

BABEŞ BOLYAI UNIVERSITY, CLUJ NAPOCA, ROMÂNIA  
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

# **Observarea și imitarea structurată a comportamentului folosind acțiuni primitive**

– Diploma thesis –

**Coordonator Științific**  
Lect.Dr. Gabriel Ioan Mircea

**Author**  
Crișan Adrian-Cătălin, Informatică - Română, 232

## Abstract

This paper aims to solve the problem of creating primitive based control structures for agents that imitate the observed behaviour. This problem is important both in the field of robotics and in commercial applications such as video games, where an algorithm capable of one-shot imitation would allow robots to be taught new behaviours easily and game AI agents could become more realistic. Using controls structures and primitives instead of probabilistic models or neural networks would also have the advantage of being more easily edited without retraining. To tackle this problem we employed an unsupervised machine learning method, Reinforcement Learning in conjunction with the mixture of experts architecture. We created special environments for each expert/agent, which are used in a hierarchical fashion to construct a simplified behaviour tree, in which no branches are present. Due to the amount of data needed to train the experts we procedurally generated data uniformly distributed over the value interval of real world data, which we then passed through a custom pre-processing function in order to solve problems related with training performance. We applied this algorithm to motor behaviour imitation and obtained moderate results that nonetheless stand as a proof of concept for the basic idea behind the proposed approach.

# Contents

<b>1</b>	<b>Introducere</b>	<b>1</b>
1.1	Structura lucrării și contribuții originale . . . . .	2
<b>2</b>	<b>Problema Științifică</b>	<b>3</b>
2.1	Relevanța științifică . . . . .	4
<b>3</b>	<b>Direcții de cercetare existente</b>	<b>6</b>
3.1	Acțiuni Primitive . . . . .	6
3.2	Arhitectura Mixture of Experts . . . . .	7
3.3	Învățare Imitativă . . . . .	9
3.3.1	Auto-imitație . . . . .	9
3.3.2	Algoritmul Neural Gas . . . . .	10
3.3.3	Comportament stratificat . . . . .	11
3.3.4	Modelarea comportamentului folosind structuri simple . . . . .	13
3.3.5	Modele probabilistice în imitație . . . . .	14
<b>4</b>	<b>Metoda propusa</b>	<b>15</b>
4.1	Pași preliminari . . . . .	16
4.1.1	Observare, stări și primitive . . . . .	16
4.1.2	Reprezentarea primitivelor . . . . .	16
4.1.3	Construirea comportamentului . . . . .	17
4.2	Setul de date . . . . .	17
4.3	Structura Expertilor . . . . .	18
4.3.1	Datele de intrare și ieșire . . . . .	18
4.4	Agenții de Reinforcement Learning . . . . .	20
4.4.1	Agentul de selectarea a primitivelor . . . . .	21
4.4.1.1	Experimentarea cu funcții de reward . . . . .	21
4.4.1.2	Alegerea Hiper-parametrilor algoritmului . . . . .	23
4.4.1.3	Optimizatori & Learning Rates . . . . .	24
4.4.1.4	Topologia expertului . . . . .	26
4.4.1.5	Pre-procesarea datelor . . . . .	27
4.4.2	Agentul de selecție a poziției de editare . . . . .	30
4.4.2.1	Topologia expertului . . . . .	31
4.4.3	Agentul de control al iterăției . . . . .	32
4.4.3.1	Topologia expertului . . . . .	33
4.5	API . . . . .	34
4.5.1	Endpoints . . . . .	35

4.5.2	Detalii de implementare . . . . .	37
4.5.3	Testarea API-ului . . . . .	37
<b>5</b>	<b>Aplicatie</b>	<b>39</b>
5.1	Arhitectura . . . . .	39
5.1.1	Cazuri de utilizare . . . . .	40
5.1.2	Design Patterns . . . . .	43
5.1.2.1	Observer . . . . .	44
5.1.2.2	Singleton . . . . .	44
5.1.2.3	Memento/Proxy . . . . .	44
5.1.2.4	Adapter . . . . .	45
5.2	Implementarea . . . . .	45
5.2.1	Technology Stack . . . . .	46
5.2.2	Testare . . . . .	47
5.2.3	Detalii de implementare . . . . .	47
5.2.4	Persistenta datelor . . . . .	48
5.2.5	UX & Manualul de utilizare . . . . .	48
<b>6</b>	<b>Concluzii și direcții viitoare</b>	<b>51</b>

# List of Tables

4.1	Parametrii de stare ce vor fi observați în procesul de imitație. (*nu sunt folosite)	19
4.2	Primitivele ce vor fi folosite și codificările acestora.	19
4.3	Configurații ale parametrilor formulei de scădere exponentiale a learning rate-ului care au fost folosite în experimente	25
4.4	Configurațiile experimentelor pentru ajustarea hiper-parametrilor agentului, *-valorile de baza pentru restul experimentelor	25
4.5	Rewardurile obținute în environment-ul creat pentru antrenarea expertului ce decide continuarea procesului de editare	33

# List of Figures

2.1	Un state machine simplificat pentru un NPC . . . . .	5
2.2	Prototip <i>Finite State Machine</i> folosit în jocuri video [13] . . . . .	5
2.3	<i>Behaviour Tree</i> creat în Unreal Engine [23] . . . . .	5
3.1	Figura prezintă siluetele obținute prin adunarea mai multor exemple și <i>motion cloud</i> -ul obținut după folosirea de <i>image subtraction</i> pe exemple, în plus fiind reprezentate și caracteristicile ce definesc primitivele: axele elipsei ce reprezinta <i>motion cloud</i> -ul și distanța și direcția de la punctul de referință la acesta [12]. . . . .	7
3.2	Arhitectura <i>Hierarchical Mixture of Experts</i> [3]. . . . .	8
3.3	O serie de rețele neuronale interconectate folosite pentru a reproduce diferite aspecte comportamentului observat [7]. . . . .	9
3.4	Rețeaua <i>neural gas</i> în diferite stagiile de antrenare folosind metoda <i>growing neural gas</i> [4]. . . . .	11
3.5	Mișcările efectuate de un jucător uman în reprezentarea topologică (stânga) și <i>potential field</i> -ul rezultat (dreapta) [22]. . . . .	12
3.6	Un exemplu concret de model Markovian ascuns cu 3 stări ascunse și 3 observații reprezentând emoții [19]. . . . .	14
4.1	Pașii algoritmului de creare a agentului AI . . . . .	15
4.2	Layerele de input și output pentru rețeaua neuronala a expertului ce decide continuarea procesului de editare . . . . .	19
4.3	Layerele de input și output pentru rețeaua neuronala a expertului ce determină poziția editării . . . . .	20
4.4	Layerele de input și output pentru rețeaua neuronala a expertului care efectuează schimbarea primitivei . . . . .	20
4.5	Am încercat modificarea pantei funcției de reward pentru a premia mai mult stările mai aproape de optim . . . . .	22
4.6	O alta variantă a funcției de reward, care premiază cu 0 soluțiile care nu obțin între 0.75 și 1 în faza de normalizare . . . . .	22
4.7	Reward-ul obținut folosind learning rate-ul $10^{-3}$ , primul episod fiind "warm-up" cu acțiuni selectate aleator . . . . .	23
4.8	Reward-ul obținut folosind learning rate-ul scăzut la $10^{-7}$ și politica Epsilon Greedy . . . . .	23
4.9	Reward-ul obținut de expert folosind politica Annealed Epsilon Greedy . . . . .	24
4.10	Learning rate-urile pentru fiecare experiment descris . . . . .	25
4.11	Topologia celei de-a treia rețele expert, pentru primul experiment . . . . .	26
4.12	Topologia celei de-a treia rețele expert, folosita în al doilea experiment . . . . .	26

4.13 Reward-ul obținut folosind o rețea de mici dimensiuni . . . . .	26
4.14 Reward-ul obținut folosind o rețea de dimensiune medie . . . . .	26
4.15 Reward-ul obținut folosind rețeaua de dimensiune mica, antrenată pe mai multe episoade . . . . .	27
4.16 Reward-ul obținut folosind rețeaua de dimensiune medie, antrenată pe mai multe episoade . . . . .	27
4.17 Distribuția grafică a diferitelor clase, pe axele <b>x</b> și <b>y</b> sunt reprezentate starea curentă respectivă starea țintă, și pe axa verticală, <b>z</b> acțiunea optimă. Graficul prezintă doar valorile din intervalul [0,10], nu întregul interval [0,360] pentru axele <b>x</b> și <b>y</b> . . . . .	28
4.18 Distribuția grafică a diferitelor clase, pe întregul interval normalizat . . . . .	28
4.19 Distribuția grafică a diferitelor clase după aplicarea funcției de preprocesare, reprezentată în spațiul normalizat . . . . .	29
4.20 Reward-ul obținut de modelul foarte simplu, folosind noua funcție de preprocesare	30
4.21 Reward-ul obținut de modelul 4.11, folosind noua funcție de preprocesare . . . .	30
4.22 Arhitectura rețelei care decide poziția de editare pentru iteratărea curentă . . . . .	31
4.23 Reward-ul obținut de agent folosind rețeaua cu 3 layere ascunse . . . . .	32
4.24 Reward-ul obținut de agent folosind rețeaua cu 6 layere ascunse . . . . .	32
4.25 Topologia rețelei primului expert, responsabilă de controlul iterărilor . . . . .	33
4.26 Reward-ul per episod obținut de expertul ce controlează iterarea . . . . .	34
 5.1 Diagrama de componente a aplicației . . . . .	40
5.2 Cazurile de utilizare identificate pentru aplicație . . . . .	40
5.3 Diagrama de secventa pentru crearea unei noi animații . . . . .	41
5.4 Diagrama de secventa pentru editarea unei animații existente . . . . .	42
5.5 Diagrama de secventa pentru imitarea unei animații observate . . . . .	43
5.6 Diagrama claselor pentru extensia dezvoltată . . . . .	45
5.7 Secțiunea de imitație a ferestrei . . . . .	49
5.8 Fereastra modală folosită pentru completarea datelor cererii de imitație . . . . .	49
5.9 Scene View-ul și fereastra editorului nostru, în secțiunea de editarea a animațiilor	49

# List of Algorithms

1	Agent Reward algorithm . . . . .	22
---	----------------------------------	----

# Chapter 1

## Introducere

Metodele actuale de învățare imitativa produc algoritmi sau agenți capabili să imite comportamente observate. O slăbiciune a acestor metode este inabilitatea de a edita cu ușurință comportamentul agentului fără a trece din nou prin procesul de antrenare. Totodată construirea unei structuri de control a agenților de către dezvoltatori este dificila, agentul putând să se afle în situații pe care dezvoltatorii nu le-au prevazut. Propunem o soluție ce îmbina învățarea imitativa cu folosirea unor structuri de control ușor de editat de dezvoltatori fără a necesita reantrenarea experților.

Problema pe care o abordam este învățarea folosirii unor primitive de acțiune abstractizate pentru a construi o structura de control imitativa.

Utilitatea rezolvării acestei probleme este în obținerea unui algoritm ce folosește unul sau mai mulți experți capabili să folosească o colecție de acțiuni primitive pentru a imita un comportament observat o singura dată. Exemple de folosire a unui astfel de algoritm sunt imitarea comportamentului unui jucător uman în jocuri video sau instruirea unui robot de către un expert uman.

Abordarea noastră este de a împărți problema în sub pași, și de a antrena o ierarhie de experți fiecare antrenat să rezolve unul din pași pentru a obține un algoritm ce poate construi o structură de control formată din acțiuni primitive pe baza unor informații de observare a unui comportament.

Rezultatele obținute folosind metoda prezentată au fost modeste, metoda propusă fiind una nouă, ce îmbină diferite aspecte ale metodelor explorate în literatura științifică.

## 1.1 Structura lucrării și contribuții originale

Principala contribuție originală a lucrării este propunerea și crearea unui algoritm capabil să înțeleagă folosirea unor primitive abstrakte pentru a imita un comportament motor. Algoritmul creat nu obține rezultate comparabile cu metodele explorate în literatură, dar funcționează ca un *proof of concept* încurajând explorarea în viitor a metodei pentru a îmbunătăți rezultatele.

O contribuție secundară este crearea unei extensii a game engine-ului Unity care permite crearea unor animații și capturarea datelor din timpul execuției acestora pentru a fi folosite ca date de intrare algoritmului. De asemenea aplicația poate reda execuția structurii de control obținute prin imitație.

Capitolul 2 prezintă problema abordată, relevanta științifică a problemei și utilizări practice, mai ales în domeniul jocurilor video a unui algoritm ce rezolvă cu succes problema imitării structurate.

Capitolul 3 prezintă direcțiile existente în cercetarea învățării imitative, folosirii primitivelor în observarea și compunerea comportamentului cat și metode probabilistice. Acest capitol prezintă și algoritmi și arhitecturi folosite pentru imitarea comportamentului. Fiecare dintre aceste metode este prezentată și în raport cu problema propusă pentru a decide utilitatea lor în rezolvarea problemei noastre.

In capitolul 4 descriem algoritmul propus pentru rezolvarea problemei imitație structurate a comportamentelor motorii, folosirea arhitecturii *mixture of experts* și metodele de inteligență artificială folosite. Capitolul prezintă aplicarea metodei Reinforcement Learning pentru a antrena experții necesari și dificultățile întâlnite în acest proces. La sfârșitul acestui capitol descriem API-ul prin care oferim funcționalitate obținute de algoritmul nostru.

Arhitectura și implementarea aplicației practice sunt descrise în capitolul 5. În acest capitol prezentăm folosirea game engine-ului Unity pentru care am dezvoltat o extensie care oferă facilități de creare și imitarea a animațiilor.

# Chapter 2

## Problema Științifica

Modelarea și modificarea comportamentului unui agent inteligent este o problema întâmpinată atât în cercetarea științifica cat și în aplicații comerciale precum jocurile video. Crearea unui comportament cu un nivel ridicat de complexitate aduce unele dificultăți.

Pe de-o parte lipsa unei structuri ușor de înțeles comportamentului creat de componenta inteligentă ce "conduce" robotul sau agentul virtual creează dificultăți în schimbarea respectivului comportament. Metode foarte eficace în modelarea de comportament precum rețelele neuronale artificiale sau modelele probabilistice sunt greu sau imposibil de înțeles de către om, astfel comportamentul este "criptat" și foarte dificil sau imposibil de schimbat în cazul în care unele aspecte ale comportamentului învățat este nedorit sau chiar periculos.

Pe de alta parte folosirea unor structuri mai simple cum ar fi *finite state machine*-ul sau *behaviour tree*-ul îngreunează creerea comportamentului complex pe care dorim sa îl exprime agentul deoarece creșterea complexității aduce cu ea și creșterea în mărime a structurii de control. Pentru a crea o structură suficient de complexă programatorul ar trebui să știe *a priori* toate situațiile în care se va afla agentul pentru a programa acțiunile adecvate, sarcina ce este foarte dificila. Ca urmare, aceste 2 metode, deși populare în industria jocurilor video înzestrează NPC-ul (un caracter ce nu este controlat de un jucător) doar cu un comportament rudimentar, care este, foarte des, ușor exploarat de jucători (pentru exemple de *finite state machine* sau *behaviour tree* folosite în jocuri vezi figurile 2.1, 2.2, 2.3).

Prin urmare, problema pe care o abordam în aceasta lucrare este crearea unui algoritm ce permite învățarea prin imitație a comportamentului și exprimarea acestuia într-un format ușor de înțeles de om și care facilitează modificarea ulterioara. Mai exact, propunem folosirea unor

componente primitive pe care le vom compune și înlănțui pentru construirea comportamentelor complexe.

## 2.1 Relevantă științifică

Relevanta științifica a abordării constă în răspunderea la următoarea întrebare: **Este posibila crearea unui algoritm capabil sa folosească primitive pentru a imita și structura o colecție de comportamente?** În plus datorita abstractizării primitivelor speram ca, modificând primitivele pe care le are la dispoziție și starea/stările pe care le observa algoritmul, acesta ar putea fi specializat pentru diferite categorii de comportament. Un set de primitive ce specifică acțiuni motorii ar putea fi folosit pentru generarea comportamentelor motorii ale unui robot, în timp ce un set de primitive ce reprezintă acțiuni abstracte, de nivel înalt, și stări adecvate, ar putea fi folosite pentru învățarea de comportamente tactice sau strategice ale unui NPC în jocuri video.

Utilitatea unui astfel de algoritm intelligent se găsește în cazurile în care, după antrenare, comportamentul unui agent este apropiat de nevoile specifice, dar există aspecte negative pe care le dorim înălțurate. În cazul altor metode, precum rețelele neuronale, este imposibilă sau extrem de dificilă alterarea unui singur aspect al comportamentului unui agent, cu precădere dacă se impune conservarea aspectelor dorite ale comportamentului agentului. În plus, o funcționalitate de editare a structurii comportamentului care nu necesită cunoștințe în domeniul informatici, putând fi folosita cu ușurință, necesitând familiarizarea doar cu formatul folosit în structurarea comportamentului și componentele primitive din care este alcătuit.

O alta circumstanță care reflectă beneficiile algoritmului se găsește în aplicații comerciale precum jocuri pe calculator, unde ar putea facilita specificarea comportamentului NPC-urilor (caracterice care nu sunt controlate de jucător.) Metodele cele mai populare în industrie sunt *behaviour tree*-ul și *finite state machine*-ul, care prezintă neajunsuri, în special dacă un comportament complex este necesar, caz în care programatorii ar trebui să prezică, pentru orice situație posibilă, comportamentul acestor actori. Un algoritm capabil să observe comportamentul jucătorilor umani și să reprezinte în mod structurat acest comportament ar îmbunătăți capacitatea agentului să acționeze în situații nefamiliare, pentru care nu a fost pregătit și, totodată, ar putea fi folosit în ordine inversă, asemenei exemplului din paragraful precedent,

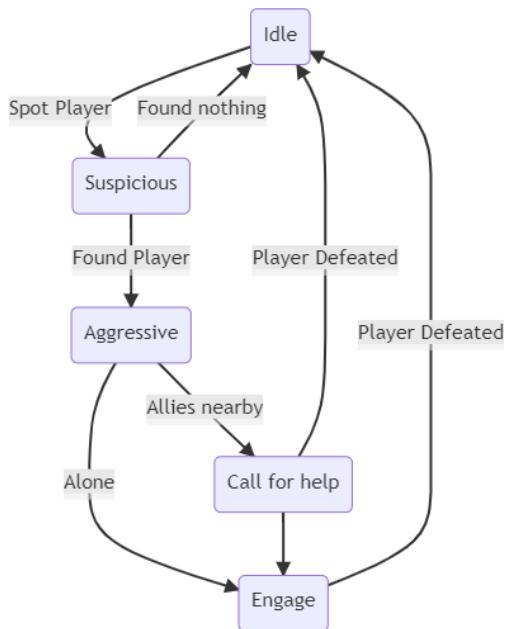


Figure 2.1: Un state machine simplificat pentru un NPC

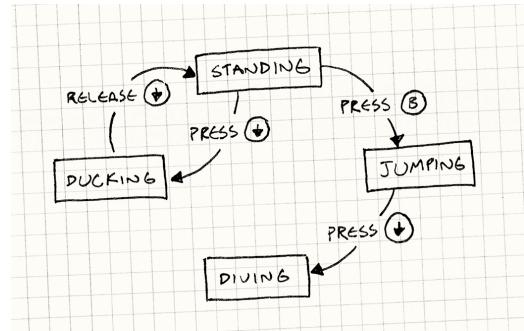


Figure 2.2: Prototip Finite State Machine folosit în jocuri video [13]

fiind antrenat în prima fază folosind funcția de observare și imitație, iar apoi comportamentul rezultat ar putea fi rafinat de programatori, fie din motive de corectitudine, fie din considerente stilistice.

