



**APLICAȚIE WEB PENTRU RECUNOAȘTEREA ȘI
TRADUCEREA LIMBAJULUI SEMNELOR**

LUCRARE DE LICENȚĂ

Absolvent: **Radu-Ionuț SUCIU**

Coordonator **Conf. dr. ing. Ion-Augustin GIOSAN**
științific:

2024



FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

DECAN,

Prof. dr. ing. Mihaela DÎNSOREANU

DIRECTOR DEPARTAMENT,

Prof. dr. ing. Rodica POTOLEA

Absolvent: **Radu-Ionut SUCIU**

**APLICAȚIE WEB PENTRU RECUNOAȘTEREA ȘI TRADUCEREA
LIMBAJULUI SEMNELOR**

1. **Enunțul temei:** Implementarea unei aplicații web în scopul susținerii oamenilor cu dizabilități și a comunicării lor mai facile în contextul social actual
2. **Conținutul lucrării:** Capitolul 1 - Introducere – Contextul proiectului, Capitolul 2 - Obiectivul proiectului, Capitolul 3 - Studiu bibliografic, Capitolul 4 - Analiză și Fundamentare Teoretică, Capitolul 5 - Proiectare de Detaliu și Implementare, Capitolul 6 - Testare și Validare, Capitolul 7 - Manual de Instalare și Utilizare, Capitolul 8 – Concluzii
3. **Locul documentării:** Universitatea Tehnică din Cluj-Napoca, Departamentul Calculatoare
4. **Consultanți:** -
5. **Data emiterii temei:** 1 Noiembrie 2023
6. **Data predării:** 12 iulie 2024

Absolvent: _____

Coordonator științific: _____

**FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL CALCULATOARE****Declarație pe propria răspundere privind
autenticitatea lucrării de licență**

Subsemnatul **Suciuc Radu-Ionut**, legitimat cu carte de identitate seria **ZS nr.374239** CNP **5020616261960**, autorul lucrării **APLICAȚIE WEB PENTRU RECUNOAȘTEREA ȘI TRADUCEREA LIMBAJULUI SEMNELOR** elaborată în vederea susținerii examenului de finalizare a studiilor de licență la Facultatea de Automatică și Calculatoare, Specializarea **TEHNOLOGIA INFORMATIEI** din cadrul Universității Tehnice din Cluj-Napoca, sesiunea **IULIE 2024** a anului universitar **2023-2024**, declar pe propria răspundere că această lucrare este rezultatul propriei activități intelectuale, pe baza cercetărilor mele și pe baza informațiilor obținute din surse care au fost citate, în textul lucrării și în bibliografie.

Declar că această lucrare nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române și a convențiilor internaționale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în fața unei alte comisii de examen de licență.

În cazul constatării ulterioare a unor declarații false, voi suporta sancțiunile administrative, respectiv, *anularea examenului de licență*.

Data

Nume, Prenume

12.07.2024**Suciuc Radu-Ionut**

Semnătura

Cuprins

Capitolul 1 Introducere	1
1.1 Contextul temei	1
1.2 Domeniul lucrării	2
1.3 Motivație	3
1.4 Structura lucrării	3
Capitolul 2 Obiectivele proiectului	5
2.1 Specificațiile proiectului	5
2.2 Obiective generale	6
2.2.1 Cerințe funcționale	6
2.2.2 Cerințe non-funcționale	7
Capitolul 3 Studiu bibliografic	8
3.1 Soluții bazate pe HMM	8
3.2 Soluții bazate pe caracteristici geometrice	10
3.3 Soluții bazate pe accesorii	11
Capitolul 4 Analiză și fundamentare teoretică	13
4.1 Arhitectura Client-Server	13
4.2 HTTP	14
4.3 JSON	15
4.4 React	15
4.5 Flask	16
4.6 Google Bucket Storage	16
4.7 MediaPipe Hands	17
4.8 Heroku	20
4.9 GitHub Pages	20
4.10 OpenCV	21
4.11 TensorFlow	22
4.12 CNN	25
4.13 LSTM	29
4.14 Transformers	32
Capitolul 5 Proiectare de detaliu și implementare	33
5.1 Rețeaua neuronală pentru recunoașterea semnelor	33
5.2 Sistemul Client-Server	41
5.3 Cazuri de utilizare	43
5.4 Serverul	46
5.5 Aplicația Client	47
5.6 Deployment	53
Capitolul 6 Testare și validare	54
6.1 Interfața grafica	54
6.2 Serverul	54
6.3 Rețeaua neuronală	55

Capitolul 7 Manual de instalare și utilizare	59
7.1 Manual de instalare	59
7.2 Manual de utilizare	60
Capitolul 8 Concluzii	64
8.1 Concluzii si contributii personale	64
8.2 Analiza critica si dezvoltari ulterioare	64
Bibliografie	66

Capitolul 1. Introducere

1.1. Contextul temei

Pentru persoanele cu probleme de auz sau vorbire, abilitatea de a se exprima și de a interacționa social nu este un lucru atât de banal precum ni se pare nouă, luptându-se zilnic pentru a se integra folosindu-se de acest limbaj. Persoanele afectate de acest tip de handicap se lovesc zi de zi de această barieră de limbaj, încercând să comunice cât mai eficient posibil.

Pentru ei, limbajul semnelor reprezintă o unealtă esențială de comunicare. Dar, deși este o unealtă atât de necesară, la momentul actual există un număr foarte nesemnificativ de unelte, tehnologii sau ajutoare pentru a ajuta aceste persoane în traducerea a ceea ce vor să transmită prin limbajul semnelor în text sau vorbire și viceversa.

Din acest motiv, este necesar cât mai mult ajutor pentru acest tip de persoane. Urgența unei unelte de acest tip este foarte mare, această comunitate fiind exclusă social și profesional zi de zi. După părerea mea, în societatea modernă cu care ne confruntăm zilnic, e esențial să empatizăm cât mai mult cu persoanele din jurul nostru și să încercăm pe cât posibil să asigurăm acces egal la resurse și oportunități pentru orice om, indiferent de dizabilități, clasele sociale, religioase sau de etnie în care se pot încadra.

Orice tip de ajutor, precum și-a propus și această aplicație să fie, poate avea un impact imens asupra vieții unui număr semnificativ de persoane, oferind șansa de a participa activ și egal la viața socială și profesională, cât și o societate mai incluzivă și accesibilă pentru toți participanții ei.

Acest proiect de licență are ca scop dezvoltarea unei aplicații web inovative care să recunoască și să traducă limbajul semnelor în timp real prin utilizarea tehnologiilor avansate de inteligență artificială, învățare automată și procesare de imagini. Aplicația va asigura persoanelor cu dizabilități o comunicare mai ușoară cu persoanele care nu cunosc acest tip de limbaj, eliminând acest baraj între persoane.

Scopul final al acestui proiect este facilitarea accesului la servicii precum educația sau oportunitățile de angajare, promovând astfel includerea în societate.

1.2. Domeniul lucrării

Lucrarea se situează la intersecția dintre aplicațiile web, procesarea imaginilor și învățare automată, fiind structurată în trei componente esențiale: interfața utilizator, serverul și aplicația pentru crearea modelului, colectarea setului de date de antrenare și testare, și antrenarea modelului. Aplicația este disponibilă tuturor prin intermediul platformelor de *hosting* oferite de Google Cloud, GitHub și Heroku.

Scopul principal al aplicației este de a ușura comunicarea pentru persoanele cu dizabilități de vorbire sau auz, oferind funcționalități avansate de recunoaștere a semnelor și translatarea lor în text și în vorbire. Aceasta permite utilizatorilor să se exprime eficient în diverse contexte, cum ar fi interviurile, conferințele sau prezentările.

Interfața utilizator a fost implementată în JavaScript, cu ajutorul *framework-ului* React, iar stilizarea a fost realizată în CSS. Pentru a scădea latența și a reduce numărul apelurilor către server, operațiile esențiale, cum ar fi recunoașterea semnelor, sunt realizate direct în partea de interfață. Fiecare cadru video este procesat utilizând MediaPipe, un *framework* puternic care recunoaște mâna și cele 21 de repere ale mâinii și degetelor în fiecare cadru. Coordonatele x, y, z ale acestor repere sunt colectate într-un vector de 30 de cadre, care este apoi trimis către un model stratificat (*layered model*) creat de la zero și antrenat cu date proprii. Modelul este stocat în Google Cloud Bucket Storage și încărcat în interfață. Procesarea rezultatului implică logică suplimentară în interfață pentru a asigura acuratețea și fluiditatea conversiei semnelor. Dezvoltarea continuă și deployment-ul interfeței este realizat prin Github Pages.

Serverul aplicației este implementat în Python și utilizează *framework-ul* Flask. Aceasta are rolul de a oferi sugestii și corecții pentru textul deja scris, utilizând un model pre-antrenat bazat pe un set de date de greșeli-corecturi. Modelul este accesat prin Hugging Face Inference API pentru scăderea costului stocării. Comunicarea dintre interfață și server se realizează prin apeluri HTTP de tip POST, folosind JSON ca format pentru transmiterea textului. Partea de deployment și dezvoltare continuă a serverului este asigurată prin Heroku, oferind scalabilitate și flexibilitate ridicată.

Aplicația pentru colectarea datelor și antrenarea modelului este scrisă în Python. Aceasta colectează date pentru fiecare semn, reținând coordonatele reperelor mâinii pentru 30 de cadre. Fiecare cadru este stocat într-un fișier, formând un tip de video scurt de 30 de cadre. Modelul este construit folosind o arhitectură de rețea neuronală convoluțională (*Convolutional Neural Network*), care include un strat de bază convoluțional, urmat de alternări de straturi LSTM, Dropout și Dense. După antrenarea modelului cu datele colectate, acesta este transformat într-un model TensorFlow JavaScrip pentru a putea fi utilizat în interfața utilizator. Modelul este stocat și accesibil continuu prin Google Cloud Bucket Storage, asigurând disponibilitatea și performanța necesară pentru aplicație.

Astfel, domeniul lucrării înglobează atât tehnologiile avansate de procesare de imagini și învățare automată, cât și platformele de *hosting* moderne pentru a crea o soluție eficientă și performantă, dezvoltată continuu.

1.3. Motivație

Proiectul are ca scop să îmbunătățească comunicarea pentru persoanele cu dizabilități, din cauză că această comunitate se confruntă zi de zi cu provocări mari în interacțiunea cu ceilalți prin lipsa unui ajutor eficient de comunicare care poate duce la izolare socială și limitări profesionale. Din acest motiv, este necesar să dezvoltăm soluții tehnologice care să le ofere sprijin. Prin această aplicație, se dorește să le fie oferit persoanelor cu dizabilități un instrument inovator pentru exprimare cât mai ușoară în diverse contexte, precum interviuri, conferințe sau prezentări.

Un alt motiv esențial este promovarea aplicațiilor web moderne, care folosesc tehnologii avansate precum Python și React. Acest proiect demonstrează importanța învățării automate pentru îmbunătățirea preciziei și performanței aplicației, un avantaj fiind recunoașterea rapidă și precisă a semnelor.

Prin utilizarea serviciilor de *hosting* și *cloud* pentru deployment se realizează accesibilitate continuă și scalabilitate, fiind asigurat utilizatorilor funcționalități fără întreruperi. Prin acest aspect, se înțelege evidențierea importanței infrastructurii *cloud* în dezvoltarea aplicațiilor web.

1.4. Structura lucrării

Lucrarea este împărțită în opt capituloare, fiecare abordând aspecte importante ale proiectului și ale implementării acestuia.

- **Introducere**

În acest capitol introductiv presupune o prezentare generală a proiectului și include contextul și domeniul în care acesta se încadrează. Tot aici se descrie structura lucrării și motivația pentru proiectul ales.

- **Obiectivele proiectului**

Acest capitol detaliază specificațiile proiectului, care includ cerințele funcționale și nefuncționale. Sunt prezentate și obiectivele generale din spatele proiectului.

- **Studiu bibliografic**

În Capitolul 3, se analizează situația actuală din domeniul în care se încadrează tema proiectului. Se prezintă lucrări și articole în care sunt prezentate alte soluții pentru aceeași problemă.

- **Analiză și Fundamentare Teoretică**

Acest capitol descrie termenii teoretici care stau la baza soluției propuse. Sunt detaliate logica soluției alese, teoria din spatele acesteia și tehnologiile folosite. Se analizează avantajele și dezavantajele diferitelor tehnologii folosite.

- **Proiectare de Detaliu și Implementare**

Aici este descrisă în detaliu arhitectura sistemului proiectat și implementarea acestuia. Capitolul oferă informații despre toate componentele proiectului, clasele importante și metodele esențiale, explicând atât funcționarea, cât și implementarea fiecărei componente.

- **Testare și Validare**

Acest capitol detaliază scenariile de testare aplicate atât pentru interfața utilizator, cât și pentru partea de server. Sunt prezentate rezultatele obținute în urma acestor teste și se discută validarea aplicației.

- **Manual de Instalare și Utilizare**

Aici sunt descriși pașii necesari pentru instalarea și utilizarea aplicației. Sunt prezentate informații despre resursele *software* și *hardware* necesare, precum și ghiduri pentru folosirea corectă a sistemului.

- **Concluzii**

În capitolul final sunt prezentate concluziile proiectului și posibilele dezvoltări ulterioare. Sunt remarcate și rezultatele cât și contribuțiile personale principale.

Capitolul 2. Obiectivele proiectului

2.1. Specificațiile proiectului

Scopul în acest proiect este studierea, implementarea și expunerea publicului a unei aplicații care oferă un sprijin în comunicarea persoanelor cu dizabilități, susținând un mediu mai inclusiv și orientat spre susținerea tuturor persoanelor.

Proiectul are ca principale funcționalități extragerea imaginii din camera web, reprezentând datele de intrare, cât și redarea ca date de ieșire a semnului recunoscut din imaginea de intrare. Etapele de procesare sunt numeroase, iar la final se dorește oferirea unor opțiuni de redare audio sau de corectare a textului, pe lângă funcționalitatea principală. Lista semnelor recunoscute este prezentată în figura 2.1 :

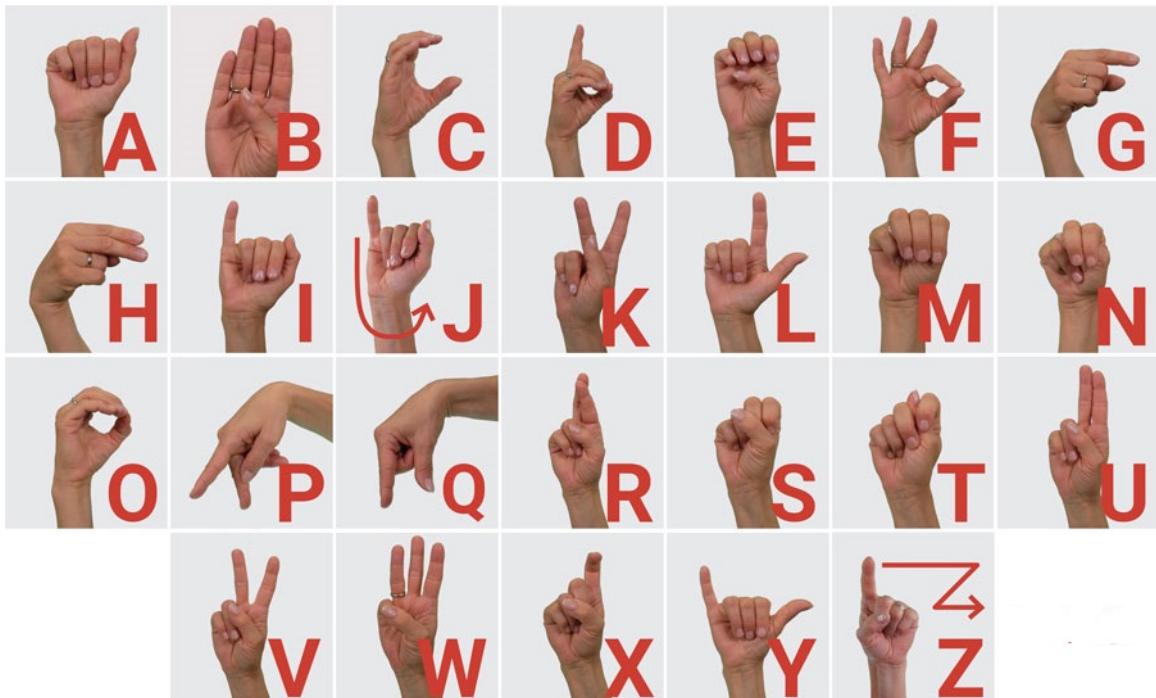


Figura 2.1: Alfabetul semnelor

La aceste semne se adaugă 3 semne utilizare: spațiu, ștergere și un semn utilitar pentru trecerea de la un caracter la următorul.

2.2. Obiective generale

Aplicația trebuie să fie accesibilă pe orice tip de browser, oferind disponibilitate continuă, astfel s-a folosit platformele de *hosting* pentru deployment-ul atât al interfeței utilizator cât și al serverului. Acest aspect acoperă nevoia unei configurații sau a unei instalări a aplicației.

Alt obiectiv principal este de a oferi o interfață cât mai sugestivă și ușor de utilizat, care ajută utilizatorul să navigheze între paginile aplicației cât și să folosească funcționalitățile oferite de aceasta.

Funcționalitățile propuse ale aplicației sunt de a oferi un sistem cât mai precis și mai coerent al recunoașterii semnelor alfabetului persoanelor cu dizabilități, clasificarea cât mai rapidă a rezultatului dar și ajutorul suplimentar oferit prin funcționalitățile de redare audio cât și a sugestiei de corectare a textului.

Un alt scop este implementarea aplicației atât în partea de frontend cât și în partea de backend, dar și utilizarea unor modele de învățare automată și a serviciilor de *hosting* și *cloud*, astfel aducând o extindere a capabilităților aplicației dar și includerea majorității domeniilor populare și folosite des în perioada actuală.

2.2.1. Cerințe funcționale

În această secțiune vor fi prezentate cerințele funcționale, care stabilesc funcționalitatea aplicației. Funcționalitățile descrise aici sunt cele care vor fi utilizate intern de către sistem, fiind invizibile pentru utilizator.

1. Colectarea imaginilor de la camera web
2. Comunicarea continuă între interfață și server
3. Recunoașterea corectă a semnelor
4. Procesarea și afișarea în timp real a rezultatelor
5. Extragerea punctelor de reper pentru fiecare cadru
6. Implementarea sistemului de redare audio
7. Colectarea datelor pentru antrenare
8. Crearea modelului pentru recunoaștere
9. Antrenarea modelului
10. Integrarea modelului de detectie mâini
11. Afișarea regulilor de folosire a sistemului
12. Navigarea fluentă între pagini
13. Sugestia cât mai potrivită pentru textul actual

2.2.2. Cerințe non-funcționale

Aceste cerințe se concentrează mai mult pe calitatea sistemului, decât pe funcționalitatea sa. Aici sunt incluse specificațiile referitoare la bibliotecile și limbajele de programare utilizate, performanța sistemului, ușurința în utilizare și alte caracteristici similare.

1. Performanță - fiind un sistem în timp real, este necesară o recunoaștere și afișare cât mai rapidă a rezultatului
2. Disponibilitate - proprietatea sistemului de a fi disponibil oricând și oriunde datorită hosting-ului aplicației
3. Flexibilitate - abilitatea de a fi disponibil de pe orice browser sau sistem de operare
4. Utilizarea limbajului JavaScript și a framework-ului React pentru frontend
5. Utilizarea limbajului Python și a framework-ului Flask pentru backend
6. Utilizarea Heroku pentru hosting-ul serverului
7. Utilizarea Google Bucket Storage pentru stocarea modelului de recunoaștere
8. Utilizarea Github Pages pentru deployment-ul interfeței
9. Utilizarea Hugging Face Inference API pentru accesarea modelului de spell-check
10. Utilizarea Visual Studio Code ca mediu de dezvoltare frontend
11. Utilizarea PyCharm ca mediu de dezvoltare backend
12. Utilizarea Jupyter Notebook pentru dezvoltare model

Capitolul 3. Studiu bibliografic

Domeniul limbajului semnelor este un domeniu cu soluții vaste din punct de vedere al sistemelor informaticice, încadrându-se în sfera de procesare a imaginilor, axată pe detecția oamenilor și a mișcărilor dinamice pe care aceștia le produc. În acest capitol, sunt prezentate unele soluții din domeniu, de la soluții folosind mănuși sau alte accesorii, la rețele neuronale care sunt antrenate și învățate să recunoască într-un mod general caracteristicile oamenilor. La aceste soluții s-a ales adăugarea și a soluției implementate în acest proiect, existând o încredere crescută că a fost adus un plus în acest domeniu prin logica de funcționare implementată.

3.1. Soluții bazate pe HMM

O soluție din domeniul actual al detecției mâinii sau recunoașterii gesturilor o reprezintă modelele ascunse Markov, fiind niște modele statice în care sistemul este descris de o serie de stări ascunse care nu sunt direct observabile și o altă serie opusă, de aspecte observabile. Pentru un sistem de recunoaștere a gesturilor, stările ascunse pot fi reprezentate de fazele gestului. Pentru aspectele observabile, în cazul recunoașterii gesturilor, reprezentarea lor poate fi coordonatele mâinii. Diferența dintre cele două componente este că aspectele observabile pot fi măsurate sau observate clar.

Una dintre soluțiile de acest tip este dezvoltată de către Chen, Fu și Huang în lucrarea [1], unde sistemul de recunoaștere bazat pe HMM primește ca input variația formei mâinii și informațiile de mișcare în urma urmăririi mișcării mâinii în timp real. Este împărțit în 3 componente: urmărire în timp real, extragerea caracteristicilor și recunoașterea gesturilor bazată pe HMM.

În această lucrare [1], autorii aplică transformata Fourier pentru a caracteriza informațiile spațiale și metoda fluxului optic pentru analiza mișcării. Combinând transformata Fourier și detaliile despre mișcare, se antrenează sistemul. Pe partea de antrenare, fiecare gest este definit în termeni de parametri. Gestul recunoscut este evaluat în funcție de diferite HMM-uri. Cine are cel mai mare scor, câștigă.

În figura 3.1 din lucrarea [1] este prezentată diagrama de flux a sistemului, împărțită în 3 pași: extragerea caracteristicilor, antrenarea și recunoașterea. Pentru fiecare vector de caracteristici, este atribuit un simbol. Secvența de imagini de intrare este reprezentată printr-o secvență de simboluri. În faza de antrenare, un HMM este construit pentru fiecare gest. În faza de recunoaștere, un gest de intrare dat este testat de fiecare HMM cu parametri de model diferenți. Rezultatul HMM-ului cu funcția de probabilitate maximă este identificat pentru a recunoaște gestul.

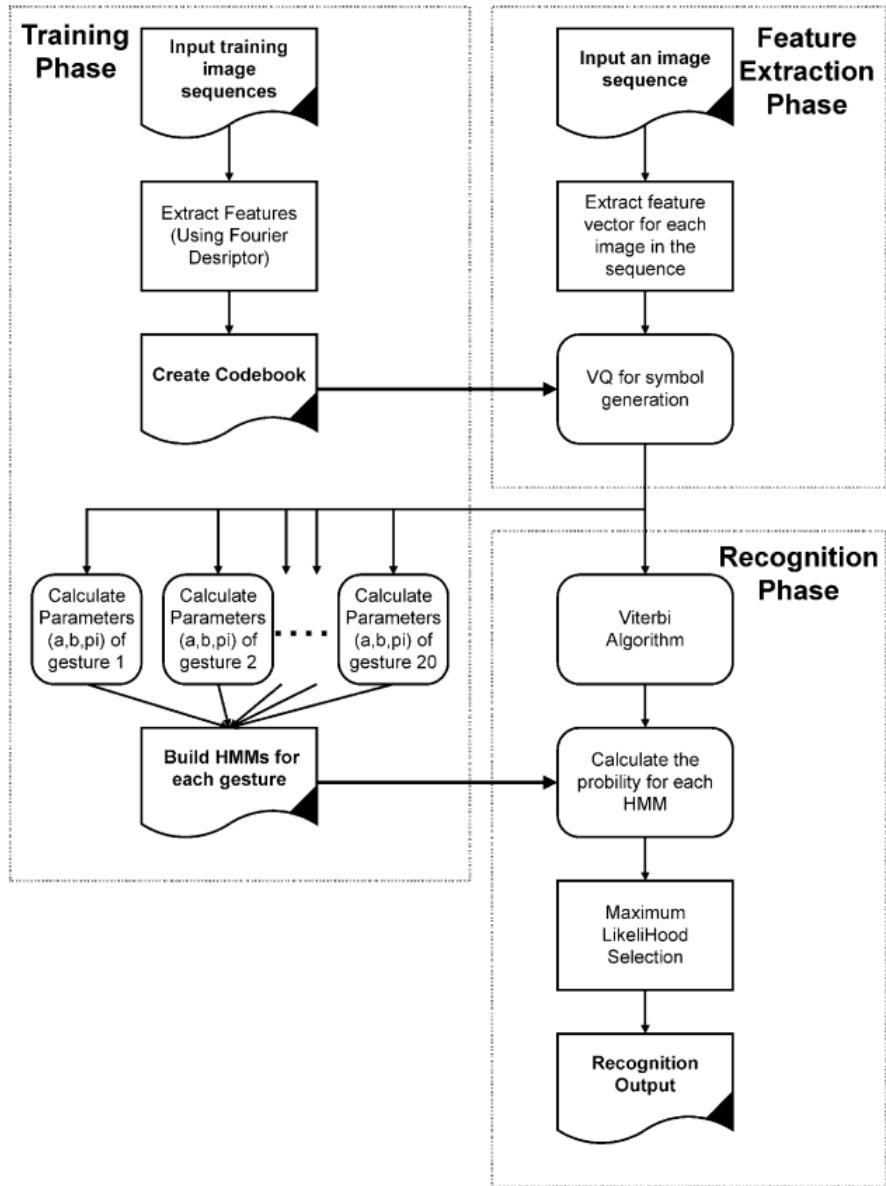


Figura 3.1: Diagrama flux sistem HMM [1]

În figura 3.2 prezentată de către Chen și colegii săi [1] este exemplificată și diagrama sistemului de urmărire a gesturilor mâinii, focalizată pe regiunea de interes în loc de întreaga imagine. Cele trei blocuri funcționale indică detectia mișcării, detectia marginilor și detectia culorii pielii, care pot opera în paralel. Este folosită o tehnică simplă de scădere a fundalului pentru a obține forma gestului mâinii. Pentru a găsi o regiune a mâinii mai precisă, se folosesc informațiile despre regiunea din prim-plan. Poziția mâinii a fost găsită utilizând informațiile despre mișcare, culoarea pielii și marginile. Sistemul general pentru urmărirea regiunii mâinii are două etape: prima etapă este concentrată pe informațiile despre mișcare, în timp ce a doua etapă se concentrează pe informațiile despre regiunea din prim-plan.

