

Retro games with Deep Learning

Schill Christian

Cuprins

1	Introducere	3
1	Motivația alegerii lucrării	3
2	Deep Learning	4
3	Deep Reinforcement Learning	6
4	Jocurile Atari	8
5	Structura lucrării	9
2	Tehnologii Utilizate	10
1	Python	10
1.1	Avantajele limbajului Python pentru implementarea Ai	10
2	NumPy	12
3	TensorFlow	15
3.1	Avantajele lui TensorFlow	16
3.2	TensorFlow vs PyTorch	17
4	TensorBoard	19
5	Gym	21
3	Tehnici Deep Learning utilizate	26
1	Rețele neurale	26
2	Reinforcement Learning	36
3	Q-Learning	38
4	Double DQN	43
4	Detalii de implementare	47
1	Arhitectura rețelei	47
2	Exploration-exploitation trade-off	49
3	Replay Memory	50
4	Rețeaua întă și actualizarea parametrilor	51
5	Ghidul utilizatorului	54

5 Concluzii și perspective de dezvoltare	57
1 Îmbunătățiri posibile	57
2 Concluzii și recomandări pentru următoarea implementare	57

Capitolul 1

Introducere

1 Motivația alegerii lucrării

În momentul de față, rețelele neurale și deep learning reprezintă viitorul din punct de vedere al inteligenței artificiale. Cu ajutorul rețelelor neurale, probleme considerate imposibil de rezolvat, pot fi rezolvate. De exemplu, una dintre probleme nerezolvate ale matematicii, "P versus NP problem", poate fi rezolvată cu ajutorul unei rețele neurale, în prezent deja existând experimente care să se apropie din ce în ce mai mult de un rezultat. [38] Totodată, mai multe domenii s-au dovedit un mediu potrivit pentru ca o rețea neurală să fie antrenată, domenii precum diagnosticul medical, detectarea fraudelor cu utilizarea de carduri de credit, clasificarea secvențelor de ADN, robotica, jocurile video, procesarea de imagini, sisteme complexe 1.1, etc.

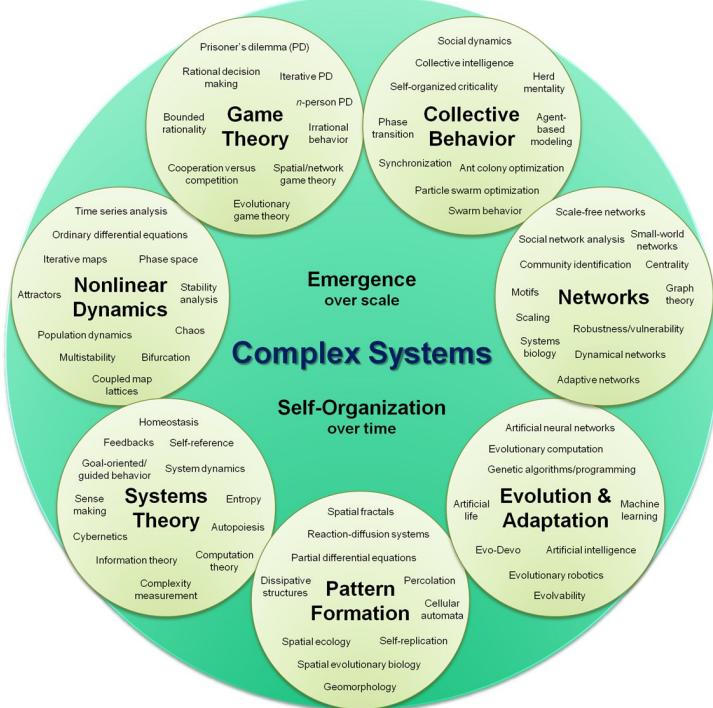


Figure 1.1: Harta sistemelor complexe. [1]

În prezent, există un număr mare de agenți inteligenți care au fost antrenați pe o varietate de jocuri video, oferind rezultate remarcabile. Am ales această lucrare deoarece îmbină două dintre pasiunile mele, deep learning-ul și jocurile video. Deep learning-ul și jocurile video sunt două domenii "future proof" datorită posibilităților nelimitate pe care le oferă.

2 Deep Learning

Deep learning este o ramură a domeniului machine learning, aceasta fiind axată pe algoritmi inspirați de către structura și funcționalitatea creierului, aceștia fiind denumiți rețele neurale artificiale. Astfel, aceste rețele neurale încearcă să simuleze creierul uman, fiind însă departe de capabilitățile acestuia, reușind să "învețe" dintr-o cantitate mare de date.

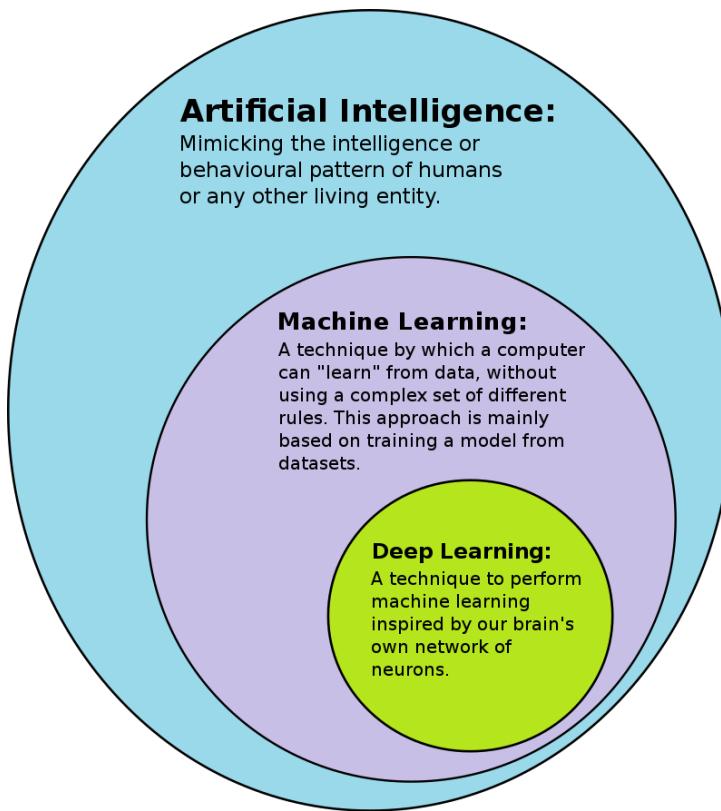


Figure 1.2: Diferenta dintre AI, ML și DL. [31]

Diferența dintre machine learning și deep learning constă în tipurile de date care se folosesc și metoda de învățare 1.2. Algoritmii machine learning folosesc date structurate, etichetate pentru a face predicții, însemnând că anumite caracteristici sunt definite de input-ul pentru model și sunt organizate în tabele. Asta nu înseamnă că machine learning-ul nu folosește date care nu sunt structurate, doar că, dacă le-ar folosi, în general acestea trec printr-o pre-procesare pentru a le organiza în date structurate.

Deep Learning elimină o parte din pre-procesarea datelor care este de obicei asociată cu machine learning. Acești algoritmi pot utiliza și procesa date nestructurate, ca de exemplu text sau imagini, și automatizează extragerea de caracteristici, eliminând parțial dependența de expertiza omului. De exemplu, dacă am avea un set de imagini cu diferite animale clasificate în "pisici", "câini" și "hamsteri", etc, algoritmii de tip deep learning pot identifica care trăsături (de ex. urechile) sunt mai importante pentru a deosebi fiecare animal. În machine learning, această ierarhie de trăsături este stabilită manual de către un om.

Cu ajutorul metodelor de gradient descent și backpropagation, algoritmii deep

learning se pot auto-ajusta, reușind astfel să facă predicții cu o precizie crescută. Modelele de machine learning și deep learning sunt capabile de diferte tipuri de învățare, acestea fiind de obicei clasate în supervised learning, unsupervised learning și reinforcement learning. Supervised learning utilizează seturi de date etichetate pentru a clasifica datele de intrare corect. În comparație, unsupervised learning nu are nevoie de seturi de date etichetate, astfel acesta detectează anumite trăsături recurente în date, grupându-le prin caracteristicile distinctive. Reinforcement learning este un domeniu al machine learning-ului, în care scopul principal este de a efectua acțiuni în favoarea maximizării recompensei într-o anumită situație.

3 Deep Reinforcement Learning

Deep reinforcement learning este un proces prin care un model devine mai precis în urma realizarea unei acțiuni într-un mediu bazat pe feedback și recompense, încercând să maximizeze recompensa. Aceasta combină rețelele neurale cu o arhitectură de tip reinforcement learning, oferind posibilitatea agentului inteligent de a învăța care este cea mai bună acțiune într-un mediu virtual pentru a-și îndeplini scopul. Reinforcement learning se referă, în general, la algoritmi orientați pe scopuri, care învăță cum să ajungă la un obiectiv complex (scopul) sau cum să maximizeze pe parcurs o anumită măsură după mai mulți pași, de exemplu, maximizarea punctelor câștigate într-un joc după mai multe mutări. Algoritmii reinforcement learning pot porni de la zero, și în condițiile potrivite, pot atinge performanțe peste capacitatele umane. Aceștia sunt antrenați prin metoda recompensei, fiind penalizați dacă gresesc și recompensați dacă fac o mutare corectă.

Algoritmii de tip reinforcement learning împreună cu rețelele neurale pot învinge experți în numeroase jocuri, precum, jocurile Atari, Starcraft II și Dota-2. Deși asta ar putea fi considerat ceva trivial pentru cei care nu se joacă jocuri video, aceste reușite sunt o îmbunătățire majoră față de reușitele precedente, iar zi de zi se obțin rezultate din ce în ce mai bune.

AlphaStar este primul agent intelligent care să învingă un jucător profesional de top în Starcraft II. Într-o serie de meciuri jucate, AlphaStar I-a învins pe Grzegors "MaNa" Komincz care face parte din Team Liquid, unul dintre cei mai buni jucători de Starcraft, în GIF-ul de mai jos [3](#), cu un scor de 5-0 . Meciurile au fost ținute în aceleași condiții ca și meciurile profesioniste, pe o harta competitivă și fără nicio restricție în joc. Rezultatul lui AlphaStar în Starcraft II este unul dintre cele mai remarcabile rezultate oferite de deep reinforcement learning datorită complexității jocului [1.3](#).



Figure 1.3: AlphaStar vs "MaNa".[23]

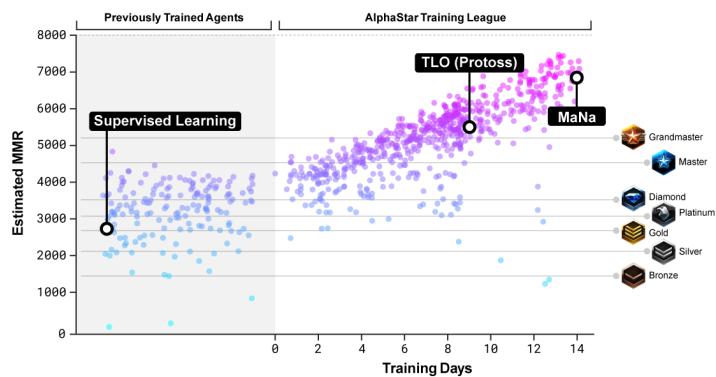


Figure 1.4: MMR-ul lui AlphaStar. MMR-ul este o valoare asignată fiecărui jucător care determină nivelul de joc al acestuia.[23]

Pentru a reuși acest lucru Alphastar a trebuit să inoveze în mai multe zone de cercetare în domeniul Ai datorită situațiilor neașteptate și a necesității de a balansa scopurile de scurtă durată cu cele de lungă durată. Astfel, pentru Alphastar au apărut următoarele provocări:

- Game theory: StarCraft este un joc în care, asemănător cu piatră-hârtie-foarfece, nu există o strategie unică pentru a câștiga. Astfel, în timpul antrenării, un Ai trebuie să proceseze, să exploreze și să dobândească cunoștințe și strategii diverse.
- Imperfect information: Spre deosebire de alte jocuri, ca de exemplu Go sau șah unde jucătorii au accesul la toate informațiile necesare, în StarCraft

informațiile cruciale sunt ascunse de jucători, acesta trebuie să meargă în recunoaștere constant pentru a le obține.

- Long term planning: Ca și multe alte probleme din viața reală, relația de cauză-efect nu este instantanee. Meciurile pot dura pună la o oră pentru a se termina, iar asta înseamnă că decizile luate la început pot să influențeze abia după mai mult timp.
- Real time: În comparație cu alte board games, unde jucătorii alternează întregi tururi pentru mutările ulterioare, jucătorii de StarCraft trebuie să acționeze permanent și continuu în timp ce jocul progresează
- Large action space: Sute de unități diferite și clădiri trebuie controlate în același timp, în timp real, astfel rezultă un spațiu combinatoric de posibilități. Pe lângă asta, acțiunile se află într-o ierarhie și pot fi modificate și mărite. Parametrizarea lui AlphaStar este aproximativ la 10-26 de acțiuni legale la fiecare pas.

Reinforcement learning rezolvă problema dificilă a corelării imediate a acțiunilor cu reacțiunile întârziate pe care le produc. Asemănător cu oamenii, algoritmii reinforcement learning câteodată trebuie să aștepte pentru a vedea rezultatul deciziilor lor. Aceștia funcționează într-un mediu cu răspuns întârziat, fiind dificil să înțeleagă care acțiune duce către fiecare rezultat. Este rezonabil să presupunem că în viitor, algoritmii de tip reinforcement learning vor obține rezultate mai bune în medii mai ambiguë, în medii asemănătoare cu viața reală, în timp ce aleg arbitrar un număr de acțiuni, în loc de acțiunile limitate și repetate ale unui joc video. Multe companii folosesc în momentul de față deep reinforcement learning pentru a rezolva probleme în diferite industrii, cum ar fi: jocurile video, controlarea avioanelor și controlarea mișcării unui robot, mașinile care se pot conduce singure, în diferite aplicații medicale, etc.

4 Jocurile Atari

Atari Games Corporation (cunoscută ca Midway Games West Inc. după 1999) a fost un producător american de jocuri arcade. Original, divizia se ocupa de jocurile arcade care funcționau pe bază de fise, aparținând de Atari Inc și a fost despărțită de aceasta în 1984, devenind propria companie. A fost una dintre numeroasele companii care au folosit numele Atari. Când divizia "Warner Communications" a Atari Inc. a pierdut 500 de milioane de \$ în primele 3 sferturi din 1983, doar divizia care se ocupa cu jocurile cu fise a rămas profitabilă. În 1984, Warner a vândut divizia de produse dedicate consumatorilor către Jack Tramiel, acesta numindu-și

compania în Atari Corporation. Warner a păstrat divizia de jocuri arcade și încă câteva bunuri și redenumește Atari, Inc în Atari Games, Inc. Aranjamentul dintre Tramiel și Warner Communications a fost ca Atari Games să includă întotdeauna logo-ul de "Games" după "Atari" și că Atari Games nu pot folosi niciodată brand-ul Atari în secțiunea de produse dedicate consumatorilor (calculatoare și console).

Cele mai bine vândute jocuri Atari sunt :

- Pac-Man lansat în 1982
- Space Invaders lansat în 1980
- Pitfall lansat în 1982
- Donkey Kong lansat în 1982
- Frogger lansat în 1982

5 Structura lucrării

În următoarele capitole se vor detalia tehnologiile utilizate, tehnici Deep Learning utilizate, arhitectura aplicației și în final concluzii și perspective de dezvoltare.

În capitolul de tehnologii utilizate se vor prezenta detalii despre limbajul de programare utilizat și bibliotecile folosite.

În capitolul de tehnici deep learning utilizare se vor descrie tehniciile folosite pentru realizarea rețelei neurale, precum: reinforcement learning, Q-learning, Dueling Networks și Double Q Learning.

Pentru capitolul de arhitectura aplicației, se vor prezenta rețea neurală implementată și componentele "Exploration-exploitation trade-off" și "Replay Memory" în amănunt.

În ultimul capitol se vor explica îmbunătățirile posibile și recomandări pentru următoarea implementare.

Capitolul 2

Tehnologii Utilizate

În acest capitol voi vorbi despre limbajul de programare Python, avantajele acestuia pentru implementare de Ai, despre biblioteca NumPy și framework-ul TensorFlow, avantaje lui TensorFlow și o comparație cu un alt framework, PyTorch și TensorBoard, o interfață de vizualizare. În final, voi prezenta toolkit-ul Gym folosit pentru cercetarea reinforcement learning-ului.

