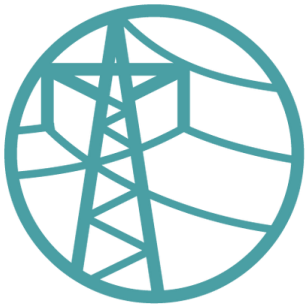


WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Karol Pękala

Zastosowanie deep reinforcement learning w silniku Unreal Engine

Projekt Inżynierski

Opiekun pracy:

dr hab. inż. Roman Zajdel

Rzeszów, 2023

Podziękowania

Składam serdeczne podziękowania

opiekunowi projektu, Panu Dr hab. inż. Romanowi Zajdelowi, prof. PRz, za cierpliwość, poświęcony czas i cenną wiedzę, życzliwie mi przekazaną,

oraz mojej żonie Nikoli, za motywację i wsparcie w trudnych chwilach.

Spis treści

1. Wstęp 5
2. Sieci neuronowe 7
2.1. Algorytmy głębokiego uczenia się 8
2.2. Uczenie się ze wzmocnieniem 10
2.3. Algorytmy DRL 12
3. Programowanie gier 14
3.1. Sztuczna Inteligencja w grach komputerowych 15
3.2. Platformy i narzędzia 17
3.3. Unreal Engine 17
3.4. MindMaker 18
4. Opis aplikacji 21
4.1. Gra 21
4.2. Agent 23
4.3. Eksperymenty 26
5. Podsumowanie 32
Literatura 34

1. Wstęp

Postęp technologiczny charakteryzuje się coraz szybciej następującymi zmianami, czasami tak nagłymi, że zostają nazwane rewolucjami. I tak, po rewolucji maszyny parowej, elektryczności i komputera, gwałtowny rozwój w dziedzinie przetwarzania informacji, który obserwujemy od ponad dekady, nazywany jest czwartą rewolucją przemysłową. [1] Wśród czołowych narzędzi, umożliwiających ten skok w postępie, obok Internetu i wirtualnej rzeczywistości, znajduje się sztuczna inteligencja.

Sztuczną inteligencję można nazwać każdy system cyfrowy lub maszynowy, który naśladuje ludzki proces myślenia, w szczególności zaś zdolność do przetwarzania i analizowania informacji oraz podejmowania decyzji. Szczególnym podzbiorem systemów klasyfikujących się jako sztuczna inteligencja jest uczenie maszynowe, opisujące algorytmy budujące modele matematyczne na podstawie otrzymywanych danych, z możliwością autonomicznego poprawiania i udoskonalania tych modeli w oparciu o zdobywane doświadczenie. Analogia do ludzkiego procesu rozumowania jest w tym przypadku jeszcze mocniejsza, ponieważ na topologię systemów uczenia maszynowego składają się sztuczne neurony, tworzące sieci. Takie sztuczne sieci neuronowe przetwarzają informacje w podobny sposób do ludzkiego mózgu - neurony przekazują między sobą informacje, wzmacniając lub osłabiając przekazywany sygnał, tworząc w ten sposób działający system, zdolny do rozwiązywania zadanego problemu. [2]

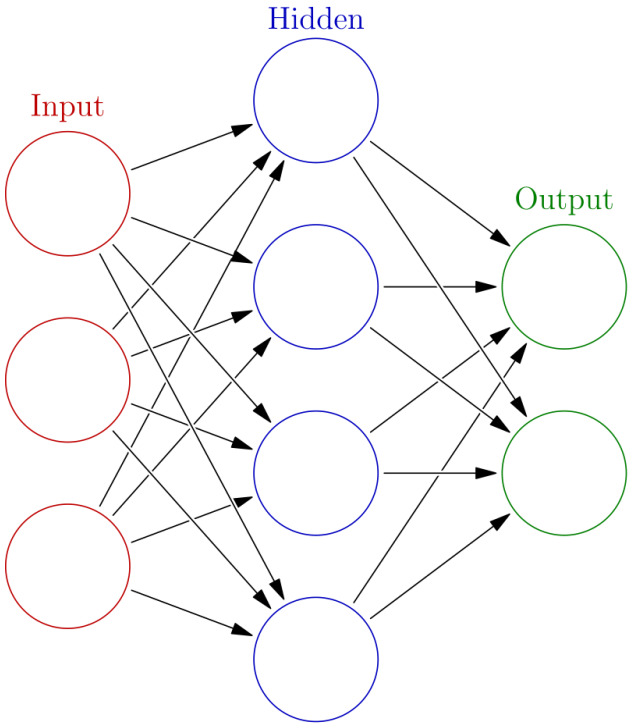
Szczególnym przypadkiem uczenia maszynowego jest uczenie się głębokie. Charakteryzuje je bardziej złożona architektura - składają się z wielu warstw neuronów ukrytych, zdolnych przetwarzać znacznie bardziej złożone dane, w większych ilościach niż inne systemy uczenia maszynowego. Sieci głębokiego uczenia się różnią się między sobą podejściem do przetwarzania danych, wyrażonym rodzajem wykorzystywanego algorytmu. Można wśród nich wyróżnić takie rodzaje, jak sieci rekurencyjne, sieci konwolucyjne, klasyczne sieci głębokie lub sieci głębokie wykorzystujące uczenie się ze wzmocnieniem. Te ostatnie są przedmiotem tego projektu, ze względu na ich charakterystyczną zdolność do uczenia się bez nadzoru, tzn. same, metodą prób i błędów, dochodzą do wniosku jak sklasyfikować daną informację. Oznacza to brak konieczność przygotowania danych uczących, ręcznie sklasyfikowanych przez człowieka, które posłużyłyby algorytmowi za przykład, z którego by się uczył. Dzięki tej szczególnej własności, głębokie uczenie się ze wzmocnieniem, nazywane w skrócie DRL (*ang. deep reinforcement learning),* znajduje szerokie zastosowanie w wielu dziedzinach nauki – robotyce, analizie danych, ale także w przemyśle rozrywkowym, w głównej mierze w grach komputerowych.

Użycie sztucznej inteligencji w programowaniu gier video jest wyjątkowym usprawnieniem, otwierającym wiele możliwości twórcom tej formy rozrywki. Najczęstszym, chociaż nie jedynym wykorzystaniem tego narzędzia w grach jest tworzenie agentów - modułów decyzyjnych, nauczonych kontrolować daną postać lub system będący częścią rozgrywki [3]. Współcześni developerzy, tworząc gry komputerowe, korzystają z silników do gier - środowisk programistycznych, oferujących gotowe rozwiązania i schematy najbardziej podstawowych i najczęściej używanych mechanizmów gier. Dodatkowo realizują integrację kodu źródłowego z modelami 3D, audio i animacjami. Przykładem takiego silnika jest Unity, GameMaker lub Unreal Engine.

Celem projektu jest zaprezentowanie możliwości implementacji algorytmów głębokiego uczenia się ze wzmocnieniem w grach komputerowych, z wykorzystaniem silnika Unreal Engine. W części teoretycznej omawiam podstawową architekturę i podział sieci neuronowych wraz z schematem działania algorytmów istotnych dla tej pracy. Przybliżam także krótką historię przemysłu gier komputerowych i rozwój technik zastosowania w nim sztucznej inteligencji. W części praktycznej projektu opisuję moją implementację ośmiu algorytmów głębokiego uczenia się ze wzmocnieniem na przykładzie prostej gry typu *tor przeszkód*, napisanej przeze mnie w wersji 4.27 silnika Unreal Engine 4, [4] z wykorzystaniem wtyczki MindMaker. [5]

2. Sieci neuronowe

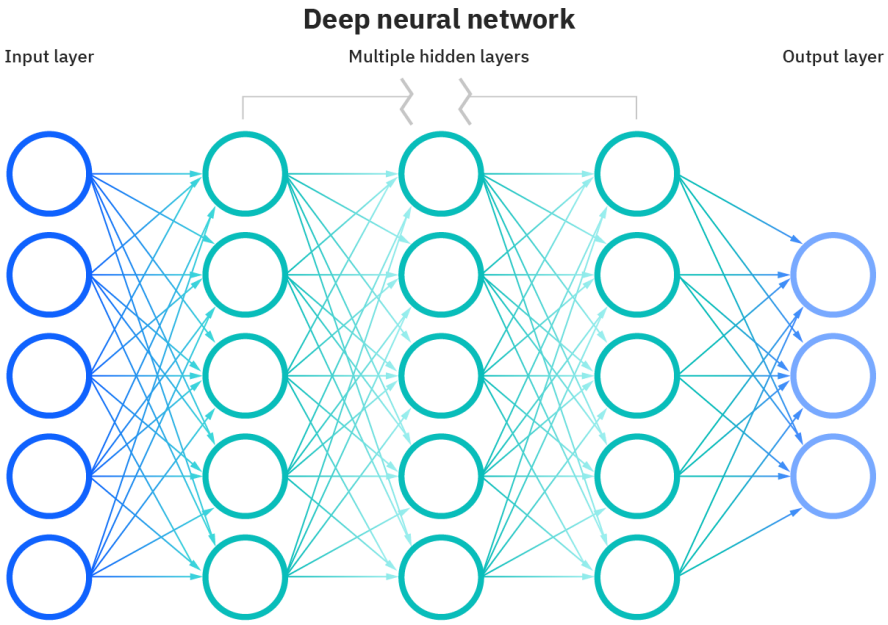
Sieci neuronowe to systemy przetwarzania informacji inspirowane biologiczną architekturą i zasadą działania układu nerwowego. Na model sieci neuronowej składają się warstwy neuronów: wejściowa, na której podawane są dane, warstwa ukryta oraz wyjściowa, z której otrzymywany jest wynik. [2] Rys. 2.1. przedstawia schemat topologii takiej sieci.



Rys. 2.1. Sieć neuronowa z jedną ukrytą warstwą neuronów [6]

Proces uczenia się sieci neuronowej może być przeprowadzony według różnych metod i algorytmów. Każdy proces odbywa się przez zadaną liczbę epok uczenia się, gdzie sieć w każdej kolejnej epoce korzysta z dotychczasowego postępu w nauce.

Tematem tej pracy jest głębokie uczenie się ze wzmocnieniem. Sieciami głębokimi nazywany sieci neuronowe złożone z wielu warstw neuronów. Jak zostało to zaprezentowane na rys. 2.2., przedstawiającym schemat topologii głębokiej sieci neuronowej, wyjście każdej z warstw neuronów służy jako wejście dla kolejnej. W ten sposób powstaje sieć znacznie bardziej złożona niż sieć składająca się z pojedynczej warstwy neuronów, ale daje to możliwość przetwarzania znacznie większej liczby danych.

