Uniwersytet Mikołaja Kopernika
Wydział Matematyki i Informatyki

Tobiasz Pokorniecki
nr albumu: 291941

Praca magisterska
na kierunku informatyk

Teoria i zastosowanie algorytmu Optymalizacji Polityki Proksymalnej w środowisku gry HaxBall

Opiekun pracy dyplomowej:
dr Michał Chlebiej
Wydział Matematyki i Informatyki
Uniwersytet Mikołaja Kopernika

Toruń, 2022

Przymuję pracę i akceptuję

.....

data i podpis opiekuna pracy

Potwierdzam złożenie pracy dyplomowej

.....

data i podpis pracownika dziekanatu

Spis treści

1 Wprowadzenie

1.1 Wstęp

Ogólnopojęta tematyka sztucznej inteligencji do tej pory nie przestała nas zaskakiwać oraz napawać dumą. Rozwiązania oraz wyniki które owy świat prezentuje mogą przerosnąć najśmielsze oczekiwania nawet samego twórcy. W samym roku 2022 byliśmy świadkami wielkich osiągnięć w dziedzinie sztucznej inteligencji przy podejmowaniu szybkich decyzji przez pojazdy autonomiczne [1], możliwości żywej współpracy robota z człowiekiem przy produkcji alternatorów [2], a nawet doświadczyliśmy przezwyciężenia najlepszych graczy e-sportowych przez sztuczną inteligencję w grze wyścigowej Gran Turismo [3]. Jednak najgłośniejszym przełomem roku 2022 mogą poszczycić się inżynierowie OpenAI. Stworzyli oni model (zwany DALL·E 2) generujący bardzo szczegółowe obrazy na podstawie opisu [4]. Wymienione powyżej osiągnięcia w większości opierają się na systemie sieci neuronowych, który jest bardzo ciekawym, a zarazem przyszłościowym tematem wartym zgłębienia. Praca ta zakłada znajomość podstawowych terminów związanych z tym obszarem. Odsyłam więc do poprzedniej, autorskiej pracy zawierającej dział poświęcony właśnie sieciom neuronowym, a więc teorii i ich zastosowaniom [5].

Owa praca na samym początku przypomni oraz dopełni rozpoczętą w poprzedniej pracy problematykę uczenia przez wzmacnianie. Zaprezentowana klasyfikacja algorytmów jest równocześnie pretekstem do uargumentowania wyboru algorytmu, który następnie będzie bardzo szczegółowo opisany oraz podparty szeregiem przykładów pomocnych w jego zrozumieniu. Dalej poruszony zostanie aspekt ekstrakcji oraz symulacji środowiska HaxBall. Po przeprowadzeniu wstępu teoretycznego do wszystkich wykorzystanych mechanizmów oraz algorytmu, szczegółowo przedstawiony zostaje cały proces rozwoju części praktycznej oraz ostateczne wyniki i wnioski. W dużym skrócie, część praktyczna przeprowadzona jest w sposób naprzemienny, najpierw uczony agent jest atakujący, a następnie agent broniący, który w zamyśle musi wyćwiczyć w sobie takie zachowania, by pokonywać tego pierwszego. Mówiąc wybiórczo o tym przy jakich terminach uczenia maszynowego obracać będzie się ta praca, można wspomnieć o schematach zachowań zwanych politykami. Na przestrzeni całej pracy wymienione i wytłumaczone zostaną niektóre typy polityk, a także na przykładzie zaproponowanego środowiska selektywnie przebadane zostanie istnienie specjalnego typu polityk zwanych kontradyktoryjnymi. Dalej znaleźć można szczegółowe badanie oddzielnych zachowań, które agent nabywa na przestrzeni procesu uczenia. Wspomniany jest również proces kształtowania nagrody (reward shaping) oraz jego wkład w rozwiązywania problemu dopasowania, a co za tym idzie uwidaczniania polityk kontradyktoryjnych.

Przy poszukiwaniu odpowiedniego środowiska głównym wymogiem było podwyższenie szczebla abstrakcji na kolejny poziom w porównaniu do pracy poprzedniej, a zatem przejście do środowiska o ciągłej przestrzeni stanów. Drugim z kolei było zapewnienie przez środowisko względnej przystępności w pobieraniu stanów metodą ekstrakcji obrazu. Mając na uwadze te dwa aspekty ostateczny wybór padł na relatywnie proste środowisko gry HaxBall. Wykorzystanie sztucznej inteligencji w tym środowisku jest zajęciem już kultywowanym, lecz w pracy skupiono się dużo bardziej na badaniu jej pomysłowości oraz umiejętności radzenia sobie z problemami przy pomocy zaproponowanej autorskiej formy nagradzania w postaci gry o sumie zerowej.

1.2 Cel i zakres pracy

Głównym celem pracy jest przedstawienie działania algorytmu PPO produkującego stochastyczne modele decyzyjne operujące w ciągłych środowiskach. Na początku pracy wytłumaczona jest kategoryzacja algorytmów uczenia przez wzmacnianie, a wyróżniony zostaje jeden, wspomniany wyżej, algorytm zwany algorytmem Optymalizacji Polityki Proksymalnej (PPO). Na podstawie tego algorytmu wyprodukowany został szeroki wachlarz modeli różniących się hiperparametryzacją, a co za tym idzie - wypracowanymi wynikami. Każdy z przytoczonych modeli został porównany i oceniony w oparciu o statystyki wyprodukowane przez nie przez cały proces ich uczenia. Statystyki te uwidocznione zostały na wygenerowanych wykresach. Szczególną uwagę w pracy skupiono na jak najlepszym wyuczeniu dwóch grających przeciw sobie modeli oraz analizie ich zachowań w środowisku. Omawiany w pracy projekt stworzony został z zamysłem wykorzystania go jako narzędzia pozwalającego na proste i modularne eksperymentowanie ze środowiskiem gry HaxBall. Skupia on podstawowe narzędzia wymagane do ekstrakcji obrazu i wprowadzania sygnałów do środowisk zewnętrznych, ale przez hierarchiczność klas i interfejsów pozostaje w tej materii elastyczny, pozwalając na implementowanie własnych technik interpretacji. Dodatkowo praca powierzchownie podejmuje poruszany obecnie w sferze inżynierii uczenia maszynowego temat problemu dopasowania i wynikających z niego trudności z zapanowaniem nad własnym tworem.

2 Uczenie przez wzmacnianie

2.1 Wstęp

Rozdział ten chciałbym rozpocząć od ponownego odniesienia się do mojej poprzedniej pracy, w której istnieje bliźniaczo zatytułowany rozdział. Nakreślam tam typ problemu zwany decyzyjnym, jak do niego podchodzić, formułować i ostatecznie - jak rozwiązywać. Ten rozdział zakłada podstawowe rozeznanie w dziedzinie uczenia maszynowego, w szczególności gałęzi algorytmów uczenia przez wzmacnianie rozwiązujących wyżej wspomniane problemy. Poruszy on temat rozróżnienia tych algorytmów oraz uargumentuje decyzję zastosowania konkretnego algorytmu w problemie przedstawionym w dalszej części pracy. Jednak w celu sprawniejszego wdrożenia się w temat, poniżej przedstawiam podstawowy słowniczek według którego budowana będzie dalsza merytoryka:

- Środowisko - przestrzeń stanów reprezentowana w danym momencie przez dokładnie jeden z nich. Udostępnia zbiór akcji manipulujących tym stanem i w zależności od podjętej akcji na danym stanie karze lub nagradza agenta.
- Agent - jednostka decyzyjna wprowadzona do przestrzeni stanów środowiska. Decyduje jaką akcję wykonać mając do dyspozycji jego obecny stan. Jego celem jest zmaksymalizowanie sumarycznej wartości nagród zwracanej przez środowisko.
- Akcja - czynność lub jej natężenie możliwe do wykonania na środowisku. Agent podejmuje akcje w oparciu o postawiony przed nim stan środowiska.
- Stan - grupa wartości z przestrzeni stanów środowiska.
- Nagroda - werdykt środowiska będący liczbą rzeczywistą. Warunkuje czy akcja wykonana na obecnym stanie jest pożądana w celu pozytywnego zakończenia. Negatywna wartość nagrody często nazywana jest karą.
- Moment decyzyjny - grupa wartości postaci: stan, akcja, nagroda, następny stan. Moment podjęcia decyzji o akcji, która następnie jest odpowiednio nagradzana (bądź karana) i prowadzi do następnego stanu środowiska.
- Epizod - skończona seria następujących po sobie momentów decyzyjnych. Najczęściej kończy się konkretnym rezultatem.
- Trajektoria - niepusty i skończony zbiór momentów decyzyjnych, podzbiór momentów decyzyjnych z epizodu.
- Polityka - termin określający podejście agenta do środowiska. Istnieją różne typy polityk, które możemy zaobserwować u agenta operującego w działającym środowisku. Typy te przedstawiać będę stopniowo na przestrzeni całej pracy.

Definicje powyżej wyjaśnionych terminów przeplatają się w naturalny sposób i każda jedna nadaje sens pozostałym. Wartym przypomnienia jest również sedno algorytmów przez wzmacnianie zawarte w definicji agenta, czyli maksymalizacja sumarycznej wartości nagrody. Za tym stwierdzeniem stoi bardzo jasny przekaz - celem agenta w każdym momencie decyzyjnym jest wykonanie takiej akcji, która zbliży go do pozytywnego ukończenia epizodu (w domyśle - akcji najbardziej nagradzanej).

2.2 Kryteria i podział algorytmów

To krótkie odświeżenie informacji pozwoli przejść nam do przedstawienia kilku cech i kryteriów według których gałąź algorytmów uczenia przez wzmacnianie można rozdzielić na pewne podgrupy.

Pierwsze dwa kryteria podziału są ściśle związane ze środowiskiem, w którym został sformułowany problem. Odnoszą się do przestrzeni stanów oraz akcji możliwych do wykonania, mogą one zawierać wartości dyskretne bądź ciągłe. Kryteria te możemy określić jako mało rygorystyczne, gdyż niekiedy w środowiskach istnieje możliwość odzwierciedlenia danego dyskretnego stanu bądź akcji na dziedzinę ciągłą i na odwrót. Dla przykładu, biorąc pod uwagę ciągłą dziedzinę akcji którą jest wartość odchylenia prostopadłego ułożenia kół względem osi w pojeździe (liczba rzeczywista o zakresie wartości od -1 do 1), możemy z łatwością zdyskretyzować akcję tego typu zachowując tylko wartości skrajne (całkowity skręt kół w lewo lub w prawo) oraz wartość neutralną (brak skrętu) tym samym ograniczając się do trzech wartości (akcji), które umożliwiają teraz zastosowanie algorytmów zupełnie innego typu.

Przed przedstawieniem kolejnego kryterium w tym rozdziale warto wytłumaczyć znaczenie pewnych konkretnych typów polityk - polityk docelowych oraz polityk zachowań. Pierwsze z nich są politykami, które ewoluują z każdą sesją uczenia zwracając raz to lepsze, raz to gorsze wyniki. Głównym ich celem jest dążenie do przejawiania jak najbardziej optymalnych zachowań. Jest to polityka, która na początku procesu uczenia wykazuje raczej losowe podejścia do zaistniałych sytuacji, a finalnie, jeśli przejawia takie zadatki, może być traktowana jako polityka bliska optymalnej. Celem polityk zachowań jest podejmowanie decyzji o akcji w konkrentym momencie decyzyjnym. Nie są one czymś czego można spodziewać się od wyuczonego agenta, ze względu na to, że w większości zachowania te są mało optymalne lub nawet są przeciwieństwem optymalności. Można o nich mówić jak o politykach pomocniczych zapoznających agenta z pewnym zalążkiem informacji na temat działania środowiska lub sugerujące podstawowe zachowania. Sprawiają, że sam początek uczenia przebiega płynniej i przy poprawnym doborze i ich zastosowaniu najoptymalniejsze zachowania zostaną uwidocznione dużo wcześniej. Takimi politykami można nazwać między innymi mechanizm losowego podejmowania akcji (podstawa eksploracji, czyli algorytmu ε-zachłannego) lub uczenia przez demonstrację. Wracając do omawianego kryterium, mówić będziemy o algorytmach z politykami włącznymi (On-Policy) i wyłącznymi (Off-Policy). Najprościej rzecz ujmując, algorytmy z politykami włącznymi to algorytmy w których polityka zachowań i polityka docelowa jest jednym, tym samym bytem. Momenty decyzyjne według których polityka docelowa będzie doskonalona są wygenerowane na podstawie wyników

obecnie zwracanych przez nią samą. W przypadku tych algorytmów część eksploracyjna wynika z samego procesu uczenia i zanika proporcjonalnie wraz z uzyskiwaniem przez model zbieżności. [6]

Ostatnie kryterium jest kategoryzacją algorytmów względem konieczności zastosowania mechanizmu prognozującego. Algorytmy dzielimy wtedy na wzorcowe (Model-based) oraz bezwzorcowe (Model-free). Zdarza się, że implementacje algorytmów w pewnym stopniu korzystają z obu podejść jednocześnie w celu wykluczenia nawzajem ich wad i skumulowania zalet. Studiując algorytmy uczenia maszynowego szybko można zauważyć, że termin modelu jest naprzemiennie używany z terminem modelu sieci neuronowej. Tutaj jednak obie te kategorie wcale nie odnoszą się, jak można by się domyśleć, do wykorzystywania modelu sieci neuronowej w algorytmie. Polskie tłumaczenie tego rozgraniczenia - algorytmy wzorcowe i bezwzorcowe - lepiej podkreślają co jest jego istotą. Wzorcem w tym przypadku określimy zadeklarowany bądź wyuczony mechanizm prognozujący przyszłe stany w środowisku pozwalający agentowi planować dalsze kroki. Warto podkreślić fakt, że istnienie wzorca nie koniecznie jest związane czysto z algorytmem, a również z samą jego implementacją. Istnieją przypadki algorytmów, które z założenia są bezwzorcowe, ale dodanie do nich mechanizmu wzorca znacząco poprawia ich wydajność [7, 8]. Wykorzystanie takiego wzorca podczas uczenia pozwoli mechanizmowi decyzyjnemu lepiej zrozumieć działanie samego środowiska, aniżeli jak w przypadku algorytmów bezwzorcowych, poznać tylko przejścia stanów środowiska na podstawie wykonywanych na nim akcji. Czas w jakim algorytmy wzorcowe osiągają zadowalający poziom jest niski, ale wykorzystuje się je rzadziej. Powodem jest słabsza generalizacja i konieczność samodzielnego zdefiniowania wzorca, który koniec końców może okazać się dla algorytmu ograniczającym wydajnościowo lub może nawet być niemożliwy do wydedukowania. Istnieją jednak algorytmy, w których koncepcie model nie jest jedynie mechanizmem podbudowującym wynik, a priorytetowym aspektem bez którego nie mogą działać. Jedną z popularniejszych implementacji algorytmu wzorcowego jest silnik AlphaZero potrafiący nauczyć się grać w takie gry jak szachy, shogi oraz Go i już po 24 godzinach uczenia osiągnąć nadludzki poziom gry. Wzorzec w tej implementacji jest jasno postawiony, jest to algorytm przeszukiwania drzewa Monte-Carlo (MCTS). W przypadku algorytmów bez wzorca uczenie głównie odbywa się poprzez zbieranie informacji o środowisku metodą prób i błędów. [9, 10, 11].

Podsumowaniem tego podrozdziału będzie wypisanie przykładów algorytmów uczenia przez wzmacnianie, sklasyfikowanie pod każdym z wymienionych powyżej kryteriów i krótki opis ich działania.

Algorytm	**Przestrzeń stanu**	**Przestrzeń akcji**	**Konieczność stosowania wzorca**	**Włączność polityki**
Q-learning	dyskretna lub zdyskretyzowana	dyskretna	nie	wyłączna (algorytm ε-zachłanny)
DQN	dyskretna lub zdyskretyzowana	dyskretna	nie	wyłączna (algorytm ε-zachłanny)
MBMF	ciągła lub dyskretna	ciągła lub dyskretna	tak	wyłączna (uczenie na wstępnie losowej parametryzacji)
DDPG	ciągła lub dyskretna	ciągła	nie	wyłączna
PPO	ciągła lub dyskretna	ciągła lub dyskretna	nie	włączna

Rysunek 1: Kategoryzacja algorytmów według opisanych kryteriów podziału.

Chcąc sprawnie opisać każdy z algorytmów, powyższą tabelę (rysunek 1) rozdzielę na trzy grupy. Głównymi założeniami pierwszych dwóch algorytmów (Q-learning, DQN) jest sprzężenie wszystkich możliwych stanów środowiska i możliwych do wykonania akcji w pary o pewnych Q-wartościach. Wielkość tej wartości i jej proporcjonalność względem innych mówi algorytmowi o optymalności akcji w sparowanym stanie. Następnie przedstawiony jest algorytm MBMF, który z założenia stosuje techniki algorytmów wzorcowych i bezwzorcowych. Ostatnia para algorytmów (DDPG i PPO) jest z rodziny optymalizacji polityki (Policy Optimization). Implementują one rozgraniczenie pomiędzy modelem przybliżającym Q-funkcję lub funkcję przewagi, a modelem polityki. Kolejność wyliczenia algorytmów w rodzinach nie jest przypadkowa, każdy następnik jest w pewnym sensie rozwinięciem poprzedniego (nie koniecznie pod względem wydajności, gdyż wszystko zależy od środowiska w którym stosujemy algorytm) lub na nim bazuje. [12, 13, 14]

Solucja algorytmu Q-learning jest tą najbardziej podstawową ze wszystkich tu wymienionych. Wyróżnia się swoją szybkością, lecz obciąża pamięciowo sprzęt, ze względu na to że do przetrzymywania par stanów i akcji wykorzystuje się najzwyklejszą wielowymiarową tablicę (zwaną Q-tablicą). Stąd, by implementacja w ogóle była możliwa,

przestrzeń stanów i akcji musi być w postaci dyskretnej. Początkowo tablica wypełniana jest losowymi wartościami, dlatego kluczowym dodatkiem jest zastosowanie algorytmu ε-zachłannego (eksploracji). Polega on na wyborze losowych akcji z pewnym zmniejszającym się w trakcie uczenia prawdopodobieństwem. Pozwala algorytmowi Q-learning poznać jak największą liczbę kombinacji stanów i akcji sprawiając, że algorytm ma szansę nauczyć się wybierać te najoptymalniejsze. Algorytm DQN (Deep Q-Network) jest rozwinięciem algorytmu Q-learning o sieci neuronowe. Do wyliczania Q-wartości i wyboru akcji wykorzystuje się sieć neuronową, która na wejściu otrzymuje stan środowiska, a oczekiwanym wyjściem jest wektor Q-wartości odpowiadających każdej z odpowiednio zaindeksowanej akcji. Rozwiązanie to jest bardziej czasochłonne, ale znosi limit pamięciowy nałożony przez algorytm Q-learning. Zapisane na dysku wagi w porównaniu do zapisanej Q-tablicy zajmują 100 razy mniej miejsca. Po dokładniejszy opis odsyłam do mojej pracy przyglądającej się oraz porównywającej oba te podejścia w przykładowym środowisku gry typu Pac-Man. [5]

Algorytm MBMF (Model-Based Model-Free), jak zaznaczają twórcy, jest pomostem pomiędzy wzorcowym i bezwzorcowym podejściem do problemów opartych na uczeniu przez wzmacnianie. Generalizując, schemat jego działania opiera się na wstępnym zapoznaniu modelu dynamicznego (termin z programowania dynamicznego) ze środowiskiem. Zapoznanie opiera się na uczeniu przejść stanów generowanych prosto ze środowiska z wykorzystaniem wielu polityk losowych, a dokładniej parametryzacji ich modelów. Mając już model dynamiczny, jesteśmy za jego pomocą w stanie przewidywać iteracyjnie trajektorie stanów w oparciu o zadaną parametryzację polityki. Następnie deklarowana jest bazowa funkcja oceniająca parametryzację, zwracająca wyniki oparte na przeszukiwaniu modelu dynamicznego metodą Monte-Carlo. Funkcja ta, jak i model dynamiczny, są cały czas udoskonalane w procesie uczenia w którym wykorzystuje się proces gaussowski (metoda regresji probabilistycznej) oraz optymalizację bayesowską (metoda wyszukiwania minimum nieznanej funkcji). Wykorzystanie procesu gaussowskiego pozwala algorytmowi na określenie nie tylko oceny parametryzacji polityki ale i wiarygodności tej oceny. Wzorcem w tym podejściu określimy model dynamiczny zwracający trajektorie stanów na podstawie parametryzacji oraz funkcję oceniającą tę parametryzację, natomiast częścią bezwzorcową jest ciągłe zbieranie informacji o środowisku i optymalizacja obu tych mechanizmów metodą prób i błędów. [12]

DDPG (Deep Deterministic Policy Gradient) jest algorytmem o którym można myśleć jak o algorytmie DQN obsługującym ciągłą (i tylko ciągłą) przestrzeń akcji i jest on pierwszym tutaj opisywanym algorytmem z rodziny aktor-krytyk. W algorytmie DQN, ze względu na dyskretną przestrzeń akcji, podjęcie decyzyji było bardzo proste, wybierana była akcja z najwyższą Q-wartością określającą jej opłacalność. W przypadku ciągłej przestrzeni ilość akcji możliwych do wykonania jest nieskończona i określenie Q-wartości dla każdej z nich jest niemożliwe. Algorytm DDPG, oprócz modelu przybliżającego Q-wartości znanego z algorytmu DQN wykorzystuje osobny model uczący się optymalnej polityki "siły" akcji w przestrzeni stanów. Model odpowiedzialny za politykę nazywany jest aktorem, a model przybliżający Q-wartość - krytykiem. Ze względu na to, że schemat uczenia opiera się na ciągłym zbieraniu informacji ze środowiska, kilkukrotnym wykorzystywaniu pojedynczego doświadczenia w procesie uczenia oraz

losowym jego doborze to obok powyżej określonych modeli, które w stu procentach biorą udział w procesie uczenia, wykorzystuje się bliźniacze modele docelowe. Modele te są uaktualniane co kilka sesji na podstawie modeli uczonych celem zminimalizowania nieprecyzyjności mogącej wystąpić w tym procesie. To właśnie model polityki docelowej odzwierciedla prawidłową politykę zobowiązaną do podejmowania akcji w działającym środowisku. Część eksploracyjna procesu uczenia zadeklarowana jest nieco inaczej niż w przypadku algorytmów z Q-rodziny. Pomimo, że wykorzystanie czystego algorytmu ε-zachłannego jest tu jak najbardziej możliwe (można by z pewnym prawdopodobieństwem generować losowe wartości z pewnego przedziału wartości "siły" akcji i wprowadzać je do środowiska), to zostało tutaj zastosowane inne podejście. Mianowicie wykorzystywana jest losowa zmienna ε z rozkładu normalnego, która dodawana jest do zwracanych przez aktora wartości co sprawia, że krzywa uczenia obu modeli jest bardziej gładka i pozwala aktorowi wykazywać zapoznaną wiedzę dużo wcześniej, tym samym utrzymując w danych dostateczną moc szumu eksploracyjnego. [14]

2.3 Algorytm Optymalizacji Polityki Proksymalnej (PPO)

Z racji, że algorytm PPO sprawdził się najlepiej w środowisku którym dysponuję na przestrzeni tej pracy, poświęcę mu osobny podrozdział. Algorytm PPO jest domyślnym algorytmem wykorzystywanym w uczeniu przez wzmacnianie w laboratorium badawczym sztucznej inteligencji OpenAI. Zachętą do użycia go są wyniki jakie jest w stanie osiągać, tym samym będąc prostszym w obsłudzie i implementacji od pokrewnych podejść które znane są do tej pory [15, 16]. Jednym z większych osiągnięć w którym bazowano na tym algorytmie jest sztuczna inteligencja OpenAI Five sterująca pięcioma graczami w okrojonym środowisku gry DotA 2. Prace nad nią rozpoczęły się w 2016 roku, a stopniowo zwiększając poziom skomplikowania w roku 2019 była ona w stanie pokonać topową jak na tamten okres drużynę [17]. Algorytm PPO jest algorytymem bezwzorcowym z polityką włączną mogącym operować zarówno w przestrzeniach dyskretnych jak i ciągłych z dodatkową możliwością zrównoleglenia całego procesu uczenia (PPO2). Podobnie jak DDPG wywodzi się z rodziny algorytmów aktor-krytyk. W większości akcje wykonywane przez aktora podejmowane są na podstawie generowanej przez algorytm w trakcie uczenia polityki stochastycznej. Oznacza to, że przez większość czasu wykonywane akcje będą poddawane pewnej dozie niepewności opartej na rozkładzie prawdopodobieństwa, a nie czystej przewagi wartości konkretnej akcji nad innymi, tak jak w algorytmach z rodziny Q-learning, które oferują znajdowanie tylko polityk deterministycznych. W idealnej sytuacji, polityka stochastyczna będzie powoli wypierana przez politykę deterministyczną przy pomocy procesu optymalizacji polityki, który cały czas zaopatrywany jest w nową, dostateczną ilość różnorodnych danych próbkowanych ze środowiska za pomocą wyżej wspomnianej niepewności w wykonywanych akcjach. Polityki stochastyczne dzielą się na polityki kategorialne oraz diagonalne polityki gaussowskie, gdzie podział zależy od postaci przestrzeni akcji (dyskretna lub ciągła) [18]. Algorytm PPO jest algorytmem z polityką włączną, gdyż decyzje wydawane przez polityki stochastyczne zapewniają odpowiednią długość eksploracji, polityki "trzecie" nie są w tym przypadku wymogiem do osiągania satysfakcjonujących wyników, zbieżność do co najmniej lokalnej optymalności jest gwarantowana. Dodatkowym

aspektem algorytmu PPO jest zwiększona efektywność pojedycznej próbki ze względu na to, że każda sesja uczenia składa się z kilku identycznie zadeklarowanych kroków zwanych epokami przeprowadzonych na jednej próbce danych (ich ilość jest dodatkowym hiperparametrem).

Skupię się teraz na wytłumaczeniu pojęcia zastosowania gradientu polityki oraz procesu optymalizacji polityki proksymalnej od strony matematycznej. Zaczynając od korzeni, celem agenta w algorytmach uczenia przez wzmacnianie jest maksymalizacja przewidywanej wartości nagrody r wydawanej przez zadeklarowany system nagród środowiska R na podstawie akcji a oraz stanu s przez pewną trajektorię momentów decyzyjnych τ, gdzie moment terminalny oznaczany jest przez T. Można to odzwierciedlić wzorem

$$R(\tau) = \sum_{t}^{T} r_t, \tag{1}$$

$$r_t = R(s_t, a_t, s_{t+1}). \tag{2}$$

Jednostką którą kieruje się agent podejmując decyzje jest polityka π, której odpowiednie sparametryzowanie θ jest kluczem do rozwiązywania wyżej postawionego problemu. Politykę opartą na sparametryzowaniu oznacza się przez π_θ. Z racji, że działania agenta kierującego się polityką są bezpośrednio skonsolidowane z jej parametryzacją, patrząc na problem bardziej ogólnie, parametryzacja polityki jest strukturą szukaną przez algorytm podczas uczenia. Inaczej możemy więc mówić, że szukamy takiej parametryzacji (polityki) θ, która maksymalizuje przewidywaną wartość nagrody $J(\theta)$ i minimalizuje błąd na przestrzeni całego środowiska, czyli

$$J(\theta) = \underset{\tau\sim\pi_\theta}{E}[R(\tau)]. \tag{3}$$

Znalezienie takiego θ, które maksymalizuje wartość J jest równoznaczne z rozwiązaniem problemu uczenia przez wzmacnianie w danym środowisku. Zawężając rozważania tylko do algorytmów opartych na sieciach neuronowych, parametryzacja θ jest równoznaczna z wagami sieci neuronowej. Przyjrzyjmy się procesowi szukania takiej parametryzacji opisanego za pomocą wzoru

$$\theta_{t+1} = \theta_t + \alpha\nabla_\theta J(\theta). \tag{4}$$

Składnik $\nabla_\theta J(\theta)$ w równaniu (4) jest wspominanym na samym początku tego podrozdziału gradientem polityki, a całe równanie nazywane jest optymalizacją polityki metodą wznoszenia gradientu. Przed próbą wnikliwej analizy powyższego terminu należy zastanowić się czym jest prawdopodobieństwo wystąpienia danej trajektorii momentów decyzyjnych przy zadanej parametryzacji, gdyż jest ono kluczowe w wyliczaniu przewidywanej wartości nagrody z równania (3) przy założeniu, że operujemy na polityce stochastycznej.