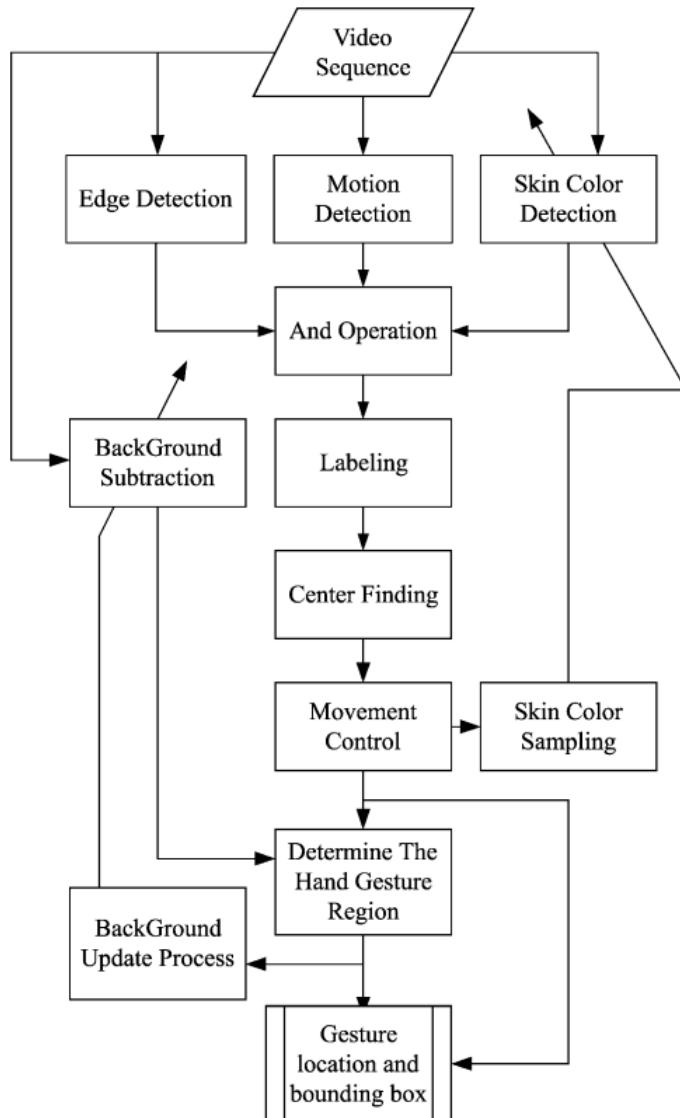


Figura 3.2: Diagrama sistemului de urmărire a gesturilor mâinii [1]

3.2. Solutii bazate pe caracteristici geometrice

Caracteristicile geometrice sunt niște proprietăți ale formelor obiectelor, extrase din imagine cu scopul de a analiza aspectele obiectelor, cum este și în cazul nostru, al mâinii. Printre acestea se numără puncte cheie precum colțuri, vârfuri sau margini, lungimi și distanțe, cum ar fi lungimea degetelor, unghiurile dintre ele sau chiar unghиurile dintre degete și palmă, arii și perimetre sau chiar rotații. Acestea pot fi folosite ca unealtă decisivă în clasificarea unor imagini în semne precise ale limbajului semnelor.

Aceste metode sunt introduse și în lucrarea [2], scrisă de Riad Elminir și Shohieb, în care sistemul conține 4 componente majore: localizarea mâinii, descrierea regiunii mâinii, extragerea caracteristicilor geometrice și, în final, clasificarea pe baza regulilor. Localizarea mâinii se face pe baza culorii mâinii, a histogramelor generate de aceasta și a regulii Bayes. Apoi se calculează regiunea de interes pentru o complexitate computațională mai scăzută.

Extragerea caracteristicilor în lucrarea [2] este cel mai important pas și se concentrează pe caracteristici precum lungimea marginii, aria, centrul de greutate, momentele de ordinul 2, compactitatea, orientarea și chiar excentricitatea. Se mai adaugă coordinatele celui mai jos și celui mai sus poziționat pixel. Ca ultim pas, există un set bine definit de reguli pentru a clasifica semnul pe baza caracteristicilor extrase. O diagramă care prezintă pașii soluției este afișată în figura 3.3.

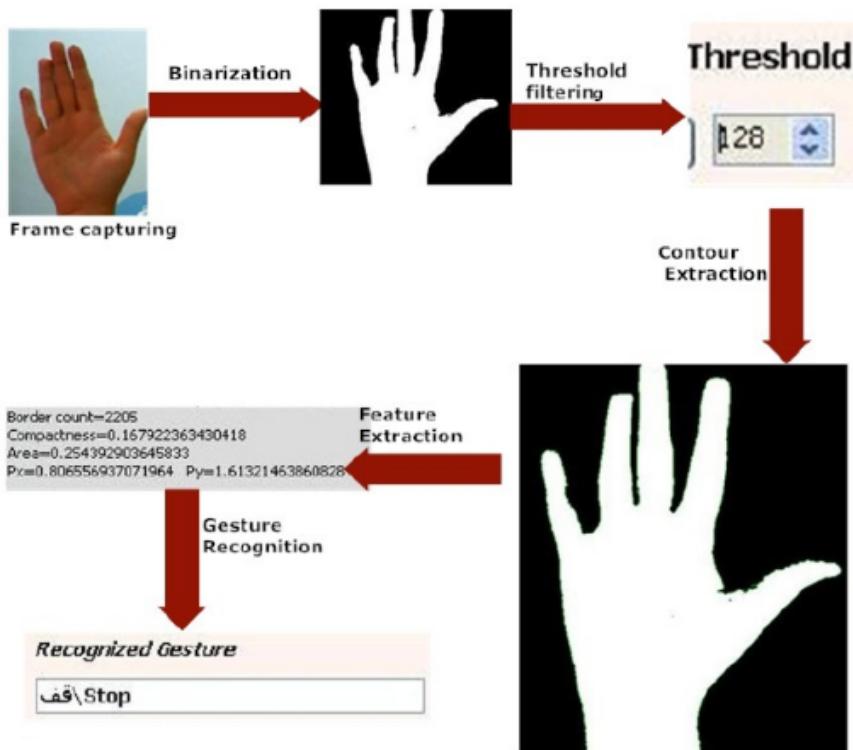


Figura 3.3: Diagrama sistemului bazat pe caracteristici geometrice [2]

3.3. Soluții bazate pe accesoriu

O variantă mai puțin facilă pentru persoanele cu acest tip de dizabilități o reprezintă folosirea unei mănuși speciale, care trimit date către un sistem ce le clasifică într-un semn specific. Această variantă are dezavantajul necesității unei unelte *hardware*, de care nu ar dispune orice persoană în nevoie, pe lângă disconfortul creat de folosirea ei. Totuși, acest tip de soluție a fost documentat foarte bine în acest domeniu, sistemele devenind din ce în ce mai precise.

O soluție de acest gen este cea expusă în lucrarea [3] de către Sadek, Mikhael și Mansour, în care sistemul se bazează pe analiza statistică. Fluxul aplicației este în mare parte similar cu al altor sisteme, doar că intervine o mănușă inteligentă pentru captarea datelor. Mănușa este echipată cu senzori de flexiune pentru degete și senzori inertiali pentru accelerare și orientare. Datele sunt apoi normalizate și filtrate pentru consistență și eliminarea zgromotului. O reprezentare a unei mănuși de acest tip și a senzorilor acesteia este prezentată în figura 3.4.

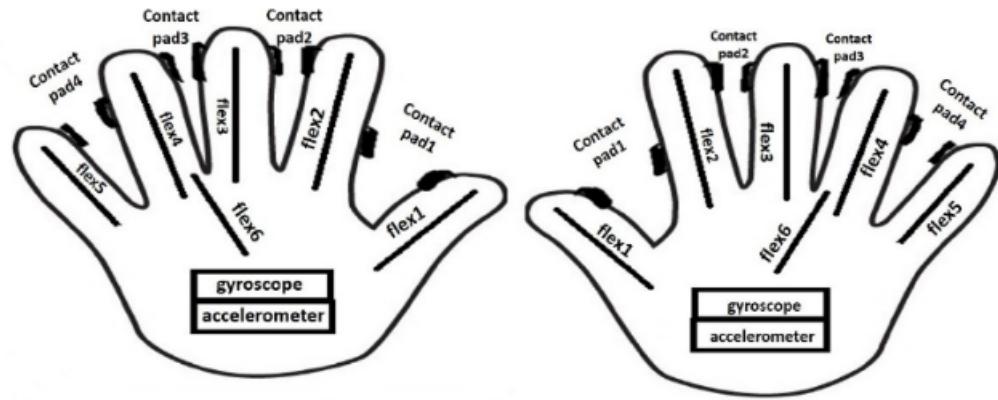


Figura 3.4: Reprezentare mănușă [3]

Punctul principal al sistemului expus de Sadek și colegii săi în lucrarea [3] este folosirea datelor pentru a extrage caracteristici geometrice și de mișcare, precum accelerarea și orientarea, pentru ca apoi acestea să fie clasificate în funcție de o analiză statistică avansată, care identifică tiparele și caracteristicile specifice ale gesturilor. Tot circuitul de comunicare folosit de mănușă care reprezintă baza acestei lucrări este expus în figura 3.5.

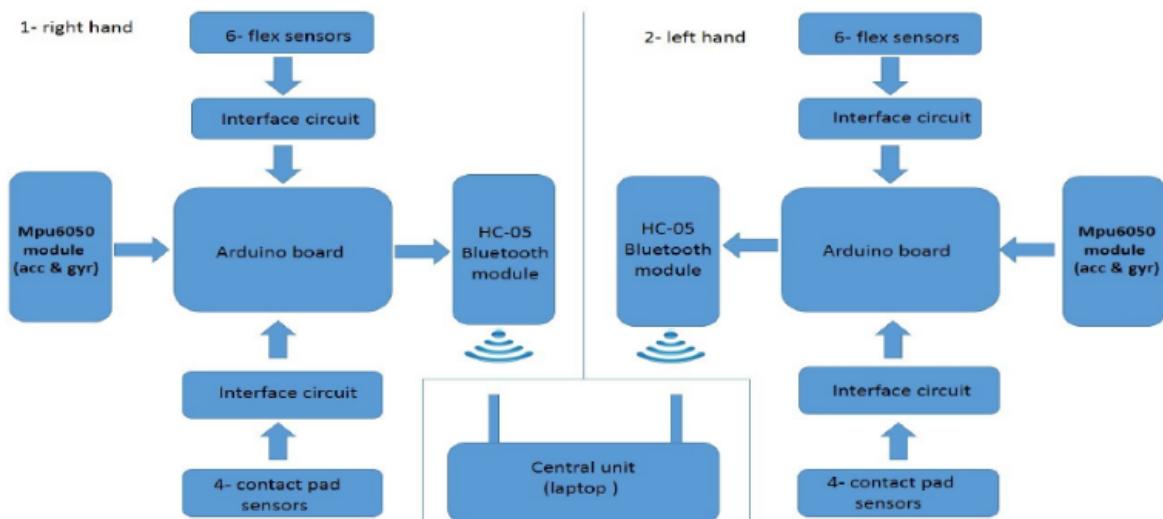


Figura 3.5: Reprezentare comunicare mănuși [3]

Capitolul 4. Analiză și fundamentare teoretică

4.1. Arhitectura Client-Server

Ca și arhitectură generală, s-a ales să folosirea arhitecturii Client-Server, fiind una dintre cele mai fiabile soluții pentru tema actuală.

Pe baza lucrării [4] s-a dedus că arhitectura client-server reprezintă un model esențial în designul și implementarea sistemelor informatici distribuite. Aceasta implică separarea funcționalităților între două entități principale: clientul, care solicită servicii, și serverul, care furnizează aceste servicii. Modelul client-server este prezent într-un număr mare aplicații moderne, de la navigarea pe web la aplicațiile mobile și sistemele de management al bazelor de date. O vedere exterioară este prezentată în figura 4.1.

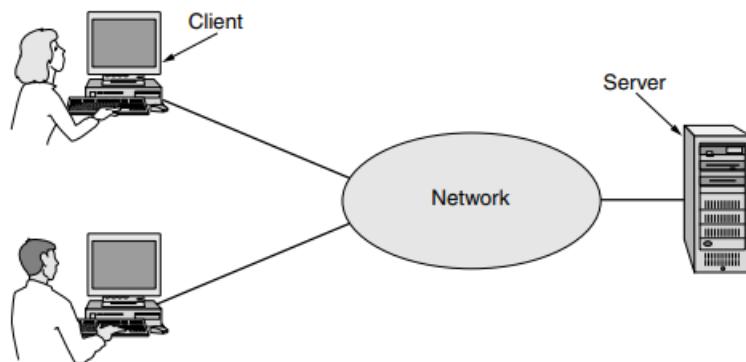


Figura 4.1: Structura arhitecturii Client-Server [5]

- **Clientul**

Este o aplicație sau un dispozitiv care trimit cereri către un server pentru a accesa resurse sau servicii specifice.

- **Serverul**

Este un sistem central care procesează cererile primite de la clienti, oferind resursele sau serviciile cerute. Serverele sunt adesea puternice din punct de vedere al capacitatei de calcul și stocare și sunt capabile să gestioneze un număr mare de cereri simultane.

- **Avantaje arhitectura Client-Server extrase din [4]**

1. **Scalabilitate** - Serverele pot fi scalate pentru a gestiona eficient un număr mare de cereri. Această scalabilitate apare adăugând resurse suplimentare (scalabilitate verticală) sau prin împărțirea cererilor pe mai multe servere (scalabilitate orizontală).
2. **Securitate** - Fiind centralizate, resursele și serviciile permit implementarea măsurilor de securitate într-un mod mai eficient, controlul accesului și protecția datelor fiind mult mai ușoare.

3. **Fiabilitate și Disponibilitate** - Arhitecturile client-server sunt populare din cauza redundanței și toleranței la eșec, asigurând continuitatea serviciilor chiar și în caz de defecțiuni hardware sau software.

- **Flux de comunicare în arhitectura Client-Server conform lucrării [5] și prezentat în figura 4.2**

1. Clientul trimite o cerere către server prin intermediul unei rețele (de exemplu, Internetul). Cererea poate include date de intrare, cum ar fi informații de autentificare, cereri de date specifice sau instrucțiuni pentru execuția unor funcții.
2. Serverul primește cererea și o procesează conform logicii implementate. Acest proces poate implica validarea datelor, accesarea și manipularea datelor dintr-o bază de date, efectuarea de calcule sau folosirea altor servicii.
3. După procesarea cererii, serverul generează un răspuns care este trimis înapoi clientului. Răspunsul poate conține datele solicitate, mesaje de confirmare sau erori în cazul în care cererea nu a putut fi procesată corect.
4. Clientul primește răspunsul și îl utilizează pentru a actualiza interfața utilizatorului sau pentru a continua alte procese.

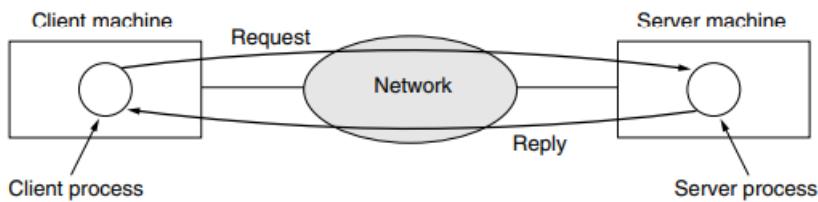


Figura 4.2: Flux comunicare Client-Server [5]

4.2. HTTP

Pentru comunicarea între client și server este folosită cea mai simplă și ușuală formă de comunicare: HTTP.

În articolul [6] *Hypertext Transfer Protocol* (HTTP) definește un set de metode pentru a solicita o operărie asupra unor resurse sau cu resursele oferite. Acesta funcționează ca un protocol de solicitare-răspuns între un client și un server. Comunicarea prin HTTP se face astfel: Un client (*browser* de exemplu) trimite o solicitare HTTP către server, iar serverul returnează un răspuns clientului pentru acea solicitare. Cererea conține informații necesare procesării acesteia, iar răspunsul conține informații despre starea solicitării și poate conține și conținutul solicitat.

Apelurile HTTP pot fi de multe tipuri, dar tipul folosit în această aplicație va fi doar metoda POST. Metoda POST trimite o entitate la resursa specificată, adesea cauzând o schimbare de stare sau efecte secundare pe server.

4.3. JSON

Pentru a transmite date printr-o cerere HTTP va fi folosit JSON. Conform [7], JSON (*JavaScript Object Notation*) este un format de schimb de date ușor și lizibil pentru oameni, utilizat pe scară largă pentru transmiterea datelor între un server și o aplicație web. JSON este un subset al sintaxei JavaScript, dar este independent de limbaj, fiind compatibil cu multe limbaje de programare, inclusiv Python, Java, C++ și altele.

- **Avantajele JSON prezentate în [7]**

1. Ușor de parsat și generat - Majoritatea limbajelor de programare moderne au biblioteci încorporate sau disponibile care facilitează parsarea și generarea JSON-ului.
2. Transport ușor - JSON este utilizat frecvent în aplicațiile web pentru a transporta date între client și server datorită simplității și compactității sale.

4.4. React

Pe lângă arhitectura principală și comunicarea între componente, un pas important îl reprezintă implementarea acestor componente. Pentru partea de client s-a ales JavaScript, iar ca *framework* de implementare s-a ales folosirea React.

Prezentată în lucrarea [8], React este o bibliotecă JavaScript *open-source* pentru construirea de interfețe de utilizator. Este utilizată pentru dezvoltarea de aplicații web interactive și scalabile, punând accent pe componentizarea și reutilizarea codului.

Principalele caracteristici ale React ar fi că promovează o arhitectură bazată pe componente, unde interfața grafică este împărțită în componente mici, independente și reutilizabile. Fiecare componentă își gestionează propriul status și logică, ceea ce face codul mai modular și mai ușor de întreținut.

Un alt beneficiu major al React, prezentat tot în [8], ar fi că utilizează *Virtual DOM*, o reprezentare virtuală a DOM-ului real. Când starea unei componente se schimbă, React actualizează Virtual DOM-ul și compară diferențele cu DOM-ul real, aplicând doar modificările necesare. Acest proces optimizează actualizările interfeței grafice și îmbunătățește performanța aplicațiilor. Pe lângă asta, folosește un flux de date unidirectional, ceea ce înseamnă că datele curg într-o singură direcție, de la componentele părinte la cele copii.

Mai cuprinde și JSX, care este o extensie de sintaxă pentru JavaScript și care permite scrierea de structuri HTML în cadrul codului JavaScript.

Avantajele devin destul de evidente, reprezentând motivul alegerii acestui framework:

1. Performanță
2. Reutilizabilitate
3. Modularitate
4. Suport comunitar și documentație - React are o comunitate largă de dezvoltatori și o documentație extensivă, oferind suport și resurse pentru a rezolva problemele și a învăța cele mai bune practici.

4.5. Flask

Pentru partea de server, s-a ales același limbaj ca și pentru crearea modelului, recoltarea datelor și antrenarea modelului, adică Python. Pentru a putea extinde acest limbaj pentru implementarea unei aplicații web s-a folosit *framework*-ul Flask.

Flask este prezentat în cartea [9] ca un *microframework web* pentru Python, cunoscut pentru simplitatea și flexibilitatea sa. Aceasta permite crearea de aplicații web rapide, fără a impune structuri rigide sau dependințe suplimentare.

Caracteristica principală pentru Flask care îl face să fie considerat un *microframework* este că nu include un ORM (*Object-Relational Mapping*) sau alte componente predefinite pe care multe alte framework-uri le au implicit. Aceasta îi conferă flexibilitate, permitând dezvoltatorilor să adauge doar componentele de care au nevoie.

Pe lângă asta, în lucrarea [9] se prezintă utilizarea unui sistem simplu de rutare care permite asocierea URL-urilor cu funcțiile Python corespunzătoare. Acest lucru face crearea și gestionarea rutelor foarte intuitivă. O altă caracteristică ar fi că Flask utilizează Jinja2, un motor de şablon rapid și expresiv, care permite utilizarea de variabile, condiționale și bucle în fișierele HTML, făcând generarea dinamică a paginilor web mai simplă și eficientă.

Avantajele prezentate în [9] și pentru care am ales Flask sunt:

1. Simplitate și Flexibilitate
2. Ușor de Învățat și Utilizat
3. Extensibilitate - Flask permite integrarea ușoară a extensiilor pentru a adăuga funcționalități suplimentare, cum ar fi autentificarea sau interacțiunea cu baze de date, fără a încărca framework-ul de bază.
4. Suport Comunitar și Documentație - Flask beneficiază de o comunitate mare și activă, care contribuie la dezvoltarea extensiilor, la rezolvarea problemelor și la oferirea de suport prin diverse forumuri și platforme.

4.6. Google Bucket Storage

Pentru a avea acces rapid la modelul de predicție și a nu stoca un model atât de mare pe o rețea de *hosting*, cât și din motive de securitate, s-a ales folosirea Google Cloud Storage pentru a stoca modelul.

Pe baza lucrării [10], Google Cloud Storage este un serviciu de stocare de obiecte oferit de Google Cloud Platform, destinat stocării și accesării datelor la scară globală. *Bucket-urile* sunt unitățile fundamentale de stocare în Google Cloud Storage, fiind containere în care utilizatorii își păstrează obiectele (fișierele).

S-a ales acest serviciu din cauză că Google Cloud Storage este proiectat pentru a gestiona volume mari de date, oferind stocare scalabilă la nivel global. Datele sunt replicate automat în mai multe locații pentru a asigura durabilitatea și disponibilitatea acestora.

Fiind o aplicație web cu scopul accesării din orice loc în lume, Google Bucket Storage oferă performanță ridicată care este asigurată prin utilizarea unei arhitecturi distribuite și optimizate, permitând accesul rapid la date indiferent de locația geografică.

Un alt plus este criptarea datelor și controlul granular al accesului, asigurând protecția datelor sensibile împotriva accesului neautorizat.

4.7. MediaPipe Hands

În limbajul semnelor, unele dintre acestea nu sunt doar o poziție anume, ci o întreagă mișcare. Din această cauză va fi folosit un scurt videoclip format dintr-un număr de cadre pentru a prezice și recunoaște semnele. Acest lucru îngreunează metoda de detectare a mâinii, variantele clasice de detectare cum ar fi metoda de extragere a conturului nu sunt cele mai optime soluții pentru problema detectării mâinii și a degetelor, fiind nevoie de detectare în timp real. Astfel s-a ales folosirea MediaPipe.

Prezentă în [11], MediaPipe Hands este o soluție avansată dezvoltată de Google Research pentru detectarea și urmărirea mâinilor în timp real, folosind tehnologii de învățare automată. Astfel oferă o identificare precisă și rapidă a poziției și mișcării mâinilor, fiind capabilă să ruleze eficient pe o varietate de dispozitive, de la telefoane mobile la computere personale, cât și pe tehnologii diverse.

Caracteristici principale

- Urmărire în Timp Real** - MediaPipe Hands oferă o urmărire a mișcărilor mâinilor în timp real, esențială pentru aplicații interactive. Permite detectarea instantă și precisă a mâinilor și a gesturilor acestora.
- Precizie Ridicată** - Utilizează modele de învățare automată antrenate pe seturi extinse de date, astfel asigurând o precizie ridicată în detectarea și urmărirea punctelor cheie ale mâinilor, inclusiv în condiții variate de lumină și fundaluri complexe.
- Procesare pe Dispozitiv** - Optimizat pentru a rula direct pe dispozitivele utilizatorilor, MediaPipe Hands minimizează latența și elimină necesitatea de a trimite date către servere pentru procesare, sporind astfel confidențialitatea și eficiența.

