Projekt IO - Snake AI

Krystian Gronkowski, indeks: 281184

Grudzień 2024

Spis treści

1 Opis projektu **3**

2 Działanie programu **4**

3 Wyniki **6**

4 Możliwy rozwój **10**

5 Fragmenty kodu **11**

1 Opis projektu

Projekt polega na utworzeniu aplikacji uczenia maszynowego, która za pomocą modelu ewolucyjnego nauczy komputer grać w Snake'a. Program tworzy węże, które patrzą w każdym kierunku i za pomocą siatki neuronowej decydują o tym, w którym kierunku się poruszać.

Na początku wszystkie węże poruszają się losowo, ale wraz z każdą generacją doskonalą swój algorytm podejmowania decyzji za pomocą naturalnej selekcji aż w końcu stają się przyzwoitymi graczami w Snake'a.

Zwiększamy liczbę punktów zyskiwanych przez AI, z każdą generacją wybierając określoną liczbę węży które zdobyły największą liczbę punktów i rozmnażanie ich, oraz poprzez pozbywanie się z puli węży o mniejszej ilości zdobytych punktów.

Aplikacja posiada interfejs graficzny stworzony za pomocą modułu PyGame.

2 Działanie programu

config.py
Zawieta zmienne konfiguracyjne programu,takie jak: szansa na małą i duża mutacje, liczbe węży w generacji oraz kolory planszy.

main.py
Główny plik programu inicjalizujący GenerationManager'a i BoardRenderer'a

generationManager.py
Tworzy kolejne generacje węży, wywołuje funkcje ruchu dla każdego węża. Gdy wszystkie węże umrą, wybiera określoną w pliku config.py liczbę węży o najlepszym wyniku i tworzy listę węży będące ich potomstwem. Te węże potem biorą udział w kolejnej generacji.

Przy tworzeniu potomstwa istnieje szansa na małą (+- 10%) i dużą (+-500%) mutację, która zmienia wartość określonego neurona. Węże które przejdą pozytywne mutacje będą otrzymywać lepsze wyniki, dzięki czemu każda kolejna generacja powinna być coraz bardziej inteligentna.

brain.py
W tym pliku zawarty jest mózg węży - słownik w którym klucz odpowiada obiektowi który wąż widzi reprezentowanym jako znak. Słownik przechowuje liste neuronów (o wartości liczby zmiennoprzecinkowej) za pomocą których będzie obliczana wyjściowa punktacja dla każdego kierunku.
Tutaj znajduje się również funkcja do obliczania wyżej wymienionego wyjściowego wyniku dla każdego kierunku, która jest dogłębniej opisana w sekcji piątej - Fragmenty kodu.

W uczeniu maszynowym warto jest dodać dużą ilość zmiennych od których zależny jest output, dzięki czemu dajemy sztucznej inteligencji większe pole do popisu. Oprócz neuronów pasujących do każdego obiektu który może napotkać wąż znajdują się tam również dodatkowe modyfikatory, które są aktywowane niezależnie od tego co widzi wąż. Na początku dodałem zmienne "Stubborness"i "Widzimisie".

Podczas gry, zauważyłem że AI lubi chodzić po przekątnych. Nie jest to błędem, ale jest to bardzo nieludzkie zachowanie. "Stuborness"dodaje dodatkowe punkty kierunkowi w którym wąż poruszał się w poprzedniej turze i sprawia że rzadziej zmieniają swój kierunek. Dzięki temu, węże zaczęły poruszać się po przekątnych bardzo rzadko, i ich spos«ób poruszania się stał się bardziej satysfakcjonujący.

"Widzimisie"dodaje nieco losowości do sposobu poruszania się węży. Sprawia to, że ich sposób poruszania sie jest nieco bardziej interesujący, ale również pomaga wężom uwolnić się z cykli. Czasami węże mogą zacząć chodzić w kółko,

w tych sytuacjach dodatkowa siła pchająca ich w losowym kierunku może im pomóc i doprowadzić do zniszczenia cyklu.

