Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Piotr Kucharski

Nr albumu: 1124564

Uczenie agentów sterowania pojazdami kosmicznymi

Praca licencjacka
na kierunku Informatyka Stosowana

Praca wykonana pod kierunkiem
prof. dr hab. Piotr Białas
Zakład Technologii Gier

Kraków 2020

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

.....	.....
Kraków, dnia	Podpis autora pracy

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

.....	.....
Kraków, dnia	Podpis kierującego pracą

Spis treści

1 Abstrakt

Niniejsza praca prezentuje proces optymalizacji parametrów sztucznej sieci neuronowej za pomocą uczenia ze wzmocnieniem. Zadaniem sieci neuronowej jest optymalne sterowanie pojazdami w środowisku symulowanej przestrzeni kosmicznej. TODO

2 Wstęp

Dziedziny głębokiego uczenia maszynowego, takie jak detekcja obiektów na zdjęciach czy rozpoznawanie mowy, są już z powodzeniem wykorzystywane w komercyjnych produktach. Uczenie ze wzmocnieniem, które polega na optymalizacji strategii podejmowania decyzji na podstawie obserwacji, odniosło już wiele spektakularnych sukcesów w odkrywaniu wydajnych strategii w symulacjach i grach komputerowych. Algorytm Atari DQN[7], zaprezentowany w 2014 roku, był w stanie znaleźć wydajną strategię w siedmiu grach na platformie Atari. Architektura sztucznej sieci neuronowej zawierała splot funkcji co pozwoliło na wykorzystanie obrazu z gry w postaci tablicy pikseli jako obserwacji. Takie podejście zwiększyło możliwości generalizacji algorytmu. Nauka nowej gry nie wymagała modyfikacji w algorytmie. W roku 2017 algorytm AlphaGo[10] zwyciężył z mistrzem świata w grę GO, która jest uważana za jedną z najtrudniejszych gier planszowych. Ze względu na zbyt dużą liczbę możliwych sekwencji ruchów, wynoszącą w przybliżeniu 250^{150}, przeszukiwanie całego drzewa możliwych ruchów jest niewydajne. Problem ten został rozwiązany ograniczając głębokość przeszukiwania zastępując poddrzewo funkcją wartości $v(s)$ aproksymującą oczekiwany wynik z danego stanu s oraz ograniczając szerokość drzewa przeszukując tylko te najbardziej prawdopodobne ułożenia kamieni które były wybrane na podstawie strategii $p(a|s)$. Funkcja wartości oraz strategia były na początku uczone w oparciu o mecze rozegrane pomiędzy ludźmi, potem w oparciu o self-play. Zaledwie rok później ten sam zespół opublikował algorytm AlphaZero[11] będący w stanie nauczyć się optymalnej strategii w szachach, shogi oraz Go pokonując dotychczasowych mistrzów świata. Algorytm był uczony bez wykorzystania eksperckiej wiedzy. W 2019 zwycięstwa w dwóch następnych grach, pokonując mistrzów świata, odniosły algorytmy AlphaStar[13] oraz OpenAI Five[8]. Gry StarCraft2 oraz Dota2 są uważane za najtrudniejsze gry dla algorytmów, ponieważ są grami czasu rzeczywistego, z niepełną informacją oraz wymagają planowania z bardzo dużym wyprzedzeniem. Obie gry gry nie mają optymalnej strategii, która gwarantuje zwycięstwo w każdej sytuacji, gdyż na każdą strategię istnieje kontra strategia na zasadzie kartka kamień nożyce. W przeciągu ostatnich lat uczenie ze wzmocnieniem było bardzo dynamicznie rozwijane. Dotychczasowi mistrzowie byli regularnie pokonywani przez algorytmy. Ten ciąg sukcesów udowodnił, że uczenie ze wzmocnieniem jest w stanie nauczyć się lepszych strategii podejmowania decyzji w krót-

kim czasie, ale jak na razie tylko w sztucznych środowiskach. Wykorzystywanie uczenia ze wzmocnieniem do rozwiązywania rzeczywistych problemów wiąże się z innymi problemami, które nie występują w sztucznych środowiskach. Jednym z dziewięciu problemów uczenia ze wzmocnieniem, opisanych w publikacji Challenges of Real-World Reinforcement Learning[3], które muszą zostać zaadresowane przed popularyzacją aplikacji uczenia ze wzmocnieniem, jest możliwość przejrzystej interpretacji zachowań agenta, który chciałbym poruszyć w tej pracy.

