

VILNIAUS UNIVERSITETAS
INFORMATIKOS INSTITUTAS
PROGRAMŲ SISTEMOS

**Skatinamojo mokymosi algoritmų skirtų
Sokoban žaidimo agento valdymui palyginimas**
**Comparison of Reinforcement Learning Algorithms for
Sokoban Game Agent Training**

Bakalauro baigiamasis darbas

Atliko:	Jokūbas Rusakevičius	(parašas)
Darbo vadovas:	vyresn. m.d. Virginijus Marcinkevičius	(parašas)
Darbo recenzentas:	j. asist. Linas Petkevičius	(parašas)

Vilnius – 2020

Dėkoju šeimai ir draugams, už meilę ir nuolatinį palaikymą bei darbo vadovui, vyresn. m.d.

Virginijui Marcinkevičiui, už visapusišką pagalbą geriausio sprendimo beieškant.

Santrauka

Šiame darbe buvo tiriamas Skatinamojo mokymosi agento pritaikymas Sokoban žaidimui. Darbe, aprašoma skatinamojo mokymosi teorija, pagrindiniai principai ir taikymo technikos. Taip pat, aprašomi skirtingi neuroniniai tinklai, dėl jų naudojimo skatinamojo mokymosi strategijų architektūroje. Darbe apžvelgiamos Sokoban žaidimo aplinkos ir modernios skatinamojo mokymosi bibliotekos, karkasai. Darbe lyginamos skirtingų skatinamojo mokymosi algoritmų implementacijos ir skirtingos strategijos Sokoban žaidimo aplinkoje. Darbe pritaikytas mokymosi žinių perdavimo principas apmokant skatinamojo mokymosi agentus žaisti sudėtingesnes Sokoban žaidimo aplinkas. Gauti žinių perdavimo taikymo rezultatai yra palyginami su agentu apmokytu vienodo sudėtingumo aplinkoje be žinių perdavimo. Išanalizuoti gauti rezultatai parodė, kad naudojant mokymosi žinių perdavimą Sokoban žaidimo aplinkai, skatinamojo mokymosi algoritmai gali pasiekti geresnių rezultatų.

Raktiniai žodžiai: Skatinamasis mokymas, Sokoban žaidimas, aktoriumi-kritiku paremti metodai, žaidimo agento mokymas, mokymosi žinių perdavimas

Summary

In this thesis, the application of the reinforcement learning agent to the Sokoban game was investigated. The paper describes the theory, basic principles and application techniques of reinforcement learning. Also, different neural networks are described, due to their use in the architecture of reinforcement learning strategies. Thesis reviews Sokoban game environments and state-of-art reinforcement learning libraries, frameworks. The paper compares implementations of different reinforcement learning algorithms and different strategies in the Sokoban game environment. The obtained results of transfer learning are compared to those of an agent trained in an environment of equal complexity without transfer learning. Analysis of the results showed that using transfer learning for the Sokoban game environment, reinforcement learning algorithms can achieve better results.

Keywords: reinforcement learning, Sokoban game, actor-critic based methods, game agent training, transfer learning

TURINYS

IVADAS	7
Problematika	7
Darbo tikslas.....	8
Darbo uždaviniai	8
1. SKATINAMASIS MOKYMAS IR DIRBTINIAI NEURONINIAI TINKLAI	9
1.1. Skatinamasis mokymasis	9
1.1.1. Skatinamojo mokymosi bendroji teorija	10
1.1.2. Markovo sprendimo priėmimo procesai	12
1.1.3. Sustiprinto mokymosi strategijos paieška	14
1.1.3.1. Strategijos gradientai	15
1.1.4. Aktoriaus-kritiko principas	15
1.2. Dirbtiniai neuroniniai tinklai	16
1.2.1. Dirbtinių neuroninių tinklų architektūros komponentai	17
1.2.1.1. Dirbtinis neuronas	17
1.2.1.2. Linijinis slenksčio vienetą	17
1.2.1.3. Pagalbiniai neuronai	18
1.2.1.4. Dirbtinių neuroninių tinklų sluoksniai	18
1.2.2. Parceptronas	18
1.2.3. Daugiasluoksniai perceptronai	19
1.2.4. Konvoliuciniai neuroniniai tinklai	21
1.2.4.1. Konvoliuciniai sluoksniai	22
1.2.4.2. Sutelkimo sluoksniai.....	23
1.2.5. Rekurentiniai neuroniniai tinklai	24
1.2.5.1. Pasikartojantys neuronai.....	24
1.2.5.2. Ilgos trumpalaikės atminties modelis	25
2. METODOLOGIJA	27
2.1. Sokoban žaidimas	27
2.1.1. Aplinka	27
2.1.1.1. Aplinkos pasirinkimas	27
2.1.1.2. OpenAI Gym aplinkos	28
2.1.1.3. Gym-Sokoban.....	29
2.2. Skatinamojo mokymosi biblioteka	30
2.3. Skatinamo mokymosi bibliotekos pasirinkimas.....	30
2.3.1. Stable Baselines skatinamojo mokymosi biblioteka	32
2.3.1.1. Stable Baselines Karkasas	33
2.3.2. Stable Baselines strategijos	34
2.3.2.1. CnnPolicy strategijos aprašymas.....	34
2.3.2.2. CnnLstmPolicy strategijos aprašymas	35
2.3.2.3. CnnLnLstmPolicy strategijos aprašymas	35
2.3.3. Stable Baselines algoritmai.....	35
2.3.3.1. A2C algoritmo aprašymas	36
2.3.3.2. ACER algoritmo aprašymas	36
2.3.3.3. PPO2 algoritmo aprašymas	36

3. EKSPERIMENTAI	38
3.1. Eksperimentinės aplinkos paruošimas	38
3.1.1. Eksperimentinės aplinkos specifikacijos	38
3.1.2. Skatinamojo mokymo modelio paruošimas	39
3.1.2.1. Sokoban aplinkos paruošimas	39
3.1.2.2. Mokymo aplinkos paruošimas	39
3.2. Eksperimento planas	40
3.3. Eksperimento eiga	41
3.4. Eksperimento rezultatų analizė	42
3.4.1. Pirmos eksperimento dalies: geriausios strategijos ieškojimo rezultatai	42
3.4.2. Antros eksperimento dalies: geriausio algoritmo ieškojimo rezultatai	43
3.4.3. Trečios eksperimento dalies: mokymo žinių perdavimo naudos tyrimo rezultatai	45
3.5. Eksperimento apibendrinimas ir išvados	47
REZULTATAI	50
IŠVADOS	51
LITERATŪRA	52
SANTRUMPOS	56
PRIEDAI	56
1 priedas. Stable Baselines skatinamojo mokymosi bibliotekos algoritmų A2C, ACER, PPO2 implementacijų numatytosios hiper-parametrų reikšmės	57
2 priedas. Sokoban aplinkai pagal OpenAI Gym principus parašytas programinis kodas	58
3 priedas. Modelio mokymo programinis kodas A2C, ACER ir PPO2 algoritmams bei CnnPolicy, CnnLstmPolicy ir CnnLnLstmPolicy strategijoms	60
4 priedas. Modelio mokymo su žinių perdavimu programinis kodas	62

Įvadas

Šiame skyriuje aprašomi bakalauro darbo problematika, tikslas ir uždaviniai.

Problematika

Kompiuterių pajėgumui ir atliekamų operacijų per sekundę skaičiui nuolatos didėjant – didėja ir lūkesčiai bei sprendžiamų uždavinių sudėtingumas. Dar reliatyviai neseniai sudėtingiausios programos ir kompiuterių sprendžiami uždaviniai susidėjo iš skaičiuotuvo operacijų ar žinučių perdavimo. Tačiau technologijoms tobulėjant, kiekvienam žmogui kišenėje besinešiojant pirmųjų kompiuterių kaip „ENIAC“ [oCom] dydį pajuokiančius kompiuterinius įrenginius, natūraliai didėja ir jiems keliami iššūkiai.

Šiais laikais kompiuteriai gali simuliuoti atominius sprogius, nuspėti orus ir atlikti kitas didžiulių skaičiavimo išteklių reikalaujančias užduotis [Ker]. Tačiau užduoties sudėtingumą gali lemti ne tik milžiniškų išteklių skaičiaus reikalavimas. 2016 metais matėme, kaip „Google’s AlphaGo“ nugalėjo pasaulio aukščiausio lygio „Go“ žaidėją ir čempioną Ke Jie [Moz17]. Autonominiai gatvėmis važinėjantys automobiliai neišvengiamai artėja, o „Boston Dynamics“ robotai stebina savo galimybėmis [Dyn20].

Šie uždaviniai nėra trivialiai aprašomi ar išsprendžiami, jiems gali net neegzistuoti sprendimas. Tokiems uždaviniams spręsti yra naudojami mašininio mokymosi metodai (pvz. neuroniniai tinklai). Viena šių metodų paradigmų yra skatinamasis mokymas – agento atliekami veiksmai yra reguliariai vertinami ir atitinkamai agentas yra apdovanojamas arba baudžiamas.

Žaidimai dėl lengvai išskiriamų gerų ir blogų veiksmų, natūraliai, tampa pagrindine skatinamojo mokymosi algoritmų treniravimosi ir žaidimų aikšte. Klasikiniai Atari žaidimai yra dažnas tyrimų objektas [MKS⁺13a]. Tačiau žaidimų sudėtingumas varijuoja ir ne visus žaidimus kompiuteriai gali išmokyti ar mokosi vienodai. Egzistuoja žaidimų ir uždavinių kategorijų, kurios skatinamojo mokymosi agentams yra ypatingai sudėtingos. Galimybė padaryti nepataisomą klaidą arba poreikis mąstyti į priekį yra tai, ko trūksta dažnam skatinamojo mokymosi agentui.

Sokoban žaidimas būtent ir priklauso: sudėtingų išmokyti kompiuteriui žaidimų kategorijai. Tai yra senas ir savo laiku dažnas mokslinių tyrimų objektas [Cul97; DZ99; JS98; Sch05]. Tačiau nepaisant galvosūkių amžiaus, tai vis dar yra aktuali skatinamojo mokymosi taikymo aplinka. Sokoban žaidimas priklauso planavimo uždavinių kategorijai, kurie, dėl savo dažnai reikalingo gebėjimo mąstyti į priekį, yra sunkiai išsprendžiami skatinamojo mokymosi būdu [Sch18].

Darbo tikslas

Šio darbo **tikslas** – palyginti skatinamojo mokymosi algoritmus, siekiant nustatyti efektyviausią algoritmą ir jo strategiją Sokoban žaidimui.

Darbo uždaviniai

Darbui iškelti **uždaviniai**:

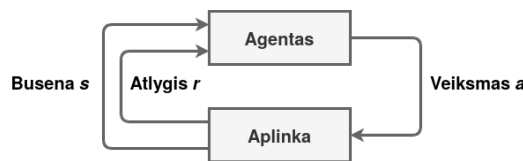
1. Atlikti skatinamojo mokymosi algoritmų analizę ir atrinkti keletą potencialiausių algoritmų Sokoban žaidimo agento mokymui.
2. Paruošti eksperimentinę aplinką Sokoban žaidimui.
3. Atlikti eksperimentą, siekiant nustatyti, kuri Sokoban žaidimo agento valdymo strategija yra geriausia.
4. Eksperimentiškai palyginti skatinamojo mokymosi algoritmus: A2C, ACER, PPO naudojant mažiausią įmanomą Sokoban žaidimo aplinką.
5. Panaudoti žinių perdavimą, siekiant Sokoban žaidimo agentą apmokyti veikti sudėtingesnėse aplinkose.

1. Skatinamasis mokymas ir dirbtiniai neuroniniai tinklai

Šiame skyriuje aprašyta teorinė bakalauro darbo dalis.

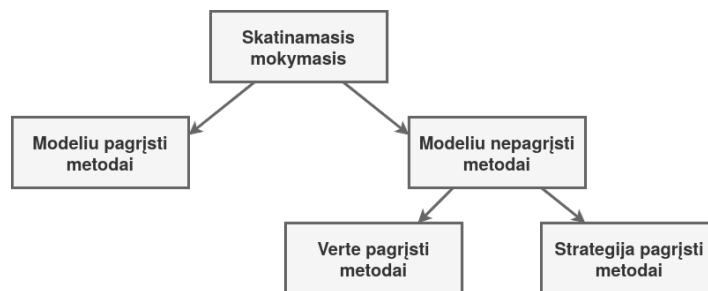
1.1. Skatinamasis mokymasis

Skatinamasis mokymasis (*angl. reinforcement learning*) (RL), kartu su prižiūrimuoju ir neprižiūrimuoju mokymu, yra viena iš pagrindinių mašininio mokymosi (ML) paradigimų. Klasikinis RL modelis susideda iš agento (*angl. agent*), kuris priima sprendimus ir atlieka veiksmus (*angl. actions*) pagal strategiją (*angl. policy*), paremtus aplinkos būseną (*angl. state*), ir siekiantis pasiekti didžiausią įmanomą atlygį (*angl. reward*) (paveikslėlis 1).



1 pav. Skatinamojo mokymosi agento sąveika su aplinka

Skirtingai nei kitos ML paradigmos, RL sprendžia ne regresijos, klasifikacijos, ar grupavimo, bet atlygiu paremtas (*angl. reward-based*) problemas, ir tai daro bandymų ir klaidų (*angl. trial and error*) principu. Dėmesys yra nukreiptas ne į sužymėtas (*angl. labeled*) įvesties ir išvesties duomenų poras, bet į balanso tarp tyrinėjimo (*angl. exploration*) ir išnaudojimo (*angl. exploitation*) ieškojimą [KLM96].



2 pav. Skatinamojo mokymosi algoritmų kategorijos

RL algoritmai yra skirstomi pagal skirtingus rodiklius į kelias skirtingas kategorijas [Ope] (paveikslėlis 2). Prieš aiškinantis kuo skiriasi kiekviena kategorija, svarbu suprasti, kad RL algoritmai susideda iš dviejų fazių:

1. **Mokymosi fazė** – (*angl. learning phase*) tai yra algoritmo apmokymo dalis, kai kiekvienas agento priimtas sprendimas ir atliktas veiksmas aplinkoje bei gautas atlygis ir nauja aplinkos būsena yra panaudojama tolimesniam modelio optimizavimui ir gerinimui.

2. **Taikymo fazė** – (*angl. interface phase*) tai yra algoritmo dalis, kai, nepriklausomai nuo atlikto veiksmo optimalumo, modelis nebėra keičiamas ir yra naudojamos iki šiol išmoktos reikšmės.

Kaip jau minėta anksčiau, RL algoritmai yra skirstomi į skirtingas kategorijas (paveikslėlis 2).

Pagal modelio struktūrą [Ope]:

1. **Pagrįsti modeliu** – (*angl. model-based*) tai yra algoritmai, kurie optimalios strategijos apskaičiavimui naudojami transakcijų funkcija (ir atlygio funkcija) (daugiau punkte 1.1.2). Pagrįsti modeliu algoritmai gali nuspėti galimus aplinkos pakitimus, kadangi naudojami apskaičiuota transakcijų funkcija. Tačiau, modelio turima funkcija gali būti tik apytikslė „tik-rajai“ funkcijai, tad modelis gali niekada nepasiekti optimalaus sprendimo.
2. **Nepagrįsti modeliu** – (*angl. model-free*) tai yra algoritmai, kurie apskaičiuodami optimalią strategiją nesinaudoja ir nebando apskaičiuoti aplinkos dinamikų (transakcijų ir perėjimų funkcijos nenaudojamos). Nepagrįsti modeliu algoritmai bando apskaičiuoti vertės arba strategijos funkciją tiesiai iš patirties (interakcijų su aplinka).

Pagal strategijos pritaikymą:

1. **Besiremiantys optimalia strategija** – (*angl. on-policy*) algoritmai, kurie, mokymosi metu rinkdamiesi veiksmą, remiasi strategija išvesta iš tuo metu apskaičiuotos optimaliausios strategijos bei atlieka atnaujinimus remdamiesi ta pačia strategija.
2. **Nesiremiantys optimalia strategija** – (*angl. off-policy*) algoritmai, kurie mokymosi metu remiasi skirtinga strategija nei tuo metu apskaičiuota optimaliausia strategija. Atnaujinimai atliekami remiantis geresnį rezultatą gražinančia strategija.

Nepagrįsti modeliu algoritmai yra skirstomi į dar dvi kategorijas, pagal tai, kuo jie yra pagrįsti:

1. **Pagrįsti verte** – (*angl. value-based*) tai yra algoritmai pagrįsti *laiko skirtumų mokymusi* (*angl. temporal difference learning*), kur yra mokomasi funkcija V^π arba V^* (daugiau punkte 1.1.1).
2. **Pagrįsti strategija** – (*angl. policy-based*) tai algoritmai, kurie tiesiogiai mokosi optimalios strategijos π^* arba bando apytiksliai surasti optimalią strategiją.

Toliau šiame poskyryje bus giliau nagrinėjamas RL ir jį sudarantys elementai.

1.1.1. Skatinamojo mokymosi bendroji teorija

Šiame darbe bus remiamasi standartine RL aplinka, kur agentas yra aplinkoje \mathcal{E} diskretų kiekį laiko vienetų arba žingsnių (*angl. time steps*). Kiekvieną žingsnį t agentas gauna informaciją apie aplinkos būseną s_t ir iš visų įmanomų veiksmų rinkinio \mathcal{A} pasirenka atitinkamą veiksmą a_t

pagal strategiją π , kur strategija π pasako kokią veiksmą a_t rinktis situacijoje s_t . Aplinka agentui grąžina informaciją apie sekančią aplinkos būseną s_{t+1} ir skaliarinę atlygio reikšmę r_t . Toks procesas yra tęsiamas, kol aplinka pasiekia galinę būseną (*angl. terminal state*). Kai galinė būsena yra pasiekama – procesas yra pradedamas iš naujo. Rezultatas $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ yra bendras žingsnyje t surinktas atlygis su nuolaidos koeficientu (*angl. discount factor*) $\gamma \in (0, 1]$. Agento tikslas yra gauti didžiausia įmanoma tikėtina rezultatą su visomis aplinkos būsenomis s_t .

Veiksmo vertė (*angl. action value*) apskaičiuojama: $Q^\pi(s, a) = \mathbb{E}[R_t | s_t = s, a]$, kur tikėtinas rezultatas \mathbb{E} gaunamas pagal strategiją π pasirinkus veiksmą a , esant situacijoje s . Optimali vertės funkcija $Q^*(s, a) = \max_{\pi} Q^\pi(s, a)$, grąžina didžiausią strategijos π pasiektą veiksmo vertę aplinkos būsenai s ir veiksmui a . Panašiai, aplinkos būsenos s vertė (*angl. state value*) remiantis strategija π yra apibrėžiama taip: $V^\pi(s) = \mathbb{E}[R_t | s_t = s]$, ir tai yra tikėtinas rezultatas būsenai s pagal strategiją π .

Verte ir modeliu pagrįstuose (poskyris 1.1) RL methoduose, veiksmo vertės funkcija yra reprezentuojama funkcijos aproksimacija (*angl. approximator*), pavyzdžiui, neuroniniu tinklu (daugiau poskyryje 1.2). Jei imame $Q(s, a; \theta)$ kaip apytikslę veiksmo vertės funkciją su parametrais θ , tada atnaujinimai skirti θ gali būti išvesti iš įvairių RL algoritmų. Pavyzdžiui, vienas tokių algoritmų gali būti Q-mokymasis (*angl. Q-learning*), kurio tikslas yra tiesiogiai surasti apytikslę optimalią veiksmo vertės funkciją: $Q^*(s, a) \approx Q(s, a; \theta)$. Vieno-žingsnio (*angl. one-step*) Q-mokymosi algoritmuose vertės funkcijos $Q(s, a; \theta)$ parametrai θ yra išmokstami iteraciškai mažinant nuostolių (*angl. loss*) funkcijos seką, kur kiekviena i -toji nuostolių funkcija yra funkcija (1) ir s' yra būsena pasiekta po būsenos s .

$$L_i(\theta_i) = \mathbb{E} \left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right)^2 \quad (1)$$

Vieno-žingsnio Q-mokymosi algoritmas yra taip pavadintas, nes jo veiksmo vertės funkcija $Q(s, a)$ yra atnaujinama link vieno-žingsnio rezultato $r + \gamma \max_{a'} Q(s', a'; \theta)$. Vienas vieno-žingsnio metodo naudojimo trūkumas yra, kad gautas atlygis r paveikia tik tiesiogiai į jį atvedusią būsenos ir veiksmo porą s, a . Kitų būsenų ir veiksmų porų vertės paveikiamos tik netiesiogiai per atnaujintą $Q(s, a)$ vertę. Tai gali lemti labai lėtą mokymosi procesą, kadangi reikia labai daug atnaujinimų paskleisti (*angl. propagate*) atlygį į aktualias ankstesnes būsenas ir veiksmus.

Vienas būdas paskleisti atlygį greičiau yra naudoti n -žingsnių (*angl. n-step*) rezultatus [PW94; Wat89]. Naudojant n -žingsnių Q-mokymosi algoritmą, $Q(s, a)$ yra atnaujinama link n -žingsnių rezultato, apibrėžto: $r_t + \gamma r_{t+1} + \dots + \gamma^{n-1} r_{t+n-1} + \max_a \gamma^n Q(s_{t+n}, a)$. Taip pasiekama, kad vienas atlygis r tiesiogiai paveikia n ankstesnių būsenų ir veiksmų porų reikšmes ir

gaunamas potencialiai daug efektyvesnis aktualioms būsenos-veiksimo poroms atlygio skleidimo procesas.

