

UNIVERSITATEA POLITEHNICA DIN BUCUREȘTI

FACULTATEA DE ȘTIINȚE APLICATE

Matematică și Informatică Aplicată în Inginerie

PROIECT DE DIPLOMĂ

**CONDUCĂTOR ȘTIINȚIFIC,
Lect.univ.dr. Iuliana MUNTEANU**

**ABSOLVENT,
Robert Daniel SOARE**

București

2021



UNIVERSITATEA POLITEHNICA DIN BUCUREȘTI
FACULTATEA DE ȘTIINȚE APLICATE
Matematică și Informatică Aplicată în Inginerie



Aprobat Decan,
Prof.dr. Emil PETRESCU

PROIECT DE DIPLOMĂ

Învățarea automată pentru un agent într-un mediu 2D

CONDUCĂTOR ȘTIINȚIFIC,
Lect.univ.dr. Iuliana MUNTEANU

ABSOLVENT,
Robert Daniel SOARE

București

2021

Cuprins

Introducere	4
1 Învățare automată	7
1.1 Istoric	7
1.2 Clasificare	8
1.3 Industrie	9
1.4 Programe software pentru dezvoltare	10
1.5 Big Data	10
2 Rețele neuronale artificiale	13
2.1 Introducere	13
2.2 Structură	13
2.3 Funcții de activare și metode de optimizare	16
2.4 Tensorflow	22
3 Metode de învățare	25
3.1 Lanț și proces Markov	25
3.2 Q-Learning	27
4 Aplicație	31
4.1 Introducere	31
4.2 Structură	32
4.3 Simulator	33
4.4 Agent	35
4.5 Model de învățare	38
4.6 Interfață Utilizator	43
Concluzii finale	44
Bibliografie	46
Index	47

Introducere

Învățarea automată a devenit un subiect de interes din ce în ce mai important, această fiind utilizată în vaste domenii, precum: industria auto, alimentară, agricolă, bancară, aerospațială și mai cu seamă în industria tehnologiei informației. Unul din rolurile ei cele mai importante constă în analiza și clasificarea datelor, predicția unor evenimente în baza unor fapte deja întâmplare, crearea unui profil virtual pentru un grup de utilizatori, etc.

Datorită marelui conectivități dintre oamenii din ziua de astăzi; sistemele politice, economice și relațiile interumane au devenit extrem de complexe. Totul a devenit interconectat. O idee a unui singur individ poate fi transmisă pe tot globul pământesc, această idee putând să afecteze milioane de oameni în diverse locuri și a cărui impact politic și economic sunt greu de estimat. De asemenea, un incident economic local, un dezastru natural, sau un conflict politic dintre două țări pot avea efecte devastatoare asupra economiei globale și a structurii geopolitice curente.

Fiecare eveniment din ziua de astăzi are o influență mai mică sau mai mare asupra acestei mari rețele de sisteme ale civilizației umane. Întrebarea naturală la această dilemă este: putem face o estimare asupra acestor evenimente și ale cazurilor lor speciale? Se poate, și asta datorită faptului ca multe evenimente sunt monitorizate și înregistrate, precum: tranzacțiile bancare, documente legislative și juridice, vremea, traseele și destinațiile automobilelor de transport marfă (mașini personale, avioane, vapoare), discursuri și opinii în rețelele sociale, date medicale din dispozitive inteligente (telefoane mobile, ceasuri și brățări), date provenite din simulări virtuale sau experimente.

Tot acest volum de informații și metodele sale de manipulare, stocare întră în așa numita categorie *Big Data*. Analiza acestui volum imens de date devine o sarcină foarte dificilă și laborioasă în cazul metodelor convenționale de analiză a datelor folosind statistica clasică. În esență, învățarea automată se folosește atât de teoria clasică cât și de noile descoperiri în calculul numeric pentru a crea modele matematice dinamice care pot acumula cunoștințe și acționa în baza lor folosind toate datele pe care le primește ca set de învățare.

În această lucrare se va analiza cum algoritmi de învățare automată pot fi folosiți în crearea unui agent autonom care să îndeplinească sarcinii într-un spațiu 2-dimensional. Problema constă în rezolvarea unui traseu de tip matrice în care agentul trebuie să ajungă la destinație fără a produce un accident. Vederea asupra acestui traseu este de sus în jos, echivalentul cu o imagine provenită de la o dronă în aer. Acest traseu este de forma unui

labirint.

Analiza se va face cu ajutorul unei aplicații web interactive în care vom simula mediul nostru 2-dimensional reprezentat de un labirint și care ne va permite analiza datelor furnizate de către agent în timpul sesiuni de antrenament pentru determinarea eficienței și fiabilității algoritmilor.

În primul capitol este descris termenul de învățare automată, care sunt subdomeniile sale, cum este folosit în industrie și ce presupune acesta.

În al doilea capitol este descrisă structura de bază a unei rețele neuronale și a elementelor din care este compusă. Exemplificarea utilizării ei prin exemple de cod.

În al treilea capitol se va descrie algoritmul de învățare Q-Learning și cum se folosește pentru conferii relații dintre datele de intrare și cele ieșire astfel încât să obținem rezultatele dorite.

În al patrulea capitol se va descrie structura aplicației web, a componentelor sale și modul cum acestea interacționează. Sunt prezentate elementele de cod componente care descriu ideile și algoritmii prezentați în capitolele precedente.

Capitolul 1

Învățare automată

1.1 Istoric

Învățarea automată este o ramură a inteligenței artificiale care se ocupă cu studiul tehnicilor și metodelor prin care se oferă unui calculator abilitatea de a învăța. Prin învățare ne referim la posibilitatea de a oferi o decizie în baza unor cunoștințe deduse din experiențele anterioare obținute în urma unui proces de antrenare.

Multe tehnici din învățarea automată au la bază modelul de interacțiune al neuronului, descris de către Donal Hebb în cartea sa *The Organization of Behavior* [2]. Termenul de învățare automată (în engleză *machine learning*) a aparut în anul 1953, dat de Arthur Samuel, creatorul unui program de jucat checker, capabil să ia decizii bazate pe experiențele anterioare [7]. În anul 1957, Frank Rosenblatt crează Perceptron-ul (folosindu-se de observațiile din lucrările lui Donald Hebb și Arthur Samuel), utilizat în crearea unui calculator capabil să recunoască forme într-o imagine. Perceptron-ul, care este echivalent cu un singur neuron, de unul singur are o putere destul de limitată, dar odată cu utilizarea sa în combinații de mai multe straturi a dat naștere structurii denumită rețea neuronală.

De-a lungul timpului, acest domeniu a avut o evoluție înceată, un factor important fiind capacitățile limitate de procesare ale calculatoarelor. Dar odată cu avansurile tehnologice, cercetarea în acest domeniu a început să fie din ce în ce mai activă, în ultimii ani culminând cu evenimente care au atras interesul publicului general, precum: IBM's Deep Blue, IBM's Watson, Google's Deepmind și Google's AlphaGo.

1.2 Clasificare

Fiind un domeniu foarte vast și cuprinzător, aceasta se împarte în 3 mari categorii:

- Învățare supervizată
- Învățare nesupervizată
- Învățare prin recompensă

În învățarea supervizată, procesul de antrenare se bazează pe analiza unor date formate din perechi de valori intrare-ieșire (set de date etichetate) pentru calibrarea funcțiilor de deducere. Este folosit pentru rezolvarea problemelor de clasificare.

Exemple de algoritmi:

- Support-vector machines
- Regresia liniară
- Regresia logistică
- Arbori de decizie
- Rețele neuronale
- Clasificator bayesian naiv

Pentru învățarea nesupervizată, procesul de antrenare constă în crearea unor modele interne de recunoaștere a unor tipare în urma analizei unui set de date neetichetat. Este deseori folosit în descoperirea similarităților și diferențelor într-un set de date.

Exemple de algoritmi:

- K-means clustering
- Autoencoders
- Analiza componentei principale
- Descompunerea valorilor singulare

În învățarea prin recompensă, procesul de antrenare constă în minimizarea unei funcții de cost care are la bază recompensele colectate de către agent în urma acțiunilor sale, modelul calibrându-se astfel încât deciziile luate să ducă spre obținerea unei recompense cât mai mari.

Exemple de algoritmi:

- Monte Carlo
- Q-learning

- SARSA
- Deep Q Network
- Proximal Policy Optimization
- Deep Deterministic Policy Gradient
- Trust Region Policy Optimization

1.3 Industrie

Tot mai multe aplicații folosesc tehnici de învățare automată pentru optimizarea produselor, serviciilor și interacțiunilor cu utilizatorii. Cele mai notabile utilizări fiind:

- Algoritmi de căutare a știrilor în baza unor preferințe oferite explicit sau implicit de către utilizator.
- Reclame personalizate generate după profilele utilizatorilor.
- Sisteme de recomandări produse.
- Etichetarea obiectelor sau persoanelor în imagini, înregistrări audio sau video.
- Sisteme robotice autonome.
- Mașini autonome.
- Sisteme meteorologice
- Sisteme de detectare a fraudelor într-un sistem bancar.
- Clasificare și predicția evenimentelor.
- Optimizarea proceselor de producție a mărfurilor.
- Optimizarea procesului de antrenare pentru atleți.

Companiile sunt foarte interesate de modul cum interacționează și percep clienții produselor lor, acestea încercând mereu să colecteze informații pentru despre modul cum sunt utilizate produsele în activitatea utilizatorului. Aceste campanii de colectare a datelor au devenit din ce în ce mai agresivă, marile companii software specializate în rețele sociale (Facebook, Twitter, Youtube, Linkedin, Reddit) vând datele utilizatorilor în vederea oferirii unui profil al consumatorului pentru a stabili interesul pentru produs. Astfel, se poate vinde clientului o reclamă personalizată după preferințele sale.

1.4 Programe software pentru dezvoltare

Interesul puternic pentru acest domeniu a venit în principal din partea marilor companii software și hardware, ele dezvoltând puternice biblioteci software și echipamente pentru procesarea datelor, crearea de rețele neuronale, algoritmi de învățare, etc. Pentru sprijinirea domeniului, aceste unelte sunt oferite după ca aplicații cu sursă deschisă (*open source*), având o licență deseori foarte permisibilă în vederea utilizării personale și comerciale (licență de tip MIT, APACHE, GNU).

Calitatea acestor unelte le-a făcut să devină un standard în industrie, atât comercială cât și academică.

