UNIVERSITATEA “LUCIAN BLAGA” DIN SIBIU
FACULTATEA DE INGINERIE
DEPARTAMENTUL DE CALCULATOARE ŞI INGINERIE ELECTRICĂ

PROIECT DE DIPLOMĂ

Conducător științific : **Conf. dr. ing. Morariu Daniel**

Absolvent: **Jurju Alexandru**
Specializarea: Calculatoare

– Sibiu, 2023 –

UNIVERSITATEA “LUCIAN BLAGA” DIN SIBIU
FACULTATEA DE INGINERIE
DEPARTAMENTUL DE CALCULATOARE ŞI INGINERIE ELECTRICĂ

Algoritmi de inteligență artificială pentru a învăța jocul de Snake

Conducător științific**: Conf. dr. ing. Morariu Daniel**

Absolvent**: Jurju Alexandru**
Specializarea**:** Calculatoare

Cuprins

1 Introducere 6

1.1 Tema și scopul lucrării 6

1.2 Structura lucrării 7

2 Considerații Teoretice 8

2.1 Utilizarea AI în jocuri 8

2.2 Rețeaua Neuronală 8

2.3 Structura rețelei neuronale Feed-Forward 8

2.3.1 Structura unui neuron 10

2.3.2 Reprezentarea ponderilor și bias-urilor 11

2.3.3 Regula Backpropagation 12

2.3.4 Funcții de activare 13

2.3.4.1 Funcția Sigmoidă 14

2.3.4.2 Funcția Tangentă hiperbolică 14

2.3.4.3 Funcția ReLU 15

2.3.5 Pasul Forward 15

2.3.6 Metode de calcul al erorii Rețelei Neuronale 17

2.3.6.1 Mean Squared Error 18

2.3.7 Pasul Backward 18

2.4 Algoritmul genetic 22

2.4.1 Funcția de Fitness 23

2.4.2 Operatori de Selecție 24

2.4.2.1 Roulette Wheel Selection 24

2.4.2.2 Elitist Selection 25

2.4.3 Operatori de Crossover 26

2.4.3.1 One-point Crossover 26

2.4.3.2 Two-point Crossover 27

2.4.4 Operatori de mutație 28

2.4.4.1 Mutație Gausiană 29

2.4.5 Folosirea algoritmului genetic pentru antrenarea rețelei neuronale 29

2.5 Distanța Manhattan 30

3 Rezolvarea Temei de proiect 31

3.1 Structura Aplicației 33

3.2 Implementarea Jocului 35
3.2.1 Clasa Snake 35
3.2.2 Clasa Model 36
3.2.3 Direcțiile jocului 40
3.2.4 Liniile de viziune 41
3.3 Îmbunătățirea vitezei de execuție pentru program 45
3.4 Implementarea Rețelei Neuronale – Clasa Dense si Clasa Activation 46
3.4.1 Arhitectura rețelei neuronale 47
3.4.2 Implementarea funcțiilor de activare 48
3.4.3 Implementarea funcțiilor de eroare 48
3.4.4 Clasa de bază pentru straturile rețelei neuronale 48
3.4.5 Constructorii claselor Dense și Activation 49
3.4.6 Pasul Forward 49
3.4.7 Pasul Backward 50
3.4.8 Clasa NeuralNetwork 51
3.4.9 Implementarea algoritmului de antrenament Backpropagation 51
3.5 Implementarea Algoritmului Genetic 53
3.5.1 Reprezentarea unui cromozom 53
3.5.2 Funcția de fitness 54
3.5.3 Operatori de selecție 55
3.5.3.1 Roulette Wheel 55
3.5.3.2 Elitist 55
3.5.4 Operatori de crossover 56
3.5.4.1 One-Point Crossover 56
3.5.4.2 Two-Point Crossover 57
3.5.5 Operatori de mutație 57
3.5.5.1 Gaussian Mutation 57
3.5.6 Folosirea operatorilor genetici 58
3.6 Rularea programului 59
3.6.1 Clasa de bază a stărilor din program 59
3.6.2 Mașina cu stări finite 61
3.6.3 Structura comuna a unei stări 63
3.6.4 Starea Options 66

3.6.5 Antrenarea rețelei folosind algoritmul genetic 67
3.6.6 Evaluarea algoritmului genetic 72
3.6.7 Antrenarea rețelei folosind Backpropagation 73
3.6.8 Rularea unei rețele neuronale antrenate 76
3.6.9 Stocarea și citirea caracteristicilor rețelelor neuronale în/din fișiere 77
4 Concluzii 80
5 Referințe bibliografice 83

1 Introducere

1.1 Tema și scopul lucrării

Scopul acestei lucrări este de a utiliza diferiți algoritmi și metode de inteligență artificială pentru a crea o aplicație care să învețe să joace un joc simplu, obiectivul final fiind obținerea unui program care poate să ia singur decizii în funcție de starea curentă a jocului astfel încât să obțină un rezultat cât mai bun, poate chiar mai bun decât rezultatele care pot să fie obținute de jucătorii umani.

Jocul pe care se aplică algoritmii este jocul Snake. Acesta este un joc video clasic care a fost inventat la sfârșitul anilor 1970 și a câștigat popularitate în 1997 când a fost lansat preinstalat pe telefonul Nokia 6110, moment în care a devenit un succes instant din cauza simplicității și a naturii care genera dependență. Astăzi, Snake a rămas un joc popular care este disponibil pe o gama largă de platforme, inclusiv calculatoare, console de jocuri și în special pe telefoane.

Într-un joc tipic de Snake jucătorul controlează un „șarpe”, un set de pătrate învecinate care se mișcă în interiorul unei table și care încearcă să ajungă la mâncarea care apare aleatoriu în suprafața de joc. Pe măsură ce șarpele mănâncă, dimensiunea lui va crește, ceea ce face mai dificilă navigarea pe ecran. Jucătorul trebuie să evite ciocnirea cu pereții sau cu corpul șarpelui. Scopul jocului este de a menține șarpele în viață cât mai mult timp posibil și de a obține un scor cât mai mare. Jocul se termină în momentul când șarpele se ciocnește cu un perete sau cu un segment al corpului, și jocul este câștigat în momentul când șarpele devine așa de mare încât nu mai există loc în care să se pună mâncare pe tablă.

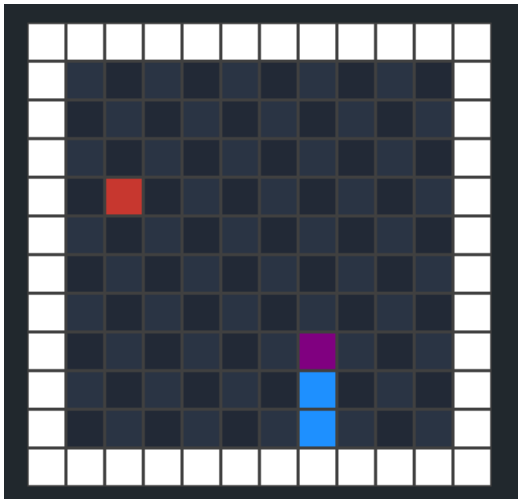

Figura 1.1 – Imagine cu rularea jocului Snake în cadrul proiectului

Pentru a realiza mișcarea șarpelui pe tablă s-a utilizat o rețea neuronală de tip Feed-Forward cu un singur nivel ascuns de neuroni care primește la intrarea rețelei o viziune redusă a tablei de joc. Viziunea redusă se referă la faptul că rețeaua neuronală nu primește la intrare toată tabla de joc, pentru obținerea intrărilor se vor folosi valorile obținute de „liniile de viziune", raze care pornesc de la primul segment al șarpelui în direcții diferite până când ating un perete și rețin distanțele între capul șarpelui și elementele de pe tablă (măr, segment al corpului și perete) care se află pe acea linie de viziune. Ieșirea rețelei neuronale va fi direcția în care șarpele trebuie să se miște.

Pentru antrenarea rețelei neuronale s-au implementat două metode separate de antrenament:

1. Antrenare automată folosind Algoritmii Genetici care nu necesită ajutorul utilizatorului pentru a realiza antrenarea
2. Antrenare supervizată folosind Regula Backpropagation care are nevoie pentru antrenarea rețelei de exemple de antrenament. Exemplele de antrenament vor fi obținute prin înregistrarea mișcărilor pe care le face un jucător uman în joc.

1.2 Structura lucrării

Capitolul 1 al proiectului reprezintă o introducere, oferind o scurtă descriere a proiectului și enumerând principalele obiective pe care acesta dorește să le abordeze.

Capitolul 2 cuprinde noțiunile teoretice necesare pentru implementarea programului, inclusiv formulele necesare, definițiile teoretice ale algoritmilor și funcțiilor care urmează să fie utilizate.

Capitolul 3 prezintă implementarea aplicației, descriind modul în care programul a fost realizat, prezintă structura acestuia și furnizează informații legate de modul de funcționare al aplicației.

Capitolul 4 prezintă rezultatele obținute, se compară obiectivele propuse la începutul proiectului cu cele realizate, și se formulează concluzii asupra acestor rezultate.

Capitolul 5 include referințe bibliografice utilizate pentru cercetare, precum și pentru implementarea aplicației.

2 Considerații Teoretice

2.1 Utilizarea AI în jocuri

Inteligența artificială (AI) reprezintă o ramură a informaticii care permite programelor să imite funcțiile cognitive umane, inclusiv percepția, învățarea, rezolvarea de probleme, permițându-le să realizeze sarcini care, de obicei, ar necesita inteligență umană. Inteligența Artificială a evoluat foarte mult în ultimii ani, s-a ajuns în stadiul în care anumite programe de inteligență artificială pot depăși performanțele umane, cum ar fi programul de șah, Stockfish, sau programul de Go, AlphaGo, care au devenit aproape imposibil de învins de către un jucător uman. Pe măsură ce inteligența artificială a devenit din ce în ce mai puternică, oamenii explorează modalități de a o utiliza în rezolvarea problemelor complexe dintr-o gamă variată de domenii, precum medicina, finanțele, robotică și multe altele. Această evoluție a tehnologiei a deschis noi oportunități, permițând oamenilor să abordeze provocări dificile cu ajutorul inteligenței artificiale.

2.2 Rețeaua Neuronală

Rețelele neuronale (NN), care se mai numesc și rețelele neuronale artificiale (ANN), sunt modele computaționale inspirate de structura și funcționarea creierului uman. O rețea neuronală este formată din mai multe noduri interconectate, sau neuroni, care lucrează împreună pentru a prelucra datele primite la intrarea rețelei pentru a genera valori la ieșire.

O rețea neuronală este formată din mai mulți neuroni aranjați pe straturi diferite. Un neuron în cadrul unei rețele neuronale artificiale este o componentă esențială care simulează funcționarea unui neuron biologic. Acești neuroni sunt interconectați într-o structură complexă, formând rețeaua neuronală.

2.3 Structura rețelei neuronale Feed-Forward

O rețea neuronală feed-forward este o rețea neuronală artificială în care conexiunile dintre noduri nu formează un ciclu și în care fluxul de informație se propagă doar într-o direcție, de la stratul de intrare al rețelei spre stratul de ieșire. Toți neuronii din stratul curent al rețelei vor fi conectați prin legături sinaptice doar cu neuronii din stratul următor al rețelei. Ieșirea unui neuron din stratul curent al rețelei va fi folosită ca intrare doar de către neuronii de pe stratul următor al rețelei până când se ajunge la ultimul strat.

Configurația unei rețele neuronale feed-forward este de forma: un strat de intrare, unul sau mai multe straturi ascunse și un strat de ieșire. Această configurație este prezentată în figura 2.1

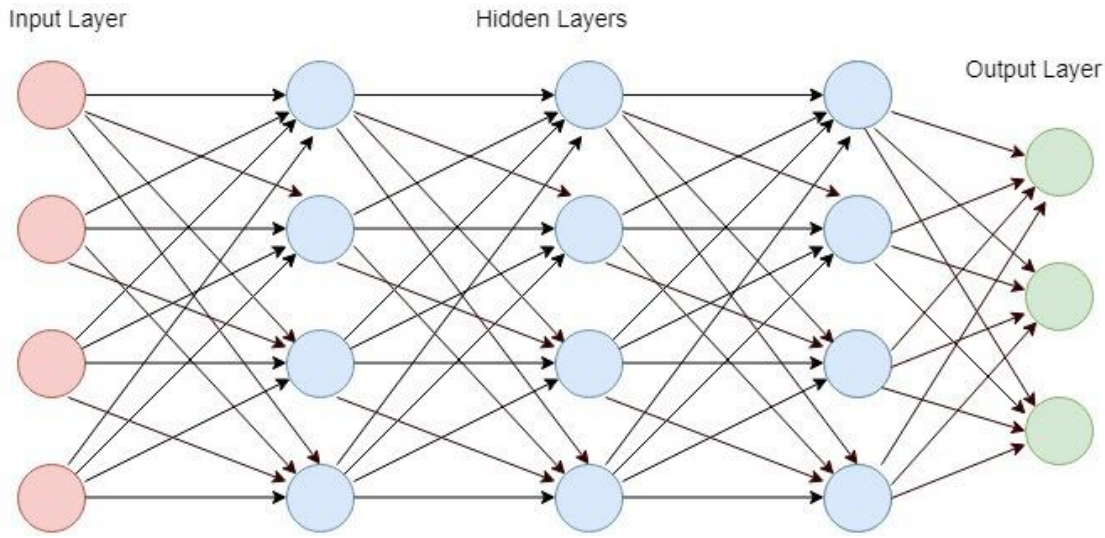

Figura 2.1 – Rețea neuronală feed-forward cu 1 strat de intrare, 3 straturi ascunse și un strat de ieșire

Stratul de intrare este primul strat de neuroni al rețelei, și este și stratul care primește datele de intrare. Numărul de neuroni din startul de intrare este determinat de numărul de valori din care sunt compuse datele de intrare ale unui exemplu de antrenament. Stratul de intrare nu realizează calcule sau transformări asupra datelor de intrare, doar le pasează mai departe către stratul următor, la primul strat ascuns al rețelei.

Straturile ascunse sunt poziționate în rețea între stratul de intrare și stratul de ieșire. Denumirea lor vine din faptul că straturile ascunse nu sunt expuse programatorului, nu sunt direct observabile din intrarea sau ieșirea rețelei, ele acționează ca un fel de cutie neagră care se ocupă de procesarea datelor. Numărul de neuroni din care este compus un strat ascuns și numărul de straturi ascunse din configurația unei rețele neuronale depind de complexitatea datelor pe care rețeaua neuronală încearcă să le învețe. Numărul de neuroni din stratul ascuns afectează mult performanța rețelei neuronale, creșterea numărului de neuroni din straturile ascunse poate îmbunătăți capacitatea rețelei de a învăța, dar conduce la o creștere a timpul de calcul necesar.

Stratul de ieșire este ultimul strat al rețelei pe care se va pune ieșirea sau predicția rețelei neuronale. Numărul de neuroni din ultimul strat depinde de problemă.

2.3.1 Structura unui neuron

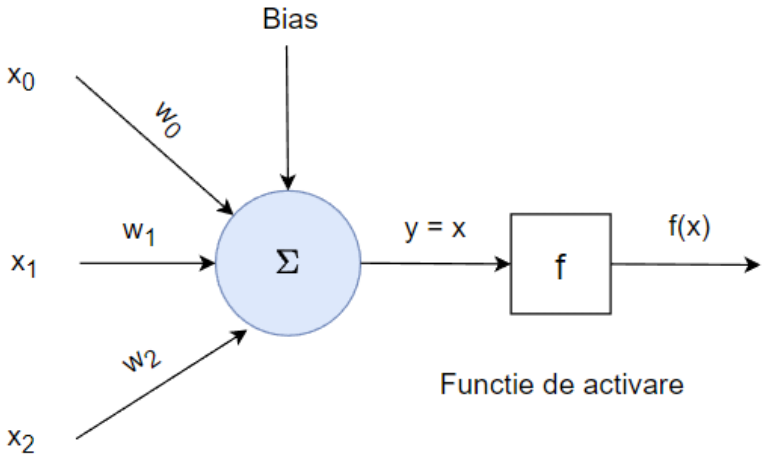

Figura 2.2 – Structura unui neuron din rețeaua neuronală artificială

Componentele neuronului prezentat în Figura 2.2 sunt :

- $x_1, x_2, \dots, x_n$ – reprezintă datele de intrare, care pot proveni de la stratul anterior al rețelei sau de la stratul de intrare
- $w_1, w_2, \dots, w_n$ – reprezintă ponderile, puterea conexiunilor dintre neuronii unei rețele.
- b – biasul, reprezintă valoarea de prag de activare a neuronului

Ponderile și bias-urile sunt componentele neuronului care permit modificarea rețelei neuronale pentru a învață și a lua decizii.

Fiecare conexiune dintre neuroni are asociată o pondere, care este o valoare numerică care reprezintă puterea legăturii și importanța pe care o are intrarea în calculul ieșirii unui neuron

Bias-ul, care mai poartă și numele de offset, reprezintă o valoare adăugată la suma ponderată a intrărilor neuronului și este folosită pentru a controla valoarea de ieșire a neuronului și pentru a schimba poziția funcției de activare, ajutând astfel rețeaua neuronală să învețe probleme mai complexe.

Ieșirea unui singur neuron, ca cel din Figura 2.2, este calculată cu formula :

$$y = x_1 * w_1 + x_2 * w_2 + \cdots + x_n * w_n + b$$

Această valoare calculată de neuron va trece apoi printr-o funcție de activare și va fi transmisă neuronilor de pe stratul următor la care este conectat neuronul curent.

2.3.2 Reprezentarea ponderilor și bias-urilor

Pentru a reprezenta ponderile și bias-urile unei rețele neuronale, se folosește o matrice de ponderi și o matrice de bias-uri pentru fiecare pereche de straturi ale rețelei.

Matricea de ponderi reprezintă o matrice bidimensională în care fiecare element reprezintă puterea legăturii între neuronii straturilor, iar matricea de bias-uri conține valoarea bias-urilor asociate cu neuronii celui de-al doilea strat din perechea de straturi. Pentru exemplu se va folosi stratul de intrare al rețelei neuronale care este prezentat în figura 2.3

Formulele prezentate sunt valabile și pentru restul straturilor ale rețelei, nu doar pentru stratul de intrare. O rețea neuronale ce conține 3 straturi (un strat de intrare, un strat ascuns și un strat de ieșire) va conține 2 pereche de straturi, strat intrare – strat ascuns și strat ascuns – strat ieșire și deci va conține 2 matrici de ponderi și 2 matrici de bias-uri.

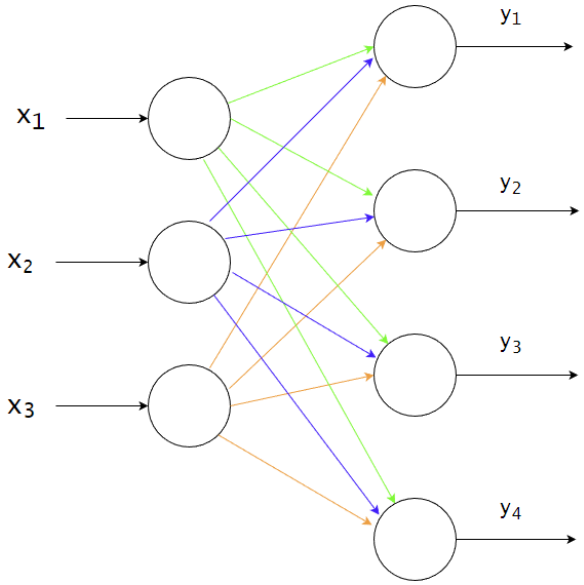

Figura 2.3 – Exemplu cu primele 2 straturi ale unei rețele neuronale

Matricea bias-urilor va fi o matrice formată dintr-o singura coloană, și un număr de n linii, unde n este număr de neuroni din al doilea strat al perechi de straturi. Pentru exemplul dat, matricea bias-urilor va fi formată din 4 linii. Valorile matricei de bias-uri pentru exemplul dat este prezentată mai jos:

$$\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

Unde $b1$ este bias-ul neuronului 1 din al doilea strat, $b2$ este bias-ul neuronului 2 din al doilea strat.

Matricea de ponderi va fi o matrice formata din n linii și m coloane, unde n este numarul de neuroni din al doilea strat, și m este numărul de neuroni din primul strat. Pentru exemplul dat, matricea ponderilor va avea 4 linii și 3 coloane. Valorile matricei de ponderi pentru exemplul dat este prezentată mai jos.

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \end{bmatrix}$$

Unde primul index al valorii este indexul neuronului din al doilea strat și al doilea index este indexul neuronului din primul strat care realizează conexiunea.

2.3.3 Regula Backpropagation

La începutul programului, o rețea neuronală este de obicei inițializată cu ponderi și bias-uri aleatorii, iar predicțiile pe care rețeaua le realizează vor fi și ele aleatorii. Trebuie folosit un algoritm de antrenament pentru a ajusta ponderile și bias-urile rețelei pentru ca rețeaua neuronală să învețe problema dată.

Regula backpropagation este un algoritm de învățare supervizat care folosește exemple de antrenament pentru a modifica ponderile rețelei neuronale astfel încât rețeaua să realizeze predicții asemănătoare cu predicțiile dorite care au fost memorate în aceste exemple de antrenament.
Un exemplu de antrenament este format din:

1. Intrare: Acesta reprezintă datele de intrare pe care rețeaua neuronală le primește pentru a face o predicție.
2. Ieșire țintă: Aceasta reprezintă valoarea corectă asociată cu intrarea respectivă.

Algoritmul backpropagation va folosi atât pasul forward al rețelei neuronale cât și pasul backward pentru modificarea ponderilor rețelei. Antrenarea folosind backpropagation va avea loc în mai iterații numite epoci. O epocă reprezintă o trecere completă prin toate exemplele de antrenament disponibile pentru a ajusta ponderile rețelei neuronale. În cadrul unei epoci se va realiza pentru fiecare exemplu de antrenament:

- Pasul forward (propagarea înainte a datelor) care va folosi datele de intrare care au fost obținute din exemplul de antrenament pentru a genera predicția rețelei neuronale neantrenate.
- Obținerea erorii dintre predicția rețelei neuronale neantrenate și ieșirea țintă care a fost memorată în exemplul de antrenament folosind o funcție de pierdere
- Pasul backward (propagarea înapoi a erorii) care folosește eroarea obținuta pentru a modifica ponderile rețelei astfel încât eroarea rețelei să se reducă. Acest lucru se realizează prin

calcularea gradientului erorii în raport cu ponderile și folosirea acestui gradient pentru a ajusta ponderile.

La sfârșitul fiecărei epoci se va verifică o condiție de oprire pentru a vedea dacă antrenarea rețelei se poate opri. Dacă condiția nu este satisfăcută atunci antrenarea va continua prin parcurgerea exemplelor de antrenament din nou trecând într-o epoca nouă. Condiția de oprire poate fi verificată prin compararea erorii obținute cu un prag prestabilit sau prin verificarea dacă numărul de epoci ale algoritmului a depășit o limită specificată.

2.3.4 Funcții de activare

Funcția de activare, sau funcția de transfer, este o funcție matematică utilizată pentru a modifica ieșirea unui neuron și pentru a redimensiona valoarea generată de acesta într-un alt domeniu. În mod general, toate straturile ascunse din rețea folosesc aceeași funcție de activare, dar stratul de ieșire poate folosi o funcție de activare diferită.

Există două tipuri de funcții de activare: funcții de activare liniare și funcții de activare neliniare.

Funcții de activare liniare

Funcțiile de activare liniare sunt simple transformări lineare ale valorilor primite la intrare. Ele sunt exprimate sub forma unei ecuații liniare și nu introduc non-linearitate în rețelele neurale.

Funcțiile de activare liniare sunt utilizate rar în straturile ascunse ale rețelelor neurale, deoarece acestea ar limita rezultatele și abilitatea de învățare a rețelei. Acest lucru apare din cauză că prin folosirea unei funcții de activare liniare, indiferent de numărul de straturi ascunse prin care vor trece datele, rezultatul este întotdeauna rezultatul unei funcții liniare. Prin urmare, rețeaua neurală cu funcții de activare liniare ar fi capabilă să aproximeze doar funcții liniare și nu ar putea învăța relații complexe.

Funcții de activare neliniare

Funcțiile de activare neliniare sunt funcții matematice care sunt folosite în rețele neuronale pentru a introduce non-linearitate, deci pentru a permite rețelei să învețe și să modeleze relații mai complexe între datele de intrare și ieșire.

Funcțiile de activare neliniare cele mai folosite sunt: sigmoida, tanh, ReLU (Rectified Linear Unit). Fiecare dintre acestea are avantajele și dezavantajele sale și este aleasă în funcție de cerințele și caracteristicile problemei de rezolvat.

