Projekt Python - Snake AI

Krystian Gronkowski, indeks: 281184

Maj 2023

Spis treści

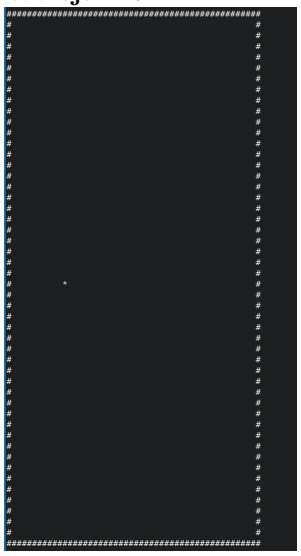
1 Opis projektu

Projekt polega na utworzeniu grę w Snake'a. Gra 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.
Użytkownik nie gra, a jedynie patrzy na węże które uczą się grać (i może bawić się zmiennymi dotyczącymi ich ewolucji).
Aplikacja posiada interfejs graficzny stworzony w module PyGame.

2 Raport z projektu

Poniżej opiszę każdą wersję gry która została wrzucona na githuba, wraz z opisem tego co zostało w danej wersji stworzone oraz jakie problemy zostały napotkane.

Wersja 1.0



Został utworzony szkielet projektu (plik main.py, snake.py oraz plik konfiguracyjny config.py). Główny plik main.py tworzył węża, a plik snake.py posiada funkcje inicjalizującą planszę gry oraz wyświetlającą ją w formie tekstowej w terminalu.
W tym wczesnym punkcie projektu jeszcze nie miałem żadnych poważnych problemów, ale istniały elementy które wymagały szybkiego wyszukania rozwiązania w internecie, takich jak na przykład w jaki sposób zadeklarować dwuwymiarową listę w pythonie[3], aby móc przechować w niej planszę gry.

Wersja 1.1

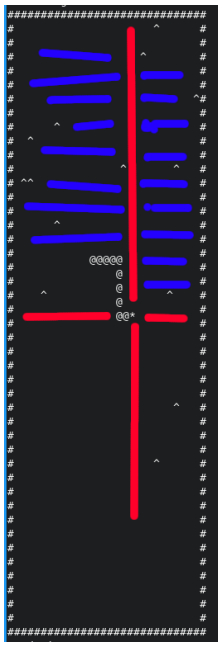
Wersja 1.1 dodała jedzenie rozmieszczane na mapie oraz ogon węża który wydłuża się wraz z każdym zdobytym punktem. Sposób implementacji ogona podążającego za graczem wymagał trochę pogłowienia się, ostatecznie wpadłem jednak na pomysł, aby każdy element ogona był zapisywany jako para współrzędnych i był przechowywany w liście, dzięki temu w każdej turze każdy element ogona mógł zmienić swoją pozycję na pozycję wcześniejszego elementu.

Wersja 1.2
Wersja 1.2 nareszcie dodała mózg węża - inicjuje on swoją ukrytą warstœwę węzłów, oraz posiada funkcję która zwraca punktacje dla każdego kierunku.

```
1 def get_output(self, obj, distance):
2     multiplayer=(max(BOARDSIZE_X,BOARDSIZE_Y)-distance)/max(
          BOARDSIZE_X,BOARDSIZE_Y)
3     out = 1.0
4     for node in range(HIDDEN_LAYER_SIZE):
5         out*=self.hidden_layer[obj][node]
6     return out
```

Biorąc na wejściu kierunek, przeszkodę jaka znajduje się w tym kierunku oraz odległość przeszkody od węzła. Mnoży on wartość wyjściową przez odpowiednie węzły w ukrytej warstwie a potem dzieli wynik przez odległość od przeszkody. Dzięki temu wąż może się poruszać, w tej wersji jeszcze zupełnie losowo.

Wersja 1.3



Wersja 1.3 zmieniła sposób w jaki węże patrzą na planszę. Wcześniej widziały one tylko 4 proste linie (czerwone linie), teraz widzą niemalże całą planszę (czerwone + niebieskie dla każdej czerwonej lini), za każdym razem gry rozglądają się w danym kierunku patrzą również w każdym innym kierunku i algorytm przypisuje danemu kierunkowi największą wartość ze wszystkich zobaczonych możliwości.
Teraz plik main.py tworzy węże, i gdy wszystkie umrą, zabija wszystkie węże oprócz kilku tych, które uzyskały najlepsze wyniki. Potem rozmnaża je ze sobą i rozpoczyna następną generacje.
Mimo wszysko, najlepszy wąż osiąga tylko 3 punkty i wszystkie węże wydają się chodzić zupełnie bez sensu.