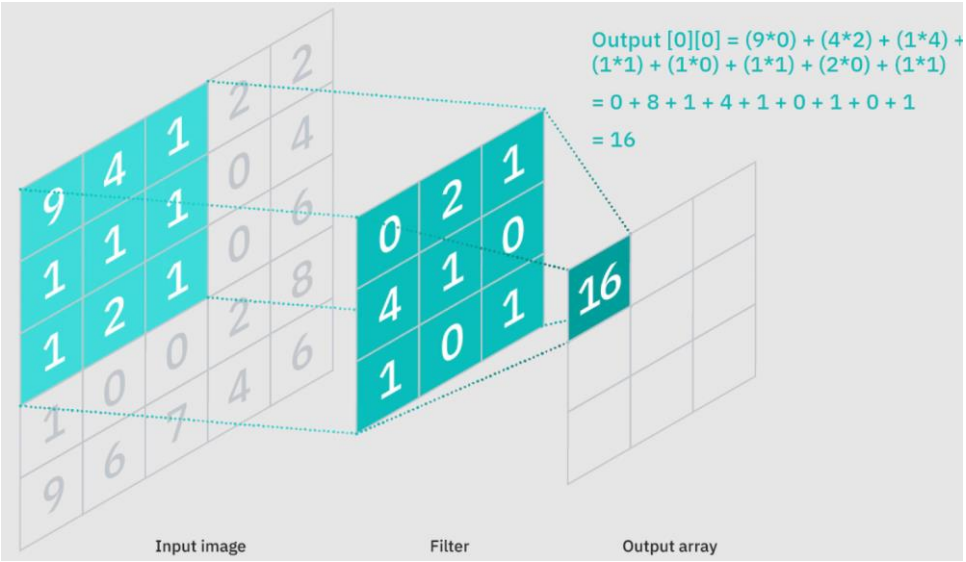


Rys 2.2. Głęboka sieć neuronowa, z wieloma ukrytymi warstwami neuronów [2]

2.1. Algorytmy głębokiego uczenia się

Cechą wspólną sieci głębokich jest ich architektura - wielość ukrytych warstw neuronów. Jednak metody i reguły, według których uczą się różne rodzaje sieci głębokich, są rozmaite i różnią się od siebie podejściem do procesu uczenia. Warto spojrzeć na ogólny zarys różnych algorytmów.

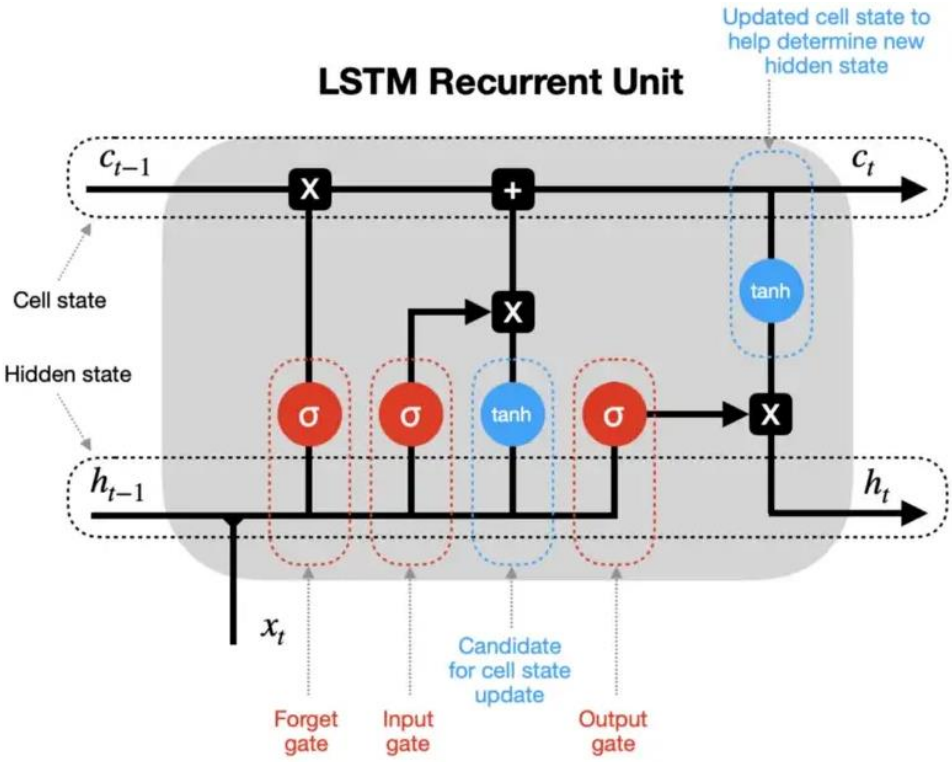
Konwolucyjne sieci neuronowe (*CNN – Convolutional Neural Network*) wykorzystują narzędzie analizy matematycznej zwane konwolucją lub splotem, używane także w dziedzinie przetwarzania obrazów. Polega ono na przemnażaniu danej funkcji, obrazu lub danych przez macierz, pełniącą rolę filtra. Macierz taką nazywamy kernelem. [7] Przykładowy efekt takiego przemnożenia jest widoczny na rys. 2.3, na którym tablica oznaczona *Input image* jest tablicą wejściową, tj. danymi które są przetwarzane, *Filter* oznacza kernel a *output array* jest tablicą, do której zostają zapisane wyniki konwolucji.



Rys. 2.3. Kernel filtrujący wejście w CNN [7]

W ten sposób sieć przetwarza wejściowy zbiór dany (np. obraz) przez warstwy konwolucyjne i zwraca na warstwie wyjściowej zestaw danych z uwypuklonymi interesującymi nas cechami. Co istotne, całe wejście jest przetwarzane przez ten sam kernel. W kolejnych krokach uczenia sieć optymalizuje stosowaną macierz w oparciu o otrzymywane wynik. Co oznacza, że do procesu uczenia potrzebne są dane uczące, do których sieć mogłaby porównywać kolejne wyjścia. [7]

Innym przykładem sieci głębokiego uczenia się są sieci LSTM (*ang. Long Short-Term Memory,* pamięć długotrwała/krótkotrwała). Cechą charakterystyczną takich sieci jest sprzężenie zwrotne wykorzystywane w algorytmie. W każdym zadanym momencie sieć nie przetwarza jedynie punktowych danych, jak obraz czy tabela, ale uwzględnia także dane, które otrzymał w przeszłości, a więc pracuje na sekwencji danych. Na rys. 2.4., przedstawiającym schemat komórki tworzącej taką sieć, należy zwrócić uwagę na elementy oznaczone symbolami c_{t-1} oraz h_{t-1}, reprezentujące połączenie ze stanem komórki i stanem ukrytym w poprzednim kroku uczenia. Nazwa sieci nawiązuje do pamięci krótkotrwałej i długotrwałej, którymi dysponuje człowiek. Analogicznie jak w tym fizjologicznym pierwowzorze, LSTM przypisuje zmieniające się wagi tym pamięciom i w ten sposób dochodzi do wyników procesu uczenia się. [8]

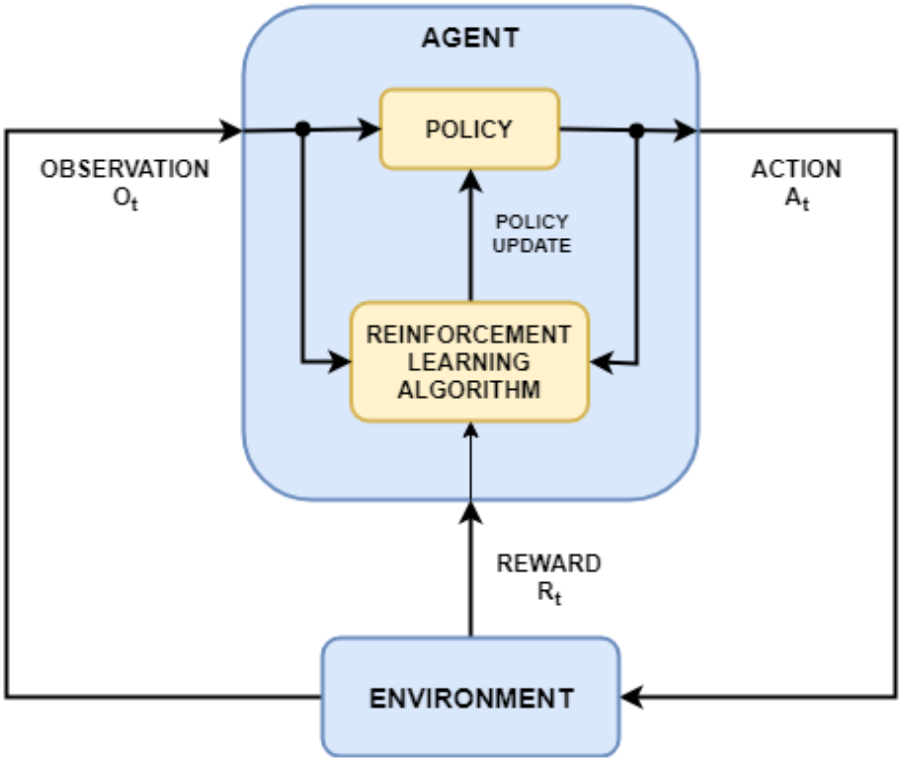


Rys. 2.4. Schemat pojedynczego neuronu sieci rekurencyjnej LSTM [8]

Trzecią grupą algorytmów głębokiego uczenia się, o której warto powiedzieć, jest uczenie się głębokie ze wzmocnieniem, użyte w tym projekcie. W celu zrozumienia zasady działania takiej sieci, należy najpierw omówić czym jest ogólnie uczenie się ze wzmocnieniem.

2.2. Uczenie się ze wzmocnieniem

Jednym z wielu sposobów uczenia maszynowego jest uczenie ze wzmocnieniem lub uczenie przez wzmocnienie (ang. *Reinforcment Learning*). W odróżnieniu od innych metod uczenia maszynowego, ta nie wymaga zbioru danych uczących. Proces uczenia agenta odbywa się poprzez udostępnienie mu środowiska ze zdefiniowanym systemem kar i nagród - akcje przybliżające agenta do założonych celów skutkują otrzymaniem pozytywnej nagrody – wzmocnienia, natomiast te które od tych celów oddalają - kary. Schemat ten jest zobrazowany na rys. 2.5. [9]



|*Rys. 2.5. Schemat uczenia się ze wzmocnieniem sieci neuronowej – agenta* [12]

Q-Learning to algorytm uczenia się ze wzmocnieniem obliczający wartość, jaką przypiszemy danej akcji. Q w tej metodzie oznacza funkcję, zgodnie z którą program oblicza spodziewaną nagrodę za podjęcie danej akcji obecnym stanie, czyli przy otrzymanych obserwacjach. Funkcja więc wartościuje zadane kombinacje akcji (A) i stanów (S):

$$Q : S \times A \to \mathbb{R}. \tag{2.1}$$

Wartość Q na początku procesu uczenia się jest arbitralnie ustawiana przez twórcę sieci. Następnie, w każdej kolejnej chwili czasu *t*, agent wybiera akcję a_t, którą wykona, zapamiętuję nagrodę r_t, którą otrzyma za wykonanie tej akcji oraz nową obserwację s_{t+1}. Na podstawie tych danych zostaje obliczona nowa wartość Q:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{current value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}^{\text{current value}} \Big)}^{\text{temporal difference}}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (temporal difference target)}}$$

(2.2)

Wzór 2.2 wprowadza trzy nowej pojęcia: współczynnik uczenia (*ang. learning rate*), czynnik osłabiający (*ang. discount factor*) oraz różnica czasowa (*ang. temporal difference*). Współczynnik uczenia to stała wartość, określana przez programistę. Odpowiada za gwałtowność zmiany Q – im mniejszą ma wartość, tym wolniej zmienia się wartość funkcji.

