

## Laborator 10

### Crearea unei rețele feedforward

Se creează cu funcția **newff** astfel:

`newff ( PR, { $k_1, \dots, k_m$ }, { $f_1, \dots, f_m$ }, BTF, BLF, PF )`

PR – [ $\min_i \max_i; \dots \min_n \max_n$ ], unde  $\min_i \max_i$  reprezintă valoarea minimă, respectiv valoarea maximă pentru semnalul de intrare în unitatea de intrare  $i$ .

$k_i$  - numărul de neuroni aflați pe nivelul  $i$

$f_i$  - funcția de transfer a nivelului  $i$ . Implicit este 'tansig' pentru nivelul ascuns și 'purelin' pentru nivelul de ieșire

BTF – funcția de antrenare a rețelei folosind algoritmul backprop, implicit = 'trainlm'(metoda Levenberg-Marquart ).

BLF – funcția de învățare a ponderilor/bias-ului, implicit = 'learngdm'(algoritmul gradientului descendent cu moment).

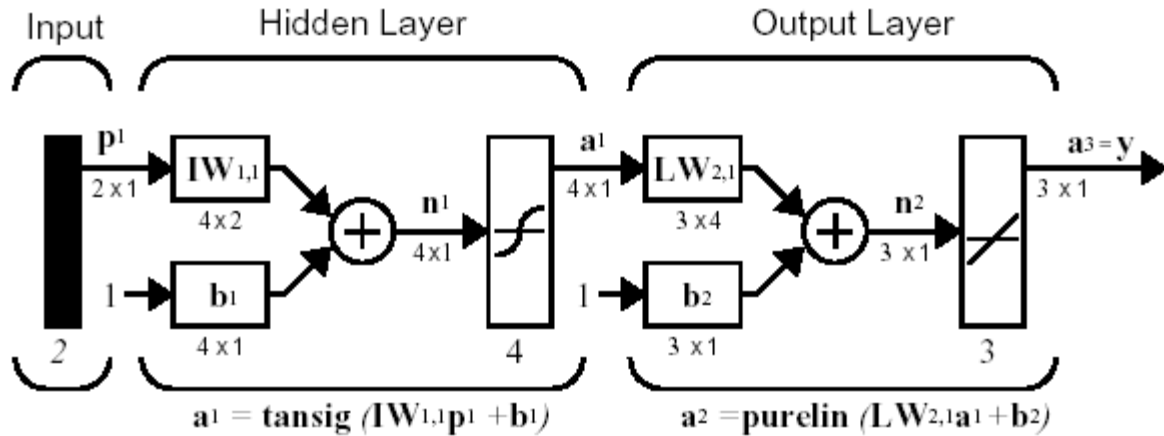
PF - funcția de performanță, implicit = 'mse'(media pătratelor erorilor).

**Backpropagarea** este o metoda de antrenare a rețelelor neurale, obținută prin generalizarea algoritmului Widrow-Hoff pentru rețele cu mai multe *layere* si cu funcții de transfer derivabile.

Funcții de transfer folosite: **logsig**, **tansig**, **purelin**.

Pentru a afla care este numele funcției care este derivata unei funcții, scriem

**tansig('deriv')**



Aceasta retea neurala poate fi antrenata pentru a aproxima orice functie care are un numar finit de puncte de discontinuitate.

Crearea unei retele feedforward

```
net=newff([-1 2; 0 5],[3,1],{'tansig','purelin'},'traingd');
```

În timpul antrenării, ponderile și *bias*-urile sunt modificate astfel încât să minimizeze funcția de performanță **net.performFcn** (implicit este mse). Există mai mulți algoritmi de antrenare a rețelelor *feedforward*, toți folosind însă gradientul funcției de performanță pentru a calcula ajustările care trebuie făcute pentru a minimiza funcția de performanță.

