UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ W LUBLINIE

Wydział Matematyki, Fizyki i Informatyki

Kierunek: Informatyka

**Adrian Kula**

nr albumu: 291038

**Uczenie przez wzmacnianie z wykorzystaniem Unity ML-Agents**

***Reinforcement learning with Unity ML-Agents***

Praca dyplomowa

napisana w Katedrze Systemów Inteligentnych

pod kierunkiem dr. Andrzeja Bobyka

**Lublin 2022**

Spis treści

[Wstęp 5](#_Toc104752563)

[Rozdział 1. Uczenie przez wzmacnianie jako część uczenia maszynowego 5](#_Toc104752564)

[1.1. Uczenie maszynowe 5](#_Toc104752565)

[1.2. Uczenie przez wzmacnianie 5](#_Toc104752566)

[1.2.1. Podstawowe pojęcia 6](#_Toc104752567)

[1.2.2. Historia uczenia przez wzmacnianie 7](#_Toc104752568)

[1.2.3. Głębokie uczenie przez wzmacnianie 9](#_Toc104752569)

[1.2.4. Przykłady wykorzystywania uczenia przez wzmacnianie 10](#_Toc104752570)

[Rozdział 2. Algorytmy uczenia przez wzmacnianie i technologie 11](#_Toc104752571)

[2.1. Podstawy uczenia przez wzmacnianie 11](#_Toc104752572)

[2.1.1. Procesy decyzyjne Markowa i równanie Bellman’a 11](#_Toc104752573)

[2.1.2. Programowanie dynamiczne 14](#_Toc104752574)

[2.2. Przykładowe algorytmy uczenia przez wzmacnianie 15](#_Toc104752575)

[2.2.1. Algorytmoy Monte-Carlo 15](#_Toc104752576)

[2.2.2. Uczenie metodą różnic czasowych 16](#_Toc104752577)

[2.2.3. Q-Learnig 17](#_Toc104752578)

[Algorytmy wykorzystywane przez ML-Agents 19](#_Toc104752579)

[2.2.4. PPO 19](#_Toc104752580)

[2.2.5. SAC 20](#_Toc104752581)

[2.3. Metody używane w uczeniu przez wzmacnianie 20](#_Toc104752582)

[2.3.1. Uczenie przez naśladowanie 21](#_Toc104752583)

[2.3.2. . Generative Adversarial Imitation Learning 21](#_Toc104752584)

[2.3.3. Klonowanie Behawioralne 21](#_Toc104752585)

[2.4. Biblioteki, technologie i programy 22](#_Toc104752586)

[2.4.1. Unity 22](#_Toc104752587)

[2.4.2. ML-Agents 24](#_Toc104752588)

[2.4.3. Język C# 24](#_Toc104752589)

[2.4.4. Język Python 24](#_Toc104752590)

[2.4.5. PyTorch 24](#_Toc104752591)

[Rozdział 3. Trenowanie agenta we własnej instancji środowiska 25](#_Toc104752592)

[3.1. Instalacja i konfiguracja 25](#_Toc104752593)

[3.1.1. Unity 25](#_Toc104752594)

[3.1.2. ML-Agents 25](#_Toc104752595)

[3.2. Stworzenie instancji środowiska i agenta 25](#_Toc104752596)

[3.3. Trenowanie agenta 25](#_Toc104752597)

[3.4. Próby 25](#_Toc104752598)

[3.5. Wnioski 25](#_Toc104752599)

[Podsumowanie 25](#_Toc104752600)

[Bibliografia 27](#_Toc104752601)

[Spis rysunków 29](#_Toc104752602)

[Spis tabel 31](#_Toc104752603)

[Streszczenie 33](#_Toc104752604)

# Wstęp

We wstępie

# Uczenie przez wzmacnianie jako część uczenia maszynowego

## Uczenie maszynowe

Uczenie maszynowe to dziedzina sztucznej inteligencji, skupiająca się na algorytmach, które uczą się z podanych danych i doskonalą się na podstawie zgromadzonej wiedzy. Algorytmy oparte na tej technologii mogą przewidywać różne wyniki, wybierać różne decyzje czy znajdować wzorce w dostarczonych danych. Z tego powodu znajdują swoje zastosowanie w wielu aspektach naszego życia między innymi w medycynie, robotyce, detekcji spamu, polecaniu treści czy przewidywaniu pogody. Systemy wykorzystujące tę technologię zwiększają swoją skuteczność wraz z czasem, a im lepszy mają dostęp do danych, tym większą wykazują dokładność. W dzisiejszych czasach wyróżniamy trzy główne kategorie uczenia maszynowego w zależności od charakteru sygnału w systemie.

Pierwszą z nich jest uczenie nadzorowane, gdzie podajemy komputerowi przykładowe zestawy wejścia i wyjścia. Na podstawie tych danych algorytm dostosowuje swój model aby wyuczyć się prawidłowej odpowiedzi. Ten rodzaj używany jest między innymi do klasyfikacji danych, modelowania regresji czy uczenia zespołowego.

Kolejną z kategorii jest uczenie nienadzorowane, gdzie podajemy komputerowi tylko dane wejściowe. Na podstawie tych danych algorytm uczenia nienadzorowanego ustala strukturę danych . Używamy go przy grupowaniu, wykrywaniu modeli czy redukcji wymiarowości.

## Uczenie przez wzmacnianie

Uczenie przez wzmacnianie jest trzecim z głównych rodzajów uczenia maszynowego. W przeciwieństwie do uczenia nadzorowanego i nienadzorowanego, algorytm uczenia przez wzmacnianie nie wykorzystuje do nauki wcześniej przygotowanych zbiorów danych, lecz całe środowisko, którym może być prawdziwy świat czy gra komputerowa. Mając to środowisko, algorytm wybiera akcję a sterowany tym algorytmem agent wykonuje ją. Środowisko z kolei zwraca agentowi swój nowy stan oraz nagrodę, która może być zarówno pozytywna jak i negatywna. Opisywaną pętlę uczenia przez wzmacnianie pokazano na rysunku 1. Celem agenta jest zmaksymalizowanie nagrody, co oznacza podejmowanie akcji, które zapewnią najlepsze wyniki i pożądany przez twórcę efekt. Sekwencja właściwych decyzji spowoduje wzmocnienie algorytmu, co przekłada się na efektywniejsze rozwiązywanie problemu.



Rysunek 1. Pętla wykorzystywana w uczeniu przez wzmacnianie

### Podstawowe pojęcia

Uczenie przez wzmacnianie wykorzystuje wiele elementów aby osiągnąć jak najlepsze wyniki. Wyróżniamy: politykę, sygnał nagrody, funkcję wartości, i opcjonalnie model środowiska.

#### Polityka

Polityka definiuje sposób w jaki osadzony w środowisku agent będzie się zachowywał. Inaczej mówiąc, polityka to mapowanie każdego ze stanów środowiska na akcje jakie mają zostać podjęte w tych stanach. W niektórych przypadkach polityka może być reprezentowana przez prostą funkcję czy tabelę, w innych może obejmować skomplikowane obliczenia jak np. proces wyszukiwania. Polityka jest najważniejszym elementem uczenia przez wzmacnianie, ponieważ określa ona zachowanie, jakim kieruje się agent.

#### Sygnał nagrody

Określa cel problemu uczenia przez wzmacnianie. W każdym kroku czasowym środowisko wysyła agentowi pewien numer zwany nagrodą. Celem agenta jest zmaksymalizowanie tego numeru. W związku z tym sygnał nagrody definiuje, co agent powinien robić a czego nie powinien. Sygnał nagrody jest również podstawą do zmiany polityki – jeśli po akcji podjętej przez politykę następuje ujemna lub niska nagroda, może ona zostać zmieniona aby w przyszłości podjąć lepszą akcję.