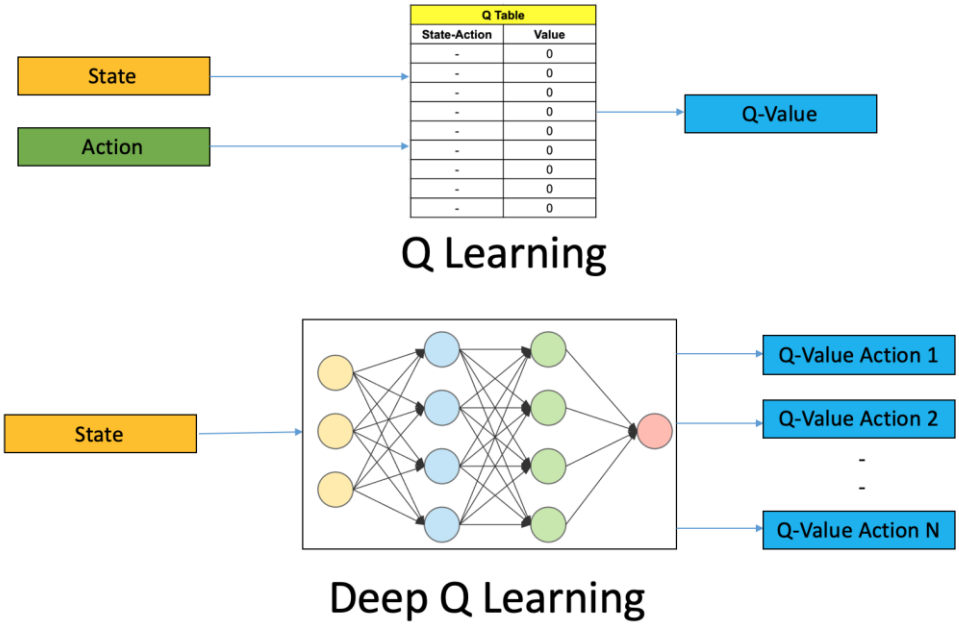
Algorytm płynniej przechodzi wtedy pomiędzy wartościami Q, ale kosztem jest mniejsza eksploracja metoda – poszukiwanie nowych taktyk, podejmując losowe decyzje i sprawdzając ich skuteczność. Czynnik osłabiający określa znaczenie przyszłych nagród. Im jest większy, tym bardziej "dalekowzroczny" będzie algorytm - większą wagę będzie przypisywał przyszłym nagrodom. [10]

Szczególnym przypadkiem uczenia się ze wzmocnieniem jest DRL (*ang. deep reinforcement learning*, głębokie uczenie się ze wzmocnieniem). Sieć implementująca algorytm DRL działa według zasad zaprezentowanych powyżej, ale z jednoczesnym wykorzystaniem architektury sieci głębokiego uczenia, o wielu złożonych warstwach neuronów ukrytych. Wraz z rozwojem tej gałęzi sieci neuronowych powstało wiele algorytmów uskuteczniających DRL, o różnym poziomie zaawansowania i różnych przeznaczeniach, znajdujące zastosowanie w rozmaitych dziedzinach i rodzajach danych. W kolejnym podrozdziale przedstawię podstawową charakterystykę tych sieci, które zostały wykorzystane w tym projekcie.

2.3. Algorytmy DRL

W tym projekcie wykorzystuję zestaw 5 algorytmów uczących i porównuję otrzymane przez nie wyniki uczenia. Należy więc się przyjrzeć różnicom w zasadzie działania dostępnych algorytmów - DQN, A2C, ACER, ACKTR, PPO. Wszystkie te algorytmy realizują głębokie uczenie ze wzmocnieniem, dlatego ich cechami wspólnymi są między innymi parametry – learning rate, gamma, layers, liczba epok.

Deep Q-Network (DQN) to najprostsza i najbardziej podstawowa z dostępnych metod. Wykorzystuje ona w pełni założenia głębokiego uczenia ze wzmocnieniem. Różnica w stosunku do klasycznego uczenia się ze wzmocnieniem polega na zastąpieniu prostej Q-funkcji (wzór 2.2) głęboką siecią: Q-Network. [11] Porównanie to obrazuje rys. 2.6.



Rys. 2.6. Porównanie topologii algorytmów Q-Learning i Deep Q-Learning [11]

Advantage Actor Critic (A2C) to metoda łączące w sobie dwa podejścia - oparte o wartościowanie (*value-based)* oraz taktyczne (*policy-based*). Agent działa więc w oparciu o dwie sieci – Krytyka i Aktora. Aktor mapuje wejście - a więc informacje o stanie środowiska - na wyjście, czyli decyzje jaką akcje podjąć. Krytyk stosuje podejście wartościujące, przypisując stanom Q-wartość (*Q-value, quality of state*). Ocenia w ten sposób, które stany są bardziej pożądane, następnie przekazuje tą informacje Aktorowi, który używa jej do znalezienia akcji prowadzącej do otrzymania większej nagrody. W metodzie tej korzysta się z obliczania różnic czasowych. [13]

Actor Critic with Experience Replay (ACER) jest metodą o podobnym działaniu co A2C, ponieważ wywodzi się z metody A3C - Asynchronous Advantage Actor Critic. Metoda A3C wprowadza pojęcie asynchroniczności do działania A2C. Tworzonych jest wiele agentów, działających w sposób niezależny od siebie, ale kontrolowanych przez jedną, globalną sieć, zasilaną przez wiedzę zdobywaną przez wszystkich agentów razem. ACER bazuje na A3C, ale stosując podejście *off-policy.* [14]

Actor Critic using Kronecker-factored Trust Region (ACKTR) to również metoda wykorzystująca podejście aktor-krytyk, ale wprowadzająca pojęcie regionu zaufania. Jest to podzbiór obszaru Q-funkcji, który jest rozszerzany lub pomniejszany, w zależności od trafności aproksymacji funkcji. [15]

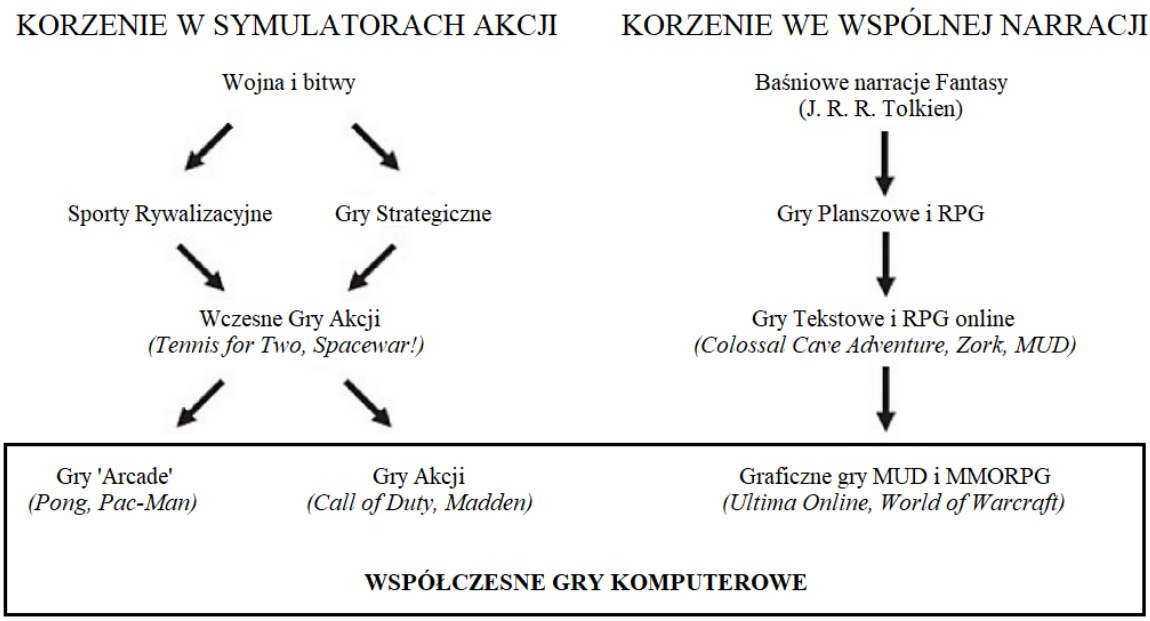
Proximal Policy Optimization (PPO) to metoda *on-policy.* Posiada ona pewne cechy wspólne z podejściami wykorzystującymi model Trust Region, ale jest znacznie prostsza i bardziej optymalna. W każdym kolejnym kroku uczenia aktualizuje taktykę tak, aby zminimalizować funkcję kosztów przy możliwe małym odstępstwie od poprzedniego kroku. Nagły przyrost lub znaczne obniżenie wartości wag powoduje często podejmowanie błędnych, a nawet szkodliwych decyzji z punktu widzenia procesu uczenia się. PPO rozwiązuje ten problem poprzez małe zmiany w polityce, które skutkują większą stabilnością sieci. [16]

3. Programowanie gier

Jedną z głównych gałęzi przemysłu rozrywkowego są gry komputerowe. Od czasu pierwszych gier, rozwijanych prawie 70 lat temu, [17] wiele się zmieniło w tym temacie. Wciąż rozwijający się przemysł, zasilany coraz bardziej zaawansowanymi technologiami, otwiera coraz to nowsze kierunki rozwoju cyfrowej rozrywki – konsole do gier, gry mobilne, wirtualna rzeczywistość, gry edukacyjne, gry online, gry multiplayer. W tym miejscu należałoby sformułować prostą definicję gier video, zanim przejdę do omawiania poszczególnych elementów tego medium, istotnych dla tej pracy.

Gra video, lub gra komputerowa to forma elektronicznej gry, która umożliwia graczowi interakcję poprzez interfejs użytkownika lub urządzenie wejściowe (np. Joystick, klawiaturę, kontroler czy urządzenie wykrywające ruch gracza) w celu wygenerowania odpowiedzi, przekazywanej graczowi za pomocą urządzenia wyjściowego (np. monitora, ekranu telefonu lub zestawu do wirtualnej rzeczywistości). Niezbędnym elementem każdej gry jest jasno określony cel, do którego ma dążyć gracz – pokonanie przeszkód, zdobycie jak największej liczby punktów, pokonanie przeciwnika – osobowego lub cyfrowego – w zadanej dyscyplinie, itd.

W grach komputerowych można wyróżnić wiele elementów, między innymi motyw, fabułę i mechanikę. Motywem gry są okoliczności towarzyszące przedstawionej w grze tematyce. Jak wygląda świat przedstawiony, jakie były inspiracje kulturowe, jakiego rodzaju jest to gra – na te wszystkie pytania odpowiada motyw. Warunkuje on też mechanikę i ewentualną fabułę gry - jeżeli jest dana gra o motywie sportowym, można się spodziewać, że będzie to gra zręcznościowa bez wyraźnej fabuły. Uproszczony schemat genezy różnych rodzajów gier komputerowych na rys. 3.1. prezentuje proces klarowania się tych elementów.



Rys. 3.1. - Geneza gier komputerowych [17]

3.1. Sztuczna Inteligencja w grach komputerowych

Potrzeba sztucznej inteligencji w grach komputerowych dla współczesnego programisty jest oczywista - jeżeli gracz gra przeciwko grze, zdolność podejmowania decyzji przez program jest konieczna. W rzeczywistości jednak, pierwsze gry komputerowe, tworzone w latach 50', były albo na tyle proste, że nie było potrzeby modułu decyzyjnego w programie (*Zork, Pong*), albo posiadały jedynie tryb multiplayer (*SpaceWar!, Tennis for Two*). [17] Z czasem jednak zaczęto rozwijać różne implementacje AI do gier video, urozmaicające rozgrywkę. Pierwszymi takimi próbami było dodanie wrogów w grze, których decyzje podejmowane są przez maszyny stanowe (*ang. Finite-State Machine,* inaczej automat skończony). Przykładem takie gry jest Pacman – przeciwnicy w grze ścigają gracza lub uciekają przed nim, w zależności od zdolności do pokonania ich. [19]

Z czasem wchodzą takie rozwiązania jak drzewa decyzyjne, algorytmy typu "pathfinding" oraz sieci neuronowe. Wraz z rozwojem technologii użycie sztucznej inteligencji przekracza jedynie sterowanie wrogimi postaciami – zostaje ona użyta także do generowania losowych poziomów, odgrywania roli sojuszników/podwładnych gracza, znajdywania rozwiązania do stawianych graczowi zadań i wielu innych funkcjonalności, wyróżnionych na rys. 3.2.