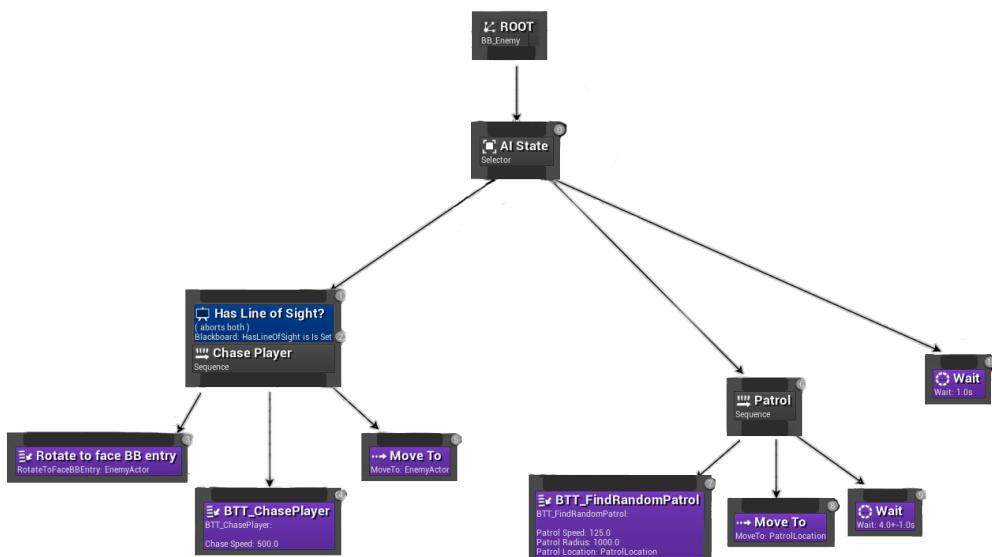


Figure 2.3: *Behaviour Tree* creat în Unreal Engine [23]

# Chapter 3

## Directii de cercetare existente

Problema definita mai sus poate fi descompusa in doua domenii largi, invatarea prin imitatie si folosirea primitivelor pentru construirea comportamentului. In aceasta sectiune vom analiza directii existente de cercetare in aceste doua domenii, acoperind si 2 modele/architecturi relevante: arhitectura **mixture of experts** (MOE) si **Neural Gas**, acestea fiind folosite si in studiul imitatiei comportamentului. In plus vom prezenta si tehnici de modificare a topologiilor retelelor neuronale.

### 3.1 Actiuni Primitive

Folosirea si compunerea de actiuni primitive pentru a obtine comportamente complexe are origini in biologie, miscarile corpului fiind de asemenea compuneri ale unor blocuri de miscare [5]. Idea de a folosi actiuni primitive pentru a compune comportamentul a fost explorata si in robotica, abstractizand primitivele de la activari musculara la actiuni de nivel mai inalt, de exemplu rotirea unei articulatii [15].

Folosirea primitivelor a fost de asemenea abordata de Moeslund *et al.* Acestia au incercat folosirea primitivelor de miscare pentru a identifica secente de gesturi umane [12]. Metoda lor presupune folosirea de *image subtraction* pentru definirea primitivelor de miscare (vezi figura 3.1), si codificarea lor in caractere, astfel o secenta de primitive e codificata ca un sir de caractere asupra caruia au aplicat *edit distance* pentru a identifica secenta. Deoarece *edit distance*-ul favorizeaza secente de caractere mai scurte, a fost modificat acest algoritm, folosind metoda descisa si in Phil *et al.* [2], care considera mai "scumpa" operatia de inlocuire, prin impartirea

distanței de editare la lungimea șirului obținut. De asemenea au ales să folosească doar unele părți din secvența de primitive în procesul de identificare. De interes în metoda prezentată este în mod special codificarea simplă a primitiveelor în caractere, care ușurează procesarea și abstractizează primitiva, o astfel de abstractizare ar putea fi utilă pentru generalizarea părții de identificarea a secvenței de primitive la primitive de orice fel, nu doar de mișcare. Pe de alta parte un neajuns al metodei este limitarea algoritmului la a identifica o singura primitiva la un moment dat, astfel observarea a 2 acțiuni primitive simultane ar fi imposibila.

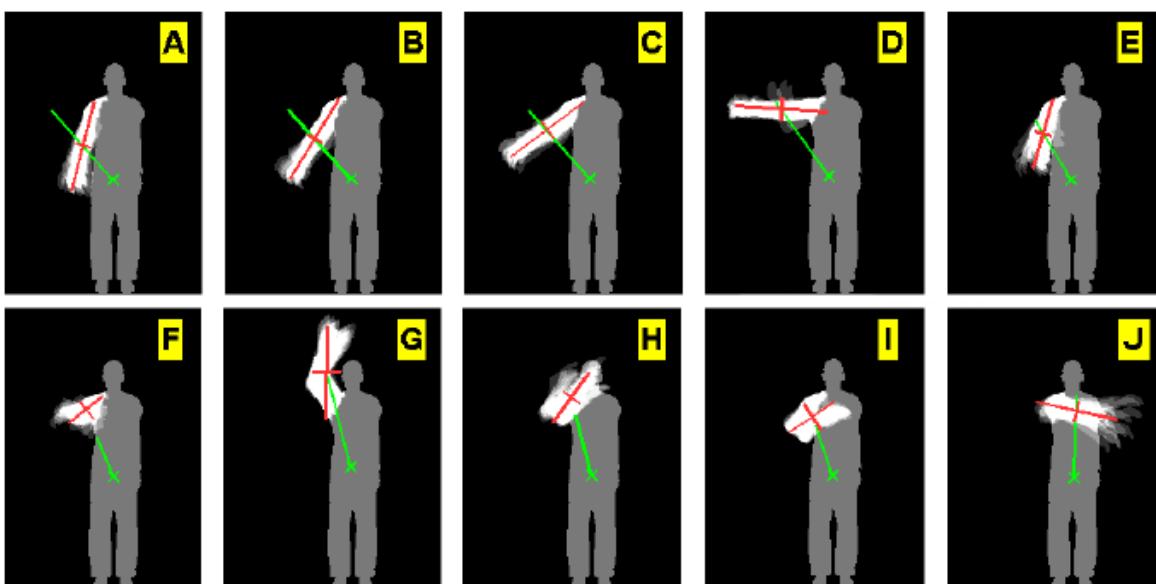


Figure 3.1: Figura prezintă siluetele obținute prin adunarea mai multor exemple și *motion cloud*-ul obținut după folosirea de *image subtraction* pe exemple, în plus fiind reprezentate și caracteristicile ce definesc primitivele: axele elipsei ce reprezinta *motion cloud*-ul și distanța și direcția de la punctul de referință la acesta [12].

## 3.2 Arhitectura Mixture of Experts

Arhitectura Mixture of Experts se bazează pe înlocuirea unei singure rețele mari cu o mulțime de rețele mai mici, astfel realizând o descompunere a funcției complexe pe care o aproximează în sub-funcții, pe care aceste rețele modulare le pot învăță mai rapid comparativ cu viteza de învățarea a funcției complexe de către o rețea monolitică. Arhitectura *modular mixture of experts*, compusă din rețele modulare și bazată pe învățare competitivă, împarte funcția pe intervalul de definire a acesteia și fiecare expert rezolvă întreaga problema pe intervalul alocat, folosirea expertilor fiind ghidată de *gating network*, o rețea ce învăță intervalele în care fiecare

expert este cel mai util [8]. În schimb, arhitectura *hierarchical mixture of experts* împarte funcția în pași logici de rezolvarea a acesteia, fiecare expert învățând să aproximeze o sub-funcție. Astfel este creata o ierarhie de experți, fiecare strat al arhitecturii reprezentând un pas în rezolvarea problemei (vezi figura 3.2). Aceasta arhitectura este excelenta în situații în care sarcina poate fi împărțită în mod natural în sub-sarcini [9].

Arhitecturile mixture of experts au fost utilizate și în domenii în afara învățării imitative, fiind folosite pentru a rezolva *speech recognition*, ierarhia expertilor putând fi statică sau creata folosind un algoritm de *tree-growing* [3].

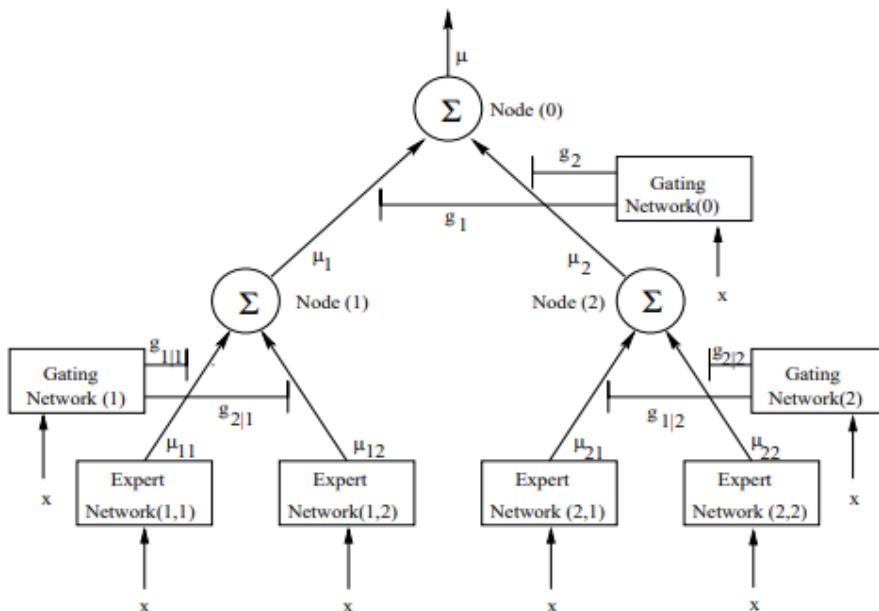


Figure 3.2: Arhitectura *Hierarchical Mixture of Experts* [3].

Pentru simplitate ne vom referi la *hierarchical mixture of experts* doar ca *mixture of experts*, aceasta varianta fiind folosita în lucrarea următoare și care ne interesează în restul lucrării. Gorman și Humphrys folosesc aceasta arhitectura pentru crearea unui agent a cărui comportament să fie apropiat de cel uman în jocul *Quake 2*. În special învățarea comportamentului de țintire este folosita arhitectura *mixture of experts* (vezi figura 3.3). În cazul de fata mai multe rețele专特izate controlează sub-sarcini cum ar fi alegera armei, unele rezultate fiind folosite ca date de intrare pentru alți experți. Modelul creat este capabil să învețe și inacuratețea intenționată necesara pentru folosirea unor arme, dar și inacuratețea neintenționată cauzată de eroare umană. Totuși problema abordată de Gorman și Humphrys nu consideră reprezentarea comportamentului într-un mod structurat, prin urmare metoda implementată nu este ușor

modificabila fără reantrenare.

Folosirea arhitecturii *mixture of experts* a dat rezultate bune pentru modelarea comportamentului de țintire și tragere în *Quake 2*, unde care fiecare categorie de arma necesita adaptarea acestor comportamente [21].

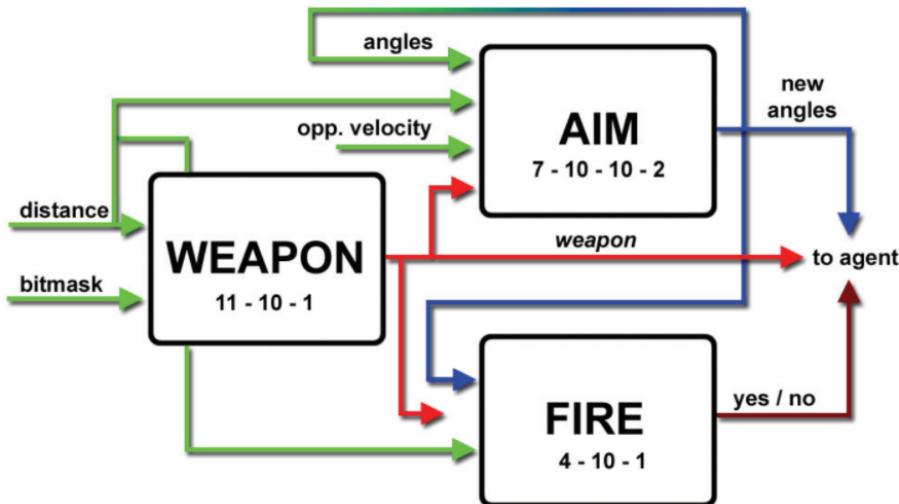


Figure 3.3: O serie de rețele neuronale interconectate folosite pentru a reproduce diferite aspecte comportamentului observat [7].

### 3.3 Învățare Imitativă

Domeniul învățării imitative poate fi împărțit în imitarea unui subiect extern sau auto-imitație. În studiul imitării unui subiect extern au fost aplicăți algoritmi de învățare a topologiilor, algoritmi probabilistici și algoritmi de inteligență artificială precum Self-Organizing Maps și arhitectura Mixture of Experts.

#### 3.3.1 Auto-imitație

Observare și imitația au fost folosite pentru a crea un mecanism ce permite unui agent uman "expert" să învețe un robot comportamentul necesar pentru rezolvarea unei probleme. Saunders *et al.* abordează problema învățării comportamentului adecvat de la un expert, dar propun folosirea *auto-imitației*, nu observarea și imitarea unui agent extern [15] [16]. Meritul unui astfel de mecanism fiind înlăturarea nevoii de a rezolva corespondența între agentul observat și

robot. Problema corespondentei este dificil de rezolvat, mai ales în cazul în care există diferențe mari între spațiul senzoriomotor al "elevului" și cel al "profesorului". Metoda propusă implica manipularea robotului de către expert folosind un set predefinit de primitive, în timp ce starea senzorilor interni (ex.: rotațiile articulațiilor) și externi este observată și asociată cu acțiunea aleasă de instructorul uman. Un spațiu  $k$ -dimensional ( $k$  fiind numărul de caracteristici ale vectorului de stare) este folosit drept memorie, memorând acțiunile ca puncte în acest spațiu. Un algoritm *k-Nearest Neighbour* este folosit pentru a alege acțiunea adecvată considerând experiențele de învățare anterioare. În plus, instructorul poate specifica fie o secvență de primitive executate independent de starea robotului fie o sarcină, care depinde de stare, și a cărui execuție are ca scop atingerea unei stări specifice de acesta. Sarcinile pot fi alcătuite din primitive, secvențe sau alte sarcini, iar pentru construirea comportamentului, instructorul înlănțuie sarcini, secvențe sau acțiuni.

Problema abordată de Saunders *et al.* este similară de cea pe care o abordăm dar difera prin folosirea *auto-imitației*. De asemenea, metoda prezentată necesită structurarea comportamentului de către instructorul uman, în timp ce în lucrarea de fata propunem să cream un mecanism ce poate genera și manipula structura comportamentului și fără intervenție umană.

### 3.3.2 Algoritmul Neural Gas

Algoritmul neural gas este folosit pentru învățarea iterativă a topologiei. Neuronii rețin câte un vector  $n$ -dimensional care le definește poziția în spațiu, acesta poziție fiind inițializată aleator. Algoritmul trece prin instante de date și modifica poziția unor neuroni apropiati formeză sinapse între aceștia. Numărul de neuroni modificați scăzând pe parcursul procesului de învățare. Sinaptele au un timp de viață limitat, la fiecare editare, timpul de viață al sinaptele celor mai apropiat neuron sunt incrementat, aceste fiind eliminate dacă depășesc un prag [10].

O îmbunătățire adusa algoritmului Neural Gas este *Growing Neural Gas*, aceasta metoda rezolva problema alegării numărului de neuroni la începutul antrenării. Metoda propusă este de a porni de la un număr mic de neuroni, adăugând la un număr fix de pași neuroni noi în zonele în care eroarea este maximă. Neuronul nou este inserat la jumătatea distanței între neuronul cu eroarea acumulată maximă și vecinul cu care media erorilor acumulate este maximă. Noul neuron va fi legat de cei 2, legătura inițială fiind eliminată [4]. Un exemplu al procesului de antrenare al acestui algoritm este prezentat în figura 3.4.

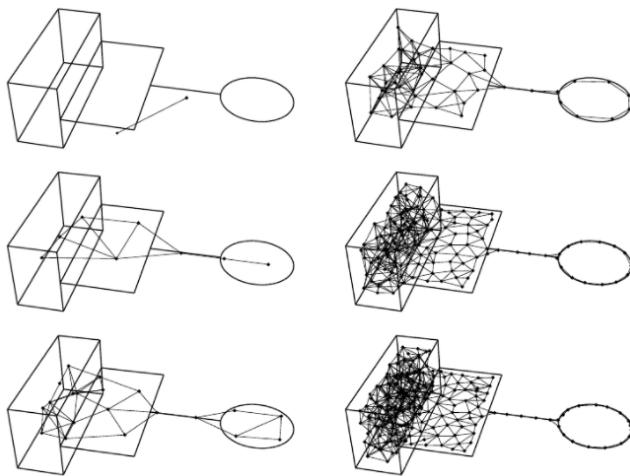


Figure 3.4: Rețeaua *neural gas* în diferite stagii de antrenare folosind metoda *growing neural gas* [4].

Algoritmul *Stable Growing Neural Gas* propune îmbunătățiri adiționale pentru *Growing Neural Gas*. Una dintre problemele ramase este creșterea continua a rețelei, criteriul de adăugare al noilor noduri fiind numărul pașilor de antrenare. Pentru a rezolva aceasta problema un nou criteriu este propus, un prag maxim de eroare care, odată depășit declanșează adăugarea noilor noduri. Acest nou criteriu păstrează abilitatea de adăugare a noilor neuroni, dar o limitează la zonele în care reprezentarea curentă a topologiei este nesatisfăcătoare. În plus criteriul specificat permite algoritmului să se adapteze la schimbări temporale ale topologiei, de exemplu descoperirea noilor zone, neexplorate în trecut, deoarece în acele zone eroare va fi mai mare comparat cu zonele deja explorate [18].

Diferite variante ale acestui algoritm sunt folosite în învățarea imitativa, pentru a învăța și a reprezenta topologia mediului înconjurător.

### 3.3.3 Comportament stratificat

Problema corespondentei este ușor evitată în medii virtuale unde agenții artificiali și cei umani interacționează cu mediul folosind aceeași interfață. În astfel de cazuri pachetele de date ce specifică acțiunile și informațiile despre mediu pot fi interceptate și folosite prin traficul de rețea [7] [22].

Thurau *et al.* au împărțit comportamentul jucătorilor pe 3 niveluri: strategic, tactic, și reacționar, urmărind să realizeze imitația folosind metode diferite pentru fiecare dintre acestea

[22]. Lucrarea lor definește comportamentul strategic ca fiind navigarea la nivel înalt între puncte în care se află resurse importante pentru jucător cunoscute și sub numele de "items" (ex.: arme, armuri sau muniție), care au efecte pe termen lung asupra stilului de joc. Pentru modelarea acestui nivel comportamental Thurau *et al.* au folosit algoritmul *Growing Neural Gas* [10] pentru a reprezenta topologia mediului, cunoscut și ca "harta". Datele de intrare ale algoritmului sunt pozițiile unui jucător pe durata unei sesiuni de joc, rezultând într-un graf similar unui *way-point map* folosit des în jocuri video. Asupra acestui graf au aplicat un *potential field* care are rolul de a ghida agentul spre cea mai atractivă poziție din punct de vedere strategic, depinzând de starea actuală a jocului, incluzând starea caracterului (vezi figura 3.5).

