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

TODO: Santrauka

Raktiniai žodžiai: Skatinamasis mokymas, Sokoban žaidimas, aktorius-kritikas based metodai, raktinis žodis 4, raktinis žodis 5

Summary

TODO: summary

Keywords: Reinforcement learning, Sokoban game, actor-critic based methods, keyword 4, keyword 5

TURINYS

Įvadas

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 sprogimus, 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 šaka yra skatinamasis mokymas – agento atliekami veiksmai yra reguliariai vertinami ir atitinkamai agentas yra apdovanojamas arba baudžiamas.

..Kažką apie sokoban...

Darbo tikslas

Šio darbo **tikslas** – išanalizavus populiariausius skatinamojo mokymosi algoritmus, pritaikyti kelis labiausiai tinkamus Sokoban žaidimo aplinkai.....

Darbo uždaviniai

Darbui iškelti **uždainiai**:

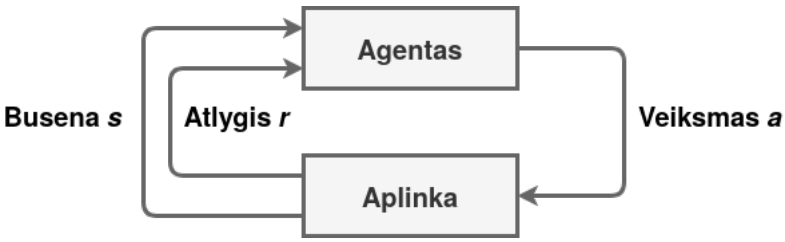
1. Paruošti eksperimentinę aplinką ir agentą.
2.

1. Teorija

Šiame skyriuje aprašyta teorinė bakalauro darbo dalis.

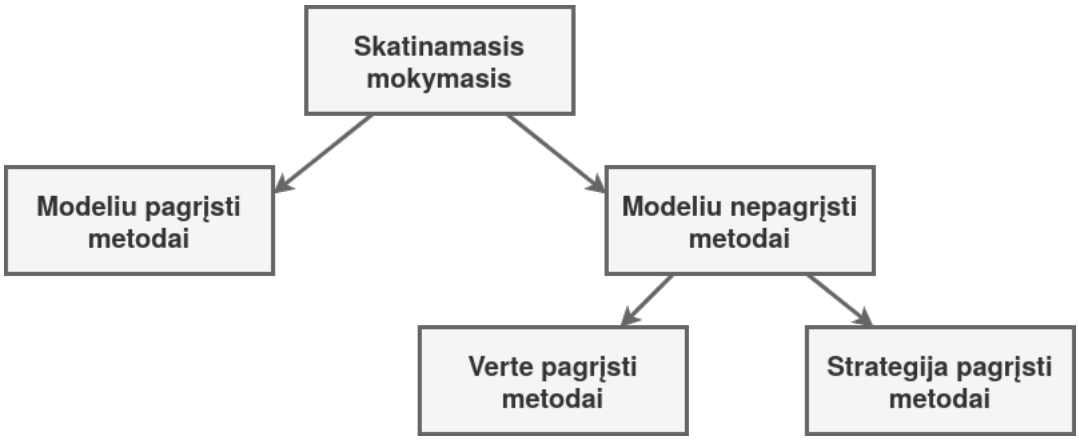
1.1. Skatinamasis mokymasis

Skatinamasis mokymasis (*angl. reinforcement learning*) (RL), kartu su prižiūrimuoju ir neprižiūrimuoju mokymu, yra viena iš pagrindinių mašininio mokymosi (ML) paradigmų. Klasikinis RL modelis susideda iš agento (*angl. agent*), kuris priima sprendimus ir atlieka veiksmus (*angl. actions*) pagal strategiją (*angl. policy*), paremtus aplinkos būsena (*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 (*ang. 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 (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:

1. **Pagrįsti modeliu** – (*angl. model-based*) tai yra algoritmai, kurie optimalios strategijos apskaičiavimui naudojasi transakcijų funkcija (ir atlygio funkcija) (daugiau punkte 1.1.2). Pagrįsti modeliu algoritmai gali nuspėti galimus aplinkos pakitimus, kadangi naudojasi apskaičiuota transakcijų funkcija. Tačiau, modelio turima funkcija gali būti tik apytikslė „tikrajai“ funkcijai, tad modelis gali niekada nepasiekti optimalaus sprendimo.
2. **Nepagrįsti modeliu** – (*angl. model-free*) tai yra algoritmai, kurie apskaičiuodami optimalią strategija 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 π yra sužymėjimas kokį 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 pasiekiama – 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 metoduose, veiksmo vertės funkcija yra reprezentuojama funkcijos apytikslinimu (*angl. approximtator*), pavyzdžiui, neuroniniu tinklu (daugiau poskyryje 1.2). Jei imame $Q(s, a; \theta)$ kaip apytikslė 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 veiksmo vertės funkcijos $Q(s, a; \theta)$ parametrai θ yra išmokstami iteratyviai mažinant nuostolių (*angl. loss*) funkcijos seką, kur kiekviena i-toji nuostolių funkcija yra funkcija 1 ir s' yra būsena pasiekta po būsenos s.