Arhitectura

Soluția de urmărire a mâinilor folosește un flux de lucru de învățare automată (ML) format din două modele prezentate în [11] care lucrează împreună:

- Un detector de palme care operează pe întreaga imagine de intrare și localizează palmele printr-o casetă de delimitare orientată a mâinii.
- Un model de repere pentru mâini care operează pe caseta de delimitare a mâinii decupate, furnizată de detectorul de palme, și returnează repere 2.5D de înaltă fidelitate.

Pentru a detecta locațiile inițiale ale mâinilor, se folosește un detector cu un singur cadru.

Autorii prezintă în [11] că primul pas implică utilizarea unui model de detecție bazat pe rețele neuronale convoluționale (CNN) pentru a identifica și localiza mâinile în imagine, deoarece estimarea casetelor de delimitare ale obiectelor rigide, cum ar fi palmele și pumnii, este semnificativ mai simplă decât detectarea mâinilor cu degete articulate. Deoarece palmele sunt obiecte mai mici, algoritmul de suprimare a maximelor non-locale funcționează bine chiar și în cazurile de auto-ocluzie a mâinilor, cum ar fi strângerile de mâină.

În al doilea pas exemplificat în [11], se folosește un extractor de caracteristici encoder-decoder similar cu FPN pentru o conștientizare mai mare a contextului scenei, chiar și pentru obiecte mici. În cele din urmă, se minimizează pierderea focală în timpul antrenamentului pentru a susține un număr mare de ancore, rezultând din variabilitatea mare a scalei. Arhitectura detectorului de palme la nivel înalt este prezentată în Figura 4.3.

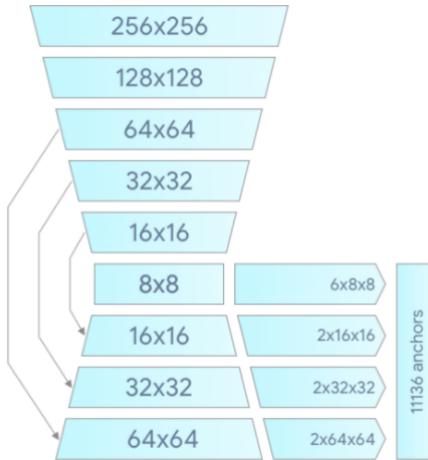


Figura 4.3: Arhitectura modelului de detectarea a pălmii [11]

După rularea detectorului de palme pe întreaga imagine, lucrarea [11] constată că modelul ulterior de repere pentru mâini realizează localizarea precisă a 21 de coordonate 2.5D în regiunile de mâină detectate prin regresie. Modelul învață o reprezentare internă consistentă a poziției mâinii și este robust chiar și în cazul mâinilor parțial vizibile și al auto-ocluziei. Modelul are trei ieșiri, vezi Figura 4.4:

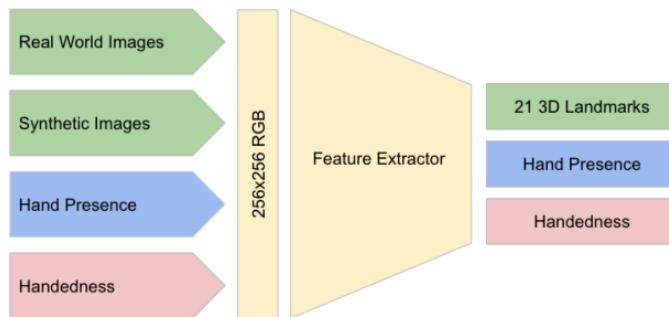


Figura 4.4: Arhitectura modelului de detectarea a reperelor mâinii [11]

1. 21 de repere ale mâinii, constând din coordonatele x, y și adâncimea relativă.
2. Un indicator al prezenței mâinii, care indică probabilitatea prezenței mâinii în imaginea de intrare.
3. O clasificare binară a lateralității, de exemplu, mâna stângă sau dreaptă.

Ca și execuție completă, tehnologia prezentată în [11] vine cu un set extensibil de calculatoare pentru a rezolva sarcini precum inferența modelului, procesarea media și transformările de date pe o varietate largă de dispozitive și platforme. Calculatoare individuale, cum ar fi decuparea, randarea și calculele rețelelor neuronale, sunt optimizate suplimentar pentru a utiliza accelerarea GPU. Graficul MediaPipe pentru urmărirea mâinilor este prezentat în Figura 4.5. Graficul constă din două subgrafice, unul pentru detectarea mâinilor și altul pentru calculul reperelor. Una dintre optimizările cheie pe care le oferă MediaPipe este că detectorul de palme rulează doar atunci când este necesar (destul de rar), economisind o cantitate semnificativă de resurse de calcul. Acest lucru se realizează derivând locația mâinii în cadrele video curente din reperele mâinii calculate în cadrul anterior, eliminând necesitatea de a aplica detectorul de palme pe fiecare cadrul. Pentru robustețe, modelul de urmărire a mâinilor generează și un scalar suplimentar care captează încrederea că o mâna este prezentă și aliniată corespunzător în cadrul decupat de intrare. Doar atunci când încrederea scade sub un anumit prag, modelul de detectare a mâinilor este reaplicat pe cadrul următor.

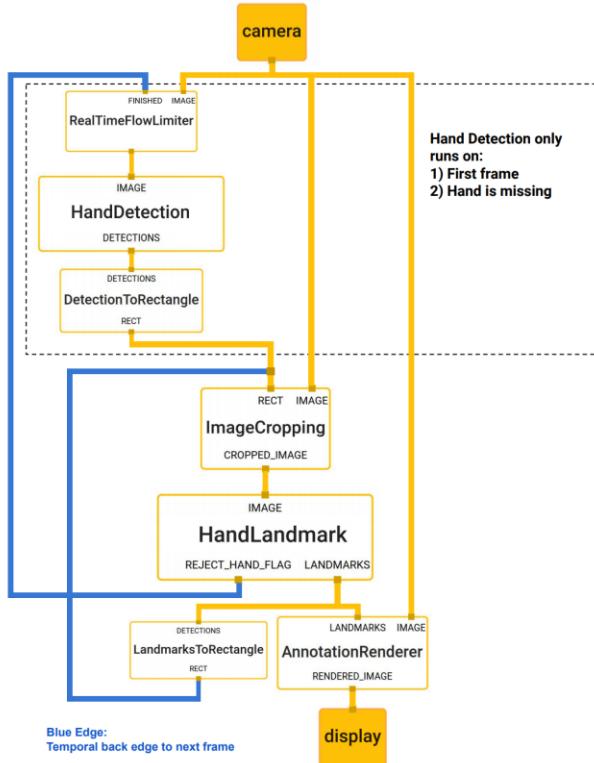


Figura 4.5: Flux de execuție MediaPipe [11]

Alte lucruri esențiale pentru detectarea în timp real pe care le detine MediaPipe sunt filtrarea temporală și ajustarea dinamică. Pentru a asigura o urmărire fluidă și stabilă, MediaPipe implementează tehnici de filtrare temporală care reduc fluctuațiile și zgomotul în predicțiile punctelor cheie între cadrele consecutive. Acest lucru este esențial pentru a evita tremurul și instabilitatea în urmărirea mișcărilor mâinii. Sistemul ajustează dinamic predicțiile pentru a compensa mișcările bruște sau schimbările de poziție, menținând astfel o urmărire consistentă.

4.8. Heroku

Pentru *hosting-ul* serverului de *backend* se va folosi Heroku, fiind foarte accesibil și ușor de configurat, astfel fiind una dintre cele mai potrivite variante pentru această aplicație.

Heroku, prezentată în [12], este o platformă *cloud* care facilitează dezvoltarea, implementarea și gestionarea aplicațiilor. Aceasta oferă un mediu de dezvoltare integrat (IDE) în *cloud*, permitând crearea de aplicații într-un mod simplu și eficient, fără a se preocupa de gestionarea infrastructurii de bază.

Caracteristicile principale care reprezintă și avantajele platformei Heroku pentru a alege acest tip de *hosting* conform [12] sunt:

1. **Platformă ca Serviciu (PaaS):** Heroku este un exemplu clasic de Platformă ca Serviciu (PaaS), care oferă un set de servicii care simplifică dezvoltarea aplicațiilor. Aceasta permite concentrarea pe scrierea codului și pe logica aplicației, fără a se preocupa de gestionarea serverelor sau a infrastructurii.
2. **Extensibilitate:** Heroku oferă o piață vastă de extensii care pot fi integrate în aplicații pentru a adăuga funcționalități suplimentare, cum ar fi baze de date, servicii de *cache*, monitoare de performanță și multe altele.
3. **Gestionarea Mediilor de Dezvoltare:** Heroku facilitează gestionarea diferitelor mediilor de dezvoltare (de exemplu, dezvoltare, testare, producție) prin intermediul conceptului de *pipelines*, care permit promovarea codului și a configurațiilor între medii.
4. **Implementare Continuă:** Platforma suportă integrarea continuă și livrarea continuă (CI/CD), permitând implementarea modificărilor de cod rapid și eficient, cu timp de nefuncționare minim.
5. **Securitate și Scalabilitate:** Heroku asigură securitatea aplicațiilor prin actualizări automate și gestionarea certificatelor SSL. De asemenea, permite scalarea aplicațiilor vertical (creșterea puterii unui singur server) și orizontal (adăugarea de servere suplimentare) în funcție de necesitățile aplicației.

4.9. GitHub Pages

Pentru găzduirea interfeței aplicației web s-a ales folosirea GitHub Pages, una dintre cele mai fiabile și accesibile variante pentru aplicații minime, fiind ușor de configurat și accesat.

Utilizând GitHub Pages, orice persoană poate accesa aplicația implementată din orice loc al globului, în orice moment, având un *deployment* continuu. Accesul se face prin linkul https://mrradu1.github.io/SR_SignLanguageRecognition/.

GitHub Pages este documentat în [13] și este un serviciu oferit de GitHub care permite utilizatorilor să găzduiască pagini web direct dintr-un depozit GitHub. Aceasta este ideal pentru proiecte *open-source*, site-uri personale, documentație și alte pagini web statice. GitHub Pages este gratuit și simplu de utilizat, integrându-se perfect cu fluxul de lucru GitHub.

Avantajele folosirii GitHub Pages prezentate în [13] sunt:

1. **Gratuitate:** GitHub Pages este gratuit pentru utilizatori și organizații, ceea ce îl face ideal pentru proiecte *open-source*, documentație și site-uri personale.
2. **Ușurință în Utilizare:** Integrarea cu GitHub simplifică fluxul de lucru pentru dezvoltatori, permitându-le să folosească aceleași instrumente de control al versiunii și colaborare pe care le utilizează pentru dezvoltarea de software.
3. **Automatizare:** Utilizarea GitHub Actions pentru automatizare permite dezvoltatorilor să implementeze site-uri web rapid și eficient, integrând testarea și construirea automatizată în fluxul de lucru.
4. **Colaborare Facilitată:** Controlul versiunilor prin Git și posibilitatea de a colabora cu alți dezvoltatori fac din GitHub Pages o soluție ideală pentru proiecte care necesită colaborare și feedback constant.

4.10. OpenCV

Pentru achiziționarea imaginilor în *real-time* și pentru gestionarea lor înainte de a le trimite ca date de intrare pentru detecție se folosește OpenCV.

OpenCV (*Open Source Computer Vision Library*) este o bibliotecă *software open-source* dedicată prelucrării și analizei imaginilor și videoclipurilor. Aceasta oferă sute de funcții pentru diverse aplicații de viziune computerizată și învățare automată, fiind utilizată pe scară largă în cercetare și industrie.

Printre caracteristicile principale ale OpenCV se numără:

- **Funcționalități Extinse de Prelucrare a Imaginilor:** OpenCV oferă o gamă largă de funcții pentru prelucrarea și analiza imaginilor, inclusiv filtrare, detecție de margini, transformări geometrice și corecții de culoare.
- **Integrare cu Biblioteci de Învățare Automată:** OpenCV poate fi integrat cu biblioteci de învățare automată și *deep learning*, cum ar fi TensorFlow, PyTorch și Caffe, pentru a îmbunătăți performanța și precizia algoritmilor de viziune computerizată.
- **Capabilități de Video Analytics:** OpenCV permite analiza videoclipurilor în timp real, oferind funcții pentru urmărirea obiectelor, stabilizarea imaginii și segmentarea video.
- **Programare Multiplatformă:** Alt beneficiu ar fi portabilitatea OpenCV, oferind ghiduri pentru utilizarea bibliotecii pe diferite platforme și în diverse limbi de programare, subliniind flexibilitatea și utilitatea sa în dezvoltarea de aplicații diverse.

4.11. TensorFlow

Pentru crearea, rularea și integrarea în aplicație a modelelor folosite de această implementare, s-a ales TensorFlow. TensorFlow este prezentat în [14] ca un *framework* complex, conceput pentru acest tip de aplicări asupra modelelor de învățare automată și *deep learning*, funcționând pe o varietate de platforme și sisteme *hardware*. Este foarte utilizat și cunoscut în toate domeniile, de la cercetare academică la producție industrială, avantajele sale fiind o arhitectură modulară și flexibilă.

Arhitectura TensorFlow conține următoarele elemente:

- **Componente de bază:**

1. *Tensors*: Acestea reprezintă unitățile de bază ale datelor în TensorFlow, fiind ca structură o matrice multidimensională care conține date de diferite tipuri.
2. *Graphs*: Pentru a reprezenta aceste date, TensorFlow folosește un graf computațional. Pe acest graf se pot observa și operațiile asupra datelor conținute, acestea fiind reprezentate ca un nod în graf, marginile grafului fiind tensorii folosiți de către aceste operații.
3. *Sessions*: Toate aceste operații, cât și transferul de tensori între acestea, sunt gestionate și coordonate de către o sesiune, în timpul căreia se întâmplă toate procesele ce aparțin de graf.

- **Biblioteci și API-uri exemplificate în [14]:**

1. Ca nucleu, TensorFlow are **Core TensorFlow API**, prin care se pot accesa toate funcționalitățile fundamentale ale acestui *framework*, printre acestea fiind operațiile matematice, cât și funcțiile de creare și manipulare a grafului modelului.
2. Pentru *pipeline-uri* de date care să fie eficiente și scalabile, cât și să ajute la o performanță optimizată a antrenării modelului, incluzând preprocesarea și încărcarea datelor, se folosește alt API, acesta fiind **tf.data API**.
3. Cel mai optimizat și cel mai potrivit API ce se ocupă de construcția, antrenarea și perfecționarea modelelor este **Keras API**, acesta fiind integrat mai recent în TensorFlow, datorită relației strânse pe care aceste două *framework-uri* o au. Keras este unul dintre cele mai folosite *framework-uri* pentru o ușoară creare a unui prototip de rețele neuronale.
4. În urma tuturor acestor operații și procese complexe, evident că e nevoie de monitorizare, evaluare și observare atentă a modelelor, cât și a antrenării acestora. De aceste aspecte se ocupă o altă unealtă inclusă în TensorFlow, aceasta fiind **TensorBoard**. Aceasta dispune de grafice și histograme pentru principalii termeni de evaluare și performanță ai modelului și ai procesului de antrenare.
5. O altă unealtă, care la fel nu neapărat de crearea și pregătirea modelului, oferit de TensorFlow, este **TensorFlow Serving**, care se ocupă de *deployment*-ul în producție al muncii efectuate. Are ca avantaje eficiență și versionarea modelelor realizate pe parcurs, cât și administrarea acestora.

- **Flow-ul Principal de Execuție în TensorFlow prezentat în [14] și prezentat în figura 4.6:**

1. **Pregătirea Datelor:** Primul pas pentru a putea construi orice tip de model este clar nevoie de un set de date. Acestea trebuie pregătite special, iar la acest aspect ne poate ajuta *tf.data API*, construind *pipeline-uri* de date, normalizare, augmentare sau împărțirea în loturi a datelor pe care dorim să le folosim.
2. **Definirea Modelului:** Aici este pasul unde este gândită toată arhitectura modelului și nevoile lui. Se aleg procesele și legăturile din graful modelului, lucruri care sunt realizate prin API-urile Core TensorFlow API și Keras, acestea fiind specializate pe crearea, stratificarea și configurarea rețelelor neuronale.
3. **Compilarea Modelului:** Având arhitectura stabilită, rămâne configurarea unor aspecte de antrenare, cum ar fi optimizatorul, funcția de pierdere, metricile de evaluare, cât și rata de învățare și mulți alți parametri specifici, toți fiind accesăți prin API-ul Keras.
4. **Antrenarea Modelului:** Aici se folosesc datele pregătite inițial pentru antrenarea modelului conceput, tot cu ajutorul API-ului Keras, prin metoda *'fit'*. Datele trec prin rețea, modelul fiind ajustat și "învățat" pentru a minimiza pierderea și a oferi o acuratețe cât mai ridicată în predicțiile ulterioare.
5. **Evaluarea Modelului:** Acest pas include atât evaluarea modelului prin metode specifice pregătite de Keras, folosindu-se de un set de date de test pregătit inițial. La acest pas ajută și TensorBoard, oferind grafice și informații bazate pe antrenarea din pasul anterior. Informații precum:

- *epoch accuracy*
- *epoch loss*
- *confusion matrix*
- *classification report*
- *f1 score*
- *recall*
- *support*
- *macro avg*
- *weighted avg*

fiind cele mai generale și importante metrici de evaluare.

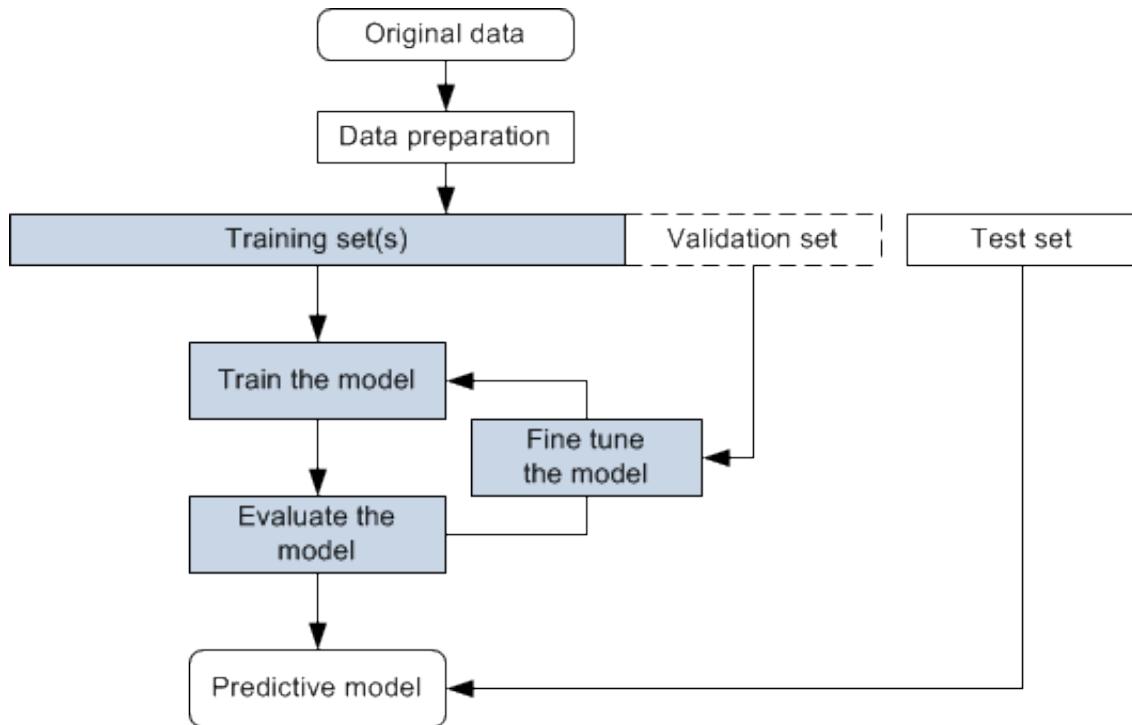


Figura 4.6: Flux de execuție TensorFlow Model [15]

Pe partea de avantaje ale TensorFlow, s-au regăsit următoarele avantaje în cartea [16] care au adus un aport pozitiv proiectului:

1. **Flexibilitate:** TensorFlow permite crearea de modele complexe folosind o gamă largă de algoritmi și tehnici de învățare automată și *deep learning*. Suportă atât modelele de învățare automată tradiționale, cât și rețele neuronale profunde (*deep neural networks*), CNN (*convolutional neural networks*), RNN (*recurrent neural networks*) și multe altele.
2. **Scalabilitate:** TensorFlow poate rula pe diverse platforme și *hardware*, inclusiv CPU-uri, GPU-uri și TPU-uri (*Tensor Processing Units*). Aceasta permite scalarea eficientă a aplicațiilor de învățare automată de la calculatoare personale la clustere de servere și *cloud*.
3. **Performanță:** TensorFlow este optimizat pentru performanță, permitând rularea eficientă a modelelor de învățare automată pe *hardware* divers. Suportul pentru accelerarea pe GPU și TPU permite antrenarea rapidă a modelelor mari și complexe.

4.12. CNN

Modelul principal al aplicației, cel de recunoaștere și clasificare a semnelor, are la bază un CNN (*Convolutional Neural Networks*), fiind unul dintre cele mai populare tipuri de modele de învățare automată pentru clasificarea de imagini. Fiind pus pe înțelesul tuturor în articolul [17], un CNN este un tip de rețea neuronală inspirată din procesele biologice din cortexul vizual al animalelor, fiind construită din neuroni care sunt capabili să învețe. Cu destule date și putere computațională, un CNN poate atinge performanțe supraomenești în ceea ce privește cerințele vizuale. Acestea au revoluționat domeniul și sunt folosite de companii precum Google, Facebook și alți giganți pentru a crește puterea de recunoaștere a imaginilor. În figura 4.7 sunt prezentate procesele unui CNN.

Cum funcționează un CNN?

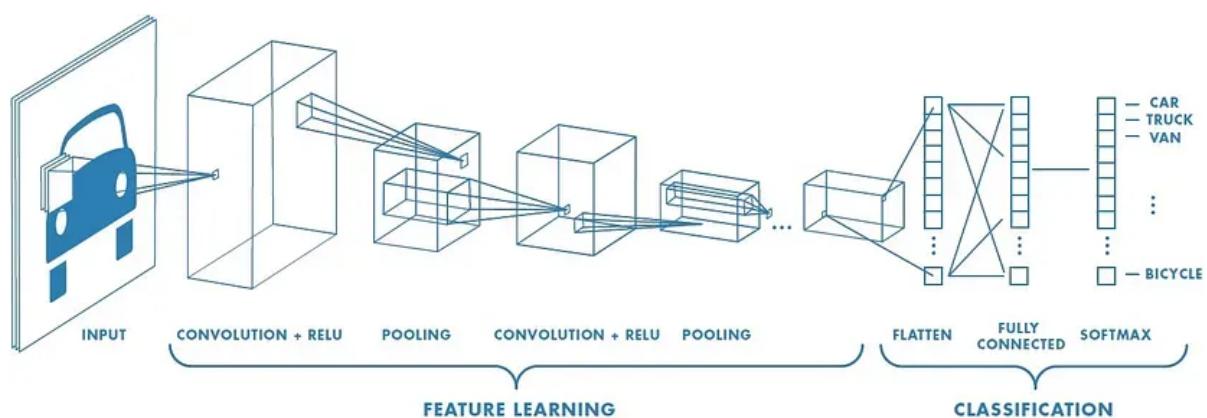


Figura 4.7: Arhitectura CNN

Metoda de bază din CNN este convolução, o operație liniară în care un filtru de dimensiuni mici este aplicat peste imaginea de intrare pentru a detecta caracteristici precum margini sau forme. Filtrele trec peste toată dimensiunea imaginii și fac produsul scalar dintre filtru și imaginea de bază, producând o hartă de activare (*activation map*) [17].

Aceste hărți de activare sunt introduse în straturi de pooling care reduc dimensiunile hărtiilor pentru a putea fi gestionate mai ușor. Pe lângă asta, face modelul mai eficient și mai robust. Ca ultim strat, există un strat complet conectat care clasifică imaginea de input în niște clase predefinite [17].

Pentru a construi un CNN, se definește o arhitectură prin selecția de parametri cum ar fi numărul de filtre, dimensiunea de pooling sau dimensiunea filtrelor. Apoi, această rețea este antrenată pe un set de date complex [17].

Straturi principale într-un CNN

1. **Stratul Convoluțional (Figura 4.8)** - Acesta aplică operația de bază, convoluția, trecând un filtru peste toată imaginea. Filtrul detectează margini, curbe sau chiar forme. Aplicarea mai multor filtre poate detecta mai multe caracteristici [17].

Operația de convoluție combină imaginea de intrare și filtrul pentru a crea o hartă de activare care arată locația și puterea caracteristicilor detectate.

