AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI, INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ

Projekt dyplomowy

Środowisko wieloagentowe uczenia przez wzmacnianie inspirowane ekosystemami naturalnymi
Multi-Agent Reinforcement Learning environment inspired by natural ecosystems

Autor: *Bartłomiej Jan Tarcholik*
Kierunek studiów: *Informatyka i Systemy Inteligentne*
Opiekun pracy: *dr hab. Adrian Horzyk, prof. AGH*

Kraków, 2024

Serdeczne podziękowania mojemu promotorowi
dr hab. Adrianowi Horzykowi, prof. AGH za wsparcie
i nieocenioną pomoc w przygotowaniu pracy.

Spis treści

1. Wstęp

Począwszy od pierwszych prób zastosowania komputerowej symulacji zachowania organizmów żywych, jednym z fundamentalnych wyzwań było stworzenie modelu maksymalnie zbliżonego do poczynań inteligentnych. Najpopularniejsza jeszcze niedawno metoda wyszukiwania optymalnej polityki zachowania opierała się w pełni na osobie programisty, który musiał zaprojektować i na sztywno wdrożyć ową politykę w kod (*ang. hardcoding*). Przez długi czas takie rozwiązanie było najszybszym i najskuteczniejszym sposobem tworzenia symulacji zachowania, opierało się bowiem na dogłębnym zrozumieniu środowiska przez jego autora, co eliminowało wszelkie wątpliwości dotyczące poprawności zachowań podejmowanych przez symulowany organizm. Biorąc pod uwagę ówczesne zasoby sprzętowe, podejście takie było niejednokrotnie jedyną opcją. Technologia wystarczająca dla jasno określonych symulacji okazywała się bowiem niewystarczająca w przypadku prób automatycznego znalezienia prawidłowej polityki zachowania. Sztywne programowanie zachowania obarczone jest jednak poważną wadą w postaci limitu wiedzy oraz zrozumienia problemu przez programistę. Przed twórcą polityki zachowania stało skomplikowane wyzwanie pełnego zrozumienia środowiska, przewidzenie wszystkich możliwości i znalezienie najlepszych lub najwierniejszych symulacji akcji dla każdej sytuacji. Wspomniane ograniczenia przekładały się bezpośrednio na trudności w prawidłowej implementacji ze względu na złożoność analizy środowiska. Oprócz symulacji, sztywne programowanie polityki zachowania znalazło również zastosowanie w grach komputerowych, gdzie jest używane do tworzenia „sztucznej inteligencji", czyli jednostek sterowanych przez komputer – taktyki wrogów i sojuszników w trakcie walki czy wierne odwzorowanie życia codziennego w przypadku postaci tła.

Od pewnego czasu, wraz ze wzrostem mocy obliczeniowej komputerów, coraz bardziej popularnym sposobem modelowania skomplikowanych problemów staje się uczenie maszynowe. Pozwala ono uniknąć długiej i wnikliwej analizy, która byłaby potrzebna w przypadku wcześniejszych rozwiązań. Dzięki metodom uczenia maszynowego, zadania takie jak rozpoznawanie chorób z opisu symptomów, przewidywanie wartości domu, czy dokładne śledzenie poruszającego się obiektu, stały się możliwe do rozwiązania dużo mniejszym kosztem. Uczenie modelu konkretnych zasad pozwala bowiem na automatyczne znalezienie kluczowych dla problemu parametrów i ich wpływu na rozwiązanie, a wszystko to z minimalnym wkładem ludzkim w porównaniu z konwencjonalnymi metodami.

W niniejszej pracy autor skupia się przede wszystkim na uczeniu przez wzmacniane (ang. *reinforcement learning, RL*), typie uczenia maszynowego wyjątkowo przydatnego w symulacji zachowania agentów. Pozwala ono na znalezienie odpowiedniej polityki zachowania w sposób w pełni automatyczny poprzez obserwacje środowiska, wykonywanie pseudolosowych akcji,

ulepszanie zdolności ich wybierania w trakcie szkolenia i tworzenie ciągu przyczynowo skutkowego prowadzącego do uzyskania jak najlepszych efektów reprezentowanych przez nagrody i kary. Te cechy umożliwiają zamodelowanie jednego lub wielu agentów, którzy sami znajdą najlepsze zachowanie dla danego środowiska.

Bezpośrednim celem pracy jest implementacja środowiska wieloagentowego zgodnego ze specyfikacjami biblioteki PettingZoo, które inspirowane jest ekosystemami naturalnymi. W środowisku występuje jeden typ agenta reprezentujący generyczną rybę wszystkożerną. Agenci poruszają się w trójwymiarowej przestrzeni dyskretnej reprezentującej zbiornik wodny z przestrzennym dnem, które może zostać wczytane z pliku graficznego. Dla każdego agenta dostępne są akcje poruszania się i żywienia wykonywane na podstawie obserwacji otrzymywanych ze środowiska. Obserwacje te są inspirowane rozumieniem przestrzennym i wzrokowym rzeczywistych organizmów żyjących w naturalnych środowiskach podwodnych. Celem każdego agenta jest zmaksymalizowanie nagrody kumulacyjnej, co wymaga efektywnego zdobywania punktów pożywienia poprzez żywienie się roślinami lub innymi agentami.

Praca ma na celu położenie podwalin pod większy projekt, który umożliwi stworzenie wiernej symulacji istniejących środowisk wodnych i pozwoli na badania nad zachowaniem organizmów w nich żyjących. Miałoby to niebagatelne znaczenie w przypadku analizy zmian składu gatunkowego ekosystemów lub prognozowania ich przyszłości na podstawie aktualnego stanu.

Pierwszym zastosowaniem środowiska, które jest efektem tej pracy, może być użycie go jako podstawy pod symulację rzeczywistego ekosystemu z agentami walczącymi o przetrwanie. Dzięki możliwości modyfikacji parametrów środowiska, użytkownik może osiągnąć różnorodne efekty w procesie szkolenia. Pozwala to na zmianę zachowań agentów pod kątem utrzymania się przy życiu, prowadzenia ataków na inne jednostki, czy też wykorzystywania różnorodnych strategii w poszukiwaniu pożywienia.

Drugim aspektem użyteczności projektu jest jego wkład w rozwój sztucznej inteligencji w dziedzinie gier komputerowych. W oparciu o pracę możliwe jest badanie opcji automatycznego znajdowania przez agentów polityki prowadzącej do osiągnięcia najkorzystniejszych dla siebie wyników. Pozwala to stworzyć agentów sterowanych komputerowo, którzy wykraczają poza szablonowe metody zachowania stworzone przez programistę.

Praca składa się ze wstępu liczonego jako rozdział pierwszy, rozdziału drugiego poświęconego części teoretycznej, rozdziału trzeciego skupiającego się na opisie implementacji środowiska oraz rozdziału czwartego dokumentującego przykładowe sposoby i wyniki szkolenia w środowisku oraz trudności z nim związane. Praca jest zakończona podsumowaniem oraz bibliografią.

W rozdziale drugim autor przybliża szereg pojęć i definicji teoretycznych związanych z prezentowaną pracą, takich jak uczenie maszynowe z podziałem na uczenie nadzorowane

i nienadzorowane czy też kluczowe dla projektu uczenie przez wzmacnianie. Przedstawione też zostają podstawowe definicje umożliwiające zrozumienie RL, omówione podstawy RL z podkreśleniem zarówno wad, jak i zalet uczenia przez wzmacnianie względem innych rodzajów uczenia maszynowego. Omówione zostają algorytmy stosowane w RL oraz środowiska uczenia przez wzmacnianie w Pythonie – biblioteki i standardy środowisk Gym, Gymnasium i PettingZoo. Ostatnie akapity rozdziału drugiego poświęcone są opisowi środowiska w standardzie Gym z wyszczególnieniem metod mających wpływ na działanie środowiska oraz sposobom zapobiegania akcjom nielegalnym.

Rozdział trzeci stanowi właściwy trzon pracy, w którym przedstawiona jest implementacja wieloagentowego środowiska w standardzie PettingZoo, opisane są jego kluczowe elementy i cechy, a także przedstawione fragmenty kodu, które opisują działanie środowiska. Przybliżone są zasady funkcjonowania agentów – dostępne akcje, obserwacje otrzymywane ze środowiska, a także nagrody i kary, które kierują algorytm do uzyskania lepszych wyników.

W rozdziale czwartym autor przedstawia przykładowy sposób szkolenia agentów przy użyciu biblioteki Stable-Baselines3 oraz wrapperów SuperSuit, opisuje wyniki szkolenia i wpływ zmian parametrów szkolenia i środowiska na finalny wynik, a także prezentuje i wyciąga wnioski z wykresów przedstawiających proces szkolenia. Porównuje też najlepsze polityki oraz opisuje wpływ parametrów na wyniki.

Podsumowanie przedstawia streszczenie osiągniętych celów, zaimplementowanych części środowiska i ich zastosowań oraz opis wyników szkolenia. Autor nakreśla też możliwe ścieżki rozwoju projektu wraz z ich praktycznym zastosowaniem.

2. Uczenie maszynowe: kontekst ogólny i uczenie przez wzmacnianie

Chociaż termin „sztuczna inteligencja” można odnaleźć zarówno w publikacjach naukowych jak i literaturze popularnonaukowej, pojęcie to nie jest proste do zdefiniowania. Trudność ta może wynikać z faktu, iż nie istnieje właściwie jasna i spójna definicja samej „inteligencji’. Z całą pewnością jednak można stwierdzić, że sztuczna inteligencja stanowi dziedzinę wiedzy, w której celem i przedmiotem badań są maszyny rozwiązujące trudne i niespecyficzne zadania, przy wykonywaniu których człowiek nieraz potrzebowałby złożonego logicznie i zawiłego rozumowania [1]. Zainteresowanie zagadnieniami AI (ang. *Artificial Intelligence*) rośnie z roku na rok, dlatego już w tym momencie znalazła ona zastosowanie w wielu dziedzinach nauki i życia codziennego.

2.1 Uczenie maszynowe

Pojęciem bezpośrednio związanym ze sztuczną inteligencją jest uczenie maszynowe (ang. *Machine learning, ML*). Uczenie maszynowe prezentuje zdolność maszyny (systemu) do rozszerzania wiedzy i udoskonalania podejmowanych przez siebie decyzji na podstawie zdobywanych doświadczeń [2]. Można więc z całą pewnością przyjąć założenie, że w przypadku uczenia maszynowego szeroko rozumiana „maszyna” posiada wiedzę oraz reaguje w sposób przypominający zachowania człowieka, a nawet przewyższający jego efektywność. Za jednego z pierwszych badaczy tematu samouczących się programów uznaje się przedstawiciela firmy IBM, który w 1952 roku rozpoczął prace nad programem uczącym się gry w warcaby. Po dwóch latach program był w stanie pokonać graczy-amatorów, zaś w 1962 roku zwyciężył z ówczesnym mistrzem warcabów – Robertem Nealey’em. Od tego momentu sztuczna inteligencja, a w szczególności uczenie maszynowe, stały się obszarami intensywnych badań, podlegając wielokrotnym udoskonaleniom oraz ulepszeniom [3]. AI przetwarza dane, co umożliwia podejmowanie konkretnych decyzji czy też tworzenie prognoz. Algorytmy uczenia maszynowego umożliwiają AI uczenie się na podstawie zdobytych danych. Początkowo proces uczenia maszynowego polega na obserwacji danych (takich jak bezpośrednie doświadczenia czy też instrukcje), co powinno umożliwić dotarcie do ich wzorca, w celu wypracowania strategii pozwalającej na podejmowanie lepszych decyzji w oparciu o dostarczone przykłady [4]. W następnej części tego rozdziału, autor przedstawia kilka podstawowych modeli uczenia maszynowego.

Uczenie nadzorowane ma szczególne zastosowanie w przypadku zbioru oznaczonych danych. Prezentowane są pary danych wejściowych oraz wyjściowych, zaś dane wyjściowe mają przydzieloną oczekiwaną wartość. Warto użyć tego typu uczenia, gdy naszym celem jest np. nauczenie systemu odróżniania psów od kotów (w takim przypadku danymi wejściowymi byłyby obrazy psów i kotów, zaś oznaczeniami tych danych klasyfikacja każdego obrazu). Algorytm uczenia nadzorowanego szacuje wyniki danych treningowych w różnych iteracjach, aby następnie zostały one skorygowane przez obserwatora Proces nauki może zostać zakończony w momencie, gdy algorytm osiąga akceptowalną wydajność. Ten typ uczenia można podzielić na dwie kategorię: regresję oraz klasyfikację. Regresja ma zastosowanie w przypadku zagadnień, których wynikiem jest konkretna liczba lub ich zestaw, (np. prognozowanie cen domów na podstawie informacji takich jaki liczba pokoi oraz ich metraż). W przypadku klasyfikacji dane wyjściowe należą do określonego zbioru klas, dzięki czemu klasyfikacja posiada zastosowanie np. w przewidywaniu czy dana wiadomość e-mail jest spamem.

Uczenie nienadzorowane stosowane jest, gdy istnieją dane wejściowe, ale nie da się przedstawić odpowiadającej im zmiennej wyjściowej. System stara się odnaleźć wzorce i korelacje, wykorzystując do tego procesu wszelkie dostępne dane. Przykładem zastosowania tego rodzaju uczenia w praktyce mogą być badania rynkowe czy też mechanizmy bezpieczeństwa cybernetycznego. Samo-nadzorowane uczenie się (ang. *Self-supervised Learning*) to rodzaj uczenia nienadzorowanego, w którym występuje automatyczne etykietowanie danych treningowych poprzez znajdowanie i wykorzystanie korelacji między różnymi funkcjami wejściowymi, co zmusza sieć do nauki semantycznej reprezentacji danych.

Uczenie częściowo nadzorowane prezentuje rozwiązanie pośrednie między technikami uczenia nadzorowanego i nienadzorowanego. Ten typ uczenia to doskonały wybór w przypadku ogromnej ilości nieuporządkowanych danych i jedynie niewielkiej ilości danych oznaczonych. Początkowo podobne do siebie dane są grupowane przy pomocy uczenia nienadzorowanego. Następnie dane oznakowane analizowane są pod kątem właściwości korelatywnych, co wykorzystywane jest do oznaczania pozostałych nieoznakowanych danych [4,5].

2.2 Uczenie przez wzmacnianie

Uczenie przez wzmacnianie to typ uczenia maszynowego, w którym agent obserwuje otoczenie i na podstawie poczynionych obserwacji podejmuje decyzje. W dziedzinie RL agent osadzony w środowisku stara się ulepszyć swoje działania w odpowiedzi na różne sytuacje. W przeciwieństwie do uczenia nadzorowanego, algorytm nie otrzymuje jasno określonych danych i oczekiwanych wyników, co skłania agenta do stopniowego eksplorowania alternatywnych działań. W uczeniu nadzorowanym wynik sieci oparty jest na przedstawionych wcześniej niezależnych od siebie nawzajem danych, zaś w przypadku uczenia przez wzmacnianie podejmowane przez agenta decyzje oparte są na wcześniejszych doświadczeniach, między którymi występują konkretne zależności. Środowisko reaguje na akcje agenta, dostarczając mu obserwacje, jak również nagrody i kary. Istota działania algorytmów uczenia przez wzmacnianie polega na gromadzeniu wyników, stopniowo doskonaląc politykę zachowań w celu zwiększenia szansy uzyskania nagrody [6]. Uczenie przez wzmacnianie ma w świecie nauki niebagatelne znaczenie, gdyż umożliwia tworzenie modeli, które potrafią w szybki sposób dostosować się do zmieniających się warunków. Samodoskonalenie swojego działania przez metodę prób i błędów stwarza potencjał do przyspieszonego rozwoju inteligentnych maszyn, czyniąc RL opcją nad wyraz korzystną w porównaniu do innych form uczenia maszynowego [7].

2.2.1 Podstawowe definicje umożliwiające zrozumienie RL

Agent – element, wchodzący w interakcję ze środowiskiem, sterowany jest przez pewną politykę lub model

Środowisko – świat, z którym agent wchodzi w interakcje (jak np. poruszanie się po danym terenie), może nim być wszystko, co przetwarza i determinuje działanie agenta, np. gra.

