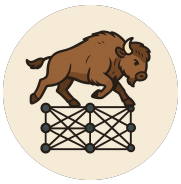
DumbCar

Uczenie autka jeździć, ewentualnie żubra.

Wydział Matematyki i Informatyki UJ
Informatyka analityczna - TCS
MPUM - dr Katarzyna Grygiel
Semestr letni 2025

Hubert Bernacki, Oskar Krygier
16.06.2025 - v0.1.0 - final



1 Wstęp

> Uwaga! Informujemy, że tworzenia projektu nie ucierpiał ani żaden żubr, ani fizyczne autko, ani nawet Niki Lauda. Jedyne co zostało uszczerbione w czasie tworzenia projektu to nasza psychika.

1.1 Projekt

Celem projektu jest oczywiście nauka samochodziku (*żubra*) poruszania się po torze. Samochodziki powinny się poruszać torem jak najszybciej i głównie właśnie takie przypadki rozważamy.

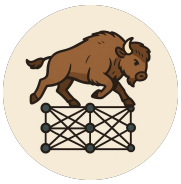
Trasy wykorzystane w pracy zostały zaprojektowane przez wybitnych fachowców, światowej klasy inżynierów, a następnie odtworzone i wprowadzone do projektu przez dwóch nieco mniej kompetentnych studentów. Wykorzystaliśmy tory zarówno proste bez przecięć jak i bardziej skomplikowane, tak jak na przykład tor wzorowany na niemieckim Nürburgringu.

1.2 Narzedzia

Projekt napisaliśmy w języku **`Odin`**, korzystając z biblioteki **`Raylib`** - do rysowania, pobierania inputu oraz matematyki związanej z kolizjami. Wykresy narysowane zostały w **`pythonie`**.

1.3 Github

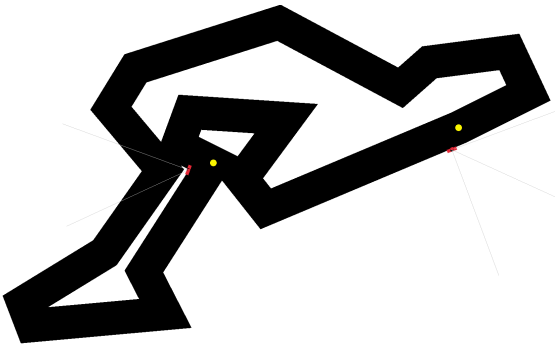
Link do repozytorium znajduje się TUTAJ



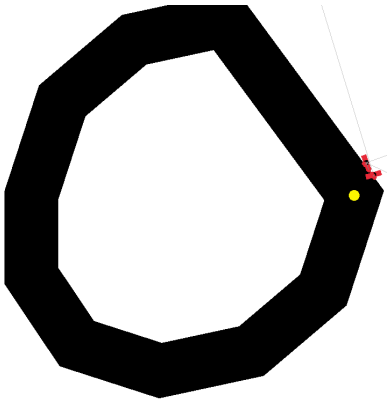
2 Mapy

Napisaliśmy własny generator map w którym stworzliśmy te mapy:

Nürburgring

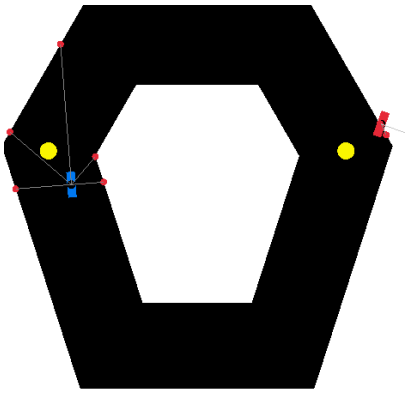


Deca

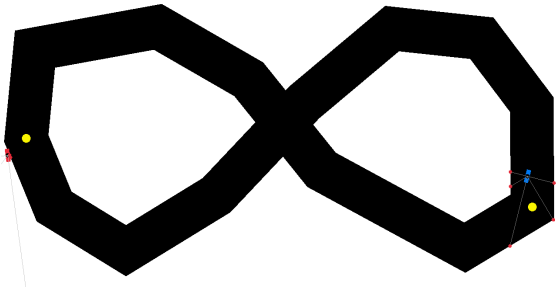


Bootcamp

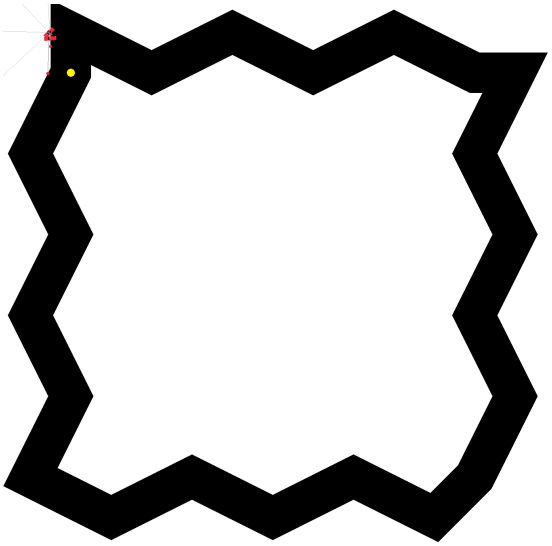
Hex



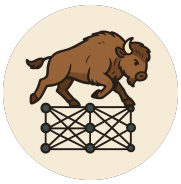
Infinity



Zigzag



Map1

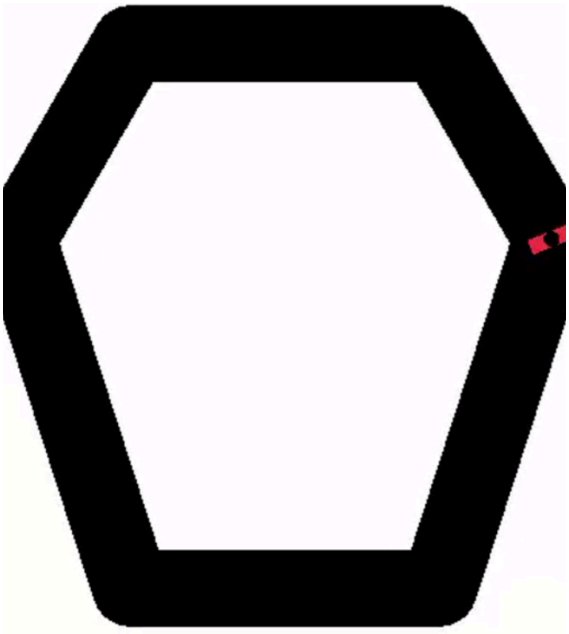


3 Wnętrzności

3.1 Tor

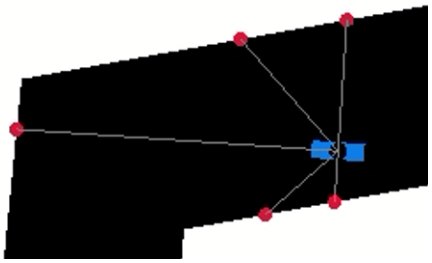
Tor, trasa bądź mapa jest u nas zdefiniowany przez tablicę **waypoint**-ów. W początkowej wersji autko mogło poruszać się tylko jeśli było w odległości nie większej niż **TRACK_WIDTH / 2** od jakiejś pary kolejnych **waypoint**-ów. W przeciwnym przypadku uznawaliśmy, że jest poza trasą - autko ginęło. Traktujemy tą tablicę cyklicznie - odcinek ostatni z pierwszym też jest na trasie. Przykładowa mapa z listą punktów:

```
HEX_MAP_SIZE :: 6
HEX_MAP :: Map(HEX_MAP_SIZE){
    [HEX_MAP_SIZE]rl.Vector2{
        rl.Vector2{473.2, 200.0},
        rl.Vector2{400.0, 73.2},
        rl.Vector2{200.0, 73.2},
        rl.Vector2{126.8, 200.0},
        rl.Vector2{200.0, 426.8},
        rl.Vector2{400.0, 426.8},
    },
}
```



