CERCETĂRI PRIVIND CREŞTEREA PERFORMANŢELOR REŢELELOR NEURONALE ÎNTR-UN MEDIU DE CALCUL DISTRIBUIT

Teză destinată obţinerii
titlului ştiinţific de doctor inginer
la
Universitatea "Politehnica" din Timişoara
în domeniul ŞTIINŢA CALCULATOARELOR
de către

Ing. Sorin Babii

Conducător ştiinţific: prof.univ.dr.ing Vladimir Creţu
Referenţi ştiinţifici: prof.univ.dr.ing Mircea Petrescu
prof.univ.dr. Viorel Negru
prof.univ.dr.ing. Ştefan Holban

Ziua susţinerii tezei: 7.12.2007

Seriile Teze de doctorat ale UPT sunt:

1. Automatică
2. Chimie
3. Energetică
4. Ingineria Chimică
5. Inginerie Civilă
6. Inginerie Electrică
7. Inginerie Electronică şi Telecomunicaţii
8. Inginerie Industrială
9. Inginerie Mecanică
10. Ştiinţa Calculatoarelor
11. Ştiinţa şi Ingineria Materialelor

Universitatea „Politehnica" din Timişoara a iniţiat seriile de mai sus în scopul diseminării expertizei, cunoştinţelor şi rezultatelor cercetărilor întreprinse în cadrul şcolii doctorale a universităţii. Seriile conţin, potrivit H.B.Ex.S Nr. 14 / 14.07.2006, tezele de doctorat susţinute în universitate începând cu 1 octombrie 2006.

România, 300159 Timişoara, Bd. Republicii 9,
tel. 0256 403823, fax. 0256 403221
e-mail: editura@edipol.upt.ro

Cuvânt înainte

Teza de doctorat a fost elaborată pe parcursul activităţii mele în cadrul Departamentului de Calculatoare al Universităţii „Politehnica" din Timişoara.

Mulţumiri deosebite se cuvin conducătorului de doctorat prof.dr.ing. Vladimir Creţu, pentru sprijinul constant, ideile şi sugestiile fără de care această teză nu ar fi fost posibilă. De asemenea, doresc să mulţumesc domnului prof.dr.ing. Ştefan Holban, pentru iniţierea în fascinantul domeniu al reţelelor neuronale şi pentru recomandările făcute pe parcursul elaborării tezei. Nu în ultimul rând, doresc să mulţumesc domnilor profesori Mircea Petrescu şi Viorel Negru, pentru răbdarea cu care mi-au lecturat teza şi pentru recomandările făcute.

Doresc să adresez mulţumiri conducerii Departamentului MRC din cadrul Alcatel-Lucent S.A şi echipei IT din acest departament, pentru sprijinul tehnic acordat pe parcursul măsurătorilor experimentale.

Nu în ultimul rând, le mulţumesc tuturor celor care m-au sprijinit pe tot parcursul elaborării tezei, pentru sfaturile şi încurajările lor.

Timişoara, Decembrie 2007 Sorin Babii

Babii, Sorin Ioan

Cercetări privind creşterea performanţelor reţelelor neuronale într-un mediu de calcul distribuit

Teze de doctorat ale UPT, Seria 10, Nr. 4, Editura Politehnica, 2007, 138 pagini, 49 figuri, 19 tabele.

ISSN: 1842-7707

ISBN: 978-973-625-559-5

Cuvinte cheie:
Reţele neuronale, BackPropagation, antrenare, algoritmi paraleli, algoritmi distribuiţi.

Rezumat:
Teza de faţă este rezultatul original al investigaţiilor efectuate în direcţia găsirii unei modalităţi avantajoase de îmbunătăţire a performanţelor în etapa de antrenare a reţelelor neuronale, pe baza resurselor computaţionale existente în cadrul Departamentului Calculatoare. Cercetările au fost axate pe dezvoltarea unor algoritmi paraleli care să evolueze într-o reţea de calculatoare, care să accelereze procesul de învăţare şi să depăşească limitările în puterea de calcul, pentru a permite investigaţii ulterioare în domeniul reţelelor neuronale.

Vreme de sute de milioane de ani, creierele vii, create şi perfecţionate continuu prin eternul proces evoluţionar al selecţiei naturale, au fost singurele dispozitive capabile să efectueze procesarea informaţiei.

Fiinţele umane au inventat calculatorul digital, un dispozitiv artificial de procesare a informaţiei, care a creat perspectiva efectuării de calcule arbitrare înafara sistemelor nervoase biologice. Totuşi, curând, a devenit evident faptul că, sub anumite aspecte, proprietăţile calculatoarelor digitale sunt mult diferite de cele ale creierelor vii.

În primul rând, avem de-a face cu o diferenţă de structură: calculatoarele digitale (de obicei) posedă o singură unitate de procesare, care este, totuşi, destul de puternică. Creierele, dimpotrivă, sunt alcătuite din neuroni intens interconectaţi, lucrând în paralel, fiecare constituind o unitate simplă (comparativ) de procesare. În privinţa capacităţii de procesare a informaţiei, însă, diferenţa structurală nu este cea mai importantă, întrucât toate calculatoarele posedă abilitatea de a simula alte structuri decât cea proprie.

Mai importantă este diferenţa în modul de a câştiga abilitatea de a efectua noi funcţiuni: creierele *învaţă*, în vreme ce calculatoarele trebuie programate. Pentru a efectua o sarcină dată, calculatorul digital necesită *software*, programe care să implementeze în detaliu modul în care acea sarcină poate fi efectuată. Un calculator fără un program nu poate să proceseze informaţie sau să efectueze calcule. De obicei, o fiinţă umană trebuie să înţeleagă o funcţiune dată de procesare a informaţiei şi să dezvolte un algoritm care s-o implementeze, şi abia după aceea calculatorul poate fi programat să efectueze acea funcţiune. Există însă o multitudine de probleme pentru care încă nu există algoritmi formali, sau pentru care este virtual imposibil să se enumere o serie de paşi logici care să determine calculatorul să ajungă la răspunsul corect. Asemenea probleme de obicei presupun observarea unui mare număr de reguli complexe, dependente de context, dintre care majoritatea încă nu sunt cunoscute.Exemple de asemenea probleme sunt cele complexe de recunoaştere a tiparelor, întâlnite în recunoaşterea vorbirii, identificarea scrisului de mână, recunoaşterea fizionomiilor, sau interpretarea spaţială a imaginilor bidimensionale.

Adesea, însă, este posibil să specificăm o problemă destul de precis oferind un set mare de exemple, care să precizeze modul în care obiectele din spaţiul intrărilor trebuie asociate cu obiecte din spaţiul ieşirilor. În mod uzual, oamenii pot învăţa să rezolve asemenea probleme, întrucât creierele animalelor superioare au evoluat către o bună generalizare atunci când li se prezintă un număr de exemple. Acest fapt a condus la dezvoltarea de *Reţele Neuronale Artificiale* [74], [77], dispozitive proiectate să modeleze activitatea reţelelor neuronale biologice la un anumit nivel de detaliu, pentru a atinge (câteva dintre) capacităţile creierului uman[1] .

Ca şi în realitate, o reţea neuronală artificială este un sistem de elemente simple de procesare, masiv paralel interconectate. În această privinţă, reţelele neuronale artificiale se bazează pe cunoştinţele noastre actuale legate de sistemele nervoase biologice. Trebuie remarcat însă faptul că un creier uman este cu câteva ordine de mărime mai complex decât oricare reţea neuronală artificială existentă. Se estimează [79] că în creierul uman există circa 10^{10}–10^{11} neuroni. Fiecare dintre aceşti neuroni recepţionează semnal de intrare de la mii şi chiar zeci de mii de sinapse care îl conectează la alţi neuroni, iar rezultatul activităţii neuronului poate fi transmis prin intermediul altor mii de sinapse către alţi neuroni, influenţându-le astfel activitatea viitoare.

Inteligenţa artificială (AI) este un domeniu de cercetare în care se depun eforturi

[1] Cercetările în materie mai sunt cunoscute sub denumirea de *Conecţionism* (datorită rolului important jucat de conexiunile din reţea) sau Procesare Paralelă Distribuită (PDP) [Rumelhart86], însă denumirea de *reţele neuronale artificiale* pare să aibă o rezonanţă deosebită şi este un termen larg utilizat.

variate în vederea modelării şi/sau re-creării aspectelor inteligenţei naturale. Una dintre cele două paradigme luate în consideraţie de comunitatea AI se bazează pe ideea re-creării comportării inteligente prin imitarea arhitecturii creierelor biologice. Asemenea imitaţii sunt întâlnite de obicei sub denumirea de reţele neuronale artificiale, iar programele utilizate pentru execuţia lor sunt denumite simulatoare de reţele neuronale.

Cuprins

Listă de tabele

1 Introducere

Reţelele neuronale nu sunt programate să îndeplinească o anumită sarcină. Ele trebuie să înveţe sarcina dată prin metoda încercărilor şi a erorilor, în care un supervizor furnizează reţelei neuronale răspunsurile corecte la toate intrările. Regula de antrenare cea mai răspândită este cea denumită *algoritmul backpropagation*.

Totuşi, antrenarea acestor reţele neuronale artificiale este o sarcină foarte costisitoare din punct de vedere computaţional, necesitând milioane de înmulţiri în virgulă flotantă, chiar şi pentru reţele şi probleme de dimensiuni reduse. De asemenea, ele necesită volume mari de memorie. Aceşti doi factori determină ca reţelele neuronale să fie mari consumatoare de timp şi impun limitări efective asupra dimensiunii problemelor care pot fi tratate.

Există mai multe căi de compensare a acestor dezavantaje.

O metodă ar fi reducerea dimensiunii problemei prin *pre-procesarea datelor de intrare*, reducând astfel fie numărul de iteraţii necesare antrenării reţelei, fie chiar dimensiunea reţelei [103]. Trebuie reţinut însă că asemenea soluţii sunt aproape întotdeauna specifice problemei, şi această abordare nu poate fi generalizată pentru acoperirea tuturor tipurilor de probleme.

O altă posibilitate ar fi îmbunătăţirea performanţelor regulii de antrenare cu backpropagation, fie prin modificări *ad hoc*, fie aplicând rezultate din teoria optimizării numerice. Lucrări care tratează aşa-numitele metode ale *gradientului conjugat* [42, 44, 60] aparţin ultimei categorii. În ultimul timp, o serie de eforturi au fost orientate spre investigarea acestei metode şi îmbunătăţirea performanţelor acestora: metode hibride [93, 19, 17, 18], metode de evaluare a erorii de învăţare [21, 38, 40, 55] sau combinarea metodei gradientului conjugat cu metode de pre-procesare [46, 27, 51, 65]. Tot în această categorie mai figurează şi abordări paralele ale metodei gradientului conjugat [11].

O a treia abordare constă în accelerarea algoritmilor existenţi, fie prin implementarea lor direct în hardware (utilizând tehnici VLSI [89], optice [1]) sau prin adaptarea lor în vederea executării pe o arhitectură paralelă. Această ultimă abordare a revitalizat interesul spre acest algoritm de antrenare cu backpropagation [52, 70, 62, 95, 91, 92, 87, 102], interes care scăzuse anterior datorită limitărilor prezentate mai sus. Majoritatea implementărilor paralele din ultimii ani au vizat arhitecturi dedicate [50, 78, 101, 80, 49]. Creşterea puterii de calcul la staţiile de lucru şi a vitezei de interconectare a lor a condus spre implementări distribuite ale algoritmului backpropagation [66, 88, 45].

În cadrul lucrării de faţă s-a ales această ultimă variantă: paralelizarea algoritmului de învăţare backpropagation, utilizând o reţea de calculatoare. După cum se va vedea, este o variantă promiţătoare, mai ales prin costul redus al echipamentului.

Teza de faţă este rezultatul original al investigaţiilor efectuate în direcţia găsirii unei modalităţi avantajoase de îmbunătăţire a performanţelor în etapa de antrenare a reţelelor neuronale, pe baza resurselor computaţionale existente în cadrul Departamentului Calculatoare. Cercetările au fost axate pe dezvoltarea unor algoritmi paraleli care să evolueze într-o reţea de calculatoare, care să accelereze procesul de învăţare şi să depăşească limitările în puterea de calcul, pentru a permite investigaţii ulterioare în domeniul reţelelor neuronale.

Necesitatea unor asemenea algoritmi a provenit dintr-o problemă concretă: antrenarea unei reţele neuronale, pe o singură maşină, dura circa o săptămână, pentru ca după un asemenea interval de timp, în care trebuia ca nimeni să nu se "atingă" de acea maşină, să se descopere că reţeaua neuronală nu prezenta convergenţă, întrucât parametrii de învăţare nu au fost aleşi corect şi trebuia reluat procesul, cu alţi parametri. Utilizând algoritmii prezentaţi în lucrare, s-a reuşit reducerea timpului de antrenare pentru o reţea neuronală, de la circa o săptămână, la câteva ore, ceea ce a permis efectuarea mai multor experimente, pe diferite seturi de parametri de antrenare, până la obţinerea de rezultate satisfăcătoare.

1.1 Obiective

Principalul obiectiv al acestei teze a fost construirea unei baze solide pentru procesarea distribuită a fazei de antrenare a reţelelor neuronale într-o reţea de calculatoare. Ţelul final a fost implementarea a două simulatoare diferite pentru antrenarea reţelelor neuronale şi măsurarea performanţelor lor, comparativ cu varianta secvenţială, atât pentru configuraţii generice de reţele neuronale, cât şi testarea lor pentru o serie de aplicaţii reale de reţele neuronale. Pentru aceasta, pot fi formulate următoarele obiective ştiinţifice:

1. Conceperea unor algoritmi care să permită accelerarea procesului de antrenare a unei reţele neuronale, prin distribuirea acestui proces într-o reţea de calculatoare şi tratarea în paralel a mai multor tipare de antrenare.

   Pornind de la o implementare secvenţială a algoritmului de antrenare cu back-propagation, capitolul 4 îşi propune să rezolve următoarele aspecte:

   - proiectarea unei scheme de colectare a gradienţilor parţiali generaţi de fiecare maşină.
   - evaluarea necesarului de memorie.
   - realizarea unei variante avansate, care să elimine redundanţa care apare prin memorarea întregii reţele neuronale pe fiecare dintre maşinile care participă la rularea algoritmului.
   - evaluarea performanţelor simulatorului bazat pe această variantă avansată, cu investigaţii asupra factorilor care pot afecta evoluţia algoritmului.

2. Realizarea unor algoritmi care să permită procesarea unor reţele de dimensiuni mari, mai mari decât cele permise de memoria existentă într-un calculator, fără pierderi în performanţă. Distribuirea reţelei neuronale se va face prin "secţionarea" acesteia şi plasarea unui număr de neuroni de pe fiecare strat, pe fiecare maşină din reţeaua de calculatoare. Un simulator bazat pe această strategie este prezentat în capitolul 5, pornind de la o serie de cerinţe:

   - simulatorul trebuie să poată fi rulat într-o reţea de calculatoare de dimensiune oarecare, conectate printr-o reţea de tip Ethernet.
   - rescrierea formulelor care intervin în algoritmul backpropagation, în vederea minimizării transmisiei de date între maşini — elementul identificat ca fiind principala cauză pentru care nu se va putea obţine accelerarea ideală.
   - se vor efectua o serie de experimente în vederea evaluării performanţelor simulatorului, cu identificarea factorilor care influenţează accelerarea şi eficienţa care se pot obţine.

3. Formularea unor recomandări în abordarea unor astfel de probleme.

Simulatoarele realizate vor permite investigaţii ulterioare în domeniul antrenării reţelelor neuronale — determinarea parametrilor optimi de învăţare şi a dimensiunii necesare a reţelelor neuronale.

1.2 Limitări

Pe parcursul acestui studiu a fost utilizat algoritmul backpropagation clasic de antrenare a reţelelor neuronale feed-forward. Nu au fost luate în consideraţie tehnici de preprocesare şi alte modalităţi elaborate în vederea accelerării procesului *secvenţial* de antrenare.

Pentru măsurarea performanţelor simulatoarelor concepute s-au utilizat reţele neuronale generice, fără aplicabilitate practică, de dimensiune convenabilă, iar seturile de tipare utilizate au fost generate aleator. Nu s-au urmărit performanţele de *învăţare*

a reţelelor neuronale – care de obicei se măsoară în număr de iteraţii necesare atingerii unui anumit grad de calitate a învăţării – ci s-a dorit să se compare evoluţia simulatoarelor distribuite realizate cu cea a unui simulator secvenţial, pentru aceeaşi problemă de reţele neurnonale.

1.3 Conţinutul tezei

Rezultatele obţinute sunt sintetizate în teza de faţă, pe parcursul a trei secţiuni.

Prima secţiune, introductivă, prezintă, pe parcursul capitolului 2, o trecere în revistă a conceptelor fundamentale privitoare la reţelele neuronale, de la modelul biologic la modelele conexioniste, respectiv reţelele cu backpropagation.

Cea de a doua secţiune prezintă două strategii de paralelizare şi algoritmii dezvoltaţi în cadrul cercetărilor întreprinse. Au fost identificate două modalităţi de distribuire a activităţii (capitolul 3):

- partiţionarea datelor – algoritmul clasic de antrenare se execută în paralel pe mai multe calculatoare, pe seturi diferite de învăţare, după care datele sunt colectate şi combinate.
- partiţionarea reţelei – reţeaua neuronală este distribuită şi evoluează pe mai multe calculatoare, comunicarea între neuroni realizându-se prin intermediul reţelei în care sunt interconectate calculatoarele.

Tot în capitolul 3 sunt prezentate elementele de analiză a acestor algoritmi paraleli, sunt enumerate câteva dintre cauzele care ar putea duce la o eficienţă scăzută a paralelizării, precum şi propuneri de experimente şi măsurători de realizat.

Capitolul 4 prezintă în detaliu algoritmul de partiţionare a datelor şi simulatorul DataParSim. Mai multe maşini execută în paralel şi independent acelaşi algoritm de antrenare, însă pe seturi de date diferite. O maşină "administrator" se îngrijeşte de distribuirea iniţială a configuraţiei reţelei neuronale şi a seturilor de date de antrenare, iar mai târziu colectează rezultatele şi le combină într-o nouă configuraţie a ponderilor din reţeaua neuronală, reluând procesul până la atingerea performanţelor prescrise în privinţa calităţii învăţării. Sunt prezentate şi analizate rezultatele unor experimente în vederea evaluării performanţelor.

Algoritmul pentru varianta cu partiţionarea reţelei şi simulatorul NetParSim sunt prezentate în capitolul 5. Reţeaua neuronală este divizată astfel încât fiecare dintre calculatoarele utilizate simulează o porţiune din reţea. În acest caz, antrenarea se face pe un singur set de date, iar evoluţia maşinilor implicate este sincronă.

Spre deosebire de varianta cu partiţionarea datelor, când singura diferenţă faţă de varianta secvenţială, clasică, este timpul necesar antrenării, partiţionarea reţelei permite procesarea unor reţele de dimensiuni mult mai mari, care altfel ar fi imposibil de simulat pe un singur calculator, datorită limitărilor de memorie. Rezultatele experimentelor realizate sunt prezentate şi analizate în finalul acestui capitol.

Secţiunea a treia este dedicată prezentării şi analizei rezultatelor experimentale pentru o serie de aplicaţii *clasice* de reţele neuronale. Testele efectuate au vizat atât studierea performanţelor algoritmilor prezentaţi, în comparaţie cu algoritmul secvenţial clasic (efectuat pe acelaşi tip de maşină), cât şi studierea limitărilor impuse ca urmare a paralelizării asupra performanţelor de invăţare. Este vorba despre studierea atât a aspectelor cantitative cât şi a celor calitative.

În capitolul 7 sunt prezentate concluziile finale şi perspectivele de cercetare în continuare.

Testele efectuate au fost realizate pe trei categorii de maşini: o reţea cu până la 12 calculatoare IBM RS6000, într-o reţea ethernet 10 BaseT, respectiv două reţele de până la 16 staţii SunBlade 150 şi de PC-uri, într-o reţea ethernet 100 BaseT. Desigur, atât algoritmul cu partiţionarea datelor cât şi cel cu partiţionarea reţelei, au fost comparaţi cu algoritmul secvenţial executat pe acelaşi tip de maşină.

Capitolul 6 prezintă rezultatele experimentale pentru simulatoare de reţele neuronale "clasice", utilizate de obicei pentru testarea performanţelor: NETtalk, recunoaştere de caractere (fonturi X11), compresie de imagine, recunoaştere de imagini (OCR), recunoaşterea ţintelor de sonar.

Teza se încheie cu o serie de concluzii desprinse pe parcursul tratării subiectelor propuse în lucrare, rezumatul contribuţiilor, şi perspectivele de continuare a cercetărilor în domeniul paralelizării fazei de antrenare a reţelelor neuronale.

Se menţionează faptul că testele au fost efectuate pe reţeaua de staţii IBM RS6000 din cadrul Laboratorului de Grafică al Departamentului de Ingineria Calculatoarelor şi a Programării, Universitatea "Politehnica" din Timişoara, respectiv pe o reţea de staţii SunBlade 150 şi pe o reţea de PC-uri Hp-Compaq, cu amabilitate puse la dispoziţie de Mobile Networks Division din cadrul Alcatel-Lucent SA, România.

2.1 Celula nervoasă

Celula nervoasă — sau *neuronul* — este elementul fundamental al sistemului nervos. Schematic, în figura 2.1 este redat simplist un lanţ de neuroni, având menţionate componentele semnificative din punctul de vedere al utilităţii în modelele matematice ulterioare [31]:

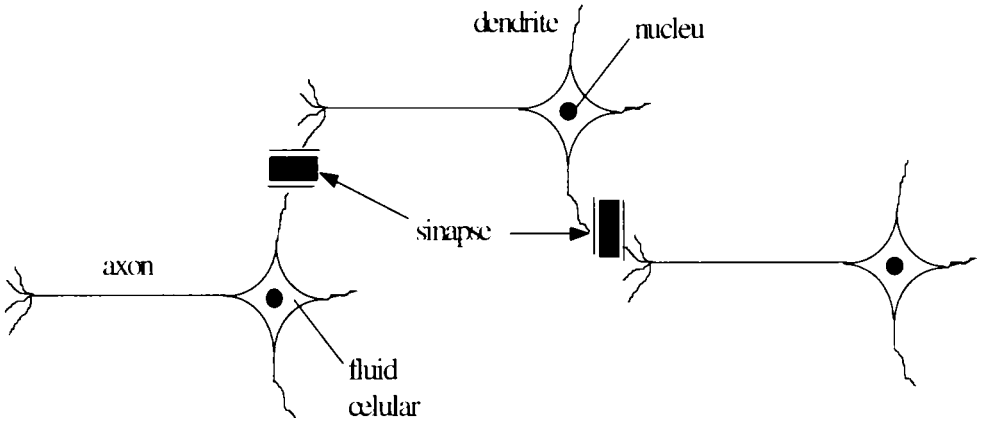


Figura 2.1: *Conexiune neuronală*

Celula nervoasă are diametrul de ordinul micronilor şi este alcătuită din:

- *corpul celular* — conţine nucleul neuronului şi guvernează transfomările biochimice de sinteză a enzimelor necesare funcţionalităţii celulei nervoase;
- *dendritele* — sunt extensii filiforme cu ramificaţii arborescente ale corpului celular, având diametrul de ordinul zecimilor de microni şi lungimi de până la 10 microni. Dendritele sunt principalii receptori de semnal ai neuronului.
- *axonul* — este o prelungire fibroasă, de lungime considerabilă în comparaţie cu restul celulei nervoase, variind între 0,2-1 centimetru pentru neuronii din cortex şi 10 centimetri pentru neuronii periferici. Axonul reprezintă conexiunea de ieşire a semnalului procesat de către neuron.

Modalitatea de interconectare spaţială a celulelor nervoase este realizată prin intermediul unor zone cu conductanţă electrică îmbunătăţită, denumite sinapse, şi care constituie practic o legatură între axonul unui neuron şi o dendrită a unui alt neuron, cu rol de comunicare între cele două celule.

Măsurătorile au arătat că eficienţa energetică a creierului uman este de aproximativ 10^{-16} J/(operaţie·s) [23], în timp ce cele mai performante calculatoare au o eficienţă de 10^{-6} J/(operaţie·s).

Neuronii au caracteristici comune cu cele ale celulelor vii în general, însă ca organizare şi sistem biochimic prezintă individualităţi ce permit specializarea lor în realizarea unor funcţiuni specifice ca:

- primesc semnale de la neuronii vecini,
- conduc semnalul nervos,
- integrează semnalele primite,

- amplifică sau mediază semnalul nervos rezultant,
- transmit semnalul nervos pe canale preferenţiale spre neuronii receptori.

2.2 Reţele neuronale

2.2.1 Modele neuronale artificiale

În forma cea mai generală - şi în accepţia iniţială - o reţea neuronală este un algoritm proiectat să modeleze modul în care creierul realizează o anumită funcţiune, estimându-se empiric, cu aproximaţii acceptabile, modalitatea de procesare a informaţiei de către cortex. Acest scop este valabil doar pentru cazul primelor modele. Ulterior modelele neuronale au părăsit ideea iniţială de simulare a propagării şi procesării semnalului nervos, păstrând însă specificul nomenclatorului biologic pentru desemnarea unor etape. Astfel, aplicaţiile tratate în continuare se vor referi la reţele neuronale care îşi realizează scopul propus printr-un proces iniţial de *învăţare*. Pentru o bună performanţă aplicativă, reţeaua îşi creează ponderi de interconectare între unităţile de intrare şi cele de ieşire, unităţi care vor fi denumite *neuroni* sau *unităţi de procesare*. O definiţie simplistă a noţiunii actuale de reţea neuronală este dată de Aleksander şi Morton, în [3]:

> O reţea neuronală este un procesor cu distribuţie paralelă care are o mare capacitate de a stoca cunoştinţe dobândite din experimente, în scopul utilizării active ale acestora.

Simularea acţiunii cerebrale este elocventă în cele două acţiuni de bază:

- cunoştinţele sunt acumulate în timpul unui proces de învăţare,
- stocarea cunoştinţelor se realizează prin valoarea conexiunilor interneuronale, denumite şi ponderi sinaptice.

2.2.2 Scurt istoric

1943 Warren McCulloch, un neurofiziolog, şi Walter Pitts, un tânăr matematician publică un articol [53] despre cum s-ar putea să lucreze neuronii. Au modelat o reţea neuronală simplă prin intermediul unor circuite electrice.

1949 Donald Hebb publică o carte [32], *Organization of Behavior*, în care reiterează conceptul de neuroni şi modul lor de lucru. El enunţă o regulă care afirmă faptul că traseele neuronale se accentuează de fiecare dată când acestea sunt utilizate.

1950– Primele modelări (rudimentare), efectuate de Nathanial Rochester în laboratoarele IBM [71]. Prima încercare a fost nereuşită, însă următoarele au fost încununate de succes. Era perioada înfloririi metodelor tradiţionale de calcul şi cercetările în domeniul neuronal au trecut în planul secund.

1956 *Dartmouth Summer Research Project on Artificial Intelligence* impulsionează cercetările atât în domeniul inteligenţei artificiale cât şi în cel al reţelelor neuronale. în anii care au urmat, John von Neumann sugerează imitarea funcţiunilor simple ale neuronilor prin intermediul releelor telegrafice şi al tuburilor electronice. De asemenea, Frank Rosenblatt la Universitatea Cornell, începe studiul Perceptronului. Acesta, implementat hardware, este cea mai veche reţea neuronală încă utilizată în zilele noastre. Din păcate, limitările perceptronului au fost evidenţiate destul de curând, ceea ce a dus la o perioadă de stagnare. Abia în 1969, Marvin Minsky şi Seymour Papert demonstrează matematic ineficienţa lui şi oferă şi rezolvarea [58].

1959 Bernard Widrow şi Marcian Hoff de la Stanford dezvoltă modelele denumite de ei ADALINE şi MADALINE — primele reţele neuronale aplicate în probleme reale, eliminarea ecoului pe liniile telefonice, care se utilizează şi în zilele noastre.

A urmat o perioadă de stagnare, datorată şi lipsei de finanţare, dar şi exagerărilor în aprecierea potenţialului reţelelor neuronale, care au dus chiar la teama de asemenea "maşini gânditoare". Scrierile lui Isaac Asimov reflectă aceste temeri prin imaginarea unor lumi în care roboţii preiau toate sarcinile oamenilor. Alţi scriitori imaginează scenarii şi mai sinistre, un singur exemplu fiind HAL din *2001: A Space Oddissey*.

1982 John Hopfield de la Caltech prezintă o lucrare la Academia Naţională de Ştiinţe [34]. Abordarea lui este nu doar simpla modelare a creierului, ci crearea de dispozitive utile. El demonstrează clar, analitic, cum ar putea lucra asemenea reţele şi ce s–ar putea face cu ele.

1982 La Kyoto, în Japonia, are loc *US-Japan Joint Conference on Cooperative/Competitive Neural Networks*. Ca urmare, Japonia anunţă efortul către Generaţia a Cincea, generând o îngrijorare în SUA că americanii ar putea rămâne în urmă. Curând, finanţarea nu a mai fost o problemă.

1985 American Institute of Physics organizează ceea ce va deveni o întâlnire anuală — *Neural Networks for Computing*.

1987 Institute of Electrical and Electronic Engineer's (IEEE) organizează prima *International Conference on Neural Networks*, la care iau parte mai bine de 1800 de participanţi.

1989 La conferinţa *Neural Networks for Defense*, Bernard Widrow afirmă că deja a început al IV-lea Război Mondial, în care câmpul de bătălie sunt producţia şi comerţul.

1990 *US Department of Defense Small Business Innovation Research Program* nominalizează 16 teme care vizau explicit reţelele neuronale, iar alte 13 menţionează posibilitatea utilizării reţelelor neuronale.

2.2.3 Modelul McCulloch & Pitts

Înainte de descoperirea modalităţii de funcţionare a sistemului nervos celular [33], pentru care Hodgkin, Huxley şi Eccles au primit Premiul Nobel în 1963, în 1943, McCulloch şi Pitts au elaborat un model simplist al funcţionării unui neuron, pornind de la funcţionalitatea acestuia [53]. Ei au presupus că neuronul se comportă în modul următor: din sinapse, prin dendrite, acesta acumulează o sarcină electrică ce provine de la axonii neuronilor vecini; în momentul în care valoarea sarcinii depăşeşte o anumită valoare T — numită *prag* — neuronul devine activ, transmiţând prin axonul său un curent constant U. Această funcţionare poate fi descrisă matematic prin:

$$o = \begin{cases} U & \text{pentru } \sum_{i=1}^{n} x_i \geq T \\ 0 & \text{pentru } \sum_{i=1}^{n} x_i < T \end{cases} \tag{2.1}$$

unde x_i = intrarea i;
o = ieşire.

Schematic, modelul este redat în figura 2.2:

O astfel de unitate este denumită *unitate de procesare*. Considerând valoarea pragului $T = 1$ şi valoarea tensiunii de ieşire $U = 1$, se pot construi, cu aceste unităţi de procesare unitare, circuite logice de tipul AND, NAND, OR, şi respectiv celule de memorie, în modul sugestiv prezentat în figura 2.3.

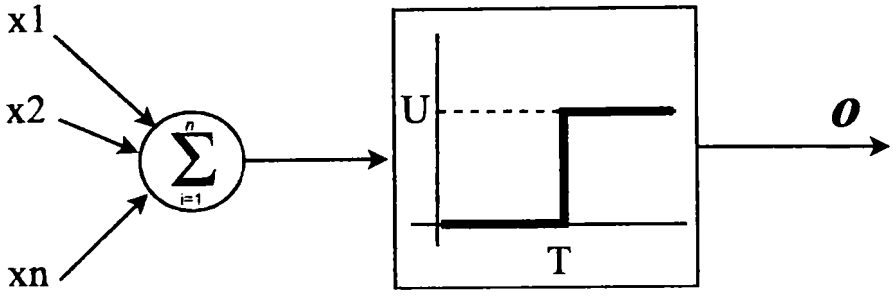


Figura 2.2: *Schema modelului matematic McCulloch & Pitts*

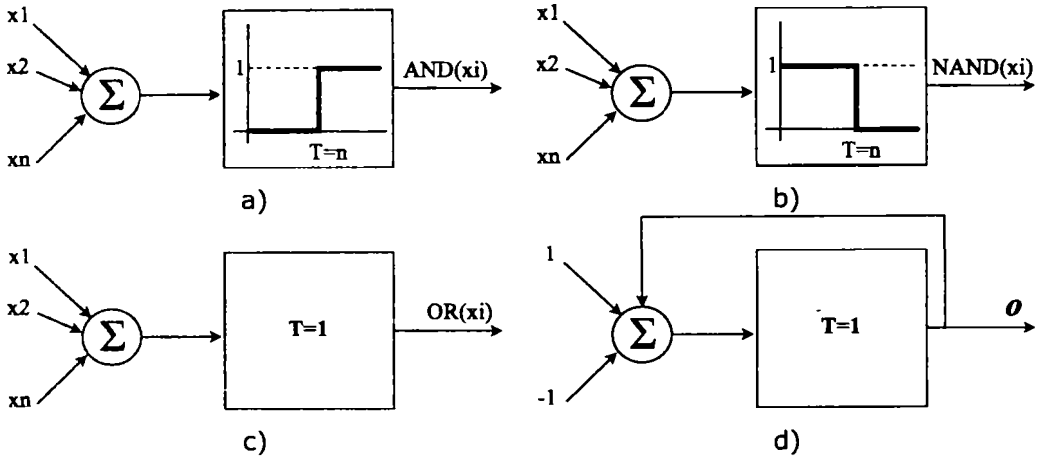


Figura 2.3: *Circuite logice realizate cu unităţi de procesare McCulloch & Pitts*

Cei doi matematicieni au sugerat posibilitatea ca şi creierul uman să fie o aglomerare de asemenea circuite logice, care ar procesa informaţia într-un mod similar cu calculatoarele digitale, al căror model era definitivat în aceeaşi perioadă.

Această prezumţie s-a dovedit a fi greşită, însă are valoare intuitivă, deoarece s-a dovedit ulterior că neuronii funcţionează oarecum după modelul activării la atingerea unui prag.

Acest model neuronal este caracterizat prin formalism şi printr-o definire matematică elegantă, având un potenţial de calcul considerabil. Simplitatea sa presupune:

- stări binare de tip 0 şi 1,
- asumarea operării în timp discret,
- sincronicitatea operării tuturor neuronilor.

2.2.4 Modelul Hebb

În 1949, Daniel Hebb, un biolog care se ocupa cu studiul comportamentului primatelor, presupune că prin modificarea sinapselor conectivitatea celulelor nervoase este într-o continuă schimbare, pe masură ce organismul îşi dezvoltă diferite aptitudini. Având la

bază prezumţiile lansate de laureatul Nobel Ramon y Cajal încă din 1899, (republicate în 1995, în *Histology of the nervous system* [10], Hebb a enunţat o *regulă a învăţării* [32], conform căreia

> Valoarea efectivă a variabilei de conexiune între doi neuroni creşte prin activarea repetată a conexiunii.

Cartea a avut un răsunet imens printre psihologi, dar a fost neglijată de ingineri, fapt ce a dus la ignorarea acestei reguli de învăţare în domeniul reţelelor neuronale. Abia în 1956 a fost utilizată această regulă de învăţare, evidenţiindu-se fenomenul de *inhibiţie* a conexiunii neuronale [71].

Conform regulii de învăţare a lui Hebb, sinapsele care conectează neuronii nu transmit axonului neuronului receptor decât o parte a sarcinii electrice primite de la axonul neuronului transmiţător. Reprezentarea matematică a acestei reguli se obţine utilizând un coeficient subunitar de transmisie, numit *pondere* şi notat cu w.

Ponderile sunt asociate neuronului receptor şi îşi pot modifica valorile în timp conform regulii:

- dacă neuronul transmiţător şi neuronul receptor sunt ambii activi în acelaşi interval de timp, valoarea ponderii asociate conexiunii creşte;
- dacă fenomenul de activare concomitentă se repetă de mai multe ori sau dacă intervalul de timp al dublei activări este mare, valoarea ponderii asociate va creşte considerabil;
- dacă neuronul transmiţător şi neuronul receptor nu sunt simultan activi într-o perioadă de timp mare, valoarea ponderii asociate conexiunii scade.

Ulterior, s-a dovedit că într-adevăr, această regulă este valabilă, şi stă la baza mecanismului de învăţare, respectiv uitare, în creierele naturale.

Plecând de la această regulă, se poate descrie o unitate de procesare, cu ponderi asociate, care este modelul matematic utilizat şi în prezent, cu variaţiuni nesemnificative. Astfel, considerând o unitate cu indicele i, cu n intrări de indice j, fiecare având asociată o pondere w_j, funcţionarea acesteia se poate descrie prin:

$$o_i = \begin{cases} 1 & \text{pentru } \sum_{j=1}^{n} x_j \cdot w_j \geq T \\ 0 & \text{pentru } \sum_{j=1}^{n} x_j \cdot w_j < T \end{cases} \tag{2.2}$$

sau prin schema din figura 2.4.

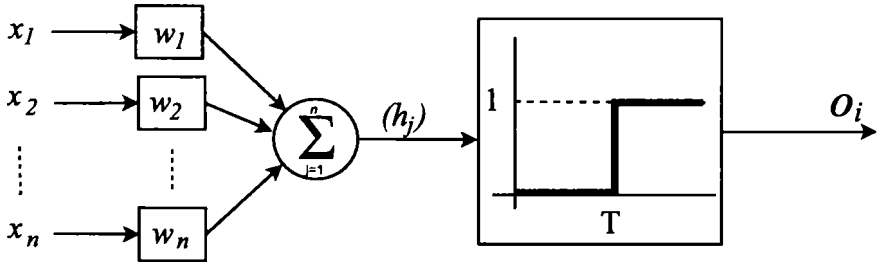


Figura 2.4: *Schema modelului matematic Hebb*

Pentru simplificare, se va nota suma ponderată care reprezintă intrarea totală în unitatea de procesare cu h_i:

$$h_i = \sum_{j=1}^{n} w_j \cdot x_j \tag{2.3}$$

O asemenea unitate poate fi privită ca un procesor de informaţie foarte primitiv. Dacă se conectează un număr de astfel de procesoare între ele, se obţine o *reţea neuronală artificială*, sau, mai corect spus, o *structură de calcul conexionistă*[1].

Modul de conectare a acestor procesoare reprezintă *topologia reţelei* şi determină modul de calcul al ponderilor, şi, implicit, funcţionalitatea reţelei.

Calculul ponderilor este denumit *învăţare* sau *antrenare*, iar metodele iterative de calcul sunt cunoscute sub denumirea de *algoritmi de învăţare* sau de *antrenare*.

Ashby, în cartea sa *Design for a brain*, [5], destinată comportamentului adaptiv, presupune şi argumentează că un sistem natural nu se naşte învăţat, ci ajunge la anumite performanţe prin corelarea variabilelor ce descriu evenimentele percepute sistemocentric. Cartea tratează aspectele dinamice ale organismului viu ca sistem abstract, mecanic, şi evaluează conceptul de stabilitate.

În 1954, teza de doctorat a lui Minsky [56], se constituie ca prolog al lucrării *Steps toward artificial intelligence* [57], ce constituie şi astăzi unul din capitolele fundamentale în domeniul reţelelor neuronale.

Steinbuch, în *Die lernmatrix* [84] continuă ideea lansată de Taylor în *Electrical simulation of some nervous system functional activities* [90] despre o *memorie asociativă* realizabilă prin aşa-numitele *matrici de învăţare*, care ar fi interpuse între câmpuri de senzori receptori şi efectori motori.

Contribuţii semnificative la teoria memoriei asociative au adus ulterior şi Anderson în *A simple neural network generating an iteractive memory* [4], Kohonen în *Correlation matrix memories* [47] şi Nakano în *Associatron – a model of associative memory* [61], în acelaşi an (1972); independent, au introdus ideea unei matrici corelative de memorie ("correlation matrix memory"), care utilizează o regulă de învăţare bazată pe produsul de ieşire al algoritmului.

O figură particulară în contextul evoluţiei conceptului reţelelor neuronale a fost von Neumann, care a utilizat ideea de redundanţă pentru proiectarea unei reţele neuronale privită ca un ansamblu unitar cu un comportament cert, alcătuit din neuroni care pot fi priviţi separat ca şi componente cu comportament incert. Această idee a fost dezvoltată de Winograd şi Cowan, care au utilizat o reprezentare redundant distribuită a reţelelor neuronale [99].

2.3 Modelul Rosenblatt — perceptronul

După 15 ani de la enunţarea modelului McCulloch & Pitts, devenit clasic, un prim pas spre rezolvarea problemei recunoaşterii tiparelor îl face Rosenblatt, prin lucrarea "*The perceptron: A probabilistic model for information storage in the brain*" [72], în care enunţă teorema convergenţei perceptronului, şi prin lucrările adiacente, în care dezvoltă această teorie [73], [75], [74].

Topologia cea mai simplă este asociată unei structuri conexioniste, denumită *perceptron*, a cărui structură schematică este prezentată în figura 2.5.

În forma cea mai simplă, el este compus dintr-un vector de intrare n-dimensional, x, o unitate de procesare şi o singură ieşire. Menirea unei astfel de unităţi este de a învăţa o transformare de tipul $T : \{-1, 1\} \rightarrow \{-1, 1\}$ utilizând seturi de tipare cu intrări de tip x şi ieşirile corespunzătoare $o = T(x)$.

Unitatea de ieşire a unui preceptron este un element liniar cu prag, care calculează ieşirea o conform relaţiei:

$$o = F\left(\sum_{i=1}^{n} w_i \cdot x_i + \theta\right) \tag{2.4}$$

[1] Expresia *structură de calcul conexionistă* reprezintă de fapt noţiunea descriptivă la care s-a ajuns prin încercarea de a exprima matematic teoriile de moment ce sugerează posibilităţi de modelare a propagării semnalului prin sistemul neural uman. Este practic o expresie mult mai adecvată atribuită algoritmilor astfel obtinuţi, care nu îşi menţin scopul iniţial, ci îl adoptă pe cel al aplicaţiilor pentru care au fost dedicaţi.

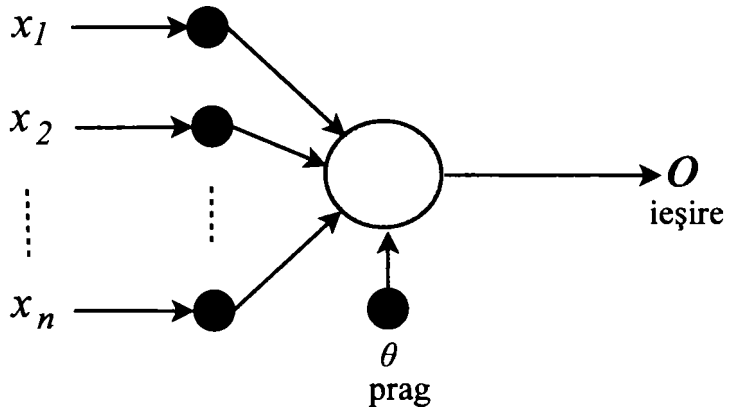


Figura 2.5: *Structura schematică a perceptronului*

în care funcţia F este funcţia Heaviside sau *signum*, iar θ poate fi orice funcţie sau constantă care ajustează conexiunile astfel încât să se obţină ieşirea dorită. În acest caz, funcţia F este:

$$F(i) = \begin{cases} 1 & \text{pentru } i > 0 \\ -1 & \text{altfel} \end{cases} \tag{2.5}$$

Un *tipar de învăţare binar* pentru un asemenea perceptron constă dintr-o pereche de vectori cu valori binare, unul de lungime n (numărul de unităţi de intrare), iar celălalt de lungime m (numărul de unităţi de ieşire). Numărul total de tipare alcătuieşte o *bază de învăţare*.

Algoritmul de învăţare este următorul:

1. Se prezintă la *intrarea* reţelei primul vector al unui tipar, activând unităţile corespunzătoare valorii 1.
2. Se prezintă la *ieşirea* reţelei vectorul 2 al aceluiaşi tipar, activând unităţile de pe nivelul 2 corespunzător valorilor 1.
3. Se caută perechi de unităţi care sunt conectate şi activate simultan, şi se ajustează valoarea ponderii conexiunii dintre acestea cu o valoare mică η. Valoarea lui η se alege în funcţie de numărul tiparelor de învăţare.
4. Se reiau paşii 1, 2, 3, pentru fiecare tipar – pereche de vectori din baza de învăţare.
5. După prezentarea tuturor tiparelor din baza de învăţare, ponderile se vor modifica astfel încât prezentând la intrare un vector 1 dintr-un tipar, activând corespunzător unităţile de pe primul nivel şi calculând ieşirile din unităţile de pe nivelul 2, se obţine vectorul 2 din tiparul de învăţare respectiv.

2.3.1 Regula delta

În 1960, Widrow şi Hoff au introdus *algoritmul celor mai mici pătrate* ("Least Mean Square Algorithm" — LMS), cunoscut şi ca *Regula Delta*, ca algoritm de corecţie a ponderilor în reţea [98]. Astfel a luat naştere elementul liniar adaptiv, denumit şi ADALINE ("ADAptive LINear Element").

Mecanismul de funcţionare care utilizează această regulă de corecţie a ponderilor este prezentat sugestiv în figura 2.6.

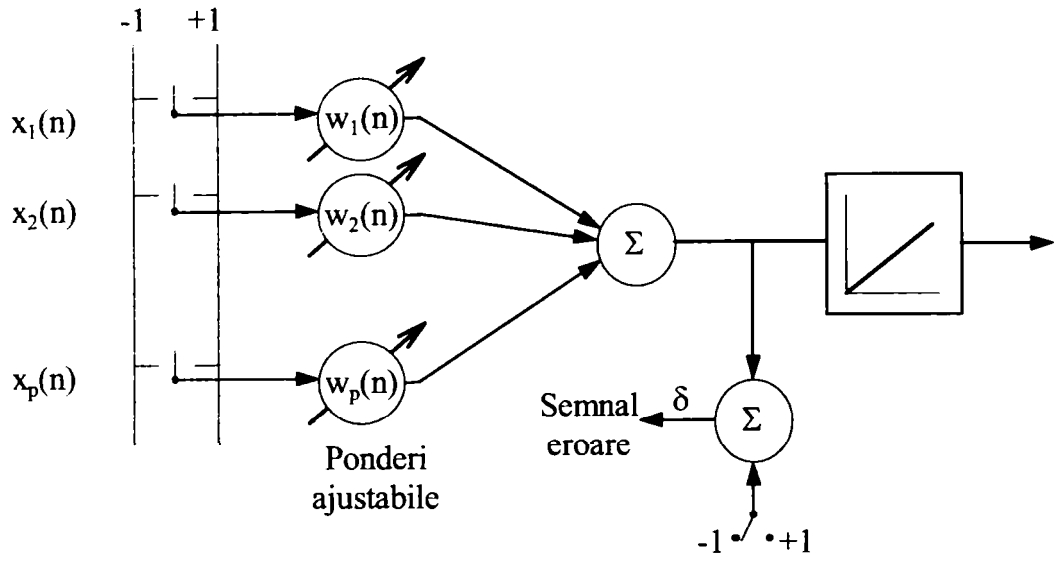


Figura 2.6: *Schema algoritmului cu LMS pentru corecţia ponderilor*

Widrow şi Hoff realizează practic o generalizare importantă a algoritmului de antrenare a perceptronului, extinzând tehnica intrărilor şi ieşirilor continue. *Regula delta* a fost aplicată mai ales unităţilor cu ieşiri pur liniare.

Pentru un astfel de perceptron, cu o singură unitate de ieşire, ieşirea este dată de relaţia:

$$o = \sum_j w_j x_j + \theta \tag{2.6}$$

Funcţia erorii, aşa cum o indică numele de "cel mai mic pătrat", este suma pătratelor erorilor. Deci eroarea totală E este definită ca:

$$E = \sum_p E^p = \frac{1}{2}\sum_p (d^p - a^p)^2 \tag{2.7}$$

unde indicele p evoluează pe setul datelor de intrare, iar E^p reprezintă eroarea setului indice p. Variabila d^p este ieşirea dorită la aplicarea setului p, iar a^p este ieşirea actuală pentru acest set. Procedura LMS găseşte valorile tuturor ponderilor care minimizează eroarea funcţiei, prin metoda numită a *gradientului descendent*. Ideea este de a efectua o schimbare în pondere, direct proporţională cu negativul derivatei erorii determinată pe setul curent, în concordanţă cu fiecare pondere:

$$\Delta^p w_j^p = -\eta \frac{\partial E^p}{\partial E^j} \tag{2.8}$$

unde η este o constantă de proporţionalitate. Derivata este

$$\frac{\partial E^p}{\partial E^j} = \frac{\partial E^p}{\partial a^p}\frac{\partial a^p}{\partial w^j} \tag{2.9}$$

Datorită liniarităţii unităţilor,

$$\frac{\partial a^p}{\partial w^j} = x_j \qquad \text{şi} \qquad \frac{\partial E^p}{\partial a^p} = -(d^p - a^p) \tag{2.10}$$

astfel încât

$$\Delta^p w_j^p = \eta \delta^p x_j \tag{2.11}$$

unde

$$\delta^p = d^p - a^p \tag{2.12}$$

este diferenţa dintre ieşirea dorită şi actuala ieşire pentru setul p. Regula delta modifică ponderea corespunzătoare ieşirii dorite pentru orice polaritate şi pentru unităţi cu intrări-ieşiri continue sau binare. Aceste caracteristici au deschis un bogat câmp de noi aplicaţii.

Spre deosebire de algoritmul iniţial, ce utilizează incrementarea progresivă, acest algoritm ia în considerare şi *greşelile* pe care le face reţeaua şi încearcă să le corecteze prin schimbarea valorilor ponderilor.

Acest algoritm are rezultate mai bune decât regula lui Hebb simplă din secţiunea 2.2.4, mai ales în cazul prezentării repetate a bazei de învăţare, luând de fiecare dată tiparele într-o altă ordine, de preferinţă aleatoare.

Etapa de traversare o dată a reţelei de către întreaga bază de învăţare poartă numele de *epocă*, iar numărul epocilor în care reţeaua învaţă cu precizia dorită întreaga bază de învăţare este un factor de performanţă al reţelei.

O deficienţă a modelului conexionist este faptul că precizia de învăţare a tiparelor variază de la un tipar la altul, în funcţie de modul în care se încadrează tiparul în logica ponderată a bazei.

În cazul în care se doreşte ca reţeaua să obţină la ieşire vectorul caracteristic unui vector de intrare care nu face parte din baza de învăţare, dar este asemănător ca distribuţie cu vectorii de intrare din baza de învăţare, este necesară definirea unui parametru ce caracterizează performanţa de învăţare şi anume *eroarea totală* — relativă la toate tiparele:

$$E = \sum_{\mu=1}^{p} \sum_{i=1}^{m} \delta_i^{\mu} \tag{2.13}$$

Eroarea maximă a reţelei este

$$E_{max} = m \cdot p$$

dacă vectorii au valori binare de tipul $(0, 1)$, şi

$$E_{max} = 2 \cdot m \cdot p$$

dacă valorile sunt de tipul $(-1, 1)$.

Raportul

$$\frac{E}{E_{max}} \cdot 100 = E_{\%}$$

este eroarea relativă procentuală a reţelei, şi este utilizat ca şi criteriu de oprire a etapei de învăţare a reţelei.

