Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Kierunek: **INF-PPT**
Specjalność: **-**

Praca Dyplomowa Inżynierska

Bot do komputerowej gry wyścigowej

Kamil Matejuk

Opiekun pracy
Dr Marcin Michalski

Słowa kluczowe: Bot, Uczenie maszynowe, Uczenie przez wzmacnianie

Wrocław 2022

Streszczenie

W poniższej pracy został opisany proces implementacji bota poruszającego się w środowisku gry wyścigowej. Gra pozwala na proceduralne generowanie tras i terenów, co pozwoliło stworzyć nieskończoną ilość środowisk testowych. Do treningu bota wykorzystano metodę uczenia maszynowego przez wzmacnianie. W opracowaniu przeanalizowane zostały różne rodzaje obserwacji, akcji oraz nagród. Finalnie bot został wytrenowany na podstawie danych o odległości pojazdu od ewentualnych przeszkód, symulując metodę używaną na co dzień w pojazdach autonomicznych - tzw. lidar [1].

Abstract

The following thesis describes a process of implementing racing bot, which has been taught to navigate racing track environment. Game allows user to procedurally create track and terrain, as well as play created levels. The bot has been trained using reinforcement learning. This thesis analyzes different observations, action types and reward functions, to finally use distance-based observations. This method is similar to how autonomous vehicles operate in real world - using lidar technology [1].

Spis treści

Spis rysunków

Spis tablic

List of Algorithms

Wstęp

W obecnych czasach obserwujemy coraz bardziej dynamiczny rozwój branży pojazdów wyposażonych w sterowanie autonomiczne. Autonomia pojazdu określana jest w skali pięciostopniowej, gdzie stopień 0 oznacza brak autonomii, a stopień 5 - pełną automatyzację kierowania pojazdem [3]. Aktualnie, jako ludzkość, jesteśmy w stanie osiągnąć autonomię poziomu 2, co oznacza, że kierowca musi nieustannie czuwać nad decyzjami samochodu, a software ma prawo nie rozumieć skrzyżowań bez sygnalizacji świetlnej czy pojazdów uprzywilejowanych.

Oprogramowanie takich pojazdów często testowane jest w symulowanym i bezpiecznym środowisku, które jednak twórcy samochodów starają się jak najbardziej upodabniać do świata realnego. Pojazdy te wykorzystują m.in. sensory lidar[1] oraz kamery jako główne źródło informacji. Następnie na podstawie tych danych i z wykorzystaniem algorytmów uczenia maszynowego poddawane są treningowi.

W poniższej pracy zostało zasymulowane środowisko naturalne oraz pojazd, wraz z zachowanymi w znacznym stopniu prawami fizyki oraz sposobem sterowania. Metody obserwacji otoczenia inspirowane były metodami wykorzystywanymi w rzeczywistości. Pojazd został wytrenowany do poruszania się po torze w różnych środowiskach.

W pierwszym rozdziale przedstawiony został proces tworzenia gry jako proceduralnego środowiska do testów. Kolejny rozdział zawiera informacje o wyborze optymalnych parametrów ze względu na rozmiar sieci, czas uczenia oraz inferencji. Finalnie przedstawiony został proces trenowania bota.

Rozdział 1

Stworzenie gry

1.1 Cel i funkcjonalności

Gra została stworzona przede wszystkim jako środowisko treningowe oraz testowe bota, dopiero w drugiej kolejności jako rozrywka dla użytkownika. Dostarczona w dodatku *A* gra pozwala graczowi na załadowanie jednego z domyślnych terenów, bądź stworzenie własnego terenu i przetestowanie go w grze. Celem rozgrywki jest przetransportowanie pojazdu od miejsca startowego do mety, nie zjeżdżając ze ścieżki. Gra kończy się po wykonaniu jednego pełnego okrążenia. Gracz konkuruje z 3 innymi pojazdami, sterowanymi przez wytrenowanego bota. Dodatkowo gracz posiada możliwość stworzenia własnego terenu i trasy, sterując kilkoma parametrami. W ten sam sposób zostały przygotowane tereny, na których został wytrenowany bot.

1.2 Wykorzystane technologie

Gra została stworzona z wykorzystaniem silnika do gier Unity Engine 3D. Jest to jeden z dwóch najpopularniejszych silników do tworzenia gier, poza Unreal Engine, natomiast posiada on niższą barierę wejścia. Wykorzystanie Unity pozwoliło na zasymulowanie praw fizyki występujących przy prowadzeniu pojazdu.

1.3 Stworzenie trasy 2D

Pierwszym etapem generowania terenu jest wybór krzywej opisującej trasę. Każda wygenerowana trasa jest krzywą zamkniętą. Składa się ona z segmentów, których ilość jest bezpośrednio powiązana z zawiłością trasy.

Aby wybrać końce segmentów, losowane są punkty $P = (P_x,P_y)$ na płaszczyźnie z uniwersalnym prawdopodobieństwem, z zakresu $P_x \in [-1,1]$, $P_y \in [-1,1]$, upewniając się że żadne 2 punkty nie są za blisko siebie.

Algorithm 1.1: Wybór punktów trasy

```
   Data:
   n ← ilość segmentów;
   d ← minimalna odległość między punktami;
 1 positions ← [];
 2 while positions.Length < n do
 3 |   pNew ← new point;
 4 |   pNew.x ← random(−1, 1);
 5 |   pNew.y ← random(−1, 1);
 6 |   tooClose ← false;
 7 |   foreach P ∈ positions do
 8 |   |   if distance(P, pNew) < d then
 9 |   |   |   tooClose ← true;
10 |   |   |   break;
11 |   if not tooClose then
12 |   |   add pNew. to positions;
```

Po wyborze odpowiedniej ilości punktów, sortowane są one zgodnie z ruchem wskazówek zegara, na podstawie kąta pomiędzy wektorem $[0, 1]$, a wektorem kończącym się w danym punkcie.

Algorithm 1.2: Wyznaczenie kąta pomiędzy punktem a prostą OY

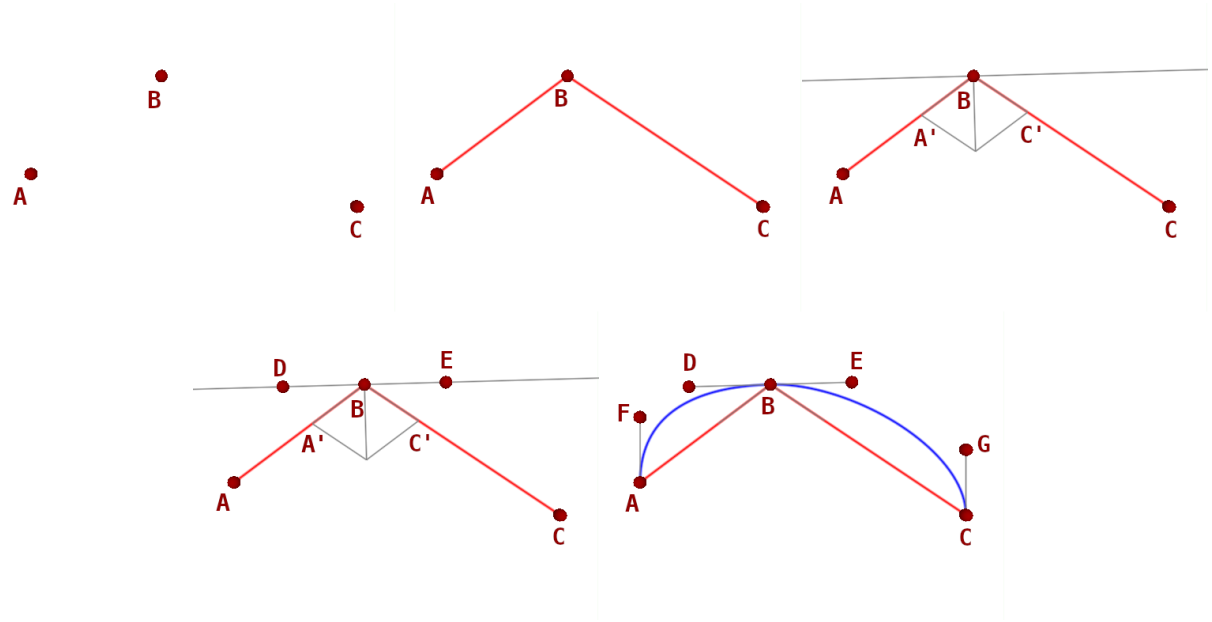
```
   Data: P ← punkt
 1 forwardVector ← [0, 1].normalized;
 2 pointVector ← [P.x, P.y].normalized;
 3 angle ← acos(dot(forwardVector, pointVector)) ;   /* Arcus cosinus iloczynu skalarnego */
 4 sign ← cross(forwardVector, pointVector).z ;      /* iloczyn wektorowy */
 5 if sign ⩾ 0 then
 6 |   return angle;
 7 else
 8 |   return −1 · angle;
```

Następnie trasa w każdym segmencie wyznaczana jest za pomocą krzywej Bezier 3 stopnia, ponieważ pozwala ona w stosunkowo prosty sposób zapisać stosunkowo komplikowane krzywe, utrzymując płynny kształt. Każda krzywa Bezier składa się z punktu startowego P_0, końcowego P_3, oraz dwóch punktów kontrolnych P_1, P_2, oraz wyraża sie wzorem:

$$b(t) = (1-t)^3 * P_0 + 3t(1-t)^2 * P_1 + 3t^2(1-t) * P_2 + t^3 * P_3 \qquad dla\ t \in [0, 1]$$

Aby zapewnić płynne połączenia pomiędzy segmentami, sąsiednie punkty kontrolne powinny znajdować sie na tej samej prostej. Dla każdego punktu startowego wyznaczane są punkty kontrolne segmentu poprzednioego i następnego, na podstawie sąsiadujących punktów startowych.

