



Universitatea Tehnică „Gheorghe Asachi” din Iași
Facultatea de Automatică și Calculatoare
Departamentul de Automatică și Informatică Aplicată

Lucrare de Licență

Dezvoltarea unui sistem software pentru jocul de șah

Absolvent
Pavel Cosmin Constantin

Îndrumător
Prof. dr. ing. Doru-Adrian Pănescu

Iași, 2024

DECLARAȚIE DE ASUMARE A AUTENTICITĂȚII PROIECTULUI DE DIPLOMĂ

Subsemnatul **PAVEL COSMIN CONSTANTIN**, legitimat cu **CI** seria **MZ** nr. **790603** ,
CNP **5010509225900**, autorul lucrării „**DEZVOLTAREA UNUI SISTEM
SOFTWARE PENTRU JOCUL DE ȘAH**” elaborată în vederea susținerii examenului
de finalizare a studiilor de licență, programul de studii **AIA** organizat de către
Facultatea de Automatică și Calculatoare din cadrul Universității Tehnice „Gheorghe
Asachi” din Iași,
sesiunea **IULIE** a anului universitar **2023-2024**, luând în considerare conținutul Art. 34
din Codul de etică universitară al Universității Tehnice „Gheorghe Asachi” din Iași
(Manualul Procedurilor, UTI.POM.02 - Funcționarea Comisiei de etică universitară),
declar pe proprie răspundere, că această lucrare este rezultatul propriei activități
intelectuale, nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu
respectarea legislației române (legea 8/1996) și a convențiilor internaționale privind
drepturile de autor.

Data

01.07.2024

Semnătura



Cuprins

Lista de figuri	v
1 Introducere	1
2 Obiectivele lucrării și contribuțiile aduse	3
3 Fundamente teoretice	5
3.1 Motorul de inferențe pentru jocul de șah	5
3.1.1 Reprezentarea tablei de șah folosind notația Forsyth-Edwards (FEN)	5
3.1.2 Algoritmul Minimax	6
3.1.3 Optimizarea algoritmului Minimax cu Alpha-Beta Pruning	8
3.2 Sisteme de Vedere Artificială	9
3.2.1 Imagine digitală	9
3.2.2 Preprocesarea Imaginilor	10
3.2.3 Utilizarea rețelelor neuronale pentru detectia de obiecte din imagini	10
3.2.4 Modelul Yolov8	11
3.2.5 Antrenarea personalizată a modelului Yolov8	12
4 Resurse folosite în cadrul proiectului	14
4.1 Limbajul de programare Python	14
4.2 Biblioteci ale limbajului Python folosite in cadrul proiectului	14
4.2.1 PyTorch	14
4.2.2 OpenCV	15
4.2.3 Tkinter si PIL	15
4.2.4 Chess	16
4.3 Camo	16
4.4 Roboflow	16
5 Implementarea motorului de inferențe pentru jocul de șah	17
5.1 Structura generala a componentelor	17
5.1.1 Clasa GameState	17
5.1.2 Clasa ChessGame	18
5.1.3 Clasa Net	19
5.1.4 Scriptul de antrenare al rețelei neuronale de evaluare a unei poziții de șah Metode ale scriptului	19
5.2 Implementarea algoritmului Minimax	20
5.3 Implementarea functie de generare a mutarilor legale	21
5.4 Implementarea funcției de evaluare bazată pe piesele prezente pe tablă	22
5.5 Implementarea optimizării Alpha-Beta	23
5.6 Implementarea unui model neuronal pentru evaluarea unei poziții de șah	25
5.6.1 Arhitectura modelului	25
5.6.2 Baza de date folosită în cadrul proiectului	26
5.6.3 Preprocesarea FEN-urilor	26
5.6.4 Prelucrarea etichetelor	27
5.6.5 Antrenarea modelului	28
5.6.6 Evaluarea modelului	30

5.6.7	Procesul de antrenare și evaluare al rețelei neuronale	31
5.7	Funcția de evaluare a unei poziții folosind modelul CNN	31
5.8	Rezultate ale motorului de inferență	32
5.8.1	Compararea timpilor de execuție ai algoritmilor Alpha-Beta și Minimax .	32
5.8.2	Rezultatele de antrenament ale modelului CNN	32
6	Implementarea sistemului de vedere artificială	35
6.1	Detectarea tablei de șah	35
6.1.1	Datele de antrenament	36
6.2	Transformarea de perspectivă și segmentarea tablei	38
6.3	Detectarea pieselor și formarea tablei în format digital	41
6.3.1	Extragerea și clasificarea pieselor de pe tabla de șah	41
6.3.2	Formarea și afișarea tablei digitale	42
Funcția <code>construct_fen</code>	42	
6.3.3	Determinarea secvenței de mutare	44
6.4	Rezultatele antrenării pentru modelele utilizate în sistemul de vedere artificială .	44
6.4.1	Explicația metricilor returnate de Yolov8 la finalul antrenamentului .	44
6.4.2	Interpretarea Rezultatelor Modelului YOLOv8 pentru Detecția Colțurilor	46
6.4.3	Interpretarea rezultatelor modelului YOLOv8 pentru detecția secvenței de mutare	47
6.4.4	Rezultatele modelului de clasificare a pieselor	47
7	Implementarea logicii de joc	49
7.1	Interfața grafică	49
7.1.1	Bucla de joc	50
7.1.2	Mesajele afișate în consolă	50
8	Concluzii și posibilități de dezvoltare	52
8.1	Concluzii legate de motorul de inferență	52
8.2	Concluzii legate de sistemul de vedere artificială	52
8.3	Posibilități de dezvoltare	52
Bibliografie		53
Anexa - Codul realizat în cadrul proiectului		56

Lista de figuri

1.1	Boris Handroid, prima automatizare a jocului de șah [6]	2
1.2	Robot ABB inclus într-un sistem de șah [8]	2
3.1	Poziția de start în jocul de șah [10]	6
3.2	Câmpul de plasare a pieselor în notația FEN [10]	6
3.3	Un exemplu de arbore de decizie pentru jocul de șah [11]	7
3.4	Exemplu de optimizare folosind tehnica Alpha-Beta	8
3.5	Digitalizare imagine [16]	10
3.6	Arhitectura generală a unui CNN[18]	11
3.7	Detectie cu Yolov8[19]	12
5.1	Interfața grafică a motorului de inferență pentru jocul de șah	19
5.2	Pseudocod al algoritmului Minimax [27]	20
5.3	Pseudocod al optimizării Alpha-Beta [27]	24
5.4	Arhitectura modelului	26
5.5	Compararea histogramelor înainte și după procesare	28
5.6	Timpi de executie Minimax si Alpha-Beta Pruning	32
5.7	Curbele de invatare returnate la finalul antrenarii	33
5.8	Matricea de confuzie generata pe datele de testare	33
6.1	Piese de șah folosite în cadrul proiectului	35
6.2	Detectarea colțurilor tablei de șah în timp real	38
6.3	Etape de segmentare a tablei în pătrate	39
6.4	Digitalizarea tablei de șah	44
6.5	Detectarea logo	44
6.6	Reprezentare vizuală a reprezentării IOU [32]	46
6.7	Rezultatele antrenamentului YOLOv8 pentru detectia colțurilor	46
6.8	Rezultatele antrenamentului YOLOv8 pentru detectia secventei de mutare	47
6.9	Matricea de Confuzie Normalizată construită cu date de validare	48
6.10	Matricea de Confuzie Normalizată construită cu date de testare	48
6.11	Rezultate de antrenare al modelului de clasificare al pieselor	48
7.1	Interfața grafică în formatul final	49

1 Introducere

Jocul de șah, aşa cum afirma Grandmaster José Raúl Capablanca, "este totul: artă, știință și sport" [1]. Acest joc a captivat mintile oamenilor de-a lungul secolelor și a fost mereu o sursă de fascinație pentru matematicieni, fizicieni și programatori, lăsând o amprentă profundă în domeniul inteligenței artificiale.

Originile șahului pot fi urmărite în India antică, unde era cunoscut sub numele de "Chaturanga" în jurul secolului al VI-lea. Jocul a evoluat de-a lungul timpului, răspândindu-se în Persia, Arabia și Europa, unde a suferit diverse modificări și adaptări. Regulile moderne ale șahului au fost stabilite în secolul al XIX-lea, iar jocul a cunoscut o popularitate tot mai mare, devenind o disciplină sportivă și o formă de artă respectată la nivel global.

Istoria programelor capabile să joace șah se întinde pe parcursul mai multor decenii și a fost marcată de momente-cheie notabile. Alan Turing a creat "Turochamp" în 1948, primul program de șah, deși nu a putut fi rulat pe hardware-ul disponibil la acea vreme. Turochamp a fost un pionier în domeniul inteligenței artificiale și al jocurilor de strategie, ilustrând pentru prima dată potentțialul calculatoarelor de a efectua analize complexe și de a lua decizii strategice. Deși programul nu a fost implementat efectiv pe un calculator din lipsa tehnologiei adecvate, Turing a simulațat manual câteva partide, calculând fiecare mutare conform algoritmilor dezvoltăți de el [2]. Acest lucru a demonstrat nu doar viabilitatea teoretică a programului, ci și capacitatea calculatoarelor de a procesa informații într-un mod care imită gândirea umană. Algoritmul propus de Turing determină toate mutările posibile ale jucătorului, urmate de toate mutările posibile ale adversarului, și evaluează apoi poziția rezultată. Fiecarui jucător îi se atribuie puncte în funcție de piesele rămase pe tablă, cu accent pe piesele situate în centrul tablei, care sunt considerate mai puternice.

IBM a adus contribuții semnificative prin "Deep Thought" în 1980, un calculator specializat în șah care a devenit campion mondial al computerelor în 1988 [3]. Motorul său de inferență funcționa similar cu predecesorul său, dar era îmbunătățit prin utilizarea tehnicii de diminuare a pozițiilor posibile, ceea ce a redus considerabil numărul de poziții ce trebuiau evaluate. În plus, euristica a fost perfecționată prin integrarea unor baze de date extinse cu deschideri și finaluri de partidă, permitând astfel o evaluare mai precisă și strategică a mutărilor.

Cea mai notabilă realizare a venit în 1997, când IBM a lansat "Deep Blue", un sistem de calcul care, pentru prima oară în istorie, a devenit campion mondial la șah [4]. Evaluarea pozițiilor includea acum factori precum structura pionilor, activitatea pieselor, controlul spațiului, potențialele atacuri și tehnici defensive. Puterea de calcul a crescut exponențial în această perioadă, permitând analizarea a aproximativ 200 de milioane de poziții pe secundă, comparativ cu Deep Thought, care putea analiza până la 10 milioane de poziții pe secundă.

Stockfish este în prezent unul dintre cele mai performante motoare de inferență pentru șah din lume [5]. Spre deosebire de Deep Blue, Stockfish beneficiază de acces la baze de date mult mai extinse cu deschideri și finaluri de partidă. Algoritmul său de evaluare este capabil să echilibreze avantajele imediate cu cele pe termen lung. Datorită îmbunătățirilor hardware semnificative, Stockfish poate acum să facă miliarde de estimări pe secundă, oferindu-i o capacitate de analiză mult mai profundă și mai rapidă comparativ cu predecesorii săi. Un element esențial în dezvoltarea continuă a proiectului este faptul că acesta este open source. Acest aspect permite

experților și pasionaților de șah din întreaga lume să contribuie la optimizarea algoritmilor existenți, să adauge informații în bazele de date și să corecteze eventualele erori.

Pe lângă performanțele sale în jocul de șah clasic, Stockfish a fost adaptat și pentru alte variante de șah și puzzle-uri, demonstrându-și flexibilitatea și puterea. De asemenea, a fost integrat în diverse platforme de șah online, unde milioane de jucători îl folosesc zilnic pentru analiză și îmbunătățirea propriilor abilități. Stockfish a devenit un instrument esențial nu doar pentru jucătorii de șah profesioniști, ci și pentru amatori și antrenori, oferind analize detaliate și sugestii care ajută la înțelegerea mai profundă a jocului.

În anul 1980 a fost lansat Boris Handroid (a se vedea Fig. 1.1), prima automatizare a jocului de șah [6]. Dispozitivul era compus dintr-o tablă de șah, piese și un braț robotic. Datorită rarității sale, găsirea de informații despre componentele acestuia este foarte dificilă. Microprocesorul utilizat nu este cunoscut, iar pentru detectarea mutărilor operatorului uman, Boris Handroid folosea un mecanism cu contacte reed, comutatoare ce se activează în prezența unui câmp magnetic. Fiecare pătrat era echipat cu un astfel de contact, iar baza pieselor conținea un magnet. Brațul robotic efectua mutările calculate de motorul de șah Sargon 2.5, folosind trei servomotoare pentru manipulare.

Fostul campion mondial la șah Vladimir Kramnik a jucat în 2011 o partidă de șah împotriva Chess Terminator, un sistem avansat creat de Konstantin Kosteniuk [7]. Poziționarea pieselor pe tablă este determinată de senzori de poziție încorporați în interiorul pieselor, iar mutarea acestora este realizată de un braț robotic cu 6 grade de libertate.

În ultimul deceniu, roboții industriali produși de KUKA Robotics, Niryo, FANUC și ABB (a se vedea Fig. 1.2) au fost integrați în sisteme autonome de jucat șah. Deciziile sunt luate de motoare moderne de inferență, precum Stockfish, iar detectarea pozițiilor pieselor se realizează fie prin senzori de poziție, fie prin sisteme de vedere artificială.



Figura 1.1: Boris Handroid, prima automatizare a jocului de șah [6]



Figura 1.2: Robot ABB inclus într-un sistem de șah [8]

2 Obiectivele lucrării și contribuțiile aduse

Obiectivul principal al acestei lucrări este dezvoltarea unui partener autonom pentru jocul de șah. Pentru a atinge acest scop, au fost abordate trei provocări: decizia mutării optime, determinarea stării curente a tablei și executarea mutărilor pieselor.

Prima provocare, cea decizională, implică identificarea mutării optime în fiecare moment al jocului. În această lucrare, au fost analizați și implementați algoritmi de căutare Minimax și Alpha-Beta Pruning. Evaluarea pozițiilor de șah s-a realizat prin două metode distincte. Prima metodă evaluează poziția pe baza informațiilor despre piesele rămase pe tablă. Pentru o estimare mai precisă și pentru îmbunătățirea performanței algoritmului, a fost implementată o a doua metodă, care utilizează o rețea neuronală antrenată pe date etichetate de Stockfish.

A doua provocare se referă la determinarea stării curente a tablei de șah. Pentru a lua decizii corecte, este esențial să avem informații precise despre configurația actuală a tablei. În acest scop, a fost dezvoltat un sistem de vedere artificială capabil să detecteze tabla de șah, piesele și momentul în care robotul trebuie să facă o mutare. Acest sistem este robust la variațiile de unghiuri de vizualizare și la schimbările de iluminare.

Ultima provocare constă în mutarea pieselor. După analiza tablei și luarea unei decizii, brațul robotic trebuie să execute mutarea aleasă într-un mod sigur și eficient, fără a pune în pericol adversarul uman sau mediul de lucru.

Integrarea acestor trei componente a condus la crearea unui sistem autonom complet, capabil să joace șah.

Contribuțiile personale în această lucrare includ implementarea algoritmilor Minimax și Alpha-Beta în limbajul Python, bazate pe explicațiile din literatura de specialitate. Au fost dezvoltate funcții pentru generarea tuturor pozițiilor posibile plecând de la o configurație inițială și o funcție de evaluare a pozițiilor, atribuind fiecărei piese un scor specific. Pentru antrenarea unui model neuronal capabil să evalueze mai precis pozițiile de șah, a fost utilizată o bază de date conținând peste 12 milioane de poziții etichetate cu evaluări ale Stockfish. Arhitectura modelului, inspirată dintr-un articol științific orientat în aceeași direcție, a fost adaptată la datele de antrenament, fiind implementate funcții de preprocesare a datelor, antrenare și evaluare a modelului. De asemenea, a fost dezvoltată o interfață grafică pentru a permite utilizatorului să vizualizeze mutările algoritmului decizional și să simuleze partide de șah.

Contribuțiile aduse în domeniul vederii artificiale includ antrenarea a trei modele neuronale utilizând arhitectura Yolov8. Fiecare dintre aceste modele are un rol specific. Primul model detectează colțurile tablei de șah, rezultatele fiind utilizate pentru a împărți tabla în cele 64 de pătrate distincte. După această împărțire, al doilea model clasifică piesele în funcție de tipul lor. Al treilea model detectează jucătorul căruia îi revine mutarea, identificând prezența unor desene specifice aflate lângă tabla de șah. Sistemul de vedere artificială se folosește de rezultatele acestor 3 modele pentru a forma o reprezentare digitală a tablei de sah, ce poate fi oferita mai departe algoritmului decizional.

În final, a fost implementată logica jocului, utilizând o cameră video îndreptată spre tabla de șah. Aceasta extrage informațiile relevante din fiecare cadru folosind sistemul de vedere artificială, trimite aceste informații algoritmului decizional și returnează mutarea aleasă de

acesta. De asemenea, a fost dezvoltată o a doua interfață grafică, care permite observarea transmisiei video, reprezentarea digitală a tablei aşa cum este percepă de sistemul de vedere artificială și mutarea aleasă de partea decizională.

3 Fundamente teoretice

3.1 Motorul de inferențe pentru jocul de șah

Un motor de inferențe în cazul jocului de șah este un sistem software specializat în analiza pozițiilor pe tabla de șah, inclusiv variante ale jocului clasic. Acest program calculează și generează o mutare sau o listă de mutări pe care le consideră cele mai avantajoase pentru jucător.

3.1.1 Reprezentarea tablei de șah folosind notația Forsyth-Edwards (FEN)

Reprezentarea unei configurații a tablei de șah într-un format ușor de procesat de către calculatoare este realizată prin intermediul Notației Forsyth-Edwards.

Acest sistem de notație, creat de programatorul Steven J. Edwards, a derivat dintr-un preexistent dezvoltat de jurnalistul David Forsyth. Cu modificările aduse, Edwards a adaptat sistemul de notație astfel încât software-ul de șah să-l poată utiliza în mod eficient [9].

FEN permite descrierea unei poziții printr-o singură linie de text, un aspect esențial pentru partajarea și recrearea pozițiilor în mediul digital al șahului. Acest fapt facilitează interacțiunea între jucători și programele de șah, eliminând necesitatea trimiterii unor fișiere extinse.

Un sir FEN (Forsyth-Edwards Notation) este structurat în șase câmpuri distincte, fiecare furnizând informații specifice despre poziția de șah. Aceste câmpuri sunt delimitate de caractere spațiu.

- Plasarea pieselor (Piece Placement):** Acest prim câmp descrie modul în care piesele sunt dispuse pe tabla de șah, începând de la rândul al optulea și continuând către primul. Literele majuscule reprezintă piesele albe (pion, turn, cal, nebun, regină, rege), în timp ce literele mici indică piesele negre. Numerele sunt utilizate pentru a indica câte pătrate goale urmează între piese.
- Culoarea activă (Active Color):** Acest câmp indică culoarea jucătorului aflat la mutare. Litera 'w' reprezintă alb (white), iar 'b' reprezintă negru (black).
- Drepturile de rocadă (Castling Rights):** Informează cu privire la drepturile de rocadă ale ambilor jucători. Literele 'K', 'Q', 'k', și 'q' semnifică disponibilitatea rocadelor regale și regine pentru alb și negru.
- Possible ținte pentru "en passant" (En Passant Targets):** Acest câmp indică păratul pe care o captură "en passant" este posibilă. Dacă nu există această posibilitate, se folosește caracterul '-'. Captura "en passant" (în trecere) este un tip unic de captură în șah. Ea apare atunci când un pion avansează două pătrate din poziția sa inițială și trece pe lângă un pion advers situat pe aceeași coloană sau pe coloana alăturată. Pionul advers are atunci posibilitatea de a captura pionul ca și cum acesta ar fi avansat doar un pătrat. Captura "en passant" trebuie efectuată imediat, în mutarea următoare a adversarului; în caz contrar, dreptul de a efectua această captură este pierdut.

5. **Ceasul de semimutări (Halfmove Clock):** Aceasta indică numărul de mutări consecutive care au avut loc fără ca vreun pion să fie mutat sau să fie realizată o captură. Acest aspect este relevant pentru regulile de remiză.
6. **Numărul total de mutări (Fullmove Number):** Acest câmp indică numărul total de mutări realizate în joc.

Figura 3.1 ilustrează poziția de start într-un joc de șah, cu toate piesele plasate în ordinea lor inițială pe tabla de șah. Acest aranjament specific de piese este reprezentat de Forsyth-Edwards Notation (FEN) prin sirul de caractere: "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPP/RNBQKBNR".

În figura 3.2 este explicat cum se formează câmpul de plasare a pieselor în notația FEN pentru o altă configurație a tablei de șah. Acest câmp descrie poziția pieselor pentru aranjamentul: "r1bk3r/p2pBpNp/n4n2/1p1NP2P/6P1/3P4/P1P1K3/q5b1"



Figura 3.1: Poziția de start în jocul de șah [10]

Figura 3.2: Câmpul de plasare a pieselor în notația FEN [10]

3.1.2 Algoritmul Minimax

Algoritmul Minimax este un algoritm de decizie utilizat în teoria jocurilor și inteligența artificială pentru a determina mișcarea optimă într-un joc de două persoane, precum șahul, plecând de la premisa că ambii jucători aleg mereu cea mai avantajoasă opțiune [11]. Principiul de bază al algoritmului este să minimizeze pierderea maximă posibilă.

În prima etapă a algoritmului Minimax, se construiește un arbore în care fiecare nod reprezintă o poziție posibilă în joc, iar fiecare ramură reprezintă o acțiune necesară pentru a ajunge dintr-un nod în altul. Algoritmul explorează acest arbore până la o adâncime prestabilită, evaluând fiecare nod cu ajutorul unei funcții de evaluare.

Algoritmului Minimax se bazează pe explorarea în profunzime a tuturor mișcărilor posibile dintr-o stare dată a jocului până la o anumită adâncime, evaluând astfel fiecare mișcare pe baza unui scor predeterminat. În fiecare nod al arborelui jocului, Minimax presupune că jucătorul aflat la mutare va alege mișcarea care îi maximizează beneficiul, iar adversarul său va alege mișcarea care minimizează beneficiul jucătorului inițial. Astfel, algoritmul construiește un arbore de joc în care frunzele reprezintă stările finale ale jocului, iar scorurile acestor stări sunt propagate înapoi prin arbore pentru a determina valoarea fiecărui nod intermediu.

Minimax funcționează recursiv prin evaluarea pozițiilor terminale și propagarea valorilor acestora înapoi către rădăcină. Nodurile corespunzătoare jucătorului care maximizează

selectează valoarea maximă dintre toate valorile fiilor săi, în timp ce nodurile corespunzătoare jucătorului care minimizează selectează valoarea minimă. Acest proces continuă până la rădăcina arborelui, unde algoritmul determină mutarea care corespunde valorii Minimax optime.

Un aspect important al algoritmului Minimax este că presupune că ambii jucători joacă optim. Cu alte cuvinte, jucătorul care maximizează încearcă să obțină cel mai mare scor posibil, în timp ce jucătorul care minimizează încearcă să obțină cel mai mic scor posibil pentru adversar. În practică, acest lucru înseamnă că algoritmul explorează toate posibilitățile și ia în considerare cele mai bune mișcări posibile ale ambilor jucători, oferind astfel o decizie optimă pentru jocul respectiv.

Algoritmul Minimax poate fi aplicat în jocul de șah prin asocierea fiecărei poziții posibile de pe tablă cu un nod, aşa cum este ilustrat în Figura 3.3. Începând cu poziția de start ca rădăcină, fiecare arc al arborelui simbolizează o mutare legală disponibilă pentru o piesă albă (corespunzător jucătorului ce începe). Nodurile asociate acestor arce reprezintă stările succesive ale jocului, reflectând evoluția poziției în urma fiecărei mutări. Procesul se extinde la nivelul următor, unde se adaugă arce corespunzătoare mișcărilor pieselor negre. Continuând acest model până la adâncimea stabilității, jocul de șah poate fi reprezentat ca un arbore de decizie, care poate fi navigat de la nivelul frunzelor. Estimările pozițiilor finale sunt propagate înapoi în arbore până la rădăcină. Când aceste valori ajung la rădăcină, jucătorul care maximizează poate alege poziția cu scorul cel mai mare, știind că aceasta conduce la cel mai bun rezultat dacă adversarul joacă cea mai bună mutare a sa. Dacă oponentul face o mutare mai slabă, rezultatul obținut va fi chiar mai favorabil decât cel prevăzut inițial.

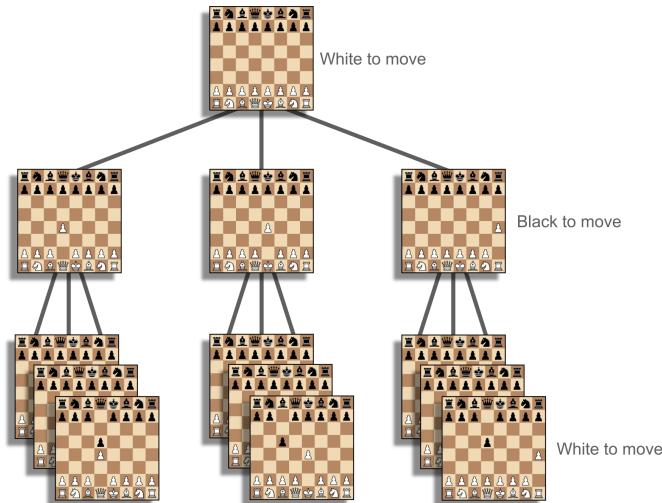


Figura 3.3: Un exemplu de arbore de decizie pentru jocul de șah [11]

Avantaje ale algoritmului Minimax:

- Determinarea optimă a mutărilor:** Algoritmul Minimax este capabil să determine cea mai bună secvență de mutări pentru un jucător într-un joc în care toate informațiile necesare pentru a lua decizii sunt disponibile și vizibile pentru ambii jucători.
- Optimizare strategică:** Prin explorarea exhaustivă a arborelui de joc, algoritmul permite identificarea unei strategii care maximizează rezultatul favorabil. Dacă arborele de joc este explorat complet și există o strategie câștigătoare, algoritmul Minimax garantează victoria.

Dezavantaje ale algoritmului Minimax:

- Complexitate computațională:** Necesitatea parcurgerii complete a arborelui poate implica un efort computațional semnificativ.

2. **Timp de calcul extins:** Parcurgerea exhaustivă a arborelui poate conduce la un timp de calcul prelungit.
3. **Dependenta de funcția de evaluare:** În cazul în care explorarea completă a arborelui nu este posibilă, algoritmul Minimax este dependent de o funcție de estimare foarte bună pentru a returna un rezultat bun. Calitatea deciziilor luate de algoritm depinde astfel de acuratețea acestei funcții de evaluare.

