

Projekt

Metody probabilistyczne w uczeniu maszynowym

Łukasz Trzos, Jakub Wieliczko

0 Opis projektu

Tematem projektu jest implementacja bota zdolnego do autonomicznej gry w prostą, przygotowaną wcześniej grę polegającą na rywalizacji 1vs1. Projekt zakłada stworzenie środowiska dla wybranej gry, a następnie zaimplementowanie botów opartych o różne algorytmy z dziedziny uczenia maszynowego, w szczególności z dziedziny uczenia ze wzmacnianiem.

Główne cele projektu to:

- Porównanie skuteczności różnych algorytmów uczących dla naszego problemu poprzez analizę wyników osiągniętych przez boty
- Próba stworzenia bota będącego w stanie rywalizować z człowiekiem
- Śledzenie i analiza procesu uczenia się

Wykorzystanie tematu gry komputerowej w projekcie umożliwia ciekawą wizualizację działania poszczególnych algorytmów. Dzięki temu abstrakcyjne algorytmy mogą stać się bardziej zrozumiałe, a projekt może wzbudzać zainteresowanie i mieć wartość edukacyjną. Automatyzacja rozgrywki jest ważnym obszarem badań na temat sztucznej inteligencji. Nawet metody wypracowane dla całkowicie abstrakcyjnych gier znajdują zastosowanie między innymi w robotyce czy wszędzie, gdzie komputery muszą podejmować szybkie, sekwencyjne decyzje. Ponadto, inteligentne zachowania przeciwników znacząco przyczyniają się do wzrostu atrakcyjności samej gry.

O dokładnej formie wykorzystywanej w projekcie gry zdecydujemy w niedalekiej przyszłości. Gra będzie oparta na bardzo prostych zasadach, ale chcemy, żeby umożliwiała uczestnikom podejmowanie różnych ciekawych strategii tak, aby nie istniała tylko jedna główna ścieżka do zwycięstwa. Będzie to gra przeznaczona dla dwóch osób polegająca na rywalizacji 1vs1.

1 Opis gry

1.1 Zasady

Omówimy po krótce format samej gry. W grze bierze udział 2 rywalizujących ze sobą graczy, każdy z nich kontroluje jeden obiekt znajdujący się na arenie. Arena jest okrągła, gracze rozpoczynają grę w równej odległości od środka po przeciwnej stronie planszy. Gracze mogą poruszać się w 2 osiach, posiadają ustalone przyspieszenie oraz maksymalną osiągalną prędkość. Mogą również kolidować ze sobą, w takiej sytuacji przekazują sobie część zgromadzonego pędu, co wpływa na ich wypadkową prędkość. W razie wypadnięcia z mapy, gracz pojawia się na niej ponownie po 3 sekundach, w jednym z dwóch miejsc, w którym gracz pojawili się na początku. Wybierane jest to, które w klatce, w której ma się pojawić gracz, jest dalej od jego przeciwnika. (chyba, że przeciwnika na niej nie ma, wtedy jest to punkt pierwotnie przypisany do danego gracza).

Wzdłuż poziomej średnicy okręgu wyznaczającego teren rozgrywki rozmieszczone są 3 punkty kontrolne. W każdej klatce, w której dokładnie jeden z graczy znajduje się w danym punkcie (koło wyznaczające strefę oraz obszar zajmowany przez postać kontrolowaną przez gracza przecinają się niepusto) punktacja w strefie zmienia się o 1 punkt na korzyść wspomnianego gracza. W pewnym momencie, gdy wartość punktowa przekroczy ustalony próg, gracz **permanentnie** zajmuje strefę, nie można już w żaden sposób zmienić wtedy jej stanu.

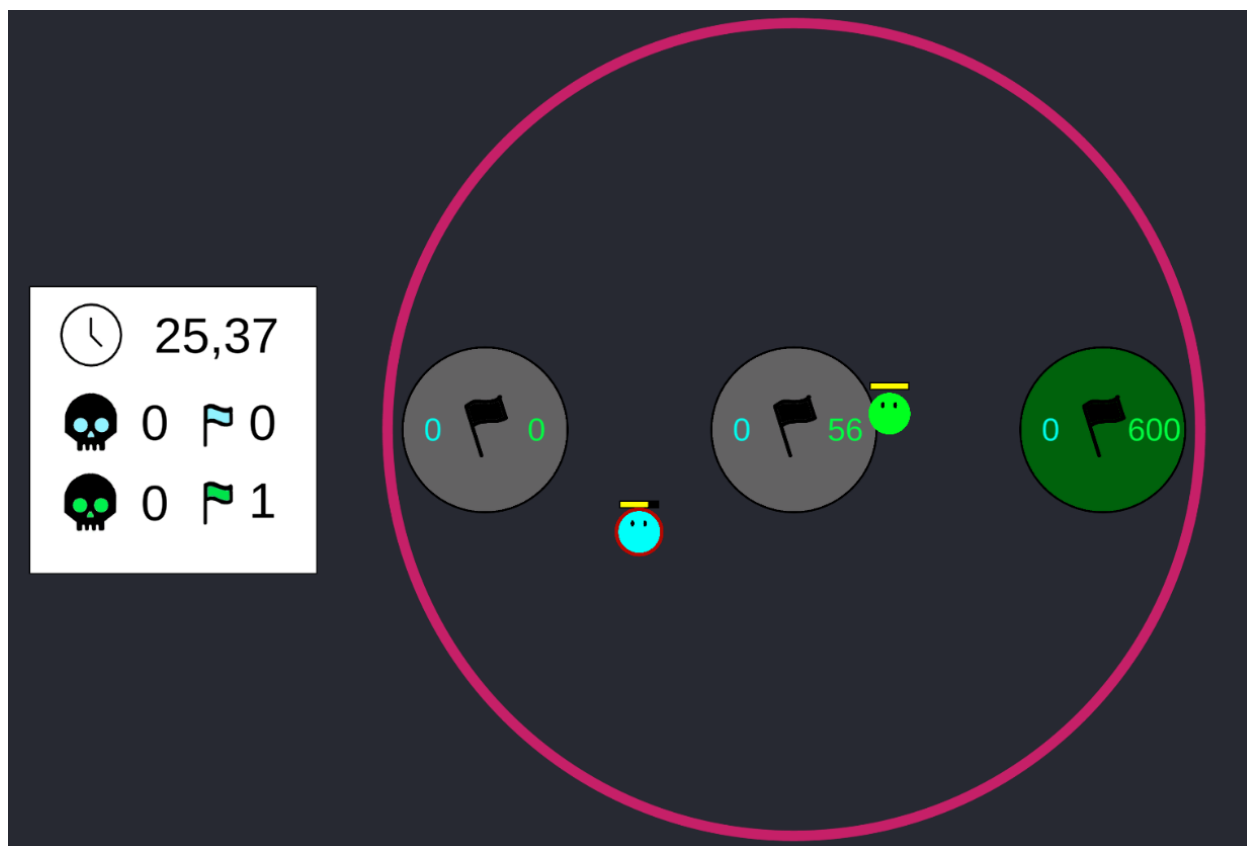
Każdy gracz dysponuje magazynem energii, która przychodzi z ustaloną, niezmienną ilością w każdej klatce rozgrywki. Pojemność magazynu, określająca maksymalną ilość przetrzymywanej na raz energii, również jest ustalonym parametrem. Gracz może wykorzystać swoją energię korzystając z dwóch umiejętności specjalnych:

- Wzmocnienie. Aktywuje się je jednorazowo poprzez wciśnięcie odpowiedniego przycisku. Podczas korzystania z niego, jego energia spada z czasem, wzmocnienie kończy się, gdy energia w magazynie spadnie do zera. Aby aktywować tę umiejętność gracz musi posiadać co najmniej ustaloną ilość energii, równą ok. $\frac{1}{3}$ pojemności całego magazynu. Podczas wzmocnienia wzrasta maksymalna osiągalna prędkość, przyspieszenie oraz moc odrzutu podczas zderzenia z drugim graczem dla obiektu kontrolowanego przez danego gracza.
- Strzał. Również aktywowany jednorazowo. Po wciśnięciu przycisku obok gracza tworzony jest pocisk poruszający się ze stałą, ustaloną prędkością w kierunku określonym podczas oddawania strzału. W razie uderzenia w przeciwnika pocisk przekazuje mu dużą część swojego pędu, często prowadząc do wyrzucenia przeciwnika z mapy lub przesunięcia go na mniej dogodną pozycję. Koszt oddania strzału to aktualnie $\frac{2}{3}$ maksymalnej pojemności magazynu. Po pojawieniu się na planszy gracz otrzymuje krótką ochronę przed pociskami, żeby uniknąć strategii polegającej na wyczekiwaniu bez możliwości odpowiedzi z drugiej strony

Gracz może zwyciężyć mecz na 2 sposoby:

- Przejęcie 2 z 3 punktów kontrolnych
- Trzykrotne wypadnięcie przeciwnika z areny

Ponadto, gra ma ograniczony czas trwania (ok. 10000 klatek, lub 3 minuty przy założeniu, że grę symulujemy w tempie 60 klatek na sekundę). Jeśli w ostatniej klatce żaden z graczy nie spełnił ani jednego wymagania, lub oboydwoje spełnili je jednocześnie, ogłaszany jest remis.



