Universitatea “Politehnica” din București
Facultatea de Electronică, Telecomunicații și Tehnologia Informației

Sistem de urmărire a mâinii în timp real folosind rețele neurale

Proiect de diplomă

prezentat ca cerință parțială pentru obținerea titlului de
Inginer în domeniul *Electronică şi Telecomunicaţii*
programul de studii de licență *Electronică Aplicată (ETC – ELA)*

Conducători științifici
Prof. dr. ing. Corneliu BURILEANU
Ana-Antonia NEACȘU

Absolvent
Gabriela-Loredana GHENEA

Anul 2019

Universitatea "Politehnica" din Bucureşti
Facultatea de Electronică, Telecomunicaţii şi Tehnologia Informaţiei
Departamentul **EAII**

Anexa 1

TEMA PROIECTULUI DE DIPLOMĂ
a studentului **GHENEA I. Gabriela-Loredana , 444B**

1. Titlul temei: Sistem de urmărire a mâinii în timp real folosind reţele neurale

2. Descrierea contribuţiei originale a studentului (în afara părţii de documentare) şi specificaţii de proiectare:
Proiectul de faţă constă în dezvoltarea unei aplicaţii ce detectează mâna utilizatorului dintr-un flux video live captat cu ajutorul unei camere. Dezvoltarea proiectului presupune mai multe etape:
- achiziţia de date - captarea unui flux video cu ajutorul unei camere de rezoluţie 1280x720. Din acest flux vom păstra un număr suficiet de cadre pentru a scoate trăsăturile relevante. Aceste trăsături vor reprezenta parametrii de intrare într-o reţea neurală.
- crearea reţelei neurale – aceasta va prelua parametrii extraşi din fluxul video şi va decide dacă există sau nu o mână în imagine. Dacă este detectată prezenţa mâinii, atunci pe ecran va apărea un pătrat care va indica poziţia acesteia în timp real. Pentru dezvoltarea reţelei vor fi abordate mai multe arhitecturi, urmând să fie aleasă cea mai bună dintre ele.
- antrenare şi testare – pentru antrenarea reţelei neurale vom utiliza o bază de date publică (precum Egohands Dataset), ulterior testând funcţionalitatea pe fluxul preluat în timp real de la camera video

3. Resurse folosite la dezvoltarea proiectului:
limbajul de programare Python, biblioteci precum tensorflow, numpy, opencv

4. Proiectul se bazează pe cunoştinţe dobîndite în principal la următoarele 3-4 discipline:
Programare obiect-orientata, Prelucrarea digitala a semnalelor, Microcontrolere, Decizie si Estimare

5. Proprietatea intelectuală asupra proiectului aparţine: U.P.B.

6. Data înregistrării temei: 2018-11-30 18:12:27

Conducător(i) lucrare,
Prof. dr. ing. Corneliu BURILEANU

semnătura:.....

NEACŞU Ana-Antonia

semnătura: .....

Student,

semnătura:.....

Director departament,
Prof. dr. ing Sever PAŞCA

semnătura:.....

Decan,
Prof. dr. ing. Cristian NEGRESCU

semnătura:.....

Cod Validare: **a1fef266cc**

Declarație de onestitate academică

Prin prezenta declar că lucrarea cu titlul "*Sistem de urmărire a mâinii în timp real folosind rețele neurale*", prezentată în cadrul Facultății de Electronică, Telecomunicații și Tehnologia Informației a Universității "Politehnica" din București ca cerință parțială pentru obținerea titlului de *Inginer* în domeniul *Electronică și Telecomunicații,* programul de studii *Electronică Aplicată (ETC – ELA)* este scrisă de mine și nu a mai fost prezentată niciodată la o facultate sau instituție de învățământ superior din țară sau străinătate.

Declar că toate sursele utilizate, inclusiv cele de pe Internet, sunt indicate în lucrare, ca referințe bibliografice. Fragmentele de text din alte surse, reproduse exact, chiar și în traducere proprie din altă limbă, sunt scrise între ghilimele și fac referință la sursă. Reformularea în cuvinte proprii a textelor scrise de către alți autori face referință la sursă. Înțeleg că plagiatul constituie infracțiune și se sancționează conform legilor în vigoare.

Declar că toate rezultatele simulărilor, experimentelor și măsurătorilor pe care le prezint ca fiind făcute de mine, precum și metodele prin care au fost obținute, sunt reale și provin din respectivele simulări, experimente și măsurători. Înțeleg că falsificarea datelor și rezultatelor constituie fraudă și se sancționează conform regulamentelor în vigoare.

București, *Iunie, 2019*

Absolvent *Gabriela-Loredana GHENEA*

(semnătura în original)

CUPRINS

Lista figurilor

Lista tabelelor

Lista acronimelor

ANN – Artificial neural network

BKP - Backpropagation

CNN – Convolutional Neural Network

DNN – Deep Neural Network

FPS – Frames Per Second

GD – Gradient Descent

IOU – Intersection over Union

ReLU – Rectified Linear Unit

SSD – Single Shot MultiBox Detector

Introducere

Motivație și obiective

Evoluția tehnologică din ultimii ani a avut un impact major asupra tuturor domeniilor de activitate. Începând de la domeniul medical până la aplicații de divertisment, orice aparat electronic conține un microcontroller programat să îndeplinească o anumită sarcină. Algoritmii sunt într-o continuă dezvoltare, sunt mai performanți, iar limbajele de programare noi apărute sunt optimizate pentru a rezolva probleme mai dificile într-un timp mult mai scurt.

Toate aceste eforturi duc într-o singură direcție, și anume cea de a construi un calculator cel puțin la fel de performant precum cel mai inteligent sistem actual: creierul uman. Calculatorul dispune de memorie mare și rapidă, de putere de calcul, însă îi lipsește un atribut extrem de important, și anume gradul de generalizare. Un algoritm este capabil să rezolve o sarcină într-un timp scurt așa cum a fost programat, însă va eșua în cazul în care este pus într-o situație neprevăzută în codul scris de programator. Tehnologia actuală își dorește să obțină această capacitate de reacție în situații pe care programatorul nu le poate preveni, iar acest obiectiv poate fi îndeplinit cu succes cu ajutorul rețelelor neurale.

Rețelele neurale au un mare succes în domeniul științific actual deoarece acestea reprezintă o modelare a modului de funcționare a creierului uman. Ele sunt alcătuite din unități funcționale numite neuroni care sunt conectați între ei, legăturile fiind asemănate cu sinapsele. Inovația acestor structuri constă în principal în procesul de învățare. Acest proces de învățare este complet diferit de procesul de memorare asociat cu diverși algoritmi dezvoltați cu alte structuri. Scopul acestei etape nu este ca sistemul să învețe o tabelă de asociere, ci să obțină o capacitate de generalizare. Domeniile de aplicabilitate ale rețelelor neurale sunt foarte extinse, însă cel mai de interes este cel de procesare al imaginilor. În cazul unor fotografii este foarte dificil să prezicem factori externi neprevăzuți, cum ar fi condițiile de iluminare. Un obiect poate apărea într-o imagine în numeroase poziții și culori, de aceea algoritmii de prelucrare de imagini sunt de o complexitate cu atât mai mare cu cât sarcina se dorește a fi mai precisă.

În cadrul acestui proiect ne-am propus să realizăm un sistem care să fie capabil să detecteze poziția unei mâini dintr-un flux video. Detecția mâinii este o sarcină dificilă, deoarece mâna umană este complexă, diferă de la individ la individ atât ca dimensiune, cât și ca pigmentație a pielii. Dezvoltarea unui algoritm eficient de detecție și urmărire a mâinii poate să pună numeroase probleme, dat fiind faptul că niciodată nu vom avea 2 mâini identice. De aceea, rețelele neurale par a fi cea mai bună soluție pentru detecția mâinilor. Ne propunem să analizăm diverse structuri de rețele neurale pentru a o alege pe cea potrivită scopului nostru, ulterior să realizăm antrenarea rețelei și testarea acesteia cu ajutorul camerei web.

Aplicabilitate

Acest proiect are aplicabilitate în mai multe domenii, cel mai interesant fiind reprezentat de aplicațiile interactive, atât pe calculator cât și pe alte dispositive inteligente cum ar fi telefoane mobile, televizoare, tablete sau console de jocuri. Un sistem de urmărire a mâinii poate să furnizeze informații importante ce pot fi procesate pentru controlul altor sisteme.

Spre exemplu, o aplicație ar fi cea de control într-un joc video pe calculator doar cu ajutor mâinii. Având la dispoziție o cameră video, sistemul va detecta în timp real poziția mâinii și va fi transmisă mai departe. În funcție de aceasta, calculatorul va lua o decizie, cum ar fi: mută personajul la dreapta/stânga sau realizează o săritură. De asemenea, o altă aplicație ar fi simplul control al

cursorului pe ecran doar cu ajutorul mâinii. Astfel, calculatoarele nu mai au nevoie de echipamente suplimentare cum ar fi mouse-ul sau touchpad-ul.

O altă arie foarte interesantă este cea a telefoanelor mobile inteligente. Acestea au avut o evoluție surprinzătoare în ultimul timp prin integrarea unor tehnologii extrem de performante în dispozitive cu dimensiuni din ce în ce mai mici. Acestea nu numai că dețin o cameră foto, ci majoritatea telefoanelor dețin cel puțin două, iar acestea oferă imagini de o calitate superioară. Această cameră poate fi folosită, spre exemplu, pentru detecția mâinii și declanșarea unui temporizator pentru fotografiere.

State of the Art

Până acum câțiva ani, toate aplicațiile de detecție sau clasificare erau realizate pe baza unor algoritmi. Majoritatea aplicațiilor bazându-se pe imagini, algoritmii presupun mai multe prelucrări ce consumă timp și resurse. Printre prelucrările de bază ce se aplică într-un algoritm de detecție enumerăm detecția de culoare a pielii, folosirea informațiilor de adâncime, extragere de contur sau extragere de fundal. Cel mai mare inconvenient în acest caz îl reprezintă condițiile externe, care trebuie să fie fixe.

Un sistem inteligent nu este suficient să memoreze o tabelă de asocieri, ci acesta poate fi considerat inteligent abia în momentul în care este capabil să facă singur generalizări și aproximări în situații necunoscute. Au fost dezvoltate numeroase aplicații cu rețele neurale convolutionale (CNN), acestea fiind specializate pe imagini. Printre primele aplicații ale CNN-urilor a fost cea de recunoaștere a cifrelor scrise de mână [1]. După aceasta au urmat numeroase aplicații în direcția recunoașterii de obiecte, una dintre cele mai remarcabile fiind cea de detectare a mâinilor și recunoaștere a activităților în interacțiuni egocentrice complexe [2]. Această aplicație este o demonstrație impresionantă a detecției de mâini, dar și a segmentării acesteia. De asemenea, s-a demonstrat că activitățile realizate de o persoană pot fi recunoscute cu succes din simpla detecție a poziției mâinii.

Aplicația de detecție a mâinii și estimare a poziției a mai fost abordată ulterior în diferite forme, însă cea dezvoltată de cei de la Indiana University este de departe cea mai complexă. Un rezultat al acestui proiect îl reprezintă și o bază de date vastă, ce conține imagini etichetate cu mâini, utilizată adesea de cercetători, deoarece oferă rezultate satisfăcătoare.

Structura

Capitolul 1 reprezintă o introducere teoretică în cadrul căreia vom prezenta noțiuni fundamentale ce stau la baza rețelelor neurale, cum ar fi neuron artificial, rețea neurală sau funcție de activare.

În capitolul 2 vom detalia ipoteza acestui proiect și vor fi prezentate informații generale despre baza de date și despre modul de prelucrare al acesteia. Ulterior se vor detalia pașii de implementare ai proiectului.

Capitolul 3 prezintă rezultatele experimentale pe care le-am obținut și observații făcute pe baza acestora.

În capitolul 4 vom face concluzii, vom evidenția contribuțiile personale, dar și optimizările viitoare.

Capitolul 1. Noțiuni teoretice

Generalități

Pentru a înțelege cum funcționează rețelele neurale trebuie să facem o analiză asupra a ceea ce se întâmplă în creierul uman. După ce înțelegem cum funcționează neuronii și cum sunt aceștia conectați între ei vom putea să analizăm modul de funcționare al neuronilor artificiali.

Analogie biologică

Creierul uman este alcătuit din aproximativ 100 miliarde de celule nervoase sau neuroni. Putem vedea structura unui neuron în figura 1.1. Neuronii comunică prin semnale electrice reprezentate de impulsuri de scurtă durată în tensiunea peretelui celular sau a membranei. Legăturile interneuroni sunt mediate de joncțiuni electrochimice numite sinapse, care se află pe ramurile celulei și sunt numite dendrite. Fiecare neuron are de obicei mii de conexiuni cu alți neuroni și, prin urmare, primește în mod constant o multitudine de semnale care intră în corpul celular. Aici, ele sunt integrate sau însumate împreună într-un anumit fel și, dacă semnalul rezultat depășește un anumit prag atunci neuronul se va „activa" și va genera un impuls de tensiune ca răspuns. Acesta este apoi transmis altor neuroni printr-o fibră ramificată numită axon. Pentru a determina dacă un impuls ar trebui să fie produs sau nu, unele semnale de intrare produc un efect inhibitor și tind să prevină activarea, în timp ce altele au efect excitator și ajută la generarea impulsurilor. Capacitatea diferită de procesare a fiecărui neuron constă în tipul lor (excitator sau inhibitor) și în puterea conexiunilor sinaptice cu alți neuroni. [3]

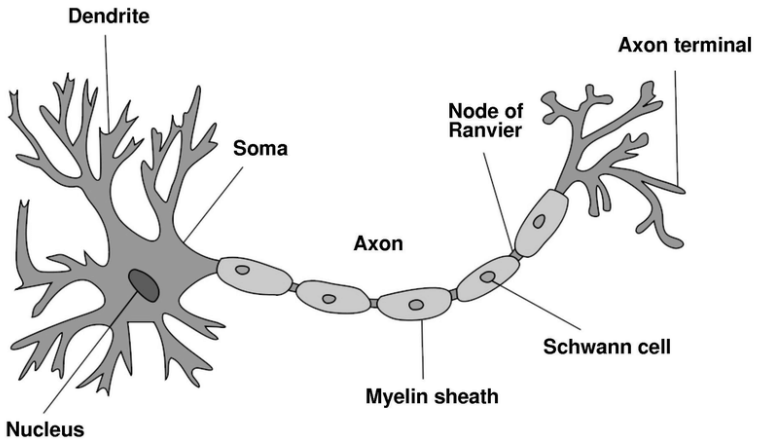


Figura 1.1. Structura unui neuron (sursa: [4])

Cele patru elemente principale: dendritele, sinapsele, corpul celular și axonul, alcătuiesc structura care este adoptată și de modelul neuronului artificial. Acesta va avea intrări, un corp celular și o ieșire. Sinapsele vor fi simulate prin punctele de contact dintre corpul celular și conexiunile de intrare/ieșire. Fiecărei sinapse îi va fi asociată o pondere. Aceste ponderi sunt echivalente cu tăria sinaptică. [5]

Modele fundamentale de neuroni artificiali

Modelul neuronal McCulloch-Pitts

Prima definiție a unui model de neuron bazat pe considerațiile simplificate ale modelului biologic a fost formulată de McCulloch și Pitts (1943). Modelul McCulloch-Pitts al neuronului este prezentat în figura 1.2. Intrările xi, i = 1, 2,. . . , n sunt 0 sau 1, în funcție de absența sau prezența

impulsului de intrare la momentul k. Semnalul de ieșire al neuronului este notat cu o. Regula de activare a acestui model este definită după cum urmează:

$$\sigma^{k+1} = \begin{cases} 1 \; dacă \sum_{i=1}^{n} w_i x_i^k \geq T \\ 0 \; dacă \sum_{i=1}^{n} w_i x_i^k < T \end{cases}$$

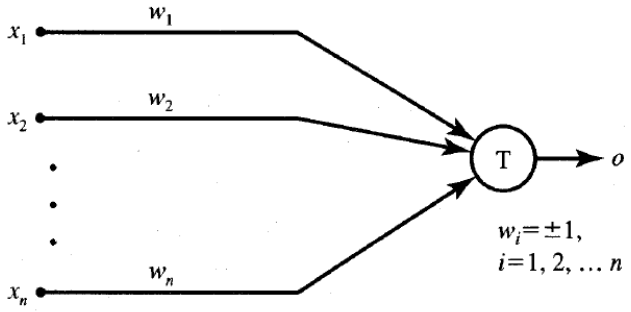


Figura 1.2. Modelul neuronal McCulloch-Pitts (sursa: [6])

Trebuie să observăm faptul că, pentru acest model, wi = + 1 pentru sinapsele excitatorii, wi = -1 pentru sinapsele inhibitorii, iar T este valoarea pragului neuronului, care trebuie să fie depășită de suma ponderată a semnalelor pentru ca neuronul să se activeze. Deși acest model de neuron este foarte simplist, are un potențial considerabil de calcul. Acesta poate efectua operațiile logice de bază NOT, OR și AND, cu condiția ca ponderile și pragurile să fie alese corespunzător. După cum știm, orice funcție combinațională multivariabilă poate fi implementată utilizând fie NOT și OR, fie alternativ operațiile NOT și AND. Exemple de porți NOR și NAND cu trei intrări utilizând modelul neuronal McCulloch-Pitts sunt prezentate în figurile 1.3 și 1.4.