1 Python

În 1989, programatorul olandez Guido van Rossum crează Python, un limbaj de programare dinamic, multi-paradigmă. În ziua de azi, Python, deoarece este un limbaj multifuncțional, este folosit de către companii precum Google sau Yahoo!, pentru programarea aplicațiilor web, dar și a aplicațiilor de divertisment sau aplicații științifice care sunt programate parțial sau în întregime în Python. Deseori este descris ca limbajul de programare cu "bateriile incluse" datorită bibliotecilor standard numeroase. Python se numără deseori printre cele mai populare limbaje de programare, fiind de cele mai multe ori în top.[\[25\]](#)

1.1 Avantajele limbajului Python pentru implementarea Ai

Proiectele Ai diferă de proiectele software standard. Diferența constă în tehnologiile folosite, cunoștințele necesare pentru a realiza un proiect bazat pe Ai și nevoia cercetării profunde. Pentru a implementa un Ai este nevoie de un limbaj de programare stabil, flexibil și care posedă uneltele necesare pentru realizarea sa. Python oferă toate acestea, iar de aceea este folosit de cele mai multe ori în proiectele Ai de astăzi.

De la implementare până la menenanță, Python ajută developeri să fie productivi datorită simplității și consecvenței, accesul către un număr mare de framework-

uri și biblioteci dedicate pentru AI și machine learning, flexibilității, a platformei independente și a comunității. Python oferă cod concis și usor de citit. În timp ce algoritmii de machine learning și AI pot fi complexi și pot avea un workflow versatil, simplicitatea lui Python ajută developeri să scrie cod fără a se lovi de problema citirii codului, iar aceștia se pot concentra cât mai mult pe rezolvarea problemei.

În plus, Python este ușor de învățat și accesibil, acesta fiind considerat mult mai intuitiv în comparație cu alte limbaje de programare. Totodată, gama largă de framework-uri, biblioteci și extensii ușurează implementarea a diferitelor funcționalități. În general este acceptat faptul că Python este un limbaj de programare potrivit pentru munca în echipă, când există dezvoltatori mulți. Deoarece este un limbaj cu un scop general, se poate realiza un set de sarcini machine learning complexe și se poate construi un prototip într-un timp scurt pentru a putea fi testat.

Implementarea unui algoritm AI și machine learning poate fi dificilă și implementarea necesită mult timp. Este vital să existe un mediu de programare bine structurat și bine testat pentru a permite dezvoltatorilor să vină cu cele mai bune soluții. Pentru a reduce timpul de dezvoltare, programatori folosesc câteva framework-uri și biblioteci. Python, datorită posibilităților tehnologice multiple, are un set larg de biblioteci și framework-ură dedicate inteligenței artificiale și machine learning-ului. Câteva dintre ele sunt:

- Keras, TensorFlow, și Scikit-learn pentru machine learning
- NumPy pentru performanță bună în calcului științific și analiza datelor
- SciPy pentru calcul avansat
- Pandas pentru analiza datelor în scop general
- Seaborn pentru vizualizarea datelor

Scikit-learn oferă algoritmi variați de clasificare, de regresie și grupare, incluzând și mașini vectoriale de sprijin, arbori de decizie aleatorii, creștere a gradientului, k-mean și DBSCAN, fiind gândit să funcționeze împreună cu bibliotecile de calcul în Python cum ar fi NumPy și SciPy.

Mai jos este o listă [1.1](#) cu cele mai populare subdomenii AI și framework-urile cele mai potrivite pentru implementarea de algoritmi a acestor subdomenii:

- Machine Learning - TensorFlow, Keras, Scikit-learn
- Computer vision - OpenCV

- Natural language processing - NLTK, spaCy

Independența platformei se referă posibilitatea unui limbaj de programare sau framework de a putea permite unui dezvoltator să implementeze ceva pe o platformă și apoi să îi ofere posibilitatea de a continua pe o altă platformă fără modificări (sau cu modificări minime). O cheie a succesului lui Python este platforma independentă. Python suportă o multitudine de platforme precum Linux, Windows și macOS. Codul Python poate fi folosit pentru a crea executabile "standalone" pentru cele mai comune sisteme de operare, ceea ce software-ul dezvoltat în Python este distribuit cu ușurință pe aceste sisteme de operare fără nevoie de un Python interpreter. În plus, dezvoltatorii folosesc de obicei servicii cum ar fi Google sau Amazon pentru sarcinile lor. Însă, se pot găsi multe companii și mulți data scientists care folosesc propriile calculatoare cu GPU-uri puternic pentru a antrena modele machine learning și deoarece Python este o platformă independentă, acest lucru face antrenarea mult mai usoară și ieftină.

În 2020 Stack a realizat un studiu al dezvoltatorilor, Python fiind în top 5 cele mai populare limbi de programare.[\[19\]](#) În alt studiu de caz, se observă că Python este în principal pentru data analysis. La prima vedere, se observă faptul că web development-ul reprezentă 27% din cazurile în care este folosit Python. Dar, dacă se combină procentajul de folosire în data analysis și machine learning, Python este folosit în 30% din aceste cazuri.[\[12\]](#) În plus Python este atât sigur și ușor de folosit încât Google îl folosește pentru bot-ul de tip web crawler(spider), Pixar pentru a produce filme animate, Spotify pentru recomandările de muzică. Este un fapt cunoscut faptul că Python a strâns o comunitate de entuziaști de AI pe întreg globul. Astfel, diferitele forum-uri și activitatea crescută a comunității oferă soluții pentru diferitele probleme legate de machine learning sau deep learning. Pentru orice sarcină, este o șansă foarte mare ca altcineva să fi avut aceeași problemă, aşadar se pot găsi dezvoltatori care pot ghida sau da sfaturi foarte ușor. Deseori, celei mai bune soluții se pot obține cu ajutorul comunității Python.

Filtrele de spam, sistemele de recomandare, motoarele de căutare, asistenții personali și sistemele de detectare a fraudelor sunt toate posibile cu ajutorul algoritmilor AI și machine learning și pe zi ce trece apar noi domenii în care inteligența artificială devine tot mai relevantă, iar în momentul de față, Python este punctul central ce unește aceste tehnologii.

2 NumPy

NumPy este una dintre numerele bibliotecii Python. NumPy adaugă suport pentru matrice și vectori pe mai multe dimensiuni. În plus, vine cu o colecție mare de funcții matematice de nivel înalt pentru vectorii și aceste matricele pe mai

multe dimensiuni. Numeric, strămoșul lui NumPy, a fost original creat de către Jim Hugunin, împreună mai multor developeri. În 2005, Travis Oliphant creează NumPy, încorporând mai multe funcționalități din Numarray în Numeric, cu modificări semnificative. NumPy este open-source și posedă numeroși contribuitori.

NumPy este pachetul fundamental pentru calculul științific în Python. Există câteva diferențe semnificative față de vectorii NumPy și cei standard din Python, cum ar fi:

- Vectorii NumPy au o dimensiune fixă în momentul creării în comparație cu listele Python (dimensiunea poate crește în mod dinamic)
- Elementele unui vector NumPy trebuie să fie de același tip, astfel ocupă aceeași dimensiune în memorie, cu excepția că pot exista vectori de obiecte, astfel este posibil să existe elemente de diferite dimensiuni.
- Vectorii NumPy facilitează operații matematice avansate și alte tipuri de operații pe un număr mare de date. De obicei, astfel de operații sunt execute în mod eficient, și cu mai puțin cod scris decât secvențele standard din Python.
- Un număr mare de pachete științifice și matematice bazate pe Python folosesc vectori NumPy. Deși acestea acceptă și secvențele standard Python ca input, ele sunt de obicei convertite în vectori NumPy înainte de procesarea lor, și deseori ca output se obțin vectori NumPy. În alte cuvinte, pentru a efectua în mod eficient majoritatea calculelor științifice în Python, cunoașterea secvențelor standard din Python nu este de ajuns, este nevoie să se cunoască și vectorii NumPy.

Punctele 1 și 3 (cele despre dimensiune și eficiență) sunt cele mai importante când vine vorba de calculul științific. Ca un simplu exemplu, se consideră cazul în care se înmulțesc elementele dintr-o secvență unidimensională cu elementele corespunzătoare ale altrei secvențe de aceeași dimensiune. Dacă secvențele sunt stocate în două liste Python, a și respectiv b, am putea itera peste fiecare element.

[2]

Listing 2.1: Înmulțirea a două liste în Python

```
c = []
for i in range(len(a)):
    c.append(a[i]*b[i])
```

Această secvență de cod 2.1 obține un rezultat corect, dar dacă a și b fiecare conțin milioane de numere, eficiența ar scădea semnificativ datorită ineficienței de a trece printr-o structură repetitivă în Python. Se poate obține același rezultat

însă, și mult mai rapid, cu cod C. Pentru claritate se negligează declararea variabilelor și inițializările, alorcările de memorie, etc.

Listing 2.2: Înmulțirea a doi vectori în C

```
for ( i = 0; i < rows; i ++): {  
    c[ i ] = a[ i ]*b[ i ];  
}
```

Această secvență de cod 2.2 elimină costul necesar pentru interpretarea codului Python și manipularea obiectelor în Python, dar și beneficiile codului Python. În plus, codul va crește odată cu creșterea numărului de dimensiuni. În cazul unui vector bidimensional 2.3, de exemplu, codul în C (cu neglijările precedente) s-ar extinde astfel:

Listing 2.3: Înmulțirea a două matrici în C

```
for ( i = 0; i < rows; i ++): {  
    for ( j = 0; j < columns; j ++): {  
        c[ i ][ j ] = a[ i ][ j ]*b[ i ][ j ];  
    }  
}
```

NumPy oferă ce este mai bun din ambele limbaje de programare: operații realizate element cu element sunt în mod implicit într-un vector de tip ndarray, dar aceste operații sunt executate într-un timp scurt cu ajutorul pre-compilat de C. În NumPy

Listing 2.4: Înmulțirea a două matrice cu NumPy

```
c = a * b
```

realizează același lucru ca exemplele precedente, dar la viteze apropiate de limbajul C și cu simplicitatea obișnuită a codului Python 2.4. Acest exemplu evidențiază două dintre cele mai importante funcționalități pe care le oferă NumPy: vectorizarea și broadcasting.

Vectorizarea reprezintă absența secvențelor repetitive, indexării etc., în cod - acestea se întâmplă în spate cu ajutorul codului precompilat și optimizat din C. Vectorizarea oferă multe avantaje, o parte din ele fiind:

- vectorizarea codului este mult mai ușor de citit și mai concisă
- mai puține linii de cod în general înseamnă mai puține bug-uri
- codul se asemănă foarte mult cu notațiile matematice standard, astfel se pot codifica construcții matematice în mod corect, și în general mult mai ușor.

- vectorizarea codului duce la un cod mai simplist, ușor de urmărit, specific Python. Fără vectorizare, codul ar fi plin de structuri repetitive "for" ineficiente și greu de citit.

Broadcasting este o metoda de verificare specifică NumPy, care verifică dacă două matrice sau două șiruri sunt compatibile dpdv al operațiilor, adică se verifică comportamentul implicit al acestor operații care se realizează element cu element. În general, operațiile NumPy nu sunt doar aritmetice, dar și logice, pe biți, funcționale, etc. În plus, în exemplul de mai sus, a și b puteau fi șiruri multidimensionale cu diferite forme, considerând faptul că șirurile mai mici puteau fi "extinse" în forme mai mari într-un mod care nu ar fi ambiguu.

Vectorizarea și broadcasting-ul sunt motivele principale pentru care se folosesc NumPy. Totodată consumă mai puțină memorie decât listele Python, este mai rapid, se implementează mai ușor vectori multidimensionali. În plus, s-au folosit funcțiile np.empty, np.transpose, np.repeat, np.append, np.mean și np.random.

3 TensorFlow

TensorFlow este o bibliotecă open-source pentru machine learning. Poate fi folosită pentru numeroase sarcini, dar este centrată în mod special pe antrenarea și inferența rețelelor neurale deep learning. TensorFlow este o bibliotecă a algebrei calculatorului, bazată pe programarea fluxului de date și programarea diferențială. Este folosită atât pentru cercetare cât și producție de către Google. TensorFlow a fost dezvoltată de către echipa Google Brain, pentru folosire internă în cadrul Google. În 2015 a fost lansată sub Apache License 2.0. TensorFlow este scris cu ajutorul limbajelor Python, C++ și CUDA.

Programarea fluxului de date este o paradigmă a programării care modelează un program ca un graf orientat al fluxului de date dintre operații, astfel implementează principiile și arhitectura fluxului de date. Programarea fluxului de date a fost introdusă de către Jack Dennis și studenții lui absolvenți ai MIT în anii 1960.[\[30\]](#)

Programarea diferențială este o paradigmă a programării în care un program al calcului numeric poate fi diferențiat în întregime cu ajutorul diferențierii automate. Dă voie optimizării bazate pe gradare a parametrilor dintr-un program, deseori prin gradient descent. Programarea diferențială este folosită în diverse scopuri, în particular în știința calculatorului și a inteligenței artificiale.[\[32\]](#)

Cele mai multe framework-uri pentru programarea diferențială construiesc un graf care conține fluxul de control al structurilor de date dintr-un program. Primele abordări se împart în două grupuri:

- Abordarea bazată pe grafuri statice, compilate cum ar fi TensorFlow, Theano și MXNet. De obicei acestea urmăresc o optimizare bună a compilatorului și scalarea ușoară pe sisteme mai mari, dar natura lor statică limitează interactivitatea și tipurile de programe care pot fi create cu ușurință (de ex. cele care conțin structuri repetitive sau recursive).
- Abordarea bazată pe grafuri dinamice, supraîncărcarea operatorilor cum ar fi PyTorch și AutoGrad. Natura lor dinamică și interactivă oferă posibilitatea de a scrie majoritatea programelor cu ușurință. Însă, acestea duc la overhead-ul translatorului (în mod special când se realizează numeroase operații mici), scalabilitate micșorată și nu beneficiază de optimizarea compilatorului pe deplin.

3.1 Avantajele lui TensorFlow

TensorFlow oferă nivele multiple de abstracție, astfel se poate alege nivelul potrivit în funcție de necesitatea. Construirea și antrenarea modelelor se realizează folosind API-ul de nivel înalt Keras, ceea ce face învățarea noțiunilor de bază a bibliotecii TensorFlow și a machine learning-ului mai ușoară.^[6]

Dacă este nevoie de flexibilitate mai mare, eager execution un debugging intuitiv și iterare imediată. Pentru sarcinile mari de antrenare machine learning, se poate folosi Distribution Strategy API pentru a distribui antrenarea pe mai multe configurații hardware fără a fi nevoie să se schimbe definiția modelului.

TensorFlow oferă suport pentru diferite platforme și limbi de programare, astfel antrenarea se poate realiza în orice moment, fie pe servere, web, dispozitive precum routere, numeroase dispozitive de acces MAN (metropolitan area network) sau WAN (wide area network). Pentru un pipeline complet pentru producerea de modele machine learning se poate folosi TensorFlow Extended (TFX), pentru inferențe pe telefoane sau alte dispozitive TensorFlow Lite. Antrenarea și implementarea modelelor în medii JavaScript se poate realiza cu ajutorul TensorFlow.js.

Construirea și antrenarea modelelor de ultimă generație se realizează fără a sacrifica viteză sau performanță. TensorFlow propune flexibilitate și control cu ajutorul funcționalităților precum Keras Functional API și Model Subclass API pentru crearea topologilor complexe. Pentru construirea de prototipuri ușoare și pentru debuggin-ul rapid, se folosește eager execution.

În prezent, TensorFlow este folosit de o gamă largă de companii, precum Coca-Cola, Google, Intel, Lenovo, PayPal, Qualcomm, Spotify, etc.^[21]

3.2 TensorFlow vs PyTorch

Atât PyTorch cât și TensorFlow sunt framework-uri foarte populare [2.1](#), atât de populare încât managerii de proiect și data scientisti le consideră ca fiind primele biblioteci către pe care le aleg când vine vorba de dezvoltarea unor aplicații deep learning inovative sau cercetare.

