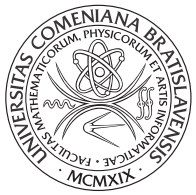


KATEDRA APLIKOVANEJ INFORMATIKY
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY
UNIVERZITA KOMENSKÉHO, BRATISLAVA

APROXIMÁCIA MOTORICKÉHO PRIESTORU RAMENA SIMULOVANÉHO ROBOTA

(Diplomová práca)

BC. RICHARD KORENČIAK

9.2.9 Aplikovaná informatika

Vedúci: doc. Ing. Igor Farkaš, PhD.

Bratislava, 2010

Čestne prehlasujem, že som túto diplomovú prácu vypracoval samostatne s použitím citovaných zdrojov.

. . . . . .

Poďakovanie

Svojmu školiteľovi doc. Ing. Igorovi Farkašovi, PhD. ďakujem za jeho čas, početné konzultácie a cenné rady počas tvorby mojej diplomovej práce. Taktiež ďakujem ľuďom v mojom okolí za drobnú pomoc a hlavne podporu.

Bibliografická identifikácia

KORENČIAK, Richard. *Aproximácia motorického priestoru ramena simulovaného robota* [Diplomová práca].
Univerzita Komenského v Bratislave. Fakulta matematiky, fyziky a informatiky; Katedra aplikovanej informatiky.
Školiteľ: doc. Ing. Igor Farkaš, PhD. Bratislava : FMFI UK, 2010. 71 s.

Abstrakt

Diplomová práca sa zaoberá úlohou dosahovania cieľových pozícií pomocou robotického ramena s kamerami v trojrozmernom pracovnom priestore. V prvej časti skúmame možné riešenie problému pomocou modelu rekurzívnej samoorganizujúcej sa mapy RecSOM, ktoré sa neskôr ukázalo ako nevhodné. Ako vhodný aparát ďalej skúmame model učenia s posilňovaním pre spojité priestory (CACLA).

Autori modelu CACLA skúmali jeho správanie na jednoduchších experimentoch, nám sa tento model podarilo natrénovať v úlohe dosahovania ľubovolnej cieľovej pozície s pomerne vysokou presnosťou z ľubovoľného počiatočného bodu v trojrozmernom priestore. Experimentálne sme overili konvergenciu modelu aj pri spojitých stavových a akčných priestoroch s rádovo vyššou dimenziou. Systematicky sme skúmali správanie modelu v závislosti od jeho parametrov, čím sme dokázali nájsť optimálny model pre daný problém.

Kľúčové slová

neurónová sieť, učenie s posilňovaním, ovládanie robotického ramena, vizuálno-motorická slučka

Bibliography identification

KORENČIAK, Richard. *Approximation of motor space of a simulated robot arm* [Master thesis].
Comenius University in Bratislava. Faculty of Mathematics, Physics and Informatics; Department of Applied Informatics.
Supervisor: doc. Ing. Igor Farkaš, PhD. Bratislava : FMFI UK, 2010. 71 p.

Abstract

The thesis deals with the task of reaching target positions using a robotic arm in a three dimensional space. First we investigate a potential solution using a recursive self-organizing map (RecSOM) that turned out not to be suitable for the task. Next, we propose another solution based on reinforcement learning model CACLA which operates on continuous state and action spaces.

The authors of CACLA algorithm tested it in simplified experiments (single target) but we managed to train this model in a task of reaching an arbitrary target with high accuracy while starting from an arbitrary position in three-dimensional working space. We experimentally verified convergence of this algorithm in a case continuous state and action spaces with considerably higher dimension. We systematically analyzed model behavior as a function of its parameters, which enabled us to find an optimal model for the given task.

Keywords

neural networks, reinforcement learning, robot arm movement control, trajectory generation, visuo-motor loop

Predhovor

Robotika je jednou z klasických oblastí uplatnenia umelej inteligencie. Ukazuje sa, že prístupy umelej inteligencie by mohli byť schopné riešiť aj požiadavku na istú autonómnosť riadených robotov.

Problém ovládania mechanických efektorov robota je možné riešiť niekoľkými prístupmi. Na jednej strane sú riešenia algoritmického charakteru, ktoré využívajú presný výpočet trajektórie pre presun daného efektora (ramena) do zvoleného bodu pomocou exaktných matematických rovníc. Tento prístup však vyžaduje vnesenie apriórnej znalosti do systému riadenia, pretože pre výpočet trajektórie je potrebné poznať kompletnú geometriu efektora, ako aj umiestnenie robotických kamier v pracovnom priestore.

Na strane druhej stoja riešenia založené na systémoch schopných učenia sa, ktoré je potrebné najprv vo fáze učenia natrénovať a ktoré po natrénovaní sú schopné autonómneho ovládania podľa naučených priestorových vzťahov. Nie je potrebné vkladať do nich explicitné pozície objektov v priestore a vzťahy. Medzi tieto riešenia patria napríklad riešenia pomocou neurónových sietí.

Inšpiráciou pre vznik témy tejto diplomovej práce bolo existujúce riešenie problému ovládania robotického ramena pomocou samoorganizujúcej sa siete SOM. Toto riešenie však nebolo schopné generovať celú trajektóriu, dokázalo iba predikovať uhly natočenia kĺbov ramena tak, aby sa dostalo do zvoleného bodu na jeden krok.

Generovanie celej trajektórie ako postupnosti elementárnych krokov považujeme však za rovnako dôležité. Predmetom diplomovej práce je navrhnúť, implementovať a analyzovať riešenie problému naučiť sa dosahovať cieľové pozície na príkladoch pomocou existujúcich modelov v umelej inteligencii. Predpokladáme, že model RecSOM by mohol byť použiteľný pre riešenie tohoto problému, preto bude primárnym modelom, pomocou ktorého sa pokúsime problém riešiť.

Obsah

Zoznam obrázkov

Kapitola 1

Úvod do problematiky a ciele práce

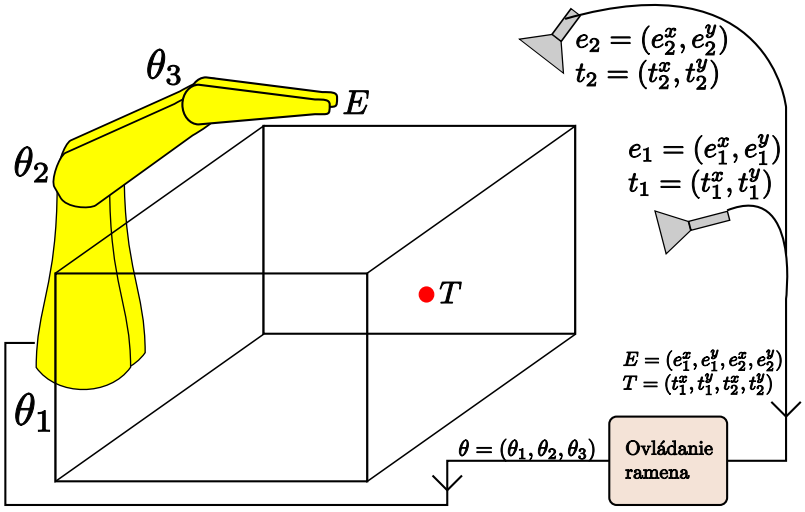
V úvodnej kapitole bude stručne popísaný problém, ktorý táto diplomová práca rieši. Bude predstavený model, ktorý už bol použitý na riešenie tohoto problému s návrhom ďalšieho možného rozšírenia ako motivácia k vzniku tejto diplomovej práce. V závere tejto kapitoly budú načrtnuté ciele diplomovej práce.

1.1 Popis pracovného priestoru

Pri simuláciách budeme uvažovať model pracovného priestoru tvorený robotickým ramenom, dvoma kamerami a jednotkou pre ovládanie ramena (obrázok 1.1). Budeme riešiť problém presunu robotického ramena do zvoleného cieľového bodu T.

Robotické rameno má tri stupne voľnosti – rotácia okolo základne (podstavy) a dva kĺby ramena. Rotácia okolo základne určuje rovinu kolmú na základňu a pomocou ďalších dvoch kĺbov je umožnený pohyb v rámci tejto roviny. Tento mechanizmus zabezpečuje dosiahnuteľnosť ľubovoľného bodu z pracovného priestoru ramena. Označme θ_1, θ_2, θ_3 veľkosť jednotlivých uhlov natočenia, potom usporiadaná trojica $\theta = (\theta_1, \theta_2, \theta_3)$ určuje natočenie celého ramena v pracovnom priestore. Každá usporiadaná trojica θ jedno-jednoznačne určuje koncový bod ramena – efektor E. Ak by sme uvažovali robotické rameno s viac ako troma stupňami voľnosti, potom by sme do vzťahu medzi uhlami natočenia a pozíciou koncového

bodu zaviedli nejednoznačnosť (jednej koncovej pozícii by zodpovedalo viacero možných konfigurácií ramena).



Obr. 1.1: Pracovný priestor simulovaného ramena

Pracovný priestor je sledovaný pomocou dvoch kamier, ktoré vnímajú dvojrozmerný obraz a sú schopné identifikovať pozíciu koncového bodu ramena E a cieľového bodu T. Prvá kamera sleduje pracovný priestor spredu a vníma pozíciu efektora ako dvojrozmerný vektor $e_1 = \left(e_1^x, e_1^y\right)$ a pozíciu cieľa ako $t_1 = \left(t_1^x, t_1^y\right)$. Druhá kamera sleduje pracovný priestor zboku a vníma vektory $e_2 = \left(e_2^x, e_2^y\right)$ a $t_2 = \left(t_2^x, t_2^y\right)$. Signály z oboch kamier sú spojené do štvorrozmerných vektorov $E = \left(e_1^x, e_1^y, e_2^x, e_2^y\right)$ a $T = \left(t_1^x, t_1^y, t_2^x, t_2^y\right)$, ktoré budeme používať namiesto súradníc efektora a cieľa v trojrozmernom pracovnom priestore (tieto vektory tvoria trojrozmerný manifold v štvorrozmernom priestore).

Signál z kamier v podobe vektorov E a T je ďalej privedený do jednotky pre ovládanie ramena, ktorá ako odpoveď vygeneruje uhol nových natočení ramena $\theta = (\theta_1, \theta_2, \theta_3)$.

1.2 Riešenie pomocou siete SOM

V Ritter a kol. (1992) je prezentované riešenie využívajúce samoorganizujúcu sa neurónovú sieť SOM, ktoré bolo inšpiráciou pre tému tejto diplomovej práce.

Model SOM Sieť SOM je samoorganizujúca sa neurónová sieť, ktorú tvoria neuróny topologicky organizované do štruktúry (napríklad mriežky). V procese trénovania sa váhové vektory neurónov v mriežke "rozprestrú" v priestore vstupov zachovávajúc distribúciu trénovacích vzoriek. Každý neurón tak získa receptívne pole, teda časť priestoru, v ktorej je jeho aktivácia najvyššia spomedzi všetkých neurónov siete.

Sieť SOM v Ritter a kol. (1992) pozostáva z neurónov, ktoré majú okrem váhových vektorov navyše priradené vektory uhlov natočenia θ_s a Jakobián A_s. Počas trénovania sa v pracovnom priestore rovnomerne rozprestrie sieť SOM a nastavia sa komponenty vektora θ_s a matice A_s.

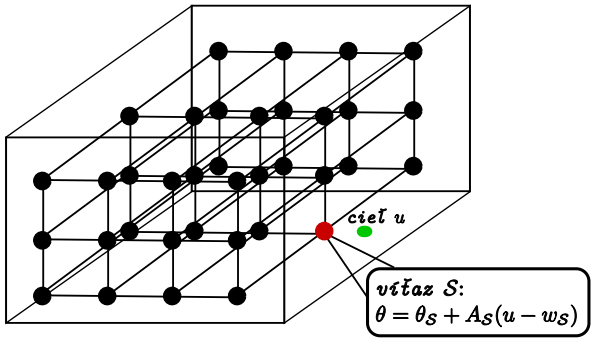
Signál z kamier, určujúci pozíciu cieľového bodu T sa následne privedie na vstup natrénovanej siete SOM a vyberie sa neurón s, v ktorého receptívnom poli bod T leží. Z tohoto neurónu sa prečítajú uhly natočenia θ_s a matica A_s, pomocou ktorých sa vypočítajú výsledné uhly natočenia θ ako

$$\theta = \theta_s + A_s(T - w_s) \tag{1.1}$$

kde w_s je váhový vektor neurónu s. Matica A_s teda slúži na korekciu odchýlky medzi reálnym cieľovým bodom a bodom kvantizovaným sieťou (obrázok 1.2).

1.3 Motivácia a ciele práce

Popísané riešenie pomocou modelu SOM nie je schopné reprezentovať trajektóriu natáčania robotického ramena do cieľového bodu. Po výbere víťaza pre konkrétny cieľ získame iba informáciu, že ak na ramene nastavíme dané uhly, potom bude efektor v cieľovom bode. Rovnako dôležitá je však aj úloha generovania celej



Obr. 1.2: Model siete SOM použitý v Ritter a kol. (1992)

trajektórie, pri ktorej sa vyskytujú ďalšie problémy (maximálna zmena uhla kĺbu v jednom kroku, obchádzanie prekážok v pracovnom priestore, zákaz niektorých konfigurácií ramena a podobne).

Táto diplomová práca si kladie za cieľ pokúsiť sa preskúmať možnosti použitia iných modelov na problém ovládania robotického ramena, ktoré by naviac boli schopné reprezentovať generovanie celej trajektórie pre presun ramena z aktuálnej pozície do cieľového bodu. Takýmto modelom by mohol byť napríklad model RecSOM, ktorý sa od siete SOM líši schopnosťou reprezentovať sekvenciu vstupov – v našom prípade jednotlivé body trajektórie.

Pomocou zvoleného modelu budú implementované simulácie a experimenty, ktorých výsledky budú analyzované.

Kapitola 2

Použité metódy

Nasledujúca kapitola popisuje teoretický základ použitých modelov. V úvodnej časti sa zaoberá sieťou RecSOM, ktorej schopnosť uchovávať sekvenčné údaje by mohla byť použitá na riešenie problému generovania trajektórie pre posun ramena do zvoleného bodu. Návrh architektúry siete RecSOM pre tento problém ako aj výsledky simulácií odhalili niekoľko problémov, preto bude ďalej predstavený nový model založený na učení s posilňovaním.

2.1 RecSOM

2.1.1 Popis modelu

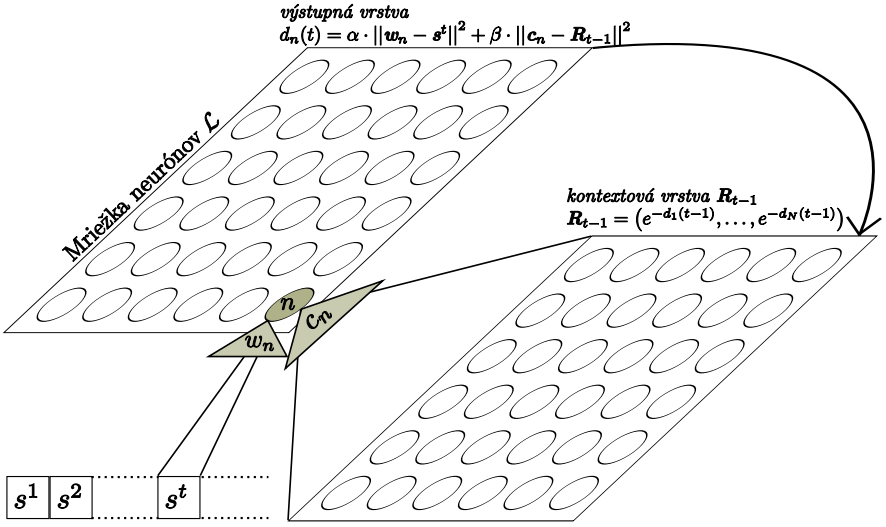
Architektúra modelu Sieť RecSOM (obrázok 2.1) je rekurzívna samoorganizujúca sa neurónová sieť schopná reprezentovať sekvenčné údaje. Vstupom pre sieť je sekvencia $s_1, s_2, \ldots, s_t, \ldots$ Každý prvok sekvencie s_i je m-rozmerný vektor.

Sieť pozostáva z neurónov $n_1, \ldots, n_N$ topologicky organizovaných pomocou mriežky $\mathcal{L}$, ktorá slúži v samo-organizujúcich sa sieťach na definovanie susednosti neurónov. Táto mriežka môže byť principiálne aj viacrozmerná, avšak najčastejšie sa používa pravidelná dvojrozmerná mriežka (Haykin, 1998). Definujeme funkciu vzdialenosti medzi neurónmi n_1 a n_2

$$\eta(n_1, n_2) = \left|\left| n_1^{\mathcal{L}} - n_2^{\mathcal{L}} \right|\right| \tag{2.1}$$

ako euklidovskú vzdialenosť pozície $n_1^{\mathcal{L}}$ a $n_2^{\mathcal{L}}$ neurónov v mriežke $\mathcal{L}$.

Neuróny v mriežke $\mathcal{L}$ tvoria výstupnú vrstvu siete. Aktivácia všetkých neurónov výstupnej vrstvy sa po prezentovaní každého prvku sekvencie skopíruje do kontextovej vrstvy R, pomocou ktorej je v sieti reprezentovaná temporálna informácia. Kontextová vrstva je v prípade siete RecSOM (na rozdiel od iných modelov rekurzívnych SOM sietí) schopná pri komplexných sekvenciách uchovať nekonečnú históriu aktivácií výstupnej vrstvy (Tiňo a kol., 2006).



Obr. 2.1: Model siete RecSOM

Neurón n pozostáva z vektora vstupných váh w_n a vektora kontextových váh c_n. Aktiváciu neurónu n pri prezentovaní prvku s_t zo vstupnej sekvencie definujeme ako $e^{-d_n(t)}$, pričom $d_n(t)$ je vzdialenosť neurónu n od podsekvencie $s_1, s_2, \ldots, s_t$ definovaná ako (Hammer a kol., 2004)

$$d_n(t) = \alpha \cdot \left|\left| \mathbf{w}_n - \mathbf{s}^t \right|\right|^2 + \beta \cdot ||\mathbf{c}_n - \mathbf{R}_{t-1}||^2 \tag{2.2}$$

Trénovanie modelu Základnými princípmi trénovania samo-organizujúcich sa sietí sú *súťaženie* neurónov, ich *spolupráca* a *adaptácia váh* neurónov (Haykin, 1998).

Každý vstup, prezentovaný sieti, vyvolá aktiváciu všetkých neurónov. Neurón, ktorého aktivácia je najväčšia, nazývame víťazom pre daný vstup (súťaženie).

Susedné neuróny spolu spolupracujú, vykonávame zmenu váh všetkých neurónov škálovanú podľa ich vzdialenosti od víťazného neurónu. Trénovanie popisuje algoritmus 1.

Algorithm 1 Trénovanie modelu RecSOM

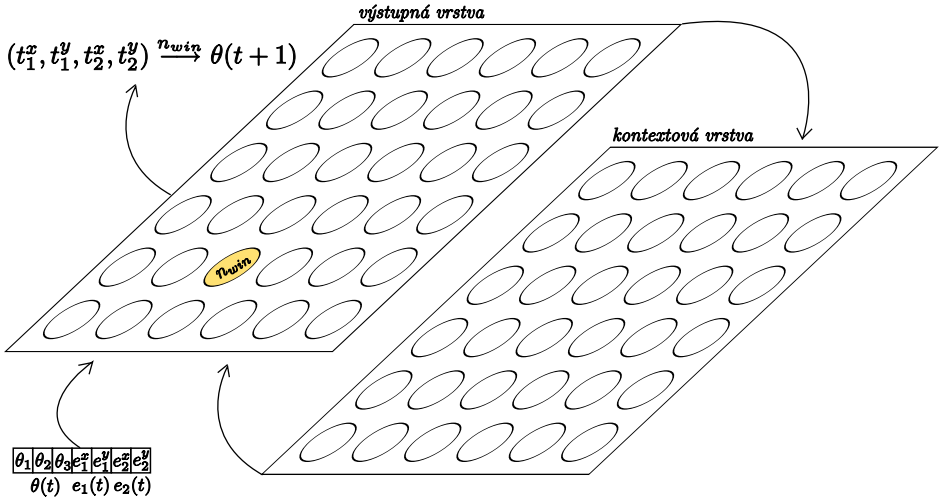
$R_0 \leftarrow (0,0,\ldots,0)$
$iter \leftarrow 1$
for $epoch = 1$ **to** $\#epoch$ **do**
  for all $\{s^1, s^2, \ldots, s^k\} \in S_{train}$ **do**
    for $t = 1$ **to** k **do**
      $n_{win} \leftarrow argmin_i \left\{ d_i(t) = \alpha \cdot ||\mathbf{w}_i - \mathbf{s}^t||^2 + \beta \cdot ||\mathbf{c}_i - \mathbf{R}_{t-1}||^2 \right\}$
      $R_t \leftarrow \left(e^{-d_1(t)}, \ldots, e^{-d_N(t)} \right)$
      for all $n \in \mathcal{L}$ **do**
        $w_n = w_n + \gamma(iter) \cdot \sigma(\eta(n, n_{win}), iter) \cdot \frac{\partial ||w_n - s^t||^2}{\partial w_n}$
        $c_n = c_n + \gamma(iter) \cdot \sigma(\eta(n, n_{win}), iter) \cdot \frac{\partial ||c_n - R_{t-1}||^2}{\partial c_n}$
      end for
      $iter \leftarrow iter + 1$
    end for
  end for
end for

2.1.2 Motivácia pre využitie modelu RecSOM

Trajektóriu presunu ramenu z počiatočného bodu do cieľa možno chápať ako sekvenciu bodov, v ktorých sa má rameno nachádzať v jednotlivých krokoch presunu. Spojitú trajektóriu vlastne diskretizujeme niekoľkými bodmi, pričom predpokladáme, že rameno je schopné v jednom kroku realizovať prechod medzi dvoma susednými bodmi trajektórie.