Exemple de biblioteci sau aplicații software:

- Tensorflow - bibliotecă dezvoltată de către Google în vederea utilizării cu ușurință algoritmilor de învățare, cât și funcții utilitare pentru manipularea datelor.
- PyTorch - bibliotecă dezvoltată de către Facebook pentru protiparea aplicațiilor de viziune computerizate, procesarea limbajului natural, etc.
- ML.NET - bibliotecă dezvoltată de Microsoft pentru crearea rapidă a unor aplicații de procesare a datelor folosind algoritmi de învățare.
- scikit-Learn - bibliotecă care conține funcții statistice folosite pentru analiza datelor.
- Apache Spark - bibliotecă de aplicații destinate pentru procesarea unui volum foarte mare de date.
- Apache Kafka - aplicație care permite stocarea și distribuirea unui volum foarte mare de date în timp real către mai mulți consumatori.
- Caffe - bibliotecă pentru dezvoltare aplicațiilor pentru medii de lucru care nu dispun de o putere de procesare foarte mare, precum dispozitivele mobile.
- Keras - bibliotecă pentru dezvoltarea rețelelor neuronale
- H2O.ai - platformă de procesare și analiză a datelor

1.5 Big Data

O componentă esențială pentru învățarea automată este gestionarea datelor care vor fi folosite și produse de către algoritmi de învățare. Această gestionare a informațiilor, de cele mai multe ori, va intra în cadrul domeniului de *Big Data*

Conform Uniunii Europene: „Big data se referă la volume de date colectate atât de mari și complexe încât este nevoie de noi tehnologii, cum ar fi inteligență artificială, pentru a le procesa. Datele provin din nenumărate surse diferite.”[9]

Volumul de date pe care omenirea îl produce crește de la an la an, ceea ce face analiza și înțelegerea datelor să fie o sarcină din ce în ce mai dificilă. Tot mai mulți oameni încep să

aibă acces la internet, iar numărul de dispozitive inteligente (smart phone, smart watch, smart TV) pe care un individ de dispune crește odată cu avansul tehnologic.

Principalele surse de proveniență ale acestor date sunt:

- Rețele sociale - mesaje, imagini create de utilizatori pentru ași exprima opinia la situația socială, economică și politică - datele pot fi utilizate pentru stabilirea unor tendințe sociale cu privire la activitatea și starea emoțională curentă și viitoare a oamenilor.
- Mediul și natura - date provenite de la sateliți și senzori pentru monitorizarea schimbărilor climatice - folosite pentru predicția posibilelor dezastre naturale cauzate de activitățile omului.
- Sector public - documente, certificate, atestate, adeverințe emise de către instituțiile publice - pot fi utilizate în eficientizarea serviciilor publice.
- Transport - date colectate prin GPS și de la diferiți operatori în domeniul transportului (transportul public, aeroporturi, gări) - pentru optimizarea rutelor și a curselor de transport.
- Sector Medical - fișe medicale ale pacienților - monitorizarea stării de sănătate a cetățenilor, utile pentru detectarea posibilelor amenințări de tip biologic.
- Internetul Lucrurilor (*Internet of Things*) - date provenite de la diverse aparate, precum: telefon, ceas, televizor, senzor de gaz, senzor de umiditate, camere video, etc. - utilizate la monitorizare activității individului cu scopul de a ușura anumite sarcini sau pentru a prevenii incidente.
- Sector industrial - rețele industriale de comunicații (senzori, magistrale de teren, rețele celulare), rapoarte economice - folosite pentru automatizare și îmbunătățirea produselor și a serviciilor.
- Sector bancar - tranzacții financiare, rapoarte - utilizate pentru detectarea fraudelor bancare, stabilirea ratelor la dobânzi, împrumuturi, schimb valutar, etc.

Toate aceste beneficii sunt importante pentru societatea din ziua de astăzi, companiile mari concurează pentru crearea de infrastructură și servicii pentru stocarea și procesarea datelor.

Exemple de servicii:

- Amazon Web Services - cel mai mare furnizor de servicii și infrastructură de tip *cloud* din lume (având peste 200 de soluții software).
- Microsoft Azure
- Google Cloud Platform

- IBM Cloud
- Oracle Cloud
- Alibaba Cloud

Capitolul 2

Rețele neuronale artificiale

2.1 Introducere

O rețea neuronală artificială este un model computațional inspirat din structura și modul de funcționare al creierului biologic. Conexiunile dintre neuronii artificiali se aseamănă sinapselor, fiecare neuron se conectează cu alt neuron prin intermediul unor muchii. Semnalul trimis prin aceste muchii este ponderat de niște parametri numiți **ponderi sinaptice** (*weights*). Mai mulți neuroni grupați formează un **strat** (*layer*), iar mai multe straturi formează o **rețea neuronală** (*neural network*).

Procesul de învățare presupune găsirea unor valori potrivite pentru ponderile sinaptice astfel încât procesarea semnalului de intrare să ofere rezultatul dorit.

2.2 Structură

Structura principală al unui neuron artificial este bazat pe modelul Perceptron-ului al lui Donald Hebb, modelul matematic fiind:

$$y = \varphi \left(\sum_{k=1}^n w_k * x_k + b \right),$$

unde x este vectorul de intrare (*input vector*), y vectorul de ieșire (*output vector*), w ponderea sinaptică (*weight*), b deplasarea (*bias*) și φ este funcția de activare sau transfer (*activation function*).

Vectorul de intrare este format din structuri de date, acestea putând reprezenta: imagini, frecvențe, etichete codificate, valori provenite de la senzori, etc. Ponderile sinaptice au rolul de a crește sau descrește puterea semnalului reprezentat de valorile vectorului de intrare. Funcția de activare preia semnalul ponderat și oferă o valoare specifică în baza acestuia. Deplasarea ajută la deplasarea semnalului ponderat pentru o mai bună aproximare, necesară pentru îndeplinirea anumitor condiții ale funcției de activare.

Exemplul 2.2.1 *Un neuron artificial care acționează precum o poarta logică SAU (OR)*

pentru două numere binare are forma:

$$y = \varphi(x_1 + x_2 - 0.5)z$$

unde $x = \{x_i | x_i \in \{0, 1\}\}$, $y \in \{0, 1\}$, $w_1 = 1$, $w_2 = 1$, $b = -0.5$, iar funcția de activare este:

$$\varphi(u) = \begin{cases} 1 & u \geq 0 \\ 0 & u < 0 \end{cases}$$

Verificare. Pentru $x = [1, 0]$, avem $u = 1 + 0 - 0.5 = 0.5$ și $y = \varphi(u) = \varphi(0.5) = 1$ (același rezultat și pentru $x = [0, 1]$ - datorită proprietății de comutativitate a adunării). Pentru $x = [1, 1]$, avem $u = 1 + 1 - 0.5 = 1.5$ cu $\varphi(u) = \varphi(2) = 1$. Ultimul caz pentru $x = [0, 0]$, vom avea $u = 0 + 0 - 0.5 = -0.5$ cu $\varphi(-0.5) = 0$.

Observația 2.1 Fără funcția de activare, perceptronul acționează precum o funcție liniară. Prin utilizarea unei funcții de activare potrivite, putem aborda mai ușor problemele neliniare, precum cele pentru clasificarea datelor în diverse categorii.

Un singur perceptron oferă doar o singură valoare de ieșire. Dacă dorim să avem mai multe valori de ieșiri trebuie să mai adăugăm perceptroni. Gruparea de neuroni artificiali se numește strat (*layer*).

Structura unui strat format din perceptroni arată astfel în formă matriceală:

$$\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ w_{n1} & w_{n2} & \cdots & w_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} \varphi(u_1) \\ \varphi(u_2) \\ \vdots \\ \varphi(u_n) \end{bmatrix}$$

Rezultatele acestui strat pot fi transmise către un alt strat care poate avea o altă funcție de activare, astfel putem crea modele matematice mai complexe. Această înșiruire de straturi se numește rețea (*network*). Straturile intermediare sunt deseori referite ca **straturi ascunse** (*hidden layers*). Iar o rețea cu foarte multe straturi ascunse poartă denumirea de **profundă** (*deep*) (Figura 2.1).

Rețele pot fi structurate și sub forma unui graf. Fiecare neuron fiind reprezentat de un nod, iar muchiile grafului sunt conexiunile dintre neuroni. Dacă graful suport nu conține cicluri, spunem că este uni-directional - o denumire uzuală peste acest tip de rețea este *feed-forward (FF)* (denumire pe care o vom folosi și în restul acestei lucrări). De asemenea, neuroni pot fi interconectați (graful suport conține cicluri), fapt care poate oferi rețelei

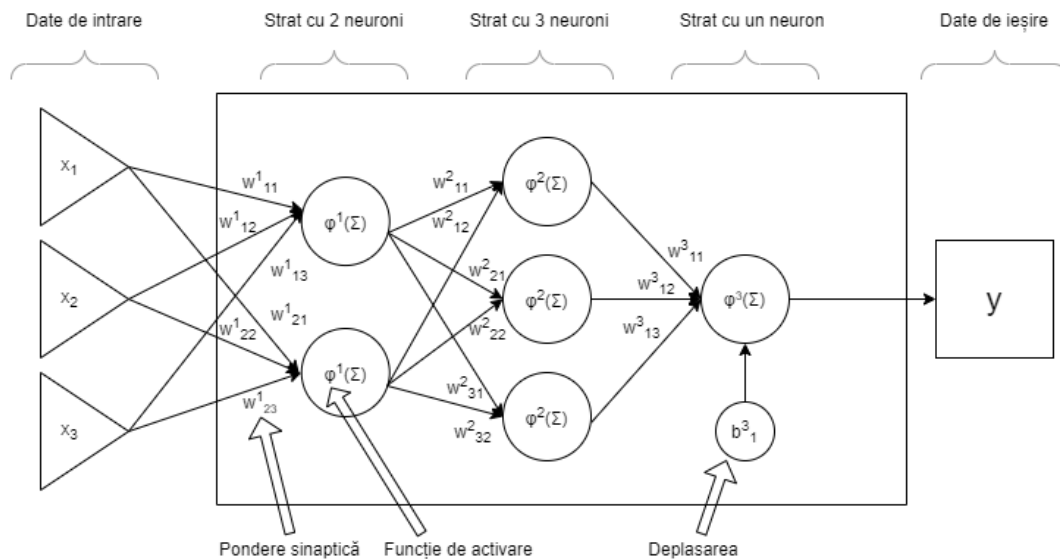


Figura 2.1: Structura unei rețele neuronale profunde

mai multă putere de modelare. Acest tip de rețea este denumit în general *recurrent neural network* (RNN)

Rețelele neuronale artificiale pot fi considerate ca fiind "aproximatori universali" [4]:

Rețelele feed-forward multistrat sunt, în condiții generale ale funcției de activare ascunsă, aproximatori universali dacă dispun de un număr suficient de unități asunse.

De-a lungul anilor, au fost create foarte multe tipuri de rețele neuronale artificiale pentru a servi la rezolvarea de probleme din domenii dificile.