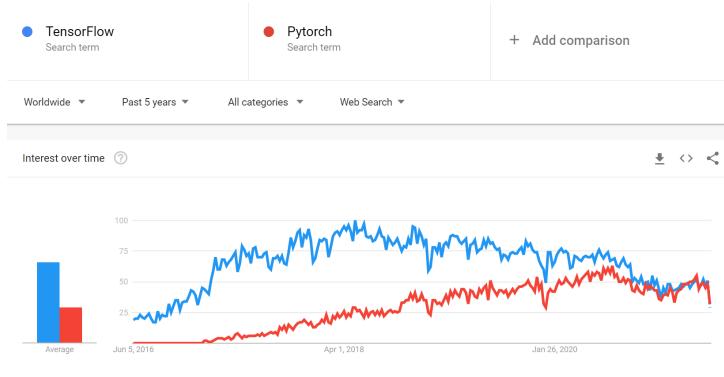


Figure 2.1: TensorFlow versus PyTorch din perspectiva popularității ¹

Și mai mult, PyTorch și TensorFlow încă posedă câteva moduri de operare diferite, chiar dacă în trecut aveau diferențe mult mai mari. Peste timp, au devenit din ce în ce mai competitive încât cele mai bune funcționalități au fost incluse în ambele framework-uri.

Mecanismul de definire al grafului

Grafurile sunt folosite pentru a descrie calculul. Un graf este o structură de date care constă în noduri și vârfuri. În timpul procesului de antrenare al rețelelor neurale deep learning, grafurile stochează funcțiile de activare a rețelelor neurale în timpul pasului de propagare înainte. Un pass de propagare înapoi folosește apoi graful pentru a calcula schimbările pentru fiecare dintre weight-urile rețelei, astfel antrenează rețeaua. Pentru a înțelege diferența dintre cele două, ar trebui să se înțeleagă ce au PyTorch și TensorFlow în comun în legătură cu definirea grafurilor. În primul rând, ambele framework-uri grafuri orientate aciclice și acționează pe tensori. Tensorii descriu relația dintre un set de obiecte într-un spațiu vectorial. Totuși, cele două au definiții ale grafurilor distincte.

TensorFlow dă voie utilizatorului să efectueze operații pe tensori după crearea grafului de flux de date cu stări. Graful este definit static înainte ca modelul să poată rula. În comparație, PyTorch este mai dinamic, dă voie utilizatorului să execute operații pe noduri în timp ce modelul rulează. În alte cuvinte, graful este

¹Sursă:<https://trends.google.com/trends/explore?date=today%205-y&q=TensorFlow,PyTorch>

creat în fiecare punct de execuție, și este posibil să se realizeze modificări pe graf în timpul runtime-ului. Din acest motiv, PyTorch este de preferat în cercetare, deoarece este mai potrivit pentru construirea de modele personalizate, și totodată pentru că e dinamic, îl poate face mai ușor de interacționat cu părțile interne ale modelelor.

Însă, în 2019, TensorFlow 2.0 introduce și el grafuri dinamice, astfel dă voie operațiilor să fie evaluate la runtime fără a genera un graf mai târziu. Prin urmare, ce era în trecut una dintre diferențele majore dintre cele două framework-uri, nu mai este la fel de semnificativă, considerând faptul că un utilizator poate utiliza atât grafuri dinamice cât și cele statice pe ambele framework-uri.

Implementarea modelului

În 2020, PyTorch introduce TorchServer, care este o unealtă de implementare a modelelor. Această unealtă prevede un set basic de funcționalități, cum ar fi valori, o specificare a punctului final, o unealtă de arhivare a modelului, etc.

În schimb, TensorFlow posedă TensorFlow Serving, care este un o unealtă de implementare a modelelor încorporată, care este folosită atât pentru modelele machine learning cât și pentru servere gRPC[5]. În plus, permite accesul de la distanță la serverele gRPC. Pe total, TensorFlow Serving permite utilizatorului să implementeze algoritmi noi în timp ce prezervă aceeași arhitectură a serverelor și a API-urilor. Această unealtă a fost testată pe multe proiecte Google și proiectată pentru mediile de producție.

Vizualizarea

PyTorch și TensorFlow ambele suportă unele de vizualizare, care facilitează debugging-ul și permite utilizatorului să vizualizeze rezultatele mai rapid și să aibă o viziune amplă asupra progresului antrenării unui model. Pe de o parte, PyTorch nu oferă o unealtă dedicată pentru vizualizare, dar deține Visdom, o unealtă minimalistă de vizualizare. Visdom poate fi folosit cu NumPy și PyTorch. Oferă funcționalități minimale și limitate, dar este și destul de ușor de folosit, flexibil și suportă tensori PyTorch. De altă parte, TensorFlow folosește TensorBoard, care oferă o suiată largă de aplicații care ajută utilizatorul să înțeleagă modelul prin cinci vizualizări diferite: (1) grafuri; (2) audio; (3) imagini; (4) distribuirile și histograme; (5) scalari.

Tensorboard este considerat o unealtă de vizualizare mult mai versatilă decât Visdom, de aceea în versiunea de PyTorch 1.2.0 este introdus suportul pentru integrarea Tensorboard.

Debugging

Debugging-ul în Python se realizează cu ajutorul uneletelor standard din Python (de ex. PyCharm debugger și pdb). De fapt, deoarece Pytorch definește grafurile dinamic la runtime, majoritatea uneltelor Python pot fi ușor integrate. În TensorFlow este mai complicat să se facă debug pentru codul modelului. În acest caz, utilizatori trebuie să învețe debugger-ul bibliotecii, tfdbg - și variabilele cerute

dintr-o sesiune.

Framework	PyTorch	TensorFlow
Origine	PyTorch este bibliotecă care a fost dezvoltată de către Facebook AI Research Lab.	TensorFlow este o bibliotecă open-source creată de către Google Brain.
Mecanismul de definire al grafului	Grafuri dinamice - utilizatorii pot executa operații pe noduri în timp ce modelul rulează.	Graf de flux de date cu stări, deși în 2019 TensorFlow 2.0 introduce de asemenea grafuri dinamice.
Implementarea modelului	În 2020 PyTorch introduce TorchServe.	TensorFlow folosește TensorFlow Serving, care are încorporat o unealtă de implementare a unui model machine learning , dar și servere gRPC.
Vizualizarea	Visdom - Versiunea de PyTorch 1.2.0 face posibilă folosirea Tensorboard.	Tensorboard - oferă o suită de aplicații care permit utilizatorului să înțeleagă modelul deep learning prin cinci vizualizări diferite.
Debugging	Debugger standard Python - de ex. PyCharm debugger și pdb	tfdbg - utilizatorul trebuie să învețe debugger-ul bibliotecii.

Table 2.1: PyTorch versus TensorFlow[14]

În concluzie, atât PyTorch cât și TensorFlow se direcționează către excelență când vine vorba de rețele neurale. Ambele framework-ui au fost în bunătăție constant, luând una de la cealaltă cele mai bune funcționalități. Asta face ca alegerea dintre cele două să fie foarte grea [2.1](#).

Se folosește TensorFlow în acest proiect datorită API-ului de nivel înalt Keras, care este un API ușor de învățat, consecvent și oferă mesaje de eroare ușor de înțeles. Totușă, oferă o documentație completă și o multitudine de ghiduri pentru învățare, și oferă o unealtă de vizualizare extensivă, TensorBoard.

4 TensorBoard

TensorBoard este o interfață folosită pentru vizualizarea grafului, pentru înțelegerea, debugging-ul și optimizarea unui model. Ajută totodată la urmărirea măsurilor

cum ar fi:

- Urmărirea și vizualizarea măsurilor, de exemplu loss și acuratețe
- Vizualizarea graf a modelului (ops și straturi)
- Vizualizarea histogramelor a weight-urilor, bias și alți tensori în timp ce se modifică
- Redarea embedding-urilor la un spațiu dimensional mai mic
- Afisarea de date ca imagini, text sau audio
- Descrierea aplicațiilor TensorFlow

[Meniu vizualizări Tensorboard. [\[22\]](#)]

Cum se poate observa în GIF-ul [4](#), meniul TensorBoard conține cinci categorii:

- Scalars - afișează informații utile în timpul antrenării unui model
- Graphs - afișează modelul
- Histograms - afișează weight-urile cu o histogramă
- Distribution - afișează distribuția weight-urilor
- Projector - afișează PCA (Principal component analysis)[\[39\]](#) și algoritmul T-SNE[\[42\]](#). Acestea sunt tehnici folosite pentru micșorarea dimensiunilor.

5 Gym

OpenAI Gym este un toolkit pentru domeniul de cercetare a reinforcement learning-ului. Include o colecție în creștere de probleme reper care expun o interfață comună. În prezent, combinarea dintre reinforcement learning și deep learning au

dus către entuziasm crescut în domeniu, în timp ce a devenit tot mai evident că algoritmii precum policy gradients[17] și Q-learning pot obține performanțe bune pentru probleme dificile. Pentru a continua progresul în reinforcement learning, comunitatea are nevoie de un reper pentru a compara algoritmii. O varietate de repere au fost lansate, ca de exemplu Arcade Learning Environment (ALE), care expun o colecție de jocuri Atari 2600 ca probleme de reinforcement learning și recent RLLab benchmark pentru controlul continuu și alte repere precum RLPy[17], RL-Glue[17], PyBrain[17] și RLLib[17]. OpenAI Gym își propune să combine cele mai bune elemente din reperele menționate anterior. Include o colecție diversă de medii cu o interfață comună care va crește odată cu timpul.

Reinforcement learning consideră că agentul este situat într-un mediu. După fiecare pas, agentul efectuează o acțiune și obține un feedback și o recompensă din partea mediului. OpenAI Gym se concentrează pe setare episodică, unde "experiența" agentului este împărțită în episoade. În fiecare episod, starea inițială a agentului este încercată aleatoriu dintr-o distribuție, și interacțiunea continuă până când mediu ajunge într-o stare terminală. În reinforcement learning-ul episodic scopul este de a maximiza aşteptarea de recompensă totală per episod, și de a atinge nivel înalt de performanță în cât mai puține episoade posibile.

Următoarea secvență de cod urmărește un singur episod cu 100 de pași. Se consideră că există un obiect numit agent, care primește feedback după fiecare pas și un obiect numit env care este mediu. OpenAI Gym nu include o clasă agent sau nu specifică agentului ce interfață să folosească. Se include un agent aici doar pentru demonstrație 5.

```
ob0 = env.reset()#sample environment state ,
return first observation
a0 = agent.act(ob0)#agent chooses first action
ob1, rew0, done0, info0 = env.step(a0)
#environment returns observation ,
#reward, and boolean flag
#indicating if the episode is complete .
a1 = agent.act(ob1)
ob2, rew1, done1, info1 = env.step(a1)
...
a99 = agent.act(o99)
ob100, rew99, done99, info2 = env.step(a99)
#done99 == True => terminal
```

OpenAI Gym este bazat pe experiența utilizatorului în dezvoltarea și comparația algoritmilor reinforcement learning. Modul de operare arată în felul următor:

- Medii, nu agenți. Cele două concepte principale ale reinforcement learning

sunt agentul și mediul. GymAI oferă o formă abstractă doar a mediului nu și a agentului. Aceasă alegere a fost făcută pentru a maximiza conveniența utilizatorilor și pentru a putea implementa diferite tipuri de agenți. De exemplu, se consideră un stil de "învățare online", unde agentul preia(feedback, recompensă, gata) ca și inout la fiecare pas și efectuează modificări incremental. Într-un alt stil, "batch update", un agent este chemat cu feedback ca input, și recompensa este colectată separat de algoritmul reinforcement learning, și mai târziu este folosită pentru a calcula o modificare. Specificând doar interfața agentului, Gym dă voie utilizatorilor să construiască agenți în oricare dintre cele două stiluri

- Sublinierea complexității exemplului, nu doar a performanței finale. Performanța unui algoritm reinforcement learning într-un mediu poate fi măsurată în două moduri: primul, performanța finală, a doua, timpul de învățare - complexitatea exemplului. Mai specific, performanța finală se referă la recompensa medie per episod, după ce învățarea s-a încheiat. Timpul de învățare poate fi măsurat în mai multe feluri. Un mod simplu de a măsura timpul de învățare este de a număra episoadele până când se depășește un prag(media nivelului de performanță). Acest prag este ales în funcție de mediu într-un mod ad-hoc. De exemplu, performanța de 90% poate fi obținută doar de un agent foarte bine antrenat. Atât performanța finală cât și complexitatea exemplului sunt foarte interesante, dar pot fi folosite calcule arbitrară pentru a spori performanța evaluarii finale, astfel comparația constă mai mult în resursele computaționale decât în calitatea algoritmului.
- Încurajarea feedback-ului, nu competiția. Site-ul lui OpenAI Gym oferă posibilitatea utilizatorilor să compare performanțele dintre algoritmi. Una dintre inspirații este Kaggle, care găzduiește un număr de competiții machine learning cu clasamente. Dar, scopul clasamentelor pentru Gym nu este competiția, ci de a stimula ideea de a împărtășii codul și alte idei, și de a fi un reper pentru evaluarea diferitelor metode de abordare. Reinforcement learning prezintă noi procări pentru repere. În supervised learning, performanța este măsurată de acuratețea predicțiilor pe un set de date, unde output-ul corect este ascuns de concurenți. În reinforcement learning, este mai indirect să se măsoare o performanță generală, în afară de a rula codul utilizatorului pe un set de medii pe care agentul nu a fost antrenat până atunci, dar asta ar fi costisitor din punct de vedere computațional. Fără un set de date de test ascuns, cineva ar trebui să verifice că algoritmul nu a făcut "overfit" pe problemele pe care a fost testat(de exemplu, prin reglarea parametrilor). OpenAI încurajează feedback-ul între utilizatori pentru in-

interpretarea rezultatelor obținute. Astfel, OpenAI Gym cere utilizatorilor să creeze notițe ce descriu un algoritm, parametrii folosiți și linkarea către codul sursă. Notițele ar trebui să ofere posibilitatea utilizatorilor să reproducă rezultatele. Cu codul sursă la îndemână, este posibil să se evaluateze dacă un algoritm face "overfit" sau nu.

- Versionare strictă a mediilor. Dacă un mediu se modifică, rezultatele de dinainte și după schimbare pot fi incomparabile. Pentru a evita astfel de probleme, după orice schimbare a mediului va fi acompaniată de o creștere a versiunii. De exemplu, dacă versiunea inițială a problemei CartPole este numită Cartpole-v0, și dacă funcționalitățile se modifică, numele va fi schimbat în Cartpole-v1.
- Monitorizare implicită. Mediile sunt construite cu un Monitor, care înregistrează de fiecare dată când un step (un pas al simulării) și reset (sampling a new initial test) sunt chemate. Comportamentul Monitorului este configurabil, și poate fi înregistrat video.

În momentul lansării lui Gym, următoarele medii au fost incluse:

- Classic control and toy text - probleme mici de reinforcement learning în literatură.
- Aglorithmic - Efectuarea calculelor cum ar fi adunarea a numerelor cu multe cifre și secvențe recursive. Majoritatea problemelor necesită memorie, și dificultatea lor poate fi aleasă prin schimbarea lungimii unei secvențe.
- Atari - jocurile clasice Atari, cu screen images sau RAM ca input, folosind Arcade Learning Environment.
- Board games - jocul Go cu dimensiunile de 9X9 și 19X19, unde motorul Pachi servește ca oponent.
- Roboți 2D și 3D - se poate controla un robot într-o simulare. Aceste probleme folosesc motorul de simulare a fizicii MuJoCo, care a fost proiectat pentru simulării rapide și cu acuratețe crescută a roboților.

De la momentul lansării au fost create mai multe medii, inclusiv cele bazate pe motoarele fizice open-source Box2D sau Doom bazat pe motorul de joc VizDoom.

Eu am folosit Gym în aplicația mea pentru a crea mediile pe care le folosesc. De exemplu, în aplicație se folosesc următoarele medii atari: BreakoutDeterministic-v4, PongDeterministic-v4, SpaceInvadersDeterministic-v4. Deterministic se referă la frameskip. Frameskip este o metodă prin care se săracă anumite cadre ale uni-

animații pentru a crește performanța în detrimentul calității vizuale. V4 reprezintă probabilitatea ca o acțiune să se repete. V0 are probabilitatea de 0.25 (asta înseamnă că în 25% din cazuri acțiunea precedentă va fi folosită, în loc de o acțiune nouă). V4 are probabilitatea de 0.

Capitolul 3

Tehnici Deep Learning utilizate

În acest capitol sunt prezentate pe larg rețelele neurale și noțiuni generale despre acestea. În plus, se vor da mai multe detalii despre reinforcement learning în contextul deep learning, Q-Learning, Dueling Networks și Double Q learning.