Sieť RecSOM je možné použiť na reprezentáciu komplexnejších štruktúrovaných údajov, ako sú sekvencie a stromy (Hammer a kol., 2004). Mohla by preto byť

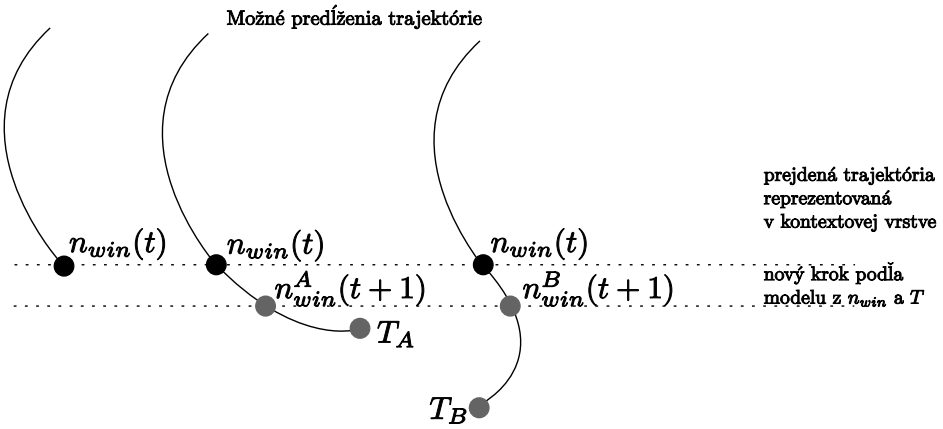
vhodným modelom pre prácu s trajektóriami v podobe sekvencií bodov. Predpokladáme, že po natrénovaní na simulovaných trajektóriách sa jednotlivé neuróny stanú citlivé na kombináciu aktuálnej pozície ramena a smeru trajektórie. Jeden neurón by sa teda stal aproximátorom časti trajektórie a celá výsledná trajektória následne vznikne kooperáciou viacerých neurónov.



Obr. 2.2: Architektúra siete RecSOM pre problém generovania trajektórie

Na obrázku 2.2 je schematicky zobrazená možná architektúra siete RecSOM pre problém generovania trajektórie. Ako vstup pre sieť sme zvolili vektor uhlov natočenia θ a aktuálnu pozíciu efektora ramena e_1, e_2. Samotná sieť RecSOM principiálne nie je schopná na základe vstupu v čase t vygenerovať nové uhly natočenia $\theta(t+1)$, pretože model RecSOM nedokáže reprezentovať mapovanie vstupov na výstupy (dokáže vstupy iba klasterizovať). Takéto mapovanie navyše nie je možné trénovať samo-organizáciou a je potrebné učenie s učiteľom. Tento problém je možné riešiť napríklad pridaním dodatočného modelu ku každému neurónu v mriežke. Pre daný vstup (aktuálnu pozíciu) a daný kontext (históriu predchádzajúcich víťazov, teda vlastne doteraz prejdenú časť trajektórie) v čase t vyberieme z mapy RecSOM víťaza, z ktorého následne prečítame model zodpovedný za vygenerovanie nových uhlov natočenia v čase $t+1$ na základe pozície cieľového bodu T. Po aplikovaní nových uhlov natočenia sa rameno dostane do novej pozície a trajektóriu opäť predĺžime pomocou modelu z nového víťaza. Model zodpovedný

za generovanie uhlov $\theta(t+1)$ by sa v tomto prípade vlastne rozhodoval, ktorý z neurónov bude použitý v ďalšom kroku na predĺženie trajektórie (obrázok 2.3).



Obr. 2.3: Predpokladané vytváranie trajektórie kooperáciou neurónov

Predpokladáme, že jednotlivé neuróny v mriežke by mali byť schopné klasterizovať vstupy podľa prevládajúceho tvaru trajektórie v kombinácii s aktuálnou pozíciou. Trénovanie navrhujeme rozdeliť do dvoch krokov – model RecSOM najprv predtrénovať na zvolených trénovacích trajektóriách a modely v neurónoch následne trénovať napríklad pomocou učenia s posilňovaním.

Použitie siete RecSOM by mohlo priniesť veľmi zaujímavú vlastnosť – schopnosť dokončovania trajektórií pri eliminovaní vizuálnej informácie. Treba si uvedomiť, že sieti RecSOM (obrázok 2.2) dávame na vstup uhly natočenia θ a taktiež vizuálnu informáciu v podobe signálu z kamier e_1 a e_2. Ako sme uviedli v časti 1.1, uhly natočenia jedno-jednoznačne určujú vizuálnu informáciu, teda sieti vlastne dávame redundantnú informáciu. Po eliminovaní vizuálnej zložky vstupného vektora by sieť mala dokázať plnohodnotne dokončiť trajektóriu pomocou uhlov natočenia a iterovaním kontextového vektora. Model by však stále potreboval poznať vizuálnu informáciu o polohe cieľového bodu. Biologická analógia zodpovedá napríklad ruke človeka, ktorá ide uchopiť nejaký predmet, no v malej vzdialenosti od predmetu človek zavrie oči. Polohu cieľa si však naďalej pamätá.

2.1.3 Experimenty

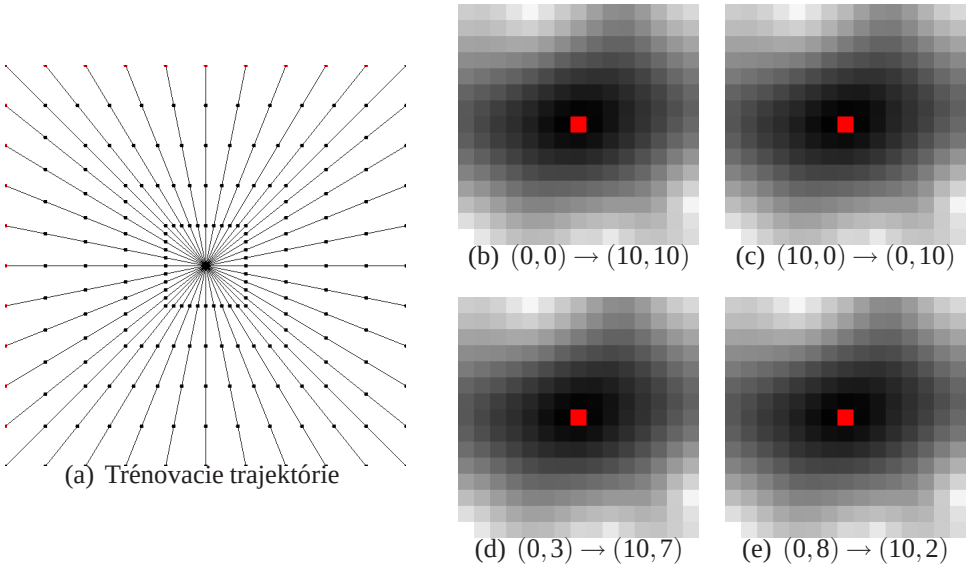
Schopnosť siete RecSOM reprezentovať trajektórie sme sa rozhodli najprv analyzovať na zjednodušenom prípade – sieti sme ako trénovacie dáta prezentovali lineárne trajektórie medzi dvoma náhodne zvolenými bodmi. Trénovanie modelov generujúcich nové uhly natočenia v jednotlivých modeloch sme v tejto fáze vypustili. Na natrénovanej sieti sme následne analyzovali aktiváciu celej mapy a trajektóriu víťazov v mriežke.

Parametre trénovania Sieť RecSOM pozostávala z dvojrozmernej mriežky s veľkosťou 15×15 neurónov. Hodnoty koeficientov pomeru vplyvu kontextu ku vplyvu aktuálneho vstupu v rovnici 2.2 pre aktiváciu neurónu sme zvolili $\alpha = 4.0$ a $\beta = 1.0$. Zvolenie vyššej hodnoty α voči hodnote β je vhodné, pretože kontextový vektor má niekoľkonásobne viac komponentov ako vstupný vektor, teda sa na ňom aj malá odchýlka prejaví výraznejšie (Vančo a Farkaš, 2010). Trénovanie trvalo 200000 iterácií, pričom jednou iteráciou rozumieme prezentovanie jedného prvku sekvencie sieti. Po ukončení jednej sekvencie (trajektórie) nebol kontextový vektor vynulovaný, pretože resetovanie kontextu podľa Hammer a kol. (2004) nemá výraznejší vplyv na správanie modelu.

Trajektórie sme vyberali zo štvorca $\{0..10\}^2$ tak, že začiatky trajektórií ležali na ľavej a vrchnej strane štvorca. Všetky trajektórie pozostávali z jedenástich bodov a prechádzali bodom $(5, 5)$ (obrázok 2.4(a)). Tento spoločný bod bol vybraný kvôli overeniu schopnosti siete diskriminovať trajektórie z rôznych smerov v tom istom bode, teda schopnosť rozlišovať smer prejdenej trajektórie.

Schopnosť reprezentovať trajektórie Ako vyplýva z obrázku 2.4, sieť nedokázala spoľahlivo diskriminovať ani lineárne trajektórie podľa smeru. Trajektória pohybu víťazov v sieti spravidla kopírovala trajektóriu prezentovaných vstupov, sieť teda v podstate diskriminovala iba podľa aktuálnej pozície a smer trajektórie nedokázala rozlíšiť. Experiment sme opakovali s inými hodnotami α a β, avšak tieto hodnoty nemali na vlastnosti modelov skoro žiaden vplyv.

Možné vysvetlenie týchto výsledkov je, že sieť obsahovala príliš málo neuró-



(a) Trénovacie trajektórie

(b) $(0,0) \rightarrow (10,10)$ (c) $(10,0) \rightarrow (0,10)$

(d) $(0,3) \rightarrow (10,7)$ (e) $(0,8) \rightarrow (10,2)$

Obr. 2.4: Trénovacie trajektórie pre sieť RecSOM a zodpovedajúce aktivácie

nov. Zvýšenie veľkosti mriežky by mohlo spôsobiť vznik oblastí neurónov, ktoré by reagovali na tú istú aktuálnu pozíciu a líšili by sa v kontextovom vektore (identifikovali by teda smer trajektórie). Zväčšovali sme preto rozmer mriežky až do veľkosti 60×60 neurónov a experiment sme opakovali s rovnakými parametrami a trajektóriami. Zväčšenie siete žiaľ nepotvrdilo želané správanie. Sieť RecSOM je z hľadiska pamäťovej i časovej zložitosti veľmi náročný model, ktorého výpočty nie je možné ani dostatočne paralelizovať. Väčšie siete sú už preto prakticky nepoužiteľné.

Eliminácia vizuálneho vstupu Úspešne sme overili schopnosť siete dokončovať trajektórie aj pri eliminovanej vizuálnej informácii o aktuálnej pozícii ramena. Sieť sme najprv natrénovali na 50 trajektóriách s dĺžkou 40 bodov. Ako vstup sme jej dávali vizuálnu informáciu o polohe ramena (vektory e_1, e_2) ako aj uhly natočenia θ. Po natrénovaní sme sledovali trajektóriu víťazov v sieti pre všetky trénovacie trajektórie bez zmeny a následne pre trénovacie trajektórie, ktorým sme v posledných 10 krokoch vynulovali hodnoty vektorov e_1, e_2. Trajektórie víťazov v sieti pre vstupy s vizuálnou informáciou a pre vstupy bez nej sa vo všetkých

prípadoch zhodovali.

2.2 Prehľad iných prístupov

Problém generovania trajektórie je v dnešnej dobe možné riešiť pomocou rôznych prístupov. Na jednej strane stoja prístupy, ktoré využívajú učenie s učiteľom a interné modely. Vstupom pre dopredný interný model (Kawato, 1999) je motorický príkaz, pre ktorý dopredný model predikuje jeho realizáciu v prostredí (predikuje teda nasledujúci perceptuálny vstup). Inverzné interné modely naopak k cieľu vygenerujú sekvenciu akcií na jeho dosiahnutie. Tieto modely je možné kombinovať do zložitejších architektúr (Mehta a Schaal, 2002).

Na strane druhej stoja prístupy využívajúce učenie s posilňovaním, ktoré nemusia obsahovať explicitný model prostredia. Ich konvergencia býva preto spravidla podstatne pomalšia, nakoľko je potrebné pracovať s veľkými spojitými priestormi stavov a akcií. Existuje tu priestor pre urýchlenie konvergencie, napríklad vhodnou organizáciou stavového priestoru v kombinácii s vhodnými funkciami odmeny Tamosiunaite a kol. (2009) dosiahli rýchlejšie konvergujúce modely. Stavový priestor je v tomto prípade dekomponovaný a generovanie trajektórie je založené na kooperácii lokálnych modelov. Princíp dekompozície priestoru a stavania lokálnych modelov sme uvažovali v prípade predchádzajúceho modelu siete RecSOM, ktorý sme navrhovali.

2.3 Učenie s posilňovaním

Jednou z metód strojového učenia, ktorá by mohla byť použitá na riešenie problému ovládania ramena, je učenie s posilňovaním (*reinforcement learning*). V probléme učenia s posilňovaním vystupuje agent, ktorý je umiestnený v prostredí. Agent sa v tomto prostredí autonómne pohybuje a vykonáva zvolené akcie. Za vykonávané akcie od prostredia dostáva odmenu alebo trest, teda informáciu o tom, ako dobre sa v prostredí správa. Cieľom agenta je naučiť sa čo možno najlepšiu stratégiu (výber akcií v konkrétnych stavoch) vzhľadom na funkciu odmeny a trestu. Tento problém

môže byť naviac ešte zložitejší, ak odmena alebo trest neprichádza po každom kroku, ale vzťahuje sa na niekoľko vykonaných akcií v minulosti. Prostredie agenta môže byť len čiastočne pozorovateľné, teda vykonanie akcie agentom môže byť zmenené vplyvmi prostredia, ktoré agent nepozná.

Klasický problém učenia s posilňovaním je definovaný ako Markovovský rozhodovací problém (MDP) (Sutton a Barto, 1998). MDP je usporiadaná štvorica $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P})$, kde

- $\mathcal{S}$ je konečná množina stavov
- $\mathcal{A}$ je konečná množina akcií
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ je funkcia ohodnocujúca správanie agenta (funkcia odmeny a trestu)
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0,1]$ je pravdepodobnostná distribúcia, $\mathcal{P}(s_t, a_t, s_{t+1})$ určuje pravdepodobnosť prechodu do stavu s_{t+1}, ak v stave s_t vykonáme akciu a_t

Agent sa v čase t nachádza v stave $s_t \in \mathcal{S}$, v ktorom môže aplikovať niektorú z akcií $\mathcal{A}$. Na základe svojej naučenej stratégie $\pi : \mathcal{S} \times \mathcal{A} \to \langle 0,1 \rangle$ zvolí najvhodnejšiu akciu, ktorá maximalizuje získané odmeny v ďalších krokoch. Tieto odmeny sú smerom do budúcnosti znižované (diskontované), aby bol agent nútený maximalizovať svoju odmenu čo možno najrýchlejšie. Odmena v budúcnosti je teda

$$r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots = \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{2.3}$$

kde $\gamma \in \langle 0,1 \rangle$ je diskontný faktor.

2.3.1 Ohodnocovacia funkcia, Q-učenie

Agent sa pri učení optimálnej stratégie môže riadiť ohodnocovaním odmeny pre jednotlivé stavy, alebo pre dvojice $(stav, akcia)$. Pri ohodnocovaní jednotlivých stavov určuje ohodnotenie $V(s)$ konkrétneho stavu s predpokladanú diskontovanú odmenu, ktorú tento stav sľubuje v budúcnosti. Pri ohodnocovaní akcií vzhľadom

na stavy ohodnotenie $Q(s,a)$ pre každú dvojicu stavu a akcie určuje opäť predpokladanú diskontovanú odmenu, ktorú je možné získať v budúcnosti po vykonaní akcie a v stave s. Označme π^* optimálnu stratégiu agenta, potom ohodnotenie stavov je rovné

$$V^*(s_t) = \max_a \sum_{s_{t+1}} \mathcal{P}(s_t, a, s_{t_1}) \left(\mathcal{R}(s_t, a, s_{t_1}) + \gamma V^*(s_{t+1})\right) \tag{2.4}$$

a pre ohodnotenie akcií vzhľadom na stavy platí

$$\max_a Q^*(s,a) = V^*(s) \tag{2.5}$$

2.4 CACLA

Problém učenia s posilňovaním formalizovaný ako MDP je však pre potreby ovládania robotického ramena nevhodný. Pracuje s konečnou množinou stavov aj akcií, no robotické rameno sa pohybuje v spojitom priestore a jeho ovládanie by tiež malo byť spojité. V nasledujúcej časti popíšeme algoritmus učenia s posilňovaním, ktorý pracuje so spojitým priestorom stavov aj akcií – algoritmus CACLA (*Continuous actor critic learning automaton*) (van Hasselt a Wiering, 2007).

2.4.1 Učenie pomocou aktéra a kritika, algoritmus ACLA

Algoritmus CACLA je rozšírením algoritmu ACLA – algoritmu z kategórie učenia s posilňovaním pomocou aktéra a kritika (Sutton a Barto, 1998), ktorý uvažuje problém MDP s diskrétnou množinou stavov a akcií. Rozoznáva dve entity – *aktéra* a *kritika*. Aktér aktívne zasahuje do správania agenta, je zodpovedný za generovanie akcií na základe stavov. Kritik sa na generovaní akcií nepodieľa, avšak ohodnocuje vhodnosť akcií vyberaných aktérom.

Aktér $Ac(s_t, a_t)$ je v podstate tabuľka, ktorá pre každú kombináciu stavu a akcie obsahuje hodnotu pravdepodobnosti výberu danej akcie v danom stave. Kritik $V(s_t)$ je opäť tabuľka obsahujúca ku každému stavu jeho ohodnotenie.

Ohodnotenie stavov podľa kritika sa počas učenia mení podľa TD pravidla

$$V_{t+1}(s_t) = V_t(s_t) + \alpha_t \delta_t \tag{2.6}$$

kde $\delta_t = r_t + \gamma V_t(s_{t+1}) - V_t(s_t)$ je chyba a α_t je rýchlosť učenia. Pravdepodobnosti uložené v tabuľke aktéra sa menia vtedy, ak ohodnotenie nového stavu s_{t+1}, do ktorého sme sa dostali zo stavu s_t aplikovaním akcie a, je lepšie. Teda ak $\delta_t > 0$ potom zvýšime pravdepodobnosť výberu akcie a v stave s_t.

2.4.2 Rozšírenie na spojitý priestor stavov a akcií

Pri prechode k spojitým stavom a akciám potrebujeme vlastne iba nahradiť "tabuľky" aktéra a kritika za spojité funkcie. Aktéra nahradíme funkciou $Ac : \mathcal{S} \to \mathbb{R}^n$, ktorá pre daný stav s_t vygeneruje akciu $Ac(s_t)$. Kritika nahradíme funkciou $V : \mathcal{S} \to \mathbb{R}$, ktorá pre daný stav s_t vypočíta jeho ohodnotenie $V(s_t)$. Na mieste aktéra a kritika teda použijeme aproximátory funkcií so sadou parametrov Φ^{Ac}, resp. Φ^V.

2.4.3 Trénovanie modelu

Parametre aproximátorov v procese učenia meníme podľa vzťahu (van Hasselt a Wiering, 2007)

$$\Phi^V_{t+1} = \Phi^V_t + \alpha_{Ac}\delta_t \frac{\partial V_t(s_t)}{\partial \Phi^V_t} \tag{2.7}$$

$$\Phi^{Ac}_{t+1} = \Phi^{Ac}_t + \alpha_V (a_t - Ac_t(s_t)) \frac{\partial Ac_t(s_t)}{\partial \Phi^{Ac}_t} \tag{2.8}$$

kde α je rýchlosť učenia. Parametre aktéra Φ^{Ac} meníme iba vtedy, ak $\delta_t > 0$. Detaily učenia popisuje algoritmus 2.

Jeden krok algoritmu pozostáva z výberu najvhodnejšej akcie $Ac_t(s_t)$ pre stav s_t, ktorá je následne modifikovaná. Touto modifikáciou získava agentov potrebný priestor pre exploráciu – objavenie vhodnejšej akcie ako je $Ac_t(s_t)$. V závislosti na probléme je možné zvoliť vhodné pravidlo explorácie, napríklad

- Gassovskú exploráciu

  Definujeme Gaussovskú pravdepodobnostnú distribúciu p nad akciami a

$$p(s_t, a) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(a - Ac_t(s_t))^2}{2\sigma^2}} \tag{2.9}$$

a podľa tejto distribúcie náhodne zvolíme akciu a_t.

- ε-greedy exploráciu

  S pravdepodobnosťou ε si zvolíme úplne náhodnú akciu a a túto akciu akceptujeme ako novú akciu a_t. S pravdepodobnosťou $(1-\varepsilon)$ ponecháme akciu $Ac_t(s_t)$ nezmenenú.

Akcia a_t vybraná v procese explorácie sa následne aplikuje a agent prejde do stavu s_{t+1}. Za tento prechod získa odmenu alebo trest r_t. Pomocou kritika sa ohodnotia stavy s_t a s_{t+1} a ak akcia a_t viedla k lepšie ohodnotenému stavu, potom sa snažíme priblížiť akciu $Ac_t(s_t)$ k akcii a_t. Aktéra aj kritika teda trénujeme simultánne, predikcie kritika sa upravujú v každom kroku a predikcie aktéra iba vtedy, ak prinesú lepšie ohodnotený stav. Ako vyplýva z algoritmu 2, tento model vyžaduje odmenu alebo trest po každom kroku.

Algorithm 2 Trénovanie modelu CACLA (van Hasselt a Wiering, 2007)

$s_0 \leftarrow$ `počiatočný_stav`
$\Phi_0^{Ac} \leftarrow$ `random()`
$\Phi_0^{V} \leftarrow$ `random()`
for $t = 0$ **to** `#krokov` **do**
  $a_t \leftarrow$ `explore`$(Ac_t(s_t))$
  $s_{t+1} \leftarrow$ `apply`(s_t, a_t)
  $r_t \leftarrow \mathcal{R}(s_t, a_t, s_{t+1})$
  $\delta_t \leftarrow r_t + \gamma V_t(s_t+1) - V_t(s_t)$
  $\Phi_{t+1}^{V} = \Phi_t^{V} + \alpha_{Ac}\delta_t \frac{\partial V_t(s_t)}{\partial \Phi_t^{V}}$
  if $\delta_t > 0$ **then**
    $\Phi_{t+1}^{Ac} = \Phi_t^{Ac} + \alpha_V(a_t - Ac_t(s_t)) \frac{\partial Ac_t(s_t)}{\partial \Phi_t^{Ac}}$
  end if
end for

2.4.4 Možnosť použitia modelu na ovládanie ramena

Model CACLA by mohol byť použiteľný na riešenie problému ovládania ramena. Jednotlivé stavy by zodpovedali pozícii ramena a cieľového bodu v pracovnom priestore a akcie by zodpovedali zmene uhlov natočenia ramena. Ako funkciu odmeny a trestu by bolo možné použiť euklidovskú vzdialenosť ramena od cieľa. Použitie modelu CACLA na ovládanie ramena a jeho vlastnosti ďalej podrobne popíšeme v kapitole 4.