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

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: $r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \max_a \gamma^n Q(s_{t+n}, a)$. Taip pasiekiama, kad vienas atlygis r tiesiogiai paveikia n ankstesnių būsenų ir veiksmų porų reikšmes ir gaunamas potencialiai daug efektyvesnis aktualioms būsenos-veiksmo poroms atlygio skleidimo procesas.

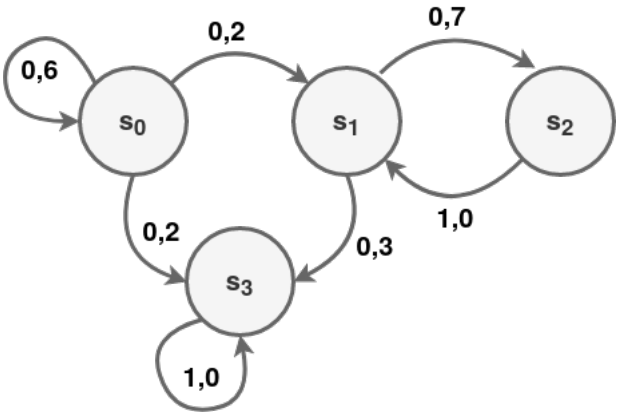
Atvirkščiai nei verte pagrįsti metodai, strategija pagrįsti ir modeliu nepagrįsti (poskyris 1.1) RL metodai tiesiogiai parametrizuoja strategiją $\pi(a|s;\theta)$ ir atnaujina parametrus θ atlikdami apytikslį gradiento pakėlimą (*angl. gradient ascent*) $\mathbb{E}[R_t]$. Vienas tokio metodo pavyzdys yra REINFORCE šeimos algoritmai [Wil92]. Standartinis REINFORCE atnaujina 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 išmoktą 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*).

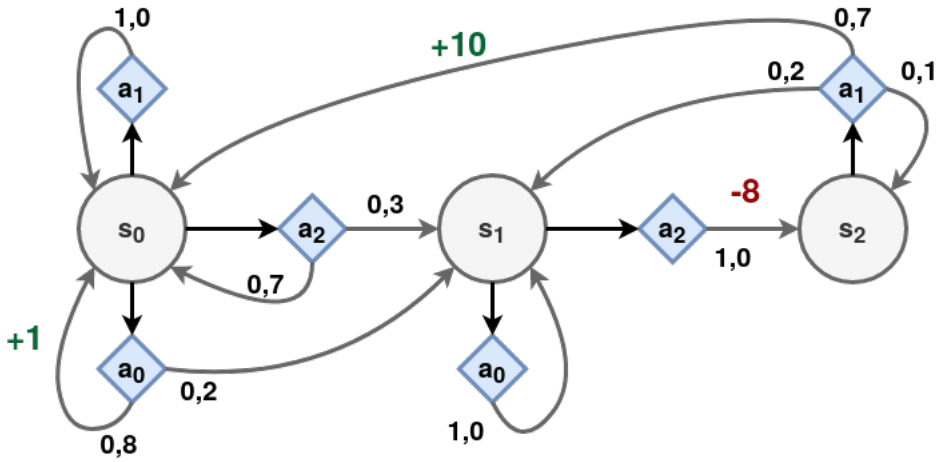
Markovo grandinė – tai kiekviename žingsnyje atsitiktinai judantis iš vienos į kitą būseną (*angl. state*) procesas su fiksuotu skaičiumi būsenų, kur tikimybė pereiti iš būsenos s į būseną s' yra taip pat fiksuota ir priklauso tik nuo poros (s, s'), ne nuo jau praėjusių būsenų. Markovo grandinės neturi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



3 pav. Markovo grandinės pavyzdys

Paveikslėlyje 3 pavaizduotas Markovo grandinės pavyzdys su keturiomis būsenomis. Jeigu laikome būseną s_0 pradine, tai yra 60% 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 strategija, kuri privestų prie didžiausio bendro atlygio per trumpiausia 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*. 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, plius visų įmanomų toliau einančių būsenų tikėtina optimali vertė.

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

- **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.
- $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0,1]$: transakcijų tikimybės pasiskirstymas $P(s'|s,a)$.
- $R : \mathcal{S} \to \mathbb{R}$: 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 įma-

nomų 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 dauguma įmanomų variantų ir pasirinkti geriausią iš jų. Tačiau, paieškos plotui didėjant, gero strategijos parametrų rinkinio paieška sudėtingėja ir reikalingų resursų skaičius eksponentiškai kylą, todėl šitas sprendimas dažniausiai nėra taikomas.