În contextul șahului, există un impresionant număr de 318,979,564,000 de moduri distințe în care jucătorii pot realiza primele 4 mutări, iar cea mai lungă partidă teoretic posibilă ar implica 5,949 de mutări [12]. Această complexitate evidențiază imposibilitatea unei căutări exhaustive a tuturor scenariilor în cadrul jocului de șah. Este evident că algoritmul Minimax, în forma sa pură, nu reprezintă o soluție viabilă pentru un motor de inferențe pentru șah.

3.1.3 Optimizarea algoritmului Minimax cu Alpha-Beta Pruning

Dezavantajul legat de timpul extins de calcul în algoritmul Minimax poate fi diminuat prin integrarea tehnicii Alpha-Beta Pruning. Aceasta reprezintă o tehnică de optimizare care reduce numărul de noduri evaluate în arborele de joc, accelerând astfel procesul de luare a deciziilor [13].

În cadrul algoritmului Minimax, Alpha-Beta Pruning funcționează prin utilizarea a doi parametri, **alpha** și **beta**, care reprezintă limitele inferioară și superioară ale valorilor posibile ale unui nod. Atunci când se explorează arborele, dacă valoarea unui nod se dovedește a fi în afara intervalului $[\alpha, \beta]$, explorarea ulterioară a acestui nod poate fi evitată, deoarece rezultatele nu vor influența decizia finală. Aceasta duce la eliminarea unor ramuri întregi ale arborelui care nu ar contribui semnificativ la găsirea celei mai bune mutări. Alpha-Beta Pruning începe cu inițializarea lui **alpha** la cea mai mică valoare posibilă și a lui **beta** la cea mai mare valoare posibilă. Pe măsură ce algoritmul explorează arborele de joc, **alpha** este actualizat cu cea mai bună valoare maximă găsită de-a lungul căii curente pentru jucătorul care maximizează, iar **beta** este actualizat cu cea mai bună valoare minimă găsită pentru jucătorul care minimizează.

În timpul explorării, dacă algoritmul determină că valoarea curentă a unui nod este mai mică decât **alpha** pentru jucătorul care minimizează sau mai mare decât **beta** pentru jucătorul care maximizează, acel nod și toate nodurile care derivă din el sunt excluse din procesul de explorare. Acest lucru se întâmplă deoarece jucătorul adversar ar evita în totdeauna o mutare care ar conduce la un rezultat mai nefavorabil decât cel deja cunoscut (**alpha** sau **beta**).

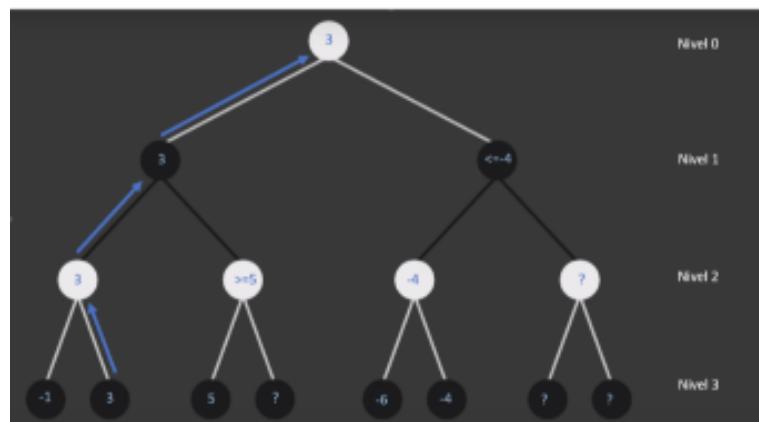


Figura 3.4: Exemplu de optimizare folosind tehnica Alpha-Beta

În Figura 3.4 este prezentat un exemplu simplificat de optimizare Alpha-Beta. Nivelul 3 reprezintă estimările pozițiilor după 3 mutări. Arcele albe indică faptul că jucătorul alb face

alegerea. Deoarece jucătorul alb maximizează, va alege cea mai mare valoare dintre nodurile copil. Se observă că valoarea care urcă la nivelul 2 este 3 pentru prima pereche de noduri și cel puțin 5 pentru a doua pereche de noduri.

Următorul set de arce semnifică faptul că următoarea mutare este pentru jucătorul negru, care minimizează. Prin urmare, nodul ales dintre cele două va fi 3. Astfel, putem observa cum eliminăm din algoritm toți copiii din al doilea set de noduri după ce am descoperit cel puțin unul cu o valoare mai mare decât minimul pentru acel nod.

3.2 Sisteme de Vedere Artificială

Sistemele de vedere artificială reprezintă o ramură a inteligenței artificiale dedicată interpretării și înțelegерii imaginilor din lumea reală de către calculatoare și sisteme automate. Aceste sisteme utilizează algoritmi complecsi și modele de învățare automată pentru a procesa și analiza datele vizuale, facilitând recunoașterea obiectelor, detectarea anomaliei, urmărirea mișcărilor și interpretarea scenelor vizuale [14].

În domeniul industrial, sistemele de vedere artificială oferă un avantaj semnificativ datorită flexibilității lor de adaptare și integrare în diverse aplicații. Aceste sisteme funcționează prin capturarea imaginilor cu ajutorul camerelor, transformându-le în date digitale care sunt apoi analizate. Procesul de analiză include preprocesarea pentru a îmbunătăți calitatea imaginii și a elimina zgomotul sau distorsiunile, detectarea marginilor și formelor și extragerea caracteristicilor relevante. Modelele de învățare automată sunt antrenate pentru a identifica informațiile esențiale, facilitând luarea deciziilor ulterioare.

3.2.1 Imagine digitală

Imaginiile pot fi de două tipuri: analogice și digitale. O imagine analogică este continuă și nediscretă, reprezentând o imagine naturală. În contrast, o imagine digitală este bidimensională și formată dintr-un set finit de valori digitale, numite elemente de imagine sau pixeli. Imaginea digitală este compusă din pixeli discreți, fiecare având o valoare numerică asociată care reprezintă luminozitatea, cunoscută și sub denumirea de nivel de gri.

O imagine reală se formează pe un senzor atunci când energia emisă atinge senzorul cu suficientă intensitate pentru a genera un semnal de ieșire. Imaginea digitală este reprezentată ca o funcție bidimensională, $f(x, y)$, unde x și y sunt coordonatele spațiale, iar amplitudinea lui f la orice pereche de coordonate (x, y) reprezintă intensitatea sau nivelul de gri al imaginii în acel punct [15].

Digitalizarea implică transformarea unei imagini reale într-un format digital(a se vedea Fig. 3.5), iar precizia acestei transformări este determinată de două tipuri de rezoluție: rezoluția de pixeli și rezoluția de luminozitate. Rezoluția de pixeli este definită de numărul de pixeli care compun imaginea; cu cât numărul de pixeli este mai mare, cu atât detaliile din imagine sunt mai clare și mai precise.

Rezoluția de luminozitate se referă la acuratețea cu care sunt cuantificate nivelurile de lumină din imagine. Aceasta este măsurată în biți, unde un număr mai mare de biți permite o gamă mai largă de valori de luminozitate, rezultând într-o imagine cu gradații de culoare și umbre mai subtile. De exemplu, o imagine cu o rezoluție de 8 biți poate reprezenta 256 de niveluri de gri, în timp ce una cu 16 biți poate reprezenta 65.536 de niveluri de gri, oferind o acuratețe și un detaliu mult mai mare în variațiile de lumină.

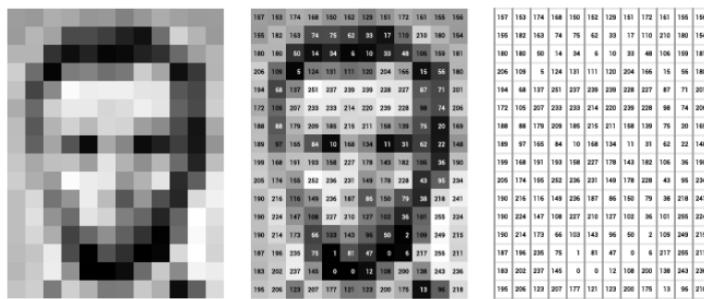


Figura 3.5: Digitalizare imagine [16]

În timpul capturării imaginii, senzorii din camerele digitale sunt acoperiți cu un filtru de culoare care permite trecerea luminii roșii, verzi sau albastre. Fiecare pixel al senzorului captează lumina pentru una dintre cele trei culori. Astfel, se generează trei imagini separate în tonuri de gri: una pentru roșu, una pentru verde și una pentru albastru. Aceste imagini sunt apoi combinate pentru a crea imaginea colorată finală.

3.2.2 Preprocesarea Imaginilor

În domeniul recunoașterii imaginilor și al detecției obiectelor, etapa de preprocesare este esențială și are un impact semnificativ asupra acurateței și eficienței procesului de detecție [14]. Preprocesarea implică diverse etape care pregătesc datele brute ale imaginilor pentru analiza ulterioară, îmbunătățind calitatea imaginii, reducând zgomotul și standardizând intrările pentru algoritmii de detecție. Printre pașii principali de preprocesare utilizati frecvent în detecția obiectelor se numără: redimensionarea imaginilor la dimensiuni standard, conversia acestora în niveluri de gri pentru a simplifica datele și a reduce complexitatea calculatoarelor, și binarizarea, care transformă imaginile în alb-negru pentru a evidenția mai clar obiectele față de fundal. Normalizarea imaginilor, prin ajustarea valorilor intensității pixelilor într-un interval standard, este esențială pentru a asigura consistența datelor.

3.2.3 Utilizarea rețelelor neuronale pentru detecția de obiecte din imagini

Rețelele neuronale, în special rețelele neuronale convoluționale (CNN-uri), sunt instrumente puternice utilizate, printre altele, pentru detecția de obiecte în imagini [17]. Acestea pot învăța să identifice și să localizeze diverse obiecte într-o imagine printr-un proces de antrenare pe seturi mari de date etichetate. Rețelele neuronale convoluționale sunt capabile să recunoască modele și să extragă caracteristici relevante din imagini, ceea ce le face extrem de eficiente în sarcini de vedere artificială.

O rețea neuronală convoluțională este un tip de rețea neuronală cu mai multe straturi, special concepută pentru a procesa date organizate într-un format bidimensional, cum ar fi imaginile. CNN-urile sunt inspirate de arhitectura cortexului vizual al mamiferelor și sunt compuse din mai multe straturi specializate care detectează diferite caracteristici ale imaginilor. Aceste straturi includ straturi convoluționale, straturi de pooling și straturi complet conectate, fiecare având un rol specific în procesarea imaginii. Arhitectura generală a unui CNN este prezentată în figura 3.6.

Straturile convoluționale sunt responsabile pentru aplicarea unor filtre (sau kerneluri) pe imaginea de intrare pentru a extrage caracteristici locale, cum ar fi marginile, colțurile și texturile. Fiecare filtru este o matrice mică de numere care trece peste imagine, calculând

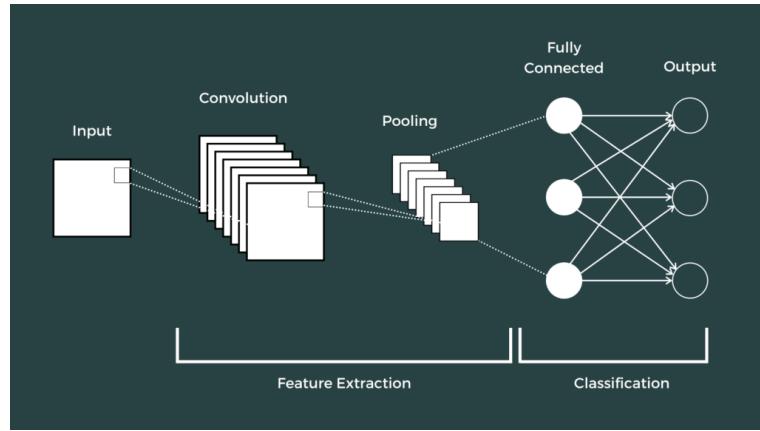


Figura 3.6: Arhitectura generală a unui CNN[18]

produse punctuale între valorile filtrului și secțiunile corespunzătoare ale imaginii. Rezultatul acestor calcule este o nouă imagine numită hartă de caracteristici (feature map), care evidențiază zonele din imaginea originală unde anumite caracteristici sunt prezente. Harta de caracteristici este o reprezentare vizuală a datelor unde valorile individuale sunt reprezentate prin culori. În contextul CNN-urilor, o hartă de caracteristici arată care părți ale imaginii inițiale sunt activate de anumite filtre, indicând zonele unde anumite caracteristici (de exemplu, margini sau texturi) au fost detectate.

Straturile de pooling au rolul de a reduce dimensiunea hărților de caracteristici prin agregarea valorilor într-o zonă specifică. Există mai multe tipuri de pooling, cele mai comune fiind max pooling și average pooling. Max pooling implică împărțirea hărții de caracteristici în regiuni mai mici și selectarea valorii maxime din fiecare regiune. De exemplu, dacă avem o zonă de 2×2 pixeli, max pooling va lua cea mai mare valoare dintre acei pixeli și va crea o hartă de caracteristici mai mică pe baza acestor valori maxime. Acest proces ajută la păstrarea celor mai importante informații, reducând în același timp dimensiunea hărții de caracteristici și complexitatea calculului. Average pooling, pe de altă parte, calculează media valorilor dintr-o regiune. Folosind același exemplu de zonă de 2×2 pixeli, average pooling va calcula media valorilor din acea zonă și va crea o hartă de caracteristici mai mică pe baza acestor medii. Aceasta metodă ajută la reducerea zgomotului și la obținerea unei reprezentări generalizate a caracteristicilor imaginii.

Straturile complet conectate sunt utilizate pentru a combina caracteristicile extrase de straturile conveționale și de pooling într-un set de ieșiri finale. Aceste straturi funcționează similar cu straturile din rețelele neuronale clasice, unde fiecare neuron este conectat la toți neuronii din stratul anterior. Straturile complet conectate sunt esențiale pentru clasificarea finală a obiectelor detectate în imagini.

Pentru a antrena eficient o rețea neuronală în recunoașterea obiectelor, este esențial să avem un set de imagini etichetate, specificând obiectele și locațiile acestora sub formă de cutii de delimitare. Imaginele trebuie preprocesate prin pași precum redimensionarea pentru dimensiuni uniforme, normalizarea pentru ajustarea valorilor de intensitate a pixelilor și convertirea în tonuri de gri pentru reducerea complexității computaționale, toate acestea îmbunătățind calitatea și consistența datelor de intrare, optimizând performanța modelului.

3.2.4 Modelul Yolov8

YOLO (You Only Look Once) este o familie de modele de învățare profundă specializate în detectarea obiectelor în imagini și videoclipuri. Aceste modele au fost dezvoltate de Joseph

Redmon și Ali Farhadi și sunt cunoscute pentru capacitatea lor de a efectua detectarea obiectelor în timp real, ceea ce le face extrem de utile în aplicații care necesită procesarea rapidă a imaginilor. YOLO transformă problema detectării obiectelor într-o singură problemă de regresie, estimând direct coordonatele casetelor de delimitare și probabilitățile claselor de obiecte într-o singură etapă, în loc de a diviza această sarcină în mai multe sub-sarcini, așa cum fac alte metode de detectare [19].

YOLOv8 reprezintă cea mai recentă versiune a acestei serii de modele, oferind îmbunătățiri semnificative în ceea ce privește acuratețea și viteza. YOLOv8 utilizează o arhitectură de rețea neuronală convolutională optimizată, capabilă să proceseze imagini la o viteză ridicată, ceea ce o face potrivită pentru aplicații de detectare a obiectelor în timp real. Una dintre cheile succesului YOLO este capacitatea sa de a împărți imaginea de intrare într-o grilă și de a face predicții simultane pentru fiecare secțiune a grilei, reducând astfel timpul de procesare comparativ cu metodele tradiționale de detecție.

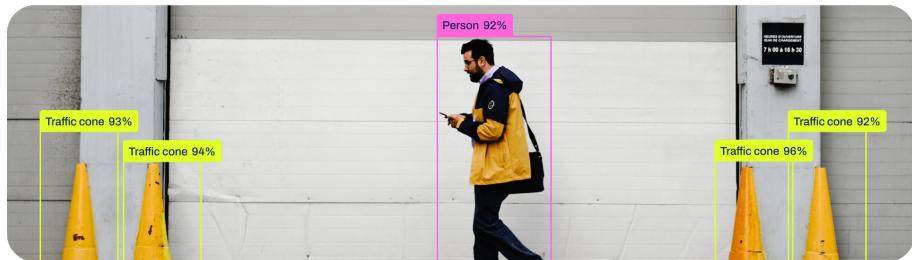


Figura 3.7: Detectie cu Yolov8[19]

În figura 3.7 este prezentat un exemplu de utilizare a modelului YOLOv8. Rezultatele acestui model, după rularea pe imagine, includ coordonatele obiectelor, clasele acestora și nivelurile de precizie. În exemplul ilustrat, chenarele sunt desenate în jurul obiectelor detectate, indicând clasa din care fac parte și nivelul de încredere al detectării.

YOLOv8 este capabil să proceseze cadre video în timp real fără a compromite semnificativ acuratețea detectiilor. Acest lucru este posibil datorită abordării unificate a YOLO, care efectuează toate operațiunile de detectare într-o singură trecere prin rețea. În plus, YOLOv8 introduce optimizări la nivel de arhitectură, algoritmi de antrenament mai eficienți și tehnici de augmentare a datelor pentru a îmbunătăți și mai mult performanța.

În acest proiect, modelul YOLO a fost utilizat pentru recunoașterea tablei și pieselor de sah.

3.2.5 Antrenarea personalizată a modelului Yolov8

Deși YOLOv8 este un model preantrenat capabil să detecteze cu acuratețe obiecte comune precum oameni sau mașini, performanța sa în scenarii specifice este redusă din cauza diversității și variabilității contextelor particulare. În cadrul acestui proiect, este necesară o etapă de antrenare personalizată, deoarece YOLOv8 nu ar putea recunoaște piesele specifice de sah fără acest pas. Antrenarea pe un set personalizat de date va permite modelului să se adapteze precis la nevoile aplicației.

Antrenarea YOLOv8 pe un set de date personalizat oferă o serie de avantaje considerabile. În primul rând, acest proces permite modelului să învețe caracteristicile unice ale obiectelor care sunt specifice aplicației respective, crescând astfel acuratețea și reducând rata de erori în detectare. În al doilea rând, modelul devine capabil să se adapteze la condițiile particulare ale mediului de utilizare, cum ar fi variațiile de iluminare, unghierile de vizualizare, și fundalurile specifice care ar putea influența negativ performanța unui model antrenat pe date generale.

Prima etapă în antrenarea modelului YOLOv8 pe un set de date personalizat constă în colectarea și etichetarea datelor. Acest proces implică adunarea unui număr suficient de imagini relevante pentru sarcina de detecție și etichetarea acestora utilizând instrumente specializate. Etichetarea include definirea precisă a regiunilor de interes (bounding boxes) și clasificarea obiectelor corespunzătoare. În situațiile în care setul de date este insuficient, augmentarea datelor reprezintă o alternativă frecvent utilizată. Aceasta implică aplicarea unor tehnici precum modificarea iluminării și a contrastului, menite să pregătească modelul pentru diverse scenarii de mediu, transformări de perspectivă pentru a permite recunoașterea obiectelor din unghiuri diferite, și deformări pentru a asigura detectarea în condiții variate de calitate a imaginii.

După ce datele au fost organizate într-un format compatibil cu modelul, următorul pas esențial pentru antrenarea YOLOv8 este ajustarea hiperparametrilor. Printre acești parametri se numără rata de învățare, care controlează viteza de actualizare a parametrilor interni ai modelului pe parcursul antrenamentului. Dimensiunea setului de date reprezintă numărul de imagini procesate simultan împreună cu actualizarea parametrilor, influențând stabilitatea și eficiența antrenamentului. Numărul de epoci se referă la numărul complet de trece prin întregul set de date de antrenament, determinând capacitatea modelului de a învăța și generaliza. Parametrii de regularizare, cum ar fi dropout-ul și ponderile L2, sunt utilizați pentru a preveni supraînvățarea, asigurând astfel o performanță bună pe date noi. În final, arhitectura rețelei, inclusiv numărul și dimensiunea straturilor convoluționale, poate fi ajustată pentru a echilibra complexitatea modelului și puterea sa de reprezentare.

4 Resurse folosite în cadrul proiectului

4.1 Limbajul de programare Python

Python, un limbaj de programare de nivel înalt dezvoltat de Guido van Rossum și lansat pentru prima dată în 1991, a evoluat constant pentru a deveni unul dintre cele mai populare limbi [20]. De la începuturile sale, Python a fost proiectat pentru a pune accentul pe lizibilitatea și simplitatea codului, oferind o sintaxă clară și intuitivă, ceea ce îl face accesibil atât pentru începători, cât și pentru programatori experimentați. Aceasta a permis dezvoltarea rapidă a unui ecosistem vast de biblioteci și instrumente specializate care susțin diverse aplicații științifice și industriale.

În contextul machine learning-ului, Python se evidențiază printr-un ecosistem bogat de biblioteci specializate, cum ar fi TensorFlow, scikit-learn și PyTorch. Aceste biblioteci oferă funcționalități avansate și algoritmi pre-construși, facilitând dezvoltarea și implementarea modelelor complexe de învățare automată. Capacitatea Python de a manipula și analiza date de mari dimensiuni cu ușurință, prin intermediul unor biblioteci precum Pandas și NumPy, îl face ideal pentru pregătirea și preprocesarea datelor, etape esențiale în machine learning. În plus, comunitatea activă și vastă din jurul Python asigură un suport constant și actualizări frecvente, menținând limbajul la curent cu cele mai recente descoperiri și practici din domeniul.

Sistemele de vedere artificială beneficiază semnificativ de pe urma utilizării Python, datorită bibliotecilor puternice precum OpenCV și PIL (Python Imaging Library). Aceste biblioteci permit dezvoltatorilor să implementeze algoritmi de procesare a imaginii și recunoaștere a obiectelor cu ușurință, oferind funcții avansate pentru detectarea, analiza și clasificarea imaginilor.

În domeniul roboticii, Python este de asemenea o alegere excelentă datorită versatilității și ușurinței sale de integrare cu diverse platforme și hardware. Biblioteci precum Robotics Toolbox pentru Python și ROS (Robot Operating System) oferă un set complet de instrumente pentru simularea, controlul și programarea roboților. Python permite interfațarea eficientă cu senzori și actuatoare, facilitând dezvoltarea de sisteme robotice.

Limbajul de programare Python a fost utilizat pentru implementarea tuturor componentelor acestui proiect.

4.2 Biblioteci ale limbajului Python folosite în cadrul proiectului

4.2.1 PyTorch

PyTorch, o bibliotecă open-source dezvoltată de Facebook's AI Research lab, a devenit rapid una dintre cele mai populare instrumente pentru dezvoltarea aplicațiilor de machine learning și deep learning [21]. Lansată în 2016, PyTorch a câștigat aprecierea comunității de cercetători și dezvoltatori datorită flexibilității sale, a simplității în utilizare și a puterii sale de expresivitate în definirea și antrenarea modelelor neuronale.

PyTorch se remarcă și prin suportul său robust pentru accelerarea hardware, în special prin integrarea CUDA (Compute Unified Device Architecture) de la NVIDIA. Această integrare permite utilizarea unităților de procesare grafică (GPU) pentru a accelera calculele necesare în antrenarea modelelor de deep learning. GPU-urile sunt esențiale pentru antrenarea rapidă și eficientă a rețelelor neuronale complexe, iar CUDA facilitează distribuirea sarcinilor de calcul pe multiple nuclee de procesare ale GPU-urilor, reducând semnificativ timpul necesar pentru antrenarea modelelor. Această capacitate este esențială pentru aplicațiile care necesită procesarea unor volume mari de date și efectuarea unui număr mare de operații matematice, cum ar fi recunoașterea imaginii, procesarea limbajului natural și simulările fizice.

În cadrul acestui proiect, PyTorch a fost folosit pentru construirea, antrenarea și evaluarea unei rețele neuronale. Pe lângă funcțiile care permit crearea facilă a strukturilor rețelei, PyTorch oferă acces la diverse funcții de activare și algoritmi de optimizare, facilitând, de asemenea, gestionarea seturilor de date. Suportul pentru accelerarea calculelor folosind GPU a fost un element esențial în realizarea acestui proiect.

4.2.2 OpenCV

OpenCV (Open Source Computer Vision Library) este o bibliotecă de vedere artificială multiplatformă gratuită, dezvoltată inițial de Intel, destinată procesării imaginilor în timp real [22]. Dezvoltarea activă a interfețelor pentru Python, Ruby, Matlab și alte limbaje de programare permite integrarea facilă a OpenCV în diverse proiecte. Cu peste 2500 de algoritmi și o documentație extensivă, OpenCV oferă cod sursă și exemple pentru vederea artificială în timp real, facilitând crearea de aplicații, produse și proiecte de cercetare.

În cadrul acestui proiect, au fost utilizate preponderent următoarele funcții ale bibliotecii OpenCV: activarea și controlul camerei pentru capturarea imaginilor, gestionarea și procesarea informației pentru fiecare cadru individual, aplicarea de filtre pentru conversia imaginilor în tonuri de gri, realizarea de transformări de perspectivă și segmentarea imaginilor.

4.2.3 Tkinter și PIL

Tkinter este biblioteca standard pentru crearea interfețelor grafice în Python, integrată direct în distribuția standard a limbajului [23]. Este o alegere populară datorită ușurinței de utilizare și a documentației accesibile, fiind ideală pentru începători. Tkinter permite dezvoltatorilor să construiască aplicații grafice cu diverse widget-uri, cum ar fi butoane, etichete, meniu și câmpuri de text, pe care le organizează într-un mod flexibil folosind layout-uri precum grid, pack și place. Este compatibilă cu multiple platforme, inclusiv Windows, macOS și Linux, ceea ce asigură portabilitatea aplicațiilor fără necesitatea modificărilor majore.

Pillow (PIL - Python Imaging Library) este o bibliotecă esențială pentru manipularea imaginilor în Python [23]. Aceasta permite deschiderea, manipularea și salvarea diverselor formate de imagine, precum JPEG, PNG și GIF. Printre funcționalitățile sale se numără redimensionarea, decuparea, rotirea și aplicarea diferitelor filtre și efecte asupra imaginilor.

Cele două biblioteci au fost utilizate împreună pentru a implementa interfețele grafice din proiect. Tkinter a fost folosit pentru construirea și organizarea elementelor grafice ale interfeței, în timp ce Pillow a permis manipularea și prelucrarea imaginilor afișate în cadrul aplicațiilor dezvoltate.

4.2.4 Chess

Biblioteca Chess este un instrument pentru analiza și gestionarea jocurilor de șah [24]. Printre cele mai importante funcționalități ale acestei biblioteci se numără reprezentarea și manipularea tablei de șah, compatibilă cu formatul FEN, și funcția de generare a tuturor mutărilor posibile, crucială pentru implementarea algoritmului Minimax. De asemenea, biblioteca oferă funcții pentru validarea mutărilor legale, asigurând conformitatea cu regulile oficiale ale jocului de șah.

