

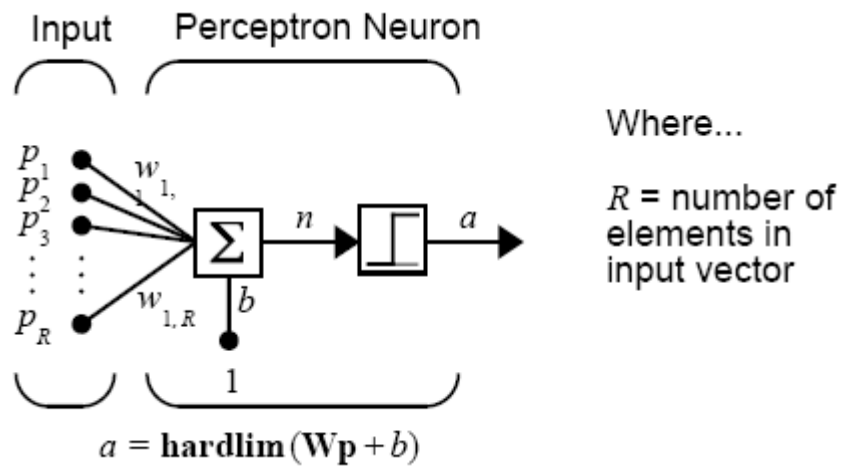
PERCEPTRON

Rețele de tip perceptron

Rețelele de tip perceptron sunt utile în special în rezolvarea unor probleme simple de clasificare a formelor. Aceste rețele pot fi create în MATLAB cu funcția **newp**, inițializate, simulate și antrenate cu funcțiile **init**, **sim**, respectiv **train** (sau **adapt**). În continuare, voi descrie modul în care lucrează aceste rețele și voi introduce aceste funcții.

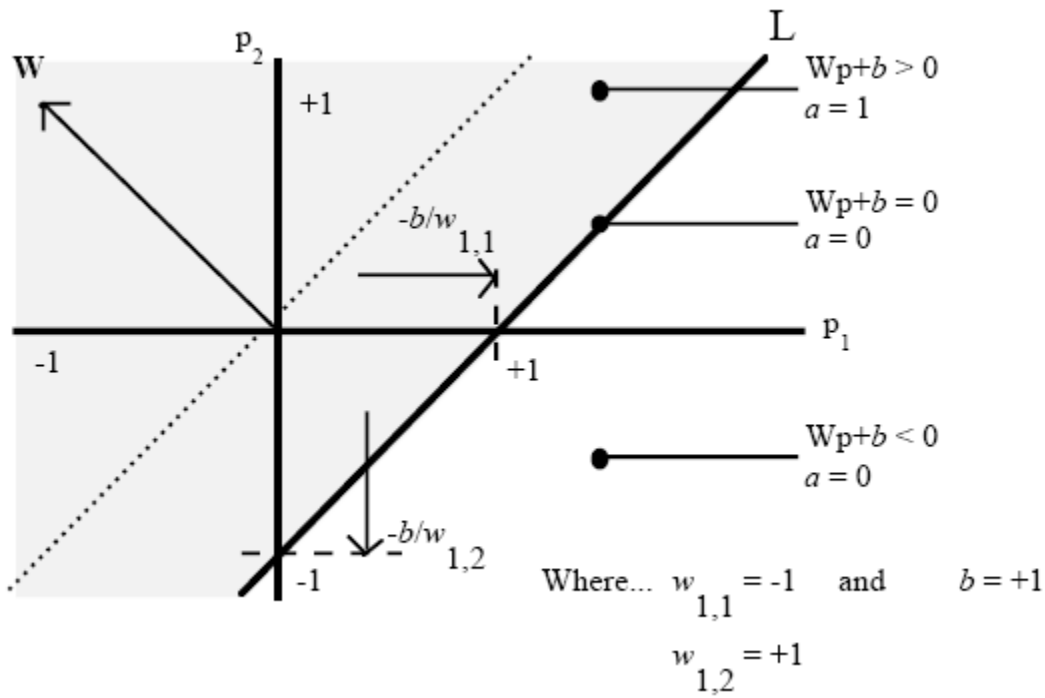
Modelul neuronului

Un perceptron care folosește funcția de transfer hardlim este reprezentat mai jos:



Funcția de transfer a perceptronului, hard-limit, determină clasificarea vectorilor de intrare în două regiuni distincte. Ieșirile rețelei vor fi 0 dacă starea neuronului, n , este strict mai mică decât 0 sau 1 dacă este pozitivă.

