**POP**

**Przeszukiwanie i optymalizacja**

**Dokumentacja końcowa projektu**

**Członkowie zespołu**:

- Jakub Ptasznik, 304 115

- Patryk Bandyra, 304 005

**Temat projektu**:

#13 – lądowanie spodkiem z użyciem algorytmu heurystycznego w środowisku OpenAI

1. **Opis problemu**

Dysponując 4 akcjami (nic nierobienie, odpalenie głównego silnika, odpalenie lewego silnika, odpalenie prawego silnika) wylądować spodkiem pomiędzy chorągiewkami. Środowisko dostarcza informacji o stanie, w którym znajduje się spodek:

- współrzędna pozioma

- współrzędna pionowa

- szybkość pozioma

- szybkość pionowa

- kąt

- szybkość kątowa

- wartość boolowska informująca czy lewa noga lądownika dotyka powierzchni

- wartość boolowska informująca czy prawa noga lądownika dotyka powierzchni

Środowisko dostarcza również punktową nagrodę w zależności od podejmowanych akcji oraz aktualnego stanu. Należy maksymalizować otrzymywaną nagrodę.

Symulacja może składać się z wielu rozgrywek. Ukształtowanie terenu pomiędzy kolejnymi rozgrywkami może ulegać zmianie (w zależności czy generator będzie resetowany tym samym ziarnem czy też nie). Zdecydowaliśmy się uczyć i testować nasze rozwiązanie w zmiennym środowisku, ale takim które zmienia się zawsze w ten sam sposób (stałe ziarno generatora).

Każda rozgrywka kończy się, gdy spodek znajdzie się w stanie terminalnym (wyląduje pomiędzy dwoma chorągiewkami) lub upłynie narzucona z góry liczba iteracji jednej rozgrywki.

1. **Opis sposobu rozwiązania problemu**

Zdecydowaliśmy się zastosować algorytm uczenia ze wzmocnieniem – Q-learning.

Dokonaliśmy takiego wyboru, ponieważ problem, który chcemy rozwiązać da się łatwo przełożyć na model Q-learning – posiadamy system wejścia oraz wyjścia, nagrody, środowisko. Dodatkowo przestrzeń stanów jest ciągła (np. ciągłe wartości prędkości czy pozycji), ale istnieje możliwość jej ograniczania poprzez zastosowania pewnych przedziałów (np. możliwe pozycje poziome spodka (współrzędna x) ograniczyć do 100 przedziałów). Liczba akcji dostępnych do wykonania przez spodek jest ograniczona (wynosi 4).

Sposób rozwiązania problemu:

- mamy agenta, który porusza się w pewnym środowisku

- środowisko nagradza lub karze agenta za podejmowane akcje

- agent uczy się podejmować akcje, które prowadzą do maksymalizacji otrzymanych nagród

- agent musi jednocześnie eksploatować oraz eksplorować dane środowisko

- nauka agenta odbywa się poprzez aktualizację wartości Q-table

Q-table przechowuje Q-values. Q-value wskazuje na jakość pewnej akcji, która może zostać podjęta w danym stanie: Q(s, a). Q-values są oszacowaniami sumy przyszłych nagród.

Wyuczona Q-table zawiera wartości, które pozwalają agentowi na podjęcie najlepszej akcji w danym stanie. Q-table reprezentuje politykę zachowania agenta w danym środowisku.

**Różnice w realizacji w stosunku do dokumentacji wstępnej i planowanego sposobu implementacji:**

Początkowo planowaliśmy zrealizować Q-table jako zwykłą tablicę wielowymiarową przechowującą wartości (Q-values). Środowisko zostałoby poddane dyskretyzacji (np. prędkość X spodka mogłaby przyjąć 4 różne wartości). Przykładowy planowany wygląd Q-table:

* Zakładając podział dziedzin otrzymywanych informacji o stanie spodka na pewne przedziały:

|  |  |
| --- | --- |
| Informacja o stanie | Liczba przedziałów |
| Współrzędna X | 100 |
| Współrzędna Y | 100 |
| Szybkość pozioma | 4 |
| Szybkość pionowa | 4 |
| Kąt | 4 |
| Szybkość kątowa | 4 |
| Czy lewa noga lądownika dotyka powierzchni | 2 |
| Czy prawa noga lądownika dotyka powierzchni | 2 |

Niestety taki sposób implementacji okazał się bardzo trudny do zrealizowania ze względu na następujące czynniki:

* Brak możliwości dokładnego określenia maksymalnych przedziałów wartości niektórych obserwacji. Taka informacja nie została podana na oficjalnej stronie OpenAi jak i na oficjalnym GitHubie. Poszukiwania odpowiedzi w Internecie okazały się bezskuteczne. Również ręczne próby uzyskania maksymalnych wartości różnych obserwacji ze środowiska nie przyniosły rezultatów. Dodatkowo okazało się, że niektóre wartości takie jak szybkość, szybkość kątowa czy kąt ułożenia spodka nie mają ograniczeń (tzn. przykładowo spodek obracający się cały czas w jednym kierunku cały czas zwiększa wartość kąta).
* Problem z dokonaniem dyskretyzacji (w których miejscach dany przedział powinien się zaczynać, a w którym kończyć).
* Nieefektywność pamięciowa. Reprezentacja Q-table w planowany przez nas sposób zajmowałaby ogromną ilość pamięci (dla przykładowej reprezentacji pokazanej w tabeli powyżej, ilość użytej pamięci wyniosłaby ok 165 MB; przy czym przykładowa tabela reprezentuje środowisko w bardzo ograniczony sposób; lepsze odwzorowanie wartości otrzymywanych ze środowiska oznaczałoby zużycie pamięci idące w GB, co jest niedopuszczalne i nieefektywne ze względu na czas potrzebny na naukę tak dużej tabeli)

