Capitolul 1

Reţele neuronale cu un singur strat

1.1. Neuronii biologici

Neuronii sunt componentele fundamentale ale sistemului nervos, a cărui complexitate este extraordinară. Creierul uman are în medie 86 de miliarde de neuroni (Herculano-Houzel, 2009). În figura 1.1 (Monoyios, 2011) se poate vedea fotografia unui neuron real. Numele vine din limba greacă, νευρών însemnând *tendon*, adică un fascicul subțire şi rezistent cu ajutorul căruia se fixează mușchii pe oase. S-a considerat că nervii au aspectul unor tendoane. Neuronii mai sunt numiţi uneori şi celule nervoase, deşi mulţi neuroni nu formează nervi iar nervii includ şi alte celule decât neuroni. Diametrul unui neuron este de 4-100 microni iar greutatea sa nu depăşeşte 10^{-6} grame (Groves & Rebec, 1988; Chudler, 2014).

Simplificând puţin lucrurile, un neuron este alcătuit din *corpul celular*, *dendrite* (din gr. δένδρον, *copac*), terminaţii cu aspect arborescent care primesc impulsuri de la alţi neuroni şi un *axon* (din gr. ἄξων, *axă*), care trimite impulsuri electrice către alţi neuroni, după cum se poate vedea în figura 1.2 (adaptată după Droual, 2011).

Neuronii sunt conectaţi prin *sinapse* (din gr. σύν, *cu, împreună* şi ἅπτω, *a prinde, a fixa* cu derivatul ἅψᾱ, *legătură*). În scoarţa cerebrală există 150 de trilioane de sinapse (Pakkenberg & Gundersen, 1997; Pakkenberg et al., 2003; Chudler, 2014). Un neuron se conectează cu alţi 1000-10000 de neuroni, în medie 7000.

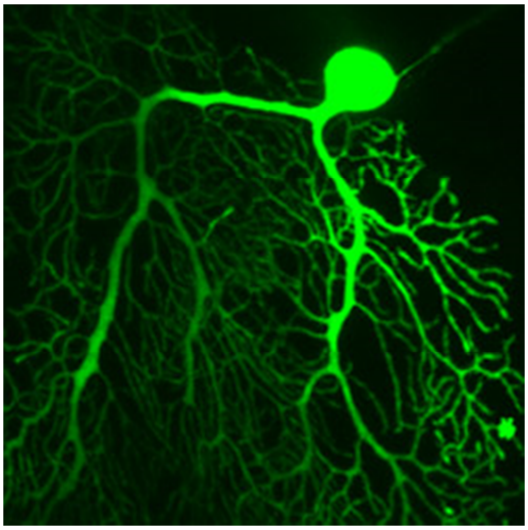


Figura 1.1. *Neuron biologic real*

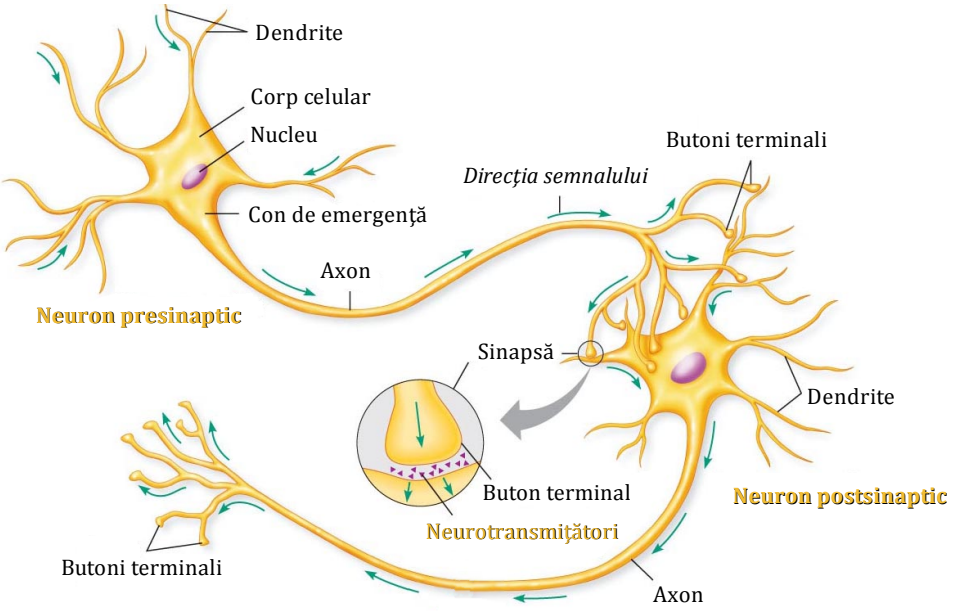


Figura 1.2. *Structura neuronilor. Sinapsă chimică*

Atât în interiorul cât şi în exteriorul celulelor neuronale se găsesc ioni pozitivi şi negativi. Schimbările de concentraţie ale acestora determină apariţia unor curenţi electrici sau impulsuri nervoase. Transmiterea de la un neuron la altul a acestor impulsuri se face prin intermediul sinapselor, unidirecţional de la axoni la dendrite (deşi există şi alte combinaţii posibile). De fapt, neuronii nu se ating direct, există o regiune foarte îngustă, de aproximativ 20 nm între membranele pre- şi postsinaptice, numită *fantă sinaptică*. Transferul impulsurilor se realizează prin intermediul unor substanţe chimice şi nu prin curenţi electrici. Acesta este cel mai des întâlnit tip de sinapsă, numit *sinapsă chimică*.

Atunci când impulsul electric ajunge într-un buton terminal al axonului presinaptic, acesta atrage ioni pozitivi de calciu care determină „vărsarea" în fanta sinaptică a unor *neurotransmiţători*. Aceştia activează unii receptori în partea postsinaptică şi determină depolarizarea neuronului postsinaptic, adică potenţialul membranei devine pozitiv deoarece în celulă intră ioni pozitivi de sodiu. Dacă depolarizarea atinge un anumit nivel, în celulă se propagă un *impuls nervos*. Unii neurotransmiţători, dimpotrivă, fac ca neuronul postsinaptic să se hiperpolarizeze, adică potenţialul membranei să devină negativ. Ionii negativi de clor intră în celulă iar ionii pozitivi de potasiu ies din celulă. Acest fapt împiedică generarea unui impuls electric. În primul caz, potenţialul sinaptic este *excitator*; în al doilea caz, este *inhibitor*.

În scurt timp după eliberarea în fanta sinaptică, neurotransmiţătorii sunt dizolvaţi de enzime sau reabsorbiţi în butonii presinaptici, iar concentraţia de ioni revine la valoarea iniţială. Imediat după generarea unui impuls urmează o aşa numită *perioadă refractară*, în care neuronul „se

încarcă" şi nu se mai poate activa din nou. Abia apoi neuronul revine în starea de repaus şi poate genera un nou impuls.

Neuronii au un *prag de depolarizare*. Dacă potenţialul creat este mai mic decât acest prag, neuronul postsinaptic nu se activează. Potenţialul creat de o sinapsă excitatoare este mult mai mic decât pragul de depolarizare, prin urmare un impuls poate fi generat doar prin efectul combinat al mai multor sinapse. Dintre miile de terminaţii sinaptice care sunt conectate la un neuron, câteva sute sunt active simultan sau la intervale de timp suficient de apropiate ca efectele lor să se poată însuma. Potenţialul membranar al neuronului postsinaptic este în fiecare moment rezultanta activităţii tuturor sinapselor active în acel moment.

În figura 1.2 se poate observa formaţiunea denumită *con de emergenţă al axonului*. Acesta este ultimul loc din corpul celular unde potenţialele din intrările sinaptice se *sumează* înainte de a fi transmise axonului.

Neuronul respectă principiul *totul sau nimic*. Dacă depolarizarea nu este suficient de puternică pentru a depăşi pragul, canalele de ioni nu se deschid. Dacă depolarizarea depăşeşte pragul, canalele se deschid şi se generează un impuls electric. Acesta este întotdeuna la fel de mare, de exemplu 40 mV, fără valori intermediare. Intensitatea unui stimul este dată de frecvenţa impulsurilor. Unui stimul mai puternic îi corespunde o frecvenţă mai mare. De exemplu, un stimul de durere puternică poate avea o frecvenţă de până la 800 Hz (Malmivuo & Plonsey, 1995; Freudenrich, 2007; Mastin, 2010; Ribrault, Sekimoto & Triller, 2011; Tamarkin, 2011; Gregory, 2014).

Acest principiu poate fi descris prin analogie cu aprinderea unui fitil, care necesită o anumită temperatură. Sub aceasta, fitilul nu se aprinde. Însă

un chibrit cu o temperatură mai mare decât pragul nu face fitilul să ardă mai repede, odată ce s-a aprins (Byrne, 2014).

În figura 1.3 (după Wikimedia Commons, 2014a) se prezintă un impuls tipic, unde se pot vedea şi valorile curenţilor propriu-zişi şi ale pragului.

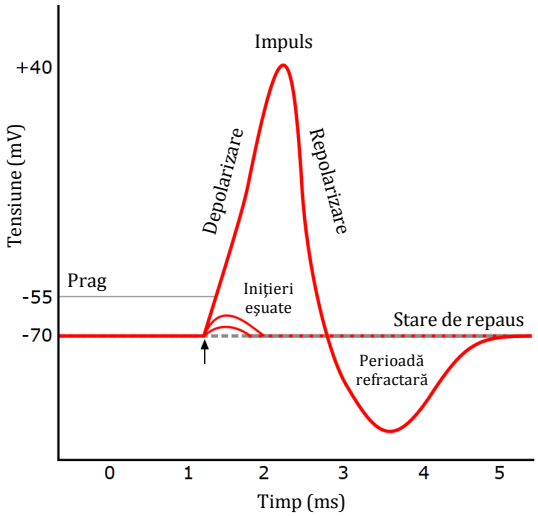


Figura 1.3. *Impuls neuronal tipic*

Un alt principiu biologic pe care se bazează o modalitate adaptivă de învăţare numită *învăţare hebbiană* (Hebb, 1949) este acela că dacă un neuron activează în mod repetat alt neuron, apar modificări fizice care cresc eficienţa acestei interacţiuni. Pe scurt, „neuronii care se activează împreună se cablează împreună” (engl. “neurons that fire together wire together”, Shatz, 1992; Doidge, 2007). Cu alte cuvinte, conexiunea dintre neuronii respectivi se întăreşte iar impulsurile se transmit mai uşor. De aceea, dacă repetăm de suficient de multe ori o acţiune, ajungem să o realizăm în mod automat (Ware, 2013).

1.2. Perceptronul

1.2.1. Neuronul McCulloch-Pitts

Primul model matematic al unui neuron a fost propus de McCulloch şi Pitts (1943; 1947). Modelul este prezentat în figura 1.4 (Ngom, 2010).

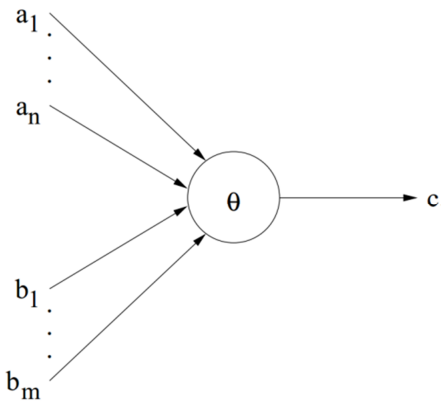


Figura 1.4. *Neuronul McCulloch-Pitts*

Ieşirea este binară: neuronul este activat (1) sau nu (0), ceea ce îl face echivalent cu o propoziţie logică, care poate fi adevărată sau falsă. Intrările sunt excitatoare (a_i) sau inhibitoare (b_j). Aceste intrări sunt sumate direct şi neuronul se activează dacă suma depăşeşte un prag fix. De asemenea, neuronul se activează doar dacă nu există intrări inhibitoare.

Funcţia de activare este următoarea:

$$c = \begin{cases} 1, \ dacă \ \sum_{i=1}^{n} a_i \geq \theta \ şi \ b_j = 0, \ \forall j = 1..m \\ 0, \ altfel \end{cases} \tag{1.1}$$

Orice problemă care poate fi reprezentată sub forma unei funcţii logice poate fi modelată de o reţea de neuroni McCulloch-Pitts deoarece orice funcţie booleană poate fi implementată folosind doar operaţiile SAU, ŞI şi NEGAŢIE. În figura 1.5 (după Ngom, 2010) sunt prezentate aceste funcţii logice elementare.

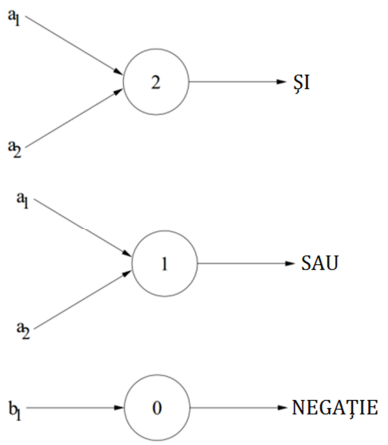


Figura 1.5. *Funcţii logice elementare implementate ca neuroni McCulloch-Pitts*

Dificultatea principală a modelului constă în faptul că îi lipseşte capacitatea de învăţare; pragurile sunt determinate analitic. Pentru funcţii complexe, dimensiunea reţelei corespunzătoare este mare.

1.2.2. Perceptronul originar al lui Rosenblatt

Problema cea mai importantă pe care a încercat să o rezolve Rosenblatt (1957; 1958) este posibilitatea de a învăţa, o calitate esenţială a reţelelor neuronale biologice. Sistemul propus de el modela sistemul vizual uman, de aceea s-a numit *perceptron* (figura 1.6). Dintr-o imagine raster, valorile pixelilor treceau prin nişte conexiuni cu valori aleatorii, rezultând

nişte trăsături sintetice ale imaginii. Aceste trăsături erau conectate la ieşire, prin modelul standard pe care îl vom discuta în secţiunea următoare. Antrenând perceptronul cu o mulţime de imagini şi ieşirile corespunzătoare, sistemul putea învăţa să clasifice imaginile (Kröse & van der Smagt, 1996; Champandard, 2003).

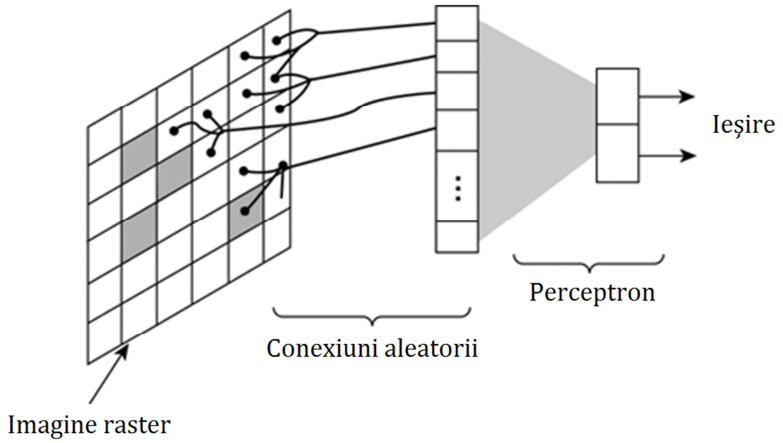


Figura 1.6. *Perceptronul originar al lui Rosenblatt*

Problema principală a acestui model este că nu s-a reuşit găsirea unei modalităţi de determinare a parametrilor conexiunilor dintre imagine (echivalentul în model al retinei) şi stratul intermediar corespunzător trăsăturilor, ci doar dintre acesta şi ieşire. Este ceea ce vom prezenta în continuare.

1.2.3. Perceptronul standard

Perceptronul este un neuron cu mai multe intrări x_i, fiecare conexiune de intrare având o valoare numită pondere w_i (engl. "weight"),

care este o măsură a importanţei acelei intrări, un prag θ şi o funcţie de activare semn sau treaptă. Structura sa generală este prezentată în figura 1.7.