Poniższy rysunek przedstawia trzy wylosowane punkty A,B,C, oraz proces generacji punktów kontrolnych dla środkowego punktu B. Na początku wyznaczana jest prosta prostopadła do dwusiecznej kąta pomiędzy wektorami BA i BC, poprzez normalizację ww. wektorów odpowiednio do BA' i BC', a następnie odjęcie powstałych wektorów. Punkt kontrolny D znajduje się na ww. prostej, w odległości równej połowie długości wektora BA. Analogicznie położony jest punkt kontrolny E. Po wykonaniu powyższych operacji dla każdego punktu startowego na wyznaczonej trasie, w każdym segmencie znajdują się oba punkty kontrolne. Po podstawieniu punktów startowych i kontrolnych (np. A,F,D,B) dla każdego segmentu wyznaczona zostanie cała trasa.



Rysunek 1.1: Generowanie punktów kontrolnych krzywej Bezier

1.4 Stworzenie terenu 3D

Każdy stworzony teren gry jest rozmiaru $512x512$, z czego 10% przeznaczone jest na marginesy z każdej strony, co pozostawia wewnętrzną przestrzeń $409x409$ na wygenerowany tor. Każdy punkt wygenerowanej pętli jest skalowany odpowiednio, aby trasa wypełniła całość dostępnej przestrzeni. Następnie dla każdego punktu terenu generowana jest wysokość, zdefiniowana jako procent maksymalnej wysokości 128. Wysokość składa się z trzech warstw, wykorzystujących Szum Perlina, każda coraz bardziej szczegółowa. Implementacja Szumu Perlina została dostarczona jako fragment biblioteki $Mathf$ przez Unity [2].

Algorithm 1.3: Wyznaczenie wysokości terenu

Data:
$x \leftarrow$ *współrzędna X punktu*;
$y \leftarrow$ *współrzędna Y punktu*;
1 $x \leftarrow x + offsetX$;
2 $y \leftarrow y + offsetY$;
3 $h1 \leftarrow detailsMain \cdot PerlinNoise(x \cdot scaleMain, y \cdot scaleMain)$;
4 $h2 \leftarrow detailsMinor \cdot PerlinNoise(x \cdot scaleMinor, y \cdot scaleMinor)$;
5 $h3 \leftarrow detailsTiny \cdot PerlinNoise(x \cdot scaleTiny, y \cdot scaleTiny)$;
6 **return** *h1 + h2 + h3*

Powyższa funkcja pozwala na sterowanie krztałtem terenu poprzez parametry *details* i *scale* dla każdej z trzech warstw *main*, *minor*, *tiny*. Warstwa *main* definiuje ogólny krztałt terenu, warstwa *minor* dodaje mniejsze pagórki, natomiast warstwa *tiny* dodaje drobne detale. Dodatkowo poprzez sterowanie $offsetX$ i $offsetY$ możliwe jest przesunięcie wszystkich wartości w płaszczyźnie poziomej.

1.5 Przypisanie tekstur

Zależnie od rodzaju terenu, wybierany jest odpowiedni zestaw tekstur. Dla każdego punktu wygenerowanego terenu przypisane są odpowiednie wartości przezroczystości tekstur, tworząc jak najbardziej realistyczne krajobrazy. Decyzja o wybraniu przezroczystości tekstur została podjęta na podstawie rodzaju terenu wybranego przez użytkownika, wysokości danego punktu, kierunku płaszczyzny zbocza, kąta

nachylenia zbocza oraz odległości punktu od centrum wygenerowanej trasy. Odczytanie wartości takich jak wysokość, kierunek płaszczyzny i kąt nachylenia można wykonać w stałym czasie $O(1)$. Natomiast znalezienie najbliższego punktu trasu jest bardziej złożonym procesem. Poniżej przedstawiono proces wyboru metody jego wyznaczania.

1.5.1 Znalezienie najbliższego punktu krzywej Bezier

Poniższa metoda dzieli każdy segment trasy na m odcinków, a następnie iteruje wszystkie końce odcinków we wszystkich segmentach. Optymalna ilość podziału wyznaczona została eksperymentalnie. Metoda ta ma złożoność czasową $O(nm)$, gdzie n oznacza liczbę segmentów trasy, natomiast m oznacza dokładność podziałów danego segmentu.

Algorithm 1.4: Znalezienie najbliższego punktu krzywej Bezier

```
   Data: P ← cel poza krzywą
 1 minDist ← infinity;
 2 minT ← 0;
 3 for i = 0; i ¡ 100; i++ do
 4 |   t ← i · 0.01;
 5 |   pProjection ← B(t) ;                    /* punkt w t% długości krzywej bezier */
 6 |   dist ← distance(P, pProjection);
 7 |   if dist ¡ minDist then
 8 |   |   minDist ← dist;
 9 |   |   minT ← t;
10 return B(minT)
```

1.5.2 Udoskonalone znalezienie najbliższego punktu krzywej Bezier

Poniższa metoda wywodzi się z artykułu [5], oraz została przystosowana dla krzywej Bezier 3 stopnia. Funkcja opisująca krzywą Bezier:

$$b(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3$$

Pierwsza pochodna po t:

$$\frac{b}{dt}(t) = -3(1-t)^2 P_0 + 3(1-t)^2 P_1 - 6t(1-t)P_1 - 3t^2 P_2 + 6t(1-t)P_2 + 3t^2 P_3$$

Aby znaleźć rzut celu na krzywą, zależy znaleźć taki punkt na krzywej, aby styczna w tym punkcie była prostopadła do odcinka łączącego go z celem. Zatem dla zadanego punktu celu C i szukanej wartości t, określającej w którym miejscu krzywej znajduje się szukany rzut, zdefiniowana została funkcja f:

$$f(t) = (C - b(t)) \cdot \frac{b}{dt}(t)$$

Funkcja f wykorzystuje iloczyn skalarny, zatem szukane t zwraca wartość $f(t) = 0$. Ze względu na kształt krzywej, może na niej znajdować się kilka punktów będących rzutami prostopadłymi celu C, natomiast poszukiwany jest punkt najbliżej celu, zatem funkcję f należy przemnożyć przez odległość od celu do znalezionego punktu.

$$f(t) = \left[(C - b(t)) \cdot \frac{b}{dt}(t)\right] distance(C, b(t))$$

Aby znaleźć najbliższy punkt na krzywej, należy znaleźć miejsce zerowe funkcji $f(t)$. W tym celu wykorzystano zmodyfikowaną metodę bisekcji. Tradycyjnie metoda bisekcji zakłada, że wartości funkcji na obu krańcach przedziału są przeciwnego znaku, następnie wybiera środkowy punkt i kontynuuje przeszukiwanie już w połowie przedziału. Natomiast w tym wypadku nie jest zagwarantowane, że funkcja będzie

miała przeciwne znaki na końcach przedziału, dlatego gdy są tego samego znaku, liczona jest bisekcja dla obu podprzedziałów.