Exemple de tipuri de rețele:

- Feed Forward (FF)
- Deep Feed Forward (DFF)
- Radial Basis Network (RBF)
- Recurrent Neural Network (RNN)
- Long/Short Term Memory (LSTM)
- Markov Chain (MC)
- Deep Convolutional Network (DCN)
- Deconvolutional Network (DN)

- Support Vector Machine (SVM)
- Deep Belief Network (DBN)

2.3 Funcții de activare și metode de optimizare

Funcția de activare ajută rețeaua neuronală pentru învățarea de tipare complexe aflate în setul de date analizat. Alegerea unei funcții de activare este critică pentru performanța rețelei, în special pentru cazul problemelor neliniare.

Unele din cele mai folosite funcții sunt:

$$\text{Identitate, Liniar} \quad \varphi(x) = x \quad (2.1)$$

$$\text{Binary Step} \quad \varphi(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases} \quad (2.2)$$

$$\text{Logistic, Sigmoid} \quad \varphi(x) = \frac{1}{1 + e^{-x}} \quad (2.3)$$

$$\text{Rectified liniar unit(ReLU)} \quad \varphi(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases} \quad (2.4)$$

$$\text{Softplus} \quad \varphi(x) = \ln(1 + e^x) \quad (2.5)$$

Funcțiile de activare pot fi aproape orice funcție liniară sau neliniară, dar unele (precum cele enumerate anterior) oferă mai multe beneficii decât altele în contextul antrenării unei rețele.

Crearea unei rețele neuronale artificiale presupune alegerea tipurilor de straturi din care să fie compusă împreună cu funcțiile lor de activare. La început, ponderile sinaptice sunt de cele mai multe ori alese aleatoriu. Cea ce face ca aproximarea oferită de rețea să nu fie una foarte bună.

Acest lucru duce la problema de optimizare a rețelei (găsirea unor valori potrivite pentru ponderile sinaptice) astfel încât rezultatul aproximării să fie unul satisfăcător. Acest proces este referit uzual ca *antrenare* (*training*). Antrenarea este unul dintre cele mai dificile capitole al acestui domeniu, alegerea unui algoritm potrivit poate beneficia extraordinar în privința găsiți unor valori optime pentru ponderile sinaptice.

Rețelele neuronale artificiale folosite în industrie (aplicații de recunoaștere a obiectelor în imagini; programe pentru traduceri și recunoașteri de voce) au o complexitate extraordinară atât din punct de vedere al arhitecturii, care poate consta dintr-un mix de diferite tipuri de rețele, cât și al nivelului imens de date (de ordinul milioanelor de terabiți). Antrenarea acestora poate dura câteva zile sau câteva luni, în cazuri rare fiind vorba de ani. Așadar, un algoritm de optimizare eficient are un rol crucial în acest proces.

Ca să putem optimiza trebuie să avem o metrice după care să evaluăm performanța, aceasta poartă de denumirea de **funcție de cost sau pierdere** (*cost/loss function*).

Aceasta se calculează folosind datele rezultate la rularea rețelei și cele pe care dorim să le avem.

În funcție de arhitectura rețelei, unele funcții de cost au o performanță mai bună decât celelalte. Câteva exemple:

1. Probleme de regresie
 - (a) Eroarea medie
 - (b) Eroarea pătratică medie
 - (c) Eroare medie absolută
2. Clasificări binare
 - (a) Binary Cross-Entropy
 - (b) Eroare Hinge
 - (c) Eroare pătratică Hinge
3. Clasificări multi-clasă
 - (a) Multi-Class Cross-Entropy
 - (b) Sparse Multiclass Cross-Entropy
 - (c) Kullback Leibler Divergence

Să presupunem că avem tabelul 2.1 cu rezultatele unei rețele neantrenate care încercă să aproximeze următoarea funcție:

$$f : [-1, 1] \rightarrow \mathcal{R}, f(x) = \frac{1}{x^2 + 1} \quad (2.6)$$

Date de intrare (x)	Rezultat estimat (\hat{y})	Rezultat dorit (y)
-1	-1.23	0.5
-0.5	0.5	0.8
0	3	1
0.5	-5	0.8
1	0.1	0.5

Tabelul 2.1: Valorile estimate ale unei rețele neantrenate pentru funcția $f(x)$ (2.6)

Aplicând eroare medie absolută și eroarea pătratică medie ca funcții de cost în tabelul 2.1, obținem următoarele rezultate din tabelul 2.2.

Rezultatele obținute din funcțiile cost le putem folosi pentru ajustarea ponderilor sinaptice și a deplasării. Algoritmii care ne ajută să minimizăm funcția de cost îi vom numi **algoritmi de optimizare**. Pentru rețele de tip *Deep Feed Forward (DFF)*, cea mai

Funcție de cost	Formulă	Valoarea erorii
Eroarea medie absolută	$\frac{1}{n} \sum_{i=1}^n y_i - \hat{y}_i $	2.04
Eroarea pătratică medie	$\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$	8.18

Tabelul 2.2: Valorile erorii ale unei rețele neantrenate pentru valorile din tabelul 2.1

comună metode de ajustare este cea a propagării inverse cu ajutorul unui gradient calculat folosind pierderile, unde valoarea erorii este transmisă în straturile interioare la fiecare neuron. Eroare este transmisă de la final până la început, iar în această tranziție valoarea ei se diminuează, cea ce face ca primele straturi să primească mai puțină informație. Aceasta este problema de diminuare a gradientului (*vanishing gradient problem*).

Optimizarea rețelei neuronale este unul din cel mai complexe capitole ale domeniului de învățare automată. Pentru lucrul cu rețele vom folosi biblioteca Tensorflow, care dispune de toate cele necesare. Algoritmii de optimizare prezenți în această bibliotecă sunt:

- Adam
- AdaDelta
- AdaGrad
- AdaMax
- Stochastic gradient descent (SGD)

În figura 2.2 putem observa rezultatul estimărilor aproximarea funcției f (2.6). Rețeaua este formată din 3 straturi: primul strat conține 32 de neuroni și folosește funcția de activare *ReLU* (2.4), al doilea are 16 neuroni și tot funcția ReLU, iar al treilea strat are doar un singur neuron (avem nevoie doar o singură valoare) cu funcția de activare identitate (2.1).

Pentru antrenarea rețelei vom folosi Adam și SGD. Setul de date e intrare va consta din 2000 de puncte din intervalul $[-1, 1]$, iar datele de comparație vor consta din valorile funcției f (2.6) în aceste puncte.

Analizând figura 2.3, putem observa că Adam are cea mai bună performanță față de SGD, acesta ajungând mai repede la starea optimă într-un mai mic de episoade/epoci. Figura 2.4 reprezintă rezultatul estimărilor pentru rețeaua neuronală care folosește algoritmul Adam.

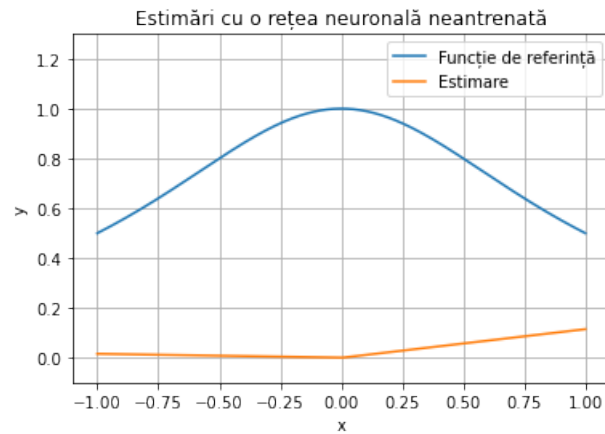


Figura 2.2: Estimarea unei funcții cu o rețea neantrenată

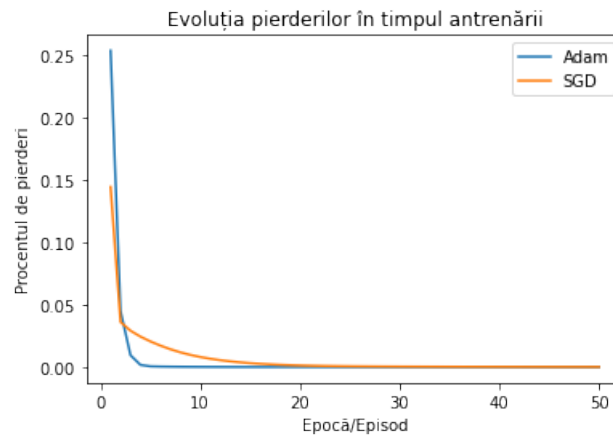


Figura 2.3: Comparația performanței a doi algoritmi de optimizare

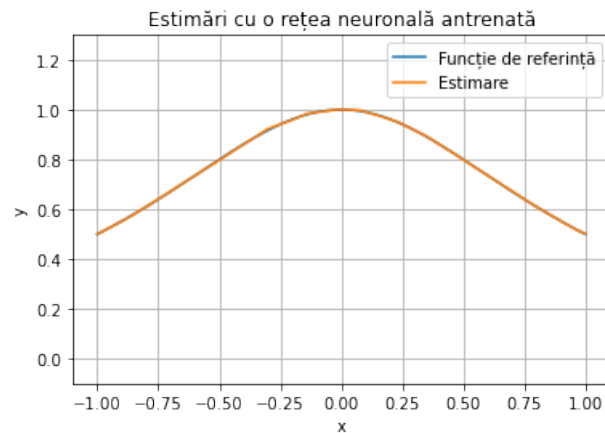


Figura 2.4: Estimarea unei funcții cu o rețea antrenată

Pentru a test cât de puternică este capacitatea de modelare a rețelei neuronale alese pentru funcția f , vom alege să aproximeze valorile unei suprafețe în loc de o simplă funcție unidimensională. Această suprafață are următoarea formulă:

$$g : \mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}, \quad g(x, y) = \frac{7xy}{e^{x^2+y^2}} \quad (2.7)$$

Datale de intrare vor fi alcătuite din perechi de tipul (x, y) , unde $x \in [-2, 2]$, $y \in [-2, 2]$. Intervalele lui x și y au fost împărțite fiecare în 500 de puncte echidistante. Numărul total de perechi este 250.000, iar datele de comparație vor fi numere asociate fiecare perechi, acestea reprezintă valoarea suprafeței în acel punct dat de pereche.

Uitându-ne în figura 2.5, observăm că rețeaua a avut succes în aproximarea rețelei. Dar cel mai important lucru este faptul că noi la început am făcut o rețea pentru un anumit tip de problemă și anume să aproximize valorile unei funcții unidimensionale, structura rețelei și a numărului de neuroni au fost date astfel încât să existe suficient de multă putere de modelare. Iar prin trecerea la estimarea unei suprafețe am putut demonstra că puterea rețelei poate să depășească domeniul pentru care aceasta fost construită inițial.

