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](#_Toc105332228)

[Rozdział 1. Uczenie przez wzmacnianie jako część uczenia maszynowego 5](#_Toc105332229)

[1.1. Uczenie maszynowe 5](#_Toc105332230)

[1.2. Uczenie przez wzmacnianie 5](#_Toc105332231)

[1.2.1. Podstawowe pojęcia w uczeniu przez wzmacnianie 6](#_Toc105332232)

[1.2.2. Historia uczenia przez wzmacnianie 7](#_Toc105332233)

[1.2.3. Głębokie uczenie przez wzmacnianie 9](#_Toc105332234)

[1.2.4. Przykłady wykorzystywania uczenia przez wzmacnianie 10](#_Toc105332235)

[Rozdział 2. Algorytmy uczenia przez wzmacnianie i wykorzystane technologie 11](#_Toc105332236)

[2.1. Podstawy uczenia przez wzmacnianie 11](#_Toc105332237)

[2.1.1. Proces decyzyjny Markowa i równanie Bellman’a 11](#_Toc105332238)

[2.1.2. Programowanie dynamiczne 14](#_Toc105332239)

[2.2. Przykładowe algorytmy uczenia przez wzmacnianie 15](#_Toc105332240)

[2.2.1. Algorytmoy Monte-Carlo 16](#_Toc105332241)

[2.2.2. Uczenie metodą różnic czasowych 16](#_Toc105332242)

[2.2.3. Q-Learnig 17](#_Toc105332243)

[2.3. Biblioteki, technologie i programy 20](#_Toc105332244)

[2.3.1. Unity 20](#_Toc105332245)

[2.3.2. ML-Agents 21](#_Toc105332246)

[2.3.3. Język C# 22](#_Toc105332247)

[2.3.4. Język Python 22](#_Toc105332248)

[2.3.5. PyTorch 23](#_Toc105332249)

[2.4. Algorytmy wykorzystywane przez ML-Agents 23](#_Toc105332250)

[2.4.3. Metody gradientu polityki 23](#_Toc105332251)

[2.4.4. Metoda aktora-krytyka 24](#_Toc105332252)

[2.4.5. Algorytm PPO 24](#_Toc105332253)

[2.4.6. Algorytm SAC 25](#_Toc105332254)

[2.5. Metody używane w uczeniu przez wzmacnianie 25](#_Toc105332255)

[2.5.1. Uczenie przez naśladowanie 26](#_Toc105332256)

[2.5.2. Generative Adversarial Imitation Learning 27](#_Toc105332257)

[2.5.3. Klonowanie Behawioralne 27](#_Toc105332258)

[Rozdział 3. Trenowanie agenta we własnej instancji środowiska 28](#_Toc105332259)

[3.1. Instalacja i konfiguracja 28](#_Toc105332260)

[3.1.1. Unity 28](#_Toc105332261)

[3.1.2. ML-Agents 28](#_Toc105332262)

[3.2. Stworzenie instancji środowiska i agenta 28](#_Toc105332263)

[3.3. Trenowanie agenta 28](#_Toc105332264)

[3.4. Próby 28](#_Toc105332265)

[3.5. Wnioski 28](#_Toc105332266)

[Podsumowanie 28](#_Toc105332267)

[Bibliografia 29](#_Toc105332268)

[Spis rysunków 31](#_Toc105332269)

[Spis tabel 33](#_Toc105332270)

[Streszczenie 35](#_Toc105332271)

# 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 tych powodów 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 [1]. W dzisiejszych czasach wyróżniamy trzy główne kategorie uczenia maszynowego w zależności od charakteru sygnału informacji zwrotnej dla systemu uczenia się [2].

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 [3].



Rysunek 1. Pętla wykorzystywana w uczeniu przez wzmacnianie

### Podstawowe pojęcia w uczeniu przez wzmacnianie

Uczenie przez wzmacnianie wykorzystuje głównie cztery elementy, 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 [6]. 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. Środowisko przyznaje nagrodę agentowi za jego zachowanie, a celem agenta jest zmaksymalizowanie tej liczby. 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 [6].

#### 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, wartości funkcji określają długoterminową atrakcyjność stanów po uwzględnieniu stanów które prawdopodobnie nastąpią, oraz nagród dostępnych w tych stanach [6]. 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 [6]. 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 [4, 5]. Drugim z nich jest nauka metodą prób i błędów zapoczątkowany w psychologii zwierząt [5]. Ostatni z wątków łączy w sobie dwa poprzednie - są to metody uczenia różnic czasowych [5]. 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 [4]. 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 [4, 5]. 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” [4]. Pomysł ten został rozszerzony w 1960 roku przez Howard ’a, który zdefiniował metodę iteracji polityki dla MDP. Równanie Bellmana ma pewną wadę, nazwaną przez jego twórcę „przekleństwem wymiarowości” [5]. Polega ona na tym, że wymagania obliczeniowe tego równania rosną wykładniczo wraz z liczbą zmiennych stanu [5]. 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 [5]. 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 [4]. W swoim doktoracie opisał on maszynę własnej konstrukcji o nazwie SANRC (Stochastic Neural-Analog Reinforcement Calculators) [5]. Maszyna ta wykorzystywała paradygmat nagrody aby przechodzić przez labirynt i była prawdopodobnie pierwszą sztuczną maszyną samouczącą się [5].