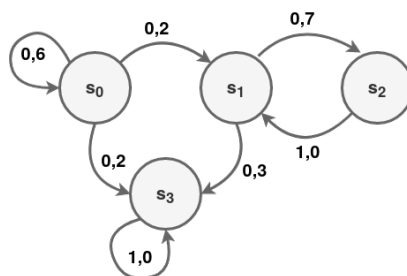
Atvirkščiai nei verte pagrįsti metodai, strategija pagrįsti ir modeliu nepagrįsti (poskyris 1.1) RL metodai tiesiogiai parametrizuoja strategiją $\pi(a|s; \theta)$ ir atnaušina parametrus θ atlikdami apytikslį gradiento padidinimą (*angl. gradient ascent*) $\mathbb{E}[R_t]$. Vienas tokio metodo pavyzdys yra REINFORCE šeimos algoritmai [Wil92]. Standartinis REINFORCE atnaušina strategijos parametrus θ kryptimi $\nabla_{\theta} \log \pi(a_t|s_t; \theta) R_t$, kas yra nešališkas (*angl. unbiased*) $\nabla_{\theta} \mathbb{E}[R_t]$ apskaičiavimas. Taip pat, yra įmanoma sumažinti šio apskaičiavimo dispersiją (*angl. variance*) išlaikant nešališkumą iš rezultato atimant išminktą būsenos funkciją $b_t(s_t)$, žinomą kaip bazė (*angl. baseline*) [Wil92]. Gautas gradientas yra $\nabla_{\theta} \log \pi(a_t|s_t; \theta)(R_t - b_t(s_t))$.

Vertės funkcijos išmoktas apskaičiavimas yra dažnai naudojamas kaip bazė $b_t(s_t) \approx V^{\pi}(s_t)$ vedanti link daug mažesnės strategijos gradiento dispersijos. Kai apytikslė vertės funkcija yra naudojama kaip bazė, skaičius $R_t - b_t$ gali būti panaudotas *pranašumo* (*angl. advantage*) apskaičiavimui veiksmui a_t būsenoje s_t arba $A(a_t, s_t) = Q(a_t, s_t) - V(s_t)$, nes R_t yra $Q^{\pi}(a_t, s_t)$ apskaičiavimas ir b_t yra $V^{\pi}(s_t)$ apskaičiavimas. Toks principas gali būti vadinamas aktoriaus-kritiko architektūra, kur strategija π yra aktorius ir bazė b_t yra kritikas [Ric98].

1.1.2. Markovo sprendimo priėmimo procesai

Pirmieji aprašyti Markovo sprendimo priėmimo procesai (*angl. Markov decision processes*, MDP) [Bel57] priminė Markovo grandines (*angl. Markov chains*).

Markovo grandinė – tai kiekviename žingsnyje atsitiktinai judantis iš vienos į kitą būseną (*angl. state*) procesas su fiksuotu skaičiumi būsenų, kur tikimybė pereiti iš būsenos s į būseną s' yra taip pat fiksuota ir priklauso tik nuo poros (s, s') , ne nuo jau praėjusių būsenų. Markovo grandinės neturi atminties [Gér17].

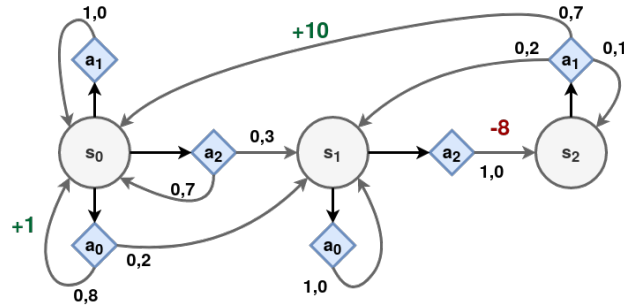


3 pav. Markovo grandinės pavyzdys

Paveikslėlyje 3 pavaizduotas Markovo grandinės pavyzdys su keturiomis būsenomis. Jeigu laikome būseną s_0 pradine, tai yra 70% tikimybė, kad procesas pasiliks šioje būsenoje ir kitą žingsnį. Po tam tikro kiekio žingsnių, procesas galiausiai paliks s_0 ir niekada nebegrįš į šią būseną,

nes jokia kita rodyklė nerodo į s_0 . Jei procesas pereis į būseną s_1 , tai yra labiausiai tikėtina (60% tikimybė), jog kita būsena bus s_2) ir tada iškart atgal į s_1 (100% tikimybė). Procesas gali pereiti per s_1 s_2 kelis kartus, prieš galiausiai patenkant į galinę būseną s_3 , kur procesas ir pasiliks.

MDP nuo Markovo grandinės skiriasi tuo, kad MDP perėjimai iš vienos būsenos į kitą, gali turėti jiems priskirtus atlygius (gali būti teigiami ir neigiami). RL problemos labai dažnai yra formuluojamos kaip MDP, kur agento tikslas yra surasti strategiją, kuri privestų prie didžiausio bendro atlygio per trumpiausią laiko tarpą [Gér17].



4 pav. Markovo proceso pavyzdys

Paveikslėlyje 4 pavaizduotas MDP pavyzdys su trimis būsenomis ir iki trijų diskrečių veiksmų per būseną. Jeigu laikome, kad agentas pradeda būsenoje s_0 , tai pirmame žingsnyje agentas gali pasirinkti vieną iš trijų galimų: a_0 , a_1 , a_2 veiksmų. Jeigu agentas pasirinktų atlikti veiksmą a_1 – jis garantuotai liktų būsenoje s_0 . Tačiau, jeigu agentas pasirinktų veiksmą a_0 – yra 80% tikimybė, gauti atlygį +1 ir likti toje pačioje būsenoje s_0 arba 20% tikimybė be atlygio patekti į būseną s_1 . Galiausiai arba a_0 , arba a_2 veiksmu agentas pateiks į būseną s_1 . Šioje būsenoje agentas gali pasirinkti tik vieną iš dviejų veiksmų: a_0 arba a_2 . Nors yra du galimi veiksmai, tik veiksmas a_2 veda į kitą būseną. Tačiau pasirinktus šį veiksmą agentas taip pat garantuotai gauna atlygį (bausmę) -8 . Pasiekus būseną s_3 yra galimas tik vienas veiksmas a_1 , bet galimi trys skirtingi rezultatai: likti toje pačioje būsenoje s_3 (10% tikimybė), pereiti į būseną s_1 (20% tikimybė) arba grįžti į pradinę būseną s_0 (70% tikimybė) ir gauti atlygį +10.

Optimaliai būsenos vertei bet kuriai būsenai s , žymimai $V^*(s)$, nustatyti, galima naudoti Belmano Optimalumo Lygtį (angl. *Bellman optimality equation*). Ši rekursyvi lygtis (2) parodo, kad jei agentas atlieka veiksmus optimaliai, tada optimali dabartinės būsenos reikšmė yra lygi vidutiniškai agento gaunamam atlygiui atlikus vieną optimalų veiksmą, plus visų įmanomų toliau einančių būsenų tikėtina optimali vertė.

$$V^* = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')] \text{ su visais } s \quad (2)$$

- **Transakcijų funkcija** (*angl. transaction function*) $T(s, a, s')$ yra perėjimo iš būsenos s į būseną s' tikimybė, agentui pasirinkus veiksmą a .
 - **Atlygio funkcija** (*angl. reward function*) $R(s, a, s')$ yra agento gaunamas atlygis, kai jis pereina iš būsenos s į būseną s' , agentui pasirinkus veiksmą a .
 - γ yra nuolaidos koeficientas.
- MDP taip pat gali būti užrašytas tokiu būdu: $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$.
- \mathcal{S} : visų būsenų rinkinys.
 - \mathcal{A} : visų veiksmų rinkinys.
 - $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$: transakcijų tikimybės pasiskirstymas $P(s'|s, a)$.
 - $R : \mathcal{S} \rightarrow \mathbb{R}$: atlygio funkcija, $R(s)$ yra atlygis būsenai s .
 - γ : nuolaidos koeficientas.

1.1.3. Sustiprinto mokymosi strategijos paieška

Agento naudojamas algoritmas veiksmo pasirinkimo sprendimui priimti yra vadinamas strategija. Strategija gali būti visiškai bet koks algoritmas, net nesvarbu ar jis yra stochastinis, ar deterministinis. Tačiau strategijos, kurios naudoja atsitiktines reikšmes yra vadinamos stochastinėmis strategijomis (*angl. stochastic policy*). Kintamieji, naudojami strategijos sprendimo priėmimui vadinami strategijos parametrais (*angl. policy parameters*) ir šių kintamųjų optimalių reikšmių ieškojimo procesas vadinamas strategijos paieška (*angl. policy search*). Strategijos parametrų įmanomų reikšmių kombinacijų rinkinys vadinamas strategijos plotu (*angl. policy space*) [Gér17].

Atlikti strategijos paiešką galima įvairiais būdais. Jei strategijos paieškos plotas yra reliatyviai mažas ir ieškoma vieno ar dviejų parametrų reikšmės, galima pasitelkti paprasčiausią brutalią jėgą (*angl. brute force*) ir išbandyti visus ar daugumą įmanomų variantų ir pasirinkti geriausią iš jų. Tačiau, paieškos plotui didėjant, gero strategijos parametrų rinkinio paieška sudėtingėja ir reikalingų resursų skaičius didėja eksponentiškai, todėl šitas sprendimas dažniausiai nėra taikomas.

Kitas būdas yra pasitelkti genetinius algoritmus [Vos99]. Sukuriant pirmąją kartą (*angl. generation*) strategijų su atsitiktinai parinktais strategijos parametrais ir iteratyviai šalinant blogiausius rezultatus rodančias vienetis bei atliekant atsitiktines mutacijas geriausiai pasirodžiusiųjų kopijoms. Taip galima iteruoti „išgyvenusiuųjų“ ir jų „vaikų“ kartas, kol pasiekama tinkama strategija.

Dar vienas sprendimas būtų naudoti optimizavimo metodus. Įvertinus atlygio gradientus, atsižvelgti į strategijos parametrus ir nežymiai juos pakeisti sekant gradientus link aukštesnio atlygio (gradiento pakilimas). Tai vadinama strategijos gradientais [Gér17].

1.1.3.1. Strategijos gradientai

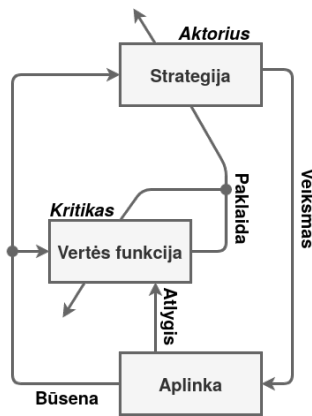
Strategijos gradientai (*angl. policy gradients*, PG) yra atsakingi už strategijos parametrų optimizavimą sekant aukštesnio atlygio link. REINFORCE algoritmas [Wil92] yra populiari PG klasė. Toliau aprašoma dažna variacija:

1. Neuroninio tinklo strategija sužaidžia žaidimą kelis kartus, apskaičiuoja kiekvieno žingsnio gradientus, kurie padidintų pasirinkto veiksmo pasirinkimo tikimybę ateityje, bet dar jų nepritaiko.
2. Po kelių sužaistų epizodų, apskaičiuojami taškai kiekvienam veiksmui.
3. Jei pasirinkto veiksmo atlygio taškai yra teigiami, reiškia tai buvo geras pasirinkimas ir anksčiau apskaičiuoti gradientai yra pritaikomi. Tačiau, jei veiksmo taškai yra neigiami, tada pritaikomi anksčiau apskaičiuotiems priešingi gradientai. Taip tinkamo veiksmo pasirinkimas tampa labiau tikėtinas ir netinkamo – mažiau.
4. Apskaičiuojamas visų galutinių gradientų vektorių vidurkis ir yra atliekamas Gradientų nusileidimo (*angl. Gradient Descent*) žingsnis.

1.1.4. Aktoriaus-kritiko principas

Aktoriaus-kritiko (*angl. actor-critic*) algoritmai yra algoritmų kategorija, kuri apjungia vertę pagrįstus ir strategija pagrįstus algoritmus. Šios algoritmų kategorijos tikslas yra pasinaudoti abiejų principų pranašumais ir pašalinti jų trūkumus. Pagrindinė idėja yra padalinti modelį į dvi dalis: viena atsakinga už veiksmo suradimą duotajai būsenai, kita už veiksmo vertės įvertinimą [Ric98].

Aktoriaus įvestis yra aplinkos būsena (paveikslėlis, 5), o išvestis geriausias jai veiksmas. Aktorius mokydamasis optimalios strategijos yra atsakingas už agento elgesį (strategija pagrįstas). Kritikas vertina veiksmo pasirinkimą apskaičiuodamas vertės funkciją (vertę pagrįstas). Abu modeliai dalyvauja žaidime ir laikui bėgant tampa geresni savo vaidmenyse. Gauta architektūra išmoksta žaisti žaidimą efektyviau nei abu metodai galėtų atskirai.



5 pav. Aktoriaus-kritiko schema

1.2. Dirbtiniai neuroniniai tinklai

Dirbtiniai neuroniniai tinklai (*angl. artificial neural networks*) (ANN) egzistuoja jau labai ilgą laiką, pirmos jų variacijos pristatytos dar 1943 metais [MP43]. Per tiek metų ANN implementacijos ir galimybės gerokai išaugo ir nors ANN perėjo per kelias populiarumo ir visiškos „užmiršties“ epochas, yra matoma nemažai priežasčių, kodėl ši technologija turės ir turi didelę įtaką mūsų gyvenimams [Gér17]. Keli pavyzdžiai:

- Dėl egzistuojančių didelių kiekių duomenų, kuriuos galima panaudoti ANN apmokymams bei dėl dažno ANN geresnių rezultatų nei kitos ML technikos pasiekimo naudojant sudėtingas ir labai dideles problemas.
- Dėl labai didelio kompiuterijos pasaulio patobulėjimo bei skaičiavimų galios išaugimo. 2000–2001 metais geriausiu laikytas superkompiuteris „ASCI White“¹ galėjo pasiekti teoretinį 12.3 TFLOPS² pajėgumą, kai šiuolaikinės vartotojams prieinamos GPU yra vertinamos net iki 14³ TFLOPS (ir nesveria šimtų tonų).
- Dėl dažno ANN pasiekiamų rezultatų atsiradimo žinių akiratyje, kas lemia didėjančią ir taip populiarių ANN technologijų finansavimą, greitėjančią progresą bei šią technologiją naudojančių naujų produktų atsiradimą[Gér17].

Toliau šiame skyriuje bus rašoma apie ANN ir jų variacijas bei susijusias technologijas.

¹<https://www.top500.org/resources/top-systems/asci-white-lawrence-livermore-national-laboratory/>

²TFLOPS arba teraFLOPS yra 10¹² FLOPS. FLOPS (*angl. floating point operations per second*) yra operacijų atliekamų per sekundę su slankaus kablelio skaičiais matmuo dažnai naudojamas kompiuterinių įrenginių pajėgumui nustatyti.

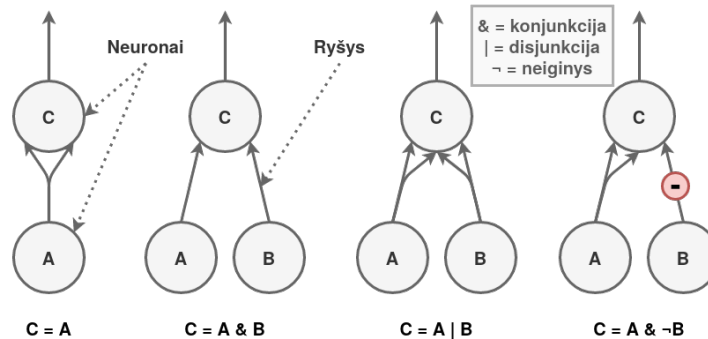
³<https://www.microway.com/knowledge-center-articles/comparison-of-nvidia-geforce-gpus-and-nvidia-tesla-gpus/>

1.2.1. Dirbtinių neuroninių tinklų architektūros komponentai

Šiame punkte aprašomi smulkūs ANN architektūros komponentai.

1.2.1.1. Dirbtinis neuronas

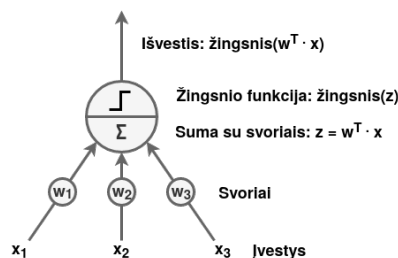
Dirbtinis neuronas (*angl. artificial neuron*) – paprastas biologinio neurono modelis, turintis vieną ar daugiau binarinių (įjungta/išjungta) įvesčių (ryšių) ir vieną binarinę išvestį. Dirbtinis neuronas aktyvuoja išvestį, kai tam tikras skaičius įvesčių yra aktyvus. Paveikslėlyje 6 pavaizduotas paprastas ANN, galintis atlikti įvairius loginius skaičiavimus, su sąlyga, kad neuronas aktyvuojamas, kai bent dvi įvestys yra aktyvios. Dirbtinis neuronas yra labai limituotas savo galimybėmis todėl sudėtingesnis neurono modelis yra reikalingas sudėtingėjant užduotims.



6 pav. Dirbtinio neurono atliekančio loginius skaičiavimus pavyzdys

1.2.1.2. Linijinis slenksčio vienetas

Linijinis slenksčio vienetas (*angl. linear threshold unit, LTU*) buvo pirmasis dirbtinis neuronas (papunktis 1.2.1.1). LTU įvestys ir išvestys yra skaičiai (ne binarinės reikšmės) ir kiekviena įvestis turi savo svorį. LTU suskaičiuoja sumą su įvesčių svoriais ($z = w_1x_1 + w_2x_2 + \dots + w_nx_n = \mathbf{w}^T \cdot \mathbf{x}$), tada suskaičiuotai sumai pritaiko žingsnio funkciją (*angl. step function*) ir išveda rezultatą: $h_{\mathbf{w}}(\mathbf{x}) = \text{žingsnis}(z) = \text{žingsnis}(\mathbf{w}^T \cdot \mathbf{x})$ (paveikslėlis 7).



7 pav. Linijinis slenksčio vienetas

1.2.1.3. Pagalbiniai neuronai

Formuojant ANN neužtenka tik skaičiavimus atliekančių neuronų. Be LTU ir dirbtinio neurono egzistuoja keli paprasti pagalbiniai neuronai:

- **Įvesties neuronas** (paveikslėlis 8a) išveda tuos pačius duomenis, kurie yra jam paduodame per įvestį.
- **Postūmio neuronas** (paveikslėlis 8b) neturi įvesties ir visada išveda tą pačią reikšmę (konstanta).



8 pav. Pagalbiniai neuronai

1.2.1.4. Dirbtinių neuroninių tinklų sluoksniai

Kiekvienas ANN yra sudaryti iš sluoksnių. Sluoksnis – tai tame pačiame ANN gylyje kartu dirbančių neuronų rinkinys. Yra keli pagrindiniai apibrėžimai susiję su ANN sluoksniais:

- **Įvesties sluoksnis** (*angl. input layer*) – tai pats pirmas sluoksnis ANN architektūroje. Dar vadinamas pereinamuoju (*angl. passthrough*), nes visi šiam sluoksniui paduoti duomenys yra tiesiogiai perduodami į tolimesnį sluoksnį be jokių pakeitimų.
- **Išvesties sluoksnis** (*angl. output layer*) – tai paskutinis sluoksnis ANN architektūroje, atsakingas už bendrą viso tinklo išvestą rezultatą.
- **Paslėptasis sluoksnis** (*angl. hidden layer*) – taip ANN architektūroje vadinami visi tarp įvesties ir išvesties sluoksnių esantys sluoksniai.

Pagal paslėptųjų sluoksnių kiekį ANN architektūros yra klasifikuojamos į dvi grupes:

- **Negilusis neuroninis tinklas** (*angl. shallow neural network*) – tai klasė tinklų, kurių architektūra yra sudaryta tik iš įvesties ir išvesties sluoksnių bei nėra nei vieno paslėpto sluoksnio.
- **Gilusis neuroninis tinklas** (*angl. deep neural network*) (DNN) – tai klasė tinklų, kurių architektūroje be įvesties ir išvesties sluoksnių yra bent vienas paslėptas sluoksnis.

1.2.2. Parceptronas

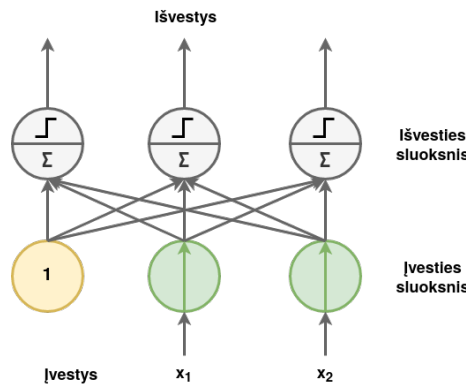
Perceptronas yra viena iš paprasčiausių ANN architektūrų ir yra paremta LTU (daugiau papunktis 1.2.1.2) [Ros57]. Perceptronuose dažniausiai naudojama sunkiosios pusės (*angl. heaviside*

step) žingsnio funkcija (3) arba kartai ženkle (*angl. sign*) funkcija (4).

$$\text{heaviside}(z) = \begin{cases} 0 & \text{jei } z < 0 \\ +1 & \text{jei } z \geq 0 \end{cases} \quad (3)$$

$$\text{sign}(z) = \begin{cases} -1 & \text{jei } z < 0 \\ 0 & \text{jei } z = 0 \\ +1 & \text{jei } z > 0 \end{cases} \quad (4)$$

Perceptronas sudarytas iš dviejų pilnai sujungtų sluoksnių (papunktis 1.2.1.4): sluoksnio sudaryto tik iš LTU ir sluoksnio sudaryto iš įvesties neuronų bei dažnai papildomo postūmio neurono ($x_0 = 1$), kuris visada grąžina 1. Paveikslėlyje 9 pavaizduotas perceptronas su dviem įvestimis ir trimis išvestimis (pavyzdžiui, galintis klasifikuoti į tris skirtingas klases).