1 Rețele neurale

Rețelele neurale artificiale (ANNs) sau mai simplu rețele neurale (NNs), sunt sisteme computaționale inspirate de rețelele neurale care constituie creierul uman. O rețea neurală este concepută dintr-o colecție de noduri conectați numiți neuroni artificiali, care se aseamănă cu neuronii din corpul uman. Fiecare conexiune, asemănător cu sinapsele, pot transmite un semnal către neuroni. Un neuron artificial primește acest semnal și îl procesează pentru a îl transmite mai departe către următorii neuroni conectați. "Semnalul" este procesat ca fiind un număr real, iar valoarea de ieșire a neuronilor va fi calculată de către o funcție de activare (o funcție neliniară) ca sumă de valori de intrare. Aceste conexiuni poartă și numele de vârfuri.

Noțiuni generale în contextul rețelelor neurale:

- Neuroni - sau nodurile, reprezintă unitatea de bază a unei rețele neurale. Primește diferite input-uri și o valoare de bias. Când primește un semnal, acesta se înmulțeste cu o valoare numită weight (pondere).
- Conexiuni - conectează un neuron dintr-un strat cu alt neuron din alt strat sau cu un alt neuron din același strat. Unei conexiuni îi este întotdeauna asociată o pondere. Scopul antrenării unei rețele este de a modifica această pondere pentru a scădea loss-ul (eroarea)

- Bias - sau decalaj, este input extra al neuronilor care este întotdeauna +1, și are propria conexiune ponderată. Astfel se asigură că și atunci când toate input-urile sunt nule (toate 0), va exista o activare în neuron.
- Funcție de activare - funcția de activare este folosită pentru a introduce nonliniaritatea în rețelele neurale. Funcțiile de activare micșorează valorile într-un interval, de exemplu funcția de activare sigmoidă micșorează valorile în intervalul [0,1]. Se folosesc multe funcții de activare în deep learning [3.1](#), dar ReLU, SeLU, TanH sunt de preferat în comparație cu funcția de activare sigmoidă .

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
Arctan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parametric Rectified Linear Unit (PReLU) ^[2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) ^[3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

Figure 3.1: O selecție de funcții de activare în deep learning.[\[27\]](#)

- Shape - sau forma input-ului. Forma input-ului este forma matricii pe care o introducem în stratul de intrare a rețelei. Dacă o rețea are un strat de intrat cu 4 neuroni și se așteaptă să primească o dată de intrare, atunci forma ideală ar fi $(1, 4, 1)$, dacă introducem pe rând datele. Dacă introducem 100 de date în același timp, forma ar arăta astfel $(100, 4, 1)$.

- Weight - sau ponderea, reprezintă cât de puternică este o conexiune dintre doi neuroni. Dacă ponderea de la nodul 1 la nodul 2 are o valoare mai mare, înseamnă că neuronul 1 are o influență mai mare decât a neuronului 2. Ponderile aproape de 0 înseamnă că, modificând valoarea input-ului nu va schimba output-ul. Ponderile negative înseamnă că dacă am crește input-ul, output-ul va descrește.
- Forward propagation - sau pasul de propagare înainte este un proces prin care se dă rețelei valorile de input și obținem ca output o valoare numită predicted value. Câteodată se referă la forward propagation ca inferență. Când se introduc inputurile în primul strat al rețelei neurale, acesta se introduce fără calcule. Al doilea strat preia valoarea de la primul, aplică operațiile de înmulțire, adunare și activare și dă mai departe valoarea către următorul strat. Acest proces se repetă de mai multe ori în straturile următoare, până când obținem o valoare output din ultimul strat [3.2](#).
- Back-Propagation - după pasul de propagare înainte obținem o valoare output pe care o numim predicted value. Pentru a calcula eroarea, comparăm predicted value cu actual output value. Folosim o funcție de loss pentru a calcula valoarea erorii. După se calculează derivata valorii de eroare, în raport cu valorile ponderilor de pe ultimul strat. Numim aceste derive, gradients(gradienți) și folosim valorile gradienților pentru a calcula gradienți din stratul precedent. Repetăm acest proces, până când obținem gradienți fiecarei pondere din rețea neurală. Apoi scădem din valoarea ponderilor acest valori ale gradienților pentru a reduce eroarea. În acest mod, ne îndreptăm (descent) către o valoare de Local Minima (valoare minimă de loss).
- Learning rate - când antrenăm o rețea neurală este ușuală folosirea gradient descent-ului pentru a optimiza ponderile. La fiecare iterație folosim back-propagation pentru a calcula derivata funcției de loss în raport cu ponderile, și o scădem din ponderi. Learning rate determină cât de repede sau cât de încet se actualizează fiecare pondere. Learning rate ar trebui să fie suficient de înalt pentru a nu dura foarte mult ca funcția să conveargă și suficient de mică pentru a găsi un minim local.
- Accuracy - acuratețea se referă la cât de aproape este o predicție de o valoare standard sau o valoare cunoscută.
- Precision - precizia se referă la cât de aproape sunt două sau mai multe valori predicții una de cealaltă. Mai exact, repetarea unei predicții.

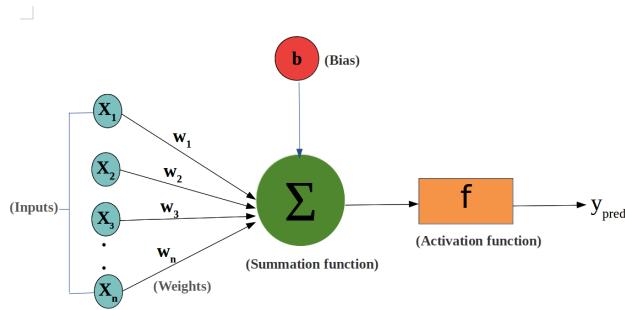


Figure 3.2: Scheletul unei rețele neurale.[7]

Există mai multe tipuri de rețele neurale, în funcție de dispozitivul disponibil. De exemplu, tipul perceptronului, feedforward, perceptron multistrat, convoluționale, recurente etc sunt câteva dintre tipurile de rețele neurale. Perceptronul este cel mai simplu neuron. Acesta este capabil să învețe să separe două multimi de puncte dar pot fi separate liniar. Perceptronii multistrat sau rețelele neurale multistrat sunt cele mai cunoscute tipuri de rețele neurale și sunt incluse în familia rețelelor cu propagare înainte. Perceptronul multistrat este o arhitectură de tip multistrat, feedforward. Neuronii perceptronului nu se numesc perceptroni. Ele se numesc tot neuroni, dar cu funcție de activare neliniară. De regulă, o rețea multistrat este formată din cel puțin trei straturi:

- Stratul de intrare - preia valorile din input-uri. Stratul de intrare nu este format din neuroni, aceasta fiind o caracteristică principală a acestuia. În plus, acesta nu are rol computațional
- Există minim un hidden layer (strat ascuns), compus din neuroni 3.3.
- Stratul de ieșire - este un strat care produce valori estimate care apoi sunt comparate cu output-urile dorite.

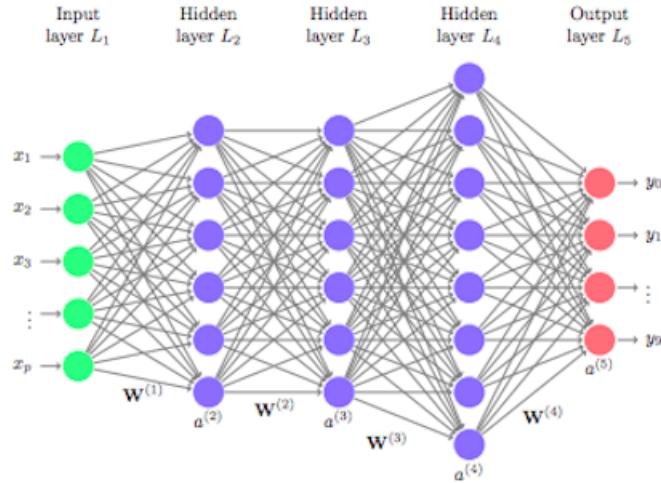


Figure 3.3: Straturile ascunse dintr-o rețea neurală.[20]

Numerotarea straturilor începe de la 0. Neuronii din straturile ascunse extrag niște trăsături (features) din datele de intrare(vectori de intrare), trăsături care sunt necesare rețelei neurale pentru producerea unei estimări. Este posibil să existe mai mulți neuroni în stratul de ieșire. De regulă, se consideră că o valoare constantă, +1, bias, care se adaugă pe lângă valorile de intrare sau valorile calculate de către un strat de neuroni, crește eficiența în momentul antrenării. Atât ponderile dintre straturi cât și ponderile de bias sunt instruibile, adică se vor modifica în timpul învățării.

Se consideră $L \geq 3$, unde L reprezintă numărul total de straturi, adică straturile de intrare, straturile ascunse și stratul de ieșire. În fiecare hidden layer l ($1 \leq l \leq L-1$) există un număr de noduri(neuroni) n_l . În primul strat, stratul de intrare, se află $n_0 = n$ neuroni, iar numărul de neuroni din stratul de ieșire este $n_{L-1} = m$, care este dat de numărul de multimi (sau clase) pentru problemele de clasificare sau estimare sau de numărul ieșirilo care urmează să fie approximate (la regresie).

Notății pentru înțelegerea rețelelor neurale:

- p - vectorul de intrare din setul de instruire
- $x^{(i)}$ - vectorul de intrare din setul de instruire, $d^{(i)} = (x_1^{(i)}, \dots, x_n^{(i)})^t$, $1 \leq i \leq p$
- $d^{(i)}$ - ieșirea asociată intrării $d^{(i)}$ din setul de instruire, $d^{(i)} = (d_1^{(i)}, \dots, d_m^{(i)})^t$, $1 \leq i \leq p$

- $z_i^{[l]}$ - este valoarea de activare a unui neuron i din stratul l , $1 \leq l \leq L - 1$, $(0 \leq i \leq n_l)$
- $\mathbf{z}^{[l]}$ - este vectorul care va conține valorile de activare ale neuronilor din stratul l , $\mathbf{z}^{[l]} = (z_1^{[l]}, \dots, z_{n_l}^{[l]})^t$, $1 \leq l \leq L - 1$
- $a_i^{[l]}$ - este valoarea de ieșire a neuronului i din stratul l , $1 \leq l \leq L - 1$, $(0 \leq i \leq n_l)$
- $\mathbf{a}^{[l]}$ - este vectorul care va conține valorile de ieșire ale neuronilor din stratul l , $\mathbf{a}^{[l]} = (a_1^{[l]}, \dots, a_{n_l}^{[l]})^t$, $1 \leq l \leq L - 1$
- $w_{ij}^{[l]}$ - este ponderea conexiunii dintre un neuron i din stratul l și neuronul j din stratul $l - 1$, $1 \leq l \leq L - 1$, $1 \leq i \leq n_l$, $1 \leq j \leq n_{l-1}$
- \mathbf{W}^l - este matricea de ponderi dintre stratul $l - 1$ și stratul l , $0 \leq l \leq L - 1$, $\mathbf{W}_{ij}^l = w_{ij}^{[l]}$, $1 \leq i \leq n_l$, $1 \leq j \leq n_{l-1}$
- $b_i^{[l]}$ - este ponderea bias pentru un nod i din stratul l , $1 \leq l \leq L - 1$, $(1 \leq i \leq n_l)$
- $\mathbf{b}^{[l]}$ este vectorul care va conține ponderile bias către stratul l , $\mathbf{b}^{[l]} = (b_1^{[l]}, \dots, b_{n_l}^{[l]}), 1 \leq l \leq L - 1$
- $f^{[l]}$ - este funcția de activare a neuronilor din stratul l , $1 \leq l \leq L - 1$
- \mathbf{W} - este secvența de matrice de ponderi ($\mathbf{W}^1, \mathbf{W}^2, \dots, \mathbf{W}^{L-1}$)
- \mathbf{b} - este secvența de vectori de ponderi ($\mathbf{b}^1, \mathbf{b}^2, \dots, \mathbf{b}^{L-1}$)
- $J(\mathbf{W}, \mathbf{b})$ - este eroarea empirică pentru un set de instruire sau minibatch
- $J(\mathbf{W}, \mathbf{b}; x^{(i)}, d^{(i)})$ - este eroarea pentru pereche de vectori de instruire $(x^{(i)}, d^{(i)})$, $1 \leq i \leq p$
- o^i - este vectorul coloană de ieșire pentru intrarea $x^{(i)}$, calculat de rețea

Valoare de activare a unui neuron i din stratul $l \geq 1$ este 3.1:

$$z_i^{[l]} = w_{i1}^{[l]} \cdot a_1^{[l-1]} + w_{i2}^{[l]} \cdot a_2^{[l-1]} + \dots + w_{i,n_{l-1}}^{[l]} \cdot a_{n_{l-1}}^{[l-1]} + b_i^{[l]} = \mathbf{W}_i^{[l]} \cdot \mathbf{a}^{[l]} + b_i^{[l]} \quad (3.1)$$

Această formulă se poate scrie matriceal astfel 3.2:

$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]} \cdot \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]} \quad (3.2)$$

Obținând valoare de activare a neuronului i din stratul l , ieșirea se calulează folosind funcția de activare $f^{[l]}$ 3.3:

$$a_i^{[l]} = f^{[l]}(z_i^{[l]}) \quad (3.3)$$

Dacă funcția se aplică pe fiecare valoare din vectorul $z^{[l]}$ m atunci ecuație devine 3.4:

$$\mathbf{a}^{[l]} = f^{[l]}(\mathbf{z}^{[l]}) \quad (3.4)$$

Pentru calcularea de forward propagation, este nevoie de funcția de activare. Pentru calcularea de back-propagation, derivata funcției de activare este necesară.

O selectiune de funcții de activare [18]:

1. Funcția logistică sigmoidă 3.5:

$$f = \sigma : \mathbb{R} \rightarrow (0, 1), f(z) = \sigma(z) = \frac{1}{1 + \exp(-z)} \quad (3.5)$$

Derivata funcției este 3.6:

$$f'(z) = \sigma' = \sigma(z)(1 - \sigma(z)) = f(z) \cdot (1 - f(z)) \quad (3.6)$$

2. Funcția tangentă hiperbolică 3.7:

$$f = \tanh : \mathbb{R} \rightarrow (-1, 1), f(z) = \tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)} \quad (3.7)$$

Derivata este 3.8:

$$f'(z) = \tanh'(z) = 1 - \tanh^2(z) = 1 - f^2(z) \quad (3.8)$$

Între funcția σ și funcția \tanh există relația 3.9:

$$\tanh(z) = 2 \cdot \sigma(2z) - 1 \quad (3.9)$$

În practică, de cele mai multe ori, funcția tangentă hiperbolică obține rezultate mai bune decât funcția logistică sigmoidă.

3. Funcția liniară 3.10:

$$f(z) = a \cdot z + b \quad (3.10)$$

Derivata acestei funcții este 3.11 :

$$f(z) = a; \quad (3.11)$$

De cele mai multe ori se consideră $a = 1, b = 0$ și funcția este folosită dacă se dorește ca valorile la ieșire să fie în afara intervalelor $(0,1)$ și $(-1,1)$

4. Funcția softmax [3.12](#):

$$\text{softmax}(z; c) = \frac{\exp(z_c)}{\sum_{i=1}^m \exp(z_i)} \quad (3.12)$$

"c" este indicele neuronulu și m este numărul total de neuroni din stratul său. Funcția softmax este utilă pentru a obține o distribuție de probabilitate dintr-un vector de valori oarecare. Softmax se folosește de regulă pentru stratul de ieșire, iar valorile output se interpretează ca probabilitatea intrării curente să fie clasa c , $1 \leq c \leq m$, unde $\text{softmax}(z; c)$ este maxim.

Derivata funcției este [3.13](#):

$$\frac{\partial \text{softmax}(z; i)}{\partial z_j} = \begin{cases} \text{softmax}(z; i) \cdot (1 - \text{softmax}(z; i)) & \text{dacă } i = j \\ -\text{softmax}(z; i) \cdot \text{softmax}(z; j) & \text{dacă } i \neq j \end{cases} \quad (3.13)$$

5. Funcția Rectified Linear Unit (ReLU) [3.14](#):

$$f(z) = \max(0, z) = \begin{cases} 0 & \text{dacă } z \leq 0 \\ z & \text{dacă } z > 0 \end{cases} \quad (3.14)$$

Derivata funcției este [3.15](#):

$$f(z) = \begin{cases} 0 & \text{dacă } z < 0 \\ 1 & \text{dacă } z > 0 \end{cases} \quad (3.15)$$

Derivata [3.15](#) este nedefinită când $z = 0$. Derivatele pe subintervale nu sunt dificil de calculat și cu execuție rapidă. Totodată, în comparație cu sigmoida și tangenta hiperbolică, ele nu saturează.

