Facultatea de Automatică şi Calculatoare
Programul de Licenţă: Calculatoare şi Tehnologia Informaţiei

Reţele neuronale de convoluţie cu valori complexe antrenate pe GPU

Proiect de diplomă

Andrei-Florin DUMA

Conducător ştiinţific:

As. dr. ing. Călin-Adrian POPA

Timişoara,

2016

Cuprins

Rezumat

Obiectivul acestui proiect este implementarea unei reţele neuronale de convoluţie cu valori complexe, folosind calculul paralel pe procesoare grafice, mai exact NVIDIA CUDA C. În ultimii ani, domeniul reţelelor neuronale cu valori complexe a cunoscut un interes crescând, mai ales datorită spectrului larg de aplicaţii, pornind de la procesarea semnalelor şi până la procesarea imaginilor. De asemenea, reţelele neuronale de convoluţie deţin cele mai bune rezultate la ora actuală în practic toate aplicaţiile specifice recunoaşterii de imagini. Proiectul are ca scop valorificarea puterii procesoarelor grafice (GPU-uri) pentru a crea o rețea de neuronală de convoluţie capabilă să proceseze date cu valori complexe. Aplicaţiile posibile ale acestui tip de reţea includ probleme de recunoaştere a imaginilor cum ar fi SAR (Synthetic Aperture Radar) şi fMRI (functional Magnetic Resonance Imaging), în care imaginile sunt nativ date de către instrumentele de măsurare în formă complexă, iar metoda actuală de a rezolva astfel de probleme este prin transformarea lor în domeniul real, pierzând astfel structura complexă a datelor. Experimente realizate pe cunoscuta bază de date MNIST au relevat o eroare de 0.90% a reţelelor de convoluţie cu valori complexe, faţă de o eroare de 1.22% a reţelelor de convoluţie cu valori reale.

Capitolul 1

Introducere

1.1 Contextul general

Reţelele neuronale cu valori complexe, pentru care intrările, ieşirile şi ponderile sunt numere complexe, au cunoscut un interes crescând în ultimii ani. Ele au fost introduse pentru prima dată în [1], unde valorile binare 0 şi 1 au fost extinse la mai multe valori situate pe cercul unitate din planul complex. Apoi, în [40] a fost propus algoritmul celor mai mici pătrate complex (complex least mean squares, CMLS). În anii 1991 şi 1992, au fost prezentate metoda gradient şi algoritmul backpropagation de învăţare, de către mai mulţi cercetători în mod independent.

În 1991, [27] a considerat o reţea neuronală cu valori complexe care are o funcţie de activare de tip sigmoid complexă complet $G(z) = \frac{1}{1+e^{-z}}$, şi a dezvoltat algoritmul de învăţare pentru o astfel de reţea folosind derivate parţiale ale părţilor reale şi imaginare ale funcţiilor care apar în cadrul metodei gradient. În 1992, [7] a considerat o reţea neuronală cu valori complexe a cărei funcţie de activare este de tip sigmoid complexă divizat $G(z) = \frac{1}{1+e^{-\mathrm{Re}z}} + i\frac{1}{1+e^{-\mathrm{Im}z}}$, $i = \sqrt{-1}$. Aceasta este una dintre cele mai simple extensii ale funcţiei de activare sigmoid din domeniul real. Avantajul ei este că putem de asemenea să obţinem algoritmul backpropagation ca o simplă extensie a aceluiaşi algoritm de la reţelele neuronale cu valori reale.

În acest context, articolul [13] discută ce clasă de funcţii de activare este cea mai potrivită, după ce face la început o analiză teoretică a funcţiilor de activare complexe divizat. Presupunând o implementare folosind circuite analogice, ei au propus o funcţie a cărei amplitudine se saturează, câtă vreme faza rămâne neschimbată, şi anume $G(z) = \frac{z}{c+|z|/r}$, unde c şi r sunt parametri reali care determină caracteristica saturării. Luând în considerare semnalele sau undele din circuitele electronice, o astfel de funcţie de activare reprezintă o alegere naturală. Tot în 1992, [19] a propus metoda gradient şi algoritmul backpropagation pentru reţele neuronale cu valori complexe care au funcţii de activare de tip amplitudine-fază, ca cea de mai sus, şi a demonstrat eficienţa lor. Ele au fost deduse folosind derivate parţiale în direcţiile amplitudinii şi fazei, care sunt compatibile cu acest tip de funcţie de activare.

După aceste începuturi, cercetarea în domeniu a luat un oarecare avânt. Se poate

demonstra că o reţea cu valori complexe nu este echivalentă cu o reţea cu valori reale de dimensiune dublă, datorită faptului că foloseşte înmulţirea în numere complexe. S-a arătat că aceste reţele au astfel o comportare diferită faţă de reţelele cu valori reale, deoarece, de exemplu, problema XOR şi problema detecţiei simetriei nu pot fi rezolvate folosind un neuron real, în timp ce ele pot fi rezolvate cu succes folosind un neuron complex [30]. Aplicaţii populare ale reţelelor cu valori complexe includ proiectarea antenelor, estimarea direcţiei de sosire şi de formare a undelor, imagistica radar, procesarea semnalelor în comunicaţii, procesarea imaginilor, şi multe altele (pentru o prezentare detaliată, a se vedea [20]).

Pe de altă parte, reţelele neuronale de convoluţie au devenit în zilele noastre cel mai de succes model pentru rezolvarea practic a tuturor problemelor legate de recunoaşterea de imagini. Apărute iniţial în varianta 1-dimensională, sub denumirea de reţele neuronale cu întârziere în timp, ele au fost folosite pentru recunoaşterea vorbirii [39]. În prezent, modele care au pornit de la această idee, reuşesc să obţină cele mai bune rezultate şi în domeniul recunoaşterii vorbirii [42].

Însă probabil cele mai importante şi cunoscute aplicaţii sunt cele ale reţelelor de convoluţie 2-dimensionale, proiectate să proceseze date în formă 2-dimensională, mai exact imagini reprezentate ca matrici de pixeli. Propuse pentru prima dată în [24] pentru recunoaşterea cifrelor scrise de mână, au fost mai apoi aplicate pentru recunoaşterea scrisului de mână în [25]. Deja în 1995, existau aplicaţii ale acestui tip de reţele în recunoaşterea imaginilor, a vorbirii şi a seriilor de timp [23]. Reţelele de convoluţie reprezintă o particularizare a reţelelor clasice multistrat, în care înmulţirea matricilor este înlocuită cu convoluţia şi matricea ponderilor cu nuclee de convoluţie care au o dimensiune mult mai mică decât cea a imaginilor primite la intrarea reţelei. Deşi deja aveau multe aplicaţii în domeniul viziunii artificiale, [26], reţelele de convoluţie au început să câştige în popularitate abia odată cu creşterea resurselor computaţionale disponibile şi cu implementarea lor folosind calculul paralel pe procesoare grafice (GPU).

Folosirea acestor resurse computaţionale a permis scăderea timpilor de antrenare şi de 100 de ori, ceea ce a dat naştere unor modele cu un număr din ce în ce mai mare de neuroni şi de parametri, inaugurând astfel domeniul numit învăţare adâncă (deep learning) [5, 6, 36, 14]. La baza acestui domeniu stau reţelele de convoluţie, pentru care creşterea numărului de straturi se traduce într-o creştere a performanţei, spre deosebire de reţelele mulistrat, a căror performanţă se degradează pentru un număr mare de straturi. Din acelaşi domeniu fac parte şi alte modele de reţele, pentru care s-a dovedit matematic că performanţa este direct proporţională cu mărimea modelului.

1.2 Motivaţia

Ideea introducerii reţelelor de convoluţie cu valori complexe a pornit de la faptul că unele imagini sunt date de instrumentele de măsură direct în formă complexă, iar orice imagine poate fi transformată în planul complex folosind transformata Fourier [22].

Radarele cu apertură sintetică (Synthetic Aperture Radars – SAR) sunt sisteme de

imagistică care produc imagini cu valori complexe ale solului [12, 34, 9]. Acestea pot fi de tip interferometric (Interferometric Synthetic Aperture Radars – InSAR) [37] sau polarimetric (Polarimetric Synthetic Aperture Radars – PolSAR) [2, 15, 16]. Modele de reţele neuronale cu valori complexe au fost aplicate pentru reducerea zgomotului, compresia şi recunoaşterea acestor tipuri de imagini complexe. Până în prezent, există doar două încercări de recunoaştere a acestui tip de imagini direct în domeniul complex, una folosind reţele neuronale multistrat cu valori complexe, şi una folosind o reţea de convoluţie cu un singur strat de convoluţie, dovedindu-se că ambele modele sunt mai potrivite acestei probleme decât reţelele neuronale cu valori reale, care ignoră dependenţele între date existente în planul complex [15, 16]. Acest fapt a dus la ideea că reţelele neuronale de convoluţie cu valori complexe adânci, cu multe straturi, implementate pe procesoare grafice, ar putea aduce un plus şi mai mare de performanţă în acest domeniu.

Un alt tip de radare care produc imagini pretabile a fi procesate în domeniul complex sunt radarele cu penetrare a solului (Ground Penetrating Radars – GPR). S-a dovedit că modelele de reţele cu valori complexe obţin rezultate superioare altor metode în detecţia minelor antipersonal folosind imagini date de acest tip de radare [17, 29, 41]. Însă, până în prezent, aceste modele au fost de tip superficial (shallow), această problemă nefiind abordată folosind modelele adânci disponibile în cadrul paradigmei de învăţare adâncă. Aceasta constituie o nouă motivaţie pentru introducerea reţelelor de convoluţie cu valori complexe.

Imagistica folosind rezonanţă magnetică funcţională (functional Magnetic Resonance Imaging – fMRI) colectează datele tot sub formă complexă, însă aproape toate studiile pe acest tip de imagistică folosesc doar amplitudinea datelor în analiză, ignorând informaţiile de frecvenţă [21]. Reţele neuronale cu valori complexe s-au dovedit a avea performanţe superioare în reconstrucţia şi analiza unor astfel de imagini [21, 18]. Metode statistice de tip analiza componentelor independente în domeniul complex au fost de asemenea aplicate cu succes pentru aceste imagini [33, 28, 43]. Observaţiile anterioare deschid posibilitatea aplicaţiei reţelelor de convoluţie cu valori complexe pentru recunoaşterea diverselor tipare care pot apărea în aceste imagini, luând în considerare toate informaţiile date de dispozitivele de imagistică, şi nu ignorând informaţiile de frecvenţă ca până în prezent.

Bazele domeniului învăţării adânci cu valori complexe au fost puse abia în ultimii ani. Deşi au existat aplicaţii ale reţelelor neuronale clasice în domeniul recunoaşterii de imagini [31], şi o reţea de convoluţie simplă, cu un singur strat, a fost propusă în [16] pentru recunoaşterea imaginilor date de radarele cu apertură sintetică polarimetrice (PolSAR), doar în ultimul timp au început să apară lucrări în care se discută algoritmi specifici învăţării adânci folosind numere complexe. Astfel, în [4] sunt propuse autoencoder-ele cu valori complexe, care sunt un tip de model care face parte din această paradigmă. În [8] se propune o reţea de tip wavelet scattering, care foloseşte numere complexe, şi explică propertăţi importante ale reţelelor neuronale de convoluţie. Sincronicitatea neuronilor într-o reţea de tip Deep Boltzmann Machine (DBM) a fost discutată în [32], arătând existenţa unor perfomanţe superioare faţă de cazul real.

În fine, articole foarte recente discută modele recurente de reţele neuronale cu valori complexe [3], precum şi capacitatea învăţării de reprezentări ale seriilor de timp folosind reţele recurente cu valori complexe [35], ambele raportând rezultate asemănătoare sau superioare celor cu valori reale şi existenţa unor proprietăţi deosebite ale acestor reţele faţă de cele cu valori reale, ceea ce le face pretabile unor anumite tipuri de aplicaţii. Unul dintre cele mai importante articole în acest domeniu este [38], care aduce o motivare matematică a reţelelor de convoluţie cu valori complexe, arătând că ele pot fi văzute ca pachete multiwavelet nonliniare (nonlinear multiwavelet packets), făcând astfel disponibilă analiza matematică din domeniul procesării de semnale pentru formularea riguroasă a proprietăţilor reţelelor neuronale de convoluţie cu valori complexe. Urmare a acestui ultim articol, este de aşteptat că domeniul învăţării adânci cu valori complexe va lua amploare în următorii ani, tendinţă în care se înscrie şi prezentul proiect.

1.3 Descrierea proiectului

Soluția propusă constă într-o aplicație software dezvoltată după modelul unei rețele neuronale de convoluție care, pornind de la un set de date de intrare extrase dintr-o imagine și stocate într-un fișier extern, va face posibilă antrenarea rețelei prin actualizarea parametrilor interni ai acesteia. Datele de intrare împreună cu rezultatele corespunzătoare acestora, care sunt cunoscute inițial, formează setul de antrenare al rețelei neuronale.

În timpul prelucrării, toate informațiile vor fi stocate în structuri de date care permit interpretarea lor ca fiind valori complexe. Sistemul software va permite configurarea parametrilor de funcționare (învățare) a rețelei pentru a se obține astfel rezultate optime și costuri minime care sunt caracterizate prin timpul de rulare al programului și memoria de stocare ocupată în timpul rulării.

Determinarea parametrilor de configurare optimi în funcție de contextul în care se aplică soluția este un pas important și se va face prin metode experimentale înainte de aplicarea propriu-zisă. Prin urmare se facilitează modificarea cu ușurință a tuturor valorilor configurabile și delimitarea acestora de implementarea în sine a rețelei pentru a face posibilă interpretarea acestora si a impactului pe care-l produce modificarea valorilor parametrilor configurabili.

Pentru interpretarea datelor și pentru aplicarea la acestea a parametrilor interni se utilizează module de calcul și operații matematice complexe. Executarea acestor operații pe o componentă GPU va spori atât rapiditatea cu care se efectuează cât și acuratețea rezultatelor, fiind posibilă împărțirea sarcinii între mai multe fire de execuție. Astfel se utilizează capacitatea de paralelism a procesoarelor grafice, caracteristică specializată a acestor unități de procesare.

Capitolul 2

Fundamentare teoretică

2.1 Reţele neuronale multistrat cu valori complexe

Mulţimea numerelor complexe este definită ca fiind

$$\mathbb{C} := \{x + iy | x, y \in \mathbb{R}, i = \sqrt{-1}\}.$$

Dacă $z = x + iy \in \mathbb{C}$, x se numeşte *partea reală* a numărului complex z, iar y se numeşte *partea imaginară* a lui z. Vom nota cu z^R partea reală a lui z şi cu z^I partea imaginară a lui z, deci vom putea scrie

$$z = z^R + iz^I, \forall z \in \mathbb{C}.$$

Pentru orice $z_1 = x_1 + iy_1$, $z_2 = x_2 + iy_2 \in \mathbb{C}$, definim *adunarea* prin

$$z_1 + z_2 := (x_1 + x_2) + i(y_1 + y_2),$$

înmulţirea prin

$$z_1 \cdot z_2 := (x_1 x_2 - y_1 y_2) + i(x_1 y_2 + x_2 y_1),$$

inversul lui z_1 prin

$$z_1^{-1} := \frac{x_1}{x_1^2 + y_1^2} - i\frac{y_1}{x_1^2 + y_1^2},$$

conjugatul lui z_1 prin

$$\overline{z_1} := x_1 - iy_1,$$

şi *modulul* lui z_1 prin

$$|z_1| := \sqrt{z_1 \overline{z_1}} = \sqrt{x_1^2 + y_1^2}.$$

Acum putem face o scurtă introducere în problematica funcţiilor definite pe mulţimea numerelor complexe. Fie $f : \mathbb{C} \to \mathbb{C}$ o funcţie complexă cu valori complexe. Pornind de la identificarea $\mathbb{C} \simeq \mathbb{R}^2$, funcţia f se poate scrie sub forma

$$f(x + iy) = u(x, y) + iv(x, y),$$

unde $u, v : \mathbb{R}^2 \to \mathbb{R}$ sunt funcţii de două variabile reale cu valori reale.