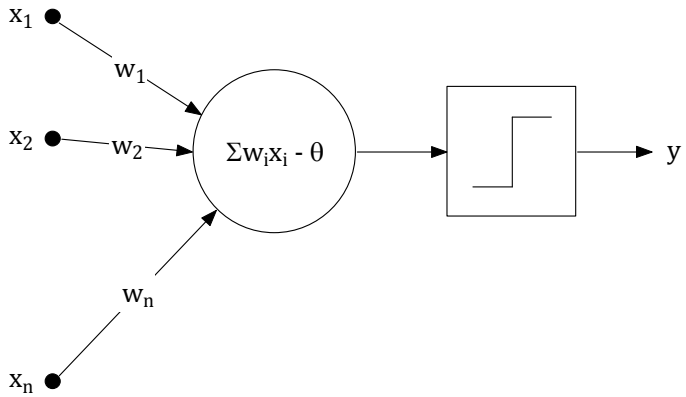


Figura 1.7. *Structura generală a perceptronului*

Se poate vedea analogia cu modul de funcţionare al unui neuron biologic, în care semnalele de intrare sunt sumate iar neuronul generează un semnal doar dacă suma depăşeşte pragul.

Ieşirea perceptronului este dată de următoarea ecuaţie:

$$y = F\left(\sum_{i=1}^{n} w_i x_i - \theta\right), \tag{1.2}$$

unde F este funcţia semn:

$$F(a) = \begin{cases} -1, & dacă\ a < 0 \\ 1, & dacă\ a \geq 0 \end{cases} \tag{1.3}$$

sau funcţia treaptă:

$$F(a) = \begin{cases} 0, & dacă\ a < 0 \\ 1, & dacă\ a \geq 0 \end{cases}. \tag{1.4}$$

Scopul perceptronului este rezolvarea problemelor de clasificare binară. Se dă o mulţime de vectori de antrenare, care conţin valori pentru cele n intrări şi valoarea ieşirii dorite. Se doreşte determinarea ponderilor şi pragului astfel încât modelul să clasifice corect *toţi* vectorii de antrenare într-una din cele 2 clase, adică ieşirea perceptronului pentru un vector de intrare să fie egală cu ieşirea dorită.

Pentru a înţelege semnificaţia parametrilor, să considerăm un perceptron cu o singură intrare. În ecuaţiile 1.3 sau 1.4, se vede că dacă a este pozitiv vectorul va fi atribuit unei clase iar dacă este negativ, vectorul va fi atribuit celeilalte clase. Prin urmare, separarea celor 2 clase este dată de o linie, care în cazul unidimensional are ecuaţia:

$$w \cdot x - \theta = 0. \tag{1.5}$$

Exemplu

Să considerăm următoarea situaţie:

$$f(x) = \frac{1}{2}x - 1,$$

reprezentată în figura 1.8a. Pentru $x < 2$, răspunsul va fi clasa $-1/0$ iar pentru $x >= 2$, răspunsul perceptronului va fi clasa 1.

Florin Leon (2014). Inteligenţă artificială: maşini cu vectori suport
Tehnopress, Iaşi, ISBN 978-606-687-155-6, http://florinleon.byethost24.com

Mai întâi să vedem ce se întâmplă când pragul rămâne constant şi se modifică ponderea. Fie următoarea situaţie, în care ponderea s-a schimbat de la $0{,}5$ la 2:

$$f(x) = 2x - 1.$$

Comparând figurile 1.8a şi 1.8b, se vede că panta diferă. Prin urmare, ponderea exprimă panta dreptei. În figura 1.8b, se vede cum punctul de intersecţie cu ordonata a rămas -1, valoare dată de prag, însă datorită schimbării ponderii, punctul de separare s-a schimbat din 2 în $0{,}5$.

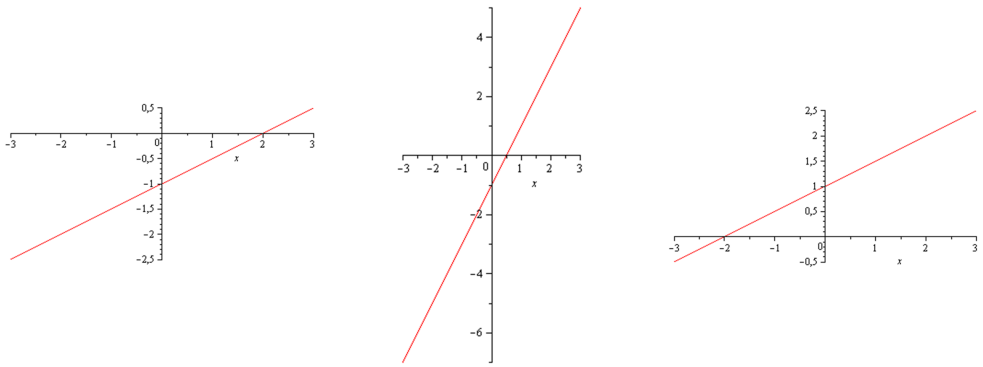


Figura 1.8. *Reprezentări geometrice ale unor decizii unidimensionale*

Acum să considerăm din nou prima situaţie, menţinând ponderea la valoarea $0{,}5$, dar modificând pragul de la 1 la -1:

$$f(x) = \frac{1}{2}x + 1.$$

Comparând figurile 1.8a şi 1.8c, se vede că pragul a translat dreapta în sus, panta rămânând aceeaşi. Punctul de separare s-a mutat în -2.

În general, pentru cazul unidimensional, punctul de separare este, conform ecuaţiei 1.5:

$$x = \frac{\theta}{w}. \tag{1.6}$$

În continuare, să considerăm un perceptron cu 2 intrări. Separarea celor 2 clase este dată de o dreaptă cu ecuaţia:

$$w_1 x_1 + w_2 x_2 - \theta = 0 \tag{1.7}$$

care poate fi rescrisă astfel:

$$x_2 = -\frac{w_1}{w_2} x_1 + \frac{\theta}{w_2}. \tag{1.8}$$

Ecuaţia este reprezentată în figura 1.9 (după Kröse & van der Smagt, 1996).

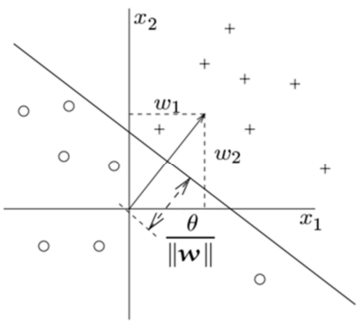


Figura 1.9. *Reprezentarea geometrică a deciziei bidimensionale*

Pentru cazul bidimensional considerat, se observă că panta dreptei de separare este dată de valoarea ponderilor. Dreapta de separare este

Florin Leon (2014). Inteligenţă artificială: maşini cu vectori suport
Tehnopress, Iaşi, ISBN 978-606-687-155-6, http://florinleon.byethost24.com

întotdeauna perpendiculară pe dreapta definită de origine şi de punctul (w_1, w_2).

Pragul marchează deplasarea dreptei de separare faţă de origine. În general, distanţa de la un punct la o dreaptă este :

$$d\left(ax + by + c = 0,\ (x_0, y_0)\right) = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}} \tag{1.9}$$

iar în cazul nostru distanţa de la origine la dreapta de separare este:

$$d = \frac{\theta}{\sqrt{w_1^2 + w_2^2}} = \frac{\theta}{\|\mathbf{w}\|}. \tag{1.10}$$

O observaţie importantă pe baza figurilor 1.8 şi 1.9 este că perceptronul poate învăţa să separe doar clase ale căror instanţe nu sunt intercalate, numite *separabile liniar*. În cazul bidimensional, avem o dreaptă care împarte planul în două. De o parte a dreptei se află o clasă iar de cealaltă parte se află cealaltă clasă. Dacă am fi avut 3 intrări, ar fi existat o suprafaţă de separare care ar fi împărţit spaţiul în 2 regiuni. În cazul general n-dimensional, perceptronul defineşte un hiperplan de separare.

1.2.4. Regula de învăţare a perceptronului

În ecuaţia 1.2 apar atât ponderile cât şi pragul. De fapt, aceşti parametri pot fi trataţi unitar, deoarece intrarea totală a neuronului reprezintă până la urmă o sumă. De aceea, pentru a simplifica modelul de calcul, se consideră că pragul defineşte încă o intrare a neuronului:

$$y = F\left(\sum_{i=1}^{n} w_i x_i - \theta\right) = F\left(w_1 x_1 + ... + w_n x_n + \theta \cdot (-1)\right). \quad (1.11)$$

Considerând această intrare suplimentară ca fiind 1 în loc de –1, pragul va fi valoarea negată a ponderii conexiunii respective. În acest mod, algoritmul de învăţare are ca scop doar determinarea unor ponderi. Arhitectura perceptronului după aceste transformări este prezentată în figura 1.10.

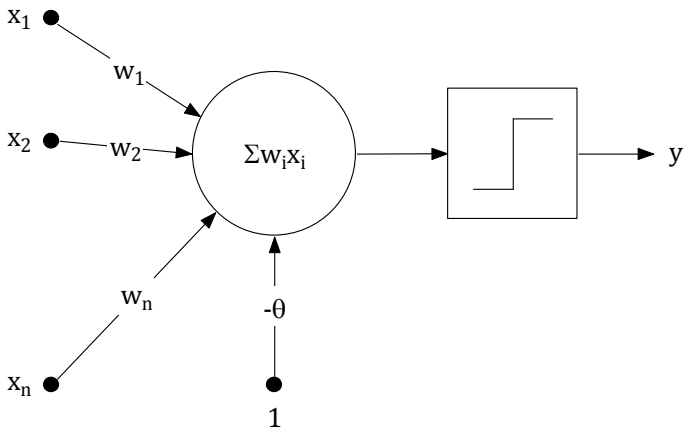


Figura 1.10. *Perceptronul. Pragul poate fi considerat ponderea unei conexiuni suplimentare*

Astfel, ieşirea este:

$$y = F\left(\sum_{i=1}^{n+1} w_i x_i\right). \quad (1.12)$$

Pentru a descrie regula de învăţare a perceptronului, vom utiliza următoarele notaţii. Fie $\mathbf{x} = (x_1, \ldots, x_n)$ un vector de intrare. Mulţimea de

antrenare conţine N astfel de vectori. Pentru vectorul $\mathbf{x}$, y este ieşirea calculată de perceptron iar y_d este ieşirea dorită (corectă, cunoscută). Fie $\mathbf{w} = (w_1, \ldots, w_n, w_{n+1})$ vectorul de ponderi. Conform celor discutate anterior, $w_{n+1} = -\theta$.

Învăţarea are loc prin modificarea valorilor ponderilor pentru a reduce diferenţa dintre ieşirile reale şi ieşirile dorite, pentru toate datele de antrenare. Instanţele de antrenare se prezintă la intrarea reţelei succesiv, calculându-se ieşirea reţelei şi eroarea. Pe baza erorii se ajustează ponderile. Prezentarea instanţelor se face iterativ, până se termină toată mulţimea de antrenare. Prezentarea tuturor instanţelor reprezintă o *epocă* de antrenare. Apoi, dacă mai există erori, procesul poate reîncepe cu o nouă epocă şi continuă până când ieşirea perceptronului este egală cu ieşirea dorită pentru toate instanţele de antrenare.

Dacă după prezentarea instanţei i ieşirea reală este y_i iar ieşirea dorită este y_{di}, atunci eroarea este:

$$e_i = y_{di} - y_i. \tag{1.13}$$

Dacă eroarea este pozitivă, trebuie să creştem ieşirea perceptronului y_i. Dacă eroarea este negativă, trebuie să micşorăm ieşirea y_i.

Exemplu

Să considerăm problema de clasificare bidimensională definită de mulţimea de antrenare din tabelul 1.1, care ne va ajuta să înţelegem algoritmul de antrenare.

Tabelul 1.1. *Mulţime de antrenare*

Intrări	Ieşire dorită
1 1	1
1 –1	0

Vom folosi pentru perceptron funcţia treaptă, însă dacă problema era definită cu valori ale clasei de –1 în loc de 0, se putea folosi funcţia semn fără alte modificări.

De asemenea, pentru a simplifica şi mai mult lucrurile, vom ignora pragul. În această situaţie, găsirea perechii de ponderi se rezumă la a găsi orientarea potrivită a unei drepte care se poate roti în jurul originii.

Dacă vectorul de ponderi este **w** = (–0,2, 0,1), ieşirile perceptronului pentru cei doi vectori vor fi:

$$y_1 = F(-0,2 \cdot 1 + 0,1 \cdot 1) = F(-0,1) = 0$$

$$y_2 = F(-0,2 \cdot 1 + 0,1 \cdot (-1)) = F(-0,3) = 0$$

Primul vector nu este clasificat corect: $y_1 = 0$ însă $y_{d1} = 1$. Eroarea este $e = 1$. Situaţia este reprezentată în figura 1.11.

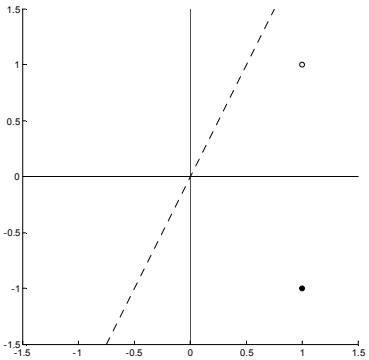


Figura 1.11. *Procesul de învăţare: ajustarea ponderilor (cazul 1)*

Se vede că ambele puncte sunt sub dreapta de separare. Ecuaţia dreptei este:

$$w_1 x_1 + w_2 x_2 = 0 \tag{1.14}$$

echivalent cu:

$$x_2 = -\frac{w_1}{w_2} x_2, \tag{1.15}$$

deci panta dreptei este $-w_1 / w_2$.

Am dori să scădem panta dreptei, astfel încât să treacă printre cele două puncte. Întrucât eroarea apare la primul vector, cel de sus, trebuie modificat w_1. Prin urmare, trebuie mărit w_1, de exemplu la valoarea $w_1 = -0{,}05$, rezultând situaţia din figura 1.12.

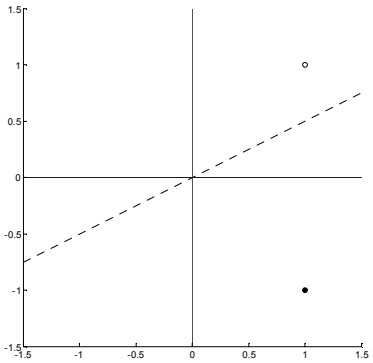


Figura 1.12. *Procesul de învăţare: ajustarea ponderilor (cazul 2)*

Acum ieşirile perceptronului vor fi:

$$y_1 = F(-0,05 \cdot 1 + 0,1 \cdot 1) = F(0,05) = 1$$

$$y_2 = F(-0,05 \cdot 1 + 0,1 \cdot (-1)) = F(-0,15) = 0$$

care sunt răspunsurile corecte.

Analog, dacă vectorul de ponderi este **w** = (0,2, 0,1), ieşirile perceptronului pentru cei doi vectori vor fi:

$$y_1 = F(0,2 \cdot 1 + 0,1 \cdot 1) = F(0,3) = 1$$

$$y_2 = F(0,2 \cdot 1 + 0,1 \cdot (-1)) = F(0,1) = 1$$

adică dreapta trece pe dedesubtul vectorului al doilea. Eroarea acestuia este $e(p) = y_d(p) - y(p) = 0 - 1 = -1$. În acest caz, panta $-w_1 / w_2$ trebuie crescută şi în consecinţă trebuie mărit w_2, să spunem până la $w_2 = 0,4$.

Ieşirile perceptronului vor fi:

$$y_1 = F(0,2 \cdot 1 + 0,4 \cdot 1) = F(0,6) = 1$$

$$y_2 = F(0,2 \cdot 1 + 0,4 \cdot (-1)) = F(-0,2) = 0$$

şi corespund răspunsurilor corecte.

Să sintetizăm rezultatele. În primul caz, $e_1 > 0$, $x_1 > 0$ şi diferenţa cu care am actualizat ponderea, $\Delta w_1 > 0$. În al doilea caz, $e_2 < 0$, $x_2 < 0$ şi $\Delta w_2 > 0$.

Acum să considerăm exemplul rotit cu 180° în jurul originii, definind problema următoare din tabelul 1.2, cu vectorul de ponderi **w** = (0,2, –0,1), după cum se poate vedea în figura 1.13.