Stan – zmienna określająca warunki środowiska; zbiór miejsc lub pozycji, do których agent może dotrzeć; można go przedstawić przy pomocy współrzędnych lub odpowiednich cyfr/liter.

Epizod – długość czasu (na przykład ilość kroków), po której resetowany jest stan środowiska. W przypadku środowiska symulującego grę w szachy może to być jedna partia [8,9].

Proces decyzyjny Markowa (MDP) – jego celem jest odwzorowanie optymalnych działań dla każdego stanu środowiska; w tym procesie decyzyjnym istotna jest jedynie teraźniejszość (informacje z przeszłości nie mają większego znaczenia), czyli przewidywanie następnego stanu jest zupełnie niezależne od stanów poprzednich. Reguły danego środowiska w tym procesie decyzyjnym pozostają niezmienne [9].

$$p(s', r \mid s, a) = P(S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a) \tag{2.1}$$

t – aktualny krok
S_t – stan w kroku t
A_t – akcja w kroku t
R_{t+1} – nagroda otrzymana za wykonanie A_t w stanie S_t
s – aktualny stan
s' – przyszły stan
r – otrzymana nagroda
a – wykonana akcja

Na podstawie poprzedniego stanu S_t oraz wykonanej przez agenta akcji A_t możliwe jest określenie rozkładu prawdopodobieństwa warunkowego zmiennych s' oraz r. Umożliwia to, znając jedynie aktualny stan, przewidzenie prawdopodobieństwa zajścia każdej „przyszłości", którą można osiągnąć z tego stanu. Jest to tak zwana dynamika środowiska.

Problem oszacowania odległych nagród (ang. *sparse reward)* – często w uczeniu przez wzmacnianie występuje sytuacja, w której agent ma wybór podjęcia wielu decyzji o różnym zwrocie. Generuje to pytanie czy powinno się wykonywać akcje, które przyniosą jak największą nagrodę lub jak najmniejszą karę w krótkim czasie, czy też wybierać akcje, które prowadzą w dalszej przyszłości do osiągnięcia większej nagrody kumulacyjnej. Aby ustalić właściwą politykę można użyć zwrotu (ang. *return*).

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+2} + \ldots = \sum_{k=0} \gamma^k R_{t+k+1} \tag{2.2}$$

t – aktualny krok
G_t – zwrot
R_{t+1} – nagroda natychmiastowa otrzymana za wykonanie A_t w stanie S_t
R_{t+n} – nagroda natychmiastowa otrzymana za wykonanie A_{t+n} w stanie S_{t+n}
γ – parametr dyskontowania

Parametr dyskontowania (ang. *discount factor*) mieści się w przedziale [0, 1], reguluje wagę przyszłych nagród w kalkulacjach optymalnej akcji dla danego stanu S_t w kroku t. Poprzez regulację parametru γ można zmienić wpływ przyszłych nagród na podejmowane przez politykę decyzje. W rzeczywistości wartość parametru dyskontowania najczęściej znajduje się w przedziale (0, 1), gdyż w przypadku wartości $\gamma = 0$ algorytm będzie brał pod uwagę jedynie nagrody natychmiastowe, zaś dla $\gamma = 1$ może wystąpić problem sumy nieskończonej, jeśli nie jest ona zbieżna.

Polityka – wskazuje, jakie działanie powinien wybrać agent, aby otrzymać nagrodę; jest to odwzorowanie zbioru stanów na właściwy zestaw działań. Bazując na polityce można stworzyć odpowiedni plan, lecz pojęcia te nie są równoznaczne. Plan to konkretna sekwencja działań od stanu początkowego aż do docelowego, zaś polityka określa co zrobić w konkretnej sytuacji by osiągnąć określony stan [8,9]. Polityka zmienia się w trakcie uczenia modelu. Z reguły zaczyna się od polityki losowej, która w trakcie uczenia jest udoskonalana aż do osiągnięcia polityki, która przynosi największe poznane przez algorytm nagrody dla wszystkich poznanych stanów. Najczęściej stosowanym typem polityki jest polityka stochastyczna (2.3), której wynikiem jest prawdopodobieństwo warunkowe dla konkretnych akcji [10].

Polityka stochastyczna jest określana w następujący sposób:

$$\pi(a|s) = P(A_t = a|S_t = s) \tag{2.3}$$

t – aktualny krok
a – akcja
s – stan
S_t – stan w kroku t
A_t – akcja w kroku t

Dużo rzadziej spotykana jest polityka deterministyczna, która dla danego stanu generuje tylko jedną prawidłową akcję. W polityce tej nie dochodzi do generacji rozkładu prawdopodobieństwa warunkowego wykonania każdej akcji – jej wynikiem jest jednoznaczne określenie jedynej akcji właściwej dla danego stanu.

Równanie Bellmana jest jedną z podstaw uczenia przez wzmacnianie. Określa relację między dostępnymi akcjami w aktualnym stanie, a nagrodami, do których prowadzą w dłuższym czasie.

Funkcja wartości (2.4) używana w równaniu wygląda następująco:

$$v_\pi(s) = \mathbb{E}_\pi(G_t|S_t = s) \tag{2.4}$$

π – polityka
t – aktualny krok
s – stan
G_t – zwrot w kroku t
S_t – stan w kroku t

Pochodzi z niej funkcja akcja-wartość:

$$q_\pi(s,a) = \mathbb{E}_\pi(G_t|S_t = s, A_t = a) \tag{2.5}$$

π – polityka
t – aktualny krok
s – stan
a – akcja zgodna z polityką
G_t – zwrot w kroku t
S_t – stan w kroku t

Funkcja akcja-wartość definiuje oczekiwany zwrot wybranej akcji w danym stanie. Pozwala to na proste określenie jaką akcję należy wykonać w stanie, w którym agent się aktualnie znajduje.

$$v_\pi(s) = \mathbb{E}_\pi(R_t + \gamma G_t + 1|S_t = s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)(r + \gamma v_\pi(s')) \tag{2.6}$$

π – polityka
t – aktualny krok
s – stan
a – akcja
G_t – zwrot w kroku t
S_t – stan w kroku t
R_t – nagroda w kroku t

Samym równaniem Bellmana (2.6) nazywamy własność funkcji wartości, która pozwala na jej zastosowanie w formie rekurencyjnej. W równaniu używane procesy decyzyjne Markowa w celu przekształcenia sumy nieskończonej do układu równań liniowych, których rozwiązanie pozwala wyznaczyć wartości akcji dla stanu [10].

2.2.2 Zalety RL w porównaniu do innych rodzajów uczenia

Uczenie przez wzmacnianie jest szczególnie przydatne w środowiskach interaktywnych o nieznanej optymalnej metodzie zachowania, w których istotne jest podejmowanie przez agenta konkretnych działań. W tych przypadkach uczenie nadzorowane jest niemożliwe ze względu na brak danych koniecznych do wyszkolenia modelu. Uczenie przez wzmacnianie rozwiązuje wspomniane problemy poprzez obserwację wpływu swoich akcji na otrzymywane nagrody.

RL niewątpliwie doskonale sprawdza się, gdy nie istnieje pełna wiedza o warunkach środowiska. Uczenie nadzorowane może w takim przypadku utrudnić fakt, że zebranie pewnych danych w niepewnym środowisku jest niezwykle skomplikowane.

Uczenie przez wzmacnianie może mieć szczególne zastosowanie w przypadku nauki złożonych zadań, jak chociażby szkolenie agenta w zakresie interakcji ze skomplikowaną grą

komputerową. Umożliwia ono wtedy uczenie się na błędach i sukcesach, które są najczęściej jedynym sposobem obserwacji skuteczności akcji gracza.

2.2.3 Wady RL w porównaniu do innych rodzajów uczenia

Jedynym sposobem na wypracowanie odpowiedniej polityki jest metoda prób i błędów, zaś wyłącznym sygnałem uczącym otrzymywanym przez agenta jest kara bądź nagroda. Czyni to proces uczenia długim i pracochłonnym.

Obserwacje agenta to wynik jego działań, więc mogą się między nimi pojawić silne korelacje czasowe. Konsekwencje konkretnych działań niejednokrotnie mogą pojawić się dopiero po wielu zmianach stanu środowiska, co również wydłuża sam proces uczenia się [11].

Problemem, na jaki można natrafić w przypadku uczenia przez wzmacnianie jest wypracowanie równowagi między eksploracją i eksploatacją. Jeśli agent wyłącznie eksploruje, może nie nauczyć się efektywnej strategii działania, zaś nadmierna eksploatacja spowoduje, że agent nie odkryje nowych, potencjalnie lepszych strategii [8].

2.2.4 Algorytmy uczenia przez wzmacnianie

Algorytmy uczenia przez wzmacnianie można podzielić na dwie kategorie: oparte na modelu oraz bez-modelowe. Wśród tych pierwszych wyróżniamy typy przedstawione poniżej.

Algorytmy optymalizujące funkcję wartości uczą się poprzez iteracyjne obliczanie funkcji wartości dla wszystkich stanów środowiska. Zaczynają od losowej funkcji wartości, aby następnie iteracyjnie ją aktualizować, w celu znalezienia strategii, która maksymalizuje oczekiwany zwrot. Algorytmy optymalizujące wartości nie wymagają zapisywania polityki, ponieważ funkcja wartości zawiera wszystkie informacje potrzebne do wyboru optymalnej akcji w każdym stanie. W porównaniu do algorytmów optymalizujących politykę optymalizacja funkcji wartości wymaga z reguły dłuższego czasu, aby osiągnąć optymalne wyniki.

Algorytmy optymalizujące politykę uczą się poprzez iteracyjne obliczanie prawdopodobieństw wyboru akcji w każdym stanie środowiska. Zaczynają od losowej polityki, przy pomocy której wybierają akcję dla określonego stanu celem obliczenia funkcji wartości odpowiadającej temu stanowi. Po obliczeniu funkcji wartości, algorytmy aktualizują politykę na bazie owej funkcji. Wadą wspomnianych algorytmów jest ryzyko szybkiego znalezienia optymalnej polityki lokalnej, która jest jednak gorsza od najlepszej polityki globalnej. Uzyskanie optymalnej polityki trwa jednak zazwyczaj znacznie krócej niż w przypadku algorytmów optymalizujących funkcję wartości.

Algorytmy aktor-krytyk łączą zalety sposobów optymalizujących wartość, jak i politykę, gdyż składają się z dwóch części – aktora optymalizującego politykę i krytyka optymalizującego funkcję wartości. Opisywane algorytmy są w stanie osiągać dobre efekty w relatywnie krótkim czasie w porównaniu z alternatywnymi metodami.

Kolejnym typem algorytmów RL są **metody oparte na modelu**. W przeciwieństwie do metod bez-modelowych, które uczą się bezpośrednio przez mapowanie akcji do stanów, metody oparte o modele tworzą predyktywny model dynamiki środowiska. Jest on używany do symulowania przyszłych stanów i potencjalnych efektów, co umożliwia agentom planowanie i wybieranie akcji biorąc pod uwagę kontekst przyszłości. Takie algorytmy mogą otrzymać gotowy model środowiska lub znaleźć go samodzielnie poprzez interakcję ze środowiskiem.

2.3 Środowiska RL w Pythonie – Gym, Gymnasium, PettingZoo

Historia ogólnodostępnych i ustandaryzowanych środowisk uczenia przez wzmacnianie rozpoczęła się w kwietniu 2016 roku, kiedy OpenAI opublikowało pierwszą wersję beta biblioteki Gym. Pakiet ten został stworzony do implementacji i porównywania algorytmów uczenia przez wzmacnianie poprzez zapewnienie ustandaryzowanego API do komunikacji między algorytmami uczenia przez wzmacnianie i środowiskiem. Dostarczył przy tym spory zestaw przykładowych środowisk. Znajdowały się wśród nich zarówno te oparte o symulację fizyki (Mountain Car, Pendulum, Cart Pole), proste gry (Frozen Lake, Blackjack) czy nawet dostosowane do użycia jako środowisko RL gry z konsol Atari, gdzie przestrzeń obserwacji była oparta o to, co widziałby gracz na ekranie. [12]

W 2021 roku wsparcie dla Gym zostało zakończone, zaś projekt zmienił się w bibliotekę Gymnasium. Ta zmiana unowocześniła istniejące standardy ustalone przez Gym i wprowadziła wrappery kompatybilności, które pozwalały użyć Gymnasium w miejsce Gym bez konieczności ekstensywnych zmian. [13]

Również w 2021 roku zostały opublikowane efekty prac nad stworzeniem wersji Gym dla środowisk wieloagentowych, która pozostała wierna ustalonym wcześniej standardom. Efektem tych prac jest PettingZoo, biblioteka oferująca z jednej strony standardy uczenia wieloagentowego oparte o Gym, z drugiej zaś rozwijająca nowe sposoby myślenia w dziedzinie uczenia przez wzmacnianie. Projekt dostarcza wiele środowisk wieloagentowych, które są w większości oparte o kooperacyjne gry fizyczne (Pistonball), tradycyjne gry planszowe (szachy, Texas Hold'em, Tic Tac Toe) oraz gry Atari z graficzną przestrzenią obserwacji. [14]

Za Gymnasium oraz PettingZoo odpowiada Farama Foundation, organizacja non-profit skupiona na rozszerzaniu horyzontów dziedziny uczenia przez wzmacnianie poprzez promowanie lepszej standaryzacji oraz narzędzi używanych zarówno przez naukowców, jak i przemysł wykorzystujący RL. Aktualnie fundacja zajmuje się utrzymaniem trzech

najważniejszych bibliotek rdzeniowych: Gymnasium, PettingZoo oraz Minari (standard dla datasetów RL).

2.3.1 Standard Gym środowisk uczenia przez wzmacnianie

Podstawą środowisk uczenia przez wzmacnianie jest powtarzany przez cały czas działania środowiska cykl pobierania obserwacji oraz podejmowania decyzji i kalkulowania ich efektów. Dzięki standaryzacji środowisk, większość tworzonych dzisiaj projektów jest zgodna z zaleceniami standardu wprowadzonego przez Gym. Każde środowisko składa się więc z kilku najważniejszych metod.

Pierwszą z nich jest inicjalizacja, która pozwala na przekazanie wartości inicjalizacyjnych środowiska (na przykład seed dla generatora szumu), zainicjowanie podstawowych wartości oraz załadowanie zasobów z plików. Ta metoda jest wykonywana jedynie raz w trakcie wczytania środowiska.

Następnie wzywana jest metoda restartująca środowisko – wszystkie zmienne są ustawiane na swoje początkowe wartości, a jakiekolwiek elementy losowe są generowane ponownie. Ponieważ agenci potrzebują informacji o ich obserwacjach na początku działania, w przypadku środowisk równoległych, ta metoda zwraca stan środowiska w postaci Dictionary z obserwacjami dla każdego agenta. W środowiskach jednoagentowych zwracany jest jego stan jako gotowe obserwacje agenta. Jeśli wieloagentowe środowisko jest sekwencyjne, każdy agent przed swoim ruchem musi pobrać nowe obserwacje. W konsekwencji potrzebna jest osobna metoda wywoływana przed ruchem każdego agenta, która zwraca jego obserwacje.

Niewątpliwie najważniejszą metodą każdego środowiska jest funkcja kroku, która w argumentach może przyjmować samą akcję pojedynczego agenta (Gym, Gymnasium), ID agenta i jego akcję (środowiska wieloagentowe sekwencyjne) lub Dictionary zawierający akcje dla każdego agenta (środowiska wieloagentowe równoległe). W trakcie kroku każda akcja jest wykonywana (w kolejności zależnej od implementacji), a jej efekty na środowisko i agentów są obliczane i wprowadzane. Jako wyjście funkcji kroku zwracane są nowe obserwacje oraz informacje o agentach. [12,13,14]

2.3.2 Problem akcji nielegalnych i sposoby jego zapobiegania

W środowiskach uczenia przez wzmacnianie, agenci podejmują akcje na podstawie obserwacji, lecz z punktu widzenia algorytmu każda akcja jest wykonalna. Generuje to poważny problem, gdyż wiele akcji może być z różnych powodów zablokowanych – przez pozycję agenta, elementy otoczenia lub inne przyczyny. Aby przeciwdziałać temu zjawisku, potrzebny jest sposób, w jaki algorytm otrzyma informację, że akcja nie może zostać wykonana – jest akcją nielegalną.