Această observație este extrem de importantă, deoarece acesta prezintă capacitatea mare de flexibilitate în modelare a valorilor unei probleme. Același model al rețelei putând să rezolve probleme din domenii diferite. Dacă am avea un o arhitectură care poate rezolva o secvență de numere date de un senzor, este posibil ca această să poată rezolva și o secvență de cuvinte. Posibilitățile sunt nelimitate, fapt care prezintă și interesul mare acordat de către companii și societățile academice.

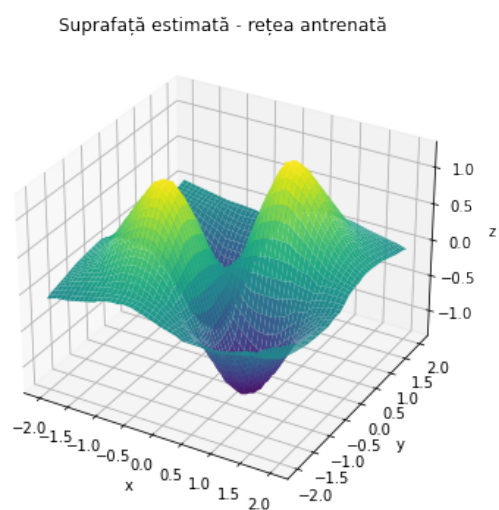
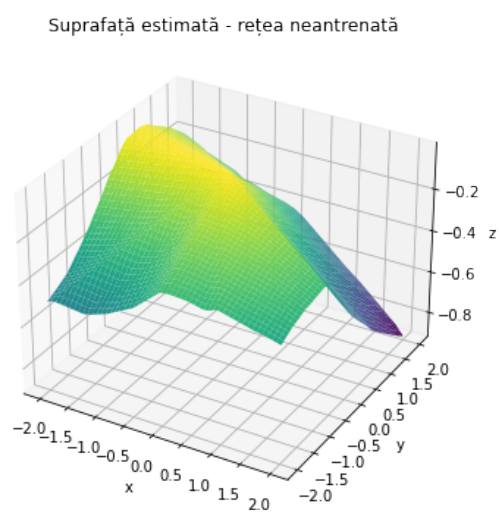
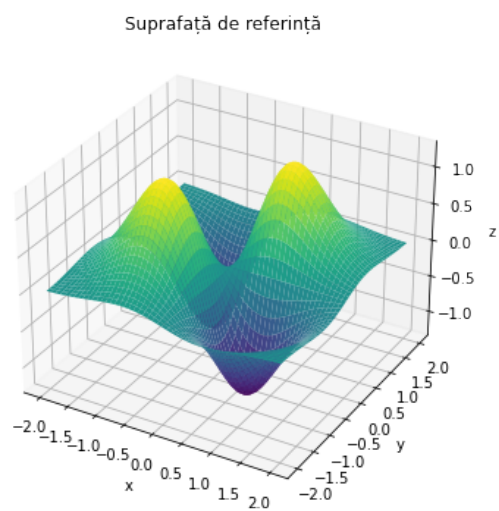


Figura 2.5: Estimarea unei suprafețe

2.4 Tensorflow

Tensorflow este o platformă dedicată dezvoltării modelelor de învățare automată. Acesta a fost creat inițial de către Google pentru a accelera dezvoltarea domeniului prin oferirea de programe ajutătoare pentru crearea rapidă a protipurilor. Datorită calității superioare a programelor de prototipare și a ușurinței de utilizare, acesta a devenit o platformă populară atât în mediul academic cât și industrial.

Datorită popularității, platforma a beneficiat de multe contribuții importante din partea marilor companii din domeniul IT și cel al semiconductoarelor, precum: PayPal, AMD, nVIDIA, Blomberg, Intel, IBM, Qualcomm, Uber, Arm, Twitter. De asemenea, platforma beneficiază de medii interactive de învățare, ideale pentru studenți sau profesioniști care doresc să dezvolte mici prototipuri de modele de învățare automată.

Exemple de programe care fac parte din platforma Tensorflow:

- Tensorflow Hub - biblioteca care găzduiește modele predefinite create de comunitate, precum: modele pentru clasificare imaginilor, analiza limbajului natural, generatoare de imagini
- Model Optimization - programe dedicate optimizării de modele
- TensorFlow Graphics - biblioteca care dispune de unelte pentru procesarea imaginilor
- TensorFlow Agents - biblioteca pentru dezvoltare agenților în cazul Învățării prin recompensă

În această lucrare vom folosi această bibliotecă pentru dezvoltarea unui model pentru agentul care va parcurge labirintul descris de către simulator. Unitatea de bază pentru lucrul cu aceasta este tensorul, reprezentat prin clasa **Tensor**. Câteva utilizări ale acestei clase pot fi observate în exemplul de cod 2.1.

```
1 // Ini
2 import * as tf from "@tensorflow/tfjs";
3 // Creare a doi tensori 1-dimensionali
4 const a = tf.tensor1d([1, 2, 3]);
5 const b = tf.tensor1d([4, 5, 6]);
6
7 // Afișarea lor în consolă
8 a.print(); // [1, 2, 3]
9 b.print(); // [4, 5, 6]
10 // Adunare a doi tensori
11 a.add(b).print(); // [5, 7, 9]
12
13 // Crearea unei constante
14 const k = tf.scalar(5);
15 // Înmulțirea cu o constantă
16 a.mul(k).print(); // [5, 10, 15]
17 // Produsul scalar
18 a.dot(b).print(); // 32
```

```

19 // Funcția de cost care folosește eroare medie pătratică
20 tf.losses.meanSquaredError(a, b).print(); // 9
21 // Crearea unui tensor 2 dimensional
22 const c = tf.tensor2d([
23     [15, 0],
24     [80, -30],
25 ]);
26 // Afișare
27 c.print(); // [[15, 0 ], [80, -30]]
28
29 // Ortogonalizare Gram-Schmidt
30 tf.linalg.gramSchmidt(c).print(); // [[1, 0 ], [0, -1]]
31 // Aplicarea funcției de activare de ReLU
32 c.relu().print(); // [[15, 0], [80, 0]]
33 // Creare unei noi matrici prin adunarea cu un număr
34 let d = c.add(125);
35 d.print()
36 // Extindere ultimei dimensiuni
37 // Tensorul devine unul 3-dimensional
38 d = d.expandDims(-1); // [[[140], [125]], [[205], [95 ]]]
39
40 // Afișarea formei dimensiunii
41 console.log(d.shape); // [2, 2, 1]
42 // Tensorul este acum de forma unei imaginii
43 // cu un singur canal de culoare
44 // Aplicare unei funcții de redimensionare a imaginii
45 tf.image.resizeBilinear(d, [4, 4]).print();
46 /*
47     [[140  ],
48      [132.5 ],
49      [125  ],
50      [125  ]],
51
52     [[172.5 ],
53      [141.25],
54      [110  ],
55      [110  ]],
56
57     [[205  ],
58      [150  ],
59      [95   ],
60      [95   ]],
61
62     [[205  ],
63      [150  ],
64      [95   ],
65      [95   ]]]
66 */

```

Listing 2.1: Exemple de folosire a bibliotecii Tensorflow

Având exemplul anterior ca demonstrare a unor funcții de bază din Tensorflow, o să

construim un mică rețea neuronală formată din doua straturi a câte 2 neuroni cu câte 2 ponderi sinaptice, fiecare strat având o altă funcție de activare, iar deplasarea va fi prezentă doar pentru primul strat. Datele de intrare vor fi reprezentate de un vector cu două valori.

```
1 // Inițiem un tensor cu datele de intrare
2 const x0 = tensor2d([[1], [2]]);
3 // Inițiem un 2 tensori cu câte 2 neuroni
4 // Fiecare linie reprezintă ponderile sinaptice al unui neuron
5 const w1 = tensor2d([
6     [2, 3],
7     [-3, 0],
8 ]);
9 const w2 = tensor2d([
10    [-1, 0.25],
11    [2, -0.8],
12 ]);
13 // Inițiem un tensor cu valorile pentru deplasare de la primul strat
14 // Al doilea îl vom omite
15 const b1 = tensor2d([[0.5], [-1]]);
16 // Facem sumarea semnalelor ponderate pentru fiecare neuron
17 // și adăugăm deplasarea
18 const y1 = tf.dot(w1, x0).add(b1);
19 // Afișăm rezultatul
20 y1.print(); // [[8.5], [-4 ]]
21 // Aplicăm funcție de activare Binary Step
22 const x1 = tf.step(y1);
23 x1.print(); // [[1], [0]]
24 // Rezultatul de primul strat îl folosesc ca date de intrare
25 // pentru al doilea
26 const y2 = tf.dot(w2, x1);
27 y2.print(); // [[-1], [2 ]]
28 // Aplic o altă funcție de activare și anume ReLU
29 const x2 = tf.relu(y2);
30 x2.print(); // [[0], [2]]
```

Listing 2.2: Exemple de folosire a bibliotecii Tensorflow

Capitolul 3

Metode de învățare

3.1 Lanț și proces Markov

La baza modelului de învățare pe care îl vom folosi în antrenarea agentului stau principiile fundamentale ale lanțului Markov și a procesului Markov. Proprietatea Markov afirmă că viitorul depinde numai de prezent și nu de trecut. Un lanț Markov este un model probabilistic care depinde numai de starea curentă pentru a prezice o stare viitoare. Așadar, un lanț Markov respectă proprietatea Markov. Trecerea de la o stare la alta se numește tranziție, iar probabilitatea ei poartă denumirea de probabilitate de tranziție.

De cele mai multe ori, lanțul Markov este reprezentat sub formă de graf orientat al cărui muchii reprezintă probabilitățile de tranziție dintre varfuri. Suma probabilităților de tranziție ale unui varf către celelalte varfuri este mereu 1.

În cazul agentului nostru, dacă ne-am imagina traseul ca fiind un lanț Markov, atunci pozițiile din traseu ar fi vârfurile grafului, iar mișcările agentului ar reprezenta muchiile. Putem asocia fiecărei mișcare o probabilitate, iar parcurgerea grafului reprezintă un posibil drum către obiectiv.

Așadar, dacă ne-am dori ca agentul să folosească această idee pentru stabilirea unui drum pentru rezolvarea obiectivului trebuie să stabilim o strategie de parcurgere a grafului. O strategie simplă ar fi una de tip *Greedy*, și anume agentul alege mereu acțiunea/muchia cu probabilitatea cea mai mare. Pentru ca această strategie să aibă succes trebuie ca valorile probabilităților acțiunilor să conducă către obiectiv în urma parcurgerii grafului.

