Chapter 5
Multi-Valued Neuron with a Periodic Activation Function

"The opposite of a profound truth may well be another profound truth"

Niels Bohr

In this Chapter, we consider MVN with a periodic activation function. As we have already seen, MVN's functionality is higher than the one of, for example, sigmoidal neurons. In this Chapter, we will consider how a single MVN may learn non-linearly separable input/output mappings in that initial n-dimensional space where they are defined. In Section 5.1, we consider a universal binary neuron (UBN), which in fact is the discrete MVN with a periodic activation function for k=2. We show how this neuron may learn non-linearly separable Boolean functions, for example, XOR and Parity n, projecting them into larger valued logic. In Section 5.2, we generalize that approach, which is used in UBN, and introduce a periodic activation function for the discrete MVN. We also consider the learning algorithm for MVN with a periodic activation functions. In Section 5.3, we show how a number of non-linearly separable benchmark classification problems can be solved using a single MVN with a periodic activation function. Concluding remarks are given in Section 5.4.

5.1 Universal Binary Neuron (UBN): Two-Valued MVN with a Periodic Activation Function

Introducing complex-valued neurons in Section 1.4, we have shown (see p. 41) that a single complex-valued neuron can learn a classical non-linearly separable problem, the XOR problem. We have mentioned there that a neuron, which can learn the XOR problem, is called the universal binary neuron (UBN).

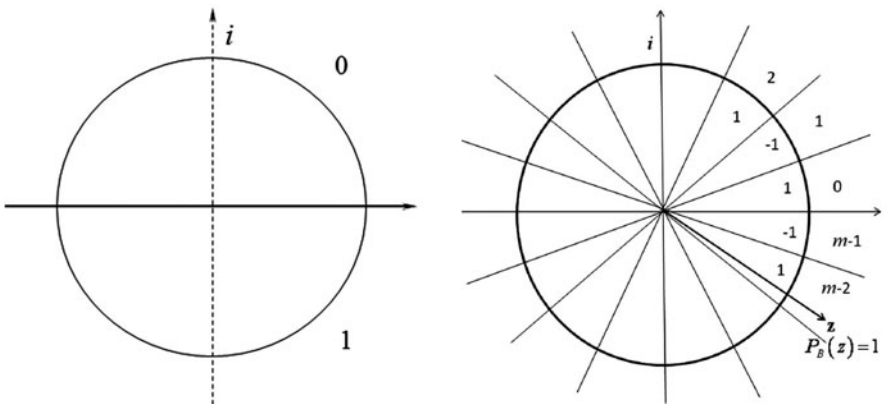
Actually, UBN is nothing else than the two-valued MVN with a periodic activation function. This periodicity is a main idea behind UBN.

5.1.1 A Periodic Activation Function for k=2

We have considered earlier the k-valued activation function (2.50) of the discrete MVN for k=2 (see p. 60). It is transformed to the two-valued function (2.52)

$$P(z)=\begin{cases} 1; 0 \le \arg z < \pi \\ -1; \pi \le \arg z < 2\pi, \end{cases}$$

which divides the complex plane into two sectors, the top and the bottom half-planes (see Fig. 5.48a). However, it was shown in [33, 35, 37, 60] that the functionality of a neuron with this activation function exactly coincides with the functionality of the threshold neuron with the activation function (1.1).



(a) MVN activation function (50) for k=2 (b) A periodic activation function

Fig. 5.48 MVN activation function for k=2 and a periodic activation function for k=2

This means that the discrete MVN with the activation function (2.50) with k=2 (or with the activation function (2.52), which is (2.50) with k=2) can learn only linearly separable input/output mappings. Thus, such binary non-linearly separable problems as XOR, Parity n and others cannot be learned using MVN with the activation function (2.52).

An idea to modify the activation function (2.52) in such a way that it should have ensured non-linearly separated Boolean functions to be learned by a single neuron was developed by the author of this book in his Ph.D. dissertation in 1986. This idea was as follows. We have to use complex weights with the binary inputs and outputs taken from the set $E_2=\{1,-1\}$. Hence, our weighted sum is a complex number. If, dividing the complex plane into two sectors, we cannot learn non-linearly separable Boolean functions, will we be able to do so dividing the complex plane into more than two sectors and determining a periodic activation function? In the paper [29], which was published in 1985, the author of this book suggested the activation function (1.40), which divides the complex plane into four sectors, and ensures the implementation of the XOR function using a single neuron with complex-valued weights (this example we have already considered in detail in Section 1.4, see p. 41). The activation function (1.40)

$$\varphi(z)=\begin{cases}1, \text{ if } 0\le\arg z<\pi/2 \text{ or } \pi\le\arg z<3\pi/2\\ -1, \text{ if } \pi/2\le\arg z<\pi \text{ or } 3\pi/2\le\arg z<2\pi\end{cases}$$

divides the complex plane into four sectors (see Fig. 1.14, p. 41), and its value (which is the neuron output) is determined as the alternating sequence 1, -1, 1, -1. Let us number sectors in the natural order (0, 1, 2, 3). If a complex weighted sum is located in an even sector, then the activation function is equal to 1, while if the weighted sum is located in an odd sector, then the activation function is equal to -1. In fact, the activation function (1.40) is periodic. Its period is 2, and its two values 1 and -1 are repeated two times each. In his Ph.D. dissertation, the author of this book also suggested the following generalization of the activation function (1.40), which was explicitly presented in [30].

Let us have a neuron with n binary inputs taken from the set $E_2=\{1,-1\}$. Let weights of this neuron are arbitrary complex numbers $w_i\in\mathbb{C}; i=0,1,...,n$. Thus, the weighted sum $z=w_0+w_1x_1+...+w_nx_n$ is also a complex number. Let us choose some even positive integer $m=2l$ where $l\ge n$. Let us consider now the following l-multiple activation function, which was suggested in [30]

$$\begin{aligned}&P_B(z)=(-1)^j, \text{if } \ 2\pi j/m\le\arg(z)<2\pi(j+1)/m;\\ &m=2l, l\ge n,\end{aligned} \tag{5.156}$$

where j is a non-negative integer $0\le j<m$.

The activation function (5.156) is illustrated in Fig. 5.48b. It divides the complex plane into m=2l equal sectors. It determines the neuron output by the alternating periodic sequence of 1, -1, 1, -1,..., depending on the parity of the sector's number. The activation function (5.156) is equal to 1 for the complex numbers located in the even sectors 0, 2, 4, ..., m-2 and to -1 for the complex numbers located in the odd sectors 1, 3, 5, ..., m-1. Similarly to the MVN activation function (2.50), function (5.156) also depends only on the argument of the weighted sum and does not depend on its magnitude. The activation function (5.156) is a periodic and l-multiple continuation of the activation function (2.52) or, which is the same, of the discrete MVN activation function (2.50) for k=2. This is clearly illustrated in Fig. 5.48. This periodicity of the activation function (5.156) is its main property. It is also easy to check, that when $l=2, m=4$ in (5.156), we obtain (1.40).

A neuron with the activation function (5.156) was called in [30] the universal logical element over the field of complex numbers. A bit later, it was suggested to call it the *universal binary neuron* (UBN) [87]. Its universality is determined by its ability to learn and implement non-linearly separable input/output mappings (along with the linearly separable ones). Let us illustrate this by the following example.

5.1.2 Implementation of the Parity n Function Using a Single Neuron

We have already considered (see p. 41) how a single UBN with the activation function (1.40) (which is the same as (4.156) with $l=2, m=4$) implements the XOR function. Let us consider now how a single UBN with the activation function (5.156) with $l=3, m=6$ (see Fig. 5.49) implements the Parity 3 function.

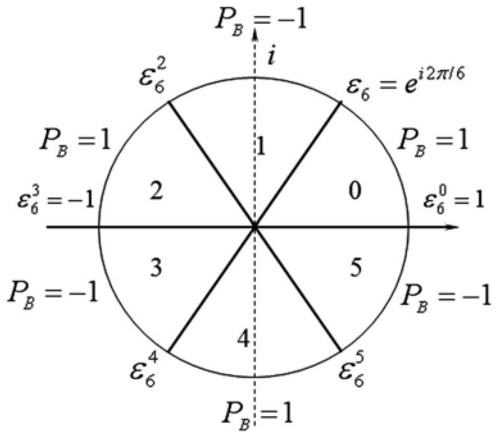


Fig. 5.49 Activation function (156) with l=3, m=6

Table 5.15 Solution of the Parity 3 problem using a single UBN with the activation function (5.156) with l=2, m=6 and with the weighting vector $W=(0,\varepsilon_6,1,1)$

x_1	x_2	x_3	$z=w_0+w_1x_1+$ $+w_2x_2+w_3x_3$	$\arg z$	Number of sector	$P_B(z)$	$f(x_1,x_2,x_3)=$ $=x_1\oplus x_2\oplus x_3$
1	1	1	ε_6+2	0.335	0	1	1
1	1	-1	ε_6	$\pi/3$	1	-1	-1
1	-1	1	ε_6	$\pi/3$	1	-1	-1
1	-1	-1	ε_6-2	2.618= $5\pi/6$	2	1	1
-1	1	1	$-\varepsilon_6+2$	$11\pi/6$	5	-1	-1
-1	1	-1	$-\varepsilon_6=\varepsilon_6^4$	$4\pi/3$	4	1	1
-1	-1	1	$-\varepsilon_6=\varepsilon_6^4$	$4\pi/3$	4	1	1
-1	-1	-1	$-\varepsilon_6-2$	3.475	3	-1	-1

While the XOR function is a mod 2 addition of two Boolean variables, the Parity *n* function is a mod 2 addition of *n* Boolean variables. The Parity *n* function is a non-linearly separable Boolean function for any *n*. Let us consider a single UBN with the activation function (5.156) with $l=3, m=6$. It is easy to check that the weighting vector $W=(0,\varepsilon_6,1,1)$ (where $\varepsilon_6=e^{i2\pi/6}$ is the primitive 6th root of a unity) implements the Parity 3 function $f(x_1,x_2,x_3)=x_1\oplus x_2\oplus x_3$. This is illustrated in Table 5.15.