4.3 Camo

Camo este o aplicație dezvoltată de Reincubate, care transformă camera telefonului mobil într-o cameră virtuală, utilizabilă pe diverse platforme și aplicații [25]. Această soluție este extrem de utilă pentru utilizatorii care doresc să îmbunătățească calitatea video a streaming-urilor live sau a altor aplicații care necesită captură video.

Camerele telefoanelor mobile moderne au evoluat considerabil, oferind rezoluții înalte. Camo permite utilizatorilor să valorifice aceste capacitați, oferind o calitate video superioară în comparație cu multe camere web tradiționale.

Integrarea Camo în aplicații proprii este simplificată prin utilizarea API-urilor oferite, care permit accesul la fluxul video. Aceasta permite dezvoltatorilor să încorporeze funcționalitățile camerei virtuale în propriile aplicații cu ușurință.

În cadrul acestui proiect, aplicația Camo a fost utilizată pentru a transmite un flux video live de la telefonul mobil către computer. Aceasta a permis folosirea camerei telefonului ca o cameră virtuală, facilitând astfel gestionarea fiecărui frame video de către codul proiectului.

4.4 Roboflow

Roboflow este o platformă pentru gestionarea și preprocesarea seturilor de date vizuale, oferind unelte pentru adnotarea și etichetarea obiectelor din imagini [26]. Aceasta facilitează întregul proces de pregătire a datelor, de la încărcarea și organizarea imaginilor până la augmentare și export. În cadrul acestui proiect, Roboflow a fost folosit pentru a eticheta imaginile care urmau să fie oferite modelelor pentru învățare.

5 Implementarea motorului de inferențe pentru jocul de șah

Această secțiune descrie implementarea motorului de inferență pentru jocul de șah. Pentru implementarea algoritmului Minimax, a fost necesară dezvoltarea unei metode pentru generarea tuturor mutărilor posibile, astfel încât algoritmul să poată construi graful de căutare. De asemenea, o funcție esențială pentru algoritm este funcția de evaluare a unei poziții, care în varianta inițială folosește o abordare simplă, acordând un scor fiecărei poziții pe baza pieselor prezente pe tablă.

Pentru a îmbunătăți calitatea mutărilor generate, a fost implementată o variantă optimizată a algoritmului folosind metoda Alpha-Beta, iar funcția de evaluare a pozițiilor a fost înlocuită. Noua funcție de evaluare se bazează pe predicțiile unui model neuronal, antrenat pe date etichetate de motorul Stockfish.

După prezentarea succintă a componentelor codului sursă, capitolul detaliază implementarea fiecărei etape.

5.1 Structura generală a componentelor

5.1.1 Clasa GameState

Clasa `GameState` este concepută pentru a gestiona starea unei partide de șah. Aceasta păstrează starea curentă a tablei de joc, jucătorul care urmează să facă o mutare și culoarea pieselor acestuia (alb sau negru). În plus, clasa conține funcții pentru evaluarea unei poziții și generarea tuturor mutărilor posibile. Aceste funcții sunt utilizate în metoda care decide mutarea ce trebuie efectuată.

Parametrii clasei GameState

- `self.board`: String, inițial FEN pentru tabla de șah de start.
- `self.white_to_move`: Boolean, indică dacă este rândul jucătorului alb să mute.
- `self.is_playing_Black`: Boolean, indică dacă utilizatorul joacă cu piesele negre.

Metodele clasei GameState

- `evaluate_fen(fen)`: Evaluează poziția pe tabla de șah folosind valori predefinite pentru piese.
- `generate_future_positions(fen)`: Generează toate pozițiile viitoare posibile și mutările legale având ca date de intrare un sir FEN.

- `minimax(fen, depth, maximizing_player=True)`: Implementarea algoritmului minimax pentru a determina cea mai bună mutare până la o anumită adâncime.
- `minimax_alpha_beta(fen, depth, alpha=float('-inf'), beta=float('inf'), maximizing_player=True, is_playing_Black=True)`: Implementarea algoritmului minimax cu optimizarea alfa-beta pentru eficientizarea evaluării mutărilor.
- `model_alpha_beta(fen, depth, alpha=float('-inf'), beta=float('inf'), maximizing_player=True, model=None, is_playing_Black=True)`: Implementarea algoritmului minimax cu optimizarea alfa-beta, utilizând un model de învățare automată pentru evaluarea poziției.
- `evaluate_board(fen, model)`: Evaluează poziția pe tabla de șah folosind un model de învățare automată.

5.1.2 Clasa ChessGame

Clasa `ChessGame` implementează o interfață grafică pentru un joc de șah (a se vedea figura 5.1), utilizând biblioteca `tkinter`. Această clasă este esențială pentru testarea algoritmilor de șah dezvoltăți, oferind un mediu interactiv în care jucătorul uman poate juca împotriva motorului de șah.

Parametrii Clasei `ChessGame`

- `gs`: Instanțiere a clasei `GameState`, responsabilă pentru stocarea stării curente a jocului.
- `boardImage`: Imaginea tablei de șah.
- `last_move_highlighting`: Highlighting pentru ultima mutare efectuată pe tabla de șah.
- `drag_data`: Datele necesare pentru implementarea funcționalității de drag-and-drop pentru mutarea pieselor.
- `root`: Fereastra principală `tkinter`.

Metodele Clasei `ChessGame`

- `create_ui(self)`: Creează interfața grafică, afișează tabla de șah și asociază evenimentele de mouse pentru interacțiunea cu piesele.
- `on_drag_start(self, event)`: Initializează procesul de drag-and-drop când jucătorul apasă pe o piesă.
- `on_drag_end(self, event)`: Finalizează procesul de drag-and-drop, validând și efectuând mutarea dacă este legală.
- `game_loop(self)`: Bucla principală a jocului care se execută până când jocul se încheie.
- `play_black_turn(self, model)`: Funcția care gestionează mutarea pieselor negre.
- `update_board(self, move)`: Actualizează imaginea tablei de șah și evidențiază ultima mutare efectuată.

- `make_black_move_minimax(self)`: Realizează mutarea pieselor negre utilizând algoritmul Minimax pentru determinarea celei mai bune mutări.

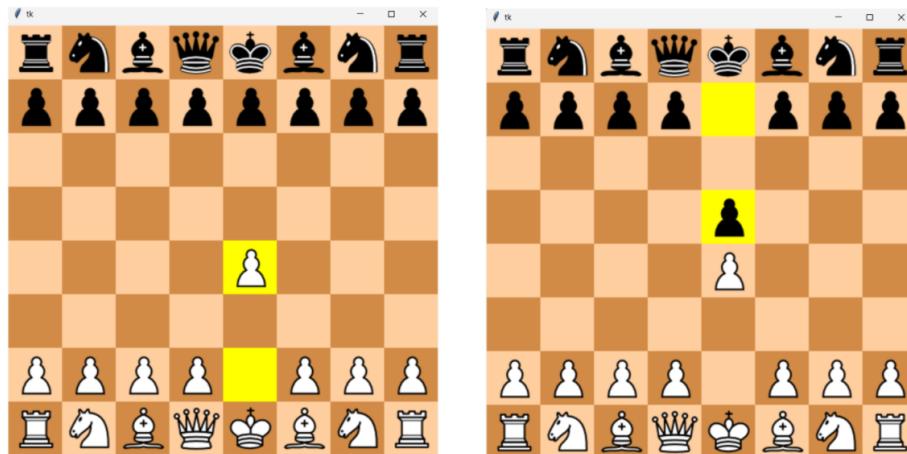


Figura 5.1: Interfața grafică a motorului de inferență pentru jocul de șah

5.1.3 Clasa Net

Clasa `Net` este responsabilă pentru definirea arhitecturii rețelei neuronale utilizate în motorul de inferență pentru jocul de șah. Aceasta include straturi convoluționale pentru extragerea caracteristicilor și straturi complet conectate pentru realizarea predicțiilor. Clasa conține metode pentru calcularea dimensiunii ieșirii din straturile convoluționale și pentru propagarea înainte a datelor prin rețea, aplicând activări ReLU și pooling maxim.

5.1.4 Scriptul de antrenare al rețelei neuronale de evaluare a unei poziții de șah

Metode ale scriptului

- `import_chess_data(file_path, skip_rows=0, num_rows=None)`: Importează datele de șah dintr-un fișier CSV.
- `preprocess_data2(data)`: Preprocesează datele, extrage și validează pozițiile FEN și evaluările, filtrând valorile nevalide.
- `quantile_transform_evaluation(evaluation_data)`: Aplică o transformare de cuantile asupra datelor de evaluare pentru a uniformiza distribuția acestora.
- `bin_evaluation_data(evaluation_data, num_bins=10)`: Împarte datele de evaluare în categorii discrete, utile pentru stratificare.
- `ChessDataset(Dataset)`: Clasă personalizată pentru gestionarea dataset-ului de șah, care returnează tensorii necesari pentru antrenare.
- `plot_histograms(original_data, transformed_data, output_dir)`: Plotează histogramele datelor.
- `plot_learning_curves(train_losses_mse, val_losses_mse, optimizer, num_epochs, batch_size, train_size, val_size, output_dir)`: Plotează curbele de învățare pentru pierderile de antrenament și de validare.

- `evaluate_model(model, criterion, data_loader, device)`: Evaluatează modelul pe un set de date.
- `train_model(model, scheduler, criterion, optimizer, train_loader, val_loader, device, batch_size, train_size, val_size, output_dir, num_epochs=10, patience=5)`: Antrenează modelul pe setul de date primit ca parametru.

5.2 Implementarea algoritmului Minimax

Plecând de la pseudocodul prezentat în figura 5.2 a fost implementat în Python algoritmul minimax pentru jocul de șah.

```
Minimax(board, player) :
    if board is a leaf node :
        return the value of board
    if player = 1 :
        best = -∞
        for each child of board :
            val = minimax(child, -1)
            best = max(val, best)
        return best
    else :
        best = ∞
        for each child of board :
            val = minimax(child, -1)
            best = min(val, best)
        return best
```

Figura 5.2: Pseudocod al algoritmului Minimax [27]

Codul următor ilustrează această implementare în cadrul funcției `minimax(fen, depth, maximizing_player=True)`:

```
1 def minimax(fen, depth, maximizing_player=True):
2     if depth == 0:
3         return fen, GameState.evaluate_fen(fen)
4
5     if maximizing_player:
6         best_move = None
7         best_value = float('-inf')
8
9         for future_fen in GameState.generate_future_positions(fen):
10             _, value = GameState.minimax(future_fen, depth - 1, False)
11             if value > best_value:
12                 best_value = value
13                 best_move = future_fen
```

```

15     return best_move, best_value
16 else:
17     best_move = None
18     best_value = float('inf')
19
20     for future_fen in GameState.generate_future_positions(fen):
21         _, value = GameState.minimax(future_fen, depth - 1, True)
22         if value < best_value:
23             best_value = value
24             best_move = future_fen
25
26     return best_move, best_value

```

- `minimax(fen, depth, maximizing_player=True)`: Antetul algoritmului Minimax. Primește ca parametri starea curentă a jocului în format FEN, adâncimea căutării (`depth`) și un flag (`maximizing_player`) care indică dacă jucătorul curent este cel care maximizează valoarea sau cel care minimizează valoarea.
- Dacă adâncimea căutării (`depth`) este 0, funcția returnează evaluarea poziției curente folosind funcția `GameState.evaluate_fen(fen)`.
- Pentru jucătorul care maximizează valoarea (`maximizing_player=True`), algoritmul initializează `best_value` cu `float('-inf')` și iterativ evaluează toate pozițiile viitoare generate de funcția `GameState.generate_future_positions(fen)`. Dacă valoarea unei poziții viitoare este mai mare decât `best_value`, algoritmul actualizează `best_value` și `best_move`.
- Pentru jucătorul care minimizează valoarea (`maximizing_player=False`), algoritmul initializează `best_value` cu `float('inf')` și similar evaluează toate pozițiile viitoare. Dacă valoarea unei poziții viitoare este mai mică decât `best_value`, algoritmul actualizează `best_value` și `best_move`.
- Funcția `minimax` se apelează recursiv pentru fiecare poziție viitoare generată, scăzând adâncimea căutării cu 1 la fiecare apel recursiv. Acest proces continuă până când adâncimea ajunge la 0, moment în care funcția returnează evaluarea poziției curente.
- În final, funcția returnează cea mai bună mutare (`best_move`) și valoarea asociată acesteia (`best_value`).

5.3 Implementarea functiei de generare a mutarilor legale

```

1 @staticmethod
2 def generate_future_positions(fen):
3     board = chess.Board(fen)
4     future_positions = []
5
6     legal_moves = list(board.legal_moves)
7     for move in legal_moves:
8         board_copy = board.copy()
9         board_copy.push(move)
10        future_positions.append((board_copy.fen(), move))
11
12    return future_positions

```

Acest cod implementează funcția de generare a mutărilor legale și include următoarele componente principale:

- `generate_future_positions(fen)`: Funcția este definită ca o metodă statică, ceea ce înseamnă că poate fi apelată fără a crea o instanță a clasei. Primește ca parametru starea curentă a jocului în format FEN (`fen`).
- `board = chess.Board(fen)`: Creează un obiect `chess.Board` folosind librăria Python-Chess, inițializând tabla de șah cu poziția specificată de șirul FEN.
- `legal_moves = list(board.legal_moves)`: Obține o listă cu toate mutările legale disponibile din poziția curentă.
- `for move in legal_moves`: Iterează prin fiecare mutare legală disponibilă.
- `board_copy = board.copy()`: Creează o copie a tablei de șah curente pentru a nu modifica starea originală a jocului.
- `board_copy.push(move)`: Aplică mutarea curentă pe copia tablei de șah.
- `future_positions.append((board_copy.fen(), move))`: Adaugă la lista de poziții viitoare o pereche formată din starea FEN a tablei după efectuarea mutării și mutarea respectivă.
- `return future_positions`: Returnează lista de poziții viitoare posibile.

5.4 Implementarea funcției de evaluare bazată pe piesele prezente pe tablă

Funcția `evaluate_fen` este utilizată pentru a evalua o poziție pe tabla de șah reprezentată în format FEN. Scorul poziției este calculat pe baza valorilor pieselor, a rândului curent de mutare și a posibilităților de rocada. În continuare, vom detalia modul în care funcționează această funcție și fiecare parte a implementării sale.

```

1 def evaluate_fen(fen):
2     piece_values = {
3         'P': 10, 'N': 30, 'B': 30, 'R': 50, 'Q': 90, 'K': 0,
4         'p': -10, 'n': -30, 'b': -30, 'r': -50, 'q': -90, 'k': 0
5     }
6
7     parts = fen.split()
8
9     if len(parts) != 6:
10        return 0 # Invalid FEN format
11
12     board_layout, turn, castling, en_passant, half_moves,
13     full_moves = parts
14
15     board_score = 0
16
17     for char in board_layout:
18         piece_value = piece_values.get(char, 0)
19         board_score += piece_value
20
21     if turn == 'w':
22         board_score -= 20
23     else:
24         board_score += 20
25
26     if 'K' in castling:

```

```

27     board_score -= 10
28     if 'k' in castling:
29         board_score += 10
30
31     min_score = -540
32     max_score = 540
33
34     normalized_score = (board_score - min_score) / (max_score - min_score)
35     return normalized_score

```

- `piece_values`: Un dicționar care asociază fiecărei piese o valoare numerică. Valorile pieselor albe sunt pozitive, iar valorile pieselor negre sunt negative.
- `parts = fen.split()`: Desparte șirul FEN în componente sale individuale. Un sir FEN valid conține șase părți.
- `if len(parts) != 6`: Verifică dacă formatul FEN este valid. Dacă nu, funcția returnează un scor de 0.
- `board_layout, turn, castling, en_passant, half_moves, full_moves = parts`: Descompune șirul FEN în componente sale: dispunerea pieselor pe tablă, jucătorul la mutare, drepturile de rocadă, captura en passant, numărul de mutări fără captură sau mutare de pion și numărul total de mutări.
- `board_score = 0`: Inițializează scorul poziției.
- `for char in board_layout`: Iterează prin fiecare caracter din dispunerea pieselor pe tablă și adaugă valoarea asociată fiecărei piese la scorul total al poziției.
- `if 'k' in castling`: Adaugă un bonus de scor pentru dreptul la rocadă al jucătorului negru.
- `if 'K' in castling`: Scade un punctaj pentru dreptul la rocadă al jucătorului alb.
- `min_score = -540, max_score = 540`: Definește limitele scorului posibil pentru normalizare.
- `normalized_score = (board_score - min_score) / (max_score - min_score)`: Normalizează scorul poziției într-un interval de la 0 la 1.
- `return normalized_score`: Returnează scorul normalizat al poziției.

5.5 Implementarea optimizării Alpha-Beta

Figura 5.3 prezintă pseudocodul optimizării Alpha-Beta. Diferențele dintre algoritmul de bază și cel optimizat sunt evidențiate în imagine în galben.

```

Minimaxab(board, player, alpha, beta) :
    if board is a leaf node :
        return the value of board
    if player = 1 :
        best = -∞
        for each child of board :
            val = minimaxab(child, -1, alpha, beta)
            best = max(val, best)
            alpha = max(best, alpha)
            if alpha ≥ beta :
                break
        return best
    else :
        best = ∞
        for each child of board :
            val = minimaxab(child, -1)
            best = min(val, best)
            beta = min(best, beta)
            if alpha ≥ beta :
                break
        return best

```

Figura 5.3: Pseudocod al optimizării Alpha-Beta [27]

```

1 def minimax_alpha_beta(fen, depth, alpha=float('-inf'), beta=float('inf'),
2 maximizing_player=True, is_playing_Black=True):
3     if depth == 0 or GameState.generate_future_positions(fen) == []:
4         score = GameState.evaluate_fen(fen)
5
6         if is_playing_Black:
7             score = -score
8         return fen, None, score
9
10    if maximizing_player:
11        best_move = None
12        best_value = float('-inf')
13
14        for future_fen, move in GameState.generate_future_positions(fen):
15            _, _, value = GameState.minimax_alpha_beta(future_fen,
16            depth - 1, alpha, beta, False, is_playing_Black)
17            if value > best_value:
18                best_value = value
19                best_move = move
20
21            alpha = max(alpha, best_value)
22            if beta <= alpha:
23                break # Pruning
24        return fen, best_move, best_value
25    else:
26        best_move = None
27        best_value = float('inf')
28
29        for future_fen, move in GameState.generate_future_positions(fen):
30            _, _, value = GameState.minimax_alpha_beta(future_fen,
31            depth - 1, alpha, beta, True, is_playing_Black)
32            if value < best_value:
33                best_value = value
34                best_move = move

```

```

35         beta = min(beta, best_value)
36         if beta <= alpha:
37             break # Pruning
38     return fen, best_move, best_value

```

Acest cod include următoarele componente suplimentare față de versiunea de bază a algoritmului Minimax discutată anterior:

- **alpha și beta**: Parametrii care definesc limitele intervalului în care valorile nodurilor sunt permise. **alpha** reprezintă cel mai bun scor pe care jucătorul care maximizează îl poate garanta până acum, iar **beta** reprezintă cel mai bun scor pe care jucătorul care minimizează îl poate garanta.
- **if beta <= alpha**: Condiție care verifică dacă intervalul a devenit invalid și determină oprirea căutării (pruning).

5.6 Implementarea unui model neuronal pentru evaluarea unei poziții de șah

Pentru a îmbunătăți capacitatea motorului de inferență de a lua decizii tactice mai precise, este esențială perfecționarea metodei de evaluare a pozițiilor de șah. Astfel, a fost dezvoltat un model de rețea neuronală convezională (CNN) pentru a recunoaște tipare complexe și pentru a interpreta structura matriceală a tablei de șah. Modelul a fost antrenat cu poziții de șah în format FEN etichetate de motorul Stockfish.

5.6.1 Arhitectura modelului

Articolul [28] prezintă diverse arhitecturi de CNN-uri folosite pentru estimarea avantajului într-o poziție de șah. Arhitectura utilizată în acest proiect se inspiră din "Model2", care constă în 5 straturi de convezie urmate de straturi de pooling și 3 straturi complet conectate. Cu toate acestea, arhitectura a fost simplificată pentru a evita suprainvățarea datorită dimensiunii setului de date utilizat.

Modelul propus constă din trei straturi convezionale urmate de straturi de pooling și, în final, două straturi complet conectate. Arhitectura este prezentată în figura 5.4, imagine generată pe baza codului rețelei utilizând site-ul [29]. Structura detaliată este următoarea:

- **Straturi convezionale:**
 - *Primul strat convezional*: Primește 13 canale corespunzătoare tablei de șah și aplică 32 de filtre convezionale de 3x3, cu padding de 1 pentru a menține dimensiunile inițiale ale imaginii.
 - *Al doilea strat convezional*: Aplică 64 de filtre de 3x3 asupra outputului primului strat.
 - *Al treilea strat convezional*: Utilizează 128 de filtre de 3x3.
- **Straturi de pooling:**

- După fiecare strat convecțional se aplică un strat de max pooling de 2x2, reducând dimensiunile spațiale, prevenind astfel suprainvățarea.
- **Straturi complet conectate:**

- *Primul strat complet conectat:* După aplativarea rezultatului din ultimul strat de pooling, acesta este conectat la un strat dens cu 128 de neuroni și aplică funcția de activare ReLU.
- *Al doilea strat complet conectat:* Strat de ieșire care produce o evaluare scalară a poziției de șah.

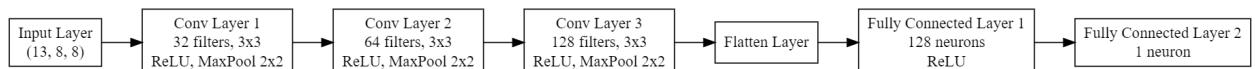


Figura 5.4: Arhitectura modelului

5.6.2 Baza de date folosită în cadrul proiectului

Pentru antrenarea și evaluarea acestui model a fost utilizată baza de date `chessData.csv` [30]. Aceasta conține peste 12.000.000 de FEN-uri diferite, etichetate de Stockfish 11.

5.6.3 Preprocesarea FEN-urilor

O metodă comună pentru prelucrarea FEN-urilor încruntă un format care poate fi utilizat de o rețea neuronală este transformarea acestora încruntă un vector de biți. Funcția descrisă mai jos, preluată din sursa [31], transformă o poziție FEN încruntă un set de 13 matrici. Primele 12 matrici reprezintă locația fiecarei piese de șah pe tablă, iar ultima matrice conține informații relevante despre poziție, cum ar fi numărul de halfmoves, posibilitățile de rocadă etc.

Funcția `fen_to_bit_vector`:

1. Împarte șirul FEN încruntă componentă, inclusiv plasarea pieselor, culoarea jucătorului activ, drepturile de rocadă, pătratul en passant, ceasul mutărilor și numărul total de mutări.
2. Creează un vector de biți tridimensional de dimensiuni (13, 8, 8) pentru a reprezenta prezența pieselor pe tablă și alte informații relevante.
3. Mapează piesele la straturi specifice încruntă vectorul de biți folosind un dicționar.
4. Parcurge plasarea pieselor și actualizează vectorul de biți corespunzător.
5. Gestionează drepturile de rocadă și pătratul en passant, actualizând vectorul de biți încruntă consecință.
6. Adaugă informații despre culoarea jucătorului activ și despre ceasurile mutărilor la vectorul de biți.

Funcția returnează vectorul de biți rezultat, care poate fi utilizat ulterior pentru antrenarea modelului CNN.

5.6.4 Prelucrarea etichetelor

A doua coloană în baza de date este reprezentată de evaluările pozițiilor returnate de Stockfish. Aceste evaluările pot varia în intervalul [-10000, 10000]. În cazul în care Stockfish consideră că un șah poate fi forțat în următoarele mutări, în loc de o evaluare numerică a poziției, acesta va eticheta poziția în formatul "#x", unde x reprezintă numărul de mutări în care șahul poate fi forțat.

```
1 def preprocess_data2(data):
2     fen_data = data['FEN']
3     evaluation_data = data['Evaluation']
4
5     valid_indices = []
6     processed_evaluation = []
7     for idx, value in enumerate(evaluation_data):
8         if isinstance(value, str) and value.startswith('#'):
9             continue
10    try:
11        int_value = int(value)
12        if int_value != 0:
13            processed_evaluation.append(int_value)
14            valid_indices.append(idx)
15        except ValueError:
16            continue
17
18    processed_evaluation_series = pd.Series(processed_evaluation)
19
20    valid_fen_data = fen_data.iloc[valid_indices].reset_index
21    processed_evaluation_series = processed_evaluation_series.reset_index
22
23    return valid_fen_data, processed_evaluation_series
24
25 def quantile_transform_evaluation(evaluation_data):
26     transformer = QuantileTransformer(output_distribution='uniform')
27     transformed_data =
28     transformer.fit_transform(evaluation_data.values.reshape(-1, 1))
29     return transformed_data
```

Funcția `preprocess_data2` realizează următoarele operațiuni:

1. Extrage datele FEN și evaluările pozițiilor din dataframe-ul de intrare.
2. Parurge evaluările și filtrează pozițiile cu evaluări valide, excludând cele marcate ca șah imminent (formatul "#x").
3. Converteste evaluările valide în valori întregi și le adaugă la lista `processed_evaluation` împreună cu indicii corespunzători.
4. Creează serii pandas pentru evaluările procesate și datele FEN corespunzătoare.
5. Returnează datele FEN și evaluările procesate.

Funcția `quantile_transform_evaluation` transformă datele de evaluare pentru a le distribui uniform:

1. Creează un transformator de cuantile cu distribuție uniformă.
2. Aplică transformarea asupra datelor de evaluare, convertindu-le într-o distribuție uniformă între 0 și 1.

3. Returnează datele transformate, gata pentru a fi utilizate în antrenarea modelului.

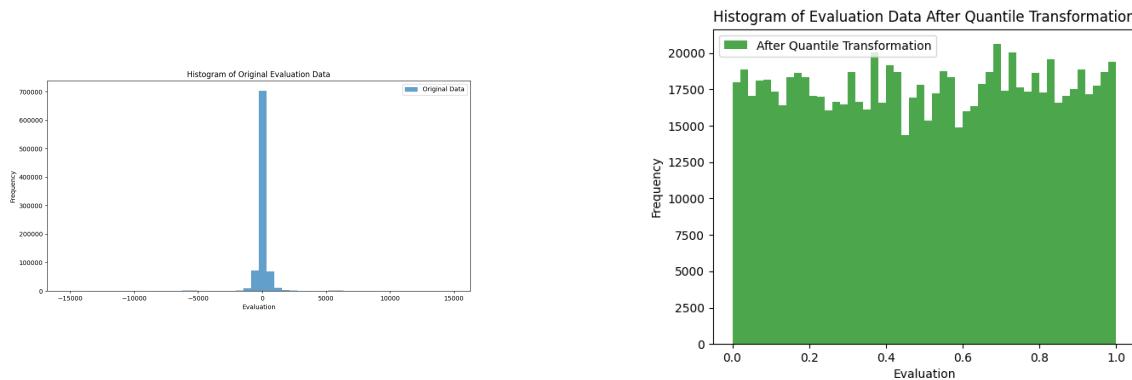


Figura 5.5: Compararea histogramelor înainte și după preprocesare

Figura 5.5 prezintă rezultatele preprocesării datelor. După aplicarea transformării de cuantile, distribuția datelor a devenit uniformă în interval, comparativ cu histograma inițială unde majoritatea valorilor erau concentrate în centru. Această uniformizare a fost efectuată pentru a îmbunătăți capacitatea modelului de a diferenția între diverse poziții de șah, facilitând astfel antrenarea acestuia.