Pentru a controla comportamentul agentului în mediul de lucru, vom folosi un sistem de recompense pentru fiecare decizie luată. Dacă dorim ca agentul să evite anumite situații precum luare de acțiuni care conduc la coliziuni cu anumite obiecte, vom asocia acestor decizii recompense negative. În contrast, dacă dorim ca agentul să facă anumite acțiuni care duc la îndeplinirea obiectivelor, acestor decizii li se vor asocia recompense pozitive.

Așadar, dorim ca agentul (pe parcursul simulării) să acumuleze cât mai multe recompense pozitive și să le evite pe cât mai mult posibil pe cele negative. Acest lucru îl vom numi **optimizare**, iar procesul de antrenament implică modificarea rețelei neuronale astfel încât acțiunile rezultate din datele de intrare să maximizeze recompensele acumulate.

Peste lanțul Markov putem construi un model matematic pentru modelarea procesului de decizie al agentului. Acest model conține următoarele elemente:

- Mulțimea stărilor (S)
- Mulțimea acțiunilor (A)
- Probabilitatea de tranziție dintr-o stare în alta pentru o acțiune ($P(s'|s, a)$)
- Recompensa primită în urma tranziției dintr-o stare în alta pentru o acțiune ($R(s'|s, a)$)
- Factor de atenuare, pondere care exprimă importanța recompenselor imediate și viitoare (γ)

Recompensele acumulate la fiecare pas de timp sunt exprimate de formula:

$$R_t = r_{t+1} + r_{t+2} + \dots, \quad (3.1)$$

unde r_{t+1} este recompensa primită prin efectuarea unei acțiuni la pasul t_1 . Așadar, R_t va reprezenta **câștigul** (*return*), pe care în mod natural dorim să-l maximizăm. Dar câștigul se poate întinde până la infinit, ca să rezolvăm această dilema a infinitului, vom introduce în formulă factorul de atenuarea (γ).

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \dots \quad (3.2)$$

$$= r_{t+1} + \gamma (r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} + \dots) \quad (3.3)$$

$$= r_{t+1} + \gamma R_{t+1} \quad (3.4)$$

$$(3.5)$$

Factorul de atenuare arată cât de importante sunt recompensele pentru efectuarea unei strategii cu câștig pe termen lung sau scurt. Acest factor ia valori între 0 și 1. Valorile mai apropiate de 0, dau mai multă prioritate acțiunilor care obțin recompense mari imediate, ignorând acțiunile mai slabe, dar care pot avea un câștig mare pe o perioadă mai lungă de timp. Pentru valorile mai apropiate de 1, acțiunile care duc la un câștig mai mare pe perioade lungi vor fi mai importante.

Dacă am avea un agent pe post de vânzător, un factor de atenuare mic ar face ca agentul să vândă produsele cât mai repede la prețul curent cel mai mare. Un factor de atenuare mare ar schimba complet strategia, și anume că agentul va aștepta să vândă la cel bun preț posibil dintr-o perioadă de timp. Dacă am face o lichidare de stoc, atunci vânzarea produselor cât mai rapidă ar avea prioritatea, deci agentul ar trebui să aleagă recompensele imediate. Pentru vânzări de produse care își cresc valoarea în timp; precum operele de artă, imobiliare, titluri financiare (acțiuni, obligațiuni) - am dori ca agentul să aștepte un preț bun de vânzare.

Pentru alegerea unei acțiuni, vom stabili o **strategie** π (*policy*), care este o funcție ce returnează acțiunea pe care trebuie să o ia agentul la o anumită stare. Așadar, avem $\pi : S \rightarrow A, \pi(s) = a$ pentru $s \in S, a \in A$.

Fiecarei stări îi putem asocia o **funcție valoare** (*function value*) $V(s)$, valoarea dată de aceasta ne va arata cât de bună este starea în care se află agentul. Pentru o strategie π , aceasta are forma:

$$V^\pi(s) = E_\pi [R_t | s_t = s] \quad (3.6)$$

$$= E_\pi [r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s, s_{t+1} = s'] \quad (3.7)$$

$$= E_\pi [r_{t+1} | s_t = s] + \gamma E_\pi [V^\pi(s_{t+1}) | s_t = s] \quad (3.8)$$

$$= E_\pi [r_{t+1} | s_t = s] + \gamma V^\pi(s) \quad (3.9)$$

Prin urmare, $V^\pi(s)$ va fi câștigul așteptat când se începe din starea s , iar acțiunile sunt date de către strategia π .

Să presupunem că avem un magazin și dorim să stabilim prețul unui produs pentru o anumită zi. Definim starea ca fiind perechea dintre zi și preț, $S = \{s | s = (zi, preț)\}$. Având o strategie π aleatoare, pentru ziua a 5-a, avem următoarele opțiuni descrise în tabelul 3.1.

Preț (lei)	Stare (s)	Câștigul așteptat ($V^\pi(s)$)
15	$s(5, 15)$	0.3
8	$s(5, 8)$	1.6
3	$s(5, 3)$	6.9
1	$s(5, 1)$	10

Tabelul 3.1: Recompensele așteptate pentru fiecare preț

Prin scurta analiză a tabelului, observăm că agentul decide că prețul de un leu pentru acel produs va fi cel mai bun. Dacă am fi luat în considerare și alte valori pentru reprezentarea stării, valori precum: stocul disponibil, rată de vânzare zilnică, și am avea aceleași valori pentru $V^\pi(s)$, putem deduce faptul că agentul sugerează să avem o lichidare de stoc, având în vedere că a ales cel mai mic preț.

3.2 Q-Learning

Noi dorim ca agentul să ia cele mai bune decizii pentru îndeplinirea sarcinilor, deci am vrea să găsim o strategie optimă π^* și funcția valoare optimă V^* care ne oferă cel mai mare câștig dintre toate celelalte. Metodele de găsire sunt inspirate din principiul de optimalitatea al lui Richard E. Bellman [1], care afirmă că: “Strategia optimă are proprietatea că indiferent de starea sau decizia inițială, deciziile care rămână trebuie să formeze o strategie optimă în ceea ce privește starea rezultată din prima decizie”.

$$V^*(s) = \max_\pi V^\pi(s) \quad (3.10)$$

Pe lângă funcția valoare V , vom avea și funcția Q care indică cât de bună este o acțiune pentru o stare având strategia π . Aceasta este de forma:

$$Q^\pi(s, a) = E_\pi [R_t | s_t = s, a_t = a] \quad (3.11)$$

Luând exemplul din tabelul 3.1, putem exemplifica valorile Q prin adaugarea acțiunilor: creștere preț (\uparrow), scădere preț (\downarrow) și păstrare preț ($=$). De asemenea, punem și condiția ca prețul să nu fie mai mic decât un leu, pentru această situație nu dorim ca produsul să ajungă să fie gratuit.

Preț (lei)	Stare (s)	Acțiune(a)	Valoare Q ($Q^\pi(s, a)$)
8	$s(5, 8)$	\uparrow	-30
8	$s(5, 8)$	$=$	-10
8	$s(5, 8)$	\downarrow	0
3	$s(5, 3)$	\uparrow	-10
3	$s(5, 3)$	$=$	25
3	$s(5, 3)$	\downarrow	35
1	$s(5, 1)$	\uparrow	-30
1	$s(5, 1)$	$=$	100
1	$s(5, 1)$	\downarrow	-120

Tabelul 3.2: Valorile Q pentru fiecare preț și acțiune

Din tabelul 3.2 se poate evidenția mai detaliat alegerea prețului de 1 leu, aceasta având cea mai mare valoare. De asemenea, se pot observa valori negative la ridicarea prețului, odată coborât prețul, agentul are șanse mici să-l crească. Dacă agentul ar fi pornit cu prețul din ziua a 4-a, presupunând că ar fi cel de 8 lei, în ziua a 5-a l-ar fi coborât până la un leu dacă strategia noastră este să alegem valoarea Q cea mai mare atunci când decidem ce acțiune vom lua. De menționat faptul că acțiunea de coborâre a prețului sub un leu are o valoare extrem de mică, asta datorită condiției impuse la început. Agentul este complet descurajat să ia acea acțiune. Această tactică de descurajare o vom folosi și în aplicație pentru rezolvarea labirintului. Acțiunile care fac agentul să intre în obstacole vor avea asociate recompense negative.

Pentru estimarea acestor Q valori vom folosi învățarea bazată pe diferențe temporale. Pentru funcția valoare V , aceasta arată astfel:

$$V^\pi(s_t) = V^\pi(s_t) + \alpha * (R_t - V^\pi(s_t)) \quad (3.12)$$

$$= V^\pi(s_t) + \alpha * (r_{t+1} + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)) \quad (3.13)$$

unde s_t și r_t sunt starea și respectiv recompensa la pasul t , α este o constantă numită în general **rata de învățare** (*learning rate*) și preiau valori din intervalul 0 și 1, similar cu

factorul de atenuare. $R_t = r_{t+1} + \gamma V^\pi(s_{t+1})$ reprezintă noua informație, formată din recompensa imediată primită la efectuarea acțiunii și câștigul așteptat pentru următoare stare. Diferența dintre aceasta și valoarea veche reprezintă valoarea învățată, iar înmulțirea cu constanta α ne arată cât preluăm din aceasta nouă valoare.

În mod similar, aplicăm și pentru valoarea Q :

$$Q^\pi(s_t, a_t) = Q^\pi(s_t, a_t) + \alpha * (R_t - Q^\pi(s_t, a_t)) \quad (3.14)$$

$$= Q^\pi(s_t, a_t) + \alpha * (r_{t+1} + \gamma Q^\pi(s_{t+1}, a_t) - Q^\pi(s_t, a_t)) \quad (3.15)$$

De asemenea, definim și strategia ε -greedy pe care vom folosi în aplicația din această lucrare, care are următoarele caracteristici:

- Acțiune din care rezultă cel mai mare câștig estimat este selectată.
- Având probabilitatea ε , se alege o acțiune în mod aleatoriu, neglijând estimările pentru recompense.
- Pentru echilibrul dintre fazele de explorare și exploatare a mediului, ε va începe la începutul simulării cu o valoare mare, iar pe parcurs, o micșorăm până la un minim stabilit.

