} & \sum_{i=1}^h \sum_{j=1}^h c_{ij} x_{(i+1)(j+1)} & \dots & \sum_{i=1}^h \sum_{j=1}^h c_{ij} x_{(i+1)(j+d-h)} \\ \dots & \dots & \dots & \dots \\ \sum_{i=1}^h \sum_{j=1}^h c_{ij} x_{(i+d-h)j} & \sum_{i=1}^h \sum_{j=1}^h c_{ij} x_{(i+d-h)(j+1)} & \dots & \sum_{i=1}^h \sum_{j=1}^h c_{ij} x_{(i+d-h)(j+d-h)} \end{bmatrix}$$

**Fig. 16.5** Convolution equation system

$$(d \times d) \odot (h \times h) \rightarrow (d - h + 1 \times d - h + 1)$$

**Fig. 16.6** Convolution results

If there is no padding, the size of the output matrix is reduced, but if the padding is given, the size of the output matrix is same to that of the input matrix.

The equation system of the convolution which is applied to a matrix is illustrated in Fig. 16.5. An input matrix and a filter matrix are involved in this operation; the size of the filter matrix is assumed to be smaller than the size of the input matrix. The element of the output matrix,  $y_{kl}$ , is computed by Eq. (16.1):

$$y_{kl} = \sum_{i=1}^h \sum_{j=1}^h c_{ij} x_{(i+k-1)(j+k-1)} \quad (16.1)$$

The size of output matrix is generated by the convolution is illustrated in Fig. 16.6. It is assumed that the input matrix is a  $d \times d$  matrix, and a filter matrix is a  $h \times h$  matrix. The size of output matrix which is generated from this operation is  $(d - h + 1) \times (d - h + 1)$ . If the  $k$  filter matrices are applied, the results are tensor with  $(d - h + 1) \times (d - h + 1) \times k$ . In the deep learning algorithm, we need to consider the expansion of a matrix into a tensor by using multiple filter matrices.

Let us make some remarks on the convolution layer for processing a matrix. The padding is the augmentation of zero values to the boundary of the input matrix. The output matrix is filled with the product of each element of the input matrix and each

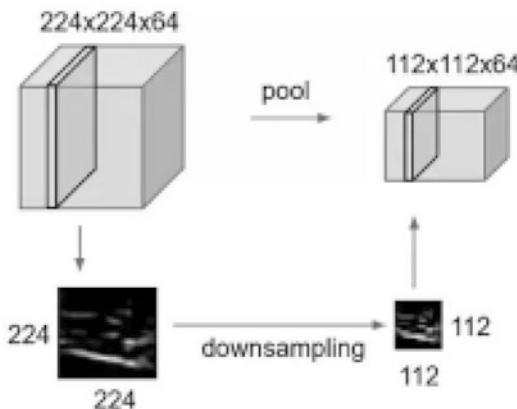
element of the filter matrix, in the convolution. If multiple filter matrices are used, the output matrix is expanded into a tensor. The convolutional neural networks were proposed as the approach to the image classification in [1].

### 16.2.3 Pooling Layer

This section is concerned with the pooling layer as the alternative one to the convolution layer. In the previous section, the convolution layer is needed for implementing the textual deep learning algorithm. In this section, we study the pooling layer which maps the input vector into another by selecting a representative one from each window. In the textual pooling, the word with its highest weight is selected in the window which slides on a text. This section is intended to describe the pooling layer as a component for implementing the textual deep learning algorithm.

The architecture of the pooling layer is illustrated in Fig. 16.7. The input is the 64 matrices, each of which has the size,  $224 \times 224$ ; the input is viewed as a  $224 \times 224 \times 64$  tensor. If the exclusive window is adopted in the pooling operation, the window size is two. If adopt the sliding window, each window size is 113. The  $224 \times 224$  matrix is reduced into  $112 \times 112$  matrix by this pooling operation.

The equation system of the pooling operation on a matrix is illustrated in Fig. 16.8. The input matrix is a  $d \times d$  square matrix which is given in Eq. (16.2), and the window is given as a  $k \times k$  square matrix.



[https://computersciencewiki.org/index.php/Max-pooling/\\_/Pooling](https://computersciencewiki.org/index.php/Max-pooling/_/Pooling)

**Fig. 16.7** Pooling layer architecture from [https://computersciencewiki.org/index.php/Max-pooling/\\_/Pooling](https://computersciencewiki.org/index.php/Max-pooling/_/Pooling)

$$\begin{bmatrix} x_{11} & x_{12} & \dots & x_{1d} \\ x_{21} & x_{22} & \dots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ x_{d1} & x_{d2} & \dots & x_{dd} \end{bmatrix} \Rightarrow \begin{bmatrix} \max_{j=1}^k \max_{i=1}^k x_{ij} & \max_{j=1}^k \max_{i=1}^k x_{(i+1)j} & \dots & \max_{j=1}^k \max_{i=1}^k x_{(i-k+d)j} \\ \max_{j=1}^k \max_{i=1}^k x_{(i+1)j} & \max_{j=1}^k \max_{i=1}^k x_{(i+1)(j+1)} & \dots & \max_{j=1}^k \max_{i=1}^k x_{(i+1)(j-k+d)} \\ \vdots & \vdots & \ddots & \vdots \\ \max_{j=1}^k \max_{i=1}^k x_{(i-k+d)j} & \max_{j=1}^k \max_{i=1}^k x_{(i-k+d)(j+1)} & \dots & \max_{j=1}^k \max_{i=1}^k x_{(i-k+d)(j-k+d)} \end{bmatrix}$$

**Fig. 16.8** Pooling equation system**Fig. 16.9** Pooling results

$$(d \times d) \odot (k \times k) \rightarrow (d - k + 1 \times d - k + 1)$$

$$\begin{pmatrix} x_{11} & x_{12} & \dots & x_{1d} \\ x_{21} & x_{22} & \dots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ x_{d1} & x_{d2} & \dots & x_{dd} \end{pmatrix}. \quad (16.2)$$

Each element in the output matrix is computed by Eq. (16.3),

$$y_{rs} = \max_{i=1}^k \max_{j=1}^k x_{(r+i-1)(s+j-1)}. \quad (16.3)$$

The maximum value is selected from the window which slide on the matrix is selected. The softmax pooling operation is adopted in this case.