Kapitola 3

Implementácia

V tejto kapitole popíšeme implementačnú časť diplomovej práce. V úvode kapitoly predstavíme základné rozdelenie komponentov aplikácie a základný popis navrhovanej implementácie. Následne detailnejšie popíšeme jednotlivé knižnice tak, aby bolo ich prípadné použitie pre iné experimenty čo možno najjednoduchšie.

3.1 Návrh

3.1.1 Programovací jazyk

Aplikácia je naprogramovaná v programovacom jazyku Java, ktorý prináša niekoľko výhod. Tento jazyk patrí medzi moderné programovacie jazyky a vznikol z jazyka C++ odstránením alebo obmedzením tých možností, ktoré spôsobovali najväčší počet chýb v programoch a boli označované ako “pasce pre programátorov”. Medzi tieto možnosti patrí napríklad smerníková aritmetika. Bol tiež zavedený tzv. *garbage collector* riešiaci problém s uvoľňovaním pamäte. Pri tvorbe jazyka JAVA boli taktiež odstránené akékoľvek väzby na platformu, na ktorej je program vykonávaný. Zdrojové kódy programu sú najprv skompilované do tzv. *bytecode*, ktorý je následne interpretovaný v interpreteri *Java virtual machine*. Tento spôsob prináša niekoľko ďalších benefitov v podobe vyššej bezpečnosti behu programov (programy nepristupujú priamo do operačnej pamäte počítača a nemôžu tak prepísať pamäťové miesto iného programu) a optimalizácie alokovania pamäte.

Z podstaty multiplatformového konceptu Javy vyplývajú samozrejme aj niektoré nevýhody, hlavne o niečo pomalší beh programu (samotný program je spomalený réžiou samotnej *Java virtual machine* a behom *garbage collector*-a). Toto spomalenie je v moderných verziách jazyka Java (aktuálna verzia 1.6) výraznejšie cítiť v podstate iba pri špecifických náročnejších výpočtoch. Tento problém sme sa rozhodli riešiť použitím natívnych metód (podrobnosti popíšeme neskôr).

Nespornou výhodou je tiež idea otvoreného softvéru. Existuje úplne otvorená implementácia jazyka Java, v ktorej má programátor prístup ku zdrojovým kódom všetkých zabudovaných knižníc jazyka. Zdarma sú dostupné tiež viaceré moderné integrované vývojové prostredia s pokročilými funkciami ako sú refaktorizácia zdrojového kódu, nástroje pre ladenie programu (*debugging*) a optimalizáciu rýchlosti behu programu (*profiling*), integrované testovanie zdrojového kódu. Na internete sú k dispozícii mnohé knižnice na riešenie najrôznejších úloh (všetky použité knižnice sú explicitne uvedené medzi zdrojmi).

3.1.2 Rozdelenie na knižnice

Implementáciu sme sa rozhodli rozdeliť do viacerých ucelených knižníc, ktoré sú znovu použiteľné aj samostatne. Ťažiskom implementácie je neurónová sieť RecSOM a model učenia s posilňovaním CACLA. Model RecSOM je výpočtovo náročný model, ktorého výpočty pracujú s rôznymi matematickými funkciami.

Implementácia niektorých matematických funkcií v jazyku Java je podstatne pomalšia, ako ich výpočet v jazyku C++ priamo na procesore mimo *Java virtual machine*. Prvou knižnicou teda bude knižnica *Math* na prácu s matematickými objektami ako sú vektory a matice, ktorú budú ostatné knižnice používať. Matematické operácie nad týmito objektami tak budú v podstate centralizované a zvýšenie efektivity výpočtov sa bude diať len na úrovni tejto knižnice bez akéhokoľvek zásahu do ostatných častí aplikácie.

Ďalším spôsobom pre zvýšenie efektivity navrhovanej implementácie je využitie paralelizmu. V dnešnej dobe je už v podstate bežné, že aj domáce počítače sú vybavené viacerými procesorovými jadrami. Preto ako ďalšiu podpornú knižnicu navrhujeme knižnicu pre paralelné výpočty *Parallel*.

Hlavnými knižnicami sú knižnice implementujúce použité modely *RecSOM* a *CACLA*. Pri ich implementácii budeme dbať na čo najväčšiu možnosť znovu použitia modelov pre iné experimenty a nové rozšírenia samotných knižníc.

Zobrazenie pracovného priestoru ramena v trojrozmernom priestore má na starosti knižnica pre grafické rozhranie. Knižnica ďalej obsahuje implementáciu objektov robotického ramena a kamier.

Poslednou knižnicou je knižnica s implementáciou servera, ktorý sa stará o výpočet zadávaných simulácií.

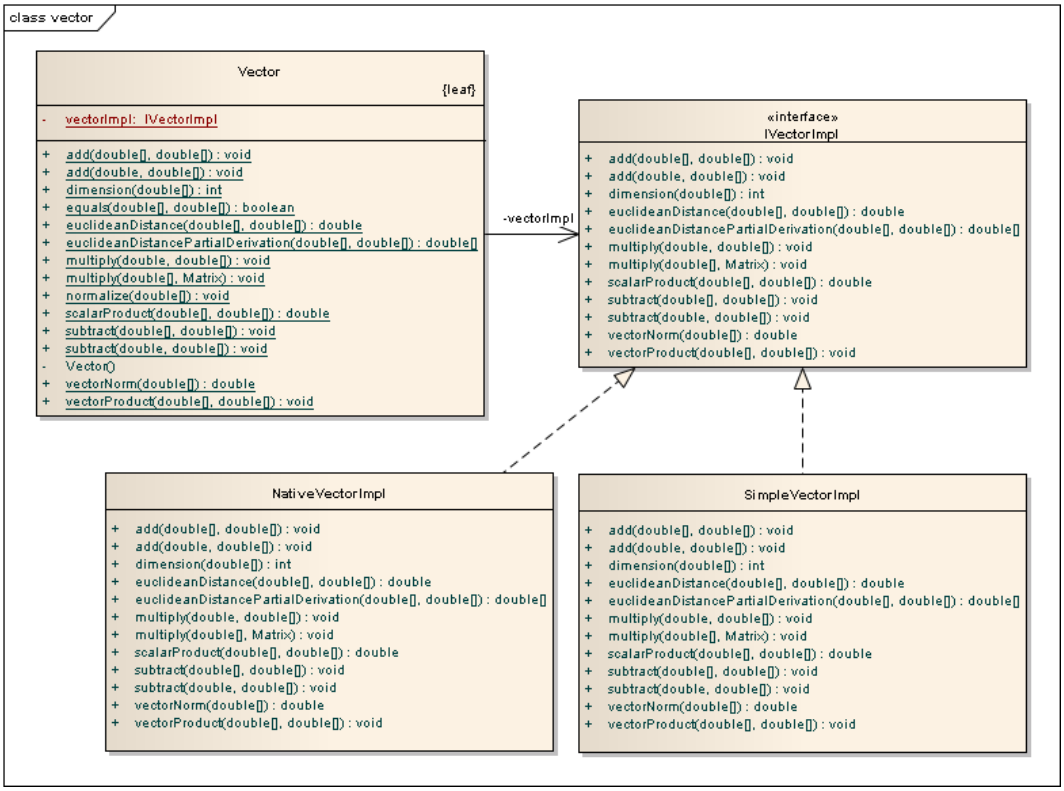
3.1.3 Knižnica *Math*

Základom knižnice *Math* je práca s vektormi a maticami implementovaná v balíku `math`. Samotné vektory sú reprezentované ako jednorozmerné pole a všetky funkcie sú sústredené do statických metód triedy `Vector`. Tento prístup je implementáciou návrhového vzoru *Flyweight* (Pecinovský, 2007), teda namiesto vytvárania veľkého množstva inštancií príliš jednoduchých objektov pre vektory ponecháme údaje, ktoré identifikujú konkrétny vektor, mimo objekt a vytvoríme len jeden spoločný objekt pre všetky inštancie vektora.

Trieda `Vector` využíva pomocou kompozície objekt typu `IVectorImpl`, v ktorom je implementovaná funkcionalita pre vektorové operácie (obrázok 3.1). Metódy triedy `Vector` vykonajú pred a po volaní metód `IVectorImpl` niekoľko kontrol vstupných a výstupných podmienok pomocou príkazu `assert`. Rozhranie `IVectorImpl` implementujú dve triedy – `SimpleVectorImpl` a `NativeVectorImpl`. Tento princíp zabezpečí, že používateľ knižnice môže pohodlne pracovať priamo s triedou `Vector`, ktorá sama automaticky použije buď natívnu alebo klasickú implementáciu vektorových funkcií. Pre používateľa je tento postup úplne transparentný.

Aplikáciou operácií na vektor sa menia priamo operandy operácie, napríklad ak chceme spočítať dva vektory pomocou metódy `add(A,B)`, potom vykonaná operácia je `A = A + B`. Tento prístup zabezpečuje šetrnejšiu alokáciu pamäte, pretože primárnym použitím vektorov bude knižnica *RecSOM*, kde počas trénovania siete budeme k vektoru váh pripočítavať vektor zmien $w_{t+1} = w_t + \delta_{w_t}$.

V tomto balíku sa ďalej nachádza analogická funkcionalita pre prácu s maticami. Matica je reprezentovaná pomocou objektu `Matrix`, ktorý obsahuje údaje aj metódy pre matice. V tomto prípade sme teda zvolili použitie návrhového vzoru *Hodnotový typ* (Pecinovský, 2007), pretože nepredpokladáme až taký veľký počet matíc v programe. Tento vzor sa používa v prípade, ak implementujeme nový dátový typ spolu s jeho operáciami. Jedna inštancia hodnotového typu obsahuje údaje, ktoré sa už ďalej nemenia. Operácie preto vracajú ako výsledok nové inštancie tohoto dátového typu. Teda napríklad, ak máme inštanciu objektu `Matrix` a pripočítame k nej inú maticu, potom ako výsledok dostaneme úplne novú inštanciu a pôvodné matice sa nezmenia. Trieda `Matrix` opäť využíva buď klasickú, alebo natívnu implementáciu maticových operácií.



Obr. 3.1: Diagram tried `Vector`

Spôsob zavedenia natívnej knižnice

Natívna knižnica pozostáva zo zdrojových kódov v jazyku C++, ktoré sa pomocou priloženého skriptu programom *make* skompilujú a zbalia do knižnice *libvector.so*. Priložený skript *Makefile* je vytvorený pre operačný systém Linux s procesorom Intel Core 2 a používa kompilátor GCC. Pre použitie na inej platforme je potrebné tento skript patrične modifikovať.

O zavedenie natívnej implementácie vektorových a maticových operácií sa stará trieda `math.Factory`. Obsahuje vymenovaný typ `Factories` s metódami pre vytváranie objektov `IVectorImpl` (resp. `IMatrixImpl`). Vo svojom konštruktore sa pokúsi zaviesť natívnu knižnicu *libvector.so* a následne otestuje, ktorá z implementácií je na danom počítači rýchlejšia. Túto implementáciu bude ďalej používať.

Objekt typu `Vector` vo svojej statickej inicializácii, ktorá sa vykoná iba raz pri načítaní triedy do *Java Virtual Machine*, pomocou objektu `Factory` popísaným spôsobom získa inštanciu `IVectorImpl`, s ktorou následne pracuje. Celý postup sa teda odohrá úplne automaticky pri prvom pokuse o použitie niektorej metódy objektu `Vector`. Analogicky tento postup funguje aj pre objekt `Matrix`.

Lineárne a plošné objekty

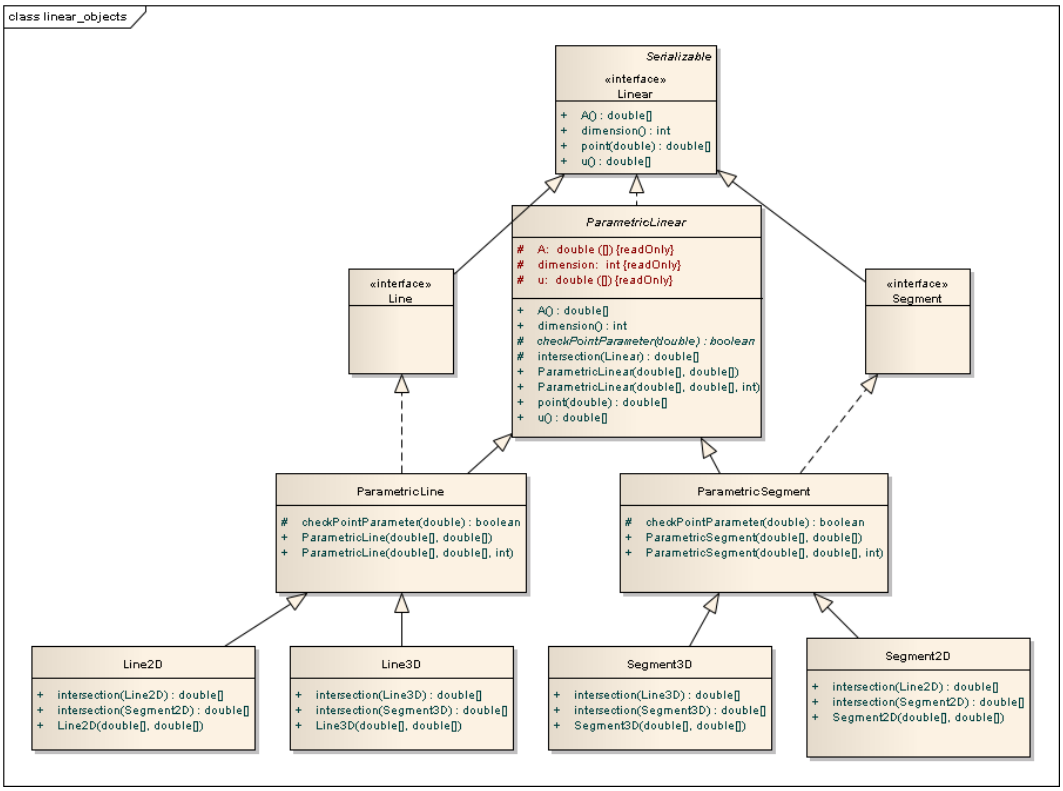
V balíku `math.linear_objects` sa nachádza implementácia objektov, ktoré reprezentujú lineárne objekty (priamky a úsečky). UML diagram tried tohoto balíka sa nachádza na obrázku 3.2. Všetky objekty implementujú rozhranie `Linear`. Spoločná implementácia parametricky vyjadrenej priamky v n-rozmernom priestore je implementovaná v abstraktnej triede `ParametricLinear`. Tá obsahuje metódy pre výpočet súradníc bodu podľa zadaného parametra a pre výpočet priesečníka s inou priamkou. Pri tomto výpočte sa používa metóda objektu `Matrix`, ktorá pomocou Gaussovho eliminačného procesu vypočíta parametre priesečníka dvoch priamok.

Potomkovia tejto triedy môžu mať obmedzenia na hodnoty parametra, napríklad pri zvolení parametra z intervalu $\langle 0, 1 \rangle$ dostávame úsečku. Potomkovia sa ďalej delia podľa dimenzie priestoru, implementovali sme dvoj- a trojrozmerné

verzie priamok a úsečiek.

V balíku `math.planar_objects` sú analogicky implementované objekty reprezentujúce rovinu `Plane` a obdĺžnikovú časť roviny `Rectangle`. Všetky tieto objekty sa používajú pri detekcii kolízií trajektórie ramena a hraníc pracovného priestoru, prípadne prekážok v pracovnom priestore.

V balíku `math.spatial_objects` sú implementované jednoduché objekty reprezentujúce podpriestor, ktoré sa používajú taktiež pre určenie pracovného priestoru ramena.



Obr. 3.2: Diagram tried lineárnych objektov

Testovanie

Pre objekty knižnice *Math* boli vytvorené testovacie metódy. Na testovanie bola použitá technológia *JUnit 4*, ktorej podpora je priamo integrovaná vo vývojovom

prostredí NetBeans. Pre ľubovoľnú triedu dokáže vývojové prostredie vygenerovať prototyp testovacieho scenára, ktorý je však takmer vždy nutné zmysluplne modifikovať. Tieto testy je možné následne spustiť a ich výsledky sa nakoniec prehľadne zobrazia.

3.1.4 Knižnica *ParallelComputing*

Knižnica *ParallelComputing* implementuje podporu pre paralelné a distribuované počítanie úloh. Všetky objekty tejto knižnice sa nachádzajú v balíku `parallel`.

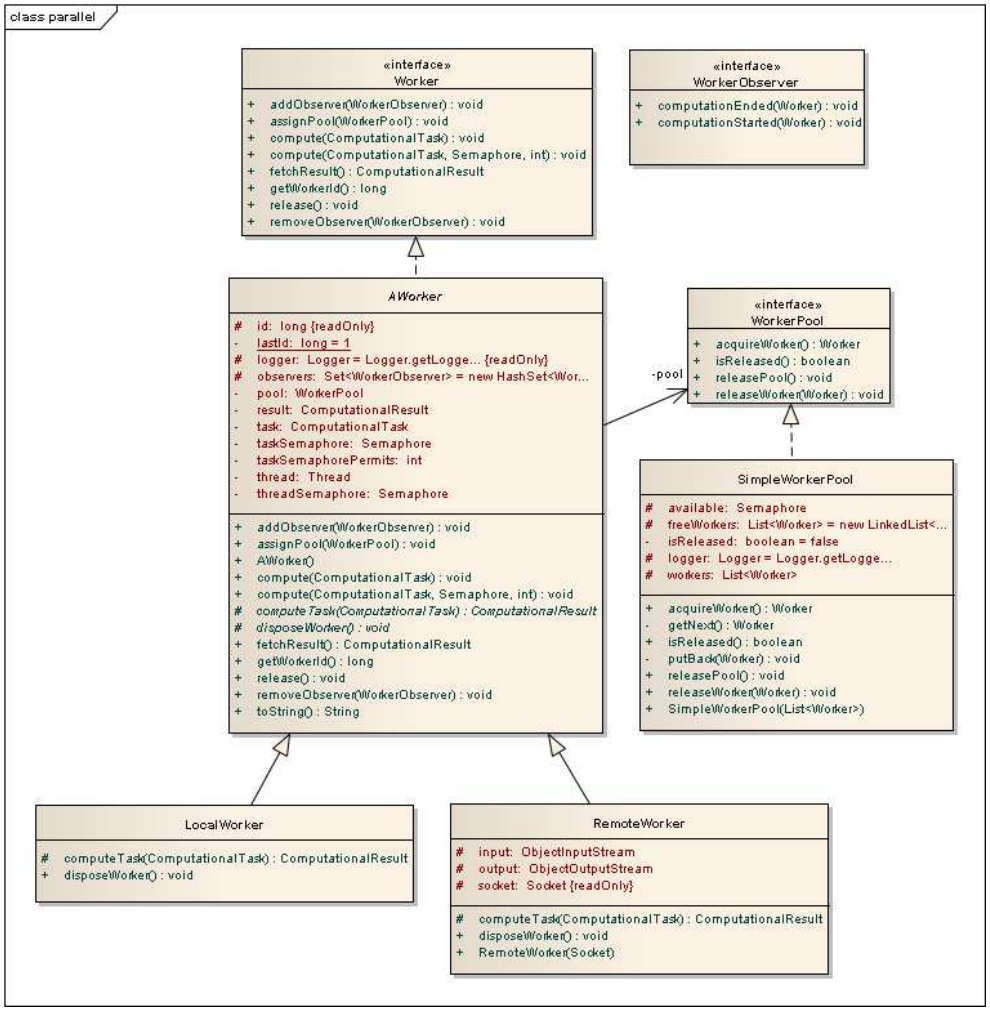
Základným rozhraním v tomto balíku je rozhranie `ComputationalTask`, ktoré implementujú všetky objekty reprezentujúce výpočtové úlohy. Výpočet úlohy začne volaním metódy `beforeExecution()`, ktorá môže obsahovať inicializačné akcie. Nasleduje volanie hlavnej metódy výpočtu `execute()` a po skončení výpočtu sa zavolá metóda `afterExecution()` s finalizačnými akciami. Metóda `execute()` vracia inštanciu triedy `ComputationalResult` s výsledkom výpočtu, ktorý môže byť serializovaný a prenášaný napríklad cez sieť.

Všetky údaje, ktoré majú byť použité počas výpočtu, musia byť serializovateľné a obsiahnuté v objekte `ComputationalTask`. Rovnako všetky údaje, ktoré majú zostať zachované po skončení výpočtu musia byť obsiahnuté v objekte `ComputationalResult`, pretože v prípade distribuovaného výpočtu budú úlohy počítané na rôznych počítačoch.

Výpočtové vlákna, ktoré budú použité pri počítaní úloh, implementujú rozhranie `Worker` (obrázok 3.3). Spoločná časť funkcionality je implementovaná v triede `AWorker`, ktorej potomkami sú dve triedy – trieda `LocalWorker` reprezentujúca lokálne výpočtové vlákno a trieda `RemoteWorker` reprezentujúca vzdialené vlákno.

Výpočet pomocou sieťového výpočtového vlákna `RemoteWorker` začne prenesením počítanej úlohy `ComputationalTask` na vzdialený počítač, na ktorom sa následne spustí výpočet, ktorého výsledok sa prenesie späť na server. Výpočty preto musia mať primeranú zložitosť, pretože inak bude riešenie neefektívne kvôli réžii prideľovania vlákna a prenosu úlohy cez sieť.