Funcțiile de activare tanh sau sigmoidă ar putea să nu fie cele mai bune alegeri pentru funcția de activare care urmează să fie folosită pentru straturile ascunse ale rețelei deoarece ele prezintă problema vanishing gradient.

Vanishing gradient apare atunci când rețeaua neuronală este antrenată folosind metode de antrenament bazate pe gradient, cum ar fi regula backpropagation. Această problemă se caracterizează prin gradienți care devin foarte mici pe măsură ce sunt propagați înapoi în rețea în pasul backward al antrenării. În algoritmul backpropagation, gradienții sunt utilizați pentru a actualiza ponderile rețelei pentru a minimiza eroarea rețelei, dar atunci când gradienții obținuți devin foarte mici modificarea ponderilor se va realiza foarte încet.

Pentru a evita vanishing gradient se poate folosi funcția ReLU (Rectified Linear Unit) în locul funcțiilor sigmoidă și tangenta hiperbolică.

2.3.4.1 Funcția Sigmoidă

Funcția de activare sigmoidă este una dintre cele mai folosite funcții de activare. Un motiv al popularității funcției este pentru că domeniul valori de ieșire al funcției este (0,1), care este asemănător o probabilitate. Prin urmare, folosirea funcției sigmoide pentru activarea stratului de ieșire va converti ieșirile rețelei în probabilități, care vor fi mai ușor de înțeles și de utilizat.

Formula funcției sigmoide este:

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

Derivata funcției sigmoide este:

$$f'(x) = f(x)(1 - f(x))$$

2.3.4.2 Funcția Tangentă hiperbolică

Tangentă hiperbolică are domeniul valorilor funcției între (-1,1). O caracteristică a funcției tanh este că intervalul său de valori este simetric în jurul valorii 0. Acest lucru înseamnă că atunci când valorile de intrare se apropie de infinit pozitiv, rezultatul se apropie de 1, iar atunci când valorile de intrare se apropie de infinit negativ, rezultatul se apropie de -1. Această proprietate poate fi avantajoasă în anumite cazuri, deoarece permite rețelei să învețe atât valori pozitive, cât și negative și să identifice relații mai complexe.

Formula funcției tanh este:

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

Derivata funcției tanh este:

$$f'(x) = 1 - f^{2}(x)$$

2.3.4.3 Funcția ReLU

ReLU sau Rectified Linear Unit este o funcție de activare care este utilizată în locul funcțiilor sigmoidă sau tanh pentru a evita problema vanishing gradient. ReLU introduce non-liniaritate în rezultatele rețelelor neuronale prin returnarea valorii de intrare dacă aceasta este pozitivă și zero în caz contrar.

Formula funcției ReLU este:

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

Derivata funcției ReLU (Rectified Linear Unit) este definită diferit pentru intrările pozitive și negative:

1. Pentru intrări pozitive (x > 0): Derivata funcției ReLU este 1. Cu alte cuvinte, derivata funcției ReLU față de intrarea să este 1 atunci când intrarea este pozitivă.
2. Pentru intrările negative (x <= 0): Derivata funcției ReLU este 0. În acest caz, derivata funcției ReLU față de intrarea să este 0. Valorile neuronilor care determina derivata funcției ReLU să fie 0 nu își vor modifica ponderile.

Din cauza ca valorile derivatei lui ReLU sunt doar 0 sau 1 nu mai apare problema vanishing gradient, valorile gradienților nu vor mai scădea in timpul pasului backward al antrenării rețelei.

2.3.5 Pasul Forward

Pasul forward este procesul prin care rețeaua neuronală prelucrează datele de intrare pentru a genera valori la ieșire. În pasul forward al rețelei neuronale, rețeaua preia datele care au fost puse pe stratul de intrare și le transmite mai departe prin toate straturile rețelei, fiecare strat transformând intrările înainte de a le transmite mai departe către stratul următor. Valorile generate de stratul de ieșire vor fi predicțiile rețelei neuronale pentru intrarea dată.

Fiecare strat ascuns aplică o transformare prin înmulțirea ieșirilor stratului ascuns cu ponderile legăturilor stratului curent și adăugarea unor bias-urilor, după care se aplică o funcție de activare și rezultatele sunt trimise la stratul următor. Procesul continuă până când se ajunge la stratul de ieșire.

În figura 2.4 se prezintă propagarea datelor în pasul forward al rețelei feed-forward. Straturile rețelei sunt L1, L2 și L3. Semnalul de intrare al rețelei este X1 și este plasat pe primul strat al rețelei, pe stratul L1. Pentru realizarea propagării forward a datelor, datele de intrare vor trece prin 2 etape succesive pentru fiecare strat:

1. Calcularea valorilor neuronilor: Se vor realiza calculele prin înmulțirea semnalelor de intrare primite cu ponderile conexiunilor neuronilor, la care se adaugă bias-ul neuronilor din stratul curent..
2. Aplicarea funcției de activare: Se va aplica funcția de activare asupra valorilor calculate în etapa anterioară înainte de a transmite rezultate mai departe în rețeaua neuronală către neuronii stratului următor.

Valorile obținute de stratul L1 sunt denumite Y1. Straturile următoare ale rețelei vor primi ca semnale de intrare valorile care au fost calculate de către stratul anterior al rețelei. Procesul de propagare continuă până când se ajunge la ultimul strat al rețelei.

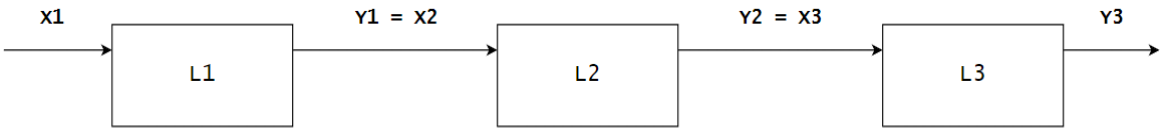

Figura 2.4 – Propagarea datelor în pasul forward al rețelei

Formulele pentru pasul Forward

Pentru a prezenta formulele pentru pasul forward al rețelei se va folosi ca exemplu propagarea datelor prin primele 2 straturile al rețelei neuronale. Straturile folosite sunt prezentate în figura 2.5.

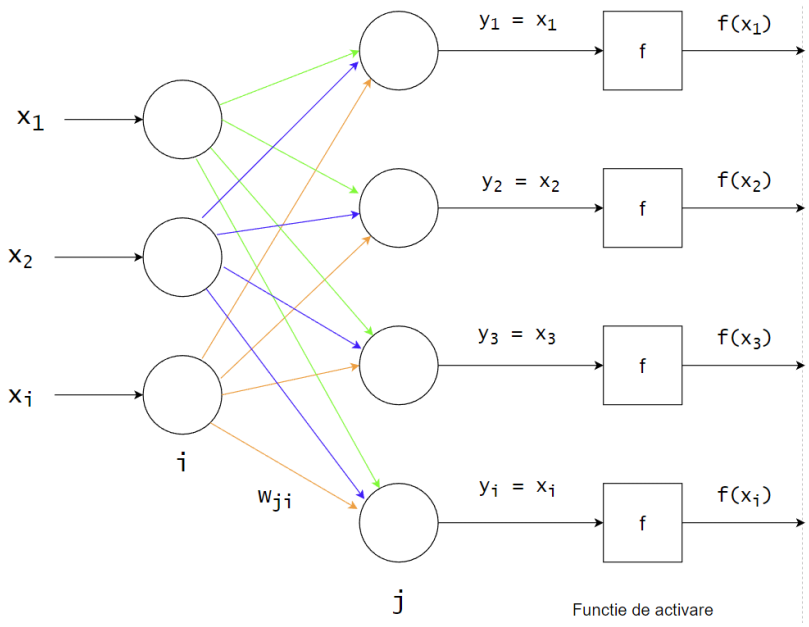

Figura 2.5 – Exemplu cu primele 2 straturi ale unei rețelei neuronale

Valorile ieșirilor neuronilor din Figura 2.5 se obțin astfel:

$$y_1 = x_1 * w_{11} + x_2 * w_{12} + \cdots + x_i * w_{1i} + b_1$$
$$y_2 = x_1 * w_{21} + x_2 * w_{22} + \cdots + x_i * w_{2i} + b_2$$
$$y_3 = x_1 * w_{31} + x_2 * w_{32} + \cdots + \mathrm{x_i} * \mathrm{w_{3i}} + \mathrm{b_3}$$
$$\vdots$$
$$y_j = x_1 * w_{j1} + x_2 * w_{j2} + \cdots + x_i * w_{ji} + b_j$$

Din formula de mai sus se poate obține formula generală de calcul a ieșirii neuronului j:

$$y_j = b_j + \sum_i x_i * w_{ji}$$

Formula de sus se poate scrie în formă matriceală ca:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_j \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1i} \\ w_{21} & w_{22} & \dots & w_{2i} \\ \vdots & \vdots & \dots & \vdots \\ w_{j1} & w_{j2} & \dots & w_{ji} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_i \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_j \end{bmatrix}$$

Ieșirile obținute de la neuroni vor trece apoi prin funcția de activare înainte de a fi transmise către stratul următor. Unde valorile x care sunt folosite pentru funcția de activare sunt valorile care au fost calculate mai sus.

$$y_1 = f(x_1)$$
$$y_2 = f(x_2)$$
$$\vdots$$
$$y_i = f(x_i)$$

2.3.6 Metode de calcul al erorii Rețelei Neuronale

Pentru ca rețeaua neuronală să producă la ieșire valori asemănătoare cu exemplele de antrenament, rețeaua trebuie să învețe din greșelile pe care le face prin calculul și propagarea erorii în rețea. Pentru a măsura diferențele dintre predicția rețelei și ieșirea dorită din exemplele de antrenament se folosește o funcție de eroare. Alegerea funcției de eroare depinde de complexitatea problemei și de datele de antrenament care sunt folosite.

2.3.6.1 Mean Squared Error

În statistică, eroarea pătratica medie (Mean Squared Error) sau abaterea pătratica medie este folosită pentru a măsura media diferențelor erorilor la pătrat.

Formula pentru MSE este:

$$E = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Parametrii funcției sunt:

- n: Numărul total de valori care alcătuiesc ieșirea rețelei neuronale sau ieșirea unui exemplu de antrenament
- y_i: Ieșirea care a fost obținută de rețeaua neuronală neantrenată
- $\hat{y}_i$: Valorile ieșirii țintă pentru rețeaua neuronală care sunt memorate în exemplele de antrenament

Un avantaj al funcției MSE este că erorile mari sunt penalizate mai mult decât erorile mici. Se poate vedea din formulă că o diferența mai mare dintre predicția rețelei și valoarea reală va avea o valoarea mai mare după ce este adusă la pătrat.

Un alt avantaj este că funcția MSE este derivabilă, deci poate să fie folosită în pasul backward pentru calculul gradientului erorii în funcție de ieșirea finală a rețelei neuronale.

Formula pentru derivata funcției MSE este:

$$\begin{aligned}\frac{\partial E}{\partial Y} &= \begin{bmatrix}\frac{\partial E}{\partial y_1} & \cdots & \frac{\partial E}{\partial y_i}\end{bmatrix} \\ &= \frac{2}{n}\begin{bmatrix}y_1 - y_1^* & \cdots & y_i - y_i^*\end{bmatrix} \\ &= \frac{2}{n}(Y - Y^*)\end{aligned}$$

2.3.7 Pasul Backward

În pasul backward se vor modifica ponderile și bias-urile rețelei neuronale pentru a reduce eroarea obținută de rețeaua neuronală. În timpul pasului backward, algoritmul trebuie să calculeze derivata erorii față de ponderile și derivata eroii față bias-urile fiecărui strat pentru a putea ajusta aceste valori și pentru a reduce eroarea rețelei. În plus, este necesară și calcularea derivatei erorii față de intrarea fiecărui strat, pentru a fi utilizată ca derivată erorii față de ieșire de către stratului următor care va aplica pasul backward.

În pasul backward se poate vedea o relație între intrările și ieșirile straturilor adiacente ale rețelei, prezentă într-o formă asemănătoare și la pasul forward, dar aici pentru derivatele erorii și

în direcție inversă fată de pasul forward: derivată erorii față de intrare pentru stratul curent va fi considerată derivata erorii față de ieșire de către stratul următor. Propagarea datelor în pasul backward este prezentată în figura 2.6.

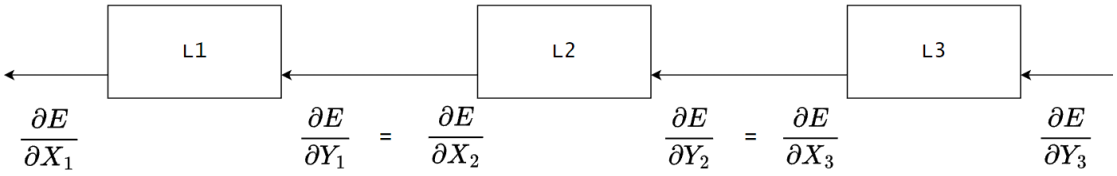

Figura 2.6 – Propagarea datelor în pasul backward

Etapele pasului backward vor fi în ordine inversă fată de pasul forward:

1. Calculul derivatei erorii în funcție de intrare folosind derivata funcției de activare. Aceasta valoare va fi folosită pentru calculul derivatei erorii fată de ponderi și calculul erorii fată de bias-uri în etapa doi a pasului backward.
2. Calculul derivatei erorii fată de ponderi și calculul derivatei erorii fată de bias-urile stratului pentru a actualiza ponderile și bias-urile pentru a reduce eroarea, și calculul derivatei erorii față de intrare care va fi transmisă către stratul anterior al rețelei.

Derivata erorii în fată de ieșire pentru ultimul strat al rețelei va fi eroarea care a fost obținută de către derivata funcției de pierdere folosită. Derivata eroii față de ieșire pentru stratul anterior al rețelei se va obține prin calcularea derivatei erorii față de intrare de pentru stratul curent. Din cauza acestei propagări a erorii înapoi în rețea acest pas poartă denumire de „backpropagation”.

Formulele pentru pasul Backward

Pentru calculul derivatei erorii fată de intrare se va folosi derivata funcției de activare și formulele care au fost folosite pentru pasul forward.

Pentru calculul derivatei erorii față de intrarea x_i se va folosi regula lanțului. După cum se poate vedea în formulele pentru stratul forward, variabila x_i este folosită pentru a calcula doar valoarea de ieșire y_i, toate derivatele ieșirilor în funcție de x_i vor fi 0 dacă ieșirea y nu se folosește x_i. Astfel, se ajunge la concluzia că derivata valorii y_i față de x_i va fi egala cu derivata funcției de activare care primește ca parametru x_i.

Se obține formula pentru derivata erorii în funcție de intrare:

$$\frac{\partial E}{\partial x_i} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial x_i} + \frac{\partial E}{\partial y_2}\frac{\partial y_2}{\partial x_i} + \cdots + \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial x_i} = \frac{\partial E}{\partial y_i} f'(x_i)$$

Formule pentru etapa a doua a pasului backward se obțin din formulele pentru etapa 1 a pasului forward folosind regula lanțului.

Derivata erorii față de ponderi:

Atunci când se calculează derivata erorii în funcție de ponderea w_{ji} se va folosi regula lanțului pentru a desparți derivata într-o suma de derivate de forma:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial w_{ji}} + \frac{\partial E}{\partial y_2}\frac{\partial y_2}{\partial w_{ji}} + \cdots + \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial w_{ji}}$$

Derivata erorii față de ieșiri este cunoscută, trebuie calculată doar derivata ieșirii față de pondere. Formula derivatei erorii față de w_{ji} va fi:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial y_j} x_i$$

Formula de sus se poate scrie și folosind matrici:

$$\frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial y_1} \\ \frac{\partial E}{\partial y_2} \\ \vdots \\ \frac{\partial E}{\partial y_j} \end{bmatrix} \begin{bmatrix} x_1 & x_2 & \dots & x_i \end{bmatrix} = \frac{\partial E}{\partial Y} \cdot X^t$$

Matricea derivatelor erorii fată de ponderi va fi egală cu matricea derivatelor erorii fată de ieșiri înmulțită folosind produs scalar cu transpusa matricei intrărilor.

Derivată erorii față de bias:

Atunci când se calculează derivata erorii în funcție de ponderea w_{ji} se va folosi regula lanțului pentru a desparți derivata într-o suma de derivate de forma:

$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial b_j} + \frac{\partial E}{\partial y_2}\frac{\partial y_2}{\partial b_j} + \cdots + \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial b_j}$$

Derivata erorii față de ieșiri este cunoscută, trebuie calculată doar derivata ieșirii față de bias. Formula derivatei erorii față de b_j va fi:

$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial y_j}$$

Derivata erorii față de input:

Atunci când se calculează derivata erorii în față de intrarea x_i se va folosi regula lanțului pentru a desparți derivata într-o suma de derivate de forma:

$$\frac{\partial E}{\partial x_i} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial x_i} + \frac{\partial E}{\partial y_2}\frac{\partial y_2}{\partial x_i} + \cdots + \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

Derivata erorii față de ieșiri este cunoscută, trebuie calculată doar derivata ieșirii față de intrare. Formula derivatei erorii față de x_i va fi:

$$\frac{\partial E}{\partial x_i} = \frac{\partial E}{\partial y_1}w_{1i} + \frac{\partial E}{\partial y_2}w_{2i} + \cdots + \frac{\partial E}{\partial y_j}w_{ji}$$

Care se poate scrie sub forma matriceală ca:

$$\frac{\partial E}{\partial X} = \begin{bmatrix} \frac{\partial E}{\partial y_1}w_{11} + \frac{\partial E}{\partial y_2}w_{21} + \cdots + \frac{\partial E}{\partial y_j}w_{j1} \\ \frac{\partial E}{\partial y_1}w_{12} + \frac{\partial E}{\partial y_2}w_{22} + \cdots + \frac{\partial E}{\partial y_j}w_{j2} \\ \vdots \\ \frac{\partial E}{\partial y_1}w_{1i} + \frac{\partial E}{\partial y_2}w_{2i} + \cdots + \frac{\partial E}{\partial y_j}w_{ji} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{21} & \dots & w_{j1} \\ w_{12} & w_{22} & \dots & w_{j2} \\ \vdots & & & \\ w_{1j} & w_{2j} & \dots & w_{ij} \end{bmatrix} \begin{bmatrix} \frac{\partial E}{\partial y_1} \\ \frac{\partial E}{\partial y_2} \\ \vdots \\ \frac{\partial E}{\partial y_j} \end{bmatrix} = W^t \frac{\partial E}{\partial Y}$$

Pentru modificarea ponderilor și bias-urilor rețelei se va folosi derivata erori în funcție de ponderi și derivata erorii în funcție de bias în cadrul fiecărui strat după cum este prezentat în formula următoare:

$$w_{ji} = w_{ji} - \alpha\frac{\partial E}{\partial w_{ji}}$$

$$b_j = b_j - \alpha\frac{\partial E}{\partial b_j}$$

Unde α este rata de invatare, o valoare între 0 și 1 care este folosită pentru a controla cu cât de mult se vor schimba ponderile și bias-urile rețelei neuronale.

2.4 Algoritmul genetic

Algoritmii genetici sunt algoritmi de căutare euristică adaptivi care se inspiră din concepte de genetică și din Teoria Evoluției Naturale propusă de Charles Darwin, mai exact din mecanismul de selecție naturală care descrie ideea de „survival of the fittest" sau „supraviețuirea celor mai adaptați". Algoritmii genetici sunt folosiți pentru a genera soluții pentru probleme de optimizare și de căutare.

Selecția naturală este procesul prin care populațiile de organisme vii se adaptează și se schimbă în timp pentru a deveni mai potrivite pentru mediul în care trăiesc. Indivizii dintr-o populație sunt în mod natural diferiți unii de alții. Această variație înseamnă că unii indivizi au trăsături care ii fac mai potriviți pentru mediul în care trăiesc decât alți indivizi. Indivizii cu trăsături avantajoase au mai multe șanse de a supraviețui și de a se reproduce, astfel trăsăturile lor au șanse mai mari de a fi transmise descendenților din generația următoare. Prin acest proces de selecție naturală, trăsăturile benefice se răspândesc în populație, iar cele dăunătoare sunt eliminate.

Algoritmii genetici simulează acest proces de selecție naturală utilizând operatori genetici de selecție, crossover și mutație, care imită procesele naturale descrise de Darwin. În esență, algoritmii genetici încearcă să realizeze procesul de "supraviețuire a celor mai adaptați" în care indivizii din populație reprezintă soluții pentru problema dată, iar soluțiile cele mai potrivite sunt acelea care obțin rezultatele cele mai bune.

În algoritmii genetici o populație de soluții candidate pentru o problema dată sunt evoluate pentru a obține soluții mai bune. Fiecare individ are un set de proprietăți sau valori care pot să fie alterate în timpul rulării programului pentru a realiza explorarea și optimizarea spațiului soluțiilor posibile.

Algoritmul începe de la o populație de câteva sute sau mii de indivizi cu configurații aleatorii. Evoluția este un proces iterativ și populația de indivizi din fiecare iterație poartă numele de generație. La sfârșitul fiecărei generații, performanța fiecărui individ din populație este evaluată prin intermediul unei funcții de fitness, care atribuie un scor care cuantifica performanța obținută de acel individ. Scorurile obținute de indivizi reflectă cât de bine se potrivește fiecare soluție pentru problema dată. Indivizii cei mai potriviți, care au un scor fitness mai mare, sunt selectați din populație pentru a generat indivizi descendenți care vor forma generația următoare a algoritmului genetic.

Fazele algoritmului genetic:

1. Inițializare: La începutul algoritmului se generează o populație în care indivizii au valorile cromozomilor inițializate aleator.
2. Evaluare: Se evaluează performanta fiecărui individ din populație folosind o funcție de fitness. Aceasta funcție măsoară cât de bine se potrivește fiecare individ cu obiectivele problemei. Rezultatul evaluării este un scor de fitness asociat fiecărui individ care reflectă calitatea soluției.
3. Selecție: În faza de selecție se alege un subset de indivizi din populație care vor trece prin procesul de crossover și mutație pentru a genera indivizii generației următoare a algoritmului genetic.
4. Crossover: În această fază a algoritmului se realizează schimbul de informație genetică între indivizii care au fost selectați de către operatorul de selecție pentru a obține indivizi noi. Prin crossover, porțiuni material genetic de la doi părinți diferiți sunt combinate pentru a crea descendenți care moștenesc caracteristici din ambele surse.
5. Mutație: Indivizii care au fost obținuți prin operatorul de crossover vor trece prin faza de mutație, în care anumite valori ale genelor lor vor fi schimbate pentru a menține diversitatea genetică a populației.
6. Finalizare: La sfârșitul fiecărei generații al din algoritm se verifică dacă criteriul de oprire a fost atins. De exemplu, algoritmul poate fi oprit atunci când nu se observă o îmbunătățire semnificativă a scorului de fitness pentru un număr de generații sau dacă numărul de generații care a fost obținut de algoritmul genetic depășește o limită. Dacă criteriul de oprire nu a fost atins atunci algoritmul genetic va continua execuția, se vor aplica pașii de evaluare, selecție, crossover și mutație asupra indivizii care au fost obținuți în ultima generație a algoritmului.

2.4.1 Funcția de Fitness

În algoritmii genetici, funcția de fitness sau funcția de evaluare este utilizată pentru a determina cât de potrivit este un individ din populație în raport cu obiectivele propuse prin atribuirea acelui individ cu o valoare numită scor de fitness. Acest scor este esențial în procesul de selecție, unde operatorii de selecție utilizează scorurile de fitness pentru a alege indivizii care vor trece prin procesul de reproducere, indivizii cu scoruri mai mari având șanse mai mari să fie selectați pentru a-și propaga informația genetică în generația următoare.

Formula folosită de către funcția de fitness depinde de problemă și de către obiectivele care se urmăresc. Definirea corectă a funcției de fitness este critică pentru obținerea de rezultate bune folosind algoritmul genetic. Prin urmare, este important să se ia în considerare obiectivele dorite

și să se definească în mod corespunzător funcția de fitness astfel încât funcția să reflecte importanța obiectivelor. În caz contrar, algoritmul poate ajunge la o soluție sub-optimală.

2.4.2 Operatori de Selecție

La sfârșitul fiecărei generații, o porțiune din indivizii populației sunt selectați pentru a se reproducere și pentru a obține noua generație a algoritmului genetic. Această selecție a indivizilor este realizată folosind un operator de selecție.

Majoritatea metodelor de selecție sunt metode dependente de fitness, în care indivizii cu un scor de fitness mai mare au șanse mai mari să fie aleși pentru reproducere. Alegerea indivizilor pe baza fitness-ului se face pentru a asigura că indivizii cei mai potriviți pentru problema au șanse mai mari de a fi selectați pentru reproducere, iar descendenții lor obținuți prin reproducere să conțină trăsături dorite, îmbunătățind astfel calitatea populației.