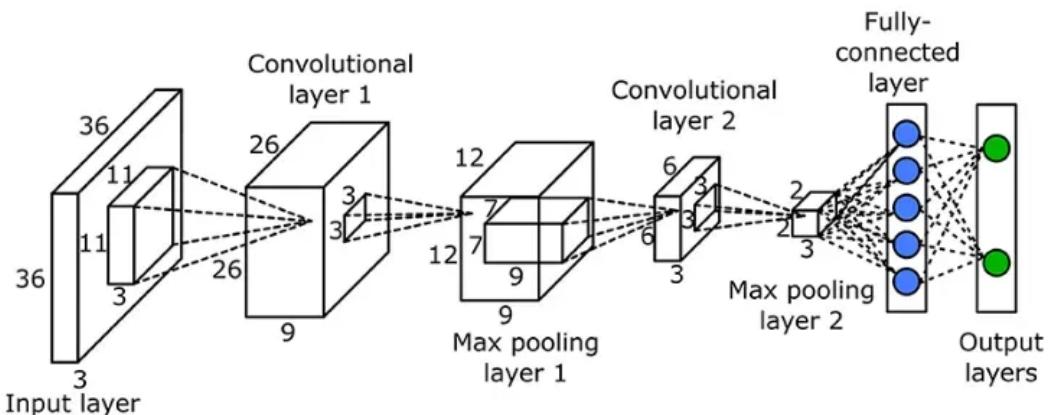


Figura 4.8: Strat Convoluție [17]

2. **Strat de pooling (Figura 4.9)** - Aceste straturi sunt inserate între straturile de convoluție. Ele scad dimensiunile hărților generate de stratul convoluțional pentru a reduce numărul de parametri, a controla supra-învățarea modelului și a face rețeaua invariantă la mici translații, reținând doar informațiile importante [17].

Cele mai comune tipuri de *pooling* sunt *max pooling*, care ia cea mai mare valoare din filtru, și *average pooling*, care ia media.

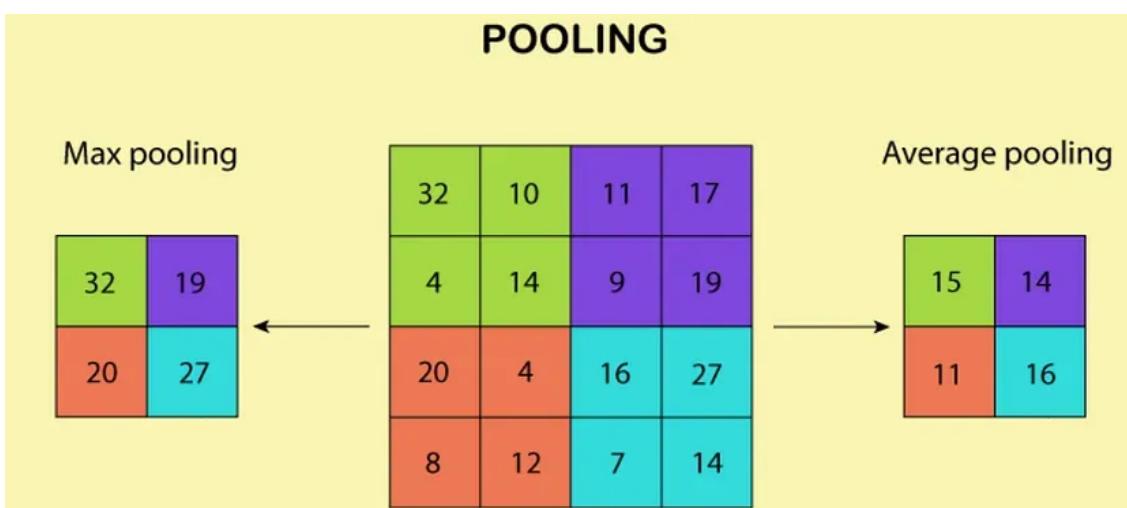


Figura 4.9: Strat Pooling [17]

3. **Strat de Activare (Figura 4.10)** - Aceste straturi aplică o funcție neliniară pe datele de ieșire de la stratul de pooling. Aplicarea acestei neliniarități în model îl ajută să învețe reprezentări mai complexe ale datelor de intrare [17].

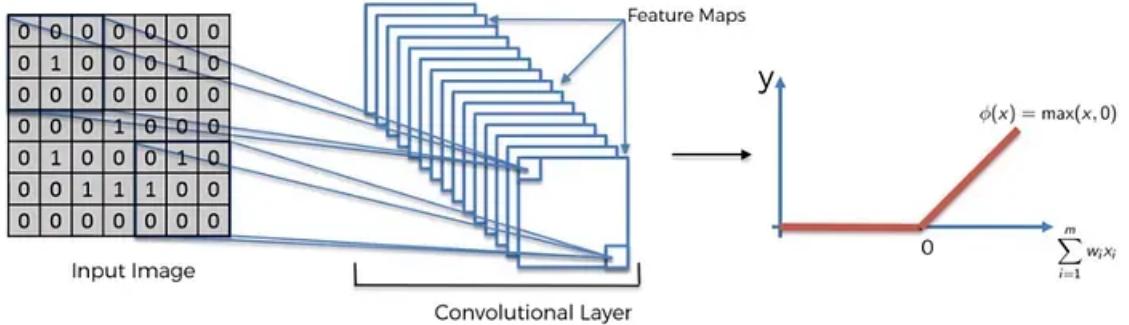


Figura 4.10: Strat de Activare [17]

Exemple de funcții neliniare explicate în Figura 4.11:

- **ReLU:**

$$f(x) = \max(0, x)$$

- **Tanh:**

$$f(x) = \tanh(x)$$

- **Sigmoid:**

$$f(x) = \frac{1}{1 + \exp(-x)}$$

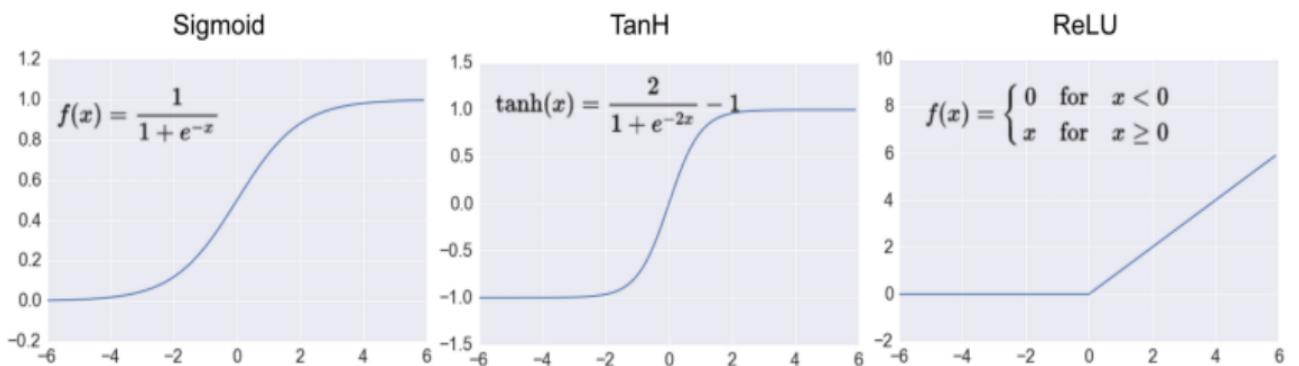


Figura 4.11: Funcții neliniare [18]

4. Strat **dropout** (Figura 4.12) - Aceste straturi renunță aleatoriu la unii neuroni din rețea în timpul antrenării și sunt folosite pentru a preveni *overfitting*-ul, lucru care ar face modelul să memoreze datele de antrenare în loc să generalizeze pentru date noi și nevăzute [17].

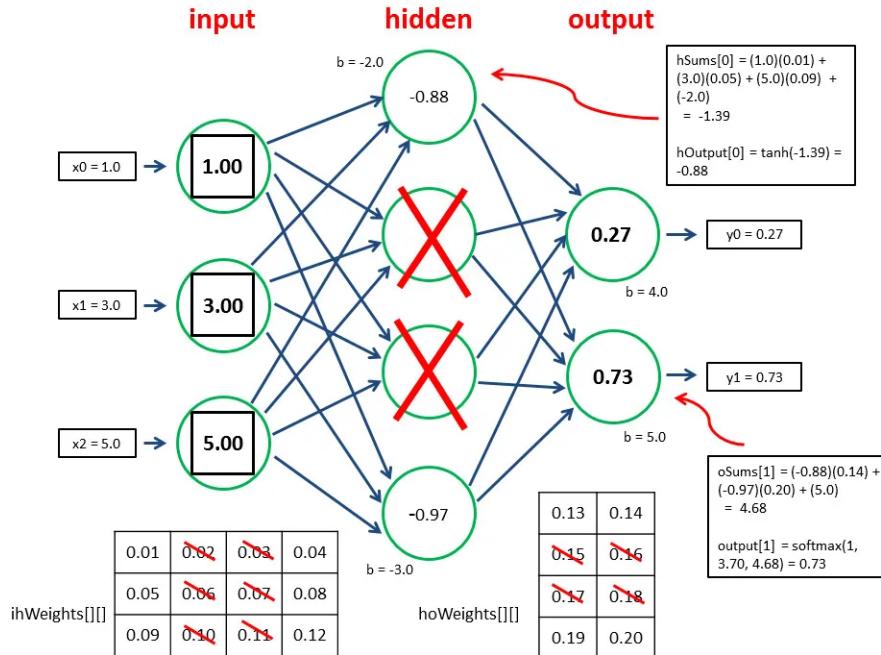


Figura 4.12: Strat de *dropout* [17]

5. Strat **dens** (Figura 4.13) - După toate straturile care extrag caracteristicile imaginii, acestea sunt combinate pentru a face o clasificare finală, iar acest lucru este realizat de un strat dens. Datele sunt aplatizate, iar apoi este efectuată o sumă ponderată peste care se aplică o funcție de activare pentru rezultatul final [17].

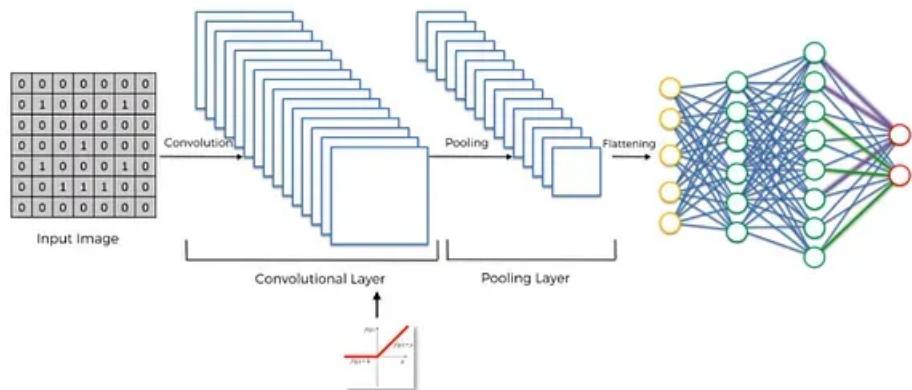


Figura 4.13: Strat dens [17]

4.13. LSTM

S-a prezentat mai sus că predicția se face pe un vector de cadre adunate, din cauza că avem unele semne ale limbajului definite ca niște mișcări, și nu ca și niște simple poziții. Pentru a reține și folosi eficient datele din cadrele precedente, integrăm în model straturi de RNN (*Recurrent Neural Networks*), care memorează și se folosesc de starea precedentă pentru gestionarea datelor noi. Acestea folosesc bucăți de rețele neuronale care arată ca în figura 4.14:

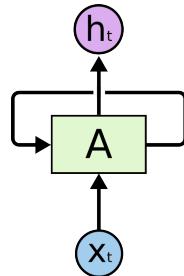


Figura 4.14: Celula RNN [19]

Dacă ar fi să descompunem acest tip de rețea neuronală, ar arăta ca în figura 4.15:

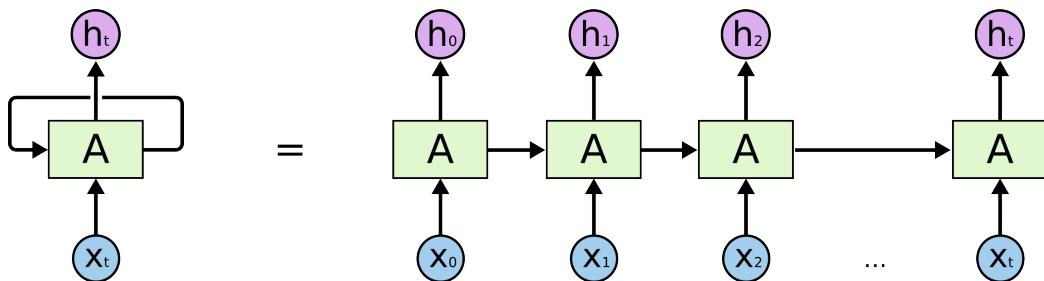


Figura 4.15: Celula RNN extinsă [19]

Sunt doar copii ale aceleiași rețele, care se folosesc de date de la nivelul trecut. Problema la RNN-uri apare când trebuie să folosim date de la pași mult prea îndepărtați, lucru pentru care se ocupă mult mai bine o versiune îmbunătățită a lor, numite LSTM (*Long Short Term Memory*) [19].

Arhitectura LSTM

O unitate dintr-un LSTM se numește unitate de memorie, care reprezintă un pas al rețelei. Aceasta are 4 rețele neuronale mai mici, numite porți. Trei dintre aceste porți se ocupă cu selecția datelor. Se numesc poarta de "uitare", poarta de intrare și poarta de ieșire. Din numele lor, se poate observa cu ce se ocupă. Poarta de "uitare" șterge informații nefolositoare din memorie, poarta de intrare adaugă informații noi iar poarta de ieșire folosește informații actuale din memorie [20].

Cea de-a 4-a poartă se ocupă doar cu crearea de noi date pentru a fi inserate în memorie. O reprezentare este afișată în figura 4.16.

În fiecare unitate de memorie avem 3 vectori de intrare. Doi dintre ei vin de la unitatea precedentă, un vector de memorie (C) și unul de output (H), iar cel de-al 3-lea este input-ul la momentul actual (X). Vectorul H este memoria de scurtă durată, de la

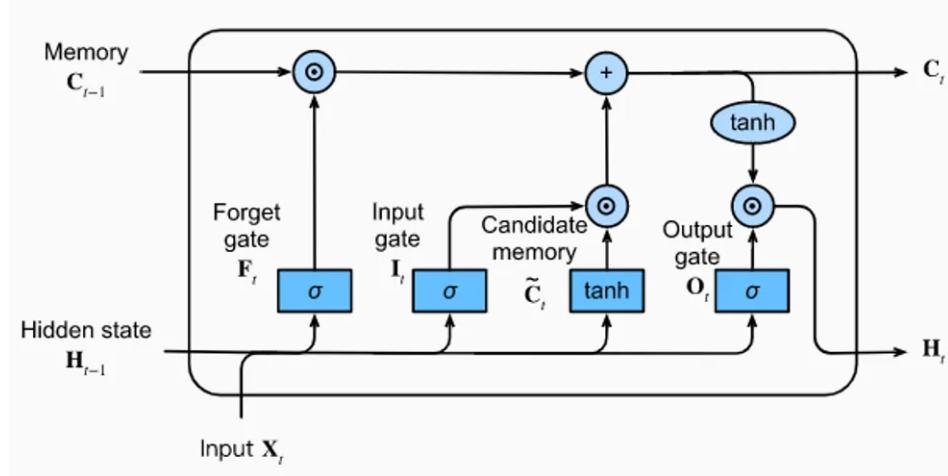


Figura 4.16: Arhitectura LSTM [20]

pasul anterior, iar C este memoria de lungă durată, de-a lungul întregului proces. Cei 3 vectori, împreună cu cele 4 porți, vor rezulta în 2 noi vectori ca ieșire, vectorii C și H modificați [20].

Porțile unui LSTM

Cele 3 porți menționate inițial sunt selectoare de informație. Sarcina lor este de a genera vectori de selecție, cu valori între 0 și 1. Înmulțind acești vectori cu datele noastre, cele cu poziții pe 0 vor dispărea și cele cu 1 vor rămâne. Toate cele 3 porți folosesc funcția sigmoid prezentată mai sus. Toate cele 3 porți folosesc vectorul H concatenat cu vectorul X [20].

Poarta de "uitare"

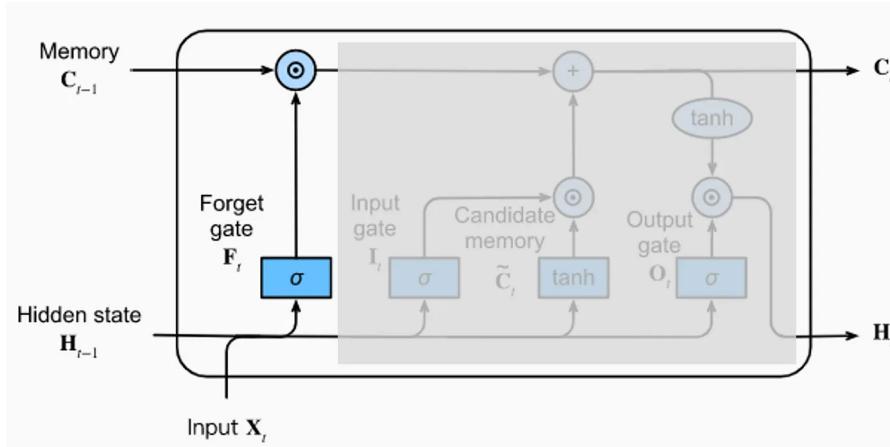


Figura 4.17: Poarta de "uitare" [20]

Prin funcția sigmoidală, din vectorul format din X și H , se generează un vector selector care alege ce date vor fi șterse și ce date rămân din memoria venită de la pasul anterior.

Poarta de intrare și memoria candidat

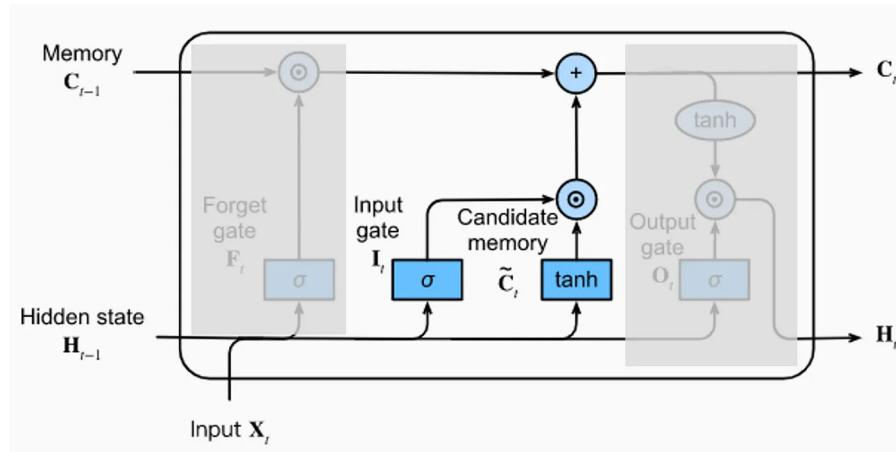


Figura 4.18: Poarta de intrare și memoria candidată [20]

Pentru generarea de noi date pentru inserarea în memorie, folosim 2 porti. Memoria candidată e responsabilă de generarea unui vector care să fie adăugat la memorie. Poarta de intrare generează un vector selector care este înmulțit cu vectorul propus, pentru a selecta ce date vor fi cu adevărat introduse în memorie. După adăugare, nu vor fi noi modificări pe vectorul de memorie de lungă durată (C) [20].

Poarta de ieșire

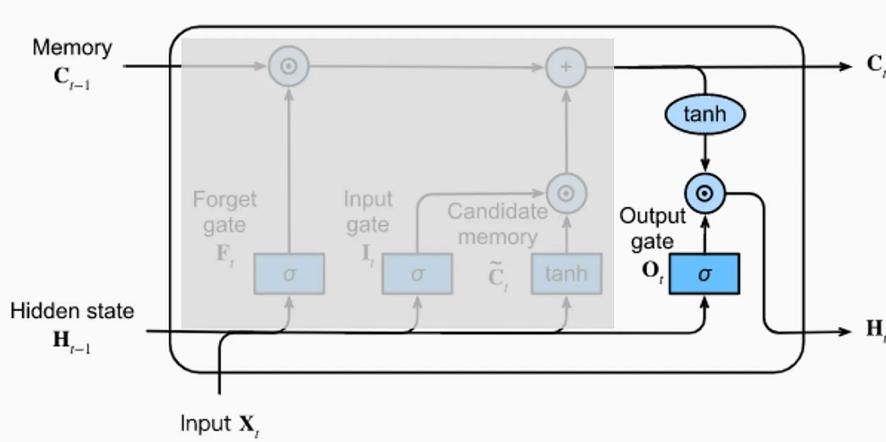


Figura 4.19: Poarta de ieșire [20]

Aceasta determină rezultatul, sau altfel zis memoria de scurtă durată, prin folosirea unei funcții hiperbolice pe vectorul C care apoi este înmulțit cu un vector selector produs de poarta de ieșire.

4.14. Transformers

Aplicația implementată are și o parte de propunere a unui cuvânt sau a unei propoziții, bazată pe textul scris până acum. Această sugestie de corectare sau completare a textului se face printr-un model de tip *transformers*.

Modelele transformers pot învăța relații complexe și pot procesa datele în paralel, fiind folosite în recunoașterea vorbirii, traducerea automată sau modelele de limbaj.

Cum funcționează un Transformer?

Acesta se bazează pe un mecanism de atenție, modelul atribuind anumite scoruri fiecărei părți din datele de intrare, astfel fiind atent doar la anumite părți atunci când generează rezultatul final. O schemă bloc este prezentată în Figura 4.20.

În aceste transformere există mecanisme de atenție numite *multi-head*, unde ”capetele” de atenție lucrează în paralel pentru a strânge informațiile și relațiile dintre cuvinte. Neavând o structură secvențială ca RNN-urile, se folosesc încorporări pozitionale pentru a menține informațiile despre cuvintele din secvență [21].

Ca arhitectură, este format din două părți principale:

- **Codificator:** Acesta primește datele de intrare și creează un set de caracteristici ale datelor, încercând să le înțeleagă.
- **Decodificator:** Se folosește de acest set de informații și de alte date de intrare pentru a genera un rezultat.

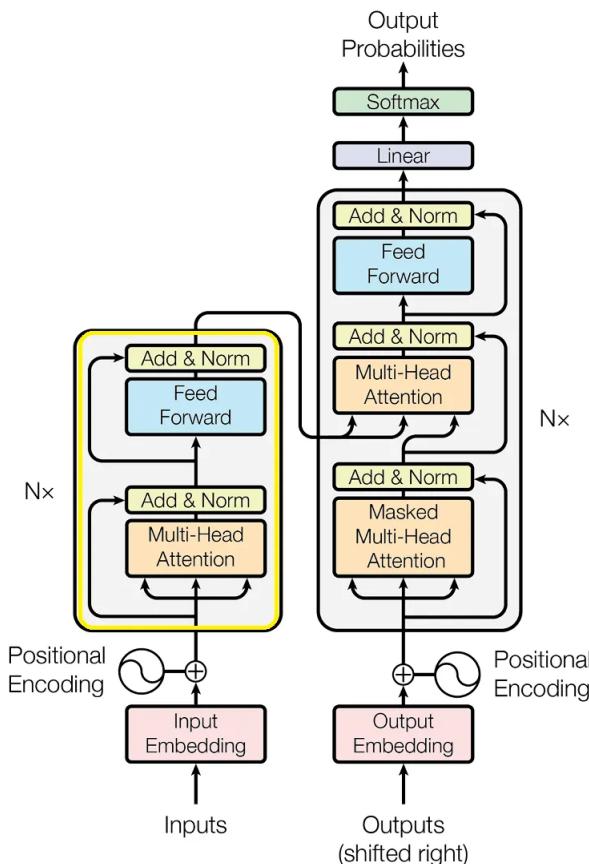


Figura 4.20: Arhitectura Transformer [20]

Capitolul 5. Proiectare de detaliu și implementare

Acet capitol prezintă în detaliu toată arhitectura sistemului, componentele și metodele care sunt folosite pentru a atinge funcționalitățile dorite, cât și tot procesul de colectare a datelor, compunerea rețelei neuronale convoluționale, antrenarea, evaluarea și testarea ei. Se vor prezenta diagrame care exemplifică structura sistemului, cât și fluxul de procese care duc la recunoașterea semnelor din limbajul semnelor, la aceasta adăugându-se funcționalitățile secundare precum conversia textului în vorbire, alături de sugestie sau corectarea textului deja recunoscut. Tot în acest capitol va fi prezentată și partea de *deployment* a aplicației, care o face accesibilă într-un mod continuu oricărui individ care are nevoie de această unealtă, de oriunde de pe glob și în orice moment.

Aplicația este un sistem SLR (*Sign Language Recognition System*) care are ca obiectiv principal traducerea semn cu semn a limbajului semnelor. Se va pune accent pe această funcționalitate principală, luând pas cu pas procesele acesteia. Sistemul se împarte în trei componente mari: o aplicație ce ține complet de colectarea datelor, construirea, antrenarea și evaluarea rețelei neuronale, aplicația client pe care o accesează utilizatorul și serverul la care clientul accesează servicii.