Datorită avantajului acestui model, de a recunoaşte vectorul de ieşire în cazul prezentării la intrare a unui vector care nu face parte din baza de învăţare, perceptronul a fost utilizat pe scară largă.

Pe scurt, diferenţele principale dintre ADALINE şi Perceptron sunt:

- la ADALINE are loc o ponderare liniară, ce se formează adaptiv (printr-o metodă de antrenare supervizată), care produce semnalul de răspuns dorit pentru vectori de intrare diferiţi; rezultatul este optim, în sensul celor mai mici pătrate.
- la Perceptron, există o unitate de procesare suplimentară, pentru extracţia combinaţiilor de semnal din tiparele de intrare. Circuitele de ieşire au ca singur scop sesizarea diferenţei dintre vectorii de intrare, fiecare din ei corespunzând unei clase diferite de tipare.

După ADALINE, Widrow propune şi realizează pentru prima dată o reţea neuronală stratificată, antrenabilă, cu elemente adaptive multiple, în *Generalization and information storage in networks of adaline "neurons"* [97].

2.3.2 Problema XOR

Euforiei generate de perceptron i-au pus capăt în 1969 Minsky şi Papert, care utilizând argumente matematice simple dar elegante, au demonstrat că un perceptroncu un singur strat, numit în continuare *perceptron uni-strat* are limitele sale [58]. În acest context, una dintre problemele care au rămas drept istorice este problema imposibilităţii unui perceptron uni-strat de a reprezenta o funcţie sau-exclusiv (XOR).

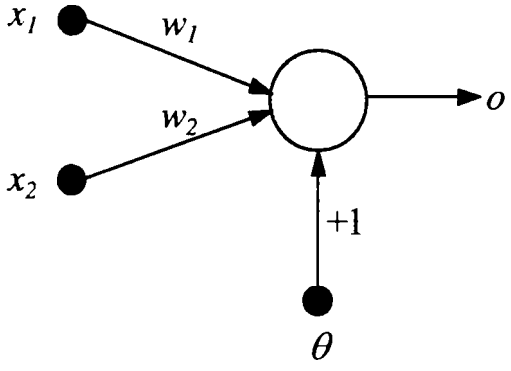


Figura 2.7: *Perceptron uni-strat cu 2 intrări*

În perceptronul din figura 2.7, valoarea totală a intrării este dată de

$$i = w_1 \cdot x_1 + w_2 \cdot x_2 + \theta \tag{2.14}$$

Pentru o constantă θ , ieşirea perceptronului este într-unul din semiplanele delimitate de linia de diviziune definită de ecuaţia

$$w_1 \cdot x_1 + w_2 \cdot x_2 = -\theta \tag{2.15}$$

Conform figurii 2.8, spaţiul de intrare constă din cele patru puncte (cercuri); cercurile pline, localizate la $(1,-1)$ şi $(-1,1)$ nu pot fi separate printr-o linie dreaptă de cercurile goale, localizate la $(-1,-1)$ şi $(1,1)$.

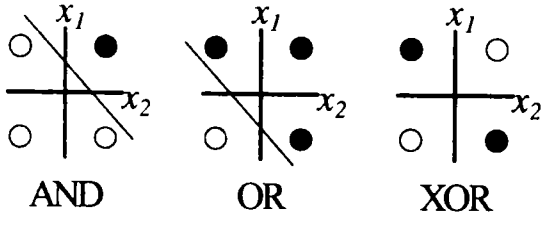


Figura 2.8: *Reprezentarea geometrică a spaţiului intrărilor pentru perceptronul uni-strat*

2.3.3 Perceptronul multistrat

Tot Minsky şi Papert au dat şi soluţia, introducând un nou strat, conectat la toate intrările. Au extins astfel reţeaua la perceptronul multistrat, care poate rezolva problema XOR .

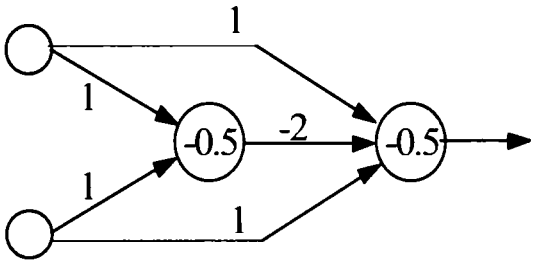


Figura 2.9: *Perceptronul cu două intrări şi unitate ascunsă pentru problema XOR*

Rezolvarea problemei XOR este prezentată în figura 2.9.

De data aceasta, cele două grupuri de puncte sunt separate spaţial printr-un plan de separare, ca în figura 2.10.

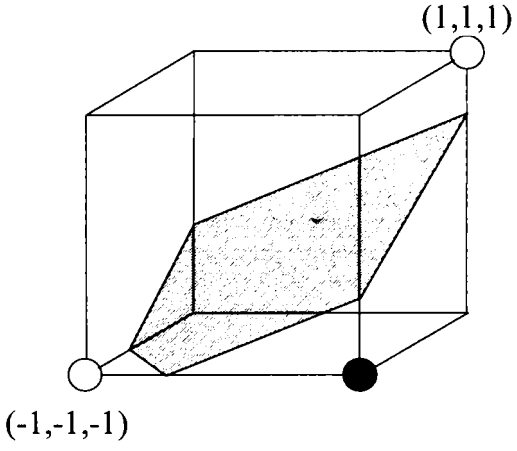


Figura 2.10: *Reprezentarea spaţială a separării celor două grupuri de puncte.*

Acest exemplu simplu demonstrează că adăugarea unor unităţi ascunse creşte clasa de aplicaţii rezolvabile de către perceptronul multistrat.

În figura 2.11 se prezintă schema unui perceptron având la intrare un vector de dimensiune n, iar la ieşire un vector m-dimensional. Fiecare intrare i este multiplicată pentru toate unităţile din reţea, care vor avea n intrări.

O problemă importantă întâlnită în proiectarea reţelelor tip perceptron multistrat a fost problema *acordării de credit neuronilor din stratul ascuns al reţelei* ("credit assignment theory"). Cel care a introdus pentru prima dată terminologia de *credit assignment* a fost tot Minsky [57], ulterior căruia în anii '60 strădaniile cercetătorilor din domeniu au fost dedicate problemelor legate de această teorie, cu rezultate contradictorii, fără a o rezolva definitiv. A trebuit să vină perioada anilor '80 pentru a găsi soluţiile de ieşire din impas. Printre motivele unei asemenea întârzieri:

- tehnologic — nu existau calculatoare personale sau staţii de lucru suficient de performante care să permită experimentarea;

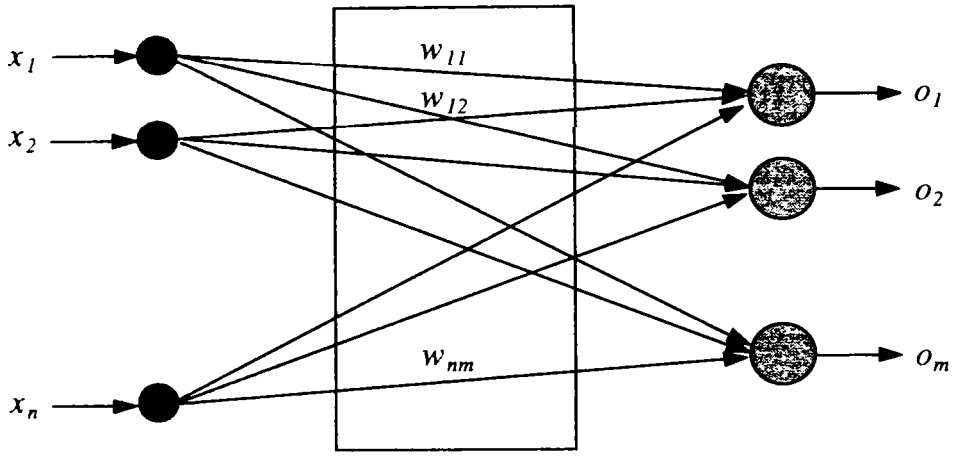


Figura 2.11: *Structura schematică a unui perceptron multistrat.*

- psihologic — monografia [58] nu a fost deloc încurajatoare pentru continuarea studiilor în domeniu;
- financiar — derivat din cel psihologic, care nu încuraja nici o agenţie în lansarea unor programe de cercetare în domeniu.

Din nefericire, aceste motive, şi altele, au dus la sistarea interesului pentru studiul reţelelor neuronale. Singurele cercetări ale anilor '70, totuşi colaterale domeniului, au fost cele în psihologie şi neurologie. De aceea, această decadă a fost caracterizată ca fiind "the dormancy decade" (decada de somnolenţă).

2.4 Modelul Hopfield

În 1982, Hopfield a utilizat ideea unei *funcţii de energie* [34], pentru a reformula un nou mod de înţelegere a calculelor dintr-o reţea neuronală, folosind conexiuni sinaptice simetrice, şi stabilind o analogie între o reţea neuronală recurentă şi *modelul Ising* utilizat în fizica statistică. Aceste tipuri de reţele sunt cunoscute sub denumirea de reţele Hopfield.

2.4.1 Modelul fizic Ising

Conform acestui model, un material magnetic este compus dintr-o reţea regulată de atomi cu proprietăţi magnetice, care au asociată o mărime numită *spin magnetic*, orientat vectorial în sensul dat de compunerea forţelor magnetice la care este supus atomul. Pentru cazul ipotetic în care este permisă orientarea spinului pe o singură direcţie, în două sensuri, valoarea acestuia poate fi memorată într-o variabilă binară s_i. Alegând pentru aceasta variabilă valorile -1 şi 1, se obţine un model simplificat - modelul Ising (figura 2.12):

Fiecare spin s_i este influenţat de câmpul magnetic H_i, care este compus din câmpul magnetic exterior H_{ext} şi câmpul produs de atomii învecinaţi.

Ising a propus pentru calculul lui H_i relaţia:

$$H_i = \sum_j p_{ij} \cdot s_j + H_{ext} \tag{2.16}$$

unde p_ij este un coeficient dat de poziţia relativă dintre atomii i şi j.

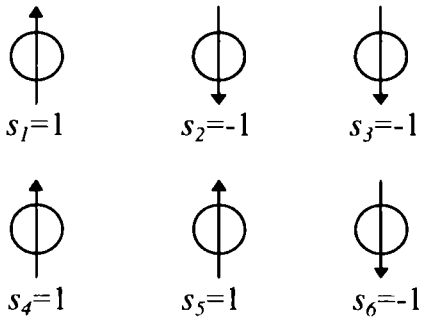


Figura 2.12: *Modelul fizic Ising.*

Materialul magnetic poate avea înmagazinată o energie potenţială a cărei valoare poate fi calculată cu relaţia:

$$H = -\frac{1}{2}\sum_i \sum_j p_{ij} \cdot s_i \cdot s_j - H_{ext} \sum_i s_i \tag{2.17}$$

Pentru simplificarea discuţiei în ce priveşte utilizarea modelului Ising la construcţia unei reţele neuronale, în continuare se va considera $H_{ext} = 0$.

2.4.2 Definirea modelului Hopfield

Trecerea de la modelul Ising la o structură conexionistă a fost definitivată de Hopfield în 1984 [35]. Reţeaua conexionistă bazată pe acest model este topologic diferită de perceptron, *având fiecare unitate legată de celelalte unităţi*, toate legăturile fiind ponderate. Nu există diferenţiere între unităţile de intrare şi cele de ieşire, ieşirea fiecărei unităţi fiind multiplicată ca intrare către toate unităţile, inclusiv ea însăşi. Ponderile reprezintă o matrice de dimensiune $N \times N$, unde N este numărul de unităţi.

Funcţionarea reţelei este, de asemenea, diferită de cea a perceptronului. Astfel, dacă la o reţea structurată pe straturi de intrare, respectiv de ieşire, aplicăm un vector de intrare primului strat şi calculăm activările din cel de al doilea strat pentru a determina ieşirea, pentru noua reţea vom considera ca intrare o stare generală, a întregii reţele, reprezentabilă printr-un vector de stare $S = [S_1, S_2, \ldots, S_N]$.

Datorită câmpului magnetic exercitat asupra lor, atomii dintr-un material magnetic îşi pot schimba spinul. Acceptând prin convenţie că o unitate este activă dacă are spinul egal cu 1 şi este inactivă dacă are spinul egal cu -1, în modelul Ising, valoarea spinului se poate calcula cu funcţia *signum*. Astfel, dacă semnul valorii câmpului din locaţia atomului este pozitiv, atunci şi spinul este pozitiv, iar dacă este negativă spinul este şi el negativ. Considerând că $h_i = H_i$, (intrarea într-o unitate este egală cu câmpul) şi $w_{ij} = p_{ij}$ (ponderea unei intrări este egală cu coeficientul de interacţiune dintre atomi), se poate afirma că unităţile acestei reţele conexioniste au următoarea funcţie de transfer:

$$s_i = signum(h_i) = \begin{cases} 1 & \text{dacă } h_i \geq 0 \\ -1 & \text{dacă } h_i < 0 \end{cases} \tag{2.18}$$

numită *lege dinamică de evoluţie*, unde:

$$h_i = \sum_{j=1}^{N} w_{ij} s_j \tag{2.19}$$

Un astfel de sistem caută să ajungă întotdeauna într-o stare de energie potenţială minimă. Unităţile îşi vor schimba valoarea de activare până la un moment în care schimbarea activării oricărei unităţi nu va mai duce la scăderea energiei potenţiale a întregului sistem.

Se poate demonstra că un astfel de sistem ajunge întotdeauna într-o stare de minim energetic — stabilă. Considerând energia potenţială dată de relaţia:

$$H = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ij}s_{ij} \tag{2.20}$$

şi $s_i' = sign(h_i')$, ea fiind starea următoare a lui s_i, vor putea exista două situaţii:

1. $-s_i' = s_i \Rightarrow$ unitatea i nu îşi schimbă activarea $\Rightarrow \Delta H = 0$
2. $-s_i' = -s_i \Rightarrow$ unitatea îşi schimbă activarea

În primul caz, $H' = H$, deci energia rămâne aceeaşi. În cel de-al doilea caz,

$$\Delta H = H' - H = -\frac{1}{2}\sum_{i,j} w_{ij}s_i's_j + \frac{1}{2}\sum_{i,j} w_{ij}s_is_j \tag{2.21}$$

Înlocuind s_i' cu $-s_i$, se obţine:

$$\Delta H = \sum_i s_i \sum_j w_{ij}s_j = \sum_i s_i h_i' \tag{2.22}$$

Având în vedere ca semnul lui h_i' este şi semnul lui s' şi că $s_i = s_i'$, rezultă că ΔH este întotdeauna negativ atunci când o unitate îşi schimbă activarea, deci sistemul va evolua întotdeauna către o stare de minim energetic.

Pentru un singur tipar de învăţare ξ constând dintr-un vector binar cu componente $(-1, 1)$ de lungime $N \times N$, relaţia de calcul a ponderilor va fi:

$$w_{ij} = \frac{1}{N}\xi_i\xi_j \qquad (i = \overline{1,N}; j = \overline{1,N}) \tag{2.23}$$

deci matricea de ponderi va fi o matrice simetrică.

Pentru mai multe tipare de învăţare, ξ^μ $(\mu = \overline{1,p})$ relaţia de calcul a ponderilor va fi:

$$w_{ij} = \frac{1}{N}\sum_{\mu=1}^{p} \xi_i^\mu \xi_j^\mu \qquad (i = \overline{1,N}; j = \overline{1,N}) \tag{2.24}$$

Având în vedere capacitatea limitată de memorare şi posibilitatea apariţiei stărilor mixate ca o combinaţie liniară a stărilor învăţate, acest tip de reţele nu au aplicaţii practice.

Publicaţiile lui Hopfield au generat numeroase controverse. În 1983 Cohen şi Grossberg au stabilit un principiu general pentru proiectarea unei memorii cu conţinut direct adresabil care a inclus şi versiunea reţelei Hopfield ca şi caz particular [14]. O componentă distinctivă a convergenţei unei astfel de reţele neuronale este modul natural în care timpul ca şi dimensiune esenţială a învăţării se manifestă în dinamica neliniară a reţelei.

In 1983, Kirkpatrick, Gellat şi Vecchi au descris o nouă procedură numită algoritm de *simulare a răcirii treptate* ("simulated annealing algorithm") [43], ideea fiind preluată de la Metropolis, Rosenbluth şi Teller [54], şi îşi are rădăcina în statistica termodinamică pentru rezolvarea unor probleme de optimizare combinatorială. Această idee a fost exploatată în 1985 de Ackley, Hinton şi Sejnowski [2] în dezvoltarea

algoritmului stohastic de învăţare ce utilizează proprietăţi ale distribuţiei Boltzmann, denumit *învăţarea Boltzmann*.

Metoda presupune pentru S_i o dependenţă de tipul:

$$S_i = \begin{cases} +1 & \text{cu probabilitatea } g(h_i) \\ -1 & \text{cu probabilitatea } 1 - g(h_i) \end{cases} \tag{2.25}$$

unde

$$h_i = \sum_j w_{ij} S_j \tag{2.26}$$

Funcţia de probabilitate $g(h)$ trebuie să îndeplinească condiţiile:

1. să ia valori în intervalul $[0, 1]$
2. să fie dependentă de temperatură.

Cea mai utilizată funcţie de probabilitate este cea sigmoidă:

$$F(x) = \frac{1}{1 + e^{-x}} \qquad F(x) : \mathbb{R} \to [0, 1] \tag{2.27}$$

Pentru a simula dependenţa de temperatură a funcţiei, se utilizează factorul de temperatură β astfel:

$$F(x) = \frac{1}{1 + e^{-\beta x}} \qquad \beta = \frac{1}{K_B \cdot T_K} \tag{2.28}$$

unde K_B este constanta lui Boltzmann, iar T_K este temperatura absolută în K. Pentru diferite valori ale lui β se obţine o variaţie a funcţiei de genul celui ilustrat în figura 2.13. Iniţial se porneşte de la o temperatură ridicată, fapt ce induce o valoare mică a valorii lui β, având ca şi consecinţă o pantă mică a funcţiei de probabilitate, urmând ca la creşterea valorii temperaturii panta funcţiei să crească în mod considerabil.

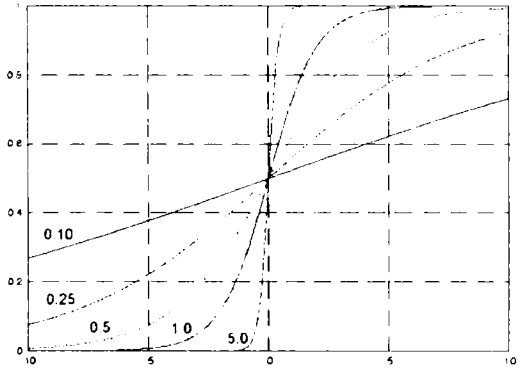


Figura 2.13: *Variaţia funcţiei sigmoidă pentru diferite valori ale lui β*

Lăsând sistemul să evolueze "la o temperatură ridicată", el va ajunge într-o stare stabilă, care nu este una din stările învăţate, dar este mai joasă din punct de vedere energetic. În acest moment, la scăderea temperaturii în mod progresiv, sistemul va ajunge la una din stările învăţate sau la negativele acestora. Capacitatea unei astfel de reţele Hopfield stohastice este similară cu cea Hopfield discretă – prezentată anterior, evoluţia fiind doar în ce priveşte lansarea în domeniu a *funcţiei de activare sigmoide* care a stat la baza trecerii la *reţele cu propagare înapoi*.

Tot în 1984, Braitenberg publică o carte "Experimente în psihologia sintetică" [8], în care descrie diverse mecanisme cu capacitate de autoorganizare, inspirate din comportamentul animal.

2.5 Reţele cu propagare înapoi (backpropagation)

Deşi fusese descris de către Bryson şi Ho în 1969 [9], şi de către Werbos în 1974 [96], abia în 1986 Rumelhart, Hinton şi Williams publică studiul algoritmului de propagare înapoi ("the back-propagation algorithm") [77], spre care se îndreaptă atenţia cercetătorilor în domeniu.

În acelaşi an, Rumelhart şi McCleland publică în două volume cartea "Procesare paralelă şi distribuită: Explorări în microstructurile cunoaşterii" ce se va constitui în perioada următoare ca referinţă de bază în studiul antrenării perceptronului multistrat [76]. Aceeaşi idee, dar cu o rezonanţă mai mică, o utilizează şi Parker în 1985–1987 [63], [64] şi Le Cun în 1985 [15].

Ideea centrală a acestui model este că erorile pentru unităţile de pe stratul intermediar sunt determinate prin *propagare înapoi*[2] a erorilor de pe stratul de ieşire. Propagarea înapoi este o generalizare a *regulii delta* pentru funcţii de activare neliniare aplicate reţelelor multistrat.

O reţea neuronală cu propagare înapoi (figura 2.14) are o structură multistrat, fiecare strat primind intrările de pe stratul anterior şi trimiţând ieşirile stratului imediat următor. În cadrul unui strat nu există conexiuni între unităţile proprii. Cele N_i intrări alimentează primul strat, cu $N_{h,1}$ unităţi ascunse. Unităţile de intrare sunt doar unităţi generatoare, în ele neavând loc nici un proces de prelucrare a datelor. Activarea unităţilor ascunse se face prin aplicarea unei funcţii F asupra ponderilor care le preced şi adăugarea unei constante denumită *bias* care nu este obligatorie. Ieşirea unităţilor ascunse este distribuită stratului imediat următor de unităţi ascunse $N_{h,2}$ care, dacă este ultimul strat de unităţi ascunse, alimentează cele N_o unităţi de pe stratul de ieşire.

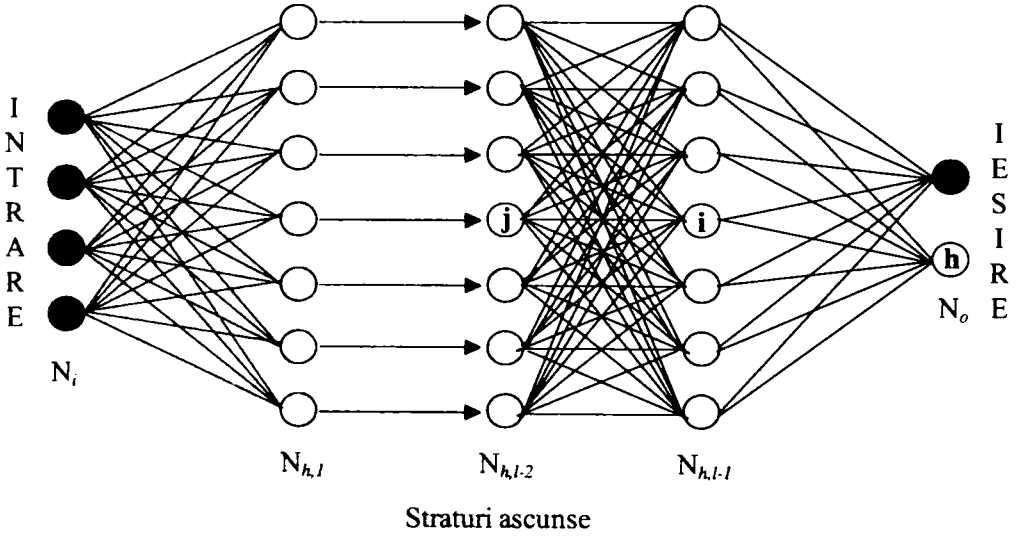


Figura 2.14: *Reţea neuronală cu l straturi ascunse*

Algoritmul de propagare înapoi poate fi aplicat oricărui tip de reţele, cu oricât de multe straturi. Pentru cazul reţelelor cu unităţi binare, s-a demonstrat, de către Hornik, Stinchcombe şi White în 1989 [36], [37] şi de către Hartman în 1990 [30], că

[2] "by back-propagation"

un singur strat de unităţi ascunse este suficient pentru a aproxima orice funcţie cu un număr mare, dar finit de discontinuităţi, pentru funcţii de activare neliniare aplicate pentru straturile ascunse.

Pentru a utiliza funcţii neliniare de activare, este necesară *generalizarea regulii delta* care este specifică funcţiilor liniare.

2.5.1 Generalizarea regulii delta

Funcţia de activare este o funcţie diferenţiabilă a intrării dată de relaţia:

$$a_i^p = F(i_i^p) \tag{2.29}$$

în care a_i^p este ieşirea unităţii i obţinută pentru intrarea i_i^p în cazul tiparului p. Valoarea de intrare este obţinută prin relaţia:

$$i_i^p = \sum_j w_{i,j} a_j^p + \theta_i \tag{2.30}$$

în care θ_i este o valoare tip *bias*.

Pentru obţinerea unei generalizări corecte a *regulii delta*, se admite:

$$\Delta^p w_{i,j} = -\eta \frac{\partial E^p}{\Delta w_{i,j}} \tag{2.31}$$

Mărimea E^p este definită ca eroarea pătratică totală pentru tiparul p la ieşire:

$$E^p = \frac{1}{2} \sum_p^{N_{tip}} (d_i^p - a_i^p)^2, \tag{2.32}$$

expresie ce denotă termenul care trebuie minimizat pe parcursul evoluţiei reţelei. Coeficientul $\frac{1}{2}$ din faţa expresiei este un coeficient de convenienţă pentru simplificarea expresiei gradientului obţinut prin derivare, acesta neafectând localizarea minimului de eroare şi nici procesul de minimizare în sine.

Termenul erorii totale a reţelei la un anumit moment, este dat de suma erorilor tuturor tiparelor:

$$E^T = \sum_p^{N_{tip}} E^p \tag{2.33}$$

Se poate scrie:

$$\frac{\partial E^p}{\partial w_{i,j}} = \frac{\partial E^p}{\partial i_i^p} \frac{\partial i_i^p}{\partial w_{i,j}} \tag{2.34}$$

Din relaţia 2.34 se observă că cel de al doilea factor este:

$$\frac{\partial i_i^p}{\partial w_i^p} = a_j^p \tag{2.35}$$

Definind:

$$\delta_i^p = \frac{\partial E^p}{\partial i_i^p} \tag{2.36}$$

se obţine o regulă nouă, care este echivalentă cu regula delta, constând dintr-un gradient descendent al suprafeţei de eroare, dacă schimbările ponderilor se vor face în concordanţă cu relaţia:

$$\Delta^p w_{i,j} = \eta \delta_i^p a_j^p \tag{2.37}$$

Ceea ce rămâne de demonstrat este ce reprezintă δ_i^p pentru fiecare unitate i din reţea. Pentru calculul lui δ_i^p se va aplica regula derivării înlănţuite pentru a scrie derivata parţială ca un produs de doi factori, unul reflectând variaţia erorii ca funcţie de ieşire, iar celălalt reflectând schimbarea ieşirii ca funcţie de schimbarea intrării. Astfel, rezultă:

$$\delta_i^p = -\frac{\partial E^p}{\partial i_i^p} = -\frac{\partial E^p}{\partial a_i^p}\frac{\partial a_i^p}{\partial i_i^p} \tag{2.38}$$

Din relaţia 2.38 se observă că:

$$\frac{\partial a_i^p}{\partial i_i^p} = F'(i_i^p), \tag{2.39}$$

expresie care este derivata funcţiei pentru unitatea i, evaluată la intrarea i_i^p a acestei unităţi. Pentru a calcula primul factor din relaţia 2.38, vor fi considerate două cazuri:

1. **Unitatea i este o unitate de ieşire a reţelei**. În acest caz, din definiţia erorii tiparului E^p, se obţine:
$$\delta_i^p = (d_i^p - a_i^p)F_i'(i_i^p) \tag{2.40}$$
pentru o unitate de ieşire i.

2. **Unitatea i nu este o unitate de ieşire**. În acest caz, nu se cunoaşte contribuţia unităţii la eroarea de ieşire a reţelei. Deci, valoarea erorii poate fi scrisă ca funcţie de intrările din stratul ascuns spre stratul de ieşire:
$$E^p = E^p(i_1^p, i_2^p, \ldots, i_i^p, \ldots) \tag{2.41}$$
Utilizând aceeaşi regulă de derivare înlănţuită, rezultă:
$$\frac{\partial E^p}{\partial a_i^p} = \sum_{h=1}^{N_h}\frac{\partial E^p}{\partial i_h^p}\frac{\partial i_h^p}{\partial a_i^p} = \sum_{h=1}^{N_h}\frac{\partial E^p}{\partial i_h^p}\frac{\partial}{\partial a_i^p}\sum_{k=1}^{N_h} w_{hk}a_k^p = \sum_{h=1}^{N_o}\frac{\partial E^p}{\partial i_h^p}w_{h,i} = -\sum_{h=1}^{N_o}\delta_h^p w_{h,i} \tag{2.42}$$
Substituind rezultatul în ecuaţia 2.38, rezultă:
$$\delta_i^p = F'(i_i^p)\sum_{h=1}^{N_o}\delta_h^p w_{h,i} \tag{2.43}$$

Ecuaţiile 2.40 şi 2.43 redau o procedură de calcul a valorii lui δ pentru toate unităţile din reţea, care la rândul lor sunt utilizate pentru a calcula schimbările în valoare ale ponderilor în concordanţă cu ecuaţia 2.37. Aceasta procedură constituie *generalizarea regulii delta* pentru o reţea cu unităţi neliniare.

2.5.2 Modul de lucru cu *backpropagation*

Aplicarea *regulii delta generalizată* implică două faze:

1. În decursul primei faze, se aplică intrarea şi se propagă de-a lungul reţelei pentru calculul valorilor a_i^p pentru fiecare termen al unităţii de ieşire; această valoare este comparată cu valoarea dorită, rezultând un semnal de eroare δ_i^p pentru fiecare termen al unităţii de ieşire;

2. Cea de-a doua fază implică o trecere înapoi prin reţea, în decursul căreia semnalul de eroare este transmis fiecărui element din reţea, calculându-se schimbările de pondere adecvate. Ponderea conexiunii este ajustată cu o cantitate proporţională cu produsul semnalului de eroare δ pe unitatea i, primind intrarea, iar ieşirea unităţii j trimiţând acest semnal de-a lungul conexiunii:
$$\Delta^p w_{ij} = \eta_{ij}\delta_i^p a_j^p \tag{2.44}$$
unde

w_{ij} ponderea conexiunii dintre unitatea i şi unitatea j

$\Delta^p w_{ij}$ termenul de ajustare a ponderii conexiunii

η_{ij} rata de învăţare asociată cu ponderea w_{ij}

δ_i^p semnalul de eroare asociat elementului i pentru vectorul p aplicat la intrarea reţelei

a_i^p valoarea de activare a elementului i al reţelei pentru vectorul p aplicat la intrarea reţelei

Dacă unitatea este situată la ieşirea din reţea, semnalul de eroare este dat de:

$$\delta_k^p = (d_k^p - a_k^p)F'(z_k^p) \tag{2.45}$$

unde

z_k^p intrarea netă a neuronului k pentru un vector de intrare p

d_k^p al k-lea element al vectorului de ieşire dorit al reţelei pentru vectorul p aplicat la intrarea reţelei

Definită ca *funcţia de activare sigmoidă*, funcţia F este:

$$a_j^p = F(y_j^p) = \frac{1}{1 + e^{-y_j^p}} \tag{2.46}$$

În cazul utilizării funcţiei de activare temperată a_j^p devine:

$$a_j^p = F(y_j^p) = \frac{1}{1 + e^{-\beta y_j^p}} \tag{2.47}$$

unde:

$$\beta = \frac{1}{K_B \cdot T_K},$$

K_B este constanta lui Boltzmann

T_K este temperatura absolută în grade Kelvin

Derivata acestei funcţii este:

$$F'(y_j^p) = a_j^p(1 - a_j^p) \tag{2.48}$$

$$F'(i_i^p) = \frac{\partial}{\partial i_i^p}\frac{1}{1 + e^{-i_i^p}} = \frac{1}{(1 + e^{-i_i^p})^2}(-e^{i_i^p}) = \frac{1}{1 + e^{-i_i^p}}\frac{-e^{i_i^p}}{1 + e^{-i_i^p}} = a_i^p(1 - a_i^p) \tag{2.49}$$

astfel încât semnalul de eroare pentru o unitate de ieşire poate fi scris ca:

$$\delta_j^p = (d_j^p - a_j^p)a_j^p(1 - a_j^p) \tag{2.50}$$

Semnalul de eroare pentru o unitate ascunsă este determinat recursiv în termenii semnalelor de eroare ale unităţii cu care este direct conectată şi ponderile acestor conexiuni. Pentru funcţia de activare sigmoidă:

$$\delta_j^p = F'(y_j^p)\sum_{k=1}^{N}\delta_k^p w_{j,k} = a_j^p(1 - a_j^p)\sum_{k=1}^{N}\delta_k^p w_{j,k} \tag{2.51}$$

2.5.3 Alte funcţii de activare neliniare

Practic, numărul funcţiilor care pot fi utilizate ca funcţii de activare este relativ nelimitat. Acestea se pot clasifica în două clase:

2.5.3.1 Funcţii nesimetrice faţă de origine similare ca alură cu funcţia sigmoidă

Din această categorie, alături de funcţia sigmoidă, cea mai utilizată este funcţia tangentă hiperbolică. Această funcţie este definită de relaţia:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.52}$$

Funcţia tangentă hiperbolică este reprezentată grafic ca în figura 2.15.

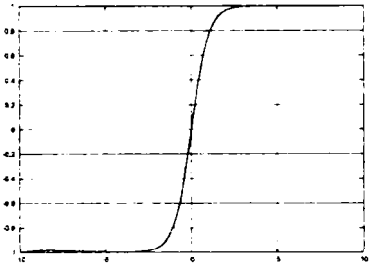


Figura 2.15: *Reprezentarea grafică a funcţiei tangentă hiperbolică*

2.5.3.2 Funcţii simetrice faţă de origine

Din această categorie fac parte:

Funcţia Gaussiană — aceasta este o funcţie radială, definită de relaţia:

$$F_{Gauss}(x) = e^{-\frac{x^2}{\nu}} \tag{2.53}$$

în care ν este o constantă predefinită, numită *varianţă*, cu ajutorul căreia este ajustată amplitudinea funcţiei. Funcţia Gaussiană este redată în figura 2.16.

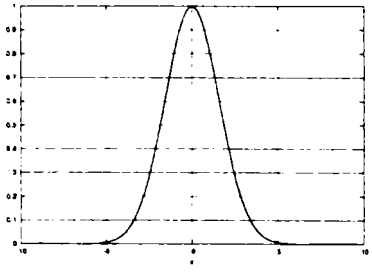


Figura 2.16: *Reprezentarea grafică a funcţiei Gaussiene.*

Funcţia secantă hiperbolică — şi această funcţie este o funcţie radială, fiind definită de relaţia:

$$\operatorname{sech}(x) = \frac{2}{e^x + e^{-x}} \tag{2.54}$$

Utilizarea unei anumite funcţii de activare este o opţiune personală a programatorului, neexistând argumente tehnice privind indici de performanţă care să justifice o selecţie preferenţială.

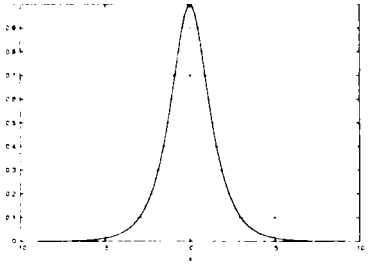


Figura 2.17: *Reprezentarea grafică a funcţiei secantă hiperbolică.*

2.5.4 Rata de învăţare şi momentul

Procedura de învăţare impune ca actualizarea ponderii să fie proporţională cu $\frac{\partial E^p}{\partial w}$. Scăderea reală a gradientului necesită utilizarea unor paşi infinitezimali. Constanta de proporţionalitate este rata de învăţare η. Pentru aplicaţiile practice se alege o rată de învăţare cât mai mare posibilă, fără a se ajunge însă la *oscilaţie*. O modalitate de evitare a oscilaţiei la valori mari ale lui η este efectuarea unei actualizări în pondere dependentă de actualizarea anterioară a ponderii prin adăugarea unui termen *moment*, γ:

$$\Delta w_{i,j}(t+1) = \eta \delta_i^p a_j^p + \gamma \Delta w_{i,j}(t) \qquad \gamma \in (0,1) \tag{2.55}$$

unde t indexează numărul epocii, iar γ este o constantă care determină efectul schimbării anterioare a ponderii. Valoarea lui γ poate fi de exemplu 0.1, scăzând la creşterea erorii sau crescând la scăderea ei.

Rolul termenului *moment* este arătat în figura 2.18. Când nu este utilizat termenul *moment*, găsirea minimului cu o rată mică de învăţare necesită o perioada lungă de timp (1), iar pentru o rată de învăţare mare, minimul nu poate fi găsit din cauza fenomenului de oscilaţie (2). Adăugarea unui termen *moment* duce la o *găsire mai rapidă a valorii de minim* (3).

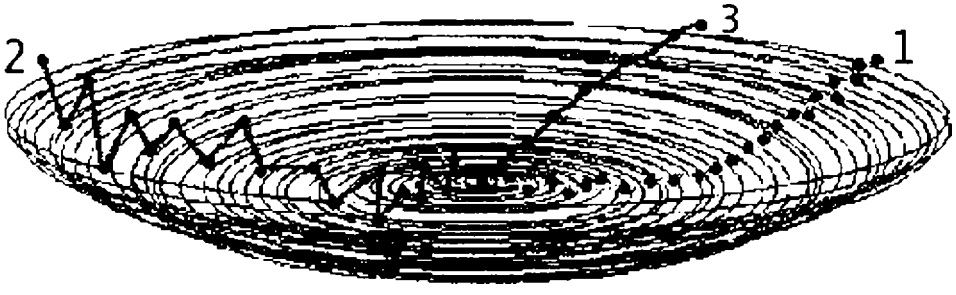


Figura 2.18: *Variaţia erorii reţelei funcţie de valoarea momentului.*

2.5.5 Învăţarea unui tipar

Teoretic, algoritmul *backpropagation* determină scăderea gradientului erorii totale a reţelei *doar dacă ponderile sunt ajustate pe întregul set de tipare*. Se va acorda o atenţie mărită ordinii în care setul de tipare este prezentat reţelei deoarece, în cazul în care este utilizată aceeaşi secvenţă, reţeaua se poate focaliza pe ultimele tipare

prezentate. Problema se poate depăşi dacă se utilizează un algoritm aleator sau o tehnică de permutare la prezentarea tiparelor în etapa de antrenare.

2.5.6 Deficienţe ale algoritmului backpropagation

Contrar succesului obţinut de algoritmul backpropagation, există câteva aspecte care fac ca utilizarea algoritmului să nu reprezinte garanţia succesului universal.

Se vor prezenta mai jos trei dintre problemele cel mai des întâlnite, oferindu-se şi câteva din motivele posibile ale manifestării lor.

O antrenare de lungă durată, cu un gradient al erorii totale scăzut — se poate datora unei selecţii neoptimale a ratei de învăţare şi a termenului moment. Există algoritmi care pornesc cu o rată a învăţării mică, în funcţie de variaţia erorii aceasta modificându-se corespunzător — la o variaţie decrementală a erorii rata de învăţare crescând, în vreme ce ea scade la o variaţie incrementală.

Paralizia reţelei — se poate datora valorilor foarte mari, în valoare absolută, pe care le pot lua ponderile pe parcursul antrenării reţelei, valori care induc valori de activare apropiate de 0 sau 1. Conform relaţiilor 2.50 şi 2.51, termenul de ajustare a valorii ponderilor, care este direct proporţional cu $a_j^p(1 - a_j^p)$, va avea valori nesemnificative, fapt ce duce la un palier staţionar de evoluţie pentru gradientul de eroare.

Convergenţa spre un minim local — având în vedere că hiper-suprafaţa de eroare pentru o reţea complexă este accidentată, există posibilitatea ca reţeaua să se afle într-un minim local de valoare nesatisfăcătoare în comparaţie cu minimul absolut al reţelei. Există metode probabilistice de tipul *algoritmilor genetici*, prin a căror utilizare se pot evita capcanele locale, dar acestea au dezavantajul de a fi lente. O altă posibilitate este de a mări numărul de unităţi de pe stratul ascuns, metodă care are acelaşi dezavantaj — este lentă.

Această problemă este cea mai spinoasă, existând în literatură o serie de articole care pretind rezolvarea parţială a ei, însă *reţetele* oferite nu au o eficienţă generală, ci doar evită problema în contextul particular în care este studiată.

Spre exemplificare, în 1994, Klawun şi Wilkins publică un articol denumit "Un nou algoritm pentru ieşirea dintr-un minim local" [44], care constă în prezentarea, în cazul sesizării unui minim local (sesizarea se face prin constatarea unei erori maxime pentru unele tipare) pentru câteva epoci, doar a tiparelor cu eroare maximă (metodă cunoscută şi sub denumirea de *flashcard-algorithm*). Pentru moment, eroarea totală a reţelei creşte dar îşi urmează cursul descendent, depăşind impasul şi convergând spre valoarea de minim global.

Această metodă este doar o observaţie de bun simţ foarte des folosită de către utilizatorii de programe personale dedicate, programe care permit monitorizarea în timpul antrenării reţelei a parametrilor de reţea şi intervenţia în momentele de antrenare ineficace. Este de remarcat însă efortul logistic şi financiar alocat pentru elaborarea metodei.

Acelaşi rezultat se obţine şi prin bruiajul reţelei, prin impunerea unei valori anormale unei conexiuni din reţea, bruiajul urmând a se propaga prin întreaga reţea, fapt ce conduce pentru moment la o valoare mare a erorii totale, concomitent cu scoaterea reţelei din minimul local.

2.6 Îmbunătăţiri ale algoritmului backpropagation

Cercetătorii în domeniu au lansat variaţiuni pe tema algoritmului backpropagation care s-au dorit a fi algoritmi îmbunătăţiţi în ce priveşte etapa de antrenare a reţelei.

Evaluarea contribuţiei fiecăruia este prematură, valoarea noilor algoritmi trebuind să treacă testul de anduranţă al aplicării şi verificării în condiţii în care metodele actuale nu dau rezultate satisfăcătoare. În această secţiune se vor discuta câţiva dintre aceşti algoritmi.

O îmbunătăţire semnificativă în domeniu este înlocuirea metodei de minimizare simplistă pe baza gradientului la fiecare pas, cu o metodă de minimizare direcţionată, cuplată cu minimizarea gradientului conjugat. Minimizarea unei funcţii f de-a lungul unei direcţii u aduce funcţia f într-un loc în care gradientul ei este perpendicular pe direcţia u. În caz contrar, procesul de minimizare de-a lungul direcţiei u este incomplet. În locul urmăririi gradientului pas cu pas, se construieşte un set de n direcţii conjugate între ele, astfel încât minimizarea de-a lungul uneia dintre ele u_j nu se repercutează în mod distructiv asupra minimizării de-a lungul unei direcţii anterioare. Practic minimizările de-a lungul direcţiilor nu interferă. Sistemul de gradul n va evolua deci, spre un minim global. Metoda este diferită de metoda gradientului descendent, ce presupune minimizarea în direcţia cu gradient descendent maxim.

Acceptând aproximarea prin serie Taylor a funcţiei ce urmează a fi minimizată, se obţine:

$$f(x) = f(p) + \sum_i \frac{\partial f}{\partial x_i}|_p x_i + \frac{1}{2}\sum_{i,j}\frac{\partial^2 f}{\partial x_i \partial x_j}|_p x_i x_j + \ldots \cong \frac{1}{2}x^T A x - b^T x + c \tag{2.56}$$

unde

- T denotă operaţiunea de transpunere
- $c = f(p)$
- $b = -\nabla f|_p$
- $[A]_{i,j} \equiv \frac{\partial^2 f}{\partial x_i \partial x_j}|_p$ este o matrice simetrică de dimensiuni $n \times n$, *pozitiv definită* $(\forall y \neq 0, y^T A y > 0)$, de fapt, *Hessiana* funcţiei f în punctul p:

$$A_p = \begin{bmatrix} \frac{\partial^2 f}{\partial x_i^2}(p) & \frac{\partial^2 f}{\partial x_i \partial x_j}(p) \\ \frac{\partial^2 f}{\partial x_i \partial x_j}(p) & \frac{\partial^2 f}{\partial x_j^2}(p) \end{bmatrix} \tag{2.57}$$

Gradientul funcţiei f este:

$$\nabla f = Ax - b \tag{2.58}$$

astfel încât o schimbare în x induce o schimbare de gradient conform relaţiei:

$$\delta(\nabla f) = A(\delta x) \tag{2.59}$$

Presupunând că funcţia f a fost minimizată de-a lungul direcţiei u_i într-un punct în care gradientul $-g_{i+l}$ al funcţiei este perpendicular pe direcţia u_i,

$$u_i^T g_{i+1} = 0 \tag{2.60}$$

se trece la minimizarea pe o nouă direcţie u_{i+1}. Pentru a ne asigura că parcurgerea direcţiei u_{i+1} nu afectează minimizarea de-a lungul direcţiei u_i este necesar ca gradientul funcţiei f să rămână perpendicular pe direcţia u_i, astfel încât:

$$u_i^T g_{i+2} = 0 \tag{2.61}$$

Altfel, va mai trebui parcursă o dată etapa de minimizare în direcţia u_i. Combinând relaţiile 2.60 şi 2.61 se obţine:

$$0 = u_i^T (g_{i+1} - g_{i+2}) = u_i^T \delta(\nabla f) = u_i^T A u_{i+1} \tag{2.62}$$

Când relaţia 2.59 se aplică pentru doi vectori u_i şi u_{i+1}, se spune despre aceştia că sunt conjugaţi.

Plecând de la acelaşi punct p_0, prima direcţie de minimizare u_0 se va lua egală cu $g_0 = -\nabla f(p_0)$. Pentru $i \geq 0$, direcţiile se calculează cu relaţia:

$$u_{i+1} = u_i + \lambda_i^u u_i \tag{2.63}$$

unde valoarea coeficientului incremental λ_i^u este aleasă astfel încât să avem $u_i^T A u_{i-1} = 0$ şi gradienţii succesivi să fie perpendiculari:

$$\gamma_i = \frac{g_{i+1}^T g_{i+1}}{g_i^T g_i} \qquad \text{cu } g(k) = -\nabla f|_{p_k} \qquad \text{pentru } k \geq 0 \tag{2.64}$$

Următorul pas este calculul:

$$p_{i+2} = p_{i+1} + \lambda_{i+1}^p \cdot u_{i+1} \tag{2.65}$$

unde coeficientul λ_{i+1}^p se alege astfel încât să minimizeze $f(p_{i+2})^2$.

Se poate demonstra că direcţiile u astfel construite sunt toate interconjugate [86], [69]. Din figura 2.19 se observă că la o nouă selecţie de direcţie, aceasta se construieşte din prima direcţie şi gradient, rezultând o minimizare în spirală.

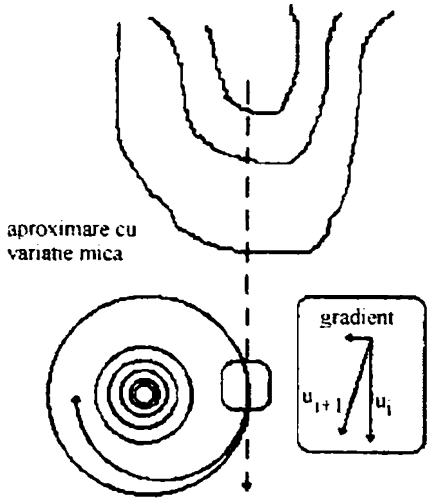


Figura 2.19: *Coborârea cu pantă mică cu gradient conjugat pentru sisteme nepatratice.*

Astfel, pentru un sistem pătratic cu n grade de libertate sunt necesare doar n iteraţii. Având în vedere că nu se urmăreşte minimizarea unui sistem pătratic, este necesară parcurgerea celor n direcţii de mai multe ori.

Powell [69] a introdus câteva îmbunătăţiri semnificative pentru a corecta comportarea metodei în sistemele nepătratice.

Van den Boomgaard şi Smeulders au arătat în 1989 [94] că pentru o reţea neuronală cu propagare înainte fără unităţi ascunse, procedura incrementală de determinare a matricii de ponderi optime necesită o ajustare a ponderilor cu:

$$\Delta W(t+1) = \eta(t+1)(d(t+1) - W(t)x(t+1))x(t+1) \tag{2.66}$$

în care η nu este o constantă, ci o variabilă tip matrice $(N_i + 1) \times (N_i + 1)$ care depinde de vectorul de intrare. Utilizând cunoştinţe *a priori* despre semnalul de intrare, necesităţile de stocare pentru valorile termenului moment γ pot fi reduse substanţial.

Silva şi Almeida au arătat [82] de asemenea avantajele utilizării unui pas de valoare specifică pentru fiecare pondere din reţea. În algoritmul lor, rata de învăţare este adaptată pentru fiecare tipar:

$$\eta_{i,j}(t+1) = \begin{cases} u\eta_{ij}(t) & \text{dacă } \frac{\partial E_{t+1}}{\partial w_{ij}} \text{ şi } \frac{\partial E_t}{\partial w_{ij}} \text{ au acelaşi semn} \\ d\eta_{ij}(t) & \text{dacă } \frac{\partial E_{t+1}}{\partial w_{ij}} \text{ şi } \frac{\partial E_t}{\partial w_{ij}} \text{ au semne diferite} \end{cases} \quad (2.67)$$

unde u şi d sunt constante pozitive cu valori puţin peste — şi respectiv puţin sub unitate. Ideea de bază a acestei metode este de a descreşte rata de învăţare în cazul oscilaţiei valorilor ponderilor.

2.6.1 Parametrii de antrenare a reţelei

În etapa de antrenare a reţelei, conform algoritmului de învăţare, există câţiva parametri care definesc această etapă. Enumerarea doar a unora nu conduce la o descriere satisfăcătoare nici a reţelei şi nici a capacităţii de învăţare a acesteia.

In figura 2.20 este prezentată variaţia funcţiei sigmoide unipolare şi a derivatei pentru valori ale factorului de temperatura β cuprinse între 0.2 şi 2 utilizând un pas de 0.1 unităţi.

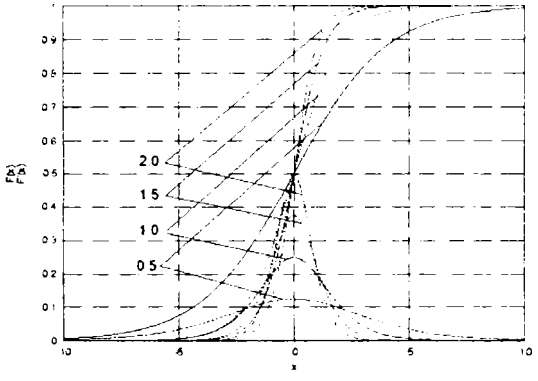


Figura 2.20: *Variaţia funcţiei sigmoidă unipolară şi a derivatei sale pentru valori ale factorului de temperatură β cuprinse între 0.5–2 cu un pas de 0.5.*

Se observă că pentru două valori învecinate ale variabilei, creşterea ambelor funcţii este mai substanţială pentru valori ale factorului de temperatură β mai mari. De aici rezultă două consecinţe:

1. la normalizarea valorilor utilizând funcţia sigmoidă, pentru două valori apropiate ale variabilei, valorile funcţiei sunt progresiv diferite odată cu creşterea factorului de temperatură;
2. având în vedere relaţia 2.55 de corecţie a ponderilor, pentru două valori apropiate ale erorii, corecţia este mai drastică la valori ale factorului de temperatură crescute.

