Laborator 10

Algoritmul clasic de backpropagation (propagare înapoi a erorii) prezentat la curs pentru antrenarea rețelelor neuronale multistrat converge în practică mult prea încet către soluția **w**, vectorul de ponderi ce minimizează (cel mai adesea **w** găsit este minim local) funcția eroare E. Prezentăm în prima parte a laboratorului dezavantajele algoritmului de backpropagation precum și variații ale acestui algoritm care îi cresc viteza de convergență făcându-l aplicabil în practică.

Dezavantaje ale algoritmului de backpropagation

Ilustrăm dezavantajele algoritmului de backpropagation folosind un exemplu de regresie, încercând să aproximăm o funcție reală f. Funcția reală pe care o vom aproxima este aleasă ca fiind dată de o rețea neuronală (netf), cu arhitectura de mai jos (stânga), cei 7 parametri luând valorile: $w^1_{1,1}$ = 10, $w^1_{2,1}$ = 10, b^1_1 = -5, b^1_2 = -5, $w^2_{1,1}$ = 1, $w^2_{2,1}$ = 1, b^2 = -1. Rețeaua astfel instanțiată implementează funcția de mai jos (partea dreapta):

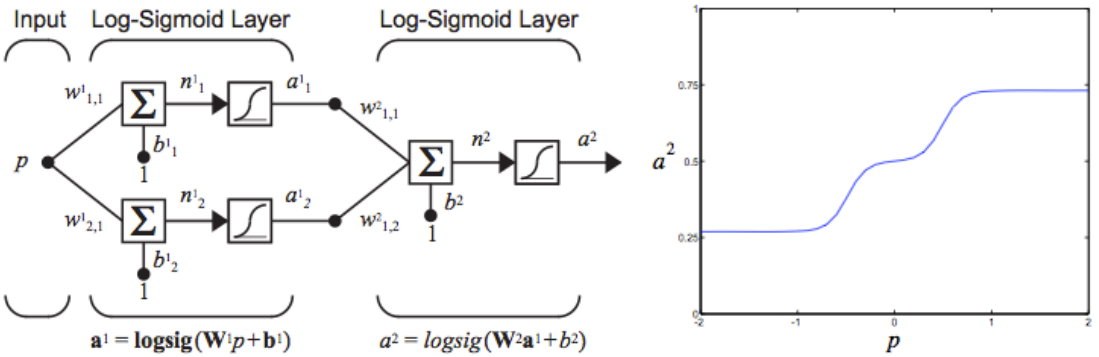


Figura 1: (stânga) Arhitectura rețelei care implementează funcția f; (dreapta) graficul funcției f pentru alegerea parametrilor.

Pentru acest exemplu vom genera câteva puncte (p_i,t_i), i=1,...,41 de pe graficul funcției (41 de puncte roșii, figura din dreapta). Funcție eroare E va fi dată de suma pătratelor erorilor pentru cele 41 de puncte:

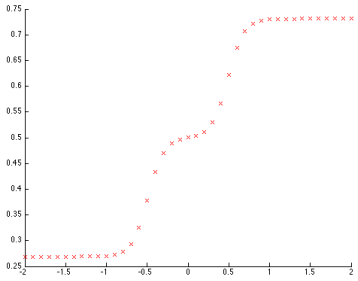


Figura 2

$E(w^1_{1,1}, w^1_{2,1}, b^1_1, b^1_2, w^2_{1,1}, w^2_{2,1}, b^2) = \sum_{i=1}^{41}(t_i - a_i^2)^2$,

unde a_i^2 reprezintă ieșirea rețelei (net) ai cărei parametri vrem să-i învățăm.

Pentru exemplu ales vrem să ilustrează suprafața de eroare E. Suprafața de eroare E nu o putem vizualiza în funcție de toți parametri (sunt 7 parametri, nu puteam vizualiza suprafețe în 8D). Întrucât putem vizualiza suprafața E în maxim 3D, vom considera maxim 2 parametri ca necunoscute, atribuind celorlați 5 parametri anumite valori. Spre exemplu putem vizualiza E în funcție numai de $w_{1,1}^1$, $w_{1,1}^2$ și fixând ceilalți 5 parametri. Alegem valorile pentru cei 5 parametri ca fiind valorile optime (folosite la generarea funcției f prin intermediul rețelei netf) $w_{2,1}^1$ = 10, b_1^1 = -5, b_2^1 = -5, $w_{2,1}^2$ = 1, b^2 = -1). Figura 3 conține vizualizarea suprafeței de eroare E($w_{1,1}^1$, $w_{1,1}^2$), care are punctul de minim global în (10, 1) – punctul albastru.