Florin Leon (2014). Inteligenţă artificială: maşini cu vectori suport
Tehnopress, Iaşi, ISBN 978-606-687-155-6, http://florinleon.byethost24.com

Tabelul 1.2. *Mulţime de antrenare*

Intrări	Ieşire dorită
–1 1	0
–1 –1	1

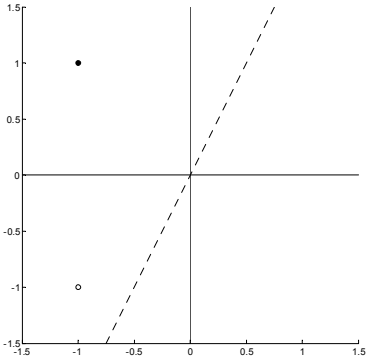


Figura 1.13. *Procesul de învăţare: ajustarea ponderilor (cazul 3)*

Ieşirile perceptronului pentru cei doi vectori vor fi:

$$y_1 = F(0,2 \cdot (-1) + (-0,1) \cdot 1) = F(-0,3) = 0$$

$$y_2 = F(0,2 \cdot (-1) + (-0,1) \cdot (-1)) = F(-0,1) = 0$$

Vectorul 2 are eroarea $e_2 = 1 - 0 > 0$. Acum am dori să scădem panta dreptei $-w_1 / w_2$, astfel încât cele două puncte să fie separate, modificând w_2. Ponderea w_2 trebuie să scadă. Pentru $w_2 = -0,3$, vom avea:

$$y_1 = F(0,2 \cdot (-1) + (-0,3) \cdot 1) = F(-0,5) = 0$$

$$y_2 = F(0,2 \cdot (-1) + (-0,3) \cdot (-1)) = F(0,1) = 1$$

ceea ce reprezintă o clasificare corectă.

În această situaţie avem $e_2 > 0$, $x_2 < 0$ şi $\Delta w_2 < 0$.

La fel putem găsi o configuraţie în care $e < 0$, $x > 0$ şi $\Delta w < 0$.

Întrucât dorim să determinăm modul în care se schimbă ponderile, vom sumariza cele 4 cazuri în tabelul 1.3.

Tabelul 1.3. *Schimbarea ponderilor în funcţie de semnele erorii şi intrării*

e	x	Δw
> 0	> 0	> 0
< 0	< 0	> 0
> 0	< 0	< 0
< 0	> 0	< 0

Se vede că semnul lui Δw este produsul semnelor lui e şi x.

Cantitatea cu care trebuie să modificăm vectorul **w** poate fi mai mare sau mai mică. În general, neştiind care este valoarea exactă a diferenţei, folosim succesiv o serie de paşi mici, până este îndeplinită condiţia de eroare. În acest sens, se utilizează un număr $\eta \in (0, 1]$ numit *rată de antrenare*, care indică mărimea paşilor făcuţi pentru găsirea soluţiei.

Pentru modificarea ponderilor se utilizează ecuaţia:

$$\Delta w = \eta \cdot x \cdot e. \tag{1.16}$$

Aceasta este relaţia fundamentală care caracterizează regula de învăţare a perceptronului.

Pseudocodul algoritmului este următorul.

Florin Leon (2014). Inteligenţă artificială: maşini cu vectori suport
Tehnopress, Iaşi, ISBN 978-606-687-155-6, http://florinleon.byethost24.com

```
se iniţializează toate ponderile wi cu 0 sau cu valori aleatorii din intervalul [–0,5, 0,5]
se iniţializează rata de învăţare eta cu o valoare din intervalul (0, 1], de exemplu 0,1
se iniţializează numărul maxim de epoci P, de exemplu 100
p = 0 // numărul epocii curente
erori = true // un flag care indică existenţa erorilor de antrenare

repetă cât timp p < P şi erori == true
{
   erori = false
   pentru fiecare vector de antrenare xi cu i = 1..N
   {
         yi = F(sum(xij * wj)) cu j = 1..n+1
         dacă (yi != ydi)
         {
            e = ydi – yi
            erori = true
            pentru fiecare intrare j = 1..n+1
               wj = wj + eta * xij * e
         }
   }
   p = p + 1
}
```

Teorema de convergenţă a regulii de învăţare a perceptronului (Rosenblatt, 1962) arată că dacă o problemă poate fi rezolvată (dacă este separabilă liniar), atunci algoritmul converge spre o soluţie într-un număr finit de paşi.

Perceptronul este cea mai simplă formă de reţea neuronală cu propagare înainte (engl. "feed forward"), în care semnalele se propagă doar de la intrări spre ieşiri, fără bucle de reacţie.

El poate învăţa tot ce poate reprezenta, dar nu poate reprezenta foarte mult (Russell & Norvig, 2002). Întrucât majoritatea problemelor interesante din viaţa reală nu sunt separabile liniar, aceasta este o deficienţă majoră a

modelului (Minsky & Papert, 1969), care a condus la scăderea interesului cercetării în domeniul reţelelor neuronale în anii 70, până când a fost propusă o nouă arhitectură, perceptronul multistrat, cu un algoritm de învăţare eficient, bazat pe retropropagarea erorii (engl. "backpropagation").

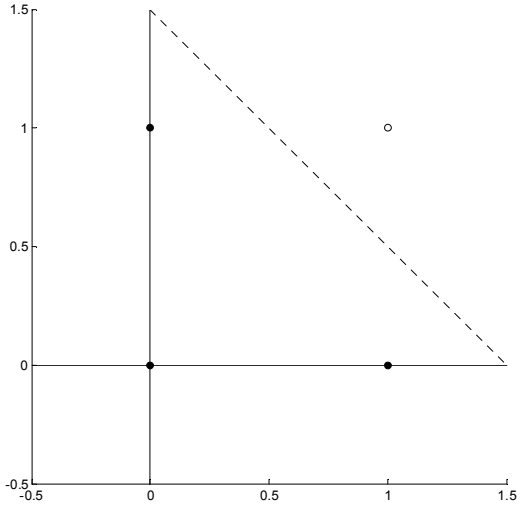


Figura 1.14. *Soluţie pentru problema logică ŞI*

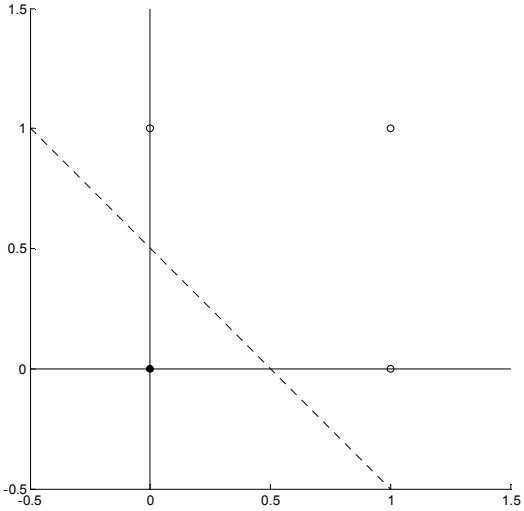


Figura 1.15. *Soluţie pentru problema logică SAU*

De exemplu, este uşor ca perceptronul să înveţe operaţii logice precum ŞI (figura 1.14) sau SAU (figura 1.15), întrucât acestea sunt separabile liniar.

Este de asemenea important de spus faptul că soluţia nu este unică, de exemplu pentru problema ŞI o soluţie alternativă este cea din figura 1.16.

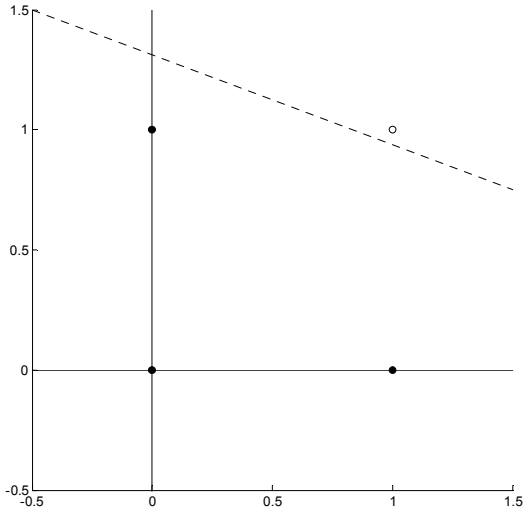


Figura 1.16. *Soluţie alternativă pentru problema logică ŞI*

Vom discuta ideea de a găsi cea mai bună soluţie în capitolul 3, în care începe descrierea maşinilor cu vectori suport.

Însă nu există nicio dreaptă care poate separa clasele operaţiei SAU-EXCLUSIV (engl. XOR); această problemă nu este separabilă liniar (figura 1.17).

Când clasele nu sunt separabile liniar, ponderile se modifică permanent pe parcursul antrenării, într-o manieră aparent aleatorie. În general, este dificilă determinarea *apriori* a separabilităţii liniare a unei

probleme, pentru a şti dacă perceptronul va putea învăţa funcţia corespunzătoare.

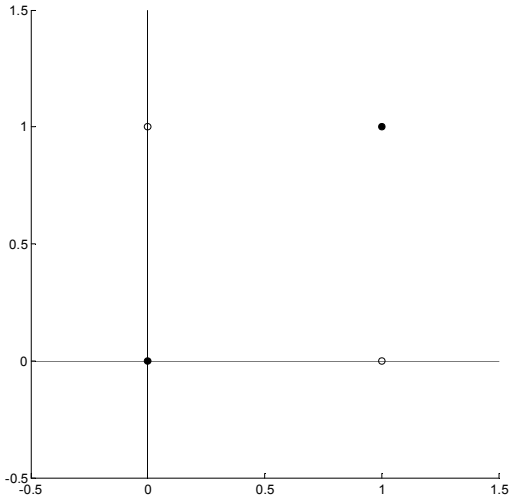


Figura 1.17. *Problema logică SAU-EXCLUSIV*

Totuşi, există şi funcţii complexe separabile linear, de exemplu funcţia majoritate cu n intrări, a cărei valoare este 1 dacă mai mult de jumătate din intrări sunt 1 şi $-1/0$ altfel. Perceptronul este un model foarte simplu care poate rezolva această problemă, pe când un arbore de decizie ar necesita $O(2^n)$ noduri pentru învăţarea sa.

1.3. Adaline. Regula delta

La scurt timp după apariţia perceptronului, Widrow şi Hoff (1960) au prezentat modelul *adaline* (abreviere de la "ADAptive LInear NEuron" şi mai târziu, după scăderea popularităţii reţelelor neuronale, de la "ADAptive LINear Element").

Ca structură, adaline este similar cu perceptronul, diferenţa fiind lipsa funcţiei treaptă sau semn; adaline are o funcţie de activare liniară:

$$y = \sum_{i=1}^{n+1} w_i x_i. \tag{1.17}$$

Avantajul folosirii funcţiei liniare este pe de o parte posibilitatea de a avea ieşiri continue şi deci o putere de reprezentare mai mare şi pe de alta, faptul că se pot utiliza metode diferenţiale pentru învăţarea valorilor ponderilor.

Obiectivul algoritmului de învăţare este minimizarea erorii medii pătratice (engl. "Mean Square Error", MSE), care este o măsură a diferenţei dintre ieşirile dorite şi ieşirile reale ale reţelei.

Eroarea corespunzătoare unui vector de antrenare este:

$$E_i = \frac{1}{2}\left(y_{di} - y_i\right)^2. \tag{1.18}$$

Eroarea pentru toată mulţimea de antrenare este media tuturor acestor erori:

$$E = \frac{1}{N}\sum_{i=1}^{N} E_i = \frac{1}{2N}\sum_{i=1}^{N}\left(y_{di} - y_i\right)^2. \tag{1.19}$$

Scopul final este minimizarea erorii totale E.

S-ar putea pune întrebarea de ce se foloseşte această măsură pătratică a diferenţei şi nu alta, de exemplu valoarea sa absolută: $|y_{di} - y_i|$.

Un motiv simplu este că această cantitate va fi derivată pentru obţinerea unei formule de actualizare a ponderilor. Fiind o funcţie cuadratică, este uşor de minimizat, iar puterea 2 se simplifică cu coeficientul 1/2, aceasta fiind explicaţia pentru includerea sa.

Un alt motiv, mai complicat, se referă la unele proprietăţi statistice ale datelor, care sunt puse în evidenţă de pătratul diferenţelor, în condiţiile în care se presupune că aceste diferenţe respectă o distribuţie gaussiană.

De asemenea, pătratul diferenţelor ne face să privim eroarea ca pe o distanţă euclidiană între valorile dorite şi valorile reale furnizate de reţea. Folosirea valorilor absolute ale diferenţelor ar fi echivalentă cu aplicarea distanţei Manhattan (Leon, 2012). Ambele variante ar putea fi utilizate, însă distanţa euclidiană poate fi considerată mai intuitivă.

În cazul modelului adaline, dorim minimizarea erorii în raport cu ponderile, adică găsirea iterativă a valorilor ponderilor care conduc la o eroare minimă.

După prezentarea unui vector de antrenare i, în conformitate cu mărimea erorii, va trebui să aplicăm o corecţie tuturor ponderilor j, proporţională cu contribuţia fiecărei ponderi la apariţia acelei erori:

$$\Delta_i w_j = -\eta \frac{\partial E_i}{\partial w_j}, \tag{1.20}$$

unde η este rata de învăţare.

Însă ponderile nu acţionează direct asupra erorii, de aceea putem scrie derivata sub următoarea formă, luând în calcul şi ieşirea reţelei:

$$\frac{\partial E_i}{\partial w_j} = \frac{\partial E_i}{\partial y_i} \frac{\partial y_i}{\partial w_j}. \tag{1.21}$$

Florin Leon (2014). Inteligenţă artificială: maşini cu vectori suport
Tehnopress, Iaşi, ISBN 978-606-687-155-6, http://florinleon.byethost24.com

Pentru calcularea primului factor, derivăm ecuaţia 1.18:

$$\frac{\partial E_i}{\partial y_i} = -(y_{di} - y_i). \tag{1.22}$$

Pentru calcularea celui de al doilea factor, ţinem cont că funcţia de activare este liniară (ecuaţia 1.17) şi vom avea:

$$\frac{\partial y_i}{\partial w_j} = \frac{\partial \left(\sum_k x_{ik} w_k \right)}{\partial w_j} = \frac{\partial x_{ij} w_j}{\partial w_j} = x_{ij}. \tag{1.23}$$

Prin urmare:

$$\Delta w = \eta \cdot x \cdot e. \tag{1.24}$$

Această ecuaţie poartă numele de *regula delta*.

Numele vine de la notaţia alternativă care foloseşte simbolul „delta” pentru diferenţa dintre ieşirea reală şi ieşirea dorită.

Se observă că formula 1.24 este identică cu formula 1.16 de la perceptron. Acest lucru nu este surprinzător, întrucât suprafaţa de separare este aceeaşi. Modul de derivare a regulii de învăţare este însă diferit şi necesită ca funcţia de activare să fie derivabilă, ceea ce nu este cazul la perceptron, deoarece funcţiile treaptă sau semn sunt discontinue. De asemenea, metoda se poate folosi cu orice altă funcţie derivabilă, de exemplu pentru funcţiile sigmoide în cazul antrenării reţelelor de tip perceptron multistrat.

Pseudocodul algoritmului de învăţare pentru adaline, foarte asemănător cu algoritmul de învăţare pentru perceptron, este următorul.

se iniţializează toate ponderile w_i cu valori aleatorii din intervalul [–0,5, 0,5]
se iniţializează rata de învăţare eta cu o valoare din intervalul (0, 1], de exemplu 0,1
se iniţializează numărul maxim de epoci P, de exemplu 100
se iniţializează eroarea maximă admisă E_{max}, de exemplu 0,001
p = 0 *// numărul epocii curente*
mse = 0 *// eroarea pătratică medie a epocii curente*

repetă cât timp p < P **şi** mse > E_{max}
{
  mse = 0
  pentru fiecare vector de antrenare x_i cu i = 1..N
  {
    $y_i = sum(x_{ij} * w_j)$ cu j = 1..n+1
    $e = y_{di} – y_i$
    $mse = mse + e^2$
    pentru fiecare intrare j = 1..n+1
      $w_j = w_j + eta * x_{ij} * e$
  }
  mse = mse / (2 * N)
  p = p + 1
}

Algoritmul converge destul de repede către o soluţie bună. Dezavantajul principal este faptul că, spre deosebire de perceptron, convergenţa nu este garantată şi algoritmul poate să nu conveargă chiar dacă clasele sunt separabile liniar. Antrenarea perceptronului se opreşte imediat ce vectorii de antrenare sunt clasificaţi corect, chiar dacă unii se găsesc la graniţa suprafeţei de separare. Regula delta, prin minimizarea erorii medii pătratice, conduce la soluţii în care vectorii de antrenare sunt mai depărtaţi de suprafaţa de separare (Georgiou, 1997).