Všetky použiteľné vlákna sú spravované triedou `WorkerPool`, ktorá imple-



Obr. 3.3: Diagram tried knižnice *ParallelComputing*

mentuje návrhový vzor *Fond* (Pecinovský, 2007). Tento vzor použijeme vtedy, ak máme obmedzený počet nejakých zdrojov, ktoré chceme prideľovať klientom. Klienti po použití vrátia pridelený zdroj späť do fondu a ten môže byť ďalej prideľovaný. Fond `WorkerPool` spravuje lokálne aj sieťové výpočtové vlákna, teda klient pri vyžiadaní vlákna nevie, či sa jeho úloha bude počítať lokálne, alebo na vzdialenom počítači. Celá réžia sa vykoná automaticky bez jeho vedomia.

Postup výpočtu úlohy

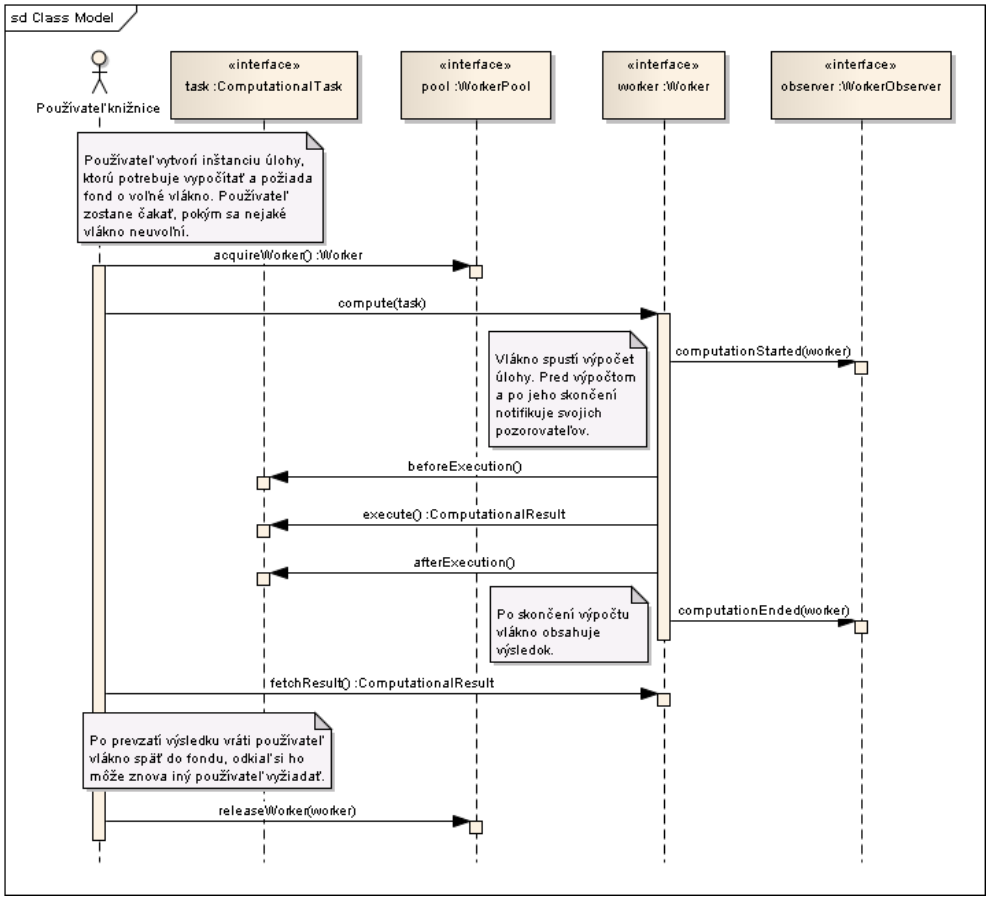
Sekvenčný UML diagram, zobrazujúci proces výpočtu jednej úlohy `ComputationalTask`, je zachytený na obrázku 3.4. Používateľ knižnice najprv vytvorí inštanciu úlohy, ktorú potrebuje vypočítať. Následne požiada fond `WorkerPool` o pridelenie výpočtovej vlákna. Fond obsahuje synchronizačný objekt typu `Semaphore`, pomocou ktorého riadi prístup k výpočtovým vláknam. Ak momentálne žiadne vlákno nie je voľné, potom používateľ knižnice zostane čakať, kým iný používateľ neuvoľní držané vlákno a tým neuvoľní semafór.

Po pridelení výpočtového vlákna používateľ volaním metódy `execute()` spustí výpočet. Výpočtové vlákno najprv notifikuje svojich pozorovateľov (Pecinovský, 2007) – objekty implementujúce rozhranie `WorkerObserver`.

Výpočet pozostáva z inicializačnej fázy, v ktorej sa zavolá metóda počítanej úlohy `beforeExecution()`. Tá umožňuje úlohe inicializovať svoj vnútorný stav pred začatím výpočtu. V ďalšom kroku sa spustí výpočet pomocou metódy `execute()` a po jeho ukončení nastane finalizačná fáza volaním metódy `afterExecution()`, v ktorej má úloha možnosť uvoľniť použité zdroje. Následne sú opäť notifikovaní všetci pozorovatelia o ukončení výpočtu. Výsledkom výpočtu je objekt implementujúci rozhranie `ComputationalResult`, ktorý nesie informácie o stave výpočtu (úspešné alebo neúspešné ukončenie výpočtu). Výsledok výpočtu ďalej obsahuje všetky výstupy výpočtu. Tento výsledok je možné získať z výpočtového vlákna pomocou metódy `fetchResult()`. Výpočtové vlákno už ďalej nie je potrebné a vráti sa späť do fondu, odkiaľ môže byť znova použité na iný výpočet.

3.1.5 Knižnica *RecSOM*

Knižnica *RecSOM* obsahuje implementáciu siete. Predpokladaným použitím tejto knižnice sú rôznorodé simulácie, preto sme sa pri implementácii snažili čo možno najviac prispôsobiť knižnicu tomuto účelu. Knižnica obsahuje viacero grafických výstupov, ktoré je možné jednoducho aktivovať a deaktivovať. Z hľadiska návrhu sa knižnica snaží byť modulárna, aby bolo možné sieť použitú v simulácii čo



Obr. 3.4: Diagram paralelného výpočtu

najviac modifikovať. Vo väčšine prípadov sme v architektonickom dizajne tried uprednostnili kompozíciu pred dedením, nakoľko dedenie vytvára príliš pevné väzby medzi triedami a často vedie ku kurióznym dizajnom (Eckel, 2005, s. 261). Knižnica ďalej umožňuje ukladanie siete do súboru a do databázy.

Hlavným balíkom knižnice *RecSOM* je `ann.recsom`. UML diagram hierarchie tried je zobrazený na obrázku 3.5. Samotná sieť RecSOM je reprezentovaná rovnomenným rozhraním `RecSOM`, skladá sa z neurónov implementujúcich rozhranie `Neuron`, ktoré sú zoradené v štruktúre implementujúcej rozhranie `NeuronMatrix`. Kontext siete RecSOM implementuje rozhranie `Context`. Pre reprezentáciu kontextu samostatnou triedou sme sa rozhodli z dôvodu vyššej modularity, pretože zámenou objektu typu `Context` môžeme jednoducho imple-

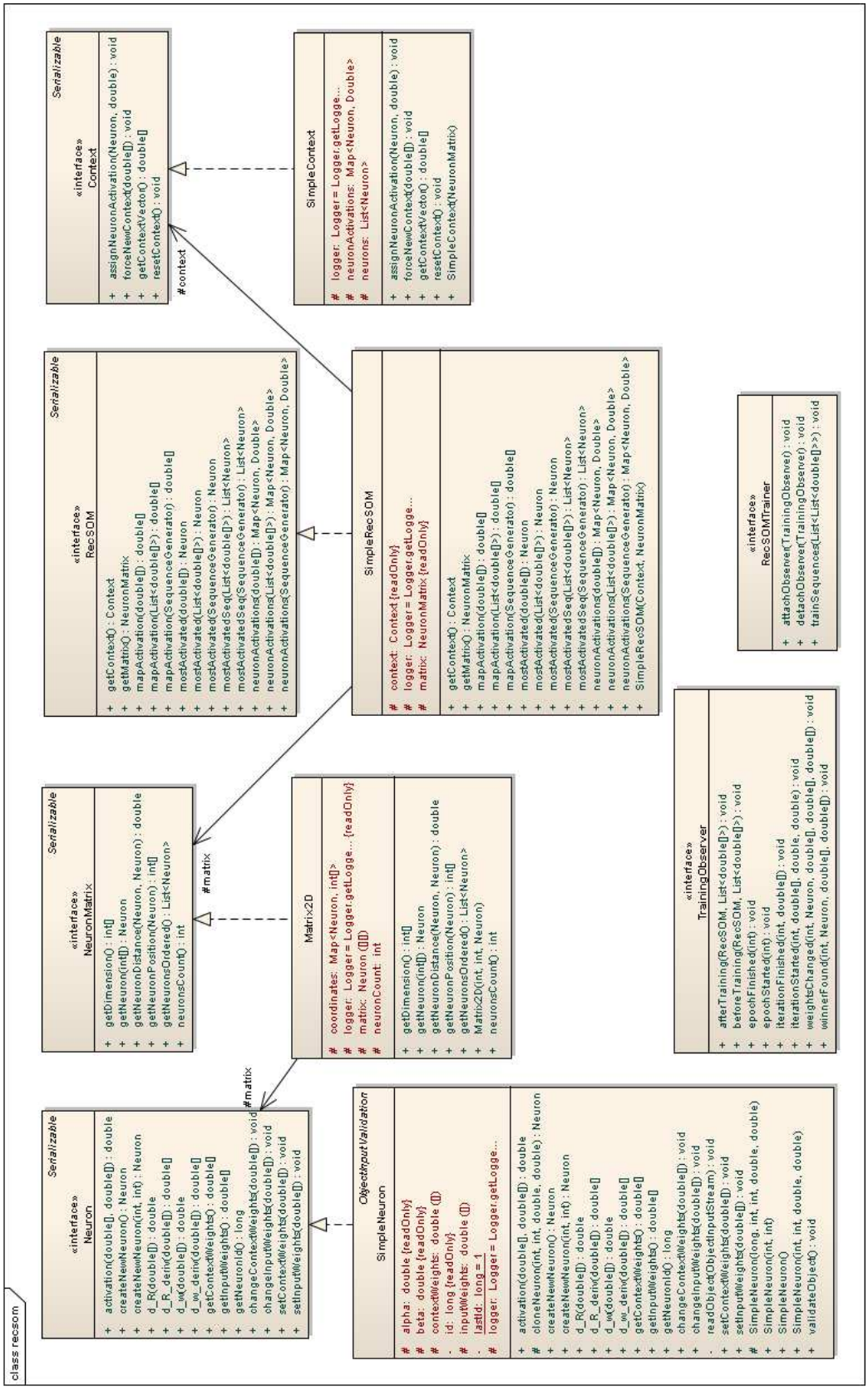
mentovať príbuzné neurónové siete – napríklad MSOM (Hammer a kol., 2004).

Na trénovanie siete `RecSOM` slúžia objekty, ktoré implementujú rozhranie `RecSOMTrainer`. Môžu k nim byť pripojení pozorovatelia trénovania implementujúci rozhranie `TrainingObserver`.

Rozhranie `RecSOM` obsahuje metódy pre aktiváciu siete pomocou sekvencií. Konkrétnou implementáciou tohoto rozhrania je trieda `SimpleRecSOM`, ktorá pomocou kompozície využíva objekty typu `NeuronMatrix` a `Context`. Tento prístup umožňuje vytvorenie siete s ľubovoľnou mriežkou neurónov – s ľubovoľným rozmerom a tiež s ľubovoľnou funkciou vzdialenosti. Implementovali sme dvojrozmernú obdĺžnikovú mriežku `Matrix2D` s euklidovskou vzdialenosťou neurónov.

Trieda `SimpleContext` implementuje kontext pre sieť RecSOM, teda i-ta zložka kontextového vektora zodpovedajúca neurónu n_i má hodnotu $e^{-d_i(t)}$. Kontext je možné vynulovať volaním metódy `resetContext()` (napríklad po ukončení každej sekvencie).

Rozhranie `Neuron` musia implementovať všetky objekty, ktoré budú použité ako neuróny v sieti RecSOM. Rozhranie obsahuje metódy pre funkcie vzdialenosti od vstupu a kontextu, ako aj ich parciálne derivácie. Vďaka tomu je možné vytvoriť sieť RecSOM tvorenú neurónmi s ľubovoľnými funkciami vzdialenosti a tým modifikovať aktiváciu vytvorenej siete. Sieť je teda univerzálna a možno ju použiť pre rôzne typy neurónov. Obsahuje ďalej metódy pre zmenu kontextových a vstupných váh a dve metódy `createNewNeuron()` a `createNewNeuron(int, int)`, ktoré slúžia na klonovanie neurónov. Objekty typu `Neuron` implementujú návrhový vzor *Prototyp* (Pecinovský, 2007). Tento vzor sa používa vtedy, keď potrebujeme v zdrojovom kóde narábať s objektami nejakého typu, ktorý budeme poznať až neskôr. Vytvoríme preto pre tieto objekty spoločné rozhranie alebo triedu – ich prototyp a pomocou neho k objektom pristupujeme. Tento prototyp nám umožňuje tiež vytváranie nových inštancií podľa našich požiadaviek. Vďaka použitiu tohoto vzoru je možné vytvoriť sieť RecSOM pozostávajúcu z neurónov, ktoré sme napríklad obdržali zo siete alebo prečítali zo súboru.



Obr. 3.5: Diagram hlavných tried *RecSOM*

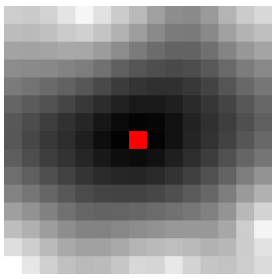
Konkrétnou implementáciou rozhrania `Neuron` je trieda `SimpleNeuron`. Implementuje neurón s euklidovskou vzdialenosťou od vstupu aj kontextu.

Trénovanie siete RecSOM nie je implementované priamo v objekte `RecSOM`, ale je extrahované do osobitného objektu. Rozhranie `RecSOMTrainer` združuje objekty, ktoré môžu byť použité na trénovanie siete RecSOM. Toto rozdelenie sme zvolili preto, aby sme oddelili model siete RecSOM od spôsobu jeho trénovania. Môžeme tak vytvoriť jednu sieť RecSOM a na jej trénovanie použiť ľubovoľný objekt implementujúci rozhranie `RecSOMTrainer`. Ak by sme vytvorili metódu pre trénovanie siete, museli by sme pre zvolenie iného algoritmu trénovania vytvárať jej potomka. To by jednak znamenalo množstvo nepotrebných tried a prišli by sme aj o možnosť zvoliť si algoritmus trénovania danej siete až počas behu programu. Hebbovský algoritmus trénovania sme implementovali v triede `HebbianTrainer`. Implementovali sme aj paralelnú verziu hebbovského učenia v triede `ParallelHebbianTrainer`, kde bol výber víťaza a počítanie zmien váh paralelný. Tento prístup však nepriniesol signifikantné urýchlenie, pretože úlohy počítané paralelne sú príliš jednoduché a veľa času strácame na réžii vlákien.

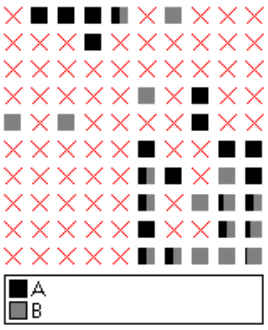
Vizualizačná časť knižnice

Pri analýze správania modelu RecSOM sú veľmi užitočné vizuálne informácie popisujúce vlastnosti siete. Knižnica *RecSOM* preto obsahuje niekoľko typov vizualizácií, ktoré je možné jednoducho aktivovať a deaktivovať. Výsledné obrázky môžu byť uložené do súborov.

- **Vizualizácia aktivácie siete.**
  Výstupný obrázok zodpovedá dvojrozmernej mriežke neurónov, kde každý neurón je zafarbený odtieňom šedej farby podľa jeho aktivácie pre daný vstup. Biela farba zodpovedá najnižšej aktivácii, čierna farba najvyššej. Víťazný neurón je zafarbený červenou farbou.

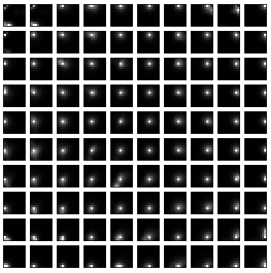


- **Vizualizácia rozdelenia kategórií**

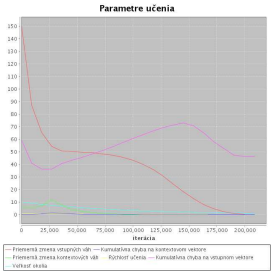
  Pri použití siete RecSOM na klasterizačné úlohy sledujeme schopnosť siete odlíšiť sekvencie rôznych kategórií, zaujíma nás preto distribúcia víťazov pre jednotlivé kategórie v mriežke siete. Vizualizácia kategórií vygeneruje pre zadané kategórie obrázok zodpovedajúci mriežke neurónov, v ktorom je každý neurón zafarbený farbami tých kategórií, pre ktorých prvky je víťazom. Pomer zafarbenia farbami kategórií zodpovedá pomeru počtu prvkov kategórií, pre ktoré je neurón víťazom. Neaktívne neuróny sú zakreslené červeným krížikom.

- **Vizualizácia distribúcie váhových vektorov**

  Výstupný obrázok pozostáva z väčšieho počtu podobrázkov, z ktorých každý zodpovedá jednému komponentu váhového vektora (pre vstupné alebo kontextové váhy). V tomto podobrázku je zobrazená mriežka neurónov, kde každý neurón je zafarbený odtieňom šedej farby podľa veľkosti daného komponentu váhového vektora neurónu. Biela farba značí najvyššiu hodnotu komponentu vektora, čierna najnižšiu. Váhové vektory neurónov sú pred vizualizáciou lineárne preškálované do intervalu $\langle 0, 1 \rangle$ cez jednotlivé komponenty.

- **Vizualizácia priebehu parametrov učenia**

  Vývoj parametrov učenia (rýchlosť učenia, veľkosť okolia, kumulatívna chyba, . . .) v čase počas trénovania siete je možné zobraziť do grafov. V jednom grafe môže byť jeden alebo viac zvolených parametrov podľa používateľových preferencií. Tieto grafy sú generované s použitím knižnice *JFreeChart*.

Ukladanie siete do databázy

Po natrénovaní siete by používateľ knižnice mal mať možnosť túto sieť uložiť do nejakej perzistentnej pamäte a neskôr ju odtiaľ načítať. Implementovali sme preto funkcionalitu pre ukladanie siete do súboru na pevnom disku a tiež pre ukladanie siete do databázy.

Pre ukladanie na pevný disk sme zvolili prístup zabudovaný v jazyku Java. Objekty, ktorých stav môže byť uložený, implementujú rozhranie `Serializable`. Tieto objekty je možné pomocou výstupného prúdu `ObjectOutputStream` napríklad zapísať do súboru alebo poslať cez sieť. Implicitný spôsob zápisu objektov je automatický a nie je nutné implementovať žiadne nové metódy pre zápis objektu. Pri zápise sa jednoducho do údajového prúdu zapíšu hodnoty všetkých atribútov objektu – primitívne typy sa zapíšu priamo, objektové typy musia tiež implementovať rozhranie `Serializable`. Zápis objektu, ktorého niektorý atribút neimplementuje rozhranie Serializable, vyvolá výnimku. Niektoré atribúty objektu však nemá význam zapisovať na disk a je potrebné vykonať ich inicializáciu aj po načítaní objektu z údajového prúdu (pri načítaní sa nevolá konštruktor objektu, ktorý by inicializoval nezapísané atribúty). Tieto atribúty označíme v deklarácii kľúčovým slovom `transient`. Ak vyžadujeme aj ich inicializáciu, potom musí trieda, ktorá ich obsahuje, implementovať aj rozhranie `ObjectInputValidation` a metódy `validateObject()` a `readObject()`. Každá trieda obsahuje statickú premennú `serialVersionUID`, ktorá je automaticky zmenená pri každej zmene tejto triedy. Tento mechanizmus stráži konzistenciu zapisovaných a čítaných objektov, pretože pri pokuse o načítanie objektu s inou hodnotou premennej `serialVersionUID`, ako je hodnota práve zavedenej triedy, vyvolá výnimku. Takéto správanie môže v našom prípade spôsobiť problém, pretože ak napríklad mierne zmeníme implementáciu triedy pre neurón (napríklad zmeníme aktivačnú funkciu), potom sa nám už nepodarí načítať zo súboru sieť RecSOM uloženú pred touto zmenou.