3.1.1 Końcowy tor

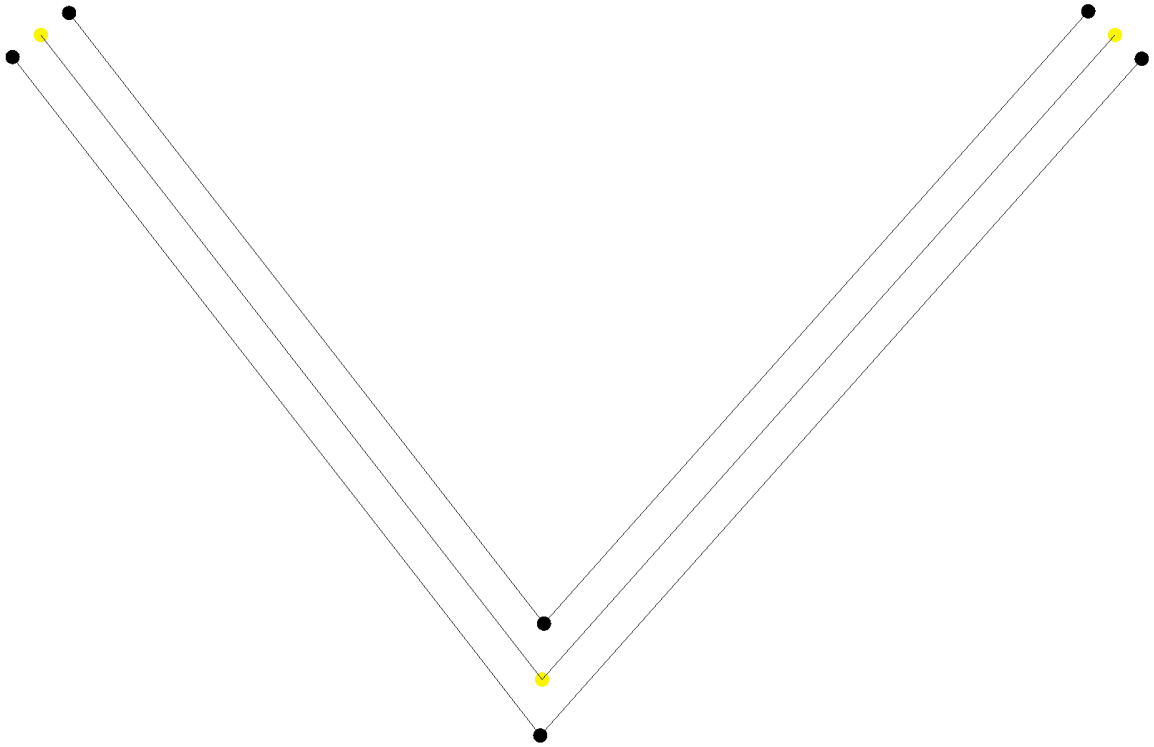
Zdefiniowanie toru jako wszystkich punktów w pewnej odległości do odcinków jest całkiem wygodne pod względem sprawdzania czy auto jest na trasie, a także fajnie wychodzi to wizualnie - są ładne zaokrąglone zakręty. Jednak problematyczne okazują się promienie. Sieć neuronowa którą uczymy jeździć po torze dostaje na wejściu odległości do najbliższych punktów na granicach toru na promieniach od środka autka, najlepiej pokazać to na zdjęciu:



Widzimy 5 promieni, każdy zaczyna się na środku auta i kończy w miejscu zderzenia z krawędzią toru, bądź kończy na maksymalnej długości tego promienia.

W modelu z zaokrąglonymi rogami matematyka stojąca za obliczaniem punktów zderzeń ray'ów jest dużo mniej przyjemna - musimy znajdować odpowiednie wycinki okręgów a następnie

przecinać je z odcinkami - nie było tego w Raylibie (bibliotece którą wykorzystujemy głównie do rysowania, ale też do jakiejś prostej geometrii), a pisanie tego samemu wydawało się nieciekawe, zatem zmieniliśmy trochę definicję naszego toru, żeby była prostsza do pracy z nią. Znowu najprostsze może okazać się pokazanie rysunku a później opisanie tego co się na nim dzieje:



Na rysunku przedstawione są 3 kolejne **`waypoint`**-y (żółty kolor), tym razem chcielibyśmy, żeby tor był wyznaczony przez przedłużenia odcinków równoległych, do odcinków wyznacznych przez 2 kolejne **`waypoint`**-y, oddalonych od siebie o **`TRACK_WIDTH / 2`**. Zatem oprócz listy **`waypoint`**-ów trzymamy jeszcze dwie listy - punktów wewnętrznych i zewnętrznych (nazewnictwo się trochę psuje w przypadku torów z samoprzecięciemi - bo wtedy czasmi wewnętrzne są na zewnątrz i vice versa). Punkty te są łatwe do wyznaczenia - wyznaczamy 2 kolejne pary odcinków przesuniętych o **`+/- TRACK_WIDTH/2`**, przecinamy odpowiadające im proste - dostajemy 2 punkty - narożniki trasy. Teraz raycasty są dużo prostsze - potrzebujemy tylko przecinać odcinki z odcinkami.

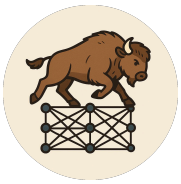
Bardzo ważnym elementem naszej symulacji jest pole **`p_now`** struktury autka - jest to indeks ostatniego **`waypoint`**-a, obok którego autko przejechało.

3.1.2 Jak wyznaczamy `p_now`?

Gdy autko zbliży się na odległość **`TRACK_WIDTH`** do następnego waypointu, to zwiększamy **`p_now`** o **`1`**. Tak **`TRACK_WIDTH`** bez **`/ 2`** - nie chcemy żeby autko mogło ściąć zakręt jadąc na tyle szybko żeby przeskoczyć okrąg w jednej klatce.

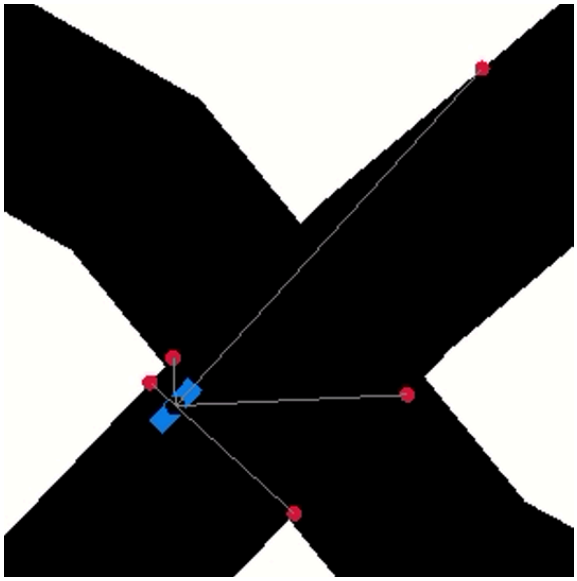
3.1.3 Gdzie wykorzystujemy `p_now`?

1. Sprawdzanie czy autko jest na trasie.
   Skoro znamy mniej więcej najbliższy **`waypoint`** autka, to nie musimy sprawdzać całej trasy - wystarczy, że sprawdzimy czy jest w tych wielokątach wyznaczanych przez najbliższe kilka



punktów na wnętrzu i zewnętrzu trasy odpowiadającym **waypoint**-om **p_now - 1**, **p_now**, **p_now + 1** .