#### Wartość funkcji

Sygnał nagrody określa co jest dobre dla agenta a co złe w sensie bezpośrednim. Wartość funkcji określa co jest dla niego dobre w większym odstępie czasu. Ogólnie wartość stanu to całkowita kwota nagrody, której agent może oczekiwać w przyszłości zaczynając od tego stanu. Podczas gdy nagrody określają natychmiastową, samoistną wartość stanów środowiskowych, ich wartości określają długoterminową atrakcyjność stanów po uwzględnieniu stanów które prawdopodobnie nastąpią, oraz nagród dostępnych w tych stanach. Przykładowo, stan może dawać niską natychmiastową nagrodę, lecz nadal mieć wysoką wartość ponieważ regularnie następujące po nim inne stany zapewniają wysokie nagrody.

#### Model

Model środowiska obecny w niektórych systemach uczenia przez wzmacnianie naśladuje zachowanie środowiska, czyli pozwala wnioskować, w jaki sposób środowisko będzie się ono zachowywać. Przykładowo, biorąc pod uwagę stan i akcję, model może przewidzieć wynikowy następny stan i następną nagrodę. Model pozwala planować, czyli decydować o sposobie działania poprzez rozważenie możliwych przyszłych sytuacji, zanim one zostaną doświadczone. W uczeniu przez wzmacnianie rozróżniamy metody wykorzystujące modele i planowanie, nazywane metodami opartymi na modelach oraz modele uczące się metodą prób i błędów, czyli metody bez modelu

### Historia uczenia przez wzmacnianie

Historia uczenia przez wzmacnianie jest rozbita na trzy główne wątki, które rozwijały się niezależnie, a zostały połączone we współczesnym uczeniu przez wzmacnianie. Pierwszy z omawianych wątków dotyczy problemu sterowania optymalnego i jego rozwiązania za pomocą funkcji wartości i programowania dynamicznego. Drugim z nich jest nauka metodą prób i błędów zapoczątkowany w psychologii zwierząt. Ostatni z wątków łączy w sobie dwa poprzednie - są to metody uczenia różnic czasowych. Wszystkie trzy wątki zostały połączone w latach 80, dzięki czemu powstało uczenie przez wzmacnianie.

Termin „Sterowanie optymalne” zaczął być używany w 1950 roku i opisywał on problem projektowania sterownika w celu zminimalizowania miary zachowania systemu dynamicznego w czasie. W 1957 roku Richard Bellman wraz z innymi naukowcami stworzył na podstawie teorii Hamilton’a-Jacobi’ego metodę, która używając stan dynamicznego systemu, dynamicznie definiuje równanie funkcyjne i zwraca optymalną wartość funkcji. Metoda ta została nazwana równaniem Bellmana i jest do dziś stosowana w problemie optymalnego sterowania. Klasa metod które rozwiązują problem sterowania optymalnego przez rozwiązanie równania Bellmana stała się znana jako programowanie dynamiczne. W tym samym roku Bellman przedstawił procesy decyzyjne Markowa (MDP), które zdefiniował jako „dyskretne, stochastyczne wersje problemu optymalnego sterowania”. Pomysł ten został rozszerzony w 1960 roku przez Howard ’a, który zdefiniował metodę iteracji polityki dla MDP. Równanie Bellmana ma jednak pewną wadę, nazwaną przez jego twórcę „przekleństwem wymiarowości”. Polega ona na tym, że wymagania obliczeniowe tego równania rosną wykładniczo wraz z liczbą zmiennych stanu. Pomimo tej wady, programowanie dynamiczne do dziś jest najwydajniejszą i najbardziej popularną metodą rozwiązywania problemu optymalnego sterowania stworzoną do tej pory.

Drugim ze wspominanych wątków jest uczenie się metodą prób i błędów które ma swoje korzenie w psychologii zwierząt. W 1911 roku Edward Thorndike zdefiniował tę metodę jako prawo efektu. Twierdził on, że podjęta przez osobnika akcja w danym środowisku przynosi konsekwencje, prawdopodobieństwo podjęcia tej samej akcji zwiększa się lub zmniejsza w zależności od rodzaju konsekwencji. Koncept ten został wykorzystany w 1954 roku przez Marvin’a Minsky’egio, któremu przypisuje się pierwsze próby zastosowania nauki metodą prób i błędów pod kątem obliczeniowym. W swoim doktoracie opisał on maszynę własnej konstrukcji o nazwie SANRC (Stochastic Neural-Analog Reinforcement Calculators), Maszyna ta wykorzystywała paradygmat nagrody aby przechodzić przez labirynt i była prawdopodobnie pierwszą sztuczną maszyną samouczącą się.

W kolejnych latach badania metody prób i błędów pod kątem obliczeniowym zostały uogólnione do rozpoznawania wzorów, a następnie przypisane jako część uczenia nadzorowanego. Z tego powodu w okresie lat 60 i 70 powstało bardzo mało publikacji adresujących uczenie przez wzmacnianie, Jednym z badaczy którzy pomimo tego dalej prowadzili badania na tym obszarze był między innymi John Andreae, który w 1963 roku stworzył system STELLA – maszynę uczącą, która za pomocą metody prób i błędów uczyła się poprzez interakcję ze środowiskiem.

Wątek metod uczenia różnic czasowych również ma swoje korzenie w psychologii zwierząt i opiera się on na pojęciu wzmocnień wtórnych. Wzmocnienie wtórne odnosi się do sytuacji, w której bodziec wzmacnia zachowanie po powiązaniu ze wzmocnieniem pierwotnym (czyli bodźcem, który jest biologicznie ważny dla organizmu jak jedzenie czy sen). Minsky był prawdopodobnie pierwszym naukowcem który w 1954 roku połączył uczenie różnic czasowych z systemami sztucznej inteligencji. Z kolei Arthur Samuel w 1959 roku zaproponował i zaimplementował metodę nauczania zawierającą ideę uczenia różnic czasowych w swoim programie grającym w warcaby. Przez kolejną dekadę po stworzeniu prac przez Minsky’egio i Samuel ‘a badacze wykonali niewiele prac zarówno na temat uczenia różnic czasowych jak i uczenia metodą prób i błędów. Osobą odpowiedzialną za przywrócenie tych konceptów jest Harry Klopf, kiedy w latach 1972-1975 zbadał uczenie się przez wzmacnianie w dużych systemach jako skonceptualizowane komponenty większego systemu, każdy z własnymi bodźcami pobudzającymi jako nagrody i bodźcami hamującymi jako kary i każdy z nich może wzmacniać się nawzajem. Sutton rozwinął ideę Klopf ‘a i w swojej pracy powiązał ją z teorią uczenia się zwierząt. Dzięki temu opisał on zasady, według których uczenie się jest napędzane przez zmiany w czasowo następujących po sobie przewidywaniach. Jak się później okazało, ta praca była bardzo znacząca w uczniu przez wzmacnianie, ponieważ na jej podstawie napisano wiele innych, co przyczyniło się do spopularyzowania tematu uczenia przez wzmacnianie.

Finalnie, w 1989 roku wszystkie trzy wątki zostały połączone przez Christopher’a Watkins ‘a, gdy opracował on Q-learning. Od tamtej pory nastąpił ogromny wzrost liczby badań nad uczeniem przez wzmacnianie. W 1992 roku duży sukces osiągnęły programy wyuczone gry Backgammon, co zwróciło jeszcze większą uwagę badaczy.