Pierwszym z tych sposobów jest penalizacja akcji nielegalnych, czyli podejście, które karze agenta próbującego wykonać niedozwoloną akcję pewną ustaloną wartością. Dzięki temu algorytm uczenia jest w stanie połączyć obserwację z wynikiem nielegalnej akcji i wywnioskować, że jest ona nieopłacalna w danej sytuacji. Omawiane podejście jest jednak nadmiernie uproszczone i wymaga olbrzymiej ilości danych uczących, by algorytm był w stanie utworzyć politykę, która rozumie zależność dozwolonych akcji od wszystkich parametrów przestrzeni obserwacji.

Drugą, dużo skuteczniejszą opcją jest maska nielegalnych akcji – tablica, która każdej akcji przypisuje wartość oznaczającą jej legalność (najczęściej 1) lub brak (najczęściej 0). Jako że sieć polityki zachowania agentów generuje rozkład korzystności akcji dla obserwacji, maska akcji może zostać nałożona na wyjście sieci w celu wyeliminowania akcji nielegalnych. Dzieje się tak poprzez ustawienie dla każdej akcji nielegalnej wartości niższej niż jakakolwiek wygenerowana (często wartość minimalna danego typu danych lub wartość oznaczająca ujemną nieskończoność), tak aby algorytm wykonał najlepszą dostępną akcję poprzez wybranie tej z najwyższą przewidywaną wartością. Przedstawione podejście działa zarówno w trakcie uczenia (algorytm nie będzie nigdy podejmował akcji nielegalnych) jak i w trakcie pracy gotowego algorytmu. Największą wadą wspomnianej opcji jest brak wbudowanego wsparcia przez większość algorytmów uczenia przez wzmacnianie, co może się jednak zmienić w przyszłości.

Ze względu na fakt, że biblioteki algorytmów RL najczęściej nie oferują obsługi maski akcji w środowiskach równoległych, potrzebny jest sposób pozwalający na zachowanie kompatybilności z nimi przy jednoczesnym przeciwdziałaniu problemowi nielegalnych akcji. Rozwiązaniem jest maska akcji przekazywana jako część obserwacji każdego agenta połączona z penalizacją nielegalnych akcji. Poprzez dołączenie tablicy maski do tablicy obserwacji, algorytm uczenia przez wzmacnianie jest w stanie w dużo prostszy sposób utworzyć korelację między elementami przestrzeni obserwacji, a legalnością akcji. Sposób ten jest dużo mniej efektywny niż czysta maska akcji, pozwala jednak na zachowanie kompatybilności z bibliotekami, co w wielu przypadkach może być kluczowym argumentem do użycia maski nielegalnych akcji w obserwacjach agenta. Jej implementacja jest przy tym dużo prostsza, gdyż nie wymaga stworzenia nowych funkcjonalności skomplikowanych bibliotek uczenia przez wzmacnianie [15].

2.3.3 Biblioteki Stable-Baselines3 oraz SuperSuit

Wśród wielu dostępnych bibliotek uczenia przez wzmacnianie, jedną z najbardziej popularnych jest Stable-Baselines3. Jest to pakiet oferujący proste, lecz sprawdzone implementacje wielu algorytmów RL bazujących na PyTorch. Algorytmy dostarczane przez ten pakiet są łatwe w implementacji ze względu na dobrą dokumentację, czysty kod oraz zunifikowaną strukturę. Dodatkowym atutem tej biblioteki jest wsparcie Tensorboard, czyli pakietu pozwalającego na zapisywanie logów procesu uczenia i następnie ich graficzne

przedstawienie. Stable-Baselines3 jest przede wszystkim przystosowane do pracy z Gym oraz Gymnasium, lecz możliwe jest użycie biblioteki dla środowisk wieloagentowych standardu PettingZoo przy użyciu wrapperów kompatybilności zapewnianych przez SuperSuit [16].

SuperSuit jest biblioteką zarządzaną przez Farama Foundation, która dostarcza dużą ilość wrapperów, czyli funkcji zmieniających sposób, w jaki agent lub algorytm wchodzi w interakcję ze środowiskiem. Mogą być one używane do modyfikacji większości parametrów środowiska lub sposobów, w jaki środowisko wykonuje interakcje z algorytmem RL [17].

3. Implementacja środowiska wieloagentowego Ecosystem

Głównym celem pracy jest przedstawiona w poniższym rozdziale implementacja wieloagentowego środowiska w standardzie PettingZoo. Opisane zostały jego kluczowe elementy, cechy i funkcjonalność, a także zaprezentowano fragmenty kodu, które dogłębnie przedstawiają działanie środowiska. Wymieniono dostępne dla agentów akcje oraz przedstawiono obserwacje, nagrody i kary.

Środowisko zostało zaimplementowane na podstawie klasy ParallelEnv będącej częścią biblioteki PettingZoo. Projekt dziedziczy z tej klasy przede wszystkim swoją strukturę, dzięki czemu dostosowany jest do aktualnych standardów wymaganych przez biblioteki uczenia przez wzmacnianie takie jak RLlib, Stable-Baselines3 czy CleanRL.

Projekt został wykonany w języku Python, który jest najpopularniejszym językiem stosowanym w uczeniu maszynowym. Oferuje on takie biblioteki ML jak PyTorch czy Tensorflow, które są złotym standardem, jeśli chodzi o rozbudowane zasoby, wsparcie sprzętu oraz CUDA czy olbrzymią bazę użytkowników.

Pełny kod źródłowy implementacji środowiska dostępny jest w publicznym repozytorium GitHub autora [18].

3.1 Logika środowiska

W PettingZoo środowiska funkcjonują na podstawie tur/rund. Wyróżniamy 2 typy środowisk – sekwencyjne oraz równoległe. Ecosystem jest środowiskiem drugiego typu, co oznacza, że wszyscy agenci wykonują ruch w tym samym momencie na podstawie obserwacji, które otrzymali jednocześnie jako efekt poprzedniego kroku. Nagrody natychmiastowe są przekazywane wszystkim aktywnym agentom w tym samym momencie. Dzięki wrapperowi *black_death* zapewnianemu przez SuperSuit, środowisko dopuszcza śmierć agentów. Wrapper ten upewnia się, że algorytmy otrzymają dla martwych agentów obserwacje wypełnione zerami.

3.1.1 Inicjalizacja środowiska oraz wartości bazowych

Krok inicjalizacji zostaje wykonany w momencie zaimportowania środowiska do projektu i przywołania klasy środowiska. W tym kroku można przekazać parametry środowiska takie jak zmienione koszty, ilość agentów lub pożywienia, a także plik graficzny dla mapy.

```
self.params = {
    "mapsize": 128,
    "min_depth": 10,
    "max_depth": 30,
    "starting_population": 2,
    "food_value": 200,
    "movement_cost": -1,
    "death_penalty": -2000,
    "illegal": -20,
    "positive_food_mult": 50,
    "food_per_agent": 5,
    "max_timesteps": 200000,
    "distance_factor": 10,
    "eyesight_range": 32,
    "visible_agents": 1,
    "visible_foods": 2,
    "max_food": 400,
    "max_hp": 1000,
    "bite_damage": 400,
    "feed_range": 3,
    "bite_range": 3,
    "dmg_penalty_mult": 1,
}
for key in params:
    if key in self.params:
        self.params[key] = params[key]
    else:
        print("Parameter", key, "is invalid")
```

Rys. 3.1 Parametry dostępne do modyfikacji przez użytkownika

Rys. 3.1 przedstawia listę parametrów możliwych do zmodyfikowania w trakcie inicjalizacji przez użytkownika. W trakcie inicjalizacji środowiska można jako argument przekazać Dictionary zawierający niektóre klucze. Parametry środowiska zostaną podmienione dla wszystkich występujących par. Jeśli któraś z nich nie występuje, środowisko poinformuje o tym użytkownika.

Mapa jest wczytywana z pliku za pośrednictwem *OpenCV*, a następnie konwertowana do przestrzeni monochromatycznej i skalowana do zadanej wielkości. Jest ona następnie przeliczana z wartości 0-255 na wartości głębokości. Są one wyliczane na podstawie wartości maksymalnej i minimalnej głębokości środowiska. Pozwala to na symulację dna morskiego w dowolnej skali i głębokości.

3.1.2 Restart środowiska – wyzerowanie zmiennych oraz danych agentów

Każda rozgrywka środowiska rozpoczyna się od zrestartowania wszystkich wartości, wylosowania potrzebnych elementów (takich jak lokalizacje początkowe pożywienia i agentów) oraz przekazania początkowych obserwacji dla każdego agenta.

```python
def reset(self, seed=None, options=None):
    self.agents = copy(self.possible_agents)
    self.agentData = {}
    self.timestep = 0
    self.map = np.zeros(self.mapsize)
    self.foods = []
    # This function starts the world:
    # loads the map from image,
    # spawns agents and generates food
    self.generateMap()
    self.terminated = []
    self.rewards = {
        agentID: 0.0 for agentID in self.agents
    }
    observations = {agent: self.observation(agent) for agent in self.agents}
    infos = {agent: {} for agent in self.agents}
    self.state = observations

    return observations, infos
```

Rys. 3.2 Metoda restartu środowiska

Na rys. 3.2 został przedstawiony kod funkcji restartu środowiska, który ma następującą strukturę:

1. Lista agentów jest odnawiana poprzez skopiowanie listy *possible_agents*, czyli wszystkich możliwych w środowisku agentów
2. Dane agentów są czyszczone, zostaną one od nowa ustalone w momencie stworzenia mapy i wygenerowania agentów
3. Licznik kroków jest zerowany, mapa jest inicjowana zerami, lista pożywienia zostaje wyczyszczona
4. Wywoływana jest funkcja generowania mapy:
    a. Mapa zostaje wczytana z przeskalowanej mapy głębokości
    b. Wygenerowane zostaje jedzenie w ilości zadeklarowanej podczas inicjalizacji środowiska
    c. Dane agentów zostają wygenerowane – pozycja wylosowana, agent otrzymuje pełne punkty zdrowia i pożywienia na start
5. Lista usuniętych z symulacji agentów oraz nagrody są zerowane
6. Obserwacje startowe są pobierane dla każdego agenta, stan środowiska jest ustalany na aktualne obserwacje
7. Lista obserwacji i puste informacje są zwracane zgodnie ze standardem PettingZoo

3.1.3 Funkcja kroku środowiska – działanie symulacji

Najważniejszym elementem każdego środowiska jest funkcja kroku, która jest wykonywana za każdym razem, gdy agenci podejmują akcje (rys. 3.3).

```
def step(self, actions):
    if not actions:
        self.agents = []
        return {}, {}, {}, {}, {}
    self.rewards = {agentID: 0.0 for agentID in self.agents}
    eaters = []
    rest = {}
    for agent, action in actions.items():
        if agent in self.agents:
            if action == 6:
                eaters.append(agent)
            else:
                rest[agent] = action
```

Rys. 3.3 Początek funkcji kroku

Funkcja ta przyjmuje jako argument akcje przypisane każdemu agentowi i dokonuje podziału agentów według wybranych akcji – ataki i żywienie są wykonywane jako pierwsze, akcje ruchu zostają wykonane zaraz po nich. Podział ten został wprowadzony w celu zapobiegnięcia sytuacji, w której agent wykonujący akcję jako pierwszy, ucieka od agenta, który chciał go zaatakować.

```
for agent in eaters:
    food = self.performAction(agent, 6)
    if food > 0:
        print(self.rewards[agent])
        self.rewards[agent] += food * self.params["positive_food_mult"]
        print(self.rewards[agent])
    else:
        self.rewards[agent] += food

for agent, action in rest.items():
    food = self.performAction(agent, action)
    self.rewards[agent] += food
```

Rys. 3.4 Funkcja kroku – wykonywanie akcji przez agentów

Rys. 3.4 przedstawia kolejność i sposób wywoływania akcji. Każdy agent wykonuje swoją akcję poprzez wywołanie funkcji wykonania akcji. Funkcja ta zwraca ilość punktów pożywienia straconych lub zyskanych w efekcie tej akcji, a także potencjalnie przesuwa agenta w przestrzeni środowiska. Wartość pożywienia jest proporcjonalnie nagradzana lub karana, tak aby każdy agent był motywowany do prawidłowego działania przez podstawową potrzebę – zdobywanie pokarmu i utrzymanie się przy życiu. W parametrach środowiska zdefiniowany może być mnożnik dla pozytywnych i negatywnych wartości uzyskanego pożywienia.

```
self.timestep += 1
env_truncation = self.timestep >= self.params["max_timesteps"]
truncations = {agent: env_truncation for agent in self.agents}
terminations = {agent: False for agent in self.agents}

for agent in self.agents:
    if self.agentData[agent]["hp"] <= 0 or self.agentData[agent]["food"] <= 0:
        self.agents.pop(self.agents.index(agent))
        terminations[agent] = True
        self.rewards[agent] += self.params["death_penalty"]
```

Rys. 3.5 Funkcja kroku – eliminacja agentów, zakończenie pracy środowiska

Kiedy każdy agent wykona akcję, sprawdzane jest czy nie została przekroczona maksymalna ilość kroków epizodu środowiska i czy wszyscy agenci powinni dalej pozostać przy życiu, co jest zależne od poziomu punktów życia i pożywienia (rys. 3.5). Dzięki zastosowaniu wrappera *black_death* od SuperSuit, agenci mogą być wykluczani z gry.

```
observations = {
    agent: self.observation(agent, actions[agent]) for agent in self.agents
}
self.state = observations
infos = {agent: {} for agent in self.agents}

if env_truncation:
    self.agents = []

if self.render_mode == "human":
    self.render()

return observations, self.rewards, terminations, truncations, infos
```

Rys. 3.6 Funkcja kroku – obserwacje i zwrócenie wartości

Na końcu dla każdego agenta, który nie został wykluczony, zostaje pobrana nowa obserwacja. Dictionary obserwacji, nagrody, informacje o wykluczeniu agentów oraz skończeniu działania zostają zwrócone jako wynik funkcji kroku. Zwrócenie informacji o agencie jest wymagane w celu zachowania standardu PettingZoo, nawet kiedy są one puste.

3.2 Cykl życia agenta

Każdy agent w środowisku ma przede wszystkim jeden cel – osiągnąć jak największą nagrodę skumulowaną poprzez skuteczne znajdowanie pożywienia lub żywienie się innymi agentami. Każdy agent posiada również punkty życia oraz pożywienia. W momencie, gdy którakolwiek z tych wartości spadnie do 0, agent zostaje usunięty z rozgrywki i otrzymuje karę za śmierć.

Ideą pracy było stworzenie środowiska, w którym agent będzie szukał pożywienia wokół siebie, poruszał się w jego stronę i z niego korzystał, a także potencjalnie atakował aktywnie lub okazyjnie innych agentów w celu uzyskania pożywienia lub prewencyjnej samoobrony. Agenci mają też możliwość podjęcia decyzji o nieopłacalności atakowania innych agentów, ze względu na ryzyko bycia przez nich zaatakowanym.

