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. Piotra Białasa
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

W niniejszej pracy przedstawiam wynik serii eksperymentów mających na celu zbadanie skuteczności i niezawodności prostego wariantu algorytmu uczenia ze wzmocnieniem Deep Q-Networks[6]. Zadaniem algorytmu jest uczenie sztucznej sieci neuronowej sterowania pojazdami w symulowanej przestrzeni kosmicznej. Z przeprowadzonych eksperymentów wynika, że algorytm jest skuteczny w znajdywaniu dobrych strategii w prostych środowiskach podczas gdy w bardziej skomplikowanych środowiskach algorytm rzadziej znajduje dobre strategie, a parametry muszą być precyzyjnie dobierane.

2 Wstęp

Uczenie nadzorowane, najczęściej wykorzystywany rodzaj uczenia maszynowego, polega na uczeniu funkcji na podstawie zbioru danych. Zbiór ten zawiera w sobie dane wejściowe oraz oczekiwany wynik. Po predykcji wyniku na podstawie danych wejściowych znana jest dokładna wartość popełnionego błędu. Uczenie ze wzmocnieniem polega na uczeniu funkcji na podstawie interakcji ze środowiskiem. Dane wejściowe są obserwowane podczas symulacji, a całkowita nagroda(błąd) poznawana jest dopiero po zakończeniu symulacji. Przykładem może być np. nauka gray w szachy, gdzie dopiero po zakończeniu gry poznajemy wynik. Nie mamy jednak informacji o wartości poszczególnych ruchów przez co jest to dużo bardziej skomplikowany problem. Algorytm Atari DQN[6] był w stanie nauczyć sztuczną sieć neuronową strategii podejmowania dobrych decyzji w siedmiu grach na platformie Atari. Nauka kolejnych gier Atari nie wymagała modyfikacji w algorytmie. Moim celem w tej pracy jest zbadanie skuteczności algorytmu w znajdywaniu dobrych strategii w trzech środowiskach.

3 Środowisko

Agent podlegający uczeniu steruje pojazdami poruszającymi się po dwuwymiarowej, ciągłej przestrzeni kosmicznej. Pojazdy poruszają się zgodnie z zasadami dynamiki Newtona, bez oporów wpływających na ich aktualną prędkość. Środowisko, napisane w C++, jest kontrolowane przez agenta, napisanego w Pythonie. Interfejs komunikacji środowiska z agentem składa się trzech funkcji: funkcji aktualizacji stanu środowiska, funkcja pozwalająca przywrócić środowisko do stanu początkowego oraz funkcji zwracającej informację o tym, czy środowisko jest nadal aktywne. Funkcja aktualizacji środowiska przyjmuje na wejściu akcje i zwraca następny stan, nagrodę oraz informacje o tym, czy pojazd został zresetowany. Funkcja resetująca środowisko do stanu początkowego zwraca tylko stan.

3.1 Akcje

Przy każdej aktualizacji środowiska zadaniem agenta jest kontrolowanie ruchu pojazdu poprzez wybranie jednej z możliwych akcji. Wybrana przez agenta akcja odpowiada z jaką mocą mają działać dwa silniki. Pierwszy silnik nadaje przyspieszenie w kierunku przodu pojazdu, drugi przyspieszenie obrotowe. Pierwszy silnik posiada dwie możliwe akcje: wyłączony albo włączony $[0, 1]$. Drugi silnik posiada trzy możliwe akcje: lewo, wyłączony, prawo $[-1, 0, 1]$. Przestrzeni akcji powstaje z iloczynu kartezjańskiego zbiorów możliwych akcji obu silników. Z przestrzeni akcji usuwana jest akcja neutralna, kiedy oba silniki są wyłączone, gdyż w początkowym etapie uczenia ta akcja była faworyzowana, co prowadziło do stagnacji procesu uczenia.

3.2 Stan

Agent podejmuje decyzję którą akcję wykonać w danym kroku na podstawie stanu otrzymanego ze środowiska. Stan obserwowany przez agenta w składa się z trzech liczb rzeczywistych odpowiadających kolejno szybkości pojazdu, kątowi pomiędzy kierunkiem statku a wektorem prędkości oraz prędkości kątowej.

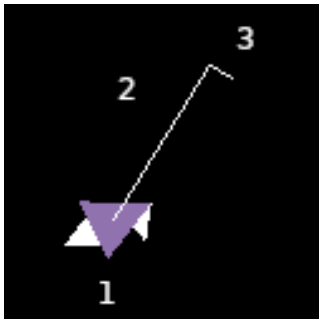
$$\begin{aligned} &1)\ |Velocity| \\ &2)\ \measuredangle ShipDirection - \measuredangle Velocity \\ &3)\ AngularVelocity \end{aligned} \tag{1}$$

W scenariuszach, w których celem jest dotarcie pojazdem do wyznaczonego punktu kontrolnego, stan zawiera dodatkowo dwie liczby rzeczywiste odpowiadające odległości pojazdu od punktu kontrolnego oraz kątowi pomiędzy kierunkiem statku a wektorem odległości do tego punktu.

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

3.3 Graficzna reprezentacja środowiska

Rysunek 1: Pojazd.



Na rysunku po lewej stronie pojazd 1 jest reprezentowany przez trzy trójkąty. Kolorowy trójkąt reprezentuje kadłub pojazd, a jego najostrzejszy wierzchołek wskazuje przód. Dwa białe trójkąty pokazują włączone silniki. Trójkąt połączony z krótszą krawędzią pojazdu reprezentuje główny silnik, a trójkąt połączony z przodem pojazdu reprezentuje silnik rotujący. Odcinki 2 - prędkości oraz 3 - prędkość obrotowa są graficzną reprezentacją stanu. Do renderowania środowiska używam biblioteki SFML[4].

3.4 Funkcja aktualizacji środowiska

Po wybraniu przez agenta akcji następuje aktualizacja stanu środowiska. Podczas kroku aktualizacji środowiska w pierwszej kolejności obliczane jest przyspieszenie działające na pojazd, które zależy od wybranej przez agenta akcji decydującej o mocy głównego silnika. Parametry pojazdu $mainEnginePower$, $mass$, $rotationEnginePower$ oraz $size$ są stałe.

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

Po zaktualizowaniu położenia pojazdu obliczana jest rotacja pojazdu.

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

Środowisko aktualizowane jest ze stałym krokiem czasowym wynoszącym dziesiątą część sekundy. Po zaktualizowaniu położenia oraz rotacji pojazdu obliczana jest nagroda oraz sprawdzane są warunki kończące symulację pojazdu. Funkcja zwraca krotkę z adresami pamięci, pod którymi znajdują się wektory zawierające następny stan, nagrodę oraz informację czy pojazd zakończył symulację w tym kroku. Do komunikacji pomiędzy Pythonem a C++ z biblioteki BOOST[1].

3.5 Scenariusze i modelowanie funkcji nagrody

Na potrzeby eksperymentów zaimplementowałem trzy scenariusze różniące się pomiędzy sobą sposobem obliczania nagrody oraz poziomem trudności.

Celem pierwszego, najprostszego 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 $r_t = |Velocity_{t-1}| - |Velocity_t|$. Pojazd ulega zniszczeniu i jest resetowany gdy jego szybkość przekroczy 20 m/s lub gry szybkość obrotowa przekroczy 2 rad/s. Agent otrzymuje karę za wybranie akcji prowadzącej do zniszczenia pojazdu. Gdy szybkość pojazdu zmaleje poniżej 2 m/s agent otrzymuje nagrodę za zaliczenie scenariusza i ocalenie pojazdu. Otrzymywana przez agenta nagroda/kara po osiągnięciu stanu terminalnego we wszystkich scenariuszach wynosi ± 1.

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 prędkości losowane z zakresu $[0, 10]$ m/s. Startową pozycją pojazdu jest środek układu współrzędnych. Współrzędne punktu kontrolnego losowane są z zakresu $[-700, 700]$ m. Gdy pojazd zbliży się do punktu kontrolnego na odległość mniejszą niż 25 m agent otrzymuje nagrodę, a pojazd oraz punkt kontrolny są losowane na nowo, . Gdy pojazd oddali się od punktu kontrolnego na więcej niż 1400 m lub jego szybkość obrotowa przekroczy 2 rad/s agent otrzymuje karę, pojazd ulega zniszczeniu a punkt kontrolny pozostaje w tym samym miejscu. W drugim scenariuszu nagroda otrzymywana przez agenta po każdej aktualizacji odpowiada odwrotnej zmianie odległości pojazdu od punktu kontrolnego $r_t = d_{t-1} - d_t$, gdzie d_t odpowiada odległości euklidesowej pojazdu od punktu kontrolnego w kroku t. W trzecim scenariuszu nagroda odpowiada zmianie szybkości w kierunku punktu kontrolnego $r_t = d_{t-2} - 2 * d_{t-1} + d_t$.