Rysunek 1: Wizualizacja planszy przedstawiającej przykładową grę

1.2 Ważne informacje

Wypiszę kilka najważniejszych cech opisanej gry, które mają decydujący wpływ w przypadku wyborów algorytmów uczących oraz możliwości realizacji projektu:

- Gra jest jedynie wizualizowana z użyciem silnika Unity, jednak sam jej kod jest kompletnie od niego niezależny. Aspekty fizyczne takie jak wykrywanie i obsługa kolizji, są napisane od zera. Dzięki temu grę można symulować znacznie szybciej niż w przypadku korzystania z pełnej oferty silnika, co oczywiście ma duże znaczenie przy prowadzeniu długich sesji treningowych.
- Sposobów na zwycięstwo lub zremisowanie przegranej sytuacji jest co najmniej kilka. W algorytmach decyzyjnych może więc występować sporo lokalnych maksimów. Wymusza to dopuszczanie dostatecznego poziomu eksploracji nowych strategii, aby boty mogły uczyć się ich i wybierać najlepsze cechy w procesie ewolucji.
- Prawdopodobnie nie można jednoznacznie stwierdzić czy dowolna, lokalnie podjęta przez gracza decyzja była w danej chwili optymalna, czy nie. Szkolone boty muszą więc być porównywane w inny sposób. Wykorzystane metryki zawierają wyniki uzyskiwane z innymi botami w grupie, z poprzednimi zwycięzami, pojedynki przeciwko wcześniej napisanym, prostym algorytmom decyzyjnym czy na podstawie pewnych heurystyk określonych na sytuacjach na planszy.
- Do kontrolowania obiektu gracza potrzebne są różne rodzaje danych. Algorytm musi umieć produkować dane z zakresu ciągłego (poruszanie się po planszy), jak i dyskretnego (korzystanie z umiejętności specjalnych)

2 Metody uczenia maszynowego

Zanim przejdziemy do algorytmu uczenia, opiszmy w jaki sposób skonstruowany jest model. Celem projektu jest napisanie modelu zdolnego do autonomicznej gry, zatem szukamy rozwiązania które dla każdego stanu gry odpowie na pytanie "jakie przyciski powinny być w tym momencie wciśnięte, aby zmaksymalizować nasze szanse dojścia do celu". Dlatego skupmy się na chwilę na opisanu przestrzeni wejść i wyjść.

2.1 Stan gry i opis akcji

Żałóśmy bardzo naturalnie, że my jako bot nie wiemy nic o strategii przeciwnika. Dlatego na potrzeby naszego projektu, wejściem dla naszego modelu będzie informacja która jednoznacznie ma opisać stan gry. Będziemy oczekiwać od naszego modelu przyporządkowania każdemu takiemu stanowi jednoznacznego zbioru akcji którą gracz powinien wykonać.

Dla zobrazowania problemu oto przykładowy zbiór wejść dla naszego modelu:

- i) Położenie, prędkości i kierunki ruchu obu graczy.
- ii) Stan planszy gry: zajętość stref, liczba zgonów obydwu graczy.
- iii) Ilość energii u obu graczy, informacja o wystrzelonych pociskach.

Dla tak zebranych danych, model powinien jednoznacznie zwrócić informację o następujących decyzjach:

- i) Kierunek i wartość prędkości gracza
- ii) Czy gracz powinien użyć przycisku strzału?
- iii) Czy gracz powinien użyć boost?

2.2 Opis modelu

Wśród danych wejściowych lista cech może być potencjalnie bardzo długa, co więcej dużo z tych danych jest ciągłych oraz ciężko jednoznacznie stwierdzić jak skuteczna jest podjęta decyzja. Do trzeciego problemu jeszcze wrócimy, ale odnosząc się do pierwszych dwóch dobrym wyborem w tym przypadku będzie jednokierunkowa sieć neuronowa (**Feedforward Neural Network, FNN**). Omówmy architekturę takiej sieci:

Na pierwszą warstwę naszej sieci będzie składać się stan gry, każdą z informacji będziemy kodować za pomocą liczb rzeczywistych z zakresu $(-1, 1)$. Na informację składa się:

Dla obydwu graczy:

1. Pozycja X, Y gracza (0 jeśli nie żyje).
2. Prędkość X, Y gracza (0 jeśli nie żyje).
3. Czy gracz żyje? $\{0, 1\}$
4. Ilość energii gracza.
5. Czy gracz ma boost / czy jest odporny na kule?
6. (opcjonalnie) Odległość gracza od środka planszy.

Ponadto informację o rozgrywce:

1. Stan wszystkich punktów kontrolnych
2. Ilość śmierci obydwu graczy
3. Informacja o pociskach, pozycja, prędkość

Łącząc to wszystko, warstwa wejścia ma 47 neuronów.

Przejdźmy teraz do opisu warstwy wyjścia. Będzie ona składać się z 4 neuronów, które przechowują kolejno następujące informacje:

1. Wartość ruchu w osi OX - przedział $[0, 1]$
2. Wartość ruchu w osi OY - przedział $[0, 1]$
3. Czy oddać strzał? - decyzja $\{0, 1\}$
4. Czy użyć boost? - decyzja $\{0, 1\}$

Żeby uzyskać prędkość w danej osi bierzemy wartość neuronu x i żeby mieć wartość z przedziału $\{-1, 1\}$ liczymy $y = 2x - 1$. Ze względu na to, że ciężko modelowi będzie podjąć decyzję o maksymalnej prędkości w danej osi, zdecydowaliśmy się żeby realną wartością prędkości był 1, gdy $y > 0.5$ a tak to \sqrt{y} , na minusie analogicznie.

Jeśli chodzi o warstwy pośrednie, sprawdzamy wyniki dla różnych ich głębokości i rozmiarów. To samo tyczy się funkcji aktywacji na kolejnych warstwach naszej sieci. Dla tak ustalonych hiperparametrów sieci neuronowej nasz model będzie jednoznacznie wyznaczany przez ciąg jej wag. Zanim przejdziemy do algorytmów, skupmy się jeszcze na chwilę na parametrach naszej sieci.

2.3 Architektura sieci

Architektura warstw ukrytych pełni szczególną rolę jeśli chodzi zarówno o przebieg uczenia, jak i wyniki końcowego modelu. Stosujemy sieć, w której dla każdego neuronu ma on osobną krawędź z wagą do wszystkich neuronów w kolejnej warstwie, zatem im szersze są warstwy tym szybciej rośnie liczba wag które chcemy znaleźć w oparciu o algorytmy uczenia. Jednak również ważnym aspektem jest głębokość sieci. Dla przykładu, ucząc sieci o jednej warstwie ukrytej 50 neuronów boty nauczyły się żeby iść do środka jednak dalsze ich uczenie nie przynosiło rezultatów. Z tego względu najlepiej sprawdzały się sieci trochę węższe jednak głębsze. Finalnie rozpatrywane modele miały 2 – 3 warstwy ukryte, np $[12, 8]$, $[16, 8, 4]$.

Dodatkowo należy wspomnieć o wyborze funkcji aktywacji do naszych modeli. Dla warstw ukrytych najwięcej testów przeprowadziliśmy używając funkcji **ReLU**, ale również sprawdzaliśmy jak modele będą się uczyć przy użyciu **tangensa hiperbolicznego**. Jednakże nie było silnych argumentów dla żadnej z funkcji, w obu przypadkach modele potrafiły wytworzyć jakieś strategie.

Ze względu na to, że algorytm ma podejmować decyzję zarówno o strzale jak i o boocie, oraz zwracać prędkości w każdej z osi, na ostatniej warstwie zdecydowaliśmy się zastosować funkcję **sigmoid**, dzięki czemu każdej z akcji przydzielamy liczbę z zakresu $[0, 1]$ którą następnie możemy dalej interpretować.

Pomysłem który sprawdziliśmy było również upraszczanie warstwy wejściowej. Motywacja za tym stała taka, że w początkowych fazach treningu modele za bardzo skupiały się na "mniej ważnych" cechach, co mogło zaburzać proces uczenia. Redukcja cech jednak nie przyniosła żadnych satysfakcjonujących rezultatów, a wyczerpanie na mniej istotne cechy malało wraz z długością treningu. Omówmy teraz metody jakich użyliśmy do trenowania naszych sieci.

2.4 Motywacja

Najpierw wspomnijmy tylko jakie podejścia uznaliśmy za niemożliwe. Przede wszystkim, skoro uczymy grać w grę opartą o rywalizację, ciężko jest zadać inne kryteria niż po prostu wygrana. W tym celu chcemy uczyć bota grać tak aby wygrał. Dlatego przykładowo **uczenie przez wzmacnianie (Reinforcement Learning)** odpada, bo gracz podejmuje akcję co klatkę gry, i ciężko dla każdej takiej akcji jednoznacznie powiedzieć czy była dobra czy nie. W tym celu zamiast skupiać się na ocenie kolejnych akcji, postanowiliśmy skupić się na ocenie rezultatów gier, czyli ocenić model poprzez kumulację jego wyniku zebranego podczas jego gry, a nie konkretnie każdej akcji z osobna. Oczywiście nie odbiera nam to możliwości karania gracza za jego zachowania, ale nie potrzebujemy w tym celu być wyrocznią.

2.5 Algorytmy

Algorytm który wybraliśmy to **Algorytm Genetyczny**. Przestrzenią przeszukiwań będą ciągi wag dla ustalonych hiperparametrów sieci neuronowej i naszym celem będzie skonstruowanie takiego ciągu który maksymalizuje funkcję **fitness**, czyli ocenę tej sieci.

Na potrzeby przeszukiwania będziemy trzymać zbiór opisów modeli - genomów na który będziemy mówić populacja. Będziemy na niej stosować operacje: **selekcji**, **krzyżowania** i **mutacji**. Za pomocą tych operacji będziemy tworzyć kolejne pokolenia, czyli przekształcać ten zbiór żeby dostawać modele z coraz większą wartością **fitness**. Omówmy przebieg działania algorytmu oraz jak konkretnie działają poszczególne operacje:

1. Na początku każdej iteracji algorytmu wyznaczamy dla każdego modelu z populacji jego wynik **fitness**. Na bazie tych wyników będziemy chcieli skonstruować nową populację.
2. **Selekcja** będzie polegać na wybraniu genomów z obecnej populacji które będą również w następnej populacji.