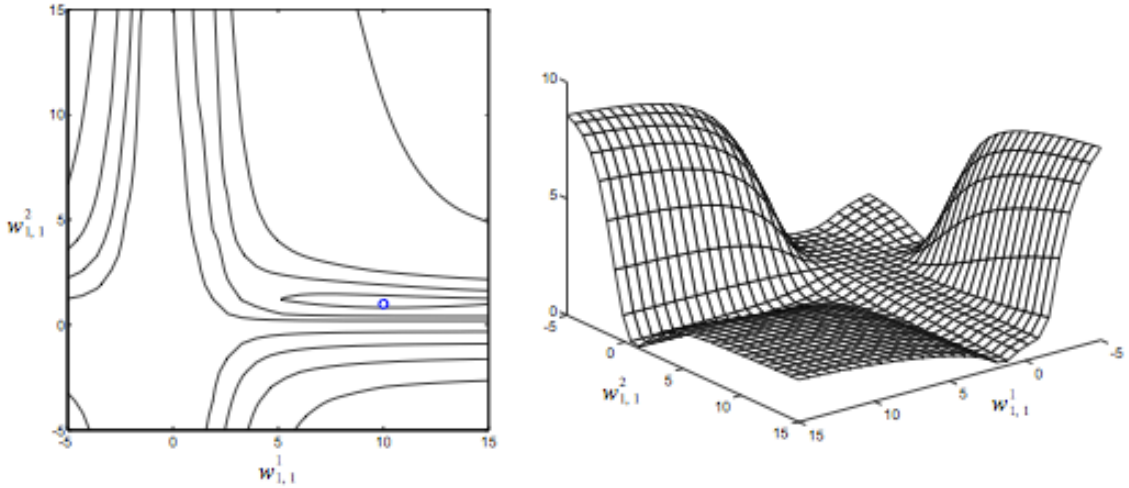


Figura 3: vizualizarea suprafeței de eroare E în funcție de $w_{1,1}^1$, $w_{1,1}^2$

Suprafața de eroare E nu este convexă, variază puternic în funcție de cei doi parametri, având uneori o formă plată (platou), alteori abruptă. Aceste caracteristici fac imposibilă alegerea unei rate de învățare optime pentru algoritmul de coborâre pe gradient:

- în regiunile de platou, ar fi de dorit o rată de învățare mare; regiunile platou sunt o caracteristică pentru funcțiile de activare logsig, ele prezintă saturație foarte repede: pentru valori absolute foarte mari graficul lor este aproape plat, deci gradientul foarte aproape de 0;
- în regiunile foarte abrupte, cu gradient foarte mare, ar fi de dorit o rată de învățare mica pentru a nu avea variații foarte mari ale funcției eroare E

O altă caracteristică a suprafeței de eroare E din figura 3, este că prezintă minime locale. Punctul ($w_{1,1}^1$ = 10 , $w_{1,1}^2$ = 1) este minim local (și global) , de

asemenea punctul $(w^1_{1,1} = 0.88\ , w^2_{1,1} = 38.6)$ este punct de minim local (dar nu și global). În funcție de inițializare, algoritmul de coborâre pe gradient implementat de backpropagation variant batch poate găsi soluții diferite. Figura 4 alăturată ilustrează un astfel de comportament: inițializarea din punctul ***a*** conduce în final la soluția optimă (minimum global) în timp ce inițializarea din punctul ***b*** conduce la soluție de tip minim local.

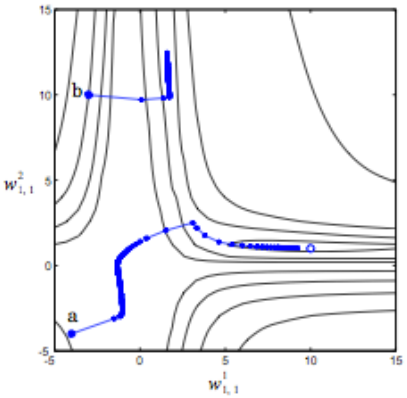


Figura 4

Evoluția erorii E în timp (în funcție de numărul iterației) pentru cele două inițializări este ilustrată în figura 5:

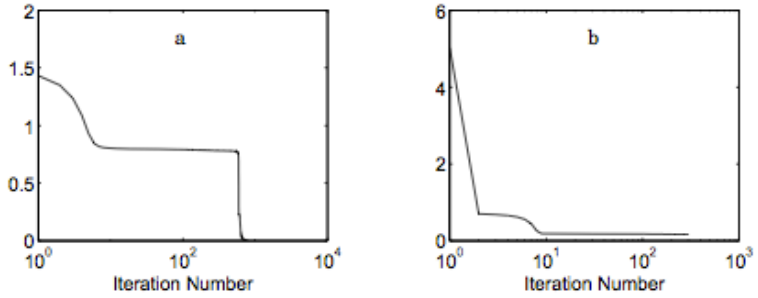


Figura 5

Regiunile în care eroarea E rămâne la aceeași valoare corespund iterațiilor la care soluția traversează regiuni plate. În aceste regiuni am dori să creștem rata de învățare (pentru a accelera procesul de învățare). Totuși, dacă vom crește rata de învățare algoritmul va deveni instabil în regiuni abrupte. Acest efect este ilustrat în figura de mai jos, unde inițializarea algorimului este realizată în punctul ***a*** dar se folosește o rată de învățare mult mai mare.

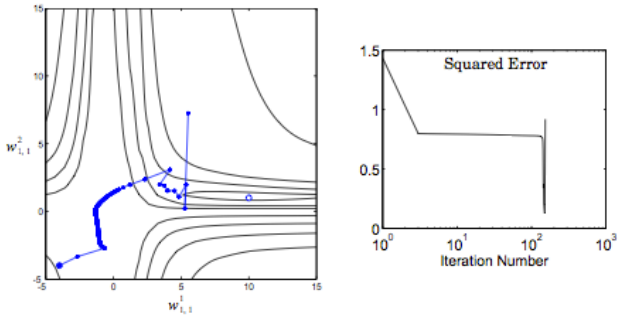


Figura 6

Algoritmul converge rapid la început dar la întâlnirea unei regiuni abrupte (cea care conține punctul de minim global) va diverge. Acest aspect sugerează ca posibilă soluție de îmbunătățire al algoritmului folosirea unei rate variabile de învățare (mare pentru regiunile plate – de gradient mic și mică pentru regiunile abrupte – de gradient mare). O altă posibilitate de îmbunătățire a algorimului de gradient descendent (coborâre pe gradient) este de a elimina oscilațiile în traiectorie prin găsirea unei traiectorii netede, realizabilă prin integrarea treptată a update-urilor gradientului de la fiecare iterație.

Întreg exemplul de mai sus se regăsește în programele demonstrative **nnd12sd1** (permite vizualizarea suprafețelor de eroare) și **nnd12sd2** (permite folosirea unei rate de învățare diferite)**.** Acesta programe permit și vizualizarea suprafeței de eroare E și rulare algoritmului de gradient descendent pentru alți parametri ($w_{1,1}^1$ și b_1^1, respectiv b_1^1 și b_2^1). Rulați aceste programe demonstrativ încercând diverse inițializări.

Implementarea exemplului de mai sus în Matlab este în cele ce urmează:

```
netf = newff([-2 2],[2,1],{'logsig','logsig'});
netf.IW{1,1} = [10 10]';
netf.b{1} = [-5 5]';
netf.LW{2,1} = [1 1];
netf.b{2} = -1;
view(netf);
%plotam functia implementata de retea
p = -2:0.001:2;
t = sim(netf,p);
figure,hold on;
plot(p,t,'b');
%luam numai cateva puncte
p = -2:0.1:2;
t = sim(netf,p);
figure, hold on;
plot(p,t,'xr');
%definim reteaua pentru a fi antrenata
net = newff([-2 2],[2,1],{'logsig','logsig'});
%definim parametri de antrenare
net.trainFcn = 'traingd'; %antrenare cu batch gradient descend -
coborare pe gradient varianta batch

net.trainParam.lr = 0.05;%rata de invatare
net.trainParam.epochs = 1000; %nr de epoci
net.trainParam.goal = 1e-5;%valoarea erorii pe care vrem sa o atingem
[net,tr]=train(net,p,t); %antrenarea retelei
plotperfm(tr);
```

Exercițiul 1: Plotați funcția învățată de rețeaua voastră pe intervalul [-2, 2]. Cum se compară graficul acestei funcții cu graficul funcției inițiale?