Algorithm 1.5: Udoskonalona bisekcja

```
   Data:
   a ← początek przedziału;
   b ← koniec przedziału;
   delta ← dokładność argumentów;
   epsilon ← dokładność wyników;
   maxit ← maksymalna ilość iteracji;
 1 if maxit < 1 then
 2     return (a+b)/2
 3 u ← f(a);
 4 v ← f(b);
 5 e ← (b − a)/2;
 6 c ← a + e;
 7 w ← f(c);
 8 if sing(u) != sign(v) then
 9     if abs(e) < delta or abs(w) < epsilon then
10         return c
11     if sing(u) != sign(w) then
12         return UdoskonalonaBisekcja(a, c, delta, epsilon, maxit - 1)
13     else
14         return UdoskonalonaBisekcja(c, b, delta, epsilon, maxit - 1)
15 else
16     left ← UdoskonalonaBisekcja(a, c, delta, epsilon, maxit/2);
17     right ← UdoskonalonaBisekcja(c, b, delta, epsilon, maxit/2);
18     if abs(f(left)) < abs(f(right)) then
19         return left
20     else
21         return right
```

1.5.3 Porównanie metod

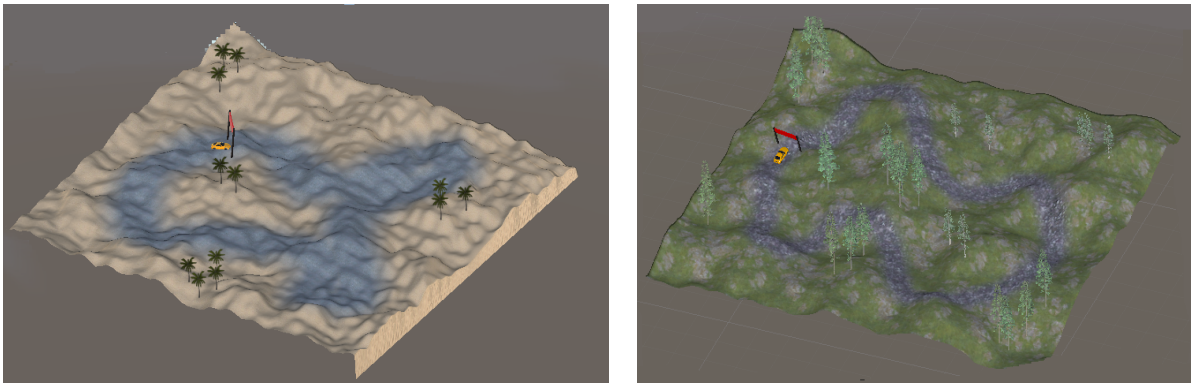
Metoda druga jest wykonywana szybciej, dodatkowo zapewniając większą szczegółowość.

Method 1		Method 2			lp
ilość podziałów	czas [ms]	ilość iteracji	odpowiadająca ilość podziałów	czas [ms]	
50	42046	5	$2^5 = 32$	30580	1
100	81421	10	$2^{10} = 1024$	59606	2
150	121224	15	$2^{15} = 32768$	70178	3

Tablica 1.1: Porównanie metod rzutowania punktu na krzywą Bezier

1.6 Dodanie obiektów

Ostatnim etapem jest dodanie obiektów, takich jak meta oraz pojazdy. Domyślnie na każdym torze znajdą się cztery pojazdy, ustawione jeden za drugim, z czego trzy sterowane przez bota, a jeden przez gracza. Dodatkowo dodawane są krawędzie, których celem jest zapobieganie spadania pojazdu poza wygenerowany teren. W celu śledzenia przebiegu trasy, w równych odstępach dodawane są punkty kontrolne. Przykładowo otrzymano poniższy efekt:



Rysunek 1.2: Wygenerowane tereny

Rozdział 2

Wytrenowanie bota

2.1 Cel i funkcjonalności

Wytrenowany bot powinien być w stanie poruszać się po wygenerowanym wcześniej terenie, rozpoznając i trzymając się drogi. Trening bota powinien wykorzystywać metody uczenia przez wzmacnianie.

2.2 Wykorzystane technologie

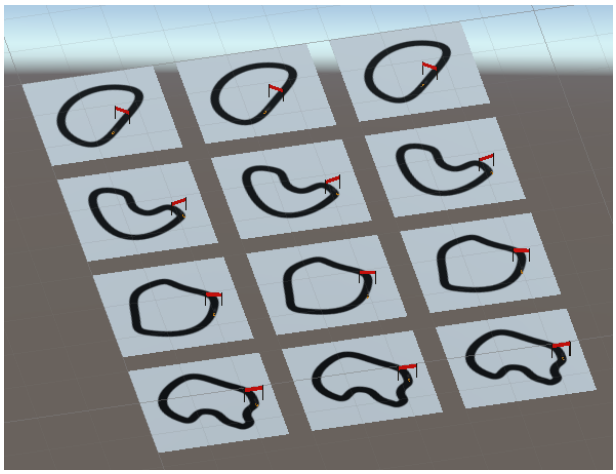
Silnik Unity udostępnia dodatek open-source ML-Agents [4] [6], który pozwala na wykorzystanie środowisk utworzonych w grze do treningu modeli. Stanowi on warstwę łączącą Unity z biblioteką Tensorflow, oraz zapewnia wiele nowoczesnych algorytmów głębokiego uczenia przez wzmacnianie (m.in. Proximal Policy Optimization (PPO), Soft Actor-Critic (SAC), self-play). W powyższej pracy wykorzystano algorytm PPO, ze względu na większą stabilność treningu [7]. Śledzenie wielu metryk postępu uczenia było dostępnych z wykorzystaniem programu tensorboard.

2.3 Uczenie przez wzmacnianie

Uczenie przez wzmacnianie jestem jednym z trzech głównych nurtów uczenia maszynowego, gdzie agent uczy się polityki optymalnej w danym środowisku metodą prób i błędów, otrzymując wyłącznie wartość nagrody jako informację zwrotną. W przeciwieństwie do uczenia nadzorowanego, metoda ta nie wymaga wcześniejszego przygotowania dużej ilości opisanych danych. Pozwala to na wprowadzenie bota do nieznanego środowiska i natychmiastowe podjęcie interakcji.
Cykl uczenia przez wzmacnianie opiera się na akcji bota i reakcji. Bot zbiera obserwacje na podstawie stanu w jakim się znajduje w środowisku. Następnie podejmuje decyzję na podstawie obserwacji. Po podjęciu decyzji wykonywana jest odpowiednia akcja, po czym za akcję przyznawana jest nagroda lub kara, w zależności czy akcja wpłynęła pozytywnie na przybliżanie się bota do celu. Na podstawie zbioru informacji zawierającego stan początkowy, podjęte akcje i otrzymaną nagrodę trenowana jest polityka, maksymalizując oczekiwaną nagrodę.

2.4 Metodyka uczenia

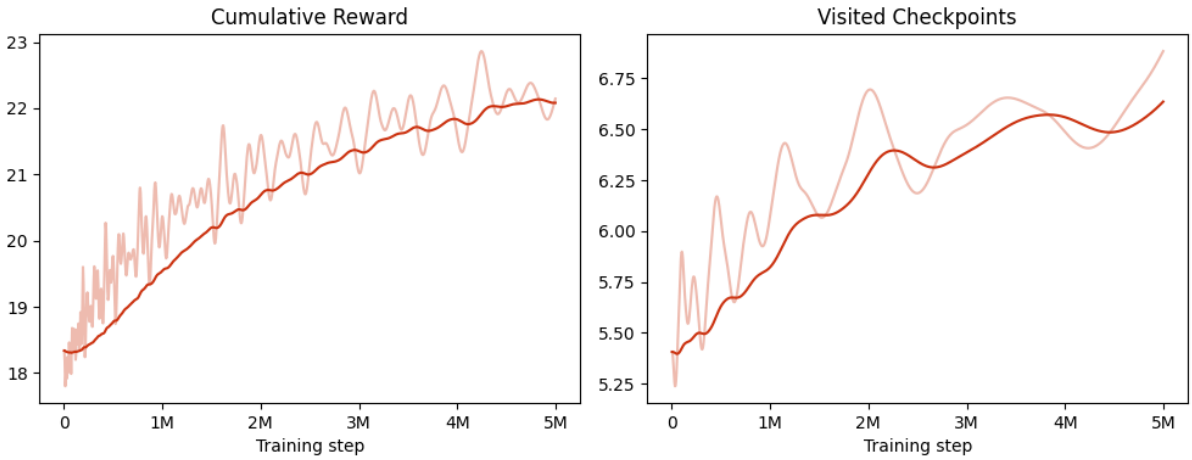
Wstępne uczenie odbywa się na terenach płaskich (wysokość każdego punktu jest równa zero) oraz w terenie ze zwiększonym kontrastem - droga jest czarna a wszystko pozostałe jest białe. Do treningu uruchomione jest naraz 12 terenów, zawierających 4 różne trasy. Na każdym z terenów znajduje się pomiędzy 1 a 4 pojazdy, co oznacza że model jest trenowany na 12-48 niezależnych instancjach naraz, w celu przyspieszenia nauki. W kolejnych etapach zwiększany jest stopień trudności trasy, reprezentowany przez ilość i ostrość zakrętów. Następnie zwiększane jest pofałdowanie terenu, oraz zmieniane są tekstury drogi i terenu.