9 pav. Perceptrono pavyzdys

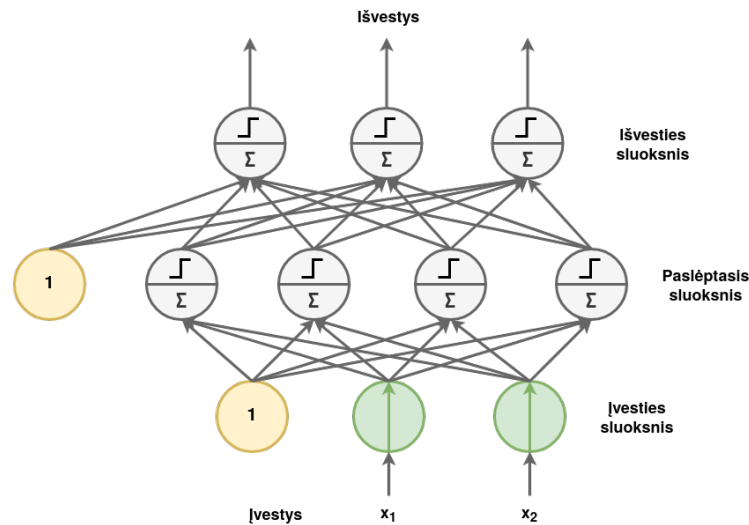
Perceptronai yra apmokomi naudojantis Hebo⁴ taisyklės variaciją, kuri atsižvelgia į tinklo padarytas klaidas ir mažina ryšių vedančių į neteisingą išvestį įtaką. Perceptrono mokymosi lygtis $w_{i,j}^{\text{ktias žingsnis}} = w_{i,j} + \eta(\hat{y}_j - y_j)x_i$ susideda iš $w_{i,j}$ yra ryšio svorio tarp i -tojo įvesties ir j -tojo išvesties neuronų, x_i i -tosios įvesties, \hat{y}_j j -tojo išvesties neurono išvesties, y_j j -tajo išvesties neurono išvesties tikslo reikšmės, bei η mokymosi greičio koeficiento.

1.2.3. Daugiasluoksniai perceptronai

Daugiasluoksniai perceptronai (*angl. multi-layer perceptron*) (MLP) yra pilnai sujungtas DNN sudarytas iš: įvesties sluoksnio, vieno ar daugiau paslėptų sluoksnių sudarytų iš LTU ir

⁴(*angl. Hebb's rule*) „Cells that fire together, wire together“ svoriai tarp neuronų yra didinami, jei jie turi vienodą išvestį.

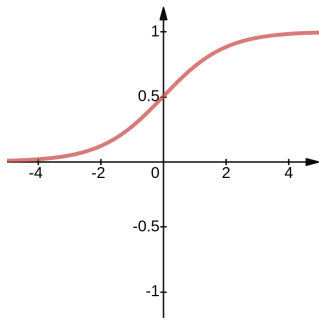
išvesties sluoksnio, taip pat sudaryto iš LTU. Visi, išskyrus išvesties, sluoksniai taip pat turi po vieną postūmio neuroną. Paveikslėlyje 10 pavaizduotas dviejų įvesčių, trijų išvesčių ir vieno paslėptąjo sluoksnio MLP.



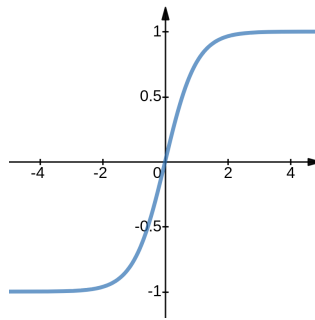
10 pav. Daugiasluoksnio perceptrono pavyzdys

MLP mokymui yra naudojamas atgalinio sklido (*angl. backpropagation*) mokymo algoritmas [RHW85], kai treniravimo metu pirma yra apskaičiuojama prognozė, išmatuojamas klaidingumas ir grįžtama atgal per sluoksnius apskaičiuojant kiekvieno sluoksnio įtaką gautam klaidingumui bei nežymiai keičiant ryšių svorius, ateities skaičiavimų klaidingumui sumažinti. Šiam procesui yra reikalingi gradientai, todėl MLP žingsnio funkcija yra pakeičiama į aktyvavimo funkciją (*angl. activation function*). Populiariausios aktyvavimo funkcijos:

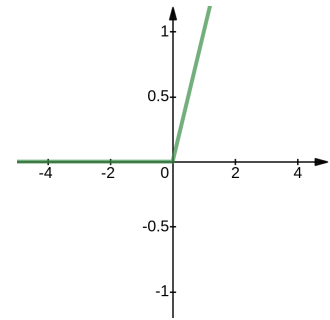
- **Logistinė funkcija** $\sigma(z) = (1 + \exp(-z))^{-1}$, su visomis reikšmėmis turi stipriai apibrėžtą nenulinę išvestinę, kas leidžia progresuoti kiekviename mokymo žingsnyje (paveikslėlis 11a).
- **Hiperbolinio tangento funkcija** $\tanh(z) = 2\sigma(2z) - 1$, panašiai kaip logistinė funkcija yra S-formos, tačiau funkcijos galimos reikšmės yra $(-1, 1)$ ribose, kas padeda normalizuoti sluoksnių išvestis ir pagreitina konvergenciją (paveikslėlis 11b).
- **ReLU funkcija** $\text{ReLU}(z) = \max(0, z)$, nors nėra diferencijuojama kai $z = 0$, praktiko pritaikyta veikia labai gerai. Didžiausias šios funkcijos privalumas – labai greitas apskaičiavimas (paveikslėlis 11c).



(a) Logistinė funkcija



(b) Hiperbolinė tangento funkcija



(c) ReLU funkcija

11 pav. Aktyvavimo funkcijų grafikai

MLP yra dažnai naudojamas klasifikavimo uždaviniuose. Kai klasės yra diskrečios (pavyzdžiui, 6 skirtingos klasės su indeksais nuo 0 iki 5), išvesties sluoksnio individualios aktyvavimo funkcijos yra dažnai pakeičiamos viena bendra minkštojo maksimumo ⁵ (*angl. softmax*) funkcija, kuri padeda normalizuoti išvestis [Gér17].

1.2.4. Konvoliuciniai neuroniniai tinklai

Konvoliuciniai neuroniniai tinklai (*angl. convolutional neural networks*) (CNN) yra viena iš pagrindinių ML kategorijų atliekant: vaizdų atpažinimą, vaizdų klasifikavimą, objektų aptikimą, veidų atpažinimą ir pan. Tačiau CNN nėra limituoti tik vizualine įvestimi, juos taip pat galima pritaikyti dirbant su garsu ar natūraliam kalbos mokymui (*angl. natural language processing*) [Gér17; Pra18]. Šiuolaikiniams CNN labai didelę įtaką turėjo regos smegenų žievės studijų įkvėptas neokognitronas⁶ [Fuk80] bei 1998 metais pristatyta garsi „LeNet-5“⁷ architektūra [LBB⁺98].

Be pilnai sujungtų sluoksnių, CNN turi du papildomus architektūrinius komponentus:

- **Konvoliucinis sluoksnis** (*angl. convolutional layers*).
- **Sutelkimo sluoksins** (*angl. pooling layer*).

Toliau šiame punkte bus aprašomi minėtieji CNN architektūriniai komponentai.

⁵Minkštojo maksimumo funkcija, dar žinoma kaip normalizuota eksponentės funkcija, yra funkcija, kuri realių skaičių vektoriu normalizuoja į tokio pat dydžio proporcingų tikimybių vektorių, kurio visų narių suma yra lygi 1. Matematinė išraiška: $\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$ su visais $i = 1, \dots, K$ ir $\mathbf{z} = (z_1, \dots, z_K) \in \mathbb{R}^K$.

⁶Neokognitronas – Kunihiko Fukushima 1979 metais pristatytas hierarchinis daugiasluoksnis ANN. Buvo naudojamas ranka parašytų simbolių bei pasikartojančių ypatybių (*angl. pattern*) atpažinimui.

⁷„LeNet-5“ – vienas pirmųjų CNN išplatintų gilųjų mokymąsi, plačiai naudojamas ant čekių ranka užrašytų skaičių atpažinimui.

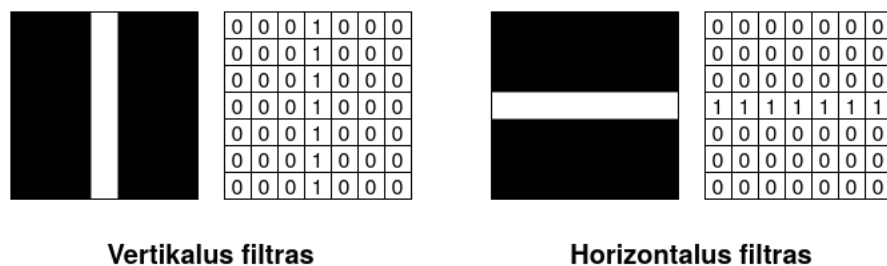
1.2.4.1. Konvoliuciniai sluoksniai

Konvoliuciniai⁸ sluoksniai yra pats svarbiausias CNN architektūrinis komponentas. Pirmojo konvoliucinio sluoksnio neuronai nėra jungiami su kiekviena tiriamojo objekto įvestimi (pavyzdžiui, paveikslėlio pikseliu), bet tik su įvestimis kurios priklauso jų matymo laukui (*angl. receptive field*). Tokiu pačiu principu, tolimesni konvoliuciniai sluoksniai yra jungiami su prieš tai buvusiais, kur gilesniame konvoliuciniame sluoksnyje esantis neuronas yra jungiamas tik su jo matymo laukui priklausančiais praeito sluoksnio neuronais. Pritaikius šią metodiką, žemesnio lygio sluoksniai yra atsakingi už paprastesnių požymių išskyrimą, o aukštesni už šių požymių apjungimą į sudėtingesnius.

Sluoksnių ir matymo laukų dydžiai ne visada persidengia lygiai. Tokiais atvejais yra laikoma, kad visos matymo lauko reikšmės išeinančios iš tiriamojo sluoksnio ribų yra nuliai. Tai vadinama papildymu nuliais (*angl. zero padding*).

Jeigu prie didesnio sluoksnio yra jungiamas mažesnis, vienas iš būdų tą pasiekti yra darant didesnius tarpus tarp matymo laukų. Atstumas tarp dviejų matymo laukų yra vadinamas žingsniu (*angl. stride*). Taip pat, verta paminėti, kad vertikalus ir horizontalus žingsniai neprivalo būti vienodo dydžio.

Neuronų svoriai gali būti atvaizduojami mažais paveikslėliais, vadinamais filtrais arba konvoliucijų branduoliais (*angl. convolution kernels*). Paveikslėlyje 12 pavaizduoti du galimi filtrų pavyzdžiai, kairėje atvaizduotas filtras išryškina vertikalias linijas duotame paveikslėlyje, dešinėje – horizontalias linijas išryškinantis filtras [Gér17; Saa17].



12 pav. Konvoliucinių filtrų pavyzdžiai

Konvoliucinio sluoksnio, kurio visi neuronai naudoja tą patį filtrą rezultatas yra vadinamas požymių žemėlapiu (*angl. feature map*). Požymių žemėlapis išryškina paveikslėlio dalis, kurios yra labiausiai panašios į filtrą. Mokymosi metu CNN ieško naudingiausių filtrų bei jų kombinacijų duotai užduočiai atlikti. Viename konvoliuciniame sluoksnyje įvesčiai yra pritaikomi iškart keli skirtingi filtrai, taip vienu metu aptinkami keli skirtingi požymiai. Kai yra naudojamas vienas

⁸Konvoliucija yra matematinė operacija, kuri per vieną funkciją pereina su kita ir išmatuoja taškinės daugybos integralą.

požymių žemėlapis, visi neuronai naudoja vienodus parametrus, tačiau skirtingi žemėlapiai gali naudoti skirtingus svorius ir poslinkius. Viena iš šio principo naudų yra: visi neuronai naudoja tuos pačius filtrus, kai CNN išmoksta naują taisyklę vienoje įvesties dalyje, vėliau ją gali atpažinti bet kur įvestyje.

Konvoliucinio sluoksnio neurono išvestį galima apskaičiuoti naudojantis formule (5), kuri apskaičiuoja visų įvesčių svorių ir postūmių sumą.

$$z_{i,j,k} = b_k + \sum_{u=1}^{f_h} \sum_{v=1}^{f_w} \sum_{k'=1}^{f'_n} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{kur} \quad \begin{cases} i' = u \cdot s_h + f_h - 1 \\ j' = v \cdot s_w + f_w - 1 \end{cases} \quad (5)$$

- $z_{i,j,k}$ konvoliucinio sluoksnio l eilutėje i , stulpelyje j , požymių žemėlapyje k esančio neurono išvestis.
- s_h ir s_w vertikalus ir horizontalus žingsniai.
- f_h ir f_w matymo lauko aukštis ir plotis.
- f'_n praeito sluoksnio $l - 1$ požymių žemėlapių skaičius.
- $x_{i',j',k'}$ sluoksnyje $l - 1$, eilutėje i' , stulpelyje j' , požymių žemėlapyje k' esančio neurono išvestis.
- b_k sluoksnyje l esančiam požymių žemėlapiui k priskirtas postūmis.
- $w_{u,v,k',k}$ ryšio svoris tarp neurono esančio požymių žemėlapyje k sluoksnyje l ir jo reliatyvios matymo laukui įvesties eilutėje u , stulpelyje v , požymių žemėlapyje k' .

1.2.4.2. Sutelkimo sluoksniai

Sutelkimo sluoksniai veikia labai panašiai kaip konvoliuciniai sluoksniai (žiūrėti papunktyje 1.2.4.1). Šio sluoksnio tikslas yra sumažinti (*angl. subsample*) įvesties objektą, taip padidinant skaičiavimų greitį bei sumažinant reikalingos atminties kiekį ir parametrų skaičių. Paveikslėlio dydžio sumažinimas padeda ANN toleruoti mažus pakitimus (pavyzdžiui, pasukimus).

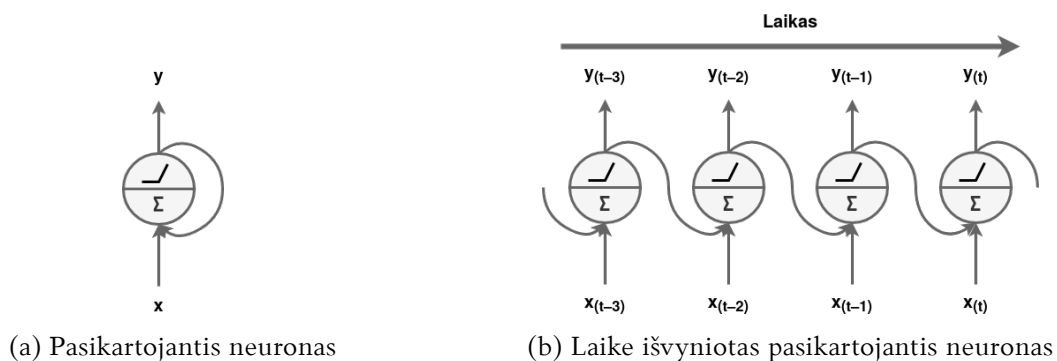
Sluoksnyje esančių neuronų įvestys yra sujungtos tik su matymo laukui priklausančia dalimi praeito sluoksnio išvesčių. Matymo laukas yra apibrėžiamas tais pačiais parametrais: dydžiu, žingsniu, užpildymo būdu. Sutelkimo, skirtingai nei konvoliucinio, sluoksnio neuronai neturi svorių. Vienintelė jų paskirtis yra surinkti ir sujungti įvestis pagal duotą funkciją (*angl. aggregation function*) (pavyzdžiui, didžiausios (*angl. max*) arba vidutinės (*angl. mean*) reikšmės). Dažniausiai naudojamas sutelkimo sluoksnis yra sutelkimo imant maksimalią reikšmę sluoksnis (*angl. max pooling layer*), kuris į tolimesnį sluoksnį perduoda tik kiekvieno matymo lauko didžiausias reikšmes [Gér17].

1.2.5. Rekurentiniai neuroniniai tinklai

Rekurentiniai neuroniniai tinklai (*angl. recurrent neural networks*) yra ANN klasė, kur ryšiai tarp sujungimo taškų formuoja kryptinį grafą (gali jungtis su prieš tai buvusiais neuronais) ir skirtingai nei perceptronas (punktas 1.2.2), MLP (punktas 1.2.3) ar CNN (punktas 1.2.4) tai nėra tik į priekį einantis (*angl. feedforward*) ANN. Ši architektūra leidžia RNN prognozuoti (iki tam tikro lygio) ateities įvykius bei analizuoti laiko eilučių (*angl. time series*) duomenis ar dirbti su nežinomo ilgio duomenų sekomis (*angl. sequences*) [Gér17].

1.2.5.1. Pasikartojantys neuronai

Neuronas, kuris gavęs įvestį apskaičiuoja išvestį ir perduoda naujai apskaičiuotą išvestį atgal sau kaip įvestį, yra vadinamas pasikartojančiu neuronu (*angl. recurrent neuron*) (paveikslėlis 13a). Paveikslėlyje 13b pavaizduotas laike išvyniotas (*angl. unrolled through time*) RNN sudaryto tik iš vieno pasikartojančio neurono pavyzdys, kur kiekvieną laiko žingsnį (*angl. time step*) t neuronas gauna įvestį x_t bei praeito žingsnio išvestį y_{t-1} .



13 pav. Pasikartojančio neurono pavyzdžiai

Sluoksniu iš pasikartojančių neuronų formavimas yra visiškai trivialus. Pagrindinis skirtumas, kad kiekviename laiko žingsnyje t kiekvienas neuronas gauna tiek visų naujų įvesčių vektorių \mathbf{x}_t , tiek praeito laiko žingsnio išvesčių vektorių \mathbf{y}_{t-1} . Be to, kiekvienas pasikartojantis neuronas turi du svorių rinkinius \mathbf{w}_x ir \mathbf{w}_y : vienas įvestims \mathbf{x}_t , kitas praeito laiko žingsnio išvestims \mathbf{y}_{t-1} . Tada pasikartojančio neurono išvestį galima apskaičiuoti pasinaudojus formule $y_t = \phi(\mathbf{x}_t^T \cdot \mathbf{w}_x + \mathbf{y}_{t-1}^T \cdot \mathbf{w}_y + b)$, kur $\phi()$ yra aktyvavimo funkcija (pavyzdžiui, ReLU) (daugiau, punkte 1.2.3), ir b yra postūmis [Gér17].

Pasinaudojus formule (6) galima apskaičiuoti viso sluoksniu išvestį vienai mini-partijai⁹ (*angl. mini-batch*).

⁹Mini-partija yra maža treniravimo duomenų dalis naudojama vienai mokymo iteracijai.

$$\begin{aligned}
\mathbf{Y}_{(t)} &= \phi(\mathbf{X}_{(t)} \cdot \mathbf{W}_x + \mathbf{Y}_{(t-1)} \cdot \mathbf{W}_y + \mathbf{b}) \\
&= \phi\left(\begin{bmatrix} \mathbf{X}_{(t)} & \mathbf{Y}_{(t-1)} \end{bmatrix} \cdot \mathbf{W} + \mathbf{b}\right) \quad \text{kur } \mathbf{W} = \begin{bmatrix} \mathbf{W}_x \\ \mathbf{W}_y \end{bmatrix}
\end{aligned} \tag{6}$$

- $\mathbf{Y}_{(t)}$ yra sluoksnio išvesties visai mini-partijai $m \times n_{\text{neuronai}}$ matrica laiko žingsnyje t , kur m yra mini-partijos dydis ir n_{neuronai} yra neuronų skaičius.
- $\mathbf{X}_{(t)}$ yra visų įvesčių $m \times n_{\text{įvestys}}$ matrica, kur $n_{\text{įvestys}}$ yra įvesčių požymių skaičius.
- \mathbf{W}_x yra dabartinio laiko žingsnio įvesties ryšių svorių $n_{\text{įvestys}} \times n_{\text{neuronai}}$ matrica.
- \mathbf{W}_y yra praeito laiko žingsnio išvesties ryšių svorių $n_{\text{neuronai}} \times n_{\text{neuronai}}$ matrica.
- \mathbf{b} yra visų neuronų postūmių n_{neuronai} dydžio vektorius.

