Laborator 10

Crearea unei rețele feedforward

Se creează cu funcţia ***newff*** astfel:

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

PR – $[\min_1 \max_1;...\min_n \max_n]$, 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 aflaţi pe nivelul *i*

f_i - funcţia de transfer a nivelului *i*. Implicit este 'tansig' pentru nivelul ascuns şi 'purelin' pentru nivelul de ieşire

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

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

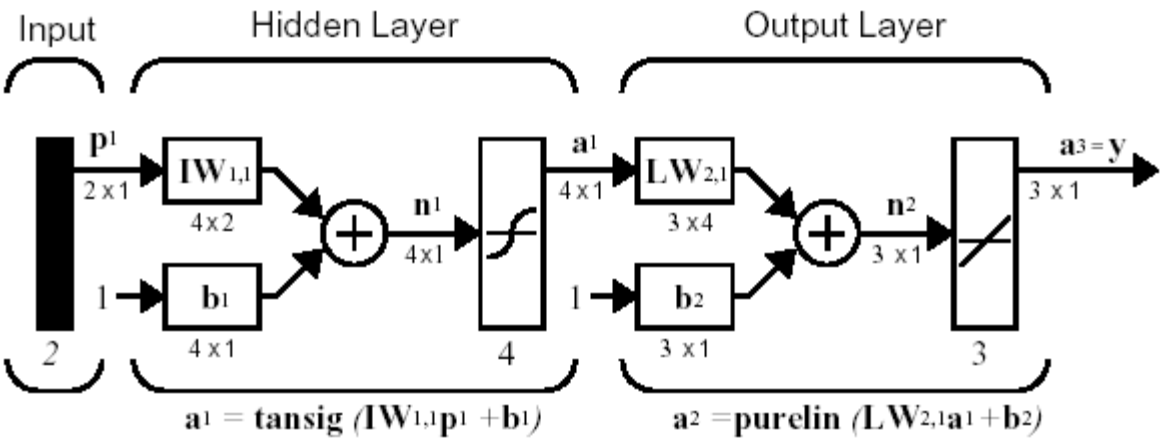
PF - funcţia de performanţă, implicit = 'mse'(media pătratelor erorilor).

Backpropagarea este o metoda de antrenare a reţelelor neurale, obţinută prin generalizarea algoritmului Widrow-Hoff pentru reţele cu mai multe *layere* si cu funcţii de transfer derivabile.

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

Pentru a afla care este numele funcţiei care este derivata unei funcţii, 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 antrenarii, ponderile şi *bias*-urile sunt modificate astfel încât să minimizeze funcţia de performanţă **net.performFcn** (implicit este mse). Există mai mulţi algoritmi de antrenare a reţelelor *feedforward*, toţi folosind însă gradientul funcţiei de performanţă pentru a calcula ajustările care trebuie făcute pentru a minimiza funcţia de performanţă.

Algoritmi de antrenare

Batch gradient descent

net.trainFcn=’traingd’;

Alţi parametri asociaţi învăţării: **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);
```

Rulaţi programul demonstrativ **nnd12sd1**.

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

Funcţia care implementează acest algoritm se numeşte **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);
```

Rulaţi 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 învăţare scade (se înmulţeşte cu **lr_dec** = 0.7). Altfel, ponderile se actualizeaza iar rata de învăţare creşte (se înmulţeşte cu **lr_inc** = 1.05).

Funcţia care implementează acest algoritm se numeşte **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);
```

Rulaţi programul demonstrativ **nnd12vl.**

Resilient Backpropagation

Dacă folosim funcţii de transfer sigmoidale, atunci pentru valori mari ale *input*-urilor gradientul are valori foarte mici. Se foloseşte doar semnul gradientului pentru a da direcţia de actualizare a ponderilor. Dacă gradientul are acelaşi semn pentru două iteraţii succesive, valoarea de actualizare se măreşte cu factorul **delt_inc**. Dacă gradientul îşi schimbă semnul atunci valoarea de actualizare se micşorează cu factorul **delt_dec**.

Funcţia care implementează acest algoritm se numeşte **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 direcţii conjugate, ceea ce produce o convergenţă mai rapidă decât aplicând metoda coborârii pe gradient. Pentru a actualiza ponderile, la fiecare iteraţie se caută în direcţia gradientului conjugat pentru a determina valoarea care minimizează funcţia de-a lungul acelei linii.

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

   La prima iteraţie algoritmul caută în direcţia gradientului

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

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

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

Următoarea direcţie de căutare se alege astfel încât să fie conjugată cu direcţiile 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);
```

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

Rulati programul demonstrativ **nnd12cg**.