Pattern	What player(s) do	Role of AI (in relation to player)	Example(s)
AI is Visualized	Observe AI state	Gives (strategic) information, showing states	Third Eye Crime
AI as Role-model	Imitate AI	Show agent actionas and behaviors, agents as puzzles	Spy Party
AI as Trainee	Teach AI	Child/student	Black & White
AI is Editable	Edit AI	Artifact/agent that player can author/manipulate	Galactic Arms Race
AI is Guided	Guide/manage the AI	Partly independent inhabitants, with players as their Gods	The Sims
AI as Co-creator	Make artifacts assisted by AI	Co-creator, making artifacts	ViewPoints AI
AI as Adversary	Play game against the opponent	Opponent (symmetric)	Chess, Go
AI as Villain	Combat the Villain(s)	Villain in game; mob, boss mob, NPC (asymmetric)	Alien Isolation
AI as Spectacle	Observe	Spectacle, enacting simulated society	Nowhere

Rys. 3.2. - Funkcjonalności AI w przemyśle gier video [18]

Wspomniane automaty skończone (Finite-State Machines) są najczęstszym narzędziem z dziedziny sztucznej inteligencji wykorzystywanym w grach komputerowych. Przyczyną może być ich prostota i łatwość implementacji – programista może w prosty sposób zdefiniować zbiór zachowań i bez potrzeby wykorzystania zewnętrznego oprogramowania zaimplementować go w swoim kodzie. Tak zwane maszyny stanowe dzielą przestrzeń zdarzeń lub zachowań obiektu w grze na stany logiczne. Każdemu stanowi przypisywany jest zestaw instrukcji wykonywanych tak długo, jak spełniony jest zadany warunek logiczny. Przykładem takiego rozwiązania może być prosty przeciwnik w dowolnej grze zręcznościowej - domyślnym stanem jest tryb czuwania lub patrolowania danego obszaru. Gdy gracz wejdzie w pole widzenia przeciwnika lub ustalony zasięg, zaczyna on ścigać gracza i go atakować, ewentualnie przeszkadzać mu w rozgrywce. Po utracie postaci w polu widzenia lub oddaleniu się poza wyznaczony obszar, wróg powraca na swoje miejsce i wchodzi na powrót w tryb czuwania. [18]

Drzewa decyzyjne są rozwiązaniem o podobnym stopniu zaawansowania i możliwościach. Różnicą drzew decyzyjnych jest brak jawnie zdefiniowanych stanów i przejść między nimi. Zamiast tego projektowane jest drzewo binarne, w którym na każdym rozgałęzieniu definiujemy warunek logiczny oraz dwie możliwe drogi wyboru. Na końcu każdej ścieżki w drzewie znajdują się instrukcje określające jakie zachowanie ma być zastosowane. [18]

Najbardziej zaawansowanym i złożonym rozwiązaniem implementującym sztuczną inteligencję w grach komputerowych są agenci. Wykorzystując omówione w poprzednim rozdziale sieci neuronowe, podejmują decyzje dotyczące wyborów i zachowań postaci i obiektów w grach komputerowych. Ze względu na możliwości przetwarzania ogromnej ilości danych, najczęściej używanymi są agenci wykorzystujący sieci głębokiego uczenia, również omówieni w rozdziale nr 2.

3.2. Platformy i narzędzia

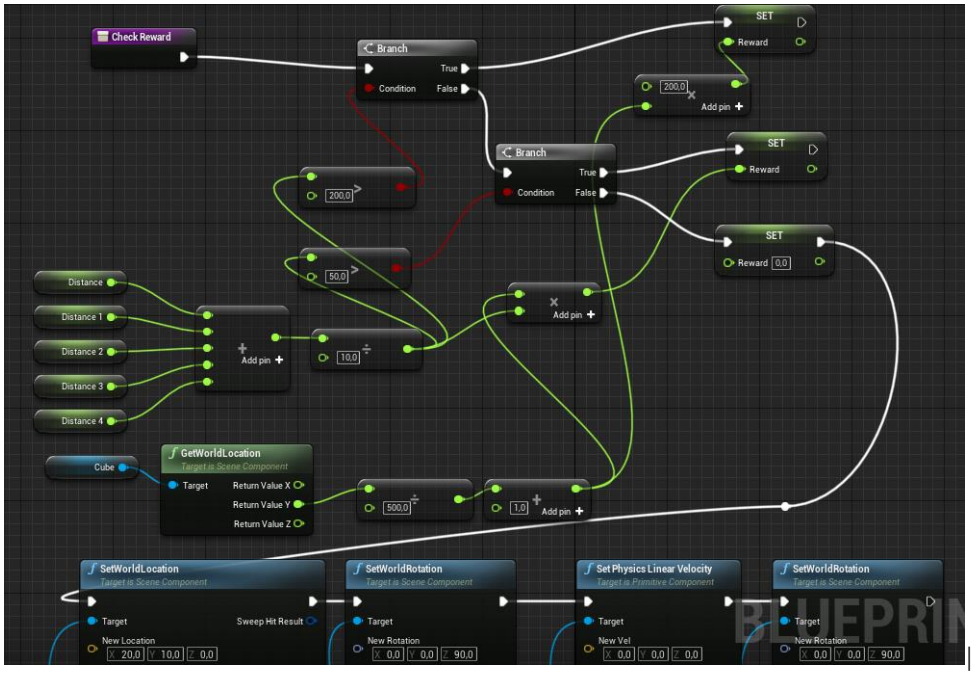
Zaczynając od pierwszych gier, tworzonych w środowiskach akademickich, przez studentów i pracowników naukowych w ramach eksperymentów lub prac naukowych w latach 50-tych, aż do wczesnych konsol (Odyssey Magnavox, Atari) i automatów do gier z początku lat 80-tych, jedynym językiem programowania używanym do ich tworzenia był język Asembler. [17] W tym czasie gry były ręcznie "programowane na sztywno" (hard-coded) od początku do końca, bez użycia żadnych gotowych klas czy szkieletów architektury oprogramowania. Dodatkowo były one dedykowane wyłącznie na daną platformę, nawet zmiana modelu konsoli na nowszy, dla przykładu o innych wymiarach wyświetlacza, skutkowała niekompatybilnością gry i koniecznością zaprogramowania jej od nowa lub poważnych zmian w kodzie.

Wraz z nowymi platformami i rozwiązaniami technologicznymi istniała coraz silniejsza potrzeba wieloplatformowości gier i możliwości ponownego użycia prostej architektury programu. Odpowiedzią na tą potrzebę był rozwój silników do gier. Silnikiem do gier nazywamy zintegrowane środowisko programistyczne oferujące zestaw narzędzi, modułów i bibliotek ułatwiających tworzenie gier komputerowych. Jednymi z pierwszych silników były Quake Engine oraz - używany w tym projekcie – Unreal Engine. Pojawiły się one w drugiej połowie lat 90-tych i używały już języka C. Współczesne silniki pozwalają na prace w różnych językach: w Unity kod jest tworzony w C#, Unreal Engine 5 używa C++, Java jest wykorzystywana w silniku jMonkeyEngine. Niektóre silniki, takie jak GameMaker Studio 2, wykorzystują własny język.

3.3. Unreal Engine

Narzędziem, które wykorzystałem w tym projekcie jest Unreal Engine 4, wersja 4.27. Ten podrozdział jest poświęcony krótkiej prezentacji tego środowiska.

Unreal Engine posiada własny, autorski system tworzenia oprogramowania – Blueprint Visual Scripting system. Jest to skryptowy system oparty o graficzny interfejs, w którym programista tworzy algorytmy za pomocą bloków i węzłów, reprezentujących funkcje programu. Rys. 3.3. prezentuje przykładowy fragment programu w systemie Blueprint. [4]



Rys. 3.3. Przykładowy fragment programu w systemie Blueprint

Programista ma wybór tworzenia projektu używając wyłącznie systemu Blueprint, lub łącząc go z klasami C++. W drugim wypadku środowisko Unreal Engine korzysta z kompilatora Visual Studio, w którym użytkownik pisze kod.

Jak większość współczesnych silników do gier, Unreal Engine posiada szereg narzędzi i systemów ułatwiających programistom tworzenie gier. Wiele schematów powtarza się we wszystkich grach, dlatego możliwość wykorzystania gotowego prototypu takich mechanizmów, jak grawitacja, pęd, współrzędne obiektu, natężenie światła czy triggery, znacznie przyśpiesza proces tworzenia oprogramowania. W omawianym silniku jest to zrealizowane poprzez system prototypów klas, zawierających wstępne zmienne i metody. Takimi klasami są np.: obiekt, aktor, charakter, komponent, źródło światła, klasa ruchu postaci. [4]

3.4. MindMaker

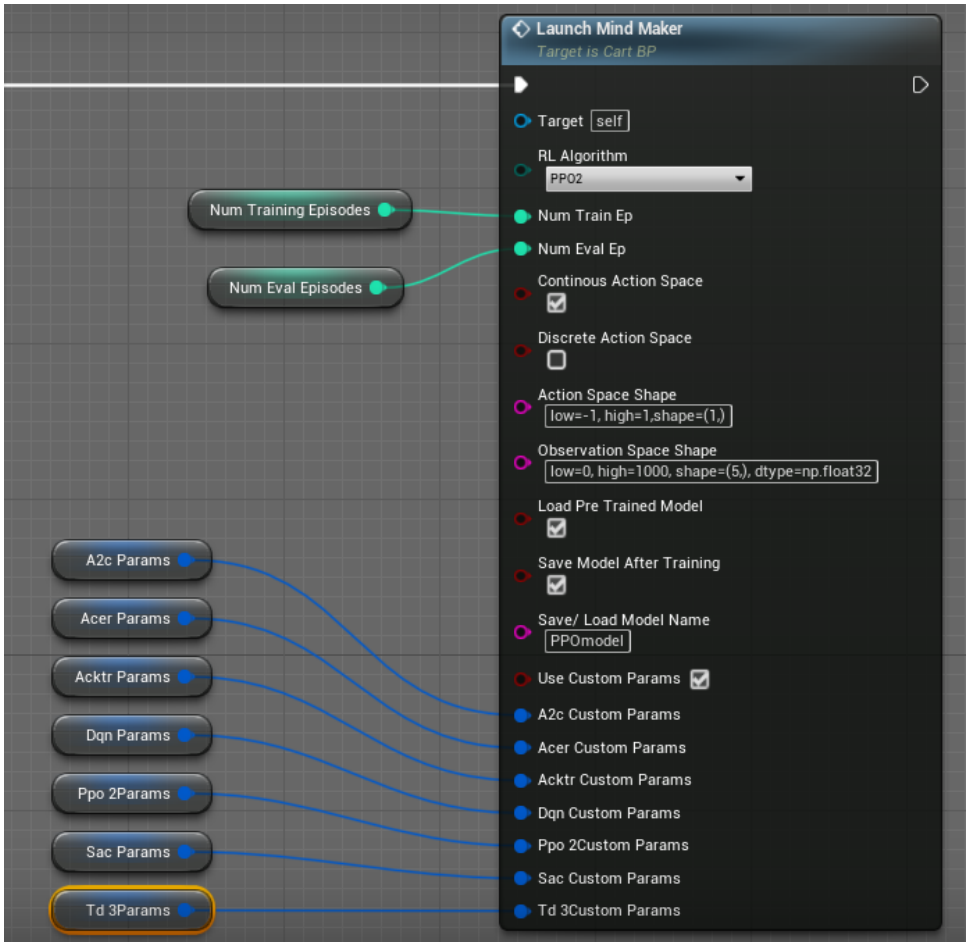
Modułem implementującym sieci neuronowe i głębokie uczenie się ze wzmocnieniem w silniku Unreal Engine jest moduł MindMaker. Nie jest on zasadniczym modułem silnika i nie należy do bazowych narzędzi środowiska. Jest to open-source'owy projekt rozwijany przez Aarona Kruminsa, oferujący wtyczkę w postaci systemu klas Blueprint realizujący uczenie się maszynowe. Główną z tych klas jest MindMakerActor, będąca prototypem agenta uczącego się. Funkcje w tej klasie nie są gotowymi rozwiązaniami, ale przestrzenią w kodzie dla programisty na zaimplementowanie kluczowych etapów uczenia maszynowego. [5]

Z pośród wielu funkcji klasy MindMakerActor, istotnymi dla tego projektu są:

- Launch Mind Maker – jest to funkcja wywoływana automatycznie w momencie połączenia silnika z serwerem. Jako argumenty tej funkcji zostają podane: rodzaj algorytmu, parametry uczenia się oraz parametry właściwe dla algorytmu. Z poziomu

tej funkcji są konfigurowane wszystkie istotne cechy sieci neuronowej, takie jak kształt przestrzeni akcji i obserwacji.