3.3 Obserwacje prowadzone przez agenta

W celu oceny tego, która akcja jest najlepsza w danej sytuacji, agent pobiera od środowiska obserwacje. Są one oparte na symulacji wzroku i odczuwaniu otoczenia przez agenta. Wartości obserwacji są znormalizowane do zakresu [-1.0, 1.0] w celu lepszej interpretacji przez sieci kierujące agentami.

```
def observation(self, agentID, action=None):
    a_x = self.agentData[agentID]["x"]
    a_y = self.agentData[agentID]["y"]
    a_d = self.agentData[agentID]["depth"]
    visibleAgents = []
    visibleFoods = []

    terrain = np.full(self.terrain_shape, 1.0)
    surrounding = [1.0, 1.0, 1.0, 1.0, 1.0]

    for x in range(-self.r_x, self.r_x + 1):
        for y in range(-self.r_y, self.r_y + 1):
            c_x = a_x + x
            c_y = a_y + y
            if 0 <= c_x < self.mapsize[0] and 0 <= c_y < self.mapsize[1]:
                terrain[x][y] = (
                    self.map[c_x][c_y] - a_d
                    ) / self.max_depth * 2 - 1

    surrounding[0] = terrain[2][1]
    surrounding[1] = terrain[0][1]
    surrounding[2] = terrain[1][1]
    surrounding[3] = terrain[1][2]
    surrounding[4] = terrain[1][0]
```

Rys. 3.7 Obserwacje agenta – teren otaczający

Przedstawioną na rys. 3.7 pierwszą częścią obserwacji jest głębokość, którą agent rejestruje pod sobą dla każdej z możliwych akcji. Jest to tablica wartości oznaczających ukształtowanie terenu jak również informujących o różnego rodzaju przeszkodach. W przypadku, w którym poziom dna sięga poza aktualne położenie wertykalne agenta, wartości to odzwierciedlą. Kraniec mapy jest reprezentowany jako bardzo wysoka ściana niezależnie od aktualnej głębokości agenta.

Następnym krokiem jest przekazanie informacji o ilości posiadanych punktów pożywienia oraz życia, przeskalowane w zakresie [-1.0, 1.0] wobec swoich maksymalnych wartości.

Kolejnym elementem obserwacji są informacje o lokalizacji jednego najbliższego agenta oraz dwóch najbliższych pożywień (w parametrach można dostosować liczbę widocznych obiektów każdego typu) w zasięgu wzroku. Położenie tych obiektów jest mierzone w stosunku do agenta. Informacje o ich lokalizacji reprezentowane są przez następujące wartości:

- Dystans w stosunku do zasięgu wzroku, obiekty tuż przy agencie będą miały wartości bliskie -1, zaś te na granicy wzroku 1.
- Orientacja wobec agenta jest reprezentowana przez liczby 0-6, każda odpowiadająca numerowi akcji, która przesunęłaby agenta w kierunku obiektu. Ta wartość jest przeskalowana w zakresie [-1.0, 1.0]. Wartość 1.0 oznacza akcję żywienia/ataku.
- Typ obiektu, gdzie 1 to pożywienie, -1 to agent, zaś 0 oznacza brak obiektu w danym polu ze względu na brak agenta/pożywienia w zasięgu wzroku.

```
for agent in self.agents:
    if agentID != agent:
        b_x = self.agentData[agent]["x"]
        b_y = self.agentData[agent]["y"]
        b_d = self.agentData[agent]["depth"]
        if self.rayCast(a_x, a_y, a_d, b_x, b_y, b_d):
            x, y, d, dist, best_action = self.calcVector(
                agentID, agent)
            visibleAgents.append([dist, best_action, -1, x, y, d])

agents = sorted(visibleAgents, key=lambda x: x[0])[:self.visible_agent_amount]
if len(agents) > 0:
    closest_agent = agents[0][0]
else:
    closest_agent = self.sight_distance
for a in agents:
    a[0] = a[0] / self.sight_distance * 2 - 1
    a[1] = a[1] / 3 - 1
    a[3] = a[3] / self.sight_distance * 2 - 1
    a[4] = a[4] / self.sight_distance * 2 - 1
    a[5] = a[5] / self.sight_distance * 2 - 1
if len(agents) < self.visible_agent_amount:
    for i in range(0, self.visible_agent_amount - len(agents)):
        agents.append([0, 0, 0, 0, 0, 0])
```

Rys. 3.8 Obserwacje agenta – najbliższe obiekty

Algorytm pobierania obserwacji najbliższych agentów jest przedstawiony na rys. 3.8, analogiczny algorytm stosowany jest również dla najbliższego pożywienia.

```
action_mask = np.ones(7)
if a_y >= self.mapsize[1] - 1:
    action_mask[0] = -1
elif a_y <= 0:
    action_mask[1] = -1
if a_x >= self.mapsize[0] - 1:
    action_mask[2] = -1
elif a_x <= 0:
    action_mask[3] = -1
if a_d >= self.map[a_x][a_y] - 1:
    action_mask[4] = -1
elif a_d <= 0:
    action_mask[5] = -1
if closest_food > self.feedRange and closest_agent > self.attackRange:
    action_mask[6] = -1
```

Rys. 3.9 Obserwacje agenta – maska nielegalnych akcji

Finalnie w obserwacji przekazywana jest maska nielegalnych akcji, której sposób obliczania przedstawia rys. 3.9. Jest to tablica o długości równej ilości akcji dostępnych w środowisku. Składa się z wartości -1 oraz 1, gdzie 1 jest akcją legalną, zaś -1 nielegalną. Tradycyjnie maska nielegalnych akcji powinna być przekazywana osobno od obserwacji, lecz z powodu braku wsparcia większości bibliotek dla tego rozwiązania, dużo bezpieczniej jest przekazać ją jako część obserwacji. Nie jest to rozwiązanie idealne, pozwala jednak na relatywnie jasne przesłanie wymaganych informacji do agenta, który może szybko połączyć fakty i wywnioskować, które pola odpowiadają danym akcjom. Ponieważ próby wykonania błędnych akcji są karane, polityka powinna mieć możliwość celnego wybierania legalnych akcji na podstawie maski akcji przekazanej w obserwacji.

3.4 Akcje dostępne w środowisku

Mapa środowiska jest mapą 3-wymiarową, co skutkuje potrzebą implementacji akcji poruszania się we wszystkich trzech osiach. Każdy agent zmienia położenie o 1 pole, co kosztuje 1 jednostkę pożywienia i zarazem w efekcie produkuje karę o wartości -1.

```
def performAction(self, agentID, action):
    if self.agentData[agentID]["hp"] <= 0 or self.agentData[agentID]["food"] <= 0:
        return 0
    x = self.agentData[agentID]["x"]
    y = self.agentData[agentID]["y"]
    d = self.agentData[agentID]["depth"]
    f = 0
```

Rys. 3.10 Metoda wykonania akcji – zmienne pomocnicze

Rys. 3.10 przedstawia początek funkcji wykonania akcji, która rozpoczyna się od wstępnego sprawdzenia, czy agent w ogóle ma prawo wykonać akcję (czy jest żywy). W przypadku potwierdzenia, że agent żyje, inicjowane są wartości pomocnicze.

```
match action:
    case 0:  # y+
        if y < self.mapsize[1] - 1 and self.depth_map[x][y + 1] > d:
            self.agentData[agentID]["y"] += 1
            f += self.params["movement_cost"]
    case 1:  # y-
        if y > 0 and self.depth_map[x][y - 1] > d:
            self.agentData[agentID]["y"] += -1
            f += self.params["movement_cost"]
    case 2:  # x+
        if x < self.mapsize[0] - 1 and self.depth_map[x + 1][y] > d:
            self.agentData[agentID]["x"] += 1
            f += self.params["movement_cost"]
    case 3:  # x-
        if x > 0 and self.depth_map[x - 1][y] > d:
            self.agentData[agentID]["x"] += -1
            f += self.params["movement_cost"]
    case 4:  # depth+
        if d < self.map[x][y] - 1:
            self.agentData[agentID]["depth"] += 1
            f += self.params["movement_cost"]
    case 5:  # depth-
        if d > 0:
            self.agentData[agentID]["depth"] += -1
            f += self.params["movement_cost"]
```

Rys. 3.11 Metoda wykonania akcji – akcje ruchu

Rys. 3.11 przedstawia algorytm wykonania akcji i sprawdzenia jej poprawności. Jeśli którakolwiek z akcji ruchu miałaby wyjść poza dostępne granice mapy, poza poziom wody (głębokość < 0) lub poniżej dna (głębokość agenta > głębokość mapy na koordynatach agenta), jest ona traktowana jako akcja nielegalna i nie zostanie wykonana. W akcjach ruchu poziomego sprawdzane jest, czy agent nie wpłynie poniżej dna mapy przenikając przez dno. Agenci, którzy próbują wykonać taką akcję są karani ze względu na brak jej legalności.

```
case 6:
    eaten = False
    for agent in self.agents:
        if agentID != agent:
            if self.getDistance(
                x, y, d, self.agentData[agent]["x"],
                self.agentData[agent]["y"],
                self.agentData[agent]["depth"]
                ) <= self.attackRange and eaten == False:
                self.damage(agent)
                f += self.bite_damage
                eaten = True
                break
    if eaten == False:
        for food in self.foods:
            if self.getDistance(
                x, y, d, food[0], food[1], food[2]
                ) <= self.feedRange and eaten == False:
                f += self.params["food_value"]
                print(f, food, self.getDistance(x, y, d, food[0], food
                [1], food[2]))
                self.foods.pop(self.foods.index(food))
                eaten = True
                self.agentData[agentID]["hp"] = self.max_hp
                self.generateNewFood()
                break
```

Rys. 3.12 Metoda wykonania akcji – akcja żywienia i ataku

Przedstawiona na rys. 3.12 akcja żywienia i ataku jest jedyną akcją, która wykonywana jest zawsze w konkretnej kolejności. Pozwala na spożycie lub zaatakowanie pojedynczego celu. Może nim być inny agent lub pożywienie znajdujące się w zasięgu.

W pierwszej kolejności wykonywana jest iteracja po liście agentów w środowisku. Jeśli dystans między agentem wykonującym akcję a innym wynosi nie więcej niż maksymalny dystans ataku, inicjator otrzymuje nagrodę i pożywienie w wysokości zadanych obrażeń, zaś uszkodzony cel traci odpowiednią ilość punktów zdrowia oraz otrzymuje karę wynoszącą ułamek utraconego życia.

Jeśli żaden agent nie znajduje się w okolicy, analogicznie wykonywana jest iteracja po liście pożywień. W przypadku odnalezienia pokarmu, agent zyska ilość pożywienia odpowiednio ustaloną w parametrach środowiska i nagrodę do tego proporcjonalną, zaś pożywienie zostanie usunięte z listy. Na jego miejsce natychmiastowo pojawi się nowe pożywienie w losowym miejscu na dnie mapy, by uzupełnić deficyt stworzony przez agenta.

```
if f == 0:
    self.rewards[agentID] += self.params["illegal"]
    f += self.params["movement_cost"]

self.agentData[agentID]["food"] += f
return f
```

Rys. 3.13 Metoda wykonania akcji – penalizacja akcji nielegalnych

Jeśli agent nie zdobył lub nie stracił dotychczas punktów pożywienia, oznacza to, że wykonał akcję nielegalną, która nie miała efektów na środowisku. Agent otrzymuje karę za próbę wykonania akcji nielegalnej oraz traci ilość pożywienia taką, jak w przypadku wykonywania ruchu (rys. 3.13). Po sprawdzeniu, metoda wykonywania akcji zwraca ilość zdobytego lub straconego pożywienia.

3.5 Polityka nagród

Podstawowym założeniem uczenia przez wzmacnianie jest karanie i nagradzanie agenta za wykonywane przez niego akcje, tak więc potrzebna jest prawidłowo zdefiniowana polityka nagród i kar. W środowisku Ecosystem dobór odpowiednich wartości oparty był o symulację żywego organizmu.

Najważniejszym motywem w polityce nagród i kar jest potrzeba żywienia każdego agenta. Agenci posiadają pewną ilość punktów pożywienia na start (ilość ta może zostać ustalona przy importowaniu i inicjowaniu środowiska w projekcie), które to punkty bezpośrednio przekładają się na nagrody. Nadrzędnym celem każdego agenta jest utrzymywanie punktów pożywienia i życia powyżej 0 w celu uniknięcia śmierci.

Wszystkie wartości nagród i kar środowiska są możliwe do modyfikacji w trakcie jego inicjalizacji lub nawet w trakcie uczenia. Pozwala to użytkownikowi na wybranie takich parametrów, które pozwolą na uzyskanie różnych wyników, analizę konkretnych sytuacji lub ulepszenie efektów szkolenia dla zastosowanej metody uczenia.

3.5.1 Konsekwencje akcji ruchu

W celu lepszego odwzorowania działania organizmu oraz motywowania do aktywnego poszukiwania jedzenia, akcje ruchu kosztują niedużą ilość pożywienia (w standardzie 1.0), a co za tym idzie, agent wykonujący daną akcję otrzymuje karę równą ilości utraconego pożywienia. To podejście ma na celu nauczenie agentów, aby przede wszystkim kierowali się

w stronę najbliższego źródła pożywienia jednocześnie minimalizując liczbę wykonanych kroków.

W przypadku, gdy wybrana akcja jest niewłaściwa do wykonania w danym momencie, podejmujący jej próbę agent zostaje obciążony znaczną karą, co ma na celu naukę powiązań między maską nielegalnych akcji zawartą w obserwacjach, a możliwościami wykonania tych akcji w sytuacji, w której agent się znajduje.

3.5.2 Akcja żywienia i ataku

Szczególnym wypadkiem jest akcja żywienia i ataku, gdyż może ona zostać wykonana jedynie, gdy w zasięgu znajduje się pożywienie lub inny agent. W przypadku, w którym żaden właściwy obiekt nie jest dostępny jako cel wspomnianej akcji, zostaje ona uznana za nielegalną, co odzwierciedla maska nielegalnych akcji. Agent, który spróbuje ją wykonać otrzyma standardową karę za nielegalną akcję.

3.5.3 Otrzymywanie obrażeń

Agent może otrzymać obrażenia od innego agenta w przypadku, kiedy znajduje się w zasięgu jego ataku, pod warunkiem, że podejmie on akcję żywienia/ataku. Kara jest wprost proporcjonalna do otrzymanych obrażeń pomnożonych przez skonfigurowany w parametrach środowiska mnożnik.

3.5.4 Kara za śmierć

W przypadku, w którym ilość punktów życia lub pożywienia agenta osiągnie 0, agent zostaje wykluczony ze środowiska. Otrzymuje karę za śmierć oraz za wszystkie otrzymane obrażenia (jeśli jakieś otrzymał, czyli został zaatakowany). Standardową wartością kary jest -2000.0, co ma zmotywować agenta do jak najskuteczniejszych prób utrzymywania pozytywnych wartości pożywienia i punktów życia.

3.5.5 Nagroda za dystans

Każdy agent jest nagradzany za zbliżanie się do najbliższego mu pożywienia w sposób wykładniczy. Nagroda w zakresie 0 do 1 jest mnożona przez wartość ustalaną w parametrach środowiska. Taka nagroda ma na celu stworzenie w polityce agenta pozytywnego powiązania, które łączy ruch w stronę pożywienia z nagrodą. Ma to motywować agenta do znalezienia się jak najbliżej pokarmu, by możliwa była akcja żywienia. Nagroda za dystans jest kalkulowana według następującej formuły:

$$r_d = (1 - \frac{d_f}{d_{max}})^2 \cdot f \tag{3.1}$$

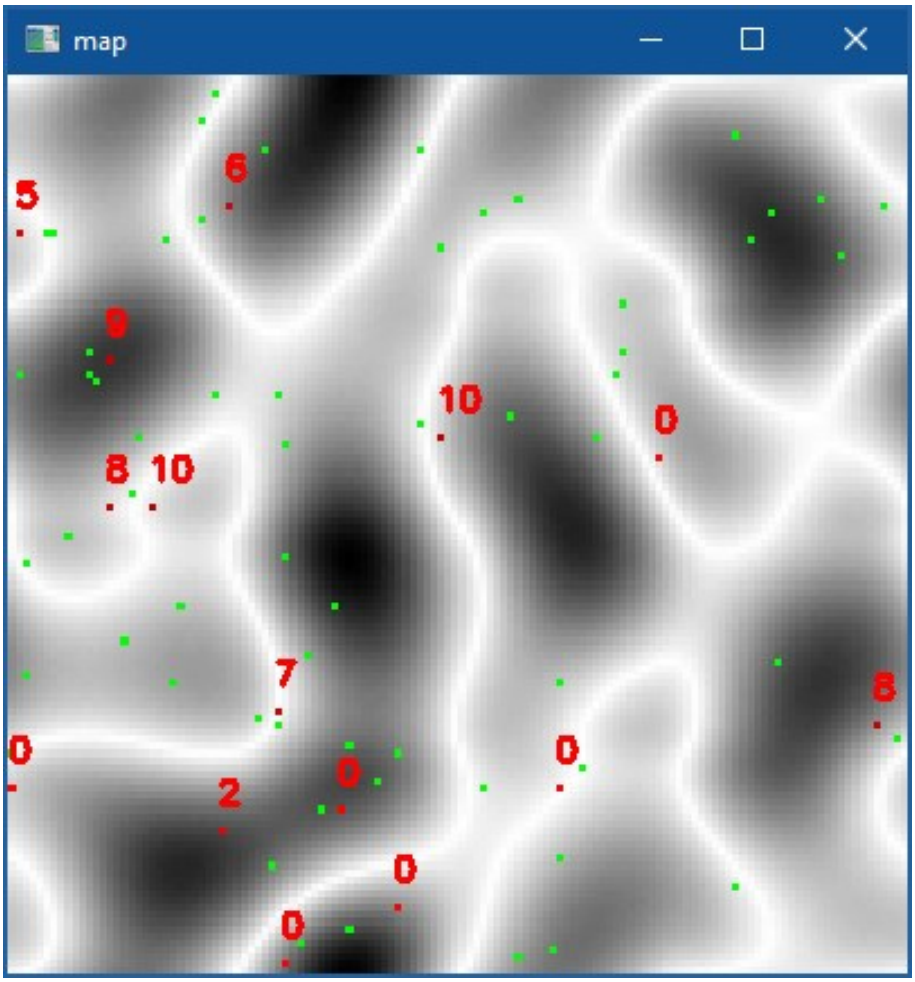
d_f – dystans między agentem, a najbliższym pożywieniem
d_{max} – zasięg widzenia agenta
f – mnożnik nagrody za dystans

Dystans między agentami, a pożywieniem jest liczony zarówno w osiach poziomych, jak i w osi pionowej.