### Głębokie uczenie przez wzmacnianie

Uczenie przez wzmacnianie jest bardzo dobrym podejściem gdy chcemy wyuczyć nasz program działania w pewnym środowisku. Niektóre z tych środowisk mogą jednak mieć ogromne ilości stanów a w każdym z tych stanów dziesiątki tysięcy możliwych do podjęcia akcji. Innymi słowy – niektóre środowiska są po prostu zbyt skomplikowane. Aby rozwiązać ten problem, naukowcy postanowili dodać sieci neuronowe do algorytmów uczenia przez wzmacnianie. W ten sposób otrzymano głębokie uczenie przez wzmacnianie, które rozwiązuje problem skomplikowanych środowisk. Słowo „głębokie” odnosi się do liczby warstw sieci neuronowej, przez które przechodzą dane. Dzięki temu połączeniu zastępujemy mapowanie każdego możliwego rozwiązania aproksymacją funkcji. W ten sposób nie tylko eliminujemy potrzebę przechowywania wszystkich możliwych rozwiązań, lecz także umożliwiamy agentowi uogólnianie wartości stanów, o których agent jeszcze nic nie wie. Dzięki głębokiemu uczeniu przez wzmacnianie algorytm może pobierać olbrzymie ilości danych (przykładowo wszystkie piksele wyrenderowane na ekranie przez grę komputerową), przeanalizować je za pomocą sieci neuronowych i zdecydować, jaką akcję podjąć.

### Przykłady wykorzystywania uczenia przez wzmacnianie

Dzięki temu, że środowiskiem mogą być najróżniejsze przestrzenie jak na przykład giełda, gra komputerowa czy prawdziwy świat, uczenie przez wzmacnianie wykorzystywane jest w najróżniejszych dziedzinach życia. Poniżej opisano niektóre z możliwych zastosowań tego rodzaju uczenia maszynowego.

#### Granie w gry

Gra „Go” grą planszową znaną na całym świecie. Dzięki uczeniu przez wzmacnianie naukowcy stworzyli programy, które nie tylko wyuczyły się zasad tej gry, lecz także potrafią grać znacznie lepiej od człowieka. Świetnym przykładem jest tutaj AlphaGo – program stworzony przez DeepMind, który dzięki uczeniu przez wzmacnianie był w stanie jako pierwszy pokonać mistrza świata w grę „Go”, i do tej pory jest najlepszym graczem tej gry w historii. Szachy, sudoku czy warcaby – to tylko niektóre gry w które komputery dzięki uczeniu przez wzmacnianie są w stanie grać, a nawet osiągać mistrzostwo.

Oprócz gier w świecie rzeczywistym, algorytmy uczenia przez wzmacnianie są również w stanie grać w gry komputerowe. Tutaj z kolei przykładem może być gra „Breakout” stworzona przez Atari. Poprzez sczytywanie pikseli na ekranie i przepuszczanie tych informacji przez kolejne warstwy sieci neuronowej, algorytm podejmuje akcję, która odpowiada za przesunięcie platformy w lewo lub prawo. Jeżeli akcja przyniesie punkty (czyli pozytywną nagrodę), algorytm wzmocni się poprzez dostosowanie swoich neuronów i będzie podejmował daną akcję częściej. Dzięki uczeniu przez wzmacnianie, algorytmy są w stanie grać zarówno w proste gry jak „Snake” czy „Flappy Bird” jak i te bardziej skomplikowane, przykładowo „Starcraft” czy „League of Legends”.

#### Pojazdy autonomiczne

Podczas jazdy samochodem, należy wziąć pod uwagę wiele czynników jak na przykład ograniczenia prędkości, znaki drogowe czy unikanie kolizji. Do zadań związanych z jazdą autonomiczną można zastosować algorytmy uczenia przez wzmacnianie. Środowiskiem autonomicznego samochodu jest droga i jej otoczenie. Dane z kamer są przekazywane do warstwy wejścia algorytmu a na wyjściu otrzymujemy ruch kierownicy. Dzięki uczeniu przez wzmacnianie samochód może nie tylko jeździć po drodze, ale także na przykład parkować.

# Algorytmy uczenia przez wzmacnianie i wykorzystane technologie

## Podstawy uczenia przez wzmacnianie

Uczenie przez wzmacnianie opiera się na procesie decyzyjnym markowa, który jest sekwencyjnym procesem decyzyjnym środowiska. Przedstawia się w nim wartości stanów oznaczające jak dobry jest dany stan oraz wartości akcji oznaczające jak dobra jest dana akcja w aktualnym stanie. Aby rozwiązać ten problem stosuje się równanie Bellmana, które rozkłada funkcje wartości na dwa czynniki – nagrodę natychmiastową i zdyskontowaną przyszłą nagrodę. Z kolei za pomocą programowania dynamicznego można iterować wartości stanu oraz politykę .

### Proces decyzyjny Markowa i równanie Bellman’a

Proces decyzyjny Markowa to probabilistyczny model sekwencyjnego problemu decyzyjnego w którym spełniona jest tzw. Własność Markowa [7]. Własność Markowa z kolei oznacza, że następny stan i przyznana nagroda w procesie Markowa zależy tylko od bieżącego stanu i podjętej w nim akcji a nie przykładowo historii stanów. Skończony proces decyzyjny Markowa można zapisać jako zestaw pięciu elementów:

gdzie: **•** jest skończonym zbiorem stanów  **•** jest skończonym zbiorem akcji, oraz jest skończonym zbiorem akcji możliwych do podjęcia w stanie środowiska w kroku czasowym t **•**  jest funkcją przejścia ze stanu do stanu takim że:

czyli prawdopodobieństwo że podjęta akcja w stanie środowiska w  kroku czasowym doprowadzi do zmiany stanu środowiska na stan w kroku czasowym . **•**  jest funkcją nagrody przyznaną agentowi po przejściu środowiska ze stanu do stanu

W każdym kroku czasowym algorytm podejmuje akcję spośród możliwych akcji w stanie . A jego celem jest zmaksymalizowanie nagród które dostanie po przejściu ze stanu do stanu .

Jak wspomniano w rozdziale 1.2.1.1. polityka to strategia którą kieruje się agent, czyli przyporządkowanie akcji każdemu z możliwych stanów. Matematyczny zapis polityki w MDP wygląda następująco:

Gdzie to prawdopodobieństwo podjęcia danej akcji w stanie pod warunkiem kierowana się przez agenta polityką . Celem MDP jest wyszukanie optymalnej polityki czyli takiej, która maksymalizuje skumulowaną nagrodę.

#### Nagrody w MDP

Wspomnianą nagrodę oznaczamy jako sumę wszystkich otrzymanych nagród od stanu początkowego aż do stanu końcowego:

Gdzie:

**•** jest sumą nagród

**•** jest otrzymaną nagrodą w kroku czasowym

**•** jest stanem kończącym

Do równania 2.4 możemy dodać czynnik dyskontowy gamma , który sprawia, że nagrody otrzymane później mają mniejsze znaczenie od tych uzyskanych wcześniej:

Wybranie niskiej wartości parametru gamma da nam zachłanną politykę co oznacza, że agent najbardziej ceni nagrody natychmiastowe, natomiast wybranie wysokiej wartości oznacza, że agent będzie kierował się długoterminowymi zyskami.

#### Wartość stanu i wartość akcji

MDP definiuje funkcję wartości stanu, która wylicza, jak dobry jest stan , gdy agent kieruje się polityką . Funkcję tę oznaczamy jako a jej wartość wyliczamy w następujący sposób:

Gdzie jest wartością oczekiwaną pod warunkiem, że agent kieruje się polityką

Podobnie do funkcji oceniania stanów, istnieje możliwość określenia, jak dobra jest akcja podjęta w danym stanie. Oznaczamy tę wartość jako a równanie tej funkcji wygląda w następujący sposób:

#### Równanie Bellmana

Równanie Bellmana jest wykorzystywane w uczeniu przez wzmacnianie, aby uprościć obliczanie wartości funkcji, poprzez rozłożenie równania wartości funkcji na dwie części – nagrodę natychmiastową plus zdyskontowane przyszłe wartości funkcji. Po zastosowaniu równania Bellman’a na funkcji wartości stanu otrzymamy:                                  

Jak wspomniano, powyższe równanie jest podzielone na dwie części. Pierwszą z nich jest nagroda jaką agent otrzyma od razu po podjęciu akcji będąc w stanie oznaczoną przez literę , natomiast to część wzoru odpowiadająca za wartość następnego stanu, które są brane pod uwagę podczas wyliczania wartości stanu obecnego.

