



Inteligentă artificială

7. Rețele neuronale

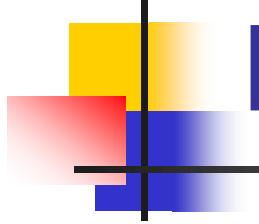
Florin Leon

Universitatea Tehnică „Gheorghe Asachi” din Iași
Facultatea de Automatică și Calculatoare

Universitatea „Al. I. Cuza” din Iași
Facultatea de Informatică

Florin Leon - Inteligenta artificiala

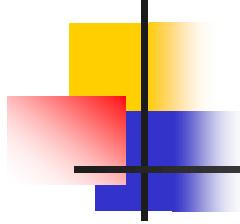
<https://sites.google.com/view/iafii/home> - http://florinleon.byethost24.com/curs_ia_info.html



Rețele neuronale

1. Introducere
2. Perceptronul. Adaline
3. Perceptronul multi-strat
4. Rețele profunde
5. Concluzii





Rețele neuronale

1. Introducere
2. Perceptronul. Adaline
3. Perceptronul multi-strat
4. Rețele profunde
5. Concluzii



Clasificarea

- Se dă o **mulțime de antrenare**: o mulțime de **instante** (vectori de antrenare, obiecte)
- Instantele au **attribute**
- Fiecare instanță are attribute cu anumite **valori**
- De obicei, ultimul atribut este **clasa**

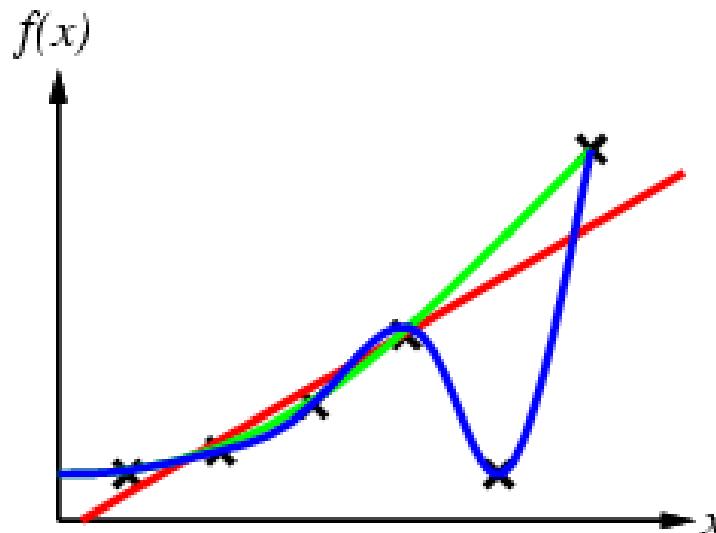
Atribute

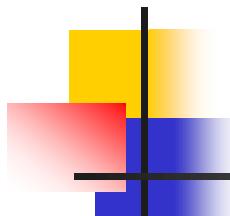
Tid	Refund	Marital Status	Taxable Income	Cheat
1	Yes	Single	125K	No
2	No	Married	100K	No
3	No	Single	70K	No
4	Yes	Married	120K	No
5	No	Divorced	95K	Yes
6	No	Married	60K	No
7	Yes	Divorced	220K	No
8	No	Single	85K	Yes
9	No	Married	75K	No
10	No	Single	90K	Yes

Instante

Regresia

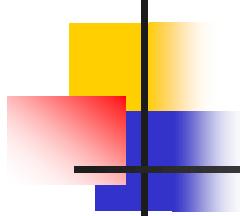
- Reprezintă aproximarea unei funcții
 - Orice fel de funcție, nu doar de tipul $f: \mathbb{R}^n \rightarrow \mathbb{R}$
 - Doar ieșirea este continuă; unele intrări pot fi discrete sau nenumerice





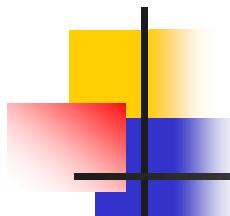
Clasificarea și regresia

- Aceeași idee de bază: învățarea unei relații între intrări (vectorul x) și ieșire (y) din date
- Singura diferență între clasificare și regresie este tipul ieșirii: discret, respectiv continuu
- Clasificarea estimează o ieșire discretă, clasa
- Regresia estimează o funcție h astfel încât $h(x) \approx y(x)$ cu o anumită precizie
- Pentru fiecare instanță de antrenare, valoarea dorită a ieșirii este dată \Rightarrow **învățare supervizată**



Rețelele neuronale

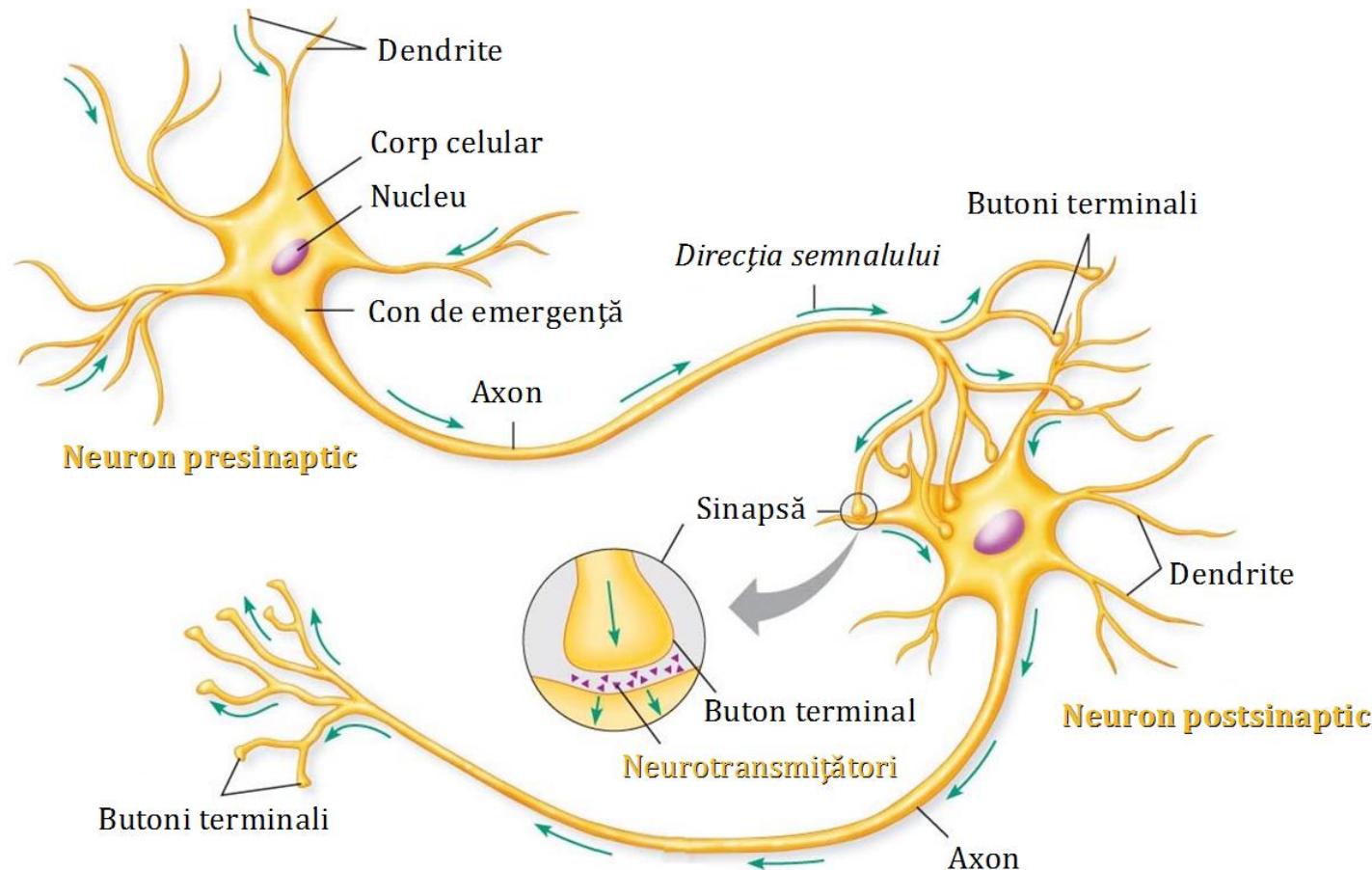
- Rețelele neuronale de tip perceptron, prezentate în acest curs, sunt folosite în general pentru probleme de regresie și clasificare



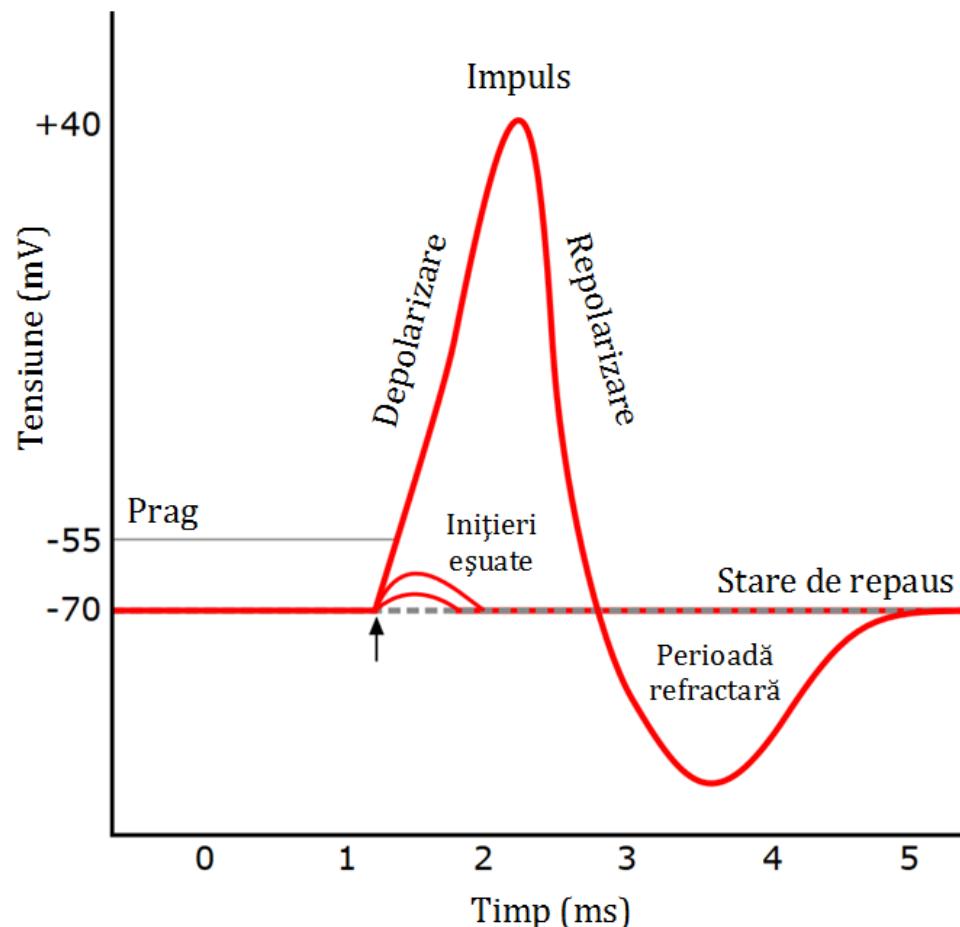
Modelul biologic al neuronilor

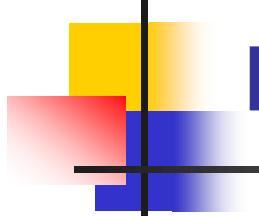
- Modul în care lucrează creierul ființelor vii este complet diferit de cel al calculatoarelor numerice convenționale
- O rețea neuronală artificială este un model simplificat al creierului biologic
- Creierul uman dispune în medie de 86 de miliarde de neuroni și 150 de trilioane de sinapse
- Calculele sunt prelucrări paralele, complexe și neliniare
- Fiecare neuron are o structură simplă (doar aparent...), dar interconectarea lor asigură puterea de calcul

Interconectarea neuronilor



Impuls neuronal tipic

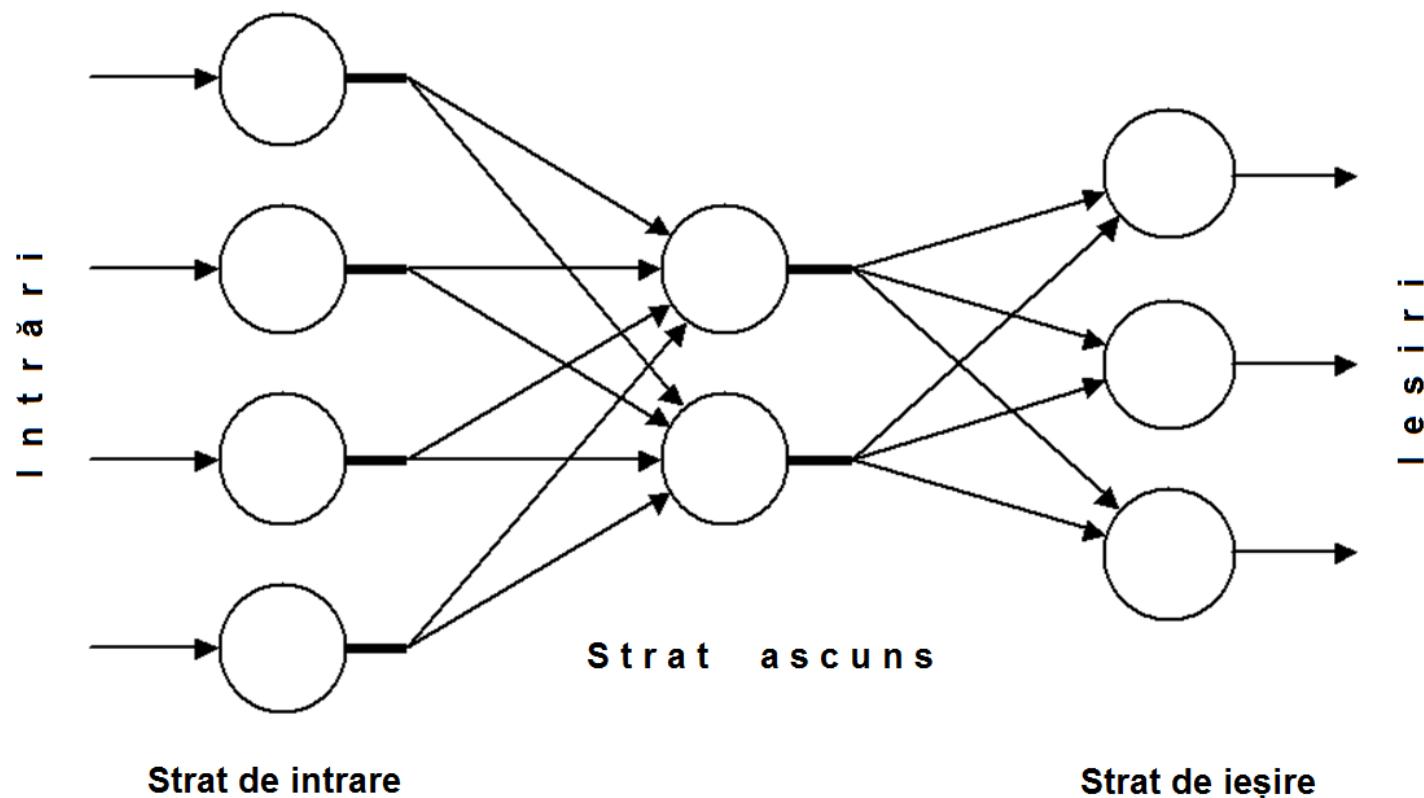


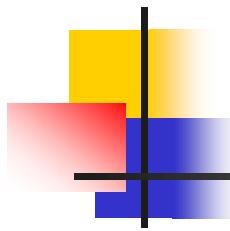


Caracteristicile unei rețele neuronale biologice

- Informațiile sunt stocate și prelucrate în toată rețeaua: **sunt globale, nu locale**
- O caracteristică esențială este plasticitatea: capacitatea de a se adapta, de a **învăța**
- Posibilitatea **învățării** a condus la ideea modelării unor rețele neuronale (mult) simplificate cu ajutorul calculatorului

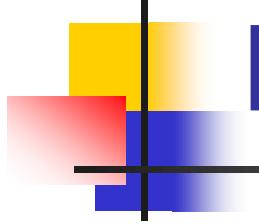
Rețele neuronale artificiale





Analogii

- | | |
|-----------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none">■ RN biologică | <ul style="list-style-type: none">■ RN artificială |
| <ul style="list-style-type: none">■ Soma (corpul celulei)■ Dendrite■ Axon■ Sinapsă | <ul style="list-style-type: none">■ Neuron■ Intrări■ Ieșire■ Pondere (<i>weight</i>) |



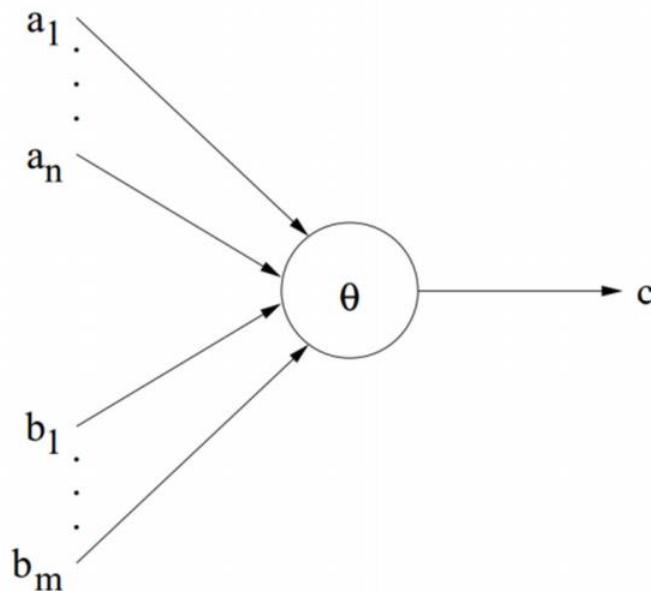
Rețele neuronale

1. Introducere
2. Perceptronul. Adaline
3. Perceptronul multi-strat
4. Rețele profunde
5. Concluzii



Neuronul artificial

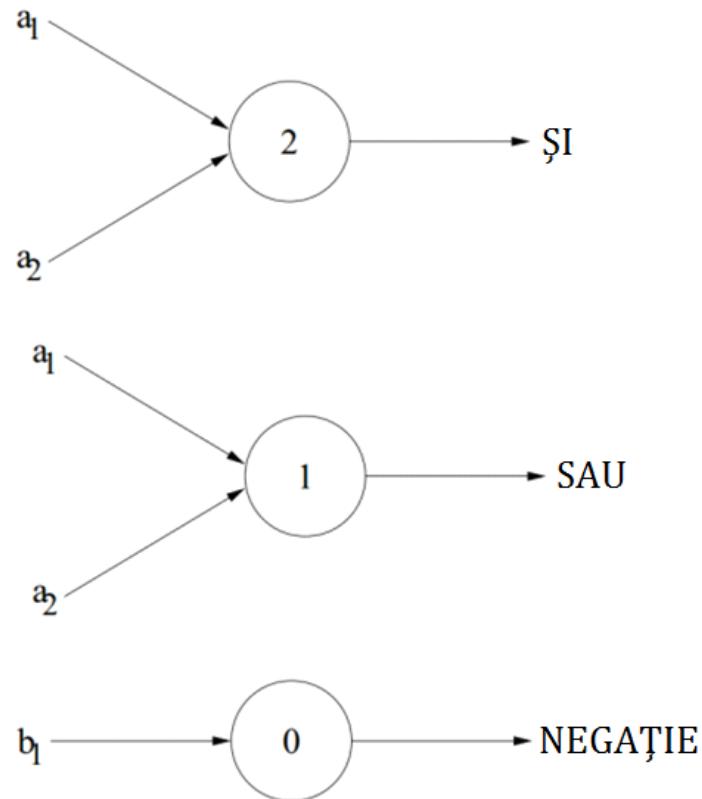
- Primul model matematic al unui neuron a fost propus de McCulloch și Pitts (1943)
- a_i sunt intrări excitatoare, b_j sunt intrări inhibitoare