Problem: węże mogą wpaść w cykl i chodzić w kółko, przez co nigdy nie przegrywają gry poprzez wejście w ścianę lub w swój własny ogon, więc nigdy nie możemy rozpocząć kolejnej generacji. Rozwiązanie - wprowadzenie głodu. Maksymalny głód jest definiowany w config.py, jeżeli wąż przez długi czas nie uzyska dodatkowego punktu, i jego głód przekroczy limit - to przegra, nawet jeżeli nie zginął.

Wersja 1.4

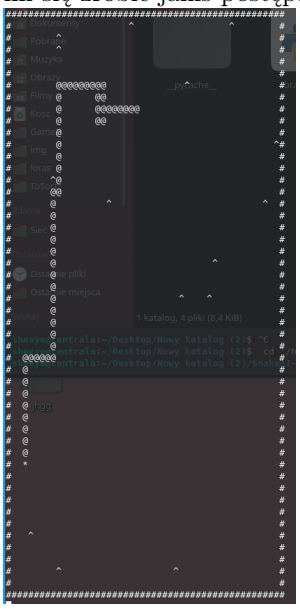
Problem wersji 1.3 został nareszcie rozwiązany. Po długim poszukiwaniu błędu okazało się, że za wszystkim stała zamiana wartości x i y:

```
# przed zmianą
    right_out = self.ultra_look(0,1)
    left_out = self.ultra_look(0,−1)
    down_out = self.ultra_look(−1,0)
    up_out = self.ultra_look(1,0)

 # po zmianie
  right_out = self.ultra_look(1,0)
    left_out = self.ultra_look(−1,0)
    down_out = self.ultra_look(0,−1)
    up_out = self.ultra_look(0,1)
```

Ten błąd oznaczał, że gdy wąż uważał, że iście w prawo jest najlepszym wyborem, to szedł do góry, a gdy uważał że iście do góry jest najlepszą decyzją, to szedł w prawo. Taki sam problem występował w innych kierunkach. Ten giga-

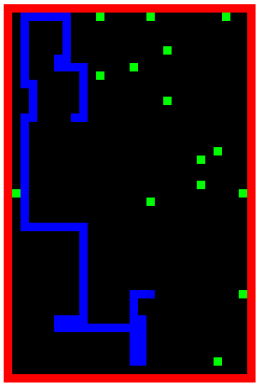
nyczny błąd wyjaśniał dziwne zachowanie węży. Po poprawie tych linijek kodu, wyniki natychmiastowo podskoczyły z 3 punktów do ponad 30. W końcu udało mi się zrobić jakiś postęp.



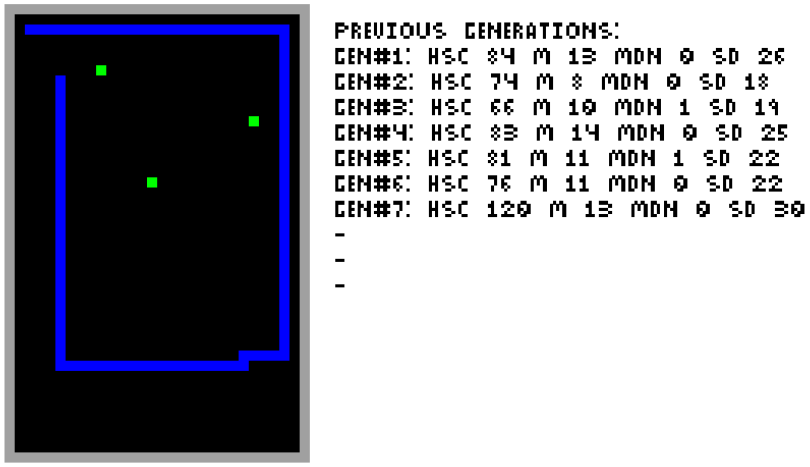
Wersja 1.5

Teraz, gdy miałem już działającą symulacje, nadszedł czas na danie aplikacji interfejs graficzny. Był do tego bardzo przydatny poradnik do tworzenia gry platformowej PyGame[4]. Mimo, że moja gra nie ma nic wspólnego z platformówką, to sposoby tworzenia okna gry i renderowania grafiki na ekranie są takie same.

Dodanie interfejsu PyGame wymagało drobnej refaktoryzacji kodu. Wcześniejsza zawartość pliku main.py trafiła do osobnej klasy GenerationManager, a teraz main.py zajmuje się tworzeniem okna gry PyGame.



Wersja 1.6
W wersji 1.6 zająłem się wyświetlaniem istotnych informacji na temat każdej generacji na ekranie. Zdecydowałem się wyświetlać maksymalną liczbę punktów uzyskanych przez jakiegoś węża, średnią liczbę punktów, medianę oraz odchylenie standardowe, aby zobaczyć jak rozsiane są wyniki.