Równanie Bellmana możemy również zastosować do funkcji wartości akcji:

#### Optymalna wartość stanu, optymalna wartość akcji i optymalna polityka

Ponieważ w każdym MDP istnieje wiele funkcji wartości stanów ze względu na wiele polityk, jakimi może kierować się agent, istnieje też co najmniej jedna optymalna funkcja wartości stanu. Optymalna funkcja wartości stanu to taka która daje maksymalną wartość w porównaniu do wszystkich innych funkcji wartości i zapisujemy ją wzorem:

Gdzie mówi nam, jaka jest maksymalna nagroda, jaką oferuje MDP.

Optymalna wartość akcji z kolei jest oznaczana następującym wzorem:

I w ten sposób otrzymujemy maksymalną nagrodę jaką agent może dostać jeżeli znajduje się w stanie s i podejmie akcję a

Polityka jest lepsza od polityki wtedy, gdy funkcja wartości wszystkich stanów jest większa od wartości wszystkich stanów polityki . Optymalna polityka to taka, która skutkuje optymalną funkcją wartości. Warto w tym momencie zaznaczyć, że optymalnych polityk może być więcej niż jedna. Matematycznie optymalną politykę można zapisać jako:

#### Przykład MDP

### Programowanie dynamiczne

Programowanie dynamiczne to zbiór algorytmów, które rozwiązują skomplikowane problemy poprzez podzielenie ich na mniejsze i znalezienie na nie optymalnego rozwiązania, co w rezultacie da optymalne rozwiązanie oryginalnego problemu. Istnieją dwa główne podejścia jakie można obrać, rozwiązując dany problem za pomocą programowania :

**• Podejście oddolne** – Zaczynamy od rozwiązania najmniejszych pod problemów jakie występują w podanym problemie, a następnie używamy tych wyników do rozwiązania pod problemu, którego częścią jest problem, który właśnie rozwiązaliśmy. Dzięki temu będziemy rozwiązywać coraz trudniejsze problemy aż dojdziemy do rozwiązania głównego problemu.

**• Podejście odgórne** – Rozkładamy nasz główny problem na szereg wszystkich możliwych pod problemów, a następnie próbujemy je wszystkie rozwiązać. Za każdym razem gdy zajmujemy się nowym problemem, sprawdzamy, czy został on już rozwiązany. Jeżeli tak, używamy jego wyniku, jeżeli nie, rozwiązujemy go. W ten sposób rekurencyjnie, krok po kroku algorytm rozwiąże wszystkie pod problemy a w konsekwencji, rozwiąże oryginalny problem.

Aby rozwiązać MDP można wykorzystać jeden z algorytmów programowania dynamicznego – iterację wartoścci. Iteracja wartości stanu wykorzystuje równanie Bellmana i pozwala na iteracyjne coraz lepsze oszacowanie wartości danego stanu. Matematyczny wzór iteracji wartości wygląda następująco:

Gdzie to prawdopodobieństwo przejścia do stanu i otrzymania nagrody r pod warunkiem znajdowania się w stanie s i podjęcia w nim akcji a. W ten sposób w każdym kroku czasowym aktualizujemy i ulepszamy wartość stanu w jakim się znajdujemy, przez co jesteśmy w stanie coraz lepiej wyliczać tę wartość.

#### Q-wartość

W przypadku gdy agent nie zna prawdopodobieństw przejścia, ani wartości nagród za znalezienie się w danym stanie, możemy użyć tak zwanej Q-wartości oznaczanej wzorem:

Różnca pomiędzy równaniem wartości funkcji a powyższym jest polega na tym, że zamiast nieznanej wartości oczekiwanej nagrody wykorzystywana jest faktycznie otrzymana wartość nagrody a zamiast oczekiwanej wartości następnego stanu, wartość stanu który faktycznie stał się stanem środowiska [12]. Dzięki temu, agent wykorzystujący Q-wartość nie musi znać modelu w przeciwieństwie do agenta iteracje wartości i polityki.

podczas gdy równanie wartości funkcji wymaga wiedzy o prawdopodobieństwach przejścia i nagrodach – (czyli modelu), cała ta wiedza jest przechowywana w funkcji Q-wartości. Dzięki temu algorytm może uaktualniać Q-wartość nie znając modelu, ponieważ informacja o następnym stanie znajduje się już w tej Q-wartości.

## Przykładowe algorytmy uczenia przez wzmacnianie

Istnieje wiele algorytmów uczenia przez wzmacnianie. Głównym podziałem tych algorytmów jest podział na algorytmy w polityce i poza polityką.

**• W polityce** – Algorytm ulepsza politykę, którą aktualnie się kieruje. Przykład: Q-learning.

**• Poza polityką** –Algorytmkieruje się jedną polityką do ulepszania innej. Przykład: SARSA.

Dodatkowym pojęciami używanymi w dalszych częściach tej pracy są eksploracja i eksploatacja. Eksploracja polega na poznawaniu środowiska, często podejmowaniu losowych akcji, dzięki czemu agent lepiej pozna model środowiska. Eksploatacja polega na korzystaniu ze zgromadzonej wiedzy, aby zmaksymalizować wyniki.

### Algorytmoy Monte-Carlo

Monte Carlo to algorytm, bazujący na metodzie prób i błędów. Przechodzi on najpierw przez cały epizod i sumuje zdyskontowaną nagrodę za znalezienie się w każdym ze stanów a następnie dystrybuuje tę nagrodę pomiędzy wszystkimi stanami w jakich się znalazł. Suma nagród zapisywana jest wzorem:

Natomiast przypisanie wartości każdemu ze stanów można zapisać wzorem:

Gdzie:

**•** to nowa wartość stanu który został odwiedzony w kroku czasowym

**•**  to aktualna wartość stanu który został odwiedzony w kroku czasowym

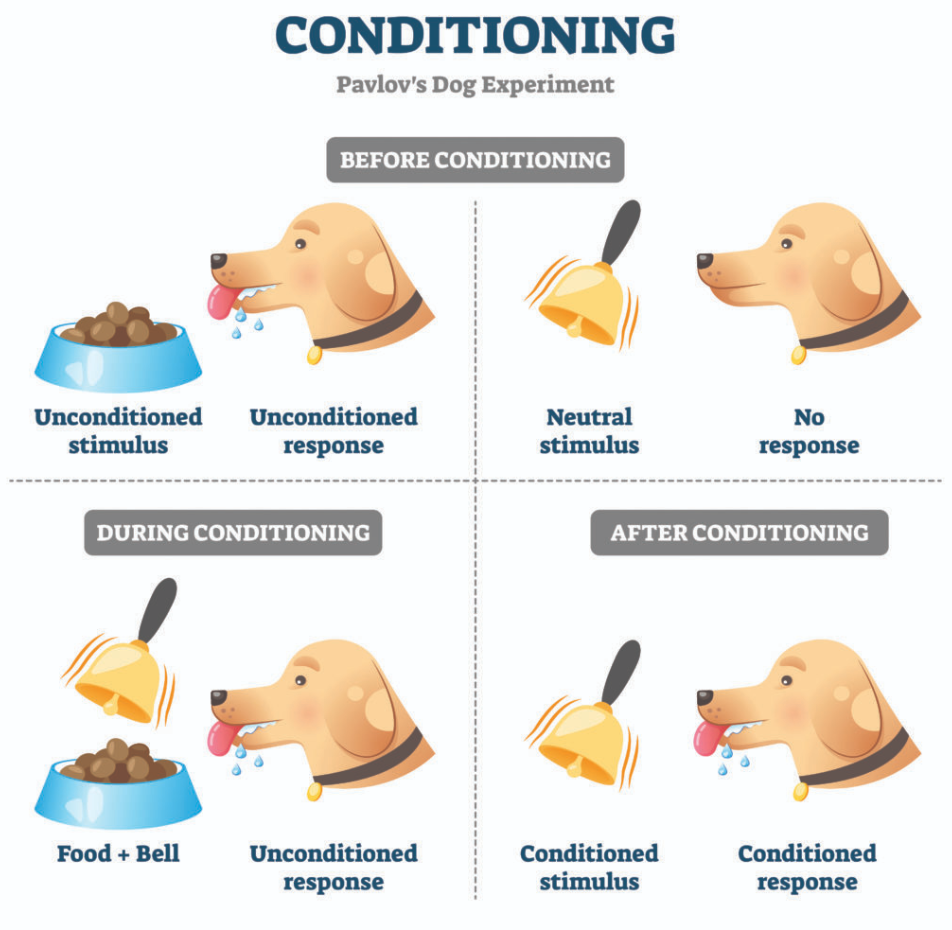
Takie podejście jest jednak bardzo nieskuteczne, ponieważ każdemu ze stanów algorytm przypisuje dokładnie taką samą wartość. Załóżmy, że algorytm gra w szachy. Gdy zrobi mnóstwo świetnych ruchów i kilka złych pod koniec, te złe wpłyną na wszystkie co  „przekłamie” nagrodę a w rezultacie wartości stanów. Z tego też powodu algorytm uczy się bardzo wolno i zanim wyuczy się rzeczywistych wartości stanów w środowisku, minie dużo czasu.