Funcția este liniară pe porțiuni, dar neliniară în ansamblu. Chiar dacă nu este derivabilă într-un punct, în practică acest lucru nu deranjează.

Grafurile acestor funcții se pot observa în tabelul [3.1](#) de mai sus.

Pasul de propagare înainte

După ce arhitectura rețelei este construită, adică numărul de straturi ascunse, numărul de neuroni în fiecare strat și funcția de activare, urmează instruirea și utilizarea rețelei. Forward propagation preia un vector de intrare $x = (x_1^{(i)}, \dots, x_n^{(i)})^t$ și realizează modificări în starea neuronilor, pornind de la intrare și acționează

succesiv asupra straturilor $1, \dots, L - 1$. Din acest motiv rețeaua face parte din familia cu numele "de propagare înainte", sau "feedforward". Ieșirile ultimului strat sunt folosite pentru predicție. De exemplu la regresie, pentru estimarea probabilității condiționate sau pentru clasificare. Deoarece stratul de intrare nu are rol computațional, valoarea de ieșire a acestuia este vectorul de intrare. Vectorul se numește și vector coloană, iar x este furnizat rețelei 3.16:

$$\mathbf{a}^{[0]} = x \quad (3.16)$$

Dacă din stratul $l - 1$ se cunosc valorile de ieșire ale nodurilor, se pot calcula valorile de activare ale neuronilor din stratul l , iar apoi valorile lor de ieșire 3.17.

$$\mathbf{z}^{[l]} = \mathbf{W}^l \cdot \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]} \mathbf{a}^{[l]} = f^{[l]} = (\mathbf{z}^{[l]}) \quad (3.17)$$

pentru straturile $l = 1, \dots, L - 1$, cu $f^{[l]}(\cdot)$ funcția de activare care se aplică pe fiecare componentă a vectorului. \mathbf{o} este vectorul de m valori de ieșire produs de către rețea 3.18:

$$\mathbf{o} = \mathbf{a}^{[L-1]} \quad (3.18)$$

Dacă se lucrează cu un set de date alcătuit din perechi, vectori intrare-ieșire, se poate concatena vectorii coloană de date într-o matrice \mathbf{X} , pe orizontală. De exemplu, pentru un set de instruire $S = \{(x^{(1)}, d^{(1)}), (x^{(2)}, d^{(2)}), \dots, (x^{(p)}, d^{(p)})\}$, matricea \mathbf{X} va arăta astfel 3.19:

$$\mathbf{X} = \begin{pmatrix} & & & \\ x^{[1]} & x^{[2]} & \dots & x^{[p]} \\ & & & \end{pmatrix} \quad (3.19)$$

Forward propagation se realizează cu pașii:

$$\mathbf{z}^{[1]} = \mathbf{W}^{[1]} \mathbf{X} + b^{[1]} \quad (3.20)$$

$$\mathbf{A}^{[1]} = f^{[1]}(\mathbf{z}^{[1]}) \quad (3.21)$$

$$\mathbf{z}^{[2]} = \mathbf{W}^{[2]} \mathbf{A}^{[1]} + b^{[2]} \quad (3.22)$$

$$\mathbf{A}^{[2]} = f^{[2]}(\mathbf{z}^{[2]}) \quad (3.23)$$

$$\dots \quad (3.24)$$

$$\mathbf{Z}^{[L-1]} = \mathbf{W}^{[L-1]} \mathbf{A}^{[L-1]} + b^{[L-1]} \quad (3.25)$$

$$\mathbf{A}^{[L-1]} = f^{[L-1]}(\mathbf{Z}^{[L-1]}) \quad (3.26)$$

unde $\mathbf{Z}^{[l]}$ și $\mathbf{A}^{[l]}$ sunt matrice cu n_l linii și p coloane. Se consideră pentru adunările (3.20, 3.22, 3.25) că se aplică mecanismul de "broadcasting". Prin copiere, vectorii $\mathbf{b}^{[l]}$, se obține o matrice care are același număr ca și matricele \mathbf{X} , respectiv $\mathbf{Z}^{[l-1]}$.

Funcțiile de cost

Pentru fiecare pereche de tipul $(x^{(i),d^i} \in S | 1 \leq i \leq p)$ se va produce o valoare pentru funcția de eroare în felul următor: ca intrare în rețea se introduce vectorul $x^{(i)}$ și se calculează un vector de ieșire (o^i) , care reprezintă estimarea produsă de rețea pentru intrarea introdusă. Apoi, se folosește o funcție de cost, sau de eroare, $J(\mathbf{W}, \mathbf{b}; x^{(i)}, d^{(i)})$. Această valoare se dorește să fie cu atât mai mică cu cât vectorul $o^{(i)}$ este mai apropiat de $d^{(i)}$, respectiv mai mare cu cât cei doi vectori sunt mai departați. De asemenea, se mai consideră o valoare, un factor de regularizare, care împiedică un comportament haotic al rețelei: de exemplu, variațiile mici ale intrării duc la salturi mari în straturile ascunse și la ieșire.

Forma funcției de eroare este în general 3.27:

$$J(\mathbf{W}, \mathbf{b}) = \underbrace{\left[\frac{1}{p} \sum_{i=1}^p \overbrace{J(\mathbf{W}, \mathbf{b}; x^{(i)}, d^{(i)})}^{\text{eroare empirică}} \right]}_{\text{Eroarea empirică pe tot setul de antrenare}} + \underbrace{\frac{\lambda}{2} \sum_{l=1}^{L-1} \sum_{i=1}^{n_{l-1}} \sum_{j=1}^{n_l} (w_{ji}^{[l]})^2}_{\text{Factor de regularizare}} \quad (3.27)$$

unde $\lambda > 0$ este coeficientul de regularizare. Ulimul termen este regulaizare L_2 , adică o sumă de patrare de norme Frobenius 3.28, peste matricele $\mathbf{W}^{[1]}, \dots, \mathbf{W}^{[L-1]}$.

$$\sum_{i=1}^{n_{l-1}} \sum_{j=1}^{n_l} (w_{ji}^{[l]})^2 \stackrel{\text{def}}{=} \|\mathbf{W}^{[l]}\|_F^2 \quad (3.28)$$

Straturile de convoluție

Straturile de convoluție sunt straturi ascunse ale rețelelor convoluționale. În rețelele convolutionale, input-ul este un tensor cu forma (numărul de input-uri) x (înălțimea input-ului) x (lățimea input-ului) x (canalele input-ului). După ce un input este trecut print-un strat de convoluție, input-ul (o imagine) se abstracționează într-un feature map (sau activation map) cu forma (numărul de input-uri) x (înălțimea feature map-ului) x (lățimea feature map-ului) x (canalele feature map-ului). Un strat de convoluție într-un CNN (rețea neurală convoluțională) are următoarele atrbute:

- Filtre/kernel-uri conoluționale definite de înălțime și lățime (hiper-parametri)
- Numărul de canale de intrare și ieșire (hiper-parametri). Numărul de canale de intrare ale unui input dintr-un strat trebuie să fie egal cu numărul de canale de ieșire. Acest lucru se mai numește și adâncimea (depth) input-ului.
- Hiperparametri adiționali pentru operațiile de conoluție cum ar fi padding, stride și dilatare.

După o conoluție, în general se folosește funcția de activare ReLU [3.14](#).

O detaliere a rețelelor conoluționale și a straturilor de conoluție se poate regăsi în [\[16\]](#).

2 Reinforcement Learning

Reinforcement learning este un subdomeniu în cadrul machine learning care se preocupă de agenții inteligenți și cum ar trebui să acționeze aceștia într-un mediu pentru a maximiza o recompensă. Reinforcement learning este una dintre principalele paradigmă machine learning, împreună cu supervise learning și unsupervised learning.

Pentru a înțelege reinforcement learning-ul se propun câteva terminologi care vor ajuta pe parcursul lucrării, cum ar fi: agenți inteligenți, medii, stări, acțiuni și recompense. Literele mari de obicei evidențiază un set de lucruri în timp ce literele mici evidențiază o instanță specifică a acelui set (de ex "A" reprezintă toate acțiunile posibile, "a" reprezintă o acțiune specifică ce aparține unui set).

- Agent intelligent - un agent care va efectua o acțiune. De exemplu o dronă care face o livrare sau Super Mario navigând un joc. Algoritmul este reprezentat de agent, fiind folosit să considerăm că în viața reală, omul reprezintă agentul.
- Acțiuni - "A" este un set cu toate mișările posibile pe care le poate face un agent. O acțiune este de la sine înțeles ce reprezintă, dar ar trebui ținut minte că agentul de obicei alege dintr-o listă discretă de acțiuni posibile. Astfel, într-un joc video, lista de acțiuni poate include fuga de la dreapta la stanga, ghemuirea sau statul pe loc. În timpul controlării unei drone, pot apărea diferite viteze și accelerări într-un spațiu 3D.
- Discount rate (DR) - acesta este multiplicat de către recompensele viitoare atunci când sunt descoperite de către agent pentru a micșora efectul recompenselor asupra acțiunii alese. Este conceput în acest mod pentru a face

recompensele viitoare să valoreze mai puțin decât recompensele imediate. Adică impune un concept pe terment scurt de hedonism [34] agentului. De obicei reprezentat prin litera grecească γ . Dacă γ este 0,8, și ar fi o recompensă de 10 puncte după 3 mișcări, valoarea acelei recompense este de $0,8^3 \times 10$. Un discount factor de 1 ar face următoarele recompense să valoreze la fel de mult care recompensele imediate. Este o luptă împotriva recompenselor întârziate.[40]

- Mediul - lumea în care agentul acționează. Mediul preia starea curentă a agentului ca input și returnează ca output recompensa și următoarea stare. În viața reală, mediul ar putea fi reprezentat de legile fizicii și de regulile societății care evaluează acțiunile unui om și determină consecințele lor.
- Starea - este o situație concretă în care agentul se află, de exemplu, un loc specific sau un moment, o configurație instantanee care pune agentul în relație cu alte lucruri semnificative, cum ar fi obstacole, inamici, premii. Poate fi returnată de către mediu sau orice situație viitoare.
- Recompensă - "R". O recompensă este un feedback cu care măsurăm rata de succes sau eșec a unei acțiuni a agentului într-o stare dată. De exemplu, într-un joc video, când Mario atinge o monedă, acesta câștigă puncte. În orice stare, agentul trimite output-ul (sub formă de acțiuni) către mediu, iar acesta returnează agentului o nouă stare și totodată recompensa, dacă primește una. Recompensa poate fi imediată sau întârziată. Ea evaluează efectiv acțiunea unui agent.
- Metoda - " π ". Metoda reprezintă strategia pe care agentul o adoptă pentru a determina următoarea acțiune bazată pe starea curentă. Își plănuiește acțiunile bazate pe stări, acțiuni care promit cele mai mari recompense.
- Valoarea - "V". Randamentul preconizat pe termen lung după discount, spre deosebire de recompensa pe termen scurt "R". Funcția " $V\pi(s)$ " este definită ca randamentul preconizat pe termen lung sub metoda " π ". Reducem recompensele, reducem valoarea lor estimată, cu cât se apar mai târziu. Reducem valoarea recompenselor viitoare deoarece dorim să distingem valoarea de Q-value sau valoarea-acțiunii (Q-value).
- Q-value - este similară cu valoarea "V", cu excepția faptului că funcția primește un parametru extra, "a", acțiunea curentă. " $Q\pi(s, a)$ " se referă la randamentul pe termen lung a unei acțiuni "a" care se află sub metoda " π " în starea curentă "s". Q marchează perechile de stare-acțiune cu recompensele.

- Procesul de decizie Markov (MDP) [36] - este un model probabilistic a unei probleme cu decizii secvențiale, unde stările sunt percepute în întregime, și starea curentă alături de acțiunea selectată determină o probabilitate de distribuire a stărilor viitoare. În alte cuvinte, rezultatul unei acțiuni dintr-o stare depinde doar de startea curentă și de acțiunea respectivă (și nu de acțiunile sau stările precedente).
- Programarea dinamică (DP) [33] - este o clasă de soluții pentru rezolvarea problemei deciziilor secvențiale cu un "compositional cost structure". Richard Bellman a fost unul dintre principalii fondatori a acestei metode.
- Monte Carlo methods [37] - este o clasă de funcții cost pentru învățare, care estimează valoarea unei stări prin mai multe încercări, începând de la acea stare și face media a recompenselor totale obținute din acele încercări.
- Temporal Difference (TD) algorithms [4] - este o clasă de metode de învățare, bazate pe ideea de a compara predicții succesive din punct de vedere temporal. Este posibil să fie idea de bază când vine vorba de reinforcement learning.

3 Q-Learning

Q-Learning este un algoritm "model-free" din reinforcement learning, care învață valoarea unei acțiuni în funcție de o stare. Astfel, pentru a rezolva probleme de tranziții și recompense stochastice [41], nu are nevoie de adaptări.

Pentru orice problemă de tip FMDP (finite Markov decision process), începând de la starea curentă, Q-learning găsește o metodă optimă pentru a maximiza valoarea dorită a recompenselor totale peste oricare pas succesiv. Dacă îi este oferit destul timp și o regulă parțial aleatorie, Q-learning poate identifica o politică de selectare a acțiunilor pentru orice FMDP.

Când vine vorba de reinforcement learning, două tipuri de învățare sunt populare:

1. Policy Based - prin această abordare, o politică, adică o funcție, care asociază o stare cu o acțiune este optimizată. Odată ce avem o funcție bine definită, agentul determină care este cea mai bună acțiune, oferind starea curentă ca input pentru funcție. Această abordare se poate împărții mai departe în două tipuri: "deterministic" și "stochastic". Această abordare poate fi împărțită mai departe în:

- "Deterministic" - o funcție care returnează la o stare dată o acțiune unică.
 $S=(s) \rightarrow A=(a)$

- "Stochastic" - în loc de a returna o acțiune unică, funcția returnează o probabilitate de distribuție a acțiunilor într-o stare dată.
Policy $\rightarrow p(A = a | S = s)$

2. Value Based - în această abordare, obiectivul este de a optimiza o funcție de cost, o funcție (se poate considera ca un Lookup table) ce la o stare curentă planifică recompensele viitoare. Valoarea fiecărei stări este recompensa totală pe care un agent de tip reinforcement learning se poate aștepta să o primească până la îndeplinirea scopului.

Q-Learning se află în categoria algoritmilor de tip Value-based. Scopul lui Q-learning este de a optimiza o funcție cost potrivită pentru problemă. "Q" înseamnă calitate ("quality"); ajută la găsirea acțiunii potrivite, rezultând astfel o stare "de cea mai bună calitate". Valorile sunt memorate într-un tabel (Q-Table).

Pentru a înțelege mai bine cum funcționează Q-learning, se va considera un joc simplu 2D într-un tabel de 4x4 [3.4](#).



Figure 3.4: Un joc 2D reprezentat într-un tabel 4x4.[8]

Scopul: Ghidează copilul către parc.

Sistemul de recompense: A. Adună bomboană = +10 puncte B. Întâlnește un câine = -50 puncte C. Ajunge în parc = +50 puncte

Finalul unui Episod: A. Întâlnește un câine. B. Ajunge în parc.

Se va observa mai se departe cum un agent de tip Q-learning va juca acest joc. În primul rând, se crează un Q-table unde se vor memora toate valorile asociate fiecărei stări. Tabelul are numărul de linii egal cu numărul de stări din problem, adică 16 în cazul nostru, iar numărul de coloane este egal cu numărul de acțiuni pe care agentul le poate lua, adică 4 în cazul nostru (sus, jos, stânga, dreapta) [3](#).

ACTIONS/ STATES	UP	DOWN	LEFT	RIGHT
1(START)	0	0	0	0
2	0	0	0	0
.....	0	0	0	0
16	0	0	0	0

Table 3.1: Un Q-table pentru un joc de 4x4 în dimensiunea 2D)

Pasul 1: Inițializarea

Când agentul se joacă pentru prima dată acest joc, nu are nicio informație despre acesta, deci vom inițializa tabelul cu valori de 0.

