UNIVERSITATEA DIN BUCUREŞTI

FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licenţă

METHODS FOR DETECTING GLITCHES IN VIDEO GAMES

Absolvent

Nacu Andrei-Emilian

Coordonator ştiinţific

Păduraru Ciprian

Bucureşti, iunie 2023

Rezumat

Jocurile video sunt programe destinate divertismentului. Din cauza complexității lor, pot exista destule erori care să fie neidentificate și să afecteze negativ experiența unui jucător. În această lucrare propun o metodă de detecție a glitchurilor în jocurile video care folosește tehnici specifice din domeniul de deep learning.

Primul pas îl reprezintă generarea unui set de date sintetic cu ajutorul aplicației Unity. Sunt generate 2500 de imagini pe baza a 10 obiecte ce sunt categorisite de una dintre cele cinci etichete folosite: „Normal”, „Low”, „Stretched”, „Missing” sau „Placeholder”. Au fost testate 4 arhitecturi diferite: ResNet18, Resnet34, ResNet50 și ShuffleNetV2, cea mai bună acuratețe obținută la validare fiind de 99,6%, pe parcursul a 100 de epoci. În plus, în perioada de testare, doar două din cele 250 de imagini folosite sunt etichetate greșit de către ResNet50, iar precizia și recallul ating valori aproape maxime pentru fiecare dintre cele 5 clase.

Rezultatele sunt analizate și comparate cu cele obținute în alte lucrări similare. De asemenea, sunt discutate și abordări diferite pentru detecția glitchurilor, unde sunt prezente tipuri noi de erori sau alte tipuri de jocuri și metode specifice inteligenței artificiale.

Abstract

Video games are programs intended for entertainment. Due to their complexity, there can be quite a few errors that may be unnoticed and negatively affect a player’s experience. In this paper I propose a method for detecting glitches in video games that uses specific techniques from deep learning.

The first step consists in the generation of a synthetic dataset using Unity engine. 2500 images from 10 different objects are generated, which are labeled by „Normal”, „Low”, „Stretched”, „Missing” or „Placeholder”. There are 4 different architectures tested: ResNet18, ResNet34, ResNet50 and ShuffleNetV2, the best accuracy score for validation being 99,6% during 100 epochs. In addition, throughout the testind period, only 2 of the 250 images used are wrongly labeled by ResNet50 and the precision and recall reach values that are close to the maximum for each of the 5 classes.

The results are analyzed and compared with those obtained in similar papers. Furthermore, different aproaches are discussed, where new types of errors or other types of games and methods specific to artificial intelligence are found.

Cuprins

1. Introducere

1.1. Motivație

Pasiunea pentru jocurile video a început să se dezvolte devreme în viața mea. Am început cu jocuri simple pe telefon precum „Snake”, ca apoi să am acces la primul meu calculator cu internet, unde posibilitatea de a găsi un joc nou era și încă este nelimitată. Odată cu înscrierea la liceu și studierea informaticii, mi-am dorit să înțeleg cum este creat un joc și să încerc să creez și eu unul.

La facultate am avut posibilitatea să urmez un internship în Machine Learning. Am fost încă de la început fascinat de complexitatea domeniului și de practicitatea sa, observând că singur reușisem să implementez un model care să facă anumite predicții pe un set de date. Inițial nu am putut să observ legătura între inteligența artificială și industria de gaming, dar apoi am aflat că anumite elemente din jocuri pot fi programate într-un mod automatizat, capabile de a lua decizii proprii. Un exemplu simplu este robotul sau calculatorul căruia îi poți fi adversar la șah și care poate fi antrenat să câștige în majoritatea cazurilor, având posibilitatea să analizeze mult mai multe mișcări într-un timp mult mai scurt decât cel necesar unui om.

Un alt factor care a contribuit la alegerea temei propuse a fost faptul că am întâlnit de foarte multe ori glitchuri care să-mi altereze negativ experiența unui joc, de la cele care apar ca un mic defect vizual, până la cele care blochează la propriu progresul jocului. Cele din urmă se rezolvau de obicei printr-un restart sau căutarea pe un forum unde cineva găsise o modalitate de a rezolva problema la nivel de program sau propunea o abordare artificială a nivelului pentru a sări peste impasul respectiv. De câteva ori am fost determinat să închid jocul pentru o perioadă sau să renunț la el complet. Povestea nu mai putea fi finalizată și atunci întreg progresul stagna sau nu mai urma să fie continuat, iar resursele personale precum banii sau timpul investit într-un anumit joc erau risipite.

Prin alegerea subiectului propus pentru lucrarea de licență, pot să îmbin cele două domenii tehnologice de care sunt pasionat și să contribui la o temă care se află încă la un nivel scăzut de abordare.

1.2. Prezentarea temei

Tema propune discutarea unor metode pentru detecția glitchurilor într-un joc, adică a erorilor de programare de obicei întâlnite la nivel vizual sau de grafică. Vor fi discutate în principal soluții particulare domeniului de deep learning, dar vor fi explorate succint și alte posibilități precum cele din reinforcement learning.

Găsirea glitchurilor poate fi complicată deoarece unele dintre ele necesită acțiuni specifice desfășurate în mod manual de către un tester de joc (game tester). Acțiunile constau în secvențe de mișcări precum alergatul sau ghemuitul unui caracter înspre anumite obiecte din joc pentru câteva clipe [1]. Acest gen de tipar poate fi greu de găsit deoarece nu reprezintă mișcări uzuale necesare pentru completarea unei anumite secțiuni din joc. Una dintre cele mai cunoscute metode pentru detectarea glitchurilor o reprezintă smoke testingul. Diferite clipuri video sunt captate de către o cameră care traveresează mediul jocului, urmând să fie evaluate de către o persoană pentru a fi identificate anumite erori și a fi corectate [2]. Procesul poate consuma destul timp pentru foarte multe date, de aceea fiind folosit un număr mic de exemple, lăsând anumite aspecte negative ale jocului nedetectate.

În lucrare vă fi detaliată o metodă care folosește un DCNN (Deep convolutional neural network) propriu și vor fi prezentate rezultatele personale, dar și comparații cu alte lucrări pe aceasta temă.

1.2.1. Introducere în jocurile video

Un joc video este un joc care poate fi jucat cu ajutorul unui calculator și care poate fi bazat pe o poveste [3]. Este un program complex la care pot contribui atât câțiva programatori, cât și zeci sau sute în funcție de bugetul alocat proiectului. Elemente cheie precum gameplay-ul, povestea și designul de artă și grafică sunt necesare pentru crearea unui produs final în care nu numai experții în a scrie cod sunt esențiali, dar și persoanele cu rol de povestitor, ilustrator sau chiar actori de scenă sau vocali.

Jocurile video se bucură de un imens succes raportat la timpul scurt pe care l-au avut pentru a se dezvolta. Crescând în popularitate din anii ’70-’80 , activitatea numită în mod popular ”gaming” a reușit să construiască mici comunități ale căror membrii urmau să petreacă ore în sir în fața ecranelor, împărtășind noua formă de entertainment. Astăzi au loc evenimente de mare amploare bazate pe competiții în Esports (sporturi electronice) la care participă sute de mii de privitori. Echipe oficiale cu sponsori și

antrenori, asemenea echipelor din sporturi obișnuite, luptă în diferite campionate unde premiile pot ajunge la valori de până la câteva milioane de dolari [4].

Date fiind popularitatea și importanța lor, este esențial ca produsul final să fie livrat complet și lipsit de erori sau cel puțin de un număr mare dintre ele. Prezența greșelilor de programare în ansamblul unui joc pot altera rezultatul dorit și contribuie la scăderea atenției jucătorului asupra mediului virtual.

1.2.2. Introducere în deep learning

Deep learning reprezintă o formă de machine learning prin care calculatoarele pot învăța prin experiență și pot înțelege lumea datorită unei ierarhii de concepte [5]. Rețelele încearcă să simuleze modul în care neuronii creierului funcționează, dar la un nivel mult mai simplu. Cu ajutorul mai multor straturi de procesare și a unor parametrii care dictează importanța anumitor caracteristici ale datelor în calculul rezultatului final, un model bazat pe deep learning este capabil să facă predicții cu o rată foarte mică de eroare.

Spre deosebire de o rețea neurală simplă care conține câteva straturi, o rețea neurală adâncă are nevoie de câteva zeci sau poate sute pentru a putea avea succes. Acesta este unul dintre motivele pentru care metoda propusă a devenit populară abia recent; o rețea atât de complexă are nevoie de un calculator puternic pentru a funcționa, mai ales de o placă video performantă care este folositoare prin capacitatea ei de a face calcule în paralel.

La nivel general, deep learning este folosit în detecția de imagini, unde este capabil să clasifice anumite obiecte și să detecteze anumite caracteristici ale imaginilor. În plus, este folosit și la procesarea naturală a limbajului (NLP), în traducerea limbii sau generarea de text, care contribuie la crearea de asistenți virtuali sau roboți care pot genera răspunsuri relativ umane. În practică, detecția imaginilor este folosită în conducerea automatizată, unde sunt detectate semne de circulație, semafoare, pietoni sau alte mașini participante la trafic. În medicină, deep learning este capabil să genereze diagnostice prin interpretarea analizelor medicale, atât din punct de vedere al diferitelor valori, cât și prin prisma înțelegerii radiografiilor sau a IRM-urilor (Imagistică prin rezonanța magnetică) [6]. În plus, pe lângă interpretarea unor poze, algoritmul poate fi antrenat să genereze imagini cu ajutorul GAN-urilor (generative adversarial networks) sau a VAE-urilor (variational autoencoders) pe baza unor descrieri prin text, ceea ce îl

face popular în diferite domenii precum cel artistic sau în entertainment.

1.3. Structura lucrării

Lucrarea urmează să fie structurată în două părți. Prima parte va ocupa o mare parte din lucrare și va descrie procesul prin care a fost creat un set de date, urmând să fie prelucrat cu ajutorul unui model. Vor fi explicate detalii despre arhitectura folosită, alegerea parametrilor și rezultatele obținute în urma antrenării modelului. Rezultatele proprii vor fi comparate cu cele din alte lucrări specifice temei. A doua parte va menționa alte metode pentru detecția glitchurilor și va conține o analiză asupra lor, comparând metodele găsite cu cea descrisă în principal în lucrare. La final se va regăsi un capitol despre eventualele dificultăți și aspecte care ar putea fi îmbunătățite.

2. Preliminarii

2.1. Terminologie

În secțiunea curentă vor fi descrise noțiuni generale și termeni specifici folosiți în domeniu și regăsiți în lucrare. Termenii punctuali, de exemplu un anumit tip de loss ales sau o arhitectură anume, vor fi explicați în momentul în care vor fi menționați.

Un joc video reprezintă orice tip de aplicație digitală la care pot participa unul sau mai mulți jucători și care este destinată divertismentului, având un set clar de reguli. Jocul poate conține mai multe erori, care sunt descrise de doi termeni specifici: glitch sau bug. Cuvintele menționate sunt folosite deseori în contexte similare, dar există anumite diferențe date de problemele cărora li se adresează. Un bug este o eroare critică ce constă în greșeli de programare și care poate determina un joc să se blocheze sau să se închidă brusc. Un glitch reprezintă o eroare de moment care poate fi rezolvată de la sine și care nu reprezintă o problema gravă, deci care poate fi ignorată adeseori. Câteva exemple de glitchuri ar fi blocarea temporară a jocului, aspectul diferit și anormal al obiectelor din joc sau poziția lor, dar și probleme la nivel audio, precum dublarea coloanei sonore.