### Uczenie metodą różnic czasowych

Kolejnym algorytmem jest algorytm uczenia metodą różnic czasowych (Temporal difference learnig - TD). Różni się on od algorytmu Monte Carlo tym, że nie wylicza wartości stanów po całym epizodzie, tylko po każdym kolejnym kroku czasowym. Bazuje on na eksperymencie psa Pawłowa [rysunek 3] który polega na tym, że za każdym razem gdy zadzwonił dzwonek, pies dostawał po pewnym czasie nagrodę w postaci jedzenia. W związku z tym, za każdym razem gdy pies usłyszał dzwonek, zaczynał się ślinić ponieważ warunkował on dostanie jedzenia dzwonieniem dzwonka. Zainspirowany tym eksperymentem algorytm uczenia metodą różnic czasowych działa podobnie – zakłada on, że akcje podjęte obecnie wiążą się z nagrodami otrzymanymi w przyszłości.

Aby uaktualnić wartość stanu w tym algorytmie korzystamy ze wzoru:

W powyższym równaniu to wartość przewidywana dla stanu , od której odejmujemy aktualną wartość stanu i otrzymujemy w ten sposób błąd różnicy czasowej – wartość która wskazuje nam, jak bardzo wartość przewidywana różni się od wartości aktualnej. Mnożymy tę wartość przez wagę a następnie dodajmy ją do aktualnej wartości stanu .



*Rysunek 3. Eksperyment psa Pawłowa [PRZYPIS]*

### Q-Learnig

Jak wspomniano w rozdziale 1.2.2. Q-learning został stworzony w 1989 roku przez Christopher’a Watkins ‘a. Wzór wyliczenia Q-wartości w Q-learningu jest podobny do wzoru wykorzystywanego w uczeniu metodą różnic czasowych i wygląda następująco:

#### SARSA

SARSA (State-Action-Reward-State-Action) jest podobnym algorytmem do Q-learning’u, Jego nazwa wywodzi się z podejścia jakim posługuje się SARSA: zaczynając w danym stanie, podejmujemy akcję, dostajemy nagrodę i po przejściu w następny stan podejmujemy kolejną akcję. Matematyczny zapis wartości funkcji wygląda następująco:

Od razu można zauważyć, że wzór jest prawie taki sam jak ten używany w Q-learningu, jednak jest w nim pewna różnica. Wyliczając wspomnianą wcześniej wartość przewidywaną dla pary bierzemy pod uwagę Q-wartość stanu co oznacza, że algorytm podejmuje akcję na podstawie polityki którą się aktualnie kieruje, podczas gdy Q-learning wylicza Q-wartość stanu co oznacza, że podejmuje akcję na podstawie optymalnej polityki.

#### DQN

W sytuacji, gdy mamy tysiące stanów i tysiące przejść między nimi, przechowywanie Q-wartości dla każdej pary stanu i akcji może być uciążliwe z dwóch powodów – przechowywanie każdej Q-wartości dla pary stanu i akcji zajmowałoby ogromne ilości pamięci, oraz samo wyliczenie tych wartości zajmowałoby ogromną ilość czasu. Rozwiązaniem tych problemów jest DQN(Deep Q-Network), czyli zastąpienie tabeli Q-wartości siecią neuronową. Podajemy stan środowiska jako wejście, natomiast jako wyjście otrzymujemy Q-wartości dla każdej z możliwych w tym stanie akcji w parze z tym stanem:

Dzięki takiemu podejściu oszczędzamy zarówno pamięć, ponieważ nie musimy zapamiętyhwać Q-wartości każdego ze stanów w tabeli, ale także czas, ponieważ dzięki użyciu sieci neuronowej algorytm może rozpoznawać stany, których jeszcze nie odwiedził.

DQN korzysta głównie z dwóch technik, aby zachować stabilność: Dwie sieci neuronowe zwane siecią docelową i siecią główną oraz bufora powtórek[13]. Bufor powtórek to tabela zawierająca doświadczenia agenta oznaczane jako:

Doświadczenia te są dodawane do tabeli po każdym ruchu agenta a najprostsza implementacja tego bufora to stworzenie tabeli o stałym rozmiarze, gdzie nowe dane dodajemy na koniec tabeli a stare kasujemy.

Algorytm DQN inicjuje na początku bufor powtórek oraz dwie sieci neuronowe. Następnie inicjalizuje on początkowy stan w podanym mu środowisku aby za pomocą na przykład podejścia -greedy opisanego w rozdziale 2.1 podjąć akcję losową lub taką, która po wykorzystaniu głównej sieci neuronowej da najwyższą Q-wartość. Po jej wykonaniu, środowisko zwraca nagrodę i nowy stan, a poprzedni stan, podjęta akcja, nagroda i nowy stan zapisywane są do bufora powtórek. W następnej kolejności algorytm pobiera próbkę doświadczeń ze wspomnianego bufora i dla każdego z nich za pomocą sieci docelowej wylicza przewidywane maksymalne Q-wartości danych par następnych stanów i wszystkich możliwych w nich akcji, z uwagą, czy dany stan jest stanem kończącym, Mając te dane i po wyliczeniu Q-wartości z głównej sieci neuronowej po podaniu jej danych z próbki, algorytm oblicza funkcję straty między siecią docelową a główną i wykonuje zejście gradientowe aby zaktualizować wagi sieci głównej. Dodatkowo, co kroków, wagi sieci głównej są kopiowane do sieci docelowej.

Pomimo wspominanych wcześniej zalet, DQN ma również kilka wad. Przykładowo, ten algorytm nie jest w stanie wyoczyć się stochastycznych polityk, ponieważ podczas wybierania akcji , wybieramy akcję która da nam największą Q-wartość. Inną z wad jest to, że DQN nie może zostać zastosowany w środowiskach, w których pole akcji jest ciągłe, ponieważ wyliczenie funkcji argmax(a) w takich środowiskach jest bardzo trudne.

## Biblioteki, technologie i programy

W niniejszej pracy zostaną wykorzystane

### Unity

Unity to wieloplatformowy silnik do tworzenia między innymi gier, animacji czy symulacji. Dzięki swojej prostocie, jest on używany przez miliony programistów na całym świecie i wykonano w nim wiele gier takich jak „Cities: Skylines”, „The forrest” czy „Firewatch”. W 2021 roku badanie przeprowadzone przez Unity Technologies wykazało, że Unity pozostaje najczęściej wybieranym silnikiem gier, ponieważ aż 61% ankietowanych programistów korzysta z tego oprogramowania [8].