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

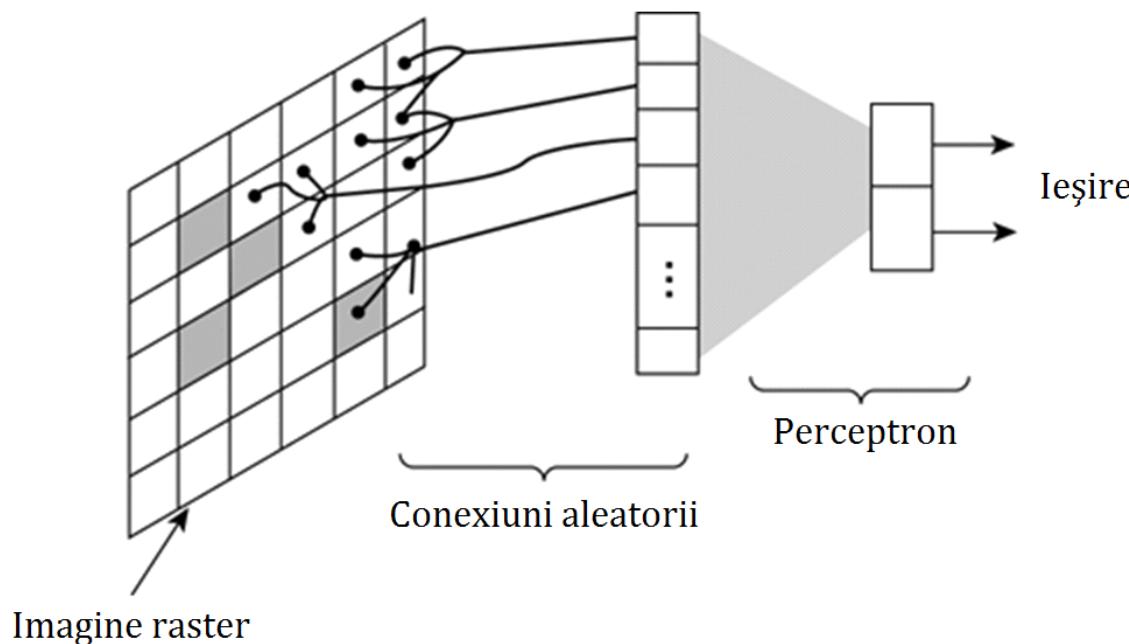
Neuronul McCulloch-Pitts

- Nu poate învăța; parametrii se stabilesc analitic



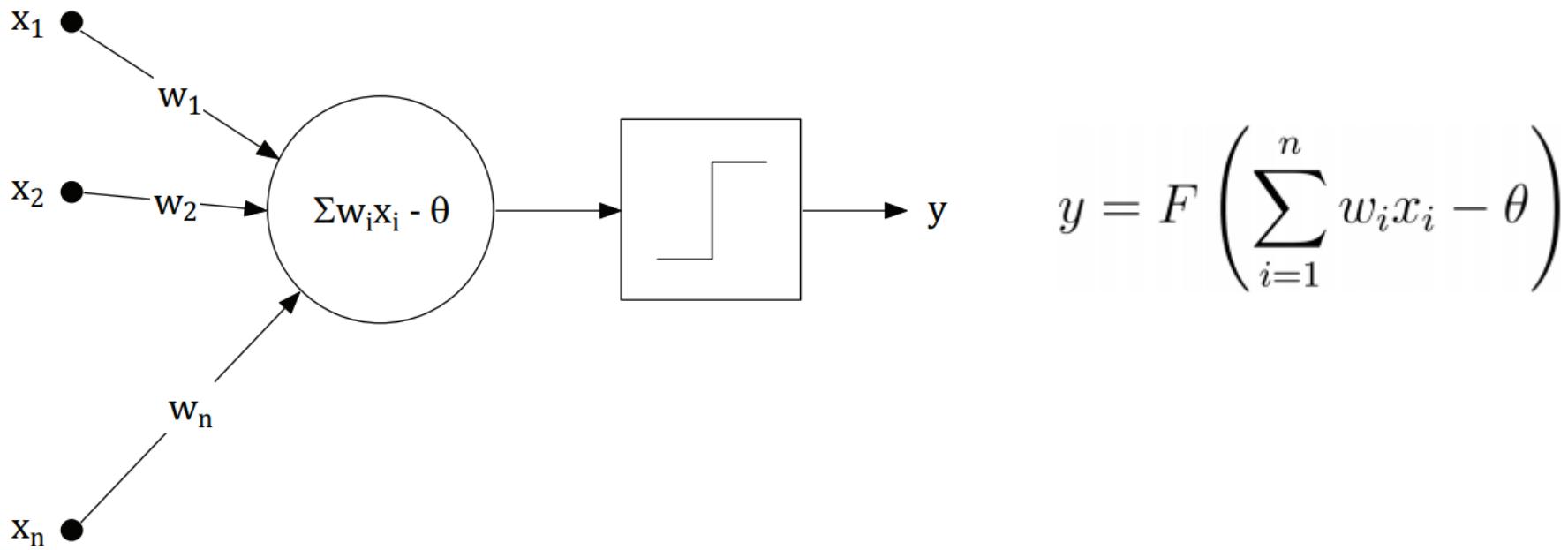
Perceptronul originar

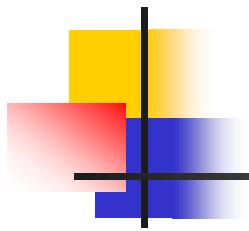
- Rosenblatt (1958), încercând să rezolve problema învățării, a propus un model de neuron numit **perceptron**, prin analogie cu sistemul vizual uman



Perceptronul standard

- Semnalele de intrare sunt sumate, iar neuronul generează un semnal doar dacă suma depășește pragul



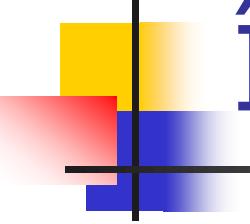


Perceptronul

- Perceptronul are ponderi sinaptice ajustabile și funcție de activare **semn** sau **treaptă**

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

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

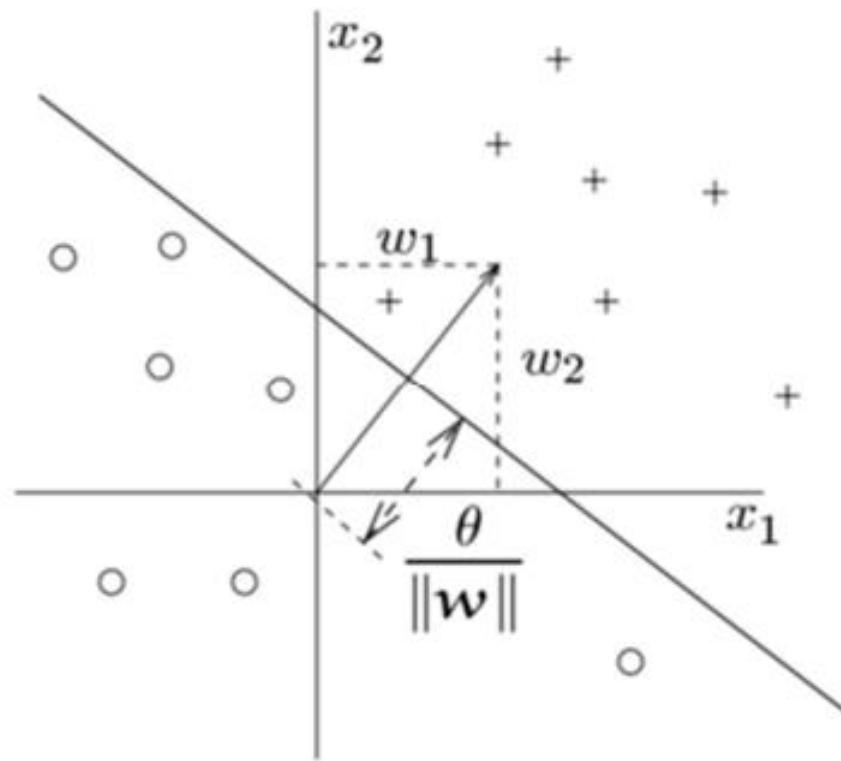


Învățarea

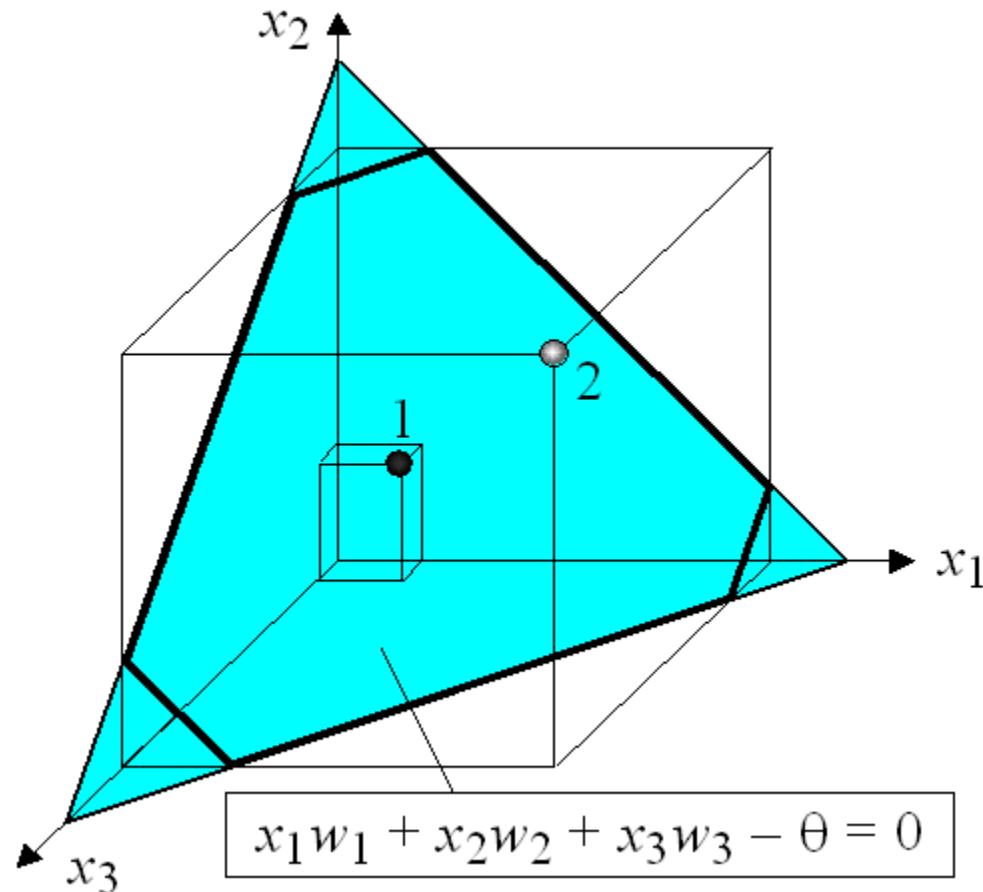
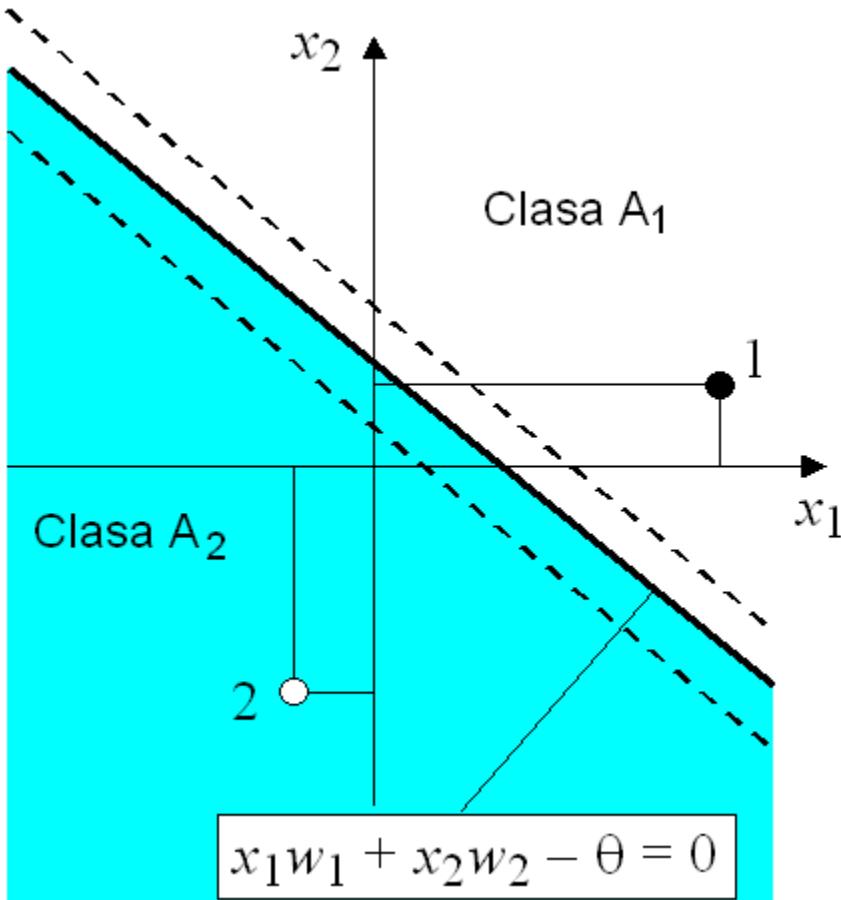
- Scopul perceptronului este să clasifice intrările x_1, x_2, \dots, x_n cu două clase A_1 și A_2
- Spațiul intrărilor, n -dimensional, este împărțit în două de un hiperplan definit de ecuația:

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

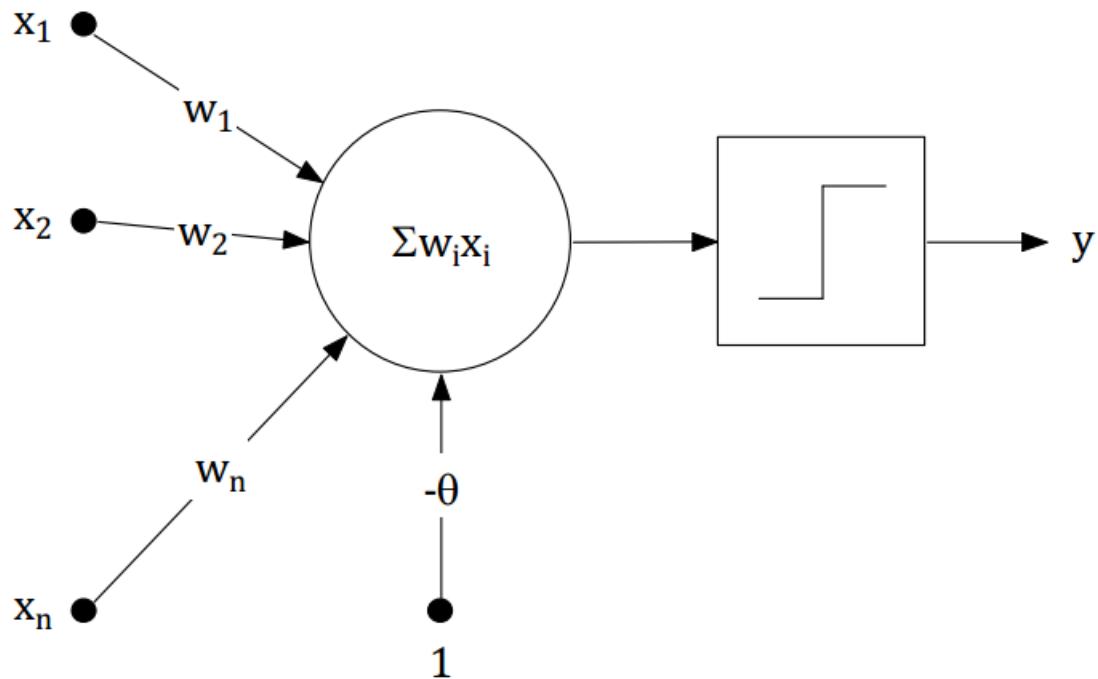
Interpretarea geometrică



Exemplu: 2, respectiv 3 intrări

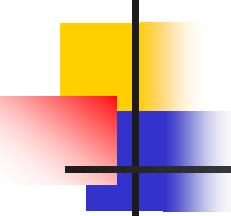


Pragul ca pondere



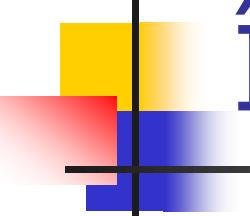
$$y = F \left(\sum_{i=1}^{n+1} w_i x_i \right)$$

net input



Învățarea

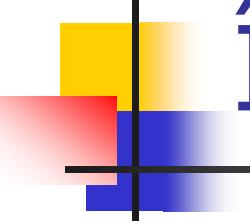
- Învățarea are loc prin ajustarea succesivă a ponderilor pentru a reduce diferența dintre ieșirile reale și ieșirile dorite, pentru **toate** datele de antrenare
- Dacă pentru un vector de antrenare ieșirea reală este y , iar ieșirea dorită este y_d , atunci eroarea este: $e = y_d - y$
- Dacă eroarea este pozitivă, trebuie să creștem ieșirea perceptronului y
- Dacă eroarea este negativă, trebuie să micșorăm ieșirea y



Învățarea

- Dacă $y = 0$ și $y_d = 1 \Rightarrow e = 1$
- $x \cdot w = 0$
- $x \cdot w_d = 1$
- Dacă $x > 0$, w trebuie să crească
- Dacă $x < 0$, w trebuie să scadă