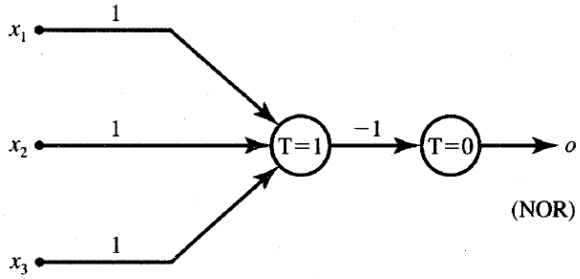


Figura 1.3. Implementarea funcției logice NOR cu 3 intrări cu ajutorul modelului McCulloch-Pitts (sursa: [6])

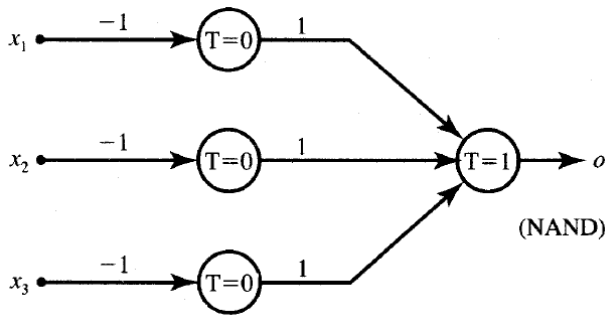


Figura 1.4. Implementarea functiei logice NAND cu 3 intrări cu ajutorul modelului McCulloch-Pitts (sursa: [6])

Acest model neuronal folosește câteva simplificări drastice: permite doar stări binare 0 și 1, ponderile și pragurile neuronilor sunt fixe în model și nu are loc nicio interacțiune între neuronii de rețea. Totuși, acesta stă la baza modelelor următoare și vom considera simbolul din figura 1.5 ca fiind reprezentativ pentru neuronul artificial. [6]

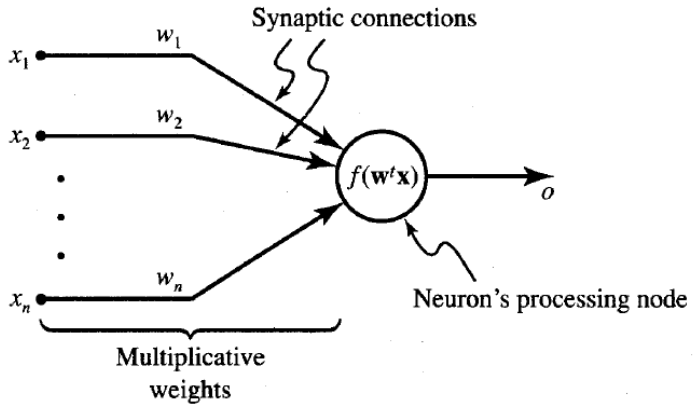


Figura 1.5. Simbolul general al neuronului (sursa: [6])

Perceptronul

Perceptronul este construit pe baza modelului McCulloch-Pitts. Acesta reprezintă cea mai simplă formă a unei rețele neuronale și este utilizat pentru clasificarea vectorilor considerați liniar separabili (adică vectori care se află în părți opuse ale unui hiperplan). Practic, acesta este alcătuit dintr-un singur neuron cu ponderi și bias ajustabile. Algoritmul utilizat pentru determinarea parametrilor acestei rețele a apărut pentru prima dată într-o procedură de învățare dezvoltată de Rosenblatt (1958, 1962) pentru modelul său de perceptron. Rosenblatt a demonstrat că, dacă vectorii folosiți pentru antrenarea perceptronului formează două clase liniar separabile, atunci algoritmul perceptronului converge și poziționează suprafața de decizie sub forma unui hiperplan între cele două clase. Acest hiperplan este definit de:

$$\sum_{i=1}^{m} w_i x_i + b = 0$$

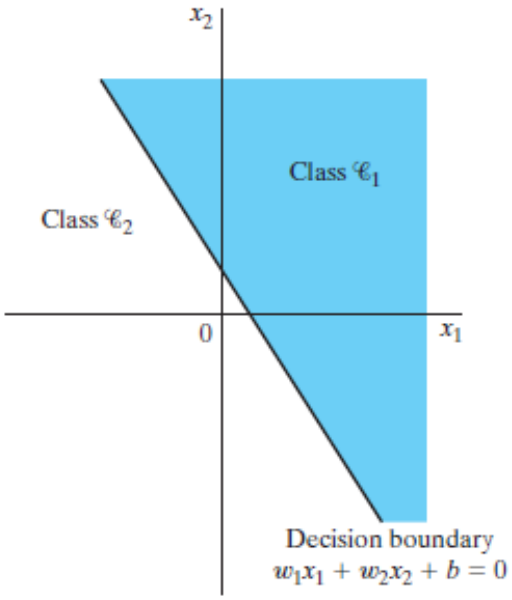


Figura 1.6. Ilustrarea hiperplanului ce reprezintă suprafața de decizie între cele două clase (sursa: [7])

În figura 1.6 este ilustrat cazul cu două variabile de intrare x1 și x2, pentru care suprafața de decizie este reprezentată de o linie dreaptă. Un punct de coordonate (x1, x2) care se află deasupra dreptei face parte din clasa c1, iar un punct care se află sub dreaptă face parte din clasa c2. Efectul bias-ului b este de a depărta suprafața de decizie de origine. Ponderile w1, w2, ..., wm ale perceptronului pot fi adaptate folosind un algoritm iterativ. Pentru adaptare, putem folosi o regulă de corectare a erorilor cunoscută sub denumirea de algoritm de convergență al perceptronului. [7]

Procedura de învățare a corecțiilor de eroare este destul de simplă: în timpul antrenării, se introduce o variabilă de intrare în rețea și se generează un set de valori la ieșire. Apoi, ieșirea actuală este comparată cu ieșirea dorită și se stabilește dacă se potrivesc sau nu. Dacă există o potrivire, nu se face nicio schimbare a ponderilor. Dacă ieșirea reală diferă de ieșirea dorită, trebuie efectuată o modificare a ponderilor. [8]

Adaline

ADALINE (Adaptive Linear Neuron) a fost dezvoltat în anul 1959 de către Bernard Widrow. Diferențele dintre perceptron și Adaline sunt următoarele: perceptronul folosește etichetele claselor în procesul de învățare a coeficienților modelului. Adaline folosește valorile prezise în mod continuu pentru a afla coeficienții modelului, algoritm care este mult mai eficient, deoarece ne spune "cât de mult" a greșit.

Deci, în algoritmul perceptronului, așa cum este ilustrat în figura 1.7, folosim etichetele de clasă pentru a actualiza ponderile, iar în Adaline, folosim un răspuns continuu. Algoritmul este același, însă diferența apare la termenul de modificare al ponderilor. [9]

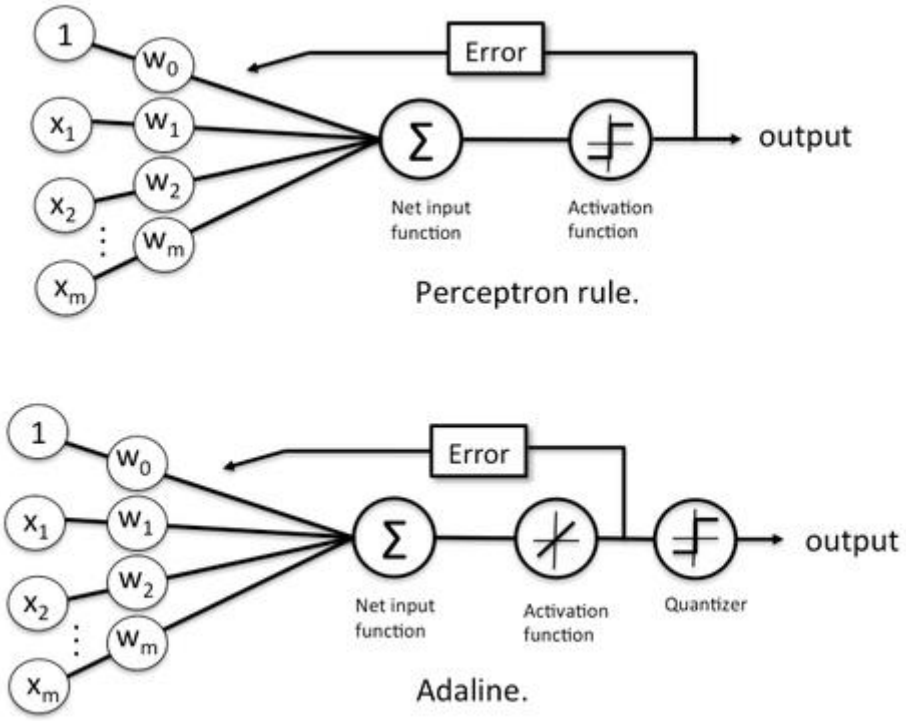


Figura 1.7. Diferența dintre Perceptron și Adaline (sursa: [9])

Totuși, și Adaline putea să clasifice doar clase liniar separabile. O problemă foarte comună care nu putea fi rezolvată cu ajutorul acestor modele rămânea problema XOR. În reprezentarea geometrică a celor două clase observăm că nu există nicio suprafață care să le separe perfect. Această problemă a fost rezolvată ulterior, la apariția Madaline (Many Adalines). Aceasta era o rețea formată din mai mulți Adaline, fiecare reprezentând câte o suprafață de separație. O arhitectura ce rezolvă problema XOR se poate vedea în figura 1.8:

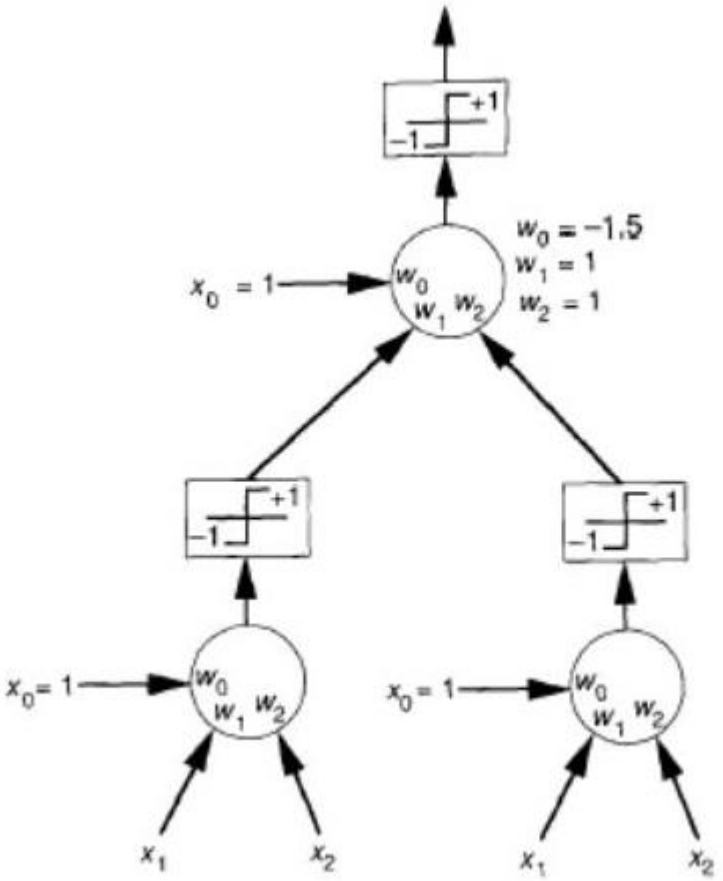


Figura 1.8. Funcția XOR realizată cu MADALINE (sursa: [10])

Rețele neurale - fundamente

După cum am observat și în prezentarea anterioară, o rețea cu un singur strat are limitări în ceea ce privește varietatea sarcinilor pe care le poate îndeplini. De aceea vom analiza în continuare rețelele cu mai multe straturi de tip feed-forward. Minsky și Papert (Minsky & Papert, 1969) au arătat în 1969 că o rețea de tip feed-forward cu două straturi poate rezolva multe dintre acele restricții, însă nu exista o soluție pentru cum să fie ajustate ponderile de la intrare către unitățile ascunse. Ulterior au apărut mai multe soluții, ideea principală din spatele soluției fiind că erorile pentru unitățile din stratul ascuns sunt determinate prin propagarea înapoi (back-propagation) a erorilor unităților de pe stratul de ieșire. Astfel a apărut metoda de învățare denumită backpropagation (BKP). Această metodă poate fi aplicată pentru rețele cu oricâte straturi, însă s-a demonstrat că un singur strat ascuns este suficient pentru a aproxima orice funcție continuă și mărginită, având un număr finit de neuroni. [11]

Funcții de activare

În esență, un neuron realizează următoarele sarcini: calculează o sumă ponderată a intrărilor, iar apoi la această sumă adună biasul, obținând ceea ce se numește potențial de activare. Apoi se aplică o funcție de activare acestui potențial pentru a decide dacă neuronul se activează sau nu.

Vom considera potențialul de activare:

$$net = bias + \sum_{i=1}^{n} w_i x_i$$

În figura 1.9 sunt ilustrate funcții de activare des folosite pentru stratul ascuns, alături de formele lor funcționale și formele derivatelor. Ieșirea reală a neuronului va fi o aplicare a funcției de activare pentru acest potențial de activare:

$$y = f(net)$$

Dacă funcția sigmoidă a fost foarte utilizată mult timp, funcția tangentă hiperbolică este preferată datorită stabilității sale în jurul lui 0. Totuși, funcția ReLU se pare că devinde din ce în ce mai populară, rezultatele acesteia fiind superioare. Această funcție este 0 pentru argumente negative, ceea ce înseamnă că unii neuroni vor răspunde cu 0 la ieșire, ceea ce poate fi de folos în multe situații. [12]

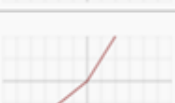
Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1+e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

Figura 1.9. Funcții de activare și derivatele lor (sursa: [13])

Optimizare: Funcția cost. Gradientul negativ.

Pentru a înțelege mai departe cum funcționează optimizarea rețelelor neurale trebuie să înțelegem mai întâi cum evaluăm performanțele acesteia. Funcția cost este utilizată pentru a estima cât de bine funcționează modelul, adică abilitatea acestuia de a estima relația dintre x și y. Adesea, funcția cost este reprezentă ca fiind diferența dintre valoarea dorită și valoarea de la ieșire:

$$E(y,o) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}(y_i - o_i)^2$$

Obiectivul nostru este să determinăm acei parametrii – ponderi sau structură – care minimizează funcția cost. Pentru aceasta se folosește algoritmul gradientului negativ (Gradient Descent – GD)

Gradientul negativ este un algoritm eficient de optimizare care încearcă să găsească minimul unei funcții. Acesta indică direcția în care un model trebuie să o ia pentru a reduce erorile. În acest sens, ponderile vor fi ajustate în mod iterativ în funcție de sensul dictat de gradient. La fiecare iterație, acesta converge treptat către minim. Se consideră că s-a găsit minimul atunci când modificările ponderilor nu mai au niciun efect asupra erorii. Se poate observa evoluția la fiecare iterație până la convergență în figura 1.10.

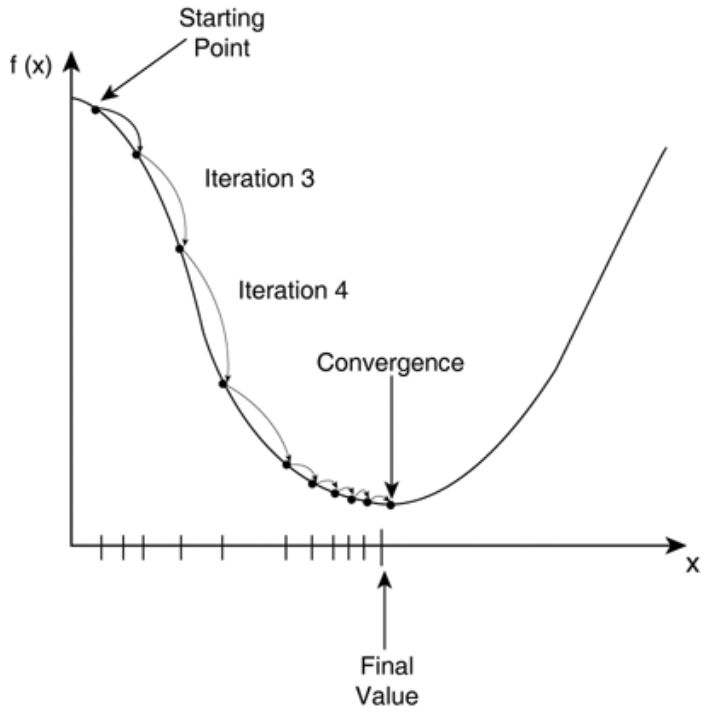


Figura 1.10. Gradient descent (sursa: [14])

Modificările ponderilor de la fiecare pas se fac cu ajutorul regulei Delta enunțată de Widrow-Hoff. Se calculează derivata în funcție de ponderi și apoi se ajustează ponderile cu o valoare mică în direcția opusă gradientului. [15]

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial y_i} \times \frac{\partial y_i}{\partial w_{ji}}$$

deci ponderile vor fi ajustate cu o valoare:

$$\Delta w = -\eta \frac{\partial E}{\partial w}$$

unde η = rata de învățare.

În acest punct, modelul și-a optimizat ponderile pentru a avea o valoare minimă a funcției cost. Acest algoritm permite procesului de antrenare să facă corectări care să ducă modelul către o combinație optimă de parametri. [14]

Vom exemplifica propagarea erorii pentru o rețea cu mai multe straturi. Modelul extins se poate observa în figura 1.11. Partea dreaptă a fiecărui neuron calculează eroarea pătratică medie $\frac{1}{2}(o_i^{(2)} - t_i)$ pentru componenta i din vectorul de ieșire, iar partea stângă calculează diferența $(o_i^{(2)} - t_i)$. Fiecare unitate de ieșire i din rețea are ca funcție de activare sigmoida și produce ieșirea $o_i^{(2)}$. Suma tuturor erorilor pătratice medii reprezintă eroarea E (sau funcția cost). [5]

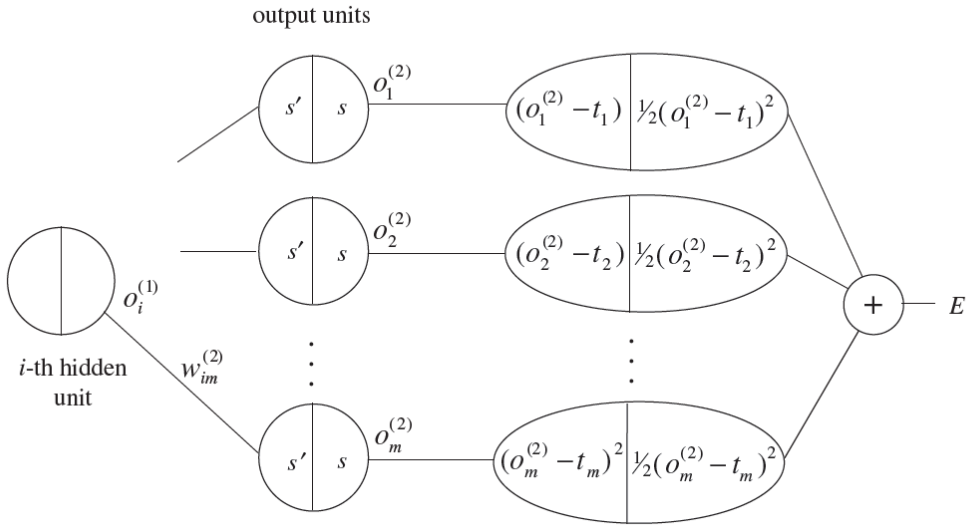


Figura 1.11. Calculul erorii într-o rețea neurală cu mai multe straturi (sursa: [5])

După ce se inițializează în mod aleator ponderile rețelei, se utilizează algoritmul BPK pentru a face corecțiile necesare, acesta oprindu-se doar în momentul în care valoarea funcției cost a devenit suficient de mică. Algoritmul se poate împărți în următorii pași:

1. *calculul direct (feed-forward)*

Vectorul o este intrarea în rețea. Vectorii $o^{(1)}$ și $o^{(2)}$ sunt calculați și stocați, împreună cu derivatele corespunzătoare.

2. *propagarea către stratul de ieșire*

Acum vom determina primul set de derivate $\frac{\partial E}{\partial w_{ij}^{(2)}}$. Calea de propagare pentru unitatea j de ieșire se poate analiza în figura 1.12.

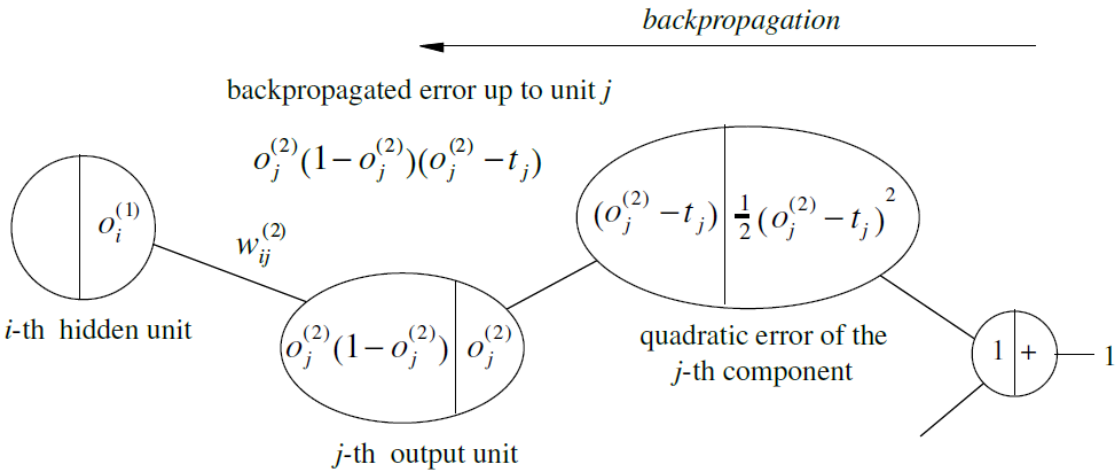


Figura 1.12. Calea de propagare pentru unitatea j de ieșire (sursa: [5])

De pe această cale vom colecta toți termenii multiplicativi care definesc eroarea propagată $\delta_j^{(2)}$.

$$\delta_j^{(2)} = o_j^{(2)}(1 - o_j^{(2)})(o_j^{(2)} - tj)$$

iar derivata parțială este:

$$\frac{\partial E}{\partial w_{ij}^{(2)}} = \left[o_j^{(2)}\left(1 - o_j^{(2)}\right)\left(o_j^{(2)} - tj\right)\right] o_i^{(1)} = \delta_j^{(2)} o_i^{(1)}$$

3. *propagarea către stratul ascuns*

Acum vom determina derivatele parțiale $\frac{\partial E}{\partial w_{ij}^{(1)}}$. Fiecare unitate j din stratul ascuns este conectată cu fiecare unitate q din stratul de ieșire printr-o legătură cu ponderea $w_{jq}^{(2)}$, q = 1,...,m. Eroarea propagată până la unitatea j din stratul ascuns trebuie calculată ținând cont de toate căile posibile, după cum se vede și în figura 1.13. Eroarea propagată va fi:

$$\delta_j^{(1)} = o_j^{(1)}(1 - o_j^{(1)}) \sum_{q=1}^{m} w_{jq}^{(2)} \delta_q^{(2)}$$

iar derivata parțială este:

$$\frac{\partial E}{\partial w_{ij}^{(1)}} = \delta_j^{(1)} o_i$$

Eroarea propagată se calculează în același mod pentru orice număr de straturi ascunse, expresiile derivatelor parțiale pentru funcția cost păstrand aceleași forme analitice.