Rysunek 2.1: Prowadzenie wielu treningów na raz

2.5 Analiza wyników

Wyniki uczenia przedstawione zostały na wykresie za pomocą dwóch metryk. Wartość *Cumulative Reward* oznacza średnią wartość nagrody z jednego epizodu dla jednego agenta. Wartosć *Visited punkt kontrolnys* informuje jak dużą część trasy przeszedł agent, uśrednioną z jednego epizodu dla jednego agenta. Pełne okrążenie zaweira w sobie 40 punktów kontrolnych. Wartości zaznaczone na wykresach bledszym kolorem są wartościami rzeczywistymi, natomiast wartości pogrubione wyliczone zostały z wykorzystaniem średniej kroczącej eksponencjalnej, aby lepiej uwidocznić tendencje zmian.



Rysunek 2.2: Przykładowa analiza wyników

2.5.1 Wybór obserwacji

Wybór informacji, jakie dostępne są dla bota znacznie wpłynie na podejmowane przez niego decyzje. Za mała ilość informacji nie pozwoli na wykrycie odpowiedniej ilości szczegółów trasy, natomiast za duża znacznie zwiększy czas treningu i inferencji, wprowadzając niepotrzebny szum.

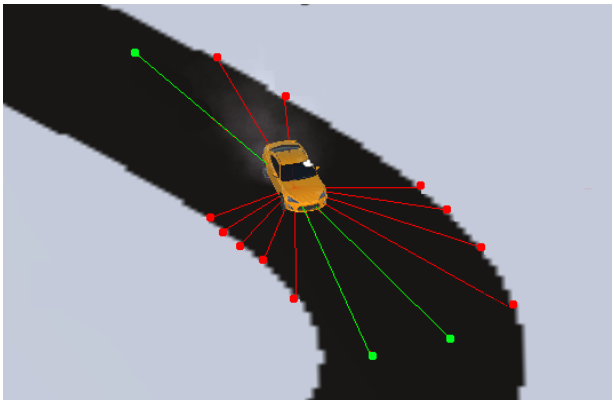
Dane wynikające z trasy

Pierwszy sposób obserwacji opiera się na wartościach, do których zwykły gracz nie ma dostępu, ponieważ wymagają m.in. znajomości dokładnego wzoru krzywej trasy do wyliczenia. Dane, które są przekazywane do modelu jako obserwacje to:

- odległość w linii prostej od aktualnej pozycji do centrum szerokości trasy
- kąt pomiędzy kierunkiem jazdy pojazdu a kierunkiem stycznej do toru
- odległość w linii prostej od aktualnej pozycji do najbliższego punktu kontrolnego
- kąt pomiędzy kierunkiem jazdy pojazdu a kierunkiem do najbliższego punktu kontrolnego
- kąt nachylenia terenu
- aktualna pozycja kół (-1 oznacza maksymalnie skręcone w lewo, 1 maksymalnie w prawo)
- aktualny stopień wciśnięcia pedału gazu (-1 oznacza jazdę do tyłu, 1 jazdę do przodu)
- aktualna prędkość

Dane odległości od krawędzi

Podobnie jak większość samochodów autonomicznych, poniższa metoda wykorzystuje wizualizację przestrzeni na podstawie odległości, nazywaną lidar [1]. Technologia ta w realnym świecie buduje model otoczenia wysyłając wiązki laserowe i mierząc trasę przebytą przez nie do przeszkody. Analogicznie, symulowany pojazd testuje odległości od krawędzi trasy oraz ewentualnych przeszkód, takich jak inne pojazdy.

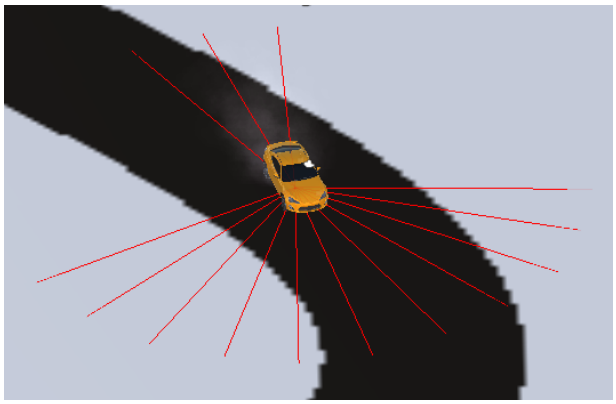


Rysunek 2.3: Obserwacje bota dystansów na około pojazdu

Dodatkowo przekazywana jest aktualna prędkość pojazdu jako obserwacja, zważywszy że nie da się jej w żaden sposób odczytać z jednej klatki pomiarów.

Dane wizualne jednowymiarowe

Poniższe podejście testuje metodę obserwacji wykorzystującą bodźce wizualne. W tym przypadku tworzona jest lista wartości kolorów dla każdego punktu w stałej odległości od pojazdu. Generuje to okrąg kolorów na około pojazdu, który powinien być w stanie przekazać takie informacje jak zakręty na trasie, będąc znacznie mniejszego rozmiaru niż obraz z kamery. Na rysunku zaznaczono kilka promieni pobierających kolory otoczenia.

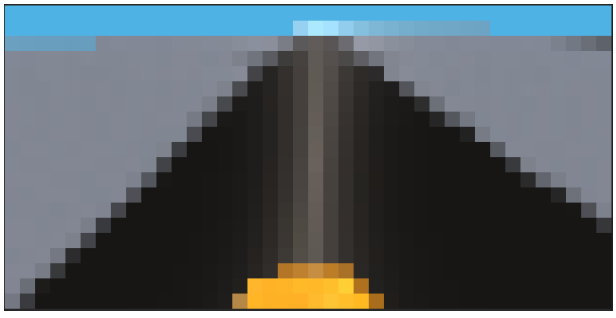


Rysunek 2.4: Obserwacje bota kolorów na około pojazdu

Dodatkowo przekazywana jest aktualna prędkość pojazdu jako obserwacja, zważywszy że nie da się jej w żaden sposób odczytać z danych wizualnych.

Dane wizualne z kamery przedniej

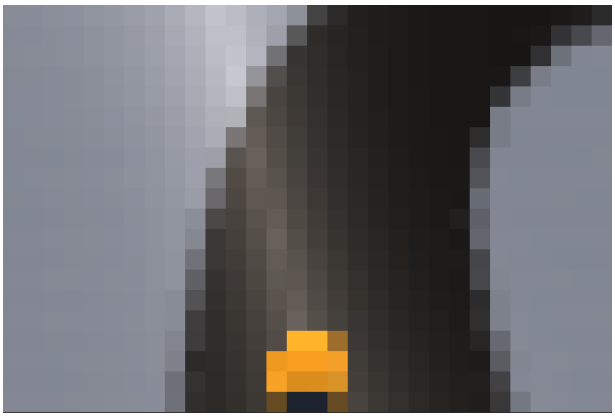
Wykorzystując imformacje w postaci w jakiej są one widoczne dla każdego gracza, kolejna metoda pobiera obraz z kamery jako obserwacje. Kamera umiejscowiona jest na przedzie pojazdu, zapewniając perspektywę podobną do zwykłego prowadzenia auta. Aby ograniczyć ilość informacji dostarczanych do sieci, obraz z kamery skalowany jest do rozdzielczości $40x20$ oraz konwertowany na skalę szarości. Dodatkowo, jak powyżej, przekazywana jest aktualna prędkość pojazdu.



Rysunek 2.5: Obserwacje bota z kamery przedniej

Dane wizualne z kamery z lotu ptaka

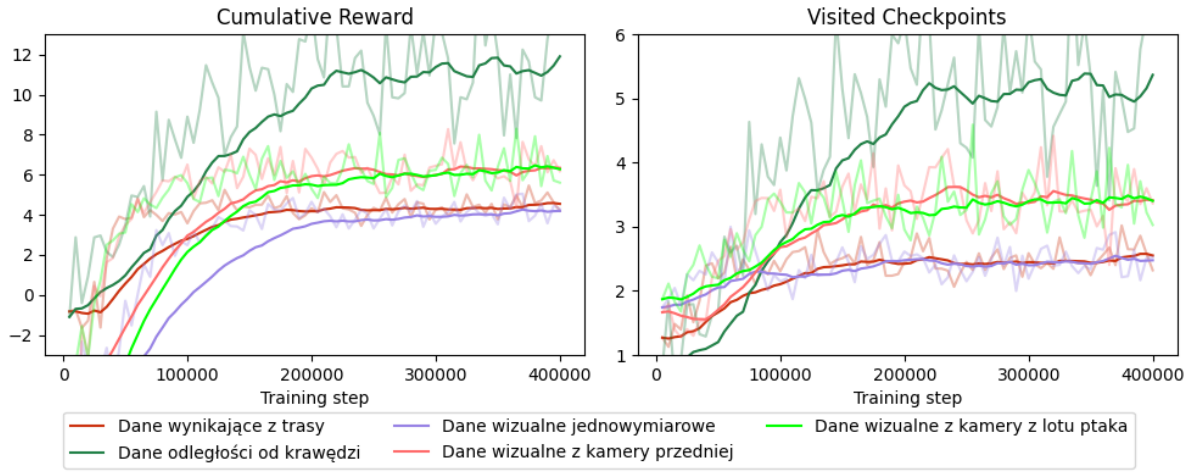
Wykorzystując imformacje widoczne przez kamerę z lotu ptaka model powinien być w stanie łatwiej zauważyć krzywiznę toru. Tak jak powyżej, obraz z kamery skalowany jest do rozdzielczości $30x20$ oraz konwertowany na skalę szarości, oraz przekazywana jest aktualna prędkość pojazdu.