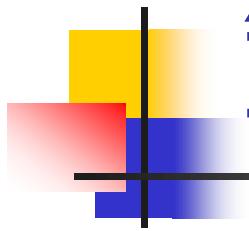
- Atunci: $\Delta w = \alpha \cdot x = \alpha \cdot x \cdot e$
- α este **rata de învățare**



Învățarea

- Dacă $y = 1$ și $y_d = 0 \Rightarrow e = -1$
- $x \cdot w = 1$
- $x \cdot w_d = 0$
- Dacă $x > 0$, w trebuie să scadă
- Dacă $x < 0$, w trebuie să crească

- Atunci: $\Delta w = \alpha \cdot (-x) = \alpha \cdot x \cdot e$



Învățarea

- Regula de învățare a perceptronului:

$$\Delta w = \alpha \cdot x \cdot e$$

- Folosind funcția de activare treaptă

se initializează toate ponderile w_i cu 0 sau cu valori aleatorii din intervalul $[-0.5, 0.5]$
se initializează rata de învățare alfa cu o valoare din intervalul $(0, 1]$, de exemplu 0.1
se initializează numărul maxim de epoci P, de exemplu 100
 $p = 0$ // numărul epocii curente
 $\text{erori} = \text{true}$ // un flag care indică existența erorilor de antrenare

repetă cât timp $p < P$ și $\text{erori} == \text{true}$

{

$\text{erori} = \text{false}$

pentru fiecare vector de antrenare x_i cu $i = 1..N$

{

$y_i = F(\text{sum}(x_{ij} * w_j))$ cu $j = 1..n+1$

dacă ($y_i \neq y_{di}$)

{

$e = y_{di} - y_i$

$\text{erori} = \text{true}$

pentru fiecare intrare $j = 1..n+1$

$w_j = w_j + \alpha * x_{ij} * e$

}

}

$p = p + 1$

}

Algoritmul de
antrenare al
perceptronului

Exemplu de antrenare: funcția ȘI

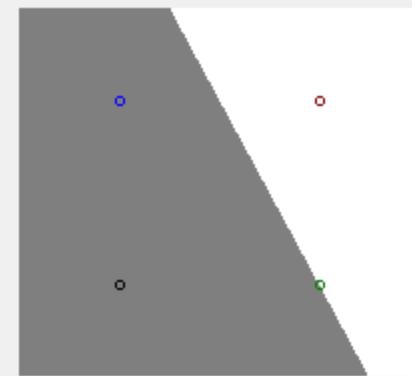
Epochă	Intrări		leșire dorită Y_d	Ponderi inițiale		leșire reală Y	Eroare e	Ponderi finale	
	x_1	x_2		w_1	w_2			w_1	w_2
1	0	0	0	0.3	-0.1	0	0	0.3	-0.1
	0	1	0	0.3	-0.1	0	0	0.3	-0.1
	1	0	0	0.3	-0.1	1	-1	0.2	-0.1
	1	1	1	0.2	-0.1	0	1	0.3	0.0
2	0	0	0	0.3	0.0	0	0	0.3	0.0
	0	1	0	0.3	0.0	0	0	0.3	0.0
	1	0	0	0.3	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

Prag : $\theta = 0.2$; rată de învățare: $\alpha = 0.1$



Multimea de antrenare

```
000  
010  
100  
111
```

Antreneaza**Deseneaza**

Iteratii

```
x1=1  x2=0  yd=0  y=0.00  
x1=1  x2=1  yd=1  y=0.00
```

Epoca 2

```
x1=0  x2=0  yd=0  y=0.00  
x1=0  x2=1  yd=0  y=0.00  
x1=1  x2=0  yd=0  y=0.00  
x1=1  x2=1  yd=1  y=0.00
```

Epoca 3

```
x1=0  x2=0  yd=0  y=0.00  
x1=0  x2=1  yd=0  y=1.00  
x1=1  x2=0  yd=0  y=0.00  
x1=1  x2=1  yd=1  y=1.00
```

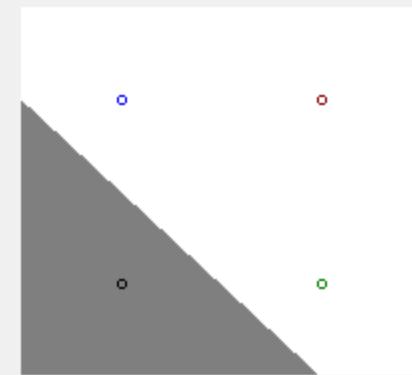
Epoca 4

```
x1=0  x2=0  yd=0  y=0.00  
x1=0  x2=1  yd=0  y=0.00  
x1=1  x2=0  yd=0  y=0.00  
x1=1  x2=1  yd=1  y=1.00
```



Multimea de antrenare

```
000  
011  
101  
111
```

Antreneaza**Deseneaza**

Iteratii

```
x1=1  x2=0  yd=1  y=1.00  
x1=1  x2=1  yd=1  y=1.00
```

Epoca 3

```
x1=0  x2=0  yd=0  y=1.00  
x1=0  x2=1  yd=1  y=1.00  
x1=1  x2=0  yd=1  y=1.00  
x1=1  x2=1  yd=1  y=1.00
```

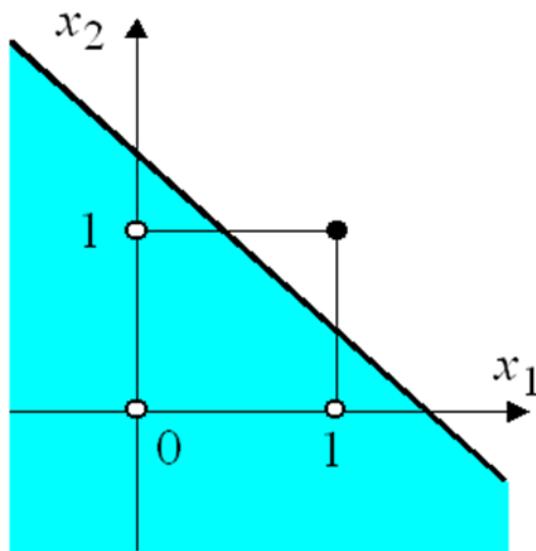
Epoca 4

```
x1=0  x2=0  yd=0  y=1.00  
x1=0  x2=1  yd=1  y=1.00  
x1=1  x2=0  yd=1  y=1.00  
x1=1  x2=1  yd=1  y=1.00
```

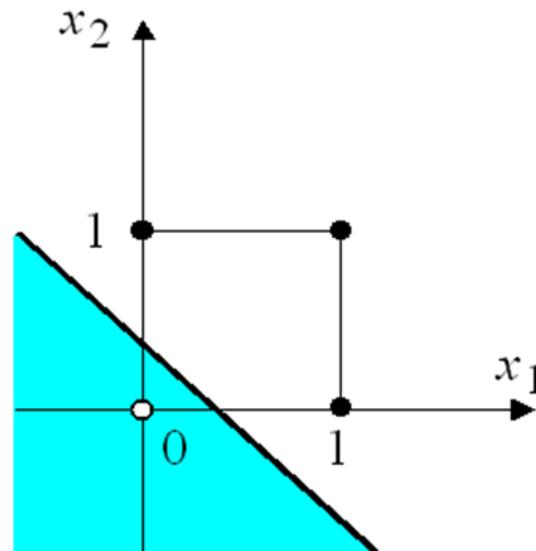
Epoca 5

```
x1=0  x2=0  yd=0  y=0.00  
x1=0  x2=1  yd=1  y=1.00  
x1=1  x2=0  yd=1  y=1.00  
x1=1  x2=1  yd=1  y=1.00
```

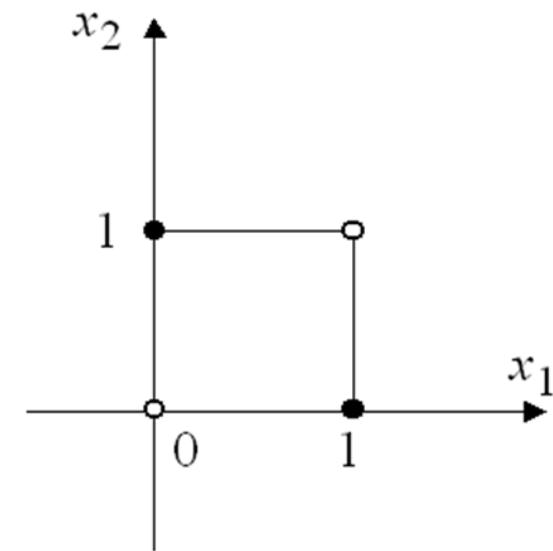
Reprezentarea grafică



(a) **ŞI** ($x_1 \cap x_2$)

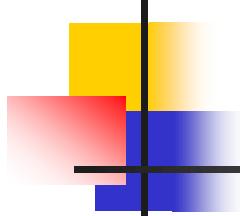


(b) **SAU** ($x_1 \cup x_2$)



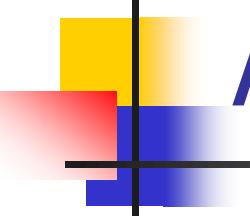
(c) **SAU Exclusiv**
($x_1 \oplus x_2$)

- Perceptronul poate reprezenta doar **funcții separabile liniar**



Discuție

- Perceptronul poate reprezenta funcții separabile liniar complexe, de exemplu, funcția majoritate
 - Un arbore de decizie ar avea nevoie de 2^n noduri
- Poate învăță tot ce poate reprezenta, dar nu poate reprezenta multe funcții
 - Nu poate reprezenta funcții neseparabile liniar, de exemplu, XOR



Adaline

- Funcție de activare liniară

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

- Eroarea unui vector de antrenare

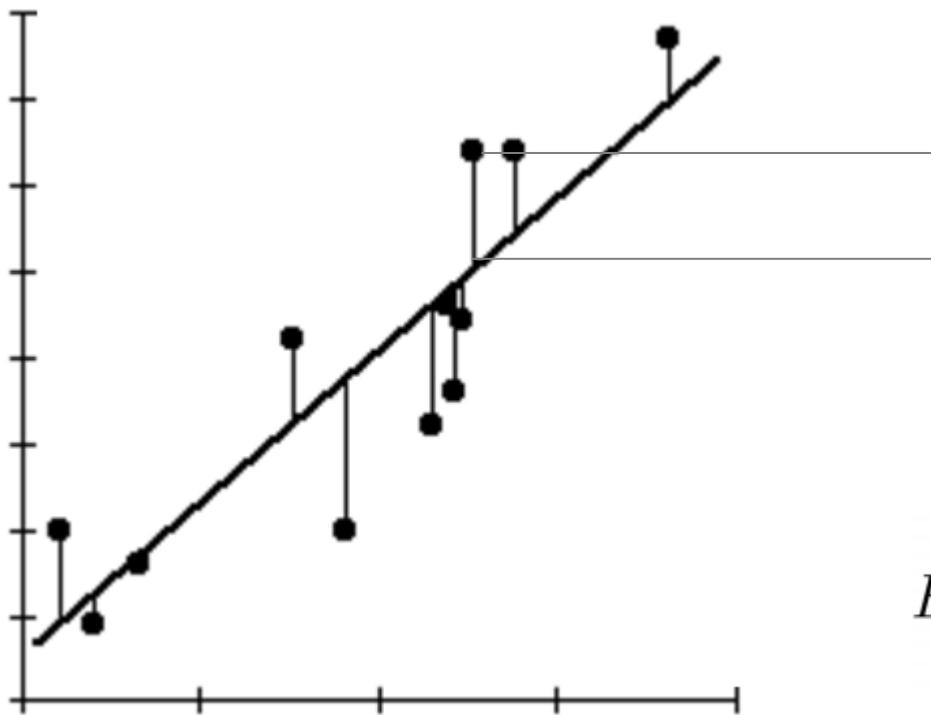
$$E_i = \frac{1}{2} (y_{di} - y_i)^2$$

- Eroarea pe mulțimea de antrenare

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

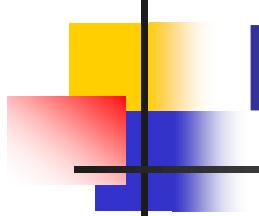
Eroarea medie pătratică

- engl. "Mean Square Error", MSE



$$E_i = \frac{1}{2} (y_{di} - y_i)^2$$

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



Regula de antrenare

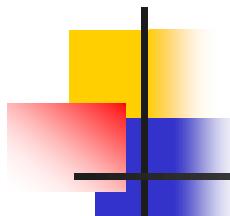
$$\Delta_i w_j = -\alpha \frac{\partial E_i}{\partial w_j}$$

$$\frac{\partial E_i}{\partial w_j} = \frac{\partial E_i}{\partial y_i} \frac{\partial y_i}{\partial w_j}$$

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

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

$$\Delta w = \alpha \cdot x \cdot e \quad \longleftarrow \quad \alpha \text{ este rata de învățare}$$



Regula delta

- În cazul general:

$$\frac{\partial E_i}{\partial w_j} = -x_{ij} (y_{di} - y_i) f'(y_i)$$

funcția de activare
(derivabilă)

- Actualizarea ponderilor:

$$\Delta_i w_j = \alpha \cdot x_{ij} \cdot e_i \cdot f'(y_i)$$

Regula delta pentru funcția de activare liniară

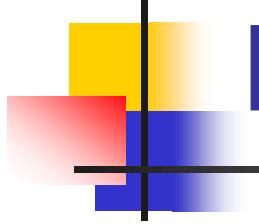
$$\Delta_i w_j = \alpha \cdot x_{ij} \cdot e_i \cdot f'(y_i) \quad f'(y_i) = 1$$

$$w_j(p+1) = w_j(p) + \alpha \cdot x_{ij}(p) \cdot e_i(p)$$



p = pasul după care se fac actualizările
(vectorul sau epoca de antrenare)

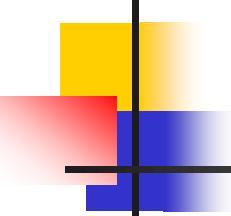
- Este aceeași expresie ca la regula de învățare a perceptronului
- Însă în cazul anterior se folosea funcția de activare prag, care nu este derivabilă
- Regula delta se poate aplica doar pentru funcții de activare derivabile



Rețele neuronale

1. Introducere
2. Perceptronul. Adaline
- 3. Perceptronul multi-strat**
4. Rețele profunde
5. Concluzii

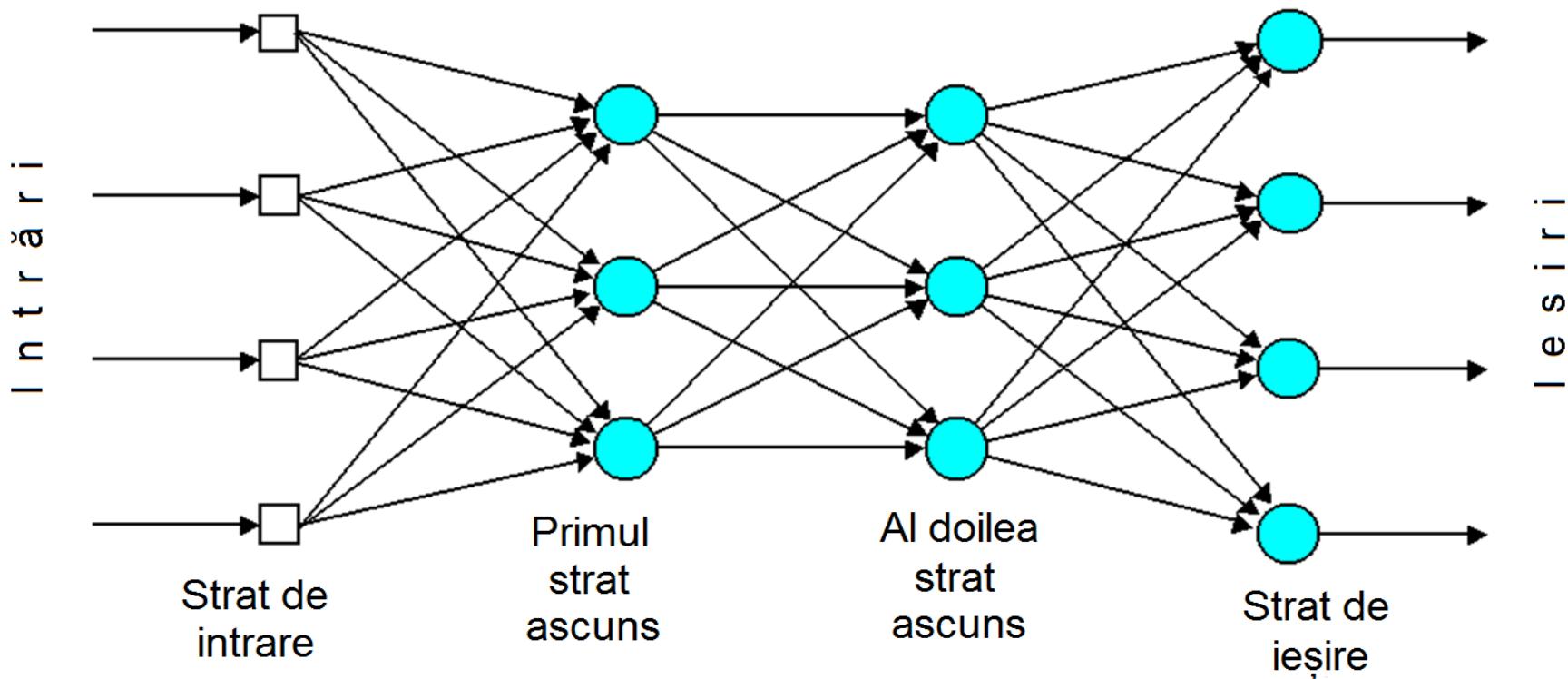


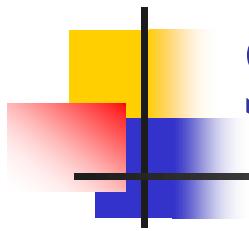


Perceptronul multi-strat

- Perceptronul multi-strat este o rețea neuronală cu propagare înainte (*feed-forward*) cu unul sau mai multe straturi ascunse
 - Un strat de intrare
 - Unul sau mai multe straturi ascunse / intermediare
 - Un strat de ieșire
- Calculele se realizează numai în neuronii din straturile ascunse și din stratul de ieșire
- Semnalele de intrare sunt propagate înainte succesiv prin straturile rețelei

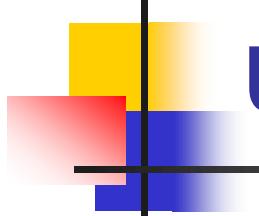
Perceptron multi-strat cu două straturi ascunse





Straturile „ascunse”