5.1. Rețea neuronală pentru recunoașterea semnelor

Dintre cele trei componente, clientul și serverul sunt legate unul de altul, iar aplicația pentru rețea este independentă de ele. Din acest motiv, este esențială prezentarea în detaliu a implementării rețelei. Aceasta este implementată în limbajul Python 3.10, din cauza necesității compatibilităților bibliotecilor utilizate. Pentru o citire mai lizibilă și pentru o prezentare mai elegantă și mai aerată a proceselor, s-a ales scrierea totală a părții de rețea neuronală într-un Jupyter Notebook. Această alegere a adus multe beneficii, fiind posibilă rularea codului în celule separate, fiind mai facilă testarea și evaluarea codului pas cu pas. Prezentarea mai elegantă și aerată este asigurată prin faptul că pas cu pas se pot adăuga documentație, cât și elemente grafice alături de codul scris. Fiind o rețea neuronală ce se bazează pe date, vizualizarea lor și evaluarea rețelei pe numeroase criterii prin elemente grafice generate de Matplotlib și adăugarea lor în documentarea aplicației a fost un mare avantaj al unui Jupyter Notebook.

Folosind Jupyter Notebook, s-a putut organiza această aplicație pe capitole, mai exact în opt capitole:

- 1. Instalarea pachetelor Python**
- 2. Importarea bibliotecilor**
- 3. Crearea variabilelor**
- 4. Definirea de funcții folosite**
- 5. Colectarea datelor pentru antrenare**
- 6. Crearea și antrenarea rețelei**
- 7. Salvarea modelului în diferite formate**
- 8. Evaluarea modelului**

- **Instalarea pachetelor Python**

Pe partea de aplicație, s-au folosit următoarele pachete sau tehnologii:

- **TensorFlow** - se ocupă de toată partea de creare și antrenare a modelului, cât și de salvarea lui.
- **TensorFlowJS** - sistemul își dorește să folosească rețeaua neuronală în partea de client, astfel avem nevoie de o conversie a modelului.
- **OpenCV** - această bibliotecă se ocupă de toate necesitățile legate de conectarea la camera web și colectarea sau procesarea constantă de imagini.
- **MediaPipe** - tehnologia care se ocupă de detecția mâinii și a celor 21 de repere ale ei, date pe care le folosim pentru antrenare și predicție.
- **Scikit-learn** - pentru evaluarea mai amănunțită a modelului.
- **Matplotlib** - generarea de grafice pentru vizualizarea datelor și a rezultatelor.

- **Importarea bibliotecilor**

Din pachetele instalate, aici s-a importat în proiect doar elementele esențiale, precum straturile modelului, optimizatoarele și funcțiile de catalogare a datelor.

- **Crearea variabilelor**

În acest capitol se definesc niște variabile esențiale folosite de-a lungul proceselor ulterioare. Principalele variabile sunt obiectul instantă al modelului MediaPipe Hands, vectorul ce conține toate literele alfabetului plus semnele adiționale, cât și parametrii impliciti ai algoritmilor sau ai rețelei neuronale.

- **Definirea de funcții folosite**

Aici sunt prezente două funcții principale folosite de-a lungul aplicației:

- **getData(results)**

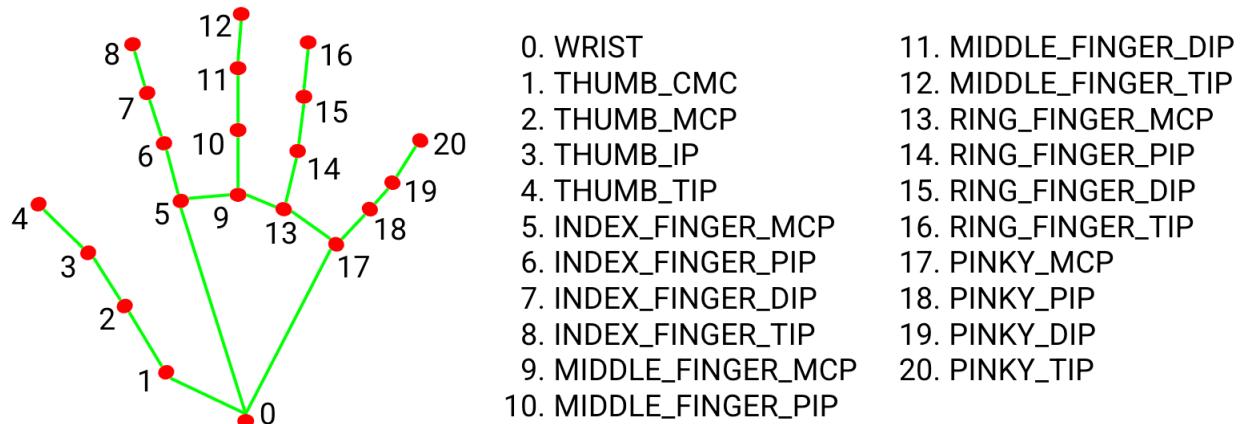


Figura 5.1: Repere Mâini [22]

Înainte de a prezenta funcționalitatea funcției *getData*, trebuie prezentat exact ce date se folosesc pentru rețeaua neuronală. Algoritmul MediaPipe va returna dacă există o mână sau două în imaginea primită și care este fiecare, stânga sau, implicit, dreapta. Dacă există mâini în imagine, pentru fiecare dintre ele, MediaPipe Hands va returna un vector de 21 de repere ale mâinii, prezentate în imaginea de mai sus. Fiecare reper este un alt vector de 3 coordonate spațiale: x, y și z, prezentate în figura 5.1. Aceste coordonate arată poziția fiecărui reper, iar pe baza acestora va fi antrenată rețeaua, cât și predicțiile ulterioare vor fi făcute utilizând acest tip de date.

Funcția *getData* primește rezultatele de la MediaPipe pe care le verifică pentru existența unei mâini. În caz că nu se regăsește nicio mână în cadru, se va returna un vector de 63 de valori de zero. În cazul în care se regăsește o mână în cadru, sunt mai multe cazuri de funcționare. Sistemul ar fi confuz dacă ar trebui să facă două predicții în paralel pentru ambele mâini; din acest motiv, predicția se face pe mână stângă tot timpul în cazul apariției ambelor mâini. În cazul în care apare o singură mână în cadru, iar aceasta este dreapta, datele sunt transpușe în oglindă pentru a simula o mână stângă. În cazul în care există o singură mână în cadru, iar aceasta este stânga, nu există logică suplimentară. Funcția va colecta cele trei coordonate din fiecare reper al mâinii și le va concatena, rezultând un vector de 63 de valori, având 21 de repere a către trei coordonate fiecare.

- **drawLandmarks(results, frame)**

Funcția *drawLandmarks* este o funcție simplă, dar des folosită, care se folosește de reperele primite ca rezultat de la tehnologia MediaPipe și desenează peste imaginea din timp real un cerc pe fiecare reper al mâinilor, plus legătura dintre ele. Un exemplu este prezentat în figura 5.2.

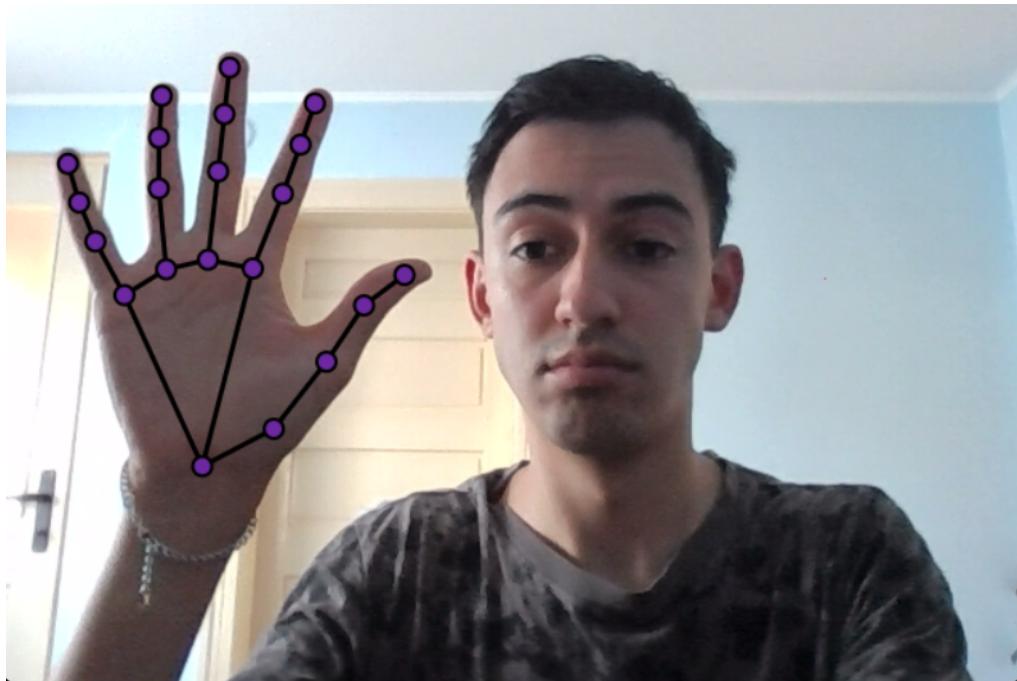


Figura 5.2: Funcționalitate *drawLandmarks*

- Colectarea datelor pentru antrenare

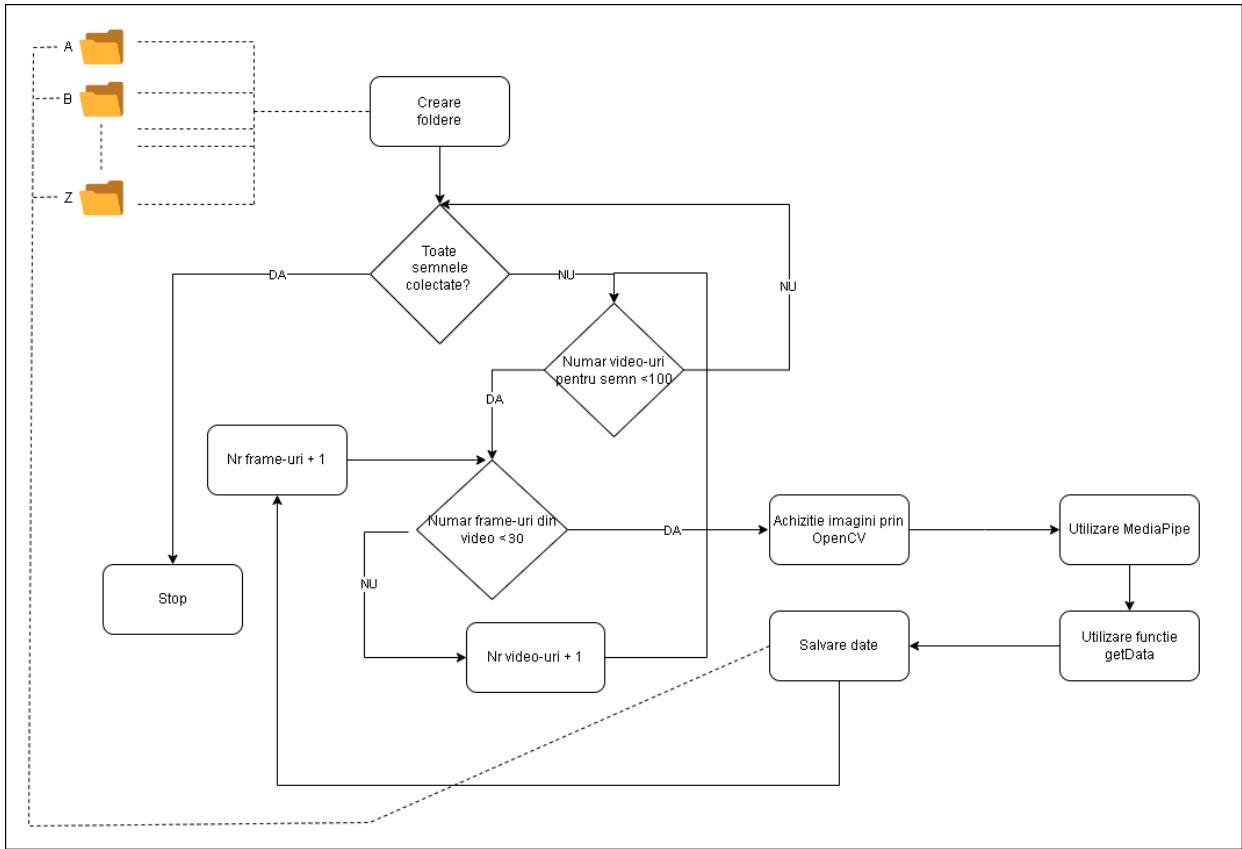


Figura 5.3: Flux de execuție al colectării datelor

Pentru colectarea datelor s-a atașat diagrama din figura 5.3 care descrie pașii în care procesele au loc. Se începe cu crearea în memoria locală a fișierelor pentru organizarea datelor colectate. Această organizare se face pe baza semnelor pe care le conține limbajul. Astfel, vor rezulta 26 de directoare pentru fiecare literă a alfabetului, la care se adaugă 3 semne ajutătoare, deci un total de 29 de directoare. Fiecare director conține alte 100 de directoare numerotate de la 1 la 100, care reprezintă "videourile" colectate. Prin video se face referință la 30 de cadre consecutive, reprezentând o mișcare dinamică a unui semn din limbajul semnelor. Astfel, fiecare din cele 100 de directoare va conține 30 de fișiere de date reprezentând un semn.

După partea de creare de foldere pentru organizare, urmează colectarea propriu-zisă a datelor. Folosind OpenCV, se sustrag imagini din camera web, care apoi sunt transmise algoritmului MediaPipe pentru fiecare cadru. Rezultatele sunt trimise funcției *getData* explicită mai sus. Aceste date sunt apoi salvate local în folderele pregătite. Se colectează toate datele pentru fiecare semn, după care se trece la următorul semn. Între fiecare video există o pauză de o secundă, pentru a putea reveni la postura inițială sau pentru a schimba postura pentru diversificarea datelor. Un exemplu de flux de execuție ar fi: video-ul 1 pentru litera A (30 de fișiere salvate în Data/A/1), pauză de 1 secundă, video-ul 2 pentru litera A (30 de fișiere salvate în Data/A/2), și aşa mai departe.

Datele sunt, în final, împărțite în 2 vectori pentru antrenare și testare în procent de 80/20, fiecare semn fiind împărțit egal între cele 2 părți.

- Crearea și antrenarea rețelei

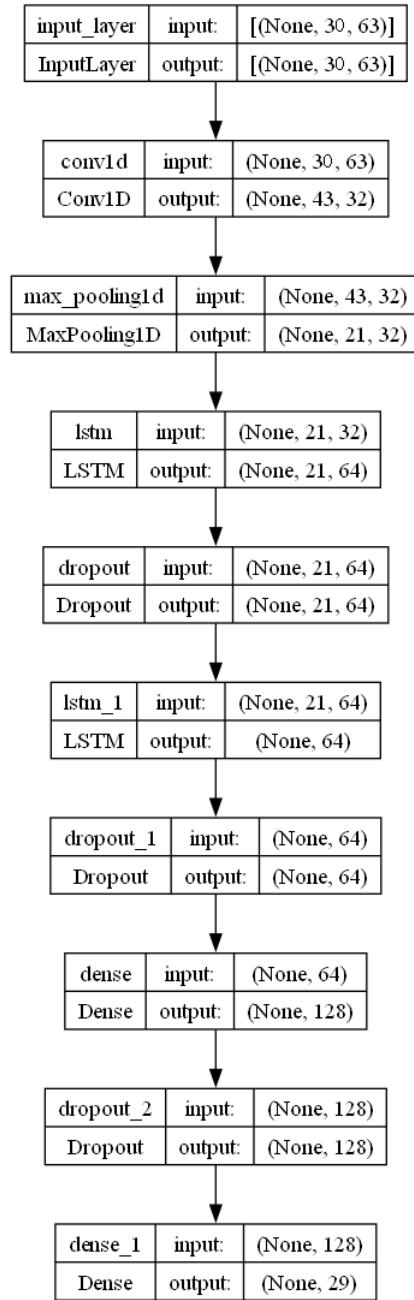


Figura 5.4: Arhitectura rețelei neuronale

Pasul cel mai important din această componentă este evident reprezentat de compunerea și antrenarea modelului. În acest capitol, va fi explicată ideea acestei structuri pentru rețeaua neuronală, importanța fiecărui strat, precum și a parametrilor folosiți. În figura 5.4 este expusă într-un mod grafic structura rețelei.

Va fi luat fiecare strat în parte, după care vor fi combinate utilitățile și va atrage atenția asupra avantajelor rețelei neuronale complete. Rețeaua este de tip secvențială, adică fiecare strat va primi date de la stratul anterior și va trimite stratului următor.

Stratul de intrare

Primul strat este un strat care primește datele inițiale, definind forma lor. Forma datelor de intrare presupune un vector de dimensiune (30, 63), cele 30 de valori fiind reprezentate de 30 de cadre care formează un video, iar cele 63 de valori sunt coordonatele x, y, z ale celor 21 de repere, rezultând astfel 63 de coordonate. Acest vector de (30, 63) reprezintă în esență o matrice bidimensională, în care fiecare rând este reprezentat de un cadru din secvență. Stratul de intrare asigură că rețeaua are datele de antrenare, testare, cât și de predicție corecte și că acestea au aceeași structură mereu.

Stratul convolutional

Stratul de conoluție recunoaște caracteristici locale și modele în secvențele temporale. Mai exact, încearcă să rețină pozițiile la anumite momente de timp ale mâinii și ale degetelor. De exemplu, degetul mic pentru litera "I" la cadrul 15 este ridicat în sus. Se folosește ca nucleu pentru conoluție un nucleu de dimensiune 3, ceea ce presupune că se utilizează datele de la distanțe de 3 cadre. Nucleul trece prin cadre și aplică 32 de filtre pentru a extrage caracteristicile importante.

Pe rezultatul conoluției este folosită funcția de activare ReLU (*Rectified Linear Unit*) pentru a introduce non-linearitate în rețea. Aceste funcții ajută mult rețeaua să învețe relații complexe între caracteristici și evită ajungerea la o combinație liniară de intrări-iesiri.

Funcția ReLU are ca avantaj rapiditatea ei, care nu introduce latență, fiind o simplă comparație și nu o funcție exponențială:

$$f(x) = \max(0, x)$$

Stratul *MaxPooling*

În general, după straturile convoluționale în care se extrag foarte multe caracteristici despre imagine, apare un strat de *max pooling*. Acesta este un strat care se ocupă cu reducerea dimensiunii, eliminând unele dintre caracteristicile extrase. Astfel, se reduce complexitatea computațională și numărul de parametri din rețea, dar se rețin caracteristicile principale.

Pentru a obține această reducere de dimensiune, stratul de *max pooling* folosește ferestre de o anumită mărime, din care selectează valoarea maximă din fiecare fereastră, alegând cele mai dominante caracteristici. În acest caz, s-a ales mărimea ferestrei să fie 2, pentru a nu renunța la prea multe valori deoarece, în situația problemei actuale, și miciile caracteristici fac diferența între unele semne asemănătoare. Prin folosirea unei ferestre de dimensiune 2, vor fi luate 2 câte 2 valori, din care va rămâne valoarea mai mare. Astfel, numărul valorilor noastre se va înjumătăți. Această tehnică de a reduce din dimensiunea datelor ajută atât la scăderea complexității, rezultând o rețea mai rapidă și mai eficientă, cât și la eliminarea zgomotului din imagini, păstrând detaliile de bază. Un alt plus secundar oferit de *pooling* este că ajută să nu se ajungă la supraînvățare, generalizând modelul prin reducerea de informații.

Straturi LSTM

După *max-pooling*, se schimbă puțin tipul de rețea folosită, trecând de la o rețea conlovuțională la una recurrentă, prin adăugarea a două straturi LSTM (*Long Short Term Memory*). Aceste straturi au rolul de a stoca dependențe temporale din datele de intrare. Sunt des folosite pentru date secvențiale, unde ordinea contează, exact cum sunt cele 30 de cadre ale acestei aplicații, care formează un semn.

S-a ales folosirea a două straturi LSTM, pentru a crește capacitatea de a învăța relațiile între cadre. Primul strat se poate concentra pe informațiile de bază, în timp ce al doilea se focusează pe cele mai complexe, pe baza ieșirilor primului strat. Primul strat are parametrul *"return_sequences"* setat la valoarea *True*, ceea ce înseamnă că va returna secvența completă următorului strat LSTM, cu scopul de a oferi setul complet de informații despre legăturile dintre cadre și detaliile lor. Cel de-al doilea strat formează relațiile temporale pe termen lung, condensând totă secvența de date, implicit de cadre, într-un singur vector ca valoare de ieșire.

Ambele straturi LSTM sunt compuse din 64 de unități de memorie. S-a ales această valoare deoarece fiecare cadru are 63 de valori, fiecare unitate axându-se pe o poziție din fiecare cadru. Aceste unități pot fi considerate ca un neuron care păstrează și gestionează informația atât pe termen lung, cât și pe termen scurt. Toate aceste unități funcționează în paralel pentru a procesa cadrul primit ca intrare. Ieșirile tuturor celor 64 de unități sunt, la sfârșit, combinate într-o singură informație.

Straturi Dense

Ca și straturi finale, s-au folosit două straturi Dense pentru a rezuma toate caracteristicile acumulate într-un rezultat final. Aceste două straturi au un rol crucial în consolidarea informațiilor și în clasificarea secvențelor de limbaj al semnelor. Aceste straturi combină caracteristicile extrase din straturile conlovuționale și LSTM, caracteristici precum locația celor 21 de repere împreună cu mișcarea lor dinamică de-a lungul celor 30 de cadre colectate. Combinarea lor duce la extragerea de trăsături mai complexe și mai relevante pentru recunoașterea semnelor. Această combinare, deși sună de parcă am avea mai multe date, duce la reducerea dimensiunii lor, acestea fiind înglobate într-un număr de neuroni redus.

Primul strat Dense folosit combină informațiile într-o rețea formată din 128 de neuroni. Aici se întâmplă practic combinarea principală. Pe ieșirea stratului Dense se folosește din nou o funcție de activare, aceeași ca la straturile precedente, funcția ReLU. Aceasta introduce din nou non-linearitate în rețea. Al doilea strat Dense este format doar din 29 de unități, exact numărul de clase pe care limbajul pe care încercăm să îl recunoaștem îl detine. Această ultimă strat face alegerea finală, clasificând caracteristicile recoltate într-un semn anume. Fiecare neuron din acest strat reprezintă probabilitatea unei clase. Ieșirea folosește o funcție de activare diferită de cea folosită până acum, adică funcția *softmax*, care transformă ieșirile brute ale neuronilor în probabilități care se însumează la 1. Formula funcției *softmax* este:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$$

Straturi Dropout

În arhitectura prezentată în imaginea de mai sus, au fost luate toate straturile rând pe rând, în afară de trei straturi peste care s-a sărit, toate aceste straturi fiind straturi *dropout*. Straturile *dropout* sunt folosite pentru a regla rețeaua, cu scopul de a preveni suprainvățarea. Suprainvățarea înseamnă că modelul ar învăța prea bine datele de antrenament, astfel că nu poate generaliza pe date noi. Straturile de *dropout* ajută la prevenirea suprainvățării prin eliminarea aleatorie a unui număr de neuroni din rețea în timpul antrenării. Eliminarea neuronilor face rețeaua să nu depindă prea mult de unele activări specifice, fiind mult mai generalizată. Se folosește o valoare de 0.5 pentru *dropout*, adică 50%, o valoare destul de comună pentru multe probleme de rețea neuronală. S-au folosit straturi de *dropout* după straturile LSTM și după primul strat Dense. Straturile LSTM au tendința de a memora foarte bine secvențele de cadre, ceea ce duce la suprainvățare. Stratul Dense reduce dimensiunea rețelei, astfel un strat de *dropout* forțează rețeaua să învețe diverse căi și caracteristici, generalizând modelul.

Compilare și antrenare model

Compilarea unui model este un pas esențial în crearea lui și în pregătirea lui pentru antrenare. Deși structura rețelei este bună, o compilare cu parametri greșiti poate scădea semnificativ performanța modelului. Compilarea îi spune modelului CUM să învețe. În acest pas se setează optimizatorul, funcția de pierdere și metricele de evaluare. În faza de optimizare, greutățile rețelei neuronale sunt ajustate pentru a scădea funcția de pierdere. Acestea sunt ajustate pe baza gradientului funcției de pierdere. S-a ales folosirea optimizatorului Adam, unul dintre cei mai populari optimizatori utilizati în rețelele neuronale. Adam calculează rate de învățare adaptive pentru fiecare parametru. Ca și funcție de pierdere, s-a folosit *Categorical Crossentropy*, funcție utilizată pentru probleme de clasificare în mai multe clase. Aceasta măsoară diferența între distribuția de probabilitate prezisă de model și distribuția de probabilitate reală. Metrica de evaluare folosită este cea clasică, cea de precizie. Aceasta măsoară proporția de predicții corecte față de numărul total de predicții.