Spațiul de intrare al unui neuron cu două intrări, având ponderile $w_{1,1} = -1$, $w_{1,2} = 1$, bias-ul $b = 1$ și funcția de transfer hard-limit este reprezentat în imaginea de mai jos:

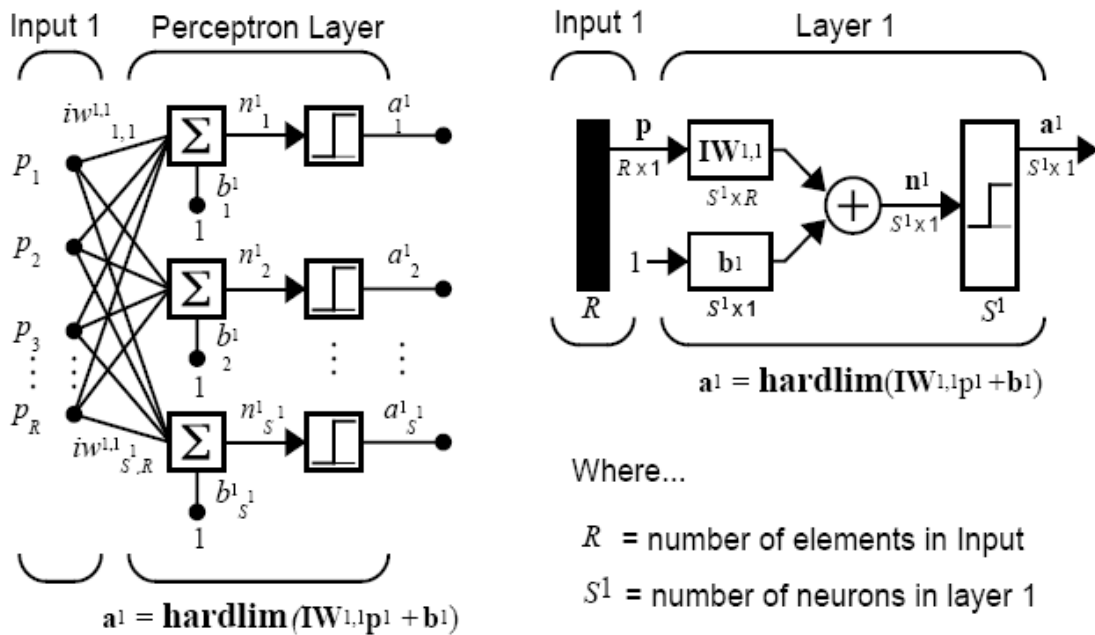


Dreapta de separare L dată de ecuația $\mathbf{W}^* \mathbf{p} + b = 0$ împarte spațiul intrărilor în două regiuni (semiplane). Această dreaptă este perpendiculară pe vectorul ponderilor ($w_{1,1}, w_{1,2}$) și are o deplasare dată de bias-ul b . Punctele aflate în semiplanul din stânga determinat de linia de separare corespund stării neuronului, $\mathbf{net} = \mathbf{W}^* \mathbf{p} + b$, mai mari decât 0 și deci li se asociază ieșirea 1 sau clasa 1, iar punctele, din celălalt semiplan determină $\mathbf{W}^* \mathbf{p} + b < 0$ și prin urmare corespund ieșirii 0 sau clasei 0. Direcția dreptei de separare și deplasarea b pot fi modificate astfel încât o mulțime de date să fie bine clasificată.

Neuronii având deplasarea $b = 0$ și funcția de transfer hard-limit au ca linie de separare o dreaptă ce trece prin origine. Adăugând un bias b , neuronul va permite rezolvarea unor probleme de clasificare în care datele de intrare nu sunt situate în cadrane diferite . Bias-ul permite ca linia de separare să poată fi translatată față de origine, cum se poate observa din figura de mai sus. Aceste observații pot fi vizualizate prin rularea programului demonstrativ **nnd4db**. Observați cum se modifică ponderile, respectiv bias-ul, precum și culoarea marginilor punctelor în momentul în care modificați poziția dreptei de separare.