Cu toate acestea, există un risc atunci când se selectează doar cei mai buni indivizi din populație: această metodă de selecție poate cauza convergența la o soluție suboptimă înainte de a atinge maximul global al problemei. Acest fenomen este cauzat de pierderea diversității genetice a populației, lucru care reduce spațiul de căutare al algoritmului.

Pentru a evita această problemă, se folosesc operatori de selecție mai puțin stricți, care oferă șanse și indivizilor mai puțin adaptați să fie selectați pentru împerechere, menținând astfel diversitatea genetică a populației. Una dintre aceste metode poate fi selecția ruletei, care acordă o șansă și indivizilor cu scoruri de fitness mai mici de a contribui la generația următoare. Prin menținerea diversității genetice, algoritmul genetic poate continua să exploreze spațiul soluțiilor în căutarea soluției optime pentru problema dată.

2.4.2.1 Roulette Wheel Selection

Este un operator de selecție în care probabilitatea ca un individ să fie ales este direct proporționala cu fitness-ul său. Dacă f_i este scorul fitness al individului i din populație, și populația conține N indivizi, atunci probabilitatea ca el să fie selectat din cadrul acelei populații este dată de formula:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$$

Metodă această de calculare a probabilităților poate fi comparată cu o ruletă de la un cazino în care fiecare individ are rezervat un spațiu pe ruletă și dimensiunea fiecărui spațiu este direct proporțională cu scorul de fitness al acelui individ. Pentru a selecta n indivizi din populație se poate roti această roata imaginara de n ori.

Utilizând această metodă de selecție, indivizii cu un fitness mai mare au șanse mai mari să fie aleși pentru reproducere, dar și indivizii mai puțin performanți au șanse de a fi selectați.

În tabelul următor în care se prezintă 5 indivizi fiecare cu un fitness diferit. Probabilitatea ca un individ să fie selectat folosind roulette wheel este prezentată în figura 2.7

Individual	Fitness
1	15
2	5
3	20
4	45
5	25

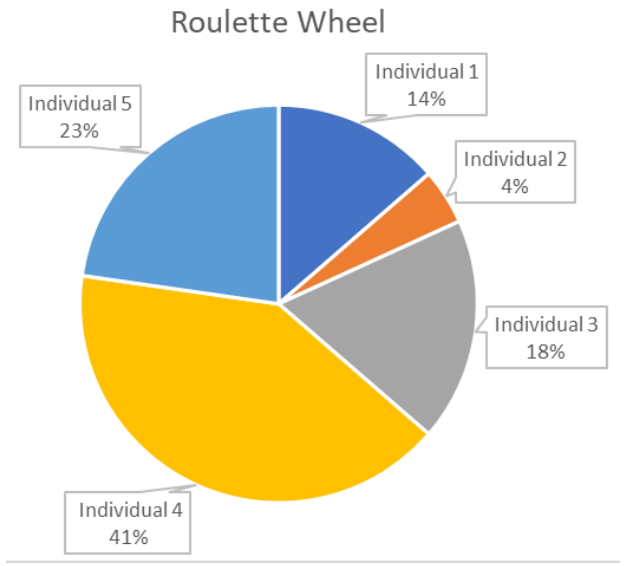

Figura 2.7 – Exemplu de funcționare al operatorului Roulette Wheel

După cum se poate vedea în figura de mai sus, individul 4, care are cel mai mare scor de fitness, va avea cea mai mare șansa să fie selectat, iar individul 2 care are cel mai mic scor de fitness are cea mai mică șansă să fie selectat.

2.4.2.2 Elitist Selection

La selecția elitistă se selectează indivizii cu cel mai mare scor de fitness din populație. Selecția elitistă nu este foarte utilă dacă este utilizată ca singura metodă de selecție, deoarece prezintă problemele menționate anterior care apar atunci când se aleg doar cei mai buni indivizi pentru reproducere. Din cauza acestor probleme, selecția elitistă este de obicei utilizată în combinație cu alte metode de selecție.

Aceasta metoda este folosită pentru a alege un subset al celor mai buni indivizi din populație care să treacă la generația următoare fără să fie afectați de recombinare și mutație, ceea ce ajută la prevenirea pierderii de material genetic valoros și poate duce la o convergență mai rapidă către o soluție optimă. Restul indivizilor care vor forma generația următoare vor fi obținuți prin reproducerea și mutația indivizilor din generația curenta.

Astfel, selecția elitistă servește la conservarea și propagarea celor mai bune trăsături genetice din populație, asigurând că acestea continuă să fie prezente în generațiile ulterioare.

2.4.3 Operatori de Crossover

Crossover-ul sau recombinarea este un operator genetic utilizat pentru a combina informația genetică a părinților și a genera descendenți cu configurații noi.

Prin combinarea materialului genetic al părinților, recombinarea ajuta la menținerea diversității populației și prevenirea convergenței algoritmului la un minim local. Prin schimbul de informații genetice între părinți, crossover-ul contribuie la generarea de descendenți care prezintă combinații unice ale trăsăturilor moștenite. Această diversitate asigură că populația continuă să exploreze diferite regiuni ale spațiului de căutare, evitând convergența către o soluție suboptimă.
Rolurile Operatorilor de Crossover:

- Menținerea diversității
- Explorarea spațiului de căutare
- Evitarea convergenței la minime locale

2.4.3.1 One-point Crossover

Aceasta metodă de crossover este caracterizată de faptul că folosește un singur punct de crossover pentru a delimita materialul genetic al părinților care va fi folosit pentru generarea copiilor. Punctul de crossover reprezintă poziția în care materialul genetic al părinților va fi tăiat și schimbat.

După alegerea punctului de crossover, materialul genetic al părinților este schimbat pentru a crea doi indivizi noi. Primul copil este obținut prin combinarea materialului genetic de la primul părinte de la început până la punctul de crossover, cu materialul genetic al celui de-al doilea părinte de la punctul de crossover până la sfârșitul cromozomului. Astfel, primul copil primește o combinație a trăsăturilor genetice ale ambilor părinți.

Al doilea copil este creat într-un mod similar, dar sursele materialului genetic folosit sunt inversate. Materialul genetic înainte de punctul de crossover este luat de la al doilea părinte, iar materialul genetic după punctul de crossover este preluat de la primul părinte. Acest proces de schimb al materialului genetic între părinți duce la crearea unui al doilea copil cu o combinație diferită de trăsături genetice față de primul copil.

Funcționarea operatorului One-point Crossover este prezentată în figura 2.8.

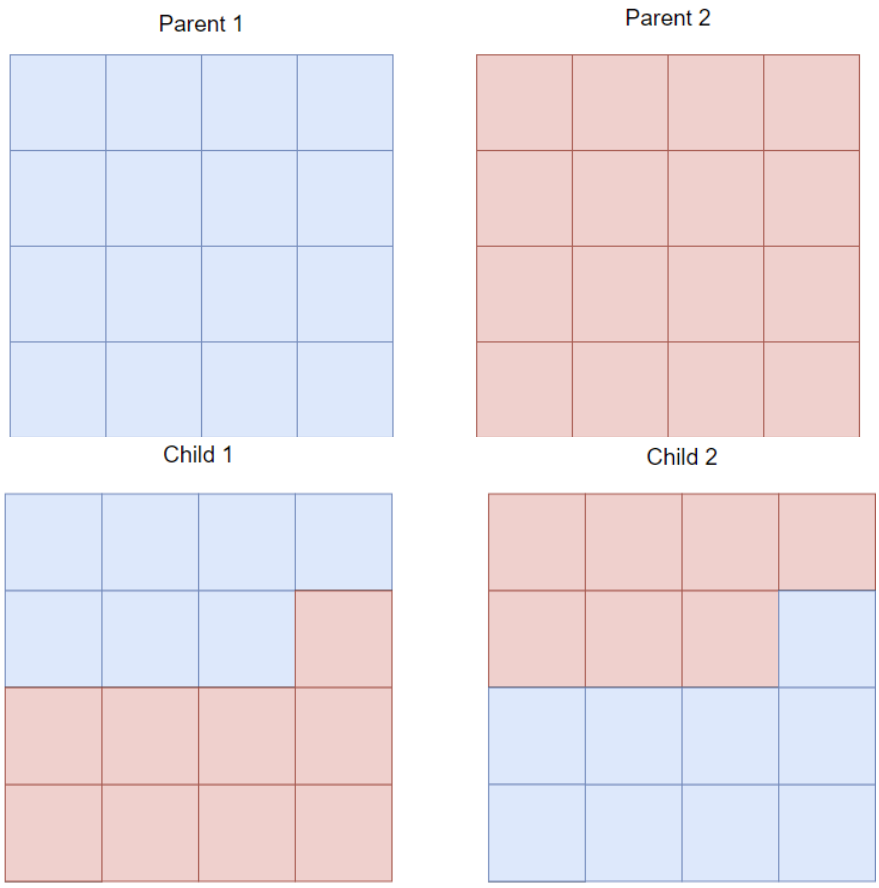

Figura 2.8 – Exemplu de funcționare al operatorului One-point Crossover

2.4.3.2 Two-point Crossover

Metoda Two-Point Crossover extinde operatorul One-Point Crossover prin utilizarea a două puncte de crossover în loc de unul singur. Această tehnică permite un schimb suplimentar de material genetic între părinți, obținând astfel descendenți cu o diversitate genetică mai mare.

Pentru a genera primul copil, se combină materialul genetic de la primul părinte de la începutul cromozomului până la primul punct de crossover, apoi se preia materialul genetic de la al doilea părinte între cele două puncte de crossover și, în final, se adaugă materialul genetic de la primul părinte de la al doilea punct de crossover până la sfârșitul cromozomului.

Al doilea copil este generat într-un mod similar. Materialul genetic de la al doilea părinte este utilizat între cele două puncte de crossover, iar materialul genetic în afara acestor puncte este preluat de la primul părinte.

Funcționarea operatorului Two-point Crossover este prezentată în figura 2.9

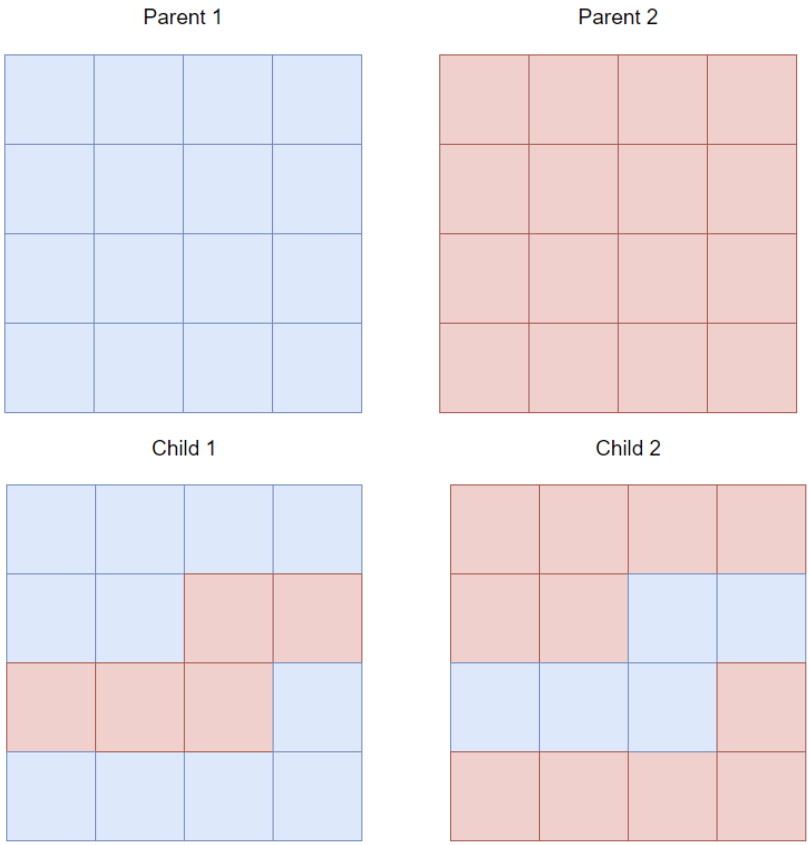

Figura 2.9 – Funcționarea operatorului Two-Point Crossover

2.4.4 Operatori de mutație

Mutația este un operator genetic crucial în algoritmul genetic care realizează schimbarea aleatoare a materialului genetic al unui individ, adăugând variații și crescând diversitatea populației.

Operatorii de mutație sunt folosiți pentru a crește diversitatea populației prin modificarea informației genetice a indivizilor și astfel pentru a evita apariției minimelor locale. Prin modificarea aleatoare a materialului genetic al indivizilor se asigură că indivizii nu devin prea similari unii cu ceilalți, permițând astfel explorarea unui spațiu de căutare mai larg

Cu toate acestea, este important să se folosească mutația cu precauție în algoritmul genetic. O rată prea mare de mutații poate duce la pierderea de informație genetică și poate încetini convergența către soluția optimă. Prin urmare, este necesar să se aleagă o rată de mutație adecvată, care să mențină diversitatea genetică în populație, fără a afecta negativ performanța algoritmului genetic.

2.4.4.1 Mutație Gausiană

Operatorul de mutație Gaussiană modifică aleator valorile cromozomului unui individ din populație prin adăugarea unei valori care a fost extrasă dintr-o distribuție Gaussiană.

Formula mutației gaussiene este:

$$Valoare\ Noua = Value\ Veche + N(\mu, \sigma)$$

Unde parametrii folosiți:

- μ: Reprezintă valoarea centrală sau valoarea medie în jurul căreia sunt distribuite simetric valorile distribuției gaussiene.
- σ: Deviația standard care controlează dispersia sau variabilitatea valorilor obținute. O deviație standard mai mare indică o dispersie mai mare a valorilor, în timp ce o deviație standard mai mică indică faptul că punctele de date sunt mai apropiate de medie.
- $N(\mu, \sigma)$: Este o valoarea aleatoare care a fost extrasa dintr-o distribuție gaussiană folosind media μ și deviația standard σ

Împreună, media și deviația standard determină forma și caracteristicile distribuției gaussiene. Media definește centrul distribuției, în timp ce deviația standard determină lățimea distribuției.

Prin ajustarea deviației standard în algoritmul genetic se poate controla magnitudinea modificărilor aduse valorilor originale ale cromozomului. Prin creșterea deviației standard intervalul de valori posibile pentru mutație se extinde, ceea ce duce la schimbări mai mari în cromozomi. În schimb, o deviație standard mai mică limitează variația adusă de mutație și reduce modificările care sunt aplicate valorilor originale.

2.4.5 Folosirea algoritmului genetic pentru antrenarea rețelei neuronale

Pentru antrenarea rețelelor neuronale se pot folosi algoritmii genetici în locul algoritmului Backpropagation. Când sunt folosiți algoritmi genetici, un cromozom va fi format din totalitatea matricelor de ponderi și bias-uri ale rețelei neuronale.

Optimizarea și schimbarea ponderilor apare prin procesul de selecție naturală, rețelele neuronale care obțin rezultate mai bune vor fi selectate pentru împerechere și atunci ponderile și bias-urile vor se vor propaga prin procesele de selecție, crossover și mutație la rețele neuronale care vor forma generația următoare a algoritmului genetic. După mai multe generații ale algoritmului se vor obține rețele care rezolva mai bine problema dată.

Un avantaj al folosirii algoritmilor genetici este că acum antrenarea este nesupervizată, nu mai este nevoie de exemple de antrenament pentru a realiza antrenarea.

2.5 Distanța Manhattan

Pentru a calcula distanțele dintre elementele care se află pe tabla jocului se folosește distanța Manhattan.

Distanta Manhattan, care se mai numește și distanța City Block, este folosită pentru a calcula distanța dintre elemente care se află pe o matrice uniforma, cum ar fi tabla de șah sau blocuri de oraș din cartierele americane. Ea se calculează ca suma absolută a diferențelor dintre coordonatele corespondente ale celor două blocuri de pe tablă.

De exemplu, pentru două puncte care se afla într-un spațiu bidimensional, cum ar fi punctul A cu coordonatele (x_1, y_1) și punctul B cu coordonatele (x_2, y_2), distanța dintre ele va fi dată de formula:

$$D = |x_1 - x_2| + |y_1 - y_2|$$

Cu alte cuvinte, distanța Manhattan este suma absolută a diferențelor dintre coordonatele corespondente ale celor două puncte pe axa orizontala și pe axa verticala și reprezintă numărul total de mutări necesare pentru a ajunge de la un punct la altul, deplasând-se doar pe orizontala și verticala, fără mișcări pe diagonală.

2	1	2
1	Start	1
2	1	2

Figura 2.10 – Exemplu de distanțe obținute folosind distanța Manhattan

3 Rezolvarea Temei de proiect

Programul folosește o rețea neuronală de tip Feed-Forward cu un singur strat ascuns de neuroni și regula delta de antrenare pentru a prezice mișcarea următoare pe care trebuie să o facă șarpele în funcție de starea în care se află tabla de joc în acel moment. Pentru predicția mișcării șarpelui se va folosi pasul Forward al rețelei neuronale care primește ca intrări o viziune redusă a tablei de joc.

La începutul jocului, rețeaua neuronală va fi inițializată cu ponderi aleatoare, și deci mișcările pe care șarpele le va face for fi și ele aleatoare. Pentru ca rețeaua să prezică corect direcția pe care să o ia șarpele se vor folosi metode de antrenament care vor modifica ponderile rețelei. Pentru antrenarea rețelei neuronale au fost implementate două metode separate de antrenament:

1. Algoritmi Genetici
2. Regula Backpropagation

Pentru umbelele metode de antrenament, intrarea rețelei neuronale va fi formată dintr-o viziune redusă a tablei de joc. Intrarea va fi formată dintr-o mulțime de „linii de viziune”, raze care se proiectează din capul șarpelui pe tablă în mai multe direcții și reține distanțele dintre capul șarpelui și diferite blocuri care se află pe acea direcție (perete, măr și segment al corpului).

Antrenarea folosind algoritmul genetic va folosi exclusiv pasul forward al rețelei neuronale, iar antrenarea folosind regula backpropagation va folosi pasul forward și pasul backward.

1. Algoritmi Genetici

Antrenarea folosind algoritmi genetici va fi realizată automat, fără intervenția utilizatorului. La începutul algoritmului se va forma o populație de indivizi care conțin rețele neuronale ce au ponderile inițializate aleator. Pentru evaluarea indivizilor, programul va lua pe rând fiecare rețea neuronală și va realiza mutarea șarpelui pe tablă folosind direcția prezisă de rețea până când șarpele moare (se lovește de un perete sau de un segment al corpului), moment în care se va folosi funcția de fitness pentru a evalua performanța obținută de rețeaua șarpelui ținând cont de numărul de pași și scorul care a fost obținut de șarpe cât timp a fost în viață.

Generația următoare a algoritmului genetic va fi obținută prin aplicarea operatorilor genetici asupra rețelelor neuronale ale indivizilor din generația curentă. Un individ care are scorul de fitness mai mare (are performanțe mai bune) are șanse mai mari să fie ales pentru reproducere, și deci pentru a-și transmite mai departe ponderile rețelei sale la rețelele neuronale care vor forma generația următoare a algoritmului. Această selecție și propagare a informației genetice prin

reproducere ajută la îmbunătățirea performanței algoritmului genetic în timp prin convergența către soluții mai bune.

Atunci când utilizatorul alege să oprească antrenarea, se vor salva ponderile rețelelor neuronale obținute de algoritm în fișiere json pentru a putea fi rulate și testate de către utilizator.

2. Regula Backpropagation

Antrenarea folosind Regula Backpropagation reprezintă o formă de antrenare supervizată, în care algoritmul ajustează ponderile rețelei neuronale pe baza exemplelor de antrenament furnizate de utilizator.

Pentru a genera aceste exemple de antrenament, algoritmul utilizează mișcările efectuate de utilizator. Programul inițializează jocul și permite utilizatorului să joace, iar mutările șarpelui pe tablă vor fi realizate prin intermediul input-ului utilizatorului. Pentru fiecare mișcare efectuată de utilizator se creează un exemplu de antrenament care conține valorile liniilor de viziune obținute pentru tabla și mișcarea realizată de utilizator pentru acea stare.

În momentul în care utilizatorul alege să oprească generarea exemplelor de antrenament se va realiza antrenarea rețelei folosind regula backpropagation. Astfel, rețeaua neuronală învață să copieze mișcările realizate de utilizator în exemplele de antrenament pentru a-și îmbunătăți performanța în joc.

După ce procesul de antrenare a fost finalizat, ponderile rețelei neuronale obținute se vor salva într-un fișier json pentru rulare și testare.

3.1 Structura Aplicației

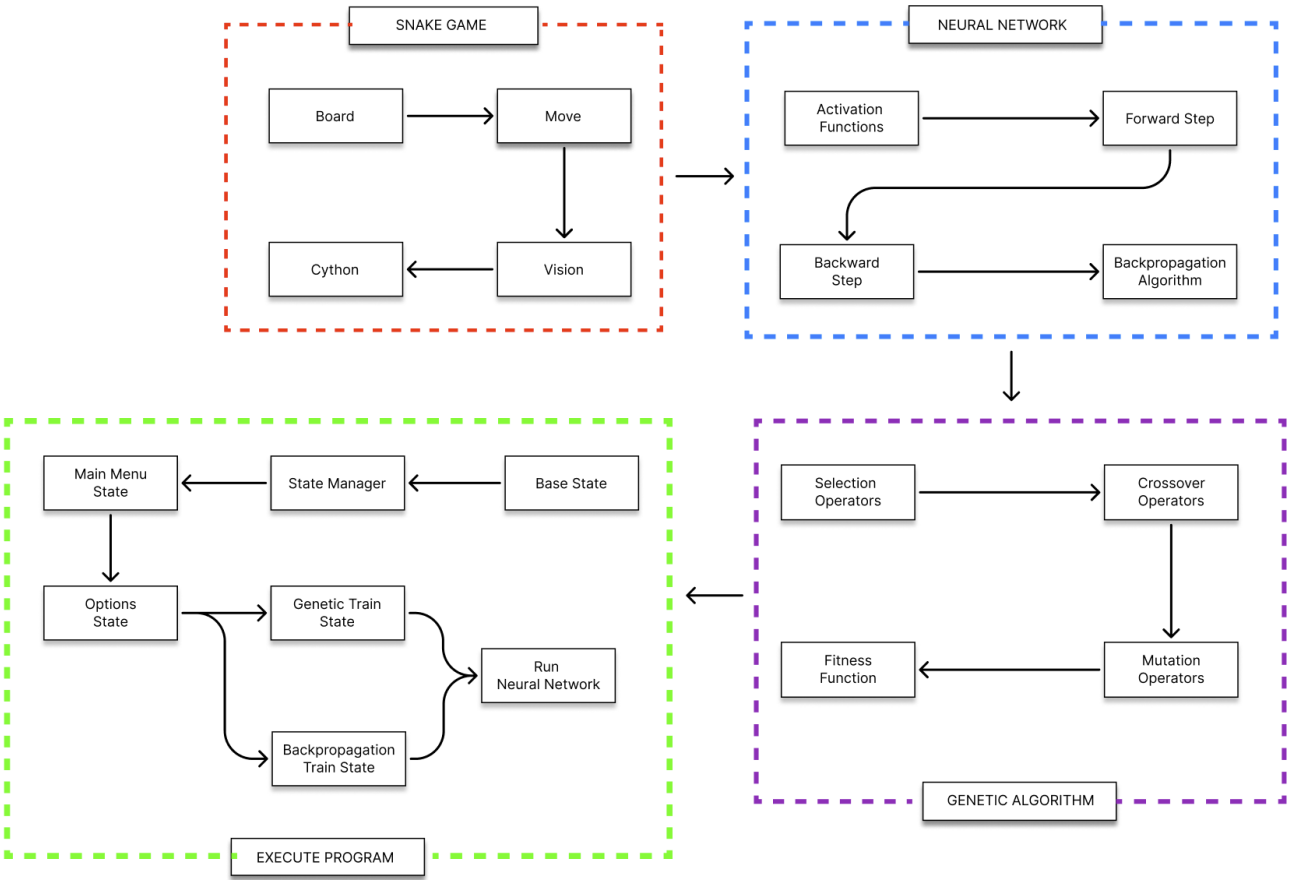

Figura 3.1 Progresul dezvoltării aplicației

Aplicația realizată poate fi împărțită logic în patru blocuri majore, care sunt ilustrate mai sus în Figura 3.1

1. Jocul de Snake: Acest modul se ocupă de implementarea funcționarii jocului Snake. Aici se creează tabla de joc, se poziționează șarpele pe tablă și se implementează mutarea acestuia. De asemenea, în acest modul se implementează liniile de viziune care vor fi folosite pentru a obține o parte din intrarea rețelei neuronale.
    a. Linii de viziune – O linie de viziune reține poziția și coordonatele obiectelor care se află pe o anumită direcție pe care privește șarpele.
    b. Îmbunătățirea timpului de execuție folosind Cython – Pentru îmbunătățirea timpul de execuție al programului s-a folosit extensia Cython pentru a rescrie funcțiile consumatoare de timp în limbajul Cython, un limbaj hibrid care permite generarea de cod C ce poate fi folosit în Python, ceea ce duce la o performanță superioară în comparație cu codul care a fost scris folosind doar Python.