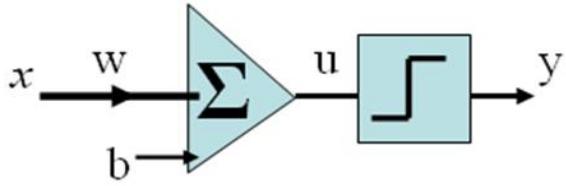
- Un strat ascuns își „ascunde” ieșirea dorită. Cunoscând corespondența intrare-iesire a rețelei („cutie neagră”), nu este evident care trebuie să fie ieșirea dorită a unui neuron dintr-un strat ascuns
- Rețelele neuronale clasice au 1 sau 2 straturi ascunse. Fiecare strat poate conține, de exemplu, 10 – 50 – 100 de neuroni



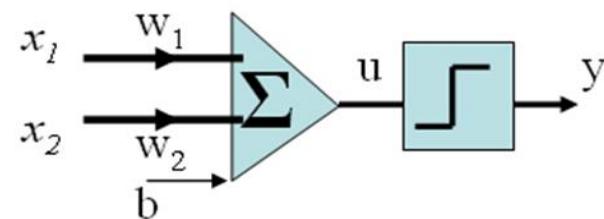
Proprietatea de aproximare universală

- O rețea neuronală cu un singur strat ascuns, cu un număr posibil infinit de neuroni, poate aproxima orice funcție reală continuă
- Un strat suplimentar poate însă reduce foarte mult numărul de neuroni necesari în straturile ascunse

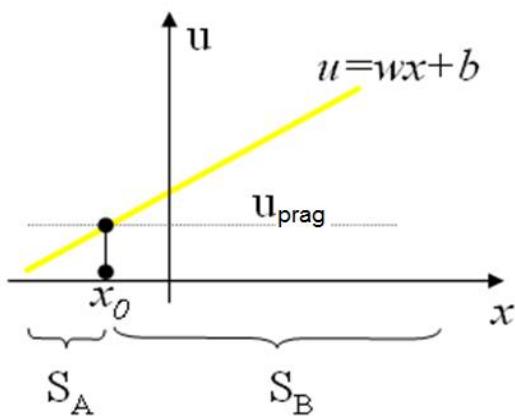
Perceptron cu o singură intrare



Perceptron cu două intrări

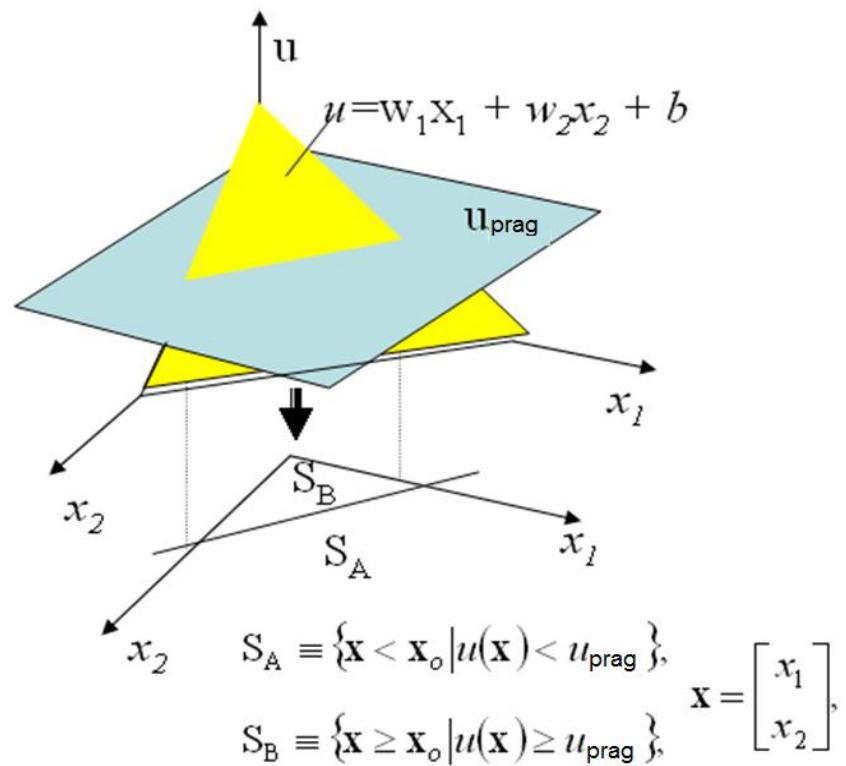


Interpretarea geometrică



$$S_A \equiv \{x < x_o | u(x) < u_{\text{prag}}\}$$

$$S_B \equiv \{x \geq x_o | u(x) \geq u_{\text{prag}}\},$$



$$S_A \equiv \{\mathbf{x} < \mathbf{x}_o | u(\mathbf{x}) < u_{\text{prag}}\},$$

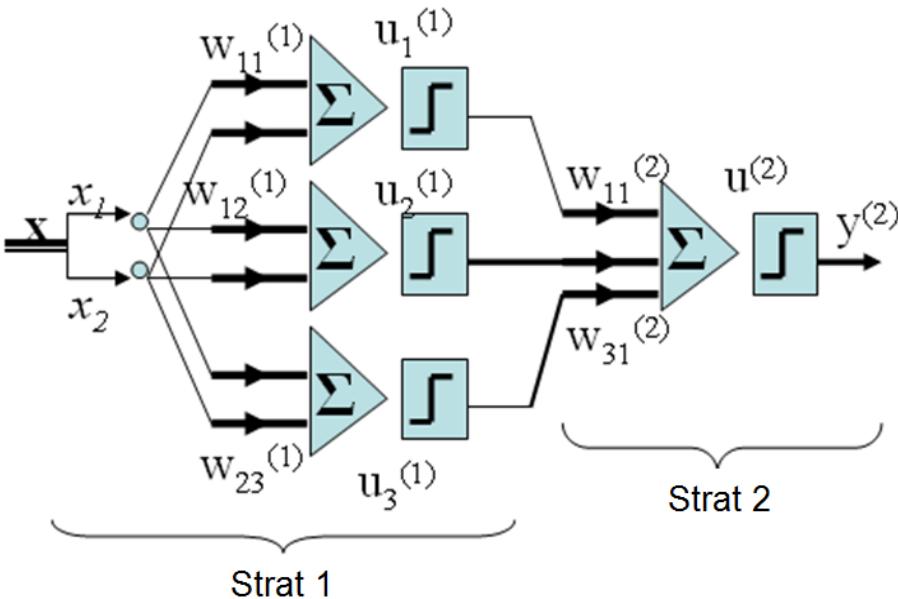
$$S_B \equiv \{\mathbf{x} \geq \mathbf{x}_o | u(\mathbf{x}) \geq u_{\text{prag}}\},$$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix},$$

1 strat ascuns

Folosind un perceptron cu 2 intrări și 2 straturi

se poate delimita o suprafață convexă



Strat 1

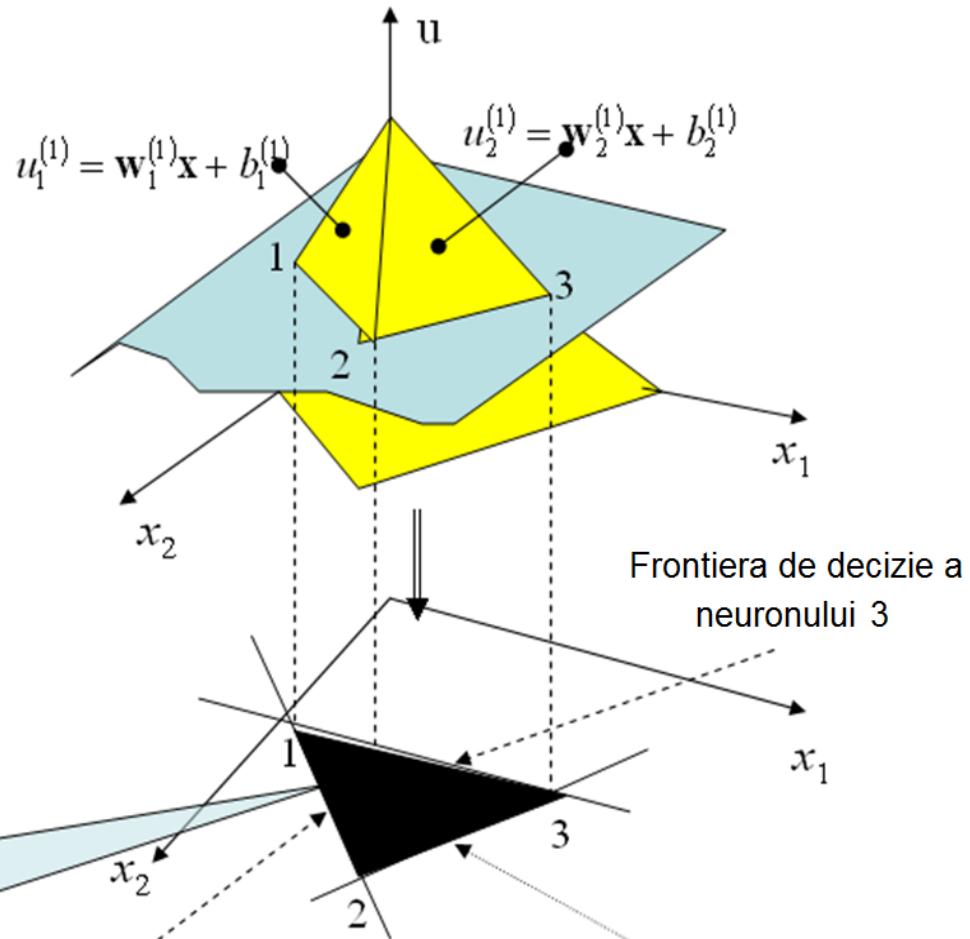
Strat 2

Frontieră convexă
separabilă în spațiul 2D

$$\Delta S_A = S_1 \cap S_2 \cap \bar{S}_3$$

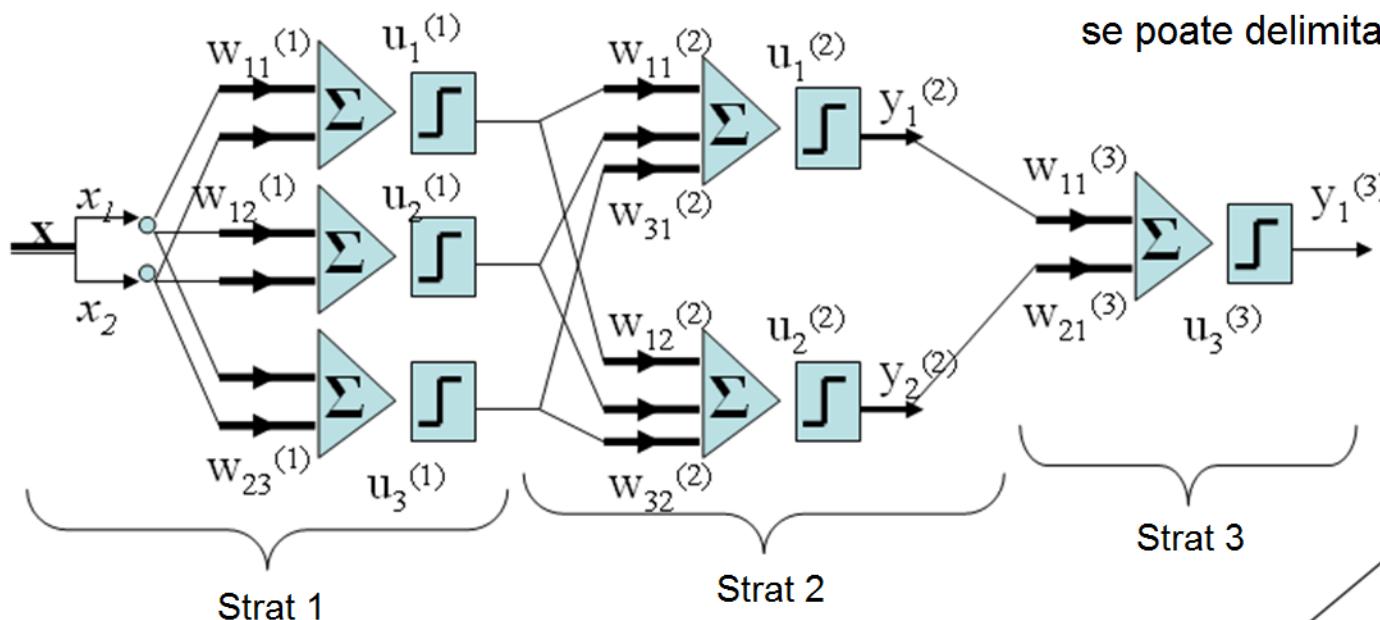
Frontiera de decizie a
neuronului 1

Frontiera de decizie a
neuronului 2



2 straturi ascunse

Folosind un perceptron cu 2 intrări și 3 straturi
se poate delimita o suprafață ne-convexă



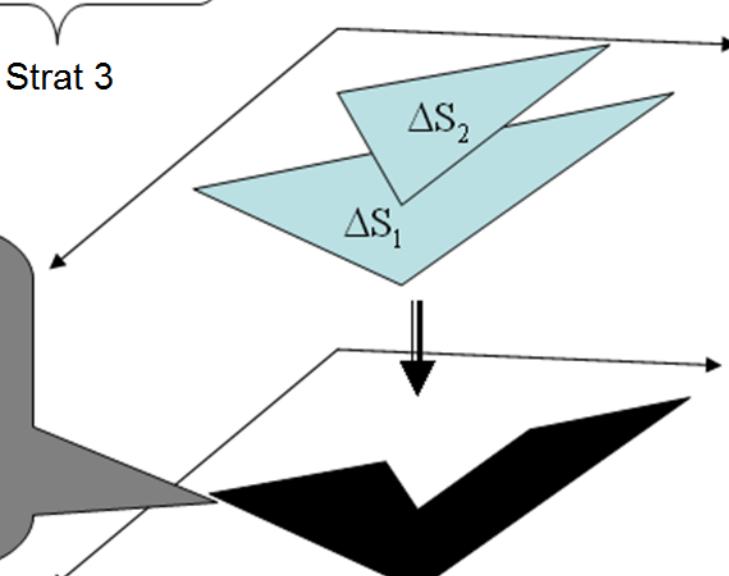
Suprafață concavă separabilă

$$Z = \Delta S_1 \cup \Delta \bar{S}_2$$

unde:

$$\Delta S_1 \equiv \left\{ \mathbf{x}_i \mid u_1^{(2)} \geq u_{\text{prag}} \right\}$$

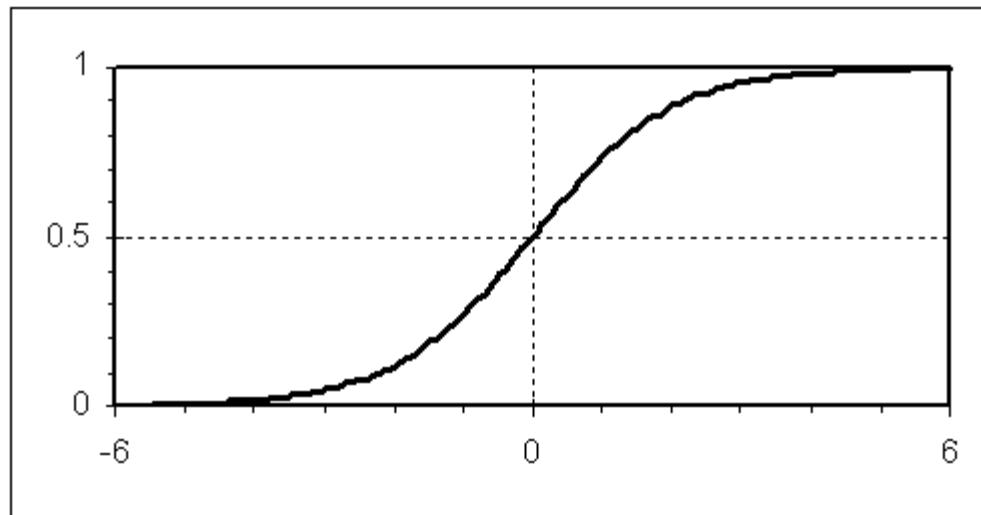
$$\Delta \bar{S}_2 \equiv \left\{ \mathbf{x}_i \mid u_2^{(2)} \geq u_{\text{prag}} \right\}$$



Functii de activare

- Functiile cel mai des utilizate sunt **sigmoide unipolară** (sau **logistică**):

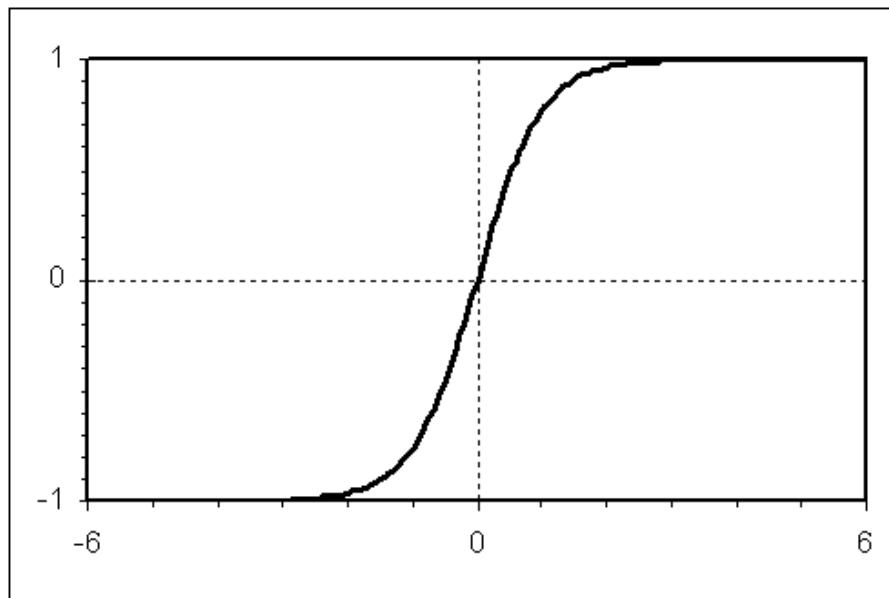
$$f(x) = \frac{1}{1 + e^{-x}}$$

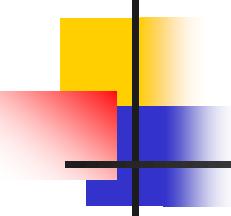


Funcții de activare

- și mai ales sigmoida bipolară (tangenta hiperbolică):

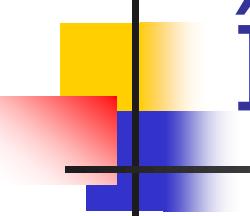
$$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$





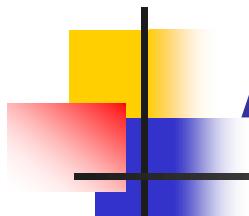
Discuție

- Un perceptron cu un singur strat are aceleasi limitări chiar dacă folosește o funcție de activare neliniară
- Un perceptron multi-strat cu funcții de activare liniare este echivalent cu un perceptron cu un singur strat
- O combinație liniară de funcții liniare este tot o funcție liniară. De exemplu:
 - $f(x) = 2x + 2$
 - $g(y) = 3y - 3$
 - $g(f(x)) = 3(2x + 2) - 3 = 6x + 3$