5.6.5 Antrenarea modelului

Funcția `train_model` este responsabilă pentru antrenarea modelului de rețea neuronală, gestionarea optimizării, și evaluarea performanței acestuia pe seturile de date de antrenament și validare. Aceasta primește următorii parametri:

- `model`: Modelul de rețea neuronală care urmează să fie antrenat.
- `scheduler`: Scheduler-ul pentru ajustarea ratei de învățare în timpul antrenamentului.
- `criterion`: Funcția de pierdere utilizată pentru a calcula eroarea.
- `optimizer`: Algoritmul de optimizare utilizat pentru a actualiza parametrii modelului.
- `train_loader`: DataLoader pentru setul de date de antrenament.
- `val_loader`: DataLoader pentru setul de date de validare.
- `device`: Dispozitivul pe care se execută antrenamentul (CPU sau GPU).
- `batch_size`: Dimensiunea lotului de date.
- `train_size`: Dimensiunea setului de date de antrenament.
- `val_size`: Dimensiunea setului de date de validare.
- `output_dir`: Directorul în care se vor salva rezultatele.
- `num_epochs`: Numărul de epoci pentru antrenament (implicit 10).
- `patience`: Numărul de epoci fără îmbunătățiri pentru oprirea anticipată (implicit 5).

```
1 def train_model(model, scheduler, criterion, optimizer, train_loader,
2 val_loader, device, batch_size, train_size, val_size, output_dir,
3 num_epochs=10, patience=5):
4     train_losses_mse = []
5     val_losses_mse = []
6     best_val_loss = float('inf')
7     epochs_no_improve = 0
8     early_stop = False
9
10    for epoch in range(num_epochs):
11        if early_stop:
12            print("Early stopping")
13            break
14
15        model.train()
16        for inputs, labels in train_loader:
17            inputs, labels = inputs.to(device), labels.to(device)
18            optimizer.zero_grad()
19            outputs = model(inputs)
20            loss_mse = criterion(outputs, labels.float().unsqueeze(1))
21            loss_mse.backward()
22            optimizer.step()
23
24        train_loss_mse, _, _ = evaluate_model(model, criterion,
25 train_loader, device)
26        train_losses_mse.append(train_loss_mse)
27
28        val_loss_mse, _, _ = evaluate_model(model, criterion, val_loader,
29 device)
29        val_losses_mse.append(val_loss_mse)
30
31        scheduler.step(val_loss_mse)
32
33        print(f"Epoch {epoch + 1}/{num_epochs}, Train Loss (MSE): {train_loss_mse:.4f}, Validation Loss (MSE): {val_loss_mse:.4f}")
34
35        if val_loss_mse < best_val_loss:
36            best_val_loss = val_loss_mse
37            epochs_no_improve = 0
38        else:
39            epochs_no_improve += 1
40
41        if epochs_no_improve >= patience:
42            early_stop = True
43
44    plot_learning_curves(train_losses_mse, val_losses_mse, optimizer,
45 num_epochs, batch_size, train_size, val_size, output_dir)
```

Funcția `train_model` realizează următoarele operațiuni:

1. Inițializează liste pentru pierderile de antrenament și validare.
2. Iterează prin fiecare epocă de antrenament, oprindu-se anticipat dacă performanța nu se îmbunătățește.
3. În faza de antrenament, pentru fiecare lot de date, transferă datele pe dispozitivul specificat, execută propagarea înainte și înapoi, și actualizează parametrii modelului.
4. Evaluatează modelul pe seturile de date de antrenament și validare, stocând pierderile calculate.

5. Ajustează rata de învățare utilizând scheduler-ul.
6. Verifică condiția de oprire anticipată pe baza pierderii de validare.
7. Plotează curbele de învățare pentru pierderile de antrenament și validare.

5.6.6 Evaluarea modelului

Funcția `evaluate_model` este responsabilă pentru evaluarea performanței modelului pe un set de date dat. Aceasta calculează pierderea totală și colectează predicțiile și etichetele reale pentru a evalua acuratețea modelului. Funcția primește următorii parametri:

- `model`: Modelul de rețea neuronală care urmează să fie evaluat.
- `criterion`: Funcția de pierdere utilizată pentru a calcula eroarea.
- `data_loader`: DataLoader pentru setul de date utilizat în evaluare.
- `device`: Dispozitivul pe care se execută evaluarea (CPU sau GPU).

```

1 def evaluate_model(model, criterion, data_loader, device):
2     model.eval()
3     running_loss = 0.0
4     all_preds = []
5     all_targets = []
6     with torch.no_grad():
7         for inputs, targets in data_loader:
8             inputs, targets = inputs.to(device), targets.to(device)
9             outputs = model(inputs)
10            loss = criterion(outputs, targets.unsqueeze(1))
11            running_loss += loss.item() * inputs.size(0)
12            all_preds.extend(outputs.cpu().numpy())
13            all_targets.extend(targets.cpu().numpy())
14    epoch_loss = running_loss / len(data_loader.dataset)
15    return epoch_loss, all_preds, all_targets

```

Funcția `evaluate_model` realizează următoarele operațiuni:

1. Setează modelul în modul de evaluare folosind `model.eval()`
2. Inițializează variabile pentru a stoca pierderea totală (`running_loss`), predicțiile (`all_preds`) și etichetele (`all_targets`).
3. Parcurge loturile de date din `data_loader`:
 - Transferă datele de intrare și etichetele pe dispozitivul specificat (`inputs.to(device)` și `targets.to(device)`).
 - Obține predicțiile modelului și calculează pierderea folosind funcția de pierdere (`criterion`).
 - Actualizează pierderea totală ponderată (`running_loss`).
4. Calculează pierderea medie pe epocă (`epoch_loss`) împărțind pierderea totală la numărul total de exemple din setul de date.
5. Returnează pierderea medie pe epocă, predicțiile și etichetele reale.

5.6.7 Procesul de antrenare și evaluare al rețelei neuronale

Funcția `main` este responsabilă pentru gestionarea întregului proces de antrenare și evaluare a modelului. Aceasta parcurge următorii pași:

1. Crearea directorului `runs` pentru stocarea rezultatelor, dacă acesta nu există.
2. Determinarea directorului de ieșire pentru rularea curentă, creând un director nou pentru fiecare rulare.
3. Setarea dispozitivului pe care se va realiza antrenamentul (`cuda` dacă este disponibil, altfel `cpu`).
4. Importarea și preprocesarea datelor de șah folosind funcțiile `import_chess_data` și `preprocess_data2`.
5. Aplicarea transformării de cuantile asupra datelor de evaluare pentru a obține o distribuție uniformă.
6. Plotarea histogramelor datelor originale și transformate și salvarea acestora în directorul de ieșire.
7. Crearea unui nou DataFrame cu datele FEN valide și evaluările transformate.
8. Împărțirea setului de date în seturi de antrenament, validare și testare folosind `StratifiedShuffleSplit` pentru a asigura o distribuție uniformă.
9. Crearea obiectelor `DataLoader` pentru seturile de date de antrenament, validare și testare.
10. Instantierea modelului de rețea neuronală (Net), a funcției de pierdere (`nn.MSELoss`), a optimizer-ului (`torch.optim.Adam`) și a scheduler-ului (`torch.optim.lr_scheduler.ReduceLROnPlateau`).
11. Antrenarea modelului folosind funcția `train_model`
12. Salvarea modelului antrenat în directorul de ieșire.
13. Evaluarea modelului pe setul de date de testare și crearea unei matrice de confuzie pentru a analiza performanța modelului.

5.7 Funcția de evaluare a unei poziții folosind modelul CNN

Funcția `evaluate_board` utilizează modelul CNN pentru a evalua o poziție de șah reprezentată printr-un sir FEN. Aceasta parcurge următorii pași:

```
1 def evaluate_board(fen, model):  
2     input_tensor = torch.tensor(fen_to_bit_vector(fen), dtype=torch.float32)  
3     input_tensor = input_tensor.to(next(model.parameters()).device)  
4     input_tensor = input_tensor.unsqueeze(0)  
5     with torch.no_grad():  
6         output = model(input_tensor)  
7     value = output.item()  
8  
9     return value
```

1. Convertește FEN-ul într-un tensor folosind funcția `fen_to_bit_vector`.
2. Transmite tensorul de intrare prin model pentru a obține predictia.
3. Convertește ieșirea modelului într-o valoare scalară.
4. Returnează valoarea scalară ca evaluare a poziției de șah.

5.8 Rezultate ale motorului de inferențe

5.8.1 Compararea timpilor de execuție ai algoritmilor Alpha-Beta și Minimax

În această secțiune, am comparat algoritmul Minimax cu optimizarea Alpha-Beta Pruning. Pentru a evalua impactul optimizării asupra timpilor de calcul, am desfășurat un experiment în care ambii algoritmi au fost testați pe aceeași succesiune de poziții aleatorii. Pentru fiecare dintre aceste poziții, am monitorizat timpul necesar fiecărui algoritm pentru a decide asupra celei mai bune mutări. La final, timpii de calcul au fost plotați pentru o analiză comparativă.

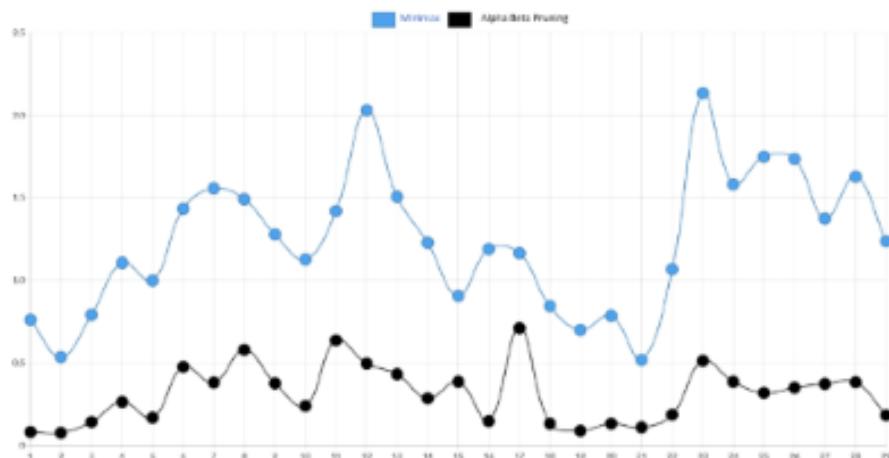


Figura 5.6: Timpi de executie Minimax si Alpha-Beta Pruning

În figura 5.6, axa X indică succesiunea pozițiilor evaluate de cei doi algoritmi, iar axa Y reprezintă timpul necesar fiecărui algoritm pentru a lua o decizie. Graficul albastru ilustrează timpii de execuție pentru algoritmul Minimax, în timp ce graficul negru reprezintă timpii de execuție pentru algoritmul Alpha-Beta Pruning.

Rezultatele au arătat că Alpha-Beta Pruning a returnat întotdeauna rezultatul mai rapid decât algoritmul Minimax, ceea ce era de așteptat. Cu toate acestea, nu s-a putut trage o concluzie clară privind reducerea exactă a timpului de calcul, deoarece ordinea în care sunt generate nodurile (pozițiile legale) influențează numărul de noduri care vor fi tăiate. Generarea nodurilor în ordine crescătoare în funcție de evaluarea pozitională pe ramura de maximizare ar duce la tăierea unui număr maxim de noduri, în timp ce ordinea descrescătoare ar duce la tăierea minimă.

5.8.2 Rezultatele de antrenament ale modelului CNN

Graficul prezentat în figura 5.7 arată curbele de învățare ale pierderii medii patratice pentru seturile de antrenament și de validare pe parcursul a 50 de epoci. Atât pierderea de antrenament

(linia albastră), cât și pierderea de validare (linia portocalie) scad constant, indicând faptul că modelul se îmbunătățește treptat pe măsură ce învăță din datele de antrenament.

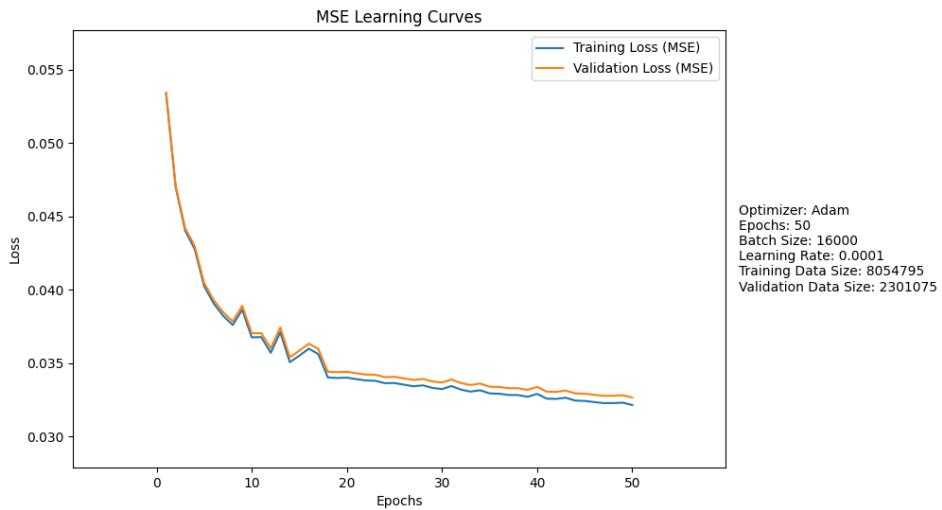


Figura 5.7: Curbele de invatare returnate la finalul antrenarii

Diferența mică între pierderea de antrenament și cea de validare sugerează că modelul se comportă similar atât pe datele de antrenament, cât și pe cele de validare, un semn pozitiv care indică absența supraînvățării. În final, valorile MSE se stabilizează în jurul valorii de 0.035.

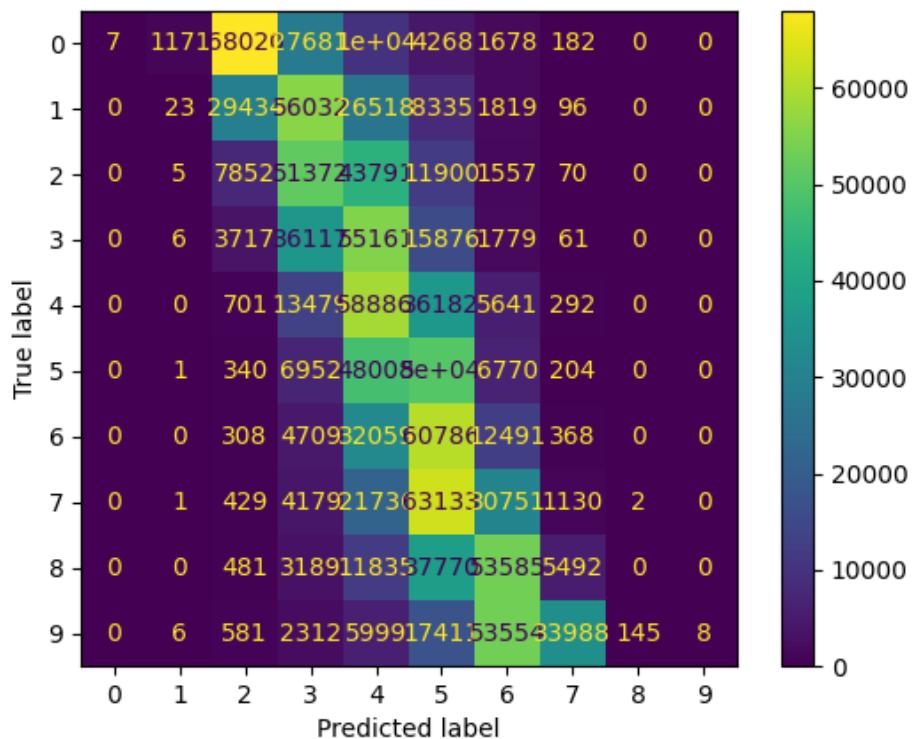


Figura 5.8: Matricea de confuzie generată pe datele de testare

Figura 5.8 prezintă matricea de confuzie generată după antrenarea modelului pe datele

de testare, care sunt date noi pentru model. Cele două axe ale matricei reprezintă 10 intervale egale (bin-uri) în care sunt împărțite valorile estimărilor. Pe axa Y sunt plasate etichetele reale, iar pe axa X sunt plasate valorile estimate de model. Într-un scenariu ideal, toate valorile ar trebui să apară doar pe diagonala principală a matricei, ceea ce ar sugera că pentru fiecare FEN, intervalul în care se află eticheta reală coincide cu intervalul în care a fost estimată valoarea. Aceasta ar indica o precizie perfectă a modelului în estimările sale.

Se poate observa că, pentru intervalele din zona de mijloc, între valorile 2-7, deși există anumite abateri, majoritatea valorilor sunt plasate corect. În schimb, pentru valorile din intervalele 0, 1, 8 și 9, acest comportament nu se aplică. Modelul evită să atribuie valori în aceste intervale.

Motivul pentru care modelul se comportă astfel este diferența mare dintre numărul de evaluări din baza de date pentru pozițiile uzuale din timpul jocului (care aparțin intervalelor 2-7) și numărul redus de poziții aferente intervalelor extreme (0, 1, 8, 9), asociate cu scoruri foarte ridicate sau foarte scăzute. Acest dezechilibru cantitativ apare deoarece baza de date a fost creată prin segmentarea unor jocuri complete de șah, unde pozițiile critice de șah mat pentru alb sau negru sunt rare, în contrast cu multitudinea de mutări din jocurile obișnuite. Din cauza acestui număr limitat de exemple în momentele critice, modelul nu reușește să învețe eficient caracteristicile care determină dacă o poziție este cu adevărat avantajoasă sau dezavantajoasă, afectându-i astfel performanța în etapa finală a jocului.

6 Implementarea sistemului de vedere artificială

În acest capitol este prezentată implementarea unui sistem de vedere artificială capabil să recunoască tabla de șah, să determine coordonatele precise ale colțurilor, să identifice poziția pieselor și să stabilească dacă este rândul robotului sau al jucătorului uman să efectueze o mutare.

Pentru a simplifica procesul de recunoaștere a pieselor și a asigura detectarea precisă evitând eventualele erori, nu a fost utilizat un set de piese tradițională. În schimb, s-a folosit o imprimantă 3D pentru a crea un set de piese personalizate (a se vedea figura 6.1). Fiecare piesă este formată dintr-o bază, care facilitează prinderea pieselor de către robot, iar deasupra bazei este imprimat un model similar cu cele întâlnite în interfețele clasice de șah. Această abordare simplifică clasificarea pieselor în categorii.

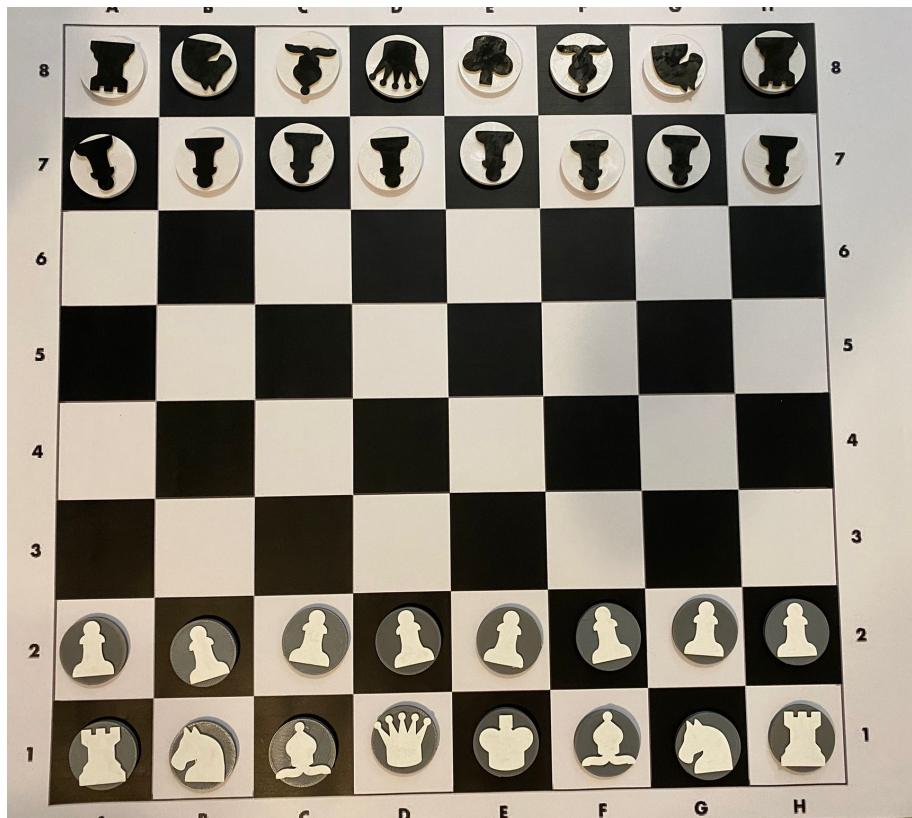


Figura 6.1: Piese de șah folosite în cadrul proiectului

6.1 Detectarea tablei de șah

Având în vedere că o tablă de șah tradițională este împărțită într-o rețea de 8x8 pătrate de dimensiuni egale, detectarea acesteia se reduce la identificarea precisă a colțurilor. În acest sens, a fost antrenat un model YOLOv8 utilizând un set de date conținând imagini în care cele patru colțuri ale tablei de șah au fost etichetate corespunzător.

6.1.1 Datele de antrenament

Pentru a crea setul de date de antrenament, au fost realizate fotografii ale tablei de șah din diverse unghiuri și condiții de iluminare. Aceste imagini au fost ulterior etichetate folosind platforma Roboflow. Setul inițial de date constă în aproximativ 200 de imagini.

Etapele de preprocesare aplicate în platforma Roboflow au inclus conversia imaginilor în format grayscale, un pas necesar deoarece culoarea nu reprezintă un criteriu esențial pentru problema noastră, iar antrenarea modelului devine astfel mai eficientă. Imaginile au fost redimensionate la 640x640 pixeli pentru a asigura consistența datelor furnizate modelului.

Pentru a îmbunătăți capacitatea modelului de a generaliza și a asigura performanțe optime în scenarii reale, datele de antrenament au fost augmentate.

Scriptul de augmentare

Scriptul definește și aplică un set de transformări asupra imaginilor dintr-un set de date, creând astfel versiuni noi ale imaginilor originale. Aceste transformări includ modificări ale luminozității și contrastului, rotații, adăugarea de zgomot gaussian și ISO, aplicarea de blur de mișcare și transformări de perspectivă.

Definirea Pipeline-ului de Augmentare

```
1 transform = A.Compose([
2     A.RandomBrightnessContrast(brightness_limit=0.3, contrast_limit=0.3,
3     p=1.0),
4     A.Affine(rotate=(-180, 180), p=1.0),
5     A.GaussNoise(var_limit=(10.0, 50.0), p=0.5),
6     A.Rotate(limit=360, p=1.0),
7     A.MotionBlur(blur_limit=5, p=0.5),
8     A.ISONoise(color_shift=(0.01, 0.05), intensity=(0.1, 0.5), p=0.5),
9     A.Perspective(scale=(0.05, 0.1), p=0.5),
10    ])
```

Funcțiile de augmentare aplicate imaginilor de antrenament au primit parametri specifici care definesc intervalele de transformare și probabilitatea aplicării fiecărei transformări. De exemplu, pentru rotație, intervalul de transformare este un număr de grade specificat și există o probabilitate care indică sansa ca transformarea să fie aplicată.

Antrenarea modelului

```
1 from ultralytics import YOLO
2 import torch
3
4 def chess_train():
5
6     model = YOLO('yolov8n.pt')
7     torch.cuda.empty_cache()
8
9     model.train(data='Chess-corner/data.yaml', epochs=50,
10     imgsz=640, device='cuda', amp=False)
```

Codul de mai sus încarcă un model preantrenat YOLOv8 folosind biblioteca `ultralytics`. După încărcarea modelului, memoria cache a GPU-ului este golită pentru a preveni eventualele probleme legate de memorie. Modelul este apoi antrenat pe un set de date personalizat, specificat în fișierul `data.yaml`, pe o durată de 50 de epoci și cu o dimensiune a imaginii de 640x640 pixeli. Antrenamentul se desfășoară pe GPU, folosind CUDA, pentru a beneficia de accelerarea hardware.

Script de detectie

După ce modelul a fost antrenat, aplicația Camo a fost utilizată pentru a transmite informații video în timp real către calculator. Acest flux video live este procesat de un script care, pentru fiecare cadru al filmării, rulează modelul de detectie YOLOv8 și desenează un pătrat în jurul fiecarui rezultat detectat.

```

1 import cv2
2 import numpy as np
3 from ultralytics import YOLO
4
5 model_corners = YOLO('runs/detect/train10/weights/best.pt')
6 model_corners.to('cuda')
7
8 cap = cv2.VideoCapture(2)
9
10 if not cap.isOpened():
11     print("Cannot open camera")
12     exit()
13
14 while True:
15     ret, frame = cap.read()
16     if not ret:
17         print("Failed to grab frame")
18         break
19
20     gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
21     gray_frame_3ch = cv2.cvtColor(gray_frame, cv2.COLOR_GRAY2BGR)
22
23     results_corners = model_corners(gray_frame_3ch)
24     centers = []
25
26     boxes_with_confidence = [(box.xyxy[0], box.conf) for result in
27     results_corners for box in result.boxes]
28
29     sorted_boxes = sorted(boxes_with_confidence, key=lambda x: x[1],
30     reverse=True)
31
32     for i in range(min(4, len(sorted_boxes))):
33         x1, y1, x2, y2 = map(int, sorted_boxes[i][0])
34         center_x = (x1 + x2) // 2
35         center_y = (y1 + y2) // 2
36         centers.append((center_x, center_y))
37
38         cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)
39
40     height, width = frame.shape[:2]
41     if height > width:
42         new_height = 800
43         new_width = int((width / height) * 800)
44     else:
```

```

45     new_width = 800
46     new_height = int((height / width) * 800)
47     frame_resized = cv2.resize(frame, (new_width, new_height))
48
49     cv2.imshow('YOLOv8 Corner Detection', frame_resized)
50     if cv2.waitKey(1) & 0xFF == ord('q'):
51         break
52
53 cap.release()
54 cv2.destroyAllWindows()

```

Următorii pași descriu modul în care scriptul de detecție funcționează:

- **Încărcarea modelului:** Modelul este încărcat și transferat pe GPU pentru a beneficia de accelerarea hardware.
- **Captura video:** Camera video este inițializată pentru a captura fluxul video live. Dacă camera nu poate fi deschisă, scriptul se oprește.
- **Procesarea fiecărui cadru:** Pentru fiecare cadru capturat:
 - Cadrul este convertit în niveluri de gri pentru a simplifica procesarea.
 - Modelul detectează colțurile tablei de șah în cadrul procesat.
 - Colțurile detectate sunt sortate în funcție de scorul de încredere și se selectează cele mai înalte 4 colțuri.
 - Un pătrat este desenat în jurul fiecărui colț detectat.
 - Cadrul procesat este afișat într-o fereastră.