Rysunek 2.6: Obserwacje bota z kamery z lotu ptaka

Porównanie

Poniżej znajdują się wykresy porównujące szybkość uczenia się w pierwszych 400 000 iteracjach. Wykres pierwszy pokazuje otrzymaną nagrodę za akcję, uśrednioną dla każdego agenta, dla każdej akcji. Drugi wykres przedstawia jak daleko udało się dojechać. Na przestrzeni całego toru rozłożone jest 40 punktów kontrolnych. Celem jest maksymalizacja obu wartości.



Rysunek 2.7: Porównanie metod obserwacji

Najlepsze wyniki otrzymano dla metody wykorzystującej odległości od przeszkód. Metoda ta zbiera łącznie 14 dystansów, co wystarcza do zawarcia najpotrzebniejszych informacji. Dane z kamery okazały się zbyt obszerne (odpowiednio 600 i 800 pikseli), spowalniając trening.

2.5.2 Wybór akcji

Po podjęciu decyzji bot musi wykonać odpowiednią akcję. Bot podejmuje decyzję w dwóch płaszczyznach. Pierwsza odnosi się do przemieszczania przód-tył, natomiast dróga odpowiada skręceniu kierownicy prawo-lewo. Decyzje te mogłyby być wybierane z przestrzeni ciągłych lub dyskretnych.

Akcje w przestrzeni ciągłej

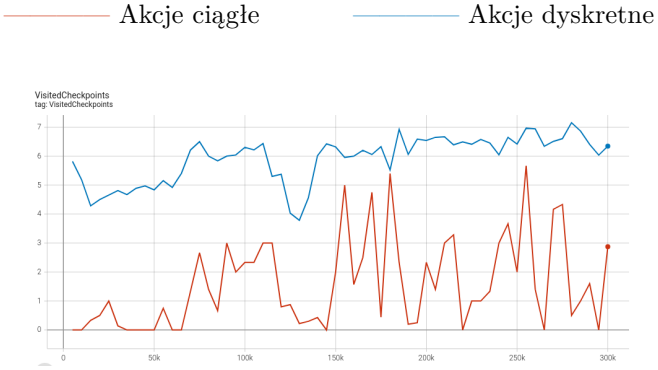
Dla każdej płaszczyzny bot zwraca wartość rzeczywistą z zakresu $[-1, 1]$. W pierwszej płaszczyźnie wartość odpowiada dokładnie stopniowi wciśnięcia pedału gazu, natomiast w drugiej - kątowi skrętu kierownicy. Poniważ bot jest w stanie zawsze ustawić konkretną wartość, przejścia pomiędzy kolejnymi akcjami nie muszą być płynne - w pierwszej klatce bot może skierować kierownicę 30 stopni w lewo, natomaist już klatkę później może ustawić ją na 20 stopni w prawo, bez stanów przejściowych.

Akcje w przestrzeni dyskretnej

Dla każdej płaszczyzny bot zwraca wartość całkowita ze zbioru $-1, 0, 1$. W pierwszej płaszczyźnie wartość -1 odpowiada zwiększeniu nacisku na pedał hamulca, 0 oznacza brak zmian, 1 oznacza zwiększenie nacisku na pedał gazu. Analogicznie w drugiej plaszczyźnie, -1 odpowiada skręceniu kierownicy mozniej w lewo, 1 mocniej w prawo, a 0 brak zmian. W ten sposób zmiany są bardziej płynne w czasie.

Porównanie

Zastosowanie akcji dyskretnych pozwoliło botowi na zancznie łatwiejsze poruszanie się w środowisku, co przełożyło sie na znacznie lepsze wyniki w pierwszych 300 000 krokach treningu.



Rysunek 2.8: Porównanie metod podejmowania akcji

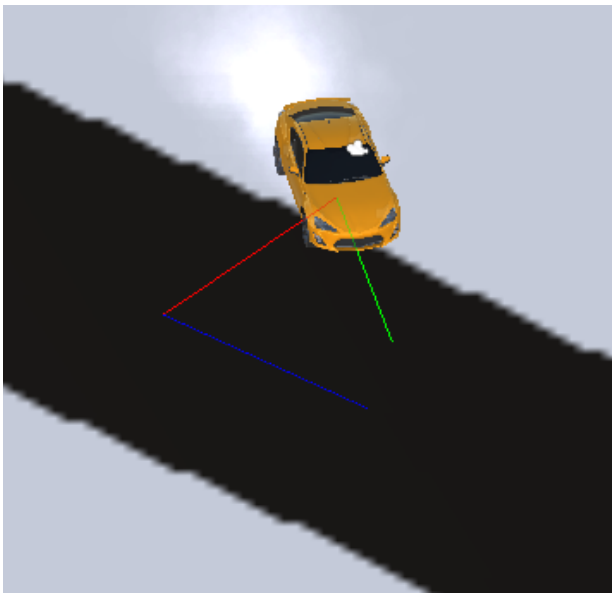
2.6 Zdefiniowanie funkcji oceny

Ocena powinna odzwierciedlać w jakim stopniu wykonana akcja przybliżyła bota do osiągnięcia celu. W każdym cyklu, podjęta akcja oceniana jest na podstawie trzech parametrów.

$$\begin{aligned} Q = &(0.5 - distanceToRoadCenter) \cdot 0.1 + \\ &(0.05 - abs(angleToTangent)) + \\ &(distanceTraveledInFrame - 0.1) + \\ &0.01 \end{aligned}$$

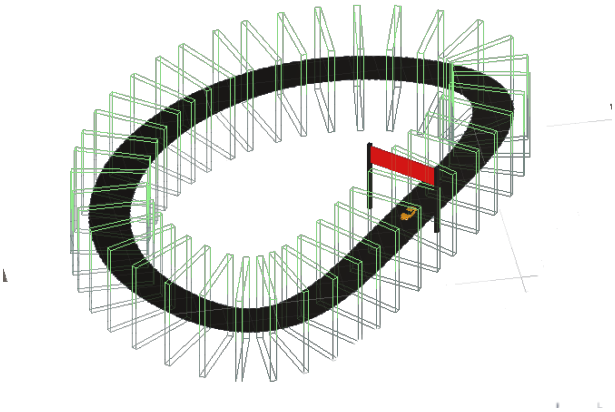
Po pierwsze przyznawana jest nagroda za utrzymywanie sie w odległości mniejszej niż 50% szerokości drogi od środka drogi (linia czerwona na poniższym rysunku), w przeciwnym przypadku kara, wprost proporcjonalna do ww. odległości. Po drugie przyznawana jest nagroda za utrzymywanie kierunku jazdy (linia zielona na poniższym rysunku) w zakresie ± 10 stopni od kierunku trasy (linia niebieska na poniższym rysunku), w przeciwnym przypadku kara, wprost proporcjonalna do w.w kąta. Po trzecie przyznawana jest

nagroda za dystans pokonany od ostatniej akcji, jeżeli wynosi co najmniej 0.1. Dodatkowo przyznawana jest nagroda za każdą klatkę, aby promować jak najdłuższe treningi.



Rysunek 2.9: Wizualizacja oceny akcji

Dodatkowo bot oceniany jest za każdym razem kiedy wejdzie w interakcję z innym obiektem. Na około całego terenu rozmieszczone są bariery, które powodują koniec epizodu po dotknięciu i powrót pojazdu na pozycję początkową, dodatkowo dodając karę 0.1. Na pierwszym etapie treningu takie same bariery ustawione są wzdłuż drogi, tak że pojazd kończy epizod za każdym razem kiedy zjedzie z drogi. Po wytrenowaniu bota w takich warunkach bariery te zostaną ściągnięte, aby zobaczyć czy mimo to bot będzie trzymał sie drogi.
Wzdłuż całej trasy rozstawione są punkty kontrolne, które pojazd powinien przebyć w odpowiedniej kolejności. Za każdy punkt kontrolny zaliczony w kolejności nagroda zwiększa się o 1. Jeżeli pojazd zahaczy o któryś z 4 sąsiednich punktów kontrolnych (2 w tył, 2 w przód) dostaje tylko karę 0.1, natomiast jeżeli dotknie innego punktu kontrolnego, epizod jest kończony z karą -1.