2.1.1. Termeni în dataset

Termenii descriși aici sunt folosiți în special pentru descrierea, explicarea creării setului de date sau punctarea modului în care este prelucrat în scopul antrenării modelului.

Un set de date (data set) se referă la o colecție de date care pot fi de mai multe tipuri: text, imagine, audio, video sau numerice. În contextul domeniului de machine learning, setul de date este împărțit în datele pentru antrenare, validare și testare. Cele trei contribuie la prepararea modelului pentru a face predicții, dar și la ajustarea parametrilor folosiți, urmând ulterior ca algoritmul să fie testat. Corectitudinea setului de date este esențială deoarece el determină ceea ce modelul învață. Dacă unele dintre date sunt incomplete, nu fac parte din categorii comune sau nu respectă diferite tehnici specifice, modelul va performa mai slab atunci când va fi testat cu date necunoscute.

Setul de date folosit pentru antrenarea modelului este unul sintetic, adică datele

conținute sunt generate și nu preluate din mediul natural. Crearea lui a avut loc cu ajutorul unei platforme sau game engine, software folosit pentru implementarea jocurilor video, pe nume Unity. Cu ajutorul unor pachete specifice, Unity poate fi folosit și pentru crearea unui set de date.

Pachetul conține un set de instrumente sau features prin intermediul cărora utilizatorul poate rezolva o anumită problemă. Pentru utilizatorii care nu doresc crearea unor obiecte, texturi sau materiale, există Unity asset store, un magazin virtual unde diferite pachete predefinite de către alți creatori pot fi achiziționate în scopul folosirii lor în diverse proiecte.

Setul de date este împărțit în mai multe categorii: învățare, validare și testare. Setul folosit pentru învățare (training set) are rolul de a ajuta modelul să se antreneze. Algoritmul învață să rezolve problema propusă prin analiza datelor și identifică o multitudine de diferențe sau asocieri în poze prin care poate clasifica datele din setul de testare. Acest set este folosit explicit pentru a testa modelul, conținând date la care programul nu a avut acces în perioada de învățare. Pe baza lui, algoritmul face predicții, rezultatele fiind analizate de către programator. Înainte de testare, este folosit setul de validare, care ajută programatorul să ajusteze anumiți parametrii pentru a îmbunătăți modelul. De obicei, 60% dintre informațiile din setul de date sunt folosite pentru învățare, 20% pentru validare și restul de 20% pentru testare.

Cu cât setul de date este mai mare, dar în același timp este și corect alcătuit, fără date inutile, cu atât modelul poate performa mai bine. În practică, alcătuirea unui set de date poate fi un proces complicat deoarece este necesar un volum mare de informații care să se potrivească din perspectiva anumitor caracteristici. Dacă datele nu sunt uniforme sau obiective, uneori algoritmul poate avea anumite ”prejudecăți”. În realitate, programul capătă o oarecare subiectivitate din cauza datelor. De exemplu, dacă modelul trebuie să genereze o imagine color dintr-o poză alb-negru în care se află o persoană care poartă o rochie, el va tinde să coloreze acea rochie cu roșu daca majoritatea pozelor care conțineau rochii, folosite în învățarea sa, erau de culoare roșie.

O modalitate prin care problema absenței unui set de date complex poate fi depășită este prin generarea datelor cu ajutorului unui calculator. Acest tip de date este sintetic și de obicei încearcă să imite caracteristici reale. O altă posibilitate care constă în prelucrarea setului astfel încât să crească în volum este prin augmentarea datelor. Tehnica aplică diferite transformări ale imaginilor astfel încât imaginile noi să poată fi folosite la rândul lor. Cele mai populare transformări pentru date de tip imagine sunt

cele geometrice, de culoare, cele prin care se aplica filtre Kernel și ștergerea aleatorie a unor părți din imagini [7]. Transformările geometrice deseori folosite sunt întoarcerea, mărirea sau rotația. Schimbările de culoare manipulează valorile canalelor RGB pentru a putea ajusta luminozitatea pozei, iar filtrele Kernel au rolul de a crește contrastul sau de a mări neclaritatea unei imagini.

2.1.2. Termeni în machine learning

Cel mai des menționat cuvânt este termenul de model. Un model este un tip de program care are capacitatea de a putea recunoaște caracteristici complexe cu ajutorul cărora pot fi făcute predicții asupra unor date neobservate [8]. De exemplu, programul propus în temă va primi mai multe imagini normale sau cu diferite tipuri de glitchuri și va prezice dacă o imagine nouă este obișnuită sau prezintă un glitch.

Arhitectura stă la baza modelului și constă în rețele neurale care sunt alcătuite din mai multe straturi. Straturile în sine sunt fondate pe anumiți algoritmi, fiecare strat ocupând-se cu extragerea anumitor caracteristici din poze. În detecția feței umane, un strat ar putea găsi colțurile din imagine, în timp ce altul ar căuta trăsături specifice definitorii omului precum ochii, nasul sau urechile.

Dacă am încadra caracteristicile de început ale datelor din set într-un grafic, scopul modelului ar fi să traseze o linie care să separe cât mai bine punctele proiectate. Doi termeni des folosiți în acest caz sunt overfit și underfit, care ar putea fi traduși ca supraadaptare și subadaptare. Supraadaptarea are loc atunci când modelul nu generalizează bine in perioada de testare [9]. Am putea spune că modelul doar memorează informația, în loc să o înțeleagă. Subadaptarea se întâmplă când programul nu se descurcă în perioada de antrenare, deci nici în cea de evaluare. Fără a fi capabil să extragă anumite caracteristici din setul de învățare, nu poate să prezică corect nici clasele datelor noi.

Modul în care modelul poate învăța se împarte în trei tipuri: învățare supervizată, fără supervizare sau supervizată parțial. Primul tip semnifică etichetarea fiecărui exemplu din setul de date, încadrându-l într-o categorie. Învățarea fără supervizare constă într-un set de date fără etichete, iar ultimul tip reprezintă o combinație dintre primele două.

Cei mai importanți factori folosiți în antrenarea unui model sunt optimizatorul, funcția de pierdere (cost), rata de învățare și numărul de epoci. Optimizatorul este un tip

de algoritm care ajustează gradual parametrii modelului pentru a minimiza funcția de cost [10]. Greutățile reprezintă coeficienți asociați neuronilor care conțin caracteristici ale imaginilor și marchează printr-o valoare numerică importanța fiecărui neuron, iar biasul este un parametru suplimentar care influențează rezultatul algoritmului. Funcția de pierdere sau de cost cuantifică numeric cât de aproape este predicția de adevăr. Cu cât funcția de cost este mai mare, cu atât programul se descurcă mai rău. Rata de învățare stabilește rapiditatea cu care modelul ar trebui să se adapteze pentru datele în schimbare. Dacă rata este mare, sistemul se adaptează ușor la date noi, dar le va uita mai rapid pe cele vechi. În cazul unei valori mici, procesul de învățare va fi mai lent, dar va putea să ignore mai ușor datele nereprezentative [10]. O epocă reprezintă o iterație în perioada de învățare în care toate exemplele din setul de date sunt folosite. Pe măsură ce trec mai multe epoci, modelul învață și își updatează parametrii.

2.2. Stadiul actual al temei

În ciuda faptului că deep learning are aplicații concrete și complexe pentru o varietate mare de teme de actualitate, menționate în secțiunea introductivă, există un număr mic de lucrări publicate care abordează tema propusă în lucrare. În plus, nu am reușit să găsesc un set de date care să poată fi antrenat, dar nici un exemplu de cod din domeniu care să rezolve problema discutată.

2.3. Obiectivele lucrării

Îmi propun să contribui în metodele pentru detecția glitchurilor în jocurile video prin crearea unui set de date simplu, care să reprezinte un punct de plecare pentru cei care doresc să studieze la rândul lor problema. Plănuiesc să public setul pe o platformă destinată prelucrării de date cu ajutorul inteligenței artificiale, cum ar fi Kaggle. Voi descrie atât conținutul setului de date, cât și pașii urmați pentru crearea acestuia, urmând să adaug și metode prin care pozele conținute ar putea fi generate într-un mod mai complex sau mai complet, astfel încât modelul să poată fi antrenat și îmbunătățit pentru problemele practice.

Codul personal scris va fi publicat la rândul sau, cu mențiunea că reprezintă un pas de început care să servească drept șablon. În lucrare voi descrie dificultățile întâmpinate, dar și aspecte care ar putea fi îmbunătățite pentru optimizarea codului.

Analiza altor lucrări sau metode reprezintă un alt aspect important pentru obiectivul lucrării. Îmi propun că aici să rezum și să explic abordările altor cercetări, cu o interpretare proprie asupra rezultatelor și a avantajelor și dezavantajelor aplicațiilor.

3. Identificarea glitchurilor

3.1. Setul de date

Varianta inițială pentru alegerea unui set de date a fost căutarea unui set complet și public. În absența sa, a fost necesară crearea unui set care să poată fi folosit de către o rețea neurală adâncă. În subcapitolele următoare, va fi descris procesul de alegere al datelor, dar și modul prin care acestea au fost generate.

3.1.1. Stabilirea tipului de date

Pentru crearea unui set de date, o modalitate ușoară de a popula setul este prin generarea de imagini. În acest fel poate fi tratată categoria principală de glitchuri, cele la nivel de imagine, dar sunt excluse cele de alte tipuri, cum ar fi cele legate de coloana sonoră. Modul abordării setului de date este inspirat din lucrarea lui Carlos Garcia Ling [2].

Cum tema abordează erori la nivel de joc video, iar un joc video este un mediu generat de un game engine, ideea folosirii unui set de date sintetic are sens. În plus, un game engine poate permite generarea unui număr mare de imagini, deci un număr mic de date într-un set nu reprezintă o problemă. Învățarea prin supervizare este accesibilă deoarece datele pot fi împărțite ușor în mai multe foldere ale căror nume poate stabili și eticheta subsetului. În general, abordarea supervizată obține rezultate mai bune decât cea în care nu sunt folosite etichete [2].

Unul dintre tipurile de glitchuri vizuale este cel legat de texturile corupte. Texturile sunt imagini care acoperă un obiect pentru a-i conferi un aspect diferit. De exemplu, în Unity poate fi creat un poligon simplu, precum un paralelipiped. Pentru a-i conferi aspectul unui zid, poate fi folosită o imagine cu un perete de cărămidă care să fie aplicată ca textură asupra obiectului.

Pentru generarea unor obiecte cu glitchuri nu am găsit o aplicație care să le poată genera în mod direct. Varianta abordată a fost schimbarea aspectului sau desfășurării texturilor în așa fel încât să reproducă cât mai aproape de adevăr un glitch care are se întâmplă în mod natural în mediul jocului. Pe lângă imaginea fără probleme (Fig. 3.1),

am ales 4 tipuri de glitchuri pe care să le încadrez în setul de date: textura întinsă (Fig. 3.2), de calitate scăzută (Fig. 3.3), textura înlocuitoare (Fig 3.4) și textura lipsă (Fig. 3.5). Exemplele de mai jos sub forma de poze sunt selectate din setul de date creat. Etichetele poartă numele de: „Stretched”, „Low”, „Placeholder” și „Missing”, unde este adăugată și eticheta „Normal” care este aplicată imaginilor fără defecte. Prin textura întinsă am definit un obiect a cărui textură nu este aplicată o singură dată, ci pe suprafața lui textura poate fi deplasată sau repetată. Glitchul de tip rezoluție scăzută reprezintă opusul celui menționat anterior, adică nu este aplicată întreaga textură pe obiect, doar o parte din aceasta. Când compoziția obiectului este înlocuită cu o textură de rezervă, mai exact una albă și transparentă, glitchul este de tip placeholder. Daca fișierul ce redă aspectul fizic al obiectului lipsește în totalitate, culoarea acestuia devine un roz puternic contrastant, care marchează faptul că textura să originală nu poate fi randată, adică aplicată asupra obiectului și afișată vizual.