3.6 Wizualna reprezentacja środowiska

Jednym z podstawowych wymogów środowiska uczenia przez wzmacnianie jest możliwość przedstawienia działania środowiska i efektów uczenia agentów w formie zrozumiałej dla człowieka. W tym celu środowiska standardu Gym najczęściej posiadają metodę pozwalającą na renderowanie obrazu przedstawiającego aktualny stan środowiska. Ta metoda jest wywoływana najczęściej co 1 lub kilka kroków i może przybierać różne formy, na przykład tekstową lub obrazową.

Projekt używa biblioteki OpenCV w celu stworzenia graficznej reprezentacji środowiska. Jest ona wyświetlana w oknie o rozmiarze 400x400px, które jest otwarte przez cały czas funkcjonowania środowiska, jeśli zostanie wybrana opcja renderowania graficznego (*render_mode=”human”*).



Rys. 3.14 Wizualna reprezentacja środowiska

Funkcja wyświetla mapę środowiska z naniesionymi na nią agentami (czerwone kwadraty) i pożywieniem (zielone kwadraty), co przedstawia rys. 3.14. Nad każdym agentem znajduje się wartość oznaczająca jego aktualną głębokość. Wyświetlana mapa funkcjonuje w formie reprezentacji głębokości, gdzie jaśniejsze punkty oznaczają mniejszą głębokość, zaś ciemniejsze oznaczają głębsze strefy.

Użycie renderowania wiąże się z olbrzymią utratą wydajności, więc nie należy go aktywować w momencie szkolenia ani potencjalnej analizy. Wyświetlenie powinno być zarezerwowane jedynie na wizualną kontrolę działania środowiska lub wyszkolonych modeli.

4. Algorytmy szkolenia i wyniki treningu agentów

Trening agentów w środowiskach uczenia przez wzmacnianie może być prowadzony przy użyciu wielu różnych algorytmów przedstawionych w wersji teoretycznej. Do przeprowadzenia testów projektu została wykorzystana biblioteka RL Stable-Baselines3, zastosowana została metoda aktor-krytyk PPO oraz podjęto próbę szkolenia A2C. Wyniki szkolenia algorytmem A2C okazały się znacząco gorsze, ponieważ agenci nie uczyli się dynamiki środowiska w sposób wystarczająco skuteczny.

Trening był wykonany na laptopie Zephyrus Duo 15 SE z następującymi parametrami:

- Procesor: AMD Ryzen 9 5900HX
- Karta graficzna: Nvidia RTX 3080 Mobile 16GB 115W
- Pamięć RAM: 32GB 3200MHz
- System operacyjny: Windows 10 Pro
- Wersja Python: 3.10.7

Algorytmy uczenia wraz z wynikami są dostępne w publicznym repozytorium GitHub autora [18].

4.1 Ogólna implementacja algorytmu uczenia

Agenci byli szkoleni przy użyciu algorytmu PPO w podstawowej implementacji algorytmu PPO dla środowiska równoległego ParallelEnv sugerowanej przez dokumentację PettingZoo [14]. Zastosowane w niej metody były w dużej mierze niezmienione, wprowadzono jednak modyfikacje dające więcej kontroli nad szkoleniem. Używanymi bibliotekami były PettingZoo, Stable-Baselines3 oraz SuperSuit. W implementacji znalazły się 2 najważniejsze metody – trening oraz ewaluacja wytrenowanego modelu. Wśród możliwych do modyfikacji parametrów znalazły się:

- Współczynnik dyskontowy (*gamma*)
- *Learning rate* oraz możliwość włączenia zmiennego *learning rate*
- Ilość i konfiguracja warstw sieci aktora oraz krytyka
- Ilość kroków szkolenia modelu
- Algorytm (PPO lub A2C)
- Wybór funkcji aktywacji: *ReLU* lub *Leaky ReLU*
- *n_steps* (opisane później)
- *Batch size*

```
def train(env_fn,
          g=0.99,
          lr=1e-3,
          net_arch=[64, 64, 32, 32],
          steps: int = 1_000_000,
          algorithm="ppo",
          leaky=True,
          n_steps = 1600,
          batch_size=32,
          lr_schedule=False,
          seed: int | None = 0,
          **env_kwargs):
    env = env_fn.parallel_env(**env_kwargs)
    env.reset(seed=seed)
    env = ss.black_death_v3(env=env)
    env = pettingzoo_env_to_vec_env_v1_black_death(env)
    env = ss.concat_vec_envs_v1(env, 10, num_cpus=1, base_class="stable_baselines3")
```

Rys. 4.1 Metoda treningu – wstęp

Metoda treningu przyjmuje jako argumenty parametry szkolenia algorytmu RL i potencjalnie parametry środowiska (rys. 4.1). Jej działanie rozpoczyna się od inicjalizacji oraz zrestartowania środowiska, obłożonego następnie wrapperami, których celem jest umożliwienie śmierci agentów oraz kompatybilności ze Stable-Baselines3. Wrappery dostarczane przez SuperSuit są wymagane, by środowisko mogło być użyte w szkoleniu ze standardowymi bibliotekami.

```
if algorithm == "ppo":
    model = PPO(
        MlpPolicy,
        env,
        n_steps= n_steps,
        verbose=3,
        gamma=g,
        learning_rate=lr,
        device="cpu",
        batch_size=batch_size,
        policy_kwargs=policy_kwargs,
        tensorboard_log=log_file
    )
elif algorithm == "a2c":
    model = A2C(
        MlpPolicy,
        env,
        verbose=3,
        gamma=g,
        device="cpu",
        learning_rate=lr,
        n_steps=n_steps,
        policy_kwargs=policy_kwargs,
        tensorboard_log=log_file)
```

Rys. 4.2 Metoda treningu – tworzenie modelu polityki

Po obłożeniu środowiska wrapperami tworzony jest model polityki PPO lub A2C (rys. 4.2), przekazywane są parametry takie jak *gamma*, *learning rate* czy argumenty polityki. Te ostatnie pozwalają na użycie niestandardowych sieci neuronowych dla aktora oraz krytyka.

```
model.learn(total_timesteps=steps)
model.save(log_file)
print("Model has been saved.")
print(f"Finished training on {str(env.unwrapped.metadata['name'])}.")
env.close()
```

Rys. 4.3 Metoda treningu – uczenie i zapisanie modelu

Model jest szkolony na odpowiedniej liczbie kroków (rys. 4.3). Po wyszkoleniu jest on zapisywany w określonym pliku. Czas szkolenia zależy przede wszystkim od:

- Ilości środowisk działających równolegle (wrapper *concat_vec_envs*)
- Ilości agentów w środowisku, którzy uczą się jednocześnie
- Wybranego algorytmu
- Użytych parametrów *batch_size* oraz *n_steps*

```
def eval(env_fn, num_games: int = 100, policy="latest", render_mode: str | None = None, **env_kwargs):
    env = env_fn.env(render_mode=render_mode)
    act_dict = {}
    print(
        f"\nStarting evaluation on {str(env.metadata['name'])}, policy {policy}, (num_games={num_games}, render_mode={render_mode})"
    )
    if policy == "latest":
        try:
            latest_policy = max(
                glob.glob(f"{env.metadata['name']}*.zip"), key=os.path.getctime
            )
        except ValueError:
            print("No policy found.")
            exit(0)
    elif policy == "ppo":
        try:
            latest_policy = max(
                glob.glob(f"{env.metadata['name']}*ppo*.zip"), key=os.path.getctime
            )
        except ValueError:
            print("PPO Policy not found.")
            exit(0)
    elif policy == "a2c":
        try:
            latest_policy = max(
                glob.glob(f"{env.metadata['name']}*a2c*.zip"), key=os.path.getctime
            )
        except ValueError:
            print("A2C Policy not found.")
            exit(0)
    if "ppo" in latest_policy:
        model = PPO.load(latest_policy)
    elif "a2c" in latest_policy:
        model = A2C.load(latest_policy)
```

Rys. 4.4 Metoda ewaluacji – wczytywanie polityki

Metoda ewaluacji wykorzystuje możliwość przekształcenia środowiska równoległego w sekwencyjne w celu łatwiejszej analizy każdego agenta, otrzymanej nagrody i podjętej akcji. Ewaluacja rozpoczyna się od wczytania najnowszej polityki (rys. 4.4) wybranego algorytmu.

```
rewards = {agent: 0 for agent in env.possible_agents}
actions = {}
for i in range(num_games):
    env.reset(seed=i)
    for agent in env.agent_iter():
        obs, reward, termination, truncation, info = env.last()

        for a in env.agents:
            rewards[a] += env.rewards[a]
        if termination or truncation:
            break
        else:
            act = model.predict(obs, deterministic=True)[0]

        if str(agent) in actions:
            actions[str(agent)].append(act.tolist())
        else:
            actions[str(agent)] = [act.tolist()]
        if str(act) in act_dict:
            act_dict[str(act)] += 1
        else:
            act_dict[str(act)] = 1
        print(agent, act)
        env.step(act)

env.close()
```

Rys. 4.5 Metoda ewaluacji – testowanie działania

Przy użyciu iteratora agentów, który jest częścią klasy *AECEnv*, każdy agent jest uruchamiany jeden po drugim. Jego obserwacje są przekazywane do modelu polityki, który następnie zwraca najlepszą dla agenta akcję (rys. 4.5). Jej wykonanie zwiększa odpowiadający licznik w celu późniejszej analizy, czy wszystkie akcje są podejmowane przez agentów oraz pokazania rozkładu akcji w trakcie prób. Środowisko jest na końcu czyszczone metodą zamknięcia (jedynie zamyka okna wizualnej reprezentacji, jeśli istnieją)

```
avg_reward = sum(rewards.values()) / len(rewards.values())
print("Rewards: ", rewards)
print(f"Avg reward: {avg_reward}")
print(act_dict)
result = {
    "rewards": rewards,
    "avg_reward": avg_reward,
    "action_dict": act_dict,
    "actions": actions
}
with open(f"{str(latest_policy)}.json", "w") as outfile:
    json.dump(result, outfile)
return result
```

Rys. 4.6 Metoda ewaluacji – zapis rezultatów

Na końcu wyświetlane są nagrody kumulacyjne dla każdego agenta, średnia nagroda kumulacyjna oraz ilość razy, które została wykonana każda akcja. Wyniki są zapisywane do pliku JSON w celu możliwej analizy.

4.2 Wyzwania w trakcie prób wyszkolenia agentów

Trenując agentów do wykonywania konkretnego celu przy użyciu uczenia przez wzmacnianie jednym z największych wyzwań okazał się problem balansu eksploracji i eksploatacji. Znalezienie prawidłowej równowagi nagród i kar okazało się bardzo skomplikowanym zadaniem, jednak próby dotarcia do tego celu dostarczyły znakomitych przykładów wyzwań stojących na drodze znalezienia optymalnych parametrów szkolenia. Problem eksploracji i eksploatacji objawia się przede wszystkim skłonnością agentów do wykonywania czynności znanych, przynoszących jak największą nagrodę natychmiastową lub jak najmniejszą karę zamiast akcji, które w odpowiedniej sytuacji oferują dużą nagrodę długofalową. W trakcie szkolenia występowały pułapki, w które wpadał algorytm próbując nauczyć się optymalnej polityki.

Pierwszą z nich było wykonywanie bez końca dwóch różnych akcji ruchu w celu zminimalizowania kary otrzymanej przez agenta. Takie zachowanie pojawiało się przede wszystkim w wyniku zbyt małego parametru dyskontowego w algorytmie uczącym. Agenci nie brali pod uwagę wystarczająco dalekich nagród i wykonywali akcje, które przynosiły im najmniejszą karę. Działo się tak ze względu na próbę uniknięcia wykonywania akcji nielegalnych generujących duże kary. Algorytm uznawał, że najlepszą polityką będzie poruszanie się w kółko akcjami minimalizującymi kary. Były to konkretnie dwie akcje ruchu, najczęściej w osi pionowej. Powtarzanie tylko jednej doprowadziłoby do otrzymania kary za próby wykonania nielegalnych akcji po dotarciu do granicy mapy lub kolizji z dnem. Więcej niż dwie akcje spowodowałyby przesuwanie agentów poza znany teren ruchu wynoszący dwa pola, który zapewniał bezpieczeństwo od kary za nielegalne akcje. Na podstawie wniosków ze szkolenia określono, że parametr dyskontowy powinien znajdować się w zakresie [0.97, 0.99] by agenci podejmowali logiczne decyzje.

Druga pułapka objawiała się w formie powtarzanej nieustannie akcji żywienia bez prób podejmowania jakichkolwiek innych akcji. Logika kierująca algorytmem uczenia dyktowała, że ograniczenie się jedynie do podejmowania akcji żywienia prowadzi do średniej nagrody wyższej niż inne alternatywne zachowania. Efekt taki był najczęściej obserwowany w przypadku, gdy nagroda za udaną akcję żywienia była zbyt wysoka i przewyższała kary za jej wielokrotne błędne wykonywanie.

Trzecia pułapka była ściśle związana z nagrodą za dystans. Agenci zamiast próbować dopłynąć do pożywienia i wykonać akcję żywienia, podpływali jedynie do pożywienia, po czym krążyli wokół niego nie podejmując próby spożycia go. Było to spowodowane przede wszystkim nieoptymalnym balansem między nagrodą za dystans, a nagrodą za zjedzenie pożywienia. Dodatkowym problemem, dla algorytmu okazało się znalezienie w odpowiednim czasie wystarczającej ilości „dowodów", które wskazywałyby na prawidłową korelację pomiędzy znajdowaniem się blisko pożywienia, a efektem akcji zdobywania pożywienia. Działo się tak ze względu na specyficzne warunki, które muszą zajść by akcja żywienia była legalna, a co za tym idzie dała nagrodę pozytywną.

4.3 Analiza procesu uczenia

W trakcie prac zostały wykonane szkolenia agentów algorytmem PPO, zaś proces uczenia został zapisany jako logi Tensorboard. Wstępnie sieci krytyka oraz aktora składały się z 2 warstw gęstych o 64 neuronach oraz 2 gęstych o 32 neuronach. Zastosowano funkcję aktywacji Leaky ReLU. Taka konfiguracja okazała się być najlepszą z testowanych wstępnie ze względu na lepsze zdolności generalizacji.