4 Definicja uczenia ze wzmocnieniem

Uczenie ze wzmocnieniem polega na optymalizacji strategii wybierania akcji na podstawie doświadczeń generowanych podczas interakcji ze środowiskiem. Matematycznym sformułowaniem procesu interakcji agenta ze środowiskiem jest proces decyzyjny Markowa, 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 serię doświadczeń $s_1, a_1, r_1, s_2, a_2, r_2 ... s_T, a_T, r_T$, gdzie T oznacza numer kroku terminalnego, w którym kończy się epizod i symulacja pojazdu jest resetowana. Obniżona nagroda geometrycznie zmniejsza wartość nagród otrzymywanych w następnych krokach.

$$R_t = \sum_{i=t}^{T} \gamma^{(i-t)} r_i \tag{5}$$

Parametr zniżki $0 < \gamma < 1$ zmniejszając przyszłe nagrody przeciwdziała zbieganiu sumy nagród do nieskończoności oraz wymusza na agencie wybieranie dobrych akcji wcześniej zamiast je odwlekać. 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^*.

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

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

Strategię π zdefiniowana jest jako funkcja rozkładu prawdopodobieństwa akcji na zadanym stanie.

$$\pi(a|s) = \mathbb{P}[a_t = a|s_t = s] \tag{8}$$

Jeśli, tak jak w przypadku tej pracy, strategia jest deterministyczna, to zwraca ona wartość 1 dla wybranej akcji i 0 dla wszystkich pozostałych. Optymalna strategia π^* w każdym stanie $s \in S$ wybierze taką akcję a, która maksymalizuje obniżoną nagrodę. Jeśli optymalna wartość $Q^*(s',a')$ w następnym kroku jest znana dla wszystkich możliwych akcji a', to optymalną strategią podejmowania akcji jest wybieranie a' maksymalizując oczekiwaną obniżoną nagrodę.

$$\pi^*(s) = arg \max_a Q^*(s,a) \tag{9}$$

Funkcję wartości oraz funkcję wartości akcji można zdefiniować dla dowolnej polityki π.

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

$$Q_\pi(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) \sum_{a'} \pi(a'|s')Q_\pi(s',a') \tag{11}$$

Podczas uczenia dążymy do minimalizacji funkcji kosztu Temporal Difference[8].

$$TD = \min_Q [Q^*(s,a) - Q(s,a)] \tag{12}$$

Funkcję (12) można minimalizować iterując poniższe równanie (13). Parametr α opisuje wielkość kroku optymalizacyjnego. Taka iteracja funkcji wartości akcji zbiega do optymalnej funkcji $Q_i \to Q^*$ dla $i \to \infty$[8].

$$Q_{i+1}(s,a) = Q_t(s,a) + \alpha[R(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q_t(s',a') - Q_t(s,a)] \tag{13}$$

W przypadku tej pracy takie podejście jest niepraktyczne, gdyż przestrzeń stanów jest za duża, agent nie zna funkcji $P(s'|s,a)$, epizody mogą być bardzo długie oraz chcemy, żeby funkcja Q była uczona w trakcie trwania symulacji, a nie dopiero po jej zakończeniu.

5 Głębokie uczenie funkcji wartości akcji Q

Bez dostępu do funkcji przejścia $P(s'|s,a)$ proces optymalizacji bazuje na seriach doświadczeń generowanych podczas interakcji ze środowiskiem. Wszystkie doświadczenia, razem z następnym zaobserwowanym stanem s', są zapisywane w pamięci M. Funkcja wartości jak i funkcja wartości akcji opiera się na założeniu, że kolejne akcje w serii były wybierane zgodnie ze strategią π.

$$V_\pi(s) = \mathbb{E}[r_t + \gamma V_\pi(s_{t+1})|s_t = s, a_t \sim \pi(s_t)] \tag{14}$$

$$Q_\pi(s,a) = \mathbb{E}[r_t + \gamma Q_\pi(s_{t+1}, a_{t+1})|s_t = s, a_t = a, \pi] \tag{15}$$

Celem znalezienia optymalnej strategii korzystam ze sztucznej sieci neuronowej z parametrami θ, której zadaniem jest uczenie optymalnej funkcji wartości akcji $Q(s,a:\theta) \approx Q^*(s,a)$. Sieć neuronową można uczyć minimalizując sekwencje funkcji kosztu $L_i(\theta_i)$. Funkcja kosztu odpowiada kwadratowi funkcji (12) z tą różnicą, że stan s, akcja a, nagroda r oraz następny stan otrzymany po aktualizacji środowiska s' są losowane z pamięci doświadczeń M.

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'\sim M}[(r + \gamma \max_{a'} Q(s', a' : \theta_{i-1}) - Q(s, a : \theta_i))^2] \tag{16}$$

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

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a,r,s'\sim M}[(r + \gamma \max_{a'} Q(s', a' : \theta_{i-1}) - Q(s, a : \theta_i))\nabla_{\theta_i} Q(s, a : \theta_i)] \tag{17}$$

Wagi θ_{t-1} są podczas treningu zamrożone. Korzystanie z wag z poprzedniej aktualizacji mogłoby spowodować wybieranie tych samych nieoptymalnych akcji na podstawie poprzednich wag. Prowadzi to do katastroficznego zapominania. Z tego powodu wagi θ_{t-1} są zastąpione wagami θ_f, które co stałą liczbę kroków $target\_update\_frequency$ przyjmują wartość θ_t.

5.1 Problemy głębokiego uczenia ze wzmocnieniem

Kolejne stany środowiska są bardzo do siebie podobne, co w procesie uczenia prowadziłoby do zbiegania do nieoptymalnego 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 stanów, na podstawie których obliczany jest krok uczący. Drugim sposobem na zwiększenie niezależności i różnorodności stanów jest jednoczesna nauka na podstawie wielu niezależnych pojazdów. Pojazdy inicjalizowane w sposób losowy prowadzą do lepszej eksploracji przestrzeni stanów. Wyniki przeprowadzonych eksperymentów wskazują na znaczącą poprawę wyników uczenia już przy pięciu symulowanych jednocześnie pojazdach. Eksperymenty przeprowadziłem symulując jednocześnie 100 pojazdów.

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[8]. 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 obserwowanych nagród. Żeby przeciwdziałać eksplodującym gradientom korzystam z funkcji kosztu $smooth\_L1\_loss$[3], która zmniejsza wpływ skrajnych pomiarów na całkowity gradient kroku uczącego.

$$smooth_{L_1}(x) = \begin{cases} 0.5x^2 & dla |x| < 1 \\ |x| - 0.5 & dla |x| \geq 1 \end{cases} \tag{18}$$

Podobnie jak w Atari DQN[6] korzystam z optymalizatora $RMSprop$ zaproponowanego przez G. Hintona[9] a po raz pierwszy opublikowanego w pracy na temat optymalizacji rekurencyjnych sieci neuronowych służących do generowania tekstu[5].

$$\mathbb{E}[g^2]_t = \beta \mathbb{E}[g^2]_{t-1} + (1-\beta)(\frac{\delta L(x)}{\delta w})^2 \tag{19}$$

$$w_t = w_{t-1} - \frac{\alpha}{\sqrt{\mathbb{E}[g^2]_t}} \frac{\delta L(x)}{\delta w} \tag{20}$$

5.2 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 stan otrzymany ze środowiska, przekształca go na wektor o wymiarze $width$ a następnie oblicza nieliniową funkcję aktywacji $ReLu$[2] na wektorze wyjściowym. Kolejne $depth$ funkcji liniowych o tej samej szerokości $width$ przekształca wektor w ten sam sposób co pierwsza funkcja. Ostatnia funkcja liniowa przekształca wektor o wymiarze $width$ na wektor wyjściowy o wymiarze odpowiadającym wielkości przestrzeni akcji $|A|$. Wartości wektora wyjściowego reprezentują wynik funkcji $Q_\pi(*|s)$. Korzystam z algorytmów uczenia maszynowego zaimplementowanych w bibiliotece PyTorch[7].