2. Rețeaua Neuronala: Acest modul implementează rețeaua neuronală Feed-Forward a cărei parte teoretică a fost prezentată în capitolul 2. Acesta include patru elemente importante:
    a. Funcțiile de activare – În acest modul sunt implementate funcțiile de activare care sunt utilizate în rețeaua neuronală, cum ar fi sigmoida, tangenta hiperbolică și ReLU.
    b. Pasul Forward - În pasul forward, intrările sunt propagate prin rețea, iar rezultatul final este generat la stratul de ieșire.
    c. Pasul Backward - În pasul backward, se calculează eroarea produsă de rețea și pe baza acesteia, folosind funcția de gradient a funcției de eroare se actualizează ponderile și bias-urile rețelei.
    d. Regula Backpropagation – Aceasta constă în parcurgerea exemplelor de antrenament date ca intrare și realizarea pasului backward pentru întreaga rețea. În acest modul, se calculează eroarea dintre rezultatul obținut de rețeaua neuronală și rezultatul corect din exemplul, iar această eroare este propagată înapoi prin rețea pentru a actualiza ponderile.

3. Algoritmul Genetic
    a. Implementarea operatorilor – se implementează operatorii genetici care au fost prezentați în capitolul 2
    b. Funcția de fitness – se implementează funcția de fitness care va fi folosită în algoritmul genetic pentru a evalua cât de bine se potrivește un individ pentru problema dată

4. Rularea Programului – În acest modul, funcționalitățile și modulele dezvoltate anterior sunt integrate într-o aplicație funcțională. Clasele implementate corespund diferitelor ferestre sau ecrane ale programului, fiecare având propriile sale funcționalități. Pentru rularea programului se folosește o mașină cu stări finite pentru a gestiona tranzițiile între diferitele ferestre și stări ale programului.
    a. State Manager – Este modulul care se ocupa cu gestionarea stărilor diferite ale programului. State Manager este implementat ca o mașină cu stări finite, care controlează tranziția între stările programului.
    b. Options – Options este o stare în care utilizatorul poate selecta diferite opțiuni pentru configurarea algoritmul de antrenare folosit
    c. Train Using Genetic Algorithm – În acest modul se folosesc operatorii genetici, selecția, încrucișarea și mutația, pentru a rula algoritmul genetic. Pornind de la un set de indivizi, fiecare reprezentând o soluție posibilă (ponderile unei rețele neuronale), se generează noi

populații de soluții prin aplicarea operatorilor genetici. Fiecare individ este evaluat în funcție de cât de bine se potrivește scopului problemei de rezolvat, iar cei mai buni indivizi sunt selectați pentru genera indivizii generației următoare.

d. Train Using Backpropagation – acest modul permite antrenarea rețelei neuronale utilizând algoritmul de învățare backpropagation. Exemplele de antrenament sunt generate folosind mișcările utilizatorului. Rețeaua neuronală va folosi aceste exemple pentru a-și ajusta ponderile și bias-urile pentru a se comporta cât mai asemănător cu ieșirea exemplele de antrenament date.
e. Run Trained Network – Această stare permite utilizatorului să aleagă și să ruleze o rețea neurală care a fost antrenată folosind una din metodele de antrenament prezentate mai sus.

3.2 Implementarea Jocului

3.2.1 Clasa Snake

Se ocupă de gestionarea și salvarea datelor legate de șarpele care se află pe tabla de joc. Aceasta păstrează informații despre scorul obținut de șarpe, poziția segmentelor corpului șarpelui pe tabla de joc, direcția sa curentă și lungimea. Pe lângă atributele care sunt folosite pentru rularea jocului, clasa Snake mai reține și variabila *brain*, creierul șarpelui, care este o rețea neuronală care va fi folosită pentru predicția mișcării următoare a șarpelui, și variabila *fitness* care reține scorul fitness al șarpelui și este folosit de algoritmul genetic pentru selecția indivizilor.

Clasa Snake conține atributele:

- *score*: Este utilizat pentru a stoca scorul șarpelui în timpul jocului, scorul se incrementează în momentul în care șarpele mănâncă un măr. Este folosit în cadrul algoritmului genetic de către funcția de fitness pentru evaluarea unui individ.
- *fitness*: Este utilizat de algoritmul genetic pentru a evalua performanța șarpelui și pentru a decide care indivizi vor fi selectați pentru a trece prin procesul de reproducere.
- *brain*: Este rețeaua neuronală a șarpelui care va fi folosită pentru a obține predicția pentru mișcarea următoare pe care trebuie să o facă șarpele.
- *body*: Este o listă ce conține pozițiile segmentelor șarpelui de pe tabla de joc
- *TTL* (Time To Live): Este o variabilă care se folosește la antrenarea rețelei folosind algoritmul genetic pentru a limita numărul de mutări pe care șarpele le poate face înainte de a fi considerat "mort". De fiecare dată când șarpele face o mutare care nu ajunge la măr, valoarea TTL este decrementată. Dacă șarpele ajunge la o valoare TTL de 0, atunci este considerat mort și jocul

se încheie pentru acel șarpe. Atunci când șarpele mănâncă un măr, valoarea TTL este resetată la valoarea *max_TTL*. Scopul variabilei TTL este de a limita numărul de mișcări irosite pe care le poate face un șarpe în timpul rulării algoritmului genetic și de a evita apariția de șerpi care se mișcă într-o buclă infinită. Prin setarea unei valori prestabilite pentru TTL, se poate asigura că șerpii generați de algoritmul genetic vor fi eficienți în mișcările lor și nu vor ajunge într-un ciclu infinit de mutări.

- *max_TTL*: Variabila reține valoarea la care TTL-ul se va reseta în momentul în care șarpele mănâncă un măr. Variabila este inițializată în momentul în care se apelează constructorul clasei *Model* cu lungimea tablei de joc ridicată la pătrat.
- *steps_taken*: Numărul de pași pe care șarpele le-a făcut de la momentul în care a fost inițializat până la momentul în care a murit. Este folosit în cadrul algoritmului genetic de către funcția de fitness pentru evaluarea unui cromozom.
- *won*: Este o variabilă booleană care reține dacă șarpele a câștigat jocul înainte să moară
- *direction*: memorează direcția curentă în care se îndreaptă șarpele

```
class Snake:
    def __init__(self, neural_network: NeuralNetwork, max_ttl: int):
        self.brain = neural_network
        self.score = 0
        self.fitness = 0

        self.body: List = []
        self.TTL: int = GameSettings.SNAKE_MAX_TTL
        self.MAX_TTL: int = max_ttl
        self.steps_taken: int = 0
        self.won: bool = False
        self.past_direction: Direction = None
```

Pe lângă constructor, clasa Snake mai conține și funcțiile folosite pentru calculul scorului de fitness care vor fi prezentate în capitolul 3.5.2

3.2.2 Clasa Model

Model se ocupă de implementarea regulilor jocului de Snake, de generarea și modificarea tablei de joc, de poziționarea și mutarea șarpelui în interiorul tablei. Constructorul clasei *Model* este folosit pentru a construi tabla de joc și pentru a inițializa obiectul *Snake*.

```
class Model:
    def __init__(self, model_size: int, snake_size: int, net: NeuralNetwork):
        self.size: int = model_size + 2
        self.board: np.ndarray = np.full((self.size, self.size), BoardConsts.EMPTY)

        self.max_score = model_size ** 2 - snake_size
        self.snake_size: int = snake_size
```

```
        self.snake: Snake = Snake(net, model_size ** 2)

        self.make_board()
        self.create_random_snake()
        self.update_board_from_snake()
        self.place_new_apple()
```

Argumentele constructorului:

- *model_size*: Dimensiunea tablei de joc, această valoare nu tine cont de pereții de pe marginile tablei (când *model_size* este 10, tabla va fi 11 x 11 când se adaugă pereții în exterior)
- *snake_size*: Dimensiunea inițiala a șarpelui la începutul execuției programului
- *net*: Rețeaua neuronală care va fi folosită de către șarpe pentru inițializarea atributului *brain*, a rețelei neuronale a șarpelui

Atributele clasei:

- *board*: Reprezintă tabla de joc folosită, valorile tablei sunt stocate sub forma unui array de tip numpy. Fiecare element al jocului este reprezentat folosind un număr în această matrice.
- *max_score*: Reține scorul maxim posibil pe care șarpele îl poate obține pentru tabla curentă. Este folosită pentru a verifica dacă șarpele a câștigat jocul. Scorul maxim posibil este calculat ca fiind numărul de blocuri din care e compusă tabla de joc de la care se scade dimensiunea de început a șarpelui.
- *snake*: Reține obiectul *Snake* care se află pe tablă.

Funcția *make_board* este responsabilă pentru crearea tablei de joc. În interiorul funcției se folosesc instrucțiuni de tip list comprehension pentru a pune blocuri *EMPTY* în interiorul tablei și pentru a pune blocuri de tip *WALL* pe marginile tablei.

```
def make_board(self) -> None:
    self.board[1:-1, 1:-1] = BoardConsts.EMPTY
    self.board[0, :] = self.board[-1, :] = self.board[:, 0] = self.board[:, -1] =
BoardConsts.WALL
```

Clasa *BoardConsts* este utilizată pentru a defini și stoca constantele care reprezintă elementele ce pot fi plasate pe tabla jocului. Aceste constante sunt:

- *APPLE*: Reprezintă un măr, un obiect pe care șarpele îl poate mânca pentru a-și crește lungimea și scorul.
- *WALL*: Reprezintă un perete, un obstacol care nu poate fi trecut de șarpe și care limitează aria de mișcare a acestuia.
- *EMPTY*: Reprezintă un spațiu gol pe tabla jocului, pe care se poate muta șarpele.

- *SNAKE_BODY*: Reprezintă corpul șarpelui, adică segmentele din care este format șarpele
- *SNAKE_HEAD*: Reprezintă capul șarpelui, care este folosit ca originea din care se trasează liniile de viziune ale șarpelui

După ce sunt determinate pozițiile segmentelor șarpelui pe tablă folosind funcția *create_random_snake*, se apelează funcția *update_board_from_snake* pentru a modifica tabla folosind noile poziții ale segmentelor șarpelui.

O funcție importantă a clasei este funcția *move*. Această funcție realizează mutarea șarpelui pe tablă, utilizând direcția specificată ca parametru și întorcând o valoare *bool* care indică dacă șarpele a murit sau nu din cauza acestei mișcări. În interiorul funcției se realizează mutarea șarpelui pe tablă și se verifică dacă șarpele a lovit un perete sau un segment al corpului, dacă șarpele a mâncat un măr și dacă șarpele a reușit să câștige jocul prin mâncarea acelui măr.

Înainte de a realiza mutarea, se actualizează direcția de mișcare a șarpelui. Folosind noua direcție, se calculează poziția nouă a capului șarpelui, denumită *next_head*. Dacă șarpele a făcut o coliziune, funcția returnează valoarea *False*, semnificând că șarpele a murit. În caz contrar, se incrementează numărul de pași făcuți de șarpe și se inserează blocul *new_head* al șarpelui la începutul listei care conține segmentele șarpelui.

```
def move(self, new_direction: Direction) -> bool:
    self.snake.direction = new_direction

    head = self.snake.body[0]
    next_head = [head[0] + new_direction.value[0], head[1] + new_direction.value[1]]
    new_head_value = self.board[next_head[0]][next_head[1]]

    if (new_head_value == BoardConsts.WALL) or (new_head_value ==
BoardConsts.SNAKE_BODY):
        return False

    self.snake.body.insert(0, next_head)
    self.snake.steps_taken += 1

    if new_head_value == BoardConsts.APPLE:
        self.update_board_from_snake()
        self.snake.score = self.snake.score + 1



        if self.max_score == self.snake.score:
            self.snake.won = True
            return False

        self.place_new_apple()
        self.snake.TTL = GameSettings.SNAKE_MAX_TTL

    else:
        self.snake.body = self.snake.body[:-1]
```

```
            self.update_board_from_snake()
            self.snake.TTL = self.snake.TTL - 1

            if self.snake.TTL == 0:
                return False

        return True
```

În continuare, se verifică dacă valoarea blocului de la poziția nouă este un măr sau un spațiu gol. Dacă este un măr, poziția șarpelui pe tablă este actualizată iar scorul șarpelui este incrementat. După aceea se verifică dacă șarpele a câștigat jocul atunci când a mâncat acel măr. În caz afirmativ, atributul *won* al șarpelui este actualizat ca fiind *True*, iar funcția returnează valoarea *False* pentru a indica că jocul a fost încheiat. Dacă șarpele nu a câștigat când a mâncat mărul, se plasează un nou măr pe tablă și se resetează valoarea *TTL* a șarpelui la valoarea *max_ttl*. În cazul în care valoarea blocului de la *new_head* nu este un măr, adică șarpele a mers într-un bloc gol, atunci se elimină ultimul segment din corpului șarpelui și se modifica poziția șarpelui pe tablă folosind *update_board_from_snake*. Acest lucru simulează mișcarea șarpelui prin adăugarea unui segment la începutul corpului și eliminarea ultimului segment de la coadă. De asemenea se decrementează contorul *TTL*. Dacă contorul a ajuns la zero înseamnă că șarpele a făcut prea multe mișcări inutile și se consideră că a murit, iar funcția întoarce *False*.

La sfârșitul funcției, dacă șarpele nu a îndeplinit nici o condiție de moarte, se întoarce *True* pentru a specifică că șarpele este încă în viață.

Restul funcțiilor care sunt implementate în clasa *Model* sunt:

- *get_random_empty_block*: Creează o listă ce conține toate blocurile goale din interiorul tablei de joc (blocuri EMPTY, care nu conțin segmente ale corpului șarpelui, măr sau perete) și returnează un bloc aleator din interiorul acestei liste.
- *place_new_apple*: Folosește *get_random_empty_block* pentru a alege aleator un bloc gol de pe tablă pe care se va pune un măr. Funcția este folosită la începutul jocului când se pune primul măr pe tablă sau atunci când se înlocuiește un măr care a fost mâncat de șarpe.
- *create_random_snake*: Generează o listă ce conține poziții aleatoare pentru segmentele corpului șarpelui. Atunci când se generează un șarpe nou se alege aleator un bloc gol de pe tabla pe care se va pune capul șarpelui, iar blocurile care vor forma corpul șarpelui se aleg din blocurile vecine pe care le are ultimul bloc din coada șarpelui care sunt goale. Când se adaugă un bloc nou în corpului șarpelui, acel bloc va fi ultimul bloc din coada șarpelui.

- *update_board_from_snake*: Actualizează valorile matricei *board* cu noile poziții ale segmentelor corpului șarpelui atunci când șarpele realizează o mișcare.

3.2.3 Direcțiile jocului

Programul se folosește de direcțiile care sunt implementate în clasa *Direction* pentru a trasa liniile de viziune ale șarpelui. Se folosește o clasă de tip *Enum* pentru a crea o enumerație cu 8 direcții posibile, adică cele patru direcții cardinale (UP, DOWN, LEFT, RIGHT) și cele patru diagonale (Q1, Q2, Q3, Q4).

```
class Direction(Enum):
    UP = [-1, 0]
    DOWN = [1, 0]
    LEFT = [0, -1]
    RIGHT = [0, 1]
    Q1 = [-1, 1]
    Q2 = [1, 1]
    Q3 = [1, -1]
    Q4 = [-1, -1]
```

Fiecare direcție este reprezentată ca o listă de două valori, fiecare listă conține offset-urile care sunt aplicate poziției curente pentru a realiza mișcarea în acea direcție pe matricea tablei de joc. Lista de valori conține schimbările pe i și j necesare pentru ca șarpele să ajungă la blocul următor din acea direcție. De exemplu, direcția *UP* are valoarea [-1, 0], deoarece mutarea șarpelui în sus implică o scădere cu o unitate a coordonatei i și o menținere a coordonatei j.

Aceste direcții sunt utilizate pentru a actualiza poziția capului șarpelui în timpul mișcării și pentru a calcula pozițiile următoarelor segmente ale corpului. De asemenea, direcțiile sunt folosite pentru a obține liniile de viziune ale șarpelui.

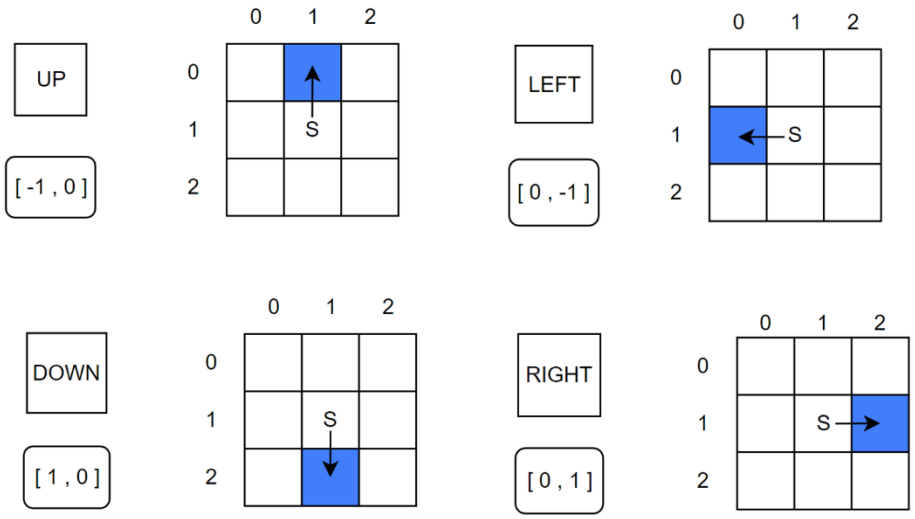

Figura 3.2 Valorile distanțelor cardinale

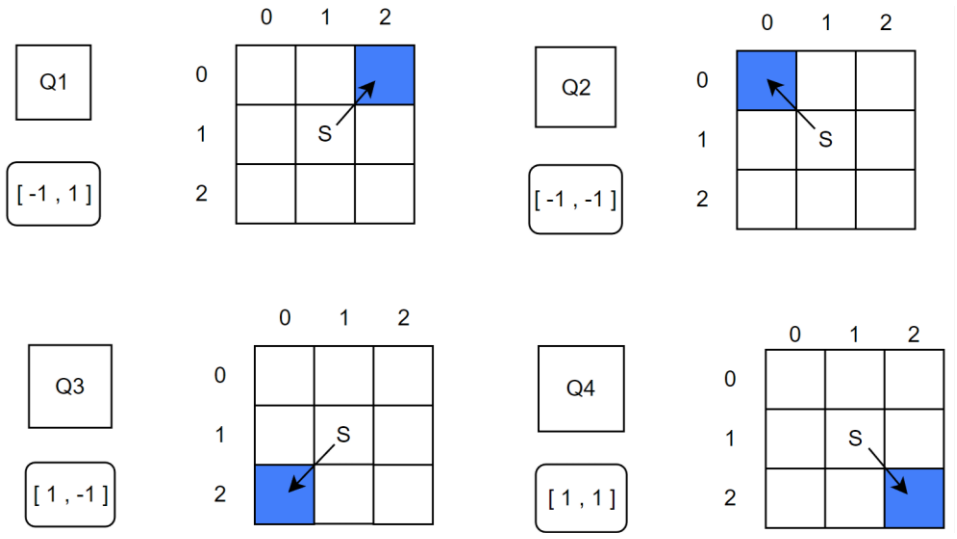

Figura 3.3 Valorile distanțelor diagonale

3.2.4 Liniile de viziune

Liniile de viziune sunt folosite pentru a reprezenta percepția șarpelui asupra mediului, adică asupra tablei de joc. Sunt linii care se extind din capul șarpelui în mai multe direcții până când ating un perete al tablei.

O linie de viziune va reține:

- distanța și coordonata de la capul șarpelui până la peretele tablei de joc
- distanța și coordonata până la primul segment al corpului sau o valoare booleană
- distanța și coordonata până la măr sau o valoare booleană

Pentru intrarea în rețeaua neuronală nu se vor folosi coordonatele obținute de linia de viziune, coordonatele sunt folosite doar pentru afișarea pe ecran a liniilor de viziune.

Distanța este numărul de pătrate de la capul șarpelui până la un segment al corpului sau măr dacă ele sunt prezente pe acea direcție, sau 0 dacă nu se află pe acea direcție. Funcția de distanță care este folosită pentru a obține numărul de pătrate dintre capul șarpelui și blocurile de pe linia de viziune este distanța Manhattan.

Distanța Manhattan este folosită deoarece în program șarpele se poate mișca direct doar în direcțiile cardinale, deci pentru mișcarea șarpelui la un bloc în sus, jos, stânga sau dreapta șarpele trebuie să facă doar o singura mișcare. Atunci când șarpele trebuie să se miște pe diagonală el trebuie să realizeze două mișcări succesive. După cum a fost prezentat în capitolul 2.5, distanța Manhattan atribuie blocurilor vecine care înconjoară un bloc origine valoarea de 1 dacă ele se află pe direcțiile cardinale și valoarea de 2 dacă ele se află pe diagonale, așadar distanța Manhattan

calculează corect numărul de mișcări pe care șarpele trebuie să le facă în program pentru a ajunge de la poziția de start la un bloc de pe tablă.

Un exemplu al distanțelor obținute folosind distanța Manhattan este prezentat în figura 3.4. În această figura dimensiunea tablei este de 12 x 12 iar originea de la care se calculează distanțele se află pe linia 5 și coloana 5.

```
[10  9  8  7  6  5  6  7  8  9 10 11]
[ 9  8  7  6  5  4  5  6  7  8  9 10]
[ 8  7  6  5  4  3  4  5  6  7  8  9]
[ 7  6  5  4  3  2  3  4  5  6  7  8]
[ 6  5  4  3  2  1  2  3  4  5  6  7]
[ 5  4  3  2  1  0  1  2  3  4  5  6]
[ 6  5  4  3  2  1  2  3  4  5  6  7]
[ 7  6  5  4  3  2  3  4  5  6  7  8]
[ 8  7  6  5  4  3  4  5  6  7  8  9]
[ 9  8  7  6  5  4  5  6  7  8  9 10]
[10  9  8  7  6  5  6  7  8  9 10 11]
[11 10  9  8  7  6  7  8  9 10 11 12]
```

Figura 3.4 – Exemplu de distanțe obținute pentru o tablă de 12x12 folosind distanța Manhattan

Pentru normalizarea distanțelor obținute, linia de viziune va memora inversul distanțelor obținute, adică 1/distanță. Această normalizare asigură că distanțele mai mici vor avea valori mai mari, în timp ce distanțele mai mari vor avea valori mai mici, astfel blocurile mai apropiate de capul șarpelui vor avea o importanță mai mare decât cele îndepărtate. Prin memorarea inverselor distanțelor, se realizează o scalare astfel încât toate valorile să fie în același interval de (0, 1], lucru care va ajuta rețeaua neuronală să învețe mai bine.

Distanțele obținute în urma normalizării sunt prezentate în figura 3.5. După cum se poate observa în figură, blocurile care sunt mai apropiate de capul șarpelui au valori mai mari decât blocurile îndepărtate

```
[0.1   0.111 0.125 0.143 0.167 0.2   0.167 0.143 0.125 0.111 0.1   0.091]
[0.111 0.125 0.143 0.167 0.2   0.25  0.2   0.167 0.143 0.125 0.111 0.1  ]
[0.125 0.143 0.167 0.2   0.25  0.333 0.25  0.2   0.167 0.143 0.125 0.111]
[0.143 0.167 0.2   0.25  0.333 0.5   0.333 0.25  0.2   0.167 0.143 0.125]
[0.167 0.2   0.25  0.333 0.5   1.    0.5   0.333 0.25  0.2   0.167 0.143]
[0.2   0.25  0.333 0.5   1.    0.    1.    0.5   0.333 0.25  0.2   0.167]
[0.167 0.2   0.25  0.333 0.5   1.    0.5   0.333 0.25  0.2   0.167 0.143]
[0.143 0.167 0.2   0.25  0.333 0.5   0.333 0.25  0.2   0.167 0.143 0.125]
[0.125 0.143 0.167 0.2   0.25  0.333 0.25  0.2   0.167 0.143 0.125 0.111]
[0.111 0.125 0.143 0.167 0.2   0.25  0.2   0.167 0.143 0.125 0.111 0.1  ]
[0.1   0.111 0.125 0.143 0.167 0.2   0.167 0.143 0.125 0.111 0.1   0.091]
[0.091 0.1   0.111 0.125 0.143 0.167 0.143 0.125 0.111 0.1   0.091 0.083]
```

Figura 3.5 – Valorile distanțelor obținute în urma normalizării distanțelor pentru tabla de 12x12

O altă opțiune care poate fi folosită pentru a memora prezenta mărului sau segmentului pe o direcție este valoarea booleană care indică existența sau lipsa a unui segment sau a unui măr într-o anumită direcție. Valoarea booleană memorează 1 dacă un segment al corpului sau un măr este prezent pe acea direcție, sau 0 dacă nu este prezent.