2) **Metoda Polak-Ribiere (traincgp)**

Diferenţa faţă de metoda anterioară constă în modul de calculare al lui β_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 Powel-Beale (traincgb)**

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

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

Dacă această condiţie este satisfăcută, direcţia 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 A_k este Hessiana funcţiei de performanţă la momentul *k*. Calcularea Hessianei în cazul unei reţele neurale *feedforward* necesită calcule complexe. Metodele quasi-Newton lucrează cu aproximări ale Hessianei.

```
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 şi mai multă memorie decât metodele gradientului conjugat. La fiecare pas se memorează aproximarea Hessianei care are dimensiunea numărului de parametri din reţea.

2) Algoritmul One Step Secant (trainoss)

Poate fi considerat un compromis între metoda gradientului conjugat şi 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 funcţia de performanţă este suma de termeni pătratici, Hessiana poate fi aproximată astfel

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

şi gradientul poate fi calculat

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

unde matricea **J** este *Jacobianul* ce conţine derivatele funcţiei de eroare în raport cu ponderile şi *bias*-urile iar **e** este vectorul erorilor. Calcularea matricei **J** este mai putin complexă decât calcularea Hessianei.

Algoritmul Levenberg-Marquardt foloseşte 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 μ este 0 se obţine metoda Newton. Dacă μ este foarte mare se obţine metoda coborârii pe gradient cu rata de învăţare foarte mică. Metoda Newton este rapidă în jurul minimului erorii. La o iteraţie μ este micşorat dacă eroarea scade şi este mărit dacă eroarea creşte. Astfel, eroarea va fi întotdeauna micşorata la fiecare pas al algoritmului.

Parametrii invăţării: **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);
```

Aplicaţii:

1. (Problemă de recunoaştere a formelor)

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

   a. Să se antreneze reţeaua până când se va obţine o eroare (MSE) mai mică decât 0.001.
   b. Să se reprezinte grafic eroarea (MSE) după fiecare epocă.
   c. Să se reprezinte grafic eroarea (MSE) la fiecare moment de timp (se poate folosi variabila *cputime*).
   d. Folosind punctul b. să se compare performanţele algoritmilor prezentaţi mai sus. Care este cel mai adecvat rezolvării acestei probleme?

2. Problema de mai sus pentru aproximarea funcţiei sinus.

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

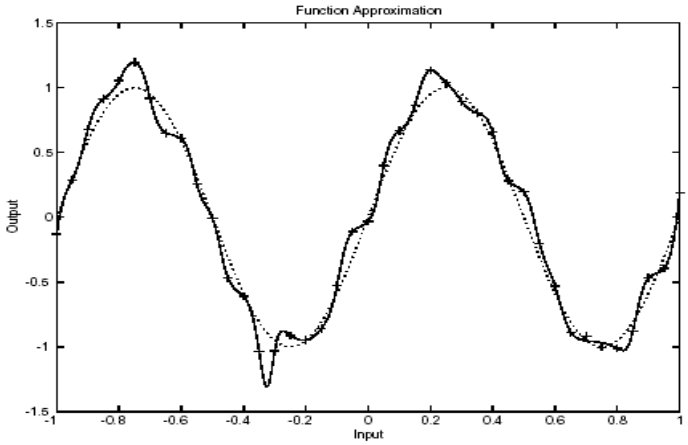
Să se construiască o reţea (21-15-3) şi să se antreneze astfel încât să prezică nivelul de colesterol (ldl, hdl and vldl) pe baza a 21 de componente spectrale (măsurători pentru 21 de proprietăţi realizate pe 264 de subiecţi) (load choles_all).

   a. Să se antreneze reţeaua până când se va obţine o eroare (MSE) mai mică decât 0.027.
   b. Să se reprezinte grafic eroarea (MSE) după fiecare epocă.