Figura 6.2 prezintă rezultatele scriptului de detecție a colțurilor.

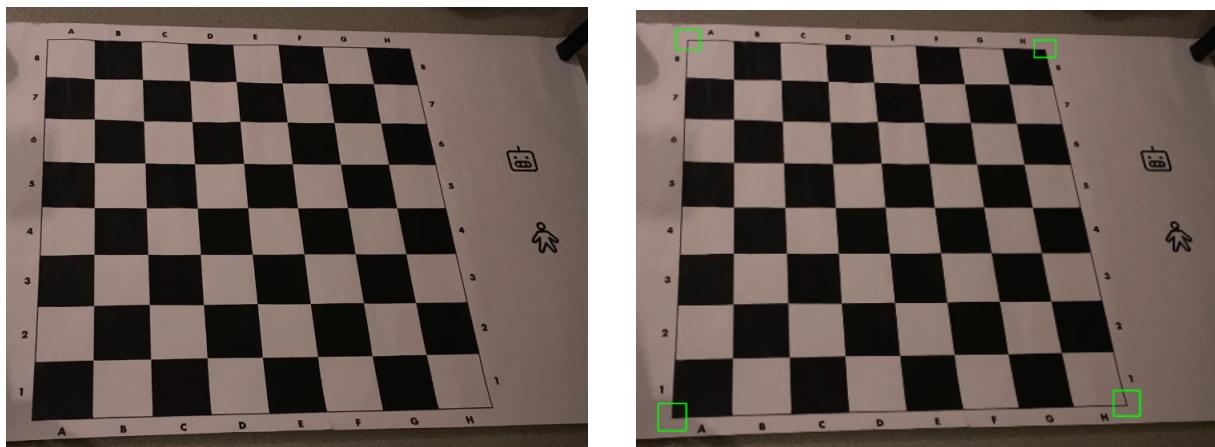


Figura 6.2: Detectarea colțurilor tablei de șah în timp real

6.2 Transformarea de perspectivă și segmentarea tablei

După determinarea coordonatelor colțurilor tablei de șah, s-a aplicat o transformare de perspectivă utilizând biblioteca OpenCV. Primele două cadre din figura 6.3 ilustrează modul în care această transformare a realocat pixelii astfel încât perspectiva camerei a devenit paralelă cu

tabla de șah. Acest pas a fost esențial deoarece a permis împărțirea distanțelor dintre colțurile tablei de șah în opt segmente egale atât pe axa verticală, cât și pe cea orizontală. Folosind matricea de rotație generată de funcția din biblioteca OpenCV, coordonatele fiecărui punct calculat în imaginea transformată au fost convertite înapoi în cadrul original. Aceste puncte au fost ulterior utilizate pentru trasarea liniilor de delimitare a pătratelor, aşa cum se poate observa în ultimul cadru al figurii 6.3.

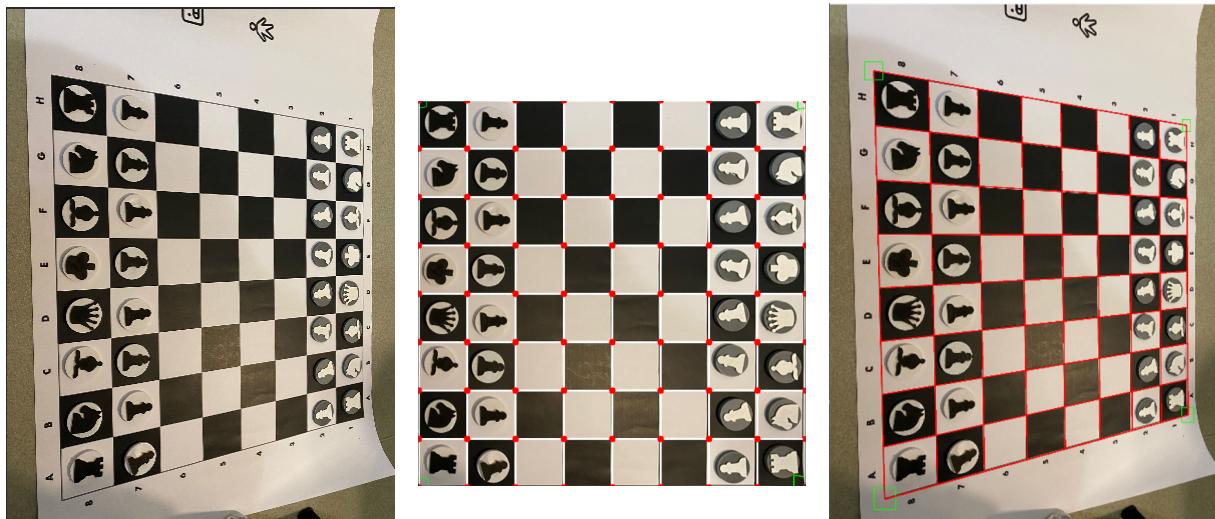


Figura 6.3: Etape de segmentare a tablei în pătrate

Următorul cod implementează pașii descriși anterior pentru a realiza transformarea de perspectivă a imaginii tablei de șah. Codul salvează fiecare pătrat al tablei de șah ca imagine separată, pentru a putea fi utilizat ulterior în etapa de clasificare a pieselor.

```

1 import cv2
2 import numpy as np
3 import os
4 import itertools
5
6 M = cv2.getPerspectiveTransform(src_points, dst_points)
7
8 if np.linalg.cond(M) < 1 / np.finfo(M.dtype).eps:
9     M_inv = np.linalg.inv(M)
10    warped_image = cv2.warpPerspective(frame, M, (width, height))
11
12    grid_rows = 8
13    grid_cols = 8
14    chessboard_grid = [[None] * grid_cols for _ in range(grid_rows)]
15
16    for i in range(grid_rows):
17        for j in range(grid_cols):
18            top_left = (int(width * j / grid_cols), int(height * i /
19 grid_rows))
20            top_right = (int(width * (j + 1) / grid_cols),
21 int(height * i / grid_rows))
22            bottom_right = (int(width * (j + 1) / grid_cols),
23 int(height * (i + 1) / grid_rows))
24            bottom_left = (int(width * j / grid_cols),
25 int(height * (i + 1) / grid_rows))
26
27            top_left_original =
28 cv2.perspectiveTransform(np.array([[top_left]]),
29                         dtype='float32'), M_inv)[0][0]

```

```

30         top_right_original =
31 cv2.perspectiveTransform(np.array([[top_right]]),
32                         dtype='float32'), M_inv)[0][0]
33         bottom_right_original =
34 cv2.perspectiveTransform(np.array([[bottom_right]]),
35                         dtype='float32'), M_inv)[0][0]
36         bottom_left_original =
37 cv2.perspectiveTransform(np.array([[bottom_left]]),
38                         dtype='float32'), M_inv)[0][0]
39
40         chessboard_grid[i][j] = (
41             top_left_original, top_right_original,
42             bottom_right_original, bottom_left_original)
43
44 all_squares = np.zeros((height + 7 * padding, width + 7 * padding, 3),
45 dtype=np.uint8)
46
47 chessboard_matrix = [[ "0" for _ in range(8)] for _ in range(8)]
48
49 for i in range(grid_rows):
50     for j in range(grid_cols):
51         square_coords = np.array(chessboard_grid[i][j], np.int32)
52         square_coords = square_coords.reshape((-1, 1, 2))
53
54         top_left, top_right, bottom_right, bottom_left =
55         chessboard_grid[i][j]
56         src_points_square = np.array([top_left, top_right,
57             bottom_right, bottom_left], dtype="float32")
58         dst_points_square = np.array([
59             [0, 0],
60             [top_right[0] - top_left[0], 0],
61             [top_right[0] - top_left[0], bottom_left[1] - top_left[1]],
62             [0, bottom_left[1] - top_left[1]]
63         ], dtype="float32")
64         M_square = cv2.getPerspectiveTransform(
65         src_points_square, dst_points_square)
66         square_img = cv2.warpPerspective(
67         frame, M_square,
68             (int(dst_points_square[2][0]), int(dst_points_square[2][1])))
69
70         resized_square_img = cv2.resize(
71         square_img, (int(width / grid_cols), int(height / grid_rows)))
72         counter = itertools.count()
73         count = next(counter)
74         square_img_path = os.path.join(
75         squares_dir, f"square_{i}_{j}_{count}.png")
76         cv2.imwrite(square_img_path, square_img)

```

- `cv2.getPerspectiveTransform(src_points, dst_points)`: Calculează matricea de transformare de perspectivă folosind coordonatele colțurilor tablei de șah.
- `np.linalg.cond(M) < 1 / np.finfo(M.dtype).eps`: Verifică dacă matricea este inversabilă, ceea ce este esențial pentru a proiecta punctele înapoi în imaginea originală.
- `cv2.warpPerspective(frame, M, (width, height))`: Aplică transformarea de perspectivă pentru a obține o vedere "de sus" a tablei de șah.
- Pentru fiecare pătrat, se determină coordonatele colțurilor în imaginea transformată și se proiectează înapoi în cadrul original.

- Linile sunt trase pe imaginea originală pentru a delimita pătratele folosind coordonatele colțurilor calculate.
- Fiecare pătrat este extras, redimensionat și salvat ca o imagine separată

6.3 Detectarea pieselor și formarea tablei în format digital

După segmentarea tablei de joc în 64 de pătrate, un model de clasificare este aplicat pe fiecare dintre aceste pătrate. Scopul acestui model este să analizeze imaginea fiecărui pătrat și să determine dacă acesta conține una dintre cele 12 piese de șah sau dacă este gol.

Procesul de clasificare este similar cu cel utilizat pentru detectarea colțurilor. Un set de date de 250 de imagini a fost etichetat manual folosind platforma Roboflow. Imaginele au fost apoi convertite în grayscale și redimensionate la 96x96 pixeli. Această rezoluție mică a fost aleasă pentru a asigura că imaginile pe care se antrenează modelul sunt cât mai apropiate de cele din realitate. În funcție de distanța camerei față de tabla de șah și de rezoluția video, dimensiunea unui pătrat individual poate varia, însă în configurația actuală, aceasta are valori apropiate de 90 de pixeli.

Aceeași metodă de augmentare a fost aplicată și pentru imaginile cu piese, pentru a simula diverse condiții de iluminare și unghiuri de vizualizare. Procesul de antrenare a fost, de asemenea, similar, cu diferența principală constând în modelul utilizat. Pentru clasificarea pieselor a fost folosit modelul `yolov8n-cls`. Arhitectura acestui model este concepută special pentru sarcini de clasificare și permite identificarea precisă a diferitelor piese de șah.

Mai jos este prezentat codul responsabil pentru extragerea informațiilor din fiecare cadru și formarea unei imagini digitale care reprezintă starea actuală a tablei de joc:

6.3.1 Extragerea și clasificarea pieselor de pe tabla de șah

Următorul cod explică cum se realizează extragerea fiecărui pătrat de pe tabla de șah, salvarea acestora ca imagini individuale și clasificarea lor folosind un model de clasificare a pieselor de șah.

```
1 chessboard_matrix = [[ "0" for _ in range(8)] for _ in range(8)]
```

Matricea `chessboard_matrix` este inițializată pentru a reprezenta o tablă de șah goală de 8x8, cu fiecare element inițializat la ”0” pentru a indica absența pieselor.

```
1 for i in range(grid_rows):
2     for j in range(grid_cols):
3         square_coords = np.array(chessboard_grid[i][j], np.int32)
4         square_coords = square_coords.reshape((-1, 1, 2))
```

Pentru fiecare pătrat al tablei de șah (`grid_rows` și `grid_cols`), coordonatele fiecărui pătrat sunt extrase și transformate într-un format compatibil cu OpenCV.

```
1     top_left, top_right, bottom_right, bottom_left =
2 chessboard_grid[i][j]
3         src_points_square = np.array([top_left, top_right, bottom_right,
4 bottom_left], dtype="float32")
5         dst_points_square = np.array([
6             [0, 0],
```

```

7             [top_right[0] - top_left[0], 0],
8             [top_right[0] - top_left[0], bottom_left[1] - top_left[1]],
9             [0, bottom_left[1] - top_left[1]]
10            ], dtype="float32")
11    M_square = cv2.getPerspectiveTransform(
12 src_points_square, dst_points_square)
13    square_img = cv2.warpPerspective(
14 frame, M_square, (int(dst_points_square[2][0]),
15 int(dst_points_square[2][1])))

```

Se obțin coordonatele fiecărui colț al pătratului, se definește matricea pentru transformarea de perspectivă și se aplică această transformare pentru a extrage imaginea pătratului.

```

1    results_pieces = model_pieces(square_img)

```

Modelul de clasificare a pieselor de șah este rulat pe fiecare imagine de pătrat pentru a determina ce piesă, dacă există, se află în acel pătrat.

```

1    if isinstance(results_pieces, list):
2        results = results_pieces[0]
3
4        class_names = results.names
5        top_class_index = results.probs.top1
6        top_class_confidence = results.probs.top1conf.item()
7        max_class_name = class_names[top_class_index]
8
9        chessboard_matrix[i][j] = max_class_name

```

Se extrag numele claselor și probabilitățile asociate cu clasificarea, identificând piesa cu cea mai mare încredere și stocând rezultatul în matricea `chessboard_matrix`.

6.3.2 Formarea și afișarea tablei digitale

Funcția `construct_fen`

Funcția transformă o matrice ce reprezintă tabla de șah într-un sir FEN. Mai jos este descris codul acestei funcții și explicațiile pentru fiecare secțiune:

```

1 def construct_fen(chessboard_matrix):
2     piece_map = {
3         "WP": "P", "WR": "R", "WN": "N", "WB": "B", "WQ": "Q", "WK": "K",
4         "BP": "p", "BR": "r", "BN": "n", "BB": "b", "BQ": "q", "BK": "k",
5         "Background": "0"
6     }
7
8     fen_rows = []
9     for row in chessboard_matrix:
10         fen_row = ""
11         empty_count = 0
12         for cell in row:
13             if cell == "Background":
14                 empty_count += 1
15             else:
16                 if empty_count > 0:
17                     fen_row += str(empty_count)
18                     empty_count = 0
19                 fen_row += piece_map.get(cell, cell)

```

```

20     if empty_count > 0:
21         fen_row += str(empty_count)
22         fen_rows.append(fen_row)
23
24     fen = "/" .join(fen_rows)
25     return fen

```

- **piece_map**: Un dicționar care mapează codurile pieselor de șah la caracterele corespunzătoare din notația FEN. De exemplu, "WP" (White Pawn) este mapat la "P", iar "BP" (Black Pawn) este mapat la "p". Celulele goale sunt reprezentate prin "Background" și sunt mapate la "0".
- **fen_rows**: O listă care va conține rândurile individuale ale șirului FEN.
- **for row in chessboard_matrix**: Itereză prin fiecare rând al matricei tablei de șah.
- **fen_row = ""**: Inițializează un sir gol pentru a construi rândul curent în notația FEN.
- **empty_count = 0**: Inițializează un contor pentru a număra celulele goale consecutive.
- **for cell in row**: Itereză prin fiecare celulă din rândul curent.
- **if cell == "Background"**: Verifică dacă celula curentă este goală.
 - **empty_count += 1**: Incrementează contorul de celule goale.
- **else**: Dacă celula curentă conține o piesă:
 - **if empty_count > 0**: Verifică dacă există celule goale numărate anterior.
 - * **fen_row += str(empty_count)**: Adaugă numărul de celule goale în șirul FEN.
 - * **empty_count = 0**: Resetează contorul de celule goale.
 - **fen_row += piece_map.get(cell, cell)**: Adaugă caracterul corespunzător piesei curente în șirul FEN.
- **if empty_count > 0**: Dacă există celule goale la sfârșitul rândului:
 - **fen_row += str(empty_count)**: Adaugă numărul de celule goale în șirul FEN.
- **fen_rows.append(fen_row)**: Adaugă rândul complet în lista de rânduri FEN.
- **fen = "/" .join(fen_rows)**: Concatenează toate rândurile FEN cu separatorul "/" pentru a forma șirul FEN complet.
- **return fen**: Returnează șirul FEN.

Pentru a ilustra poziția curentă, a fost reutilizată funcția `draw_chessboard`, implementată în prima etapă a acestui proiect. Această funcție primește ca parametru șirul FEN calculat anterior și returnează o imagine care reprezintă starea actuală a tablei de șah. Figura 6.4 prezintă procesul de digitalizare a tablei de șah, de la starea inițială, la segmentarea imaginii și, în final, la generarea imaginii grafice.

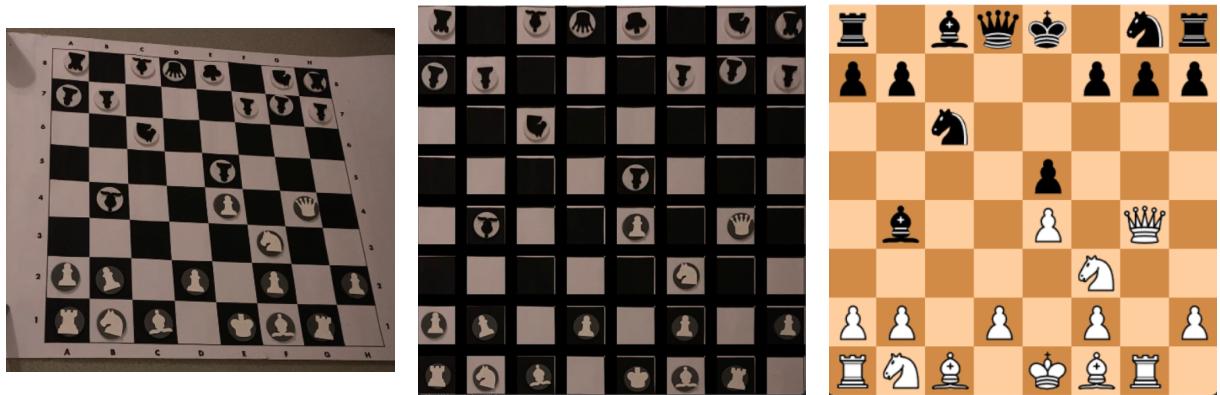


Figura 6.4: Digitalizarea tablei de șah

6.3.3 Determinarea secvenței de mutare

În plus față de starea curentă a tablei de șah, algoritmul decizional trebuie să primească informații cu privire la momentul în care trebuie să ia o decizie. În acest scop, lângă tabla de șah au fost plasate două logo-uri (a se vedea figura 6.5). Unul reprezintă rândul robotului, iar celălalt rândul operatorului. Pentru a delimita rândurile, după ce jucătorul cu piesele albe face o mutare, un obstacol va fi mutat de pe logo-ul robotului pe cel al operatorului. Astfel, în primul cadru în care logo-ul robotului este detectat, robotul poate prelua datele relevante ale poziției de pe tablă și poate lua o decizie.

Pentru realizarea acestui pas, a fost utilizat modelul Yolov8 în același mod ca la detectarea colțurilor. După adnotarea imaginilor, acestea au fost augmentate, iar modelul a fost antrenat.

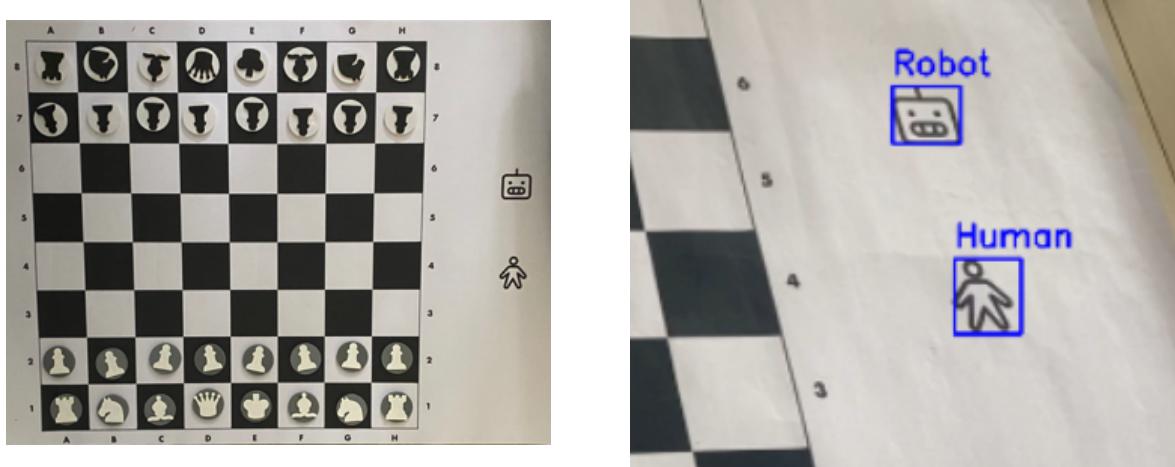


Figura 6.5: Detectarea logo

6.4 Rezultatele antrenării pentru modelele utilizate în sistemul de vedere artificială

6.4.1 Explicația metricilor returnate de Yolov8 la finalul antrenamentului

După finalizarea procesului de antrenare a unui model YOLOv8, sunt returnate o serie de metrii pentru a evalua performanța acestuia [19]. Metricile de performanță sunt calculate atât în timpul antrenamentului, cât și în timpul validării modelului pe un set de date de testare

- **Box Loss (train/box_loss și val/box_loss)**
 - Măsoară eroarea dintre coordonatele casetelor de delimitare prezise de model și cele reale.
- **Classification Loss (train/cls_loss și val/cls_loss)**
 - Indică cât de bine poate modelul să clasifice corect obiectele în categoriile corespunzătoare.
 - Se calculează comparând probabilitățile prezise de model pentru fiecare clasă cu etichetele reale.
- **Precision (metrics/precision(B))**
 - Măsoară proporția de predicții corecte între toate predicțiile făcute de model.
 - Formula pentru precizie este:
$$\text{Precizie} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

unde TP reprezintă *true positives* (predicții corecte) și FP reprezintă *false positives* (predicții incorecte).
- **Recall (metrics/recall(B))**
 - Măsoară proporția de exemple corecte detectate de model dintre toate exemplele reale.
 - Formula pentru recall este:
$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

unde FN reprezintă *false negatives* (exemple reale care nu au fost detectate).
- **Mean Average Precision (metrics/mAP50(B) și metrics/mAP50-95(B))**
 - **Intersection over Union (IoU)** este o măsură a suprapunerii dintre două casete de delimitare, definită ca raportul dintre aria de suprapunere a casetelor și aria totală combinată a casetelor (a se vedea figura 6.6). IoU măsoară cât de bine se suprapun casetele de delimitare prezise de model cu cele reale.
 - **mAP50** reprezintă media precisiei la un prag de 50% *Intersection over Union* (IoU). O predicție este considerată corectă dacă IoU între caseta prezisă și cea reală este de cel puțin 50%.
 - **mAP50-95** este media precisiei la diverse praguri de *Intersection over Union* (IoU), de la 50% la 95% în pași de 5%. Aceasta oferă o măsură mai cuprinzătoare a performanței modelului la diverse nivele de precizie.

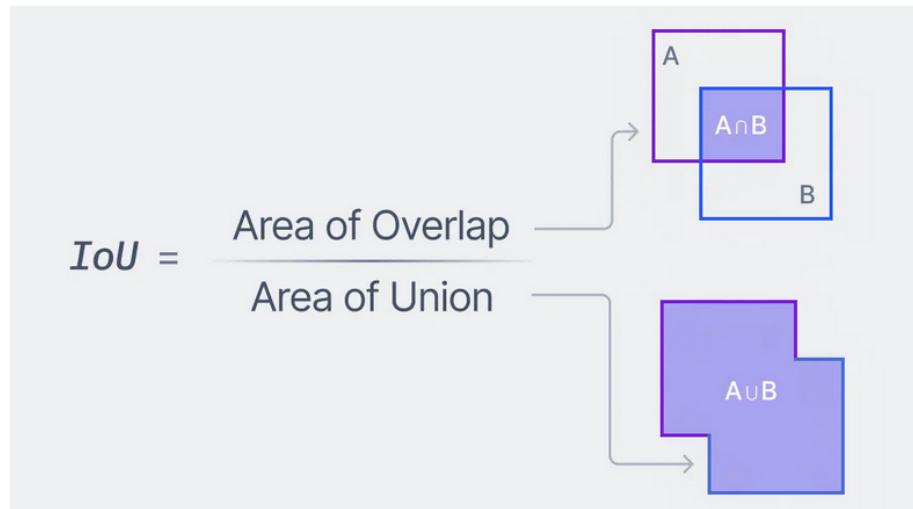


Figura 6.6: Reprezentare vizuală a reprezentării IOU [32]

6.4.2 Interpretarea Rezultatelor Modelului YOLOv8 pentru Detecția Colțurilor

Graficul din figura 6.7 prezintă evoluția diferitelor metrici și pierderi ale modelului YOLOv8 pe parcursul antrenamentului pentru detecția colțurilor. Pe axa X este prezent numărul de epoci, iar pe axa Y sunt prezente valorile metricilor despre care am discutat mai sus. Observăm că pierderile de antrenament (*train/box_loss* și *train/cls_loss*) scad constant, indicând faptul că modelul învăță să prezică mai bine coordonatele casetelor de delimitare și să clasifice corect obiectele. Aceeași tendință descrescătoare este observată și în pierderile de validare (*val/box_loss*, *val/cls_loss* și *val/dfl_loss*), sugerând că modelul generalizează bine la date noi).

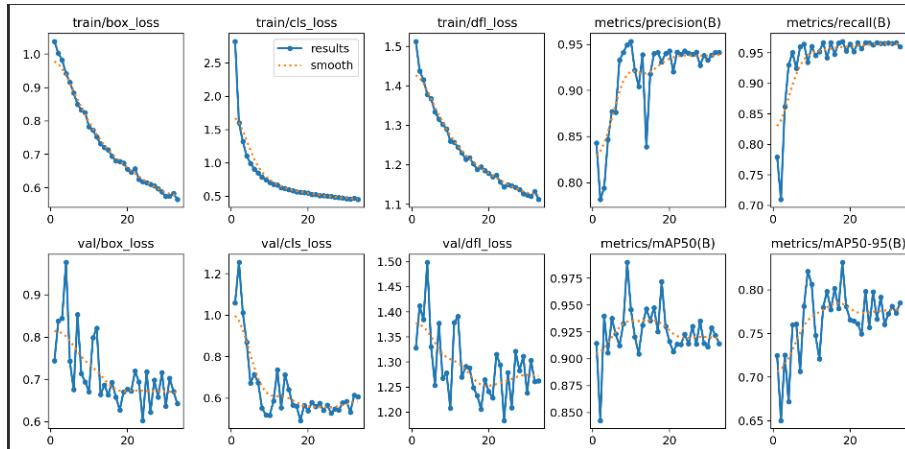


Figura 6.7: Rezultatele antrenamentului YOLOv8 pentru detecția colțurilor

În ceea ce privește metricile de performanță, precizia (*metrics/precision(B)*) și *recall*-ul (*metrics/recall(B)*) cresc pe parcursul antrenamentului și se stabilizează la valori ridicate, în jur de 0.95, ceea ce indică o detecție robustă. Cu toate acestea, metricile de *mean average precision* (*metrics/mAP50(B)* și *metrics/mAP50-95(B)*) prezintă fluctuații mai mari. *mAP50* rămâne relativ ridicat, în jur de 0.95, ceea ce sugerează că modelul are o performanță bună la un prag de 50% *Intersection over Union* (IoU). Totuși, *mAP50-95* variază mai mult, ceea ce indică o variabilitate în performanță la diferite nivele de precizie ale IoU, dar se stabilizează în jurul valorii de 0.75, reflectând o performanță rezonabilă la diverse praguri de suprapunere.