In [88], it was experimentally shown by the author of this book that a single UBN easily solves the Parity *n* problem up to *n*=14. This will be considered in detail in Section 5.3.5. The ability of a single UBN, a neuron with complex-valued weights, to implement non-linearly separable input/output mappings one more time shows that the functionality of a single neuron with complex-valued weights is higher than the functionality of real-valued neurons.

5.1.3 Projection of a Two-Valued Non-linearly Separable Function into an m-Valued Threshold Function

Let us now consider in detail that mechanism, which makes it possible implementation of non-linearly separable input/output mappings by a single UBN. The following theorem is very important.

Theorem 5.20. If the input/output mapping $f(x_1,...,x_n):E_2^n\to E_2$ can be implemented using a single UBN with the activation function (5.156) and the weighting vector $W=(w_0,w_1,...,w_n)$, then there exist a partially defined *m*-valued threshold function $\tilde{f}(x_1,...,x_n):E_2^n\to E_m$, which can be implemented using a single discrete MVN with the activation function (2.50) (where *k*=*m*) and the same weighting vector $W=(w_0,w_1,...,w_n)$ as the function *f*.

Proof. Since a single UBN implements the input/output mapping $f(x_1,...,x_n):E_2^n\to E_2$ with the weighting vector $W=(w_0,w_1,...,w_n)$, then $\forall(x_1,...,x_n)\in E_2$ $w_0+w_1x_1+...+w_nx_n=z$ such that $P_B(z)=f(x_1,...,x_n)$. This means that if $f(x_1,...,x_n)=1$, then *z* is located in one of the "even" sectors (0, 2, …, *m*-2) in which the activation function (5.156) divides the complex plane (see Fig. 5.48b and Fig. 5.49). If $f(x_1,...,x_n)=-1$, then *z* is located in one of the "odd" sectors (1, 3, …, *m*-1) in which the activation function (5.156) divides the complex plane (see again Fig. 5.48b and Fig. 5.49). This means that the number of a sector where the weighted sum *z* can be located, belongs to the set $M=\{0,1,...,m-1\}$.

Let us apply the discrete MVN activation function (2.50) to z. Then we obtain $P(z)=\varepsilon_m^j$, ($j\in M$ is the number of the sector on the complex plane where z is located). Let us build a partially defined m-valued function $\tilde{f}\left(x_1,...,x_n\right): E_2^n \to E_m$ in the following way (it is partially defined because $E_2^n \subset E_m^n$, thus, it is defined only on the binary inputs). Let us set

$$\tilde{f}\left(x_1,...,x_m\right)=P\left(z\right)=P\left(w_0+w_1x_1+...+w_nx_n\right)=\varepsilon_m^j \in E_m;$$
$$j\in\{0,1,...,m-1\}$$

From the composition of the function $\tilde{f}$ it is clear that it is an m-valued threshold function with the weighting vector $W=\left(w_0,w_1,...,w_n\right)$ according to Definition 2.5. Theorem is proven.

On the one hand, the function $\tilde{f}\left(x_1,...,x_n\right): E_2^n \to E_m$ is a partially defined m-valued function because $E_2^n \subset E_m^n$ (its domain is a subset of E_m^n). On the other hand, Theorem 5.20 can easily be generalized for any function $f: T \to E_2$, where $T \subset O^n$, and O is the set of points located on the unit circle. This generalization leads us to the following statement.

If the input/output mapping $f\left(x_1,...,x_n\right): T \to E_2$ (where $T \subset O^n$) can be implemented using a single UBN with the activation function (5.156) and the weighting vector $W=\left(w_0,w_1,...,w_n\right)$, then there exist a partially defined m-valued threshold function $\tilde{f}\left(x_1,...,x_n\right): T \to E_m$, which can be implemented using a single discrete MVN with the activation function (2.50) (where k=m) and the same weighting vector $W=\left(w_0,w_1,...,w_n\right)$ as the function f.

Theorem 5.20 and its generalization establish the mechanism that projects a two-valued function $f\left(x_1,...,x_n\right)$, which can be implemented using a single UBN, into an m-valued threshold function $\tilde{f}\left(x_1,...,x_n\right)$. The most important here is the following.

If the two-valued function $f\left(x_1,...,x_n\right)$ is a non-linearly separable function in the real domain and cannot be implemented using a single real-valued neuron, but can be implemented using a single UBN, then its projection $\tilde{f}\left(x_1,...,x_n\right)$, is an m-valued threshold function, which can be learned using a single MVN.

For example, the Parity 3 function $f(x_1,x_2,x_3)=x_1\oplus x_2\oplus x_3$ is non-linearly separable in the real domain. It follows from Theorem 5.20 that there exists its projection $\tilde{f}(x_1,x_2,x_3)$, which is built using the weighting vector $W=(0,\varepsilon_6,1,1)$, as it is shown in Table 5.16.

Table 5.16 Projection of the non-linearly separable Parity 3 function into 6-valued multiple-valued threshold function using a single UBN with the activation function (5.156) with *l*=2, *m*=6 and with the weighting vector $W=(0,\varepsilon_6,1,1)$

x_1	x_2	x_3	$z=w_0+w_1x_1+$ $+w_2x_2+w_3x_3$	$\arg z$	$P_B(z)$	$f(x_1,x_2,x_3)=$ $=x_1\oplus x_2\oplus x_3$	$\tilde{f}(x_1,x_2,x_3)$
1	1	1	ε_6+2	0.335	1	1	ε_6^0
1	1	-1	ε_6	$\pi/3$	-1	-1	ε_6
1	-1	1	ε_6	$\pi/3$	-1	-1	ε_6
1	-1	-1	ε_6-2	2.618= $5\pi/6$	1	1	ε_6^2
-1	1	1	$-\varepsilon_6+2$	$11\pi/6$	-1	-1	ε_6^5
-1	1	-1	$-\varepsilon_6=\varepsilon_6^4$	$4\pi/3$	1	1	ε_6^4
-1	-1	1	$-\varepsilon_6=\varepsilon_6^4$	$4\pi/3$	1	1	ε_6^4
-1	-1	-1	$-\varepsilon_6-2$	3.475	-1	-1	ε_6^3

It follows from Table 5.16 that the function $\tilde{f}(x_1,x_2,x_3)$ is a partially defined 6-valued threshold function with the weighting vector $W=(0,\varepsilon_6,1,1)$.

According to Definition 2.9 (see p. 72) it is also a complex-valued threshold function (we can always set $P(0)=\varepsilon_6^0=1$, for example). Then according to Definition 2.10 (see p. 82) such a 6-edge $\bar{Q}=\{Q_0,Q_1,...,Q_5\}$ exists that $\forall\alpha=(\alpha_1,\alpha_2,\alpha_3)\in E_2^3\cap Q_j\;\; P\left(\tilde{f}(\alpha)\right)=\varepsilon_6^j;\, j=0,1,...,5$. The last equation means that this 6-edge separates a 3-dimensional space where the function $\tilde{f}(x_1,x_2,x_3)$ is defined, into six edges (subspaces) $Q_0,Q_1,...,Q_5$, where our function takes the values $\varepsilon_6^0,\varepsilon_6^1,...,\varepsilon_6^5$, respectively. However, the same 6-edge also separates the 1s of the Parity 3 function from its -1s! This is illustrated

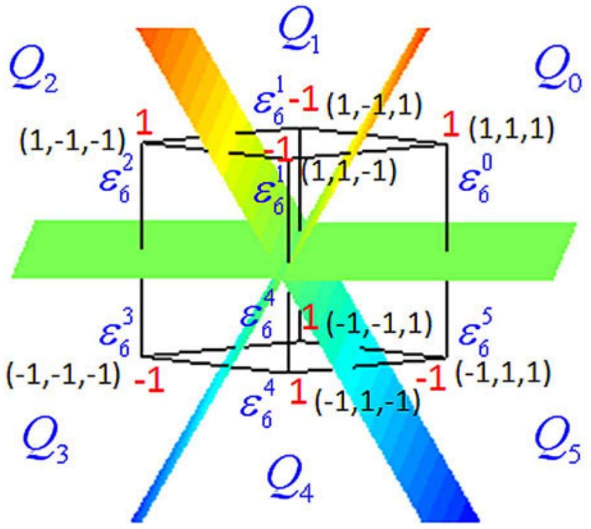


Fig. 5.50 6-edge separates a 3-dimensional space where the Parity 3 function and its 6-valued projection are defined

in Fig. 5.50. The planes, which create the edges of the 6-edge are shown in color. Since the Parity 3 function is defined on the set E_2^3, its 8 values are located in the vertices of the cube, which is also shown in Fig. 5.50. The values of the Parity 3 function are shown in red, while the values of the function $\tilde{f}(x_1, x_2, x_3)$ located in the same cube vertices are shown in blue. The labels $Q_0, Q_1, ..., Q_5$ of the edges of the 6-edge are also shown in blue. As we see, the cube vertices (1, 1, 1) and (1, -1, -1) where the Parity 3 function takes the same value 1 are located in the different edges - Q_0 and Q_2, respectively (the function $\tilde{f}(x_1, x_2, x_3)$ takes there the values ε_6^0 and ε_6^2, respectively). The cube vertices (-1, 1, 1) and (-1, -1, -1) where the Parity 3 function takes the same value -1 are also located in the different edges - Q_5 and Q_3, respectively (the function $\tilde{f}(x_1, x_2, x_3)$ takes there the values ε_6^5 and ε_6^3, respectively). At the same time, the cube vertices (1, -1, 1) and (1, 1, -1) where the Parity 3 function takes the same value -1 are located in the same edge Q_1 where the function $\tilde{f}(x_1, x_2, x_3)$ takes the value ε_6^1. The cube vertices (-1, -1, 1) and (-1, 1, -1) where the Parity 3 function takes the same value 1 are also located in the same edge Q_4 where the function $\tilde{f}(x_1, x_2, x_3)$ takes the value ε_6^4.