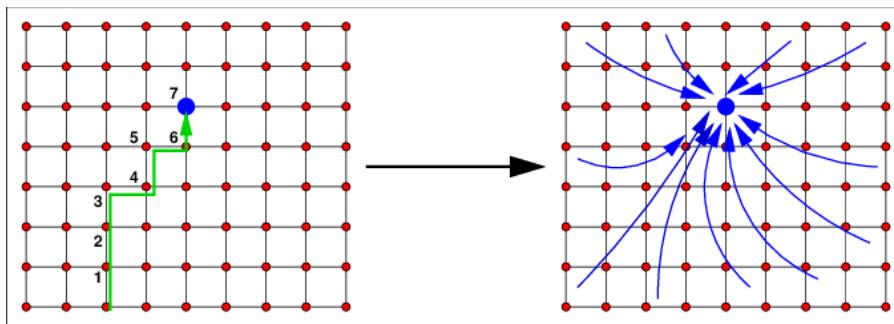


Figure 3.5: Misiunile efectuate de un jucător uman în reprezentarea topologică (stânga) și *potential field*-ul rezultat (dreapta) [22].

Stările observate în timpul antrenării sunt grupate în spațiul stărilor jocului. Nivelul reacționar reprezintă acțiuni efectuate în raport cu stare observată momentan de jucător, aceste acțiuni fiind inițial separate folosind un *Self Organizing Map* (SOM) [14]. În al doilea pas fiecărui neuron din SOM ii sunt asociate rețele neuronale antrenate pentru a îndeplini diferite funcții.

O îmbinare a modelului bazat pe primitive cu modelul comportamentului stratificat a fost realizată, primitivele motorii fiind folosite pentru a realiza un comportament de navigare între obiectivele strategice alese de către componenta strategică a modelului de comportament stratificat [6]. Problema înțelegerei comportamentului rămâne însă nerezolvată, metodele folosite neavând parametrii ușor editabili.

### 3.3.4 Modelarea comportamentului folosind structuri simple

Lent, Fisher, și Mancuso au creat un sistem de inteligență artificială care își poate explica propriul comportament, o funcționalitate care ar rezolva problema înțelegerii acțiunilor algoritmilor de inteligență artificială [25]. Lucrarea lor explica componenta inteligență a unui program de simulări militare folosit în antrenamentul soldaților, numit *Full Spectrum Command*. Acest program permite selectarea oricărui soldat și obținerea de informații despre comportamentul unității sale. În arhitectura sistemului agenții înțelegeți ce controlează soldații și cei ce controlează comandanții funcționează paralel și comunică prin mesaje, cele două componente sunt denumite *Control AI* respectiv *Command AI*. *Control AI* fiind responsabil de comportamentul reacționar și *Command AI* de cel tactic și strategic. Deși foarte eficient, sistemul poate folosi doar comportamente predefinite în setul de reguli atât pentru *Control AI* cat și pentru *Command AI*, fiind rigid în consecință.

Lent și Laird au creat un sistem ce creează un set de condiții și acțiuni observând un expert în procesul de completarea unei sarcini. Sistemul lor observă sub-sarcini selectate de expert în pilotarea unui avion, setul de condiții în care expertul efectuează diferite acțiuni și starea în care a fost aleasa respectiv completate o sarcina sau sub-sarcina. Aceste informații sunt mai apoi translate în reguli de producție ce coordonează un agent capabil să înlocuiască expertul uman. Un avantaj al sistemului creat este modularitatea acestuia, algoritmul de învățare putând fi înlocuit fără să afecteze sistemul de observație. De asemenea comportamentul este translatat într-un mod structurat, în reguli de producție. Cu toate acestea modelul lor nu funcționează în cazul în care seturile de condiții nu sunt disjuncte [26].

Aler *et al.* prezintă un alt exemplu care explorează folosirea structurilor simple *if-then-else*, de data asta în modelarea și imitarea comportamentului uman în jocul RoboSoccer. Comportamentul jucătorilor este observat sub forma de perechi *input-output* [1]. Ulterior setul de perechi este translatat în reguli de forma *if(state) then action* care sunt folosite pentru a controla agentul inteligent. Metoda lor produce rezultate mulțumitoare în contextul unei versiuni simplificate a jocului, dar setul mare de reguli care este generat face modificarea ulterioara impractică. De asemenea metoda de observație aleasa facilitează doar învățarea comportamentului simplu, de moment, învățarea comportamentului tactic fiind imposibila.

### 3.3.5 Modele probabilistice în imitație

Modele probabilistice au fost de asemenea folosite pentru a crea un agent AI în jocuri cu un grad mai mare de credibilitate. Un model Markovian a fost folosit pentru a selecta probabilistic acțiuni (vezi figura 3.6), cuplând cu aceasta metoda o împărțire a stimulilor în stimuli *high-level* și *low-level* [19] [20]. Un sistem ce simulează atenția a fost creat pentru a limita numărul stimulilor luați în considerare un moment dat. În aceste lucrări probabilitățile modelului Markovian sunt obținute prin observarea unui jucător uman în prealabil, observarea fiind realizată pe întreaga durată a vieții unui agent în joc, pentru a înțelege scopul acțiunilor acestuia. Împreună aceste sisteme creează un agent ce reacționează în mod variat stării în care se află și care e capabil să anticipeze pe termen scurt. Acest sistem este totuși incapabil să planuiască acțiuni pe termen lung, limitare considerată acceptabilă în contextul creării unui agent credibil în jocuri din genul *shooter*.

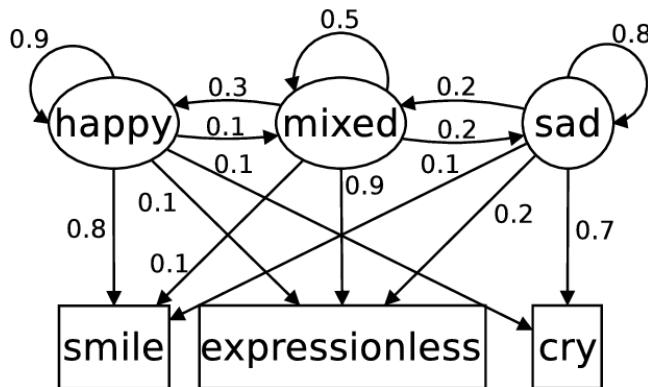


Figure 3.6: Un exemplu concret de model Markovian ascuns cu 3 stări ascunse și 3 observații reprezentând emoții [19].

Structura simplă a lanțului de probabilități este ușor de înțeles și modificat dar complexitatea înțelegerei și modificării crește cu numărul de legături. De asemenea dat fiind caracterul probabilistic al comportamentului acesta, rafinarea unei secvențe specifice este și ea îngreunată. În plus nevoia de a observa pe o perioadă lungă un agent în faza de antrenare prezintă un neajuns. O utilizare mai limitată a comportamentului probabilistic inserată în cadrul unei structuri în mare parte deterministe ar putea fi folosita, reducând totuși gradul de credibilitate, dar obținând astfel un model mai ușor de modificat.

# Chapter 4

## Metoda propusa

Rezolvarea eficientă a problemei necesită împărțirea acesteia în sub-probleme, urmând să propunem rezolvări pentru fiecare dintre acestea.

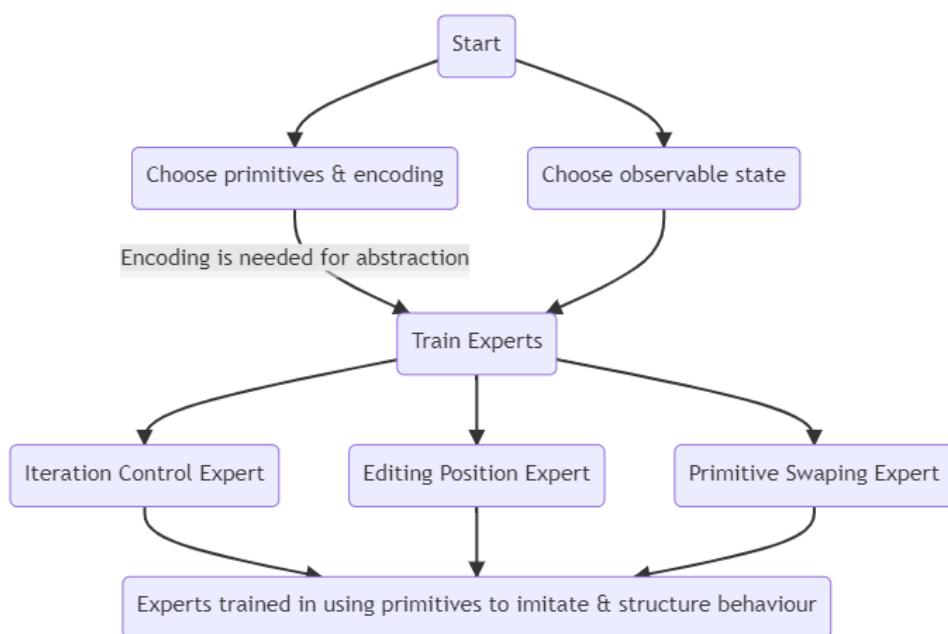


Figure 4.1: Pașii algoritmului de creare a agentului AI

Figura 4.1 prezintă pașii necesari în crearea algoritmului propus. Primii pași sunt alegerea stării observate și a primitivelor respectiv a codificării lor pentru abstractizarea acestora. Odată alese starea și acțiunile, următorul pas este antrenarea experților ce vor rezolva problema imitației, cei trei experți fiind fiecare responsabil de cate o parte a problemei. Odată antrenați experții ar trebui să fie capabili să imite orice comportament observat prin crearea unei structuri.

## 4.1 Pași preliminari

Algoritmul necesita rezolvarea unor probleme preliminare precum deciderea primitivelor care vor fi folosite, starea care va fi observata, reprezentarea acestora și deciderea experților ce vor completa fiecare sub-sarcina a imitării.

### 4.1.1 Observare, stări și primitive

O prima sub-problema ce trebuie rezolvata este selectarea stării ce va fi observate și setul de primitive ce urmează să fie folosite. Acesta problema nu are o soluție universală, fiind necesara selectarea manuală a primitivelor pe care le consideram adecvate, și a stării relevante depinzând de specificul problemei.

Problema colectării datelor prin observare este trivială în mediul virtual unde date ce descriu comportamentul putând fi accesate fie interceptând traficul de rețea fie folosind un API special conceput. În schimb observarea realizată folosind senzori în lumea reală este o problema complexă dar care nu face obiectul lucrării prezente.

### 4.1.2 Reprezentarea primitivelor

Reprezentarea primitivelor în structura creata este de asemenea un aspect ce necesita considerare. Reprezentarea propusa este un vector în care fiecare poziție reprezinta o zonă în care pot fi aplicate primitive. Valoarea unei poziții fiind o codificare numerică ce reprezinta varianta de primitivă folosită, sau absența unei primitive. Avantajul acestei reprezentări sta în aceasta fiind ușor de modificat algoritmice. În plus deși mai puțin compactă este utilă în modelarea comportamentelor motorii, pe care ne vom axa în lucrarea de fata.

De asemenea trebuie considerată structura pe care o vom crea. Datorita ușurinței de folosire și editare am ales să folosim un *behaviour tree*. În contextul actual acesta va fi unul simplificat, fiind mai apropiat de o secvență de noduri legate sub forma unei liste înlăntuite datorită lipsei de decizii, acesta fiind complet liniar.

### 4.1.3 Construirea comportamentului

Construirea comportamentului prezintă mai multe sub-probleme: identificarea stării de oprire respectiv a stării de continuare a editării, și realizarea modificării ce a fost decisa necesara. În plus reprezentarea aleasa introduce sub-probleme specifice. Mai exact, nevoia identificării poziției în vector unde se va realiza o modificare.

Primele două probleme sunt conectate una de alta, cele două decizii fiind exclusive mutual. Astfel aceasta sub-problema poate fi interpretată ca o problema de clasificare, unde clasificăm actuala stare a structurii comportamentului ca fiind fie adekvata, fie necesitând editarea. Aceasta sarcina consideram ca poate fi rezolvată de un singur expert. Problemele următoare nu pot fi însă rezolvate în aceeași manieră. Aici este necesara separarea în sub-experti fiecare responsabil de rezolvarea unei parti din problema.

Concret este nevoie de un expert ce determină poziția din vectorul de primitive unde trebuie efectuată o schimbare, iar în cele din urmă un expert ce poate modifica primitiva folosită. În acest caz editarea poate fi privită ca o problema de regresie sau ca una de clasificare. Alegerea punctului unde va fi realizată editarea va fi interpretat sub forma unei probleme de clasificare.

Structura de experti interconectați, care folosesc decizii luate de expertii anterior invocați reprezinta şablonul *mixture of experts* [9]. Avantajul acestui şablon sta în abilitatea acestuia de a evita una dintre cele mai mari probleme ale rețelelor neuronale, limitarea acestora la a învăța o singură sarcină specifică sau un set limitat de sarcini foarte similare.

După completarea etapei de antrenare a modelului urmează folosirea acestuia pentru a crea imitări structurate. Procesul de creare a comportamentului este unul iterativ, dar care va folosi doar o singura instanta de observare. Expertii vor fi folosiți pentru a genera o structură ce imita un comportament observat o singura data, astfel oferind capabilitatea de a realiza *single-take learning*. Odată structurat comportamentul, acesta este ușor de alterat atât pentru algoritm, cât și manual.

## 4.2 Setul de date

Pentru obținerea unui set de date pentru antrenarea expertilor am folosit aplicația practica dezvoltată, pe care o discutăm în detaliu în capitolul 5. Datele extrase în aplicație constau în rotațiile fiecărei articulații pentru toate cadrele ce formează o animație de mers. Animăția

folosita pentru generarea setului de date este un ciclu de mers, deoarece pe parcursul acestui ciclu sunt reprezentate aproximativ egal toate situațiile posibile de folosire a primitivelor, inclusiv mișcări subtile, cu rotații mici și mișcări rapide ce necesită rotații relative mari. Setul de date obținut conține aproximativ cinci sute de sample-uri pentru 8 perechi de articulații. Am observat totuși că mărimea setului de date este mult prea mică pentru a fi folosit în reinforcement learning, astfel am decis să generam date aleatorii pentru antrenare, aceste date acoperind uniform întregul interval de valori posibile, mai exact perechi de valori între 0 și 360, cu diferențe absolute de 1.75, aceasta valoare de a diferenței fiind aleasă pentru antrenarea în folosirea primitivelor descrise în 4.3.1 (vezi tabelul 4.2). Pentru antrenarea primilor doi experți va fi nevoie și de valorile configurației nodului actual, acestea vor fi de asemenea generate aleator, valoarea fiecărui element fiind codificarea prezentată în tabelul 4.2.

## 4.3 Structura Expertilor

În continuare vom explica detaliat structura expertilor. Pentru a ușura înțelegerea vom descrie structura acestora presupunând utilizarea lor în recrearea unui comportament motor observat la un model antropomorf simplificat (stick-man). În cazul imitării unui astfel de subiect vom considera doar observarea stării interne a acestuia, adică rotațiile articulațiilor acestuia pentru o durată arbitrară (tabelul 4.1).

Pentru simplitate vom observa o singura axă de rotație, mișcările posibile fiind destul de variate pentru a crea o secvență de mișcare destul de complexă. Odată decise datele ce vor fi observate trebuie alese și primitivele ce vor fi utilizate. În acest caz vom defini 4 primitive identice pentru fiecare articulație (tabelul 4.2). Două dintre ele vor reprezenta rotații în sens trigonometric de diferite amplitudini, celelalte două fiind oglinditul acestora în sens anti-trigonometric. Următorul pas este colectarea unui set de date pentru antrenament. Odată obținute aceste date, experții pot fi antrenați.

### 4.3.1 Datele de intrare și ieșire

Fiecare dintre experți va avea o sarcină diferită și în consecință fiecare dintre aceștia va avea date de intrare sau de ieșire diferite. Datele de intrare ale tuturor expertilor au fost însă normalizate, pentru a face mai ușor procesul de antrenare al acestora.

Parametru
Rotație bazin*
Rotație torso*
Rotație sold stâng
Rotație genunchi stâng
Rotație sold drept
Rotație genunchi drept
Rotație umăr stâng
Rotație cot stâng
Rotație umăr drept
Rotație cot drept
Rotație gât*

Table 4.1: Parametrii de stare ce vor fi observați în procesul de imitație. (\*nu sunt folosite)

Primitiva	Codificare
Primitiva neutra	0
$0.25^\circ$ trigonometric	1
$0.5^\circ$ trigonometric	2
$1^\circ$ trigonometric	3
$1.5^\circ$ trigonometric	4
$-0.25^\circ$ trigonometric	5
$-0.5^\circ$ trigonometric	6
$-1^\circ$ trigonometric	7
$-1.5^\circ$ trigonometric	8

Table 4.2: Primitivelor ce vor fi folosite și codificările acestora.

Primul expert, cel care decide continuarea iterațiilor de editare a unui nod, va avea ca date de intrare starea curentă, starea țintă și configurația actuală a nodului. Datele de ieșire sunt rezultatul clasificării în una din două clase: *accept* sau *continue* (figura 4.2.)

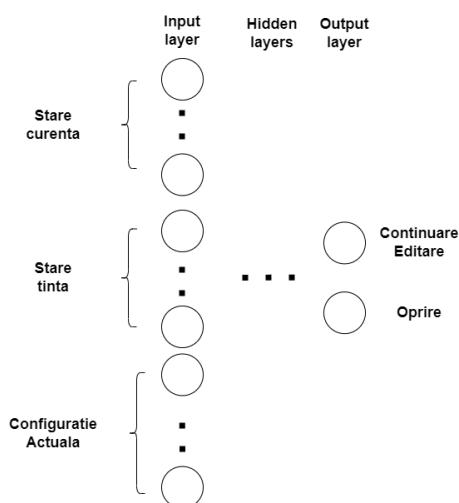


Figure 4.2: Layerele de input și output pentru rețea neuronală a expertului ce decide continuarea procesului de editare

Următorul expert invocat va decide poziția unde va fi realizată o modificare în cazul în care primul expert a considerat necesara continuarea editării nodului. Datele de intrare ale acestuia sunt identice primului expert. Acest expert va avea  $n$  clase ca posibile rezultate, acestea reprezentând pozițiile din vectorul de reprezentare (figura 4.3.) Va fi aleasa clasa cu cel mai mare grad de încredere rezultat.

Ultimul expert invocat în procesul de editare este cel responsabil cu schimbarea primitivei folosite pe poziția determinată anterior. În aceasta lucrare vom interpreta operația de editare ca fiind o problema de clasificare, fiecare primitivă sau lipsa unei primitive fiind reprezentate de o clasa codificată și sub forma numerică. Datele de intrare ale acestuia sunt starea curentă, starea țintă, selectând pentru ambele doar valorile de pe poziția aleasă la pasul anterior, rezultatul fiind clasa ce reprezinta noua primitivă (figura 4.4.)