The size of output matrix by the pooling operation is illustrated in Fig. 16.9. The size of input matrix is  $d \times d$ , and the size of window matrix is  $k \times k$ . If the sliding pooling operation is adopted, the size of output matrix is  $(d - k + 1) \times (d - k + 1)$ . For example, if the size of input matrix is  $128 \times 128$  and the size of window is  $12 \times 12$ , the size of output matrix is  $117 \times 117$ . If multiple pooling operations are defined, the results from applying them are given as a tensor, rather than a matrix.

Let us make some remarks on the pooling operation which is applied to a matrix. The softmax pooling is adopted, and a window is given as a square matrix. The maximum value is selected from the window which slides on the matrix. If the input matrix is a  $d \times d$  matrix and the window is a  $k \times k$  matrix, the output matrix becomes a  $(d - k + 1) \times (d - k + 1)$  times matrix. We consider encoding a text into a matrix as well as an image.

#### 16.2.4 Design

This section is concerned with the design of the deep learning algorithm which is applied to the text classification. In the previous sections, we studied the parts of the text classification system where the deep learning algorithm is adopted. In

this section, we study the entire organization of the part in the text classification system, which is called convolutional text classifier. The machine learning algorithm which is adopted for implementing the system is modified into its deep version. This section is intended to describe the entire architecture of the text classification system.

Let us mention the pooling layer in designing the text classification system with the deep learning algorithm. If the input data is a numerical vector, the role of the pooling layer is to downsize the input vector. If the input data is a text, it is downsized by the keyword extraction and the text summarization. A list of important words is extracted from a text from the keyword extraction, and some paragraphs are extracted as the summary by the text summarization. A text is downsized into a smaller text as the effect of textual pooling layer.

Let us mention the convolution layer in designing the text classification system. The role of the layer is to generate more training examples from a particular training example. If a single filter is involved, the convolution layer looks a pooling layer, but in using multiple filters, multiple vectors or matrices are generated from an input vector or matrix. In the textual convolution, multiple texts are generated from an input text by trying random partition selection multiple times. The effector of the textual convolution layer is to downsize a text and generate multiple texts.

Let us consider adopting a machine learning algorithm for implementing the text classification system. There are two ways of doing that: adoption of an existing machine learning algorithm by installing the pooling and the convolution and adoption of a deep learning algorithm. If an existing machine learning algorithm is adopted, the sample texts are mapped into other forms by the textual deep operations, the mapped texts are encoded into numerical vectors, and they are mapped into other vectors by the numerical deep operations. The machine learning algorithm is trained with the finally mapped numerical vectors. We consider the choice of a single machine learning algorithm or multiple machine learning algorithms in implementing the text classification system.

Let us make some remarks on designing the text classification system with the deep learning algorithm. The pooling layer is added for reducing the dimensionality and the sparsity of the numerical vector. The convolution layer is added for reducing the dimensionality and generating artificial training examples. An existing machine learning algorithm may be selected, together with the pooling layer and the convolution layer, or a deep learning algorithm such as the restricted Boltzmann machine (RBM) and the recurrent neural networks. Dual convolution layers and dual pooling layers may be considered for implementing the text classification system: textual ones before the encoding process and numerical ones after it.

### 16.3 Text Classification Process

This section is concerned with the process of classifying a text by the deep learning algorithms. In Sect. 16.3.1, we apply the convolutional k-nearest neighbor (KNN)

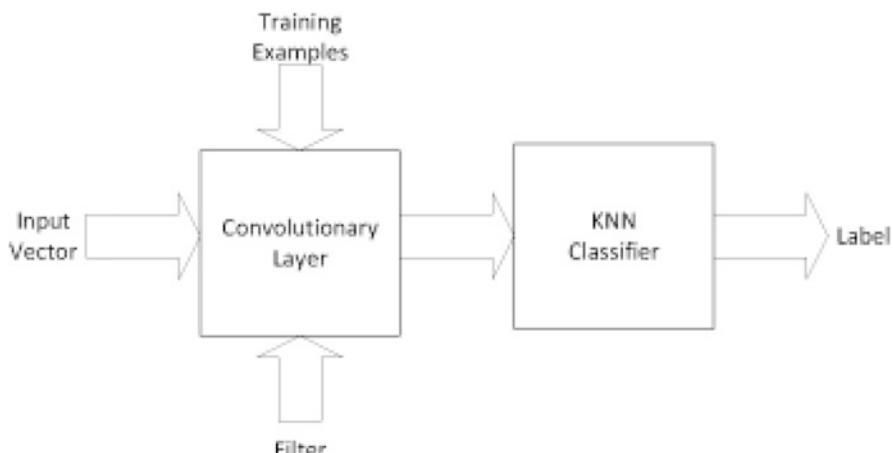
algorithm to the text classification. In Sect. 16.3.2, we apply the convolutional Naive Bayes to the text classification. In Sect. 16.3.3, we apply the RBM to the text classification. In Sect. 16.3.4, we apply convolutional neural networks to the task.

### 16.3.1 Convolutional KNN

This section is concerned with the convolutional KNN as an approach to the text classification. In the previous sections, we mentioned the frame of designing the text classification system which adopts the deep learning algorithm. In this section, we study the process of applying the convolution KNN which is covered in Chap. 5 to the text classification. The convolution KNN is the deep version of KNN with installing the convolution layer and the pooling layer. This section is intended to describe the convolution KNN with respect to its classification process.

The architecture of the convolutional KNN algorithm which is applied to the text classification is illustrated in Fig. 16.10. We consider the two convolution types: textual convolution and numerical convolution. Sample texts which are labeled with their own categories are prepared and transformed into other forms by the convolution layer. A novice text is given as the input and is transformed into another form by the convolution layer. The similarity between the transformed ones is computed in this version of KNN algorithm.