Informaţia despre valoarea absolută a acestui factor nu este relevantă decât dacă este completată cu valorile ratei de învăţare η şi ale termenului moment γ.

Aceste variabile de reţea intervin doar în corecţia ponderilor în vederea reducerii erorii reţelei de la o epocă la alta.

Rata de învăţare este de fapt o constantă de proporţionalitate cu valori între 0 şi 1 cu care se ajustează corecţia ponderii. O valoare a ratei de învăţare de 0.1 înseamnă

practic că se ţine seama doar de 10% din valoarea derivatei funcţiei sigmoidă din termenul de corecţie al ponderii.

Odată caracterizat termenul de corecţie a ponderilor prin valoarea factorului de temperatură β şi prin valoarea ratei de învăţare, în cazul utilizării termenului *moment*, se intervine din nou printr-un proces de ajustare a acestuia conform relaţiei 2.55 în funcţie de viteza de convergenţă a reţelei spre valoarea de minim de eroare.

Importanţa termenului *moment* este discutabilă, deoarece acesta face o ajustare a algoritmului de corecţie, deci o scădere a vitezei de convergenţă spre minim chiar şi atunci când reţeaua se află pe drumul descendent către minimul global. Oricum, valoarea sugerată în literatură [48] $\gamma = 0.9$ este mult prea mare.

Această valoare impune de fapt o corecţie actuală nesemnificativă, accentul fiind pus pe corecţia exercitată într-o epocă anterioară. Aplicarea termenului moment conduce de fapt la o mediere ponderată şi consecutivă a factorului de corecţie a ponderilor.

În concluzie, o comparare a funcţionării unor reţele în etapa de antrenare este incompletă dacă nu sunt prezentaţi *toţi* factorii discutaţi anterior.

2.6.2 Accelerarea convergenţei — rată de învăţare adaptivă

În această secţiune se vor descrie şi analiza câteva metode euristice descrise de Jacobs ca modalităţi de accelerare a convergenţei învăţării prin adaptarea ratei de învăţare [39].

Metoda euristică 1. — Fiecare parametru ajustabil al reţelei care descrie şi funcţia de cost a învăţării trebuie să fie evaluat în particular la o valoare optimă proprie valorii de moment a funcţiei de cost.

Această metodă propune o valoare diferită pentru fiecare pondere, argumentând prin faptul că la un anumit moment, o anumită ajustare poate fi benefică pentru o anumită pondere şi total necorespunzătoare pentru o altă pondere.

Metoda euristică 2. — Valoarea ratei de învăţare trebuie să fie diferită de la o epocă la alta.

Această metodă are în vedere faptul că suprafaţa de eroare diferă de-a lungul regiunii corespunzătoare unei anumite ponderi.

Metoda euristică 3. — Când derivata funcţiei de cost în raport cu o anumită pondere are acelaşi semn pentru câteva iteraţii consecutive ale algoritmului, rata de învăţare pentru acea pondere trebuie crescută.

Metoda are în vedere posibilitatea că punctul de operare curent, în spaţiul ponderii vizate, poate ajunge pe o porţiune plană a suprafeţei de eroare, fapt rezultat din consecvenţa semnului pe parcursul mai multor epoci. Printr-o creştere a valorii ratei de învăţare, numărul de iteraţii pentru traversarea acestei suprafeţe plane va fi mai mic, fapt ce duce la o îmbunătăţire a valorii funcţiei de cost.

Metoda euristică 4. — Când semnul algebric al derivatei funcţiei de cost în raport cu o anumită pondere alternează consecutiv pe parcursul mai multor iteraţii ale algoritmului, rata de învăţare pentru acea pondere trebuie micşorată.

Metoda are în vedere posibilitatea ca punctul de operare curent în spaţiul ponderii vizate să ajungă pe o porţiune foarte curbată a suprafeţei de eroare, fapt rezultat din inconsecvenţa semnului pe parcursul mai multor epoci. Astfel, valoarea ponderii oscilează ca semn, rămânând relativ constantă în valoare absolută ajungându-se la o creştere a funcţiei de cost fără vreun real progres.

Este cert că utilizarea parametrilor de reţea în raport cu fiecare dintre ponderi conform acestor metode euristice duce inevitabil la o modificare fundamentală a algoritmului *backpropagation*, care nu mai corespunde unei căutări în direcţia descendentă spre minim.

Deci ajustările aplicate ponderilor se bazează pe:

- derivatele parţiale ale suprafeţei de eroare în raport cu ponderile;
- estimarea cuadraturii suprafeţei de eroare pe care se situează punctul de operare în spaţiul ponderilor.

Astfel, toate cele patru metode euristice satisfac doar necesităţi locale ale reţelei, fiind benefice pentru anumite puncte şi total necorespunzătoare pentru altele.

2.7 O implementare secvenţială a algoritmului backpropagation

Se prezintă în continuare un exemplu de implementare a unui simulator secvenţial pentru algoritmul backpropagation, realizată în limbajul C.

Pentru implementarea structurii reţelei se utilizează două tipuri de structuri de date, una pentru implementarea unui strat al reţelei – `LAYER`, şi una pentru reprezentarea reţelei:

```
typedef struct {          /* Un strat al reţelei:              */
  int       Units;        /* - numărul de unităti din strat    */
  double*   Output;       /* - ieşirile unităţilor             */
  double*   Error;        /* - eroarea în fiecare unitate      */
  double**  Weight;       /* - ponderea conexiunilor care      */
                          /*   intră în strat                  */
  double**  WeightSave;   /* - ponderi salvate pentru oprirea  */
                          /*   învăţării                       */
  double**  dWeight;      /* - ultimele actualizări de ponderi */
                          /*   pentru termenul moment          */
} LAYER;


typedef struct {          /* O REŢEA:                          */
  LAYER** Layer;          /* - straturile reţelei              */
  LAYER*  InputLayer;     /* - stratul de intrare              */
  LAYER*  OutputLayer;    /* - stratul de ieşire               */
  double  Alpha;          /* - factorul moment                 */
  double  Eta;            /* - rata de învăţare                */
  double  Gain;           /* - câştigul funcţiei sigmoidă      */
  double  Error;          /* - eroarea totală                  */
} NET;
```

În partea de iniţializare trebuie să se stabilească configuraţia reţelei: număr de straturi, număr de unităţi pe fiecare strat, valori pentru factorul moment, rata de învăţare, funcţia sigmoidă – uzual 0.9, 0.25, respectiv 1.

După citirea setului de tipare de antrenare, ponderile conexiunilor între straturi trebuie iniţializate la valori mici aleatoare, în intervalul $(-0.5, 0.5)$.

Dacă NI, NH, şi NO reprezintă numărul de unităţi de pe stratul de intrare, stratul ascuns şi respectiv cel de ieşire, următoarele două ecuaţii dau valoarea activării neuronilor. Pentru un tipar de intrare p, activarea unităţii ascunse j, a_{pj}^H este dată de:

$$a_{pj}^H = f(net_{pj}^H) = f(\sum_{i=0}^{NI-1} a_{pi}^I w_{i\to j}^H) \tag{2.68}$$

unde $w_{i\to j}^H$ este ponderea conexiunii de la neuronul i de pe stratul de intrare la neuronul j de pe stratul ascuns, iar a_{pi}^I este activarea neuronului i de pe stratul de intrare;

similar, pentru o unitate k de pe stratul de ieşire, activarea este dată de:

$$a_{pk}^{O} = f(net_{pk}^{O}) = f(\sum_{j=0}^{NH-1} a_{pj}^{H} w_{j \to k}^{O}) \tag{2.69}$$

unde $w_{j \to k}^{O}$ este ponderea conexiunii de la neuronul j de pe stratul ascuns la neuronul k de pe stratul de ieşire, iar a_{pj}^{H} este activarea neuronului j de pe stratul ascuns.

Funcţiile `PropagateLayer`, respectiv `PropagateNet` implementează pasul înainte conform relaţiilor de mai sus:

```
void PropagateLayer(NET* Net, LAYER* Lower, LAYER* Upper)
{
  int  i,j;
  double Sum;

  for (i=1; i<=Upper->Units; i++) {
    Sum = 0;
    for (j=0; j<=Lower->Units; j++) {
      Sum += Upper->Weight[i][j] * Lower->Output[j];
    }
    Upper->Output[i] = 1 / (1 + exp(-Net->Gain * Sum));
  }
}


void PropagateNet(NET* Net)
{
  int l;

  for (l=0; l<NUM_LAYERS-1; l++) {
    PropagateLayer(Net, Net->Layer[l], Net->Layer[l+1]);
  }
}
```

Apoi se calculează eroarea reţelei, prin comparare cu valoarea corespunzătoare din tiparul curent:

```
void ComputeOutputError(NET* Net, double* Target)
{
  int  i;
  double Out, Err;

  Net->Error = 0;
  for (i=1; i<=Net->OutputLayer->Units; i++) {
    Out = Net->OutputLayer->Output[i];
    Err = Target[i-1]-Out;
    Net->OutputLayer->Error[i] = Net->Gain*Out*(1-Out)*Err;
    Net->Error += 0.5 * sqr(Err);
  }
}
```

După aceea, eroarea este propagată înapoi, conform ecuaţiilor 2.50 şi 2.51:

```
void BackpropagateLayer(NET* Net, LAYER* Upper, LAYER* Lower)
{
  int  i,j;
  double Out, Err;

  for (i=1; i<=Lower->Units; i++) {
```

```
    Out = Lower->Output[i];
    Err = 0;
    for (j=1; j<=Upper->Units; j++) {
      Err += Upper->Weight[j][i] * Upper->Error[j];
    }
    Lower->Error[i] = Net->Gain * Out * (1-Out) * Err;
  }
}


void BackpropagateNet(NET* Net)
{
  int l;

  for (l=NUM_LAYERS-1; l>1; l--) {
    BackpropagateLayer(Net, Net->Layer[l], Net->Layer[l-1]);
  }
}
```

În final, se actualizează ponderile conform ecuaţiei 2.55:

```
void UpdateWeights(NET* Net)
{
  int  l,i,j;
  double Out, Err, dWeight;

  for (l=1; l<NUM_LAYERS; l++) {
    for (i=1; i<=Net->Layer[l]->Units; i++) {
      for (j=0; j<=Net->Layer[l-1]->Units; j++) {
        Out = Net->Layer[l-1]->Output[j];
        Err = Net->Layer[l]->Error[i];
        dWeight = Net->Layer[l]->dWeight[i][j];
        Net->Layer[l]->Weight[i][j] +=
                Net->Eta * Err * Out + Net->Alpha * dWeight;
        Net->Layer[l]->dWeight[i][j] = Net->Eta * Err * Out;
      }
    }
  }
}
```

Antrenarea reţelei constă în prezentarea câte unui tipar din setul de antrenare, propagarea lui înainte prin reţea, propagarea înapoi a erorii, urmată de actualizarea ponderilor. O epocă va consta din prezentarea tuturor tiparelor din setul de antrenare, în ordine aleatoare. După terminarea unei epoci, se testează calitatea învăţării, prezentând din nou toate tiparele, fără antrenare, doar pentru calculul erorii. Dacă eroarea totală este sub o valoare prestabilită, se poate considera că învăţarea a luat sfârşit şi se poate salva configuraţia reţelei.

2.7.1 Analiza simulatorului secvenţial

Algoritmul backpropagation utilizează intens operaţii cu virgulă flotantă. În cele ce urmează se va încerca estimarea timpului necesar prelucrării unui tipar, pentru a se putea face o previzionare a timpului necesar pentru o antrenare.

Pentru experimente, au fost utilizate staţii IBM RS6000, SunBlade, respectiv PC-uri HP-Compaq, pe care s-au făcut determinări ale timpilor necesari efectuării fiecărei operaţii.

Din ecuaţiile 2.68 şi 2.69, rezultă, pentru pasul înainte, că sunt necesare câte o însumare şi câte o înmulţire pentru fiecare pondere, atât pentru stratul ascuns cât şi pentru stratul de ieşire. În plus, s-a determinat că mai sunt necesare 7.16 μsecunde

în cazul staţiilor RS 6000, 3.35 μsecunde pe SunBlade, respectiv 3.86 μsecunde pe procesoarele Intel utilizate, pentru calculul valorii activării fiecărui neuron de pe stratul intermediar şi cel de ieşire, în principal pentru calculul funcţiei exponenţiale.

La pasul înapoi, calculul valorilor delta pentru unităţile de ieşire şi cele ascunse, conform ecuaţiilor 2.50 şi 2.51, vor necesita 2 scăderi şi 2 înmulţiri pentru fiecare unitate de ieşire, respectiv 1 scădere şi 2 înmulţiri pentru fiecare unitate de pe stratul ascuns, plus câte 1 adunare şi 1 înmulţire pentru fiecare conexiune către stratul de ieşire.

La calculul actualizărilor de ponderi, conform ecuaţiei 2.55, algoritmul necesită câte 1 adunare şi 3 înmulţiri pentru fiecare pondere, atât pentru stratul ascuns cât şi pentru cel de ieşire, iar actualizarea ponderilor mai necesită câte o adunare pentru fiecare pondere, pentru ambele straturi. Tabelul 2.1 sintetizează consideraţiile de mai sus, prezentând timpii necesari (în μsecunde) pentru unităţi şi ponderi de conexiuni, pentru cele două platforme utilizate (în cazul neuronilor a fost luat în consideraţie şi timpul necesar calculării activării).

Operaţii	+/-	*	timp RS6000	timp SunBlade	timp Intel
Per unitate ascunsă	1	2	7.502	3.51	4.02
Per unitate de ieşire	2	2	7.566	3.54	4.08
$w^H_{i\to j}$	3	4	0.748	0.35	0.48
$w^O_{j\to k}$	4	5	0.951	0.44	0.53

Tabela 2.1: *Operaţii în virgulă flotantă necesare.*

Pentru exemplificare, pentru o reţea 30-30-30, cu câte 30 de unităţi pe fiecare strat, sunt necesare 930 de ponderi între stratul de intrare şi cel ascuns (900 de ponderi între unităţile de pe cele două straturi plus 30 de ponderi pentru bias către stratul ascuns). Similar, mai sunt necesare 930 de ponderi între stratul ascuns şi cel de ieşire, rezultând un total de 1860 de ponderi în reţea.

Rezultă un total de 6600 de adunări şi 8490 de înmulţiri. Luând în consideraţie şi timpul necesar calculării funcţiei de activare, 7.16 μsecunde pe RS6000, respectiv 3.35 μsecunde pe SunBlade pentru fiecare dintre cei 60 de neuroni de pe straturile ascuns şi de ieşire, rezultă în total un necesar de aproximativ 2.5 milisecunde pe RS6000, respectiv 1.150 milisecunde pe SunBlade, pentru un singur pas înainte şi înapoi.

Pentru un număr de 1000 de tipare de intrare şi circa 1000 de treceri ale fiecărui tipar (în funcţie de rata de învăţare, momentul şi alte caracteristici ale problemei), vor fi necesare 2500 de secunde pe RS6000, respectiv 1150 de secunde pe SunBlade. În plus, orice implementare mai presupune operaţii suplimentare – calcule de indici, salvări de ponderi etc – care la rândul lor mai necesită timp suplimentar, care nu a fost luat în consideraţie în calculele de mai sus.

Există mai multe modalităţi de compensare a acestor dezavantaje ale reţelelor neuronale.

Prima abordare constă în reducerea dimensiunilor problemei prin pre-procesarea datelor de intrare, rezultând astfel fie o reducere a numărului de iteraţii necesare antrenării reţelei, sau chiar reducerea dimensiunilor reţelei. Asemenea reduceeri sunt aproape întotdeauna specifice problemei şi această abordare nu poate fi generalizată în vedera tratării tuturor genurilor de probleme.

O altă posibilitate constă în îmbunătăţirea performanţelor algoritmului back-propagation, fie aplicând modificări ad hoc, fie utilizând rezultate din teoria optimizării numerice. Din această categorie face parte metoda gradientului conjugat [42].

O a treia abordare constă în accelerarea algoritmilor existenţi prin implementarea lor direct în hardware (utilizând tehnici VLSI [89] sau optice [1]) sau modificarea lor pentru execuţie pe o arhitectură paralelă. Acesta este subiectul tratat în această lucrare: paralelizarea algoritmului de învăţare back-propagation.

2.8 Concluzii

Comparând reţelele neuronale artificiale cu cele biologice, trebuie notat faptul că, dintr-o serie de motive, nivelul curent de detaliu în modelarea neuronilor individuali este destul de simplist. Unul dintre motive îl constituie cunoştinţele prezente, întrucâtva limitate, despre fiziologia neuronilor biologici. Un alt motiv important este faptul că nu toate aspectele neuro-fiziologice sunt relevante, dacă principalul scop este atingerea adaptabilităţii creierelor vii, şi nu crearea de sisteme care să modeleze natura cât mai aproape posibil.

Spre deosebire de sistemele de calcul tradiţionale, o reţea neuronală artificială este un sistem de procesare a informaţiei neprogramat, adaptiv, care învaţă din experienţă. Pe parcursul fazei de învăţare i se prezintă un număr de exemple despre cum trebuie să se comporte la un anumit input. Gradual, reţeaua neuronală se adaptează singură, prin încercări. În loc să i se prezinte un set de intrucţiuni care să precizeze cum trebuie îndeplinită o anumită funcţionalitate, reţeaua este capabilă să-şi genereze propriile reguli interne care guvernează asocierea dintre intrare şi ieşire. Aceste reguli sunt construite şi rafinate continuu prin compararea rezultatelor produse de către reţea cu cele găsite în exemple.

O reţea neuronală artificială constă dintr-un set de elemente simple de procesare, numite *unităţi* (uneori mai sunt denumite şi *neuroni*, datorită asocierii cu sistemele nervoase biologice), şi un set de *conexiuni* între aceste unităţi. Activitatea se propagă prin reţea de la o unitate la alta, prin intermediul conexiunilor, fiecare având asociată o *pondere* (sau o *tărie a conexiunii*). Ponderea, care determină mărimea efectului unei unităţi asupra alteia, este de obicei reprezentată printr-un număr real. În funcţie de semnul ponderii, conexiunea va fi fie una *excitatoare*, fie una *inhibitoare*, adică va creşte sau va descreşte activitatea unităţii receptoare. Intrarea fiecărei unităţi din reţea este formată prin combinarea ieşirilor tuturor unităţilor conectate, cu ponderile conexiunilor corespunzătoare. Activitatea fiecărei unităţi este apoi determinată prin aplicarea unei *funcţii de activare* asupra intrării recepţionate şi, posibil, asupra activităţii curente. În final, *funcţia de ieşire* transformă activitatea unităţii într-un semnal de ieşire, care este apoi propagat prin intermediul conexiunilor ca intrare pentru alte unităţi. De obicei, funcţia de ieşire este pur şi simplu funcţia identitate, astfel încât ieşirea unei unităţi este egală cu activitatea ei.

Intrarea dinspre mediul înconjurător este furnizată reţelei prin stimularea unor unităţi speciale, numite *unităţi de intrare* (sau *senzori*). Tiparele de activitate observate la un anumit set de unităţi, numite *unităţi de ieşire*, sunt interpretate ca răspuns al reţelei la intrarea dată.

După cum s-a văzut, răspunsul reţelei la un tipar dat de intrare este determinat de ponderile conexiunilor dintre unităţi. Funcţionalitatea reţelei poate deci să fie modificată prin modificarea acestor ponderi. Ca urmare, tiparul conectivităţii este ceea ce cunoaşte sistemul şi determină răspunsul acestuia la o intrare arbitrară. Însă dacă cunoştinţele sunt reprezentate de tăria conexiunilor, atunci învăţarea trebuie să conste din găsirea unor valori adecvate pentru aceste ponderi. Rezultă că se poate formula o *regulă de învăţare* a reţelei, care să precizeze modul în care trebuie modificate ponderile ca răspuns la ieşiri incorecte sau parţial corecte produse de către reţea.

În secţiunea 2.7 a fost prezentată o implementare secvenţială a algoritmului backpropagation. Calculele arată că pentru o reţea uzuală timpul de antrenare şi necesarul de memorie pot deveni prea mari, putând descuraja investigaţiile în domeniu prin experimente cu diferite configuraţii şi parametri de antrenare. Capitolele următoare încearcă să depăşească aceste neajunsuri, prin realizarea unor implementări paralele ale algoritmului backpropagation.

3 Paralelizarea reţelelor neuronale

3.1 Strategii de paralelizare

Pornind de la paralelismul intrinsec al corespondentului biologic, reţelele neuronale au fost implementate pe o serie de arhitecturi paralele. Pe baza algoritmului backpropagation, au fost identificate mai multe posibilităţi de paralelizare:

- **Paralelizare la nivelul sesiunilor de antrenare:** Procesul de antrenare se face pornind de la parametri de antrenare iniţiali diferiţi pe diferite procesoare.
- **Paralelizare la nivelul setului de antrenare:** Tiparele din setul de antrenare sunt grupate pe diferitele procesoare, fiecare procesor executând aceeaşi reţea neuronală.
- ***Pipelining*:** Straturile reţelei sunt plasate pe procesoare diferite. Procesorul care posedă neuronii de pe stratul de ieşire poate să calculeze valorile de ieşire şi cele de eroare pentru tiparul curent, în vreme ce procesorul care deţine stratul intermediar procesează tiparul următor. Această abordare limitează însă numărul de procesoare care pot fi utilizate în paralel.
- **Paralelizare la nivelul nodurilor:** Un procesor deţine neuroni de pe mai multe straturi.

Prima paralelizare, cea la nivelul sesiunilor de antrenare, reprezintă doar o valoare teoretică, întrucât ea nu aduce avantaje din punctul de vedere al accelerării antrenării şi deci a reducerii timpului de antrenare, decât în faza de determinare a parametrilor pentru care reţeaua prezintă cea mai bună convergenţă.

Fiecare dintre aceste abordări prezintă un anumit grad maxim de paralelizare posibil, dat de diferiţi factori:

- Paralelizarea la nivelul setului de antrenare: numărul de tipare din setul de antrenare.
- Pipelining: Numărul de straturi din reţea.
- Paralelizare la nivelul nodurilor: numărul de neuroni de pe straturi.

Vom prezenta în continuare o serie de realizări din acest domeniu.

3.1.1 Paralelizarea la nivelul setului de antrenare

Această abordare mai poartă şi denumirea de paralelizare cu partiţionarea datelor, întrucât datele de intrare — tiparele de antrenare — se distribuie pe diferite procesoare.

O schemă de distribuire a tiparelor şi de colectare a rezultatelor, într–o topologie în inel, este prezentată pentru maşina GF11 în lucrarea lui Witbrock şi Zagha *"An implementation of backpropagation learning on GF11, a large SIMD parallel computer"* [100]. Maşina IBM GF11 este o maşină experimentală cu 566 procesoare, interconectate într–o reţea specifică. Aproape aceeaşi abordare este utilizată în lucrarea lui Bourrely *"Paralelisation of a neural learning algoritm on a hypercube"* [7], însă pe un hipercub.

O abordare alternativă mai eficientă este de asemenea prezentată în [100]. Facilităţile de comunicare ale maşinii GF11 permit o inter-conectivitate mult mai complexă decât în inel. Aici se descrie cum P procesoare pot colecta şi însuma gradienţii componenţi în $O(log_2 P)$ paşi, într-o configuraţie astfel încât orice procesor q poate comunica cu toate procesoarele $(q + 2^i)$ mod P, pentru $i = 0, \ldots, log_2 P$.

3.1.2 Paralelizarea la nivelul nodurilor

Această abordare mai este cunoscută şi sub denumirea de partiţionarea reţelei, întrucât, spre deosebide de metoda anterioară, de data aceasta se distribuie neuronii din reţea.

Strategia de partiţionare a reţelei în vederea paralelizării a fost investigată în cadrul câtorva cercetări în domeniu.

Chinn şi colegii, în [12] prezintă o asemenea implementare pe maşina MasPar MP–1 — un calculator masiv paralel care poate conţine un array de 128x128 de procesoare. Pe aceeaşi maşină, mai poate fi menţionată şi lucrarea lui d'Acierno si Vaccaro [16], care combină paralelizarea la nivelul nodurilor cu cea la nivelul setului de antrenare.

Abordări similare se pot întâlni şi în lucrările lui Ernoult – *"Performance of backpropagation on a parallel transputer-based machine"*, [22], del Rey Millan şi Bofill – *"Learning by backpropagation: Computing in a systolic way"*, [20], şi Zhang, McKenna, Mesirov şi Waltz – *"The backpropagation algorithm on grid and hypercube architectures"*, [104].

După cum se observă din lucrările enumerate, implementările paralele au fost realizate pe maşini dedicate, cu număr mare de procesoare şi o arhitectură deosebită a comunicaţiei. Analizând dotarea laboratoarelor de calculatoare din universităţi şi institute de cercetare, am identificat un potenţial ridicat de calcul paralel şi distribuit deja existent — un număr de calculatoare secvenţiale independente (PCuri) interconectate într–o reţea ethernet. Investigaţiile noastre se îndreaptă spre cercetarea posibilităţilor de utilizare în vederea antrenării cu algoritmul backpropagation, a unei asemenea reţele de calculatoare, fie grupate într–un cluster eterogen, fie grupate într–un cluster Beowulf [85], pe baza instrumentelor software disponibile. Posibilitatea implementării de algoritmi paraleli pe asemenea reţele de calculatoare este prezentată şi de Polze şi Malek în [67], iar Suresh şi colegii, în [88], prezintă o asemenea paralelizare a algoritmului backpropagation pe un set de staţii de lucru. Implementarea acestora realizează paralelizarea într–o abordare hibridă, la nivelul setului de antrenare *şi* la nivelul nodurilor. Abordarea lor, însă, prezintă doar posibilitatea unei asemenea implementări şi nu a permis investigarea performanţelor şi a factorilor care le pot afecta. Această teză propune abordări distincte pentru cele două moduri de paralelizare, şi realizarea unei serii de experimente în vederea identificării factorilor care ar putea influenţa performanţele fiecărei metode utilizate.

3.2 Direcţii de acţiune

La transformarea unui algoritm secvenţial într-unul paralel trebuie luate în considerare cel puţin două strategii. În contextul reţelelor neuronale, cele două strategii sunt cele numite în mod obişnuit *partiţionarea datelor*, respectiv *partiţionarea reţelei*.

3.2.1 Partiţionarea datelor

Dacă acelaşi algoritm se aplică de un număr mare de ori pe diferite seturi de date, atunci este foarte eficient să se execute aceste aplicaţii concurent, pe diferite procesoare, cu condiţia ca task-urile să fie complet independente, adică execuţia unui task să nu depindă de rezultatele altor task-uri. Această strategie de paralelizare este uneori denumită paralelism *la nivel de job* [25] sau *partiţionarea datelor* [68].

Dacă actualizarea ponderilor dintr-o reţea neuronală se face doar după prezentarea mai multor tipare, fiecare dintre aceste prezentări de tipare este un task independent, care se poate executa concurent. Ca urmare, dacă la antrenarea unei reţele neuronale actualizarea ponderilor se face în mod *batch* sau după o epocă, atunci abordarea prin partiţionarea datelor se aplică foarte uşor: datele de antrenare sunt distribuite uniform procesoarelor disponibile, fiecare simulând întreaga reţea, dar pe

sub-seturi diferite de date de antrenare. Apoi rezultatele prezentării diferitelor grupuri de tipare sunt combinate şi se face actualizarea ponderilor.

Aplicarea strategiei de partiţionare a datelor la reţele neuronale mai este denumită şi *paralelizarea antrenării* [20], din motive evidente.

Trebuie subliniat aici faptul că partiţionarea datelor nu este posibilă dacă ponderile sunt actualizate după prezentarea fiecărui tipar. În acest caz, starea reţelei se modifică în urma fiecărei prezentări de tipar. Ca urmare, rezultatul prezentării unui tipar depinde de rezultatele *tuturor* tiparelor prezentate anterior, ceea ce înseamnă că task-urile prezentării tiparelor individuale nu mai pot fi considerate independente.

În capitolul 4 se vor descrie şi analiza proprietăţile unei paralelizări care exploatează strategia de partiţionare a datelor.

3.2.2 Partiţionarea reţelei

Un alt mod de paralelizare este utilizarea paralelismului aşa-numit *geometric* [25] sau *spaţial* [20]. În această abordare se paralelizează execuţia algoritmului pe *un* set de date. Aceasta se realizează prin distribuirea datelor către procesoare astfel încât toate datele necesare unui procesor sunt stocate pe acel procesor sau sunt uşor accesibile la nevoie de la unul din procesoarele vecine.

Aplicarea acestei strategii la reţele neuronale mai este denumită şi *partiţionarea reţelei* [68], întrucât procesarea unui tipar poate fi paralelizată prin divizarea reţelei, fiecare procesor având de prelucrat o mică porţiune din reţea.

În capitolul 5 se prezintă o discuţie a diferitelor moduri în care reţeaua poate fi divizată, precum şi o descriere detaliată a unei implementări a strategiei de partiţionare a reţelei pentru paralelizarea reţelelor neuronale.

Cele două strategii conduc la aplicaţii între care se pot identifica o serie de diferenţe semnificative:

1. Varianta cu partiţionarea datelor:
   - prezintă o oarecare limitare în privinţadimensiunii reţelei neuronale care poate fi simulată, limitare generată de volumul de memorie disponibilă.
   - permite doar actualizarea ponderilor la epocă, după prezentarea unui număr de tipare.
2. Varianta cu partiţionarea reţelei:
   - permite simularea unor reţele de dimensiuni foarte mari, mai mari decât cele care pot fi procesate într-un algoritm secvenţial.
   - are o granularitate mai fină în privinţa actualizării ponderilor, care se poate face după fiecare tipar. Pentru unele aplicaţii de reţele neuronale această proprietate poate fi esenţială pentru calitatea antrenării (vezi aplicaţia NETTalk, secţiunea 6.1)

3.3 Analiza algoritmilor paraleli

Vor fi introduse aici o serie de concepte, utile în analiza performanţelor algoritmilor paraleli pentru reţele neuronale. Principalul obiectiv constă în cât mai buna exploatare a resurselor procesoarelor. Gradul de exploatare este denumit *eficienţă*. La analiza unui algoritm paralel ne interesează modul în care este influenţată eficienţa algoritmului prin modificarea unui număr de parametri. Asemenea parametri sunt cei specifici reţelei neuronale, cum ar fi numărul de unităţi de pe fiecare strat, numărul de ponderi, frecvenţa de actualizare a acestor ponderi etc. Alţi parametri ţin însăşi de paralelizare, cel mai important dintre aceştia fiind numărul de procesoare utilizate şi modul de comunicare între ele.

Pentru a putea formula o definiţie a eficienţei trebuie cunoscută o limită în privinţa a cât de rapid ar putea să fie un algoritm paralel care utilizează un număr de P procesoare. Pentru aceasta, va fi introdus un alt concept uzual, denumit *accelerare* ("speed-up") [26]. Întrucât obiectivul principal al oricărei paralelizări este reducerea timpului de execuţie, un mod natural de măsurare a performanţei unui algoritm paralel este compararea directă a timpului de execuţie pentru o problemă dată, cu timpul de execuţie al algoritmului secvenţial corespondent, pentru a determina cu cât este mai rapid algoritmul paralel.

Formal, se poate spune că, pentru un algoritm paralel, pentru o problemă dată, *accelerarea* este raportul dintre timpul de execuţie T_{seq} al algoritmului secvenţial şi timpul de execuţie T_{par} al algoritmului paralel, când ambii algoritmi sunt aplicaţi aceleiaşi probleme:

$$S(P) =_{def} \frac{T_{seq}}{T_{par}(P)} \tag{3.1}$$

unde P este numărul de procesoare utilizate la execuţia algoritmului paralel. În general, algoritmul secvenţial utilizat va fi *cel mai rapid cunoscut*. Totuşi, pentru a putea utiliza această măsură a *accelerării* în vederea evaluării succesului în paralelizarea unui algoritm dat, se vor mai impune o serie de restricţii asupra modului în care se obţine T_{seq}. Ca urmare, algoritmul secvenţial trebuie să fie implementat în acelaşi limbaj de programare ca şi versiunea paralelă, iar procesorul pe care se execută algoritmul secvenţial ar trebui să fie identic cu procesoarele utilizate la execuţia algoritmului paralel. Întrucât în studiul de faţă s-a propus să se utilizeze pentru varianta paralelă o reţea de calculatoare, în care pot coexista mai multe platforme şi implicit procesoare diferite, se va lua în consideraţie pentru T_{seq} valoarea cea mai mică, adică cea obţinută pe maşina cea mai performantă.

Pe de altă parte, ambii algoritmi, şi cel secvenţial, şi cel paralel, trebuie executaţi pe exact acelaşi set de date, având aceiaşi parametri specifici reţelei neuronale, inclusiv frecvenţa actualizărilor.

Orice paralelizare a algoritmului backpropagation va fi obligatoriu să conţină toate calculele din algoritmul secvenţial. Ca urmare, dacă singurul aspect al unui algoritm paralel pentru reţele neuronale este faptul că acum calculele se pot efectua concurent, atunci este evident că *accelerarea* $S(P)$ al unui algoritm paralel pe P procesoare este limitată de valoarea lui P. Având de P ori mai multe resurse de calcul, în cel mai bun caz este de aşteptat o reducere a timpului de execuţie de P ori.

Totuşi, această valoare nu va putea fi atinsă în cazul de faţă, când execuţia se face într-o reţea cu platforme diferite şi deci cu performanţe diferite.

Se poate beneficia însă de reduceri ale timpului de execuţie datorită altor factori, cum ar fi parametrii specifici hardware-ului. Un asemenea exemplu este cazul memoriei *cache* existentă pe fiecare procesor, şi care este foarte rapidă. Dacă în cazul algoritmului secvenţial aceasta nu este suficient de mare pentru a stoca toate ponderile dintr-o reţea, în cazul algoritmului paralel s-ar putea realiza o asemenea partiţionare încât ea să conţină toate ponderile necesare pentru acea porţiune din reţea pe care o are de prelucrat *un* procesor. Având P procesoare, s-ar putea deci obţine un *speed-up* chiar mai mare decât P (!?) În studiul de faţă nu este luat în calcul acest fenomen.

Pe baza definiţiilor de mai sus, se poate acum exprima eficienţa ca raportul dintre *speed-up*-ul obţinut experimental şi cel optim. Dacă se presupune că *speed-up*-ul se datorează numai efectelor calculelor concurente, adică $S(P)$ are limita P, se propune următoarea definiţie formală a eficienţei:

$$E(P) =_{def} \frac{S(P)}{P} = \frac{T_{seq}}{T_{par} \cdot P} \tag{3.2}$$

După cum se observă, cu presupunerile făcute, $E(P)$ are valori cuprinse în intervalul (0,1].

3.4 Principalele obiective ale paralelizării

Cel mai important motiv pentru paralelizarea unui algoritm este de obicei necesitatea de a obţine o reducere a timpului de execuţie pentru acel algoritm. O asemenea reducere a timpului în procesarea unui set de date este absolut necesară, întrucât nu numai că permite prelucrarea mai multor task-uri de aceeaşi dimensiune, dar şi face posibilă prelucrarea unor task-uri mai mari din punct de vedere computaţional, după cum arată şi Fox, Johnson, Lyzenga, Otto, Salmon şi Walker în *Solving Problems On Concurrent Processors* [26].

De asemenea, pot fi prelucrate task-uri mai mari în privinţa necesarului de memorie (de exemplu, probleme cu seturi de date mai mari), întrucât paralelizarea permite distribuirea datelor pe care lucrează algoritmul, reducând astfel necesarul de memorie al procesoarelor individuale. Ca urmare, necesarul total de memorie pentru un algoritm paralel este de preferat să nu fie mai mare decât cel al algoritmului secvenţial.

Dacă se poate proiecta o paralelizare eficientă a unui algoritm, acesta este un mod simplu şi economic de accelerare a execuţiei unui algoritm. Presupunând că se păstrează eficienţa algoritmului paralel chiar şi atunci când se utilizează un număr mare de procesoare, un sistem de procesoare care lucrează în paralel va putea să se comporte asemenea unui singur calculator, mai puternic. Şi mai mult, un sistem paralel este uşor de extins, astfel încât se poate câştiga în viteză adăugând procesoare.

Totuşi, în general, paralelizarea unui algoritm nu constituie o sarcină uşoară. Vor trebui luate în consideraţie o serie de aspecte, inclusiv proprietăţile hardware-ului disponibil. În cazul abordat în lucrarea de faţă, când fiecare procesor are memoria proprie şi este fără acces direct la memoria altor procesoare, este important de cunoscut cât de rapidă este comunicaţia între procesoare în comparaţie cu performanţele computaţionale ale fiecărui procesor. Interesează în acest caz, când transferul de date şi calculul se pot efectua concurent, să se obţină timpi de comunicaţie mult mai reduşi decât cei pentru calcule.

Pe de altă parte, se va căuta să se asigure o încărcare cât mai uniformă a procesoarelor, întrucât aceasta va asigura timpul de execuţie cel mai scurt.

3.5 Surse de ineficienţă într-un algoritm paralel

Există o serie de motive pentru care un algoritm paralel poate să nu utilizeze resursele computaţionale la fel de eficient ca şi algoritmul secvenţial corespunzător. Atunci când se observă o scădere a eficienţei într-un anumit algoritm, cel mai probabil, aceasta va fi rezultatul câtorva dintre cauzele enumerate mai jos. În cadrul acestei lucrări se va analiza în ce măsură aceste cauze se aplică fiecărui algoritm în parte, în secţiunea corespunzătoare.

3.5.1 Software adiţional

În vederea procesării datelor provenind de la alte procesoare, poate fi necesar să se introducă o serie de calcule suplimentare, de exemplu pentru indexare, sau chiar mai complicate. De asemenea, uneori este necesară modificarea secvenţei calculelor, din diferite motive (vezi mai jos), astfel încât este posibil ca unele rezultate temporare să nu mai fie disponibile şi deci să necesite recalcularea. Orice asemenea sarcini suplimentare vor reduce eficienţa algoritmului paralel.

Totuşi, dacă software-ul suplimentar constituie o fracţiune constantă din sarcinile fiecărui procesor, indiferent de numărul de procesoare care sunt utilizate, atunci cantitatea totală de efort computaţional care corespunde software-ului adiţional este independentă de numărul de procesoare. Întrucât această sarcină trebuie efectuată în paralel, eficienţa algoritmului se reduce întotdeauna cu acelaşi factor constant,

indiferent de numărul de procesoare utilizate în execuţia algoritmului:

$$E(P) = \frac{T_{seq}}{T_{par}(P) \cdot P} = \frac{T_{seq}}{\frac{T_{seq}+T_{soft}}{P} \cdot P} = \frac{T_{seq}}{T_{seq}+T_{soft}} \tag{3.3}$$

În formula de mai sus s-a presupus că singura cauză a ineficienţei algoritmului paralel este software-ul adiţional, independent de numărul procesoarelor. T_{soft} este timpul necesar unui procesor pentru execuţia calculelor suplimentare.

Întrucât în formula finală din relaţia 3.3 nu mai apare numărul de procesoare utilizate, eficienţa este redusă printr-un factor *constant*, T_{soft}. Acest tip de software adiţional nu poate să impună o limită a accelerării care poate fi obţinută pentru o problemă de reţele neuronale dată. Prezenţa acestui software duce la o mai slabă utilizare a fiecărui procesor individual implicat în simularea unei reţele neuronale date.

Pe de altă parte, este posibil ca acest software suplimentar să ducă la creşterea volumului de muncă — pentru fiecare procesor şi pentru sistem în ansamblu — odată cu creşterea numărului de procesoare. Evident că în acest caz şi eficienţa se va deteriora.

3.5.2 Echilibrarea încărcării

O reţea de maşini lucrând concurent nu a terminat un task decât atunci când *toate* procesoarele şi-au terminat sarcinile. Ca urmare, este important ca încărcarea să fie distribuită uniform pe toate procesoarele, astfel încât fiecare maşină să efectueze aceeaşi cantitate de muncă. Mai mult, încărcarea trebuie echilibrată, uniform, *pe tot parcursul* execuţiei algoritmului, altfel procesoarele vor alterna stările de lucru, respectiv de aşteptare, într-un asemenea mod încât chiar dacă au de efectuat aceeaşi cantitate de muncă, ele nu o vor termina în paralel.

Este posibil ca problemele de echilibrare a încărcării să devină mai pronunţate odată cu creşterea numărului de procesoare, funcţie de configuraţia utilizată şi de natura problemei de reţele neuronale de tratat.

3.5.3 Comunicaţia suplimentară

Orice cantitate de timp utilizată pentru comunicaţie va reduce eficienţa algoritmului, întrucât este vorba despre operaţii suplimentare, faţă de algoritmul secvenţial respectiv. Ca urmare, se va încerca minimizarea comunicaţiei între procesoare, cel puţin dacă acest transfer de date nu poate avea loc în paralel cu alte calcule.

Un aspect important care trebuie luat în considerare în construcţia algoritmului paralel este rearanjarea secvenţei de calcule astfel încât să se asigure cel mai înalt grad posibil de paralelism între comunicaţie şi calcule. Uneori, totuşi, este posibil să nu mai fie de efectuat calcule care să nu depindă de datele comunicate până în acel moment. De asemenea, trebuie remarcat faptul că deşi costul comunicaţiei poate fi redus semnificativ prin efectuarea în paralel cu calculele, acest cost nu va ajunge niciodată neglijabil.

În câteva aplicaţii, cantitatea de date care se transmit de către fiecare procesor nu depinde de numărul de procesoare implicate. Aceasta înseamnă că timpul de transmisie pentru fiecare procesor nu se reduce la creşterea numărului de procesoare. Din moment ce fracţiunea alocată fiecărui procesor dintr-o problemă de reţele neuronale devine din ce în ce mai mică, odată cu creşterea numărului de procesoare utilizate pentru simulare, aceasta determină ca o fracţiune tot mai mare din timp să fie alocată comunicării, ducând astfel la scăderea eficienţei.

Pentru a demonstra aceasta, se presupune că pentru un algoritm paralel oarecare singura cauză pentru care eficienţa nu este cea optimă este transmisia, adică totul, în afara comunicaţiei, este perfect paralelizat. Mai mult, presupunând că timpul utilizat în fiecare procesor pentru transmisie depinde numai de dimensiunea problemei

de reţele neuronale, şi deci este independent de numărul de procesoare, se va obţine

$$S(P) = \frac{T_{seq}}{T_{par}(P)} = \frac{T_{seq}}{\frac{T_{seq}}{P} + T_{comm}} \tag{3.4}$$

De remarcat că dacă T_{comm} — timpul consumat în fiecare procesor pentru comunicaţie — este nul, atunci *speed-up*-ul este cel optim. Pe de altă parte, dacă T_{comm} nu este nul, transmisia va impune o limită superioară pentru *speed-up*, $\frac{T_{seq}}{T_{comm}}$.

În general, aceleaşi consideraţii legate de efectele software-ului adiţional se aplică şi la transmisie. Dacă timpul de transmisie nu se reduce la creşterea numărului de procesoare, atunci aceasta va duce la deteriorarea eficienţei la mărirea numărului de procesoare.

3.5.4 Porţiuni inerent secvenţiale

Pot exista porţiuni din algoritm care sunt inerent secvenţiale, adică pur şi simplu nu pot fi paralelizate, de exemplu datorită faptului că execuţia câte unui pas în algoritm depinde de finalizarea pasului anterior. În contextul reţelelor neuronale, asemenea porţiuni inerent secvenţiale pot fi adesea găsite în distribuţia iniţială a ponderilor sau a tiparelor de antrenare, ca şi în colectarea finală a rezultatelor parţiale de la procesoarele individuale. Un alt exemplu ar putea fi calculul produsului scalar global (ca în cazul algoritmului gradientului conjugat [42]). Trebuie menţionat faptul că o porţiune inerent secvenţială a algoritmului uneori poate fi totuşi paralelizată, în sensul că toate procesoarele efectuează aceleaşi calcule, fiecare procesor calculându-şi singur rezultatele. Aceasta însă nu constituie o reală paralelizare, din moment ce timpul de rulare necesar obţinerii rezultatului nu se reduce în comparaţie cu algoritmul secvenţial. Totuşi, ar putea fi o soluţie mai elegantă (şi mai eficientă) decât varianta în care un singur procesor efectuează calculele, urmând apoi un *broadcast* pentru a le transmite celorlalte procesoare.

Dacă un algoritm conţine porţiuni inerent secvenţiale, timpul necesar executării acestor porţiuni va determina o limitare a *speed-up*-ului posibil de obţinut, independent de numărul de procesoare utilizate. Acest fenomen este cunoscut ca *legea lui Amdahl* [26]. Aceasta spune că dacă o porţiune inerent secvenţială a algoritmului necesită $\frac{1}{\alpha}$ din timpul de execuţie atunci când algoritmul se execută pe *un* procesor, atunci nu este posibil să obţinem un *speed-up* mai mare decât α. Pe măsură ce adăugăm procesoare, timpul necesar execuţiei porţiunii secvenţiale va reprezenta o fracţiune tot mai mare din timpul total de execuţie, reducând astfel eficienţa.

3.5.5 Limitări specifice problemei

Atâta vreme cât doar o singură strategie de paralelizare este utilizată, întotdeauna există o limită teoretică a *speed-up*-ului care poate fi atins pentru o problemă de dimensiune dată.

Orice paralelizare constă în divizarea problemei într-un număr de sub-probleme, care pot fi distribuite pe procesoarele disponibile. Diferitele strategii de paralelizare divid problema în moduri diferite — se poate spune de dimensiuni diferite. Pentru fiecare dimensiune/strategie de paralelizare, există o limită a numărului de subprobleme în care poate fi descompusă problema originală. Acest număr de sub-probleme determină o limită superioară a numărului de procesoare care pot fi utilizate eficient în algoritmul paralel, întrucât fiecare procesor va trata cel puţin o singură sub-problemă.

Ca exemplu, se poate lua în consideraţie o implementare simplă a strategiei de partiţionare a datelor pentru paralelizarea reţelelor neuronale. În contextul unui asemenea algoritm, prezentarea unui tipar din setul curent de tipare de antrenare reprezintă o sub-problemă, care poate fi rezolvată de către un singur procesor. Totuşi, în acest caz problema iniţială nu poate fi divizată în mai multe asemenea sub-probleme decât numărul de tipare din setul de antrenare, întrucât fiecare procesor trebuie să

trateze cel puţin un asemenea tipar (sau nu va face nimic altceva decât să "deranjeze" celelalte procesoare care "au de lucru"). Cu alte cuvinte, în abordarea partiţionării datelor nu se poate obţine un *speed-up* mai mare decât dimensiunea setului de antrenare utilizat.

Similar, se observă că într-o implementare a partiţionării reţelei nu se pot utiliza mai multe procesoare decât numărul de unităţi din reţea, limitându-se astfel *speed-up*-ul care poate fi atins la numărul de unităţi din reţea.

De notat totuşi că limitările de mai sus sunt mai mult teoretice, întrucât de obicei *speed-up*-ul este mult mai limitat de efectele unui număr de alte cauze, mai ales software-ul suplimentar şi transmisia de date.

3.6 Consideraţii specifice paralelizării reţelelor neuronale

În general, la paralelizarea unui algoritm, se consideră că singura diferenţă faţă de algoritmul secvenţial va consta în viteza cu care se execută cei doi algoritmi. Cu alte cuvinte, cei doi algoritmi vor fi identici din punct de vedere funcţional, adică vor produce acelaşi *output*.

În contextul reţelelor neuronale, aceasta înseamnă că prin paralelizarea algoritmului se va păstra calitatea învăţării, în privinţa puterii de generalizare şi a vitezei de învăţare (având ca măsură numărul de prezentări de tipare de învăţare). Totuşi, deşi ar putea suna ciudat, la algoritmii de învăţare paralelizaţi nu se întâmplă întotdeauna aşa.

Problema apare datorită modificării frecvenţei cu care sunt actualizate ponderile conexiunilor din reţeaua neuronală. Calitatea învăţării este puternic influenţată de numărul de tipare prezentate reţelei între două actualizări (adică de dimensiunea unui set de antrenare). Desigur, algoritmul *backpropagation* secvenţial poate utiliza orice frecvenţă de actualizare a ponderilor, potrivită problemei tratate. Totuşi, în cazul paralelizării (în special în cazul partiţionării datelor), adesea ponderile nu sunt actualizate cu frecvenţa cea mai adecvată problemei date. Pentru o paralelizare eficientă, se alege un alt număr de prezentări de tipare între două actualizări de ponderi. Ca rezultat, dimensiunea unui set de învăţare depinde de obicei de numărul de procesoare implicate în execuţia algoritmului paralel.

Algoritmii de paralelizare pentru reţele neuronale sunt paralelizaţi din dorinţa de a mări viteza (măsurată în timp absolut) cu care reţeaua poate fi antrenată pentru a efectua o sarcină dată, cu un grad dat de perfecţiune. Ca urmare, ca o consecinţă a efectului descris, de modificare a dimensiunii unui set de învăţare, modul corect de comparare a diferiţilor algoritmi paraleli este *nu* prin măsurarea numărului de tipare care pot fi prezentate reţelei pe secundă, întrucât aceasta nu ia în consideraţie nici o diferenţă în efectul asupra învăţării a fiecărei prezentări de tipar. Ideal ar fi să se compare algoritmii paraleli prin măsurarea timpului în care fiecare algoritm poate să antreneze o reţea neuronală dată pentru a efectua o anumită sarcină, cu un grad dat de perfecţiune. Oricum, efectul dimensiunii setului de antrenare asupra calităţii învăţării diferă de la aplicaţie la aplicaţie şi deci dacă doi algoritmi paraleli utilizează seturi de învăţare de dimensiuni diferite, evaluarea lor va depinde de problema luată ca etalon.

Concluzia la cele de mai sus este că, pentru a realiza într-adevăr o comparaţie corectă a doi algoritmi paraleli — rulaţi sau nu pe acelaşi sistem de procesoare — ei trebuie aplicaţi aceleiaşi probleme, având toţi parametrii identici, mai ales dimensiunea setului de învăţare.

3.7 Experimente în vederea analizei

performanţelor

În general, în cadrul tezei de faţă s-a urmărit efectul tuturor tipurilor de parametri asupra eficienţei algoritmului paralel, fie că este vorba despre parametri specifici reţelei neuronale, cum ar fi dimensiunea setului de antrenare, fie că este vorba despre parametri specifici hardware-ului, cum ar fi structura configuraţiei de procesoare sau numărul de procesoare utilizate în execuţia algoritmului paralel.

După cum s-a menţionat mai devreme, prin paralelizarea unui algoritm se urmăresc două obiective principale: reducerea timpului necesar aplicării algoritmului pentru o problemă dată, respectiv creşterea dimensiunii problemelor care pot fi tratate.

3.7.1 Probleme de dimensiune fixă

Pentru un număr de probleme, se vor prezenta efectele creşterii numărului de procesoare utilizate în execuţia algoritmului paralel. Dacă fracţiunea de timp necesitată de fiecare procesor este proporţională cu numărul de sub-probleme tratate de către acel procesor, atunci graficele *speed-up*-ului vor fi liniare, ceea ce indică faptul că *speed-up* este liniar cu numărul de procesoare (cât timp avem mai puţine procesoare decât numărul de sub-probleme), iar graficul eficienţei va fi o dreaptă orizontală, ceea ce ne arată că eficienţa este independentă de numărul de procesoare utilizate.