Arhitectura perceptronului

O rețea de perceptroni constă dintr-un singur nivel de S perceptroni, conectați cu R input-uri cărora le corespund ponderi de forma $w_{i,j}$ (pondere ce conectează intrarea p_j și neuronul i) având reprezentarea de mai jos:



Where...

R = number of elements in Input

S^1 = number of neurons in layer 1

Regula de învățare a perceptronului, pe care o vom descrie este capabilă să antreneze doar o rețea având un singur nivel.

Crearea unui perceptron (newp)

Pentru crearea unei rețele de perceptroni se folosește funcția **newp**:

`net = newp(PR, S)` sau `net = newp(PR, S, TF, LF)`,

unde:

PR este o matrice de dimensiune $R \times 2$ în care sunt precizate minimul și maximul valorilor datelor de intrare

S reprezintă numărul perceptronilor din rețea

TF (Transfer Function) reprezintă funcția de transfer din rețea; poate lua două valori *hardlim* (funcția implicită) sau *hardlims*

LF (Learning Function) reprezintă funcția (tipul) de învățare; poate lua două valori: *learnp* (funcția implicită) sau *learnpn*.

Exemplu: Codul următor creează o rețea de tip perceptron cu un singur neuron și o singură intrare ce ia valori în intervalul [0, 2].

```
net = newp([0, 2], 1);
```

Funcția *newp* generează o structură ce conține toate informațiile legate de rețea, cum ar fi: matricea *ponderilor* și vectorul *bias-urilor*.

Pentru a afla acești parametri ai rețelei vom da comenzile:

```
weights = net.IW{1,1}
```

```
bias = net.b{1}
```

Pentru a vizualiza parametrii rezultați în urma creării rețelei se execută următorul cod:

```
inputweights = net.inputweights{1,1} care va returna
```

```
inputweights =  
delays: 0  
initFcn: 'initzero'  
learn: 1  
learnFcn: 'learnp'  
learnParam: []  
size: [1 1]  
userdata: [1x1 struct]  
weightFcn: 'dotprod'
```

Trebuie să observăm că funcția implicită de învățare este *learnp* pe care o vom discuta puțin mai târziu, iar intrarea rețelei către funcția de transfer *hardlim* este *dotprod*, funcție care generează produsul scalar dintre matricea ponderilor și vectorul de intrare la

care adaugă bias-ul pentru a calcula intrarea rețelei. De asemenea, funcția de inițializare implicită este *initzero*, funcție folosită pentru a inițializa valorile ponderilor.

În mod similar, avem:

biases = net.biases{1} care returnează

```
biases =  
    initFcn: 'initzero'  
    learn: 1  
    learnFcn: 'learnp'  
    learnParam: []  
    size: 1  
    userdata: [1x1 struct]
```

Observăm de asemenea că *bias*-ul este inițializat cu 0.

Exercițiu: Modificați valorile ponderilor și ale bias-ului (atribuindu-le valori aleatoare în intervalul [0,1]) și afișați valorile acestora pentru rețeaua creată în modul următor:

```
net = newp([-1 1;-1 1],1);
```

Verificați ce fac comenzile:

```
net = newp([-1 1;-1 1],1);  
net.inputweights{1,1}.initFcn = 'rands';  
net.biases{1}.initFcn = 'rands';
```

Simularea unei rețele (sim)

Pentru a arăta modul în care lucrează funcția *sim*, vom considera următorul exemplu.

Considerăm un perceptron cu un singur vector de intrare bidimensional, având setate ponderile și bias-ul ca perceptronul prezentat în figura liniei de separare.