Limita funcţiei f în punctul z_0 este acel număr complex L care satisface: pentru orice $\varepsilon > 0$, există $\delta > 0$, astfel încât, pentru orice $z \in \mathbb{C}$ care satisface $|z - z_0| < \delta$, avem că $|f(z) - L| < \varepsilon$. Notăm

$$\lim_{z \to z_o} f(z) = L.$$

Se poate observa imediat că

$$\lim_{z \to z_o} f(z) = \lim_{(x,y) \to (x_0,y_0)} u(x,y) + i \lim_{(x,y) \to (x_0,y_0)} v(x,y),$$

ceea ce ne arată că f este continuă în z_0 (şi, mai general, pe $\mathbb{C}$) dacă şi numai dacă u şi v sunt continue în (x_0, y_0) (sau, mai general, pe $\mathbb{R}^2$).

Putem acum defini *derivata complexă* a lui f în punctul z_0 ca fiind

$$f'(z_0) = \lim_{z \to z_o} \frac{f(z) - f(z_0)}{z - z_0}.$$

Se pot defini *derivatele parţiale* ale lui f în punctul z_0 prin

$$\frac{\partial f}{\partial x}(z_0) := \lim_{h \to 0} \frac{f(z_0 + h) - f(z_0)}{h},$$

$$\frac{\partial f}{\partial y}(z_0) := \lim_{h \to 0} \frac{f(z_0 + ih) - f(z_0)}{h}.$$

Observăm că dacă există $f'(z_0)$, atunci trebuie să avem

$$f'(z_0) = \frac{\partial f}{\partial x}(z_0) = -i \frac{\partial f}{\partial y}(z_0). \tag{2.1}$$

Pe de altă parte, putem scrie că

$$\begin{aligned} \frac{\partial f}{\partial x}(z_0) &= \lim_{h \to 0} \frac{u(x_0 + h, y_0) - u(x_0, y_0)}{h} + i \lim_{h \to 0} \frac{v(x_0 + h, y_0) - v(x_0, y_0)}{h} \\ &= \frac{\partial u}{\partial x}(x_0, y_0) + i \frac{\partial v}{\partial x}(x_0, y_0), \end{aligned}$$

şi analog

$$\begin{aligned} \frac{\partial f}{\partial y}(z_0) &= \lim_{h \to 0} \frac{u(x_0, y_0 + h) - u(x_0, y_0)}{h} + i \lim_{h \to 0} \frac{v(x_0, y_0) - v(x_0, y_0 + h)}{h} \\ &= \frac{\partial u}{\partial y}(x_0, y_0) + i \frac{\partial v}{\partial y}(x_0, y_0). \end{aligned}$$

Din relaţia (2.1), rezultă că, pentru ca $f'(z_0)$ să existe, trebuie ca

$$\frac{\partial u}{\partial x}(x_0, y_0) + i \frac{\partial v}{\partial x}(x_0, y_0) = -i \frac{\partial u}{\partial y}(x_0, y_0) + \frac{\partial v}{\partial y}(x_0, y_0),$$

adică

$$\begin{cases} \frac{\partial u}{\partial x}(x_0, y_0) = & \frac{\partial v}{\partial y}(x_0, y_0) \\ \frac{\partial v}{\partial x}(x_0, y_0) = & -\frac{\partial u}{\partial y}(x_0, y_0) \end{cases},$$

care poartă denumirea de *ecuaţiile Cauchy-Riemann.*

Dacă funcţia f este derivabilă complex în fiecare punct z_0 al unei mulţimi deschise $U \subset \mathbb{C}$, se spune că f este *olomorfă* pe U. Spunem că f este olomorfă în punctul z_0 dacă este olomorfă pe o vecinătate a lui z_0. O funcţie olomorfă a cărui domeniu este întreg planul complex se numeşte *funcţie întreagă*.

Acum, Teorema lui Liouville ne spune că orice funcţie olomorfă pe $\mathbb{C}$ (întreagă) şi mărginită este constantă. Acest fapt va avea consecinţe importante în domeniul reţelelor neuronale, deoarece ideal am vrea ca funcţiile de activare să fie atât mărginite cât şi derivabile complex pe toată mulţimea numerelor complexe. Cum acest deziderat nu poate fi atins decât de funcţiile constante, care nu sunt acceptabile ca funcţii de activare, un compromis este necesar. Unii autori aleg funcţii de activare mărginite, dar care nu sunt derivabile complex, iar alţii aleg funcţii de activare derivabile complex, care însă au puncte în care sunt nemărginite.

O altă problemă ridicată de analiza complexă este aceea că funcţiile de eroare folosite în reţelele neuronale au valori reale. După cum se poate observa uşor, singurele funcţii $f : \mathbb{C} \to \mathbb{R}$ derivabile complex sunt constantele. Prin urmare, pentru deducerea algoritmilor de învăţare pentru reţelele neuronale cu valori complexe, nu vom putea utiliza derivata complexă a funcţiei de eroare, deoarece aceasta nu există. Singurele derivate care există pentru o astfel de funcţie sunt derivatele parţiale $\frac{\partial f}{\partial x}$ şi $\frac{\partial f}{\partial y}$, şi respectiv derivatele parţiale ale componentelor u şi v ale funcţiei f, adică $\frac{\partial u}{\partial x}$, $\frac{\partial u}{\partial y}$, $\frac{\partial v}{\partial x}$, $\frac{\partial v}{\partial y}$.

Să presupunem că avem o reţea neuronală cu valori complexe de tip multistrat, total conectată, care are L straturi, unde stratul 1 este stratul de intrare, stratul L este stratul de ieşire, iar straturile notate cu $\{2, \ldots, L-1\}$ sunt straturi ascunse. Funcţia de eroare $E : \mathbb{C}^N \to \mathbb{R}$ pentru o asemenea reţea este definită prin

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{c} (y^L[i] - t[i])(\overline{y^L[i] - t[i]}), \tag{2.2}$$

unde $\mathbf{y}^L = (y^L[i])_{1 \le i \le c}$ reprezintă *ieşirile* reţelei, $\mathbf{t} = (t[i])_{1 \le i \le c}$ reprezintă *ieşirile dorite* (*target*-urile) ale reţelei, iar $\mathbf{w}$ este vectorul tuturor celor N *ponderi* şi *bias*-uri ale reţelei.

Dacă notăm cu $w^l[p, r]$ ponderea care leagă neuronul p din stratul $l-1$ de neuronul r din stratul l, pentru orice $l \in \{2, \ldots, L\}$, putem defini *pasul de actualizare* al ponderii $w^l[p, r]$ în epoca t ca fiind

$$\Delta w^l[p, r](t) = w^l[p, r](t+1) - w^l[p, r](t).$$

Cu această notaţie, pentru *metoda gradient*, *regula de actualizare* pentru ponderea $w^l[p, r]$ este

$$\Delta w^l[p, r](t) = -\varepsilon \left(\frac{\partial E}{\partial w^{l,R}[p, r]}(t) + i \frac{\partial E}{\partial w^{l,I}[p, r]}(t) \right),$$

unde ε este un număr real care reprezintă *rata de învăţare*, şi am notat cu a^R partea reală a numărului complex a, şi cu a^I partea imaginară a numărului complex a, i.e. $a = a^R + ia^I$, $i = \sqrt{-1}$.

Prin urmare, pentru a minimiza funcţia de eroare E, avem nevoie să calculăm derivatele parţiale de forma $\frac{\partial E}{\partial w^{l,R}[p,r]}$, $\frac{\partial E}{\partial w^{l,I}[p,r]}$.

Pentru aceasta, facem următoarele notaţii

$$s^l[p] := \sum_r w^l[p,r]x^{l-1}[r], \tag{2.3}$$

$$y^l[p] := G^l(s^l[p]), \tag{2.4}$$

unde G^l este *funcţia de activare* a stratului $l \in \{2,\ldots,L\}$, $\mathbf{x}^1 = (x^1[r])_{1\le r\le d}$ sunt intrările reţelei, şi avem că $x^l[r] = y^l[r]$, $\forall l \in \{2,\ldots,L-1\}$, $\forall r$. Scriind relaţiile de mai sus sub formă vectorială, avem

$$\mathbf{s}^l := \mathbf{W}^l\mathbf{x}^{l-1}, \tag{2.5}$$

$$\mathbf{y}^l := G^l(\mathbf{s}^l). \tag{2.6}$$

Vom calcula mai întâi actualizările pentru ponderile dintre penultimul strat ascuns $L-1$ şi stratul de ieşire L, i.e.

$$\Delta w^L[p,r] = -\varepsilon\left(\frac{\partial E}{\partial w^{L,R}[p,r]} + i\frac{\partial E}{\partial w^{L,I}[p,r]}\right).$$

Folosind regula înlănţuirii, putem scrie următoarele relaţii:

$$\frac{\partial E}{\partial w^{L,R}[p,r]} = \frac{\partial E}{\partial s^{L,R}[p]}\frac{\partial s^{L,R}[p]}{\partial w^{L,R}[p,r]} + \frac{\partial E}{\partial s^{L,I}[p]}\frac{\partial s^{L,I}[p]}{\partial w^{L,R}[p,r]},$$

$$\frac{\partial E}{\partial w^{L,I}[p,r]} = \frac{\partial E}{\partial s^{L,R}[p]}\frac{\partial s^{L,R}[p]}{\partial w^{L,I}[p,r]} + \frac{\partial E}{\partial s^{L,I}[p]}\frac{\partial s^{L,I}[p]}{\partial w^{L,I}[p,r]}.$$

Aplicând din nou regula înlănţuirii pentru $\partial E/\partial s^{L,R}[p]$ şi $\partial E/\partial s^{L,I}[p]$, avem că

$$\begin{aligned}\frac{\partial E}{\partial s_j^{L,R}[p]} &= \frac{\partial E}{\partial y^{L,R}[p]}\frac{\partial y^{L,R}[p]}{\partial s^{L,R}[p]} + \frac{\partial E}{\partial y^{L,I}[p]}\frac{\partial y^{L,I}[p]}{\partial s^{L,R}[p]}\\ &= (y^{L,R}[p]-t^R[p])\frac{\partial G^{L,R}(s^L[p])}{\partial s^{L,R}[p]} + (y^{L,I}[p]-t^I[p])\frac{\partial G^{L,I}(s^L[p])}{\partial s^{L,R}[p]},\end{aligned}$$

$$\begin{aligned}\frac{\partial E}{\partial s^{L,I}[p]} &= \frac{\partial E}{\partial y^{L,R}[p]}\frac{\partial y^{L,R}[p]}{\partial s^{L,I}[p]} + \frac{\partial E}{\partial y^{L,I}[p]}\frac{\partial y^{L,I}[p]}{\partial s^{L,I}[p]}\\ &= (y^{L,R}[p]-t^R[p])\frac{\partial G^{L,R}(s^L[p])}{\partial s^{L,I}[p]} + (y^{L,I}[p]-t^I[p])\frac{\partial G^{L,I}(s^L[p])}{\partial s^{L,I}[p]},\end{aligned}$$

unde am ţinut seama de notaţia (2.4) şi de expresia pentru funcţia de eroare (2.2). Din (2.3), deducem că

$$\frac{\partial s^{L,R}[p]}{\partial w^{L,R}[p,r]} = x^{L-1,R}[r],\ \frac{\partial s^{L,R}[p]}{\partial w^{L,I}[p,r]} = -x^{L-1,I}[r],$$

$$\frac{\partial s^{L,I}[p]}{\partial w^{L,R}[p,r]} = x^{L-1,I}[r],\ \frac{\partial s^{L,I}[p]}{\partial w^{L,I}[p,r]} = x^{L-1,R}[r],$$

şi folosind notaţia $\delta^L[p] := \partial E/\partial s^L[p]$, obţinem expresia pentru actualizarea dorită:

$$\Delta w^L[p,r] = -\varepsilon \delta^L[p] \overline{x^{L-1}[r]}.$$

Acum, vom calcula actualizarea pentru o pondere arbitrară $w^l[p,r]$, unde $l \in \{2,\ldots,L-1\}$. În primul rând, putem scrie că

$$\Delta w^l[p,r] = -\varepsilon \left(\frac{\partial E}{\partial w^{l,R}[p,r]} + i \frac{\partial E}{\partial w^{l,I}[p,r]} \right),$$

iar apoi, din regula înlănţuirii, avem că

$$\frac{\partial E}{\partial w^{l,R}[p,r]} = \frac{\partial E}{\partial s^{l,R}[p]} \frac{\partial s^{l,R}[p]}{\partial w^{l,R}[p,r]} + \frac{\partial E}{\partial s^{l,I}[p]} \frac{\partial s^{l,I}[p]}{\partial w^{l,R}[p,r]},$$
$$\frac{\partial E}{\partial w^{l,I}[p,r]} = \frac{\partial E}{\partial s^{l,R}[p]} \frac{\partial s_j^{l,R}}{\partial w^{L,I}[p,r]} + \frac{\partial E}{\partial s^{l,I}[p]} \frac{\partial s^{l,I}[p]}{\partial w^{l,I}[p,r]}.$$

Aplicând din nou regula înlănţuirii, obţinem că

$$\frac{\partial E}{\partial s^{l,R}[p]} = \sum_m \frac{\partial E}{\partial s^{l+1,R}[m]} \frac{\partial s^{l+1,R}[m]}{\partial s^{l,R}[p]} + \frac{\partial E}{\partial s^{l+1,I}[m]} \frac{\partial s^{l+1,I}[m]}{\partial s^{l,R}[p]}, \tag{2.7}$$

$$\frac{\partial E}{\partial s^{l,I}[p]} = \sum_m \frac{\partial E}{\partial s^{l+1,R}[m]} \frac{\partial s^{l+1,R}[m]}{\partial s^{l,I}[p]} + \frac{\partial E}{\partial s^{l+1,I}[m]} \frac{\partial s^{l+1,I}[m]}{\partial s^{l,I}[p]}, \tag{2.8}$$

unde sumele sunt luate după toţi neuronii m din stratul $l+1$ către care neuronul p din stratul l trimite conexiuni. Apoi

$$\frac{\partial s^{l+1,R}[m]}{\partial s^{l,R}[p]} = \frac{\partial s^{l+1,R}[m]}{\partial y^{l,R}[p]} \frac{\partial y^{l,R}[p]}{\partial s^{l,R}[p]} + \frac{\partial s^{l+1,R}[m]}{\partial y^{l,I}[p]} \frac{\partial y^{l,I}[p]}{\partial s^{l,R}[p]},$$

şi relaţiile analoage. Din nou din (2.3), avem

$$\frac{\partial s^{l+1,R}[m]}{\partial y^{l,R}[p]} = w^{l+1,R}[m,j], \ \frac{\partial s^{l+1,R}[m]}{\partial y^{l,I}[p]} = -w^{l+1,I}[m,j],$$
$$\frac{\partial s^{l+1,I}[m]}{\partial y^{l,R}[p]} = w^{l+1,I}[m,j], \ \frac{\partial s^{l+1,I}[m]}{\partial y^{l,I}[p]} = w^{l+1,R}[m,j].$$

Acum, punând toate cele de mai sus împreună, relaţiile (2.7) şi (2.8) devin

$$\frac{\partial E}{\partial s^{l,R}[p]} = \left(\sum_m \overline{w^{l+1}[m,j]} \delta^{l+1}[m] \right)^R \frac{\partial G^{l,R}(s^l[p])}{\partial s^{l,R}[p]} + \left(\sum_m \overline{w^{l+1}[m,j]} \delta^{l+1}[m] \right)^I \frac{\partial G^{l,I}(s^l[p])}{\partial s^{l,R}[p]},$$

$$\frac{\partial E}{\partial s^{l,I}[p]} = \left(\sum_m \overline{w^{l+1}[m,j]}\delta^{l+1}[m]\right)^R \frac{\partial G^{l,R}(s^l[p])}{\partial s^{l,I}[p]} + \left(\sum_m \overline{w^{l+1}[m,j]}\delta^{l+1}[m]\right)^I \frac{\partial G^{l,I}(s^l[p])}{\partial s^{l,I}[p]}.$$