Po stworzeniu projektu w Unity Hub, wyświetli się główny edytor Unity z otwartym projektem. Interfejs Unity podzielony jest na kilka sekcji



*Rysunek 3. Interfejs Unity [przypis]*

**• A** –Pasek narzędzi daje dostęp do różnych funkcji silnika unity. Są to między innymi: Opcje włączenia i wstrzymania gry, zmianę narzędzia na narzędzie do przesuwania czy obracania, zarządzanie kontem czy wybór warstw będących w projekcie.

**• B** –Okno hierarchii pokazujące obiekty znajdujące się na scenie. Na każdym z obiektów można wykonywać podstawowe operacje jak dodawanie obiektu, usuwanie, zmiana jego nazwy czy duplikowanie. Obiekty można również dziedziczyć po sobie co pozwala na uzależnianie jednych obiektów od drugich.

**• C** –Okno gry symuluje w jaki sposób będzie wyglądać gra po jej włączeniu z widoku umieszczonych kamer. W momencie kliknięcia przycisku włączenia gry, ta symulacja się rozpoczyna.

**• D** –Oknosceny pozwala na zarządzanie sceną i obiektami na niej w sposób wizualny.

**• E** –Nakładki to podstawowe narzędzia pozwalające na manipulację zarówno sceną jak i obiektami na niej.

**• F** –Każdy z obiektów widocznych w sekcji A ma swoje własne właściwości, a zaznaczenie jednego z nich pokaże je w oknie B zwanym Inspektorem. Właściwości obiektu można dodawać, usuwać czy edytować. Każdy obiekt w oknie A ma własne właściwości, w związku z tym, to okno będzie się zmieniać za każdym razem, gdy użytkownik wybierze inny obiekt.

**• G** –Okno zasobów projektu pokazuje pliki i foldery obecne w projekcie. Importowanie zasobów czy tworzenie nowych powoduje, że pojawiają się one w tym oknie.

**• H** - Pasek powiadomień informuje o procesach które Unity wykonuje w tle.

### ML-Agents

Unity ML-Agents to projekt typu open-source, który pozwala stworzonym przez programistę grom i symulacjom działać jako środowisko w którym może uczyć się inteligentny agent. Ten zestaw narzędzi zapewnia implementację wspomnianych wcześniej algorytmów uczenia przez wzmacnianie (PPO i SAC), dzięki czemu programiści mogą z łatwością tworzyć i trenować agentów w stworzonych przez siebie środowiskach. Połączenie możliwości silnika Unity z uczeniem przez wzmacnianie sprawia, że ML-Agents jest korzystny zarówno dla twórców gier jak i naukowców zajmujących się sztuczną inteligencją. Projekt dostarcza ponad 18 stworzonych środowisk i wytrenowanych w nim agentów co pozwala na przenalizowanie kodu i zapoznanie się z podejściami obranych w tych przykładach. Oprócz algorytmów uczenia przez wzmacnianie, ML-Agents zapewnia również wsparcie dla uczenia przez naśladowanie, opisanego w rozdziale 2.4.3. co dodatkowo wzbogaca zakres możliwości tego projektu.

ML-Agents składa się z pięciu elementów, a pierwsze cztery z nich pokazano na rysunku 4:

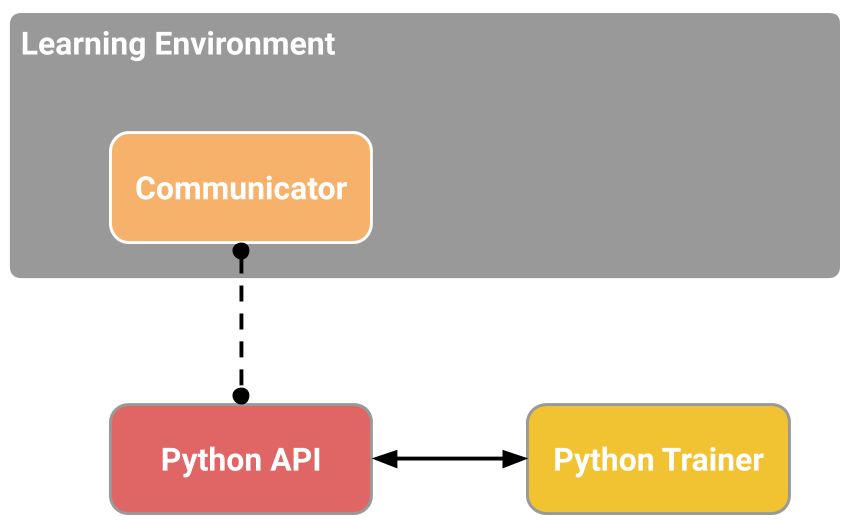
**•** Środowisko – Scena Unity zawierające wszystkie obiekty. W środowisku agent może zbierać obserwacje, wykonywać akcje i się uczyć.

**•** Komunikator – Łączy środowisko z API python’a

**•** API python’a – Zawiera interfejs Python’a do interakcji i manipulo

**•** Zbiór bibliotek python’a – Zawiera biblioteki uczenia maszynowego, co umożliwia trenowanie agentów.

**•** Gym wrapper – Jednym ze sposobów w jaki badacze mogą wchodzić w interakcje ze środowiskami jest stworzone przez OpenAI oprogramowanie Gym. ML-Agents dostarcza Gym wrapper, który pozwala na wykorzystanie Gym do interakcji ze środowiskiem stworzonym w Unity.



*Rysunek 4. Uproszczony schemat ML-Agents[PRZYPIS]*

### Język C#

Język C# to wysokopoziomowy, obiektowy język programowania stworzony w latach 1998-2001 przez firmę Microsoft. Programiści wykorzystują go głównie do tworzenia stron internetowych za pomocą platformy .NET czy aplikacji działających w systemie Windows. Unity wykorzystuje ten język do definiowania zachowywania obiektów w środowisku. Najpopularniejszym edytorem języka jest Visual Studio Code [9] stworzony przez Microsoft.

### Język Python

Python jest wysokopoziomowym językiem programowania ogólnego przeznaczenia. Cechą charakterystyczną tego języka jest to, że zamiast nawiasów klamrowych używa on wcięć, co sprzyja czytelności kodu. Ten stworzony w późnych latach 80 przez Guido Van Rossuma język posiada mnóstwo bibliotek pozwalających na wykonywanie najróżniejszych operacji co czyni go najpopularniejszym językiem na świecie [10]. ML-Agens używa tego języka aby wykonywać algorytmy uczenia przez wzmacnianie.

### PyTorch

PyTorch to biblioteka języka Python dotycząca uczenia maszynowego. Bazuje ona na bibliotece Torch i została stworzona przez Facebook’a 2016 roku. Z PyTorch’a korzysta wiele firm wytwarzających oprogramowanie związanych z głębokim uczeniem jak na przykład OpenAI czy Microsoft.

## Algorytmy wykorzystywane przez ML-Agents

ML-Agents korzysta z dwóch algorytmów - PPO (Proximal Policy Optimization) i SAC (Soft-actor critic). A

### Metody gradientu polityki

Metody gradientu polityki podobnie do Q-learning’u bazują na Procesie Decyzyjnym Markowa, a co za tym idzie – korzysta z podobnych konceptów i wzorów. Jak wspomniano wcześniej, celem agenta jest zmaksymalizowanie oczekiwanej nagrody podążając za polityką . Gdy dodamy do tego sieć neuronową wprowadzamy nowy symbol , czyli wagi sieci. Zakładając że sekwencja stanów, akcji i nagród oznaczona zostanie jako a funkcja będzie oznaczać sumę nagród z danej sekwencji, otrzymamy wzór cel zapisany wzorem matematycznym:[22]

Co oznacza, że funkcja J() jest równa wartości oczekiwanej nagrody kierując się polityką i dokładnie tę wartość nasz algorytm stara się zmaksymalizować.