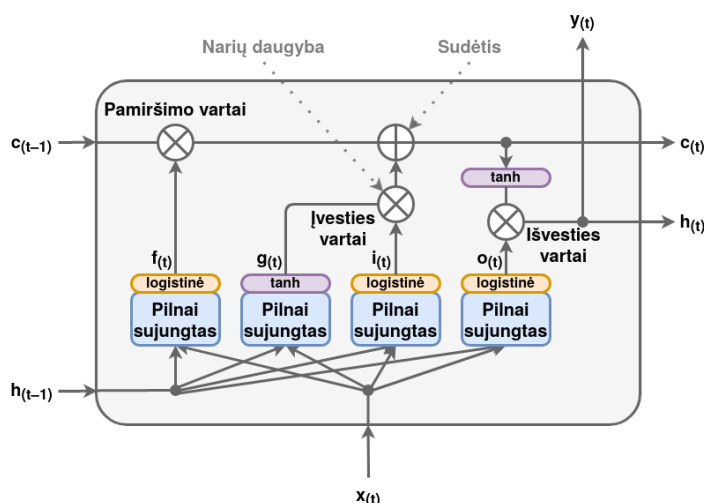
Jei skaičiuojama pirmo sluoksnio išvestis, buvusio laiko žingsnio išvestis (kadangi dar nėra įvykusi) dažniausiai laikoma nuliais.

Pasikartojančio neurono išvestis galima išreikšti $\mathbf{h}_{(t)}$ formule nuo laiko žingsnių t , kuri yra priklausoma nuo visų iki tol buvusių įvesčių $\mathbf{h}_{(t)} = f(\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)})$, kur $\mathbf{x}_{(t)}$ yra įvesčių vektorius žingsnyje t . Todėl tai galima vadinti atmintimi. Neuroninio tinklo dalis sauganti tam tikrą būseną laike yra vadinama atminties ląstele (*angl. memory cell*). Paveikslėlyje 13 pavaizduota pasikartojanti ląstelė ar iš jų sudarytas sluoksnis yra laikomi paprastosiomis atminties ląstelėmis (*angl. basic cell*).

1.2.5.2. Ilgos trumpalaikės atminties modelis

Ilgos trumpalaikės atminties modelio (*angl. long short-term memory*, LSTM) ląstelė [HS97; SSB14] gali būti naudojama panašiai kaip paprastosios atminties ląstelės, tačiau LSTM dažniausiai veiks geriau ir konverguos greičiau bei aptiks ilgalaikes priklausomybes duomenyse [Gér17].

LSTM ląstelės (paveikslėlis 14), skirtingai nei paprastoji atminties ląstelė turi du būsenos vektorius: $\mathbf{h}_{(t)}$ trumpalaikiai būsenai ir $\mathbf{c}_{(t)}$ – ilgalaikiai. Pagrindinė modelio idėja yra galimybė išmokyti ką laikyti ilgalaikėje būsenoje ir žinoti kokius duomenis atmesti, ir kokius naudoti. Tai galima pamatyti paveikslėlyje 14 sekant rodyklę $\mathbf{c}_{(t-1)}$, kurios pati pirma operacija ląstelėje yra vadinama pamiršimo vartais (*angl. forget gate*). Išmetus dalį ilgalaikės būsenos informacijos ir pridėjus naujai gautą informaciją iš naujos įvesties, gautas vektorius $\mathbf{c}_{(t)}$ yra tiesiai išvedamas kaip nauja ilgalaikė būsena.



14 pav. Ilgos trumpalaikės atminties modelis

LSTM (paveikslėlis 14) darbas prasideda su nauju įvesties vektorius $\mathbf{x}_{(t)}$ ir praeita trumpalaikė būsena $\mathbf{h}_{(t-1)}$, kurie yra paduodami į keturis skirtingus pilnai sujungtus sluoksnius. Kiekvienas iš šių sluoksnių turi skirtinga paskirtį:

- Pagrindinis sluoksnis yra su išvestimi $\mathbf{g}_{(t)}$. Jo paskirtis yra išanalizuoti naujai paduoto įvesties vektoriaus $\mathbf{x}_{(t)}$ ir praeitą trumpalaikių būsenų vektorių $\mathbf{h}_{(t-1)}$.
- Kiti trys sluoksniai yra vadinami vartų valdikliai (*angl. gate controllers*). Šie sluoksniai naudoja logistinę aktyvavimo funkciją, todėl jų išvestys yra $(0, 1)$ ribose. Jų visų išvestys yra paduodamos į atitinkamus vartus, kur yra atliekama narių daugyba¹⁰. Taip gauti 0 „uždaro“ vartus ir 1 „atidaro“.
- **Pamiršimo vartai** – valdomi vektorium $\mathbf{f}_{(t)}$ nustato kokia informacija turėtų būti pašalinama iš ilgalaikės būsenos.
- **Įvesties vartai** – valdomi vektorium $\mathbf{i}_{(t)}$ nustato kokia dalis vektoriaus $\mathbf{g}_{(t)}$ turėtų būti pridedama į ilgalaikę būseną.
- **Išvesties vartai** – valdomi vektorium $\mathbf{o}_{(t)}$ nustato kokia dalis ilgalaikės būsenos turėtų būti išvesta šį laiko žingsnį į vektorius $\mathbf{y}_{(t)}$ ir $\mathbf{h}_{(t)}$.

¹⁰Apskaičiuojamas „Hadamardo daugyba“ (*angl. Hadamard product*). Hadamardo produktas tai yra matrica gaunama tarpusavyje sudauginus visus dviejų vienodo dydžio matricių elementus esančius tose pačiose vietose. Pavyzdžiui,

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \circ \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 4 & 9 \\ 16 & 25 & 36 \end{bmatrix}.$$

2. Metodologija

Šiame skyriuje analizuojama Sokoban aplinka ir jos pasirinkimai. Taip pat analizuojama RL biblioteka ir jos pasirinkimai. Analizuojami algoritmai ir strategijos.

2.1. Sokoban žaidimas

Sokoban (iš japonų kalbos išvertus „sandėlio prižiūrėtojas“) yra 1981 metais Hiroyuki Ima-bayashi sukurtas tradicinis galvosūkis. Tai yra sudėtingas vieno žaidėjo žaidimas, kuriame tikslas yra vaikstant po labirintą priminantį „sandėlį“ nustumti dėžes ant tikslo langelių. Kai visos dėžės yra ant tikslo langelių – Sokoban laikomas išspręstu. Žaidimo sudėtingumas kyla iš jo apribojimų:

1. Žaidėjas gali judėti į visas keturias puses, tačiau negali pereiti kiaurai sienų ar dežių.
2. Žaidėjas gali pastumti šalia jo esančią dėžę, jei stūmimo kryptimi už jos esantis laukelis yra tuščias.
3. Žaidėjas negali pastumti daugiau nei vienos dėžės vienu metu.
4. Žaidėjas negali traukti dėžių.

Sokoban neturi bendrojo sprendinio ar sprendimo būdo. Yra įrodyta, kad Sokoban yra NP-hard¹¹ [DZ99] ir PSPACE-complete¹² [Cul97] uždavinys.

Papildomas sudėtingumo veiksnys yra neišsprendžiamų būsenų atliekant netinkamą veiksmą egzistavimas. Kadangi žaidėjui yra leidžiama tik stumti dėžes, situacijos, kur galvosūkis nebėra išsprendžiamas yra dažnos. Tokios žaidimo būsenos yra vadinamos aklavietėmis (*angl. deadlock*) [JS98]. Aptikti Sokoban aklavietes yra taip pat ar net sudėtingiau nei išspręsti patį galvosūkį, todėl tai irgi yra NP-hard uždavinys [Sch05].

2.1.1. Aplinka

Šiame punkte aprašoma aplinka ir jos pasirinkimo procesas.

2.1.1.1. Aplinkos pasirinkimas

Sokoban yra dažnas įvairių tyrimų objektas, todėl jam egzistuoja jau paruoštos aplinkos. Šias aplinkas galima panaudoti RL algoritmų apmokymams, taip pakartotinai nenaudojant resursų aplinkos implementacijai.

Galimos viešai prieinamos atviro kodo Sokoban aplinkos:

¹¹NP-hard yra problemos, kurios yra tokio pat ar didesnio sudėtingumo, nei sudėtingiausios žinomos NP problemos. NP problemos yra klasė skaičiavimo problemų, kurios yra išsprendžiamos nedeterministiniu būdu polinomi-niame laike.

¹²PSPACE-complete yra problemos, kurios gali būti išspręstos naudojantis polinominiu kiekiu atminties.

- **XSokoban**¹³ [Mye95] – moksliniuose darbuose dažna Sokoban žaidimo implementacija. XSokoban yra sudarytas iš 90 sudėtingų galvosūkių ir leidžia varžytis su kitais geriausiais Sokoban žaidėjais. Tačiau ši platforma veikia tik su Unix (Linux) sistemomis ir paskutinį kartą buvo atnaujinta tik 1997 metų liepą.
- **Gym-Sokoban** [Sch18] – ant OpenAI Gym [BCP⁺16] karkaso pastatyta Sokoban aplinka, kurios implementacija yra paremta „DeepMind¹⁴“ pateiktu taisyklių rinkiniu aplinkoms [RWR⁺17]. Taip pat ši aplinka generuoja kambarius atsitiktinai, kas leidžia mokyti DNN be perpildymo (*angl. overfitting*).

Darbai atlikti buvo pasirinkta naudoti Gym-Sokoban aplinką, dėl jos modernumo ir implementacijos paremtos OpenAI Gym karkasu, kuris yra tapęs RL aplinkų standartu. Taip pat, kadangi darbo metu bus naudojami modernūs DNN paremti RL algoritmai – atsitiktinis kambarių generavimas yra didelis pranašumas.

2.1.1.2. OpenAI Gym aplinkos

Gym yra, „OpenAI¹⁵“ kompanijos kuriamas, RL algoritmų kūrimo ir palyginimo įrankių rinkinys. Šie įrankiai nedaro jokių prielaidų apie agento struktūrą bei yra pritaikomi su daugeliu skaičiavimo bibliotekų [BCP⁺16].

`gym`¹⁶ yra atviro kodo Python biblioteka. Centrinis bibliotekos interfeisas yra unifikuota aplinkos sąsaja `Env`, kurios pagrindiniai metodai yra:

- **`reset(self) -> observation`** – atstato aplinkos parametrus į pradinę reikšmę, bei grąžina naują jos būseną.
- **`step(self, action) -> observation, reward, done, info`** – aplinkos laiko žingsnio metodas, kuriam paduodamas pasirinktas veiksmas ir grąžinama: nauja aplinkos būsena, atlikto veiksmo atlygis, `bool` tipo reikšmė, nurodanti ar aplinka buvo užbaigta, bei papildoma (paprastai nenaudotina algoritmo mokymui) informacija apie aplinką.
- **`render(self, mode='human')`** – atvaizduojamas vienas aplinkos kadras. Numatytoju būdu, metodas turėtų atlikti tai žmogui lengvai suprantamu būdu.

`Wrapper` yra dekoratoriaus dizaino modelio implementacija, leidžianti dinamiškai papildyti naujomis ar išplėsti esamas `Env` aplinkos objekto funkcijas.

¹³<http://www.cs.cornell.edu/andru/xsokoban.html>

¹⁴„Google“ priklausanti dirbtinio intelekto tyrimų kompanija, garsi savo „AlphaGo“ programa, 2016 metais laimėjusia prieš „Go“ žaidimo pasaulio čempioną Lee Sedol.

¹⁵Dirbtinio intelekto sprendimų vystymo ir pritaikymo kompanija. <https://openai.com/about/>

¹⁶<https://github.com/openai/gym>








2.1.1.3. Gym-Sokoban

`gym-sokoban` yra Python biblioteka, kuri leidžia sukurti OpenAI Gym (papunktis 2.1.1.2) paremtą Sokoban žaidimo aplinką pritaikytą RL algoritmų apmokymams. Pagrindinis bibliotekos objektas yra, `gym.Env` interfeiso implementacija, `SokobanEnv` – Sokoban žaidimo aplinka.

`gym-sokoban` turi penkias skirtingas Sokoban žaidimo variacijas (pavyzdžiui, variacija, kur žaidėjas gali stumti ir traukti dėžes), tačiau darbas bus atliekamas ir toliau aprašoma tik standartinė, artimiausia tradiciniam Sokoban žaidimui, variacija.

Kambariai yra sudaromi iš penkių tipų elementų: sienų, grindų, dėžių, tikslų ir žaidėjo, tačiau kai kurie iš jų gali persidengti ir turi kelias skirtingas būsenas. Visos šios kambario elementų ir būsenų variacijos yra reprezentuojamos žaidime atskiru 16×16 pikselių paveikslėliu (lentelė 1).

1 lentelė. Sokoban žaidimo aplinkos elementų būsenos ir jų grafinės reprezentacijos

Tipas	Būsena	Grafinė reprezentacija
Siena	Statinė	
Grindys	Tuščios	
Tikslas	Tuščias	
Dėžė	Ne ant tikslo	
Dėžė	Ant tikslo	
Žaidėjas	Ne ant tikslo	
Žaidėjas	Ant tikslo	

Standartinė aplinka priima devynis skirtingus veiksmus: nieko neveikimo veiksmas (0), po veiksmą dėžės pastūmimams (1–5) ir po veiksmą paėjimams (6–7) į visas keturias puses (aukštyn, žemyn, kairėn ir dešinėn). Judėjimo veiksmai yra atliekami tik tuo atveju, jei laukelis nurodyta kryptimi yra tuščias; pastūmimo veiksmai turi dvi variacijas, jei laukelis nurodytai kryptimi tuščias – paeinama į jį, jei tas laukelis yra dėžė ir už jos tuščia – pastumiama dėžė.

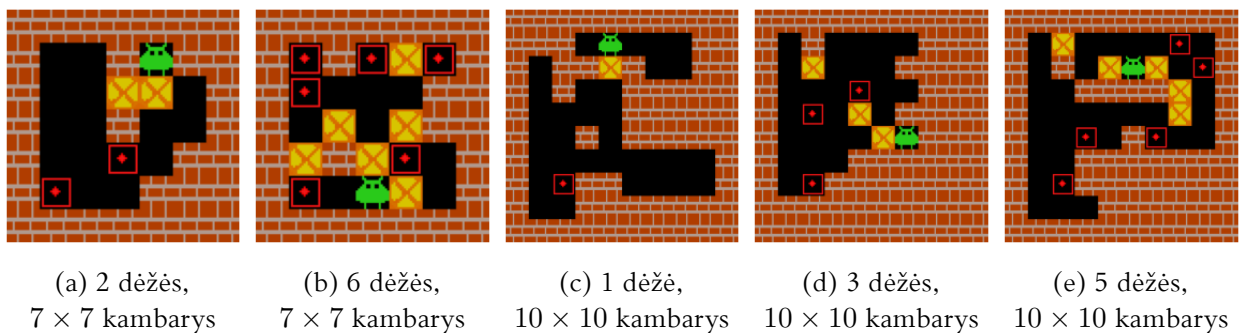
Visos RL aplinkos turi duoti agentui atlygį už atliktą veiksmą (poskyris 1.1). Gym-Sokoban aplinka galimai duoda vieną ar kelis iš keturių atlygių:

- **+10,0** – už visų dėžių buvimą ant tikslo laukelių.
- **+1,0** – už dėžės užstūmimą ant tikslo.
- **−1,0** – už dėžės nustūmimą nuo tikslo.
- **−0,1** – už atliktą veiksmą.

Aplinka bibliotekoje yra apibrėžiama `SokobanEnv` klase, kuri yra atsakinga už kambarių generavimą ir veiksmų interpretavimą, bei atlygio už atliktą veiksmą apskaičiavimą. Generuojamų kambarių savybės (paveikslėlis 15) yra nurodomos klasės konstruktoriumi, kurios leidžia modifikuoti parametrus:

- **Kambario dydį** (`tuple(int, int)`) – generuojamų kambarių aukštis ir plotis.
- **Dėžių kambaryje skaičių** (`int`) – kiek dėžių ir tikslo laukelių bus sugeneruojame per kambarį.
- **Maksimalų žingsnių skaičių** (`int`) – jei žaidėjas neužbaigia kambario per nurodytą žingsnių skaičių, aplinka yra užbaigiama.

Agentui aplinkos būsenos yra duodamos trimačiu $(16n_{\text{aukštis}}) \times (16n_{\text{plotis}}) \times n_{\text{kanalų skaičius}}$ dydžio masyvu, kur aukštis ir plotis yra kambario dydis laukeliais (dauginamas iš 16 dėl pikselių skaičiaus per laukelį), $n_{\text{kanalų skaičius}} = 3$ yra RGB spalvų kanalų skaičius, o reikšmės yra sveiki skaičiai $[0, 255]$ ribose.



15 pav. Sokoban žaidimo aplinkos skirtingo dydžio ir dėžių skaičiaus pavyzdžiai

2.2. Skatinamojo mokymosi biblioteka

Šiame poskyryje aprašoma skatinamojo mokymosi biblioteka ir jos pasirinkimo procesas.

2.3. Skatinamo mokymosi bibliotekos pasirinkimas

ML ir RL esant didelio susidomėjimo ir nuolatinių tyrimų objektais, natūraliai, moksliniuose darbuose pasiūlytos algoritmų implementacijos atsiranda ir cirkuliuoja viešai prieinamoje internetinėje erdvėje. Iš šių algoritmų yra formuojami rinkiniai ir RL bibliotekos. Tačiau bibliotekos turi savo naudojamus karkasus ir dažnai standartizuotą jų panaudojimo būdą. Renkantis RL biblioteką yra svarbu atsižvelgti į įvairius faktorius: bibliotekos algoritmų rinkinį ir jų modernumą, kuriamo modelio aplinką ir jo karkasą, algoritmų naudojamas skaičiavimų bibliotekas, bibliotekos apibrėžtų objektų išplečiamumą ar naujų implementacijos sudėtingumą, pačios bibliotekos naudojimo sudėtingumą, programavimo kalbą ir panašiai [Sim19].

Kelios dažniausiai naudojamos RL bibliotekos yra:

- **TF-Agents** [Ser18] – šiame sąrašė naujausia RL biblioteka, su daug žadančia moduline struktūra bei aukštos kokybės algoritmų implementacija. Ši biblioteka yra neoficialiai, kuriama pačio TensorFlow. Tačiau, kadangi tai yra jaunas produktas, biblioteka turi trūkumų: neturi dokumentacijos, dėl to, panaudojimas ar savo aplinkos pritaikymas nėra visiškai trivialus. Pilnas TensorBoard¹⁷ palaikymas.
- **KerasRL** [Pla16] – RL biblioteka paremta Keras¹⁸. Nors kodas yra gerai komentuotas ir naudojamas geras agento, strategijos ir atminties atskyrimas, biblioteka nebuvo atnaujinta nuo 2019 metų. Bibliotekos algoritmų rinkinys yra limituotas, nors rinkinyje esami algoritmai yra modernus. Naujos aplinkos pridėjimas nėra galimas, nebent aplinka yra paremta OpenAI Gym (papunktis 2.1.1.2).
- **Tensorforce** [KSF17] – modalių komponentų pagrindų sukurta, TensorFlow paremta RL biblioteka. Bibliotekos algoritmų rinkinyje yra didžioji dalis modernių RL algoritmų, tačiau pritaikymas nėra elementarus, dėl nepilnos algoritmų dokumentacijos. Biblioteka yra nuolat atnaujinama bei yra lengvai modifikuojama, tačiau kodas beveik neturi komentarų. Pilnas TensorBoard palaikymas.
- **OpenAI Baselines** [DHK⁺17] – aukštos kokybės modernių RL algoritmų implementacijų rinkinys. Tačiau dokumentacijos dėl dokumentacijos trūkumo, naudotis biblioteka, suprasti kodą ar pritaikyti savo aplinkai yra sudėtinga. Nėra palaikomas TensorBoard.
- **Stable Baselines** [HRE⁺18] – gerokai pagerinta OpenAI Baselines atšaka, su unifikuota algoritmų struktūra bei plačia dokumentacija. OpenAI Baselines modernių algoritmų implementacijos yra praplėstos naujais algoritmais, taip pat, originalus kodas yra papildytas komentarais ir TensorBoard palaikymu. Lengva pradėti naudotis, tačiau aplinkos yra limituotos OpenAI Gym (papunktis 2.1.1.2) karkaso.

Darbo metu buvo nuspręsta naudotis Stable Baselines, dėl kelių pagrindinių priežasčių. Kadangi Sokoban žaidimo pasirinkta aplinka (papunktis 2.1.1.1) yra paremta OpenAI Gym karkasu, tai nėra laikoma aplinkos trūkumu. Pagrindinis rodiklis buvo modernių algoritmų implementacijų rinkinys, kuo KerasRL nepasižymi ir buvo atmestas. Kitas rodiklis buvo dokumentacija ir jos kokybė, ko TF-Agents ir OpenAI Baselines bibliotekos neturi, o Tensorforce turi tačiau algoritmų dokumentacijos yra limituotos. Dar vienas rodiklis buvo, TensorBoard palaikymas, dėl galimybės stebėti algoritmo mokymą realiu laiku, ko neturi OpenAI Gym. Dėl šių priežasčių, pasirinkimo

¹⁷TensorFlow vizualizacijos įrankis. <https://github.com/tensorflow/tensorboard>

¹⁸Aukšto lygio ANN API parašyta su Python, kurios pagrindinis tikslas yra sudaryti sąlygas greitam eksperimentavimui.

sąrašas buvo susiaurintas iki dviejų: Stable Baselines ir Tensorforce. Nors abu pasirinkimai yra geri, buvo nuspręsta naudotis Stable Baselines dėl paprastesnės naudojimo metodikos.