În fine, dacă notăm $\delta^l[p] := \partial E/\partial s^l[p]$, şi folosim (2.3) pentru a calcula

$$\frac{\partial s^{l,R}[p]}{\partial w^{l,R}[p,r]} = x^{l-1,R}[r],\ \frac{\partial s^{l,R}[p]}{\partial w^{l,I}[p,r]} = -x^{l-1,I}[r],$$

$$\frac{\partial s^{l,I}[p]}{\partial w^{l,R}[p,r]} = x^{l-1,I}[r],\ \frac{\partial s^{l,I}[p]}{\partial w^{l,I}[p,r]} = x^{l-1,R}[r],$$

obţinem

$$\Delta w^l[p,r] = -\varepsilon\delta^l[p]\overline{x^{l-1}[r]},\ \forall l \in \{2,\ldots,L-1\}.$$

În concluzie, avem că

$$\Delta w^l[p,r] = -\varepsilon\delta^l[p]\overline{x^{l-1}[r]},\ \forall l \in \{2,\ldots,L\},$$

unde

$$\delta_j^l = \begin{cases} \left(\sum_m \overline{w^{l+1}[m,j]}\delta^{l+1}[m]\right)^R \left(\frac{\partial G^{l,R}(s^l[p])}{\partial s^{l,R}[p]} + i\frac{\partial G^{l,R}(s^l[p])}{\partial s^{l,I}[p]}\right) \\ + \left(\sum_m \overline{w^{l+1}[m,j]}\delta^{l+1}[m]\right)^I \left(\frac{\partial G^{l,I}(s^l[p])}{\partial s^{l,R}[p]} + i\frac{\partial G^{l,I}(s^l[p])}{\partial s^{l,I}[p]}\right), & l \le L-1 \\ (y^{l,R}[p] - t^R[p])\left(\frac{\partial G^{l,R}(s^l[p])}{\partial s^{l,R}[p]} + i\frac{\partial G^{l,R}(s^l[p])}{\partial s^{l,I}[p]}\right) \\ +(y^{l,I}[p] - t^I[p])\left(\frac{\partial G^{l,I}(s^l[p])}{\partial s^{l,R}[p]} + i\frac{\partial G^{l,I}(s^l[p])}{\partial s^{l,I}[p]}\right), & l = L \end{cases}.$$

Observăm că aceste relaţii nu depind de tipul funcţiei de activare G^l: aceasta poate să fie *complexă complet* (tratează numărul complex ca un întreg) sau *complexă divizat* (tratează separat părţile reală şi imaginară ale numărului complex).

Spre exemplu, dacă $G^l(z) = \tanh z$, adică G^l este complexă complet, avem că

$$\begin{aligned} \frac{\partial G^{l,R}(z)}{\partial z^R} + i\frac{\partial G^{l,R}(z)}{\partial z^I} &= \frac{\partial(\tanh z)^R}{\partial z^R} + i\frac{\partial(\tanh z)^R}{\partial z^I} \\ &= \frac{\partial}{\partial z^R}\left(\frac{\sinh 2z^R}{\cosh 2z^R + \cos 2z^I} + i\frac{\sin 2z^I}{\cosh 2z^R + \cos 2z^I}\right)^R \\ &\quad + i\frac{\partial}{\partial z^I}\left(\frac{\sinh 2z^R}{\cosh 2z^R + \cos 2z^I} + i\frac{\sin 2z^I}{\cosh 2z^R + \cos 2z^I}\right)^R \\ &= \frac{\partial}{\partial z^R}\left(\frac{\sinh 2z^R}{\cosh 2z^R + \cos 2z^I}\right) + i\frac{\partial}{\partial z^I}\left(\frac{\sinh 2z^R}{\cosh 2z^R + \cos 2z^I}\right) \\ &= \frac{2(1 + \cosh 2z^R \cos 2z^I)}{(\cosh 2z^R + \cos 2z^I)^2} + i\frac{2\sinh 2z^R \sin 2z^I}{(\cosh 2z^R + \cos 2z^I)^2}, \end{aligned}$$

şi analog pentru $\frac{\partial G^{l,I}(z)}{\partial z^R} + i\frac{\partial G^{l,I}(z)}{\partial z^I}$. Dacă, în plus, G^l este derivabilă complex, atunci au loc reţiile Cauchy-Riemann

$$\begin{cases} \frac{\partial G^{l,R}(z)}{\partial z^R} = & \frac{\partial G^{l,I}(z)}{\partial z^I} \\ \frac{\partial G^{l,I}(z)}{\partial z^R} = & -\frac{\partial G^{l,R}(z)}{\partial z^I} \end{cases},$$

şi obţinem că

$$\frac{\partial G^{l,R}(z)}{\partial z^R} + i\frac{\partial G^{l,R}(z)}{\partial z^I} = -i\left(\frac{\partial G^{l,I}(z)}{\partial z^R} + i\frac{\partial G^{l,I}(z)}{\partial z^I}\right) = \frac{\partial G^{l,R}(z)}{\partial z^R} - i\frac{\partial G^{l,I}(z)}{\partial z^R} = \overline{(G^l)'(z)}.$$

Dacă G^l este complexă divizat, de exemplu $G^l(z) = G^l(z^R + iz^I) = \tanh z^R + i\tanh z^I$, atunci

$$\begin{aligned} \frac{\partial G^{l,R}(z)}{\partial z^R} + i\frac{\partial G^{l,R}(z)}{\partial z^I} &= \frac{\partial(\tanh z^R + i\tanh z^I)^R}{\partial z^R} + i\frac{\partial(\tanh z^R + i\tanh z^I)^R}{\partial z^I} \\ &= \frac{\partial(\tanh z^R)}{\partial z^R} + i\frac{\partial(0)}{\partial z^I} \\ &= 1 - \tanh^2 z^R, \end{aligned}$$

şi analog pentru $\frac{\partial G^{l,I}(z)}{\partial z^R} + i\frac{\partial G^{l,I}(z)}{\partial z^I}$.

Dacă presupunem că funcţiile G^l, $l \in \{2, \ldots, L\}$ sunt derivabile complex, relaţiile de mai sus se scriu sub formă matricială astfel:

$$\Delta \mathbf{W}^l = -\varepsilon \boldsymbol{\delta}^l (\mathbf{x}^{l-1})^H, \ \forall l \in \{2, \ldots, L\}, \tag{2.9}$$

$$\boldsymbol{\delta}^l = \begin{cases} \left((\mathbf{W}^{l+1})^H \boldsymbol{\delta}^{l+1}\right) \circ \overline{(G^l)'(\mathbf{s}^l)}, & l \leq L-1 \\ (\mathbf{y}^l - \mathbf{t}) \circ \overline{(G^l)'(\mathbf{s}^l)}, & l = L \end{cases}, \tag{2.10}$$

unde $\mathbf{A}^H$ reprezintă transpusa conjugată a lui $\mathbf{A}$ şi $\mathbf{A} \circ \mathbf{B}$ reprezintă înmulţirea pe componente.

În cazul în care funcţiile G^l, $l \in \{2, \ldots, L\}$ sunt complexe divizat, relaţiile pentru calculul valorilor $\boldsymbol{\delta}^l$ au următoarea formă matricială

$$\Delta \mathbf{W}^l = -\varepsilon \boldsymbol{\delta}^l (\mathbf{x}^{l-1})^H, \ \forall l \in \{2, \ldots, L\},$$

$$\boldsymbol{\delta}^l = \begin{cases} \left((\mathbf{W}^{l+1})^H \boldsymbol{\delta}^{l+1}\right) \star \frac{\partial G^l(\mathbf{s}^l)}{\partial \mathbf{s}^l}, & l \leq L-1 \\ (\mathbf{y}^l - \mathbf{t}) \star \frac{\partial G^l(\mathbf{s}^l)}{\partial \mathbf{s}^l}, & l = L \end{cases}, \tag{2.11}$$

unde $\mathbf{A} \star \mathbf{B} = \mathbf{A}^R \circ \mathbf{B}^R + i\mathbf{A}^I \circ \mathbf{B}^I$, iar actualizările ponderilor sunt date prin aceeaşi formulă (2.9).

2.2 Reţele neuronale de convoluţie cu valori complexe

Reţelele neuronale de convoluţie reprezintă o variaţie a reţelelor neuronale multistrat, fiind folosite pentru procesarea datelor care au forma unei „grile". Ne vom opri în cele ce

urmează asupra grilelor 2D, care corespund pixelilor dintr-o imagine. Astfel, înmulţirea matricilor din (2.5) este înlocuită în acest tip de reţele cu operaţia de convoluţie 2D:

$$s[i,j] = (x * k)[i,j] = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} x[m,n]k[i-m, j-n],$$

unde facem convenţia ca acei termeni ai căror indici depăşesc dimensiunea grilei să fie consideraţi egali cu 0, ceea ce înseamnă că această relaţie presupune o sumă finită de termeni. De exemplu, avem

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} * \begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix}$$

$$= \begin{bmatrix} x_{11}k_{22} + x_{12}k_{21} + x_{21}k_{12} + x_{22}k_{11} & x_{12}k_{22} + x_{13}k_{21} + x_{22}k_{12} + x_{23}k_{11} \\ x_{21}k_{22} + x_{22}k_{21} + x_{31}k_{12} + x_{32}k_{11} & x_{22}k_{22} + x_{23}k_{21} + x_{32}k_{12} + x_{33}k_{11} \end{bmatrix}.$$

Convoluţia poate fi privită ca înmulţire cu o matrice, care are mai multe elemente egale între ele. În plus convoluţia prespune ca această matrice să aibă multe elemente egale cu 0, mai exact să fie o matrice rară. Aceasta se datorează faptului că nucleele de convoluţie, care corespund ponderilor din reţelele multistrat, au în general o dimensiune mult mai mică decât cea a imaginilor de intrare. Prin urmare, orice algoritm de antrenare care nu depinde de forma matricii ponderilor se aplică şi reţelelor de convoluţie, fără a fi necesare niciun fel de alte modificări.

O reţea neuronală de convoluţie are, în general, două tipuri de straturi: straturi de *convoluţie* şi straturi de *pooling*.

Straturile de convoluţie sunt foarte asemănătoare cu straturile dintr-o reţea multistrat, cu deosebirea menţionată anterior a înlocuirii operaţiei de înmulţire cu matricea ponderilor cu o operaţie de convoluţie cu mai multe nuclee de convoluţie. Reprezintă o particularitate a reţelelor de convoluţie, existenţa mai multor canale pe fiecare strat, care pot să corespundă, spre exemplu, canalelor RGB dintr-o imagine.

Să presupunem că avem un strat de convoluţie l care are I canale de intrare notate $\mathbf{x}^{l-1}[i]$, $i \in \{1, \ldots, I\}$. Evident, pentru primul strat, avem că $I = 1$ pentru imagini alb-negru şi $I = 3$ pentru imagini color. Dacă stratul are J canale de ieşire, notate $\mathbf{y}^l[j]$, $j \in \{1, \ldots, J\}$, atunci relaţiile (2.5)-(2.6) devin:

$$\mathbf{s}^l[j] = \sum_{i=1}^{I} \mathbf{x}^{l-1}[i] * \mathbf{k}^l[i,j], \ \forall j \in \{1, \ldots, J\},$$

$$\mathbf{y}^l[j] = G^l(\mathbf{s}^l[j]), \ \forall j \in \{1, \ldots, J\}.$$

cu observaţia făcută şi în cazul reţelelor multistrat că $\mathbf{x}^l[j] = \mathbf{y}^l[j]$, $\forall j \in \{1, \ldots, J\}$. $\mathbf{k}^l[i,j]$ reprezintă nucleele de convoluţie, care sunt în număr de IJ, adică produsul dintre numărul de canale de intrare şi numărul de canale de ieşire. Acestea înlocuiesc ponderile $\mathbf{W}^l$ din reţelele multistrat. Dimensiunea lor este, de obicei, mult mai mică decât cea a canalelor de intrare, dând două proprietăţi importante ale reţelelor de convoluţie, şi

anume *partajarea ponderilor* şi *conectivitatea „rară”*, care nu există în cadrul reţelelelor multistrat. Aceasta înseamnă că trebuie memoraţi mai puţini parametri, ceea ce reduce cerinţele de memorie ale reţelelor. De asemenea, calculele se fac mult mai repede dacă înmulţirea cu o matrice de dimensiuni mari este înlocuită cu convoluţia cu un nucleu de convoluţie de dimensiuni mult mai mici.

O importanţă deosebită o au în cadrul reţelelor de convoluţie straturile de pooling. Acestea induc o a treia propritate importantă a reţelelor de convoluţie, şi anume *invarianţa la translaţii* mici ale intrării. Aceasta este foarte importantă în recunoaşterea de imagini, pentru că, în astfel de aplicaţii, ne interesează mai mult prezenţa anumitor obiecte în acele imagini, mai degrabă decât poziţia lor exactă. Ceea ce se întâmplă în aceste straturi este că ieşirea stratului anterior este înlocuită cu un rezumat statistic al ieşirilor din imediata sa vecinătate. Vom considera că această statistică este dată de media aritmetică, astfel relaţiile (2.5)-(2.6) având forma:

$$\mathbf{s}^l[j] = \frac{1}{n^2}\mathbf{x}^{l-1}[j] * \mathbf{1}_{n\times n},\ \forall j \in \{1,\dots,J\}$$

$$\mathbf{x}^l[j] = \text{down}(\mathbf{s}^l[j]),\ \forall j \in \{1,\dots,J\}.$$

unde J reprezintă numărul de canale de intrare în stratul de pooling, care este egal cu numărul de canale de ieşire, aceasta fiind o caracteristică a acestui tip de strat. $\mathbf{1}_{n\times n}$ reprezintă o matrice care are toate intrările egale cu 1, iar down reprezintă operaţia de downsampling prin care se reduce dimensiunea intrării în stratul de pooling de n ori, n fiind dimensiunea vecinătăţii pe care se face medierea. Prima relaţie este tot o convoluţie, care înlocuieşte fiecare valoare dintr-o regiune pătratică de dimensiune n cu media artimetică a tuturor valorilor din această regiune, urmând ca a doua relaţie să păstreze doar această medie, ignorând valorile care au dat-o. De exemplu,

$$\text{down}\left(\frac{1}{2^2}\begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}\right)$$

$$= \text{down}\left(\begin{bmatrix} \frac{x_{11}+x_{12}+x_{21}+x_{22}}{4} & \frac{x_{12}+x_{13}+x_{22}+x_{23}}{4} & \frac{x_{13}+x_{14}+x_{23}+x_{24}}{4} \\ \frac{x_{21}+x_{22}+x_{31}+x_{32}}{4} & \frac{x_{22}+x_{23}+x_{32}+x_{33}}{4} & \frac{x_{23}+x_{24}+x_{33}+x_{34}}{4} \\ \frac{x_{31}+x_{32}+x_{41}+x_{42}}{4} & \frac{x_{11}+x_{12}+x_{21}+x_{22}}{4} & \frac{x_{33}+x_{34}+x_{43}+x_{44}}{4} \end{bmatrix}\right)$$

$$= \begin{bmatrix} \frac{x_{11}+x_{12}+x_{21}+x_{22}}{4} & \frac{x_{13}+x_{14}+x_{23}+x_{24}}{4} \\ \frac{x_{31}+x_{32}+x_{41}+x_{42}}{4} & \frac{x_{33}+x_{34}+x_{43}+x_{44}}{4} \end{bmatrix}.$$

Se observă că straturile de pooling nu au ponderi.

Configuraţia generală a unei reţele de convoluţie presupune alternarea straturilor de convoluţie cu straturile de pooling. Astfel, fiecărui strat de convoluţie îi urmează un strat de pooling, care are acelaşi număr de canale de intrare şi de ieşire. De obicei, ultimul strat al reţelei este unul total conectat, identic cu cele dintr-o reţea multistrat.