3. **Krzyżowanie** to operacja która dostając dwa genomy tworzy nowy, czyli tworzy sieć neuronową o **takim samym** rozmiarze co obydwie wybrane sieci, przepisując wagi z jednej i drugiej sieci.

4. **Mutacja** to operacja która dostaje sieć i zamienia niektóre z jej wag. Algorytm dla każdej wagi

Jeśli chodzi o konkretne implementacje funkcji **fitness**, parametrów populacji i technik wyszukiwania, przetestujemy i opiszemy różne rozwiązania w dalszej części raportu.

2.6 Metody oceny botów

Przy porównywaniu wyników osiąganych w grze przez różne wytrenowane algorytmy musimy posługiwać się odpowiednio dobraną metryką. Wszystkie algorytmy będą bazować na idei maksymalizowania tego wskaźnika, zatem musi być on starannie dobrany.

Po pierwsze, bota nie powinniśmy oceniać jedynie za końcowy efekt jego gry, powinniśmy się przyglądać akcjom, które do niego doprowadziły. Jeśli ocena będzie polegała na wąskiej liczbie zero-jedynkowych kryteriów, takich jak wygrana/przegrana lub osiągnięcie odpowiedniej liczby punktów w strefach kontrolnych, algorytmy oparte na niewielkich modyfikacjach nie będą w stanie zwiększać systematycznie wartości tego wskaźnika. Umożliwiamy więc ocenianie bota "klatka po klatce".

Wadą oceny bota za końcowy efekt jest również potencjalne nagradzanie za błędy przeciwnika. Na początku korzystaliśmy z funkcji, która nagradza boty za każde wypadnięcie przeciwnika z planszy, przez co otrzymywały wyższy wynik punktowy grając przeciwko botom, które same z siebie wypadały z mapy. Podobnie, boty otrzymywały bonus za to, że pocisk przeciwnika nie leciał w ich stronę, z myślą nauczania ich unikania, jednak nie powinny one być nagradzane za to, że przeciwnik strzela w ścianę.

Tworzyliśmy funkcje oceniające stan gry na podstawie wyżej wymienionych uwag. Każdy stan miał przypisywaną wartość punktową mówiącą o tym, jak dobra jest sytuacja każdego gracza oraz jak dobrze zachował się w danej klatce (w przypadku akcji takich, jak oddanie strzału). Gracz mógł otrzymywać punkty w oparciu o rozmaite heurystyki, takie jak:

- Odległość od środka planszy
- Odległość od najbliższego, możliwego do zdobycia punktu kontrolnego
- Wyniki zgromadzone w punktach kontrolnych oraz ewentualne zajęte punkty
- Wykorzystywanie maksymalnej prędkości podczas poruszania się na dłuższe dystanse
- Oddawanie strzału w stronę przeciwnika (dodatkowo gracz tracił punkty za oddawanie kompletnie nietrafionych strzałów)

Wymienione parametry oczywiście nie określają jednostajnie tego, czy dany ruch był dobry, ponieważ w różnych sytuacjach mogła być konieczna reakcja gorsza pod względem danej heurystyki, jednak zazwyczaj lepsza wartość tych parametrów oznacza lepszą sytuację. Przykładowo, podczas próby wyrzucenia przeciwnika z planszy, nasza odległość od środka planszy rośnie, ale i tak będzie ona "lepsza" od przeciwnika zakładając, że znajduje się blisko krawędzi podczas próby wyrzucenia z planszy.

Wszystkie te parametry były oceniane w sposób ciągły tak, aby podczas trenowania nowych botów widoczne były lekkie poprawy wyniku po coraz lepszym opanowywaniu docelowej umiejętności. Dla przykładu, punkty za oddanie strzału zależały liniowo od kąta pomiędzy wektorem różnic pozycji gracza atakowanego i atakującego i wektorem kierunku poruszania się pocisku. Im mniejszy kąt, tym lepszy strzał. Ważne, żeby taki system oceny nie był zero-jedynkowy, aby zachęcać algorytm do stopniowych modyfikacji parametrów i, wynikowo, oddawanie coraz celniejszych strzałów.

Dodatkowo, podczas pierwszych procesów treningowych odkryliśmy, że boty powinny za każdym razem grać ze sobą dwumecz tak, aby każdy grał każdym kolorem tyle samo razy. Zaobserwowaliśmy, że boty w sytuacji pojawienia się na miejscu gracza zielonego wygrywają znaczącą większość gier, i starają się to robić coraz szybciej, natomiast boty niebieskie zazwyczaj kręciły się w miejscu, nie podejmując walki. Działo się tak dlatego, że do następnych iteracji algorytmu przechodziły głównie boty, które większość lub wszystkie mecze rozegrały jako gracz zielony.

W dalszej części raportu omówimy dokładniej różne strategie których spróbowaliśmy i ich wyniki

3 Opis i porównanie zaimplementowanych algorytmów

W celu wybrania optymalnej strategii w poniższym rozdziale opiszemy przebieg i wyniki treningu dla różnych implementacji naszych funkcji. Celem tej części jest nakreślenie z jakimi trudnościami spotkaliśmy się podczas trenowania naszych modeli oraz opisanie naszych prób rozwiązania ich. Na samym wstępie warto wspomnieć, że wszystkie algorytmy napisane zostały w języku C# bez korzystania z zewnętrznych bibliotek z dziedziny uczenia maszynowego.

3.1 Algorytm Ewolucyjny

Jednym z najdłużej działających eksperymentów był kod realizujący algorytm ewolucyjny, czyli szczególną formę wspomnianego już wcześniej algorytmu genetycznego. Optymalizacja parametrów sieci neuronowej odbywała się poprzez stopniowe wylanianie najlepszych "wojowników" (botów sterowanych kontrolerami opartymi o sieci), które rywalizowały ze sobą w symulowanym środowisku.

Algorytm swoją nazwę zawdzięcza inspiracji wszechobecną biologiczną ewolucją, w której mechanizmy selekcji i krzyżowania najsilniejszych osobników mają doprowadzić do jak najdłuższego przetrwania gatunków.

3.1.1 Inicjalizacja populacji

Początkowo ładujemy do puli "currentWarriors" najlepsze dotychczas znalezione genomy. Dzięki temu możemy rozpoczynać kolejne procesy wykorzystując postępy poczynione w innych, niezależnych eksperymentach. Pierwszym, bardzo ważnym parametrem algorytmu jest rozmiar grupy jednocześnie porównywanych botów, czyli w naszym przypadku, rozmiar tablicy currentWarriors. Zbyt mała wartość prawdopodobnie doprowadziłaby do niewielkich szans na pokonanie "elity" przez nowo generowane genomy, wstrzymując tym samym proces ewolucji. Z kolei zbyt duża, jak się przekonałismy, znacząco wydłuża czas uczenia się, a wcale nie daje szczególnie lepszych efektów, bo dużo czasu poświęcamy na pojedynki botów, których geny i tak nie zostaną przekazane. W przeprowadzanych przez nas eksperymentach ostatecznie zdecydowaliśmy się na wartość parametru oscylującą wokół 50. Jeśli gotowych genomów było mniej, resztę slotów inicjalizowaliśmy jako genomy losowe.

3.1.2 Pojedynki oraz punktowanie genomów

Dla danej populacji chcemy stwierdzić, które boty radzą sobie najlepiej, aby móc wykorzystać ich najlepsze cechy w dalszym procesie nauczania. Zdecydowaliśmy, że najlepiej będzie, gdy każdy bot rozegra na każdego potencjalnego przeciwnika pełną grę, w której będziemy mogli go ocenić za podejmowane akcje oraz ich wyniki.


Tutaj ważną kwestią, i jednym z popełnionych początkowo błędów, jest fakt, że boty powinny grać ze sobą dwumecz, tak, aby każdy zagrał dokładnie raz każdym kolorem gracza. Ze względu na specyfikę gry, z perspektywy sieci neuronowej gracz niebieski musi w początkowych chwilach podejmować zupełnie inne decyzje niż gracz zielony. Oczywiście, dla nas, ludzi jest to zupełnie naturalne, widzimy że te decyzje są de facto symetryczne, jednak sieci na razie nie mają możliwości takiego wnioskowania. W jednej z instancji eksperymentu boty grały tylko 1 mecz ze sobą, w szczególności, bot o wyższym ID na liście (przez co zazwyczaj lepszy) zawsze grał kolorem zielonym. Sieć nauczyła się wygrywać bardzo szybko jako gracz zielony (wygrana przez strefy zajmowała jej ok. 30 sekund, podczas, gdy minimalny czas potrzebny na ich zajęcie wynosi 20 sekund), natomiast genomy odpowiadające za poprawę jakości gry gracza niebieskiego nie były w stanie przebić się na tyle wysoko w tabeli, by być uwzględniane w dalszych procesach ewolucyjnych.

Tak jak już wcześniej wspominalismy, botów nie będziemy oceniać jedynie za wynik końcowy ich meczów, ze względu na nieciągłość tego parametru oraz niewielkie znaczenie w kontekście nauczania. Nie chcemy przecież nagrodzić jednego bota za szybkie zwycięstwo tylko dlatego, że jego przeciwnik nie wiedzieć czemu postanowił wyjść sobie z areny. Do oceny botów służyły funkcje z modułu "ModelEvaluationFunctions", które brały jako argument parę wojowników, symulowały pełną grę z ich udziałem i zwracały ich ocenę, obliczoną na różnorakie sposoby.