Definim rețeaua folosind comanda:

```
net = newp([-2 2;-2 2],1);
```

Cum am spus și mai sus, această comandă setează ponderile și bias-ul 0; dacă vrem să le modificăm trebuie să le atribuim noi valori. Considerăm ponderile și bias-ul setate ca la perceptronul prezentat în figura liniei de separare:

$$\text{net.IW}\{1,1\} = [-1 \ 1];$$

$$\text{net.b}\{1\} = [1];$$

Pentru a verifica corectitudinea setărilor folosim comenzile:

$$\text{net.IW}\{1,1\}$$

ans =

$$\begin{matrix} -1 & 1 \end{matrix}$$

$$\text{net.b}\{1\}$$

ans =

$$1$$

Simulăm comportamentul rețelei pentru două date de intrare $p1 = [1;1]$ și $p2 = [1;-1]$ (puncte aflate în semiplane diferite); ne propunem să aflăm din ce clase fac parte cele două puncte.

$$p1 = [1;1];$$

$$a1 = \text{sim}(\text{net},p1)$$

a1 =

$$1$$

și

$$p2 = [1;-1];$$

$$a2 = \text{sim}(\text{net},p2)$$

a2 =

$$0$$

Se observă că rețeaua clasifică corect cele două date de intrare (ieșirile sunt 0 și 1 corespunzătoare funcției hardlim).

Exercițiu: Reprezentați grafic linia de separare determinată de parametrii din figură și adăugați 4 puncte figurii (pe care să le rețineți într-o matrice pe coloane) atribuindu-le

simboluri potrivite în funcție de clasa din care fac parte și returnați ieșirile rețelei corespunzătoare punctelor care indică această clasă.

Putem introduce datele de intrare și printr-o matrice de celule, iar rezultatul va fi returnat tot într-o matrice de celule:

```
p3 = {[1;1] [1;-1]};  
a3 = sim(net,p3)  
a3 =  
      [1] [0]
```

Inițializarea unui perceptron (*init*)

Folosim funcția *init* pentru a reseta ponderile și bias-urile asociate rețelei la valorile inițiale (valorile cu care s-a inițializat rețeaua). Presupunem că am definit rețeaua:

```
net = newp([-2 2;-2 2],1);
```

Ponderile rețelei le putem vizualiza folosind comenzile:

```
wts = net.IW{1,1}
```

care returnează valorile așteptate

```
wts =  
      0      0
```

În același mod, se poate observa că bias-ul este inițial 0, utilizând comanda:

```
bias = net.b{1}  
bias =  
      0
```

Setăm ponderilor valorile 3 și respectiv 4 și bias-ului valoarea 5.

```
net.IW{1,1} = [3,4];  
net.b{1} = 5;
```

Pentru a reinițializa ponderile (atribuim ponderilor valorile inițiale) folosim funcția *init* cu sintaxa:

```
net = init(net);
```

Putem modifica modul în care un perceptron este inițializat folosind funcția *init*. De exemplu, putem redefini ponderile și bias-ul rețelei astfel:

```
net.inputweights{1,1}.initFcn = 'rands';  
net.biases{1}.initFcn = 'rands';  
net = init(net);
```

Regula de învățare a perceptronului

Definim o regulă de învățare ca fiind o metodă de modificare a ponderilor și a bias-ului rețelei cu scopul de a antrena rețeaua în vederea realizării unei sarcini.

Considerăm ca fiind dată o mulțime de antrenare $\{(p_1, t_1), (p_2, t_2), \dots, (p_m, t_m)\}$, unde p_i reprezintă intrarea rețelei, iar t_i valoarea dorită corespunzătoare intrării p_i . Obiectivul învățării constă în minimizarea erorii date prin $e = t - a$, unde t este valoarea target corespunzătoare unei intrări p , iar a reprezintă ieșirea rețelei.

Regula implicită de învățare a perceptronului - **learnp** – este o regulă de învățare supervizată ce calculează noile ponderi, respectiv bias-ul pentru un vector de intrare p dat și pentru eroarea calculată e . Regula de învățare a perceptronului constă de fapt în găsirea unei drepte de separare care să clasifice corect mulțimea vectorilor de intrare.

La fiecare nouă aplicare a regulii **learnp**, probabilitatea ca perceptronul să returneze o ieșire corectă crește. S-a demonstrat că regula perceptronului converge la o soluție într-un număr finit de iterații, dacă există o soluție (adică dacă mulțimea este liniar separabilă).