5.3 Algorytm

Algorithm 1: Głębokie uczenie Q-network z wykorzystaniem pamięci

Inicjalizacja parametrów θ funkcji wartości akcji Q, $\theta_f = \theta$
Inizjalizacja środowiska oraz otrzymanie pierwszego stanus_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 i 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

6 System do przeprowadzania eksperymentów

Chcąc przetestować w jaki sposób algorytm uczenia 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 pomiędzy wieloma maszynami. Obie maszyny posiadały karty graficzne Nvidia RTX 2080 TI oraz 32 GB RAMu. Jedna maszyna miała Intel i7-6850K, druga Ryzen 7 3900X.

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 procesorów kart graficznych.

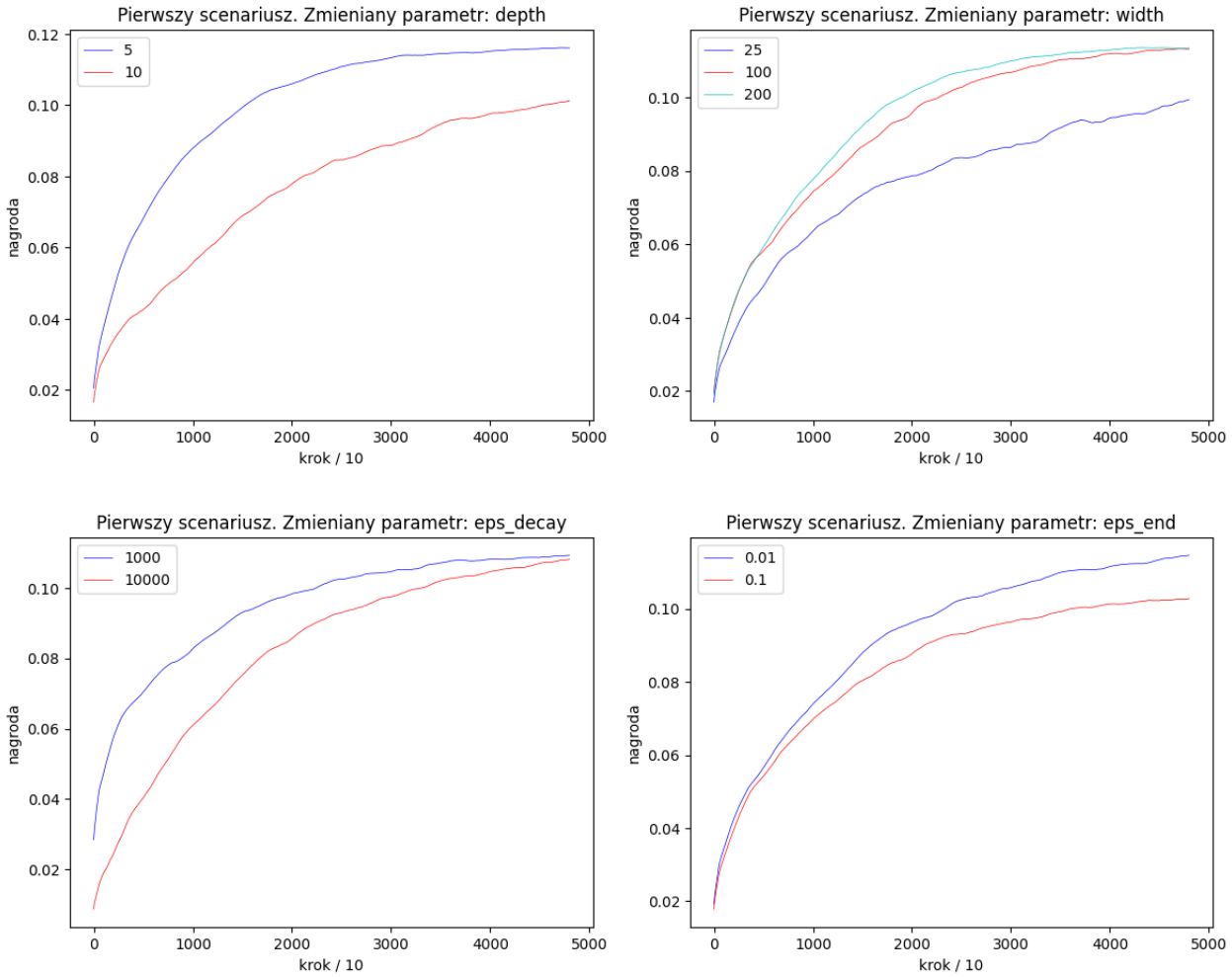
6.3 Synchronizacja wyników

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 i wyniki treningu

Chcąc dokładnie przetestować możliwości algorytmu zdecydowałem się na zastosowanie metodologii przeszukiwania siatki, co pozwoliło mi sprawdzić wpływ wszystkich parametrów na wyniki uczenia. W każdym z trzech scenariuszy sprawdzałem inne kombinacje parametrów. Wadą takiego podejścia okazał się być czas wymagany do zakończenia wszystkich eksperymentów. W drugim i trzecim scenariuszu długość trwania eksperymentów wynosi milion kroków aktualizacji środowiska. W pierwszym scenariuszu agent jest w stanie nauczyć się dobrej strategii już po 50 tysiącach kroku, po którym kończyłem uczenie. Na wykresach przedstawiam wpływ poszczególnych parametrów na przebieg uczenia. Wyniki eksperymentów są dzielone na grupy na podstawie wartości badanego parametru. Wykres przedstawia różnymi kolorami średnią nagrodę otrzymaną w każdym kroku przez wszystkie eksperymenty z danej grupy. Nagrody otrzymywane podczas treningu są uśredniane pomiędzy zapisaniem ich do wyników. Nagrody są uśredniane przez 10 lub 100 kroków aktualizacji środowiska.