La etapa de etichetare a colțurilor, dimensiunea chenarului care încadă colțurile variază de la imagine la imagine, ceea ce face ca aria să varieze. Această variabilitate poate influența

calculul *Intersection over Union* și implicit metricile de performanță, ducând la fluctuații mai mari în valorile *mAP50-95*.

6.4.3 Interpretarea rezultatelor modelului YOLOv8 pentru detecția secvenței de mutare

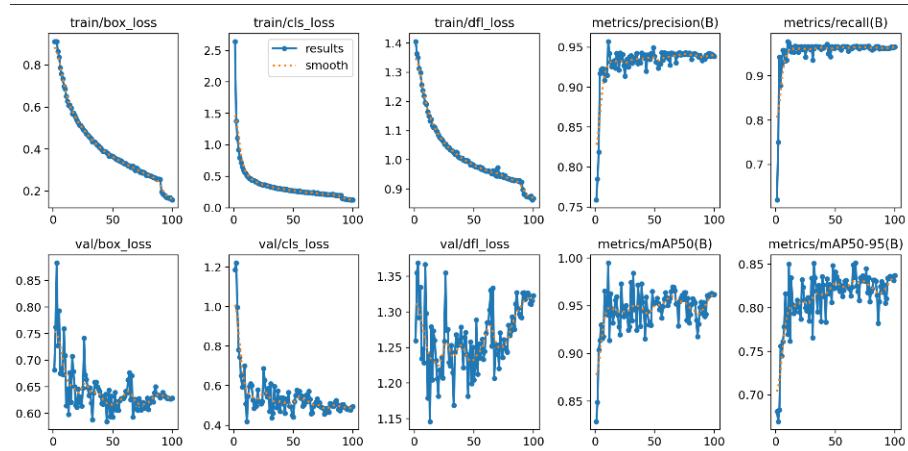


Figura 6.8: Rezultatele antrenamentului YOLOv8 pentru detecția secvenței de mutare

Rezultatele acestui model, prezentate în figura 6.8 sunt similare cu rezultatele prezentate anterior. Graficele de pierdere pentru seturile de date de antrenament și validare (train/box_loss, train/cls_loss, val/box_loss, val/cls_loss) arată o scădere continuă și stabilizare spre finalul antrenamentului. Metricile de precizie și recall se stabilizează în jurul valorilor ridicate, semnalând o performanță solidă a modelului în identificarea corectă a claselor.

Graficul arată că modelul detectează cu o acuratețe de aproximativ 95% pe setul de date de antrenament și de validare. *Mean Average Precision (metrics/mAP50(B))* se menține stabilă în jur de 95%, în timp ce *metrics/mAP50-95(B)* variază mai mult, dar se stabilizează în jurul valorii de 85%, indicând o performanță rezonabilă la diverse nivele de suprapunere.

6.4.4 Rezultatele modelului de clasificare a pieselor

Interpretarea Matricei de Confuzie Normalizate Matricea de confuzie normalizată construită în timpul antrenamentului (a se vedea figura 6.9) arată performanța modelului de clasificare pentru fiecare clasă specifică. În această matrice, fiecare rând reprezintă instanțele din clasa reală, iar fiecare coloană reprezintă instanțele prezise de model. Diagonala principală conține valorile de 1.00 pentru majoritatea claselor, indicând o acuratețe perfectă în prezicerea acestor clase.

In matricea construită cu date de testare (a se vedea figura 6.10), valorile de pe diagonala principală variază ușor, dar indică un model precis capabil să clasifice corect aproape toate instanțele pentru fiecare clasă specifică.

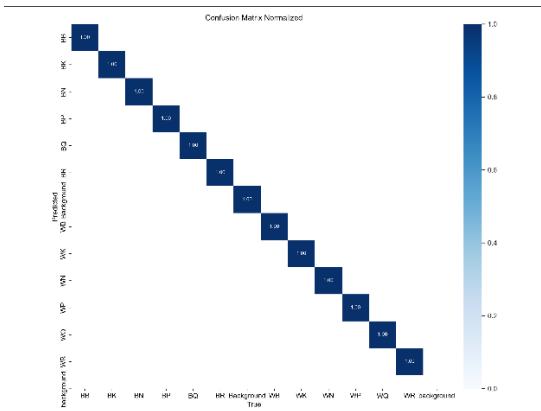


Figura 6.9: Matricea de Confuzie Normalizată construită cu date de valdare

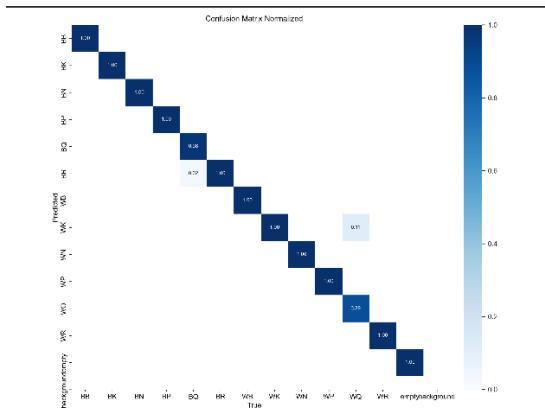


Figura 6.10: Matricea de Confuzie Normalizată construită cu date de testare

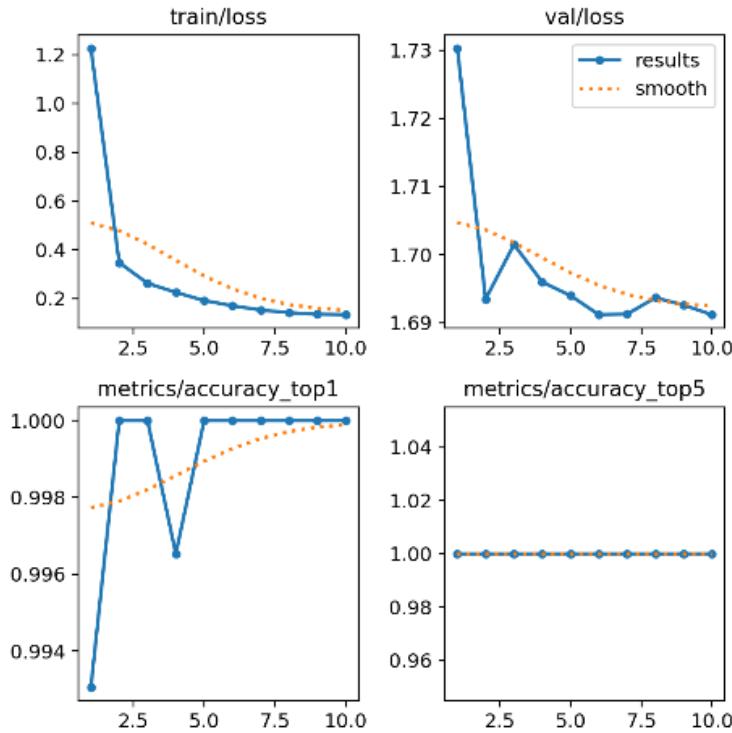


Figura 6.11: Rezultate de antrenare al modelului de clasificare al pieselor

Interpretarea Graficelor de Pierdere și Acuratete

- **train/loss și val/loss:** Graficele de pierdere pentru antrenament și validare arată o scădere semnificativă a pierderilor pe parcursul celor 10 epoci (a se vedea figura 6.11).
- **metrics/accuracy_top1 :** Graficul de acuratețe top-1 arată valori foarte ridicate, aproape de 1.00, cu fluctuații minime. Aceasta indică faptul că modelul prezice clasa corectă în prima sa alegere pentru aproape toate instanțele.
- **metrics/accuracy_top5 :** Graficul de acuratețe top-5 rămâne constant la 1.00, sugerând că modelul include întotdeauna clasa corectă în primele cinci predicții.

7 Implementarea logicii de joc

Această secțiune pune împreună implementările despre care am discutat până acum și definește bucla de joc.

7.1 Interfața grafică

O nouă interfață grafică a fost dezvoltată pentru a integra toate componentele necesare jocului de șah. Această interfață trebuie să conțină informații legate de starea și răspunsurile oferite de motorul de inferențe, starea curentă a fiecărui cadru și tabla digitalizată văzută de cameră.

Pentru a configura interfața grafică, am folosit biblioteca `tkinter` pentru a crea și gestiona ferestrele și widget-urile. Layout-ul este împărțit în patru secțiuni egale, fiecare având un rol specific (a se vedea figura 7.1):

- **Secțiunea 1:** Afisează imaginea curentă a tablei de șah în interiorul motorului de inferență.
- **Secțiunea 2:** Afisează conținutul capturat de cameră.
- **Secțiunea 3:** Afisează imaginea digitalizată a tablei de șah văzută de camera video.
- **Secțiunea 4:** Acționează ca o consolă pentru afișarea mutărilor.

Fiecare secțiune este configurată pentru a se redimensiona automat și a se adapta la dimensiunile ferestrei principale.

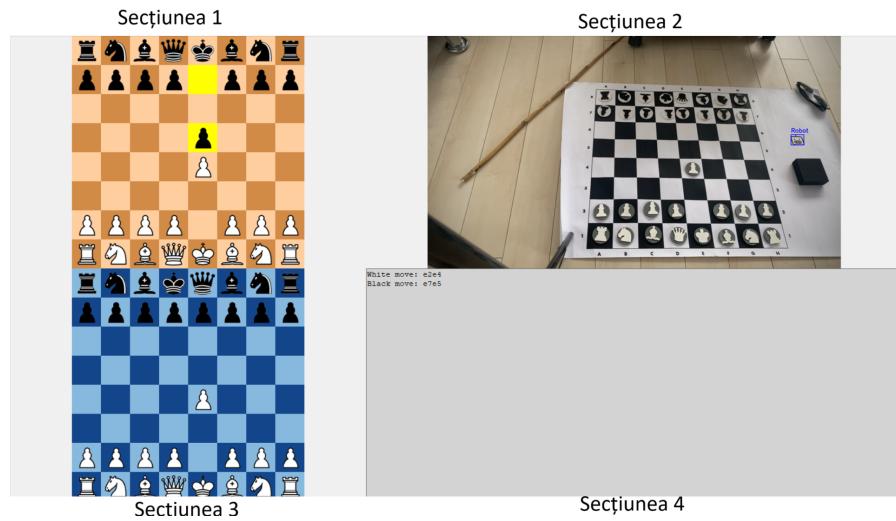


Figura 7.1: Interfața grafică în formatul final

7.1.1 Bucla de joc

Bucla de joc este responsabilă de actualizarea continuă a stării jocului și a interfeței grafice. Aceasta este implementată folosind un fir de execuție separat, care procesează fiecare cadru capturat de cameră și actualizează starea jocului și interfața grafică în consecință.

```

1 def process_frames(camera_processor, root, update_camera_content,
2 section4_text, chessGame):
3     previous_robot_count = camera_processor.robot_count
4
5     while True:
6         ret, frame = camera_processor.cap.read()
7         if ret:
8             camera_processor.process_frame(frame)
9             root.after(10, update_camera_content)
10            previous_fen = chessGame.gs.board.fen()
11
12            if camera_processor.robot_count > previous_robot_count:
13                previous_robot_count = camera_processor.robot_count
14                fen_string = camera_processor.get_fen_string()
15                start_square, end_square = get_move(previous_fen,
16 fen_string)
17
18                if start_square and end_square:
19                    move_notation = (start_square + end_square).lower()
20                    section4_text.insert(tk.END,
21 f"White move: {move_notation}\n")
22                    section4_text.see(tk.END)
23
24                    move = chess.Move.from_uci(move_notation)
25                    if move in chessGame.gs.board.legal_moves:
26                        chessGame.gs.board.push(move)
27                        chessGame.update_board(move)
28                        chessGame.gs.white_to_move = False
29                        black_move = chessGame.play_black_turn(model)
30                        section4_text.insert(tk.END,
31 f"Black move: {black_move}\n")
32                        section4_text.see(tk.END)
33                    else:
34                        pass

```

În bucla principală, fiecare cadru este preluat de la cameră și procesat pentru a detecta modificările de pe tablă. Dacă este detectată o nouă mutare (indicată de creșterea contorului `robot_count`, care reprezintă numărul de apariții al logo-ului "Robot"), se determină mutarea efectuată folosind funcția `get_move`. Dacă mutarea este legală, noua stare a tablei de joc este trimisă către motorul de inferență din Secțiunea 1. Motorul răspunde cu o mutare, iar ciclul reîncepe. În cazul în care mutarea făcută de operatorul uman nu este legală, un mesaj este afișat în Secțiunea 4 pentru a alerta jucătorul. Acesta trebuie să revină la starea prezentă în secțiunea 1 și să refacă mutarea.

7.1.2 Mesajele afișate în consolă

```

1 section4_text.insert(tk.END, f"White move: {move_notation}\n")
2 section4_text.see(tk.END)
3 black_move = chessGame.play_black_turn(model)
4 section4_text.insert(tk.END, f"Black move: {black_move}\n")
5 section4_text.see(tk.END)

```

Secțiunea 4 acționează ca o consolă unde fiecare mutare efectuată este afișată pe rând. Mutările sunt afișate succesiv, oferind utilizatorului o vizualizare clară și continuă a progresului jocului. Fiecare mutare este adăugată la textul existent și consola este derulată automat pentru a arăta ultima mutare, asigurând astfel că informațiile sunt mereu actualizate. Mesajele afișate în consolă au rolul principal de a testa funcționalitatea, dar și de a comunica cu operatorul uman în cazul în care acesta efectuează o mutare ilegală. În astfel de situații, sistemul va informa operatorul că trebuie să revină la starea anterioară și să refacă mutarea.

8 Concluzii și posibilități de dezvoltare

Această lucrare subliniază potențialul rețelelor neuronale în abordarea problemelor complexe atunci când sunt susținute de seturi de date adecvate. Acest lucru a fost demonstrat prin utilizarea rețelei CNN pentru evaluarea pozițiilor de șah și prin implementările sistemului de vedere artificială.

Tehnologiile software disponibile în prezent oferă soluții eficiente pentru integrarea componentelor necesare, facilitând dezvoltarea unor proiecte de acest tip. Instrumentele analizate și implementate în cadrul acestei lucrări evidențiază capacitatea de automatizare a sarcinilor complexe, cum ar fi jocul de șah.

8.1 Concluzii legate de motorul de inferență

Motorul de inferență pentru jocul de șah, deși nu excelează, este capabil să joace partide de șah împotriva unui adversar uman. Jocul de șah necesită o înțelegere profundă și abilitatea de a lua decizii care să echilibreze avantajele pe termen scurt cu cele pe termen lung. Motorul de inferență dezvoltat în această lucrare poate fi îmbunătățit prin antrenarea pe seturi de date care includ poziții de final de partidă, pentru a asigura performanțe consistente pe toată durata jocului.

8.2 Concluzii legate de sistemul de vedere artificială

Modelele preantrenate de recunoaștere a obiectelor sunt instrumente puternice. YOLOv8 s-a dovedit a fi eficient în detectarea pozițiilor pe tabla de șah, iar digitalizarea acestor poziții a fost facilitată prin utilizarea Python și a bibliotecii OpenCV. Sistemul de vedere artificială poate fi însă perfecționat prin antrenarea pe seturi de date mai variate, care să includă diferite tipuri de piese de șah. Astfel, sistemul va deveni mai robust și versatil, putând fi utilizat în diverse scenarii.

8.3 Posibilități de dezvoltare

În stadiul actual, proiectul permite integrarea unui robot în sistem, automatizând complet partenerul de șah. Robotul poate utiliza sistemul de vedere artificială pentru a executa mutările decise de componenta decizională.

Motorul de inferență poate fi îmbunătățit prin expunerea la seturi de date focusate pe finaluri de partidă și prin optimizarea algoritmului de căutare, astfel încât acesta să își ajusteze dinamic adâncimea de căutare în funcție de timpul disponibil pentru fiecare mutare. Această îmbunătățire ar fi deosebit de utilă în fazele finale ale partidei, când numărul de piese pe tablă este redus, permitând o explorare explorare pe mai multe niveluri a posibilităților.

Sistemul de vedere artificială poate fi perfecționat și antrenat pe seturi de date extinse, pentru a recunoaște o varietate mai mare de piese de șah. Acest lucru ar permite utilizarea sistemului într-o gamă mai largă de scenarii, sporindu-i astfel aplicabilitatea și eficiența.

Modalitatea de detectare a rândului fiecărui jucător ar putea fi îmbunătățită prin integrarea unui alt model neuronal capabil să recunoască o mână de om. Astfel de modele preantrenate există și pot reduce riscul de eroare în evaluarea tablei de șah în momente nepotrivite.

Bibliografie

- [1] José Raúl Capablanca. *Capablanca's Last Chess Lectures*. Grosset & Dunlap, 1966, p. 24.
- [2] B. Jack Copeland. *The Essential Turing: Seminal Writings in Computing, Logic, Philosophy, Artificial Intelligence, and Artificial Life, Plus The Secrets of Enigma*. Oxford University Press, 2004, pp. 563–567.
- [3] Feng-Hsiung Hsu. *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press, 2002, pp. 1–50.
- [4] Feng-Hsiung Hsu. *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press, 2002, pp. 100–150.
- [5] The Stockfish Developers. *Stockfish: The Strong Open-Source Chess Engine*. 2024. URL: <https://stockfishchess.org/>.
- [6] The Chess Programming Wiki and ChessBase Authors. *Boris Handroid: The First Commercial Chess Robot*. 1980. URL: <https://www.chessprogramming.org/Boris>.
- [7] Rick Martin and Eldar Mukhametov. *Chess Terminator: Robotic Chess Match against Vladimir Kramnik*. 2011. URL: <https://newatlas.com/chess-terminator-robot-takes-on-former-champ-kramnik-in-blitz-match/20402/>.
- [8] Jim West. *ABB Robotics Showcases Chess-Playing Robot at Newlab in Detroit*. 2023. URL: <https://www.alamy.com/detroit-michigan-usa-25th-apr-2023-abb-robotics-showed-a-chess-playing-robot-as-ford-motor-co-opened-newlab-a-center-for-startup-companies-and-entrepreneurs-newlab-has-meeting-and-laboratory-space-it-is-adjacent-to-the-long-abandoned-michigan-central-railroad-station-which-ford-is-renovating-credit-jim-westalamy-live-news-image547783560.html>.
- [9] Steven J. Edwards. *Full Definition for FEN*. 2003. URL: <https://www.stmartz.com/ccc/index.php?id=53266>.
- [10] Chess.com. *Forsyth-Edwards Notation (FEN) in Chess*. 2023. URL: <https://www.chess.com/terms/fen-chess>.
- [11] Santiago Gonzalez. *How Does Chess AI Work?* 2021. URL: <https://medium.com/@santiagu.gap/how-does-chess-ai-work-73993a22d5c3>.
- [12] National Museums Liverpool. *Which is greater: the number of atoms in the universe or the number of chess moves?* 2023. URL: <https://www.liverpoolmuseums.org.uk/stories/which-greater-number-of-atoms-universe-or-number-of-chess-moves>.
- [13] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed. Prentice Hall, 2010, pp. 170–176.
- [14] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer Science & Business Media, 2010, pp. 1–20.
- [15] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. 3rd ed. Prentice Hall, 2008, pp. 26–28.
- [16] StudentsXStudents. *Object Detection with Artificial Intelligence*. 2021. URL: <https://studentsxstudents.com/object-detection-artificial-intelligence-34b513dfad30>.

- [17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016, pp. 326–370.
- [18] The Click Reader. *Introduction to Convolutional Neural Networks*. 2020. URL: <https://www.theclickreader.com/introduction-to-convolutional-neural-networks/>.
- [19] Ultralytics. *Ultralytics Documentation*. 2024. URL: <https://docs.ultralytics.com/>.
- [20] Wesley J. Chun. *Core Python Programming*. Prentice Hall, 2001, pp. 10–20. URL: https://www.oreilly.com/library/view/core-python-programming/0130260363/0130260363_ch01lev1sec2.html.
- [21] PyTorch Contributors. *PyTorch Documentation*. 2023. URL: <https://pytorch.org/docs/stable/index.html>.
- [22] OpenCV Team. *OpenCV: Open Source Computer Vision Library*. 2024. URL: <https://opencv.org/>.
- [23] Tkinter Developers. *Tkinter: The Standard Python Interface to the Tk GUI toolkit*. 2024. URL: <https://docs.python.org/3/library/tkinter.html>.
- [24] Niklas Fiekas. *python-chess: A chess library for Python*. 2024. URL: <https://python-chess.readthedocs.io/>.
- [25] Reincubate. *Camo*. 2023. URL: <https://reincubate.com/camo/>.
- [26] Inc. Roboflow. *Roboflow*. 2023. URL: <https://roboflow.com/>.
- [27] Cassandra Felstiner and Dennis Guichard. *Group Theory and the Rubik's Cube*. 2019. URL: <https://www.whitman.edu/documents/Academics/Mathematics/2019/Felstiner-Guichard.pdf>.
- [28] Adrian Ramos Bardin. *Application of convolutional neural network in chess evaluation*. 2021. URL: https://github.com/jhubar/Application-of-convolutional-neural-network-in-chess-evaluation/blob/master/Deep_Learning_Project_Report.pdf.
- [29] URL: <https://creately.com/diagram/example/gvt0PxHTJ8L/neural-network-diagram>.
- [30] utadcm0927. *Chess Position Evaluator*. 2023. URL: <https://www.kaggle.com/code/utadcm0927/chess-position-evaluator/input>.
- [31] utadcm0927. *Chess Position Evaluator Notebook*. 2023. URL: <https://www.kaggle.com/code/utadcm0927/chess-position-evaluator/notebook>.
- [32] V7 Labs. *Intersection over Union (IoU) for object detection*. 2023. URL: <https://www.v7labs.com/blog/intersection-over-union-guide#h2>.

Anexa - Codul realizat în cadrul proiectului

fisierul ChessEngine.py

```
1 import chess
2 import torch
3 import numpy as np
4 from model_train_again import fen_to_bit_vector
5
6 class GameState:
7     def __init__(self):
8         self.board = "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1"
9         self.board = chess.Board(self.board)
10        self.white_to_move = True
11        self.is_playing_Black = True;
12
13    @staticmethod
14    def evaluate_fen(fen):
15        piece_values = {
16            'P': 10, 'N': 30, 'B': 30, 'R': 50, 'Q': 90, 'K': 0,
17            'p': -10, 'n': -30, 'b': -30, 'r': -50, 'q': -90, 'k': 0
18        }
19
20        parts = fen.split()
21
22        if len(parts) != 6:
23            return 0 # Invalid FEN format
24
25        board_layout, turn, castling, en_passant, half_moves, full_moves = parts
26
27        board_score = 0
28
29        for char in board_layout:
30            piece_value = piece_values.get(char, 0)
31            board_score += piece_value
32
33        if turn == 'w':
34            board_score -= 20
35        else:
36            board_score += 20
37
38        if 'K' in castling:
39            board_score -= 10
40        if 'k' in castling:
41            board_score += 10
42
43        min_score = -540
44        max_score = 540
45
46        normalized_score = (board_score - min_score) / (max_score - min_score)
47        return normalized_score
48
49    @staticmethod
50    def generate_future_positions(fen):
51
52        board = chess.Board(fen)
53        future_positions = []
```

```
54
55     legal_moves = list(board.legal_moves)
56     for move in legal_moves:
57         board_copy = board.copy()
58         board_copy.push(move)
59         future_positions.append((board_copy.fen(), move))
60
61     return future_positions
62
63 @staticmethod
64 def minimax(fen, depth, maximizing_player=True):
65     if depth == 0:
66         return fen, GameState.evaluate_fen(fen)
67
68     if maximizing_player:
69         best_move = None
70         best_value = float('-inf')
71
72         for future_fen in GameState.generate_future_positions(fen):
73             _, value = GameState.minimax(future_fen, depth - 1, False)
74             if value > best_value:
75                 best_value = value
76                 best_move = future_fen
77
78         return best_move, best_value
79     else:
80         best_move = None
81         best_value = float('inf')
82
83         for future_fen in GameState.generate_future_positions(fen):
84             _, value = GameState.minimax(future_fen, depth - 1, True)
85             if value < best_value:
86                 best_value = value
87                 best_move = future_fen
88
89         return best_move, best_value
90
91 @staticmethod
92 def minimax_alpha_beta(fen, depth, alpha=float('-inf'), beta=float('inf'),
93 maximizing_player=True, is_playing_Black=True):
94     if depth == 0 or GameState.generate_future_positions(fen) == []:
95         score = GameState.evaluate_fen(fen)
96         if is_playing_Black:
97             score = -score
98         return fen, None, score
99
100    if maximizing_player:
101        best_move = None
102        best_value = float('-inf')
103
104        for future_fen, move in GameState.generate_future_positions(fen):
105            _, _, value = GameState.minimax_alpha_beta(future_fen, depth - 1,
106            alpha, beta, False, is_playing_Black)
107            if value > best_value:
108                best_value = value
109                best_move = move
110
111                alpha = max(alpha, best_value)
112                if beta <= alpha:
113                    break # Pruning
114                return fen, best_move, best_value
115            else:
116                best_move = None
117                best_value = float('inf')
118
119                for future_fen, move in GameState.generate_future_positions(fen):
```

```
120     _, _, value = GameState.minimax_alpha_beta(future_fen, depth - 1,
121     alpha, beta, True, is_playing_Black)
122     if value < best_value:
123         best_value = value
124         best_move = move
125
126     beta = min(beta, best_value)
127     if beta <= alpha:
128         break # Pruning
129     return fen, best_move, best_value
130
131 @staticmethod
132 def model_alpha_beta(fen, depth, alpha=float('-inf'), beta=float('inf'),
133 maximizing_player=True, model=None, is_playing_Black=True):
134     if depth == 0 or GameState.generate_future_positions(fen) == []:
135         score_board = GameState.evaluate_board(fen, model)
136         score_fen = GameState.evaluate_fen(fen)
137         score = 0.5 * score_board + 0.5 * score_fen
138
139     if is_playing_Black:
140         score = -score
141     return fen, None, score
142
143 if maximizing_player:
144     best_move = None
145     best_value = float('-inf')
146
147     for future_fen, move in GameState.generate_future_positions(fen):
148         _, _, value = GameState.model_alpha_beta(future_fen, depth - 1,
149         alpha, beta, False, model, is_playing_Black)
150         if value > best_value:
151             best_value = value
152             best_move = move
153
154         alpha = max(alpha, best_value)
155         if beta <= alpha:
156             break # Pruning
157     return fen, best_move, best_value
158 else:
159     best_move = None
160     best_value = float('inf')
161
162     for future_fen, move in GameState.generate_future_positions(fen):
163         _, _, value = GameState.model_alpha_beta(future_fen, depth - 1,
164         alpha, beta, True, model, is_playing_Black)
165         if value < best_value:
166             best_value = value
167             best_move = move
168
169         beta = min(beta, best_value)
170         if beta <= alpha:
171             break # Pruning
172     return fen, best_move, best_value
173
174 def evaluate_board(fen, model):
175
176     input_tensor = torch.tensor(fen_to_bit_vector(fen), dtype=torch.float32)
177     input_tensor = input_tensor.to(next(model.parameters()).device)
178     input_tensor = input_tensor.unsqueeze(0)
179
180     with torch.no_grad():
181         output = model(input_tensor)
182
183     value = output.item()
184
185     return value
```