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

Dar vienas sprendimas būtu naudoti optimizavimo metodikas. 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

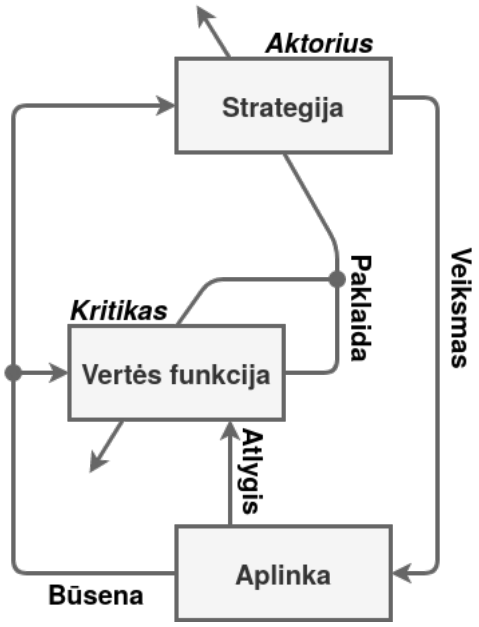
Strategijos gradientai (*angl. policy gradients*) (PG) yra atsakingi už strategijos parametrų optimizavimą sekant aukštesnio atlygio link. REINFORCE algoritmas [Wil92] yra populiari PG klasė. Toliau aprašoma dažna variacija:

1. Neuroninio tinklo strategija sužaidžia žaidimą kelis kartus, apskaičiuoja kiekvieno žingsnio gradientus, kurie pakeltų pasirinkto veiksmo pasirinkimo tikimybę ateityje, bet dar jų nepritaiko.
2. Po kelių sužaistų epizodų, apskaičiuojami taškai kiekvienam veiksmui.
3. Jei pasirinkto veiksmo 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 verte 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 (verte 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 variacijo pristatytos dar 1943 metais [MP43]. Per tiek metu 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 TFLOPS[2] pajėgumą, kai šiuolaikinės vartotojams prieinamos GPU yra vertinamos

[1] `https://www.top500.org/resources/top-systems/asci-white-lawrence-livermore-national-laboratory/`

[2] TFLOPS arba teraFLOPS yra 10^{12} FLOPS. FLOPS (*angl. floating point operations per second*) yra operacijų atliekamų per sekundę su slankaus kablelio skaičiais matmuo dažnai naudojamas kompiuterinių įrenginių pajėgumui

net iki 14[3] TFLOPS (ir nesveria šimtų tonų).

- Dėl dažno ANN pasiekiamų rezultatų atsiradimo žinių akiratyje, kas lemia didėjantį ir taip populiarų ANN technologijų finansavimą, greitėjantį progresą bei šią technologiją naudojančių naujų produktų atsiradimą[Gér17].

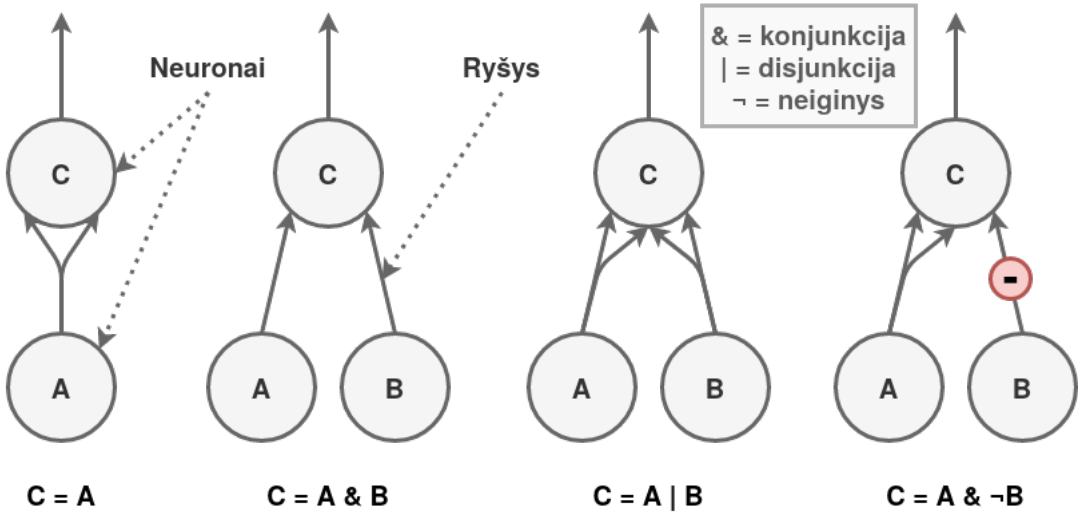
Toliau šiame skyriuje bus rašoma apie ANN ir jų variacijas bei susijusias technologijas.