The process of classifying a data item by the convolutional KNN is illustrated in Fig. 16.11. The convolution is applied to each training example with multiple filters. For each training example, the similarities of the novice item with its convoluted ones are computed, and the average of them is the similarity between a training example and a novice item. The training examples with their highest similarities are



**Fig. 16.10** Convolutionary KNN architecture

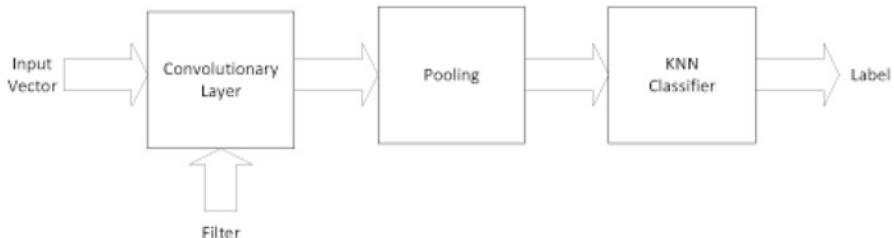
```

Category classifyByConvolutionaryKNN(Item dataItem, List trainExampleList, List filterList, int K){
    int trainExampleSize = trainExampleList.size();
    int filterSize = filterList.size();

    for(int i = 0; i < trainExampleSize; i++){
        Item trainExample = trainExampleList.getElement(i);
        double totalSimilarity = 0.0;
        for(int j = 0;j < filterSize;j++){
            Filter ff = filterList.getElement(j);
            Item filteredDataItem = ff.filter(dataItem);
            Item filteredTrainExample = ff.filter(trainExample);
            double similarity = filteredDataItem.computeSimilarity(filteredTrainExample);
            totalSimilarity = totalSimilarity + similarity;
        }
        double similarity = dataItem.computeSimilarity(trainExample);
        trainExample.setSimilarity(totalSimilarity);
        trainExampleList.setElement(trainExample,i);
    }
    List nearestNeighborList = trainExampleList.getNearestNeighborList(K);
    Category cc = nearestNeighborList.vote();
    return cc;
}

```

**Fig. 16.11** Classification process in convolutionary KNN



**Fig. 16.12** Advanced convolutionary KNN architecture

selected as the nearest neighbors, and their labels are voted for deciding the label of the novice item. The fact that the similarities of the novice item with multiple convoluted vectors are computed for each training example is the difference from the traditional version of KNN algorithm.

The advanced architecture of the convolution KNN algorithm is illustrated in Fig. 16.12. A text is encoded into a numerical vector, and the convolution is applied to it with multiple filter vectors. Multiple transformed vectors as many as filter vectors are generated from the convolution layer, and the pooling is applied to them. The similarities of finally transformed vectors with the finally transformed training vectors are computed for selecting nearest neighbors, and the label is decided by voting ones of the nearest neighbors. The role of the convolution layer is to generate more examples from a single example, and the role of the pooling layer is to reduce the dimensionality.

Let us make some remarks on the convolution KNN algorithm which is adopted as an approach to the text classification. The convolutional layer and the KNN classifier are the components in this version. In the convolution layer, a text is mapped into another text in the textual convolution layer, the mapped text is encoded into a numerical vector, and the vector is mapped into another vector in the numerical convolution layer. The pooling layer is added for implementing more advanced convolution KNN algorithm. Multiple KNN classifiers are considered to the case where multiple vectors are generated by applying the convolution with multiple filter vectors.

### 16.3.2 *Convolutional Naive Bayes*

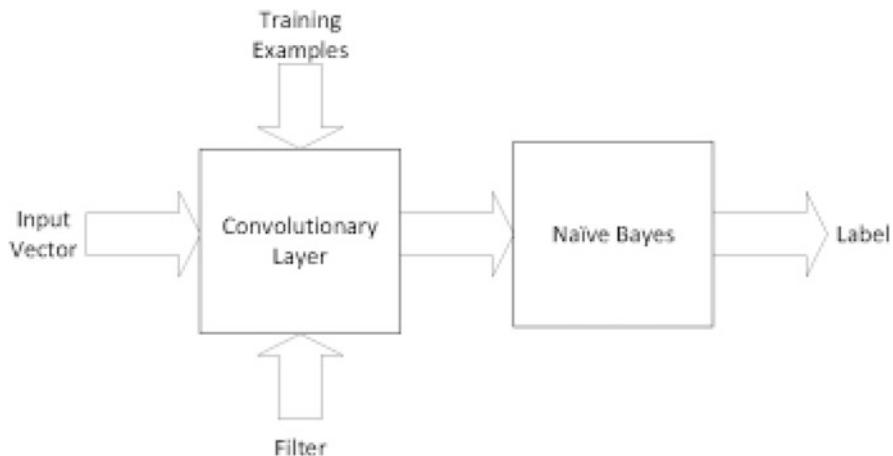
This section is concerned with the convolutional Naive Bayes as an approach to the text classification. In the previous section, the KNN algorithm is modified into the deep version by adding the convolutional layer. The Naive Bayes is modified into its deep version by doing so as the approach to the text classification. If the convolution is applied with multiple filter vectors, there are two ways of dealing with multiple vectors: integrated scheme and independent scheme. This section is intended to describe the convolutional Naive Bayes which is applied to the text classification.

The architecture of deep version of Naive Bayes is illustrated in Fig. 16.13. Some filter vectors are defined for applying the convolution to both sample texts and a novice text. If multiple filter vectors are defined, multiple vectors are generated from a single vector through the convolution layer. The Naive Bayes is applied to the vectors which are mapped from the convolution layer. We may consider the two ways of dealing with the multiple vectors which are mapped from a single vector.

The scheme of applying the Naive Bayes for classifying multiple vectors which represent a data item. They are generated by applying the convolution to a single novice input vector with multiple filter vectors. Each of the  $K$  vectors is classified by the Naive Bayes. The final label is decided by voting the labels of the  $K$  vectors. A single Naive Bayes is trained with all the mapped vectors which represent the training examples, in this scheme.

Another scheme of applying the Naive Bayes to the classification of multiple vectors. In the above scheme, multiple vectors which represent a single novice text are classified by a single Naive Bayes. In this scheme, they are classified by different Naive Bayes models, and the voting is applied to them. In the training phrase, each training example is mapped into  $K$  mapped examples by the convolution layer, the different  $K$  training sets are constructed, and each Naive Bayes is trained with its own training set. In this scheme, each Naive Bayes model is trained with its own view to the training examples.