Rysunek 2.10: Rozmieszczenie punktów kontrolnych

2.7 Dobranie parametrów

Wybrany algorytm Proximal Policy Optimization pozwala na osiągnięcie nie najgorszych wyników, natomiast dostosowanie hyperparametrów potrafi znacznie poprawić efekty treningu.

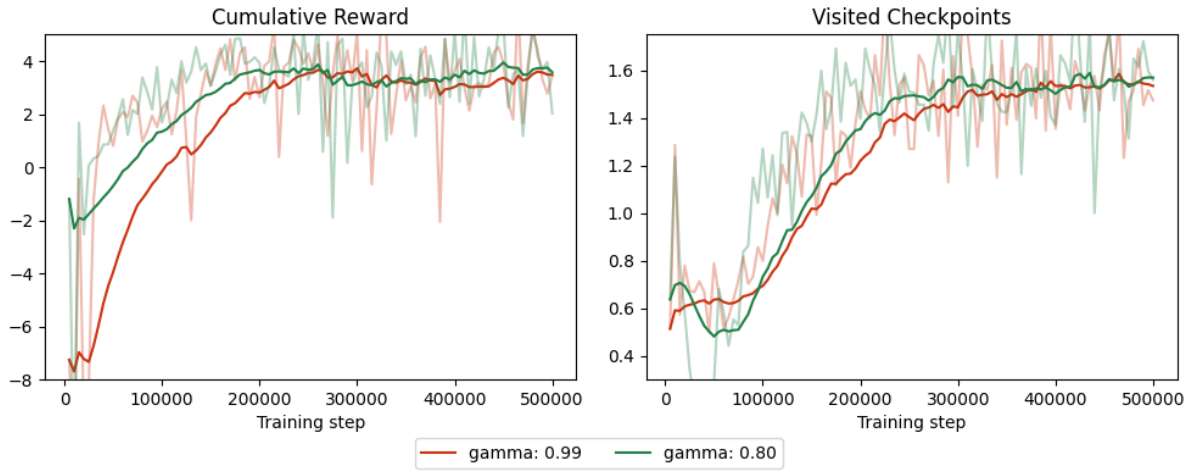
Algorithm 2.1: Hiperparametry początkowe treningu

```
1  hyperparameters :
2      batch_size : 1024
3      buffer_size : 10240
4      learning_rate : 0.0003
5      beta : 0.005
6      epsilon : 0.2
7      lambd : 0.95
8      num_epoch : 3
9      learning_rate_schedule : linear
10 reward_signals :
11     extrinsic :
12         gamma : 0.99
13         strength : 1.0
```

Poniżej zostały przedstawione wybory poszczególnych parametrów. Porównane zostały konfiguracje dla pierwszych 500000 iteracji, za każdym razem zaczynając od losowych wag. Po wyborze optymalnej wartości parametru, była ona aktualizowana i wykorzystywana w kolejnych testach jako domyślna.

2.7.1 gamma

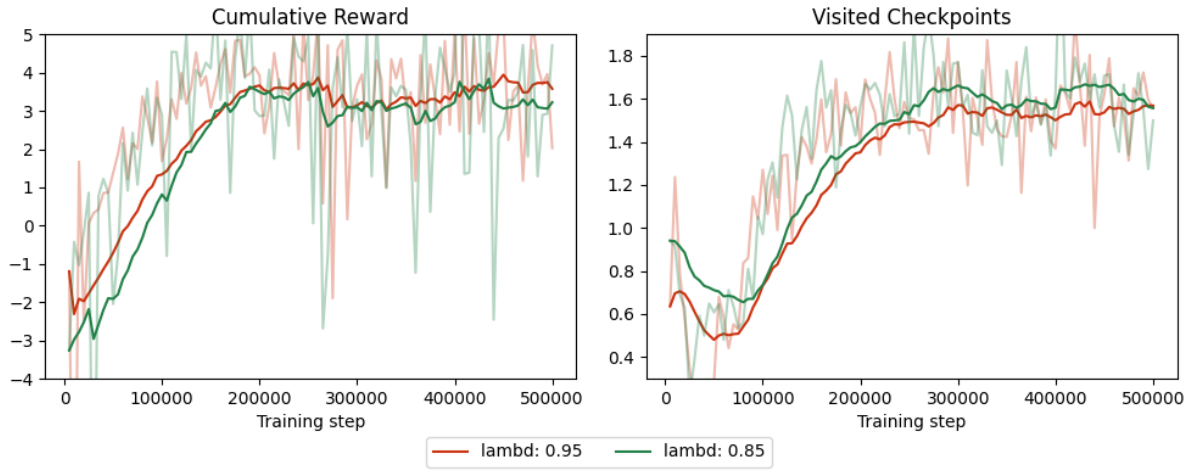
Parametr *gamma* można rozumieć jako jak daleko w przyszłość agent powinien dbać o możliwe nagrody. W sytuacjach gdy agent powinien podejmować decyzje w oczekiwaniu na przewidywane nagrody w przyszłości, wartość *gamma* powinna być większa (0.995), natomiast jeżeli agent powinien bardziej dbać o nagrody natychmiastowe wartość *gamma* może osiągać niższe wartości (0.8). Po porównaniu granic typowego zakresu wartości, na wykresach poniżej widać, że wartość niższa przekłada się na minimalnie lepszy trening.



Rysunek 2.11: Wybór hiperparametrów: gamma

2.7.2 lambda

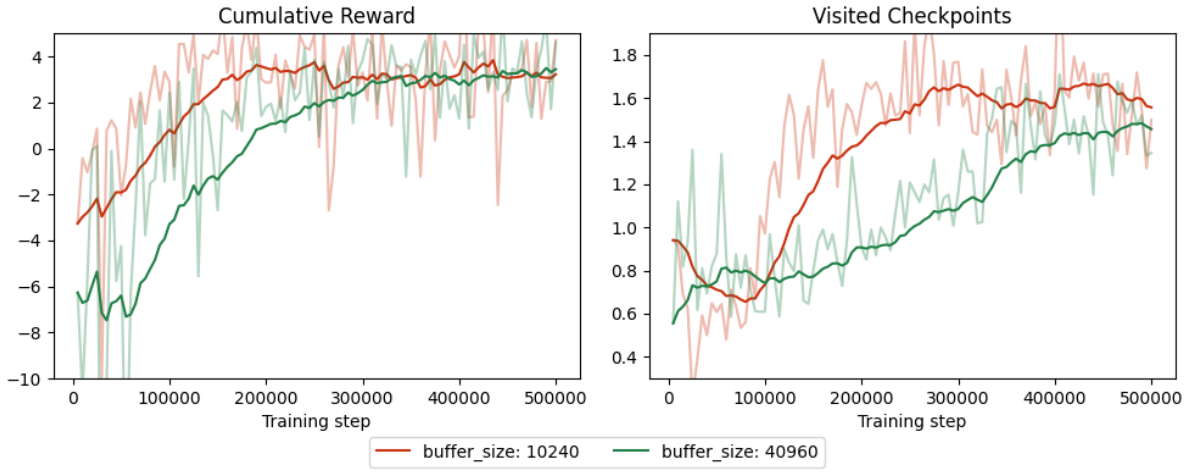
Parametr *lambda* definiuje w jakim stopniu agent polega na przewidzianej wartości podczas przewidywania kolejnej wartości. Niższe wartości *lambda* oznaczają poleganie w większym stopniu na przewidzianej wartości, natomiast większa *lambda* odpowiada poleganiu bardziej na rzeczywiście otrzymanych nagrodach. Po porównaniu granic typowego zakresu wartości ($0.85 - 0.95$), na wykresach poniżej widać, że wartość niższa przekłada się na minimalnie lepszy trening.