## Algoritmi de antrenare

### Batch gradient descent

```
net.trainFcn='traingd';
```

Alți parametri asociați învățării: **epochs**, **show**, **goal**, **time**, **min\_grad**, **lr**.

```
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'traingd');
net.trainParam.lr = 0.05;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
```

Rulați programul demonstrativ **nnd12sd1**.

**Batch Gradient Descent with Momentum** - nu se ține cont doar de gradientul local ci și de tendința recentă a suprafeței de eroare. În acest mod se poate evita blocarea într-un minim local al funcției de eroare. Ponderile se modifică cu suma ponderată dintre ultima actualizare și gradient. Parametrul care ponderează suma ia valori între 0 și 1. Dacă **mc** este 0 actualizarea se bazează numai pe gradient. Dacă **mc** este 1 actualizarea se face cu valoarea de la pasul anterior și gradientul este ignorat.

Funcția care implementează acest algoritm se numește **traingdm**.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'traingdm');
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.mc = 0.9;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
```

Rulați programul demonstrativ **nnd12mo**.

## Variable Learning Rate

Dacă eroarea la un moment este mai mare cu **max\_perf\_inc** (de obicei 1.04) decât eroarea la momentul anterior, actualizarea nu are loc iar rata de învățare scade (se înmulțește cu **lr\_dec** = 0.7). Altfel, ponderile se actualizează iar rata de învățare crește (se înmulțește cu **lr\_inc** = 1.05).

Funcția care implementează acest algoritm se numește **traingda** (**traingdx** cu momentum).

```
p = [-1 -1 2 2;0 5 0 5];
```

```

t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'traingda');
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.lr_inc = 1.05;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);

```

Rulați programul demonstrativ **nnd12vl**.

## Resilient Backpropagation

Dacă folosim funcții de transfer sigmoidale, atunci pentru valori mari ale *input*-urilor gradientul are valori foarte mici. Se folosește doar semnul gradientului pentru a da direcția de actualizare a ponderilor. Dacă gradientul are același semn pentru două iterații succesive, valoarea de actualizare se mărește cu factorul **delt\_inc**. Dacă gradientul își schimbă semnul atunci valoarea de actualizare se micșorează cu factorul **delt\_dec**.

Funcția care implementează acest algoritm se numește **trainrp**.

```

p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'trainrp');
net.trainParam.show = 10;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);

```

## Metoda gradientului conjugat

Căutarea se face de-a lungul unor direcții conjugate, ceea ce produce o convergență mai rapidă decât aplicând metoda coborârii pe gradient. Pentru a actualiza ponderile, la fiecare iterație se caută în direcția gradientului conjugat pentru a determina valoarea care minimizează funcția de-a lungul acelei linii.

### 1) Metoda Fletcher-Reeves (traincgf)

La prima iterație algoritmul caută în direcția gradientului

$$\mathbf{p}_0 = -\mathbf{g}_0$$

Se face o căutare liniară de-a lungul direcției curente de căutare pentru a determina pasul de actualizare

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$$

Următoarea direcție de căutare se alege astfel încât să fie conjugată cu direcțiile anterioare de căutare

$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1}$$

unde

$$\beta_k = \frac{\mathbf{g}_k^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}$$

```
p = [-1 -1 2 2;0 5 0 5];  
t = [-1 -1 1 1];  
net=newff(minmax(p),[3,1],{'tansig','purelin'},'traincgf');  
net.trainParam.show = 5;  
net.trainParam.epochs = 300;  
net.trainParam.goal = 1e-5;  
[net,tr]=train(net,p,t);
```

Funcția implicită de căutare liniară este **srchcha** (se poate schimba modificând valoarea parametrului **srchFcn**).

Rulați programul demonstrativ **nnd12cg**.

## 2) Metoda Polak-Ribiere (traincgp)

Diferența față de metoda anterioară constă în modul de calculare al lui  $\beta_k$

$$\beta_k = \frac{\Delta \mathbf{g}_{k-1}^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}$$