2.3.1. Stable Baselines skatinamojo mokymosi biblioteka

Stable Baselines [HRE⁺18] yra rinkinys pagerintų RL algoritmų paremtų OpenAI Baselines [DHK⁺17] implementacijomis. Ši RL biblioteka sudaryta iš plataus modernių algoritmų pasirinkimo. Tačiau, priklausomai nuo aplinkos veiksmų ir būsenų tipų, ne visi algoritmai gali jai būti pritaikyti (lentelė 2). Taip pat, biblioteka yra padalinta į dvi dalis: algoritmus naudojančius OpenMPI¹⁹ jų lygiagretinimui, ir nenaudojančius OpenMPI.

2 lentelė. Stable Baselines algoritmai ir jų priimami aplinkų veiksmų ir būsenų tipai

Algoritmai	Galimi veiksmų tipai				Galimi būsenų tipai				MPI
	Discrete	Box	MultiDiscrete	MultiBinary	Discrete	Box	MultiDiscrete	MultiBinary	
A2C	✓	✓	✓	✓	✓	✓	✓	✓	✗
ACER	✓	✗	✗	✗	✓	✓	✓	✓	✗
ACKTR	✓	✓	✗	✗	✓	✓	✓	✓	✗
DDPG	✗	✓	✗	✗	✓	✓	✓	✓	✓
DQN	✓	✗	✗	✗	✓	✓	✓	✓	✗
GAIL	✓	✓	✗	✗	✓	✓	✓	✓	✓
PPO1	✓	✓	✓	✓	✓	✓	✓	✓	✓
PPO2	✓	✓	✓	✓	✓	✓	✓	✓	✗
SAC	✗	✓	✗	✗	✓	✓	✓	✓	✗
TD3	✗	✓	✗	✗	✓	✓	✓	✓	✗
TRPO	✓	✓	✓	✓	✓	✓	✓	✓	✓

Kadangi bakalauro darbui pasirinktos aplinkos (papunktis 2.1.1.3) veiksmų tipas yra Discrete²⁰, algoritmai, kurie nepalaiko šio tipo, nėra pritaikomi. Taip pat, autorių siūlymu, dėl esamos programinės klaidos, algoritmai naudojančys OpenMPI, darbo metu nebus naudojami.

¹⁹Žinučių perdavimo sąsajos biblioteka.

²⁰OpenAI Gym diskrečių sveikų skaičių masyvo tipas, kur sukurtas n dydžio kintamasis susideda iš $(0, 1, \dots, n-1)$ skaitmenų.

Atmetus algoritmus pagal šiuos du kriterijus lieka tik penki:

- **A2C** [MBM⁺16] – sinchroninis, determinuotas pranašumo aktorius-kritiko RL algoritmas, naudojantis kelis darbuotojus, pakartojimo buferio išvengimui.
- **ACER** [WBH⁺16] – aktorius-kritiko RL algoritmas, sujungia keletą RL algoritmų idėjų: naudojami keliais darbuotojais (kaip A2C), implementuoja patirties pakartojimo buferį (kaip DQN), naudoja atsekimo funkciją (*angl. Retrace*) Q-vertės įvertinimui, svarbos atrinkimui ir pasitikėjimo sričiai.
- **ACKTR** [WML⁺17] – aktorius-kritiko RL algoritmas, naudojami Kroneckerio (*angl. Kronecker-factored*) apskaičiuotą apytikslį kreivumą pasitikėjimo srities optimizavimui.
- **DQN** [MKS⁺13b] – RL algoritmas derinantis Q-mokymąsi su DNN, leidžiančiais RL dirbti sudėtingose, didelėse erdvėse.
- **PPO2** [SWD⁺17] – aktorius-kritiko RL algoritmas sujungia A2C (turinčio kelis darbuotojus) ir TRPO (algoritmas naudoja pasitikėjimo sritį aktorius gerinimui).

2.3.1.1. Stable Baselines Karkasas

Stable Baselines turi standartizuotą algoritmų naudojimo būdą. Pagrindiniai modelių metodai:

- **Konstruktorius** – konstruktoriui yra paduodami algoritmo ir strategijos hiper-parametrai bei aplinka. Grąžinamas paruoštas mokymui modelis.
- **learn(total_timesteps, ...)** – grąžina nurodytą laiko žingsnių kiekį apmokytą modelį. Taip pat, galima nurodyti ankstesnio atšaukimo ar kitas taisykles bei duomenų registravimo vardą ar dažnumą.
- **save(save_path, ...)** – išsaugo modelį į nurodyto adreso failą.
- **load(load_path, env=None, ...)** – užkrauna modelį iš failo arba iš Dictionary tipo objekto. Jei naudojama tik veiksmų prognozėms, aplinkos nurodyti nereikia, tačiau jei bus mokoma toliau, reikia nurodyti aplinką.
- **predict(observation, ...)** – padavus aplinkos būseną, modelis atlieką prognozės apskaičiavimą ir grąžina geriausią veiksmą. Galima papildomai nurodyti ar naudoti determinuotą principą ar ne.

Stable Baselines turi būdą sudėti kelias aplinkas į vieną vektorių. Šie aplinkų vektoriai yra pritaikomi daugeliui bibliotekos algoritmų ir leidžia vienu metu treniruoti modelį su keliomis aplinkomis. Jei naudojamos vektorinės aplinkos, vietoje būsenos ir veiksmų, modeliui yra paduodami aplinkų būsenų ir veiksmų vektoriai. Karkase yra apibrėžtos dviejų rūšių vektorinės aplinkos:

- **DummyVecEnv** – paprastas vektorinis adapteris, kviečiantis visas aplinkas iš eilės.
- **SubprocVecEnv** – daugiaprocesis vektorinis adapteris, kur kiekviena aplinka turi savo procesą ir kviečiant aplinkos funkciją, kadangi visos aplinkos veikia lygiagrečiai, atlieką skaičiavimus vienu metu.

Mokymo metu galima modeliui paduoti iškviečiamas (*angl. callback*) funkcijas, kurios priklausomai nuo funkcijos gali būti iškviečiamos nurodytą mokymo proceso etapą. Naudojant šias funkcijas, galima pasiekti vidines modelio būsenas bei nutraukti mokymo procesą anksčiau. Iškviečiamų funkcijų rašymui yra apibrėžtas `BaseCallback` interfeisas. Taip pat, yra kelios funkcijų implementacijos, kurias galima panaudoti skirtingų užduočių atlikimui: periodiniam saugojimui, modelio tarpiniam vertinimui, išankstinio modelio mokymo stabdymui pasiekus nurodytą slenksutinį atlygį ir panašiai.

2.3.2. Stable Baselines strategijos

Stable Baselines turi rinkinį numatytųjų strategijų, paremtų ANN (poskyris 1.2). Pagrindinės dvi kategorijos yra MLP (punktas 1.2.3) ir CNN (punktas 1.2.4). Tačiau, kadangi bakalauro darbui pasirinktos aplinkos būsenos yra pateikiamos kadro pikselių masyvais (papunktis 2.1.1.3), mokymo metu buvo pasirinktos tik CNN strategijos.

Stable Baselines turi trijų rūšių CNN paremtas strategijas: paprastą CNN, LSTM (papunktis 1.2.5.2) su CNN požymių išskyrimu, normalizuotą LSTM su CNN požymių išskyrimu. Toliau, šiame punkte bus aprašomos šios trys strategijos.

2.3.2.1. CnnPolicy strategijos aprašymas

`CnnPolicy` yra viena iš Stable Baselines bibliotekos numatytųjų strategijų. Ši strategija naudoja CNN (punktas 1.2.4) požymių išskyrimui ir dėl to yra geriausiai pritaikoma aplinkoms, kurių būsenos yra paduodamos aplinkos kadrų pikselių masyvais.

Strategijos DNN architektūra yra sudaryta iš trijų konvoliucinių sluoksnių (papunktis 1.2.4.1) ir vieno paslėpto pilnai sujungto sluoksnio (lentelė 3). Visi šie sluoksniai naudoja ReLU aktyvavimo funkcijas (punktas 1.2.3), tačiau išvesties aktyvavimo funkcija priklauso nuo aplinkos veiksmo tipo (lentelė 2). Bakalauro darbui parinkta Sokoban aplinka (papunktis 2.1.1.3) naudoja `Discrete` tipo veiksmus, tokiu atveju išvesčiai naudojama bendra minkštojo maksimumo aktyvavimo funkcija (punktas 1.2.3).

3 lentelė. CnnPolicy strategijos ANN architektūrą sudarančių sluoksnių parametrai

Pavadinimas	Sluoksnio tipas	Filtrų skaičius	Filtrų dydis	Žingsnis	Neuronų skaičius
c1	Konvoliucinis	32	8	4	–
c2	Konvoliucinis	64	4	2	–
c3	Konvoliucinis	64	3	1	–
fc1	Pilnai sujungtas	–	–	–	512

2.3.2.2. CnnLstmPolicy strategijos aprašymas

CnnLstmPolicy strategija yra labai panaši CnnPolicy strategijai (papunktis 2.3.2.1). Tačiau šios strategijos architektūra yra RNN (punktas 1.2.5) ir tie patys CnnPolicy sluoksniai yra naudojami LSTM atminties ląstelės (papunktis 1.2.5.2) įvesčiai. LSTM ląstelė yra standartinė, su keturiais pilnai sujungtais sluoksniais, kurių kiekvieno dydis yra 256 neuronai. Išvestis yra valdoma identišškai CnnPolicy strategijai, priklausomai nuo aplinkos veiksmų tipo. Bakalauro darbo aplinkos atveju – minkštojo maksimumo bendra aktyvavimo funkcija (punktas 1.2.3).

2.3.2.3. CnnLnLstmPolicy strategijos aprašymas

CnnLnLstmPolicy strategija naudoja tą pačią CnnLstmPolicy strategijos (papunktis 2.3.2.2) architektūrą su vienu skirtumu: LSTM ląstelės sluoksniai yra normalizuoti naudojant sluoksnių normalizaciją (*angl. layer normalization*). Normalizuojant RNN, efektyviai stabilizuojama paslėptoji ląstelės būseną bei pasiekiamas greitesnis mokymosi laikas [BKH16]. Visos kitos strategijos savybės yra identiškos CnnLstmPolicy strategijai.

2.3.3. Stable Baselines algoritmai

Iš Stable Baselines algoritmų rinkinio tik penki yra pritaikomi (pункte 2.3.1) pasirinktai Sokoban aplinkai (papunktis 2.1.1.3): A2C, ACER, ACKTR, DQN ir POP2. Tačiau, darbo rašymo metu, ACKTR kode yra programinė klaida, kuri neleidžia naudoti CnnLnLstmPolicy strategijos (papunktis 2.3.2.3) su bibliotekoje neapibrėžta aplinka, todėl šis algoritmas buvo atmestas. DQN taip pat buvo atmestas, tačiau dėl algoritmo vektorizuotų aplinkų (papunktis 2.3.1.1) nepalaikymo, kas žymiai sulėtintų mokymosi procesą.

Toliau šiame punkte bus aprašomi trys pasirinktai aplinkai pritaikomi ar kitaip neatmesti algoritmai: A2C, ACER ir POP2.

2.3.3.1. A2C algoritmo aprašymas

A3C [MBM⁺16] yra RL algoritmas veikiantis aktorius-kritiko principu (papunktis 1.1.4). Šis algoritmas kombinuoja kelias pagrindines idėjas:

- Atnaujinimo schema veikia naudojant fiksuoto ilgio patirties dalis, kurios yra naudojamos pranašumo ir rezultatų funkcijų įvertinimo apskaičiavimams.
- Architektūra dalijasi sluoksniais tarp strategijos ir vertės funkcijų.
- Asinchroniniai atnaujinimai.

A3C sulaukus didelio populiarumo, buvo sukurta sinchroninė, deterministinė algoritmo implementacija, kuri laukia kol visi aktoriai pabaigia jų patirties segmentą, prieš pradedant atnaujinimus pagal visų aktorių rezultatų vidurkį. Šis algoritmas yra vadinamas A2C. Vienas papildomas sinchroninės A3C algoritmo versijos A2C yra efektyvesnis GPU panaudojimas. A2C algoritmo autoriai teigia, kad jų sinchroninė implementacija rodo geresnius rezultatus nei asinchroninė, bei yra efektyvesnė[Wu19].

Stable Baselines A2C algoritmo implementacija turi originaliame darbe aprašytas numatytasias hiper-parametrų reikšmes (priedas nr. 1). Bakalauro darbo metu, bus naudojamas algoritmas su jo numatytaisiais hiper-parametrais.

2.3.3.2. ACER algoritmo aprašymas

ACER [WBH⁺16] – tai yra aktorius-kritiko principu (papunktis 1.1.4) veikiantis algoritmas or naudojantis A2C pristatytas idėjas (papunktis 2.3.3.1), tačiau į šio algoritmo implementaciją taip pat pridėtas patirties pakartojimo buferis (*angl. experience replay buffer*). Papildomai, ACER naudoja: atsekamą Q-vertės įvertinimą, svarbos atranką (*angl. importance sampling*) bei pasitikėjimo sritį (*angl. trust region*).

Dėl naudojamo patirties pakartojimo buferio, algoritmo mokymas yra efektyvesnis, ir reikalingų žingsnių skaičius konvergavimui yra daug mažesnis. Tačiau papildoma papildomi laikomi duomenys reikalauja daugiau operatyviosios bei grafinės atminties resursų.

Stable Baselines ACER implementacijos numatytosios reikšmės yra paremtos originalaus tiriamojo darbo hiper-parametrais (priedas nr. 1), kurie bus naudojami bakalauro darbo metu.

2.3.3.3. PPO2 algoritmo aprašymas

PPO2 yra OpenAI [DHK⁺17] PPO algoritmo [SWD⁺17] implementacija pritaikyta GPU. Daugiaprocesiam veikimui PPO2 naudoja aplinkų vektorius (papunktis 2.3.1.1), kai PPO1 tam naudoja OpenMPI (punktas 2.3.1). Taip pat, PPO2 autorių teigimu, veikia tris kartus greičiau

nei bazinė PPO naudojant klasikinių žaidimų aplinką, Atari [Sch19].

PPO yra aktoriaus-kritiko principu (papunktis 1.1.4) paremtas algoritmas, kombinuojantis A2C (papunktis 2.3.3.1) kelių aktorių naudojimo ir taip pat pasitikėjimo srities idėjas.

Stable Baselines pateikta PPO2 algoritmo implementacija naudoja OpenAI algoritmą ir jo pateiktas numatytąsias hiper-parametrų reikšmes (priedas nr. 1), tačiau algoritmas turi kelis pakeitimus ir nėra identiškas originaliam algoritmo pristatymo moksliniam darbui: vertės funkcija yra apkarpyta (*angl. clipped*) ir pranašumai yra normalizuoti. Bakalauro darbo metu naudojamas algoritmas su numatytomis hiper-parametrų reikšmėmis.

3. Eksperimentai

Šiame skyriuje aprašomi bakalauro darbo metu atlikti eksperimentai bei jiems paruošta eksperimentinė aplinka.

3.1. Eksperimentinės aplinkos paruošimas

Eksperimentai atlikti naudojantis kompiuteriu su „Ubuntu“ OS. Minėtame kompiuteryje įdiegta „Anaconda“ paketų valdymo ir dislokavimo sistema, naudojama aplinkų atskyrimui. Didžioji programinė dalis eksperimento atliekama „Jupyter Notebook“ programavimo aplinkoje naudojantis „Python“ kalba.

3.1.1. Eksperimentinės aplinkos specifikacijos

Eksperimentas atliekamas naudojantis kompiuteriu su „Ubuntu“ OS.

1. Kompiuterio techninė specifikacija:

- (a) Procesorius (CPU) – „**Intel Core i5-9600K**“ (6 branduoliai, bazinis greitis 3.70 GHz).
- (b) Grafinė vaizdo plokštė (GPU) – „**Nvidia GeForce RTX 2070 Super**“ (8GB GDDR6, 1770 MHz).
- (c) Operatyvioji atmintis – „**HyperX Predator Black**“ (32GB, 3200MHz, DDR4, CL16).
 - i. Papildomai paskirta: **16GB** „Swap“ atminties²¹.
- (d) Pastovioji atmintis – „**Western Digital**“ (1TB).

2. Kompiuterio programinė įranga:

- (a) Operacinė sistema – „**Ubuntu 18.04 LTS**“ (versija: **18.04.4 LTS**).
- (b) Paketų ir aplinkų valdymo sistema – „**Anaconda**“ (versija: **2020.02**).
- (c) Programavimo kalba – „**Python**“ (versija: **3.7.6**).
- (d) Atviro kodo programa kintančio kodo, matematinių funkcijų, teksto bei duomenų vizualizavimui – „**Jupyter Notebook**“ (versija: **6.0.3**).
- (e) Atviro kodo ML platforma su lanksčia įrankių ir bibliotekų ekosistema skirta šiuolaikinių ML sprendimų kūrimui ir gerinimui – „**TensorFlow**“ (versija: **1.14.0**).
- (f) „TensorFlow“ vizualizavimo įrankis – „**TensorBoard**“ (versija: **1.14.0**).
- (g) RL algoritmų kūrimo ir vertinimo įrankių rinkinys – „**OpenAI Gym**“ (versija: **0.17.1**).

²¹ „Swap“ atmintis – tai pastoviojoje atmintyje paskirta atminties dalis virtualiai operatyviajai atminčiai, kuri yra naudojama, kai vykdomoms operacijoms trūksta fizinės operatyviosios atminties.

- (h) Modernių RL algoritmų implementacijų rinkinys (*angl. state-of-art*) – „**Stable Baselines**“ (versija: **2.10.1a0**).
- (i) Išskirstyta VSC sistema pakeitimų sekimui programiniame kode – „**Git**“ (versija: **2.23.0**)

3.1.2. Skatinamojo mokymo modelio paruošimas

Šiame punkte aprašomi Sokoban aplinkai padaryti pakeitimai bei mokymo modelio paruošimo procesas.

3.1.2.1. Sokoban aplinkos paruošimas

Gym-Sokoban (papunktis 2.1.1.3) veiksmų aibė susideda iš devynių skirtingų veiksmų, tačiau dėl eksperimentinės aplinkos ribotų resursų ir mokymui skirtu laiku, aibė buvo sumažinta iki penkių. Viena iš veiksmų kiekio mažinimo priežasčių: stūmimo ir paprasto paėjimo veiksmų persiliejimas. Atliekant pastūmimo veiksmą, jei atitinkama kryptimi yra dėžė – ji yra pastumiama, tačiau, jei dėžės ar kitokios kliūtys nėra – vykdomas paprasto paėjimo veiksmas. Dėl šios priežasties, paprasto pastūmimo veiksmai agentui nėra būtini ir buvo išmesti, taip supaprastinant modelį. Aplinkos veiksmų sumažinimui buvo parašytas dekoratorius (priedas nr. 2) pagal `gym.Wrapper` interfeisą (papunktis 2.1.1.2).

Tradiciškai, Sokoban žaidimo agento kokybė yra vertinama pagal išspręstų galvosukių procentą, o ne pagal atlygį. Iš Stable Baselines (punktas 2.3.1) TensorBoard integracijoje stebimų kintamųjų nėra būdo išvesti tokios reikšmės. Tam buvo sukurtas dar vienas aplinkos dekoratorius (priedas nr. 2), renkantis sėkmingai išspręstų ir neišspręstų aplinkų skaičius, kuriuos yra leidžiama pasiekti kitiems objektams. Tai yra ypač aktualu, mokant aplinkas su daugiau nei viena dėže, kur neužtenka patikrinti ar galutinis agento, aplinkoje surinktas, atligis yra virš minimalios reikšmės, nes žaidėjas gali būti užstūmęs dėžę ant tikslo laukelio ir gavęs tam tikrą atlygį.

3.1.2.2. Mokymo aplinkos paruošimas

Eksperimentui buvo paruoštos dvi mokymo aplinkos: viena skirtingų algoritmų ir strategijų apmokymui naudojant paprastas Sokoban žaidimo aplinkas (priedas nr. 3) jų palyginimui, kita mokymo žinių perdavimui su sudėtingesne Sokoban žaidimo aplinka (priedas nr. 4). Žinių perdavimui buvo sukurtas papildoma iškvietimo funkcija (papunktis 2.3.1.1), kuri vizualizuoja sėkmingai išspręstų kambarių dalį Tensorboard įrankyje.

3.2. Eksperimento planas

Bakalauro darbo metu atliktas eksperimentas susideda iš trijų dalių. Kiekvienai daliai sudarytas planas: kaip bus atliekamas eksperimentas, kokia bus naudojama aplinka ir kokių rezultatų yra tikimasi.

1. Pirma eksperimento dalis: **geriausios strategijos ieškojimas.**

- (a) **Tikslas:** palyginti Sokoban žaidimo agento mokymosi efektyvumą paprastoje aplinkoje su skirtingomis tiriamomis strategijomis ir geriausiai pasirodžiusią taikyti tolimesniuose tyrimuose.
- (b) **Aplinka:** Sokoban žaidimo aplinka iš 7×7 dydžio kambarių su viena dėže per kambarį.
- (c) **Atlikimo procesas:** atsitiktinai parenkamas ir apmokomas vienas tiriamasis algoritmas (punktas 2.3.3). Mokymas atliekamas kelis kartus, su visomis tiriamomis strategijomis (punktas 2.3.2).
- (d) **Tikėtini rezultatai:** sudėtingesnė strategija, naudojanti LSTM atminties ląstelės modelį (papunktis 1.2.5.2) mokysis efektyviau. Taip pat tikimasi, jog normalizuota LSTM strategijos variacija mokysis greičiau nei nenormalizuota.