Învăţarea într-o reţea de convoluţie se bazează pe acelaşi algoritm backpropagation, ca în cazul reţelelor multistrat. Actualizarea ponderilor are loc doar pentru straturile

de convoluţie (şi pentru stratul total conectat), însă valori $\boldsymbol{\delta}^l$ trebuie calculate şi pentru straturile de pooling. Pentru stratul total conectat, relaţia (2.10) sau (2.11) dă valoarea lui $\boldsymbol{\delta}^l$. În schimb, pentru un strat de convoluţie, relaţia (2.10) se poate scrie

$$\boldsymbol{\delta}^l[j] = \text{up}(\boldsymbol{\delta}^{l+1}[j]) \circ \overline{(G^l)'(\mathbf{s}^l[j])},\ \forall j \in \{1, \ldots, J\},$$

iar relaţia (2.11):

$$\boldsymbol{\delta}^l[j] = \text{up}(\boldsymbol{\delta}^{l+1}[j]) \star \frac{\partial G^l(\mathbf{s}^l[j])}{\partial \mathbf{s}^l[j]},\ \forall j \in \{1, \ldots, J\},$$

unde J reprezintă numărul de canale de ieşire ale stratului de convoluţie, care este egal cu numărul de canale de intrare şi ieşire ale stratului de pooling $l+1$. Operaţia up reprezintă o operaţie de upsampling, prin care se măreşte dimensiunea argumentului de n ori, n fiind dimensiunea vecinătăţii pe care se face medierea în stratul de pooling $l+1$, fiind opusă operaţiei de downsampling care reducea dimensiunea intrării în stratul de pooling. Practic, fiecare element al lui $\boldsymbol{\delta}^{l+1}[j]$ este transformat într-o matrice bloc de dimensiune n, care are toate elementele egale cu acest element din matricea iniţială. De exemplu,

$$\text{up}\left(\begin{bmatrix}\delta_{11} & \delta_{12}\\ \delta_{21} & \delta_{22}\end{bmatrix}\right) = \begin{bmatrix}\delta_{11} & \delta_{11} & \delta_{12} & \delta_{12}\\ \delta_{11} & \delta_{11} & \delta_{12} & \delta_{12}\\ \delta_{21} & \delta_{21} & \delta_{22} & \delta_{22}\\ \delta_{21} & \delta_{21} & \delta_{22} & \delta_{22}\end{bmatrix}.$$

Acum, valorile $\boldsymbol{\delta}^l$ pentru un strat de pooling se calculează cu formula

$$\boldsymbol{\delta}^l[i] = \sum_{j=1}^{J} \boldsymbol{\delta}^{l+1}[j] * \text{rot180}(\overline{\mathbf{k}^{l+1}[i,j]}),\ \forall i \in \{1, \ldots, I\}, \tag{2.12}$$

unde I reprezintă numărul de canale de intrare şi de ieşire ale stratului de pooling, iar J reprezintă numărul de canale de ieşire ale stratului de convoluţie $l+1$. Operaţia rot180 roteşte matricea argument cu 180 de grade, fiind corespunzătoare în cazul convoluţiei a transpusei din cazul înmulţirii de matrici, prezente în ecuaţiile (2.10)-(2.11). Prin urmare, $\text{rot180}(\overline{\mathbf{k}^{l+1}[i,j]})$ este echivalent cu $(\mathbf{W}^{l+1})^H$ de la reţelele multistrat. De exemplu,

$$\text{rot180}\left(\overline{\begin{bmatrix}k_{11} & k_{12}\\ k_{21} & k_{22}\end{bmatrix}}\right) = \overline{\begin{bmatrix}k_{22} & k_{21}\\ k_{12} & k_{11}\end{bmatrix}} = \begin{bmatrix}\overline{k_{22}} & \overline{k_{21}}\\ \overline{k_{12}} & \overline{k_{11}}\end{bmatrix}.$$

Mai trebuie făcută observaţia că operaţia de convoluţie din (2.12) se realizează prin bordarea matricii $\boldsymbol{\delta}^{l+1}[j]$ cu zerouri, astfel încât dimensiunea matricii rezultate în urma convoluţiei este egală cu dimensiunea matricii $\boldsymbol{\delta}^{l+1}[j]$ plus dimensiunea nucleului de convoluţie minus 1. De exemplu,

$$\begin{bmatrix}0 & 0 & 0 & 0\\ 0 & \delta_{11} & \delta_{12} & 0\\ 0 & \delta_{21} & \delta_{22} & 0\\ 0 & 0 & 0 & 0\end{bmatrix} * \text{rot180}\left(\overline{\begin{bmatrix}k_{11} & k_{12}\\ k_{21} & k_{22}\end{bmatrix}}\right) = \begin{bmatrix}0 & 0 & 0 & 0\\ 0 & \delta_{11} & \delta_{12} & 0\\ 0 & \delta_{21} & \delta_{22} & 0\\ 0 & 0 & 0 & 0\end{bmatrix} * \begin{bmatrix}\overline{k_{22}} & \overline{k_{21}}\\ \overline{k_{12}} & \overline{k_{11}}\end{bmatrix}$$

$$= \begin{bmatrix} \delta_{11}\overline{k_{22}} & \delta_{11}\overline{k_{21}} + \delta_{12}\overline{k_{22}} & \delta_{12}\overline{k_{21}} \\ \delta_{11}\overline{k_{12}} + \delta_{21}\overline{k_{22}} & \delta_{11}\overline{k_{11}} + \delta_{12}\overline{k_{12}} + \delta_{21}\overline{k_{21}} + \delta_{22}\overline{k_{22}} & \delta_{12}\overline{k_{11}} + \delta_{22}\overline{k_{21}} \\ \delta_{21}\overline{k_{12}} & \delta_{21}\overline{k_{11}} + \delta_{22}\overline{k_{12}} & \delta_{22}\overline{k_{11}} \end{bmatrix}.$$

În final, mai trebuie să dăm o formulă pentru actualizarea nucleelor de convoluţie, deoarece formula pentru actualizarea ponderilor stratului total conectat din reţeaua de convoluţie este aceeaşi cu cea dată în relaţia (2.9). Această relaţie are, pentru nucleul de convoluţie $\mathbf{k}^l[i,j]$, următoarea formă:

$$\Delta\mathbf{k}^l[i,j] = -\varepsilon \text{rot180}(\overline{\mathbf{x}^{l-1}[i]}) * \boldsymbol{\delta}^l[j],\ \forall i \in \{1,\ldots,I\},\ \forall j \in \{1,\ldots,J\},$$

unde ε reprezintă rata de învăţare, analoagă celei de la reţelele multistrat, I reprezintă numărul de canale de intrare în stratul de convoluţie l, iar J reprezintă numărul canalelor de ieşire din stratul de convoluţie l. Din nou, se observă că $(\mathbf{x}^{l-1})^H$ a fost înlocuit în acest caz cu $\text{rot180}(\overline{\mathbf{x}^{l-1}[i]})$.

Capitolul 3

Specificaţiile proiectului

Presupunând că utilizatorii soluției prezentate vor face parte din domenii apropiate cu cel al tehnologiei informației și rezultatul dezvoltării se va folosi în scopuri de cercetare si testare, componenta software va avea o interfață cu utilizatorul minimală.

La rularea programului, datele de intrare extrase eventual dintr-o imagine care se dorește a fi analizată vor fi introduse de către utilizator fie prin intermediul unei ferestre care se va deschide și va permite incărcarea directă a imaginii, fie prin crearea unui fișier ce va conține informațiile care caracterizează imaginea/setul de imagini care se doresc a fi clasificate.

Prin această metodă se vor permite introducerea datelor sub o formă neprelucrată (prin încarcarea directă a imaginii), dar și date de intrare care caracterizează o imagine/un set de imagini pe care s-au aplicat anumite prelucrări sau filtre. În general rezultatele obținute pot fi îmbunătățite prin filtrarea sau aplicarea de transformări asupra imaginilor care se doresc a fi analizate/clasificate, însă efectul acestor prelucrări este strict dependent de domeniul din care sunt preluate imaginile, respectiv de calitatea instrumentelor care le-au generat inițial.

Un al doilea aspect important în interfața programului cu utilizatorul îl constituie setarea parametrilor de configurare menționați. Acești parametrii au rolul de a defini structura internă a rețelei și de asemenea de a seta o eroare maximă acceptată pentru a se finaliza antrenarea și coeficientul de actualizare ai parametrilor de control interni.

Structura rețelei de convoluție se definește prin următoarele elemente configurabile: numărul de straturi, tipul fiecărui strat și numărul de hărți de intrare/ieșire pe fiecare strat. Acestea pot fi de tipul strat de convoluție sau pooling. Astfel, pe lângă aceste elemente esențiale de configurare, se adaugă și elementele specifice fiecărui strat. Printre acestea se numără dimensiunea nucleului de convoluție și numarul de nuclee pentru fiecare dintre straturile de tip convoluție, iar pentru straturile de pooling se poate specifica rata de sub-sampling.

Pornind de la un strat de intrare, care conține datele extrase din imagine sau primite într-un fișier extern dupa o anumită prelucrare, se pot adăuga un număr arbitrar de straturi. Determinarea numărului potrivit de straturi este strâns legată de tipul și de dimensiunea hărților de intrare folosite pentru antrenarea rețelei. Se ține cont astfel

de reducerea în dimensiune a acestor hărți în funcție de stratul prin care acestea sunt prelucrate. În cazul unui strat de convoluție, rezultatul operațiilor va avea dimensiuni reduse față de cele ale hărților de intrare in stratul respectiv în funcție de dimensiunea nucleelor de convoluție care se aplică. În mod similar, dimensiunea rezultatului operației de max-pooling va fi mai mică decât cea a hărților de intrare pe stratul de pooling, în funcție de rata de sub-sampling.

Pentru a putea memora configurația rețelei se vor folosi structuri de date particulare specializate în acest scop. Conținutul acestor structuri de date va ramâne neschimbat pe parcursul antrenării și va putea fi setat doar înainte de începerea acestei etape.

O altă facilitate importantă inclusă constă în abilitatea de a citi și interpreta informațiile de pe senzorul de temperatură al procesorului grafic, daca acesta există. Aceste informații pot fi utile pentru a monitoriza performanța componentei si gradul de utilizare al acesteia. În funcție de nivelul temperaturii se poate afla dacă unitatea de procesare este utilizată la capacitate maximă sau dacă utilizarea acesteia este ineficientă. În cazul detectării unei astfel de probleme acțiunea recomandată va fi aceea de a verifica și modifica numărul de fire de execuție în care se împarte sarcina în timpul unei operații efectuate de către GPU.

În cazul în care dintr-un motiv sau altul nu se poate continua antrenarea rețelei, programul își va încheia execuția cu un cod de eroare specific. De exemplu, funcționarea nu poate continua dacă apare o eroare în alocarea zonei de memorie de lucru, sau în cazul în care nu se poate deschide un fișier pentru citire/scriere. O altă eroare de execuție care poate apărea este inexistența unui GPU capabil să execute operațiile necesare sau în cazuri extrem de rare nerecunoașterea acestuia. Aceste tipuri de erori sunt tratate corespunzător și utilizatorul poate fi anunțat de motivul încheierii execuției programului.

Capitolul 4

Proiectare şi implementare

4.1 Programarea pe GPU

Datorită cererii tot mai mari de grafică 3D high-definition obținută în timp real, unitățile programabile Graphic Processor Unit(GPU) au cunoscut în ultimii ani o evoluție spre procesoare multi-core capabile de a efectua operații în paralel, pe mai multe fire de execuție obținându-se astfel o putere de calcul imensă. În același timp se recunoaște o creștere și în lățimea de bandă de memorie. Aceste fapte sunt vizibile în figurile următoare.

Figura 4.1 conține un grafic prin care se efectuează comparația unor modele de GPU produse de NVIDIA și diferite generații de procesoare de uz general CPU produse de Intel din punctul de vedere al numărului de operații în virgulă flotantă efectuate într-o secundă.

În Figura 4.2 se compară lățimea de bandă pentru memorie a procesoarelor GPU cu cea a procesoarelor de uz general:

Se observă așadar că din aceste puncte de vedere performanța GPU-urilor arată o creștere mult mai mare decât cea a procesoarelor de uz general. Motivul pentru care procesoarele GPU sunt mai performante din punctul de vedere al capabilității computaționale în virgulă flotantă este acela că procesoarele grafice sunt specializate în computații intensive în mod paralel, aceasta fiind cerința principală pentru redarea grafică. Prin urmare GPU-urile sunt proiectate astfel încât majoritatea tranzistorilor sunt dedicați procesării de date spre deosebire de cei ai unui CPU care sunt orientați și spre cache sau controlul fluxului de date. Acest fapt este evidențiat schematic în Figura 4.3.

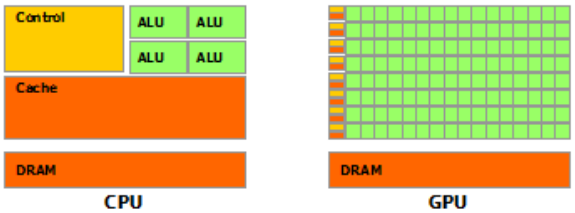

Figura 4.3: GPU-ul este orientat spre procesarea datelor [11]

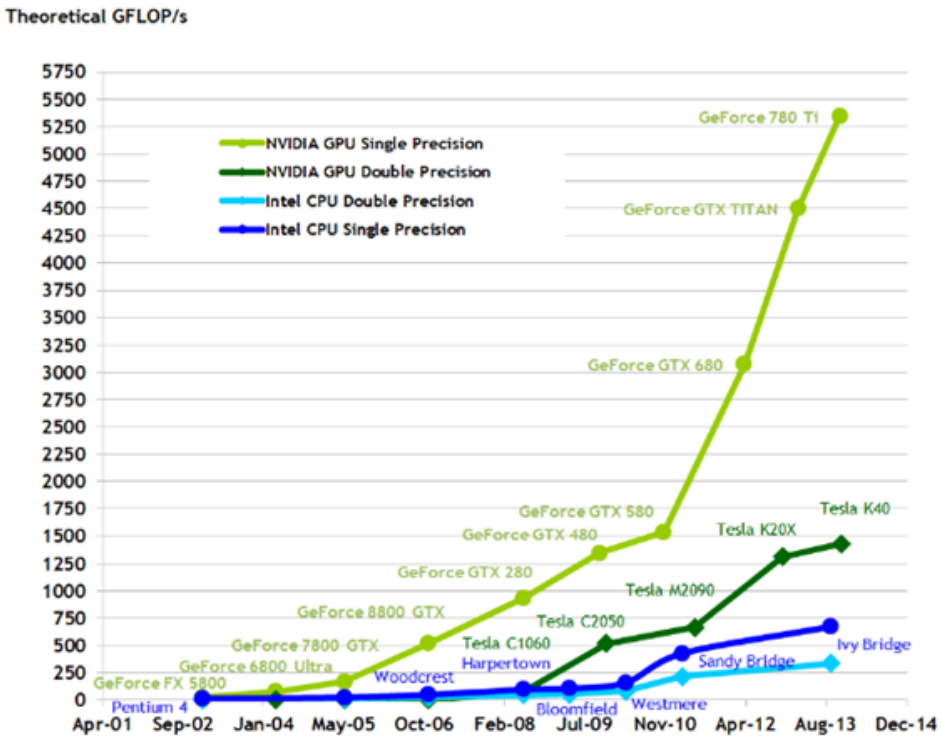

Figura 4.1: Operaţii în virgulă flotantă pe secundă pentru CPU şi GPU [11]

Mai precis, unitatea de procesare grafică este deosebit de potrivită pentru pentru a aborda problemele care pot fi exprimate sub formă de calcule paralele, adică același program/rutină care se rulează de mai multe ori pe seturi de date diferite și care are o intensitate aritmetică ridicată. Acest fapt este determinat din raportul dintre numărul de operații aritmetice si numărul de accesări al memoriei în timpul unei execuții al programului/rutinei.

Întrucât același program este rulat pe seturi de date diferite, cerința pentru un control sofisticat al fluxului de date este mică, iar pentru că intensitatea aritmetică este una ridicată latența de acces la memorie poate fi ascunsă cu calcule în locul cache-urilor mari de date.