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 medelis yra reikalingas sudėtingėjant užduotims.



6 pav. Dirbtinio neurono atliekančio loginius skaičiavimus pavyzdys

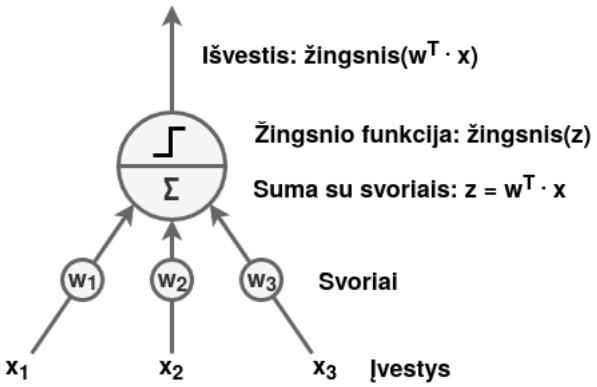
1.2.1.2. Linijinis slenksčio vienetas

Linijinis slenksčio vienetas (*angl. linear treshold unit*) (LTU) yra paremtas kiek pakeistu dirbtiniu neuronu (papunktis 1.2.1.1). LTU įvestys ir išvestys yra skaičiai (ne binarinės reikšmės)

nustatyti.

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

ir kiekviena įvestis turi savo svorį. LTU suskaičiuoja sumą su įvesčių svoriais ($z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = \mathbf{w}^T \cdot \mathbf{x}$), tada suskaičiuotai sumai pritaiko žingsnio funkcija (*angl. step function*) ir išveda rezultatą: $h_{\mathbf{w}}(\mathbf{x}) = \text{žingsnis}(z) = \text{žingsnis}(\mathbf{w}^T \cdot \mathbf{x})$ (paveikslėlis 7).

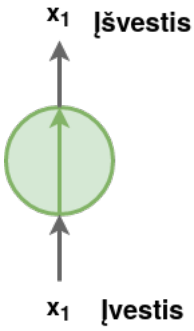


7 pav. Linijinis slenksčio vienetas

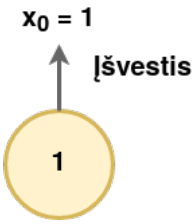
1.2.1.3. Pagalbiniai neuronai

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).



(a) Įvesties neuronas



(b) Postūmio neuronas

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.

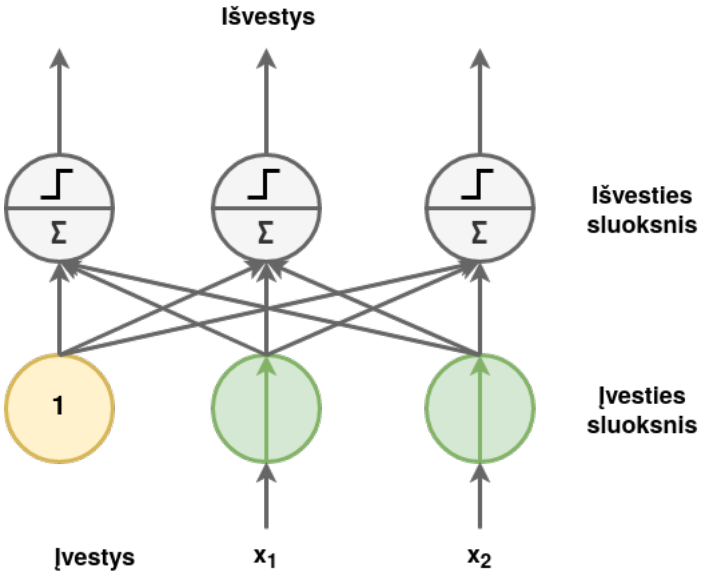
1.2.2. Parceptronas

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 ženklo (*angl. sign*) funkcija 4.

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

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

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).