Po zakończeniu wszystkich pojedynków turniejowych, genomy są oceniane za sumę zgromadzonych podczas wszystkich gier punktów. Wyższa wartość oznacza lepsze przystosowanie. Tutaj szczególnie istotne jest to, żeby faktycznie lepsza wartość punktowa implikowała skuteczniejszą grę, czyli żeby funkcja ewaluująca była starannie dobrana do eksperymentu. Jej szczegółowe implementacje wraz z sukcesami i błędami omówimy w innym podrozdziale.


```
1 private void TryLoadBestGenomes()
2 {
3     string bestGenomesDir = "./ArenaExperiment/BestGenomes";
4
5     if (!Directory.Exists(bestGenomesDir))
6         return;
7
8     var files = Directory.GetFiles(bestGenomesDir, "Genome_*.json");
9
10    if (files.Length == 0)
11        return;
12
13    // Sort files by the integer value after "Genome_"
14    var topFiles = files
15        .Select(f => new { File = f, Num = ExtractNumberFromFilename(f) })
16        .OrderByDescending(f => f.Num)
17        .Take(groupSize)
18        .ToList();
19
20    currentWarriors.Clear();
21
22    foreach (var file in topFiles)
23    {
24        if (currentWarriors.Count < groupSize)
25        {
26            currentWarriors.Add(Genome.Load(file.File));
27            currentPoints.Add(0f);
28        }
29    }
30
31    while (currentWarriors.Count < groupSize)
32    {
33        currentWarriors.Add(Genome.CreateNew(midLayerSizes));
34        currentPoints.Add(0f);
35    }
36
37    // Update the currentTournamentID to be one more than the highest found
38    if (topFiles.Any())
39    {
40        currentTournamentID = topFiles.Max(f => f.Num) + 1;
41    }
42    else
43    {
44        currentTournamentID = 1;
45    }
46 }
```

Rysunek 2: Funkcja inicjalizująca eksperyment

A screenshot of a code editor with a dark background and light-colored text. The code is in C# and defines a method named RunOneTournament(). The code is numbered from 1 to 35 on the left side. The method starts with a console message, initializes a queue, and then enters a loop where it processes matches between players. It uses a List<NNBot> for tournament players, a Queue for the match queue, and a List<float> for match results. The method ends with a call to CreateNewGeneration().

```
1 public void RunOneTournament()
2 {
3     Console.WriteLine("New tournament starting!");
4     InitializeQueue();
5     Console.WriteLine($"There will be {matchQueue.Count} matches!");
6
7     List<NNBot> tournamentPlayers = new List<NNBot>();
8
9     for (int i = 0; i < groupSize; i++)
10    {
11        tournamentPlayers.Add(new NNBot(currentWarriors[i]));
12        currentPoints[i] = 0f;
13    }
14
15    int currentMatchNumber = 1;
16
17    while (matchQueue.Count > 0)
18    {
19        currentMatchNumber++;
20
21        var top = matchQueue.Peek();
22        matchQueue.Dequeue();
23
24        int X = top.x;
25        int Y = top.y;
26
27        List<float> matchResult = ModelEvaluationFunctions.ModifiedOverallExperiment(
28            tournamentPlayers[X], tournamentPlayers[Y]);
29
30        currentPoints[X] += matchResult[0];
31        currentPoints[Y] += matchResult[1];
32    }
33
34    CreateNewGeneration();
35 }
```

Rysunek 3: Funkcja odpowiedzialna za przebieg turnieju

3.1.3 Proces reprodukcji

Uzyskując listę genomów z przypisanymi im wynikami chcemy w jakiś sposób utworzyć nową pulę 50 osobników do rozegrania kolejnego turnieju. Wykorzystamy tutaj kilka technik często stosowanych w algorytmach ewolucyjnych.

Selekcja elitarna: Pewna liczba genomów danego turnieju, które osiągnęły najwyższe wyniki, awansuje bezpośrednio do kolejnych iteracji. Dzięki temu mamy pewność, że w dalszych eksperymentach bierzemy pod uwagę dotychczas najlepiej sprawujące się sieci, a znalezione dla nich usprawnienia na pewno nie zostaną utracone. Zazwyczaj rozmiar elity ustalany jest jako 5 – 10% rozmiaru całej grupy.

Selekcja turniejowa: Pozostałe miejsca zajmą boty stanowiące mutacje lub krzyżówki zawodników turnieju, przy czym będziemy chcieli, aby geny były dobierane z większym prawdopodobieństwem od osobników o wyższych wynikach punktowych. Będziemy tu korzystać z opisanych wcześniej operacji krzyżowania oraz mutacji dla genomów.

Po pierwsze, ustalamy globalny parametr **crossover rate**. Mówi on, jak często nowo tworzone boty będą uzyskiwane poprzez skrzyżowanie dwóch uczestników turnieju. W naszym eksperymencie ustaliliśmy ten parametr dokładnie na wartość $\frac{1}{2}$. Jeśli wylosujemy, że ma nastąpić krzyżowanie, to w jaki sposób wybrać dwa boty w nim uczestniczące? Szeroko stosowanym rozwiązaniem jest wylosowanie niezależnie k botów z rozkładem jednostajnym, a następnie wybranie spośród nich tego, który uzyskał w turnieju najwyższy wynik. Dzięki temu faworyzujemy boty prawdopodobnie lepsze, ale dopuszczamy również krzyżówki tych nieco słabszych. Ważne jednak, żeby k nie było zbyt duże, wtedy możemy w ogóle nie dopuszczać algorytmów implementujących nowe podejścia nawet, jeśli zajmą one stosunkowo wysokie miejsca w rozegranym turnieju.

W przeciwnym wypadku, gdy nie następuje krzyżowanie, punktem startowym dla nowego zawodnika będzie po prostu pojedynczy uczestnik turnieju. Jest on wybierany na dokładnie takich samych zasadach jak boty przeznaczone do krzyżowania, z tym samym parametrem k .

Następnie, mając już gotowy punkt startowy, dokonujemy na nim operacji mutacji. Każdą wagę (lub ich losowo wybrany podzbiór) biorącą udział w procesie decyzyjnym danej sieci neuronowej, alterujemy zwiększając ją o zmienną losową otrzymaną z rozkładu Gaussa z ustalonym wcześniej parametrem **siły mutacji**. Jego wartość jest kluczowa w całym procesie reprodukcji, ponieważ to tutaj sieci są stopniowo lekko poprawiane w celu coraz lepszej gry. Początkowo przyjęliśmy zdecydowanie zbyt wysoką wartość tego parametru, przez co bardzo rzadko udało się znaleźć lepszego bota. W teorii większy parametr oznacza większy potencjalny zysk, jednak prawdopodobieństwo otrzymania takiego rezultatu drastycznie maleje. Małe wartości parametru, mimo wolniejszej zbieżności, znacząco ułatwiają stopniowe szukanie miejsc na niewielkie potencjalne usprawnienia. Oczywiście, przesadzać również nie można, najlepiej dobrać jak najwyższy parametr, ale na tyle niski, żeby regularnie znajdować lepszych zawodników. Po eksperymentowaniu i notowaniu wyników kolejnych turniejów dla różnych jego wartości stwierdziliśmy, że powinien on znajdować się w okolicach 0.01.

Po utworzeniu takiej liczby botów, żeby ich łączna ilość znów wynosiła 50, zerujemy wszystkie wyniki i rozpoczynamy nowy turniej. W międzyczasie zapisujemy dokładne wartości wag w pliku json pozwalające rekonstruować sieć odpowiedzialną za grę zwycięzcy turnieju, oraz osiągniętą przez nią sumę wyników w celach porównawczych. Oczywiście, wyższa wartość tego wyniku nie zawsze będzie oznaczać lepszego bota, gdyż zależy od siły przeciwników. W szczególności, niski wynik zwycięzcy nie świadczy o pomyłkach algorytmów, a częściej o powolnym wykształcaniu się nowej strategii, przez którą poprzedni zwycięzcy nie mogą już tak łatwo uzyskać przewagi.

Jak można zauważyć, w eksperymencie występuje dość dużo parametrów, z czego każdy ma ogromne znaczenie dla przebiegu eksperymentu. Na niżej umieszczonym rysunku, parametrami są: *winnerGroupSize*, *groupSize*, *crossoverRate*, *mutationStrength* oraz k zaszyte w kodzie funkcji *SelectParent()*. Każdy z nich zmieniał się kilkakrotnie podczas prowadzenia eksperymentów.

```
1 private void CreateNewGeneration()
2 {
3     List<KeyValuePair<Genome, float>> evaluatedGenomes =
4     currentWarriors.Zip(currentPoints, (genome, points) =>
5     new KeyValuePair<Genome, float>(genome, points))
6         .OrderByDescending(kvp => kvp.Value)
7         .ToList();
8
9     //Zapisanie najlepszego zawodnika oraz jego wyniku w odpowiednich miejscach
10
11     currentTournamentID++;
12
13     List<Genome> nextGeneration = new List<Genome>();
14
15     for (int i = 0; i < Math.Min(winnerGroupSize, evaluatedGenomes.Count); i++)
16     {
17         nextGeneration.Add(evaluatedGenomes[i].Key.Clone());
18     }
19
20     while (nextGeneration.Count < groupSize)
21     {
22         Genome parent1 = SelectParent(evaluatedGenomes);
23         Genome parent2 = SelectParent(evaluatedGenomes);
24
25         Genome offspring;
26
27         if (RANDOM.NextDouble() < crossoverRate)
28         {
29             offspring = Genome.Crossover(parent1, parent2);
30         }
31         else
32         {
33             offspring = parent1.Clone();
34         }
35
36         offspring = Genome.Mutate(offspring, mutationStrength);
37
38         nextGeneration.Add(offspring);
39     }
40
41     currentWarriors = nextGeneration;
42
43     currentPoints.Clear();
44     for (int i = 0; i < groupSize; i++)
45     {
46         currentPoints.Add(0f);
47     }
48 }
49
```

Rysunek 4: Funkcja odpowiedzialna za utworzenie nowej generacji

3.2 Algorytm wspinaczkowy

Jest to alternatywny zaimplementowany algorytm. Jego ideą jest ciągle doskonalenie i ulepszanie pojedynczego bota określanego mianem "najlepszego". Szkolimy go w taki sposób, że każdy kolejny bot musi pokonać listę wcześniej wyselekcjonowanych, silnych przeciwników o ustalonym od góry rozmiarze. Początkiem algorytmu jest tzw. "bot zerowy", który stoi w miejscu i nie wykonuje żadnych akcji. Sukces bota mierzony jest więc wyłącznie na podstawie porównania wyników osiągniętych w grach przeciwko poprzednim czempionów.