Procesarea paralelă a datelor distribuie sarcina de calcul între mai multe fire de execuție. Un număr ridicat de aplicații care prelucrează seturi mari de date pot utiliza un model de programare paralelă pentru a accelera calculele. În cazul redării 3D, seturi mari de pixeli și noduri sunt mapate la fire paralele. În mod similar, aplicațiile de procesare a imaginilor și a altor obiecte multimedia cum ar fi post-procesarea imaginilor redate, scalarea imaginilor, codarea si decodarea video pot asocia blocuri de imagini și pixeli firelor de procesare paralelă. De asemenea, mulți algoritmi din afara domeniului de procesare de imagini pot utiliza procesarea paralelă, de la procesarea în general a semnalelor sau simulări din domeniul fizicii, până la inclusiv în domeniul finanțelor și al biologiei.

Din aceste considerente, am ales pentru a accelera calculele folosite pentru antrenarea unei rețele neuronale, ca implementarea să implice și utilizarea unui GPU(produs de

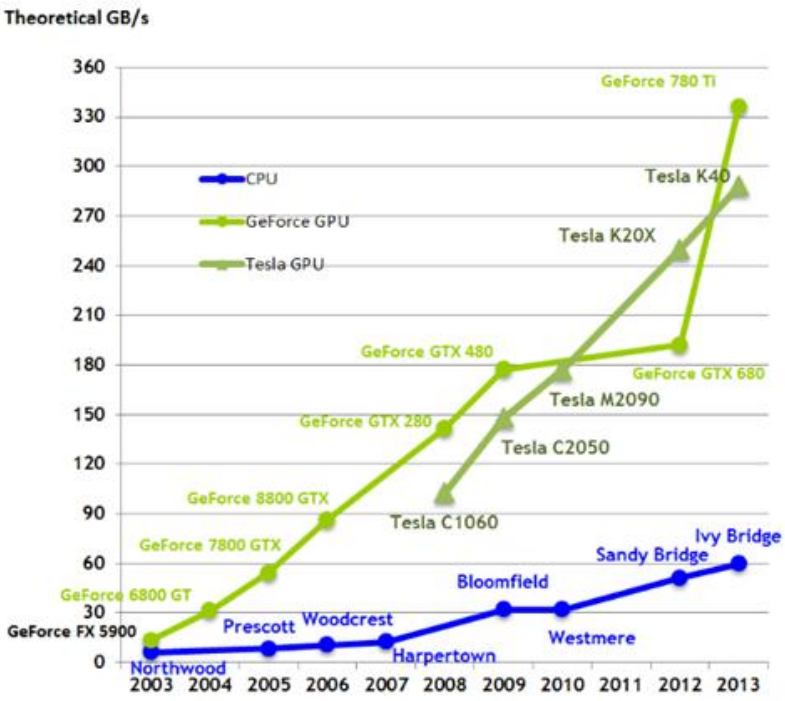

Figura 4.2: Banda de trecere a memoriei pentru CPU şi GPU [11]

NVIDIA). Pentru a putea utiliza capacitatea de calcul a unei unități grafice, aceasta trebuie în primul rând sa fie programabilă și să suporte platforma și modelul de programare CUDA.

CUDA a fost introdus de către NVIDIA în noiembrie 2006. Aceasta este o platformă de calcul paralel de uz general și un model de programare care folosește motorul de calcul paralel din unitățile de procesare de la NVIDIA pentru a rezolva probleme complexe de calcul într-un mod mai eficient decât pe un procesor de uz general.

Împreună cu această platformă vine și un mediu de dezvoltare software care permite dezvoltatorilor să utilizeze limbajul C ca un limbaj de programare de nivel înalt. Din documentația platformei pusă la dispoziție de NVIDIA reiese faptul că sunt suportate și alte limbaje de programare sau abordări bazate pe directive cum ar fi FORTRAN, DirectCompute sau OpenACC. Provocarea este de a dezvolta o aplicație software care-și scalează într-un mod transparent paralelismul astfel încât să poate face față numărului tot mai mare de nuclee de procesare, la fel cum aplicațiile de grafică 3D își scalează paralelismul la diverse unități grafice cu număr variabil de nuclee.

Modelul de programare paralelă CUDA este proiectat întocmai pentru a depăși această provocare menținând în același timp o curbă de învățare scăzută pentru programatorii familiarizați cu limbaje de programare standard, cum ar fi C. CUDA se bazează pe trei abtracțiuni cheie: o ierarhie a grupurilor de fire de execuție, memoria partajată și sincronizarea acestora, care sunt expuse către dezvoltator ca un set minim de extensii ale limbajului de programare.

Aceste abstracțiuni furnizează paralelismul de date cu granulație fină și paralelismul firelor de execuție. Ele ghidează dezvoltatorul să partajeze problema într-un număr de

probleme mai mici care pot fi apoi rezolvate în mod independent de blocuri de fire de execuție. Această partajare permite păstrarea expresivității limbajului permițând comunicarea și cooperarea firelor de execuție în rezolvarea unei sarcini și în același timp permite scalabilitatea automată. Astfel, modelul scalabil de programare permite soluției dezvoltate să ruleze pe o gamă diversă de arhitecturi GPU cu număr variabil de nuclee de procesare prin simpla eșalonare a numărului de multiprocesoare și partițiilor de memorie de care dispune unitatea respectivă.

În Figura 4.4 se arată cum un program care a respectat scalabilitatea transparentă poate rula pe două unități grafice distincte cu arhitecturi diferite. Fiind efectuată partajarea care permite executarea independentă a sarcinilor care compun o problemă, un GPU cu un număr mai mare de multiprocesoare va rezolva problema dată într-un timp mai scurt decât un GPU cu un număr mai mic de nuclee.

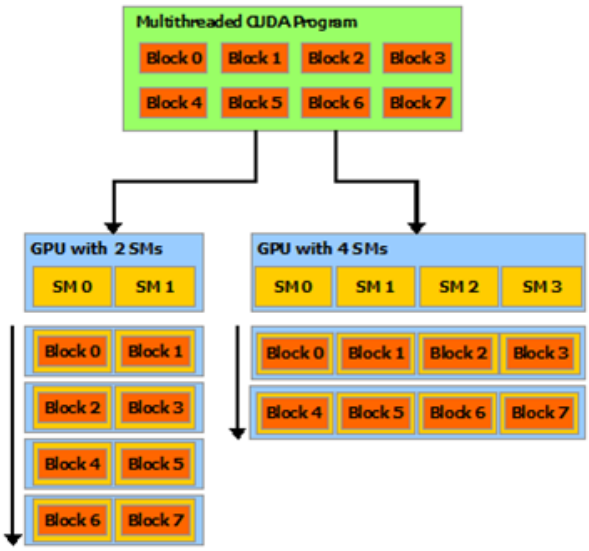

Figura 4.4: Scalabilitatea transparentă [11]

4.1.1 CUDA C

CUDA C oferă o cale simplă pentru utilizatorii familiarizați cu limbajul de programare C pentru a scrie cu ușurință programe care vor fi executate de către dispozitiv. Acesta este compus dintr-un set minim de extensii pentru limbajul C și o bibliotecă de execuție. Extensiile de bază permit programatorilor să definească un nucleu(kernel) de execuție ca o funcție C și să utilizeze unele elemente de sintaxă noi pentru a specifica dimensiunea grilei si a blocului de fiecare dată când funcția este apelată. Orice fișier sursă care conține unele dintre aceste extensii trebuie să fie compilate cu NVCC.

NVCC este un mecanism de compilare care simplifică procesul de compilare al codului sursă. Acesta oferă opțiuni de executare din linia de comandă familiare și executarea lor invocând o colecție de unelte care implementează diferitele etape ale compilării.

4.1.2 Kernel-uri

CUDA C ca extensie a limbajului de programare C permite programatorului sa defineasca funcții C numite kernels care atunci când sunt apelate se execută de un număr N de ori de către N fire de execuție CUDA diferite, spre deosebire de funcțiile C obișnuite. Un nucleu de execuție(kernel) se definește utilizând specificatorul de declarație „__global__" și numărul de fire CUDA care îl execută întrucat un apel al kernel-ului se formează utilizând o sintaxă nouă de configurare a execuției, și anume "<<<...>>>". Fiecărui fir care execută kernel-ul definit i se atribuie un identificator unic care este accesibil prin variabila threadIdx oferită.

Prin convenție, variabila prin care i se atribuie fiecărui fir de execuție un identificator unic este un vector cu trei componente. Astfel firele de execuție se pot identifica utilizând un index al firului care poate avea una, două sau trei dimensiuni și care poate forma un bloc de fire unidimensional, bidimensional sau tridimensional. Acest aspect facilitează un mod natural de a invoca o anumită operație pe elementele unui domeniu cum ar fi un vector sau o matrice.

Datorită presupunerii că toate firele unui bloc de execuție vor aparține aceluiași nucleu al procesorului pe care rulează și sunt nevoite să împartă resursele limitate ale acelui nucleu, există limitări în ceea ce privește numărul de fire de execuție care pot fi conținute de un bloc. Pentru GPU-urile actuale, aceasta limită este de 1024 fire de execuție per bloc. Totuși, un kernel poate fi executat de către mai multe blocuri de fire de execuție având formă similară și astfel numărul total de fire de execuție va fi egal cu numărul de fire dintr-un bloc înmulțit cu numărul total de blocuri.

Un bloc de fire de execuție poate fi organizat ca o grilă unidimensională, bidimensională sau tridimensională, fapt ilustrat in Figura 4.5. Numărul blocurilor de fire de execuție este în general dictat de dimensiunea datelor care trebuie procesate și de numărul de procesoare care fac parte din sistem.

Numărul firelor de execuție dintr-un bloc si numărul de blocuri dintr-o grilă specificate prin sintaxa <<<...>>> pot fi de tip int sau dim3. Fiecare bloc aparținând grilei poate fi identificat de un index unidimensional, bidimensional sau tridimensional, accesibil in interiorul kernel-ului prin variabila blockIdx oferită. De asemenea este posibilă accesarea directă din interiorul kernel-ului a dimensiunii blocului de fire de execuție curent prin variabila blockDim oferită.

Este necesar ca un bloc de fire de execuție să se poată executa in mod independent de celalalte, fie că execuția se face in paralel sau secvențial. Este important ca blocurile de fire să se poată executa in orice ordine. Această caracteristică de independență descrisă permite execuția blocurilor de fire în orice ordine pe mai multe nuclee de procesare ale dispozitivului și astfel permite programatorului să scrie cod scalabil cu numărul de nuclee care variază de la un dispozitiv la altul.

Firele de execuție aparținând unui bloc pot coopera interschimbând date printr-o memorie comună și sincronizându-și execuția pentru a coordona accesele la această memorie. O asemenea sincronizare se poate specifica în interiorul unui kernel apelând funcția

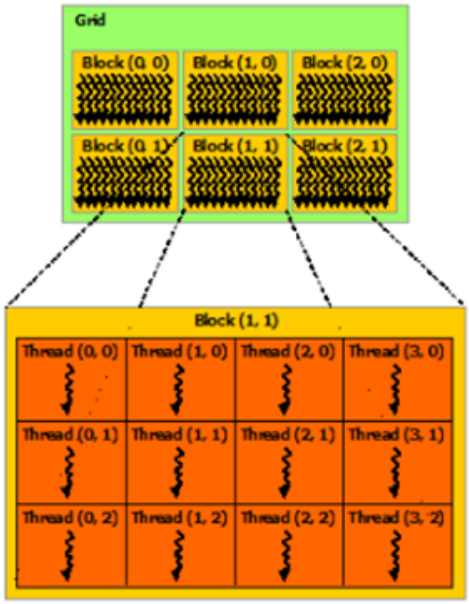

Figura 4.5: Grilă de blocuri de fire de execuţie [11]

_syncthreads(). Aceată funcție prezintă comportamentul unei bariere unde toate firele de execuție dintr-un bloc trebuie să aștepte înainte sa li se permită să continue. Pentru o cooperare eficientă a execuției firelor din interiorul unui bloc se așteaptă ca zona de memorie comună să aibă o latență scăzută.

4.1.3 Ierarhia de memorie

Firele de execuție CUDA pot accesa date din multiple surse de memorie în timpul execuției lor, fapt ilustrat in Figura 4.6. Fiecare fir are propria memorie locală privată. Fiecare bloc de fire deține o zona de memorie accesibilă tuturor firelor de execuție pe care le conține. Aceste zone de memorie sunt disponibile doar pe perioada de timp în care se execută blocul de fire de execuție. În plus, toate blocurile de fire au acces la o aceeași zonă de memorie globală. Există de asemenea două zone de memorie adiționale accesibile de către toate firele de execuție, și anume zona constantelor și zona texturilor. Aceste zone sunt persistente între lansări de kernel-uri din interiorul aceleiași aplicații.

În modelul de programare CUDA se presupune că firele de execuție CUDA se vor executa pe un dispozitiv separat care acționează ca un co-procesor pentru procesorul gazdă pe care se rulează programul C. Într-adevăr asta se intamplă atunci când kernel-urile sunt lansate spre execuție pe dispozitivul GPU iar restul programului se execută pe CPU. Acest mecanism este evidențiat în Figura 4.7.

De asemenea modelul de programare CUDA presupune ca dispozitivul host si dispozitivul ajutător să acceseze zone diferite ale memoriei DRAM, zone ce poartă denumirea de memoria host și memoria dispozitivului.

Zona de memorie a dispozitivului se poate aloca fie ca fiind memorie liniară sau ca

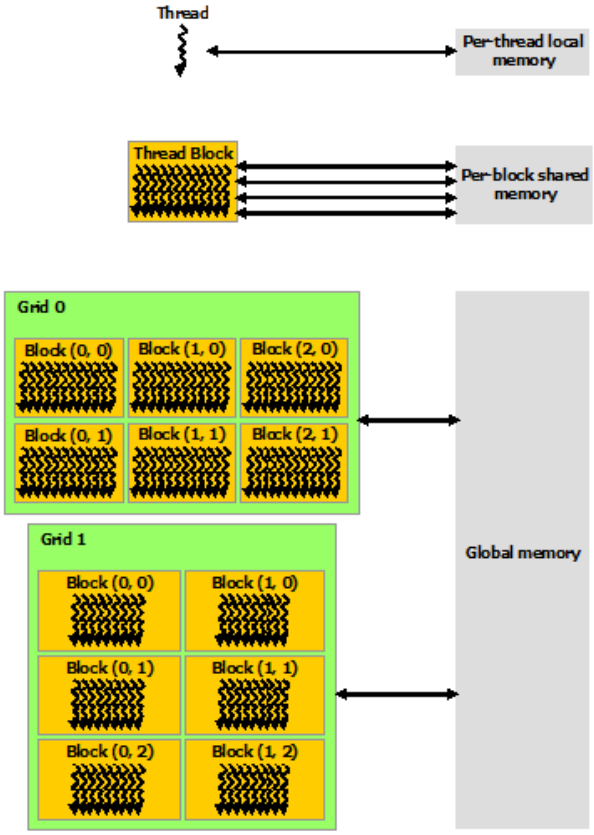

Figura 4.6: Ierarhia de memorie [11]

fiind vectori CUDA. Vectorii CUDA sunt zone de memorie cu un aspect opac optimizate pentru preluarea de texturi.

Memoria liniară este disponibilă pe dispozitiv într-un spațiu de adresare de 40 biți astfel încât entitățile alocate separat pot face referire una la alta folosind pointerii. Memoria liniară se alocă în general folosind funcția cudaMalloc() și se eliberează utilizând funcția cudaFree(). Transferul de date între dispozitivul gazdă și GPU se face folosind cudaMemcpy().

Din aceste considerente, am ales pentru a accelera calculele folosite pentru antrenarea unei rețele neuronale, ca implementarea să implice și utilizarea unui GPU(produs de NVIDIA). Pentru a putea utiliza capacitatea de calcul a unei unități grafice, aceasta trebuie în primul rând sa fie programabilă și să suporte platforma și modelul de programare CUDA.