fisierul ui.py

```
1 import random
2 import time
3 import torch
4 import numpy as np
5 import chess
6 import ChessEngine
7 from fentoboardimage import loadPiecesFolder, fenToImage
8 from PIL import Image, ImageTk, ImageDraw
9 import tkinter as tk
10
11 from model import Net
12
13 class ChessGame:
14     def __init__(self):
15         self.label = None
16         self.image_tk = None
17         self.last_move_highlighting = None
18         self.boardImage = None
19         self.root = tk.Tk()
20         self.gs = ChessEngine.GameState()
21         self.drag_data = {}
22         self.create_ui()
23
24     def create_ui(self):
25         self.boardImage = fenToImage(
26             fen=self.gs.board.fen(),
27             squarelength=80,
28             pieceSet=loadPiecesFolder("./images"),
29             darkColor="#D18B47",
30             lightColor="#FFCE9E",
31             highlighting=self.last_move_highlighting
32         )
33         self.image_tk = ImageTk.PhotoImage(self.boardImage)
34         self.label = tk.Label(self.root, image=self.image_tk)
35         self.label.pack()
36         self.label.bind("<ButtonPress-1>", self.on_drag_start)
37         self.label.bind("<ButtonRelease-1>", self.on_drag_end)
38         self.game_loop()
39
40     def on_drag_start(self, event):
41         col, row = event.x // 80, event.y // 80
42         piece = self.gs.board.piece_at(chess.square(col, 7 - row))
43         if piece and piece.color == chess.WHITE and self.gs.white_to_move:
44             self.drag_data = {'piece': piece, 'start_col': col, 'start_row': row}
45
46     def on_drag_end(self, event):
47         if self.drag_data:
48             col, row = event.x // 80, event.y // 80
49             start_square = chess.square(self.drag_data['start_col'],
50                 7 - self.drag_data['start_row'])
51             end_square = chess.square(col, 7 - row)
52             move = chess.Move(start_square, end_square)
53             if move in self.gs.board.legal_moves:
54                 self.gs.board.push(move)
55                 self.gs.white_to_move = not self.gs.white_to_move
56                 self.update_board(move)
57                 print("Move made:", move)
```

```
58         self.drag_data = {}
59
60     def game_loop(self):
61         model = Net()
62         model_path = r'D:\PycharmProjects\Licenta-v2\runs\train9\chessModel.pth'
63
64         model.load_state_dict(torch.load(model_path))
65         model.eval()
66
67         device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
68         print("Using device:", device)
69         while not self.gs.board.is_game_over():
70             if self.gs.white_to_move:
71                 self.root.update()
72                 self.root.after(100)
73             else:
74                 self.play_black_turn(model)
75                 self.root.update()
76                 self.root.after(100)
77             print("Joc încheiat")
78
79     def play_black_turn(self, model):
80         if not self.gs.white_to_move:
81             old_position = self.gs.board.fen()
82             _, best_move, _ = ChessEngine.GameState.model_alpha_beta(
83                 old_position, depth=3, alpha=float('-inf'), beta=float('inf'),
84                 maximizing_player=True, model=model, is_playing_Black=True
85             )
86
87             board_copy = chess.Board(old_position)
88             board_copy.push(best_move)
89             new_position = board_copy.fen()
90
91             print("old_pos:", old_position)
92             print("new_pos:", new_position)
93             print("best_move:", best_move)
94
95             self.gs.board.set_fen(new_position)
96             self.gs.white_to_move = not self.gs.white_to_move
97             self.update_board(best_move)
98
99     def update_board(self, move):
100         start_square = move.from_square
101         end_square = move.to_square
102
103         start_pos = chess.square_name(start_square)
104         end_pos = chess.square_name(end_square)
105
106         self.last_move_highlighting = {"yellow": [start_pos, end_pos]}
107
108         self.boardImage = fenToImage(
109             fen=self.gs.board.fen(),
110             squarelength=80,
111             pieceSet=loadPiecesFolder("./images"),
112             darkColor="#D18B47",
113             lightColor="#FFCE9E",
114             highlighting=self.last_move_highlighting
115         )
116         self.image_tk = ImageTk.PhotoImage(self.boardImage)
117         self.label.configure(image=self.image_tk)
118
119     def run(self):
120         self.root.mainloop()
121
122     def run_experiment(self, num_moves=50):
123         with open("minimax_times.txt", "w") as minimax_file, \
```

```

124         open("alpha_beta_times.txt", "w") as alpha_beta_file:
125             for move_num in range(num_moves):
126                 print(f"\nMutare {move_num + 1}: ")
127                 self.make_random_white_move()
128                 self.saved_board_state = self.gs.board.fen()
129
130                 print("\nCalculare miscare cu Minimax")
131                 start_time_minimax = time.time()
132                 self.make_black_move_minimax()
133                 end_time_minimax = time.time()
134                 print(f"Timp pentru Minimax: {end_time_minimax -
135 start_time_minimax:.4f} secunde")
136                 minimax_file.write(f"{end_time_minimax -
137 start_time_minimax:.4f}\n")
138
139                 self.gs.board.set_fen(self.saved_board_state)
140                 self.gs.white_to_move = not self.gs.white_to_move
141
142                 print("\nCalculare miscare cu Alpha-Beta Pruning...")
143                 start_time_alpha_beta = time.time()
144                 self.make_black_move_alpha_beta()
145                 end_time_alpha_beta = time.time()
146                 print(f"Timp pentru Alpha-Beta Pruning: {end_time_alpha_beta -
147 start_time_alpha_beta:.4f} secunde")
148                 alpha_beta_file.write(f"{end_time_alpha_beta -
149 start_time_alpha_beta:.4f}\n")
150
151     def make_black_move_minimax(self):
152         if not self.gs.white_to_move:
153             best_move, _ = ChessEngine.GameState.minimax(
154                 self.gs.board.fen(), depth=3, maximizing_player=True
155             )
156             self.gs.board.set_fen(best_move)
157             self.gs.white_to_move = not self.gs.white_to_move
158
159     def make_black_move_alpha_beta(self):
160         if not self.gs.white_to_move:
161             best_move, _ = ChessEngine.GameState.minimax_alpha_beta(
162                 self.gs.board.fen(), depth=5, alpha=float('-inf'),
163                 beta=float('inf'),
164                 maximizing_player=True
165             )
166             self.gs.board.set_fen(best_move)
167             self.gs.white_to_move = not self.gs.white_to_move
168
169 if __name__ == "__main__":
170     game = ChessGame()
171     game.run()
172     # game.run_experiment(num_moves=30)

```

fisierul model-train.py

```

1 import datetime
2 import os
3
4 from matplotlib import pyplot as plt
5 from torch.utils.data import DataLoader, Dataset
6 import pandas as pd
7 import torch
8 import torch.nn as nn
9 from sklearn.preprocessing import QuantileTransformer

```

```
10| from sklearn.model_selection import StratifiedShuffleSplit
11| from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
12| import re
13| from model import Net
14| import numpy as np
15| from constants import FILE_PATH, SKIP_ROWS, NUM_ROWS, BATCH_SIZE,
16| NUM_EPOCHS, LEARNING_RATE
17|
18| def fen_to_bit_vector(fen):
19|     parts = re.split(" ", fen)
20|     piece_placement = re.split("/", parts[0])
21|     active_color = parts[1]
22|     castling_rights = parts[2]
23|     en_passant = parts[3]
24|     halfmove_clock = int(parts[4])
25|     fullmove_clock = int(parts[5])
26|
27|     bit_vector = np.zeros((13, 8, 8), dtype=np.uint8)
28|
29|     piece_to_layer = {
30|         'R': 1,
31|         'N': 2,
32|         'B': 3,
33|         'Q': 4,
34|         'K': 5,
35|         'P': 6,
36|         'p': 7,
37|         'k': 8,
38|         'q': 9,
39|         'b': 10,
40|         'n': 11,
41|         'r': 12
42|     }
43|
44|     castling = {
45|         'K': (7, 7),
46|         'Q': (7, 0),
47|         'k': (0, 7),
48|         'q': (0, 0),
49|     }
50|
51|     for r, row in enumerate(piece_placement):
52|         c = 0
53|         for piece in row:
54|             if piece in piece_to_layer:
55|                 bit_vector[piece_to_layer[piece], r, c] = 1
56|                 c += 1
57|             else:
58|                 c += int(piece)
59|
60|             if en_passant != '-':
61|                 bit_vector[0, ord(en_passant[0]) - ord('a'),
62|                           int(en_passant[1]) - 1] = 1
63|
64|             if castling_rights != '-':
65|                 for char in castling_rights:
66|                     bit_vector[0, castling[char][0],
67|                               castling[char][1]] = 1
68|
69|             if active_color == 'w':
70|                 bit_vector[0, 7, 4] = 1
71|             else:
72|                 bit_vector[0, 0, 4] = 1
73|
74|             if halfmove_clock > 0:
75|                 c = 7
```

```
76     while halfmove_clock > 0:
77         bit_vector[0, 3, c] = halfmove_clock % 2
78         halfmove_clock = halfmove_clock // 2
79         c -= 1
80         if c < 0:
81             break
82
83     if fullmove_clock > 0:
84         c = 7
85         while fullmove_clock > 0:
86             bit_vector[0, 4, c] = fullmove_clock % 2
87             fullmove_clock = fullmove_clock // 2
88             c -= 1
89             if c < 0:
90                 break
91
92     return bit_vector
93
94 def import_chess_data(file_path, skip_rows=0, num_rows=None):
95     try:
96         data = pd.read_csv(file_path, skiprows=
97             range(1, skip_rows + 1), nrows=num_rows)
98         return data
99     except FileNotFoundError:
100         print("File not found. Please make sure the file path is correct.")
101     return None
102
103 def preprocess_data2(data):
104     fen_data = data['FEN']
105     evaluation_data = data['Evaluation']
106
107     valid_indices = []
108     processed_evaluation = []
109     for idx, value in enumerate(evaluation_data):
110         if isinstance(value, str) and value.startswith('#'):
111             continue
112         try:
113             int_value = int(value)
114             if int_value != 0:
115                 processed_evaluation.append(int_value)
116                 valid_indices.append(idx)
117             except ValueError:
118                 continue
119
120     processed_evaluation_series = pd.Series(processed_evaluation)
121
122     valid_fen_data = fen_data.iloc[valid_indices].
123     reset_index(drop=True)
124     processed_evaluation_series = processed_evaluation_series.
125     reset_index(drop=True)
126
127     return valid_fen_data, processed_evaluation_series
128
129 def quantile_transform_evaluation(evaluation_data):
130     transformer = QuantileTransformer(
131         output_distribution='uniform')
132     transformed_data = transformer.fit_transform(
133         evaluation_data.values.reshape(-1, 1))
134     return transformed_data
135
136 def bin_evaluation_data(evaluation_data, num_bins=10):
137     """ Bin evaluation data into discrete
138     categories for stratification. """
139     binned_data = pd.cut(evaluation_data, bins=num_bins, labels=False)
140     return binned_data
141
```

```
142 file_path = FILE_PATH
143 skip_rows = SKIP_ROWS
144 num_rows = NUM_ROWS
145 batch_size = BATCH_SIZE
146 num_epochs = NUM_EPOCHS
147 lr = LEARNING_RATE
148
149 class ChessDataset(Dataset):
150     def __init__(self, data_frame):
151         self.data_frame = data_frame
152
153     def __len__(self):
154         return len(self.data_frame)
155
156     def __getitem__(self, index):
157         fen = self.data_frame.iloc[index]["FEN"]
158         eval_ = self.data_frame.iloc[index]["Evaluation"]
159
160         fen_tensor = torch.from_numpy(fen_to_bit_vector(fen)).float()
161         eval_tensor = torch.tensor(eval_).float()
162
163         return fen_tensor, eval_tensor
164
165 def plot_histograms(original_data, transformed_data, output_dir):
166     plt.figure(figsize=(12, 6))
167
168     plt.hist(original_data, bins=50, alpha=0.7,
169             label='Original Data')
170     plt.xlabel('Evaluation')
171     plt.ylabel('Frequency')
172     plt.title('Histogram of Original Evaluation Data')
173     filename_original = os.path.join(output_dir,
174                                     'evaluation_histogram_original.png')
175     plt.legend()
176     plt.savefig(filename_original)
177     plt.show()
178
179     plt.hist(transformed_data, bins=50, alpha=0.7,
180             label='After Quantile Transformation', color='green')
181     plt.xlabel('Evaluation')
182     plt.ylabel('Frequency')
183     plt.title('Histogram of Evaluation Data After Quantile Transformation')
184     filename_transformed = os.path.join(output_dir,
185                                         'evaluation_histogram_transformed.png')
186     plt.legend()
187     plt.savefig(filename_transformed)
188     plt.show()
189
190 def plot_learning_curves(train_losses_mse, val_losses_mse, optimizer,
191 num_epochs, batch_size, train_size, val_size, output_dir):
192     plt.figure(figsize=(12, 6))
193
194     plt.plot(range(1, len(train_losses_mse) + 1), train_losses_mse,
195             label='Training Loss (MSE)')
196     plt.plot(range(1, len(val_losses_mse) + 1), val_losses_mse,
197             label='Validation Loss (MSE)')
198     plt.xlabel('Epochs')
199     plt.ylabel('Loss')
200     plt.title('MSE Learning Curves')
201     plt.legend()
202
203     text = f"Optimizer: {optimizer.__class__.__name__}\nEpochs: {num_epochs}\nBatch Size: {batch_size}\nLearning Rate: {optimizer.param_groups[0]['lr']}\nTraining Data Size: {train_size}\nValidation Data Size: {val_size}"
204     plt.text(1.02, 0.5, text, horizontalalignment='left')
```

```
208     , verticalalignment='center', transform=plt.gca().transAxes)
209
210 plt.subplots_adjust(right=0.7)
211 plt.margins(0.2)
212
213 filename_mse = os.path.join(output_dir, 'mse_learning_curves.png')
214 plt.savefig(filename_mse)
215 plt.show()
216
217 def evaluate_model(model, criterion, data_loader, device):
218     model.eval()
219     running_loss = 0.0
220     all_preds = []
221     all_targets = []
222     with torch.no_grad():
223         for inputs, targets in data_loader:
224             inputs, targets = inputs.to(device), targets.to(device)
225             outputs = model(inputs)
226             loss = criterion(outputs, targets.unsqueeze(1))
227             running_loss += loss.item() * inputs.size(0)
228             all_preds.extend(outputs.cpu().numpy())
229             all_targets.extend(targets.cpu().numpy())
230     epoch_loss = running_loss / len(data_loader.dataset)
231     return epoch_loss, all_preds, all_targets
232
233 def train_model(model, scheduler, criterion, optimizer, train_loader,
234 val_loader, device, batch_size, train_size, val_size, output_dir,
235 num_epochs=10, patience=5):
236     train_losses_mse = []
237     val_losses_mse = []
238     best_val_loss = float('inf')
239     epochs_no_improve = 0
240     early_stop = False
241
242     for epoch in range(num_epochs):
243         if early_stop:
244             print("Early stopping")
245             break
246
247         model.train()
248
249         for inputs, labels in train_loader:
250             inputs, labels = inputs.to(device), labels.to(device)
251             optimizer.zero_grad()
252             outputs = model(inputs)
253             loss_mse = criterion(outputs, labels.float().unsqueeze(1))
254             loss_mse.backward()
255             optimizer.step()
256
257             train_loss_mse, _, _ = evaluate_model(model, criterion,
258             train_loader, device)
259             train_losses_mse.append(train_loss_mse)
260
261             val_loss_mse, _, _ = evaluate_model(model, criterion,
262             val_loader, device)
263             val_losses_mse.append(val_loss_mse)
264
265             scheduler.step(val_loss_mse)
266
267             print(f"Epoch {epoch + 1}/{num_epochs}, Train Loss (MSE): {train_loss_mse:.4f}, Validation Loss (MSE): {val_loss_mse:.4f}")
268
269             if val_loss_mse < best_val_loss:
270                 best_val_loss = val_loss_mse
271                 epochs_no_improve = 0
272             else:
```

```
274         epochs_no_improve += 1
275
276     if epochs_no_improve >= patience:
277         early_stop = True
278
279     plot_learning_curves(train_losses_mse, val_losses_mse,
280     optimizer, num_epochs, batch_size, train_size, val_size, output_dir)
281
282 def main():
283
284     if not os.path.exists('runs'):
285         os.makedirs('runs')
286
287     run_id = 1
288     while os.path.exists(f'runs/train{run_id}'):
289         run_id += 1
290     output_dir = f'runs/train{run_id}'
291     os.makedirs(output_dir)
292
293     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
294     print("Using device:", device)
295
296     chess_data = import_chess_data(file_path, skip_rows, num_rows)
297     fen_data, evaluation_data = preprocess_data2(chess_data)
298
299     print("FEN Data:")
300     print(fen_data.head())
301     print("\nEvaluation Data:")
302     print(evaluation_data.head())
303
304     transformed_evaluation_data = quantile_transform_evaluation(
305     evaluation_data)
306     print("Transformed Evaluation Data:")
307     print(transformed_evaluation_data[:5])
308
309     plot_histograms(evaluation_data, transformed_evaluation_data,
310     output_dir)
311
312     chess_data_valid = pd.DataFrame({'FEN': fen_data, 'Evaluation':
313     transformed_evaluation_data.flatten()})
314
315     chess_data_valid['Evaluation_Binned'] = bin_evaluation_data(
316     chess_data_valid['Evaluation'], num_bins=10)
317
318     stratified_split = StratifiedShuffleSplit(n_splits=1,
319     test_size=0.1, random_state=42)
320     for train_val_index, test_index in stratified_split.
321     split(chess_data_valid, chess_data_valid['Evaluation_Binned']):
322         train_val_data = chess_data_valid.iloc[train_val_index].
323         drop(columns=['Evaluation_Binned'])
324         test_data = chess_data_valid.iloc[test_index].
325         drop(columns=['Evaluation_Binned'])
326
327     stratified_split_val = StratifiedShuffleSplit(n_splits=1,
328     test_size=0.2222, random_state=42)
329     train_val_data['Evaluation_Binned'] = bin_evaluation_data
330     (train_val_data['Evaluation'], num_bins=10)
331     for train_index, val_index in stratified_split_val.split
332     (train_val_data, train_val_data['Evaluation_Binned']):
333         train_data = train_val_data.iloc[train_index].
334         drop(columns=['Evaluation_Binned'])
335         val_data = train_val_data.iloc[val_index].
336         drop(columns=['Evaluation_Binned'])
337
338     train_dataset = ChessDataset(train_data)
339     val_dataset = ChessDataset(val_data)
```

```

340     test_dataset = ChessDataset(test_data)
341
342     train_loader = DataLoader(train_dataset,
343         batch_size=batch_size, shuffle=True, num_workers=0)
344     val_loader = DataLoader(val_dataset,
345         batch_size=batch_size, num_workers=0)
346     test_loader = DataLoader(test_dataset,
347         batch_size=batch_size, num_workers=0)
348
349     model = Net().to(device)
350     criterion = nn.MSELoss()
351     optimizer = torch.optim.Adam(model.parameters(), lr=lr)
352     scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
353         optimizer, mode='min', factor=0.1, patience=2, verbose=True)
354
355     try:
356         train_model(model, scheduler, criterion, optimizer,
357             train_loader, val_loader, device, batch_size, len(train_data),
358             len(val_data), output_dir, num_epochs, patience=5)
359     except Exception as e:
360         print(f"An error occurred: {e}")
361
362     model_path = os.path.join(output_dir, 'chessModel.pth')
363     torch.save(model.state_dict(), model_path)
364
365     test_loss, test_preds, test_targets = evaluate_model(model,
366             criterion, test_loader, device)
367     test_preds = np.array(test_preds).flatten()
368     test_targets = np.array(test_targets).flatten()
369     print(f"Test Loss (MSE): {test_loss:.4f}")
370
371     categories = bin_evaluation_data(pd.Series(test_targets), num_bins=10)
372     pred_categories = bin_evaluation_data(pd.Series(test_preds), num_bins=10)
373
374     cm = confusion_matrix(categories, pred_categories)
375     disp = ConfusionMatrixDisplay(confusion_matrix=cm)
376     disp.plot()
377
378     filename_cm = os.path.join(output_dir, 'confusion_matrix.png')
379     plt.savefig(filename_cm)
380     plt.show()
381
382 if __name__ == "__main__":
383     main()

```

fisierul model.py

```

1 # model.py
2 import torch
3 import torch.nn as nn
4 import torch.nn.functional as F
5
6 class Net(nn.Module):
7     def __init__(self):
8         super(Net, self).__init__()
9         self.conv1 = nn.Conv2d(13, 32, kernel_size=3, padding=1)
10        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
11        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
12
13        self._to_linear = None
14        self._get_conv_output_size()

```

```
15         self.fc1 = nn.Linear(self._to_linear, 128)
16         self.fc2 = nn.Linear(128, 1)
17
18     def _get_conv_output_size(self):
19         x = torch.randn(1, 13, 8, 8)
20         x = F.max_pool2d(F.relu(self.conv1(x)), 2)
21         x = F.max_pool2d(F.relu(self.conv2(x)), 2)
22         x = F.max_pool2d(F.relu(self.conv3(x)), 2)
23         self._to_linear = x.view(x.size(0), -1).size(1)
24
25     def forward(self, x):
26         x = F.relu(self.conv1(x))
27         x = F.max_pool2d(x, 2)
28         x = F.relu(self.conv2(x))
29         x = F.max_pool2d(x, 2)
30         x = F.relu(self.conv3(x))
31         x = F.max_pool2d(x, 2)
32
33         x = x.view(x.size(0), -1)
34
35         x = F.relu(self.fc1(x))
36         x = self.fc2(x)
37         return x
38
39
40 if __name__ == "__main__":
41     net = Net()
42     print(net)
```

fisierul constants.py

```
1 FILE_PATH = "chessData.csv"
2 SKIP_ROWS = 1
3 NUM_ROWS = 15_000_000
4 BATCH_SIZE = 16_000
5 NUM_EPOCHS = 50
6 LEARNING_RATE = 0.001
```