Florin Leon (2014). Inteligenţă artificială: maşini cu vectori suport
Tehnopress, Iaşi, ISBN 978-606-687-155-6, http://florinleon.byethost24.com

IA11. Reţele neuronale (I)
Suport de curs

Selecţie materiale: prof. dr. ing. Florin Leon

What is a neural network?

A neural network can be defined as a model of reasoning based on the human brain. The brain consists of a densely interconnected set of nerve cells, or basic information-processing units, called **neurons**. The human brain incorporates nearly 10 billion neurons and 60 trillion connections, **synapses**, between them (Shepherd and Koch, 1990). By using multiple neurons simultaneously, the brain can perform its functions much faster than the fastest computers in existence today.

Although each neuron has a very simple structure, an army of such elements constitutes a tremendous processing power. A neuron consists of a cell body, **soma**, a number of fibres called **dendrites**, and a single long fibre called the **axon**. While dendrites branch into a network around the soma, the axon stretches out to the dendrites and somas of other neurons. Figure 6.1 is a schematic drawing of a neural network.

Signals are propagated from one neuron to another by complex electro-chemical reactions. Chemical substances released from the synapses cause a change in the electrical potential of the cell body. When the potential reaches its threshold, an electrical pulse, **action potential**, is sent down through the axon. The pulse spreads out and eventually reaches synapses, causing them to increase or decrease their potential. However, the most interesting finding is that a neural network exhibits **plasticity**. In response to the stimulation pattern, neurons demonstrate long-term changes in the strength of their connections. Neurons also can form new connections with other neurons. Even entire collections of neurons may sometimes migrate from one place to another. These mechanisms form the basis for learning in the brain.

Our brain can be considered as a highly complex, nonlinear and parallel information-processing system. Information is stored and processed in a neural network simultaneously throughout the whole network, rather than at specific locations. In other words, in neural networks, both data and its processing are **global** rather than local.

Owing to the plasticity, connections between neurons leading to the 'right answer' are strengthened while those leading to the 'wrong answer' weaken. As a result, neural networks have the ability to learn through experience.

Learning is a fundamental and essential characteristic of biological neural networks. The ease and naturalness with which they can learn led to attempts to emulate a biological neural network in a computer.

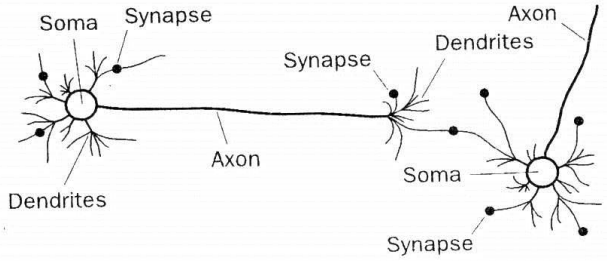


Figure 6.1 Biological neural network

Although a present-day artificial neural network (ANN) resembles the human brain much as a paper plane resembles a supersonic jet, it is a big step forward. ANNs are capable of 'learning', that is, they use experience to improve their performance. When exposed to a sufficient number of samples, ANNs can generalise to others they have not yet encountered. They can recognise hand-written characters, identify words in human speech, and detect explosives at airports. Moreover, ANNs can observe patterns that human experts fail to recognise. For example, Chase Manhattan Bank used a neural network to examine an array of information about the use of stolen credit cards – and discovered that the most suspicious sales were for women's shoes costing between $40 and $80.

How do artificial neural nets model the brain?

An artificial neural network consists of a number of very simple and highly interconnected processors, also called neurons, which are analogous to the biological neurons in the brain. The neurons are connected by weighted links passing signals from one neuron to another. Each neuron receives a number of input signals through its connections; however, it never produces more than a single output signal. The output signal is transmitted through the neuron's outgoing connection (corresponding to the biological axon). The outgoing connection, in turn, splits into a number of branches that transmit the same signal (the signal is not divided among these branches in any way). The outgoing branches terminate at the incoming connections of other neurons in the network. Figure 6.2 represents connections of a typical ANN, and Table 6.1 shows the analogy between biological and artificial neural networks (Medsker and Liebowitz, 1994).

How does an artificial neural network 'learn'?

The neurons are connected by **links**, and each link has a **numerical weight** associated with it. Weights are the basic means of long-term memory in ANNs. They express the strength, or in other words importance, of each neuron input. A neural network 'learns' through repeated adjustments of these weights.

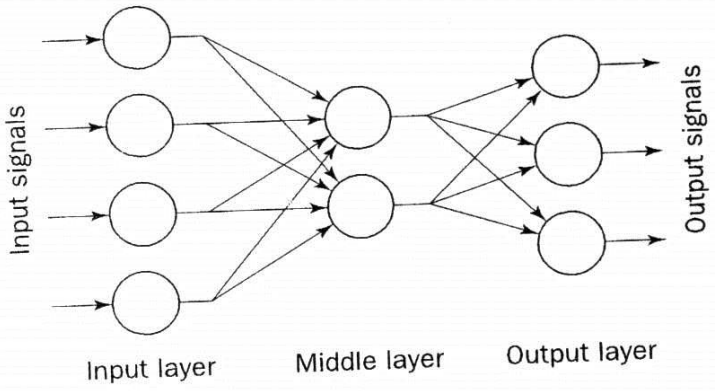


Figure 6.2 Architecture of a typical artificial neural network

Table 6.1 Analogy between biological and artificial neural networks

Biological neural network	Artificial neural network
Soma	Neuron
Dendrite	Input
Axon	Output
Synapse	Weight

But does the neural network know how to adjust the weights?

As shown in Figure 6.2, a typical ANN is made up of a hierarchy of layers, and the neurons in the networks are arranged along these layers. The neurons connected to the external environment form input and output layers. The weights are modified to bring the network input/output behaviour into line with that of the environment.

Each neuron is an elementary information-processing unit. It has a means of computing its **activation level** given the inputs and numerical weights.

To build an artificial neural network, we must decide first how many neurons are to be used and how the neurons are to be connected to form a network. In other words, we must first choose the network architecture. Then we decide which learning algorithm to use. And finally we train the neural network, that is, we initialise the weights of the network and update the weights from a set of training examples.

Let us begin with a neuron, the basic building element of an ANN.

6.2 The neuron as a simple computing element

A neuron receives several signals from its input links, computes a new activation level and sends it as an output signal through the output links. The input signal can be raw data or outputs of other neurons. The output signal can be either a final solution to the problem or an input to other neurons. Figure 6.3 shows a typical neuron.

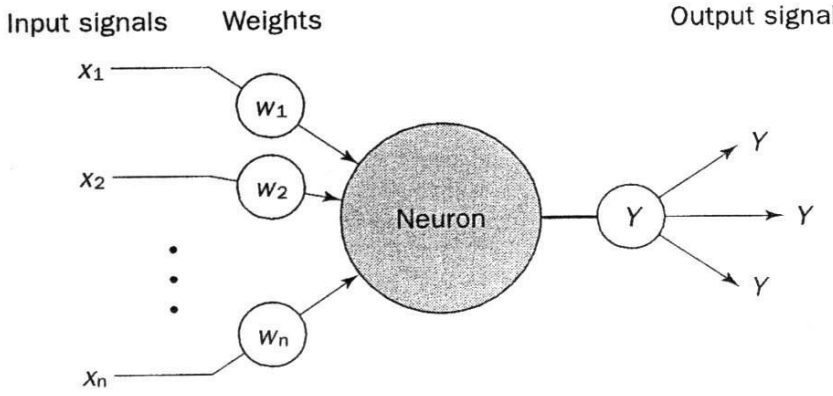


Figure 6.3 Diagram of a neuron

How does the neuron determine its output?

In 1943, Warren McCulloch and Walter Pitts proposed a very simple idea that is still the basis for most artificial neural networks.

The neuron computes the weighted sum of the input signals and compares the result with a threshold value, θ. If the net input is less than the threshold, the neuron output is -1. But if the net input is greater than or equal to the threshold, the neuron becomes activated and its output attains a value $+1$ (McCulloch and Pitts, 1943).

In other words, the neuron uses the following transfer or **activation function**:

$$X = \sum_{i=1}^{n} x_i w_i \tag{6.1}$$

$$Y = \begin{cases} +1 & \text{if } X \geqslant \theta \\ -1 & \text{if } X < \theta \end{cases},$$

where X is the net weighted input to the neuron, x_i is the value of input i, w_i is the weight of input i, n is the number of neuron inputs, and Y is the output of the neuron.

This type of activation function is called a **sign function**.

Thus the actual output of the neuron with a sign activation function can be represented as

$$Y = sign\left[\sum_{i=1}^{n} x_i w_i - \theta\right] \tag{6.2}$$

Is the sign function the only activation function used by neurons?

Many activation functions have been tested, but only a few have found practical applications. Four common choices – the step, sign, linear and sigmoid functions – are illustrated in Figure 6.4.

The **step** and **sign** activation functions, also called **hard limit functions**, are often used in decision-making neurons for classification and pattern recognition tasks.

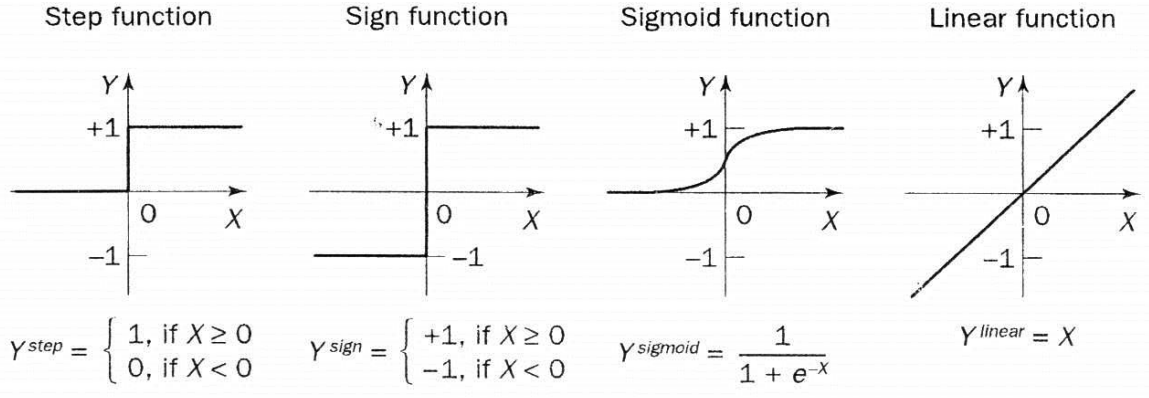


Figure 6.4 Activation functions of a neuron

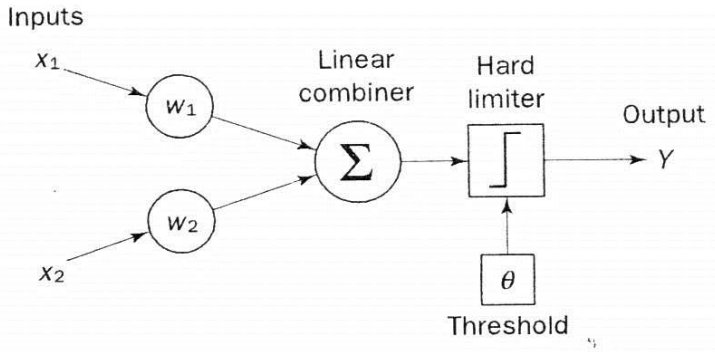


Figure 6.5 Single-layer two-input perceptron

The **sigmoid function** transforms the input, which can have any value between plus and minus infinity, into a reasonable value in the range between 0 and 1. Neurons with this function are used in the back-propagation networks.

The **linear activation function** provides an output equal to the neuron weighted input. Neurons with the linear function are often used for linear approximation.

Can a single neuron learn a task?

In 1958, Frank Rosenblatt introduced a training algorithm that provided the first procedure for training a simple ANN: a **perceptron** (Rosenblatt, 1958). The perceptron is the simplest form of a neural network. It consists of a single neuron with **adjustable** synaptic weights and a **hard limiter**. A single-layer two-input perceptron is shown in Figure 6.5.

6.3 The perceptron

The operation of Rosenblatt's perceptron is based on the McCulloch and Pitts neuron model. The model consists of a linear combiner followed by a hard limiter. The weighted sum of the inputs is applied to the hard limiter, which produces an output equal to $+1$ if its input is positive and -1 if it is negative. The aim of the perceptron is to classify inputs, or in other words externally applied stimuli $x_1, x_2, \ldots, x_n$, into one of two classes, say A_1 and A_2. Thus, in the case of an elementary perceptron, the n-dimensional space is divided by a **hyperplane** into two decision regions. The hyperplane is defined by the **linearly separable** function

$$\sum_{i=1}^{n} x_i w_i - \theta = 0 \tag{6.3}$$

For the case of two inputs, x_1 and x_2, the decision boundary takes the form of a straight line shown in bold in Figure 6.6(a). Point 1, which lies above the boundary line, belongs to class A_1; and point 2, which lies below the line, belongs to class A_2. The threshold θ can be used to shift the decision boundary.

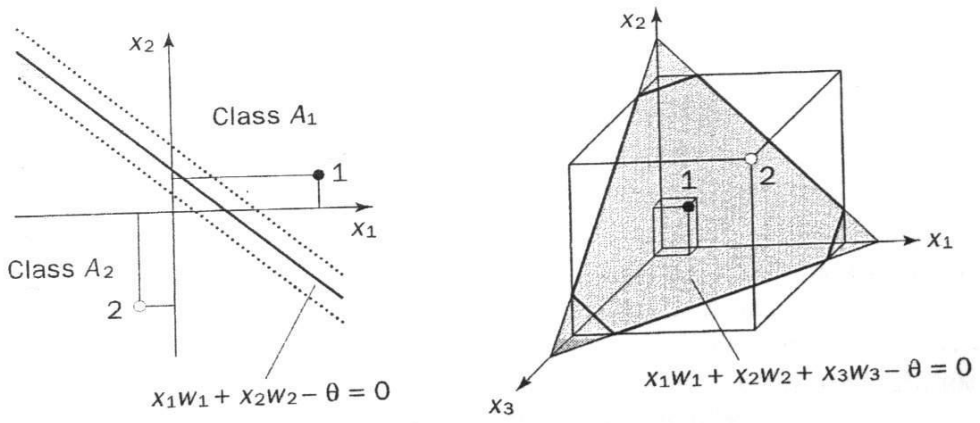


Figure 6.6 Linear separability in the perceptrons: (a) two-input perceptron; (b) three-input perceptron

With three inputs the hyperplane can still be visualised. Figure 6.6(b) shows three dimensions for the three-input perceptron. The separating plane here is defined by the equation

$$x_1 w_1 + x_2 w_2 + x_3 w_3 - \theta = 0$$

But how does the perceptron learn its classification tasks?
This is done by making small adjustments in the weights to reduce the difference between the actual and desired outputs of the perceptron. The initial weights are randomly assigned, usually in the range $[-0.5, 0.5]$, and then updated to obtain the output consistent with the training examples. For a perceptron, the process of weight updating is particularly simple. If at iteration p, the actual output is $Y(p)$ and the desired output is $Y_d(p)$, then the error is given by

$$e(p) = Y_d(p) - Y(p) \qquad \text{where } p = 1, 2, 3, \ldots \tag{6.4}$$

Iteration p here refers to the pth training example presented to the perceptron.

If the error, $e(p)$, is positive, we need to increase perceptron output $Y(p)$, but if it is negative, we need to decrease $Y(p)$. Taking into account that each perceptron input contributes $x_i(p) \times w_i(p)$ to the total input $X(p)$, we find that if input value $x_i(p)$ is positive, an increase in its weight $w_i(p)$ tends to increase perceptron output $Y(p)$, whereas if $x_i(p)$ is negative, an increase in $w_i(p)$ tends to decrease $Y(p)$. Thus, the following **perceptron learning rule** can be established:

$$w_i(p+1) = w_i(p) + \alpha \times x_i(p) \times e(p), \tag{6.5}$$

where α is the **learning rate**, a positive constant less than unity.