W kolejnych latach badania metody prób i błędów pod kątem obliczeniowym zostały uogólnione do rozpoznawania wzorów, a następnie przypisane jako część uczenia nadzorowanego [5]. 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 [4].

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) [6]. Minsky był prawdopodobnie pierwszym naukowcem który w 1954 roku połączył uczenie metodą różnic czasowych z systemami sztucznej inteligencji [4]. Z kolei Arthur Samuel w 1959 roku zaproponował i zaimplementował metodę nauczania zawierającą ideę uczenia różnic czasowych w swoim programie grającym w warcaby [5]. 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 [5]. Sutton rozwinął ideę Klopf ‘a i w swojej pracy powiązał ją z teorią uczenia się zwierząt [5]. Dzięki temu opisał on zasady, według których uczenie się jest napędzane przez zmiany w czasowo następujących po sobie przewidywaniach [5]. 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 [5]. 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 [7]. 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 [8]. 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 [9]. 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 sensorów są przekazywane do warstwy wejścia algorytmu a na wyjściu otrzymujemy ruch kierownicy [10]. 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 (MDP - Markov Decision Process), 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. Pomocne w rozwiązaniu MDP jest 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 [11]. 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 [11]. 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 [13]. 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 skumulowaną nagrodę oznaczamy jako sumę wszystkich otrzymanych nagród od stanu początkowego aż do stanu końcowego [5]:

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 [13]:

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 [5]. 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 [13]. 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 [14]. 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 [14]:

#### 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 [14]. 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 akcję która da maksymalną nagrodę jaką agent może dostać jeżeli znajduje się w stanie .

Polityka jest lepsza od polityki wtedy, gdy funkcja wartości wszystkich stanów jest większa od wartości wszystkich stanów polityki [14]. 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 [5]. 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 [15]. 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 [15]. Za każdym razem gdy zajmujemy się nowym problemem, sprawdzamy, czy został on już rozwiązany [15]. 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 wartości. Iteracja wartości stanu wykorzystuje równanie Bellmana i pozwala na iteracyjne coraz lepsze oszacowanie wartości danego stanu [5]. 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.

## Przykładowe algorytmy uczenia przez wzmacnianie

Istnieje wiele algorytmów uczenia przez wzmacnianie. Głównym podziałem tych algorytmów jest podział na algorytmy w polityce i poza polityką. Algorytmy w polityce (na przykład Q-learning) ulepszają politykę, którą aktualnie się kierują [16]. Algorytmy poza polityką (na przykład SARSA) kierują się jedną polityką do ulepszania innej [16].

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. Częstym podejściem mówiącym, kiedy algorytm ma eksplorować a kiedy eksploatować, jest tak zwane podejście -greedy [17]. Polega ono na tym, że wybieramy epsilon , i jeżeli losowa wartość jest większa od , wtedy algorytm podejmuje eksploatację, natomiast jeżeli ta wartość jest mniejsza, wtedy algorytm eksploruje [17].

### 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 [5]. 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 [18]. 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 [19]. Zainspirowany tym eksperymentem algorytm uczenia metodą różnic czasowych działa podobnie – zakłada on, że akcje podjęte obecnie wiążą się z nagrodami otrzymanymi w przyszłości [18].

Aby uaktualnić wartość stanu w tym algorytmie korzystamy ze wzoru:

Gdzie:

**•**  to aktualna wartość stanu w kroku czasowym

**•**  to wartość przewidywana dla stanu ,

Odjęcie aktualnej wartości stanu od przewidywanej wartości stanu da nam błąd różnicy czasowej oznaczający, jak bardzo wartość przewidywana różni się od wartości aktualnej [5].



*Rysunek 3. Eksperyment psa Pawłowa [19]*

### Q-Learnig

Jak wspomniano w rozdziale 1.2.2. Q-learning został stworzony w 1989 roku przez Christopher’a Watkins ‘a. Q-learning polega na przypisaniu w tabeli wartości każdej możliwej akcji w danym stanie, dzięki czemu algorytm może po prostu podejmować akcje które mają najwyższe wartości. Wzór wyliczenia Q-wartości w Q-learningu korzysta z równania Bellmana i jest podobny do wzoru wykorzystywanego w uczeniu metodą różnic czasowych, a wygląda następująco [5, 18]:

Oznacza ono, że Q-wartość akcji a w stanie s jest równa nagrodzie jaką agent otrzymał za znalezienie się w stanie plus maksymalna przyszła nagroda w stanie . Aby zaktualizować Q-wartość w algorytmie Q-learning’u wykorzystuje się następujący wzór:

Gdzie:

**•**  to nowa wartość pary w kroku czasowym

**•**  to aktualna wartość pary w kroku czasowym ,

**•**  to szybkość uczenia,

**•**  to maksymalna przyszła nagroda w stanie

#### 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 na aktualizację Q-wartości jest prawie taki sam jak ten używany w Q-learningu (2.13), jednak jest w nim pewna różnica. Wyliczając wspomnianą wcześniej wartość przewidywaną dla pary bierzemy pod uwagę Q-wartość następnego stanu i następnej akcji co oznacza, że akcja została już podjęta (zazwyczaj za pomocą podejścia ) pomimo tego, że algorytm jeszcze faktycznie nie znalazł się w następnym stanie [5, 18, 20]. Z kolei Q-learning bierze pod uwagę następny stan i akcję, która zapewni maksymalną Q-wartość co oznacza, że podejmuje akcję na podstawie optymalnej polityki [5, 18, 20].

#### DQN

Jak wspomniano w rozdziale 1.2.3, 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 zapamiętywać 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: Dwóch sieci neuronowych zwanych siecią docelową i siecią główną oraz z bufora powtórek [21-23]. 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. Z kolei użycie sieci docelowej o takiej samej architekturze, lecz różnych wagach zapewnia stabilność sieci podczas nauki [21].

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 2.2 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 wyuczyć się stochastycznych polityk, ponieważ podczas wybierania akcji , wybieramy akcję która da nam największą Q-wartość [24].

## Biblioteki, technologie i programy

### 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 forest” 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 [25].

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

**• 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 B ma swoje własne właściwości, a zaznaczenie jednego z nich pokaże je w oknie F zwanym Inspektorem. Właściwości obiektu można dodawać, usuwać oraz edytować. Każdy obiekt w oknie B ma własne właściwości, w związku z tym, okno F 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.



*Rysunek 2: Interfejs Unity[26].*

### 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 [27]. Ten zestaw narzędzi zapewnia implementację dwóch algorytmów uczenia przez wzmacnianie (PPO i SAC), dzięki czemu programiści mogą z łatwością tworzyć i trenować agentów w stworzonych przez siebie środowiskach [27]. 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 2.5.1. co dodatkowo wzbogaca zakres możliwości tego projektu.

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

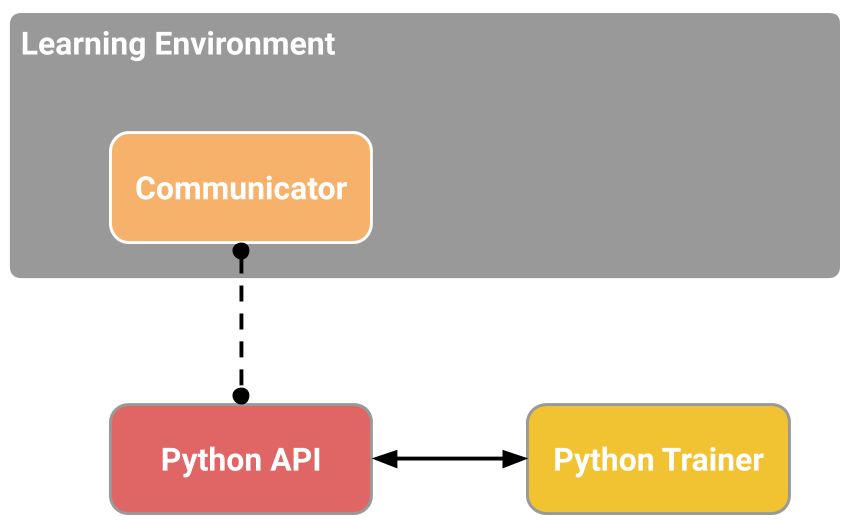
**• Środowisko** – Scena Unity zawierające wszystkie obiekty. W środowisku agent może zbierać obserwacje, wykonywać akcje i się uczyć.

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

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

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

**• Gym wrapper** – Jednym ze sposobów w jaki badacze mogą wchodzić w interakcje z symulowanymi środowiskami jest stworzone przez OpenAI oprogramowanie Gym. ML-Agents dostarcza Gym wrapper, który pozwala na wykorzystanie Gym do interakcji ze środowiskiem stworzonym w Unity [28].



*Rysunek 3. Uproszczony schemat ML-Agents[28].*

### 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 [29] 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 we wczesnych latach 90 przez Guido Van Rossum’a język posiada mnóstwo bibliotek pozwalających na wykonywanie najróżniejszych operacji co czyni go najpopularniejszym językiem na świecie [30, 31]. 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) [28]. Oba te algorytmy korzystają między innymi z metody gradientu polityki oraz metody aktora-krytyka.

### 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 [12]. Zakładając że sekwencja stanów, akcji i nagród w środowisku oznaczona zostanie jako a funkcja będzie oznaczać sumę nagród z danej sekwencji, otrzymamy wzór na cel agenta zapisany wzorem matematycznym [33, 34, 37]:

Co oznacza, że funkcja J() jest równa wartości oczekiwanej nagrody z sumy nagród sekwencji 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 [33, 34, 37]. 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 [36].