While in the real domain the Parity 3 function is not linearly separable and cannot be implemented using a single real-valued neuron, it becomes linearly separable in the complex domain. This separation is utilized by the 6-edge. As we have seen, the Parity 3 function can be implemented using a single UBN. We have also seen that this implementation is equivalent to the implementation of the partially defined 6-valued threshold function $\tilde{f}(x_1, x_2, x_3)$ using a single MVN.

5.1.4 UBN Learning

According to Theorem 5.20, if a Boolean function $E_2^n \rightarrow E_2$ can be implemented using a single UBN with the activation function (5.156), then there exist a partially defined *m*-valued threshold function $E_2^n \rightarrow E_m$, which can be implemented using a single MVN with the activation function (2.50). Thus the UBN learning can be reduced to the MVN learning. This means that the same learning rules (3.81) or (3.92) or (3.94)-(3.98) that are used for the MVN learning can be used for the UBN learning. The only special moment is specification of the desired output for either of these learning rules. If we have to learn any multiple-valued function, the desired output for each element of the learning set is always unambiguous. However, if we have to learn a Boolean function or a mapping like $O \rightarrow E_2$, the desired output in terms of multiple-valued logic is ambiguous. Indeed, the activation function (5.156) divides the complex plane into *m* sectors (see Fig. 5.48b). In a half of them the UBN output is equal to 1, while in another half it is equal to -1. Where we have to direct the weighted sum during the learning process? How we can specify the desired output, to be able to use the MVN learning rules?

It was suggested in [60] by the author of this book, Naum Aizenberg, and Joos Vandewalle to resolve this problem in the following way. The choice of the desired sector *q* in either of (3.81) or (3.92) or (3.94)-(3.98) should be based on the closeness of the current weighted sum to the right or left adjacent sector. Indeed, if the current UBN output is incorrect, this means that it should become corrected if the weighted sum is moved to either left or right adjacent sector. This follows from the construction of the activation function (5.156) (see also Fig. 5.48b). The adjacent sector, which is closer to the current value of the weighted sum in terms of angular distance, is chosen as the "correct" one. The number *q* of this sector determines the desired output in either of the learning rules (3.81) or (3.92) or (3.94)-(3.98).

Hence, the UBN learning algorithm can be described as follows.

Let X_j^t be the *t*th element of the learning set *A* belonging to the learning subset A_j. Let *N* be the cardinality of the set *A*, $|A| = N$. Let $Y_t \in E_2$ and

$D_t \in E_2$ be the actual and the desired UBN outputs, respectively, corresponding to the tth element of the learning set.

Let *Learning* be a flag, which is "True" if the weights adjustment is required and "False", if it not required, and r be the number of the weighting vector in the sequence S_w of weighting vectors obtained during the learning process. Let z_t be the weighted sum corresponding to the tth element of the learning set.

```
Step 1. The starting weighting vector $W_0$ is chosen arbitrarily (e.g., real and
        imaginary parts of its components can be random numbers); $m$=0;
        $t$=1; Learning = "False";
Step 2. Check for $X_j^t$:
        if $Y_t = D_t$
        then go to the step 5
        else begin Learning = "True"; go to Step 3 end;
Step 3. Find $P(z_t) = \varepsilon_m^s$ (where $P$ is the MVN activation function (2.50)).
        Find $q_1 \in M = \{0, 1, ..., m-1\}$, which determines the adjacent sector
        from the right (to the $s$th one), and find $q_2 \in M$, which determines
        the adjacent sector from the left (to the $s$th one) one, where the output
        is correct.
        If
        $\left(\arg z_t - \arg\left(e^{i(q_1+1)2\pi/m}\right)\right) \bmod 2\pi \le \left(\arg\left(e^{iq_2 2\pi/m}\right) - \arg z_t\right) \bmod 2\pi$
        then $q = q_1$
        else $q = q_2$, where $q$ is the number of the desired sector.
Step 4. Obtain the vector $W_{r+1}$ from the vector $W_r$ by setting the desired out-
        put to $\varepsilon_m^q$ and applying either of the learning rules (3.81) or (3.92) or
        (3.94)-(3.98);
Step 5. $t$= $t$+1;   if $t \le N$
                      then go to Step 2
                      else if Learning = "False"
                           then the learning process is finished successfully
                           else begin $t$=1; Learning = "False"; go to Step 2; end.
```

Since this UBN learning algorithm is reduced to the MVN learning algorithm, its convergence directly follows from the convergence of the MVN learning algorithm (see Theorem 3.16 and Theorem 3.17).

5.2 *k*-Valued MVN with a Periodic Activation Function

5.2.1 Some Important Fundamentals

We have just considered UBN – the universal binary neuron. We have shown that UBN is nothing else than the discrete MVN with the activation function (2.50) with *k*=2 (or simply with the activation function (2.52), which is (2.50) for *k*=2), periodically extended. This periodic extension transforms (2.52) to the binary periodic activation function (5.156). While the activation function (2.52) divides the complex plane into two sectors (top and bottom half-planes), the periodic activation function (5.156) divides the complex plane into *m*=2*l* equal sectors. In this case, the neuron output is determined by the alternating periodic sequence of 1, -1, 1, -1,…, depending on the parity of the ordinal sector's number.

As we have seen, this approach leads to one very important advantage. A single UBN may implement those input/output mappings that are non-linearly separable in the real domain. Perhaps, the most convincible examples, which illustrate this advantage of UBN, are XOR and Parity *n* that can easily be learned by a single UBN, without any network. Actually, this is achieved by the projection of 2-valued logic, where the initial non-linearly separable problem is defined, to *m*-valued logic. While in 2-valued logic our input/output mapping is not linearly-separable, in *m*-valued logic it becomes linearly separable.

A natural question is whether it is possible to generalize this approach for multiple-valued input/output mappings that is for the activation function (2.50) with $k > 2$? In other words, if there is some *k*-valued input/output mapping $T \rightarrow E_k$ (where $T = E_k^n$ or $T \subseteq O^n$), which is not a *k*-valued threshold function (and therefore it cannot be learned using a single MVN with the *k*-valued activation function (2.50)), can the same input/output mapping be a partially defined *m*-valued threshold function $T \rightarrow E_m$ for $m > k$?

This question is very important because there is a great practical sense behind it. Suppose we have to solve some *n*-dimensional *k*-class classification problem and the corresponding classes are non-linearly separable. The commonly used approach for solving such a problem, as we already know from this book, is its consideration in the larger dimensional space. One of the ways to utilize this approach is a neural network, where hidden neurons form a new space, and a problem becomes linearly separable. Another popular machine learning approach to solving non-linearly separable problems projecting them into a higher dimensional space is the support vector machine (SVM) introduced in [25]. In SVM, a larger dimensional space is formed using the kernels and a problem becomes linearly separable in this new space. We would like to approach the same problem from a different angle, that is, to consider an *n*-dimensional *k*-class classification problem as an *n*-dimensional *m*-class classification problem (where $m > k$ and each of *k* initial classes is a union of some of *t* disjoint subclasses (clusters) of an initial class):

$$C_j = \bigcup_{i=1}^{t_j} \tilde{C}_i^j, j = 1,...,k; 1 \le t_j < m; \tilde{C}_t^j \cap \tilde{C}_s^j = \varnothing, t \ne s,$$

where $C_j, j = 1,...,k$ is an initial class and each $\tilde{C}_i^j, i = 1,...,m$ is a new subclass). Thus, we would like to modify the formation of a decision rule instead of increasing the dimensionality. In terms of neurons and neural networks this means increasing the functionality of a single neuron by modification of its activation function.

Recently, this problem was comprehensively considered by the author of this book in his paper [61]. Let us present these considerations here adding more details.

5.2.2 Periodic Activation Function for Discrete MVN

Let us consider an MVN input/output mapping described by some *k*-valued function

$$f(x_1,...,x_n): T \to E_k$$

where $T = E_k^n$ or $T \subseteq O^n$).

It is important to mention that since there exists a one-to-one correspondence between the sets $K = \{0,1,...,k-1\}$ and $E_k = \{\varepsilon_k^0, \varepsilon_k, ..., \varepsilon_k^{k-1}\}$ (see p. 49), our function *f* can also be easily re-defined as $f_K : T \to K$. These both definitions are equivalent.

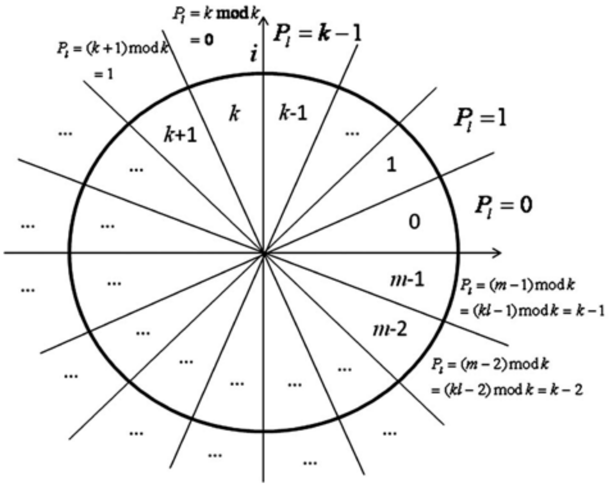
Suppose that the function $f(x_1,...,x_n)$ is not a *k*-valued threshold function. This means that it cannot be learned by a single MVN with the activation function (2.50).

Let us now project the *k*-valued function $f(x_1,...,x_n)$ into *m*-valued logic, where $m = kl$, and $l \ge 2$ similarly to what we have done in Section 5.1 for 2-valued functions projecting them into *m*-valued logic where we used $m = 2l$. To do this, let us define the following new discrete activation function for MVN

$$\begin{aligned} &P_l(z) = j \bmod k, \text{ if } 2\pi j / m \le \arg z < 2\pi (j+1)/m, \\ &j = 0,1,...,m-1; \; m = kl, l \ge 2. \end{aligned} \tag{5.157}$$

This definition is illustrated in Fig. 5.51. The activation function (5.157) divides the complex plane into *m* equal sectors and $\forall d \in K$ there are exactly *l* sectors, in which the activation function (5.157) equals d.