- Make Observation - wywoływana jako pierwsza w każdej iteracji procesu uczenia się, funkcja ta formułuje sposób zbierania i zapisywania obserwacji przez agenta. Jej ciało zostaje uzupełnione przez programistę, w zależności od cech środowiska i typu danych, które agent obserwuje.
- Receives Action – w oparciu o obserwację, program wybiera akcję, którą agent ma wykonać w następnej iteracji. Akcja tak jest w tej funkcji odbierana i parsowana do odpowiedniego formatu, następnie wysyłana do serwera, odpowiadającego za sterowanie agentem.
- Check Reward – w tej funkcji agentowi przydzielana jest nagroda w oparciu o poprawność podejmowanych przez niego akcji i stopień realizacji założeń procesu uczenia.
- Event Graph – jest to funkcja, którą można porównać do metody *main* w innych językach programowania – jest wykonywana w pętli i zawiera wywołania wszystkich poprzednich funkcji.



Rys. 3.4. Wywołanie funkcji LaunchMindMaker

Wtyczka MindMaker pozwala zaimplementować do projektu sieci głębokiego uczenia wykorzystujące algorytmy zaprezentowane w tab. 3.1., wraz z ich parametrami.

Nazwa algorytmu	Parametry wspólne algorytmów	Parametry charakterystyczne
A2C - Advantage Actor Critic	policy, gamma, n_steps, vf_coef, ent_coef, max_grad_norm, learning_rate, lr_schedule, verbose, network_arch, act_func	alpha, epsilon
ACER - Actor Critic with Experience Replay	policy, gamma, n_steps, vf_coef, ent_coef, max_grad_norm, learning_rate, lr_schedule, verbose, network_arch, act_func	alpha, epsilon, r_prop_alpha, r_prop_epsilon, q_coef, delta, buffer_size, replay_ratio, replay_start, correction_term,
ACKTR - Actor Critic using Kronecker-factored Trust Region	policy, gamma, n_steps, vf_coef, ent_coef, max_grad_norm, learning_rate, lr_schedule, verbose, network_arch, act_func	nprocs, vf_fisher_coef, kfac_clip, kfac_update, gae_lambda
PPO - Proximal Policy Optimization	policy, gamma, n_steps, vf_coef, ent_coef, max_grad_norm, learning_rate, lr_schedule, verbose, network_arch, act_func	lambda, cliprange, cliprange_vf
DQN – Deep Q-Network	policy, gamma, n_steps, learning_rate, lr_schedule, verbose, network_arch, act_func	buffer_size, double_q

Tab. 3.1. Wykorzystywane w projekcie algorytmy DRL wraz z ich parametrami

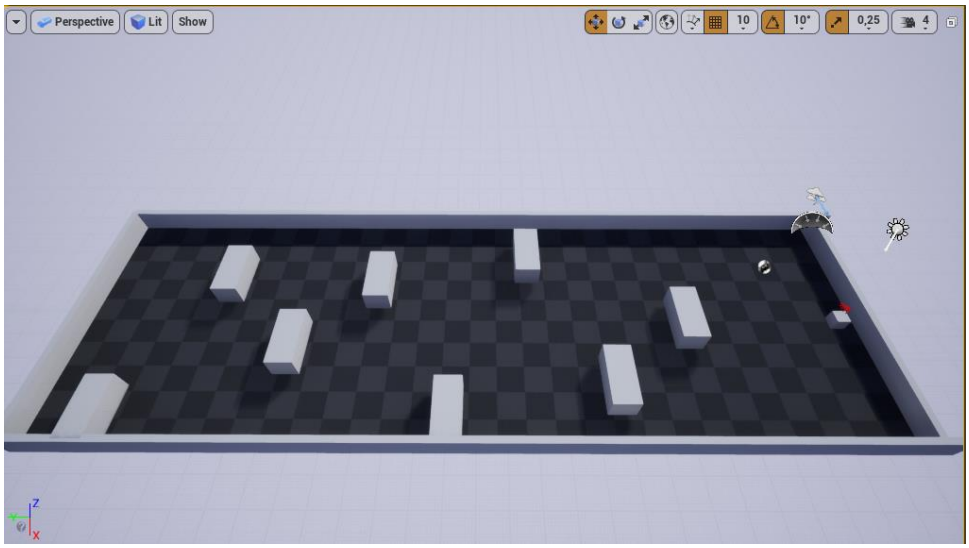
Dwa ostatnie algorytmy: SAC i TD, widoczne na Rys. 3.4., nie będą używane w tym projekcie, dlatego nie zostały szczegółowo opisane. Zrezygnowanie z wykorzystania tych algorytmów wynika z problemów z implementacją - zostały one zaprojektowane w inny sposób niż pierwsze 5 i wymagałyby zmian w programie i agencie.

4. Opis aplikacji

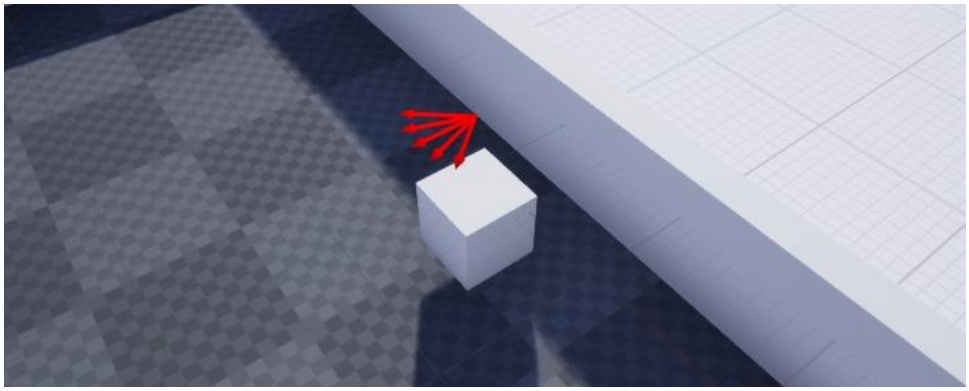
Po dokonaniu prostego przeglądu technik i narzędzi, którymi współcześnie można tworzyć gry komputerowe, w tym rozdziale zaprezentuje użycie wybranych narzędzi na przykładzie prostej gry typu *Tor przeszkód*. W grze tego typu gracz steruje postacią biegnącą przez generowane losowo kolejne poziomy z przeszkodami. Gra może mieć różne warianty, np. *Endless Runner* - bieg z zasady nigdy się nie kończy, a celem gry jest przebiegnięcie jak najdłuższego odcinka w jak najkrótszym czasie. Przykładami takich gier są: Sonic Dash (Hardlight, Sega Sammy Holdings, 2013), Crash Bandicoot: On the Run! (King.com, 2021), Lara Croft: Relic Run (Crystal Dynamics, Simutronics, 2015) lub prekursor gatunku - B.C.'s Quest for Tires (Sydney Development, 1983). [20]

4.1. Gra

Poziom składa się z toru o długości 3000 m, ogrodzonego z czterech stron ścianami. Awatarem, reprezentującym gracza – a w tym przypadku sieć neuronową - jest biały sześcian (rys 4.2). Pomiędzy jego pozycją startową (po prawej) a metą (po lewej) znajduje się 8 bloków, będących przeszkodami. Każda przeszkoda znajduje się co kolejne 300 metrów toru, w równych odstępach od siebie, ale w każdym kolejnym uruchomieniu poziomu bloki są losowo umiejscawiane na osi X. Celem gry jest ominięcie przeszkód i dotarcie na drugą stronę toru. Uderzenie w przeszkodę lub ścianę skutkuje przegraną. Rzut poziomu z lotu ptaka jest widoczny na rys. 4.1.



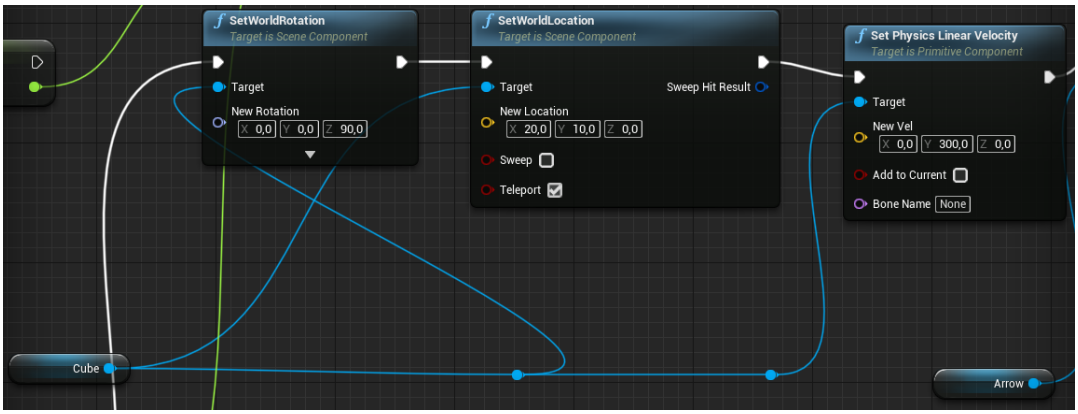
Rys. 4.1 Poziom gry



Rys. 4.2 Awatar gracza

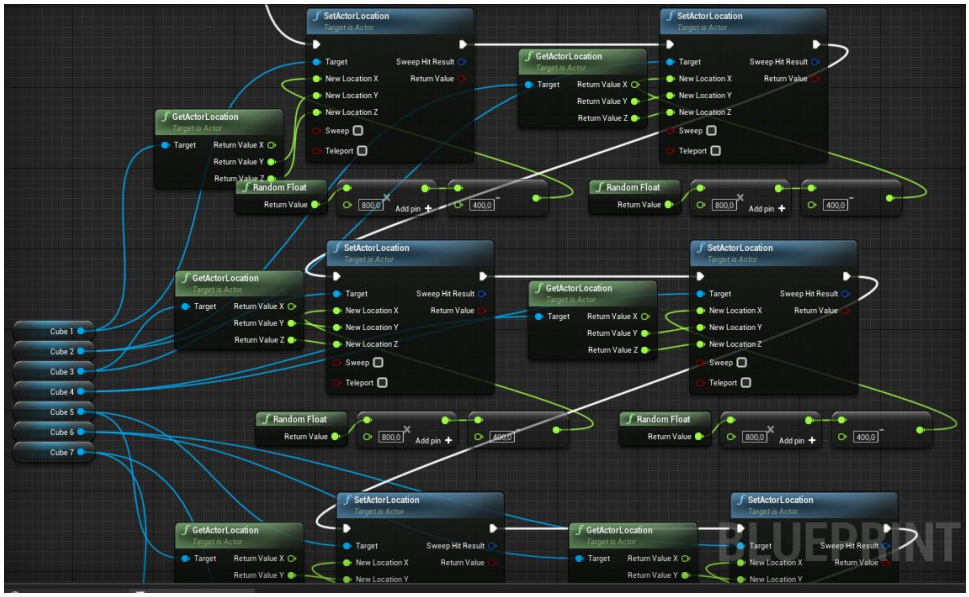
Na potrzeby wizualizacji procesu uczenia się, dodatkowo zostaje wyświetlona strzałka ponad pozycją startową awatara, wskazująca w którym kierunku w danej chwili porusza się agent, oraz promienie obliczające odległość od sześcianu do najbliższej przeszkody. Są one reprezentowane przez cienki, czerwone linie wychodzące z awatara.