3 Środowisko

Na potrzeby tej pracy zaimplementowałem w języku C++ ciągłą, dwuwymiarową przestrzeń po której poruszają się pojazdy, którymi steruje wątek uczenia ze wzmocnieniem. Pojazdy przemieszczają się zgodnie z zasadami dynamiki Newtona, bez oporów wpływających na ich aktualną prędkość. Środowisko jest obiektem tworzonym i kontrolowanym przez wątek uczenia ze wzmocnieniem w Pythonie. Środowisko korzysta z bibliotek SFML[5], wykorzystywanej do renderowania stanu środowiska za pomocą prymitywnych kształtów, oraz BOOST[1], która dostarcza możliwość wywoływania funkcji zaimplementowanych w C++ z poziomu wątku Pythona. BOOST implementuje również możliwość przekazywania adresów pamięci wektorów Numpy pomiędzy symulacją a algorytmem uczenia maszynowego. Interfejs komunikacji ze środowiskiem złożony jest z pięciu funkcji: konstruktor zwracający inteligentny współdzielony wskaźnik, funkcja aktualizacji stanu środowiska, funkcja pozwalająca przywrócić środowisko do stanu początkowego, informacja o tym czy środowisko jest nadal aktywne oraz destruktor. Funkcja aktualizacji stanu środowiska przyjmuje na wejściu akcje i zwraca obserwacje, nagrodę oraz informacje o tym, czy pojazd został zresetowany. Funkcja przywracająca Środowisko do stanu początkowego zwraca tylko obserwacje.

3.1 Akcje

Algorytm uczenia ze wzmocnieniem kontroluje ruch pojazdu przy każdej aktualizacji stanu środowiska podejmując decyzję z jaką mocą mają działać dwa silniki. Pierwszy silnik nadaje przyspieszenie w kierunku przodu pojazdu, a drugi przyspieszenie obrotowe. Dla uproszczenia treningu przestrzeń możliwych akcji pierwszego silnika jest dwuelementowa: włączony i wyłączony $[1, 0]$. Drugi silnik posiada trzy możliwe akcje: lewo, prawo, wyłączony $[-1, 0, 1]$. Przestrzeni akcji powstaje z iloczynu kartezjańskiego wektorów możliwych akcji obu silników. Dodatkowo z przestrzeni akcji usuwana jest akcja neutralna, oba silniki są wyłączone, gdyż w początkowym etapie uczenia ta akcja była faworyzowana, co prowadziło do stagnacji procesu uczenia. Daje to pięcioelementową przestrzeń akcji.

3.2 Obserwacje

Algorytm uczenia podejmuje decyzję którą akcję wykonać w danym kroku na podstawie obserwacji otrzymywanych ze środowiska. Obserwacje składają się z trzech liczb rzeczywistych odpowiadających kolejno szybkości pojazdu, kątowi pomiędzy kierunkiem statku a wektorem prędkości oraz szybkości kątowej.

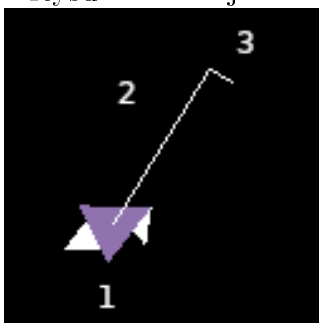
$$\begin{aligned} &1)\ |Velocity| \\ &2)\ \measuredangle ShipDirection - \measuredangle Velocity \\ &3)\ AngualrVelocity \end{aligned} \tag{1}$$

Dodatkowo w scenariuszach w których celem jest dotarcie pojazdem do wyznaczonego punktu kontrolnego obserwacje rozszerzane są o dwie następne liczby rzeczywiste odpowiadające odległości pojazdu od wyznaczonego punktu oraz kątowi pomiędzy kierunkiem statku a wektorem odległości.

$$\begin{aligned} &4)\ |CheckpointPosition - ShipPosition| \\ &5)\ \measuredangle ShipDirection - \measuredangle(CheckpointPosition - ShipPosition) \end{aligned} \tag{2}$$

3.3 Graficzna reprezentacja środowiska

Rysunek 1: Pojazd.