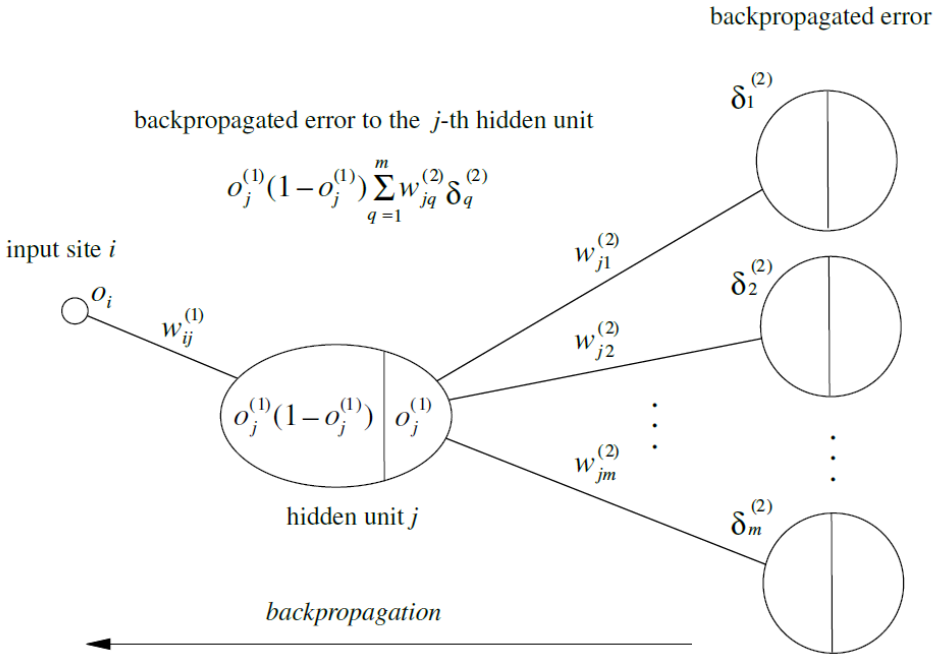


Figura 1.13. Toate căile de propagare către intrarea i (sursa: [5])

4. *actualizarea ponderilor*

După ce s-au calculat toate derivatele parțiale ponderile rețelei sunt actualizate în direcția opusă a gradientului. Constantă η reprezintă rata de învățare și definește pasul de ajustare. Modificările ponderilor sunt date de:

$$\Delta w_{ij}^{(2)} = -\eta o_i^{(1)} \delta_j^{(2)}$$, pentru i = 1,...k+1; j = 1,...m

și

$$\Delta w_{ij}^{(1)} = -\eta o_i \delta_j^{(1)}$$, pentru i = 1,...,n+1; j = 1,...,k

Folosim convenția $o_{n+1} = o_{k+1}^{(1)} = 1$. Este foarte important ca ajustările ponderilor să fie făcute doar după ce eroarea propagată a fost calculată pentru toate unitățile din rețea. Atlfel, ajustările se suprapun cu procesul de propagare iar corecțiile calculate nu mai corespund cu direcția opusă a gradientului. [5]

Rețele neurale convoluționale

Odată cu evoluția rețelelor neurale, numărul straturilor ascunse a început să crească și astfel au apărut rețelele neurale cu învățare profundă (deep neural networks – DNN). Acestea oferă o putere de calcul impresionantă, semnificativ mai mare decât o rețea de tip shallow (cu un singur strat ascuns) cu același număr de unități. Pentru a exemplifica, vom considera 2 structuri, una cu un singur strat ascuns cu un numar *2d* de neuroni, iar altă structură cu 2 straturi ascunse, fiecare strat având *d* neuroni. Pentru rețeaua de tip shallow vom avea o complexitate de calcul $\vartheta(d)$, pe când pentru structura de tip deep obținem o complexitate $\vartheta(d^2)$.

În ceea ce privește funcționalitatea modelului, în cazul structurii de tip deep avem un sistem inteligent, adaptiv în totalitate, după cum se observă în figura 1.14.

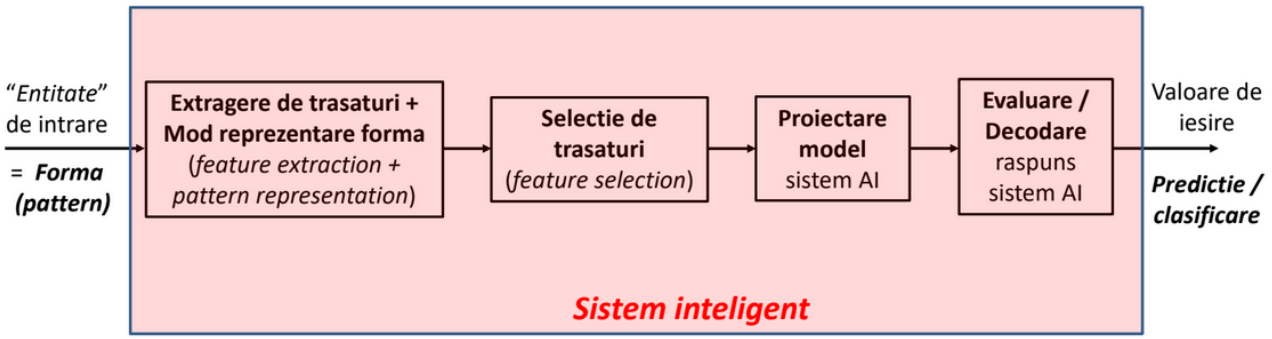


Figura 1.14. Structura unui sistem de inteligența artificială (sursa: [16])

După cum se observă, absolut toate etapele sunt realizate de către sistem. Trăsaturile și structura se detemină adaptiv pe parcursul antrenării rețelei. Pentru aceleași forme de intrare, pentru aplicații diferite, se pot obține etape de prelucrare total diferite. Dacă am utiliza o structură de tip shallow, atunci alegerea trăsăturilor și a structurii se bazează pe experiența celui care o proiectează. Odată alese, ele rămân fixe pe tot parcursul antrenării și funcționării rețelei. În acest caz, pentru același tip de formă de intrare, prelucrările sunt aproximativ aceleași. Dezavantajul structurilor de tip deep este că modelul general este prea complex, de aceea ele trebuie specializate. [16]

Rețelele neurale convoluționale (CNN) se bazează în principal pe intrări reprezentate de imagini. Una dintre diferențele principale este că neuronii din CNN sunt organizați în trei dimensiuni, dimensiunile spațiale ale intrării (înalțime și lățime) și adâncime. Adâncimea nu se referă la numărul total de straturi din rețea, ci la a 3-a dimensiune a volumului de activare. Spre deosebire de ANN-urile clasice, neuronii dintr-un strat nu sunt conectați în totalitate cu stratul precedent, ci doar cu o porțiune din acesta. În practică, asta înseamnă că, dacă avem volumul de intrare cu dimensiunile 64x64x3 (înălțime, lățime, adâncime), acesta va conduce către un strat de ieșire final comprimat de dimensiune 1x1x*n* (unde *n* reprezintă numărul de clase posibile).

CNN-urile sunt alcătuite din trei tipuri de straturi: convoluțional, pooling și fully-connected. Când acestea sunt combinate, se formează o arhitectură de tip CNN. Putem observa o astfel de arhitectură în figura 1.15. Aceasta este o arhitectură simplificată utilizată pentru clasificarea MNIST (clasificarea cifrelor).

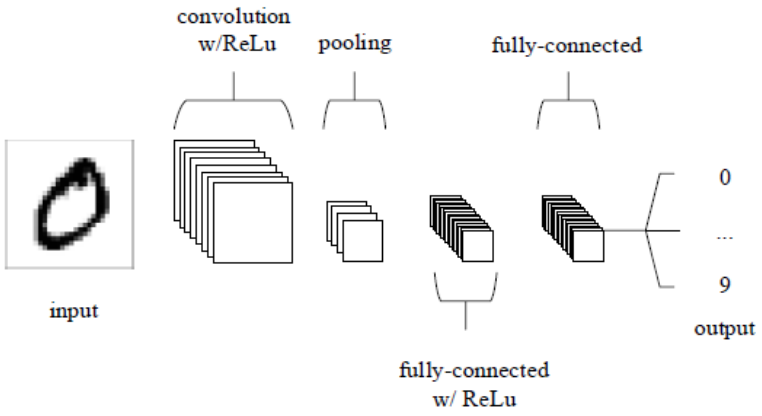


Figura 1.15. Arhitectura CNN, alcătuită din 5 straturi (sursa: [17])

Funcționalitatea de bază pentru exemplul de CNN de mai sus se împarte în patru arii importante [17]:

1. stratul de intrare, care conține valorile pixelilor din imagine
2. stratul convoluțional va determina ieșirea neuronilor care sunt conectați la intrare prin calculul produsului scalar dintre ponderi și intrări, apoi se va aplica ReLU.
3. stratul pooling va subeșantiona spațiul de intrare, reducând numărul de parametrii
4. stratul fully-connected are aceeași funcționalitatea ca straturile dintr-un ANN standard.

Stratul convoluțional

După cum îi spune și numele, acest strat joacă un rol esențial în modul de operare al CNN-urilor. Parametrii acestui strat se centrează pe utilizarea anumitor filtre. Când datele de intrare ajung la un strat convoluțional, acesta realizează o convoluție între fiecare filtru și toată suprafața de intrare. În timp ce filtrul este glisat peste vectorul de intrare, se calculează produsul scalar pentru fiecare valoare din filtru. În figura 1.16 putem observa cum elementrul central al filtrului este plasat peste un pixel din vectorul de intrare, care ulterior va fi calculat și înlocuit cu o sumă ponderată între el și vecinii săi.

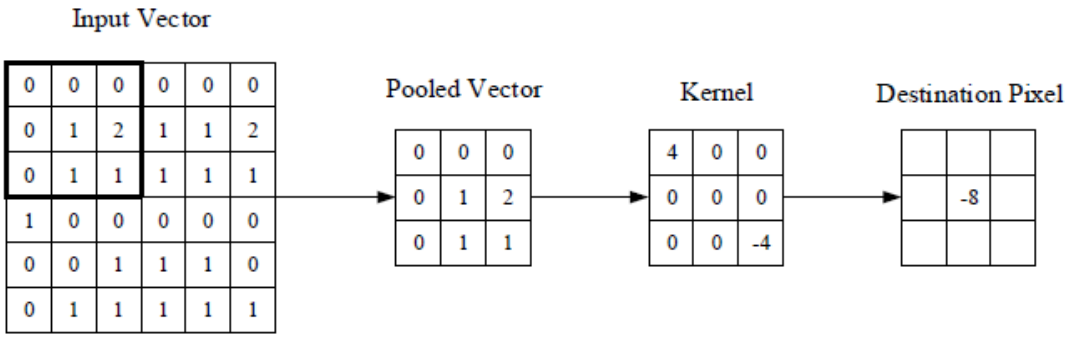


Figura 1.16. Reprezentarea stratului convoluțional (sursa: [17])

Straturile convoluționale sunt capabile să reducă semnificativ complexitatea modelului prin optimizarea ieșirii. Aceste optimizări sunt realizate prin trei hiperparametri: adâncime, pasul și setarea "zero-padding".

Adâncimea volumului de ieșire produs de stratul convoluțional poate fi setată manual prin numărul de neuroni din strat. Reducând acest hiperparametru se poate minimiza semificativ numărul total de neuroni din rețea, dar totodată poate să reducă și capabilitățile de recunoaștere ale modelului.

De asemenea, mai putem seta pasul cu care ne deplasăm în spațiul de intrare. Setarea zero-padding este procesul de adăugare a unei borduri în vectorul de intrare, ceea ce este foarte importantă pentru controlul dimensiunilor de ieșire.

Este foarte important să înțelegem că prin aceste tehnici se alterează dimensiunile spațiului de ieșire din stratul convoluțional. Pentru a calcula aceste dimensiuni, putem folosi formula:

$$\frac{(V - R) + 2Z}{S + 1}$$

unde V reprezintă dimensiunile volumului de intrare (înălțime x lățime x adâncime), R reprezintă dimensiunea câmpului receptiv, adică regiunea de conexiune dintre intrare și stratul convoluțional,

Z este valoarea setată pentru zero-padding, iar S este reprezentată de stride (pasul). Daca rezultatul acestui calcul nu este egal cu un număr întreg, atunci valoarea pentru stride a fost setată greșit. [17]

Straturile de tip pooling

Aceste straturi au ca scop reducerea treptată a dimensiunilor reprezentării, deci reducerea parametrilor și a complexității de calcul a modelului.

Stratul pooling operează pe fiecare hartă de activare din vectorul de intrare, scalând dimensiunile folosind funcția MAX. În majoritatea CNN-urilor, aceste straturi sunt de tipul max-pooling cu filtre de dimensiuni 2x2 aplicate cu un pas egal cu 2 pe suprafața de intrare. Această tehnică scalează intrarea reducând dimensiunile acesteia cu 25% din cele originale, menținând adâncimea neschimbată.

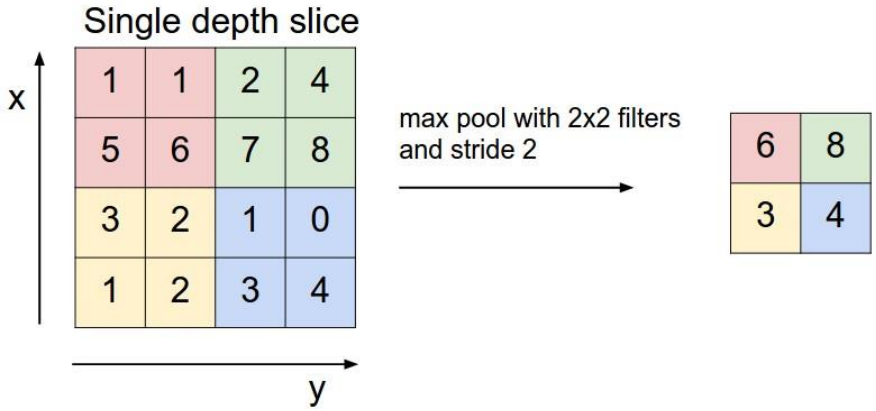


Figura 1.17. Aplicarea max-pooling (sursa: [18])

Straturile de tip fully-connected

Aceste straturi conțin neuroni care sunt conectați direct cu neuronii din 2 straturi adiacente, similar cu poziționarea neuronilor din ANN-urile clasice.

Arhitecturi CNN

Există mai multe arhitecturi CNN care s-au remarcat, cele mai cunoscute fiind [18]:

- **LeNet.** Primele aplicații de succes ale CNN-urilor au fost dezvoltate de Yann Lecun în anii 1990. Printre acestea, cea mai cunoscută este arhitectura LeNet care a fost folosită la citirea codurilor poștale, a cifrelor, etc.

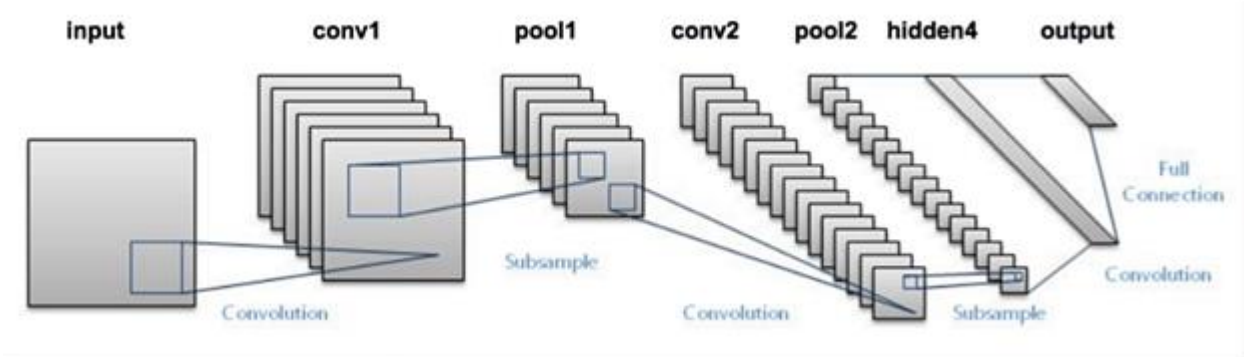


Figura 1.18. Arhitectura LeNet (sursa: [19])

- **AlexNet.** Această arhitectură a fost dezvoltată de Alex Krizhevsky, Ilya Sutskever și Geoff Hinton. Cu o arhitectură foarte similară cu precedenta, aceasta era mult mai mare și conținea straturi convoluționale suprapuse unul peste celelălalt, în timp ce LeNet avea un sigur strat convoluțional precedat imediat de un strat de tip pooling

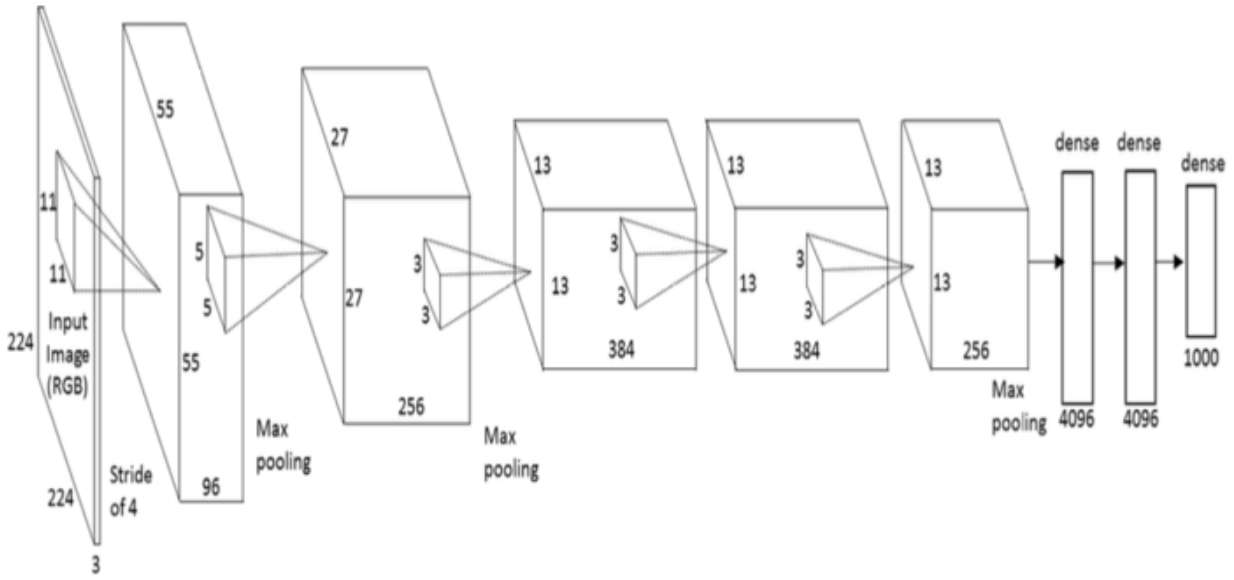


Figura 1.19. Arhitectura AlexNet (sursa: [20])

- **ZF Net.** CNN dezvoltat de Matthew Zeiler and Rob Fergus. Reprezenta o îmbunătățire a rețelei AlexNet prin modificarea hiperparametrilor, în mod special prin mărirea dimensiunii stratului convoluțional de mijloc și micșorarea pasului și a dimensiunii filtrului din primul strat.