The perceptron learning rule was first proposed by Rosenblatt in 1960 (Rosenblatt, 1960). Using this rule we can derive the perceptron training algorithm for classification tasks.

Step 1: ***Initialisation***

Set initial weights $w_1, w_2, \ldots, w_n$ and threshold θ to random numbers in the range $[-0.5, 0.5]$.

Step 2: ***Activation***

Activate the perceptron by applying inputs $x_1(p), x_2(p), \ldots, x_n(p)$ and desired output $Y_d(p)$. Calculate the actual output at iteration $p = 1$

$$Y(p) = step\left[\sum_{i=1}^{n} x_i(p)w_i(p) - \theta\right], \tag{6.6}$$

where n is the number of the perceptron inputs, and *step* is a step activation function.

Step 3: ***Weight training***

Update the weights of the perceptron

$$w_i(p+1) = w_i(p) + \Delta w_i(p), \tag{6.7}$$

where $\Delta w_i(p)$ is the weight correction at iteration p.

The weight correction is computed by the **delta rule**:

$$\Delta w_i(p) = \alpha \times x_i(p) \times e(p) \tag{6.8}$$

Step 4: ***Iteration***

Increase iteration p by one, go back to Step 2 and repeat the process until convergence.

Can we train a perceptron to perform basic logical operations such as AND, OR or Exclusive-OR?

The truth tables for the operations AND, OR and Exclusive-OR are shown in Table 6.2. The table presents all possible combinations of values for two variables, x_1 and x_2, and the results of the operations. The perceptron must be trained to classify the input patterns.

Let us first consider the operation AND. After completing the initialisation step, the perceptron is activated by the sequence of four input patterns representing an **epoch**. The perceptron weights are updated after each activation. This process is repeated until all the weights converge to a uniform set of values. The results are shown in Table 6.3.

Table 6.2 Truth tables for the basic logical operations

Input variables		AND	OR	Exclusive-OR
x_1	x_2	$x_1 \cap x_2$	$x_1 \cup x_2$	$x_1 \oplus x_2$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

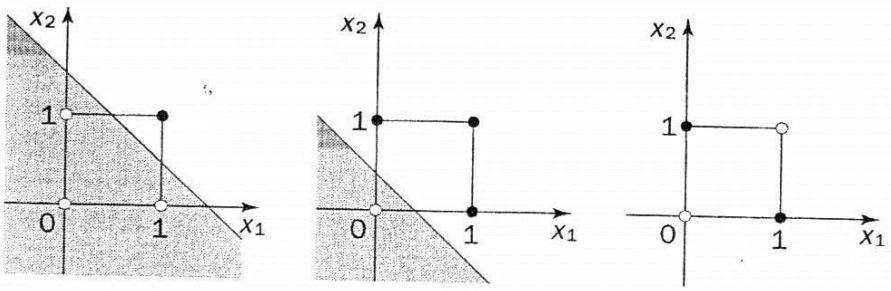
Table 6.3 Example of perceptron learning: the logical operation AND

Epoch	Inputs		Desired output	Initial weights		Actual output	Error	Final weights	
	x_1	x_2	Y_d	w_1	w_2	Y	e	w_1	w_2
1	0	0	0	0.3	−0.1	0	0	0.3	−0.1
	0	1	0	0.3	−0.1	0	0	0.3	−0.1
	1	0	0	0.3	−0.1	1	−1	0.2	−0.1
	1	1	1	0.2	−0.1	0	1	0.3	0.0
2	0	0	0	0.3	0.0	0	0	0.3	0.0
	0	1	0	0.3	0.0	0	0	0.3	0.0
	1	0	0	0.3	0.0	1	−1	0.2	0.0
	1	1	1	0.2	0.0	1	0	0.2	0.0
3	0	0	0	0.2	0.0	0	0	0.2	0.0
	0	1	0	0.2	0.0	0	0	0.2	0.0
	1	0	0	0.2	0.0	1	−1	0.1	0.0
	1	1	1	0.1	0.0	0	1	0.2	0.1
4	0	0	0	0.2	0.1	0	0	0.2	0.1
	0	1	0	0.2	0.1	0	0	0.2	0.1
	1	0	0	0.2	0.1	1	−1	0.1	0.1
	1	1	1	0.1	0.1	1	0	0.1	0.1
5	0	0	0	0.1	0.1	0	0	0.1	0.1
	0	1	0	0.1	0.1	0	0	0.1	0.1
	1	0	0	0.1	0.1	0	0	0.1	0.1
	1	1	1	0.1	0.1	1	0	0.1	0.1

Threshold: $\theta = 0.2$; learning rate: $\alpha = 0.1$.

In a similar manner, the perceptron can learn the operation OR. However, a single-layer perceptron cannot be trained to perform the operation Exclusive-OR.

A little geometry can help us to understand why this is. Figure 6.7 represents the AND, OR and Exclusive-OR functions as two-dimensional plots based on the values of the two inputs. Points in the input space where the function output is 1 are indicated by black dots, and points where the output is 0 are indicated by white dots.



(a) AND ($x_1 \cap x_2$) (b) OR ($x_1 \cup x_2$) (c) Exclusive-OR ($x_1 \oplus x_2$)

Figure 6.7 Two-dimensional plots of basic logical operations

In Figures 6.7(a) and (b), we can draw a line so that black dots are on one side and white dots on the other, but dots shown in Figure 6.7(c) are not separable by a single line. A perceptron is able to represent a function only if there is some line that separates all the black dots from all the white dots. Such functions are called **linearly separable**. Therefore, a perceptron can learn the operations AND and OR, but not Exclusive-OR.

But why can a perceptron learn only linearly separable functions?

The fact that a perceptron can learn only linearly separable functions directly follows from Eq. (6.1). The perceptron output Y is 1 only if the total weighted input X is greater than or equal to the threshold value θ. This means that the entire input space is divided in two along a boundary defined by $X = \theta$. For example, a separating line for the operation AND is defined by the equation

$$x_1 w_1 + x_2 w_2 = \theta$$

If we substitute values for weights w_1 and w_2 and threshold θ given in Table 6.3, we obtain one of the possible separating lines as

$$0.1x_1 + 0.1x_2 = 0.2$$

or

$$x_1 + x_2 = 2$$

Thus, the region below the boundary line, where the output is 0, is given by

$$x_1 + x_2 - 2 < 0,$$

and the region above this line, where the output is 1, is given by

$$x_1 + x_2 - 2 \geqslant 0$$

The fact that a perceptron can learn only linear separable functions is rather bad news, because there are not many such functions.

Can we do better by using a sigmoidal or linear element in place of the hard limiter?

Single-layer perceptrons make decisions in the same way, regardless of the activation function used by the perceptron (Shynk, 1990; Shynk and Bershad, 1992). It means that a single-layer perceptron can classify only linearly separable patterns, regardless of whether we use a hard-limit or soft-limit activation function.

The computational limitations of a perceptron were mathematically analysed in Minsky and Papert's famous book *Perceptrons* (Minsky and Papert, 1969). They proved that Rosenblatt's perceptron cannot make global generalisations on the basis of examples learned locally. Moreover, Minsky and Papert concluded that

the limitations of a single-layer perceptron would also hold true for multilayer neural networks. This conclusion certainly did not encourage further research on artificial neural networks.

How do we cope with problems which are not linearly separable?

To cope with such problems we need multilayer neural networks. In fact, history has proved that the limitations of Rosenblatt's perceptron can be overcome by advanced forms of neural networks, for example multilayer perceptrons trained with the back-propagation algorithm.

6.4 Multilayer neural networks

A multilayer perceptron is a feedforward neural network with one or more hidden layers. Typically, the network consists of an **input layer** of source neurons, at least one middle or **hidden layer** of computational neurons, and an **output layer** of computational neurons. The input signals are propagated in a forward direction on a layer-by-layer basis. A multilayer perceptron with two hidden layers is shown in Figure 6.8.

But why do we need a hidden layer?

Each layer in a multilayer neural network has its own specific function. The input layer accepts input signals from the outside world and redistributes these signals to all neurons in the hidden layer. Actually, the input layer rarely includes computing neurons, and thus does not process input patterns. The output layer accepts output signals, or in other words a stimulus pattern, from the hidden layer and establishes the output pattern of the entire network.

Neurons in the hidden layer detect the features; the weights of the neurons represent the features hidden in the input patterns. These features are then used by the output layer in determining the output pattern.

With one hidden layer, we can represent any continuous function of the input signals, and with two hidden layers even discontinuous functions can be represented.

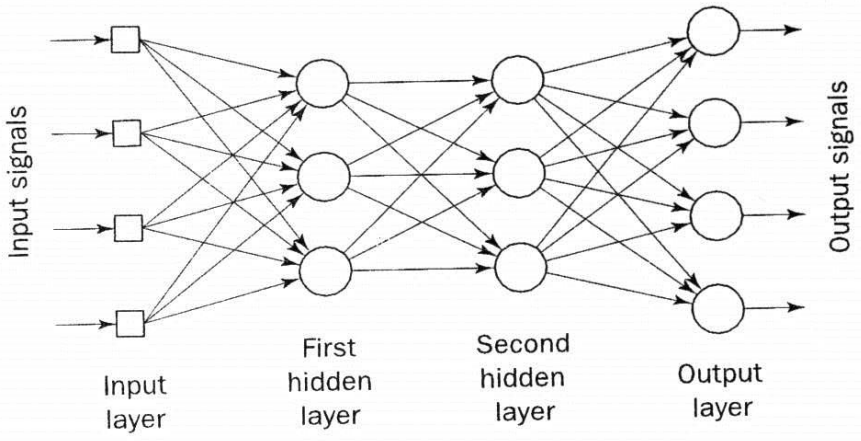


Figure 6.8 Multilayer perceptron with two hidden layers

Why is a middle layer in a multilayer network called a 'hidden' layer? What does this layer hide?

A hidden layer 'hides' its desired output. Neurons in the hidden layer cannot be observed through the input/output behaviour of the network. There is no obvious way to know what the desired output of the hidden layer should be. In other words, the desired output of the hidden layer is determined by the layer itself.

Can a neural network include more than two hidden layers?

Commercial ANNs incorporate three and sometimes four layers, including one or two hidden layers. Each layer can contain from 10 to 1000 neurons. Experimental neural networks may have five or even six layers, including three or four hidden layers, and utilise millions of neurons, but most practical applications use only three layers, because each additional layer increases the computational burden exponentially.

How do multilayer neural networks learn?

More than a hundred different learning algorithms are available, but the most popular method is back-propagation. This method was first proposed in 1969 (Bryson and Ho, 1969), but was ignored because of its demanding computations. Only in the mid-1980s was the back-propagation learning algorithm rediscovered.

Learning in a multilayer network proceeds the same way as for a perceptron. A training set of input patterns is presented to the network. The network computes its output pattern, and if there is an error – or in other words a difference between actual and desired output patterns – the weights are adjusted to reduce this error.

In a perceptron, there is only one weight for each input and only one output. But in the multilayer network, there are many weights, each of which contributes to more than one output.

How can we assess the blame for an error and divide it among the contributing weights?

In a back-propagation neural network, the learning algorithm has two phases. First, a training input pattern is presented to the network input layer. The network then propagates the input pattern from layer to layer until the output pattern is generated by the output layer. If this pattern is different from the desired output, an error is calculated and then propagated backwards through the network from the output layer to the input layer. The weights are modified as the error is propagated.

As with any other neural network, a back-propagation one is determined by the connections between neurons (the network's architecture), the activation function used by the neurons, and the learning algorithm (or the learning law) that specifies the procedure for adjusting weights.

Typically, a back-propagation network is a multilayer network that has three or four layers. The layers are **fully connected**, that is, every neuron in each layer is connected to every other neuron in the adjacent forward layer.

A neuron determines its output in a manner similar to Rosenblatt's perceptron. First, it computes the net weighted input as before:

$$X = \sum_{i=1}^{n} x_i w_i - \theta,$$

where n is the number of inputs, and θ is the threshold applied to the neuron.

Next, this input value is passed through the activation function. However, unlike a percepron, neurons in the back-propagation network use a sigmoid activation function:

$$Y^{sigmoid} = \frac{1}{1 + e^{-X}} \tag{6.9}$$

The derivative of this function is easy to compute. It also guarantees that the neuron output is bounded between 0 and 1.

What about the learning law used in the back-propagation networks?

To derive the back-propagation learning law, let us consider the three-layer network shown in Figure 6.9. The indices i, j and k here refer to neurons in the input, hidden and output layers, respectively.

Input signals, $x_1, x_2, \ldots, x_n$, are propagated through the network from left to right, and error signals, $e_1, e_2, \ldots, e_l$, from right to left. The symbol w_{ij} denotes the weight for the connection between neuron i in the input layer and neuron j in the hidden layer, and the symbol w_{jk} the weight between neuron j in the hidden layer and neuron k in the output layer.

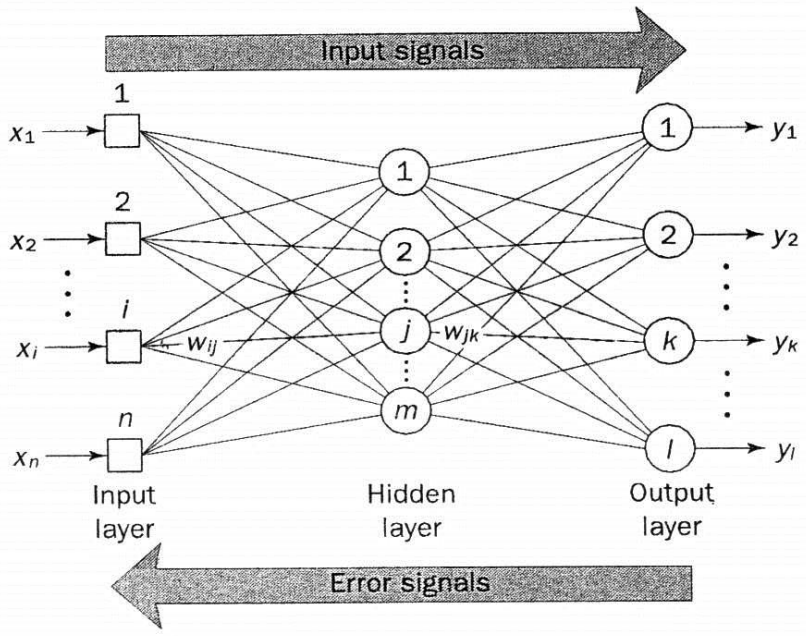


Figure 6.9 Three-layer back-propagation neural network

To propagate error signals, we start at the output layer and work backward to the hidden layer. The error signal at the output of neuron k at iteration p is defined by

$$e_k(p) = y_{d,k}(p) - y_k(p), \tag{6.10}$$

where $y_{d,k}(p)$ is the desired output of neuron k at iteration p.

Neuron k, which is located in the output layer, is supplied with a desired output of its own. Hence, we may use a straightforward procedure to update weight w_{jk}. In fact, the rule for updating weights at the output layer is similar to the perceptron learning rule of Eq. (6.7):

$$w_{jk}(p+1) = w_{jk}(p) + \Delta w_{jk}(p), \tag{6.11}$$

where $\Delta w_{jk}(p)$ is the weight correction.

When we determined the weight correction for the perceptron, we used input signal x_i. But in the multilayer network, the inputs of neurons in the output layer are different from the inputs of neurons in the input layer.

As we cannot apply input signal x_i, what should we use instead?

We use the output of neuron j in the hidden layer, y_j, instead of input x_i. The weight correction in the multilayer network is computed by (Fu, 1994):

$$\Delta w_{jk}(p) = \alpha \times y_j(p) \times \delta_k(p), \tag{6.12}$$

where $\delta_k(p)$ is the error gradient at neuron k in the output layer at iteration p.

What is the error gradient?

The error gradient is determined as the derivative of the sigmoid activation function, F', multiplied by the error at the neuron output.

Thus, for neuron k in the output layer, we obtain

$$\delta_k(p) = F'[X_k(p)] \times e_k(p), \tag{6.13}$$

where $X_k(p)$ is the net weighted input to neuron k at iteration p:

$$X_k(p) = \sum_{j=1}^{m} x_{jk}(p) \times w_{jk}(p) - \theta_k,$$

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

Thus, Eq. (6.13) can be represented as

$$\delta_k(p) = y_k(p) \times [1 - y_k(p)] \times e_k(p), \tag{6.14}$$

where

$$y_k(p) = \frac{1}{1 + e^{-X_k(p)}}.$$

How can we determine the weight correction for a neuron in the hidden layer?