Pojazd (1) jest reprezentowany przez trzy trójkąty. Pierwszy największy kolorowy trójkąt reprezentuje pojazd. Najostrzejszy wierzchołek wskazuje przód pojazdu. Wielkość pozostałych dwóch trójkątów odpowiada mocy silników. Trójkąt połączony z krótszą krawędzią pojazdu reprezentuje główny silnik, a trójkąt połączony z najostrzejszym wierzchołkiem pojazdu reprezentuje silnik rotujący. Odcinki (2) oraz (3) są graficzną reprezentacją stanu pojazdu i odpowiadają prędkości oraz prędkości obrotowej.

3.4 Funkcja aktualizacji środowiska

Algorytm uczenia ze wzmocnieniem aktualizuje stan środowiska podając indeks wybranej akcji. W pierwszej kolejności obliczane jest przyspieszenie działające na pojazd, które zależy od wybranej przez algorytm mocy głównego silnika.

$$\begin{cases} Acceleration_x = firstAction * cos(ShipDirection) * mainEnginePower/mass \\ Acceleration_y = firstAction * sin(ShipDirection) * mainEnginePower/mass \end{cases} \tag{3}$$

Następnie obliczana jest prędkość pojazdu $Velocity = Velocity + Acceleration * timeStep$ oraz nowa pozycja pojazdu $ShipPosition = ShipPosition + Velocity * timeStep$. Środowisko aktualizowane jest ze stałym krokiem czasowym wynoszącym dziesiątą część sekundy.

$$
\begin{aligned}
AngularAcceleration &= secondAction * rotationEnginePower/(0.5 * mass * size) \\
AngualarVelocity &= AngualarVelocity + AngularAcceleration * timeStep \\
ShipDirection &= ShipDirection + AngularVelocity * timeStep
\end{aligned} \tag{4}
$$

Po zaktualizowaniu położenia pojazdu obliczana jest nagroda oraz sprawdzane są warunki kończące symulację pojazdu. Po aktualizacji funkcja zwraca krotkę z adresami pamięci, pod którymi znajdują się wektory zawierające następne obserwacje, nagrody oraz informację mówiącą o tym, czy pojazd zakończył symulację.

3.5 Scenariusze i modelowanie funkcji nagrody

Na potrzeby tej pracy zaimplementowałem trzy scenariusze różniące się pomiędzy sobą sposobem obliczania nagrody oraz początkowym stanem pojazdu. We wszystkich scenariuszach pojazd posiada te same akcje, parametry oraz sposób aktualizacji. Scenariusz wybierany jest na podstawie parametru podanego podczas inicjalizacji obiektu środowiska. Interfejs środowiska jest niezmienny.

Celem pierwszego scenariusza jest nauczenie agenta zmniejszania szybkości rozpędzonego pojazdu. Pojazd rozpoczyna symulację posiadając losową rotację, losową prędkość obrotową z zakresu $[-2, 2]$ rad/s oraz wektor prędkości którego elementy losowane są z zakresu $[5, 10]$ m/s. Po każdej aktualizacji środowiska agent otrzymuje nagrodę odpowiadającą odwrotnej zmianie szybkości w danym kroku $r_t = |Velocity_{t-1}| - |Velocity_t|$. Gdy szybkość pojazdu przekroczy 20 m/s lub gry szybkość obrotowa przekroczy 2 rad/s stan pojazdu jest ponownie losowany tak samo jak na początku symulacji, a agent otrzymuje ujemną nagrodę za wybranie akcji prowadzącej do spełnienia jednego z powyższych warunków. Agent otrzymuje dodatnią nagrodę po wykonaniu akcji prowadzącej do zmniejszenia szybkość pojazdu poniżej 2 m/s, po której stan pojazdu również jest losowany na nowo.