```
p = [-1 -1 2 2;0 5 0 5];  
t = [-1 -1 1 1];  
net=newff(minmax(p),[3,1],{'tansig','purelin'},'traincgp');  
net.trainParam.show = 5;  
net.trainParam.epochs = 300;  
net.trainParam.goal = 1e-5;  
[net,tr]=train(net,p,t);
```

## 3) Metoda Powell-Beale (traincgb)

În cazul tuturor metodelor gradientului conjugat, direcția de căutare va fi resetată periodic la valoarea opusă a gradientului. Momentul standard de resetare are loc când numărul iterațiilor este egal cu numărul parametrilor rețelei. Pentru metoda Powell-Beale resetarea se face când proprietatea de ortogonalitate între gradientul curent și cel anterior nu mai este îndeplinită. Acest lucru este testat cu inegalitatea

$$\left| \mathbf{g}_{k-1}^T \mathbf{g}_k \right| \geq 0.2 \|\mathbf{g}_k\|^2$$

Dacă această condiție este satisfăcută, direcția de căutare este resetată la valoarea negativă gradientului.

```
p = [-1 -1 2 2;0 5 0 5];  
t = [-1 -1 1 1];  
net=newff(minmax(p),[3,1],{'tansig','purelin'},'traincgb');  
net.trainParam.show = 5;
```

```
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
```

## Algoritmi quasi-Newton

### 1) Algoritmul BFGS (trainbfg)

Metoda lui Newton este o alternativă la metodele gradientului conjugat pentru optimizare rapidă. Actualizarea parametrilor se face cu regula

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{A}_k^{-1} \mathbf{g}_k$$

unde  $\mathbf{A}_k$  este Hessiana funcției de performanță la momentul  $k$ . Calcularea Hessiane în cazul unei rețele neurale *feedforward* necesită calcule complexe. Metodele quasi-Newton lucrează cu aproximații ale Hessiane.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'trainbfg');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
```

Acest algoritm necesită mai multe calcule și mai multă memorie decât metodele gradientului conjugat. La fiecare pas se memorează aproximarea Hessiane care are dimensiunea numărului de parametri din rețea.

### 2) Algoritmul One Step Secant (trainoss)

Poate fi considerat un compromis între metoda gradientului conjugat și metoda quasi-Newton. Nu se memorează matricea Hessiană complet; la fiecare pas se presupune că aproximarea anterioară este matricea identitate.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
```

```

net=newff(minmax(p),[3,1],{'tansig','purelin'},'trainoss');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);

```

### 3) Algoritmul Levenberg-Marquardt (trainlm)

Când funcția de performanță este suma de termeni pătratici, Hessiana poate fi aproximată astfel

$$\mathbf{H} = \mathbf{J}^T \mathbf{J}$$

și gradientul poate fi calculat

$$\mathbf{g} = \mathbf{J}^T \mathbf{e}$$

unde matricea  $\mathbf{J}$  este *Jacobianul* ce conține derivatele funcției de eroare în raport cu ponderile și *bias*-urile iar  $\mathbf{e}$  este vectorul erorilor. Calcularea matricei  $\mathbf{J}$  este mai puțin complexă decât calcularea Hessienei.

Algoritmul Levenberg-Marquardt folosește următoarea regulă de actualizare a parametrilor

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T \mathbf{J} + \mu \mathbf{I}]^{-1} \mathbf{J}^T \mathbf{e}$$

Dacă  $\mu$  este 0 se obține metoda Newton. Dacă  $\mu$  este foarte mare se obține metoda coborârii pe gradient cu rata de învățare foarte mică. Metoda Newton este rapidă în jurul minimului erorii. La o iterație  $\mu$  este micșorat dacă eroarea scade și este mărit dacă eroarea crește. Astfel, eroarea va fi întotdeauna micșorată la fiecare pas al algoritmului.

Parametrii învățării: **mu**, **mu\_dec**, **mu\_inc**, **mu\_max**.