Implementovali sme preto aj iný prístup k ukladaniu siete – ukladanie do databázy. Rozhodli sme sa pre použitie ORM knižnice *Hibernate*, ktorá prináša množstvo výhod. Táto knižnica tvorí rozhranie medzi databázovou vrstvou a

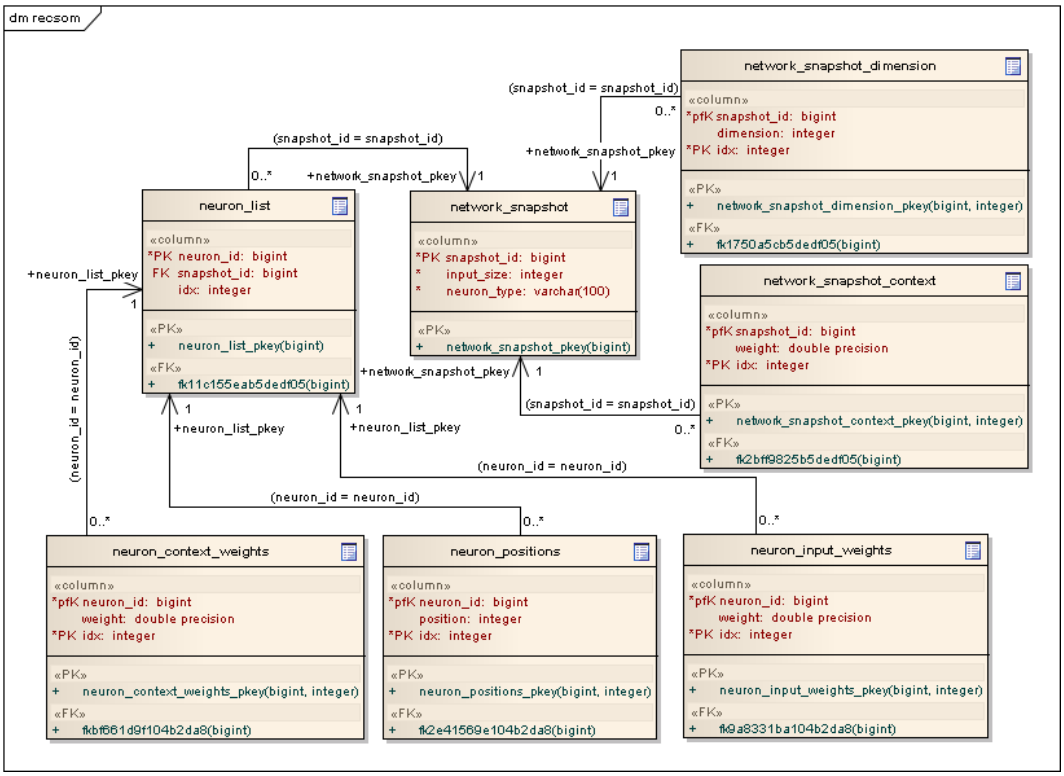
aplikáciou a umožňuje tak jednoducho mapovať perzistentné objekty programu priamo na databázové tabuľky. Z pohľadu aplikácie je rozhranie vždy rovnaké, bez ohľadu na použitú databázu. Aplikácia je teda univerzálna a jej používateľ má možnosť zvoliť si databázu podľa svojich požiadaviek.

Knižnica *Hibernate* je konfigurovateľná pomocou konfiguračných súborov vo formáte XML. Základnými nastaveniami je typ databázy a nastavenie prístupu k databázovému serveru. V ďalších súboroch sa špecifikujú mapovania tried programu na databázové tabuľky. Atribúty perzistentných tried sú priradené k jednotlivým stĺpcom databázovej tabuľky prislúchajúcej k danej triede. Je možné špecifikovať typ atribútov ako niektorý z primitívnych typov (integer, double, string). Zaujímavejšou možnosťou je špecifikácia typu objekt, kedy sa knižnica *Hibernate* sama postará o vytvorenie potrebných indexov a kľúčov v databázovej schéme. Je možné pohodlne pracovať aj s kolekciami objektov, kedy knižnica dokáže opäť samostatne vygenerovať potrebné tabuľky a závislosti v databázovej schéme.

Po zadefinovaní mapovania pre jednotlivé triedy knižnica vytvorí vo vybranej databáze potrebné tabuľky a následne je schopná synchronizovať objekty v pamäti programu s databázou.

Vytvorili sme rozhranie `RecSOMPersister`, ktoré združuje objekty schopné zapísať ľubovoľnú sieť `RecSOM` na nejaké perzistentné úložisko.

V balíku `ann.recsom.persisters.hibernatepersister` sa nachádza trieda `HibernateRecSOMPersister`, ktorá obsahuje implementáciu pre ukladanie siete do databázy s využitím popísanej knižnice *Hibernate*. V tomto balíku sa ďalej nachádzajú triedy, pomocou ktorých je možné reprezentovať celý stav siete RecSOM v ľubovoľnom čase. Schéma databázových tabuliek, ktorá bola vygenerovaná knižnicou *Hibernate* na základe nami zadefinovanej štruktúry v konfiguračných súboroch je zobrazená na obrázku 3.6. Trieda `HibernateRecSOMPersister` poskytuje tiež možnosť pripojenie na objekt typu `RecSOMTrainer` a počas trénovania dokáže po každej iterácii uložiť aktuálnu podobu siete.



Obr. 3.6: Entitno-relačný diagram *RecSOM*

3.1.6 Knižnica *CACLA*

Knižnica *CACLA* obsahuje implementáciu rovnomenného algoritmu učenia s posilňovaním, pomocných objektov a rôznych grafických výstupov.

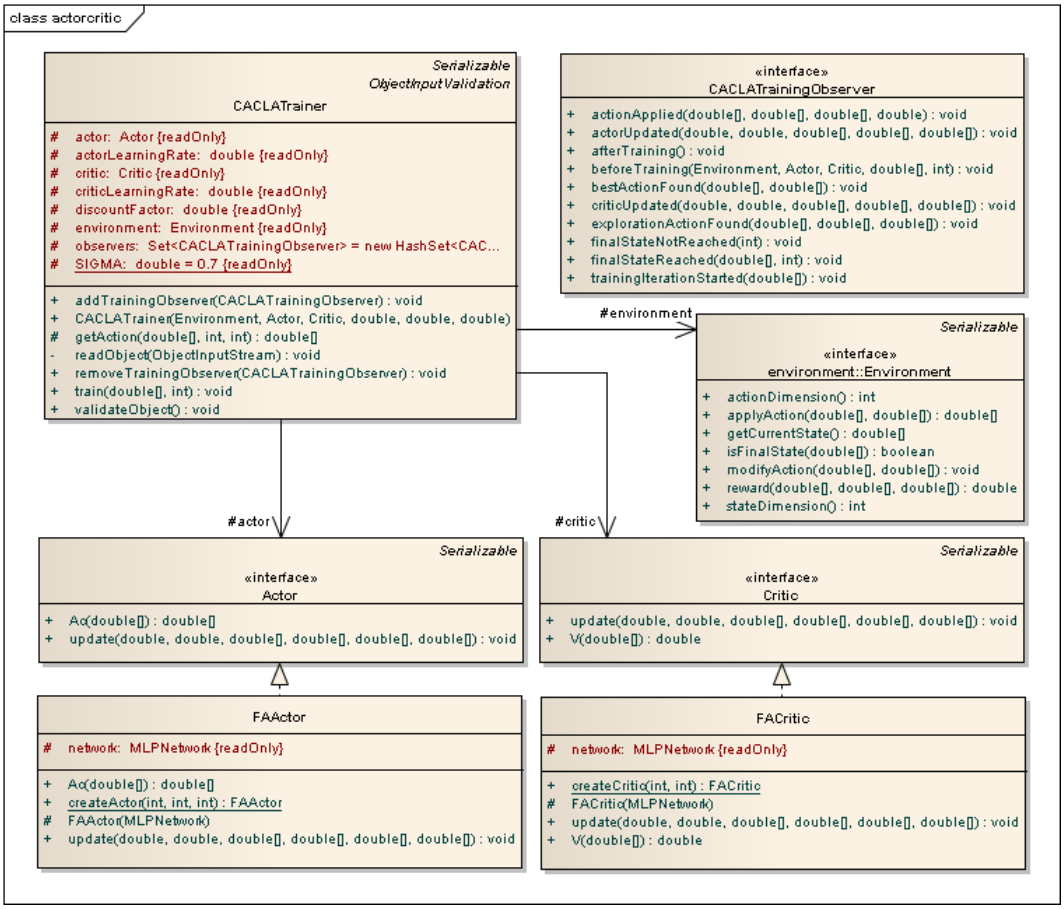
UML diagram hierarchie základných rozhraní knižnice je zobrazený na obrázku 3.7. Základné entity algoritmu – aktér a kritik, implementujú rozhrania `Actor` a `Critic`. Tieto rozhrania definujú metódy aktéra a kritika na vysokej úrovni abstrakcie a preto je možné použiť ako aproximátor funkcie v aktérovi a kritikovi v podstate ľubovoľný model. Požadované metódy sú iba metódy pre vygenerovanie akcie (resp. ohodnotenie stavu) a úprava parametrov aproximátora.

Konkrétnej implementácii aktéra a kritika, využívajúcej model viacvrstvovej doprednej siete, zodpovedajú triedy `FAActor` a `FACritic`. Model viacvrstvovej doprednej siete je implementovaný v balíku `ann.mlp`. Učenie tejto siete je realizované pomocou algoritmu spätného šírenia chyby – *back-propagation* (Haykin,

1998).

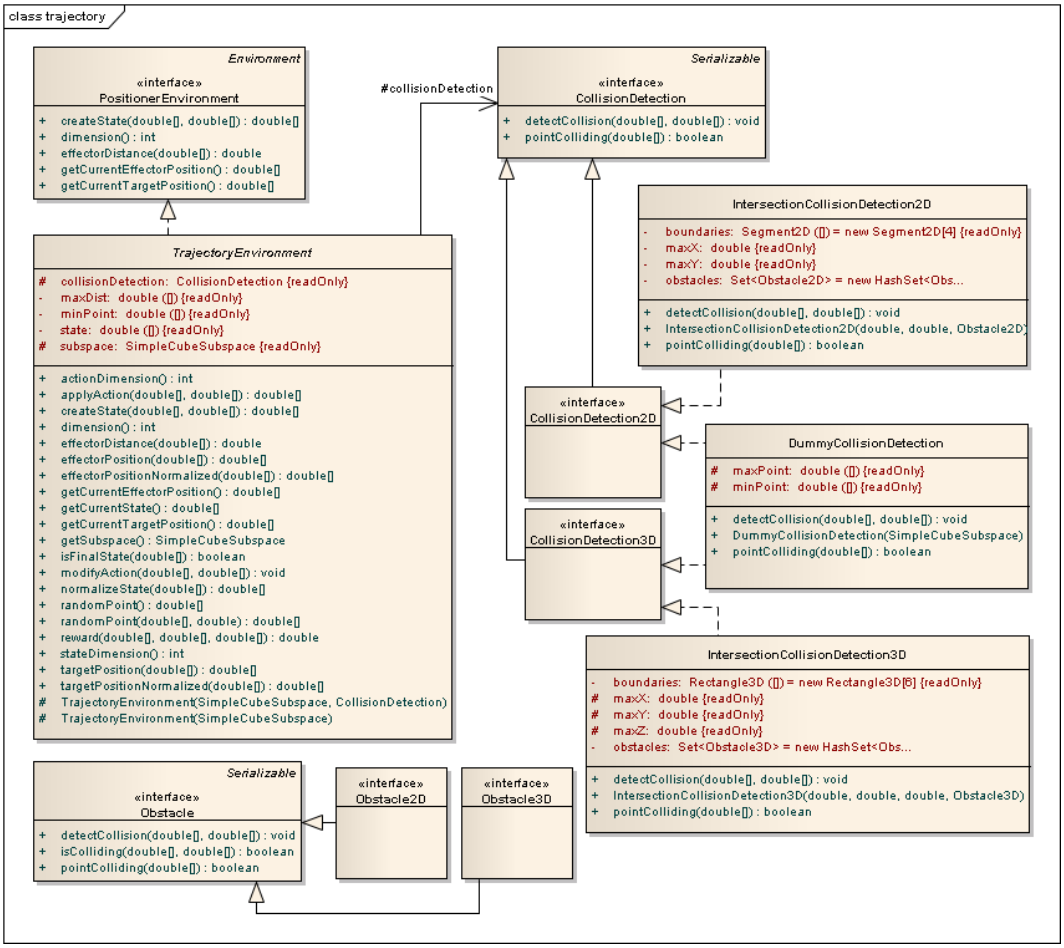
Prostredie, v ktorom sa agent nachádza, je reprezentované rozhraním `Environment`. Toto rozhranie obsahuje opäť iba základné metódy, ako je práca so stavmi, aplikovanie akcie a pridelenie odmeny. Všetky ostatné detaily prostredia sú ponechané na používateľa knižnice, aby bolo riešenie čo možno najuniverzálnejšie a použiteľné aj na riešenie iných problémov.

Trénovanie modelov aktéra a kritika má za úlohu trieda `CACLATrainer`, ktorá môže byť použitá na trénovanie ľubovoľného aktéra typu `Actor` a kritika typu `Critic`. Je teda opäť univerzálna. Priebeh trénovania modelov je možné sledovať pomocou pozorovateľa implementujúceho rozhranie `CACLATrainingObserver`.



Obr. 3.7: Diagram základných tried knižnice *CACLA*

V balíku `rl.environment.trajectory` sú implementované objekty reprezentujúce prostredie pre pohyb agenta (v našom prípade robotického ramena) v priestore. Abstraktná trieda `TrajectoryEnvironment` (obrázok 3.8) obsahuje spoločnú funkcionalitu prostredia pre pohyb agenta. Pomocou kompozície využíva triedu `CollisionDetection`, ktorá slúži na detekciu kolízií pri pohybe agenta v priestore. Detekcia kolízií sa používa na ohraničenie pracovného priestoru agenta a na riešenie kolízií s prekážkami. Pri kolízii modifikuje nové súradnice agenta tak, aby zodpovedali miestu stretnutia agenta s prekážkou. Prekážky v priestore sú reprezentované rozhraním `Obstacle`.



Obr. 3.8: Diagram tried prostredia trajektórií

Implementovali sme dva spôsoby detekcie kolízií:

1. detektor `DummyCollisionDetection` je jednoduchý detektor, ktorý reprezentuje pracovný priestor obdĺžnikového tvaru (v trojrozmernom prípade kváder), ktorý pri snahe agenta vystúpiť z pracovného priestoru do bodu $[n_x, n_y, n_z]$ vráti ako novú pozíciu bod

$$[\min(\max(n_x, 0), width), \min(\max(n_y, 0), height), \min(\max(n_z, 0), depth)]$$

2. detektor `IntersectionCollisionDetection` reprezentuje taktiež pracovný priestor obdĺžnikového (kvádrového) tvaru, avšak pri pokuse agenta o vystúpenie z pracovného priestoru vypočíta presné súradnice nárazu agenta o hranicu priestoru. Na výpočet priesečníku používa knižnicu *Math*.

Objekty pracovného prostredia sú rozdelené podľa dimenzie na dvoj a trojrozmerné, aby bola zaistená ich kompatibilita.

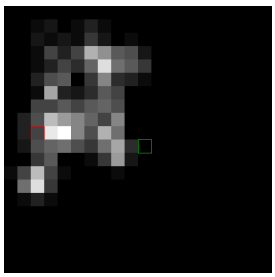
Vizualizačná časť knižnice

Počas hľadania vhodných parametrov učenia algoritmu CACLA sme využili niekoľko grafických výstupov knižnice, ktoré nám pomohli identifikovať a odstrániť niekoľko problémov. Triedy generujúce grafické výstupy sa nachádzajú v balíku `rl.actorcritic.visualizers`.

Prvú skupinu výstupov tvoria grafy zachytávajúce priebeh parametrov učenia počas trénovania, ktoré boli taktiež vytvorené s použitím knižnice *JFreeChart*. V jednom grafe je opäť možné zobraziť jeden alebo viacero parametrov. Vytvorili sme preto rozhranie `ObserverChartSource`, ktoré rozširuje rozhranie `CACLATrainingObserver`. Každému parametru učenia potom zodpovedá samostatná trieda implementujúca toto rozhranie, ktorú je možné pripojiť k objektu typu `CACLATrainer`. Počas trénovania táto trieda zbiera informácie o vývoji sledovaného parametra a po skončení trénovania dokáže zozbierané informácie poskytnúť ako jednu dátovú sériu zobraziteľnú v grafe. Do vytváraného grafu potom pridáme ľubovoľný počet dátových sérií, ktoré sa v ňom vykreslia.

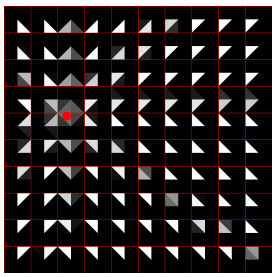
Druhú skupinu tvoria grafické výstupy zobrazujúce stav pracovného priestoru. Tieto výstupy pracujú iba v dvojrozmernom prípade a boli použité pri hľadaní vhodných parametrov učenia. Patria medzi ne:

- `StateDistributionVisualizer`

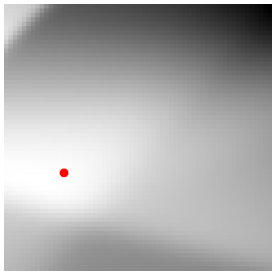
  Frekvenciu výskytu ramena v jednotlivých častiach pracovného priestoru pre prípad dvojrozmerného priestoru zobrazuje trieda `StateDistributionVisualizer`. Počas trénovania sleduje, ako často sa rameno vyskytne v tom ktorom segmente pracovného priestoru. Frekvenciu pre každý segment následne zobrazí v obrázku, kde biela farba zodpovedá najčastejšie navštevovanému stavu a čierna farba zodpovedá stavu navštevovanému najmenej. Zeleným štvorčekom je zobrazený počiatočný a červeným koncový bod trajektórie.

- `TrajectoryActionVisualizer`

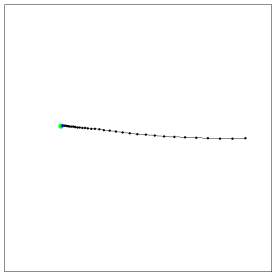
  Prevládajúci smer akcií, ktoré natrénovaný model CACLA v daných stavoch volí, je možné zobraziť pomocou triedy `TrajectoryActionVisualizer`. Pracovný priestor je rozdelený na zadaný počet segmentov. V rámci jedného segmentu sú rovnomerne zvolené počiatočné body, z ktorých necháme model ovládať rameno smerom k zvolenému cieľovému bodu. Po prejdení piatich krokov zastavíme generovanie trajektórie a vypočítame smer vektora z počiatočného bodu do bodu, v ktorom sa rameno nachádza po prejdení týchto krokov. Segment následne rozdelíme na osem častí a každú zafarbíme farbou vyjadrujúcou početnosť výberu daného smeru (biela farba zodpovedá najpočetnejšiemu smeru, čierna najmenej početnému). Celý obrázok teda vlastne zobrazuje prevládajúce smery, ktorými sa rameno začne pohybovať pri generovaní trajektórie do zvoleného cieľa (zakresleného červeným kruhom) zo všetkých častí pracovného priestoru.

- `TrajectoryCriticVisualizer`

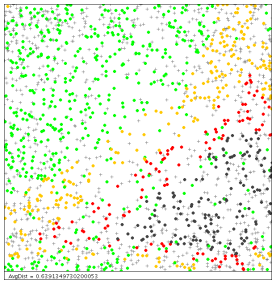
  Ohodnotenie stavov pracovného priestoru kritikom zobrazuje trieda `TrajectoryCriticVisualizer`. Pracovný priestor je opäť rozdelený na zvolený počet segmentov. Každý segment je následne pomocou kritika ohodnotený – bod zo stredu segmentu je považovaný za aktuálnu pozíciu ramena a je k nemu pripojená informácia o cieľovom bode. Aktivácia kritika pre zvolenú kombináciu aktuálnej pozície a cieľového bodu je zakreslená v obrázku (biela farba znamená najvyššie ohodnotenie, čierna najnižšie). Obrázok teda vlastne znázorňuje, ako kritik hodnotí jednotlivé stavy s ohľadom na cieľový bod.

- `TrajectoryDrawer`

  Proces vytvárania trajektórie je možné zobraziť pomocou triedy `TrajectoryDrawer`. Pre zvolený počiatočný a cieľový bod výstupný obrázok zachytáva celú trajektóriu, ktorú natrénovaný model CACLA generuje. Časť trajektórie medzi dvoma čiernymi bodmi zodpovedá posunu ramena v jednom kroku, maximálny počet krokov je obmedzený konštantou. Reálny koniec trajektórie je spojený farebnou čiarou so zvoleným cieľovým bodom, ktorý je zafarbený farbou podľa vzdialenosti medzi cieľovým bodom a bodom, do ktorého sa rameno po prejdení daného počtu krokov dostalo. Ak sa rameno dostane do euklidovskej vzdialenosti menšej ako 0.5, potom je cieľový bod zafarbený zelenou farbou. Oranžová farba zodpovedá vzdialenosti menšej ako 0.9, červená menšej ako 1.3, čierna vzdialenosti väčšej.

- `TargetsDistance`

  Schopnosť obsiahnuť jednotlivé časti pracovného priestoru je možné graficky zobraziť pomocou triedy `TargetDistance`. Z pracovného priestoru náhodne vyberie začiatočné a cieľové body, ktorých vzdialenosť je väčšia ako nastavená hranica. Následne pomocou natrénovaného aktéra vygeneruje trajektóriu medzi zvolenými bodmi. Na pozícii zvoleného cieľa v obrázku sa následne zakreslí farebný krúžok, ktorého farba opäť určuje vzdialenosť reálneho konca vygenerovanej trajektórie od cieľa (farebné označenie je rovnaké, ako v triede `TrajectoryDrawer`). Šedým krížikom môžu byť v obrázku zaznačené cieľové body použité počas trénovania. Tento grafický výstup teda zobrazuje distribúciu trénovacích cieľových bodov a umožňuje analyzovať schopnosť modelu generovať trajektórie končiace v rôznych častiach priestoru.

Tieto grafické výstupy sú jednoducho prístupné pomocou statických metód triedy `VisualizerUtils`, ktorá implementuje návrhový vzor *Fasáda* (Pecinovský, 2007). Metódy tejto triedy zjednodušujú prístup k vnútornej funkcionalite balíka `rl.actorcritic.visualizers`, používateľ teda nemusí poznať presný postup vygenerovania daného grafického výstupu pomocou vnútorných tried balíka. Namiesto toho zavolá iba jednu metódu fasády, ktorá implementuje tento postup a sprístupňuje tak funkcionalitu balíka.

3.1.7 Grafická časť aplikácie

V balíku `dt.scene` sa nachádza implementácia objektov pracovného priestoru ramena. Pracovný priestor je reprezentovaný rozhraním `Scene`, ktoré definuje ramenom dosiahnuteľný priestor tvaru kvádra. Kompozíciou využíva objekty reprezentujúce kamery scény – rozhranie `Camera` a rameno umiestnené v scéne – rozhranie `RobotArm`.

Základná spoločná implementácia pre kameru scény je implementovaná v

triede `ACamera` a kompletná implementácia funkcionality kamery je v triede `SimpleCamera`. Túto kameru je možné umiestniť na ľubovoľnú pozíciu v pracovnom priestore a ďalej jej nastaviť horizontálny sklon a vertikálne natočenie. Kamera si následne vytvorí maticu transformácie, ktorou transformuje pozíciu ramena a cieľa z trojrozmerného súradnicového systému pracovného priestoru do dvojrozmerného súradnicového systému kamery.

