UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Învățarea folosind rețele neuronale pentru jocul Flappy Bird

propusă de

Adrian Petercă

Sesiunea: iunie, 2022

Coordonator științific

Asist. Dr. Croitoru Eugen

UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

Învățarea folosind rețele neuronale pentru jocul Flappy Bird

Adrian Petercă

Sesiunea: iunie, 2022

Coordonator științific

Asist. Dr. Croitoru Eugen

Cuprins

Motivație **2**

Introducere **4**

1 Descrierea aplicației **5**

1.1 Punctul de start 5

1.2 Descrierea setărilor posibile 6

1.3 Procesul de evoluție 6

1.4 Cazuri excepționale 7

2 Implementarea jocului **9**

2.1 Generarea automată a nivelelor 10

2.2 Tipuri de scor 10

2.3 Idei de îmbunătățire pe viitor 11

3 Implementarea antrenării **13**

3.1 Clasa Matrix 13

3.2 Clasa NeuralNetwork 15

3.3 Clasa GeneticAlgorithm 18

3.4 Parametrii de input/output 19

4 Dificultăți întâmpinate în dezvoltarea aplicației **21**

4.1 Lipsa unui set de date de antrenament 21

4.2 Configurarea rețelei neuronale 22

4.2.1 Structura 22

4.2.2 Funcții de activare 24

4.3 Configurarea algoritmului genetic 27

4.3.1 Operatorul de selecție 27

4.3.2 Operatorul de mutație 28
4.3.3 Operatorul de încrucișare 29
4.4 Speed-up 33
4.5 Absența unei librării standardizate 33

Concluzii **35**

Bibliografie **37**

Motivație

Conceputul de self learning în jocurile video reprezintă o idee abordată pentru prima dată în anii '50, pentru jocuri simple dar care prezintă o vastă arie de mutări posibile (de exemplu, jocul de Checkers [1]) și pentru care un algoritm determinist ar fi prea dificil de implementat. Prin această abordare se căuta rezolvarea unor probleme prin aproximarea unei soluții optime, soluție mai ușor de calculat față de una perfectă, mergându-se pe ideea că "done is better than perfect, because perfect never gets done". Astfel, s-a ajuns în prezent ca majoritatea jocurilor mari să se folosească de aceste inovații, precum: path-finding-ul din cadrul unui NPC (non-playable character), motion tracking (deseori folosit în jocuri de tip "Shooter"), decision making sau chiar generarea de nivele unice.

Plecând de la aceste întâmplări, am ales să folosesc cunoscutul joc de Flappy Bird[2] pentru a evidenția procedeul de implementare și dezvolatare a unui astfel de algoritm de învățare, plecând de la zero. Am considerat că implementând personal atât jocul suport, cât și algoritmul în sine, gradul de înțelegere al conceptului va fi mai ridicat, iar problemele ce vor apărea mă vor face să înțeleg mai în detaliu ce anume se întâmplă în spate.

Pe scurt, Flappy Bird este un joc 2D (camera jucatorului nu are perspectivă de adâncime) în care jucătorul trebuie să se ferească de o infinitate de "pipe"-uri generate automat pe diferite nivele de înălțime, urmărind să obțină un scor cât mai mare. În situația în care jucătorul atinge un asemenea "pipe", jocul se va termina, putând fi reluat într-o viitoare încercare.

Algoritmul folosit este cunoscut sub numele de NeuroEvolution, un algoritm genetic care folosește rețele neuronale artificiale pe post de cromozomi pentru a evolua populația de start. Pe scurt, pornind de la o populație de X rețele neuronale arficiale, se aplică o funcție de fitness pe acești indivizi, după care se "amestecă" - anume, se aplică operatori genetici, precum cross-over, mutație sau selecție - pentru a se obține

următoarea generație. În capitolele următoare, acest procedeu va fi descris mai pe larg, pornind de la implementările inițiale până la versiunea finală.

Îmbinând aceste concepte, aplicația simulează evoluția unei populații de indivizi (care sunt, în sine, jucători ce iau decizii inițial aleatorii) până ajunge la un anumit rezultat țintă.

Introducere

Lucrarea de față aduce în discuție un subiect contemporan și de mare interes pentru orice pasionat de inovare și dezvoltare în domeniul inteligenței artificiale, anume utilizarea unei rețele neuronale artificiale pentru a coordona mișcările unei ”păsări” în cadrul cunoscutului joc Flappy Bird.

Intitulată **FlappyBird-NES** (acronim de la NeuroEvolution Simulation), lucrarea în sine constă într-o aplicație care simulează antrenamentul unui jucător controlat de o rețea neuronală artificială obținută prin abordarea unui algoritm genetic care selectează acei indivizi care au cel mai bun scor într-o iterație a jocului. Redactată în C# și proiectată în Unity, aplicația poate rula pe diverse configurații de Windows, având o interfață user-friendly ce permite selectarea unor setări de bază pentru a observa procedeul de simulare sau pentru a testa un individ deja antrenat.

La momentul redactării, aplicația se află în versiunea `4.2`, urmând ca pe viitor să fie adăugate diverse îmbunătățiri.

La nivelul conținutului, lucrarea de față va detalia tot procesul dezvoltării aplicației în cauză, pornind de la sursele de inspirație și ideile de început până la alegerile de design și provocările ce au apărut la fiecare pas.

Capitolul 1

Descrierea aplicației

Aplicația dispune momentan de suport integral pe Windows, putând fi descărcată de pe GitHub[3] pentru a putea fi testată local. Nu este nevoie de instalare sau alte descărcări.

1.1 Punctul de start

Problema dezvoltării unui algoritm de auto-învățare nu este una nouă, după cum este descris și în Introducere. Multe alte jocuri au astfel de implementări, folosind diverse tehnologii, precum **MarI/O**[4] care utilizează conceptul de NEAT (NeuroEvolution with Augmented Topologies), **Snake**, antrenat prin DQL (Deep Q Learning) pentru a încerca să obțină un scor perfect (toată tabla de joc să fie acoperită) sau chiar **Șah** folosind CNN (Convoluted Neural Network). Aceste metode de învățare provoacă autorul spre a găsi un echilibru între algoritmul propriu-zis și elementele importante care trebuie transmise ca input.

Lucrarea de față este inspirată din multe astfel de proiecte, ieșind în evidență prin simplitatea sa (nefolosirea de librării externe sau algoritmi deja implementați) și prin nivelul de modularizare posibil. Totodată, alți autori au venit cu implementări proprii, precum Daniel Shiffman, fondatorul canalului *The Coding Train*[5], care a ales să abordeze problema într-un mediu Web, folosind JavaScript și p5.js, păstrând ideea de NeuroEvolution.

1.2 Descrierea setărilor posibile

Odată pornită aplicația, se pot observa 3 module:

- evoluția standard - aceasta presupune antrenarea generației de start (a cărei dimensiune este configurabilă) până la un anumit număr de generații stabilit înaintea începerii simulării;
- evoluția pe scor - aceasta presupune antrenarea generației de start (a cărei dimensiune este configurabilă) până la un anumit scor stabilit (sau până un individ depășește acest scor);
- modulul de testare - permite încărcarea unui individ deja antrenat pentru a se observa capabilitățile sale.

Pe lângă cele menționate anterior, modulele de evoluție pot începe simularea pornind de la o generație aleatoare (setarea de bază) sau de la un individ model (populația de start va fi astfel identică, urmând se evolueze îndependent pe parcursul simulării).

1.3 Procesul de evoluție

Pentru o înțelegere a procesului de evoluție mai simplificată, aplicația dispune de câteva statistici de bază, precum: un cronometru, numărul generației curente, numărul de generații final (în cazul evoluției standard), scorul curent (numărul total de pipe-uri trecute), scorul maxim curent, scorul țintă (în cazul evoluției pe scor) și numărul de indivizi rămași.