```

p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'trainlm');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;

```



```
[net,tr]=train(net,p,t);
```

## Aplicații:

### 1. (Problemă de recunoaștere a formelor)

Să se definească o rețea 3-10-10-1 care să recunoască paritatea unui număr de 3 biți. (Se dă un număr de 3 biți. Acesta va fi considerat par dacă numărul biților de 1 este par și eticheta asociată va fi 1 și impar în rest, având ca etichetă -1).

- Să se antreneze rețeaua până când se va obține o eroare (MSE) mai mică decât 0.001.
- Să se reprezinte grafic eroarea (MSE) după fiecare epocă.
- Să se reprezinte grafic eroarea (MSE) la fiecare moment de timp (se poate folosi variabila *cputime*).
- Folosind punctul b. să se compare performanțele algoritmilor prezentați mai sus. Care este cel mai adecvat rezolvării acestei probleme?

### 2. Problema de mai sus pentru aproximarea funcției sinus.

### 3. (Problemă de aproximare a unei funcții sau de regresie neliniară)

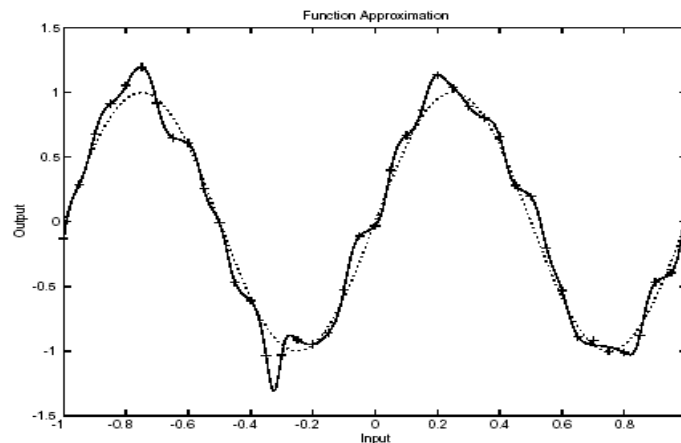
Să se construiască o rețea (21-15-3) și să se antreneze astfel încât să prezică nivelul de colesterol (ldl, hdl and vldl) pe baza a 21 de componente spectrale (măsurători pentru 21 de proprietăți realizate pe 264 de subiecți) (load choles\_all).

- Să se antreneze rețeaua până când se va obține o eroare (MSE) mai mică decât 0.027.
- Să se reprezinte grafic eroarea (MSE) după fiecare epocă.

### 4. Implementați metoda validării încrucișate pentru a estima performanța algoritmilor de mai sus.

## Îmbunătățirea generalizării

O problemă care poate apărea în timpul antrenării unei rețele neurale este “supra-învățarea”. Eroarea obținută pentru datele de antrenare este foarte mică, dar pentru date noi este mare. Rețeaua nu are capacitate de generalizare.



În figura de mai sus, linia continuă reprezintă răspunsul rețelei 1-20-1 antrenată pentru a aproxima o funcție sinus cu zgomot. Datele de antrenare sunt reprezentate cu '+' iar funcția sinus este reprezentată prin linia întreruptă. Se observă că rețeaua a învățat foarte bine mulțimea de antrenare însă nu poate generaliza.

O metodă pentru a rezolva această problemă este să folosim o rețea de dimensiune mai mică pentru o aproximare adecvată. Cu cât rețeaua conține mai mulți perceptroni, cu atât funcția aproximată este mai complexă. Dacă rețeaua este de dimensiune mai mică, nu va avea puterea să supra-învete. Dacă numărul parametrilor dintr-o rețea este cu mult mai mic decât numărul datelor de intrare atunci nu există pericolul supra-învătării.

## Metode de îmbunătățire a generalizării

### 1) Regularizarea

Această tehnică implică modificarea funcției de performanță prin adăugarea unui termen ce constă din media patratelor ponderilor și *bias*-urilor

$$msereg = \gamma mse + (1 - \gamma)msw$$

unde

$$msw = \frac{1}{n} \sum_{j=1}^n w_j^2$$

Folosind această funcție de performanță, rețeaua va avea ponderi mai mici iar răspunsul va avea variații mai mici.

$p = [-1 \ -1 \ 2 \ 2; 0 \ 5 \ 0 \ 5];$