Wersja 1.7
W uczeniu maszynowym warto jest dodać większą ilość zmiennych od których zależny jest output, dzięki czemu dajemy sztucznej inteligencji większe pole do popisu. Miałem problem z wymyśleniem nowych wejściowych danych od których zależny byłby wynik, ale dodałem nowe 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 żadziej zmieniają swój kierunek. Dzięki temu, węże teraz poruszają się po przekątnych bardzo rzadko, i ich spos«ób poruszania się jest bardziej satysfak-

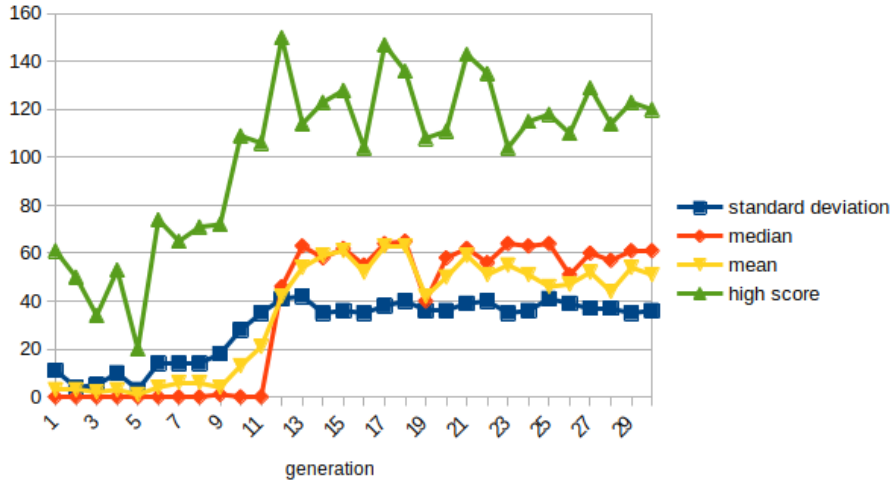
cjonujący.

"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.

3 Wnioski

Poniżej przedstawiam wykres zawierający największy wynik, średni wynik, mediane oraz odchylenie standardowe dla 30 genracji:



Można zauważyć, że największy wynik zwiększa się z zakresu 20-70 do zakresu 100-140. Oznacza to, że sztuczna inteligencja rzeczywiście uczy się grać. Dalsza symulacja do 50+ generacji nie wydaje się już zwiększać punktacji powyżej tej granicy. 120-170 punktów to sufit symulacji.

Mediana wynosi 0 przez 11 początkowych generacji, aż nagle gwałtownie się zwiększa. Po przekroczeniu średniej, mediana zwykle nigdy już nie opada poniżej średniej punktacji.

4 Możliwy rozwój

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, co w chwili wykonywania tego projektu jest poza moimi umiejętnościami.

Oprócz tego, możnabyłoby stworzyć inne sposoby 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.

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:
```

```
21      self.snakes.append(s)
22  # Dodajemy losowe węże
23  # 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
24  for s in range(NEW_SNAKES):
25      sn = Snake()
26      self.snakes.append(sn)
```

Funkcja wyświetlania planszy, bierze plansze w formacie tekstowym, i zmienia kolor każdej płytki aby odpowiadała wejściowej planszy. Płytki są tworzone na starcie aplikacji i są przetrzymywane w dwuwymiarowej liście.

```
1  # Zmienia kolor każdej płytki, aby pasował do danej planszy tekstowej
2  def render(self, text_board, alive):
3      for y in range(len(self.board)):
4          for x in range(len(self.board[0])):
5              if text_board[y][x]=="#": # Ściana
6                  self.board[y][x].change_color(WALL_COLOR)
7              if text_board[y][x]==" ":
8                  self.board[y][x].change_color(EMPTY_COLOR)
9              if text_board[y][x]=="^": # Jedzenie
10                 self.board[y][x].change_color(FOOD_COLOR)
11             if text_board[y][x] in ["@","*"]: # Ogon
12                 if alive:
13                     self.board[y][x].change_color(TAIL_COLOR)
14                 else:
15                     self.board[y][x].change_color(DEAD_COLOR)
```

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.
```

Literatura

[1] Dokumentacja modułu Pygame, https://www.pygame.org/docs/

[2] Dokumentacja języka Python, https://docs.python.org/pl/3/

[3] Poradniki z języka Python i modułu Pygame
https://www.geeksforgeeks.org

[4] Poradniki PyGame
https://coderslegacy.com/python/pygame-platformer-game-development/

[5] Odpowiedź na pytanie z PyGame
https://stackoverflow.com/questions/20842801/
how-to-display-text-in-pygame