Dacă însă fracţiunea de timp necesară fiecărui procesor creşte odată cu numărul de procesoare, eficienţa se va deteriora pe măsură ce adăugăm procesoare în sistem.

Pentru fiecare experiment, se va încerca să se interpreteze graficele, în scopul de a determina care din circumstanţele menţionate anterior joacă cel mai important rol în reducerea eficienţei algoritmului în cauză.

3.7.2 Probleme de dimensiune variabilă

De asemenea, se va examina relaţia dintre dimensionalitatea problemei de reţele neuronale şi numărul de procesoare, pentru a determina care este dimensiunea maximă a problemelor care pot fi tratate (cu acelaşi grad de eficienţă) când se utilizează mai multe procesoare. Ideal ar fi să se utilizeze de două ori mai multe procesoare atunci când se dublează dimensiunea problemei, însă, după cum se va vedea, este dificil să se măsoare dimensiunea unei probleme de reţele neuronale.

Mai precis, *dimensiunea* unei probleme este dată de cantitatea de efort computaţional asociat rezolvării problemei (utilizând algoritmul secvenţial). În domeniul reţelelor neuronale, *rezolvarea problemei* înseamnă antrenarea reţelei până la atingerea unui grad dat de perfecţiune.

De aceea, de obicei este foarte dificil să se compare dimensiunile pentru două probleme de reţele neuronale, întrucât adăugarea de unităţi în reţea sau mărirea setului de antrenare pot sau nu să modifice numărul de cicluri de antrenare necesare atingerii unui nivel dat de perfecţiune, în funcţie de natura antrenării dorite.

Pentru un tip dat de antrenare, nu se poate doar să se presupună că o reţea conţinând un număr dublu de unităţi ascunse reprezintă o problemă (aproximativ) de două ori mai mare[1], întrucât numărul de cicluri de antrenare necesare atingerii unei performanţe date poate fi mare atât în cazul în care avem prea puţine unităţi ascunse, cât şi atunci când ele sunt prea multe.

Similar, creşterea dimensiunii setului de antrenare poate conduce la scăderea performanţei reţelei în privinţa numărului de cicluri de antrenare necesare atingerii unui nivel dat de perfecţiune. Dublarea dimensiunii setului de antrenare poate mări dimensiunea computaţională a problemei cu un factor mult mai mare de 2. Sau, în cazul altor aplicaţii, este posibil să reducă numărul de cicluri de antrenare necesare producerii unei soluţii, reducând chiar dimensiunea problemei.

[1] La dublarea numărului de unităţi ascunse, se dublează şi numărul de ponderi din reţea. Totuşi, întrucât numărul de unităţi de ieşire este acelaşi, efortul de calcul la fiecare ciclu de antrenare va fi doar *aproximativ* dublat.

Datorită dificultăţilor enumerate mai sus, nu se va încerca să se măsoare efectele scalării dimensiunii problemei cu numărul de procesoare asupra eficienţei. În schimb, se va examina influenţa asupra eficienţei a variaţiei altor parametri, ca dimensiunea reţelei şi dimensiunea setului de antrenare.

3.7.2.1 Număr variabil de sub-probleme

În loc să se studieze eficienţa atunci când *dimensiunea* problemei este scalată cu numărul de procesoare, se va examina gradul în care se păstrează eficienţa atunci când numărul de *sub-probleme* este scalat cu numărul de procesoare. Cu alte cuvinte, se va verifica dacă se pot utiliza mai multe procesoare, fără a reduce din eficienţă, atât timp cât fiecare procesor prelucrează acelaşi număr de sub-probleme.

După cum s-a precizat, chiar dacă o problemă nouă poate fi divizată într-un număr dublu de sub-probleme, aceasta nu implică neapărat dublarea dimensiunii problemei. Într-o paralelizare bazată pe partiţionarea reţelei, numărul de sub-probleme este dublat prin dublarea numărului de unităţi din fiecare strat al reţelei. Nu se poate preciza efectul asupra dimensiunii problemei, întrucât acesta depinde de aplicaţia de reţele neuronale. Însă ceea ce se cunoaşte este faptul că dublarea numărului de unităţi înseamnă mărirea numărului de ponderi de 4 ori, conducând la mărirea de aproape 4 ori a efortului computaţional la prezentarea unui tipar.

În cazul paralelizării prin partiţionarea datelor, dublarea numărului de sub-probleme se obţine prin dublarea dimensiunii setului de antrenare. Din nou, influenţa asupra dimensiunii problemei depinde de aplicaţia de reţele neuronale tratată. Se poate afirma, totuşi, că, întrucât se dublează numărul de prezentări de tipare dintre două actualizări de ponderi, volumul operaţiilor asociate unui singur ciclu de antrenare se va dubla la rândul său[2].

Ca urmare, se poate afirma că, dintr-un punct de vedere, având o problemă nouă, care conţine un număr mai mare de sub-probleme, nu este vorba atât de o problemă mai mare, cât de pur şi simplu o problemă *diferită*, cu proprietăţi diferite.

Pentru câteva seturi de asemenea numere fixe de sub-probleme per procesor, se va studia efectul măririi numărului de procesoare. La fel ca în cazul experimentelor cu probleme de dimensiune fixă, se va încerca să se interpreteze graficele *speed-up*-ului şi eficienţei rezultate. Dacă graficele eficienţei nu sunt drepte orizontale, se poate concluziona că activitatea suplimentară din fiecare procesor nu este independentă de numărul de procesoare utilizate la execuţia algoritmului.

3.7.2.2 Număr fix de sub-probleme

Se vor examina efectele modificării de parametri specifici reţelelor neuronale care nu influenţează numărul de sub-probleme, întrucât cunoaşterea acestor efecte este necesară cercetătorului în reţele neuronale, cu scopul alegerii celei mai potrivite paralelizări pentru aplicaţia dată. Ca urmare, se va examina, de asemenea, efectul modificării dimensiunii reţelei în cazul paralelizării prin partiţionarea datelor, precum şi influenţele asupra eficienţei a modificării dimensiunii setului de antrenare în cazul abordării cu partiţionarea reţelei.

3.8 Concluzii

Acest capitol realizează o investigaţie asupra diverselor abordări în paralelizarea algoritmului backpropagation. Au fost identificate două strategii majore: cea cu partiţionarea datelor şi cea cu partiţionarea reţelei. În urma acestor investigaţii se evidenţiază o serie de aspecte care ar putea afecta performanţele algoritmilor care urmează a fi prezentaţi în capitolele 4 şi 5. Utilizarea unei reţele de calculatoare impune o atenţie

[2] Mai precis, va fi *aproape* dublu, întrucât efortul necesar actualizării ponderilor nu se modifică.

sporită în ceea ce priveşte comunicaţia între maşini, aceasta fiind una dintre cauzele pentru care accelerarea obţinută nu va putea fi cea ideală.

Capitolul trasează şi o serie de experimente care vor trebui efectuate în vederea măsurării performanţelor algoritmilor propuşi. În principal, se va urmări comportarea acestor algoritmi atât pentru probleme de dimensiune fixă — cum afectează mărirea numărului de maşini pe care rulează simulatorul — precum şi pentru probleme de dimensiune variabilă — respectiv relaţia dintre dimensiunea problemei şi numărul de procesoare utilizate.

4 DataParSim – simulator paralel cu partiţionarea datelor

Capitolul de faţă prezintă algoritmii care au stat la baza construcţiei simulatorului DataParSim — simulator paralel cu partiţionarea datelor, implementat şi testat pe trei reţele de calculatoare, una conţinând 12 staţii IBM RS6000, una cu 16 staţii SunBlade150 şi o reţea de 16 PC–uri cu procesor Pentium la 2,8 GHz. Secţiunile 4.1, 4.2 descriu algoritmii utilizaţi, în secţiunea 4.3 este prezentat simulatorul realizat pe baza acestor algoritmi, urmând ca în 4.4 să fie prezentate şi evaluate rezultatele experimentelor efectuate.

După cum s-a precizat în capitolul anterior, un mod simplu de paralelizare a unei reţele neuronale, cu actualizarea ponderilor după epocă sau în mod *batch*, este aşa-numita strategie de *partiţionare a datelor*, în care datele de antrenare sunt distribuite uniform pe procesoare. Toate procesoarele simulează întreaga reţea, dar pentru sub-seturi diferite de antrenare. Pe parcursul fiecărui ciclu de învăţare, fiecare procesor prezintă tiparele din "felia" sa din setul curent[1]. Gradienţii calculaţi în acest mod de către fiecare dintre procesoare îi vom numi *gradienţi componenţi*, întrucât fiecare dintre ei este rezultatul prezentării către reţea doar a unei porţiuni din setul de antrenare. Odată ce toate tiparele din fiecare set au fost prezentate de către diferitele procesoare, gradienţii componenţi rezultaţi sunt combinaţi (însumaţi) într-un gradient global, care este apoi utilizat pentru calculul modificării fiecărei ponderi. Apoi, ponderile sunt actualizate şi noile valori sunt transmise tuturor procesoarelor, ca fiind noul set de ponderi care vor fi utilizate în prezentările de tipare din următorul ciclu de învăţare.

Aceasta este cea mai simplă formă de partiţionare a datelor pentru paralelizarea reţelelor neuronale. Se va discuta implementarea acestei forme în secţiunea următoare. În secţiunea 4.2 se va prezenta o variaţiune a partiţionării datelor, în care fiecare procesor nu tratează o întreagă copie a reţelei.

4.1 O implementare simplă a strategiei de partiţionare a datelor

Procesul de implementare a acestei forme de partiţionare a datelor este foarte simplu şi evident, întrucât se dispune deja de o implementare secvenţială a algoritmului de backpropagation. Tot ce trebuie făcut este să se pună împreună un set de procesoare, fiecare dintre ele rulând algoritmul secvenţial clasic, în mod independent. Având P procesoare, vor fi deci P copii identice ale întregii reţele. Pe parcursul fiecărui ciclu de învăţare, fiecare procesor q prezintă reţelei tiparele din porţiunea sa, B_q, din setul curent B de tipare de antrenare. Ca rezultat, în procesorul q se calculează gradientul component $\overline{g}^{<q>}$:

$$\overline{g}^{<q>} = \sum_{p \in B_q} \frac{\partial E_p}{\partial \overline{w}} \tag{4.1}$$

unde $\overline{w}$ denotă setul tuturor ponderilor.

Pe baza ecuaţiilor

$$\Delta w^o_{j \to k} = -\eta \sum_p \frac{\partial E_p}{\partial w^o_{j \to k}} = \eta \sum_p \delta^o_{pk} a^H_{pj} \tag{4.2}$$

şi

$$\Delta w^o_{j \to k}(n+1) = \eta \sum_p \delta^o_{pk} a^H_{pj} + \alpha \Delta w^o_{j \to k}(n) \tag{4.3}$$

[1] În această secţiune nu se face distincţie între actualizarea la epocă şi cea în mod *batch*, întrucât din punctul de vedere al paralelizării unui algoritm, actualizarea la epocă poate fi privită ca un caz special de mod *batch*, în care setul de antrenare este constituit din întregul set de tipare.

modificarea ponderilor conform tiparelor din setul B poate fi exprimată acum ca sumă a gradienţilor componenţi $\overline{g}^{<q>}$:

$$\Delta_B \overline{w}(n+1) = -\eta \sum_{p \in B} \frac{\partial E_p}{\partial \overline{w}} + \alpha \Delta_B \overline{w}(n) = -\eta \sum_{q=0}^{P-1} \overline{g}^{<q>} + \alpha \Delta_B \overline{w}(n) \qquad (4.4)$$

Ca urmare, trebuie efectuată o singură modificare faţă de algoritmul secvenţial: o schemă de colectare şi însumare a tuturor gradienţilor componenţi $\overline{g}^{<q>}$ şi distribuirea ponderilor actualizate (*broadcast*) în modul cel mai eficient posibil.

4.1.1 Configuraţie inel. Configuraţie arbore

În funcţie de configuraţia de procesoare, trebuie utilizate scheme de comunicaţie diferite, în funcţie de modul de conectare.

Având o configuraţie în inel de P procesoare, se poate implementa însumarea gradienţilor componenţi astfel încât fiecare procesor, după $P-1$ paşi, să deţină gradientul total. Iniţial, fiecare procesor comunică gradientul component propriu către predecesorul său din inel (figura 4.1). Apoi toate procesoarele adună gradientul component recepţionat la o sumă temporară. În acelaşi timp, gradientul component este transmis predecesorului, concurent cu recepţionarea unui nou gradient component.

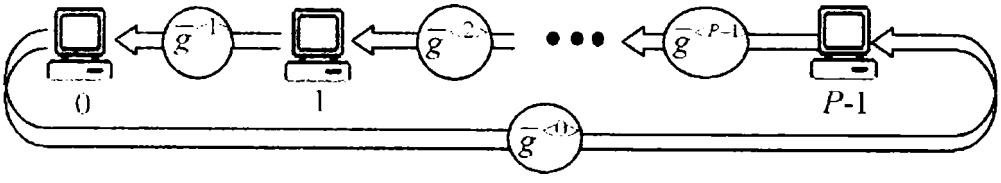


Figura 4.1: *Însumarea gradienţilor componenţi.*

După $P-1$ paşi, fiecare procesor a recepţionat gradienţii componenţi de la toate celelalte procesoare. În acest moment, el va deţine suma tuturor gradienţilor componenţi şi va putea face actualizarea ponderilor.

O schemă similară este prezentată pentru maşina GF11 în lucrarea lui Witbrock şi Zagha *"An implementation of backpropagation learning on GF11, a large SIMD parallel computer"* [100], şi aproape aceeaşi abordare este utilizată în lucrarea lui Bourrely *"Paralelisation of a neural learning algoritm on a hypercube"* [7], însă pe un hipercub.

O abordare alternativă mai eficientă este de asemenea prezentată în [100]. Facilităţile de comunicare ale maşinii GF11 permit o inter-conectivitate mult mai complexă decât în inel. Aici se descrie cum P procesoare pot colecta şi însuma gradienţii componenţi în $O(log_2 P)$ paşi, într-o configuraţie astfel încât procesorul q poate comunica cu toate procesoarele $(q+2^i) \bmod P$, pentru $i = 0, \ldots, log_2 P$.

Acest mod de interconectare nu poate fi însă aplicat, datorită traficului încărcat care ar rezulta. Totuşi, este posibilă însumarea gradienţilor componenţi în $O(log_2 P)$ paşi, configurând procesoarele sub forma unui arbore binar[2].

După prezentarea tuturor tiparelor din setul curent, toate procesoarele fără succesori în structura de arbore transmit gradienţii componenţi acumulaţi către procesorul părinte (Figura 4.2a). Toate celelalte procesoare colectează gradienţii componenţi de la succesorii lor imediaţi, îl însumează pe cel propriu şi transmit rezultatul în sus în ierarhie (Figura 4.2b). După $\lfloor log_2 P \rfloor$ paşi, procesorul rădăcină va deţine gradientul

[2]S-ar putea utiliza şi arbori de nivel superior, reducând astfel numărul de paşi şi înălţimea arborelui, însă ar creşte necesarul de memorie pentru comunicarea gradienţilor componenţi, iar traficul ar fi acelaşi.

total, care poate fi utilizat în calculul modificării ponderilor. Utilizând structura de arbore din nou, procesorul rădăcină poate transmite ponderile actualizate tuturor celorlalte procesoare în $\lfloor log_2 P \rfloor$ paşi.

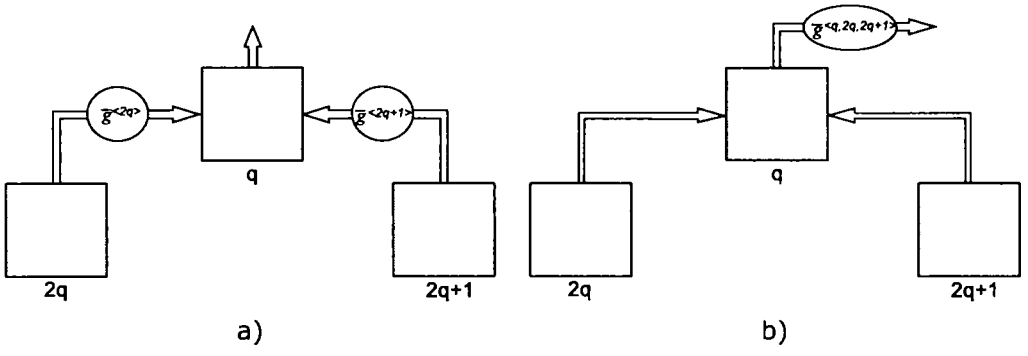


Figura 4.2: *Însumarea gradienţilor componenţi într-un arbore binar.*

P procesoare în inel pot actualiza ponderile în $P-1$ paşi, în timp ce sunt necesari $\lfloor log_2 P \rfloor$ paşi într-o configuraţie în arbore binar. Aceasta înseamnă că dacă se utilizează mai mult de 5 procesoare, configuraţia în arbore necesită mai puţini paşi. Este deci evident că se va prefera configuraţia de arbore atunci când este disponibil un număr mare de procesoare. În schimb, în configuraţia în inel se poate efectua însumarea gradienţilor componenţi concurent cu transmisia şi recepţia de (alţi) gradienţi componenţi. De remarcat însă faptul că efectuarea în paralel a comunicaţiei şi a calculelor economiseşte cel mult timpul asociat task-ului cel mai scurt[3].

4.1.1.1 Necesarul de memorie

Necesarul de memorie este aproximativ acelaşi pentru ambele configuraţii. În ambele cazuri, fiecare procesor trebuie să rezerve spaţiu în plus pentru câte două valori reale pentru fiecare pondere din reţea, necesare la transmisia gradienţilor componenţi.

Singura diferenţă semnificativă în privinţa necesarului de memorie dintre cei doi algoritmi este următoarea: în configuraţia în inel fiecare procesor trebuie să păstreze modificările anterioare ale ponderilor, pe lângă valorile calculate curent şi valorile ponderilor. Aceasta nu mai este necesar în configuraţia de arbore, întrucât noul set de ponderi se calculează în procesorul rădăcină. Ca urmare, doar procesorul rădăcină trebuie să memoreze modificările anterioare. Toate celelalte procesoare necesită spaţiu numai pentru ponderi, gradienţii componenţi calculaţi şi câte doi vectori de ponderi pentru comunicaţie.

Întrucât fiecare procesor păstrează de asemenea o copie a tuturor unităţilor din reţea, dacă NI, NH şi NO reprezintă numărul de neuroni de pe stratul de intrare, cel ascuns, respectiv stratul de ieşire, se obţine următorul necesar de memorie[4] (măsurat în valori reale) pentru fiecare procesor (cu excepţia celui rădăcină):

$$2(NI + NH + NO) + 4((NI + 1)NH + (NH + 1)NO) \tag{4.5}$$

Acest necesar enorm de memorie impune o limită mult prea strictă asupra dimensiunii reţelelor neuronale care pot fi paralelizate utilizând implementarea simplă a strategiei de partiţionare a datelor. Există însă diferite moduri de evitare a unui asemenea necesar de memorie. Unul ar fi să se încerce reducerea redundanţei cât

[3] Cel puţin în principiu; trebuie ţinut însă cont de faptul că transmisia se face printr-o reţea Ethernet.
[4] Au fost incluse doar variabile influenţate de dimensiunea reţelei.

mai mult posibil; altă soluţie ar fi să nu se transmită un întreg gradient component dintr-odată. Ambele strategii vor fi utilizate într-un algoritm prezentat în secţiunea 4.2.

4.2 O implementare avansată a strategiei de partiţionare a datelor

Pornind de la o scurtă descriere a unui algoritm similar construit de Pomerleau şi colegii [68] pentru o maşină Warp, această teză propune o soluţie originală, construită cu scopul minimizării necesarului de memorie şi al eliminării datelor redundante.

Se prezintă în această secţiune o implementare mai puţin costisitoare din punctul de vedere al memoriei, a strategiei de partiţionare a datelor. Procesoarele individuale nu mai păstrează propria copie a întregii reţele. Ponderile sunt memorate într-un singur procesor (numit *administrator*) şi circulă, una câte una, între procesoare, în funcţie de necesităţi.

Administratorul nu procesează tipare de antrenare şi nu le prezintă el însuşi reţelei neuronale. Această sarcină este efectuată exclusiv de către restul procesoarelor (numite *slave*), cărora li se distribuie în mod uniform tiparele de antrenare. Administratorul controlează circulaţia ponderilor şi efectuează actualizarea lor[5].

Întrucât fiecare procesor *slave* nu păstrează permanent nici o pondere, este esenţial ca fiecare pondere recepţionată să fie utilizată în cât mai multe calcule posibil, pentru a evita astfel transmisiile inutile de ponderi. Pe parcursul pasului înainte, aceasta înseamnă că la transmisia de ponderi care conectează un strat cu următorul, aceste ponderi trebuie utilizate în propagarea activităţii corespunzătoare pentru cât mai multe tipare posibil, adică procesorul q va utiliza ponderile pentru propagarea activităţii tuturor tiparelor din porţiunea sa B_q din setul curent de tipare. Pentru aceasta, fiecare procesor slave q trebuie să păstreze atâtea copii ale tuturor unităţilor din reţea câte tipare sunt în B_q.

Ca rezultat al consideraţiilor de mai sus, toate tiparele din B_q sunt propagate înainte de către procesorul q înainte ca valorile de eroare generate de către aceste tipare să fie propagate înapoi în reţea. Aceasta contravine algoritmului descris în secţiunea 4.1, în care un tipar era propagat înainte *şi* înapoi, şi se calcula un singur gradient component, înainte de prezentarea următorului tipar.

4.2.1 Topologia procesoarelor

Toate procesoarele slave trebuie să recepţioneze fiecare pondere din reţea o singură dată pe parcursul pasului înainte, începând cu ponderile dintre unităţile de pe stratul de intrare şi cel ascuns. La prima vedere, o configuraţie de inel poate să nu pară ideală, din moment ce numărul de paşi de comunicare necesari pentru transmiterea unei ponderi de la administrator la câţiva slave poate fi destul de mare. Totuşi, toate procesoarele slave necesită aceleaşi ponderi, însă nu este neapărat necesar să utilizeze aceleaşi ponderi în acelaşi timp. Ca urmare, ponderile pot fi transmise una câte una prin inel, în stil *pipe*. În acest fel, nici un procesor nu va avea de aşteptat pentru nici o pondere, cu excepţia fazelor de pornire şi de oprire, care constituie o mică parte din timpul total de execuţie, întrucât numărul de ponderi este de obicei mult mai mare decât numărul de procesoare.

4.2.2 Comunicaţia

După cum s-a menţionat în secţiunea anterioară, ponderile reţelei neuronale sunt circulate una câte una prin inelul de procesoare pe parcursul activităţii de propagare.

[5] În realitate, se va vedea că şi administratorul va procesa câteva tipare, mai puţine decât procesoarele *slave*. Totuşi, pentru moment se va presupune că administratorul nu prezintă tipare reţelei.

Aceasta înseamnă că pe parcursul pasului înainte fiecare procesor trebuie să recepţioneze şi să transmită fiecare pondere din întreaga reţea, ceea ce înseamnă că fiecare procesor trebuie să participe într-un număr dublu de comunicaţii faţă de numărul de ponderi din reţea. La fiecare pondere comunicată avem asociat un anumit volum de calcule. Utilizând un asemenea volum mare de comunicaţii, este important să construim algoritmul astfel încât să asigure efectuarea concurentă a calculelor şi a transmisiei de ponderi. Mai mult, dacă timpul necesar calculelor este mult mai mare decât timpul de comunicaţie, transmisia ponderilor poate avea loc fără încetinirea semnificativă a calculelor efectuate de *slave*.

De aceea, se recomandă o transmisie asincronă a acestor ponderi.

4.2.3 Pasul înainte

Pe parcursul propagării activităţii ponderile sunt circulate una câte una prin inelul de procesoare de către administrator, aşa cum este arătat în figura 4.3.

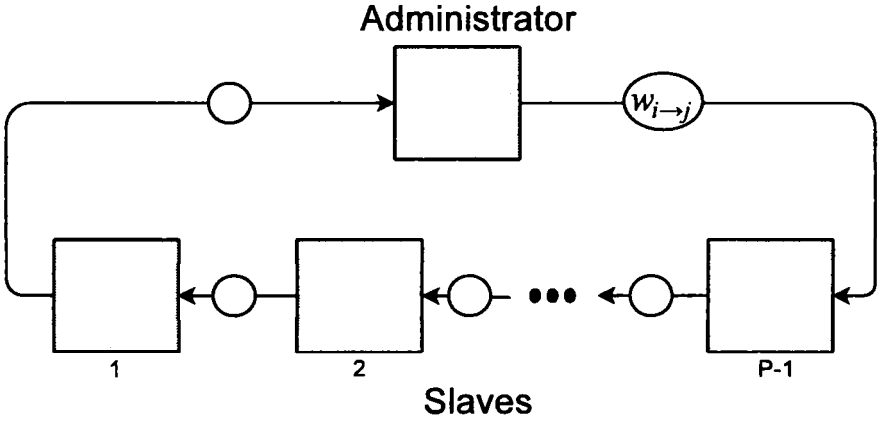


Figura 4.3: *Circulaţia ponderilor în pasul înainte.*

Iniţial, sunt circulate ponderile dintre stratul de intrare şi cel ascuns, permiţând astfel propagarea activităţii de la unităţile de intrare către cele ascunse. Apoi se transmit ponderile dintre unităţile ascunse şi cele de ieşire şi se calculează tiparul de activitate pe unităţile de ieşire. Aceşti doi paşi de propagare înainte a activităţii se efectuează în acelaşi mod, singura diferenţă constând în ponderile care se transmit şi unităţile utilizate în calcule. În continuare vom descrie în detaliu numai propagarea activităţii de la unităţile de intrare către cele ascunse.

De fiecare dată când un procesor q recepţionează o pondere $w^H_{i\to j}$ care conectează unitatea i din stratul de intrare cu unitatea j din stratul ascuns, transmite în continuare ponderea către următorul procesor din inel (procesorul $q-1$). Concurent, această pondere este utilizată pentru calculul contribuţiei unităţii i la intrarea totală a unităţii j pentru toate tiparele din porţiunea B_q a acestui procesor din întregul set B. Cu alte cuvinte, pentru fiecare tipar p din B_q, la recepţionarea lui $w^H_{i\to j}$ se calculează a i-a parte din suma:

$$\sum_{i=0}^{NI-1} a^I_{pi} w^H_{i\to j} = net^H_{pj}, \qquad \forall p \in B_q \tag{4.6}$$

După ce toate ponderile care conectează cele două straturi au trecut pe la toate procesoarele, fiecare procesor q conţine pentru toate unităţile din stratul ascuns cele

$|B_q|$ intrări generate de tiparele din B_q. Apoi fiecare procesor q calculează activarea fiecărei unităţi, pentru fiecare tipar p din B_q:

$$a_{pj}^{H} = f(net_{pj}^{H}), \qquad \forall p \in B_q, \qquad \forall j \in H \tag{4.7}$$

Odată calculată activitatea unităţilor ascunse, este posibil să se determine tiparele activării pentru unităţile de ieşire. Aceasta se efectuează exact în acelaşi mod ca mai sus, cu diferenţa că de data aceasta administratorul transmite ponderile dintre unităţile ascunse şi cele de ieşire.

4.2.4 Pasul înapoi

Propagarea înapoi a valorilor delta şi calculul gradienţilor componenţi este realizată în trei paşi.

4.2.4.1 Pasul întâi

Iniţial, trebuie calculate valorile delta pentru toate unităţile de ieşire, pentru fiecare din tiparele din setul corespunzător fiecărui procesor. Pentru toate tiparele p din B_q, procesorul q calculează valoarea delta pentru fiecare unitate de ieşire:

$$\delta_{pk}^{O} = a_{pk}^{O}(1 - a_{pk}^{O})(t_{pk} - a_{pk}^{O}), \qquad \forall p \in B_q, \qquad \forall k \in O \tag{4.8}$$

După cum se observă, acest pas nu necesită circularea de ponderi şi poate fi efectuat independent, de către fiecare dintre procesoare.

4.2.4.2 Pasul al doilea

Următorul pas constă în propagarea înapoi a valorilor delta către unităţile ascunse. Acest pas necesită circulaţia din nou a ponderilor dintre unităţile ascunse şi cele de ieşire, pentru a calcula contribuţia fiecărei unităţi de ieşire la eroarea din fiecare unitate ascunsă. Aceasta se efectuează într-un mod similar celui descris la pasul înainte.

Atunci când procesorul q recepţionează o pondere $w_{j\to k}^{O}$ conectează unitatea j din stratul ascuns cu unitatea k din stratul de ieşire, se transmite această pondere către următorul procesor din inel (procesorul $q-1$). Concurent, ponderea este utilizată pentru a calcula, pentru toate tiparele din B_q, contribuţia unităţii de ieşire k la eroarea din unitatea ascunsă j. La recepţionarea lui $w_{j\to k}^{O}$, pentru fiecare tipar p din B_q se calculează a k-a parte din următoarea sumă:

$$\sum_{k=0}^{NO-1} \delta_{pk}^{O} w_{j\to k}^{O} = e_{pj}^{H}, \qquad \forall p \in B_q \tag{4.9}$$

După ce toate ponderile conexiunilor dintre unităţile ascunse şi cele de ieşire au trecut pe la toate procesoarele, procesorul q conţine, pentru toate unităţile ascunse, cele $|B_q|$ valori de eroare generate de tiparele din B_q. Fiecare procesor calculează apoi valorile delta corespunzătoare pentru toate tiparele din B_q:

$$\delta_{pj}^{H} = a_{pj}^{H}(1 - a_{pj}^{H})e_{pj}^{H}, \qquad \forall p \in B_q, \qquad \forall j \in H \tag{4.10}$$

4.2.4.3 Pasul al treilea

Al treilea pas calculează gradientul rezultat în urma prezentării tuturor tiparelor din B.

Calculul gradienţilor componenţi parţiali asociaţi ponderilor conexiunilor dintre unităţile de intrare şi cele ascunse se efectuează în modul următor. Pe baza ecuaţiei (prezentată în secţiunea 2.1)

$$\Delta_B \overline{w}(n+1) = -\eta \sum_{p \in B} \frac{\partial E_p}{\partial \overline{w}} + \alpha \Delta_B \overline{w}(n) = -\eta \sum_{q=0}^{P-1} \overline{g}^{<q>} + \alpha \Delta_B \overline{w}(n) \tag{4.11}$$

putem obţine modificarea $\Delta_B w_{i\to j}^H$ a ponderii $w_{i\to j}^H$ corespunzător tiparelor din setul B:

$$\Delta_B w_{i\to j}^H(n+1) = -\eta \sum_{p\in B} \frac{\partial E_p}{\partial w_{i\to j}^H} + \alpha \Delta_B w_{i\to j}^H(n) = -\eta \sum_{q=0}^{P-1} g_{i\to j}^{H<q>} + \alpha \Delta_B w_{i\to j}^H(n) \quad (4.12)$$

Cu alte cuvinte, pentru actualizarea ponderii $w_{i\to j}^H$ administratorul trebuie să cunoască suma gradienţilor componenţi *parţiali* $g_{i\to j}^{H<q>}$ de la toate procesoarele *slave*. Un asemenea gradient component parţial poate fi calculat cu:

$$g_{i\to j}^{H<q>} = \sum_{p\in B_q} \frac{\partial E_p}{\partial w_{i\to j}^H} = \sum_{p\in B_q} \delta_{pj}^H a_{pi}^H \quad (4.13)$$

Calculul fiecărui asemenea gradient component parţial poate fi efectuat de către procesorul respectiv, întrucât deja au fost calculate valorile delta şi cele de activare pentru toate tiparele $p \in B_q$, pentru toate unităţile din reţea. Însumarea gradienţilor componenţi parţiali se efectuează într-o manieră similară calculului erorilor în unităţile ascunse. Însă în loc să se transmită ponderi, se transmit gradienţii componenţi parţiali calculaţi, după cum se vede în Figura 4.4[6].

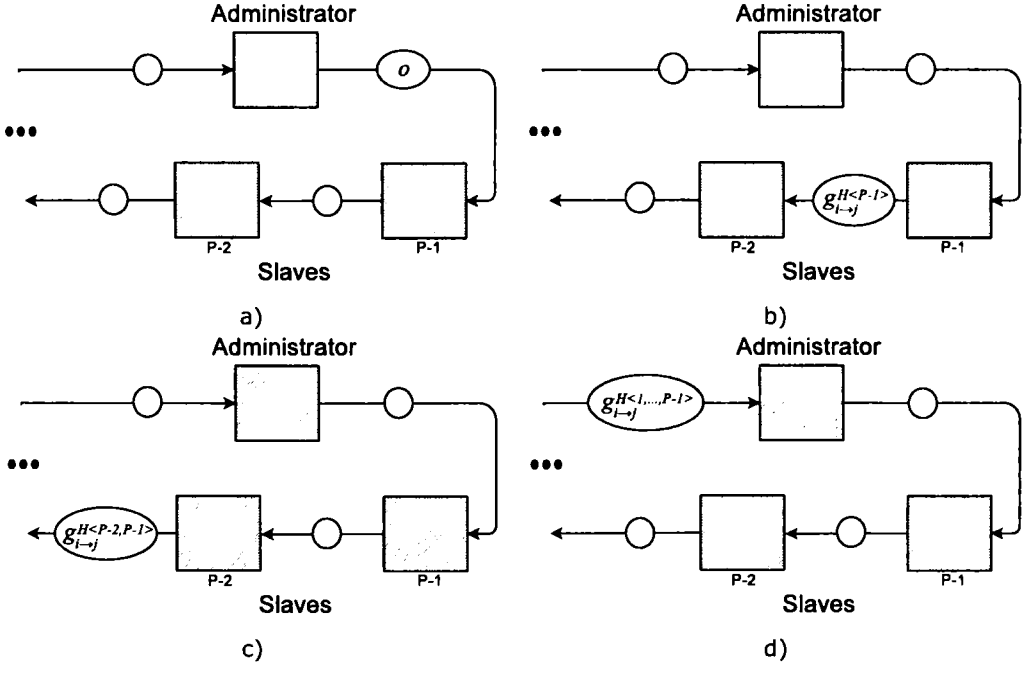


Figura 4.4: *Calculul gradienţilor parţiali la pasul înapoi.*

Iniţial, pentru fiecare gradient parţial $g_{i\to j}^H$ de calculat, administratorul transmite valoarea 0 procesorului $P-1$ (Figura 4.4a). Concurent, procesorul $P-1$ calculează contribuţia sa $g_{i\to j}^{H<P-1>}$. Apoi gradientul component parţial al procesorului $P-1$ este adunat la 0, iar rezultatul este transmis mai departe procesorului $P-2$ (Figura 4.4b).

[6] De fapt, se transmit *negatele* gradienţilor componenţi parţiali. În acest fel se evită negarea gradientului parţial total la calculul modificării ponderii.

În acest timp, procesorul $P-2$ calculează $g_{i\to j}^{H<P-2>}$ şi, la recepţionarea lui $g_{i\to j}^{H<P-1>}$ de la procesorul $P-1$, le însumează şi transmite rezultatul $g_{i\to j}^{H<P-2,P-1>}$ către procesorul $P-3$ (Figura 4.4c).

În general, procesorul q calculează $g_{i\to j}^{H<q>}$, în timp ce recepţionează gradientul component parţial $g_{i\to j}^{H<q+1,\ldots,P-1>}$ de la procesorul $q+1$. Gradientul component parţial calculat de procesorul q este adunat gradientului component parţial sosit, iar rezultatul $g_{i\to j}^{H<q,\ldots,P-1>}$ este transmis procesorului $q-1$ concurent cu calculul următorului gradient component parţial (corespunzător ponderii care urmează după $w_{i\to j}^{H}$).

În acest mod, gradienţii componenţi parţiali sunt însumaţi de către toate procesoarele *slave*, astfel încât la sfârşit (după P paşi) administratorul recepţionează $g_{i\to j}^{H<1,\ldots,P-1>} = g_{i\to j}^{H}$, gradientul parţial rezultat în urma prezentării tuturor tiparelor din B (Figura 4.4d). Administratorul va utiliza acest gradient parţial pentru calculul actualizării $\Delta_B w_{i\to j}^{H}$ pentru ponderea $w_{i\to j}^{H}$.

Gradienţii componenţi parţiali $g_{i\to j}^{O}$ asociaţi ponderilor conexiunilor dintre unităţile ascunse şi cele de ieşire se calculează în acelaşi mod.

După ce au fost recepţionaţi toţi gradienţii parţiali şi au fost actualizate toate ponderile, administratorul poate transmite prima pondere dintre unităţile de intrare şi cele ascunse, iar prezentarea unui alt set de antrenare poate să înceapă.

4.2.5 O îmbunătăţire a eficienţei

Administratorul efectuează circularea ponderilor, colectarea gradienţilor parţiali şi actualizarea ponderilor. Numai ultima dintre aceste sarcini presupune un efort computaţional. Ca urmare, pe parcursul pasului înainte administratorul nu va fi ocupat majoritatea timpului, având doar de aşteptat transmisia sau recepţia următoarei ponderi. Pe parcursul pasului înapoi, administratorul va fi ceva mai ocupat, având de actualizat ponderile pe măsură ce recepţionează gradienţii parţiali. Totuşi, avem de-a face cu o diferenţă importantă între efortul administratorului şi cel al procesoarelor *slave*: efortul computaţional în slave este proporţional cu numărul de tipare din porţiunea sa din setul curent de antrenare, în timp ce efortul computaţional asociat actualizării ponderilor de către administrator este constant, independent de dimensiunea setului de antrenare.

Consideraţiile de mai sus conduc la concluzia că probabil se pierde timpul în administrator atunci când porţiunea din setul de antrenare aferentă unui *slave* este suficient de mare. Soluţia este simplă: şi administratorul va primi de procesat o "felie" din setul de antrenare.

Dificultatea apare în determinarea dimensiunii acestei porţiuni pe care o va procesa administratorul. Evident, va trebui să proceseze mai puţine tipare decât procesoarele slave, altfel nu va mai rămâne timp pentru celelalte activităţi. Însă procesarea de către administrator a unor tipare va creşte eficienţa numai dacă aceasta reduce încărcarea procesoarelor *slave*, ceea ce va creşte viteza procesoarelor *slave*. Nu ajută la nimic însă dacă se reduce încărcarea pe unele dintre procesoarele *slave*, întrucât în inel intervalul dintre transmisii este determinat de procesorul care transmite cel mai rar. Aceasta înseamnă că administratorul trebuie să reducă încărcarea pe *slave* într-un asemenea mod încât încărcarea să fie echilibrată uniform pe toate procesoarele *slave*. Numărul de tipare de procesat în administrator trebuie să fie astfel încât procesoarele slave să trateze "felii" la fel de mari din setul de antrenare, pentru ca să nu avem procesoare care să fie în aşteptarea altuia să fie gata pentru comunicaţie.

Pe de altă parte, dacă administratorul are de prelucrat acelaşi număr de tipare ca şi oricare *slave*, nu va mai avea timp pentru actualizarea ponderilor. Mai rău, va încetini activitatea tuturor procesoarelor *slave*, întrucât acestea vor avea de aşteptat până ce administratorul termină de actualizat ponderile. Este obligatoriu deci ca administratorul să prelucreze ceva mai puţine tipare decât procesoarele *slave*. Va trebui investigat care este valoarea optimă care trebuie aleasă pentru numărul de tipare de

procesat în administrator. Empiric, s-a ajuns la o formulă de calcul a dimensiunilor seturilor de tipare prelucrate de către administrator, respectiv de către procesele *slave*, formulă prezentată în 4.3.2.

4.2.6 Necesarul de memorie

Unul dintre motivele proiectării acestei variante avansate l-a constituit necesarul excesiv de memorie în cazul algoritmului prezentat mai devreme în secţiunea 4.1. Acolo, fiecare procesor păstra o copie a întregului set de ponderi. Proiectarea acestui algoritm s-a bazat pe încercarea de a reduce necesarul de memorie prin evitarea redundanţelor introduse de necesitatea de a memora P copii identice ale tuturor ponderilor. În acest algoritm numai administratorul păstrează întregul set de ponderi, în vreme ce toate celelalte procesoare au nevoie doar de capacitatea de a memora două sau trei ponderi. Totuşi, fiecare procesor *slave* trebuie să păstreze câteva copii ale tuturor *unităţilor* din reţea, întrucât prezentarea completă a unui tipar (incluzând propagarea înainte a activităţii şi propagarea înapoi a erorii) nu se poate realiza într-un singur pas. Din fericire, acest necesar suplimentar de memorie nu este nici pe departe atât de mare în comparaţie cu memoria economisită prin eliminarea setului redundant de ponderi în fiecare procesor.

În cele ce urmează, interesul se va concentra asupra variabilelor care variază fie cu dimensiunea reţelei, fie cu cea a setului de antrenare, şi vor fi excluse toate variabilele constante în toate execuţiile. Numărul maxim de tipare din porţiunea unui procesor *slave* este $\lceil B/(P-1) \rceil$, întrucât ar putea fi posibil să nu se permită administratorului să participe la procesarea de tipare. Ca urmare, se obţine următorul necesar de memorie pentru fiecare procesor *slave*(vezi 4.1.1.1):

$$\lceil B/(P-1) \rceil \cdot 2(NI + NH + NO) \tag{4.14}$$

Se poate spune că necesarul de memorie pentru procesoarele slave este independent de numărul de ponderi din reţea, contrar cazului primului algoritm de partiţionare a datelor. Acesta este un avantaj, întrucât numărul de ponderi dintr-o reţea creşte cuadratic cu numărul de unităţi. După cum era de aşteptat, algoritmul descris va fi mai puţin consumator de memorie la execuţia pe reţele mari.

4.3 Simulatorul DataParSim

A fost realizat un simulator — DataParSim constând din două programe: un program *administrator* şi un program *slave*. Algoritmul utilizat este cel descris în secţiunea 4.2 – implementarea avansată, în care administratorul transmite în reţea câte o pondere, eliminându-se astfel necesitatea memorării tuturor ponderilor local, la fiecare procesor. S-a încercat îmbunătăţirea eficienţei simulatorului prin procesarea unui număr de tipare şi pe maşina *administrator*.

Informaţiile necesare celor două programe se transmit prin intermediul unui *fişier de configurare*, care precizează parametrii reţelei neuronale şi cei ai reţelei de calculatoare în care va evolua simulatorul. Acest fişier, precum şi modul în care lucrează *administratorul* şi programele *slave* sunt descrise în continuare.

4.3.1 Fişierul de configurare

Cele două programe, *administratorul* şi procesele *slave*, primesc informaţiile necesare prin intermediul unui fişier de configurare, identificat prin intermediul variabilei de mediu `DATAPARSIM_CFG`, fişier care conţine informaţii despre reţeaua neuronală de simulat şi despre staţiile care alcătuiesc reţeaua de calculatoare. O informaţie suplimentară se transmite programelor *slave* în linia de comandă – numărul procesorului în configuraţia de inel.

Se dă mai jos un exemplu de fişier de configuraţie. Valorile sunt precizate câte una pe linie, sub forma

```
<variabilă>=<valoare>
```

Liniile goale se ignoră, iar textul de la semnul '#' până la sfârşitul liniei se ignoră (comentariu).

```
# Reteaua neuronala:
eta = 0.1                    # rata de invatare
alpha = 0.9                  # momentul

NrInput = 50                 # neuroni pe stratul de intrare
NrHidden = 50                # neuroni pe stratul intermediar
NrOutput = 50                # neuroni pe stratul de iesire

NrPatt = 800                 # numar total de tipare
PattFile = /work/tipare.dat  # fisier tipare de invatare

BatchSize = 800              # dimensiune batch

NrIter = 1000                # numar de iteratii
Err = 0.01                   # eroare acceptata

# Reteaua de calculatoare:
NrProcs = 12                 # numar procesoare in retea
HostsFile = /work/hosts.dat  # lista numelor statiilor


BatDelta = 103     #
```

Parametrul `NrIter` precizează numărul total de iteraţii care se efectuează. Dacă el este 0, atunci se ia ca şi criteriu de oprire atingerea unei erori totale mai mică decât `Err`.

Parametrul `BatDelta` este un coeficient care exprimă raportul dintre mărimea setului de tipare de antrenare pentru un *slave* şi cel pentru *administrator*. Valoarea lui a fost determinată empiric, astfel încât setul de tipare pentru *administrator* să fie cât mai mare posibil, însă nu atât de mare încât procesele *slave* să rămână în aşteptare. Pentru exemplul dat, s-a constatat o evoluţie convenabilă pentru un număr de tipare pentru *administrator* cu circa 3% mai mic decât pentru *slave*.

4.3.2 Programul administrator

După cum este descris în secţiunea 4.2, *administratorul* este singurul proces care păstrează întreaga reţea neuronală, transmiţând câte o pondere la procesele *slave*. Programul administrator execută mai mulţi paşi:

- extrage din fişierul de configurare parametrii necesari reţelei neuronale – număr de neuroni pe fiecare strat, rata de învăţare şi momentul şi criteriile de oprire (număr de iteraţii, eroarea totală), împreună cu structura reţelei de calculatoare şi setul de tipare de antrenare;
- determină dimensiunile seturilor de tipare pentru *administrator* şi pentru *slave* şi mulţimea de tipare corespunzătoare fiecărui proces:

```
NrSlaves = NrProcs - 1;

AdminBatchMax = (100 * BatchSize) /
                    ((BatDelta * NrSlaves) + 100);
SlaveBatch = ((BatchSize - AdminBatchMax) + (NrSlaves - 1)) /
                    NrSlaves;
AdminBatch = BatchSize - (NrSlaves * SlaveBatch);
```

```
if (AdminBatch < 0) {
    AdminBatch = 0;
    SlaveBatch = (BatchSize + (NrSlaves - 1)) / NrSlaves;
}

AdminPatts = (NrPatt * AdminBatch) / BatchSize;
```

După acelaşi algoritm, se vor calcula aceşti parametri şi în procesele *slave*;

- iniţializează ponderile conexiunilor din reţeaua neuronală cu valori mici aleatoare;
- lansează câte un proces *slave* pe fiecare staţie;
- deschide canalele de comunicaţie între procesele *slave* şi le transmite aceste informaţii, astfel încât ele să formeze o configuraţie în inel;
- lansează procesul de simulare — un ciclu, conţinând `NrIter` iteraţii, sau până se atinge eroarea totală dorită. La fiecare iteraţie, procesul administrator este mai întâi *transmiţător* – trimite câte o pondere la primul *slave* din inel, atât pentru pasul înainte, cât şi pentru pasul înapoi. Aceeaşi pondere este utilizată pentru propagarea câte unui tipar din setul propriu de antrenare.

```
/* Sender: */
/* pas inainte */
for (j = 0; j < NrHidden; j++) {
  for (p = 0; p < BatCnt; p++); {
    hidden_net[p][j] = 0;
  }
  for (i = 0; i < NrInput; i++) {
    /* calcul intrare in unitati hidden */
    weight_output = hidden_weight[i][j];
    Send (weight_output);
    for (p = 0; p < BatCnt; p++) {
      pattern = (p + batch_start) % PattCnt;
      hidden_net [p][j] += input_pattern[pattern][i] *
                           hidden_weight[i][j];
    }
  }
}

for (j = 0; j < NrHidden; j++) {
  /* calcul intrare de la bias si activare in
     unitati hidden */
  weight_output = hidden_weight[NrInput][j];
  Send (weight_output);
  for (p = 0; p < BatCnt; p++) {
    pattern = (p + batch_start) % PattCnt;
    hidden_net[p][j] += hidden_weight[NrInput][j];
    hidden_act[p][j] = calculate_activity (hidden_net[p][j]);
  }
}

for (j = 0; j < NrOutput; j++) {
  for (p = 0; p < BatCnt; p++); {
    output_net[p][j] = 0;
  }
  for (i = 0; i < NrHidden; i++) {
    /* calcul intrari in unitati output */
    weight_output = output_weight[i][j];
    Send (weight_output);
    for (p = 0; p < BatCnt; p++) {
```

```
      output_net [p][j] += hidden_act[p][i] *
                           output_weight[i][j];
    }
  }
}

for (j = 0; j < NrOutput; j++) {
  /* calcul intrare de la bias si activare
     unitati output */
  weight_output = output_weight[NrHidden][j];
  Send (weight_output);
  for (p = 0; p < BatCnt; p++) {
    output_net[p][j] += output_weight[NrHidden][j];
    output_act[p][j] = calculate_activity (output_net[p][j]);
  }
}

Send (zero_value); /* dummy */

/* pas inapoi */
/* transmite ponderi hidden-output */
change_work = 0.0; /* dummy val */
for (j = 0; j < NrOutput; j++) {
  /* calcul eroare la output si actualizare ponderi bias */
  change_output = change_work;
  change_work = 0.0;
  Send (change_output);
  for (p = 0; p < BatCnt; p++) {
    pattern = (p + batch_start) % PattCnt;
    oua = output_act[p][j];
    output_err[p][j] = target_pattern[pattern][j] - oua;
    output_dlt[p][j] = output_err[p][j] * oua * (1.0 - oua);
    change_work += output_dlt[p][j];
  }
}

Send (change_work);

change_work = 0.0; /* dummy val */
for (i = 0; i < NrHidden; i++) {
  for (p = 0; p < BatCnt; p++) {
    hidden_err[p][i] = 0.0;
  }
  for (j = 0; j < NrOutput; j++) {
    /* calcul actualizare ponderi si eroare in hidden */
    weight_output = output_weight[i][j];
    change_output = change_work;
    change_work = 0.0;
    Send (weight_output); Send (change_output);
    for (p = 0; p < BatCnt; p++) {
      hidden_err[p][i] += output_dlt[p][j] *
                          output_weight[i][j];
      change_work += output_dlt[p][j] *
                     hidden_act[p][j];
    }
  }
}

Send (zero_value); Send (change_work);
```

```
/* transmite actualizare initiala pentru ponderi
catrla unitati hidden: */
change_work = 0.0;
for (j = 0; j < NrHidden; j++) {
  /* calcul delta in unitati hidden */
  change_output = change_work;
  change_work = 0.0;
  Send (change_output);
  for (p = 0; p < BatCnt; p++) {
    hua = hidden_act[p][j];
    hidden_dlt[p][j] = hidden_err[p][j] * hua * (1.0 - hua);
    change_work += hidden_dlt[p][j];
  }
  for (i = 0; i < NrInput; i++) {
    /* calcul actualizari */
    change_output = change_work;
    change_work = 0.0;
    Send (change_output);
    for (p = 0; p < BatCnt; p++) {
      pattern = (p + batch_start) % PattCnt;
      change_work += hidden_dlt[p][j] *
                     input_pattern[pattern][i];
    }
  }
}
Send (change_work);
```

După cum se observă, *administratorul* calculează gradienţii corespunzători setului propriu de tipare, după care îi transmite în reţea, la primul proces din inel. Urmează apoi rolul de *receptor* – primeşte de la ultimul proces din reţea gradienţii acumulaţi şi calculează noile ponderi în reţeaua neuronală:

```
/* Receiver: */
/* pas inainte: */
for (j = 0; j < NrHidden; j++) {
  for (i = 0; i < NrInput; i++) {
    Receive (weight);
  }
}
for (j = 0; j < NrHidden; j++) { /* bias */
  Receive (weight);
}
for (j = 0; j < NrOutput; j++) {
  for (i = 0; i < NrHidden; i++) {
    Receive (weight);
  }
}
for (j = 0; j < NrOutput; j++) { /* bias */
  Receive (weight);
}
Receive (weight); /* dummy */

/* pas inapoi: */
/* calcul noile ponderi hidden-output */
Receive (change); /* dummy */
for (j = 0; j < NrOutput; j++) {
  Receive (change);
  change = (alpha * prev_output_change[NrHidden][j]) +
           (eta * change);
```

```
    output_weight[NrHidden][j] += change;
    prev_output_change[NrHidden][j] = change;
  }
  Receive (weight); Receive (change); /* dummy */
  for (i = 0; i < NrHidden; i++) {
    for (j = 0; j < NrOutput; j++) {
      Receive (weight); Receive (change);
      change = (alpha * prev_output_change[i][j]) +
               (eta * change);
      output_weight[i][j] += change;
      prev_output_change[i][j] = change;
    }
  }

  /* calcul noile ponderi input-hidden */
  Receive (change); /* dummy */
  for (j = 0; j < NrHidden; j++) {
    Receive (change);
    change = (alpha * prev_hidden_change[NrInput][j]) +
             (eta * change);
    hidden_weight[NrInput][j] += change;
    prev_hidden_change[NrInput][j] = change;
    for (i = 0; i < NrInput; i++) {
      Receive (change);
      change = (alpha * prev_hidden_change[i][j]) +
               (eta * change);
      hidden_weight[i][j] += change;
      prev_hidden_change[i][j] = change;
    }
  }
```

- măsoară timpul necesar unei iteraţii şi întregului proces;
- salvează noua configuraţie a reţelei neuronale pentru utilizări ulterioare.