Învățarea

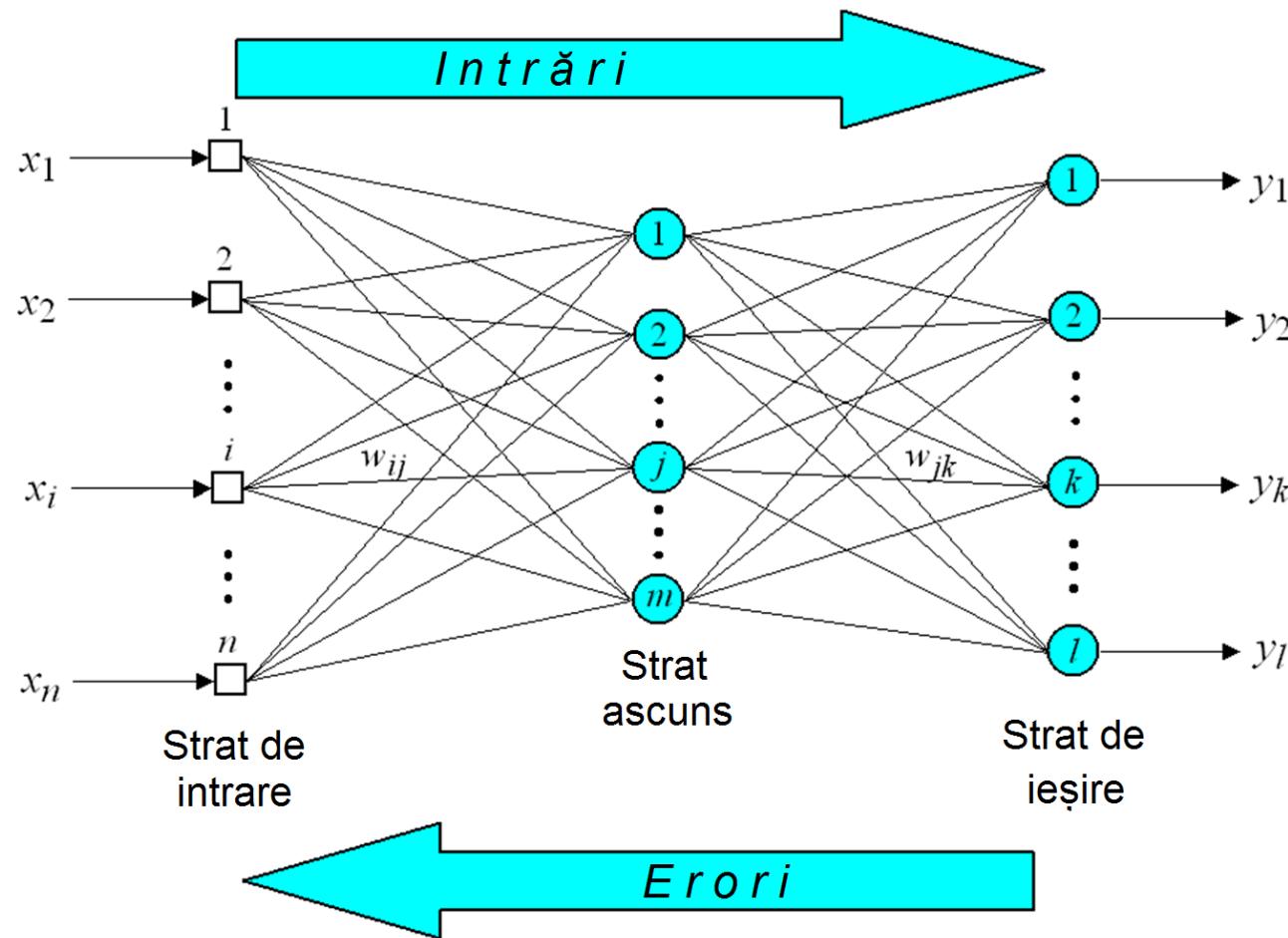
- O rețea multi-strat învăță într-un mod asemănător cu perceptronul
- Rețeaua primește vectorii de intrare și calculează vectorii de ieșire
- Dacă există o eroare (o diferență între ieșirea dorită și ieșirea efectivă), ponderile sunt ajustate pentru a reduce eroarea

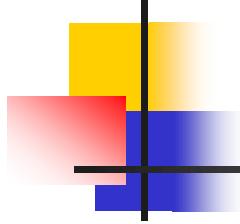


Algoritmul de retro-propagare

- engl. “backpropagation”
 - Bryson & Ho, 1969
 - Rumelhart, Hinton & Williams, 1986
- Algoritmul are două faze:
 - Rețeaua primește vectorul de intrare și propagă semnalul **înainte**, strat cu strat, până se generează ieșirea
 - Semnalul de **eroare** este propagat **înapoi**, de la stratul de ieșire către stratul de intrare, ajustându-se ponderile rețelei

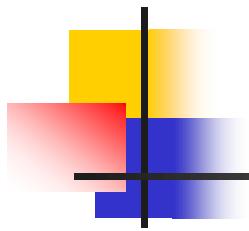
Fazele algoritmului de retro-propagare





Pasul 1. Inițializarea

- Ponderile și pragurile se inițializează cu valori aleatorii mici, dar diferite de 0
- În general, pot fi valori din intervalul [-0.1, 0.1]
- Ouristică recomandă valori din intervalul $(-2.4 / F_i, 2.4 / F_i)$, unde F_i este numărul de intrări al neuronului i (*fan-in*)
- Inițializarea ponderilor se face în mod independent pentru fiecare neuron
- În continuare, vom utiliza următorii indici: i pentru stratul de intrare, j pentru stratul ascuns și k pentru stratul de ieșire

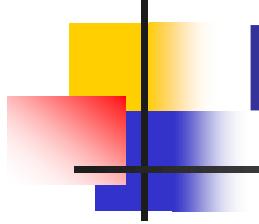


Pasul 2. Activarea

- Se activează rețeaua prin aplicarea vectorului de antrenare $x_1(p), x_2(p), \dots, x_n(p)$ cu ieșirile dorite $y^d_1(p), y^d_2(p), \dots, y^d_o(p)$
- Se calculează ieșirile neuronilor din stratul ascuns:

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

unde n este numărul de intrări ale neuronului j din stratul ascuns și se utilizează o funcție de activare sigmoidă



Pasul 2. Activarea

- Se calculează ieșirile reale ale neuronilor din stratul de ieșire:

$$y_k(p) = \text{sigmoid} \left[\sum_{j=1}^m y_j(p) \cdot w_{jk}(p) - \theta_k \right]$$

unde m este numărul de intrări al neuronului k din stratul de ieșire

Pasul 3a. Ajustarea ponderilor

- Se calculează **gradientii de eroare** ai neuronilor din stratul de ieșire, în mod similar cu regula delta pentru perceptron

$$\delta_k(p) = \overbrace{y_k(p) \cdot [1 - y_k(p)]}^{\text{gradient de eroare}} \cdot e_k(p)$$

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

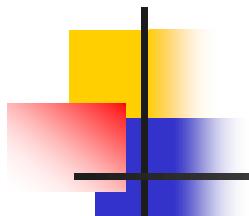
explicație în
slide-ul următor

Se calculează corectiile ponderilor:

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

Se actualizează ponderile neuronilor de ieșire:

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



Derivata funcției de activare

Dacă folosim sigmoida unipolară, derivata acesteia este:

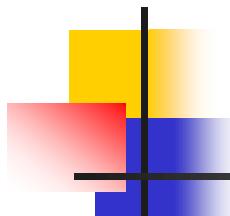
$$f'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = f(x) \cdot (1 - f(x)).$$

Dacă folosim sigmoida bipolară, derivata acesteia este:

$$f'(x) = \frac{2a \cdot e^{-ax}}{(1 + e^{-ax})^2} = \frac{a}{2} \cdot (1 - f(x)) \cdot (1 + f(x)).$$

În continuare, să presupunem că funcția utilizată este sigmoida unipolară:

$$\delta_k(p) = y_k(p) \cdot (1 - y_k(p)) \cdot e_k(p).$$



Pasul 3b. Ajustarea ponderilor

- Se calculează gradientii de eroare ai neuronilor din stratul ascuns:

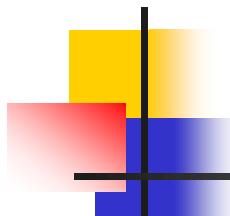
$$\delta_j(p) = y_j(p) \cdot [1 - y_j(p)] \cdot \sum_{k=1}^l \delta_k(p) w_{jk}(p)$$

- Se calculează corecțiile ponderilor:

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

- Se actualizează ponderile neuronilor din stratul ascuns:

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

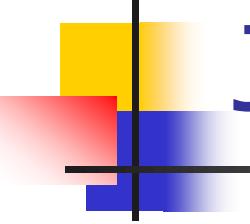


Pasul 4. Iterația

- Se incrementează p
- Prezentarea tuturor vectorilor (instanțelor) de antrenare constituie o **epochă**
- Antrenarea rețelei continuă până când **eroarea medie pătratică** ajunge sub un prag acceptabil sau până când se atinge un număr maxim prestabilit de epoci de antrenare

$$E = \frac{1}{V} \sum_{p=1}^V \sum_{k=1}^O \left(y_k^d(p) - y_k(p) \right)^2$$

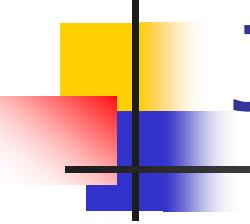
V este numărul de vectori de antrenare
O este numărul de ieșiri



Justificare

- Formulele de calcul al gradientilor se bazează pe ideea minimizării erorii rețelei în raport cu ponderile
- Însă $\partial E / \partial w$ nu se poate calcula direct, astfel încât se aplică regula de înlănțuire:

$$\frac{\partial E}{\partial w} = \underbrace{\frac{\partial E}{\partial y}}_e \cdot \underbrace{\frac{\partial y}{\partial net}}_{f'} \cdot \underbrace{\frac{\partial net}{\partial w}}_x$$



Justificare

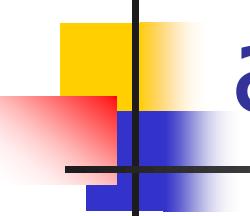
- Formulele de calcul al gradientilor se bazează pe ideea minimizării erorii rețelei în raport cu ponderile
- Însă $\partial E / \partial w$ nu se poate calcula direct, astfel încât se aplică regula de înlănțuire:

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial net} \cdot \frac{\partial net}{\partial w}$$

- Diferențialele din partea dreaptă se pot calcula, iar în final ponderile sunt ajustate:

$$\Delta w_{jk} = \alpha \cdot y_j \cdot [y_k (1 - y_k)] \cdot e_k$$

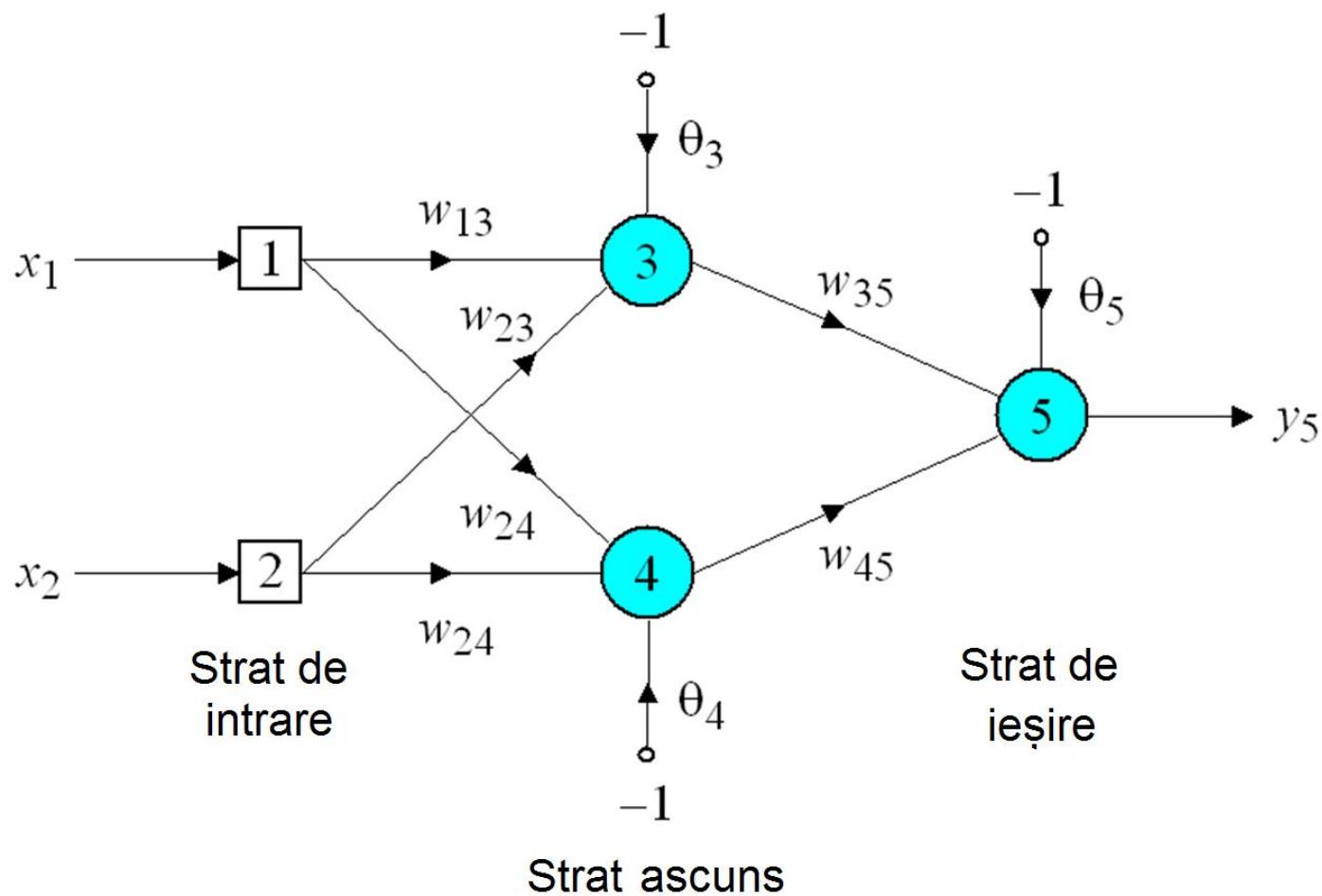
$$\Delta w_{ij} = \alpha \cdot x_i \cdot [y_j (1 - y_j)] \cdot \sum_k \delta_k w_{jk}$$

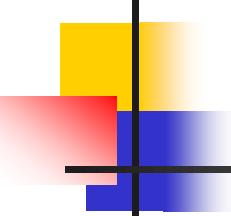


Rețele cu mai multe straturi ascunse

- Ajustarea ponderilor din stratul de ieșire se face la fel ca în pasul 3a
- Ajustarea ponderilor din fiecare strat ascuns se face ca în pasul 3b
- Dacă rețeaua are multe straturi ascunse, gradienții sunt mici iar antrenarea se face foarte lent sau nu converge deloc
- De aceea, pentru arhitecturile recente de învățare profundă există și alte metode de antrenare

Exemplu: rețea cu un strat ascuns pentru aproximarea funcției binare XOR





Inițializarea

- Pragul aplicat unui neuron dintr-un strat ascuns sau de ieșire este echivalent cu o altă conexiune cu ponderea egală cu θ , conectată la o intrare fixă egală cu -1
- Ponderile și pragurile sunt inițializate aleatoriu, de exemplu:
 - $w_{13} = 0.5, w_{14} = 0.9, w_{23} = 0.4, w_{24} = 1.0$
 - $w_{35} = 1.2, w_{45} = 1.1$
 - $\theta_3 = 0.8, \theta_4 = 0.1, \theta_5 = 0.3$

- Să considerăm vectorul de antrenare unde intrările x_1 și x_2 sunt egale cu 1, iar ieșirea dorită y_5^d este 0
- Ieșirile reale ale neuronilor 3 și 4 din stratul ascuns sunt calculate precum urmează:

$$y_3 = \text{sigmoid}(x_1 w_{13} + x_2 w_{23} - \theta_3) = 1 / \left[1 + e^{-(1 \cdot 0.5 + 1 \cdot 0.4 - 1 \cdot 0.8)} \right] = 0.5250$$

$$y_4 = \text{sigmoid}(x_1 w_{14} + x_2 w_{24} - \theta_4) = 1 / \left[1 + e^{-(1 \cdot 0.9 + 1 \cdot 1.0 + 1 \cdot 0.1)} \right] = 0.8808$$

- Se determină ieșirea reală a neuronului 5 din stratul de ieșire:

$$y_5 = \text{sigmoid}(y_3 w_{35} + y_4 w_{45} - \theta_5) = 1 / \left[1 + e^{(-0.5250 \cdot 1.2 + 0.8808 \cdot 1.1 - 1 \cdot 0.3)} \right] = 0.5097$$

- Se obține următoarea eroare:

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

- Se propagă eroarea e din stratul de ieșire către stratul de intrare
- Mai întâi se calculează gradientul erorii pentru neuronul 5 din stratul de ieșire:

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

- Apoi se calculează corecțiile ponderilor și pragului
 - Presupunem că rata de învățare α este 0.1

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

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

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

- Apoi se calculează gradientii de eroare pentru neuronii 3 și 4 din stratul ascuns:

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

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

- Apoi se determină corecțiile ponderilor și pragurilor:

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

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

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

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

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

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

- În final, se actualizează ponderile și pragurile:

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

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

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

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

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

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

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

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

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

- Procesul de antrenare se repetă până când suma erorilor pătratice devine mai mică decât o valoare acceptabilă, de exemplu aici 0.001