Rysunek 2.12: Wybór hiperparametrów: lambd

2.7.3 buffer size

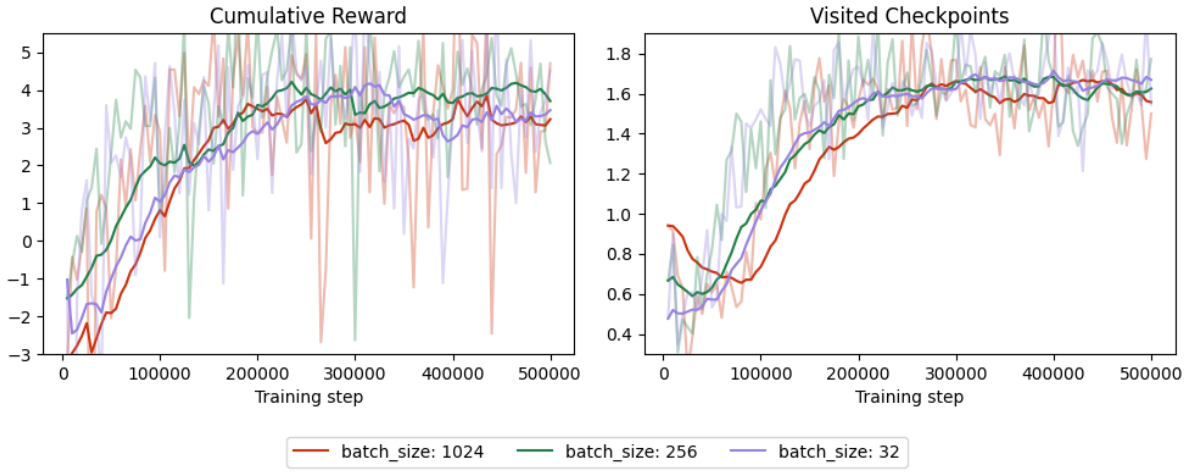
Rozmiar bufora odpowiada ilości cykli treningowych (zebranie obserwacji, podjęcie akcji, otrzymanie nagrody) które są zbierane przed aktualizacją modelu. Większa wartość przekłada się na bardziej stabilny trening, kosztem czasu. Po porównaniu wartości 10240 i 40960 można zauważyć, że niższa wartość zapewnia zadowalającą stabilność, otrzymując podobne wartości znacznie wcześniej.



Rysunek 2.13: Wybór hiperparametrów: buffer size

2.7.4 batch size

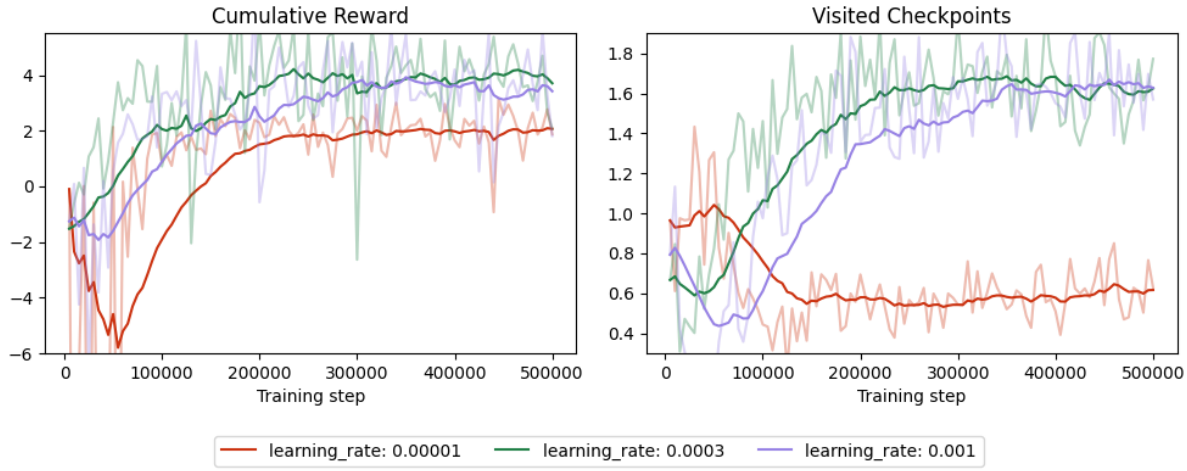
Parametr *batch_size* definiuje ilość cykli treningowych wykorzystywanych przy propagacji wstecznej. Przy dyskretnej przestrzeni akcji wartość ta powinna być mniejsza, niż dla akcji ciągłych. Zmniejszenie *batch_size* do 256 minimalnie poprawiło trening, natomiast spadek do 32 nie wprowadził zauważalnych różnic.



Rysunek 2.14: Wybór hiperparametrów: batch size

2.7.5 learning rate

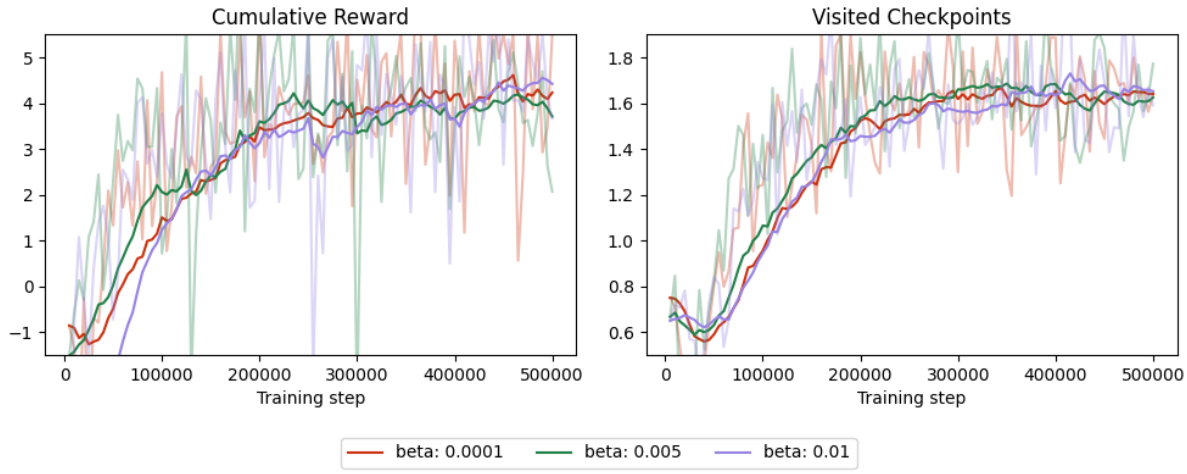
Prędkość uczenia bezpośrednio odnosi się do siły każdego kroku propagacji wstecznej. Analizując poniższe wykresy, wartość pośrodku typowego zakresu $(1e-3, 1e-5)$ pozwala na najbardziej optymalną prędkość uczenia w skali 500000 iteracji.



Rysunek 2.15: Wybór hiperparametrów: learning rate

2.7.6 beta

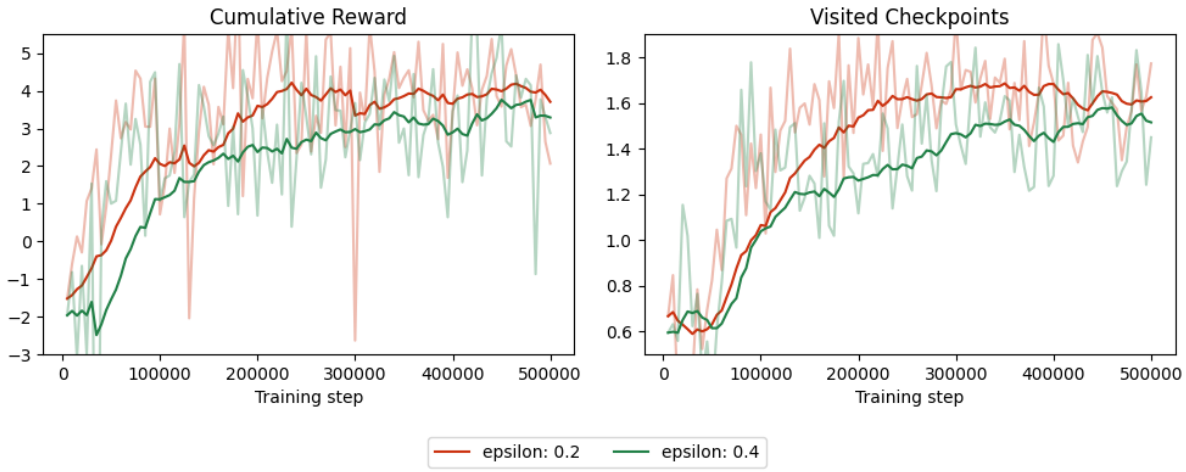
Parametr *beta* definiuje skalę randomizacji polityki. Większe wartości parametru powodują podejmowanie przez bota większej ilości losowych akcji, zwiększając ilość sprawdzanych rozwiązań. Dla zadanego problemu niższa *beta* powoduje szybsze unormowanie się wartości nagrody oraz postępu trasy, dlatego została wybrana wartość 0.005.