This means that the activation function (5.157) establishes mappings from K into $M=\{0,1,...,k-1,k,k+1,...,m-1\}$, and from E_k into $E_m=\{1,\varepsilon_m,\varepsilon_m^2,...,\varepsilon_m^{m-1}\}$, respectively.



$P_l = k \bmod k = 0$
$P_l = (k+1) \bmod k = 1$
i
$P_l = k-1$
$P_l = 1$
$P_l = 0$
k
k-1
k+1
1
0
m-1
m-2
$P_l = (m-1) \bmod k = (kl-1) \bmod k = k-1$
$P_l = (m-2) \bmod k = (kl-2) \bmod k = k-2$

Fig. 5.51 Geometrical interpretation of the *k*-periodic *l*- multiple discrete-valued MVN activation function (5.157)

Since $m=kl$, then each element from M and E_m has exactly l prototypes in K and E_k, respectively. In turn, this means that the neuron's output determined by (5.157) is equal to

$$\underbrace{\underbrace{0,1,...,k-1}_{0},\underbrace{0,1,...,k-1}_{1},...,\underbrace{0,1,...,k-1}_{l-1},}_{lk=m} \tag{5.158}$$

depending on which one of the *m* sectors (whose ordinal numbers are determined by the elements of the set M) the weighted sum is located in.

Hence, the MVN activation function in this case becomes *k*-periodic and *l*-multiple.

In terms of multiple-valued logic, the activation function (5.157) projects a *k*-valued function $f(x_1,...,x_n)$ into an *m*-valued function $\tilde{f}(x_1, ..., x_n)$. Evidently, $\tilde{f}(x_1,...,x_n)$ is a partially defined function in *m*-valued logic because $K\subset M, E_k\subset E_m$, and $E_k^n\subset E_m^n$.

If $f(x_1,...,x_n): T \to E_k$ is not a k-valued threshold function, then its domain does not allow the *edged decomposition* (see Section 2.3) $T = [C_0, C_1, ..., C_{k-1}]$. Projecting $f(x_1,...,x_n)$ into $kl = m$-valued logic using the activation function (5.157), we create in this m-valued logic the function $\tilde{f}(x_1,...,x_n): T \to E_m; T \subseteq E_k^n \vee T \subseteq O^n$ whose domain may have the edged decomposition $T = \left[\tilde{C}_0, \tilde{C}_1, ..., \tilde{C}_{k-1}, \tilde{C}_k, \tilde{C}_{k+1}, ..., \tilde{C}_{m-1}\right]$. Moreover, if this edged decomposition exists, it exactly follows from its existence that

$$C_j = \bigcup_{i=1}^{t_j} \tilde{C}_i^j, j = 0,...,k-1; 1 \le t_j < m; \tilde{C}_t^j \cap \tilde{C}_s^j = \varnothing, t \ne s .$$

It is important that both functions $f(x_1,...,x_n)$ and $\tilde{f}(x_1,...,x_n)$ have the same domain T. This means that the initial function $f(x_1,...,x_n)$, not being a k-valued threshold function, is projected to a partially defined m-valued threshold function.

Let us refer the MVN with the activation function (5.157) as the *multi-valued neuron with a periodic activation function* (MVN-P).

The following theorem, which generalizes Theorem 5.20 for k-valued input/output mappings, is proven by the last considerations.

Theorem 5.21. If the input/output mapping $f(x_1,...,x_n): T \to E_k$ can be implemented using a single MVN-P with the activation function (5.157) and the weighting vector $W = (w_0, w_1, ..., w_n)$, then there exist a partially defined m-valued threshold function $\tilde{f}(x_1,...,x_n): T \to E_m$, which can be implemented using a single discrete MVN with the activation function (2.50) (where k=m) and the same weighting vector $W = (w_0, w_1, ..., w_n)$.

It is important to mention that if $l = 1$ in (5.157) then m=k and the activation function (5.157) coincides with the activation function (2.50) accurate within the interpretation of the neuron's output (if the weighted sum is located in the jth sector, then according to (2.50) the neuron's output is equal to $e^{ij2\pi/k} = \varepsilon^j \in E_k$, which is the jth of kth roots of unity, while in (5.157) it is equal to $j \in K$), and MVN-P becomes regular MVN. It is also important to mention that if k=2 in (5.157), then the activation function (5.157) coincides with the UBN activation function (5.156), sequence (5.158) becomes an alternating sequence 1, -1, 1, -1, ..., and MVN-P becomes UBN. Hence, the MVN-P is a neuron, for which both MVN and UBN are its particular cases.

MVN-P may have a great practical sense if $f(x_1,...,x_n)$, being a non-threshold function of *k*-valued logic, could be projected into a partially defined threshold function of *m*-valued logic and therefore it will be possible to learn it using a single MVN-P with the activation function (5.157).

When we told that the activation function (5.157) projects a *k*-valued function $f(x_1,...,x_n)$ into *m*-valued logic, we kept in mind that the weighted sum *z*, on which the activation functions depends, is known. But in turn, the weighted sum is a function of the neuron weights for the fixed inputs $x_1,...,x_n$

$$z = w_0 + w_1x_1 + ... + w_nx_n .$$

This means that to establish the projection determined by the activation function (5.157), we have to find the corresponding weights. To find them, a learning algorithm should be used.

5.2.3 Learning Algorithm for MVN-P

A learning algorithm, which we are going to present here, was recently developed by the author of this book and comprehensively described in the paper [61]. On the one hand, this learning algorithm is based on the modification of the MVN learning algorithm considered above in Section 3.1. On the other hand, this learning algorithm is based on the same idea as the UBN learning algorithm, which we have just presented in Section 5.1. The latter becomes clear when we take into account that UBN is nothing else than a particular case of MVN-P, just for *k*=2.

Let us take the MVN learning algorithm described in Section 3.1 and based on either of the error-correction learning rules (3.92) or (3.94)-(3.96) as the initial point for out MVN-P learning algorithm. Let us adapt this MVN learning algorithm to MVN-P. Thus, we have to modify the standard MVN learning algorithm based on the error-correction rule in such a way that it will work for MVN with the periodic activation function (5.157) (for the MVN-P). Let us assume for simplicity, but without loss of generality that the learning rule (3.92) will be used. It is important to mention that the learning rules (3.94)-(3.96) can also be used because, as we have seen (Theorem 3.17), the convergence of the MVN learning algorithm with the error-correction learning rule does not depend on its modification ((3.92) or (3.94)-(3.96)).

Let the MVN input/output mapping is described by the *k*-valued function $f(x_1,...,x_n)$, which is not a threshold function of *k*-valued logic. Since this function is non-threshold, there is no way to learn it using a single MVN with the activation function (2.50). Let us try to learn $f(x_1,...,x_n)$ in *m*-valued logic using a single MVN-P with the activation function (5.157).

Thus, the expected result of this learning process is the representation of $f(x_1,...,x_n)$ according to (2.51), where the activation function P_l determined by (5.157) substitutes for the activation function P determined by (2.50)

$$f(x_1,...,x_n)=P_l(w_0+w_1x_1+...+w_nx_n) \tag{5.159}$$

To organize this learning process, we will use the same learning rule (3.92).

To determine a desired output in the error-correction learning rule (3.92), we may use the same approach, which we used for the UBN learning algorithm in Section 5.1.

So, the learning rule (3.92) requires that a desired neuron output is predetermined. Unlike the case of regular MVN with the activation function (2.50), a desired output in terms of m-valued logic cannot be determined unambiguously for MVN-P with the activation function (3.157) for $l \geq 2$. According to (3.157), there are exactly l sectors out of m on the complex plane, where this activation function is equal to the given desired output $d \in K$ (see (5.158) and Fig. 5.51). Therefore, there are exactly l out of m mth roots of unity that can be used as the desired outputs in the learning rule (3.92). Which one of them should we choose?

Let us make this choice using two self-adaptive learning strategies, which will make it possible to determine a desired output during the learning process every time, when the neuron's output is incorrect.

The first strategy is based on the same idea, which was used for UBN. Since MVN-P is a generalization of UBN for $k>2$, we suggest using here the same learning strategy that was used in the UBN error-correction learning algorithm in Section 5.1. There is the following idea behind this approach. The UBN activation function (5.156) determines an alterning sequence 1, -1, 1, -1, … with respect to sectors on the complex plane. Hence, if the actual output of UBN is not correct, in order to make the correction, we can "move" the weighted sum into either of the sectors adjacent to the one where the current weighted sum is located. It was suggested to always move it to the sector, which is the closest one to the current weighted sum (in terms of angular distance).

Let us employ the same approach here for MVN-P. Let $l \geq 2$ in (5.157) and $d \in \{0,1,...,k-1\}$ be the desired output. The activation function (5.157) determines the k-periodic and l-multiple sequence (5.158) with respect to sectors on the complex plane. Suppose that the current MVN-P output is not correct and the current weighted sum is located in the sector $s \in M = \{0,1,...,m-1\}$, where $m=kl$.

Since $l \geq 2$ in (5.157), there are exactly l sectors on the complex plane, where function (5.157) takes a correct value (see also Fig. 5.51). Two of these l sectors are the closest ones to sector s (from right and left sides, respectively). From these two sectors, we choose sector q whose border is closer to the current weighted sum z in terms of the angular distance. Then the learning rule (3.92) can be applied. Hence, the first learning strategy for the MVN-P with the activation function (5.157) is as follows. Let a learning set for the function (input/output mapping) $f\left(x_1,...,x_n\right)$ to be learned contains N learning samples and $j \in \{1,...,N\}$ be the number of the current learning sample, *r* be the number of the learning iteration, and *Learning* is a flag, which is "True" if the weights adjustment is required and "False" otherwise.