Pasul 2: Explotare sau Explorare

Acum agentul poate interacționa cu mediul în două moduri: poate să folosească informațiile deja dobândite din Q-table, adică să exploateze aceste informații, ori poate să exploreze mai departe. Explotarea devine foarte folositoare când un agent a trecut printr-un număr mare de episoade și posedă informații despre mediu. În schimb, explorarea devine importantă atunci când un agent este "naiv" și nu are destulă experiență. Acest schimb (tradeoff) între exploatare și explorare poate fi obținut printr-o funcție de cost epsilon. Ideal, în primele etape se dorește mai multă explorare, iar în final mai multă exploatare.

În pasul 2, agentul face o acțiune (explotare sau explorare).

Pasul 3: Evaluarea recompensei

După ce agentul ia o acțiune decisă în pasul 2, ajunge la următoarea stare, s' . În starei s' se poate realiza din nou 4 acțiuni, iar fiecare dintre ele duce la alt scor [3.5](#).

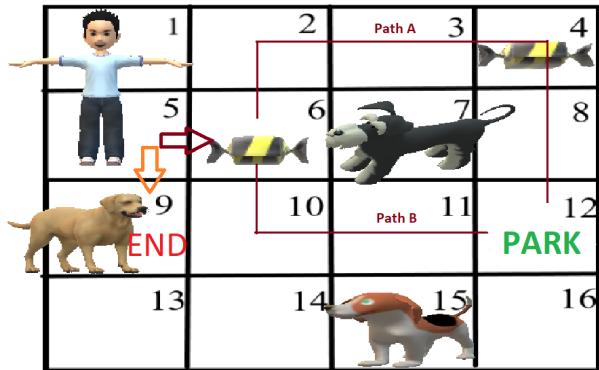


Figure 3.5: Scenarii posibile dacă agentul se mută în starea 5.[8]

De exemplu, copilul se mută de la 1->5 și acum poate merge fie spre 6 sau spre 9. Pentru a găsi valoarea recompensei pentru starea 5, vom calcula recompensele de la stările viitoare, adică în exemplul nostru, starea 6 și 9 și vom selecta valoarea maximă.

În starea 5, există 2 opțiuni (Pentru simplicitate nu se va include și reconstituirea pașilor) -

1. Merge spre 9 : Sfârșit de episod. 2. Merge spre 6 : La starea 6 sunt din nou 3 opțiuni -

1. Merge spre 7 : Sfârșit de episod 2. Merge spre 2 - Continuă fiecare pas până când se termină episodul și găsește recompensa. 3. Merge spre 10 - Continuă acest pas, găsește recompensa.

Pentru drumul A, recompensa este = $10 + 50 = 60$

Pentru drumul B, recompensa este = 50

Recompensa maximă = 60 (drumul A)

Recompensele totale la starea 5: -50 (întâlnirea cu un câine la starea 9), $10+60$ (recompensa maximă de la starea 6 încolo)

Valoarea recompensei la starea 5 = $\text{Max}(-50, 10+60) = 70$

Pasul 4: Actualizare Q-table

Recompense calculate în pasul anterior sunt folosite pentru a actualiza valoarea la starea 5 folosind ecuația de optimizare a lui Bellman (modificată) 3.29

$$\underbrace{\text{New } Q(s, a)}_{\text{New Q value}} = \underbrace{Q(s, a)}_{\text{Current Q values}} + \alpha \underbrace{[R(s, a) + \gamma \max_{a'} Q'(s', a')]}_{\text{Reward}} - \underbrace{Q(s, a)}_{\text{Current Q values}} \quad (3.29)$$

unde α este Learning Rate și γ este Discount Rate. Learning rate este o constantă care determină ce pondere va avea valoarea nouă în comparație cu valoarea veche. Discount rate este o constantă care reduce efectul recompenselor viitoare (ia valori între 0,8 și 0,99), adică balansează efectul recompenselor viitoare în valorile noi. Agentul va itera prin acești pași și va obține un Q-table cu valori actualizate. Folosind acest Q-table, este ușor de ales o acțiune într-o stare, care va duce către o stare cu Q-value maxim.

4 Double DQN

În ultimii ani, au fost foarte multe succese folosind deep reinforcement learning. Dar, multe dintre acestea folosesc arhitecturi convoluționale (convolutional neural networks [29]), LSTMs (long-short term memory neural networks [35] sau auto-encoders [28]. Dueling networks [26] sunt un nouă arhitectură de rețele neurale pentru reinforcement learning de tip "model-free". Dueling networks sunt reprezentate prin doi estimatori separați, unul pentru funcția de cost într-o stare și unul pentru funcția de avantaj a unei acțiuni dependente de o stare. În general, este cunoscut faptul că algoritmul Q-learning poate supraestima valorile acțiunilor în unele condiții. Nu se știe dacă, în practică, aceste supraestimări sunt comune, sau dacă afectează performanța într-un mod negativ, și dacă pot fi prevenite în general. În mod particular, cel mai recent algoritm DQN, care combină Q-learning cu o rețea neurală, suferă de supraestimări semnificative în jocuri din mediul Atari 2600. În trecut, aceste supraestimări au fost corelate cu flexibilitatea scăzută a funcției de aproximare [24] și a zgomotului [9].

Beneficiul principal al dueling networks este că poate fi combinată ușor cu atât algoritmii din prezent cât și cei din viitor. Această arhitectură separă în mod explicit valorile stărilor și avantajele acțiunilor dependente de stări. Arhitectura constă în două canale 3.6, care reprezintă funcțiile pentru valoare și funcțiile pentru avantaj, în timp ce au în comun un modul convoluțional comun de învățare a unui feature. Cum se poate observa în figura 3.6, sus un Q-network cu un singur canal și jos se află Dueling Q-Network. Dueling Q-Network are două canale pentru a estima separat valoarea unei stări (scalar) și avantajul pentru fiecare acțiune. Ambele rețele oferă ca output un Q-value pentru fiecare acțiune.

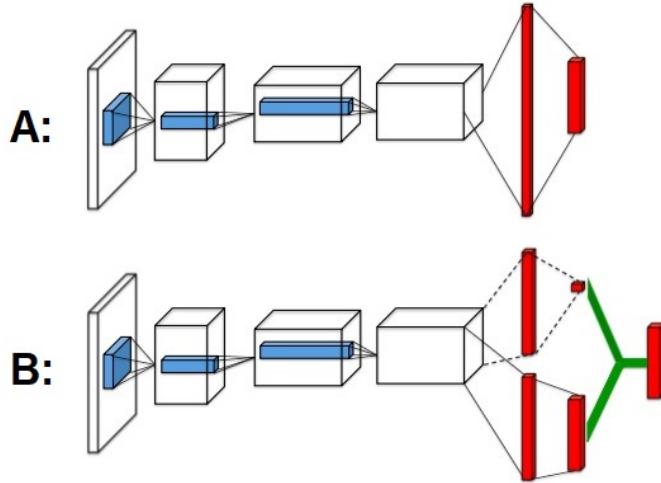


Figure 3.6: A: DQN obișnuită cu un singur canal pentru Q-values. B: Dueling DQN unde valoarea și avantajul sunt calculate separat și apoi combinate în stratul final într-un Q-value.[13]

Cele două canale sunt combinate de un strat de agregare special pentru a produce o estimare a valorii funcției state-action Q. Putem considera astfel că este o singură rețea Q cu două canale, doar că înlocuiește rețeaua cu un singur canal Q în algoritmii deja existenți cum ar fi Deep Q-Networks [3]. Dueling networks produc automat estimări separate a funcției de cost a stării și a funcției avantaj, fără nicio supraveghere extra.

Se consideră o problemă de luare a decizilor secvențială, în care un agent interacționează cu un mediu \mathcal{E} . Într-un mediu Atari, de exemplu, agentul primește un video s_t ce conține M cadre de imagini: $s_t = (x_{t-M+1}, \dots, x_t) \in S$ la pasul t . Agentul apoi alege o acțiune din $a_t \in A = \{1, \dots, |A|\}$ și remarcă un semnal de recompensă r_t produs de către emulatorul jocului. Agentul caută să maximizeze R_t (discounted return), unde $R_t = \sum_{\tau=t}^{\infty} \gamma^{\tau-t} r_{\tau}$. În această formulare, $\gamma \in [0, 1]$ este discount factor care realizează căre decide importanța recompenselor immediate cu cele următoare.

Pentru un agen care se comportă după o regulă stochastică π , valoarea perechii (s, a) și a stării s sunt definite astfel 3.30:

$$\begin{aligned} Q^{\pi}(s, a) &= \mathbb{E}[R_t | s_t = a_t = a, \pi] \\ V^{\pi}(s) &= \mathbb{E}_{a \sim \pi(s)}[Q^{\pi}(s, a)] \end{aligned} \quad (3.30)$$

Funcția Q 3.30 se poate calcula recursiv cu programarea dinamică:

$$Q^{\pi}(s, a) = \mathbb{E}_{s'}[r + \gamma \mathbb{E}_{a' \sim \pi(s')}[Q^{\pi}(s', a')]] | s, a, \pi \quad (3.31)$$

Definim $Q^*(s, a) = \max_{\pi} Q^\pi(s, a)$ ca fiind optim. Sub poliță deterministă $a = \arg\max_{a' \in A} Q^*(s, a')$, urmează ca $V^*(s) = \max_a Q^*(s, a)$. Din asta obținem Q^* satisfacă ecuația lui Bellman 3.32:

$$Q^*(s, a) = \mathbb{E}_{s'}[r + \gamma \underbrace{\max_{a'}}_{a'} Q^*(s', a') | s, a] \quad (3.32)$$

Definim o altă funcție importantă, funcția avantaj 3.33:

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s) \quad (3.33)$$

Se reține că

$$\mathbb{E}_{a' \sim \pi(s')} [A^\pi(s, a)] = 0 \quad (3.34)$$

Funcțiile de cost precedente sunt obiecte de mai multe dimensiuni. Pentru a le aproxima folosim un Q -network: $Q - (s, a; \theta)$ de parametru θ . Pentru a estima această rețea, optimizăm următoarea secvență de funcții de loss la iterația i [3.35][3.36]:

$$L_i(\theta_i) = \mathbb{E}_{s, a, r, s'} \left[(y_i^{DQN} - Q(s, a; \theta_i))^2 \right] \quad (3.35)$$

cu

$$y_i^{DQN} = r + \gamma \underbrace{\arg\max}_{a'} Q(s', a'; \theta^-) \quad (3.36)$$

unde θ^- reprezintă parametrii fixați a unei rețele separată, rețea țintă. Se poate încerca învățarea Q-learning standard pentru a învăța parametrii unei rețele $Q(s, a; \theta)$, dar, acestă estimare va performa slab în practică. O invoație cheie în [3] a fost de a îngheța parametrii rețelei țintă $Q(s', a', \theta^-)$ pentru un număr fix de iterații, în timp ce se actualizează rețeaua $Q(s, a; \theta_i)$, cu gradient descent. (această metodă îmbunătățește semnificativ stabilitatea algoritmului). Actualizarea specică a gradientului este 3.37:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s, a, r, s'} \left[(y_i^{DQN} - Q(s, a; \theta_i)) \nabla_{\theta_i} Q(s, a; \theta_i) \right] \quad (3.37)$$

Această abordare este "model-free" prin faptul că stările și recompensele sunt produse de către mediu. Totodată, este "off-policy" deoarece stările și recompensele sunt obținute printr-un "behaviour policy" (epsilon greedy în DQN).

Un alt punct cheie al succesului DQN este "experience replay". În timpul antrenării, un agent acumulează un dataset $\mathcal{D}_t = \{e1, e2, \dots, et\}$ al experiențelor $e_t = (s_t, a_t, r_t, s_{t+1})$ din mai multe episoade. Când se antrenează Q-network, în loc să se folosească doar experiențele prescrise de învățarea temporal-difertă standard [4], rețeaua este antrenată probarea unor mini-pachete (batches) de experiențe din \mathcal{D} în mod uniform și aleatoriu.

Loss-ul va lua forma următoare 3.38:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{U}(\mathcal{D})} \left[(y_i^{DQN} - Q(s, a; \theta_i))^2 \right] \quad (3.38)$$

Acstea sunt componentele principale ale DQN 3.7. Double DQN (DDQN) este un algoritm îmbunătățit al algoritmului propus de [10] se poate observa în figura. În Q-learning și DQN, operatorul max folosește aceleași valori pentru a selecta cât și a evalua o acțiune. Acest lucru poate duce la supraestimarea valorilor estimate. Pentru a estompa această problemă. DDQN folosește următoarea rețea tință 3.39:

$$y_i^{DDQN} = r + \gamma Q \underbrace{\operatorname{argmax}_{a'}}_{\mathbf{a}'} Q(s', a'; \theta^-) \quad (3.39)$$

Algorithm 1: deep Q-learning with experience replay.

```

Initialize replay memory  $D$  to capacity  $N$ 
Initialize action-value function  $Q$  with random weights  $\theta$ 
Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$ 
For episode = 1,  $M$  do
    Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequence  $\phi_1 = \phi(s_1)$ 
    For  $t = 1, T$  do
        With probability  $\epsilon$  select a random action  $a_t$ 
        otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$ 
        Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$ 
        Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$ 
        Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $D$ 
        Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $D$ 
        Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$ 
        Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  with respect to the
        network parameters  $\theta$ 
        Every  $C$  steps reset  $\hat{Q} = Q$ 
    End For
End For

```

Figure 3.7: Pseudocod pentru algoritmul DQN. [3]

Capitolul 4

Detalii de implementare

În acest capitol se vor urmări pașii necesari implementării, o detaliere a tehnicii de "exploration-exploitation trade-off" și arhitecturii replay memory, dar și un scurt ghid al utilizatorului.

1 Arhitectura rețelei

În dezvoltarea arhitecturii rețelei, un prim pas este acela de preprocesare. Pentru partea de preprocesare, se folosește gym. În cazul de față, gym asigură mediul Atari pentru rețea. Un frame returnat de către mediul Atari are forma (210, 160, 3), unde 3 reprezintă cele 3 canale de culoare RGB, 210 reprezintă înățimea frame-ului și 160 reprezintă lățimea frame-ului. Un astfel de frame este procesat de către o clasă *FrameProcessor*, care va transforma frame-ul în (84, 84, 1), unde 1 indică că în loc de cele 3 canale RGB este un singur canal pentru tonurile de gri și înățimea și lățimea devin 84.

Preprocesarea se realizează (conform [15]), deoarece, dacă se lucrează cu "cadre Atari neprocesate, care sunt imagini de 210 x 160, cu o gamă de culori de 128, poate fi solicitant din punct de vedere computațional. Astfel, preprocesăm frame-urile pentru a reduce dimensiunile. Frame-urile neprocesate se convertesc mai întâi din reprezentarea lor RGB în tonuri gri prin down-sampling, aducând imaginea la 110 x 84. Input-ul final este obținut prin "tăiere", rezultă o regiune din imagine de 84 x 84 care prinde în mare parte zona de interes a jocului. Ultima parte, partea de "tăiere", este necesară pentru că se folosesc convoluții 2D implementate pe GPU, unde așteprarea este ca input-ul să fie pătratic."

Pentru implementare se reține faptul că valorile pixelilor din input trebuie să fie normalizate între [0, 1]. Acest lucru se realizează prin împărțirea input-ului la 0xFF=255. Motivul acestei împărțiri se datorează faptului că valorile pixelilor din cadre, returnate de către mediul, sunt uint8, deci poate stoca valori în intervalul

$[0, 255]$.

Arhitectura descrisă în [15] sau [3], este înlocuită cu arhitectura de dueling networks din [26]. Atât arhitectura de dueling networks din Mnih2015 cât și Wang2016 folosesc aceeași structură "low-level" a straturilor de conoluție, după cum urmează:

- Primul strat de conoluție are 32 de filtre 8×8 cu un stride (pas) de 4.
- Al doilea strat constă în 64 de filtre 4×4 cu stride-ul 2.
- Ultimul strat de conoluție, stratul al treilea, constă în 64 de filtre de 3×3 cu stride de 1.

În arhitectura normală de DQN, vezi figura 3.6, situația A: are un strat final ascuns care este fully-connected. Valorile de ieșire sunt $Q(s, a; \theta)$ valori pentru acțiunea a în starea s .

Nu o să se obțină câte un singur Q -value pentru fiecare acțiune, deoarece situația B: împarte ultimul strat de conoluție în două canale. Funcția de cost are valoarea stării $V(s)$ care depinde doar de stare, iar funcția avantaj are valoarea $A(s, a)$ care depinde de stare și acțiunea respectivă. Această separare este folosită în cazul stărilor în care acțiunile nu afectează mediul într-un mod relevant. Implicit, vor exista două straturi fully connected.