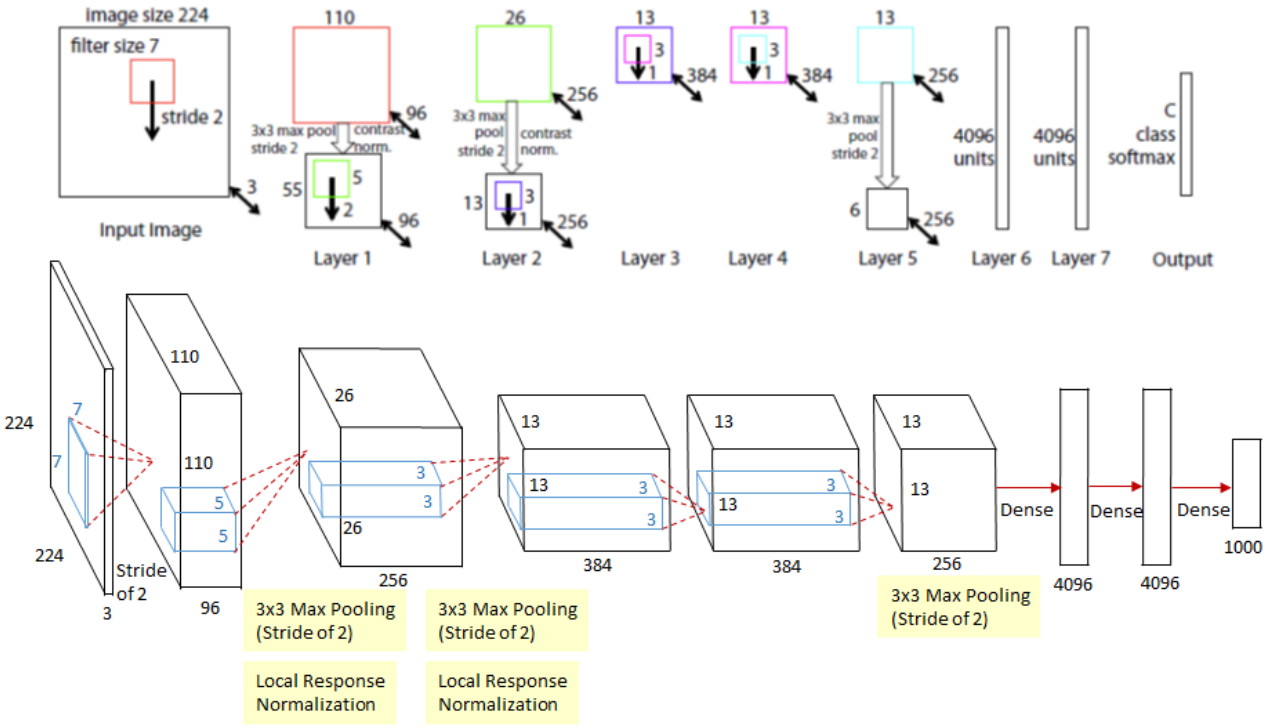


Figura 1.20. Arhitectura ZFNet (sursa: [21])

- **GoogLeNet.** O arhitectură dezvoltată de Szegedy și alți cercetători de la Google. Contribuția principală a fost dezvoltarea unui modul inițial care reducea semnificativ numărul de parametri din rețea (4M comparat cu 60M de la AlexNet). În plus, folosea average pooling în loc de straturi fully-connected la sfârșitul rețelei, eliminând o cantitate mare de parametri care nu contau. Aceasta are în total 22 de straturi. În figura 1.21 cu albastru sunt figurate straturile convoluționale, cu roșu straturile de tip max-pooling, cu verde straturi de normalizare.

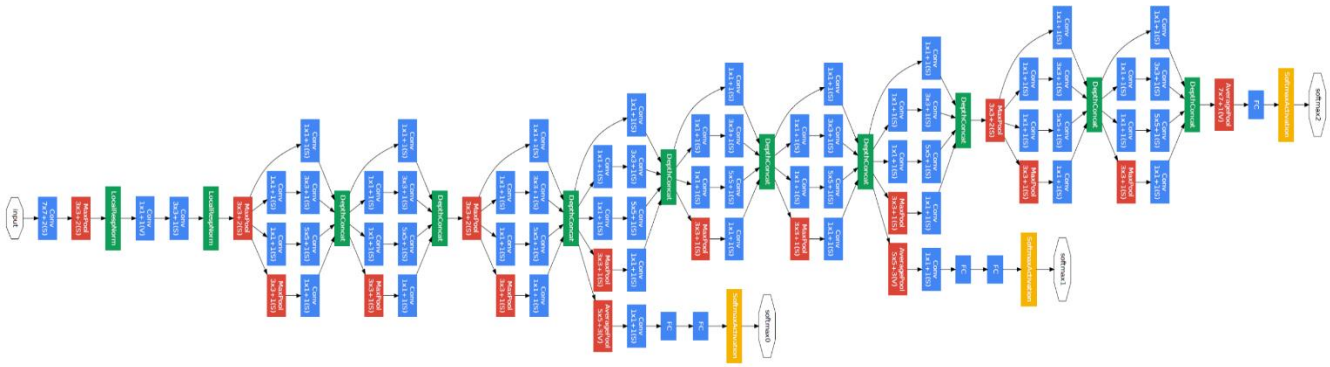


Figura 1.21. Arhitectura GoogLeNet (sursa: [22])

- **VGGNet.** O rețea dezvoltată de Karen Simonyan și Andrew Zisserman. Aceștia au demonstrat că adâncimea rețelei este o componentă critică în evaluarea performanței. Rezultatul final conține 16 straturi convoluționale complet conectate și o arhitectură omogenă care realizează convoluții de tip 3x3 și pooling 2x2 de la început până la sfârșit.

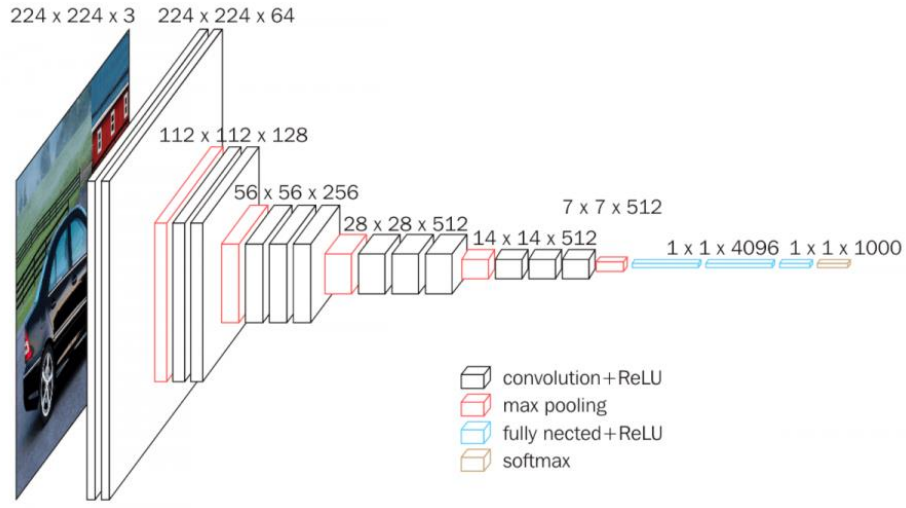


Figura 1.22. Arhitectura VGGNet (sursa: [23])

- **ResNet.** Residual Network a fost dezvoltată de Kaiming He și alții. Aceasta are conexiuni speciale și realizează normalizare pe batch. Arhitectura nu conține straturi fully-connected la sfârșitul rețelei.

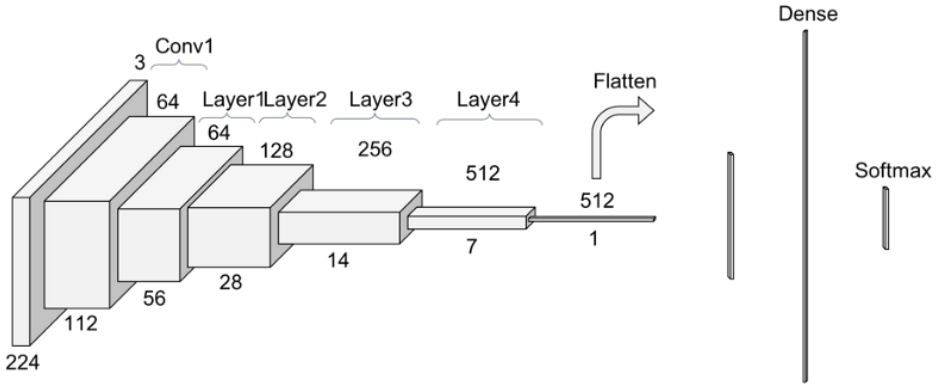


Figura 1.23. Arhitectura Resnet34 (sursa: [24])

Capitolul 2. Setup experimental

Formularea problemei

În principiu, detectarea unei mâini într-un flux video pare a fi simplă, dacă aplicăm algoritmi generali de detecție a obiectelor. Însă, în practică, această sarcină este mult mai complicată și trebuie luate în considerare anumite aspecte. Mâinile sunt obiecte foarte flexibile, al căror aspect și poziție pot varia considerabil, de aceea modelul nostru trebuie să fie suficient de complex pentru a diferenția o mână într-o scenă. Rețelele neurale convoluționale (CNN) oferă o performanță foarte bună în aplicații de clasificare, dar, pentru detecția de obiecte, noua abordare este de a împărți imaginea în mai multe zone, redimensionarea fiecărei zone, ajustarea parametrilor rețelei CNN pentru aceasta, iar apoi combinarea ieșirilor fiecărei zone pentru a obține detecția obiectului. Numărul de zone pe care le putem alege este foarte mare, de aceea este important să alegem regiunile optime, care cuprind cât mai multe obiecte. [2]

În cadrul acestui proiect ne propunem să realizăm o aplicație ce detectează mâna utilizatorului dintr-un flux video live captat cu ajutorul camerei web. Scopul acestui proiect este de a evidenția potențialul rețelelor neurale în aplicațiile de detecție și nu numai. Vom avea mai multe etape pe parcursul dezvoltării aplicației, printre care achiziția de date cu ajutorul camerei web, cu aceasta vom realiza și o bază de date, dar și captarea fluxului în timp real pentru testare. În continuare vom testa mai multe arhitecturi de rețele neurale. Rețeaua va prelua parametrii extrași din fluxul video și va decide dacă există o mână în imagine. La ieșirea rețelei vom avea o aproximare a poziției acesteia, care va fi folosită pentru desenarea unui dreptunghi în jurul mâinii detectate. Pentru antrenarea rețelei neurale ne propunem să utilizăm atât o bază de date publică, cât și una proprie, realizată în cadrul proiectului, ulterior testând funcționalitatea pe fluxul preluat în timp real de la camera video.

Partea practică a proiectului va conține două părți. În prima parte vom defini propria arhitectură de rețea, vom realiza o bază de date pe care vom face antrenarea, iar apoi vom testa detecția în timp real. În cea de-a doua parte vom alege un model preantrenat din pachetul Tensorflow și vom realiza o reantrenare a straturilor finale pe baza de date Egohands Dataset [2]. Detecția o vom face folosind modelul reantrenat de noi cu ajutorul aplicației open-source Tensorflow Object Detection API [25]. Vom compara rezultatele obținute.

Baza de date

Acest proiect conține două parți, de aceea vom utiliza două baze de date. În continuare vom detalia conținutul acestor baze de date și modul în care acestea au fost utilizate.

Baza de date proprie

Intuitiv, atunci când dezvoltăm o rețea care vrem să detecteze mâini, aceasta trebuie antrenată pe un set de date corespunzător. Problema apare atunci când trebuie să etichetăm aceste date. Trebuie să determinăm care este eticheta corectă pentru aplicația noastră. Dacă vrem să facem o simplă detecție, adică ieșirea rețelei noastre să fie: “DA, există o mână în imagine” sau “NU, nu există o mână în imagine”, atunci sarcina este ușoară. Dacă dorim să estimăm și unde în imagine se află mâna, atunci și etichetele bazei noastre de date trebuie să fie corespunzătoare.

În prima parte a proiectului am început cu o arhitectură simplă de rețea, ce detectează dacă este sau nu o mână în imagine. Pentru această etapă, am realizat o bază de date ce conține în total 517 imagini: 390 de imagini care conțin o mână și 127 imagini care nu conțin mâini. Pentru utilizarea lor

în rețea am împărțit setul de date sub următoarea structură: 360 de imagini cu mână și 115 imagini fără mână pentru antrenare, iar pentru testare am folosit 30 de imagini cu mână și 12 imagini fără mână.

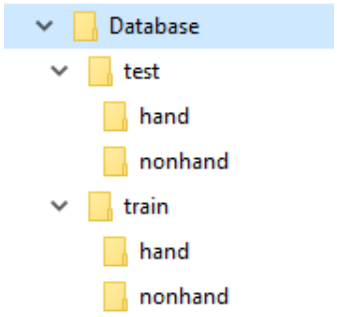


Figura 2.1. Structura bazei de date pentru detecția mâinii

Etichetarea pentru această etapă este destul de simplă, întrucât se rezumă la o problemă de clasificare: vom eticheta cu 1 imaginile care conțin mâini și cu 0 imaginile care nu conțin mâini. Un exemplu avem mai jos:

- această imagine va primi eticheta 1 deoarece conține o mână

Figura 2.2. Imagine din baza de date proprie conținând o mână

- această imagine va primi eticheta 0 deoarece nu conține o mână

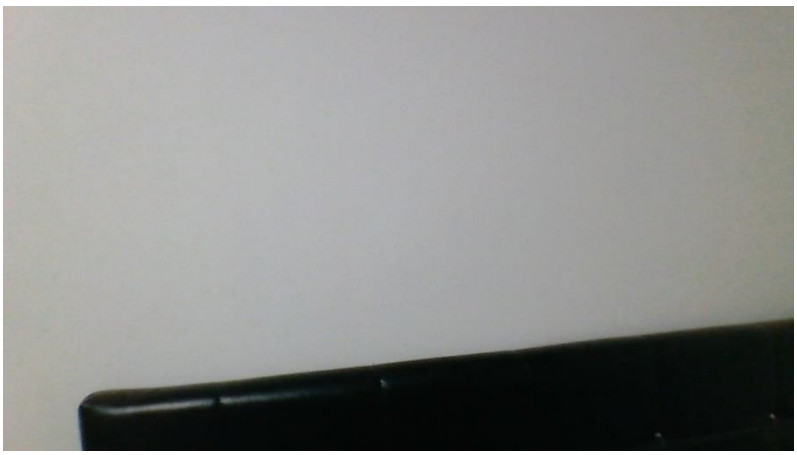


Figura 2.3. Imagine din baza de date proprie care nu conține o mână

În continuare am vrut să prezicem și poziția mâinii prin încadrarea acesteia cu un dreptunghi, iar pentru acest task am fost nevoiți să restructurăm baza de date. În acest caz am folosit doar poze ce conțineau mâini, respectiv cele 390 de imagini menționate mai sus. Baza de date arată astfel:

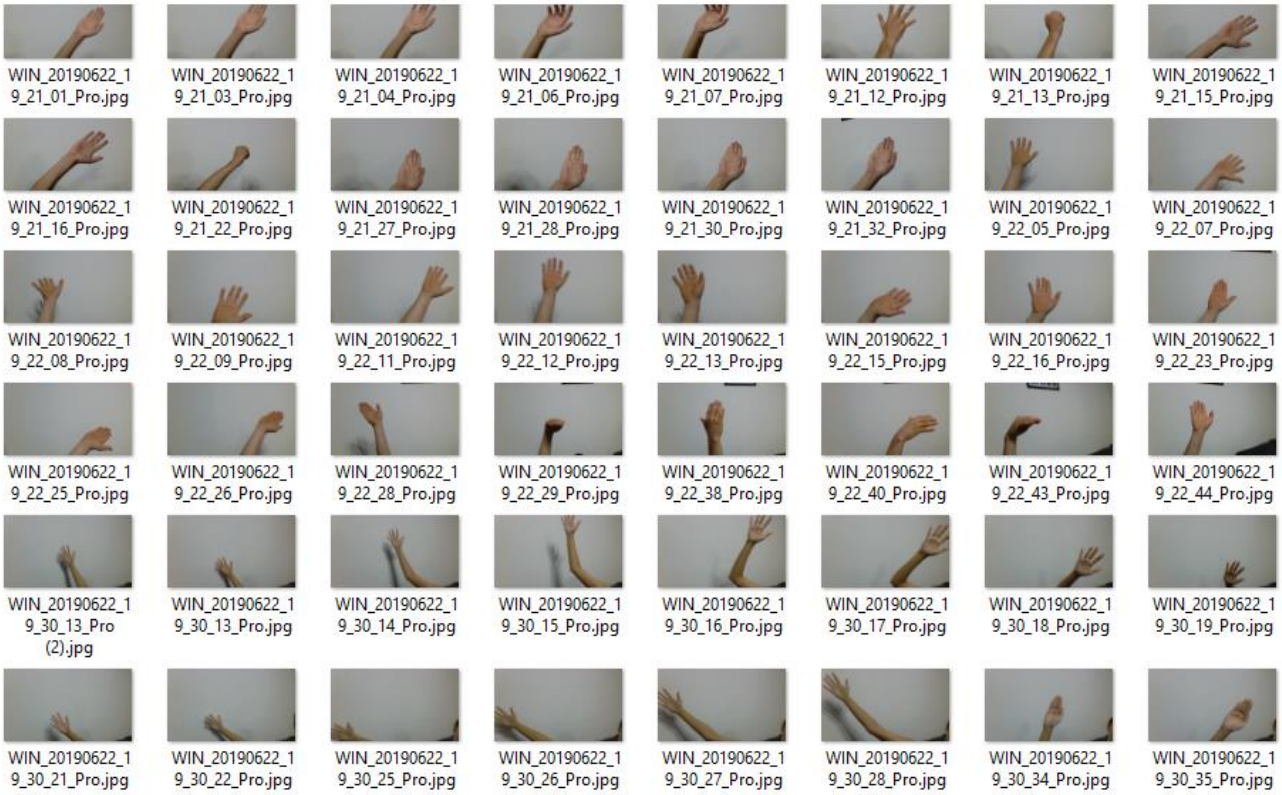


Figura 2.4. Baza de date proprie folosită pentru detecția poziției mâinii

Am împarțit această bază de date în două: 360 de imagini pentru antrenare și 30 de imagini pentru testare. Apoi a urmat partea de etichetare a imaginilor. Întrucât noi dorim ca la ieșire să avem un dreptunghi pe ecran (bounding box) care să ne încadreze mâna, la intrare etichetele imaginilor trebuie să fie chiar aceste dreptunghiuri.

Pentru a desena un dreptunghi pe ecran sunt suficiente doar două puncte, și anume colțurile acestuia. Astfel că am etichetat manual fiecare poză, această etichetă fiind compusă din 4 elemente, mai exact 4 coordonate: coordonatele celor 2 puncte prin care trece dreptunghiul.

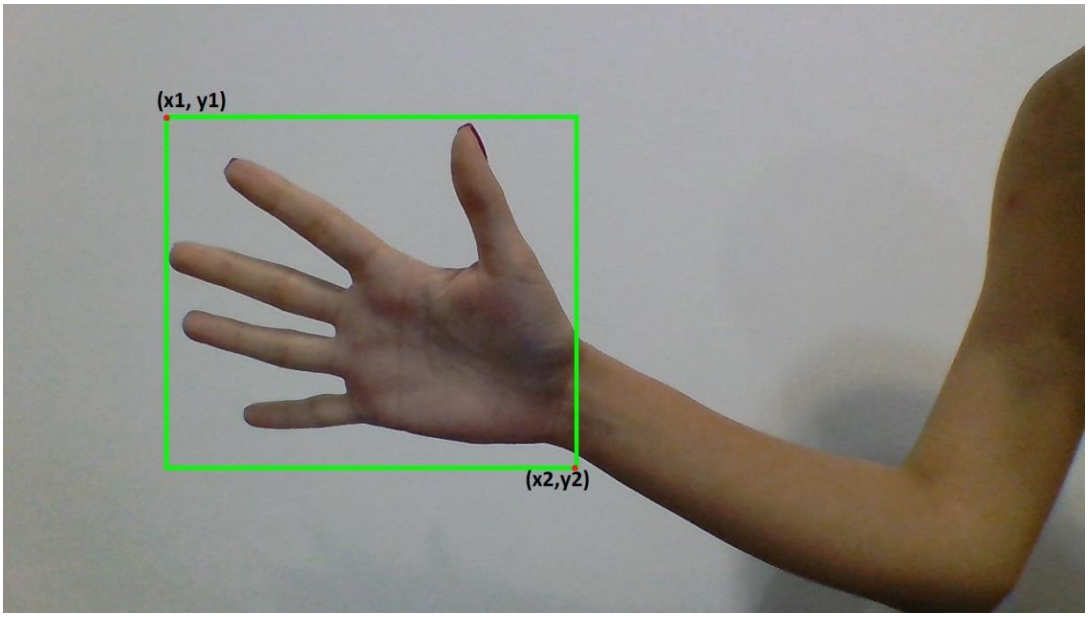


Figura 2.5. Exemplu de etichetare a imaginii

Pentru fiecare poză în parte am salvat 4 etichete: coordonatele x și y ale punctului din colțul stânga sus, respectiv coordonatele x și y ale punctului din colțul dreapta jos. Toate aceste informații au fost salvate într-un fișier .csv, care are 360 de linii corespunzătoare fiecărei imagini și conține pe fiecare linie numele imaginii, x1, y1, x2 și y2.