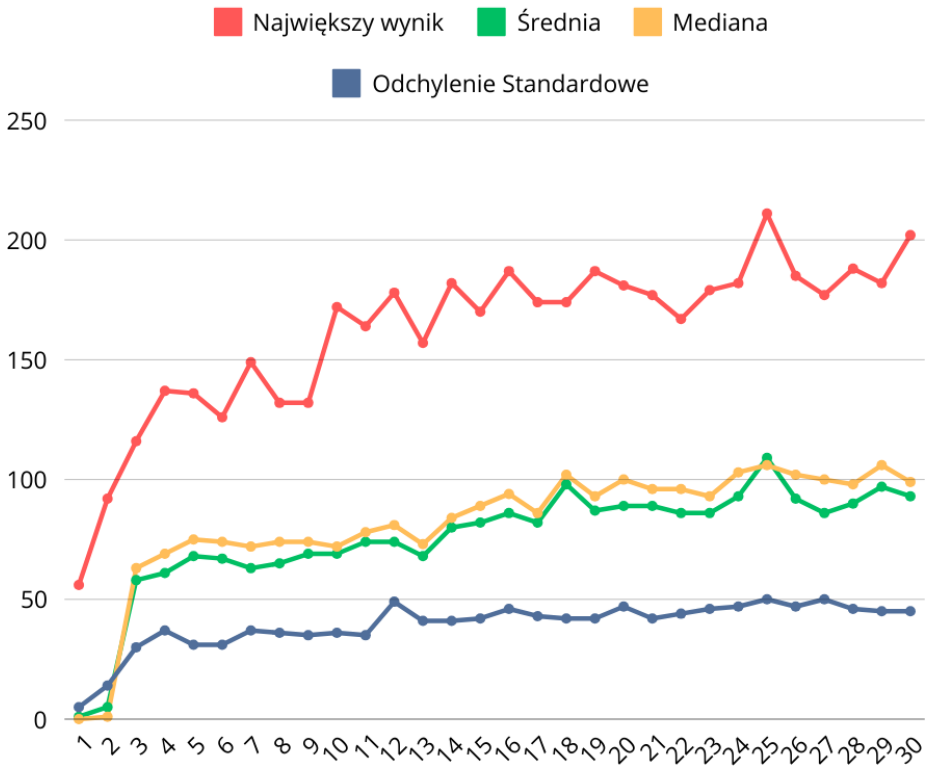
Wypróbowałem różne dodatkowe zmienne jak "Caution"który dodawał średnią wartość wszystkich outputów do wezła wyjściowego, dzięki temu wąż miał patrzeć nie tylko na najlepszy możliwy rezultat poruszania się w danym kierunku, ale miał również brać pod uwagę średnią wartość wszystkich rezultatów, pomnożonych przez "Caution". Okazało się, że inteligencja węży spadła drastycznie, i zdecydowałem się usunąć tą mechanikę.
Kolejnym niewypalonym pomysłem było "Craziness", tym bardziej głodny jest wąż, "Craziness"miał odejmować od najlepszych kierunków i dodawać do najgorszych. W ten sposób, jeżeli przez długi czas wąż nie zdobył żadnego punktu, to zaczynałby poruszać się zupełnie inaczej i "zmieniałby swoją taktykę", to również nie przyniosło jednak zamierzonych rezultatów.

Snake.py
Odpowiada za między innymi poruszanie się węży po planszy, kolizje, wydłużanie się ogona po kontakcie z jedzeniem. Szczególnie ważna jest funkcja look która zwraca wartość wyjściową dla kierunku. Jest ona opisana w sekcji piątej.