Următoarea etapă constă în combinarea costului și avantajului în Q -values $Q(s, a)$. Acest lucru se face cu ajutorul ecuației 4.1:

$$Q(s, a) = V(s) + \left(A(s, a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a') \right) \quad (4.1)$$

Conform 3.34, nu se pot aduna pur și simplu $V(s)$ și $A(s, a)$.

Arhitectura rețelei folosită în aplicație arată în felul următor 4.1:

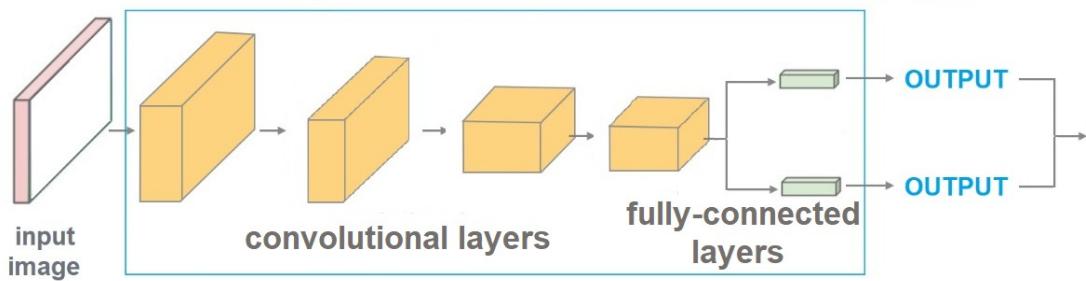


Figure 4.1: Arhitectura folosită în aplicație.

În comparație cu arhitecturile menționate precedent, în aplicația curentă, înainte de stratul fully connected, se adaugă un al patrulea strat de convoluție cu 1024 de filtre. Output-ul acestui strat va avea forma $(1, 1, 1024)$ și apoi va fi impărțit (cu funcția `tf.split`) în două canale cu formele $(1, 1, 512)$. În continuare, output-urile vor trece prin funcția de flatten, funcție care va converti harta de features rezultată într-o singură coloană, care este transmisă mai departe către straturile fully-connected din rețea (cu `tf.layers.dense`). Pentru inițializarea corectă a ponderilor, DQN folosește ReLU ca și funcție de activare, iar inițializarea corectă a ponderilor [11] se realizează cu `tf.compat.v1.variance_scaling_initializer`, unde `scale = 2`. Pentru optimizare, se folosește optimizatorul Adam.

În timpul antrenării, gradienții de eroare cresc semnificativ. Pentru a stabiliza rețeaua și pentru a nu obține valori NaN, se extrage eroarea din actualizare pentru a avea valori între -1 și 1. Această problemă, numită și exploding gradient, poate fi, până la un punct, evitată dacă se folosește un prag. Dacă gradientul este mai mare decât valoarea pragului (x), se consideră că valoarea gradientului este x . Pentru a extrage eroare se folosește funcția `tf.losses.huber_loss`.

2 Exploration-exploitation trade-off

În momentul de față, rețeaua poate prezice următoarea acțiune a sistemului. Rețeaua va considera cea mai bună acțiune (`self.best_action`) luând ca argument Q -value ca fiind maxim. Inițial, agentul nu știe strategia de joc. Dacă se exploatează în permanentă și nu se explorează, adică se alege întotdeauna cel mai mare Q -value (greedy), agentul va rămâne la o singură strategie pe care o descoperă și va returna mereu o recompensă mică. Algoritmul ϵ -greedy oferă o soluție simplă pentru această problema. Se alege de obicei o acțiune pe care rețeaua o consideră cea mai bună, dar cu o probabilitate ϵ se va alege o acțiune aleatorie. ϵ este o funcție a numărului de frame-uri pe care agentul le-a vazut. Pentru primele 50 000 de frame-uri agentul doar explorează, adică $\epsilon = 1$, iar pentru următoarele 1 milion de cadre, ϵ este linear scăzut până la valoarea 0.1, însențând că agentul va începe să exploateze din ce în ce mai mult. Dacă arhitecturiile precedente, valoare ϵ stagnează la 0.1, în aplicația dezvoltată în această lucrare valoarea descrește până la 0.01.

Funcția `get_action` implementează acest comportament. Se calculează mai întâi ϵ după numărul frame-ului curent și apoi se returnează o acțiune aleatorie cu probabilitatea ϵ sau acțiunea pe care DQN o consideră cea mai bună. Parametrii pe care îi primește constructorul sunt efectiv panta și ordonata la origine.

3 Replay Memory

Învățarea directă doar din cadre consecutive este ineficientă, datorită corelațiilor puternice dintre cadre; randomizând cadrele se îintrerup corelațiile și se reduce inconsistența actualizărilor. Așadar, dacă acțiunea maximizată este de a se mișca la stânga, atunci cadrele pentru antrenare vor fi dominate cu cadre pe partea stângă și vice-versa. Acest lucru poate duce la parametri blocați într-un minim local nedorit, sau pot diverge. Astă înseamnă că, atunci când se alege o acțiune și se realizează un pas pentru a obține o recompensă, rețeaua nu învăță de la ultimul pas, mai degrabă se adaugă tranziții în replay memory. Apoi alege un minibatch aleatoriu din replay memory pentru a realiza un pas de gradient descent.

Replay memory stochează ultimele 1 milion de tranziții. O tranziție este de forma $[state, action, reward, terminal, new_state]$:

- $state$ care este compus din 4 stări stochate împreună.
- $action$ care este acțiunea curentă
- $reward$ care este recompensa
- new_state care este produs de un frame după ce se realizează o acțiune și este pus în $state$ apoi se elimină cel mai vechi state.
- $terminal$ care este un boolean returnat de gym, care verifică dacă jocul s-a terminat. Tot de gym este returnat și un Python dictionary, care conține numărul de vieți rămase ale agentului ($ale.lives$).

Dacă în $state$ și în new_state se stochează câte 4 frame-uri fiecare, atunci se obțin 8 milioane de frame-uri. Dar, din moment ce new_state se crează punând cel mai nou frame în $state$ și eliminând cel mai vechi frame, cele 2 au în comun 3 frame-uri. În plus, new_state a unei tranziții este $state$ unei tranziții $i+1$. Astă înseamnă că, este suficient să se stocheze ultimele 1 milion de frame-uri (84*84 pixeli) ca un tensor de forma (1 million, 84, 84) și apoi să tăiem 4 frame-uri din tensor când este nevoie de un $state$ sau new_state .

Pentru 1 milion de frame-uri cu pixeli de 84 x 84, trebuie un tip de dată care poate stoca acești pixeli în memorie. Mediul returnează frame-uri cu valorile pixelilor ca fiind *uint8*, iar valorile pot fi în intervalul [0, 255]. Rețeaua se așteaptă la un input *tf.float32* cu valorile pixelilor între 0 și 1 (care ocupă de patru ori mai mult spațiu decât *unit8*). Pentru a reduce cerințele de memorie, stocăm frame-urile în *unit8* și le împărțim la 255 înainte de a intra în rețea.

În constructorul din clasa *ReplayMemory*, pre-alocăm memorie atât pentru frame-uri, acțiuni, recompense și stările terminale, cât și pentru stările curente și stările noi din minibatch.

În funcția `add_experience` toate componentele de mai sus sunt puse în `self.frames` la indexul `self.current` care este apoi crescut cu 1. Când `self.current` ajunge la maximul dimensiunii lui `replay memory` (un milion), se resetează la zero pentru a suprascrie cele mai vechi frame-uri. Această metodă `_get_state` elimină din `self.frames` 4 frame-uri și le returnează ca `state`.

Pentru a înțelege ce face metoda `_get_valid_indices`, trebuie să se înțeleagă ce este un index invalid. Se stochează toate frame-urile pe care agentul le vede în `self.frames`. Când un joc se termină (`terminal = True`) la indexul `i`, frame-ul la indexul `i` aparține unui episod diferit decât frame-ul `i+1`. Același lucru se poate întâmpla la indexul `self.current`. Se dorește să se evite crearea unui `state` cu frame-uri din două episoade diferite.

În final, trebuie să se asigure că un index nu este mai mic decât numărul de frame-uri stochate împreună pentru a crea un `state` (`self.agent_history_length = 4`), astfel încât un `state` sau `new_state` să poată fi extrase din array.

Funcția `_get_valid_indices` găsește 32 (dimensiunea unui minibatch) de indici valizi. Funcția `get_minibatch` returnează tranzițiile pentru acești indici.

Este important de precizat că, pentru `self.states` și `self.new_states` trebuie să fie transpusă înainte de a se returna. În DQN se așteaptă un input de dimensiune `[None, 84, 84, 4]`, iar `_get_state` returnează un `state` de dimensiune `[4, 84, 84]`.

4 Rețeaua țintă și actualizarea parametrilor

În implementarea aplicației s-au folosit 2 rețele, reprezentate prin funcțiile pentru valoare-acțiunie și țintă valoare-acțiune. Se reține că, înainte de actualizarea parametrilor a rețelei, se extrage un minibatch cu 32 de tranzitii. Pentru simplificare, se va considera o singură tranzitie pentru moment. Aceasta constă într-un `state`, un `action` (care este realizat într-un `state`), `reward`-ul primit, `new_state` și un boolean semnalează dacă un episod este gata.

Când se realizează un pas de gradient descent, rețeaua principală (main network) analizează o stare și estimează $Q_{prediction}$ -values care arată cât de valoiosă este fiecare acțiune obținută. Însă, se dorește urmărirea ecuației lui Bellman pentru Q -values. Așadar, se calculează Q_{target} -values conform ecuației lui Bellman [3.32] (adică cum se doresc Q -values să fie), și se compară estimările cu $Q_{prediction}$ cu Q_{target} . Se consideră funcția de loss quadratică în loc de funcția Huber pentru simplicitate 4.2.

$$L = \frac{1}{2} (Q_{prediction} - Q_{target})^2 \quad (4.2)$$

Se asigură astfel regresia din $Q_{prediction}$ -values curente pentru `state` către Q_{target} -values date de ecuația lui Bellman.

$Q_{prediction}$ este calculat în clasa DQN (`self.q_values`). Acesta depinde de $state$ -ul curent pe care l-am extras din minibatch și de parametrii θ a rețelei care estimează.

Q_{target} value este calculat conform ecuației lui Bellman, adică suma recompenselor imediate r , obținute prin realizarea acțiunii a în starea s (extrase din minibatch) și a valorii maxime Q -value peste toate acțiunile posibile a' în s' (new_state din minibatch). Acest lucru nu se realizează în clasa DQN , ci în funcția `learn`. Valoarea calculată este trimisă către un placeholder numit `self.target_q` din clasa DQN . Acolo, funcția de loss este definită și se realizează pasul de gradient descent.

Motivul pentru care se folosesc 2 rețele este că atât $Q_{prediction}$ cât și Q_{target} folosesc aceeași parametri θ . Dacă o singură rețea este folosită astăzi poate duce la instabilitate în momentul regresiei $Q_{prediction}$ către Q_{target} , deoarece target "se mișcă constant". Pentru a se asigura un target "fix", se introduce o două rețea cu parametri fizici (sau actualizați doar ocazional), care va estima target Q -values.

Astfel, o rețea se folosește pentru a prezice $Q_{prediction}$ -value și cealaltă rețea (fixată) pentru a prezice Q -value. Rețeaua principală este optimizată în timpul etapei de gradient descent, și la fiecare 10 000 pași, parametrii se actualizează și sunt copiați în rețeaua ţintă. Calcularea frecvenței pentru actualizarea parametrilor este măsurată în numărul de acțiuni/frame-uri și nu de numărul de actualizări a parametrilor (care se întâmplă la fiecare 4 frame-uri).

Așa cum s-a demonstrat în secțiunea 4, DQN poate estima Q -values nerealist de mari. Q -values estimate au zgromot. Datorită estimărilor cu zgromot, câteva Q -values pot fi pozitive, altele negative. Operația de max din ecuația lui Bellman va alege întotdeauna valorile pozitive, chiar dacă aceste acțiuni ar putea aduce mai puține recompense decât cele negative. Q -values sunt "biased" către valorile mai mari. Se evită acest lucru prin estimarea $Q - value$ în următoarea stare $Q(s', a')$ cu rețeaua ţintă, se folosește rețeaua principală pentru estimarea acțiunii cele mai bune și se folosește rețeaua ţintă pentru a determina care este cel mai mare Q -value pentru acea acțiune. În acest mod, rețeaua principală va prefera în continuare valorile pozitive, dar datorită zgromotului, rețeaua ţintă va estima un Q -value pozitiv sau negativ într-o acțiune și în medie, Q -values vor fi mai apropiate de 0.

Matematic, motivul supraestimărilor este că aşteptarea unui maxim este mai mare sau egală decât maximul unei aşteptări. Ecuația lui Bellman atunci va fi folosită în modul următor 3.39, cu y^{DDQN} fiind egal cu $Q_{target}(s, a)$.

Ca un exemplu, DQN normal: interoghează rețeaua ţintă care are cel mai mare Q -value. Dacă valorile zgromotului sunt de exemplu $(0.1, -0.1)$ pentru acțiunile cu indexul 0 și 1 respectiv, rețeaua ţintă va răspunde 0.1.

Rețeaua Double DQN: interoghează rețeaua principală despre acțiune cu cel mai mare Q -value. Dacă valorile zgromotului sunt $(0.1, -0.1)$ pentru acțiunile la

index 0 și respectiv 1, rețeaua principală va răspunde cu acțiunea de la index 0 care cel mai mare Q -value. Apoi se întreabă rețeaua ţintă, care are valorile de zgromot diferite, care este Q -value pentru acțiune cu indexul ales (0 în acest exemplu). Se consideră că valorile zgromotului din rețeaua ţintă sunt $(-0.05, 0.3)$, iar rețeaua va răspunde la -0.05 .

Astfel, se rezolvă (parțial) problema supraestimării a Q -values datorită celor două rețele cu zgromot diferit și a bias-ului către valorile pozitive.

Dacă jocul se termină, adică ($terminal = True$) pentru că agentul fie a câștigat, fie a pierdut, nu există nicio stare următoare deci Q_{target} -value este recompensa r .

Mediul pentru învățare este furnizat de către gym. Trebuie ținut cont de versiunea fiecărui mediu. De exemplu, *BreakoutDeterministic* – v3 are șase acțiuni *BreakoutDeterministic* – v4 are un set de minim 4 acțiuni. În plus, fiecare acțiune face ca învățarea să devină mai grea pentru agent, care poate altera scorul în timpul evaluării.

Când se pierde o viață în timpul jocului, se salvează *terminal_life_lost* = *True* în replay memory. Se face acest lucru, deoarece când se pierde o viață nu există nicio "pedeapsă", recompensa va fi 0. Acest lucru va ajuta angetul să evite pierderea de vieți dacă se consideră că la pierderea unei vieți se termină episodul. Dar, nu se dorește resetarea jocului în momentul pierderii primei vieți. Astfel, este nevoie de două stări terminale, *terminal* și *terminal_life_lost*. Clasa *Atari* va îngloba mediul gym, se va ocupa de frame-uri stivuite una peste cealaltă pentru a crea stări, apoi se va ocupa de resetarea mediului când un episod se termină, dar și de verificarea dacă după un pas s-a pierdut o viață.

În timpul evaluării, la începutul fiecărui episod, acțiunea 1 ('Trage') este repetată pentru un număr aleatoriu de pași, între 1 și *no_op_steps* = 10. Asta asigură că agentul va începe dintr-o situație diferită de fiecare dată și nu va învăța o secvență fixă de acțiuni. [3] folosește un număr aleatoriu între 1 și 30 de acțiuni 'NOOP'. Dar, în Breakout, nu se întâmplă nimic dacă nu se efectuează acțiunea de tip 'Trage'. Odată ce bila e în joc, 'Trage' nu va mai face nimic. Prin urmare, se limitează numărul de acțiuni 'Trage' la 10.

Atât Mnih et al. 2013 cât și Mnih et al. 2015 restricționează recompensele, autorii explicând faptul că scorurile diferă semnificativ de la joc la joc, astfel se fixează pentru toate recompensele pozitive 1 și pentru toate recompensele negative -1. Recompenselor 0 vor rămâne neschimbate. Restricționarea recompenselor în această manieră limitează scara derivatelor de eroare și face mai ușoră folosirea aceluiași learning rate pentru mai multe jocuri. În același timp, se poate afecta performanța agentului deoarece nu poate diferenția între recompensele de magnitudine diferită.