3.2.1 Inicjalizacja

Dokładnie, jak w przypadku omówionego wcześniej algorytmu ewolucyjnego, korzystamy z listy gotowych parametrów sieci neuronowych odpowiadających najlepszym genomom znalezionym w innym eksperymencie. Jeśli lista jest pusta, to jak wcześniej wspomnieliśmy, zaczynamy z botem nie podejmującym żadnych działań.

Tutaj ważne jest ustalenie ilości poprzedników do pokonania. Warto zaznaczyć, że przez pokonanie nie rozumiemy zwycięstwa jako ostateczny wynik gry, a jedynie uzyskanie lepszego wyniku punktowego według stosowanej funkcji ewaluującej, nawet jeśli różnica jest bardzo niewielka. Dlatego algorytm miał w ogóle szanse zadziałać, zupełnie losowy bot prawdopodobnie nigdy nie pokonałby bota stojącego w miejscu, który de facto nie popełnia żadnych poważnych błędów. Dlatego ten parametr mógł mieć dość dużą wartość, eksperymentowaliśmy w wieloma możliwościami, zwykle w okolicach 10 – 25.

3.2.2 Pojedyncza iteracja

W jednej metodzie wywoływanej cyklicznie mamy za zadanie spróbować utworzyć nowego bota. Robimy to w taki sposób: wybieramy jednego z botów obecnych w naszej puli wojowników do pokonania i wywołujemy na jego genomie operację mutacji, z wcześniej wybranym współczynnikiem odchylenia standardowego. Tutaj o wiele lepiej sprawdzały się niewielkie wartości, ponieważ sam algorytm bazuje na niewielkich, stopniowo wprowadzanych usprawnieniach.

Po wyłonieniu kandydata na dodanie do puli, symulujemy jego rozgrywkę z każdym botem z wcześniej wspomnianego zbioru. Różnica od poprzedniego algorytmu polega na tym, że nie patrzymy jedynie na sumę, a wymagamy, by każdy bot został pokonany. Jeśli mu się to uda, wstawiamy go na początek listy, a jego genom zapisywany jest w formacie json w celu umożliwienia odtworzenia odpowiadającej mu sieci neuronowej.

Algorytm próbuje również dostosowywać siłę mutacji do uzyskiwanych rezultatów. Jeśli przez odpowiednio długi czas algorytm utknął, i nie znalazł żadnego lepszego bota, to potencjalnie znajduje się w lokalnym minimum lub na płaskowyżu. W takim przypadku zwiększana jest siła mutacja, ma to na celu przeskoczenie lokalnych problemów. Po każdym dopisaniu nowego bota do listy, współczynnik jest resetowany do jego domyślnej wartości. Ma to na celu znalezienie równowagi pomiędzy stosowaniem mechanizmów **eksploatacji**, czyli precyzyjnego dostrajania aktualnie optymalnego rozwiązania, oraz **eksploracji**, czyli poszukiwania potencjalnie lepszych rozwiązań po wykryciu stagnacji. Przez domyślną wartość rozumiemy tę samą, którą uznaliśmy za optymalną podczas eksperymentu ewolucyjnego, czyli 0.01.

Napomnę, że implementując taki algorytm warto wcześniej pomyśleć również o ustaleniu maksimum. Trochę się zdziwiłem, kiedy eksperyment przez bardzo długi czas nie znalazł żadnej poprawy, jeszcze wtedy nie wiedząc, że operuje na czterocyfrowej wartości odchylenia standardowego...

```
1 private void RunOneExperiment()
2 {
3     Genome testedGenome;
4
5     if (currentBestGenome == null)
6     {
7         testedGenome = Genome.CreateNew(midLayerSizes);
8     }
9     else
10    {
11        testedGenome = Genome.Mutate(currentBestGenome, currentMutationDeviation);
12    }
13
14    GamePlayerController nnbot = new NNBot(testedGenome);
15
16    bool undefeatable = true;
17
18    foreach (GamePlayerController opponent in currentWarriors)
19    {
20        List<float> performance = ModelEvaluationFunctions.OverallExperiment(nnbot, opponent);
21
22        if (performance[0] <= performance[1])
23        {
24            undefeatable = false;
25            break;
26        }
27    }
28
29    if (undefeatable)
30    {
31        currentWarriors.Insert(0, nnbot);
32
33        currentTournamentID++;
34
35        currentMutationDeviation = MinMutationDeviation;
36    }
37    else
38    {
39        stagnationCounter++;
40        if (stagnationCounter > StagnationThreshold)
41        {
42            currentMutationDeviation += MutationIncreaseFactor;
43            currentMutationDeviation = Math.Min(currentMutationDeviation, MaxMutationDeviation);
44            stagnationCounter = 0;
45        }
46    }
47 }
```

Rysunek 5: Pojedyncza iteracja nowego eksperymentu

3.2.3 Uzyskane wyniki

Na samym wstępie podrozdziału warto powiedzieć, że eksperyment poradził sobie znacznie gorzej od pozostałych. Co prawda był jednym z prostszych do zaimplementowania, a mechanizm wykrywania stagnacji pozwolił na naprawdę szybkie znajdowanie nowych najlepszych zawodników. Jednak algorytm ten posiada kilka kluczowych cech, które po prostu nie najlepiej współgrały z charakterem zaimplementowanej przez nas gry:

Przede wszystkim, bardzo dużo utraciliśmy na różnorodności genetycznej. Algorytm skupiał się na mutacji garstki najlepszych botów, nie dopuszczał on do ich grona nowych, potencjalnie przełomowych zawodników. Nastąpiło zjawisko **przedwczesnej konwergencji**, polegające na znalezieniu lokalnego optimum i kurczowym trzymaniu się wartości parametrów sieci, dla których jest ono osiągnięte.

Przykładowo, w naszej grze boty nauczyły się strategii, którą nazwaliśmy "powolnym umieraniem". Każdy obierał sobie za cel jedną strefę, następnie poruszał się w jej kierunku z bardzo wolną prędkością. Prędkość była na tyle wolna, że udało mu się ją zajmować, zdobywając przy tym ogromny bonus od funkcji ewaluującej, a optymalizacja polegała na tym, by zatrzymać się jak najbliżej krawędzi mapy.

Dopuszczenie kompletnie nowych strategii byłoby jednak w przypadku naszej gry ciężkie, bo do zwycięstwa może prowadzić bardzo wiele ścieżek. Ciekawym pomysłem byłoby puszczenie algorytmu kilkakrotnie w taki sposób, żeby utknął w różnych punktach optimum, a następnie zestawienie ze sobą wynikowych botów i przeprowadzenie na nich operacji krzyżowania. Jednak wymuszanie znalezienia się w innym optimum jest dość trudnym procesem.

Algorytm wspinaczkowy zazwyczaj lepiej się sprawdza w prostych grach. W grze, z której zaczerpnęliśmy pomysł na jego implementację, przestrzeń wejść była ograniczona jedynie do ruchu w poziomie, sama koncepcja gry również była prosta. Być może algorytm zadziałałby lepiej, gdyby pozbyć się stref i nagradzać jedynie za wyrzucanie przeciwników, lub zrobić to na odwrót.

Po drugie, wymóg bycia absolutnie najlepszym trochę za bardzo ograniczał możliwe ścieżki rozwoju, niekoniecznie w sposób dobry dla ogółu. W eksperymencie z areną, bot nie musiał pokonywać absolutnie wszystkich. O posiadaniu dobrej cechy, z której warto korzystać przy tworzeniu nowych populacji, świadczył chociażby wysoki wynik, który dawał większe możliwości znalezienia się w przyszłych turniejach. Nasz algorytm był jednak dość rygorystyczny. Zauważenie nowej strategii najczęściej wymaga tego, że stosujące ją boty początkowo nie będą najlepsze, ale z biegiem czasu przechodząc do kolejnych rund i krzyżując się nauczą się, jak w optymalny sposób wprowadzić ją do gry. Nasz eksperyment czegoś takiego nie dopuszczał, co jest ogromnym minusem dla gry, w której jest bardzo dużo dostępnych akcji i ścieżek do zwycięstwa lub niwelowania przewagi przeciwnika.

4 Opis i porównanie funkcji ewaluujących skuteczność botów

4.1 Zarys funkcji i popełnione błędy przy jej definiowaniu

Wielokrotnie mówiliśmy o tym w jaki sposób ocena botów wpływa na dalszy przebieg algorytmów, oraz omówiliśmy jak z grubsza wygląda szkielet funkcji ewaluującej. W tym rozdziale przyjrzymy się szczegółom implementacyjnym, znalezionym innowacjom, popełnionym błędom i metodom ich korekcji.