Vytvorenú scénu je možné vykresliť do objektu typu `Canvas` použitím triedy implementujúcej rozhranie `SceneVisualizer`. V balíku `dt.visual.opengl` je implementované vykresľovanie scény pomocou knižnice *JOGL* – implementácia grafickej knižnice *OpenGL* pre platformu Java. Trieda `GLSceneVisualizer` s použitím tried `ArmVisualizer`, `BallVisualizer`, `CameraVisualizer` a `WorkingSpaceVisualizer` dokáže vykresliť celú scénu do objektu typu `Canvas`. Použitie knižnice *OpenGL* sme zvolili pre zjednodušenie implementácie, aby sme nemuseli v aplikácii riešiť problém vykresľovania trojrozmerného priestoru.

3.1.8 Program pre spúšťanie simulácií

Počas práce s modelmi CACLA a RecSOM bolo potrebné vytvoriť mnoho simulácií, ktoré boli v niektorých prípadoch výpočtovo náročnejšie. Vytvorili sme preto mechanizmus pre automatický výpočet ľubovoľných simulácií.

Trieda `SimulationServer` poskytuje služby pre simulácie, ktoré budú spúšťané. Počas svojej inicializácie zavedie knižnicu *Math* a vytvorí fond výpočtových vlákien – zadaný počet lokálnych vlákien a prípadne tiež zvolené sieťové výpočtové vlákna. Vytvorí tiež spojenie s databázovým serverom, do ktorého je možné ukladať výsledky simulácií.

Jednotlivé simulácie implementujú rozhranie `Simulation` s jedinou metódou `execute()`, v ktorej musia vytvoriť objekty pre počítané úlohy typu `ComputationalTask` a od fondu `WorkerPool` vyžiadať výpočtové vlákna, pomocou ktorých budú tieto úlohy vypočítané. Spoločná funkcionalita pre simulácie je implementovaná v triede `SimpleMultithreadSimulation`.

Každá simulácia má pridelenú inštanciu objektu `SimulationProper-`

`ties`, z ktorého môže získať koreňový adresár, v ktorom bola spustená. Do neho potom potenciálne môže ukladať svoje výstupy v podobe súborov. Taktiež jej server sprístupňuje pripojenie k databáze, ktoré môže použiť na ukladanie výsledkov.

Postup vytvorenia novej simulácie spočíva v implementácii algoritmu simulácie do triedy implementujúcej rozhranie `Simulation` a jej skompilovaní. Informácie o simulácii je potrebné následne popísať pomocou XML súboru, v ktorom zadefinujeme jej meno a uvedieme všetky triedy, ktoré tvoria simuláciu. Všetky skompilované triedy spolu s týmto XML súborom následne prenesieme do adresára, ktorý sleduje spustená inštancia simulačného servera. Simulačný server po objavení nevypočítanej simulácie najprv zavedie potrebné triedy do *Java Virtual Machine*, v ktorej sám beží. Následne spustí výpočet simulácie, ktorý je zahájený v momente, keď sa vo fonde `WorkerPool` nachádza nejaké voľné vlákno.

Kapitola 4

Výsledky

V tejto kapitole opíšeme správanie zvoleného modelu pre problém generovania trajektórie. Stručne načrtneme problémy, ktoré bolo potrebné riešiť počas hľadania vhodného modelu. Zhrnieme dosiahnuté výsledky a analyzujeme vplyv voľby parametrov učenia a spôsobu trénovania na schopnosti a vlastnosti modelu.

4.1 Zjednodušený model prostredia

Model CACLA sme najprv skúmali na zjednodušenom probléme generovania trajektórie ramena v dvojrozmernom prostredí. Stavom, ktorý reprezentuje konfiguráciu pracovného priestoru v čase t, rozumieme vektor

$$s_t = \left[x_t^{act}, y_t^{act}, x_t^{tgt}, y_t^{tgt}\right]$$

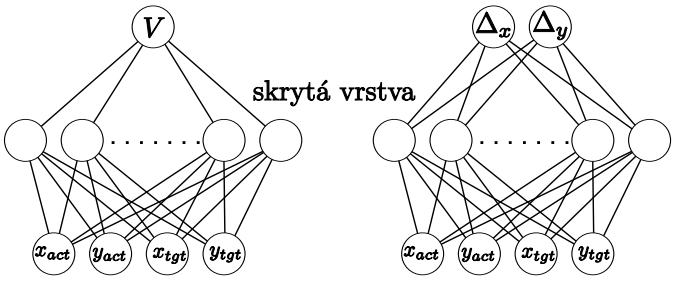
kde súradnice x_t^{act} a y_t^{act} určujú aktuálnu pozíciu ramena a súradnice x_t^{tgt} a y_t^{tgt} zodpovedajú cieľovému bodu. Súradnice určujú pozíciu v karteziánskom súradnicovom systéme pracovného priestoru. V tejto fáze sme teda ešte nepoužili perceptuálne vnímanie cieľa a polohy ramena pomocou dvoch kamier. Akcia vygenerovaná modelom CACLA v čase t je reprezentovaná vektorom $a_t = [\Delta_x, \Delta_y]$ a prechod do nového stavu s_{t+1} definujeme ako

$$s_{t+1} = \left[x_t^{act} + \Delta_x, y_t^{act} + \Delta_y, x_t^{tgt}, y_t^{tgt}\right]$$

Tento zjednodušený model sme zvolili z dôvodu jednoduchšej vizualizácie a analýzy správania modelu. Jeho rozšírenie do trojrozmerného priestoru je triviálne a nevyžaduje žiadnu principiálnu zmenu modelu. Rozšírenie modelu do trojrozmerného priestoru s nahradením súradníc $[x,y,z]$ za signál z kamier $\left[e_1^x, e_1^y, e_2^x, e_2^y\right]$ opäť nevyžaduje žiadnu principiálnu zmenu modelu, preto je možné začať analýzu modelu CACLA v tomto zjednodušenom prostredí.

4.2 Zvolené aproximátory funkcií

Pri experimentoch sme ako aproximátory funkcií pre kritika aj aktéra zvolili doprednú sieť s jednou skrytou vrstvou. Na skrytej vrstve sme použili 20 spojitých perceptrónov so sigmoidálnou aktivačnou funkciou. Použili sme aj siete s 15, 25 a 30 skrytými perceptrónmi a použili sme tiež perceptróny s aktivačnou funkciou hyperbolický tangens. Tieto siete dosahovali rovnaké výsledky a počet skrytých neurónov v tomto intervale nevplýval na vlastnosti modelov. Vyšší počet neurónov však už neviedol ku konvergentným modelom, pretože aproximátor funkcie obsahoval príliš veľké množstvo trénovateľných parametrov.



Obr. 4.1: Aktér a kritik zjednodušeného prostredia

Na výstupnej vrstve boli použité lineárne perceptróny. Výstupný perceptrón kritika mal aktivačnú funkciu $f(net) = net$. Aktiváciu výstupných perceptrónov aktéra sme väčšinou lineárne znížili, použili sme aktivačnú funkciu $f(net) = \frac{net}{10}$. Takéto preškálovanie vplýva na voľbu parametrov učenia, tomuto vplyvu sa budeme detailnejšie venovať v nasledujúcich častiach.

Učenie modelov aktéra a kritika je realizované algoritmom spätného šírenia chyby. V prípade kritika v čase t použijeme ako chybu na výstupnej vrstve rozdiel medzi ohodnotením stavu $V_t(s_t)$ a odmenou $r_t + \gamma V_t(s_{t+1})$. V prípade aktéra ako chybu na výstupnej vrstve používame rozdiel medzi predikovanou akciou $[\Delta_x, \Delta_y]$ a lepšou akciou a_t vybranou počas explorácie.

4.3 Schémy trénovania a funkcia odmeny

Uvažovali sme štyri rôzne schémy trénovania modelu:

1. schéma *one-to-one* – generovanie trajektórie z fixného začiatočného do fixného cieľového bodu
2. schéma *many-to-one* – generovanie trajektórie z ľubovoľného začiatočného bodu do fixného cieľového bodu
3. schéma *one-to-many* – generovanie trajektórie z fixného začiatočného do ľubovoľného cieľového bodu
4. schéma *many-to-many* – generovanie trajektórie z ľubovoľného začiatočného do ľubovoľného cieľového bodu

Počas trénovania zvolíme podľa vybranej schémy náhodný alebo fixný začiatočný a cieľový bod z pracovného priestoru a necháme rameno pohybovať v priestore. Ak sa rameno dostane do euklidovskej vzdialenosti menšej ako 0.1, potom túto epochu ukončíme, zvolíme podľa schémy nové body a spustíme novú epochu. Dĺžka trvania jednej epochy je obmedzená, ak sa teda rameno za zvolený počet krokov nepriblíži k cieľu, prerušíme epochu umelo.

Ako funkciu odmeny a trestu sme zvolili zápornú euklidovskú vzdialenosť ramena od cieľového bodu. Túto funkciu sme ďalej lineárne preškálovali do intervalu $[-0.5, 0.5]$. Zvolená funkcia odmeny nezohľadňuje vytváranie celej trajektórie, pretože je nejednoznačná. Odmena je rovnaká po celej kružnici (v trojrozmernom prípade na povrchu gule) so stredom v cieľovom bode.

4.4 Ohodnotenie natrénovaných modelov

Pri hľadaní vhodných parametrov učenia sme úspešnosť modelov vyhodnocovali nasledovne. Podľa zvolenej trénovacej schémy sme vytvorili zvolený počet trajektórií, na ktorých sme natrénovali model CACLA. Tento model sme následne testovali pomocou testovacích trajektórií opäť podľa zvolenej trénovacej schémy. Vypočítali sme priemernú vzdialenosť medzi cieľovým a reálnym koncovým bodom cez všetky testovacie trajektórie. Modelov CACLA sme natrénovali viacero a vypočítali sme strednú hodnotu a štandardnú odchýlku vzdialeností všetkých modelov.

Algorithm 3 Algoritmus ohodnotenia modelu

for $t = 1$ **to** `#modelov` **do**
  $\mathcal{M}_t \leftarrow$ `inicializuj_model()`
  for all $[x_{start}, y_{start}, x_{tgt}, y_{tgt}] \in Sch_{train}$ **do**
    `natrénuj_model(`$\mathcal{M}_t$, $[x_{start}, y_{start}, x_{tgt}, y_{tgt}]$`)`
  end for
  for all $[tst_{start}, tst_{tgt}] \in Sch_{test}$ **do**
    $real \leftarrow$ `spusti_model(`$\mathcal{M}_t$, tst_{start}, tst_{tgt}`)`
  end for
  $avgdist_{\mathcal{M}_t} = avg\left\{distance(tst_{tgt}, real)\right\}$
end for
$\mu \leftarrow avg_{\mathcal{M}_t}\left\{avgdist_{\mathcal{M}_t}\right\}$
$\sigma \leftarrow stdev_{\mathcal{M}_t}\left\{avgdist_{\mathcal{M}_t}\right\}$

4.5 Generovanie trajektórie v priestore 2D

Analýzu správania modelu CACLA pre zjednodušený problém generovania trajektórie sme začali trénovacou schémou *one-to-one* – snažili sme sa model naučiť generovať jedinú trajektóriu z bodu $[5,5]$ do bodu $[1,4.5]$.

Experimentálne sme našli parametre učenia, ktoré maximalizovali úspešnosť modelu pri generovaní trajektórií. Učenie s posilňovaním algoritmom CACLA

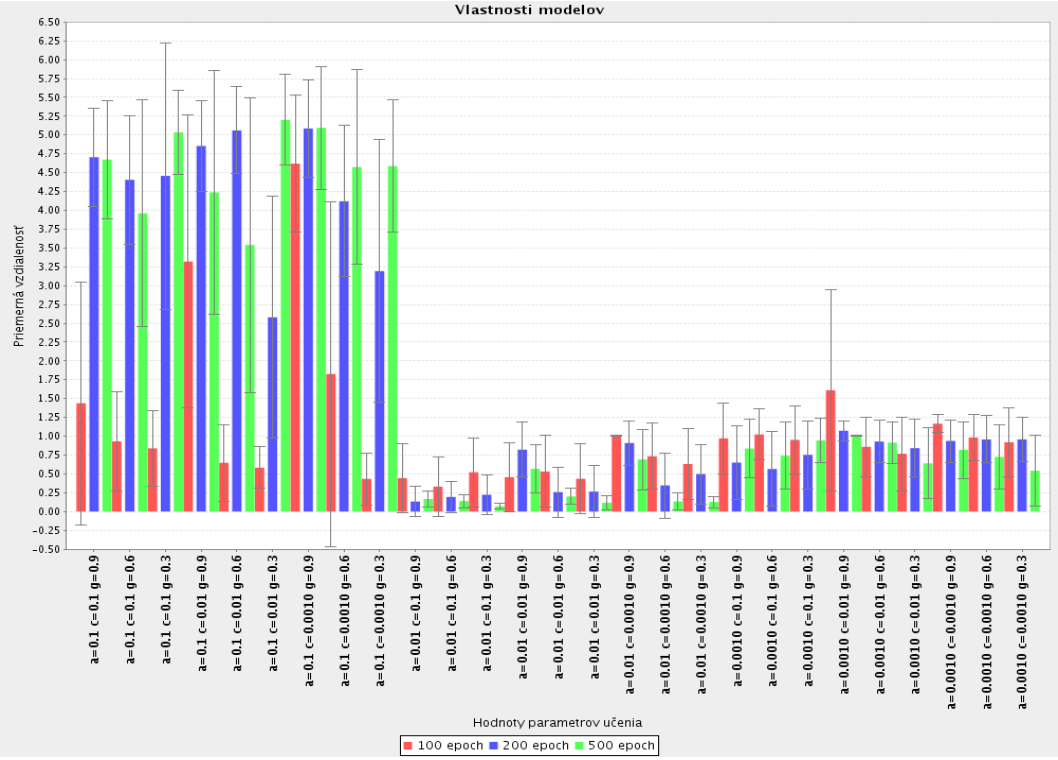
je enormne citlivé na voľbu parametrov, ktoré je zrejme nutné voliť špecificky pre každý problém. Veľké rozdiely v kvalite natrénovaných modelov spôsoboval taktiež počet trénovacích epoch.

Pri experimentoch sme používali Gassovskú exploráciu s parametrom $\sigma = 0.7$. Rýchlosť učenia kritika a aktéra sme uvažovali v krokoch 0.1, 0.01 a 0.001. Diskontný faktor γ v rovnici 2.3 sme uvažovali 0.9, 0.6, 0.3 a počet epoch (trajektórií) 100, 200 a 500.

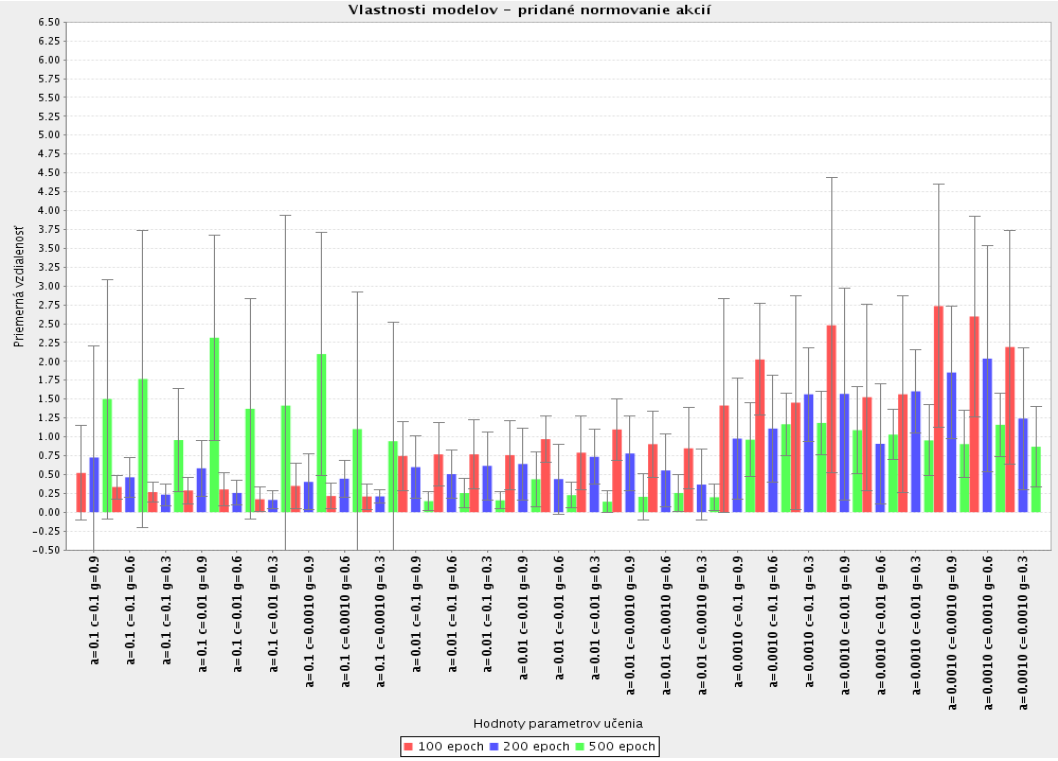
Modely sme ohodnotili podľa algoritmu 3 – nezávisle sme natrénovali 10 modelov s náhodne inicializovanými váhami. Trénovanie prebiehalo zvolený počet epoch – trajektórií začínajúcich v bode $[5, 5]$ s cieľom $[1, 4.5]$. Každá epocha trvala maximálne 2000 krokov. Kvalitu modelu sme potom verifikovali ohodnotením vzdialenosti konca trajektórie po maximálne 1000 krokoch od cieľového bodu. V prípade schémy *one-to-one* postačuje verifikácia na jednom opakovaní tejto trajektórie, pretože správanie modelu po natrénovaní je už deterministické. Strednú hodnotu a štandardnú odchýlku tejto vzdialenosti cez daných 10 modelov sme zaznamenali v grafe na obrázku 4.2(a). Na horizontálnej osi grafu sú opísané parametre modelov – rýchlosť učenia aktéra je označená písmenom *a*, rýchlosť učenia kritika písmenom *c* a písmenom *g* je označený diskontný faktor γ.

Z grafu 4.2(a) vyplýva, že modely sa boli schopné naučiť generovanie trajektórie až od rýchlosti učenia aktéra $a = 0.01$ a menšej. Najlepšie modely sme dostali pre rýchlosti učenia kritika $c = 0.1$ a $c = 0.01$. Pri týchto rýchlostiach učenia potreboval model viac trénovacích epoch, najlepšie vlastnosti dosahovali modely s 500 trénovacími epochami.

Pri najlepších modeloch v tomto grafe je možné pozorovať zaujímavý trend. Modely trénované pri vyššom počte epoch dosahujú lepšie výsledky pri nižších hodnotách diskontného faktora γ (resp. *g*). Predpokladáme, že je to spôsobené tým, že pri tomto type učenia agent dostáva odmenu po každom kroku. Nepotrebuje teda ohodnocovať svoje stavy s ohľadom na predikované zlepšenie budúcich stavov. Postačujúcou informáciou pre ohodnotenie aktuálneho stavu je odmena pridelená v tomto stave a ohodnotenie nasledujúceho stavu. Ohodnotenia ďalších stavov do budúcnosti sa už takmer zabúdajú.



(a) Model bez normovania akcií



(b) Model s pridaným normovaním akcií

Obr. 4.2: Priemerná vzdialenosť modelov od cieľa pri schéme *one-to-one*

Model CACLA sa nám teda podarilo úspešne použiť pre problém generovania trajektórie podľa schémy *one-to-one* v dvojrozmernom prípade. Ukázalo sa, že je možné (niekedy dokonca potrebné) algoritmu CACLA pomôcť vhodnou úpravou explorovaných akcií a tiež preškálovaním výstupnej aktivácie aktéra. Tieto metódy preto detailne opíšeme v nasledujúcej časti.

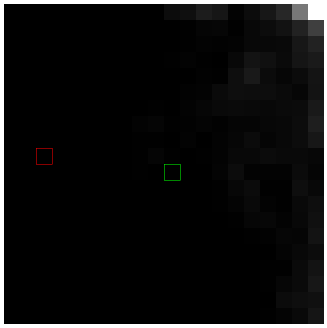
4.5.1 Normovanie akcií počas explorácie

Rýchlosť konvergencie modelov je možné signifikantne urýchliť vhodnými úpravami akcií a, ktoré vyberáme podľa predikcie aktéra Ac v procese explorácie.

Aktiváciu výstupných lineárnych perceptrónov v doprednej sieti aktéra (obrázok 4.1) sme najprv znížili použitím aktivačnej funkcie

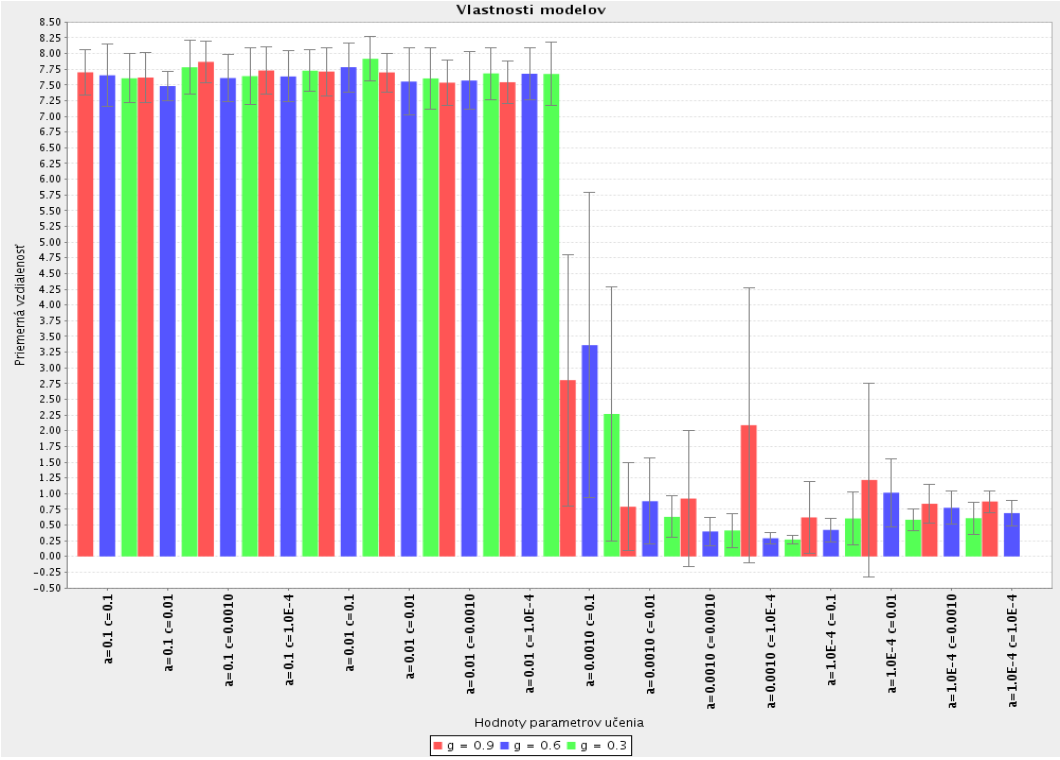
$$f(net) = \frac{net}{10}$$

aby sme zmenšili veľkosť akcií pri ešte nenatrénovanom modeli. Eliminovali sme tak prípady, keď aktér predikuje príliš veľké akcie a rameno tým dorazí do niektorého okrajového bodu pracovného priestoru, v ktorom uviazne (obrázok 4.3).