Pe baza acestor măsurători au fost făcute îmbunătățiri atât la configurația rețelei neuronale, cât și la parametrii și operatorii algoritmului genetic, pentru a optimiza procesul de evoluție. Astfel, se poate observa o îmbunătățire drastică de la versiune la versiune.

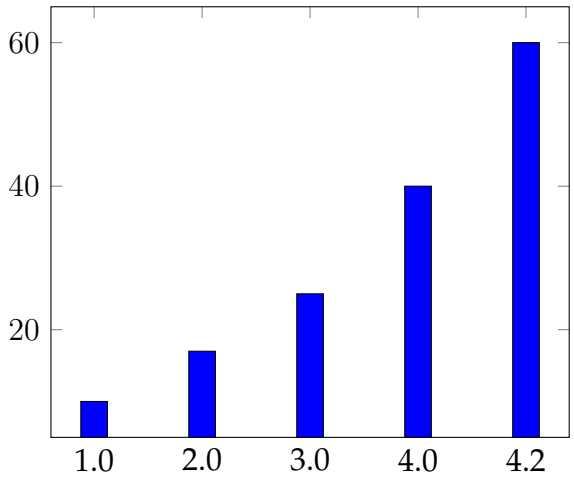

Figura 1.1: Scor mediu per versiune

Inițial, timpul de evoluție pentru a atinge un scor aproximativ de 10 puncte era de 15-20 de minute, plecând de la o generație creată aleator. În ultima versiune (anume `4.2`), acest timp a fost diminuat cu aproximativ 50%, existând cazuri excepționale care au avut o generare foarte bună și au putut oferi un scor mare încă din primele rulări.

1.4 Cazuri excepționale

Așa cum este specificat anterior, având în discuție un algoritm nedeterminist, nu se poate garanta o bună evoluție în mod constant, existând situații excepționale, atât în partea pozitivă (anume indivizi cu o perfomanță deosebită) cât și în partea negativă (indivizi care au un punct de start slab și rămân blocați într-un *local minimum*).

Pentru exemplu, pot exista cazuri pozitive de indivizi care au o evoluție foarte mare (a se vedea Fig. 1.2). Aceștia, de multe ori, sunt un caz special într-o populație oarecum standard, dar fiind un punct de plecare potrivit pentru evoluția unei generații ce pleacă de la un individ de bază.

Figura 1.2: Evoluție excepțională

Pe de altă parte, pot exista cazuri precum Fig.1.3, unde populația nu reușește să evolueze aproape deloc.

Figura 1.3: Evoluție slabă

Capitolul 2

Implementarea jocului

Această secțiune tratează procesul de dezvolatare a jocului de bază, plecând de la împlementarea originală (care presupune un jucător uman) la cea automatizată. Jocul este dezvoltat în Unity[6] (versiunea exactă `2019.4.11.f1`), un game engine popular în piața globală, folosind C# ca limbaj de programare.

Pe scurt, jocul implică acumularea unui scor cât mai mare prin evitarea unor obstacole numite *pipe*-uri, generate automat între limite (limita maximă superioară și limita minimă inferioară). Jucătorul trebuie să apese tasta **Space** pentru a sări, altfel va cădea constant. La final, scorul acumulat reprezintă numărul de pipe-uri traversate cu succes.

La nivel de implementare, jocul este alcătuit din mai multe obiecte independente:

- `PlayerController`, folosit în tot ce ține de un jucător/individ aparținând populației, anume incrementarea scorului, aplicarea mișcării, detectarea coliziunilor etc.;
- `PipeController`, folosit la controlarea mișcarii unui pipe și la distrugerea obiectului în momentul în care acesta iese din aria vizuală;
- `GuiManager`, responsabil cu afișarea informațiilor relevante pentru utilizator (scor, număr de generații, număr de indivizi rămași etc.);
- `GlobalManager`, care se ocupă de transmiterea infomațiilor de la o scenă la alta prin stocarea lor într-un format aparte.

Totodată, componentele enumerate anterior sunt legate de un așa-numit `GameManager`, o clasa specială care este abordată în formă de Singleton, pentru o bună etică a codului și a aplicației în sine.

Pentru mai multe detalii, aplicația dispune de o documentație în detaliu, disponibilă în bibliografie[7].

2.1 Generarea automată a nivelelor

Fiecare rulare a jocului este unică, deoarece metoda de generare a unui obstacol (numit, de altfel, și *pipe*), se bazează pe două limite: una superioară și una inferioară. Astfel, folosind aceste limite pe axa Y, un obstacol este garantat să se instanțieze corect (fără să fie în afara ecranului sau să nu permită depășirea lui).

În primele versiuni ale jocului, două valori consecutive din intervalul $[value^Y_{min}, value^Y_{max}]$ difereau prin valoarea minimă a unui număr de tip `float` din C#, iar acest aspect făcea ca jocul să instanțieze valori diferite, însă nu tocmai aleatoare. Aceasta rezulta într-o secvență de obstacole continuă foarte asemănatoare (de exemplu, 3 obstacole erau instanțiate la nivelul Y 1.05, 1.17, 0.89 - valori diferite, dar apropiate), iar pentru a construi un nivel care să pară aleator pentru un utilizator obișnuit, a fost nevoie de o revizuire a acestui algoritm.

Astfel, în ultima versiune generarea se face alegând (tot în mod aleator) o valoare dintr-o mulțime predefinită de valori cu o diferență semnificativă între ele.

2.2 Tipuri de scor

Pentru a putea aduce în discuție ideea folosirii unui algoritm genetic, este necesar ca fiecare individ dintr-o populație să poată fi evaluat pe baza unui *fitness*, calculat în funcție de scorul său. Pe scurt, o funcție fitness reprezintă o metodă de a evalua cât de bun este un individ dintr-o populație, sintetizând (de obicei) această informație într-un număr real.

În cazul de față, fiecare individ dispune de două scoruri diferite:

- **scorul de tip I** reprezintă numărul de obstacole depășite cu succes. Acesta este folosit pentru a arata utilizatorului progresul și pentru a fi ușor de înteles. Pentru ultima versiune a aplicației, un scor sub 20 poate fi considerat slab, unul între 20-50 poate fi considerat mediu, iar peste 50 poate fi considerat mare;

- **scorul de tip II** este folosit în procesul de selecție și reprezintă o valoare care crește constant atât timp cât individul nu a atins un obstacol. Astfel, se pune

accent pe cât de mult a rezitat un individ în aer, evitând situații precum Fig.2.1 în care individul albastru ar fi evaluat ca fiind la fel de bun ca individul verde (chiar dacă cel din urmă a parcurs o distanță mai mare).