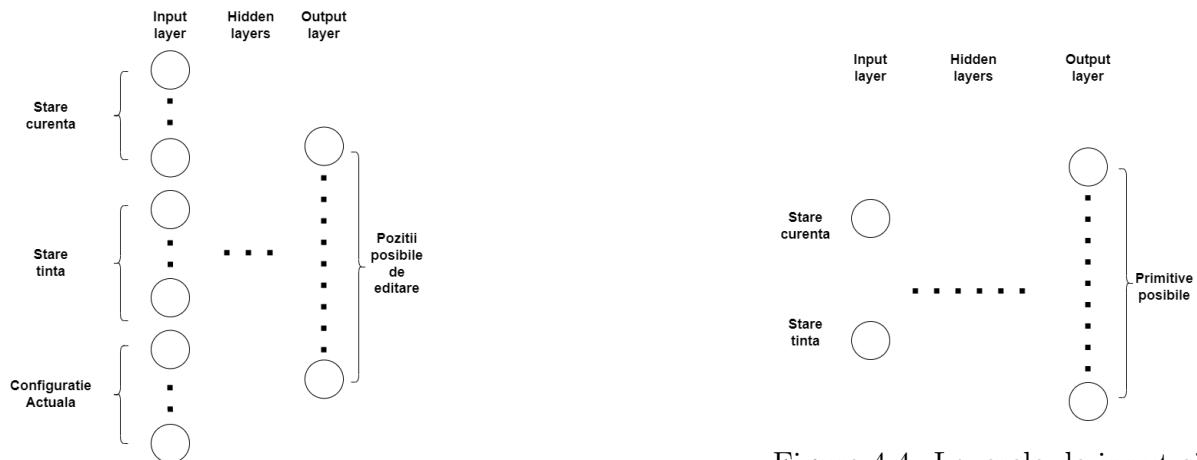


Figure 4.3: Layerele de input și output pentru rețeaua neuronală a expertului ce determină poziția editării

Figure 4.4: Layerele de input și output pentru rețeaua neuronală a expertului care efectuează schimbarea primitivei

Odată antrenați experții aceștia ar trebui să înțeleagă folosirea diferitelor variante ale primitivelor pentru a face tranziția între 2 stări observate. Pentru a obține o structură ce imita un comportament un subiect va fi observat manifestând comportamentul respectiv și se vor extrage date despre starea subiectului în diferite momente în timp. Acestea vor fi folosite drept date de intrare pentru algoritm. Datele de ieșire ale algoritmului fiind secvența de noduri ce reprezinta behaviour-tree-ul.

## 4.4 Agentii de Reinforcement Learning

Datorită caracterului problemei este dificila obținerea unui set de date potrivit pentru *Supervised Learning*, de aceea ne propunem să folosim *Reinforcement Learning* pentru a crea agenți pentru fiecare expert necesar, și de a îi antrena folosind funcții de reward personalizate, aceasta sarcina fiind mult mai ușoara comparativ cu obținerea unui set de date ce conține *label*-uri

pentru antrenarea fiecărui expert.

În scopul antrenării agenților este necesara crearea de *environment*-uri potrivite pentru fiecare dintre agenți. Agenții vor fi implementați folosind metoda DQN *Deep Q Networks*, descrisă de Mnih *et al.* [11]. Aceștia vor avea rețea neuronala pentru predicția rewardului pentru fiecare acțiune posibilă în funcție de starea observată (Q-network) și o politica ce gestionează alegerea acțiunii folosind predicțiile făcute de Q-network.

#### 4.4.1 Agentul de selectarea a primitivelor

Environment-ul dedicat antrenării expertului care alege primitiva optima pentru poziția aleasă de expertul anterior a fost primul dezvoltat, aceasta fiind sarcina elementara a algoritmului. Prima versiune a environment-ului a fost creată iterând prin toate pozițiile posibile pentru tot setul de date.

##### 4.4.1.1 Experimentarea cu funcții de reward

Am experimentat cu mai multe funcții de reward. O prima versiune a funcției de reward compara diferența între configurația anterioară a noului în poziția selectată cu noua configurație. Compararea configurațiilor se făcea judecând dacă noua configurație acoperă mai exact diferența de rotație între starea țintă și starea curentă, pentru articulația respectivă poziției selectate. Reward-ul era 1 dacă noua configurație scădea sau menținea eroarea absolută, respectiv 0 dacă noua configurație conducea spre o eroare mai mare. Aceasta funcție nu a dat rezultate, și în consecință am considerat folosirea altrei funcții. Următoarele iterații ale funcției au renunțat la folosirea stării precedente, schimbând abordarea pentru calcularea reward-ului la a judeca cât de aproape este noua primitiva de optim. Pentru a realiza acest calcul, la fiecare pas de antrenare evaluam toate primitivele posibile pentru a găsi primitiva optima și respectiv cea mai puțin corectă. Primitiva aleasă primește o valoare normalizată între 1 și 0, 1 semnificând alegerea primitivei optime și 0 reprezentând alegerea primitivei ce generează eroarea maxima. Algoritmul 1 reprezinta formula de calcul a reward-ului normalizat.

Aceasta varianta a funcției de reward a suferit mai multe modificări, precum ridicarea la putere a reward-ului normalizat, pentru a premia exponențial mai mult soluțiile mai apropiate de optim (vezi figura 4.5). O alta modificare a fost lărgirea intervalului de reward, de la [0,1] la [0,10] respectiv [0, 100], pentru a diferenția mai mult soluțiile bune de cele rele. Folosind

**Algorithm 1** Agent Reward algoritm

---

```

BEGIN
stateDelta = targetState - currentState
currentDelta = |stateDelta - chosenPrimitive|
optimalDelta = currentDelta
worstDelta = currentDelta
for primitive in primitiveSet do
    primitiveDelta  $\leftarrow$  |stateDelta - primitive|
    if primitiveDelta > worstDelta then
        worstDelta  $\leftarrow$  primitiveDelta
    end if
    if primitiveDelta < optimalDelta then
        optimalDelta  $\leftarrow$  primitiveDelta
    end if
end for
reward = 1 - (delta - optimalDelta) / (worstDelta - optimalDelta)
END

```

---

aceste modificări am observat o îmbunătățire în reward-ul mediu notabilă, dar nu suficient de mare pentru a obține un model util în aplicații practice.

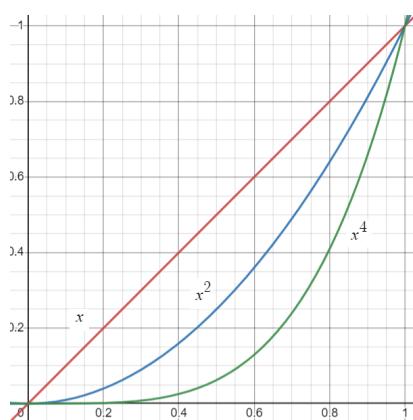


Figure 4.5: Am încercat modificarea pantei funcției de reward pentru a premia mai mult stările mai aproape de optim

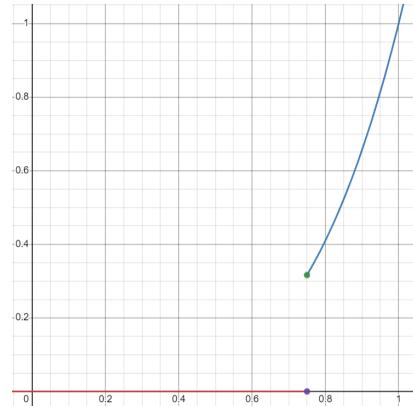


Figure 4.6: O alta variantă a funcției de reward, care premiază cu 0 soluțiile care nu obțin între 0.75 și 1 în faza de normalizare

În cele din urma am încercat folosirea unui *cut-off* pentru valorile funcției de reward, premiind cu 0 soluțiile care se află în intervalul  $[0, 0.75]$  în faza de normalizare (vezi figura 4.6), lăsând intact premiul în intervalul  $[0.75, 1]$ . După aplicarea *cut-off*-ului am păstrat modificarea pantei, prin ridicarea la a 4-a și am largit intervalul de valori la  $[0, 10]$ .

#### 4.4.1.2 Alegerea Hiper-parametrilor algoritmului

O alta problema întâlnita este lipsa unui trend de îmbunătățire pe parcursul antrenării, chiar și după 200 de episoade de antrenare (vezi figura 4.7). Aceasta problema a fost atribuită estimărilor nerealistic de mari făcute de Q-network, de peste 300, intervalul de valori ale reward-ului fiind  $[0, 10]$ , împreună cu tăierea valorilor pe care o făcea politica Boltzmann. Acest fenomen a fost corectat schimbând politică la una Epsilon Greedy și scăzând substanțial leaning rate-ul de la  $10^{-3}$ , o valoare potrivită în general pentru *supervised learning*, la  $10^{-7}$  (vezi figura 4.8). Politica Epsilon Greedy, balansează explorarea și exploatarea folosind o valoare epsilon pentru a determina fie alegerea unei acțiuni aleatoare fie a celei mai bune acțiuni prezisa de Q-network. Pentru acest prim experiment am ales o valoare de 0.2. Deși ploturile au scale diferite pentru reward, provenite din experimentarea cu diferite funcții de reward intre experimente, lipsa unei îmbunătățiri este aparentă când comparați prima figura cu cea de-a două. Am încercat și folosirea metodei *double-networks*, descrisă de Van Hasselt *et al.*, care ajuta la ameliorarea tendinței de preziceri supra-optimiste și a fenomenului de *overfitting* ale metodei DQN clasice [24], aceasta nu a oferit totuși schimbări observabile în performanța modelului.

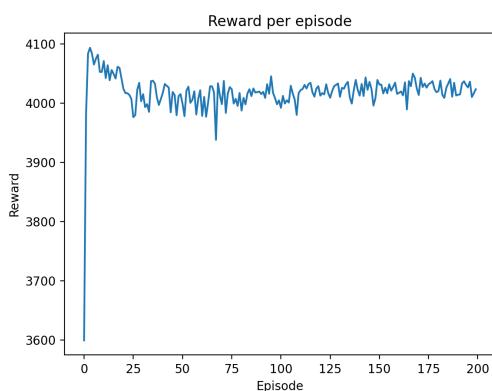


Figure 4.7: Reward-ul obținut folosind learning rate-ul  $10^{-3}$ , primul episod fiind "warm-up" cu acțiuni selectate aleator

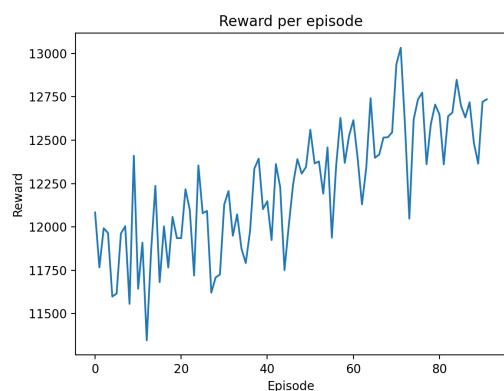


Figure 4.8: Reward-ul obținut folosind learning rate-ul scăzut la  $10^{-7}$  și politica Epsilon Greedy

O alta direcție de experimentare a fost folosirea unei politicii Linear Anneal combinată cu Greedy Epsilon, aceasta combinație rezultând în folosirea unei valori epsilon interpolate între o valoare de pornire și un minim pe o perioadă dată. În acest caz am folosit o valoare inițială 1 și minimul 0.1, cu o perioadă de interpolare de 50 de episoade. Pentru acest experiment lungimea episoadelor este de 500 de pași unici, fiecare pas fiind repetat de 2 ori, deci 1000 de pași. Modelul

a fost antrenat pe 1000 de episoade, cu un *batch size* de 500 de sample-uri și o memorie de 10000 de sample-uri (ultimele 10 episoade), funcția de reward folosita a fost varianta "cut-off" descrisa la sfârșitul secțiunii 4.4.1.1 figura 4.9 arata rezultatul obținut. Am observat un trend de învățare în primele 500 de episoade, după care acest trend a fost pierdut.

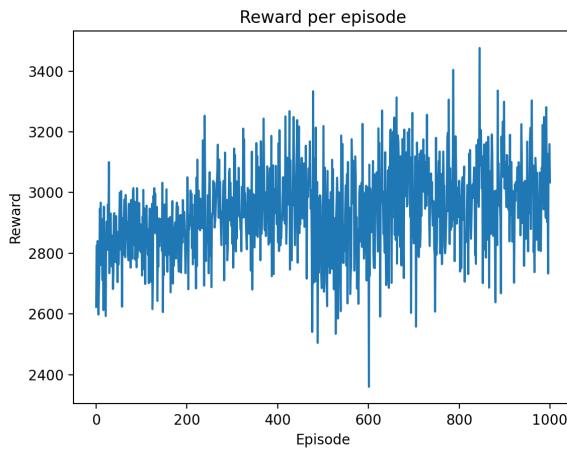


Figure 4.9: Reward-ul obținut de expert folosind politica Annealed Epsilon Greedy

Am organizat mai multe seturi de experimente (vezi tabelul 4.4), fiecare set ajutând valoarea unei parametri al agentului sau a politicii acestuia precum: **Anneal Period**, **Min Epsilon**, **Episode Length**, **Memory**, și **Batch Size**, fiecare dintre acestea având un rol important în procesul de învățare.

Modificarea valorii parametrului **Anneal Period** nu a rezultat în schimbări observabile ale rewardului obținut, acesta menținându-se în intervalul [2600-3400], care este apropiat de valorile obținute ale cănd aleatoriu o acțiune la fiecare pas al episodului. Alterarea experimentală a valorilor parametrilor **Min Epsilon**, **Epsiode Length**, **Memory**, și **Batch size** de asemenea nu au rezultat în schimbări în performanța agentului. Aceste rezultate ne indică faptul că problema ce împiedică procesul de învățare nu este legată de alegerea nepotrivita a parametrilor agentului.

#### 4.4.1.3 Optimizatori & Learning Rates

Deoarece experimentele descrise în secțiunea 4.4.1.2 nu au rezultat în îmbunătățiri ne-am întors atenția spre optimizatori și parametrii acestora. Am dorit să comparăm rezultatele obținute de optimizatorul folosit pana acum, Adam, cu un optimizator mai simplu, anume SGD (Stochastic

Base Rate	Decay Period	Decay Rate
$10^{-6}$	500	0.5
$10^{-6}$	500	0.2
$10^{-6}$	500	0.8
$10^{-6}$	1000	0.5

Table 4.3: Configurații ale parametrilor formulei de scădere exponențială a learning rate-ului care au fost folosite în experimente

Parametru Ajustat	Valoare
Anneal Period	50*
Min Epsilon	100
Epsiode Length	0.1*
Memory	0.05
Batch size	1000*
	2000
	10000*
	20000
	1000*
	2000

Table 4.4: Configurațiile experimentelor pentru ajustarea hiper-parametrilor agentului, \*-valorile de baza pentru restul experimentelor

Gradient Descent). Performanta agentului nu a fost afectata de aceasta schimbare, rezultatele amândurora fiind apropiate de rezultate obtinute prin alegeri aleatorii.

Următorul experiment a implicat ajustarea parametrilor optimizatorului. Am dorit să schimbam learning rate-ul de la o valoare fixă la una dinamică, care să pornească de la o valoare mai mare și care să scadă pe parcursul antrenării folosind formula 4.1. Am folosit mai multe configurații pentru acest experiment (vezi tabelul 4.3) care creează următoarele grafice pentru learning rate (vezi figura 4.10).

$$\text{learningRate}_{\text{step}} = \text{baseRate} * \text{decayRate}^{\frac{\text{step}}{\text{decayPeriod}}} \quad (4.1)$$

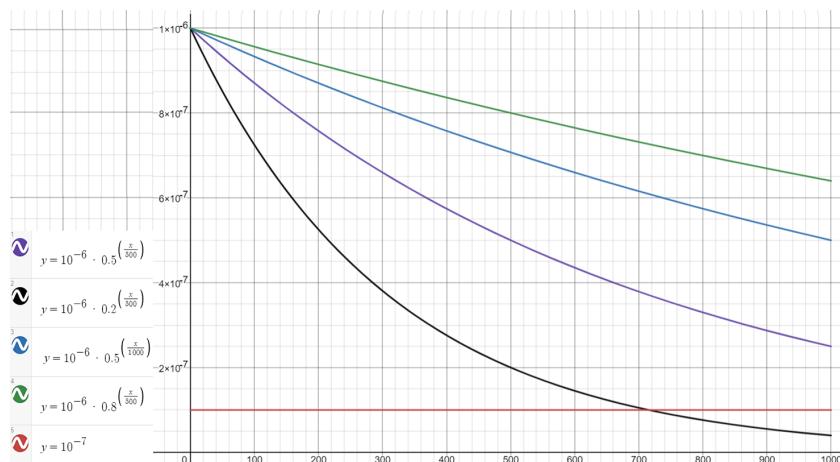


Figure 4.10: Learning rate-urile pentru fiecare experiment descris

Ajustarea parametrilor nu a cauzat schimbări observabile în reward-ul obținut de agent.

#### 4.4.1.4 Topologia expertului

Am testat experimental modele diferite pentru expertul de editare/selectare a primitivelor. Primul model testat este compus din 3 layere ascunse cu cate 9 neuroni fiecare (vezi figura 4.11). A doua variantă a rețelei pentru al treilea expert este formata din 9 layere ascunse, de cate 9 neuroni fiecare 4.12.

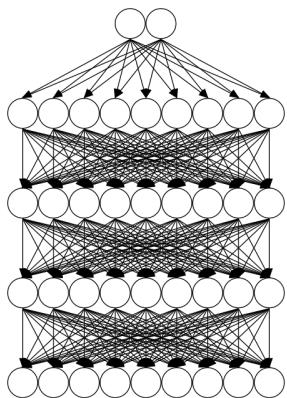


Figure 4.11: Topologia celei de-a treia rețele expert, pentru primul experiment

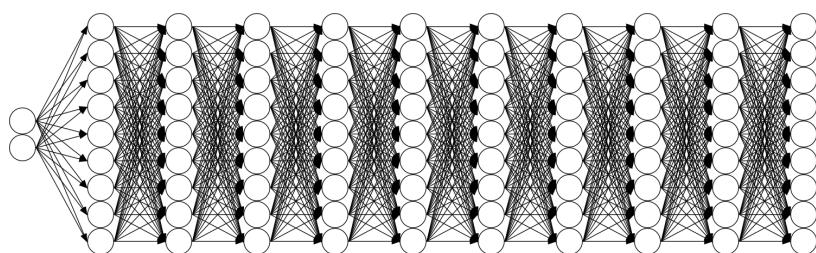


Figure 4.12: Topologia celei de-a treia rețele expert, folosita în al doilea experiment

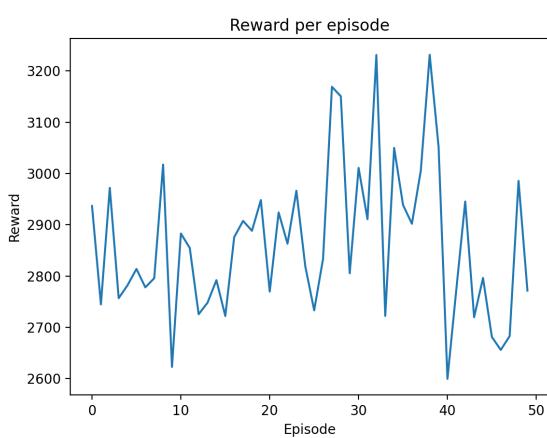


Figure 4.13: Reward-ul obținut folosind o rețea de mici dimensiuni

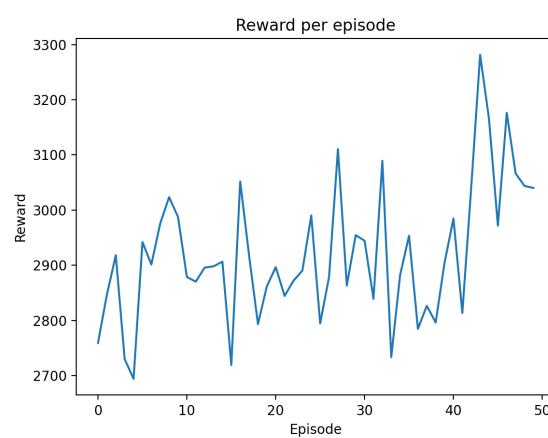


Figure 4.14: Reward-ul obținut folosind o rețea de dimensiune medie

Am observat ca obținem rezultate similare în experimentele de antrenare a celor două topologii, folosind un număr mic de episoade pentru ambele (vezi figurile 4.13, 4.14).