Figura 3.1

Magazin - Normal

Obiect normal, fără glitchuri

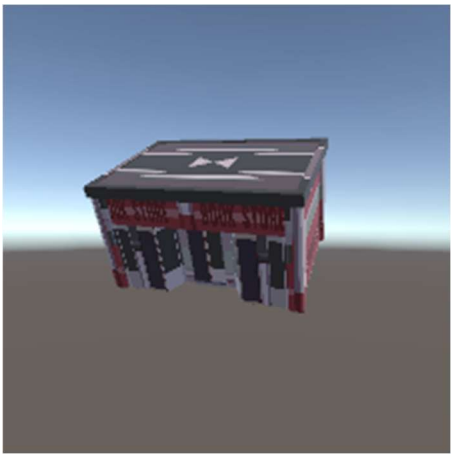

Figura 3.2

Magazin - Stretched

Obiect cu textura intinsă

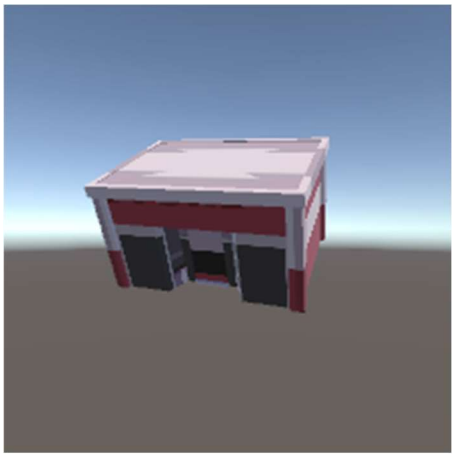

Figura 3.3

Magazin - Low resolution

Obiect cu rezoluție scazută

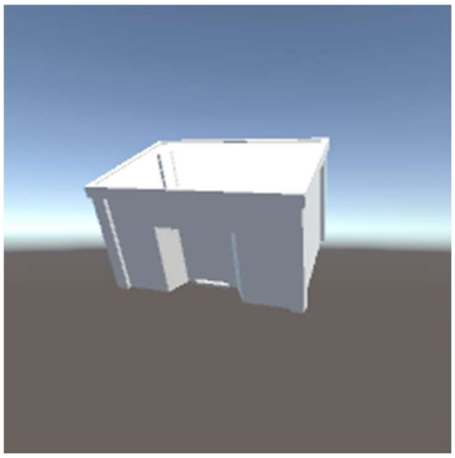

Figura 3.4

Magazin - Placeholder

Obiect cu textura inlocuită

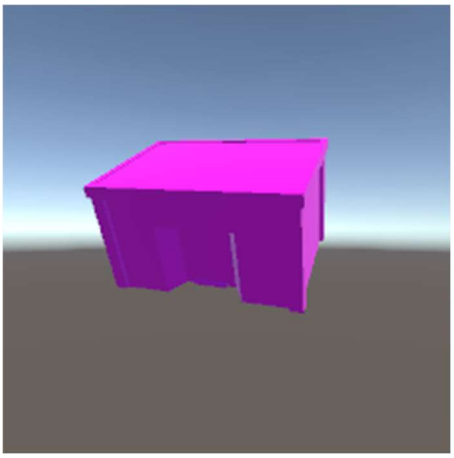

Figura 3.5

Magazin - Missing

Obiect cu textura lipsă

3.1.2. Crearea setului de date

Setul de date a fost creat în Unity, cu ajutorul pachetului din Unity pentru percepție (Unity perception package). Pachetul conține un set de instrumente, precum identificarea obiectelor și segmentarea semantică, destinat generării unui set de date sintetic. Au fost alese 10 prefabricate, mai exact obiecte, din pachete descărcate din Unity asset store. Patru dintre cele zece componente de joc provin din „Simple Town Lite – Cartoon Assets", Synty Studios, două sunt preluate din „Apartment kit", Brick Project Studio, iar restul fac parte din „RPG/FPS Assets for PC/Mobile", Dmitrii Kutsenko. În total au fost generate 2500 de poze, fiecare obiect având câte 50 de imagini pentru fiecare dintre cele 5 clase.

Camera principală conține un script numit „Perception camera", unde poate fi setat modul în care sunt captate pozele. Prin bifarea opțiunii de salvare a rezultatului RGB pe un disc local, imaginile vor fi descărcate într-un folder. Modul de capturare este setat pe varianta automată, parametrul „simulation delta time" stabilind numărul de cadre pe secundă. Valoarea folosită este cea prestabilită, la fiecare 0.0166 secunde, o imagine este salvată, ceea ce este echivalent cu 60 FPS. În fereastra care descrie jocul vizual, poate fi aleasa o rezoluție a pozelor. Dimensiunea setată este de 224x224 pixeli, astfel încât capturile să poată fi ușor procesate ulterior de model, dar în același timp fără a pierde detalii importante. În setările camerei, mărimea câmpului de vizualizare al imaginii a fost micșorată pentru ca obiectul să poată fi privit mai îndeaproape în imaginea rezultată. În plus, camera este centrata pe obiect, în afara de axa verticală, obiectul situându-se în partea de sus a cadrului pentru a garanta

captura lui în întregime atunci când va urma să fie rotit aleatoriu. În scena principală este creat un obiect de joc nou, redenumit ca „Scenariu”, căruia îi adăugăm o componentă numită scenariu de lungime fixă. Aici pot fi setate seedul pentru factorul aleatoriu, care se asigură că pentru fiecare obiect vom avea aceeași desfășurare a acțiunilor aleatoare, startul iterației, mai exact valoarea de la care să înceapă numărătoarea, dar și numărul de iterații, care este setat la 50. În secțiunea de randomizatori, este adăugat un randomizator de rotire, care schimbă poziția obiectului pe axe pentru a putea fi privit din unghiuri diferite (Fig 3.6, Fig 3.7).

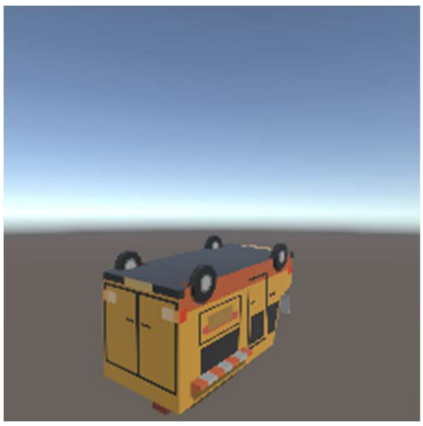

Figura 3.6

Mașină rasturantă

Un tip de rotire aleatoare

Figura 3.7

Masină rotită în lateral

Un tip de rotire aleatoare

Limitele rotirii au fost păstrate la setările prestabilite, intervalul de rotire fiind între 0° și 360°. Pentru ca obiectul să preia setările scenariului, i se aplică un tag numit „Rotation randomizer tag”. Prin apăsarea butonului de play, programul se desfășoară corect și salvează pozele în folder, alături de un fișier json care conține configurațiile scriptului. Există în final un singur folder care conține 5 dosare, fiecare reprezentând o etichetă, unde sunt prezente câte 500 de imagini. Pentru că există conflicte de nume, fiecare imagine fiind generată cu un nume cuprins între „rgb_1” si „rgb_50”, am schimbat numele lor cu ajutorul unui instrument online pentru ca fiecare folder specific să aibă datele denumite precum „001.jpeg”, până la „500.jpeg”. Va exista un alt mod descris în care fișierele vor fi împărțite pentru a facilita citirea lor de către program.

Scopul rotirilor este de a diversifica setul de date și de surprinde obiectul din mai multe perspective pentru analiza glitchurilor din unghiuri diferite în cazul în care el este

parțial vizibil. Un obiect poate fi folosit în poziții diferite în scenarii diverse în joc. De exemplu, o mașină răsturnată (Fig. 3.6) s-ar potrivi unui eveniment de joc în care a avut loc un accident. În plus, daca podeaua autovehiculului ar conține un glitch, el nu ar putea fi identificat dacă imaginea nu captează mașina de dedesubt (Fig 3.7). Există și alte efecte posibile ce pot fi aplicate pentru a crește complexitatea setului de date, precum randomizarea poziției, aplicarea unui fundal aleatoriu sau adăugarea altor obiecte în scenă. Pentru simplitatea setului de date și pentru a creste șansa de învățare corectă a modelului, a fost aplicată doar rotirea aleatoare. Faptul că nu sunt cuprinse scenarii cu efecte menționate anterior nu reprezintă un impediment deoarece orice obiect poate fi ușor extras și centrat fară să aibă un mediu în care să fie încadrat. Astfel, modelul prezintă aplicabilitate practică.

3.2. Modelul

3.2.1. Prelucrarea setului de date

Setul de date va fi împărțit cu ajutorul funcției datasets.ImageFolder, care provine din librăria torchvision și care necesită o aranjare specifică a fișierelor, cu calea de acces sub forma de 'root/folder_nume_de_eticheta/imagine' [11].

```
source_folder = 'Complete samples/Missing'
destination_folder = 'test/Missing'

image_files = [image for image in os.listdir(source_folder)]
selected_images = random.sample(image_files, 50)

for image in selected_images:
    source_path = os.path.join(source_folder, image)
    destination_path = os.path.join(destination_folder, image)
    shutil.move(source_path, destination_path)
```

Codul menționat înlocuiește procesul prin care pozele să fie recategorisite manual, unde șansa de eroare este mai crescută și alegerea pozelor nu este neapărat aleatoare. Variabilele „source_folder” și „destination_folder” rețin căile absolute către dosarul din care urmează să fie mutate date, respectiv cel în care urmează să fie transportate. Este ales un număr de imagini aleatoare cu ajutorul funcției random.sample(). Al doilea parametru este ales în așa fel încât imaginile să fie împărțite în 80% date de învățare, 10% validare și 10% testare. Astfel, 2000 de imagini sunt folosite la partea de training,

apoi câte 250 pentru fiecare dintre celelalte categorii.

Cu ajutorul librăriei torchvision, sunt folosite funcții specifice de tip transforms. Rolul lor este de a augmenta datele din set, mai exact de a mari artificial și de a diversifica setul de date de învățare prin aplicarea unor transformări. Sunt aplicate funcții precum RandomHorizontalFlip(), RandomVerticalFlip(), care învârt imaginea pe orizontală, respectiv verticală, cu o probabilitate prestabilită de 50%, dar și RandomRotation(45), care rotește o imagine la 45°. Cu ajutorul librăriei matplotplib.pyplot pot fi vizualizate mai multe exemple de imagini din setul de date.