To calculate the weight correction for the hidden layer, we can apply the same equation as for the output layer:

$$\Delta w_{ij}(p) = \alpha \times x_i(p) \times \delta_j(p), \tag{6.15}$$

where $\delta_j(p)$ represents the error gradient at neuron j in the hidden layer:

$$\delta_j(p) = y_j(p) \times [1 - y_j(p)] \times \sum_{k=1}^{l} \delta_k(p) w_{jk}(p),$$

where l is the number of neurons in the output layer;

$$y_j(p) = \frac{1}{1 + e^{-X_j(p)}};$$

$$X_j(p) = \sum_{i=1}^{n} x_i(p) \times w_{ij}(p) - \theta_j;$$

and n is the number of neurons in the input layer.

Now we can derive the back-propagation training algorithm.

Step 1: ***Initialisation***

Set all the weights and threshold levels of the network to random numbers uniformly distributed inside a small range (Haykin, 1994):

$$\left(-\frac{2.4}{F_i}, +\frac{2.4}{F_i}\right),$$

where F_i is the total number of inputs of neuron i in the network. The weight initialisation is done on a neuron-by-neuron basis.

Step 2: ***Activation***

Activate the back-propagation neural network by applying inputs $x_1(p), x_2(p), \ldots, x_n(p)$ and desired outputs $y_{d,1}(p), y_{d,2}(p), \ldots, y_{d,n}(p)$.

(a) Calculate the actual outputs of the neurons in the hidden layer:

$$y_j(p) = sigmoid\left[\sum_{i=1}^{n} x_i(p) \times w_{ij}(p) - \theta_j\right],$$

where n is the number of inputs of neuron j in the hidden layer, and *sigmoid* is the sigmoid activation function.

(b) Calculate the actual outputs of the neurons in the output layer:

$$y_k(p) = sigmoid\left[\sum_{j=1}^{m} x_{jk}(p) \times w_{jk}(p) - \theta_k\right],$$

where m is the number of inputs of neuron k in the output layer.

Step 3: ***Weight training***

Update the weights in the back-propagation network propagating backward the errors associated with output neurons.

(a) Calculate the error gradient for the neurons in the output layer:

$$\delta_k(p) = y_k(p) \times [1 - y_k(p)] \times e_k(p)$$

where

$$e_k(p) = y_{d,k}(p) - y_k(p)$$

Calculate the weight corrections:

$$\Delta w_{jk}(p) = \alpha \times y_j(p) \times \delta_k(p)$$

Update the weights at the output neurons:

$$w_{jk}(p+1) = w_{jk}(p) + \Delta w_{jk}(p)$$

(b) Calculate the error gradient for the neurons in the hidden layer:

$$\delta_j(p) = y_j(p) \times [1 - y_j(p)] \times \sum_{k=1}^{l} \delta_k(p) \times w_{jk}(p)$$

Calculate the weight corrections:

$$\Delta w_{ij}(p) = \alpha \times x_i(p) \times \delta_j(p)$$

Update the weights at the hidden neurons:

$$w_{ij}(p+1) = w_{ij}(p) + \Delta w_{ij}(p)$$

Step 4: ***Iteration***

Increase iteration p by one, go back to Step 2 and repeat the process until the selected error criterion is satisfied.

As an example, we may consider the three-layer back-propagation network shown in Figure 6.10. Suppose that the network is required to perform logical operation Exclusive-OR. Recall that a single-layer perceptron could not do this operation. Now we will apply the three-layer net.

Neurons 1 and 2 in the input layer accept inputs x_1 and x_2, respectively, and redistribute these inputs to the neurons in the hidden layer without any processing:

$x_{13} = x_{14} = x_1$ and $x_{23} = x_{24} = x_2$.

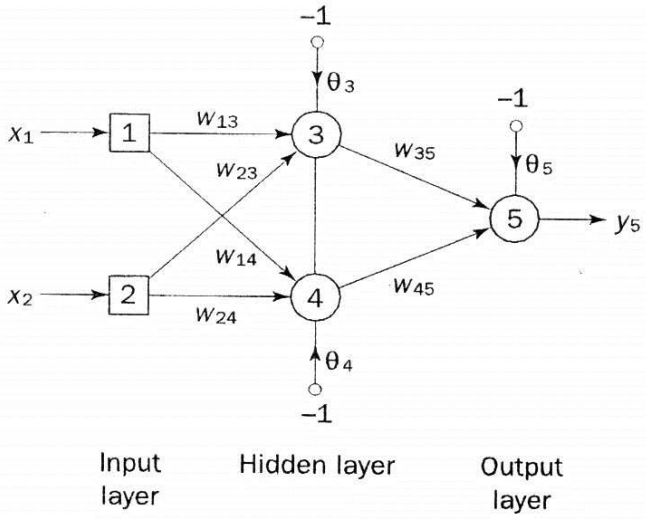


Figure 6.10 Three-layer network for solving the Exclusive-OR operation

The effect of the threshold applied to a neuron in the hidden or output layer is represented by its weight, θ, connected to a fixed input equal to -1.

The initial weights and threshold levels are set randomly as follows:

$w_{13} = 0.5$, $w_{14} = 0.9$, $w_{23} = 0.4$, $w_{24} = 1.0$, $w_{35} = -1.2$, $w_{45} = 1.1$, $\theta_3 = 0.8$, $\theta_4 = -0.1$ and $\theta_5 = 0.3$.

Consider a training set where inputs x_1 and x_2 are equal to 1 and desired output $y_{d,5}$ is 0. The actual outputs of neurons 3 and 4 in the hidden layer are calculated as

$$y_3 = sigmoid\,(x_1 w_{13} + x_2 w_{23} - \theta_3) = 1/[1 + e^{-(1\times0.5+1\times0.4-1\times0.8)}] = 0.5250$$

$$y_4 = sigmoid\,(x_1 w_{14} + x_2 w_{24} - \theta_4) = 1/[1 + e^{-(1\times0.9+1\times1.0+1\times0.1)}] = 0.8808$$

Now the actual output of neuron 5 in the output layer is determined as

$$y_5 = sigmoid\,(y_3 w_{35} + y_4 w_{45} - \theta_5) = 1/[1 + e^{-(-0.5250\times1.2+0.8808\times1.1-1\times0.3)}] = 0.5097$$

Thus, the following error is obtained:

$$e = y_{d,5} - y_5 = 0 - 0.5097 = -0.5097$$

The next step is weight training. To update the weights and threshold levels in our network, we propagate the error, e, from the output layer backward to the input layer.

First, we calculate the error gradient for neuron 5 in the output layer:

$$\delta_5 = y_5(1 - y_5)e = 0.5097 \times (1 - 0.5097) \times (-0.5097) = -0.1274$$

Then we determine the weight corrections assuming that the learning rate parameter, α, is equal to 0.1:

$$\Delta w_{35} = \alpha \times y_3 \times \delta_5 = 0.1 \times 0.5250 \times (-0.1274) = -0.0067$$

$$\Delta w_{45} = \alpha \times y_4 \times \delta_5 = 0.1 \times 0.8808 \times (-0.1274) = -0.0112$$

$$\Delta \theta_5 = \alpha \times (-1) \times \delta_5 = 0.1 \times (-1) \times (-0.1274) = -0.0127$$

Next we calculate the error gradients for neurons 3 and 4 in the hidden layer:

$$\delta_3 = y_3(1 - y_3) \times \delta_5 \times w_{35} = 0.5250 \times (1 - 0.5250) \times (-0.1274) \times (-1.2) = 0.0381$$

$$\delta_4 = y_4(1 - y_4) \times \delta_5 \times w_{45} = 0.8808 \times (1 - 0.8808) \times (-0.1274) \times 1.1 = -0.0147$$

We then determine the weight corrections:

$$\Delta w_{13} = \alpha \times x_1 \times \delta_3 = 0.1 \times 1 \times 0.0381 = 0.0038$$

$$\Delta w_{23} = \alpha \times x_2 \times \delta_3 = 0.1 \times 1 \times 0.0381 = 0.0038$$

$$\Delta \theta_3 = \alpha \times (-1) \times \delta_3 = 0.1 \times (-1) \times 0.0381 = -0.0038$$

$$\Delta w_{14} = \alpha \times x_1 \times \delta_4 = 0.1 \times 1 \times (-0.0147) = -0.0015$$

$$\Delta w_{24} = \alpha \times x_2 \times \delta_4 = 0.1 \times 1 \times (-0.0147) = -0.0015$$

$$\Delta \theta_4 = \alpha \times (-1) \times \delta_4 = 0.1 \times (-1) \times (-0.0147) = 0.0015$$

At last, we update all weights and threshold levels in our network:

$$w_{13} = w_{13} + \Delta w_{13} = 0.5 + 0.0038 = 0.5038$$

$$w_{14} = w_{14} + \Delta w_{14} = 0.9 - 0.0015 = 0.8985$$

$$w_{23} = w_{23} + \Delta w_{23} = 0.4 + 0.0038 = 0.4038$$

$$w_{24} = w_{24} + \Delta w_{24} = 1.0 - 0.0015 = 0.9985$$

$$w_{35} = w_{35} + \Delta w_{35} = -1.2 - 0.0067 = -1.2067$$

$$w_{45} = w_{45} + \Delta w_{45} = 1.1 - 0.0112 = 1.0888$$

$$\theta_3 = \theta_3 + \Delta \theta_3 = 0.8 - 0.0038 = 0.7962$$

$$\theta_4 = \theta_4 + \Delta \theta_4 = -0.1 + 0.0015 = -0.0985$$

$$\theta_5 = \theta_5 + \Delta \theta_5 = 0.3 + 0.0127 = 0.3127$$

The training process is repeated until the sum of squared errors is less than 0.001.

Why do we need to sum the squared errors?

The sum of the **squared** errors is a useful indicator of the network's performance. The back-propagation training algorithm attempts to minimise this criterion. When the value of the sum of squared errors in an entire pass through all

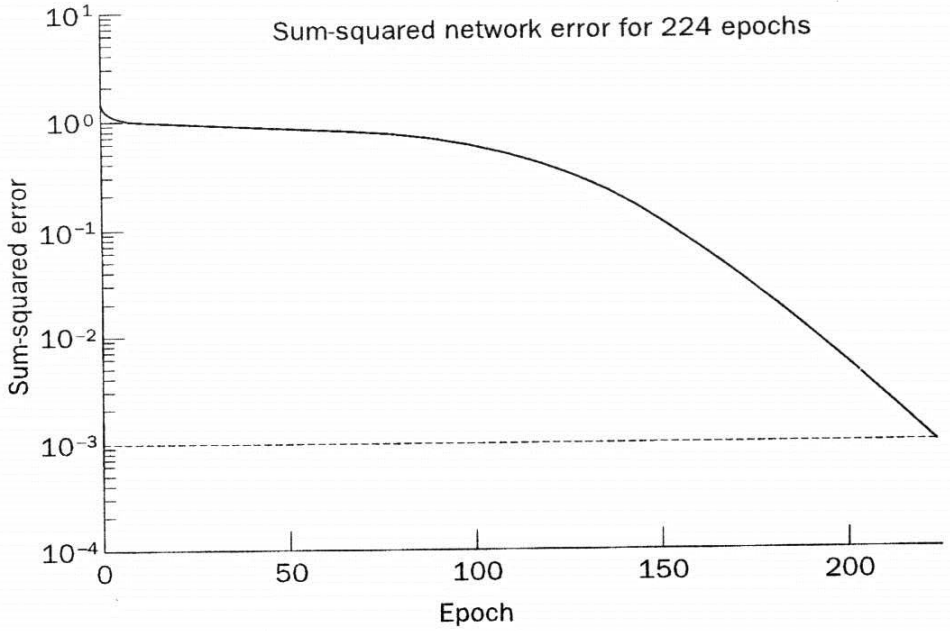


Figure 6.11 Learning curve for operation Exclusive-OR

training sets, or epoch, is **sufficiently small**, a network is considered to have **converged**. In our example, the sufficiently small sum of squared errors is defined as less than 0.001. Figure 6.11 represents a learning curve: the sum of squared errors plotted versus the number of epochs used in training. The learning curve shows how fast a network is learning.

It took 224 epochs or 896 iterations to train our network to perform the Exclusive-OR operation. The following set of final weights and threshold levels satisfied the chosen error criterion:

$$w_{13} = 4.7621,\ w_{14} = 6.3917,\ w_{23} = 4.7618,\ w_{24} = 6.3917,\ w_{35} = -10.3788,$$
$$w_{45} = 9.7691,\ \theta_3 = 7.3061,\ \theta_4 = 2.8441 \text{ and } \theta_5 = 4.5589.$$

The network has solved the problem! We may now test our network by presenting all training sets and calculating the network's output. The results are shown in Table 6.4.

Table 6.4 Final results of three-layer network learning: the logical operation Exclusive-OR

Inputs		Desired output	Actual output	Error	Sum of squared errors
x_1	x_2	y_d	y_5	e	
1	1	0	0.0155	−0.0155	0.0010
0	1	1	0.9849	0.0151	
1	0	1	0.9849	0.0151	
0	0	0	0.0175	−0.0175	

The initial weights and thresholds are set randomly. Does this mean that the same network may find different solutions?

The network obtains different weights and threshold values when it starts from different initial conditions. However, we will always solve the problem, although using a different number of iterations. For instance, when the network was trained again, we obtained the following solution:

$$w_{13} = -6.3041,\ w_{14} = -5.7896,\ w_{23} = 6.2288,\ w_{24} = 6.0088,\ w_{35} = 9.6657,$$
$$w_{45} = -9.4242,\ \theta_3 = 3.3858,\ \theta_4 = -2.8976 \text{ and } \theta_5 = -4.4859.$$

Can we now draw decision boundaries constructed by the multilayer network for operation Exclusive-OR?

It may be rather difficult to draw decision boundaries constructed by neurons with a sigmoid activation function. However, we can represent each neuron in the hidden and output layers by a McCulloch and Pitts model, using a sign function. The network in Figure 6.12 is also trained to perform the Exclusive-OR operation (Touretzky and Pomerlean, 1989; Haykin, 1994).

The positions of the decision boundaries constructed by neurons 3 and 4 in the hidden layer are shown in Figure 6.13(a) and (b), respectively. Neuron 5 in the output layer performs a linear combination of the decision boundaries formed by the two hidden neurons, as shown in Figure 6.13(c). The network in Figure 6.12 does indeed separate black and white dots and thus solves the Exclusive-OR problem.

Is back-propagation learning a good method for machine learning?

Although widely used, back-propagation learning is not immune from problems. For example, the back-propagation learning algorithm does not seem to function in the biological world (Stork, 1989). Biological neurons do not work backward to adjust the strengths of their interconnections, synapses, and thus back-propagation learning cannot be viewed as a process that emulates brain-like learning.

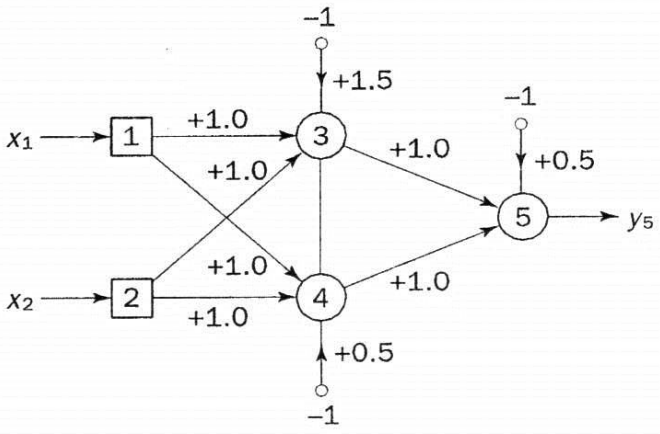


Figure 6.12 Network represented by McCulloch–Pitts model for solving the Exclusive-OR operation.