Baza de date EgoHands

În cea de-a doua parte a proiectului vom folosi o bază de date mult mai mare și mai complexă. Această bază de date se numește EgoHands Dataset [2] și este disponibilă pentru descărcare pe site-ul web al celor de la Indiana University.

Figura 2.6. Baza de date EgoHands (sursa: [2])

Acest set de imagini este unul foarte complex din mai multe motive. Baza de date EgoHands conține 48 de videoclipuri înregistrate cu ajutorul Google Glass cu interacțiuni complexe între două persoane stând față în față. Pentru a crea o bază de date cât se poate de realistă, au fost colectate date

de la diferite perechi de 4 participanți care stăteau față în față în timp ce realizau diverse activități. Au fost alese 4 activități care implică interacțiunea și mișcarea mâinilor:

- jocul de cărți, mai exact o versiune simpla a jocului Mau Mau
- jocul de șah, unde, pentru eficiență, participanții au fost încurajați să se concentreze mai degrabă pe viteză decât pe strategie
- rezolvarea unui puzzle Jigsaw de 24/48 de piese
- jocul Jenga, care implică eliminarea pieselor dintr-un puzzle 3D până când acesta se prăbușește

De asemenea, s-a variat și contextul imaginilor prin înregistrarea videoclipurilor în 3 locații diferite: o masă dintr-o sală de conferință, o masă dintr-o grădină și o masă de cafea într-o casă. Videoclipurile au fost filmate în mai multe zile, iar participanții nu au fost restricționați în ceea ce privește îmbrăcămintea, astfel că s-a obținut o varietate foarte mare în imagini. S-au obținut 48 de combinații unice de videoclipuri, înregistrate în dimensiunea 720x1280px la 30Hz. Acestea au fost eșantionate și într-un final s-a obținut un set de date cu 4800 de imagini.

Baza de date este structurată astfel: directorul "_LABELLED_SAMPLES" conține 48 de foldere, corespunzătoare fiecărui videoclip, denumite în funcție de activitate (cărți, șah, Jenga, puzzle), respectiv locație (grădină, birou, sufragerie).

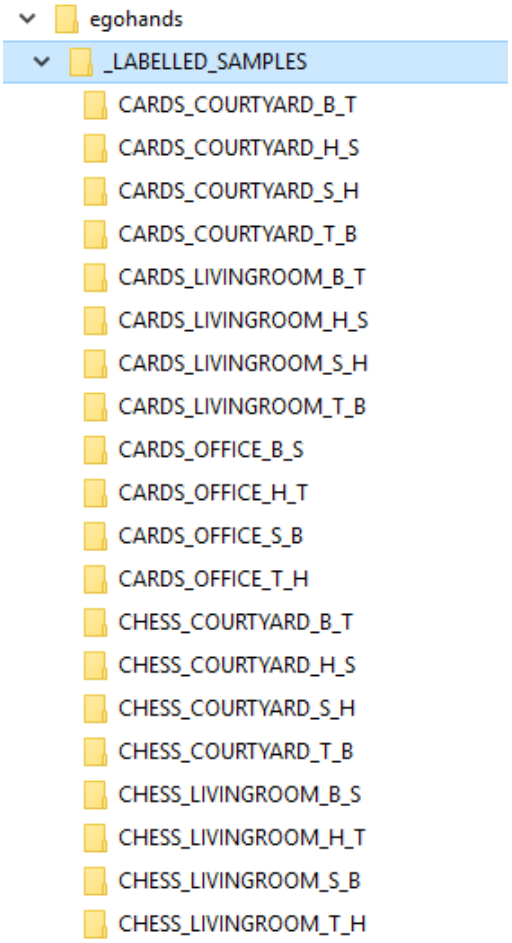


Figura 2.7. Structura bazei de date EgoHands

Fiecare folder conține 100 de cadre sub formă de fișiere JPEG. De asemenea, fiecare folder, pe lângă cele 100 de imagini, conține un fișier denumit *"polygons.mat"*. Acest fișier conține o matrice cu 100 de linii, corespunzătoare fiecărei imagini. Pe fiecare linie avem 4 elemente: *myleft*, *myright*, *yourleft*, *yourright*. Fiecare dintre aceste elemente conține un vector cu 2 coloane, acestea reprezentând coordonatele x și y ale unor puncte. Aceste puncte descriu poligonul care încadrează mâna corespunzătoare din imaginea respectivă.

polygons

1x100 struct with 4 fields

F...	myleft	myright	yourleft	yourright
1	[]	[]	97x2 doub...	111x2 double
2	[]	[]	86x2 doub...	131x2 double
3	[]	91x2 doub...	149x2 dou...	106x2 double
4	[]	[]	144x2 dou...	115x2 double
5	[]	[]	124x2 dou...	123x2 double
6	[]	[]	126x2 dou...	126x2 double
7	[]	[]	95x2 doub...	116x2 double
8	[]	[]	138x2 dou...	106x2 double
9	[]	[]	105x2 dou...	102x2 double
10	[]	[]	98x2 doub...	89x2 double

polygons(1).yourleft

	1	2
1	692.6237	487.8402
2	684.7062	493.1186
3	680.7474	493.1186
4	676.7887	493.1186
5	675.4691	494.4381
6	672.8299	494.4381
7	668.8711	494.4381
8	666.2320	495.7577
9	662.2732	497.0773
10	659.6340	497.0773

Figura 2.8. Conținutul fișierului polygons.mat și detalierea unui element

Spre exemplu, în prima imagine din folderul respectiv apar doar mâinile partenerului, de aceea câmpurile *myleft* și *myright* de pe prima linie a matricei sunt goale. Câmpurile *yourleft* și *yourright* conțin câte un set de puncte ce încadrează mâinile partenerului. După cum putem vedea, poligonul corespunzător mâinii drepte este descris de mai multe puncte, ceea ce înseamnă că se află mai aproape. Un exemplu de imagine poate fi văzut în figura 2.9:

Figura 2.9. Poligoanele ce încadrează mâinile din imagini (sursa: [26])

Implementare

Pentru prima parte a proiectului, după cum am menționat, vom face o simplă detecție a prezenței mâinii în imagine. Această implementare, deși este o problemă de clasificare, ne va ajuta să înțelegem cum funcționează un astfel de sistem inteligent și cum să optimizăm implementările viitoare. Schema acestei etape se poate vedea în figura 2.10.

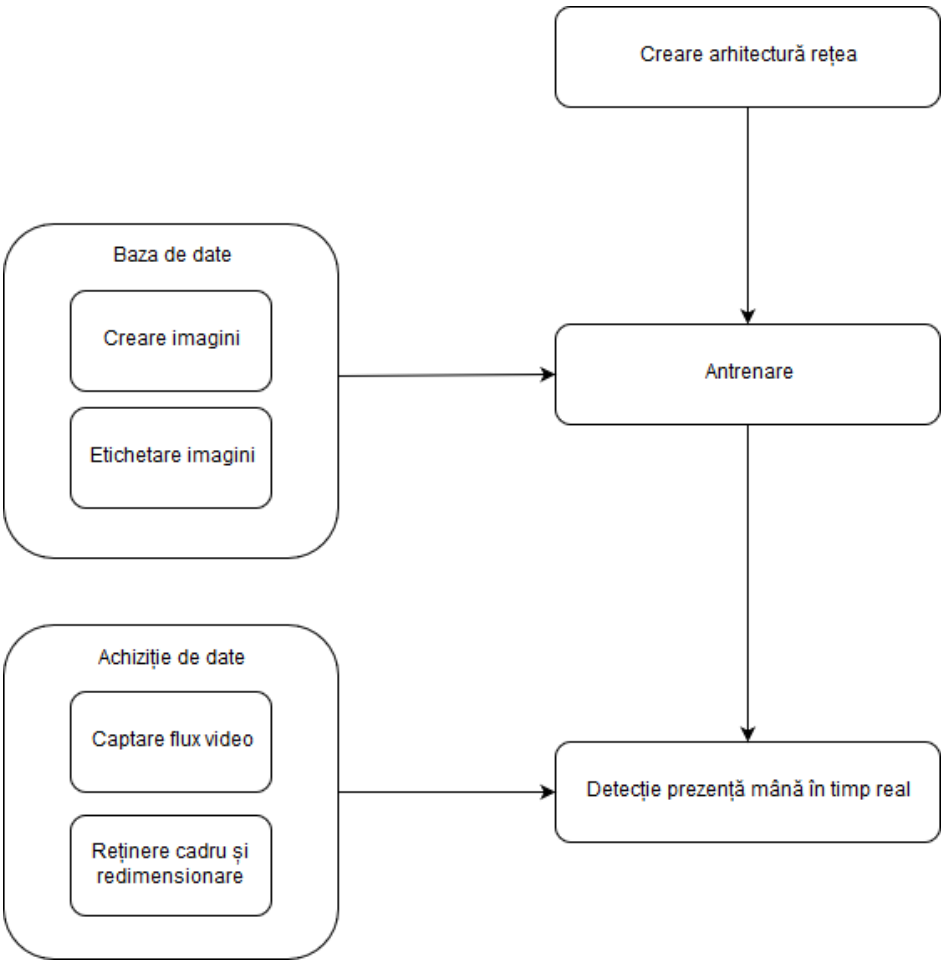


Figura 2.10. Schema bloc a algoritmului de detecție a prezenței mâinii

După cum se poate vedea, în prima parte vom realiza baza de date. Acest procedeu a fost detaliat în secțiunea anterioară. Ne amintim faptul că în acest pas intrările sunt: un set de imagini care conțin mâini și imagini care nu conțin mâini, etichetate corespunzător – 1 dacă este prezentă și 0 dacă nu este. După ce vom concepe și arhitectura de rețea, vom efectua antrenarea acestuia folosind setul de date generat. În urma procesului de antrenare se va genera un model care va fi folosit mai departe pentru prezicerea în timp real din blocul de detecție. Achiziția de date se va face cu ajutorul camerei web, se va reține câte un cadru care va fi redimensionat și apoi va fi folosit ca intrare pentru modelul generat anterior. La ieșire, acest sistem ne va spune dacă există sau nu o mână în imagine.

În cea de-a doua parte, vom realiza un sistem capabil să ne spună nu doar dacă se află sau nu o mână în imagine, ci ne va indica și locația acesteia. În acest caz, nu mai avem o problemă de clasificare, ci una de regresie liniară. O schemă detaliată a acestui sistem poate fi analizată în figura 2.11. Observăm că schema este foarte asemănătoare cu cea anterioară, având doar un bloc suplimentar, însă funcționalitatea este total diferită.

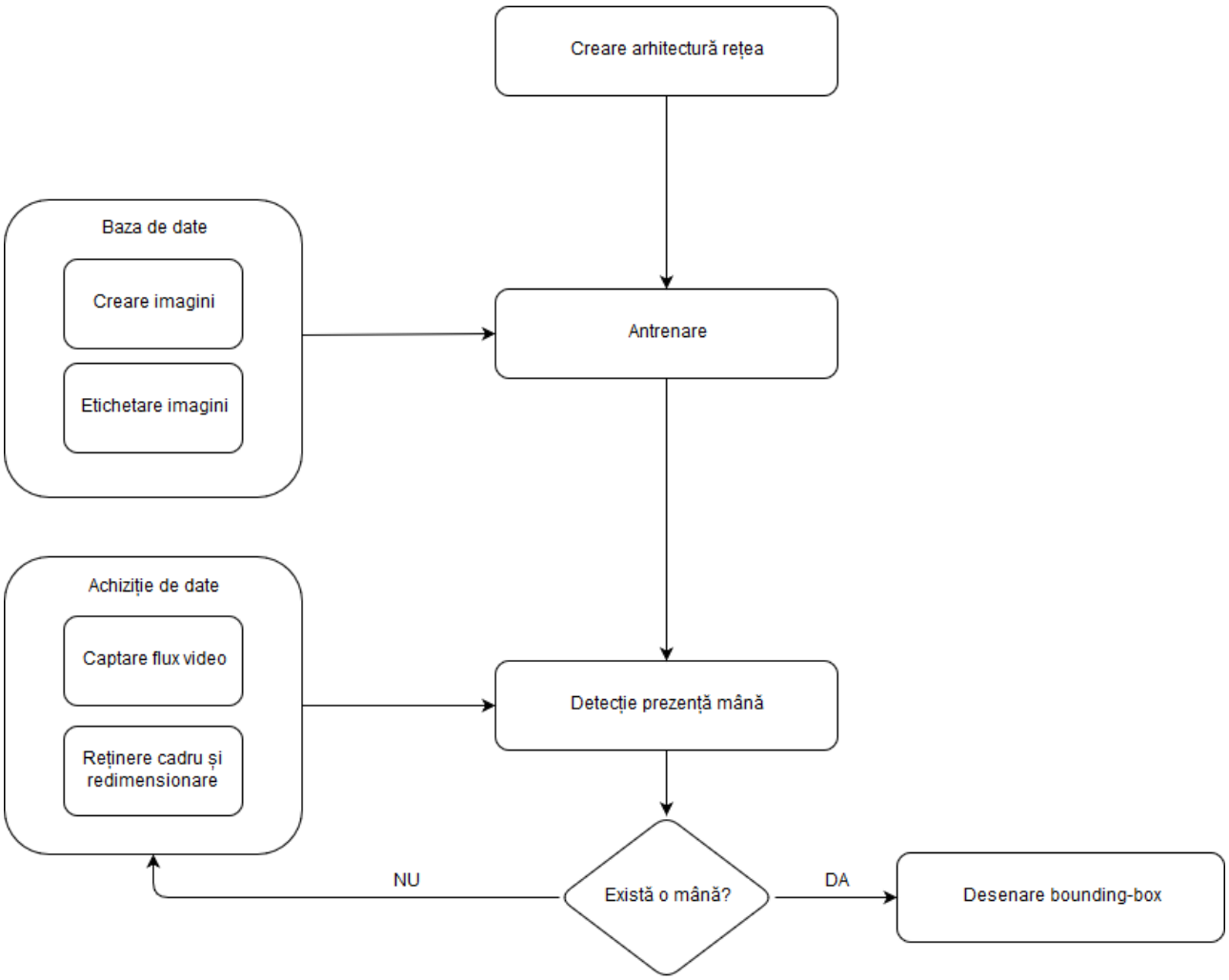


Figura 2.11. Schema bloc a sistemului de urmărire a mâinii

Realizarea bazei de date în acest caz este mai dificilă. După cum am detaliat în secțiunea anterioară, în cazul în care vrem să detectăm unde anume se află mâna în imagine, nu putem folosi aceeași bază de date ca în cazul anterior. Trebuie să păstrăm doar imaginile care conțin mâini și să etichetăm manual fiecare imagine cu coordonatele a 2 puncte ce reprezintă colțurile dreptunghiului ce încadrează mâna. Vom defini o arhitectură de rețea și vom face antrenarea acesteia pe noua bază de date. În urma acestei antrenări vom obține un model pe care îl vom folosi în partea de detecție. Pentru a urmări mâna în fluxul video live trebuie mai întâi să stabilim dacă este sau nu o mână în imagine. Dacă este, atunci modelul va prezice un set de coordonate și va fi desenat un dreptunghi pe ecran. Dacă nu, se va continua achiziția de date fără să se deseneze nimic.

În cea de-a treia parte a proiectului, vom folosi Tensorflow Object Detection API pentru a antrena un model preantrenat și a detecta obiecte dintr-o captură de la camera web.

Rețelele neurale convoluționale au fost aplicate inițial cu succes în clasificarea imaginilor, după care a urmat o tendință de utilizare a rețelelor neurale de tip deep (DNN) într-o varietate de aplicații. Funcționalitatea CNN-urilor a fost apoi extinsă la detecția de obiecte și localizarea lor folosind rețele ce realizează regresie liniară. O rețea CNN poate fi considerată ca fiind alcătuită din cel putin 2 clase de straturi: straturi convoluționale și straturi de clasificare. Straturile convoluționale realizează o extragere adaptivă de trăsături, sunt capabile să învețe și să descompună intrările

originale în trăsături. Rețelele convoluționale deja antrenate pot fi reutilizate pentru recunoașterea altor obiecte prin reantrenarea straturilor de clasificare cu noua bază de date. [27]

O schemă bloc ce descrie pașii necesari acestei etape este ilustrată în figura 2.12.

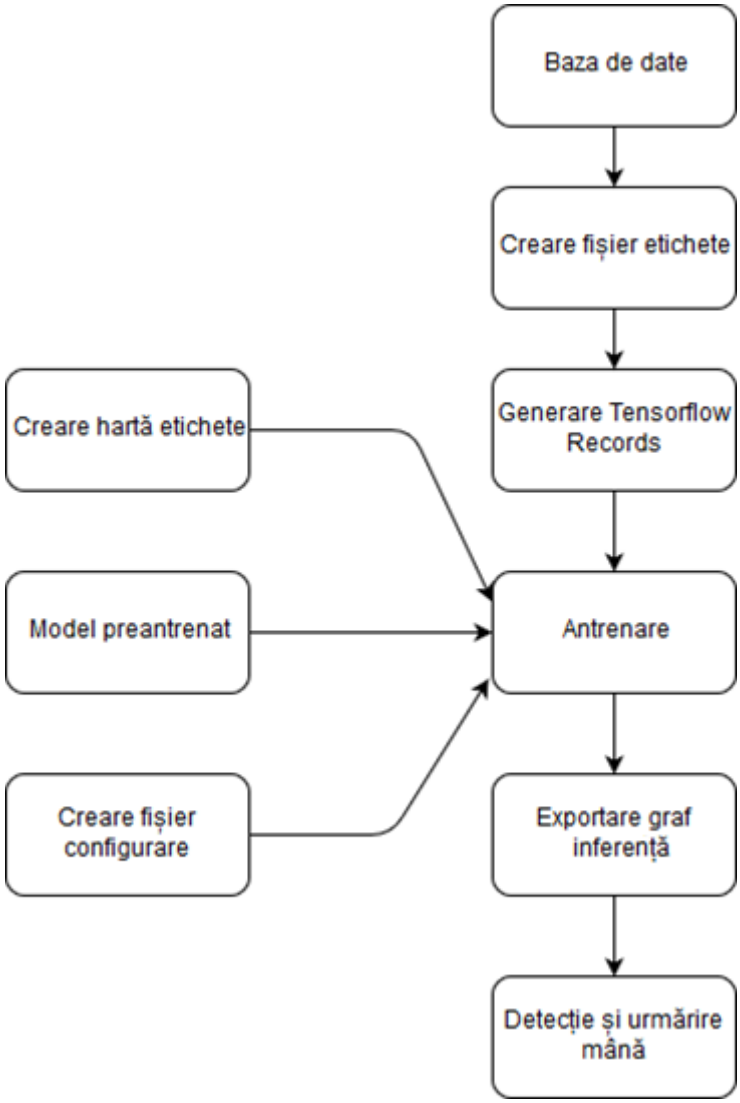


Figura 2.12. Schema bloc a detecției mâinii folosind Tensorflow Object Detection API

După cum am menționat și în secțiunea precedentă, baza de date folosită pentru acest experiment este una publică, și anume EgoHands dataset. Aceasta vine deja cu pozele etichetate, însă noi trebuie să salvăm aceste etichete în format .csv, pentru a le transforma mai apoi în fișiere de tip Tensorflow records (.record). Acesta este un format propriu Tensorflow de stocare binară. Dacă lucrăm cu baze de date mari, timpul de antrenare poate fi îmbunătățit prin utilizarea unui fișier de stocare binar. Acesta ocupă mult mai puțin spațiu, se copiază mai rapid și poate fi citit mult mai eficient de sistem. Pe lângă avantajul performanței, formatul TFRecord poate combina cu ușurință mai multe seturi de date și se integrează foarte ușor în funcționalitățile de importare și preprocesare de date integrate în librăria Tensorflow.