Koncepcja funkcji jest prosta: Każda z nich przyjmuje 2 parametry klasy `GamePlayerController`, każda z nich zwraca również listę zmiennych typu `float` stanowiących oceny botów. Ich struktura pozwala na przyglądanie się każdej grze klatka po klatce, przez co możemy wyciągać znacznie bogatsze wnioski na temat zachowań obydwu graczy. Mechanizm wygląda bardzo podobnie, tworzymy nową instancję gry, podmieniamy kontrolery, wywołujemy `Update()` aż stan zmieni się z trwającej rozgrywki na inny, ściągamy wszystkie potrzebne informacje i dokonujemy ewaluacji.

Funkcje te były wielokrotnie modyfikowane ze względu na wysoką podatność na błędy. Są one kluczowe w działaniu dowolnego algorytmu, de facto samodzielnie decydują, czy dany gracz będzie miał w ogóle szanse przekazać swoje geny, dlatego muszą być starannie zaimplementowane. Oto kilka błędów, które popełniliśmy:

- **Założenie, że wyniki graczy powinny sumować się do 0.** Na początku obliczając dowolną akcję wykonaną przez gracza, odejmowaliśmy drugiemu od wyniku równowartość oceny dokonanej akcji. Oczywiście konsekwencją było nieplanowane nagradzanie botów za błędy przeciwnika, które nie były w żaden sposób związane ze skutecznością ich gry. Jeden z pierwszych eksperymentów, oparty o funkcję ewaluującą o tym charakterze, zakończył się wielkim niepowodzeniem. Boty uznały, że obydwoje otrzymają solidne oceny, jeśli zaczną po prostu wylaatywać jak najszybciej z planszy. Dodatkowo, często jako najlepsi zawodnicy byli wybierani słabi gracze, którzy nie robili poważnych błędów, ale również nie grali ani trochę logicznie.
- **Brak ciągłości w zależności otrzymanej nagrody od akcji.** Jeśli będziemy chcieli nagrodzić gracza jedynie za spełnienie jakiegoś wymogu w stu procentach, przy tym nie dając mu żadnej premii, gdy nie osiągnął całego założonego celu, ale się do niego zbliżył, to najprawdopodobniej algorytm w życiu nie zdoła tej nagrody otrzymać. Dzieje się tak dlatego, że dokonując małych zmian nie sprawimy nagle, że cel zostanie osiągnięty, a w tej sytuacji nie jesteśmy w żaden sposób nagradzani za podjęcie próby. W jednej z funkcji stosowaliśmy ogromną nagrodę za wygraną przez zajęcie dwóch stref, ze względu na to, że boty nie chciały tego robić. Jednak w przypadku, gdyby zabrakło botowi jednej klatki do przejścia strefy, nie otrzymałby on żadnej nagrody. Łatwo zauważyć, że taki system ani trochę nie promuje czynienia niewielkich postępów, oczekiwania były postawione zdecydowanie za wysoko.
- **Nieskuteczne metody punktacji za oddawane strzały.** Jest to dosyć specyficzna uwaga, opisemy więc przykłady. Najtrudniejszym zadaniem było stwierdzenie, od jakiego kąta chybień strzał powinien być nagradzany. Początkowo ten kąt miał zbyt dużą wartość, przez co najlepszą strategią było po prostu oddawanie losowych strzałów. Dodatkowo, oddanie strzału skutkowało dość nagłą zmianą parametru wejściowego w sieci (obecność pocisku zmieniała się z 0 na 1), a boty dość dziwnie zachowywały się podczas takiej zmiany. Metą w takiej konfiguracji stała się więc "taktyczna dezorientacja przeciwnika". Z kolei po dość drastycznym zmniejszeniu kąta oczywistym dla algorytmów wnioskiem stało się to, że przycisk strzału w zasadzie jest do niczego niepotrzebny, i wciśnięcie go wiąże się z otrzymaniem surowej kary. Próbowaliśmy również całkowicie ignorować otrzymywanie punktów za strzały. Ponieważ w funkcji mierzyliśmy inne, ściśle związane parametry takie jak odległość od strefy nasza oraz przeciwnika, umiejętne użycie pocisku w zasadzie doprowadzało i tak do zwiększenia się liczby punktów w tej kategorii. Przychód ten był jednak na tyle nieduży, że umiejętny strzał dawał mikroskopijna poprawę uzyskiwanego wyniku, podczas gdy istniały znacznie bardziej obfite jej źródła takie, jak umiejętne pozycjonowanie się oraz pozostawianie blisko środka mapy.

Ostatecznie stanęło na tym, że strzał był premiowany wtedy i tylko wtedy, gdy kąt błędu wynosił co najwyżej $\frac{\pi}{2}$. Dodatkowo, wartość premii była ciągła, i o wiele większa niż wartość kary w przypadku strzału poza tolerowanym zakresem. Nie jestem pewny, czy to przyniosło poprawę, ponieważ pokolenie trenowane przy takich zasadach nauczyło się z kolei bardzo obficie korzystać z umiejętności wzmocnienia, więc praktycznie nigdy nie mieli wystarczająco dużo zmagazynowanej energii na oddanie kosztownego strzału.

- **Brak wyrównania w wartościach nagród za poszczególne elementy rozgrywki.** Tego błędu prawdopodobnie nie da się w stu procentach wyeliminować. W rozgrywce, w której ważne jest stosunkowo dużo czynników, takich jak umiejętne pozycjonowanie się na planszy, kontestowanie stref, oddawanie celnych strzałów, rozważne korzystanie z zapasów energii i inne, możliwości zbalansowania nagród otrzymywanych za je wszystkie jest zadaniem trudnym. W większości eksperymentów, któraś metoda uzyskiwania punktów brała górę, a boty uczyły się zyskiwać punkty na jeden określony sposób. Próbowaliśmy również stosować podejście, w którym tworzymy osobne funkcje ewaluujące dla różnych elementów rozgrywki, a następnie okresowo podmieniamy je w algorytmach. Zazwyczaj nie działało to najlepiej. W przeciwieństwie do osiągania zamierzonego skutku, jakim było osiągnięcie równowagi, boty przerzucały całą swoją uwagę na nowo pojawiający się cel, bagatelizując dobre geny odpowiedzialne za dbanie o poprzednie warunki. Dla przykładu, początkowo kara za wypadnięcie z mapy była zdecydowanie zbyt niska. Najlepszą strategią wydawało się wtedy zrobienie sprintu przez cały obszar rozgrywki, przejmując przy tym kawałek strefy, a następnie bez zatrzymania się wylecenie na drugim końcu. Cały ten ciąg operacji skutkował wynikiem pozytywnym.
- **Brak bezpośrednich nagród za zabójstwa.** Ten błąd wynika głównie z tego, jak początkowo wyglądały rozgrywki, i czego chcieliśmy uniknąć. Oceniając grę klatka po klatce dość trudno jest stwierdzić, czy dane wylecenie z mapy było spowodowane bezpośrednio przez oponenta, czy było po prostu głupim błędem. Dodatkowo, statystyka ta jest skazana na brak ciągłości, bo ciężko stwierdzić jak bliskie było potencjalne wypadnięcie, które nie doszło do skutku. Twierdziliśmy również, że w zasadzie nagrody za zabójstwa lub bycie blisko pojawiają się w innych ocenianych parametrach takich, jak różnica odległości od środka mapy, czy stopniowo rosnące postępy w przejmowaniu stref. Nie premiowaliśmy również zwycięstw osiągniętych takim sposobem, ponieważ początkowo takie zwycięstwa zdarzały się w 99% gier, zatem wyniki były zdecydowanie wygórowane. (w jednej z pierwszych funkcji zwycięstwo dawało tak dużą liczbę punktów, której nie dało się zdobyć na wszelkie inne sposoby bez wygrywania. Z perspektywy czasu, była to wyraźnie nieprzemysłana decyzja).

4.2 Ostateczny wygląd wykorzystywanych funkcji

Ponieważ tych funkcji było naprawdę dużo, i nie wszystkie przetrwały do finalnej wersji projektu, ciężko jednoznacznie wskazać najlepszą. Wymienimy jednak cechy, które były obecne w większości z nich, doprowadzając do jak najlepszych wyników:

- Boty otrzymywały punkty za zmniejszanie odległości do najbliższej, niezajętej strefy. Maksymalnie mogły otrzymać 1 punkt na każdą klatkę, wartość ta zbliżała się do 0 gdy znajdowały się na pionowych brzegach areny. Dodatkowe punkty otrzymywał również bliższy bot, były one zależne od różnicy ich odległości do ważnych punktów, ich maksymalna wartość również wynosiła 1 per klatkę, natomiast w praktyce były to niższe wartości. Taki system zachęcał przede wszystkim do dbania o własny interes i kręcenia się wokół ważnych punktów, ale również z czasem do upewniania się, że przeciwnik znajduje się od nich dalej, niż my.
- Boty otrzymywały punkty za posiadane udziały w strefach. Maksymalnie można było zyskiwać pasywnie 1 punkt na klatkę za posiadanie w pełni zajętej strefy, ilość punktów malała liniowo do 0 wraz z miarą postępów w jej przejmowaniu. Teoretycznie przejęcie strefy nie musiało być wynikiem dobrych działań podjętych w danej klatce, natomiast nie zdecydowaliśmy się na system, w którym bot otrzymuje punkty wtedy, i tylko wtedy, gdy aktualnie zdobywa progress. Mogłoby to doprowadzić do zabawy w kotka i myszkę, w której boty nie starają się w żaden sposób bronić zdobytych wcześniej punktów, tylko na zmianę dają i odbierają żeby zyskiwać punkty.
- Premia za zwycięstwo oraz kara za przegraną na strefy zależna od czasu. Początkowo najbardziej zależało nam na zwycięstwach przez strefy. Wymagane 3 zabójstwa zdarzały się w praktycznie każdej początkowej grze, takie zwycięstwo nie było więc niczym niezwykłym. Zależność od czasu wprowadziła jednak potrzebną ciągłość. Dzięki temu, zarówno bot wygrywający miał na celu jak najszybsze zwycięstwo bez walki z drugiej strony, jak i bot przegrywający starał się opóźnić nadchodzący koniec meczu.