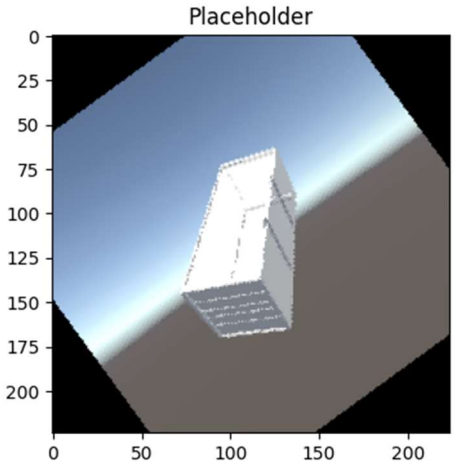

Figura 3.8

Poză rotită

O poză cu un container cu glitch de tip placeholder, rotită la 45°

Datele din seturile de validare și testare nu sunt augmentate deoarece ar fi populate cu date care nu respectă realitatea setului și modelul ar putea fi subiectiv în prezicerea imaginilor cărora li s-au aplicat transformări. Toate imaginile au fost normalizate, o tehnică de preprocesare pentru a aduce valorile tuturor pixelilor la o scală comună. Valoarea mean a fost setată la 0, iar deviația standard la 1. Funcția ImageFolder() este un data loader generic care necesită o structură specifică a fișierelor [11]. Prin intermediul ei sunt create seturile de învățare, validare și testare, cu etichetele specifice. Din modulul torch, este folosită funcția utils.data.DataLoader(), care stabilește numărul de imagini care urmează să treacă prin arhitectură la o iterație. Astfel, este folosita mai puțină memorie și modelul este antrenat mai rapid deoarece poate fi încărcat un număr mult mai mic de date decât mărimea setului. În plus, valoarea parametrului shuffle este setată cu „True” pentru a amesteca aleatoriu datele în așa fel încât modelul să rețină caracteristicile generale, nu ordinea în care pozele sunt parcurse. Batch_size este un parametru pentru care vor fi încercate mai multe valori în scopul de a găsi o valoare optimă pentru antrenarea modelului.

3.2.2. Caracteristicile arhitecturii

Rețeaua principală folosită pentru testarea mai multor parametrii este o rețea de tip rezidual numită ResNet18, a cărei performanță a fost recunoscută la competițiile ILSVRC și COCO 2015. Principala caracteristică o reprezintă proprietatea de a sări peste anumite conexiuni din straturi (skip connections, fig. 3.9), rezolvând problema diminuării gradientului [12], care este responsabil de ajustarea greutăților neuronilor. Problema este des întâlnită în rețele cu un număr mare de straturi, unde propagarea gradientului este diminuată pe parcurs.

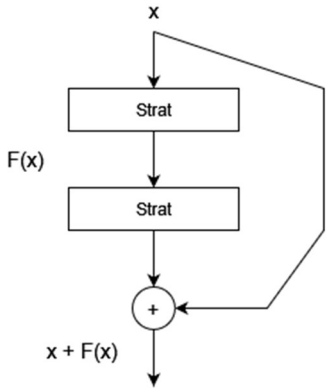

Figura 3.9

Invățare reziduală

Un fragment din arhitectură

Sunt rezervate între 50 și 100 de epoci pentru antrenarea modelului deoarece sunt obținute rezultate bune, fără a fi necesară mărirea numărului de cicluri de antrenare. Modelul cu configurațiile optime poate fi antrenat pe parcursul a mai multor epoci, în încercarea de a obține rezultate mai bune. În majoritatea antrenărilor sunt folosite greutățile prestabilite ale arhitecturii deoarece a fost observată o performanță mai bună ce poate fi justificată prin faptul că imaginile sintetice cu obiecte din mediul jocului tind să imite realitatea [2]. O parte dintre pozele selectate pentru setul de date conțin obiecte cu o înfățișare de tip desen animat pentru a diversifica setul de date, dar fără să afecteze negativ performanța modelului atunci când folosește o rețea preantrenată. Au fost adăugate două straturi la finalul rețelei, un strat de dropout și unul liniar. Dropout-ul reprezintă o tehnică de regularizare care previne supraadaptarea prin renunțarea temporară la anumiți neuroni și a legăturilor lor pe parcursul antrenării. Astfel, este eliminată interdependența lor din luarea predicțiilor corecte [13]. Stratul liniar este ultimul strat și transformă ieșirea rețelei într-un rezultat de 5 noduri, care vor reprezenta probabilitatea ca imaginea respectivă să se încadreze într-una dintre cele 5 clase. Optimizatorul

principal este Adam, o variantă îmbunătățită a SGD, care este propus pentru cel mai bun optimizator stocastic și care are ca proprietate principală adaptarea ratei de învățare pe baza primelor două momente ale gradientului [14]. Funcția de cost folosită este CrossEntropyLoss, care este cunoscută pentru eficiența ei în problemele de clasificare cu mai multe clase. Funcția măsoară calitatea predicției și penalizează modelul atunci când estimează o probabilitate scăzută pentru clasa corectă [10].

3.2.3. Funcţia pentru învăţare şi validare

Epocile sunt parcurse de la epoca salvată (saved_epoch + 1), până la finalul stabilit (saved_epoch + epochs + 1, unde variabila epochs se referă la numărul de epoci de parcurs). Pentru fiecare iterație sunt reținute costul total și acuratețea atât pentru antrenare, cât și pentru validare, în total fiind 4 valori așteptate ca rezultate. În plus, sunt folosite 4 liste diferite pentru a reține aceste valori care vor contribui la încadrarea numerelor într-un grafic.

În perioada de învățare, este folosită funcția .train(), care îi comunică modelului că trebuie să învețe și astfel să nu folosească anumite tehnici prezente care nu sunt specifice acestui segment, cum ar fi stratul de dropout. În codul menționat, imaginile și etichetele lor sunt preluate din train_loader. Funcția tqdm() este folosită pentru a afișa o bară de

```
for images, labels in tqdm(train_loader):
        images = images.to(device)
        labels = labels.to(device)
        outputs = net(images)
        correct_labels = 0
        total_labels = 0

        #Get correct predicted labels for train
        for j, output in enumerate(outputs):
            sm_output = F.softmax(output, dim=0)
            label = torch.argmax(sm_output)
            if labels[j] == label:
                correct_labels += 1
            total_labels += 1

        loss = loss_fn(outputs, labels)
        epoch_train_loss += loss.item()
        train_acc = correct_labels / total_labels
        epoch_train_acc += train_acc
        optimizer.zero_grad()
```

```
        loss.backward()
        optimizer.step()
```

progres pentru timpul rămas până la găsirea rezultatului într-o epocă [15]. Funcția .to(device) specifică faptul că tensorii vor fi prelucrați de un anumit dispozitiv, în acest caz fiind folosită o placă video. Variabila outputs conține predicția modelului pentru fiecare imagine din lot. Următorul bloc repetitiv calculează numărul de predicții corecte. Funcția softmax() calculează un scor pentru fiecare clasă, ca apoi să estimeze probabilitatea pentru fiecare etichetă [10], iar funcția argmax() selectează eticheta cu probabilitatea cea mai mare. Dacă modelul a prezis corect eticheta, variabila correct_labels este incrementată, iar total_labels va reține numărul total de etichete primit. Funcția de cost calculează costul pe baza diferenței dintre predicții și valorile originale, iar costul pentru lotul curent este adăugat în epoch_train_loss. Acuratețea este calculată în train_acc și constă în raportul dintre numărul de clase prezise corect și numărul total de clase. Rezultatul este adăugat în epoch_train_acc, procesul fiind similar cu însumarea costurilor. Cu ajutorul funcției zero_grad(), sunt resetați gradienții pentru a nu fi acumulați pe parcursul a mai multor loturi. Backward() produce efectul de retropropagare care ajustează parametrii modelului. Funcția step() actualizează parametrii cu valorile noi prin intermediul optimizatorului ales. Blocul de validare este similar cu cel de învățare, însă o diferență semnificativă constă în folosirea funcției torch.no_grad(). Prin intermediul ei este dezactivat calculul gradienților deoarece în segmentul curent de cod are loc o verificare a performanței modelului. În plus, este folosită mai puțină memorie, deci și timpul de calcul este mai rapid. Mai jos este menționat un exemplu de afișare al funcției tqdm(). Poate fi observat faptul că perioada de învățare a durat 20 de secunde, cu 24 de iterații pe secundă, în timp ce procesul de validare a durat doar o secundă și a fost parcurs cu aproape 44 de iterații pe secundă:

Învățare: 100%|██████████| 500/500 [00:20<00:00, 24.21it/s]

Validare: 100%|██████████| 63/63 [00:01<00:00, 43.97it/s].

La finalul buclelor repetitive, sunt calculate costul și acuratețea prin împărțirea totalului de valori obținute la numărul de date procesate din setul de învățare sau validare. Codul de mai jos are rolul de a salva anumite epoci și rezultate în memoria locală. La început se stabilește dacă epoca curentă are cea mai bună acuratețe a validării dintre toate epocile parcurse până la momentul respectiv. În caz afirmativ, este salvată cu ajutorul funcției torch.save(), care descarcă un fișier cu mai multe date pe un disc local [16]. Sunt reținute

valori precum numărul epocii, rețeaua folosită, optimizatorul, costul și acuratețea învățării și validării. Salvarea lor într-un fișier cu nume specific, care conține anumite variabile ce sunt deseori modificate pentru a găsi și compara mai multe rezultate, favorizează localizarea ușoară a fișierelor căutate pentru a putea fi folosite în secțiunea de analiză a ieșirilor modelului. De asemenea, este salvată și ultima epoca parcursă deoarece este posibilă antrenarea modelului de la o anumită configurație salvată, fără a fi necesar ca acesta să fie antrenat din nou de la început. Aici nu sunt salvate valori precum costul sau acuratețea deoarece nu sunt necesare pentru a rezuma procesul de rulare.

```
#Saved best epoch based on validation accuracy
    if avg_val_acc >= max_validation_acc:
        max_validation_acc = avg_val_acc

        torch.save({
            'epoch': epoch,
            'model_state_dict': net.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'train_loss': avg_train_loss,
            'validation_loss': avg_val_loss,
            'train_accuracy': avg_train_acc,
            'validation_accuracy:': avg_val_acc,
        },
f"model/best_epoch_{net_name}_{pretrained_name}_{optimizer_name}_{lr_name}_{lo
ss_fn_name}.pth")

    #Save last epoch
    if epoch == saved_epoch + epochs:
        torch.save({
            'epoch': epoch,
            'model_state_dict': net.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': avg_train_loss,
        },
f'./model/model_{epoch+1}_{net_name}_{pretrained_name}_{optimizer_name}_{lr_na
me}_{loss_fn_name}.pth')
```

Codul de mai jos încarcă din memorie datele salvate ale unei anumite epoci pentru a continua antrenarea din acel stadiu. De exemplu, calea fișierului este către o rețea ResNet18 aflată la epoca 20, care nu folosește costuri prestabilite și care optimizează cu ajutorul algoritmului Adam, cu o rată de învățare de 0.0001. Funcția de cost este CrossEntropyLoss.

```
path = "model\model_20_ResNet18_NotPretrained_Adam_10^-4_CrossEntropy.pth"
checkpoint = torch.load(path)
net.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
saved_epoch = checkpoint['epoch']
loss = checkpoint['loss']
```

Istoricul acurateței și al costului pe parcursul epocilor sunt afișate la finalul execuției programului și salvate în memoria locala cu ajutorul librăriei matplotlib.pyplot. Funcția figure() stabilește mărimea graficului, iar funcțiile title(), xlabel() și ylabel() numesc titlul graficului și denumirile celor două axe. Savefig() salvează rezultatul în memorie, iar show() îl afișează în cadrul proiectului. Funcția plot() determină ce date sunt afișate, mai jos fiind descris procesul prin care costul antrenării și al validării este afișat și salvat. Exemple de grafice vor fi prezentate în secțiunea de rezultate.

```
plt.figure(figsize=(10, 5))
  plt.title("Training and Validation Loss")
  plt.plot(validation_losses, label="validation")
  plt.plot(train_losses, label="train")
  plt.xlabel("epochs")
  plt.ylabel("Loss")
  plt.legend()
  plt.savefig(f"./loss_and_acc/loss_plot_{epoch+1}_{net_name}_{pretrained_name
}_{optimizer_name}_{lr_name}_{loss_fn_name}.png")
  plt.show()
```

3.2.4. Funcţia de testare

Funcția de testare este similară cu cea de validare, compară predicția modelului cu valoarea reală a etichetelor și calculează acuratețea prin raportul dintre predicțiile corecte și numărul total de date. În plus, sunt calculate precizia și recall-ul care stabilesc dacă clasele sunt balansate corect. De exemplu, modelul ar putea prezice mai ușor dacă un obiect are un glitch cu eticheta „Missing” decât unul cu eticheta „Stretched” deoarece primul constă în schimbarea texturii într-o culoare roz, ceea ce poate fi ușor de identificat.