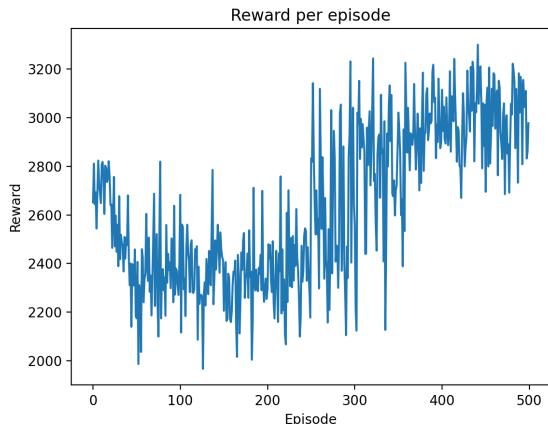


Figure 4.15: Reward-ul obținut folosind rețeaua de dimensiune mica, antrenată pe mai multe episoade

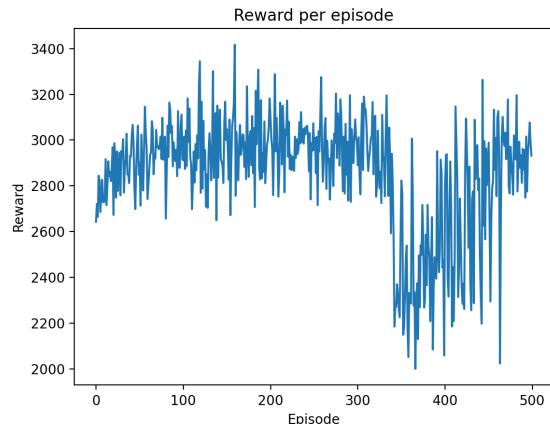


Figure 4.16: Reward-ul obținut folosind rețeaua de dimensiune medie, antrenată pe mai multe episoade

Am repetat procesul de antrenare pentru un număr mai mare de episoade, folosind ambele rețele. Nici una dintre rețele nu a obținut rezultate mai bune odată cu mărirea perioadei de antrenare (vezi figurile 4.13, 4.14). Putem concluziona ca mărimea modelului sau o perioada de antrenare insuficientă nu stau la sursa aceste probleme.

#### 4.4.1.5 Pre-procesarea datelor

Deoarece am explorat toate celelalte posibilități în limite rezonabile și nu am găsit o soluție pentru a remedia inabilitatea agentului nostru de a învăță sarcina curentă suntem nevoiți să re-evaluăm un aspect fundamental al problemei: relația între datele de intrare și acțiunile optime.

Datele de intrare ale acestui expert au trecut deja printr-o fază de preprocesare, valorile, original în intervalul [0,360] fiind normalize și aduse în intervalul [0,1]. Acest proces nu rezolvă însă o mare problema intrinsecă setului de date, datele de intrare, starea curentă respectiv starea țintă a unei articulații au o proprietate ce face extrem de dificila învățarea acțiunilor potrivite, aceste două valori difera foarte puțin una de alta, aproximativ  $\pm 2^\circ$ . Din aceasta cauza, perechile de date care ar trebui să conducă la acțiuni diferite sunt extrem de apropiate. Astfel învățarea de acțiuni este foarte dificila deși datele pot fi foarte bine separate (vezi figurile 4.17, 4.18). Aceasta distribuție a datelor explica și inaptitudinea agentului de a învăța pana acum, precum și nevoie de a folosi un learning rate foarte mic pentru a obține vreun fel de îmbunătățiri,

deoarece puncte foarte apropiate necesită acțiuni diferite ajustările făcute, deși foarte mici afectează enorm performanța modelului. O dispersie mai mare a datelor ar face procesul de învățare mult mai ușor. Pentru a mari intervalul de dispersie a valorilor am considerat necesar un pas adițional de preprocesare.

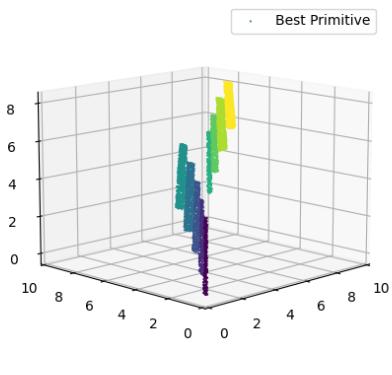


Figure 4.17: Distribuția grafică a diferitelor clase, pe axele  $x$  și  $y$  sunt reprezentate starea curentă respectiv starea țintă, și pe axa verticală,  $z$  acțiunea optimă. Graficul prezintă doar valorile din intervalul  $[0,10]$ , nu întregul interval  $[0,360]$  pentru axele  $x$  și  $y$

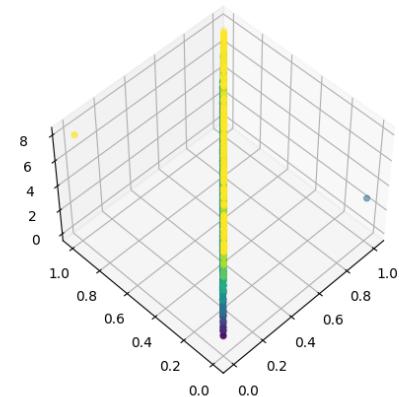


Figure 4.18: Distribuția grafică a diferitelor clase, pe întregul intervalul normalizat

În starea actuală perechile de date aparținând de clase diferite se asemănă unui fascicul, benzile fiecărei clase fiind paralele fata de diagonala. Dorim ca noua funcție de preprocesare să imite efectul unei lentile *fisheye*, curbând benzile care nu se află direct pe diagonala destul de mult încât să formeze diferențe mai mari intre acestea, pe care algoritmul ar putea să le învețe mai ușor. Pentru a obține acest efect am folosit funcția descrisă de ecuația 4.2. Funcția are ca date de intrare  $x$  - starea curentă normalizată intre 0 și 1 și  $c$  - diferența intre stări înainte de normalizare, rezultatul funcției fiind starea țintă cu valori în intervalul  $[0,1]$ .

$$F(x, c) = \begin{cases} 1 - (1 - x^w)^{\frac{1}{w}} & w = e^{|c|}, \quad c \geq 0 \\ (1 - (1 - x)^w)^{\frac{1}{w}} & w = e^{|c|}, \quad c < 0 \end{cases} \quad (4.2)$$

Folosind aceasta funcție obținem o dispersie egală pe ambele axe. Figura 4.19 reprezinta noua distribuție, în spațiul normalizat  $[0,1]$ . Putem observa păstrarea delimitărilor clare intre diferitele clase, acum disperse într-o arie mai mare a spațiului normalizat.

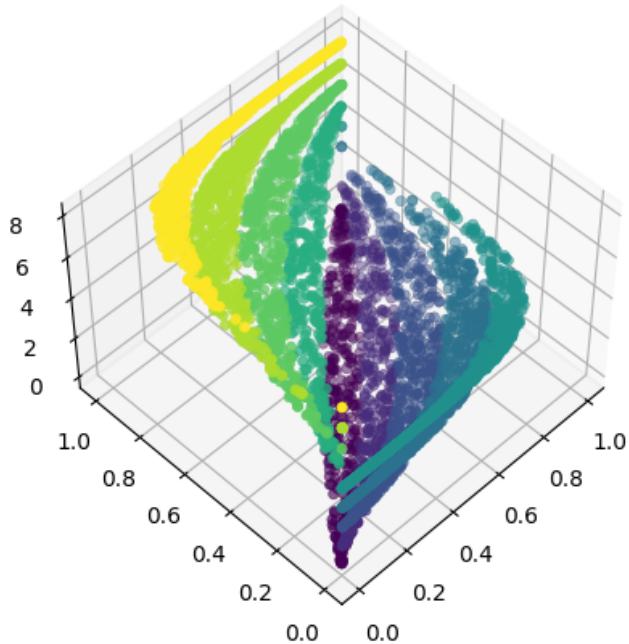


Figure 4.19: Distribuția grafica a diferitelor clase după aplicarea funcției de preprocesare, reprezentata în spațiul normalizat

Am dorit ca funcția de preprocesare să fie simetrică față de funcția identitate  $y = x$  pentru perechi  $(c, -c)$ , pe intervalul  $[0, 1]$ . Pentru a obține o astfel de funcție aveam nevoie de o funcție cu ramuri, cu proprietatea ca sub-funcția ce da valori pentru  $c \geq 0$  să fie inversa sub-funcției ce da valori pentru  $c < 0$ . Am pornit de la funcția  $y = 1 - (1 - x^2)^{\frac{1}{2}}$  înlocuind 2 cu  $w$ , a cărui valoare a fost inițial  $c$ , diferența între stări. Am determinat că  $w = c$  nu este corect deoarece pentru  $c = 0$ , însemnând ca stările coincid, funcția obținuta nu era funcția identitate. Pentru funcția finală am determinat  $w = e^{|c|}$  ca fiind valoarea potrivita. Folosind aceasta formula obținem funcția identitate pentru  $c = 0$ , respectiv o funcție convexă pentru  $c \geq 0$ . Din nefericire aceasta funcție nu generează curbe simetrice față de funcția identitate pentru  $c$  și  $-c$ . Din aceasta cauza am fost nevoiți să folosim inversa funcției pentru a obține simetria pentru valorile negative ale lui  $c$ , adică  $y = (1 - (1 - x)^w)^{\frac{1}{w}}$ , păstrând aceeași formula pentru  $w$ .

Am efectuat un prim experiment de folosire a acestei noi reprezentări a datelor pentru a observa efectul pe care îl are asupra abilității agentului de a învăță. Acest prim experiment a fost efectuat folosind un model foarte simplu cu 2 layere ascunse cu cate 9 neuroni, totodată

crescând learning rate-ul la  $10^{-5}$ . Experimentul a avut rezultate pozitive, reward-ul obținut spre sfârșitul antrenării fiind în intervalul [4500, 5000], o îmbunătățire majoră fata de rezultatele experimentelor anterioare, care obțineau reward-uri în intervalul [2500, 3000]. Văzând ca acest model foarte simplu a reușit să obțină un astfel de rezultat după 500 de episoade am decis să folosim modelul descris în figura 4.11, care nu a reușit să obțină reward-uri în afara intervalului [2500-3000]. Modelul a fost antrenat pentru 1000 de episoade, dublând și memoria agentului de la 10000 de sample-uri la 20000. Acest experiment a avut de asemenea succes, obținând reward-uri în intervalul [5500, 6000] spre sfârșitul antrenării (vezi figurile 4.20, 4.21).

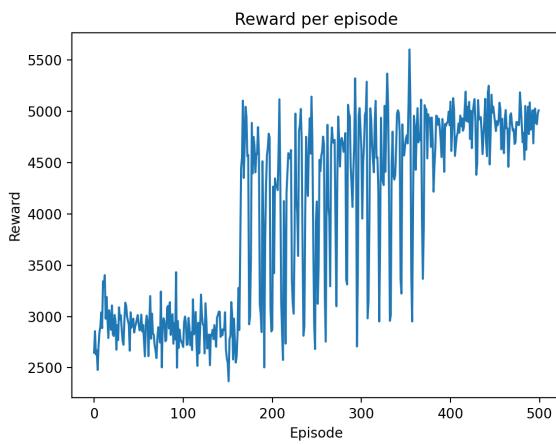


Figure 4.20: Reward-ul obținut de modelul foarte simplu, folosind noua funcție de preprocesare

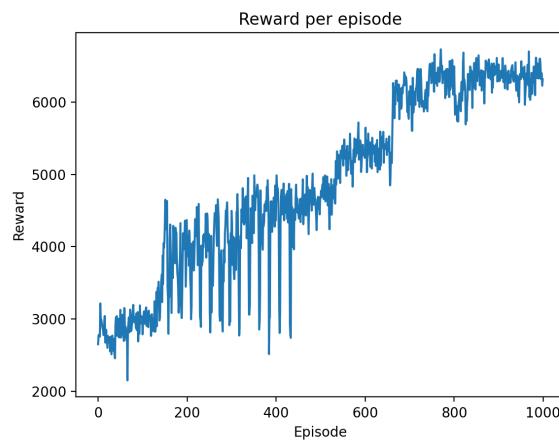


Figure 4.21: Reward-ul obținut de modelul 4.11, folosind noua funcție de preprocesare

#### 4.4.2 Agentul de selecție a poziției de editare

Pentru expertul care decide poziția de editare am folosit o metoda similară metodei finale pentru expertul discutat anterior. Environment-ul acestui expert constă într-un spațiu de observare format din starea curentă, starea țintă și configurația actuală a nodului, spațiu de acțiuni fiind format din acțiuni corespunzătoare către unei poziții din nod. Reward-ul pentru acțiuni este fie 0 fie 1. Pentru a calcula reward-ul determinam dacă se poate face o modificare pe poziția indicată care să rezulte într-o aproximare mai exactă a diferenței între starea curentă și starea țintă pentru poziția aleasă. Dacă există o astfel de modificare considerăm că reward-ul este 1, iar în caz contrar reward-ul va fi 0.

Datele de antrenare pentru acest expert au fost generate aleator, datorită acelorași prob-

leme întâlnite la expertul prezentat anterior. Pentru a obține un set de date cât mai balansat pentru antrenare am creat următoarea procedura pentru schimbarea stării după o acțiune. Dacă acțiunea selectată a fost o poziție pe pentru care putem obține o configurație mai bună vom modifica primitiva acelei poziții, înlocuind-o cu cea optimă. În toate celelalte cazuri, dacă starea nu este optimă, vom selecta o poziție aleator dintre pozițiile non optime și o vom modifica, realizând aceeași înlocuire descrisă mai sus, iar dacă starea este optimă cream o nouă stare aleatoare. Astfel, indiferent de configurația nodului sau a stărilor nodul se va mișca spre optim, asigurând ca după cel mult 8 pași prezentăm o stare optimă, care în mod real este mult mai rara.

#### 4.4.2.1 Topologia expertului

Am testat experimental 2 modele pentru expertul de selectare a poziției de editare. Primul model testat este compus din 3 layere ascunse cu cate 24, 16 respective 8 neuroni fiecare (vezi figura 4.22). Al doilea expert a fost creat prin dublarea fiecărui layer ascuns, având în aceasta ordine 2 layere cu cate 24 de neuroni, 2 layere cu cate 16 neuroni respectiv 2 layere cu cate 8 neuroni.

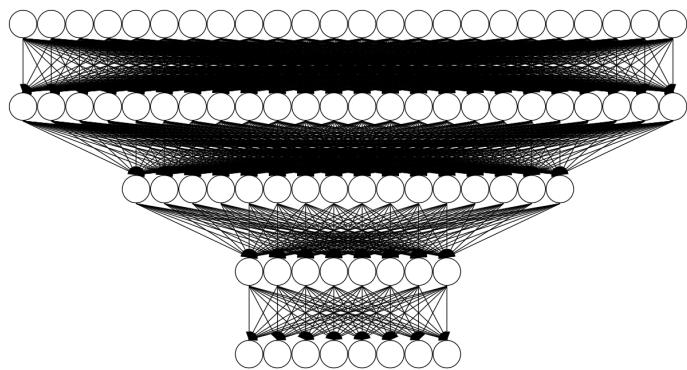


Figure 4.22: Arhitectura rețelei care decide poziția de editare pentru iterarea curentă

Folosind lectiile învățate antrenând expertul de selectare a primitivei optime am reușit să obținem mult mai ușor un model ca obține rezultate. Am antrenat ambele modele descrise mai sus, pentru 1000 de episoade a cate 1000 de pași fiecare, agentul folosind o politica Annealed Epsilon Greedy cu epsilon inițial 1, epsilon final 0.1 și o perioada de interpolare de 100 de episoade. Agentul are o memorie de 20000 de pași, selectând batch-uri de cate 1000 la fiecare

pas de antrenare, la sfârșitul fiecărui episod.

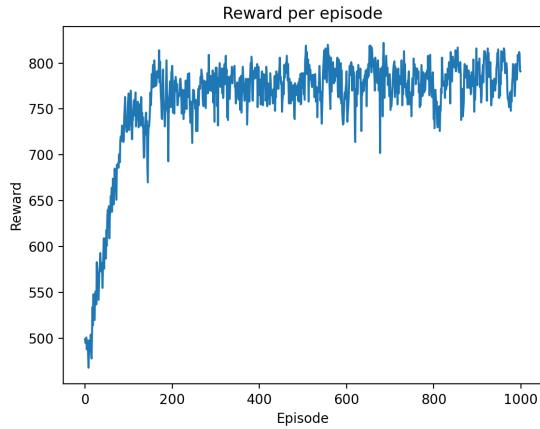


Figure 4.23: Reward-ul obținut de agent folosind rețeaua cu 3 layere ascunse

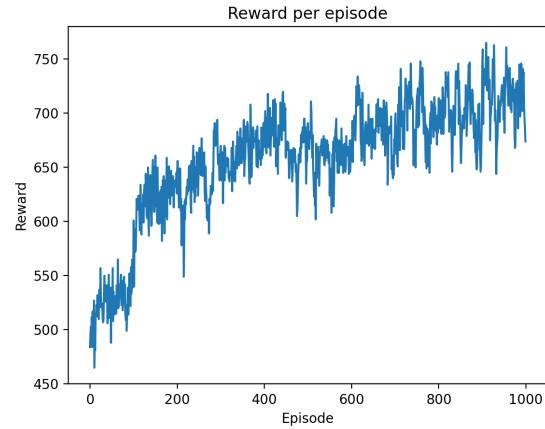


Figure 4.24: Reward-ul obținut de agent folosind rețeaua cu 6 layere ascunse

Cele două rețele au obținut rezultate similare, obținând un reward mediu de aproximativ 0.8 pe pas, rezultând într-un reward per episod de 800, maximul fiind 1000 (vezi figurile 4.23 și 4.24). A doua figura, reprezentând reward-ul obținut de agentul cu rețeaua de dimensiuni mai mari nu obține în aceeași perioadă rezultate mai bune, ambii agenți convergând, primul după 200 de episoade, al doilea doar după 1000 de episoade.

Având în vedere rezultatele obținute, concluzionam ca rețeaua de mici dimensiuni este suficientă pentru aceasta sarcina.