Let us make some remarks on the convolutional Naive Bayes as the approach to the text categorization. Multiple vectors are generated from a single vector by the convolution layer. Multiple vectors which are mapped from a single vector are



**Fig. 16.13** Convolutionary Naive Bayes architecture

classified by a classifier, and their classified labels are voted for deciding the final label of a novice data item. Each Naive Bayes which corresponds to a filter vector is trained with its own training set, and the ensemble learning is applied to the multiple Naive Bayes models. The pooling layers may be added to this version of Naive Bayes.

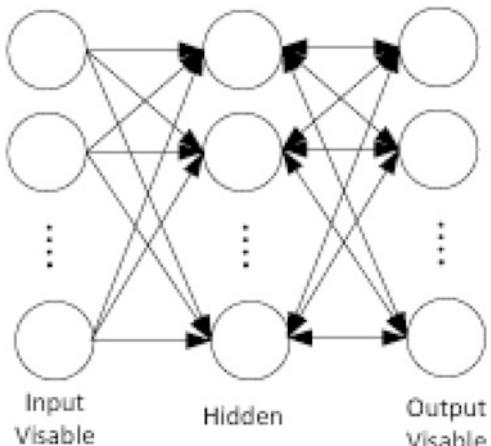
### 16.3.3 Restricted Boltzmann Machine

This section is concerned with the RBM as an approach to the text classification. The RBM is designed for the associative memory which restores an incomplete input into its completed one, as a single version. The RBM is designed as a supervised learning algorithm by stacking two RBMs. The stacked version of RBM is applied to the text classification. This section is intended to describe the stacked version as an approach to the text classification.

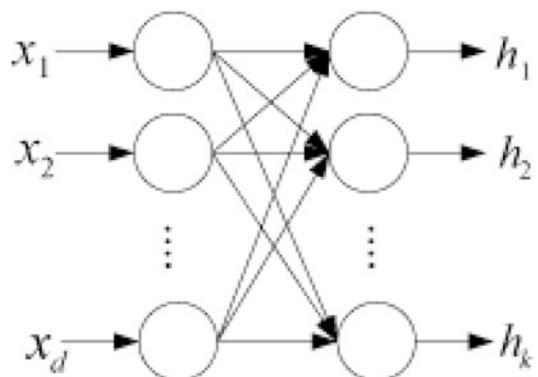
The entire architecture of RBM for implementing the text classification system is illustrated in Fig. 16.14. The dual RBMs are designed for classifying a text, and both the input vector and the target output vector are given as the visible vector. The input layer and the hidden layer are involved in the left RBM, and the hidden layer and the output layer are involved in the right RBM. Even if they have identical architectures, and the dual RBM and the MLP are different from each other. Its learning process of this version of RBM is described in Sect. 16.4.3.

The left part of the stacked RBM is illustrated in Fig. 16.15. In this section, we mentioned the process of classifying a text and mention the process of training it in Sect. 16.4.3. It is assumed that the weights in the left part are optimized and the hidden vector is computed by the novice input vector. Even if the input vector is

**Fig. 16.14** Architecture of RBM as text classifier



**Fig. 16.15** RBM between input layer and hidden layer

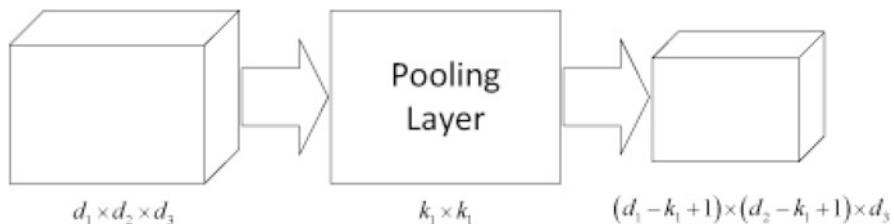
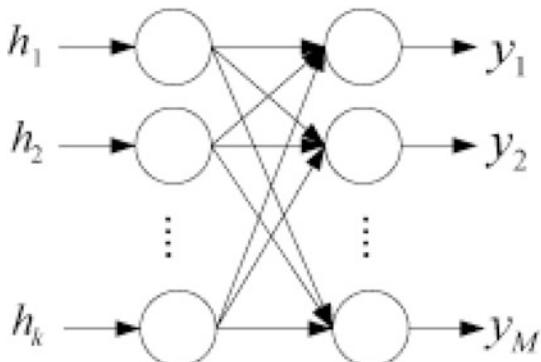


computed by the hidden vector in the RBM, we are not interested in the computed input vector. The hidden vector is used for computing the output vector in the right part of the stacked RBM.

The right part of stacked RBM is illustrated in Fig. 16.16. The input layer in the left part is the visible layer, and the output in the right part is also the visible layer. The weights in the right part are assumed to be optimized, and the output values are computed by the hidden values. The output values are computed from the input layer by means of the hidden layer. The process of computing the output values is identical to one in the MLP.

Let us make some remarks on the application of the RBM to the text classification. The stacked version with the two RBMs is adopted as the approach to the text classification. The hidden node values are computed by the input vector which represents a text in the left RBM. The output values are computed by the hidden node values in the right RBM. In the future study, multiple stacked RBM is considered as another approach to the text classification.

**Fig. 16.16** RBM between hidden layer and output layer



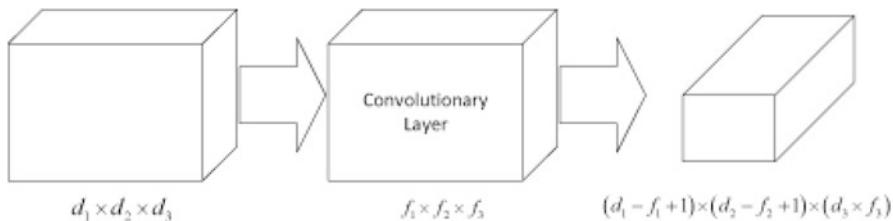
**Fig. 16.17** Pooling layer in convolutionary neural networks

### 16.3.4 Convolutional Neural Networks

This section is concerned with the process of applying the CNN to the text classification. The CNN was described in Chap. 12 as a deep learning algorithm. In this section, we study the process of classifying a text into one among the predefined categories using the CNN. It is possible to encode a text into a matrix as well as a vector by assigning multiple values to each feature. This section is intended to describe the process of applying the CNN to the text classification.