Antrenarea a fost făcută în 75 de epoci. O iterație presupune procesarea unui lot de date din setul de antrenament, dar nu tot setul. O epocă conține mai multe iterații, în funcție de mărimea unui lot. Antrenarea rețelei cu întreg setul de date, de un număr repetat de ori, este recomandată pentru a ajusta greutățile modelului, pe baza rezultatelor unei epoci anterioare. Mai pe scurt, modelul, la fel ca un om, învață din greșelile făcute. După mai multe teste, un număr de 75 de epoci a părut cel mai potrivit pentru a încadra modelul între *underfitting* și *overfitting*.

- **Salvarea modelului în diferite formate**

Având modelul complet antrenat, se salvează atât în tipul H5, pentru evaluare ulterioară, cât și în formatul TFJS, pentru folosirea lui în aplicația client.

- **Evaluarea modelului**

Această temă va fi abordată în capitolul următor, testare și evaluare.

5.2. Sistemul Client-Server

În partea prezentată, toate aspectele legate de aplicatia retelei au avut ca scop crearea rețelei neuronale, fără a avea legătură directă cu aplicația noastră propriu-zisă. Acum urmează folosirea rețelei create în sistemul format din celelalte două componente mari ale proiectului, clientul și serverul. Acestea funcționează după arhitectura clasică Client-Server prezentată în figura 4.1.

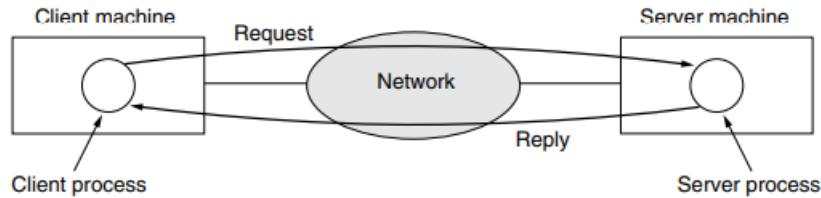


Figura 5.5: Flux comunicare Client-Server [5]

În acest sistem, s-a ales accesarea rețelei neuronale direct pe partea de client, pentru a reduce latenta generată de numărul excesiv de predicții și accesări ale acesteia prin comunicarea către server, dacă s-ar fi ales varianta de stocare și accesare a modelului pe server. Acest lucru este posibil prin conversia modelului la un model TFJS, specific pentru procese de acest tip. Astfel, un flux de bază al arhitecturii este următorul: partea de client capturează datele prin camera web, imagini din care se extrag cele 21 de repere necesare tot prin tehnologia MediaPipe. Aceste date sunt trimise pentru predicție către rețeaua neuronală, care face predicția și trimit datele printr-o metodă POST spre endpoint-ul serverului, unde are loc post-procesarea rezultatului. Datele se întorc ca răspuns al cererii HTTP, după care sunt folosite pentru a actualiza interfața clientului. O diagramă pentru acest flux este expusă în figura 5.6.

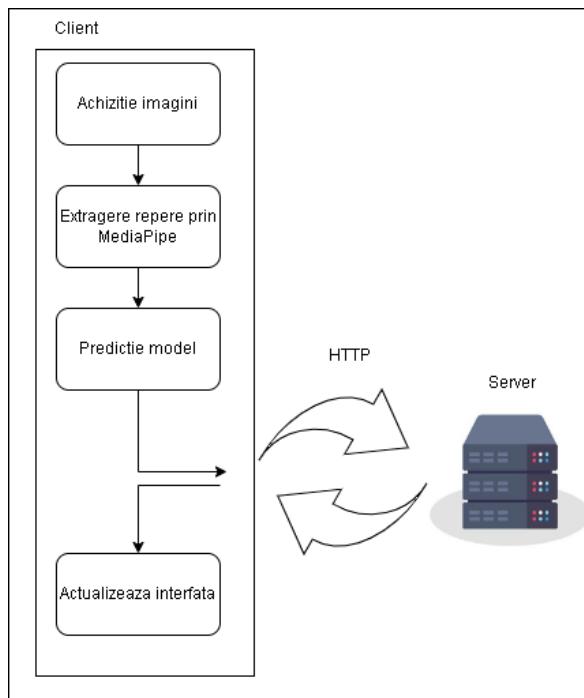


Figura 5.6: Flux recunoaștere semn

Există totuși problema momentului când trebuie să fie citit următorul semn. Cum știe sistemul că s-a trecut la alt semn? Ar exista varianta de a începe o nouă predicție după o perioadă de timp. Această opțiune ar fi perfectă dacă omul s-ar comporta ca un robot și ar face schimbarea de semn la un interval fix de secunde. Această abordare poate duce la apariția a două semne de același tip dacă persoana este prea lentă în a schimba poziția, sau un semn greșit dacă se schimbă poziția prea devreme.

O altă opțiune ar fi de a face diferența între rezultatul actual al predicției și rezultatul următor. Totul pare în regulă la această abordare până când apare problema când persoana chiar dorește să aibă două litere consecutive de același tip. De exemplu, când se dorește scrierea unui cuvânt cu doi de ”i” sau alte tipuri de cuvinte care au două litere consecutive identice. În acest caz, a face diferența de cadre nu ajută, deoarece nu se va schimba poziția mâinii.

Pentru a rezolva această problemă, s-a ales introducerea unui semn suplimentar, inventat de mine, pentru a arăta trecerea de la o literă la alta. Astfel, înainte de fiecare literă, este nevoie aratărea acestui semn distinctiv pentru a putea face recunoaștere. Mai pe scurt, se anunță sistemul că o să urmeze transmiterea unui semn. Deși această abordare scade viteza sistemului în a traduce un text, aduce simplitate în implementare și corectitudine în traducere.

Semnul este unul destul de simplu: este o simplă palmă ridicată, asemănătoare cu semnul ”stop” din alte sisteme. S-a ales acest semn pentru că este unul simplu, rapid și ușor de realizat pentru a nu pierde prea mult timp pe lucruri tehnice. O reprezentare a semnului este prezentată în figura 5.7.



Figura 5.7: Semn suplimentar pentru fluxul sistemului

Pe lângă acest semn, a mai fost nevoie de două semne suplimentare adăugate limbajului semnelor. Astfel, am inventat două semne noi, din figura 5.8, unul pentru spațiu între cuvinte, iar al doilea pentru a șterge un caracter, fie el greșala utilizatorului, fie a sistemului.



Figura 5.8: Semn suplimentar pentru spatiu si stergere

5.3. Cazuri de utilizare

Folosirea aplicației în modul scris

Actor principal: Utilizatorul sistemului

Pre-condiții:

1. Utilizatorul cunoaște regulile de funcționare ale aplicației.
2. Utilizatorul deține o cameră web.
3. Utilizatorul cunoaște limbajul semnelor.

Post-condiții:

Semnul arătat camerei să fie transpus în caseta text a interfeței.

Scenariu:

1. Utilizatorul pornește aplicația.
2. Utilizatorul arată semnul special pentru notificarea sistemului.
3. Utilizatorul așteaptă culoarea verde a semnalului sistemului.
4. Utilizatorul arată semnul dorit.
5. Utilizatorul așteaptă ca predicția să fie realizată.
6. Utilizatorul poate continua cu următorul semn.

Folosirea aplicației în mod vorbire

Actor principal: Utilizatorul sistemului

Pre-condiții:

1. Utilizatorul cunoaște regulile de funcționare ale aplicației.
2. Utilizatorul deține un sistem sonor.

Post-condiții:

Sistemul va emite în mod audio textul scris.

Scenariu:

1. Utilizatorul pornește aplicația.
2. Utilizatorul folosește limbajul semnelor pentru a scrie textul dorit.
3. Utilizatorul apasă butonul "Hear the text".
4. Utilizatorul ascultă versiunea audio a textului.

Ștergerea unui caracter nedorit

Actor principal: Utilizatorul sistemului

Pre-condiții:

1. Utilizatorul cunoaște regulile de funcționare ale aplicației.
2. Utilizatorul cunoaște semnul special pentru ștergere.
3. Utilizatorul deține o cameră web.

Post-condiții:

Sistemul va șterge ultimul caracter adăugat textului.

Scenariu:

1. Utilizatorul pornește aplicația.
2. Utilizatorul arată semnul special pentru notificarea sistemului.
3. Utilizatorul așteaptă culoarea verde a semnalului sistemului.
4. Utilizatorul arată semnul special de ștergere.
5. Utilizatorul așteaptă ca predicția să fie realizată.
6. Utilizatorul așteaptă ca textul să fie actualizat.
7. Utilizatorul poate continua cu următorul semn.

Corectarea unei secvențe sau completarea ei folosind sugestia

Actor principal: Utilizatorul sistemului

Pre-condiții:

Utilizatorul cunoaște regulile de funcționare ale aplicației.

Post-condiții:

Sistemul va actualiza textul cu sugestia oferită.

Scenariu:

1. Utilizatorul pornește aplicația.
2. Utilizatorul folosește limbajul semnelor pentru a scrie textul dorit.
3. Utilizatorul apasă pe sugestia oferită.
4. Utilizatorul așteaptă ca textul să fie actualizat.
5. Utilizatorul poate continua cu următorul semn.

Spătiera între caractere

Actor principal: Utilizatorul sistemului

Pre-condiții:

1. Utilizatorul cunoaște regulile de funcționare ale aplicației.
2. Utilizatorul cunoaște semnul special pentru spătire.
3. Utilizatorul deține o cameră web.

Post-condiții:

Sistemul va adăuga un spațiu la sfârșitul textului scris.

Scenariu:

1. Utilizatorul pornește aplicația.
2. Utilizatorul arată semnul special pentru notificarea sistemului.
3. Utilizatorul așteaptă culoarea verde a semnalului sistemului.
4. Utilizatorul arată semnul special de spătire.
5. Utilizatorul așteaptă ca predicția să fie realizată.
6. Utilizatorul așteaptă ca textul să fie actualizat.
7. Utilizatorul poate continua cu următorul semn.

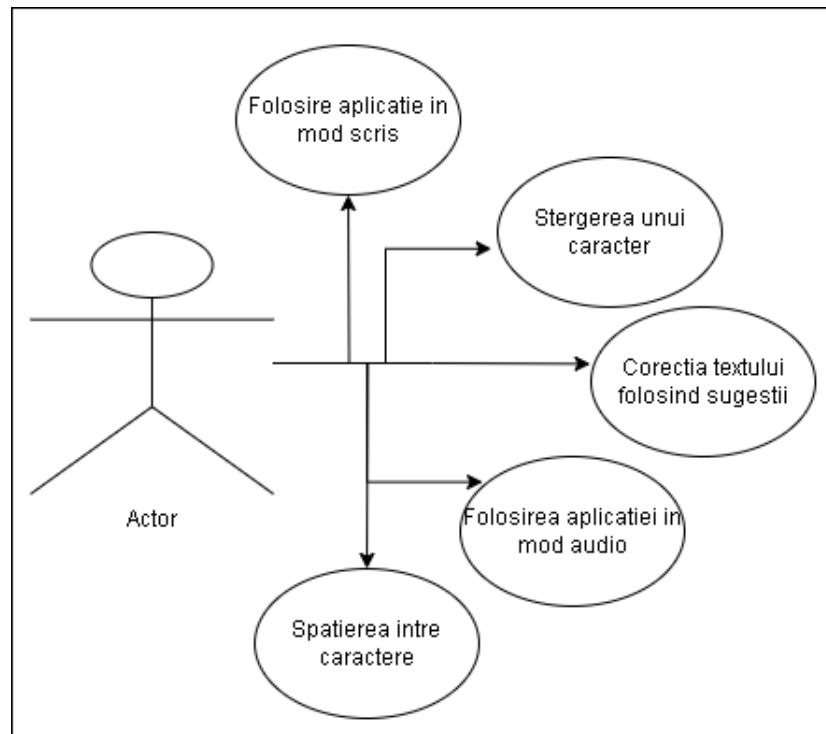


Figura 5.9: Cazuri de utilizare

5.4. Serverul

Partea de server este scrisă în Python cu ajutorul *framework*-ului Flask, care face posibilă expunerea de puncte de acces spre server, prin care clientul poate accesa funcționalitățile serverului. Astfel, serverul primește cereri HTTP și răspunde corespunzător în funcție de logica implementată. Partea de server se ocupă cu dicționarul de sugestii și corecție a textului scris în partea de client. Când clientul decide să trimită textul de pe partea sa, serverul primește o cerere HTTP de tip POST ce conține ca și *body* un JSON cu textul. Textul folosește un model de tip *transformer* pentru a genera sugestia. Detaliile despre acest model sunt prezentate ulterior.

Logica serverului pentru acest caz de utilizare este următoarea: serverul reține 2 variabile globale, *"latest_text"* și *"latest_response"*. Dacă textul primit este identic cu cel din *"latest_text"*, predicția nu se mai face și se returnează *"latest_response"*. Altfel, dacă textul este diferit, se actualizează *"latest_text"*, se face predicția și se actualizează *"latest_response"*, după care se returnează ca răspuns predicția. Această logică aduce consistență, fiind returnat mereu același rezultat pentru aceeași cerere. Însă nu acesta este motivul principal pentru care se evită predicțiile multiple. Rețeaua neuronală din partea de client face predicții fără oprire, astfel că și cererile spre server sunt într-un număr foarte ridicat. Dacă pentru același text s-ar face tot timpul noi predicții, asta ar introduce latență în comunicarea spre client. În figura 5.10 este prezentată o diagramă pentru această logică.

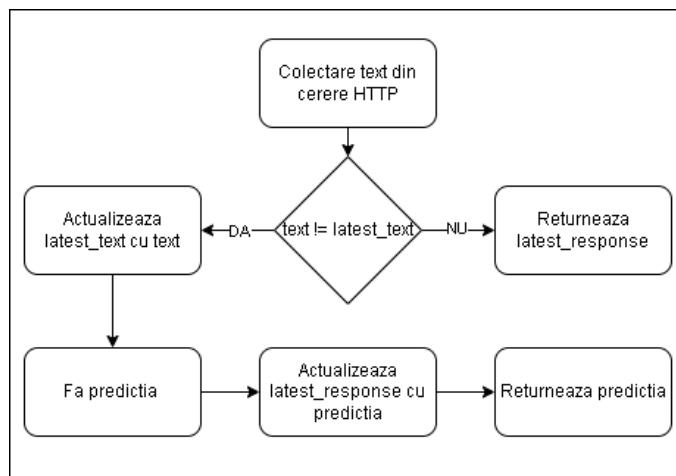


Figura 5.10: Logica generării unei noi sugestii pentru textul scris

Modelul folosit pentru acest dicționar este preluat de la oliverguhr și utilizat pentru a aduce un aport și o unealtă de ajutor în plus la aplicația de bază. Modelul este de tip *transformer* și este folosit pentru *text2text generation*. Are ca și set de date niște fraze cu greșeli care sunt legate de frazele lor corectate. Astfel, modelul învăță nu doar greșeli-corecții ca într-un dicționar simplu, ci și cum ar trebui formulată o frază în mod general, făcând sugestii pentru cuvinte următoare.

Din cauza că serverul are *deployment* făcut în Heroku, iar modelul are dimensiuni destul de mari având ca și consecință un cost ridicat pentru urcare pe platformă, s-a ales să nu se integreze direct acest model, ci folosirea prin intermediul Hugging Face Inference API, care utilizează funcționalitățile modelului într-un mod *serverless*. Astfel, logica serverului este un avantaj și din acest punct de vedere, deoarece cu scăderea cererilor pe server scad și cererile spre Hugging Face.

5.5. Aplicația Client

Partea de client este implementată în JavaScript, iar *styling*-ul este realizat în CSS. Ca framework de implementare s-a folosit React. Partea de client este împărțită în două pagini. Una dintre pagini este pagina de detectie, care deține toate funcționalitățile aplicației. Cea de-a doua pagină a aplicației are rol de prezentare a regulilor de folosire pentru o funcționare facilă. Aici sunt prezentate regulile legate de semnul special dinaintea semnelor dorite, semnele speciale pentru stergere și spațiere, precum și toate semnele corespunzătoare limbajului semnelor. Trecerea de la o pagină la alta se face din bara de navigare, prezentată în figura 5.11.



Figura 5.11: Bara de navigare

Pagina de detectare este descrisă în fișierul *Detection.js*, cu fișierul CSS de stilizare aferent. Va fi descris pas cu pas ce se întâmplă în ordine secvențială când pornește aplicația. Există două funcții de tip *useEffect()* care rulează la pornirea aplicației. Aceste *hook*-uri din React sunt instruite să ruleze fix după randarea paginii sau când o variabilă definită deja și adăugată la parametrul de dependențe își schimbă valoarea între dependențe. Aplicația are mai multe *hook*-uri de acest tip, dar momentan se va pune accent pe cele două care sunt rulate la începutul aplicației.

Primul *useEffect* se ocupă cu încărcarea în context a rețelei neuronale. Această încărcare se face cu ajutorul funcției *loadLayersModel* din biblioteca TensorFlow, care încarcă modelul de tip TFJS de la o locație anume, fie ea locală, fie ea din rețeaua de internet. Acest apel de funcție este incorporat într-un bloc *try-catch*, pentru a loga eventualele erori.

Se va încărca modelul dintr-un Google Cloud Storage Bucket, care conține rețeaua neuronală și o expune ca două fișiere: un fișier JSON de descriere a arhitecturii, și a altor detalii despre model, precum și rețeaua neuronală propriu-zisă, cu greutățile și legăturile ei, într-un fișier *.bin*, fiind un *stream* de octeți.

S-a ales stocarea într-un *bucket* de *cloud storage* și nu împreună cu aplicația din mai multe motive, atât pentru securitatea oferită de *cloud storage*, cât și pentru disponibilitatea continuă și refacerea în caz de eșec. Pe lângă asta, din cauza că aplicația web este *hostată* în GitHub Pages, ar fi mai costisitor găzduirea modelului în partea de client, Google Cloud Storage fiind fără costuri pentru dimensiunea modelului.

Cel de-al doilea *useEffect* se ocupă tot cu încărcarea unui model, dar acum este despre modelul MediaPipe pentru detectarea mâinii și a reperelor acesteia. Accesarea se face prin funcția *Hands* a bibliotecii MediaPipe și acceseză fișierul sursă din partea lor de *hosting*. După aceea, în același *useEffect*, se setează parametrii modelului, precum numărul maxim de mâini ce poate fi detectat, încrederea minimă pentru a detecta o mâna, cât și încrederea minimă de urmărire a mâinii. Se setează și o funcție de *onResults*, care se apelează de fiecare dată când algoritmul MediaPipe detectează o mâna și returnează niște rezultate. Tot în acest *useEffect* se creează un obiect de tip *Camera* din biblioteca MediaPipe, la care se setează referința la *webcam*-ul aplicației și se definește funcția

onFrame, care trimite fiecare cadru la algoritmul Hands definit anterior. Pe lângă această logică, funcția mai conține verificări pentru funcționalitatea camerei, cât și curățenie a referințelor la returnare.

Componenta React conține o funcție utilitară, *extractKeyPoints*, care pe baza rezultatelor MediaPipe, returnează un vector de 63 de valori. Aceasta primește rezultatele de la MediaPipe pe care le verifică pentru existența unei mâini. În caz că nu se regăsește nicio mână în cadru, se va returna un vector de 63 de valori de zero. În cazul în care se regăsește o mână în cadru, sunt mai multe cazuri de funcționare. Sistemul ar fi confuz dacă ar trebui să facă două predicții în paralel pentru ambele mâini; din acest motiv, predicția se face pe mână stângă tot timpul în cazul apariției ambelor mâini. În cazul în care apare o singură mână în cadru, iar aceasta este dreapta, datele sunt transpuse în oglindă pentru a simula o mână stângă. În cazul în care există o singură mână în cadru, iar aceasta este stânga, nu există logică suplimentară. Funcția va colecta cele trei coordonate din fiecare reper al mâinii și le va concatena, rezultând un vector de 63 de valori, având 21 de repere a către trei coordonate fiecare.

Această funcție de *extractKeyPoints* este folosită în funcția *onResults*, menționată anterior. Funcția *onResults* primește vectorul de coordonate prin funcția *extractKeyPoints*, pe care le folosește ulterior pentru a desena *canvas*-ul aplicației, peste imaginea sursă oferită de camera web, astfel rezultând figura 5.12.



Figura 5.12: Evidențierea reperelor mâinii

Pe lângă desenarea reperelor, funcția *onResults* adaugă rezultatele returnate de funcția *extractKeyPoints* la un vector de vectori de rezultate, care reprezintă datele din cele 30 de cadre pe care aplicația încearcă să le adune pentru a face o predicție asupra semnului prezentat. O diagramă pentru tot ce s-a exemplificat până în acest moment este prezentată în figura 5.13.

Așa cum s-a expus mai sus, logica implementată strânge datele cadru cu cadru într-un vector de vectori, dar până când se face asta? Asta se va întâmpla non-stop, dar la fiecare nouă adăugare a unui vector de date se apelează un nou *hook useEffect*, cu

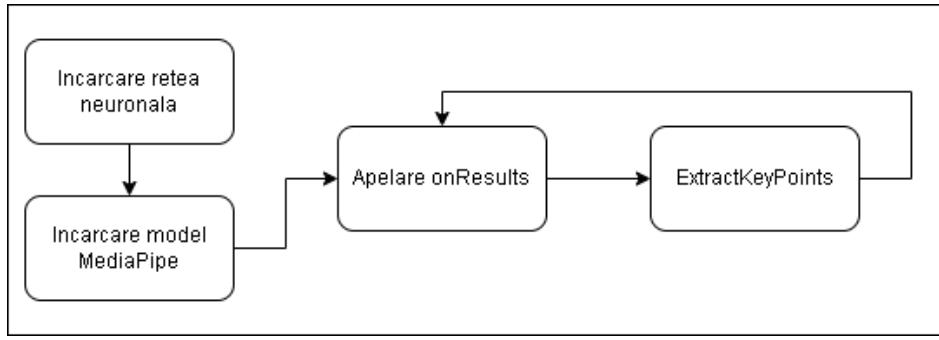


Figura 5.13: Flux al startului aplicației client

dependentă setată pe vectorul de date. Acesta verifică la fiecare pas dacă s-a ajuns la un număr de 30 de cadre sau dacă vectorul de date nu este plin de valori de 0, semn că nu a existat nicio mână în cadru pe parcursul celor 30 de cadre. Dacă cele două condiții sunt îndeplinite, se face predicția pe setul de date strâns. Dacă semnul recunoscut este semnul suplimentar, se apelează direct funcția *handleFinalPrediction*, care va fi prezentată mai jos. În cazul în care predicția nu este semnul suplimentar, se verifică dacă predicția actuală este aceeași cu predicția de la cadrul anterior. Această verificare se face cu scopul de a avea două predicții consecutive identice pentru a evita eventualele predicții din tranzitiiile între semne. De ce nu se face asta și pentru semnul suplimentar? Dacă există un semn suplimentar deja înregistrat și urmează încă unul, aplicația nu este afectată, iar dacă un alt semn este înregistrat ca ultima predicție, aplicația oricum are nevoie de un semn suplimentar la acest pas, deci doar se grăbește procesul. Revenind la fluxul de execuție, dacă predicția actuală e diferită de cea anterioară, se actualizează variabila *"lastPrediction"* și se golește prima jumătate a vectorului de date. În cazul în care este aceeași predicție, se apelează funcția *handleFinalPrediction* și se golește complet vectorul de date. De ce se tratează diferit vectorul de date? În cazul în care predicțiile nu se potrivesc, se va goli doar o parte din vector deoarece oricum urmează o altă predicție care ar trebui să fie identică cu cea actuală, deci există siguranță și în acest mod se grăbește procesul de scriere. În caz că s-a înregistrat un semn, se va goli vectorul deoarece urmează un semn complet diferit, deci va fi nevoie de toate datele.

Funcția *handleFinalPrediction*, de care s-a vorbit mai sus, se ocupă cu gestionarea predicțiilor constante care vin de la rețeaua neuronală. Are mai multe variante de verificare, pentru a respecta regulile definite ale sistemului. Primul caz, în care dacă predicția e semnul suplimentar și dimensiunea textului scris este 0, se stochează această predicție în vectorul final. Dacă dimensiunea vectorului de predicții este mai mare de 0, ultima predicție din vector e diferită de semnul suplimentar și predicția actuală e semnul suplimentar, se stochează această predicție în vectorul final. Al treilea caz este când dimensiunea vectorului este mai mare de 0, iar ultima predicție din vector este semnul suplimentar. Dacă predicția nu e semnul pentru ștergere, atunci se stochează în vector. Ultimul caz este când dimensiunea vectorului este mai mare de 0, iar ultima predicție din vector este semnul suplimentar, iar predicția este semnul special pentru ștergere. În acest caz, se șterg toate valorile din vectorul de predicții de la sfârșit și până la penultima predicție de semn suplimentar, inclusiv și predicția de semn anterioară ștergerii. În figura 5.14 este prezentată logica funcției *handleFinalPrediction*.