2. Antra eksperimento dalis: **geriausio algoritmo ieškojimas.**

- (a) **Tikslas:** palyginti Sokoban žaidimo agento mokymosi efektyvumą paprastoje aplinkoje su skirtingais tiriamaisiais algoritmais.
- (b) **Aplinka:** ta pati, kaip ir pirmoje dalyje.
- (c) **Atlikimo procesas:** apmokomi visi tiriamieji algoritmai naudojantis tuos pačius mokymo kriterijus ir aplinką kaip pirmoje dalyje. Mokymui pritaikoma pirmoje dalyje geriausiai pasirodžiusi strategija.
- (d) **Tikėtini rezultatai:** visų algoritmų panašus efektyvumas, tačiau ACER turėtų pasiekti patenkinamus rezultatus per mažesni žingsnių skaičių nei A2C, dėl patirties pakartojimo buferio. Taip pat, A2C laiko atžvilgiu turėtų baigti mokymosi procesą daug greičiau nei kiti algoritmai, dėl paprastesnės architektūros. Galiausiai, PPO2 yra paremtas A2C su pridėtais pagerinimais, tikimasi jog PPO2 mokysis greičiau nei A2C laiko žingsnių atžvilgiu.

3. Trečia eksperimento dalis: **mokymo žinių perdavimo naudos tyrimas.**

- (a) **Tikslas:** palyginti Sokoban žaidimo agento mokymo efektyvumą sudėtingesnėje aplinkoje, tarp apmokymo nuo pradžių ir apmokymo naudojant žinių perdavimo principą, pirmiausiai agentą apmokius paprastesnėje aplinkoje, tada sudėtingesnėje.

- (b) **Aplinka:** Sokoban žaidimo aplinka iš 10×10 dydžio kambarių su dviem dėžėmis per kambarį.
- (c) **Atlikimo procesas:** bus apmokomi du agentai, vienas bazinis, atramos taško nustatymui, kitas naudojantis žinių perdavimą. Sokoban žaidimo mokymui naudojami, pirmos ir antros eksperimentų dalių pagrįsti, algoritmai ir strategija. Abiejų mokymų atveju, agentai mokomi iki parinktų mokymo nutraukimo kriterijų.
- (d) **Tikėtini rezultatai:** panaudoju žinių perdavimą Sokoban žaidimo agentui, modelį galima apmokyti iki aukštesnių išsprendimo rezultatų. Taip pat tikimasi, pasiekti bazinio modelio rezultatus per mažesnę laiko žingsnių kiekį ir laiką.

3.3. Eksperimento eiga

Remiantis eksperimento planu (punktas 3.2), buvo atliktas eksperimentas. Toliau aprašoma kiekvienos eksperimento dalies eiga:

1. Pirmas eksperimento dalis: **geriausios strategijos ieškojimas**.
 - (a) Atsitiktinai parinktas A2C algoritmas (papunktis 2.3.3.1).
 - (b) Plane aprašytai Sokoban žaidimo aplinkai uždėtas 200 žingsnių limitas per kambarį ir sudarytas 64 aplinkų dydžio paprastas vektorius (papunktis 2.3.1.1) efektyvesniam mokymui pasiekti.
 - (c) Apmokymui paruošta ir naudota mokymo programa (priedas nr. 3).
 - (d) Mokymai atlikti su trimis strategijomis (punktas 2.3.2): `CnnPolicy`, `CnnLstmPolicy` ir `CnnLnLstmPolicy`.
 - (e) Mokymo eigoje rezultatai registruoti naudojantis TensorBoard.
 - (f) Galutiniai modeliai išsaugoti į ZIP failus.
2. Antra eksperimento dalis: **geriausio algoritmo ieškojimas**.
 - (a) Mokymams naudota `CnnLnLstmPolicy` strategija.
 - (b) Mokymo metu naudota ta pati aplinka ir mokymo programa kaip ir pirmos eksperimento dalies metu.
 - (c) Apmokyti trys Sokoban žaidimo agentai su skirtingais algoritmais (punktas 2.3.3): A2C, ACER ir PPO2.
 - (d) Mokymo eigoje rezultatai registruoti naudojantis TensorBoard.
 - (e) Galutiniai modeliai išsaugoti į ZIP failus.
3. Trečia eksperimento dalis: **mokymo žinių perdavimo naudos tyrimas**.
 - (a) Mokymams naudota `CnnLnLstmPolicy` strategija.

- (b) Plane aprašyti Sokoban žaidimo aplinkai uždėtas 100 žingsnių limitas per kambarį ir sudarytas aplinkų vektorius. Dėl atminties limito ir didesnių kambarių (lyginant su pirma ir antra eksperimento dalimis), aplinkos vektoriaus dydis sumažintas iki 16 aplinkų bei naudojama daugiaprocesė variacija efektyvesniam resursų išnaudojimui pasiekti.
- (c) Apmokymui paruošta ir naudota mokymo programa (priedas nr. 4).
- (d) Parinkti mokymo nutraukimo kriterijai: mokymas nutraukiamas apmokius bent penkias iteracijas (iteracijos dydis 10^6 laiko žingsnių), jei paskutinėje iteracijoje išspręstų kambarių dalis yra didesnė nei 50% bei absoliutus skirtumas tarp paskutinių penkių iteracijų išspręstų kambarių dalies ir paskutinės iteracijos yra mažiau nei 1%, arba paskutinės iteracijos išspręstų kambarių dalis didesnė nei 90%. Šie kriterijai padėjo nustatyti modelio mokymosi sulėtėjimą ir stabilumą, taip nutraukiant mokymą, kai tolimesnis mokymas tampa lėtas ir brangus resursų atžvilgiu.
- (e) Apmokyti modeliai su visais tiriamaisiais algoritmais su sudėtinga aplinka, atramos taškom nustatymui.
- (f) Apmokyti Sokoban žaidimo agentai, su paprastesne aplinka – viena dėže.
- (g) Sukurti nauji Sokoban žaidimo agentai, kuriems perduotos paprastos aplinkos apmokymo žinios ir mokymas tęstas sudėtingesnėje aplinkoje.
- (h) Mokymo eigoje rezultatai registruoti naudojantis TensorBoard.
- (i) Galutiniai ir tarpiniai modeliai išsaugoti į ZIP failus.

3.4. Eksperimento rezultatų analizė

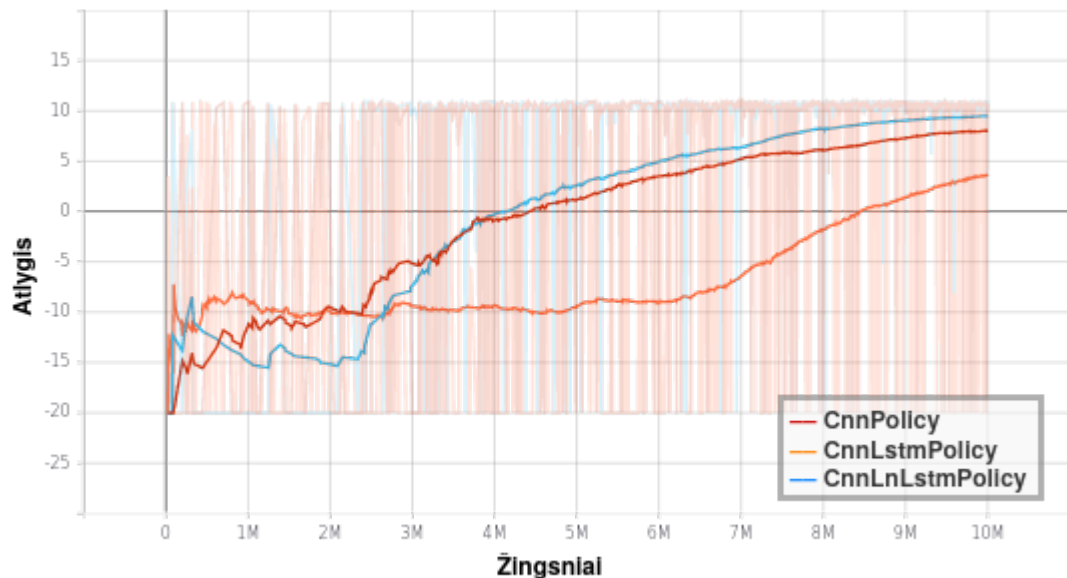
Šiame poskyryje aprašoma eksperimento metu gautų duomenų analizė. Eksperimentas atliktas iš trijų dalių.

3.4.1. Pirmos eksperimento dalies: geriausios strategijos ieškojimo rezultatai

Paveikslėlyje 16 atvaizduoti agentų mokymo procesų su A2C algoritmu ir skirtingomis strategijomis rezultatai. Pastebėta, kad `CnnPolicy` ir `CnnLstmPolicy` strategijos mokymasis pagreitėja panašiai $2,5 \cdot 10^6$ žingsnių taške, tačiau `CnnLstmPolicy` nepradedą mokytis iki $7 \cdot 10^6$. Kadangi mokymas buvo apribotas iki 10^7 žingsnių, `CnnLstmPolicy` nepasiekė geriausių galimų rezultatų. Todėl planavimo metu padaryta prognozė, kad sudėtingesnės LSTM atminties ląstelės modelio strategijos mokysis greičiau, nepasitvirtino.

Vienas iš tyrimo tikslų buvo išrinkti strategiją tolimesniems tyrimams. `CnnLstmPolicy`

strategija parodė geriausius rezultatus testavimo ribose, todėl buvo toliau naudota ir kituose tyrimuose.

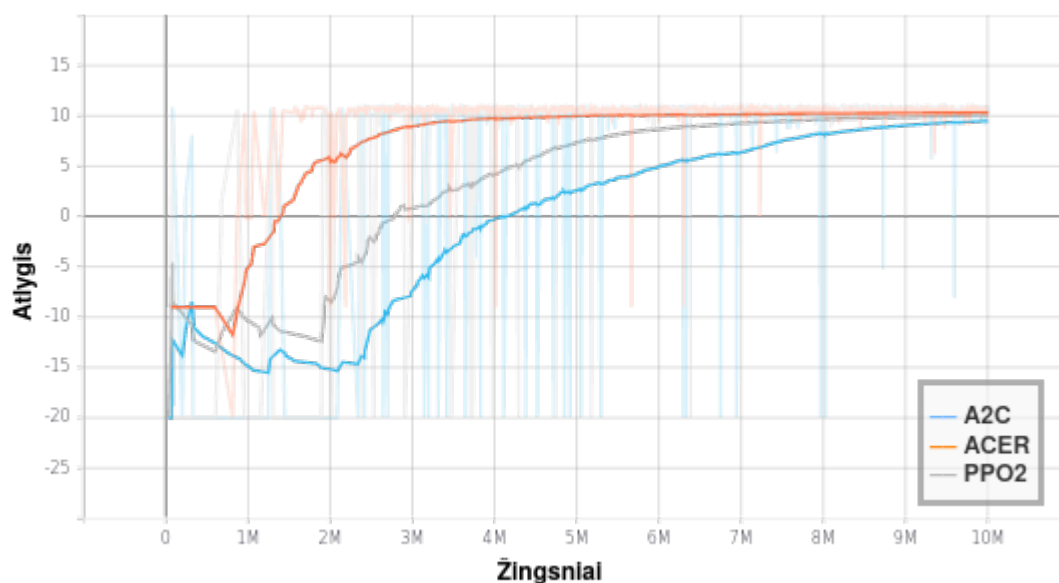


16 pav. Agento mokymosi procesas su skirtingomis strategijomis. Tyrimai atlikti paprastoje Sokoban žaidimo aplinkoje, t.y. 7×7 dydžio su viena dėže kambariai. Raudona linija – A2C algoritmas naudojantis CnnPolicy strategija; oranžinė linija – A2C algoritmas naudojantis CnnLstmPolicy strategija; mėlyna linija – A2C algoritmas naudojantis CnnLnLstmPolicy strategija.

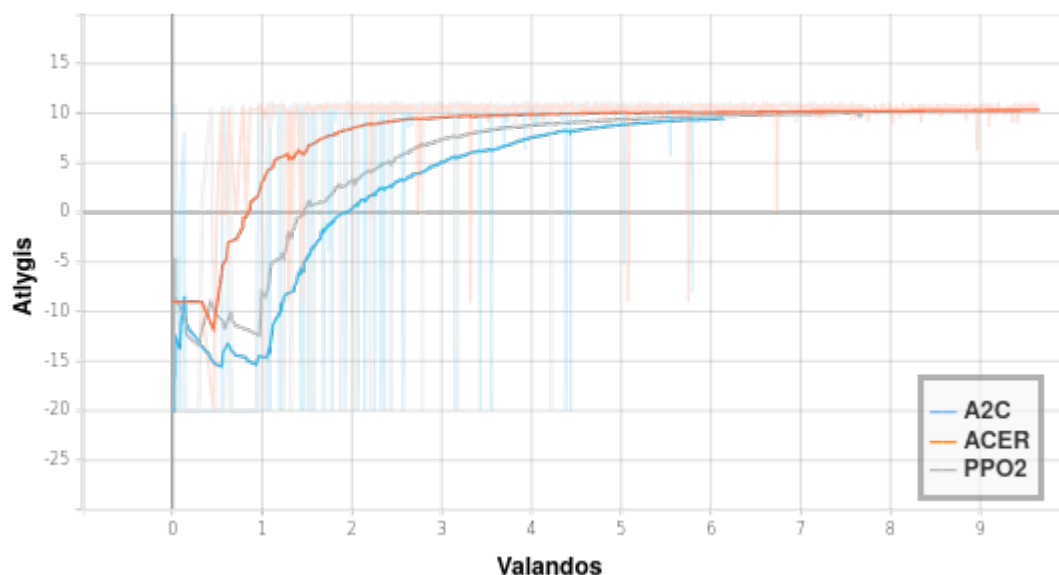
3.4.2. Antros eksperimento dalies: geriausio algoritmo ieškojimo rezultatai

Paveikslėlyje 17 atvaizduoti agentų mokymo procesų su skirtingais algoritmais rezultatai. Kaip ir prognozuota, visi trys algoritmai pasiekia panašaus efektyvumo rezultatus, tačiau visiems šiems algoritmams prireikė skirtingo laiko žingsnių skaičiaus. ACER pasiekia optimalius rezultatus po $3 \cdot 10^6$ žingsnių, PPO2 panašius rezultatus pasiekia apie $4 \cdot 10^6$ žingsnyje, o A2C prireikia visų testavimo žingsnių (10^7) pasiekti tuos pačius rezultatus.

Nors ACER mokėsi efektyviausiai žingsnių atžvilgiu, mokymui nustatyti 10^7 žingsnių užtruko beveik 10 valandų, kai PPO2 užtrunka 8, o A2C – 6 valandas (paveikslėlis 18).



17 pav. Agento mokymosi procesas su skirtingais algoritmais. Tyrimai atlikti paprastoje Sokoban žaidimo aplinkoje, t.y. 7×7 dydžio su viena dėže kambariai. Mėlyna linija – A2C algoritmas naudojantis CnnLnLstmPolicy strategija; oranžinė linija – ACER algoritmas naudojantis CnnLnLstmPolicy strategija; pilka linija – PPO2 algoritmas naudojantis CnnLnLstmPolicy strategija.



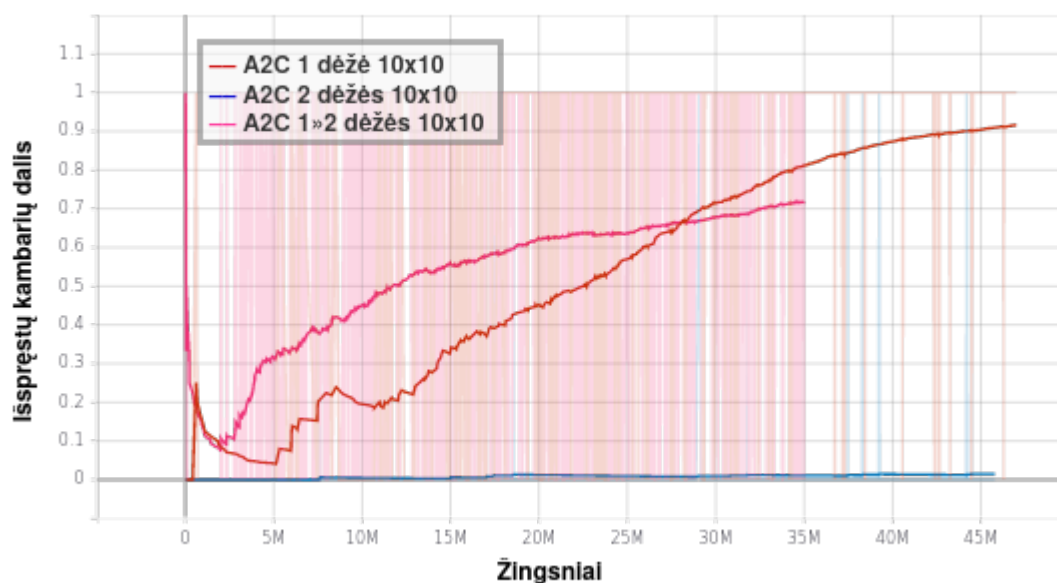
18 pav. Agento mokymosi procesas su skirtingais algoritmais, laike. Tyrimai atlikti paprastoje Sokoban žaidimo aplinkoje, t.y. 7×7 dydžio su viena dėže kambariai. Mėlyna linija – A2C algoritmas naudojantis CnnLnLstmPolicy strategija; oranžinė linija – ACER algoritmas naudojantis CnnLnLstmPolicy strategija; pilka linija – PPO2 algoritmas naudojantis CnnLnLstmPolicy strategija.

3.4.3. Trečios eksperimento dalies: mokymo žinių perdavimo naudos tyrimo rezultatai

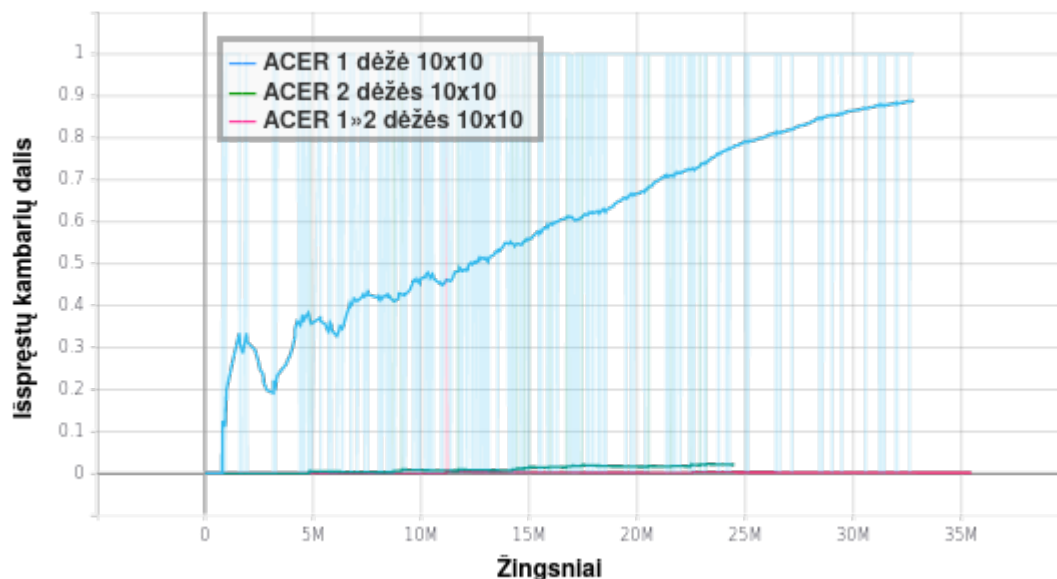
Paveikslėlyje 19 atvaizduoti A2C agentų mokymo procesų rezultatai. Bazinio modelio mokymas buvo nutrauktas po $4,5 \cdot 10^7$ žingsnių ir 50 valandų, dėl mokymosi progreso nebuvimo. Tokie pat rezultatai buvo užfiksuoti su ACER algoritmu (paveikslėlis 20), kurio bazinio modelio mokymas buvo nutrauktas po $3,5 \cdot 10^7$ ir 48 valandų. PPO2 (paveikslėlis 21) kita vertus, mokėsi ir su sudėtinga aplinka, tačiau mokymas buvo nutrauktas po $4,8 \cdot 10^7$ žingsnių ir 70 valandų, dėl nuolatos lėtėjančio mokymo proceso.

Nutraukus bazinių modelių mokymą, buvo pradėta mokyti modelius su paprastesne Sokoban žaidimo aplinka, tačiau didesnio (nei pirmoje ir antroje eksperimento dalyse) ploto kambariais. Pastebėta, kad ACER algoritmas, nors rodė gerus rezultatus antroje eksperimento dalyje, naujame teste laiko atžvilgiu pasirodė labai panašiai kaip A2C. Abu algoritmai apmokant agentą užtruko apie 37 valandas, o PPO2 užtruko tik 22 valandas, iki kol buvo pasiekta 90% iteracijos išspręstų kambarių dalis (lentelė 4, papunktis 3.1.2.1).