Celem drugiego i trzeciego scenariusza jest nauczenie agenta dolatywania pojazdem do wyznaczonego losowo punktu kontrolnego. Podobnie jak w pierwszym scenariuszu pojazd na początku posiada losową rotację, prędkość obrotową $[-1, 1]$ rad/s, oraz elementy wektora prędkość z zakresu $[0, 10]$ m/s. Pozycja pojazdu ustawiana jest na środek układu współrzędnych. Współrzędne punktu kontrolnego losowane są z zakresu $[-700, 700]$ m. Gdy pojazd zbliży się do punktu kontrolnego na mniej niż 25 metrów pojazd oraz punkt kontrolny są losowane na nowo, a agent otrzymuje dodatnią nagrodę. Gdy pojazd oddali się od punktu kontrolnego na więcej niż 1400 m lub jego szybkość obrotowa przekroczy 2 rad/s agent otrzymuje ujemną nagrodę, a stan pojazdu jest na nowo losowany. Punkt kontrolny pozo-

staje w tym samym położeniu. W drugim scenariuszu nagroda otrzymywana przez agenta po każdej aktualizacji odpowiada odwrotnej zmianie odległości pojazdu od punktu kontrolnego. W trzecim scenariuszu nagroda odpowiada zmianie szybkości w kierunku punktu kontrolnego.

4 Definicja uczenia ze wzmocnieniem

Uczenie ze wzmocnieniem polega na optymalizacji strategii wybierania akcji w oparciu o doświadczenia generowane podczas interakcji ze środowiskiem. Proces decyzyjny Markowa jest matematycznym sformułowaniem procesu interakcji agenta ze środowiskiem, którego celem jest osiągnięcie najwyższej możliwej nagrody. Składa się z krotki (S, A, P, R), gdzie S to zbiór stanów środowiska w jakich może się znaleźć symulowany pojazd, A to zbiór akcji, $P(s'|s, a)$ to rozkład prawdopodobieństwa przejścia pomiędzy stanami w zależności od wybranej akcji, $R(s, a)$ to funkcja nagrody otrzymywanej przez agenta natychmiast po wykonaniu akcji w zadanym stanie. Proces interakcji ze środowiskiem tworzy sekwencję doświadczeń $S_1, A_1, R_1, S_2, A_2, R_2 ... S_T, A_T, R_T$, gdzie T oznacza numer kroku, w którym symulacja pojazdu jest resetowana. Politykę π definiuje się jako funkcję rozkładu prawdopodobieństwa akcji na zadanym stanie.

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s] \tag{5}$$

Optymalna polityka π^* w każdym stanie ze zbioru S wybierze taką akcję, która da maksymalną obniżoną nagrodę którą agent jest w stanie uzyskać z danego stanu. Parametr zniżki $0 < \gamma < 1$ zmniejszający nagrody z przyszłych kroków zmusza agenta do podejmowania dobrych decyzji wcześniej. Gdyby zniżki przyszłych nagród nie było agent mógłby odwlekać wybranie dobrej akcji w nieskończoność. Maksymalna obniżona nagroda możliwa do uzyskania z danego stanu zdefiniowana jest jako optymalna funkcja wartości.

$$V^*(s) = \max_a [R(s, a) + \gamma \sum_{s'} P(s'|s, a) V^*(s')] \tag{6}$$

Maksymalna oczekiwana nagroda do uzyskania z danego stanu po podjęciu określonej akcji to optymalna funkcja wartości akcji Q^*.

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q^*(s', a') \tag{7}$$

$$V^*(s) = \max_a Q^*(s, a)$$

Jeśli optymalna wartość $Q^*(s', a')$ w następnym kroku jest znana dla wszystkich możliwych akcji a' to optymalną polityką podejmowania akcji jest wybieranie a' maksymalizując oczekiwaną obniżoną nagrodę.

$$\pi^*(s) = arg \max_a Q^*(s, a) \tag{8}$$

$$V_\pi(s) = \sum_a \pi(a|s)[R(s,a) + \gamma \sum_{s'} P(s'|s,a)V_\pi(s')] \tag{9}$$

$$Q_\pi(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) \sum_{a'} \pi(a'|s')Q_\pi(s',a') \tag{10}$$

$$V_\pi(s) = \sum_a \pi(a|s)Q_\pi(s,a)$$

Funkcję (10) można rozwiązać iterując równanie Temporal Difference[12], gdzie α to wielkość kroku optymalizacyjnego. Taka iteracja funkcji wartości akcji zbiega do optymalnej funkcji $Q_i \to Q^*$ dla $i \ \to \infty$[12].

$$Q_{i+1}(s,a) = Q_t(s,a) + \alpha[R(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q_t(s',a') - Q_t(s,a)] \tag{11}$$

Takie podejście jest niepraktyczne w przypadku tej pracy, gdyż przestrzeń stanów jest za duża oraz algorytm uczący nie ma dostępu do funkcji $P(s'|s,a)$.