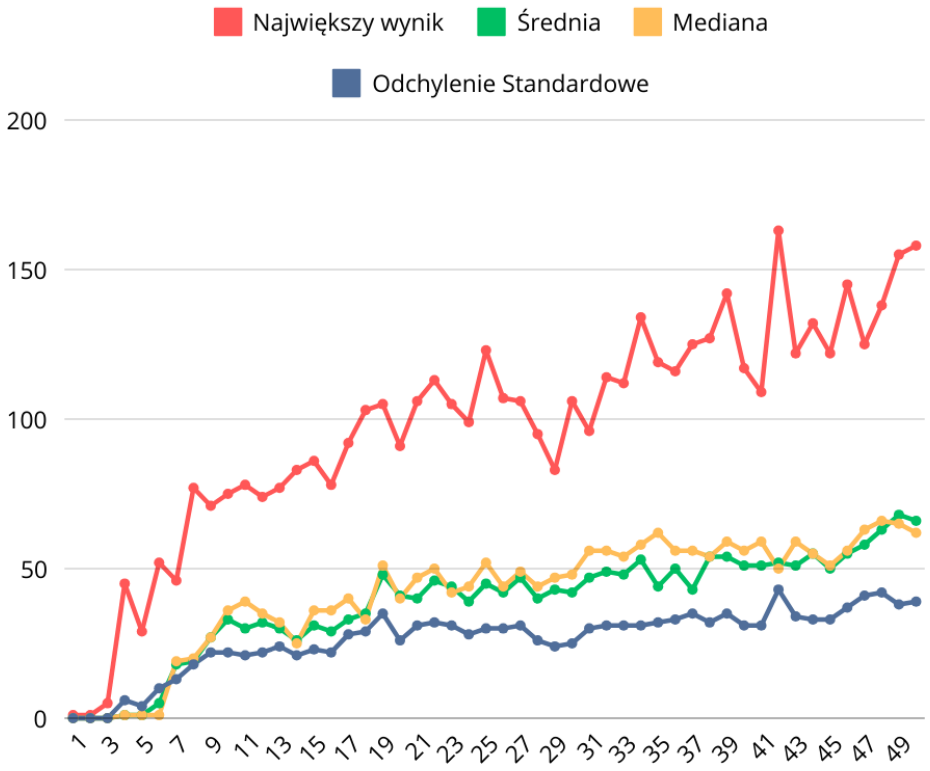
BoardManager.py
Wyświetla planszę na ekranie.

3 Wyniki

Poniżej rezultat treningu przez 30 generacji, przy 137 wężach z ukrytą wartwą sieci neuronowej o wielkości 9:



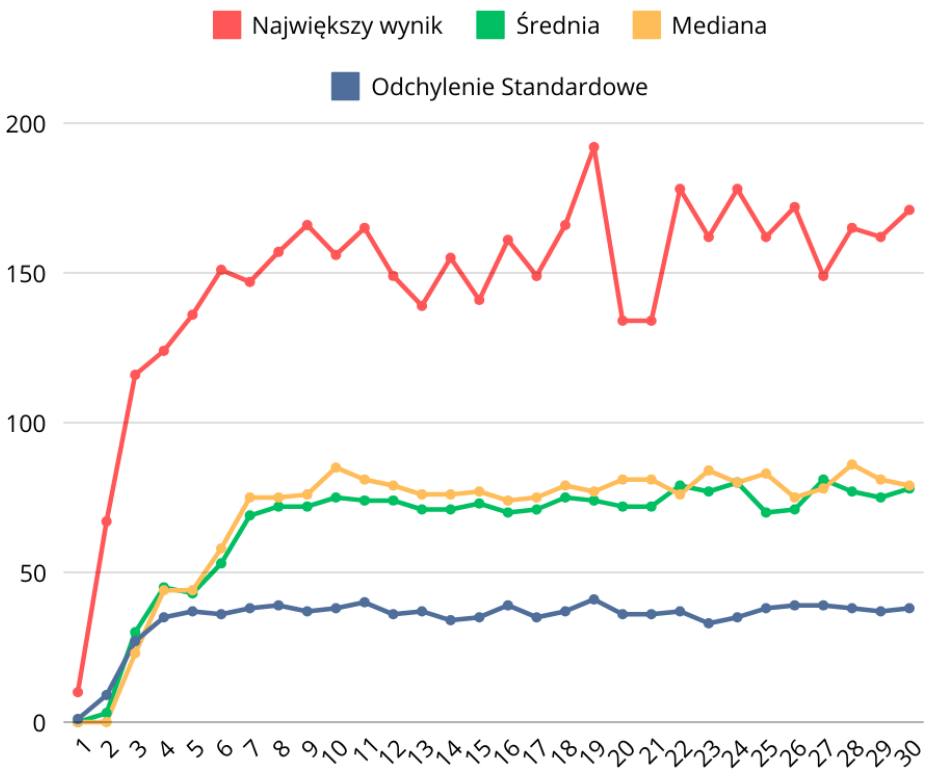
Poniżej rezultat treningu przez 50 generacji, przy użyciu jedynie 40 węży z ukrytą warstą sieci neuronowej o wielkości 9. Ten wykres przedstawia stopniowe polepszanie się wyniku w lepszy sposób, przez pierwsze 3 generacje największy wynik nie osiągnął nawet 5, a średnia i mediana były równe 0. W generacji 4 nastąpił gwałtowny wzrost uzyskanych wyników, a potem stopniowe polepszanie wyniku aż do generacji 50.