The pooling layer in applying the CNN to the text classification is illustrated in Fig. 16.17. The input is assumed as a tensor with the size,  $d_1 \times d_2 \times d_3$ :  $d_3$   $d_1 \times d_2$  matrices. The pooling operation with the  $k_1 \times k_1$  window is applied to each  $d_1 \times d_2$  matrix; the maximum value is selected among the values in the sliding window. The output tensor with the size,  $(d_1 - k_1 + 1) \times (d_2 - k_1 + 1) \times d_3$ , results from applying the pooling operation. The role of the pooling operation is to reduce the size of input data.

The convolution layer in applying the CNN to the text classification is illustrated in Fig. 16.18. The input for the convolution consists of  $d_3$   $d_1 \times d_2$  matrices, and the filter consists of  $f_3$   $f_1 \times f_2$  matrices. The convolution is applied to the input matrix with the size,  $d_1 \times d_2$  with each filter matrix with the size,  $f_1 \times f_2$ . The  $d_3 \times f_3$  matrices with the size,  $(d_1 - f_1 + 1) \times (d_2 - f_2 + 1)$  are obtained as



**Fig. 16.18** Convolutionary layer in convolutionary neural networks

the results from doing that. If the convolution is applied, the tensor with the size,  $(d_1 - f_1 + 1) \times (d_2 - f_2 + 1) \times (d_3 - f_3 + 1)$  is obtained.

Let us mention the process of classifying a text by the CNN. If the textual convolution is applied, it is applied to the raw text, and the transformed one is encoded into a numerical vector. If the numerical convolution is applied, the text is encoded into a numerical vector, and the operation is applied to it. The final numerical vector is classified by the MLP or the Perceptron. The CNN is viewed as the MLP or the Perceptron which is augmented with the convolution and the pooling.

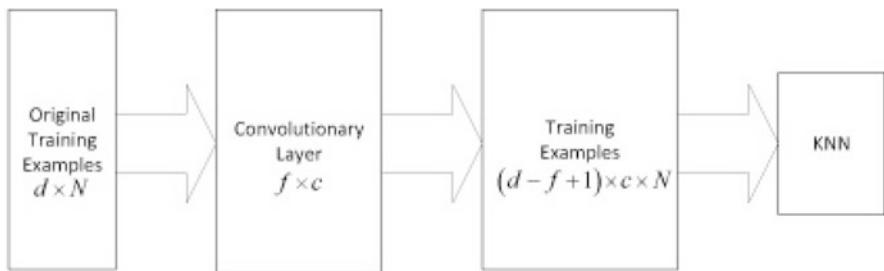
Let us make some remarks on the application of CNN to the text classification. A text is encoded into a matrix with multiple sets of features, and the input data to the pooling is assumed as a tensor. The convolution is applied with the assumption of a tensor as multiple matrices. The final structured form which is mapped by the pooling and the convolution is classified by the Perceptron and the MLP. The number of CNN versions is almost infinite, depending on the design of the pooling layer and the convolution layer.

## 16.4 Learning Process

This section is concerned with the process of training deep learning algorithms which are adopted for implementing the text classification system. In Sect. 16.4.1, we study the process of training the convolutional KNN algorithm. In Sect. 16.4.2, we study the process of training the convolutional Naive Bayes. In Sect. 16.4.3, we study the process of training the RBM. In Sect. 16.4.4, we study the process of training the convolutional neural networks.

### 16.4.1 Convolutional KNN

This section is concerned with the learning process of the convolutional KNN algorithm. In Sect. 16.3.1, we studied the process of classifying a text by it. In this



**Fig. 16.19** Training examples mapped by convolutional layer

section, we study the process of training the convolutional KNN algorithm with the training examples. Its learning process is to map the training example into other forms by the pooling and the convolution. This section is intended to describe the process of training the convolutional KNN algorithm.

The convolutional layer for processing the training examples is illustrated in Fig. 16.19. Each training example is given as a  $d$  dimensional vector, and the number of training examples is  $N$ . In the convolutional layer,  $c$  filter vectors with the  $f$  dimensionality are defined. The  $N$  numerical vectors with the  $d$  dimensionality are transformed into  $N \times c$  numerical vectors with  $(d - f + 1)$  dimensionality. More transformed training examples with less dimensionality are generated by the convolutional layer.

Let us mention the application of the convolution to each training example with multiple filter vectors. The training set is notated by  $Tr = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$ , and the filter vectors are notated by  $\mathbf{f}_1, \mathbf{f}_2, \dots, \mathbf{f}_c$ . The training example and the filter vector are notated, respectively, by  $\mathbf{x}_i = [x_{i1} \ x_{i2} \ \dots \ x_{id}]$  and  $\mathbf{f}_j = [f_{j1} \ f_{j2} \ \dots \ f_{jh}]$ , and each element in the transformed vector is computed by Eq. (16.4):

$$r_{ijk} = \sum_{l=1}^h x_{i(k+l-1)} \cdot f_{jl}. \quad (16.4)$$

The training set,  $Tr$ , which consists of  $N d$  dimensional vectors, is mapped into the training set,  $Tr_{conv} = \{\mathbf{r}_{11}, \dots, \mathbf{r}_{1c}, \dots, \mathbf{r}_{N1}, \dots, \mathbf{r}_{Nc}\}$ , which consists of  $N \times c$   $(d - h + 1)$  dimensional vectors by the convolutional layer. The element of the training set,  $Tr_{conv}$ , is used for computing their similarities with a novice item.

Let us mention the process of classifying a text by the convolution KNN. A text is encoded into a numerical vector,  $\mathbf{x}$ . By the convolution layer, the vector,  $\mathbf{x}$ , is transformed into  $c$  vectors,  $\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_c$ . Each transformed vector,  $\mathbf{r}_i$ , is classified, and the labels of  $c$  vectors,  $\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_c$ , are voted. We consider using  $c$  KNN algorithms each of which corresponds to its own channel.