Parametr *batch_size* początkowo ustalony na wartość 256, okazał się być zbyt duży. Jego zmniejszenie do 32 pozwoliło na lepszą analizę przez algorytm doświadczeń agentów, co udoskonaliło zdolności generalizacji modelu.

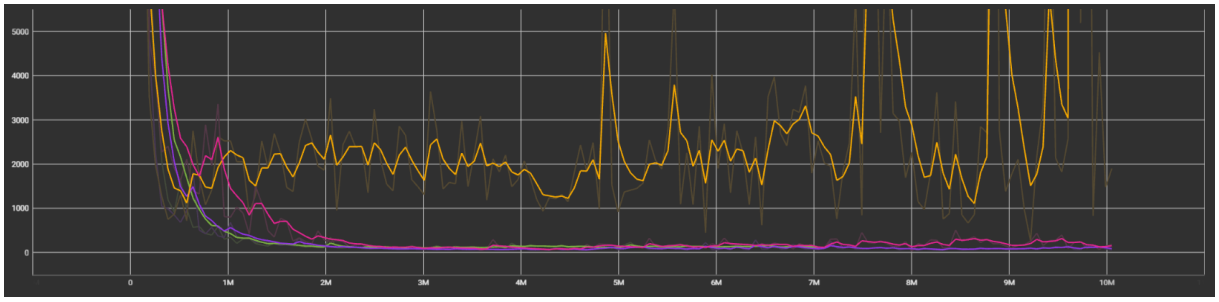
Następnym bardzo ważnym parametrem był *n_steps*. W Stable-Baselines3 omawiany parametr dyktuje długość epizodów branych przez algorytm do analizy. W przypadku jego wartości standardowej, algorytm rozpatrywał doświadczenia agentów zbyt punktowo, co uniemożliwiło stworzenie polityki z dobrym połączeniem przyczynowo-skutkowym. Zmiana parametru *n_steps* na wartość 1600 przyniosła dużo lepsze efekty, gdyż algorytm był w stanie znaleźć miejsca, w których agent podejmował z sukcesem akcję karmienia. Wartość 1600 została wybrana jako przeciętna długość trwania epizodu środowiska.

$$n\_updates = \frac{total\_timesteps}{n\_steps \cdot n\_envs} \tag{4.1}$$

n_updates – ilość aktualizacji polityki i funkcji wartości w trakcie szkolenia
total_timesteps – całkowita ilość kroków szkolenia
n_steps – ilość kroków brana pod uwagę na raz przez algorytm
n_envs – ilość środowisk działających równolegle obok siebie, tutaj 10

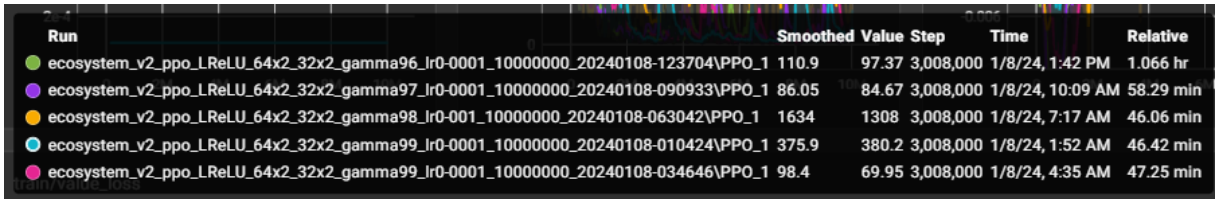
Standardową praktyką w implementacji PPO przy użyciu Stable-Baselines3 jest ustawienie *n_steps* na wartość, która jest podzielna przez *batch_size*. Formuła (4.1) pokazuje, że od parametru *n_steps* zależy ilość aktualizacji polityki w trakcie uczenia [16].

4.3.1 Ogólne obserwacje z wcześniejszych partii uczenia



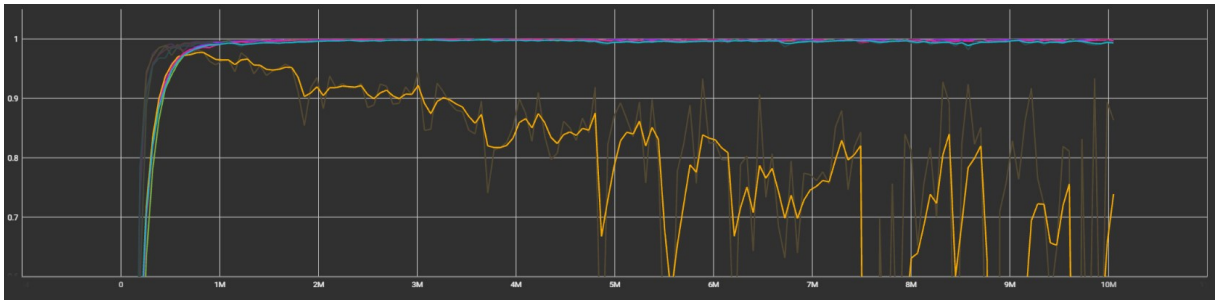
Rys. 4.7 Value Loss dla wczesnych partii uczenia

Obserwując wykres *value loss* (rys. 4.7) można zobaczyć efektywność szkolenia z wyżej wymienionymi parametrami. Jednym z pierwszych rzucających się w oczy odstępstw jest żółty graf. Jest to partia szkolenia, w której wartość *learning rate* wynosiła $1x10^{-3}$, zamiast standardowego $1x10^{-4}$ w innych próbach. Stwierdzono też negatywny wpływ szkolenia ponad 5 milionów kroków w większości przypadków.



Rys. 4.8 Value loss w kroku $3*10^6$

Co więcej, na rys. 4.8 przedstawione są wartości *value loss* dla kroku w okolicach 3 milionów. Można zaobserwować, że większość algorytmów dla różnych parametrów dyskontowych znajduje dobrą wartość już docierając do 3 milionów kroków, lub nawet wcześniej.



Rys. 4.9 Explained Variance dla wczesnych partii uczenia

Rys. 4.9 przedstawia *explained variance*, czyli wartość określającą, jak dobry jest model w porównaniu z wykonywaniem jedynie akcji bazowej. 0 oznacza skuteczność podobną do wybierania zawsze akcji bazowej, 1 oznacza teoretycznie perfekcyjny wybór. Wartości poniżej 0 są świadectwem, że model sprawuje się gorzej, niż podczas wybierania jedynie akcji

bazowej. Rys. 4.9 sugeruje, że model z *learning rate* 0.001 sprawował się najgorzej ze wszystkich, co ostatecznie nie okazało się prawdą. Cała reszta modeli osiągnęła wartości bliskie 1, co oznaczało, że powinny być optymalnie wyszkolone dla znalezionej przez nie polityki. Oceniając efekty szkolenia jedynie za pomocą wartości *explained variance*, szkolenie powinno zostać przerwane w okolicach kroku $3*10^6$.

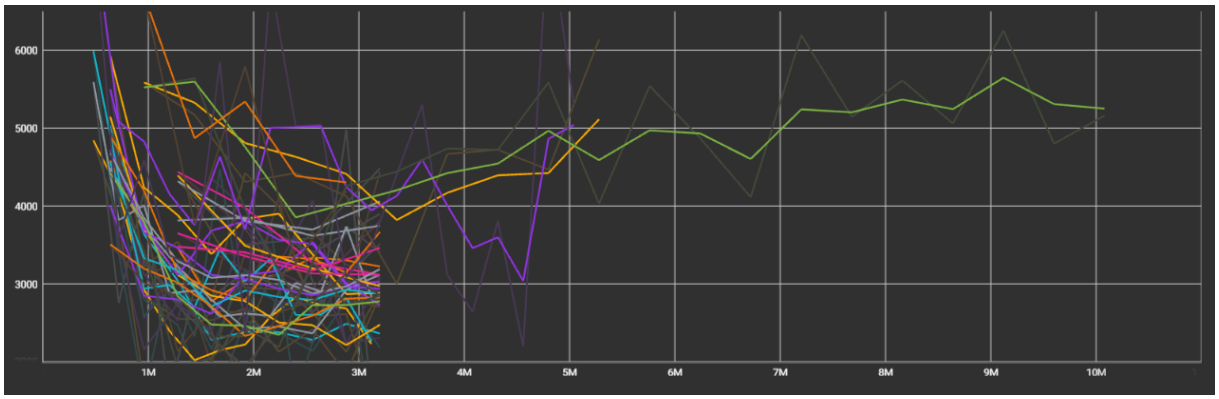
Niestety, mimo że wykresy szkolenia wskazują na prawidłowe wyniki, w przypadku uczenia przez wzmacnianie nie jesteśmy w stanie określić skuteczności wyszkolonej polityki jedynie na podstawie logów. W rzeczywistości model w trakcie szkolenia może znaleźć błędną politykę i udoskonalać ją w kierunku minimalizacji kar zamiast zdobywania optymalnych nagród. Innym zagrożeniem jest również znalezienie polityki pozytywnej, lecz nieoptymalnej w porównaniu z najlepszą możliwą. Najprostszą pułapką, w którą wpadał algorytm w trakcie uczenia było skupianie się jedynie na nagrodzie za dystans i stopniowa eliminacja prób żywienia się.

Najlepszym przykładem potwierdzającym brak skuteczności oceny jedynie przy użyciu wykresów są wyniki szkolenia. Algorytm używający *learning rate* o wartości 0.001 jako jedyny osiągnął główny cel agentów – nurkowanie do pożywienia. Zachowanie agentów szkolonych tym algorytmem przejawiało przez większość czasu chaotyczny brak zrozumienia obserwacji, lecz w pozornie losowych momentach wykazywało cechy zrozumienia dynamiki środowiska. Agenci wykonywali losowe akcje, lecz jeśli znaleźli się blisko pożywienia, nurkowali do głębokości pozwalającej go dosięgnąć. Wykonywali jednorazowo akcję żywienia zdobywając tym samym nagrodę, a następnie wracali na powierzchnię. Szczególnie ciekawe jest właśnie wspomniane wynurzenie, gdyż jest ono prawdopodobnie taktyką opracowaną przez algorytm, która pozwala na ominięcie dna morskiego w trakcie eksploracji.

Pozostałe algorytmy używające mniejszych wartości *learning rate* prezentują zgoła inną taktykę, która opiera się jedynie na zbliżeniu do pożywienia w celu otrzymania nagrody za dystans, lecz bez żadnych prób wykonania akcji żywienia/ataku na innych agentach czy pożywieniu. W efekcie tacy agenci kończą swój żywot ze sporą karą za śmierć, co czyni tą taktykę wielce nieefektywną.

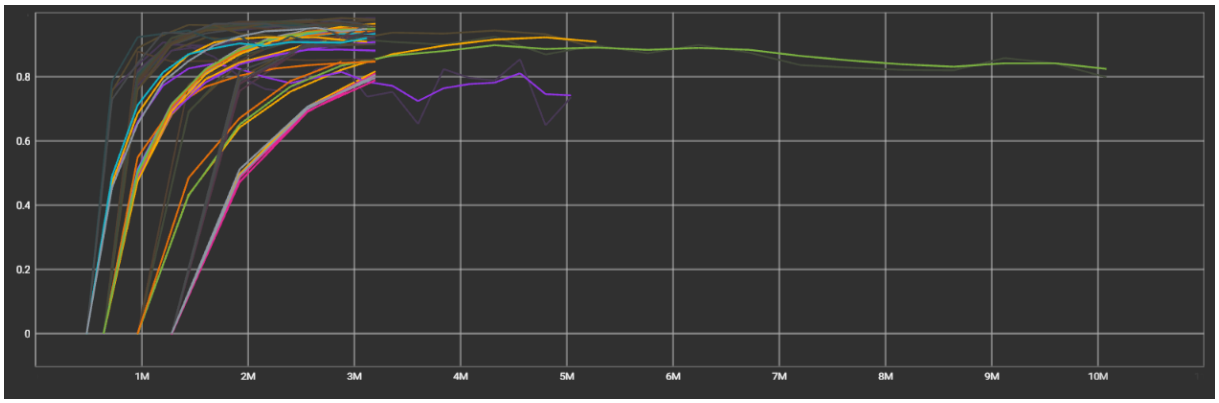
4.3.2 Metoda uczenia ulepszona na podstawie wcześniejszych obserwacji

Wyciągając wnioski z poprzednich szkoleń, autor pracy podjął nowe badania. Poprzez testy różnych parametrów udało się osiągnąć 3 polityki, które mają większy sukces w środowisku oraz przejawiają lepsze nagrody. *Learning rate* został ustalony na 0.001, gdyż przy tej wartości agenci podejmowali próby korzystania z pożywienia.



Rys. 4.10 Value Loss dla ulepszonej partii uczenia

Jak widać na rys. 4.10, nowe metody szkolenia ograniczone zostały do czasu 3 milionów kroków, aczkolwiek przeprowadzono również próby szkolenia do 10 milionów. Najczęściej jednak wyszkolona polityka osiągała najlepsze wartości właśnie w okolicach 3 milionów kroków, zaś dalsze szkolenie pogarszało wyniki.



Rys. 4.11 Explained Variance dla ulepszonej partii uczenia

Na rys. 4.11 przedstawiono wyniki *explained variance* dla nowo wyszkolonych modeli. Ewidentnym jest, że nie osiągają one tak szybko wartości bliskich 1.0 ze względu na inne postrzeganie środowiska przez model. Jest to efekt przede wszystkim zmiany wartości *learning rate, n_steps* oraz *batch_size*. Widocznym jest też jednak, że wiele modeli w momencie zakończenia swojego szkolenia w 3 milionach kroków, dalej poprawiało swoje wyniki, co wskazuje, że lepszym byłoby wybranie 5 milionów kroków jako optymalnej długości

procesu szkolenia. Wykresy dla modeli szkolonych dłużej pokazują, że po 5 milionach kroków wartość *explained variance* spada, więc dalsze uczenie miałoby negatywne skutki.

Ta faza szkolenia doprowadziła do uzyskania rezultatów bardziej zbieżnych z oczekiwanymi. Agenci skupiali się na aktywności związanej z pozyskiwaniem pożywienia lub atakowaniu siebie nawzajem. Warto jednak zaznaczyć, iż agenci nie szukali bezpośrednio okazji do ataku - dochodziło do niego głównie, gdy agenci znajdowali się blisko siebie w trakcie poszukiwań pożywienia. Należy przy tym nadmienić, że większość modeli wyszkolonych w tej partii nie było w pełni sprawnych - często powtarzały te same akcje lub miały trudności w pokonywaniu zróżnicowanego terenu dna.