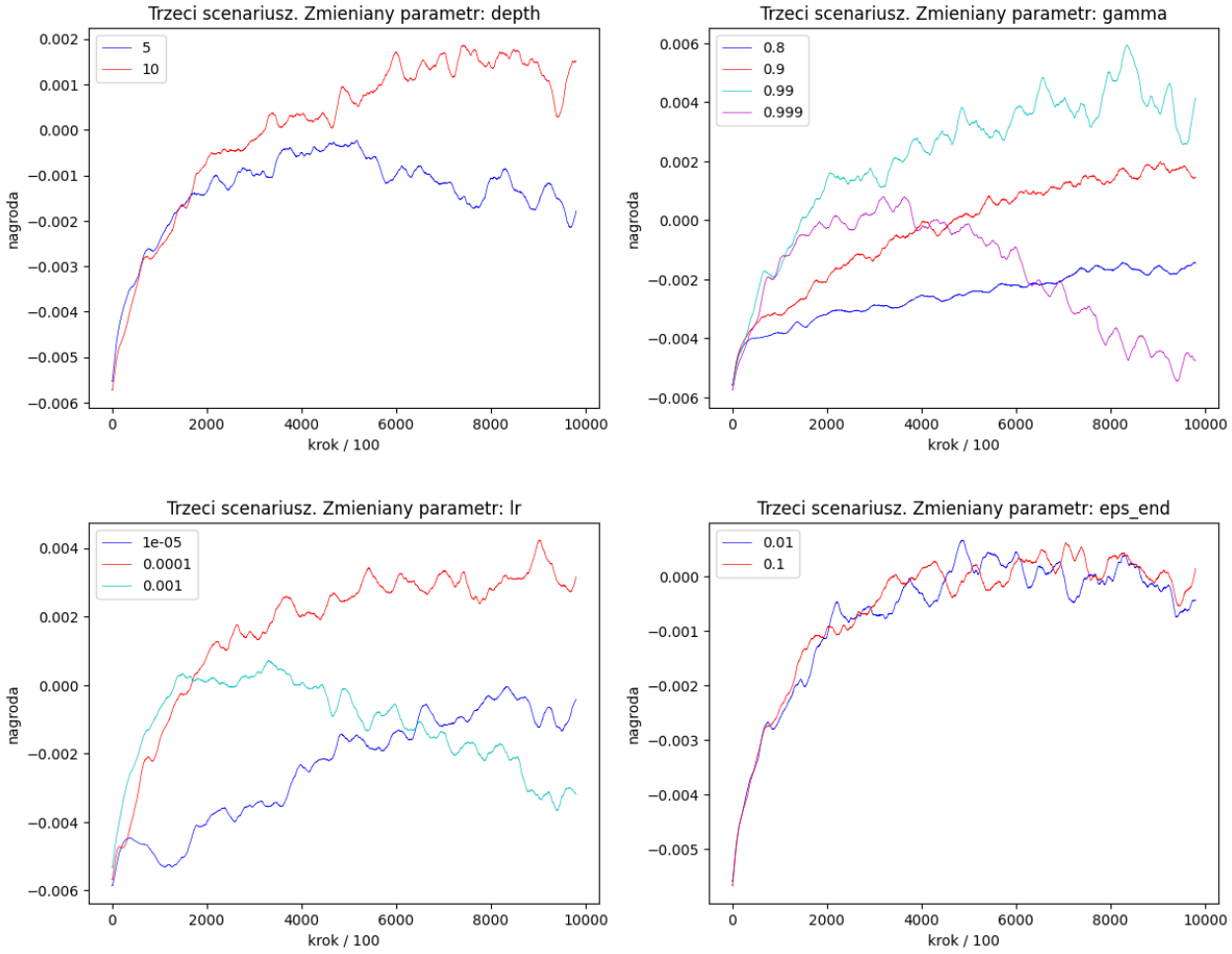
7.1 Pierwszy scenariusz



Rysunek 2: Wpływ parametrów na zdobywaną nagrodę w pierwszym środowisku.

Pierwszy scenariusz polegający na zatrzymywaniu pojazdu jest najłatwiejszym środowiskiem do nauczenia. Przestrzeń stanu jest najmniejsza oraz zauważalnie lepsze wyniki osiągały płytkie i szerokie sieci neuronowe. Parametry kontrolujące sposób eksploracji wpływają na wynik treningu zgodnie z oczekiwaniami. Większy *eps_end* powoduje częstsze wybieranie losowej akcji co zaniża zdobywaną nagrodę agenta dobrze dopasowanego. Mniejsza wartość *eps_decay* pozwala na lepsze dopasowanie agenta kosztem ilości kroków eksploracyjnych. W dłuższym treningu lepsza eksploracja na początku uczenia mogłaby zaowocować lepszym dopasowaniem. W trudniejszych środowiskach dłuższa eksploracja zwiększonym parametrem *eps_decay* może przynieść lepsze wyniki.

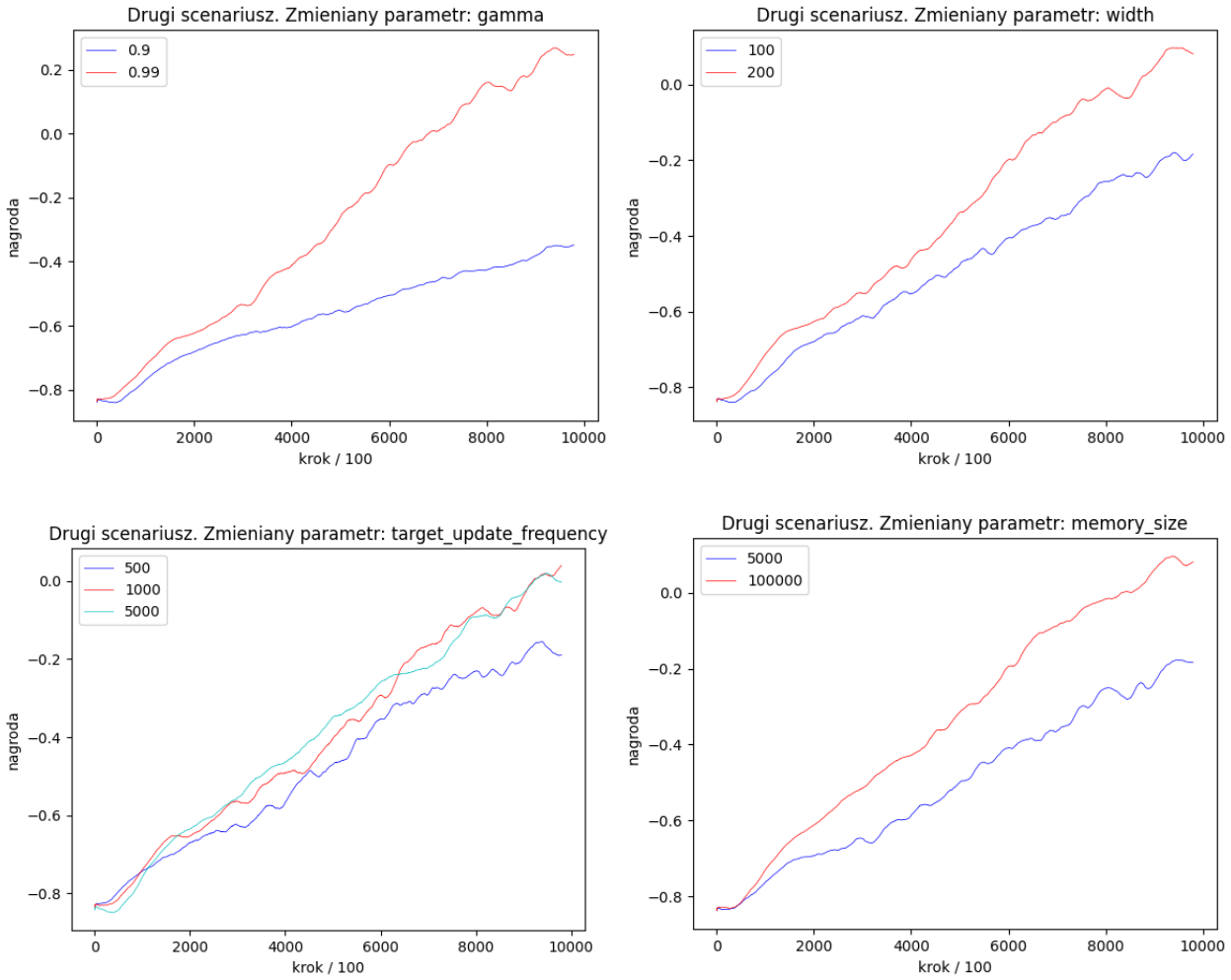
7.2 Trzeci scenariusz



Rysunek 3: Wpływ parametrów na zdobywaną nagrodę w trzecim środowisku.

Drugi i trzeci scenariusz jest trudniejszy od pierwszego gdyż przestrzeń stanu jest większa oraz epizody są znacząco dłuższe. Zwiększenie głębokości sieci neuronowej przyniosło lepsze rezultaty. Wyraźnie widać, że parametr *gamma* nie powinien przyjmować skrajnych wartości bliskich jeden, gdyż dochodzi do tragicznego zapominania. Mniejsza wartość tego parametru nie wpływa tak negatywnie na wydajność nauki jak za duża wartość. Krok uczący o wielkości $lr = 0.0001$ okazał się być najlepszy i został zastosowany w pozostałych środowiskach.

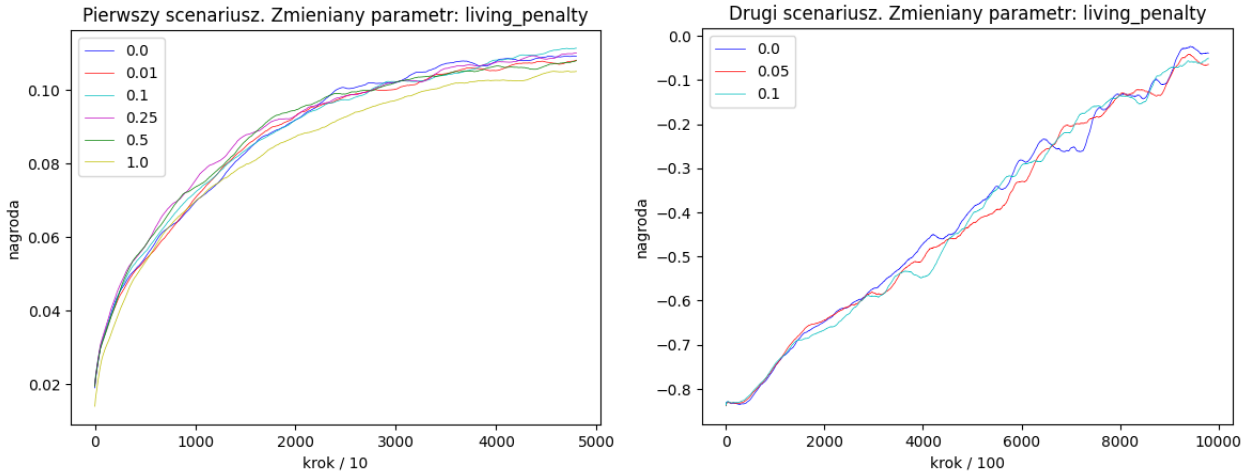
7.3 Drugi scenariusz



Rysunek 4: Wpływ parametrów na zdobywaną nagrodę w drugim środowisku.