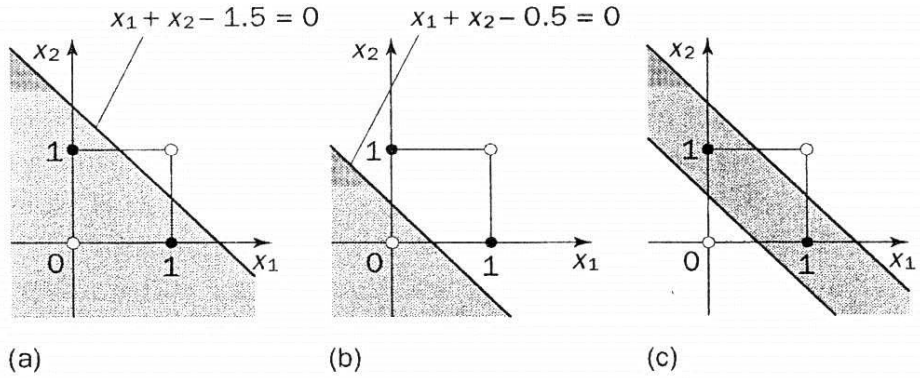


Figure 6.13 (a) decision boundary constructed by hidden neuron 3 of the network in Figure 6.12; (b) decision boundary constructed by hidden neuron 4; (c) decision boundaries constructed by the complete three-layer network

Another apparent problem is that the calculations are extensive and, as a result, training is slow. In fact, a pure back-propagation algorithm is rarely used in practical applications.

There are several possible ways to improve the computational efficiency of the back-propagation algorithm (Caudill, 1991; Jacobs, 1988; Stubbs, 1990). Some of them are discussed below.

6.5 Accelerated learning in multilayer neural networks

A multilayer network, in general, learns much faster when the sigmoidal activation function is represented by a **hyperbolic tangent,**

$$Y^{tanh} = \frac{2a}{1 + e^{-bX}} - a, \tag{6.16}$$

where a and b are constants.

Suitable values for a and b are: $a = 1.716$ and $b = 0.667$ (Guyon, 1991).

We also can accelerate training by including a **momentum term** in the delta rule of Eq. (6.12) (Rumelhart *et al.*, 1986):

$$\Delta w_{jk}(p) = \beta \times \Delta w_{jk}(p-1) + \alpha \times y_j(p) \times \delta_k(p), \tag{6.17}$$

where β is a positive number $(0 \leqslant \beta < 1)$ called the momentum constant. Typically, the momentum constant is set to 0.95.

Equation (6.17) is called the **generalised delta rule.** In a special case, when $\beta = 0$, we obtain the delta rule of Eq. (6.12).

Why do we need the momentum constant?

According to the observations made in Watrous (1987) and Jacobs (1988), the inclusion of momentum in the back-propagation algorithm has a **stabilising effect** on training. In other words, the inclusion of momentum tends to

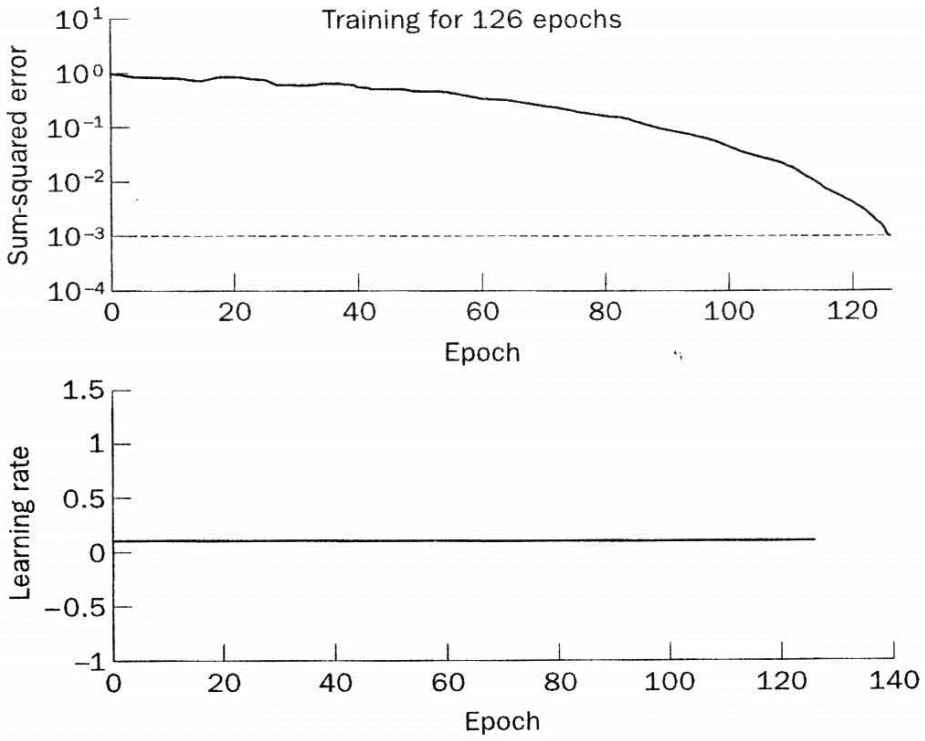


Figure 6.14 Learning with momentum

accelerate descent in the steady downhill direction, and to slow down the process when the learning surface exhibits peaks and valleys.

Figure 6.14 represents learning with momentum for operation Exclusive-OR. A comparison with a pure back-propagation algorithm shows that we reduced the number of epochs from 224 to 126.

In the delta and generalised delta rules, we use a constant and rather small value for the learning rate parameter, α. Can we increase this value to speed up training?

One of the most effective means to accelerate the convergence of back-propagation learning is to adjust the learning rate parameter during training. The small learning rate parameter, α, causes small changes to the weights in the network from one iteration to the next, and thus leads to the smooth learning curve. On the other hand, if the learning rate parameter, α, is made larger to speed up the training process, the resulting larger changes in the weights may cause instability and, as a result, the network may become oscillatory.

To accelerate the convergence and yet avoid the danger of instability, we can apply two heuristics (Jacobs, 1988):

- **Heuristic 1.** If the change of the sum of squared errors has the same algebraic sign for several consequent epochs, then the learning rate parameter, α, should be increased.
- **Heuristic 2.** If the algebraic sign of the change of the sum of squared errors alternates for several consequent epochs, then the learning rate parameter, α, should be decreased.

Adapting the learning rate requires some changes in the back-propagation algorithm. First, the network outputs and errors are calculated from the initial learning rate parameter. If the sum of squared errors at the current epoch exceeds the previous value by more than a predefined ratio (typically 1.04), the learning rate parameter is decreased (typically by multiplying by 0.7) and new weights and thresholds are calculated. However, if the error is less than the previous one, the learning rate is increased (typically by multiplying by 1.05).

Figure 6.15 represents an example of back-propagation training with adaptive learning rate. It demonstrates that adapting the learning rate can indeed decrease the number of iterations.

Learning rate adaptation can be used together with learning with momentum. Figure 6.16 shows the benefits of applying simultaneously both techniques.

The use of momentum and adaptive learning rate significantly improves the performance of a multilayer back-propagation neural network and minimises the chance that the network can become oscillatory.

Neural networks were designed on an analogy with the brain. The brain's memory, however, works by association. For example, we can recognise a familiar face even in an unfamiliar environment within 100–200 ms. We can also recall a complete sensory experience, including sounds and scenes, when we hear only a few bars of music. The brain routinely associates one thing with another.

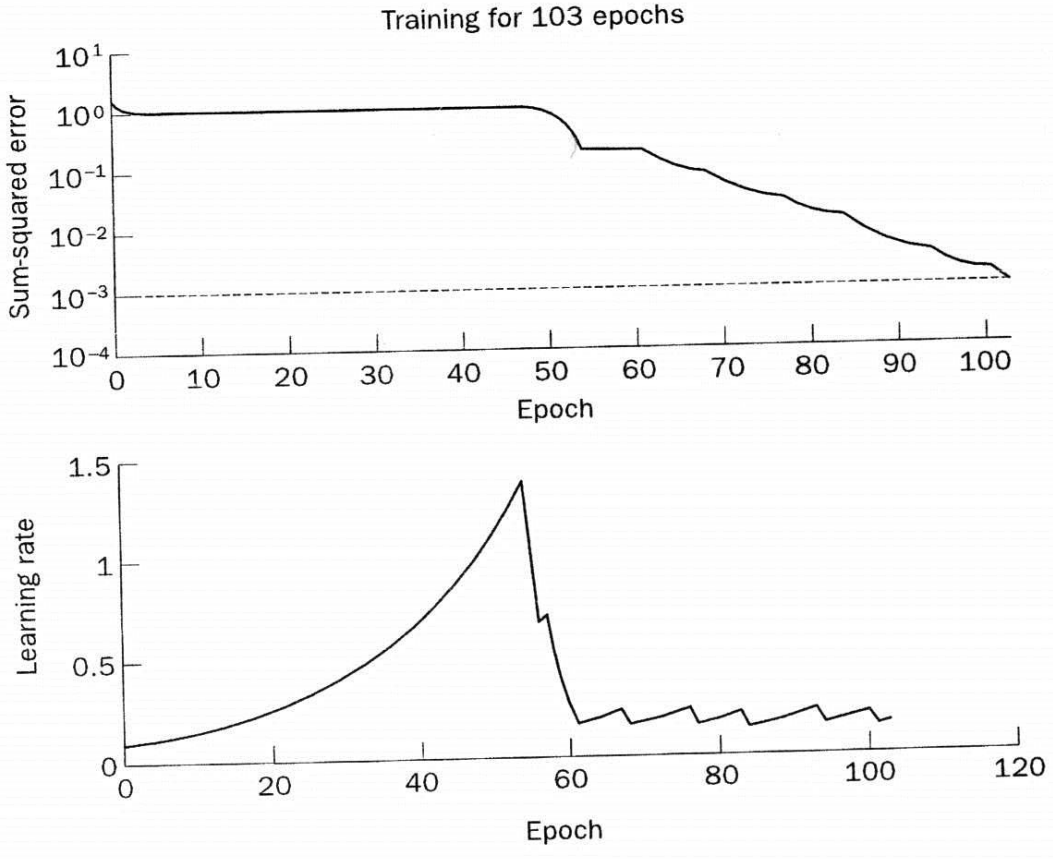


Figure 6.15 Learning with adaptive learning rate

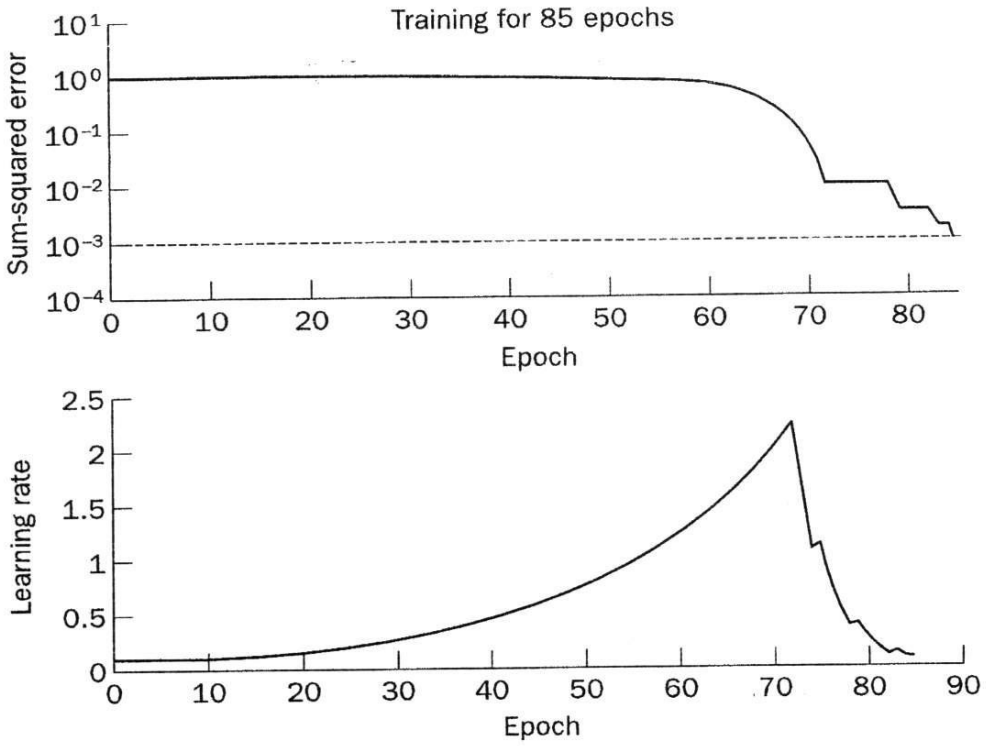


Figure 6.16 Learning with momentum and adaptive learning rate

C1.6.1 Introduction

Most of the models and algorithms described in this section are constituted of three or four layers (including the input layer). Each layer may use different neuron models, may have different topology, and may be associated with a specialized task.

In neural models described in the previous sections of this chapter, neurons are basically computing units whose output is a sigmoidal function, a Heaviside function, or some other function of its activation. The activation is the inner product between the input vector and the weight vector of the neuron. Other neural models have been proposed, based on another neural model whose output is a nonlinear decreasing function of the distance between the input vector and the weight vector: such a neuron will be called a *kernel neuron* or a *radial basis function (RBF) neuron* in this section. Neurons used in RBF neural networks, in kernel neural networks (KNNs)—also called probabilistic neural networks (PNNs)—and in the famous *self-organizing feature maps* (SOFMs) belong to this family.

Note that, in both cases, from a statistical point of view, the neuron is nothing other than a particular nonlinear regressor, whose assemblies have the interesting property of being able to model any nonlinear function.

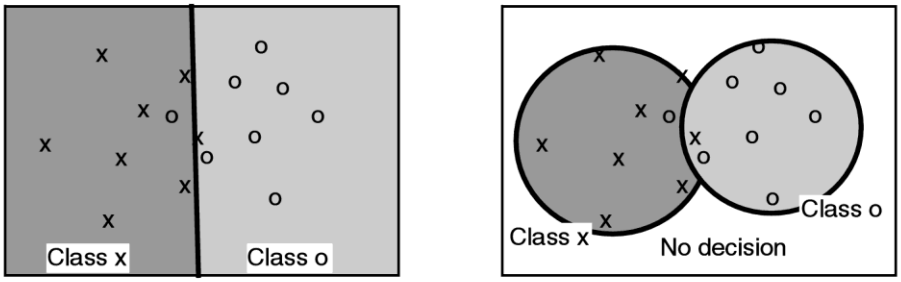


Figure C1.6.1. Boundaries defined by (*a*) hard-limiter neuron, (*b*) RBF neuron.

The difference and the interest of the two neuron models can be easily explained within the framework of classification. For the sake of simplicity, consider a simple binary classification task. In both cases,

each neuron is a basic discriminant function, which divides the observation (features or patterns) space into two parts. However, with the hard-limiter neuron the boundary is a hyperplane, while with the RBF neuron the boundary is the circumference of a hypervolume (hypersphere with Euclidean distance) (see figure C1.6.1) centered around class samples. The RBF neuron gives a local decision, and regions of the feature located too far from samples are not classified. This neuron property, which allows rejection and avoids misclassification, is preserved at network level.

This section is devoted to *supervised* composite networks.

C1.6.2 Radial basis function neural networks

C1.6.2.1 Introduction

In the neural network world, the paradigm of RBF was first introduced by Broomhead and Lowe (1988) and Moody and Darken (1989). Another major contribution was the paper by Poggio and Girosi (1990) who explained the design and interest in RBF networks with regularization theory. RBF networks can perform both classification and function approximation. For classification, the interest in RBF can be explained by the concept of ϕ*-separable* patterns proposed by Cover (1965). Concerning function approximation, theoretical results on multivariate approximation constitute the basic framework. For more details see Powell (1985), Poggio and Girosi (1990), Haykin (1994 ch 7, pp 237–44).

C1.6.2.2 Purpose of the model

RBF neural networks are general purpose approximators. In the literature, numerous applications involving function approximation as well as classification properties are encountered: time series analysis (Saha and Keeler 1990, Kadirkamanathan *et al* 1991), equalization (Cheng *et al* 1992), classification of seismic events (Chang and Lippmann 1992), handwritten digit recognition (Lee 1991), speech recognition (Lee and Lippmann 1990), adaptive control (Sanner and Slotine 1992), spectral estimation (Nedir *et al* 1993), and so on.