4. Implementaţi metoda validării încrucişate pentru a estima performanţa algoritmilor de mai sus.

Îmbunătăţirea generalizării

O problemă care poate apărea în timpul antrenării unei reţele neurale este “supra-învăţarea”. Eroarea obţinută pentru datele de anternare este foarte mică, dar pentru date noi este mare. Reţeaua nu are capacitate de generalizare.



In figura de mai sus, linia continuă reprezintă răspunsul reţelei 1-20-1 antrenată pentru a aproxima o funcţie sinus cu zgomot. Datele de antrenare sunt reprezentate cu ‘+’ iar functia sinus este reprezentată prin linia intreruptă. Se observă că reteaua a învatat foarte bine mulţimea de antrenare însă nu poate generaliza.

O metodă pentru a rezolva această problemă este să folosim o reţea de dimensiune mai mică pentru o aproximare adecvată. Cu cât reţeaua conţine mai mulţi perceptroni, cu atât funcţia aproximată este mai complexă. Dacă reţeaua este de dimensiune mai mică, nu va avea puterea sa supra-înveţe. Dacă numărul parametrilor dintr-o reţea este cu mult mai mic decât numărul datelor de intrare atunci nu există pericolul supra-învătării.

Metode de îmbunătăţire a generalizării

1) Regularizarea

Această tehnică implică modificarea funcţiei de performanţă prin adăugarea unui termen ce constă din media patratelor ponderilor si *bias*-urilor

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

unde

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

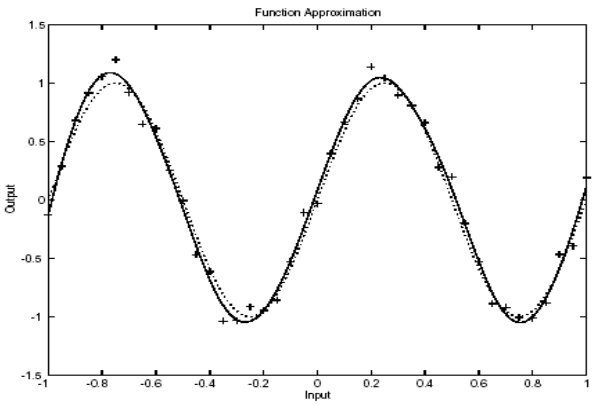
Folosind această funcţie de performanţă, reţeaua va avea ponderi mai mici iar răspunsul va avea variaţii mai mici.

```
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 γ. O abordare constă în estimarea lui folosind tehnici statistice şi 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 afisează numărul de parametri utilizaţi în învăţare.
In figura de mai jos se observă răspunsul reţelei antrenate.



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

2) Early stopping

Datele de antrenare sunt împărţite în 3 submulţimi. Prima submulţime se foloseşte pentru calcularea gradientului şi actualizarea ponderilor şi a *bias*-urilor. A doua submulţime se foloseşte pentru validare. Când reţeaua începe să supra-înveţe, eroarea datelor de validare începe să crească. Dacă această eroare creşte pentru un numar de iteraţii, antrenarea se opreşte şi se returnează ponderile pentru care eroarea de validare era minimă. A treia submulţime (de testare) nu se foloseşte în timpul antrenării, ci pentru a compara diferite modele.

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

Aplicaţie:

Aplicaţi metoda 2) pentru a aproxima funcţia sinus împărţind mulţimea de învăţare într-o mulţime de antrenare (1/2), o mulţime de validare (1/4) şi o mulţime de testare (1/4).
(Indicaţie: pentru ca împărţirea să fie reprezentativă putem considera următoarele mulţimi:
[R Q] = size(p);
indici_testare = 2:4:Q;
indici_validare = 4:4:Q;
indici_antrenare = [1:4:Q 3:4:Q];
)

a. Să se antreneze reţeaua cât timp eroarea de validare scade şi să se oprească dacă aceasta a crescut pentru un anumit număr de iteraţii şi să se reprezinte grafic cele două erori în acelaşi grafic.
b. Să se reţină ponderile obţinute pentru eroarea de validare minimă.
c. Să se compare algoritmii studiaţi pe mulţimea de test.