Printre uneltele puse la dispoziție prin platforma de dezvoltare CUDA folosită se numără și următoarele biblioteci optimizate pentru accelerarea GPU:

- cuFFT – transformate Fourier rapide;
- cuBLAS – subrutine pentru algebra liniară de bază;
- cuSPARSE – rutine pentru operații pe matrici rare;
- cuRAND – generare de numere aleatoare;

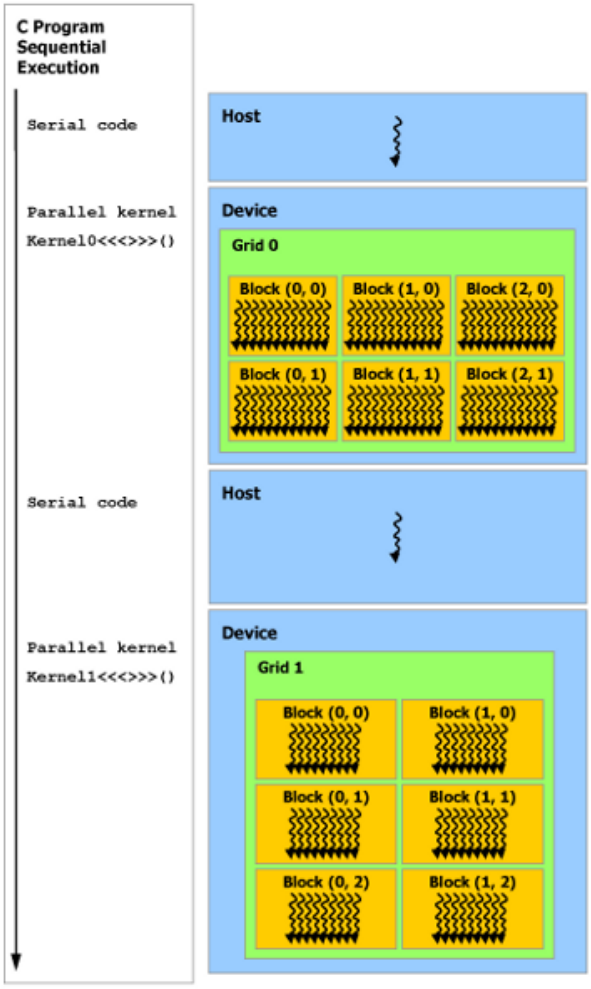

Figura 4.7: Programarea eterogenă [11]

- NPP – primitive pentru procesarea de imagini și video;
- nvGRAPH – bibliotecă pentru analiză grafică NVIDIA;
- biblioteca pentru operații matematice CUDA.

4.1.4 Biblioteca cuBLAS

CUBLAS este o implementare a BLAS (Basic Linear Algebra Subroutines) având la bază modelul de programare CUDA de la NVIDIA [10]. Prin utilizarea acestei biblioteci se face posibil accesul resurselor computaționale a GPU-urilor NVIDIA. Biblioteca este independentă, adică nu este nevoie de o interacțiune directă cu driver-ul CUDA.

Utilizarea acestei biblioteci în implementarea soluției curente constă în crearea de obiecte de tip matrice și vector în spațiul de memorie al unității grafice, apelarea unor funcții cuBLAS pe datele respective și, în final, transferul rezultatelor de pe GPU înapoi în zona de memorie generală de lucru. Aceste acțiuni sunt posibile datorită existenței unor funcții ajutătoare în biblioteca cuBLAS, care se folosesc pentru alocarea de memorie, pentru scrierea de date în zona de memorie alocată, pentru transferul de date între GPU

și CPU, iar în final ștergerea datelor din spațiul de memorie al GPU atunci când nu se mai folosesc informațiile respective.

Întrucât funcțiile de bază ale acestei biblioteci (spre deosebire de funcțiile ajutătoare menționate) nu returnează în mod direct un cod de eroare atunci când este cazul, s-a implementat o funcție specializată în acest sens care poate prelua ultima eroare înregistrată. Acest aspect este important pentru modul de depanare al aplicației dezvoltate.

Accelerarea calculelor folosind această bibliotecă este ilustrată în Figura 4.8. Comparația se face între modelul GPU NVIDIA Tesla K40M folosind cuBLAS și procesorul de uz general Intel IvyBridge single socket 12 -core E5-2697 v2 @ 2.70GHz folosind biblioteca echivalentă MKL (Math Kernel Library) de la Intel. Se observă că numărul de operații în virgulă flotantă pe secundă este considerabil mai mare cu implementarea cuBLAS.

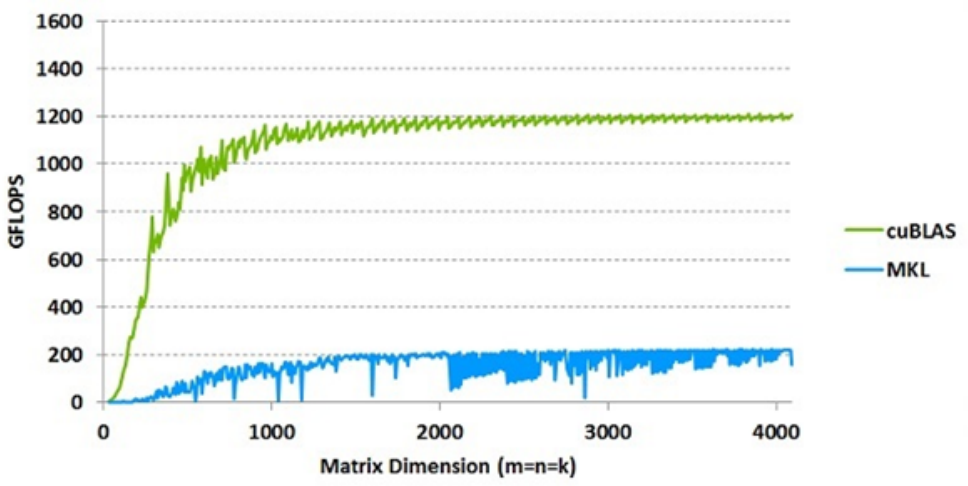

Figura 4.8: Accelerarea calculelor folosind biblioteca cuBLAS [10]

În implementarea cuBLAS se definește un singur tip de date, și anume tipul cublasStatus. Acest tip este folosit pentru aflarea stării după ce se apelează o funcție din această bibliotecă. În cazul funcțiilor de bază care nu returnează starea finală, acest parametru se poate afla apelând cublasGetError() care va returna ultima stare înregistrată.

4.1.5 Biblioteca cuComplex

Biblioteca cuComplex oferă sprijin în utilizarea numerelor complexe la dezvoltarea unei soluții în limbajul de programare C. În această bibliotecă se găsesc implementări pentru subrutine de bază care descriu operațiile întâlnite în rezolvarea unei probleme având ca date numere complexe. Un număr complex este astfel reprezentat printr-un tip de date abstract având două componente. Prin aceste componente se pot memora/accesa în mod natural cele două părți ale unui număr complex: partea reală și partea imaginară. Cele două componente ale numerelor complexe se reprezintă prin numere reale, care în cazul de față pot fi de tipul simplă precizie (float) sau dublă precizie (double). Astfel se pot crea două tipuri de numere complexe, și anume tipul cuFloatComplex și respectiv tipul

de date cuDoubleComplex. Componentele primului tip – cuFloatComplex vor fi numere reale de tip float cu simplă precizie, iar pentru structura cuDoubleComplex sunt utilizate două câmpuri de tip double, adică avand dublă precizie. Pentru fiecare din aceste tipuri se pun la dispoziție câte un set de funcții ce vor primi ca parametru/parametri unul din tipurile de date descris anterior. În general și valoarea returnată va fi de același tip cu cel al parametrilor funcției apelate.

Operațiile puse la dispoziție de biblioteca cuComplex sunt des utilizate în cadrul soluției curente. Sunt apelate funcții pentru a crea variabile de tip complex, pentru efectuarea operațiilor aritmetice între aceste variabile și în declararea unor structuri de date mai complexe utilizate în rezolvarea problemei. Pentru a putea accesa componentele variabilei de tip complex, se folosesc câmpurile care definesc acest tip notate cu x respectiv y. Prin câmpul x se acceseaza partea reală a numărului complex iar prin câmpul y partea imaginară a acestuia.

Alternativ se poate apela una din funcțiile următoare:

```
__host__ __device__ static __inline__ double cuCreal(cuDoubleComplex x)
__host__ __device__ static __inline__ double cuCimag(cuDoubleComplex x)
```

dacă se utilizează tipurile de date cu dublă precizie, respectiv:

```
__host__ __device__ static __inline__ float cuCrealf(cuFloatComplex x)
__host__ __device__ static __inline__ float cuCimagf(cuFloatComplex x)
```

Antetul acestor funcții sugerează faptul că pot fi apelate atât din secvențele de cod care vor fi eventual executate de un procesor de uz general(host) cât și de către procesorul grafic(device). Fiind denumite sugestiv, nu este necesar ca dezvoltatorii să cunoască și câmpurile structurii tipului complex. Funcțiile prezentate anterior vor returna o valoare reală reprezentând partea reală a unui număr complex respectiv partea imaginară a acestuia.

Pentru construirea unui număr complex se pun la dispoziție funcțiile:

```
__host__ __device__ static __inline__ cuFloatComplex
    make_cuFloatComplex(float r, float i)
__host__ __device__ static __inline__ cuDoubleComplex
    make_cuDoubleComplex(double r, double i)
```

Utilitatea acestora constă în declararea automată a unei variabile de tip complex, setarea câmpurilor acesteia prin inițializarea cu valorile date ca parametru și returnarea unei referințe către noua variabilă creată. Din antetul acestor funcții se observă de asemenea contextul din care pot fi apelate: host și device.

Funcțiile puse la dispoziție pentru operații aritmetice sunt:

- cuCadd – pentru adunarea a două numere complexe

- cuCsub – calculează diferența dintre două numere complexe
- cuCmul – returnează rezultatul înmulțirii parametrilor dați ca intrare
- cuCdiv – pentru raport
- cuCabs – returnează modulul numărului dat ca parametru de intrare.

4.2 Soluția propusă

Implementarea aplicației s-a realizat folosind mediul de dezvoltare Visual Studio oferit de Microsoft. Acesta permite instalarea extensiei CUDA oferită de NVIDIA. Versiunea folosită pentru dezvoltare este CUDA 6.5.

Programul este format din partea principală și funcții ajutătoare. În prima parte sunt declarate structurile de date specializate prin care se inițializează și apoi se accesează parametrii de antrenare ai rețelei. Acestea sunt prezentate succint în cele ce urmează.

Structura de date numită options va memora în câmpurile sale câteva opțiuni generale de care se va ține cont în antrenarea rețelei.

```
typedef struct {
    int alpha;
    int batchsize;
    int numepochs;
} options;
```

În câmpul alpha se memorează rata de învațare care se aplică în timpul antrenării. Prin intermediul valorii următorului câmp, batchsize, se specifică dimensiunea unui lot de antrenare. Un lot se obține din setul de hărți de intrare iar dimensiunea lotului reprezintă numărul de hărți de intrare pe care se efectuează o epocă de antrenare la un moment-dat. În cel de-al treilea camp, numepochs, se memorează numarul de epoci de antrenare care vor fi efectuate pentru fiecare set de hărți de intrare.

Prin intermediul structurii complexMatr se vor memora valorile complexe pe care se vor efectua majoritatea operațiilor presupuse de antrenarea rețelei.

```
typedef struct {
    cuDoubleComplex *mat;
    int msize, nsize;
} complexMatr;
```

Câmpul mat reprezintă vectorul unde se stochează valorile complexe. Acesta va fi interpretat însă ca fiind o matrice cu valori complexe. Această interpretare se face prin intermediul câmpurilor msize și nsize în care sunt memorate dimensiunile matricei. Se poate observa că valorile reale care compun fiecare număr complex, adică partea reală și partea imaginară a acestuia cât și rezultatul operațiilor care se efectuează pe aceste numere au precizie dublă, caracterisitcă dată de utilizarea tipului cuDoubleComplex.

Necesitatea formatării în acest mod a datelor din fiecare matrice complexă provine de la formatul zonelor de memorie care se vor aloca pe GPU. Acestea vor fi tot sub forma unor vectori interpretate ca matrici bidimensionale cunoscând dimensiunile acestora.

Utilizând această structurare a datelor, se definesc în continuare două tipuri compuse ce reprezintă interfațarea cu parametrii și valori intermediare interne ale rețelei. Mai precis aceasta se realizează prin definirea structurilor network și layer.

Structura ce poartă denumirea network va memora detalii referitoare la fiecare strat al rețelei și valori de interes folosite în antrenarea acesteia.

```
typedef struct{
    layer layers[N_LAYERS + 1];
    cuDoubleComplex *ffb;
    complexMatr ffw;
    complexMatr *fv;
    complexMatr *e;
    complexMatr *o;
    complexMatr *od;
    complexMatr *fvd;
    complexMatr dffw;
    complexMatr dffb;
    double L;
} network;
```

Vectorul layers de tip layer are ca scop stocarea datelor specifice fiecărui strat al rețelei. Prin urmare acesta se definește folosind elementul de configurare N_LAYERS care specifică numărul de straturi setat în configurare. În câmpul L se va memora și modifica după fiecare lot de intrare prelucrat eroarea obținută în timpul antrenării, adică diferența dintre rezultatul propus și rezultatul obținut prin prelucrarea datelor de intrare. Această eroare generală se calculează pornind de la vectorul în care se memorează erorile individuale pentru fiecare dintre ieșirile generate, și anume complexMatr *e. La rândul lui, acesta conține pe fiecare poziție diferența dintre valoarea rezultatului propus asociat poziției și valoarea generată pentru această ieșire a rețelei.

În câmpul ffw se va memora matricea de ponderi prin care se face interconectarea ultimului strat al rețelei cu rezultatele de ieșire asociate. Acesta de adaptează în funcție de dffw calculat. În mod asemănator se adaptează și setul de valori bias memorate in ffb, cu ajutorul dffb calculat.

Stocarea informațiilor particulare fiecărui strat se face într-un vector de tipul layer. Acesta este definit având următoarele câmpuri:

```
typedef struct{
    char type;
    int scale;
    int outputmaps;
```

```
    int kernelsize;
    complexMatr **a;
    complexMatr *k;
    cuDoubleComplex *b;
    complexMatr **d;
    complexMatr *dk;
    cuDoubleComplex *db;
} layer;
```

Detaliile specifice fiecărui strat sunt:

- Tipul stratului curent, specificat în câmpul type. Acesta poate lua valorile 's' sau 'c' dacă stratul este de tip sub-sampling sau de convoluție.
- Scalarea care se produce pe straturile de tip sample, memorată în scale.
- Numărul hărților de ieșire din strat care se stochează în outputmaps.
- Dimensiunea nucleului de convoluție, kernelsize. Acesta are valoare atașată doar daca tipul stratului este de convoluție.
- Hărțile de intrare memorate în câmpul complexMatr **a.
- Nucleele de convoluție memorate in k dacă tipul stratului este de convoluție.
- Vectorul de bias-uri și vectorul folosit pentru adaptarea acestor valori, b respectiv db.
- Vectorul de matrici *dk care se folosește pentru actualizarea valorilor din nucleele de convoluție.

4.2.1 Funcții și kernel-uri

Pentru o structurare ordonată și lizibilitate sporită, funcționalitatea soluției propuse este împărțită în mai multe module independente (funcții C). Aceste rutine specializate sunt definite pentru a fi utile într-o gamă cât mai largă de aplicații dar în primul rând pentru a fi folosite în cadrul acestei aplicații. Câteva dintre rutinele de bază și funcționalitatea acestora sunt descrise în cele ce urmează.

Operațiile pe matrici cu valori complexe sunt cele mai frecvente din cadrul antrenării rețelei neuronale. Prin urmare se definesc funcții C care efectuează aceste operații cu matrici de valori complexe. Operațiile de bază și cele mai des întâlnite sunt înmulțirea a două matrici și operația de convoluție.