Jak widać na równaniu

$$P(\tau|\theta) = \rho_0(s_0) \prod_t^T P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t), \tag{5}$$

prawdopodobieństwo uzyskania konkretnej trajektorii przy zadanej parametryzacji zależy między innymi od prawdopodobieństwa rozpoczęcia trajekorii z pewnego wyróżnionego stanu ($\rho_0(s_0)$) oraz prawdopobieństw ułożenia się przejść stanów, uzyskiwanych na podstawie polityki, zgodnie z zadaną trajektorią [18]. Rozwijając równanie (3), mając możliwość wyliczenia prawdopodobieństwa zadanej trajektorii oraz gdy znany jest system według którego trajektoria jest oceniana, jesteśmy w stanie wyliczyć przewidywaną wartość nagrody dla parametryzacji w sposób

$$J(\theta) = \underset{\tau\sim\pi_\theta}{E}[R(\tau)] = \int_\tau P(\tau|\theta)R(\tau). \tag{6}$$

Tutaj leży spory problem. Nie jesteśmy w stanie dokładnie określić przewidywanej wartości nagrody dla każdej możliwej trajektorii. Nawet najmniejsza zmiana może diametralnie wpłynąć na ostateczny wynik. Dodatkowo, w środowisku mogą występować stany, które wpływają na przewidywania, a są niezależne od polityki. Istnieje jednak pewne przekształcenie równania (6) upraszczające to zadanie. Wykorzystywane jest ono w takiej formie między innymi w najbardziej podstawowym wariancie algorytmu gradientu polityki - algorytmie VPG (Vanilla Policy Gradient) [19]. Przedstawione poniżej przekształcenie (7) nie tylko unieważnia mus wyliczenia prawdopodobieństw wystąpienia wszystkich stanów, ale również bardzo upraszcza wyliczanie gradientu, dlatego, że

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \int_\tau P(\tau|\theta)R(\tau) \\
&= \int_\tau \nabla_\theta P(\tau|\theta)R(\tau) \\
&= \int_\tau P(\tau|\theta)R(\tau)\nabla_\theta \log P(\tau|\theta)R(\tau) \\
&= \underset{\tau\sim\pi_\theta}{E}[\nabla_\theta \log P(\tau|\theta)R(\tau)] \\
&= \underset{\tau\sim\pi_\theta}{E}\Big[\sum_t^T \nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)\Big],
\end{aligned} \tag{7}$$

więc

$$\nabla_\theta J(\theta) = \underset{\tau\sim\pi_\theta}{E}[R(\tau)] = \underset{\tau\sim\pi_\theta}{E}\Big[\sum_t^T \nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)\Big]. \tag{8}$$

Podsumowując, z równań (3 - 8) wynika, że jeśli chcemy maksymalizować przewidywaną wartość nagrody $J(\theta)$ musimy zastosować gradient w obrębie całego środowiska. Jest to bardzo ciężkie zadanie. Natomiast według równania (7) zadanie to jest równoznaczne z zastosowaniem gradientu logarytmu prawdopodobieństwa wybrania konkretnej akcji przez politykę wzdłuż trajektorii z uwzględnieniem jej wyniku końcowego. Wartość przewidywana sumy takich gradientów może być wtedy niczym innym jak ich uśrednionym gradientem, który możemy zastosować w równaniu (4). Jednakże podejściem zapewniającym lepszą generalizację jest zastosowanie średniego gradientu z większej ilości trajektorii (bądź pojedycznej trajektorii rozbitej na mniejsze).

Wyliczenie pojedynczego gradientu $\hat{g}$ dla kolejnego kroku optymalizacji wygląda więc następująco:

$$\hat{g} = \frac{1}{N} \sum_{\tau \in D} \sum_{t}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) = \hat{E}, \tag{9}$$

gdzie D jest serią trajektorii $\{\tau_i\}_{i=0,\ldots,N}$. [18, 20, 21, 22]

W bardziej zaawansowanych metodach gradientu polityki, na wcześniej zadeklarowanym systemie nagród R z równania (1), bazują pewne funkcje zwracające bogatsze w informacje wartości. Są one wykorzystywane jako zastępstwo za wartości z podstawowego systemu nagród. Na potrzeby tej pracy przedstawię funkcję nazywaną funkcją przewagi opisaną wzorem

$$A_\theta(s_t, a_t) = Q_\theta(s_t, a_t) - V_\theta(s_t). \tag{10}$$

Funkcja przewagi jest różnicą dwóch innych funkcji. Funkcja Q jest funkcją Q-wartości określającą wartość nagrody którą otrzyma agent przebiegajac po trajektorii o długości N według ustalonej polityki z parametryzacją π_θ i kończąc w pewnym stanie terminalnym T przyjmując, że rozpocznie on interakcję ze środowiskiem w stanie s i wykona akcję a określona wzorem

$$Q_\theta(s_t, a_t) = \sum_{n=0}^{N} \gamma^n r_n. \tag{11}$$

Funkcja V nazywana jest funkcją wartości. Jest ona zdefiniowana identycznie tak jak funkcja Q, w postaci

$$V_\theta(s_t) = \sum_{n=0}^{N} \gamma^n r_n. \tag{12}$$

Kluczowa różnica między tymi funkcjami leży w sposobie ich interpretacji. Funkcja V określa wartość nagrody dla danego stanu s jeszcze przed wykonaniem jakiejkolwiek akcji, a funkcja Q - po wykonaniu akcji a. Oczywistym może się wydać, że przy większym skomplikowaniu środowiska nie jesteśmy jednoznacznie wyznaczyć każdej wartości obu tych funkcji, dlatego posługiwać będziemy się tylko ich wartościami przewidywanymi. Ostateczne, dalej wykorzystywane oznaczenie przewidywanej wartości funkcji przewagi wyglądać będzie następująco:

$$\hat{A}_\theta(s_t, a_t) = E[Q_\theta(s_t, a_t) - V_\theta(s_t)].[23] \tag{13}$$

Znając podstawy zagadnienia gradientu polityki, możemy zabrać się za zrozumienie głównej idei stojącej za algorytmem PPO, czyli algorytmem optymalizacji polityki proksymalnej. PPO swoje korzenie znajduje w algorytmie TRPO (Trust Region Policy Optimization), który ze względu na swój poziom skomplikowania jest wykorzystywany rzadziej. Algorytm PPO (i TRPO) opiera się na koncepcji podejmowania jak największego kroku poprawiającego politykę na podstawie zebranych danych. Krok ten nie może być zbyt duży, by nie doprowadzić do załamania wydajności. Algorytm PPO istnieje w dwóch wersjach: PPO-Penalty, który podobnie jak TRPO wykorzystuje mechanizm KL-ograniczonej aktualizacji polityki lecz w trochę zmienionej formie "płynnego" ograniczenia oraz wersja PPO-Clip, którą uznałem za najlepszą w postawionym

sobie problemie. Mechanizm algorytmu PPO-Clip opiera się na specjalnie zadeklarowanym wycinaniu nadmiernych zmian w polityce (przez co aktualizacje parametryzacji są o wiele stabilniejsze) oraz na trzymaniu się pesymistycznej granicy wydajności polityki, co sprawia że algorytym nie wyciąga zbyt pochopnych wniosków, które mogłyby doprowadzić do utraty wydajności. Spadki wydajności w algorytmach gradientu polityki są częstym problemem z którym należy walczyć. Występują one głównie z racji wykorzystywania w nich bardzo chaotycznej funkcji przewagi, która jest nośnikiem ważnych dla poprawnego działania algorytmu informacji oraz z powodu ryzyka ugrzęźnięcia w lokalnych minimach już nawet po kilku źle przeprowadzonych krokach optymalizacji. [15]

Istotną różnicą w algorytmach optymalizacji polityki w porównaniu do typowego zastosowania gradientu polityki jest główny cel optymalizacji. Zamiast szukania takiej polityki, która maksymalizuje przewidywaną wartość nagrody (równanie (3)), szukać będziemy polityki nie odbiegającej daleko od poprzedniej, ale dążącej do spełnienia pewnego celu zastępczego, gwarantującego względną poprawę polityki. Dla PPO-Clip cel zastępczy zdefiniowany jest następująco:

$$J^{CLIP}(\theta) = E[min(r(\theta)\hat{A}_{\theta_{old}}(s,a), clip(r(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_{\theta_{old}}(s,a))], \tag{14}$$

gdzie

$$r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} \tag{15}$$

jest stosunkiem prawdopodobieństwa wykonania akcji w danym stanie posługując się polityką z obecną parametryzacją θ, a polityką z parametryzacją z kroku poprzedniego θ_{old}. Rozbijmy równanie (14) na fragmenty i wytłumaczmy każdy jego aspekt. Funkcja *min* określać będzie mniejszy z dwóch możliwych kroków przemieszczenia się prawdopodobieństwa wykonywania danej akcji w stanie. Bierze ona pod uwagę wartość funkcji przewagi dla tych wartości (czy akcja ta była opłacalna) oraz stosunek prawdopodobieństwa wykonania tej akcji przy obecnej parametryzacji do prawdopodobieństwa wykonania jej przy parametryzacji z kroku poprzedniego. Operacja ta zapewnia nas, że parametryzacja cały czas zakrawa o swój pesymistyczny wariant. Drugi składnik (funkcja *clip*) przycina stosunek prawdopodobieństwa wykonania akcji do określonych przez hiperparametr ϵ ram. Hiperparametr ten jest liczbą o wartościach w granicach od 0.2 do 0.1 lub mniejszych. Utrzymywanie stosunku prawdopobieństw w tych ramach nie pozwala algorytmowi wykonywać zbyt dużych kroków. Celem lepszego zrozumienia współpracy funkcji *min* z funkcją *clip* przedstawione zostanie kilka uproszczonych przykładów wraz z wyliczeniem i wyciągnięciem wniosków. [24]

Przykład 1

Opis:

$r(\theta) = 1$ ($\theta = \theta_{old}$, jest to pierwsza epoka sesji uczenia),
$A = 25$ (akcja wykonana przez agenta okazała się opłacalna),
$\epsilon = 0.1$

Wyliczenie: $min(1 * 25, clip(1, 0.9, 1.1) * 25) = min(25, 25) = 25$

Wniosek: W przypadku, gdy $r(\theta) = 1$, funkcje *min* oraz *clip* nie zostają zastosowane. Cel redukuje się do celu zastępczego wykorzystywanego w TRPO. Prawdopodobieństwo wykonania akcji a na stanie s zwiększa się o pewien nieoprawiony krok.

Przykład 2

Opis:

$r(\theta) = 1.2$ (Wykonanie akcji a w obecnym stanie s jest bardziej prawdopodobne posługując się parametryzacją θ w porównaniu do parametryzacji z kroku poprzedniego θ_{old}),
$A = 25$,
$\epsilon = 0.1$

Wyliczenie:
$min(1.2 * 25, clip(1.2, 0.9, 1.1) * 25) = min(30, 1.1 * 25) = min(30, 27.5) = 27.5$

Wniosek: Obie funkcje znalazły swoje zastosowanie. Krok został pomniejszony o odstającą wartość stosunku prawdopodobieństw, lecz nadal jest on większy niż krok z pierwszego przykładu. Jest to przypadek jak najbardziej korzystny dla polityki, gdyż $\pi_\theta(a|s)$ zwiększa się o pewien ostrożny krok w stronę pozyskiwania wyższych wartości z funkcji przewagi.

Przykład 3

Opis:

$r(\theta) = 0.8$ ($\pi_\theta(a|s) < \pi_{\theta_{old}}(a|s)$),
$A = 25$,
$\epsilon = 0.1$

Wyliczenie:
$min(0.8 * 25, clip(0.8, 0.9, 1.1) * 25) = min(20, 0.9 * 25) = min(20, 22.5) = 20$

Wniosek: Funkcja min w tym przypadku zwróciła pierwszą, nieokrojoną wartość. Przyczyna nie jest oczywista na pierwszy rzut oka. Z racji, że $\pi_\theta(a|s) < \pi_{\theta_{old}}(a|s)$, a wiemy, że a jest akcją opłacalną ($A = 25$) to algorytm podjął w trakcie sesji uczenia niepoprawny krok i powinien się raczej z niego wycofać. Poprawiając swój błąd, algorytm zwiększy $\pi_\theta(a|s)$ o bardziej ostrożny krok, który może w przyszłości pomóc uniknąć ponowne wykonanie błędnego kroku.

Przykład 4

Opis:

$r(\theta) = 2$

$A = -25,$

$\epsilon = 0.1$

Wyliczenie:

$min(2 * (-25), clip(2, 0.9, 1.1) * (-25)) = min(-50, -27.5) = -50$

Wniosek: W trakcie sesji uczenia popełniono kategoryczny błąd, prawdopodobieństwo wykonania nieopłacalnej akcji a wzrosło dwukrotnie. Należy zmniejszyć jej prawdopodobieństwo o jak największy krok.

Przykład 5

Opis:

$r(\theta) = 0.5$

$A = -25,$

$\epsilon = 0.1$

Wyliczenie:

$min(0.5 * (-25), clip(0.5, 0.9, 1.1) * (-25)) = min(-12.5, -22.5) = -22.5$

Wniosek: Zmniejszenie prawdopodobieństwa nieopłacalnej akcji jest krokiem w dobrym kierunku, dobrze żeby był jak największy.

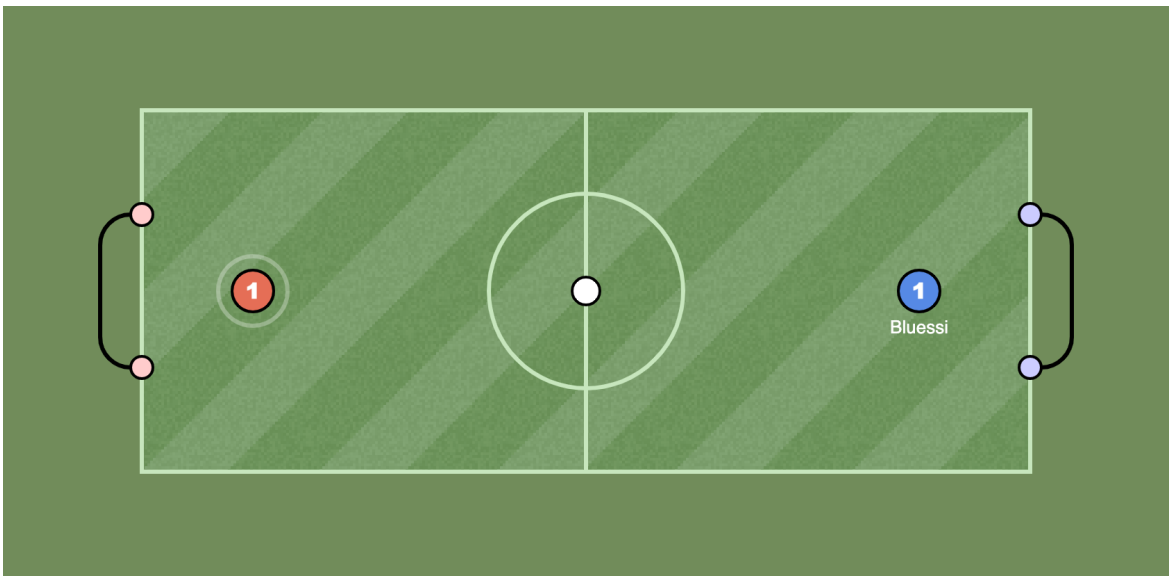
Podsumowując wnioski wysnute przy pomocy powyższych przykładów, algorytm starał się będzie poprawiać politykę w sposób bardzo ostrożny. To, która granica przedziału funkcji *clip* będzie brana pod uwagę zależne jest od wartości zwracanej przez funkcję przewagi. Pogoń za maksymalizowaniem funkcji przewagi jest konsekwentnie opóźniania, poddając poprawnie wykonane akcje pewnej dozie niepewności. Natomiast w przypadku wykrycia przez algorytm podjęcia błędnej decyzji wycofuje się on z dokonanych poprzednio zmian (zmniejszając prawdopodobieństwo akcji), a następnie podchodzi do problemu ostrożniej. Działanie celu zastępczego zastosowanego w algorytmie PPO-Clip jest bardzo przemyślanym, logicznym, a zarazem prostym rozwiązaniem.

Dodatkowym wartym wspomnienia mechanizmem możliwym do zaimplementowania w algorytmie PPO-Clip jest mechanizm early stopping, którego zamysłem jest wcześniejsze kończenie obecnie prowadzonej sesji uczenia po przekroczeniu przez algorytm pewnego ustalonego progu. Progiem tym jest średnia rozbieżność prawdopodobieństw (KL-divergence) pomiędzy polityką używaną podczas próbkowania środowiska (polityką zachowań), a polityką uczoną (polityką docelową). Mechanizm ten ma na celu zminimalizowanie szansy nadmiernego wykorzystania silnie oddziałowujących momentów decyzyjnych co może doprowadzić do zachwiania wydajnością [25]. Mechanizm ten dla testu został zaimplementowany w omawianym projekcie, ale nie znalazł zastosowania przez to, że średnia rozbieżność prawdopodobieństw przy wykorzystaniu hiperparametryzacji omawianej później w pracy była niemal 2 rzędy niższa od najniższego proponowanego przez twórców progu. Mechanizm po prostu w żadnym przypadku nie był wykorzystywany.

3 Środowisko i struktura projektu

3.1 Opis środowiska gry HaxBall

Przed przystąpieniem do przedstawienia projektu oraz implementacji algorytmu PPO, omówię zasady którymi kieruje się wybrane przeze mnie środowisko. HaxBall jest wieloosobową grą online dostępną z poziomu przeglądarki opartą na fizyce w której dwie drużyny grają przeciwko sobie na wybranej planszy w uproszczoną wersję gry w piłkę nożną [26]. Każdy zespół składa się z dowolnej liczby graczy i domyślnie celem każdego z nich jest umieszczenie piłki w bramce przeciwnika. Gra ta jest również dobrą bazą do tworzenia odmiennych od piłki nożnej gier opartych na fizyce i ma bardzo prężną społeczność cały czas rozwijającą grę o nowe tryby i sposoby spędzania w niej czasu. W tej pracy skupimy się tylko na jednej, najbardziej podstawowej konfiguracji rozgrywki, czyli gry w piłkę nożną jeden na jednego na planszy stworzonej przez twórców gry o nazwie "Small". Sposobem w jaki gracze mogą ingerować w środowisko jest sterowanie dyskiem w kolorze swojej drużyny z wykorzystaniem klawiszy kierunkowych kontrolera oraz klawisza specjalnego pozwalającego na kopnięcie piłki w kierunku oznaczonym przez wektor od środka koła gracza do środka koła piłki z ustaloną przez grę siłą pod warunkiem, że gracz znajduje się w odpowiednio małej odległości od piłki.



Rysunek 2: Początkowa konfiguracja podstawowej planszy "Small" gry HaxBall z perspektywy gracza czerwonego.

W skład planszy przedstawionej na rysunku (2) wchodzi:

- Tło - zielony obszar otaczający planszę. Po tle mogą poruszać się gracze, ale może znaleźć się tam piłka.
- Pole gry - prostokątny, zielony, paskowany obszar pomiędzy bramkami otoczony białą linią po którym może poruszać się piłka. Piłka odbija się od białych linii oprócz odcinka pomiędzy słupkami bramki.
- Piłka - biały dysk z czarny obramowaniem widoczny na samym środku planszy z rysunku (2).

- Bramki - figury z czarnymi łukami umiejscowione po lewej i prawej stronie pola gry zakończone kołami w kolorze drużyny. Koła te imitują słupki bramki. Wnętrze bramki jest jedynym miejscem w kolorze tła po którym może poruszać się piłka.
- Gracze - czerwone lub niebieskie dyski z ustalonym przez gracza awatarem znakowym (w powyższym przypadku są to znaki "1"). Wokół dysku gracza z którego perspektywy oglądane jest środowisko widnieje szara obwódka pomagająca zidentyfikować go w środowisku. Dodatkowo, czarna obwódka wokół dysku gracza zmienia kolor na biały, jeśli gracz obecnie wykonuje akcję specjalną - kopnięcie. Pod każdym graczem równieśniczym przedstawiona jest jego nazwa, którą każdy wprowadza przed dołączeniem do rozgrywki.

Każdy epizod rozpoczyna się w przedstawionej na rysunku (2) konfiguracji. Ważnym terminem który wprowadzę oraz będę się posługiwał na przestrzeni tej pracy jest termin drużyny atakującej oraz broniącej (lub gracza atakującego lub broniącego). Drużynę atakującą można określić drużyną rozpoczynającą rozgrywkę i od drużyny broniącej różni się możliwością wkroczenia w szary, otaczający piłkę okrąg na samym środku planszy. Broniący nie będą mogli wejść w pole tego okręgu dopóki atakujący nie poruszą piłki. W prawdziwej rozgrywce strona atakująca i broniąca zmienia się w zależności od tego, która drużyna straciła punkt. W mojej pracy, by w delikatnym stopniu uprościć obsługę środowiska, po zarejestrowanym punkcie zdobytym przez byle którą drużynę, środowisko jest resetowane i zawsze stroną atakującą jest drużyna czerwona, a drużyna niebieska jest stroną broniącą.

Pomocniczym aspektem gry HaxBall jest możliwość uruchomienia rozgrywki w trybie "Headless", czyli w trybie bez konieczności konsolidacji wszystkiego co dzieje się na planszy z zewnętrznym serwerem gry [27]. Rozgrywka uruchamiana jest poprzez ustawienie w konsoli przeglądarki odpowiednich pól i wywołań zwrotnych klas związanych z grą oraz wywołanie specjalnych funkcji będąc tym samym na stronie internetowej podanej w przypisie [27]. Cała konfiguracja skonstruowana jest w oparciu o język programowania JavaScript. Plik zawierający serię komend uruchamiających pokój rozgrywki w konfiguracji wykorzystywanej w tej pracy zamieszczony jest w pliku "hax.js". Składa się on z przykładowej, podanej przez twórców, konfiguracji w której zmieniona została tylko wykorzystywana plansza oraz dopisano dwa wywołania zwrotne resetujace środowisko przy zarejestrowanym punkcie oraz przy napisaniu na czacie gry litery "r" przez byle jakiego gracza. Po dokładny opis wszystkich możliwych ustawień pokoju rozgrywki odsyłam do dokumentacji HaxBall Headless [28].

Dodatkowo w pracy wykorzystywana jest pewna implementacja gry HaxBall *haxballgym* stworzona i udostępniona przez użytkownika GitHub Wazarr94 w repozytorium pypi.org [29, 30] specjalnie na potrzeby wykorzystania w badaniach związanych z uczeniem maszynowym. Implementacja ta nazywana dalej Symulatorem pozwala na uruchomienie środowiska gry HaxBall na wybranej planszy bez konieczności jej renderowania, co znacząco przyspiesza wykonywanie w porównaniu do metody ekstrakcji obrazu. Opis tej i poprzedniej metody pracy na środowisku gry HaxBall omówię szczegółowo w poświęconych im podrozdziałach.

3.2 Opis projektu oraz implementacja obsługi środowiska i algorytmu

Projekt został napisany w sposób umożliwiający sprawną i czytelną modyfikację wszelkich jego aspektów oraz proste tworzenie nowych skryptów uruchomieniowych w języku Python. Trzonem struktury projektu jest kilka interfejsów zawierających konieczne do zaimplementowania metody, pola i wywołania zwrotne. Kluczowymi są również klasy pomocnicze w module *utils* implementujace między innymi:

- dostosowujące się do platformy mechanizmy nasłuchiwania na naciśnięcia klawiszy klawiatury jak i programistyczne wysyłanie sygnałów naciskania klawiszy klawiatury do komputera (klasa *KeyboardListener* oraz *KeyboardController* z modułu *utils.keyboard*),
- mechanizm synchronizacji wykonywania akcji ograniczający wykonywaną liczbę akcji na sekundę do konkretnych wartości, wykorzystywany w środowisku opartym na analizie obrazu (klasa *Synchronizer* z *utils.synchronizer*),
- mechanizmy wyświetlające sformatowane informacje o pracującym algorytmie (metody *format[...]* z *utils.formatters*),
- mechanizm kontroli wykonywania skryptu uruchamiającego który nasłuchuje na wywołanie komendy (naciśnięcie klawisza lub kombinacji klawiszy) po której wykonywanie algorytmu powinno się zakończyć (klasa *Runner* z *utils.runner*),
- klasa odpowiedzialna za przechowywanie oraz zapis w porcjach informacji o momentach decyzyjnych celem późniejszej analizy (klasa *Statistics* z *utils.statistics*),
- klasa zawierająca metody obsługujące przechowywanie i pobieranie informacji o wspomnieniach agenta wykorzystywanych później w procesie uczenia (klasa *Memory* i *Experience* z *utils.memory*).

Przejdę teraz do opisu poszczególnych interfejsów projektu oraz klas podrzędnych je implementujących. Pierwszym interfejsem który opiszę jest *Environment* będący fundamentem budowy obsługi środowiska gry. Konstruktor przyjmuje wartość *timeToLive*, czyli długość epizodu liczona w ilości momentów decyzyjnych oraz deklaruje informacje pomocnicze, liczenie długości obecnego epizodu i ilości wykonanych epizodów, wartości *age* i *episodes*. Pole *lastState* odpowiedzialne jest za przetrzymywanie ostatnio pobranego stanu gry w dowolnym formacie, więcej o przydatności tego pola w rozdziałach opisujących klasy podrzędne interfejsu *Environment*. Wewnątrz interfejsu zadeklarowano domyślne wartości ściśle związane z wykorzystywaną na przestrzeni tej pracy konfiguracją środowiska HaxBall (przy chęci zmiany konfiguracji należy wziąć pod uwagę zwykłe przeciążenie tych wartości) oraz podklasy wykorzystywane w jego metodach i z nim związane, tymi podklasami są:

- *Action* - enumeracja zawierająca wszelkie udostępnione i możliwe do przetworzenia przez środowisko akcje.
- *Team* - enumeracja określająca przynależność gracza do drużyny, wykorzystywana między innymi przy określaniu wartości nagrody.

- *Reward* - pomniejsza klasa przechowująca informacje dotyczące sumarycznej wartości nagrody, domyślnych wartości nagród za konkretny rezultat epizodu, składowych nagrody i wartość boolowska będąca indykatorem ukończenia środowiska.
- *State* - klasa przechowująca informacje o stanie środowiska. W skład stanu wchodzą trzy obiekty klasy *MapObject* określające położenie oraz wektor ruchu graczy oraz piłki na planszy. Stan jest możliwy do znormalizowania przy pomocy metody *normalize* celem wprowadzenia stanu jako danych do sieci neuronowej. Udostępnia on również metodę *toStateVector* parsująca obiekt klasy *State* na listę wartości z nim związanych w odpowiedniej predefiniowanej kolejności.

Taka struktura projektu w postaci podklas wykorzystywana jest również w innych interfejsach i podobnie jak wartości domyślne środowiska można je przeciążyć podczas implementacji interfejsu nazywając je tak samo i po nich dziedzicząc. Najważniejszymi metodami interfejsu *Environment* są metody:

- *doAction* - metoda przyjmująca dwie wartości typu *Action*, jest odpowiedzialna za wykonywanie wybranych akcji graczy w obecnym momencie decyzyjnym.
- *getState* - metoda pobierająca i zwracająca obecny stan środowiska.
- *getReward* - metoda zwracająca obiekt klasy *Reward* z wyliczonymi na podstawie stanu i wybranej drużyny wartościami.

Dodatkowo w interfejsie zawarte są dwa wywołania zwrotne *onLearn* i *dispose* przeznaczone do wykonania odpowiednio na początku procesu uczenia jak i na koniec wykonywania całego skryptu. Istnieją również dwie statyczne metody pomocnicze, *distance* wykorzystywana przy obliczaniu wartości nagrody oraz *getRewardComponentsKeys* zwracająca klucze wykorzystywane przy wypełnianiu pola *components* obiektu *Reward* związanego ze składowymi wartości nagrody.

Przed przejściem do opisu każdego z zaimplementowanych środowisk oraz późniejszym ich wykorzystaniu, należy przyjrzeć się szczegółowo funkcji nagrody *getReward* zaimplementowanej w interfejsie *Environment* i której pseudokod przedstawiony został poniżej. Na podstawie owej funkcji oraz jej dynamiki w środowisku budowana będzie cała polityka agenta. Można powiedzieć, że jest to najważniejszy i najbardziej wrażliwy aspekt całego procesu uczenia. Proces modelowania funkcji nagrody, tego za co nagradzać agenta i jak wyważone powinny być komponenty, nazywany jest kształtowaniem nagrody (reward shaping). W tym przypadku wartość nagrody obliczana jest poprzez operacje iloczynów skalarnych i wyliczeń odległości pomiędzy punktami na planszy (szczegóły poniżej), a cała funkcja nagrody przedstawiona jest w postaci gry o sumie zerowej (zero-sum game). W skrócie, komponentami nagrody są odległości graczy od piłki oraz kąt nachylenia ruchu piłki w stronę bramek, a ta sama wartość nagrody nadawana jednemu agentowi jest odejmowana drugiemu. Na wczesnym etapie prac zaimplementowany był również komponent nagradzający za odległość piłki od bramek, lecz został on odrzucony w czasie kształtowania nagrody ze względu na bardzo mały wpływ na politykę, brak sensownego wyważenia i niepotrzebne zwiększenie chaotyczności funkcji nagrody.

Pseudokod funkcji nagrody zaimplementowanej w interfejsie *Environment*

K0: Opis

K0.0: Wejście: stan środowiska S, kolor drużyny T.

K0.1: Wyjście: informacje o nagrodzie R.

K1: Jeśli środowisko jest przestarzałe, to zwróć R z informacją o zakończeniu środowiska, brakiem komponentów oraz domyślną dla remisu wartością nagrody.