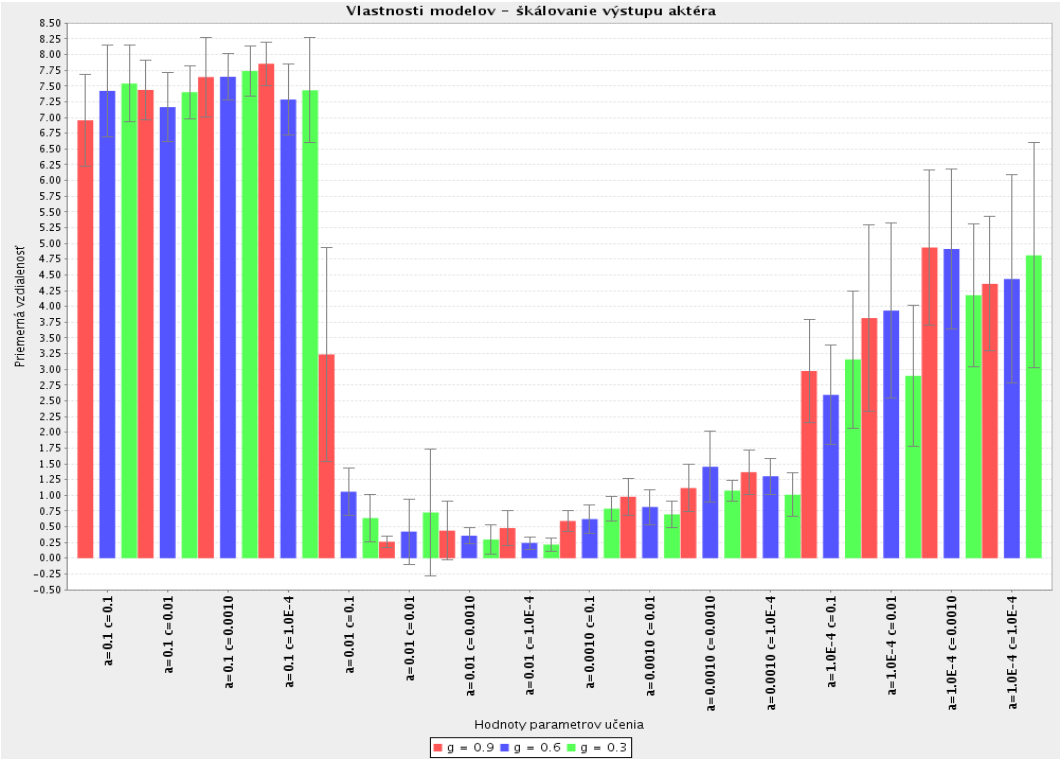


Obr. 4.3: Uviaznutie ramena v okrajovom bode

Vplyv zníženia veľkosti akcií na vlastnosti modelov je zobrazený na obrázku 4.4. Modely bez znížených akcií konvergovali až pre rádovo menšie hodnoty rýchlostí učenia aktéra. Ukazuje sa tu iba malá robustnosť učenia s posilňovaním vzhľadom na konkrétne hodnoty volených parametrov – ak používame aktéra, ktorého akcie majú napríklad číselne väčšie hodnoty, potom musíme voliť iné hodnoty rýchlosti učenia, ako napríklad v prípade číselne menších hodnôt. Parametre učenia sú teda závislé aj od konkrétneho problému, ktorý pomocou učenia s posilňovaním chceme riešiť. Model je teda relatívne citlivý na numerické hodnoty stavov a akcií, s ktorými pracuje.



(a) Vzdialenosť modelov bez znížených akcií

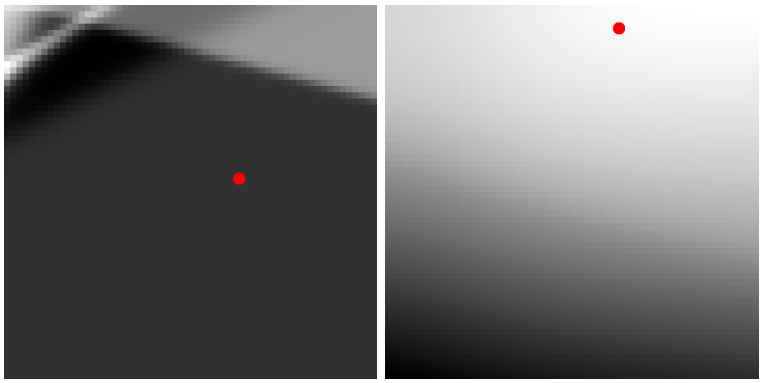


(b) Vzdialenosť modelov so znížený mi akciami

Obr. 4.4: Vplyv zníženia veľkosti akcií na vlastnosti modelov pri schéme *one-to-one*

Akcie vyberané v procese explorácie následne modifikujeme podľa ich vzdialenosti od cieľového bodu. Ak sa rameno nachádza vo vzdialenosti väčšej ako 1, potom veľkosť vybranej akcie *a* normujeme na 1. V opačnom prípade normujeme akciu na 0.3 iba v prípade, ak jej norma je väčšia ako 0.3. Nútime tak rameno robiť väčšie kroky, pokiaľ je dostatočne ďaleko od cieľového bodu. Keď sa k cieľu priblíží, nedovolíme mu naopak robiť príliš veľké akcie, aby sa z okolia cieľa rýchlo nevzdialil.

Vplyv uvedeného normovania akcií na vlastnosti modelov je viditeľný v grafoch na obrázkoch 4.7, 4.2 a 4.10. V rozsahu hodnôt parametrov, v ktorých modely konvergovali, spôsobilo normovanie akcií rýchlejšiu konvergenciu, lepšie vlastnosti modelov a zaručilo väčšiu stabilitu trénovania pri rôznych počtoch trénovacích epoch. Kým modely bez normovaných akcií trénované počas 1000 epoch zväčša neboli schopné konvergencie, modely s normovanými akciami skonvergovali takmer vždy.



(a) Model bez normovaných akcií (b) Model s normovanými akciami

Obr. 4.5: Ohodnotenie stavov kritikom schéme *many-to-many*

Obrázok 4.5 (zhotovený podľa postupu opísaného v časti 3.1.6) zobrazuje ohodnotenie jednotlivých stavov kritikom pre model natrénovaný bez normovaných akcií a s normovanými akciami. Kritik modelu s normovanými akciami hodnotí stavy správne, okolie cieľového bodu vykazuje vyššie ohodnotenie, ako ostatné stavy. Ohodnotenie stavov klesá zhruba lineárne s rastúcou vzdialenosťou od cieľa. Naproti tomu ohodnotenie kritika modelu bez normovaných akcií je re-

latívne chaotické, sú tu viditeľné viaceré deliace priamky. Predpokladáme, že je to spôsobené numerickou nestabilitou modelu, pretože ak rameno vykonáva príliš malé akcie (teda malé posuny), potom sa nachádza príliš veľa iterácií po sebe vo veľmi blízkych stavoch. Kritik sa zrejme naučí tieto stavy ohodnocovať podľa príliš jemných rozdielov medzi stavmi. Stavy v blízkosti cieľa majú priradené nižšie ohodnotenie, ako iné stavy. Najvyššie hodnotené stavy sa nachádzajú v ľavom hornom rohu. Celkové ohodnotenie stavov pôsobí nespojite.

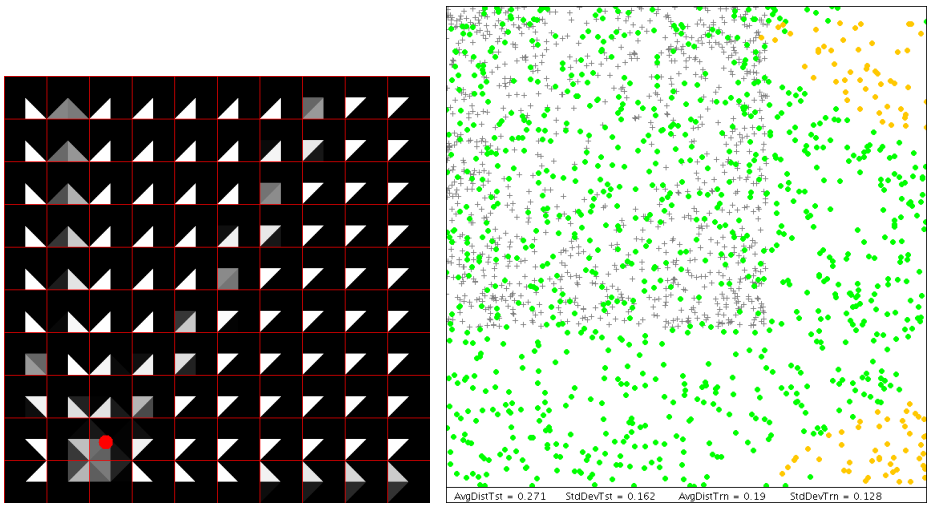
Normovanie akcií v našom prípade dovoľuje tiež v modeli zachytiť fyzikálne možnosti posunu ramena. Normovanie akcií na hodnotu 1 hovorí, že rameno je v jednom kroku schopné posunu o vektor dĺžky maximálne 1.

Schéma *many-to-one* Ako sme už uviedli, model CACLA sa nám podarilo uspokojivo natrénovať pre generovanie trajektórie podľa schémy *one-to-one*. Modely trénované podľa schémy *many-to-one* vykazovali takmer identické vlastnosti. Je potrebné si uvedomiť, že počas úvodných epoch trénovania modelu CACLA sa rameno pohybuje v podstate po všetkých častiach pracovného priestoru, nakoľko je model inicializovaný náhodne a teda aj generované akcie majú náhodné smery.

V neskorších fázach trénovania rameno taktiež prechádza väčšou časťou pracovného priestoru vďaka náhodnej explorácii. Model CACLA ďalej nijako nereprezentuje históriu predchádzajúcich stavov. Z toho dôvodu vôbec nerozlišuje medzi prípadom, keď začíname novú epochu v bode $[x,y]$, a prípadom, keď sa do bodu $[x,y]$ dostane počas inej epochy začínajúcej v inom bode.

Model trénovaný podľa schémy *one-to-one* zvládol dokonca takmer rovnako dobre generovať trajektórie do zvoleného koncového bodu, ako model trénovaný podľa schémy *many-to-one* pre ten istý koncový bod. Ako vidno na obrázku 4.6(a), natrénovaný model generuje trajektórie "spojite" nad celým pracovným priestorom. Ak teda počas trénovania niektoré časti priestoru model nemal možnosť prehliadnuť, dokáže sa z nich vydať správnym smerom k častiam priestoru v menšej vzdialenosti od cieľa, z ktorých už do cieľa spoľahlivo dorazí.

Niektoré modely boli schopné po natrénovaní na menšej časti pracovného priestoru generovať trajektórie do väčšej časti priestoru (obrázok 4.6(b)).

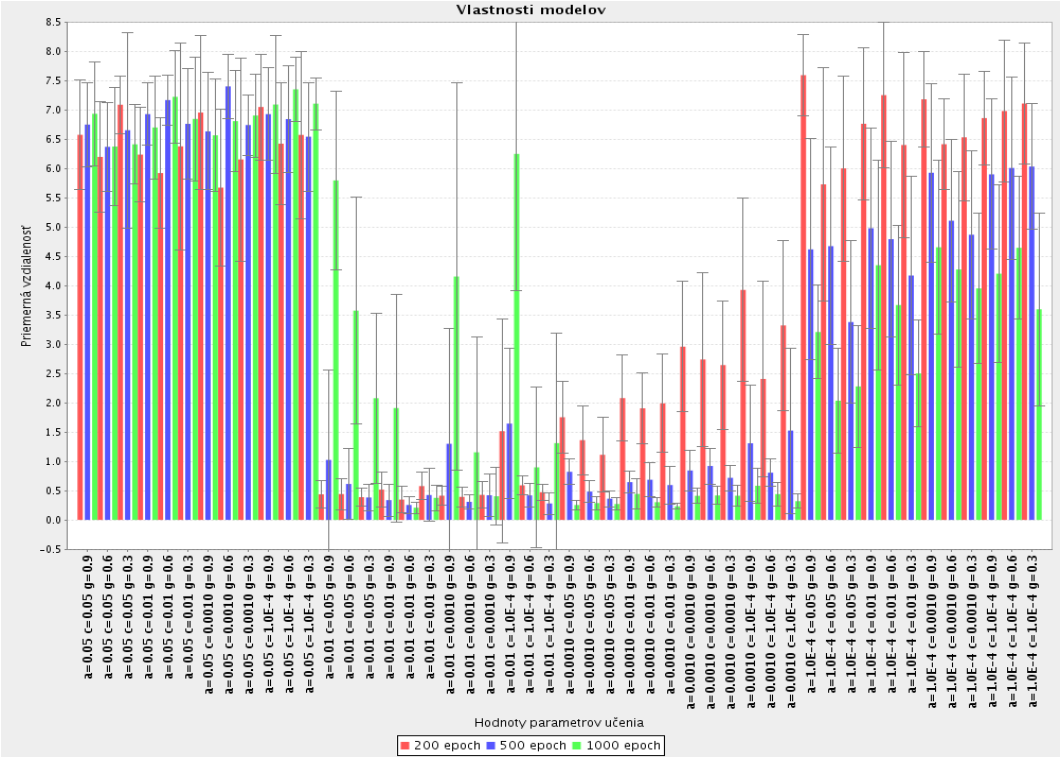


(a) Smery generovaných trajektórií (b) Trénovanie na menšej časti priestoru

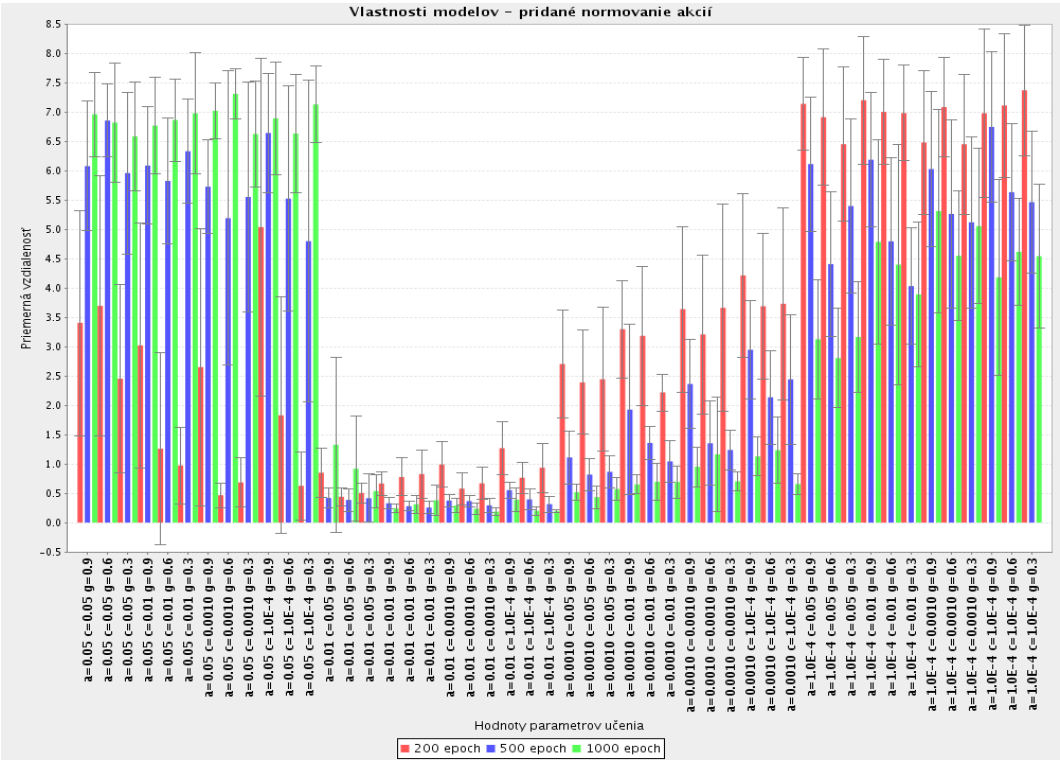
Obr. 4.6: Schopnosť zovšeobecnenia modelu

Schémy *one-to-many* a *many-to-many* Schémy *one-to-many* a *many-to-many* predstavujú zložitejší problém generovania trajektórie. Zatiaľ čo modely pre predchádzajúce dve schémy musia počas trénovania zistiť vhodnú stratégiu pre presun ramena vždy do rovnakého bodu, schémy *one-to-many* a *many-to-many* musia byť schopné nájsť vhodnú stratégiu pre ľubovoľný koncový bod. Schému *one-to-many* sme zvolili ako zjednodušenie schémy *many-to-many*, avšak ukázalo sa, že pre model CACLA nemalo ohraničenie začiatočných bodov trajektórie skoro žiaden prínos a problém generovania trajektórie bol pre model rovnako zložitý, ako problém *many-to-many*.

Podarilo sa nám úspešne nájsť parametre učenia pre problém *many-to-many* v dvojrozmernom pracovnom priestore. Priemerná vzdialenosť koncových bodov trajektórií generovaných modelmi pre tento problém je zobrazená na obrázku 4.7. Aktiváciu výstupných neurónov aktéra bolo opäť potrebné znížiť pomocou aktivačnej funkcie $f(net) = \frac{net}{10}$.



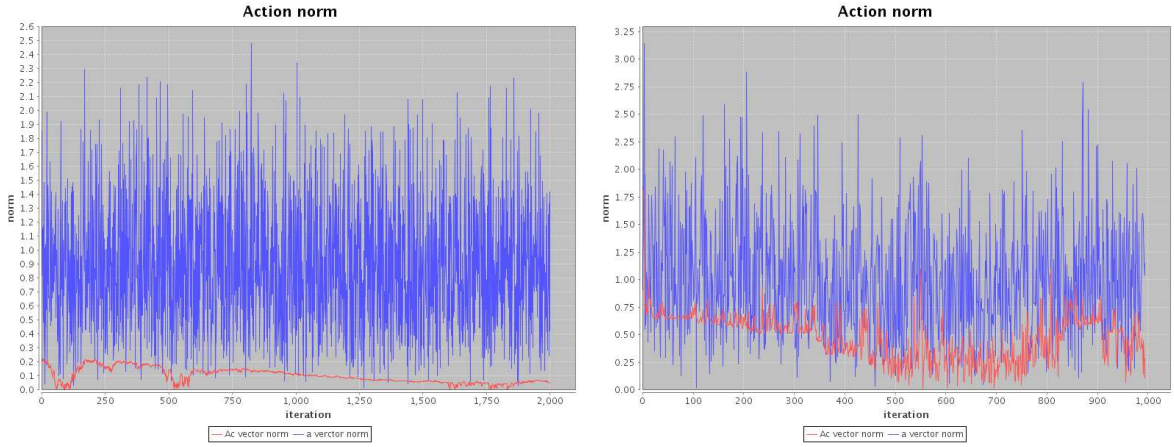
(a) Vzdialenosť modelov bez normovania akcií



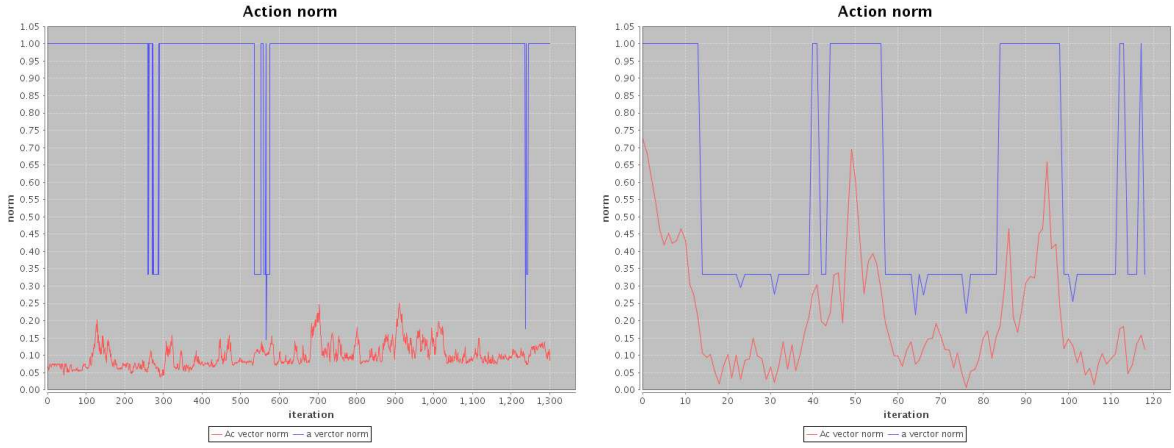
(b) Vzdialenosť modelov s normovanými akciami

Obr. 4.7: Vplyv normovania akcií na vlastnosti modelov pri schéme *many-to-many*

Zjemňovanie krokov trajektórie, hladkosť trajektórií Model natrénovaný s použitím normovaných akcií sa dokáže veľmi presne naučiť stratégiu zmenšovania akcií v závislosti od približovania sa k cieľu (obrázok 4.8(b)). Pri modeli trénovanom bez normovania akcií sa takéto správanie samo nevytvorí (obrázok 4.8(a))).