Rysunek 2.16: Wybór hiperparametrów: beta

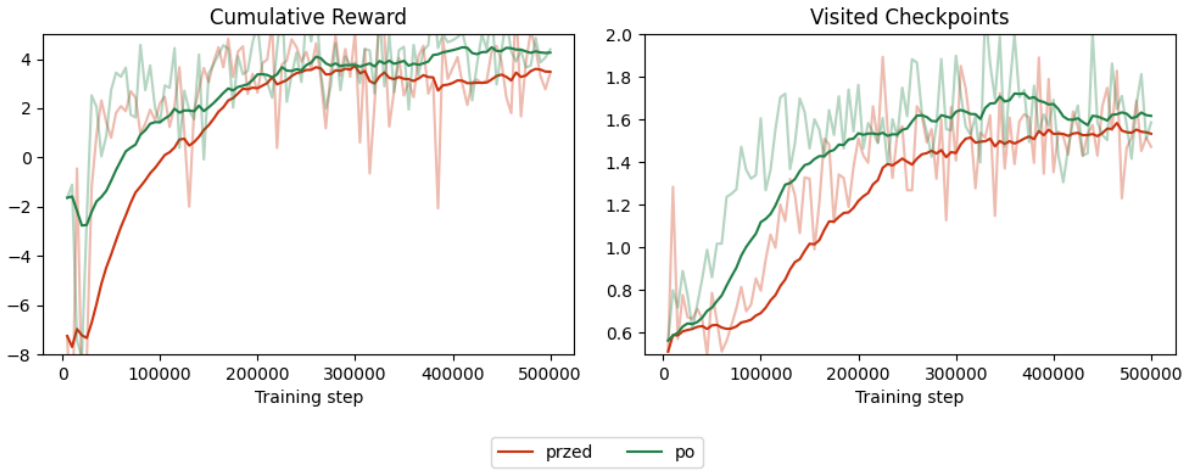
2.7.7 epsilon

Zmienna *epsilon* informuje w jakim stopniu akceptowane są rozbieżności pomiędzy starą i nową polityką podczas propagacji wstecznej. Niższe wartości powodują bardziej stabilny postęp kosztem czasu treningu. Zwiększenie wartości do 0.4 nie spowodowało jednak przyspieszenia treningu.



Rysunek 2.17: Wybór hiperparametrów: epsilon

Finalnie zostały wybrane poniższe hiperparametry, co pozwoliło na zauważalną poprawę treningu. Średnia wartość nagrody dla całego okresu wzrosła z 2.46 do 3.26 (+32.5%), natomiast stopień postępu trasy wzrósł z 1.305 do 1.499 (+14.9%).



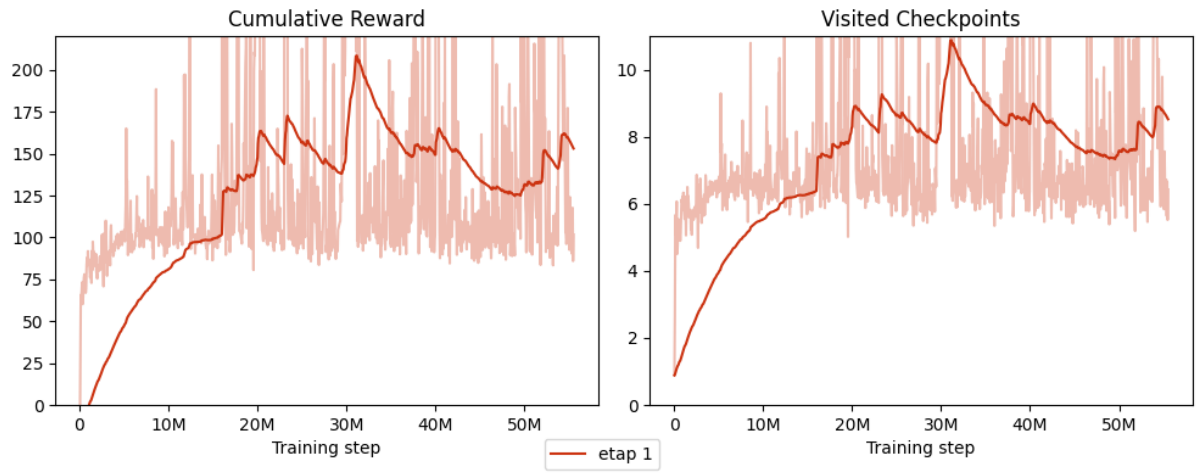
Rysunek 2.18: Wybór hiperparametrów: porównanie wyników

Algorithm 2.2: Hiperparametry finalne treningu

```
hyperparameters :
    batch_size : 256
    buffer_size : 10240
    learning_rate : 0.0003
    beta : 0.005
    epsilon : 0.2
    lambd : 0.85
    num_epoch : 8
    learning_rate_schedule : linear
reward_signals :
    extrinsic :
        gamma : 0.8
        strength : 1.0
```

2.8 Dalsze etapy treningu

Przy trenowaniu ważne jest, aby początkowe zadania były proste, aby agent był w stanie je łatwo wykonać. Następnie należy stopniowo zwiększać poziom trudności, dając agentowi odpowiednio dużo czasu na przystosowanie się do zmian. Dlatego pierwszy etap treningu polega na przejechaniu trasy będącej okręgiem, po płaskim terenie, bez wykorzystania tekstur (droga jest czarna, otoczenie białe) oraz bez przeciwników (w każdym środowisku znajduje się jeden agent).



Rysunek 2.19: Postep treningu: etap 1

Etap 2 zakłada dodanie większej różnorodności torów, zachowując stosunkowo prosty kształt, tylko nieznacznie odbiegający od okręgu, z małą ilością ostrych zakrętów.
Etap 3 sprawdza adaptowalność pojazdu na skomplikowanym torze, z dużą ilością ostrych zakrętów.
Etap 4 zakłada dodanie nierówności terenu, tworząc góry i pagórki na trasie.

Podsumowanie

W powyższej pracy została stworzona gra wyścigowa, wraz z botem. Bot został wytrenowany z wykorzystaniem uczenia przez wzmacnianie przy pomocy Unity ML-Agents. Jako interakcję ze środowiskiem bot posiada do dystpozycji dwie akcje w przestrzeni dyskretnej o wartościach -1, 0, 1 definiujące poruszanie się do przodu/tyłu oraz kąt skrętu kierownicy. Pojazd porusza się zgodnie z uproszczonymi zasadami fizyki symulowanymi przez silnik Unity. Jako obserwacje przyjmuje odległości pojazdu od ewentualnych przeszkód oraz krawędzi drogi i aktualną prędkość. Taki zbiór danych wejściowych ma na celu symulować podejście wykorzystywane aktualnie w samochodach autonomicznych.

Proces implementacji bota, wraz z wyborem obserwacji, podejmowanych akcji, funkcji nagrody i wyboru hiperparametrów przedstawiony został na wykresach ilustrujących efekty podjętej decyzji. W pracy zostały również opisane kolejne etapy treningu bota.

Bibliografia

[1] Lidar. URL: https://pl.wikipedia.org/wiki/Lidar.

[2] Perlin noise. URL: https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html.

[3] Poziomy autonomii pojazdów sae. URL: https://www.sae.org/blog/sae-j3016-update.

[4] Unity ml-agents repository. URL: https://github.com/Unity-Technologies/ml-agents/.

[5] Xiao-Diao Chen et al. Improved algebraic algorithm on point projection for bézier curves. *Proceedings of the Second International Multi-Symposiums on Computer and Computational Sciences*, pages 158–163, 2007. URL: https://hal.inria.fr/file/index/docid/518379/filename/Xiao-DiaoChen2007c.pdf, doi:10.1109/IMSCCS.2007.17.

[6] Arthur Juliani et al. Unity: A general platform for intelligent agents. 2020. URL: https://arxiv.org/pdf/1809.02627v2.pdf.

[7] Thomas Nakken Larsen et al. Comparing deep reinforcement learning algorithms' ability to safely navigate challenging waters. *FrontRobotAI*, 2021. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8473616/, doi:10.3389/frobt.2021.738113.

Załącznik A

Zawartość płyty CD

Na płycie CD znajduje się kod źródłowy programu. Wybudowana gra nie zmieściła się w zakresie 100MB załącznika w systemie ASAP.
Aby otworzyć kod źródłowy, należy:

- zainstalować silnik Unity zgodnie z instrukcjami z oficjalnej strony
  ($https://unity.com/download$)
- zainstalować pakiet Ml-Agents
  ($https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Installation.md$)
- uruchomic Unity
- podmienić folder *Assets* na folder dołączony w dodatku
- zaimportować poniższe pakiery z *UnityAssetStore*

  Best Sports CARS - Pro 3D Models

  Coconut Palm Tree Pack

  Conifers [BOTD]

  Standard Assets

  TextMesh Pro
- wybrać projekt *game-code*
- załadować scenę $Scenes/MainMenu$, oraz uruchomić

Dodatkowo pełny kod źródłowy dostępny jest pod linkiem:
https://github.com/KamilMatejuk/Praca-Inzynierska-Gra