K2: Zadeklaruj pustą strukturę przechowującą komponenty nagrody C.

K3: Na podstawie drużyny T oraz znanych współrzędnych ustal współrzędne pozycji środka bramki swojej i przeciwnika.

K4: Jeśli współrzędna x piłki pobrana z S jest równa lub większa niż znana współrzędna x bramki niebieskiej, to zwróć R z informacją o zakończeniu środowiska, brakiem komponentów oraz domyślną wartością nagrody zależną od drużyny T.

K5: Jeśli współrzędna x piłki pobrana z S jest równa lub mniejsza niż znana współrzędna x bramki czerwonej, to zwróć R z informacją o zakończeniu środowiska, brakiem komponentów oraz domyślną wartością nagrody zależną od drużyny T.

K6: Przypisz komponentowi "piłka - bramka" wartość 0.

K6.1: Jeśli wektory dx i dy piłki z S są zerowe, to przejdź do K7.

K6.2: Oblicz iloczyn skalarny dla wektorów ruchu piłki i pozycji środka bramki swojej i przeciwnika.

K6.3: Oblicz różnicę iloczynów skalarnych odejmując wynik dla bramki sojuszniczej od wyniku dla bramki przeciwnika.

K6.4: Przypisz komponentowi "piłka - bramka" w C wartość wyniku wcześniej obliczonej różnicy przemnożonej przez wagę równą 5e-4.

K7: Przypisz komponentowi "gracz - piłka" wartość 0.

K7.1: Jeśli wektory dx i dy gracza w kolorze drużyny T są zerowe to przejdź do K8.1.

K7.2: Oblicz odległość d1 piłki do gracza.

K7.3: Oblicz odległość d2 piłki do pozycji gracza powiększonej o połowę wektora ruchu gracza.

K7.4: Powiększ komponent "gracz - piłka" w C o wartość różnicy d1 i d2.

K8.1: Jeśli wektory dx i dy przeciwnika w kolorze przeciwnym do drużyny T są zerowe to przejdź do K9.

K8.2: Oblicz odległość d1 piłki do przeciwnika.

K8.3: Oblicz odległość d2 piłki do pozycji przeciwnika powiększonej o połowę wektora ruchu przeciwnika.

K8.4: Pomniejsz komponent "gracz - piłka" w C o wartość różnicy d1 i d2.

K9: Przemnóż komponent "gracz - piłka" w C przez wagę równą 8e-2.

K10: Zwróć R bez informacji o zakończeniu środowiska, z zebranymi komponentami C i ich sumą.

3.2.1 Ekstrakcja i analiza obrazu z gry

Klasa obsługująca tytułowe zagadnienie ekstrakcji obrazu to klasa podrzędna *ImageExtractingEnvironment* implementująca interfejs *Environment*. Zamysłem samego mechanizmu ekstrakcji jest ciągłe pobieranie obrazu z określonej części ekranu zawierającego pole gry, wyszukiwanie na nim konkretnych elementów rozgrywki oraz określenie ich pozycji na obszarze planszy. Przyglądając się całej implementacji od początku:

1. W konstruktorze klasy określane jest umiejscowienie górnego lewego rogu pola gry na ekranie (pola *top* i *left*), żądana maksymalna ilość akcji na sekundę możliwych do wykonania (*aps*), inicjalizacja obiektu klasy pobierającej zrzut ekranu *screenRecorder* z modułu zewnętrznego *mss*, do pamięci wczytywane są szablony (obrazy) trzech obiektów które będą wyszukiwane na zrzucie ekranu (tymi obiektami jest dysk piłki i obu graczy) oraz inicjalizowany jest obiekt klasy *Synchronizer* z podaną w argumencie ilością akcji na sekundę.
2. W funkcji *getState* najpierw pobierany jest zrzut ekranu pola gry z którego wycinany jest kanał alfa (metoda *__getFrame*). Pozycja zrzutu określana jest przez metodę *getMonitor*.
3. Pobrany zrzut ekranu parsowany jest na wartości stanu z wykorzystaniem funkcji wyszukującej szablony w obrazie *matchTemplate* z modułu *opencv2*. Przeszukiwanie jest asynchroniczne z wykorzystaniem trzech wątków i w zależności od istnienia wartości zmiennej *lastState* następuje albo w obrębie całego obrazu albo w obrębie części wokół ostatniej pobranej pozycji obiektu, co kilkukrotnie przyspiesza wyszukiwanie pojedynczego elementu.
4. W zależności od ustawienia zmiennej boolowskiej *bindState* obecnie pobierany stan jest zapisywany w zmiennej *lastState* co pozwala na późniejsze rozszerzenie stanu o wektory ruchu obiektów.
5. Określone położenia wszystkich obiektów pozwalają na zwrócenie odpowiednio wypełnionego obiektu klasy *State* obiektami klasy *MapObject*.

W przypadku tej jak i następnej klasy opisanej w kolejnym podrozdziale, zmienna *lastState* wykorzystywana jest przy przechowywaniu współrzędnych x i y wszystkich obiektów na planszy, co pozwala na późniejsze określenie wektorów ruchu obliczając je poprzez różnicę $currentState - lastState$. W przypadku braku wartości *lastState* przyjmuje się stacjonarność wszystkich obiektów, czyli zerowość wektorów ruchu. Klasa środowiska oparta na ekstrakcji obrazu z początku była głównym mechanizmem wykorzystywanym przy próbkowaniu środowiska, ale wyparta została przez bardziej wygodne i dużo mniej czasochłonne podejście. Środowisko oparte na ekstrakcji obrazu nie jest jednak bez znaczenia, gdyż na końcowym etapie eksperymentów związanych ze środowiskiem pełni ona równie kluczową rolę wizualizatora osiągniętej wydajności.

3.2.2 Symulator

Drugą klasą opartą na interfejsie *Environment* jest klasa *SimulationEnvironment*. Jest to klasa obsługująca treningową wersję gry HaxBall - *haxballgym*. Na przestrzeni tej pracy mechanizm potocznie nazywany będzie symulatorem. Wersja ta została zaimplementowana przez użytkownika Wezarr94 [29] specjalnie na potrzeby zastosowań w uczeniu maszynowym. Nie obsługuje ona graficznego interfejsu gry, a wyłącznie całą logikę oraz fizykę gry i pozwala na programistyczne podejście do informacji możliwych do wygenerowania w trakcie rozgrywki. Wykorzystanie tego rozwiązania znacząco przyspiesza proces próbkowania środowiska i co za tym idzie, uczenia algorytmu.

Konstruktor klasy *SimulationEnvironment* przyjmuje wartość *timeToLive* (opisywaną przy okazji poprzedniego mechanizmu próbkowania środowiska) oraz nazwę pliku zawierającego konfigurację, czyli informacje o zasadach gry, początkowym ustawieniu oraz rozmiarach i kształcie planszy. Domyślnie jest to *small.hbs*, czyli konfiguracja planszy przedstawionej na rysunku (2). Tylko na tej konfiguracji opierać będzie się ta praca. Ważną uwagą w przypadku zamiaru pracy symulatorem jest, że wyżej wspomniana konfiguracja w postaci pliku może nie być dostępna w świeżo zainstalowanym pakiecie *haxballgym*, dlatego umieszczona została ona w folderze *utils* projektu i w przypadku wystąpenia błędu wskazującego, że plik *small.hbs* nie istnieje pod daną ścieżką, należy go tam umieścić. Przechodząc dalej, konstruktor po otrzymaniu parametrów inicjalizuje klasę nadrzędną oraz tworzy grę (*gym*) z wykorzystaniem funkcji *haxballgym.make* podając jako argument obiekt klasy *haxballgym.Game*. Reszta argumentów nie jest podawana, gdyż nie leżą one w obszarze zapotrzebowań tego projektu, co sprawia że przyjmują one wartości domyślne. Na sam koniec, konstruktor profilaktycznie resetuje środowisko przy pomocy metody *game.reset*.

Metoda *getState* implementuje pobieranie stanu środowiska w nieco odmienny niż zamierzony przez twórcę środowiska sposób. Powodem jest konieczność dostosowania pobieranych z symulatora danych do znanych i działających w tym projekcie ram i praktyk z mechanizmu ekstrakcji obrazu. Celem utożsamienia ze sobą pobieranych z obu mechanizmów stanów i możliwości wykorzystywania ich przemiennie (na przykład do przedstawiania wyników) dane otrzymywane są z prywatnych pól symulatora i są poddawane odpowiednim przekształceniom. Sam symulator jest w stanie obsłużyć większą ilość graczy jednak na potrzeby tego projektu istnieją tylko odwołania do dwóch pierwszych graczy (pierwszy z drużyny atakującej, drugi z broniącej). Każda pozycja obiektu na planszy ustalona przy pomocy symulatora jest przesuwana o wartości x i y równe połowie szerokości i wysokości planszy. Dalsza obsługa wektorów ruchu jest identyczna jak w przypadku klasy *ImageExtractingEnvironment* (wykorzystanie pola *lastState*). Metoda *doAction* zaimplementowana w sposób trywialny, wykorzystuje metodę *step* klasy *Gym* podając akcje które ma wykonać każdy z graczy, które uprzednio są parsowane na zakodowaną postać wektorową (funkcja statyczna *parseAction*) o wartościach od 0 do 2 na pierwszych dwóch pozycjach i 0 lub 1 na trzeciej. Sama funkcja *step* zwraca wiele informacji dotyczących środowiska oraz funkcji nagrody uzyskiwanej przez agentów, jednak są to informacje zbędne patrząc z perspektywy tego projektu.

3.3 Implementacja i wykorzystanie algorytmu PPO w środowisku

Chcąc omówić istotę algorytmu PPO wykorzystywanego w tym środowisku, przybliżę wpierw interfejs *PPOModel* oraz wykorzystującą go jedną z klas implementującą interfejs *Agent*. Zaczynając od interfejsu *Agent*, jego jedyną metodą jest *chooseAction*. Mechanizm podejmowania akcji oparty jest właśnie na tej metodzie i różni się on w zależności od klasy implementującej, te klasy to:

- *RandomAgent* - klasa w której funkcja wybierająca akcje zwraca je w sposób losowy z równym prawdopodobieństwem.
- *StationaryAgent* - klasa w której zwracana jest zawsze akcja *Action.NO* równa braku akcji.
- *PPOAgent* - jedyna klasa spajająca środowisko z algorytmen PPO, zawiera ona pola i metody z nim związane.

Głównym zadaniem klasy *PPOAgent* jest przechowywanie zadanej ilości momentów decyzyjnych w pamięci zainicjalizowanej przy pomocy obiektu klasy *Memory*. Dodatkowo decyduje ona kiedy jest odpowiednia pora na wykorzystanie posiadanych próbek w procesie uczenia (metoda *tryToLearn* opisana w dalszym rozdziale) oraz posiada zaszyty w metodzie *chooseAction* mechanizm wyboru akcji na postawie zadanego modelu (obiektu klasy *PPOModel*). Konstruktor przyjmuje instancję klasy *PPOModel* oraz rozmiar pamięci momentów decyzyjnych. Funkcje logiczne (zwracające wartości boolowskie) *canLearn* oraz *canForceLearn* sprawdzają, czy w pamięci istnieje dostateczna do przeprowadzenia procesu uczenia ilość spróbkowanych momentów decyzyjnych. Funkcje *learn* oraz *chooseAction* są funkcjami obsługującymi model. Metoda *learn* odpowiedzialna jest za uruchomienie procesu uczenia na podstawie wprowadzonych próbek. Dodatkowo zaznacza wykorzystanie planera (mechanizmu do planowania procesu uczenia opisanego w dalszej części pracy) oraz po przeprowadzonym uczeniu czyści wykorzystaną już pamięć i ostatecznie zwraca otrzymane z procesu uczenia błędy sieci aktora oraz krytyka w postaci dwóch instancji podklas *PPOModel.Loss*. Funkcja *chooseAction* jest bezpośrednim wykorzystaniem wyników zwracanych przez sieć, a jej jedynym argumentem jest obiekt klasy *State* przetrzymujący wartości stanu, zwraca natomiast, oprócz samej akcji do wykonania, informacje wykorzystywane później w procesie uczenia: prawdopodobieństwo wykonania wybranej akcji, wartość zwracaną przez sieć krytyka oraz prawdopodobieństwa wykonania wszystkich możliwych akcji celem późniejszego zestawienia. By móc przejść dalej, przyjrzymy się jakie informacje przechowywane są w pamięci agenta, a dokładniej jak wygląda struktura klasy *Memory* i co zawiera pojedynczy rekord wspomnienia - *Experience*.

Klasa *Memory* składa się z serii struktur zwanych *deque*. Struktura charakteryzuje się ogranicznikiem ilości przetrzymywanych w niej danych. Gdy dodawany jest do niej nowy rekord, ale ich liczba przekroczyłaby ustalony limit, usuwany jest z niej "najstarszy" z nich, a nowy przychodzi na "najmłodsze" miejsce. Mechanizm ten naturalnie kojarzy się z pamięcią w której nowe wspomnienia wypychają te najstarsze. Całą klasę inicjalizuje się właśnie takim ustalonym rozmiarem pamięci. Przyglądając się polom klasy *Experience* wyszczególnić możemy przechowanie następujących informacji:

- *state* - stan środowiska, wektor zawierający uporządkowane informacje, generowany na podstawie obiektu klasy *State* przy pomocy metody *toStateVector*;
- *normalizedState* - znormalizowany wektor stanu, normalizacja następuje przy wykonaniu metody *toStateVector* z argumentem *normalized* o wartości *True* na obiekcie *State*;
- *reward* - sumaryczna wartość nagrody zwracanej przez metodę *getReward* z obiektu środowiska;
- *rewardComponents* - komponenty składowe wartości nagrody nadawane za konkretne zachowania (takie jak przybliżanie się do piłki i kopanie jej do bramki) wykorzystywane przy określaniu statystyk związanych z wartością nagrody;
- *done* - określenie, czy to wspomnienie (moment decyzyjny) jest wspomnieniem terminalnym, kończącym epizod;
- *actionIndex* - zakodowany indeks wykonanej akcji;
- *val* - wartość zmiennoprzecinkowa zwrócona przez krytyka po predykcji z wykorzystaniem znormalizowanego stanu z tego wspomnienia;
- *prob* - prawdopodobieństwo wykonania akcji określonej przez *actionIndex* w tym stanie według sieci aktora.

Znając już środowisko, strukturę wspomnień oraz metodę działania i nagradzania agenta możemy przejść do omówienia interfejsu *PPOModel* zawierającego cały mechanizm uczący, w tym implementację algorytmu PPO. Implementując ten interfejs możemy określić własnoręcznie wszystkie hiperparametry uczenia, ale posiada on pewne domyślne wartości. Kluczowymi hiperparametrami zadeklarowanymi już na początku istnienia obiektu implementującego *PPOModel* są:

- lrA, lrC - wskaźniki uczenia sieci neuronowych aktora i krytyka, wprowadzane one są do osobnie zainicjalizowanych optymalizatorów *Adam* za pomocą których sieci są uczone;
- discountFactor - współczynnik dyskontowy, wykorzystywany przy wyliczaniu wartości funkcji przewagi;
- lambdaVal - wartość lambda, również wykorzystywana przy wyliczaniu wartości funkcji przewagi;
- batchSize - rozmiar próbki (trajektorii) wprowadzanej do algorytmu uczącego;
- epochs - ilość iterowań pojedynczej próbki, wskaźnik wydajności próbki;
- clip - wartość przycięcia prawdopodobieństw w algorytmie PPO-Clip omówiona w rozdziale jej poświęconym.

Dodatkowo przypisywanymi mu wartościami są referencje do dwóch modeli z modułu *tensorflow*, nazwane *actor* i *critic* odpowiednio dla modelu sieci neuronowej aktora i krytyka. Dalej przedstawione są pola-liczniki ustalające etap planera uczenia *stage*, liczbę przeprowadzonych sesji uczenia *learningSessions* oraz wartość *saveWeightsPerLS* mówiąca o tym, co ile sesji uczenia dokonywany będzie zapis wag modelu do pliku. Koniecznymi informacjami są również: nazwa modelu *name*, rozmiary przestrzeni stanu i akcji pobierane z interfejsu *Environment* pod nazwami *stateShape* i *actionQuantity* oraz opcjonalne ścieżki do zapisanych wag modeli możliwych do wczytania.

Konstruktor *PPOModel* przyjmuje wspomniane wcześniej opcjonalne ścieżki do zapisanych wag modelu, liczbę wykonanych już na zadanym modelu sesji uczenia (najczęściej ta wartość podawana jest jako argument uruchomieniowy) oraz nazwę modelu. Wartości te są zapisywane w polach obiektu, a dalej wywoływane są metody odpowiedzialne za określenie rozmiarów przestrzeni stanu i akcji, budowa modelu aktora i krytyka, wczytanie do nich wag oraz początkowe określenie etapu planera. Przechodząc do metod interfejsu *PPOModel*, zacznę od tych mniej istotnych. Interfejs udostępnia szereg funkcji ustawiających hiperparametry algorytmu znajdujących zastosowanie w planerze. W większości działają one w sposób trywialny, przypisując zadaną wartość do pola, wyjątkiem są wartości współczynników uczenia przy których zmianie konieczne jest ponowne zainicjalizowanie optymalizatora *Adam* modelu. Metody obsługujące zapis i wczytywanie wag modelu to kolejno *saveWeights* oraz *loadWeights*. Do zapisu wykorzystywana jest podana w inicjalizatorze nazwa modelu konieczna do określenia (i jeśli zachodzi taka potrzeba - utworzenia) podkatalogu w którym zapisywane są owe wagi. Nazwa pojedynczego pliku zawierającego wagi ma rozszerzenie *.hdf5* i sformatowana jest następująco:

"[Nazwa Modelu]-[Liczba ukończonych sesji]-[Stempel czasowy]-[actor/critic].hdf5".

Metody *actorPredict* i *criticPredict* przyjmujące obiekty typu *State* wykonują odpowiednie metody celem pobrania wyniku odpowiadających im sieci neuronowych. Dokonują rzutowania potrzebnego do normalizacji i przygotowują wymiarowość wymaganą do wprowadzenia danych o stanie środowiska do sieci, następnie z pobranych wyników wycinane są niepotrzebne wymiary i z metody zwracana jest najpotrzebniejsza informacja. By przedstawić zamierzone działanie metody tworzącej modele sieci neuronowych przedstawię jej przykładową deklarację z klasy implementującej *SmallPPOModel*, której nazwa odzwierciedla charakterystykę tworzonych modeli.

```
1  def buildModels(self) -> (Model, Model):
2      AinputL = Input(shape=(self.stateShape,))
3      AhiddenL1 = Dense(64, activation="tanh")(AinputL)
4      AhiddenL2 = Dense(64, activation="tanh")(AhiddenL1)
5      AprobabilityL = Dense(self.actionQuantity, activation="softmax")(AhiddenL2)
6
7      CinputL = Input(shape=(self.stateShape,))
8      ChiddenL1 = Dense(64, activation="tanh")(CinputL)
9      ChiddenL2 = Dense(64, activation="tanh")(ChiddenL1)
10     CqValueL = Dense(1, activation=None)(ChiddenL2)
11
12     actor = Model([AinputL], [AprobabilityL])
13     critic = Model([CinputL], [CqValueL])
14
15     return actor, critic
```

Rysunek 3: Przykładowa implementacja funkcji budującej modele aktora i krytyka - *buildModels*.

Odwołując się do rysunku (3), w liniach 1 i 2 następuje oczywista deklaracja metody oraz importowanie potrzebnych w tym przypadku warstw sieci. Dalej następują dwie prawie analogiczne deklaracje modeli. Zadeklarowana w linijce 3 oraz 8 warstwa wejściowa (*Input*) przyjmuje informacje o wymiarowości podanej jako argument *shape*. Wartość *self.stateShape* jest określana przy inicjalizacji i pobierana jest z klasy *Environment.State*. Jest ona zgodna z rozmiarem wektora stanu środowiska. W liniach 4-5 i 9-10 deklarowane są po dwie identyczne warstwy gęste (*Dense*) zawierające 64 neurony każda. Funkcją aktywacji jest funkcja hiperboliczna - *tanh*. Jedyną różnicą pomiędzy tymi modelami są warstwy wyjściowe zadeklarowane w liniach 6 i 11. Dla modelu aktora rozmiar wyjściowy odpowiada liczbie akcji możliwych o wykonania zawartej w zmiennej *self.actionQuantity*, a wartość wynikowa warstwy określana jest przy pomocy funkcji aktywacji *softmax* (zwracane wartości sumują się do jedynki). W przypadku modelu krytyka, rozmiar warstwy wyjściowej jest stały i równy 1 oraz nie jest ona oparta o żadną funkcję aktywacji, zwraca czystą wartość liczbową. Posiadając zadeklarowane sekwencje warstw możemy zainicjalizować obiekt klasy *Model* podając jako argument warstwy wejściowe oraz wyjściowe oraz zwrócić je z omawianej metody. Podsumowując działanie metody *buildModels*, istnieje pełna dowolność w konstrukcji modelów, wybranych warstw, ich rozmiarów oraz wykorzystywanych funkcji aktywacji. Podejściem wartym uwagi jest deklaracja obu modeli jako jedna, ta sama sieć, niestety projekt nie został przygotowany z takim zamierzeniem i mogą wystąpić błędy. Wymagałoby to lekkich zmian w metodzie uczącej, a dokładniej w wykorzystaniu uzyskiwanych podczas uczenia błędów.

Do planowania zmian w domyślnej hiperparametryzacji zawartej w obiekcie klasy *PPOModel* na początku procesu uczenia oraz w jego trakcie możemy wykorzystać metodę *usePlanner*. Poniżej przedstawiono przykładową implementację tej metody.

```
1  def usePlanner(self):
2      if self.stage == −1:
3          self.setBatchSize(8)
4          self.setActorLearningRate(1e−5)
5          self.setCriticLearningRate(1e−4)
6          self.setEpochs(8)
7          self.stage = 0
8
9      if self.stage == 0 and self.learningSessions >= 50_000:
10         self.setActorLearningRate(5e−6)
11         self.setCriticLearningRate(5e−5)
12         self.stage = 1
```

Metoda ta jest wywoływana po każdej sesji uczenia. Pole *stage* klasy *PPOModel* wspomaga planowanie określając kolejne etapy planera. Przejście do etapu deklarowane jest w postaci warunkowych sekcji kodu, ważne jest by w każdej z nich dokonywać zmiany etapu, czyli wartości pola *stage*. Pierwszą wartością zmiennej *stage* po uruchomieniu projektu jest zawsze wartość -1. Cały mechanizm działa w sposób przyrostowy, nadpisując wszystkie dotychczas wprowadzone zmiany. Dla przykładu, jeśli obiekt klasy implementującej *PPOModel* korzystający z powyższego planera został zainicjalizowany z wartością *learningSessions* powyżej 50_000, najpierw uruchomiona zostanie sekcja z warunkiem $self.stage == -1$ (gdyż początkowa wartość pola *stage* to -1) określająca współczynnik uczenia, wielkość próbki oraz jej wydajność, następnie w tym samym wywołaniu sekcja warunkowa następnego etapu nadpisze wartości współczynnika uczenia pozostając przy ustalonej wielkości oraz wydajności próbki. Powyżej przedstawiony kod to tylko przykładowy sposób działania, którego implementacja może być zróżnicowana. Nie poleca się jednak modyfikacji (nawet początkowej) hiperparametrów poza przedstawioną powyżej metodą.

```
1  def calculateAdvantages(self, batch, memory: Memory):
2      advantages = []
3      normRewards = np.array(memory.rewards)
4      normRewards = (normRewards − normRewards.mean()) \
5                    / (normRewards.std() + 1e−10)
6
7      for t in range(memory.memories − batch, memory.memories):
8          advantage = 0
9          reduction = 1
10
11         for i in range(t, memory.memories − 1):
12             reward = norm_rewards[i]
13             done = memory.dones[i]
14             val = memory.vals[i]
15             nextVal = memory.vals[i+1]
16             advantage += reduction ∗ \
17                     ((reward + self.discountFactor ∗ nextVal) − val)
18             reduction = reduction ∗ self.discountFactor ∗ self.lambdaVal
19
20         if t == memory.memories − 1:
21             advantage = normRewards[t]
22
23         advantages.append(advantage)
24
25     advantages = np.array(advantages)
26     advantages = (advantages − advantages.mean()) / (advantages.std() + 1e−10)
27
28     return advantages
```

Przed samym przybliżeniem funkcji uczącej algorytmu omówię metodę *calculateAdvantages* przez nią wykorzystywaną. Tak jak przedstawiono w rozdziale poświęconym algorytmowi PPO, wyliczenie wartości funkcji przewagi odbywa się przy pomocy wzoru (10). Jest to różnica dwóch funkcji, z których pierwsza określa wartość nagrody w danym stanie po wykonaniu określonej akcji, druga natomiast określa wartość nagrody w tym stanie przed wykonaniem jakiejkolwiek akcji. Warto wspomnieć, że wzór (10) jest koniecznym trzonem funkcji przewagi, lecz nie istnieje jedna, idealna, deklaracja owej funkcji. Z racji, że wszelkie kalkulacje potrzebne do wyliczenia wartości funkcji przewagi opierają się jednocześnie na wartościach nagrody otrzymywanych przez agenta w środowisku oraz przybliżeń zwracanych przez sieć neuronową krytyka, wyniki te z natury są bardzo chaotyczne, dodatkowo więc metoda ta jak najbardziej stara się tę chaotyczność zniwelować. Metoda przyjmuje dwa argumenty: liczbę ostatnich wspomnień z których należy wyliczyć wartości oraz pamięć z której należy owe wspomnienia pobrać. Na samym początku funkcji wszystkie nagrody pobrane ze wspomnień są normalizowane (linie 3 i 4). Iteracja w linii 7 zapętla kod celem obliczenia wartości przewagi dla każdego wspomnienia zaczynając od indeksu wspomnienia określonego przez wartość różnicy ilości posiadanych wspomnień i żądanej porcji do samego końca pamięci. Wartość przewagi dla każdego wspomnienia na początku jest zerowa-

na oraz określana jest wartość redukcji przyszłych wspomnień (*reduction*). Redukcja sprawia, że relatywnie późniejsze wspomnienia mają dla jego wartości mniejsze znaczenie. Zagnieżdżona w linii 11 iteracja przetwarza wspomnienia zaczynając od tego obecnie branego pod uwagę do przedostatniego wspomnienia. Ostatnie wspomnienie jest traktowane specjalnie, wartość znormalizowanej nagrody jest przepisywana bez jakichkolwiek obliczeń (linia 20, 21). Na początku zagnieżdżonej pętli następuje wybranie potrzebnych do obliczeń wartości: *reward* - wartość nagrody zwrócona przez środowisko w tym wspomnieniu, *val* - wartość zmiennoprzecinkowa zwrócona przez sieć krytyka podczas próbkowania środowiska, *nextVal* - wartość zwrócona przez sieć krytyka w następnym w kolejności wspomnieniu. Przechodząc do kalkulacji wartości przewagi i przyrównując ją do wzoru (10), wartość funkcji $Q(s_t, a_t)$ (jakość stanu po wykonaniu akcji a) wyliczona jest na podstawie nagrody uzyskanej po przejściu do tego stanu z dodaniem predykcji krytyka (z pewnym zapasem ufności określonym przez *discountFactor*). Wartość funkcji $V(s_t)$ określająca czystą jakość stanu reprezentowana jest przez zmienną *val*, czyli pobierana jest z sieci krytyka. Wartość przewagi dla danego wspomnienia w każdej iteracji powiększana jest o redukowaną z czasem różnicę wartości obu funkcji. Sama wartość redukcji również jest pomniejszana przy pomocy zmiennych *discountFactor* oraz *lambdaVal* będących hiperparametrami algorytmu. W niektórych implementacjach funkcji przewagi pod uwagę dodatkowo brana jest zależność, czy dane wspomnienie jest wspomnieniem terminalnym, wartość nagrody takiego wspomnienia jest wtedy po prostu przepisywana. W zadeklarowanej powyżej funkcji przewagi taki przypadek nie jest uwzględniany ze względu na specyfikację projektu która zapewnia, że terminalne stany są zawsze na końcu zadanej trajektorii wspomnień i rozpatrywane są w warunku z linii 20. Wartość wyliczona dla każdego wspomnienia przechowywana jest w tablicy (indeks wartości w tablicy jest indeksem wspomnienia do którego się odwołuje) która następnie jest normalizowana, podobnie jak początkowa lista nagród. Technika normalizacji wartości funkcji przewagi niestety nie jest wspominana w cytowanej wcześniej pracy o algorytmie PPO [13], jednak jej brak wymaga ostrożniejszego podejścia do wartości nagród zwracanych przez środowisko, a zwłaszcza ich wielkości i odchyleń względem siebie. Normalizacja bardzo dobrze wpływa na redukcję chaotyczności funkcji przewagi i znacznie poprawia długoterminową wydajność systemu decyzyjnego.