Procesul de antrenare necesită mai multe resurse: un fișier de configurare pipeline, fișierele TFRecord, o hartă cu etichete și modelul pe care vrem să îl reantrenăm. Fișierele TFRecord sunt cele despre care am discutat anterior. Harta cu etichete mapează fiecare etichetă utilizată unei valori întregi. În cazul nostru, nu vrem să detectăm decât mâna deci vom avea o singură etichetă cu valoarea

1. Fișierele ce conțin harta etichetelor au extensia .pbtxt. Mai jos putem vedea conținutul fișierului nostru:

```
item{
        id: 1
        name: 'hand'
 }
```

Pentru antrenare vom utiliza un model pre-antrenat oferit de Tensorflow. Tensorflow oferă mai multe modele, antrenate pe seturile de date COCO, Kitti, Open Images, AVA v2.1 și setul de date iNaturalist Species Detection. În tabelul 2.1 avem denumirile modelelor antrenate pe setul de date COCO (o bază de date foarte mare pentru detecție de obiecte). Pe lângă numele acestora, sunt listate viteza modelului, raportată ca timpul de execuție în ms pentru o imagine de dimensiuni 600x600, dar și performanța de detecție specificată în mAP (mean Average Precision).

Vom alege un model din această listă și îl vom descărca din baza de date cu modele Tensorflow. În continuare trebuie să realizăm un fișier de configurare pentru modelul descărcat. Un template pentru acest fișier vine odată cu modelul, noi trebuie să precizăm numărul de clase, numele modelului pre-antrenat, căile către fișierul TFRecord și către harta de etichete. După care vom putea începe antrenarea. Deoarece modelele sunt complexe și baza de date destul de mare, acest proces va consuma destule resurse și va dura destul de mult timp. Odată ce s-a terminat antrenarea, vom avea niște fișiere checkpoint în folderul de antrenare pe care le vom transforma în graf de inferența (inference graph). Acesta va fi folosit mai departe în detecție.

Model name	Speed (ms)	COCO mAP[^1]
ssd_mobilenet_v1_coco	30	21
ssd_mobilenet_v1_0.75_depth_coco	26	18
ssd_mobilenet_v1_quantized_coco	29	18
ssd_mobilenet_v1_0.75_depth_quantized_coco	29	16
ssd_mobilenet_v1_ppn_coco	26	20
ssd_mobilenet_v1_fpn_coco	56	32
ssd_resnet_50_fpn_coco	76	35
ssd_mobilenet_v2_coco	31	22
ssd_mobilenet_v2_quantized_coco	29	22
ssdlite_mobilenet_v2_coco	27	22
ssd_inception_v2_coco	42	24
faster_rcnn_inception_v2_coco	58	28
faster_rcnn_resnet50_coco	89	30
faster_rcnn_resnet50_lowproposals_coco	64	-
rfcn_resnet101_coco	92	30
faster_rcnn_resnet101_coco	106	32
faster_rcnn_resnet101_lowproposals_coco	82	-
faster_rcnn_inception_resnet_v2_atrous_coco	620	37
faster_rcnn_inception_resnet_v2_atrous_lowproposals_coco	241	-
faster_rcnn_nas	1833	43
faster_rcnn_nas_lowproposals_coco	540	-
mask_rcnn_inception_resnet_v2_atrous_coco	771	36
mask_rcnn_inception_v2_coco	79	25
mask_rcnn_resnet101_atrous_coco	470	33
mask_rcnn_resnet50_atrous_coco	343	29

Tabelul 2.1. Modele de arhitecturi pre-antrenate din Tensorflow (sursa: [28])

Capitolul 3. Experimente și rezultate

În continuare vom prezenta rezultatele obținute în urma experimentelor efectuate. După cum am menționat și în capitolul anterior, am efectuat mai multe experimente pentru a analiza eficiența metodelor de detecție.

Dispozitivul pe care au fost efectuate toate experiementele este un notebook cu următoarele caracteristici: procesor Intel Core i7-7700HQ @ 2.80GHz, 12 GB memorie RAM și procesor grafic NVIDIA GeForce GTX 1050 4GB. De asemenea, am instalat varianta pentru GPU a librăriei Tensorflow și CUDA toolkit oferit de NVIDIA ce oferă suport pentru calculul cu ajutorul procesorului grafic.

În primă fază, am dorit ca sistemul nostru să detecteze dacă există sau nu o mână în imagine, fără să estimeze și poziția acesteia. În acest caz, vom avea o problemă de clasificare cu două clase: o clasă etichetată cu 1 corespunzătoare imaginilor cu mâini și una etichetată cu 0 corespunzătoare imaginilor fără mâini. Pentru acest scop am utilizat o bază de date proprie, compusă din 390 de imagini cu mâini și 127 de imagini fără (ulterior am ajuns la 150 de imagini). Arhitectura de rețea abordată pentru această etapă este următoarea:

a)

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 98, 98, 8)	224
max_pooling2d_1 (MaxPooling2	(None, 49, 49, 8)	0
conv2d_2 (Conv2D)	(None, 47, 47, 12)	876
max_pooling2d_2 (MaxPooling2	(None, 23, 23, 12)	0
conv2d_3 (Conv2D)	(None, 21, 21, 16)	1744
max_pooling2d_3 (MaxPooling2	(None, 10, 10, 16)	0
flatten_1 (Flatten)	(None, 1600)	0
dense_1 (Dense)	(None, 128)	204928
dense_2 (Dense)	(None, 1)	129

Total params: 207,901
Trainable params: 207,901
Non-trainable params: 0

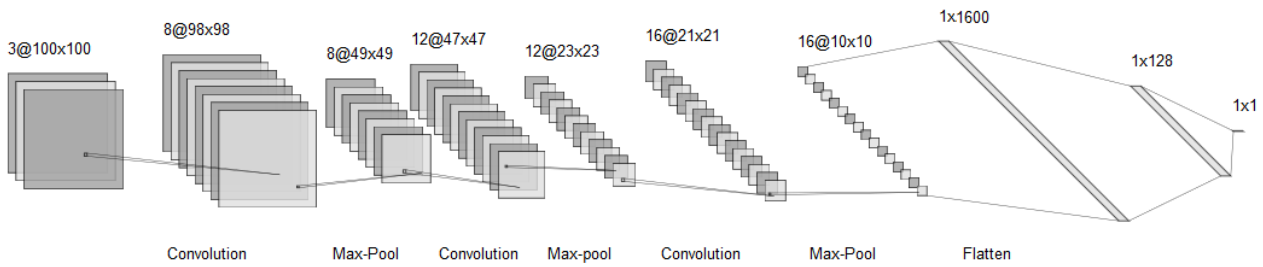


Figura 3.1. a) Sumarul modelului antrenat pentru detecția prezenței mâinii; b) arhitectura rețelei

Am făcut antrenarea acestei rețele (vezi Anexa 1) pe 100 de epoci și am obținut o minimizare a funcției cost până la valoarea de 2.6754×10^{-4}. De asemenea, imaginile au fost împărțite în două foldere, 90% din imagini au fost folosite pentru antrenare și 10% pentru testare. Pe imaginile de test,

am obținut o eroare de 0.3437. Pe parcursul antrenării am monitorizat procesul cu ajutorul pachetului Tensorboard.

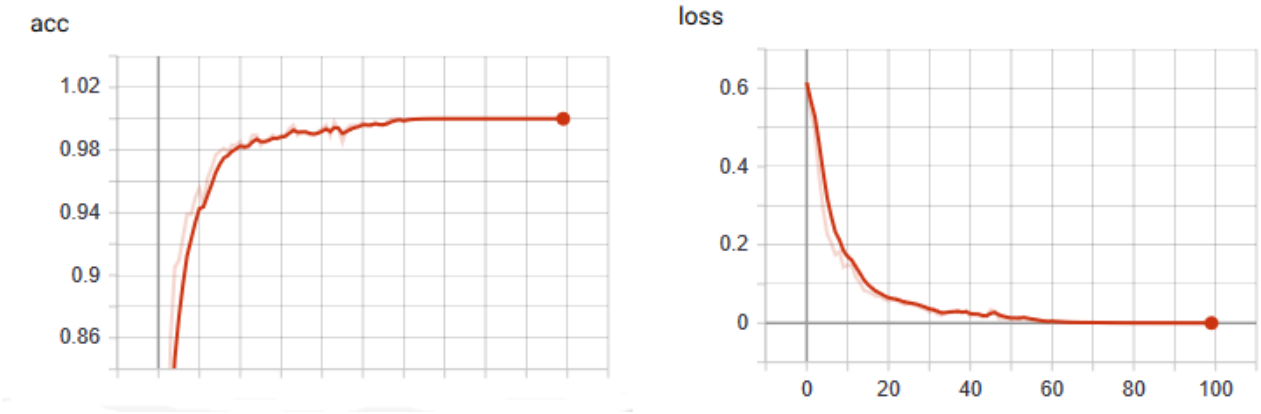


Figura 3.2. Evoluția costului și a acurateței pe parcursul antrenării

Modelul astfel antrenat a fost salvat și utilizat ulterior în partea de detecție în timp real (vezi Anexa 2). Pentru aceasta am realizat un program care captează în timp real fluxul video de la camera web a calculatorului. Am setat parametrul FPS (frames per second – număr de cadre pe secundă) la 15. Fiecare cadru captat devine input pentru modelul salvat de noi anterior și va fi prezisă o valoare. Pe stratul final avem un singur neuron cu funcție de activare sigmoidă, ceea ce înseamnă că valoarea noastră de ieșire va fi între 0 și 1. Decizia se face astfel: dacă valoarea de ieșire este mai mare ca 0.5, atunci decidem că este o mână în imagine și afișam mesajul *"DA"* în colțul stânga sus. În caz contrar, dacă valoarea de ieșire este mai mică decât 0.5, atunci nu există nicio mână în imagine și vom afișa mesajul *"NU"*. Se poate observa o demonstrație în figura 3.3.

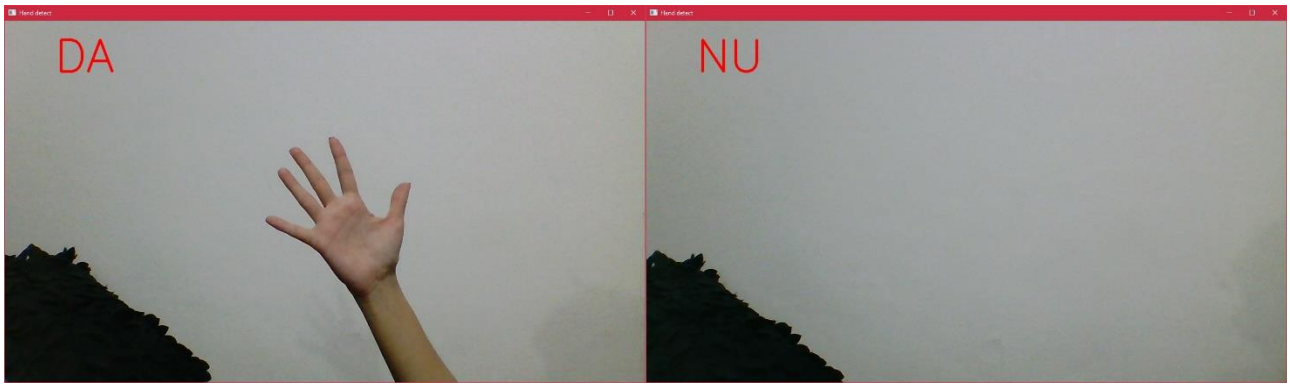


Figura 3.3. Detecția de mână în timp real

Pentru evaluare performanței modelului vom prezenta în tabelul 3.1 numărul de imagini clasificate corect și numărul de imagini clasificate greșit. De menționat că imaginile din baza de test sunt foarte similare cu cele din baza de antrenare, acestea fiind realizate în aceleași condiții externe. Se observă că 88% din imaginile de test au fost clasificate corect.

Clasă	**Nr. imagini**	**Nr. imagini clasificate corect**	**Nr. imagini clasificate gresit**
HAND	30	30	0
NON_HAND	12	7	5
TOTAL	42	37	5

Tabelul 3.1. Rezultatele obținute în urma clasificării imaginilor

În continuare ne dorim să prezicem și poziția mâinii, de aceea am modificat baza de date, păstrând doar imaginile cu mâini și etichetând imaginile cu bounding boxes. De asemenea, am modificat și modelul rețelei, pentru acest set de date abordând o altă arhitectură care se poate observa în figura 3.4, respectiv 3.5. Deoarece aceasta este o problemă de regresie liniară, la ieșire nu vom mai avea un singur neuron ci 4, corespunzător celor 4 ieșiri: (x1,y1) – coordonatele colțului stânga sus al dreptunghiului și (x2,y2) – coordonatele colțului dreapta jos al dreptunghiului.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 98, 98, 96)	2688
max_pooling2d_1 (MaxPooling2	(None, 49, 49, 96)	0
conv2d_2 (Conv2D)	(None, 47, 47, 256)	221440
max_pooling2d_2 (MaxPooling2	(None, 23, 23, 256)	0
conv2d_3 (Conv2D)	(None, 21, 21, 384)	885120
conv2d_4 (Conv2D)	(None, 19, 19, 384)	1327488
conv2d_5 (Conv2D)	(None, 17, 17, 256)	884992
max_pooling2d_3 (MaxPooling2	(None, 8, 8, 256)	0
flatten_1 (Flatten)	(None, 16384)	0
dense_1 (Dense)	(None, 100)	1638500
dropout_1 (Dropout)	(None, 100)	0
dense_2 (Dense)	(None, 100)	10100
dropout_2 (Dropout)	(None, 100)	0
dense_3 (Dense)	(None, 100)	10100
dense_4 (Dense)	(None, 4)	404

Total params: 4,980,832
Trainable params: 4,980,832
Non-trainable params: 0

Figura 3.4. Sumarul modelului rețelei neurale folosite pentru urmărirea mâinii

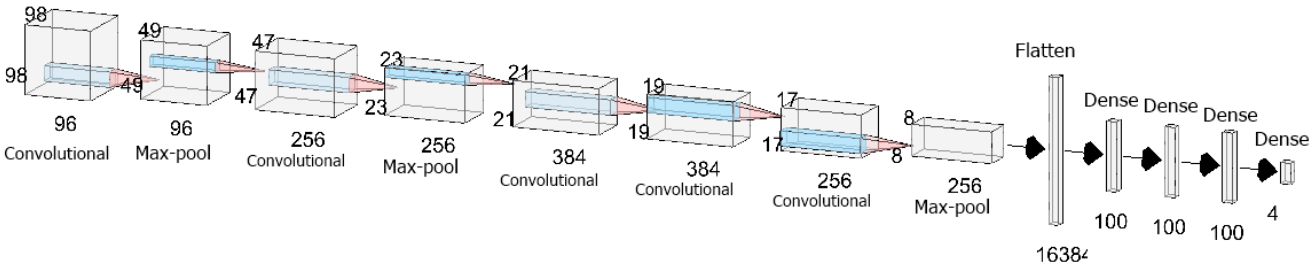


Figura 3.5. Arhitectura rețelei neurale folosita pentru urmărirea mâinii

Pentru acest model am realizat antrenarea pe 100 de epoci (vezi Anexa 3) și am obținut o minimizare a costului la o valoare 0.0037. În figura 3.6 putem vedea evoluția funcției cost pe parcursul procesului de antrenare.

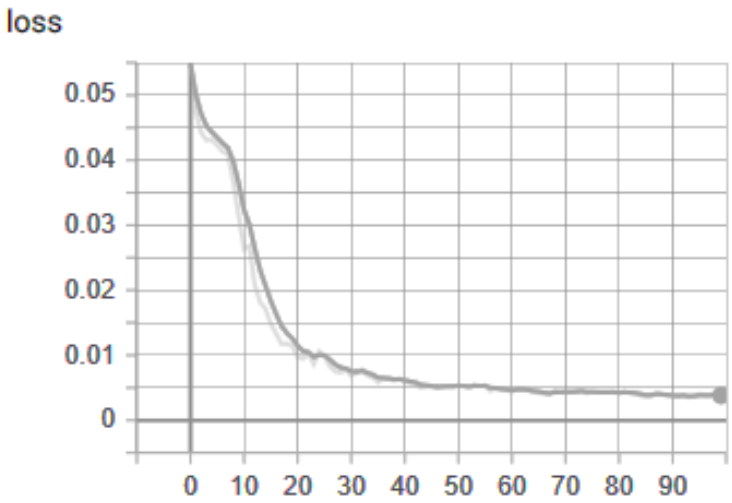


Figura 3.6. Minimizarea funcției cost în procesul de antrenare

De asemenea, am folosit și de această dată 10% din baza de date pentru testare. Pentru imaginile de test am obținut o eroare de 0.00283. Pentru a verifica corectitudinea coordonatelor prezise, am făcut un test și am analizat comparativ dreptunghiul etichetat de mine și dreptunghiul prezis de rețeaua neurală. Această comparație este ilustrată în figura 3.8. În stânga este poligonul etichetat de noi, iar în partea dreaptă este dreptunghiul obținut din coordonatele prezise de model. Pe aceste imagini am calculat performanța sub formă de IOU (Intersection Over Union). Acest parametru reprezintă raportul dintre aria de intersecție și aria totală a celor două dreptunghiuri (după cum vedem în figura 3.7). Pentru imaginile noastre de test, valoarea IoU este 0.89.

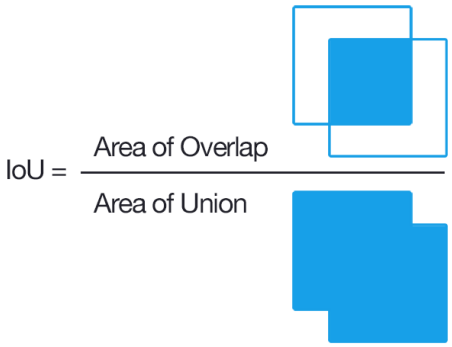


Figure 3.7. Ilustrarea parametrului IoU (sursa: [29])

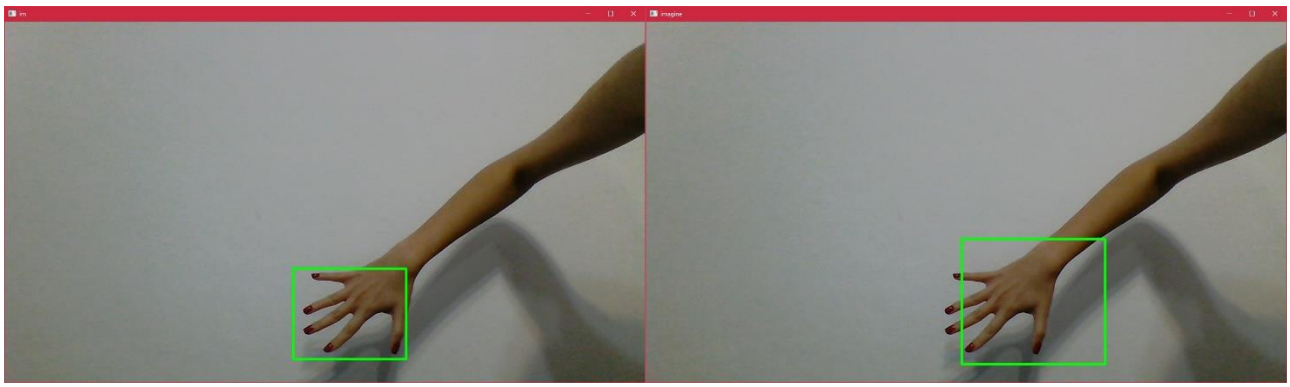


Figura 3.8. Comparație între ieșirea dorită și ieșirea reală

Mai departe, am salvat modelul pentur a-l integra în aplicația de urmărire în timp real. La fel ca la pasul anterior, am setat ca fluxul video să fie captat cu 15 FPS și am pornit achiziția (vezi Anexa 4). Fiecare cadru reprezintă un input către modelul salvat de noi anterior.