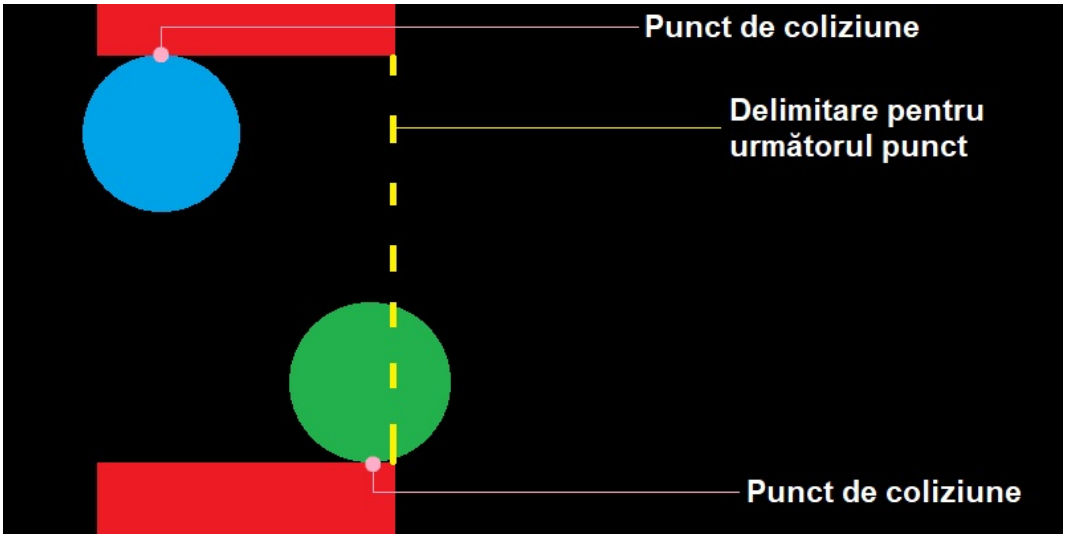

Figura 2.1: Indivizi cu același scor de tip I, dar diferiți pentru scor de tip II

2.3 Idei de îmbunătățire pe viitor

Pornind de la versiunea actuală, consider că există o gamă de îmbunătățiri ce pot fi aduse atât jocului în sine, cât și prezentării acestuia. Ca exemple, există:

- adăugarea posibilității de a juca jocul independent de interfața de simulare a evoluției. Mai exact, să existe posibilitatea ca un utilizator normal să experimenteze jocul de sine stătător, încercând să observe cât de departe poate ajunge;
- construirea unui *individ fantomă* ce va completa experiența descrisă mai sus, utilizatorul putându-și compara performanța cu cea a unui jucător antrenat de o rețea neuronală artificială;
- portarea aplicației pe mai multe platforme, pentru a putea fi distribuită în cât mai multe medii de testare;
- adăugarea unei teme specifice și realizarea unor texturi detaliate, pentru o mai bună experiență pe plan de UI/UX;

- adăugarea mai multor medii de antrenare, pentru diverse jocuri asemănătoare în complexitate (de exemplu, *World's Hardest Game*[8], *Crossy Road*[9], *Pacman*[10] etc.);
- adăugarea unor moduri de dificultate care să vizeze distanța variabilă între obstacole, dimensiunea acestora, viteza jocului sau alte aspecte ce pot face simularea mai ușoară sau mai dificilă.

Capitolul 3

Implementarea antrenării

Probabil cel mai dificil (atât din punct de vedere al implementării, cât și din punct de vedere computațional) este procesul de antrenare. Acesta face diferența între un joc obișnuit care necesită input din partea unei persoane și o aplicație care dispune de resursele necesare pentru a se auto-învăța. În cele ce urmează, vor fi detaliate clasele, metodele și parametrii folosiți în acest proces, detaliând motivația de alegere și felul în care se leagă aceste componente.

Pentru o mai bună înțelegere a metodei de utilizare în cod pentru oricare clasă descrisă mai jos, documentația[7] oferă o descriere amplă, în format HTML, folositoare în ideea dezvoltării viitoare.

3.1 Clasa Matrix

```
public class Matrix
{
    public Matrix(int rows, int columns);
    public Matrix(int rows, int columns, float min, float max);
    public Matrix(Matrix m);

    public int getRows();
    public int getColumns();

    public Matrix T();
```

```
    public float at(int row, int column);
    public void set(int row, int column, float value);

    public static Matrix operator+(Matrix m1, Matrix m2);
    public static Matrix operator*(Matrix m1, Matrix m2);

    override public string ToString();
}
```

Clasa `Matrix` reprezintă clasa de bază pentru orice calcul matriceal întâlnit pe parcursul programului. După cum reiese și din codul de mai sus, aceasta dispune de mai multe metode:

- `public Matrix(int rows, int columns)` : construiește un obiect având `rows` linii și `columns` coloane, fără a adăuga valori;
- `public Matrix(int rows, int columns, float min, float max)` : construiește un obiect având `rows` linii și `columns` coloane, inițializând fiecare valoare cu una aleatorie din intervalul $[min, max]$;
- `public Matrix(Matrix m)` : construiește un **deep copy** al unui obiect `Matrix` dat ca parametru;
- `public int getRows()` : returnează numărul de linii pentru obiectul curent;
- `public int getColumns()` : returnează numărul de coloane pentru obiectul curent;
- `public Matrix T()` : construiește și returnează matricea transpusă (sub formă de obiect `Matrix`) asociată obiectului curent;
- `public float at(int row, int column)` : funcție membră folosită în accesarea valorilor din interiorul matricei. Generează erori în cazul în care accesul se face pe valori incorecte;
- `public void set(int row, int column, float value)` : setează o valoare dată pe o poziție specificată. Generează erori în cazul în care accesul se face pe valori incorecte;

- `public static Matrix operator+(Matrix m1, Matrix m2)` : funcție ce implementează adunarea a două obiecte de tip `Matrix`;
- `public static Matrix operator*(Matrix m1, Matrix m2)` : funcție ce implementează înmulțirea a două obiecte de tip `Matrix`;
- `override public string ToString()` : funcție folosită pentru a genera și returna o reprezentare citibilă a informației din matrice.

3.2 Clasa NeuralNetwork

Indivizii implicați în antrenare folosesc o rețea neuronală artificială pentru luarea deciziilor în cadrul jocului. Pe scurt, aceștia observă mediul înconjurător (anume altitudinea, distanța până la următorul obstacol, velocitatea curentă, pozițiile colțurilor pentru următorul obstacol - a se vedea Fig.3.1) și decid pe baza acestora ce acțiune vor face (vor "sări" sau nu).

Structura rețelei este una simplă, cu 3 straturi (altfel numite și `layers`): unul de intrare, unul ascuns (altfel numit și `hidden layer`) și unul de ieșire, responsabil de decizia finală. Fiecare strat are o așa-numită *funcție de activare* (în engleză `activation function`), care, în urma unor calcule matematice complexe, decide dacă acel neuron *se va activa* sau nu. Aspectele detaliate legate de numărul exact de neuroni, de diferitele versiuni ale rețelei, de funcțiile de activare folosite vor fi tratate mai târziu în lucrare, anume în secțiunea 4.2.

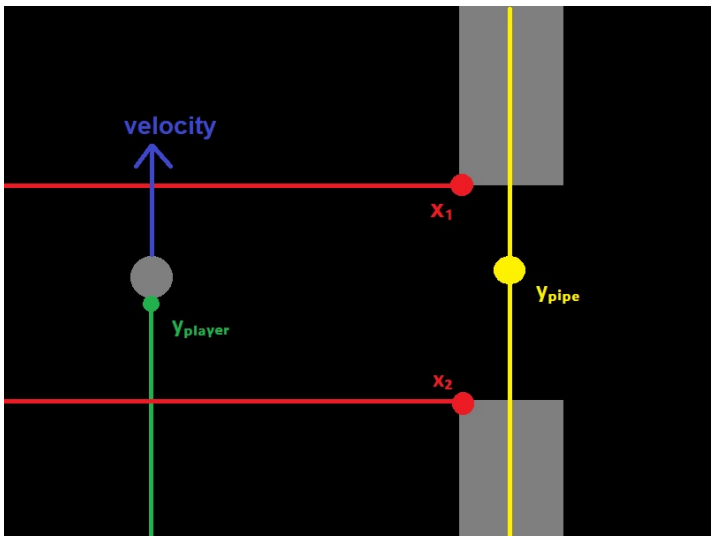

Figura 3.1: Informațiile adunate de un individ

```
public class NeuralNetwork
{
    public NeuralNetwork(int[] layers);
    public NeuralNetwork(NeuralNetwork n);
    public NeuralNetwork(string filePath);

    public int Guess(Matrix inputs);

    private void SigmoidActivation(Matrix m1);
    private void SoftmaxActivation(Matrix m1);

    public void Mutate(float chance);

    public void Export(string path);

    public override string ToString();

    static public void Crossover(NeuralNetwork n1, NeuralNetwork n2);
}
```