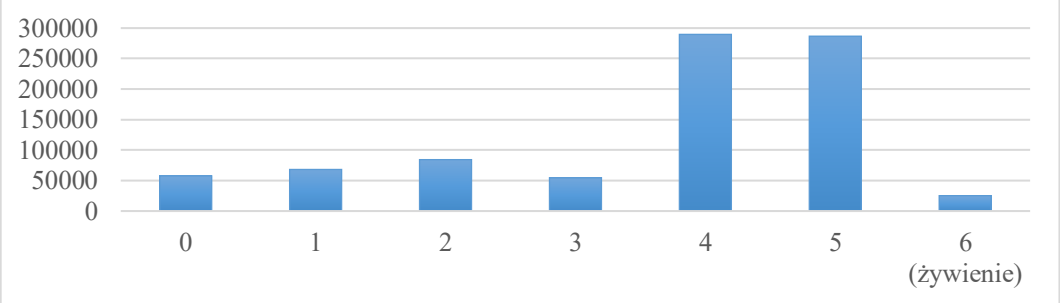
4.4 Porównanie wyszkolonych modeli

4.4.1 Najlepsza osiągnięta polityka

Najlepszą politykę osiągnięto używając następujących parametrów:

- Ilość kroków: 3 000 000
- Architektura sieci: 1 warstwa 128-neuronowa, 2 warstwy 64-neuronowe, 1 warstwa 32-neuronowa, funkcja aktywacji: *Leaky ReLU*
- *Learning rate*: 0.001
- *n_steps*: 1600
- *batch_size*: 32
- Współczynnik dyskontowy: 0.99

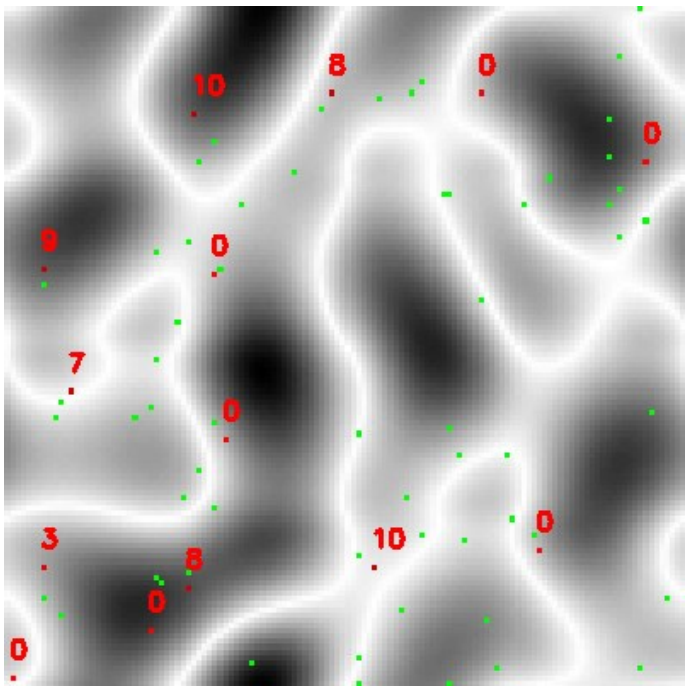
Ta polityka przejawia bardzo dużą przewagę nad innymi, agenci dużo częściej wykonują prawidłowe akcje prowadzące do znalezienia pożywienia, innego agenta lub nawigacji terenu, unikając jednocześnie akcji nielegalnych w ich stanie.



Rys. 4.12 Rozkład akcji dla 100 gier ewaluujących najlepszą politykę

W trakcie ewaluacji polityki na 100 grach (rys. 4.12), akcja żywienia wykonywana była najrzadziej, co jest prawidłowym zachowaniem – powinna być wykonywana jedynie, gdy

pożywienie lub inny agent znajduje się w zasięgu. Reszta akcji służyła do manewrowania w terenie i poruszania się w stronę źródeł pożywienia. Widoczna jest jednak przewaga akcji płynięcia w górę i dół, które agenci wykonywali, gdy nie widzieli lepszego wyjścia. Przeciętna nagroda agenta wynosiła -488,498 na grę, co wskazuje, że agenci mimo, iż nie byli w stanie w efektywny sposób osiągać długiego życia poprzez zdobywanie pożywienia, prezentowali ku temu dobre skłonności.



Rys. 4.13 Wizualna reprezentacja najlepszej polityki

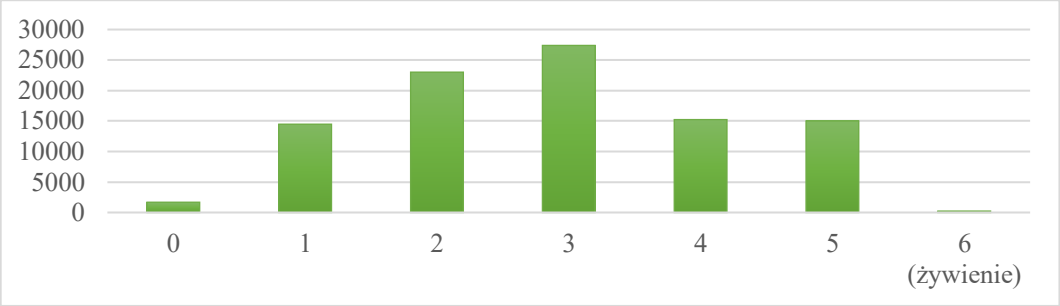
Analiza wizualna działania polityki pokazuje, że agenci ewidentnie dążą w kierunku pożywienia (rys. 4.13) kierowani zarówno nagrodą za dystans, jak i motywacją płynącą z punktów pożywienia. Używający tej polityki agenci wykonują też akcje ataku innych agentów, jeśli tylko nadarzy się taka okazja, lecz nie szukają aktywnie konfrontacji. Liczby nad agentami (oznaczające ich głębokość) wskazują na niechęć do podejmowania ryzyka nurkowania, gdyż wiąże się to z niebezpieczeństwem kolizji z dnem i otrzymaniem kary za próbę wykonania akcji nielegalnej. Zamiast tego agenci poruszali się przede wszystkim w okolicach górnych partii dna szukając płytko położonego pożywienia.

4.4.2 Druga najlepsza polityka

Mimo, że najlepsza metoda używała innych parametrów szkolenia, ogólnie skuteczne okazało się zwiększenie *n_steps* do 3200. Druga najlepsza polityka podobnie jak pierwsza szukała pożywienia, lecz przejawiała mniejszą świadomość otaczającego terenu i obiektów.

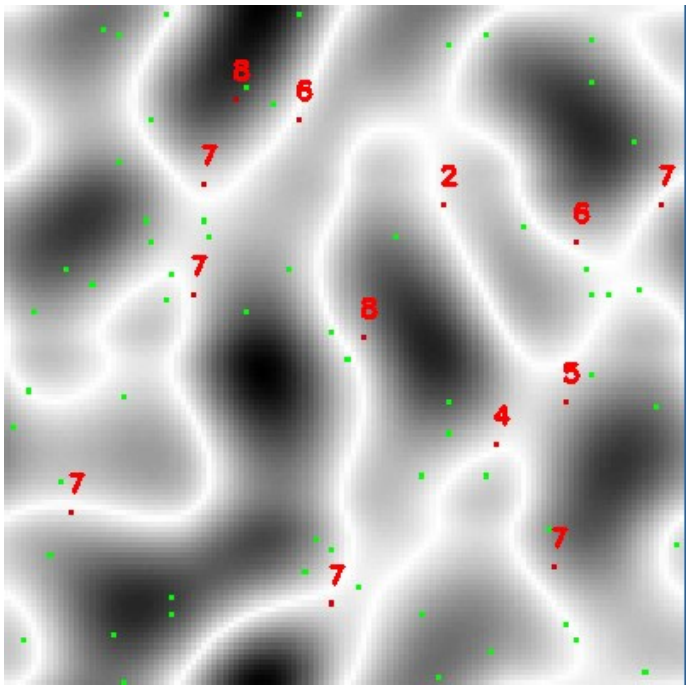
Do wyszkolenia metody zostały użyte następujące parametry:

- Ilość kroków: 3 000 000
- Architektura sieci: 2 warstwy 64-neuronowe, 2 warstwy 32-neuronowe, funkcja aktywacji: *Leaky ReLU*
- *Learning rate*: 0.001
- *n_steps*: 3200
- *batch_size*: 32
- Współczynnik dyskontowy: 0.99



Rys. 4.14 Rozkład akcji dla 100 gier ewaluujących drugą najlepszą politykę

Przedstawiony na rys. 4.14 rozkład akcji przedstawia dużo bardziej zbalansowany rozkład decyzji niż w przypadku pierwszej polityki – akcje ruchu w pionie przestały być dominującą częścią, a ruch w osi X stał się najczęściej wykonywaną akcją. Akcja żywienia wykonywana bardzo sporadycznie sugeruje prawidłowe zrozumienie zależności położenia pożywienia i legalności akcji żywienia/ataku, lecz może też wskazywać na rzadkie spełnianie warunków potrzebnych do wykonania jej z sukcesem. Przeciętna nagroda na grę osiągana przez agentów wyniosła -521,544, co potwierdza drugą teorię odnośnie sporadycznego wykonywania akcji żywienia.



Rys. 4.15 Wizualna reprezentacja drugiej najlepszej polityki

Analiza wizualna działania potwierdza problemy płynące z rozkładu akcji – agenci rzadziej wykonują akcje prowadzące ich w stronę żywienia, a wiele z podejmowanych decyzji jest nieoptymalne dla stanu, w którym się znajdują. Prezentują jednak dobre rozumienie zależności akcji żywienia od dystansu do jedzenia i często decydują się na spożycie pokarmu, jeśli znajdują się blisko niego.

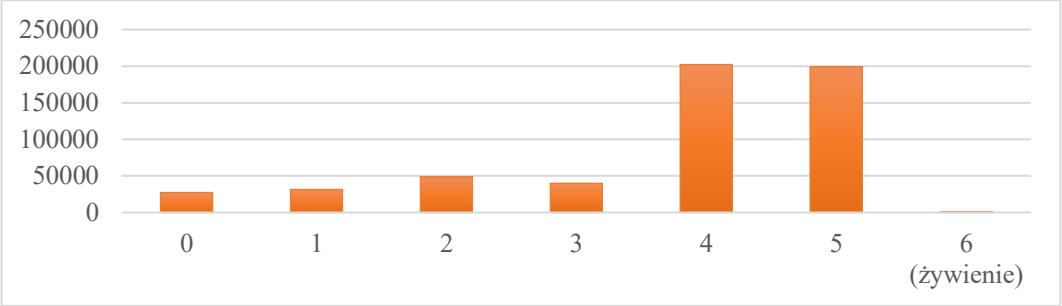
4.4.3 Trzecia najlepsza polityka

Trzecia, jeśli chodzi o skuteczność polityka jest znakomitym przykładem dysonansu między rezultatami otrzymanymi z analizy gier, a weryfikacją wizualną. Mimo, iż nie przejawia dobrej skuteczności, jeśli chodzi o otrzymaną nagrodę, podejmowane przez agentów decyzje we właściwych warunkach są lepsze od pozostałych polityk.

Do wyszkolenia metody zostały użyte następujące parametry:

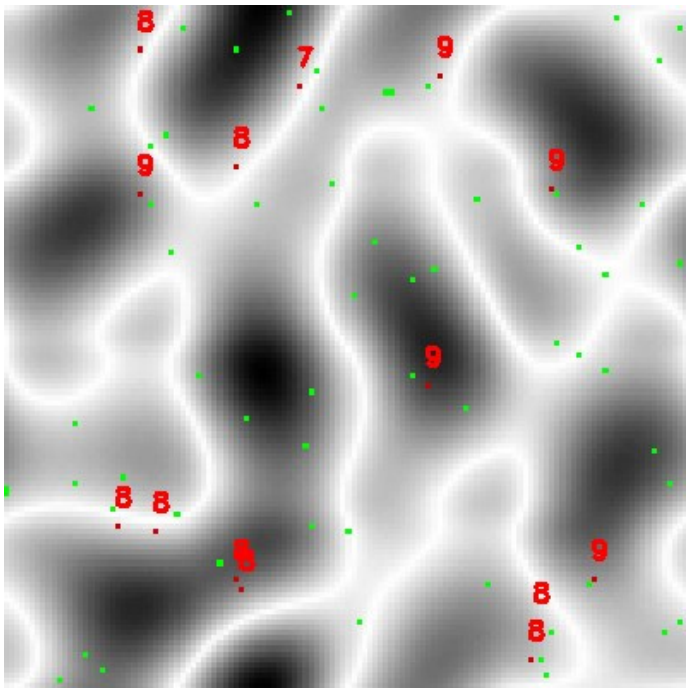
- Ilość kroków: 3 000 000
- Architektura sieci: 2 warstwy 64-neuronowe, 2 warstwy 32-neuronowe, funkcja aktywacji: *Leaky ReLU*
- *Learning rate*: 0.001
- *n_steps*: 3200
- *batch_size*: 64
- Współczynnik dyskontowy: 0.99

Dodatkową zmianą zastosowaną w trakcie uczenia była zmiana mnożnika nagrody za dystans do wartości 4.0.



Rys. 4.16 Rozkład akcji dla 100 gier ewaluujących trzecią najlepszą politykę

Z rozkładu akcji (rys. 4.16) można zauważyć, że agenci wykonują przede wszystkim te akcje, które zapobiegają otrzymaniu kary za próbę podjęcia akcji nielegalnej. Rozkład wykonanych przez agentów akcji pokazuje jednak, że istnieje dobry balans między akcjami ruchu poziomego. Akcja żywienia jest wykonywana bardzo rzadko, co wskazuje na zrozumienie relacji między dystansem do pożywienia, a legalnością akcji. Mimo tego, przeciętna nagroda całkowita agenta za grę wyniosła -1486,691, co jest dużo gorszym wynikiem.



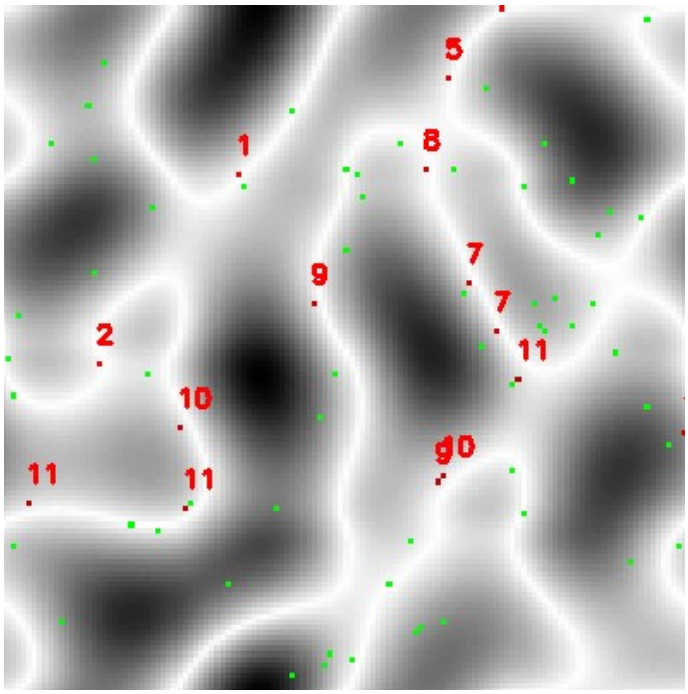
Rys. 4.17 Wizualna reprezentacja trzeciej najlepszej polityki

Obserwując analizę wizualną (rys. 4.17) można jednak zauważyć, że agenci wyszkoleni w tej partii prezentowali lepsze zrozumienie lokalizacji pożywienia. Mieli tendencję do poruszania się w jego stronę i podpływania relatywnie blisko niego, lecz w bezpiecznej odległości od minimalnego poziomu dna. Jako że agenci nie decydowali się zaryzykować zejścia głębiej, rzadko kiedy znajdowali w swoim zasięgu pożywienie, które mogliby spożyć. Skupiają się zamiast tego przede wszystkim na maksymalizacji nagrody za dystans

do najbliższego pożywienia, co widać na przykładzie analizie wizualnej – każdy agent jest blisko pożywienia, lecz zbyt daleko, by móc go spożyć.

4.4.4 Próby szkolenia algorytmem A2C

Wśród wielu prób szkolenia przy użyciu algorytmu PPO, podjęto również próby szkolenia metodą A2C w implementacji Stable-Baselines3. Ta metoda z reguły okazywała się być około 152,6% wolniejsza niż PPO, zaś jej wyniki były poważnie gorsze, agenci wyszkoleni przy użyciu algorytmu A2C mieli tendencję do wykonywania w kółko jednej akcji ruchu poziomo.



Rys. 4.18 Zachowanie agentów wyszkolonych algorytmem A2C

Na rys. 4.18 przedstawione zostało zachowanie agentów wyszkolonych wspomnianą metodą. Niemal każdy z nich znajduje się z lewej strony przeszkodu terenowej, zaś w prawej krawędzi okna widać następnych dwóch agentów. Wyszkoleni algorytmem A2C agenci powtarzali nieustannie akcję płynięcia w prawą stronę, więc ich skuteczność była zerowa. Większość prób szkolenia agentów tą metodą prezentowała podobne efekty. Lepsze dopasowanie hiperparametrów prawdopodobnie umożliwiłoby osiągnięcie zadowalającej polityki, lecz ze względu na dużo lepszą skuteczność metody PPO, dalsze próby zostały porzucone.