Rezultate finale

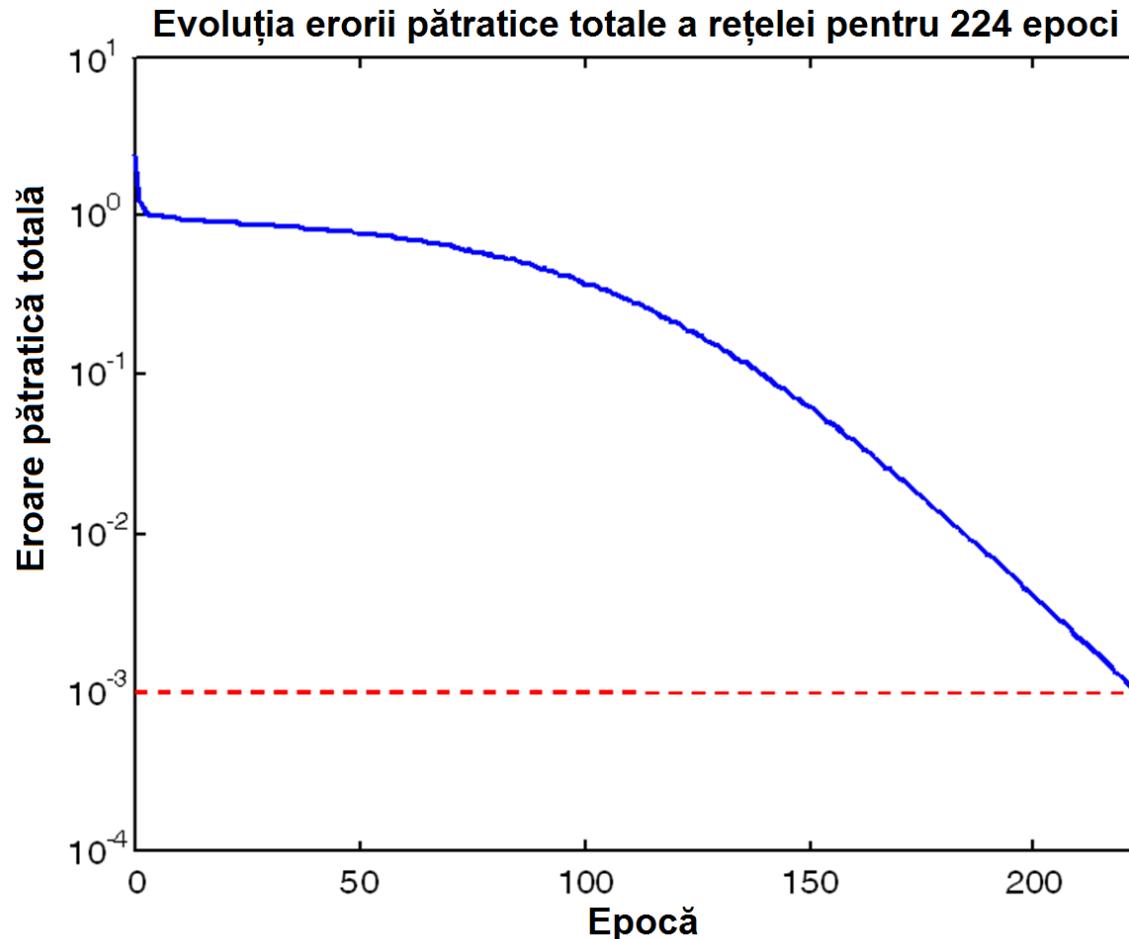
Intrări		leșire dorită	leșire reală	Eroare	Suma erorilor pătratice
x_1	x_2	y_d	y_5	e	
1	1	0	0.0155	-0.0155	0.0010
0	1	1	0.9849	0.0151	
1	0	1	0.9849	0.0151	
0	0	0	0.0175	-0.0175	

V este numărul de vectori de antrenare (aici 4)

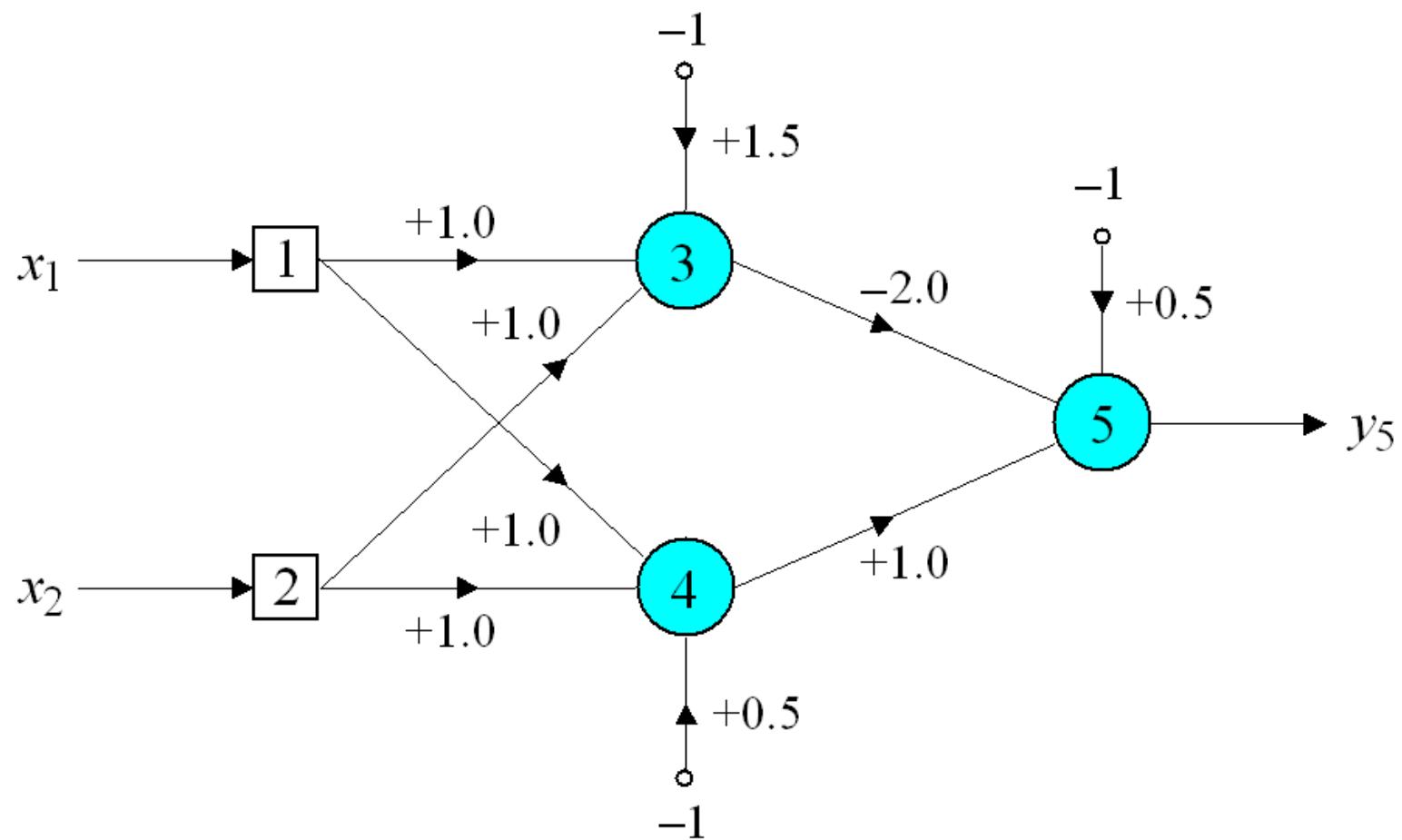
O este numărul de ieșiri (aici 1)

$$E = \frac{1}{V} \sum_{p=1}^V \sum_{k=1}^O (y_k^d(p) - y_k(p))^2$$

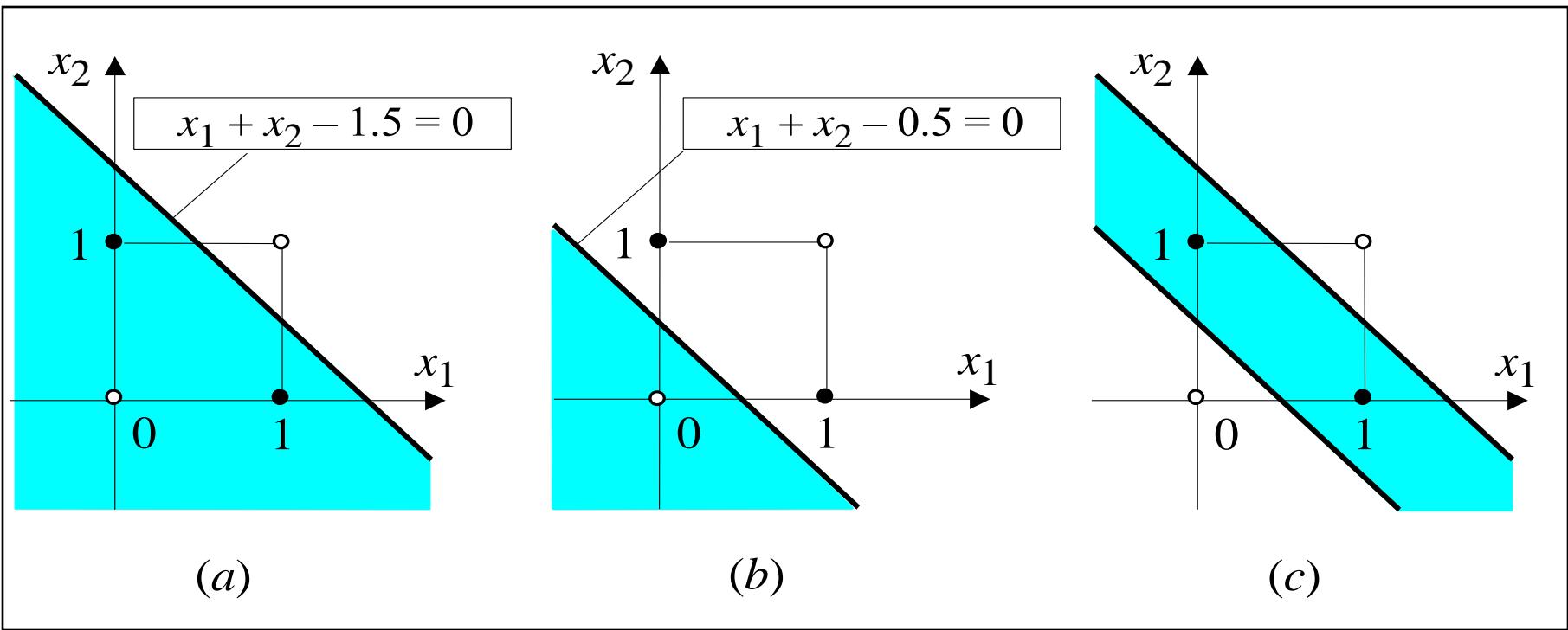
Evoluția erorii pentru problema XOR



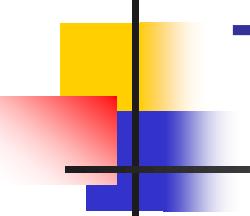
Parametrii finali ai rețelei



Regiuni de decizie

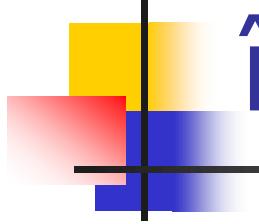


- (a) Regiunea de decizie generată de neuronul ascuns 3
- (b) Regiunea de decizie generată de neuronul ascuns 4
- (c) Regiunea de decizie generată de rețeaua completă: combinarea regiunilor de către neuronul 5



Tipuri de antrenare

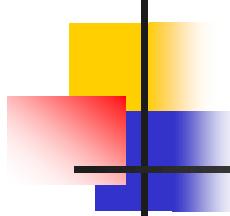
- Învățarea incrementală (*online learning*)
 - Ponderile se actualizează după prelucrarea **fiecărui** vector de antrenare
 - Ca în exemplul anterior
- Învățarea pe lot (*batch learning*)
 - După prelucrarea unui vector de antrenare se acumulează gradienții de eroare în corectările ponderilor Δw
 - Ponderile se actualizează o singură dată la sfârșitul unei epoci, după prezentarea **tuturor** vectorilor de antrenare:
 $w \leftarrow w + \Delta w$
 - **Avantaj:** rezultatele antrenării nu mai depind de ordinea în care sunt prezențați vectorii de antrenare



Metode de accelerare a învățării

- Uneori, rețelele învăță mai repede atunci când se folosește funcția sigmoidă bipolară (tangenta hiperbolică) în locul sigmoidei unipolare

$$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$



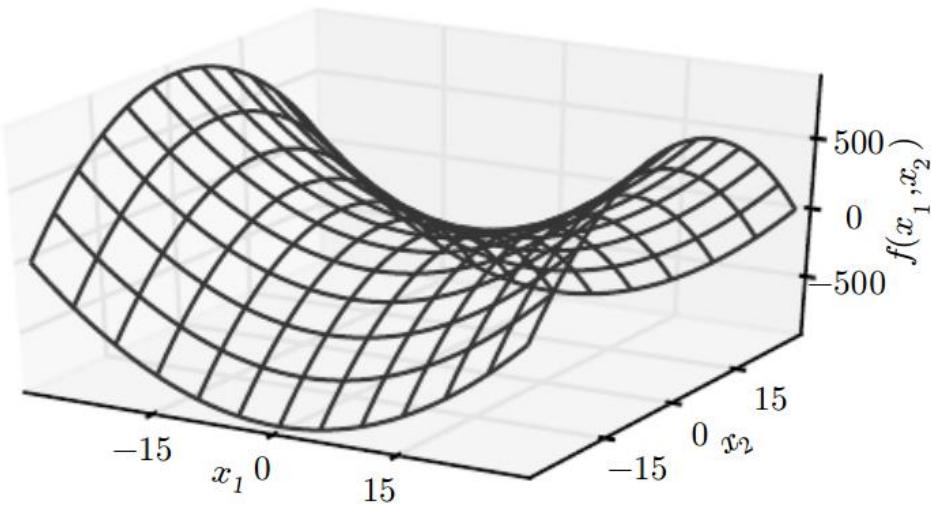
Metoda momentului

- Regula delta generalizată:

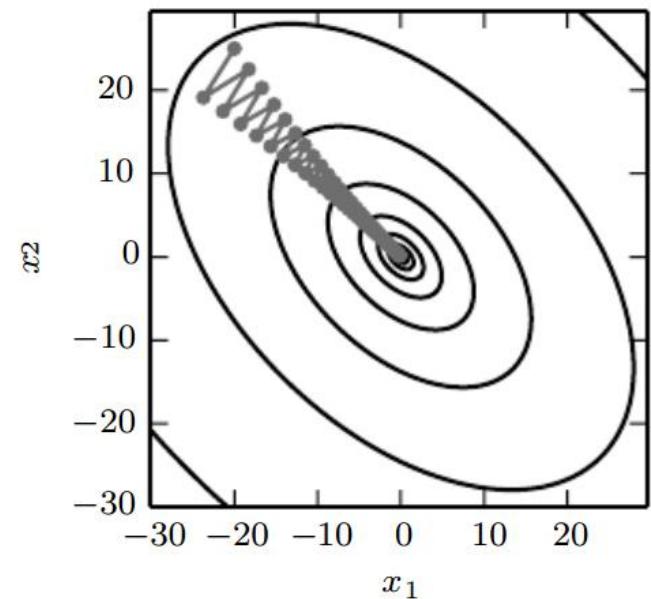
$$\Delta w_{jk}(p) = \beta \cdot \Delta w_{jk}(p-1) + \alpha \cdot y_j(p) \cdot \delta_k(p)$$

- Termenul β este un număr pozitiv ($0 \leq \beta < 1$) numit **constantă de moment sau inertie (momentum)**
- De obicei, β este în jur de 0.95