Let us make some remarks on the learning process of the CNN which is applied to the text classification. It is composed with the feature extraction part and the KNN part. The convolution is applied to the training examples which

are given as numerical vectors with multiple filter vectors. A novice input vector which represents a novice text is mapped into multiple vectors with their lower dimensionalities, each of them is classified, and their classified labels are voted. We consider the ensemble learning of KNN algorithm to the multiple vectors which represent a single text.

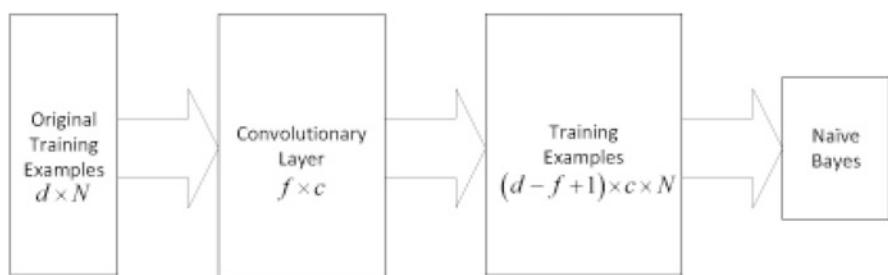
### 16.4.2 Convolutional Naive Bayes

This section is concerned with the learning process of Convolutional Naive Bayes. In Sect. 16.3.2, we studied the process of classifying a text by the Convolutional Naive Bayes. In this section, we study the process of training it with the training examples. Each training example is mapped into multiple transformed vectors with their less dimensionality by the convolution with multiple filter vectors. This section is intended to describe the process of training the convolutional Naive Bayes with the training example.

The architecture of the convolutional Naive Bayes is illustrated in Fig. 16.20. In this architecture, the convolutional layer is the feature extraction part, and the Naive Bayes is the classification part. The convolution is applied to each training example with multiple filter vectors. The number of vectors which are mapped by the convolution is multiple times of the number of training examples. The Naive Bayes are applied to the mapped vectors, instead of the training examples.

Let us mention the process of estimating the parameters as the learning process of Naive Bayes. Each training example is mapped into multiple vectors by applying the convolution with multiple filter vectors. The likelihoods of individual attribute values are computed from the mapped vectors. As the alternative way, the likelihoods are computed channel by channel. The likelihoods of individual attribute values are used for classifying a novice input vector.

Let us mention the process of classifying a novice text. A novice text is encoded into a numerical vector, and it is encoded into multiple vectors with the convolutional layer. Their likelihoods to the categories are computed based on the



**Fig. 16.20** Naive Bayes with convolutionary layer

parameters which are estimated by the above process. The category is decided by the maximal likelihood for each mapped vector, and the final category is decided by voting the classified ones of multiple vectors. We consider multiple Naive Bayes which correspond to channels.

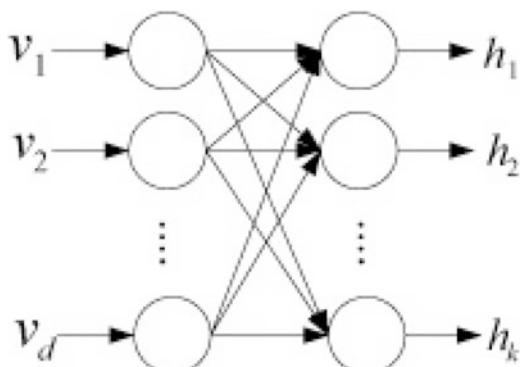
Let us make some remarks on the learning process of the convolutional Naive Bayes. The deep version consists of the convolutional layer and the Naive Bayes. In the learning process, the training examples are mapped through the convolutional layer, and the parameters which are the likelihoods of individual attribute values are estimated from the mapped ones. In the classification process, the input vector is mapped into multiple vectors by the convolutional layer, each of them is classified by computing its maximal likelihoods, and their classified labels are voted for deciding the final label. If multiple convolutional layers are put in front of the Naive Bayes, they are organized hierarchically.

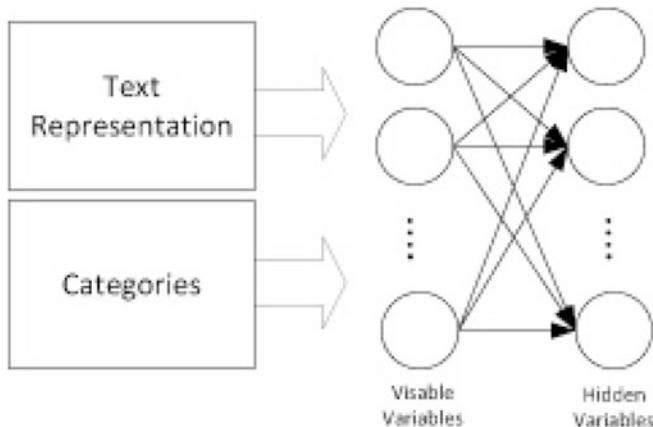
### 16.4.3 Restricted Boltzmann Machine

This section is concerned with the learning process of RBM which is applied to the text classification. In Sect. 16.3.3, we studied the classification process of RBM. In this section, we explain the RBM with the focus on its learning process. The input vector which represents a text is augmented with its target category as the visible vector in applying the RBM to the text classification. This section is intended to describe the process of training the RMB as an approach to the text classification.

The single RBM is illustrated in Fig. 16.21, and let us review its learning process. The weights are initialized at random, and the hidden values are computed by the visible values and the weights. The visible values are computed by the weights and the computed hidden values, and the weights are optimized for minimizing the difference between the target visible vector and the computed visible vector. The single RBM is expanded into the multiple stacked one by connection multiple

**Fig. 16.21** Single restricted Boltzmann machine





**Fig. 16.22** Restricted Boltzmann machine for text classification

RBMs serially. If the RBM is applied to the classification task, the input vector is augmented with its target label as a visible vector.

The process of applying the RBM to the text classification is illustrated in Fig. 16.22. The task is assumed as the soft classification where each text is allowed to be labeled with multiple categories. A text is encoded into a numerical vector, and multiple categories are codified into a binary vector. The numerical vector which represents the text is augmented with the binary vector which indicates its labeled categories into the visible vector. The weights between the visible layer and the hidden layer are optimized for minimizing the difference between the target visible vector and the computed one.