5 Wyniki eksperymentów

W tej sekcji skupimy się bezpośrednio na obserwowalnych zachowaniach botów.

W katalogu "Projekt/Report/Videos" umieszczamy filmiki zatytułowane "Gameplay_X", gdzie $X \in \mathbb{N}$. Ilustrują one przykładowe rozgrywki botów trenowanych na wszelakie sposoby, o każdym postaramy się jak najdokładniej opowiedzieć, wskazując na prawdopodobne przyczyny obserwowanych nienaturalnych zachowań botów, ale również punktując ich dobre strony. Jak już wynika z wniosków z poprzednich rozdziałów, najlepiej sprawdził się algorytm ewolucyjny. Był również najdłużej działającym algorytmem, dlatego większość materiałów wideo pochodzi właśnie od niego.

Na samym wstępie chcemy zaznaczyć, że trudno mówić o najlepszych uzyskanych wynikach w sytuacji, gdy podczas pisania raportu eksperymenty są nadal aktywne, a każdy dzień przynosi nowe, niespodziewane zmiany w zachowaniu botów. Cieszymy się, że mamy ogromne pole do prowadzenia dalszych procesów nauczania i analizowania wyników. O wynikach zamieszczonych w tym raporcie należy myśleć w taki sposób, że były one z edukacyjnego punktu widzenia najciekawsze lub najlepsze do przedstawienia publiczności. Jeśli to będzie możliwe, postaramy się je co jakiś czas aktualizować uwzględniając najnowsze rezultaty.

5.1 Analiza materiałów wideo

5.1.1 Gameplay_1

Jest to jeden z pierwszych powstałych materiałów, zupełnie na początku procesu treninowego. Największym problemem, dającym się zauważyć po zachowaniu botów, był tutaj niewielki rozmiar sieci. Sieć, na której oparte były zachowania przedstawione na filmie, miała tylko jedną ukrytą warstwę, a na dodatek znajdowało się w niej niewystarczająco dużo neuronów (12 lub 15). Dawało to zbyt mało możliwości wykształcenia nowych strategii w sytuacji, gdy populację przejmował wadliwy gen, lub utykała w lokalnym optimum.

Ten eksperyment wykorzystywał funkcję ewaluującą opartą o promowanie celności strzałów oraz pozostawanie na planszy jak najdłużej, ale również końcowy wynik, co, jak wcześniej wspomnieliśmy, nie było najlepszym pomysłem. Jak widać, w początkowej fazie rozgrywki, strzały są całkiem celne. Proces uczenia ładnie ilustruje przykład, w którym gracz niebieski podąża swoim kierunkiem wystrzału za graczem zielonym. W pewnym momencie zaczynają oni jednak swój charakterystyczny taniec, czyli kręcenie się w kółko bez jednoczesnego przejmowania punktu kontrolnego oraz towarzyszące temu strzelanie donikąd.

Ze względu na niewielki rozmiar sieci, podczas dalszych iteracji tego eksperymentu nie notowaliśmy praktycznie żadnej poprawy. Jak widać na filmie, gracz zielony był naprawdę blisko zakończenia gry z pozytywnym rezultatem, nie był jednak w stanie nauczyć się do tego dążyć. Po wykryciu długo ciągnącej się stagnacji w procesie nauczania wprowadziliśmy wiele zmian, które opisywaliśmy już dokładniej w poprzednich rozdziałach.

5.1.2 Gameplay_2

Ten materiał w zasadzie ilustruje dokładnie te same problemy co przedstawiony w poprzednim rozdziale. Również powstał on dość wcześnie.

Widzimy na nim, że boty często podejmowały dość dobre lokalnie decyzje. Widoczna jest walka o punkty za zdobywanie różnych punktów kontrolnych. Na razie jedyną metodą walki z przeciwnikiem były pociski. Przy tym eksperymencie funkcja nagradzająca strzały była wadliwa. Strzał był nagradzany w taki sposób, że w pierwszej klatce, w której pocisk się pojawił, był on przypisywany do gracza znajdującego się aktualnie bliżej, co nie zawsze było trafne. Dodatkowo, gracze byli nagradzani za chybione strzały przeciwników, w celu nauczania ich "unikania". Te dwie cechy przyniosły jednak niezamierzone skutki, ponieważ najbardziej opłacało się podejść do przeciwnika na bliski dystans, wytrzeleć w jego stronę i otrzymać dużą premię za uniknięcie jego strzału, gdyż pocisk lecący kompletnie w przeciwną stronę do naszego zawodnika był mu przypisywany.

Można jednak śmiało powiedzieć, że walki wyglądały coraz lepiej. Boty starały się podchodzić na bliski dystans, żeby zablokować nieodwracalne przejście strefy, oraz strzelały w różne strony próbując trafić swoich oponentów, tym samym wyrzucając ich z mapy. Ogromny problem nadal stanowił zbyt niewielki rozmiar sieci, tutaj jeszcze w połączeniu z absurdalną metodą nagradzania strzałów.

5.1.3 Gameplay _3

Jak już wspominaliśmy wcześniej w sekcji błędów przy implementacjach algorytmów, pewna funkcja oceniająca używana w dość długo działającym eksperymencie mocno faworyzowała genomy umiejętnie sterujące kolorem zielonym, dawała natomiast znacznie mniejszą szansę na ewoluowanie przełomowych strategii u graczy niebieskich. Na tym filmie widzimy skutek takiego trenowania.

Gracze nie rozgrywali wtedy swoich meczy symetrycznie, jeden gracz po prostu otrzymywał kolor niebieski, drugi zielony. Co więcej, zielony zawsze przypadał graczowi o wyższym wewnętrznym ID, co w przypadku tamtego eksperymentu było równoważne temu, że lepszy gracz zawsze otrzymywał kolor zielony. W ten sposób na szczyt tabeli o wiele łatwiej było się dostać nowym genomom, które niejako próbowały jeszcze lepiej działać kolorem zielonym niż takim, które próbowały alterować zachowanie gracza niebieskiego, a przez niesprawiedliwy system, były skazane na porażkę. Prawdopodobnie zaczęło się od kilku początkowych iteracji, w których wykształciła się leciutka przewaga zielonych, która następnie stopniowo wzrastała aż do momentu, w którym nie dało się już jej zatrzymać naturalnymi metodami.

Z pozytywów, gracz zielony posiada kilka obserwowalnych cech, które były pożądane. Przede wszystkim, bardzo szybko jest w stanie zmienić cel. Kiedy przejmuje strefę szybko odnajduje się w nowej sytuacji. Znajduje najszybszą drogę niejako omijającą drugiego gracza i pozostaje do końca rozgrywki w środkowym punkcie. Zaletą jest również to, że nie wychodzi w losowych momentach z przejmowanych stref, przez co jest w stanie je naprawdę szybko przejmować. Wydaje mi się, że jest to skutek o wiele lepszej funkcji ewaluującej, która nie dość, że promowała zwycięstwa same w sobie, to jeszcze w przypadku zakończenia rozgrywki przed czasem, zwycięzca otrzymywał punkty za przejęte przez siebie strefy tak, jakby rozgrywka toczyła się do końca.

Oczywiście, ogromną wadą tej populacji jest brak jakichkolwiek umiejętności radzenia sobie ze starciami, gdyż jej członkowie nie stawiali sobie nawzajem pod tym względem żadnego oporu. Większość zachowań było dla nich zupełnie nowych, było to szczególnie widać w testach człowiek przeciwko sieci. Kiedy używaliśmy umiejętności strzału, z której populacja zrezygnowała na rzecz częstego korzystania ze wzmocnienia, boty wchodziły na zupełnie nowe, niezbadane terytorium. Sieć de facto pracowała z parametrami odpowiedzialnymi za pocisk ustawionymi stale na zero, przez co wagi w tamtych obszarach mogły być zupełnie losowe, bo i tak nie miały żadnego wpływu na ocenę i przetrwanie wśród pozostałych zawodników turnieju. Doprowadziło to do sytuacji, w której możemy bardzo łatwo zdezorientować bota. Prawdopodobnie dobrym rozwiązaniem mogłoby być zestawienie ich wraz z populacją, która dużo częściej korzysta ze strzału. W takim wypadku jednak bardzo trudno byłoby dobrać funkcję ewaluującą tak, aby nie faworyzowała odgórnie jednej ze stron. Są to dwie zupełnie różne strategie, za które ciężko przyznawać punkty w taki sposób, żeby obydwie miały pole do rozwoju bez bycia pozostawionymi w cieniu przez drugą.

Dalsza optymalizacja tej populacji skupiała się na poprawie gry gracza zielonego. Wykorzystaliśmy ją jednak ponownie, tym razem ustawiając sprawiedliwą funkcję ewaluującą jako sędziego turnieju.

5.1.4 Gameplay_4

Tutaj już funkcja ewaluująca była w pełni sprawiedliwa dla obydwu stron. Napomnę, że punktem startowym dla populacji, której wycinek przedstawia materiał wideo, byli między innymi liderzy poprzedniego eksperymentu, w którym gracz zielony był faworyzowany. Jak widać, ta sytuacja, na szczęście, uległa zmianie.