2. Obliczanie **score** - funkcji oceny - jednym ze składników tej funkcji jest **p_now**, a także jak daleko jest autko między **waypoint**-ami **p_now** i **p_now + 1** - zwykły iloczyn skalarny dwóch wektorów, ograniczony (**clamp**) do przedziału $[-1, 1]$.
3. Raycasty - tutaj rola **p_now** jest bardzo ważna - przy sprawdzaniu przecięć promieni autka z trasą, najpierw wyznaczamy przedział $[j_0, ..., \textbf{p\_now}, ..., j_1]$ - sprawdzamy kolejne **waypoint**-y w przód i w tył aż znajdziemy **waypoint** oddalony od autka na co najmniej **RAY_LENGTH** - na nim kończymy. Będziemy robili raycasty tylko na odcinki wewnętrzne i zewnętrzne odpowiadające **waypoint**-om z tego przedziału. Po pierwsze ogranicza to potrzebne porównania - nie musimy przecinać każdego Ray'a z każdym brzegiem trasy. Po drugie - najważniejsze - pozwala na samoprzecięcia na trasie:

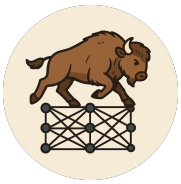


3.1.4 Jakie to powoduje problemy?

1. Gdy **waypoint**'y są bardzo blisko siebie możemy od razu 'zdobyć' kilka następnych **waypoint**-ów, ze względu na to, że zwiększamy **p_now** gdy jesteśmy w odległości **TRACK_WIDTH** od następnego **waypoint**'a - tutaj sensowniejszym byłoby sprawdzanie czy przeszliśmy na drugą stronę prostej wyznaczonej przez punkty odpowiadające następnemu **waypointow**-i z wnętrza i zewnętrza toru - jednak zajęliśmy się implementacją innych rzeczy.
2. Auta zawracają po zdobyciu **waypoint**'a na ostrych zakrętach ($> 90°$) - to chwilowo lokalnie zwiększa score, ale maksymalnie o 1, bo autko nie dojedzie do następnego **waypoint**-a.
3. Przy za długich ray'ach nie działają samoprzecięcia trasy - ustawiamy zatem na odpowiednich mapach ray'e na odpowiednio krótszą długość.
4. Przy wielu za bliskich **waypoint**-ach możemy w głupi sposób zostać potraktowani jakbyśmy wypadli z trasy - jednak tak tworzyliśmy mapy żeby uniknąć tego problemu - zmiana logiki zwiększania **p_now** (jak wyżej) by to naprawiła i uprościła.

3.2 Sterowanie

Stworzyliśmy model sterowania autka taki jaki mógłby być w typowej grze wyścigowej - gracz ma kontrolę nad szybkością samochodu oraz nad prędkością obrotową - w przypadku wersji



sterowania poprzez klawisze **WASD** klawisze **W** oraz **S** ustawiają w danej klatce symulacji fizycznej wartość przyspieszenia samochodziku na **ACC**, odpowiednio **-ACC**, a klawisze **A** oraz **D** zmieniają prędkość kątową na **ROTATION_SPEED**, odpowiednio **-ROTATION_SPEED**.

Zatem, żeby interfejs sterowania dla algorytmów był podobny, to algorytm sterujący może ustawiać w każdej klatce symulacji 2 wartości - przyszpieszenie na wartość z przedziału $[-1, 1]$, oraz prędkość kątową, również na wartość z $[-1, 1]$. Następnie te wyniki algorytmu sterującego są przeskalowywane przez **ACC** oraz **ROTATION_SPEED**.

Szybkość auta jest ograniczona przez 2 stałe - **MIN_SPEED** oraz **MAX_SPEED**, to oznacza że autko zawsze będzie się musiało poruszać, ale też nie może poruszać się za szybko. Parametr **MIN_SPEED** dodaliśmy stosunkowo późno, żeby zobaczyć jak wpłynie on na algorytm uczenia sterowania.

3.2.1 Sieć neuronowa

Ostatnia warstwa sieci w oczywisty sposób musi mieć 2 liczby - odpowiadające przyspieszeniu i prędkości obrotowej.

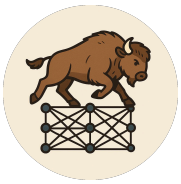
W warstwach ukrytych mamy dowolność wybierania, ale skupialiśmy się głównie na testowaniu 2 bądź 3 warst ukrytych z kilkoma / kilkunastoma wierzchołkami w każdej z tych warstw.

Najciekawszym jest wejście do sieci neuronowej, bo istnieje dużo parametrów, które możemy tam wrzucić. Bazą każdego inputu do sieci są promienie(Ray'e), z autka w kilku (najczęściej 5) różnych kierunkach puszczane są Ray'e o pewnej maksymalnej długości - **MAX_RAY_LEN**. Sprawdzamy, czy zderzą się one z brzegiem toru wyścigowego i do sieci wrzucamy znormalizowane odległości do punktów zderzeń, jeśli nie to po prostu dajemy **1**. Oprócz tych promieni sprawdziliśmy kilka różnych opcji na inne dane wejściowe:

1. Szybkość - sieć uczy się sterować znając tylko informacje z promieni oraz szybkość autka. Jest to najprostszy sposób i ma porządaną własność - sieć nie dostaje na wejściu pozycji w symulacji, to nie może nauczyć się jeździć na pamięć po współrzędnych mapy. Jednak możliwe jest nauczenie się przez sieć stałej prędkości kątowej oraz przyspieszenia - z tym przyszło nam walczyć.

   Pojawia się tu także ciężki do obejścia problem ostrych zakrętów - na prostych trasach możemy wyobrazić sobie, że autko może nauczyć się jeździć na środku toru - utrzymywać jak najbardziej podobne odległości do lewego i prawego brzegu trasy oraz jak największą pustą przestrzeń przed maską. Jednak skoro autko nie ma żadnego pojęcia o kierunku w którym ma jechać, taka strategia ma duże problemy, jak autko dojedzie do ostrego zakrętu, bo będąc blisko ściany autko może chcieć skręcić wcześniej. Na mapach z ostrymi zakrętami czasami autka się dobrze uczą, a czasmi nie, potrafią zawracać bo zdobyciu **waypoint**'a co ze względu na naszą funkcją oceniania może lokalnie zwiększyć wynik - ale autko nie zdobędzie następnego **waypoint**'a.
2. Prędkość + pozycja na mapie + kierunek patrzenia - tutaj mamy dużo więcej bo aż 6 dodatkowych parametrów. Sieci tak nauczone uczą się po części samej mapy, co widać, bo gdy puszczamy te sieci na innej mapie to w niektórych miejscach niepotrzebnie zwalniają, a w innych jadą bardzo szybko.
3. Szybkość + kąt między kierunkiem patrzenia, a kierunkiem do następnego **waypointa** na torze. Znowu chcemy dać jakoś sieci wiedzę o to w jakim kierunku ma jechać, tym razem dodając tylko jeden dodatkowy parametr i nie dając sieci informacji o faktycznej pozycji. Dawało to całkiem dobre wyniki i na niektórych mapach widać było, że może się to nauczyć lepiej niż bazowa sieć.

Jako funkcję aktywacji wykorzystujemy tangens hiperboliczny, zwraca on liczby z przedziału $[-1, 1]$, czyli takie jakich potrzebujemy na wyjściu. Na wczesnym etapie projektu próbowaliśmy



wykorzystywać ReLU bądź sigmoida, nie działały one dobrze, jednak mogło być to spowodowane błędami w normalizacji / skalowaniu albo innymi problemami które wtedy mieliśmy w kodzie.

3.2.2 Heurystyka

W pewnym momencie postanowiliśmy również rzucić wyzwanie modelom zbudowanym na sieci neuronowej i zdecydowaliśmy się umieścić również algorytm heurystyczny sterowania samochodzikiem. W tym momencie wykraczamy nieco poza ML'ową część sztucznej inteligencji, natomiast uznaliśmy, że to jest jeden z niewielu sposobów, które mogłyby sobie dobrze poradzić w tym przypadku.

3.2.2.1 Idea

Nasza heurystyka bazuje na kilku parametrach. Są to

1. Wektor prowadzący od miejsca, w którym aktualnie się znajdujemy, do docelowego, którym jest kolejny punkt (przegięcia toru).
2. Wektor przemieszczenia w ostatnim wykonanym ruchu.
3. Odległość do krawędzi jezdni w kierunku takim, jak jest ustawiony aktualnie samochodzik.

Idea tego algorytmu jest *bardzo prosta*. Znając współrzędne kolejnego punktu na trasie, algorytm ma za zadanie kierować się dokładnie w jego stronę. Jeżeli kolejny punkt znajduje się pod dużym kątem względem ostatniego kroku, samochodzik musi wykonać **jak największy skręt i odpowiednio zredukować prędkość**. W przypadku, gdy samochodzik znajduje się w miejscu, gdzie nie ma przed sobą żadnego obiektu i jest ustawiony w kierunku kolejnego punktu, powinien **przyspieszyć**. Jeśli zobaczy przed sobą jakiś obiekt, **zwalnia**.