Nietypowy, liniowy przyrost zdobywanej przez agenta nagrody wynika ze sposobu obliczania nagrody, która odpowiada zmianie odległości, a nie szybkości na którą agent ma bezpośredni wpływ. Zwiększenie pamięci poprawiło osiągane wyniki oraz stabilność treningu. Zwiększenie parametru *gamma* kontrolującego planowanie również poprawiło wyniki. Pojawia się też zauważalny wpływ częstotliwości aktualizacji wag sieci na podstawie której obliczana jest maksymalna wartość akcji z następnego kroku.

Parametr living penalty, niewielka kara dawana za każdy krok, w teorii powinien zmuszać agenta do szukania bardziej ryzykownych strategii które prowadzą do szybszego zakończenia symulacji. Nagrody prezentowane na wykresach były zapisywane przed aplikacją living penalty.

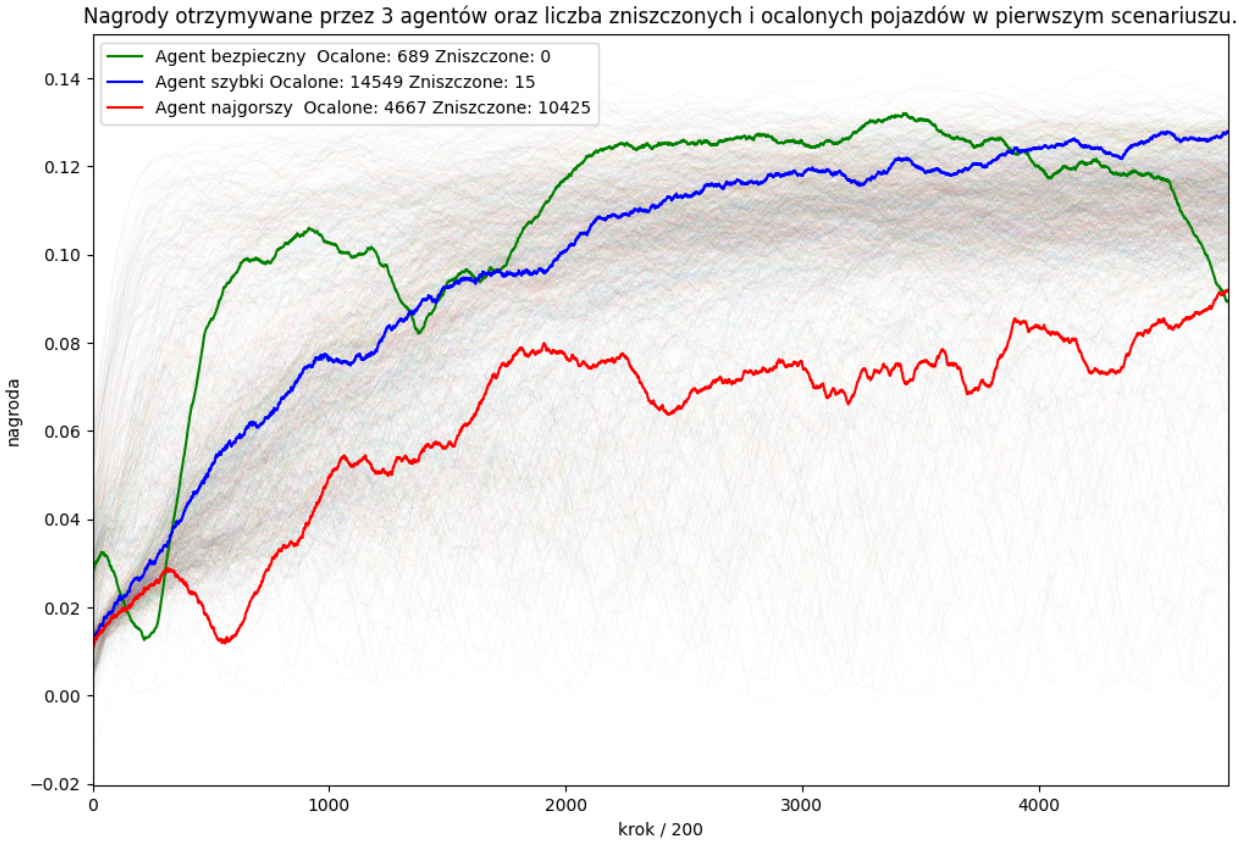


Rysunek 5: Wpływ parametru living penalty na proces uczenia.

8 Wyniki testów

Na koniec treningu agenta a każdego eksperymentu zapisywane były wagi wytrenowanej sieci. Testowanie polegało na zliczaniu pojazdów, które zakańczały symulację z nagrodą (Ocalone) bądź karą (Zniszczone) w przeciągu 100.000 kroków aktualizacji środowiska. Podczas testowania, podobnie jak podczas treningu, agent kontrolował 100 niezależnych pojazdów jednocześnie. Agent po osiągnięciu stanu terminalnego zapisywał informację o tym, czy został zniszczony czy ocalony, a następnie resetował swój stan do stanu początkowego aby kontynuować symulację. Na poniższych wykresach kolorem czerwonym zaznaczam najgorszego agenta, który zniszczył największą liczbę pojazdów, kolorem niebieskim najszybszego agenta, który ocalił największą ilość statków, oraz kolorem zielonym najbezpieczniejszego agenta, który miał najmniejszą ilość zniszczonych pojazdów. Wyniki otrzymywanej podczas treningu nagrody przez trzech wybranych agentów rysuję na tle wszystkich innych trenowanych agentów. Parametry różniące tych trzech agentów są wypisane w tabelce znajdującej się pod wykresami. Linki znajdujące się w ostatniej kolumnie prowadzą do krótkich nagrań prezentujących zachowanie agenta. Dla czytelności nagrań ilość pojazdów jest zmniejszona.

8.1 Pierwszy scenariusz



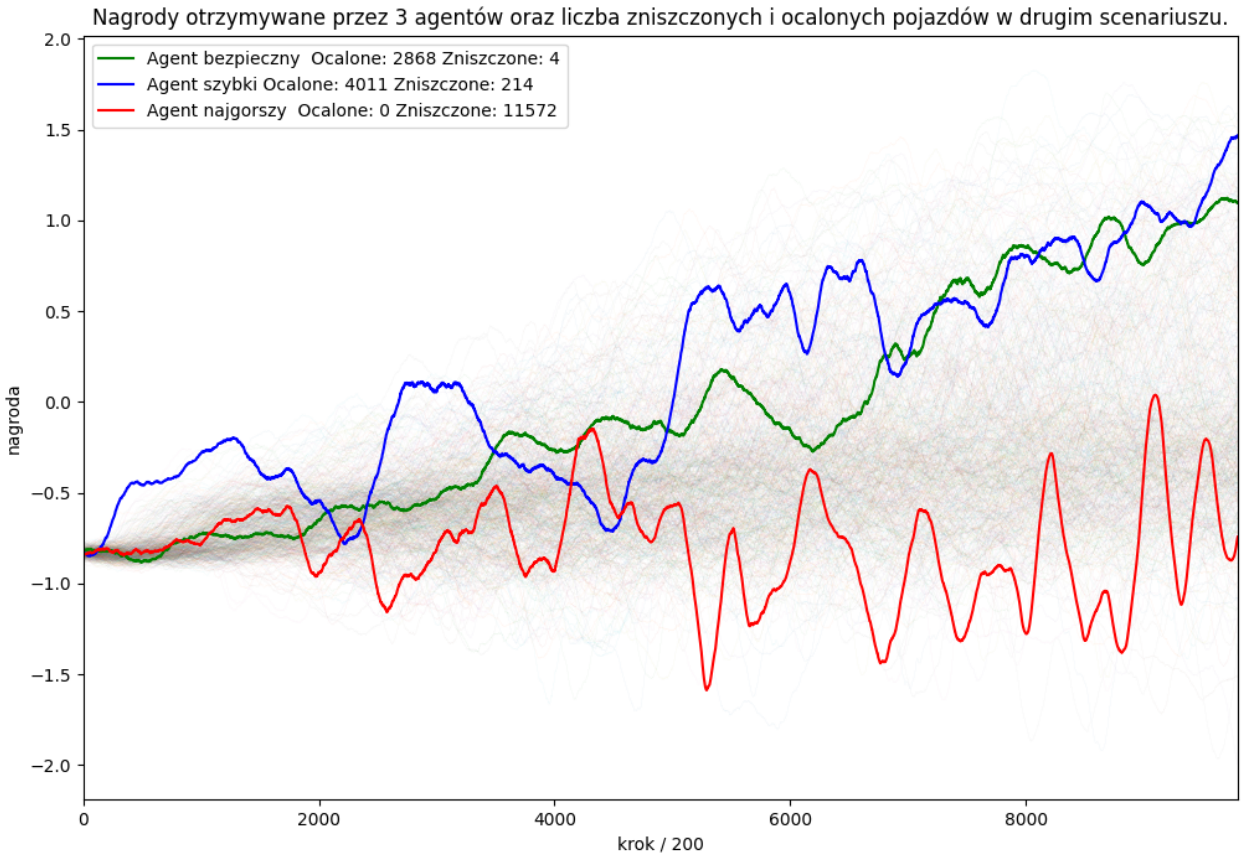
Rysunek 6: Wyniki testu w pierwszym środowisku.