W każdej iteracji wykonywania się programu, sieć neuronowa przykłada siłę o stałej wartości do sześcianu, powodując toczenie go w wybranym kierunku. Dostępnymi akcjami agenta są: zmiana kąta przyłożenia siły o 1 stopień w prawo lub w lewo albo użycie takiego samego kąta jak w poprzedniej iteracji. Przegranie poziomu przez agenta powoduje załadowanie gry od nowa – awatar zostaje przeniesiony na pozycję startową oraz przeszkody otrzymują nowe, losowe pozycje. Fragment kodu realizujący to przeładowanie poziomu pokazują rys. 4.3. i rys. 4.4. Na rys 4.3. program odwołuje się do obiektu *Cube*, będącego awatarem agenta i wywołuje jego metody: *SetWorldRotation*, *SetWorldLocation* oraz *SetPhysicsLinearVelocity*. Z widocznymi parametrami, metody te kolejno: przypisują awatarowi rotację i współrzędne w poziomie takie, jak w pozycji startowej oraz nadają mu wektor prędkości taki, jak w pierwszym kroku w każdym epizodzie.



Rys. 4.3 Fragment kodu przenoszący gracza na pozycję startową

Rysunek 4.4. przedstawia sekwencję bloków realizujących przypisanie przeszkodom (*Cube1, Cube2, Cube3, Cube4, Cube5, Cube6* oraz *Cube7*) losowe współrzędne X.



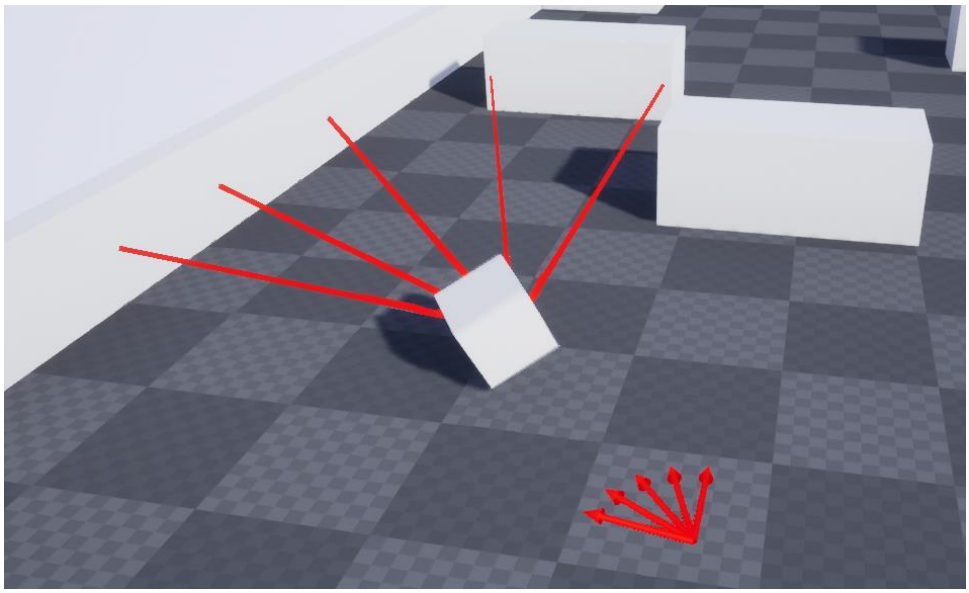
Rys. 4.4 Fragment kodu przenoszący przeszkody na losowe pozycje

4.2. Agent

Jak zostało to opisane w podrozdziale 3.4., wtyczka MindMaker oferuje jedynie prototypy klas i funkcji realizujących głębokie uczenie się ze wzmocnieniem. [5] Szczegóły implementacji zostają do zaprojektowania i wykonania przez programistę. W tym podrozdziale pokazuję moje rozwiązania następujących procesów - wykonanie obserwacji, obliczenie nagrody, wykonanie akcji oraz sprawdzenie warunków przegranej.

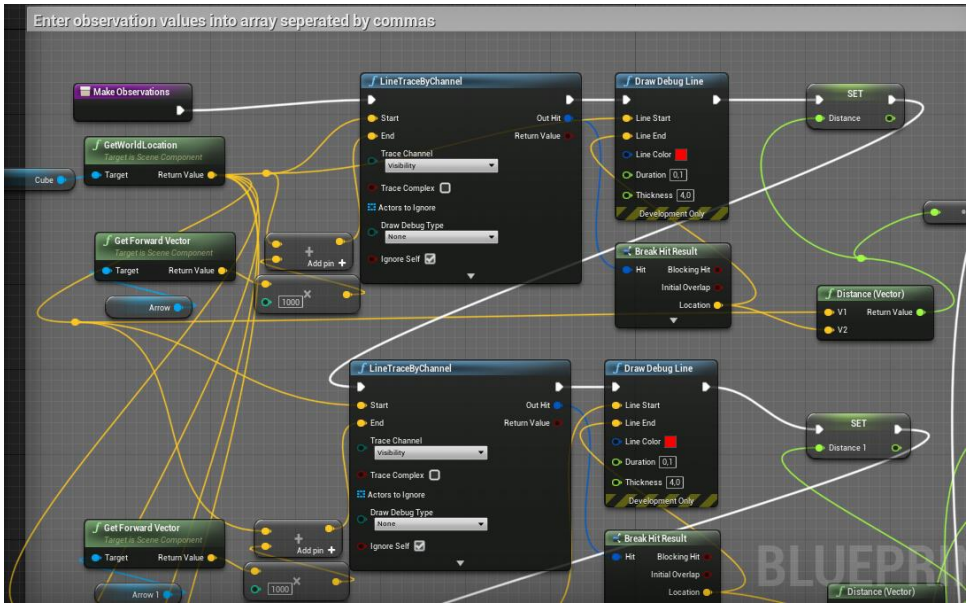
Ponieważ w każdym kolejnym załadowaniu poziomu, przeszkody znajdują się na losowych pozycjach, agent nie może jedynie uczyć się przechodzić stały układ przeszkód - musi nauczyć się dynamicznie omijać przeszkody, bez względu na ich współrzędne. Kluczowe w tej kwestii jest więc ustalenie jakie dokładnie dane środowiska będzie obserwował agent. Ponieważ ma on naśladować ludzi, którzy bez trudu byliby w stanie pokonać taki tor przeszkód, jaki znajduje się w poziomie, dobrym pomysłem jest uczynienie obserwacji agenta również podobnymi do obserwacji, jakie wykonują ludzie w takich sytuacjach. Idealnym rozwiązaniem byłoby przekazywać agentowi obraz, który widziałby, gdyby mógł zrobić zdjęcie w danej chwili, patrząc przed siebie, z perspektywy awatara. Jednak taki typ danych jest bardzo ciężki i wymagałby dużego nakładu pracy w celu przetworzenia. Nie wykluczone, że sieci DRL poradziłyby sobie z taką obserwacją, jednak da się uprościć ten typ danych. To, co kluczowe dla agenta, to odległość do najbliższej przeszkody. Unreal Engine posiada takie narzędzie, jak LineTraceByChannel, pozwalające na wypuszczenie promienia - można to porównać do puszczania wiązki światła z laseru - w danym kierunku. Metoda ta zwraca informacje o pierwszym napotkamy obiekcie, w tym jego współrzędne. Można więc łatwo obliczyć w ten sposób odległość agenta do najbliższej przeszkody. Jednak pierwsze próby implementacji takiego rozwiązania pokazały, że jedna wartość to za mało informacji dla agenta - łatwo w ten sposób nie zarejestrować przeszkody, jeżeli nie znajduje się ona centralnie naprzeciwko awatara. Zdecydowałem się więc na 5 promieni obliczających odległość - jeden

centralnie przed awatarem, oraz po dwa na prawo i lewo od niego, pod kątami 22.5 stopnia i 45 stopni (rys. 4.5.).

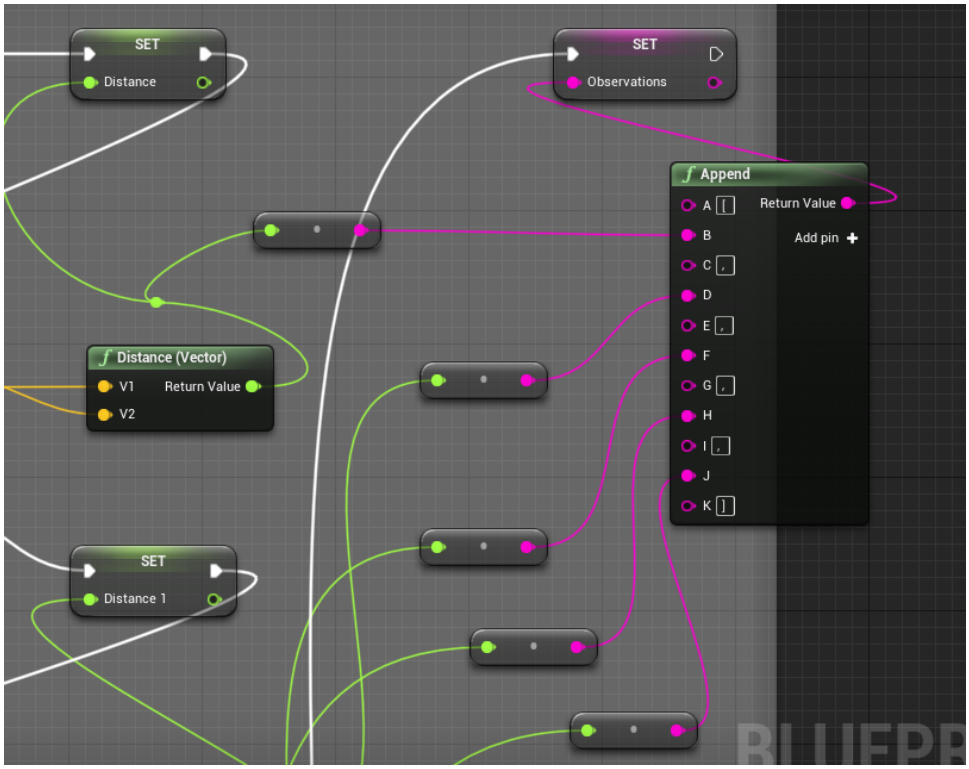


|*Rys. 4.5. Czerwone promienie, obliczające odległość do najbliższej przeszkody*

Na rys. 4.6. i 4.7. zaprezentowałem kod realizujący opisany proces. Biała linia, łącząca poszczególne bloki kodu oznacza kolejność wywoływania funkcji. Powtarzany jest 5 razy proces wyznaczania kierunku i obliczania odległości do najbliższej przeszkody oraz generowania fizycznej reprezentacji tego promienia. Na rys. 4.7. widać przypisanie zwracanych wartości do tablicy string'ów, która jest przekazywana na serwer z agentem.