```

t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'trainbfg');
net.performFcn = 'msereg';
net.performParam.ratio = 0.5;           % parametrul gama
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);

```

Partea dificilă este alegerea parametrului  $\gamma$ . O abordare constă în estimarea lui folosind tehnici statistice și este implementată cu algoritmul **trainbr**.

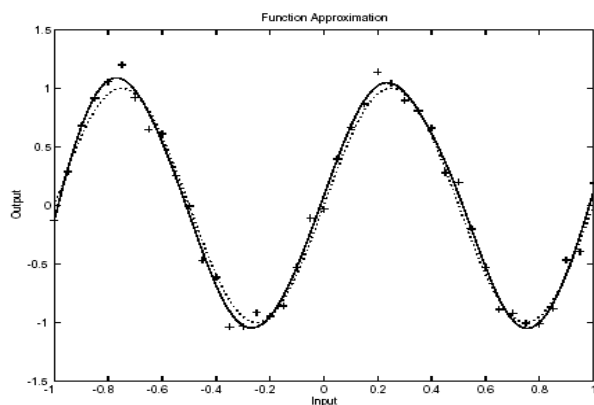
```

p = [-1:.05:1];
t = sin(2*pi*p)+0.1*randn(size(p));
net=newff(minmax(p),[20,1],{'tansig','purelin'},'trainbr');
%net=newff(minmax(p),[20,1],{'tansig','purelin'},'trainbfg');
%net.performFcn = 'msereg';
net.trainParam.show = 10;
net.trainParam.epochs = 50;
net = init(net);
[net,tr]=train(net,p,t);

```

O caracteristică a algoritmului este că afișează numărul de parametri utilizați în învățare.

În figura de mai jos se observă răspunsul rețelei antrenate.



Algoritmul **trainbr** funcționează foarte bine dacă datele de intrare sunt în intervalul  $[-1,1]$  și trebuie lăsat să ruleze până când numărul de parametri folosiți efectiv se stabilizează (converge)

## 2) Early stopping

Datele de antrenare sunt împărțite în 3 submulțimi. Prima submulțime se folosește pentru calcularea gradientului și actualizarea ponderilor și a *bias*-urilor. A doua submulțime se folosește pentru validare. Când rețeaua începe să supra-învețe, eroarea datelor de validare începe să crească. Dacă această eroare crește pentru un număr de iterații, antrenarea se oprește și se returnează ponderile pentru care eroarea de validare era minimă. A treia submulțime (de testare) nu se folosește în timpul antrenării, ci pentru a compara diferite modele.

Această metodă poate fi folosită folosind orice algoritm de antrenare. Trebuie doar specificate datele de validare (care trebuie să fie reprezentative pentru mulțimea de antrenare).

### Aplicație:

Aplicați metoda 2) pentru a aproxima funcția sinus împărțind mulțimea de învățare într-o mulțime de antrenare (1/2), o mulțime de validare (1/4) și o mulțime de testare (1/4).

Indicație: pentru ca împărțirea să fie reprezentativă putem considera următoarele mulțimi:

$[R \ Q] = \text{size}(p);$

$\text{indici\_testare} = 2:4:Q;$

$\text{indici\_validare} = 4:4:Q;$

$\text{indici\_antrenare} = [1:4:Q \ 3:4:Q];$

- Să se antreneze rețeaua cât timp eroarea de validare scade și să se oprească dacă aceasta a crescut pentru un anumit număr de iterații și să se reprezinte grafic cele două erori în același grafic.
- Să se rețină ponderile obținute pentru eroarea de validare minimă.
- Să se compare algoritmi studiați pe mulțimea de test.