Let us mention the process of classifying a text by the RBM. The weights between the visible layer and the hidden layer are optimized. The input text is encoded into a numerical vector, and it is augmented with the zero vector with the dimension as number of the predefined categories. The hidden vector is computed, and the visible vector is recomputed by the computed hidden vector. The label of the novice vector is decided from the part which is augmented of the input vector.

Let us make some remarks on the learning process of RBM which is applied to the text classification. Its learning process is to iterate computing both the hidden vector and the visible vector and updating the weights. The text classification is assumed as the soft classification which multiple categories are allowed to be assigned to each text, and the numerical vector which represents the input text is augmented with the binary vector which indicates the target output as the visible vector. The weights between the visible layer and the hidden layer are updated. The multiple stacked RBM is applied with the input vector as the first visible vector and the target output vector as the last visible vector to the text classification as the alternative way.

### 16.4.4 Convolutional Neural Networks

This section is concerned with the learning process of CNN which is applied to the text classification. In Chap. 12, we already studied the CNN as the deep learning algorithm, and we studied the process of classifying by the CNN in Sect. 16.3.4. In this section, we focus on the learning process of CNN in applying it to the text classification. The query-based text summarization is adopted as an instance of convolution; the query which is given in this type of text summarization plays the role of a filter vector in the numerical convolution. This section is intended to describe the learning process of CNN as the approach to the text classification.

The architecture of CNN which is applied to the text classification is illustrated in Fig. 16.23. The architecture was initially mentioned in Sect. 12.4.4. A text is encoded into a numerical vector, it is mapped into the vector with its less dimensionality by the pooling layer, and it is mapped into multiple vectors by the convolutional layer. The multiple vectors are classified into one among the predefined categories by the ReLU. The textual convolution is considered before the process of encoding a text into a numerical vector.

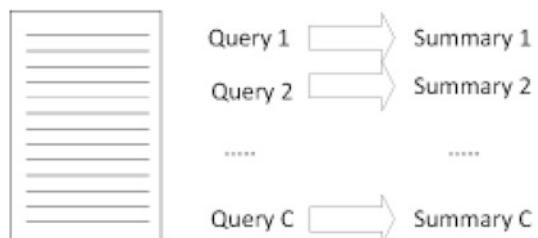
The textual convolution by the query-based text summarization is illustrated in Fig. 16.24. The text summarization which was covered in Chap. 14 is viewed as a textual pooling in that the text length is reduced. If different queries are provided for summarizing a text, different paragraphs may be extracted as summary, depending on the query. The role of queries is the filter vectors in the numerical convolution; the query-based text summarization is interpreted into filtering contents depending on the query. The text convolution is arranged in front of the step of encoding the text into a numerical vector.

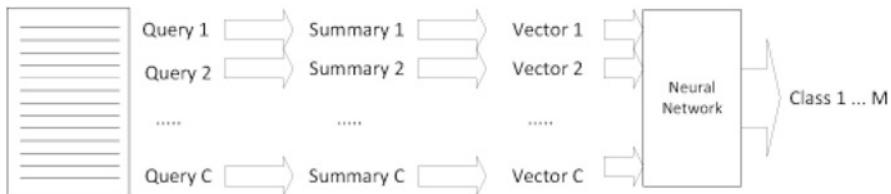
The entire process of classifying a text by the convolutional neural networks is illustrated in Fig. 16.25. The input text is summarized based on the  $C$  queries; the  $C$  summary versions are extracted. Each version is encoded into a numerical vector,



**Fig. 16.23** Architecture of convolutionary neural networks

**Fig. 16.24** Textual convolution by text summarization





**Fig. 16.25** Convolved vectors by textual convolutionary layer

and the  $C$  numerical vectors are generated from a single input text. The  $C$  numerical vectors are classified, and the category is decided by their classified labels. The role of query-based text summarization is to map a text into multiple texts.

Let us make some remarks on the learning process of CNN which is applied to the text classification. The version which consists of a pooling layer, a convolutional layer, and a ReLU is adopted. In addition, we consider the query-based text summarization as the textual convolution. Multiple vectors which are generated by the textual convolution are classified, and their labels are voted for deciding the category. We consider encoding paragraphs, instead of a single entire text into multiple numerical vectors, each of which corresponds to a paragraph.

## 16.5 Summary and Further Discussions

This section is concerned with the summary and the further discussions on what is studied in this chapter. The convolution layers and the pooling layers are alternatively arranged in the architecture of the deep learning algorithms which are applied to the text classification. The four deep learning algorithms, the convolutional KNN, the convolutional Naive Bayes, the RBM, and the CNN are decided as the approach candidates to the text classification. The CNN which is mentioned in this chapter is different from one which is covered in Chap. 12, in that the pooling layer and the convolution layer on textual data are considered.

Let us consider combining two machine learning algorithms, the KNN algorithm, and the Naive Bayes. The ensemble learning is applied to them as the independent combination where both algorithms are masters. In the combination where the KNN algorithm is the master, the KNN algorithm used for classifying data items, and the Naive Bayes is used for classifying each training example into nearest neighbor or not. In the combination where the Naive Bayes is the master, the KNN algorithm is used for involving training examples in computing the likelihoods, and the Naive Bayes is used for classifying data items. One among the three combinations is applied to text classification.

The textual deep operations are attached to a combination of the KNN algorithm and the Naive Bayes. The textual deep operations are described in Chap. 15 and attached to the machine learning algorithms in this chapter. The combination of

them may be modified into a deep version by attaching the textual deep operations. This textual deep learning algorithm may be adopted for implementing the text classification system. If this deep learning algorithm is applied to other areas, the textual deep operations may be replaced by ones on numerical vectors in the combination.

We mention applying the textual deep learning operations to the decision tree. In Chap. 7, we studied the modification of the decision tree into its deep version by adding the convolution operation. The textual convolution is attached before encoding a text into a numerical vector. The numerical convolution is attached between the encoding and the classification for implementing a further deep version. The text classification system may be implemented by adopting the textual convolution decision tree.