Przejdę teraz do wspomnianej wcześniej funkcji uczącej *learn* zadeklarowanej w interfejsie *PPOModel*. Celem skrupulatnego jej omówienia, kod metody podzielę na części i szczegółowo omówię każdą z nich określając jej zadanie.

```
1  def learn(self, memory: Memory) -> (Loss, Loss):
```

Jedynym argumentem metody *learn* jest obiekt klasy *Memory* będący odwołaniem do pamięci agenta zadeklarowanej w obiekcie typu *PPOAgent*.

```
2      batchesQuan = memory.newMemories // self.batchSize
3      lastMemories = batchesQuan * self.batchSize
4      batch = np.arange(memory.memories - lastMemories, memory.memories)
5      np.random.shuffle(batch)
6      batches = [
7          batch[i * self.batchSize:(i + 1) * self.batchSize]
8              for i in range((len(batch) + self.batchSize - 1) // self.batchSize)
9          ]
```

Pierwszym krokiem jest przygotowanie porcji danych (trajektorii) na których będzie dokonywane uczenie. Przygotowanie opiera się na zainicjalizowaniu indeksów reprezentujących dane za pomocą których następować będzie odwoływanie do konkretnych wspomnień z pamięci. Dodatkowo zaimplementowany został mechanizm oszczędności doświadczeń pozwalający na przeprowadzenie uczenia nawet jeśli pamięć nie jest wypełniona, bądź w pełni zastąpiona nowymi doświadczeniami. Spoglądając w kod, wyliczona wartość zmiennej *batchesQuan* określa liczbę możliwych do pobrania z pamięci porcji o rozmiarze *self.batchSize* (hiperparametr). Zmienna *lastMemories* mówi o sumie doświadczeń które zostaną pobrane. Następnie, w linii 4 deklarowana jest wstępna, uporządkowana tablica indeksów (liczb naturalnych). Posiłkując się przykładem, załóżmy że w pamięci jest 100 wspomnień, z czego 50 jest nowych, a 50 zostało już użytych w poprzedniej sesji uczenia. Wymagany rozmiar próbki to 10, *batchesQuan* przyjmuje wtedy wartość 5 (50/10), a *lastMemories* wartość 50. Odwołamy się więc do ostatnich 50 wspomnień tworząc listę indeksów zaczynających się od 50 ($memory.memories - lastMemories$) do samego końca pamięci (*memory.memories*). Tak przygotowane indeksy są tasowane, a następnie w liniach (6-9) są one dzielone na porcje.

```
10     actorLosses = []
11     criticLosses = []
```

Dalej następuje inicjalizacja początkowo pustych tablic w które wprowadzane będą wartości błędów zwracane przez każdy z modeli podczas uczenia.

```
12     advantages = self.calculateAdvantages(lastMemories, memory)
```

Przy pomocy metody *calculateAdvantages*, której definicję omówiono powyżej, dla podanej ilości ostatnich wspomnień wyliczane są wartości funkcji przewagi.

```
13     for batch in batches:
14         states = memory.getStatesBatch(batch)
15         actions = memory.getActionsBatch(batch)
16         oldProbs = memory.getProbsBatch(batch)
17         vals = memory.getValsBatch(batch)
18         adv = [advantages[i - (memory.memories - lastMemories)] for i in batch]
19         adv = self.tf.convert_to_tensor(adv, dtype=self.tf.float32)
```

Dalej następuje iteracja po każdej wcześniej wylosowanej porcji indeksów. Dla każdej z nich z pamięci wybierane są odpowiadające im informacje: *states* - stany środowiska zakodowane jako znormalizowane wektory, *actions* - indeksy wykonanych w tych wspomnieniach akcji, *oldProbs* - ówczesne prawdopodobieństwa wykonania akcji ze wspomnienia, *vals* - wartości predykcji krytyka na stanach, *adv* - wartości z wcześniej wyliczonej funkcji przewagi. Wartości zmiennej *adv* w linii 19 są konwertowane na tensor celem wykonywania na nich równoległych obliczeń. Pozostałe wartości pobierane przy pomocy metod *get[...]* obiektu klasy *Memory* są podobnie rzutowane na tensor w ciele odpowiadającej im metody.

```
20          for _ in range(self.epochs):
```

Zagnieżdżona w linii 20 pętla odpowiada za wydajność pojedynczej próbki. Każda pojedyncza próbka przetwarzana jest *epochs*-razy.

```
21              with self.tf.GradientTape() as tape1, self.tf.GradientTape() as tape2:
22                  actionsProb = self.actor(states)
23                  actionsProb = self.tf.clip_by_value( \
24                              actionsProb, \
25                              clip_value_min=1e-30, clip_value_max=1e+30)
26                  newVals = self.critic(states)
27
28                  dists = self.Categorical(probs=actionsProb)
29                  newProbs = dists.prob(actions)
30                  probRatios = self.tf.divide(newProbs, oldProbs)
31
32                  advantageProbs = self.tf.multiply(adv, probRatios)
33                  clippedAdvantageProbs = self.tf.multiply( \
34                                          self.tf.clip_by_value( \
35                                              probRatios, \
36                                              1-self.clip, \
37                                              1+self.clip), \
38                                          adv)
39
40                  actorLoss = self.tf.multiply( \
41                              self.tf.constant(-1.0), \
42                              self.tf.reduce_mean( \
43                                  self.tf.minimum( \
44                                      advantageProbs, \
45                                      clippedAdvantageProbs)
46                                  )
47                              )
48                  criticLoss = self.tf.subtract(self.tf.add(adv, vals), newVals)
49                  criticLoss = self.tf.square(criticLoss)
50                  criticLoss = self.tf.reduce_mean(criticLoss)
```

Linijki od 21 do 50 są deklaracją niestandardowego procesu uczenia wykorzystywanego przez moduł *tensorflow*. Na początku z obu sieci pobierane są predykcje oparte na przygotowanych wcześniej wektorach stanów. Predykcje z sieci aktora dodatkowo są

przycinane do zakresu $[1e{-}30 : 1e{+}30]$ celem uniknięcia ryzyka dzielenia przez 0. Dalej przygotowywane są prawdopodobieństwa wykonania akcji spróbkowanych uprzednio ze środowiska by następnie wyliczyć stosunki prawdopodobieństw nowej parametryzacji (θ) względem starej (θ_{old}) jak widnieje na wzorze (15). Następny krok to wyliczenie dwóch elementów celu zastępczego (14) z których wybrany zostanie ten minimalny. Zmienna *advantageProbs* zawiera przemnożenie $r(\theta)\hat{A}_{\theta_{old}}(s,a)$, natomiast zmienna *clippedAdvantageProbs* zawiera podobne przemnożenie z przycięciem stosunków prawdopodobieństw na podstawie hiperparametru *clip* jak w części $clip(r(\theta), 1-\epsilon, 1+\epsilon)$ celu zastępczego. Wybór minimum obu tych zmiennych zagnieżdżony jest w wyliczeniu błędu sieci aktora w liniach (43-46). Przemnożenie błędu przez stałą -1.0 w liniach (40-41) jest koniecznością, ze względu na standard wykorzystywanego kroku uczenia modułu *tensorflow*. Domyślnie proces uczenia wykorzystuje metodę gradientu prostego (Gradient Descent), jednak wymogiem algorytmu PPO jest przeprowadzenie operacji odwrotnej (Gradient Ascent) co jest powodem powyższego odwrócenia wartości błędu. Błąd sieci krytyka wyliczany jest w prostszy sposób w liniach (48-51) według wzoru $(advantage + vals - newVals)^2$.

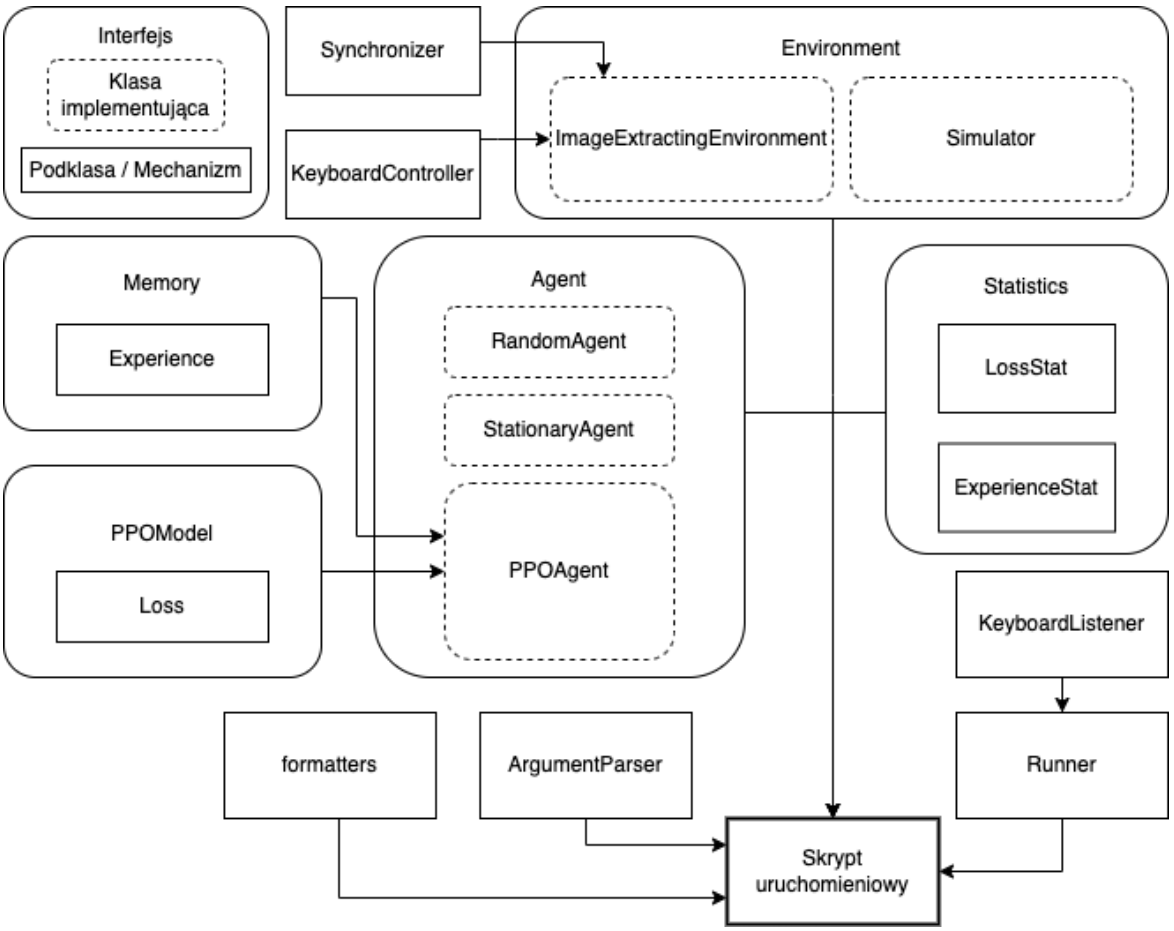
```
51              grad1 = tape1.gradient(actorLoss, self.actor.trainable_variables)
52              grad2 = tape2.gradient(criticLoss, self.critic.trainable_variables)
53              self.actorOptimizer.apply_gradients( \
54                      zip(grad1, self.actor.trainable_variables))
55              self.criticOptimizer.apply_gradients( \
56                      zip(grad2, self.critic.trainable_variables))
57              criticLosses.append(criticLoss.numpy())
58              actorLosses.append(actorLoss.numpy())
```

Tak wyliczone błędy obu sieci wykorzystywane są do przeprowadzenia metody gradientu prostego na możliwych do uczenia wartościach sieci. Są one również dopisywane do odpowiadających im list.

```
59      self.learningSessions += 1
60
61      if self.learningSessions % self.saveWeightsPerLS == 0:
62          self.saveWeights()
63
64      actorLosses = np.array(actorLosses)
65      criticLosses = np.array(criticLosses)
66      return PPOModel.Loss(
67          mean=actorLosses.mean(),
68          minimum=actorLosses.min(initial=1e+5),
69          maximum=actorLosses.max(initial=-1e+5)
70      ), PPOModel.Loss(
71          mean=criticLosses.mean(),
72          minimum=criticLosses.min(initial=1e+5),
73          maximum=criticLosses.max(initial=-1e+5)
74      )
```

Linia 59 zwiększajaca licznik sesji uczenia jest pierwszą linią poza pętlą obsługującą wylosowane porcje. W przypadku podzielności licznika sesji uczenia przez wartość *saveWeightsPerLS* wagi obu sieci są zapisywane na dysk. Rzutowanie uzbieranych w czasie sesji uczenia błędów na tablice modułu *numpy* przeprowadzane jest ze względu na wygodę wypełnienia obiektu klasy *PPOModel.Loss* wartościami statystycznymi (*min*, *max*, *mean*), które ostatecznie zwracane są z funkcji *learn*.

Podsumowując ten podrozdział przedstawiam diagram prezentujący pełną strukturę projektu.



Rysunek 4: Diagram prezentujący strukturę projektu: interfejsy, klasy, podklasy, mechanizmy i relacje pomiędzy nimi.

3.4 Deklarowanie skryptów uruchomieniowych

Znając strukturę całego projektu oraz większość klas i interfejsów możemy zaznajomić się z deklarowaniem koniecznych do uruchomienia projektu skryptów wykorzystujących wszystkie powyżej opisane mechanizmy. Przeanalizujmy stopniowo przykładowy skrypt uruchomieniowy.

```
1  args = ArgumentParser()
```

ArgumentParser jest pierwszą z klas pomocniczych wartych wykorzystania. Obsługuje ona argumenty podawane przy uruchomieniu skryptu. Klasa ta wykorzystując wbudowany w język Python moduł *argparse* inicjalizuje możliwość wykorzystania flag oraz przełączników pomagających obsługiwać zadeklarowane później modele i środowisko. Wykorzystanie tego mechanizmu jest opcjonalne, możliwe jest wprowadzanie wszystkich danych w sposób manualny, wprowadzając je w kod skryptu.

```
2  env = SimulationEnvironment(timeToLive=10 * 120)
```

To podejście korzystać będzie ze środowiska symulatora z długością pojedynczego epizodu ograniczoną do $10 * 120 = 1200$ momentów decyzyjnych.

```
3  redaldo = PPOAgent(
4      model=SmallPPOModel(
5          actorWeightsPath=args.redActorWeights,
6          criticWeightsPath=args.redCriticWeights,
7          learningSessions=args.learningSessions,
8          name="PPO_RED_VS_RANDOM_BLUE"
9      ),
10     memorySize=120,
11 )
12
13 bluessi = RandomAgent()
```

W liniach od 3 do 11 odbywa się inicjalizacja agenta opartego o model sieci neuronowej określonej w implementacji *SmallPPOModel* (struktura obu modeli określona jest na rysunku 3) oraz o długość pamięci wynoszącą 120 momentów decyzyjnych. Przy konstruowaniu modelu podawane są wartości ustalone na podstawie argumentów uruchomieniowych. Domyślnie ścieżki do wag obu sieci posiadają wartość *None* i w takiej sytuacji są one generowane losowo, a ilość sesji uczenia (*learningSessions*) jest równa 0. Wprowadzona w pole *name* nazwa modelu wykorzystywana jest przy zapisie wag do pliku. Linia 13 deklaruje agenta wykonującego losowe akcje niezależnie od postawionego stanu.

```
14 stats = Statistics(
15     sampleSize=10_000,
16     statisticsSubDir="PPO_RED_VS_RANDOM_BLUE"
17 )
```

Celem zbierania informacji o przebiegu procesu uczenia warto wykorzystać obiekt klasy *Statistics* przechowujący oraz zapisujący statystyki do wprowadzonego w polu *statistcsSubDir* katalogu. Pole *sampleSize* określa co jaką ilość zarejestrowanych momentów

decyzyjnych ma nastąpić zrzut statystyk do pliku. Na statystyki składają się nie tylko informacje o momentach decyzyjnych ale również zapisy błędów zwracanych z sieci neuronowych po sesji uczenia. Warto zauważyć, że w pojedynczym zrzucie statystyk, po osiągnęciu określonej liczby momentów decyzyjnych ilość zrzuconych informacji o błędach sieci nie jest ściśle określona i zazwyczaj w każdym zrzucie będzie się ona różnić.

```
18 runner = Runner(command="q", dualCommand=True)
```

By wprowadzić program w główną pętlę rozgrywki oraz by mieć kontrolę nad późniejszym jej opuszczeniem można wykorzystać mechanizm dostarczany przez klasę *Runner*. Wykorzystuje ona klasę *KeyboardListener*. Po wejściu w poniżej przedstawioną pętlę, w przypadku podania argumentu *dualCommand=False*, klikając na klawiaturze podany w *command* znak pętla zostanie przerwana. Mechanizm *dualCommand* rozszerza sposób przerwania pętli o konieczność wprowadzenia jeszcze jednego znaku, który otrzymamy przy przytrzymaniu klawisza *Shift* klawiatury oraz ponownym wprowadzeniu znaku z argumentu *command*. Dla powyższej deklaracji pętla zostanie przerwana po wprowadzeniu na klawiaturze kombinacji znaków "qQ".

```
19 while runner.running:
20     state = env.getState(bindState=True)
21
22     actionRed, prob, val, probs = redaldo.chooseAction(state)
23     actionBlue = bluessi.chooseAction(state)
24
25     env.doAction(actionRed, actionBlue)
26     reward = env.getReward(env.getState(bindState=False), env.Team.Red)
27
28     experience = Memory.Experience(
29         normalizedState=state.toStateVector(normalized=True),
30         state=state.toStateVector(normalized=False),
31         reward=reward.value,
32         rewardComponents=reward.components,
33         done=reward.done,
34         actionIndex=actionRed.value,
35         val=val,
36         prob=prob,
37     )
```

Po wejściu do kontrolowanej przez obiekt *Runner* pętli można rozpocząć próbkowanie środowiska. Na początku każdej iteracji pobierany jest najnowszy stan wcześniej zadeklarowanego środowiska przy pomocy metody *getState* z argumentem wiązania *bindState* ustawionym na *True*. Argument ten daje mechanizmowi sygnał, by następny pobrany w ten sposób stan związać z poprzednim celem zarejestrowania wektorów ruchu. Następnie obaj agenci dokonują decyzji co do akcji w przedstawionym im stanie. W przypadku agenta uczonego przypisywane są dodatkowe informacje wykorzystywane później przy jego uczeniu. Wybrane akcje są wprowadzane do środowiska przy pomocy metody *doAction*, a następnie z obiektu środowiska pobierany jest zestaw informacji o nagrodzie skojarzonej z kolorem drużyny agenta uczonego na podstawie ponow-

nie pobranego stanu środowiska, tym razem bez wykorzystania mechanizmu wiązania (*bindState=False*). Tak pobrany pakiet informacji jest zbierany w grupę informacji do obiektu klasy *Experience*.

```
38      print(
39          formatLearningSessionInfo(newMemories=redaldo.memory.newMemories,
40                                    memorySize=redaldo.memory.size,
41                                    learningSessions=redaldo.model.learningSessions,
                                      ) + \
42          "␣" + \
43          formatEnvironmentCompletionInfo(envAge=env.age,
44                                          envTimeToLive=env.timeToLive,
45                                          envEpisodes=env.episodes) + \
46          formatActionProbabilities(probs) + \
47          "␣" + \
48          formatAction(actionRed) + \
49          "␣" + \
50          formatReward(reward.value)
51      )
```

Gdy znane już są wszelkie informacje o tym momencie decyzyjnym, opcjonalnym, lecz przydatnym jest ich wyświetlenie w czytelnej formie na wyjściu standardowym. Wykorzystywana do tego jest komenda *print* wraz z szeregiem pomocniczych funkcji formatujących zwracających łańcuchy znaków. To jakie informacje będą wyświetlane zależy od wykorzystanych funkcji.

```
52      redaldo.memory.remember(experience)
53      stats.addExperience(experience)
```

Koniecznością w kwestii uczenia agenta jest przekazanie zgrupowanych informacji do jego pamięci oraz opcjonalnie do obiektu zajmującego się statystykami.

```
54      actorLoss, criticLoss = redaldo.tryToLearn(experience, env)
55      if actorLoss is not None and criticLoss is not None:
56          print(formatLosses(actorLoss, criticLoss))
57          stats.addActorLoss(actorLoss)
58          stats.addCriticLoss(criticLoss)
```

Dalej, w linii 54, dane momentu decyzyjnego razem z obiektem środowiska wprowadzane są do metody *tryToLearn*. W przypadku gdy metoda zwróci dwie wartości nie będące *None* oznacza to, że w tym momencie decyzyjnym środowisko, agent i moment decyzyjny spełniły warunki uczenia i sesja przebiegła pomyślnie. W takiej sytuacji, zwrócone wartości będące błędami sieci neuronowych można wyświetlić wykorzystując funkcję formatującą oraz zanotować je w obiekcie klasy *Statistics*.

```
59  runner.dispose()
60  env.dispose()
61  stats.dump()
62  redaldo.model.saveWeights()
```

Po opuszczeniu pętli kontrolowanej obiektu *Runner* należy przeprowadzić utylizację obiektów ich metodami *dispose*, zapisać niedomiar zgromadzonych statystyk oraz wagi

sieci neuronowych, by nie utracić wypracowanego postępu. Tak przygotowany skrypt uruchomieniowy można następnie uruchomić za pomocą przykładowej komendy:

```
> py hax_ppo_vs_random.py −ra '.\model\actor−1000.hdf5' −rc '.\model\critic
    −1000.hdf5' −ls 1000
```

Oznacza to, że program ma wczytać z pliku istniejące już wagi sieci agenta wyuczonego w drużynie czerwonej oraz informujemy program o dotychczasowym przejściu 1000 sesji uczenia.

Na zakończenie przybliżmy jeszcze metodę *tryToLearn* klasy i przekonajmy się jakie warunki muszą spełnić obiekty, by sesja uczenia mogła być przeprowadzona.

```
1  def tryToLearn(self, experience: Memory.Experience, environment: Environment):
2      if experience.done or environment.isOld():
3          environment.refresh()
4          if self.canForceLearn():
5              environment.onLearn()
6              actorLoss, criticLoss = self.learn()
7              return actorLoss, criticLoss
8          else:
9              self.memory.refresh()
10             return None, None
11     elif self.canLearn():
12         environment.onLearn()
13         actorLoss, criticLoss = self.learn()
14         return actorLoss, criticLoss
15     return None, None
```

Jak sama nazwa metody wskazuje, agent podczas jej wywołania podejmie próbę przeprowadzenia sesji uczenia na samym sobie. Sesja zostanie przeprowadzona bez koniecznych podwarunków w sytuacji, gdy agent będzie miał już w swojej pamięci odpowiednią ilość momentów decyzyjnych (warunek z linii 11). W przypadku gdy warunkiem wprowadzenia w sesję uczenia jest przestarzenie się środowiska lub ukończenie środowiska sygnalizowane jest terminalnym momentem decyzyjnym (warunek z linii 2), sprawdzany jest dodatkowy podwarunek agenta *self.canForceLearn* (w linii 4). Warunek ten zapewnia, że agent posiada w swojej pamięci minimalną do utworzenia pojedynczej porcji ilość momentów decyzyjnych. W przeciwnym przypadku uzbierane do tej pory informacje są usuwane z pamięci i sesja uczenia nie dochodzi do skutku.

4 Eksploracja statystyk

4.1 Proces powstawania części praktycznej

Tóż przed przejściem do krótkiego opisu narzędzi do analizy statystyk omówione zostaną wszelkie aspekty procesu który doprowadził do ostatecznych wyników wyprodukowanych w trakcie pracy.

Pierwsze próby uczenia sieci neuronowej podejmowane były w pełni w oparciu o środowisko ekstraktujące obraz z gry. Uczenie odbywało się w sposób bardzo powolny. Otrzymywane wtedy wyniki były na niskim poziomie. Przyczyną była niedopracowana funkcja nagrody, brak jakiejkolwiek normalizacji oraz błędy w kodzie obsługującym środowisko. Na samym początku wybór algorytmu również nie był kwestią oczywistą. Ostatecznym wyborem okazał się algorytm PPO, jednakże na drodze pojmowania postawionego sobie problemu oraz charakterystyk algorytmów operujących na ciągłej przestrzeni stanów przewinęły się takie algorytmy jak DRQN (Deep Recurrent Q-Networks) w oparciu o warstwy GRU oraz LSTM, algorytm A2C (Advantage Actor-Critic) oraz SAC (Soft Actor-Critic). Po wyborze algorytmu i uporaniu się z większością błędów, najlepsza na tamten czas sieć neuronowa grająca po stronie atakującej posiadała nadwyraz dużą strukturę trzech gęstych warstw ukrytych z funkcją aktywacji ReLU po 512 neuronów każda. Nauczanie sieci zamykało się w nieco ponad 8.000 sesji uczenia, które przeprowadzone zostały niemalże dzień w dzień na przestrzeni tygodnia. Sieć ta była w stanie zagwarantować pomyślne ukończenie środowiska w prawie każdej konfiguracji przy założeniu braku akcji ze strony przeciwnika broniącego. Fakt sporego rozmiaru sieci oraz to jakie zachowania zostały przez nią wypracowane sugerował jakoby sieć nie opanowała logiki środowiska, a była swoistym bankiem pamięci z zapamiętaną dostateczną liczbą konfiguracji. Ponowna próba wyuczenia sieci neuronowej podobych rozmiarów, ale z głębszą strukturą klepsydrową (128-64-32-64-128) wymuszająca w lekkim stopniu potrzebę zrozumienia środowiska przez sieć nie przyniosła oczekiwanych rezultatów, co może potwierdzić, że w przypadku poprzedniej sieci pierwsze skrzypce grał słabo generalizujący aspekt pamięciowy.

Po kilku miesiącach pracy nad nauczaniem przy pomocy analizy obrazu, światło dzienne ujrzał moduł *haxballgym* (symulator), który niemal natychmiastowo został doimplementowany do projektu. Wykorzystanie symulatora znacznie przyspieszyło eksperymenty związane z hiperparametryzacją i strukturami sieci neuronowych. Na potrzeby badań z wykorzystaniem symulatora zaimplementowany został również planer uczenia, którego wykorzystanie pozwalało na delikatne polepszanie wydajności. Ostatecznie jednak największy skok wydajnościowy uświadczony został po pewnym czasie, po wprowadzeniu normalizacji funkcji przewagi. Normalizacja była kamieniem milowym części praktycznej, który pozwolił przeskoczyć z zachowań czysto pamięciowych na rozumieniowe, a ponadto struktury wykorzystywanych sieci neuronowych znacząco zmalały (dwie warstwy ukryte po 64 neurony). Mniej więcej w takiej postaci struktury sieci oraz działanie algorytmu pozostało nie zmienione do samego końca prowadzonych na potrzeby tej pracy badań.

Dodatkowym zagadnieniem (które początkowo miało zostać przybliżone pod koniec pracy) były polityki kontradyktoryjne oraz próba ich produkcji w przedstawionym środowisku. Polityka kontradyktoryjna to zestaw zachowań agenta grającego przeciw innemu, wyuczonemu agentowi, które na pierwszy rzut oka nie wydają się zachowaniami logicznymi mogącymi doprowadzić w jakikolwiek sposób do pomyślnego ukończenia środowiska, ale jednak to robią. Inspiracją do omówienia tego tematu była praca opisująca oraz potwierdzająca istnienie takich polityk oraz książka omawiająca problem dopasowania [31, 32]. Twierdzeniem postawionym w tamtej pracy jest, że polityki te są łatwiejsze do uwypuklenia w wysokowymiarowych środowiskach, a trudniejsze w tych o niższej wymiarowości, jednak nie zostały podane żadne konkretne wartości z tym związane. Końcówka tej pracy miała w obszerniejszy sposób podjąć temat wymiarowości środowiska, a jego możliwością produkcji polityk kontradyktoryjnych. Statystyki analizowane w wyżej przytoczonej pracy wykazywały, że agent posługujący się polityką kontradyktoryjną był w stanie osiągnąć lepszą wydajność wykonując losowe na pierwszy rzut oka akcje, a tak naprawdę wykorzystując niedociągnięcia wypracowane przez sieć przeciwną (która uczona była przeciw agentowi o polityce losowej). Polityki kontradyktoryjne produkowane były w sposób szybszy (w kontekście ilości sesji uczenia) w porównaniu do polityk agentów charakteryzujacych się logicznymi zachowaniami wyprodukowanych podczas uczenia na politykach losowych. Kolejne próby produkcji polityki kontradyktoryjnej podejmowane na potrzeby tej pracy nie przyniosły oczekiwanych efektów. Nie oznacza to jednak, że produkcja takich polityk w środowisku HaxBall nie jest możliwa. Przesłanki wskazujące na ich istnienie uwidocznione zostały na bardzo wczesnym etapie pracy. Wracając myślami do pierwszej sieci neuronowej wyprodukowanej przy pomocy metody ekstrakcji obrazu wspominanej na początku tego podrozdziału, jednym z przeprowadzonych już na tamtym etapie badań była próba wyuczenia agenta kontradyktoryjnego. Agent ten we wczesnym etapie uczenia wychwycił, że sieć neuronowa opierająca się na aspekcie pamięciowym miewa problemy z rozpoczęciem rozgrywki (wykopaniem piłki ze środka planszy) podczas, gdy przeciwnik znajduje się w dolnym lewym rogu swojej połowy planszy. Taka konfiguracja środowiska sprawiała, że przy próbie wykopania piłki przez agenta atakującego prawdopodobieństwo wykonania akcji ruchu w dół zwiększała się do około 14%, co sprawiało, że agent ten nie trafiał w piłkę z zamysłem jej wykopania, a nagle znajdował się za nią i chcąc wtedy wrócić do konfiguracji pozwalającej na wykopanie jej w stronę bramki przeciwnika niekiedy niechcący kopał ją wprost do swojej bramki. Agent broniący w tej sytuacji charakteryzował się kontradyktoryjnym zachowaniem, który w większej mierze składał się z ruchu w dół oraz lewo. Polityka ta, chociaż zawierała zalążki polityki kontradyktoryjnej nie jest efektem, który byłby w jakikolwiek sposób satysfakcjonujący, gdyż prawdopodobieństwo wystąpienia takiej sytuacji było niskie (najpierw agent atakujący musi nie trafić w piłkę, a następnie wykonać akcję kopnięcia podczas powracania na optymalne do wykopania stanowisko) i nie gwarantowało znacznej przewagi wydajnościowej nad przeciwnikiem.