### 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 [32, 33, 37]. Druga to krytyk, który ocenia działania podjęte przez aktora na podstawie polityki [32, 33, 37]. Uczenie metodą aktora-krytyka to uczenie 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.

Dodatkowo, opisywana metoda korzysta z konceptu entropii. O entropii można myśleć jako o wskaźniku, jak bardzo nieprzewidywalna jest dana zmienna [38, 40]. 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[39, 40].

### 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 różnych mechanik, takich jak region zaufania, czy dywergencja Kullbacka-Leiblera [33 - 35]. 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 [33 - 35]. Region zaufania jest techniką w której algorytm wybiera punkt na wykresie wokół którego tworzy region zaufania i w tym regionie wybierany jest najniższy punkt [33 - 35]. Całość powtarzana jest aż do znalezienia najniższego punktu. Dodatkowo, aby lepiej kontrolować szybkość uczenia, możemy w czasie działanie rozszerzać lub kurczyć ten region [39].

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 [33 – 35]. 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 hiperparametrach, oraz jest bardzo wymagający obliczeniowo.

Odpowiedzią na te problemy jest PPO, który ma na celu znalezienie równowagi między łatwością implementacji, złożonością próbek i łatwością dostrajania hiperparametrów [42]. Ten algorytm nie używa niektórych metod używanych przez TRPO, jak np. dywergencji Kullbacka-Leibera, co sprawia dotykowo, że PPO jest mniej złożony obliczeniowo [33 - 35]. Proximal Policy optimization stał się popularny, gdy OpenAI opublikowało program, który korzystając z tego algorytmu, został wytrenowany do gry w „Dota 2” i pokonał najlepszych graczy na świecie [9].

### Algorytm SAC

Soft actor-critic to algorytm, który wyróżnia się tym, że oprócz zmaksymalizowania nagród, próbuje też zmaksymalizować entropię polityki opisaną w rozdziale 2.4.4. 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 [40].

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 [28]. Minusem SAC jest jednak to, że wymaga on więcej aktualizacji modelu [28]. 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 [28].

## Metody używane w uczeniu przez wzmacnianie

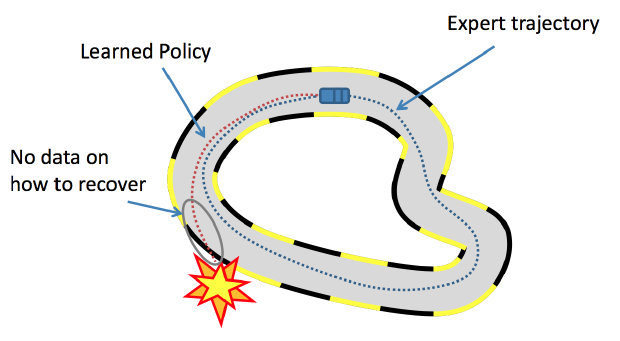
Czasami samo uczenie przez wzmacnianie nie wystarcza, ponieważ środowiska mogą być zbyt skomplikowane. Aby poradzić sobie z tym problemem, naukowcy stworzyli metody, które w pewien sposób pozwalają pokazać agentowi, w jaki sposób ma zachowywać się w danym środowisku. Metoda ta to uczenie przez naśladowanie.

### Uczenie przez naśladowanie

W sytuacjach, gdy środowisko w jakim znajduje się agent jest skomplikowane, korzystanie z metody prób i błędów może być mało skuteczne. Rozwiązaniem tego problemu jest zademonstrowanie zachowania, jakiego oczekujemy od agenta. Przykładowo, zamiast pośrednio szkolić samochód w symulowanym środowisku 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 [28]. Dodatkowo, uczenie przez naśladowanie może być stosowane samodzielnie lub w połączeniu z uczeniem przez wzmacnianie, co powinno dać jeszcze lepsze efekty [28]. Stosowany samodzielnie może zapewnić mechanizm uczenia się określonego typu zachowania. W połączeniu z uczeniem poprzez wzmacnianie może radykalnie skrócić czas potrzebny do agentowi do nauki środowiska, co pokazano na rysunku 2 [28].

Unity ML-Agents pozwala na nagranie demonstracji, czyli 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.

Korzystając z uczenia przez naśladowanie warto pamiętać, że gdy agent podejmie niewłaściwy krok, (co pokazano czerwoną linią na rysunku 5) względem zademonstrowanego toru jazdy (czyli niebieskiej linii na rysunku 5) agent nie będzie w stanie nagle powrócić na właściwy tor w następnym kroku czasowym, tylko będzie musiał podjąć zestaw akcji, aby to zrobić. Przez strukturę MDP błędy popełnione w kolejnych krokach czasowych nakładają się na siebie, co może skutkować niepożądanym zachowaniem [41].