Precizia este calculată ca raportul dintre pozitivele prezise corect și suma dintre pozitivele prezise corect și cele prezise incorect. Recall-ul este calculat ca raportul dintre pozitivele corecte și suma dintre pozitivele corecte și negativele incorecte. Un pozitiv corect reprezintă cazul în care modelul prezice corect clasa căutată. Un negativ constă în predicția incorectă a exemplului negativ, negativ însemnând în modelul propus o clasă care nu este

$$Precision = \frac{TP}{TP+FP}$$

Ecuația 3.1.

Precizie

Calculul preciziei

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

Ecuația 3.2.

Recall

Calculul recallului

căutată în momentul respectiv. Pentru fiecare dintre cele 5 clase există variabilele din formulă, de exemplu pentru pozitivele corecte sunt Fp_low, Fp_missing etc. În cazul în care eticheta prezisă este și cea adevărată, incrementam cu 1 valoarea variabilei TP. În caz contrar, creștem valoarea pentru FP și în funcție de eticheta prezisă, este mărită și valoarea FN. De exemplu, dacă modelul a prezis eticheta „Low”, iar valoarea adevărată este „Missing”, atunci va fi incrementată Fn_low.

3.2.5. Matricea de confuzie

Matricea de confuzie este un tabel care rezumă prin câteva valori performanța unui model. Numărul de rânduri și coloane este egal cu numărul total de etichete și, de obicei, pe linie se regăsesc etichetele reale, iar pe coloana etichetele prezise [10]. Astfel, o celulă cuprinde o valoare ce punctează numărul de date cu o anumită etichetă care au fost prezise cu o alta etichetă sau cu aceeași de pe linie. De exemplu, pe diagonala principală a matricei se află numărul de imagini prezise corect.

```
def get_cm(loader, set_type="train"):
    predicted_labels = []
    correct_labels = []

    for images, labels in loader:
        images = images.to(device)
        labels = labels.to(device)

        outputs = net(images)

        for j, output in enumerate(outputs):
            sm_output = F.softmax(output, dim=0)
            label = torch.argmax(sm_output)

            correct_label = labels[j].cpu().detach().numpy()
            predicted_label = label.cpu().detach().numpy()
            predicted_labels.append(predicted_label)
```

```
            correct_labels.append(correct_label)

    cm = confusion_matrix(predicted_labels, correct_labels)
    plot_cm(cm, set_type, "number")

    cmn = cm.astype(float) / cm.sum(axis=1)[:, np.newaxis]
    cmn = 100 * cmn
    plot_cm(cmn, set_type, 'percentage')
```

Funcția de mai sus descrie modul în care sunt preluate datele pentru a putea fi creată matricea de confuzie. După un proces asemănător funcției de antrenare și validare, etichetele corecte și prezise sunt convertite în liste de tip Numpy pentru a putea fi adăugate în listele predicted_labels și correct_labels, folosite în alcătuirea matricei. Pentru matricea de confuzie normalizată, care afișează valorile ca și procente, sunt convertite rezultatele matricei originale în numere de tip float și împărțite la suma de pe rând, apoi este adăugată o axă nouă rezultatului pentru a putea fi compatibil la împărțire. Valorile sunt înmulțite cu 100 pentru a fi convertite la procente și sunt afișate cu ajutorul funcției plot_cm. Funcția inițializează o figură prin intermediul plt.figure(), setându-i dimensiunea la 10x10inchi. Parametrii set_type și numeric_type stabilesc daca matricea este folosită pentru afișarea rezultatului din antrenare, validare sau testare, respectiv specifică dacă valorile sunt numerice sau sub formă de procente. O axă noua este creată, care determină locul în care matricea va fi desenată. Cu ajutorul librăriei seaborn este folosită o hartă termografică prin intermediul căreia valorile vor fi mai ușor de interpretat. Parametrul annot=True permite afișarea valorilor în fiecare celulă, iar fmt='g' specifică tipul parametrilor, mai exact tipul string. Un text este încadrat de-a lungul axelor Ox și Oy prin intermediul funcțiilor set_xlabel(), respectiv set_ylabel(). Funcția set_ticklabels(classes) precizează că valorile trebuie preluate din variabila precizată ca parametru. Textul de lângă axe este întors cu xticks() și yticks() pentru a fi orientat într-un fel în care să poată fi ușor de citit. Într-un final, matricea este salvată și afișată.

3.3. Rezultate

În secțiunea curentă este descrisă modalitatea prin care au fost aleși diferiți parametrii sau funcții pentru optimizarea modelului. În plus, rezultatele vor fi analizate

cu ajutorul unor ilustrații ce conțin evoluția costului și acurateței, tabele cu valori, dar și matricele de confuzie specifice.

3.3.1. Procesul testării

Pentru început au fost aleși parametrii uzuali folosiți în domeniu. Fiecare rețea a avut la dispoziție 100 de epoci. Prima rețea testată este ResNet18, cu costurile sau greutățile modelului preantrenat. Optimizatorul folosit este Adam, cu rata de testare de început egală cu 10^{-3}. Funcția de cost rămâne aceeași pentru toate experimentele și este CrossEntropyLoss(), iar mărimea lotului de date este 4. Pe parcursul antrenării, au fost descoperiți alți parametrii care ajutau modelul să funcționeze mai bine, iar experimentarea care a continuat din acel punct a fost făcută cu noii parametrii.

Rețelele testate sunt ResNet18, Resnet34, ResNet50 și SuffleNetv2_X1_0 (la care se va face referire prin scurtarea denumirii la ShuffleNetV2), cu greutățile preantrenate, în afară de ResNet18, a cărui antrenare a avut loc și fără valorile prestabilite. Valorile ratei de învățare încercate sunt 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5} și 10^{-6}. Mărimile lotului sunt puteri ale lui 2: 4, 8, 16 și 32. Optimizatorii folosiți sunt Adam, AdamW, RMSProp și Adagrad.

3.3.2. Rata de învăţare

Rata de învățare	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}
Acuratețe - validare	0.64285	0.88095	0.99206	0.98809	0.92460
Acuratețe - învățare	0.5985	0.8725	0.976	0.9895	0.8385
Cost - validare	0.67648	0.27725	0.04063	0.07433	0.22216
Cost - învățare	0.69602	0.28410	0.07564	0.03421	0.43205
Epoca optimă	62	94	66	96	91

Tabelul 3.1

Rata de invățare

Experimentarea ratei de învățare in ResNet18

Tabelul 3.1 conține rezultatele obținute prin testarea diferitelor rate de învățare pentru o rețea de tip ResNet18, cu mărimea lotului egală cu 4 și optimizatorul Adam. O rată de învățare de 10^{-4} reprezintă valoarea optimă pentru antrenare, dar și cu rata de

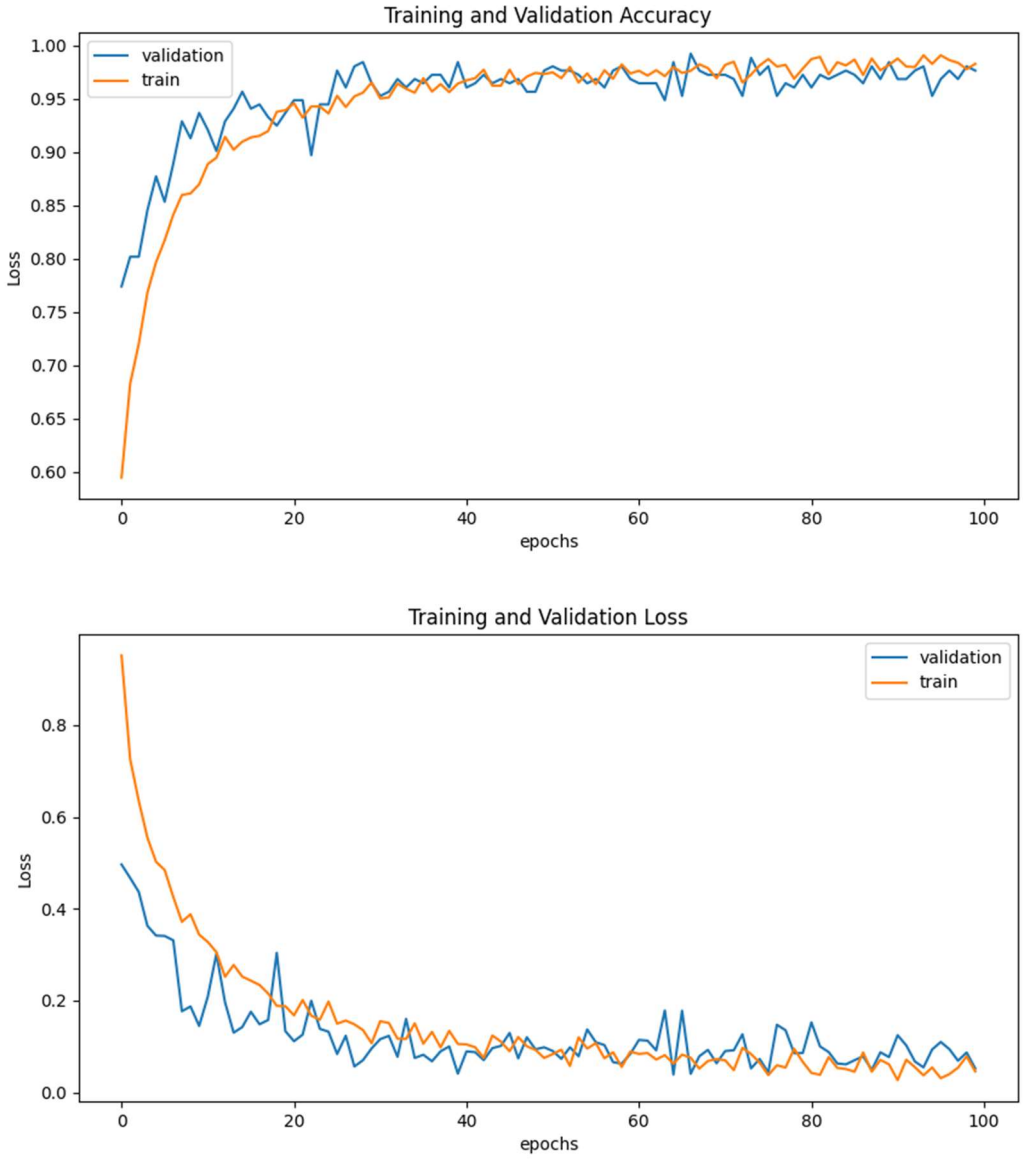

Figura 3.10

Grafice pentru acurateţe şi cost în ResNet18

Arată evoluţia valorilor pe parcursul celor 100 de epoci
cu rata de învăţare 10^{-4}

10^{-5} au fost obţinute valori apropiate. A fost aleasă folosirea primei rate menţionate pentru următoarele modele deoarece acurateţea la validare este mai mare decât restul datelor, dar şi numărul epocii optime este semnificativ mai mic. În cazul ratelor mai mari decât 10^{-4}, modelul nu reuşeşte să găsească minimul necesar, sărind peste el din cauza paşilor prea mari. Începând cu valori mai mici, de exemplu 10^{-6}, modelul este lent şi nu are destul timp pentru a ajunge la un minim optim. Un număr mai mare de epoci ar putea rezolva problema, însă nu este necesară experimentarea suplimentară deoarece

există deja alte valori care găsesc rezultate bune în cele 100 de epoci.