Aby zaktualizować naszą politykę musimy zaktualizować sieć neuronową za pomocą zejścia (lub wejścia) gradientowego. Otrzymujemy więc taki wzór aby to zrobić:

Gdzie:

to wagi sieci neuronowej

to gradient funkcji

to szybkość uczenia

Aktualizowanie wag sieci neuronowej sieci zmienia całą politykę, co oznacza, że prawdopodobieństwa podjęcia akcji zmieniają się.

### Metoda aktora-krytyka

Metoda aktora-krytyka, to metoda która korzysta z dwóch sieci neuronowych. Pierwsza z nich to aktor, który podejmuje działania na podstawie danego stanu. Druga to krytyk, który ocenia działania podjęte przez aktora na podstawie polityki. Uczenie metodą aktora-krytyka to uczrenie w polityce, ponieważ krytyk musi uczyć się i skrytykować politykę, którą aktualnie stosuje aktor. Krytyka przybiera formę błędu różnicy czasowej, co opisano w rozdziale 2.2.2.

### Algorytm PPO

PPO to algorytm bardziej zaawansowany i stabilny od DQN. Został przedstawiony po raz pierwszy w pracy „Proximal Policy Optimization Algorithms” autorsta John’a Schulman’a z OpenAI [17]. PPO jest rozszerzeniem algorytmu TRPO (Trust Region Policy Optimization), który korzysta z metody polityki gradientu, oraz kilku podstawowych mechanik, takich jak region zaufania, czy dywergencja Kullbacka-Leiblera. Framework ML-Agents używa algorytm PPO jako domyślny.

Jak wspomniano wcześniej, metody gradientu polityki podczas aktualizacji wag, zmieniają całą politykę. TRPO próbuje zastosować pewne ograniczenie aby nowa polityka nie różniła się zbyt bardzo od poprzedniej. Tym ograniczeniem jest region zaufania (trust region), który jest jednym z dwóch głównych metod optymalizacji sieci neuronowych. Region zaufania jest techniką w której algorytm wybiera punkt na wykresie wokół którego tworzy region zaufania i w tym regionie wybierany jest najniższy punkt. Całość powtarzana jest aż do znalezienia najniższego punktu. Dodatkowo, aby lepiej kontrolować szybkość uczenia, możemy w czasie działanie rozszerzać lub kurczyć ten region

TRPO korzysta z wielu metod i mechanizmów podczas swojego działania. Jednym z nich jest dywergencja Kullbacka-Leibera , która pozwala zmierzyć różnicę między poprzednią a nową polityką. Wzór na dywergencję Kullbacka-Leibera dla rozkładów dyskretnych wygląda następująco:

Gdy to równanie jest równe zero, wtedy stara i nowa polityka nie różnią się wcale a im wyższa jest wartość tego równania, tym bardziej stara polityka różni się od nowej.

Pomimo tego, że algorytm TRPO sprawuje się bardzo dobrze co zostało poddane próbie wiele razy, ma on też sporą ilość wad. Przykładowo, jest bardzo wrażliwy na jakiekolwiek zmiany w hiper parametrach, oraz jest bardzo wymagający obliczeniowo.

Odpowiedzią na te problemy jest PPO, który miał na celu znalezienie równowago pomiędzy łatwością implementacji, łatwością dostrajania i wieloma innymi czynnikami. PPO próbuje obliczyć aktualizację w każdym kroku czasowym która minimalizuje funkcję straty jednocześnie upewniając się, że odchylenie od polityki nie będzie zbyt duże. Ten algorytm nie używa niektórych metod używanych przez TRPO, jak np. dywergencji Kullbacka-Leibera, co sprawia dotykowo, że PPO jest mniej złożony obliczeniowo.

### Algorytm SAC

Soft actor-critic to algorytm, który wyróżnia się tym, że oprócz zmaksymalizowania nagród, próbuje też zmaksymalizować tak zwaną entropię polityki. O entropii można myśleć jako o wskaźniku, jak bardzo nieprzewidywalna jest dana zmienna. Jeżeli zmienna jest stała, ten wskaźnik wynosi zero, ponieważ ta zmienna jest przewidywalna. Jeżeli natomiast dana zmienna może być losową liczbą zawierającą się w liczbach rzeczywistych, wtedy wskaźnik entropii jest wysoki, ponieważ ta liczba jest nieprzewidywalna. Im wyższa entropia polityki, tym bardziej algorytm zachęcany jest do eksploracji zamiast eksploatacji. Dzięki temu, proces uczenia może zostać przyśpieszony, ponieważ algorytm częściej odwiedza nowe stany i ulepsza swoją wiedzę. Dodatkowo, takie podejście może również zapobiec przedwczesnej konwergencji polityki do złego optimum lokalnego [23].

SAC jest algorytmem poza polityką, dzięki czemu może uczyć się z doświadczeń zdobytych w przeszłości. Podobnie do DQN, algorytm SAC przechowuje te dane w tabeli o stałym rozmiarze, a następnie próbkuje losowe z nich aby się ulepszyć. Dzięki temu, SAC wymaga od 5 do 10 razy mniej próbek do nauki w porównaniu do PPO, aby osiągnąć te same lub podobne wyniki. Minusem SAC jest jednak to, że wymaga on więcej aktualizacji modelu. Algorytm SAC jest dobrym wyborem, gdy środowisko w którym znajduje się agent, jest wolniejsze, czyli pojedynczy krok czasowy zajmuje około 0.1 sekundy lub więcej.

## Metody używane w uczeniu przez wzmacnianie

Czasami samo uczenie przez wzmacnianie nie wystarcza, ponieważ środowiska mogą być zbyt skomplikowane. Aby poradzić sobie z tym problemem, naukowcy stworzyli metody, które w pewien sposób pozwalają pokazać agentowi, w jaki sposób ma zachowywać się w danym środowisku. Metoda ta to uczenie przez naśladowanie.

### Uczenie przez naśladowanie

Czasami lepszym podejściem metodą prób i błędów jest zademonstrowanie zachowania, jakiego oczekujemy od agenta. Przykładowo, zamiast pośrednio szkolić samochód za pomocą funkcji nagrody, możemy dać mu rzeczywiste przykłady obserwacji z toru i działań z klawiatury. Uczenie przez naśladowanie wykorzystuje pary obserwacji i działań z demonstracji aby wyuczyć się polityki. Uczenie przez naśladowanie może być stosowane samodzielnie lub w połączeniu z uczeniem przez wzmacnianie, co powinno dać jeszcze lepsze efekty. Stosowany samodzielnie może zapewnić mechanizm uczenia się określonego typu czy też stylu zachowania. W połączeniu z uczeniem poprzez wzmacnianie może radykalnie skrócić czas potrzebny do agentowi do nauki środowiska, co pokazano na rysunku 2.

Unity ML-Agents pozwala na nagranie demonstracji pokazanie przez programistę zachowania oczekiwanego od agenta, które z kolei agent będzie starał się naśladować. Warto tutaj zaznaczyć, że demonstracje zaprezentowane przez programistę nie muszą być optymalne. To zadaniem agenta jest znalezienie takiego podejścia, które da mu największą nagrodę.

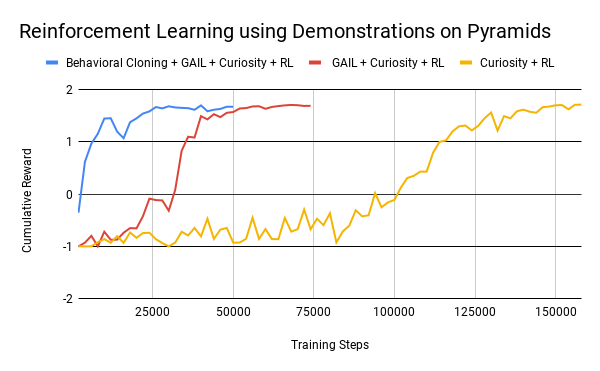
### Generative Adversarial Imitation Learning

Generative Adversarial Imitation Learning - GAIL) nagradza agenta za zachowanie podobne do zaprezentowanego. GAIL może być używany z nagrodami środowiskowymi lub bez nich i działa dobrze, gdy liczba demonstracji jest ograniczona. W tej metodzie, sieć neuronowa zwana dyskryminatorem jest uczona w celu rozróżnienia, czy obserwacja/działanie pochodzi z demonstracji, czy też jest wytworzone przez sieć neuronową agent. Dyskryminator może następnie przekazać agentowi nagrodę w oparciu o to, jak blisko ta nowa obserwacja/działanie znajduje się w jego przekonaniu od przedstawionych demonstracji.