Regula de învățare a perceptronului (de actualizare a ponderilor și a bias-ului) este dată prin relațiile:

$$W^{\text{new}} = W^{\text{old}} + ep^T$$

$$b^{\text{new}} = b^{\text{old}} + e$$

Regula de învățare **learnpn** a perceptronului calculează ponderile actualizate în același mod ca funcția **learnp**, dar pentru vectorii de intrare normați, adică vectorii de intrare vor fi de forma:

$$p \mapsto \frac{p}{\|p\|}$$

Folosirea acestei reguli este indicată în cazul în care mulțimea de antrenare conține date având valori discordante în raport cu celelalte exemple, în sensul că norma acestora are o valoare extremă mare/mică față de celelalte date. Această discordanță între date poate determina un proces de antrenare de o complexitate mult mai mare, întrucât în momentul introducerii vectorului având o valoare discordantă ponderile vor fi actualizate prin adăugarea acestui vector în cazul în care este misclasificat, ceea ce va implica un număr mai mare de parcurgeri ale mulțimii de antrenare astfel încât să se manifeste și influența celorlalte date în modificarea ponderilor.

Exemplu:

Fie o rețea cu un singur perceptron, având ca intrare un vector cu două elemente. Considerăm rețeaua creată astfel:

$$net = newp([-2 \ 2; -2 \ 2], 1);$$

Considerăm parametrii rețelei cu următoarele setări:

$$net.b\{1\} = [0];$$

$$w = [1 \ -0.8];$$

$$net.IW\{1,1\} = w;$$

Perechea input-target (p, t) este dată de:

$$p = [1; 2];$$

$$t = [1];$$

Putem calcula ieșirea rețelei și eroarea folosind comenzile:

$$a = sim(net,p)$$

$$a =$$

$$0$$

$$e = t-a$$

$$e =$$

$$1$$

și în final aplicăm funcția **learnp** pentru a actualiza ponderile (se returnează diferența dintre ponderea modificată și ponderea inițială, notată prin dw)

$$dw = \text{learnp}(w, p, [], [], [], [], e, [], [], [])$$

$$dw =$$

$$1 \quad 2$$

Noile ponderi sunt obținute astfel:

$$w = w + dw$$

$$w =$$

$$2.0000 \quad 1.2000$$

Procesul de determinare a ponderilor (a bias-urilor) poate fi repetat până când eroarea devine 0, deoarece se știe că algoritmul de determinare a ponderilor este convergent într-un număr finit de pași pentru toate problemele care pot fi rezolvate folosind un perceptron, adică pentru problemele în care mulțimea datelor de intrare este liniar separabilă. Datele pe care vrem să le clasificăm vor fi în acest caz separate de o dreaptă.

Exercițiu: Studiați programul demonstrativ **nnd4pr**, care vă permite să adăugați noi date de intrare și să observați cum se modifică ponderile în momentul în care aplicăm regula de învățare cu scopul de a clasifica corect datele.

Antrenarea perceptronului (**train**)

În cazul în care problema este liniar separabilă, aplicând de un număr finit de ori în mod repetat funcțiile **learnp** și **sim** perceptronul poate determina dreapta de separare. Fiecare parcurgere a mulțimii de antrenare se numește *epocă* sau *ciclu*. Funcția **train** efectuează un astfel de ciclu. La fiecare pas, pentru o secvență de input-uri, funcția **train** calculează ieșirea și eroarea și modifică ponderile și bias-ul dacă e nevoie pentru fiecare dintre datele din secvența de intrare.

Trebuie reținut faptul că aplicând funcția **train** pentru o singură epocă nu se garantează faptul că rețeaua rezultată îndeplinește funcția țintă. Valorile **W** și **b** pot fi

verificate prin calcularea ieșirilor rețelei pentru fiecare vector de intrare pentru a vedea dacă valorile target sunt atinse.

Exemplu: Fie mulțimea de antrenare având următoarele date:

$$\left\{ p_1 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}, t_1 = 0 \right\}, \left\{ p_2 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, t_2 = 1 \right\}, \left\{ p_3 = \begin{bmatrix} -2 \\ 2 \end{bmatrix}, t_3 = 0 \right\}, \left\{ p_4 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, t_4 = 1 \right\}$$

Dorim să clasificăm corect aceste date folosind o rețea de tip perceptron cu un neuron și având ca date de intrare vectori cu două componente. Inițializăm ponderile și bias-ul cu 0. Astfel, vom avea $W(0) = [0 \ 0]$, $b(0) = 0$ (indicăm pasul algoritmului prin indicii din paranteze asociați ponderilor și bias-ului).