*Rysunek 4: Przykładowa demonstracja toru jazdy dla samochodu (niebieska linia) i próba odtworzenia jej przez agenta (czerwona linia)[41].*

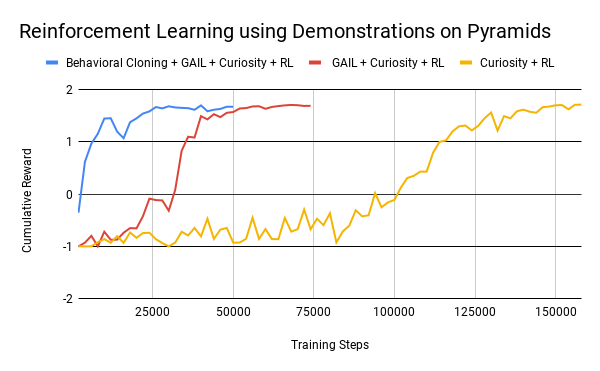
### Generative Adversarial Imitation Learning

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 [28]. 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 [28]. Dyskryminator może następnie przekazać agentowi nagrodę w oparciu o to, jak blisko ta nowa obserwacja/działanie znajduje się w jego przekonaniu od przedstawionych demonstracji [28].

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” [28]. 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, co widać na rysunku 2 [28].



Rysunek 5. Porównanie szybkości uczenia algorytmu uczenia przez wzmacnianie z wykorzystaniem różnych kombinacji metod w środowisku „Piramidy”[28].

# Trenowanie agenta we własnej instancji środowiska

## Instalacja i konfiguracja

Aby rozpocząć trenowanie agenta, należy wcześniej pobrać i zainstalować kilka programów i pakietów. Podstawowymi są oczywiście Unity oraz ML-Agents. Dodatkowo, należy również pobrać Python, stworzyć za jego pomocą wirtualne środowisko i zainstalować w nim odpowiednie pakiety.

### Unity i ML-Agents

Aby zainstalować Unity, należy wejść na stronę <https://unity3d.com/get-unity/download>, kliknąć przycisk pobrania Unity Hub i zainstalować to oprogramowanie. Następnie w oknie Unity Hub należy zalogować się na swoje konto Unity, oraz pobrać edytor. W tej pracy użyto edytora w wersji 2020.30.f1.

W następnej kolejności należy pobrać repozytorium ML-Agents ze strony <https://github.com/Unity-Technologies/ml-agents> i za pomocą Unity Hub otworzyć znajdujący się w tym repozytorium projekt. Po włączeniu edytora, ten wyświetli błędy dotyczące nierozpoznawalnych nazw w kodzie. Aby to naprawić, włączyć Package Manager i zaimportować w nim pakiety ML-Agents oraz ML-Agents extensions. Opcjonalnie można zaimportować paczkę Visual Studio, aby ten edytor kodu poprawnie wykrywał nazwy, które zostaną dodane przez wspomniane paczki.

### Python

Do trenowania agenta wymagane jest stworzenie wirtualnego środowiska, które można stworzyć za pomocą pythona. W pierwszej kolejności należy go pobrać ze strony <https://www.python.org/downloads/> a następnie stworzyć wspomniane środowisko, najlepiej w pobranym repozytorium ML-Agents. Kolejnym krokiem jest pobranie dwóch odpowiednich pakietów:

**• torch** W pracy użyto Torch w wersji 1.7.0

**• mlagents** W pracy użyto mlagents w wersji 0.29.0

### CUDA

Opcjonalnym krokiem, jaki można wykonać jest pobranie CUDA oraz cuDNN, aby do obliczeń podczas trenowania agenta została częściowo wykorzystana karta graficzna. Aby pobrać CUDA należy wejść na stronę <https://developer.nvidia.com/cuda-toolkit-archive>, w pracy użyto CUDA w wersji 10.1. Z kolei aby pobrać cuDNN należy wejść na stronę <https://developer.nvidia.com/rdp/cudnn-archive>. W pracy wykorzystano cuDNN w wersji 7.6.4.

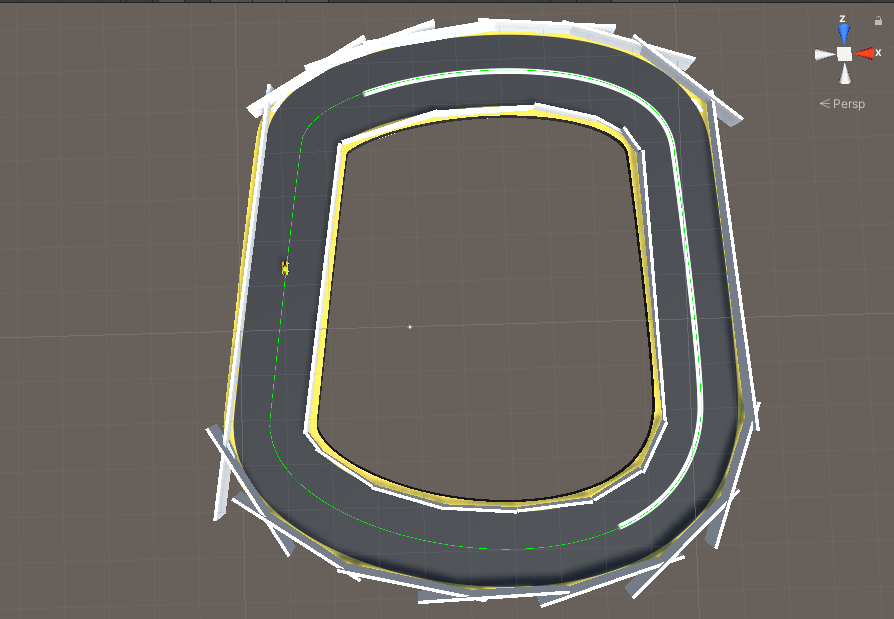
## Stworzenie instancji środowiska i agenta