4.3.3 Programul *slave*

Procesele *slave* implementează algoritmul backpropagation pentru o singură iteraţie, conform celor prezentate anterior:

- extrage din fişierul de configurare parametrii necesari reţelei neuronale – număr de neuroni pe fiecare strat, rata de învăţare şi momentul, precum şi structura reţelei de calculatoare şi fişierul conţinând setul de tipare de antrenare. Din linia de comandă cu care este lansat extrage numărul procesorului în configuraţia în inel (argumentul `processor_number`);
- determină dimensiunile seturilor de tipare pentru *administrator*, respectiv pentru *slave* şi mulţimea de tipare corespunzătoare fiecărui proces, în aceeaşi manieră ca la *administrator*:

```
NrSlaves = NrProcs - 1;

MaxBatCnt = (BatchSize + (NrSlaves - 1)) / NrSlaves;
MaxPattCnt = (NrPatt + (NrSlaves - 1)) / NrSlaves;

/* calcul batch size pentru acest procesor */
AdminBatchMax = (100 * BatchSize) /
                ((BatDelta * NrSlaves) + 100);
SlaveBatch = ((BatchSize - AdminBatchMax) +
                          (NrSlaves - 1)) / NrSlaves;
```

```
AdminBatch = BatchSize - (NrSlaves * SlaveBatch);
if (AdminBatch < 0) {
  AdminBatch = 0;
  SlaveBatch = (BatchSize + (NrSlaves - 1)) / NrSlaves;
}

temp = (BatchSize - AdminBatch) % NrSlaves;
if (temp == 0) {
  BatCnt = SlaveBatch;
}
else if (temp < processor_number) {
  BatCnt = SlaveBatch - 1;
}
else
  BatCnt = SlaveBatch;

/* calcul numar tipare pt acest procesor */
AdminPatts = (NrPatt * AdminBatch) / BatchSize;
SlavePatts = NrPatt - AdminPatts;

PattCnt = (SlavePatts + (NrSlaves - 1)) / NrSlaves;
ptemp = SlavePatts % NrSlaves;
if ((ptemp != 0) && (ptemp < processor_number))
  PattCnt--;
```

- încarcă setul de tipare de antrenare pe care le va procesa;
- se conectează la canalele de comunicaţie corespunzătoare – este vorba despre procesul precedent şi cel următor;
- execută un ciclu infinit, în care propagă prin reţea tiparele din setul propriu şi calculează actualizările de ponderi; după cum se va vedea mai jos, aceste două operaţii presupun recepţia de la procesul precedent şi transmisia la procesul următor din inel:

```
propagate_activity (batch_start);
calculate_weight_changes (batch_start);
batch_start += BatCnt; batch_start %= PattCnt;
```

Rutina `propagate_activity` primeşte ca parametru poziţia de start în lista de tipare şi trece prin reţeaua neuronală `BatCnt` tipare. Calculele se efectuează recepţionând câte o pondere de la precedentul proces în inel şi trimiţând-o mai departe la următorul proces:

```
void propagate_activity (int batch_start)
{
  double weight_input, weight_work, weight_output;
  int pattern;

  Receive (weight_input);

  for (j = 0; j < NrHidden; j++) {
    for (p = 0; p < BatCnt; p++) {
      hidden_net[p][j] = 0.0;
    }
    for (i = 0; i < NrInput; i++) {
      /* calcul intrari in hidden */
      weight_work = weight_input;
      weight_output = weight_input;
      Send (weight_output);
      Receive (weight_input);
```

```
        for (p = 0; p < BatCnt; p++) {
          pattern = (p + batch_start) % PattCnt;
          hidden_net[p][j] +=
                    input_pattern[pattern][i] * weight_work;
        }
      }
    }
    for (j = 0; j < NrHidden; j++) {
      /* calcul intrare de la bias in hidden si activare */
      weight_work = weight_input;
      weight_output = weight_input;
      Send (weight_output);
      Receive (weight_input);
      for (p = 0; p < BatCnt; p++) {
        hidden_net[p][j] += weight_work;
        hidden_act[p][j] = calculate_activity (hidden_net[p][j]);
      }
    }

    for (j = 0; j < NrOutput; j++) {
      for (p = 0; p < BatCnt; p++) {
        output_net[p][j] = 0.0;
      }
      for (i = 0; i < NrHidden; i++) {
        /* calcul intrari in output */
        weight_work = weight_input;
        weight_output = weight_input;
        Send  (weight_output);
        Receive (weight_input);
        for (p = 0; p < BatCnt; p++) {
          output_net[p][j] += hidden_act[p][i] * weight_work;
        }
      }
    }
    for (j = 0; j < OUTPT_UNITS; j++) {
      /* calcul intrare de la bias in output si activare */
      weight_work = weight_input;
      weight_output = weight_input;
      Send (weight_output);
      Receive (weight_input);
      for (p = 0; p < BatCnt; p++) {
        output_net[p][j] += weight_work;
        output_act[p][j] = calculate_activity (output_net[p][j]);
      }
    }

    Send (weight_input); /* dummy */

  } /* propagate_activity() */
```

Rutina `calculate_weight_changes` însumează la gradientul ce a fost recepţionat de la procesul precedent, propriul gradient, şi-l transmite mai departe, la procesul următor în inel:

```
void calculate_weight_changes (int batch_start)
{
  double weight_input, weight_work, weight_output;
  double change_input, change_work, change_output;
  double oua, hua; /* tmp */
  int pattern;
```

```
Receive (change_input); /* dummy */
change_work = 0.0; /* dummy */
for (j = 0; j < NrOutput; j++) {
  /* calcul eroare in output si actualizare ponderi bias */
  change_output = change_work + change_input;
  Send (change_output);
  Receive (change_input);
  change_work = 0.0;
  for (p = 0; p < BatCnt; p++) {
    pattern = (p + batch_start) % PattCnt;
    oua = output_act[p][j];
    output_err[p][j] = target_pattern[pattern][j] - oua;
    output_dlt[p][j] = output_err[p][j] * oua * (1.0 - oua);
    change_work += output_dlt[p][j];
}
change_output = change_work + change_input;
Send (change_output);
Receive (weight_input); Receive (change_input);
change_work = 0.0; /* dummy */
for (i = 0; i < NrHidden; i++) {
  for (p = 0; p < BatCnt; p++) {
    hidden_err[p][i] = 0.0;
  }
  for (j = 0; j < NrOutput; j++) {
    /* calcul actualizari ponderi si eroare in hidden */
    weight_work = weight_input;
    weight_output = weight_input;
    change_output = change_work + change_input;
    Send (weight_output); Send (change_output);
    Receive (weight_input); Receive (change_input);
    change_work = 0.0;
    for (p = 0; p < BatCnt; p++) {
      hidden_err[p][i] += output_dlt[p][j] * weight_work;
      change_work += output_dlt[p][j] * hidden_act[p][i];
    }
  }
}
weight_output = weight_input;
cange_output = change_work + change_input;
Send (weight_output); Send (change_output);
Receive (change_input); /* dummy */
change_work = 0.0; /* dummy */
for (j = 0; j < NrHidden; j++) {
  /* calcul delta in hidden */
  change_output = change_work + change_input;
  Send (change_output);
  Receive (change_input);
  change_work = 0.0;
  for (p = 0; p < BatCnt; p++) {
    hua = hidden_act[p][j];
    hidden_dlt[p][j] = hidden_err[p][j] * hua * (1.0 - hua);
    change_work += hidden_dlt[p][j];
  }

  for (i = 0; i < NrInput; i++) {
    /* calcul actualizari ponderi */
    change_output = change_work + change_input;
    Send (change_output);
```

```
        Receive (change_input);
        change_work = 0.0;
        for (p = 0; p < BatCnt; p++) {
          pattern = (p + batch_start) % PattCnt;
          change_work +=
                  hidden_dlt[p][j] * input_pattern[pattern][i];
        }
      }
    }
    Send (change_output);
  } /* calculate_weight_changes() */
```

În acest fel, la sfârşitul unei iteraţii, *administratorul* va deţine gradientul total, va fi în măsură să calculeze valorile actualizărilor ponderilor din reţeaua neuronală şi va putea relua procesul de învăţare cu noile valori ale ponderilor.

4.4 Rezultate experimentale

S-a exprimat eficienţa ca raportul dintre accelerarea obţinută şi cea ideală. Trebuie subliniat aici faptul că *între* actualizările de ponderi eficienţa *trebuie* să fie 100%, întrucât în acest timp toate procesoarele lucrează independent unul faţă de altul, fiecare procesor evoluând exact în acelaşi mod ca şi în varianta secvenţială. Colectarea tuturor gradienţilor componenţi şi distribuirea ponderilor actualizate constituie sarcini suplimentare în comparaţie cu algoritmul secvenţial şi deci va fi nevoie de timp suplimentar. Mai mult, calculul noilor valori ale ponderilor *după* ce toţi gradienţii componenţi au fost colectaţi nu a fost deloc paralelizat, întrucât doar procesorul rădăcină (0) efectuează acest calcul. Ineficienţa generată de acest calcul secvenţial al ponderilor creşte cu numărul procesoarelor: cu cât mai multe procesoare sunt utilizate în execuţia algoritmului, cu atât mai multe procesoare vor fi inactive pe parcursul actualizării. Desigur, aceasta este o proprietate nefericită a oricărui algoritm paralel, mai ales atunci când în realitate calculul noilor valori ale ponderilor nu constituie o fracţiune prea mare din totalul volumului de calcul.

Pe scurt: Întrucât între actualizările de ponderi algoritmul se execută pe fiecare procesor exact în acelaşi mod ca şi la algoritmul secvenţial, se prevede ca reducerea de eficienţă care ar fi observată la algoritmul paralel să fie determinată în primul rând de circumstanţele legate de actualizarea ponderilor.

În continuare sunt prezentate cele trei platforme hardware utilizate în acest studiu pentru implementarea algoritmilor distribuiţi de antrenare pentru backpropagation. În plus, vor fi discutate condiţiile în care a fost efectuate experimentele. În continuare vor fi prezentate rezultatele experimentelor efectuate.

Cele trei platforme hardware pe care s-au efectuat testele prezintă sisteme de operare de tip UNIX, pe care s-a utilizat limbajul de programare C, mediile PVM şi MPI, şi debugger-ul GNU (pe perioada dezvoltării aplicaţiilor). În vederea ridicării rezultatelor experimentale, s-a renunţat la facilităţile de depanare.

Pentru testare, simulatorul a fost executat pe configuraţii de reţele neuronale generice, cu numărul precizat de neuroni pe fiecare strat, şi cu câte un set de tipare de antrenare generate aleator. Nu s-au urmărit performanţele de învăţare din punctul de vedere al *calităţii învăţării*, ci doar accelerarea, respectiv eficienţa faţă de varianta corespunzătoare rulată secvenţial.

4.4.1 Platforme hardware

După cum se poate observa, procesul *administrator* şi cele *slave* ar putea să evolueze pe staţii de lucru diferite, chiar cu sisteme de operare diferite, atât timp cât se asigură compatibilitatea la nivelul informaţiei transmise. Pentru simplificare, s-a preferat însă omogenitatea reţelei de calculatoare pe care se efectuează simularea.

4.4.1.1 IBM RS/6000

Primul set de staţii de lucru pe care s-au efectuat experimentele din acest studiu constă în 12 staţii de lucru IBM RS/6000 Model 230, produse de firma IBM, având următoarele specificaţii:

- procesor POWER Single Chip la 45 MHz
- cache de 128 KB L2
- memorie 64 MB standard PS/2
- disc intern de 1 GB
- adaptor grafic POWER Gt1x
- Ethernet controller 10 BaseT
- sistem de operare AIX 4.1.5

Prin comparaţie cu celelalte seturi de staţii de test, aceste staţii IBM RS/6000 prezintă performanţe modeste, atât în privinţa procesorului, cât şi a memoriei şi reţelei de interconectare. De aceea, ele au fost utilizate doar în faza de dezvoltare a simulatorului.

4.4.1.2 SunBlade 150

Al doilea set de staţii de lucru constă în 16 staţii de lucru SunBlade 150, produse de firma Sun, având următoarele specificaţii:

- procesor UltraSPARC IIi pe 64 de biţi, la 650 MHz,
- cache de 512 KB L2 on-die,
- memorie de 1 GB,
- disc intern de 40 GB la 7200 rpm,
- placă de reţea Ethernet/Fast Ethernet, twisted-pair, 100 Base-T,
- sistem de operare Solaris 8.

După cum se observă, este vorba de procesoare mult mai rapide, şi, de asemenea, reţeaua în care sunt conectate asigură o viteză mai mare de transmisie, aspect foarte important în algoritmul nostru.

4.4.1.3 Linux

Al treilea set de staţii de lucru constă în 16 staţii de lucru HP/Compaq, având următoarele specificaţii:

- procesor Pentium la 2.8 GHz,
- memorie de 1 GB,
- disc intern SATA 160 GB,
- placă de reţea Ethernet/Fast Ethernet, twisted-pair, 100 Base-T,
- sistem de operare Linux.

Şi de data aceasta, este vorba de procesoare mai rapide, şi, de asemenea, reţeaua în care sunt conectate asigură o viteză mai mare de transmisie, aspect foarte important în algoritmul nostru.

Întrucât performanţele staţiilor IBM RS6000 erau mai slabe decât în celelalte două cazuri, aceste staţii au fost utilizate doar în faza de dezvoltare şi testare a simulatorului, iar rezultatele experimentale nu au fost incluse în această lucrare, un alt motiv fiind faptul că au fost disponibile doar 12 staţii.

4.4.2 Influenţa dimensiunii variabile a setului de antrenare

Consideraţiile de la începutul acestui capitol sugerează faptul că eficienţa va fi scăzută la utilizarea unui set de antrenare de dimensiune mică, şi că eficienţa va creşte dacă se măreşte dimensiunea setului de antrenare.

În tabelul 4.1 şi în figura 4.5 (a) SunBlade150, b) Linux) sunt prezentate rezultatele pentru o reţea neuronală 50-50-50 (50 de unităţi pe stratul de intrare, 50 pe cel ascuns, respectiv 50 de unităţi pe stratul de ieşire), cu varierea dimensiunii setului de antrenare, pe numărul maxim de staţii de lucru în fiecare caz (16). Simulatorul de reţea neuronală a fost rulat pentru 10 cicluri de învăţare, ceea ce înseamnă că ponderile au fost actualizate de acelaşi număr de ori. De remarcat faptul că, întrucât numărul de procesoare şi dimensiunea reţelei sunt fixe, timpul necesar efectuării unei actualizări de ponderi este de aşteptat să rămână acelaşi la toate execuţiile, independent de dimensiunea setului de antrenare utilizat.

Dimensiune batch	SunBlade150		Linux	
	normal	0comm	normal	0comm
25	28.06	48.04	26.02	44.03
50	38.98	72.11	37.06	70.01
100	59.02	85.07	58.00	84.50
150	70.04	88.03	69.11	88.14
200	75.98	88.52	75.51	88.65
250	77.01	89.04	78.03	89.22
300	79.03	89.25	79.13	90.23
350	79.19	89.23	79.34	90.23
400	79.31	89.26	79.47	90.24

Tabela 4.1: *Eficienţa funcţie de dimensiunea setului de antrenare*

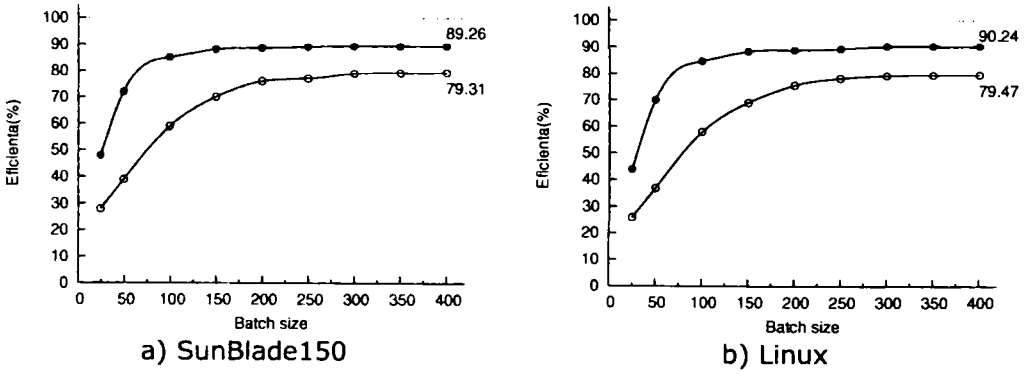


Figura 4.5: *Eficienţa funcţie de dimensiunea setului de antrenare*

S-a notat cu cerculeţe goale rezultatele execuţiilor normale ale algoritmului, adică atunci când procesoarele comunică între ele, cum este normal. Cerculeţele pline marchează rezultatele aceloraşi execuţii, însă fără a avea comunicaţie. De fiecare dată când un procesor ar dori să comunice, el a executat de fapt o instrucţiune NOP. Observând diferenţele dintre cele două grafice, se poate estima măsura în care comunicarea afectează eficienţa globală.

După cum era de aşteptat, eficienţa algoritmului este foarte scăzută pentru dimensiunea cea mai mică a setului de învăţare, întrucât fiecare procesor prezintă în acest caz doar un singur tipar, după care se face actualizarea reţelei. Cu alte cuvinte,

ponderile sunt actualizate relativ des. La creşterea dimensiunii setului de învăţare timpul de execuţie devine tot mai dominat de timpul utilizat de prezentarea tiparelor şi calculul gradienţilor componenţi, întrucât numărul de actualizări de ponderi nu creşte. În acest fel, timpul utilizat pentru actualizarea ponderilor devine o porţiune tot mai mică din timpul total de execuţie, deci eficienţa creşte.

Cel de-al doilea grafic, cu cerculeţe pline, confirmă faptul că timpul necesar comunicaţiei pe parcursul fiecărei actualizări de ponderi, este în parte responsabil de eficienţa observată, sub cea optimă. Mai mult, timpii totali de execuţie arată faptul că timpul necesar comunicării este independent de dimensiunea setului de antrenare, după cum era de aşteptat.

Totuşi, graficele indică faptul că încă mai pot exista şi alte cauze ale ineficienţei. Unele dintre acestea ar putea fi generate de instrucţiunile suplimentare pentru determinarea succesorilor în inel, a însumării gradienţilor componenţi, cât şi pentru gestionarea comunicării.

4.4.3 Influenţa numărului variabil de procesoare

În continuare vor fi prezentate rezultatele execuţiei pe aceleaşi staţii, când problema este fixă şi se măreşte numărul de procesoare. Întrucât mărirea setului de antrenare a dus la creşterea eficienţei, este de aşteptat să poată fi utilizate mai multe procesoare pentru seturi mari de antrenare. Experimentele au fost efectuate utilizând trei seturi de antrenare de dimensiune *fixă*, cu o reţea $50 - 50 - 50$.

În tabelul tabelul 4.2 şi figura 4.6 (SunBlade150) şi respectiv tabelul 4.3 şi figura 4.7 (Linux) este prezentată relaţia dintre accelerare şi numărul de procesoare care execută algoritmul paralel. Se observă că pentru toate cele trei dimensiuni ale setului de antrenare (cu excepţia, poate, a celui mai mic) accelerarea nu a atins valoarea maximă posibilă chiar şi atunci când a fost utilizat numărul maxim de procesoare (16). Aceasta ar însemna, probabil, că s-ar putea obţine chiar şi o accelerare mai bună având la dispoziţie mai multe staţii.

Număr	Accelerarea			Eficienţa		
procesoare	400	600	800	400	600	800
1	1.00	1.00	1.00	100.00	100.00	100.00
2	1.51	1.88	1.95	75.50	94.00	97.50
3	2.08	2.60	2.84	69.33	86.67	94.67
4	2.70	3.20	3.57	67.65	80.00	89.25
5	3.30	3.95	4.40	66.00	79.00	88.00
6	3.91	4.64	5.20	65.17	77.33	86.67
7	4.38	5.35	6.07	62.57	76.42	86.65
8	4.97	6.10	6.93	62.12	76.25	86.63
9	5.50	6.84	7.79	61.11	76.07	86.60
10	5.91	7.49	8.50	59.10	74.90	85.00
11	6.30	8.12	9.19	57.27	73.82	83.55
12	6.87	8.60	9.95	57.25	71.67	82.92
13	7.18	9.08	10.41	55.23	69.85	80.08
14	7.50	9.61	10.98	53.57	68.64	78.43
15	7.95	10.01	11.42	53.00	66.73	76.13
16	8.20	10.50	12.07	51.25	65.63	75.43

Tabela 4.2: *Accelerarea – trei dimensiuni ale setului de antrenare: SunBlade*

Se mai poate observa, însă, că pe măsură ce se utilizează tot mai multe procesoare, eficienţa scade (figura 4.6b, 4.7b). Pe baza rezultatelor obţinute în secţiunea anterioară, se poate conchide că există o explicaţie: pe măsură ce creşte numărul de procesoare, fiecare procesor are de prelucrat o porţiune tot mai mică din setul de antrenare. Cu alte cuvinte, setul de antrenare per procesor scade, determinând

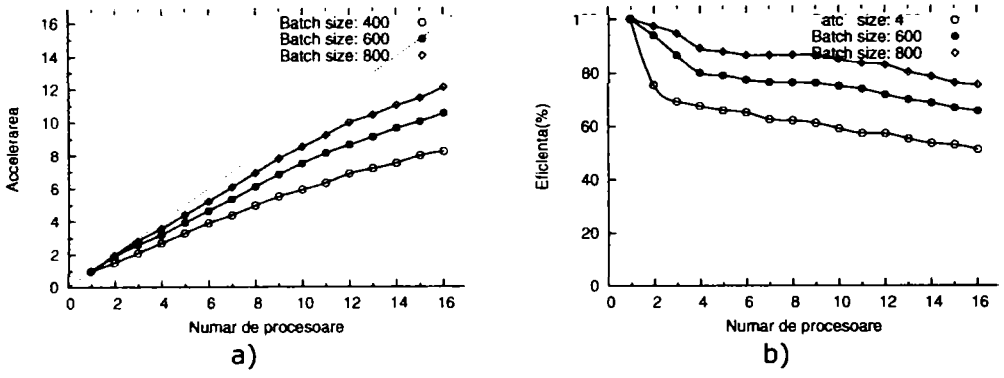


Figura 4.6: *Accelerarea – trei dimensiuni ale setului de antrenare: SunBlade*

Număr procesoare	Accelerarea			Eficienţa		
	400	600	800	400	600	800
1	1	1	1	100	100	100
2	1.59	1.91	1.96	79.50	95.50	98.00
3	2.15	2.65	2.86	71.66	88.33	95.33
4	2.75	3.27	3.61	68.75	81.75	90.25
5	3.3	3.96	4.44	66.00	79.20	88.80
6	3.87	4.63	5.21	64.50	77.16	86.83
7	4.31	5.31	6.01	61.57	75.85	85.85
8	4.90	6.09	6.88	61.25	76.12	86.00
9	5.0	6.80	7.73	55.55	75.55	85.88
10	5.83	7.43	8.43	58.30	74.30	84.30
11	6.22	8.09	9.11	56.54	73.54	82.81
12	6.8	8.55	9.88	56.66	71.25	82.33
13	7.09	9.02	10.35	54.53	69.38	79.61
14	7.41	9.54	10.90	52.92	68.14	77.85
15	7.88	9.87	11.35	52.53	65.80	75.66
16	8.11	10.32	11.99	50.68	64.50	74.94

Tabela 4.3: *Accelerarea – trei dimensiuni ale setului de antrenare: Linux*

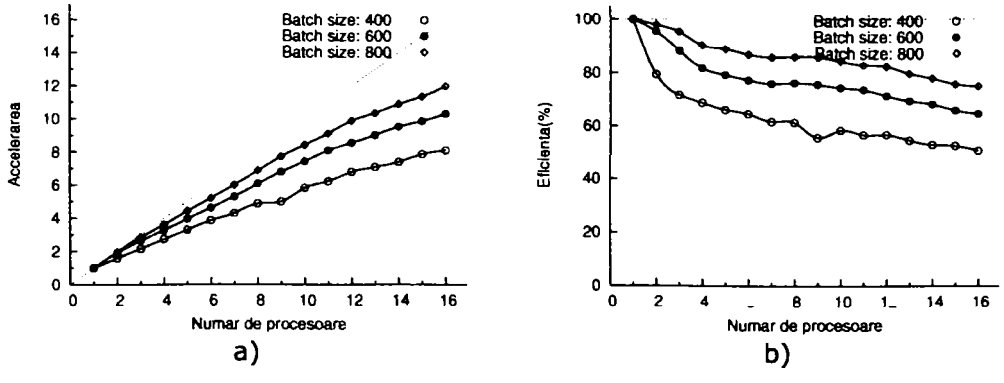


Figura 4.7: *Accelerarea – trei dimensiuni ale setului de antrenare: Linux*

scăderea eficienţei după cum s-a demonstrat în secţiunea anterioară. Acest efect poate fi observat şi prin simpla comparaţie a celor trei grafice din figura 4.6b, 4.7b.

Totuşi, eficienţa depinde nu numai de dimensiunea setului de antrenare per procesor, ci în aceeaşi măsură şi de numărul de procesoare, deci trebuie să mai existe şi alte cauze pentru scăderea eficienţei, întrucât aceasta depinde nu numai de dimensiunea setului de antrenare per procesor, ci şi de numărul de procesoare. Analizând figura 4.6b, 4.7b, se observă că pentru acelaşi set de antrenare per procesor, eficienţa scade dacă se măreşte numărul de procesoare. Aceasta se datorează probabil timpului necesar colectării gradienţilor componenţi şi distribuirii ponderilor actualizate.

4.4.4 Set de antrenare variabil cu numărul de procesoare

A treia clasă de experimente care s-au efectuat examinează efectele scalării dimensiunii setului de antrenare cu numărul de procesoare utilizate la execuţia algoritmului. Investigaţiile au fost efectuate cu scopul de a determina dacă o creştere a numărului de sub-probleme (tipare în set) va permite să fie mărit corespunzător numărul de procesoare fără a reduce eficienţa.

Reţeaua testată a fost una $50-50-50$, deci cu câte 50 de unităţi pe fiecare strat. Numărul de tipare din seturile de antrenare – generate aleator – a fost ales astfel încât fiecare procesor să proceseze câte 1, 2, 10, respectiv 20 de tipare.

În tabelul tabelul 4.4 şi figura 4.8 (SunBlade) şi respectiv tabelul 4.5 şi figura 4.9 (Linux) dimensiunea setului de antrenare *per procesor* este fixă (în fiecare dintre cele patru serii de experimente). Aceasta înseamnă că, între actualizările de ponderi, fiecare procesor prezintă un număr fix de tipare, independent de numărul de procesoare.

Număr procesoare	Accelerarea				Eficienţa			
	1	2	10	20	1	2	10	20
1	1.00	1.00	1.00	1.00	100.00	100.00	100.00	100.00
2	1.19	1.33	1.80	1.88	59.50	66.50	90.05	94.04
3	1.23	1.62	2.39	2.75	41.00	54.00	79.67	91.67
4	1.50	1.80	3.00	3.52	37.50	45.00	75.00	88.00
5	1.60	2.00	3.69	4.34	32.00	40.00	73.80	86.80
6	1.70	2.23	4.35	5.16	28.33	37.17	72.76	86.00
7	1.80	2.48	5.02	5.98	25.71	35.42	71.71	85.43
8	1.90	2.71	5.72	6.68	23.75	33.88	71.58	83.50
9	2.02	2.93	6.43	7.40	22.44	32.56	71.44	82.22
10	2.11	3.05	7.01	8.11	21.10	30.50	70.10	81.10
11	2.19	3.28	7.55	8.72	19.91	29.82	68.64	79.27
12	2.28	3.51	8.08	9.33	19.00	29.25	67.33	77.75
13	2.39	3.62	8.53	9.95	18.38	27.85	65.62	76.54
14	2.51	3.75	8.98	10.49	17.93	26.79	64.14	74.93
15	2.62	3.88	9.41	10.96	17.47	25.87	62.73	73.07
16	2.70	4.00	9.71	11.41	16.88	25.00	60.69	71.31

Tabela 4.4: *Set de antrenare fix per procesor: SunBlade*

În figurile 4.8b, 4.9b se observă că pentru un număr mare de procesoare, dacă se poate mări setul de antrenare, numărul de procesoare afectate problemei poate fi mărit corespunzător, fără scăderea prea mare a eficienţei.

În comparaţie cu figurile 4.6, 4.7, se poate observa că graficele eficienţei din figurile 4.8 şi 4.9 sunt foarte similare. De fapt, ele sunt aproape identice ca formă. Aceasta confirmă ceea ce s-a enunţat deja, şi anume faptul că acest algoritm paralel pierde din eficienţă în principal pe parcursul actualizării ponderilor.

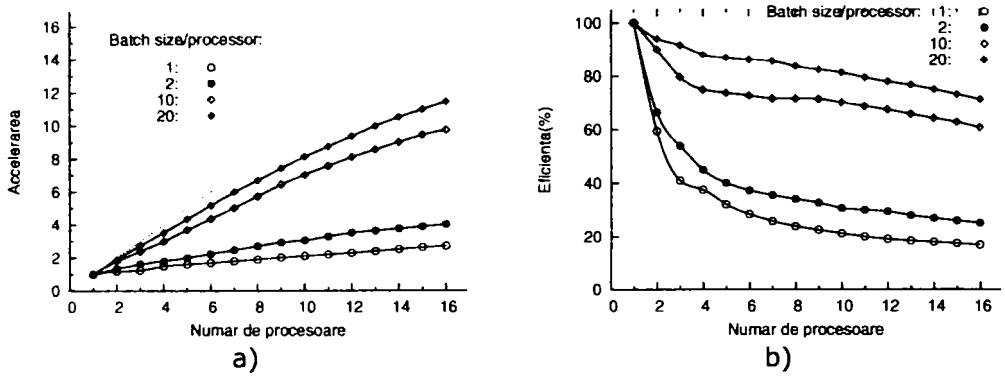


Figura 4.8: *Set de antrenare fix per procesor: SunBlade*

Număr procesoare	Accelerarea				Eficienţa			
	1	2	10	20	1	2	10	20
1	1.00	1.00	1.00	1.00	100.00	100.00	100.00	100.00
2	1.16	1.31	1.78	1.85	58.00	65.50	89.00	92.50
3	1.20	1.59	2.36	2.71	40.00	53.00	78.66	90.33
4	1.47	1.78	2.98	3.49	36.75	44.50	74.50	87.25
5	1.59	1.98	3.67	4.30	31.80	39.60	73.40	86.00
6	1.68	2.20	4.31	5.15	28.00	36.66	71.83	85.83
7	1.78	2.44	4.99	5.94	25.42	34.85	71.28	84.85
8	1.89	2.67	5.68	6.66	23.62	33.37	71.00	83.25
9	2.0	2.90	6.40	7.40	22.22	32.22	71.11	82.22
10	2.10	3.00	6.98	8.08	21.00	30.00	69.80	80.80
11	2.17	3.23	7.49	8.67	19.72	29.36	68.09	78.81
12	2.24	3.48	8.07	9.31	18.66	29.00	67.25	77.58
13	2.34	3.58	8.49	9.90	18.00	27.53	65.30	76.15
14	2.49	3.70	8.95	10.42	17.78	26.42	63.92	74.42
15	2.59	3.85	9.38	10.90	17.26	25.66	62.53	72.66
16	2.68	3.97	9.67	11.37	16.75	24.81	60.43	71.06

Tabela 4.5: *Set de antrenare fix per procesor: Linux*

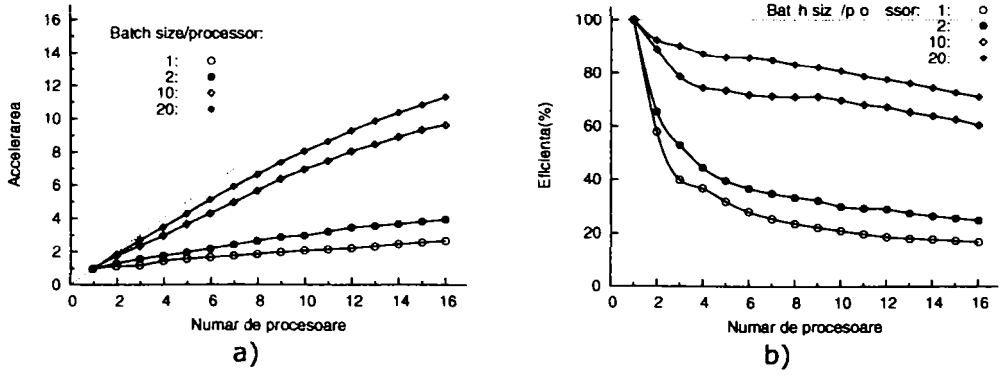


Figura 4.9: *Set de antrenare fix per procesor: Linux*

Pe baza acestor rezultate se poate decide dacă ar fi util să se mai adauge procesoare în cazul unei probleme date. Cu alte cuvinte, în ce măsură poate fi crescută accelerarea prin adăugarea de procesoare. Crescând numărul de procesoare cu un factor oarecare va determina o creştere a accelerării numai dacă eficienţa scade cu un factor mai mic sau cel mult egal, ca rezultat al procesoarelor suplimentare.

Ar fi ideal dacă s-ar putea formula o regulă privitor la cât de multe procesoare pot fi utilizate pentru o problemă specifică. Un exemplu ar fi, să zicem, să se găsească o relaţie între numărul de procesoare şi dimensiunea setului de antrenare. Aceasta ar însemna că atât timp cât fiecare procesor are de prezentat *cel puţin* un anumit număr de tipare, accelerarea va creşte la adăugarea de alte procesoare. În general, aceasta ar fi posibil dacă eficienţa algoritmului ar depinde numai de dimensiunea setului de antrenare per procesor, şi nu ar depinde de însăşi numărul de procesoare (în acest caz, graficele eficienţei din figurile 4.8b şi 4.9b, ar fi fost linii drepte, orizontale). După cum se poate observa, acest algoritm paralel nu prezintă aceste proprietăţi şi, într-adevăr, se pare că nu se poate formula o regulă generală.

4.5 Concluzii

Acest capitol prezintă un simulator pentru antrenarea cu algoritmul backpropagation, care utilizează strategia de paralelizare cu partiţionarea datelor. Pornind de la o variantă simplă, în care singura modificare faţă de varianta secvenţială constă în utilizarea şi însumarea unor *gradienţi parţiali*, a fost descrisă şi implementată o variantă avansată, în care s-a urmărit minimizarea necesarului de memorie – oarecum excesiv în varianta simplă – şi au fost efectuate o serie de experimente în vederea determinării factorilor care afectează performanţele simulatorului, comparativ cu varianta secvenţială.

S-a demonstrat că paralelizarea algoritmului backpropagation este posibilă, într-un mod foarte simplu, utilizând strategia de partiţionare a datelor. De asemenea, experimentele au demonstrat că eficienţa creşte la mărirea dimensiunii setului de tipare de antrenare prezentate reţelei între actualizările de ponderi. Aceasta înseamnă că dacă pot fi utilizate seturi de antrenare foarte mari, este posibil să se obţină o accelerare substanţială utilizând acest tip de paralelizare.

Contribuţiile acestui capitol sunt:

1. Dezvoltarea şi implementarea unui algoritm de paralelizare cu partiţionarea datelor. Un dezavantaj major al acestui algoritm îl constituie necesarul excesiv de memorie, datorat faptului că fiecare procesor trebuie să păstreze o copie a întregii reţele neuronale. O asemenea redundanţă ar trebui evitată într-un algoritm paralel, întrucât aceasta înseamnă că paralelizarea nu permite creşterea dimensiunii problemelor care pot fi procesate (din punctul de vedere al necesarului de memorie).

2. Dezvoltarea si implementarea unui algoritm avansat de paralelizare cu partiţionarea datelor, cu un proces *administrator* şi mai multe procese *slave*. Programul administrator transmite în reţea, către *slave*, pe rând, câte o pondere din reţeaua neuronală, urmând ca procesele *slave* să utilizeze această valoare a ponderii în propagarea tuturor tiparelor asignate. Pentru o îmbunatăţire a exploatării maşinilor, şi procesul administrator prelucrează un număr de tipare.

3. Evaluarea necesarului de memorie al algoritmilor prezentaţi. Algoritmul avansat de paralelizare prezentat, pe lângă faptul că permite scurtarea timpului de antrenare a unei reţele neuronale pentru seturi mari de tipare, mai permite şi procesarea unor reţele neuronale de dimensiuni mai mari decât valorile posibile pentru cazul secvenţial.

4. Studiul experimental al influenţei factorilor care pot afecta performanţele evoluţiei simulatorului într-o reţea de calculatoare. Concluzia care se poate deduce

este că performanţele simulatorului vor fi cu atât mai bune cu cât numărul de tipare de procesat pe fiecare maşină este mai mare. Din experimentele efectuate, s-au observat valori satisfăcătoare pentru accelerare, respectiv eficienţă atunci când au putut fi asigurate cel puţin zece tipare de antrenare per procesor. Investigaţii pe această temă au fost prezentate în [BaC03] şi [BCP07].

5 NetParSim – simulator paralel cu partiţionarea reţelei

În acest capitol se va trata dezvoltarea şi analiza unui algoritm care utilizează o strategie de partiţionare a reţelei pentru antrenarea unei reţele neuronale într-o reţea de calculatoare.

5.1 Construcţia algoritmului paralel

În această secţiune vor fi descrise bazele paralelizării, modul în care este divizată reţeaua între procesoare, topologia procesoarelor, iar în final va fi prezentat modul în care se face paralelizarea pasului înainte, respectiv a pasului înapoi.

5.1.1 Divizarea reţelei

Atunci când se doreşte să se distribuie unităţile şi ponderile pe un număr de procesoare, nu există prea multe alternative la dispoziţie. O variantă este divizarea reţelei orizontal, în trei porţiuni, astfel încât un procesor să conţină unităţile unui anumit strat. Aceasta nu e o idee prea bună, întrucât în mod normal o reţea neuronală conţine puţine straturi şi deci nu vor putea fi utilizate prea multe procesoare. Mai mult, dacă se insistă pe utilizarea unei anumite scheme de actualizare a tiparelor, această formă de paralelizare nici nu este posibilă, întrucât paşii înainte şi înapoi sunt secvenţiali — numai unităţile dintr-un singur strat efectuează calcule la un moment dat.

Reţeaua poate fi divizată vertical, astfel încât unităţile dintr-un strat sunt distribuite uniform pe procesoare. Fiecare procesor va conţine deci unităţi din toate straturile. Sunt posibile două modalităţi evidente de distribuire a ponderilor:

1. Toate ponderile pentru conexiunile care *intră* într-o unitate se memorează în procesorul care simulează acea unitate.
2. Toate ponderile pentru conexiunile care *ies* dintr-o unitate se memorează în procesorul care simulează acea unitate.

În realitate nu există o diferenţă între cele două strategii. În primul caz pasul înainte va fi simplu de paralelizat, în vreme ce pasul înapoi va fi mai dificil. În al doilea caz — invers. Aceste aspecte vor fi evidente pe parcursul dezvoltării paralelizării. În cadrul lucrării de faţă a fost aleasă prima strategie.

Toate unităţile sunt distribuite uniform pe procesoare, astfel încât toate procesoarele conţin acelaşi număr de unităţi de intrare, ascunse şi de ieşire, deşi nu este necesar ca numărul de procesoare să dividă numărul de unităţi de pe un strat. În acest caz, unele procesoare vor simula cu o unitate mai mult decât altele. Pentru simplificare, se va considera însă în continuare că numărul de unităţi de pe orice strat este divizibil cu numărul de procesoare.

Ponderile conexiunilor din reţea le distribuim în modul următor: ponderile pentru toate conexiunile care intră într-o unitate de ieşire sau ascunsă se memorează în procesorul care simulează acea unitate. Acest mod conduce la o partiţionare echilibrată a ponderilor. În Figura 5.1 se prezintă ponderile conţinute în procesorul 1.

Strategia de partiţionare a reţelei în vederea paralelizării a fost investigată în cadrul câtorva cercetări în domeniu. Pentru abordări similare, se pot consulta lucrările lui Ernoult – *"Performance of backpropagation on a parallel transputer-based machine"*, [22], Zhang, McKenna, Mesirov şi Waltz – *"The backpropagation algorithm on grid and hypercube architectures"*, [104] şi del Rey Millan şi Bofill – *"Learning by backpropagation: Computing in a systolic way"*, [20]. Spre deosebire de lucrările enumerate, care utilizează arhitecturi de transputere, respectiv hypercuburi, arhitecturi cu caracteristici specifice legate de modul de lucrul al procesoarelor şi modul în care ele sunt interconectate, teza de faţă propune utilizarea unei reţele de

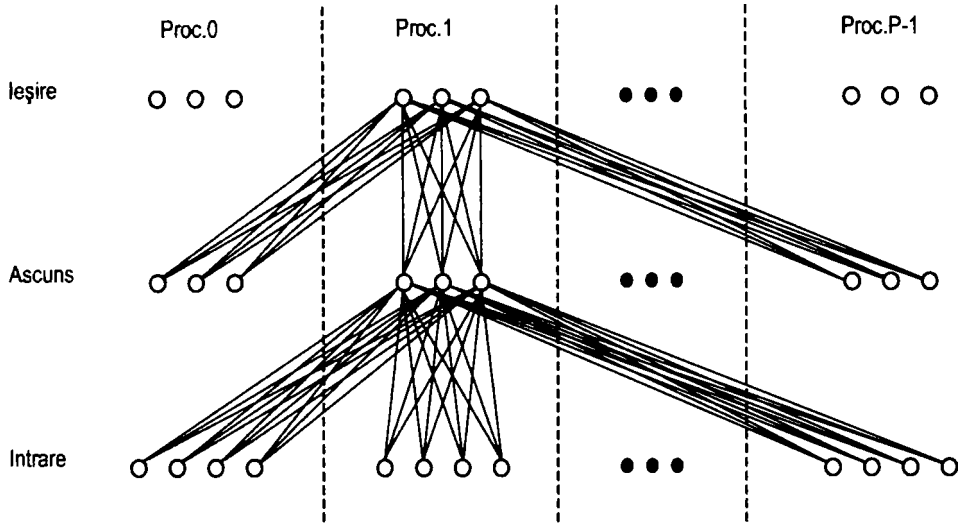


Figura 5.1: *Distribuirea unităţilor şi a ponderilor.*

calculatoare, uzuale într-un laborator universitar. Desigur, interconectarea maşinilor printr-o reţea ethernet a impus diferenţe în abordarea diferitelor aspecte, în special în cazul comunicaţiei, în vederea reducerii timpului necesar pentru transmiterea datelor de la un procesor la altul.

5.1.2 Topologia procesoarelor

A fost aleasă o conectare în inel. O configuraţie de inel este simplu de extins pentru a include mai multe elemente şi poate să conţină orice număr de procesoare. După cum se va vedea, este o topologie naturală pentru partiţionarea reţelei aşa cum e prezentată în figura 5.1. Probabil că ar prezenta interes şi alte topologii, dar acesta este un subiect pentru dezvoltări ulterioare.

Procesoarele sunt conectate în inel, aşa cum este ilustrat în figura 5.2. De notat direcţia legăturilor. Un procesor trimite date către predecesorul său şi recepţionează de la succesor.

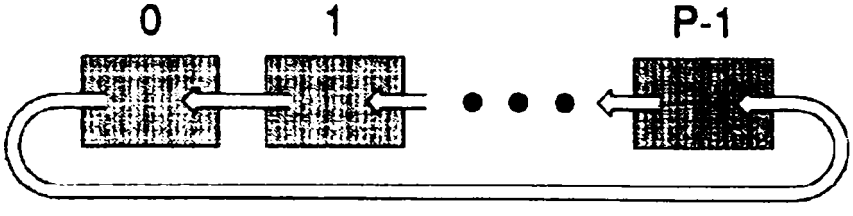


Figura 5.2: *Inel de procesoare.*

Chiar dacă fiecare procesor simulează doar o parte din unităţile de intrare şi cele ascunse, toate procesoarele trebuie să aibă capacitatea de a stoca activitatea tuturor unităţilor de intrare, respectiv ascunse, întrucât aceasta este necesară în ambii paşi,

cel înainte şi cel înapoi, din algoritmul de backpropagation. Memorând activitatea tuturor unităţilor se evită o serie de comunicaţii suplimentare.

Contrar paralelizărilor cu partiţionarea datelor descrise în capitolul anterior, aici toate procesoarele execută acelaşi program. Procesul administrator diferă de cele *slave* numai atunci când intervin operaţii de intrare/ieşire.

5.1.3 Notaţii

Înainte de a începe prezentarea paralelizării, este necesar să fie introduse o serie de notaţii

Mai întâi, câteva abrevieri convenabile. Mai devreme s-a definit P ca fiind numărul de procesoare, NI, NH şi NO ca fiind numărul de unităţi de pe stratul de intrare, ascuns şi de ieşire. Se notează cu PI, PH, şi PO — numărul de unităţi de intrare, ascunse şi de ieşire simulate de fiecare procesor. Fie O mulţimea de indici de unităţi de ieşire $(0, 1, ..., NO-1)$. Sub-seturile indicilor de unităţi de ieşire simulate de către procesoarele individuale vor fi:

$$\begin{array}{cccccc} O = & \{0,1,\ldots,PO-1, & PO,\ldots,2\cdot PO-1, & \ldots, & (P-1)\cdot PO,\ldots,P\cdot PO-1\} \\ & \underset{0}{O_{[0]}} & \underset{1}{O_{[1]}} & & \underset{P-1}{O_{[P-1]}} \end{array}$$

Indicii pentru unităţile ascunse şi cele de intrare se definesc similar.

Se notează cu a_j^H activarea unităţii ascunse j. Vectorul activărilor tuturor unităţilor ascunse va fi $\overline{a}^H$, iar $\overline{a}_{[p]}^H$ este sub-vectorul activărilor unităţilor ascunse memorate în procesorul p. Aceasta se poate scrie ca:

$$\begin{array}{cccccc} \overline{a}^H = & (a_0^H, a_1^H, \ldots, a_{PH-1}^H & a_{PH}^H, \ldots, a_{2\cdot PH-1}^H, & \ldots, & a_{(P-1)\cdot PH}^H, \ldots, a_{P\cdot PH-1}^H\} \\ & \underset{0}{\overline{a}_{[0]}^H} & \underset{1}{\overline{a}_{[1]}^H} & & \underset{P-1}{\overline{a}_{[P-1]}^H} \end{array}$$

Vectorii activărilor unităţilor de intrare, respectiv de ieşire, $\overline{a}^I$ şi $\overline{a}^O$ se definesc în mod similar. La fel pentru vectorii valorilor delta pentru unităţile ascunse, respectiv de ieşire, $\overline{\delta}^I$ şi $\overline{\delta}^O$. În sfârşit, intrarea de reţea pentru o unitate din stratul ascuns, respectiv de ieşire va fi net_j^H, respectiv net_k^O.

De notat faptul că simbolurile pentru intrarea din reţea, activare şi delta nu au un indice p (pentru tipar). Acesta nu este necesar, întrucât algoritmul pe care îl descriem corespunde metodei gradientului stohastic.

Ponderile conexiunilor care intră în stratul de ieşire de la unitatea ascunsă j la unitatea de ieşire k vor fi notate cu $w_{j\to k}^O$. $\overline{w}_{\to k}^O$ va fi vectorul ponderilor conexiunilor care intră în unitatea de ieşire k dinspre toate unităţile ascunse, iar $\overline{w}_{j\to}^O$ — vectorul ponderilor conexiunilor spre toate unităţile de ieşire de la unitatea ascunsă j. La fel ca în cazul activărilor, se notează cu $\overline{w}_{[p]\to k}^O$ sub-vectorul de ponderi spre unitatea de ieşire k dinspre unităţile ascunse simulate de procesorul p, şi cu $\overline{w}_{j\to[p]}^O$ sub-vectorul de ponderi care duc spre unităţile de ieşire simulate de către procesorul p de la unitatea ascunsă j. Ponderile conexiunilor dinspre stratul de intrare către stratul ascuns se exprimă în mod similar.

5.1.4 Paralelizarea

Paralelizarea propusă este formată din şapte paşi, doi paşi pentru pasul înainte şi cinci paşi pentru pasul înapoi. Întrucât reţeaua neuronală este distribuită în reţeaua de calculatoare, au fost necesare o serie de modificări în ordinea efectuării calculelor în algoritmul clasic de backpropagation, având ca scop adaptarea la comunicaţia în reţeaua de calculatoare şi minimizarea timpului necesar transmisiei de date. Pentru evidenţierea modificărilor efectuate, au fost rescrise formulele de calcul, grupând

termenii pe procesoare la calculul valorilor δ pentru stratul ascuns, calcule efectuate pe parcursului pasului al patrulea.

5.1.4.1 Pasul întâi

Primul pas calculează activarea unităţilor ascunse. De fapt, el constă din doi sub-paşi, primul fiind distribuirea activării unităţilor de intrare, astfel încât toate procesoarele vor avea acces la activarea tuturor unităţilor de intrare, iar în al doilea sub-pas se efectuează calculul activării unităţilor ascunse.

Vectorul activării unităţilor de intrare, $\overline{a}^I$, este distribuit astfel încât procesorul p deţine sub-vectorul $\overline{a}^I_{[p]}$. Distribuirea tuturor sub-vectorilor către toate procesoarele este uşor de efectuat într-un inel: fiecare procesor îşi transmite propriul sub-vector către predecesorul său în inel. Apoi toate procesoarele recepţionează şi transmit sub-vectorii calculaţi de celelalte procesoare, până când deţin întregul vector al activării unităţilor de intrare. Aceasta necesită $P(P-1)$ comunicaţii. Operaţia este ilustrată în Figura 5.3.