Przede wszystkim widać, że gracz niebieski stał się o wiele bardziej aktywny na polu bitwy. Częściej korzystał ze swojej maksymalnej możliwej prędkości oraz z możliwości przyspieszenia, żeby jak najszybciej znajdować się w punktach kontrolnych. Co prawda widoczne jest to, że lata w pewien sposób losowo po punktach, niekoniecznie będąc w stanie trzymać się kurczowo jednego, co doprowadziłoby do jego zwycięstwa. Prawdopodobnie na tym etapie uczenia się stara się "pogodzić" obydwie cele przejmując na raz dwa punkty. Jest to raczej przejściowa kwestia, spodziewamy się, że ta cecha zostanie wyparta w przyszłości.

Pierwszy raz zaobserwowaliśmy również inną strategię bitewną od wykonywania strzałów z bliskiej odległości. Boty nauczyły się, że największe szanse na zdobycie punktu mają wtedy, gdy poruszają się szybko w jego stronę, oraz używają wzmocnienia. Strategią, jaką widzimy praktycznie co chwilę, stało się powolne wycofywanie się z punktu na niewielką odległość celem zmagazynowania energii, a po przekroczeniu minimalnego progu wparowanie do strefy z impetem, zdobywając przy tym w niej udziały i wyrzucając potencjalnego przeciwnika.

Końcowa faza rozgrywki pokazała jednak, że różnorodność genowa opisywanej populacji jest dość wąska. W zasadzie, była to jedna z niewielu dopuszczalnych przez nie strategii. Najbardziej podejrzewamy dwie potencjalne przyczyny. Po pierwsze, sytuacja, w której jedyną wolną strefą była lewa, a gra nie zakończyła się automatycznie, była na tym etapie dość niespotykana. Jak wspominaliśmy w analizie poprzedniego materiału, genomy były mocno nastawione na zwyciężanie jako gracz zielony, więc przejście dwóch stref zwykle wiązało się z natychmiastową wygraną. Po drugie, końcowe fazy rozgrywki również otrzymywały diametralnie inny input. Jeden z neuronów wejściowych przeznaczony był do przekazywania informacji o czasie, jego dynamicznie zmieniająca się wartość mogła więc stopniowo osłabiać boty, wprowadzając je do nowych, nieznanych wcześniej sytuacji. Ciekawym pomysłem na przyszłe prace nad projektem zdaje się być testowanie genomów na różnych ramach czasowych, na przykład sprawdzając, jak dużo punktów są w stanie zdobyć w grze trwającej 10 sekund. To nieco lepiej przygotowywałoby sieci na grę w końcówkach, ponieważ byłoby więcej okazji do poprawy zachowania, gdy neuron czasowy ma wysoką wartość. Nie pomagał również fakt, że populacja inicjalizacyjna tego eksperymentu była przystosowana jedynie do krótszych rozgrywek, obejmujących zazwyczaj około 30 sekund z dostępnych 180. (podajemy czas w sekundach, ponieważ wygodniej się na nich operuje ludziom w porównaniu do liczby klatek)

W roli podsumowania, był to jeden z pierwszych eksperymentów, w którym dało się zauważyć sensowne rezultaty. Ten eksperyment trwa do chwili, w której piszę ten raport, i co chwilę otrzymuję komunikaty, że nowy najlepszy genom został odnaleziony. Z tego powodu trudno jest opowiedzieć tu o najlepszych uzyskanych rezultatach. Wstawiłem więc film, który wraz z innymi sporządziłem podczas analizowania nowej populacji, i uznałem za dość ciekawy do opisanie rezultatów treningu.

5.1.5 Gameplay_5 i Gameplay_6

Cofnijmy się teraz trochę wstecz. Następne dwa materiały wideo prezentują rozwój pobocznego eksperymentu który prowadziliśmy, w którym testowaliśmy hipotezę o szumie informacyjnym i tym że za dużo danych wejściowych blokuje możliwość rozwoju modeli. W tym celu zredukowane zostały informacje dotyczące pocisków, ilości śmierci graczy i zliczające przejęte strefy (choć ich stan był nadal śledzony), czym zredukowaliśmy ilość neuronów do 23.

Kolejną ze zmian było wymienienie funkcji aktywacji na **tangens hiperboliczny**. Motywowane to było chęcią sprawdzenia, czy model będzie w stanie uczyć się mniej liniowych zależności i tym samym pozwoli wykształcić się bardziej skomplikowanym i precyzyjnym strategiom w oparciu o ograniczoną ilość informacji.

Na etapie **Gameplay_6** używałem również bardzo uproszczonej funkcji **fitness**, w której modele nagradzane były jedynie za zwycięstwo (1 punkt) oraz za przejęcie strefy (0.6 punkta). Zatem najlepszą formą zdobywania punktów było wygranie przez przejęcie obu stref, przy zaciętej walce przeciwnik również nie wychodził z niczym jeśli udało mu się przejąć inną ze stref.

Progress całego treningu został szybko zatrzymany, gdy w pewnym momencie eksperyment znalazł optymalną strategię gry i w każdej kolejnej populacji optymalny genom grał następująco jak na materiale. Gracze w dodatku byli niezmiernie wyczuleni na ilość energii przeciwnika więc ich ruchy były silnie od tego zależne. Dalszy trening po pierwsze - nie dawał żadnych rezultatów bo najlepszy model i tak był ten sam, po drugie - trening zajmował niezmiernie długo bo najlepsze modele przechodzące do kolejnych generacji parami ze sobą remisowały.

W związku z tym wyciągnąłem wnioski żeby może zmienić trochę metodę mutacji. Zamiast losować kompletnie losowy gen, będę mutować częściej ale mutacja będzie opierała się na losowym odchyłaniu danego genu (dodając liczbę z rozkładu **Uni**($-x, x$)). Ponadto uznałem żeby zmienić funkcję **fitness** na bardziej szczegółową, uwzględniającą między innymi odległość od niezajętej strefy, nagradzającą za trafne strzały itp (znamy tą funkcję np. z eksperymentu 4). Właśnie w ten sposób powstało nagranie **Gameplay_5**.

Mimo że nie było już sytuacji w której dokładnie ten sam model dominuje dziesiątki pokoleń, to jednak zbiór genów w populacji okazał się nie zmieniać na tyle często w kolejnych jej generacjach na ile powinien. Dlatego skoro w pewnym momencie wygrała strategia oparta o zajęcie lewego checkpointa, populacja została przejęta przez geny odpowiedzialne za tą akcję i siła mutacji nie była na tyle duża żeby zmienić tą sytuację.

Ostatecznie obydwa eksperymenty, jak i kilka innych związanych z redukcją danych wejściowych czy zmianą funkcji aktywacji, nie przyniosły żadnych ciekawych wyników. Gdy ograniczyliśmy dane wejściowe, algorytm i tak farowyzował te "mniej istotne". Pomysłem który próbowaliśmy zrealizować było wprowadzenie również redundantnych cech odległości między dwoma graczami, czy odległości gracza od środka planszy, gdyż one wydały się parametrem na który warto zwracać uwagę, jednak również i te eksperymenty nie przyniosły żadnych satysfakcjonujących rezultatów. Stąd wziął się wniosek że to nie problem w tym, że algorytm nie potrafi zinterpretować danych i zależności między nimi, tylko większy problem stanowi zachowanie różnorodności populacji czy potrzeba stawiania modelem w różnych sytuacjach.

5.1.6 Gameplay_7

Ten materiał ilustruje przykład mocno nieudanego eksperymentu. Był to eksperyment oparty o opisywany wcześniej algorytm wspinaczkowy. Dodatkowo, z dość niskim współczynnikiem określającym liczbę poprzednich czempionów koniecznych do pokonania.

Rezultatem eksperymentu była populacja o niezwykle niskiej różnorodności genetycznej. Ze względu na sposób, w jaki inicjalizowane były nowe genomy, wyształcenie się nowych strategii nie było możliwe. Żeby wejść do grupy czempionów, nowy zawodnik musiał bezpośrednio pokonać wybraną liczbę swoich poprzedników. Naturalne jest jednak, że gdyby chciał spróbować wprowadzić nową strategię do gry, to wiązałoby się to z początkową lekką utratą skuteczności. Przykładowo, nauka strzelania wymagałaby oddania kilku niecelnych strzałów, potencjalnie również zmian w pozycjonowaniu faworyzujących strzał zamiast pewnych punktów za przebywanie blisko nieprzejętej strefy. Niestety, forma naszego eksperymentu nie dopuszczała ani chwili słabszego zachowania ze strony bota, po nawet lekkiej przegranej z czempionem był on kasowany.

Widzimy, że topową strategią było więc podążanie w stronę przejmowanej strefy, po której przejściu jedyną opcją było już tylko jechanie w prostej linii, ale na tyle wolno, żeby nie wyjechać z mapy. W zasadzie iteracje algorytmu przypominały konkursy na największego lenia, ponieważ nowi czempioni starali się poruszać się coraz wolniej tak, aby zakończyć grę jak najbliżej środka, w szczególności nie wypaść z mapy, za co były bardzo surowo karane.

Podejrzewam, że dalszy proces trenowania wcale nie przyniósłby poprawy. Warto zaznaczyć, że film wideo przedstawia już dość mocno wytrenowaną generację, eksperyment ten trwał wyjątkowo długo jak na swój brak skuteczności. Prawdopodobnie główną wadą był opisywany wcześniej brak różnorodności. Duża część tego, co chciałbym tu napisać, została już bardziej szczegółowo opisana w sekcji o algorytmie wspinaczkowym, gdzie wyjaśniliśmy, dlaczego prawdopodobnie nie będzie on najlepszym wyborem dla naszego eksperymentu. Na pewno przekonaliśmy się, że ciekawym uczuciem jest ekscytacja możliwością zobaczenia świeżo wytrenowanego bota w akcji, po której następuje ujrzenie obrazu widocznego na materiale.