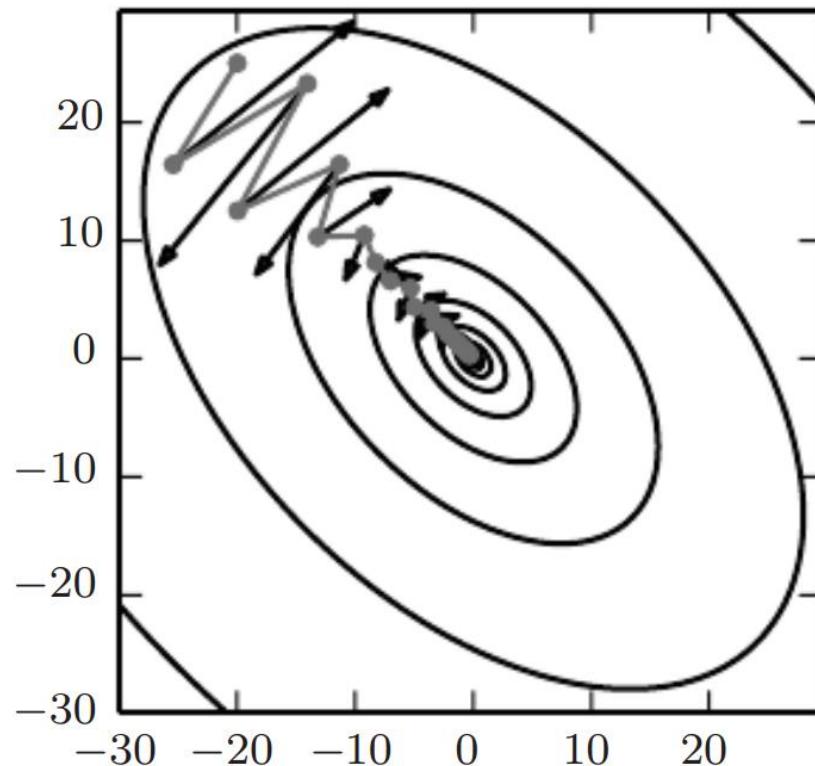
Exemplu: Gradient Descent



$$f(\mathbf{x}) = x_1^2 - x_2^2$$

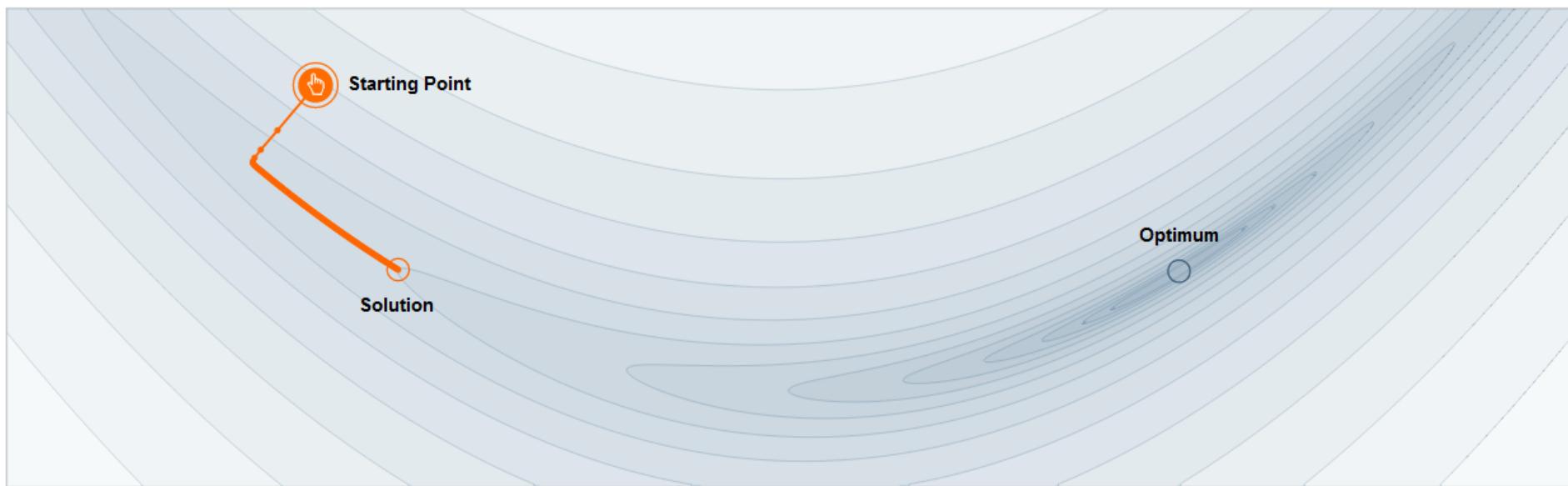


Exemplu: GD cu moment



cu gri: GD+M
cu negru: ce pas ar face GD

Rata de învățare și momentul



Step-size $\alpha = 0.00096$



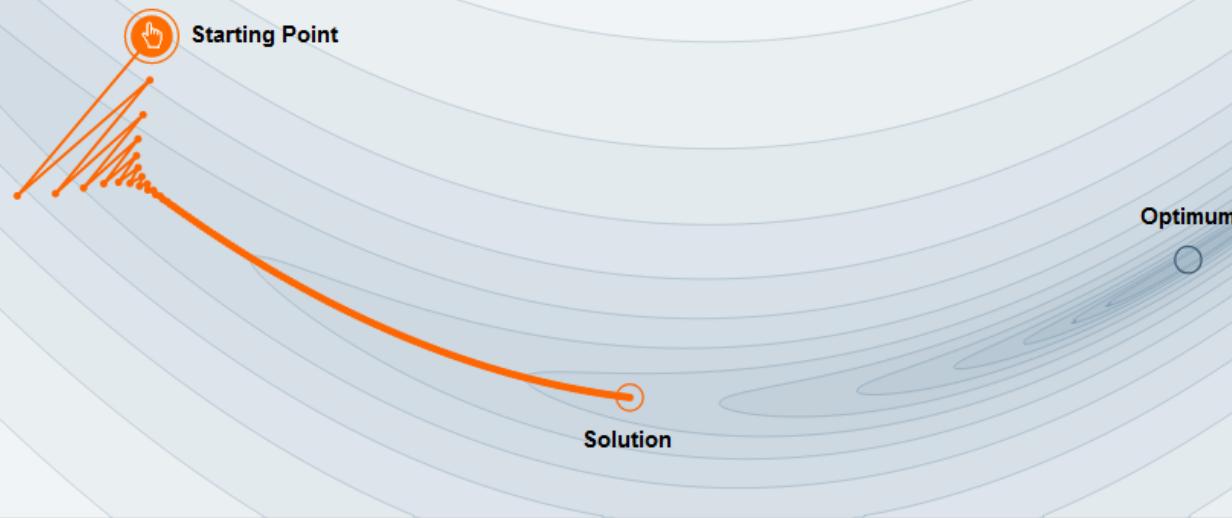
Momentum $\beta = 0.0$



We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

Why Momentum Really Works, <https://distill.pub/2017/momentum/>

Rata de învățare și momentul



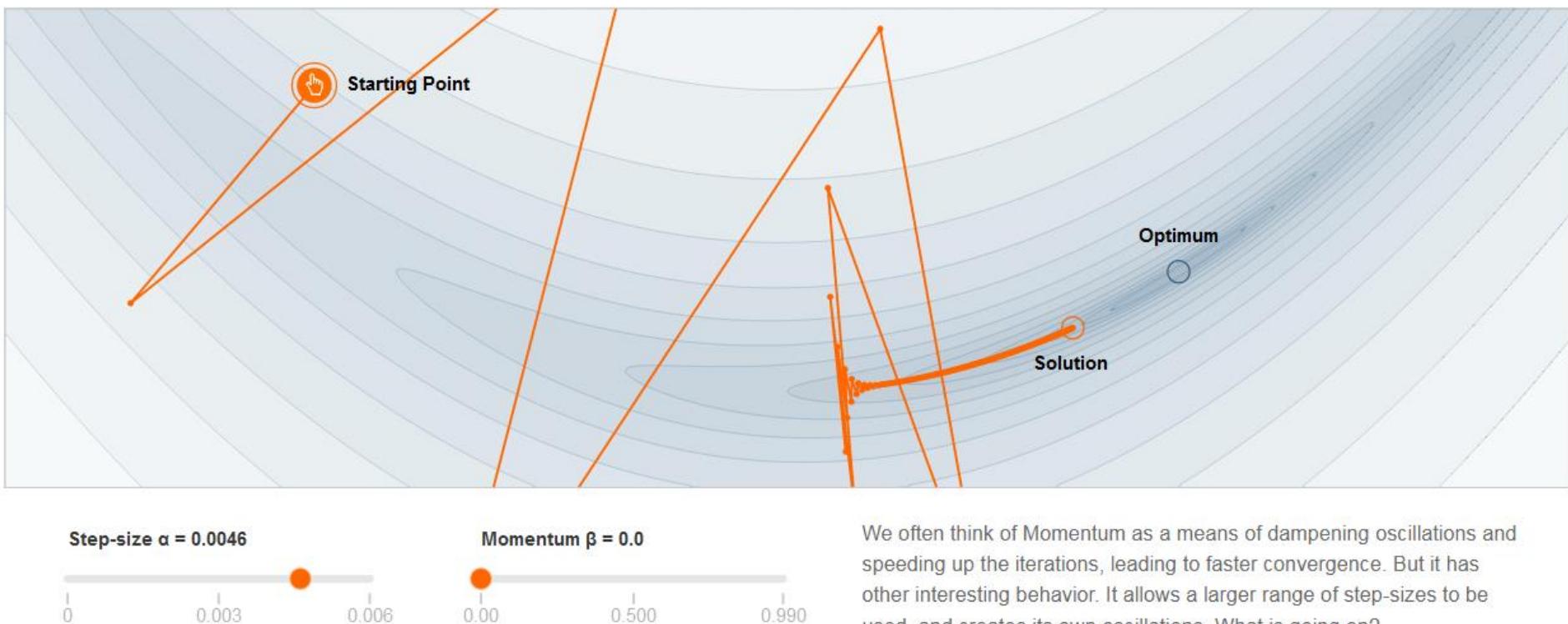
Step-size $\alpha = 0.0028$

Momentum $\beta = 0.0$

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

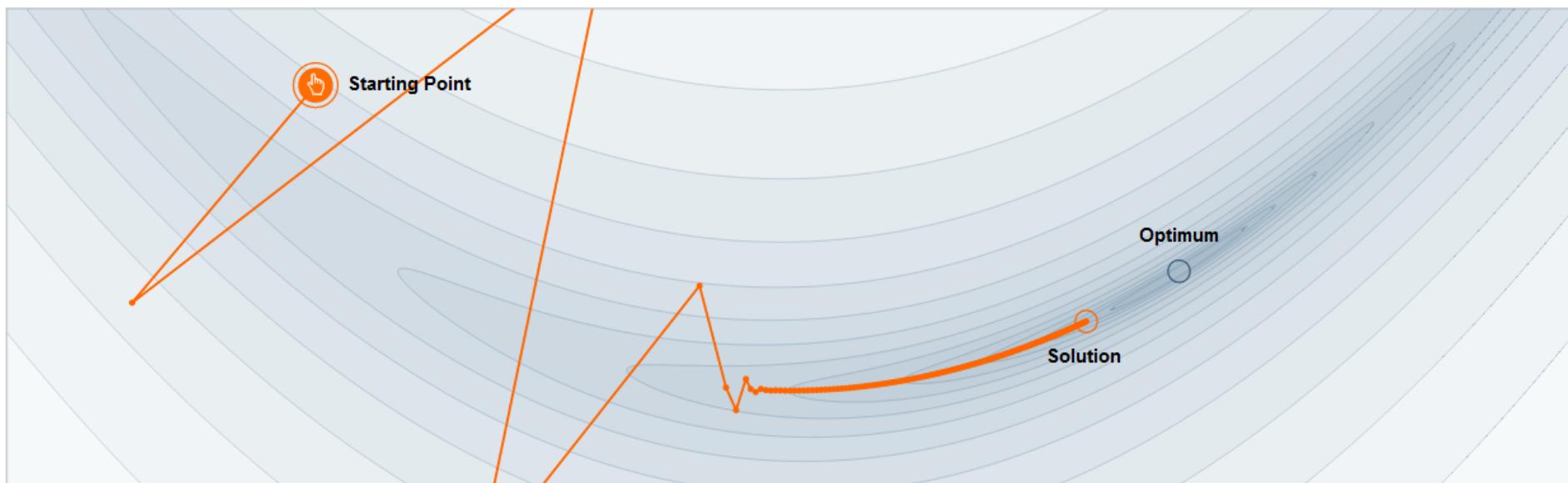
Why Momentum Really Works, <https://distill.pub/2017/momentum/>

Rata de învățare și momentul



Why Momentum Really Works, <https://distill.pub/2017/momentum/>

Rata de învățare și momentul



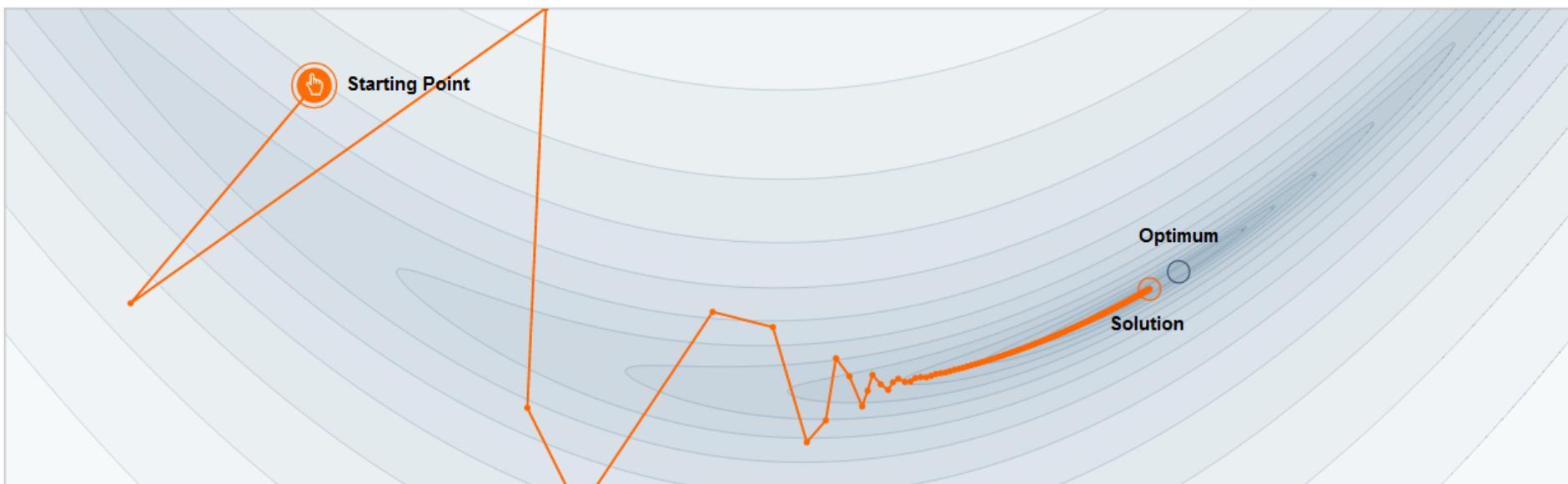
Step-size $\alpha = 0.0046$

Momentum $\beta = 0.22$

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

Why Momentum Really Works, <https://distill.pub/2017/momentum/>

Rata de învățare și momentul



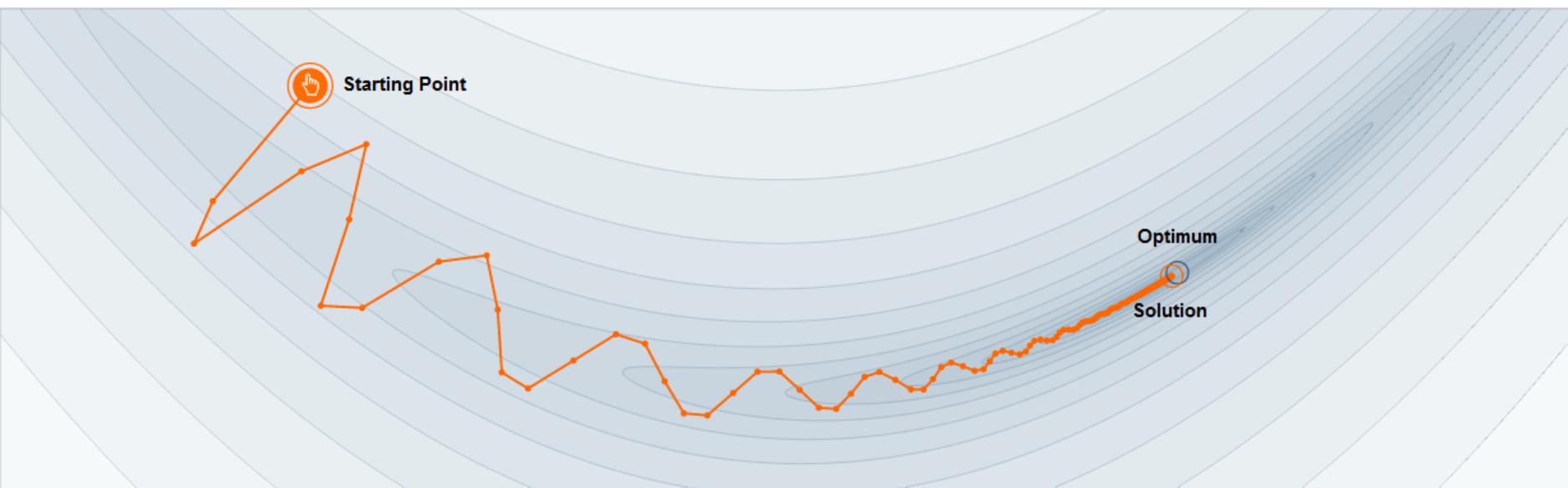
Step-size $\alpha = 0.0046$

Momentum $\beta = 0.55$

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

Why Momentum Really Works, <https://distill.pub/2017/momentum/>

Rata de învățare și momentul



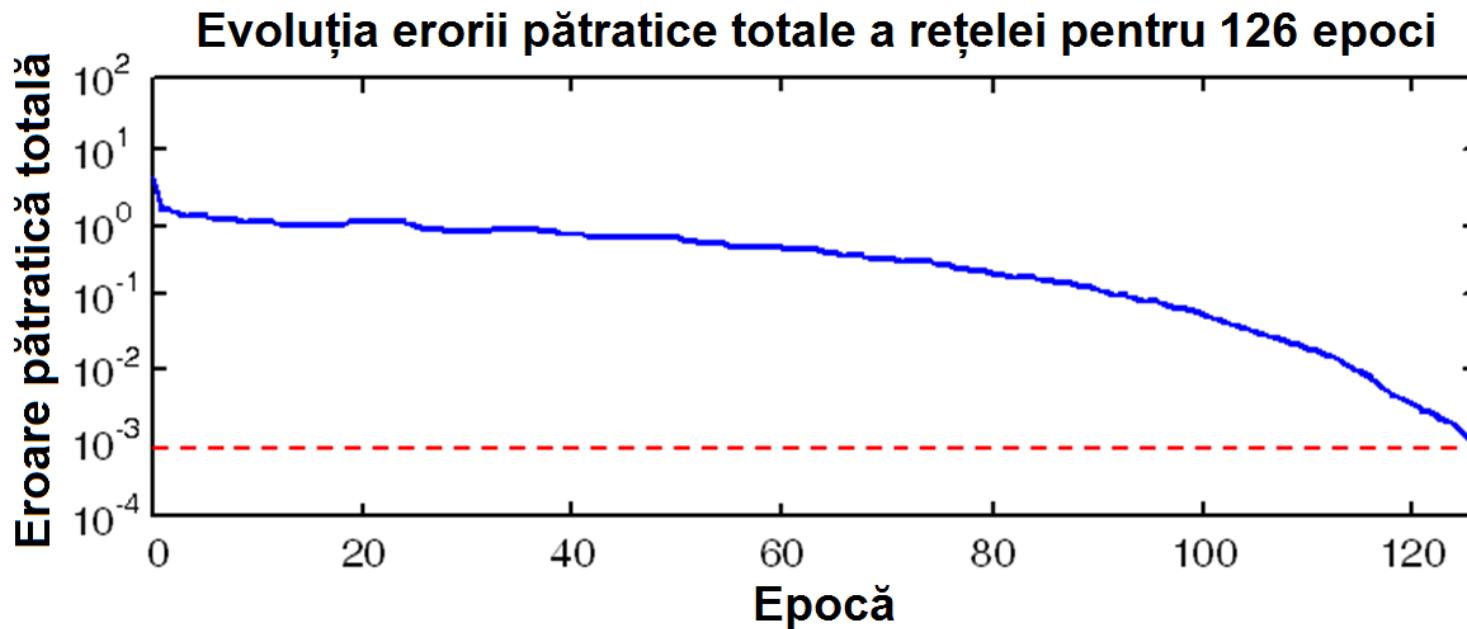
Step-size $\alpha = 0.0024$

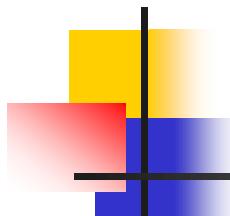
Momentum $\beta = 0.85$

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

Why Momentum Really Works, <https://distill.pub/2017/momentum/>

Învățarea cu moment pentru problema XOR





Rata de învățare adaptivă

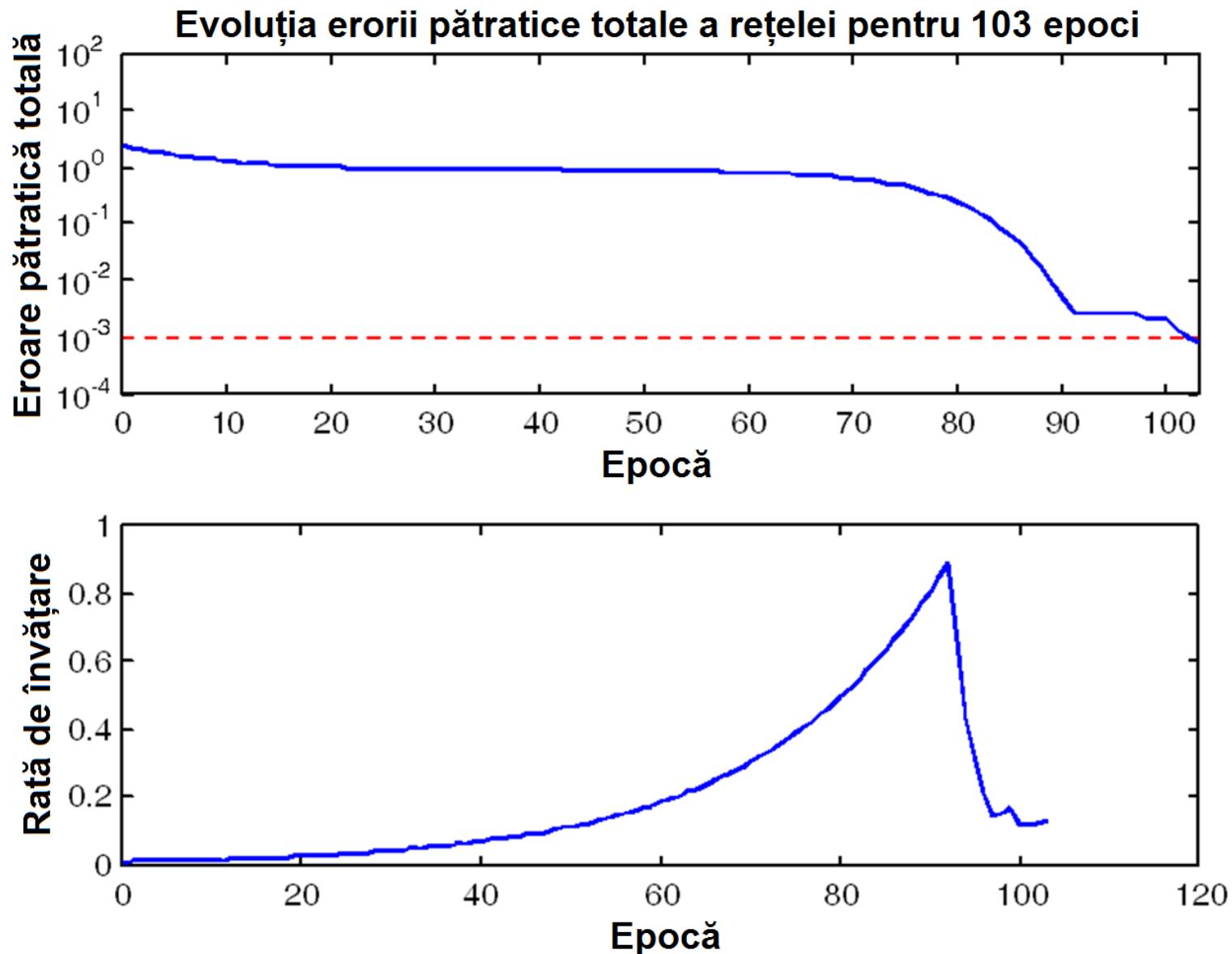
- Pentru accelerarea convergenței și evitarea instabilității, se pot aplica două euristică:
 - Dacă variația sumei erorilor pătratice ΔE are același semn algebric pentru mai multe epoci consecutive, atunci rata de învățare α trebuie să crească
 - Dacă semnul lui ΔE alternează timp de câteva epoci consecutive, atunci rata de învățare α trebuie să scadă