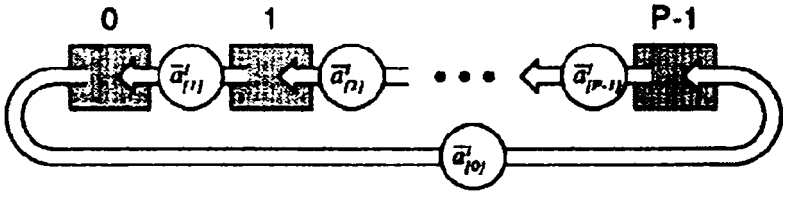


Figura 5.3: *Broadcast pentru activarea unităţilor de intrare.*

La terminarea acestei operaţii, toate procesoarele deţin întregul vector al activării unităţilor de intrare, $\overline{a}^I$. Procesorul p simulează unităţile ascunse cu indicii $H_{[p]}$ şi ponderile pentru toate conexiunile care intră în aceste unităţi sunt memorate în procesorul p. Procesorul p poate acum să calculeze intrarea de reţea pentru toate unităţile ascunse pe care le simulează:

$$net_j^H = \overline{a}^I \star \overline{w}^H_{i \to j} = \sum_{i=0}^{NI-1} a_i^I w^H_{i \to j}, \qquad \forall j \in H_{[p]} \tag{5.1}$$

unde operatorul $\star$ reprezintă înmulţire vectorială.

Pe baza intrării din reţea, se poate calcula activarea unităţilor ascunse:

$$a_j^H = f(net_j^H), \qquad \forall j \in H_{[p]} \tag{5.2}$$

unde f este funcţia nonliniară de activare. Calculele din cele două ecuaţii de mai sus pot fi efectuate de către toate procesoarele în paralel, fără comunicaţie între ele, întrucât toate variabilele care intervin sunt memorate local.

Partea de comunicaţie din pasul întâi nu poate fi evitată. Totuşi, comunicaţiile se pot efectua asincron şi în paralel cu calculele. Se va exploata această facilitate pentru a combina cei doi sub-paşi ai pasului întâi. Dacă se efectuează calculele în timp ce se transmit activările unităţilor de intrare, iar timpul de calcul este mai mare decât cel de transmisie, volumul mare de comunicaţii necesar în acest pas nu va avea importanţă.

Calculul intrării într-o unitate ascunsă, dat în ecuaţia 5.1, poate fi rescris în felul următor:

$$net_j^H = \sum_{i=0}^{NI-1} a_i^I w^H_{i \to j} = \sum_{q=0}^{P-1} \overline{a}^I_{[q]} \star \overline{w}^H_{[q] \to j}, \qquad \forall j \in H_{[p]} \tag{5.3}$$

Ultima expresie din ecuaţie este o sumă pentru toate procesoarele. Fiecare sub-vector $\overline{a}^I_{[q]}$ din sumă este memorat în câte un procesor.

Combinarea celor doi sub-paşi începe prin transmisia de către fiecare procesor a acestui sub-vector, către procesorul precedent din inel, după cum se vede în Figura 5.3. În timpul acestei transmisii, toate procesoarele îşi calculează contribuţia locală la unităţile ascunse pe care le simulează. Pentru toate unităţile ascunse simulate de procesorul p, se calculează:

$$\overline{a}^I_{[p]} \star \overline{w}^H_{[p]\to j}, \qquad \forall j \in H_{[p]} \tag{5.4}$$

La terminarea acestui calcul, procesorul recepţionează sub-vectorul de la succesorul său şi îl transmite către procesorul predecesor. Acest sub-vector recepţionat de procesorul p conţine activările unităţilor de intrare ale procesorului $p+1$, iar acum procesorul p poate să calculeze termenii:

$$\overline{a}^I_{[p+1]} \star \overline{w}^H_{[p+1]\to j}, \qquad \forall j \in H_{[p]} \tag{5.5}$$

care se însumează la termenii calculaţi în ecuaţia 5.4.

Această activitate de calcul şi transmisie în paralel continuă până la completarea calculului lui net^H_j pentru toţi $j \in H_{[p]}$. Se aplică apoi funcţia de activare pentru a obţine activările unităţilor ascunse.

Activările unităţilor de intrare sunt necesare din nou în pasul şase, când se calculează actualizările ponderilor. Ca urmare, este necesar să se memoreze sub-vectorii activărilor unităţilor de intrare, pentru utilizare ulterioară.

5.1.4.2 Pasul al doilea

Al doilea pas calculează activarea unităţilor de ieşire. Acest pas este în mare măsură similar pasului întâi. Calculul intrării într-o unitate de ieşire poate fi rescris după cum urmează:

$$net^O_k = \sum_{j=0}^{NH-1} a^H_j w^O_{j\to k} = \sum_{q=0}^{P-1} \overline{a}^H_{[q]} \star \overline{w}^O_{[q]\to k}, \qquad \forall k \in O_{[p]} \tag{5.6}$$

Ca şi la pasul întâi, fiecare sub-vector $\overline{a}^H_{[q]}$ din această sumă este memorat în procesoare separate. Este necesar din nou un broadcast, de data aceasta pentru activările unităţilor ascunse.

5.1.4.3 Pasul al treilea

Al treilea pas calculează valorile delta pentru unităţile de ieşire şi aceasta se poate efectua exclusiv pe baza datelor locale. Valoarea delta pentru unitatea de ieşire simulată de procesorul p, cu indicele k se calculează pe baza relaţiei 5.7:

$$\delta^O_k = (t_k - a^O_k) a^O_k (1 - a^O_k), \forall k \in O_{[p]} \tag{5.7}$$

5.1.4.4 Pasul al patrulea

În acest pas se calculează valorile delta pentru unităţile ascunse. Pentru calculul valorii delta a unei unităţi ascunse, δ^H_j, în procesorul p se calculează:

$$\delta^H_j = a^H_j (1 - a^H_j)(\overline{\delta}^O \star \overline{w}^O_{j\to}) = a^H_j (1 - a^H_j) \sum_{k=0}^{NO-1} \delta^O_k w^O_{j\to k}, \forall j \in H_{[p]} \tag{5.8}$$

Pasul al patrulea este foarte asemănător cu pasul doi, întrucât în esenţă constă dintr-un produs vectorial. Totuşi, aici ponderile nu sunt memorate la locul potrivit.

Atunci când procesorul p calculează un produs vectorial el utilizează ponderile $\overline{w}^O_{j\rightarrow}$ pentru toţi $j \in H_{[p]}$. Aceste ponderi, însă, nu sunt memorate în procesorul p. Ele sunt partiţionate în mod egal între toate procesoarele, iar distribuirea ponderilor este mult prea mare consumatoare de timp.

Ecuaţia 5.8 poate fi rescrisă astfel încât să exprimăm sumele pe procesoare:

$$\delta^H_j = a^H_j(1-a^H_j)\sum_{k=0}^{NO-1}\delta^O_k w^O_{j\rightarrow k} = a^H_j(1-a^H_j)\sum_{q=0}^{P-1}\overline{\delta}^O_{[q]} \star w^O_{j\rightarrow[q]}, \qquad \forall j \in H_{[p]} \tag{5.9}$$

Ultima expresie este o sumă pe toate procesoarele. Fiecare termen din sumă poate fi calculat numai de către procesorul care memorează ponderile şi valorile delta respective. Ca urmare, dacă $j \in H_{[a]}$ este indicele unei unităţi ascunse simulată de către procesorul a, procesorul b poate calcula termenii $\overline{\delta}^O_{[b]} \star \overline{w}^O_{j\rightarrow[b]}$ necesari procesorului a în calculul lui $\delta^H_{[a]}$. Se notează cu $a_{[b]}$ toţi aceşti termeni:

$$a_{[b]} = \left\{\overline{\delta}^O_{[b]} \star \overline{w}^O_{j\rightarrow[b]}\right\}_{\forall j \in H_{[a]}} \tag{5.10}$$

Ca urmare, $a_{[b]}$ este un *set* calculat de procesorul b, necesar procesorului a.

Toate procesoarele necesită câte un asemenea set din partea celorlalte procesoare. În total sunt $P(P-1)$ asemenea seturi. Toate aceste seturi sunt diferite, deci nu există un mod simplu să fie transmise între procesoare, adică nu se poate efectua în exact acelaşi mod ca şi în paşii 1 şi 2.

Există totuşi o cale de a efectua transmisia într-un mod similar celui din pasul 2. Pe baza faptului că seturile nu sunt total independente, toate fiind parte a unei însumări, se va demonstra în continuare că este posibil să fie transmise şi însumate seturile astfel încât toate transmisiile pot fi efectuate în $P-1$ paşi secvenţiali (cu P transmisii paralele/asincrone în fiecare pas), la fel ca şi în paşii 1 şi 2.

Generalizând ecuaţia 5.10:

$$a_{[b,\ldots,c]} = \left\{\overline{\delta}^O_{[b]} \star \overline{w}^O_{j\rightarrow[b]} + \cdots + \overline{\delta}^O_{[c]} \star \overline{w}^O_{j\rightarrow[c]}\right\}_{\forall j \in H_{[a]}} \tag{5.11}$$

În această ecuaţie se însumează un număr de seturi calculate de procesoarele de la b la c, necesare procesorului a. Suma este o parte a întregii sume utilizată în calculul lui $\delta^H_{[a]}$.

Figura 5.4 ilustrează operaţiile de efectuat:

1. Procesorul 0 calculează setul său pentru procesorul 1, notat cu $1_{[0]}$. Procesorul 1 calculează setul său pentru procesorul 2 — $2_{[1]}$, şi aşa mai departe.
2. Calculul şi transmisia se efectuează în paralel. Procesorul 0 calculează $2_{[0]}$ în timp ce transmite $1_{[0]}$ către procesorul $P-1$. Procesorul 1 calculează $3_{[1]}$ în timp ce transmite $2_{[1]}$ către procesorul 0 şi aşa mai departe.
3. Procesorul 0 recepţionează $2_{[1]}$, îl adună la setul $2_{[0]}$ pe care tocmai l-a calculat, obţinând $2_{[0,1]}$, care este următorul set de transmis. Procesorul 1 recepţionează $3_{[2]}$, îl adună la setul $3_{[1]}$ pe care tocmai l-a calculat, obţinând $3_{[1,2]}$. La fel pentru celelalte procesoare.

În acest moment se repetă paşi similari paşilor 2 şi 3, până când toate procesoarele deţin întregul set necesar lor, adică procesorul p deţine setul

$$p_{[0,\ldots,P-1]} = \sum_{q=0}^{P-1}\overline{\delta}^O_{[q]} \star \overline{w}^O_{j\rightarrow[q]}, \qquad \forall j \in H_{[p]} \tag{5.12}$$

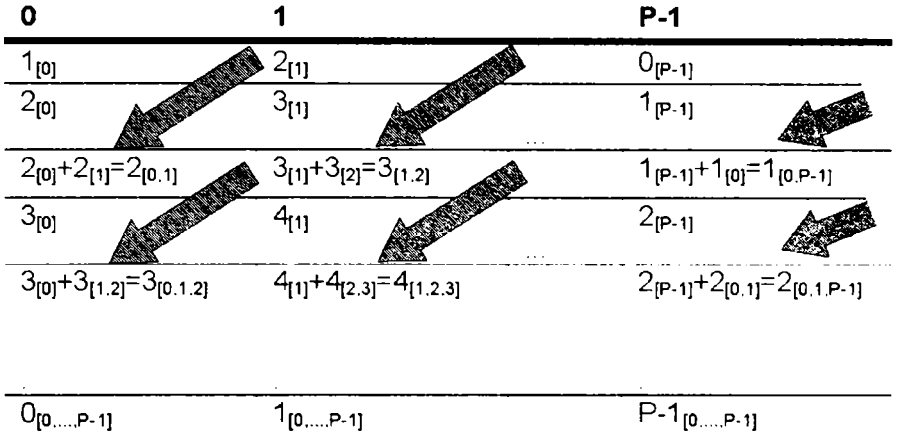


Figura 5.4: *Calculul valorilor delta pentru unităţile ascunse.*

Aceasta este exact suma din ultima expresie din ecuaţia 5.9 pentru procesorul p. Acum este posibil ca procesorul p să calculeze $\delta^H_{[p]}$.

5.1.4.5 Pasul al cincilea

Al cincilea pas calculează actualizările de ponderi pentru conexiunile dintre stratul ascuns şi cel de ieşire. Ecuaţia este:

$$\Delta w^O_{j\to k}(n+1) = \eta \delta^O_k a^H_j + \alpha \Delta w^O_{j\to k}(n) \tag{5.13}$$

Întrucât activările unităţilor ascunse, calculate în pasul doi, au fost memorate, este posibil ca procesoarele să calculeze toate actualizările de ponderi exclusiv pe baza datelor locale.

5.1.4.6 Pasul al şaselea

Acest pas este similar pasului cinci, cu excepţia faptului că se calculează actualizările pentru ponderile conexiunilor dintre stratul de intrare şi cel ascuns. Ecuaţia de calcul este:

$$\Delta w^H_{i\to j}(n+1) = \eta \delta^H_j a^I_i + \alpha \Delta w^H_{i\to j}(n) \tag{5.14}$$

Din nou, întrucât activările unităţilor de intrare calculate în pasul întâi au fost memorate, este posibil ca procesoarele să calculeze toate actualizările de ponderi exclusiv pe baza datelor locale.

5.1.4.7 Pasul al şaptelea

Acest pas are drept scop efectuarea actualizărilor de ponderi. Din nou, şi aceasta se poate efectua numai pe baza datelor locale, fără nici o comunicaţie, deci în paralel.

5.1.5 Memorarea tiparelor de intrare şi de ieşire

În funcţie de dimensiunea tiparelor de intrare/ieşire, se pot utiliza strategii diferite de memorare a acestor date în procesoare. Dacă întregul set de tipare de intrare/ieşire este mic, toate perechile de tipare pot fi memorate în întregime în toate procesoarele, astfel încât toate procesoarele să deţină copii identice. În acest caz, pasul întâi al

algoritmului devine foarte simplu. Întrucât tiparele de intrare (activările unităţilor de intrare) sunt cunoscute tuturor procesoarelor, nu mai este necesar un broadcast.

Dacă setul de tipare de intrare/ieşire este prea mare pentru a-l plasa pe toate procesoarele, ele vor fi distribuite uniform între procesoare. După cum s-a menţionat în secţiunea 5.1.4, de fapt se distribuie unităţile de intrare în mod uniform între toate procesoarele, ca şi unităţile ascunse şi cele de ieşire, iar fiecare procesor va deţine din tiparele de intrare exact porţiunea care îi este asociată pentru unităţile de intrare pe care le simulează. Similar pentru tiparele de ieşire.

5.1.6 Îmbunătăţiri ale algoritmului

Dacă timpul de comunicaţie din pasul patru constituie o problemă, o îmbunătăţire evidentă a acestui algoritm în şapte paşi este să se combine paşii patru şi cinci, întrucât calculul actualizărilor de ponderi dintre straturile ascuns şi cel de ieşire din pasul cinci poate fi efectuat odată cu terminarea pasului trei şi deci în paralel cu pasul patru. Actualizarea ponderilor conexiunilor dintre stratul ascuns şi cel de ieşire (parte din pasul şapte) poate fi efectuată de asemenea în pasul patru.

5.2 Analiza algoritmului

În această secţiune se va face o analiză a factorilor care influenţează comportarea algoritmului. Testele care se vor efectua vor fi pe reţele de dimensiuni convenabile; ele nu vor fi reţele din aplicaţii reale şi ca urmare în realitate nu se va efectua nici o antrenare.

Dintre cei şapte paşi ai algoritmului, comunicaţii au loc doar în paşii unu, doi şi patru. Cu excepţia unor activităţi suplimentare, pentru comutarea între procese şi pentru calculul de indici, probabil că numai comunicaţia din aceşti trei paşi ar putea împiedica atingerea maximumului de *speed-up* faţă de versiunea secvenţială.

5.2.1 Necesarul de memorie

Un avantaj al acestui algoritm îl constituie necesarul scăzut de memorie. Ponderile, activările şi valorile delta din reţea sunt distribuite uniform între procesoare. Totuşi, valorile de activare pentru toate unităţile de intrare şi pentru cele ascunse sunt memorate în toate procesoarele, întrucât este vorba despre valori utilizate de două ori pe parcursul algoritmului. Activările unităţilor de intrare sunt utilizate în pasul unu şi şapte, iar activările unităţilor ascunse sunt utilizate în pasul doi şi pasul cinci.

Necesarul de memorie al fiecărui procesor, conform notaţiilor din secţiunea 5.1.3 (măsurat în număr de valori reale pe 64 de biţi — *double*) va fi:

$$NI + NH + PH + 2PO + 2((NI + 1) \cdot PH + (NH + 1) \cdot PO) \tag{5.15}$$

Atunci când reţelele simulate sunt de dimensiune mare, singura contribuţie semnificativă este cea a ponderilor.

5.2.2 Pasul înainte

Cei doi paşi care formează pasul înainte sunt în esenţă similari. Se va analiza în consecinţă doar unul dintre aceşti paşi. Întrucât s-a discutat pe larg primul pas în secţiunea 5.1.4, aici va fi analizat pe larg acest pas. Se va demonstra că timpul de calcul depinde liniar atât de numărul de unităţi ascunse simulate de fiecare procesor — PH, cât şi de numărul de unităţi de intrare simulate de fiecare procesor — PI. Timpul de comunicaţie depinde însă liniar numai de PI. Cu alte cuvinte, mărind PH este posibil să se mărească timpul de calcul fără să afecteze timpul de comunicaţie. Astfel, alegând un PH suficient de mare, timpul de comunicaţie poate să devină nesemnificativ în comparaţie cu timpul de calcul.

Procesorul p calculează intrarea din reţea pentru una din unităţile ascunse pe care le simulează conform ecuaţiei 5.3. Acest calcul se execută în paralel (asincron) cu broadcast-ul activării unităţilor de intrare, cu observaţia că transmisia trebuie să se facă cu prioritate maximă, pentru a termina cât mai repede iniţializarea şi a putea trece cât mai repede la efectuarea calculelor.

La fiecare iteraţie, calculul termenului $\overline{a}^I_{[q]} \star \overline{w}^H_{[q]\ ,j}$ conţine PI înmulţiri şi PI adunări. Timpul de calcul pentru fiecare iteraţie depinde deci liniar de PI. Întrucât calculul se efectuează pentru toate unităţile ascunse simulate de fiecare procesor, deci pentru toţi $j \in H_{[p]}$, timpul de calcul va depinde liniar şi de PH.

Situaţia din pasul doi este exact aceeaşi cu cea descrisă pentru pasul întâi. Pentru pasul doi timpul de calcul depinde liniar de PH şi PO, în vreme ce timpul de comunicaţie depinde liniar numai de PH.

Concluzia la consideraţiile de mai sus este că numărul de unităţi simulate de fiecare procesor, PI, PH şi respectiv PO determină calitatea paralelizării unei reţele. Dacă numărul de unităţi per procesor este mare, timpul de calcul este mai mare decât cel de comunicaţie, caz în care se poate considera că este vorba despre o bună paralelizare. Dacă numărul de unităţi per procesor este mic, calculele sunt încetinite de comunicaţie.

5.2.3 Pasul înapoi

Atenţia se va concentra asupra pasului patru, în care sunt combinate calcule şi transmisii. Seturile transmise sunt vectori de dimensiune PH. Calculul setului $a_{[p]}$ corespunde ecuaţiei 5.10. Timpul de calcul pentru un termen din $\overline{\delta}^O_{[p]} \star \overline{w}^O_{j \to [p]}$ depinde liniar de PO. Există câte un asemenea termen pentru fiecare unitate ascunsă simulată de un procesor. Ca urmare, timpul de calcul pentru întregul tablou depinde liniar de PO şi PH.

Ca şi mai sus, timpul de calcul depinde liniar de doi factori: numărul de unităţi ascunse simulate de fiecare procesor şi numărul de unităţi de ieşire simulate de fiecare procesor. Din nou, timpul de comunicaţie depinde liniar numai de unul dintre aceşti factori — de data aceasta numărul de unităţi de ieşire simulate de fiecare procesor. Similar, se poate anticipa o performanţă mai bună a algoritmului atunci când numărul de unităţi simulate de un procesor este mare.

5.3 Simulatorul NetParSim

Pe baza consideraţiilor teoretice anterioare, a fost realizat un simulator pentru antrenarea reţelelor neuronale cu metoda backpropagation — *NetParSim*.

Antrenarea reţelei neuronale se face distribuit, prin metoda partiţionării reţelei. Fiecărui procesor din reţeaua de calculatoare i se atribuie un număr de neuroni de pe fiecare strat al reţelei neuronale. Comunicaţia între procesoare se realizează conform topologiei în inel, astfel încât este simplu de modificat numărul de maşini care execută simulatorul.

Simulatorul implementează paralelizarea în şapte paşi propusă anterior, cu actualizarea ponderilor conexiunilor neuronale după fiecare tipar.

La pasul întâi se face prezentarea unui tipar de antrenare. Pentru a putea calcula activarea fiecărui neuron de pe stratul ascuns, fiecare procesor necesită întregul vector de intrare reprezentat de tiparul curent. Reamintim în acest sens faptul că un neuron de pe un strat are conexiuni de la toţi neuronii de pe stratul anterior.

În al doilea pas se calculează activarea unităţilor de ieşire, similar cu pasul întâi.

Pasul trei calculează valorile *delta* pentru unităţile de ieşire, prin compararea cu valorile de ieşire precizate de tiparul de antrenare.

Pasul al patrulea calculează valorile *delta* pentru unităţile ascunse, conform schemei din figura 5.4.

Paşii şase şi şapte calculează şi actualizează ponderile conexiunilor care intră în unităţile de ieşire, respectiv ascunse.

Pentru evaluarea performanţelor acestui simulator, prin comparaţie cu varianta secvenţială rulată pe acelaşi tip de maşină, se vor efectua o serie de experimente, care vor încerca să determine influenţa mai multor factori, în principal dimensiunea reţelei neuronale antrenate, respectiv dimensiunea reţelei de calculatoare pe care se rulează simulatorul.

Întrucât *calitatea* antrenării, respectiv numărul de iteraţii de efectuat necesar atingerii unui anumit grad de antrenare, depinde de parametri de antrenare — rata de învăţare, momentul — care oricum sunt aceiaşi şi în cazul variantei secvenţiale, vor fi utilizate reţele neuronale generice, cu dimensiunile precizate la fiecare tip de experiment, şi seturi de tipare generate aleator. Comparaţia între varianta secvenţială şi cea cu partiţionarea reţelei se va face prin măsurarea timpului necesar efectuării unui număr de iteraţii (10). Valorile prezentate pentru accelerare, respectiv eficienţa algoritmului, vor reprezenta media unui număr de 10 experimente/rulări din categoria respectivă.

Primul set de experimente vizează efectul modificării dimensiunii reţelei neuronale asupra eficienţei algoritmului. Pentru numărul maxim de staţii din reţea (16), se va aplica algoritmul pentru reţele neuronale astfel dimensionate încât, pentru fiecare procesor, numărul de neuroni de pe fiecare strat să varieze între 1 şi 20, adică reţele neuronale cu 16 neuroni pe stratul de intrare, cel intermediar şi cel de ieşire, 32 de neuroni şi aşa mai departe.

În al doilea set de experimente se va determina influenţa modificării numărului de procesoare pe care rulează simulatorul. Pentru o reţea neuronală de dimensiune fixă, se va modifica numărul de procesoare, între 2 şi 16. Experimentele vor fi efectuate pentru mai multe reţele neuronale, dimensionate astfel încât numărul de neuroni de pe fiecare strat să fie divizibil cu numărul maxim de procesoare utilizate.

În al treilea set de experimente se va determina în ce măsură este posibil să se utilizeze mai multe procesoare pe măsură ce se măreşte dimensiunea reţelei neuronale. Utilizând între 2 şi 16 procesoare, se vor efectua măsurători pentru reţele neuronale de dimensiuni alese astfel încât pentru P procesoare să avem pe fiecare strat un număr de neuroni proporţional cu P. În acest set de experimente, la adăugarea unui procesor în reţeaua de calculatoare, se va adăuga şi un număr de neuroni pe fiecare strat al reţelei neuronale.

Al patrulea set de experimente va determina în ce măsură este afectată eficienţa algoritmului de modificarea dimensiunii setului de antrenare. Deşi algoritmul este asfel proiectat încât actualizarea ponderilor se face după prezentarea *fiecărui* tipar, acest algoritm poate fi uşor modificat pentru însumarea unui număr de gradienţi înainte de actualizarea ponderilor.

Simulatorul NetParSim conţine un singur program. Prin urmare, procesele rulate pe fiecare staţie din reţeaua de calculatoare sunt identice. Diferenţierea rolurilor, în *administrator* şi *slave* se realizează prin intermediul argumentului transmis în linia de comandă – `processor_number`. În cazul în care acesta lipseşte sau este 0, procesul este *administrator* şi se va ocupa cu pornirea proceselor *slave*, cu numărul procesului dat de poziţia în configuraţia de inel.

Algoritmul utilizat este cel descris în acest capitol, cu comasarea paşilor al patrulea, al cincilea şi, parţial, al şaptelea, după cum s-a sugerat în secţiunea 5.1.6.

5.3.1 Fişierul de configurare

Programul primeşte informaţiile necesare prin intermediul unui fişier de configurare, identificat de variabila de mediu `NETPARSIM_CFG`, similar cu cel descris în secţiunea 4.3.1. El conţine informaţii despre reţeaua neuronală de simulat şi despre staţiile care alcătuiesc reţeaua de calculatoare. După cum s-a precizat, o informaţie suplimentară se transmite în linia de comandă – numărul procesorului în configuraţia de inel.

Se dă mai jos un exemplu de fişier de configuraţie. Valorile sunt precizate câte una pe linie, sub forma

```
<variabilă>=<valoare>
```

Liniile goale se ignoră, iar textul de la semnul '#' până la sfârştul liniei se ignoră (comentariu).

```
# Reteaua neuronala:
eta = 0.2                    # rata de invatare
alpha = 0.9                  # momentul

NrInput = 60                 # neuroni pe stratul de intrare
NrHidden = 60                # neuroni pe stratul intermediar
NrOutput = 60                # neuroni pe stratul de iesire

NrPatt = 10                  # numar total de tipare
PattFile = /work/tipare.dat  # fisier tipare de invatare

NrIter = 100                 # numar de iteratii
Err = 0.01                   # eroare acceptata

# Reteaua de calculatoare:
NrProcs = 12                 # numar procesoare in retea
HostsFile = /work/hosts.dat  # lista numelor statiilor
```

Parametrul `NrIter` precizează numărul total de iteraţii care sunt efectuate. Dacă el este 0, atunci se ia ca şi criteriu de oprire atingerea unei erori totale mai mică decât `Err`.

5.3.2 Codul aplicaţiei

Pentru început, aplicaţia determină numărul de neuroni de pe fiecare strat pe care îi are de simulat fiecare proces:

```
/* Calcul valori per procesor */
ProcInput = (NrInput + (NrProcs - 1)) / NrProcs;
ProcHidden = (NrHidden +(NrProcs - 1)) / NrProcs;
ProcOutput = (NrOutput +(NrProcs - 1)) / NrProcs;
```

Apoi, se determină indicii corespunzători neuronilor simulaţi de fiecare proces:

```
/* calcul indici input si lungime date pentru toate procesoare */
/* Strat intrare: */
input_ndx[0] = 0;
input_cnt[0] = ProcInput;
for (i = 1; i < NrProcs; i++) {
  input_ndx[i] = input_ndx[i-1] + input_cnt[i-1];
  tval = NrInput % NrProcs;
  if (tval == 0) {
    input_cnt[i] = ProcInput;
  }
  else if (tval > i) {
    input_cnt[i] = ProcInput;
  }
  else {
    input_cnt[i] = ProcInput - 1;
  }
}
LocalInput = input_cnt[processor_number];
```

```
/* Strat ascuns: */
hidden_ndx[0] = 0;
hidden_cnt[0] = ProcHidden;
for (i = 1; i < NrProcs; i++) {
  hidden_ndx[i] = hidden_ndx[i-1] + hidden_cnt[i-1];
  tval = NrHidden % NrProcs;
  if (tval == 0) {
    hidden_cnt[i] = ProcHidden;
  }
  else if (tval > i) {
    hidden_cnt[i] = ProcHidden;
  }
  else {
    hidden_cnt[i] = ProcHidden - 1;
  }
}
LocalHidden = hidden_cnt[processor_number];

/* Strat ieşire: */
output_ndx[0] = 0;
output_cnt[0] = ProcOutput;
for (i = 1; i < NrProcs; i++) {
  output_ndx[i] = output_ndx[i-1] + output_cnt[i-1];
  tval = NrOutput % NrProcs;
  if (tval == 0) {
    output_cnt[i] = ProcOutput;
  }
  else if (tval > i) {
    output_cnt[i] = ProcOutput;
  }
  else {
    output_cnt[i] = ProcOutput - 1;
  }
}
LocalOutput = output_cnt[processor_number];
```

După cum s-a precizat, singura diferenţă dintre *administrator* şi *slave* este dată de numărul procesului în inel, cel cu numărul 0 fiind *administrator*. Acesta trebuie să pornească celelalte procese *slave* şi să deschidă canalele de comunicaţie dintre ele, pentru configuraţia de inel. Fiecare proces va recepţiona date de la procesul precedent în reţea şi va transmite la cel următor.

După pornirea tuturor proceselor, acestea trebuie să aloce spaţiu pentru structurile de date utilizate, să iniţializeze aceste date şi să încarce tiparele de antrenare. Întrucât la iniţializarea ponderilor se utilizează valori mici aleatoare, trebuie asigurat ca acestea să fie identice pentru toate procesele – se utilizează aceeaşi rutină de generare de numere aleatoare, cu *aceeaşi rădăcină* (`seed`).

Procesul *administrator* porneşte măsurarea timpului, după care toate procesele execută un ciclu relativ simplu, de `NrIter` ori, în care se prezintă reţelei neuronale un tipar – selectat aleator – şi se actualizează ponderile conexiunilor:

```
/* PatternUpdate () */
for (iteration = 0; iteration < NrIter; iteration++) {
  pattern = (int)random() % NrPatt;
  PropagateActivity (pattern);
  CalculateWeightChanges(pattern);
}
```

Rutina `PropagateActivity` prezintă reţelei neuronale câte un tipar de antrenare. Pentru a putea propaga activarea prin reţeaua neuronală, trebuie să recepţioneze

activările neuronilor din stratul de intrare, respectiv cel ascuns, care se află în alte procesoare. Mai întâi se calculează activarea unităţilor ascunse din procesorul curent:

```
for (j = 0; j < LocalHidden; j++) {
  hidden_net[j] = hidden_weight[TotalInput][j]; /* bias */
}

p = processor_number;
for (k = 0; k < (NrProcs - 1); k++, p++) {
  processor = p % NrProcs;
  Send (input_activity[processor]);
  Receive (input_activity[(processor+1)%NrProcs]);
  StartNdx = input_ndx[processor];
  for (j = 0; j < LocalHidden; j++) {
    for (i = 0; i < input_cnt[processor]; i++) {
      hidden_net[j] += input_activity[processor][i] *
                       hidden_weight[StartNdx+i][j];
    }
  }
}

processor = (processor_number + (NrProcs - 1)) % NrProcs;
StartNdx = input_ndx[processor];
for (j = 0; j < LocalHidden; j++) {
  for (i = 0; i < input_cnt[processor]; i++) {
    hidden_net[j] += input_activity[processor][i] *
                     hidden_weight[StartNdx+i][j];
  }
  hidden_activity[processor_number][j] =
                     CalculateActivity (hidden_net[j]);
}
```

Se transmit activările neuronilor de intrare proprii şi cele recepţionate de la celelalte procesoare (pentru claritate, în apelurile rutinelor `Send` şi `Receive` nu s-a precizat şi lungimea tablourilor de valori implicate). Toate aceste valori sunt utilizate în calculul activării unităţilor ascunse simulate. Apoi se efectuează aceleaşi operaţii pentru stratul de ieşire:

```
for (j = 0; j < LocalOutput; j++) {
  output_net[j] = output_weight[TotalHidden][j]; /* bias */
}

p = processor_number;
for (k = 0; k < (NrProcs - 1); k++, p++) {
  processor = p % NrProcs;
  Send (hidden_activity[processor]);
  Receive (hidden_activity[(processor+1)%NrProcs]);
  StartNdx = hidden_ndx[processor];
  for (j = 0; j < LocalOutput; j++) {
    for (i = 0; i < hidden_cnt[processor]; i++) {
      output_net[j] += hidden_activity[processor][i] *
                       output_weight[StartNdx+i][j];
    }
  }
}

processor = (processor_number + (NrProcs - 1)) % NrProcs;
StartNdx = hidden_ndx[processor];
for (j = 0; j < LocalOutput; j++) {
  for (i = 0; i < hidden_cnt[processor]; i++) {
```

```
      output_net[j] += hidden_activity[processor][i] *
                       output_weight[StartNdx+i][j];
    }
    output_activity[processor_number][j] =
                       CalculateActivity (output_net[j]);
  }
```

De data aceasta, se transmit activările neuronilor de pe stratul ascuns simulaţi de procesul curent şi cele primite de la celelalte procesoare.

Pentru calculul actualizărilor ponderilor conexiunilor din reţeaua neuronală (rutina `CalculateWeightChanges`), mai întâi se calculează valorile *delta* în neuronii de ieşire, conform relaţiei 5.7:

```
for (i = 0; i < LocalOutput; i++) {
  tmp = output_activity[i];
  output_delta[i] = (target_pattern[pattern][i] - tmp) *
                    (tmp * (1.0 - tmp));
}
```

Apoi se calculează valorile delta pentru unităţile ascunse şi valorile cu care trebuie actualizate ponderile conexiunilor dintre stratul ascuns şi cel de ieşire, conform celor descrise în secţiunile 5.1.4.4 şi 5.1.4.5:

```
processor = (processor_number + 1) % NrProcs;
StartNdx = hidden_ndx[processor];
for (i = 0; i < hidden_cnt[processor]; i++) {
  out_data[i] = 0.0;
  for (j = 0; j < LocalOutput; j++) {
    out_data[i] += (output_delta[j] *
                    output_weight[StartNdx+i][j]);
    /* calcul actualizare ponderi hidden-output */
    output_change[StartNdx + i][j] =
           (eta * (output_delta[j] *
                   hidden_activity[processor][i])) +
           (alpha * output_change[StartNdx+i][j]);

    /* efectueaza actualizarea ponderi hidden-output */
    output_weight[StartNdx+i][j] += output_change[StartNdx+i][j];
  }
}

p = processor_number + 2;
for (k = 0; k < (NrProcs - 1); k++, p++) {
  Send (out_data);
  Receive (in_data);
  processor = p % NrProcs;
  StartNdx = hidden_uint_index[processor];
  for (i = 0; i < hidden_cnt[processor]; i++) {
    hidden_delta[i] = 0.0;
    for (j = 0; j < LocalOutput; j++) {
      hidden_delta[i] += (output_delta[j] *
                          output_weight[StartNdx+i][j]);
      /* calcul actualizare ponderi hidden-output */
      output_change[StartNdx+i][j] = (eta * (output_delta[j] *
                                  hidden_activity[processor][i])) +
                           (alpha * output_change[StartNdx+i][j]);
     /* efectueaza actualizare ponderi hidden-output */
     output_weight[StartNdx+i][j] +=
           output_change[StartNdx+i][j];
    }
  }
```

```
    for (i = 0; i < hidden_cnt[processor]; i++) {
      out_data[i] = in_data[i] + hidden_delta[i];
    }
  }

  for (i = 0; i < LocalHidden; i++) {
    hidden_delta[i] = out_data[i] *
            (hidden_activity[processor_number][i] *
            (1.0 - hidden_activity[processor_number][i]));
  }

  for (j = 0; j < LocalOutput; j++) {
    /* calcul actualizare ponderi hidden-output */
    output_change[TotalHidden][j] = (eta * output_delta[j]) +
            (alpha * output_change[TotalHidden][j]);
    /* efectueaza actualizarea ponderi hidden-output */
    output_weight[TotalHidden][j] +=
            output_change[TotalHidden][j];
  }
```

Similar se procedează pentru ponderile conexiunilor dintre stratul de intrare şi cel ascuns, conform secţiunii 5.1.4.6:

```
  /* calcul actualizare ponderi input-hidden */
  for (p = 0; p < NrProcs; p++) {
    StartNdx = input_ndx[p];
    for (j = 0; j < LocalHidden; j++) {
      for (i = 0; i < input_cnt[p]; i++) {
        hidden_change[StartNdx+i][j] =
            (eta * (hidden_delta[j] * input_activity[p][i])) +
            (alpha * hidden_change[StartNdx + i][j]);
      }
    }
  }
  for (j = 0; j < LocalHidden; j++) { /* pentru bias */
    hidden_change[TotalInput][j] = (eta * hidden_delta[j]) +
            (alpha * hidden_change[TotalInput][j]);
  }

  /* actualizeaza ponderi input-hidden */
  for (j = 0; j < LocalHidden; j++) {
    for (i = 0; i < (TotalInput + 1); i++) {
      hidden_weight[i][j] += hidden_change[i][j];
    }
  }
```

După cum se poate observa, pasul al şaptelea – actualizarea ponderilor – se efectuează chiar în momentul în care s-a terminat calculul actualizărilor, mai întâi pentru conexiunile dintre stratul ascuns şi cel de ieşire, apoi pentru cele dintre stratul de intrare şi stratul ascuns.

5.4 Rezultate experimentale

Simulatorul NetParSim a fost testat pe aceleaşi platforme ca şi simulatorul DataParSim (vezi secţiunea 4.4.1). Rezultatele prezentate sunt mediile rezultatelor a câte 10 experimente identice din categoria respectivă. Un experiment a constat din câte 100 de cicluri de prezentare, tiparele utilizate fiind generate aleator.

5.4.1 Influenţa dimensiunii variabile a reţelei neuronale

Tabelul 5.1 şi figura 5.5 (SunBlade) şi respectiv tabelul 5.2 şi figura 5.6 (Linux) prezintă eficienţa utilizând numărul maxim de staţii de lucru în fiecare dintre cele două configuraţii (16 staţii). Atât varianta secvenţială cât şi cea paralelă au fost executate pe reţele de exact aceleaşi dimensiuni. În figurile 5.5a) şi 5.6a) reţelele sunt astfel dimensionate încât numărul de unităţi simulate de către fiecare procesor să fie de la 1 la 20, ceea ce înseamnă 16 neuroni de intrare, pe stratul ascuns şi la ieşire, 32 de neuroni şi aşa mai departe.

Unităţi per procesor (U)	Dimensiune reţea				
	16U 16U 16U n	16U 16U 16U fc	16U 160 160	160 16U 160	160 160 16U
1	32.80	45.69	79.29	72.09	80.71
2	57.71	75.71	80.02	76.02	80.84
3	70.73	82.22	80.50	78.61	81.01
4	75.71	84.09	81.03	79.13	81.16
5	78.69	84.91	81.52	79.59	81.28
6	80.22	85.49	82.03	80.11	81.71
7	81.72	85.74	82.49	80.82	81.83
8	82.39	85.71	83.08	81.29	82.02
9	82.52	85.79	82.51	81.82	82.15
10	82.61	85.91	83.09	82.31	82.29
11	82.73	86.11			
12	82.81	86.09			
13	82.89	86.10			
14	83.02	86.19			
15	83.50	86.20			
16	82.51	84.29			
17	82.49	84.31			
18	82.52	84.29			
19	82.14	84.22			
20	82.81	84.23			

Tabela 5.1: *Dimensiune variabilă a reţelei: SunBlade.*

Se observă că atunci când există doar o unitate per procesor în fiecare dintre cele trei straturi, simulatorul paralel nu prezintă o eficienţă satisfăcătoare – doar 35%. Atunci când sunt două unităţi per procesor, eficienţa creşte la 59%, iar la circa şapte unităţi per procesor eficienţa se stabilizează la aproximativ 85% – probabil eficienţa maximă care se poate obţine.

Una dintre cauzele pentru care nu se poate obţine o eficienţă mai bună s-ar putea datora modului în care algoritmul accesează elementele vectorului $\overline{\alpha}^{l}_{[q]}$, pe parcursul transmisiei.

O altă cauză este ilustrată în cele două figuri 5.5a şi 5.6a. Cerculeţele pline reprezintă execuţia în absenţa comunicaţiei (înlocuită cu instrucţiuni NOP). Spaţiul dintre cele două curbe reprezintă influenţa comunicaţiei. Există un spaţiu între curba cu cerculeţe pline şi linia de eficienţă 100%, care trebuie să corespundă altor cauze software, întrucât nu există comunicaţie.

În celelalte trei grafice din figurile 5.5 şi 5.6 a fost fixat numărul de neuroni pe două dintre cele trei straturi, şi s-a modificat numărul de unităţi pe al treilea strat. În figurile b) a fost fixat numărul de unităţi ascunse şi de ieşire la 160, şi a fost ajustat numărul de unităţi de intrare între 16 şi 160, astfel încât fiecare procesor să simuleze

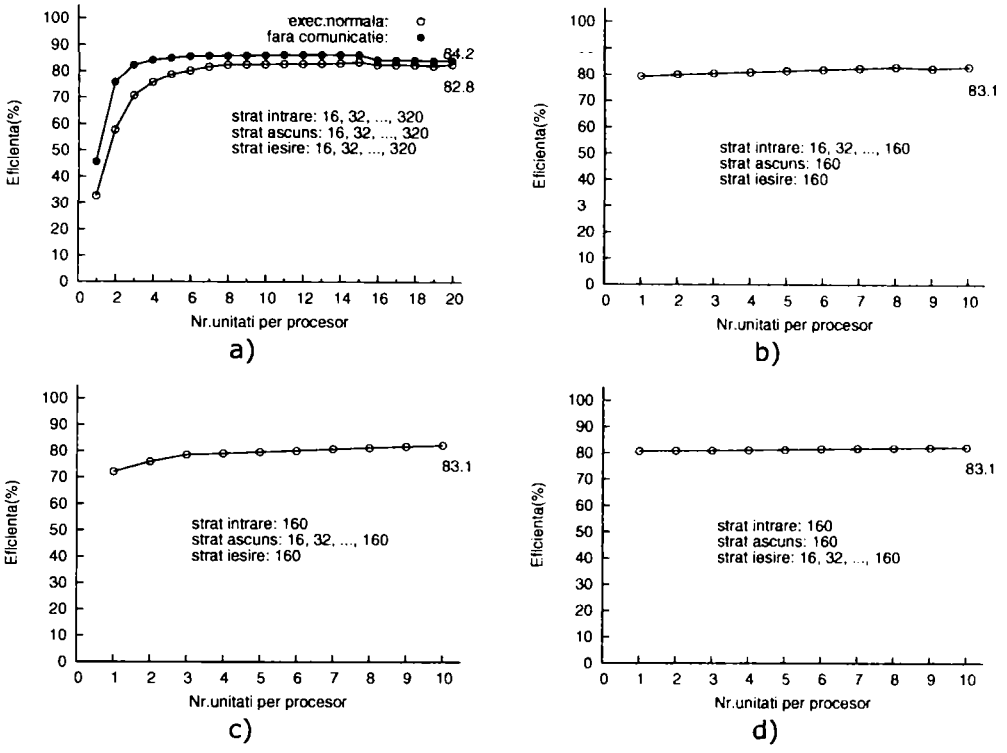


Figura 5.5: *Dimensiune variabilă a reţelei – SunBlade.*

câte 10 unităţi de pe straturile ascuns şi de ieşire şi între 1 şi 10 neuroni de pe stratul de intrare. S-a putut astfel evalua efectul unui număr foarte mic de unităţi de intrare per procesor. Oricum, fără vreun efect notabil. Eficienţa este cu foarte puţin mai mică atunci când există doar un neuron de intrare pentru fiecare procesor.

În figurile c) şi d) s-a modificat numărul de unităţi de pe stratul ascuns, respectiv de ieşire. Se observă că atunci când numărul de unităţi ascunse per procesor este prea mic, eficienţa scade puţin, însă nu foarte mult. Numărul de unităţi de ieşire per procesor nu are efect asupra eficienţei câtă vreme sunt simulate suficient de multe unităţi din celelalte straturi.

Concluzia care se poate deduce este că atunci când numărul de unităţi per procesor este mic pentru toate straturile eficienţa este scăzută, însă faptul că procesoarele deţin puţine unităţi doar într-unul dintre straturi are un efect neglijabil asupra eficienţei.

5.4.2 Efectul numărului variabil de procesoare

Tabelul 5.3 şi figura 5.7 şi respectiv tabelul 5.4 şi figura 5.8 reprezintă accelerarea pentru diferite dimensiuni ale reţelei utilizând până la numărul maxim de procesoare (16). În fiecare grafic individual dimensiunea reţelei rămâne fixă şi se modifică numărul de procesoare. Dimensiunile reţelei au fost alese astfel încât numărul maxim de procesoare (16) să dividă numărul de unităţi din fiecare strat. Experimentele au fost realizate pentru reţele: 16-16-16, 32-32-32, 48-48-48, 80-80-80, 160-160-160.

Graficele sugerează, după cum era de aşteptat, faptul că pentru un număr mic de neuroni pentru fiecare procesor accelerarea şi eficienţa sunt departe de valorile

Unităţi per procesor (U)	Dimensiune reţea				
	16U 16U 16U n	16U 16U 16U fc	16U 160 160	160 16U 160	160 160 16U
1	30.6	43.9	76.9	69.8	77.9
2	55.2	73.1	78.3	73.2	78.1
3	69.1	80.4	78.7	75.1	79.33
4	71.7	82.5	79.8	76.5	79.9
5	76.2	83.1	79.85	76.75	80.1
6	78.9	83.3	80.3	78.1	79.7
7	79.8	83.65	80.7	78.25	79.9
8	80.1	83.5	81.2	79.3	80.1
9	80.4	83.6	80.5	79.8	80.3
10	80.5	83.7	81.7	80.25	80.4
11	80.6	84.0			
12	80.7	84.0			
13	80.8	84.0			
14	81.1	84.1			
15	81.3	84.1			
16	80.9	82.2			
17	80.85	82.2			
18	80.44	82.2			
19	80.1	82.0			
20	80.6	82.1			

Tabela 5.2: *Dimensiune variabilă a reţelei: Linux.*

optime.

Pentru reţele neuronale de aceleaşi dimensiuni, graficele nu diferă semnificativ pentru cele două platforme hardware, SunBlade, respectiv Linux. În privinţa valorilor obţinute pentru accelerare, se observă că în cazul reţelelor de dimensiuni reduse, 16-16-16, respectiv 32-32-32, accelerarea maximă obţinută este foarte departe de valoarea ideală, atunci când simulatorul utilizează maximul de procesoare (16), valoarea cea mai bună obţinându-se pentru reţeaua neuronală cu dimensiunea cea mai mare (160-160-160). Alura graficelor sugerează faptul că pentru reţelele neuronale cu dimensiunile precizate, performanţele simulatorului (accelerarea, respectiv eficienţa) este posibil să se îndepărteze şi mai mult de valorile ideale în cazul în care am adăuga mai multe procesoare.

Concluzia care rezultă din acest experiment este că pentru a ne situa cât mai aproape de valorile optime pentru accelerare, respectiv eficienţă, este necesar să asigurăm un număr cât mai mare de neuroni pentru fiecare procesor. Pentru cazul experimentelor efectuate, se pare că performanţe acceptabile se pot obţine dacă avem cel puţin 10 neuroni de pe fiecare strat asignaţi fiecărei maşini din reţeaua de calculatoare.

5.4.3 Influenţa scalării reţelei cu numărul de procesoare

În această secţiune s-a încercat să se determine în ce măsură este posibil să se utilizeze mai multe procesoare pe măsură ce se măreşte dimensiunea reţelei. În tabelele 5.5, 5.6 şi respectiv în figurile 5.9, 5.10 este reprezentată eficienţa când sunt utilizate între 2 şi 16 procesoare. Graficul pentru cea mai scăzută eficienţă a fost obţinut fixând dimensiunea reţelei astfel încât fiecare procesor să simuleze câte o unitate din fiecare strat, ceea ce înseamnă că pentru P procesoare este simulată o reţea $P-P-P$.