Konkretna polityka przedstawiona w tym rozdziale została wyszkolona z użyciem następujących parametrów:

- Ilość kroków: 5 000 000
- Architektura sieci: 1 warstwa 128-neuronowa, 2 warstwy 64-neuronowe, 1 warstwa 32-neuronowa, funkcja aktywacji: *Leaky ReLU*
- *Learning rate*: 0.001
- *n_steps*: 3200
- *batch_size*: 16
- Współczynnik dyskontowy: 0.99

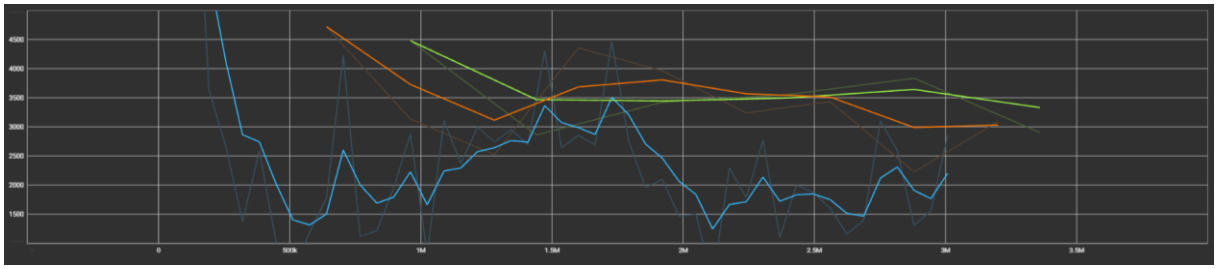
Rys. 4.19 Rozkład akcji dla 100 gier ewaluujących przykładową politykę A2C

Na rys. 4.19 przedstawiono rozkład akcji dla 100 iteracji środowiska używając omawianej polityki. Akcję 0 wykonano 592199 razy, akcja 4 została powtórzona 8337 razy, zaś akcja 1 zaledwie 564. Pozostałe akcje nie zostały w trakcie 100 iteracji ani razu podjęte. Przeciętna nagroda kumulacyjna agenta wynosiła -7 409,945 na grę, co wskazuje na całkowity brak zrozumienia nielegalności akcji. Ich powtarzanie doprowadzało agentów do otrzymywania poważnych kar, co przekładało się na wyjątkowo zły wynik.

4.4.4 Porównanie procesu szkolenia najlepszych polityk

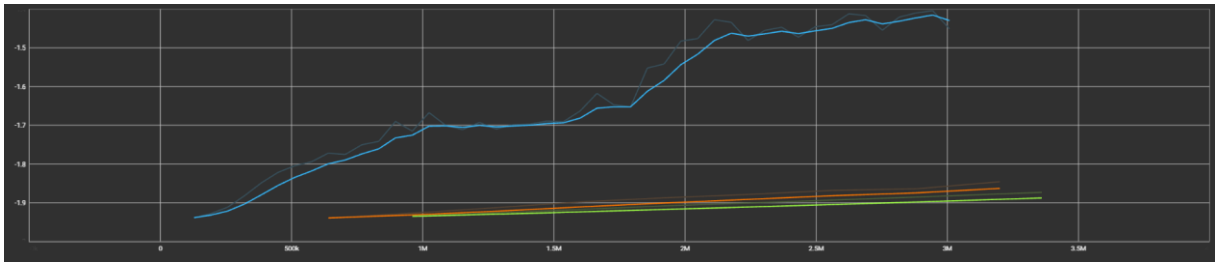
Dla poniższych wykresów zastosowano kolorystykę odpowiadającą wykresom polityk w podrozdziałach 4.4.1-4.4.3, czyli:

1. Najlepsza polityka – niebieski
2. Druga najlepsza polityka – zielony
3. Trzecia najlepsza polityka – pomarańczowy



Rys. 4.20 Value Loss dla trzech najlepszych polityk

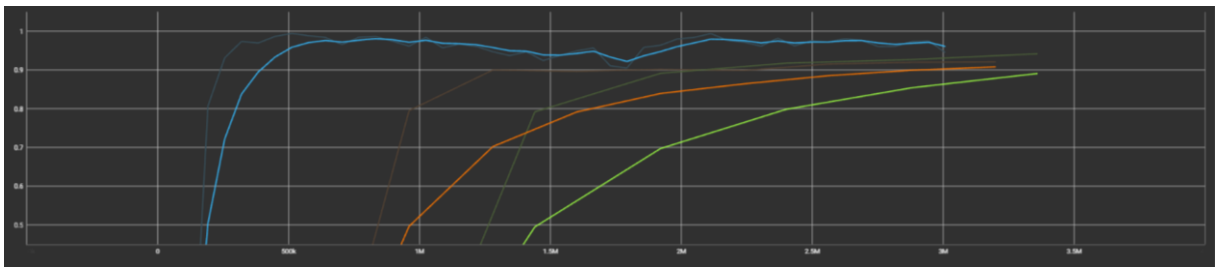
Wykres *value loss* przedstawiony na rys. 4.20 pokazuje wyraźną różnicę pomiędzy szkolonymi politykami. Najlepsza posiada wykres, który dosyć szybko osiągnął niskie wartości, lecz następnie chaotycznie one wzrastały i opadały. Pozostałe dwie polityki prezentują dużo bardziej stonowane i przewidywalne zachowanie, lecz potencjalnie mogą zostać na swoim poziomie. Nawet jeśli pozwoliłoby się na dłuższe szkolenie, mogą nigdy nie poprawić swoich wyników. Z reguły stopniowe i stabilne zmniejszanie *value loss* jest bardziej pożądane, lecz w zastosowaniach RL może to wskazywać na zbytnią generalizację i brak zrozumienia dynamiki środowiska.



Rys. 4.21 Entropy Loss dla trzech najlepszych polityk

Analizując wartości *entropy loss* (rys. 4.21) możemy określić jaką dana polityka ma znajomość stanów oraz poziom eksploracji. Im niższa jest wartość *entropy loss*, tym algorytm jest mniej pewny swojej znajomości stanów i przeprowadza więcej eksploracji. Dzięki *entropy loss* można zapobiec zbyt wczesnemu znalezieniu optimum lokalnego przez model. Wartości prezentowane na wykresie wskazują na sporą niepewność stanów polityki drugiej oraz trzeciej, które bardzo powoli się zmniejszają (wartości wzrastają, dążą ku 0). Najlepsza polityka dużo szybciej zmniejszała swoją niepewność. Prawdopodobnie dalsze

szkolenie poprawiło by jej wyniki jeszcze bardziej, lecz już teraz prezentuje ona dużo lepszą świadomość w porównaniu z pozostałymi modelami.



Rys. 4.22 Explained Variance dla trzech najlepszych polityk

Rys. 4.22 prezentuje wartości *explained variance*. Polityka pierwsza bardzo szybko osiągnęła wartości bliskie 1.0, po czym została na tym poziomie z lekkimi fluktuacjami. Znaczącym jest fakt, że osiągnęła ona swoje optimum lokalne, aczkolwiek nie była w stanie znaleźć lepszego planu działania przez cały czas szkolenia. Sprawowała się lepiej niż pozostałe opcje, lecz nie prezentowała tendencji do dalszej poprawy. Pozostałe dwie polityki poprawiały swoje wartości *explained variance* i prawdopodobnie czyniłyby to dalej. Trudno jednak określić, czy przyniosłoby to duże zyski.

4.4.5 Wpływ parametrów na efekty szkolenia

Po wyszkoleniu polityk z różnymi parametrami szkolenia, największy wpływ na wyniki okazały się mieć:

- *n_steps*, których zwiększenie znacząco poprawiło zdolności rozumowania wyszkolonej polityki.
- *batch_size*, którego zmniejszenie pomogło polepszyć generalizację modeli
- Parametr dyskontowy, którego wartość drastycznie wpływała na zachowanie agentów, jego optymalna wartość wynosiła 0.99.
- *Learning rate*, którego zbyt małe wartości powodowały słabe rozumowanie modelu, najlepszą wartością okazało się być 0.001.
- Nagroda za dystans w środowisku oraz jej balans z innymi nagrodami, jej zwiększenie stworzyło większą inicjatywę do osiągania prawidłowych wartości.

Wpływ pozostałych parametrów takich jak ilość neuronów poszczególnych warstw, ilość warstw lub funkcja aktywacji był znaczący, lecz ich zmiany nie powodowały zmian aż tak znaczących, jak wyżej wymienione opcje.

5. Podsumowanie i przyszłość projektu

W ramach pracy stworzono środowisko wieloagentowego uczenia przez wzmacnianie Ecosystem, które jest zgodne ze standardem PettingZoo i możliwe do użycia w większości bibliotek uczenia przez wzmacnianie bez modyfikacji ich funkcjonalności. Autorowi udało się zaimplementować agenta reprezentującego generyczną rybę, a także zapewnić zestaw akcji pozwalających na nawigację po środowisku oraz zdobywanie pożywienia w celu przedłużenia życia agenta i maksymalizacji nagrody kumulacyjnej. Stworzony jako część pracy algorytm zdobywania obserwacji umożliwia agentom analizę środowiska w sposób, w jaki przeprowadzałby to organizm żywy – na podstawie widoczności obiektów, ich dystansu oraz położenia względem agenta. Użytkownik ma możliwość zmiany ilości widocznych obiektów, zasięgu wzroku agenta oraz nagrody za zbliżanie się do najbliższego pokarmu.

Projekt pozwala na wczytanie dowolnego pliku mapy dna i skonfigurowanie jej granic. Umożliwia to użytkownikowi wczytywanie własnego terenu, jeśli potrzebna jest reprezentacja konkretnego miejsca, którym może być na przykład akwarium, jezioro czy nawet okolice wraku statku.

Środowisko jest czytelne i pozwala na proste dodawanie nowych możliwości takich jak akcje, inne sposoby obserwacji czy zmiana nagród i kar. Dzięki wbudowanej liście parametrów, większość zmian wprowadzanych przez użytkownika jest możliwa do zrobienia bez ingerencji w pliki środowiska.

W ramach pracy zostało przeprowadzone szkolenie agentów algorytmem PPO na środowisku, co poskutkowało stworzeniem polityki pozwalającej agentom na znajdowanie pożywienia oraz zapewniającej im akceptowalne zdolności nawigacji terenu. Wyniki szkolenia zostały przeanalizowane i opisane, zaś wnioski z wcześniejszych prób wyszkolenia modelu polityki posłużyły do poprawienia wyników następnych partii. Trzy najlepsze polityki zostały opisane i porównane między sobą w oparciu o logi Tensorboard, rozkłady akcji na podstawie 100 próbnych gier w środowisku oraz przeciętne nagrody uzyskiwane przez agentów. Przedstawiono próbę uczenia z użyciem algorytmu A2C. W czasie szkolenia osiągnięto zestaw wytrenowanych polityk, które prawidłowo podejmują decyzje na podstawie stanów, w których znajdują się agenci. Udało się uzyskać modele przedstawiające różne podejścia polityki zachowania do planu zdobywania pożywienia i maksymalizacji nagród. Efekty szkolenia doprowadziły do zrozumienia i opisania wpływu parametrów środowiska i algorytmów uczenia na sprawność modeli.

Przedstawione w pracy badania są wstępem do szerokiego projektu mającego na celu stworzenia wiarygodnej symulacji środowisk wodnych wraz z różnorodnością ich fauny i flory. Wzbogacenie pracy o nowe gatunki i ich interakcje umożliwiłoby badania nad ewentualnym wpływem modyfikowania składu gatunkowego w rzeczywistym zbiorniku wodnym. Dzięki wieloletniej analizie istniejących ekosystemów możliwe byłoby stworzenie ich odwzorowania w oparciu o efekty tej pracy.

Bibliografia

[1] Różanowski K. „Sztuczna inteligencja rozwój, szanse i zagrożenia”. (2007). *Zeszyty Naukowe Warszawskiej Wyższej Szkoły Informatyki*, *2*(2), 109-135.

[2] Chrzanowska A. „Uczenie maszynowe w Pythonie”. (2021). Praca licencjacka. *Uniwerstet Jagielloński.*

[3] Wereszczyńska K. „Wykorzystanie uczenia maszynowego do selekcji miejscowości i dróg na mapach przeglądowych”. (2022).

[4] Ghahremani-Nahr, Javid & Nozari, Hamed & Sadeghi, Mohammad Ebrahim. „Artificial intelligence and Machine Learning for Real-world problems (A survey)”. (2021). *International Journal of Innovation in Engineering*, *1*(3), 38-47. *DOI:10.59615/ijie.1.3.38*

[5] Sah, Shagan. “Machine learning: a review of learning types”. (2020). *DOI:10.20944/preprints202007.0230.v1*

[6] Konstantinos Mitsopoulos, Sterling Somers, Joel Schooler, Christian Lebiere, Peter Pirolli, Robert Thomson. “Toward a Psychology of Deep Reinforcement Learning Agents Using a Cognitive Architecture”. (2022). *Topics in cognitive science*, *14*(4), 756-779. *DOI:10.1111/tops.12573*

[7] Chen, Tong & Liu, Jiqiang & Xiang, Yingxiao & Niu, Wenjia & Tong, Endong & Han, Zhen. “Adversarial attack and defense in reinforcement learning-from AI security view”. (2019). *Cybersecurity*, *2*, 1-22. *DOI:10.1186/s42400-019-0027-x*

[8] Muddasar Naeem and Syed Tahir Hussain Rizvi and Antonio Coronato. “A gentle introduction to reinforcement learning and its application in different fields”. (2020). *IEEE access*, *8*, 209320-209344. *DOI:10.1109/ACCESS.2020.3038605*

[9] Krzysztof Sawka, „Uczenie maszynowe z użyciem Scikit-Learn i TensorFlow, wyd. II, aktualizacja do modułu TensorFlow 2”. (2020). *Helion.*

[10] Dawid Kwapisz. “Implementacja prostej gry zręcznościowej i algorytmu korzystającego z uczenia (ze wzmocnieniem) uczącego się grać w tę grę”. (2022). Praca inżynierska. *Akademia Górniczo Hutnicza im. Stanisława Staszica w Krakowie.*

[11] Arulkumaran, Kai and Deisenroth, Marc Peter and Brundage, Miles and Bharath, Anil Anthony. “A brief survey of deep reinforcement learning”. (2017). *DOI:10.1109/msp.2017.2743240*

[12] Greg Brockman and Vicki Cheung and Ludwig Pettersson and Jonas Schneider and John Schulman and Jie Tang and Wojciech Zaremba. “OpenAI Gym”. (2016). *arXiv: 1606.01540*

[13] Towers, Mark and Terry, Jordan K. and Kwiatkowski, Ariel and Balis, John U. and Cola, Gianluca de and Deleu, Tristan and Goulão, Manuel and Kallinteris, Andreas and KG, Arjun and Krimmel, Markus and Perez-Vicente, Rodrigo and Pierré, Andrea and Schulhoff, Sander and Tai, Jun Jet and Shen, Andrew Tan Jin and Younis, Omar G. “Gymnasium”. (2023). *DOI: 10.5281/zenodo.8127026*

[14] Terry, J and Black, Benjamin and Grammel, Nathaniel and Jayakumar, Mario and Hari, Ananth and Sullivan, Ryan and Santos, Luis S and Dieffendahl, Clemens and Horsch, Caroline and Perez-Vicente, Rodrigo and others. “Pettingzoo: Gym for multi-agent reinforcement learning”. (2021). *Advances in Neural Information Processing Systems, vol. 34*

[15] Shengyi Huang, Santiago Ontañón. “A Closer Look at Invalid Action Masking in Policy Gradient Algorithms”. (2022). *arXiv:2006.14171*

[16] Antonin Raffin and Ashley Hill and Adam Gleave and Anssi Kanervisto and Maximilian Ernestus and Noah Dormann. “Stable-Baselines3: Reliable Reinforcement Learning Implementations”. (2021). *http://jmlr.org/papers/v22/20-1364.html*

[17] Terry, J. K and Black, Benjamin and Hari, Ananth. “SuperSuit: Simple Microwrappers for Reinforcement Learning Environments”. (2020). *arXiv:2008.08932*

[18] Bartłomiej Tarcholik. Repozytorium zawierające implementację środowiska wieloagentowego “Ecosystem”, algorytmów uczenia oraz wyników szkolenia agentów na środowisku. (2024). *https://github.com/Vedemin/Ecosystem*