Zauważyłem że sposób w jaki program wzmacnia największy wynik może mieć negatywny wpływ na rezultat treningu. Jeżeli jest sytuacja w której tylko 6 z 120 węży z największym wynikiem stworzy potomstwo dla następnej generacji, może zaistnieć sytuacja w której wzmacniane są również ruchy prowadzące do "huśtania"wyników. Wąż który czasami zdobywa 50 punktów a czasami zdobywa 300 punktów ma większe szanse na rozmnożenie się niż wąż który zawsze osiąga stabilny wynik 200 punktów. To może ostatecznie ograniczać poziom który węże mogą osiągnąć - każda generacja będzie pełna węży które czasami stają na podium tylko dlatego że miały szczęście zamiast węży które są lepsze w większości przypadków.

Długo myślałem nad rozwiązaniem tego problemu - jedną możliwością było by otworzenie tej samej generacji kilkadziesiąt razy a potem dla każdego węża obliczyć jego średni wynik zamiast odpalania każdej generacji tylko raz. Jednakże jest to niesamowicie kosztowne - jeżeli zdeydujemy się odpalić każdą generacje 10 razy to oczywiście trening będzie trwał 10 razy dłużej.

Ostatecznie zdecydowałem się na stworzenie następującego eksperymentu - za każdym razem gdy generacja się skończy, sprawdzimy czy średni wynik wszystkich węży w tej generacji jest większy od poprzedniego rekordu. Jeżeli nie - zabijamy wszystkie węże w tej generacji i tworzymy nowe potomstwo dla generacji z największym średnim wynikiem, do skutku. W ten sposób zamiast dbać jedynie o największy wynik, zwiększamy najpierw średni wynik, a drugoplanowo zwiększamy największy wynik.



Rezultaty są widoczne na wykresie powyżej Ostatecznie skończyłem eksperyment po 30 generacjach widząc, że efekty nie są zadawalające. Te rozwiązanie było zbyt naiwne i ma oczywisty problem - istnieje szansa na to, że jednej generacji się poszczęści, i jedzenie poustawia się w wygodnych dla węży pozycjach w taki sposób, że średni wynik wzrośnie. W takiej sytuacji możemy zmarnować bardzo dużo czasu, na trenowanie kilkudziesięciu generacji tylko po to żeby je od razu zabić.

Nie jest to optymalne podejście, więc wycofałem tą zmiane i wróciłem do wzmacniania jedynie największego wyniku.

4 Możliwy rozwój

Projekt jest w stanie dosyć kompletnym, lecz możnabyłoby go rozbudować poprzez dodanie innych sposobów mutacji i generacji nowych węży, aby przyśpieszyć trening i upewnić się że przez przypadek nie wytrenujemy z węży cennych cech. Można również stworzyć system zapisu i wczytywania, co pozwoliłoby na trenowanie węży do tysięcy generacji bez potrzeby utrzymywania włączonego komputera przez cały czas.

Główny problem projektu to niski sufit umiejętności możliwych do nabycia przez AI. Powodem tego, jest mała ilość danych wejściowych. Wąż patrzy jedynie jeden krok do przodu, kiedy zobaczy jedzenie, zwykle będzie poruszać się w jego kierunku, nie myśląc o tym, co zrobi później. Z tego powodu często wchodzi w pułapki, z których nie ma wyjścia. Jest to problem, którego nie da się rozwiązać, nie ważne jak długo węże będą trenować, potrzeba fundamentalnej zmiany sposobu w jaki wąż patrzy na plansze.