The single decision tree is expanded into the random forest which consists of multiple decision tree. In the random forest, the training set is partitioned into subsets, and each tree is constructed with its own subset of training examples. The textual deep operations are attached in front of each decision tree. A raw text is converted into another form by the textual deep operations, and the mapped form is encoded into a numerical vector. This version of random forest may be applied to the text classification.

## Reference

1. C.C. Aggarwal, Neural Networks and Deep Learning, Springer. 2018

# Index

## A

Abstracting, 14, 355–363, 376

## B

Backpropagation, 238, 239

Basic recurrent neural networks, 248, 257–260, 275

Bayes classifier, viii, 5, 9, 17, 111, 139, 141, 144–146, 149–153, 156–164

Bayes rule, 140

Boltzmann machine, v, vi, ix, 223, 277–302, 410, 414–416, 420–421

## C

Cascading, vii, 19–21, 83, 84, 92, 96–99, 108, 137, 177

Convolution, 3, 127, 156, 191, 212, 226, 266, 297, 303, 379

Convolutional neural networks (CNN), v, vi, ix, 4, 18, 221–223, 226, 245, 246, 303–326, 400, 403, 404, 408, 411, 416–418, 422–423

## D

Data classification, 5, 6, 9, 31, 35, 108, 167, 196, 198

Data clustering, 6, 16–18, 57, 58, 62, 63, 73, 107, 108, 131, 133, 177

Decision tree, 5, 30, 165, 424

Deep learning, v–viii, x–xi, 3–6, 11–19, 26, 27, 29, 30, 37, 54, 57, 58, 83, 114, 122, 124, 125, 129–131, 133, 135, 137, 141, 150, 153, 159–164, 166, 179, 180, 186, 189, 193, 194, 204–209, 211, 216, 217, 221, 225, 226, 232, 236, 277, 278, 298, 302, 303, 329, 355, 379, 380, 383, 384, 387, 400, 403–411, 416, 417, 422–424

## E

Ensemble learning, v–vii, 3, 9, 19–22, 26, 27, 49–51, 83–109, 137, 176, 179, 304, 309–310, 325, 414, 419, 423

Expert gate, vii, 19–22, 50, 83, 84, 90, 92, 94–97, 99, 108, 137, 176, 179, 190, 302

## H

Hierarchical clustering, 57, 185

## I

Index expansion, x, 329–353, 391

Index optimization, 330, 338, 345, 350–353, 362

## K

KNN algorithm, vi–viii, 5, 6, 9, 22, 30, 32, 35, 37, 54, 55, 113–137, 344, 410–413, 417–419, 423

**L**

- Language modeling, 267, 271–273  
 Linear classifier, vi, viii, 6, 24, 51, 193–222  
 Long short term memory (LSTM), 248, 256, 263–267, 274, 275

**M**

- Multiple layer perceptron (MLP), v, vi, ix, 4, 15, 21, 27, 30, 40, 41, 44, 54, 225–248, 255–259, 262, 268, 274, 280–282, 303, 304, 319, 320, 325, 404, 414, 415, 417  
 Multiple text summarization, 370–377

**N**

- Naive Bayes, viii, 6, 9, 29, 30, 45–47, 54, 55, 80, 111, 146, 153–155, 158, 162, 163, 181–183, 413–414, 417, 419–420, 423  
 Nearest neighbor, viii, 22, 30, 31, 35–37, 54, 55, 72, 113–121, 125, 131–133, 135–137, 412, 423  
 Neural networks, v, vi, ix, x, 29, 30, 37–45, 54, 58, 80, 166, 225, 226, 232, 247–275, 277, 278, 280, 282, 285, 286, 302, 303, 320, 386, 387, 410  
 Numerical deep operation, x, 379–387, 404, 410

**P**

- Perceptron, v, vi, ix, 21, 29, 30, 37, 40–45, 54, 225–246, 248, 282–284, 286, 287, 289, 303, 320, 325, 404, 417  
 Pooling, 4, 122, 149, 175, 209, 246, 266, 297, 309, 353, 356, 379, 403

**Q**

- Query based text summarization, 325, 363–369, 399, 422, 423

**R**

- Random forest, 45, 49–51, 54, 165, 166, 174–179, 183, 187–192, 424  
 Restricted Boltzmann machine (RBM), v, vi, ix, 277–302, 320, 410, 414–417, 420–421, 423

**S**

- Stacked KNN algorithm, 129, 133–134, 137  
 Stacked RBM, 293–301, 414, 415, 421  
 Supervised learning, v–viii, 3–6, 8, 9, 15–17, 26, 29–55, 58, 67, 69, 81, 96, 101, 104–106, 113, 129–131, 157, 178, 183, 213, 216, 218, 279, 290, 298, 414  
 Support vector machine (SVM), v, viii, 13, 24, 29, 45, 51–54, 193–206, 208, 209, 211–213, 217–222, 268, 353

**T**

- Text indexing, x, 329–338, 352, 353, 365, 377, 381, 382, 390, 394, 397, 400  
 Textual classification, x, xi, 137, 164, 269, 327, 344, 353, 362, 376, 380, 399, 400, 403–424  
 Textual convolution, x, xi, 137, 164, 325, 379, 380, 387–394, 399, 400, 403, 404, 410, 411, 413, 422–424  
 Textual deep operation, vi, x, 330, 353, 356, 379–400, 403–406, 423, 424  
 Textual pooling, x, 325, 379, 380, 394–400, 404, 408, 410, 422

**U**

- Unsupervised decision tree, 183–188  
 Unsupervised KNN Algorithm, 114, 131–135  
 Unsupervised learning, v–vii, 3, 4, 6–9, 15–17, 26, 41, 57–81, 86, 90, 101, 106–108, 129–131, 177, 214, 215, 218, 219, 279, 375  
 Unsupervised probabilistic learning, v, vi, 6, 158  
 Unsupervised SVM, 217–220

**V**

- Voting, vii, 19–21, 35, 50, 54, 80, 83, 84, 90–94, 96–99, 106–108, 113, 115–121, 131, 135–137, 176, 177, 179, 190, 207, 302, 309, 310, 383, 412, 413, 420