Rezultatul înmulțirii a două matrici se obține prin apelarea funcției:

```
complexMatr mulMatr(complexMatr a, complexMatr b);
```

Valoarea returnată în caz de succes este rezultatul înmulțirii matricilor a si b. În caz de eroare programul își va încheia execuția cu un cod de eroare corespunzător, nefiind posibilă înaintarea în rezolvarea problemei. Funcția mulMatr nu efectuează înmulțirea propriu-zisă, ci pregătește datele și execută un kernel care va rula pe GPU. Mai precis, în acest pas se vor aloca zone de memorie din memoria procesorului grafic pentru cele doua matrici de intrare dar și pentru rezultat. Zona se alocă în funcție de dimensiunile celor doua matrici si prin deducerea dimensiunii matricei rezultat. De asemenea, tot în acest pas se vor transfera valorile complexe conținute în cele doua matrici în zona de memorie pusă la dispoziție de GPU. După ce s-a asigurat alocarea corectă a zonelor de memorie, este lansat kernel-ul asociat înmulțirii de matrici, și anume:

```
__global__ void multiplicationKernel(cuDoubleComplex* ad,
    cuDoubleComplex* bd, cuDoubleComplex* cd, int m1, int n1, int n2)
```

Pointerii ad, bd și cd indică spre zonele de memorie alocate pe dispozitivul grafic, iar ceilalți parametrii furnizează dimensiunile matricilor. Prin apelarea acestui kernel, calculul matricei rezultat se produce pe dispozitivul grafic, sarcina fiind împărțită între mai multe fire de execuție.

Alocarea memoriei pe dispozitivul grafic se face apelând funcția cudaMalloc. Aceasta returnează codul specific stării de succes sau în caz contrar eroarea care a provocat eșecul în alocare.Următorul apel este un exemplu de alocare și de verificare a stării returnate:

```
cudaStatus = cudaMalloc(&d_a,
    a.msize * a.nsize * sizeof(cuDoubleComplex));
if (cudaStatus != cudaSuccess) {...}
```

Transferul datelor între CPU si GPU se face prin apelarea funcției cudaMemcpy care primește ca parametrii sursa și destinația copierii de date cât și o constantă care specifică dacă se dorește copierea unei zone de memorie a procesorului grafic spre CPU sau dacă destinația este memoria GPU-ului. Un exemplu de apelare al acestei funcții este următorul:

```
cudaMemcpy(d_a, a.mat, a.msize * a.nsize *sizeof(cuDoubleComplex),
    cudaMemcpyHostToDevice);
```

prin care se copiază conținutul matricei a.mat spre o zonă de memorie din cadrul dispozitivului grafic spre a putea fi prelucrat.

Lansarea unui kernel de execuție pe dispozitivul grafic se face folosind sintaxa <<<...>>> prin care se specifică numărul și dimensiunea blocurilor de fire de execuție, ca în exemplul următor:

```
multiplicationKernel << <grid, block >> >(d_a,
    d_b, d_c, a.msize, a.nsize, result.nsize);
```

În mod asemănător operației de înmulțire a două matrici cu valori complexe, rezultatul operației de convoluție între două matrici se obține prin apelarea funcției:

```
complexMatr convolution(complexMatr a, complexMatr b);
```

Valoarea returnată în caz de succes este rezultatul convoluției matricilor a si b. În caz de eroare programul își va încheia execuția cu un cod de eroare corespunzător, nefiind posibilă înaintarea în rezolvarea problemei. Funcția convolution nu efectuează operația propriu-zisă, ci pregătește datele și execută un kernel care va rula pe GPU. Mai precis, în acest pas se vor aloca zone de memorie din memoria procesorului grafic pentru cele doua matrici de intrare dar și pentru rezultat. Zona se alocă în funcție de dimensiunile celor doua matrici si prin deducerea dimensiunii matricei rezultat. De asemenea, tot în acest pas se vor transfera valorile complexe conținute în cele doua matrici în zona de memorie pusă la dispoziție de GPU. După ce s-a asigurat alocarea corectă a zonelor de memorie, este lansat kernel-ul asociat operației de convoluție, și anume:

```
__global__ void convolutionKernel(cuDoubleComplex* ad,
    cuDoubleComplex* bd, cuDoubleComplex* cd, int ma, int na,
    int mb, int nb)
```

În același mod au fost adaptate pentru valori complexe cu dublă precizie o mulțime de alte funcționalități, în special pentru operații cu matrici complexe. Printre acestea se numără: transpusa unei matrici, rotirea la 180 de grade, suma și diferența a două matrici, bordarea cu zerouri a unei matrici și funcții pentru generarea matricilor cu valori pseudo-aleatoare utilizate pentru testare.

Pentru citirea datelor de intrare dintr-un fișier extern se utilizează funcția readInput-Matrices care va returna un tablou de structuri definite pentru memorarea de matrici cu valori complexe:

```
complexMatr* readInputMatrices(char *filename,
    long nbinputs, int msize, int nsize)
```

Această funcție va primi ca parametri la apelare numele fișierului în care sunt înregistrate valori reale normalizate extrase eventual din setul de imagini dat pentru antrenarea rețelei. În plus, se specifică numărul de matrici ce se dorește extras din fișiere dat prin paranetrul nbinputs și dimensiunea acestora dată de parametrii msize respectiv nsize. Fișierul poate fi structurat de exempu astfel încât să conțină pe fiecare linie msize*nsize elemente reprezentând câte o hartă de intrare pe care se va efectua antrenarea.

Valorile extrase în acest mod din fișiere externe vor fi memorate și stocate în structuri ce descriu numerele complexe, inițializându-se partea reală a acestora. Apelul funcției poate returna atât hărțile de intrare cât și ieșirile cu care se vor compara valorile rezultate în urma unei epoci a antrenării. Fiecare dintre acestea vor fi alocate în zona de lucru generală și se pot folosi în pașii de antrenare ca tipuri de matrici pentru operațiile implementate sau pentru a fi copiate în zona de memorie a dispozitivului grafic, urmând a fi procesate de acesta. Stocarea informațiilor din fișiere în structuri ce descriu numerele complexe se face prin apelul funcției:

```
make_cuDoubleComplex(re, im);
```

respectiv

```
make_cuFloatComplex(re, im);
```

în funcție de precizia datelor cu care se lucrează. În cazul unei erori întâmpinate la citirea fișierului extern, de exemplu inexistența acestuia sau lipsa drepturilor de scriere/citire din fișier, execuția programului va fi întreruptă cu un mesaj de eroare corespunzător.

Efectuarea operației de scădere între două matrici cu valori complexe se face prin apelul funcției:

```
complexMatr difMatr(complexMatr a, complexMatr b)
```

Unde parametrii de intrare sunt în ordinea apariției: matricea a reprezintă descăzutul iar al doilea parametru, matricea b reprezinta scăzătorul. Apelul va returna rezultatul operației dacă se execută cu succes și un mesaj de eroare în cazul în care dimensiunile matricilor date ca parametru nu sunt egale. Astfel operația de scădere nu poate fi efectuată iar execuția programului va fi întreruptă. Prin urmare, precondițiile acestei funcții sunt: existența în memorie a matricilor date ca parametri de intrare la apelare și potrivirea dimensiunilor acestora.Scăderea propriu-zisă a elementelor matricilor se face apelând funcția specializată din biblioteca cuComplex - cuCsub.

Pentru generarea unei matrici de dimensiuni oarecare conținând numere complexe pseudo-aleatoare se utilizează funcția:

```
complexMatr genRandK(int msize, int nsize)
```

Unde parametrii de intrare msize și nsize setează dimensiunile matricei care se va genera. Apelul va returna o referință către o matrice cu numere complexe generate aleator și se poate utiliza, de exemplu, pentru a inițializa nucleele de convoluție k în pregătirea parametrilor interni ai rețelei.

În mod asemănator, funcția complexMatr genZeroComplexMatrix(int msize, int nsize) se utilizează pentru construirea unei matrici cu numere complexe, în acest caz însă ea va fi inițializată cu zerouri. Se efectuează astfel alocarea matricei având dimensiunile msize, nsize în zona de memorie de lucru și setarea elementelor acesteia la zero.

Pentru extragerea unui subset din cadrul hărților de intrare pentru a putea fi procesat in mod individual se folosește funcția ajutătoare:

```
complexMatr* batchCopy(complexMatr *m, int b, int batchsize)
```

unde parametrii de intrare au următoarea semnificație: m reprezintă toate matricile corespunzătoare intrărilor din care se dorește extragerea unui subset, b reprezintă numărul setului și este în strânsă legătură cu batchsize care specifică dimensiunea setului extras. Astfel dacă se extrag pe rând un număr de subseturi având dimensiunea batchsize, prin

b se memorează practic poziția de început și poziția de sfârșit a subsetului din cadrul setului principal de date prin înmulțirea acestuia cu parametrul batchsize. Acest fapt este evidențiat prin declararea a doua variabile care indică poziția de început al subsetului din cadrul setului principal și respectiv ultima poziție:

```
int firstIndex = b * batchsize;
int lastIndex = (b + 1) * batchsize;
```

Elementele tabloului principal care se află în acest interval vor fi copiate în subset și returnat la apelul acestei funcții.

Efectuarea operației de adunare între două matrici cu valori complexe se face prin apelul funcției:

```
complexMatr sumMatr(complexMatr a, complexMatr b)
```

Unde parametrii de intrare sunt termenii operației de adunare. Apelul va returna rezultatul operației dacă se execută cu succes și un mesaj de eroare în cazul în care dimensiunile matricilor date ca parametru nu sunt egale. Astfel operația de adunare nu poate fi efectuată iar execuția programului va fi întreruptă. Prin urmare, precondițiile acestei funcții sunt: existența în memorie a matricilor date ca parametri de intrare la apelare și potrivirea dimensiunilor acestora. Adunarea propriu-zisă a elementelor matricilor se face apelând funcția specializată din biblioteca cuComplex:

```
__host__ __device__ static __ cuDoubleComplex
    cuCadd(cuDoubleComplex x, cuDoubleComplex y)
```

Aplicarea funcției sigmoide asupra elementelor unei matrici având valori complexe se face prin apelarea funcției ajutătoare:

```
complexMatr sigm(complexMatr a)
```

Unde parametrul de intrare este matricea asupra căreia se dorește aplicarea funcției iar valoarea returnată este o nouă matrice conținând rezultatele aplicării.

Atunci când antrenarea este în stagiul de feed-forward și stratul curent este un strat de tip sample atunci se utilizează funcția downSample pentru a genera o hartă formată din caracteristici esențiale ale hărților de intrare pe stratul respectiv:

```
complexMatr downSample(complexMatr a)
```

Apelul funcției va returna o matrice formată din mostre ale matricei a primite ca parametru de intrare. Rezultatul va avea dimensiunile înjumătățite ale matricei a și astfel se păstrează acele caracteristici ale hărților de intrare care sunt cele mai accentuate. Pentru realizarea acestui aspect s-a considerat ca fiind element al matricei rezultat, valoarea maximă a unui grup de patru elemente din matricea dată ca parametru de intrare. Compararea elementelor se face utilizând funcția de bibliotecă cuCabs care va calcula modulului numărului complex primit ca parametru.

Operația inversă celei efectuate de funcția downSample se întâlnește în stagiul de backpropagation și se face prin apelul funcției ajutătoare upSample, având antetul:

```
complexMatr upSample(complexMatr a)
```

Unde matricea returnată va avea dimensiuni duble față de matricea primită ca parametru de intrare, iar fiecare grup de câte patru elemente se determină prin multiplicarea a câte un element din matricea a.

Calculul transpusei unei matrici cu valori complexe se efectuează apelând funcția ajutătoare transpose care are următorul antet:

```
complexMatr transpose(complexMatr m)
```

Unde matricea returnată se formează prin transpunerea elementelor matricei date ca parametru, m. Rezultatul va avea aceleași dimensiuni setate de matricea m.

O altă operațiune de interes folosită cu scopul de a pregăti o matrice înaintea aplicării operației de convoluție este bordarea cu zerouri. Aceasta se efectuează prin apelarea funcției:

```
complexMatr border(complexMatr m, int size)
```

care primește ca parametru de intrare matricea m la care se dorește bordarea și o valoare size reprezentând numărul de linii și coloane de zerouri care se vor borda la m. Dimensiunea matricei returnate va fi egală cu suma dintre dimensiunea matricei primite ca parametru m la care se adaugă un număr de size linii și size coloane.

Funcția complexMatr clone(complexMatr m) crează o nouă matrice având elementele identice cu cele ale matricei primite ca parametru, m. Utilitatea acestei funcții constă în crearea unor structuri de date eventual cu caracter temporar asupra cărora se pot produce modificări fără a afecta elementele matricei de la care s-a plecat. Prin apelarea clone se va aloca în memoria de lucru o altă matrice care se poate șterge după utilizare fără a afectarea elementelor lui m.

Atunci când pentru a efectua o anumită operație pe două matrici(de exemplu convoluția) se dorește aplicarea unei rotiri la una din matricile operanzi, această aplicare se face prin apelul funcției:

```
complexMatr rot180(complexMatr m)
```

unde matricea returnată reprezintă rezultatul rotirii matricei date ca parametru de intrare, m. Noua matrice creată se va șterge dupa utlizare pentru a nu ocupa inutil spațiul memoriei de lucru iar utlitatea constă în faptul ca nu sunt afectate elementele primite ca parametru.

Capitolul 5

Rezultate experimentale

Am testat reţeaua neuronală de convoluţie cu valori complexe pe celebra bază de date MNIST, care conţine imaginile unor cifre scrise de mână în format alb-negru [25]. Această bază de date a fost construită de către autorii articolului [25] pornind de la bazele de date Special Database 1 şi Special Database 3 colectate de către NIST (National Institute of Standards and Technology) din Statele Unite ale Americii. SD-1 conţinea 58537 de imagini ale unor cifre scrise de mână de către 500 de scriitori diferiţi, care au fost împărţiţi 250 în setul de antrenare şi 250 în setul de testare. Setul de antrenare a fost completat cu imagini din SD-3, pentru a forma un set de 60000 de tipare de antrenare. În mod similar, setul de testare a fost completat cu tipare până la numărul de 60000, de asemenea. Însă, s-a hotărât să se folosească doar 10000 de tipare de testare (5000 din SD-1 şi 5000 din SD-3), dând astfel naştere bazei de date Modified NIST sau MNIST, care a devenit azi standardul în ceea ce priveşte testarea diferitelor arhitecturi şi algoritmi de învăţare din domeniul reţelelor neuronale şi al învăţării automate în general.

Imaginile sunt în format nuanţe de gri, şi au 28 de pixeli înălţime şi 28 de pixeli lăţime. Exemple de tipare de antrenare din această bază de date se pot vedea în Figura 5.1.

În experimentele noastre, am lucrat direct pe baza de date cu cele 60000 de tipare de antrenare şi 60000 de tipare de testare, fără a augmenta această bază de date cu deformări ale imaginilor, care s-au dovedit în literatură a îmbunătăţi performanţa reţelelor neuronale.

Pentru o comparaţie onestă, am antrenat o reţea neuronală de convoluţie cu valori reale care are aceeaşi parametri şi aceeaşi arhitectură cu reţeaua neuronală de convoluţie cu valori complexe. Trebuie precizat faptul că, în experimentele noastre, datele au fost aceleaşi pentru ambele reţele, fără a încerca transformarea imaginilor în domeniul complex, folosind, de exemplu transformata Fourier. Acest tip de experimente se află printre ideile viitoare de extindere a prezentului proiect.

Astfel, setul de antrenare a fost împărţit în pachete de 25 de imagini, care au fost trecute prin reţea, într-o subiteraţie a unei epoci. Acest algoritm de învăţare poartă denumirea de metoda gradient stochastică, deoarece nu foloseşte toate datele disponibile în fiecare iteraţie, ci doar o populaţie statistică a acestora. Rata de învăţare pentru acest algoritm a fost aleasă ca fiind 0.1, aceasta rămânâd constantă pe tot parcursul învăţării.