O scurtă descriere a metodelor prezentate anterior se află în paragraful următor:

- `public NeuralNetwork(int[] layers)` : construiește un obiect de tip `NeuralNetwo` folosind un vector dat, luând fiecare valoare din acesta ca fiind numărul de neuroni de pe stratul respectiv. În altă ordine de idei, $layers[i]$ reprezintă numărul de neuroni de pe stratul i (considerând stratul 0 ca fiind cel de intrare);
- `public NeuralNetwork(NeuralNetwork n)` : construiește un obiect de tip `NeuralNetwork` prin metoda **deep copy**;
- `public NeuralNetwork(string filePath)` : metodă ce permite construirea unei rețele neuronale pornind de la una exportată anterior către un fișier țintă (folosită de altfel în procedeul de utilizare a unui individ deja antrenat);
- `public int Guess(Matrix inputs)` : funcția de bază ce implementează algoritmul de **feed-forward**. Fiind optimizată pentru jocul actual, aceasta returnează direct o valoarea de `true/false` ce reprezintă decizia de a sări sau nu;

- `private void SigmoidActivation(Matrix m1)` : funcția de activare de tip `sigmoid`. Aceasta va fi discutată în detaliu în secțiunea 4.2;
- `private void SoftmaxActivation(Matrix m1)` : funcția de activare de tip `softmax`. Aceasta va fi discutată în detaliu în secțiunea 4.2;
- `public void Mutate(float chance)` : metodă implementată pentru a facilita operatorul de mutare (în eng. *mutation operator*). Folosind o anumită șansă, modifică valorile *weight*-urilor (metoda va fi explicată în detaliu în secțiunea 4.2);
- `public void Export(string path)` : metodă ce se ocupă cu exportarea informației din obiectul curent către un fișier (în momentul redactării, fișierul are mereu denumirea de **bestBird.txt**). Formatul unui astfel de fișier este descris atât în documentație[7], cât și în Fig.3.2;
- `public override string ToString()` : metodă folosită pentru a descrie pe scurt informația dintr-un obiect de tip `NeuralNetwork`;
- `static public void Crossover(NeuralNetwork n1, NeuralNetwork n2)` : metodă implementată pentru a facilita operatorul de *crossover* din cadrul algoritmului genetic. Implementarea acestuia va fi discutată în secțiunea 4.3.

Figura 3.2: Structura unui obiect `NeuralNetwork` exportat într-un fișier

Pentru asigurarea unui cod corect, funcțiile descrise anterior conțin metode de generare a erorilor în cazuri speciale (precum parametrii incorecți, citiri greșite, fișiere inexistente etc.), acestea fiind o necesitate atât în procesul de dezvoltare al aplicației, dar și în procesul de menținere și îmbunătățire a acesteia.

3.3 Clasa GeneticAlgorithm

Conceptul de NeuroEvoluție se bazează pe un algoritm genetic care folosește o populație de rețele neuronale artificiale pentru a maximiza scorul dat de o funcție fitness. În cazul de față, clasa `GeneticAlgorithm` este principalul actor în acest proces, după cum se poate observa mai jos:

```
public class GeneticAlgorithm
{
    static private GameObject[]
                    GetBestK(GameObject[] currentGeneration);

    static private float Fitness(int score);

    static private void Selection();
    static private void Mutation();
    static private void Crossover();

    static public GameObject[]
                    GetNextGeneration(GameObject[] currentGeneration);
}
```

O scurtă descriere a funcțiilor prezente (descriere ce va fi detaliată în secțiunea 4.3):

- `static private GameObject[] GetBestK(GameObject[] currentGeneration` : construiește din populația dată ca parametru una nouă pe care a fost aplicat conceput de *elitism*;
- `static private float Fitness(int score)` : funcție ce determină cât de bun este un individ, bazându-se pe scorul său;
- `static private void Selection()` : metodă ce aplică operatorul genetic de selecție (prin turneu) pe populația curentă;
- `static private void Mutation()` : metodă ce aplică operatorul de mutație pe populația curentă;

- `static private void Crossover()` : metodă ce aplică operatorul de crossover pe populația curentă;
- `static public GameObject[] GetNextGeneration(GameObject[] currentGe` : singura metodă publică prin care alte scripturi pot interacționa cu această clasă. Apelează, pe rând, fiecare operator genetic, returnând în final noua generație.

3.4 Parametrii de input/output

Așa cum este discutat și în Fig.3.1, actuala versiune folosește drept input-uri următoarele caracteristici:

- velocitate individului;
- poziția individului pe axa Y;
- poziția celui mai apropiat obstacol pe axa Y;
- poziția colțului drept inferior al părții superioare pe axa X;
- poziția colțului drept superior al părții inferioare pe axa X.

Totuși, acești parametrii nu reprezintă singurele abordări pentru un astfel de algoritm. Inițial, rețeaua neuronală primea alte informații (de exemplu, distanța de la individ la cel mai apropiat obstacol, dacă acesta sare sau cade, etc.) însă acestea nu ofereau rezultate bune în timpi rezonabili. Pentru a o îmbunătăți, era necesară o imbunătățire, fie prin reevaluarea informației importante dată de mediu (metodă aleasă în final), fie prin selectarea intregului ecran și convertirea sa într-un șir de pixeli. Această din urmă metodă este folosită cu preponderență în rețelele neuronale convulționale (în engleză CNN[11]).

Ideea de a folosi o simplă captură de ecran față de informații exacte alese strategic pare a fi o abordare lipsită de probleme și dificultăți, cel puțin la prima vedere. Însă, pe parcurs, apar neclarități ce necesită o rezolvare delicată:

- odată făcută captura de ecran, cum trebuie păstrată aceasta? Color sau în nuațe de gri?
- dacă rezoluția jocului este foarte mare, atunci captura va ocupa o cantitate mai mare de memorie. În această situație, este benefic folosirea procesului de *downscaling*[12]?

- în luarea unei decizii, pentru un individ contează doar următorul obstacol. Așadar, ce parte a ecranului trebuie capturată?
- deoarece o captură de ecran reprezintă un volum mare de informație, cât de des ar trebui făcută aceasta?

Referitor la partea de output, rețeaua returnează un șir cu 2 valori normalizate (anume că suma lor trebuie să fie 1): prima reprezintă un procent pentru decizia de a sări, iar a doua reprezintă opusul. Decizia se ia prin compararea celor două valori.

Acestea fiind spuse, următorul capitol va trata dificultățile de implementare, atât la nivel de cod cât și la nivel structural, urmărind totodată și cum anume au fost acestea rezolvate.

Capitolul 4

Dificultăți întâmpinate în dezvoltarea aplicației

> We try to solve the problem by rushing through the design process so that enough time is left at the end of the project to uncover the errors that were made because we rushed through the design process.

Citatul de mai sus aparține lui Glenford Myers și reprezintă destul de precis calea pe care o au majoritatea proiectelor din domeniul IT, punând în evidență punctul slab pe care îl împărtășesc aceastea: lipsa unui design clar, lipsit de ambiguități sau situații incerte.