(a) Model bez normovania akcií

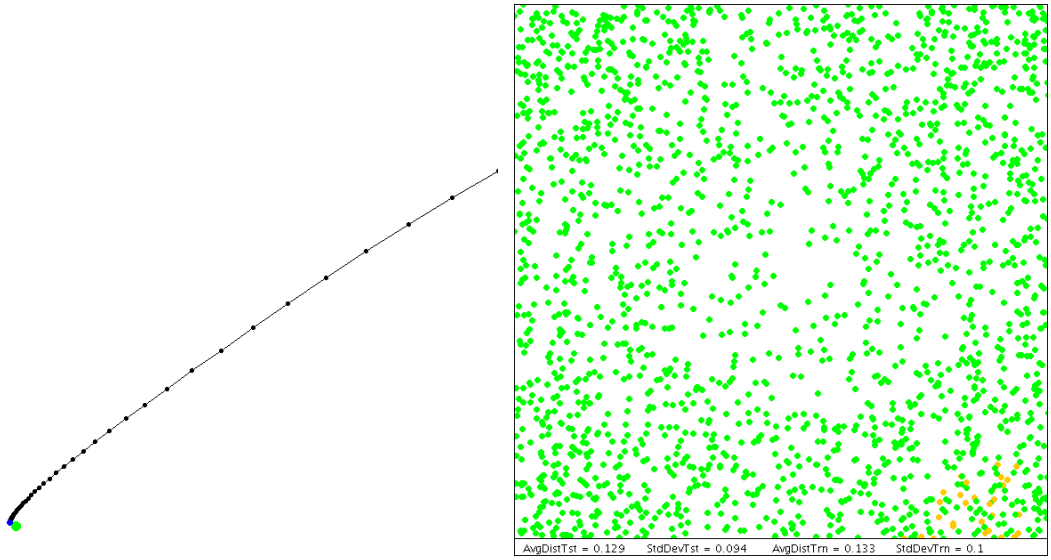


(b) Model s normovaním akcií

Obr. 4.8: Veľkosť akcií po prvej a poslednej trénovacej epoche

Ak teda znížime hodnoty akcií generovaných aktérom, ktorého váhy sú inicializované náhodnými malými hodnotami, potom v prvotných trénovacích epochách aj akcie sú veľmi malé. Následne sa počas trénovania majú možnosť zvyšovať po

nami volené veľkosti explorovaných akcií. Po ukončení trénovania je medzi veľkosťami predikovaných a explorovaných akcií vidieť takmer totožný trend. Ak je rameno vo väčšej vzdialenosti od cieľa, potom akcia po explorácii nadobúda normu 1 a samotný aktér tiež predikuje akciu s väčšou normou. Ak je rameno bližšie, potom aj norma akcie v grafe je malá.



(a) Trajektória vygenerovaná modelom (b) Dosiahnuteľnosť celého priestoru modelom

Obr. 4.9: Generované trajektórie

Postupné zjemňovanie trajektórie je tiež viditeľné na reálne generovaných trajektóriách po natrénovaní (obrázok 4.9(a)). Trajektórie sú dostatočne hladké a spravidla takmer lineárne, rameno sa teda naučí pohybovať vždy v smere rastu funkcie odmeny a trestu. Model CACLA je teda schopný uspokojivo generovať hladké trajektórie z ľubovoľného bodu do ľubovoľného bodu pracovného priestoru.

Obchádzanie prekážok V niekoľkých experimentoch sme sa snažili natrénovať model CACLA v pracovnom priestore obsahujúcom prekážky. Cieľom bolo zistiť, či je model schopný tieto prekážky obchádzať v následne generovaných trajektóriách.

Nepodarilo sa nám natrénovať konvergentné modely pre prostredie s prekáž-

kami. Natrénovaný model pri generovaní trajektórie po "náraze" do prekážky zostal stáť v mieste nárazu a nebol schopný prekážku obísť. Model sa vždy snažil ísť iba lineárne v smere najväčšej odmeny. Domnievame sa preto, že obchádzanie prekážok by bolo možné modelu uľahčiť voľbou vhodnej funkcie odmeny a trestu.

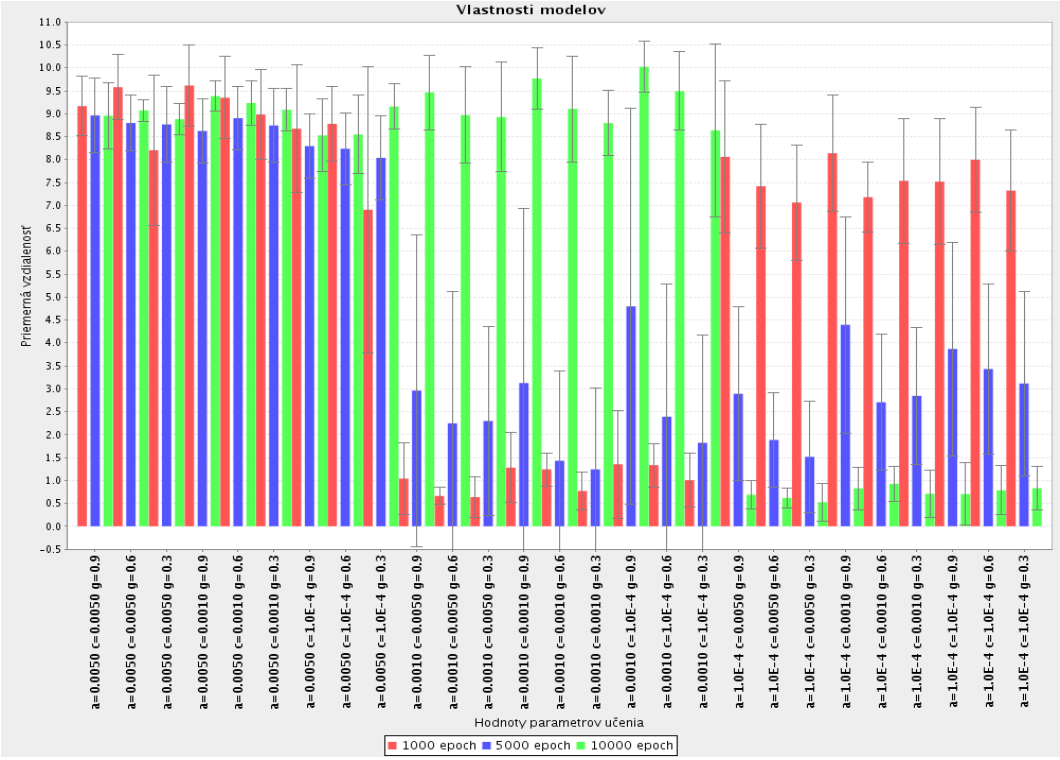
4.6 Generovanie trajektórie v priestore 3D

Pri rozšírení problému do trojrozmerného priestoru sme zachovali architektúru aproximátorov funkcií z obrázka 4.1. Rozdiel spočíva v rozšírení stavov zo štvorrozmerného vektora na vektor šesťrozmerný. Počet neurónov na skrytej vrstve sme ponechali rovný 20. Na výstupnej vrstve aktéra sú tri neuróny, ktoré majú aktivačnú funkciu zníženú, ako bolo opísané vyššie.

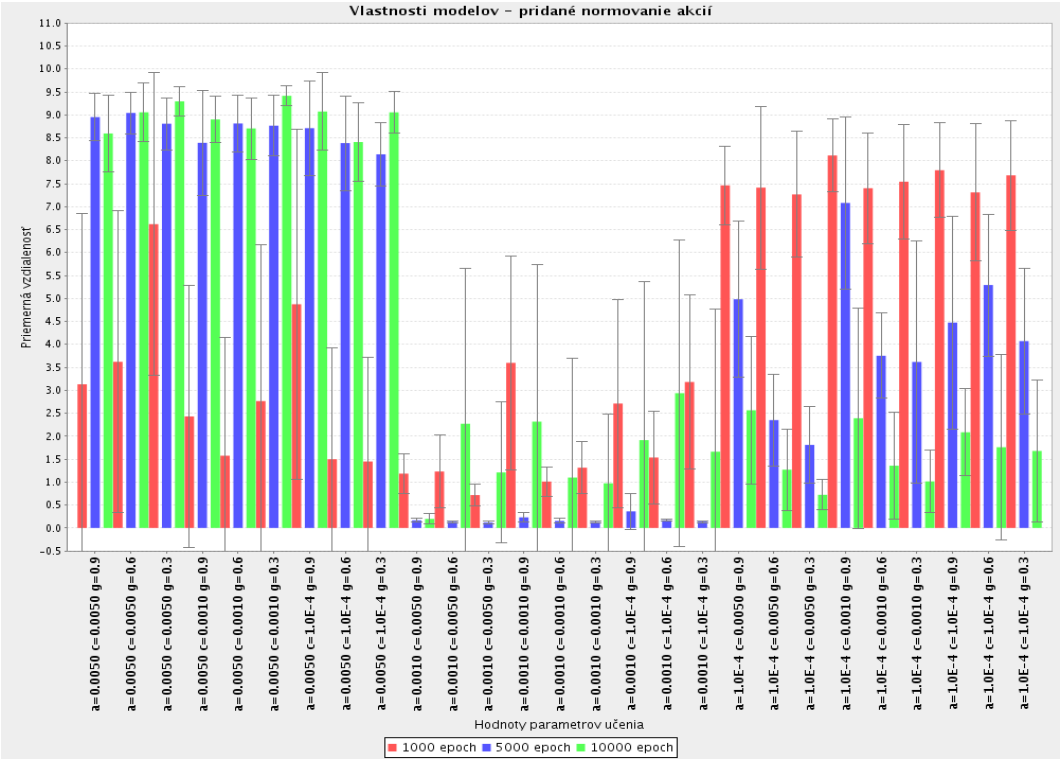
V trojrozmernom priestore sme sa zaoberali už iba schémou trénovania *many-to-many*, teda generovanie trajektórií medzi dvoma ľubovoľnými bodmi. Modely sme ohodnocovali opäť podľa algoritmu 3, ohodnotenie modelov podľa parametrov učenia je zobrazené na obrázku 4.10.

V trojrozmernom priestore sa normovanie akcií ukázalo v podstate nevyhnutné, modely bez normovania neboli schopné dosiahnuť priemernú vzdialenosť nižšiu ako 0.5. Prechodom do trojrozmerného priestoru sa podstatne zväčší stavový aj akčný priestor, preto bolo nutné pri trénovaní zvýšiť počet epoch. Najlepšie výsledky dosahovali modely pri trénovaní počas 5000 epoch (rôznych trajektórií), ktoré mali opäť maximálnu dĺžku 2000 bodov.

Model CACLA je teda schopný generovať trajektórie medzi dvoma ľubovoľnými bodmi aj v trojrozmernom priestore. Keďže prechodom k trojrozmernému priestoru sa podstatne zväčší stavový aj akčný priestor, je nutné modelu počas trénovania prezentovať viac trénovacích trajektórií. Kým v prípade dvojrozmerného priestoru sme najlepšie výsledky dosiahli s modelmi trénovanými počas 500 - 1000 epoch s rýchlosťou učenia aktéra 0.01, v priestore trojrozmernom to bolo 5000 epoch pri nižšej rýchlosti učenia aktéra 0.001. Pri najlepších modeloch je opäť viditeľné, že modely s nižším diskontným faktorom dosahujú lepšie výsledky.



(a) Model trénovaný bez normovania akcií



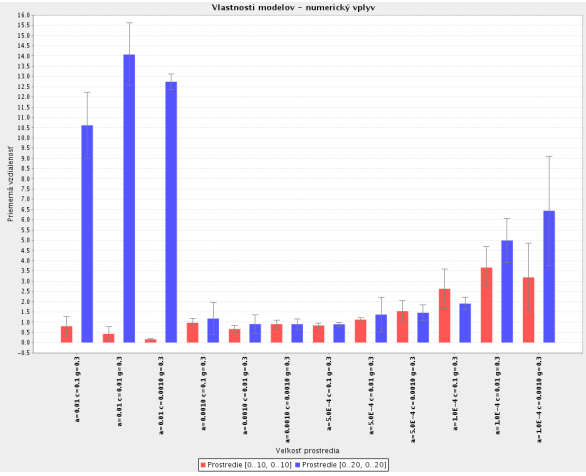
(b) Model trénovaný s normovanými akciami

Obr. 4.10: Priemerná vzdialenosť modelov od cieľa pri schéme *many-to-many* v 3D

4.7 Prepojenie na vizuálnu percepciu pomocou kamier

Vo finálnej fáze sme model CACLA prepojili s vizuálnou percepciou pomocou dvoch kamier scény a snažili sme sa ho natrénovať pre generovanie uhlov natočenia ramena. Ako vstup sme modelu prezentovali osemrozmerný vektor, ktorý pozostával z vektorov $E = \left(e_1^x, e_1^y, e_2^x, e_2^y\right)$ a $T = \left(t_1^x, t_1^y, t_2^x, t_2^y\right)$, ktoré zodpovedali zobrazeniu koncového bodu efektora a cieľového bodu pomocou dvoch kamier (obrázok 1.1). Výstupom modelu CACLA bol vektor $\Delta\theta = (\Delta\theta_1, \Delta\theta_2, \Delta\theta_3)$, ktorý zodpovedal zmenám jednotlivých uhlov natočenia ramena.

Podarilo sa nám nájsť parametre učenia tak, aby modely konvergovali, avšak priemerná vzdialenosť koncových bodov generovaných trajektórií od reálnych cieľov bola horšia (väčšinou v rozmedzí hodnôt 1.5 až 2.5. Predpokladáme však, že je možné nájsť parametre učenia, ktoré zabezpečia natrénovanie modelov s lepšími vlastnosťami. Model CACLA sa ukázal byť enormne numericky citlivý. Napríklad pri probléme *many-to-many* v 2D iba zmena rozmeru pracovného priestoru z $[0..10, 0..10]$ na rozmer $[0..20, 0..20]$ zapríčinila, že modely pre rýchlosť učenia aktéra 0.01 neboli schopné skonvergovať (obrázok 4.11).



Obr. 4.11: Numerická citlivosť modelu CACLA

Predpokladáme preto, že príčina horších parametrov modelov s vizuálnou per-

cepciou je spôsobená iba takýmto numerickým problémom. Problém navrhujeme riešiť voľbou iných parametrov učenia, prípadne vhodným preškálovaním stavov a akcií.

Kapitola 5

Záver

Ciele diplomovej práce sa nám podarilo splniť. Prvotná hypotéza, ktorá viedla k vypracovaniu tejto diplomovej práce, sa nepotvrdila. Sieť RecSOM v našich experimentoch nebola schopná dostatočne presne diskriminovať rôzne tvary trajektórií, no opierala sa takmer výlučne o aktuálnu pozíciu v priestore. Z toho dôvodu sieť RecSOM nie je vhodným aparátom, pomocou ktorého by sa dal problém dosahovania cieľových pozícií riešiť.

Pre riešenie problému generovania trajektórie sme sa ďalej rozhodli použiť prístupy z kategórie učenia s posilňovaním. Navrhli sme riešenie pomocou modelu CACLA – *Continuous Actor Critic learning automaton*, ktorý pracuje nad spojitým priestorom stavov a akcií. Tento model sa nám podarilo úspešne použiť na ovládanie robotického ramena v trojrozmernom priestore. Model zvládol pomerne presne generovať hladké trajektórie medzi ľubovoľnými dvoma bodmi. Model CACLA bol v našich experimentoch pomerne citlivý na voľbu parametrov učenia. Systematicky sme ohraničili vhodné parametre a vytvorili sme štatistické ohodnotenie modelov podľa parametrov. Navrhli sme spôsob modifikácie explorovaných akcií počas trénovania a ukázali sme pozitívny vplyv na kvalitu modelov. Pozorovali sme trend zlepšovania modelov pri nízkych hodnotách diskontného faktora γ.

Model CACLA je v dobe písania diplomovej práce relatívne novým modelom. Jednoduchšie experimenty z pôvodného článku, z ktorého sme vychádzali, sme

rozšírili o experimenty s podstatne väčším stavovým a akčným priestorom. Ukázali sme, že jediný model CACLA je schopný generovať trajektórie medzi ľubovoľnými dvoma bodmi v trojrozmernom priestore. Taktiež sa nám podarilo prepojiť model s vizuálnou percepciou pomocou dvoch kamier tak, aby generoval zmeny uhlov natočenia robotického ramena. Vlastnosti modelu boli v tomto prípade mierne horšie.

Z implementačného hľadiska bola práca pomerne rozsiahla. V zvolenom jazyku Java sme implementovali viaceré knižnice – knižnicu pre matematické výpočty, knižnicu pre paralelné výpočty, knižnicu pre prácu so sieťou RecSOM, knižnicu pre model CACLA s použitím doprednej neurónovej siete. Implementovali sme vizualizáciu robotického ramena a pracovného priestoru v trojrozmernom priestore s využitím knižnice OpenGL. Implementácia nám priniesla množstvo nových vedomostí, využili sme pokročilejšie programátorské techniky ako volanie natívnych metód, dynamické zavádzanie tried a programovanie s využitím návrhových vzorov.

Literatúra

Eckel, B.: 2005, *Thinking in Java (4th Edition)*, Prentice Hall PTR, Upper Saddle River, NJ, USA.

Hammer, B., Micheli, A., Sperduti, A. a Strickert, M.: 2004, Recursive self-organizing network models, *Neural Networks* **17**(8-9), 1061–1085.

Haykin, S.: 1998, *Neural Networks: A Comprehensive Foundation*, Prentice Hall PTR, Upper Saddle River, NJ, USA.

Kawato, M.: 1999, Internal models for motor control and trajectory planning, *Current Opinion in Neurobiology* **9**(6), 718–727.

Lazaric, A., Restelli, M. a Bonarini, A.: 2007, Reinforcement learning in continuous action spaces through sequential monte carlo methods, *Advances in Neural Information Processing Systems*.

Macura, Z., Cangelosi, A., Ellis, R. a Bugmann, D.: n.d., A cognitive robotic model of grasping.

Mehta, B. a Schaal, S.: 2002, Forward models in visuomotor control, *The Journal of Neurophysiology* **88**(2), 942–953.

Oztop, E., Bradley, N. S. a Arbib, M. A.: 2004, Infant grasp learning: a computational model, *Experimental Brain Research* **158**(4), 480–503.

Pazis, J. a Lagoudakis, M. G.: 2009, Binary action search for learning continuous-action control policies, *ICML '09: Proceedings of the 26th Annual International Conference on Machine Learning*, ACM, New York, NY, USA, pp. 793–800.

Pecinovský, R.: 2007, *Návrhové vzory*, Computer Press.

Ritter, H., Martinetz, T. a Schulten, K.: 1992, *Neural Computation and Self-Organizing Maps; An Introduction*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Sakai, K., Kitaguchi, K. a Hikosaka, O.: 2003, Chunking during human visuomotor sequence learning, *Experimental Brain Research* **152**(2), 229–242.

Santamaría, J. C., Sutton, R. S. a Ram, A.: 1998, Experiments with reinforcement learning in problems with continuous state and action spaces, *Adaptive Behavior* **6**, 163–218.

Schaal, S.: 2002, Arm and hand movement control, *The handbook of brain theory and neural networks, Second Edition*, MIT Press, pp. 110–113.

Sutton, R. S. a Barto, A. G.: 1998, *Reinforcement Learning: An Introduction*, The MIT Press.

Tamosiunaite, M., Asfour, T. a Wörgötter, F.: 2009, Learning to reach by reinforcement learning using a receptive field based function approximation approach with continuous actions, *Biological Cybernetics* **100**(3), 249–260.

Tiňo, P., Farkaš, I. a van Mourik, J.: 2006, Dynamics and topographic organization of recursive self-organizing maps, *Neural Computation* **18**(10), 2529–2567.

van Hasselt, H. a Wiering, M. A.: 2007, Reinforcement learning in continuous action spaces, *Approximate Dynamic Programming and Reinforcement Learning, 2007. ADPRL 2007. IEEE International Symposium on*, pp. 272–279.

Vančo, P. a Farkaš, I.: 2010, Experimental comparison of recursive self-organizing maps for processing tree-structured data, *Neurocomputing* **73**(7-9), 1362–1375.

von Hofsten, C.: 2004, An action perspective on motor development, *Trends in Cognitive Sciences* **8**(6), 266–272.

Dodatok A

Použité knižnice

Pri implementácii sme použili niekoľko hotových knižníc, ktoré sú uvedené nižšie spolu s licenciami, pod ktorými sú šírené. Použili sme tiež všetky potrebné knižnice, ktoré sú uvedenými knižnicami vyžadované a dodávajú sa priamo s nimi.

- *JOGL*
  Implementácia grafickej knižnice OpenGL pre jazyk Java. Využili sme zásuvný modul pre programovacie prostredie NetBeans.
  `http://kenai.com/projects/netbeans-opengl-pack/pages/Home`
  Licencia: `GPL v2.0`

- *Log4J* 1.2.15
  Implementácia pokročilého zaznamenávania udalostí.
  `http://logging.apache.org/log4j/1.2/`
  Licencia: `The Apache Software License, Version 2.0`

- *Dom4J* 1.6.1
  Podpora práce s XML súbormi.
  `http://dom4j.sourceforge.net/`
  Licencia: `BSD License`

- *Hibernate* 3.2.5.*ga*
  ORM mapovací nástroj pre perzistenciu objektov do databázy.

`http://www.hibernate.org/`
Licencia: `LGPL v2.1`

- *JFreeChart* 1.0.13
  Implementácia generovania grafov.
  `http://www.jfree.org/jfreechart/`
  Licencia: `LGPL`

Dodatok B

Prílohy

Prílohy diplomovej práce sa nachádzajú na priloženom médiu. Médium obsahuje

- zdrojové kódy všetkých vytvorených knižníc
- dokumentáciu všetkých knižníc vytvorenú štandardným systémom *javadoc*
- niekoľko natrénovaných modelov