W pierwotnej wersji algorytmu używane były jedynie pierwsze dwa punkty. Algorytm działał bardzo słabo na praktycznie każdym torze, gdyż samochodzik za szybko wchodził w zakręty, stąd powstał pomysł, żeby dodać również trzeci punkt.

3.2.2.2 Rozwiązania techniczne

Niech $\vec{v}$ – wektor ostatniego przemieszczenia, oraz $\vec{w}$ – wektor do kolejnego punktu docelowego. Połóżmy również θ – kąt między v, a w. Rownież niech d – odległość do przeszkody.

Procedura **`next_step`** przyjmuje elementy wymionione w poprzedniej sekcji, a zwraca jedynie wartość przyspieszenia **`acc`** oraz wartość skrętu **`rot`**. Obie wartości są z przedziału $[-1, 1]$.

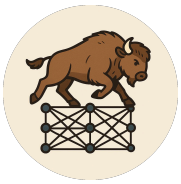
Kluczowa obserwacja, jest taka, że jeżeli $\cos\theta > 0$, to znaczy, że punkt jest gdzieś z boku nas albo przed nami, zatem

$$\text{rot} := \begin{cases} \sin\theta & \text{gdy } \cos\theta > 0 \\ \text{sign}(\sin\theta) & \text{wpp} \end{cases} \tag{1}$$

Dla klarowności zdefiniujemy definiujemy też $\text{clamp}(x, a, b) := \max(a, \min(x, b))$

Przypadku **`acc`**, $\cos\theta$ jest bazą. W tym przypadku bierzemy również pod uwagę wymienioną wcześniej *odległość do przeszkody*. Sposób liczenia w tym miejscu jest często zmienny, ponieważ ciężko jest znaleźć złoty środek. Przykładowa nienajgorzej działająca formuła prezentuje się następująco:

$$\text{acc} := \text{clamp}\left(\cos\theta - \frac{\exp(d) \cdot |\vec{v}|}{3} + 1, -1, 1\right) \tag{2}$$



Algorytm na niektórych mapach radzi sobie bardzo sprawnie, a na innych trochę mniej. Stanowi nie najgorszego konkurenta dla samochodzików z siecią neuronową.

3.3 Uczenie

3.3.1 Funkcja oceny

Naturalnie chcemy w jakiś sposób klasyfikować przejazdy na lepsze lub gorsze. W tym celu definiujemy sobie funkcję oceny. Tak naprawdę jest to liczba przejechanych **`waypoint`**'ów na trasie wliczając również znormalizowaną do 1 odległość jaką przejechaliśmy między poprzednim punktem, a kolejnym. W oczywisty sposób ta funkcja jest rosnąca.

Przewidujemy również mandaty karne. W naszym świecie nie obowiązują kary za szybką lub wolną jazdę (mamy za to limit górny i dolny na prędkość, których **nie da się przekroczyć**). Jest za to kara za spowodowanie kolizji z barierką (samochodziki nie kolidują ze sobą).

Ogromna kara **3 punktów karnych** powędruje do kierowcy, który, umyślnie bądź też nie, wyląduje poza trasą. Wtedy kolor jego samochodu robi się **agresywnie czerwony**. Zabraniamy również takim kierowcom dalszej jazdy. De facto **–3** oznaczają to samo, co gdyby kierowca zatrzymał się po wyznaczonym czasie w miejscu odpowiadającym 3 **`waypoint`**'y wstecz i bez kolizji.

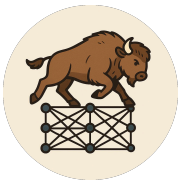
3.3.2 Algorytmy genetyczne

Mamy już naszą funkcję kosztu, ale wciąż nie wiadomo jak w ogóle trenować sieć neuronową. Aby poradzić sobie z tym problemem, zaimplementowaliśmy **algorytm genetyczny**. Załóżmy, że chcemy jednocześnie utrzymywać N sztucznych sieci neuronowych. Zasada działania algorytmu jest następująca:

1. W pierwszej iteracji, dla każdej z N sieci neuronowych, losujemy *wagi krawędzi* oraz *bias* dla każdego węzła w sieci neuronowej. Chcemy wykonać sumarycznie K kroków. Przechodzimy do punktu (3).
2. Bierzemy k najlepszych zawodników z poprzedniej iteracji. Oprócz tego, kilkukrotnie losowo wybieramy któregoś spośród nich, kopiujemy go i losowo go *mutujemy* (czyli zmieniamy nieznacznie i losowo *wagi krawędzi* oraz *bias*).
   Bierzemy również kilkukrotnie dwóch losowych zawodników ze zbioru najlepszych i uśredniamy ich parametry.
   Chcemy żeby było sumarycznie N sieci neuronowych, więc dopełniamy losowymi zawodnikami aby osiągnąć tę wartość.
3. Dla każdej sieci neuronowej włączamy symulację na określonej mapie, która zwraca nam wynik z *funkcji oceny*, zdefiniowanej powyżej. Sortujemy wyniki rosnąco. Inkrementujemy numer kroku. Jeśli numer kroku jest mniejszy niż K, to przechodzimy do punktu (2), w przeciwnym przypadku idziemy do (4).
4. Zwracamy zawodnika z najwyższym wynikiem i to jest już nasza wytrenowana sieć neuronowa.

3.4 Załączniki

W tym projekcie bardzo ważny jest graficzny element tego co się dzieje - możemy łatwo ocenić gołym okiem jak dobrze radzi sobie algorytm sterowania poprzez po prostu obejrzenie jak radzi sobie on na trasie. W związku z tym w paczce znajduje się dużo różnych nagrań.