Na każdym etapie szkolenia agent stara się nauczyć jak zmaksymalizować tę nagrodę i jednocześnie dyskryminator jest szkolony aby coraz lepiej odróżniać demonstracje od stanu/działań agenta. W ten sposób, podczas gdy agent staje się coraz lepszy w naśladowaniu demonstracji, dyskryminator staje się coraz bardziej rygorystyczny a agent musi bardziej starać się go „oszukać”. Dzięki temu GAIL uczy polityki, która wytwarza stany i działania podobne do demonstracji.

### Klonowanie Behawioralne

Klonowanie Behawioralne (Behavioral Cloning – BC) szkoli politykę agenta aby dokładnie naśladować działania pokazane w zestawie demonstracji. Ponieważ klonowanie behawioralne nie może uogólniać poza przykłady pokazane w demonstracjach, działa ono najlepiej, gdy istnieją demonstracje dla prawie wszystkich stanów, których może doświadczyć agent, lub w połączeniu z GAIL i/lub zewnętrzną nagrodą.



Rysunek 2. Porównanie szybkości uczenia algorytmu uczenia przez wzmacnianie z wykorzystaniem różnych kombinacji metod w środowisku „Piramidy”

# Trenowanie agenta we własnej instancji środowiska

## Instalacja i konfiguracja

### Unity

### ML-Agents

## Stworzenie instancji środowiska i agenta

## Trenowanie agenta

## Próby

## Wnioski

# Podsumowanie

Podsumowanie jest rozdziałem nienumerowanym – należy w nim jeszcze raz (podobnie, jak we wstępie) krótko opisać, co zostało zrobione w pracy i czy jej cel został osiągnięty. Można tez dodać ew. plany na przyszłość – dalszego rozwoju części praktycznej, testów itp.

# Bibliografia

1. Autor, *Tytuł książki*, Wydawnictwo, 2016
2. Autor (jeśli znany), *Tytuł strony internetowej*, http://pełen.url./strony, (dostęp: 12.12.2017)
3. Terri Williams, *Reinforcement Learning Vs. Deep Reinforcement Learning: What’s the Difference?* [https://www.techopedia.com/reinforcement-learning-vs-deep-reinforcement-learning-whats-the-difference/2/34039#](https://www.techopedia.com/reinforcement-learning-vs-deep-reinforcement-learning-whats-the-difference/2/34039%23), (Dostęp: 03. 04. 2022)
4. Chris Mahoney,  *Reinforcement Learning,* [https://towardsdatascience.com/reinforcement-learning-fda8ff535bb6#5554](https://towardsdatascience.com/reinforcement-learning-fda8ff535bb6%235554%20)(Dostęp: 03. 04. 2022)
5. Sutton, Barto, *Reinforcement Learning: An Introduction*, MIT Press, 2018
6. <https://kcir.pwr.edu.pl/~witold/ai/ai_markov_s.pdf>
7. <https://smartlabai.medium.com/reinforcement-learning-algorithms-an-intuitive-overview-904e2dff5bbc>
8. <https://create.unity.com/2021-game-report>
9. <https://stackify.com/best-csharp-tools/#:~:text=Visual%20Studio%20Code&text=Visual%20Studio%20Code%20is%20the,for%20C%23%20IntelliSense%20and%20debugging>.
10. <https://pypl.github.io/PYPL.html>
11. <https://www.cs.cmu.edu/~katef/DeepRLFall2018/lecture2_mdps.pdf>
12. <https://wazniak.mimuw.edu.pl/index.php?title=Sztuczna_inteligencja/SI_Modu%C5%82_13_-_Uczenie_si%C4%99_ze_wzmocnieniem#eq_bellman.q.2A>
13. <https://towardsdatascience.com/deep-q-network-dqn-ii-b6bf911b6b2c>
14. <https://www.reddit.com/r/MachineLearning/comments/bfny3m/d_deep_q_learning_for_continuous_action_space/>
15. <file:///C:/Users/Adrian/OneDrive%20-%20Uniwersytet%20Marii%20Curie-Sk%C5%82odowskiej/Pulpit/LICENCJAT/2.%20kwerenda%20literaturowa/%C5%BAr%C3%B3d%C5%82a/11.%20An%20Introduction%20to%20Deep%20learning.pdf>
16. <https://ai.stackexchange.com/questions/6196/what-is-the-relation-between-q-learning-and-policy-gradients-methods>
17. <file:///C:/Users/Adrian/OneDrive%20-%20Uniwersytet%20Marii%20Curie-Sk%C5%82odowskiej/Pulpit/LICENCJAT/2.%20kwerenda%20literaturowa/%C5%BAr%C3%B3d%C5%82a/17.%20Deep%20Reinforcement%20Learning%20in%20Action%20(2020).pdf>
18. <file:///C:/Users/Adrian/OneDrive%20-%20Uniwersytet%20Marii%20Curie-Sk%C5%82odowskiej/Pulpit/LICENCJAT/2.%20kwerenda%20literaturowa/%C5%BAr%C3%B3d%C5%82a/20%20sanghi_nimish_deep_reinforcement_learning_with_python_with_p.pdf> (PPO KL)
19. [file:///C:/Users/Adrian/OneDrive%20-%20Uniwersytet%20Marii%20Curie-Sk%C5%82odowskiej/Pulpit/LICENCJAT/2.%20kwerenda%20literaturowa/%C5%BAr%C3%B3d%C5%82a/1.[KSI%C4%84%C5%BBKA%20OPIYWANA]%20Deep%20reinforcement%20learning%20in%20Unity.pdf](file:///C:/Users/Adrian/OneDrive%20-%20Uniwersytet%20Marii%20Curie-Sk%C5%82odowskiej/Pulpit/LICENCJAT/2.%20kwerenda%20literaturowa/%C5%BAr%C3%B3d%C5%82a/1.%5bKSI%C4%84%C5%BBKA%20OPIYWANA%5d%20Deep%20reinforcement%20learning%20in%20Unity.pdf)
20. <https://jonathan-hui.medium.com/rl-the-math-behind-trpo-ppo-d12f6c745f33>
21. <file:///C:/Users/Adrian/OneDrive%20-%20Uniwersytet%20Marii%20Curie-Sk%C5%82odowskiej/Pulpit/LICENCJAT/2.%20kwerenda%20literaturowa/%C5%BAr%C3%B3d%C5%82a/8.%20Imitation%20Learning%20with%20the%20Unity%20Machine%20Learning%20Agents%20Toolkit.pdf>
22. <https://towardsdatascience.com/policy-gradients-in-a-nutshell-8b72f9743c5d>
23. <https://spinningup.openai.com/en/latest/algorithms/sac.html>
24. Autor, *Tytuł książki*, Wydawnictwo, 2016

# Spis rysunków

[Rysunek 1. Ilustracja demonstracyjna. 11](file:///C:\Users\Adrian\Downloads\szablon%20pracy%20dyplomowej%20(1).docx#_Toc524265005)

# Spis tabel

[Tabela 1. Przykładowa tabelka . 11](file:///C:\Users\Adrian\Downloads\szablon%20pracy%20dyplomowej%20(1).docx#_Toc524265033)

# Streszczenie

Tu streszczenie pracy po polsku (ok. pól strony).

**Abstract**

Tu streszczenie pracy po angielsku (ok. pól strony).