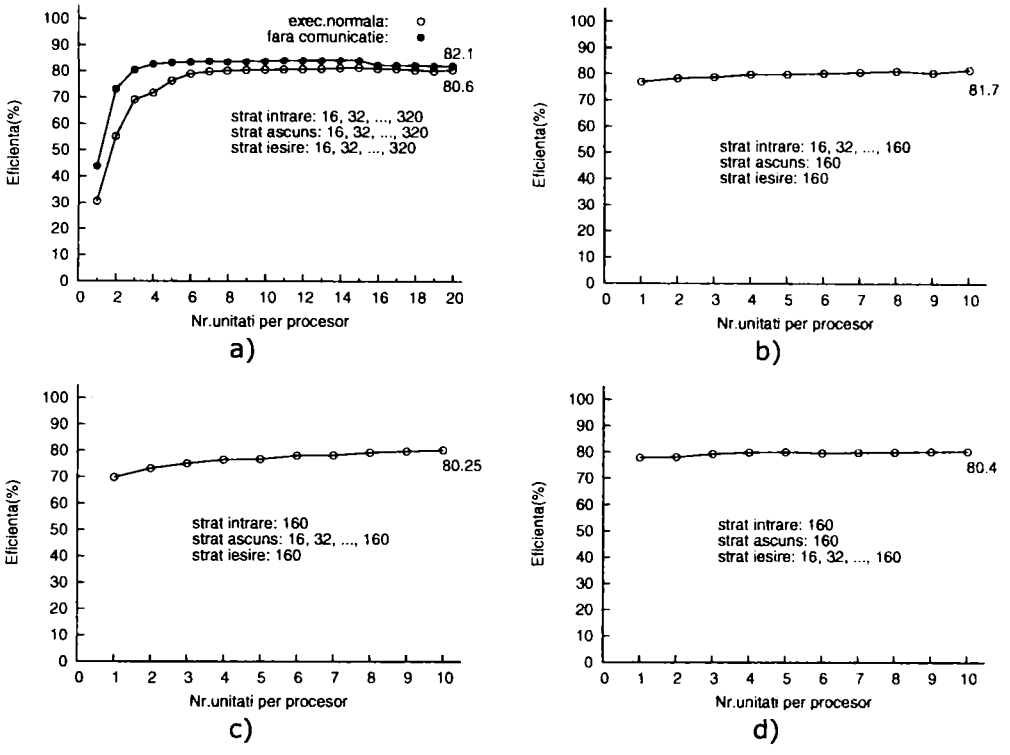


Figura 5.6: *Dimensiune variabilă a reţelei – Linux.*

Număr procesoare	Dimensiune reţea				
	16 16 16	32 32 32	48 48 48	80 80 80	160 160 160
2	1.19	1.20	1.32	1.38	1.60
3	2.01	2.21	2.22	2.29	2.36
4	2.92	3.09	3.09	3.22	3.20
5	2.91	3.41	3.81	4.13	4.14
6	3.93	4.89	4.63	5.11	4.90
7	3.91	5.71	5.11	5.72	5.96
8	4.62	6.31	6.01	6.79	6.89
9	4.61	6.30	7.52	7.03	7.49
10	4.60	6.93	7.51	8.10	8.42
11	4.59	6.92	7.53	8.51	9.47
12	4.57	6.91	8.52	8.50	9.90
13	4.56	6.95	8.51	9.31	11.31
14	4.55	6.94	8.50	9.30	11.62
15	4.54	7.01	8.49	9.29	13.04
16	5.40	7.92	9.51	10.02	13.60

Tabela 5.3: *Accelerarea – număr variabil de procesoare: SunBlade.*

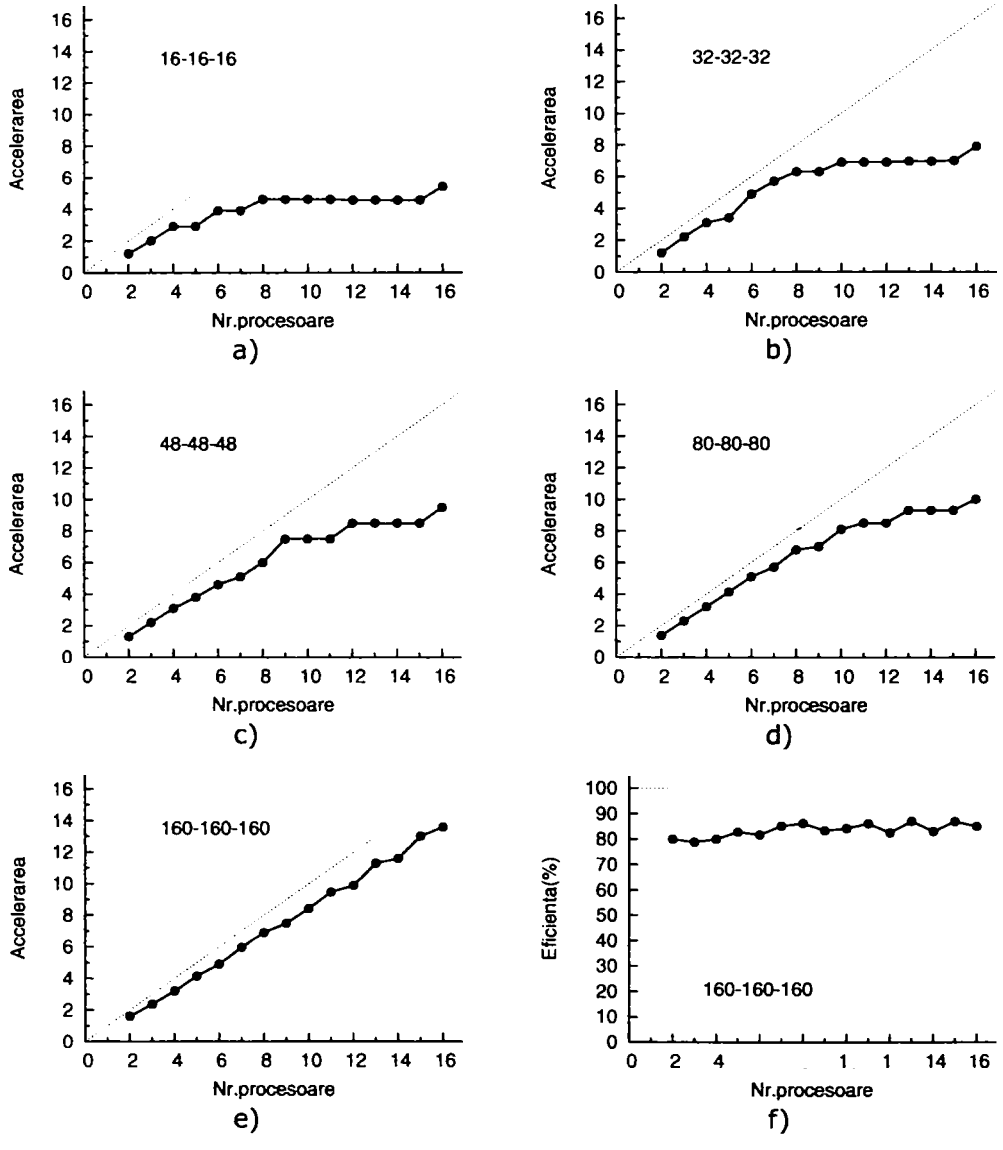


Figura 5.7: *Accelerarea – număr variabil de procesoare: SunBlade.*

Număr procesoare	Dimensiune rețea				
	16	32	48	80	160
	16	32	48	80	160
	16	32	48	80	160
2	1.15	1.1	1.15	1.2	1.25
3	1.88	1.9	2.0	2.1	2.1
4	2.7	2.85	2.85	2.95	3.0
5	2.75	3.15	3.5	3.8	3.9
6	3.3	4.4	4.22	4.88	4.6
7	3.5	5.3	4.9	5.3	5.7
8	4.1	5.9	5.8	6.2	6.5
9	4.15	6.0	7.1	6.9	7.1
10	4.2	6.2	7.15	7.8	8.2
11	4.25	6.3	7.2	8.1	9.15
12	4.15	6.35	8.0	8.2	9.6
13	4.2	6.4	8.1	8.9	10.95
14	4.25	6.4	8.15	9.0	11.1
15	4.2	6.85	8.2	9.1	12.0
16	4.9	7.2	9.3	9.8	12.7

Tabela 5.4: *Accelerarea - număr variabil de procesoare: Linux.*

Următoarele grafice sunt pentru câte o rețea $2P - 2P - 2P$, $3P - 3P - 3P$, $5P - 5P - 5P$ și respectiv $10P - 10P - 10P$.

Graficul cu cea mai scăzută eficiență descrește considerabil pe prima porțiune. Aceasta se datorește faptului că pentru un număr mic de procesoare P care simulează o rețea $P - P - P$, evident, rețeaua nu este prea mare. S-a văzut mai devreme că pentru rețele mici, timpul de calcul al funcțiilor de activare reprezintă o porțiune considerabilă a timpului total de execuție. Această parte a algoritmului este desigur bine paralelizată și deci de aici eficiența mai mare la începutul graficelor.

După cum s-a arătat în secțiunea 5.4.1, simularea rețelelor cu număr mic de unități per procesor prezintă o eficiență scăzută. Utilizând cel puțin 5 unități per procesor eficiența este aproape constantă. Ca urmare, este posibil să se mărească numărul de procesoare fără a pierde din eficiență dacă numărul de unități per procesor rămâne constant (deci mărind numărul total de unități proporțional cu numărul de procesoare). De remarcat faptul că numărul de ponderi crește aproape cuadratic atunci când numărul de unități de pe straturi crește liniar și deci și volumul de calcul crește aproape cuadratic.

Curbele din figurile 5.9, 5.10 sunt linii aproape orizontale. Dacă se presupune că graficele continuă în același mod și pentru un număr mai mare de procesoare, se poate decide dacă este posibil să se utilizeze mai multe procesoare pentru o problemă dată. Dacă într-o simulare procesoarele simulează câte 10 unități din fiecare strat cu o eficiență de 85.6%, este posibil să se utilizeze de două ori mai multe procesoare (5 unități per procesor) cu o eficiență de 80.2%. Este chiar mai preferabil să fie o unitate per procesor în loc de două, chiar dacă procesoarele suplimentare nu sunt utilizate prea bine: având de două ori mai multe procesoare se va obține o eficiență de 31.5% în loc de 58.2%, dar se va constata și o creștere a vitezei de procesare.

5.4.4 Modificarea dimensiunii setului de antrenare

Contrar paralelizării cu partiționarea datelor din capitolele anterioare, schema cu partiționarea rețelei actualizează ponderile *după fiecare tipar*. Desigur, este posibil să se actualizeze ponderile mai puțin frecvent. Modificând foarte puțin programul, a fost posibil să se utilizeze un set de antrenare de dimensiune oarecare. Modificarea

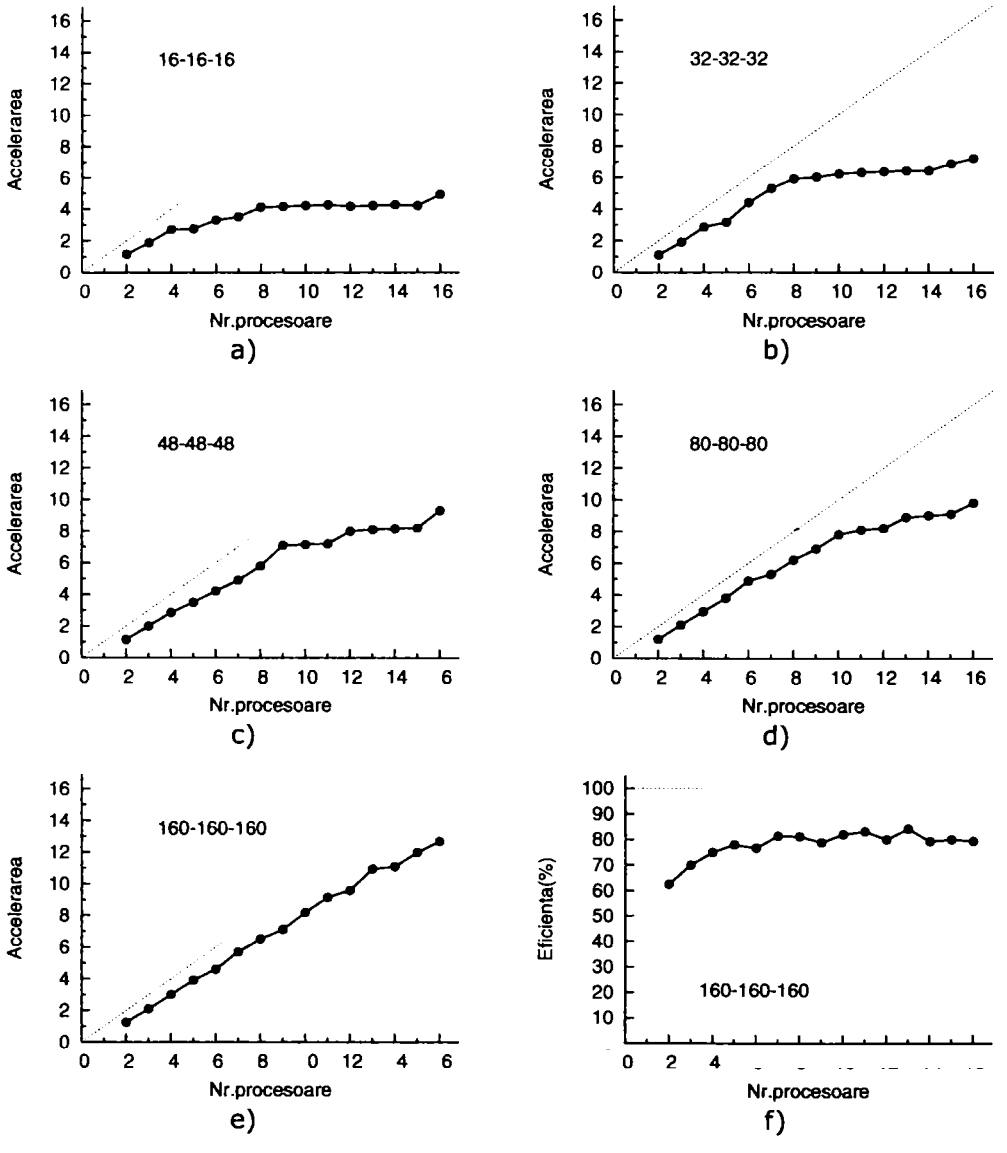


Figura 5.8: *Accelerarea – număr variabil de procesoare: Linux.*

Nr. Procesoare	Dimensiune reţea				
	1P	2P	3P	5P	10P
2	61.01	71.19	79.31	83.21	88.29
3	46.99	64.99	74.98	81.04	87.03
4	44.02	64.03	74.52	81.19	86.87
5	39.98	62.98	74.01	80.79	86.76
6	39.01	61.01	73.27	80.63	86.52
7	38.52	59.19	73.02	80.56	86.24
8	36.97	58.51	72.48	80.24	86.01
9	35.31	58.09	71.99	80.31	85.79
10	34.02	58.21	71.01	80.13	85.88
11	32.98	58.02	70.98	80.52	85.94
12	32.26	58.49	71.31	80.46	85.93
13	32.19	58.46	71.24	80.39	85.91
14	32.13	58.39	71.18	80.34	85.86
15	32.11	58.33	71.16	80.31	85.81
16	32.04	58.31	71.09	80.23	85.72

Tabela 5.5: *Scalarea reţelei cu numărul de procesoare: SunBlade.*

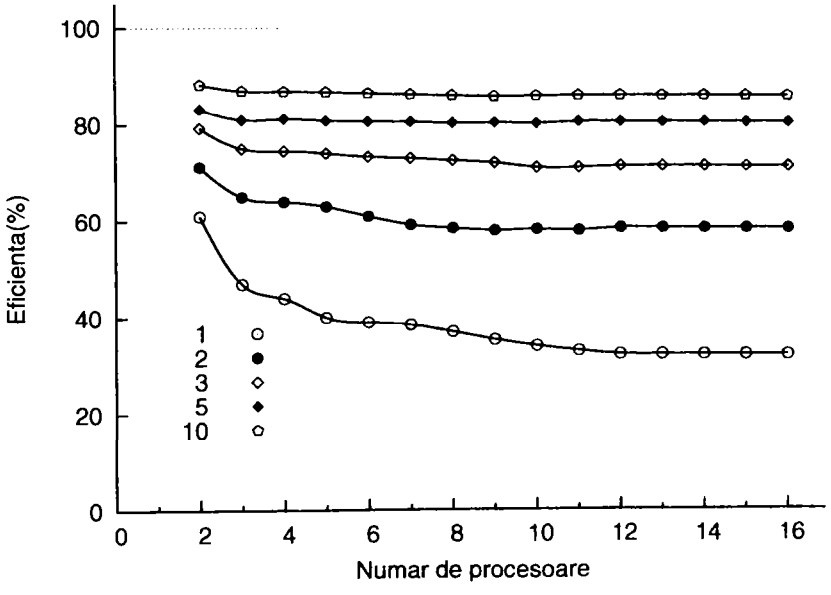


Figura 5.9: *Eficienţa în cazul scalării reţelei cu numărul de procesoare – SunBlade.*

Nr. Procesoare	Dimensiune reţea				
	1P	2P	3P	5P	10P
2	58.9	69.1	77.3	81.3	85.8
3	45.8	63.2	73.1	79.2	85.1
4	42.1	61.95	72.7	79.2	84.7
5	38.95	61.15	72	78.7	84.35
6	37.1	59.3	71.15	78.95	84.2
7	36.7	57.3	71.15	79.2	84.15
8	35.2	56.4	70.7	78.1	84.2
9	33.1	56.2	70.1	78.3	83.7
10	32.1	56.3	69.6	78.0	83.8
11	30.9	56.1	69.1	78.3	83.75
12	30.4	56.4	69.33	78.25	83.7
13	30.0	56.35	69.25	78.2	83.8
14	30.2	56.4	69.15	78.15	83.65
15	30.0	56.3	69.0	78.2	83.5
16	30.05	56.25	69.15	78.25	83.5

Tabela 5.6: *Scalarea reţelei cu numărul de procesoare: Linux.*

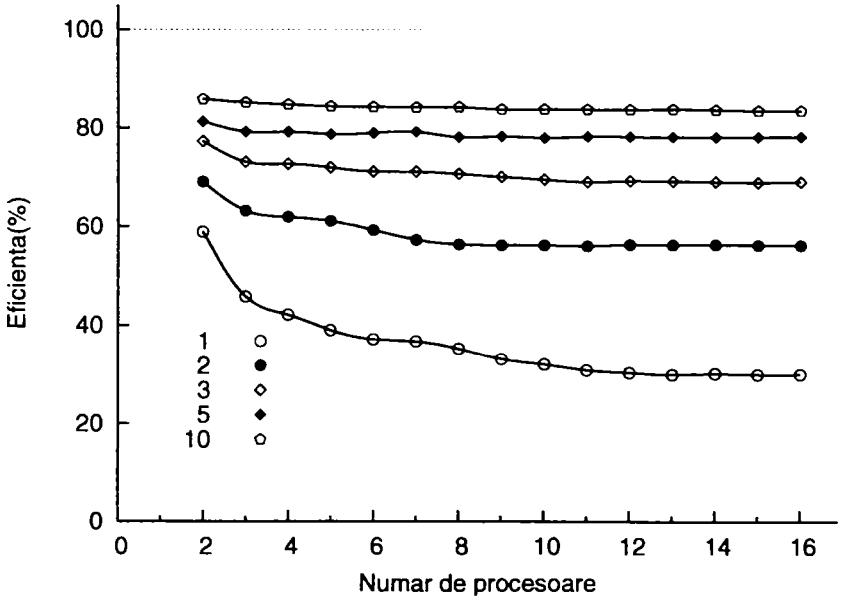


Figura 5.10: *Eficienţa în cazul scalării reţelei cu numărul de procesoare – Linux.*

constă în însumarea simplă a gradienţilor înainte de actualizarea ponderilor. Această modificare are însă un dezavantaj — nu mai este posibilă combinarea paşilor patru, cinci şi şapte din algoritm.

Dimensiune batch	Eficienţa
1	85.22
2	92.00
3	91.31
4	90.59
5	90.33
6	90.01
7	89.52
8	88.99
9	88.61
10	88.03
11	87.62
12	87.21
13	86.84
14	86.41
15	86.02
16	85.64
17	85.33
18	85.00
19	84.51
20	84.09

a) SunBlade: reţea 160-160-160

Dimensiune batch	Eficienţa
1	83.0
2	90.1
3	89.1
4	88.7
5	88.45
6	88.15
7	87.6
8	87.1
9	86.3
10	86.2
11	85.9
12	85
13	84.7
14	84.45
15	84.15
16	83.7
17	83.25
18	83.1
19	82.8
20	82.0

b) Linux: reţea 160-160-160

Tabela 5.7: *Eficienţa pentru actualizarea la epocă*

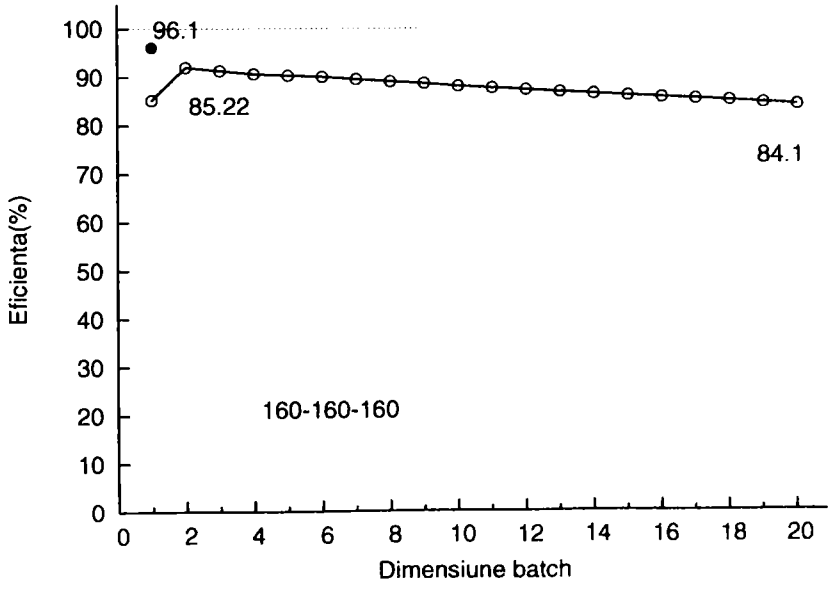


Figura 5.11: *Eficienţa pentru actualizarea la epocă: SunBlade*

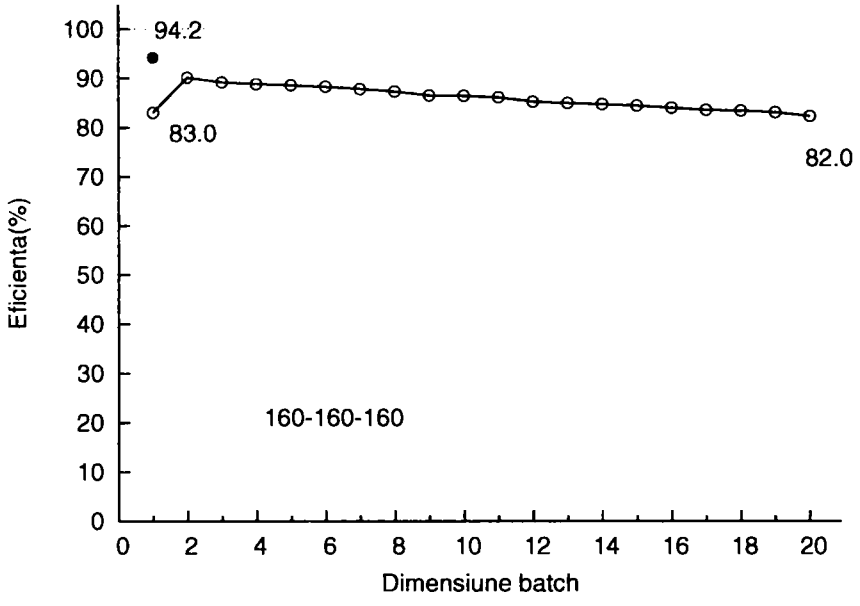


Figura 5.12: *Eficienţa pentru actualizarea la epocă: Linux*

În tabelul 5.7 şi figurile 5.11, 5.12 este prezentată eficienţa execuţiilor pentru numărul maxim de procesoare (16) cu modificarea dimensiunii setului de antrenare. De remarcat că pentru dimensiunea 1 s-a reprezentat atât varianta modificată (cerculeţul negru) cât şi implementarea originală (cerculeţul alb). Pentru toate celelalte dimensiuni ale setului de antrenare a fost utilizată, desigur, varianta modificată. Implementarea modificată pare să fie mai rapidă decât varianta originală. Totuşi, nu este chiar aşa, întrucât cele două implementări paralele ar trebui comparate de fapt cu două implementări secvenţiale diferite, iar implementarea secvenţială a actualizării după fiecare tipar este mai rapidă decât versiunea corespunzătoare pentru dimensiunea 1 a setului de antrenare. În tabelele 5.8, 5.9 sunt prezentaţi timpii de execuţie pentru 10 cicluri de antrenare pentru reţele $160 - 160 - 160$.

Versiunea	16 procesoare paralel	Secvenţial
Actualizare la tipar	0.52 sec	4.5 sec
Actualizare batch	0.64 sec	6.2 sec

Tabela 5.8: *Timpii de execuţie - SunBlade.*

Versiunea	16 procesoare paralel	Secvenţial
Actualizare la tipar	0.54 sec	4.65 sec
Actualizare batch	0.68 sec	6.34 sec

Tabela 5.9: *Timpii de execuţie - Linux.*

În figurile 5.11, 5.12 se observă că eficienţa implementării cu actualizarea în mod batch scade puţin atunci când sunt folosite seturi de antrenare de dimensiune

mare. Aceasta se datorează faptului că operaţiile de calcul al actualizărilor ponderilor sunt efectuate mai puţin frecvent.

5.5 Concluzii

În această secţiune a fost implementat şi testat un simulator paralel de backpropagation cu actualizarea la tipar – NetParSim. Algoritmul utilizează o partiţionare a reţelei. Mai mult, comunicaţia a fost redusă la un nivel la care are un efect redus asupra algoritmului.

Algoritmul utilizează schema de actualizare la tipar, ceea ce înseamnă că actualizări foarte frecvente pot fi utilizate fără a influenţa algoritmul. Totuşi, se poate utiliza eficient un număr mare de procesoare doar dacă se simulează reţele neuronale de dimensiune mare.

Datorită acestor caracteristici, această strategie de paralelizare, cu partiţionarea reţelei este foarte diferită de varianta cu partiţionarea datelor discutată în capitolul anterior. Eficienţa acelor algoritmi cu partiţionarea datelor era aproape independentă de dimensiunea reţelei simulate, iar dimensiunea setului de antrenare era un factor determinant pentru creşterea eficienţei algoritmului de partiţionare a datelor.

O observaţie importantă rezultă din experimentele care au fost prezentate în secţiunea 5.4.1: performanţele simulatorului sunt afectate în principal de numărul de neuroni de pe stratul de intrare şi cel intermediar, şi mult mai puţin de numărul de neuroni de pe stratul de ieşire.

Contribuţiile acestui capitol sunt:

1. Dezvoltarea şi implementarea unui algoritm de paralelizare cu partiţionarea reţelei. Algoritmul divide reţeaua neuronală, astfel încât fiecare maşină care participă la simulator va deţine un număr de neuroni de pe fiecare strat al reţelei neuronale.

2. Reformularea calculelor care se efectuează în algoritmul backpropagation, în vederea distribuirii lor pe o reţea de calculatoare, cu minimizarea timpului necesar transmisiei de date.

3. Evaluarea necesarului de memorie al algoritmului prezentat. Pe baza formulelor prezentate (formula 5.15) se poate face o evaluare a dimensiunii maxime a reţelei neuronale posibil de simulat de către NetParSim.

4. Studiul experimental al influenţei factorilor care pot afecta performanţele evoluţiei simulatorului într-o reţea de calculatoare. Concluzia care se poate deduce este că performanţele simulatorului vor fi cu atât mai bune cu cât numărul de neuroni de pe fiecare strat de procesat pe fiecare maşină este mai mare. Din experimentele efectuate, s-au observat valori satisfăcătoare pentru accelerare, respectiv eficienţă atunci când reţeaua neuronală a avut asemenea dimensiuni încât să fie alocaţi cel puţin zece neuroni de pe fiecare strat, pentru fiecare maşină din reţea. Investigaţii pe această temă au fost prezentate în [Ba07a] şi [BCP07].

6 Experimente cu aplicaţii uzuale

Capitolul de faţă prezintă rezultatele obţinute pe simulatoarele prezentate în capitolele anterioare pentru un set de aplicaţii uzuale, des utilizate pentru măsurarea performanţelor reţelelor neuronale.

Aplicaţiile utilizate în experimente sunt listate în tabelul 6.1.

Aplicaţia	Dimensiune reţea	Nr. tipare
NETtalk [81]	$203 \times 240 \times 26$	5438
Recunoaştere caractere (set de fonturi X11)	$40 \times 30 \times 26$	26
Compresie de imagine [13]	$64 \times 16 \times 64$	4096
Recunoaştere de imagini/OCR	$1024 \times 512 \times 64$	4096
Recunoaştere ţinte sonar [29], [28]	$60 \times 24 \times 2$	208

Tabela 6.1: *Aplicaţii de back-propagation utilizate pentru măsurarea performanţelor antrenării.*

6.1 NETtalk

În capitolele anterioare am analizat două implementări paralele ale unor simulatoare de reţele neuronale strict din punctul de vedere al eficienţei. În cazul utilizării unui set de antrenare de dimensiuni mari, varianta cu paralelizarea datelor poate părea superioară celei cu partiţionarea reţelei întrucât se pot obţine eficienţe ridicate. Totuşi, asemenea consideraţii nu sunt suficiente în contextul reţelelor neuronale. Modificarea dimensiunii setului de antrenare modifică de asemenea chiar procesul de învăţare. Se va evidenţia în continuare importanţa actualizării frecvente a ponderilor şi influenţa nefastă asupra antrenării în cazul NETtalk când se utilizează un set de antrenare de dimensiune mare. Ca urmare, avantajul utilizării unei scheme cu partiţionarea datelor poate să nu mai existe.

În continuare vor fi prezentate rezultatele rulării unei aplicaţii cunoscută sub numele de NETtalk. Au fost utilizate frecvenţe specifice de actualizare a ponderilor, întrucât ele au fost testate şi de către alţi cercetători pe diferite arhitecturi paralele (vezi [22], [7], [20], [100] şi [104]).

6.1.1 Setul de date NETtalk

În această secţiune va fi descris setul de date NETtalk, care a fost construit de Sejnowski şi Rosenberg [81] în 1986. Sejnowski şi Rosenberg au antrenat o reţea neuronală feed-forward pentru pronunţarea cuvintelor în limba engleză. Setul de date este larg utilizat ca termen de referinţă în domeniul reţelelor neuronale pentru a testa viteza de antrenare şi abilităţile de generalizare. A fost astfel posibilă compararea rezultatelor proprii cu cele obţinute de alţi cercetători.

Setul complet de date constă din 20008 cuvinte în limba engleză, împreună cu fonemele corespunzătoare şi informaţiile de accentuare. Modul normal de utilizare a datelor este însă să se facă învăţarea doar pe un subset, circa 1000 de cuvinte uzuale. Cuvintele rămase pot fi apoi utilizate pentru a testa abilităţile de generalizare ale reţelei.

Setul de date este astfel construit încât să existe o corespondenţă unu-la-unu între literele unui cuvânt şi foneme şi informaţia de accentuare. De exemplu cuvântul "pronounce" are ca şi corespondent un şir de foneme de aceeaşi lungime (`prxnW-ns-`) şi un şir de simboluri de accentuare (`>>0>1<<<<`). De fapt, simbolurile de accentuare constă atât din informaţie despre accente, cât şi despre silabe. În total sunt 52 de foneme diferite şi 5 simboluri de accentuare diferite. [81] conţine informaţii mai detaliate despre aceste simboluri.

Traducerea unei singure litere dintr-un cuvânt în fonemul corect depinde de literele înconjurătoare din cuvânt, de aceea se utilizează o "fereastră" de 7 caractere. Reţeaua neuronală trebuie să genereze fonemul corespunzător literei din centrul ferestrei. Cuvântul defilează prin fereastră astfel încât toate literele din cuvânt sunt plasate, pe rând, în centrul ferestrei de 7 caractere. Se prezintă reţelei neuronale toate literele din fereastră, după cum este ilustrat în figura 6.1. În figură, reţeaua trebuie să genereze fonemul "x", care corespunde literei "o" (din centrul ferestrei). Pentru aceasta, se prezintă reţelei toate caracterele din fereastră, adică "_pronou".

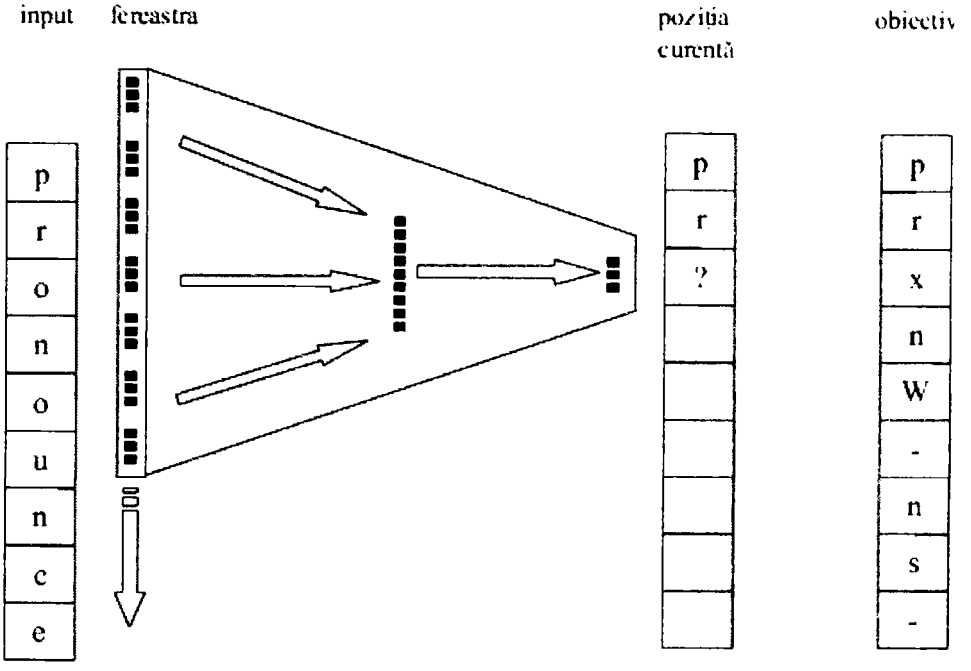


Figura 6.1: *Fereastra de prezentare utilizată de NETtalk.*

Utilizând o fereastră de 7 caractere, ca şi Sejnowski şi Rosenberg, este evident că reţeaua neuronală nu va putea pronunţa corect toate cuvintele. Un exemplu ar fi cazul cuvintelor *finite* şi *infinite*. Prezentând reţelei şirul "finite_" în fereastra de 7 caractere, pentru a determina fonemul pentru litera din centrul ferestrei, reţeaua nu are cum să ştie dacă întregul cuvânt este *finite* sau *infinite*. Pronunţarea literei 'i' (fin*i*te sau infin*i*te) diferă la cele două cuvinte. Au mai fost date ca exemple *though* şi *thought*, şi altele. Desigur, problema se rezolvă utilizând o fereastră mai mare. Totuşi, nu s-a urmărit construirea reţelei neuronale perfecte pentru NETtalk, ci doar să se poată compara rezultatele obţinute cu ale altor cercetători, iar majoritatea au utilizat o fereastră de 7 litere.

6.1.2 Implementare

A fost utilizată o reprezentare ortogonală a fiecărei litere pentru unităţile de intrare, cu un alfabet cu 26 de litere şi o dimensiune a ferestrei de citire de 7, rezultând un total de 182 de unităţi de intrare. Aceasta înseamnă că pentru fiecare literă din fereastră este activă numai una dintre cele 26 de unităţi de intrare corespunzătoare. Dacă una dintre litere este caracterul spaţiu – nu este activată nici una dintre cele 26 de unităţi de intrare.

Spre deosebire de unităţile de intrare, Sejnowski şi Rosenberg utilizează o reprezentare distribuită pentru unităţile de ieşire. În total sunt 52 de foneme, exprimate prin intermediul a 21 de atribute de articulare, cum ar fi punctul de articulare, înălţimea vocalei şi aşa mai departe. Cele cinci simboluri de accentuare sunt reprezentate ortogonal, având ca rezultat 26 de unităţi de ieşire. Pentru a putea compara rezultatele, în general se utilizează 60 de unităţi pe stratul ascuns. Desigur, ar fi posibil să se obţină performanţe mai bune în privinţa antrenării dacă s-ar utiliza mai multe unităţi ascunse, însă, după cum s-a mai precizat, nu s-a căutat să se îmbunătăţească performanţele reţelelor neuronale, ci doar să se poată compara rezultatele.

Setul de date NETtalk, compus din cuvinte, foneme şi simboluri de accentuare, este foarte mic în comparaţie cu alte aplicaţii de reţele neuronale. În total, el necesită circa 15 Kbytes pentru un set de date de 1000 de cuvinte.

Dispunând de un set de date atât de mic, a fost posibil să se plaseze câte o copie a întregului set de date în fiecare dintre procesoare, ceea ce prezintă un avantaj pentru paralelizarea cu partiţionarea reţelei. În acest fel, a fost nevoie numai de o distribuire iniţială a datelor, iar pe parcursul simulării nu a mai fost necesară altă comunicaţie. Toate procesoarele au acces direct la toate tiparele. Ca urmare, pasul unu din paralelizarea cu partiţionarea reţelei este simplu, aşa cum s-a menţionat în secţiunea 5.1.5.

Întrucât este utilizată o reprezentare binară (şi ortogonală) pentru unităţile de intrare, calculul activării unităţilor ascunse poate fi simplificat. Pentru o singură literă din fereastră, numai una dintre cele 26 de unităţi de intrare este activă (cu valoarea 1), celelalte 25 fiind inactive (valoarea 0). Ca urmare, în formula:

$$a_{pj}^{H} = f(net_{pj}^{H}) = f(\sum_{i=0}^{NI-1} a_{pi}^{I} w_{i \to j}^{H}) \tag{6.1}$$

nu este necesar să se efectueze toate înmulţirile. Este suficientă însumarea ponderilor de la unităţile de intrare cu activarea 1, lucru întotdeauna posibil atunci când se utilizează o reprezentare binară.

6.1.3 Simulări şi rezultate

În această secţiune va fi prezentat efectul alegerii dimensiunii setului de antrenare asupra capabilităţilor de antrenare. Au fost utilizate câteva dimensiuni specifice, întrucât ele au mai fost aplicate şi de alte implementări paralele pe diferite arhitecturi [68], [100].

Rezultatele obţinute corespund celor obţinute de Sejnowski şi Rosenberg. Pentru fiecare execuţie s-a precizat atât rata de învăţare cât şi valoarea momentului, în vederea reproducerii acestor rezultate.

Sejnowski şi Rosenberg au utilizat două măsuri ale performanţei. Ieşirea este considerată *potrivire perfectă*[1] dacă valoarea fiecărei unităţi de ieşire diferă cu cel mult 0.1 faţă de valoarea dorită.

Ieşirea este considerată *cea mai bună alegere*[2] dacă vectorul ţintă corect este acel vector dintre cei 52 de vectori ţintă posibili care formează cel mai mic unghi cu vectorul actual al activării unităţilor de ieşire. Aceasta înseamnă că toţi cei 52 de vectori ţintă posibile (fiecare compus din 26 de valori) se compară cu vectorul generat la ieşire şi se determină vectorul ţintă cel mai apropiat. Dacă acest vector ţintă este vectorul ţintă corect, ieşirea reprezintă cea mai bună alegere. Ca urmare, vectorul de ieşire poate să fie foarte diferit de vectorul ţintă corect atunci când se utilizează acest criteriu.

[1] "perfect match"
[2] "best guess"

Evident, criteriul celei mai bune alegeri va da cel mai mare procentaj de răspunsuri corecte (o potrivire perfectă este întotdeauna cea mai bună alegere, dar inversa nu este întotdeauna adevărată). Din păcate, criteriul celei mai bune alegeri necesită resurse computaţionale mult mai mari.

Pentru antrenare a fost utilizat un set de 1000 de cuvinte mai frecvente din limba engleză. După fiecare prezentare a celor 1000 de cuvinte, reţeaua a fost testată pe aceleaşi 1000 de cuvinte, pentru a-i măsura performanţele. Graficele din figura 6.2 prezintă performanţele reţelei la antrenarea cu diferite frecvenţe de actualizare a ponderilor. În toate sesiunile de antrenare, fiecare cuvânt a fost prezentat de câte 100 de ori, în ordine aleatoare. Valoarea termenului moment a fost cea uzuală de 0.9.

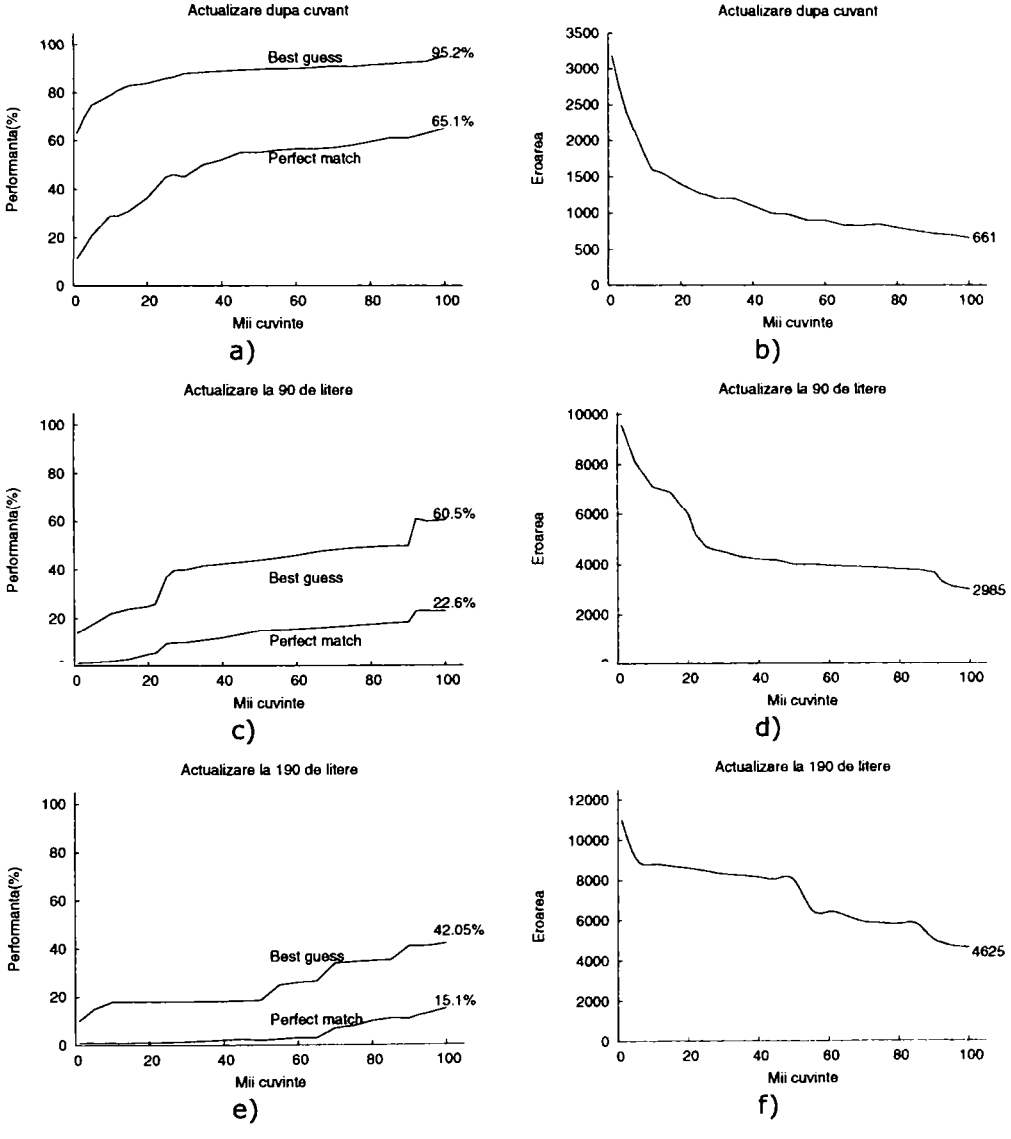


Figura 6.2: *Performanţele NETtalk după prezentarea a 100000 de cuvinte.*

Graficele reprezintă media mai multor rulări, cu puncte de pornire aleatoare în

spaţiul ponderilor, însă diferenţele au fost nesemnificative, lucru explicabil dacă se iau în considerare numărul relativ mare de ponderi din reţea (12566, incluzând şi ponderile de bias).

În figurile 6.2a şi 6.2b sunt prezentate rezultatele în cazul actualizării după cuvânt — ponderile sunt actualizate numai după ce au fost prezentate toate literele unui cuvânt. Aceasta este frecvenţa de actualizare utilizată de Sejnowski şi Rosenberg. De remarcat faptul că ponderile sunt actualizate foarte frecvent, însă neregulat, întrucât cuvintele diferă în lungime (între 1 şi 14, având o medie de 5.4 pentru cele 1000 de cuvinte. Rata de învăţare utilizată a fost de 0.2.

În figura 6.2a este prezentată performanţa în procente, măsurată atât ca potrivire perfectă cât şi pentru cea mai bună alegere. Se observă că în cazul celei mai bune alegeri performanţa creşte rapid până la aproape 90% şi apoi mult mai încet, atingând 94.3% după 100000 de prezentări de cuvinte. Pentru potrivirea perfectă avem o creştere mai moderată, atingând circa 64%, însă se pare că am putea obţine un procentaj şi mai bun dacă s-ar continua antrenarea.

În figura 6.2b este prezentată eroarea standard, calculată ca:

$$E = \sum_{p=1}^{5438} \sum_{j=1}^{26} (t_{pj} - a_{pj})^2 \qquad (6.2)$$

pentru cele 26 de unităţi de ieşire şi cele 5438 de tipare (numărul de litere din cele 1000 de cuvinte).

În figurile 6.2c şi 6.2d sunt prezentate rezultatele în cazul actualizării ponderilor după prezentarea a 90 de litere (dimensiunea setului de antrenare este 90). Pomerleau şi alţii [68] au utilizat această frecvenţă de actualizare a ponderilor pe o arhitectură Warp cu 10 procesoare. A fost utilizată o rată de învăţare de 0.02, necesară pentru a preveni *oscilaţiile*: dimensiunea setului de antrenare este mare, deci se însumează mulţi gradienţi componenţi, iar pentru a păstra modificarea ponderilor la o magnitudine acceptabilă este necesară o asemenea rată mică de învăţare. Dacă modificările ponderilor sunt prea mari se obţin oscilaţii şi nu are loc un proces de învăţare.

Se observă în figura 6.2c că reţeaua are o performanţă scăzută pentru ambele criterii, în comparaţie cu actualizarea la cuvânt. De asemenea, se remarcă faptul că graficele sunt neregulate. Mai mult, eroarea rămâne mare chiar şi după 100000 de prezentări de cuvinte, după cum se observă în figura 6.2d.

În ultimul set de grafice sunt reprezentate rezultatele pentru actualizarea ponderilor după prezentarea a 190 de litere. A fost necesară o rată de învăţare de 0.01, pentru a preveni oscilaţiile. Se constată o performanţă şi mai slabă decât în cele două cazuri precedente.

Cele două grafice – pentru actualizarea după 90 de litere, respectiv după 190 de litere, par să indice faptul că s-ar putea creşte perfomanţele dacă s-ar prelungi durata de antrenare. Experimentele efectuate au confirmat această supoziţie, în cazul actualizării după 90 de litere şi circa 256000 de prezentări de cuvinte, însă pentru actualizarea după 190 de litere rezultatele au fost nesatisfăcătoare şi indicau necesitatea unui număr mult mai mare de reluări.

Au fost efectuate experimente şi pentru dimensiuni ale setului de antrenare mai mari — 512, respectiv 800 de litere, însă performanţele obţinute au fost mult prea slabe.

Concluzia care se poate deduce din rezultatele obţinute este că pe măsură ce ponderile se actualizează mai rar, tot mai multe prezentări de tipare sunt necesare pentru a obţine aceeaşi calitate a antrenării. La o concluzie similară au ajuns atât Bourrely [7], la paralelizarea algoritmului de backpropagation pe un hipercub cu 32 de procesoare, cât şi Witbrock [100], pe o arhitectură GF11 cu 512 procesoare.

6.2 Compresie de imagine

O altă utilizare a reţelelor neuronale multistrat este comprimarea datelor dintr-o imagine prin reducerea redundanţei spaţiale [83].

6.2.1 Suportul teoretic

Structura unei astfel de reţele neuronale este ilustrată în figura 6.3. Reţeaua conţine trei straturi: un strat de intrare, unul de ieşire şi unul ascuns. Stratul de intrare şi stratul de ieşire au acelaşi număr de neuroni ($NI = NO$). Compresia de imagine are loc prin alegerea unei valori pentru NH — numărul de neuroni de pe stratul ascuns — mai mică decât numărul de neuroni de pe stratul de intrare şi cel de ieşire. Astfel, ieşirea stratului ascuns este o imagine comprimată, iar stratul de ieşire de-comprimă imaginea. Un număr mic de unităţi pe stratul ascuns conduce la un raport mare de compresie.

Imaginea de intrare este divizată în blocuri, de exemplu 8×8, 4×4 sau 16×16. Aceste blocuri trebuie convertite în vectori de intrare cu valori normalizate ($x_i \in [0, 1]$) corespunzătoare valorilor pixelilor pentru nivele de gri in $[0, 255]$, întrucât s-a demonstrat [6], [41] faptul că reţelele neuronale lucrează mai eficient atunci când atât intrarea cât şi ieşirea sunt limitate la domeniul $[0, 1]$.

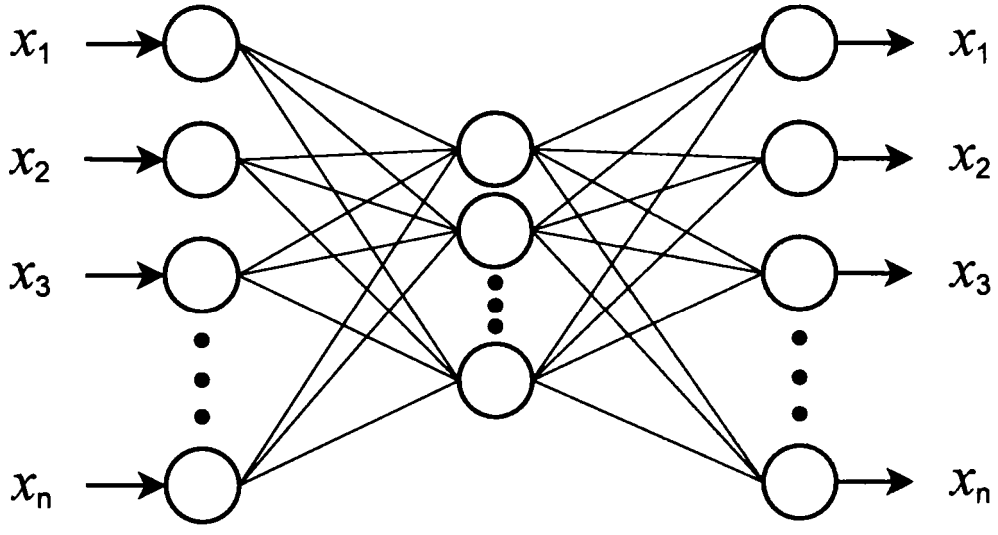


Figura 6.3: *Reţea neuronală pentru compresia de imagine.*

Dacă un vector de intrare este de dimensiune NI, egal cu numărul de pixeli din fiecare bloc, atunci se pot nota cu w_{ji} ponderile conexiunilor dintre stratul de intrare şi cel ascuns, cu $i \in [1, NI]$ şi $j \in [1, NH]$, care mai pot fi considerate ca o matrice de dimensiune $NH \times NI$. Ponderile conexiunilor dintre stratul ascuns şi cel de ieşire, de asemenea pot fi notate cu w'_{ij}, $i \in [1, NO]$ şi $j \in [1, NH]$, care formează de asemenea o matrice, de dimensiune $NO \times NH$. Compresia imaginii poate fi descrisă matematic după cum urmează:

$$h_j = \sum_{i=1}^{NI} w_{ji} x_i, \qquad 1 \leq j \leq NH \tag{6.3}$$

pentru compresie şi

$$\overline{x_i} = \sum_{j=1}^{NH} w'_{ij} h_j, \qquad 1 \leq i \leq NO \tag{6.4}$$

pentru decompresie.

Utilizând o asemenea reţea neuronală, compresia de imagine necesită o fază de antrenare. În această fază, trebuie selectat un set de eşantioane pentru antrenarea reţelei cu algoritmul backpropagation, având fiecare vector de intrare ca ieşire dorită. Astfel, se comprimă intrarea printr-un canal mai îngust, reprezentat de stratul ascuns, după care stratul de ieşire reconstruieşte imaginea.

Pentru utilizare, după prezentarea unui bloc de imagine, se preia ieşirea stratului ascuns, ceea ce reprezintă imaginea comprimată, care apoi poate fi reconstruită utilizând ponderile conexiunilor dintre stratul ascuns şi cel de ieşire, conform relaţiei 6.4.

6.2.2 Simulări şi rezultate

Compresia de imagine necesită o reţea de dimensiuni reduse [13], însă presupune un volum mare de date, atât pentru antrenare cât şi pentru compresie în timp-real. Ca urmare, s-a considerat că varianta cea mai indicată este cea de paralelizare cu partiţionarea datelor.

Pentru efectuarea experimentelor au fost necesari mai mulţi paşi:

Achiziţia de imagine — imaginile au fost scanate cu ajutorul unui scanner şi salvate în format TIFF (Tagged Image Format). Apoi, cu ajutorul programului GIMP, au fost convertite în format BMP, cu nivele de gri (monocrom).