The iterative learning process consists of the following steps:

```
Learning Strategy 1.
1) Set r=1, j=1, and Learning='False'.
2) Check (5.159) for the learning sample j.
3) If (5.159) holds
   then set $j = j+1$, otherwise set Learning='True' and
   go to Step 5.
4) If $j \le N$ then go to Step 2, otherwise go to Step 9.
5) Let z be the current value of the weighted sum and $P(z) = \varepsilon_m^s, s \in M$,
   $P(z)$ is the activation function (2.50), where m is substituted for k. Hence
   the MVN-P actual output is $P_l(z) = s \bmod k$. Find
   $q_1 \in M = \{0,1,...,m-1\}$, which determines the closest sector to the sth
   one, where the output is correct, from the right, and find $q_2 \in M$, which de-
   termines the closest sector to the sth one, where the output is correct, from
   the left (this means that $q_1 \bmod k = d$ and $q_2 \bmod k = d$).
6) If $\left(\arg z - \arg\left(e^{i(q_1+1)2\pi/m}\right)\right) \bmod 2\pi \le \left(\arg\left(e^{iq_2 2\pi/m}\right) - \arg z\right) \bmod 2\pi$
   then $q = q_1$
   else $q = q_2$.
7) Set the desired output for the learning rule (3.92) equal $\varepsilon_m^q$.
8) Apply the learning rule (3.92) to adjust the weights.
9) Set $j = j+1$ and return to Step 4.
10) If Learning='False'
   then go to Step 10,
   else set r=r+1, j=1, Learning='False' and go to Step 2.
11) End.
```

Let us now consider the second learning strategy, which is somewhat different. The activation function (5.157) divides the complex plane into l domains, and each of them consists of *k* sectors (Fig. 5.51). Since a function *f* to be learned as a partially defined function of *m*-valued logic ($m = lk$) is in fact a *k*-valued function, then each of *l* domains contains those *k* values, which may be used as the desired outputs of the MVN-P. Suppose that the the current MVN-P output is not correct, and the current weighted sum is located in the sector $s \in M = \{0,1,...,m-1\}$. This sector in turn is located in the *t*th *l*-domain (out of *l*, $t = [s/k]$). Since there are *l* *l*-domains and each of them contains a potential correct output, we have *l* options to choose the desired output. Let us choose it in the same *t*th *l*-domain, where the current actual output is located. Hence, $q = tk + f_K(x_1,...,x_n)$, where $f_K(x_1,...,x_n)$ is a desired value of the function to be learned in terms of traditional multiple-valued logic ($f_K(x_1,...,x_n) \in K = \{0,1,...,k-1\}$ and respectively, $f(x_1,...,x_n) \in E_k = \{1, \varepsilon_k, \varepsilon_k^2, ..., \varepsilon_k^{k-1}\}$). Once q is determined, this means that ε_m^q be the desired output and the learning rule (3.92) can be applied.

Let again a learning set for the function (input/output mapping) $f(x_1,...,x_n)$ to be learned contains *N* learning samples, $j \in \{1,...,N\}$ be the number of the current learning sample, *r* be the number of the learning iteration, and *Learning* is a flag, which is "True" if the weights adjustment is required and "False" otherwise. The iterative learning process for the second strategy consists of the following steps:

Learning Strategy 2.

```
1) Set $r=1, j=1$, and Learning='False'.
2) Check (5.159) for the learning sample $j$.
3) If (5.159) holds
   then set $j = j+1$,
   else set Learning ='True' and go to Step 5.
4) If $j \le N$
   then go to Step 2,
   else go to Step 8.
5) Let the actual neuron output is located in the sector
   $s \in M = \{0,1,...,m-1\}$. Then $t = [s/k] \in \{0,1,...,l-1\}$ is the number
   of that l-domain, where sector s is located. Set $q = tk + f_K(x_1,...,x_n)$.
6) Apply the learning rule (3.92) to adjust the weights.
7) Set $j = j+1$ and return to Step 4.
8) If Learning='False'
   then go to Step 9,
   else set $r=r+1, j=1$, Learning='False' and go to Step 2.
9) End.
```

The learning strategies 1 and 2 determine two variants of the same MVN-P learning algorithm, which can be based on either of the learning rules (3.92), (3.94)-(3.98). The convergence of this learning algorithm follows from the convergence of the regular MVN learning algorithm with the error-correction learning rule (see Theorem 3.17). Indeed, if our input/output mapping $f\left(x_1,...,x_n\right)$ is a non-threshold function of *k*-valued logic, but it can be projected to a partially defined threshold function $\tilde{f}\left(x_1,...,x_n\right)$ of *m*-valued logic (where $m = kl, l \geq 2$), then the MVN learning algorithm has to converge for the last function according to Theorem 3.17. The MVN-P learning algorithm based on the either of learning rules (3.92), (3.94)-(3.96) differs from the MVN learning algorithm only at one point. While for the regular MVN learning a desired output is pre-determined, for the MVN-P learning a desired output in terms of *m*-valued logic should be determined during the learning process. If the function $\tilde{f}\left(x_1,...,x_n\right)$ obtained using either of Learning Strategies 1 or 2 is a partially defined *m*-valued threshold function, its learning has to converge to a weighting vector of this function (a weighting vector of this function can always be obtained after a finite number of learning iterations).

Thus, in other words, if a non-threshold *k*-valued function $f\left(x_1,...,x_n\right)$ can be projected to and associated with a partially defined *m*-valued threshold function $\tilde{f}\left(x_1,...,x_n\right)$, then its learning by a single MVN-P is reduced to the learning of the function $\tilde{f}\left(x_1,...,x_n\right)$ by a single MVN.

We have to mention that we do not consider here any general mechanism of such a projection of a *k*-valued function into *m*-valued logic that the resulting *m*-valued function will be threshold and therefore it will be possible to learn it by a single neuron. It is a separate problem, which is still open and can be a good and interesting subject for the further work.

It is interesting that in terms of learning a *k*-valued function, the learning algorithm presented here is supervised. However, in terms of learning an *m*-valued function, this learning algorithm is unsupervised. We do not have a prior knowledge about those *m*-valued output values, which will be assigned to the input samples. A process of this assignment is self-adaptive, and this adaptation is reached by the learning procedure (Strategies 1 and 2), if a corresponding function is a partially defined *m*-valued threshold function.

It should be mentioned that for *k*=2 in (5.157) the MVN-P learning algorithm (Strategy 1) coincides with the UBN learning algorithm based on the error-correction rule (see Section 5.1). On the other hand, for *k*>2 and *l*=1 in (5.157) the MVN-P learning algorithm (both Strategy 1 and Strategy 2) coincides with the MVN learning algorithm based on the error-correction rule (see Sections 3.1 and 3.3).

This means that a concept of the MVN-P generalizes and includes the corresponding MVN and UBN concepts.

5.3 Simulation Results for *k*-Valued MVN with a Periodic Activation Function

As it was shown above, MVN-P can learn input/output mappings that are non-linearly separable in the real domain. We would like to consider here a number of non-linearly separable benchmark classification problems and a non-linearly separable $\operatorname{mod} k$ addition problem, which can be learned using a single MVN-P. Moreover, we would like to show that a single MVN-P not only formally learns non-linearly separable problems, but it can really be successfully used for solving non-linearly separable classification problems, showing very good results that are better or comparable with the solutions obtained using neural networks or support vector machines. However, it is very important to mention that MVN-P is just a single neuron, and it employs fewer parameters than any network or SVM.

So, let us consider some examples. Most of them were presented by the author of this book in his recently published paper [61], some of them will be presented here for the first time, but even those published earlier will be presented here in more detail. In all simulations, we used the MVN-P software simulator written in Borland Delphi 5.0 environment, running on a PC with the Intel® Core™2 Duo CPU.

5.3.1 Iris

This famous benchmark database was downloaded from the UC Irvine Machine Learning Repository [89]. The data set contains 3 classes of 50 instances each, where each class refers to a type of iris plant. Four real-valued (continuous) features are used to describe the data instances. Thus, we have here 4-dimensional 3-class classification problem. It is known [89] that the first class is linearly separable from the other two but the latter are not linearly separable from each other. Thus, a regular single MVN with the discrete activation function (2.50), as well as any other single artificial neuron cannot learn this problem completely.

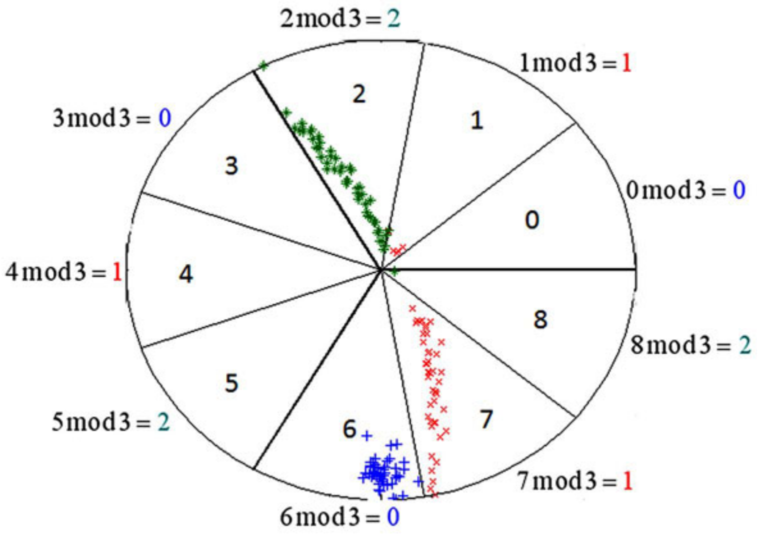
However, a single MVN-P with the activation function (5.157) ($l = 3, k = 3, m = 9$) learns the Iris problem completely with no errors. To transform the input features into the numbers located on the unit circle, we used the linear transformation (2.53) with $\alpha = 2\pi / 3$ (this choice of α is based on the consideration that there are exactly 3 classes in this problem). It is necessary to say that the problem is really complicated and it is not so easy to learn it. For example, the learning algorithm based on the Learning Strategy 1 does not converge even after 55,000,000 iterations independently from the learning rule, which is applied, although the error decreases very quickly and after 50-100 iterations there are stably not more than 1-7 samples, which still require the weights adjustment. However, the learning algorithm based on the Learning Strategy 2 and the learning rule (3.96) converges with the zero error. Seven independent runs of the learning algorithm starting from the different random weights[1] converged after 9,379,027 – 43,878,728 iterations. Every time the error decreases very quickly and after 50-100 iterations there are stably 1 or just a few more samples, which still require the weights adjustment, but their final adjustment takes time (5-12 hours). Nevertheless, this result is very interesting, because to our best knowledge this is the first time when the "Iris" problem was learned using just a single neuron.