Îmbunătățiri ale algoritmului de backpropagation

Algoritmii care îmbunătățesc backpropagation se pot împărți în două categorii: cei bazați pe anumite euristici (rată de învățare variabilă, învățare cu moment) sau tcei bazați pe metode de optimzare numerică (învățare folosind gradientul conjugat, metoda Newton și derivata ei algoritmul Levenberg-Marquardt). Toți acești algoritmi au la bază backpropagation (coborârea pe gradient), folosind derivatele funcției eroare E calculate de la stratul output spre primul strat hidden. Ei diferă numai în felul în care folosesc aceste derivate în updatarea noilor ponderi.

Învățare cu moment

Învățarea cu moment are rolul de a elimina oscilațiile în traiectorie a algoritmului de coborâre pe gradient. Acest lucru se realizează prin a se ține cont nu 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:

$$\Delta w^{k+1} = mc * \Delta w^{k} + (1 - mc) * \nabla E(w^{k})$$

Parametrul care ponderează suma, momentul *mc*, 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. Programul demonstrativ **nnd12mo** exemplifică această tehnică. Figura 7 ilustrează această tehnică (valaoea mc = 0.8), traiectoria nu mai oscilează (are "momentum").

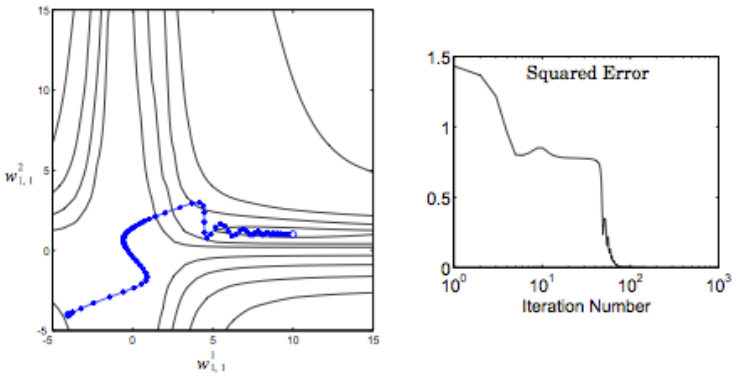


Figura 7

Funcţia care implementează în Matlab acest algoritm este *traingdm*.
Codul Matlab pentru antrenarea rețelei din exemplul inițial pe bază de coborâre pe gradient cu moment este:

```
net = newff([-2 2],[2,1],{'logsig','logsig'});
%definim parametri de antrenare
net.trainFcn = '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);
```

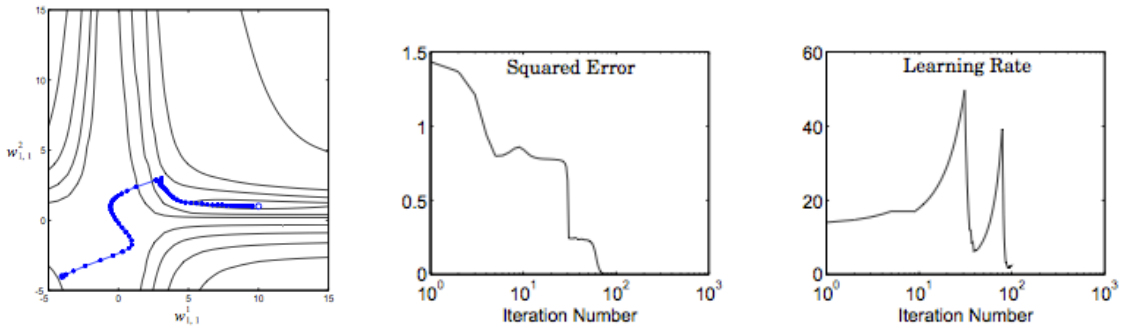
Exercițiul 2: Plotați funcția învățată de rețeaua voastră pe intervalul [-2, 2]. Cum se compară graficul acestei funcții cu graficul funcției inițiale?

Învățare cu rată variabilă

Algoritmii cu rată de învățare variabilă exploatează observația anterioară: folosirea unei rate de învățare cu valoare mare pentru regiunile plate – de gradient mic și a unei rate de învățare cu valoare mică pentru regiunile abrupte – de gradient mare. Acești algoritmi urmează următoarele reguli:

1. Dacă valoarea funcției eroare E crește cu un procent x (de obicei între 1% și 5%, parametru dat de *max_perf_inc* = 1.04 implicit) după realizarea unui update, atunci update-ul nu se mai realizează, rata de învățare curentă se înmulțește cu o valoarea subunitară 0 < lr_dec< 1 (implicit parametrul *lr_dec* = 0.7), iar coeficientul mc (în cazul în care este folosit) care specifică momentul se setează la 0.
2. Dacă valoarea funcției eroare E scade după realizarea unui update, se acceptă acest update iar rata de învățare curentă se înmulțește cu o valoarea supraunitară (parametrul *lr_inc* = 1.05 implicit). Dacă coeficientul mc care specifică momentul este setat la 0, se resetează la valaorea inițială.