Aby zastosować ML-Agents i zaimplementowany przez ten zestaw narzędzi algorytm PPO, należy przygotować środowisko, oraz stworzyć agenta. Środowiskiem będzie droga ze ścianami przy krawędziach, natomiast agentem będzie samochód którego zadaniem będzie jazda po drodze i unikanie ścian.

### Stworzenie środowiska

Aby stworzyć środowisko, zaimportowano darmowy pakiet „Bézier Path Creator”, który pozwala na proste stworzenie dróg, po których samochód będzie mógł jeździć [43]. Po stworzeniu pustego obiektu w hierarchii obiektów, należy dodać do niego komponent „Path Creator”. W oknie sceny powinny pojawić się dwie czerwone kule połączone zieloną linią. To jest ścieżka, która posłuży do stworzenia drogi. Wciskając kombinację shift+lewy przycisk myszy na zielonej linii można dodawać punkty zagięcia linii i ustawiać ich pozycję, natomiast za pomocą niebieskich kul można manipulować kątami nachylenia tej linii. Aby stworzyć zamknięty obwód należy zaznaczyć w inspektorze opcję „Closed Path”. Na koniec, aby stworzyć drogę do obiektu zawierającego ścieżkę dodano komponent „Mesh Road Creator”, oraz dodać do nowo powstałego komponentu mesh collider, aby samochód nie przenikał przez drogę.

Kolejnym krokiem, jaki należy wykonać jest ustawienie ścian wzdłuż krawędzi drogi, aby samochód nie wyjeżdżał poza ustalony obszar. Aby to zrobić, stworzono nowy obiekt kostki i wykorzystano go do stworzenia barierek przy drodze. Aby rozróżnić ten obiekt od pozostałych, dodano do niego nowy skrypt o nazwie „Railing”. Gotowe środowisko pokazano na rysunku 6.



*Rysunek 6: Gotowe środowisko Unity.*

### Stworzenie agenta

Tworzenie agenta rozpoczęto od zaimportowania pakietu „ARCADE: FREE Racing Car”, aby nadać mu wygląd samochodu [44]. Następnie w hierarchii stworzono obiekt „Car”, który przechowuje w sobie wszystkie obiekty, które zostaną wymienione w tym podrozdziale. Do obiektu „Car” dodano komponent „Box Collider”, aby móc przyczepić do niego koła, oraz komponent „Rigidbody”, aby nadać samochodowi masę oraz sprawić, że grawitacja będzia miała na niego wpływ. Podkomponentami obiektu „Car” są „Body” i „Wheels”. „Body” zawiera w sobie komponent „Mesh Renderer” w którym użyto modelu samochodu ze wcześniej zaimportowanego pakietu. „Wheels” zawiera w sobie obiekt „Meshess”, który z kolei zawiera w sobie cztery komponenty „Wheel”, po jednym dla każdego koła w pojeździe. Dodatkowo do obiektów „Wheel” dodano komponenty „WheelCollider” który sprawia, że koło imituje zachowanie prawdziwego.

#### Kod agenta

Po umieszczeniu agenta na pozycji startowej, stworzono i dodano do samochodu skrypt CarScript, który pozwala na jego sterowanie, odbieranie danych i system nagród. Listing 1 pokazuje klasę którą użyto do stworzenia osi samochodu, oraz początek klasy CarScript. Każda oś ma obiekty typu WheelCollider, oraz dwie zmienne typu bool oznaczające, czy koła mogą się obracać oraz czy mogą skręcać. Klasa CarSctipt dziedziczy po klasie Agent, która jest klasą dostarczaną przez ML-Agents i dzięki której obiekt może się uczyć. CarScript zawiera liczni sekund kolizji ze ścianą, listę osi, maksymalną prędkość i kąr skręcania oraz począrkową pozycję.



Listing 2 pokazuje metodę, która odpowiada za ruch pojazdu. Jest ona wykonywana w podczas każdej klatki, ponieważ agent cały czas dostaje nowe instrukcje od algorytmu. Po otrzymaniu instrukcji w argumencie actions, metoda sprawdza akcję prędkości wybrana przez algorytm PPO czyli do przodu, do tyłu lub nic, a także akcję skrętu w lewo, prawo lub wcale. Następnie nowe zostają zastosowane dla każdego koła w samochodzie.