Rata de învățare adaptivă

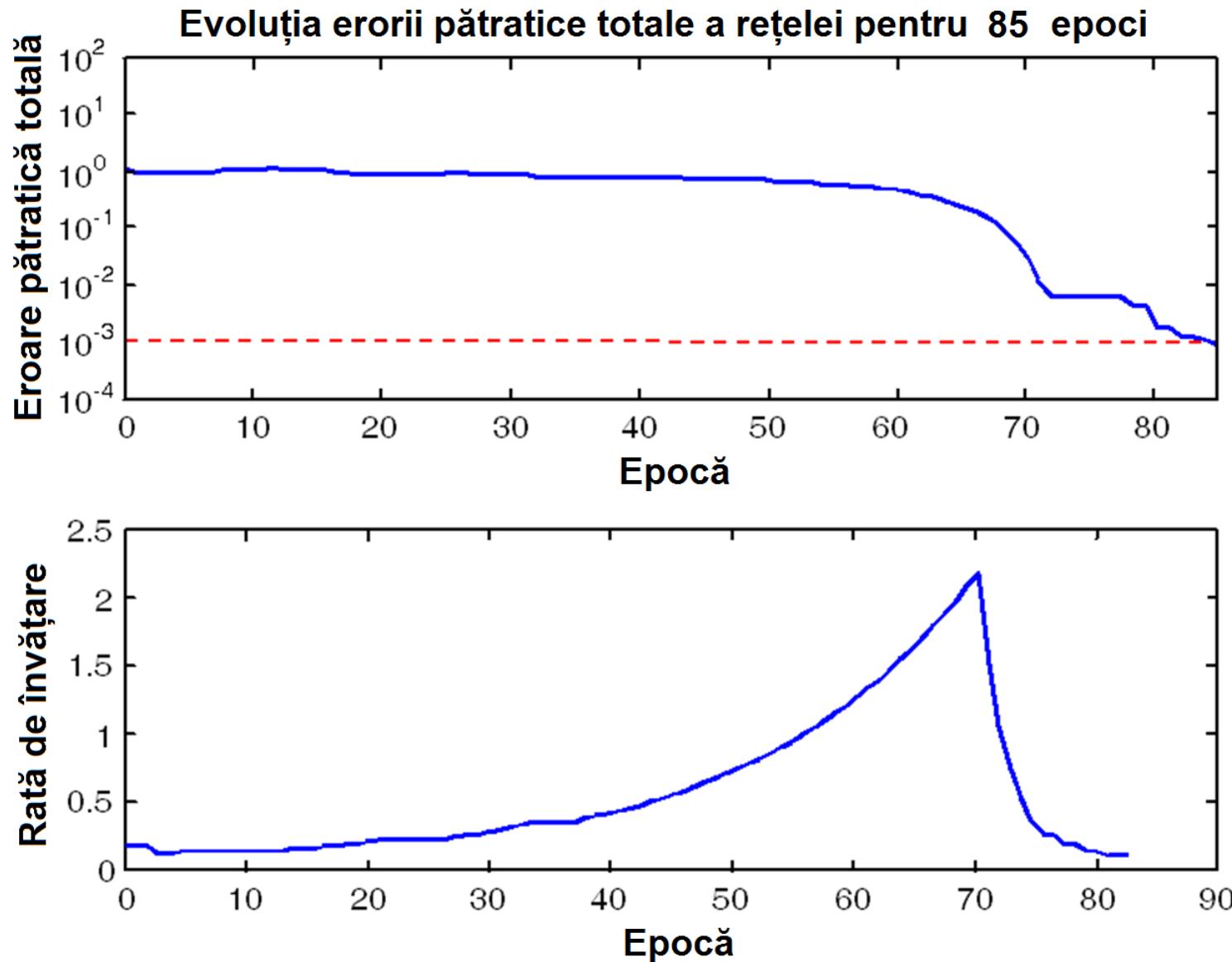
- Fie E_p suma erorilor pătratice în epoca prezentă
- Dacă $E_p > r \cdot E_{p-1}$, atunci $\alpha \leftarrow \alpha \cdot d$
- Dacă $E_p < E_{p-1}$, atunci $\alpha \leftarrow \alpha \cdot u$
- Apoi se calculează noile ponderi
- Valori tipice: $r = 1.04$, $d = 0.7$, $u = 1.05$

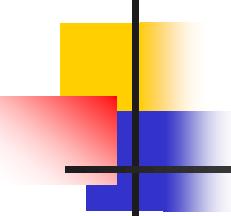


Învățarea cu rată adaptivă



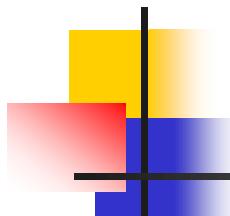
Învățarea cu moment și rată adaptivă





Considerente practice

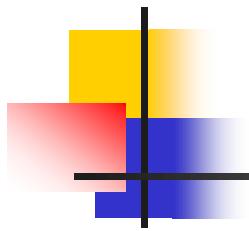
- Pentru a nu satura funcțiile sigmoide, ceea ce ar conduce la scăderea vitezei de convergență, intrările și ieșirile sunt scalate de obicei în intervalul $[0.1, 0.9]$, sau $[-0.9, 0.9]$, în funcție de tipul de sigmoidă folosit
- Dacă rețeaua este utilizată pentru regresie și nu pentru clasificare, funcția de activare a neuronilor de ieșire poate fi liniară sau semiliniară (limitată la 0 și 1, respectiv la -1 și 1) în loc de sigmoidă
- Stabilirea numărului de straturi și mai ales a numărului de neuroni din fiecare strat este relativ dificilă și poate necesita numeroase încercări pentru a atinge performanțele dorite



Euristică: relația lui Kudrycki

- $H = 3 \cdot O$ sau
- $H_1 = 3 \cdot H_2$

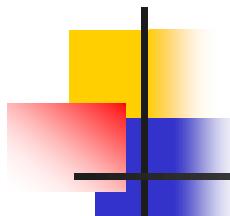
- unde:
 - O numărul de ieșiri
 - H numărul de neuroni din stratul ascuns (pentru un singur strat ascuns)
 - H_1 numărul de neuroni din primul strat ascuns și H_2 numărul de neuroni din al doilea strat ascuns (pentru 2 straturi ascunse)



Rețele neuronale

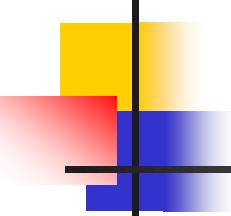
1. Introducere
2. Perceptronul. Adaline
3. Perceptronul multi-strat
4. Rețele profunde
5. Concluzii





Rețele clasice și rețele profunde

- Rețele clasice
 - 1-2 straturi
 - Funcții de activare sigmoide
 - Funcții de cost bazate pe MSE
 - Algoritmi de antrenare: Backpropagation, RProp, Levenberg-Marquardt etc.
- Rețele profunde
 - Mai multe straturi
 - Funcții de activare mai simple: ReLU
 - Funcții de cost bazate pe MLE
 - Algoritmi de antrenare: SGD, RMSProp, Adam etc.
 - Alte metode de inițializare a ponderilor, regularizare, pre-antrenare
- În afară de numărul de straturi, diferențele nu sunt stricte!



Gradienti instabili

- Primele straturi ale unui perceptron multi-strat clasic cu un număr mai mare de straturi ascunse nu se antrenează bine
- Când ponderile sunt mici sau funcțiile de activare sigmoide sunt saturate (gradienti mici), corecțiile ponderilor Δw sunt mici, iar antrenarea este foarte lentă
- De asemenea, ultimele straturi, cele apropiate de ieșire, pot învăța problema „destul de bine” și deci semnalul de eroare trimis către primele straturi devine și mai mic
- Sunt necesare alte metode de antrenare pentru a pune în valoare primele straturi

Functia de activare ReLU

- *ReLU (Rectified Linear Unit)*

$$f(x) = \max(0, x)$$

- *Leaky ReLU*

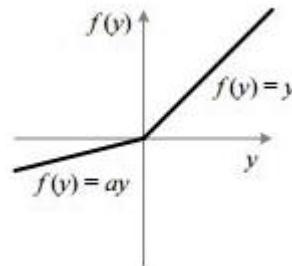
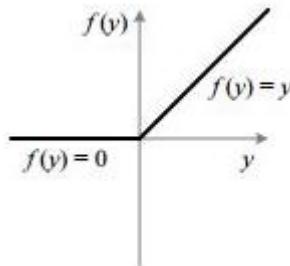
$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases}$$

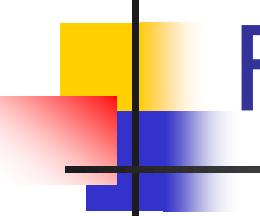
- *Parametric ReLU (PReLU)*

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ ax & \text{otherwise} \end{cases}$$



a poate fi învățat





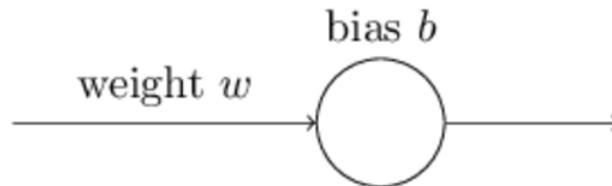
Funcția de activare ReLU

- ReLU este o funcție de activare recomandată în multe situații pentru rețele profunde
- Este neliniară
- Este ușor de optimizat prin metode diferențiale
 - Chiar dacă în partea negativă gradientul este 0, în practică funcționează bine
 - S-a observat că învățarea este de circa 6 ori mai rapidă față de funcțiile sigmoide

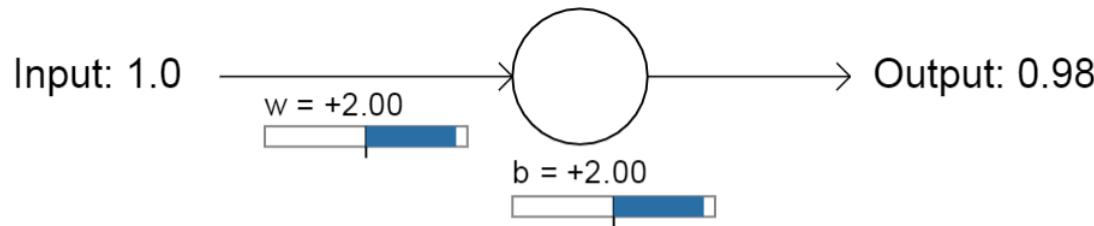
$$\frac{df}{dx} = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x < 0 \end{cases}$$

Functia de cost

- Exemplu: un neuron care primește intrarea 1 și trebuie să producă ieșirea 0

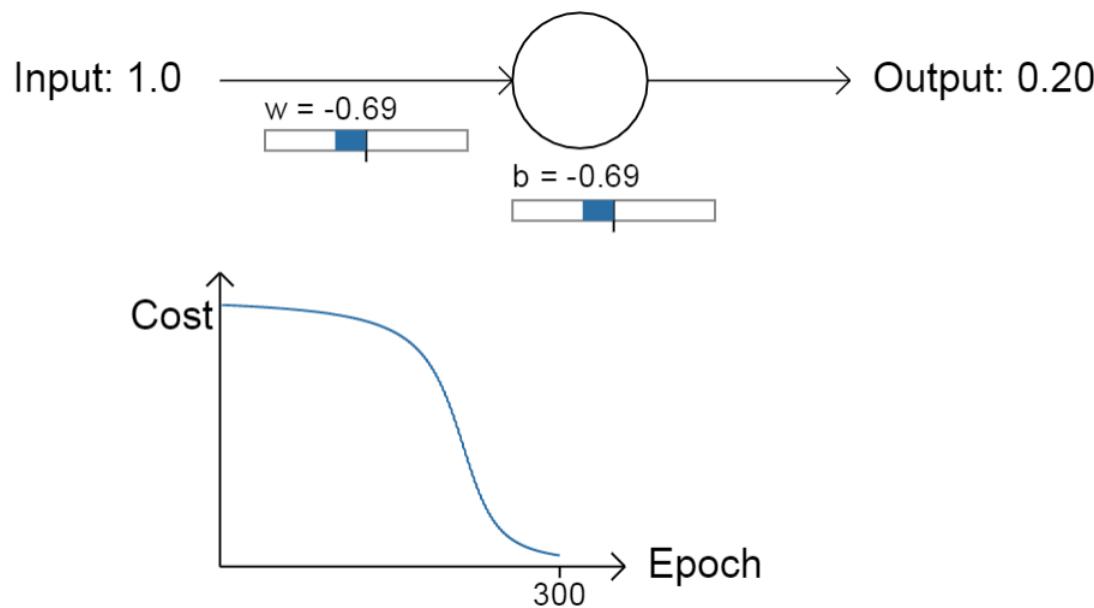


- Starea inițială



Antrenarea

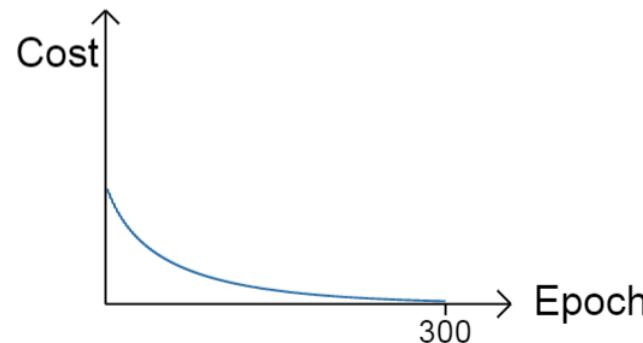
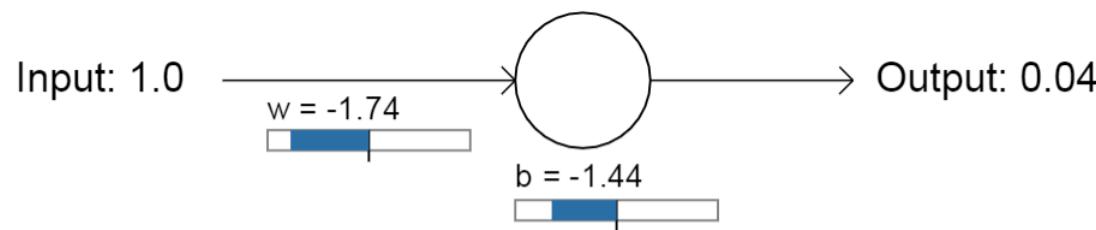
- Cu funcția de cost MSE

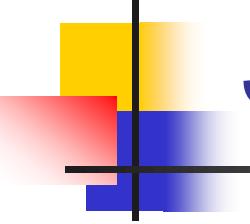


Antrenarea

- Cu funcția de cost *cross-entropy*

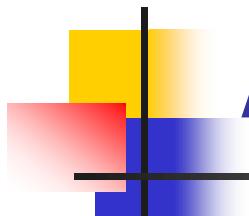
$$w_j = w_j + \alpha \sum_{i=1}^n x_{ij} (y_i - p_i)$$





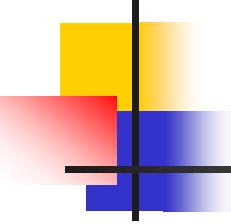
Stochastic Gradient Descent

- Pentru problemele actuale, numărul instanțelor de antrenare poate fi foarte mare
- SGD lucrează asemănător cu gradientul DESCENDENT, dar nu ia în calcul toate instanțele la calcularea gradientilor, ci doar un **mini-lot** (*minibatch*)
- La fiecare iterație, se aleg aleatoriu alte instanțe
- Gradientii sunt doar o aproximatie a gradientilor reali
- Complexitatea de timp poate fi controlată prin dimensiunea mini-lotului



Alți algoritmi de antrenare

- RMSProp (Root Mean Square Propagation)
- Adam (Adaptive Moment)
- RMSProp cu moment Nesterov
- AdaDelta
- AdaGrad
- AdaMax
- Nadam (Nesterov-Adam)



Concluzii

- Perceptronul lui Rosenblatt poate învăța funcții separabile liniar
- Perceptronul multi-strat poate aproxima funcții neseparabile liniar, cu regiuni de decizie complexe
- Cel mai utilizat algoritm de antrenare pentru perceptronul multi-strat este algoritmul de retro-propagare a erorilor (*backpropagation*)
- Rețelele profunde, pe lângă un număr mai mare de straturi, pot avea funcții de activare și algoritmi de antrenare diferenți de rețelele clasice, pe lângă alte optimizări