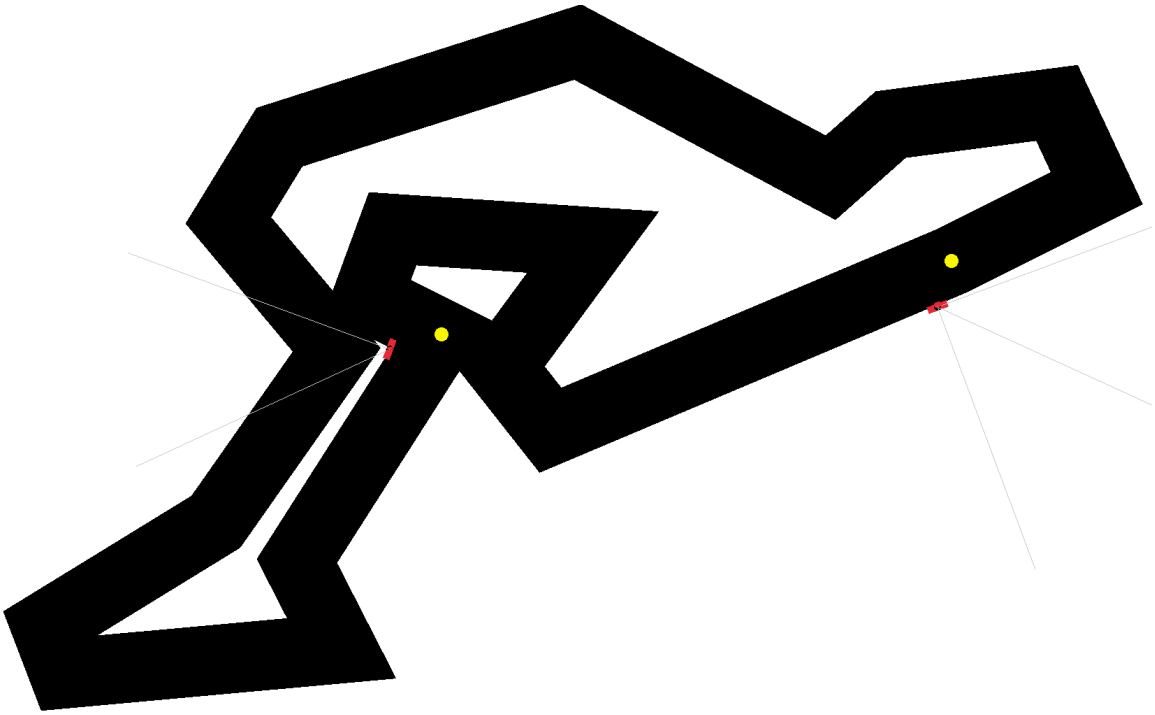
4 Eksperymenty

Przeprowadziliśmy testy uczenia przez 15 sekund z parametrami genetycznymi - 1000 aut, 20 generacji, 300 powstaje jako średnia 2 najlepszych, 400 jako mutacja najlepszego, 50 najlepszych przeżywa i reszta zostaje wylosowana na nowo. Porównywaliśmy jak radzi sobie auto z siecią podstawową - Ray'e + szybkość vs heurystyka oraz z siecią z **`kompasem`** - Ray'e + szybkość + kąt między kierunkiem auta, a kierunkiem do następnego waypointa, a heurystyką. Sprawdziliśmy także 3 rozmiary sieci - z 3, 4 oraz 5 ukrytymi warstwami. Na wykresach jest porównanie tego co się dzieje gdy nauczymy sieć na jednej mapie i przetestujemy na drugiej. Na pierwszym wykresie zaznaczam które AI sieć czy heurystyka poradziła sobie lepiej - w przypadku gdy obie potrafią zrobić okrążenie wygrywa szybsza, a gdy obie umierają wygrywa ta która dalej dojechała. Na drugim i trzecim wykresie rysuje czy sieć wytrenowana na danej mapie pokonuję inną mapę treningową.

Te testy mają raczej na celu porównanie tego jak sensowne wyniki mogą dać te sieci z takimi parametrami. Do testów wykorzystaliśmy 3 mapy:

4.1 Nürburgring

Mapa przerysowana ręcznie z mapy faktycznej trasy formuły 1. Ma kilka bardzo ostrych zakrętów.

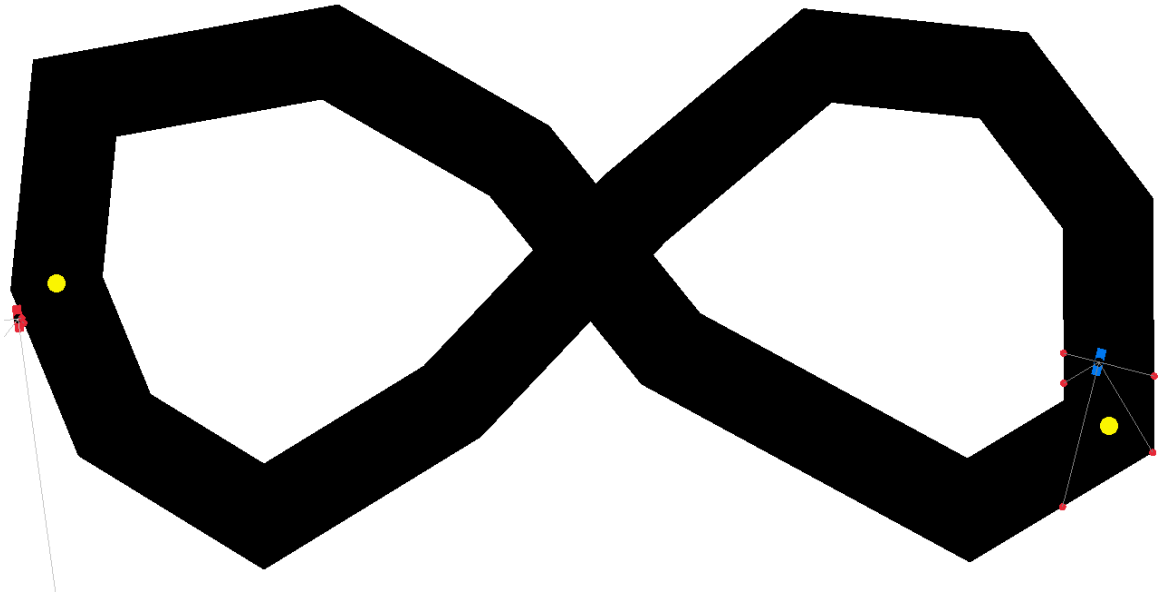


4.2 Bootcamp

Mapa narysowana, żeby miała dużo ostrych i ciasnych zakrętów, ale też długie proste, po których trzeba hamować.

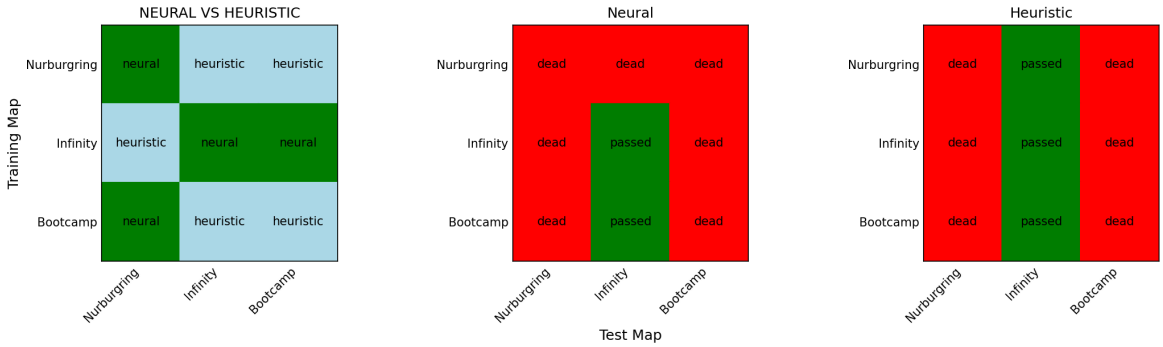
4.3 Infinity

Prosta mapa z samoprzecięciem i zakrętami zgodnie z ruchem wskazówek zegara jak i bez.