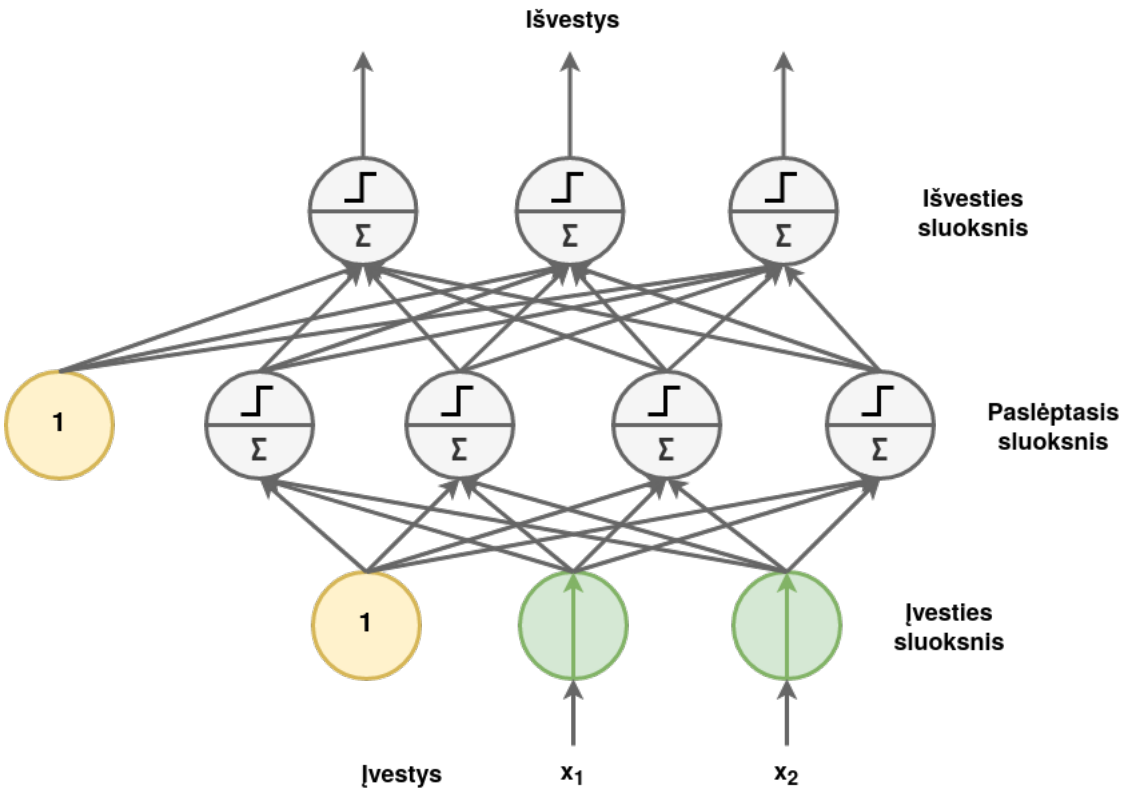
9 pav. Perceptrono pavyzdys

Perceptronai yra apmokomi naudojantis Hebo[4] 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 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 paslėptojo sluoksnio MLP.

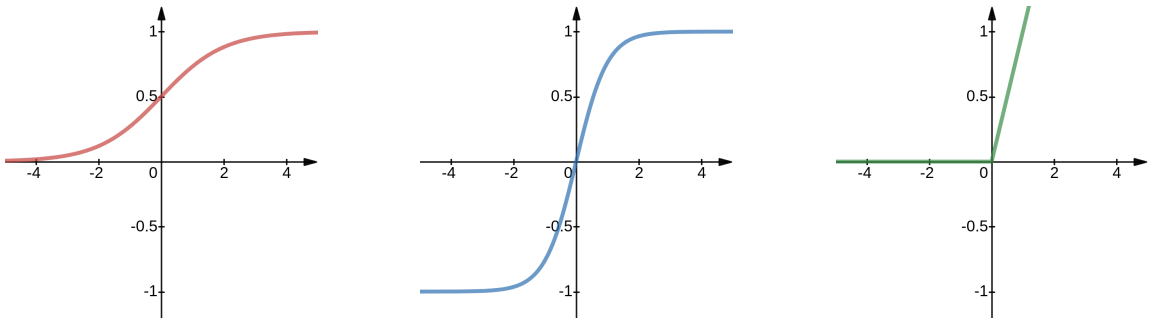
[4](*angl. Hebb's rule*) „Cells that fire together, wire together" svoriai tarp neuronų yra didinami, jei jie turi vienodą išvestį.



10 pav. Daugiasluoksnio perceptrono pavyzdys

MLP mokymui yra naudojamas atgalinio sklidimo (*angl. backpropogation*) 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).
- **Hiperbolinė 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 konvergencija (paveikslėlis 11b).
- **ReLU funkcija** $\mathrm{ReLU}(z) = \max(0, z)$, 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

Konvoluciniai 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 neokognitronas[6] [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.

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

[6] Neokognitronas – 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 išplatinęs gilųjį mokymąsį, plačiai naudojamas ant čekių ranka urašytų skaičių atpažinimui.

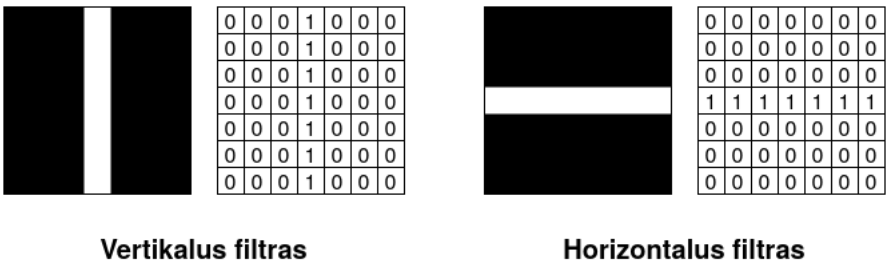
1.2.4.1. Konvoliuciniai sluoksniai

Konvoliuciniai[8] 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 iškarto keli skirtingi filtrai, taip vienu metu aptinkami keli skirtingi požymiai. Kai yra naudojamas vienas

[8]Konvoliucija 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.