Algoritmul Q-Learning este format din următori pași:

1. Se inițializează $Q(s, a)$ în mod aleatoriu
2. Inițiere episod
 1. Inițializare stare s
 2. Repetă
 1. Alegem acțiunea a folosind strategia aleasă (presupunem ca este ε -greedy)
 2. Executăm acțiunea a și primim recompensa r și următoarea stare s'
 3. Actualizăm valoarea Q a stării curente în funcție de noile informații:

$$Q^\pi(s, a) \leftarrow Q^\pi(s, a) + \alpha [r(s, a) + \gamma \max_{a'} Q^\pi(s', a') - Q^\pi(s, a)] \quad (3.16)$$
 4. Trecem la următoarea stare: $s = s'$
3. Repetăm pașii până când se ajunge la o stare terminală
3. Repetăm pașii atât timp cât nu am ajuns la episodul final

Pentru formula 3.16, $\max_{a'} Q^\pi(s', a')$ este recompensa maximă ce poate fi obținută în starea s' care urmează stării actuale s (recompensa dacă se ia cea mai bună acțiune apoi).

Capitolul 4

Aplicație

4.1 Introducere

Pentru aplicația în care vom implementa algoritmul de învățare ne dorim să avem disponibile următoarele funcționalități:

- Vizualizare grafică pentru mediul simulat - Afișarea labirintului sub forma unei poze
- Grafice pentru analizelor datelor opțiune din timpul sesiunilor de antrenament
- Elemente interactive care să ne permită modificare de parametri interni în algoritmi
- Tabele care să afișeze informații/date care urmează să fie procesate și rezultatele acestora

Având în vedere cerințele menționate anterior, platforma care ne poate permite în o dezvoltare rapidă asupra unei interfeței interactive foarte bogate este cea a aplicațiilor destinate web-ului.

Pentru contruirea aplicației vom folosi următoarele tehnologii:

- Svelte - program pentru construirea aplicațiilor web care va reprezenta interfața interactivă dedicată utilizatorului ([10])
- Tensorflow.js - bibliotecă dedicată pentru contruirea și antrenarea rețelelor neuronale pentru aplicații web ([12])
- Echarts - bibliotecă pentru contruirea de graficelor dedicate vizualizării de date ([11])
- Konva - bibliotecă grafică pentru generarea imaginilor ([13])

Elementele de bază vor fi construite folosind Svelte, acestea fiind: butoane, câmpuri de introducere a datelor, tabele pentru prezentarea informațiilor, containere pentru poziționare. Construirea labirintului se va face prin intermediul lui Konva, care ne permite și modificare rapidă a imaginii în funcție de starea internă a simulatorului. Datele provenite din

sesiunile de antrenament vor fi afișate imediat după fiecare sesiune cu ajutorul graficelor generate de Echarts.

Algoritmii de învățare vor fi construiți folosind funcțiile oferite de către Tensorflow, acestea oferind următoarele capabilități: crearea de straturi cu diverse funcționalități, funcții de activare, algoritmi de optimizare, metode utilitare de salvare și de creare a unor modele secvențiale formate din mai multe straturi de neuroni.

4.2 Structură

Labirintul este de forma unei matrici, fiecare celulă îndeplinește un anumit rol: drum, obstacol, ieșire, etc. Clasa principală dedicată definirii structurii de reprezentare a labirintului este denumită **Board**. Această clasă definește mediul simulat și interacțiunile disponibile pentru agent.

Pentru codificare vom avea următoarele reguli: spațiul liber va avea cifra 1, un obstacol cifra 2, iar pentru ieșire cifra 3. Pentru pozițiile agentului, la codificarea celulei de matrice se va adauga prefixul 1. Exemple de definiri ale mediului simulat prin codificare se pot observa în tabelul 4.1.

11	1	2	1	1	2	1	1	2	1	1	2	1	1	2
1	2	2	11	2	2	1	2	2	1	2	2	1	2	2
1	1	3	1	1	3	11	1	3	1	11	3	1	1	13

Tabelul 4.1: Exemple de codificări ale structurii labirintului

Sunt disponibile patru acțiuni pe care agentul poate să le ia: sus, jos, stânga și dreapta. Ca o acțiune să fie validă, aceasta trebuie să îndeplinească următoarea condiție: acțiunea nu trebuie să facă agentul să iasă din labirint atunci când se află la margine. Dacă se întâmplă acest lucru, clasa **Board** va pastra poziția agentului și va transmite faptul că mutare este invalidă celorlalte componente, astfel încât acestea să poată stabili o recompensă ca pe viitor agentul să evite astfel de situații.

Reprezentarea labirintului sub formă de imagine este dată de clasa **BoardUI**. Aceasta are rolul să creeze reprezentarea vizuală a labirintului folosind datele furnizate de către clasa **Board**. Această reprezentare poate fi observată în figura 4.1. Agentul este prezentat sub forma unui cerc albastru, spațiul liber sub forma unui pătrat gri, iar obiectivul cu un pătrat violet.

Clasă **BoardUI** este legată de clasa **Board** printr-un sistem reactiv de notificare, astfel încât orice schimbare care duce la modificarea stării labirintului (exemplu: miscarea agentului) se propaga imediat către aceasta, astfel imaginea este modificată imediat cu noile date (figura 4.2).

Toate aceste imagini sunt generate folosind biblioteca Konva care ne permite atât generarea imaginii pentru afișare în pagina web cât și opțiune de interacțiune cu browserul

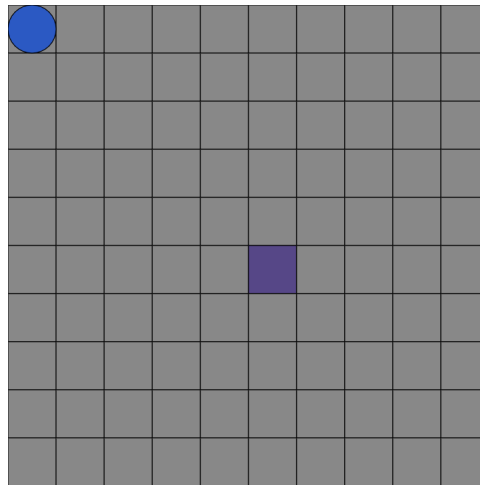


Figura 4.1: Reprezentarea vizuală a labirintului în starea inițială

care ne permit să transmitem evenimentele date de mouse (exemplu: click) către clasa **Board**.

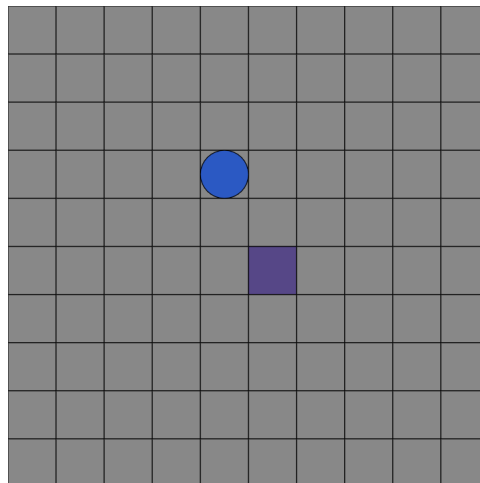


Figura 4.2: Reprezentarea vizuală a labirintului după o serie de acțiuni ale agentului

4.3 Simulator

Simulatorul acționează ca o interfață între agent și mediul reprezentat de labirint. Acesta are rolul să furnizeze informații către agent, precum: codificarea curentă a labirintului sau reprezentarea sa sub formă de imagine; dacă simularea este terminată; recompensa pentru fiecare acțiune luată.

Clasa **Env** este cea care definește structura modului de acționare a simulatorului. Acesta definește trei funcții principale, iar principiul lor de funcționare este inspirat după

standardul definit de către OpenAi Gym [3]. Așadar avem următoarele funcții:

- `reset` - această funcție aduce mediul simulat la starea inițială (figura 4.1) și returnează această stare sub forma dorită (codificare sau imagine).
- `step(acțiune)` - aceasta transmite acțiunea luată de agent către clasa **Board**, după acțiunea este procesată, clasa **Board** transmite înapoi valoare recompensei al acelei acțiuni dacă este validă, altfel doar anunța invaliditatea. După preluarea rezultatului, se decide care este recompensa și dacă simularea s-a încheiat în urma aceste acțiuni. Valorile noi stării, a recompensei și a semnalului de terminare sunt returnate la finalul evaluării.
- `actionSample` - această funcție returnează o acțiune aleatorie disponibilă în mediul simulat

```
1 class Env {
2     ACTIONS = ['UP', 'DOWN', 'RIGHT', 'LEFT']
3     invalidState = false
4     /**
5      *
6      * @param {Board} board
7      */
8     constructor(board) {
9         this.board = board
10    }
11
12    //
13    setAgentStartPosition(pos) {
14        this.board.playerDefaultPos = pos
15    }
16
17    //
18    step(action) {
19        this.invalidState = !this.board.move(this.ACTIONS[action])
20        return this.board.getBoardState(), this._getReward(), this._isDone
21    }
22
23    //
24    reset() {
25        this.board.playerReset()
26        return this.board.getBoardState()
27    }
28
29    //
30    actionSample() {
31        return Math.floor(Math.random() * this.ACTIONS.length)
32    }
33 }
```

```

34 //
35 _getReward() {
36     return (this.invalidState && !this.board.isOnExit() && -100) ||
37     this.board.getPlayerCellValue()
38 }
39 //
40 _isDone() {
41     return this.board.isOnExit() || this.invalidState
42 }
43 //
44 clone() {
45     return new Env(this.board.clone())
46 }
47 }
48 }

```

Listing 4.1: Definirea clasei Env

4.4 Agent

Clasa **Agent** definește structura internă a rețelei neuronale. Pentru construirea rețeleor neuronale folosim biblioteca Tensorflow care ne oferă o multitudine de funcții pentru definirea straturilor, a funcțiilor de activare și diverse metode de manipulare a datelor. Funcțiile din Tensorflow lucrează în principal cu valori reprezentate prin tensori definiți de clasa **Tensor** a bibliotecii. Monitorizarea numărului de tensori creați de-a lungul sesiunilor de antrenament ale agentului are o deosebită importanță în detectarea problemelor legate de consumul de resurse al calculatorului.