Kolejną metodą agenta jest metoda OnCollisionStay, która wykrywa kolizje między samochodem a ścianą, i jeżeli samochód zderza się z nią, dostaje ujemną nagrodę, a jeżeli robi to przez dwie sekundy, dodatkowo następuje zakończenie epizodu, co skutkuje wywołaniem funkcji OnEpisodeBegin. Metoda ta resetuje pozycje agenta i zeruje mu prędkość (listing 4).



#### Komponenty agenta

Aby agent mógł się uczyć, oprócz kodu potrzebuje on dodatkowo trzech komponentów które dostarczane są przez pakiet ML-Agents:

* Behaviour parameters – Komponent określający nazwę zachowania agenta, rozmiar wektora obserwacji, jakie agent może zbierać, rodzaj i ilość akcji jakie agent może podjąć, wytrenowany model oraz kilka innych funkcji.
* Decision requester – Komponent, który automatycznie żąda decyzji dla agenta w reguralnych odstępach czasu.
* Ray perception sensor – Komponent który wysyła promienie odpowiadające za obserwacje agenta w środowisku. Dzięki zawartym w tym komponencie opcjom, właściwości promieni takie jak ich długość czy ilość można dostosować według potrzeb.

## 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. *Czym jest uczenie maszynowe?*, <https://www.sap.com/poland/insights/what-is-machine-learning.html>, (Dostęp 08.06.2022)
2. *Machine learning*, <https://en.wikipedia.org/wiki/Machine_learning>, (Dostęp 08.06.2022)
3. *Analityka danych i uczenie maszynowe*, <https://www.ibm.com/pl-pl/analytics/machine-learning> (Dostęp 08.06.2022)
4. Chris Mahoney,  *Reinforcement Learning,* <https://towardsdatascience.com/reinforcement-learning-fda8ff535bb6> (Dostęp 08.06.2022)
5. Sutton, Barto, *Reinforcement Learning: An Introduction*, MIT Press, 2018
6. Warunkowanie sprawcze, <https://openstax.org/books/psychologia-polska/pages/6-3-warunkowanie-sprawcze> (Dostęp 08.06.2022)
7. 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)
8. *AlphaGo*, <https://www.deepmind.com/research/highlighted-research/alphago> (Dostęp 08.06.2022)
9. Volodymyr Mnih, Koray Kavukcuoglu i inni, *Playing Atari with Deep Reinforcement Learning,* <https://arxiv.org/pdf/1312.5602v1.pdf> (Dostęp 08.06.2022)
10. Behzad Benam, *Applying of Reinforcement Learning for Self-Driving Cars*, <https://towardsdatascience.com/applying-of-reinforcement-learning-for-self-driving-cars-8fd87b255b81> (Dostęp 08.06.2022)
11. *Podejmowanie złożonych decyzji — sekwencyjne problemy decyzyjne*, <https://kcir.pwr.edu.pl/~witold/ai/ai_markov_s.pdf> (Dostęp 08.06.2022)
12. SmartLab AI, *Reinforcement Learning algorithms — an intuitive overview*, <https://smartlabai.medium.com/reinforcement-learning-algorithms-an-intuitive-overview-904e2dff5bbc> (Dostęp 08.06.2022)
13. Rohan Jagtap, Understanding Markov Decision Process (MDP), <https://towardsdatascience.com/understanding-the-markov-decision-process-mdp-8f838510f150> (Dostęp 08.06.2022)
14. Blackburn, *Reinforcement Learning: Bellman Equation and Optimality (Part 2)*, <https://towardsdatascience.com/reinforcement-learning-markov-decision-process-part-2-96837c936ec3#:~:text=Bellman%20Expectation%20Equation,-A%20quick%20review&text=From%20the%20above%20equation%2C%20we,stands%20for%20Bellman%20Expectation%20Equation> (Dostęp 08.06.2022)
15. Steve Brunton, *Model Based Reinforcement Learning: Policy Iteration, Value Iteration, and Dynamic Programming*, <https://youtu.be/sJIFUTITfBc> (Dostęp 08.06.2022)
16. Ram Sagar, *On-Policy VS Off-Policy Reinforcement Learning*, <https://analyticsindiamag.com/reinforcement-learning-policy/#:~:text=On%2Dpolicy%20methods%20attempt%20to,used%20to%20generate%20the%20data>. (Dostęp 08.06.2022)
17. Samishawl, *Epsilon-Greedy Algorithm in Reinforcement Learning*, <https://www.geeksforgeeks.org/epsilon-greedy-algorithm-in-reinforcement-learning/> (Dostęp 08.06.2022)
18. Steve Brunton, *Q-Learning: Model Free Reinforcement Learning and Temporal Difference Learning*, <https://youtu.be/0iqz4tcKN58> (Dostęp 08.06.2022)