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

- $z_{i,j,k}$ 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).

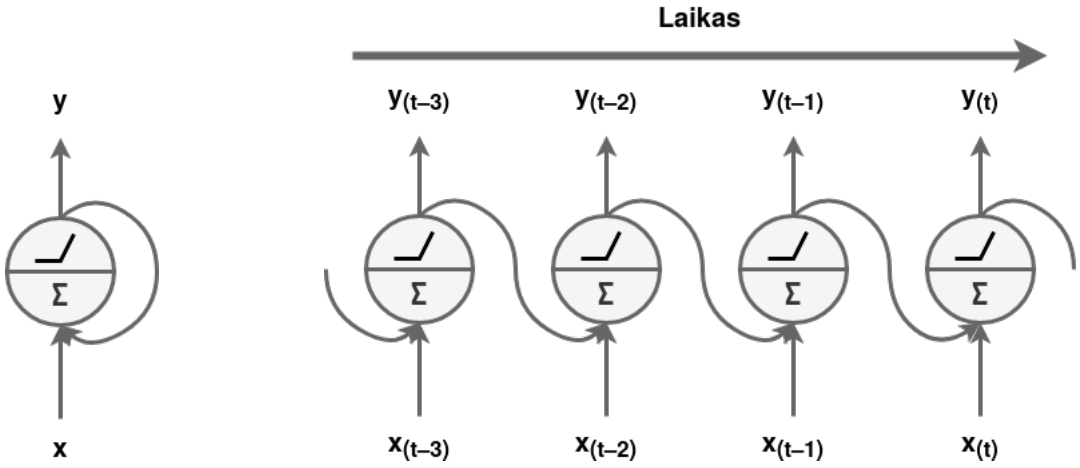
Sluoksnyje esančių neuronų įvestys yra sujungtos tik su matymo laukui priklausančia dalimi praeito sluoksnio išvesčių. Matymo laukas yra apibrėžiamas tais pačiais parametrais: dydžiu, žingsniu, užpildymo būdu. Sutelkimo, skirtingai nei konvoliucinio, sluoksnio neuronai neturi svorių. Vienintelė jų paskirtis 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. Pasikartojantys neuroniniai tinklai

Pasikartojantys 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 išvynioto (*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 $y_{(t-1)}$.



(a) Pasikartojantis neuronas (b) Laike išvyniotas pasikartojantis neuronas

13 pav. Pasikartojančio neurono pavyzdžiai

Sluoksnio 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 sluoksnio išvestį vienai mini-partijai[9] (*angl. mini-batch*).

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

- $\mathbf{Y}_{(t)}$ 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 neurono 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.

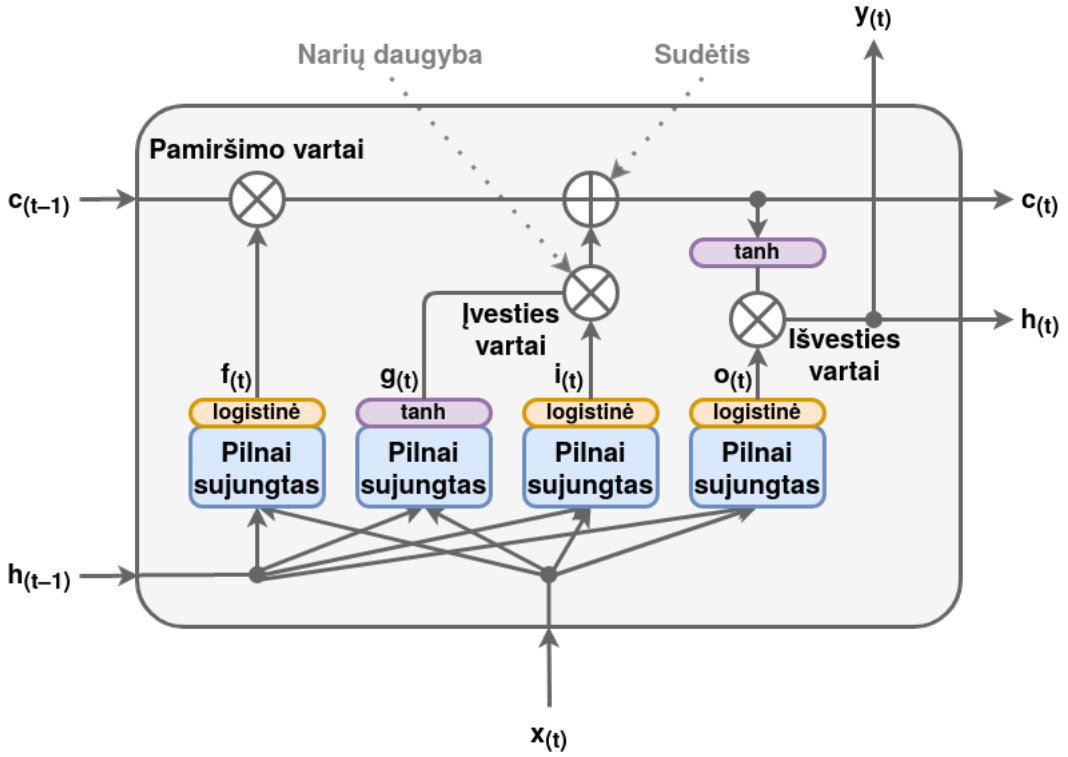
Pasikartojančio neurono išvestis galima išreikšti formule nuo laiko žingsnių t, kuri yra priklausoma nuo visų iki tol buvusių įvesčių ir tai galima vadinti atmintimi. Neuroninio tinklo dalis sauganti tam tikrą būseną laike yra vadinama atminties ląstele (*angl. memory cell*). Paveikslėlyje 13 pavaizduota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