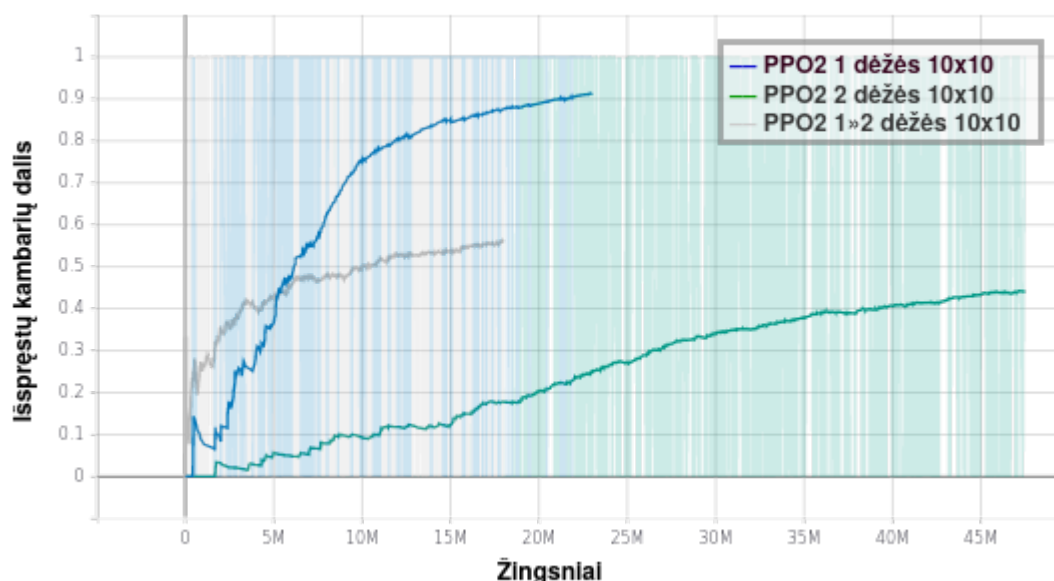
Apmokius agentus su Sokoban aplinkomis, kuriose žaidėjas turėjo išspręsti galvosūkį tik su viena dėže, agento svoriai buvo perkelti į naują agentą, kuris tęsė mokymą ant sudėtingesnės aplinkos, su dviem dėžėmis. Pradžioje (paveikslėlis 19), visų naujų agentų išspręstų kambarių kiekis, natūraliai, nukrito iki beveik nulio. Tačiau PPO2 ir A2C pakankamai greitai pradėjo mokytis naujoje aplinkoje ir pasiekė eigoje minėtus kriterijus per 30 ir 68 valandas atitinkamai (A2C lėtesnis). ACER nepradėjo mokytis ir po 36 valandų, mokymas buvo nutrauktas. PPO2 agentas pradėjo mokytis anksčiau, bet reliatyviai greitai sulėtėjo iki stabdymo kriterijaus reikšmės ir pasiekė tik apie 55% kambarių išsprendimo dalį, kita vertus, A2C nors mokėsi lėčiau, tačiau tai atliko stabiliau ir pasiekė virš 70% galvosūkių su dviem dėžėmis išsprendimo dalį (lentelė 4, papunktis 3.1.2.1).



19 pav. Agento mokymosi procesas su A2C algoritmu ir žinių perdavimu. Tyrimai atliekami Sokoban žaidimo aplinkoje su 10×10 dydžio kambariais. Raudona linija – A2C algoritmo mokymas kambariuose su viena dėže; mėlyna linija – A2C algoritmo mokymas kambariuose su dviem dėžėmis; rožinė linija – A2C algoritmo mokymas dviejų dėžių kambariuose su perduotomis žiniomis iš mokymosi su viena dėže.



20 pav. Agento mokymosi procesas su ACER algoritmu ir žinių perdavimu. Tyrimai atliekami Sokoban žaidimo aplinkoje su 10×10 dydžio kambariais. Mėlyna linija – ACER algoritmo mokymas kambariuose su viena dėže; žalia linija – ACER algoritmo mokymas kambariuose su dviem dėžėmis; rožinė linija – ACER algoritmo mokymas dviejų dėžių kambariuose su perduotomis žiniomis iš mokymosi su viena dėže.



21 pav. Agento mokymosi procesas su PPO2 algoritmu ir žinių perdavimu. Tyrimai atliekami Sokoban žaidimo aplinkoje su 10×10 dydžio kambariais. Mėlyna linija – PPO2 algoritmo mokymas kambariuose su viena dėže; žalia linija – PPO2 algoritmo mokymas kambariuose su dviem dėžėmis; pilka linija – PPO2 algoritmo mokymas dviejų dėžių kambariuose su perduotomis žiniomis iš mokymosi su viena dėže.

3.5. Eksperimento apibendrinimas ir išvados

Eksperimentas parodė, kad pritaikant Stable Baselines (punktas 2.3.1) algoritmų (A2C, ACER ir PPO2) implementacijas su numatytaisiais hiper-parametrais ir kitais nustatymais, Sokoban žaidimo agento mokymui, geriausias pasirinkimas iš tirtųjų yra `CnnLnLstmPolicy` strategija (lentelė 4). Kadangi tyrimai atlikti su mažu (7) Sokoban žaidimo aplinkos kambariu ir paprastomis sąlygomis (1 dėžė per kambarį), yra tikimybė, jog kitokios kambario specifikacijos rodytų kitokius rezultatus. Tačiau, jei mokomas paprastas Sokoban žaidimo agentas, `CnnPolicy` ir `CnnLnLstmPolicy` yra geros strategijos.

Algoritmų mokymo palyginimas naudojant paprastą Sokoban žaidimo aplinką parodė, kad ACER ir PPO2 algoritmai mokosi greičiau nei A2C algoritmas žingsnių atžvilgiu. Tačiau šis tyrimas buvo atliktas su paprasta aplinka. Vėlesniuose tyrimuose buvo pastebėta, kad ACER algoritmas, kuris pasirodė geriausiai paprastos aplinkos mokyme, visiškai nesimoko sudėtingesnėje ir didesnėje aplinkoje, bent su autorių numatytaisiais nustatymais ir testuota Sokoban aplinka. Todėl, jei yra mokomas agentas paprastoje Sokoban žaidimo aplinkoje, ACER algoritmas yra geriausias pasirinkimas.

Sokoban žaidimo agentas naudojantis PPO2 algoritmą parodė gerus rezultatus tiek papras-

tos aplinkos apmokyme, tiek sudėtingos. Taip pat, PPO2 buvo vienintelis algoritmas, kuris (nors ir lėtai), bet mokėsi sudėtingoje Sokoban žaidimo aplinkoje be žinių perdavimo. PPO2 agentas, lyginant su kitais tirtais, greičiausiai pasiekė mokymo kriterijus (daugiau nei 90% išspręstų kambarių dalis) sudėtingoje aplinkoje. Dėl to, mokant Sokoban žaidimo agentą be žinių perdavimo su didesniais (10×10) ar sudėtingesniais (dvi dėžės) kambariais, geriausias pasirinkimas yra PPO2 algoritmas.

Labiausiai eksperimento metu pasižymėjo Sokoban žaidimo agentai su A2C algoritmu. Nors šie agentai neparodė geriausių rezultatų lyginant su ACER ir PPO2 algoritmu paprastoje Sokoban žaidimo aplinkoje, jų mokymosi rezultatai buvo stabiliausi sudėtingoje aplinkoje. A2C agentai, iš visų tirtų algoritmų, pasiekė aukščiausią Sokoban galvosūkių su dviem dėžėmis išsprendimo procentą (72%) prieš pasiekdami sustabdymo kriterijus. Nors bandant apmokyti su ta pačia aplinka nuo pradžių, agentas nerodė jokio progreso. Tai parodė, kad žinių perdavimas Sokoban žaidimo agentui paremtu A2C algoritmu, yra naudingas ir padeda agentui pasiekti aukštesnių išsprendimo rezultatų.

Žinių perdavimas padėjo ne tik A2C, bet ir PPO2 algoritmui. PPO2 algoritmo Sokoban žaidimo agentas su žinių perdavimu pasiekė aukštesnius rezultatus ir daug greičiau nei be: agentas be žinių perdavimo užtruko 70 valandų ir pasiekė 44%, agentai apmokyti su paprastesne aplinka ir perduotomis žiniomis užtruko bendrai 53 valandas ir pasiekė 56% išsprendimo procentą (papunktis 3.1.2.1).

4 lentelė. Eksperimento metu atlikti mokymai. Dėžės skaičiai su rodyklėmis žymi mokymosi žinių perdavimą.

Algoritmas ir strategija	Kambarių dydis	Dėžių skaičius	Mokymosi laikas	Mokymosi žingsniai (10^7)	Mokymosi atlygis	Išsprendimo procentas
A2C	7×7	1	5 val.	1	8,02	–
CnnPolicy			18 min.			
A2C	7×7	1	4 val.	1	3,67	–
CnnLstmPolicy			41 min.			
A2C	7×7	1	6 val.	1	9,49	–
CnnLnLstmPolicy			8 min.			
ACER	7×7	1	9 val.	1	10,06	–
CnnLnLstmPolicy			39 min.			

4 lentelė. Eksperimento metu atlikti mokymai. Dėžės skaičiai su rodyklėmis žymi mokymosi žinių perdavimą.

Algoritmas ir strategija	Kambarių dydis	Dėžių skaičius	Mokymosi laikas	Mokymosi žingsniai (10^7)	Mokymosi atlygis	Išsprendimo procentas
PPO2 CnnLnLstmPolicy	7×7	1	7 val. 38 min.	1	-9,60	-
A2C CnnLnLstmPolicy	10×10	1	67 val. 18 min.	4,7	8,52	92%
A2C CnnLnLstmPolicy	10×10	2	50 val. 12 min.	4,6	10,32	1%
A2C CnnLnLstmPolicy	10×10	$1 \rightarrow 2$	37 val. 4 min.	3,5	4,43	72%
ACER CnnLnLstmPolicy	10×10	1	38 val. 27 min.	3,3	8,74	92%
ACER CnnLnLstmPolicy	10×10	2	35 val. 47 min.	2,5	-9,64	2%
ACER CnnLnLstmPolicy	10×10	$1 \rightarrow 2$	48 val. 5 min.	3,5	-9,85	0%
PPO2 CnnLnLstmPolicy	10×10	1	22 val. 58 min.	2,3	8,38	91%
PPO2 CnnLnLstmPolicy	10×10	2	69 val. 48 min.	4,7	-0,34	44%
PPO2 CnnLnLstmPolicy	10×10	$1 \rightarrow 2$	29 val. 27 min.	1,8	0,16	56%

Rezultatai

Šio darbo metu buvo pasiektas **tikslas** – palyginti skatinamojo mokymosi algoritmai, nustatyti efektyviausi algoritmai ir strategijos Sokoban žaidimui skirtingose situacijose.

Igyvendinti darbui iškelti **uždaviniai**:

1. Atlikta skatinamojo mokymosi algoritmų analizė, parinkta Stable Baselines RL biblioteka ir atrinkti trys potencialūs algoritmai Sokoban žaidimo agento mokymui: A2C, ACER ir PPO2.
2. Paruošta eksperimentinė Sokoban žaidimo aplinka. Gym-Sokoban OpenAI Gym aplinkai suprogramuoti pagalbiniai dekoratoriai. Mokymui paruoštos programos ir iškviečiamos funkcijos, mokymosi stebėjimui.
3. Atliktas eksperimentas, nustatyta geriausia, naudotos RL bibliotekos strategija Sokoban žaidimo agento valdymui.
4. Eksperimentiškai palygintos Stable Baselines RL algoritmų implementacijos su numatytais nustatymais: A2C, ACER ir PPO2. Palyginimui naudota mažiausia įmanoma Sokoban žaidimo aplinka: 7×7 kambario dydis su viena dėže per kambarį.
5. Panaudotas žinių perdavimas, apmokyti Sokoban žaidimo agentai veikti sudėtingesnėse aplinkose, t.y. 10×10 kambario dydžio su dviem dėžėmis. Dar sudėtingesnės aplinkos nebuvo bandomos, dėl labai ilgo apmokymo laiko.

Išvados

Šiame darbe atlikti eksperimentai leidžia daryti tokias išvadas:

1. Eksperimentiškai palyginus Stable Baselines RL bibliotekos strategijas nustatyta, kad `CnnLnLstmPolicy` strategija yra geriausia Sokoban žaidimo agento mokymui lyginant su `CnnPolicy` ir `CnnLstmPolicy` (paveikslėlis 16, lentelė 4). Antra pagal gerumą strategija yra `CnnPolicy`, ji pasiekia 15% mažesnę tikslumą, lyginant su `CnnLnLstmPolicy`, Sokoban žaidimo aplinkoje su viena dėže ir 7×7 kambario dydžiu.
2. Atlikus lyginamuosius eksperimentus, nustatyta, kad Sokoban žaidimo agentas, naudojantis ACER algoritmą (su numatytaisiais parametrais) yra geriausiai besimokantis mažiausioje Sokoban žaidimo (7×7 kambario dydžio su viena dėže) aplinkoje lyginant su A2C ir PPO2 algoritmais (paveikslėlis 17, lentelė 4), tačiau tai jis padaro per ilgiausią laiko tarpą.
3. Palyginus Sokoban žaidimo agentų mokymosi rezultatus naudojant ir nenaudojant mokymosi žinių perdavimą sudėtingesnėje (10×10 kambario dydžio su dviem dėžėmis) aplinkoje, nustatyta, kad agentas naudojantis mokymosi žinių perdavimą pasiekia geresnius rezultatus per trumpesnę laiką ir mažesnę žingsnių skaičių dviem iš trijų tirtų atvejų:
 - (a) Eksperimento metu gauti rezultatai rodo, kad naudojant ACER algoritmą (su numatytaisiais parametrais) sudėtingesnės (10×10 kambario dydžio su dviem dėžėmis) Sokoban žaidimo aplinkos mokymui, geri rezultatai per priimtina laiką negaunami, nepriklausomai ar mokymosi žinių perdavimas yra taikomas ar ne (paveikslėlis 20, lentelė 4).
 - (b) Mokinant PPO2 algoritmo (su numatytaisiais parametrais) agentą su mokymosi žinių perdavimu, gauti geresni rezultatai. Nors PPO2 agentas mokėsi sudėtingoje aplinkoje be žinių perdavimo (pasiekė 44% kambarių išsprendimo procentą per duotą 70 valandų mokymosi laiką), agentas su žinių perdavimu pasiekė aukštesnius rezultatus (56% kambarių išsprendimo procentą) per trumpesnę bendrą (pirminio agento ir perėmusio žinias agento) mokymosi laiką (53 valandas).
 - (c) A2C (su numatytaisiais parametrais) agentas su žinių perdavimu parodė geriausius rezultatus ir pasiekė 72% kambarių išsprendimo procentą (lentelė 4). A2C agentas be žinių perdavimo nesimokė ir per priimtina mokymosi laiką nepakilo aukščiau 1%.

Literatūra

- [BCP⁺16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang ir Wojciech Zaremba. Openai gym, 2016. eprint: arXiv:1606.01540.
- [Bel57] Richard Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957. ISSN: 00959057, 19435274. URL: <http://www.jstor.org/stable/24900506>.
- [BKH16] Jimmy Lei Ba, Jamie Ryan Kiros ir Geoffrey E. Hinton. Layer normalization, 2016. arXiv: 1607.06450 [stat.ML].
- [Cul97] Joseph Culberson. Sokoban is pspace-complete, 1997.
- [DHK⁺17] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol ir k.t. Openai baselines. <https://github.com/openai/baselines>, 2017.
- [Dyn20] Boston Dynamics. Boston dynamics. 2020. URL: <https://www.bostondynamics.com/about> (tikrinta 2020-03-19).
- [DZ99] Dorit Dor ir Uri Zwick. Sokoban and other motion planning problems. *Computational Geometry*, 13(4):215–228, 1999.
- [Fuk80] Kunihiro Fukushima. Neocognitron: a self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.
- [Gér17] A. Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O’Reilly Media, 2017. ISBN: 9781491962244. URL: <https://books.google.lt/books?id=bRpYDgAAQBAJ>.
- [HRE⁺18] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave ir k.t. Stable baselines. <https://github.com/hill-a/stable-baselines>, 2018.
- [HS97] Sepp Hochreiter ir Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. DOI: 10.1162/neco.1997.9.8.1735.
- [JS98] Andreas Junghanns ir Jonathan Schaeffer. Sokoban: evaluating standard single-agent search techniques in the presence of deadlock. *Conference of the Canadian Society for Computational Studies of Intelligence*, p. 1–15. Springer, 1998.

- [Ker] Kate Kershner. What are supercomputers currently used for? URL: <https://computer.howstuffworks.com/supercomputers-used-for1.htm> (tikrinta 2020-03-19).
- [KLM96] L. P. Kaelbling, M. L. Littman ir A. W. Moore. Reinforcement learning: a survey, 1996. URL: <https://doi.org/10.1613/jair.301>.
- [KSF17] Alexander Kuhnle, Michael Schaarschmidt ir Kai Fricke. Tensorforce: a tensorflow library for applied reinforcement learning. Web page, 2017. URL: <https://github.com/tensorforce/tensorforce>.
- [LBB⁺98] Yann LeCun, Léon Bottou, Yoshua Bengio ir Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- [MBM⁺16] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver ir Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016. arXiv: 1602.01783 [cs.LG].
- [Mye95] Andrew Myers. Xsokoban, 1995.
- [MKS⁺13a] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra ir Martin Riedmiller. Playing atari with deep reinforcement learning, 2013. arXiv: 1312.5602 [cs.LG].
- [MKS⁺13b] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra ir Martin Riedmiller. Playing atari with deep reinforcement learning, 2013. arXiv: 1312.5602 [cs.LG].
- [Moz17] Paul Mozur. Google’s alphago defeats chinese go master in win for a.i. 2017. URL: <https://www.nytimes.com/2017/05/23/business/google-deepmind-alphago-go-champion-defeat.html> (tikrinta 2020-03-19).
- [MP43] Warren S. McCulloch ir Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943. DOI: 10.1007/bf02478259.
- [oCom] History of Computers. History of computers. URL: <https://homepage.cs.uri.edu/faculty/wolfe/book/Readings/Reading03.htm> (tikrinta 2020-03-19).
- [Ope] OpenAI. Part 2: kinds of rl algorithms. URL: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html.

- [Pla16] Matthias Plappert. Keras-rl. <https://github.com/keras-rl/keras-rl>, 2016.
- [Pra18] Prabhu. Understanding of convolutional neural network (cnn) — deep learning, 2018. URL: <https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148>.
- [PW94] Jing Peng ir Ronald J. Williams. Incremental multi-step q-learning. *Machine Learning Proceedings 1994*:226–232, 1994. DOI: 10.1016/b978-1-55860-335-6.50035-0.
- [RHW85] David E Rumelhart, Geoffrey E Hinton ir Ronald J Williams. Learning internal representations by error propagation. Tech. atask., California Univ San Diego La Jolla Inst for Cognitive Science, 1985.
- [Ric98] Andrew G. Barto Richard S. Sutton. *Reinforcement Learning: An Introduction*. MIT Press, 1998.
- [Ros57] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.
- [RWR⁺17] Sébastien Racanière, Theophane Weber, David Reichert, Lars Buesing ir k.t. Imagination-augmented agents for deep reinforcement learning. I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan ir R. Garnett, redaktoriai, *Advances in Neural Information Processing Systems 30*, p. 5690–5701. Curran Associates, Inc., 2017. URL: <http://papers.nips.cc/paper/7152-imagination-augmented-agents-for-deep-reinforcement-learning.pdf>.
- [Saa17] Saama.com. Different kinds of convolutional filters, 2017. URL: <https://www.saama.com/different-kinds-convolutional-filters/>.
- [Sch05] Tom Schaul. Evolving a compact concept-based sokoban solver. *Master’s thesis, École Polytechnique Fédérale de Lausanne*, 2005.
- [Sch18] Max-Philipp B. Schrader. Gym-sokoban. <https://github.com/mpSchrader/gym-sokoban>, 2018.
- [Sch19] John Schulman. Proximal policy optimization, 2019-03. URL: <https://openai.com/blog/openai-baselines-ppo/>.

- [Ser18] Sergio Guadarrama, Anoop Korattikara, Oscar Ramirez, Pablo Castro, Ethan Holly, Sam Fishman, Ke Wang, Ekaterina Gonina, Neal Wu, Efi Kokiopoulou, Luciano Sbaiz, Jamie Smith, Gábor Bartók, Jesse Berent, Chris Harris, Vincent Vanhoucke, Eugene Brevdo. TF-Agents: a library for reinforcement learning in tensorflow. <https://github.com/tensorflow/agents>, 2018. URL: <https://github.com/tensorflow/agents>. [Online; accessed 25-June-2019].
- [Sim19] Thomas Simonini. Choosing a deep reinforcement learning library, 2019-06. URL: <https://medium.com/data-from-the-trenches/choosing-a-deep-reinforcement-learning-library-890fb0307092>.
- [SSB14] Hasim Sak, Andrew W Senior ir Franoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling, 2014.
- [SWD⁺17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford ir Oleg Klimov. Proximal policy optimization algorithms, 2017. arXiv: 1707.06347 [cs.LG].
- [Vos99] Michael D. Vose. *The Simple Genetic Algorithm Foundations and Theory*. MIT Press, 1999.
- [Wat89] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. *Robotics and Autonomous Systems*, 1989.
- [WBH⁺16] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu ir Nando de Freitas. Sample efficient actor-critic with experience replay, 2016. arXiv: 1611.01224 [cs.LG].
- [Wil92] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Reinforcement Learning*:5–32, 1992. DOI: 10.1007/978-1-4615-3618-5_2.
- [WML⁺17] Yuhuai Wu, Elman Mansimov, Shun Liao, Roger Grosse ir Jimmy Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation, 2017. arXiv: 1708.05144 [cs.LG].
- [Wu19] Yuhuai Wu. Openai baselines: acktr & a2c, 2019-03. URL: <https://openai.com/blog/baselines-acktr-a2c/>.