Lucrarea de față a avut la rândul ei o mulțime de astfel de provocări, fie ele ce țin de design-ul aplicației în sine (cum anume interacționează utilizatorii cu aplicația, ce parametrii să fie configurabili, cum anume sunt gestionate erorile etc.), fie din cauza limitărilor de sistem (număr maxim de indivizi, viteza antrenamentului) sau fie datorită algoritmului de evoluție (aplatizarea rezultatelor cu cât mai mult este antrenată o generație). În cele ce urmează, vor fi tratate cele mai importante provocări întâlnite și modalitatea prin care acestea au fost rezolvate.

4.1 Lipsa unui set de date de antrenament

Algoritmul folosit a fost ales atât din punct de vedere al interesului personal, cât și al faptului că o abordare cu antrenare supervizată (*supervised learning* în engleză) nu este posibilă. Astfel, încercarea implementării unei rețele neuronale ce folosește **feed-forward** și **back-propagation** nu este posibilă fără un set de date destul de mare.

Pentru exemplu, proiectul de bază pentru acest algoritm este construirea unei rețele neuronale artificiale care recunoaște cifre scrise de mână. Dat fiind setul de antrenament al celor de la MNIST[13], o implementare simplă are nevoie de aproximativ 100.000 de imagini pentru o rată de eroare suportabilă (aproximativ 10%). Pentru a atinge asemenea rezultate în cazul jocului de Flappy Bird, ar fi necesare mult prea multe imagini cu diverse scenarii (diferite poziții pentru individ, obstacole de diferite dimensiuni, poziționări diferite, mutări ale imaginilor etc.), scenarii care ar trebui în final categorizate în două secțiuni: cele care implică efectuarea unui salt și cele care nu.

Acestea fiind spuse, o întrebare apare: de ce nu a fost folosit algoritmul de învățare nesupervizată, prin memorare (*reinforcement learning* în engleză), iar răspunsul ține de motivația personală și de dorința de a implementa ceva aparte, diferit și interesant în același timp.

4.2 Configurarea rețelei neuronale

4.2.1 Structura

Odată cu terminarea aplicației, un pas foarte important (care, de altfel, poate face diferența între o evoluție bună și una slabă) îl reprezintă configurarea rețelei neuronale. Figura 4.1 reprezintă structura standard a rețelei neuronale, cu cinci input-uri și două output-uri (discutate în secțiunea 3.4). Întrebarea care răsare este următoarea: cum anume sunt stabilite numărul de straturi ascunse (*hidden layers* în engleză)?

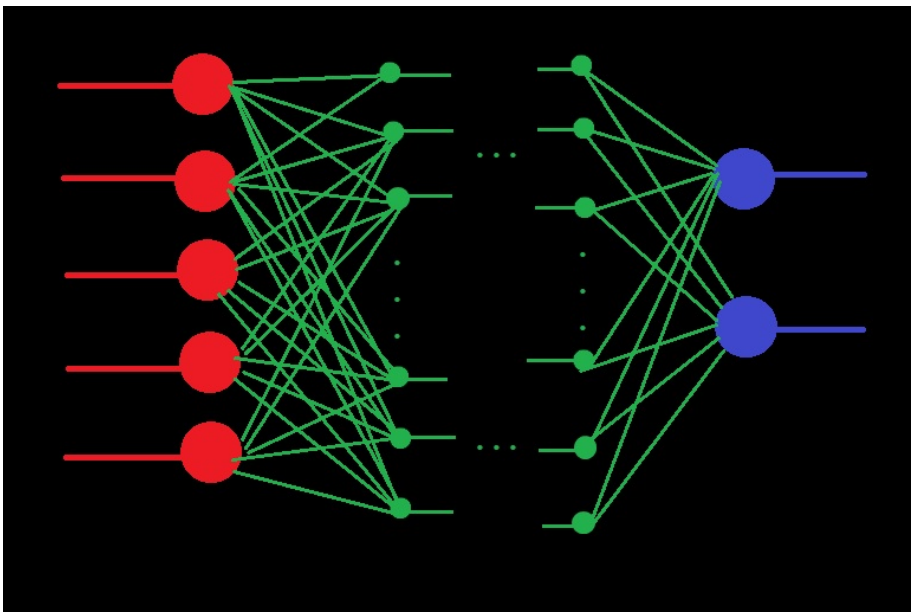

Figura 4.1: Structura rețelei neuronale

În această situație, există două soluții posibile:

- un număr mare de straturi, dar cu puține noduri pe fiecare strat;
- un număr mic de straturi, dar cu multe noduri pe fiecare strat.

Fiecare din aceste implementări vine cu avantaje și dezavajtaje, discutate mai jos.

Implementarea folosind multe straturi ascunse dar cu număr redus de noduri reprezintă abordarea clasica la capitolul rețele neuronale adânci (*Deep Neural Networks* în engleză), unde este necesar ca rețeaua să ”înteleagă” anumite trăsături ale informației primite pentru a o putea clasifica. Spre exemplu, un DNN poate fi folosit pentru antrenarea unei rețele care recunoaște pisici dintr-un set de imagini (fiecare strat va ”învăța” anumite trăsături: o pisică are capul rotund, are mustăți, are diverse culori, are ochii mari etc.).

Pe de altă parte, rețelele cu numar mic de straturi dar multe noduri pe fiecare strat sunt excelente în problemele de clasificare binară (altfel numite ”probleme de decizie”). În această situație, informația primită nu trebuie împărțită pe diferite categorii. În schimb, pe baza acesteia, trebuie oferit un răspuns binar (de cele mai multe ori **da** sau **nu**), iar acest aspect o face perfectă pentru lucrarea de față. În ultima versiune, aplicația dispune de o configurație de forma 5-40-2 (5 noduri de input, 40 de noduri ascunse, 2 noduri de output).

4.2.2 Funcții de activare

Odată stabilită structura rețelei, un alt pas important este stabilirea funcțiilor de activare. Acestea sunt funcții care analizează valoarea numerică primită de un nod și decid dacă acesta se "activează" (*fires* în engleză) sau nu. Mai jos sunt descrise pe scurt cele mai importante funcții de activare și unde sunt aceastea întâlnite:

- **Funcția de activare perceptron**

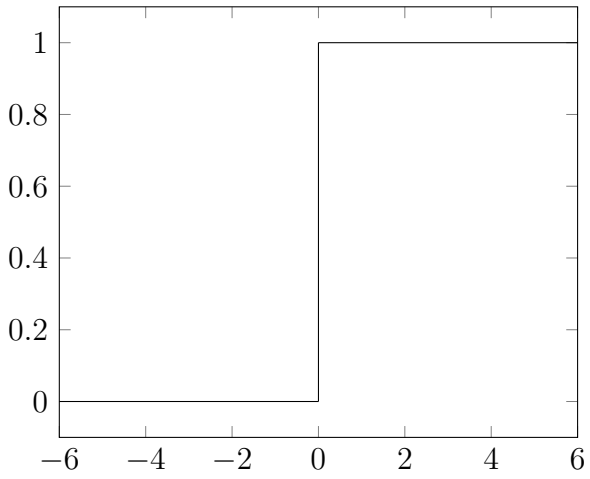

Figura 4.2: Funcția de activare perceptron

Reprezintă cea mai simplă funcție de activare, dată de formula

$$f(x) = \begin{cases} 0, & \text{dacă } x \leq 0 \\ 1, & \text{altfel.} \end{cases}$$

Este folosită în implementarea de bază a perceptronilor datorită simplității acesteia și a vitezei mari de calcul. Totuși, aceasta ignoră valorile negative primite ca date de intrare, ceea ce o face de evitat în situații complexe.