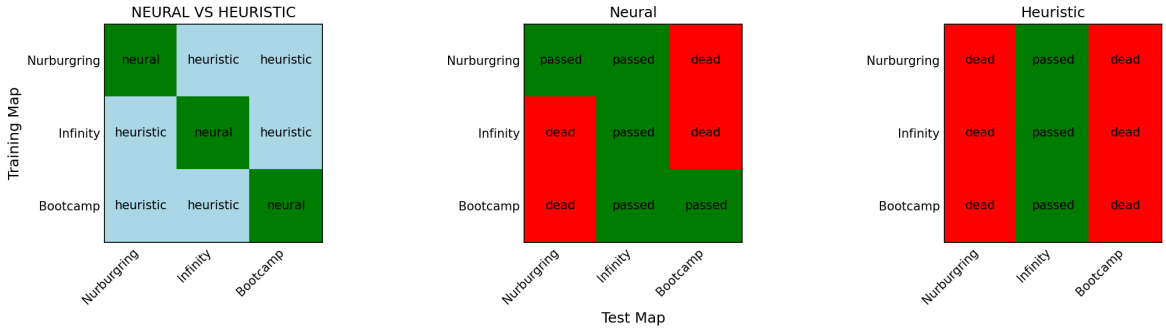
4.4 Kompas - rozmiar 3

wymiary sieci : {7, 20, 2}



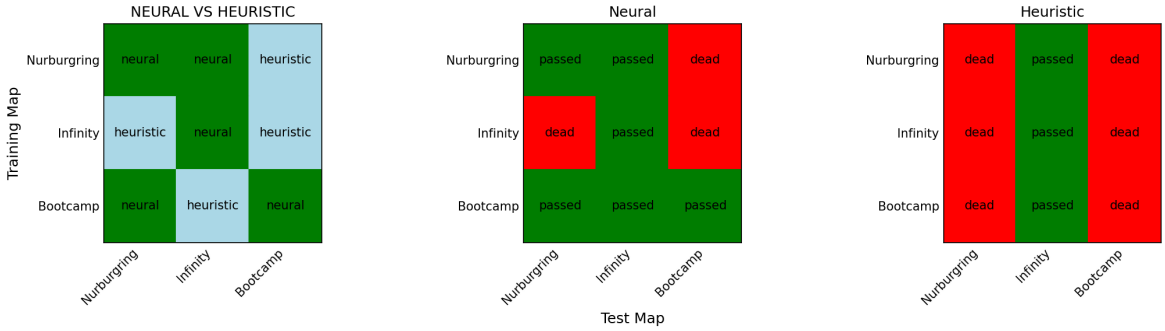
4.5 Kompas - rozmiar 4

wymiary sieci : {7, 8, 6, 2}



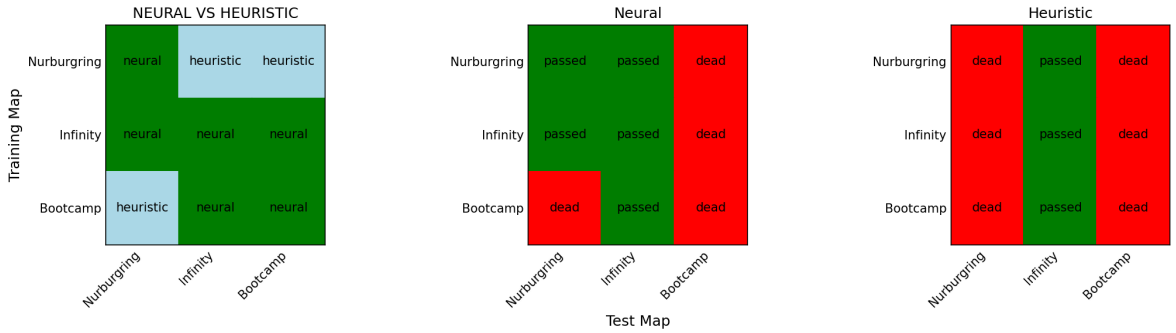
4.6 Kompas - rozmiar 5

wymiary sieci : {7, 7, 6, 5, 2}



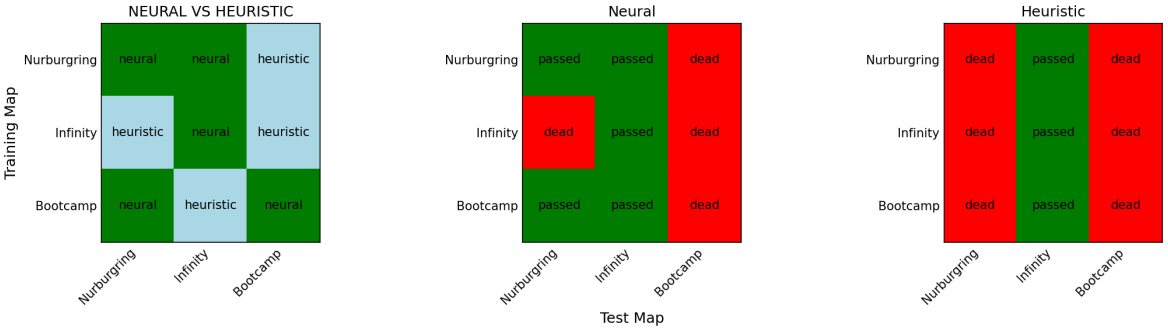
4.7 Bez kompasu - rozmiar 3

wymiary sieci : {6, 20, 2}



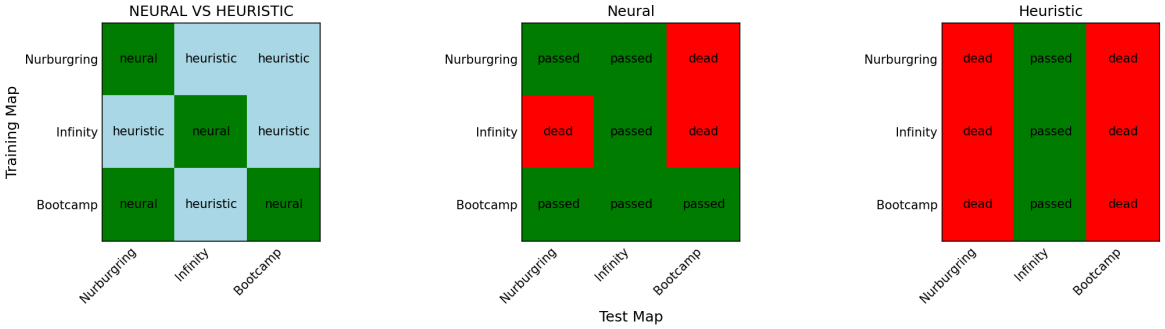
4.8 Bez kompasu - rozmiar 4

wymiary sieci : {6, 8, 6, 2}



4.9 Bez kompasu - rozmiar 5

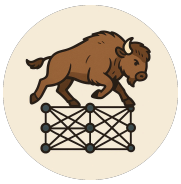
wymiary sieci : {6, 7, 6, 5, 2}



4.10 Opis i wnioski

Widzimy, że niezależnie od wyboru sieci jest zawsze (oprócz kompas 3 na mapie Bootcamp) sieć wygrywa z heurystyką jeśli uczy się na tej samej mapie, na której jest testowana.

Widzimy także, że mapa na której uczymy sieć ma duży wpływ na jej osiągi - najlepszy do nauki wydaje się bootcamp - najwięcej sieci które się na nich uczyło nie umarło na innych mapach, a Infinite loop wydaje się gorszy od Nürburgring'u.



Sieć zdecydowanie lepiej radzi sobie pod względem przeżywalności - heurystyka potrafi przejechać tylko mapę infinity, którą sieć pokonuje prawie zawsze, a do tego sieć radzi sobie także z innymi mapami.

Kompas wydaje się generalnie gorszy od braku kompasu, ale zapewne po prostu jest on niedouczony - dostaje parametr więcej, a czas uczenia i reszta warstw w sieci jest taka sama, jak uczyłem go dłużej to radził sobie całkiem dobrze. Jednak, żeby zebrać tak duże dane porównawcze musiałem ograniczyć czas nauki, bo ogarnięcie jednego przykładu mapa treningowa, testowa + rozmar sieci + kompas czy nie zajmowało kilka minut.

4.11 ! Nagrania !

Do każdej pary zamieszczonej na wykresach jest odpowiedni film jak przebiegał wyścig na mapie testowej.

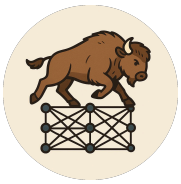
Zielone autko - sieć neuronowa
Niebieskie autko - heurystyka

Nazwa pliku COMPASS/NO - zależnie czy kompas czy nie 3/4/5 - liczba warstw w sieci **`MAPTRAIN_MAPTEST`** - mapa treningowa i mapa testowa (skrócona). Parę ciekawych filmików:

- **`NO_3_BOOT_BOOT.mp4`**
- **`NO_4_BOOT_NUR.mp4`**
- **`COMPASS_3_BOOT_NUR.mp4`**

4.12 Tylko pozycja

Przetestowaliśmy także, co się dzieje gdy nauczymy sieć z inputem będącym parą floatów - znormalizowaną pozycją gracza. Sieć potrafiła się nauczyć pierwszych kilku zakrętów, ale później wpadała w ścianę.



5 Krzywe uczenia

W tej sekcji przedstawiamy krzywe uczenia. Wykresy są tworzone na podstawie *funkcji oceny*, która jest brana z **najlepszego** zawodnika w każdym „kroku" algorytmu genetycznego. Rozważamy uczenie zawodnika na mapie *testowej* oraz jednocześnie testowanie jej (bez uczenia) na torze *testowym.*

Przeważnie używane były sieci neuronowe z trzema ukrytymi warstawami. Głębsze sieci nie wykazywały dużo większej poprawy i wymagały dużo większej ilości czasu do nauki, stąd też nie mamy w raporcie pokazane niz z głębszymi sieciami.

Po licznych obserwacjach wysnuwamy następujące wnioski: Najbardziej użyteczne w treningu są mapy, które na starcie mają dużo zakrętów, ponadto również kawałek prostej. Wtedy samochodziki uczą się przyspieszać tam, gdzie trzeba i jechać wolno przy zakrętach.

5.1 Bootcamp - Infinity

W tym przypadku rozważamy trenowanie na mapie **Bootcamp** oraz testowanie na mapie **Infinity**.