Santrumpos

Darbe naudojamų **santrumpų paaiškinimai**:

- **A2C** – (*angl. Advantage Actor-Critic*) pranašumo aktorius-kritiko algoritmas.
- **A3C** – (*angl. Asynchronous Advantage Actor-Critic*) asinchroninis pranašumo aktorius-kritiko algoritmas.
- **ACER** – (*angl. Actor-Critic with Experience Replay*) aktorius-kritiko su patirties pakartojimu algoritmas.
- **ANN** – (*angl. Artificial Neural Network*) dirbtinis neuroninis tinklas.
- **API** – (*angl. Application Programming Interface*) aplikacijų programavimo sąsaja.
- **CNN** – (*angl. Convolutional Neural Network*) konvoliucinis neuroninis tinklas.
- **CPU** – (*angl. Central Processing Unit*) centrinis procesorius.
- **DNN** – (*angl. Deep Neural Network*) gilusis neuroninis tinklas.
- **GPU** – (*angl. Graphics Processing Unit*) grafinis procesorius.
- **LSTM** – (*angl. Long Short-Term Memory*) ilgos trumpalaikės atminties modelis.
- **LTU** – (*angl. Linear Treshold Unit*) linijinis slenksčio vienetas.
- **MDP** – (*angl. Markov Decision Processes*) Markovo sprendimo priėmimo procesai.
- **ML** – (*angl. Machine Learning*) mašininis mokymasis.
- **MLP** – (*angl. Multi-Layer Perceptrons*) daugiasluoksnis perceptronas.
- **PG** – (*angl. Policy Gradient*) strategijos gradientas.
- **PPO** – (*angl. Proximal Policy Optimization*) proksimalinis strategijos optimizavimo algoritmas.
- **RGB** – (*angl. Red Green Blue colour model*) raudona, žalia, mėlyna spalvos.
- **RL** – (*angl. Reinforcement Learning*) skatinamasis mokymas.
- **RNN** – (*angl. Recurrent Neural Network*) rekurentinis neuroninis tinklas.
- **VSC** – (*angl. Version-Control System*) versijų tvarkymo sistema.

Priedas nr. 1

Stable Baselines skatinamojo mokymosi bibliotekos algoritmų A2C, ACER, PPO2 implementacijų numatytosios hiper-parametrų reikšmės

Bakalauro darbo metu yra naudojamos Stable Baselines skatinamojo mokymosi bibliotekos algoritmų implementacijos su jų autorių numatytosiomis hiper-parametrų reikšmėmis, kurios yra nurodytos lentelėje 5.

5 lentelė. A2C, ACER, PPO2 algoritmų implementacijų numatytosios hiper-parametrų reikšmės

Hiper-parametro pavadinimai	Algoritmai		
	A2C	ACER	PPO2
gamma	0,99	0,99	0,99
n_steps	5	20	128
vf_coef	0,25	–	0,5
q_coef	–	0,5	–
ent_coef	0,01	0,01	0,01
max_grad_norm	0,5	10	0,5
learning_rate	$7 \cdot 10^{-4}$	$7 \cdot 10^{-4}$	$2,5 \cdot 10^{-4}$
alpha	0,99	0,99	–
momentum	0,0	–	–
epsilon	10^{-5}	–	–
lr_schedule	–	'linear'	–
rprop_alpha	–	0,99	–
rprop_epsilon	–	10^{-5}	–
buffer_size	–	5000	–
replay_ratio	–	4	–
replay_start	–	1000	–
correction_term	–	10,0	–
lam	–	–	0,95
nminibatches	–	–	4
noptepochs	–	–	4
cliprange	–	–	0,2

Priedas nr. 2

Sokoban aplinkai pagal OpenAI Gym principus parašytas programinis kodas

Čia apibrėžti trys papildomi dekoratoriai paremti OpenAI Gym `Wrapper` aplinkos dekoratoriaus interfeisu (papunktis 2.1.1.2). `ActionWrapper` limituoja leidžiamų atlikti veiksmų sritį. `ExposeCompletedEnvironments` naudojamas su bakalauro darbo metu parašytais per-kvietimo funkcijomis. Šis dekoratorius, padaro sėkmingai užbaigtų aplinkų skaitinę reikšmę pasiekiamą iš išorės. `CombinedWrappers` pagalbinis dekoratorius, leidžiantis lengvai sukurti aplinkos objektą su keliais dekoratoriais iš karto.

```
1 import gym
2 import numpy as np
3 from gym.spaces.discrete import Discrete
4
5 class ActionWrapper(gym.Wrapper):
6     def __init__(self, env, new_action_space=None, **kwargs):
7         super(ActionWrapper, self).__init__(env)
8         if new_action_space is not None:
9             self.env.action_space = Discrete(new_action_space)
10
11 class ExposeCompletedEnvironments(gym.Wrapper):
12     def __init__(self, env, done_info_keyword=None, **kwargs):
13         super(ExposeCompletedEnvironments, self).__init__(env)
14         self.done_info_keyword = done_info_keyword
15         self.env_done = False
16         self.env_completed = False
17         self.cleaned = True
18         self.reset_totals()
19     def reset(self, **kwargs):
20         observation = super(ExposeCompletedEnvironments, self).reset(**kwargs)
21         self.cleaned = True
22         return observation
23     def step(self, action):
24         observation, reward, done, info = super(ExposeCompletedEnvironments, self).step(action)
25         if self.cleaned:
26             self._clean_env()
27         if done:
28             self.env_done = True
```

```

29         if self.done_info_keyword in info:
30             self.env_completed = info[self.done_info_keyword]
31         else:
32             self.env_completed = False;
33             info['puzzle_completed'] = self.env_completed
34             self.total_completed_num += int(self.env_completed)
35             self.total_done_num += 1
36         return observation, reward, done, info
37     def completed_ratio(self):
38         if self.total_done_num > 0:
39             return self.total_completed_num / self.total_done_num
40         return np.nan
41     def _clean_env(self):
42         self.env_done = False
43         self.env_completed = False
44         self.cleaned = False
45     def reset_totals(self):
46         self.total_completed_num = 0
47         self.total_done_num = 0
48     def __getattr__(self, name):
49         if name is "completed_ratio":
50             return self.completed_ratio
51         elif name is "reset_totals":
52             return self.reset_totals
53         elif name is "env_done":
54             return self.env_done
55         elif name is "env_completed":
56             return self.env_completed
57         attr = super(ExposeCompletedEnvironments, self).__getattr__(name)
58         if attr is not None:
59             return attr
60
61 class CombinedWrappers(gym.Wrapper):
62     def __init__(self, env, wrappers=[], **kwargs):
63         for wrapper in wrappers:
64             env = wrapper(env, **kwargs)
65         super(CombinedWrappers, self).__init__(env)

```

Priedas nr. 3

Modelio mokymo programinis kodas A2C, ACER ir PPO2 algoritams bei CnnPolicy, CnnLstmPolicy ir CnnLnLstmPolicy strategijoms

Originaliai, programa parašyta naudojantis Jupyter Notebook aplinka. Čia pateikiamas programos atitikmuo Python skripto failu. Programa leidžia nurodyti tiriamą algoritmą (punktas 2.3.3) bei strategiją (punktas 2.3.2). Nustatoma modelio aplinka: 7×7 dydžio kambarys, viena dėžė bei penki galimi veiksmai. Nurodytas modelis yra apmokomas 10^7 žingsnių ir išsaugomas į failą. Mokymosi procesas yra stebimas naudojantis TensorBoard, iš kurio yra gaunami duomenys analizei.

```
1 import warnings
2 warnings.filterwarnings('ignore')
3
4 import argparse, os, sys
5 import numpy as np
6 import gym, gym_sokoban
7 from gym_sokoban.envs import SokobanEnv
8 from stable_baselines import A2C, ACER, PPO2
9 from stable_baselines.common.cmd_util import make_vec_env
10 from stable_baselines.common.policies import CnnPolicy, CnnLstmPolicy, ↵
    CnnLnLstmPolicy
11 from my_wrappers import ActionWrapper
12
13 steps = int(1e7)
14 room_dimensions = (7, 7)
15 box_number = 1
16 max_steps_in_env = 200
17 n_envs = 64
18
19 def main(name, algorithm, policy):
20     save_file_name = f"{name}_sokoban_10M"
21     tensorboard = f"./{name}_sokoban_tensorboard"
22
23     sokoban_env = SokobanEnv(dim_room=room_dimensions, max_steps=↵
max_steps_in_env, num_boxes=box_number)
24     create_env_fn = lambda: ActionWrapper(sokoban_env, new_action_space=5)
25     env = make_vec_env(create_env_fn, n_envs=n_envs)
26
27     model = algorithm(policy, env, verbose=1, tensorboard_log=tensorboard)
28     print(f'Starting {name} model')
```

```

29
30     model.learn(total_timesteps=steps, reset_num_timesteps=False)
31     model.save(save_file_name)
32     print(f"Training done, model saved:", save_file_name)
33
34 if __name__ == "__main__":
35     parser = argparse.ArgumentParser(description="Training algorithms")
36     parser.add_argument('--algorithm', type=str, default=None)
37     parser.add_argument('--policy', type=str, default=None)
38     args = parser.parse_args()
39     algorithm = args.algorithm.upper()
40     policy = args.policy.upper()
41
42     if algorithm == 'A2C':
43         algorithm_fn = A2C
44     elif algorithm == 'ACER':
45         algorithm_fn = ACER
46     elif algorithm == 'PP02':
47         algorithm_fn = PP02
48     else:
49         algorithm_fn = None
50
51     if policy == 'CNN':
52         policy_fn = CnnPolicy
53     elif policy == 'CNNLSTM':
54         policy_fn = CnnLstmPolicy
55     elif policy == 'CNNLNLSTM':
56         policy_fn = CnnLnLstmPolicy
57     else:
58         policy_fn = None
59
60     if algorithm_fn is not None and policy_fn is not None:
61         main(algorithm, algorithm_fn, policy_fn)
62     else:
63         print("No or incorrect algorithm or policy name given.")

```

Priedas nr. 4

Modelio mokymo su žinių perdavimu programinis kodas

Python programa apmokanti nurodyto algoritmo (punktas 2.3.3) ir CnnLnLstmPolicy strategijos (papunktis 2.3.2.3) modelį su nurodytais pradiniais aplinkos sudėtingumo nustatymais: 10×10 dydžio kambarys, viena dėžė bei penki galimi veiksmi. Mokoma iki kol nurodyti mokymo kriterijai yra pasiekiami: mokymas trunka bent 5 iteracijas, paskutinės iteracijos išspręstų kambarių dalis yra bent 50%, paskutinių 5 iteracijų vidurkio ir paskutinės iteracijos išspręstų kambarių absoliutus skirtumas yra žemiau 1% arba paskutinės iteracijos išspręstų kambarių dalis yra daugiau nei 90%. Kai kriterijai pasiekiami, modeliui duodama sudėtingesnė aplinka ir mokymas yra kartojamas iki nurodyto sudėtingumo.

```
1 import warnings
2 warnings.filterwarnings('ignore')
3 from distutils.util import strtobool
4 import argparse, os, sys
5 import numpy as np
6 import tensorflow as tf
7 from typing import Optional
8 from collections import deque
9 import gym, gym_sokoban
10 from gym_sokoban.envs import SokobanEnv
11 from stable_baselines import A2C, ACER, PP02
12 from stable_baselines.common.vec_env import SubprocVecEnv, DummyVecEnv
13 from stable_baselines.common.cmd_util import make_vec_env
14 from stable_baselines.common.policies import CnnLnLstmPolicy
15 from stable_baselines.common.callbacks import BaseCallback, CallbackList, ←
    CheckpointCallback, EveryNTimesteps, EventCallback
16 from my_wrappers import ActionWrapper, ExposeCompletedEnvironments, ←
    CombinedWrappers
17 verbose=1
18 save_dir = "./saves/"
19 tensorboard_path = "./tensorboard"
20 max_amount_of_steps_per_iteration = int(1e8)
21 iteration_steps = int(1e6)
22 iteration_count_before_progressing = 5
23 min_iteration_solverate_to_progress = 0.5
24 min_abs_solverate_and_avg_difference_to_progress = .01
25 ignore_avg_treshold_to_progress = .9
26 room_dimensions = (10, 10)
27 max_steps_in_env = 100
28 initial_box_number = 1
```

```

29 max_box_number = 2
30 n_envs = 16
31 def create_current_env(num_boxes, vec_env_cls=SubprocVecEnv):
32     env_kwargs = {'dim_room':room_dimensions, 'max_steps':max_steps_in_env,↵
33                   'num_boxes':num_boxes}
34     wrapper = lambda env: CombinedWrappers(env, wrappers=[ActionWrapper, ↵
35         ExposeCompletedEnvironments], new_action_space=5, done_info_keyword=("↵
36         all_boxes_on_target"))
37     env = make_vec_env(SokobanEnv, n_envs=n_envs, env_kwargs=env_kwargs, ↵
38         wrapper_class=wrapper, vec_env_cls=vec_env_cls)
39     return env
40
41 def copy_and_replace_model(create_fn, model, env, tensorboard_full_path):
42     new_model = create_fn(CnnLstmPolicy, env=env, verbose=verbose, ↵
43         tensorboard_log=tensorboard_full_path)
44     print("New model created")
45     model_parameters = model.get_parameters()
46     new_model.load_parameters(model_parameters)
47     print("Parameters copied")
48     return new_model
49
50 class TensorboardCompletionTrackingCallback(BaseCallback):
51     def __init__(self, verbose=0):
52         super(TensorboardCompletionTrackingCallback, self).__init__(verbose)
53     def _on_step(self) -> bool:
54         dones = self.training_env.get_attr('env_done')
55         completes = self.training_env.get_attr('env_completed')
56         for done, complete in zip(dones, completes):
57             if done:
58                 writer = self.locals['writer']
59                 if writer is not None:
60                     value = int(complete)
61                     summary = tf.Summary(value=[tf.Summary.Value(tag='↵
62                     completions', simple_value=value)])
63                     self.locals['writer'].add_summary(summary, self.↵
64                     num_timesteps)
65         return True
66
67 class CompletionEvaluationCallback(BaseCallback):
68     def __init__(self, box_number = 1, verbose=1):
69         super(CompletionEvaluationCallback, self).__init__(verbose)
70         self.current_solverate_queue = deque(maxlen=↵
71         iteration_count_before_progressing)

```

```

61     self.current_local_iteration = 0
62     self.box_number = box_number
63     def _on_step(self) -> bool:
64         # Track current iteration
65         self.current_local_iteration += 1
66         # Get each environment complete ratio
67         env_ratios = [fn() for fn in self.training_env.get_attr('←
completed_ratio')]
68         env_ratios_cleaned = [0 if np.isnan(x) else x for x in env_ratios]
69         # Reset totals for next iteration
70         for reset_totals_fn in self.training_env.get_attr('reset_totals'):
71             reset_totals_fn()
72         # Calculate total solverate from all of the environements
73         solverate = np.mean(env_ratios_cleaned)
74         self.current_solverate_queue.append(solverate)
75         if self.verbose > 0:
76             print(f"Model's current solverate is '{solverate}'.")
77         # Progress if minimum amount of iterations have passed
78         if self.current_local_iteration < iteration_count_before_progressing←
:
79             if self.verbose > 0:
80                 print(f"Current iteration number {self.←
current_local_iteration} is below minimum before progressing iteration ←
number {iteration_count_before_progressing}.")
81                 return True
82             if self.verbose > 0:
83                 print(f"Current iteration number {self.current_local_iteration} ←
is sufficient to progress.")
84             # Pass if solverate is above the minimum expected solvrte for ←
progression
85             if solverate < min_iteration_solverate_to_progress:
86                 if self.verbose > 0:
87                     print(f"Current iteration solverate {solverate} is not ←
sufficient to progress. Min required: {←
min_iteration_solverate_to_progress}.")
88                     return True
89                 if self.verbose > 0:
90                     print(f"Current iteration solverate {solverate} is sufficient to←
progress.")
91         avg_solverate = np.mean(self.current_solverate_queue)

```



```

92     abs_diff = abs(solverate - avg_solverate)
93     # Pass for progress with difficulty if abs diff of average over last
few and current is less then expected
94     # Ignore above condition if solverate is above the ignore treshold,
ignore this if this is the last run
95     if (abs_diff > min_abs_solverate_and_avg_difference_to_progress and
96         (self.box_number >= max_box_number or
97          solverate < ignore_avg_treshold_to_progress)):
98         if self.verbose > 0:
99             print(f"Current iteration solverate {solverate} and last {
iteration_count_before_progressing} iteration average solvareate {
avg_solverate} absolute differacnce {abs_diff} is not sufficient to
progress.")
100         return True
101     if self.verbose > 0:
102         print(f"Current iteration solverate {solverate} and last {
iteration_count_before_progressing} iteration average solvareate {
avg_solverate} absolute differacnce {abs_diff} is sufficient to progress.
")
103     self.current_local_iteration = 0
104     return False
105 def main(name, create_fn, load_fn=None, load_filename = None, starting_block
= None, copy=False, save_path=None):
106     # Create initial environment
107     current_box_number = initial_box_number if starting_block is None else
starting_block
108     env = create_current_env(current_box_number)
109     # Create model
110     print(f"Creating {name} model")
111     tensorboard_full_path = os.path.join(tensorboard_path, name)
112     if load_filename is None:
113         model = create_fn(CnnLnLstmPolicy, env=env, verbose=verbose,
tensorboard_log=tensorboard_full_path)
114         print(f"{name} model created")
115     else:
116         load_path = os.path.join(save_dir, name, load_filename)
117         model = load_fn(load_path, env=env, verbose=verbose, tensorboard_log
=tensorboard_full_path)
118         print(f"{name} model loaded")
119     if copy:

```

```

120         model = copy_and_replace_model(create_fn, model, env, ↵
tensorboard_full_path)
121     # Save path
122     full_save_path = os.path.join(save_dir, name)
123     if save_path is not None:
124         full_save_path = os.path.join(full_save_path, save_path)
125     # Create callbacks
126     tensorboard_callback = TensorboardCompletionTrackingCallback()
127     checkpoint_callback = CheckpointCallback(save_freq=1, save_path=↵
full_save_path)
128     iteration_test_callback = CompletionEvaluationCallback(box_number=↵
current_box_number)
129     iteration_callback_list = CallbackList([checkpoint_callback, ↵
iteration_test_callback])
130     iteration_callback = EveryNTimesteps(n_steps=iteration_steps, callback=↵
iteration_callback_list)
131     callbacks = CallbackList([tensorboard_callback, iteration_callback])
132     # Start Curriculum learning
133     while True:
134         if verbose > 0:
135             print(f"Starting training with {current_box_number} boxes")
136             # Create unique name for each iteration and different box numbers
137             log_name = f"{name}_box_{current_box_number}_v2"
138             # Train model for this iteration
139             model.learn(total_timesteps=max_amount_of_steps_per_iteration, ↵
tb_log_name=log_name, reset_num_timesteps=False, callback=callbacks)
140             # Save final run of the current difficulty
141             save_name = f'{name}_box_{current_box_number}_final'
142             final_save_path = os.path.join(save_dir, name)
143             if save_path is not None:
144                 final_save_path = os.path.join(final_save_path, save_path)
145                 final_save_path = os.path.join(final_save_path, save_name)
146                 model.save(final_save_path)
147                 current_box_number += 1
148             if current_box_number <= max_box_number:
149                 if verbose > 0:
150                     print(f"Increasing difficutly to {current_box_number} boxes"↵
)
151
152             # Close previous environment to save resources
153             model.get_env().close()

```

```

153         new_env = create_current_env(current_box_number)
154         if copy:
155             model = copy_model(create_fn, model, new_env, ↵
tensorboard_full_path)
156         else:
157             model.set_env(new_env)
158     else:
159         break
160 if __name__ == "__main__":
161     parser = argparse.ArgumentParser(description="Training algorithms")
162     parser.add_argument('--algorithm', type=str, default=None)
163     parser.add_argument('--load', type=str, default=None)
164     parser.add_argument('--starting', type=int, default=None)
165     parser.add_argument('--copy', type=lambda b: bool(strtobool(b)), default=↵
=False)
166     parser.add_argument('--save', type=str, default=None)
167     args = parser.parse_args()
168     algorithm = args.algorithm.upper()
169     load_filename = args.load
170     starting_block = None if args.starting is not None and args.starting > ↵
max_box_number else args.starting
171     copy = args.copy
172     save_path = args.save
173     if algorithm == 'A2C':
174         create_fn = A2C
175         load_fn = A2C.load
176     elif algorithm == 'ACER':
177         create_fn = ACER
178         load_fn = ACER.load
179     elif algorithm == 'PP02':
180         create_fn = PP02
181         load_fn = PP02.load
182     else:
183         create_fn = None
184         load_fn = None
185     if create_fn is not None:
186         main(algorithm, create_fn, load_fn, load_filename, starting_block, ↵
copy, save_path)
187     else:
188         print("No or incorrect algorithm name given.")

```