- **Funcția de activare sigmoidă**

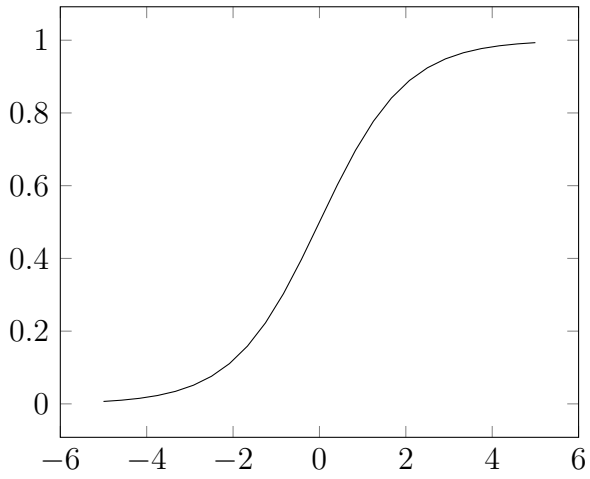

Figura 4.3: Funcția de activare sigmoidă

Dată de formula

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

aceasta reprezintă funcția de bază în domeniul rețelelor neuronale artificiale, luând în calcul atât valori pozitive cât și valori negative ca date de intrare, dar invalidând acele valori exagerate (care se îndepărtează mult de valoarea zero). Astfel, pentru rezultate corecte, este necesar ca valorile de intrare să fie normalizate înainte. Aplicația folosește acest tip de funcție pentru activarea nodurilor din stratul ascuns.

- **Funcția de activare arctangentă**

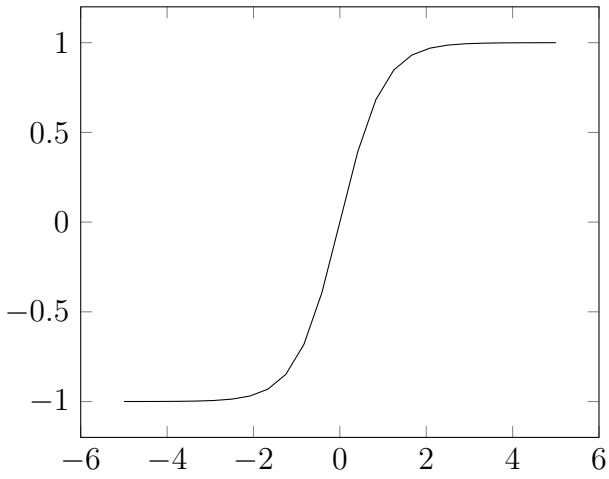

Figura 4.4: Funcția de activare arctangentă

Foarte asemănătoare cu funcția sigmoidă, această funcție are formula

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

singura diferență notabilă fiind codomeniul acesteia $(-1, 1)$ în loc de $(0, 1)$. Această diferență face ca funcția arctangentă să poată activa negativ un nod, ducând la rezultate impresionante.

- **Funcția de activare ReLU**

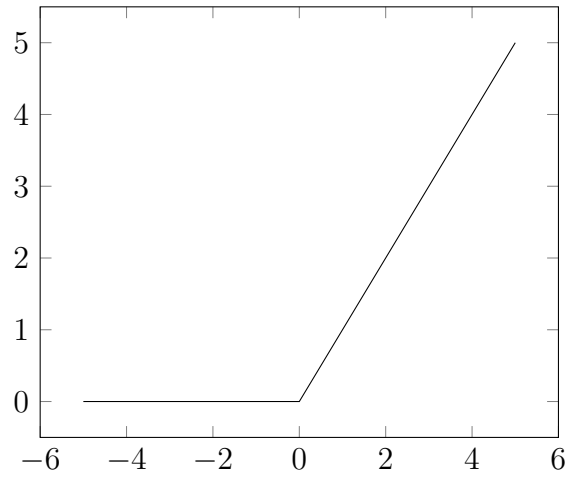

Figura 4.5: Funcția de activare ReLU

Aceasta este o funcție mai complexă, fiind în esență o combinație între o funcție liniară și funcția perceptron. Formula acesteia este

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

ceea ce o face o alegere foarte bună pentru rețele neuronale convulționale (CNN) sau rețele neuronale adânci (DNN).

- **Funcția de activare softmax**

Aceasta din urmă reprezintă un caz special, fiind aplicată pe o mulțime de valori în loc de o singură valoare. Formula funcției este

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}, \text{ pentru } i = 1, ..., K$$

urmărind normalizarea șirului de valori. Această funcție este folosită pe ultimul strat pentru a obține rezultate ușor de înteles (cu valori între 0 și 1, asemănătoare unor procente), după cum este cazul și în aplicația de față.

4.3 Configurarea algoritmului genetic

Dacă rețeaua neuronală este responsabilă cu luarea deciziilor (în cazul de față fiind doar una, anume ”a sări sau nu”), algoritmul genetic se ocupă cu optimizarea populației. În alte cuvinte, prin generații repetate, se încearcă depășirea dificultăților și creșterea continuă în eficiența indivizilor din generația următoare.

La nivel de bază, un algoritm genetic conține o serie de operatori ce sunt aplicați pentru a simula diversitatea întâlnită în natură și pentru a imita cât mai bine procesul complex de evoluție. Operatorii de bază sunt:

- selecția (*selection* în engleză) : se ocupă cu selectarea celor mai buni indivizi din populația curentă, clasamentul fiind dat de funcția fitness aleasă;
- mutația (*mutation* în engleză) : se ocupă cu adăugarea de mutații aleatoare în informația unui individ. De multe ori aceste mutații pot duce la indivizi excepționali sau la indivizi foarte slabi, așadar este necesar ca rata de mutație să fie aleasă corect;
- încrucișarea (*crossover* în engleză) : se ocupă cu ”înmulțirea” a doi indivizi pentru a crea alți doi în generația următoare (pe scurt, doi indivizi vor fi ”părinți” iar prin ”încrucișarea” acestora se vor obține doi ”copii”).

În cele ce urmează, vor fi prezentate operatorii folosiți în aplicația de față, împreună cu alte idei pentru implementări ulterioare sau probleme ce au apărut în versiuni anterioare.

4.3.1 Operatorul de selecție

Așa cum este menționat anterior, selecția se ocupă cu alocarea unei șanse fiecărui individ din populație, șansă pe baza căruia va fi sau nu introdus în generația următoare. Scopul acestui operator este de a promova acei indivizi care au un fitness ridicat, în speranța că viitoarea generație va avea un rezultat și mai bun. În acest sens, pot fi două tipuri de selecții: cele cu o presiune ridicată (care selectează mulți indivizi foarte adaptați) și cele cu o presiune scăzută. Prima metodă duce la o evoluție rapidă, dar și la o aplatizare a diversității - mulți indivizi vor fi foarte asemănători. Pe de altă parte, a doua metodă permite o diversitate mai mare, însă scade în același timp viteza de

evoluție (va fi necesar un timp mai mare de evoluție pentru a se ajunge la un individ bun).

Există mai multe metode de selecție:

- **roata norocului** : fiecare individ are un număr proporțional de urmași cu fitness-ul acestuia. Pe scurt, fiecărui individ îi este asociată o șansă pe baza careia este sau nu ales (poate fi ales de mai multe ori);
- **bazată pe rang** : fiecare individ este sortat pe baza fitness-ului său. Șansa de a fi selectat este egală cu ordinea sa în această listă (cel mai slab individ va avea rang 1, deci va fi ales cel mai rar - cel mai adaptat individ va avea rang N, unde N este numărul de indivizi din populație);
- **turneu** : se selectează aleator o mulțime X de indivizi ($|X| < |P|$), unde P este populația), după care se aleg cei mai buni $|Y|$ indivizi ($|Y| < |X|$) din mulțimea X. Acest proces se repetă până când se obține numărul dorit de indivizi (de cele mai multe ori $|P|$).