Normalizare — întrucât valorile pixelilor rezultaţi erau întregi, între 0 şi 255, a fost necesară preprocesarea lor, pentru a avea valori în intervalul $[0, 1]$.

Segmentarea imaginii — imaginea a fost divizată în blocuri mai mici, 8×8, şi convertite în vectori. Dacă dimensiunea imaginii nu a fost divizibilă cu dimensiunea blocului dorit, imaginea a trebuit completată cu pixeli cu valoarea zero.

Pregătirea setului de antrenare — s-a generat un fişier în care fiecare bloc a fost scris sub forma a doi vectori identici – perechea de antrenare.

Eliminarea blocurilor similare — după segmentare este posibil să rezulte blocuri identice, ceea ce reprezintă informaţie redundantă şi care ar putea chiar să deranjeze în faza de antrenare.

Antrenarea reţelei neuronale — se furnizează reţelei neuronale tiparele de antrenare şi se efectuează procesul de învăţare până când se atinge nivelul de eroare acceptabil.

Realizarea unui program pentru reconstrucţia imaginii — s–au construit două reţele neuronale, cu câte două straturi în loc de trei, având ponderile conexiunilor rezultate la faza de antrenare. Una dintre reţele va efectua compresia imaginii, iar cealaltă – reconstrucţia.

În figurile 6.4 şi 6.5 sunt reprezentate performanţele unei reţele cu 64 de unităţi la intrare şi 64 de unităţi la ieşire, pentru diferite rapoarte de compresie – numărul de unităţi de pe stratul ascuns de 4, 8, respectiv 16. Graficele prezintă accelerarea faţă de varianta secvenţială pentru faza de antrenare, pentru 16 imagini 256×256, măsurând timpul necesar antrenării pentru a ajunge la o eroare sub 0.1, atât pentru varianta secvenţială, cât şi pentru cea paralelă.

După cum se poate observa, s-a obţinut o accelerare mai bună – şi o eficienţă mai bună – comparativ cu varianta secvenţială, în cazul în care reţeaua neuronală a

Număr procesoare	Dimensiune reţea					
	64-16-64		64-8-64		64-4-64	
	acc	ef	acc	ef	acc	ef
2	1.80	90.00	1.79	89.50	1.75	87.50
3	2.52	84.00	2.50	83.33	2.32	77.33
4	3.27	81.75	3.06	76.54	2.83	70.75
5	4.00	80.00	3.72	74.40	3.35	67.00
6	4.68	78.00	4.36	72.67	3.94	65.60
7	5.43	77.50	5.03	71.86	4.55	64.97
8	6.16	77.00	5.67	70.88	5.13	64.10
9	6.88	76.50	6.36	70.72	5.73	63.62
10	7.62	76.25	7.06	70.60	6.34	63.38
11	8.36	76.00	7.76	70.55	6.95	63.18
12	9.09	75.75	8.46	70.48	7.56	63.02
13	9.81	75.50	9.13	70.21	8.19	62.98
14	10.50	75.00	9.75	69.62	8.80	62.90
15	11.18	74.50	10.35	69.01	9.67	62.47
16	11.90	74.38	10.95	68.44	9.98	62.38

Tabela 6.2: *Performanţele compresiei de imagini: SunBlade150*

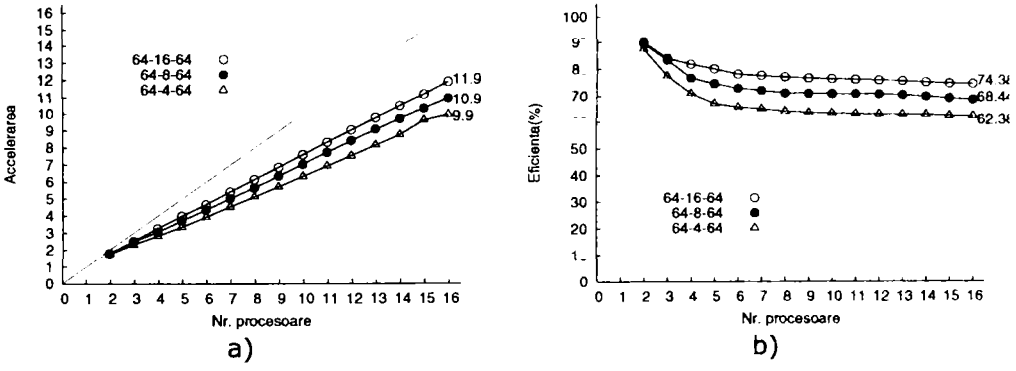


Figura 6.4: *Performanţele compresiei de imagini: SunBlade150*

avut cele mai multe unităţi pe stratul ascuns (16), deşi aceasta duce la compresia cea mai slabă, însă au fost comparaţi timpii necesari atingerii unei rate a erorii totale de sub 10^{-2}, fără a lua în consideraţie performanţele compresiei de imagine.

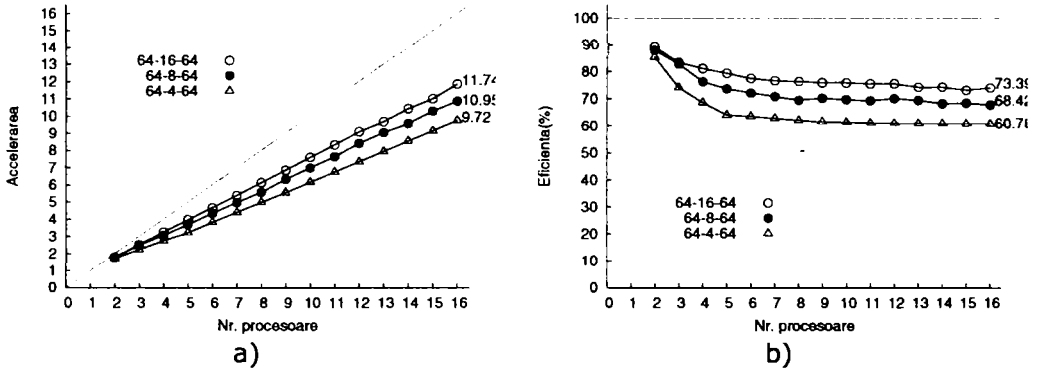
Nu se observă diferenţe semnificative între rezultatele pentru cele două platforme hardware utilizate, SunBlade, respectiv Linux. În ambele cazuri, accelerarea, respectiv eficienţa cea mai bună se obţine pentru numărul cel mai mare de neuroni de pe stratul ascuns. Numărul mai mare de neuroni pe stratul ascuns duce la un număr mai mare de ponderi în reţea, deci un volum mai mare de calcul, iar rezultatele confirmă concluziile din capitolele anterioare, şi anume faptul că eficienţa este cu atât mai bună cu cât volumul de calcul *per procesor* este mai mare.

6.3 Recunoaşterea ţintelor de sonar

O altă aplicaţie utilizată în testarea reţelelor neuronale este cea de recunoaştere a ţintelor de sonar. Gorman şi Sejnowski [29], [28] au realizat un studiu în care au

Număr procesoare	Dimensiune reţea					
	64-16-64		64-8-64		64-4-64	
	acc	ef	acc	ef	acc	ef
2	1.79	89.29	1.76	88.09	1.71	85.51
3	2.51	83.53	2.49	82.99	2.23	74.25
4	3.25	81.30	3.06	76.41	2.75	68.71
5	3.98	79.55	3.69	73.78	3.21	64.10
6	4.66	77.65	4.34	72.25	3.82	63.60
7	5.38	76.81	4.97	70.94	4.40	62.90
8	6.12	76.51	5.56	69.51	4.97	62.10
9	6.83	75.90	6.31	70.15	5.54	61.59
10	7.59	75.88	6.97	69.65	6.14	61.38
11	8.31	75.51	7.62	69.27	6.73	61.14
12	9.07	75.57	8.40	70.02	7.32	61.02
13	9.65	74.26	9.02	69.40	7.92	60.95
14	10.40	74.28	9.54	68.18	8.53	60.90
15	10.97	73.11	10.26	68.38	9.12	60.81
16	11.83	73.97	10.83	67.72	9.72	60.78

Tabela 6.3: *Performanţele compresiei de imagini: Linux*



a) b)

Figura 6.5: *Performanţele compresiei de imagini: Linux*

utilizat o reţea neuronală pentru diferenţierea rocilor şi a minelor de pe fundul mării cu ajutorul ecoului sonarului. Semnalele de sonar reflectate prezintă diferenţe în privinţa volumului de energie pentru anumite frecvenţe, în funcţie de obiectul întâlnit. În experimentele lor, Gorman şi Sejnowski au cuantizat frecvenţa răspunsului într-un set de câte 60 de eşantioane în care fiecare valoare reprezintă energia semnalului de răspuns, în cazul unui cilindru metalic, respectiv al unei roci de formă aproximativ cilindrică.

6.3.1 Configuraţia reţelei

Reţeaua neuronală utilizată este prezentată în figura 6.6. Reţeaua este complet conectată, cu 60 de unităţi de intrare cu valori reale, 24 de unităţi pe stratul ascuns şi două unităţi pe stratul de ieşire ("rocă" / "mină").

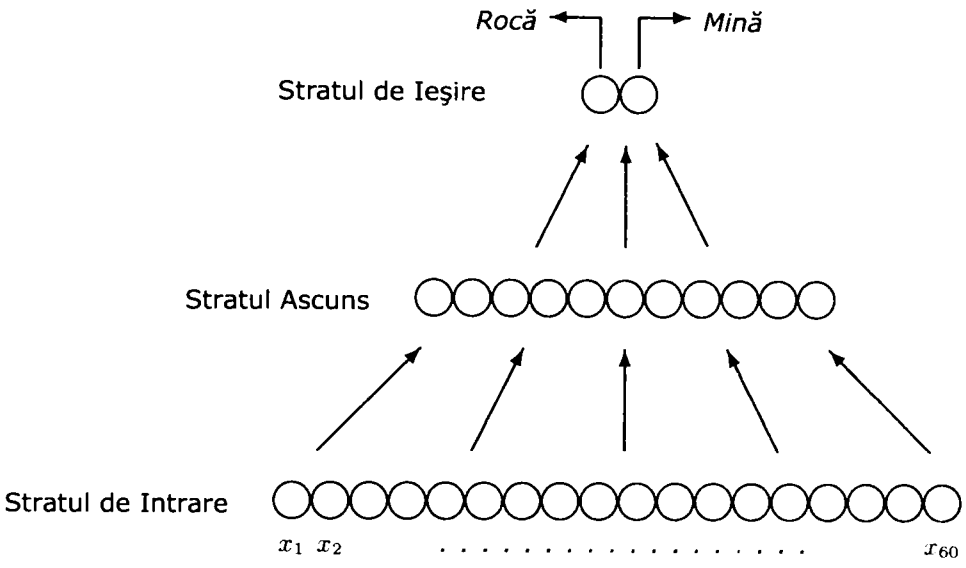


Figura 6.6: *Configuraţia reţelei pentru recunoaşterea ţintelor de sonar*

6.3.1.1 Setul de Date

Pentru antrenare au fost utilizate două seturi de date: 111 tipare obţinute prin reflexia pe un cilindru metalic, la diferite unghiuri, respectiv 97 de tipare provenind de la roci, în condiţii similare.

Fiecare tipar constă dintr-un set de 60 de valori între 0.0 şi 1.0, fiecare număr reprezentând energia într-o anumită bandă de frecvenţă, integrată peste un anumit interval de timp.

6.3.2 Simulări şi rezultate

A fost aleasă o rată de învăţare de 2.0 şi un moment de 0.0. Ponderile iniţiale au fost valori aleatoare uniform distribuite în intervalul $[-0.3, +0.3]$. Reţeaua a fost antrenată pe parcursul a 300 de epoci.

Întrucât pe stratul de ieşire sunt numai două unităţi, au fost efectuate experimente numai cu partiţionarea datelor, pe un număr variabil de procesoare. Figurile 6.7, 6.8 prezintă graficele accelerării, respectiv eficienţei în cazul celor două tipuri de maşini pe care s-au efectuat experimentele.

6.4 Concluzii

Algoritmul de antrenare cu backpropagation a fost paralelizat pe baza celor două abordări – cea cu partiţionarea datelor, prezentată în capitolul 6, respectiv partiţionarea reţelei, prezentată în capitolul 7. După cum s-a văzut, atunci când dimensiunea setului de antrenare este mare, varianta cu partiţionarea datelor poate utiliza în mod eficient mai multe procesoare, independent de dimensiunile reţelei neuronale[3] .

Prin comparaţie, varianta cu partiţionarea reţelei prezintă o serie de avantaje:

- nu depinde de dimensiunea setului de antrenare;

[3]De fapt, *aproape* independent de dimensiunile reţelei neuronale

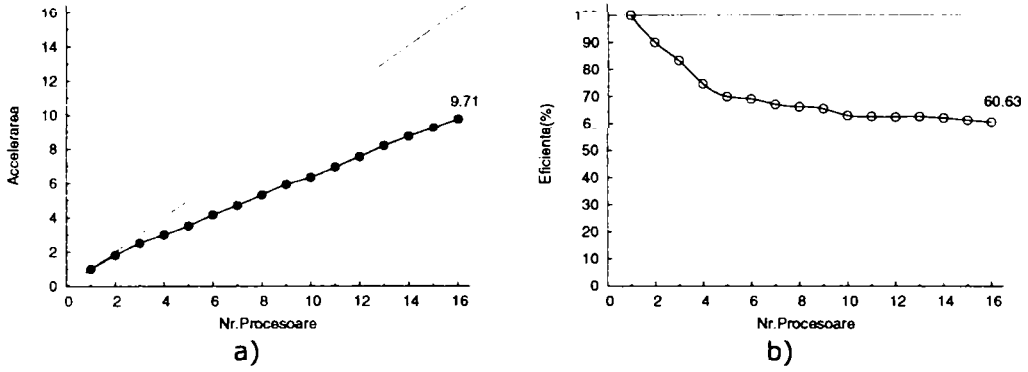


Figura 6.7: *Performanţele aplicaţiei Sonar: SunBlade150.*

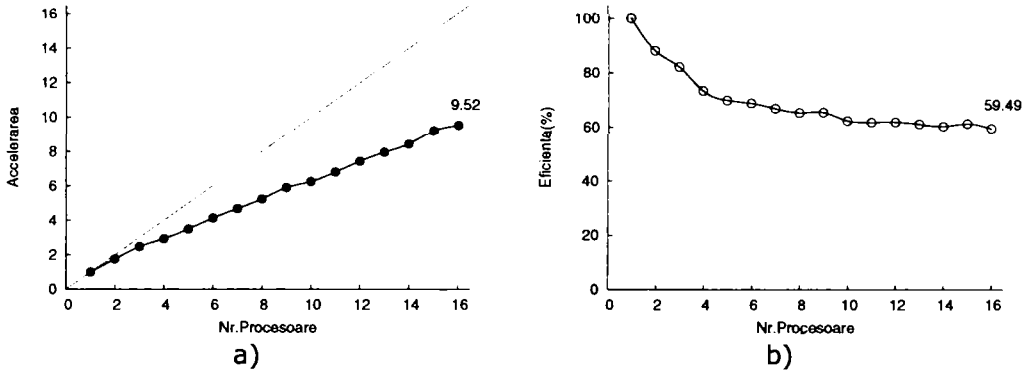


Figura 6.8: *Performanţele aplicaţiei Sonar: Linux.*

- permite actualizări frecvente ale ponderilor în reţea;

însă necesită dimensiuni mari ale reţelei neuronale pentru a utiliza cât mai eficient mai multe procesoare.

Experimentele efectuate cu aplicaţia NETtalk au arătat că paralelizarea cu partiţionarea datelor este superioară celei cu partiţionarea reţelei, dacă luăm ca unic criteriu numărul de operaţii efectuate, însă este posibil ca rezultatul antrenării să fie nul. Acest lucru a fost sesizat şi de către alţi cercetători, pe implementările paralele realizate [100], [7]. [68] arată că: *Metoda tradiţională de măsurare a performanţelor unui simulator sub forma numărului de conexiuni pe secundă poate conduce la rezultate eronate. Mai precis, este posibil ca implementările de reţele neuronale care simulează un număr mare de tipare în paralel, şi care deci necesită simularea mai multor tipare înainte de actualizarea ponderilor, să înveţe mai încet şi mai puţin robust decât implementările care actualizează ponderile mai frecvent.*

Ca urmare, singura modalitate corectă de comparaţie între diferitele implementări ale algoritmului backpropagation este să se utilizeze exact aceiaşi parametri: setul de antrenare, dimensiunea reţelei, frecvenţa de actualizare a ponderilor.

Alte aplicaţii au arătat că pentru obţinerea unor performanţe bune sunt necesare volume mari de date de antrenare – vezi compresia de imagine – mult mai mari decât numărul de procesoare utilizate.

Implementarea aleasă trebuie deci să fie în funcţie de aplicaţia de reţele ne-

uronale *dată*. Uneori, se obţin rezultate mai bune dacă se utilizează *actualizarea la epocă*, alteori *actualizarea la tipar* se poate dovedi mai eficientă. Deşi eficienţa cu care sunt utilizate procesoarele este departe de cea optimă, timpul necesar obţinerii unor rezultate concludente se reduce drastic, ceea ce va permite reluarea experimentelor, până la determinarea valorilor optime pentru parametrii antrenare — rată de învăţare, momentul, frecvenţa de actualizare a ponderilor, dimensiunea reţelei neuronale.

7.1 Concluzii

În cadrul acestei teze a fost studiată paralelizarea algoritmului de antrenare backpropagation pentru reţele neuronale feed-forward. Principalele obiective ale studiului de faţă au vizat realizarea unor algoritmi de antrenare bazaţi pe metoda backpropagation, care, spre deosebire de alte cercetări în domeniu, care au vizat implementări paralele pe arhitecturi paralele şi masiv paralele, să evolueze într-o reţea de calculatoare, precum şi studiul performanţelor acestor algoritmi:

- Dezvoltarea şi implementarea unui algoritm de antrenare bazat pe metoda partiţionării datelor – mai multe procesoare rulează acelaşi algoritm de antrenare, pe seturi diferite de date; rezultatele antrenării se colectează, se evaluează şi se decide dacă să se continue antrenarea, până la atingerea nivelului de învăţare prescris.

  Experimentele efectuate au demonstrat posibilitatea utilizării unei asemenea abordări în cazul în care pot fi utilizate seturi de tipare de antrenare de dimensiuni foarte mari. S-a constatat o relaţie directă între dimensiunile setului de tipare de antrenare şi eficienţa acestui algoritm.

  Pe baza acestei abordări, s-au obţinut accelerări considerabile ale procesului de învăţare, în privinţa timpului total necesar, avantaj contrabalansat însă de necesarul excesiv de memorie, provenit din faptul că fiecare procesor trebuie să păstreze o copie a întregii reţele neuronale simulate. Acest dezavantaj major duce la o limitare – uneori drastică – a dimensiunii reţelei neuronale, şi deci a problemelor abordate.

- Dezvoltarea şi implementarea unui algoritm de antrenare bazat pe metoda de partiţionare a reţelei – fiecare procesor simulează câte o porţiune din reţeaua neuronală, respectiv neuronii de pe fiecare strat sunt distribuiţi cât mai uniform către toate procesoarele. Principala problemă care a trebuit rezolvată în această abordare este reducerea la minimum a comunicaţiei între procesoare, având în vedere că este necesară transmisia de date între neuroni de pe straturi diferite, simulaţi de procesoare diferite.

  Deşi, comparat cu metoda partiţionării datelor, accelerările obţinute – şi implicit eficienţa – au fost mai mici, această variantă a prezentat o serie de avantaje, dintre care amintim faptul că este posibilă actualizarea ponderilor *la tipar*, modalitate care poate fi importantă în unele aplicaţii reale.

  Un alt avantaj este posibilitatea extinderii limitărilor în dimensiunile reţelei neuronale posibil de simulat, la valori virtual imposibil de simulat în condiţii acceptabile pe o singură maşină. Rezultatele experimentelor efectuate în capitolul 5 demonstrează faptul că eficienţa acestui algoritm a fost cu atât mai bună şi a fost posibilă utilizarea mai multor procesoare cu cât reţeaua neuronală de simulat a fost mai mare, şi independent de dimensiunile setului de tipare de antrenare.

Rezultatele experimentelor efectuate pentru cele două simulatoare sugerează o serie de valori minime pentru dimensiunea setului de tipare de antrenare, pentru metoda cu paralelizarea setului de antrenare, respectiv pentru dimensiunea reţelei, în cazul metodei cu paralelizarea nodurilor, valori pentru care eficienţa obţinută va fi cel puţin satisfăcătoare. Aceste valori sunt precizate sintetic în tabelul 7.1. În tabel au putut fi incluse şi recomandări pentru reţele de 32 de staţii, întrucât graficele eficienţei au fost linii aproape orizontale, după cum s–a precizat şi la capitolele respective. Aceleaşi concluzii sugerează faptul că tabelul ar putea continua şi pentru un număr mai mare de maşini care rulează fiecare dintre simulatoare.

În cazul metodei cu partiţionarea datelor, s-a constatat că eficienţa, respectiv accelerarea, iau valori acceptabile atunci când numărul de tipare din setul de

Nr. procesoare	Partiţionarea datelor (Nr. tipare)	Partiţionarea reţelei (dimensiune reţea neuronală)
4	40	$40 \times 40 \times 4$
8	80	$80 \times 80 \times 8$
16	160	$160 \times 160 \times 16$
32	320	$320 \times 320 \times 32$

Tabela 7.1: *Dimensiuni minime optime.*

antrenare, alocate fiecărei maşini, este de cel puţin zece tipare. Astfel, volumul computaţional dintre două actualizări de ponderi, devine mult mai mare decât timpul necesar comunicaţiei, şi astfel accelerarea, respectiv eficienţa nu vor fi afectate într-o măsură prea mare de timpul necesar transmisiei de date între maşini.

Pentru paralelizarea cu partiţionarea reţelei, valori bune ale eficienţei şi accelerării au fost obţinute atunci când fiecărei maşini i-au fost alocate cel puţin 10 unităţi de pe cel puţin două dintre cele trei straturi. Performanţele nu au fost prea mult afectate de faptul că pe cel de-al treilea strat au fost mai puţini neuroni — dar având cel putin câte un neuron pentru fiecare procesor.

7.2 Rezumat al contribuţiilor

Lucrarea aduce o serie de contribuţii în domeniul procesării paralele a fazei de antrenare a reţelelor neuronale:

1. dezvoltarea unui algoritm de distribuire în vederea execuţiei cu partiţionarea datelor, cu scopul minimizării timpului de antrenare a unei reţele neuronale;
2. este propusă, şi a fost implementată, o schemă de distribuire a task-urilor şi de colectare a rezultatelor, care să respecte tehnica de antrenare cu backpropagation;
3. realizarea simulatorului DataParSim, utilizând o strategie avansată de partiţionare a datelor: un program *administrator* transmite câte o pondere a reţelei neuronale către procesele *slave*, care o utilizează pentru propagarea tuturor tiparelor de antrenare;
4. evaluarea necesarului de memorie pentru fiecare dintre cei doi algoritmi cu partiţionarea datelor, cu posibilitatea de determinare a dimensiunii maxime a reţelei neuronale care se poate antrena utilizând aceşti algoritmi;
5. dezvoltarea unui algoritm pentru partiţionarea reţelei, care să permită depăşirea limitelor impuse asupra problemelor de tratat. Algoritmul se bazează pe o secţionare a reţelei neuronale şi distribuirea cât mai echilibrată, având în vedere şi minimizarea comunicaţiei în reţeaua de calculatoare.
6. o schemă de calcul care reformulează expresiile utilizate în algoritmul clasic de backpropagation, care permite calculul valorilor δ în cazul strategiei cu partiţionarea reţelei.
7. realizarea simulatorului NetParSim, bazat pe strategia de paralelizare cu partiţionarea reţelei.
8. evaluarea performanţelor simulatoarelor DataParSim şi NetParSim, atât pentru reţele neuronale generice, cât şi pentru aplicaţii uzuale.
9. enunţarea de recomandări în vederea utilizării cât mai eficiente a resurselor de calcul.

Investigaţiile pe aceste teme au fost prezentate în [BaC03], [Ba07a] şi [BCP07].

7.3 Concluzii finale

Au fost prezentate două strategii de paralelizare – cu partiţionarea datelor, respectiv cu partiţionarea reţelei – care au fost implementate pe o arhitectură alcătuită din mai multe calculatoare interconectate printr-o reţea locală (LAN). Nu se poate însă preciza care dintre cele două abordări este de preferat în general. Alegerea depinde de problema de reţele neuronale tratată şi de frecvenţa cu care se fac actualizările de ponderi.

Dacă, din raţiuni specifice problemei tratate, se preferă un set mare de tipare de antrenare, probabil că se va utiliza o paralelizare cu partiţionarea datelor, care, în cazul utilizării mai multor procesoare va conduce la o accelerare mai bună decât în cazul paralelizării cu partiţionarea reţelei, cel puţin în cazul unei reţelei neuronale de dimensiuni relativ mici.

Atunci când sunt necesare actualizări frecvente ale ponderilor trebuie utilizată paralelizarea cu partiţionarea reţelei, întrucât eficienţa acestui algoritm nu depinde de o dimensiune mare a setului de antrenare. În plus, este singurul algoritm care să permită actualizarea după un tipar. Totuşi, pentru a obţine o eficienţă ridicată, fiecare procesor trebuie să prelucreze mai multe unităţi din fiecare strat al reţelei neuronale. Aceasta conduce la limitarea drastică a numărului de procesoare care pot fi utilizate pentru simularea unor reţele mici. Însă atunci când atât dimensiunile reţelei cât şi ale setului de antrenare sunt reduse, nici durata unui ciclu de antrenare nu este mare, nici chiar pentru algoritmul secvenţial.

Trebuie subliniat faptul că la alegerea simulatorului paralel nu trebuie să se aleagă o dimensiune mare a setului de tipare după care se face actualizarea doar pentru a obţine o aplicare eficientă a uneia dintre metodele de paralelizare, întrucât mărirea dimensiunii setului după care se face actualizarea poate să reducă drastic viteza de învăţare şi să scadă calitatea învăţării în privinţa posibilităţilor de generalizare. Acest lucru a fost subliniat în cazul aplicaţiei NETtalk (secţiunea 6.1), când actualizarea la cuvânt – sau la tipar – a condus la performanţe mult mai bune.

De aceea, la compararea diferitelor implementări de simulatoare de reţele neuronale trebuie să se ia ca măsură timpul absolut de execuţie necesar pentru a atinge un anumit nivel de perfecţiune pe o problemă specifică de reţele neuronale. Desigur, aceasta nu este o soluţie satisfăcătoare, întrucât influenţa dimensiunii setului de antrenare după care se face actualizarea asupra vitezei de învăţare şi a posibilităţilor de generalizare variază de la problemă la problemă. Pot exista probleme în care actualizarea ponderilor după o epocă este superioară actualizărilor mai frecvente, în privinţa vitezei de învăţare şi a calităţii. Cu alte cuvinte, probabil că nu este posibil să se facă o recomandare *generală* în privinţa schemei de paralelizare de utilizat, ci doar pentru tipuri specifice de probleme de reţele neuronale.

7.4 Perspective de cercetare

Studiul de faţă demonstrează posibilitatea accelerării timpului de antrenare pentru reţele neuronale feed-forward cu algoritmul backpropagation, utilizând resursele disponibile în laboratoarele departamentului. Prin reducerea considerabilă obţinută, teza de faţă oferă o unealtă utilă celor care explorează domeniul reţelelor neuronale, permiţând reluarea multiplă a experimentelor, cu parametri diferiţi, până la obţinerea rezultatelor dorite.

Teza deschide calea unor investigaţii ulterioare, dintre care au fost identificate câteva:

- determinarea dimensiunilor optime a reţelelor neuronale
- adaptarea tehnicilor utilizate şi pentru alte tipuri de reţele – nu doar feed-forward, ci şi cele recurente sau cele parţial conectate.

- investigarea posibilităţilor de paralelizare a altor algoritmi de antrenare, un prim exemplu fiind algoritmul de antrenare *quickprop* [24] şi cel al gradientului conjugat [59].
- studiul posibilităţilor de paralelizare a tehnicilor de preprocesare în vederea reducerii dimensionalităţii problemelor.
- experimente în vederea determinării utilităţii mai multor straturi intermediare.

[1] Yaser S. Abu-Mostafa and Demetri Psaltis. Optical neural computers. *Scientific American*, 256(3):88–95 (Intl. ed. 66–73), March 1987.

[2] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski. A learning algorithm for Boltzmann machines. *Cognitive Science*, 9:147–169, 1985.

[3] I. Aleksander and H. Morton. *An Introduction to Neural Computing.* Chapman and Hall, 1990.

[4] James A. Anderson. A simple neural network generating an interactive memory. *Mathematical Biosciences*, 14:197–220, 1972.

[5] W. R. Ashby. *Design for a Brain*. Wiley & Sons, New York, 1952.

[6] Y. Benbenisti, D. Kornreich, H. B. Mitchell, and P. A. Schaefer. New simple three-layer neural network for image compression. *Optical Engineering*, 36:1814–1817, June 1997.

[7] Jean Bourrely. Paralelisation of a neural learning algoritm on a hypercube. In F. André and J. P. Verjous, editors, *Hypercube and Distributed Computers*, pages 219–229. Elsevier Science Publishers B.V. (North Holland), 1989.

[8] V. Braitenberg. *Vehicles: Experiments in Synthetic Psychology*. MIT Press, Cambridge, MA, USA, 1984.

[9] A. E. Bryson and Y.-C. Ho. *Applied Optimal Control*. Blaisdell, New York, 1969.

[10] S.R. Cajal. *Histology of the nervous system of man and vertebrates*. Oxford University Press, New York, 1995.

[11] G. F. Carey, Y. Shen, and R. T. McLay. Parallel conjugate gradient performance for least-squares finite elements and transport problems. *International Journal for Numerical Methods in Fluids*, 28:1421–1440, December 1998.

[12] G. Chinn, K. A. Grajski, C. Chen, C. Kuszmaul, and S. Tomboulian. Systolic array implementations of neural nets on the maspar mp–1 massively parallel processor. In *International Joint Conference on Neural Networks*, volume 2, pages 169–173, San Diego, CA, USA, 1990.

[13] K. B. Cho, Cheol Hoon Park, and Soo-Young Lee. Image compression using multi-layer perceptron with block classification and SOFM coding. In *Proc. WCNN'94 – World Congress on Neural Networks*, volume III, pages 26–31, Hillsdale, NJ, 1994. INNS, Lawrence Erlbaum.

[14] M. A. Cohen and Steven Grossberg. Stability of global pattern formation and parallel memory storage by competitive neural networks. *IEEE Transactions on Systems, Man and Cybernetics*, 13:815–826, 1983.

[15] Y. Le Cun. Une procédure d'apprentissage pour Réseau à seuil assymétrique. *Cognitiva 85: A la Frontière de l'Intelligence Artificielle des Sciences de la Connaissance des Neurosciences*, pages 599–604, 1985.

[16] A. D'Acierno and R. Vaccaro. A parallel implementation of the back-propagation of errors learning algorithm on a simd parallel computer. In Kappen B. and Gielen S., editors, *Proc. of International Conference on Artificial Neural Networks (ICANN '93)*, pages 1074–1077, Berlin/Heidelberg/New York/Tokyo, 1993. Springer.

[17] Y.-H. Dai. A family of hybrid conjugate gradient methods for unconstrained optimization. *Mathematics of Computation*, 72:1317–1328, 2003.

[18] Y.-H. Dai, L.-Z. Liao, and D. Li. On Restart Procedures for the Conjugate Gradient Method. *Numerical Algorithms*, 35:249–260, 2004.

[19] Y. H. Dai and Y. Yuan. A three-parameter family of nonlinear conjugate gradient methods. *Mathematics of Computation*, 70:1155–1167, 2001.

[20] J. del R. Millan and P. Bofill. Learning by back-propagation: Computing in a systolic way. In *Parallel Architectures and Language Europe - Vol.2: Parallel Languages (PARLE '89)*, pages 235–252, Berlin - Heidelberg - New York, June 1989. Springer.

[21] T. DeLillo and T. Hrycak. A stopping rule for the conjugate gradient regularization method for ill-posed problems. *Acoustical Society of America Journal*, 112:2381–2381, November 2002.

[22] C. Ernoult. Performance of backpropagation on a parallel transputer-based machine. In *Neuro-Nimes '88: International Workshop on Neural Networks and their Applications*, pages 311–24, Nimes, France, 1988.

[23] Federico Faggin. Neural network hardware. In *IEEE International Joint Conference on Neural Networks (6th IJCNN'92)*, volume I, page 153, Baltimore, MD, June 1992. IEEE/INNS.

[24] S. E. Fahlman. Faster-learning variations on back-propagation: an empirical study. In D.S. Touretzky, G.E. Hinton, and T.J. Sejnowski, editors, *Proc. of the 1988 Connectionist Models Summer School*, pages 38–51, San Mateo, CA, 1989. Morgan Kaufmann.

[25] B. M. Forrest, D. Roweth, N. Stroud, D. J. Wallace, and G. V. Wilson. Implementing neural network models on parallel computers. *The Computer Journal*, 30(5):413–419, October 1987.

[26] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon, and David W. Walker. *Solving Problems On Concurrent Processors, Volume 1, General Techniques and Regular Problems*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[27] C. K. Gan, P. D. Haynes, and M. C. Payne. Preconditioned conjugate gradient method for the sparse generalized eigenvalue problem in electronic structure calculations. *Computer Physics Communications*, 134:33–40, February 2001.

[28] R. P. Gorman and T. J. Sejnowski. Analysis of hidden units in a layered network trained to classify sonar targets. *Neural Networks*, 1:75–89, 1988.

[29] R. Paul Gorman and T. J. Sejnowski. Learned classification of sonar targets using a massively parallel network. *IEEE Tran. on Acoustics, Speech, and Signal Processing*, 36(7):1135–1140, July 1988.

[30] E. Hartman, J. D. Keeler, and J. M. Kowalski. Layered neural networks with Gaussian hidden units as universal approximations. *Neural Computation*, 2(2):210–215, 1990.

[31] Simon Haykin. *Neural Networks*. Macmillan College Publishing Company, New York, 1994.

[32] D. O. Hebb. *The Organization of Behavior*. Wiley & Sons, New York, 1949.

[33] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, 117:500–544, 1952.

[34] J. J. Hopfield. Neural networks and physical systems with emergent collective behavior. *PNAS USA*, 79:2554–2558, 1982.

[35] J. J. Hopfield. Collective processing and neural states. In C. Nicollini, editor, *Modeling and Analysis in Biomedicine*, pages 371–389. 1984.

[36] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[37] K. Hornik, M. Stinchcombe, and H. White. Universal approximation of an unknown function and its derivatives using multilayer feedforward networks. *Neural Networks*, 3:551–560, 1990.

[38] R. A. Hyman, J. Tetzlaff, and B. Doporcyk. Simple and unified derivation of conjugate gradient and variable metric minimization. *APS Meeting Abstracts*, pages 1069–+, March 2003.

[39] Robert A. Jacobs. Increased rates of convergence through learning rate adaptation. *Neural Networks*, 1(4):295–307, 1988.

[40] H. Jiang and W. Yang. Conjugate-gradient optimization method for orbital-free density functional calculations. *jcp*, 121:2030–2036, August 2004.

[41] J. Jiang. Image compression with neural networks – a survey. In *Signal Processing: Image Communication 1999*, volume 14, pages 737–760, 1999.

[42] E. M. Johansson, F. U. Dowla, and D. M. Goodman. Back-propagation learning for multi-layer feed-forward neural networks using the conjugate gradient method. *International Journal of Neural Systems*, 2:291–302, 1991.

[43] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[44] Christoph Klawun and Charles L. Wilkins. A novel algorithm for local minimum escape in back-propagation neural networks: application to the interpretation of matrix isolation infrared spectra. *Journal of Chemical Information and Computer Sciences*, 34(4):984–993, 1994.

[45] B. Knutson, J. Taylor, M. Kaufman, R. Peterson, and G. Glover. Distributed neural representation of expected value. *Journal of Neuroscience*, 25(19):4806–4812, 2005.

[46] Randal A. Koene and Yoshio Takane. Discriminant component pruning: Regularization and interpretation of multi-layered back-propagation networks. In *Neural Computation*, volume 11, pages 783–802, 1999.

[47] Teuvo Kohonen. Correlation matrix memories. *IEEE Transactions on Computers*, C-21:353–359, 1972.

[48] Ben J. A. Kröse and P. Patrick van der Smagt. *An Introduction to Neural Networks*. University of Amsterdam, Amsterdam, 1993.

[49] Toshiro Kubota. Massively parallel networks for edge localization and contour integration-adaptable relaxation approach. In *Neural Networks*, volume 17, pages 411–425, 2004.

[50] Han-Wook Lee and Chan-Ik Park. An efficient parallel block backpropagation learning algorithm in transputer mesh-connected parallel computers. In *IEICE Trans. Inf. & Syst.*, volume E83-D, pages 1622–1630, 2000.

[51] Tara M. Madhyastha and Daniel A. Reed. Learning to classify parallel input/output access patterns. *IEEE Trans. Parallel Distrib. Syst.*, 13(8):802–813, 2002.

[52] Q.M. Malluhi, M.A. Bayoumi, and T.R.N. Rao. An efficient mapping of multilayer perceptron with backpropagation anns on hypercubes. In *Proc. of Symposium on Parallel and Distributed Systems*, pages 368–375. IEEE Computer Society Press, 1994.

[53] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–137, 1943.

[54] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equations of state calculations by fast computing machine. *Journal of Chemical Physics*, 21:1087–1091, 1953.

[55] G. Meurant. Estimates of the l_2 norm of the error in the conjugate gradient algorithm. *Numerical Algorithms*, 40:157–169, 2005.

[56] M. L. Minsky. *Theory of neural-analog reinforcement systems and its application to the brain-model problem*. Ph.D. dissertation, Princeton University, N.J., 1954.

[57] Marvin Minsky. Steps toward artificial intelligence. pages 406–450, 1995.

[58] Marvin Minsky and Seymour Papert. *Perceptrons*. MIT Press, Cambridge, MA, 1969.

[59] Martin Fodslette Moller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*, 6:525–533, 1993.

[60] F.C. Morabito. Fast backpropagation using modified sigmoidal functions. In Marinaro M. and Morasso P., editors, *ICANN'94: Proceedings of the International Conference on Artificial Neural Networks*, pages 537–540, Berlin/Heidelberg/New York/Tokyo, 1994. Springer.

[61] K. Nakano. Associatron - a model of associative memory. *IEEE transactions on Systems, Man, and Cybernetics*, SMC2(3):380–388, 1972.

[62] Tomas Nordström. *Highly Parallel Computers for Artificial Neural Networks*. PhD thesis, Luleå University of Technology, Luleå, Sweden, 1995.

[63] D. B. Parker. Learning logic. Technical Report TR-47, Center for Computational Research in Economics and Management Science, MIT, Cambridge, MA, 1985.

[64] D. B. Parker. Second order back propagation: an optimal adaptive algorithm for any adaptive network. *Proceedings of the IEEE First Annual International Conference on Neural Networks*, 1987.

[65] Alpha V. Pernia-Espinoza, Joaquin B. Ordieres-Mere, Francisco J. Martinez de Pison, and Ana Gonzalez-Marcos. Tao-robust backpropagation learning algorithm. *Neural Netw.*, 18(2):191–204, 2005.

[66] M. Pethick, M. Liddle, P. Werstein, and Z. Huang. Parallelization of a backpropagation neural network on a cluster computer. In *Proceedings of the 15th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, pages 574–582, Marina Del Rey, California, 2003.

[67] A. Polze and M. Malek. Parallel computing in a world of workstations. In M. H. Hamza, editor, *Proceedings of the Seventh IASTED/ISMM International Conference Parallel and Distributed Computing and Systems*, pages 72–4. IASTED-ACTA Press, Anaheim, CA, USA, 1995.

[68] D. A. Pomerleau, G. L. Gusciora, D. S. Touretzky, and H. T. Kung. Neural network simulation at warp speed: How we got 17 million connections per second. In *IEEE Second International Conference on Neural Networks (2nd ICNN'88)*, volume II, pages 143–150, San Diego, CA, jul 1988. IEEE.

[69] J.M.D. Powell. Restart procedures for the conjugate gradient method. *Math. Prog.*, 12:241–254, 1977.

[70] M. Raus and W. Ameling. A parallel algorithm for a dynamic eta/alpha estimation in backpropagation learning. In Marinaro M. and Morasso P., editors, *ICANN'94: Proceedings of the International Conference on Artificial Neural Networks*, pages 639–642, Berlin/Heidelberg/New York/Tokyo, 1994. Springer.

[71] N. Rochester, J. H. Holland, L. H. Haibt, and W. L. Duda. Tests on a cell assembly theory of the action of the brain, using a large digital computer. *IRE Transactions on Information Theory*, 2:80–93, 1956.

[72] F. Rosenblatt. The perceptron: A probabilistic model for information storage in the brain. *Psychological review*, 65:386–408, 1958.

[73] F. Rosenblatt. Perceptron simulation experiments. *Proceedings of the IRE*, 48:301–309, 1960.

[74] F. Rosenblatt. *Principles of Neurodynamics*. Spartan, New York, 1962.

[75] Frank Rosenblatt. On the convergence of reinforcement procedures in simple perceptrons. Report, Cornell Aeronautical Laboratory, Ithaca, New York, 1960.

[76] D. E. Rumelhart and J. L. McClelland. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. I: Foundations.* MIT Press, Cambridge, MA, 1986.

[77] David E. Rumelhart, Geoffrey E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart, J. L. McClelland, and the PDP research group, editors, *Parallel distributed processing: Explorations in the microstructure of cognition, Volume 1: Foundations*. MIT Press, 1986.

[78] P. Saratchandran, N. Sundararajan, and Shou King Foo. *Parallel Implementations of Backpropagation Neural Networks on Transputers: A Study of Training Set Parallelism*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2000.

[79] J. T. Schwartz. The new connectionism: Developing relationships between neuroscience and artificial intelligence. In S. R. Graubard, editor, *The Artificial Intelligence Debate: False Starts, Real Foundations*, pages 123–141. MIT Press, Cambridge, MA, 1988.

[80] Udo Seiffert. Artificial neural networks on massively parallel computer hardware. In *ESANN 2002, 10th European Symposium on Artificial Neural Networks*, pages 319–220, Bruges, Belgium, 2002.

[81] Terrence J. Sejnowski and Charles Rosenberg. NETtalk: A parallel network that learns to read aloud. Technical Report JHU-EECS-86-01, Johns Hopkins University, 1986.

[82] Fernando M. Silva and Luis B. Almeida. Speeding up backpropagation. In R. Eckmiller, editor, *Advanced Neural Computers*, pages 151–158. Elsevier North Holland, Amsterdam, 1990.

[83] Noboru Sonehara, Mitsuo Kawato, Sei Miyake, and Kazunari Nakane. Image data compression using a neural network model. In *Proc. of Int. Joint Conference on Neural Networks*, volume 2, pages 35–40, Washington, DC, 1989.

[84] K. Steinbuch. Die lernmatrix. *Kybernetik (Biological Cybernetics)*, 1:36–45, 1961.

[85] Thomas L. Sterling, John Salmon, Donald J. Becker, and Daniel F. Savarese. *How to build a Beowulf: a guide to the implementation and application of PC clusters*. MIT Press, Cambridge, MA, USA, 1999.

[86] Josef Stoer and Roland Bulirsch. *Introduction to Numerical Analysis*. Springer Verlag, 2002.

[87] N. Sundararajan and P. Saratchandran, editors. *Parallel Architectures for Artificial Neural Networks – Paradigms and Implementations*. IEEE Computer Society,, 1998.

[88] S. Suresh, S. N. Omkar, and V. Mani. Parallel implementation of back-propagation algorithm in networks of workstations. *IEEE Trans. Parallel Distrib. Syst.*, 16(1):24–34, 2005.

[89] David W. Tank and John J. Hopfield. Collective computation in neuronlike circuits. *Scientific American*, 257(6):104–114, December 1987.

[90] W. K. Taylor. Electrical simulation of some nervous system functional activities. In E. C. Cherry, editor, *Information Theory*, pages 314–328, London 1985, 1956. Butterworths, London.

[91] Jim Torresen, Hiroshi Nakashima, Shinji Tomita, and Olav Landsverk. Exploiting parallel computers to reduce neural network training time of real applications. In *International Symposium on High Performance Computing*, Fukuoka, Japan, 1997.

[92] Jim Torresen and Shinji Tomita. *Parallel Architectures for Artificial Neural Networks*, chapter Implementation of Backpropagation Training of Neural Networks On Large Parallel Computers. IEEE CS Press, 1998.

[93] L.A. Trejo and C. Sandoval. Improving back-propagation: Epsilon-back-propagation. In Mira J. and Sandoval F., editors, *From Natural to Artificial Neural Computation, Proc. of International Workshop on Artificial Neural Networks*, pages 427–432, Malaga-Torremolinos, Spain, 1995. Springer.

[94] Rein van den Boomgaard and Arnold W. M. Smeulders. Self learning image processing using a priori knowledge of spatial patterns. In Takeo Kanade, F. C. A. Groen, and L. O. Hertzberger, editors, *Intelligent Autonomous Systems 2, An International Conference, Amsterdam, The Netherlands, 11-14 December 1989*, pages 305–314. IOS Press, 1989.

[95] J. Wawrzynek, K. Asanovic, B. Kingsbury, J. Beck, D. Johnson, and N. Morgan. Spert-i1: A vector microprocessor system and its application to large problems in backpropagation training. In et al. Touretzky D.S., editor, *Advances in Neural Information Processing Systems 8 (NIPS'96)*, pages 619–625, Cambridge/Boston/London, 1996. MIT Press.

[96] P. J. Werbos. *Beyond Regression: New Tools for Predicition and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

[97] B. Widrow. Generalization and information storage in networks of adaline "neurons". In M. C. Yovits, G. T. Jacobi, and G. D. Goldstein, editors, *Self-Organizing Systems 1962*, pages 435–461, Chicago, Illinois, 1962. Spartan.

[98] B. Widrow and M. E. Hoff. Adaptive switching circuits. In *1960 IRE WESCON Convention Record*, pages 96–104, New York, 1960.

[99] S. Winograd and J. D. Cowan. *Reliable Computation in the Presence of Noise*. MIT Press, Cambridge, Massachusetts, 1963.

[100] M. Witbrock and M. Zagha. An implementation of backpropagation learning on GF11, a large SIMD parallel computer. *Parallel Computing*, 14(3):329–346, August 1990.

[101] Y.S. Wong, B.H.K. Lee, and T.K.S. Wong. Parameter extraction by parallel neural networks. *Intelligent Data Analysis*, pages 59–72, 2001.

[102] Kunihito Yamamori, Toru Abe, and Susumu Horiguchi. Theoretical learning-speed evaluation of parallel back-propagation algorithms. In *Systems Research and Information Systems*, volume 9, pages 121–148, 2000.

[103] H.H.T. Yeung. Online semantic extraction by backpropagation neural network with various syntactic structure representations. In McGuinness D.L. and Ferguson G., editors, *Proceedings of the 19th National Conference on Artificial Intelligence, 16th Conference on Innovative Applications of Artificial Intelligence (AAAI-04)*, pages 1042–1043. AAAI Press, Menlo Park, CA, 2004.

[104] Xiru Zhang, M. McKenna, J. P. Mesirov, and D. L. Waltz. The backpropagation algorithm on grid and hypercube architectures. *Parallel Computing*, 14(3):317–327, August 1990.

Lista lucrărilor publicate

[Ba94] Sorin Babii Experiments with a Distributed Ray Tracing Algorithm *Buletinul Ştiinţific al UTT, Seria Automatică şi Calculatoare*, 39(53):81–87, 1994.

[BaC03] Sorin Babii, Vladimir Creţu. Parallelizing the Training Phase of BackPropagation in a LAN of Workstations. *Concurrent Information Processing and Computing – NATO Advanced Research Workshop*, pages 29–36, IOS Press, ISBN 1-58603-502-9, ISSN 1387-6694, Sinaia, July 2003.

[BBD95] Sorin Babii, Laura Barbulescu, Romeo Dumitrescu. A Client–Server Application Model for Distributed Programming in the Netware Environment. *Buletinul Ştiinţific al UTT, Seria Automatică şi Calculatoare*, 40(54):67–77, 1995.

[BC03] Sorin Babii, Vladimir Creţu. A Parallel Algorithm for Neural Networks. *Buletinul Ştiinţific al Universităţii "Politehnica" din Timişoara, Seria Automatică şi Calculatoare*, 48(62):35–39, Timişoara 2003.

[BC04] Sorin Babii, Vladimir Creţu. A Distributed Algorithm for Neural Networks Training in a Network of Computers. The 6th International Conference on Technical Informatics, in *Buletinul Ştiinţific al Universităţii "Politehnica" din Timişoara, Seria Automatică şi Calculatoare*, 49(63),3:233–236, Timişoara 2004.

[BDB95] Sorin Babii, Romeo Dumitrescu, Laura Barbulescu. High Level Interface to the IPX Based Netware Communication Services. *Buletinul Ştiinţific al UTT, Seria Automatică şi Calculatoare*, 40(54):57–66, 1995.

[BS94] Sorin Babii, Nicolae Szirbik. Intermediate Representation Based on Conceptual Graph Model. *10th Conference on Control Systems and Computer Science – CSCS10*, 3:84–93, Bucureşti 1995.

[SB96] Nicolae Szirbik, Sorin Babii. The Basic Ideas for Developping Connectionist–Symbolic Cognitive Systems. *Buletinul Ştiinţific al UTT, Seria Automatică şi Calculatoare*, 41(55), 1996.

[SBD93] Nicolae Szirbik, Sorin Babii, Călin Dragomirescu. A Method for Inserting PROLOG Rules in a Neural Network. *Conferinţa internaţională de calculatoare*, Bucureşti, 2:202–206, mai 1993.

[Ba06] Sorin Babii. Reţele neuronale – Stadiu actual. Perspective. *Raport de cercetare ştiinţifică*, Universitatea "Politehnica" din Timişoara, Departamentul de Ingineria Calculatoarelor şi a Programării, 2006.

[Ba06a] Sorin Babii. DataParSim: Simulator paralel cu partiţionarea datelor. *Raport de cercetare ştiinţifică*, Universitatea "Politehnica" din Timişoara, Departamentul de Ingineria Calculatoarelor şi a Programării, 2006.

[Ba07] Sorin Babii. NetParSim: Simulator paralel cu partiţionarea reţelei. *Raport de cercetare ştiinţifică*, Universitatea "Politehnica" din Timişoara, Departamentul de Ingineria Calculatoarelor şi a Programării, 2007.

[Ba07a] Sorin Babii. Performance Evaluation for Training a Distributed Back-Propagation Implementation. *4th International Symposium on Applied Computational Intelligence and Informatics*, 273–276, ISBN: 1-4244-1234-X, Timişoara, 2007.

[BCP07] Sorin Babii, Vladimir Creţu, Emil Petriu. Performance Evaluation of Two Distributed BackPropagation Implementations. *The 2007 International Joint Conference on Neural Networks (IJCNN)*, Orlando, FL, 2007, ISBN: 1-4244-1380-X, ISSN: 1098-7576, IEEE Catalog Number: 07CH37922C.