1.2.5.2. Ilgos trumpalaikės atminties modelis

Ilgos trumpalaikės atminties modelio (*angl. long short-temr 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: $\mathbf{h}_{(t)}$ trumpalaikei būsenai ir $\mathbf{c}_{(t)}$ – ilgalaikei. Pagrindinė modelio idėja yra galimybė išmokti 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.

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



14 pav. Konvoliucinių filtrų pavyzdžiai

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

- Pagrindinis sluoksnis yr su išvesti $\mathbf{g}_{(t)}$. 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 daugyba [10]. Taip gauti 0 „uždaro“ vartus ir 1 „atidaro“.
  - **Pamiršimo vartai** – valdomi vektoriau $\mathbf{f}_{(t)}$ nustato kokia informacija turėtų būti pašalinama iš ilgalaikės būsenos.
  - **Įvesties vartai** – valdomi vektoriau $\mathbf{i}_{(t)}$ nustato kokia dalis vektoriaus $\mathbf{g}_{(t)}$ turėtų būti pridedama į ilgalaikę būseną.
  - **Išvesties vartai** – valdomi vektoriau $\mathbf{o}_{(t)}$ nustato kokia dalis ilgalaikės būsenos turėtų būti išvesta šį laiko žingsnį į vektorius $\mathbf{y}_{(t)}$ ir $\mathbf{h}_{(t)}$.

[10] Apskaičiuojamas „Hadamardo produktas“ (*angl. Hadamard product*). Hadamardo produktas tai yra matrica gaunama tarpusavyje sudauginus visus dviejų vienodo dydžio matricų 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 aprašyta..... KOL KAS NIEKAS

2.1. Sokoban žaidimas

Sokoban (iš japonų kalbos išvertus „sandėlio prižiūrėtojas“) yra 1981 metais Hiroyuki Imabayashi sukurtas tradicinis galvosūkis. Tai yra sudėtingas vieno žaidėjo žaidimas, kuriame tikslas yra vaikštant 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:

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

Sokoban neturi bendrojo sprendinio ar sprendimo būdo. Taip pat yra įrodyta kad Sokoban yra NP-Hard[11] [DZ99] ir PSPACE-complete[12] [Cul97] uždavinys.

2.1.1. OpenAI Gym

2.2. Skatinamojo mokymosi bibliotekos parinkimas

2.2.1. Stable Baselines architektūra

2.2.1.1. A2C aprašymas

2.2.1.2. ACER aprašymas

2.2.1.3. POP2 aprašymas

[11] NP-hard yra problemos, kurios yra tokio pat ar didesnio sudėtingumo, nei sudėtingiausios žinomos NP problemos. NP problemos yra klasė skaičiavimo problemų, kurios yra išsprendžiamos nedetriministiniu būdu polinominiame laike.

[12] PSPACE-complete yra problemos, kurios gali būti išsprestos naudojantis polinominiu kiekiu atminties.

3. Eksperimentai

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

3.1. Sokoban aplinkos paruošimas

Šiame poskyryje yra aprašomas metodas eksperimento metu tiriamos Sokoban aplinkos paruošimui.

3.1.1.

3.2. Eksperimentinė aplinka

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

3.2.1. Eksperimentinės aplinkos specifikacijos

Eksperimentas atliekamas naudojantis realią „Ubuntu“ mašiną.

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“ atminties[13].
   (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**).

[13] „Swap“ atmintis – tai pastoviojoje atmintyje paskirta atminties dalis virtualiai operatyviajai atminčiai, kuri yra naudojama kai fizinės operatyviosio atminties neužtenka vykdomoms operacijoms.