Teorią, dlaczego na wczesnym etapie pracy ze środowiskiem HaxBall produkcja polityki kontradyktorynej wydawała się prostsza, jest to, że istnienie polityk kontradyktoryjnych oraz prostota ich produkcji jest związana ze stopniem rozwiązania problemu dopasowania w trakcie uczenia sieci "początkowej". Problem dopasowania (Alignment problem) jest dobrze znanym w sferze bezpieczeństwa uczenia maszyno-

wego problemem, który w skrócie mówi, że zadanie przedstawione algorytmowi nie zawsze jest odpowiednio doprecyzowane, a niedoprecyzowanie to wynika z ludzkiej natury (nie zawsze to co mówimy, to to co mamy na myśli) [32]. Słowo klucz "dopasowanie" odnosi się do odpowiedniego sformułowania problemu, który na wejściu ma być zrozumiały dla algorytmu, ale ostatecznie wynik uczenia musi być "odwracalny" i ma rozwiązać realny problem. Przykładem uwydatnienia problemu dopasowania może być algorytm prowadzący symulator społeczeństwa, którego zadaniem jest inteligentne zmniejszenie reproduktywności ludności manipulując środowiskiem. Ostatecznie, może on doprowadzić do wymuszonego zmniejszenia całkowitej liczby ludności. Nie jest to zachowanie które możnaby zastosować w realnym środowisku ze względu na etykę, ale z perspektywy algorytmu problem został rozwiązany, gdyż zmniejszona liczba ludności wiąże się z ich zmniejszoną reproduktywnością. Kontynuując wygłaszaną tu teorię, długotrwała praca nad hiperparametryzacją algorytmu, środowiska oraz odpowiedni wybór i wyważenie komponentów funkcji nagrody nie tylko przyczyniło się do zwiększenia wydajności wyprodukowanych polityk, ale również w pewnym stopniu do rozwiązania problemu dopasowania w środowisku HaxBall. To, że taki problem istniał oraz że na przestrzeni pracy został on zniwelowany świadczyć mogą wcześniej wypracowane polityki strony atakującej statycznego przeciwnika, których zachowania ograniczały się tylko i wyłącznie do naprzemiennego oddalania i przybliżania się do piłki lub mocnego kopania piłki w stronę bramki przeciwnika tylko po to by następnie przemieścić ją spowrotem na swoją połowę i ponowić czynność mocnego kopnięcia. Potwierdzona na podstawie dalej przedstawionych badań jest korelacja wartości nagrody za komponent kopania w stronę bramki, a ilością trafionych goli, lecz polityki te otrzymując wysokie nagrody za kopanie nie zakańczały środowiska, gdyż były one efektem nieodpowiedniego wyważenia komponentów nagrody, czyli niezbyt dokładnego przedstawienia zadania.

4.2 Przedstawienie narzędzi statystycznych i wykorzystanego sprzętu

Gromadzone podczas próbkowania środowiska i uczenia agenta statystyki posłużą do wizualizacji dotychczasowej wydajności mechanizmów decyzyjnych oraz porównywania jej między podejściami o różnej parametryzacji. W tym celu przygotowane zostały skrypty notatnikowe języka Python (IPython Notebooks) generujące wizualizacje w postaci wykresów. Pliki te nazwane zostały *hax_stat_analysis.ipynb* oraz *hax_stat_compare.ipynb* z których pierwszy przygotowany został z zamierzeniem produkcji wysokorozdzielczościowych wykresów prezentujących statystyki wyprodukowane przez pojedynczą sieć neuronową, a drugi prezentuje je w zminiaturyzowanej formie wykorzystując dwa zestawy statystyk celem ich porównania. Zawartość tych notatników nie będzie ściśle analizowana ze względu na ich roboczy aspekt oraz ciągle zachodzące w nich zmiany uwzględniające chociażby aspekty pojedynczego, obecnie analizowanego podejścia. Ogólnie rzecz ujmując, wykorzystują one między innymi pakiet *pandas* do obsługi i parsowania danych, pakiet *matplotlib* do ich wizualizacji oraz metodę statyczną *load* klasy *Statistics* do wczytywania ich z plików.

Wybrane hiperparametryzacje podejść jak i informacje związane z ich procesami uczenia zawarte zostaną w formie tabel informacyjnych, natomiast statystyki które będą przedstawiane na przestrzeni tego oraz następnego rozdziału to:

- wykres słupkowy przedstawiający pogrupowane liczby zwycięstw, porażek oraz remisów, gdzie wielkość pojedynczej grupy jest równa ilości momentów decyzyjnych po podzieleniu przez ich wybraną ilość;
- uśrednione, przy pomocy średniej kroczącej, wykresy przedstawiające błędy obu sieci neuronowych (aktora i krytyka);
- uśredniony wykres zdobywanych nagród w zależności od powodu jej uzyskania (każdy komponent nagrody jest osobno nałożony na wykres);
- uśredniony wykres przedstawiający długość epizodu.

Sprzęt wykorzystany do uczenia wszystkich przedstawionych tu sieci neuronowych to komputer z podzespołami:

- procesor: AMD Ryzen 5 3600XT,
- karta graficzna: Gigabyte GeForce RTX 3060Ti Eagle,
- pamięć RAM: 16GB.

Sprzęt pomocniczy (MacBook Pro):

- procesor: Intel Core i7,
- karta graficzna: Intel Iris Pro 1536MB,
- pamięć RAM: 16GB.

4.3 Badanie hiperparametryzacji

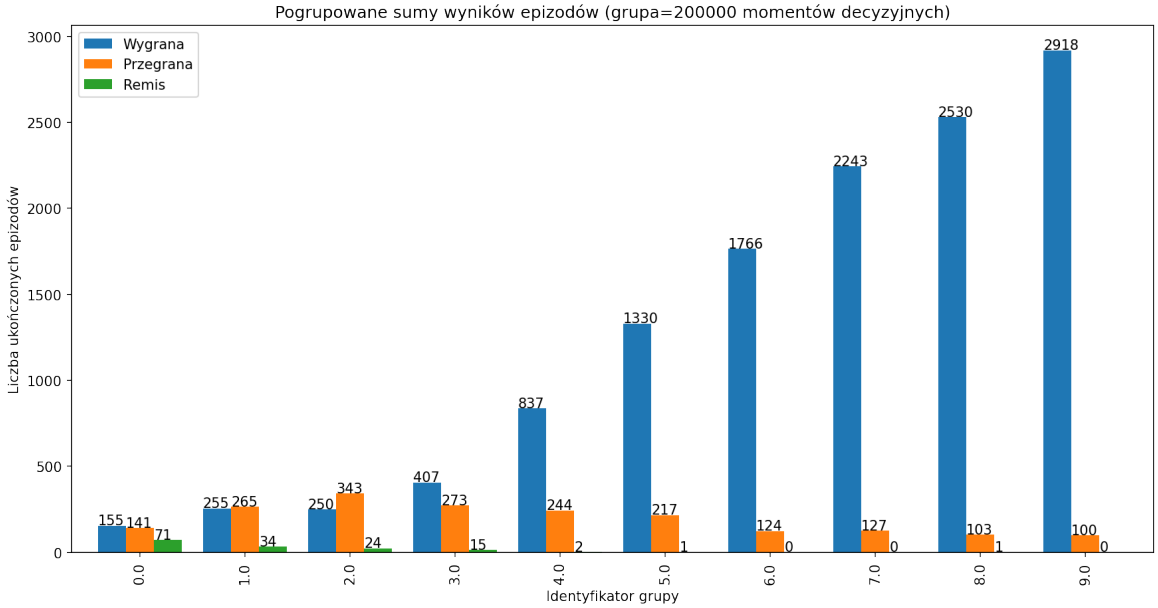
Przechodząc do sedna części praktycznej pracy na samym początku przedstawię hiperparametryzację wypracowaną na przestrzeni kilku miesięcy, która wydaje się generować najbardziej optymalne wyniki (z większości wypróbowanych), a następnie przejdę do szczątkowych badań niektórych z hiperparametrów i przyrównam je do wyników już osiągniętych. Celem zmniejszenia losowości przy porównaniach, początkowym krokiem było wyuczenie agenta atakującego w jakimkolwiek stopniu, tak by był w stanie ukańczać środowisko na umiarkowanie zadowalającym poziomie. Agent ten został wyuczony podejmować decyzję grając po stronie atakującej w grze przeciw agentowi posługującym się polityką losową (*RandomAgent*). By sprawnie rozróżniać podejścia będą one nazywane względem ich przeznaczeń lub charakterystyk, agent w tym podejściu nazwany został **napastnikiem startowym**. Tabela informacyjna napastnika startowego przedstawiona została na rysunku 5. Przyjrzyjmy się wykresom wydajności którą osiąga.

Napastnik startowy		
	Aktor	Krytyk
Model sieci	2x Dense(tanh)	2x Dense(tanh)
Współczynnik uczenia	1e-5	1e-4

Maksymalna długość epizodu w momentach decyzyjnych	1200
Rozmiar pamięci agenta w momentach decyzyjnych	120
Współczynnik dyskontowy (discount factor)	0,95
Wartość lambda (lambda value)	0,99
Wartość przycinania (clip)	0,2
Rozmiar próbki (batch size)	8
Wydajność próbki (epochs)	8

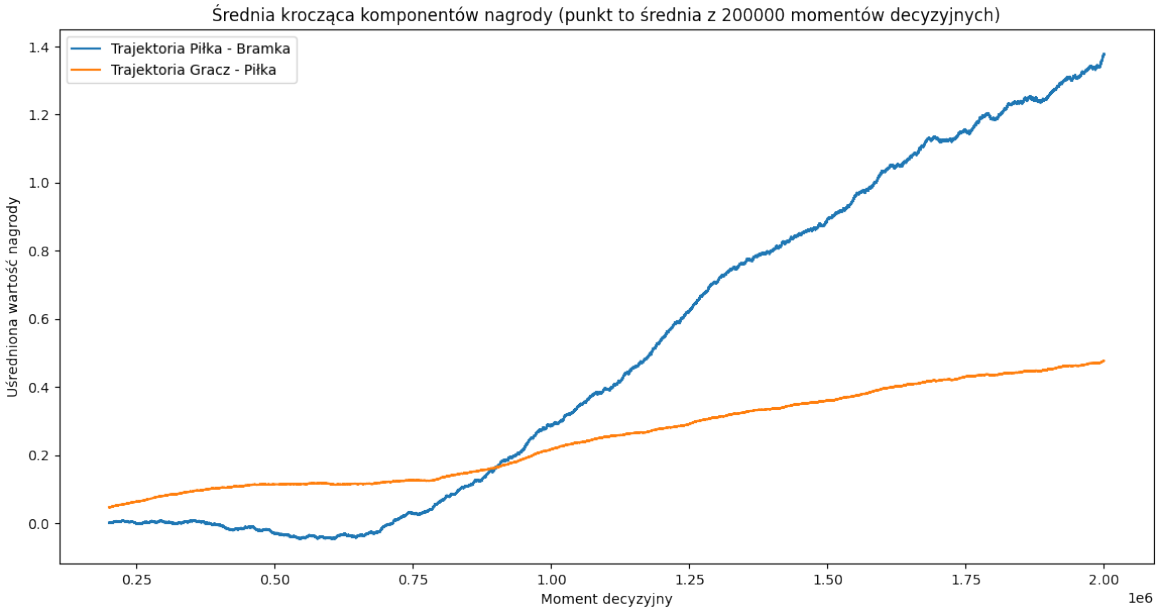
Drużyna	atakująca
Polityka przeciwnika	losowa
Ilość przepracowanych momentów decyzyjnych	2.000.000
Ilość przepracowanych epizodów	∼14.777
Ilość sesji uczenia	24.769
Czas uczenia	18h

Rysunek 5: Tabela informacyjna napastnika startowego.



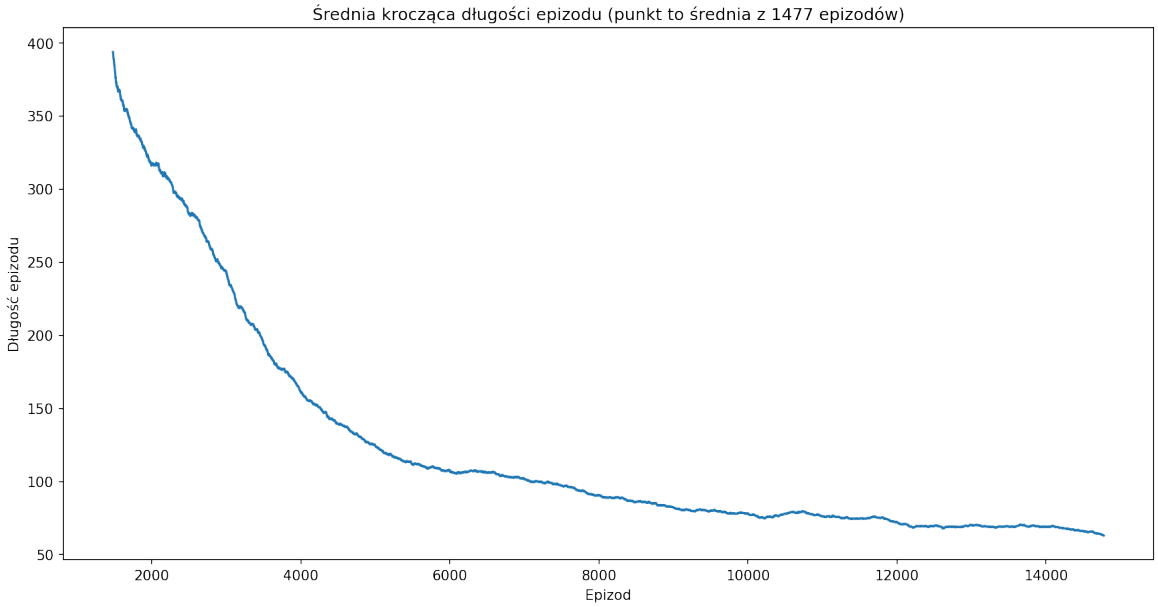
Rysunek 6: Wykres przedstawiający pogrupowane sumy wyników epizodów napastnika startowego.

Na wykresie 6 na pierwszy rzut oka widać postępującą poprawę wydajności względem czasu. Pierwszą znaczącą przewagę w epizodach zakończonych zwycięstwem możemy zaobserwować w czwartej grupie, czyli po $200.000 * 4 = 800.000$ przepracowanych momentach decyzyjnych. Przegrane epizody przez większość czasu nie odbiegają od okolic wartości 275, lecz są z czasem zauważalnie niwelowane. Remisy natomiast w pewnym momencie osiągają wartość zerową i stają się rzadkością.



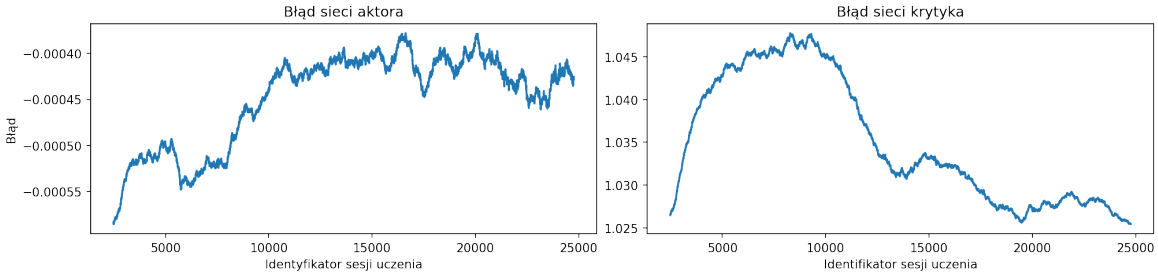
Rysunek 7: Wykres przedstawiający średnią kroczącą komponentów nagrody uzyskiwanych przez napastnika startowego.

Wykres z rysunku 7, a w szczególności krzywa reprezentująca uśrednioną wartość nagrody za kopanie piłki do bramki (trajektoria piłka - bramka) wydaje się być odzwierciedleniem krzywej epizodów wygranych z wykresu poprzedniego (rysunek 6). Teoria ta również w pewnym stopniu sprawdzi się na wykresach innych podejść. Widok obu wzrastających wartości komponentów jest wyznacznikiem ogólnej poprawy wydajności agenta.



Rysunek 8: Wykres przedstawiający średnią kroczącą długości epizodów wypracowanych przez napastnika startowego.

Rysunek 8 przedstawiający średnią długość epizodów naturalnie zmniejsza swoją wartość w miarę coraz bardziej przemyślanej eksploracji środowiska, lepszego jego zrozumienia i poznania możliwości jego kończenia. Najniższa średnia wartość jaką agent osiąga to około 63 momenty decyzyjne.



Rysunek 9: Wykresy przedstawiające błędy sieci neuronowych zwrócone przez napastnika startowego w trakcie procesu uczenia.

Ostatnimi z zestawu wykresów jest para wizualizująca zwracane błędy obu sieci neuronowych agenta. Wykresy te są mniej priorytetowe oraz nie są ustalane żadne założenia wydajnościowe jak w przypadku poprzednich wykresów (dla przykładu, zakłada się że liczba wygranych epizodów powinna się zwiększać). Wartości te są w większej mierze poglądowe ze względu na wysokie zróżnicowanie stanów środowiska. Można jednak przyjąć, że gdy wartości te odbiegają od pewnej średniej, agent w tym okresie podejmuje większe kroki w uczeniu co może być wyznacznikiem natrafienia na przełomowe odkrycie bądź wycofywanie się z poważnego błędu.

Przejdę teraz do przedstawienia serii statystyk wypracowanych przez wiele podejść delikatnie różniących się hiperparametryzacją. Każde z poniższych podejść było uczone, by pokonać napastnika startowego. Najpierw przedstawię statystyki podejścia którego hiperparametryzacja nie różni się niczym od hiperparametryzacji jego przeciwnika, a dodatkowo długoterminowo wyprodukowała najlepsze podejście do problemu i zachowania. Podejście to nazwane zostało **obrońcą startowym**.

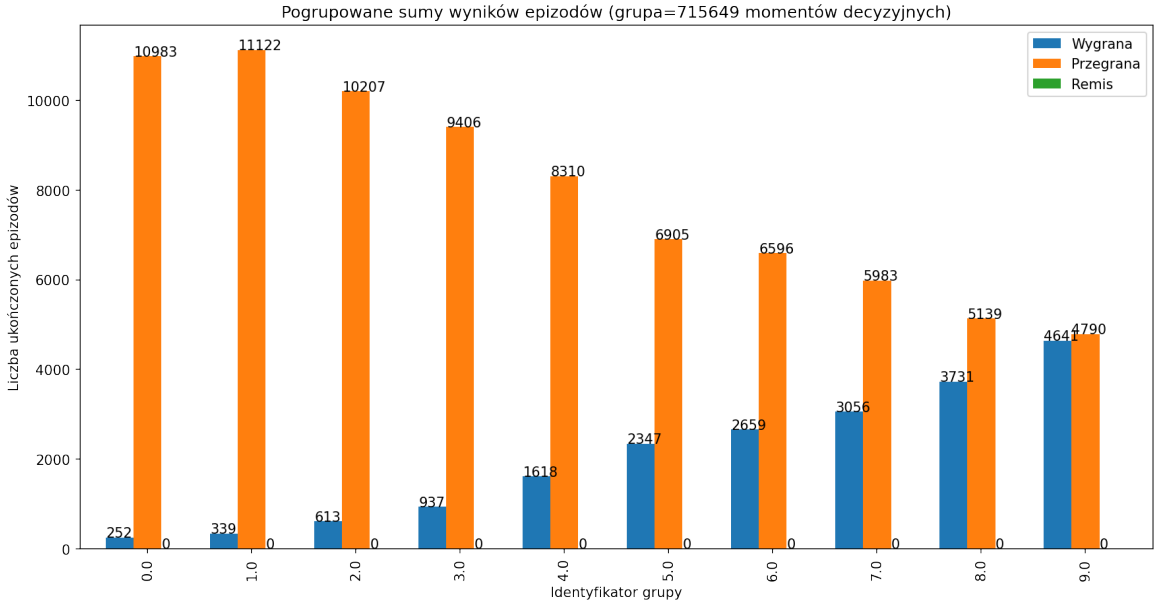
Obrońca startowy		
	Aktor	Krytyk
Model sieci	2x Dense(tanh)	2x Dense(tanh)
Współczynnik uczenia	1e-5	1e-4

Maksymalna długość epizodu w momentach decyzyjnych	1200
Rozmiar pamięci agenta w momentach decyzyjnych	120
Współczynnik dyskontowy (discount factor)	0,95
Wartość lambda (lambda value)	0,99
Wartość przycinania (clip)	0,2
Rozmiar próbki (batch size)	8
Wydajność próbki (epochs)	8

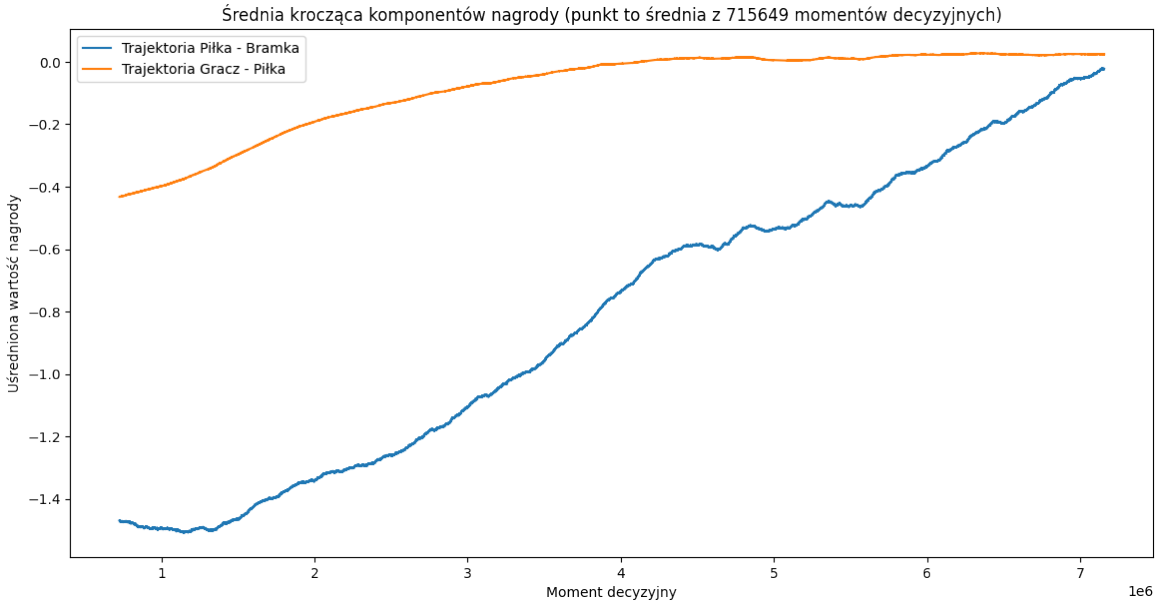
Drużyna	broniąca
Polityka przeciwnika	wyuczona (napastnik startowy)
Ilość przepracowanych momentów decyzyjnych	7.156.489
Ilość przepracowanych epizodów	∼99.635
Ilość sesji uczenia	117.874
Czas uczenia	64h

Rysunek 10: Tabela informacyjna obrońcy startowego.

Próba ta była podtrzymywana do czasu osiągnięcia proporcji 1:1 w epizodach wygranych i przegranych w najmłodszej grupie. Jak można zauważyć w powyżej przedstawionej tabeli informacyjnej, osiągnięcie tej propocji wymagało przeprowadzenia prawie czterokrotnie dłuższego procesu uczenia pod względem przepracowanych momentów decyzyjnych w porównaniu do przeciwnika startowego, co może oznaczać, że zadanie postawione przed agentem broniącym jest dużo trudniejsze, ale ten temat jeszcze zostanie poruszony w dalszej części pracy. Przyjrzyjmy się statystykom obrońcy startowego.

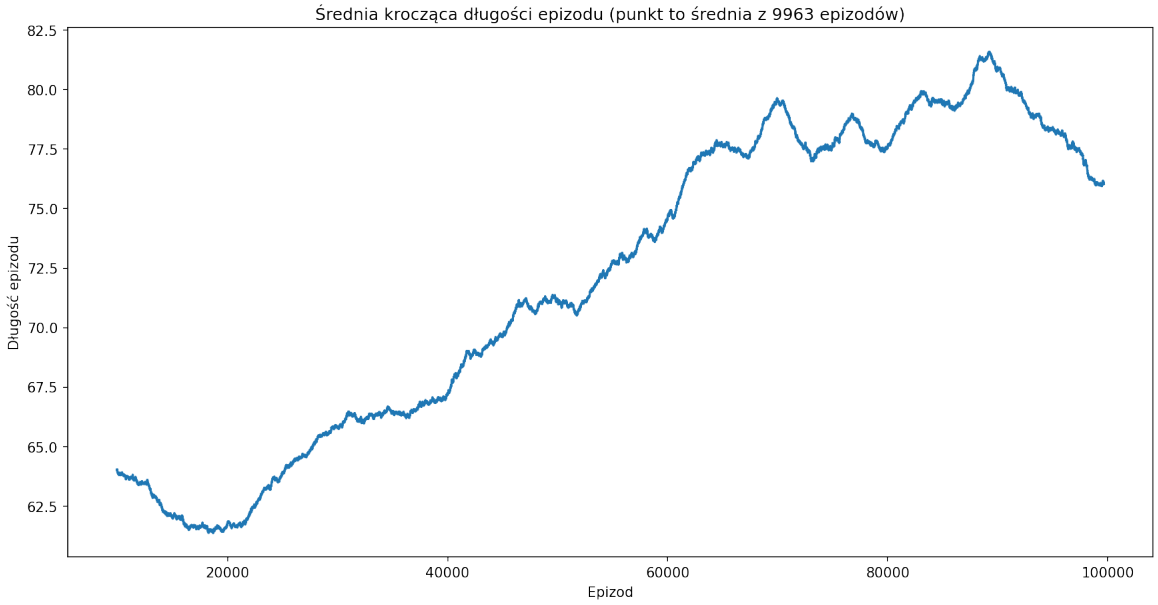


Rysunek 11: Wykres przedstawiający pogrupowane sumy wyników epizodów obrońcy startowego.



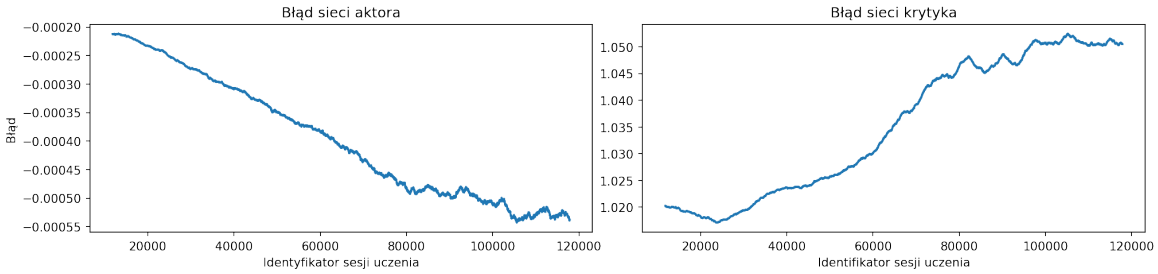
Rysunek 12: Wykres przedstawiający średnią kroczącą komponentów nagrody uzyskiwanych przez obrońcę startowego.

Przyglądając się rysunkom 11 i 12 można zauważyć pewną korelację pomiędzy przecięciem się prostej $y = 0.0$ na wykresie z wykresem trajektorii piłka-bramka, a przybliżaniem się proporcji epizodów wygranych do przegranych w grupach. W trakcie procesu uczenia agentów ostatecznych (w następnym rozdziale) podejmiemy próbę ponownego sprawdzenia teorii korelacji komponentu trajektorii piłka - bramka z proporcją wygranych epizodów do przegranych. Maksymalne średnie wartości wypracowane w tym podejściu oscylują wokół zera.