Agent	living penalty	depth	width	eps end	eps decay	batch size	video
Najgorszy	1.0	10	25	0.1	10000	128	link
Szybki	0.01	5	200	0.01	10000	32	link
Bezpieczny	0.25	5	25	0.01	1000	32	link

Tablica 1: Parametry trzech wybranych agentów z pierwszego środowiska.

Bezpieczny agent pomimo bardzo dobrych wyników w zdobywanej nagrodzie podczas treningu nie nauczył się optymalnej strategii. Na wykresie (5) widać, że skrajnie wysoka wartość parametru living penalty powodowała zmniejszenie otrzymywanej ze środowiska nagrody. W pierwszym scenariuszu, gdzie maksymalna nagroda do uzyskani w pojedynczym kroku wynosi jeden, taka wartość powodowała obniżenie nagród do samych wartości ujemnych. W wyniku tego agentowi niebezpiecznemu nie zależało na odratowywaniu pojazdów które potrzebowały więcej kroków do ocalenia. Zamiast tego mniejszą karę otrzymywał przez szybkie niszczenie takich pojazdu.

8.2 Drugi scenariusz



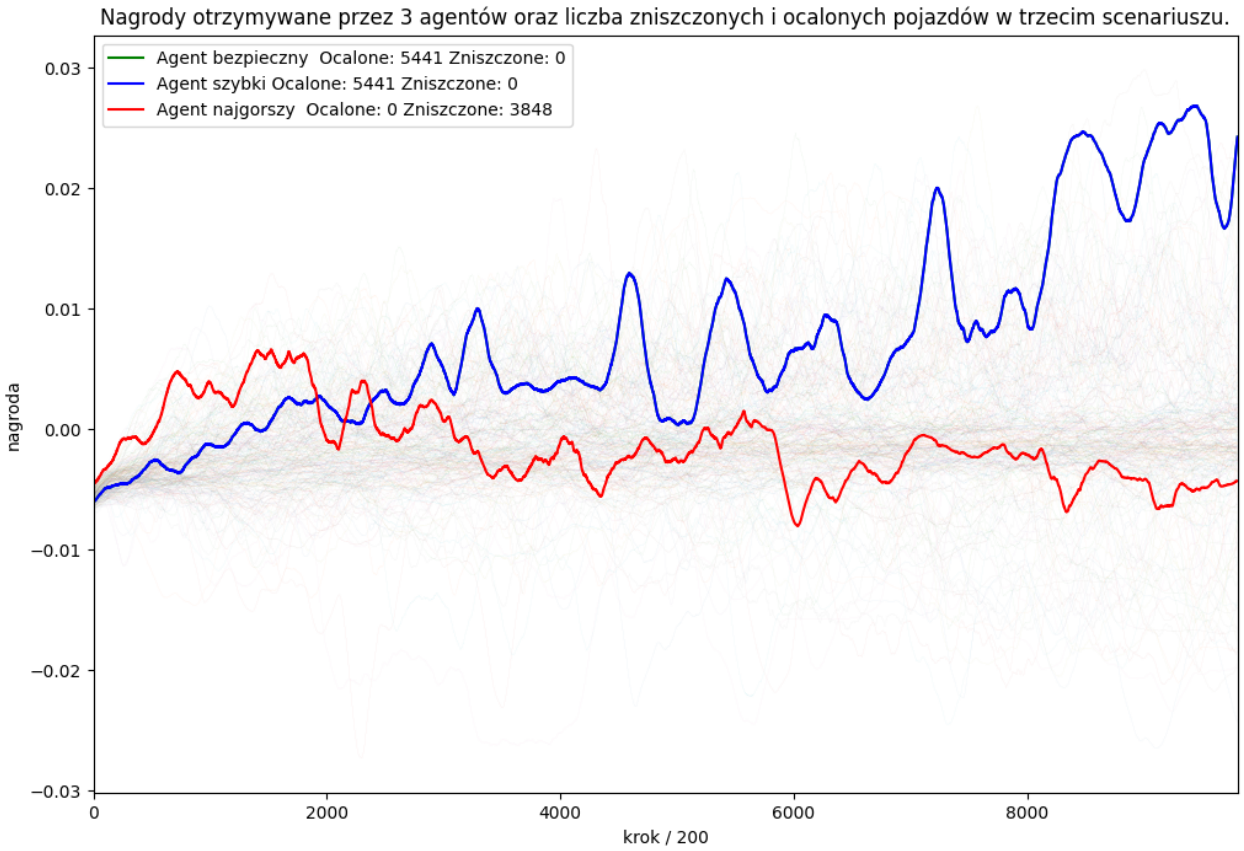
Rysunek 7: Wyniki testu w drugim środowisku.

Agent	depth	width	eps end	memory size	target update	living penalty	video
Najgorszy	10	100	0.01	5000	500	0	link
Szybki	10	200	0.1	100000	1000	0.1	link
Bezpieczny	20	200	0.1	100000	5000	0.05	link

Tablica 2: Parametry trzech wybranych agentów z pierwszego środowiska.

W przeciwieństwie do wyników pierwszego scenariusza nic żaden parametr nie wskazuje wyraźnie na przyczynę tak złego dopasowania najgorszego agenta. Gdyby była to kwestia funkcji nagrody która uniemożliwia naukę strategii to pozostali agenci też mieliby problem z nauką.

8.3 Trzeci scenariusz



Rysunek 8: Wyniki testu w trzecim środowisku. Agent szybki jest jednocześnie bezpiecznym.