(e) ML atviro kodo platforma su lanksčia įrankių ir bibliotekų ekosistema skirta kurti ir gerinti šiuolaikinius ML sprendimus – „**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**).

(h) Modernių RL algoritmų implementacijų rinkinys (*angl. state-of-art*) – „**Stable Baselines**“ (versija: **2.10.1a0**).

(i) Išskirstyta VSC sistema pakeitimų sekimui kode – „**Git**“ (versija: **2.23.0**)

3.2.2. Ekseperimentinės aplinkos paruošimas

3.3. Eksperimento planas

Darbo metu atliktas eksperimentas susideda iš trijų dalių. Šiame skyriuje yra aprašomi šių trijų eksperimentų planai: kaip bus atliekamas eksperimentas, kokia bus naudojama aplinka, kokių rezultatų yra tikimasi ir pan.

3.3.1. Pirmo eksperimento planas: Geriausios strategijos ieškojimas

3.4. Eksperimentas

Literatūra

[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.

[Cul97] Joseph Culberson. Sokoban is pspace-complete, 1997.

[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] Kunihiko 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.

[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.

[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.

[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.

[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).

[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.

[Saa17] Saama.com. Different kinds of convolutional filters, 2017. URL: https://www.saama.com/different-kinds-convolutional-filters/.

[SSB14] Hasim Sak, Andrew W Senior ir Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling, 2014.

[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.

[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.

Santrumpos

Darbe naudojamų **santrumpų paaiškinimai**:

- **A2C** – (*angl. Advantage Actor-Critic*) Pranašumo Aktoriaus-Kritiko algoritmas.
- **A3C** – (*angl. Asynchronous Advantage Actor-Critic*) Asinchroninis Pranašumo Aktoriaus-Kritiko algoritmas.
- **ACER** – (*angl. Actor-Critic with Experience Replay*) Aktorius-Kritiko su patirties pakartojimu algoritmas.
- **ANN** – (*angl. Artificial Neural Network*) dirbtinis neuroninis tinklas.
- **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.
- **PPO2** – (*angl. Proximal Policy Optimization*) Proksimalinis Strategijos Optimizavimo algoritmas.
- **RL** – (*angl. Reinforcement Learning*) skatinamasis mokymas.
- **RNN** – (*angl. Recurrent Neural Network*) pasikartojantis neuroninis tinklas.
- **VSC** – (*angl. Version-Control System*) versijų tvarkymo sistema.

4. Sokoban aplinkai parašyti OpenAI Gym principus sekantys kodas

```
import gym
import numpy as np

class ActionWrapper(gym.Wrapper):
    def __init__(self, env, new_action_space=None, **kvars):
        super(ActionWrapper, self).__init__(env)
        from gym.spaces.discrete import Discrete
        if new_action_space is not None:
            self.env.action_space = Discrete(new_action_space)

class ExposeCompletedEnvironments(gym.Wrapper):
    def __init__(self, env, done_info_keyword=None, **kvars):
        super(ExposeCompletedEnvironments, self).__init__(env)
        self.done_info_keyword = done_info_keyword
        self.env_done = False
        self.env_completed = False
        self.cleaned = True

        self.reset_totals()

    def reset(self, **kwargs):
        observation = super(ExposeCompletedEnvironments, self).reset(**↩
    kwargs)
        self.cleaned = True
        return observation

    def step(self, action):
        observation, reward, done, info = super(ExposeCompletedEnvironments,↩
     self).step(action)
        if self.cleaned:
            self._clean_env()
        if done:
            self.env_done = True
            if self.done_info_keyword in info:
                self.env_completed = info[self.done_info_keyword]
            else:
                self.env_completed = False;
```

```
            info['puzzle_completed'] = self.env_completed

            self.total_completed_num += int(self.env_completed)
            self.total_done_num += 1
        return observation, reward, done, info

    def completed_ratio(self):
        if self.total_done_num > 0:
            return self.total_completed_num / self.total_done_num
        return np.nan

    def _clean_env(self):
        self.env_done = False
        self.env_completed = False
        self.cleaned = False

    def reset_totals(self):
        self.total_completed_num = 0
        self.total_done_num = 0

    def __getattr__(self, name):
        if name is "completed_ratio":
            return self.completed_ratio
        elif name is "reset_totals":
            return self.reset_totals
        elif name is "env_done":
            return self.env_done
        elif name is "env_completed":
            return self.env_completed
        attr = super(ExposeCompletedEnvironments, self).__getattr__(name)
        if attr is not None:
            return attr

class CombinedWrappers(gym.Wrapper):
    def __init__(self, env, wrappers=[], **kvars):
        for wrapper in wrappers:
            env = wrapper(env, **kvars)
        super(CombinedWrappers, self).__init__(env)
```