Din figura 3.10 reiese că modelul nu este supraadaptat, valorile pentru validare și învățare fiind relativ similare. Pentru costul validării este observabil că există o oarecare fluctuație în valorile preluate, mici spike-uri în care numărul crește sau descrește brusc. O cauză ar putea fi mărimea modestă a setul de date, care nu este suficient de complex pentru a ajuta modelul să generalizeze mai bine. O altă modalitate prin care situația ar putea fi îmbunătățită este prin scăderea ratei de învățare. Astfel, în imaginea de mai jos (Fig. 3.11), pentru o rata de 10^{-5}, poate fi observată o stabilitate mai bună a modelului.

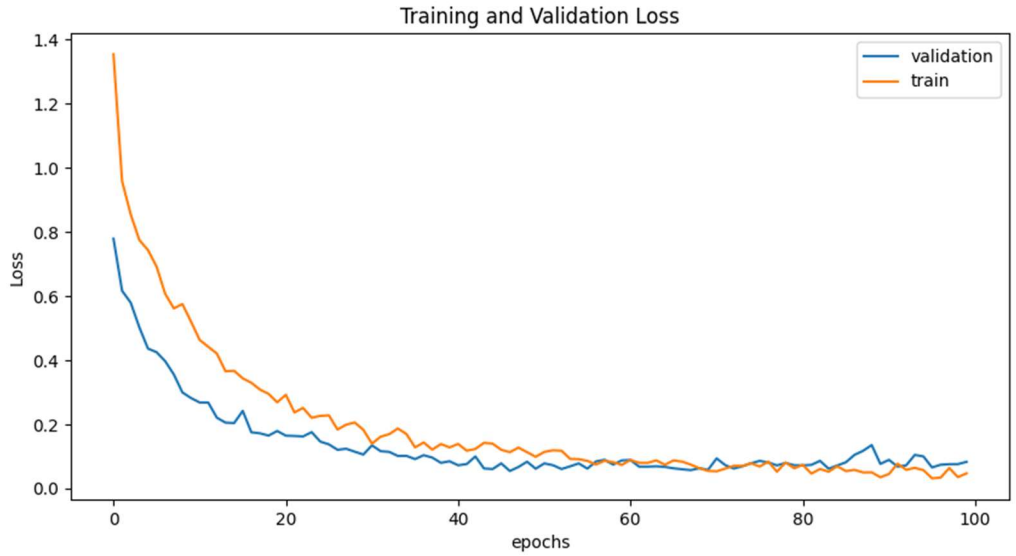

Figura 3.11

Evoluția costului rețelei ResNet18

Arată evoluția costului pe parcursul celor 100 de epoci cu rata de învățare 10^{-5}

3.3.3. Mărimea lotului

Dimensiunea lotului determină numărul de imagini care sunt procesate într-o iterație. Timpul de rulare devine mai scurt odată cu creșterea mărimii lotului deoarece pe o placă video pot fi executate mai multe procese în paralel, în detrimentul memoriei care este ocupată mai rapid. Din datele tabelului 3.2 reiese că cea mai potrivită valoare pentru mărimea lotului este 8. În majoritatea următoarelor rezultate vă fi folosit acest număr în mod implicit.

Mărimea lotului	4	8	16	32
Acuratețe - validare	0.99206	0.99218	0.98984	0.98828
Acuratețe - învățare	0.976	0.9825	0.9895	0.99652
Cost - învățare	0.07564	0.04986	0.03447	0.01314
Cost - validare	0.04063	0.02982	0.10356	0.06523

Tabelul 3.2

Dimensiunea lotului

Experimentarea mărimii lotului în ResNet18

3.3.4. Folosirea altor rețele

Rețea	ResNet18	ResNet34	ResNet50	ShuffleNetV2
Acuratețe - validare	0.99218	0.98828	0.99609	0.99218
Acuratețe - învățare	0.9825	0.9865	0.9875	0.9855
Cost - validare	0.02982	0.04580	0.03221	0.03506
Cost - învățare	0.04986	0.03738	0.04466	0.03784
Epoca optimă	76	59	43	86

Tabelul 3.3

Compararea rețelelor

Rezultate obținute pentru 4 rețele diferite

Toate rețelele au obținut rezultate similare, ResNet50 având cea mai mare valoare pentru validare. În plus, a reușit să găsească rezultatul optim mai rapid decât restul rețelelor, la epoca 43. Diferența dintre ResNet18, ResNet34 și ResNet50 constă în numărul de straturi folosit, precizat în denumirea fiecăreia dintre arhitecturi. Un număr mare de straturi poate avea la ieșire o acuratețe mai bună, dar timpul de execuție este mai lent. Pentru ResNet50, folosirea unui lot de 4 imagini determină că iterația unei epoci să dureze un minut și 10 de secunde, în timp ce o epocă pentru ResNet18 este executată în 25 de secunde. Totuși, cu folosirea unui lot cu 8 date, ResNet50 rulează o epocă în 40 de secunde.

A fost testată și rețeaua ResNet18 fără greutățile predefinite. Acuratețea validării a fost de 0.89285, puțin mai slabă decât cea a modelelor preantrenate. Pentru obținerea unei valori mai bune, numărul de epoci ar putea fi mărit, modelul având mai mult timp la dispoziție să se antreneze.

ShuffleNetV2 este cunoscută prin rapiditatea sa, reușind să mențină o acuratețe bună

[17] . Este o rețea care obține rezultate relativ asemănătoare, în Tabelul 3.3 obținând aceeași acuratețe de validare că ResNet18, dar cu cost computațional mai mic. Arhitectura ShuffleNetV2 are o arhitectura mai simplă decât ResNet18, ceea ce o face mai rapidă.

Din graficele din figura 3.12, poate fi observat că pentru ShuffleNetV2 costurile de învățare și validare descresc încontinuu și sunt foarte similare că valori, deci nu are loc supraadaptarea. Chiar dacă ResNet50 găsește un cost mai mic, este mai instabil și pe alocuri costul validării este mai mare decât costul din perioada învățării.

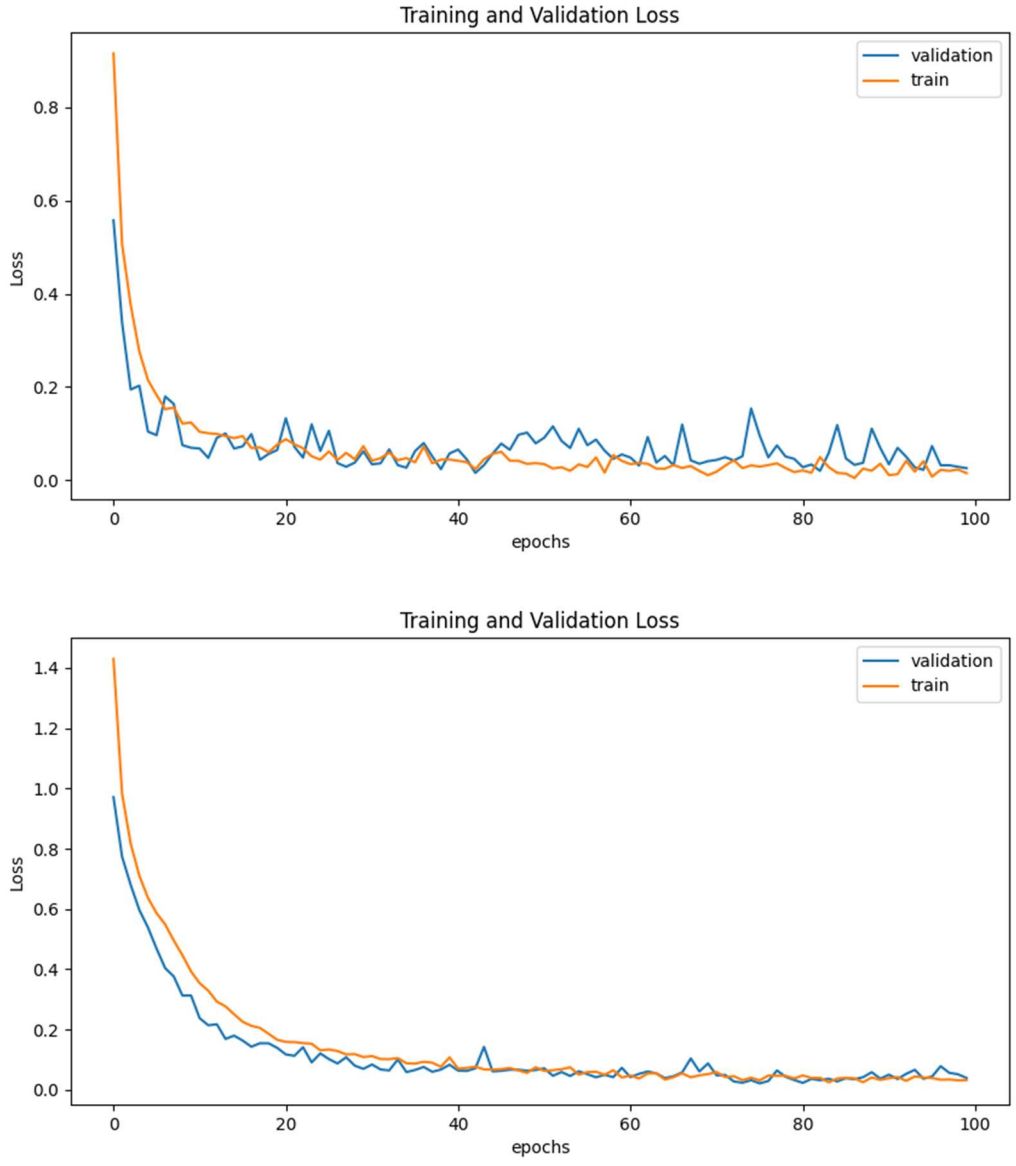

Figura 3.12

Evoluția costurilor pentru ResNet50 si ShuffleNetV2

Arata evoluția costului, rata de învățare este 10^{-4} și mărimea lotului 8

Pentru testare, cel mai bun rezultat obținut este al rețelei ResNet50 (Fig. 3.13). Pentru imaginile etichetate ca „Low” și „Normal”, a avut o acuratețe de 98%, în timp ce pentru restul etichetelor este de 100%. Astfel, dintr-un total de 250 de imagini