Se pot folosi opțiuni diferite pentru liniile de viziune utilizate în joc. O opțiune este să se folosească doar liniile de viziune pentru direcțiile cardinale (UP, DOWN, LEFT, RIGHT), sau să se includă și liniile de viziune pentru diagonale. De asemenea, se poate alege tipul de informație care va fi folosit pentru a reține prezenta unui măr sau segment al corpului pentru linia de viziune. De exemplu, se poate alege ca liniile de viziune să întoarcă distanța de la capul șarpelui la măr sau la un segment, sau se poate opta pentru o valoare booleana, care indica prezenta segmentul sau mărul pe acea linie de viziune.

```
class VisionLine:
    def __init__(self, wall_coord, wall_distance, apple_coord, apple_distance,
segment_coord, segment_distance, direction: Direction):
        self.wall_coord = wall_coord
        self.wall_distance = wall_distance
        self.apple_coord = apple_coord
        self.apple_distance = apple_distance
        self.segment_coord = segment_coord
        self.segment_distance = segment_distance
        self.direction = direction
```

Folosind funcția *get_vision_lines* se poate obține o listă cu liniile de viziune ale șarpelui pe tabla de joc. Această funcție primește ca parametrii tabla de joc, poziția capului șarpelui pe tablă, numărul de direcții în care se va uita șarpele și tipul de informație care va fi reținut pentru măr și segment al corpului de către liniile de viziune.

```
def get_vision_lines (board: np.ndarray, snake_head, vision_direction_count: int,
apple_return_type: str, segment_return_type: str) -> List[VisionLine]:
    distance_function = manhattan_distance
    directions = [Direction.UP, Direction.DOWN, Direction.LEFT, Direction.RIGHT]
    if vision_direction_count == 8:
        directions += [Direction.Q1, Direction.Q2, Direction.Q3, Direction.Q4]

    vision_lines = []
    for direction in directions:
        apple_coord = None
        segment_coord = None

        current_block = [snake_head[0] + direction.value[0], snake_head[1] +
direction.value[1]]

        while board[current_block[0]][current_block[1]] != BoardConsts.WALL:
            if board[current_block[0]][current_block[1]] == BoardConsts.APPLE and
apple_coord is None:
                apple_coord = current_block
```

```
            elif board[current_block[0]][current_block[1]] == BoardConsts.SNAKE_BODY
and segment_coord is None:
                segment_coord = current_block
            current_block = [current_block[0] + direction.value[0], current_block[1] +
direction.value[1]]

        wall_coord = current_block
        wall_distance = distance_function(snake_head, wall_coord)
        wall_output = 1 / wall_distance

        if apple_return_type == "boolean":
            apple_output = 1.0 if apple_coord is not None else 0.0
        else:
            apple_output = 1.0 / distance_function(snake_head, apple_coord) if
apple_coord is not None else 0.0

        if segment_return_type == "boolean":
            segment_output = 1.0 if segment_coord is not None else 0.0
        else:
            segment_output = 1.0 / distance_function(snake_head, segment_coord) if
segment_coord is not None else 0.0

        vision_lines.append(VisionLine(wall_coord, wall_output, apple_coord,
apple_output, segment_coord, segment_output, direction))
    return vision_lines
```

Pentru a determina direcțiile în care se va uita șarpele, se utilizează o listă de direcții care conține inițial direcțiile cardinale. Dacă utilizatorul alege opțiunea pentru 8 direcții, atunci se adaugă și direcțiile diagonale la listă.

În interiorul funcției se parcurge tabla de joc începând de la poziția capului șarpelui în direcția specificată, memorându-se coordonatele primului segment și primului măr întâlnite. Aceste coordonate sunt utilizate apoi pentru a calcula distanțele, dacă tipul de valoare întoarsă specificat de utilizator este distanța, altfel se va întoarce o variabila booleana.

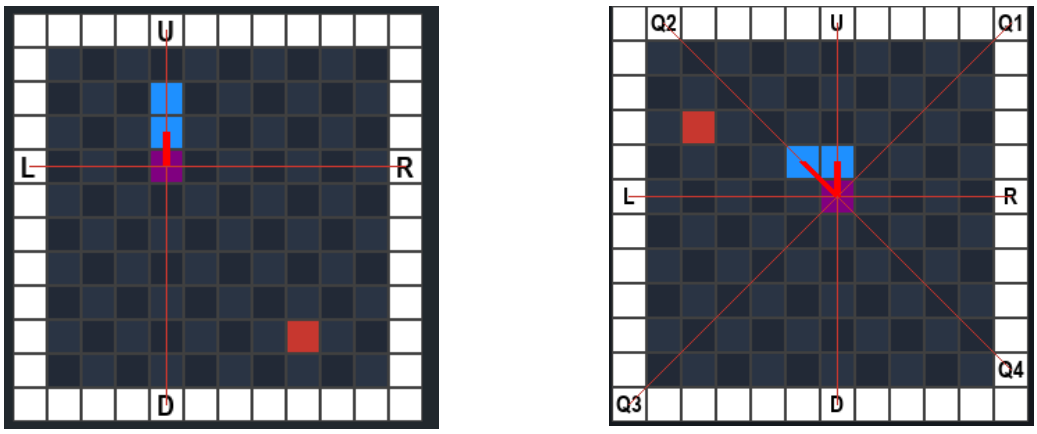

Figura – 3.6 Rularea jocului cu 4 linii de viziune și cu 8 linii de viziune

În tabelele următoare se prezintă valorile liniilor de viziune obținute pentru tabelele prezentate mai sus. Pentru informația reținuta de liniile de viziune s-a folosit distanța pentru corpul șarpelui și boolean pentru prezenta mărului pe acea direcție.

Direction	Wall Value	Apple Value	Segment Value
UP	0.25	0.00	1.00
DOWN	0.1428571	0.00	0.00
LEFT	0.25	0.00	0.00
RIGHT	0.1428571	0.00	0.00

Figura 3.7 – Valorile liniilor de viziune pentru exemplul care folosește 4 linii de viziune

Direction	Wall Value	Apple Value	Segment Value
UP	0.25	0.00	1.00
DOWN	0.1428571	0.00	0.00
LEFT	0.25	0.00	0.00
RIGHT	0.1428571	0.00	0.00
Q1	0.1	0.00	0.00
Q2	0.1	0.00	0.50
Q3	0.8333333	0.00	0.00
Q4	0.1	0.00	0.00

Figura 3.8 – Valorile liniilor de viziune pentru exemplul care folosește 8 linii de viziune

3.3 Îmbunătățirea vitezei de execuție pentru program

Cython este un limbaj de programare care extinde limbajul Python combinând sintaxa și semantica codului Python cu performanța limbajului C/C++. Cython a fost creat pentru a permite dezvoltatorilor să scrie cod Python care să poată fi compilat în cod C și executat direct fără a necesita interpretarea de către interpretorul Python, obținând astfel un câștig semnificativ de performanță.

Pentru a compila și executa codul Cython, este necesară instalarea compilatorului C pe sistemul de operare, crearea unui fișier de tipul .pyx (fișier sursă Cython), rescrierea funcțiilor de Python folosind sintaxa Cython și compilarea codului. Codul Cython este compilat în cod C sau C++, după care este modificat pentru a produce un modul care poate să fie folosit în codul Python normal.

Câștigul de performanță adus de Cython este obținut prin declararea statica a variabilelor care se folosesc, dezactivarea verificărilor erorilor cum ar fi oprirea verificări de împărțire cu zero sau dezactivarea erorii de depășire a indexului unei matrice. Limbajul Cython are acces la funcțiile care provin din modulele de Python, dar pentru a crește câștigul de performanță și mai mult se

poate eliminare folosirea funcțiilor care provin din aceste module, cum ar fi funcțiile care provin din modulul *numpy*.

Pentru a îmbunătăți viteza de execuție a programului s-au rescris în limbajul Cython funcțiile programului care sunt apelate des și cele care consumă foarte mult timp.
Funcțiile care au fost rescrise în limbajul Cython sunt:

- *get_random_empty_block*
- *update_board_from_snake*
- *manhattan_distance*
- *get_vision_lines*

De exemplu, prin folosirea variantei Cython a funcției *get_vision_lines* timpul de execuție al funcției este de 5 ori mai rapid decât varianta prezentată în capitolul trecut care era implementat folosind Python pur.

3.4 Implementarea Rețelei Neuronale – Clasa Dense si Clasa Activation

Pentru implementarea rețelei neuronale se va împărții un strat al rețelei neuronale în 2 clase pentru a simplifica codul rețelei neuronale: clasa *Dense* și clasa *Activation*.

Clasa *Dense* se ocupă cu calculele pentru obținerea valorilor neuronilor, în interiorul acestei clase se aplică înmulțirea ponderilor rețelei cu semnalele de intrare pentru neuroni. Clasa *Activation* se ocupă cu aplicarea funcției de activare asupra valorilor calculate de către clasa Dense.

Ambele clase implementează aceleași funcții, adică funcțiile *forward* și *backward* care for fi folosite pentru pasul forward și respectiv pasul backward.

Structura rețelei neuronale care a fost prezentată în capitolul 2 devine acum asemănătoare cu figura prezentata mai jos.

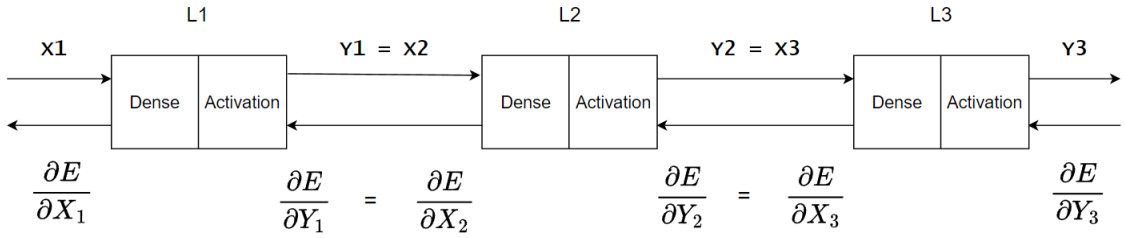

Figura 3.9 – Propagarea datelor în rețeaua neuronala care folosește clasele Dense și Activation

3.4.1 Arhitectura rețelei neuronale

Intrarea rețelei neuronale va fi formată din distanțele obținute de liniile de viziune și încă 2 variabile care sunt folosite pentru a memora offset-urile direcției anterioare în care s-a mișcat șarpele.

Atunci când utilizatorul alege opțiunea ca șarpele să proiecteze liniile de viziune doar pe direcțiile cardinale stratul de intrare al rețelei va avea 4 * 3 + 2 neuroni, adică 14 neuroni în total, deoarece vor fi 4 linii de viziune fiecare cu cate 3 valori ale distanței pentru perete zid și măr, și 2 neuroni care rețin offset-urile direcției anterioare ale șarpelui, iar pe stratul ascuns vor fi 18 neuroni.

Dacă se folosesc direcțiile cardinale și diagonalele pentru liniile de viziune atunci stratul de intrare va avea 8 * 3 + 2 neuroni, adică 26 neuroni, iar pe stratul ascuns vor fi 20 de neuroni.

Numărul de neuroni care vor fi folosiți pentru stratul ascuns poate să fie schimbat de către utilizator înainte de a începe rularea algoritmului.

Stratul de ieșire al rețelei va avea întotdeauna același și numărul de neuroni, va avea 4 neuroni în care rețeaua neuronală va calcula predicția pentru direcția următoare pe care șarpele trebuie să o ia (4 direcții posibile: UP, DOWN, LEFT, RIGHT). Direcția aleasă de către program este aceea care are valoarea cea mai mare din valorile neuronilor.

Un exemplu de configurație pentru rețeaua neuronală este prezentat în figura 3.10. Această rețea folosește doar direcțiile cardinale pentru a obține intrările și va folosi distanța pentru a memora prezenta mărului și segmentului pe o linie de viziune

		0.89152448		
		0.		
UP WALL	0.5	0.62310582		
UP APPLE	0.	0.		
UP SEGMENT	0.	2.14548522		
DOWN WALL	0.11111111	0.		
DOWN APPLE	0.	0.		
DOWN SEGMENT	0.	0.	0.45188645	UP
LEFT WALL	0.5	1.65992792	0.43922582	DOWN
LEFT APPLE	0.	1.26355896	0.13225657	LEFT
LEFT SEGMENT	1.	0.55835483	0.74699737	RIGHT
RIGHT WALL	0.11111111	0.77538532		
RIGHT APPLE	1.	0.		
RIGHT SEGMENT	0.	1.36016603		
OFFSET X	0.	0.		
OFFSET Y	1.	0.		
		0.		
		0.		

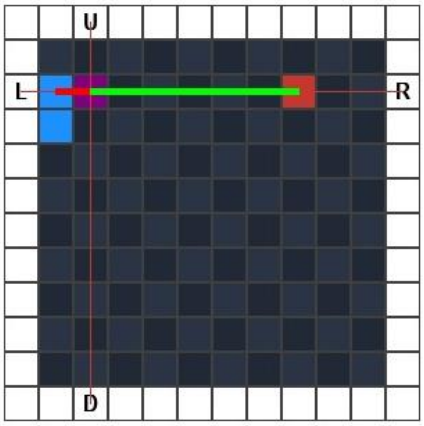

Figura 3.10 – Exemplu de rețea neuronală ce folosește 4 direcții pentru liniile de viziune

3.4.2 Implementarea funcțiilor de activare

Funcțiile de activare folosite sunt funcțiile care au fost prezentate în capitolul 2. De asemenea, pentru fiecare funcție de activare care se folosește s-a implementat și derivata funcției pentru ca funcția să poată fi folosită în faza de antrenare folosind regula backpropagation.

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def sigmoid_prime(x):
    aux = sigmoid(x)
    return aux * (1 - aux)

def tanh(x):
    return np.tanh(x)
def tanh_prime(x):
    return 1 - np.tanh(x) ** 2

def relu(x):
    return np.maximum(0.0, x)
def relu_prime(x):
    return np.where(x <= 0, 0.0, 1.0)
```

3.4.3 Implementarea funcțiilor de eroare

Se implementează funcția de eroare și derivata funcției de eroare prezentată în capitolul 2. Funcțiile de eroare calculează diferența dintre ieșirea dorită de către rețeaua neuronală care se află în exemplele de antrenament și ieșirea obținută de către rețeaua neuronală care nu a fost antrenată. Funcțiile de eroare implementate vor fi folosite doar când se realizează antrenarea rețelei folosind regula backpropagation.

Parametrii funcțiilor *mse* și *mse_prime* sunt *target_y*, care este un vector ce conține valorile țintă ale rețelei neuronale preluate din exemplele de antrenament, și *predicted_y* care este vectorul ce conține valorile reale prezise de rețeaua neuronală care se antrenează în timpul procesului de antrenament ce folosește regula backpropagation.

```
def mse(target_y, predicted_y):
    return np.mean(np.power(target_y - predicted_y, 2))

def mse_prime(target_y, predicted_y):
    return (2 / np.size(target_y)) * (predicted_y - target_y)
```

3.4.4 Clasa de bază pentru straturile rețelei neuronale

Clasa *Layer* este clasa de bază a rețelei neuronale, care este moștenită de clasele *Dense* și *Activation*. Din moment ce clasa *Layer* definește metodele *forward* și *backward* fără a le implementa, aceasta poate fi considerată o clasă abstractă. Implementarea acestor metode va fi realizată de către clasele moștenitoare. Clasa *Layer* a fost folosită pentru a obține polimorfism în cadrul clasei *NeuralNetwork*, care conține o listă de obiecte de tip *Dense* și *Activation*.

```
class Layer:
    def __init__(self):
        self.inputs: int
        self.output = None

    def forward(self, inputs):
        pass

    def backward(self, output_gradient, learning_rate):
        pass
```

3.4.5 Constructorii claselor Dense și Activation

Constructorul clasei Dense primește ca parametrii numărul neuronilor din stratul curent și numărul neuronilor din stratul următor, care sunt stocați în atributele clasei pentru a putea fi utilizați în calculele ulterioare. De asemenea, matricele de ponderi și bias-uri ale stratului sunt inițializate cu valori aleatoare mici folosind funcția *random.uniform*. Mai exact, se generează o matrice cu *output_size* linii și *input_size* coloane, care conține valori uniform distribuite din intervalul [-1, 1).

```
class Dense(Layer):
    def __init__(self, input_size, output_size):
        super().__init__()
        self.input = None
        self.input_size = input_size
        self.output_size = output_size
        self.weights = np.random.uniform(-1, 1, (output_size, input_size))
        self.bias = np.random.uniform(-1, 1, (output_size, 1))
```

Constructorul clasei *Activation* primește ca parametrii funcția de activare și derivata funcției de activare. Aceste funcții sunt esențiale în calculele efectuate de rețeaua neuronală. Funcția de activare folosită este aplicată la ieșirile fiecărui strat din rețea când se folosește pasul forward. Derivata funcției de activare este necesara în timpul algoritmului de antrenare al rețelei neuronale și este folosită pentru a actualiza ponderile și bias-urile neuronilor în funcție de erorile produse în timpul antrenării folosind regula backpropagation.

```
class Activation(Layer):
    def __init__(self, activation, activation_prime):
        super().__init__()
        self.input = None
        self.activation = activation
        self.activation_prime = activation_prime
```

3.4.6 Pasul Forward

Pasul forward al rețelei neuronale este folosit pentru a prezice direcția următoare în care trebuie să se deplaseze șarpele în funcție de valorile care sunt primite la intrarea rețelei. Pasul forward este folosit la:

1. Antrenarea rețelei folosind Algoritmul Genetic: La evaluarea unui individ al rețelei neuronale prin mutarea acestuia pe tablă folosind direcția prezisa de rețeaua neuronală și memorarea scorului si numărului de pași care au fost realizați înainte ca acesta sa moară.
2. Antrenarea rețelei folosind Regula Backpropagation: Pentru a obține eroarea dintre valoarea prezisă de rețeaua neuronală care nu a fost antrenată și ieșirea care a fost înregistrată în exemplele de antrenament generate de utilizator.
3. Rularea unei rețele neuronale antrenate: Pentru a realiza mișcarea șarpelui pe tablă pentru rețeaua neuronală care a fost obținută după procesul de antrenament și care a fost aleasă de utilizator

Formulele pentru pasul forward sunt implementate în metodele *forward* care se află în clasa *Dense* și *Activation*.

Metoda *forward* a clasei *Dense* realizează produsul scalar dintre matricea ponderilor și input-urile primite, la care se adaugă matricea bias-urilor.

```
class Dense(Layer):
def forward(self, input):
    self.input = input
    return np.dot(self.weights, self.input) + self.bias
```

Metoda *forward* a clasei *Activation* aplică input-urilor primite funcția de activare care a fost primită ca parametru în constructor.

```
class Activation(Layer):
def forward(self, inputs):
    self.input = inputs
    self.output = self.activation(inputs)
    return self.output
```

3.4.7 Pasul Backward

Pasul backward este folosit doar când se folosește regula backpropagation pentru antrenarea rețelei neuronale, mai precis în funcția *train* a rețelei neuronale care folosește funcțiile *backward* ale straturilor *Dense* și *Activation*

În clasa *Activation* metoda *backward* primește ca parametru gradientul erorii și rata de învățare. Rata de învățare primită ca parametru este folosită pentru a scala gradientul obținut în funcție de cât de repede se dorește să se modifice ponderile și bias-urile în timpul antrenamentului

Metoda *backward* calculează derivata erorii față de intrare folosind regula lanțului și derivata funcției de activare. Pentru calcul se înmulțește *output_gradient*, care este derivata erorii față de ieșire, cu valorile obținute de derivata funcției de activare folosind intrările stratului obținute de la pasul Forward

```
class Activation(Layer):
```

```
def backward(self, output_gradient, learning_rate):
    input_gradient = output_gradient * self.activation_prime(self.input)
    return input_gradient
```

Pentru stratul *Dense*, metoda *backward* primește aceiași parametrii ca la clasa *Activation*: gradientul erorii și rata de învățare. Metoda *backward* calculează derivata erorii față de intrare și derivata erorii față de ponderi. În același timp, valorile ponderilor și bias-urile stratului sunt actualizate utilizând folosind gradientul calculat.

Derivata erorii față de intrări este calculată prin produsul scalar între gradientul erorii de la stratul precedent și transpusa matricei ponderilor curente. Aceasta derivată va fi transmisă către stratul următor, ca input pentru calculul gradienților acestuia în cadrul metodei *backward*.

```
class Dense(Layer):
def backward(self, output_gradient, learning_rate):
    input_gradient = np.dot(self.weights.T, output_gradient)
    weights_gradient = np.dot(output_gradient, self.input.T)

    self.weights -= learning_rate * weights_gradient
    self.bias -= learning_rate * output_gradient

    return input_gradient
```

3.4.8 Clasa NeuralNetwork

Clasa *NeuralNetwork* este folosită pentru a construi o rețea neuronală completă, care conține mai multe straturi *Dense* și *Activation*.

Metoda *feed_forward* a clasei *NeuralNetwork* realizează pasul Forward pentru întreaga rețea prin parcurgerea fiecărui strat din lista de straturi ale rețelei și apelarea metodei *forward* a acelui strat. În timpul acestei etape, intrarea în stratul următor din rețea va fi rezultatul obținut de stratul curent. După ce au fost parcurse toate straturile rețelei, rezultatul final al pasului forward va fi predicția rețelei neuronale.

```
def feed_forward(self, input) -> np.ndarray:
    nn_input = input
    for layer in self.layers:
        nn_input = layer.forward(nn_input)
    return nn_input
```

3.4.9 Implementarea algoritmului de antrenament Backpropagation

Pentru această metoda de antrenare programul are nevoie de exemple de antrenament care sunt generate de către utilizator. Pentru generarea exemplelor de antrenament utilizatorul va se va juca jocul și programul va memora starea tablei și mișcările făcute de utilizator. În momentul în care utilizatorul decide să nu mai genereze exemple de antrenament, va avea loc antrenarea rețelei folosind algoritmul backpropagation.

Un exemplu de antrenament conține starea tablei (valorile liniilor de viziune și offset-urile direcției anterioare a șarpelui) și mișcarea realizată de utilizator pentru acea stare. Exemplele de antrenament sunt memorate de program folosind obiecte de tipul *TrainingExample*.

```
class TrainingExample:
    def __init__(self, past_direction: Direction, vision_lines: List[VisionLine],
user_move: List[float]):
        self.past_direction = current_direction
        self.vision_lines = vision_lines
        self.user_move = user_move
```

Metoda *train* a clasei *NeuralNetwork* este folosită pentru a realiza antrenarea rețelei neuronale folosind Regula Backpropagation. Parametrii funcției sunt:

- *loss*: Este funcția de eroare care va fi folosită pentru calculul erorii între valoarea prezisă de rețea neuronală neantrenată și mișcarea realizată de utilizator din exemplele de antrenament
- *loss_prime*: Este derivata funcției de eroare care va fi folosită pentru a calcula gradientul erorii.
- *x_train*: Un vector ce conține intrările rețelei neuronale care au fost stocate în exemplele de antrenament
- *y_train*: Un vector conține mișcările care au fost realizate de către utilizator pentru datele de intrare memorate.
- *learning_rate*: O valoare ce determină cu cât se actualizează ponderile și bias-urile în timpul pasului *backward* al antrenării.

Pentru a realiza antrenarea rețelei, se va lua pe rând fiecare exemplu de antrenament și se va folosi pasul forward al rețelei neantrenate pentru a obține predicția rețelei folosind datele de intrare care au fost memorate în acel exemplu de antrenament. Folosind funcția de pierdere care a fost primită ca parametru se va calcula eroarea dintre predicția obținută de rețeaua neantrenată și mișcarea care a fost realizată de către utilizator, și folosind derivata funcției de pierdere se va calcula gradientul erorii dintre predicția rețelei neantrenate și mișcarea utilizatorului din exemplul de antrenament. Folosind gradientul erorii se va realiza pasul backward pentru fiecare strat al rețelei neuronale în ordine inversă pentru a se realiza actualizarea ponderilor.

După parcurgerea fiecărui exemplu de antrenament se incrementează numărul de epoci și se verifică condiția de oprire. Dacă condiția de oprire nu a fost realizată, atunci se va lua din nou pe rând fiecare exemplu de antrenament pentru modificarea ponderilor rețelei. Antrenarea se va opri atunci când eroarea rețelei este mai mică decât 0.5, dacă suma tuturor erorilor obținute pentru exemplele de antrenament este mai mică decât 0.5, sau dacă limita de epoci care a fost primită ca parametru a fost atinsă, limita maximă de epoci pe care algoritmul va rula este de 10 ori numărul de exemple de antrenament primite.