Tehnici ca învăţarea folosind moment sau scăderea cu un factor a ratei de învăţare nu au fost folosite, fiind de asemenea idei de încercat pe viitor.

Figura 5.1: Exemplu de imagini din baza de date MNIST [25]

Arhitectura reţelor pentru care s-au obţinut cele mai bune rezultate este după cum urmează:

- un strat de intrare cu un singur canal, de dimensiune 28×28
- un strat de convoluţie cu 6 canale, de dimensiune 24×24, pentru care dimensiunea nucleurilor de convoluţie este de 5×5
- un strat de pooling cu 6 canale, de dimensiune 12×12, dimensiunea vecinătăţii pe care se face medierea fiind 2
- un strat de convoluţie cu 12 canale, de dimensiune 8×8, pentru care dimensiunea nucleurilor de convoluţie este de 5×5
- un strat de pooling cu 12 canale, de dimensiune 4×4, dimensiunea vecinătăţii pe care se face medierea fiind 2
- un strat total conectat cu 192 de neuroni ($12 \times 4 \times 4 = 192$)
- un strat de ieşire cu 10 neuroni, câte unul pentru fiecare cifră de la 0 la 9.

Funcţia de activare cu cele mai bune performanţe pentru straturile de convoluţie şi stratul total conectat este funcţia sigmoid

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

pentru reţelele cu valori reale, respectiv funcţia sigmoid complexă pe componente (split complex)

$$f(x + iy) = \frac{1}{1 + e^{-x}} + i\frac{1}{1 + e^{-y}},$$

pentru reţelele cu valori complexe. Iniţializarea ponderilor s-a făcut în intervalul

$$\left[-\sqrt{\frac{6}{fan\_in + fan\_out}}, \sqrt{\frac{6}{fan\_in + fan\_out}}\right],$$

care este recomandată a fi folosită pentru funcţia sigmoid, unde fan_in reprezintă numărul de conexiuni care intră într-un strat, şi fan_out numărul de conexiuni care ies dintr-un strat. Antrenarea a fost realizată timp de 50 de epoci.

Procesul de antrenare s-a făcut pe un calculator cu procesor Intel Core 2 Quad de 2.24 GHz, 4 GB de memorie RAM şi placă video NVIDIA Quadro NVS 290. Aceasta dispune de (doar) 16 nuclee CUDA şi o frecvenţă a procesorului grafic de 460 MHz. Datorită acestui fapt, antrenarea a durat aproximativ patru zile. Pe viitor, în măsura în care va fi disponibil un calculator cu o placă video superioară ca performanţe, se vor putea face mai multe experimente şi cu arhitecturi mai complicate de reţele neuronale.

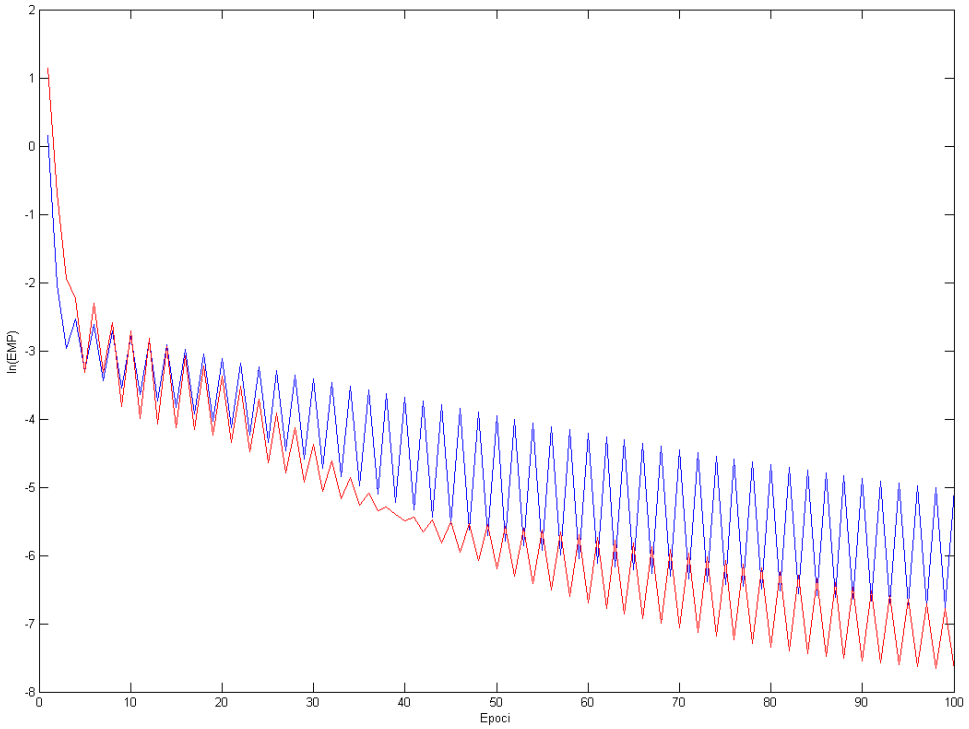

Figura 5.2: Dinamica învăţării

Dinamica învăţării este prezentată în Figura 5.2, în care cu albastru apare logaritm natural din eroarea medie pătratică a reţelei cu valori reale, şi cu roşu apare logaritm

natural din eroarea medie pătratică a reţelei cu valori complexe, măsurată la fiecare jumătate de epocă, în decursul celor 50 de epoci, o epocă fiind considerată după ce toate tiparele de antrenare au fost prezentate reţelei.

În aceste condiţii, eroarea pentru setul de testare conţinând 10000 de imagini a fost de **1.22**%, adică **122** de imagini greşit clasificate, pentru reţeaua cu valori reale, respectiv de **0.9**%, adică **90** de imagini greşit clasificate pentru reţeaua cu valori complexe. Având în vedere că state of the art-ul la ora actuală pe această bază de date este **0.23**%, obţinut cu ansamble de reţele de convoluţie foarte adânci şi cu augmentarea cu până la 10 ori a setului de antrenare folosind deformări, cu până la 11 straturi efective (13 straturi în total), reţelele neuronale cu valori complexe au obţinut rezultate promiţătoare, care vor trebui extinse pe viitor şi pentru acest tip de reţele foarte adânci, pentru a vedea care este comportamentul reţelor cu valori complexe în aceste cazuri.

O altă observaţie care se poate face este aceea că reţelele cu valori reale şi cu valori complexe „greşesc" în locuri diferite, ceea ce ar putea sugera că un ansamblu hibrid, format atât din reţele cu valori reale, cât şi din reţele cu valori complexe, ar putea avea rezulate mai bune decât ambele tipuri de reţele omogene. Astfel, se constată că cele două reţele clasifică ambele greşit doar **62** de imagini, ceea ce înseamnă o eroare de **0.62**%, o valoare foarte apropiată de state of the art, mai ales în condiţiile în care cele două reţele au o arhitectură relativ simplă.

Capitolul 6

Concluzii

Implementarea rețelei neuronale de convoluție cu valori complexe folosind capabilitatea de calculul paralel a procesoarelor grafice a făcut posibilă efectuarea unei comparații între o rețea neuronală cu valori complexe și o rețea neuronală cu valori reale. În plus, aplicarea convoluției având cele mai bune rezultate la ora actuală în domeniul procesării și a recunoașterii de imagini, a făcut posibilă o accelerare a antrenării rețelei neuronale. Utilizarea procesoarelor grafice pentru calcul paralel a rezultat în reducerea timpilor de antrenare.

Astfel se poate evita pierderea unor informații esențiale ale imaginilor procesate în cazul în care acestea sunt date de către instrumentele de măsurare direct în formă complexă, mulțumită dependențelor care pot apărea în domeniul complex.

Experimente pe cunoscuta bază de date MNIST au arătat că eroarea unei reţele neuronale de convoluţie cu valori complexe este de 0.9% pe când cea a unei reţele de convoluţie cu valori reale cu arhitectură identică şi antrenată în acelaşi fel, a fost de 1.22%. S-a constatat, de asemenea că eroarea „comună" a celor două reţele este de doar 0.62%, ceea ce înseamnă că modele hibride vor putea avea performanţe care să depăşească perfomanţele individuale ale reţelelor cu valori reale şi ale reţelor cu valori complexe, luate separat.

În ceea ce privește posibilele direcții de evoluție viitoare, se poate în primul rând experimenta prin modificări ale arhitecturii rețelei. Pentru testele curente s-a folosit o arhitectură relativ simplă din lipsa unui dispozitiv GPU performant dintr-o generație mai recentă. O altă optimizare ar putea fi considerată implementând tehnici de învățare mai performante cum ar fi învățarea folosind moment sau scăderea cu un factor a ratei de învățare. De asemenea o idee care poate fi aplicată în viitor o reprezintă augmentarea bazei de date pe care s-a testat rețeaua cu deformări ale imaginilor. În literatura de specialitate s-a dovedit că această acțiune sporește performanțele rețelelor neuronale. Un alt tip de idee în ceea ce privește experimentul este transformarea imaginilor în domeniul complex, folosind de exemplu transformata Fourier pentru a realiza acest lucru.

Bibliografie

[1] N.N. Aizenberg, Y.L. Ivaskiv, and D.A. Pospelov. A certain generalization of threshould functions. *Dokrady Akademii Nauk SSSR*, 196:1287 – 1290, 1971.

[2] E. Angiuli, F. Del Frate, B. Polsinelli, and D. Solimini. Towards complex-valued neural algorithms for forest parameters estimation from polinsar data. In *IEEE International Geoscience and Remote Sensing Symposium, IGARSS*, volume II, pages 641 – 644. IEEE, July 2008.

[3] M. Arjovsky, A. Shah, and Y. Bengio. Unitary evolution recurrent neural networks. In *International Conference on Learning Representations*, 2016.

[4] P. Baldi and Z. Lu. Complex-valued autoencoders. *Neural Networks*, 33:136 – 147, 2012.

[5] Y. Bengio. Learning deep architectures for ai. *Foundations and Trends in Machine Learning*, 2(1):1 – 127, 2009.

[6] Y. Bengio, Y. LeCun, and G. Hinton. Deep learning. *Nature*, 521:436 – 444, 2015.

[7] N. Benvenuto and F. Piazza. On the complex backpropagation algorithm. *IEEE Transactions on Signal Processing*, 40(4):967 – 969, April 1992.

[8] J. Bruna and S. Mallat. Invariant scattering convolution networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1872 – 1886, 2013.

[9] A. Chaturvedi, R. Sharma, D. Wadekar, A. Bhandwalkar, and S. Shitole. Adaptive parametric estimator for complex valued images. In *International Conference on Technologies for Sustainable Development (ICTSD)*, 2015.

[10] NVIDIA Corporation. cuBLAS Documentation, http://docs.nvidia.com/cuda/cublas/, 2015.

[11] NVIDIA Corporation. CUDA C Programming Guide, http://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2015.

[12] P. Eichel and R.W. Ives. Compression of complex-valued sar images. *IEEE Transactions on Image Processing*, 8(10):1483 – 1487, 1999.

[13] G.M. Georgiou and C. Koutsougeras. Complex domain backpropagation. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 39(5):330 – 334, May 1992.

[14] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[15] R. Hansch and O. Hellwich. Classification of polarimetric sar data by complex valued neural networks. In *ISPRS Workshop*, 2009.

[16] R. Hansch and O. Hellwich. Complex-valued convolutional neural networks for object detection in polsar data. In *European Conference on Synthetic Aperture Radar (EUSAR)*, 2010.

[17] T. Hara and A. Hirose. Plastic mine detecting radar system using complex-valued self-organizing map that deals with multiple-frequency interferometric images. *Neural Networks*, 17(8 - 9):1201 – 1210, October - November 2004.

[18] L. Hernandez-Garcia, A.L. Vazquez, and D.B. Rowe. Complex-valued analysis of arterial spin labeling based fmri signals. *Magnetic Resonance in Medicine*, 62(6):1597 – 1608, 2009.

[19] A. Hirose. Continuous complex-valued back-propagation learning. *Electronics Letters*, 28(20):1854 – 1855, September 1992.

[20] A. Hirose. *Complex-Valued Neural Networks: Advances and Applications*. John Wiley & Sons, Inc., 2013.

[21] Y. Hui and M.R. Smith. Mri reconstruction from truncated data using a complex domain backpropagation neural network. In *Pacific Rim Conference on Communications, Computers, and Signal Processing (PACRIM)*, 1995.

[22] V. Laparra, M.U. Gutmann, J. Malo, and A. Hyvarinen. Complex-valued independent component analysis of natural images. In *International Conference on Artificial Neural Networks (ICANN)*, 2011.

[23] Y. LeCun and Y. Bengio. *Convolutional networks for images, speech, and time series*, pages 255 – 258. MIT Press, 1995.

[24] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in Neural Information Processing Systems (NIPS)*, 1989.

[25] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278 – 2324, 1998.

[26] Y. LeCun, K. Kavukcuoglu, and C. Farabet. Convolutional networks and applications in vision. In *International Symposium on Circuits and Systems (ISCAS)*, 2010.

[27] H. Leung and S. Haykin. The complex backpropagation algorithm. *IEEE Transactions on Signal Processing*, 39(9):2101 – 2104, September 1991.

[28] H. Li, N.M. Correa, P.A. Rodriguez, V.D. Calhoun, and T. Adali. Application of independent component analysis with adaptive density model to complex-valued fmri data. *IEEE Transactions on Biomedical Engineering*, 58(10):2794 – 2803, 2011.

[29] S. Masuyama, K. Yasuda, and A. Hirose. Multiple-mode selection of walled-ltsa array elements for high-resolution imaging to visualize antipersonnel plastic landmines. *IEEE Geoscience and Remote Sensing Letters*, 5(4):745 – 749, October 2008.

[30] T. Nitta. Solving the xor problem and the detection of symmetry using a single complex-valued neuron. *Neural Networks*, 16(8):1101 – 1105, October 2003.

[31] A. Pande and V. Goel. Complex-valued neural network in image recognition a study on the effectiveness of radial basis function. *World Academy of Science, Engineering and Technology*, 20:220 – 225, 2007.

[32] D.P. Reichert and T. Serre. Neuronal synchrony in complex-valued deep networks. In *International Conference on Learning Representations*, 2014.

[33] P.A. Rodriguez, N.M. Correa, T. Adali, and V.D. Calhoun. Quality map thresholding for de-noising of complex-valued fmri data and its application to ica of fmri. In *International Workshop on Machine Learning for Signal Processing*, 2009.

[34] S. Samadi, M. Cetin, and M.A. Masnadi-Shirazi. Sparse signal representation for complex-valued imaging. In *Digital Signal Processing Workshop and Signal Processing Education Workshop (DSP/SPE)*, pages 365 – 370, 2009.

[35] A.M. Sarroff, V. Shepardson, and M.A. Casey. Learning representations using complex-valued nets. In *International Conference on Learning Representations*, 2016.

[36] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85 – 117, 2015.

[37] A.B. Suksmono and A. Hirose. Interferometric sar image restoration using monte carlo metropolis method. *IEEE Transactions on Signal Processing*, 50(2):290 – 298, 2002.

[38] M. Tygert, J. Bruna, S. Chintala, Y. LeCun, S. Piantino, and A. Szlam. A mathematical motivation for complex-valued convolutional networks. *Neural Computation*, 28(5):815 – 825, May 2016.

[39] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K.J. Lang. Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(3):328 – 339, 1989.

[40] B. Widrow, J. McCool, and M. Ball. The complex lms algorithm. *Proceedings of the IEEE*, 63(4):719 – 720, April 1975.

[41] C.-C. Yang and N.K. Bose. Landmine detection and classification with complex-valued hybrid neural network using scattering parameters dataset. *IEEE Transactions on Neural Networks*, 16(3):743 – 753, May 2005.

[42] D. Yu and L. Deng. *Automatic Speech Recognition: A Deep Learning Approach*. Springer, 2015.

[43] M.-C. Yu, Q.-H. Lina, L.-D. Kuang, X.-F. Gong, F. Cong, and V.D. Calhoun. Ica of full complex-valued fmri data using phase information of spatial maps. *Journal of Neuroscience Methods*, 249:75 – 91, 2015.