#### 4.4.3 Agentul de control al iterăției

În cele din urma, pentru expertul care decide continuarea editării, am creat un environment similar celui descris mai sus, acesta fiind diferențiat prin funcția de reward și spațiul de acțiuni. Acest agent va putea alege între 2 acțiuni, 0 sau 1, reprezentând oprirea respective continuarea procesului de editare. Funcția de reward a acestui environment ia fie valoarea 0 fie 1 și este calculată în următorul fel. Dacă există orice acțiune de editare care rezulta într-o diferență totală mai mică între mișcarea observată și mișcarea efectuată de agent și agentul a ales acțiunea 1 sau dacă configurația actuală este optimă iar agentul a ales acțiunea 0 atunci reward-ul este 1, altfel reward-ul este 0 (vezi tabela 4.5). Environment-ul de antrenare se mută la fiecare pas în care se alege continuare iterării spre o configurație optimă a nodului, pentru a simula funcționarea celorlalți doi experti.

Stare Acțiune \	Non-optim	Optim
Acțiune 0	0	1
Acțiune 1	1	0

Table 4.5: Rewardurile obținute în environment-ul creat pentru antrenarea expertului ce decide continuarea procesului de editare

Și pentru acest expert am aplicat pasul de preprocesare descris mai sus, de aceasta data pe întregul vector de stare curentă respectiv stare țintă.

#### 4.4.3.1 Topologia expertului

Datorita experimentelor realizate în timpul antrenării expertilor anterior descriși am considerat ca nu mai este necesara folosirea unor rețele mai mari pentru expertul responsabil de controlul iterătiei. Am ales sa folosim o arhitectura similară expertului ce decide poziția de editare, adică 3 layere ascunse, formate din cte 24, 16 respectiv 8 neuroni.

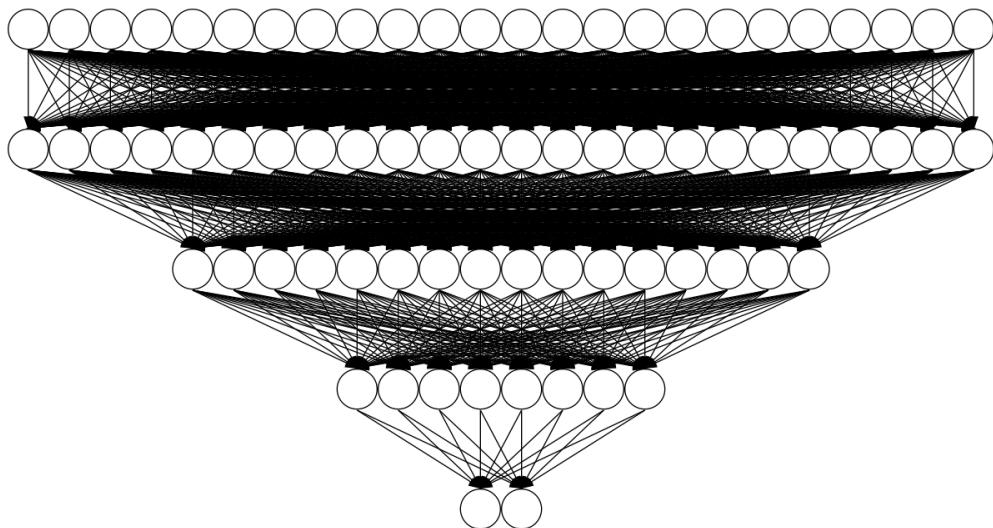


Figure 4.25: Topologia rețelei primului expert, responsabila de controlul iterătiilor

Antrenarea acestui expert a dat rezultate bune, reward-ul mediu final fiind 0.85 cu un maxim de 1. Agentul a atins punctul de convergență într-un timp comparabil cu expertul anterior (vezi figura 4.26), după doar 200 de episoade, 1000 de episoade fiind excesive și pentru acest expert.

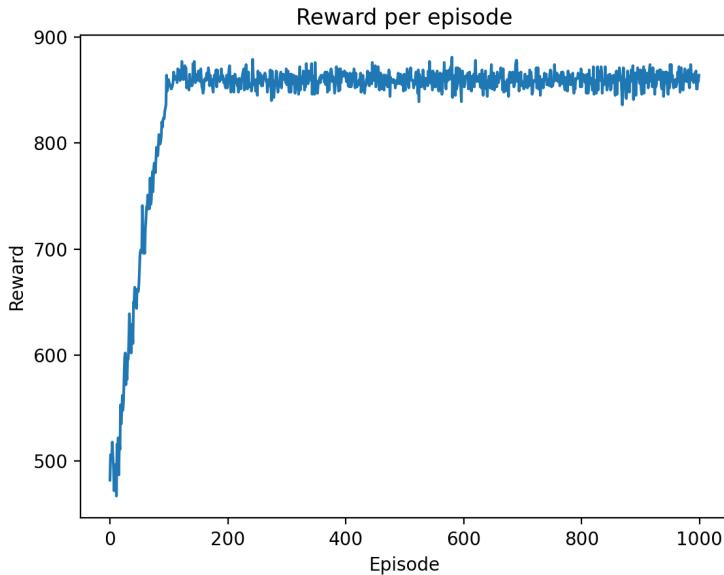


Figure 4.26: Reward-ul per episod obținut de expertul ce controlează iterarea

## 4.5 API

Accesarea și folosirea modelului descris mai sus va fi realizata folosind un API găzduit local. Acesta va pune la dispoziție următoarele funcționalități: listarea modelelor existente, crearea și antrenarea unui nou model, folosirea unui model antrenat deja existent, ștergerea unui model, încărcarea unui set de date pentru imitație, cererea creării unei structuri imitative, preluarea rezultatelor imitației.

Dat fiind specificul lucrării nu consideram ca acest API va trebui să fie accesibil printr-o aplicație online, deoarece cazurile de folosire ale acestuia sunt majoritar fie sub forma unei unelte în dezvoltarea de jocuri, fie sub o formă similară pentru definirea comportamentelor în robotica, și nu sub formă unei aplicații comerciale. Am ales totuși să oferim serviciul sub formă unui API REST datorită flexibilității pe care o oferă în comunicarea cu alte aplicații ce rulează local.

Dorim să folosim aspectul local al aplicației în avantajul nostru, prin evitarea transmiterii de volume mari de date prin cereri HTTP, folosind în schimb fișiere pentru stocarea seturilor de date, și comunicând doar calea de acces a acestora în conținutul cererilor HTTP.

#### 4.5.1 Endpoints

În continuare vom descrie endpoint-urile și modul de folosire a acestora. Primul set de endpoint-uri este responsabil de management-ul modelelor. Acesta expune 4 metode, pentru a afișa unul sau toate modelele, pentru crearea de noi modele, respectiv pentru ștergerea modelelor. Calea de acces pentru aceste endpoint-uri este "/**models**", metodele POST, DELETE, și GET (all) fiind accesate în acest mod, endpoint-ul pentru afișarea unui singur model fiind accesat prin "/**models/<modelName>?modelType=<modelType>**". Tipul modelului specifică pentru care dintre experți dorim modelul, numele modelului specificând care dintre modele dorim. Dacă tipul modelului nu este specificat nici un model nu va fi returnat, ci request-ul va primi codul 404.

Endpoint-ul **Get Model** este intenționat pentru a verifica dacă antrenarea unui model este finalizată după folosirea endpoint-ului **Post Model**, iar endpoint-ul **Get Models** afișează toate modelele existente.

```
1 {  
2     "modelName": "example_Model",  
3     "modelType": "primitive"  
4 }
```

Listing 4.1: "POST model body"

Endpoint-ul **Post Model** necesită specificarea în body a unui modelName respectiv modelType (vezi 4.5.1). Dacă un model cu tipul și numele specificate există vom returna un cod de eroare și nu vom crea un nou model. Dacă datele sunt valide vom returna un mesaj pentru a anunța utilizatorul ca antrenarea a început, după care vom începe antrenarea.

Deoarece numele unui model nu îl identifică unic metoda DELETE are și ea nevoie de modelName și modelType. Dacă un model cu numele și tipul specificat există acesta va fi șters, altfel vom returna codul de eroare 404, menținând ca modelul nu a fost găsit. Deoarece antrenarea unui model este foarte costisitoare, nu vom șterge modelul în sine ci doar referința la acesta în server, acesta putând încă fi accesat din nou dacă fișierul repository este șters, acesta regenerându-se și citind toate modele existente în directorul în care salvăm noi modele. Dacă ștergerea definitiva a modelului este dorita aceasta operație se poate face ștergând fișierul modelului după realizare unei cereri de ștergere.

Următorul set de endpoint-uri este cel responsabil de cererile de predicție. Deoarece acestea pot dura o perioadă mai mare am decis să cream un request handle pe care il vom actualiza cu progresul cererii, acesta fiind șters când utilizatorul cere rezultatul unei cereri finalizate. În acest set avem următoarele metode GET(all), GET, și POST. O metoda DELETE nu este necesară deoarece odată ce este apelata metoda GET pe o cerere, dacă aceasta este finalizată vom returna calea spre fișierul cu rezultatele, și vom șterge cererea din server. Metodele POST și GET(all) sunt utilizate folosind calea de acces "/**requests**", respectiv "/**requests/<requestName>**" pentru GET.

```

1  {
2      "requestName": "Example_Request",
3      "models": [
4          {"modelName": "model_primitive", "modelType": "primitive"},
5          {"modelName": "model_position", "modelType": "position"},
6          {"modelName": "model_iteration", "modelType": "iteration"}
7      ],
8      "observations" : "D:\MyApp\observationData.json"
9 }
```

Listing 4.2: "POST prediction request body"

O cerere de POST are nevoie de un requestName, prin care putem identifica cererea, de o lista cu exact 3 modele, unul din fiecare tip: "**primitive**", "**position**", respectiv "**iteration**" și observations, un string ce indică calea de acces a fișierului cu datele de observație (vezi 4.5.1). Dacă requestName-ul este deja folosit, unul dintre modele nu există vom returna un cod de eroare, iar dacă toate datele sunt valide vom returna un mesaj pentru a însăși utilizatorul ca cererea a fost înregistrată, un handle va fi creat pentru cerere, și modelele vor fi folosite pentru a crea o structură de noduri ce imita datele din fișierul primit. Formatul fișierului este unul de tip JSON ce conține o listă de observații a stării modelului, conținând rotația fiecărei articulații. Din păcate nu putem verifica în timp destul de scurt dacă structura fișierului de input este invalidă, deoarece am putea avea fișiere cu mii de cadre de observații. Într-un astfel de caz în procesul de antrenare va apărea o eroare, iar cererea va rămâne nefinalizată.

Ultimul set de endpoint-uri au un rol administrativ, unul fiind intenționat pentru a veri-

fica starea server-ului "online" sau "offline" accesibil folosind metoda HEAD și calea de acces `/status` iar celălalt fiind folosit pentru a opri serverul folosind metoda GET pe calea de acces `/shutdown`. Acest endpoint este necesar pentru a opri serverul din extensia Unity a cărei implementare este descrisă în capitolul 5. Endpoint-ul de verificare a statusului este de asemenea folosit pentru a face mai ușoara folosirea serverului împreună cu extensia creata în Unity.

#### 4.5.2 Detalii de implementare

Implementarea a fost realizata în Python, un limbaj foarte flexibil, preferat în domeniul inteligenței artificiale. Agenții au fost implementați folosind framework-ul **keras-rl2** baza pe **keras**, împreună cu **gym** de la OpenAI, care a fost folosit pentru crearea environment-urilor necesare antrenării modelelor. Serverul a fost implementat folosind **Flask**, un framework pentru crearea de servere simple, acesta a fost ales deoarece oferă toate facilitățile de care am avut nevoie pentru realizarea acestui API.

Deoarece unele cererile făcute necesita o perioada lungă de execuție am decis să returnam un mesaj de succes sau eroare utilizatorului iar în paralel să începem realizarea cererilor. Pentru acest scop am folosit clasa **Thread** din Python, deoarece un thread în Python nu necesita un join, execuția thread-urilor încheindu-se odată cu terminarea metodelor apelate în "run". Acest comportament a făcut trivială obținerea comportamentului dorit. Thread-urile sunt folosite pentru efectuarea cererilor de antrenare și predicție, aceste fiind cele mai costisitoare operații pe care le oferă serverul.

#### 4.5.3 Testarea API-ului

Testarea API-ului a fost realizata în principal manual, testarea automata fiind incompatibila în multe cazuri pentru API-ul nostru. Testarea componentelor excluzând componenta inteligentă a fost realizata folosind atât teste automate cât și testare manuală.

Clasa de repository a fost testata unitar, integrarea acesteia cu serverul fiind testata manual. Testarea prin integrare a endpoint-urilor a fost realizata manual folosind un client REST, și înlocuind comportamentul componentei inteligente cu log-uri în consola în cazul cererilor de antrenare pentru a verifica funcționarea corecta a acestora. Pentru a simula o astfel de cerere

am creat o sarcina artificială, și am verificat primirea unui răspuns de la server fără a fi nevoie de a aștepta timpul real necesar completării sarcinii. Cererile de predicție au fost testate folosind componenta inteligentă, iar în cele din urmă comportamentul componentei inteligente a fost testat manual atât pentru antrenare cât și pentru predicție.

# Chapter 5

## Aplicatie

Aplicația propusa pentru prezentarea metodei noastre este o extensie a game engine-ului Unity ce permite unui utilizator sa creeze o animație bazata pe *key-frames* (rotații ale articulațiilor specificate absolut, prin care modelul tranziționează interpolând liniar intre pași) care va fi observata și imitata folosind metoda propusa. De asemenea extensia va putea fi folosita pentru selectarea modelului ce va fi folosit în imitație, încărcarea datelor observe pentru imitare și folosirea structurii imitative rezultate [17].

Motivația pentru crearea acestei extensii este de a prezenta un caz de folosire al algoritmului nostru, crearea unei unelte ce permite imitarea comportamentelor în procesul de dezvoltare a unui joc sau a unui alt produs software. Algoritmul nostru este integrat cu Unity printr-o extensie a editorului. Am ales sa folosim game engine-ul Unity deoarece este unul dintre cele mai populare pe piață, acesta fiind de asemenea foarte extensibil, permitând crearea de plug-in-uri folosind C#, același limbaj care este folosit și pentru a scrie codul pentru aplicații.

Extensia va folosi API-ul descris în secțiunea 4.5 pentru a accesa interacționa cu componenta inteligenta dezvoltata.

### 5.1 Arhitectura

In dezvoltarea extensiei vom avea în vedere folosirea unei arhitecturi software stratificate. Deoarece Unity implementează layer-ele de view și repository (în mare măsura pentru scopurile noastre), ne vom concentra doar pe straturile de UI controller și service. Serverul va avea de asemenea o arhitectura stratificata cu 2 straturi 5.1.

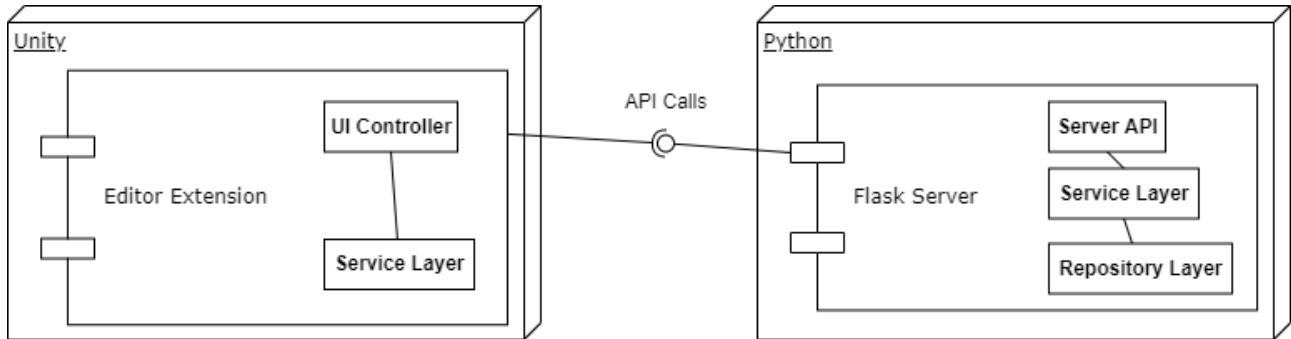


Figure 5.1: Diagrama de componente a aplicației

Aplicația este compusă din 2 componente, serverul ce găzduiește componenta inteligentă și clientul, sub forma unei extensii a game engine-ului Unity.

### 5.1.1 Cazuri de utilizare

In faza de proiectare a aplicație am identificat mai multe cazuri de utilizarea a acesteia (vezi figura 5.2)

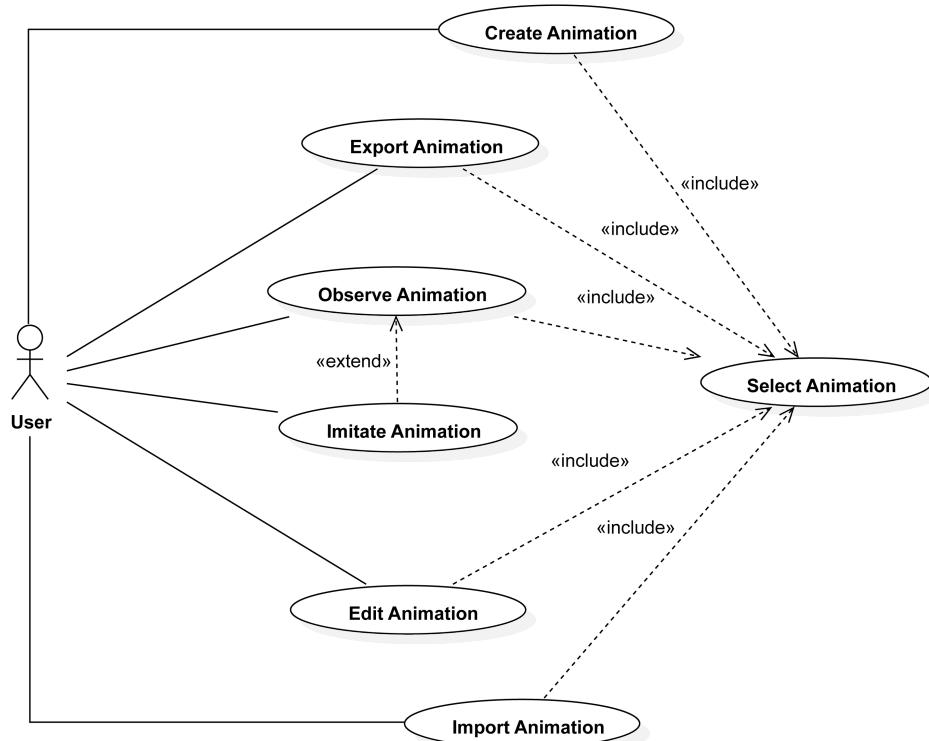


Figure 5.2: Cazurile de utilizare identificate pentru aplicație

Cele mai importante cazuri de utilizare a aplicației sunt crearea și editarea de animații baza

pe key-frame-uri, și observarea și imitarea unui model ce executa o animație selectata. Alte cazuri de utilizare sunt exportarea și importarea de animații, aceste doua cazuri de utilizare fiind utile deoarece Unity folosește un sistem propriu de asset-uri și serializare a acestora, datele unei animații nefiind accesibile în formatul stocat de game engine.

Cazul de imitare a unei animații este o extensie a observării, acesta din urma fiind executat în Play Mode în editor. Observarea unei animații salvează datele observate într-un fișier JSON care va fi folosit în imitarea animației.