Rysunek 13: Wykres przedstawiający średnią kroczącą długości epizodów wypracowanych przez obrońcę startowego.

Z racji gry na wyuczonego przeciwnika zakłada się, że w miarę uczenia średnia długość epizodu powinna najpierw rosnąć z racji odnajdywania kolejnych sposobów na stawianie oporu przeciwnikowi.



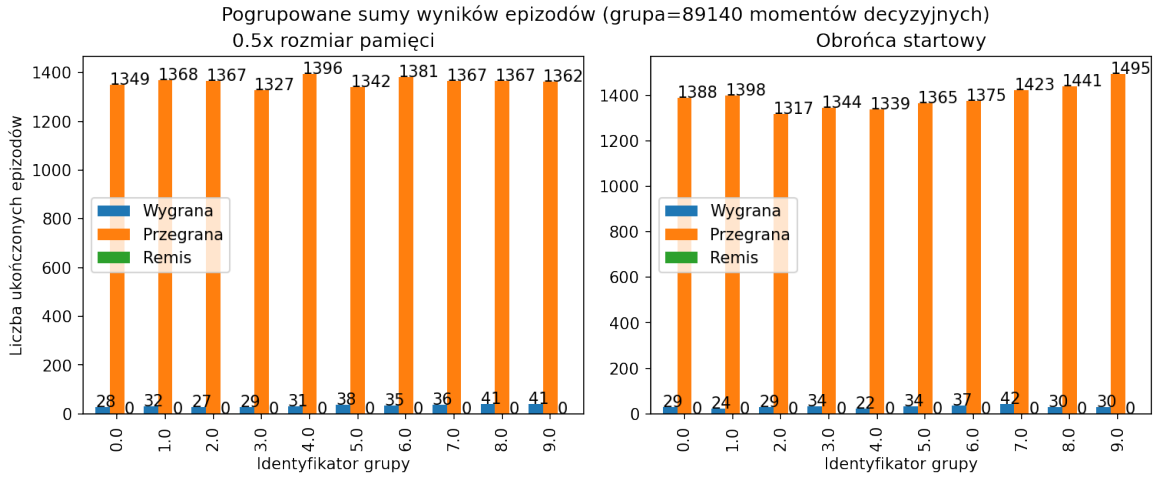
Rysunek 14: Wykresy przedstawiające błędy sieci neuronowych zwrócone przez sieci obrońcy startowego w trakcie procesu uczenia.

Przyjrzyjmy się wykresom innych hiperparametryzacji, wyciągnijmy wnioski z osiągniętych wyników oraz sprawdźmy jak działają na algorytm. Znając już w miarę sprawnie działającej hiperparametryzacji oraz z powodu przeszkody relatywnie długiego czasu uczenia sieci do zadowalającego poziomu, który również będzie brany pod uwagę, ilości próbek z których poniższe wykresy zostały wygenerowane nie są identyczne, a będą porównywane do powyższego podejścia przycinanego za każdym razem do tej samej długości. Przez małą obrazowość wydajności, wykresy błędów sieci zostaną na ten czas pominięte.

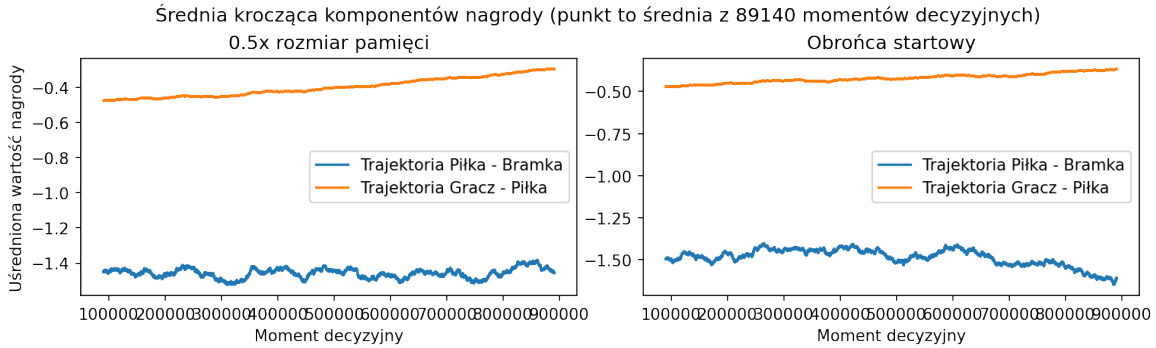
Pierwszym przykładem będzie agent przeprowadzony przez proces uczenia z taką samą parametryzacją jak obrońca startowy, lecz z połową rozmiaru pamięci. Oznacza to, że sesje uczenia były uruchamiane 2 razy częściej, lecz z racji mniejszej pamięci proces uczenia nie trwał dużo dłużej. Na przedstawionej poniżej mini-tabeli, czas odpowiadający z ostatniego rzędu informacyjnego oznacza czas w jaki obrońca startowy przeszedł przez tę samą ilość sesji uczenia.

Różnica w hiperparametryzacji względem obrońcy startowego	połowa rozmiaru pamięci
Drużyna	broniąca
Polityka przeciwnika	wyuczona (napastnik startowy)
Ilość przepracowanych momentów decyzyjnych	891.399
Ilość przepracowanych epizodów	∼14.000
Ilość sesji uczenia	21.673
Czas uczenia	8h
Czas odpowiadający obrońcy startowemu	13h

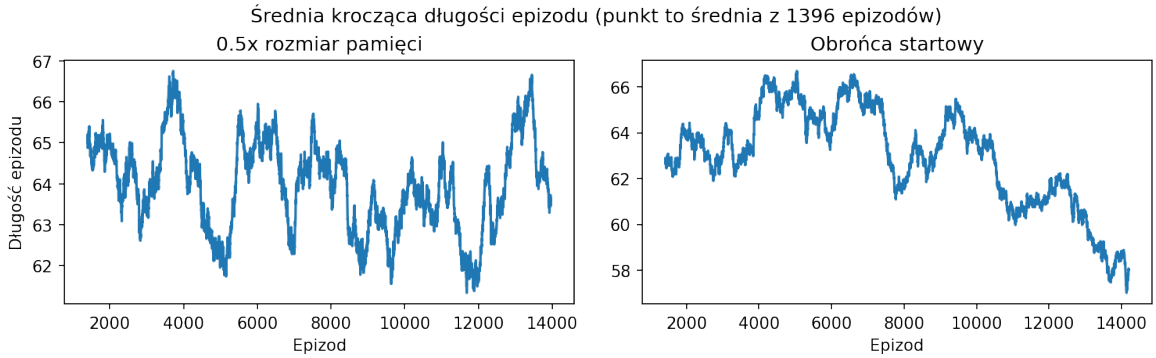
Rysunek 15: Mini-tabela informacyjna podejścia z połową rozmiaru pamięci.



Rysunek 16: Wykres przedstawiający porównanie pogrupowanych sum wyników epizodów agenta z połową rozmiaru pamięci i obrońcy startowego.



Rysunek 17: Wykres przedstawiający porównanie średnich kroczących komponentów nagrody uzyskiwanych przez agenta z połową rozmiaru pamięci i obrońcy startowego.

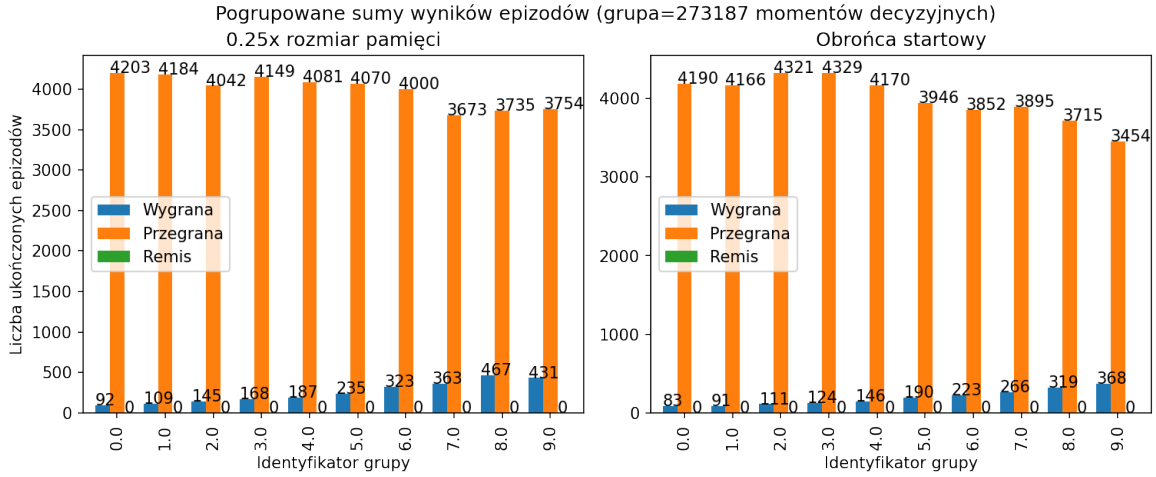


Rysunek 18: Wykres przedstawiający porównanie średnich kroczących długości epizodów wypracowanych przez agenta z połową rozmiaru pamięci i obrońcy startowego.

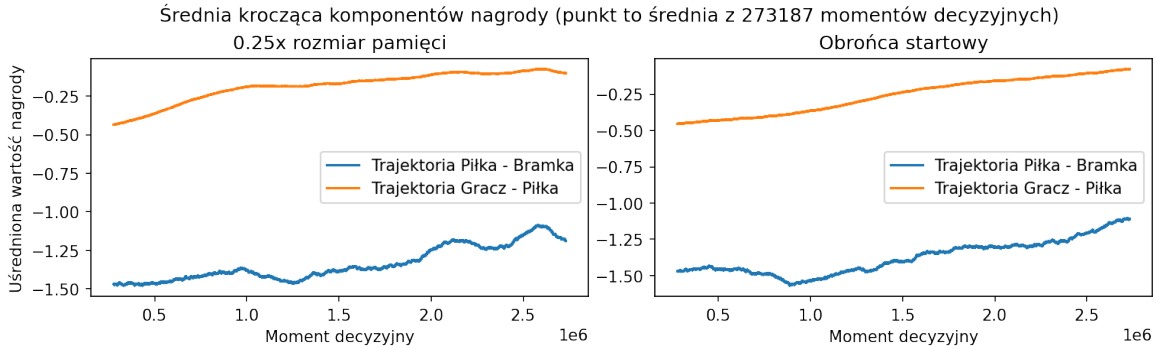
W tym przypadku dostrzegalny jest brak szczególnych różnic pomiędzy obrońcą startowym jeśli chodzi o ogólną wydajność zaobrazowaną w wykresie nagród, lecz widoczna jest dużo większa chaotyczność w średniej długości epizodu, co może mówić o większej chaotyczności w próbkowaniu środowiska. Optymistyczny jednak może wydać się czas w jaki ta wydajność została osiagnięta, gdyż jest on prawie dwa razy niższy. Celem wyciągnięcia trafniejszych wniosków na temat badanej hiperparametryzacji rozmiaru pamięci agenta, eksperyment został powtórzony, tym razem z ćwiercią rozmiaru pamięci obrońcy startowego.

Różnica w hiperparametryzacji względem obrońcy startowego	ćwierć rozmiaru pamięci
Drużyna	broniąca
Polityka przeciwnika	wyuczona (napastnik startowy)
Ilość przepracowanych momentów decyzyjnych	2.731.874
Ilość przepracowanych epizodów	∼42.410
Ilość sesji uczenia	100.367
Czas uczenia	28h
Czas odpowiadający obrońcy startowemu	54h

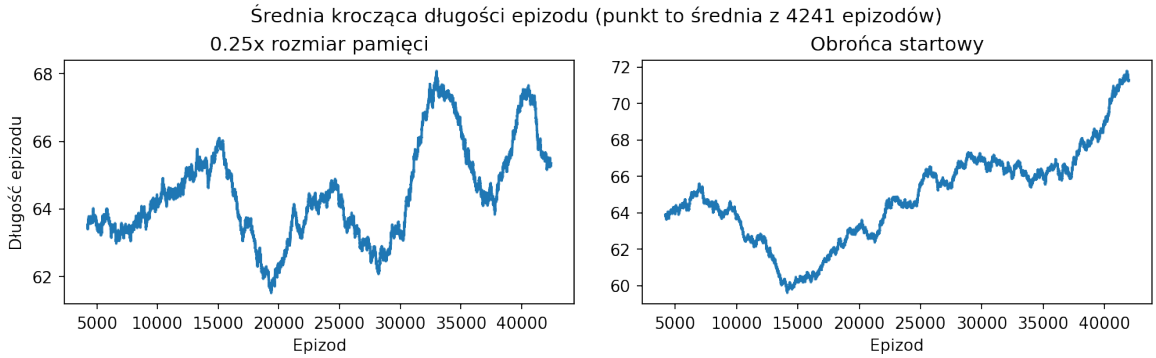
Rysunek 19: Mini-tabela informacyjna podejścia z ćwiercią rozmiaru pamięci.



Rysunek 20: Wykres przedstawiający porównanie pogrupowanych sum wyników epizodów agenta z ćwiercią rozmiaru pamięci i obrońcy startowego.



Rysunek 21: Wykres przedstawiający porównanie średnich kroczących komponentów nagrody uzyskiwanych przez agenta z ćwiercią rozmiaru pamięci i obrońcy startowego.

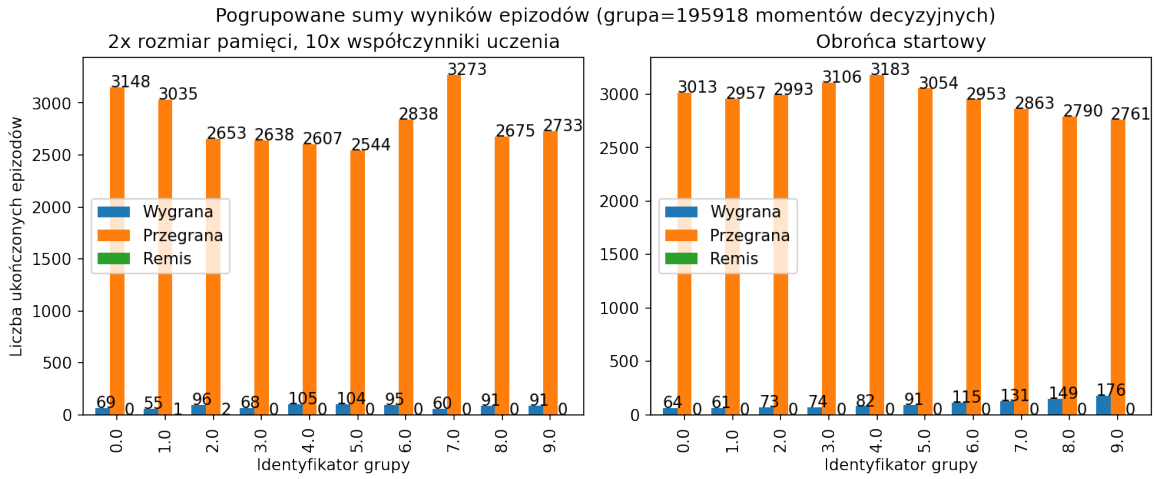


Rysunek 22: Wykres przedstawiający porównanie średnich kroczących długości epizodów wypracowanych przez agenta z ćwiercią rozmiaru pamięci i obrońcy startowego.

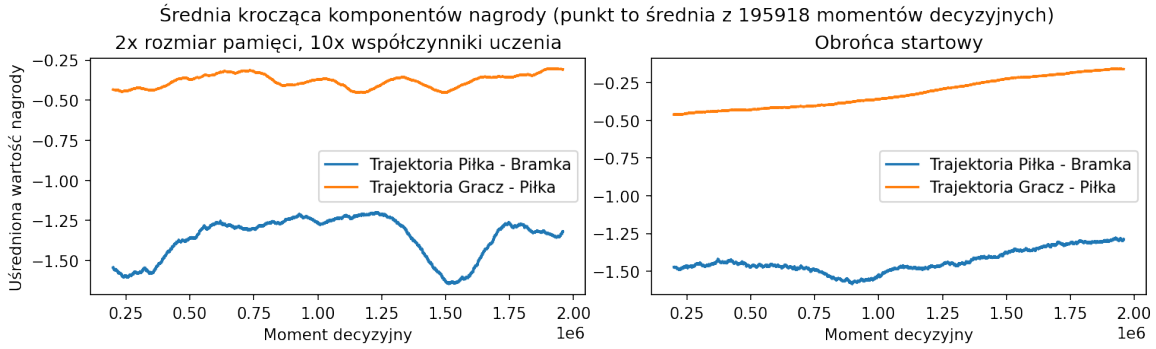
Tak jak w poprzednim przypadku, różnice w wydajności nie są wielkie. Bardziej uwidoczniona została jednak spora niestabilność agenta uwypuklona na wykresie z rysunku 22. Wnioskiem pracy nad zmniejszonym rozmiarem pamięci agenta jest zmniejszona stabilność procesu uczenia, lecz wiąże się to również ze zmniejszoną ilością czasu potrzebnego na przepracowanie tej samej ilości momentów decyzyjnych. Wyniki do pewnego czasu pozostają porównywalne. Wiedząc, że zmniejszenie rozmiaru pamięci wpływa na stabilność agenta w sposób negatywny, przyjrzyjmy się podejściu w którym rozmiar pamięci został zwiększony dwukrotnie, co powinno zwiększyć stabilność. Dodatkowo zwiększony 10-cio krotnie został współczynnik uczenia obu sieci, celem przeprowadenia równoległego eksperymentu, czy zwiększenie rozmiaru pamięci wpłynie dostatecznie mocno na stabilność krzywej uczenia, by było możliwe zwiększenie współczynnika, co może przyspieszyć uczenie.

Różnica w hiperparametryzacji względem obrońcy startowego	podwojony rozmiar pamięci 10x współczynniki uczenia
Drużyna	broniąca
Polityka przeciwnika	wyuczona (napastnik startowy)
Ilość przepracowanych momentów decyzyjnych	1.959.180
Ilość przepracowanych epizodów	∼29.000
Ilość sesji uczenia	29.502
Czas uczenia	20h
Czas odpowiadający obrońcy startowemu	16h

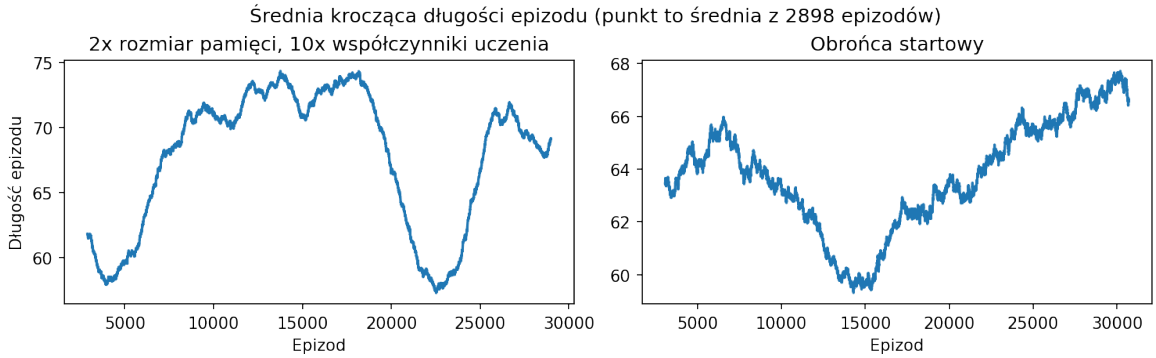
Rysunek 23: Mini-tabela informacyjna podejścia z podwójnym rozmiarem pamięci i 10 razy większym współczynnikiem uczenia obu sieci.



Rysunek 24: Wykres przedstawiający porównanie pogrupowanych sum wyników epizodów agenta z podwojonym rozmiarem pamięci i większym współczynnikiem uczenia, a obrońcą startowym.



Rysunek 25: Wykres przedstawiający porównanie średnich kroczących komponentów nagrody uzyskiwanych przez agenta z podwojonym rozmiarem pamięci i większym współczynnikiem uczenia, a obrońcą startowym.



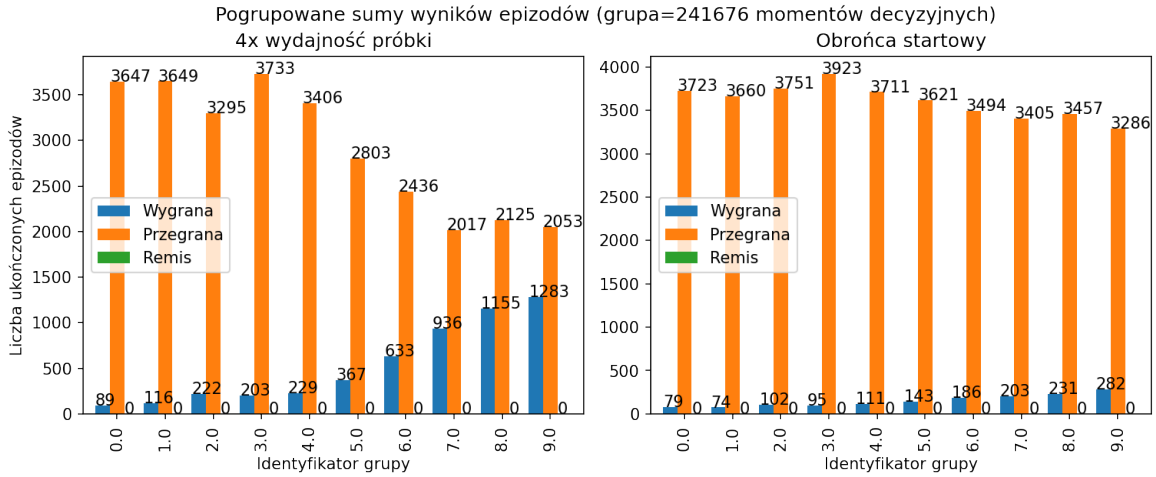
Rysunek 26: Wykres przedstawiający porównanie średnich kroczących długości epizodów wypracowanych przez agenta z podwojonym rozmiarem pamięci i większym współczynnikiem uczenia, a obrońcą startowym.

W tym przypadku widać dużo większą niestabilość spowodowaną zapewne zwiększonym współczynnikiem uczenia. Niestety, z powodu ograniczonych ram czasowych, eksperyment precyzujący wpływ zwiększonego rozmiaru pamięci (bez zwiększonego współczynnika uczenia) nie został uwzględniony. Podejrzewa się, że rozmiar pamięci ma ostatecznie mniejszy wpływ na ogólną wydajność, a przyczynia się głównie do stabilności uczenia.

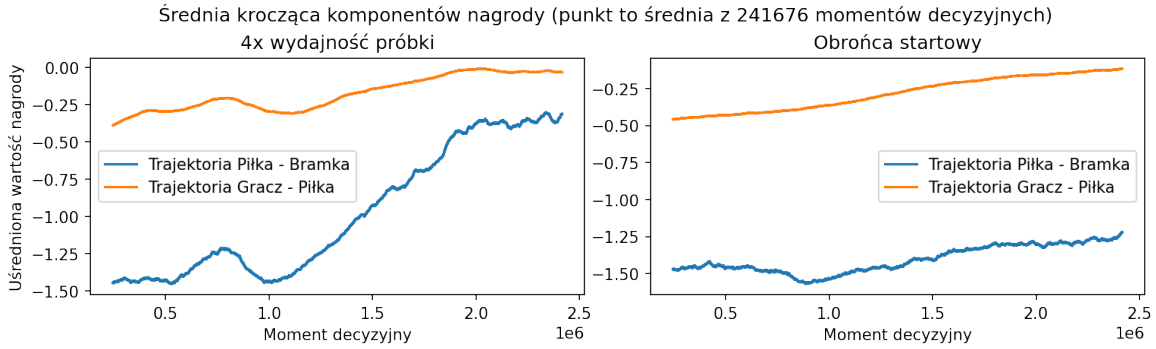
Kolejnym hiperparametrem podejrzewanym o możliwość większego wpływu na wyniki agenta jest wydajność próbki (*epochs*). Kolejny eksperyment został przeprowadzony z użyciem agenta z hiperparametryzacją obrońcy startowego z czterokrotnie większą wydajnością próbki (32 epoki).

Różnica w hiperparametryzacji względem obrońcy startowego	czterokrotnie wyższa wydajność próbki
Drużyna	broniąca
Polityka przeciwnika	wyuczona (napastnik startowy)
Ilość przepracowanych momentów decyzyjnych	2.416.761
Ilość przepracowanych epizodów	∼34.400
Ilość sesji uczenia	40.221
Czas uczenia	43h
Czas odpowiadający obrońcy startowemu	21h

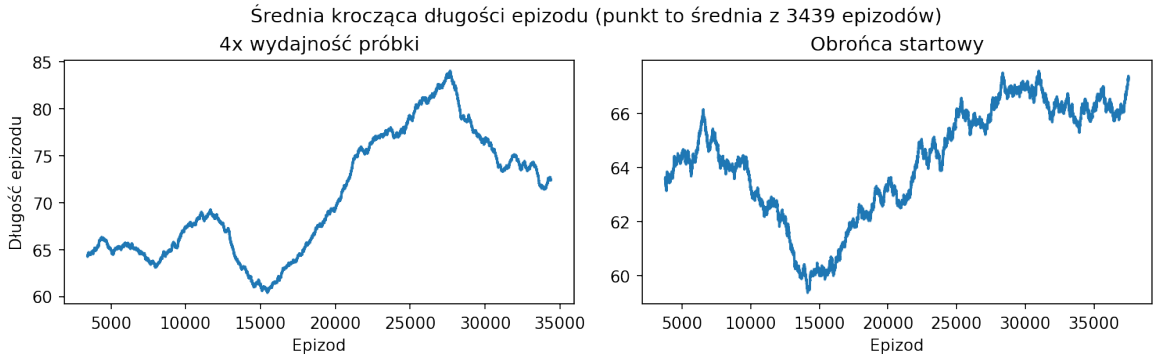
Rysunek 27: Mini-tabela informacyjna podejścia z powiększoną wydajnością próbki.



Rysunek 28: Wykres przedstawiający porównanie pogrupowanych sum wyników epizodów agenta z powiększoną wydajnością próbki i obrońcy startowego.



Rysunek 29: Wykres przedstawiający porównanie średnich kroczących komponentów nagrody uzyskiwanych przez agenta z powiększoną wydajnością próbki i obrońcy startowego.



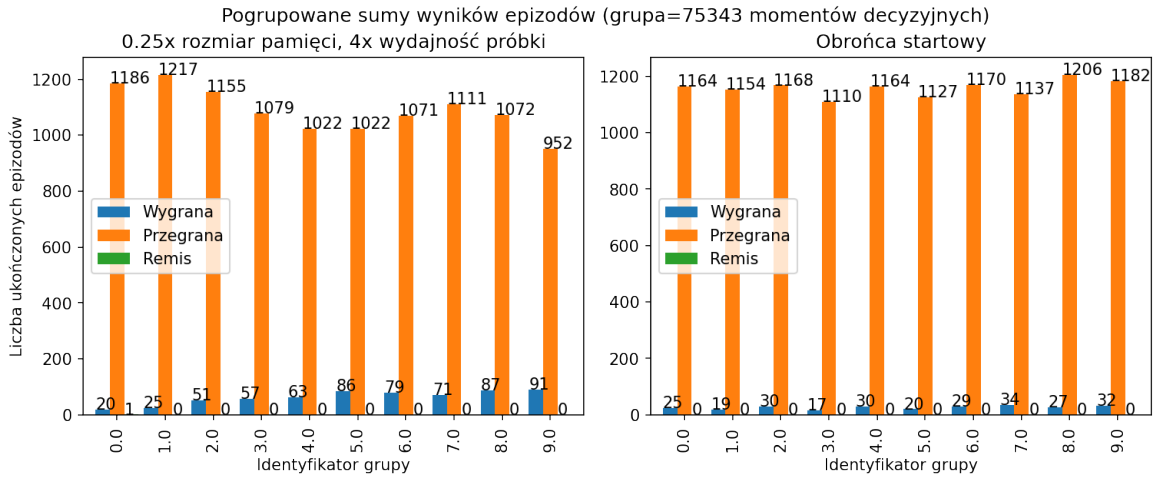
Rysunek 30: Wykres przedstawiający porównanie średnich kroczących długości epizodów wypracowanych przez agenta z powiększoną wydajnością próbki i obrońcy startowego.

Na pierwszy rzut oka widać znaczą dominację tej hiperparametryzacji. Niestety, jest to złudne wrażenie, lecz wykresy te są przykładem przedstawiającym siłę algorytmu PPO uwzględniającego hiperparametr wydajności próbki. Przeważającym argumentem przeciw tej parametryzacji jest czas wykonywania sesji uczenia, który ze względu na 4-krotnie większą liczbę iteracji próbki, jest 4-krotnie dłuższy. W tym samym czasie jesteśmy w stanie przepracować 4-razy więcej bardziej zróżnicowych próbek. W przypadku jednak, gdy próbkowanie środowiska byłoby czasochłonne, podejście to byłoby o wiele lepsze od podejścia obrońcy startowego, jednak w przypadku posiadania mechanizmu symulatora próbkującego środowisko w błyskawicznym tempie, zmniejszona wydajność próbki koniec końców zapewni lepszą generalizację oraz stabilość systemu decyzyjnego agenta.