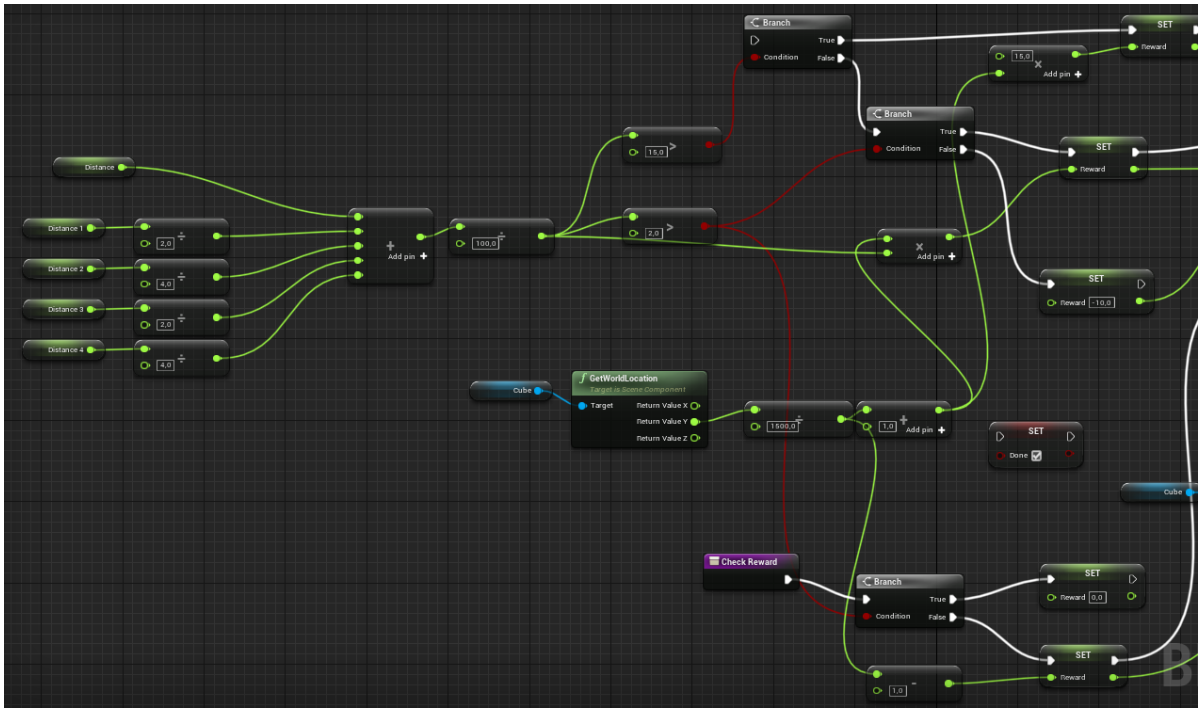


Rys. 4.6. Wywołanie metody LineTraceByChannel w ciele funkcji MakeObservation



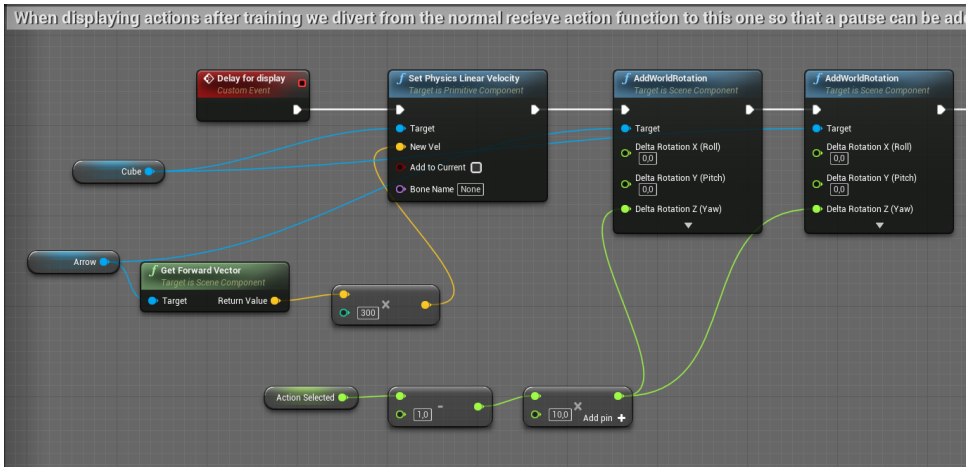
Rys. 4.7. Funkcja MakeObservation zwraca obserwacje w formie tablicy String'ów.

Funkcja CheckReward, zaprezentowana na rys. 4.8., sumuje zaobserwowane dystanse, nadając im wagi. Sumowane jest 100% środkowego dystansu (Distance), 50% dystansów pod kątem 22.5 stopnia (Distance1 i Distance3) oraz 25% dystansów pod kątem 45 stopni (Distance2 i Distance4). Obliczamy to w celu ustalenia czy awatar uderzył w przeszkodę lub ścianę. Ponieważ agent jest mało zwrotny, nie chcę, aby poziom kończył się, gdy awatar jedynie lekko obije się o przeszkodę swoim bokiem. Jeżeli obliczona suma jest zbyt mała, poziom zostaje zresetowany i agent otrzymuję odpowiednią nagrodę. W każdym innym przypadku, agent otrzymuje nagrodę równą zero. Nagroda na koniec poziomu obliczana jest według następującej zależności: przebyty dystans zostaje podzielony przez wartość 1500 (połowa długości toru) oraz pomniejszony o 1. W ten sposób przydzielamy ujemną nagrodę z przedziału wartości od –1 do 0, jeżeli agentowi uda się przebyć mniej niż połowę długości toru, albo nagrodę z przedziału wartości od 0 do 1 w przeciwnym wypadku.



Rys. 4.8. Ciało funkcji CheckReward

Rys. 4.9. prezentuje fragment kodu przydzielający akcję do wykonania. Ponieważ agent zwraca akcję ze zbioru {0, 1, 2}, należy pomniejszyć otrzymaną wartość o 1 i pomnożyć przez 10. W ten sposób obliczamy kąt, o jaki obróci się agent: 10 stopni, 0 lub –10 stopni.

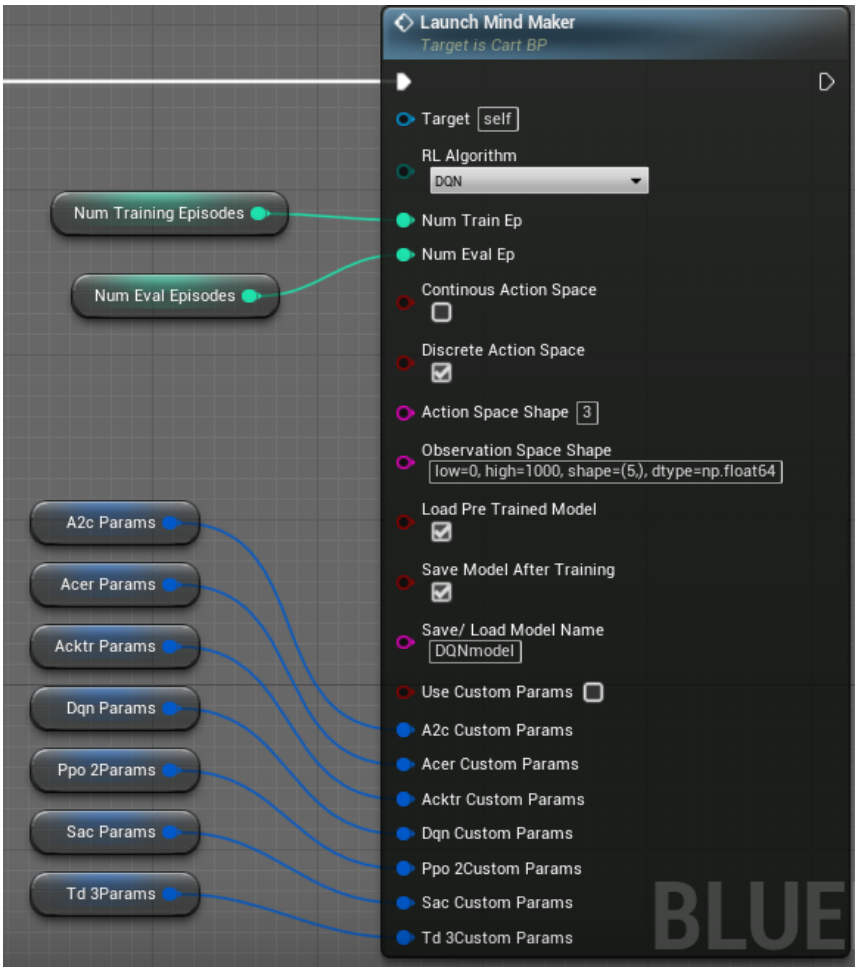


Rys. 4.9. Fragment kodu przydzielający akcję do wykonania

4.3. Eksperymenty

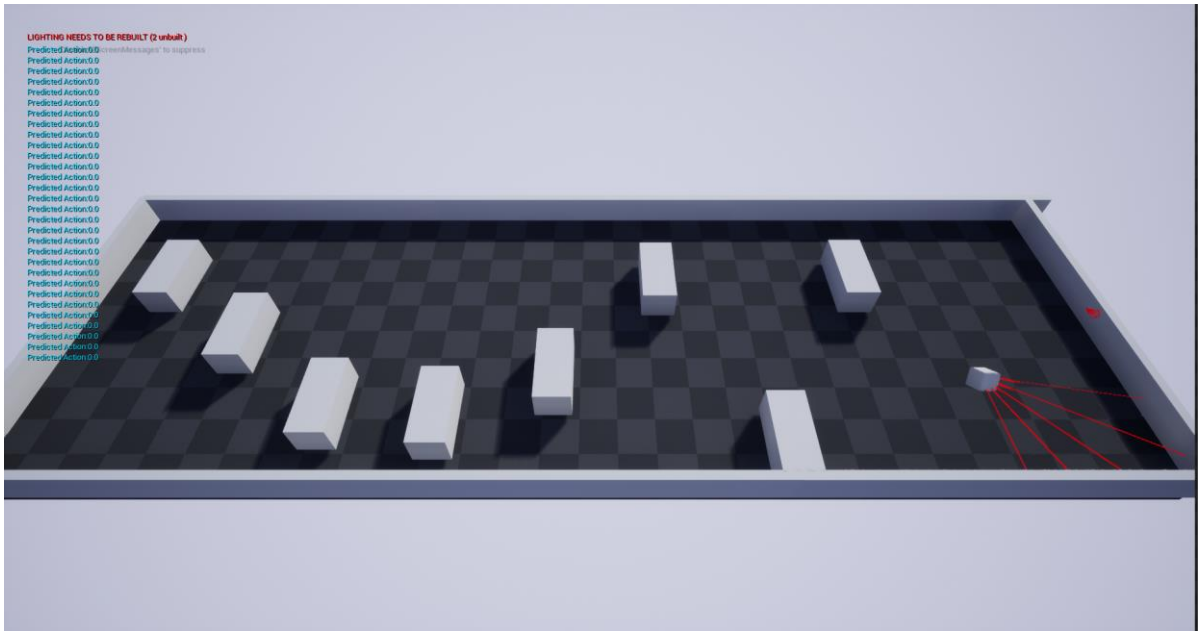
Uruchomienie procesu uczenia posiada wiele edytowalnych własności: rodzaj algorytmu, liczba epizodów uczenia, liczba epizodów ewaluacyjnych, rodzaj i kształt tablicy akcji i obserwacji. Szczególnie praktycznymi parametrami są *Load Pre Trained Model* oraz *Save Model After Training*. Pozwalają one na nauczenie danej sieci, a następnie odtworzenie

samych efektów procesu uczenia się. Istotna dla eksperymentów będzie także zmienna *Num Training Episodes*, widoczna na rys. 4.10, określająca liczbę epizodów uczących.