Exportarea folosește animația selectata curent în fereastra de editor creata de noi și, folosind un serializer simplu, extrage datele fiecărui key-frame și le combina într-un singur fișier JSON.

Importarea unei animații nu folosește un serializer personalizat, și folosind obiecte DTO transformam datele din format JSON în obiecte intermediare care sunt transmise unui service care creează asset-uri Unity.

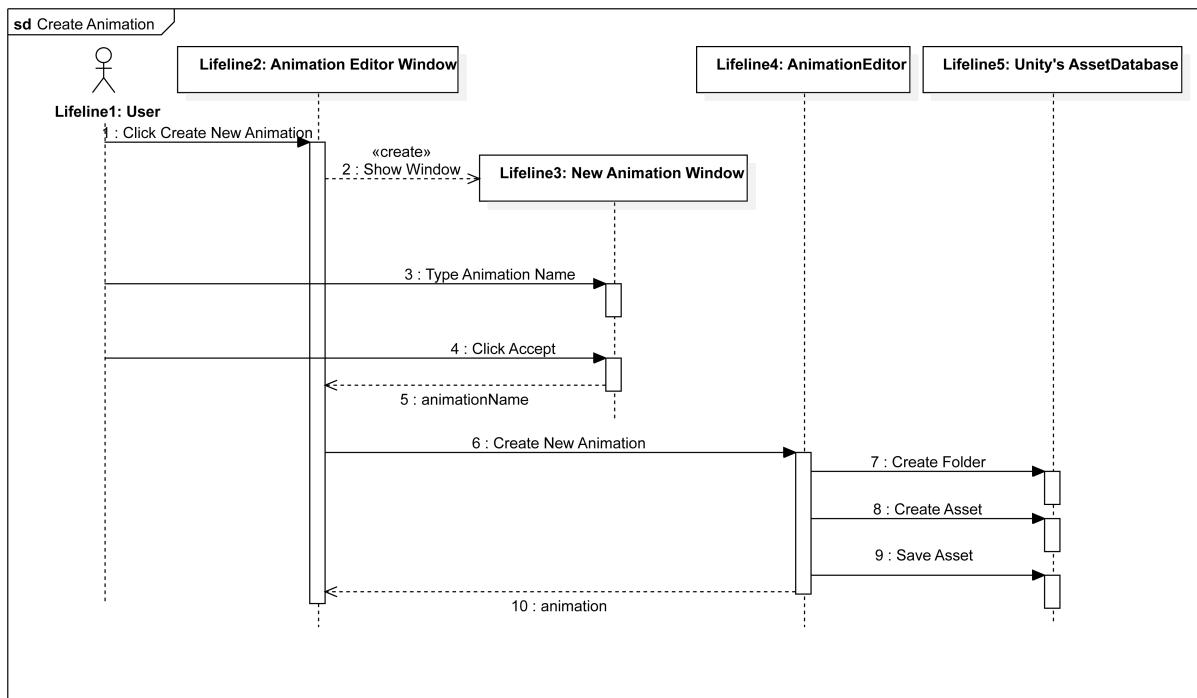


Figure 5.3: Diagrama de secvență pentru crearea unei noi animații

Crearea unei noi animații este un use case disponibil doar dacă fereastra de editor nu are nici o animație existentă selectată. Aceasta urmează diagrama de secvență 5.3. Primul pas în acest use case este interacțiunea utilizatorului cu fereastra editorului, care conduce la crearea unei noi ferestre modale în care actorul introduce numele noii animații. Dacă utilizatorul închide

fereastra cu un răspuns pozitiv procesul de creare a animației continua, service AnimationEditor fiind responsabil de interacționarea cu AssetDatabase-ul Unity care este responsabil de interacțiunea cu fișierele proiectului.

La crearea unei animații vom crea un folder cu același nume, în care cream o animație și un key-frame pe care îl asociem acesteia.

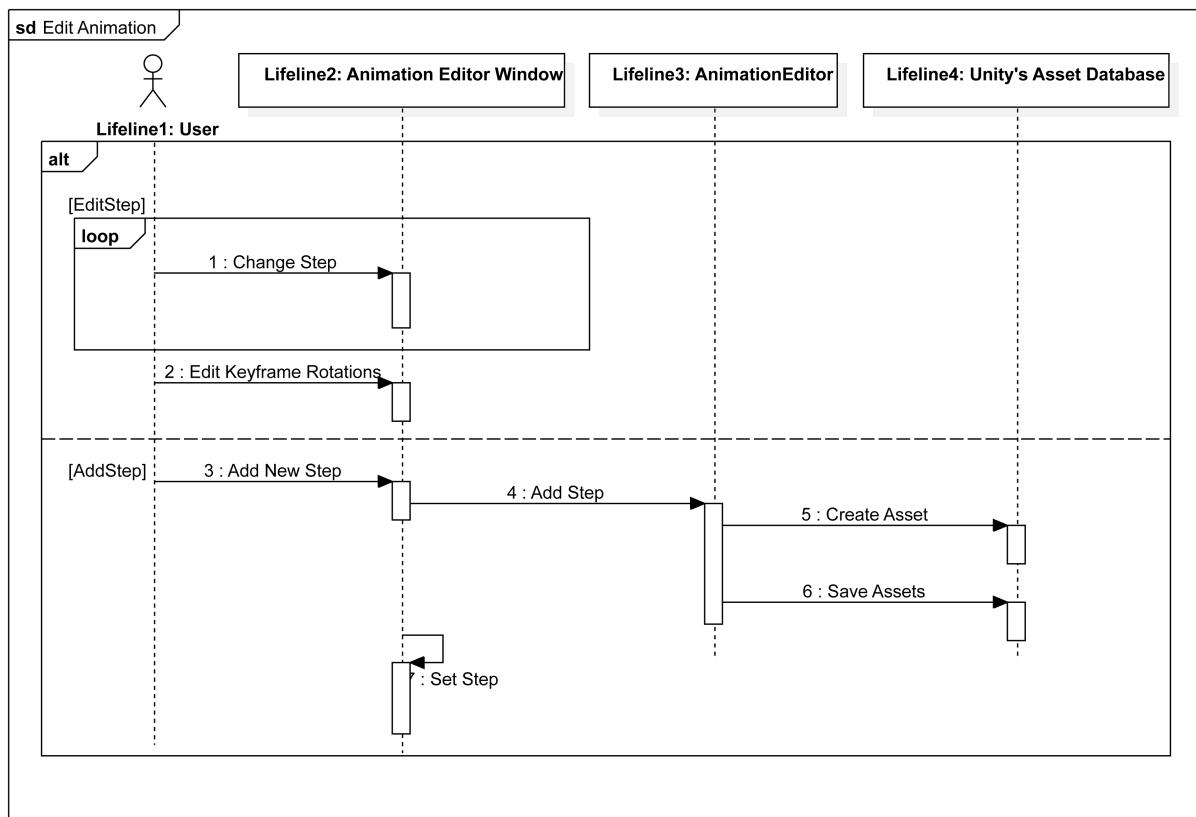


Figure 5.4: Diagrama de secvență pentru editarea unei animații existente

Editarea unei animații poate fi făcuta în două moduri diferite, schimbarea unui key-frame sau adăugarea unui key-frame în animație (vezi figura 5.4). Fereastra editorului afișează un key-frame al animației, utilizatorul putând cicla prin key-frame-urile existente pentru a ajunge la key-frame-ul pe care dorește să îl schimbe. Pentru realizarea acestei operații este suficientă implicarea ferestrei de editor.

Adăugarea unui nou key-frame necesita din nou implicarea service-ului Animation Editor de către fereastra, acesta interacționând cu AssetDatabase-ul Unity pentru crearea asset-urilor. După adăugare fereastra afișează key-frame-ul nou creat.

Imitarea unei animații observate necesită crearea și trimiterea unei cereri de predicție, și

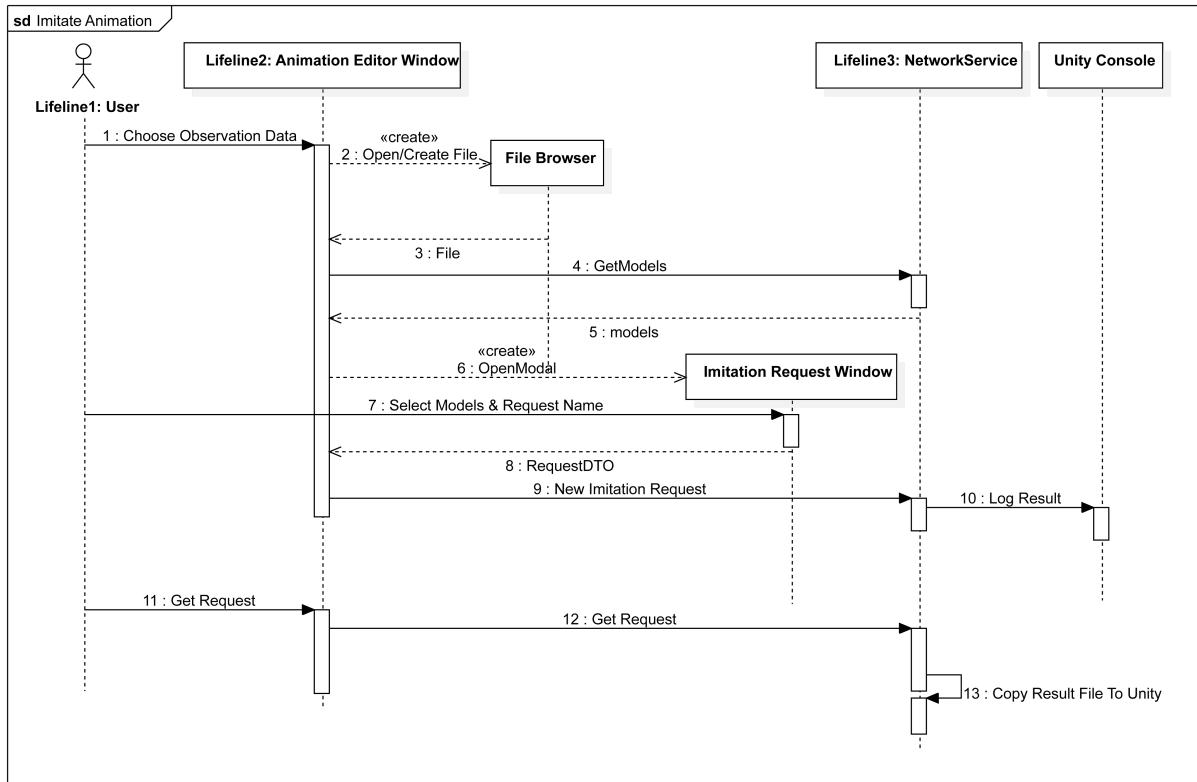


Figure 5.5: Diagrama de secvență pentru imitarea unei animații observate

obținerea rezultatelor printr-o cerere separată (vezi figura 5.5). Pentru a crea o cerere de predicție utilizatorul trebuie să aleagă un fișier ce conține date de observație. După care o nouă fereastră este creata, căreia ii este transmisa o lista de modele pe care fereastra principală le-a obținut de la service-ul NetworkService. În noua fereastră utilizatorul trebuie să aleagă numele cererii, și cate un model din fiecare tip ce vor fi folosite pentru predicție. Fereastra întoarce un request care este transmis de fereastra principală NetworkService-ului. Pentru a obține rezultatele predicției o nouă cerere este necesara, fereastra principală transmînd NetworkService-ului numele cererii pe care o dorim. Dacă cererea a fost finalizată vom primi calea de acces a unui fișier ce conține structura imitativa, pe care îl vom copia în structura de fișiere a proiectului Unity.

### 5.1.2 Design Patterns

Pe parcursul dezvoltării aplicației și serverului am întâlnit probleme a căror soluții sunt cel mai bine rezolvate de design-pattern-uri cunoscute precum Observer, Singleton, Adapter, Proxy și Memento.

### 5.1.2.1 Observer

Am folosit o versiune a *design pattern*-ului Observer în implementarea mecanismului de observare a unui subiect din scena. Am realizat acest pattern folosind funcționalitatea de delegare a limbajului C#, subiectul de observație expunând un delegate, și funcțiile de adăugare și ștergere a delegaților. Acești delegați vor fi apelați, asemeni cu event-ul pattern-ului clasic Observer. Clasa observatoare implementează o astfel de funcție pe care o adaugă în lista delegaților, urmând să fie apelata la sfârșitul unei secvențe de animație, pentru a semnala sfârșitul unei secțiuni de observare, și facilită procesul de imitare folosind datele colectate.

### 5.1.2.2 Singleton

Am folosit *design pattern*-ul Singleton în implementarea clasei **NetworkService**. Folosirea acestui design pattern a fost utilă deoarece astfel putem grupa toate apelurile HTTP în aceasta singura clasa, și instanta fiind unică, evităm crearea mai multor clienți HTTP, limitându-ne la unul singur pentru clasa singleton.

### 5.1.2.3 Memento/Proxy

Este necesară transmiterea de informații între clasele de extensie a editorului Unity și clase ce extind **MonoBehaviour** o clasa ce permite atașarea unui obiect de un **GameObject** în scena. Life-cycle-ul unui Editor Window nu permite comunicarea directă cu un **MonoBehaviour**. Aceasta problema apare fiindcă starea unui Editor Window este resetată la intrarea în Play Mode. De asemenea folosirea unei clase statice sau a unui singleton nu este viabilă deoarece toate instancele respectiv modificările aduse atributelor statice sunt resetate deoarece toate scripturile sunt resetate înainte de a intra în Play Mode. Soluția la acesta problema este crearea unei clase **EditorProxy** care are doar metode statice care primesc un atribut al clasei **AnimationEditorWindow** și o salvează într-un fișier. Apoi, când un **MonoBehaviour** dorește să citească aceste date o metoda statică va citi din fișier aceste date. Deși aceasta clasa nu este pur un Adapter sau un Proxy, problema pe care o rezolvă are aspecte similare cu problemele rezolvate de ambele design pattern-uri.

### 5.1.2.4 Adapter

Datele de observație primite de la client pentru realizarea unei predicții sunt salvate în formatul JSON, care este foarte util pentru stocare lor, dar este nepotrivit pentru folosirea acestora în timpul predicției. O clasa de adaptează aceste date din formatul primit într-un format potrivit pentru predicție, transformând dicționarele cu perechi cheie valoare în liste care păstrează doar valorile. Pentru transmiterea predicției, datele sunt din nou adaptate din acest format în JSON și salvate în fișier urmând ca path-ul fișierului să fie transmis clientului.

## 5.2 Implementarea

Luând arhitectura abstractă descrisă mai sus am început implementarea prin crearea unor clase și pachete importante (vezi figura 5.6). În diagrama de clase am definit relațiile între clase din diferite pachete și relațiile claselor ce aparțin aceluiași pachet.

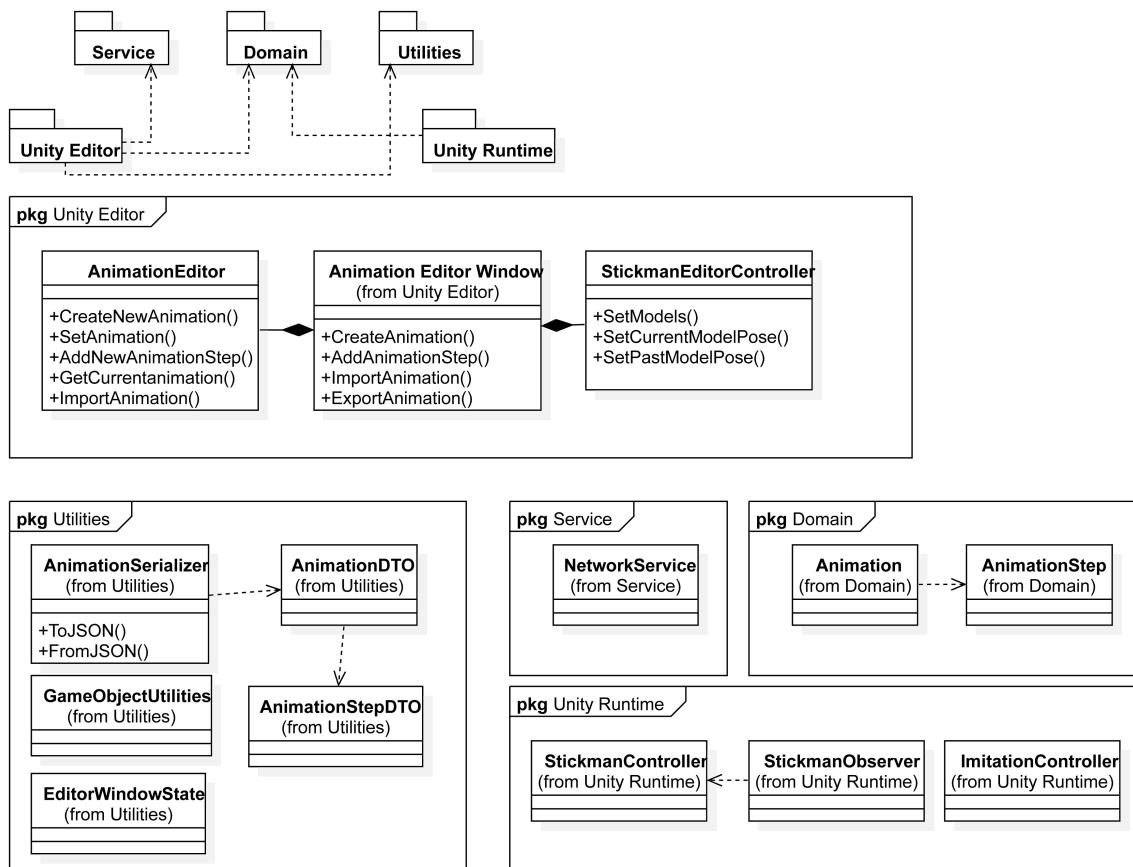


Figure 5.6: Diagrama claselor pentru extensia dezvoltată

Pachetul Domain conține clasele ce descriu entitățile pe care le folosim în extensie, Animation și AnimationStep. Clasa Animation conținând o lista de AnimationSteps care descriu un key-frame al animației. Pachetul Service este compus din clasa NetworkService, responsabilă de comunicare cu serverul și pornirea acestuia. Pachetul Utilities conține diverse clase utilitare folosite de alte clase, precum AnimationSerializer, GameObjectUtilities care oferă funcții pentru serializarea animațiilor respectiv pentru căutare de GameObjects în scena.

Pachetul UnityRuntime conține clase responsabile pentru comportamentul GameObject-urilor în scena când editorul se află în Play Mode. StickmanController este clasa responsabilă de redarea animațiilor create, StickmanObserver colectează date de rotație ale unui model observat, și ImitationController redă animația obținuta prin imitarea mișcării unui model.

Pachetul UnityEditor conține clasele ce extind comportamentul editorului. AnimationEditorWindow creează și controlează un nou tip de fereastră în editor, AnimationEditor fiind clasa care interacționează cu clasa AssetDatabase oferita de Unity pentru managementul asset-urilor pentru proiect. La nivel arhitectural aceasta clasa se încadrează în layer-ul de service, dar a fost inclusă în acest pachet din deoarece clasa depinde de namespace-ul UnityEditor din Unity (a nu se confunda cu pachetul UnityEditor). StickmanEditorController are o funcție similară clasei StickmanController, aceasta oferind funcționalitatea de afișare a modelului în poziția dictată de key-frame-ul curent pentru a ușura editarea animațiilor.