5 Głębokie uczenie funkcji wartości akcji Q

Bez dostępu do funkcji przejścia $P(s'|s,a)$ proces optymalizacji musi bazować na seriach doświadczeń, złożonych z krotek zawierających obserwację S_t, akcję A_t oraz otrzymaną nagrodę R_t, które są generowane podczas interakcji ze środowiskiem. Elementy wszystkich serii są na zapisywane w pamięci M. Podczas analizy serii, bez podanej funkcji $P(s'|s,a)$, funkcja wartości jak i wartości akcji opiera się na założeniu, że kolejne akcje w serii były wybierane zgodnie z polityką π.

$$V_\pi(s) = \mathbb{E}[R_t + \gamma V_\pi(S_{t+1})|S_t = s, A_t \sim \pi(S_t)] \tag{12}$$

$$Q_\pi(s,a) = \mathbb{E}[R_t + \gamma Q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a, \pi] \tag{13}$$

Celem znalezienia optymalnej strategii korzystam z funkcji z parametrami θ która aproksymuje optymalną funkcję wartości akcji $Q(s,a:\theta) \approx Q^*(s,a)$. W tej pracy funkcją jest głęboka sieć neuronowa z nieliniowymi aktywacjami. Sieć neuronową można uczyć minimalizując sekwencje funkcji kosztu $L_i(\theta_i)$. Funkcja kosztu odpowiada kwadratowi funkcji Temporal Difference z tą różnicą, że stan s, akcja a, nagroda r oraz następny obserwowany stan s' są losowane z pamięci wcześniejszych doświadczeń M.

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s' \sim M}[(r + \gamma \max_{a'} Q(s',a':\theta_{i-1}) - Q(s,a:\theta_i))^2] \tag{14}$$

Wagi sieci neuronowej optymalizuje się za pomocą stochastycznego spadku wzdłuż gradientu.

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a,r,s' \sim M}[(r + \gamma \max_{a'} Q(s',a':\theta_{i-1}) - Q(s,a:\theta_i))\nabla_{\theta_i} Q(s,a:\theta_i)] \tag{15}$$

Krok optymalizacyjny następuje po każdej aktualizacji stanu środowiska. Wagi θ_{t-1} są podczas treningu zamrożone. Korzystanie z wag z poprzedniej aktualizacji mogłoby prowadzić do katastroficznego zapominania, dlatego wagi θ_{t-1} zastępowane są wagami θ_f które co stałą ilość kroków przyjmują aktualną wartość θ_t .

5.1 Problemy głębokiego uczenia ze wzmocnieniem

Kolejne stany środowiska są bardzo do siebie podobne, co w procesie uczenia prowadziłoby do nadmiernego dopasowania funkcji Q bądź zbiegania do lokalnego minimum. Celem uniknięcia tej sytuacji pamięć M ma dużą pojemność n, a partie do uczenia o wielkości $b < n$ są wybierane z pamięci z jednakowym prawdopodobieństwem $(S_b, A_b, R_b, S'_b) \sim M$. Prowadzi to do zwiększenia niezależności obserwacji, na podstawie których obliczany jest krok uczący. Drugim sposobem na przeciwdziałanie podobieństwa stanów jest jednoczesne symulowanie wielu niezależnych pojazdów, które są inicjalizowane w sposób losowy. Znacząco zwiększa to różnorodność obserwowanych stanów. Prowadzi to do lepszej eksploracji przestrzeni stanów oraz pozwala na lepszą ewaluację aktualnej polityki podczas treningu. Wyniki przeprowadzonych eksperymentów wskazują na znaczącą poprawę wyników już przy pięciu symulowanych jednocześnie pojazdach.