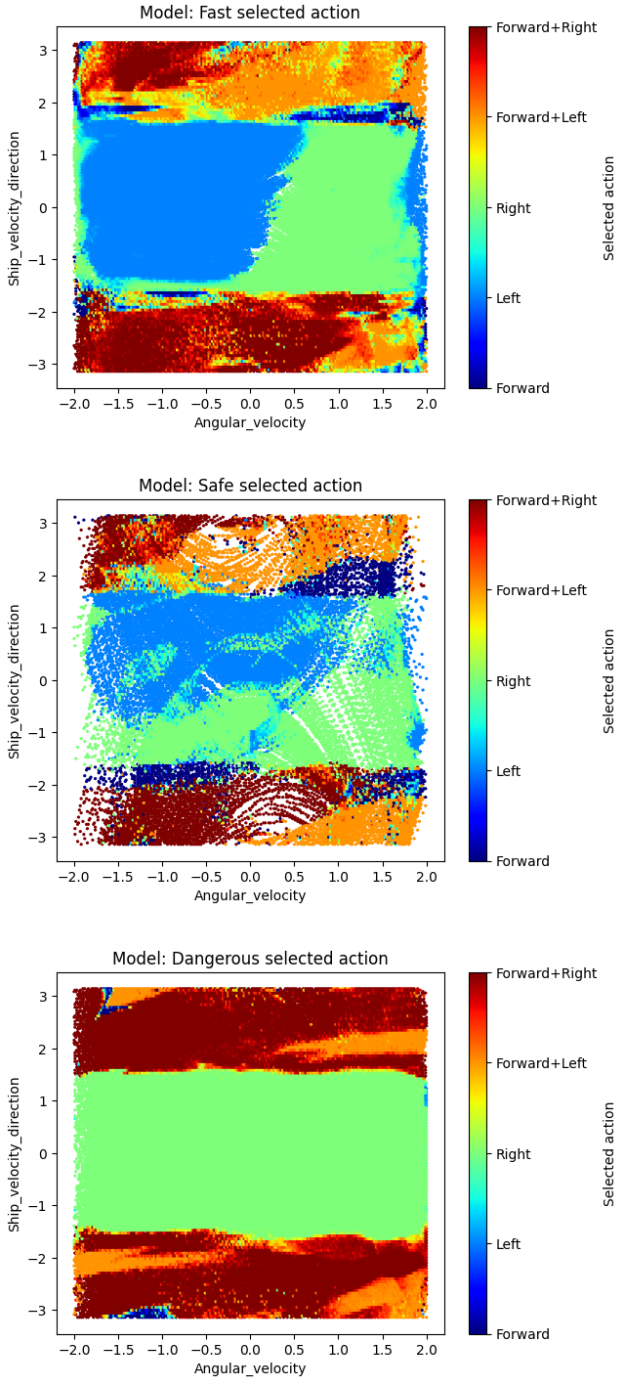
Agent	eps end	eps decay	memory size	lr	video
Najgorszy	0.1	100000	10000	0.001	link
Szybki i bezpieczny	0.01	1000000	100000	0.0001	link

Tablica 3: Parametry trzech wybranych agentów z pierwszego środowiska.

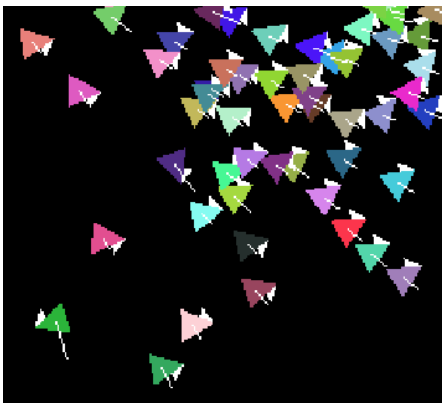
Agent który na samym początku uczenia zapowiadał się na najlepszego w okolicy kroku 20.000 zmienił trajektorię nauki która zakończyła się najgorszym wynikiem. Porównując wyniki tego testu z wynikami poszczególnych parametrów podczas treningu powodem tak tragicznego spadku wydajności może być za duża wartość parametru lr.

9 Analiza trzech agentów z pierwszego środowiska

Rysunek 9: Akcje wybrane podczas testów.

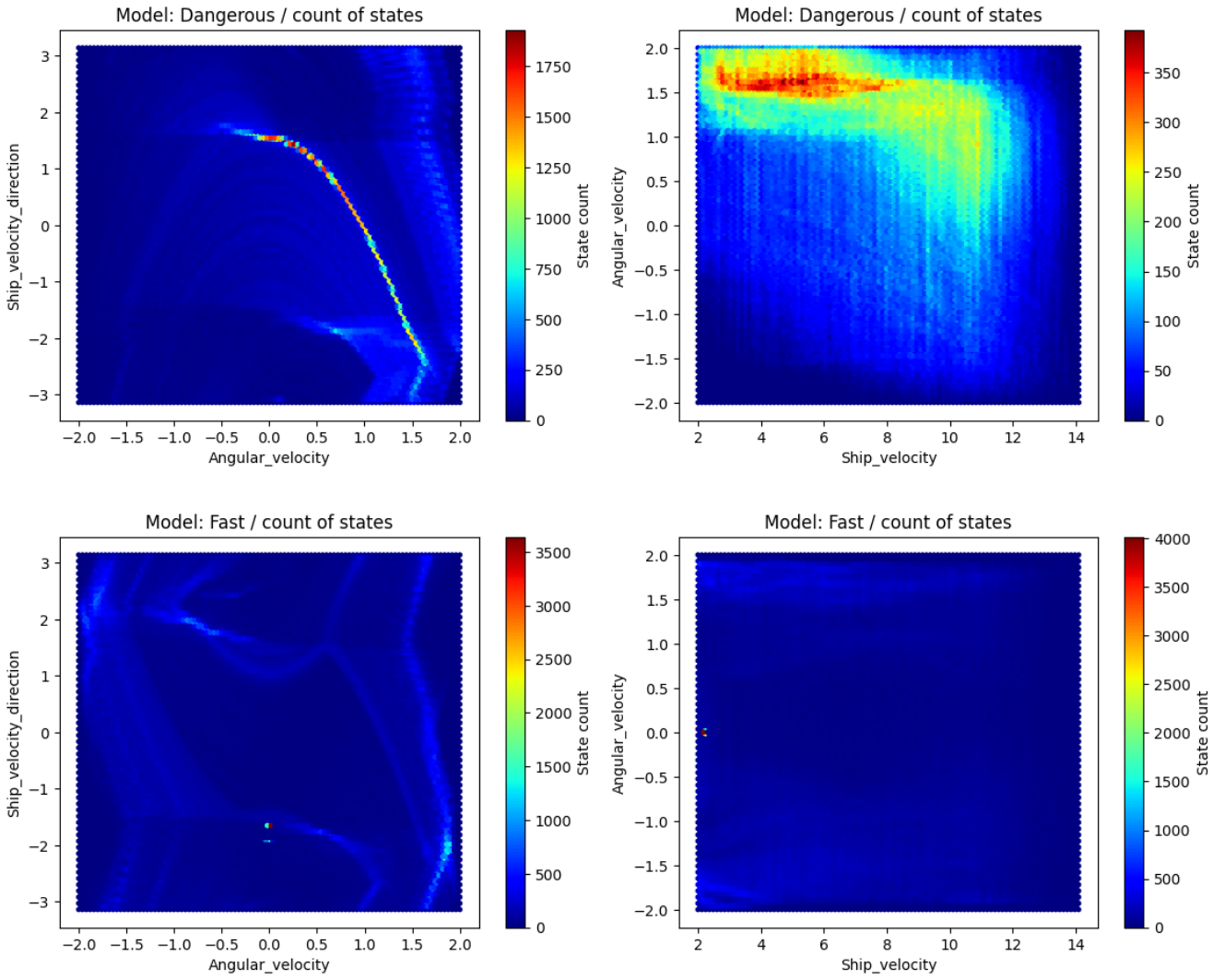


Na wykresach (9) przedstawione są wybrane przez agentów akcje podczas testowania. Stan przedstawiony jest na osiach XY, akcja wybrana w danym stanie jest zaznaczona odpowiednim kolorem. Oś X odpowiada prędkości obrotowej a oś Y kątowi pomiędzy kierunkiem statku a wektorem prędkości oraz prędkości kątowej. Statek zwalnia najefektywniej w momencie kiedy pojazd zwrócony jest w przeciwną stronę co prędkość, czyli wartość na osi Y jest bliska $\pm\pi$ *rad*. Nauczoną funkcję wartości stanu można zobaczyć na wykresach (13). Agent bezpieczny ma na wykresie dużo białych plam co odpowiada braku doświadczeń z tego stanu. Wynika to z tego, że pomimo takiej samej ilości kroków uczących i otrzymanych obserwacji agent wybierał negującą się akcje uruchamiając na przemian prawy i lewy silnik utrzymując stałą prędkość i rotację pojazdu jak widać na załączonym poniżej rysunku (10). Agent jest "bezpieczny"gdyż większość pojazdów nigdy nie kończy symulacji.