Figura 8 de mai jos ilustrează comportamentul unui algoritm de învățare cu rată variabilă și moment.



Programul demonstrativ **nnd12vl** exemplifică această tehnică.

Implementarea în Matlab a acestor algoritmi este dată de funcția *traingda* (antrenare cu rată variabilă fără moment) sau *traingdx* (antrenare cu rată variabilă cu moment).

Codul Matlab pentru antrenarea cu rata variabilă de învățare și moment pentru exemplul inițial este:

```
p = -2:0.1:2;
t = sim(netf,p);
net = newff([-2 2],[2,1],{'logsig','logsig'});

net.trainFcn = 'traingdx';
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.mc = 0.9;
net.trainParam.lr_inc = 1.05;
net.trainParam.lr_dec = 0.7;
net.trainParam.max_perf_inc = 1.04;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
plotperform(tr);
```

Exercițiul 3: Plotați funcția învățată de rețeaua voastră pe intervalul [-2, 2]. Cum se compară graficul acestei funcții cu graficul funcției inițiale?

Alte metode de îmbunătățire (bazate pe tehnici de optimizare):

- metoda gradientului conjugat (în diverse variante: funcțiile *trainscg, traincgb, traincgf, traincgp* în Matlab);
- metoda quasi-Newton (funcția *trainbfg* în Matlab)
- metoda Leverberg-Marquardt (funcția *trainlm* în Matlab)

Exercițiul 4:
Să se definească o rețea 3-10-10-1 care să recunoască paritatea unui număr de 3 biţi. Un număr de 3 biți se consideră par dacă numărul biților de 1 este par (i se va asocial eticheta 0), și impar dacă numărul biților de 1 este impar (i se va asocial eticheta 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. Folosind punctul b să se compare performanţele algoritmilor prezentaţi mai sus.

Care este cel mai adecvat rezolvării acestei probleme?

Exercițiul 5:
Aproximați funcția sinus folosind o rețea neuronală similară cu cea de la Exercițiul 4 dar adaptată problemei de regresie.

Îmbunătățirea capacității de generalizare a unui rețele

O problemă care poate apărea în timpul antrenării unei reţele neurale este *supraînvăţarea* (*overfitting*). Ea constă în următorul fenomen: eroarea obţinută pentru datele de antrenare este foarte mică, dar pentru date noi este mare. În consecință rețeaua nu are capacitate de generalizare.

Figura 9 alăturată ilustrează fenomenul de overfitting. Rețeau neuronală încearcă să aproximeze funcția reprezentată în albastru, pe baza unor perechi de puncte (cercuri albe) care sunt afectate de zgomot (conțin erori). Curba neagră reprezintă funcția învățată de rețea. După cum se poate observa rețeaua oferă răspunsul perfect (cercuri negre care se suprapun peste cele albe) în datele de antrenare (eroare de antrenare zero) dar nu are capacitate de generalizare.

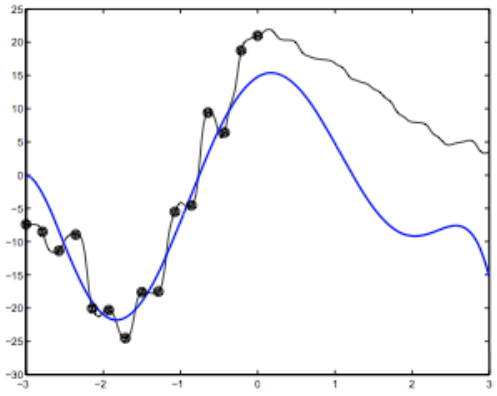
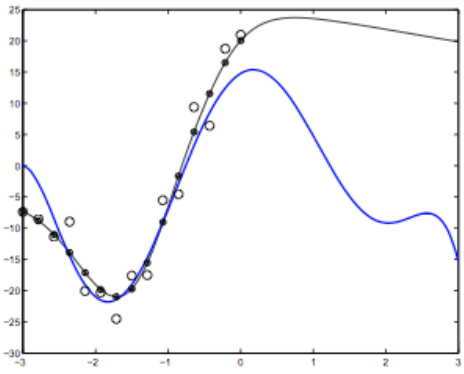