fisierul camera.py

```
1 import cv2
2 import numpy as np
3 from ultralytics import YOLO
4 from PIL import Image, ImageTk
5
6 class CameraProcessor:
7     def __init__(self, camera_index=2, model_paths=None):
8         self.padding = 10
9         self.fen_string = ""
10        self.robot_detected = False
11        self.robot_count = 0
12        self.robot_message = ""
13        self.camera_index = camera_index
14        self.model_corners = YOLO(model_paths['corners'])
15        self.model_pieces = YOLO(model_paths['pieces'])
16        self.model_turn = YOLO(model_paths['turn'])
```

```
17     self.chessboard_image = None
18     self.annotated_frame = None # Initialize annotated_frame here
19     self.model_corners.to('cuda')
20     self.model_pieces.to('cuda')
21     self.model_turn.to('cuda')
22     self.cap = cv2.VideoCapture(self.camera_index)
23     if not self.cap.isOpened():
24         print("Cannot open camera")
25         exit()
26     self.width, self.height = 300, 400
27     self.dst_points = np.array([
28         [0, 0],
29         [self.width - 1, 0],
30         [self.width - 1, self.height - 1],
31         [0, self.height - 1]
32     ], dtype="float32")
33
34     def get_camera_content(self):
35         ret, frame = self.cap.read()
36         if ret:
37             return cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
38         else:
39             return None
40
41     def hex_to_bgr(self, hex_color):
42         hex_color = hex_color.lstrip('#')
43         return tuple(int(hex_color[i:i + 2], 16) for i in (4, 2, 0))
44
45     def draw_chessboard(self, fen, square_size=50,
46                         dark_color="#D18B47", light_color="#FFCE9E"):
47         def hex_to_bgr(hex_color):
48             hex_color = hex_color.lstrip('#')
49             return tuple(int(hex_color[i:i + 2], 16) for i in (4, 2, 0))
50
51         dark_color = hex_to_bgr(dark_color)
52         light_color = hex_to_bgr(light_color)
53         board_size = square_size * 8
54         chessboard_image = np.zeros((board_size, board_size, 3), dtype=np.uint8)
55
56         piece_paths = {
57             'P': 'images/white/Pawn.png', 'R': 'images/white/Rook.png',
58             'N': 'images/white/Knight.png',
59             'B': 'images/white/Bishop.png', 'Q': 'images/white/Queen.png',
60             'K': 'images/white/King.png',
61             'p': 'images/black/Pawn.png', 'r': 'images/black/Rook.png', 'n':
62             'images/black/Knight.png',
63             'b': 'images/black/Bishop.png', 'q': 'images/black/Queen.png',
64             'k': 'images/black/King.png'
65         }
66
67         # Draw the chessboard squares
68         for i in range(8):
69             for j in range(8):
70                 y, x = i * square_size, j * square_size
71                 color = light_color if (i + j) % 2 == 0 else dark_color
72                 cv2.rectangle(chessboard_image, (x, y), (x + square_size,
73 y + square_size), color, -1)
74
75         # Print the FEN for debugging
76         print(f"FEN: {fen}")
77
78         # Draw the pieces
79         rows = fen.split('/')
80         for i, row in enumerate(rows):
81             col = 0
82             for char in row:
```

```
83         if char.isdigit():
84             col += int(char)
85         else:
86             piece_path = piece_paths.get(char, None)
87             if piece_path:
88                 piece_img = cv2.imread(piece_path, cv2.
89                 IMREAD_UNCHANGED)
90                 if piece_img is None:
91                     print(f"Could not load image for piece:
92 {char} at {i},{col}")
93                     continue
94                 piece_img = cv2.resize(piece_img, (square_size,
95                 square_size))
96                 y, x = i * square_size, col * square_size
97                 for c in range(3):
98                     chessboard_image[y:y + square_size, x +
99                     square_size, c] = \
100                         piece_img[:, :, c] * (piece_img[:, :, 3]
101                         / 255.0) + \
102                         chessboard_image[y:y + square_size, x
103                         + square_size, c] * (
104                             1.0 - piece_img[:, :, 3] / 255.0)
105                         col += 1
106
107             return chessboard_image
108
109     def construct_fen(self, chessboard_matrix):
110         piece_map = {
111             "WP": "P", "WR": "R", "WN": "N", "WB":
112             "B", "WQ": "Q", "WK": "K",
113             "BP": "p", "BR": "r", "BN": "n", "BB":
114             "b", "BQ": "q", "BK": "k",
115             "Background": "0"
116         }
117
118         fen_rows = []
119         for row in chessboard_matrix:
120             fen_row = ""
121             empty_count = 0
122             for cell in row:
123                 if cell == "Background":
124                     empty_count += 1
125                 else:
126                     if empty_count > 0:
127                         fen_row += str(empty_count)
128                         empty_count = 0
129                     fen_row += piece_map.get(cell, cell)
130             if empty_count > 0:
131                 fen_row += str(empty_count)
132             fen_rows.append(fen_row)
133
134         fen = "/".join(fen_rows)
135         return fen
136
137     def order_points_tuples(self, pts):
138         pts_np = np.array(pts, dtype="float32")
139         rect = np.zeros((4, 2), dtype="float32")
140         s = pts_np.sum(axis=1)
141         rect[0] = pts_np[np.argmin(s)]
142         rect[2] = pts_np[np.argmax(s)]
143         diff = np.diff(pts_np, axis=1)
144         rect[1] = pts_np[np.argmin(diff)]
145         rect[3] = pts_np[np.argmax(diff)]
146         ordered_points = [tuple(point) for point in rect]
147         return ordered_points
148
```

```
149     def process_frame(self, frame):
150         gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
151         gray_frame_3ch = cv2.cvtColor(gray_frame, cv2.COLOR_GRAY2BGR)
152
153         results_corners = self.model_corners(gray_frame_3ch)
154         centers = []
155
156         boxes_with_confidence = [(box.xyxy[0], box.conf)
157             for result in results_corners for box in result.boxes]
158         sorted_boxes = sorted(boxes_with_confidence,
159             key=lambda x: x[1], reverse=True)
160
161         for i in range(min(4, len(sorted_boxes))):
162             x1, y1, x2, y2 = map(int, sorted_boxes[i][0])
163             center_x = (x1 + x2) // 2
164             center_y = (y1 + y2) // 2
165             centers.append((center_x, center_y))
166
167         if len(centers) == 4:
168             corners = self.order_points_tuples(centers)
169             src_points = np.array(corners, dtype="float32")
170             M = cv2.getPerspectiveTransform(src_points,
171                 self.dst_points)
172
173             if np.linalg.cond(M) < 1 / np.finfo(M.dtype).eps:
174                 M_inv = np.linalg.inv(M)
175                 warped_image = cv2.warpPerspective(frame, M,
176                     (self.width, self.height))
177
178             grid_rows = 8
179             grid_cols = 8
180             chessboard_grid = [[None] * grid_cols for _ in range(grid_rows)]
181
182             for i in range(grid_rows):
183                 for j in range(grid_cols):
184                     top_left = (int(self.width * j / grid_cols),
185                         int(self.height * i / grid_rows))
186                     top_right = (int(self.width * (j + 1) / grid_cols),
187                         int(self.height * i / grid_rows))
188                     bottom_right = (int(self.width * (j + 1) / grid_cols),
189                         int(self.height * (i + 1) / grid_rows))
190                     bottom_left = (int(self.width * j / grid_cols),
191                         int(self.height * (i + 1) / grid_rows))
192
193                     top_left_original = cv2.perspectiveTransform(
194                         np.array([[top_left]]), dtype='float32'), M_inv)[0][0]
195                     top_right_original = \
196                         cv2.perspectiveTransform(np.array([[top_right]]),
197                             dtype='float32'), M_inv)[0][0]
198                     bottom_right_original = \
199                         cv2.perspectiveTransform(np.array([[bottom_right]]),
200                             dtype='float32'), M_inv)[0][0]
201                     bottom_left_original = \
202                         cv2.perspectiveTransform(np.array([[bottom_left]]),
203                             dtype='float32'), M_inv)[0][0]
204
205                     chessboard_grid[i][j] = (
206                         top_left_original, top_right_original,
207                         bottom_right_original, bottom_left_original)
208
209             all_squares = np.zeros((self.height + 7 * self.padding,
210                 self.width + 7 * self.padding, 3),
211                             dtype=np.uint8)
```

```
215     chessboard_matrix = [["0" for _ in range(8)] for _ in range(8)]
216
217     for i in range(grid_rows):
218         for j in range(grid_cols):
219             square_coords = np.array(chessboard_grid[i][j], np.int32)
220             square_coords = square_coords.reshape((-1, 1, 2))
221
222             top_left, top_right, bottom_right, bottom_left =
223             chessboard_grid[i][j]
224             src_points_square = np.array([top_left, top_right,
225                 bottom_right, bottom_left], dtype="float32")
226             dst_points_square = np.array([
227                 [0, 0],
228                 [top_right[0] - top_left[0], 0],
229                 [top_right[0] - top_left[0], bottom_left[1] -
230                 top_left[1]],
231                 [0, bottom_left[1] - top_left[1]]
232             ], dtype="float32")
233             M_square = cv2.getPerspectiveTransform(
234                 src_points_square, dst_points_square)
235             square_img = cv2.warpPerspective(frame, M_square,
236                 (int(dst_points_square[2][0]),
237                  int(dst_points_square[2][1])))
238
239             resized_square_img = cv2.resize(square_img,
240                 (int(self.width / grid_cols),
241                  int(self.height / grid_rows)))
242
243
244             all_squares[i * (int(self.height / grid_rows) +
245 self.padding):(i + 1) * int(
246                 self.height / grid_rows) + i * self.padding,
247                 j * (int(self.width / grid_cols) + self.padding):(j + 1) *
248             int(
249                 self.width / grid_cols) + j * self.padding] =
250             resized_square_img
251
252             results_pieces = self.model_pieces(resized_square_img)
253
254             if isinstance(results_pieces, list):
255                 results = results_pieces[0]
256
257                 class_names = results.names
258                 top_class_index = results.probs.top1
259                 top_class_confidence = results.probs.top1conf.item()
260
261                 max_class_name = class_names[top_class_index]
262
263                 chessboard_matrix[i][j] = max_class_name
264
265                 fen_part = self.construct_fen(chessboard_matrix)
266                 self.chessboard_image = self.draw_chessboard(fen_part,
267                     dark_color="#8B4513", light_color="#DEB887")
268                 self.fen_string = fen_part
269
270                 results_turn = self.model_turn(gray_frame_3ch)
271                 robot_found = False
272                 for result in results_turn:
273                     for box in result.boxes:
274                         x1, y1, x2, y2 = map(int, box.xyxy[0])
275                         confidence = box.conf[0]
276                         class_id = box.cls[0]
277                         class_name = self.model_turn.names[int(class_id)]
278
279                         if confidence > 0.0:
280                             cv2.rectangle(frame, (x1, y1), (x2, y2), (255, 0, 0), 2)
```

```

281             cv2.putText(frame, class_name, (x1, y1 - 10),
282                         cv2.FONT_HERSHEY_SIMPLEX, 0.9, (255, 0, 0), 2)
283             if class_name == "Robot":
284                 robot_found = True
285
286             if robot_found and not self.robot_detected:
287                 self.robot_count += 1
288                 self.robot_message += f"Robot found {self.robot_count} times\n"
289                 self.robot_detected = True
290             elif not robot_found:
291                 self.robot_detected = False
292
293             self.annotated_frame = frame
294
295     def get_annotated_frame(self):
296         return self.annotated_frame
297
298     def get_robot_message(self):
299         return self.robot_message
300
301     def get_fen_string(self):
302         return self.fen_string
303
304     def run(self, label):
305         def update_label():
306             ret, frame = self.cap.read()
307             if ret:
308                 self.process_frame(frame)
309                 cv2image = cv2.cvtColor(frame, cv2.COLOR_BGR2RGBA)
310                 img = Image.fromarray(cv2image)
311                 imgtk = ImageTk.PhotoImage(image=img)
312                 label.imgtk = imgtk
313                 label.config(image=imgtk)
314                 label.after(10, update_label)
315
316         update_label()

```

fisierul main.py

```

1 import chess
2 import cv2
3 import torch
4 from model import Net
5 import ChessGame
6 import tkinter as tk
7 from PIL import Image, ImageTk
8 import threading
9 from camera import CameraProcessor
10
11 def parse_fen(fen):
12     rows = fen.split(' ')[0].split('/')
13     board = []
14     for row in rows:
15         parsed_row = []
16         for char in row:
17             if (char.isdigit()):
18                 parsed_row.extend([' '] * int(char))
19             else:
20                 parsed_row.append(char)
21         board.append(parsed_row)
22
23     return board

```

```
23
24 def get_move(old_fen, new_fen):
25     old_board = parse_fen(old_fen)
26     new_board = parse_fen(new_fen)
27     start = None
28     end = None
29
30     print("Old Board:")
31     for row in old_board:
32         print(row)
33     print("New Board:")
34     for row in new_board:
35         print(row)
36
37     for r in range(8):
38         for c in range(8):
39             if old_board[r][c] != new_board[r][c]:
40                 if old_board[r][c] != '' and new_board[r][c] == '':
41                     start = (r, c)
42                 elif old_board[r][c] == '' and new_board[r][c] != '':
43                     end = (r, c)
44                 elif old_board[r][c] != '' and new_board[r][c] != '':
45                     and old_board[r][c] != new_board[r][c]:
46                     end = (r, c)
47
48     if start is None or end is None:
49         print(f"Start or end is None: start={start}, end={end}")
50     return None, None
51
52     print(f"Start: {start}, End: {end}")
53
54     def index_to_square(index):
55         files = 'abcdefgh'
56         ranks = '87654321'
57         return files[index[1]] + ranks[index[0]]
58
59     start_square = index_to_square(start)
60     end_square = index_to_square(end)
61
62     if old_board[start[0]][start[1]].lower() == 'k' and abs(end[1] -
63     start[1]) == 2:
64         if end_square == 'g1' or end_square == 'g8':
65             return ("e1", "g1") if start_square == "e1" else ("e8", "g8")
66         elif end_square == 'c1' or end_square == 'c8':
67             return ("e1", "c1") if start_square == "e1" else ("e8", "c8")
68
69     return start_square, end_square
70
71 def setup_ui(root, board_image, camera_processor, chessGame):
72     for i in range(2):
73         root.grid_rowconfigure(i, weight=1, minsize=400)
74         root.grid_columnconfigure(i, weight=1, minsize=400)
75
76     photo = ImageTk.PhotoImage(board_image)
77
78     label_section1 = tk.Label(root, image=photo)
79     label_section1.image = photo
80     label_section1.grid(row=0, column=0, sticky="nsew")
81
82     section2_frame = tk.Frame(root, bg="lightgray")
83     section2_frame.grid(row=0, column=1, sticky="nsew")
84
85     section2_label = tk.Label(section2_frame)
86     section2_label.pack(expand=True, fill="both")
87
88     section3_label = tk.Label(root)
```

```
89     section3_label.grid(row=1, column=0, sticky="nsew")
90
91     section4 = tk.Frame(root, bg="lightgray")
92     section4.grid(row=1, column=1, sticky="nsew")
93
94     section4_text = tk.Text(section4, bg="lightgray")
95     section4_text.pack(expand=True, fill="both")
96
97     def update_camera_content():
98
99         annotated_frame = camera_processor.get_annotated_frame()
100        if annotated_frame is not None:
101            camera_image = Image.fromarray(cv2.cvtColor(annotated_frame,
102                                            cv2.COLOR_BGR2RGB))
103            label_width = section2_label.winfo_width()
104            label_height = section2_label.winfo_height()
105            camera_image.thumbnail((label_width, label_height),
106                                   Image.LANCZOS)
107            photo = ImageTk.PhotoImage(camera_image)
108
109            section2_label.config(image=photo)
110            section2_label.image = photo
111
112        chessboard_image = camera_processor.chessboard_image
113        if chessboard_image is not None:
114            chessboard_image_pil = Image.fromarray(chessboard_image)
115            chessboard_photo = ImageTk.PhotoImage(chessboard_image_pil)
116            section3_label.config(image=chessboard_photo)
117            section3_label.image = chessboard_photo
118
119        current_board_image = chessGame.get_board_image()
120        current_board_photo = ImageTk.PhotoImage(current_board_image)
121        label_section1.config(image=current_board_photo)
122        label_section1.image = current_board_photo
123
124    def resize_handler(event):
125        update_camera_content()
126
127    section2_label.bind("<Configure>", resize_handler)
128
129    return update_camera_content, section4_text, label_section1
130
131 def process_frames(camera_processor, root, update_camera_content,
132 section4_text, chessGame):
133     previous_robot_count = camera_processor.robot_count
134
135     while True:
136         ret, frame = camera_processor.cap.read()
137         if ret:
138             camera_processor.process_frame(frame)
139             root.after(10, update_camera_content)
140             previous_fen = chessGame.gs.board.fen()
141
142             if camera_processor.robot_count > previous_robot_count:
143                 previous_robot_count = camera_processor.robot_count
144                 fen_string = camera_processor.get_fen_string()
145                 start_square, end_square = get_move(previous_fen,
146                                           fen_string)
147
148                 if start_square and end_square:
149                     print("Start: ", start_square, "end: ", end_square)
150
151                     move_notation = (start_square + end_square).lower()
152                     section4_text.insert(tk.END, f"White move:
153 {move_notation}\n")
154                     section4_text.see(tk.END)
```

```

155
156         move = chess.Move.from_uci(move_notation)
157
158     if move in chessGame.gs.board.legal_moves:
159         chessGame.gs.board.push(move)
160         chessGame.update_board(move)
161         chessGame.gs.white_to_move = False
162
163         previous_fen = chessGame.gs.board.fen()
164
165         black_move = chessGame.play_black_turn(model)
166
167         fen_string_after_black = chessGame.gs.board.fen()
168
169         black_start_square, black_end_square =
170         get_move(previous_fen, fen_string_after_black)
171
172         if black_start_square and black_end_square:
173             if chessGame.gs.board.is_capture(chess.Move.from_uci
174             (black_start_square + black_end_square)):
175                 section4_text.insert(tk.END, f"Black move:
176                 {black_end_square}outside\n")
177                 section4_text.insert(tk.END, f"Black move:
178                 {black_start_square}{black_end_square}\n")
179                 section4_text.see(tk.END)
180
181         else:
182             pass
183
184 model = torch.load('chessModel.pth')
185 model.eval()
186
187 def main():
188     chessGame = ChessGame.ChessGame()
189     print(chessGame.gs.board.fen())
190     chessGame.create_ui()
191
192     board_image = chessGame.get_board_image()
193
194     root = tk.Tk()
195     root.title("Chess Board")
196     root.geometry("800x800")
197
198     model_paths = {
199         'corners': 'runs/detect/train10/weights/best.pt',
200         'pieces': 'runs/classify/train30/weights/best.pt',
201         'turn': 'runs/detect/train47/weights/best.pt'
202     }
203     camera_processor = CameraProcessor(model_paths=model_paths)
204
205     update_camera_content, section4_text, label_section1 = setup_ui(root,
206     board_image, camera_processor, chessGame)
207
208     camera_thread = threading.Thread(target=process_frames,
209                                         args=(camera_processor, root,
210                                               update_camera_content, section4_text,
211                                               chessGame))
212     camera_thread.daemon = True
213     camera_thread.start()
214
215     root.mainloop()
216
217 if __name__ == "__main__":
218     main()

```

fisierul augment-dataset.py

```

1 import os
2 import cv2
3 import albumentations as A
4 import numpy as np
5
6 transform = A.Compose([
7     A.RandomBrightnessContrast(brightness_limit=0.2,
8         contrast_limit=0.2, p=1.0),
9     A.Affine(rotate=(-180, 180), p=1.0),
10    A.Rotate(limit=360, p=1.0),
11    A.Perspective(scale=(0.05, 0.1), p=0.5),
12 ], bbox_params=A.BboxParams(format='yolo',
13 label_fields=['class_labels']))
14
15 def read_yolo_annotation(annotation_path):
16     with open(annotation_path, 'r') as f:
17         annotations = f.readlines()
18     bboxes = []
19     class_labels = []
20     for annotation in annotations:
21         class_id, x_center, y_center, width, height =
22             map(float, annotation.strip().split())
23         bboxes.append([x_center, y_center, width, height])
24         class_labels.append(int(class_id))
25     return bboxes, class_labels
26
27 def write_yolo_annotation(annotation_path, bboxes, class_labels):
28     with open(annotation_path, 'w') as f:
29         for bbox, class_label in zip(bboxes, class_labels):
30             f.write(f"{class_label} {' '.join(map(str, bbox))}\n")
31
32 def augment_image(image_path, annotation_path,
33 save_image_path, save_annotation_path, num_augmentations=5):
34     image = cv2.imread(image_path)
35     if image is None:
36         print(f"Could not read the image from {image_path}")
37         return
38
39     bboxes, class_labels = read_yolo_annotation(annotation_path)
40     if not bboxes:
41         print(f"No bounding boxes found for {image_path}")
42         return
43     print(f"Read {len(bboxes)} annotations from {annotation_path}")
44
45     for i in range(num_augmentations):
46         augmented = transform(image=image, bboxes=bboxes,
47             class_labels=class_labels)
48         augmented_image = augmented['image']
49         augmented_bboxes = augmented['bboxes']
50         augmented_class_labels = augmented['class_labels']
51
52         base_name = os.path.basename(image_path)
53         name, ext = os.path.splitext(base_name)
54         new_image_name = f"{name}_aug_{i}{ext}"
55         new_image_path = os.path.join(save_image_path, new_image_name)
56         new_annotation_path = os.path.join(save_annotation_path,
57             f"{name}_aug_{i}.txt")
58
59         cv2.imwrite(new_image_path, augmented_image)
60         write_yolo_annotation(new_annotation_path,
61             augmented_bboxes, augmented_class_labels)
62         print(f"Saved augmented image and annotation:
63             {new_image_path}, {new_annotation_path}")
64

```

```

65 def augment_folder(image_folder_path, label_folder_path, save_image_path,
66 save_annotation_path, num_augmentations=5):
67     if not os.path.exists(image_folder_path):
68         print(f"Image folder not found: {image_folder_path}")
69         return
70
71     if not os.path.exists(label_folder_path):
72         print(f"Label folder not found: {label_folder_path}")
73         return
74
75     for file in os.listdir(image_folder_path):
76         if file.lower().endswith('.png', '.jpg', '.jpeg'):
77             image_path = os.path.join(image_folder_path, file)
78             annotation_path = os.path.join(label_folder_path,
79             os.path.splitext(file)[0] + '.txt')
80             if os.path.exists(annotation_path):
81                 print(f"Processing {image_path} and {annotation_path}")
82                 augment_image(image_path, annotation_path, save_image_path,
83                 save_annotation_path, num_augmentations)
84             else:
85                 print(f"Annotation not found for {image_path}")
86
87 def augment_dataset(dataset_path, num_augmentations=5):
88     for split in ['train', 'test']:
89         image_folder_path = os.path.join(dataset_path, split, 'images')
90         label_folder_path = os.path.join(dataset_path, split, 'labels')
91         save_image_path = image_folder_path
92         save_annotation_path = label_folder_path
93         print(f"Augmenting dataset split: {split}")
94         augment_folder(image_folder_path, label_folder_path, save_image_path,
95                     save_annotation_path, num_augmentations)
96
97 dataset_path = 'D:\PycharmProjects\CameraSetup\corner-dataset'
98 augment_dataset(dataset_path, 20)

```

fisierul detect-chessboard.py

```

1 import cv2
2 import numpy as np
3 from ultralytics import YOLO
4
5 model_corners = YOLO('runs/detect/train10/weights/best.pt')
6 model_corners.to('cuda')
7
8 def order_points_tuples(pts):
9     pts_np = np.array(pts, dtype="float32")
10    rect = np.zeros((4, 2), dtype="float32")
11    s = pts_np.sum(axis=1)
12    rect[0] = pts_np[np.argmin(s)]
13    rect[2] = pts_np[np.argmax(s)]
14    diff = np.diff(pts_np, axis=1)
15    rect[1] = pts_np[np.argmin(diff)]
16    rect[3] = pts_np[np.argmax(diff)]
17    ordered_points = [tuple(point) for point in rect]
18    return ordered_points
19
20 cap = cv2.VideoCapture(2)
21
22 if not cap.isOpened():
23     print("Cannot open camera")
24     exit()

```

```
25
26 width, height = 800, 800
27 dst_points = np.array([
28     [0, 0],
29     [width - 1, 0],
30     [width - 1, height - 1],
31     [0, height - 1]
32 ], dtype="float32")
33
34 while True:
35     ret, frame = cap.read()
36     if not ret:
37         print("Failed to grab frame")
38         break
39
40     gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
41     gray_frame_3ch = cv2.cvtColor(gray_frame, cv2.COLOR_GRAY2BGR)
42
43     results_corners = model_corners(gray_frame_3ch)
44     centers = []
45
46     boxes_with_confidence = [(box.xyxy[0], box.conf) for result
47                               in results_corners for box in result.boxes]
48
49     sorted_boxes = sorted(boxes_with_confidence, key=lambda x:
50                          x[1], reverse=True)
51
52     for i in range(min(4, len(sorted_boxes))):
53         x1, y1, x2, y2 = map(int, sorted_boxes[i][0])
54         center_x = (x1 + x2) // 2
55         center_y = (y1 + y2) // 2
56         centers.append((center_x, center_y))
57
58         cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)
59
60     if len(centers) == 4:
61         corners = order_points_tuples(centers)
62         src_points = np.array(corners, dtype="float32")
63         M = cv2.getPerspectiveTransform(src_points, dst_points)
64
65         if np.linalg.cond(M) < 1 / np.finfo(M.dtype).eps:
66             M_inv = np.linalg.inv(M)
67             warped_image = cv2.warpPerspective(frame, M,
68                     (width, height))
69
70             for i in range(9):
71                 x = i * (width // 8)
72                 y = i * (height // 8)
73                 cv2.line(warped_image, (x, 0), (x, height), (255, 255, 255), 1)
74                 cv2.line(warped_image, (0, y), (width, y), (255, 255, 255), 1)
75
76             grid_points = []
77             for i in range(9):
78                 for j in range(9):
79                     grid_points.append((i * (width // 8), j * (height // 8)))
80
81             for point in grid_points:
82                 cv2.circle(warped_image, point, 3, (0, 0, 255), -1)
83
84             grid_points_original = cv2.perspectiveTransform(
85                 np.array([grid_points]), dtype='float32'), M_inv)[0]
86
87             for i in range(9):
88                 pt1 = tuple(map(int, grid_points_original[i * 9]))
89                 pt2 = tuple(map(int, grid_points_original[(i + 1) * 9 - 1]))
90                 cv2.line(frame, pt1, pt2, (255, 0, 0), 1)
```

```
91         for i in range(9):
92             pt1 = tuple(map(int, grid_points_original[i]))
93             pt2 = tuple(map(int, grid_points_original[i + 72]))
94             cv2.line(frame, pt1, pt2, (255, 0, 0), 1)
95
96             cv2.imshow('Warped Chessboard', warped_image)
97
98     height, width = frame.shape[:2]
99     if height > width:
100         new_height = 800
101         new_width = int((width / height) * 800)
102     else:
103         new_width = 800
104         new_height = int((height / width) * 800)
105     frame_resized = cv2.resize(frame, (new_width, new_height))
106
107     cv2.imshow('YOLOv8 Corner Detection', frame_resized)
108     if cv2.waitKey(1) & 0xFF == ord('q'):
109         break
110
111 cap.release()
112 cv2.destroyAllWindows()
```

fisierul piece-detect.py

```
1 import cv2
2 import numpy as np
3 from ultralytics import YOLO
4
5 model = YOLO('runs/detect/train42/weights/best.pt')
6 model.to('cuda')
7
8 cap = cv2.VideoCapture(2)
9
10 if not cap.isOpened():
11     print("Cannot open camera")
12     exit()
13
14 while True:
15     ret, frame = cap.read()
16     if not ret:
17         print("Failed to grab frame")
18         break
19
20     gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
21
22     gray_frame = cv2.normalize(gray_frame, None, 0, 255,
23     cv2.NORM_MINMAX)
24
25     blurred_frame = cv2.GaussianBlur(gray_frame, (5, 5), 0)
26
27     filtered_frame_stacked = np.stack((blurred_frame,) * 3, axis=-1)
28
29     results = model(filtered_frame_stacked)
30
31     filtered_frame_display = cv2.cvtColor(blurred_frame,
32     cv2.COLOR_GRAY2BGR)
33
34     centers = []
35
36     for result in results:
```

```
37     for box in result.boxes:
38         if box.conf[0] > 0:
39             x1, y1, x2, y2 = map(int, box.xyxy[0])
40             confidence = box.conf[0]
41             class_id = box.cls[0]
42             class_name = model.names[int(class_id)]
43
44             cv2.rectangle(filtered_frame_display,
45                           (x1, y1), (x2, y2), (0, 255, 0), 2)
46
47             text = f'{class_name}: {confidence:.2f}'
48             cv2.putText(filtered_frame_display, text,
49                         (x1, y1 - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)
50
51             center_x = (x1 + x2) // 2
52             center_y = (y1 + y2) // 2
53             centers.append((center_x, center_y))
54
55 cv2.imshow('YOLOv8 Detection', filtered_frame_display)
56
57 if cv2.waitKey(1) & 0xFF == ord('q'):
58     break
59
60 cap.release()
61 cv2.destroyAllWindows()
```

fisierul train_chessboard.py

```
1 from ultralytics import YOLO
2 import torch
3
4 def main():
5     model = YOLO('yolov8n.pt')
6     torch.cuda.empty_cache()
7     model.train(data='corner-dataset/data.yaml', epochs=100,
8                  imgsz=640, device='cuda', amp=False, patience=5)
9
10 if __name__ == '__main__':
11     main()
```

fisierul train_pieces.py

```
1 import torch
2 from ultralytics import YOLO
3
4 def main():
5     model = YOLO('yolov8n-cls.pt')
6     torch.cuda.empty_cache()
7
8     data_path = "D:\PycharmProjects\CameraSetup\chess-piece-classification-v6"
9     print(f"Training data path: {data_path}")
10
11     model.train(data=data_path, epochs=10, imgsz=96, device='cuda',
12                  amp=False, patience=5, dropout=0.5, weight_decay=0.001, cos_lr=True)
13
14 if __name__ == '__main__':
```

15 | main()

fisierul train_turn.py

```
1 import multiprocessing
2 from multiprocessing import freeze_support
3
4 from ultralytics import YOLO
5
6 def main():
7     model = YOLO('yolov8n.pt')
8     model.train(data='logo-detect.v5/data.yaml', epochs=200,
9                  imgsz=640, device='cuda', amp=False, patience=5)
10
11 if __name__ == '__main__':
12     main()
```