Funcțiile principale ale clasei sunt:

- predict (date de intrare) - aceasta ne oferă rezultatul din urma procesării datelor de intrare oferite ca parametru
- fit(date de intrare, date de ieșire) - aplică algoritmul de optimizare astfel încât rezultatul datelor de intrare să fie cât mai aproape de rezultatul dorit definit în datele de ieșire date ca parametru

Pentru crearea rețelei neuronale vom folosi funcția **sequential** care cere ca parametri o listă de straturi definite de pachetul **layers**. Cel mai important strat este cel de tip **dense**, care reprezintă structura de baza al unui strat definit în capitlul 2. Acesta este compus din ponderi sinaptice și opțional deplasarea și funcția de activare. Un exemplu de definire al unei simple rețele neuronale cu un singur strat și un singur neuron, fără funcție de activare și deplasare (bias), care acceptă un vector de valori de lungime 3 arată astfel:

```

1 import * as tf from '@tensorflow/tfjs';
2
3 const rn = tf.sequential();

```

```
4 rn.add( tf.layers.dense({ units: 1, inputShape: [3], useBias: false }) );
```

Listing 4.2: Exemplu de creare a unei rețele neuronale simple

Funcția de activare nu a fost specificată, iar deplasarea (bias) a fost dezactivată, prin urmare la evaluare vectorului se va afișa produsul scalar dintre vectorul cu datele de intrare și vectorul ponderilor sinaptice. Formula arată astfel:

$$rezultat = \langle (w_1, w_2, w_3), (x_1, x_2, x_3) \rangle = w_1 * x_1 + w_2 * x_2 + w_3 * x_3$$

Pentru evaluarea vectorului vom folosi funcția `predict` definită de `sequential`. Este important să reținem faptul că `predict` trebuie să primească un vector de date de intrare, iar rezultatul său este un vector de date de ieșire. Așadar, vectorul nostru de 3 elemente trebuie pus într-un alt vector, care în final arată precum o matrice care va fi transformat într-un tensor 2-dimensional prin folosirea funcției `tensor2d`. Biblioteca Tensorflow lucrează în principal cu tensori, așa că este foarte important să transmitem datele în formatul corect. Dacă am fi avut o matrice ca date de intrare, atunci ar fi trebuit să creăm un tensor 3-dimensional. Inițierea valorilor ponderilor sinaptice este aleatoare, și pentru vizualizarea lor punem în datele de intrare o valoare cu 1 și restul cu 0.

```
1 // Afișează valoarea primei ponderi sinaptice
2 rn.predict(tf.tensor2d([[1, 0, 0]])).print(); // [[1.1913936],]
3 // Afișează valoarea pentru a doua pondere sinaptică
4 rn.predict(tf.tensor2d([[0, 1, 0]])).print(); // [[0.665537],]
5 // Afișează valoarea pentru a treia pondere sinaptică
6 rn.predict(tf.tensor2d([[0, 0, 1]])).print(); // [[-0.9657576],]
7 // Afișează suma celor trei ponderi sinaptice
8 rn.predict(tf.tensor2d([[1, 1, 1]])).print(); // [[0.891173],]
```

Listing 4.3: Exemplu de evaluare a unei simple rețele neuronale

Se poate observa că pe linia 8 se va afișa suma valorilor de pe liniile 2, 4, 6. Vom adauga o funcție de activare și anume ReLU (*Rectified Linear Unit*), descrisă în capitolul 2. Această funcție de activare transformă valorile negative în valoarea 0, astfel având un rol de filtrare a valorilor negative. Exemplu:

```
1 const rn = tf.sequential();
2 rn.add( tf.layers.dense({ units: 1, inputShape: [3], activation: "relu",
   useBias: false }) );
3 rn.predict(tf.tensor2d([[1, 0, 0]])).print(); // [[0.1161228],]
4 rn.predict(tf.tensor2d([[0, 1, 0]])).print(); // [[0],]
5 rn.predict(tf.tensor2d([[0, 0, 1]])).print(); // [[0.3614732],]
6 rn.predict(tf.tensor2d([[1, 1, 1]])).print(); // [[0],]
```

Listing 4.4: Exemplu de evaluare a unei simple rețele neuronale cu funcție de activare

Se observă faptul că a doua pondere sinaptică are o valoare negativă față de celelalte două, iar din linia 6, reiese faptul că aceasta este mult mai mare decât suma celorlalte două, ceea ce duce ca rezultatul final să fie 0. Interpretarea valorilor obținute este un factor important în înțelegerea rețelei neuronale. Din exemplul anterior, a doua pondere sinaptică

are o influență mult mai mare asupra rezultatului decât restul. Dacă dorim ca aceasta să aibă o influență mult mai mică, trebuie să folosim procedeul de antrenare a rețelei care ne va calibra valorile acestor ponderi sinaptice, astfel încât să obținem o valoare finală mult mai aproape de cea ce dorim.

Pentru antrenarea rețelei neuronale vom folosi funcția `fit` din `sequential`. De asemenea, trebuie să alegem și o un optimizer care va fi `adam`, iar pentru calculul erori vom folosi metoda celor mai mici pătrate (mean square error). Vom folosi modelul anterior pentru exemplificare și vom antrena rețeaua astfel încât prima pondere sinaptică să fie cât mai aproape de 0 ca valoare.

```

1 const rn = tf.sequential();
2 rn.add( tf.layers.dense({ units: 1, inputShape: [3], activation: "liniar",
   useBias: false }) );
3 rn.predict(tf.tensor2d([[1, 0, 0]])).print(); // [[0.1678354],]
4 // Adaug optimizerul și funcția pentru calculul erorii
5 rn.compile({ loss: "meanSquaredError", optimizer: "adam" });
6 // Creez o funcție de antrenare care va încerca să modifice prima pondere
7 // sinaptică în decursul a 500 de episoade/iterații
8 (async () => {
9   for (let i = 0; i < 500; ++i) {
10     await rn.fit(tf.tensor2d([[1, 0, 0]]), tf.tensor2d([[0]]));
11   }
12   // Afișez rezultatul antrenamentului
13   rn.predict(tf.tensor2d([[1, 0, 0]])).print(); // [[0.0001513],]
14 })();

```

Listing 4.5: Exemplu de antrenare a unei simple rețele neuronale cu funcție de activare

Având codurile anterioare ca exemplificare a funcțiilor din biblioteca `Tensorflow`, putem construi structura și modul de funcționare a clasei `Agent`.

```

1 import * as tf from '@tensorflow/tfjs';
2
3 class Agent {
4   constructor(model) {
5     /**
6      * Inițiez rețeaua neuronală
7      * @type {tf.Sequential}
8      */
9     this.model = model || this.buildModel()
10   }
11
12   // Funcție care construiește rețeaua neuronală
13   buildModel() {
14     const model = tf.sequential() // Creare rețea
15     // Adaug primul strat care va primi pixelii pixeli imaginii
16     // labirintului
17     model.add(tf.layers.dense({ units: 25, inputShape: [25, 25],
18     activation: 'relu' })))
19     // Adaug un strat intermediar care îmi înlocuiește unele
20     // de date de intrare pentru următorul strat cu valoarea 0

```

```

19     model.add(tf.layers.dropout({ rate: 0.4 }))
20     // Un strat ascuns care va suma pe fiecare linie rezultatele
    anterioare
21     model.add(tf.layers.dense({ units: 1, activation: 'relu' }))
22     // Un strat intermediar care îmi va reduce dimensiunea, astfel din
    matrice toate
23     // toate datele de intrare devin un singur vector
24     model.add(tf.layers.flatten())
25     model.add(tf.layers.dropout({ rate: 0.2 }))
26     // Stratul final care ne oferă rezultatul sub forma unui vector de
    4 elemente
27     // ele reprezentând valoarea acțiunilor pentru stare dată
28     model.add(tf.layers.dense({ units: 4, activation: 'linear' }))
29     // Adaug optimizatorul și funcția de calcul a erorii
30     model.compile({ loss: 'meanSquaredError', optimizer: 'adam',
    metrics: ['accuracy'] })
31     return model
32 }
33
34 // Funcție de antrenare
35 async fit(input, output) {
36     await this.model.fit(input, output, { epochs: 1 })
37 }
38
39 // Funcție care evaluează datele de intrare
40 predict(input) {
41     return this.model.predict(input.expandDims(0))
42 }
43
44 // Funcție care evaluează datele de intrare și returnează
45 // poziția pentru cel mai mare element din datele de ieșire procesate
46 getAction(input) {
47     return tf.tidy(() => {
48         const result = this.predict(input)
49         return tf.argmax(result, 1).arraySync()[0]
50     })
51 }
52 }
53
54 export default Agent

```

Listing 4.6: Structura clasei Agent

4.5 Model de învățare

Pentru implementarea modelului de învățare vom folosi algoritmul Q-Learning descris în capitolul 3. Vom avea nevoie de 3 clase principale: **Agent**, **Env** și **Memory**.

Clasa **Memory** va fi cea care va păstra rezultatele experiențelor pe care agentul le va produce de-a lungul sesiunii de antrenament. Memoria va consta dintr-o listă de lungime

data în care se vor adauga experiențele, dacă se va depăși capacitatea, atunci experiențele vechi vor fi șterse. Structura clasei `Memory` arată astfel:

```
1 class Memory {
2   constructor(capacity, cleanFun) {
3     // Setare capacitate maximă
4     this.capacity = capacity || 5000
5     // Inițiere listă
6     this.experiences = []
7     // Setare funcție de curățare a memoriei fizice pentru valorile
    care vor fi distruse
8     this.cleanFun = cleanFun
9   }
10
11  // Funcție care adaugă o experiență în listă
12  add(exper) {
13    // Verific dacă am depășit capacitatea
14    if (this.experiences.length + 1 > this.capacity) {
15      // Scot elementul vechi din listă
16      const exper = this.experiences.shift()
17      // Curăț elementul din memoria fizică
18      this.cleanFun?.(exper)
19    }
20    // Adaug nouă experiență
21    this.experiences.push(exper)
22  }
23
24  // Preiau o serie fixă de experiențe amestecate aleatoriu din lista
25  sample(batch) {
26    // Amestec toate experiențele
27    const randomExperiences = Memory.shuffle([...this.experiences])
28    // Preiau primele experiențe ca serie
29    return randomExperiences.slice(0, batch)
30  }
31
32  // Golosesc toată lista de experiențe
33  clean() {
34    // Aplic funcție de curățare a memorie fizice
35    // că să nu am posibile reziduri
36    this.experiences.forEach(exper => {
37      this.cleanFun?.(exper)
38    })
39    this.experiences = []
40  }
41
42  // Amestec o copie a unui vector dat folosind algoritmul Fisher-Yates
43  static shuffle(array) {
44    // Inițiere variabile
45    let m = array.length, t, i;
46    // Cât timp mai sunt elemente de amestecat
47    while (m) {
48      // Iau o poziție aleatorie
```

```

49         i = Math.floor(Math.random() * m--);
50         // Fac schimb de poziții cu elementul din poziția m
51         t = array[m];
52         array[m] = array[i];
53         array[i] = t;
54     }
55     return array;
56 }
57 }
58
59 export default Memory