Am observat că dreptunghiul apărea în permanență și într-un mod haotic pe ecran atunci când mâna lipsea. Acest lucru se întâmpla din cauza faptului că modelul nostru a fost antrenat doar cu imagini ce conțineau mâini, iar în momentul în care mâna dispărea din ecran acesta continua să prezică fără să știe ca acolo, de fapt, nu se află nicio mână. Pentru a rezolva acest inconvenient am integrat și modelul antrenat în prima parte, cel de detecție a mâinii. Astfel, imaginea era mai întai analizată cu ajutorul primului model, care decidea dacă există sau nu o mână în cadru. Dacă exista, atunci imaginea era analizată de cel de-al doilea model care prezicea un set de coordonate și desena un dreptunghi pe ecran ce încadra mâna. Această problemă ar fi putut fi rezolvată și dacă în baza de date includeam imagini goale, care nu conțineau mâini, și le etichetam cu coordonatele (0,0), (0,0). În figura 3.9 putem vedea un exemplu de predicție în timp real:

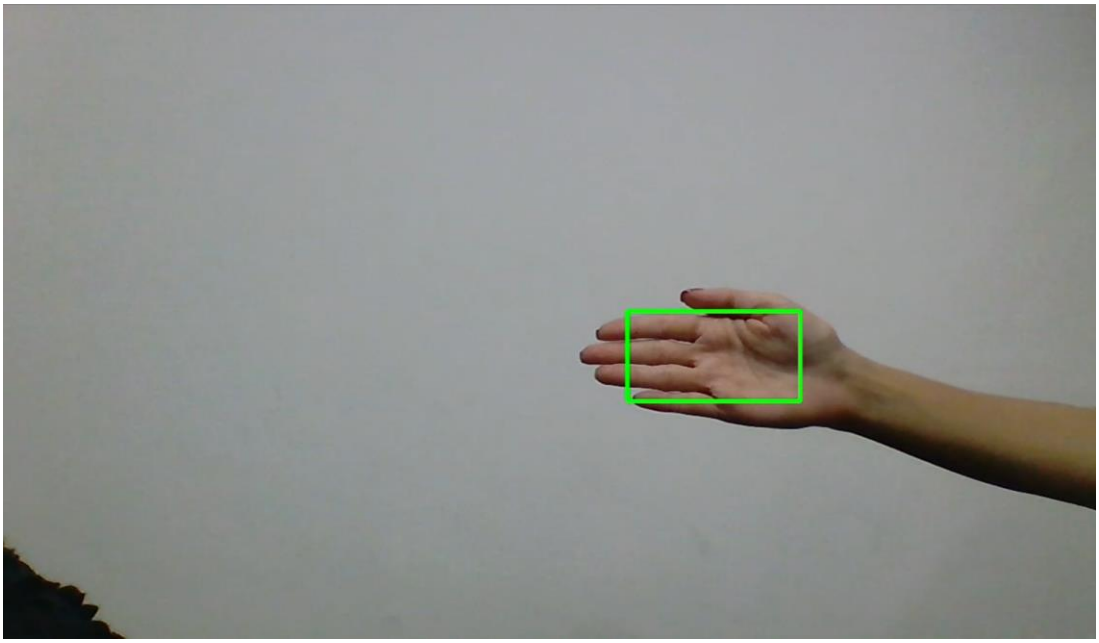


Figura 3.9. Exemplu de urmărire a mâinii în timp real

Se poate observa că acesta a detectat că este o mână în cadru, dar și poziția acesteia. Un detaliu foarte important care trebuie menționat este că până în acest moment arhitecturile de rețea au fost implementate cu ajutorul librăriei Keras în limbajul de programare Python.

În cea de-a treia etapă a proiectului am vrut să testăm funcționalitatea unui tool open-source: Tensorflow Object Detection API. Detaliile de implementare ale aplicației cu ajutorul acestui tool au fost detaliate în capitolul anterior.

După cum am menționat, Tensorflow oferă niște modele pre-antrenate pe baze de date foarte mari, special pentru detecția de obiecte. Acestea au fost evaluate și sunt disponibile în librărie. Ceea ce vom face în cadrul acestui experiment este să reantrenăm unul dintre aceste modele pe un set de date cu mâini și să realizăm o aplicație care face detecția mâinii. Acest procedeu se numește învățare prin transfer (transfer learning). Un model pre-antrenat este o rețea salvată care a fost antrenată anterior pe o bază de date foarte mare, pentru o sarcină de clasificare a imaginilor. Putem fie să folosim modelul pre-antrenat așa cum este el, fie să folosim învățarea prin transfer pentru a particulariza modelul pentru un anumit task.

Procedeul transfer learning înseamnă de fapt să păstrăm nemodificate straturile convoluționale (cele care realizează extragerea și selecția de trăsături), adică să le "înghețăm" astfel încât ponderile acestora să nu se modifice în procesul de antrenare. Apoi, ceea ce vom face este să adăugăm un clasificator la finalul rețelei și să îl antrenăm doar pe acesta pentru sarcina noastră.

De asemenea, pentru a îmbunătăți performanțele aplicației, se pot ajusta fin ponderile din straturile convoluționale. Acest proces trebuie făcut doar după ce s-a antrenat clasificatorul de la finalul rețelei și, de asemenea, trebuie să se modifice ponderile doar pentru un număr mic de straturi la un moment dat. Noi nu vom realiza acest pas.

În continuare, am ales să reantrenăm modelul *ssd_mobilenet_v1_coco*. SSD – Single Shot Multibox Detector – este un algoritm foarte popular în domeniul detecției de obiecte, fiind mai rapid decât arhitectura Faster RCNN. Arhitectura SSD are la bază una dintre cele mai cunoscute arhitecturi de rețele neurale convoluționale – VGGnet. Denumirea arhitecturii vine de la: Single Shot – înseamnă că localizarea obiectului și clasificarea se face într-un singur pas direct în rețea, MultiBox – este numele unei regresii de tip bounding box dezvoltată de Szegedy și alții, Detector – rețeaua este un detector de obiecte care totodată clasifică obiectele detectate. [30]

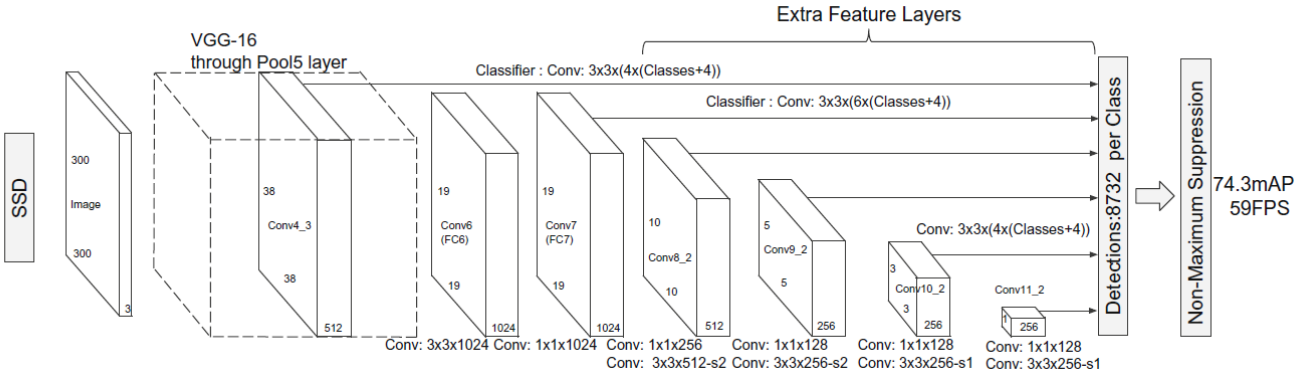


Figura 3.10. Arhitectura rețelei SSD (sursa: [31])

Pentru dezvoltarea acestei rețele s-a folosit ca bază rețeaua VGG-16, peste care s-a adăugat o structură suplimentară, printre care straturi convoluționale la sfârșitul rețelei ce permit detecție la o scală variată. [31]

După ce am trecut prin toți pașii descriși în secțiunea *Implementare*, am pornit antrenarea. Am rulat antrenarea pe GPU și fiecare pas a durat aproximativ o secundă. Deși am pornit procesul de antrenare pe GPU, arhitectura rețelei fiind foarte complexă și setul de date foarte mare, această etapă a solicitat o putere de calcul foarte mare, resursele dispozitivului fiind utilizate la capacitate aproape maximă. În figura 3.11 se poate observa monitorizarea procesului de antrenare.

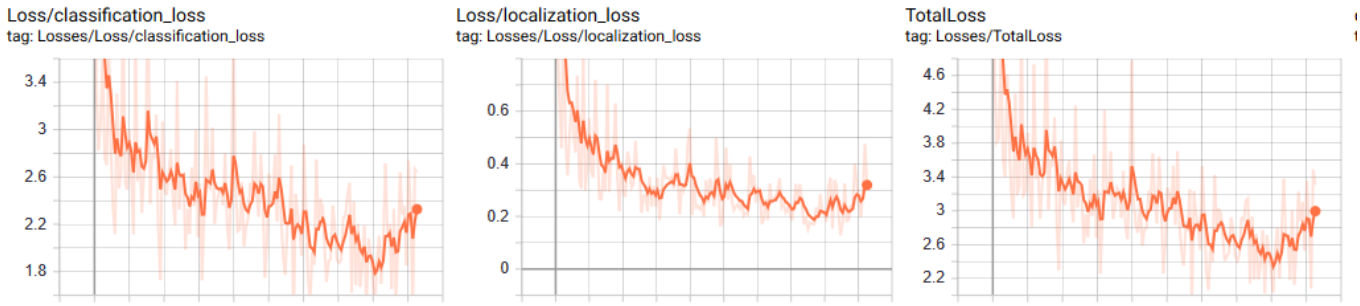


Figura 3.11. Monitorizarea procesului de antrenare a rețelei SSD

După cum se poate observa, funcția cost în acest caz nu mai are o evoluție la fel de constantă precum în cazurile anterioare. Variațiile sunt destul de mari. Eroarea de clasificare a fost minimizată la aproximativ 2, eroarea de localizare la ~0.3, iar costul total la o valoare în jurul lui 2.5. Aceste valori sunt din momentul în care am oprit procesul de antrenare, adică la pasul 18515. Odată ce am terminat antrenarea, pentru a putea folosi API-ul oferit de tensorflow, am exportat graful de inferența și apoi l-am utilizat în aplicația de detecție.

În afara faptului că acest model de rețea este cu mult mai complex decât cele anterioare, iar bazele de date pe care a fost antrenat sunt foarte variate, un avantaj foarte important al acestui API este că poate detecta mai multe obiecte în același timp. Ceea ce înseamnă că vom detecta mai multe mâini în imagine. Aceasta este o îmbunătățire considerabilă dat fiind faptul că până acum noi am antrenat modelul nostru doar pentru a recunoaște o singură mână la un moment dat. Exemplu de detecție în timp real poate fi analizat în figura 3.12.

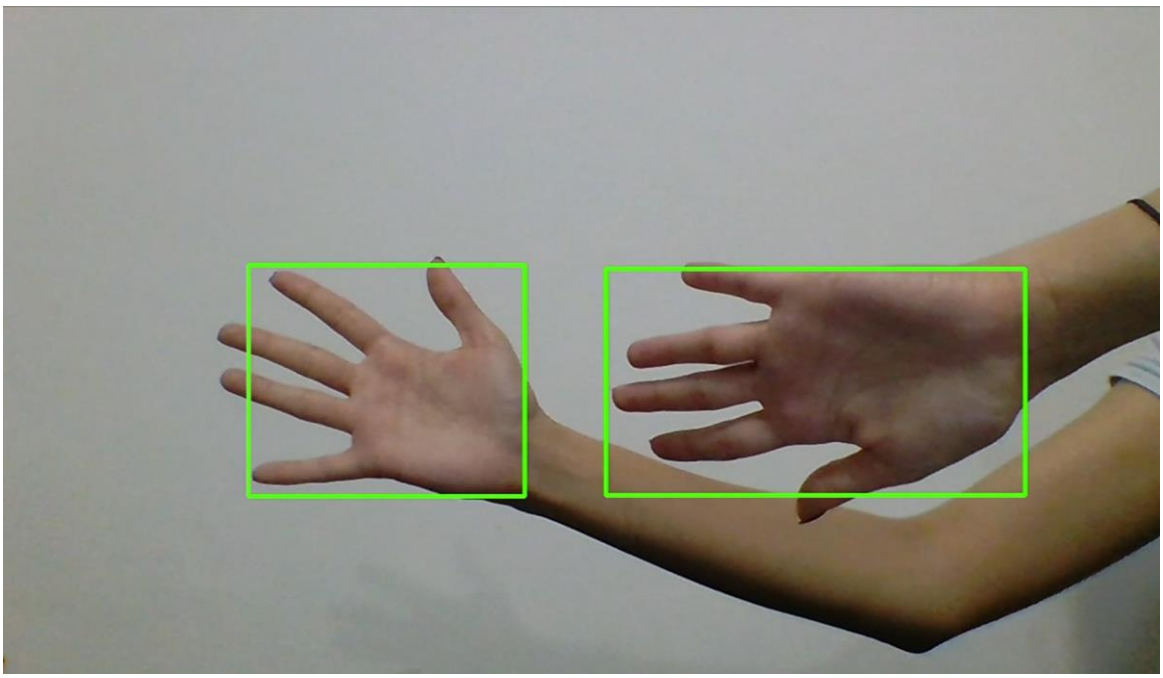


Figura 3.12. Urmărirea mâinilor în timp real folosind Tensorflow Object Detection API

Rezultatele obținute cu această metodă sunt semnificativ mai bune. Nu numai că poate urmări cursiv mâna pe ecran, dar totodată putem urmări chiar mai multe mâini în același timp. De asemenea, se observă diferența procesului de antrenare. Dacă pentru arhitecturile mai simple din prima parte puteam minimiza funcția cost la o valoare sub 1 în 100 de epoci, observăm că în acest caz, dupa 18

515 pași nu am putut minimiza eroarea decât până la o valoare apropiată de 2. Chiar și așa, cu această valoare finală, performanțele sunt foarte bune, sistemul îndeplinind cu succes sarcina pentru care a fost proiectat. O concluzie ar fi că valoarea la care eroarea trebuie minimizată este relativă, depinzând puternic de complexitatea modelului. Dacă pentru un model simplu funcția cost trebuie minimizată la o valoare subunitară, în cazul modelelor complexe o eroare supraunitară nu înseamnă performanțe scăzute, ci rezultatele pot fi chiar foarte bune.

Ulterior, am vrut să continuăm antrenarea modelului SSD pentru a-i evalua performanțele, însă se poate observa în figura 3.13 că funcția cost nu se minimiza iar timpul necesar pentru fiecare pas a început să crească, ceea ce însemna că ar fi durat mult mai mult să ajungem la o valoare considerabil mai bună.

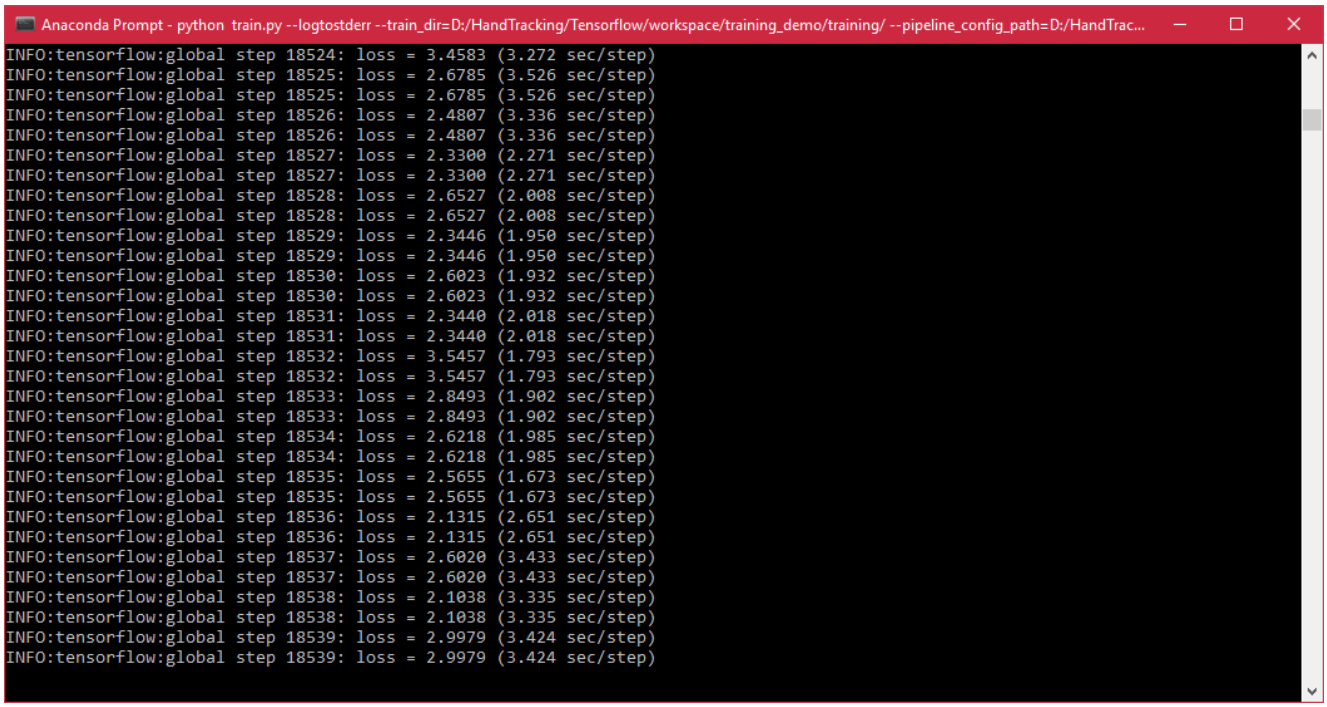


Figura 3.13. Captură din timpul antrenării rețelei SSD

Concluzii

Concluzii generale

Scopul acestui proiect a fost acela de a realiza un sistem care să urmărească mâna în timp real într-o captură realizată cu ajutorul camerei web. Am reușit să implementăm două arhitecturi de rețele neurale și să le testăm pe fluxul video live. De asemenea, am testat și funcționalitatea unei aplicații open source ce ne-a oferit posibilitatea să antrenăm o arhitectură de rețea mult mai complexă, pe o bază de date publică mult mai mare. De asemenea, am realizat o bază de date proprie și am efectuat manual etichetarea imaginilor cu bounding boxes.

Una dintre concluziile esențiale ale acestui proiect este că rețelele neurale convoluționale au un potențial uriaș în domeniul prelucrării de imagini. Acestea pot atinge performanțe pe care nu le putem obține cu niciun alt algoritm de prelucrare dacă alegem un set de date suficient de mare și de o calitate foarte bună. Rețelele neurale convoluționale sunt rețele de tip deep (cu învățare profundă), dar specializate pentru anumite tipuri de intrări, și anume imagini. În interiorul unei astfel de rețele pot fi mai multe tipuri de straturi. Acestea se împart în 2 mari categorii: învățare de trăsături și clasificare. Partea de învățare de trăsături se realizează cu ajutorul straturilor convoluționale și a celor de pooling, iar în partea de clasificare avem straturi complet conectate. Arhitectura CNN este inspirată din organizarea cortexului vizual și este asemănătoare cu tiparul de conexiuni dintre neuronii din creierul uman.

Am demonstrat că rețelele CNN pot fi un clasificator foarte bun pentru task-ul de detecție a mâinii și că rezultatele pot fi foarte bune. Această aplicație poate fi foarte utilă în aplicații interactive. Atât pentru copii, cât și pentru bătrâni, controlul cu ajutorul mâinii ar reprezenta o simplificare primordială a vieții cotidiene și a interacțiunii acestora cu dispozitivele inteligente. De asemenea, poate fi utilizată și pentru divertisment, întrucât își poate găsi aplicabilitate în foarte multe jocuri atât pe calculator, cât și pe dispozitive mobile sau console.

Contribuții personale

Pentru acest proiect, contribuțiile mele personale au fost următoarele:

- crearea unei baze de date proprii cu ~500 imagini
- etichetarea manuală a imaginilor cu bounding boxes
- implementarea și antrenarea unui model ce determină prezența unei mâini
- implementarea și antrenarea unui model ce realizează urmărirea mâinii dintr-un flux video
- reantrenarea unui model pre-antrenat oferit de Tensorflow
- realizarea unui sistem de detecție și urmărire a mâinii cu ajutorul Tensorflow Object Detection API

Dezvoltări viitoare

Detecția și urmărirea mâinii este un proiect ce poate fi dezvoltat în mai multe direcții. În primul rând, pe viitor îmi doresc să testez și alte modele pre-antrenate de la Tensorflow pentru a analiza și performanțele lor. De asemenea, există și alte baze de date publice cu mâini ce pot fi utilizate. Universitatea Oxford are o astfel de bază de date, ce conține imagini cu mâini din diferite surse publice, chiar și din filme precum “Apollo 13” sau “Forrest Gump”. Etichetele acestor imagini sunt de asemenea niște dreptunghiuri (bounding rectangles).