La primul pas al învățării calculăm ieșirea asociată intrării p_1 folosind ponderile inițiale astfel:

$$\begin{aligned} a &= \text{hardlim}(W(0)p_1 + b(0)) \\ &= \text{hardlim}([0 \ 0] \begin{bmatrix} 2 \\ 2 \end{bmatrix} + 0) = \text{hardlim}(0) = 1 \end{aligned}$$

Ieșirea a nu este egală cu ieșirea target 0, deci vom aplica regula de învățare a perceptronului pentru a modifica în manieră incrementală ponderile și bias-ul bazându-ne pe eroarea e . Obținem astfel:

$$\begin{aligned} e &= t_1 - a = 0 - 1 = -1 \\ \Delta W &= ep_1^T = (-1)[2 \ 2] = [-2 \ -2] \\ \Delta b &= e = (-1) = -1 \end{aligned}$$

Putem calcula noile ponderi și bias-ul folosind regula perceptronului în modul următor:

$$\begin{aligned} W^{new} &= W^{old} + ep^T = [0 \ 0] + [-2 \ -2] = [-2 \ -2] = W(1) \\ b^{new} &= b^{old} + e = 0 + (-1) = -1 = b(1) \end{aligned}$$

Introducem acum cel de al doilea vector de intrare p_2 și obținem ieșirea:

$$\begin{aligned}
a &= \text{hardlim}(W(1)p_2 + b(1)) \\
&= \text{hardlim}([-2 \ -2] \begin{bmatrix} 1 \\ -2 \end{bmatrix} - 1) = \text{hardlim}(1) = 1 \\
e &= t_2 - a = 1 - 1 = 0 \\
W(2) &= W(1) = [-2 \ -2] \\
b(2) &= b(1) = -1 \\
e &= t_2 - a = 1 - 1 = 0
\end{aligned}$$

Deoarece atât ieșirea target, cât și ieșirea rețelei sunt 1 rezultă că eroarea este zero, deci ponderile nu se modifică și avem $W(2) = W(1) = [-2 \ -2]$ și $b(2) = b(1) = -1$

După primii 4 pași, adică după prima epocă, obținem $W(4) = [-3 \ -1]$ și $b(4) = 0$. Pentru a determina dacă am obținut o soluție trebuie să parcurgem mulțimea de antrenare și să verificăm dacă ieșirea target a fiecărui punct corespunde ieșirii rețelei determinate de ponderile și bias-ul obținute după o epocă. Întrucât nu toate punctele sunt bine clasificate (punctul p_2 din mulțimea de antrenare este misclasat) algoritmul continuă până când obținem $W(6) = [-2 \ -3]$ și $b(6) = 1$.

Putem descrie algoritmul prezentat mai sus și folosind funcția **train**.

Fie rețeaua de tip perceptron definită ca mai jos. Vrem să antrenăm rețeaua folosind mulțimea de antrenare de mai sus într-o singură epocă și să observăm că se obțin aceleași ponderi ca în algoritmul prezentat mai devreme.

$$\begin{aligned}
net &= \text{newp}([-2 \ 2; -2 \ 2], 1); \\
net.trainParam.epochs &= 1;
\end{aligned}$$

Vectorii de intrare și valorile target corespunzătoare sunt:

$$\begin{aligned}
p &= [[2; 2] [1; -2] [-2; 2] [-1; 1]] \\
t &= [0 \ 1 \ 0 \ 1]
\end{aligned}$$

Acum antrenăm rețeaua utilizând funcția **train**:

$$net = \text{train}(net, p, t);$$

După această primă epocă vom obține $W = [-3 \ -1]$ și $b = 0$.

Pentru a verifica dacă ponderile și bias-ul rezolvă problema de clasificare simulăm rețeaua astfel:

$$a = \text{sim}(net, p)$$

$$a = \begin{bmatrix} [0] & [0] & [1] & [1] \end{bmatrix}$$

$$e = t - a$$

Întrucât vectorul de eroare nu are toate componentele nule, rezultă că nu am obținut parametrii care definesc linia de separare, deci trebuie să continuăm antrenarea rețelei pentru un număr mai mare de epoci. Pentru un număr de epoci egal cu 2 vom obține vectorul de eroare e având toate componentele nule, deci obținem parametrii corecți care clasifică corect datele din mulțimea de antrenare.