Gdyby agent od początku chciwie podejmował akcje na podstawie losowo zainicjalizowanej funkcji wartości akcji $a_t = \max_a Q(s_t, a : \theta)$ mógłby, przez niedostateczną eksplorację, pozostać w lokalnym minimum. Eksplorację wymusza się na agencie poprzez wybieranie z prawdopodobieństwem ϵ losowej akcji $a_t \sim A$, a z prawdopodobieństwem $1-\epsilon$ akcji chciwej $a_t = \max_a Q(s_t, a : \theta)$. Ten sposób podejmowania akcji nazywa się ϵ-greedy[12]. Wartość ϵ jest warunkowana początkową wartością ϵ_{start}, końcową wartością ϵ_{end} oraz liczbą kroków ϵ_{decay} po których ϵ ma osiągnąć wartość końcową. W trakcie treningu ϵ zmniejsza się z każdym krokiem zgodnie z równaniem $\epsilon_t = \epsilon_{end} + (\epsilon_{start} - \epsilon_{end})e^{(-t/\epsilon_{decay})}$.

Następnym problemem w głębokim uczeniu ze wzmocnieniem są eksplodujące gradienty. Główną przyczyną eksplozji jest duże odchylenie standardowe zaobserwowanych nagród, obserwacje zawierające zmienne o różnych skalach wielkości. Wagi sieci θ_{t-1} służą do obliczania funkcji kosztu jak i na podstawie wag θ są generowane obserwacje podczas interakcji ze środowiskiem. Z tych powodów problem eksplodujących gradientów występuje w uczeniu ze wzmocnieniem częściej niż w przypadku innych dziedzin uczenia maszynowego. Żeby przeciwdziałać eksplodującym gradientom korzystam z funkcji kosztu $smooth\_L1\_loss$[4] celem

zmniejszenia wpływu skrajnych pomiarów na całkowity gradient danego kroku uczącego.

$$smooth_{L_1}(x) = \begin{cases} 0.5x^2 & dla|x| < 1 \\ |x| - 0.5 & dla|x| \geq 1 \end{cases} \tag{16}$$

Podobnie jak w Atari DQN[7] korzystam z optymalizatora *RMSprop* zaproponowanego przez G. Hintona[14] i po raz pierwszy opublikowanego w pracy na temat optymalizacji rekurencyjnych sieci neuronowych służących do generowania tekstu[6]. Optymizator przechowuje średnią kroczącą kwadratów gradientów dla każdej wagi $w \in \theta$, a podczas aktualizacji wagi dzieli gradient przez pierwiastek kwadratowy tej średniej.

$$\mathbb{E}[g^2]_t = \beta\mathbb{E}[g^2]_{t-1} + (1-\beta)(\frac{\delta L(x)}{\delta w})^2 \tag{17}$$

$$w_t = w_{t-1} - \frac{\alpha}{\sqrt{\mathbb{E}[g^2]_t}}\frac{\delta L(x)}{\delta w} \tag{18}$$

5.2 Algorytm

Algorithm 1: Głębokie uczenie Q-network z wykorzystaniem pamięci

Inicjalizacja parametrów θ funkcji wartości akcji Q, $\theta_f = \theta$
Inizjalizacja środowiska oraz otrzymanie pierwszej obserwacji s_1
for *t=0; t<T; t++* **do**
  Z prawdopodobieństwem ϵ wybierz losową akcję $a_t \sim A$,
  w innym przypadku wybierz $a_t = \max_a Q(s_t, a : \theta)$
  Zaktualizuj stan środowiska wykonując akcję a_t otrzymując (r_t, s_{t+1})
  Zapisz w pamięci element (s_t, a_t, r_t, s_{t+1})
  if *t>b* **then**
    $(S_b, A_b, R_b, S'_b) \sim M$
    Zaktualizuj wagi θ zgodnie z $(R_b + \gamma \max_{a'} Q(S'_b, a' : \theta_f) - Q(S_b, A_b : \theta))^2$
  end
  if *(t % częstotliwość aktualizacji θ_f) == 0* **then**
    $\theta_f = \theta$
  end
end

5.3 Sztuczna sieć neuronowa

Funkcja aproksymująca Q^* jest złożona z wielu warstw funkjci liniowych zawierających nieliniowe aktywacje pomiędzy kolejnymi warstwami. Pierwsza funkcja liniowa przyjmuje na

wejściu wektor obserwacji otrzymany ze środowiska, przekształca na wektor o wymiarze w a następnie oblicza nieliniową funkcję aktywacji $ReLu$[2] na wektorze wyjściowym. Kolejne d funkcji liniowych o tej samej szerokości w przekształca wektor w ten sam sposób co pierwsza funkcja. Ostatnia funkcja liniowa przekształca wektor o wymiarze w na wektor o wymiarze odpowiadającym rozmiarowi przestrzeni akcji A. Wektor wyjściowy odpowiada wartości wszystkich akcji. W tej pracy korzystam z algorytmów uczenia maszynowego zaimplementowanych w bibiliotece PyTorch[9].