The results of the one of the learning sessions are shown in Fig. 5.52. In Fig. 5.52a, the normalized weighted sums are plotted for all the 150 samples from the data set. It is interesting that after the learning process converges, for the Class "0" (known and referred to as "Iris Setosa" [89]), the weighted sums for all instances appear in the same single sector on the complex plane (sector 6, see Fig. 5.52a). By the way, according to the activation function (5.157), the MVN-P output for all the samples from this class is equal to $6 \bmod 3 = 0$. Thus, Class "0" is a single cluster class.

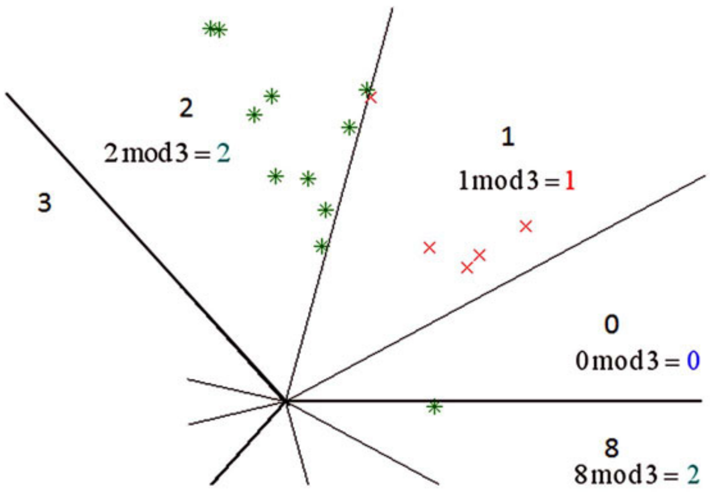
Each of two other classes contains two different clusters (this is why they cannot be linearly separated from each other in the real domain!). For the second class ("Iris Versicolour"), 45 out of 50 learning samples appear in the sector 7, but other 5 learning samples appear in the sector 1 located in the different "*l*-domain" (cluster) (see Fig. 5.52a). According to the activation function (5.157), the MVN-P output for all the samples from this class is equal to 1 ($7 \bmod 3 = 1$ and $1 \bmod 3 = 1$). For the third class ("Iris Virginica"), the weighted sums for all the instances except one appear in the same single sector on the complex plane (sector 2), but for the one instance (every time the same) the weighted sum appears in the different sector (sector 8) belonging to the different "*l*-domain" (cluster). According to the activation function (5.157), the MVN-P output for all the samples from this class is equal to 2 ($2 \bmod 3 = 2$ and $8 \bmod 3 = 2$). For the reader's convenience, a fragment showing where exactly five "special" elements from the Class 1 and one "special" element from the Class 2 are located is enlarged in Fig. 5.52b. Hence, the second and the third classes, which initially are known as non-linearly separable (in the real domain), become linearly separable in the complex domain. This means that while there is no 3-edged decomposition $T = [C_0, C_1, C_2]$ (where C_0, C_1, C_2 are our three classes) for the Iris problem, there exists the 9-edged decomposition $T = [A_0, A_1, \dots, A_8]$. Subsets A_0, A_3, A_4, A_5 are empty. Other subsets of the edged decomposition contain all the elements of the Iris dataset as follows (see Fig. 5.52a)

$$C_0 = A_6; C_1 = A_1 \cup A_7; C_2 = A_2 \cup A_8.$$

[1] Here and further the initial weights (both real and imaginary parts) are random numbers from the interval [0, 1] generated using a standard generator.



(a) The results of learning of the "Iris" problem. + - Class 0, x – Class 1, * - Class 2 While Class 0 contains a single cluster (sector 6), Class 1 and Class 2 contain two clusters each (Class 1 - sectors 1 and 7, Class 2 – sectors 2 and 8)



(b) 5 out of 50 representatives of Class 1 belong to the cluster located in the sector1, and a single representative of Class 2 belong to the cluster located in the sector 8

Fig. 5.52 Learning of the "Iris" problem. Three "Iris" classes are linearly separated in 9-valued logic

It is interesting that this effect is achieved by the self adaptation of the MVN-P learning algorithm.

Another important experiment with the "Iris" data set was checking the MVN-P's ability to solve a classification problem. We used 5-fold cross validation. The data set was every time randomly separated into a learning set containing 75 samples (25 from each class) and a testing set also containing 75 samples. The best results are obtained for the activation function (5.157) with $l = 2, k = 3, m = 6$. The Learning Strategy 1 and the learning rule (3.92) were used. The learning algorithm requires for its convergence with the zero error 10-288 iterations (which takes just a few seconds). The classification results are absolutely stable: 73 out of 75 instances are classified correctly (the classification rate is 97.33%). All instances from the first class are always classified correctly and there is one classification error in each of other two classes. These results practically coincide with the best known results for this benchmark data set [90]: (97.33 for the one-against-one SVM and 97.62 for the dendogram-based SVM). However, it is important to mention that the one-against-one SVM for 3 classes contains 3 binary decision SVMs, the dendogram-based SVM for 3 classes contains 5 binary decision SVMs, while we solved the Iris problem using just a single MVN-P.

5.3.2 Two Spirals

The two spirals problem is a well known non-linearly separable classification problem, where the two spirals points (see Fig. 5.53) must be classified as belonging to the 1st or to the 2nd spiral. Thus, this is 2-dimensional, 2-class classification problem. The standard two spirals data set usually consists of 194 points (97 belong to the 1st spiral and other 97 points belong to the 2nd spiral). The following results are known as the best for this problem so far. The two spirals problem can be learned completely with no errors by the MLMVN [62] containing 30 hidden neurons in a single hidden layer and a single output neuron. This learning process requires about 800,000 iterations. The best known result for the standard backpropagation network (MLF) with the same

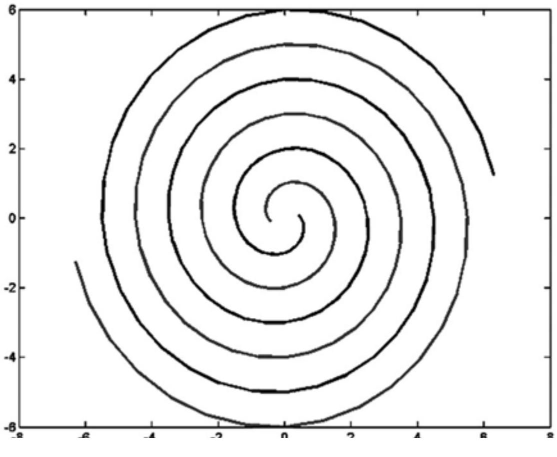


Fig. 5.53 Two spirals

topology is 14% errors after 150,000 learning iterations [91]. For the cross-validation testing, where each second point of each spiral goes to the learning set and each other second point goes to the testing set, one of the best known results is reported in [92]. The classification accuracy up to 94.2% is shown there by BDKSVM, which employs along with a traditional SVM the merits of the kNN classifier. A fuzzy kernel perceptron shows the accuracy up to 74.5%. [93]. The MLMVN shows the accuracy of about 70% [62].

A single MVN-P with the activation function (5.157) $(l=2, k=2, m=4)$ significantly outperforms all mentioned techniques. Just 2-3 learning iterations are required to learn the two spirals problem completely with no errors using the Learning Strategy 1 and learning rule (3.92). Just 3-6 iterations are required to achieve the same result using the Learning Strategy 1 and learning rule (3.94). These results are based on the ten independent runs of the learning algorithm for each of the learning rules. We also used ten independent runs to check the classification ability of a single MVN-P with the activation function (5.157) $(l=2, k=2, m=4)$ using the cross-validation. The two spirals data were divided into the learning set (98 samples) and testing set (96 samples). We reached the absolute success in this testing: 100% classification accuracy is achieved in all our experiments. Just 2-3 iterations were needed to learn the learning set using the Learning Strategy 1 and the learning rule (3.92), and 3-5 iterations were needed to do the same using the Learning Strategy 1 and the learning rule (3.94).

5.3.3 Breast Cancer Wisconsin (Diagnostic)

This famous benchmark database was downloaded from the UC Irvine Machine Learning Repository [89]. The data set contains 2 classes, which are represented by 569 instances (357 benign and 212 malignant) that are described by 30 real-valued features. Thus, this is 30-dimensional, 2-class classification problem. To transform the input features into the numbers located on the unit circle, we used the linear transformation (2.53) with $\alpha = 6.0$. The whole data set may be easily learned by a single MVN-P with the activation function (5.157) $(l=2, k=2, m=4)$. Ten independent runs give from 280 to 370 iterations for the Learning Strategy 1 – learning rule (3.92) and from 380 to 423 iterations for the Learning Strategy 2 – learning rule (3.92) (there are the best results among different combinations of learning strategies and rules).