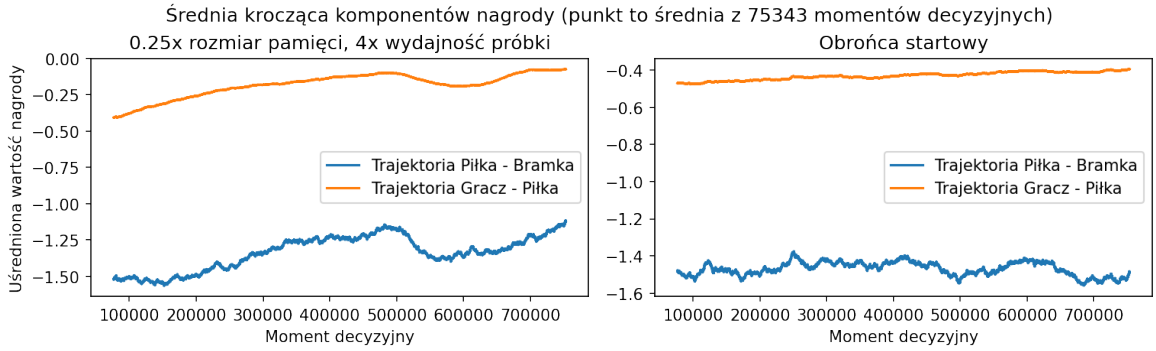
Ostatnim eksperymentalnym podejściem jest połączenie podejścia ze zwiększoną wydajnością próbki oraz czterokrotnie zmniejszonym rozmiarem pamięci.

Różnica w hiperparametryzacji względem obrońcy startowego	czterokrotnie wyższa wydajność próbki ćwierć rozmiaru pamięci
Drużyna	broniąca
Polityka przeciwnika	wyuczona (napastnik startowy)
Ilość przepracowanych momentów decyzyjnych	753.426
Ilość przepracowanych epizodów	∼11.500
Ilość sesji uczenia	34.157
Czas uczenia	13h
Czas odpowiadający obrońcy startowemu	18h

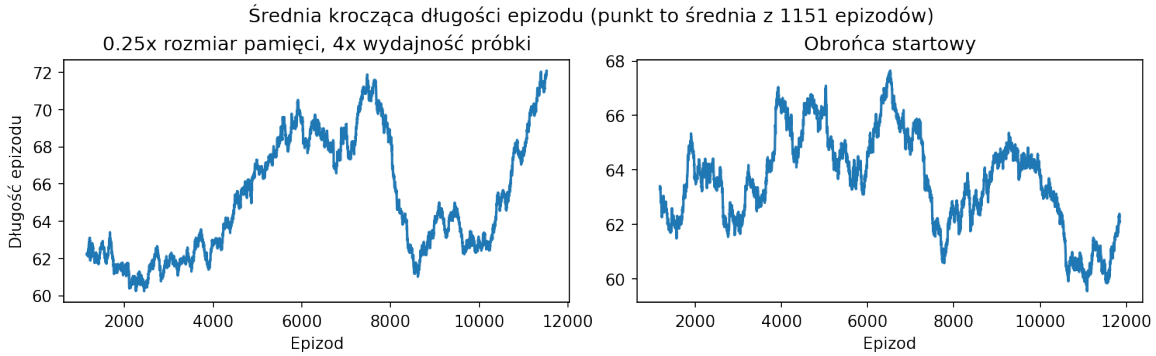
Rysunek 31: Mini-tabela informacyjna podejścia z powiększoną wydajnością próbki i ćwiercią rozmiaru pamięci.



Rysunek 32: Wykres przedstawiający porównanie pogrupowanych sum wyników epizodów agenta z czterokrotnie większą wydajnością próbki i ćwiartką rozmiaru pamięci, a obrońcą startowym.



Rysunek 33: Wykres przedstawiający porównanie średnich kroczących komponentów nagrody uzyskiwanych przez agenta z czterokrotnie większą wydajnością próbki i ćwiartką rozmiaru pamięci, a obrońcą startowym.



Rysunek 34: Wykres przedstawiający porównanie średnich kroczących długości epizodów wypracowanych przez agenta z czterokrotnie większą wydajnością próbki i ćwiartką rozmiaru pamięci, a obrońcą startowym.

Podejście to było próbą wykorzystania pozytywnych aspektów hiperparametru wydajności próbki algorytu PPO wraz ze zwiększoną częstością ich wykorzystywania poprzez zmniejszenie rozmiaru pamięci. Tak jak w przypadku podejścia z samym zwiększeniem wydajności próbki widać tutaj poprawę wydajności, lecz kosztem destabilizacji procesu uczenia. Ostatecznie czas przeprowadzonego procesu uczenia jest, prawie podobnie jak w poprzednim przypadku, około 3 razy większy.

Podsumowując, biorąc pod uwagę czynniki prędkości uczenia, stabilności i ilości próbek koniecznych do wyuczenia agenta, agenci ostateczni zaprezentowani w następnym podrozdziale wykorzystywać będą hiperparametryzację przedstawioną na samym początku tego rozdziału. Na koniec należy wtrącić słowo o pewnym marginesie błędu ostatecznie wybranej hiperparametryzacji. Być może, że przedstawione w powyższych eksperymentach hiperparametryzacje mogłyby na dłuższą metę okazać się w pewnych aspektach lepsze, gdyby pozostawić je w procesie uczenia dłużej. Jednak przez ograniczone ramy czasowe postanowiono pozostać przy hiperparametryzacji przeprowadzającej proces uczenia w sposób stabilny oraz zapewniający zadowalające wyniki.

4.4 Agenci ostateczni

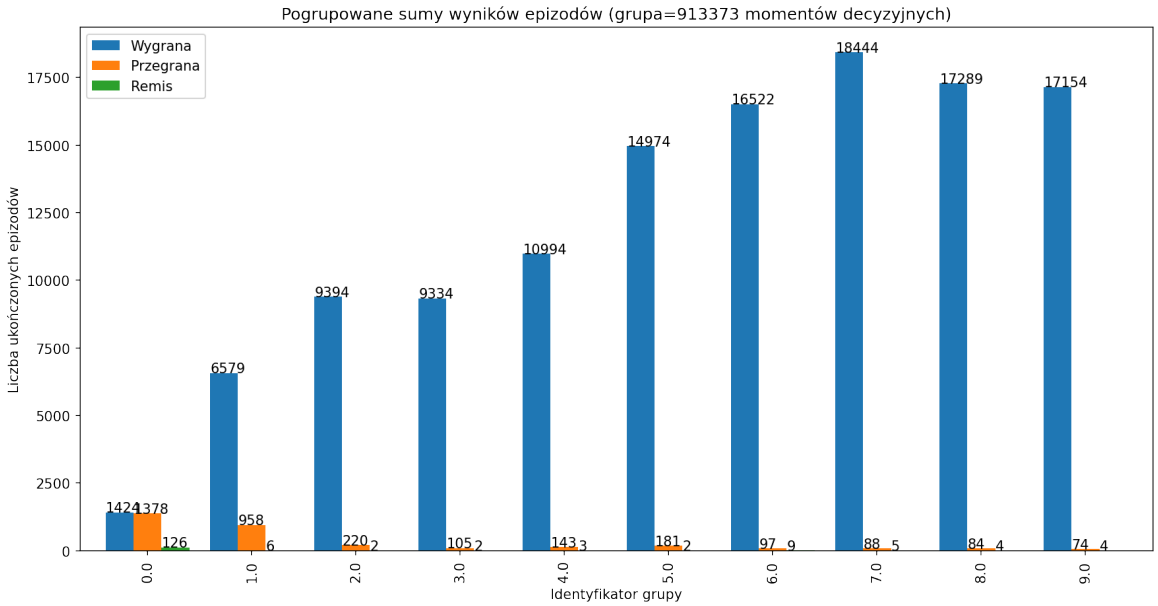
Po wybiórczym przebadaniu hiperparametryzacji oraz wyborze jednej konkretnej prezentującej się w sposób przedstawiony na rysunku 35 przejdźmy do omówienia statystyk wyprodukowanych przez agentów uczonych z zamierzeniem uzyskania najlepszej możliwej wydajności. Każdy z agentów po wyuczeniu został wnikliwie przebadany pod względem zachowań jakie reprezentował oraz w jaki sposób stopniowo jego wydajność wzrastała.

Ostateczny agent atakujący		
	Aktor	Krytyk
Model sieci	2x Dense(tanh)	2x Dense(tanh)
Współczynnik uczenia	1e-5	1e-4

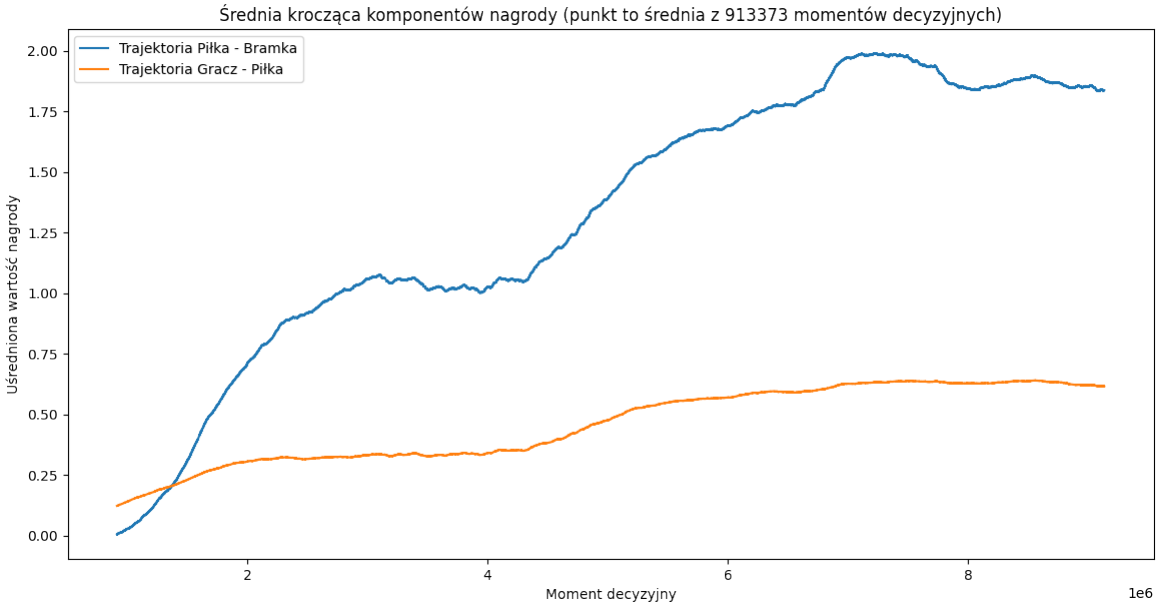
Maksymalna długość epizodu w momentach decyzyjnych	1200
Rozmiar pamięci agenta w momentach decyzyjnych	120
Współczynnik dyskontowy (discount factor)	0,95
Wartość lambda (lambda value)	0,99
Wartość przycinania (clip)	0,2
Rozmiar próbki (batch size)	8
Wydajność próbki (epochs)	8

Drużyna	atakująca
Polityka przeciwnika	losowa
Ilość przepracowanych momentów decyzyjnych	9.133.733
Ilość przepracowanych epizodów	~125.600
Ilość sesji uczenia	150.000
Czas uczenia:	76h

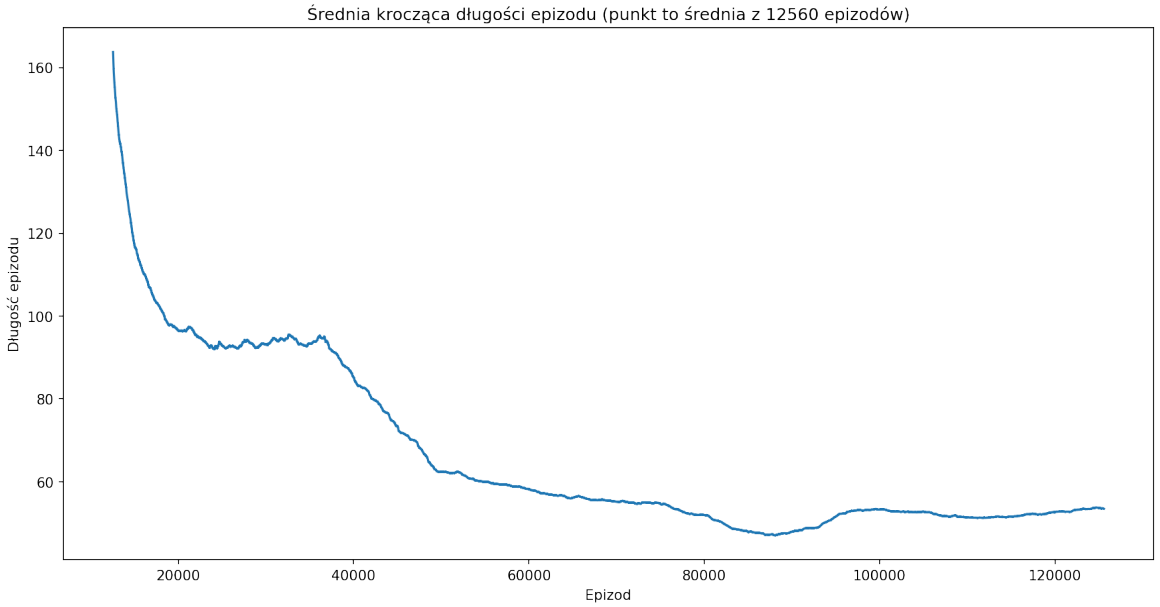
Rysunek 35: Tabela informacyjna ostatecznego agenta atakującego.



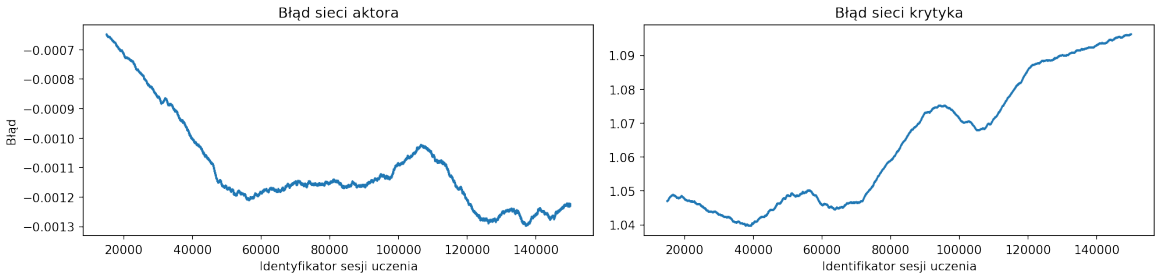
Rysunek 36: Wykres przedstawiający pogrupowane sumy wyników epizodów ostatecznego agenta atakującego.



Rysunek 37: Wykres przedstawiający średnią kroczącą komponentów nagrody uzyskiwanych przez ostatecznego agenta atakującego.



Rysunek 38: Wykres przedstawiający średnią kroczącą długości epizodów wypracowanych przez ostatecznego agenta atakującego.



Rysunek 39: Wykresy przedstawiające błędy sieci neuronowych zwrócone przez ostatecznego agenta atakującego w trakcie procesu uczenia.

Przyglądając się wykresowi z rysunku 37 widzimy jednostajny wzrost średniej aż do przepracowania około siedmiu milionów momentów decyzyjnych, dalej możemy zanotować lekki spadek. Powodem tego może być charakter algorytmu PPO, który w mniej ufny sposób podchodzi do "poprawnie" wykonanych akcji, a kroki "wstecz" są dużo większe w przypadku akcji zwracających negatywne wyniki. Unaocznieniem tego zjawiska jest również spadek strzelonych bramek na wykresie rysunku 36. Agent w pewnym okresie procesu uczenia był w stanie poświęcić możliwość strzelenia ponad tysiąca bramek w grupie, celem delikatnego zmniejszenia liczby bramek utraconych. Ciekawym aspektem jest również coś, co nie miało miejsca na etapie eksprymentowania z hiperparametryzacją. Mianowicie, po czasie agent zaczął rownież pojmować znaczenie remisów w rozgrywce i takowe zaczęły się na nowo pojawiać. Remisy osiągają wartości relatywnie zerowe, lecz poprzednie eksperymenty wykazywały, że agent jest w stanie w łatwy sposób osiągnąć wydajność i zrozumienie środowiska, by remisy wcale nie dochodziły do skutku. Przejdźmy do tabeli na rysunku 40 przedstawiającej badanie zachowań na przestrzeni całego procesu uczenia. Wnioski wyciągane były przeciwko polityce na której był uczony (losowej) co kilka tysięcy sesji uczenia.

Dokonane sesje uczenia	Opis zachowań
0	Losowe akcje z odchyleniami do 1% w zależności od konfiguracji
1.000	W większości losowe akcje, skoki prawdopodobieństw na poziomie 5% w kierunku piłki, spora wrażliwość na ruchy przeciwnika, brak zróżnicowania pomiędzy typem ruchu (sam ruch czy z kopnięciem), duże problemy z wykopaniem piłki.
3.000	Nieco mniejsza losowość, chęć podążania za piłką, trudności z przejściem na drugą stronę piłki, tendencja do ustawiania się z piłką w pionowej linii.
4.000	Skoki prawdopobieństw do 30-40% w przypadku pewnej akcji, zwiększona losowość przy piłce, mniejsze problemy z wykopaniem piłki, dalsze problemy z ominięciem piłki.
7.000	Bardziej zdecydowane akcje, prawdopodobieństwo akcji w niektórych sytuacjach spada do 0%, mniejsze problemy z ominięciem piłki, nadal duża losowość w rzadkich konfiguracjach, kierunek akcji stawiany jest ponad jej typ, małe problemy z rozpoczęciem.
9.000	Zdecydowane skoki prawdopobieństw w stronę optymalnej pozycji względem piłki, problem z dobiciem piłki do bramki.

15.000	Nadal spore nieuzasadnione różnice prawdopobieństw pomiędzy typami ruchów, brak problemów z wykopaniem piłki, tendencja wykopywania w górny róg, lekkie problemy z dobiciem piłki do bramki, pomysł na omijanie piłki poprzez odbicie od autu.
20.000	Aktywne pozycjonowanie na tym samym poziomie co piłka, płynne ruchy, nadal kierunek ruchu jest stawiany ponad jego typ, dodatkowe pomysły na sprawne omijanie piłki, brak wrażliwości na pozycję przeciwnika.
25.000	Pomyślne próby precyzyjnego celowania strzału na pustą bramkę, brak problemu z ominięciem piłki, pomysł na wprowadzanie piłki do bramki w przypadku jej utknięcia w rogu planszy, w przypadku tego zachowania nie jest wykorzystywana akcja kopnięcia, co może sugerować zrozumienie różnic pomiędzy typami akcji.
30.000	Brak problemów z poprawnym wybiciem piłki, priorytetyzowane są wybicia w górę od bramki przeciwnika, lecz zdażają się strzały centralnie na bramkę, dużo krótkich epizodów z obiciem piłki od górnego autu, bardziej precyzyjne podejście do wrażliwych sytuacji.
40.000	Nieoczekiwane problemy z rozgrywaniem piłki z dolnego prawego rogu planszy. W początkowej konfiguracji środowiska szansa wykonania akcji ruchu w prawo bez kopnięcia dobija 100%.
50.000	Pewne akcje wokół piłki sugerują dobrą znajomość poprawnego kierunku jej rozgrywania. Brak problemów z pozytywnym zakończeniem środowiska w oczywistych sytuacjach. Powolne przeobrażenie się taktyki wprowadzania piłki do bramki z rogu planszy na dośrodkowywanie (wykorzystanie kopnięcia).
60.000	Taktyka wprowadzania piłki zastąpiona taktyką dośrodkowania, mała długość epizodu.
75.000	Zerowe problemy z ominięciem piłki, małe niezdecydowanie (losowość) w większości sytuacji.
90.000	Zwiększona świadomość wykorzystania akcji kopnięcia, widocznie mniejsze zrozumienie konfiguracji planszy w której agent przebywa na lewej połowie.
100.000	Zwiększone wykorzystanie akcji na skos, precyzyjniejsze pozycjonowanie, polepszona znajomość swojej połowy planszy.
120.000	Sprawne rozgrywanie i planowanie przyszłych akcji, przewidywanie ruchu piłki. Wybijanie piłki w większości przypadków w górę planszy.
150.000	Agent zna zestaw często sprawdzających się zachowań, dobrze określa przyszłe pozycje piłki. Charakteryzuje się dużą precyzyjnością strzałów i praktycznym brakiem wrażliwości na pozycję i ruchy przeciwnika.

Rysunek 40: Tabela opisująca zachowania na przestrzeni procesu uczenia ostatecznego agenta atakującego.

Podsumowując cały rozwój zachowań, wydaje się on jak najbardziej zdrowym modelem cały czas eksplorującym i poznającym środowisko. W miarę uczenia zmieniającym nawet te dobrze wyuczone taktyki (problem z rogami) na bardziej optymalne. Zbyt pochopne określenie według agenta optymalnych zachowań (lokalne minima) i zbyt długie posługiwanie się nimi może doprowadzić do zaniedbania konfiguracji rzadkich, jak w przypadku sytuacji na lewej połowie planszy, lecz prostota środowiska w tym przypadku pozwoliła na lekko opóźnione, ale poprawne doinformowanie agenta. Przedstawiona wcześniej w pracy wymodelowana i wyważona funkcja nagrody spełnia swoje zadanie dla tego agenta na bardzo zadowalającym poziomie.

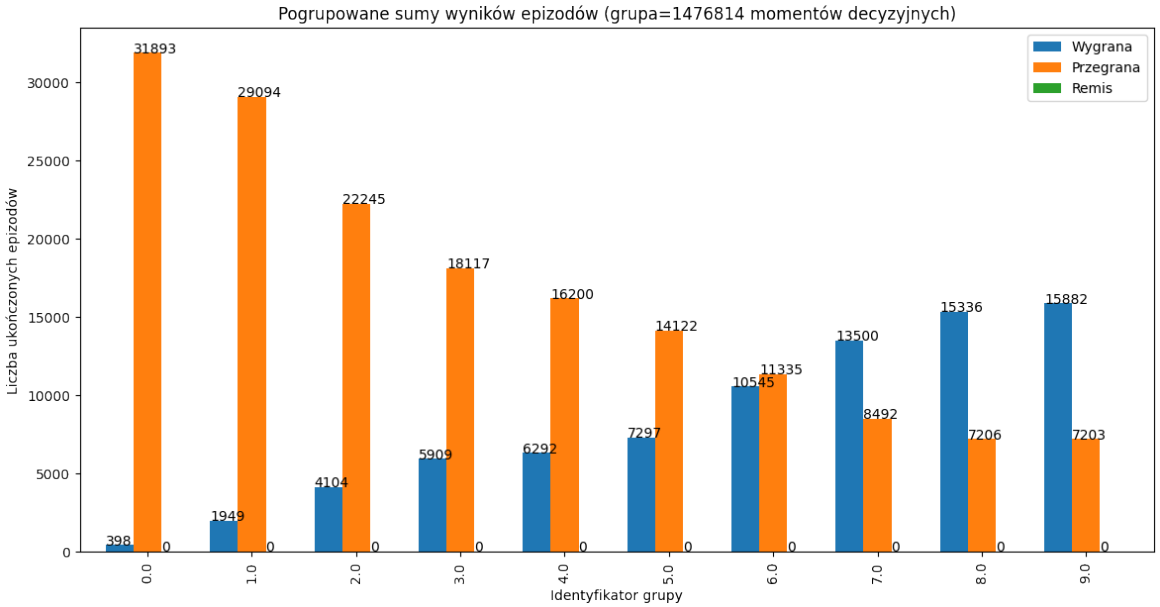
Teraz pora na zaprezentowanie wydajności agenta uczonego pokonywać przed chwilą przedstawionego ostatecznego agenta atakującego. Agent ten uczony był z dokładnie taką samą hiperparametryzacją algorytmu PPO, a nazwany został analogicznie **ostateczny agent broniący**.

Ostateczny agent broniący		
	Aktor	Krytyk
Model sieci	2x Dense(tanh)	2x Dense(tanh)
Współczynnik uczenia	1e-5	1e-4

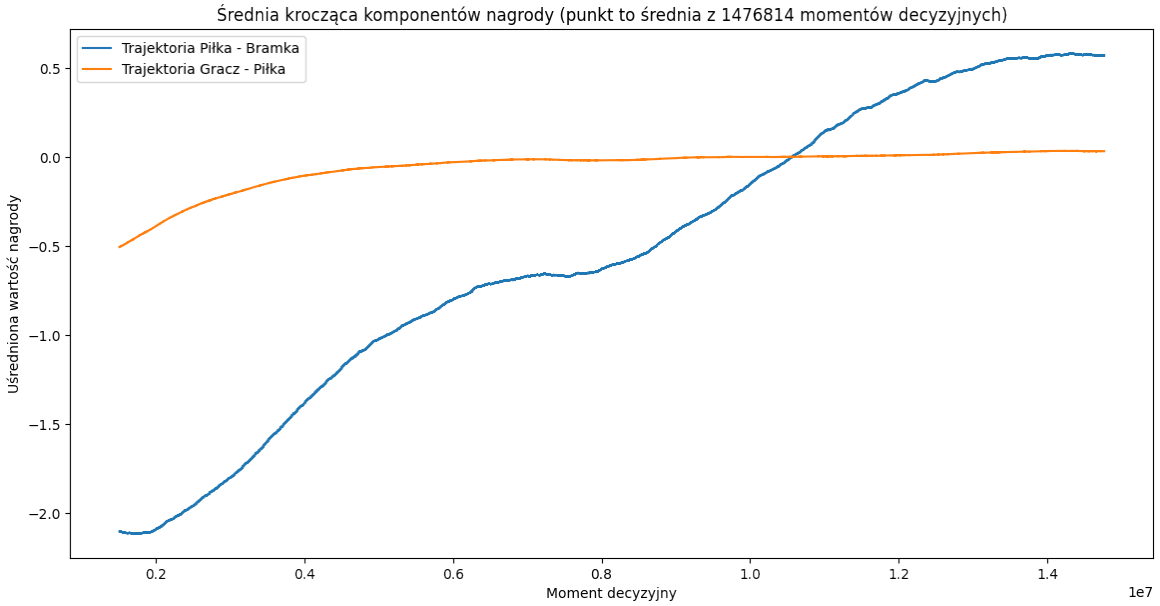
Maksymalna długość epizodu w momentach decyzyjnych	1200
Rozmiar pamięci agenta w momentach decyzyjnych	120
Współczynnik dyskontowy (discount factor)	0,95
Wartość lambda (lambda value)	0,99
Wartość przycinania (clip)	0,2
Rozmiar próbki (batch size)	8
Wydajność próbki (epochs)	8

Drużyna	broniąca
Polityka przeciwnika	wyuczona (ostateczny agent atakujący)
Ilość przepracowanych momentów decyzyjnych	14.768.139
Ilość przepracowanych epizodów	∼247.120
Ilość sesji uczenia	275.000
Czas uczenia:	158h

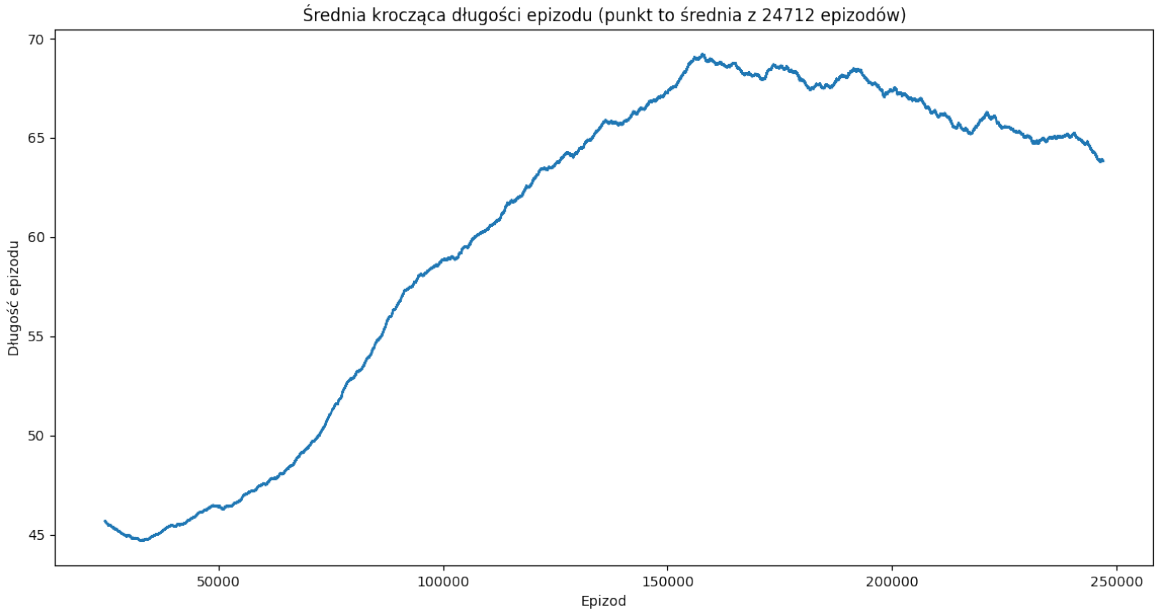
Rysunek 41: Tabela informacyjna ostatecznego agenta broniącego.