Trzeba pamiętać jednak, że sam pomysł stworzenia algorytmu maszynowego służy bardziej jako ciekwawy projekt, a nie jest prawdziwym rozwiązaniem problemu zdobycia jak największej liczby punktów. Banalne byłoby ręczne utworzenie węża poruszającego się w cyklu hamiltona - wtedy zawsze uzyskiwalibyśmy maksymalną liczbę punktów. Ale nie o to chodzi.

Za każdym razem, gdy dodajemy dodatkowe mechaniki mające na celu ręczne poprawienie niedociągnięć algorytmu poruszania się będacego rezultatem uczenia maszynowego, tym bardziej oddalamy się od pierwotnego celu projektu. Myślę więc, że fakt, iż bardzo prosty algorytm uczenia maszynowego jest w stanie osiągnąć te - moim zdaniem - dość imponujące wyniki - wskazuje na sukces projektu.

5 Fragmenty kodu

Kilka istotnych fragmentów kodu

Tworzenie nowych generacji węży:

```
# Przygotuj następną generacje!

# Sortowanie od najlepszych węży
self.snakes = sorted(self.snakes, key=lambda snake : snake.points, reverse = True)

# Zostawiamy najlepsze węże
self.snakes = self.snakes[0:BEST_SAMPLES]
print("kept!")

# Najlepsze węże mają dzieci ze sobą, prowadzi do uśredniania wszystkich wartości
newSnakes = []
for repeat in range(REPEATS):
    for mother in range(BEST_SAMPLES):
        for father in range(BEST_SAMPLES):
            if mother != father:
                offspring = Snake()
                # Mózg potomka jest średnią wartością mózgów rodziców
                offspring.get_brain().mix_brains(self.snakes[mother].get_brain(),self.
                    snakes[father].get_brain())
                newSnakes.append(offspring)
for s in newSnakes:
    self.snakes.append(s)
# Dodajemy losowe węże
# Mogą wydawać się stratą mocy procesora, ale losowe węże dają nam pewność że nie
    pozbędziemy się żadnych cech z naszej puli. Cecha która zniknie może wrócić
    wraz z losowymi wężami
for s in range(NEW_SNAKES):
    sn = Snake()
    self.snakes.append(sn)
```

Funkcja look patrzy w danym kierunku dopóki nie zobaczy przeszkody, przekazuje rodzaj przeszkody i odległość do mózgu, który oblicza punktacje dla danego kierunku:

```
1 def look(self, start_x, start_y, x, y, add_dist):
2     dist = 1
3     looking_for = ["#","@","^"] # Przeszkody które zauważamy
4     while self.board[start_y+y][start_x+x] not in looking_for: # Dopóki nie
          napotkasz przeszkody, patrz dalej
5         start_x+=x
6         start_y+=y
7         dist+=1
8     return self.brain.get_output(self.board[start_y+y][start_x+x],dist+add_dist) #
          Napotkałeś przeszkode, przekaż do mózgu.
```

Funkcja get_output jest odpowiedzialna za przypisanie kierunkowi jego wartości, pobiera ona obiekt który wąż widzi gdy patrzy w tym kierunku oraz to, jak daleko ten obiekt od niego jest. Następnie wczytuje ona odpowiednią listę neuronów i dla każdego z nich, mnoży wartość wyjściową z wartością neurona, a następnie wykonuje funkcje aktywacji na wartości wyjściowej - tanH (hiperboliczny tangens). Na końcu do wartości wyjściowej jest dodawany neuron odpawiadający za bias.

```
1     # Przypisuje wartość do kierunku
2     def get_output(self, obj, distance):
3         out = 1
4         # Sumuje wartości wszystkich neuronów które są w warstwie odpowiadającej
              napotkanemu obiektowi
5         for node in range(1,HIDDEN_LAYER_SIZE):
6             out*=self.hidden_layer[obj][node]
7             math.tanh(out)
8         # Tym dalej obiekt, tym mniejszy wpływ ma na wynik końcowy
9         out+=self.hidden_layer[obj][0]
10        out/=distance
11        return out
```