To check the classification ability of a single MVN-P, we used 10-fold cross-validation as it is recommended for this data set, for example, in [89] and [94]. The entire data set was randomly divided into the 10 subsets, 9 of them contained 57 samples and the last one contained 56 samples. The learning set every time was formed from 9 of 10 subsets and the remaining subset was used as the testing set. We used the same parameters in the activation function (5.157) $(l=2, k=2, m=4)$. The best average classification accuracy (97.68%) was achieved with the Learning Strategy 2 and learning rule (3.92). The learning process required from 74 to 258 iterations. The classification accuracy is comparable with the results reported for SVM in [94] (98.56% for the regular SVM and 99.29% for the SVM with an additional "majority decision" tool) and a bit better than 97.5% reported as the estimated accuracy for the different classification methods in [89]. However, a single MVN-P use fewer parameters than, for example, SVM (the average amount of support vectors for different kernels used in [94] is 54.6, whereas the MVN-P uses 31 weights).

5.3.4 Sonar

This famous benchmark database was also downloaded from the UC Irvine Machine Learning Repository [89]. It contains 208 samples that are described by 60 real-valued features. Thus, this is 30-dimensional, 2-class classification problem, which is non-linearly separable. To transform the input features into the numbers located on the unit circle, we used (2.53) with $\alpha = 6.0$. There are two classes ("mine" and "rock") to which these samples belong. The whole data set may be easily learned by a single MVN-P with the activation function (5.157)$\left(l = 2, k = 2, m = 4\right)$. Ten independent runs give from 75 to 156 iterations for the Learning Strategy 1 – learning rule (3.92) and from 59 to 78 iterations for the Learning Strategy 2 – learning rule (3.92) (there are the best results among different combinations of learning strategies and rules).

To check the classification ability of a single MVN-P, we divided the data set into a learning set and a testing set (104 samples in each), as it is recommended by the developers of this data set [89]. The same parameters were used in the activation function (5.157) $\left(l = 2, k = 2, m = 4\right)$. The best classification results were achieved using the Learning Strategy 2 - learning rule (3.92). The learning process required from 24 to 31 iterations. The average classification accuracy for 10 independent runs is 86.63% and the best achieved accuracy is 91.3%. This is comparable to the best known results reported in [93] – 94% (Fuzzy Kernel Perceptron), 89.5% (SVM), and in [62] - 88%-93% (MLMVN). It is important to mention that all mentioned competitive techniques employ more parameters than a single MVN-P. It is also necessary to take into account that a Fuzzy Kernel Perceptron is much more sophisticated tool than a single neuron.

5.3.5 Parity N Problem

Parity *n* problem is a mod 2 addition of *n* variables. Its particular case for *n*=2 is XOR, perhaps the most popular non-linearly separable problem considered in the literature. We have already convinced that the XOR problem can easily be solved using a single UBN (see Table 1.7, p. 42). We have also seen (see Table 5.15, p. 176) that the Parity 3 problem can be solved using a single UBN. As we have mentioned, it was experimentally shown by the author of this book in [88] that the Parity *n* problem is easily solvable using a single UBN up to *n*=14 (this does not mean that for larger *n* it is not solvable, simply experimental testing was not performed for *n*>14). Let us summarize the results of this experimental testing here.

Actually, as we have seen, UBN is nothing else than MVN-P for *k*=2. So, we used MVN-P with the activation function (5.157) and the learning algorithm with the Learning Strategy 1, which we have just described. The learning rule (3.92) was used in this learning algorithm. The results are summarized in Table 5.17. Everywhere we show the results for such minimal *l* in (5.157), for which the learning process converged after not more than 200,000 iterations.

Table 5.17 The results of solving Parity n problem for $3 \le n \le 14$ using a single MVN-P

Number of variables, *n* in the Parity *n* problem	*l* in (157)	Number of sectors (*m* in (157))	Number of learning iterations (average of 5 independent runs)
3	3	6	8
4	4	8	23
5	5	10	37
6	6	12	52
7	7	14	55
8	8	16	24312
9	11	22	57
10	14	28	428
11	15	30	1383
12	18	36	1525
13	19	38	16975
14	22	44	3098

5.3.6 Mod k Addition of n k-Valued Variables

This problem may be considered as a generalization of the famous and popular Parity *n* problem for the *k*-valued case. In fact, Parity *n* problem is a mod 2 addition of *n* variables. mod *k* addition of *n* variables is a non-threshold *k*-valued function for any *k* and any *n* and therefore it cannot be learned by a single MVN. To our best knowledge there is no evidence that this function can be learned by any other single neuron. However, as we will see now, it is not a problem to learn this problem using a single MVN-P with the activation function (5.157).

We do not have a universal solution of the problem of $\bmod k$ addition of *n* variables in terms of the relationship between k and n on the one side and *l* in (5.157) on the other side. However, we can show here that this multiple-valued problem is really solvable at least for those *k* and *n*, for which we have performed experimental testing [61, 95].

The experimental results are summarized in Table 5.18. Since the Learning Strategy 1 showed better performance for this problem (fewer learning iterations and time), all results are given for this strategy only.

Our goal was to find a minimal l in (5.157), for which the learning process converges. For each combination of k and n on the one hand, and for each learning rule, on the other hand, such a minimal value of l is presented. The average number of iterations for seven independent runs of the learning process and its standard deviation are also presented for each combination of k and n for each learning rule. We considered the learning process non-converging if there was no convergence after 200,000 iterations. The learning error for all learning rules drops very quickly, but fine adjustment of the weights takes more time. For some k and n and for some of learning rules we used a staggered learning technique. This means that unlike a regular learning technique, where all learning samples participate in the learning process from the beginning, the staggered method expands a learning set step by step. For example, let A be a learning set and its cardinality is N. The set A can be represented as a union of non-overlapping subsets $A_1, A_2, ..., A_s$. Then the learning process starts from the subset A_1. Once it converged, it has to continue for the extended learning set $A_1 \bigcup A_2$ starting from the weights that were obtained for A_1. Once it converges, the learning set has to be extended to $A_1 \bigcup A_2 \bigcup A_3$, adding one more subset after the previous learning session converged. Finally, we obtain the learning set $A_1 \bigcup A_2 \bigcup ... \bigcup A_{s-1} \bigcup A_s = A$. This approach is efficient when the function to be learned has a number of high jumps.

If there are exactly s high jumps, then s sequential learning sessions with s expanding learning sets $A_1, A_1 \bigcup A_2, ..., A_1 \bigcup ... \bigcup A_s = A$ lead to faster convergence of the learning algorithm. For example, the function mod k addition of n variables has exactly k^n learning samples. For any k and n, this function has multiple high jumps from k-1 to 0. These jumps can be used to determine partitioning of the corresponding learning set into s non-overlapping subsets $A_1, A_2, ..., A_s$. First, the learning process has to be run for A_1. Once it converges, it has to be run for $A_1 \bigcup A_2$. This set contains one high jump, but since the starting weighting vector, which already works for A_1, better approaches the resulting weighting vector, the learning process for $A_1 \bigcup A_2$ converges better starting from this weighting vector than starting from a random one. Then this process has to be continued up to the learning set $A_1 \bigcup ... \bigcup A_s = A$. We used this staggered learning technique, for example, for $k = 5, n = 2$, for $k = 6, n = 2$ and some other k and n (see the footnote in Table 5.18.). While neither of learning rules (3.92), (3.94)-(3.96) leads to the convergence of the standard learning algorithm after 200,000 iterations for these specific k and n, the staggered technique leads to very quick convergence of the learning process for all four learning rules.

Thus, for all the experiments, we show in Table 5.18 the smallest l in (5.157), for which the convergence was reached.

Table 5.18 Simulation Results for mod k addition of n k-valued variables

k	n	Average number of learning iterations (Iter.) for 7 independent runs, its standard deviation (SD), and minimal value of l in (5.157), for which the learning process converged.											
		Learning rule (3.92)			Learning rule (3.94)			Learning rule (3.95)			Learning rule (3.96)		
		Iter.	SD	l	Iter.	SD	l	Iter.	SD	l	Iter.	SD	l
3	2	14	5	2	18048	44957	2	54	14	2	1005	1384	2
3	3	2466	2268	10	3773	1721	10	2568	1988	10	2862	676	12
3	4	4296	2921	11	391404	158688	7	2002	1272	14	1728	767	11
3	5	78596	87158	18	344440	308044	22	236242	188276	18	23372	8255	24
3	6	237202	172100	36	50292	91260	39	291950	346862	27	41083	23117	30
3	7	518313	395671	41	1556379	798841	41	489366	229706	22	390786	260953	25
4	2	2693	3385	3	135	163	3	94[1]	23	3	660[1]	567	3
4	3	2571	1772	7	12175	5407	7	411	190	7	602	436	7
4	4	50151	35314	10	140850	88118	10	47818	53349	13	3797	2756	13
4	5	734691[1]	231353	13	355910[1]	98208	13	174649[1]	148655	16	209629[1]	189481	15
4	6	131139[1]	185316	42	171269[1]	104685	15	59807[1]	57060	34	306672[1]	312548	30
4	7	108050	30309	39	110809	37286	38	90734	35474	37	95055	39581	38
5	2	96[1]	22	4	81[1]	16	4	82[1]	23	4	197[1]	82	5
5	3	1202[1]	193	9	1419[1]	264	9	1460[1]	308	9	1470[1]	191	9
5	4	4604[1]	393	13	4893[1]	211	13	5606[1]	374	13	6182[1]	616	13
5	5	22812[1]	3977	22	17274[1]	1682	18	17402[1]	3415	18	21470[1]	1959	18
5	6	105672[1]	20071	26	196441[1]	5635	22	66609[1]	9888	33	2693[1]	310	24
5	7	630490	192494	52	557635	305579	49	16192	24618	35	5280	2433	31
6	2	272[1]	110	4	120[1]	33	4	95[1]	35	4	295[1]	65	4
6	3	109733	2400	14	4131	4039	10	861	150	11	834	221	10
6	4	118128	15596	14	48002	11652	14	9615[1]	1159	16	10951[1]	1265	16
6	5	71241	74463	21	15986	14835	21	128550	105115	20	42122	26631	20
6	6	92257[1]	4773	33	194544[1]	203936	26	130405[1]	25104	27	122814[1]	9447	27
6	7	247232[1]	64747	31	262564[1]	13614	31	260654[1]	32977	37	257298[1]	2291	37

1 - staggered learning technique used. This means that a learning set was extended step by step. Initially first k samples were learned, then starting from the obtained weights $2k$ samples were learned, then $3k$, etc. up to k^n samples in a whole learning set

As we have discovered above (Theorem 5.21), if some k-valued input/output mapping is a non-threshold k-valued function, but it can be learned using a single MVN-P, this means that this non-threshold k-valued function is projected to a partially defined $kl = m$-valued function. In practice, this partially-defined m-valued threshold function, which is not known prior to the learning session, is generated by the learning process.