```
def train(self, loss, loss_prime, x_train, y_train, learning_rate) -> None:
    error = 10000
    epoch_limit = 10 * x_train
    epoch = 0
    while epoch < epoch_limit and error > 0.1:
        error = 0
        for x, y in zip(x_train, y_train):
            output = self.feed_forward(x)
            error += loss(y, output)
            gradient = loss_prime(y, output)
            for layer in reversed(self.layers):
                gradient = layer.backward(gradient, learning_rate)
        epoch += 1
        if epoch % 100 == 0:
            print(f"epoch = {epoch}, error = {error}")
    print(f"final error {error}  \n")
```

3.5 Implementarea Algoritmului Genetic

Antrenarea folosind algoritmul genetic va fi formată din două parți:

1. În prima parte se va realiza evaluarea ponderilor rețelei neuronale ale șarpelui prin mutarea șarpelui pe tabla folosind direcțiile prezise de pasul Forward al rețelei sale neuronale până când acesta moare. În timp ce șarpele se mișcă pe tablă, se va reține numărul de pași și numărul de mere care au fost mâncate de către șarpe pentru a putea fi folosite pentru evaluare de către funcția de fitness. În momentul în care un șarpe moare se va folosi funcția de fitness pentru a calcula scorul de fitness al cromozomului folosind pașii și scorul obținuți de către șarpe. Când toți șerpii din populații au murit se trece la partea a două a algoritmului,
2. În partea a doua se aplică operatorii genetici implementați în capitolul 3.5 pentru a obține următoarea generație a algoritmului genetic.

Pașii I și II se vor executa până când utilizatorul alege să oprească antrenarea, moment în care rețelele neuronale obținute de algoritm vor fi salvate în fișiere json pentru a putea fi rulate și testate de către utilizator.

În prima generație se creează un număr de cromozomi care sunt inițializați cu ponderi aleatoare. Cromozomii generațiilor următoare vor fi obținuți prin aplicarea operatorilor genetici asupra indivizilor din generația curentă.

3.5.1 Reprezentarea unui cromozom

Structura unui cromozom este compusă din totalitatea matricelor de ponderi și bias-uri ale rețelei neuronale. Matricea folosită pentru memorarea ponderilor este de tipul *np.ndarray* care reține valori flotante. În figura 3.12 se prezintă un exemplu de matrice de ponderi pentru un strat

Dense al rețelei neuronale care este format din 10 neuroni de intrare și 10 neuroni de ieșire și care a fost inițializată cu valori aleatorii.

```
[-0.3746  0.1767 -0.4483  0.163  -0.8618 -0.2664  0.3931 -0.5893  0.6328 -0.3214]
[ 0.8274 -0.6012 -0.2126 -0.9157 -0.7028  0.8904  0.4448 -0.61   -0.1446 -0.4758]
[ 0.3262  0.8023 -0.1174 -0.9971  0.8566 -0.6588  0.7072 -0.6068 -0.8809  0.5773]
[ 0.1706  0.528  -0.2245  0.2379 -0.2716 -0.4106  0.5081  0.8346  0.8668 -0.8298]
[ 0.0076 -0.0558 -0.2434  0.3841 -0.9288  0.9748 -0.2834 -0.5543  0.3257  0.8376]
[-0.5563  0.5187  0.1935 -0.7242  0.1253  0.4183 -0.6665  0.7716  0.3147 -0.7464]
[-0.0716  0.338  -0.9096  0.5545 -0.8292  0.9243 -0.0309 -0.154   0.4813 -0.5587]
[-0.7311 -0.6851  0.6929  0.1211  0.6116  0.3789 -0.9102  0.8895  0.2037  0.4743]
[ 0.0202  0.5456  0.6552 -0.471   0.7242 -0.2153 -0.8576  0.0754 -0.8187 -0.0559]
[ 0.7373  0.8308 -0.9887  0.6482  0.3615 -0.002   0.7508  0.6505 -0.2355 -0.4908]
```

Figura 3.11 – Valorile matricei de ponderi pentru un strat Dense cu 10 neuroni de intrare și 10 neuroni de ieșire

3.5.2 Funcția de fitness

În cadrul proiectului au fost implementate și testate două funcții separate de fitness. Metodele de calcul al scorului de fitness sunt:

Metoda 1:

$$Fitness = steps + 2^{score} + score^{2.1} * 500 - (0.25 * steps)^{1.3} * score^{1.2}$$

Implementarea metodei de fitness în program:

```
def method1(self) -> float:
    fitness_score = self.steps + ((2 ** self.score) + (self.score ** 2.1) * 500) -
(((.25 * self.steps) ** 1.3) * (self.score ** 1.2))
    return fitness_score
```

Metoda 2:

$$Fitness = \frac{steps * score}{steps + score}$$

Implementarea metodei de fitness în program:

```
def method2(self):
    if self.steps == 0:
        return 0
    return (self.steps * self.score) / (self.steps + self.score)
```

Unde:

- steps: numărul de pași făcuți de șarpe de la începutul jocului până când a murit
- score: numărul de mere mâncate de șarpe

Metoda 1 a fost preluata din cadrul proiectului [8]. Metoda calculează scorul de fitness al individului folosind ponderi diferite pentru scorul și pașii care au fost obținuți de șarpe. Șarpele

este recompensat atunci când își crește numărul de pași și scorul și este penalizat dacă numărul de pași obținuți pentru scor este prea mare.

Metoda 2 a fost implementata în cadrul proiectului și este doar a medie armonica aplicata asupra pașilor și scorului șarpelui la care s-a omis înmulțirea cu constanta 2. Prin aplicarea mediei armonice se încurajează evoluția indivizilor care reușesc să obțină un scor mare într-un număr cât mai mic de pași. Prin această metoda se încearcă obținerea unor strategii eficiente care maximizează atât scorul, cât și eficiența mișcărilor șarpelui.

3.5.3 Operatori de selecție

Toți operatorii de selecție implementați în program primesc ca parametrii o listă *population* care conține toți indivizii din populația curentă a algoritmului și *selection_count* care specifică câți indivizi trebuie selectați din populația dată la intrarea funcției. Toți operatorii de selecție întorc o listă cu indivizii selectați.

3.5.3.1 Roulette Wheel

Funcția *roulette_selection* folosești scorul de fitness al indivizilor ca procentaj de selecție. Funcția începe prin calcularea sumei scorurilor de fitness pentru toți indivizii din populație, această valoare este folosită pentru a calcula procentajul ca un individ să fie selectat din populație.

Următorul pas constă în calcularea probabilităților de selecție pentru fiecare individ din populație prin utilizarea unei instrucțiuni de list comprehension. În această instrucțiune, scorul de fitness al fiecărui individ este împărțit la scorul total de fitness al întregii populații.

În final se folosești *random.choice* pentru a extrage indivizii din populație folosind probabilitățile de selecție calculate anterior.

```
total_population_fitness = sum(individual.fitness for individual în population)

selection_probabilities = [individual.fitness / total_population_fitness for
individual in population]

return list(np.random.choice(population, size=selection_count,
p=selection_probabilities))
```

3.5.3.2 Elitist

În selecția elitistă, se sortează populația în ordine descrescătoare în funcție de fitness-ul fiecărui individ. Funcția returnează indivizii cu cele mai mari scoruri de fitness până când numărul de indivizi selectați ajunge la *selection_count*.

```
selected = sorted(population, key=lambda individual: individual.fitness, reverse=True)
return selected[:selection_count]
```

3.5.4 Operatori de crossover

Operatorii de crossover realizează schimbul de informație genetică prin generarea de indivizi noi care conțin informații amestecate ale părinților. În cazul rețelelor neuronale, operatorii de crossover se aplică pentru matricea de ponderi și de bias a rețelei neuronale. Prin urmare, o funcție de crossover primește ca parametrii două matrici de tipul *numpy.ndarray*, *parent1_matrix* și *parent2_matrix*, care reprezintă matricele de ponderi sau bias ale părinților și întorc un tuplu care sunt matricele obținute în urma procesului de crossover.

3.5.4.1 One-Point Crossover

În cadrul funcției care implementează crossover-ul One-Point, se va proceda astfel: la începutul funcției se va selecta un punct de crossover prin alegerea aleatorie a unei coloane și a unei linii din numărul de linii și coloane din care sunt formate matricele de ponderile ale părinților, după care se va realiza interschimbarea valorilor matricelor între ele.

În această funcție se generează copii ale matricelor părinților numite *child1_matrix* și *child2_matrix*, după care se generează coordonatele punctului de crossover format din *crossover_row* și *crossover_col* care sunt variabile aleatorii luate din intervalul (0,matrix_row) și (0,matrix_col) unde *matrix_row* și *matrix_col* sunt numărul de linii și de coloane ale matricelor părinților.

Matricea primului copil va conține aceleași valori ca primul părinte până la punctul de crossover, iar după punctul de crossover va conține valorile de la matricea celui de-al doilea părinte. Pentru al doilea copil procesul este același, doar că ordinea părinților de la care se obțin valorile este inversat ă, matricea lui va conține informația de la al doilea părinte până la punctul de crossover, după care va conține informația primului părinte pana la sfârșitul matricei.

```
matrix_row, matrix_col = np.shape(parent1_matrix)

child1_matrix = parent1_matrix.copy()
child2_matrix = parent2_matrix.copy()

crossover_point = (np.random.randint(0, matrix_row), np.random.randint(0, matrix_col))

child1_matrix[:crossover_point[0], :] = parent2_matrix[:crossover_point[0], :]
child1_matrix[crossover_point[0], :crossover_point[1]] =
parent2_matrix[crossover_point[0], :crossover_point[1]]

child2_matrix[:crossover_point[0], :] = parent1_matrix[:crossover_point[0], :]
child2_matrix[crossover_point[0], :crossover_point[1]] =
parent1_matrix[crossover_point[0], :crossover_point[1]]

return child1_matrix, child2_matrix
```

3.5.4.2 *Two-Point Crossover*

Metoda aceasta este asemănătoare cu One-Point Crossover doar că aici se folosesc două puncte de crossover pentru a schimba materialul genetic.

La începutul funcției, se folosește funcția *np.shape* pentru a determina numărul de linii și numărul de coloane ale matricei unui părinte. După aceea, se creează două matrici goale cu dimensiunea părinților.

Folosind numărul de linii și de coloane se generează două puncte aleatoare pe matrice. În interiorul celor două for-uri se verifică prin intermediul unei condiții if dacă indexul curent se află între cele două puncte alese. Dacă indexul este între cele două puncte, atunci materialul genetic se schimbă cu materialul genetic al celuilalt părinte. Pentru matricea copil 1, materialul genetic va fi luat din interiorul punctelor de crossover de la părintele 2 și în afara punctelor de crossover de la părintele 1. Procesul este similar și pentru matricea copil 2, însă părinții sunt inversați.

```
matrix_row, matrix_col = np.shape(parent1_matrix)

child1_matrix = np.empty((matrix_row, matrix_col))
child2_matrix = np.empty((matrix_row, matrix_col))

point1 = (np.random.randint(0, matrix_row), np.random.randint(0, matrix_col))
point2 = (np.random.randint(0, matrix_row), np.random.randint(0, matrix_col))

for i in range(matrix_row):
    for j in range(matrix_col):
        if point1 < (i, j) < point2:
            child1_matrix[i, j] = parent2_matrix[i, j]
            child2_matrix[i, j] = parent1_matrix[i, j]
        else:
            child1_matrix[i, j] = parent1_matrix[i, j]
            child2_matrix[i, j] = parent2_matrix[i, j]

return child1_matrix, child2_matrix
```

3.5.5 Operatori de mutație

3.5.5.1 *Gaussian Mutation*

Operatorul *gaussian_mutation* folosit în lucrare primește ca parametrii matricea de valori a ponderilor sau a bias-urilor care urmează să treacă prin procesul de mutație și *mutation_rate*, probabilitatea ca o valoarea din interiorul matricei să fie schimbată.

```
after_mutation = matrix
mutation_array = np.random.random(after_mutation.shape) < mutation_rate

gauss_values = np.random.normal(size=after_mutation.shape)
after_mutation[mutation_array] += gauss_values[mutation_array]

return after_mutation
```

Funcția *random.random* este utilizata pentru a genera o matrice de aceeași dimensiune ca matricea *after_mutation* care conține valori aleatoare între 0 și 1. Aceasta matrice este folosită apoi pentru a determina care elemente ale matricei *matrix* vor trece prin procesul de mutație. Prin compararea valorilor aleatoare cu *mutation_rate*, se obține o matrice cu valori booleane *True* sau *False*, care indica dacă un element al matricei *matrix* va fi sau nu supus mutației.

În continuare, funcția *random.normal* este folosită pentru a genera o matrice de valori distribuite normal, având aceeași dimensiune ca și matricea *after_mutation*. În mod implicit, functia *np.random.normal* va genera valori care aparțin unei distribuții normale standard ce va avea media 0 si deviația standard 1.

Aceasta matrice de valori este apoi folosită pentru a efectua mutația elementelor din matricea *matrix* care corespund cu valorile *True* din matricea *mutation_array*. Mutația în sine se realizează prin adăugarea unei valori din matricea de valori distribuite normal generata înainte la valoarea din matricea original ă primită ca parametru.

3.5.6 Folosirea operatorilor genetici

Pentru folosirea operatorilor genetici în program s-au implementat două funcții care să realizeze procesul de crossover și mutație pentru toate matricele de ponderi și bias-uri care alcătuiesc rețeaua neuronală a unui individ.

Funcția *full_crossover* primește ca parametrii rețelele neuronale ale celor doi indivizi care trebuie să treacă prin procesul de crossover, și funcția crossover care se folosește. În interiorul funcției se parcurg straturile Dense ale părinților și se realizează crossover-ul pentru matricele de ponderi și pentru matricele de bias-uri ale părinților, care sunt apoi atribuite matricelor copiilor. În final, funcția întoarce cei doi indivizi copil care au fost obținuți.

```
def full_crossover(parent1: NeuralNetwork, parent2: NeuralNetwork, operator) ->
Tuple[NeuralNetwork, NeuralNetwork]:
    child1 = copy.deepcopy(parent1)
    child2 = copy.deepcopy(parent2)

    child1_dense_layers = child1.get_dense_layers()
    child2_dense_layers = child2.get_dense_layers()
    parent1_dense_layers = parent1.get_dense_layers()
    parent2_dense_layers = parent2.get_dense_layers()

    for i in range(len(parent1_dense_layers)):
        child1_dense_layers[i].weights, child2_dense_layers[i].weights =
operator(parent1_dense_layers[i].weights, parent2_dense_layers[i].weights)
        child1_dense_layers[i].bias, child2_dense_layers[i].bias =
operator(parent1_dense_layers[i].bias, parent2_dense_layers[i].bias)

    return child1, child2
```

Funcția *full_mutation* este asemănătoare cu *full_crossover*, doar că aici funcția primește ca parametru doar rețeaua neuronală a unui singur individ, rata de mutație și operatorul de mutație care se aplică. Funcția va parcurge toate straturile Dense ale rețelei neuronale și va realiza mutația asupra matricelor de ponderi și bias-uri folosind funcția care a fost primită ca parametru.

```
def full_mutation(individual: NeuralNetwork, mutation_rate: float, operator) -> None:
    individual_dense_layers = individual.get_dense_layers()

    for layer in individual_dense_layers:
        layer.weights = operator(layer.weights, mutation_rate)
        layer.bias = operator(layer.bias, mutation_rate)
```

3.6 Rularea programului

În cadrul aplicației există mai multe ferestre, iar utilizatorul va realiza antrenarea rețelei neuronale folosind fereastra pentru antrenare folosind algoritmul genetic, *GeneticTrainNetwork* sau fereastra pentru antrenare folosind regula Backpropagation, *BackpropagationTrainNetwork*. Ambele metode de antrenament vor salva rețelele neuronale care se obțin în fișiere json care vor fi folosite de utilizator în fereastra *RunTrainedNetwork* pentru a testa rețeaua neuronală care a fost obținută in urma antrenării.

Pentru afișarea ferestrelor aplicației și pentru desenarea pe ecran a elementelor de interfață se folosesc librăriile de Python: Pygame și Pygame-GUI.

Pentru a face gestionarea și trecerea dintre stările programului mai ușoară, se utilizează o mașină cu stări finite implementată în clasa *StateManager*. Mașina cu stări finiție reprezintă un model care descrie comportamentul unui sistem în funcție de stările în care se poate afla și de tranzițiile care pot apărea între acestea. Fiecare stare poate fi privită ca un mod de funcționare sau comportament pentru sistem, iar mașina cu stări finite are un număr limitat de stări, astfel încât sistemul poate fi într-o singură stare la un moment dat. Schimbarea de la o stare la alta se numește tranziție, și ea poate fi declanșată de evenimente interne, cum ar fi apariția unei erori sau terminarea execuției unui algoritm sau de evenimente externe, cum ar fi obținerea unui input de la utilizator.

3.6.1 Clasa de bază a stărilor din program

Fiecare stare posibila a programului este reprezentata implementând o clasă separata care moștenește clasa abstractă *BaseState*. Aceasta clasă abstractă definește metodele de bază pe care clasele derivate trebuie să le implementeze. Prin moștenirea clasei *BaseState*, fiecare stare poate fi gestionată în mod polimorfic în cadrul aplicației, asigurând astfel modularitate și flexibilitate mai bună. Astfel, se poate adaugă cu ușurința noi stări sau modifica comportamentul existent al stării fără a afecta restul aplicației.

```
class BaseState:
    def __init__(self, state_id: State):
        self.state_id = state_id
        self.target_state = None

        self.transition = False

        self.dată_to_send = {}
        self.dată_received = {}
```

În constructor se setează variabilele clasei:

- *state_id*: Este un valoarea unui enumerator care este folosit pentru a identifica starea folosită
- *target_state*: Este starea nouă la care va trece mașina cu stări finite în momentul când apare o tranziție
- *transition*: Este o variabila booleana care este verificata de mașina cu stări finite pentru a vedea dacă tranziția apare sau nu
- *dată_to_send* și *dată_received*: sunt dicționare folosite pentru a transmite date între stări. În momentul în care o stare vrea să transmită date către alta stare, acele date se vor pune în *dată_to_send*, iar pentru citirea datelor primite, starea la care se va realiza tranzitia va accesa *dată_received*

Funcțiile clasei:

- *trigger_transition*: Este apelată în momentul când o stare dorește să facă tranziție spre o stare nouă. Apelul funcției *trigger_transition* va seta variabila *transition* la *True*, indicând astfel faptul că este necesară tranziția la starea următoare. Tranziția efectiva la starea următoare va fi realizată de către *StateManager*.
- *set_target_state_name*: Este responsabilă pentru setarea stării următoare spre care programul trebuie să facă tranziția, în cazul în care se apelează funcția *trigger_transition*.
- *start*: Este apelată în momentul în care o nouă stare este activată. Aceasta funcție are rolul de a inițializa elementele de interfață specifice acelei stări, de a seta variabilele necesare și de a realiza orice alte acțiuni necesare pentru a pregăti starea pentru a fi utilizată. De exemplu, în cadrul unei stări de antrenare a rețelei neuronale, funcția start poate inițializa structura rețelei neuronale care va fi folosită pentru antrenare.
- *end*: Este apelată de către starea curentă în momentul în care apare o tranziție pentru a șterge de pe ecran toate obiectele care au fost desenate și inițializate în acea stare.
- *run*: Este apelată de către o stare pentru a-și rula codul său specific. Funcția primește ca parametrii variabila *surface* care este un obiect de tip *Surface* al bibliotecii Pygame, zona care

se afișează pe ecran, și *time_delta* care este timpul scurs între două frame-uri succesive ale buclei jocului și este folosit pentru a actualiza obiectele de pe ecran.

Funcțiile *run*, *end* și *start* nu sunt implementate în această clasă, ele trebuie să fie implementate în clasele *State* moștenitoare.

3.6.2 Mașina cu stări finite

În program, mașina cu stări finite a fost implementată în clasa *StateManager* care este responsabilă pentru gestionarea stărilor programului, a tranzițiilor dintre stări și pentru executarea comportamentului corespunzător pentru fiecare stare.

```
class StateManager:
    def __init__(self):
        self.states: Dict[State, BaseState] = {}
        self.active_state = None
        self.clock = pygame.time.Clock()
```

Constructorul clasei conține următoarele variabile:

- *states*: *StateManager* folosește dicționarul *states* pentru a memora toate stările posibile care au fost înregistrate în program. Fiecare intrare din dicționar este asociata cu o valoare a Enum-ului *State* care este folosit pe post de cheie pentru dicționar. Când trebuie să aibă loc o tranziție la alta stare, *StateManager* obține starea destinație din interiorul acestui dicționar.
- *active_state*: Este folosită pentru a memora starea activa a mașinii cu stări finite.
- *clock*: Este un obiect al bibliotecii Pygame care este folosit pentru a controla numărul maxim de frame-uri la care poate rula aplicația și pentru a calcula diferența de timp intre 2 cadre succesive pentru a realiza actualizarea elementelor vizuale pe ecran.

Metoda *execute_state* este metoda responsabilă de execuția aplicației și actualizarea stării programului. Instrucțiunea *clock.tick(MAX_FPS)* are două roluri: menținerea vitezei constantă în program și calculul diferenței de timp între două cadre succesive. Constanta *MAX_FPS* este utilizată în funcția *tick* pentru a controla viteza programului, astfel programul nu va rula cu mai mult de *MAX_FPS* cadre pe secunda.

Funcția *tick* întoarce numărul de milisecunde care au trecut de la ultimul apel al funcției în variabila *time_delta*, care reprezintă timpul care a trecut de când ultimul cadru a fost desenat pe ecran. Diferența de timp între cadre, *time_delta*, este folosită pentru a controla rata la care apar animațiile pe ecran sau pentru a controla alte comportamente dependente de timp în interfața grafică.

După obținerea variabilei *time_delta* se realizează rularea codului care corespunde cu starea activă folosind funcția *active_state.run*. Funcția *run* poate să modifice starea automatului prin crearea unei cereri de tranziție, care se va verifică în linia următoare a funcției. Dacă starea următoare a automatului este *States.QUIT* înseamnă că programul s-a terminat și metoda *execute_state* întoarce *False* pentru a spune programului principal să se oprească.

```
def execute_state(self, surface):
    if ViewSettings.DRAW:
        time_delta = self.clock.tick(ViewSettings.MAX_FPS) / 1000.0
    else:
        time_delta = 0

    if self.active_state is not None:
        self.active_state.run(surface, time_delta)

        if self.active_state.transition:
            if self.active_state.target_state == State.QUIT:
                return False

            self.active_state.transition = False
            new_state_id = self.active_state.target_state
            self.active_state.end()

            dată_to_send_copy = copy.deepcopy(self.active_state.dată_to_send)
            self.active_state = self.states[new_state_id]
            self.active_state.dată_received = dată_to_send_copy
            self.active_state.start()

        if ViewSettings.DRAW:
            pygame.display.flip()

    return True
```

Dacă starea următoare nu este *States.QUIT* atunci se pregătește mașina cu stări finite pentru tranziție. Mai întâi, variabila *transition* a stării curente este setată la *False*, iar apoi se obține ID-ul stării destinație și se apelează funcția *end* a stării active pentru a opri desenarea obiectelor pe ecran.

În continuare, se realizează o copie a datelor care sunt trimise de starea activ ă în variabila *dată_to_send_copy*, se caută starea următoare în dicționarul *states* folosind *new_state_id* pe post de cheie, și se actualizează datele primite de starea nouă ca fiind datele trimise de starea precedentă. După ce starea activ ă a fost modificată în mașina cu stări finite, se apelează funcția start a stării curente pentru a inițializa elementele vizuale specifice stării noi.

La finalul funcției se apelează *display.flip* pentru actualiza ecranul, pentru a desena din nou toate obiectele pe ecran, și se returnează *True* pentru a indica programului principal să continue execuția.

Inițializarea mașini cu stări, adăugarea stărilor și executarea stărilor se face în fișierul *main.py*, în interiorul funcției *main*. Adăugarea stărilor se face folosind funcția *add_state* a obiectului *StateManager*.

După adăugarea stărilor se setează starea inițiala a automatului la starea *MAIN_MENU* folosind funcția *set_initial_state*. Execuția codului are loc în *while* cât timp starea automatului nu este starea *States.QUIT*. Dacă programul iese din bucla se apelează instrucțiunea *pygame.quit()* pentru a termina toate modulele pygame care au fost folosite în program și pentru a elibera toate resursele ocupate de pygame.

Pe lângă inițializarea mașinii cu stări, în funcția *main* se inițializează configurații legate de afișarea programului pe ecran. Instrucțiunile folosite sunt:

- *pygame.init* inițializează modulul *pygame* care va fi folosit pentru desenarea pe ecran,
- *os.environ['SDL_VIDEO_CENTERED'] = '1'* setează poziția ferestrei programului în centrul ecranului.
- *screen=pygame.display.set_mode((ViewSettings.WIDTH, ViewSettings.HEIGHT))* setează dimensiunea ferestrei programului

Pe lângă instrucțiunile prezentate mai sus, se va realiza si inițializarea unei variabile *ui_manager* care este un obiect *UIManager* al clasei Pygame-gui si este folosit pentru a desena elementele de interfața pe ecran. Constructorul primește ca parametrii dimensiunea ecranului si calea către un fișier json care conține opțiuni care modifică aspectul vizual al obiectelor de interfața Pygame-Gui.