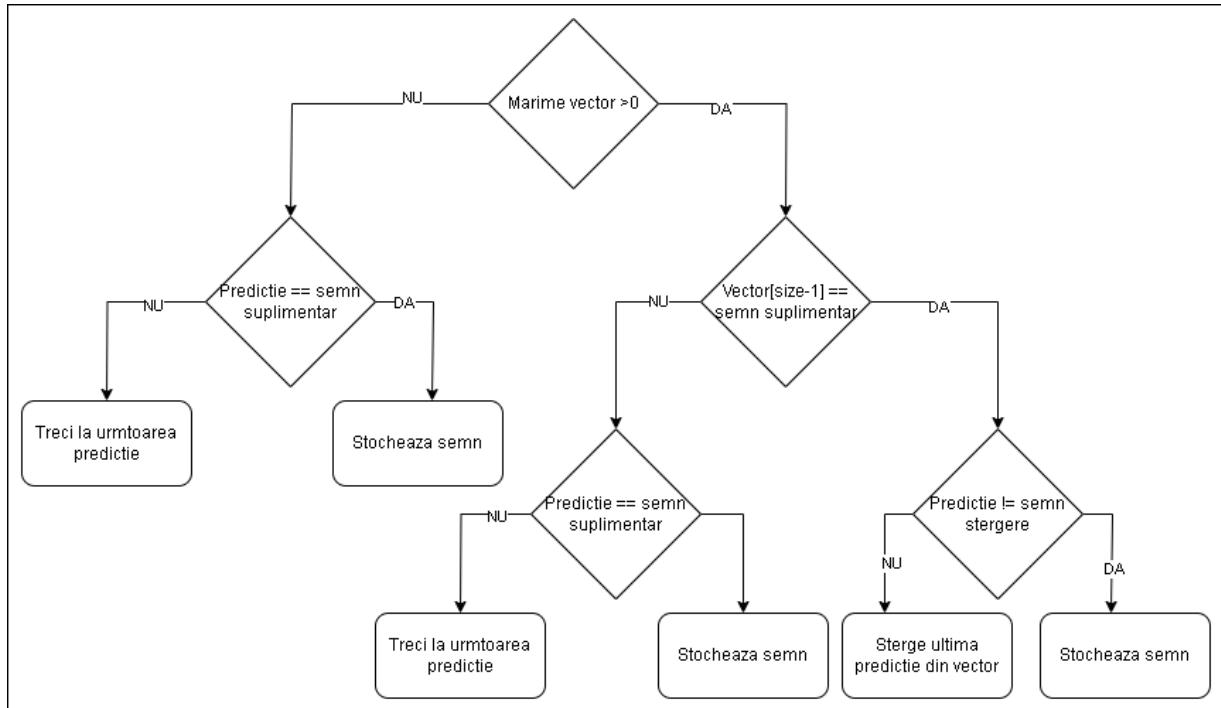


Figura 5.14: Logică gestionare predicție

După fiecare predicție înregistrată cu succes în text, se apelează un al patrulea *hook* de tip *useEffect*, acesta având ca parametru de dependență vectorul de predicții înregistrate. Acest *useEffect* se ocupă cu apelul spre server pentru cererea de sugestii sau corecții a textului. Acest apel completează arhitectura client-server menționată mai sus, post-procesând datele predicțiilor pe partea de server.

Acest *useEffect* creează o cerere HTTP de tip POST spre *endpoint*-ul expus de către aplicația găzduită pe platforma Heroku. Folosește ca *body* textul scris până acum și așteaptă o posibilă corecție sau sugestie. Nu folosește niciun tip de criptare sau autenticare deoarece datele transmise nu sunt confidențiale sau de tip personal, ci doar un text scris cu scopul de comunicare. Dacă codul primit de la server este 200, comunicarea s-a realizat cu succes, iar ca răspuns vom primi sugestia făcută de modelul *transformer* de pe partea de server.

Se face o verificare suplimentară: dacă sugestia este identică cu textul deja scris, nu se va întâmpla nimic, neavând un motiv de a schimba ceva, deoarece serverul nu are o sugestie pentru ce avem scris deja. Dacă răspunsul este diferit de textul actual, se afisează în interfață utilizatorului și acesta are posibilitatea de a actualiza textul actual cu sugestia. De menționat că predicțiile din vector sunt reprezentate ca numere ce reprezintă semnul, acesta fiind modul de predicție al rețelei neuronale. Din acest motiv, înainte de apelul spre server există o funcție care translatează vectorul din valori numerice în textul scris, ignorând valorile pentru semnele suplimentare.

În jurul acestei funcționalități a fost expusă multă logică și poate fi puțin greu de asimilat, aşa că am expus toată logica de pe partea de client în figura 5.15.

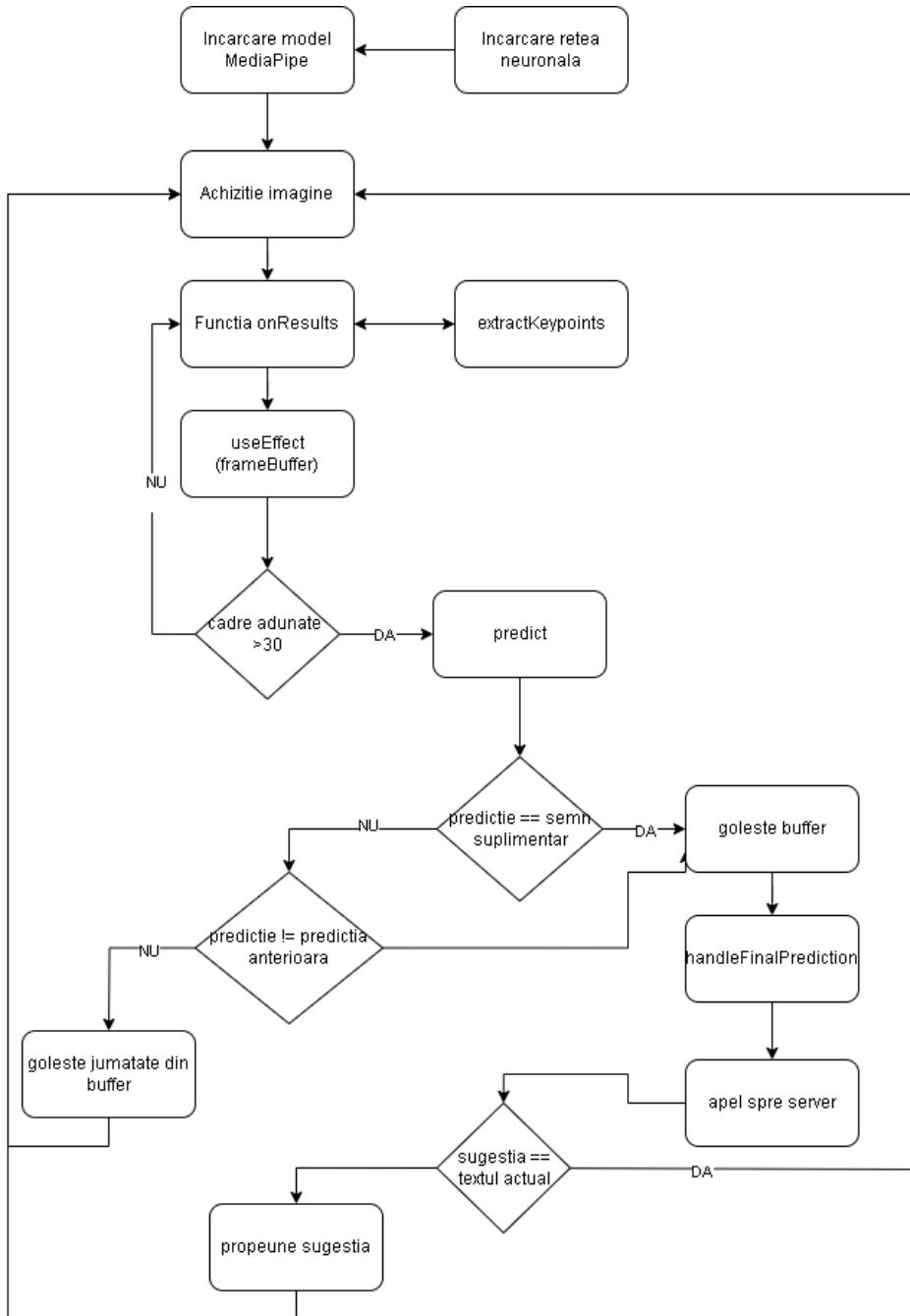


Figura 5.15: Diagramă execuție client

Ce a fost prezentat până în prezent a fost funcționalitatea de bază a aplicației, la care se adaugă două noi funcționalități. Una dintre ele este logica pentru folosirea sugestiilor venite de pe partea de server, iar cealaltă funcționalitate este reprezentată de expunerea textului scris ca și audio, pentru a transmite mesajul pentru mai multe persoane care ascultă o prezentare, cum ar fi o conferință. Aceste două metode extra sunt destul de banale, nefiind nevoie de o diagramă explicită pentru logica din spate.

Pentru cazul alegerii unei sugestii din cele expuse, se înlocuiește secvența textului scris cu sugestia aleasă. De menționat că textul scris e sub forma de numere, având atât caracterul suplimentar între fiecare literă. Pentru asta, s-a implementat o funcție *StringToArray* care face conversia. Pentru funcționalitatea de audio a textului, s-a ales

folosirea *SpeechSynthesisUtterance*, care folosește textul scris pentru a genera o secvență audio. Dacă sistemul pe care rulează aplicația dispune de un sistem audio, secvența va fi redată la apăsarea unui buton.

Fiind acoperite toate funcționalitățile, se poate prezenta și implementarea grafică a interfeței de pe partea de client. Pagina pentru detectii este prezentată în figura 5.16.

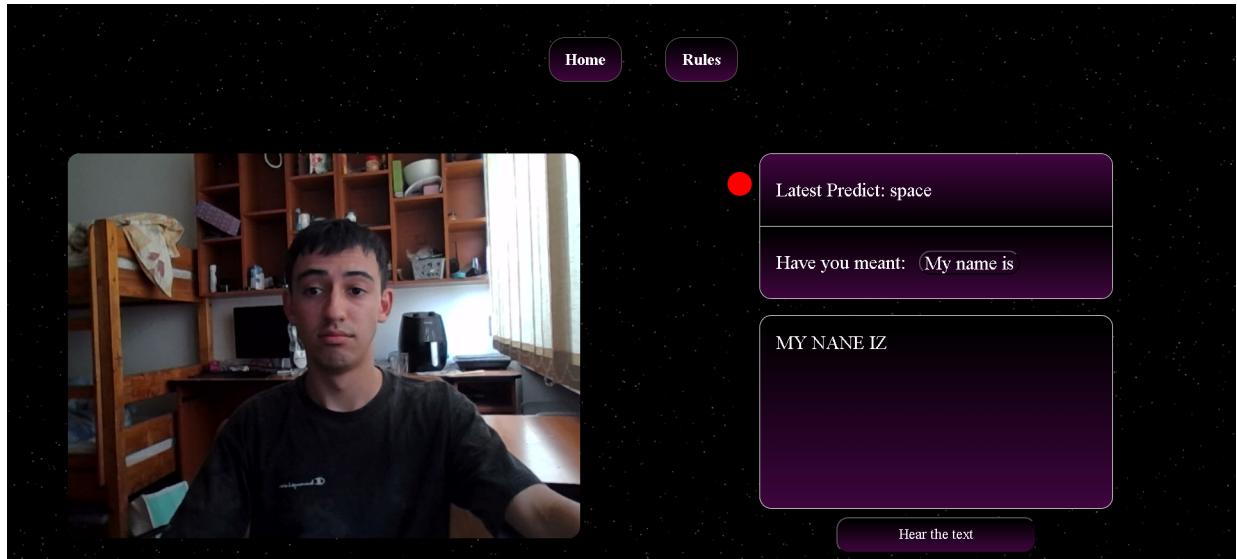


Figura 5.16: Pagină detectii

Cu ajutorul barei de navigare, se poate trece cu ușurință pe celalaltă pagină a aplicației, pagina *Rules*. Aici sunt prezentate regulile aplicației, atât pentru semnul suplimentar, cât și pentru semnele ajutătoare de stergere și spațiere. Ca și adăugare, sunt plasate și toate literele cu semnul lor specific al limbajului semnelor. Interfața este prezentată în figura 5.17.

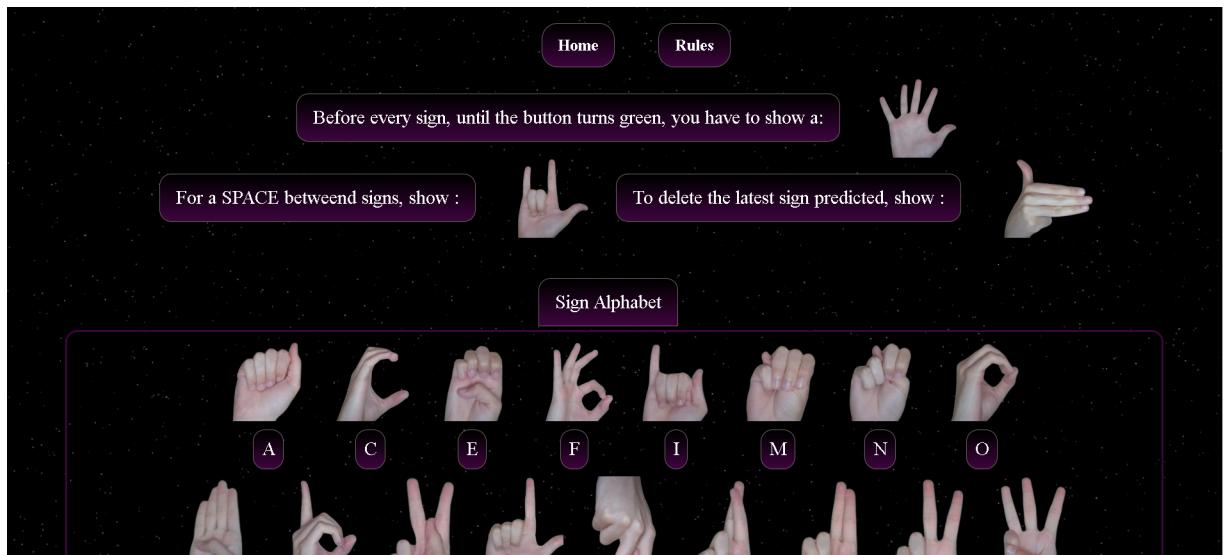


Figura 5.17: Pagină reguli

5.6. Deployment

Scopul aplicației a fost de la bun început să ofere un sprijin persoanelor în nevoie, cu dizabilități de vorbire, care încearcă să se exprime și să se integreze în societatea din ziua de azi. Astfel, acest ajutor pe care aplicația încearcă să-l ofere acestei comunități trebuie să fie unul continuu, ușor accesibil oricui și de oriunde, în orice moment când nevoia apare. Din această cauză, funcționalitățile implementate, cât și întreaga aplicație, sunt hostate și disponibile pe internetul larg, cu o singură accesare a unui link. Ca urmare, aplicația este disponibilă la https://mrradu1.github.io/SR_SignLanguageRecognition/.

Hostarea este împărțită în 3 părți: modelul este stocat în Google Cloud Storage din cauza dimensiunii, partea de server este hostată în Heroku, iar, pentru că oferea un host mai plăcut, s-a ales separarea părții de client și server și hostarea părții de client în GitHub Pages.

Toate cele trei resurse sunt publice, nefiind nevoie de autentificare și autorizare pentru acces. Nu există probleme de securitate din acest motiv, deoarece nu se lăză cu date private sau confidențiale. Un avantaj pentru acest hosting public este că aplicația poate fi considerată *open-source*, oricine având posibilitatea de a o extinde cu noi capabilități, cât și să folosească rețeaua neuronală creată în acest proiect în aplicații noi, create complet de la zero de către alți utilizatori.

Capitolul 6. Testare și validare

În acest capitol sunt prezentate rezultatele obținute în urma mai multor testări pe parcursul implementării. Testarea are un rol important în dezvoltarea oricărei aplicații, confirmând astfel o funcționare corectă a sistemului și a fiecărui subsistem al acestuia. Prin testarea treptată, făcută în paralel cu implementarea, se pot observa din timp defectiunile aplicației cât și locul exact al acestora. S-a folosit o formă de testare manuală pentru toate componentele proiectului.

6.1. Interfata grafica

Interfața grafică a fost testată și validată manual, astfel fiind verificate toate butoanele aplicației, integrarea componentei hardware reprezentată de camera web în partea de client, redirecționarea corectă între paginile interfeței, cât și ca apelurile spre server să fie fără erori pe partea de client în transmiterea cererii și primirea răspunsului.

6.2. Serverul

Pe partea de server s-au testat și validat manual capetele de acces ale aplicației backend, pentru ca acestea să primească corect cererea și să transmită răspunsul fără erori pe partea de server. Această testare s-a realizat cu ajutorul aplicației Postman și este prezentată în figura 6.1. S-a validat și intern, tot manual, apelul spre Hugging Face Inference API, care face posibilă folosirea modelului de sugestii.

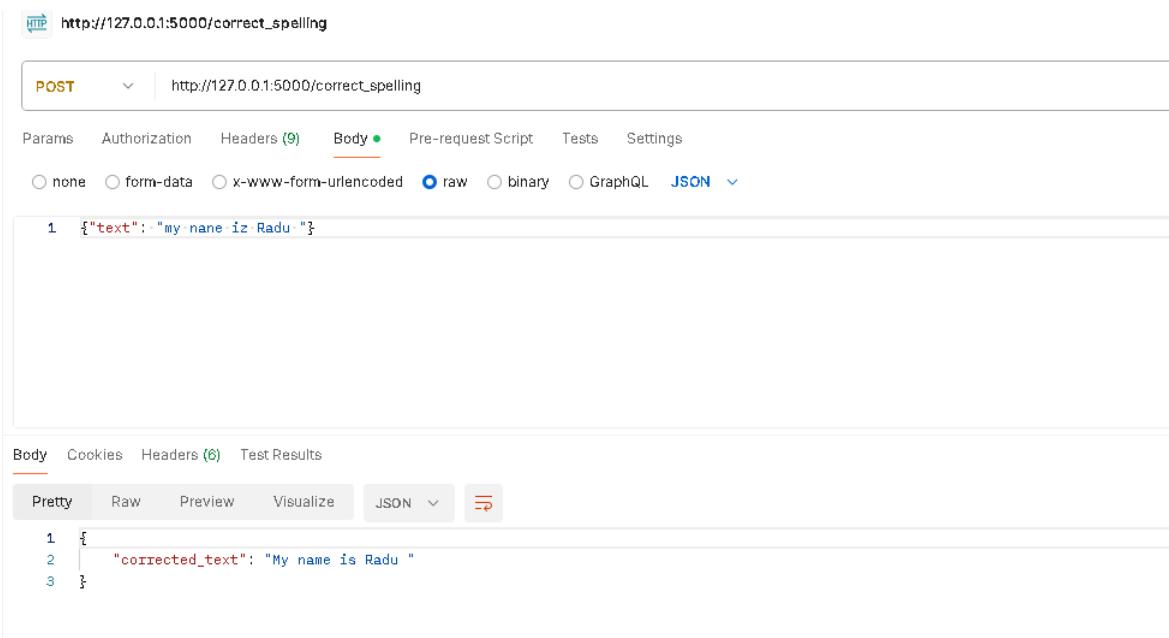


Figura 6.1: Apel spre server prin Postman

6.3. Rețeaua neuronală

Partea de rețea neuronală presupune colectarea datelor și crearea și antrenarea rețelei, aspecte testate și validate manual. Partea de colectare a datelor include folosirea bibliotecii MediaPipe, integrarea și validarea corectitudinii ei fiind testată, testarea fiind reprezentată de verificarea unei grafici asupra imaginii returnate de camera web, aşa cum este prezentat în figura 6.2, cât și returnarea unui set de 21 de vectori a către 3 coordonate fiecare, aspect testat manual.

O altă parte validată manual a fost verificarea creării cu succes a celor 29×100 de directoare și salvarea a către 45 de directoare în fiecare folder, reprezentând datele adunate.



Figura 6.2: Testare MediaPipe

Partea de rețea neuronală a avut o primă variantă testată în care rețeaua conoluțională creată a fost antrenată pe setul de date colectat, cu un număr de filtre conoluționale de 64, antrenată cu o dimensiune a lotului de 73 de-a lungul a 200 de epoci, astfel rețeaua ajungând la o precizie de 99,8%. O precizie atât de ridicată duce la suprainvățare, o stare în care rețeaua neuronală este foarte bazată pe datele pe care a fost antrenată, fiind incapabilă să mai generalizeze pe date noi, pe care nu le-a primit până acum. Graficele pentru evoluția acurateții și pierderii sunt afișate în figurile 6.5 și 6.6.

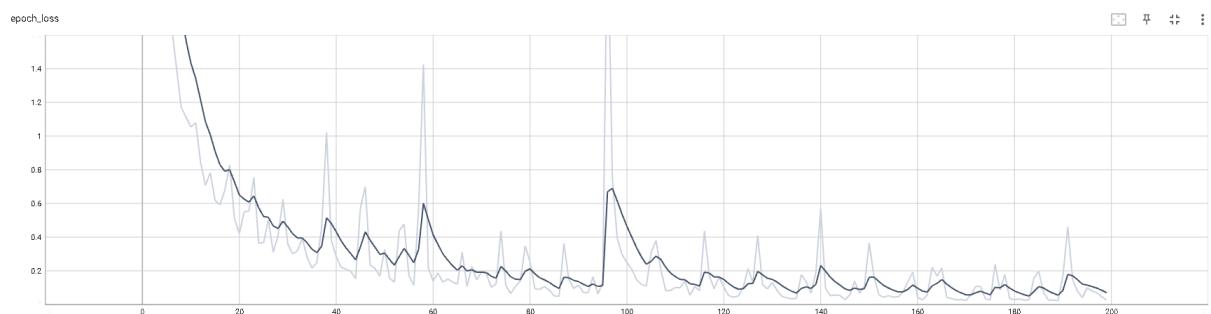


Figura 6.3: Pierdere rețea antrenată în 200 epoci

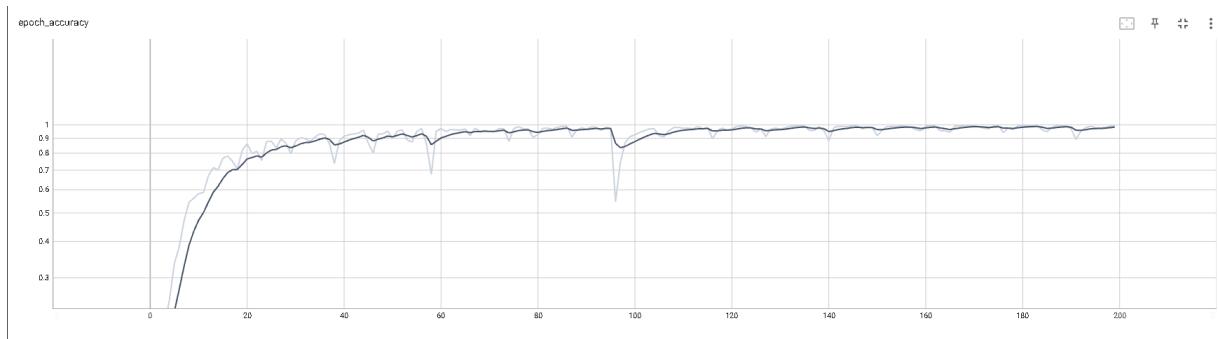


Figura 6.4: Acuratețe rețea antrenată în 200 epoci

Pentru a rezolva această problemă, s-a decis reducerea numărului de filtre al primului strat conoluțional din rețea, de la 64 la 32, iar antrenarea să se facă de-a lungul a 75 de epoci, punct în care s-a observat, în testarea cu 200 de epoci, că este momentul în care acuratețea crește la un nivel optim pentru a putea fi folosită în aplicație, fiind capabilă de o generalizare mai bună a datelor variate pe care le va primi în aplicație. Ca rezultat, s-a ajuns la o acuratețe de 93%. Graficele pentru acuratețe și pierdere în noua rețea sunt prezentate în figură.