Figura 9

Pe lângă problema de overfitting figura 9 prezintă și un alt tip de eroare: rețeaua nu are capacitatea de a extrapola dincolo de intervalul [-3 0]. Rețelei i se cere ma mult decât i se oferă: datele de antrenare provin numai din intervalul [-3 0], în timp ce testarea se realizează pe întreg intervalul [-3 3].
Figura alăturata 10 prezintă răspunsul unei rețele care elimină fenomenul de overfitting. De această dată rețeaua nu oferă răspunsul perfect



(cercurile negre nu se suprapun peste cele albe) în datele de antrenare dar are capacitate de generalizare pe intervalul [-3 0].

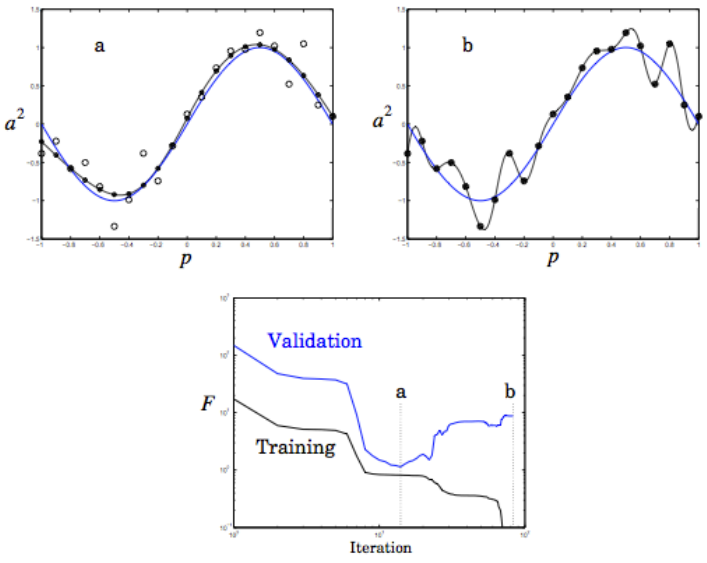
O posibilă metodă de eliminare a problemei de supra-învățare este folosirea unei reţele de dimensiune cât mai mici (număr de perceptroni cât mai mic) pentru o aproximare adecvată. Cu cât reţeaua conţine mai mulţi perceptroni, cu atât funcţia aproximată este mai complexă. Dacă reţeaua este de dimensiune 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.

În cele ce urmează przentăm o metodă de îmbunătățire a capacității de generalizare a unei rețele prin care nu punem accent pe limitarea numărului de perceptroni ci pe valorile învățate. Ea poate fi aplicată pentru orice arhitectură de rețea.

Early stopping

Datele inițiale etichetate (de antrenare) sunt împărţite în 3 submulţimi: antrenare propriu-zisă, validare și testare. 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 pe mulțimea 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.

Figura alăturată prezintă tehnica early stopping. Graficul a prezintă funcția învățată de rețea (culoare neagră) pe baza parametrilor obținuți la punctul de early stopping, punctul a în care eroare pe mulțimea de validare crește, cu toate că eroare pe mulțimea de antrenare scade. Dacă s-ar fi continuat antrenarea, parametric corespunzătorii erorii de antrenare 0 conduc la graficul b care demmonstrează incapacitatea rețelei de a generaliza.



De obicei datele se impart în felul următor: 70% pentru antrenare, 15% pentru validare, 15% pentru testare. Aceste valori sunt și cele implicite pentru funcțiile *dividerand, divideblock, divideint, divideind* care se pot seta pentru o rețea în câmpul *divideFcn.*

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).

Exercițiul 6.
Codul Matlab de mai jos antrenează o rețea pentru aproximarea funcției sin(2*pi*x) folosind o mulțime de antrenare (p_i, t_i) în care targeturile t_i sunt afectate de zgomot. Rețeaua este antrenată folosind criteriul de early stopping.

```
p = [-1:.05:1];
t = sin(2*pi*p)+0.1*randn(size(p));
figure, plot(p,t);
net=newff(minmax(p),[20,1],{'tansig','purelin'},'traincgb');
net.trainParam.show = 10;
net.trainParam.epochs = 50;
net = init(net);
net.divideFcn = 'dividerand';
[net,tr]=train(net,p,t);
```

Plotați funcția implementată de rețea și comparați cu funcția ce se dorește a fi aproximată.