După selectarea metodei dorite, asupra operatorului de selecție mai poate fi adăugat un pas important care de multe ori duce la o sporire considerabilă a calității și eficienței evoluției. Prin utilizarea **elitismului**, care presupune selectarea din generația curentă a celor mai buni K indivizi și adăugarea acestora fără modificări în generația următoare, se garantează păstrarea anumitor calități si aptitudini cheie, care de altfel ar putea fi modificate sau alterate.

Observând pe parcursul dezvoltării aplicației o îmbunătățire considerabilă a procesului de evoluție atunci când elitismul este folosit, s-a ajuns ca acesta să fie întrodus definitiv (valoarea lui K se află în tabelul 4.1).

4.3.2 Operatorul de mutație

Mutația reprezintă modificarea unei gene alese aleatoriu din materialul genetic al unui individ, în cazul de față fiind valorile prezente în rețeaua neuronală (*weights* și *biases*). Pentru cazul de față, au fost încercate două implementări ale acestui operator:

- mutația totală : reprezintă alegerea aleatorie a unei valori din mulțimea *weight*-urilor și înlocuirea acesteia cu o valoare nouă cuprinsă într-un interval prestabilit (de obicei, $[-1, 1]$);

- mutație parțială : reprezintă alegerea aleatorie a unei valori din mulțimea *weight*-urilor și modificarea acesteia prin adunarea unei valori alese aleator din intervalul $[-0.1, 0.1]$.

În implementările inițiale, a fost folosită prima metodă pentru a testa eficiența acesteia, dar înlocuirea valorii cu una total aleatorie s-a dovedit a fi prea drastică pentru procesul de evoluție, ducând deseori la valori mai slabe sau chiar la o absență totală a evoluției. Pe de altă parte, ușoara modificare a valorii deja existente s-a dovedit a fi o alegere mai bună, ducând la rezultate concrete.

4.3.3 Operatorul de încrucișare

Poate cel mai ”realist” operator, *crossover*-ul se ocupă cu generarea de noi indivizi (numiți ”copii”) plecând de la o pereche de indivizi (numiți ”părinți”) din generația curentă. Acesta încrucișează informația genetică în speranța obținerii unor indivizi cu un fitness mai ridicat în generația următoare.

Pentru aplicația de față, sunt posibile 3 tipuri de încrucișări:

- **încrucișarea prin copiere totală**

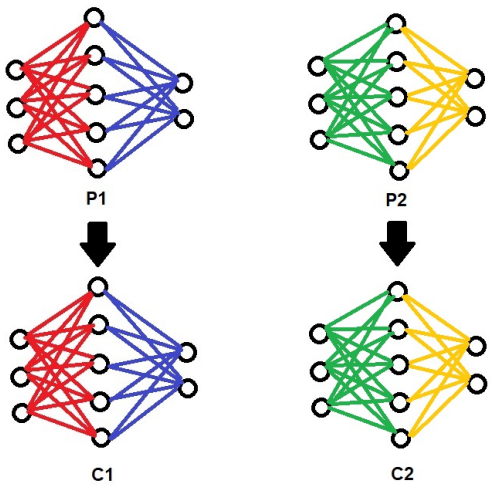

Figura 4.6: Încrucișarea prin copiere totală

Așa cum este prezentat si în figura 4.6, aceasta este cea mai simplă metodă de încrucișare: o simplă copiere între părinte și copil. Foarte eficientă din punct de vedere al timpului de execuție, dar nu aduce diversitate în următoarea generație.

- **încrucișarea prin copiere parțială**

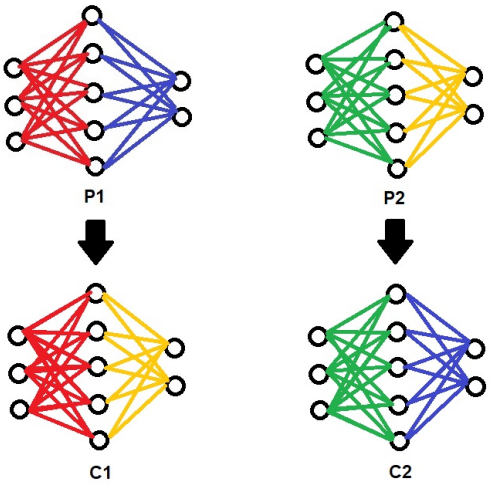

Figura 4.7: Încrucișarea prin copiere parțială

Această metodă construiește urmași luând pe rând informații din fiecare părinte, sporind astfel diversitatea dar păstrând totodată informația acumulată până în acel moment. În pseudocod, algortimul are următoarea structură (de altfel vizibilă și in figura 4.7):

```
function crossover(parent_1, parent_2)
    kid_1 = null
    kid_2 = null

    i = 0
    foreach info_gene in parent_1:
        if i % 2 == 0
            kid_1.add_gene(parent_1.info_gene[i])
        else
            kid_1.add_gene(parent_2.info_gene[i])
        i++

    i = 0
    foreach info_gene in parent_1:
        if i % 2 == 1
            kid_2.add_gene(parent_1.info_gene[i])
        else
```

```
            kid_2.add_gene(parent_2.info_gene[i])
        i++

    return kid_1, kid_2
```

De menționat este că membrul `info_gene` reprezintă informația genetică asociată, în cazul de față fiind un strat din rețeaua neuronală.

- **încrucișarea pe biți**

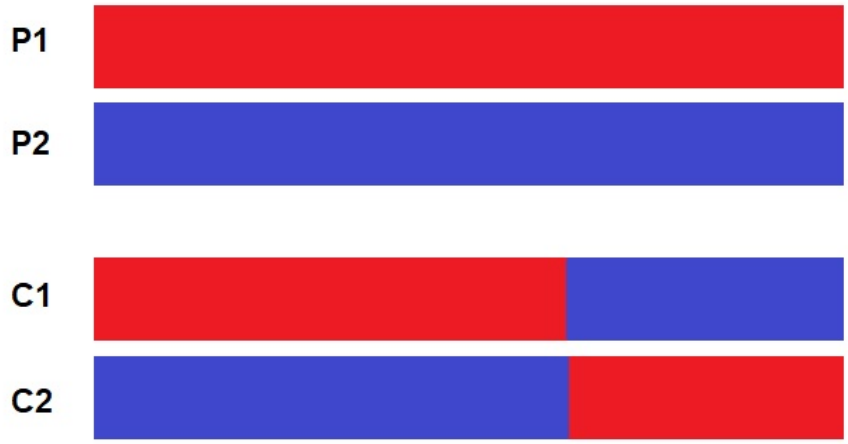

Figura 4.8: Încrucișarea pe biți

Metoda încrucișării pe biți are ca avantaj lipsa necesității de adaptare, indiferent de problemă. Singura provocare este de a găsi o reprezentare corectă (și folositoare) a informației de interes, în cazul de față fiind valorile *weight*-urilor și ale *bias*-urilor. Pentru a putea aplica această metodă, valorile menționate sunt inserate într-un vector (câte unul pentru fiecare părinte) după care se alege (aleator sau predefinit) un punct de tăiere unde se va efectua interschimbarea (există și posibilitatea alegerii unui număr mai mare de puncte de tăiere). Mai jos este oferit un exemplu pentru o mai bună înțelegere (se consideră punctul de tăiere ales aleator - pentru exemplu, valoarea va fi 2):

```
parent_1 = [11, 12, 13, 14, 15]
parent_2 = [21, 22, 23, 24, 25]

child_1 = [11, 12, 23, 24, 25]
```

```
child_2 = [21, 22, 13, 14, 15]
```