```

Listing 4.7: Structura clasei Memory

Toate aceste piese vor fi folosite în clasa **Trainer**, pentru crearea programului final de implementare al modelului de învățare. Această clasa aplica algoritmul de Q-Learning (descriș în capitolul 3) pentru antrenarea rețelei neuronale al agentului descrișă de clasa **Agent**.

```

1  class Trainer {
2      totalEnvs = 2
3      /**
4       * Inițiere componente
5       * @param {Env} env
6       * @param {Agent} agent
7       * @param {Memory} memory
8       */
9      constructor(env, agent, memory) {
10         // Setare memorie
11         this.env = env
12         // Setare agent
13         this.agent = agent
14         // Setare memorie
15         this.memory = memory
16         // Inițiere listă de simulatoare
17         this.envs = [{ id: 1, env: this.env }]
18     }
19
20     async train(episodes = 150, cb = () => { }) {
21         const discount = 0.985; // Factor de atenuare
22         // const lr = 0.1
23         let epsilon = 1 // Probabilitatea unei acțiuni aleatoare
24         const epsilon_min = 0.0 // Probabilitatea minimă a unei acțiuni
25         aleatoare
26         const epsilon_decay = (epsilon - epsilon_min) / episodes // Rata de
27         scădere a probabilității
28         const maxIterations = 75 // Numărul de iterații maxime
29
30         // Simulări episod
31         for (let eps = 1; eps <= episodes; eps++) {
32             const t0 = performance.now() // Timpul de incepere

```



```

32     const rewardsAnaly = {} // Obiecte cu date de tip analitic
33     // Rulare simulării unui episod în fiecare simulator în
nregistrat în listă
34     await Promise.all(this.envs.map(async ({ id, env }) => {
35         // Re-inițiere simulator și preluarea stării de început
36         let state = await env.reset()
37
38         // Rulare simulare
39         for (let iter = 0; iter < maxIterations; iter++) {
40             // Alegerea acțiunii
41             const action = Math.random() < epsilon ? env.
actionSample() : this.agent.getAction(state)
42             // Procesarea acțiunii și colectarea rezultatului
43             const [nextState, reward, done] = await env.step(action
)
44             // Salvare în memorie
45             this.memory.add({ state, nextState, reward, done,
action })
46
47             // Sumarea recompenselor adunate pe parcursul
episodului
48             rewardsAnaly[id] = rewardsAnaly[id] ? rewardsAnaly[id]
+ reward : reward
49             // Oprește simulare în cazul semnalului de stop
50             if (done) {
51                 break
52             }
53             // Preluare stării viitoare
54             state = nextState
55         }
56     })))
57
58     const tData = performance.now() // Timpul de începere a procesă
rii de date
59
60     // Alegerea a 100 de experiențe aleatoare și procesarea lor
61     const trainData = this.memory.sample(100).filter(exper => !
exper.state.isDisposed && !exper.nextState.isDisposed).reduce((acc,
exper) => {
62         return tf.tidy(() => {
63             // Preiau datele din experiență
64             const { nextState, reward, done, state, action } =
exper
65             // Calculez valoare Q pentru viitoare stare
66             const nextQ = (this.agent.predict(nextState).arraySync
())[0]
67             // Calculez valoarea Q curentă
68             const newCurrentQ = (this.agent.predict(state).
arraySync())[0]
69             // Aplic ecuația Bellman
70             newCurrentQ[action] = done ? reward : reward + discount

```

```

    * Math.max(...nextQ)
    // Salvez rezultatele
    acc.states.push(state); acc.newQValues.push(newCurrentQ)
    return acc
  })

  }, { states: [], newQValues: [] })

  const tTrain = performance.now() // Timpul de începere al
  antrenării rețelei neuronale
  await this.agent.fit(tf.stack(trainData.states), tf.tensor2d(
  trainData.newQValues))
  const tEnd = performance.now() // Timpul de sfârșit de episod

  // Reduc probabilitatea în funcție de rata sa
  if (epsilon > epsilon_min) {
    epsilon -= epsilon_decay
    epsilon = Math.max(epsilon, 0)
  }

  // Trimit datele analitice către interfața de utilizator
  cb({
    episode: eps, // Numărul episodului
    episodeTime: tEnd - t0, // Durata episodului
    dataPreparation: tTrain - tData, // Durata procesării de
    date
    fitDuration: tEnd - tTrain, // Durata de antrenament a
    rețelei

    episodeRewards: rewardsAnaly, // Recompensele acumulate
    numTensors: tf.memory().numTensors, // Numărul de tensori
    numBytes: tf.memory().numBytes // Spațiul de memorie ocupat
  })

  // La fiecare 50 de episoade curăț toată memoria
  if (eps % 50 === 0 && eps > 1) {
    console.log('CLEAN ALL MEMORY', eps)
    this.memory.clean()
  }
}

// Semnalez că antrenamentul s-a încheiat
return 'completed'
}
}
export default Trainer

```

Listing 4.8: Structura clasei Trainer

Având implementarea tuturor claselor necesare, o sesiune de antrenament poate fi inițiată astfel:

```

1 // Inițiere labirint
2 const board = new Board(10, 10);
3 // Inițieri reprezentare vizuală a labirintului

```

```

4  const boardUI = new BoardUI(board, new Konva.Stage({
5    container: "container",
6    width: 600,
7    height: 600,
8  }));
9  // Inițiere simulator
10 const env = new ImageEnv(board, boardUI);
11 // Inițiere agent
12 const agent = new ImageAgent();
13 // Inițiere memorie
14 const memory = new Memory(100, cleanMemoryExperience);
15 // Inițiere mediu de antrenare
16 const trainer = new ImageTrainer(env, agent, memory);
17 // Inițiere sesiune de antrenament
18 trainer.train()

```

Listing 4.9: Inițierea unei sesiuni complete de antrenament

Având programul complet, vom folosi datele analitice furnizate pentru crearea unei interfețe interactive pentru utilizator. Acestea pot vedea durata antrenamentului, performanța dobândită de-a lungul sesiunilor, etc.

4.6 Interfață Utilizator

Toată aplicația este concepută ca un site web, iar marele avantaj al acestui lucru este că putem dezvolta o interfață interactivă complexă destinată utilizatorului, în care acesta poate interacționa foarte ușor cu modele de învățare și vizualiza datele care au rezultat în urma simulării folosind grafice care sunt actualizate într-un timp extrem de scurt.

După inițierea aplicației, va apărea o imagine cu reprezentarea grafică a labirintului prin clasa `BoardUI` și un tabel care prezintă codificarea sa dat de clasa `Board`. Pentru inițierea agentului trebuie să apăsăm pe butonul *Inițializează un nou agent*.

Odată apăsând butonul, observăm că am avea disponibile următoarele butoane:

- **Atrenare** - buton care inițializează sesiunea de antrenament
- **Rulare** - buton care inițiază o sesiune de test pentru observarea progresului
- **Reset** - buton care aduce labirintul la starea inițială

Pe lângă aceste butoane, avem și opțiuni care ne permit să modificăm poziția agentului cu ajutorul mouse-ului sau să adăugăm un obstacol în labirint. În dreapta labirintului sunt prezente etichete care ne arată dacă agentul se află într-o sesiune de antrenament, informații legate de valoare acțiunilor din poziția curentă a agentului în labirint și un câmp de date care ne permite să introducem numărul de episoade al sesiunii de antrenament.

Dacă apăsăm pe butonul *Rulare*, observăm un comportament aleatoriu din partea agentului, deoarece rețeaua neuronală nu este antrenată, iar valorile ponderilor sinaptice au fost inițiate aleatoriu.

Odată apăsător pe botonul *Antrenare*, observăm că avem noi elemente vizuale. Acestea sunt cele care vor conține date despre performanța agentului și al sesiunii de antrenament.

Concluzii finale

În această lucrare am analizat cum elementele de învățare automată (rețele neuronale, Q-Learning) ajută la abordarea de probleme complexe, precum antrenarea de agenți autonomi care să îndeplinească obiectivele date în constrângerile din mediul de lucru. Folosind biblioteca Tensflow am arătat cum putem face un prototip pentru agent care poate să parcurgă un labirint pentru a ajunge la o poziție dorită.

Partea complexă pentru rezolvarea sarcinei este procesarea datelor de intrare care sunt sub formă de imagini. Dacă am fi ales să rezolvăm această problemă într-o manieră clasică, și anume prin analiza imaginilor folosind tehnici de prelucrare a imaginilor (filtre pentru detectarea de muchii, analiza formelor geometrice) și apoi transformarea lor într-un graf care să reprezinte modul de modelare al mediului simulat. Apoi a aplicat un algoritm care să ne ofere traseul până la obiectiv.

Utilizând rețele neuronale nu am avut nevoie de o analiză asupra imaginilor sau a modului cum funcționează simularea la inițiere, deoarece aceasta vor deduce de către algoritmul Q-Learning pe parcursul procesului de antrenare care folosește de recompensele acumulate de agent în urma acțiunilor sale pentru a stabili care sunt cele mai bune decizii pentru fiecare situație.

Folosind interfața dedicată utilizatorului am putut observa cum decurge o sesiune de antrenament, cum se îmbunătățesc decizii date de rețea, volumul de resurse consumate și timpul necesar al întregii operațiuni.

Cu toate aceste avantaje, învățarea automată nu este o soluție perfectă, consumul mare de resurse și timp, cât și natura ei de tip cutie neagră (faptul că nu putem observa ce se întâmplă cu procesele interioare) fac ca aceasta să fie o soluție excelentă mai mult pe partea de prototipuri pentru construcția sau testarea de produse derivate din idei complexe.

Bibliografie

- [1] Bellman, Richard, On the Theory of Dynamic Programming, National Academy of Sciences, 1952
- [2] Hebb, D. O. The organization of behavior : a neuropsychological theory / D.O. Hebb, Wiley New York, 1949
- [3] Greg Brockman and Vicki Cheung and Ludwig Pettersson and Jonas Schneider and John Schulman and Jie Tang and Wojciech Zaremba, OpenAI Gym, CoRR, 2016
- [4] Kurt Hornik, Approximation capabilities of multilayer feedforward networks, Neural Networks, 1991
- [5] Ferrari, Silvia & Stengel, Robert, Smooth Function Approximation Using Neural Networks. Neural Networks, 2005
- [6] Zhou, Zhenghua and Zhao, Jianwei, Approximation of Curves Contained on the Surface by Freed-Forward Neural Networks, Springer Berlin Heidelberg, 2011
- [7] <http://infolab.stanford.edu/pub/voy/museum/samuel.html>
- [8] <https://www.ibm.com/cloud/learn/unsupervised-learning>
- [9] <https://www.europarl.europa.eu/news/ro/headlines/society/20210211ST097614/big-data-de>
- [10] <https://svelte.dev>
- [11] <http://echarts.apache.org/en/index.html>
- [12] <https://www.tensorflow.org/js>
- [13] <https://konvajs.org/docs/index.html>
- [14] <https://www.mathworks.com/products/computer-vision.html>

Index

Învățare automată, 7, 13

capitol

 C2, 13

 C3, 19, 21

concluzii, 25

secțiune

 S1.1, 7