W pierwszym podejściu rozważamy sieć neuronową postaci $[7, 8, 8, 6, 2]$ (przyjmuje 7 parametrów na wejściu). Na wejściu mamy 5 promieni pod kątem $[90°, 45°, 0°, -45°, -90°]$ względem przodu pojazdu. Rezultat jest pokazany poniżej. Co ciekawe, na mapie testowej znacznie szybciej zaczyna dobrze jeździć, mimo że wcale się na niej nie wzoruje.

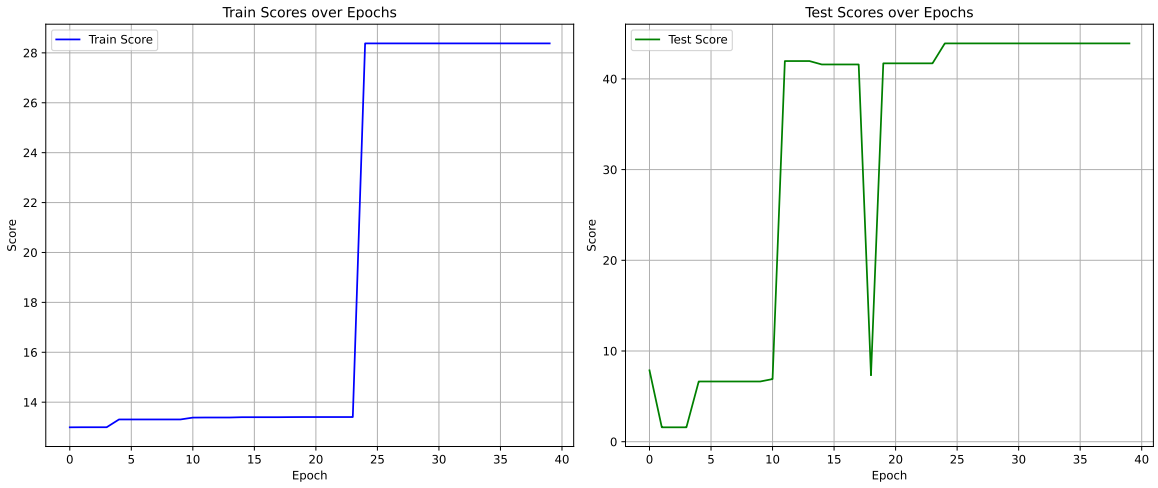


Figure 1 - Krzywe uczenia dla promieni $[90°, 45°, 0°, -45°, -90°]$

W drugim podejściu dodaliśmy więcej promieni. Mamy $[90°, 60°, 30°, 0°, -30°, -60°, -90°]$. Sieć neuronowa naturalnie urosła do postaci [9,8,8,6,2]. Rezultat jest zdecydowanie dużo lepszy. Można zauważyć, że samochodzik również się nauczył dużo szybciej.

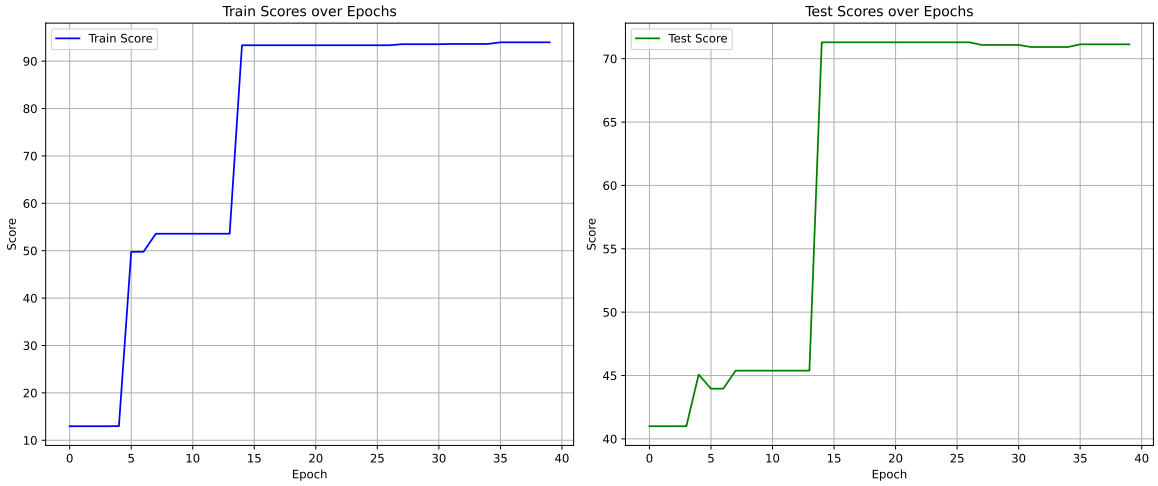


Figure 2 - Krzywe uczenia dla promieni $[90°, 60°, 30°, 0°, -30°, -60°, -90°]$

5.2 ZigZag - Nürburgring

W tym przypadku od razu zostały zastosowane promienie $[90°, 60°, 30°, 0°, -30°, -60°, -90°]$, a sieć neuronowa jest postaci [9, 10, 8, 6, 2]. Rezultat okazał się fanalny dla przypadku testowego co w sumie nie powinno dziwić.

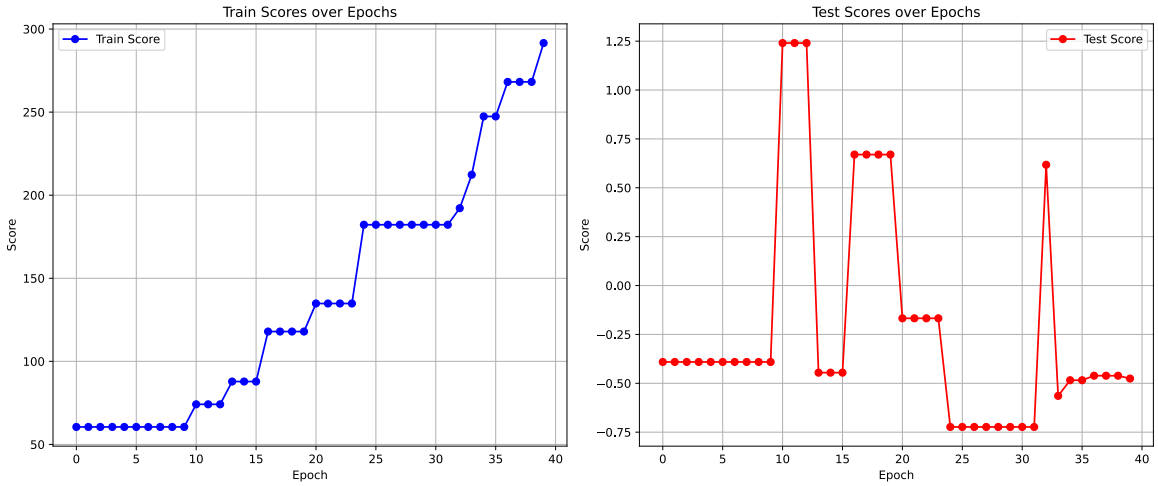


Figure 3 - Trening na ZigZag, test na Nürburgring

5.2.1 Zamiana

Zróbmy małe śmieszne zamieszanie i odwróćmy kolejnością. Co ciekawe, bardzo ładnie się wytrenował na Nürburgring, a jednocześnie świetnie sobie poradził jeżdżąc na ZigZagu. Tego niestety nie możemy tutaj pokazać, natomiast samochód jechał o wiele szybciej niż heurystyka. Często niestety jest odwrotnie.