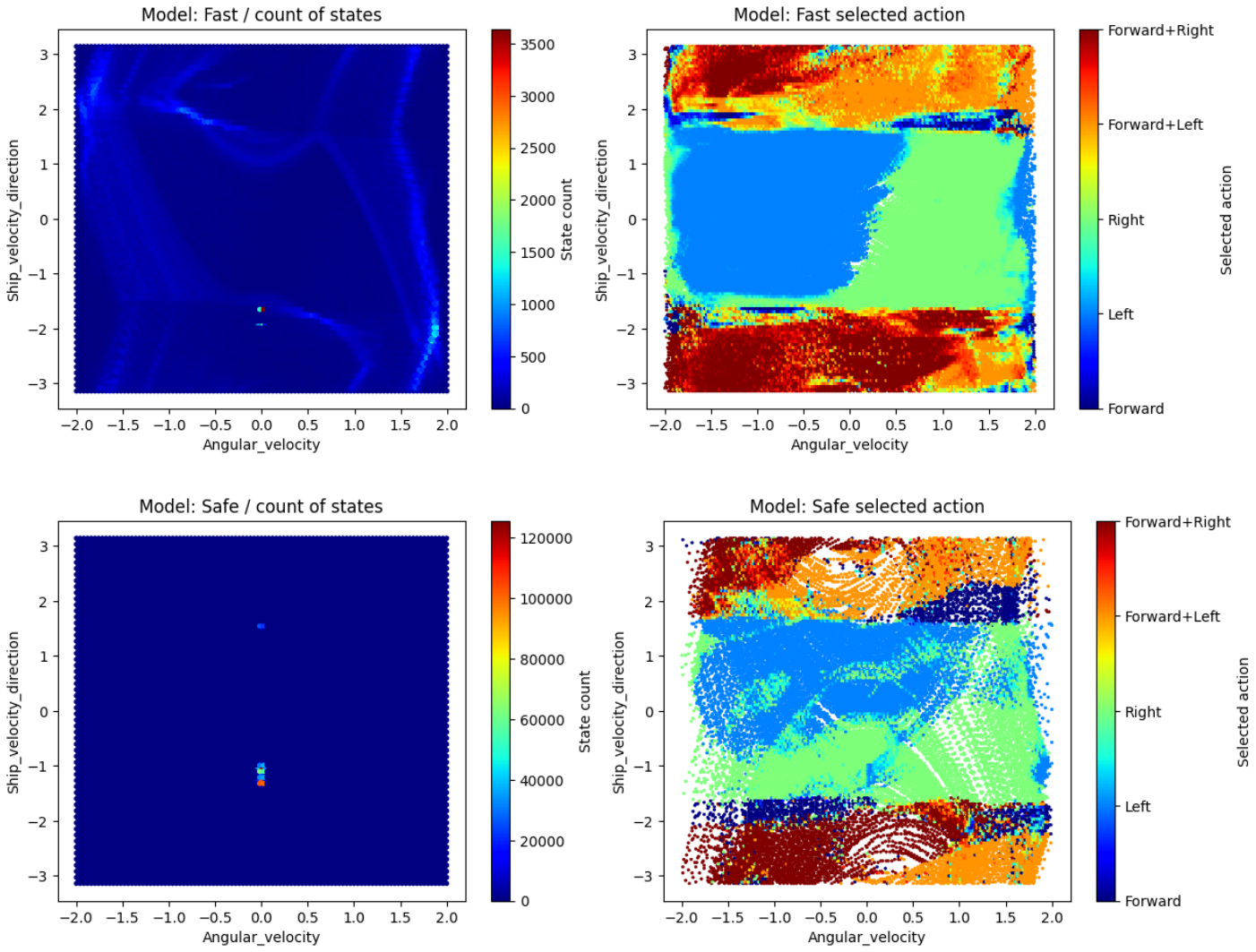


Rysunek 10: Duża część populacji pojazdów zablokowana w zapętlonych akcjach przez strategię bezpiecznego agenta

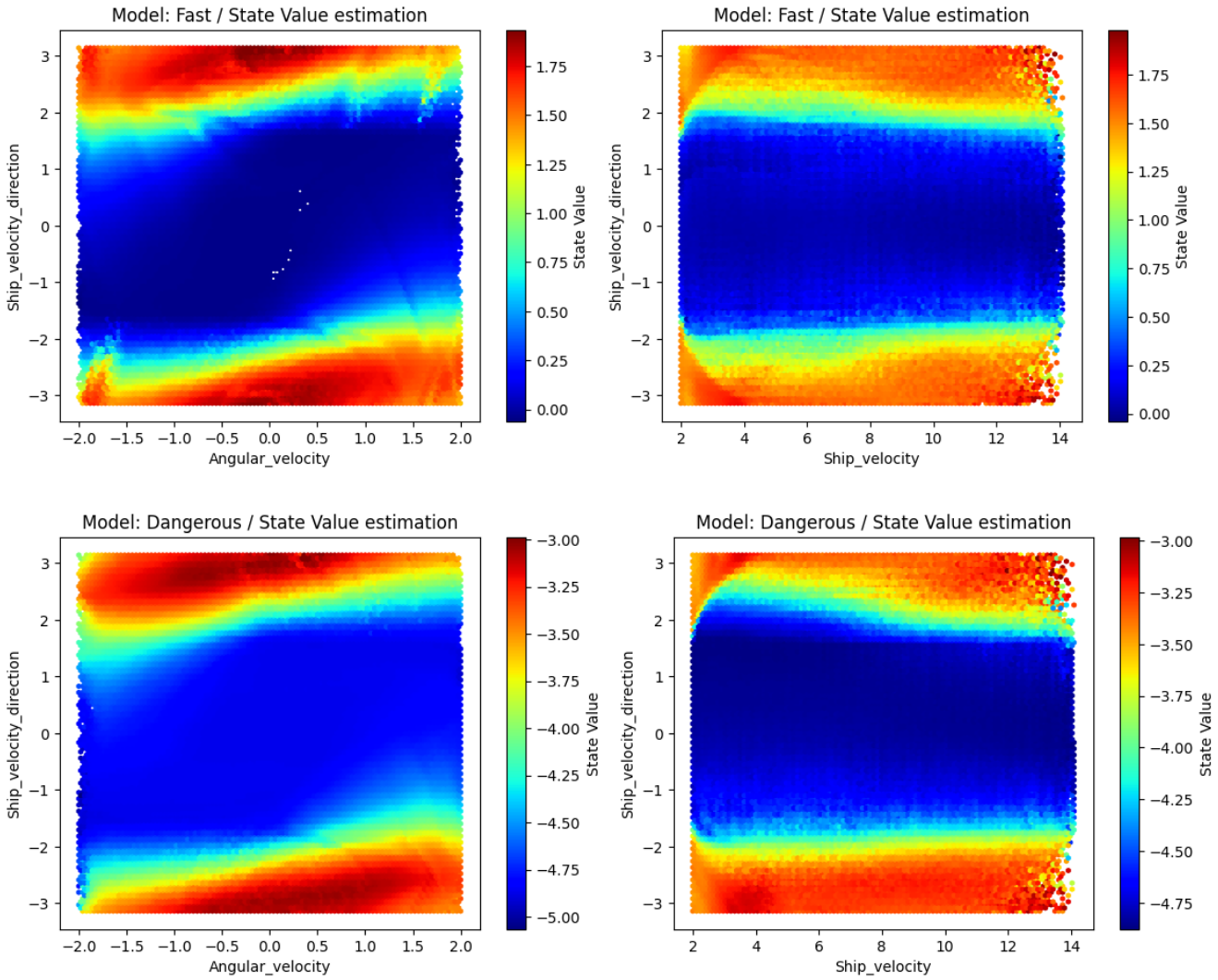
Największą różnicę widać w środkowej części wykresów gdzie agent zwrócony jest przodem do kierunku lotu, to jest nie może jeszcze hamować. Agent niebezpieczny w ogóle nie korzysta z akcji skręcania w lewo. Wszystkie pojazdy przez większość stanów skręcają w prawo, co powoduje znaczącą zmianę dystrybucji obserwowanych stanów co wpływa na dalszą naukę agenta.



Rysunek 11: różnica w dystrybucji stanów obserwowanych podczas testów przez agenta niebezpiecznego i szybkiego. Chcę sprawdzić czym jest ten zielono czerwony punkt w dolnej części lewego dolnego wykresu



Rysunek 12: W podobnych miejscach jest czerwona plamka powodująca zapętlenie pojazdu w nieskończonym cyklu prawo/lewo. Chciałbym czerwonym markerem zaznaczyć dokładne miejsca w strategiach które powodują takie zachowanie



Rysunek 13: Wartości stanów wg szybkiego i niebezpiecznego są kształtem do siebie podobne, chociaż agent niebezpieczny przez living penalty przewiduje najlepszą oczekiwaną nagrodę -3

Literatura

[1] boost c++ libraries.

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

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

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

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

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

[7] 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.

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

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

Spis rysunków

Spis tablic