În continuare se declară constantele care definesc comportamentul de învățare a agentului și hiper-parametrii. Apoi se crează rețelele și se configurează rezu-

matele tensorboard pentru loss, pentru media recompenselor, pentru evaluare și pentru parametrii rețelei cu scopul observării procesului de învățare.

Se implementează funcția de antrenare [3.7](#):

- Se inițializează replay memory.
- Funcția Q -value a acțiunii este rețeaua DQN implementată.
- Se inițializează funcția ţintă.
- La începutul fiecărui episod trebuie inițializată o secvență. Acest lucru este implementat prin stocarea a câte 4 frame-uri.
- Se selectează o acțiune cu ϵ -greedy.
- Când se realizează o acțiune, mediul returnează o tranziție din replay memory.
- Se returnează un minibatch din replay memory și se realizează un pas de gradient descent.
- În final, se resetează rețeaua ţintă Q la rețeaua principală Q .

5 Ghidul utilizatorului

În această secțiune se vor detalia pașii pentru reproducerea proiectului.

- Primul pas este de a instala Anaconda Individual Edition și Jupyter Notebook.
- În al doilea pas, pentru reproducerea environmentului, se va folosi un fișier numit `environment.yml`. În fișierul `environment.yml` se alfă toate dependințele necesare pentru rularea aplicației, mai exact, este o copie identică a environment-ului pe care s-a antrenat rețeaua. Pentru a reproduce environment-ul, se deschide un terminal anaconda și se introduce comanda `conda env create -f environment.yml`
- În al treilea pas se activează environmentul. Pentru activarea environment-ului se folosește comanda `conda activate myenv`, care .
- În al patrulea pas, se verifică dacă s-a instalat corect environment-ul. Pentru verificare se folosește comanda `conda env list`.

În continuare urmează instrucțiunile necesare pentru rularea corectă a aplicației.

- Pentru antrenarea rețelei, se va seta $TRAIN = True$.
- Se va selecta environment-ul dorit prin modificarea ENV_NAME
- Output-ul va fi generat sub formă de GIF-uri. Funcția `generate_gif` construiește un GIF dintr-o secvență de frame-uri. În timpul antrenării, aceste GIF-uri se vor genera la fiecare evaluare în folder-ul ...\\output. Pentru a genera GIF-uri pentru o rețea antrenată se setează $TRAIN = False$. GIF-urile generate când $TRAIN = False$ vor fi generate în folder-ul ...\\GIF
- Pentru vizualizare în timpul antrenamentului, se folosește TensorBoard.
- Pentru vizualizarea GIF-urilor, se folosește o aplicație realizată în C# cu WPF. Aceasta este constituită din 2 butoane pentru a trece printr-o listă cu GIF-uri. Un buton va fi pentru a merge parcurge GIF-urile în stânga, iar un alt buton pentru a parcuge GIF-urile în dreapta.

Opțional, pentru conectarea la unealta TensorBoard se vor realiza următorii pași: pentru vizualizarea corectă în TensorBoard, mai întâi utilizatorul preia din folder-ul ...\\summaries\\run_1 fișierul events.out.tfevents (fișierul este generat de fiecare dată când se antrenează rețea) și se mută în folder-ul ...\\summaries \\last. De reținut, în folder-ul ...\\last poate exista un singur fișier events.out.tfevents. Apoi se deschide un terminal anaconda, se navighează la folder-ul ...\\summaries și se introduce comanda **tensorboard --logdir="last"**. Se preia link-ul generat și se introduce într-un browser web. Utilizatorul va putea alege între Scalars, Distributions, Histograms și Time Series pentru vizualizare.

Pentru a verifica dacă antrenarea progresează într-un mod corespunzător, se pot urmări graficele pentru loss și pentru recompense din TensorBoard.



Figure 4.2: Vizualizarea loss-ului în Tensorboard.

În graficul 4.2 se poate observa loss-ul rețelei în timpul antrenării. Se dorește ca *Performance/loss* să crească cât mai mult, adică loss-ul să scadă cât mai mult până când se ajunge la valori apropiate de 0.

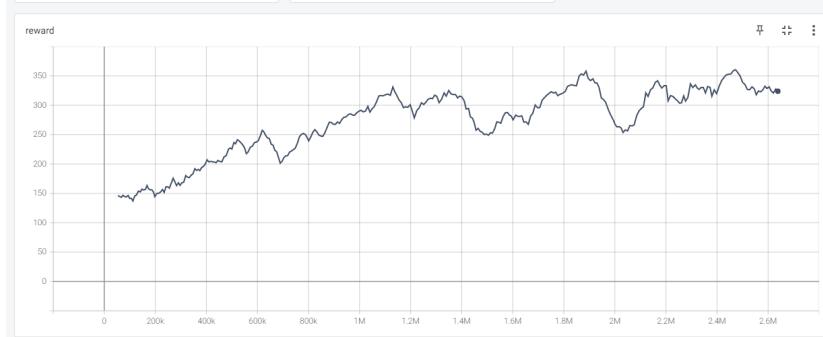


Figure 4.3: Vizualizarea recompenselor în Tensorboard.

Cum se poate observa în graficul 4.3, recompensele cresc constant. Acest lucru este de preferat până când rețeaua reușește să obțină o valoare medie a recompenselor care să fie apropiate de un maxim local.

Capitolul 5

Concluzii și perspective de dezvoltare

În acest capitol se vor prezenta câteva îmbunătățiri posibile pentru implementarea DDQN și concluziile și recomandările pentru următoarea implementare.

1 Îmbunătățiri posibile

- Implementarea cu un alt framework, de exemplu PyTorch sau Apache MXNet.
- Modificarea arhitecturii rețelei, de exemplu: introducerea de alte straturi, găsirea unei alte funcții de loss, schimbarea mediului (o alternativă pentru Gym).
- Antrenarea pe mai multe jocuri. În momentul de față rețeaua este antrenată pentru 3 jocuri: Pong, Breakout și Space Invaders
- Găsirea unui learning rate optim pentru fiecare dintre următoarele jocuri antrenate.
- Antrenarea cu alt optimizator, de exemplu RMSprop.
- Găsirea unei alternative pentru restricționarea recompenselor.

2 Concluzii și recomandări pentru următoarea implementare

În general, rețelele de neurale pot fi destul de dificil de implementat. DQN nu este diferit, atenția la detalii fiind absolut necesară. De la hiper-parametrii la setarea environment-ului, la maximizarea recompensei și rezolvarea problemei

de exploding gradients, DQN rămâne o arhitectură greu de implementat. Trebuie ținut cont mereu de mediul folosit, de arhitectura rețelei, de funcțiile de cost și avantaj, de recompense, de stările și acțiunile agentului. Însă, odată implementată aceasta obține rezultate remarcabile și este o oportunitate bună pentru învățarea rețelelor neurale și a reinforcement learning-ului, în mod special când vine vorba de debugging și de îmbunătățirea unui algoritm deja existent. Arhitectura poate fi extinsă pentru a fi folosită mai multe domenii, nu doar pentru învățarea jocurilor video. Oportunitățile sunt vaste, și pe zi ce trece apar noi probleme care pot fi rezolvate cu ajutorul rețelelor neurale în diverse domenii, precum data mining, chimia quantică, reconstruirea 3D, finanțe, filtrarea de e-mail spam, identificarea fețelor etc. Toate aceste domenii asigură un viitor în care rețele neurale vor fi un punct cheie în viațile oamenilor.

Bibliografie

- [1] [Online; accessed 2-June-2021]. 2021. URL: https://en.wikipedia.org/wiki/File:Complex_systems_organizational_map.jpg.
- [2] Harris C.R. et al. "Array programming with NumPy". In: *Nature* 585.7825 (2020), pp. 357–362. DOI: [10.1038/s41586-020-2649-2](https://doi.org/10.1038/s41586-020-2649-2). URL: <https://doi.org/10.1038/s41586-020-2649-2>.
- [3] Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 00280836. URL: <http://dx.doi.org/10.1038/nature14236>.
- [4] Andre Violante. *Simple Reinforcement Learning: Temporal Difference Learning*. [Online; accessed 6-June-2021]. 2018. URL: <https://medium.com/@violante.andre/simple-reinforcement-learning-temporal-difference-learning-e883ea0d65b0>.
- [5] TensorFlow Developers. *Building Standard TensorFlow ModelServer*. [Online; accessed 4-June-2021]. 2021. URL: https://www.tensorflow.org/tfx/serving/serving_advanced.
- [6] TensorFlow Developers. *TensorFlow*. Version v2.5.0. Specific TensorFlow versions can be found in the "Versions" list on the right side of this page. See the full list of authors "<https://github.com/tensorflow/tensorflow/graphs/contributors>" on GitHub. May 2021. DOI: [10.5281/zenodo.4758419](https://doi.org/10.5281/zenodo.4758419). URL: <https://doi.org/10.5281/zenodo.4758419>.
- [7] Satya Ganesh. *What's The Role Of Weights And Bias In a Neural Network?* [Online; accessed 4-June-2021]. 2020. URL: <https://towardsdatascience.com/whats-the-role-of-weights-and-bias-in-a-neural-network-4cf7e9888a0f>.
- [8] Great Learning Team. *Simplified Reinforcement Learning: Q Learning*. [Online; accessed 8-June-2021]. 2020. URL: <https://www.mygreatlearning.com/blog/simplified-reinforcement-learning-q-learning/>.

- [9] Hado Hasselt. “Double Q-learning”. In: *Advances in Neural Information Processing Systems*. Ed. by J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta. Vol. 23. Curran Associates, Inc., 2010. URL: <https://proceedings.neurips.cc/paper/2010/file/091d584fcfed301b442654dd8c23b3fc9-Paper.pdf>.
- [10] Hado van Hasselt, Arthur Guez, and David Silver. *Deep Reinforcement Learning with Double Q-learning*. 2015. arXiv: [1509.06461 \[cs.LG\]](https://arxiv.org/abs/1509.06461).
- [11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. 2015. arXiv: [1502.01852 \[cs.CV\]](https://arxiv.org/abs/1502.01852).
- [12] JetBrains. *2020 Developer Survey*. [Online; accessed 31-May-2021]. 2020. URL: <https://www.jetbrains.com/lp/python-developers-survey-2020/>.
- [13] Arthur Juliani. *Simple Reinforcement Learning with Tensorflow Part 4: Deep Q-Networks and Beyond*. [Online; accessed 8-June-2021]. URL: <https://awjuliani.medium.com/simple-reinforcement-learning-with-tensorflow-part-4-deep-q-networks-and-beyond-8438a3e2b8df>.
- [14] Mariana Berga, Pedro Coelho. *PYTORCH VS TENSORFLOW: COMPARING DEEP LEARNING FRAMEWORKS*. [Online; accessed 2-June-2021]. 2021. URL: <https://www.imaginarycloud.com/blog/pytorch-vs-tensorflow/>.
- [15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: [1312.5602 \[cs.LG\]](https://arxiv.org/abs/1312.5602).
- [16] Keiron O’Shea and Ryan Nash. “An Introduction to Convolutional Neural Networks”. In: *ArXiv e-prints* (Nov. 2015).
- [17] J. Peters. “Policy gradient methods”. In: *Scholarpedia* 5.11 (2010). revision #137199, p. 3698. DOI: [10.4249/scholarpedia.3698](https://doi.org/10.4249/scholarpedia.3698).
- [18] Lucian M. Sasu. *Curs de Inteligență artificială*. [Online; accessed 6-June-2021]. 2021. URL: <https://github.com/lmsasu/cursuri/blob/master/InteligentaArtificiala/curs/InteligentaArtificiala.pdf%7D>.
- [19] Stack Overflow. *2020 Developer Survey*. [Online; accessed 31-May-2021]. 2020. URL: <https://insights.stackoverflow.com/survey/2020#overview>.
- [20] Sunpark. *It’s Deep Learning Times: A New Frontier of Data*. [Online; accessed 4-June-2021]. 2019. URL: <https://towardsdatascience.com/its-deep-learning-times-a-new-frontier-of-data-a1e9ef9fe9a8>.

- [21] TensorFlow Developers. *Case Studies*. [Online; accessed 2-June-2021]. 2021. URL: <https://www.tensorflow.org/about/case-studies>.
- [22] TensorFlow Developers. *TensorBoard: TensorFlow's visualization toolkit*. [Online; accessed 2-June-2021]. 2021. URL: <https://www.tensorflow.org/tensorboard>.
- [23] The AlphaStar team. *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II — DeepMind*. [Online; accessed 24-May-2021]. 2019. URL: <https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii>.
- [24] Sebastian Thrun and A. Schwartz. “Issues in Using Function Approximation for Reinforcement Learning”. In: *Proceedings of the 1993 Connectionist Models Summer School*. Ed. by M. Mozer P. Smolensky D. Touretzky J. Elman and A. Weigend. Erlbaum Associates, 1993.
- [25] TIOBE Index. *Python back at second position*. [Online; accessed 24-May-2021]. 2021. URL: <https://www.tiobe.com/tiobe-index>.
- [26] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. *Dueling Network Architectures for Deep Reinforcement Learning*. 2016. arXiv: [1511.06581 \[cs.LG\]](https://arxiv.org/abs/1511.06581).
- [27] Wikipedia contributors. *Activation function — Wikipedia, The Free Encyclopedia*. [Online; accessed 11-June-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Activation_function&oldid=1026815756.
- [28] Wikipedia contributors. *Autoencoder — Wikipedia, The Free Encyclopedia*. [Online; accessed 8-June-2021]. 2021. URL: <https://en.wikipedia.org/w/index.php?title=Autoencoder&oldid=1022461793>.
- [29] Wikipedia contributors. *Convolutional neural network — Wikipedia, The Free Encyclopedia*. [Online; accessed 8-June-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Convolutional_neural_network&oldid=1026397751.
- [30] Wikipedia contributors. *Dataflow programming — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Dataflow_programming&oldid=1019083582. [Online; accessed 2-June-2021]. 2021.
- [31] Wikipedia contributors. *Deep learning — Wikipedia, The Free Encyclopedia*. [Online; accessed 24-May-2021]. 2021. URL: https://en.wikipedia.org/wiki/Deep_learning#/media/File:AI-ML-DL.svg.

- [32] Wikipedia contributors. *Differentiable programming* — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Differentiable_programming&oldid=1023129317. [Online; accessed 2-June-2021]. 2021.
- [33] Wikipedia contributors. *Dynamic programming* — Wikipedia, The Free Encyclopedia. [Online; accessed 6-June-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Dynamic_programming&oldid=1022502054.
- [34] Wikipedia contributors. *Hedonism* — Wikipedia, The Free Encyclopedia. [Online; accessed 30-May-2021]. 2021. URL: <https://en.wikipedia.org/w/index.php?title=Hedonism&oldid=1021290227>.
- [35] Wikipedia contributors. *Long short-term memory* — Wikipedia, The Free Encyclopedia. [Online; accessed 8-June-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Long_short-term_memory&oldid=1027406795.
- [36] Wikipedia contributors. *Markov decision process* — Wikipedia, The Free Encyclopedia. [Online; accessed 6-June-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Markov_decision_process&oldid=1019172798.
- [37] Wikipedia contributors. *Monte Carlo method* — Wikipedia, The Free Encyclopedia. [Online; accessed 6-June-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Monte_Carlo_method&oldid=1025566764.
- [38] Wikipedia contributors. *P versus NP problem* — Wikipedia, The Free Encyclopedia. [Online; accessed 22-May-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=P_versus_NP_problem&oldid=1021789526.
- [39] Wikipedia contributors. *Principal component analysis* — Wikipedia, The Free Encyclopedia. [Online; accessed 3-June-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Principal_component_analysis&oldid=1025280869.
- [40] Wikipedia contributors. *Stanford marshmallow experiment* — Wikipedia, The Free Encyclopedia. [Online; accessed 24-May-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Stanford_marshmallow_experiment&oldid=1023073905.
- [41] Wikipedia contributors. *Stochastic* — Wikipedia, The Free Encyclopedia. [Online; accessed 7-June-2021]. 2021. URL: <https://en.wikipedia.org/w/index.php?title=Stochastic&oldid=1022310858>.

- [42] Wikipedia contributors. *T-distributed stochastic neighbor embedding* — Wikipedia, The Free Encyclopedia. [Online; accessed 3-June-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=T-distributed_stochastic_neighbor_embedding&oldid=1020310146.