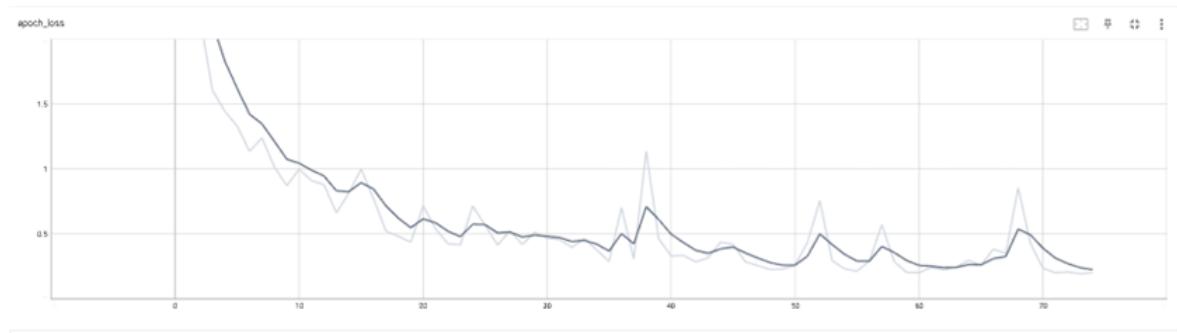


Figura 6.5: Pierdere rețea antrenată în 75 epoci

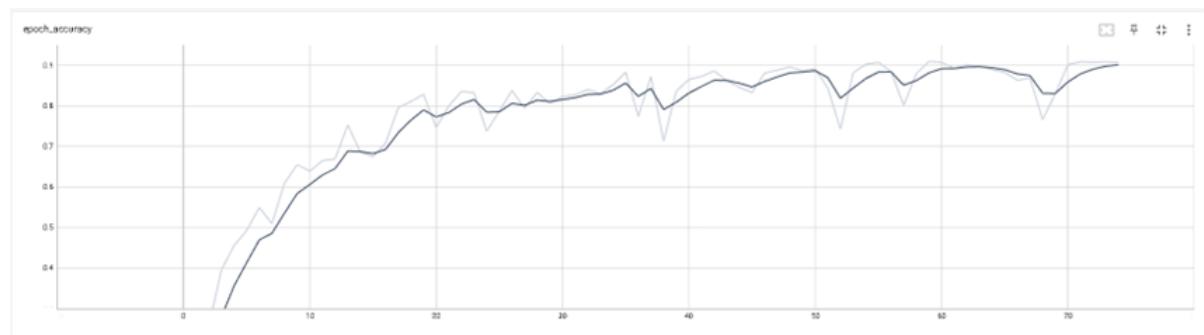


Figura 6.6: Acuratețe rețea antrenată în 75 epoci

Alte date pentru a evalua rețeaua neuronală construită au fost raportul de clasificare și matricea de confuzie. O matrice de confuzie este o matrice care rezumă performanța unui model de clasificare prin reprezentarea corectitudinii pentru fiecare clasă. Astfel, este împărțită în 4 celule: număr de exemple corect clasificate ca pozitive, număr de exemple corect clasificate ca negative, număr de exemple pozitive clasificate ca negative și număr de exemple negative clasificate ca pozitive. Matricele pentru fiecare clasă sunt reprezentate în figura 6.7.



Figura 6.7: Confusion matrix

Un raport de clasificare este un raport detaliat care oferă o imagine de ansamblu asupra performanței modelului de clasificare. Acesta este prezentat în figura 6.8 și include mai multe metri esențiale pentru evaluarea modelului:

- **Precizie** - Proporția de exemple corect prezise ca pozitive din totalul exemplelor prezise ca pozitive.

$$\text{Precizie} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

- **Recall** - Proporția de exemple corect prezise ca pozitive din totalul exemplelor pozitive reale.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

- **F1 Score** - Media armonică între precision și recall, oferind un echilibru între acestea două.

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

- **Support** - Numărul de exemple reale din fiecare clasă.

Classification Report

	precision	recall	f1-score	support
0	1.0	1.0	1.0	20.0
1	1.0	1.0	1.0	20.0
2	1.0	1.0	1.0	20.0
3	1.0	1.0	1.0	20.0
4	1.0	1.0	1.0	20.0
5	1.0	1.0	1.0	20.0
6	0.6666666666666666	0.7	0.6829268292682927	20.0
7	0.6842105263157895	0.65	0.6666666666666666	20.0
8	1.0	1.0	1.0	20.0
9	1.0	1.0	1.0	20.0
10	1.0	1.0	1.0	20.0
11	1.0	1.0	1.0	20.0
12	1.0	0.9	0.9473684210526315	20.0
13	0.9	0.9	0.9	20.0
14	1.0	1.0	1.0	20.0
15	1.0	1.0	1.0	20.0
16	1.0	1.0	1.0	20.0
17	0.5714285714285714	1.0	0.7272727272727273	20.0
18	1.0	1.0	1.0	20.0
19	0.9090909090909091	1.0	0.9523809523809523	20.0
20	0.8421052631578947	0.8	0.8205128205128205	20.0
21	0.5	0.2	0.2857142857142857	20.0
22	1.0	0.9	0.9473684210526315	20.0
23	1.0	1.0	1.0	20.0
24	1.0	1.0	1.0	20.0
25	1.0	1.0	1.0	20.0
26	1.0	1.0	1.0	20.0
27	1.0	1.0	1.0	20.0
28	1.0	1.0	1.0	20.0
accuracy	0.9327586206896552	0.9327586206896552	0.9327586206896552	0.9327586206896552
macro avg	0.9335690322986149	0.9327586206896552	0.9286279697903796	580.0
weighted avg	0.9335690322986148	0.9327586206896552	0.9286279697903796	580.0

Figura 6.8: Raport de clasificare

Capitolul 7. Manual de instalare și utilizare

În această secțiune vor fi prezentate metodele prin care se poate instala sau accesa aplicația, necesitățile *hardware* și *software* pentru o funcționare optimă, cât și pașii și regulile de utilizare pentru a beneficia de funcționalitățile aplicației în totalitate.

7.1. Manual de instalare

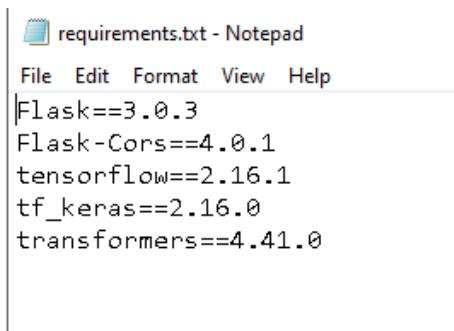
Ca necesitate principală este, evident, nevoie de o cameră web pentru a putea prelua cadrele pentru detectarea mâinilor și recunoașterea semnelor. Această necesitate se aplică în ambele cazuri de instalare sau accesare a aplicației.

Unul dintre cazuri presupune accesarea directă, fără nevoie unei instalări sau configurări a aplicației, aplicația fiind hostată pe mai multe platforme, pentru a oferi accesibilitate, suport și continuitate aplicației. Este nevoie doar de o conexiune stabilă la internet și orice tip de browser pentru accesarea aplicației. Aceasta se poate face prin accesarea următorului link: https://mrradu1.github.io/SR_SignLanguageRecognition/.

O altă abordare, în cazul în care nu există o conexiune la internet sau este doar rularea în modul *debug* sau chiar modificarea unor parametri sau ajustarea unor funcționalități, există și opțiunea de instalare locală a aplicației. Aceasta necesită instalarea inițială a Python și NodeJS, pentru rularea de comenzi *python*, *pip*, *node* și *npm* pentru instalarea altor biblioteci ulterioare. Eu am folosit Python 3.10.11 pentru partea de rețea și Python 3.12.3 pentru partea de server, din motive de compatibilitate și suport pentru bibliotecile folosite în cele două implementări.

Pentru accesarea aplicație care se ocupă cu colectarea datelor, crearea și antrenarea rețelei neuronale, este necesară instalarea Jupyter Notebook. Pentru acest lucru se rulează comanda *pip install jupyter*. Restul bibliotecilor necesare instalării sunt incluse într-o linie de cod de la începutul aplicației.

Pentru aplicația de server din *backend* implementată în Python cu ajutorul *framework-ului* Flask, este necesară instalarea mai multor biblioteci, acestea fiind adăugate într-o listă de necesități, prezentată în figura 7.1.



```
requirements.txt - Notepad
File Edit Format View Help
Flask==3.0.3
Flask-Cors==4.0.1
tensorflow==2.16.1
tf_keras==2.16.0
transformers==4.41.0
```

Figura 7.1: Listă biblioteci necesare

Pentru instalarea tuturor dependențelor se va rula următoarea comandă în linia de comandă deschisă în directorul cu fișierele sursă:

- `pip install -r requirements.txt`

Pentru pornirea aplicației se poate folosi un IDE sau următoarea comandă în linia de comandă:

- `python main.py`

Pentru partea de client, este necesară instalarea următoarelor pachete:

- react
- react-dom
- react-webcam
- react-router-dom
- @mediapipe/camera_utils
- @mediapipe/hands
- @tensorflow/tfjs

Instalarea tuturor acestor dependențe și apoi pornirea aplicației de client sunt realizate prin rularea comenziilor:

- `npm install`
- `npm start`

Pentru ca sistemul să funcționeze, este necesară pornirea atât a părții de server, cât și a părții de client.

7.2. Manual de utilizare

Interfața grafică destinată utilizatorului are ca audiенă persoanele cu dizabilități, astfel realizarea unei interfețe cât mai simple și sugestive a fost unul dintre scopurile principale înaintea cât și în timpul implementării. Astfel, aplicația are un număr redus de butoane sau pași necesari pentru a atinge rezultatul dorit în urma funcționalităților oferite. La lansarea aplicației, pagina deschisă va fi chiar pagina unde se fac detectiile și clasificările semnelor, cât și funcționalitățile adiționale. Aceasta este prezentată în figura 7.2.

Este recomandată inițial trecerea spre pagina de Reguli ale aplicației, pentru o cunoaștere completă a pașilor de urmat pentru aplicație. Acestea vor fi prezentate și aici, dar sunt afișate și în acea pagină a aplicației, pentru persoanele care vor accesa direct aplicația. Trecerea spre această pagină se face prin apăsarea butonului intitulat *Rules* din bara de navigare. Această pagină este prezentată în figura 7.3.

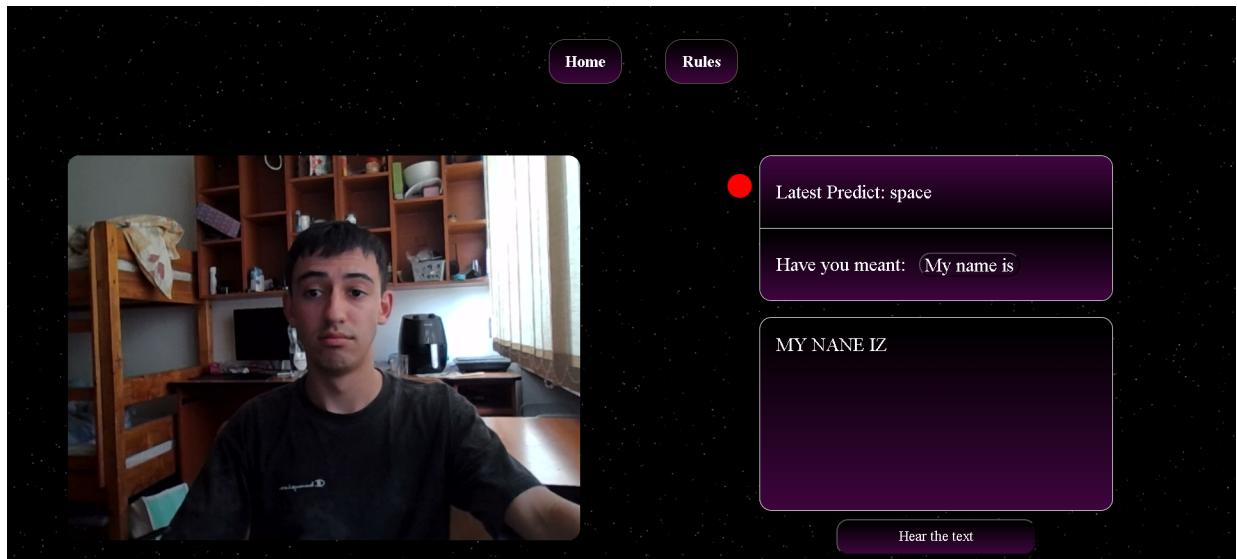


Figura 7.2: Pagina detectii

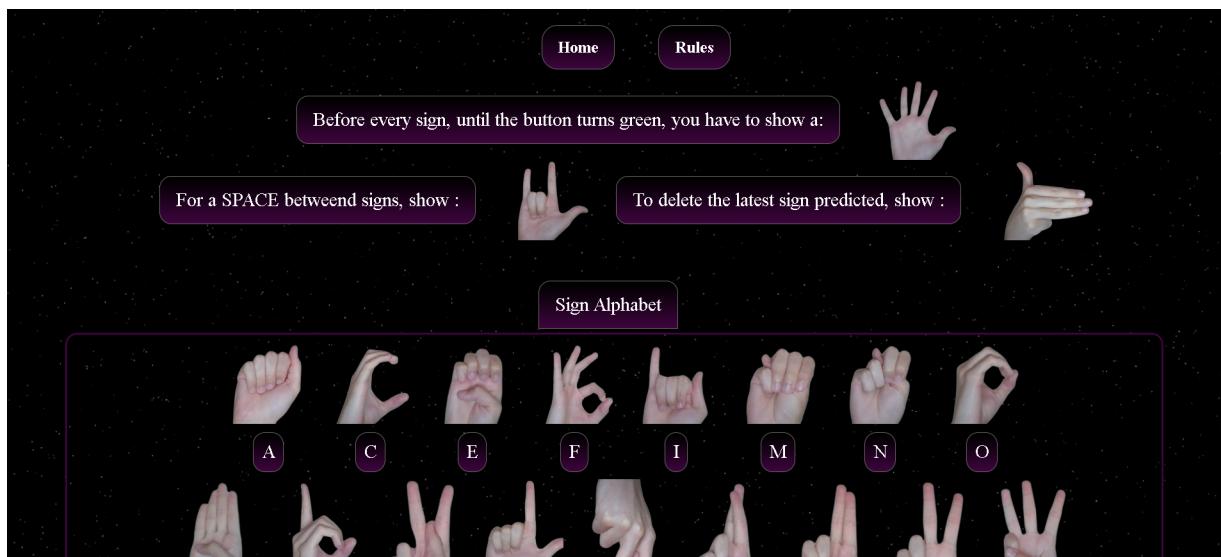


Figura 7.3: Pagina reguli

Revenirea la pagina de detecții se face tot prin bara de navigare din partea de sus a aplicației. Funcționalitățile se produc toate în chenarul prezentat în figura 7.4. Există acel indicator de culoare roșie care indică faptul că e nevoie de semnul suplimentar prezentat în figura 7.5. Dacă semnul s-a înregistrat cu succes, acel indicator va trece la culoarea verde, care indică că se poate arăta semnul dorit din limbajul semnelor. După înregistrarea cu succes a semnului dorit, acesta va apărea în partea de text a chenarului, iar indicatorul va trece din nou pe culoarea roșie. Procesul se repetă pentru fiecare literă nouă dorită.

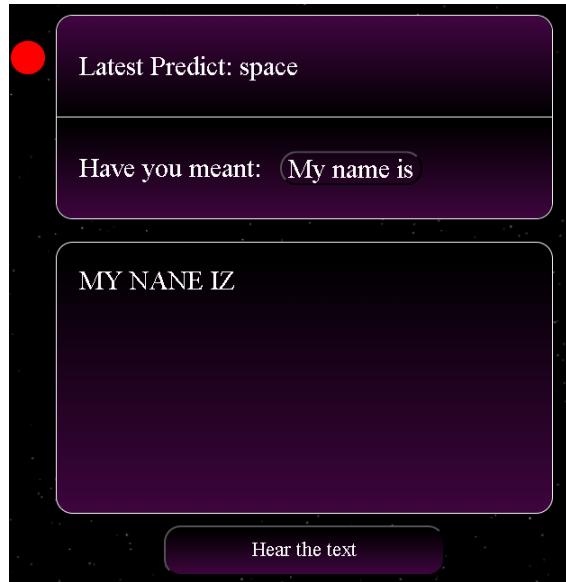


Figura 7.4: Chenar pentru funcționalități



Figura 7.5: Semn suplimentar

Pentru spațiere sau ștergere, procesul este la fel. Când indicatorul este pe culoarea verde și este momentul introducerii unui semn, se folosesc semnele adăugate suplimentar din figura 7.6.

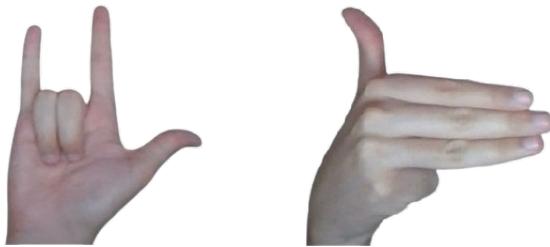


Figura 7.6: Semn suplimentar pentru spațiere și ștergere

Funcționalitatea de audio este realizată prin apăsarea unui buton din partea de jos a chenarului intitulat *Hear the text*. Dacă sistemul dispune de un sistem audio, se va reda în mod audio secvența scrisă până în momentul apărării butonului.

Dacă partea de server consideră că o corecție este necesară sau presupune o continuare a textului, funcționalitatea este accesată din partea de sus a chenarului, afișată după textul *Have you meant:*, secțiune prezentată în figura 7.7, iar utilizarea acesteia

se realizează prin apăsarea pe sugestie/corecție, dacă este dorită această acțiune. Acest lucru duce la schimbarea textului scris cu sugestia accesată.

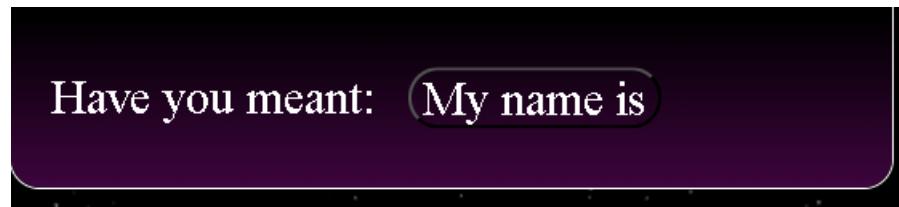


Figura 7.7: Chenar pentru sugestie

Capitolul 8. Concluzii

8.1. Concluzii si contributii personale

Ca și concluzie a proiectului propus și implementat, s-a realizat cu succes un sistem SLR (*Sign Language Recognition System*). Scopul proiectului a fost atât combinarea cât și înglobarea a cât mai multe domenii științifice, a cât mai multe biblioteci și medii de dezvoltare populare și de actualitate la momentul realizării proiectului, cât și utilizarea unui număr semnificativ de cunoștințe acumulate pe parcursul anilor de studiu în cadrul universității, la nivelul licenței. Un alt scop a fost unul social, acela de a oferi un suport pentru persoanele cu dizabilități, acestea fiind zilnic excluse și discriminate din cauza neputinței lor, neputință care s-a dorit a fi acoperită de funcționalitatea aplicației mele.

Ca și contribuție personală, consider că am contribuit prin următoarele:

1. Colectarea unui set de date consistent și variat.
2. Crearea unei structuri proprii a unei rețele neuronale care să fie cât mai precisă pentru problema noastră.
3. Alegerea proprie a unor parametri exacti de antrenare pentru aducerea rețelei la un nivel de acuratețe ridicat, evitând pe cât posibil suprainvățarea.
4. Crearea unei logici complexe în jurul acestei rețele pentru implementarea funcționalităților finale.
5. Îmbinarea cu succes a mai multor tehnologii și platforme complexe.
6. Crearea unei interfețe grafice simple și sugestive, cu un aspect plăcut.

O altă contribuție personală, din punctul meu de vedere, este una de nivel social, prin aducerea în discuție a acestei teme, punând accent pe nevoia de ajutor a acestei categorii de oameni cu scopul de a atrage atenția și implicarea a cât mai multor persoane.

8.2. Analiza critica si dezvoltari ulterioare

Ca rezultate, sistemul reușește cu succes recunoașterea tuturor semnelor limbajului, atât cele statice, cât și cele dinamice, cu o acuratețe de 93%.

Totuși, sistemul are și el unele lipsuri sau dezavantaje. Setul de date pe care a fost antrenată rețeaua neuronală a fost adunat doar prin imagini care folosesc mâna mea, lucru care ar putea duce la un eșec în a recunoaște mâini de forme sau dimensiuni diferite, datele fiind diferite decât cele cu care a fost antrenată rețeaua.

Un alt aspect negativ poate fi nevoia de o iluminare bună sau de o calitate ridicată a imaginii, pentru că în cazul în care imaginea de intrare nu este foarte clară, algoritmul MediaPipe variază pe cadrele primite, astfel că datele adunate și trimise spre rețea pentru clasificare sunt incorecte, ducând la o clasificare greșită. O dezvoltare ulterioară poate fi înlocuirea MediaPipe cu un algoritm mai bun, invariant la lumină sau claritate.

Una dintre cele mai mari probleme ale sistemului este însă cauzată de viteza puțin greoaie de procesare, prin introducerea semnului suplimentar după fiecare literă. O

dezvoltare ulterioară care ar ajuta mult aplicația ar fi implementarea unei logici care să funcționeze perfect pe cazurile de gestionare a imaginilor primite, împărțirea lor în semnul anterior, semnul curent și semnul viitor, astfel fiind eliminat jumătate din timpul de procesare a unui text introdus de utilizator.

O altă posibilă dezvoltare ulterioară este adăugarea unei procesări *multi-threading* a aplicației, fiind separate firele de execuție a procesării cadrelor și a predicției, acestea având loc în același timp, cadrele pentru următoarea predicție fiind în paralel cu predicția actuală.

Ca ultimă dezvoltare ulterioară, aş propune extinderea sistemului pentru a recunoaște și expresii sau cuvinte întregi, nu doar litere ale alfabetului, dezvoltare care ar duce la un nivel mult mai ridicat de ajutor pentru aceste persoane prin exprimarea foarte rapidă pe care o pot produce prin această dezvoltare.

Bibliografie

- [1] F. Chen, C. Fu, and C. Huang, “Hand gesture recognition using a real-time tracking method and hidden markov models,” *Image Vis. Comput.*, 2003.
- [2] A. Riad, “Hand gesture recognition system based on a geometric model and rule based classifier,” *British Journal of Applied Science and Technology*, 2014.
- [3] M. I. Sadek, M. N. Mikhael, and H. A. Mansour, “A new approach for designing a smart glove for arabic sign language recognition system based on the statistical analysis of the sign language,” in *2017 34th National Radio Science Conference (NRSC)*. IEEE, 2017.
- [4] A. S. Tanenbaum and M. van Steen, *Distributed systems - principles and paradigms, 2nd Edition*. Pearson Education, 2007.
- [5] A. S. Tanenbaum and D. Wetherall, *Computer networks, 5th Edition*. Pearson, 2011.
- [6] “HTTP Request Methods,” https://www.w3schools.com/tags/ref_httpmethods.asp [Accessed 2024.04.11]. [Online]. Available: https://www.w3schools.com/tags/ref_httpmethods.asp.
- [7] “JSON Official Website,” json.org/ [Accessed 2024.04.11]. [Online]. Available: json.org/.
- [8] A. Banks and E. Porcello, *Learning React: Modern Patterns for Developing React Apps*. O'Reilly Media, 2017.
- [9] M. Grinberg, *Flask Web Development: Developing Web Applications with Python (2nd Edition)*. O'Reilly Media, 2018.
- [10] J. Geewax, *Google Cloud Platform in Action*. Manning Publications, 2018.
- [11] F. Zhang, V. Bazarevsky, A. Vakunov, A. Tkachenka, G. Sung, C. Chang, and M. Grundmann, “Mediapipe hands: On-device real-time hand tracking,” *CoRR*, vol. abs/2006.10214, 2020. [Online]. Available: <https://arxiv.org/abs/2006.10214>
- [12] N. Middleton and R. Schneeman, *Heroku: Up and Running: Effortless Application Deployment and Scaling*. O'Reilly Media, 2013.
- [13] A. Pipinellis, *Github Essentials: Unleash the Power of GitHub to Collaborate, Host, and Deploy Your Projects*. Packt Publishing, 2016.
- [14] A. Gulli, A. Kapoor, and S. Pal, *Packt Publishing*. O'Reilly Media, 2019.
- [15] “Workflow and Data Preparation in Tensorflow,” <https://www.theobjects.com/dragonfly/dfhelp/2020-1/Content/Artificial>

- [16] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition.* O'Reilly Media, 2019.
- [17] “Convolutional Neural Networks for Dummies,” <https://medium.com/@prathammodi001/convolutional-neural-networks-for-dummies-a-step-by-step-cnn-tutorial-e68f464d608f> [Accessed 2024.05.20]. [Online]. Available: <https://medium.com/@prathammodi001/convolutional-neural-networks-for-dummies-a-step-by-step-cnn-tutorial-e68f464d608f>.
- [18] “The Math behind Artificial Neural Networks,” <https://towardsdatascience.com/the-heart-of-artificial-neural-networks-26627e8c03ba> [Accessed 2024.06.18]. [Online]. Available: <https://towardsdatascience.com/the-heart-of-artificial-neural-networks-26627e8c03ba>
- [19] “Understanding LSTM Networks,” <https://colah.github.io/posts/2015-08-Understanding-LSTMs/> [Accessed 2024.06.10]. [Online]. Available: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>
- [20] “An Intuitive Explanation of LSTM,” <https://medium.com/@ottaviocalzone/an-intuitive-explanation-of-lstm-a035eb6ab42c> [Accessed 2024.06.11]. [Online]. Available: <https://medium.com/@ottaviocalzone/an-intuitive-explanation-of-lstm-a035eb6ab42c>
- [21] “A Clear Explanation of Transformer Neural Networks,” https://medium.com/@ebinbabuthomas_21082/decoding-the-enigma-a-deep-dive-into-transformer-model-architecture-749b49883628 [Accessed 2024.04.11]. [Online]. Available: https://medium.com/@ebinbabuthomas_21082/decoding-the-enigma-a-deep-dive-into-transformer-model-architecture-749b49883628
- [22] “MediaPipe Hands,” <https://mediapipe.readthedocs.io/en/latest/solutions/hands.html> [Accessed 2024.03.18]. [Online]. Available: <https://mediapipe.readthedocs.io/en/latest/solutions/hands.html>