Exercițiu: Antrenați rețeaua de mai sus timp de o epocă după care reprezentați grafic punctele, respectiv dreapta de separare. Repetați antrenarea rețelei timp de patru epoci și reprezentați grafic punctele, respectiv dreapta de separare obținută. (Realizați ambele reprezentări grafice în aceeași fereastră). Interpretați rezultatele obținute!

Exercițiu: Rulați programul demonstrativ demop1 (program care clasifică o mulțime de antrenare și ilustrează modul în care rețeaua este antrenată).

Funcția **train** realizează o antrenare **serială** (de tip incremental) în sensul că după introducerea fiecărei date din mulțimea de antrenare se realizează ajustarea parametrilor, dar matricea ponderilor $\text{net.IW}\{1,1\}$ și vectorul bias $\text{net.b}\{1\}$ va conține valorile actualizate după parcurgerea întregii mulțimi de antrenare.

Voi introduce o nouă funcție utilizată în antrenarea perceptronului, funcția **adapt**, care însă realizează o antrenare de tip **batch** în care ponderile $\text{net.IW}\{1,1\}$ și bias-ul $\text{net.b}\{1\}$ se modifică pe baza tuturor input-urilor. Ponderile și bias-ul se modifică prin adăugarea la vechile valori suma punctelor misclasificate. În cazul funcției **adapt** trebuie să setăm numărul de parcurgeri ale mulțimii de antrenare, necesar antrenării rețelei. Funcția **adapt** poate realiza și o antrenare de tip incremental dacă este aplicată unei singure date de intrare. Astfel pentru antrenarea rețelei prezentate mai sus folosind o antrenare de tip **batch** vom utiliza următorul cod:

```
net = newp([-2 2; -2 2], 1);
p = [2 1 -1 -1; 2 -2 2 1] % datele de intrare
t = [0 1 0 1] % valorile target
net.adaptParam.passes = 100;
```

```

net = adapt(net, p, t);
plotpv(p,t); % reprezintă grafic punctele
hold on
plotpc(net.IW{1,1}, net.b{1}); % reprezintă grafic dreapta de separare

```

Exercițiu: Scrieți un program MATLAB care să definească o rețea de tip perceptron utilizată pentru reprezentarea funcțiilor logice AND și OR pe care să o antrenați pentru a recunoaște aceste funcții și reprezentați grafic punctele, respectiv dreapta de separare.

Aplicație:

Se dă următoarea mulțime de antrenare formată din perechi de puncte și respectiv etichetele corespunzătoare:

$$\left\{ \left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, 1 \right), \left(\begin{bmatrix} 0 \\ 0.5 \end{bmatrix}, 1 \right), \left(\begin{bmatrix} 0 \\ 1 \end{bmatrix}, 1 \right), \left(\begin{bmatrix} 0.5 \\ 0 \end{bmatrix}, 1 \right), \left(\begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}, -1 \right), \left(\begin{bmatrix} 0.5 \\ 1 \end{bmatrix}, -1 \right), \left(\begin{bmatrix} 1 \\ 0 \end{bmatrix}, -1 \right), \left(\begin{bmatrix} 1 \\ 0.5 \end{bmatrix}, -1 \right) \right\}$$

Implementați în MATLAB varianta **batch** a algoritmului de antrenare a perceptronului (fără a utiliza funcțiile MATLAB **adapt** sau **train**).

- Repetăți algoritmul de antrenare de un număr maxim de epoci de 500. În momentul determinării ponderilor care dau direcția dreptei de separare, care clasifică corect punctele din mulțimea de antrenare, încheiați procesul de antrenare și plotați punctele din mulțimea de antrenare și dreapta de separare.
- Pe baza antrenării într-un număr maxim de 500 de epoci, afișați dacă mulțimea de antrenare este liniar separabilă (dacă s-a determinat dreapta care clasifică corect punctele) sau neliniar separabilă (dacă după 500 de epoci nu a fost găsită dreapta care clasifică corect punctele) folosind funcția **disp**. Dacă mulțimea de antrenare nu este liniar separabilă determinați dreapta care misclasifică cele mai puține puncte și afișați punctele misclasificate.
- Studiați convergența algoritmului perceptronului, reprezentând grafic iterația versus valoarea cosinusului dintre doi vectori de ponderi obținuți la iterații consecutive.