### 5.2.1 Technology Stack

In dezvoltarea aplicației și a serverului am folosit Unity și C# respectiv Python, Keras-RL2, Gym și Flask.

Pe partea de server am ales să folosim Python datorita facilităților oferite pentru AI, pachetele Keras-RL2 și Gym fiind folosite în scrierea procedurilor de antrenare și a environment-urilor pentru agentii de Reinforcement Learning. Flask a fost ales ca framework pentru server deoarece oferă toate funcționalitățile necesare fără nevoie de a scrie mult boiler-plate.

Pentru dezvoltarea aplicației am ales un game engine deoarece unul dintre cazurile de utilizare a algoritmului nostru este folosirea acestuia ca o unealta de dezvoltare pentru jocuri. Am ales Unity specific atât datorita familiarității cu game engine-ul cat și datorita facilităților oferite de acesta, precum și documentația clara atât pentru scrierea codului pentru joc cat și pentru extinderea editorului.

### 5.2.2 Testare

Deoarece multe din componentele principale nu se pretează testării automate, deoarece acestea depind de Unity, testarea extensiei a fost realizata manual.

De exemplu pentru clasa AnimationEditor nu putem testa adăugarea corecta a unei animații deoarece unele din funcțiile clasei AssetDatabase nu returnează coduri de eroare în cazul unor probleme cu datele de intrare transmise acestora. Testarea scripturilor ce rulează în Play Mode este de asemenea dificila, fiind greu de stabilit rezultatul corect, de exemplu pentru valoarea rotației unei articulații după un anumit număr de cadre.

Așadar am testat toate funcționalitățile oferite de extensie manual: crearea și editarea de animații, importarea și exportarea acestora, observarea și redarea de animații, și pornirea/-comunicarea cu serverul. Pentru a verifica comportarea corecta a componentelor am folosit logurile afișate în consola Unity.

### 5.2.3 Detalii de implementare

Deoarece aplicația noastră este o extensie a editorului Unity, și server-ul cu care comunicam este găzduit local am considerat utilă oferirea posibilității de a porni serverul direct din extensie. Pentru aceasta am creat un nou proces în C# care executa un batch script ce pornește serverul. Oprirea serverului însă nu a putut fi realizata prin oprirea procesului deoarece serverul devine un proces separat, care nu poate fi oprit astfel. Deoarece Unity suportă un standard vechi de C# oprirea unui arbore de procese nu este suportată, soluțiile ramase fiind oprirea procesului după nume sau trimiterea unei cereri HTTP de oprire. Deoarece oprirea serverului folosind comanda kill ar opri orice alt server Flask care rulează pe calculator am ales să folosim o cerere HTTP.

O problema întâlnita în implementarea clasei StickmanController a fost redarea animațiilor ce modifica valoarea rotațiilor în afara intervalului de valori 0-360. Pentru a remedia aceasta problema am convertit una dintre valorile rotației curente sau rotației întâi la începutul tranziției între două key-frame-uri pentru a obține rotația aplicată la fiecare cadru, și convertind valorile ce ies din interval.

O alta problemă legată de redarea animațiilor cauza citirea incorecta a valorilor de rotație în unele cazuri. Aceasta defectiune se datoră faptului că Unity folosește cuaternioni pentru

rotații, acești cuaternioni putând fi reprezentați în mai multe moduri în rotații pe 3 axe. Astfel în cazuri limita, atingerea unei rotații cauzează algoritmul de animație să fie blocat într-un ciclu. Am evitat aceasta problema citind și păstrând o copie inițială a rotațiilor pe care o actualizam în paralel cu actualizarea rotațiilor modelului folosind componenta folosită de Unity.

O alta particularitate a Unity este faptul că toate scripturile sunt reîncărcate la intrarea în Play Mode, acest aspect făcând dificila comunicare între scripturi de editor și scripturi din scena care sunt folosite în Play Mode. Pentru a rezolva aceasta problema am creat clasa `EditorProxy` despre care am vorbit în secțiunea de Design Patterns. Aceasta clasa citește și salvează în fișiere datele care trebuie comunicate.

#### 5.2.4 Persistența datelor

Persistența datelor este realizată folosind diferite metode, pentru datele de observație generate, vom folosi fișiere, în care vom salva datele în format JSON, pentru compatibilitate ușoara cu serverul. Structura de control pe care o vom primi de la server va fi și ea citită din fișiere JSON. Animațiile bazate pe key-frame-uri vor utiliza sistemul de Asset Management din Unity pentru persistența, cu opțiunea de a exporta/importa animații format JSON, pentru a facilita folosirea aplicației.

Datorită specificului aplicației noastre o baza de date nu ar aduce beneficii, overhead-ul necesar fiind costisitor în situația actuală.

#### 5.2.5 UX & Manualul de utilizare

Deoarece am implementat o extensie a game engine-ului Unity am dorit să oferim o experiență mai nativă pentru utilizatorul extensiei, care să fie intuitivă. Datorită uneltele oferite pentru crearea extensiilor acest obiectiv a fost ușor de atins, componentele vizuale ale extensiei folosind aceleași elemente de UI pe care le folosesc ferestrele obișnuite din editor.

Pentru accesarea cazurilor de utilizare asociate cu observarea și imitația avem tabul `Imitation` din fereastra extensiei (vezi figura 5.7). Aceasta conține controale pentru managementul stării de funcționare a serverului. Avem de asemenea posibilitatea de a alege fișierul cu date de observație sau de a crea unul nou, și de a crea o nouă cerere de imitație. Pentru a crea o astfel de cerere am creat o fereastra modală în care utilizatorul va alege numele și modelele care vor

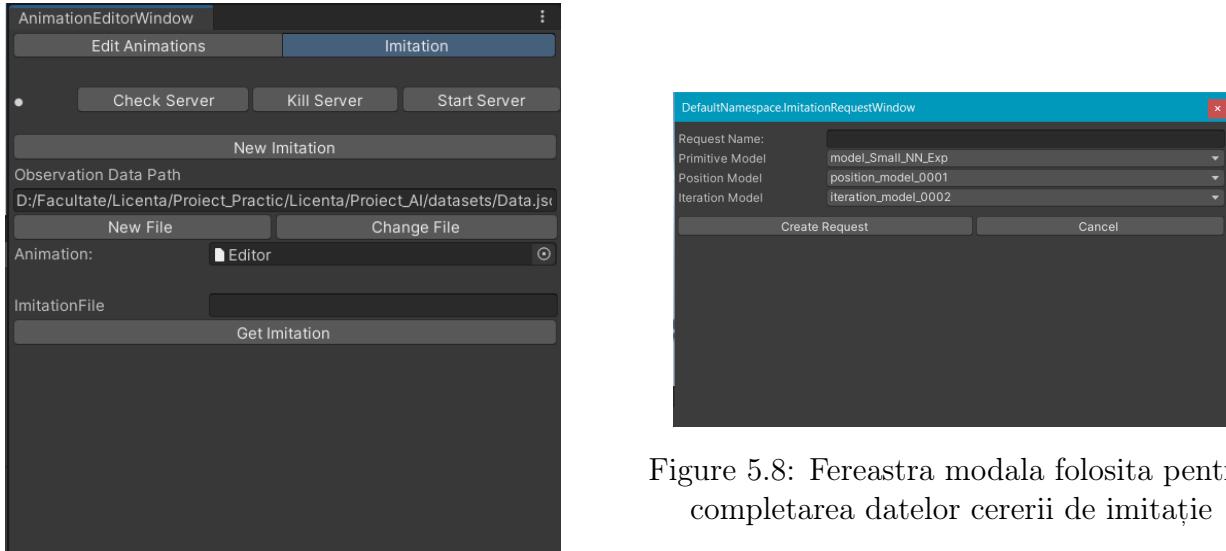


Figure 5.7: Secțiunea de imitație a ferestrei

Figure 5.8: Fereastra modală folosită pentru completarea datelor cererii de imitație

fi folosite (vezi figura 5.8).

Pentru a obține rezultatele unei cereri avem butonul Get Imitation în josul ferestrei, câmpul fișierului de imitație fiind folosit pentru a transmite unui script din scena calea de acces a datelor pentru a reda animația obținuta prin imitare.

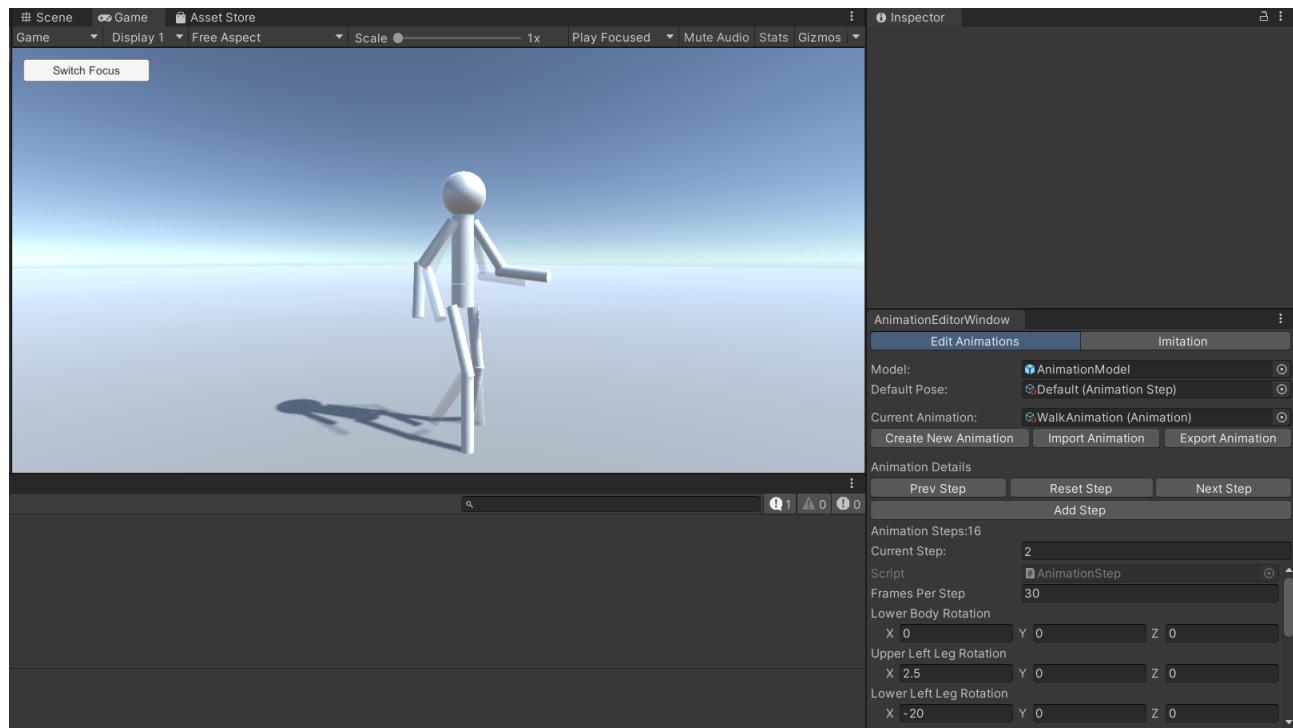


Figure 5.9: Scene View-ul și fereastra editorului nostru, în secțiunea de editarea a animațiilor

Secțiunea de editarea și creare a animațiilor este și ea creată într-un mod intuitiv. În figura

5.9 putem vedea în stânga fereastra Game View și în stânga fereastra extensiei noastre. Pentru a facilita crearea de animații un model poate fi selectat sau creat pentru a reda pozițiile date de key-frame-ul curent respective cel anterior (dacă există). Modelul solid reprezintă poziția curentă, cel transparent prezentând poziția data de key-frame-ul anterior.

Fereastra permite importarea și exportarea de aplicații, precum și crearea unei noi animații. În josul ferestrei, dacă o animație este selectată avem butoane respectiv un câmp de text pentru navigarea prin key-frame-uri și un mini-inspector pentru editarea valorilor unui key-frame. Fereastra permite și adăugarea de noi key-frame-uri în animații.

Unele cazuri de utilizare sunt ascunse dacă condițiile de accesare a acestora nu sunt îndeplinite, de exemplu creare unui model nu este prezentată dacă un model este deja selectat. Acest comportament a fost dorit pentru a reduce numărul de elemente inutile care ii sunt afișate utilizatorului, și pentru a îl opri din a utiliza în mod greșit unele funcționalități.

# Chapter 6

## Concluzii și direcții viitoare

Am obținut o serie de experți capabili să imite un comportament observat printr-o metoda nouă bazată pe crearea unei structuri de control formata din acțiuni primitive. Antrenarea experților a fost dificila, datorita datelor de intrare, fiind necesar un pas de preprocesare neobișnuit care alterează dispersia datelor pentru a face mai ușoara separarea și învățarea acțiunilor ce reprezinta diferite primitive/pozitii de editare.

Deși rezultatele obtinute de model nu sunt suficient de bune pentru a spune ca am obținut un mecanism de imitare capabil, rezultatele noastre sunt totuși încurajatoare, sugerând posibilitatea de a obține rezultate mai bune prin modificări ulterioare procesului nostru de antrenare. Dat fiind faptul ca metoda propusa este una neexplorata în forma pe care am propus-o consideram ca aceasta lucrarea servește ca un *proof of concept* pentru crearea de structuri imitative folosind AI. Desigur îmbunătățiri majore sunt necesare pentru a obține o performanta apropiata de metodele deja existente.

Noi reprezentări ale nodurilor structurii de control vor fi necesare pentru a încerca folosirea acestei metode cu primitive de acțiuni mai abstracte, metoda propusa de noi fiind utilă în cazul imitării comportamentului motor, dar nu consideram ca este fezabilă pentru alte aplicații. De asemenea, pentru a obține structuri de control ce iau în considerare contextul extern al subiectului imitat o metoda de creare a ramurilor în behaviour tree va fi necesara. Aceasta problema este posibil mult mai dificila decât problema propusa în aceasta lucrare.

Testarea acestei metode a fost limitata în primul rând de lungimea perioadei de antrenare și de generarea datelor necesare. Deoarece nu am avut la dispoziție un calculator/server dedicat pentru antrenare nu am reușit să antrenam pe o perioadă mai îndelungată modelele. De

asemenea, implementarea actuala depinde mult de CPU și, în consecință nu beneficiază mult de accelerările oferite de o implementare ce poate folosi GPU-ul. Presupunem ca modele de dimensiuni mai mari ar avea nevoie de perioade mult mai lungi de antrenare pentru a spera la predicții mai bune decât cele obținute de modele simple pe care le-am antrenat.

O alta problema întâmpinată a fost cea a datelor necesare antrenării. Acestea au fost generate procedural, deoarece cantitatea de date necesara antrenării unui agent prin Reinforcement Learning este foarte mare, nefiind practică crearea unor animații pentru extragerea unui set de date. De asemenea antrenarea primilor doi experți a necesitat balansare prin metodele descrise datorita reprezentării nebalansate a acțiunilor, vasta majoritate a stărilor posibile fiind non-optime.

Aplicația și serverul dezvoltat ar putea fi de asemenea îmbunătățite. O prima limitare a acestora este folosirea unei singure configurații ale modelului imitat. O iterare ulterioară ar putea schimba implementarea acestora pentru a permite schimbarea subiectului imitat.

# Bibliography

- [1] Ricardo Aler, Jose M. Valls, David Camacho, and Alberto Lopez. Programming robosoccer agents by modeling human behavior. *Expert Systems with Applications*, 36(2, Part 1):1850–1859, 2009.
- [2] Preben Fihl, Michael B Holte, Thomas B Moeslund, and Lars Reng. Action recognition using motion primitives and probabilistic edit distance. In *International Conference on Articulated Motion and Deformable Objects*, pages 375–384. Springer, 2006.
- [3] Jürgen Fritsch. Modular neural networks for speech recognition. Technical report, Carnegie Mellon University Pittsburgh, PA, Department of Computer Science, 1996.
- [4] Bernd Fritzke. A growing neural gas network learns topologies. *Advances in neural information processing systems*, 7, 1994.
- [5] Zoubin Ghahramani. Building blocks of movement. *Nature*, 407(6805):682–683, 2000.
- [6] Bernard Gorman and Mark Humphrys. Towards integrated imitation of strategic planning and motion modeling in interactive computer games. *Computers in Entertainment (CIE)*, 4(4):10–es, 2006.
- [7] Bernard Gorman and Mark Humphrys. Imitative learning of combat behaviours in first-person computer games. *Proceedings of CGAMES*, pages 85–90, 2007.
- [8] Robert A Jacobs, Michael I Jordan, and Andrew G Barto. Task decomposition through competition in a modular connectionist architecture: The what and where vision tasks. *Cognitive science*, 15(2):219–250, 1991.
- [9] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

- [10] Thomas Martinetz and K. Schulten. A "neural-gas" network learns topologies. *Artificial neural networks*, 1:397–402, 01 1991.
- [11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.
- [12] Thomas B. Moeslund, Preben Fahl, and Michael Boelstoft Holte. Action recognition using motion primitives. In *The 15th Danish conference on pattern recognition and image analysis*, 2006.
- [13] Robert Nystrom. Game programming patterns. [Game Programming Patterns](#).
- [14] Helge J. Ritter, Thomas Martinetz, and Klaus Schulten. Neural computation and self-organizing maps - an introduction. In *Computation and neural systems series*, 1992.
- [15] Joe Saunders, Chrystopher Nehaniv, Kerstin Dautenhahn, and Aris Alissandrakis. Self-imitation and environmental scaffolding for robot teaching. *International Journal of Advanced Robotics Systems, Special Issue on Human - Robot Interaction*, 4:109–124, 03 2007.
- [16] Joe Saunders, Chrystopher L Nehaniv, and Kerstin Dautenhahn. Using self-imitation to direct learning. In *ROMAN 2006-The 15th IEEE International Symposium on Robot and Human Interactive Communication*, pages 244–250. IEEE, 2006.
- [17] Unity Technologies. Unity game engine. [Unity Technologies Website](#).
- [18] Fabien Tencé, Laurent Gaubert, Julien Soler, Pierre De Loor, and Cédric Buche. Stable growing neural gas: A topology learning algorithm based on player tracking in video games. *Applied Soft Computing*, 13(10):4174–4184, 2013.
- [19] Fabien Tencé. *Probabilistic Behaviour Model and Imitation Learning Algorithm for Believable Characters in Video Games*. PhD thesis, Université de Bretagne occidentale - Brest, 11 2011.
- [20] Fabien Tencé, Laurent Gaubert, Julien Soler, Pierre De Loor, and Cédric Buche. Chameleon: Online learning for believable behaviors based on humans imitation in computer games. *Computer Animation and Virtual Worlds (CAVW)*, 09 2013.

- [21] Christian Thurau. *Behavior acquisition in artificial agents*. PhD thesis, Bielefeld University, 2006.
- [22] Christian Thurau, Christian Bauckhage, and Gerhard Sagerer. Imitation learning at all levels of game-ai. In *Proceedings of the international conference on computer games, artificial intelligence, design and education*, volume 5, 2004.
- [23] Epic Games Unreal Engine. Behaviour tree. [Unreal Engine Documentation](#).
- [24] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.
- [25] Michael van Lent, William Fisher, and Michael Mancuso. An explainable artificial intelligence system for small-unit tactical behavior. In *AAAI*, 2004.
- [26] Michael van Lent and John E Laird. Learning procedural knowledge through observation. In *Proceedings of the 1st international conference on Knowledge capture*, pages 179–186, 2001.