19. *Classical conditioning*, <https://en.wikipedia.org/wiki/Classical_conditioning> (Dostęp 08.06.2022)
20. *What is the difference between Q-learning and SARSA*?, <https://stackoverflow.com/questions/6848828/what-is-the-difference-between-q-learning-and-sarsa> (Dostęp 08.06.2022)
21. Takuma Seno, *Welcome to Deep Reinforcement Learning Part 1 : DQN*, <https://towardsdatascience.com/welcome-to-deep-reinforcement-learning-part-1-dqn-c3cab4d41b6b> (Dostęp 08.06.2022)
22. Jordi TORRES.AI , *Deep Q-Network (DQN)-II*, <https://towardsdatascience.com/deep-q-network-dqn-ii-b6bf911b6b2c> (Dostęp 08.06.2022)
23. Vincent François-Lavet, Peter Henderson i inni, *An Introduction to Deep Reinforcement Learning,* now publishers, 2018
24. BCS Member Groups , *2) Deep Q Network DQN*, <https://youtu.be/By6TYFSIFVE> (Dostęp 08.06.2022)
25. *2021 Gaming Report*, <https://create.unity.com/2021-game-report> (Dostęp 08.06.2022)
26. *Unity’s interface*, <https://docs.unity3d.com/Manual/UsingTheEditor.html> (Dostęp 08.06.2022)
27. *Unity ML-Agents Toolkit*, <https://github.com/Unity-Technologies/ml-agents> (Dostęp 08.06.2022)
28. *ML-Agents Toolkit Overview*, <https://github.com/Unity-Technologies/ml-agents/blob/release_19_docs/docs/ML-Agents-Overview.md> (Dostęp 08.06.2022)
29. <https://stackify.com/best-csharp-tools/#:~:text=Visual%20Studio%20Code&text=Visual%20Studio%20Code%20is%20the,for%20C%23%20IntelliSense%20and%20debugging> (Dostęp 08.06.2022)
30. Charris Herrera, *The Ultimate List of C# Tools: IDEs, Profilers, Automation Tools, and More*, <https://en.wikipedia.org/wiki/Python_(programming_language)> (Dostęp 08.06.2022)
31. PYPL PopularitY of Programming Language, <https://pypl.github.io/PYPL.html> (Dostęp 08.06.2022)
32. Brandon Brown, Alexander Zai, *Deep Reinforcement Learning in Action,* Manning Publications, 2020
33. Nimish Sanghi, *Deep Reinforcement Learning with Python With PyTorch, TensorFlow and OpenAI Gym*, APRESS, 2021
34. Abhilash Majumder, Deep Reinforcement Learning in Unity With Unity ML Toolkit, APRESS, 2021
35. Jonathan Hui , *RL — The Math behind TRPO & PPO*, <https://jonathan-hui.medium.com/rl-the-math-behind-trpo-ppo-d12f6c745f33> (Dostęp 08.06.2022)
36. Mario Rubak , *Imitation Learning with the Unity Machine Learning Agents Toolkit,* FH Campus Wien, 2021
37. Sanyam Kapoor , *Policy Gradients in a Nutshell*, <https://towardsdatascience.com/policy-gradients-in-a-nutshell-8b72f9743c5d> (Dostęp 08.06.2022)
38. *Soft Actor-Critic*, <https://spinningup.openai.com/en/latest/algorithms/sac.html> (Dostęp 08.06.2022)
39. Jonathan Hui, *RL — Trust Region Policy Optimization (TRPO) Explained*, <https://jonathan-hui.medium.com/rl-trust-region-policy-optimization-trpo-explained-a6ee04eeeee9> (Dostęp 08.06.2022)
40. Vaishak V.Kumar , *Soft Actor-Critic Demystified*, <https://towardsdatascience.com/soft-actor-critic-demystified-b8427df61665> (Dostęp 08.06.2022)
41. SmartLab AI, *A brief overview of Imitation Learning*, <https://smartlabai.medium.com/a-brief-overview-of-imitation-learning-8a8a75c44a9c> (Dostęp 08.06.2022)
42. John Schulman, Oleg Klimov i inni, *Proximal Policy Optimization*, <https://openai.com/blog/openai-baselines-ppo/> (Dostęp 08.06.2022)
43. Sebastian Lague, *Bézier Path Creator*, <https://assetstore.unity.com/packages/tools/utilities/b-zier-path-creator-136082#description> (Dostęp 08.06.2022)
44. Mena , *ARCADE: FREE Racing Car*, <https://assetstore.unity.com/packages/3d/vehicles/land/arcade-free-racing-car-161085#description> (Dostęp 08.06.2022)
45. *Create a vehicle with Wheel Colliders*, <https://docs.unity3d.com/Manual/WheelColliderTutorial.html> (Dostęp 08.06.2022)
46. Code Monkey, *Simple Checkpoint System in Unity*, <https://youtu.be/IOYNg6v9sfc> (Dostęp 08.06.2022)

# 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).