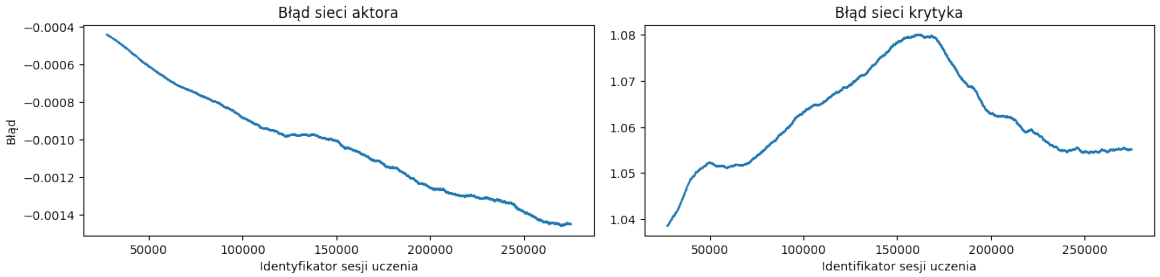
Rysunek 42: Wykres przedstawiający pogrupowane sumy wyników epizodów ostatecznego agenta broniącego.



Rysunek 43: Wykres przedstawiający średnią kroczącą komponentów nagrody uzyskiwanych przez ostatecznego agenta broniącego.



Rysunek 44: Wykres przedstawiający średnią kroczącą długości epizodów wypracowanych przez ostatecznego agenta broniącego.



Rysunek 45: Wykresy przedstawiające błędy sieci neuronowych zwrócone przez ostatecznego agenta broniącego w trakcie procesu uczenia.

Dokonane sesje uczenia	Opis zachowań
0	Losowe akcje z odchyleniami do 1% w zależności od konfiguracji
5.000	Szczątkowa wrażliwość na pozycję piłki, w większości losowe akcje, problemy z obroną początkowego strzału strony atakującej.
10.000	Zalążek taktyki obronnej (25% szansy na ruch w górę i w lewo) w odpowiedzi na tendencję do strzałów w górny róg przez agenta atakującego. Podtrzymująca się duża losowość akcji.
15.000	Przebłyski chęci podążania za piłką, spora wrażliwość na ruch przeciwnika.
20.000	Widocznie duże zainteresowanie górnym lewym rogiem swojej połowy planszy, eksplorowanie taktyki obronnej, spora losowość.
30.000	Nieudane próby wyprodukowania skłonności fizycznego odpychania przeciwnika od piłki z obawy przed utratą punktu, spora losowość.

40.000	Pierwsze próby ataku na bramkę przeciwnika po przejęciu piłki, mniejsza losowość akcji, zachowanie wyczekiwania na rozpoczęcie rozgrywki przez przeciwnika, ruchy w stronę piłki, utrzymywanie pozycji po swojej stronie planszy w niebezpiecznych sytuacjach.
50.000	Coraz większa skuteczność obrony pierwszego strzału, zwiększona świadomość ruchów przeciwnika, dla początkowej konfiguracji akcja ruchu w lewo osiąga wartość 50%.
75.000	Ataki na bramkę przeciwnika, pierwsze skuteczne techniki obrony obszaru wokół bramki, płynniejsze ruchy.
100.000	Spore prawdopodobieństwo obrony pierwszego strzału, agent nadal próbuje spychać przeciwnika, nieoczekiwane zachowanie śledzenia przeciwnika (ustawianie się za nim na jednej linii z piłką), pewne strzały na pustą bramkę.
125.000	O wiele płynniejsze i pewniejsze ruchy, pierwsze zdecydowane rozgrywania piłki, lekkie problemy z dobiciem piłki do bramki, pozycjonowanie się bliżej piłki od przeciwnika, świadome wybicia piłki na stronę przeciwnika, wzmożona ilość trafień do swojej bramki powiązana z poprzednim, zachowanie śledzenia przeciwnika przeobrażona w zachowanie czekania aż przeciwnik go prześcignie.
150.000	Świadomość o pozycji piłki, dobra obrona i wykopywanie, coraz mniejsza liczba strzałów na swoją bramkę, świadome wykorzystywanie autów, pozycjonowanie po prawidłowej stronie piłki, zachowanie śledzenia oraz wyczekiwania przestało zachodzić.
200.000	Pomyślne próby spowalniania i spychania przeciwnika z jego toru ruchu w stronę piłki, wykorzystywanie okazji na łatwego gola, kierunek akcji jest stawiany ponad jej typ, agent na początku rozgrywki ustawia się w gotowości do obrony na strzał w górę oraz prosto na bramkę, branie pod uwagę ruchu piłki, częste ataki na bramkę przeciwnika, bardzo dobra obrona gdy piłka zbliżaja się w stronę bramki.
225.000	Zwiększona precyzja strzałów, planowanie akcji, małe problemy z dobiciem piłki do bramki, zrozumienie różnic między typami akcji, szybsze przemieszczanie się po planszy w porównaniu do przeciwnika, lekkie problemy z rogami planszy po stronie przeciwnika, zaniedbana obrona strzałów prosto na bramkę (częste utraty goli z tego powodu).
250.000	Świadomość stylu gry przeciwnika, zagrywki kontrujące, częste akcje ofensywne, wykorzystanie akcji NO (brak akcji) nawet do 30% przy niektórych konfiguracjach.
275.000	Zwiększone prawdopodobieństwo wykorzystania akcji NO przy obronie bramki, doskonała obrona przed akcjami ofensywnymi, dobre ataki, niemal idealne zrozumienie typów akcji.

Rysunek 46: Tabela opisująca zachowania na przestrzeni procesu uczenia ostatecznego agenta broniącego.

Podsumowując po kolei wszystkie powyżej przedstawione informacje dotyczące ostatecznego agenta broniącego, pierwszym co rzuca się w oczy na tabeli informacyjnej z rysunku 41 jest ilość sesji uczenia i co za tym idzie, czas potrzebny na wyuczenie tego agenta. Ilość sesji jest prawie dwukrotnie większa. Ilość przepracowanych momentów decyzyjnych nie odzwierciedla długości w taki sam sposób ze względu na małą długość epizodu w pierwszej połowie procesu uczenia. Przechodząc do wykresu słupkowego z wynikami epizodów oraz wykresu komponentów nagrody - gołym okiem można zauważyć wcześniej wspominaną w tej pracy, niejako oczywistą korelację między wykresem komponentu nagrody za trajektorię "piłka - bramka", a sumami wyników epizodów w kolejnych grupach. Proporcje sum wyników w najmłodszej grupie (identyfikator 9.0) dają pretekst do stwierdzenia, że wydajność agenta broniącego na tym etapie jest ponad dwukrotnie większa od agenta atakującego (proporcja sum 1:2.2). Na wykresie z rysunku 42 można zaobserować również brak epizodów zakończonych remisem. Wykres średniej kroczącej długości epizodów w sposób przewidywalny najpierw zwiększa wartość do poniżej 70 momentów decyzyjnych na epizod w nieco ponad połowie procesu uczenia, a następnie konsekwentnie i powoli spada. Podsumowując kompleksowo tabelę wychwyconych zachowań agenta broniącego z rysunku 46, to sam początek procesu uczenia w porównaniu do procesu ostatecznego agenta atakującego wydaje się być nierozróżnialny. W pewnym momencie jednak dużym problemem staje się zwiększony próg wymaganej precyzyjności akcji. Agent broniący, na podstawie nic nie znaczących dla niego informacji, musi od samego początku zrozumieć z nich dużo więcej. Proces uczenia napastnika, a dokładniej mała sensowność (duża losowość) akcji przeciwnika, pozwoliła mu na szybkie odrzucenie mniej istotnych dla niego na tym etapie informacji. Wskazuje na to porównywalnie szybsze wyuczenie niskiej wrażliwość prawdopodobieństw na ruchy przeciwnika. Fakt ten oraz to, że agent broniący przez tak długi okres nie ma pomysłu na stawienie oporu przeciwnikowi może sugerować, że informacja o pozycji przeciwnika, przynajmniej przez pewien czas, nie jest dla dobrego wyuczenia agenta tak kluczowa jak można by się spodziewać. Wszelkie zachowania typowe, takie jak podążanie za piłką, wykopowanie jej oraz płynne, zdecydowane ruchy, wyuczone zostają bardzo późno. Innym powodem opóźnionego skoku rozwojowego może być wprowadzenie agenta do dużo bardziej chaotycznego środowiska w porównaniu do początków ostatecznego agenta atakującego. Skoki w funkcji nagrody są dużo wyższe oraz większość sytuacji kluczowych (gole/mocne kopnięcia) nie jest wygenerowanych przez akcje agenta broniącego. Nawet ruch piłki zachodzący w środowisku na początku procesu nie jest w żaden sposób spowodowany akcjami dopiero uczącej się strony. Wtem kolejność nauki zachowań wydaje się być w nieco zmienionej od optymalnej kolejności, co skutkuje częstym "wycofywaniem się" algorytmu z podejmowanych zachowań, a to zaś jest efektem wynikającym z problemów uzyskania przez agenta jakiejkolwiek dostatecznie "pozytywnej" nagrody na początku uczenia.

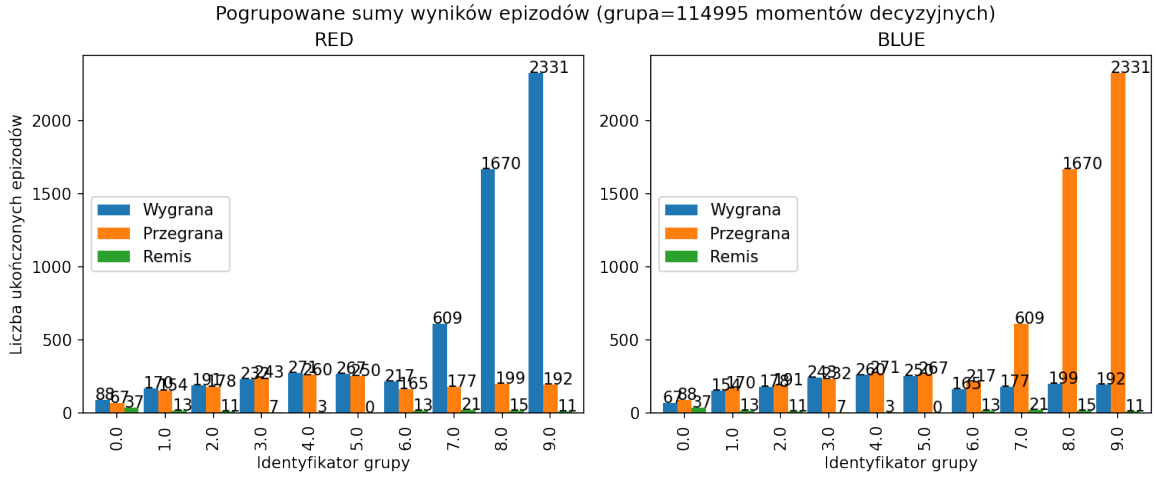
Wspomniana wcześniej teoria podwyższonej chaotyczności początkowej środowiska i jej wpływ na opóźniony skok wydajnościowy agenta broniącego została jednak podważona. Dwukrotnie wykonany eksperyment zwrócił niemalże identyczne wyniki w postaci próby wyuczenia obu sieci jednocześnie od zera. Wyniki te podkreślają więc wniosek, że zadanie postawione przed agentem broniącym jest po prostu trudniejsze. Okrojone informacje o jednym z podejść eksperymentu prezentują się następująco.

Jednoczesna nauka obu agentów		
	Aktor	Krytyk
Model sieci	2x Dense(tanh)	2x Dense(tanh)
Współczynnik uczenia	1e-5	1e-4

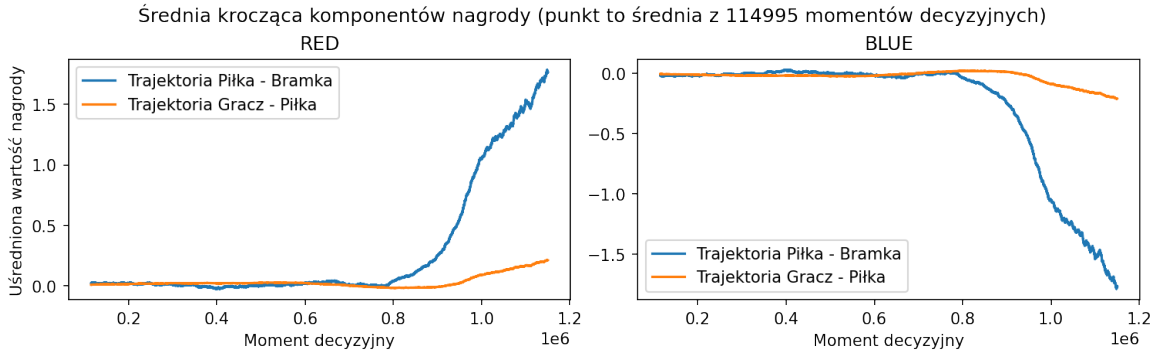
Maksymalna długość epizodu w momentach decyzyjnych	1200
Rozmiar pamięci agenta w momentach decyzyjnych	120
Współczynnik dyskontowy (discount factor)	0,95
Wartość lambda (lambda value)	0,99
Wartość przycinania (clip)	0,2
Rozmiar próbki (batch size)	8
Wydajność próbki (epochs)	8

Ilość przepracowanych momentów decyzyjnych	1.149.947
Ilość przepracowanych epizodów	∼8.060
Ilość sesji uczenia	15.001
Czas uczenia:	17h

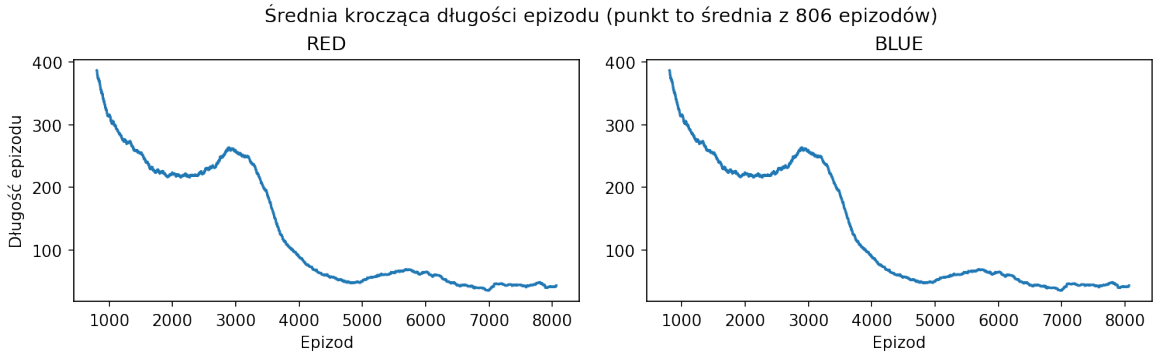
Rysunek 47: Tabela informacyjna podejścia jednoczesnej nauki obu agentów.



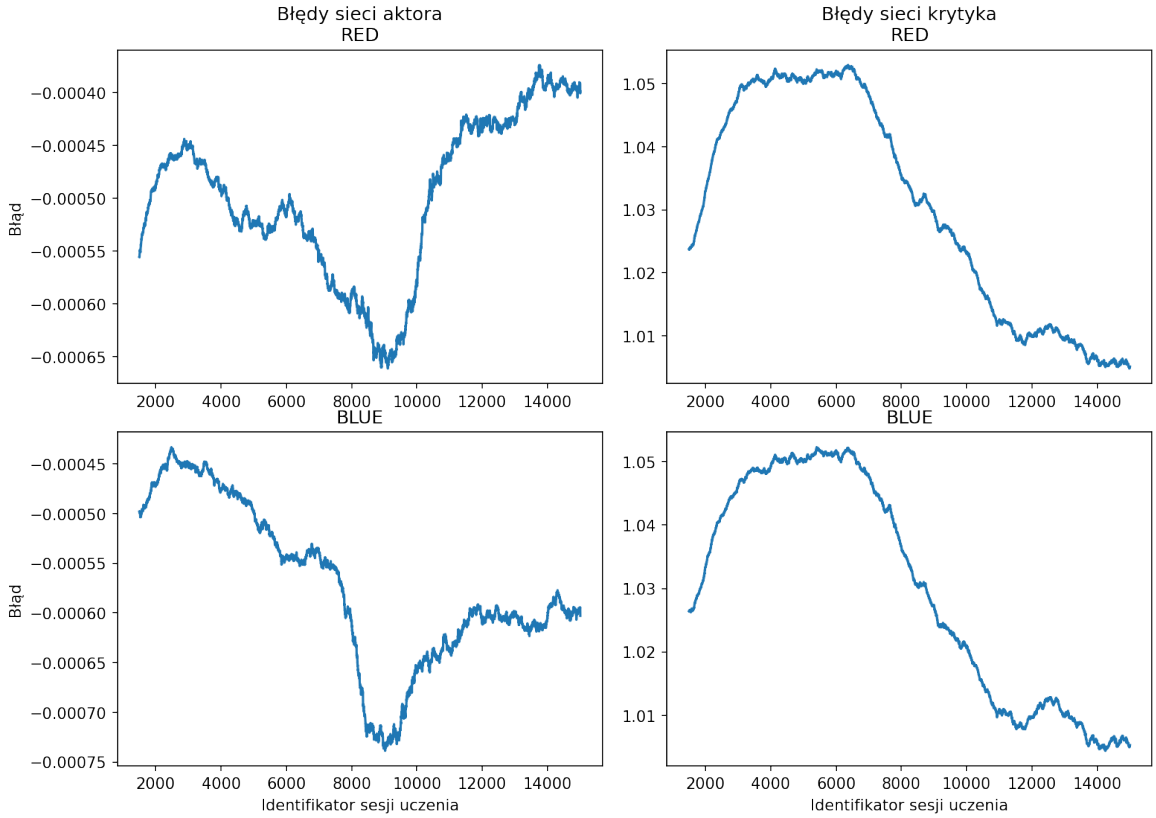
Rysunek 48: Wykresy przedstawiające porównanie pogrupowanych sumy wyników epizodów podejścia jednoczesnej nauki obu agentów.



Rysunek 49: Wykresy przedstawiające średnią kroczącą komponentów nagrody uzyskiwanych podczas jednoczesnej nauki obu agentów.



Rysunek 50: Wykresy przedstawiające średnią kroczącą długości epizodów wypracowanych podczas jednoczesnej nauki obu agentów.



Rysunek 51: Wykresy przedstawiające błędy sieci neuronowych zwrócone przez obu agentów w trakcie procesu uczenia.

Nagła różnica w wydajności sugeruje, że zadanie postawione przed agentem po stronie atakującej być może jest prostsze w zrozumieniu, aniżeli zadanie postawione przed agentem broniącym. Ciekawym jest również aspekt bardzo dużego podobieństwa wykresów błędów obu sieci, które w dużej mierze zapewne jest skutkiem przepracowania identycznych momentów decyzyjnych. Może świadczyć to o podobnym zrozumieniu środowiska, jednak ograniczenia początkowe narzucone przez środowisko na agenta broniącego (nie ma dostępu do piłki w początkowej konfiguracji) nie pozwala mu na samym początku maksymalizować swojej funkcji nagrody. To zaś może być w pewnym stopniu mankamentem samej funkcji nagrody zakładającej, że środowisko jest grą o sumie zerowej i ostatecznie nie jest ona najoptymalniejsza.

5 Podsumowanie

Podsumowując ogół pracy, na samym początku uargumentowany został wybór oraz przedstawione zostało sedno działania bardzo wydajnego jak na dzisiejszy stan techniki algorytmu uczenia przez wzmacnianie zwanego algorytmem optymalizacji polityki proksymalnej (PPO), który sam w sobie może być wykorzystany w bardzo szerokiej gamie środowisk. Jak widać przy załączonych badaniach hiperparametryzacji oraz procesie uczenia agentów ostatecznych, twórcy algorytmu bardzo skrupulatnie podeszli do swojej pracy z dywizą zapotrzebowania na algorytm wszechstronny ale i dostatecznie stabilny. Praca z samym algorytmem oraz proces wyboru jego hiperparametryzacji dzięki mechanizmowi wycinania nadmiaru rozbieżności prawdopodobieństw jest dużo prostszy w porównaniu do algorytmów pokrewnych takich jak A2C lub SAC, a jego stabilność i jej wpływ na zwracane wyniki w trakcie procesu uczenia jest satysfakcjonująca. Przez relatywnie prostą postać algorytmu oraz niskie zapotrzebowania które musi zapewnić środowisko jest on otwarty na wszelkie rozwinięcia oraz udoskonalenia, które w przyszłości mogą zapewnić jeszcze lepiej funkcjonującą sztuczną inteligencję. Do tej pory główną przeszkodą wydaje się być zapotrzebowanie czasowe, które w dużym stopniu może zostać zaspokojone poprzez możliwość zrównoleglenia całego procesu uczenia na wiele jednostek próbkujących. Wybór tego algorytmu jako głównego przez inżynierów OpenAI jest jak najbardziej trafną decyzją.

Wybór gry HaxBall jako środowiska uczenia maszynowego jest bardzo dobrą bazą poznawczą dla algorytmów operujących na ciągłej przestrzeni stanów. Wypracowane przez sieci neuronowe polityki gry w HaxBall nieustannie wzbudzają niedosyt, a ciągła chęć doskonalenia zwracanych w środowisku wyników wydaje się być zajęciem nie mającym końca. Skomplikowanie zachowań możliwych do wypracowania nie jest duże i pozwala na łatwą ich analizę oraz wnioskowanie przyczynowości, lecz aspekt potrzeby planowania pełnych zagrań może ukazywać siłę wybranego algorytmu. Środowisko pozwala nie tylko na łatwą ekstrakcję oraz analizę obrazu z gry, ale również przez prostotę jej logiki pozwala się w oczywisty i jednoznaczny sposób symulować.

Wyniki uzyskane przez połączenie wspomnianego powyżej algorytmu oraz środowiska są jak najbardziej zadowalające. Jednak zdecydowanie nie należy o nich sądzić jako o najwyższych możliwych do osiągnięcia w tym środowisku. Pewnym jest, że przedstawieni w tej pracy agenci przy dłuższym pozostawieniu w procesie uczenia osiągnęliby jeszcze lepszą wydajność, na opisanym etapie nie są jeszcze u szczytu swoich możliwości. Jednakże, kolejnym krokiem który należałoby postawić powinna być próba większej generalizacji zachowań w przypadku obu agentów. Niestety na tym etapie ich zachowania w większości odpowiadają temu na co pozwoli im przeciwnik i nie mają oni większych szans z ludzkimi graczami.

5.1 Oprogramowanie / Biblioteki

Projekt był uruchamiany na platformach opartych na systemie operacyjnym Windows 10 oraz MacOS. W tym podrozdziale wymienione zostało wszelkie wykorzystane oprogramowanie:

- Visual Studio Code 1.67.2
- PyCharm 2020.30.2 (Community Edition)
- Python 3.10.2, moduły:
  - TensorFlow ver. 2.7.0
  - NumPy ver. 1.19.4
  - pygame ver. 2.0.0
  - haxballgym ver. 0.3.0
  - Pillow ver. 8.0.1
  - OpenCV2 ver. 45.1.48
  - mss ver. 6.1.0
  - pynput ver. 1.7.3
  - pyautogui ver. 0.9.52
  - pydirectinput ver. 1.0.4
  - pandas ver. 1.2.3
  - matplotlib ver. 3.5.1
- Microsoft Edge 102.0.1245.33
- Google Chrome 102.0.5005.61

6 Bibliografia

Literatura

[1] Guofa Li, Yifan Yang, Shen Li, Xingda Qu, Nengchao Lyu, and Shengbo Eben Li. Decision making of autonomous vehicles in lane change scenarios: Deep reinforcement learning approaches with risk awareness. *Transportation Research Part C: Emerging Technologies*, 134:103452, January 2022.

[2] Rong Zhang, Qibing Lv, Jie Li, Jinsong Bao, Tianyuan Liu, and Shimin Liu. A reinforcement learning method for human-robot collaboration in assembly tasks. *Robotics and Computer-Integrated Manufacturing*, 73:102227, February 2022.

[3] Peter R. Wurman, Samuel Barrett, Kenta Kawamoto, James MacGlashan, Kaushik Subramanian, Thomas J. Walsh, Roberto Capobianco, Alisa Devlic, Franziska Eckert, Florian Fuchs, Leilani Gilpin, Piyush Khandelwal, Varun Kompella, HaoChih Lin, Patrick MacAlpine, Declan Oller, Takuma Seno, Craig Sherstan, Michael D. Thomure, Houmehr Aghabozorgi, Leon Barrett, Rory Douglas, Dion Whitehead, Peter Dürr, Peter Stone, Michael Spranger, and Hiroaki Kitano. Outracing champion Gran Turismo drivers with deep reinforcement learning. *Nature*, 602(7896):223–228, February 2022. Number: 7896 Publisher: Nature Publishing Group.

[4] DALL·E 2, https://openai.com/dall-e-2/.

[5] Tobiasz Pokorniecki. Teoria i praktyczne zastosowanie uczenia maszynowego w grze typu Pac-Man. *UMK 10-I-INF-291941*, January 2021.

[6] Abhishek Suran. On-Policy v/s Off-Policy Learning, July 2020.

[7] Stephen Zhen Gou and Yuyang Liu. DQN with model-based exploration: efficient learning on environments with sparse rewards. *arXiv:1903.09295 [cs, stat]*, March 2019. arXiv: 1903.09295.

[8] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous Deep Q-Learning with Model-based Acceleration. *arXiv:1603.00748 [cs]*, March 2016. arXiv: 1603.00748.

[9] Michael Janner, Justin Fu, Marvin Zhang, and Sergey Levine. When to Trust Your Model: Model-Based Policy Optimization. *arXiv:1906.08253 [cs, stat]*, November 2021. arXiv: 1906.08253.

[10] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *arXiv:1712.01815 [cs]*, December 2017. arXiv: 1712.01815.

[11] OpenAI. spinningup.openai.com, Part 2: Kinds of RL Algorithms.

[12] Somil Bansal, Roberto Calandra, Kurtland Chua, Sergey Levine, and Claire Tomlin. MBMF: Model-Based Priors for Model-Free Reinforcement Learning. *arXiv:1709.03153 [cs]*, October 2017. arXiv: 1709.03153.

[13] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv:1707.06347 [cs]*, August 2017. arXiv: 1707.06347.

[14] OpenAI. Deep Deterministic Policy Gradient — Spinning Up documentation, https://spinningup.openai.com/en/latest/algorithms/ddpg.html.

[15] OpenAI. Proximal Policy Optimization — Spinning Up documentation, https://spinningup.openai.com/en/latest/algorithms/ppo.html.

[16] OpenAI. Proximal Policy Optimization, https://openai.com/blog/openai-baselines-ppo/, July 2017.

[17] OpenAI Five, https://openai.com/five/, December 2019.

[18] OpenAI. Part 1: Key Concepts in RL — Spinning Up documentation, https://spinningup.openai.com/en/latest/spinningup/rl_intro.html.

[19] Vanilla Policy Gradient — Spinning Up documentation, https://spinningup.openai.com/en/latest/algorithms/vpg.html.

[20] Richard S Sutton and Andrew G Barto. Reinforcement Learning: An Introduction. *The MIT Press*, 2014.

[21] Sanyam Kapoor. Policy Gradients in a Nutshell, https://towardsdatascience.com/policy-gradients-in-a-nutshell-8b72f9743c5d, June 2018.

[22] Lilian Weng. Policy Gradient Algorithms, https://lilianweng.github.io/2018/04/08/policy-gradient-algorithms.html, April 2018.

[23] Shaked Zychlinski. The Complete Reinforcement Learning Dictionary, https://towardsdatascience.com/the-complete-reinforcement-learning-dictionary-e16230b7d24e, November 2019.

[24] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust Region Policy Optimization. February 2015.

[25] OpenAI. Part 3: Intro to Policy Optimization — Spinning Up documentation, https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html.

[26] HaxBall, https://www.haxball.com/.

[27] HaxBall Headless Host, https://www.haxball.com/headless.

[28] Headless Host Documentation, https://github.com/haxball/haxball-issues.

[29] Wazarr94 - GitHub, https://github.com/Wazarr94.

[30] haxballgym · PyPI, https://pypi.org/project/haxballgym/.

[31] Adam Gleave, Michael Dennis, Cody Wild, Neel Kant, Sergey Levine, and Stuart Russell. Adversarial Policies: Attacking Deep Reinforcement Learning. Technical Report arXiv:1905.10615, arXiv, January 2021. arXiv:1905.10615 [cs, stat] type: article.

[32] Brian Christian. *The Alignment Problem: Machine Learning and Human Values*. W.W. Norton, 2020. Google-Books-ID: VmJIzQEACAAJ.