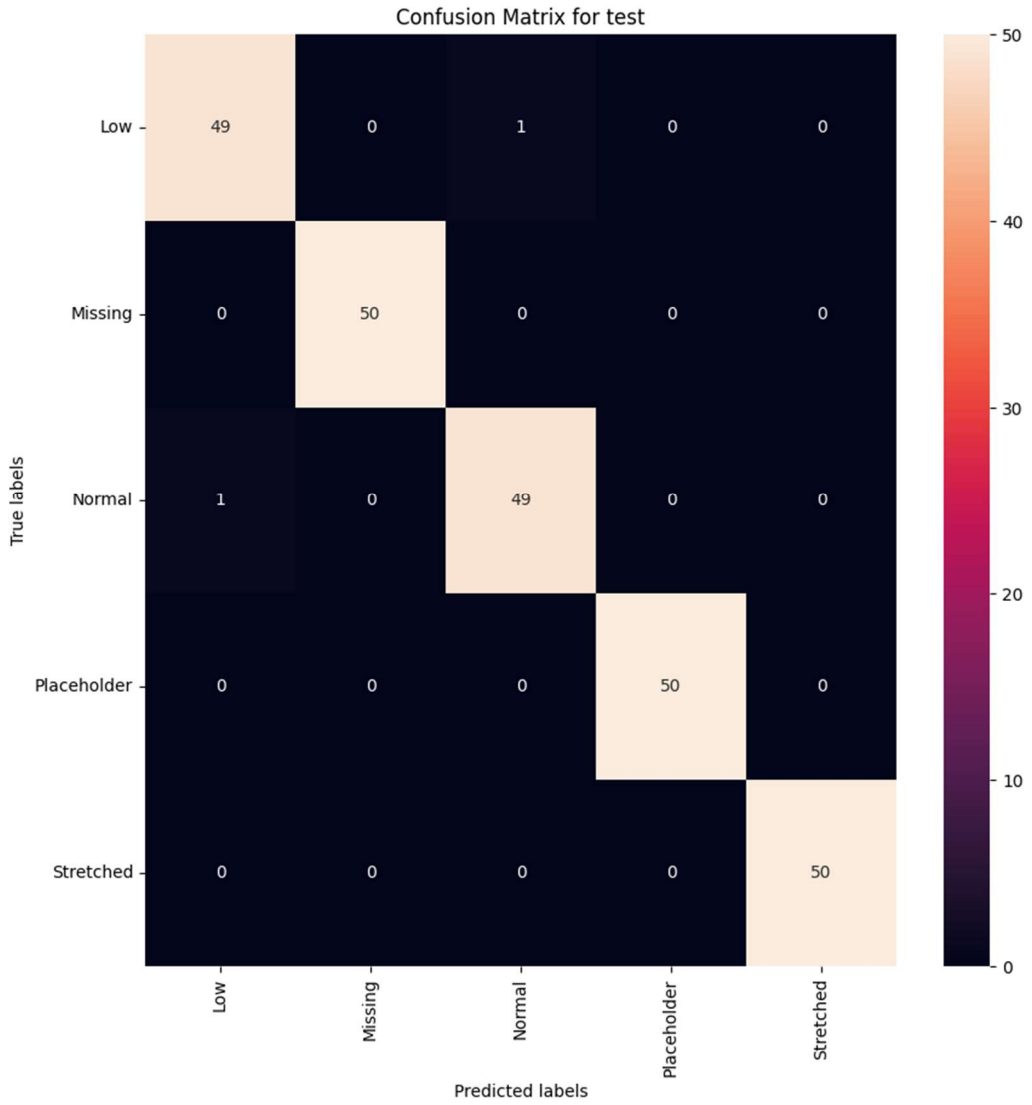

Figura 3.13

Matricea de confuzie pentru ResNet50

Matricea de confuzie pentru setul de testare

disponibile pentru testare, a prezis greșit doar o imagine din cele 50 din categoria „Low”, respectiv „Normal”. În cazul validării, au mai fost prezise greșit în plus două imagini cu textura „Stretched”. Pentru învățare, modelul a categorisit greșit doar 5 imagini din cele 2000. În general, la nivelul tuturor arhitecturilor, procentele mai mici sunt obținute pentru imagini fără defecte, cu textura de calitate scăzută și cele cu textura întinsă, iar

în destule cazuri, cele de tip „Placeholder” și „Missing” sunt determinate corect. Din cauza asemănărilor, sunt mai greu de prezis deoarece nu există o culoare evidentă care să le diferențieze, așa cum se întâmplă în cazul celor cu textura lipsă sau înlocuită cu o culoare albă.

3.3.5. Alți optimizatori

Optimizatori	Adam	AdamW	Adagrad	RMSProp
Acuratețe – validare	0.99218	0.98828	0.69921	0.99218
Acuratețe – învățare	0.9855	0.983	0.5965	0.9735
Cost – validare	0.03506	0.03195	1.29849	0.04831
Cost - învățare	0.03784	0.04106	1.34299	0.06966
Epoca optimă	86	85	24	60

Tabelul 3.4

Compararea optimizatorilor

4 optimizatori testati pe reteaua ShuffleNetV2

Pe lângă optimizatorul principal folosit Adam, au fost testați alți 3 optimizatori: AdamW, Adagrad și RMSProp. Adagrad prezintă o rată de învățare adaptivă, care ajustează valoarea ei în așa fel încât să fie redusă mai mult pentru descreșteri abrupte decât în cazul celor mai lente. O problemă în Adagrad constă în faptul că rata de învățare descrește atât de mult încât algoritmul se poate opri în totalitate, de aceea și rezultatul mult mai mic decât restul valorilor. RMSProp rezolvă problema prin acumularea gradienților doar din iterațiile recente. Adam reprezintă o combinație dintre Adagrad și RMSProp, iar AdamW este o completare pentru Adam care decuplează scăderea greutăților de actualizarea gradienților [10]. Din tabelul 3.4 reiese că Adagrad nu funcționează atât de bine, în timp ce RMSProp, care este popular în folosirea clasificării de imagini, atinge aceeași acuratețe la validare că și Adam.

3.3.6. Precizia si recallul

Din tabelul 3.5 reiese că toate modelele obțin rezultate bune pentru precizie și recall. Pentru imaginile cu eticheta „Normal”, toate arhitecturile identifică câteva imagini că fiind fără glitchuri, în mod eronat. În plus, și cele cu eticheta „Low” sunt identificate în totalitate corect doar de către ResNet18. Rezultatele sunt în conformitate cu matricele de confuzie, clasele „Placeholder” și „Stretched” fiind cele mai ușoare de etichetat, iar restul având

rezultate imperfecte în anumite cazuri, din cauza asemănării lor. ResNet18 a performat în mod excepțional, obținând pentru 4 din cele 5 etichete valori maxime. În plus, arhitecturile ResNet au obținut rezultate mai bune față de ShuffleNet, care a avut câteva scoruri mai mici în anumite cazuri.

Precizie și recall	Precizie(low, missing, normal, placeholder, stretched)					Recall(low, missing, normal, placeholder, stretched)				
ResNet18	1.0	1.0	0.94	1.0	1.0	0.94	1.0	1.0	1.0	1.0
ResNet34	0.94	1.0	0.9412	1.0	1.0	0.94	1.0	0.96	1.0	0.98
ResNet50	0.98	1.0	0.98	1.0	1.0	0.98	1.0	0.98	1.0	1.0
ShuffleNetV2	0.9792	1.0	0.9074	1.0	1.0	0.94	1.0	0.98	1.0	0.96

Tabelul 3.5

Precizie si recall

Precizia și recall-ul după 100 de epoci

3.3.7. Rezultate obţinute în alte lucrări

O lucrare cu o metodă similară de lucru este cea scrisă de către Garcia Ling – „Graphical Glitch Detection in Video Games Using CNNs” [2]. Rezultatele conțin comparări asemănătoare de valori și sunt folosite aceleași rețele.

O diferență majoră constă în complexitatea setului de date. Setul său cuprinde 12700 imagini de mărime 800x800 RGB, iar obiectele sunt însoțite de un fundal pentru a fi încadrate într-un mediu natural din joc.

Din cauza complexității datelor, tipurile de glitchuri sunt prezise mai greu, dar imaginile se apropie mai mult de realitatea din jocuri decât cele din setul de date propriu. O dificultate întâmpinată de către autorul lucrării a fost segmentarea semantică a obiectelor, nereușind să găsească o metodă pentru a genera o mască de segmentare. Problema poate fi rezolvată ușor tot prin pachetul Unity pentru percepție, care prezintă posibilitatea de a adăuga o eticheta ce atașează un tag obiectului, urmând ca la generarea imaginilor să fie colorat cu o culoare precizată.

Din punct de vedere al rezultatelor propuse, mărimile loturilor obțin rezultate similare, iar valoarea prestabilită este 8. Rata de învățare este aceeași, fiind aleasă valoarea de 10^{-4}. Arhitectura principală testată este ShuffleNetV2, care ajută prin rapiditatea ei, dat fiind setul de date complex.

ShuffleNetV2	Precizie
Stretched	80.0%
Low	64.3%
Missing	99.2%
Placeholder	97.0%

Tabelul 3.6

Precizie în alte lucrări

Cel mai bun rezultat obținut cu configurație optimă [2]

Din tabelul 3.6 poate fi dedus că și în cazul modelului lui Ling, glitchurile de tip lipsă sau înlocuitor sunt cele care sunt identificate cel mai ușor. În plus, din faptul că cea mai bună acuratețe a sa are o valoare aproximativă de 0.72 [2], reiese că modelul nu a fost la fel de eficient ca cel propriu, dar trebuite luat în considerare și factorul de complexitate al setului de date.

Un alt aspect de menționat este cazul în care obiectul de identificat este în mare parte alb. Din această cauză poate fi confundat ușor cu un glitch de tip „Placeholder”. Prin coincidență, setul de date creat nu conține o imagine cu un obiect alb. Un mod prin care problema poate fi rezolvată este folosirea unui alt tip de înlocuitor cu o textură diferită, de exemplu unul în care să fie un text adăugat: „Debug” [2].

4. Alte metode

În capitolul curent vor fi discutate alte metode pentru detecția glitchurilor în jocurile video. În plus, sunt menționate mai multe categorii de erori sau tipuri de jocuri care nu au făcut parte din obiectivul lucrării personale. Scopul capitolului este să descrie pe scurt alte lucrări din domeniu și să prezinte eventuale soluții sau legături cu abordarea glitchurilor descrisă până acum.

4.1. Jocuri în HTML5 <canvas>

Există posibilitatea creării unor jocuri în pagini web cu ajutorul HTML5 canvas. De obicei sunt jocuri simple, care se pot desfășura în browser, programate cu ajutorul HTML, dar și cu alte limbaje precum CSS sau Java Script.

O metodă uzuală pentru detecția erorilor în acest tip de joc este testarea instantanee (snapshot testing). Metoda constă în compararea unor capturi de ecran din aplicație cu anumite cadre pentru a detecta anumite diferențe, dar capturile folosite pentru testare sunt verificate manual. Problema acestei abordări constă în faptul că unele diferențe dintre imagini nu reprezintă erori în sine, ci fac parte din dinamica jocului [18].