Un aspect important al acestui tip de *crossover* este că poate fi aplicat direct pe numere (caz în care este mai rapid) sau pe reprezentarea binară a acestora (dar fiind mai lent datorită conversiei dintr-un număr într-un vector reprezentând valoarea binară). Exemplul anterior evidențiază primul caz. Mai jos este exemplificată a doua metodă (se consideră ca un număr este reprezentat pe 4 biți, iar punctul de tăiere este 6):

```
parent_1 = [10, 2, 7, 15]
parent_2 = [1, 9, 3, 5]


p1_binary = "1010001001111111"
p2_binary = "0001100100110101"

k1_binary = "1010000100110101"
k2_binary = "0001101001111111"

kid_1 = [10, 1, 3, 5]
kid_2 = [1, 10, 7, 15]
```

Date fiind descrierile anterioare, ultima versiune a aplicației folosește următoarele valori pentru operatorii genetici:

Operator genetic	Valoare
Tipul de selecție	roata norocului combinat cu elitism
Rata de mutație	0.03
Rata de încrucișare	0.7
Indivizi selectați pentru elitism	10 (din toată populația)

Tabela 4.1: Operatori genetici folosiți

4.4 Speed-up

Dat fiind faptul că un proces de antrenare, chiar și pentru probleme simple, durează destul de mult timp, implementarea inițială a aplicației avea o setare ce permitea utilizatorului setarea vitezei de învățare: pentru a observa procesul și diferențele de la o generație la alta, aceasta putea fi setată pe minim, iar pentru rezultate rapide, putea fi setată pe maxim.

Problema care a apărut ulterior ține de modul în care Unity (mediul de dezvoltare folosit) implementează conceptul de **timp**. Cu cât acesta trece mai repede, cu atât anumite calcule sunt omise, iar desincronizările între frame-uri nu întârzie să apară, ducând la multe erori ale jocului (indivizi care nu apar, care trec prin obstacole, obstacole suprapuse etc.) și la o experiență neplăcută pentru utilizator.

O primă rezolvare a constat în eliminarea nevoii de a desena partea grafică a procesului de evoluție, simulând acțiunile indivizilor și antrenamentul în sine, dar afișând doar rezultatul final. Dar această rezolvare nu a putut fi implementată, deoarece obiectele interacționează folosind fizică ce trebuie recalculată în fiecare frame pentru a actualiza poziția, velocitatea sau rotația obiectelor.

În final, această idee a fost abandonată, neavând o rezolvare benefică utilizatorului, urmând să fie revizuită în versiunile următoare.

4.5 Absența unei librării standardizate

Așa cum este menționat și în introducere, una din țintele acestui proiect a fost realizarea antrenamentului fără utilizarea unui tool extern sau dezvoltat de altcineva. Prin această decizie, o serie de probleme (precum cele menționate mai sus) au apărut, făcându-mă să regândesc anumite aspecte și să găsesc soluții pentru cele mai complicate. În final însă, nimic nu e perfect și mereu e loc de mai bine, iar din această cauză se regăsesc mai jos câteva optimizări posibile ce vor fi implementate pe viitor:

- optimizarea calculului matriceal prin diverse metode (de exemplu, *multi-threading*);
- reducerea spațiului de stocare necesar pentru exportul informației (o mai bună structură a fișierului exportat);
- simularea mediului de antrenare prin calcule matematice simple față de utilizarea unui mediu complex ce utilizează interacțiuni continue între obiecte;

- studierea și adăugarea mai multor variații de operatori genetici;
- implementarea unui meta-GA (*Genetic Algorithm*, din engleză) pentru optimizarea parametrilor algoritmului genetic.

Concluzii

Orice proiect are două mari obstacole care apar pe parcursul acestuia: primul constă în concretizarea ideilor de bază prin prima linie de cod scrisă, linie care marchează începutul unui proces a cărei destinație poate fi ușor observată, dar pașii până în acel punct sunt deseori ascunși de incertitudine, provocări, neclarități și `Error on line X`. Cel de-al doilea obstacol este mai greu de perceput la început, dar devine din ce în ce mai clar cu cât proiectul avansează, și anume finalitatea acestuia. Orice proiect trebuie să tindă spre un scop, o finalitate precisă, iar eu consider că această aplicație a ajuns în punctul său respectiv.

Din această lucrare pot fi trase o multitudine de concluzii, dar, atât pentru simplitate cât și pentru păstrarea unui ton academic, le voi împărți în două categorii: concluzii personale și concluzii analitice.

Pe partea personală, consider că acest proiect m-a determinat să învăț mai mult, să descopăr mai mult și să înțeleg pe deplin cum anume funcționează atât un proces de self-learning, cât și un proces de dezvoltare al unei aplicații standalone. Am întâmpinat dificultăți care m-au forțat să cresc, să caut mai în detaliu și să revizuiesc ce făcusem deja, aspecte de care sunt mândru că le-am experimentat.

Pe partea analitică, consider că lucrarea de față demonstrează cât de eficient poate fi un algortim genetic în combinație cu rețelele neuronale artificiale și, totodată, aduce acest subiect la un nivel ușor de înțeles, deschizând astfel calea spre proiecte de o magnitudine mai mare pentru jocuri (sau situații asemănătoare) mai complexe. Pentru ultima versiune, tabelul de mai jos 4.2 prezintă rezultate a 7 rulări independente ale algoritmului:

Număr de generații	Scor maxim	Gen. în care a fost atins	Timp
46	287	45	30 min.
9	357	8	15 min.
27	305	26	15 min.
36	182	28	10 min.
150	10	102	15 min.
3	2531	2	2 ore
3	peste 23000	2	9 ore

Tabela 4.2: Rezultate la diferite rulări ale evoluției

După cum se poate observa, cea mai bună rulare este de aproximativ 9 ore (moment în care am decis să închid simularea) obținută prin reantrenarea unui individ aproape perfect în rularea 6. În cazul unui individ uman, aceste performanțe sunt de neatins, ceea ce demonstrează încă o dată performanța algortimului.

În final, consider că proiectul și-a atins scopul, dat fiind motivele aduse anterior în discuție, și, în încheiere, țin să mulțumesc domnului coordonator Croitoru Eugen, colegilor și prietenilor care m-au ajutat prin testarea aplicației, prin idei sau printr-o simplă încurajare. Vă mulțumesc pe această cale și în continuare voi lucra la această aplicație pentru a o inova constant, pentru a-i aduce îmbunătățiri și pentru a o face cât mai accesibilă.

Bibliografie

[1] Machine Learning in games

https://developer.ibm.com/articles/machine-learning-and-gaming/

[2] Flappy Bird

https://en.wikipedia.org/wiki/Flappy_Bird

[3] GitHub project page

https://github.com/adipeterca/bachelor-degree

[4] MarI/O project developed by SethBling

https://www.youtube.com/watch?v=qv6UVOQ0F44

[5] The Coding Train site

https://thecodingtrain.com/

[6] Unity site

https://unity.com/

[7] Documentația proiectului

https://github.com/adipeterca/bachelor-degree/documentation

[8] World's Hardest Game

https://theworldshardestgame.fandom.com/wiki/The_Worlds_Hardest_Game

[9] Crossy Road

https://en.wikipedia.org/wiki/Crossy_Road

[10] Pacman

https://en.wikipedia.org/wiki/Pac-Man

[11] Rețele neuronale convulționale

https://en.wikipedia.org/wiki/Convolutional_neural_network

[12] Procesul de *downscaling*

https://en.wikipedia.org/wiki/Downscaling

[13] MNIST handwritten digits database

http://yann.lecun.com/exdb/mnist/