6 System do przeprowadzania eksperymentów

Chcąc przetestować w jaki sposób algorytm uczenia DQN reaguje na różne parametry potrzebowałem systemu przechowującego wyniki przeprowadzonych eksperymentów, kolejkę eksperymentów do wykonania oraz proces, który wykonuje kolejne eksperymenty. Dzięki uprzejmości Naukowego Koła Robotyki i Sztucznej Inteligencji na Uniwersytecie Jagiellońskim miałem dostęp do dwóch maszyn obliczeniowych, co wymagało napisania dodatkowych narzędzi odpowiedzialnych za synchronizację kolejki eksperymentów i wyników.

6.1 Tworzenie eksperymentów

Szablon eksperymentów tworzony jest po wskazaniu plików tekstowych w formacie JSON, które zawierają domyślne parametry potrzebne do uruchomienia danego eksperymentu. Szablon można uruchomić jako eksperyment lub można go wykorzystać jako bazę do stworzenia wielu eksperymentów, nazywanych serią, różniących się pomiędzy sobą parametrami. Serię eksperymentów tworzy się podając folder, w którym znajduje się szablon, oraz listy wartości parametrów. Eksperymenty tworzone są z elementów iloczynu wszystkich list wartości parametrów, a następnie zapisywane są w folderze kolejki.

6.2 Uruchamianie obliczeń

Po stworzeniu serii można lokalnie uruchomić skrypt *runq.sh* wykonujący lokalnie eksperymenty z kolejki. W celu wysłania części eksperymentów na zdalne maszyny trzeba uruchomić skrypt *split_training.sh* który dzieli eksperymenty z kolejki i wysyła je przez ssh do maszyn obliczeniowych. Po uruchomieniu skryptu eksperymenty są losowo dzielone na równe części dla każdej z maszyn zapisanych w pliku $remote\_config.py$. Po podzieleniu eksperymenty są kompresowane a następnie przesyłane na maszynę. Po przesłaniu pliku eksperymenty są rozpakowywane i umieszczane w folderze kolejki. Za pomocą skryptu *work.py* można uruchomić skrypt *runq.sh* na wszystkich zdalnych maszynach. Podczas obliczeń można uruchomić

skrypt *status.py* celem sprawdzenia ilości pozostałych eksperymentów na maszynach oraz utylizacji procesora kart graficznych.

6.3 Synchronizacja wyników

Żeby pobrać z maszyn wyniki eksperymentów które zostały zakończone trzeba uruchomić skrypt *pull.py*. Skrypt ten wysyła na maszynę listę plików które znajdują się lokalnie. Maszyna po porównaniu lokalnych plików z otrzymaną listą kompresuje te pliki, które nie znajdowały się na liście. Skrypt następnie pobiera skompresowane wyniki i rozpakowuje je lokalnie.

7 Przeprowadzone eksperymenty

8 Wyniki i ich interpretacja

8.1 Pierwszy scenariusz

8.2 Drugi scenariusz

8.3 Trzeci scenariusz

9 pomysły i propozycje następnych eksperymentów, badań

Literatura

[1] boost c++ libraries.

[2] Abien Fred Agarap. Deep learning using rectified linear units (relu). *CoRR*, abs/1803.08375, 2018.

[3] Gabriel Dulac-Arnold, Daniel J. Mankowitz, and Todd Hester. Challenges of real-world reinforcement learning. *CoRR*, abs/1904.12901, 2019.

[4] Ross B. Girshick. Fast R-CNN. *CoRR*, abs/1504.08083, 2015.

[5] Laurent Gomila. Simple and fast multimedia library sfml.

[6] Alex Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[8] OpenAI. Openai five defeats dota 2 world champions, 2019.

[9] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alche-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[10] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[11] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362:1140–1144, 12 2018.

[12] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[13] Oriol Vinyals, Igor Babuschkin, Wojciech Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John Agapiou, Max Jaderberg, and David Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575, 11 2019.

[14] Geoffrey Hinton with Nitish Srivastava Kevin Swersky. Neural networks for machine learning.

Spis rysunków

Spis tablic