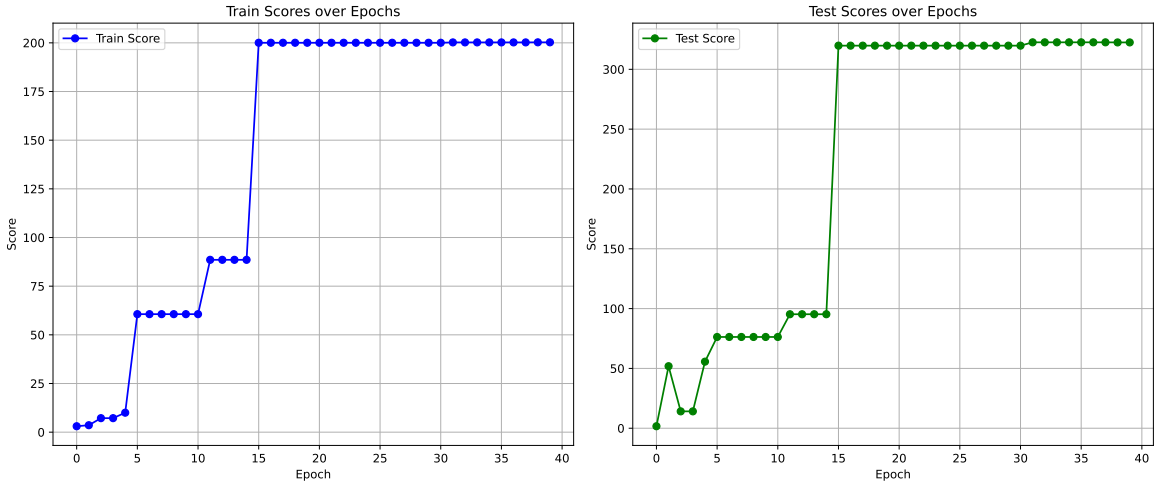


Figure 4 - Uczenie na Nürburgring, testy na ZigZagu

5.2.2 Mniejsza sieć neuronowa

W poprzednim przypadku sieć neuronowa poradziła sobie świetnie. Sprawdźmy co w przypadku mniejszej sieci neuronowej postaci [9, 8, 6, 2]. Wynik niestety jest dużo gorszy, ale samochodzik wciąż jedzie szybciej niż heurystyka.

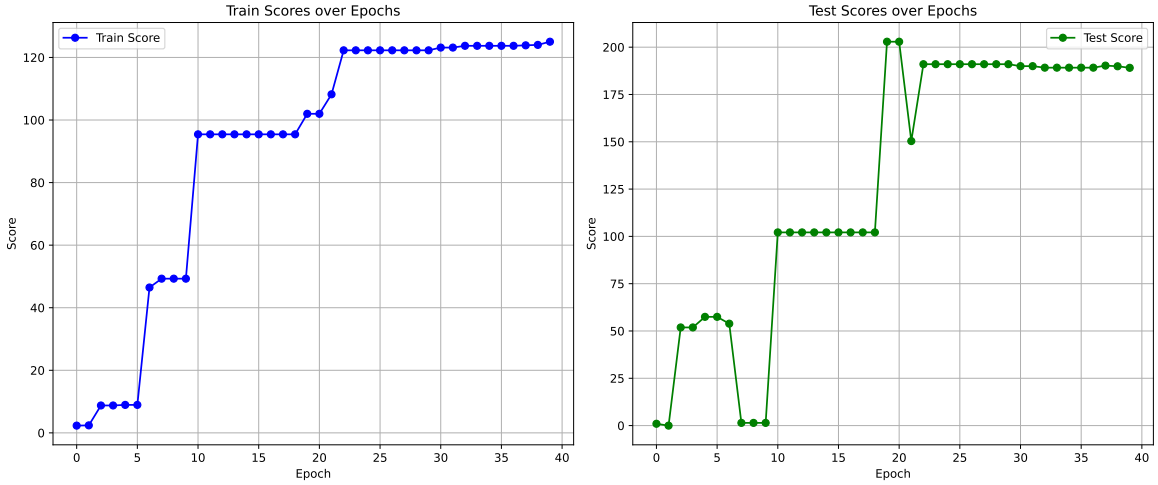


Figure 5 - Uczenie na Nürburgring, test na ZigZag – mniejsza sieć neuronowa

5.3 Nürburgring – Infinity

W tej sekcji przedstawione są rezultaty uczenia na Nürburgring, a testy na Infinity. Jak widać, wylosowało się nieco gorzej niż w poprzednim przypadku. Niemniej jednak można zauważyć, że algorytm się nauczył na obu mapach do pewnego momentu, a potem nic więcej nie znalazł.

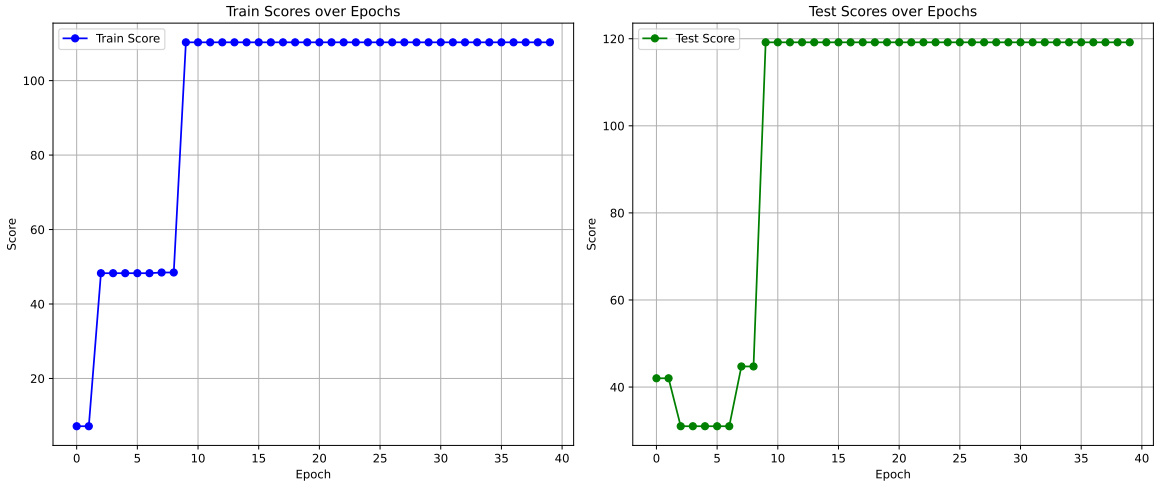
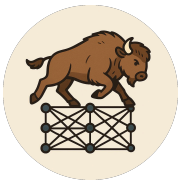


Figure 6 - Uczenie na Nürburgring, testy na Infinity



6 Losowe ciekawostki

6.1 Jazda tyłem

Gdy autko zaczyna obrócone o 180° względem kierunku do następnego **`waypoint`**'a i pozwalamy mu jeździć z ujemną szybkością, a sieć jako input bierze tylko ray'e i szybkość, to faktycznie sieć potrafi się nauczyć jeździć tyłem - oczywiście ze słabymi osiągami na trudnych mapach, ale nadal wygląda to ciekawie np. **`JAZDA_TYLEM_HEX.mp4`**.

Co ciekawe w związku z tym jak działają implementacje sprawdzania czy autko jest na mapie i funkcja score na mapie sześciokątnej autko które postawi się tyłem do kierunku jazdy - przeciwnie do **`waypoint`**-ów uczy się nie jeżdżenia tyłem, ale jeżdżenia przodem pod prąd! - **`PO_CO_TYLEM_JAK_MOZNA_DO_PRZODU.mp4`**

6.2 Zwiększenie minimalnej szybkości

Okazuje się, że zwiększenie minimalnej szybkości, z którą jedzie autko pomaga w uczeniu go, obstawiam, że to dlatego że karamy autka które się rozbiją, a premiujemy te które dojadą najdalej. Zatem gdy autko musi jechać z jakąś minimalną prędkością to albo wpadnie w ścianę, albo będzie musiało przejechać dalej. Oczywiście nie możemy jej ustawić na za wysoką, bo wtedy nie będzie mogło fizycznie zakręcić na niektórych zakrętach.

6.3 Zwiększenie maksymalnej szybkości

To wpływa negatywnie na uczenie autka - z jednej strony ma ono więcej opcji na to z jaką szybkością może jechać, więc powinno móc wyciągnąć lepsze czasy na prostych, ale z drugiej jest je ciężej nauczyć, bo użyteczny range prędkości znormalizowanych jest mniejszy.

Sprawdziliśmy to eksperymentalnie – jeżeli samochodziki miały ustawioną bardzo dużą maksymalną prędkość, to jechały **znacznie** wolniej niż w przypadku wyższych wartości.