```
def main():
pygame.init()
screen = pygame.display.set_mode((ViewSettings.WIDTH, ViewSettings.HEIGHT))
ui_manager = UIManager(screen.get_size(), "data/themes/ui_theme.json")
state_manager = StateManager()
state_manager.add_state(MainMenu(ui_manager))
state_manager.add_state(Options(ui_manager))
state_manager.add_state(RunTrained(ui_manager))
state_manager.add_state(GeneticTrainNetwork(ui_manager))
state_manager.add_state(BackpropagationTrainNetwork(ui_manager))
state_manager.set_initial_state(State.MAIN_MENU)
running = True
while running:
    running = state_manager.execute_state(screen)
pygame.quit()
```

3.6.3 Structura comuna a unei stări

Cum a fost menționat și în capitolul 3.6.1, fiecare obiect de tip State moștenește clasa *BaseState* și trebuie să implementeze metodele *start*, *run* și *end*. Majoritatea claselor State urmează

aceeași structură, și din acest motiv se vor explica părțile comune folosind un singur exemplu, folosind clasa *MainMenu*.

Constructorul primește ca parametru *ui_manager* și apelează constructorul clasei de bază, adică constructorul clasei *BaseState*, cu valoarea unui enum care va fi folosit pentru a identifica starea, după care se inițializează obiectele clasei cu *None*, urmând ca ele să fie inițializate corect în interiorul funcției *start*

```
class MainMenu(BaseState):
    def __init__(self, ui_manager: UIManager):
        super().__init__(State.MAIN_MENU)
        self.ui_manager = ui_manager
        self.title_label = None
        self.button_backpropagation_menu = None
        self.button_genetic_menu = None
        self.button_quit = None
```

Funcția *start* inițializează obiectele clasei, aici se inițializează butoanele care vor fi folosite pentru navigarea în stările programului. Obiectele pentru interfața provin din biblioteca Pygame-GUI și majoritatea constructorilor primesc ca parametrii un obiect de tip *pygame.Rect* care este folosit pentru a defini poziția și dimensiunile obiectului. Un obiect de tip *Rect* este un dreptunghi care primește ca parametrii poziția pe ecran, lățimea și lungimea dreptunghiului. Obiectul *Rect* obținut este folosit ca primul argument în constructori, ceilalți doi parametrii sunt textul care îl va conține obiectul când va fi desenat pe ecran și *ui_manager* care gestionează cum se vor desena obiectele pe ecran.

```
self.title_label = UILabel(pygame.Rect(ViewSettings.TITLE_LABEL_POSITION,
ViewSettings.TITLE_LABEL_DIMENSION), "Main Menu", self.ui_manager)

self.button_backpropagation_menu = UIButton(pygame.Rect(((ViewSettings.WIDTH - 250) //
2, 250), (250, 35)), "Backpropagation", self.ui_manager)

self.button_genetic_menu = UIButton(pygame.Rect(((ViewSettings.WIDTH - 250) // 2,
350), (250, 35)), "Genetic Algorithm", self.ui_manager)

self.button_quit = UIButton(pygame.Rect(((ViewSettings.WIDTH - 150) // 2, 500), (150,
35)), "QUIT", self.ui_manager)
```

În metoda *end* a clasei se apelează funcția *clear_and_reset* pentru a reseta starea ecranului la starea de bază. Când este apelată, această funcție va șterge de pe ecran toate obiectele care au fost desenate și va reseta starea variabilei *ui_manager*.

```
def end(self):
    self.ui_manager.clear_and_reset()
```

În cadrul metodei *run*, se implementează funcționalitățile fiecărei stări din program. În cazul clasei *MainMenu*, aceasta nu conține un cod propriu pentru funcționalități specifice, ci se

ocupă doar cu verificarea evenimentelor care au apărut în cadrul jocului pentru a vedea care este fereastra următoare la care va trece programul.

Metoda începe prin verificarea tuturor evenimentelor din coada de evenimente aferente bibliotecii *pygame* folosind instrucțiunea *pygame.event.get*. Tipul evenimentelor care se verifică sunt:

- *pygame.QUIT*: Apare când utilizatorul închide fereastra programului, atunci înseamnă că programul a fost închis și se setează starea următoarea ca fiind *State.QUIT* pentru a spune mașinii cu stări finite să oprească execuția programului.
- *pygame.KEYDOWN*: Apare atunci când utilizatorul apasă o tastă. Pentru a identifica care tastă a fost apăsata se compara variabila evenimentului, *event.key* , cu taste care au fost pre-stabilite în cod. De exemplu, pentru a verifica dacă tasta ESCAPE a fost apăsata se va verifica instrucțiunea *if event.key == pygame.K_ESCAPE* care întoarce *True* daca evenimentul a fost declanșat de pasarea tastei Escape și fals altfel.
- *UI_BUTTON_PRESSED*: Apare atunci când un buton e apăsat de către utilizator. Pentru identificarea sursei se compară atributului *event.ui_element* cu toate butoanele care au fost definite de către clasă.

```
def run(self, surface: pygame.Surface, time_delta):
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            self.set_target_state_name(State.QUIT)
            self.trigger_transition()

        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_ESCAPE:
                self.set_target_state_name(State.QUIT)
                self.trigger_transition()

        self.ui_manager.process_events(event)
        if event.type == pygame_gui.UI_BUTTON_PRESSED:
            if event.ui_element == self.button_backpropagation_menu:
                self.set_target_state_name(State.BACKPROPAGATION_MENU)
                self.trigger_transition()
            if event.ui_element == self.button_genetic_menu:
                self.set_target_state_name(State.GENETIC_MENU)
                self.trigger_transition()
            if event.ui_element == self.button_quit:
                self.set_target_state_name(State.QUIT)
                self.trigger_transition()

    surface.fill(self.ui_manager.ui_theme.get_colour("dark_bg"))
    self.ui_manager.update(time_delta)
    self.ui_manager.draw_ui(surface)
```

Comanda *ui_manager.update* realizează actualizarea tuturor elementelor UI care sunt controlate de *ui_manager*, realizează actualizarea poziției, a dimensiunii și a aspectului fiecărui

element care este gestionat de *ui_manager*. Instrucțiunea *draw_ui* desenează toate elementele controlate de *ui_manager* pe ecranul care este primit ca parametru.

3.6.4 Starea Options

Aceasta stare este folosită pentru a permite utilizatorului să aleagă parametrii care vor fi folosiți pentru rularea programului. Se folosești o singura stare pentru selecția opțiunilor atât pentru starea care va realiza antrenarea folosind Regula Backpropagation cât și pentru starea care va folosi Algoritmul Genetic pentru antrenare.

Opțiunile pe care utilizatorul le poate configura sunt:

- Starting Snake Size: Dimensiunea inițiala a șarpelui la începutul programului
- Board Size: Dimensiunea tablei pe care să ruleze jocul de Snake
- Hidden Activation Function : Funcția de activare care va fi folosită de rețeaua neuronală pentru straturile ascunse. Opțiunile posibile pentru ambele opțiuni includ funcția sigmoidă, tangentă hiperbolică, ReLU.
- Segment return type / Apple return type: Tipul de informație care va fi reținut de liniile de viziune dacă se găsește un măr sau segment pe o linie. Poate să fie distanță sau valoare booleană
- Input Direction Count: Specifică numărul de linii de viziune pe care se va uita șarpele. Opțiunile sunt 4, doar direcțiile cardinale, sau 8 care conține direcțiile cardinale și diagonalele.
- Hidden Layer Neurons: Specifică numărul de neuroni care vor fi folosiți pentru formarea stratul ascuns

Următoarele opțiuni apar doar pentru algoritmul genetic:

- Individuals în population: Specifică numărul de indivizi pe care o generație din algoritmul genetic trebuie să ii aibă
- Crossover Operators / Mutation Operators: Specifică operatorii genetici care vor fi folosiți în algoritmul genetic.

Pentru a obține input-ul utilizatorului se folosesc obiectele din biblioteca pygame-gui: *UILabel* pentru a scrie pe ecran numele opțiunii, *UITextEntryLine* care este un fel de *TextBox* în care utilizatorul poate introduce valori de la tastatură, și *UIDropDownMenu* în care utilizatorul poate alege a valoarea din mai multe opțiuni posibile predefinite.

După ce utilizatorul selectează toate opțiunile dorite, se va apăsa butonul "Run" pentru a iniția tranziția și pentru a trimite opțiunile selectate către starea următoare. Opțiunile selectate sunt

salvate într-un dicționar numit *dată_to_send*, care va fi transmis în momentul realizării tranziției la starea următoare.

```
if self.options_target == "genetic":
    self.set_target_state_name(State.GENETIC_TRAIN_NEW_NETWORK)
    self.dată_to_send = {
        "input_direction_count":
int(self.dropdown_input_direction_count.selected_option),
        "segment_return_type": self.dropdown_segment_return.selected_option,
        "apple_return_type": self.dropdown_apple_return.selected_option,
        "distance_function": self.dropdown_distance.selected_option,
        "file_name": self.file_name_entry.text,
        "hidden_activation": self.dropdown_hidden_function.selected_option,
        "output_activation": self.dropdown_activation_function_output.selected_option,
        "input_layer_neurons":
int(self.neural_network_layers_entries["input"][0].text),
        "hidden_layer_neurons":
int(self.neural_network_layers_entries["first"][0].text),
        "output_layer_neurons":
int(self.neural_network_layers_entries["output"][0].text),
        "population_count": int(self.population_count_entry.text),
        "selection_operator": self.selection_operators_dropdown.selected_option,
        "crossover_operator": self.crossover_operators_dropdown.selected_option,
        "mutation_operator": self.mutation_operators_dropdown.selected_option,
        "mutation_rate": float(self.mutation_rate_entry.text),
        "initial_snake_size": int(self.starting_snake_size_entry.text),
        "board_size": int(self.board_size_entry.text)
    }
self.trigger_transition()
```

3.6.5 Antrenarea rețelei folosind algoritmul genetic

Procesul de antrenare a rețelei neuronale utilizând algoritmi genetici se realizează prin intermediul clasei *GeneticTrainNetwork*, care este o stare a mașinii cu stări finite.

Constructorul clasei definește mai multe obiecte care vor fi folosite în timpul rulării algoritmului genetic:

- *model*: Va memora obiectul de tipul Model, care este o reprezentare a jocului Snake
- *generation*: Este un contor care va reține numărul generației la care se afla algoritmul genetic
- *parent_list*: Este o listă care conține obiecte de tipul *Snake*, care sunt indivizi care au rulat și au fost evaluați în generația curentă și care vor trece prin procesele de selecție, crossover și mutație pentru a genera indivizi pentru generația nouă
- *offsprint_list*: Reține indivizii noi care au fost obținuți în urma folosirii operatorilor genetici folosind indivizii din *parent_list*. Aceasta listă reține indivizii care urmează să fie evaluați de către algoritmul genetic
- *networks*: Reține indivizii care obțin cel mai mare raport de mere/pași din ultimele 10 generații ale algoritmului genetic. În momentul în care utilizatorul decide să oprească antrenarea, rețelele memorate vor fi rulate din nou pentru a testa performanța obținută.

În funcția *start* are loc inițializarea rețelei neuronale și a obiectului Model care vor fi utilizate în timpul antrenării, și citirea opțiunilor care au fost alese de către utilizator pentru execuția algoritmului genetic.

```
def start(self):
    self.initial_board_size = self.dată_received["board_size"]
    self.initial_snake_size = self.dată_received["initial_snake_size"]
    self.input_direction_count = self.dată_received["input_direction_count"]
    self.segment_return_type = self.dată_received["segment_return_type"]
    self.apple_return_type = self.dată_received["apple_return_type"]
    self.population_count = self.dată_received["population_count"]
    self.crossover_operator = getattr(genetic_operators,
self.dată_received["crossover_operator"])
    self.mutation_operator = getattr(genetic_operators,
self.dată_received["mutation_operator"])
    self.mutation_rate = self.dată_received["mutation_rate"]
    self.file_name = self.dată_received["file_name"]

    input_neuron_count = self.dată_received["input_layer_neurons"]
    hidden_neuron_count = self.dată_received["hidden_layer_neurons"]
    output_neuron_count = self.dată_received["output_layer_neurons"]

    hidden_activation= getattr(neural_network,self.dată_received["hidden_activation"])
    output_activation= getattr(neural_network,self.dată_received["output_activation"])

       net = NeuralNetwork()
       net.add_layer(Dense(input_neuron_count, hidden_neuron_count))
       net.add_layer(Activation(hidden_activation, hidden_activation))
       net.add_layer(Dense(hidden_neuron_count, output_neuron_count))
       net.add_layer(Activation(output_activation, output_activation))
```

Pasul I al algoritmului genetic se realizează folosind funcția *run_genetic* și Pasul II se realizează folosind funcția *next_generation*.

Pasul I: Funcția run_genetic – evaluarea cromozomilor

Funcția *run_genetic* evaluarea cromozomilor din generația curenta a algoritmului. Pentru evaluarea cromozomilor, programul va realiza mișcarea șarpelui pe tablă până când acesta moare, dopa care va folosi funcția de fitness pentru a evalua performanta.

Pentru a determina direcția următoare în care se va muta șarpele se folosește doar pasul Forward al rețelei neuronale care primește la intrare liniile de viziune pentru tabla curentă a jocului. Folosind direcția prezisă de rețeaua neuronală, se apelează funcția *move* care realizează mișcarea șarpelui pe tablă și întoarce un *bool* care indică dacă șarpele a trăit sau a murit din cauza acestei mișcări.

În continuare, se verifică dacă șarpele a murit. Dacă nu a murit, atunci se continua execuția funcției folosind același șarpe. Dacă șarpele a murit atunci se calculează fitness-ul șarpelui și se introduce în lista *parent_list*

```
def run_genetic(self, surface):
vision_lines = get_vision_lines_snake_head(self.model.board, self.model.snake.body[0],
self.input_direction_count, apple_return_type=self.apple_return_type,
segment_return_type=self.segment_return_type)

neural_net_prediction = self.model.get_nn_output(vision_lines)
next_direction = self.model.get_nn_output_4directions(neural_net_prediction)
is_alive = self.model.move(next_direction)
```

În continuare, se verifică dacă numărul de indivizi evaluați a ajuns la numărul maxim de indivizi pe care o generație ar trebui să ii conțină, care a fost specificat de utilizator când au fost alese setările algoritmului. Dacă da, atunci se trece în pasul următor al algoritmului genetic, altfel se continuă execuția pasului curent prin evaluarea unui cromozom nou.

Dacă algoritmul este doar la prima generație, atunci cromozomul care se evaluează în continuare va conține o rețea neuronală ce are ponderile inițializate aleator.

În cazul în care algoritmul a trecut de prima generație, cromozomul care se evaluează în continuare este unul din cromozomii care au fost obținuți în pasul II al algoritmului genetic

```
if not is_alive:
    self.model.snake.calculate_fitness(self.model.max_score)
    self.parent_list.append(self.model.snake)
    if len(self.parent_list) == self.population_count:
        self.next_generation()

    if self.generation == 0:
        self.model.snake.brain.reinit_weights_and_biases()
        self.model = Model(self.initial_board_size, self.initial_snake_size, True,
self.model.snake.brain)
    else:
        self.model = Model(self.initial_board_size, self.initial_snake_size, True,
self.offspring_list[len(self.parent_list) - 1])
```

Pasul II: Funcția next_generation – aplicarea operatorilor genetici

În acest pas se aplică operatorii genetici care au fost prezentați în lucrare pentru a obține cromozomii care vor forma generația următoare a algoritmului genetic. Operatorii genetici vor fi aplicați în interiorul funcției *next_generation*. Pe lângă aplicarea operatorilor genetic, în această funcție se realizează salvarea informațiilor legate de execuția algoritmului, cum ar fi scorul mediu obținut de indivizii populației, media scorului de fitness al indivizilor și numărul de indivizi care au reușit să câștige jocul.

Primul lucru care se realizează în funcție este identificarea și salvarea indivizilor care obțin cel mai mare scor din generația curentă. Din generația curentă sunt salvați 15 indivizi care obțin cel mai mare raport de mere/pași.

```
sorted_individuals = sorted(
    self.parent_list,
    key=lambda individual: (
        individual.score,
        individual.score / individual.steps_taken if individual.steps_taken != 0 else
0),reverse=True)

best_generation_individuals = sorted_individuals[:15]

for individual in best_generation_individuals:
    if len(self.networks) == 150:
        self.networks.pop(0)

    self.networks.append([self.generation, individual.brain])
```

În continuare se folosești selecția elitista pentru a alege 10% indivizi care au cel mai mare scor de fitness din generația curentă. Acești indivizi vor fi puși în generația următoare a algoritmului fără să mai treacă prin procesul de crossover și mutație.

```
elitist_parents = elitist_selection(self.parent_list, self.population_count // 10)
for parent în elitist_parents [:self.population_count // 10]:
    self.offspring_list.append(parent.brain)
```

Pentru selecția indivizilor care vor trece prin procesul de reproducere se folosește operatorului genetic de selecție *roulette_selection* care realizează selecția unui număr de *population_count* indivizi din generația curentă.

```
selected_parents = genetic_operators.roulette_selection(self.parent_list,
(self.population_count))
```

În continuare se folosește o buclă *while* pentru a obține cromozomii generației următoare a algoritmului. Pentru fiecare pereche de indivizi care au fost selectați de către operatorul de selecție se realizează:

- se apelează funcția *full_crossover* care folosești operatorul de crossover primit ca argument pentru a realiza încrucișarea materialului genetic
- se apelează funcția *full_mutation* pentru fiecare individ obținut prin împerecherea părinților folosind operatorul de mutație care a fost ales de către utilizator
- la sfârșitul buclei se adaugă indivizii generați în lista *offspring_list*

```
selected_parents = genetic_operators.roulette_selection(self.parent_list,
(self.population_count - self.population_count // 10))
for i in range(0, (len(selected_parents)) - 1, 2):
    parent1 = selected_parents[i].brain
```

```
    parent2 = selected_parents[i + 1].brain

    child1, child2 = full_crossover(parent1, parent2, self.crossover_operator)

    full_mutation(child1, self.mutation_rate, self.mutation_operator)
    full_mutation(child2, self.mutation_rate, self.mutation_operator)

    self.offspring_list.append(child1)
    self.offspring_list.append(child2)
```

După ce toți indivizii care au fost selectați au trecut prin procesul de împerechere se va părăsi bucla și va avea loc inițializarea modelului cu rețeaua neuronală a primului individ din generația nouă și incrementarea contorului generației curente.

```
self.model = Model(self.initial_board_size, self.initial_snake_size, True,
self.offspring_list[0])
self.generation += 1
self.parent_list = []
```

După aplicarea pasului II, programul va trece din nou la pasul I al algoritmului genetic pentru a realiza evaluarea cromozomilor obținuți.

Algoritmul genetic va continua să ruleze până când utilizatorul decide să îl oprească prin apăsarea tastei Escape. În mod obișnuit, algoritmul a fost oprit manual când 10% din indivizii populației care au fost obținuți de algoritmul genetic au reușit să câștige jocul. Oprirea algoritmului se face în acest moment deoarece s-a considerat ca rezultatele obținute în acest punct sunt destul de bune.

La oprirea algoritmului se vor evalua rețelele care au fost salvate în *self.network* pentru a testa performantele obținute. Se vor parcurge pe rând rețelele salvate și se vor rula 100 de jocuri separate pentru fiecare rețea, iar media scorului, numărul de jocuri câștigate de rețea și media raportului de mere/pași obținute de rețea vor fi scrise într-un fișier text în același folder în care se salvează rețelele neuronale obținute pentru ca utilizatorul să se decidă care rețea este mai bună.

```
results = ""
for i, net in enumerate(sorted_individuals):
    current_string = "Position: " + str(i) + " Generation: " + str(net[0]) + " Average
Scores: " + str(net[2]) + " Average Ratios: " + str(net[3]) + " Won Counts: " +
str(net[4]) + " Won Ratios: " + str(net[5])
    print(current_string)
    results += current_string + "\n"

    data_to_save = {
        "generation": net[0],
        "initial_board_size": self.initial_board_size,
        "initial_snake_size": self.initial_snake_size,
        "input_direction_count": self.input_direction_count,
        "apple_return_type": self.apple_return_type,
        "segment_return_type": self.segment_return_type
    }
```

```
        name = "Position" + str(i) + " Generation" + str(net[0])
        nn_path = GameSettings.GENETIC_NETWORK_FOLDER + "/" + self.file_name + "/" + name
        save_neural_network_to_json(data_to_save, net[1], nn_path)

    write_genetic_training(results,GameSettings.GENETIC_NETWORK_FOLDER + self.file_name,
    False)
    write_genetic_training(self.training_data, GameSettings.GENETIC_NETWORK_FOLDER +
    self.file_name, False)
    print("DONE WRITING")
```

Un exemplu în care se prezintă cum arată fișierul text obținut în urma evaluării rețelelor neuronale este prezentat în figura 3.13

```
Position: 0 Generation: 429 Average Scores: 87.22 Average Ratios: 0.0400768875309254 Won Counts: 89
Position: 1 Generation: 429 Average Scores: 86.82 Average Ratios: 0.03897127705626748 Won Counts: 89
Position: 2 Generation: 426 Average Scores: 88.25 Average Ratios: 0.04062630262726004 Won Counts: 87
Position: 3 Generation: 424 Average Scores: 85.72 Average Ratios: 0.03861931553722006 Won Counts: 87
Position: 4 Generation: 429 Average Scores: 83.88 Average Ratios: 0.03752039244169904 Won Counts: 86
Position: 5 Generation: 429 Average Scores: 82.91 Average Ratios: 0.04305217846272 Won Counts: 85
Position: 6 Generation: 430 Average Scores: 83.68 Average Ratios: 0.04038545599889545 Won Counts: 85
Position: 7 Generation: 425 Average Scores: 92.66 Average Ratios: 0.0394731956324354 Won Counts: 85
Position: 8 Generation: 425 Average Scores: 88.79 Average Ratios: 0.038347247057288036 Won Counts: 85
Position: 9 Generation: 430 Average Scores: 82.85 Average Ratios: 0.03747125953903781 Won Counts: 85
Position: 10 Generation: 425 Average Scores: 83.3 Average Ratios: 0.036956035503020596 Won Counts: 85
Position: 11 Generation: 422 Average Scores: 83.43 Average Ratios: 0.03889257562867252 Won Counts: 84
Position: 12 Generation: 430 Average Scores: 84.19 Average Ratios: 0.038470225340980616 Won Counts: 84
Position: 13 Generation: 424 Average Scores: 83.68 Average Ratios: 0.03787466170793042 Won Counts: 84
Position: 14 Generation: 429 Average Scores: 82.31 Average Ratios: 0.0365656992498278 Won Counts: 84
Position: 15 Generation: 428 Average Scores: 85.65 Average Ratios: 0.03871168223909737 Won Counts: 83
Position: 16 Generation: 423 Average Scores: 87.8 Average Ratios: 0.038590740658104 Won Counts: 83
Position: 17 Generation: 422 Average Scores: 83.42 Average Ratios: 0.03857576132890538 Won Counts: 83
Position: 18 Generation: 427 Average Scores: 81.56 Average Ratios: 0.03928539527416082 Won Counts: 82
Position: 19 Generation: 425 Average Scores: 85.6 Average Ratios: 0.038413829124105354 Won Counts: 82
Position: 20 Generation: 421 Average Scores: 82.27 Average Ratios: 0.03810101994016761 Won Counts: 82
Position: 21 Generation: 429 Average Scores: 80.96 Average Ratios: 0.03738359572995308 Won Counts: 82
Position: 22 Generation: 425 Average Scores: 84.31 Average Ratios: 0.03841750433645955 Won Counts: 81
Position: 23 Generation: 428 Average Scores: 82.51 Average Ratios: 0.03787460856831038 Won Counts: 81
Position: 24 Generation: 427 Average Scores: 82.96 Average Ratios: 0.03750431743512542 Won Counts: 81
```

Figura 3.12 – Exemplu de rezultat obținut în urma evaluării rețelelor neuronale generate de algoritmul genetic

3.6.6 Evaluarea algoritmului genetic

Pentru evaluare s-a folosit o combinație între media scorului tuturor indivizilor din populație, raportul mediu al tuturor indivizilor din populație, numărul de indivizi care reușesc să câștige jocul și raportul de scor/pași pentru indivizii care reușesc să câștige jocul.

Evaluarea opțiunilor folosite pentru antrenare folosind algoritmul genetic:

- Funcția de fitness: Ambele metode de fitness care au fost folosite obțin performanțe apropiate. În urma mai multor teste s-a ajuns la concluzia că atât Metoda 1 cât și Metoda 2 de fitness vor obține indivizi care au media scorului și raportul de mere / pași asemănătoare.

- Informația stocată în liniile de viziune: De remarcat este că atunci când se folosește distanța segmentului în locul unei variabile boolene pentru liniile de viziune va scădea timpul de antrenament al algoritmului genetic și raportul de mere/pași va crește
- Numărul de linii de viziune: Atunci când se folosesc 8 direcții pentru proiecția liniilor de viziune raportul mediu al numărului de mere mâncate / pași va crește, dar procentul de jocuri câștigate de către rețea nu se schimbă.

Algoritmul genetic va obține indivizi care obțin cel mai bun raport atunci când se folosește 8 direcții pentru liniile de viziune, distanța pentru memorarea prezentei unui segment pe linia de viziune și valoare booleana pentru prezenta mărului pe acea linie de viziune. Pentru aceasta configurație, șerpii obținuți au raportul maxim de mere/pași între 0.06 – 0.065 și raportul mediu al șerpilor care reușesc să câștige jocul între 0.05 – 0.055 de mere/pași. Raportul de 0.06 indică faptul că șarpele are nevoie în medie de 16 pași pentru a manca un măr, iar raportul de 0.05 indică faptul că șarpele are nevoie în medie de 20 pași ca să mănânce un măr.

Șarpele va avea un raport mere/pași mai mare la începutul jocului, deoarece va trebui să facă mai puțini pași pentru a ajunge la măr, iar pe măsură ce șarpele mănâncă mai multe mere și devine mai mare, el va trebui să facă mai multe mișcări inutile pentru a ajunge la următorul măr, ceea ce va determina scăderea acestui raport pe parcursul jocului. Spre sfârșitul jocului, când nu vor mai fi locuri în care merele să fie plasate, raportul va începe să crească din nou.

3.6.7 Antrenarea rețelei folosind Backpropagation

Antrenarea rețelelor neuronale folosind Regula Backpropagation are loc în clasa *BackpropagationTrainNetork* care este o stare a mașinii cu stări finite.

Generarea exemplelor de antrenament care vor fi folosite pentru antrenarea rețelei se realizează în funcția *play_game_manual*. În interiorul funcției se realizează citirea, înregistrarea mișcărilor făcute de utilizator pentru starea jocului și mutarea șarpelui pe tablă în funcție de mișcarea înregistrată.

La începutul funcției se verifică dacă starea obținută pentru tabla curentă se află deja în exemplele de antrenament care au fost generate de către utilizator. Dacă da, atunci se desenează pe tablă mișcarea care a fost înregistrată în exemplul de antrenament generat în trecut. Utilizatorul poate să păstreze aceeași mișcare, sau poate să introducă una nouă. În acest caz nu se adaugă un exemplu nou de antrenament în listă, doar se înlocuiește mișcarea exemplului trecut cu noua mișcare realizată de utilizator.