O altă direcție în care îmi doresc să dezvolt acest proiect este extinderea bazei de date proprii. Aceasta conține la momentul actual ~500 imagini. Îmi doresc extinderea bazei de date la cel puțin 1000 de imagini. De asemenea, această extindere ar implica și modificarea etichetelor, întrucât dreptunghiurile etichetate la momentul actual nu reprezintă cea mai bună încadrare a mâinilor. Pe de altă parte, imaginile actuale conțin o singură mână, pe viitor am putea adăuga imagini cu mai multe mâini.

Una dintre îmbunătățirile esențiale care ar putea fi aduse acestui proiect este cea de modificare a arhitecturii rețelei. Pentru acest task este nevoie de un model mai complex, mai performant, care să poată să funcționeze în orice condiții și chiar pentru camere video de calitate mai slabă. De asemenea, modelul actual este realizat pentru detecția unei singure mâini. Pentru aplicații mai complexe, trebuie implementată detecția unui număr mai mare de mâini ce apar într-un cadru.

Desigur, cea mai importantă dezvoltare ar fi integrarea detecției într-o aplicație interactivă. Fie că vorbim de un joc sau de controlul unui dispozitiv, controlul cu ajutorul mâinii poate fi un real ajutor în viața noastră de zi cu zi.

BIBLIOGRAFIE

[1] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard și L. D. Jacke „Backpropagation Applied to Handwritten Zip Code Recognition," *Neural computation,* vol. 1 pp. 541--551, 1989.

[2] S. Bambach, S. Lee, D. J. Crandall și C. Yu, „Lending a hand: Detecting hands and recognizin activities in complex egocentric interactions," *Proceedings of the IEEE International Conferenc on Computer Vision,* pp. 1949--1957, 2015.

[3] K. Gurney, An Introduction to Neural Networks, London: CRC Press, 1997.

[4] P. Amar, F. Kepes și V. Norris, Proceedings of The 2012 Evry Spring School on Modellin Complex Biological Systems in the Context of Genomics, Mai, 2012.

[5] R. Rojas, Neural networks: a systematic introduction., Berlin, 1996.

[6] J. M. Zurada, Introduction to artificial neural systems, West publishing company St. Paul, 199

[7] S. Haykin, Neural Networks and Learning Machines, 3 ed., Pearson education Upper Saddl River, 2009.

[8] R. Fuller, 29 August 2011. http://uni-obuda.hu/users/fuller.robert/perception.pdf. [Accesat 10 0 2019].

[9] S. Raschka, 2013 https://sebastianraschka.com/faq/docs/diff-perceptron-adaline-neuralnet.htm [Accesat 11 06 2019].

[10] D. Lucas, „UNIT-I INTRODUCTION TO ARTIFICIAL NEURAL NETWORK," 2014. https://slideplayer.com/slide/252461/. [Accesat 10 06 2019].

[11] B. Kröse și P. van der Smagt, An introduction to neural networks (Eighth edition), University o Amsterdam, 1996.

[12] I. Witten, E. Frank, M. Hall și C. Pal, „Deep learning," în *Data mining,* Morgan Kaufmann, 201 pp. 417-466.

[13] S. Sharma, „Activation Functions in Neural Networks," 6 September 201 https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6. [Accesa 19 06 2019].

[14] C. McDonald, „Machine learning fundamentals (I): Cost functions and gradient descent," 2 November 2017. https://towardsdatascience.com/machine-learning-fundamentals-via-linear regression-41a5d11f5220. [Accesat 10 06 2019].

[15] B. Graham, 2014. http://www.cs.stir.ac.uk/courses/CSC9YF/lectures/ANN/3-DeltaRule.pd [Accesat 10 06 2019].

[16] Prof. Dr. Ing. Ovidiu Grigore, „Note de curs RNSF," 2019. http://ai.pub.ro/. [Accesat 11 0 2019].

[17] R. Nash și K. O'Shea, „An introduction to convolutional neural networks," *arXiv preprir arXiv:1511.08458,* 2015.

[18] Stanford University, „Convolutional Neural Networks for Visual Recognition, http://cs231n.github.io/convolutional-networks/. [Accesat 11 03 2019].

[19] „Deep Learning With Dataiku Data Science Studio,”. https://blog.dataiku.com/deep-learning with-dss. [Accesat 20 06 2019].

[20] D. A Shoieb, S. Youssef și W. Aly, „Computer-Aided Model for Skin Diagnosis Using Dee Learning,” *Journal of Image and Graphics*, vol. 4, pp. 116-121, December 2016.

[21] T. Sik-Ho, „Review: ZFNet — Winner of ILSVRC 2013 (Image Classification),” 19 Augus 2018. https://medium.com/coinmonks/paper-review-of-zfnet-the-winner-of-ilsvlc-2013-image classification-d1a5a0c45103. [Accesat 20 06 2019].

[22] T. Sik-Ho, „Review: GoogLeNet (Inception v1)— Winner of ILSVRC 2014 (Imag Classification),” 24 August 2018. https://medium.com/coinmonks/paper-review-of-googlenet inception-v1-winner-of-ilsvlc-2014-image-classification-c2b3565a64e7. [Accesat 20 06 2019].

[23] „VGG16 – Convolutional Network for Classification and Detection,” 20 Noiembrie 201 https://neurohive.io/en/popular-networks/vgg16/. [Accesat 20 06 2019].

[24] P. Ruiz, „Understanding and visualizing ResNets,” 8 Octombrie 2018. https://towardsdatascience.com/understanding-and-visualizing-resnets-442284831be8. [Accesa 20 06 2019].

[25] „Speed/accuracy trade-offs for modern convolutional object detectors,” în *Proceedings of th IEEE conference on computer vision and pattern recognition*, 2017, pp. 7310-7311.

[26] V. Dibia, „HandTrack: A Library For Prototyping Real-time Hand TrackingInterfaces usin Convolutional Neural Networks,” 2017. https://github.com/victordibia/handtracking. [Accesa 10 06 2019].

[27] A. KNUTSSON, *Hand Detection and Pose Estimation using Convolutional Neural Network* KTH Royal Institute of Technology, Sweden.

[28] „Tensorflow detection model zoo,”.: https://github.com/tensorflow/models/blob/master/

research/object_detection/g3doc/detection_model_zoo.md. [Accesat 15 06 2019].

[29] A. Rosebrock, „Intersection over Union (IoU) for object detection,” https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/. [Accesat 25 06 2019].

[30] E. Forson, „Understanding SSD MultiBox—Real-Time Object Detection In Deep Learning,” 1 Noiembrie 2017. https://towardsdatascience.com/understanding-ssd-multibox-real-time-object detection-in-deep-learning-495ef744fab. [Accesat 20 06 2019].

[31] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu și A. C. Berg, „Ssd: Single sho multibox detector,” în *European conference on computer vision*, Springer, 2016, pp. 21-37.

Anexa 1. Codul sursă pentru antrenarea rețelei de detecție a mâini

```
#importarea pachetelor de care vom avea nevoie
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np
import os
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Flatten
from keras.layers import Dense
from keras.callbacks import TensorBoard
from time import time

#calea către directorul cu imagini de antrenare cu mâini
trainHandPath = 'train/hand'
#calea către directorul cu imagini de antrenare fără mâini
trainNonHandPath = 'train/nonhand'
#calea către directorul cu imagini de testare cu mâini
testHandPath = 'test/hand'
#calea către directorul cu imagini de testare fără mâini
testNonHandPath = 'test/nonhand'

#Vom stoca numărul de vectori de antrenare și testare în patru variabile
trainHandData = len(os.listdir(trainHandPath))
trainNonHandData =len(os.listdir(trainNonHandPath))
testHandData = len(os.listdir(testHandPath))
testNonHandData = len(os.listdir(testNonHandPath))

print("Număr vectori antrenare: Mână -"+str(trainHandData)+"; Fără mână -
"+str(trainNonHandData))
print("Număr vectori testare: Mână -"+str(testHandData)+"; Fără mână -
"+str(testNonHandData))

#inițializăm o matricea ce va conține vectorii de intrare în rețea cu 0
trainData=np.zeros([trainHandData+trainNonHandData,100,100,3])

#pe primele linii ale matricei trainData vom salva imaginile de antrenare cu mâini
for index, filename in enumerate(os.listdir(trainHandPath)):
    picture = Image.open(os.path.join(trainHandPath,filename))
    picture = picture.resize((100,100),Image.ANTIALIAS)
    im = np.array(picture)
    trainData[index,:,:,:]=im

#pe următoarele linii ale matricei trainData vom salva imaginile de antrenare fără mâini
for index,filename in enumerate(os.listdir(trainNonHandPath)):
    picture = Image.open(os.path.join(trainNonHandPath,filename))
    picture = picture.resize((100,100),Image.ANTIALIAS)
    im=np.array(picture)
    trainData[trainHandData+index,:,:,:]=im

#imaginile sunt reprezentate în RGB, pixelii au valori între 0 și 255; împărțim la 255
pentru a obține o valoare între 0 și 1
trainData = trainData/255

#definim matricea cu etichete; acestea vor fi 1 pentru imaginile cu mâini și 0 pentru cele
fără
trainLabels=np.array([])
trainLabels=np.append(np.append(trainLabels,[1]*trainHandData),[0]*trainNonHandData)

#definim modelul rețelei neurale
model = Sequential()
model.add(Conv2D(8,(3,3),input_shape=(100,100,3),activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2),strides=2))
```

```
model.add(Conv2D(12,(3,3),activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2),strides=2))
model.add(Conv2D(16,(3,3),activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(units=128,activation='relu'))
model.add(Dense(units=1,activation='sigmoid'))

#configurăm modelul pentru antrenare
model.compile(optimizer='adam',loss='binary_crossentropy',metrics=['accuracy'])

#inițializăm un obiect de tip TensorBoard pentru a putea monitoriza procesul de antrenare
tensorboard = TensorBoard(log_dir="logs/{}".format(time()))

#pornim procesul de antrenare
model.fit(trainData,trainLabels,epochs=100, verbose=1, callbacks=[tensorboard])

#inițializăm matricea pentru testare cu zerouri
testData=np.zeros([testHandData+testNonHandData,100,100,3])

#stocăm pe primele linii din matricea de test imaginile cu mâini
for index, filename in enumerate(os.listdir(testHandPath)):
    picture = Image.open(os.path.join(testHandPath,filename))
    picture = picture.resize((100,100),Image.ANTIALIAS)
    im = np.array(picture)
    testData[index,:,:,:]=im

#pe următoarele linii vom avea imaginile fără mâini
for index, filename in enumerate(os.listdir(testNonHandPath)):
    picture = Image.open(os.path.join(testNonHandPath,filename))
    picture = picture.resize((100,100),Image.ANTIALIAS)
    im = np.array(picture)
    testData[testHandData+index,:,:,:]=im

#imaginile sunt reprezentate în RGB, pixelii au valori între 0 și 255; împărțim la 255
pentru a obține o valoare între 0 și 1
testData = testData/255

#creăm etichetele pentru imaginile de test, păstrând aceeași convenție: 1 pentru imagini cu
mâini și 0 pentru imagini fără mâini"
testLabels=np.array([])
testLabels=np.append(np.append(testLabels,[1]*testHandData),[0]*testNonHandData)

#folosim imaginile salvate in testData ca intrare în modelul antrenat și salvăm valoarea
prezisă în variabila networkOutput
networkOutput = model.predict(testData)

#evaluăm performanțele modelului
print(model.evaluate(testData,testLabels))
print(model.metrics_names)

#arhitectura rețelei
model.summary()

#salvăm modelul
model.save('handDetectModel.h5')
```

Anexa 2. Codul sursă pentru detecția prezenței mâinii

```
#importăm modulele necesare
import cv2
import numpy as np
from PIL import Image
from keras import models
import time

#încărcam modelul salvat anterior
model = models.load_model('handDetectModel.h5')

#pornim captura de la camera, setăm FPS la 15 și dimensiunile cadrului
videoStream = cv2.VideoCapture(0)
videoStream.set(cv2.CAP_PROP_FPS,15)
videoStream.set(cv2.CAP_PROP_FRAME_WIDTH, 1280)
videoStream.set(cv2.CAP_PROP_FRAME_HEIGHT, 720)

#variabile necesare pentru afișarea pe ecran a predicției
last = 0
check = 1
string = ""

while True:
        #citim un cadru de la captura live
        _, frame = videoStream.read()

        #convertim cadrul în format RGB
        im = Image.fromarray(frame, 'RGB')

        #am antrenat modelul pe imagini 100x100, așa că redimensionăm, apoi împărțim la 255
pentru a obține valori între 0 și 1
        im = im.resize((100,100),Image.ANTIALIAS)
        picture =np.array(im)
        picture=picture/255

        #modelul keras acceptă un tensor 4-dimensional, așa că vom extinde dimensiunile de
la 100x100x3 la 1x100x100x3
        picture = np.expand_dims(picture, axis=0)

        #calculăm ieșirea rețelei pentru cadrul curent
        networkOutput = model.predict(picture)

        #pe stratul de ieșire avem o sigmoidă, așa că decizia de clasificare va fi:
        #- dacă networkOutput>0.5 atunci există o mână în imagine
        #- altfel nu există o mână în imagine
        if networkOutput > 0.5:
            check = 1
        else:
            check = 0

        #verificăm dacă predicția s-a schimbat față de cadrul anterior pentru a afișa alt
mesaj pe ecran
        if check != last:
            if check == 1:
                string = "DA"
            else:
                string = "NU"
        last = check

        #scriem pe cadrul curent predicția rețelei și apoi îl afișăm
        cv2.putText(frame,string, (100,100), cv2.FONT_HERSHEY_SIMPLEX, 3, (0,0,255), 3,
cv2.LINE_AA)
        cv2.imshow("Hand detect", frame)
        key=cv2.waitKey(1)
```

```
        if key == ord('q'):
                break
videoStream.release()
cv2.destroyAllWindows()
```

Anexa 3. Codul sursă pentru antrenarea modelului ce prezice poziția mâinii în cadru

```
#importăm modulele necesare
from PIL import Image
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import numpy as np
import os
import cv2
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Flatten
from keras.layers import Dense
from keras.layers import Dropout
from keras.callbacks import TensorBoard
from time import time

#definim caile către foldere cu imagini de antrenare/testare și către fișierele cu etichete
trainImagesPath = 'trainImages'
trainLabelsPath = 'handCoordsTrain.csv'
testImagesPath = 'testImages'
testLabelsPath = 'handCoordsTest.csv'

#etichetele sunt stocate în fișiere de tip .csv, le vom citi și le vom salva în două
variabile
trainLabelsFile ={}
with open(trainLabelsPath) as f:
    lines = f.read().splitlines()
for line in lines:
    elements=line.split(',')
    trainLabelsFile[elements[0]]=elements[1:]


testLabelsFile ={}
with open(testLabelsPath) as f:
    lines = f.read().splitlines()
for line in lines:
    elements=line.split(',')
    testLabelsFile[elements[0]]=elements[1:]

#salvăm numărul de imagini de antrenare și testare în două variabile
trainDataLength = len(os.listdir(trainImagesPath))
testDataLength = len(os.listdir(testImagesPath))
print("Număr imagini antrenare: "+str(trainDataLength))
print("Număr imagini testare: "+str(testDataLength))

#inițializăm matricea de intrare și matricea de etichete cu 0
trainData = np.zeros([trainDataLength,100,100,3])
trainLabels = np.zeros([trainDataLength,4])

for index,filename in enumerate(os.listdir(trainImagesPath)):
    picture = Image.open(os.path.join(trainImagesPath,filename))
    picture = picture.resize((100,100),Image.ANTIALIAS)
    picture = np.array(picture)
    trainData[index,:,:,:]=picture
    trainLabels[index,0]=float(trainLabelsFile[filename][0])/1280
    trainLabels[index,1]=float(trainLabelsFile[filename][1])/720
    trainLabels[index,2]=float(trainLabelsFile[filename][2])/1280
    trainLabels[index,3]=float(trainLabelsFile[filename][3])/720

#imaginile sunt reprezentate în RGB, pixelii au valori între 0 și 255; împărțim la 255
pentru a obține o valoare între 0 și 1
trainData = trainData/255
```

```
#definim modelul
model = Sequential()
model.add(Conv2D(filters=96,kernel_size=(3,3),input_shape=(100,100,3),activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(filters=256,kernel_size=(3,3),activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(filters=384,kernel_size=(3,3),activation='relu'))
model.add(Conv2D(filters=384,kernel_size=(3,3),activation='relu'))
model.add(Conv2D(filters=256,kernel_size=(3,3),activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(units=100,activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(units=100,activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(units=100,activation='relu'))
model.add(Dense(units=4,activation='sigmoid'))

#inițializăm un obiect de tip TensorBoard pentru a putea monitoriza procesul de antrenare
tensorboard = TensorBoard(log_dir="logs/{}".format(time()))

#configurăm modelul pentru antrenare
model.compile(optimizer='adam',loss='mean_squared_error', metrics=['accuracy'])

#pornim procesul de antrenare
model.fit(trainData,trainLabels,epochs=100, verbose=1, callbacks=[tensorboard])

#salvăm în matricea testData imaginile de test și în testLabels etichetele imaginilor
testData = np.zeros([testDataLength,100,100,3])
testLabels = np.zeros([testDataLength,4])
for index,filename in enumerate(os.listdir(testImagesPath)):
    picture = Image.open(os.path.join(testImagesPath,filename))
    picture = picture.resize((100,100),Image.ANTIALIAS)
    picture = np.array(picture)
    testData[index,:,:,:]=picture
    testLabels[index,0]=float(testLabelsFile[filename][0])/1280
    testLabels[index,1]=float(testLabelsFile[filename][1])/720
    testLabels[index,2]=float(testLabelsFile[filename][2])/1280
    testLabels[index,3]=float(testLabelsFile[filename][3])/720
testData = testData/255

#evaluăm modelul
model.evaluate(testData,testLabels)

#prezicem ieșirea rețelei
networkOutput=model.predict(testData)
print(model.evaluate(testData,testLabels))
print(model.metrics_names)

#desenăm pe ecran dreptunghiurile prezise pentru a analiza rezultatele obținute
for filename,x in zip(os.listdir(testImagesPath),networkOutput):
    picture = cv2.imread('testImages/%s' % filename, cv2.IMREAD_COLOR)
    cv2.rectangle(picture, (int(x[0]*1280), int(x[1]*720)), (int(x[2]*1280), int(x[3]*720)),
(0, 255, 0), 3)
    cv2.imshow("Imagine",picture)
    cv2.waitKey(0)

#arhitectura rețelei
model.summary()

#salvăm modelul
model.save('handTrackModel.h5')
```

Anexa 4. Codul sursă pentru urmărirea mâinii în timp real

```
#importăm modulele necesare
import cv2
import numpy as np
from PIL import Image
from keras import models

#încărcam modelele antrenate anterior
handDetectModel = models.load_model('handDetectModel.h5')
handTrackModel = models.load_model('handTrackModel.h5')
videoStream = cv2.VideoCapture(0)
videoStream.set(cv2.CAP_PROP_FPS,5)
videoStream.set(cv2.CAP_PROP_FRAME_WIDTH, 1280)
videoStream.set(cv2.CAP_PROP_FRAME_HEIGHT, 720)

while True:
        #citim un cadru de la captura live
        _, frame = videoStream.read()

        #convertim cadrul captat în spațiul RGB
        picture = Image.fromarray(frame, 'RGB')

        #redimensionăm cadrul la 100x100 deoarece modelul a fost antrenat pe imagini de
această dimensiune
        picture = picture.resize((100,100),Image.ANTIALIAS)
        currentImage =np.array(picture)
        currentImage=currentImage/255

        #modelul Keras folosește un tensor 4-dimensional așa că vom redimensiona de la
100x100x3 la 1x100x100x3
        currentImage = np.expand_dims(currentImage, axis=0)

        #calculăm ieșirea pentru modelele salvate anterior
        detectHand = handDetectModel.predict(currentImage)
        trackHand = handTrackModel.predict(currentImage)
        if detectHand > 0.5:
            if np.all(trackHand > 0):
                frame= cv2.rectangle(frame, (int(trackHand[0][0]*1280),
int(trackHand[0][1]*720)), (int(trackHand[0][2]*1280),int(trackHand[0][3]*720)),(0,255,0),3)
        cv2.imshow("Hand tracking", frame)
        key=cv2.waitKey(1)
        if key == ord('q'):
                break
videoStream.release()
cv2.destroyAllWindows()
```