În lucrare sunt prezentate două capturi de ecran din jocul autorului în perioada de testare, alături de imaginea cu rezultatul. În a doua captură, eroarea se află la nivelul mâinilor personajului viking, căruia îi lipsește un buștean din brațe. Chiar dacă algoritmul identifică problema în imaginea rezultată, el surprinde absența unor obiecte care nu conțin un anumit glitch, dar care doar lipsesc la momentul respectiv. Soluția propusă în lucrare este selectarea obiectelor din joc care pot conține glitchuri și aplicarea unei măști prin care să fie comparate diferențele. Sunt injectate erori precum scăderea în calitate a vikingului, colorarea diferită a unor elemente sau poziționarea diferită a lor [18].

Pot fi identificate câteva similarități cu metoda proprie, cea mai evidentă fiind selectarea separată a obiectelor pentru a fi trimise către testare. În plus, anumite tipuri de glitchuri sunt asemănătoare, precum scăderea în calitate a unui obiect sau folosirea altei culori în textură. Un nou tip de glitch care apare aici îl reprezintă absența unui obiect, care nu este tratată în setul propriu descris în lucrare. Metoda ce folosește deep

learning poate fi folosită în detecția erorilor în jocurile web. De exemplu, poate fi creat un model care să determine diferențele dintre două poze. În cazul modelului descris până acum în lucrare, setul de date poate fi împărțit în imagini cu eticheta „Normal” și imagini cu eticheta corespunzătoare tipului de glitch, însă ar trebui gândit un proces prin modelul să extragă diferite obiecte din imagine, de exemplu prin segmentare semantică.

4.2. Artefacte în jocurile video

Artefactele în jocurile video seamănă cu glitchurile prin faptul că produc erori la nivel vizual, însă de obicei sunt produse din cauza suprasolicitării plăcii video. Ce este de remarcat aici este faptul că acest tip de erori nu au neapărat legătură cu obiectele din joc.

În lucrarea „Automating artifact detection în video games”, sunt prezentate câteva exemple de artefacte și modul în care pot fi detectate. În primul rând, a fost creată o aplicație „Glitchify”, prin care pot fi reproduse erorile asupra unor imagini din jocuri selectate din videoclipurile de pe Youtube [19]. Aplicația însă nu tratează glitchurile din setul propriu de date, dar este utilă pentru a exemplifica și alte tipuri de probleme. De exemplu, un tip de artefact este produs din cauza coruperii shaderului, care este responsabil pentru randarea suprafețelor precum texturile sau lumina [19]. Este sesizabil faptul că acest tip de eroare nu are loc la nivelul unui obiect, ci are efect asupra întregului cadru. Un artefact care poate afecta un obiect în mod direct este decolorarea, care înlocuiește textura sa cu o culoare stridentă . Alte tipuri de artefacte menționate în lucrare sunt cele de formă, modelul codului morse, linii punctate, linii paralele, triangularea, pixelarea liniei, stutteringul ecranului și ”ruperea” sa [19]. Modelul codului morse face referire la blocarea celulelor de memorie care afișează aceleași valori în loc de cele noi. Liniile punctate sunt formate din pixeli cu culori anormale, iar liniile paralele sunt segmente colorate care se întind, pornind dintr-un anumit punct. Fenomenul de triangulare apare de obicei în jocurile 3D, unde suprafețele sunt randate ca mai multe triunghiuri. Pixelarea liniilor se referă la linii din mai mulți pixeli de culori aleatoare. Artefactul ce descrie stutteringul ecranului inversează liniile și coloanele vecine din imagine, iar ruptura ecranului îmbină două cadre consecutive în aceeași imagine [19].

Rezolvarea abordată în lucrare este specifică domeniului de machine learning și folosește un model de ansamblu, care constă în îmbinarea mai multor modele.

Algoritmul propriu poate fi folosit dacă imaginile ar fi etichetate, însă ar putea obține rezultate mai slabe din cauza complexității setului de date.

4.3. Reinforcement learning

Un agent de reinforcement learning reprezintă o metodă prin care inteligența artificială interacționează cu un anumit mediu, încercând să mimeze un comportament uman. Practic, un agent învață din propriile acțiuni și rezultatele lor, maximizând o valoare care îi dictează succesul în rezolvarea unei anumite probleme [20].

În lucrarea „Inspector: Pixel-Based Automated Game Testing via Exploration, Detection, and Investigation" este descris modul în care agentul este capabil să exploreze un mediu virtual și să interacționeze, asemănător unui om, cu obiecte din joc [21]. De exemplu, într-un joc de tip shooter, algoritmul controlează caracterul principal și îl ghidează spre un health pack, pe care îl înconjoară pentru a-l studia și interacționă cu el în diferite moduri.

Una dintre erorile descoperite de către agent are loc atunci când controlează un caracter și îl aproprie de o stâncă pentru a o inspecta. La următoarea înaintare a personajului, el pătrunde în spatele texturii pietrei. Un alt glitch descoperit are loc atunci când personajul principal descoperă un avatar care plutește în aer, fără să fie susținut de o suprafață [21].

Ceea ce este interesant la abordarea prin reinforcement learning este că pe lângă capacitatea de a detecta glitchuri, agentul este capabil să le descopere și să le analizeze individual. Modelul descris prin deep learning are posibilitatea de a studia imagini simple, iar erorile se află la nivel de textură. Cazul în care ar primi o imagine pentru a detecta faptul că un jucător se află în aer ar putea fi prea complex de clasificat pentru modelul propus.

5. Concluzii

5.1. Îmbunătățiri posibile

Primul aspect ce va fi explorat se referă la complexitatea setului de date. Există un număr restrâns de imagini în set, iar creșterea totalului de date ar ajuta modelul să desprindă o diversitate mai mare de caracteristici din poze. Aici pot fi folosite mai multe categorii de obiecte, care fac parte din pachete sau jocuri cu tematici mult mai diverse. În plus, ar trebui adăugat un fundal sau un mediu obiectelor pentru simula cadrul în care s-ar găsi în joc. Obiectelor li s-ar putea aplica mai multe transformări, precum schimbarea aleatoare a poziției în cadru. Automatizarea procesului de creare a setului de date ar putea fi îmbunătățită: extragerea fiecărui obiect și etichetarea fiecărei imagini în parte poate fi un proces destul de lent, în special pentru generarea unui set mult mai mare de date. De asemenea, poate fi încercată extinderea setului în așa fel încât să includă mai multe tipuri de glitchuri, cum ar fi și cele care nu au loc doar asupra unui obiect, asemenea celor menționate în capitolul precedent. Un alt aspect de încercat ar putea fi selectarea automată a unor cadre din jocuri care să fie adăugate în set și să conțină glitchuri, însă este probabil ca setul să necesite verificare manuală.

Din punct de vedere al modelului, poate fi antrenat pe parcursul a mai multor epoci. Au fost alese doar 100 de epoci deoarece au fost obținute rezultate bune în această perioadă, dar creșterea complexității setului va influența și timpul necesar pentru învățare. În plus, ar ajuta colectarea altor tipuri de rezultate, de exemplu ar fi de folos vizualizarea imaginilor care au fost etichetate eronat deoarece ar oferi o perspectivă mai largă asupra exemplelor problematice sau greu de categorisit de către model.

5.2. Critică asupra rezultatelor

Rezultatele obținute descriu o probabilitate foarte mare de reușită a modelului în categorisirea datelor, însă este important de menționat nivelul relativ scăzut de complexitate al setului. Totuși, metodele încercate pot fi testate și pentru alte tipuri de seturi și ajustate în funcție de rezultate. Aplicația reprezintă un punct de start care, din perspectivă proprie, poate fi dezvoltat ușor către o direcție mai complexă.

Bibliografie

[1] Bainbridge, W. A., & Bainbridge, W. S. (2007). Creative uses of software errors: Glitches and cheats. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=afc601b6d7560e44f28256c37fc642ba7d335783

[2] Ling, C., Tollmar, K., & Gisslén, L. (2020). Using deep convolutional neural networks to detect rendered glitches in video games. URL: http://kth.divaportal.org/smash/get/diva2:1432668/FULLTEXT01.pdf

[3] Esposito, N. (2005). A Short and Simple Definition of What a Videogame Is. URL: http://www.digra.org/wp-content/uploads/digital-library/06278.37547.pdf

[4] Block, S., & Haack, F. (2021). eSports: a new industry. URL: https://www.shsconferences.org/articles/shsconf/pdf/2021/03/shsconf_glob20_04002.pdf

[5] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. URL:https://books.google.ro/books?hl=ro&lr=&id=omivDQAAQBAJ&oi=fnd&pg=PR5&dq=deep+learning&ots=MNU3gvmAPU&sig=x--XTLAEqm6C8UVwmFBlcuJ2yEo&redir_esc=y#v=onepage&q=deep%20learning&f=false

[6] Shen, D., Wu, G., & Suk, H. I. (2017). Deep learning in medical image analysis. URL: https://www.annualreviews.org/doi/full/10.1146/annurev-bioeng-071516-044442

[7] Shorten, C., & Khoshgoftaar, T. M. (2019). A survey on image data augmentation for deep learning. URL: https://journalofbigdata.springeropen.com/counter/pdf/10.1186/s40537-019-0197-0.pdf

[8] Murdoch, W. J., Singh, C., Kumbier, K., Abbasi-Asl, R., & Yu, B. (2019). Interpretable machine learning: definitions, methods, and applications. URL: https://arxiv.org/abs/1901.04592

[9] Ying, X. (2019, February). An overview of overfitting and its solutions. URL: https://iopscience.iop.org/article/10.1088/1742-6596/1168/2/022022/meta

[10] Géron, A. (2017). *Hands-on machine learning with scikit-learn and tensorflow: Concepts. Tools, and Techniques to build intelligent systems.*

[11] URL: https://pytorch.org/vision/main/generated/torchvision.datasets.ImageFolder.html

[12] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. URL:https://arxiv.org/abs/1512.03385

[13] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. URL: https://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf?utm_content=buffer79b43&utm_medium=social&utm_source=twitter.com&utm_campaign=buffer,
[14] Ajagekar, A. (2021). Adam. URL: https://optimization.cbe.cornell.edu/index.php?title=Adam
[15] URL: https://github.com/tqdm/tqdm
[16] URL: https://pytorch.org/docs/stable/generated/torch.save.html
[17] Zhang, X., Zhou, X., Lin, M., & Sun, J. (2018). Shufflenet: An extremely efficient convolutional neural network for mobile devices. URL: https://openaccess.thecvf.com/content_cvpr_2018/papers/Zhang_ShuffleNet_An_Extremely_CVPR_2018_paper.pdf
[18] Macklon, F., Taesiri, M. R., Viggiato, M., Antoszko, S., Romanova, N., Paas, D., & Bezemer, C. P. (2022). Automatically Detecting Visual Bugs in HTML5 Canvas Games.
URL: https://arxiv.org/abs/2208.02335
[19] Davarmanesh, P., Jiang, K., Ou, T., Vysogorets, A., Ivashkevich, S., Kiehn, M., ... & Malaya, N. (2020). Automating artifact detection in video games. URL: https://arxiv.org/pdf/2011.15103v1.pdf
[20] Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. URL: https://books.google.ro/books?hl=ro&lr=&id=uWV0DwAAQBAJ&oi=fnd&pg=PR7&dq=Reinforcement+learning&ots=mivJs1Z0i1&sig=3r4fXOQU0jycYtNV6Ocghj3Theg&redir_esc=y#v=onepage&q=Reinforcement%20learning&f=false
[21] Liu, G., Cai, M., Zhao, L., Qin, T., Brown, A., Bischoff, J., & Liu, T. Y. (2022). Inspector: Pixel-Based Automated Game Testing via Exploration, Detection, and Investigation. URL: https://arxiv.org/abs/2207.08379