```
def play_game_manual(self, surface, time_delta):
    snake_head = np.asarray(self.model.snake.body[0], dtype=np.int32)
    vision_lines = cvision.get_vision_lines_snake_head(self.model.board, snake_head,
self.input_direction_count, apple_return_type=self.apple_return_type,
segment_return_type=self.segment_return_type)
    old_lines = vision.cvision_to_old_vision(vision_lines)
    example_index = self.check_if_already_seen(old_lines)
```

Dacă nu exista nici un exemplu de antrenament care conține starea tablei atunci programul va desena tabla și liniile de viziune pe ecran, după care va aștepta ca utilizatorul să introducă mișcarea șarpelui.

Se folosesc tastele WASD pentru a specifică direcția pe care șarpele a fi trebuit să o ia pentru starea curentă. De exemplu, dacă utilizatorul apasă tasta W atunci se va pune în output-ul exemplului de antrenament [1.0, 0.0, 0.0, 0.0], care are forma asemănătoare cu forma rezultatului care este produs de rețeaua neuronală.

```
input_string = self.wait_for_key()
target_output = [0.0, 0.0, 0.0, 0.0]
direction_to_move = None
if input_string == "W":
    target_output[0] = 1.0
    direction_to_move = Direction.UP
if input_string == "S":
    target_output[1] = 1.0
    direction_to_move = Direction.DOWN
if input_string == "A":
    target_output[2] = 1.0
    direction_to_move = Direction.LEFT
if input_string == "D":
    target_output[3] = 1.0
    direction_to_move = Direction.RIGHT
```

După ce mișcarea utilizatorului a fost înregistrată se creează un obiect *TrainingExample* ce conține starea curentă și mișcarea realizată de utilizator, se adaugă exemplu de antrenament generat la lista de exemple de antrenament și se realizează mișcarea șarpelui pe tablă, după care programul va aștepta din nou ca utilizatorul să introducă mișcarea șarpelui.

Pentru a încheia generarea exemplelor de antrenament și a porni antrenarea rețelei utilizatorul va apasă tasta ESCAPE. Atunci când tasta e apăsata se vor stoca exemplele de antrenament generate într-un fișier json și se va apela funcția de antrenare backpropagation a rețelei neuronale care va folosi exemplele de antrenament pentru antrenarea rețelei.

Conținutul unui exemplu de antrenament salvat în fișier json este prezentat mai jos. Un exemplu de antrenament memorează offset-urile direcției anterioare pe care s-a mișcat șarpele în variabila *current_direction*, liniile de viziune obținute pentru starea tablei se memorează în

vision_lines și la sfârșit se memorează mișcarea care a fost făcută de utilizator pentru liniile de viziune ale exemplului de antrenament.

```
"current_direction": "DOWN",

"vision_lines":[
{"direction": "UP", "wall_distance": 0.14285714285714285, "apple_distance": 0.0,
"segment_distance": 0.0},

{"direction": "DOWN", "wall_distance": 0.25, "apple_distance": 0.0,"segment_distance":
0.0},

{"direction": "LEFT", "wall_distance": 0.1, "apple_distance": 0.0, "segment_distance":
1.0},

{"direction": "RIGHT", "wall_distance": 1.0, "apple_distance": 0.0,"segment_distance":
0.0}
],
"up": 0.0, "down": 1.0, "left": 0.0, "right": 0.0
```

Antrenarea rețelei neuronale se realizează folosind funcția *read_training_dată_and_train* care citește exemplele de antrenament care au fost stocate în fișierul json și apelează funcția *train* a rețelei neuronale. Variabila *x* este o listă cu input-urile rețelei neuronale (valorile liniilor de viziune și offset-urile direcției anterioare a rețelei) care au fost citite din exemplele de antrenament și variabila *y* este o listă ce conține mișcările care au fost realizate de către utilizator pentru stările citite. Stocarea exemplelor de antrenament generate în fișiere json a fost realizată pentru a se putea testa funcționarea corecta a antrenării folosind regula backpropagation.

```
def read_training_dată_and_train(network: NeuralNetwork, file_path: str) -> None:
    x, y = read_training_dată_json(file_path)

    input_neuron_count = network.get_dense_layers()[0].input_size
    output_neuron_count = network.get_dense_layers()[-1].output_size

    x = np.reshape(x, (len(x), input_neuron_count, 1))
    y = np.reshape(y, (len(y), output_neuron_count, 1))

    network.train(mse, mse_prime, x, y, 0.5)
```

Când se va finaliza antrenarea, ponderile rețelei neuronale și opțiunile folosite pentru antrenarea rețelei vor fi salvate într-un fișier json.

```
case pygame.K_ESCAPE:
    ViewSettings.DRAW = False
    data_to_save = {
        "generation": -1,
        "initial_board_size": self.initial_board_size,
        "initial_snake_size": self.initial_snake_size,
        "input_direction_count": self.input_direction_count,
        "apple_return_type": self.apple_return_type,
        "segment_return_type": self.segment_return_type
    }
```

```
    file_path = "Backpropagation_Training/" + str(self.input_direction_count) +
"_in_directions_" + str(self.dată_received["output_layer_neurons"]) +
"_out_directions.json"
    write_examples_to_json_4d(self.training_examples, file_path)

    self.model.snake.brain.reinit_weights_and_biases()
    self.model = Model(self.initial_board_size, self.initial_snake_size, False,
self.model.snake.brain)
    read_training_dată_and_train(self.model.snake.brain, file_path)

    save_neural_network_to_json(data_to_save,
                                self.model.snake.brain,
                                GameSettings.BACKPROPAGATION_NETWORK_FOLDER +
self.dată_received["file_name"])

    self.set_target_state_name(State.MAIN_MENU)
    self.trigger_transition()
    ViewSettings.DRAW = True
    break
```

3.6.8 Rularea unei rețele neuronale antrenate

Pentru rularea unei rețele neuronale care a fost antrenată deja se va folosi starea *RunTrainedNetwork*. Această stare este folosită de către utilizator pentru a încarcă un fișier json care conține ponderile și opțiunile care au fost folosite pentru antrenarea rețelei și pentru a rula jocul folosind predicțiile rețelei neuronale citite din fișier.

Utilizatorul va folosi butonul *Load Network* pentru a deschide o fereastră ce conține folderele în care s-au plasat fișierele json în care sunt stocate informațiile despre rețeaua neuronală. În momentul în care utilizatorul alege un fișier json, program va citi informațiile care au fost stocate în fișier (ponderile rețelei neuronale, numărul de direcții pe care au fost proiectate liniile de viziune)

```
if event.type == pygame_gui.UI_FILE_DIALOG_PATH_PICKED:
    try:
        self.file_path = create_resource_path(event.text)
        config = read_all_from_json(self.file_path)
        if config["generation"] == -1:
            self.state_target = "backpropagation"
        else:
            self.state_target = "genetic"
        self.network = config["network"]
        self.input_direction_count = config["input_direction_count"]
        self.apple_return_type = config["apple_return_type"]
        self.segment_return_type = config["segment_return_type"]
        self.board_size_entry.set_text(str(config["board_size"]))
        self.snake_size_entry.set_text(str(config["snake_size"]))
        self.label_return_type.set_text("Segment: " + self.segment_return_type + "
Apple: " + self.apple_return_type)
        self.button_load.enable()
        self.button_run.enable()
```

După ce fișierul json a fost ales și încărcat utilizatorul va folosi butonul *Run* pentru a începe execuția jocului. Pentru a prezice următoare direcție de mișcare, șarpele va folosi doar pasul forward al rețelei neuronale.

3.6.9 Stocarea și citirea caracteristicilor rețelelor neuronale în/din fișiere

Pentru scrierea și citirea rețelelor neuronale au fost implementate mai multe funcții în program care să permită salvarea și restaurarea caracteristicilor unei rețele neuronale și a opțiunilor care au fost folosite de program pentru obținerea ei într-un fișier json.

Caracteristicile rețelei neuronale care sunt salvate sunt:

- *board_size*: Dimensiunea tablei pentru care s-a realizat antrenarea
- *input_direction_count*: Numărul de direcții care au fost folosite pentru proiectarea liniilor de viziune
- *apple_return_type*: Tipul de valoare care a fost folosit pentru a memora prezenta mărului pentru liniile de viziune
- *segment_return_type*: Tipul de valoare care a fost folosit pentru a memora prezența segmentului pentru liniile de viziune
- *generation*: Generația curentă pentru care a fost obținută rețeaua care se salvează. Atunci când metoda de antrenament este Backpropagation se salvează valoarea -1.
- *network*: O listă care conține informații despre straturile rețelei neuronale care se salvează. Informațiile salvate depind de tipul stratului:
    - *Dense*: ponderile și bias-urile ale stratului, numărul de neuroni de intrare și de ieșire al stratului
    - *Activation*: funcția de activare și derivata funcției de activare

Funcția *save_neural_network_to_json* generează un fișier json în care se scriu informațiile rețelei neuronale și opțiunile folosite pentru antrenare care au fost prezentate mai sus.

Funcția primește ca parametru opțiunile folosite pentru configurarea și antrenarea rețelei, rețeaua neuronală care se salvează și calea la care se va salva rețeaua. La început, funcția parcurge toate straturile rețelei neuronale și salvează parametrii rețelei într-un dicționar. Dacă stratul care se verifică este de tip *Activation* atunci se va salva tipul de layer ca fiind ”activation”, după care se vor salva numele funcției de activare și numerele derivatei funcției de activare folosite de rețea. Dacă stratul este de tip *Dense*, atunci se salvează stratul ca fiind ”dense”, se salvează numărul neuronilor de intrare și numărul neuronilor de ieșire ai stratului, după care se salvează matricele de ponderi și bias-uri.

```
def save_neural_network_to_json(data_to_save: Dict, network: NeuralNetwork, path: str)
-> None:
    network_list = []
    for i, layer in enumerate(network.layers):
        if type(layer) is Activation:
            layer_dict = {
                "layer": "activation",
                "activation": layer.activation.__name__,
                "activation_prime": layer.activation_prime.__name__
            }
        else:
            layer_dict = {
                "layer": "dense",
                "input_size": layer.input_size,
                "output_size": layer.output_size,
                "weights": layer.weights.tolist(),
                "bias": layer.bias.tolist()
            }
        network_list.append(layer_dict)

    option_dict = {"generation": data_to_save["generation"],
                   "board_size": data_to_save["initial_board_size"],
                   "snake_size": data_to_save["initial_snake_size"],
                   "input_direction_count": data_to_save["input_direction_count"],
                   "apple_return_type": data_to_save["apple_return_type"],
                   "segment_return_type": data_to_save["segment_return_type"],
                   "distance_function": data_to_save["distance_function"],
                   "network": network_list}

    path_tokens = path.split("/")
    path_tokens = path_tokens[:-1]
    real_path: str = ""
    for token în path_tokens:
        real_path += token + "/"

    if not os.path.exists(real_path):
        os.makedirs(real_path)

    network_file = open(path + ".json", "w")
    json.dump(option_dict, network_file)
    network_file.close()
```

Pentru citirea informațiilor scrise în fișierele json se folosește funcția *read_all_from_json*. La început, se citește fișierul json din calea primită ca parametru și se obține un fișier *json_object*. Funcția va realiza citirea configurației rețelei neuronale care a fost salvată în fișier. Pentru acest lucru se va lua pe rând fiecare strat salvat în fișier.

Dacă stratul este de tip ”dense” atunci se citește numărul de neuroni de intrare și numărul de neuroni de ieșire ai stratului din fișier, după care se citesc ponderile și bias-urile memorate în fișier. După ce au fost citite toate informațiile pentru strat, se creează un nou obiect de tip *Dense* și se adaugă în listă de straturi a rețelei neuronale.

Dacă stratul este de tip ”activation” atunci se citește numele funcției de activare și a derivatei funcției de activare din fișier și se folosește instrucțiunea *getattr* pentru a obține funcțiile din fișierul *neural_network* ce au același nume cu cele citite din fișier. După citirea funcțiilor de activare se creează un nou obiect *Activation* și se adaugă în lista de straturi a rețelei neuronale.

După ce toate straturile au fost citite, se actualizează valorile din obiectul json și obiectul modificat este returnat pentru a putea fi folosit de către program.

```
def read_all_from_json(path: str) -> Dict:
    json_file = open(path, "r")
    json_object = json.load(json_file)

    output_network = NeuralNetwork()
    if json_object:
        for layer in json_object["network"]:
            if layer["layer"] == "dense":
                input_size = layer["input_size"]
                output_size = layer["output_size"]
                weights = np.reshape(layer["weights"], (output_size, input_size))
                bias = np.reshape(layer["bias"], (output_size , 1))
                dense_layer = Dense(input_size, output_size)
                dense_layer.weights = weights
                dense_layer.bias = bias
                output_network.add_layer(dense_layer)
            else:
                activation_str = layer["activation"]
                activation_prime_str = layer["activation_prime"]

                activation = getattr(neural_network, activation_str)
                activation_prime = getattr(neural_network, activation_prime_str)

                activation_layer = Activation(activation, activation_prime)
                output_network.add_layer(activation_layer)
    output = json_object
    output["network"] = output_network
    return output
```

4 Concluzii

Obiectivele atinse cu succes în proiect:

În program s-a implementat o metodă de obținere a intrărilor rețelei neuronale folosind o viziune redusă a tablei de joc. Pentru intrarea rețelei se folosesc valorile liniilor de viziune, raze care pornesc din capul șarpelui în direcții diferite, care rețin distanța de la capul șarpelui pană la măr, segment al corpului și peretele din acea direcție.

Proiectul a reușit cu succes să integreze rețelele neuronale împreună cu algoritmii genetici, obținându-se astfel o metodă de antrenament automată și nesupervizată care nu necesită exemple de antrenament. Rețelele neuronale au fost folosite pentru a realiza mutarea șarpelui pe tablă în funcție de valorile liniilor de viziune obținute pentru tablă, iar algoritmii genetici au fost folosiți pentru a realiza evaluarea configurațiilor rețelelor neuronale ale indivizilor din populație și pentru a îmbunătății ponderile și bias-urile rețelelor indivizilor din generațiile următoare folosind operatorii genetici de selecție, crossover și mutație.

Pe lângă algoritmii genetici, în cadrul proiectului s-a implementat și algoritmul backpropagation pentru antrenarea rețelelor. Pentru generarea exemplelor de antrenament necesare, în proiect a fost realizat un mecanism care generează exemple de antrenament folosind mișcările realizate de utilizator. Programul pornește jocul și așteaptă ca utilizatorul să mute șarpele pe tablă, iar pentru fiecare mișcare efectuată de utilizator se creează un exemplu de antrenament care conține valorile liniilor de viziune obținute pentru tablă și mișcarea realizată de utilizator pentru acele linii de viziune.

Concluzii pentru antrenarea rețelei folosind Algoritmul Genetic:

Pentru antrenarea rețelei folosind algoritmii genetici a fost necesar un timp destul de scurt pentru antrenare. Pentru antrenarea rețelei folosind algoritmului genetic care a conținut o populație de 1000 de indivizi și folosind o tablă de 10x10 a durat în medie între 15 – 25 de minute până când algoritmul a obținut primul individ care să reușească să câștige jocul, și intre 40 – 60 de minute până când 10% din indivizii populației reușesc să câștige jocul, moment în care execuția algoritmul a fost oprită.

O problemă care apare în program este obținerea de mișcări suboptime de către șarpe. Din cauza viziunii reduse a tablei de joc, din cauză că se folosesc linii de viziune pentru a obține intrarea rețelei neuronale, șarpele nu va realiza de fiecare dată mișcarea optimă, va realiza mișcarea sigură pentru valorile liniilor de viziune pe care le primește. Pozițiile segmentelor corpului șarpelui pe tablă se pot mișca și folosind liniile de viziune șarpele nu poate să vadă toate segmentele

corpului care se află pe tablă. Atunci când șarpele are oportunitatea să meargă direct la măr, el ar putea să se blocheze în interiorul corpului său și să moară. În timpul antrenării sunt descoperite mai multe strategii încearcă să rezolve această problemă.

Șarpele învață să parcurgă tabla folosind două direcții, o direcție principală de deplasare care va fi folosită pentru parcurgerea tablei și o direcție secundară pe care se va deplasa atunci când descoperă un măr. Ca un exemplu, se consideră că șarpele va parcurge tabla folosind direcția principală de sus în jos, și direcția secundară de la dreapta la stânga. La început, șarpele se va deplasa în colțul din partea dreapta sus al tablei, după care va începe să se miște în jos în timp ce rămâne lipit de peretele din dreapta. În momentul în care șarpele vede un măr se va deplasa la el folosind direcția secundară, în exemplu se va deplasa la stânga până mănâncă mărul. După ce șarpele mănâncă mărul, el se va întoarce la peretele din dreapta înainte să se deplaseze din nou în jos. Când șarpele ajunge la peretele de jos al tablei, atunci când a parcurs toată tabla, el se va întoarce la peretele de sus al tablei, după care va relua din nou mișcarea în jos. Strategia aceasta se asigura că șarpele nu va obține situații în care el să se blocheze și să moară din cauza viziunii reduse.

O altă strategie care a fost descoperită este că atunci când șarpele devine destul de mare el începe să se încolăcească ca să evite mai ușor segmentele corpului pentru a reduce mișcările inutile făcute. Pentru exemplul prezentat mai sus, când șarpele devine destul de mare se va muta la început astfel încât segmentele corpului să ocupe partea de sus a tablei, astfel se reduc numărul de situații în care el realizează mișcări inutile pentru a evita segmentele corpului.

În urma testelor s-a constatat că șerpii obținuți de algoritmul genetic nu vor reuși să câștige jocul de fiecare dată. Pentru o tablă de 10x10, rețelele neuronale obținute vor câștiga jocul cu o probabilitate intre 70% si 90% cu un raport mediu de mere / pași de 0.045.

Concluzii pentru antrenarea rețelei folosind Regula Backpropagation:

Din cauza viziunii reduse a tablei de joc, va fi dificil ca utilizatorul să genereze exemple de antrenament corecte. Dacă exemplele nu sunt realizate ținând cont că șarpele percepe tabla doar folosind liniile de viziune, atunci rețeaua neuronală obținută în urma antrenamentului nu va obține rezultate bune. Mișcările pe care utilizatorul trebuie să le realizeze trebuie să fie sigure, să țină cont de faptul că șarpele nu poate să vadă toata tabla, mișcările realizate să fie potrivite în toate cazurile posibile indiferent de pozițiile segmentelor corpului șarpelui care nu pot să fie văzute.

Dacă utilizatorul nu respectă această limitare atunci când generează exemplele de antrenament, atunci performanță șarpelui obținut va fi limitată. Șarpele va reuși sa învețe din exemplele de antrenament obținute de la utilizator, dar din cauza vizunii reduse, el se va afla în

foarte multe situații în care realizează o mișcare prin care se va bloca în interiorul corpului și va muri.

În urma mai multor teste, s-a ajuns la concluzia că pentru generarea exemplelor de antrenament corecte utilizatorul ar trebui să urmeze o strategie asemănătoare cu strategia obținută atunci când s-au folosit algoritmii genetici pentru antrenarea rețelei.

Folosind algoritmul backpropagation pentru antrenarea rețelei neuronale s-au obținut șerpi cu performanțe mai slabe decât atunci când s-a folosit algoritmul genetic pentru antrenare. Folosind 2000 de exemple de antrenament s-au obținut rețele care reușesc sa câștige jocul, dar procentajul de jocuri câștigate este mai mic decât atunci când s-au folosit algoritmii genetici pentru antrenament. Pentru o tabla de 10x10, cea mai buna rețea care a fost obținuta folosind algoritmul backpropagation are o șansa de 40% de a câștiga jocul, cu un scor mediu de 60. Acest lucru se întâmpla din 2 motive: dificultatea generării exemplelor de antrenament corecte care a fost prezentată mai sus, și numărul de exemple de antrenament care au fost folosite a fost insuficient pentru ca rețeaua neuronală să învețe să joace complet jocul.

Dezvoltări ulterioare:

Un aspect care ar putea fi îmbunătățit în program pentru a permite șarpelui să realizeze mișcări mai bune, atât când se folosesc algoritmii genetici pentru antrenare cât și atunci când se folosește regula backpropagation, este utilizarea unui număr mai mare de linii de viziune. Prin adăugarea unui număr mai mare de linii de viziune șarpele poate obține informații suplimentare despre poziția merelor, segmentelor corpului și zidurilor de pe tabla de joc. Utilizând aceste informații suplimentare, mișcările realizate de șarpe pot fi mai bine informate și mai adaptate la starea tablei de joc, șarpele va putea realiza mișcarea optimă pentru starea curentă a tablei de joc.

5 Referințe bibliografice

[1] BÄCK T. - Evolutionary Algorithms în Theory and Practice, Oxford Univ. Press, 1996

[2] GOLDBERG D. – Genetic Algorithms în Search, Optimization and Machine Learning, 1989

[3] GOLDBERG D., DEB K. – A Comparative Analysis of Selection Schemes Used în Genetic Algorithms, 1991

[4] MCCULLOCH W., WALTER P. – A Logical Calculus of Ideas Immanent în Nervous Activity, 1943

[5] HINKELMANN, K – Neural Networks, 2018

[6] NIELSEN, MICHAEL A. – How the Backpropagation algorithm works, Determination Press, 2015

[7] https://towardsdatascience.com/math-neural-network-from-scratch-in-python-d6da9f29ce65 "Neural Network from scratch în Python" - Accesat în 14.08.2022

[8] https://chrispresso.io/AI_Learns_To_Play_Snake „AI Learns To Play Snake” - Accesat în 14.08.2022

[9] https://www.probabilitycourse.com/chapter9/9_1_5_mean_squared_error_MSE.php - „Mean Squared Error” – Accesat în 20.10.2022