As it follows from our experiments mod k addition of n variables functions for any $k \geq 2$ and any n are projected into partially defined $kl = m$-valued threshold functions, which are so-called minimal monotonic functions.

This means the following. Let $X_i = \left(x_1^i, ..., x_n^i\right)$ and $X_j = \left(x_1^j, ..., x_n^j\right)$. Vector X_i precedes to vector X_j ($X_i \prec X_j$) if $x_s^i \leq x_s^j$, $\forall s = 1, ..., n$. Function $f\left(x_1, ..., x_n\right)$ is called *monotonic* if for any two sets of variables X_i and X_j, such that $X^i \prec X^j$, the following holds

$$f\left(X_i\right) = f\left(x_1^i, ..., x_n^i\right) \leq f\left(X_j\right) = f\left(x_1^j, ..., x_n^j\right).$$

An m-valued function $f\left(x_1, ..., x_n\right)$ is called *minimal monotonic* [88], if it is monotonic and for any two closest comparable sets of variables $X_i = \left(x_1^i, ..., x_n^i\right)$ and $X_j = \left(x_1^j, ..., x_n^j\right)$, if $X_i \prec X_j$, then $f\left(x_1^j, ..., x_n^j\right) - f\left(x_1^i, ..., x_n^i\right) \leq 1$, that is $f\left(x_1^j, ..., x_n^j\right)$ is either equal to $f\left(x_1^i, ..., x_n^i\right)$ or is greater than $f\left(x_1^i, ..., x_n^i\right)$ by exactly 1.

A very interesting experimental fact is that all partially defined m-valued functions, to which mod k additions of n variables were projected by the learning process are minimal monotonic m-valued functions. Let us consider several examples for different k and n (see Table 5.19 - Table 5.24).

Table 5.19 XOR – mod 2 addition of 2 variables, l=2, m=4 in (5.157)

x_1	x_2	$f\left(x_1, x_2\right) =$ $= \left(x_1 + x_2\right) \bmod 2$	$j \in M = \{0,1,2,3\}$ 2x2=4-valued function $\tilde{f}\left(x_1, x_2\right)$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	2

In all these tables, the first columns contain values of the corresponding inputs (input variables) $x_1, ..., x_n$. The second to the last column contains values of the k-valued function $f\left(x_1, ..., x_n\right)$, which we learn, and the last column contains values of that partially defined kl=m-valued function $\tilde{f}\left(x_1, ..., x_n\right)$, to which the initial function was projected by the learning algorithm. For simplicity, we show the values of the input variables and functions f and $\tilde{f}$ in the regular multiple-valued alphabets $K=\{0,1,...,k-1\}$ and $M=\{0,1,...,k,...,m-1\}$. The reader may easily convert these values to such that belong to E_k and E_m, respectively (if $j\in K$, then $e^{i2\pi j/k}\in E_k$).

Table 5.20 Parity 3 – mod 2 addition of 3 variables, *l*=3, *m*=6 in (5.157)

x_1	x_2	x_3	$f(x_1,x_2,x_3)=$ $=(x_1+x_2+x_3)\bmod 2$	$j\in M=\{0,1,...,5\}$ 2x3=6-valued function $\tilde{f}(x_1,x_2,x_3)$
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	2
1	0	0	1	1
1	0	1	0	2
1	1	0	0	2
1	1	1	1	3

Table 5.21 Parity 4 – mod 2 addition of 4 variables, *l*=3, *m*=6 in (5.157)

x_1	x_2	x_3	x_4	$f(x_1,x_2,x_3,x_4)=$ $=(x_1+x_2+x_3+x_4)\bmod 2$	$j\in M=\{0,1,...,7\}$ 2x4=8-valued function $\tilde{f}(x_1,x_2,x_3,x_4)$
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	1	1
0	0	1	1	0	2
0	1	0	0	1	1
0	1	0	1	0	2
0	1	1	0	0	2
0	1	1	1	1	3
1	0	0	0	1	1
1	0	0	1	0	2
1	0	1	0	0	2
1	0	1	1	1	3
1	1	0	0	0	2
1	1	0	1	1	3
1	1	1	0	1	3
1	1	1	1	0	4

Table 5.22 mod 3 addition of 3 variables, *l*=3, *m*=9 in (5.157)

x_1	x_2	x_3	$f(x_1,x_2,x_3)=$ $=(x_1+x_2+x_3)\bmod 3$	$j\in M=\{0,1,...,9\}$ 3x3=9-valued function $\tilde{f}(x_1,x_2,x_3)$
0	0	0	0	21
0	0	1	1	22
0	0	2	2	23
0	1	0	1	22
0	1	1	2	23
0	1	2	0	24
0	2	0	2	23
0	2	1	0	24
0	2	2	1	25
1	0	0	1	22
1	0	1	2	23
1	0	2	0	24
1	1	0	2	23
1	1	1	0	24
1	1	2	1	25
1	2	0	0	24
1	2	1	1	25
1	2	2	2	26
2	0	0	2	23
2	0	1	0	24
2	0	2	1	25
2	1	0	0	24
2	1	1	1	25
2	1	2	2	26
2	2	0	1	25
2	2	1	2	26
2	2	2	0	27

As it is clearly seen from all three examples, the corresponding *m*-valued functions are minimal monotonic functions. All these functions can be learned using a single MVN with the activation function (2.50) because they are partially defined *kl*=*m*-valued threshold functions.

Table 5.23 mod 5 addition of 2 variables, *l*=4, *m*=20 in (5.157)

x_1	x_2	$f(x_1,x_2)=$ $=(x_1+x_2)\bmod 5$	$j\in M=\{0,1,...,9\}$ 5x4=20-valued function $\tilde{f}(x_1,x_2)$
0	0	0	0
0	1	1	1
0	2	2	2
0	3	3	3
0	4	4	4
1	0	1	1
1	1	2	2
1	2	3	3
1	3	4	4
1	4	0	5
2	0	2	2
2	1	3	3
2	2	4	4
2	3	0	5
2	4	1	6
3	0	3	3
3	1	4	4
3	2	0	5
3	3	1	6
3	4	2	7
4	0	4	4
4	1	0	5
4	2	1	6
4	3	2	7
4	4	3	8

Table 5.24 mod 6 addition of 2 variables, l=4, m=24 in (5.157)

x_1	x_2	$f(x_1,x_2)=$ $=(x_1+x_2)\bmod 6$	$j\in M=\{0,1,...,23\}$ 6x4=24-valued function $\tilde{f}(x_1,x_2)$
0	0	0	6
0	1	1	7
0	2	2	8
0	3	3	9
0	4	4	10
0	5	5	11
1	0	1	7
1	1	2	8
1	2	3	9
1	3	4	10
1	4	5	11
1	5	0	12
2	0	2	8
2	1	3	9
2	2	4	10
2	3	5	11
2	4	0	12
2	5	1	13
3	0	3	9
3	1	4	10
3	2	5	11
3	3	0	12
3	4	1	13
3	5	2	14
4	0	4	10
4	1	5	11
4	2	0	12
4	3	1	13
4	4	2	14
4	5	3	15
5	0	5	11
5	1	0	12
5	2	1	13
5	3	2	14
5	4	3	15
5	5	4	16

It should be mentioned that neither of the learning rules (3.92), (3.94)-(3.96) can be distinguished as the "best". Each of them can be good for solving different problems. It is also not possible to distinguish the best among Learning Strategies 1 and 2. For example, the "Iris" problem can be learned with no errors only using the Strategy 2, while for some other problems Strategy 1 gives better results.

A deeper study of advantages and disadvantages of the developed learning strategies and rules will be an interesting direction for the further research.

5.4 Concluding Remarks to Chapter 5

In this Chapter, we have considered the multi-valued neuron with a periodic activation function (MVN-P). This is a discrete-valued neuron (whose inputs can be discrete or continuous), which can learn input/output mappings that are non-linearly separable in the real domain, but become linearly separable in the complex domain.

First, we have considered the universal binary neuron (UBN). This neuron with a binary output was a prototype of MVN-P. We have shown that UBN with its periodic activation function projects a binary input/output mapping, which is non-linearly separable, into an $2l = m$-valued partially defined threshold function, which can be learned using a single neuron. We have considered the UBN learning algorithm, which is based on the MVN learning algorithm employing also self-adaptivity. We have shown that a single UBN may easily learn such problems as XOR and Parity.

Then we have introduced MVN-P. The MVN-P concept generalizes the two-valued UBN concept for the k-valued case. The MVN-P has a periodic activation function, which projects k-valued logic into $kl = m$-valued logic. Thus, this activation function is k-periodic and l-repetitive. The most wonderful property of MVN-P is its ability to learn k-valued input/output mappings, which are non-linearly separable in k-valued logic, but become linearly separable in $kl = m$-valued logic. This means that MVN-P may project a non-linearly separable k-valued function into a partially defined linearly separable m-valued function.

We have considered the MVN-P learning algorithm with the two learning strategies. This learning algorithm is semi-supervised and semi-self-adaptive. While a desired neuron output is known in advance, a periodicity of its activation function allows its self-adaptation to the input/output mapping. We have shown that the MVN-P learning can be based on the same error-correction learning rules that the regular MVN learning. The most important application of MVN-P is its ability to solve multi-class and multi-cluster classification problems, which are non-linearly separable in the real domain, without any extension of the initial space where a problem is defined.

MVN-P can also be used as the output neuron in MLMVN. This may help to solve highly nonlinear classification problems.