C1.6.2.3 Topology

An RBF network consists of three layers:

(i) the first contains simple neurons which transmit input without distortion,
(ii) the second (hidden) layer contains the RBF neurons,
(iii) neurons in the output layer are simple linear units.

Each layer is fully-connected to the next one with simple first-order connections (figure C1.6.2).

Basically, the number of input units (output units, respectively) is equal to the dimension n of the input vectors (of the output space, respectively). However, the number of output units can vary according to the coding of the outputs. For instance, for binary classification, we can choose:

(i) 1 output unit which is close to 0 for class 0, and close to 1 for class 1,
(ii) 2 output units: unit 0 is close to 1 and unit 1 is close to 0 if class 0 is decided, and vice versa.

Finally, the number N_2 of RBF units is equal to the number of samples, N, in the learning database. The weight vector between the input vector and the jth RBF unit is simply equal to the input vector of the jth samples of the database: $\boldsymbol{w}_j = \boldsymbol{x}^j$.

The output of the ith neuron of the output layer is then

$$y_i(\boldsymbol{x}) = \sum_{j=1}^{N} w_{ij}\phi(\|\boldsymbol{x} - \boldsymbol{x}^j\|) \tag{C1.6.1}$$

where $\phi(\cdot)$ is a function from $\mathcal{R}^+$ to $\mathcal{R}$, generally decreasing, $\boldsymbol{x}$ is the input vector, and $\boldsymbol{x}^j$ are input examples of the learning database. In equation (C1.6.1), the weights w_{ij} (between RBF units and output units) are tuned during the training, as we will explain in section C1.6.2.5. In what follows, for the sake

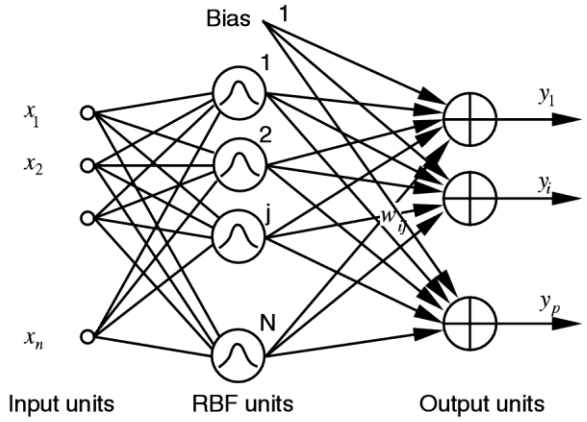


Figure C1.6.2. Topology of an RBF network.

of simplicity, we always consider 1-output networks, and we omit the index i. Equation (C1.6.1) then becomes:

$$y(\boldsymbol{x}) = \sum_{j=1}^{N} w_j \phi(\|\boldsymbol{x} - \boldsymbol{x}^j\|)\,. \tag{C1.6.2}$$

Finally, we remark that the number of RBF units becomes very large with a huge learning database: practical methods to reduce the number will also appear in section C1.6.2.5.

C1.6.2.4 Choice of function

There now remains an essential question: what radial basis function must we use? Poggio and Girosi (1990) addressed this question in the framework of multivariate interpolation with regularization, for function f from $\mathcal{R}^n$ to $\mathcal{R}$. In fact, learning consists of designing a mapping f from N empirical input/output examples $(\boldsymbol{x}^j, d^j)$, $1 \leq j \leq N$, which are currently noisy examples. It is thus an ill-posed problem in the Hadamard sense, especially since the same input can produce various outputs, in which case we must exploit other information in order to transform the problem to a well-posed problem. This can be done by looking for the function f minimizing a functional consisting of two terms:

$$\begin{aligned}\mathcal{E}[f] &= \sum_j (\|y(\boldsymbol{x}^j) - d^j\|)^2 + \lambda(\|Pf\|)^2 \\ &= \sum_j (\|f(\boldsymbol{x}^j) - d^j\|)^2 + \lambda(\|Pf\|)^2\end{aligned} \tag{C1.6.3}$$

where d^j is the (noisy) target output in response to input $\boldsymbol{x}^j$, λ is a scalar parameter (regularization parameter), and P is usually a differential operator. The first term of the functional measures the fitting on data, while the second term imposes smoothing on f. It can be shown that equation (C1.6.3) leads to a Euler–Lagrange partial differential equation, solutions of which involve Green's functions $G(\boldsymbol{x}, \boldsymbol{x}^j)$:

$$f(\boldsymbol{x}) = \frac{1}{\lambda} \sum_{j=1}^{N} (d^j - f(\boldsymbol{x}^j)) G(\boldsymbol{x}, \boldsymbol{x}^j)\,. \tag{C1.6.4}$$

The optimal choice of the Green function depends on the operator P. For instance, for one-dimensional data, P can be defined such that

$$(\|Pf\|)^2 = \int_{\mathcal{R}} \left[\frac{\mathrm{d}^2 f(x)}{\mathrm{d}x^2}\right]^2 \mathrm{d}x\,. \tag{C1.6.5}$$

In that case, the Green function is a cubic spline (Haykin 1994, pp 249–50). Furthermore, if we constrain the operator P to be invariant under rotations and translations, a solution of the Green function leads to the Gaussian RBF (Poggio and Girosi 1990):

$$G(\boldsymbol{x}, \boldsymbol{x}^j) = \exp\left(-\frac{1}{2\sigma_j^2}\|\boldsymbol{x} - \boldsymbol{x}^j\|^2\right). \tag{C1.6.6}$$

Finally, the RBF network approximation becomes

$$f(\boldsymbol{x}) = \sum_{j=1}^{N} w_j \exp\left(-\frac{1}{2\sigma_j^2}\|\boldsymbol{x} - \boldsymbol{x}^j\|^2\right) \tag{C1.6.7}$$

that is, a linear superposition of Gaussian RBF, whose centers are samples $\boldsymbol{x}^j$ and variances are σ_j^2.

C1.6.2.5 Learning

Three types of parameter are adjusted by training. Weights w_j and variances σ_j^2 are trained by supervised learning. Finally, to avoid too large a complexity, the number of RBF units may be reduced by selecting, usually unsupervised, a small but representative number of samples in the database. RBF networks being universal approximators, there is no restriction on either inputs or outputs, which may be integer as well as real.

Weight computation without center selection. In the simplest case, all the RBFs have the same width. The location of the RBF and the weight w_j must be computed. If the number of samples, N, in the learning database is not too large, one chooses N RBF units, each one being centered on each sample. The N weights w_j, $1 \leq j \leq N$, are solutions of the set of N linear equation:

$$y(\boldsymbol{x}^k) = d^k = \sum_{j=1}^{N} w_j \phi(\|\boldsymbol{x}^k - \boldsymbol{x}^j\|) \qquad 1 \leq k \leq N \tag{C1.6.8}$$

which can be written:

$$\Phi \boldsymbol{w} = \boldsymbol{y} \tag{C1.6.9}$$

where $\Phi = (\phi_{kj})$ is an $N \times N$ matrix such that $\phi_{kj} = \phi(\|\boldsymbol{x}^k - \boldsymbol{x}^j\|)$, $\boldsymbol{w}$ is the unknown vector $(w_1, w_2, \ldots, w_N)^{\mathrm{T}}$ and $\boldsymbol{y}$ is the target vector $(d^1, d^2, \ldots, d^N)^{\mathrm{T}}$.

According to Light's theorem (Light 1992), the matrix Φ is positive definite if the input vectors $\boldsymbol{x}^1$, $\boldsymbol{x}^2$, $\ldots$, $\boldsymbol{x}^N$ are distinct. If so, the above set of equations has a unique solution. Note that such a set of N equations must be solved for each output unit.

Weight computation with center selection. If N is large, to avoid computation and memory being too large, one selects $N_2 \ll N$ samples in the learning database, which will be associated with N_2 RBF units. Such RBF neural networks are usually called *generalized RBF* (GRBF) neural networks. The selection of representative samples is usually done by simple *vector quantization* (VQ) algorithms or *Kohonen's algorithm*, which are unsupervised algorithms. We denote these N_2 new samples, usually different from database samples, by $\boldsymbol{c}_j$. Then, for each output unit, we have a set of N equations with N_2 unknowns:

$$y(\boldsymbol{x}^k) = d^k = \sum_{j=1}^{N_2} w_j \phi(\|\boldsymbol{x}^k - \boldsymbol{c}_j\|) \qquad 1 \leq k \leq N \tag{C1.6.10}$$

or again:

$$\Phi \boldsymbol{w} = \boldsymbol{y}\,. \tag{C1.6.11}$$

The matrix Φ is now rectangular, $N \times N_2$. An optimal solution, in the mean-square error sense, is then given by

$$\boldsymbol{w} = \Phi^{+}\boldsymbol{y} \tag{C1.6.12}$$

where Φ^{+} denotes the pseudo-inverse matrix of Φ. This pseudo-inverse matrix can be computed by direct computation, using the relation $\Phi^{+} = (\Phi^{\mathrm{T}}\Phi)^{-1}\Phi^{\mathrm{T}}$, iterative algorithm or adaptive (least-square) algorithms.

Supervised selection of centers. Supervised selection of centers was first proposed by Poggio and Girosi (1990), and is more efficient than unsupervised selection (Wettschereck and Dietterich 1992). The idea is based on gradient descent of the cost function:

$$\mathcal{E} = \tfrac{1}{2} \sum_{k=1}^{N} \Big(d^k - \sum_{j=1}^{N_2} w_j \phi(\|\boldsymbol{x}^k - \boldsymbol{c}_j\|)^2 \Big) . \tag{C1.6.13}$$

Gradients of $\mathcal{E}$ with respect to w_j, $\boldsymbol{c}_j$ are easy to compute (see Haykin 1994 for detailed gradient computations) and adaptive algorithms are simply

$$w_i(t+1) = w_i(t) - \mu_w \frac{\partial \mathcal{E}(t)}{\partial w_i(t)} \tag{C1.6.14}$$

$$\boldsymbol{c}_j(t+1) = \boldsymbol{c}_j(t) - \mu_c \frac{\partial \mathcal{E}(t)}{\partial \boldsymbol{c}_j(t)} . \tag{C1.6.15}$$

Adaptation of the radial basis function width. In the most general case, it is interesting to have non-radial basis functions (Poggio and Girosi 1990). This is equivalent to having a weighted norm and is simply obtained by replacing $\|\boldsymbol{x}^k - \boldsymbol{c}_j\|^2$ by $(\boldsymbol{x}^k - \boldsymbol{c}_j)^{\mathrm{T}} \Sigma^{-1} (\boldsymbol{x}^k - \boldsymbol{c}_j)$, where Σ^{-1} is a positive definite matrix. The weighting matrix can also be adapted by a gradient procedure on the cost $\mathcal{E}$ (see Haykin 1994 for details) and the learning rule is:

$$\Sigma^{-1}(t+1) = \Sigma^{-1}(t) - \mu_s \frac{\partial \mathcal{E}(t)}{\partial \Sigma^{-1}(t)} . \tag{C1.6.16}$$

With radial functions, the covariance matrix reduces to $\Sigma = \sigma^2 \mathbf{I}$, and (C1.6.16) adapts only the parameter σ^2.

For RBF classifiers, Musavi *et al* (1992) proposed another approach to adjust the matrix Σ^{-1} of an RBF centered on a point $\boldsymbol{c}_i$. We briefly explain the procedure for a 2-class problem. We first assume that the cluster i centered on $\boldsymbol{c}_i$ corresponds to class i. The idea is to define the largest cluster possible using a Gram–Schmidt procedure. First, one looks for the nearest input of $\boldsymbol{c}_i$, for instance $\boldsymbol{x}_1^i$ belonging to the opposite class. The vector $\boldsymbol{e}_1 = \boldsymbol{x}_1^i - \boldsymbol{c}_i$ determines the least principal axis. Then, one looks for the nearest input, for instance $\boldsymbol{x}_2^i$, with respect to $\boldsymbol{e}_1$, whose projection on $\boldsymbol{e}_1$ is less than $\|\boldsymbol{e}_1\|$. The second principal axis is then $\boldsymbol{e}_2 = (\boldsymbol{x}_2^i - \boldsymbol{c}_i) - [\boldsymbol{e}_1^{\mathrm{T}}(\boldsymbol{x}_2^i - \boldsymbol{c}_i)\boldsymbol{e}_1]/\|\boldsymbol{e}_1\|^2$, and so on. Finally, eigenvalues of Σ are defined from $\|\boldsymbol{e}_i\|$, with a correction factor taking into account the empty space phenomenon for high dimensions.

Orthogonal least-square learning. Chen *et al* (1991) proposed an orthogonal least-square (OLS)—supervised—algorithm to select, one by one, the best centers $\boldsymbol{c}_i$ within database samples $\boldsymbol{x}^j$. Assume the best approximation with q RBF units involves the input samples $\boldsymbol{x}^i$, $1 \le i \le q$, as centers:

$$y(\boldsymbol{x}) = \sum_{j=1}^{q} w_j \phi(\|\boldsymbol{x} - \boldsymbol{x}^j\|) . \tag{C1.6.17}$$

To improve the approximation, we choose, within the remaining $N - q$ samples of the database, the vector $\boldsymbol{x}^k$ which constitutes the best $(q+1)$th regressor, that is, minimizing the square error on the whole database. Note that the criterion must be computed for every remaining point! The algorithm is still a variation of the Gram–Schmidt orthonormalization procedure. Its main drawback is computational cost, the main attractions are incrementality and the small size of the network with respect to a random selection.

6.9 Summary

In this chapter, we introduced artificial neural networks and discussed the basic ideas behind machine learning. We presented the concept of a perceptron as a simple computing element and considered the perceptron learning rule. We explored multilayer neural networks and discussed how to improve the computational efficiency of the back-propagation learning algorithm.

The most important lessons learned in this chapter are:

- Machine learning involves adaptive mechanisms that enable computers to learn from experience, learn by example and learn by analogy. Learning capabilities can improve the performance of an intelligent system over time. One of the most popular approaches to machine learning is artificial neural networks.
- An artificial neural network consists of a number of very simple and highly interconnected processors, called neurons, which are analogous to the biological neurons in the brain. The neurons are connected by weighted links that pass signals from one neuron to another. Each link has a numerical weight associated with it. Weights are the basic means of long-term memory in ANNs. They express the strength, or importance, of each neuron input. A neural network 'learns' through repeated adjustments of these weights.

- In the 1940s, Warren McCulloch and Walter Pitts proposed a simple neuron model that is still the basis for most artificial neural networks. The neuron computes the weighted sum of the input signals and compares the result with a threshold value. If the net input is less than the threshold, the neuron output is −1. But if the net input is greater than or equal to the threshold, the neuron becomes activated and its output attains a value +1.
- Frank Rosenblatt suggested the simplest form of a neural network, which he called a perceptron. The operation of the perceptron is based on the McCulloch and Pitts neuron model. It consists of a single neuron with adjustable synaptic weights and a hard limiter. The perceptron learns its task by making small adjustments in the weights to reduce the difference between the actual and desired outputs. The initial weights are randomly assigned and then updated to obtain the output consistent with the training examples.
- A perceptron can learn only linearly separable functions and cannot make global generalisations on the basis of examples learned locally. The limitations of Rosenblatt's perceptron can be overcome by advanced forms of neural networks, such as multilayer perceptrons trained with the back-propagation algorithm.
- A multilayer perceptron is a feedforward neural network with an input layer of source neurons, at least one middle or hidden layer of computational neurons, and an output layer of computational neurons. The input layer accepts input signals from the outside world and redistributes these signals to all neurons in the hidden layer. The hidden layer detects the feature. The weights of the neurons in the hidden layer represent the features in the input patterns. The output layer establishes the output pattern of the entire network.
- Learning in a multilayer network proceeds in the same way as in a perceptron. The learning algorithm has two phases. First, a training input pattern is presented to the network input layer. The network propagates the input pattern from layer to layer until the output pattern is generated by the output layer. If it is different from the desired output, an error is calculated and then propagated backwards through the network from the output layer to the input layer. The weights are modified as the error is propagated.
- Although widely used, back-propagation learning is not without problems. Because the calculations are extensive and, as a result, training is slow, a pure back-propagation algorithm is rarely used in practical applications. There are several possible ways to improve computational efficiency. A multilayer network learns much faster when the sigmoidal activation function is represented by a hyperbolic tangent. The use of momentum and adaptive learning rate also significantly improves the performance of a multilayer back-propagation neural network.