Rys. 4.10. Wywołanie funkcji LaunchMindMaker

W pierwszym eksperymencie, wszystkie algorytmy zostały uruchomione w ten sam sposób - z domyślnymi wartościami parametrów i liczbą epizodów uczenia równą 1000. W takim ujęciu wszystkie sieci prezentują się dosyć prymitywnie – nie są w stanie omijać przeszkód, a algorytmy PPO i ACKTR nie dochodzą dalej niż druga przeszkoda. Częstym problemem jest kręcenie się w kółko - agent wybiera jedną akcję i powtarza ją przez cały epizod. Najlepsze wyniki osiągnęła najprostsza sieć - DQN. Wpada ona w ściany i przeszkody, ale udaje jej się dotrzeć do drugiej połowy toru. W kolejnych podejściach będę manipulował parametrami sieci w celu zbadania ich potencjalnej efektywności.



Rys. 4.11. Zrzut ekranu prezentujący trwający proces uczenia się

Rys. 4.12. i 4.13 prezentuje zestawienie domyślnych wartości parametrów uczących poszczególnych algorytmów. Dodatkowo, wywołując funkcję LaunchMindMaker, można ustalić liczbę epizodów uczenia. Warto zwrócić uwagę na współczynnik uczenia – *learning rate*. Algorytmy różnią się między sobą domyślnymi wartościami tego parametru. W kolejnym eksperymencie będę zmieniał wartość współczynnika uczenia.

Rys. 4.12. Zestawienie domyślnych wartości parametrów uczących algorytmów A2C, ACER i ACKTR

Rys. 4.13. Zestawienie domyślnych wartości parametrów uczących algorytmów DQN i PPO

W drugim eksperymencie uruchamiam proces uczenia się poszczególnych algorytmów z tą samą liczbą epizodów uczących, co w pierwszym, równą 1000, ale z 10-krotnie większą wartością współczynnika uczenia się niż domyślna dla każdego z algorytmów. Na algorytmy A2C, ACER, ACKTR i PPO zmiana ta zdaje się nie mieć dużego wpływu - dalej nie są w stanie dojść dalej niż połowa toru. Istotną zmianą jest brak zapętlania się agentów od PPO i A2C. Zwiększenie współczynnika uczenia się zwiększyło eksplorację tych algorytmów. Algorytm DQN po tej zmianie nieznacznie, ale w stopniu zauważalnym pogorszył swoje wyniki. Dochodzi średnio o jedną przeszkodę mniej niż w poprzednim eksperymencie. Można wysnuć wniosek, że domyślna wartość współczynnika uczenia się była lepszą wartością dla tego modelu. W kolejnym eksperymencie zmienię liczbę epizodów uczenia się.

W trzecim eksperymencie uruchamiam proces uczenia się poszczególnych algorytmów z tą samą wartością współczynnika uczenia się, co za pierwszym razem, ale z liczbą epizodów uczenia się równą 5000. Przy takich parametrach w dalszym ciągu faworytem jest algorytm DQN - dzięki dłuższemu procesowi uczenia się udało mu się wypracować odpowiednią strategie i za każdym razem dochodzi on do końca toru, otrzymując najwyższą nagrodę. Pozostałe algorytmy również robią zauważalny postęp, jednak żadnego nie udaje się w dalszym ciągu dojść do końca toru.

Tabela 4.1. zawiera zestawienie średnich odległości od początku toru osiąganych przez poszczególne algorytmy w kolejnych eksperymentach. Przypomnę, że cały tor ma długość 3000m. Średnia była liczona z 5 podejść do toru po przejściu całego procesu uczenia się. Wartości zostały zaokrąglone do 50m. W pierwszej kolumnie wraz z numerem eksperymentu znajduję się wartość współczynnika uczenia (*learning_rate*) oraz liczba epizodów uczących (*Num Training Episodes)*.

	DQN	A2C	ACER	ACKTR	PPO
I Eksperyment *Learning_rate* =default *Num Training Episodes* =1000	2100m	450m	600m	650m	550m
II Eksperyment *Learning_rate* =default * 10 *Num Training Episodes* =1000	1900m	1500m	400m	500m	700m
III Eksperyment *Learning_rate* =default *Num Training Episodes* =5000	3000m	900m	1300m	1800m	2050m

Tab. 4.1. Zestawienie wyników poszczególnych algorytmów w każdym z eksperymentów

Z trzech przeprowadzonych eksperymentów można wyciągnąć kilka interesujących wniosków. W rozwiązywaniu zadanego problemu najskuteczniejszy okazał się algorytm Deep Q-Network. Jest to zaskakujący wniosek, ponieważ algorytm ten jest prostszy od pozostałych algorytmów. Zmiany parametrów uczących miały wpływ na skuteczność przeprowadzanego procesu uczenia się. Algorytmy A2C i PPO w istotny sposób poprawiły swoje wyniki po 10-krotnym zwiększeniu współczynnika uczenia, natomiast pozostałe algorytmy działały podobnie lub nieznacznie gorzej. Każdy z algorytmów w pewnym stopniu nauczył się omijać przeszkody, jednak poza DQN, inne robiły to z małą skutecznością.

5. Podsumowanie

Celem projektu było zaprezentowanie możliwości implementacji algorytmów głębokiego uczenia się ze wzmocnieniem w grach komputerowych, z wykorzystaniem silnika Unreal Engine. Używając wtyczki MindMaker w silniku Unreal Engine, zaprojektowałem agentów realizujących 5 różnych algorytmów DRL: DQN, A2C, ACER, ACKTR i PPO, oraz prostą grę typu *tor przeszkód*, którą agenci uczyli się przechodzić. Następnie dokonałem analizy i porównania wyników procesu uczenia się pod względem używanego algorytmu. Cel pracy uważam za zrealizowany.

Efektem pracy jest projekt w środowisku Unreal Engine 4 w wersji 4.27 o nazwie *MindMakerDRLStockBot*, zawierający poziom z torem przeszkód oraz klasę awatara o nazwie *Cart_BP*, będącą moim rozszerzeniem prototypu klasy *MindMakerActor_BP*, która jest częścią biblioteki MindMaker. Projekt zajmuje niecałe 6GB miejsca na dysku i może być uruchomiony na dowolnym urządzeniu, które posiada zainstalowany i zaktualizowany silnik Unreal Engine 4 w wersji 4.26 lub wyższej.

Za wkład własny uważam projekt, będący efektem pracy, poziom z torem przeszkód i mechanikę gry oraz w szczególności mój sposób konfiguracji i implementację do projektu wtyczki MindMaker, wraz ze wszystkimi modyfikacjami prototypów klas oferowanymi przez to rozwiązanie.

W czasie realizacji napotkałem kilka problemów. Szczególnym wyzwaniem okazała się implementacja wtyczki MindMaker. Zintegrowanie wszystkich metod i elementów dostępnej klasy wymagało dużego nakładu pracy i przyswojenia wielu technicznych informacji. Szczególnie dumny jestem z zaproponowanego przeze mnie systemu wykonywania obserwacji. Utworzony przeze mnie system jest skuteczny, co było kluczowe dla poprawnego działania procesu uczenia się.

Dostrzegam kilka możliwości dalszego rozwoju projektu. Ponieważ projekt skupiał się przede wszystkim na kwestii algorytmów uczenia się, sama gra przyjęła prostą formę, bez dodatkowych warunków i urozmaiceń. Można by rozwinąć poziom: wydłużyć go, dodać ruchome przeszkody oraz interakcję z awatarem, np. przyciski, na które awatar musiałby wejść, aby otworzyć sobie przejście do dalszej części poziomu. Awatar mógłby dostać dodatkową akcję - skok, czyniący go bardziej mobilnym. Takie zmiany wymagałyby także zmian konfiguracji agenta – formatu odbieranych obserwacji, kształtu przestrzeni akcji, wartości przydzielanych nagród itd. Możliwe, że w takiej wersji gry algorytmy wykazałyby inną skuteczność i wymagałyby innego doboru parametrów.

Inną możliwością rozbudowy projektu byłoby dodanie do gry innych poziomów, reprezentujących inne wyzwania, przed którymi mieliby stanąć agenci. Bieg na czas, trafianie do celu lub przechodzenie labiryntu – wszystkie te dyscypliny byłyby możliwe do zrealizowania w silniku Unreal Engine. Wykorzystani w projekcie agenci, po dostosowaniu ich do nowego założenia gry, również byliby zdolni do nauczenia się takiej formy zasad.

Trzecim pomysłem na dalszy rozwój projektu jest wzbogacenie go o drugą platformę, np. silnik do gier Unity. [21] Stworzony do projektowania prostych gier, takich jak ta zaprezentowana w tej pracy, silnik Unity idealnie nadawałby się do stworzenia analogicznego

projektu i porównania możliwości obu środowisk. Unity posiada również własną bibliotekę, dedykowaną do implementacji w silniku agentów głębokiego uczenia się ze wzmocnieniem - ML_Agents. [22] Pozwala ona na wykorzystanie algorytmów PPO i SAC, a więc byłaby to również możliwość porównania implementacji tych algorytmów przez oba środowiska.

Literatura

1. https://pl.wikipedia.org/wiki/Czwarta_rewolucja_przemys%C5%82owa - dostęp z dnia 18.01.2023
2. https://www.ibm.com/cloud/learn/neural-networks - dostęp z dnia 18.01.2023
3. Ian Millington, John Funge: Artificial Inteligence for Games. Morgan Kaufmann Publishers, Burlington, 2018.
4. https://docs.unrealengine.com/4.27/en-US/Basics/GettingStarted/ - dostęp z dnia 18.01.2023
5. https://github.com/krumiaa/MindMaker - dostęp z dnia 18.01.2023
6. https://en.wikipedia.org/wiki/Artificial_neural_network - dostęp z dnia 18.01.2023
7. https://miroslawmamczur.pl/jak-dzialaja-konwolucyjne-sieci-neuronowe-cnn/ - dostęp z dnia 18.01.2023
8. https://towardsdatascience.com/lstm-recurrent-neural-networks-how-to-teach-a-network-to-remember-the-past-55e54c2ff22e - dostęp z dnia 18.01.2023
9. https://www.mathworks.com/help/reinforcement-learning/ - dostęp z dnia 18.01.2023
10. https://en.wikipedia.org/wiki/Q-learning - dostęp z dnia 18.01.2023
11. https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/ - dostęp z dnia 18.01.2023
12. https://medium.com/intro-to-artificial-intelligence/soft-actor-critic-reinforcement-learning-algorithm-1934a2c3087f - dostęp z dnia 18.01.2023
13. https://towardsdatascience.com/advantage-actor-critic-tutorial-mina2c-7a3249962fc8 - dostęp z dnia 18.01.2023
14. Ziyu Wang, Victor Bapst: Sample Efficient Actor-Critic with Experience Replay. ICLR 2017
15. https://towardsdatascience.com/trust-region-policy-optimization-trpo-explained-4b56bd206fc2 - dostęp z dnia 18.01.2023
16. https://towardsdatascience.com/proximal-policy-optimization-ppo-explained-abed1952457b - dostęp z dnia 18.01.2023
17. James D. Ivory: A Brief History of Video Games. The Video Game Debate. Routledge, New York, 2016.
18. Treanor, Mike: AI-based game design patterns. American University, 2015.
19. Penelope Sweetser, Janet Wiles: Current AI in Games: A Review. Queensland University of Technology, Brisbane, 2002.
20. https://en.wikipedia.org/wiki/Endless_runner - dostęp z dnia 18.01.2023
21. https://docs.unity3d.com/Manual/index.html - dostęp z dnia 18.01.2023
22. https://github.com/Unity-Technologies/ml-agents - dostęp z dnia 18.01.2023