Rozważając dostępne opcje, zdecydowaliśmy się zaimplementować Q-table w postaci sieci neuronowej (perceptronu wielowarstwowego). Dzięki temu nasze rozwiązanie nie boryka się z wyżej wymienionymi problemami, ponieważ sieć sama uczy się dyskretyzacji środowiska i zakresu wartości, a zużycie pamięci jest mniejsze. Mamy również możliwość eksperymentowania z doborem modelu sieci w celu polepszenia zachowania agenta.

Sieć odpowiada dokładnie zachowaniu Q-table, gdyż przyjmując na wejściu stan agenta, zwraca zalecaną akcję, która powinna zostać podjęta w celu maksymalizacji otrzymywanych nagród.

- Q-table jest uczone zgodnie z poniższymi wzorami:

gdzie:

- – chwilowa różnica dla akcji podjętej w poprzednim stanie

- – nagroda otrzymana za akcję podjętą w poprzednim stanie

- – dyskonto (0-1); hiper parametr

- – największa wartość Q-value dostępna dla dowolnej akcji w danym stanie

- – Q-value akcji podjętej w poprzednim stanie

gdzie:

- – nowa wartość Q-value dla akcji podjętej w poprzednim stanie

- – stara wartość Q-value dla akcji podjętej w poprzednim stanie

- – współczynnik nauki; hiper parametr

- – chwilowa różnica dla akcji podjętej w poprzednim stanie

Jako, że zdecydowaliśmy się implementować Q-table, jako perceptron wielowarstwowy, to drugi wzór jest w pełni realizowany przez sieć neuronową.

1. **Eksperymenty**
2. **Porównanie średniego wyniku otrzymywanego przez różne rozwiązania**

Liczba rozgrywek = 1000

Generator zasilany tym samym ziarnem

Podejście losowe – agent podejmuje losowe akcje:

Średni wynik = -180,4

Podejście „hard coded” – agent podejmuje akcje zgodnie z określonymi optymalnymi przedziałami wartości różnych obserwacji (np. jeśli kąt jest pomiędzy X a Y, to wykonaj A):

Średni wynik = 230,57

Nasze rozwiązanie:

Średni wynik = 215,13

Po przeanalizowaniu wyników działania naszego agenta udało się nam zdiagnozować sytuacje, które wpływają na znaczące obniżenie średniego wyniku. Pierwsza sytuacja to taka, w której statek ześlizguje się po pochyłym podłożu wcześniej wylądowawszy w wyznaczonym miejscu - statek wciąż się porusza, więc stan terminalny nie jest osiągany (statek otrzymuje kary). Druga sytuacja polega na lądowaniu na granicy wyznaczonego lądowiska – jedna z nóg lądownika znajduje się poza obszarem (nie jest osiągany stan terminalny; statek otrzymuje kary).

Rozwiązaniem powyższych problemów mógłby okazać się dobór lepszej architektury modelu lub ingerencja w proces uczenia się agenta przez określenie pewnych twardych warunków.

Nie udało się nam dobrać lepszej architektury, a ingerencja w proces uczenia nie przyniósł zadawalających rezultatów (otrzymane wyniki były gorsze od dotychczasowych).

1. **Modyfikacje naszego modelu**

Model nasz jest perceptronem wielowarstwowym o następującej strukturze:

Model(

(fc1): Linear(in\_features=8, out\_features=512, bias=True)

(fc2): Linear(in\_features=512, out\_features=512, bias=True)

(fc3): Linear(in\_features=512, out\_features=512, bias=True)

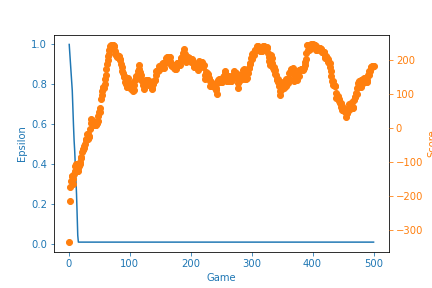
(fc4): Linear(in\_features=512, out\_features=256, bias=True)

(fc5): Linear(in\_features=256, out\_features=4, bias=True)

(loss): MSELoss())

Z współczynnikiem uczenia równym 0.002 i współczynnikiem gamma równym 0.99.

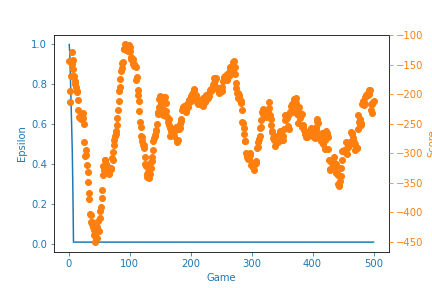
Proces uczenia dla modelu:



Dropout

Dołożenie maskowania losowych neuronów w celu uniknięcia przeuczenia modelu skutkowało obniżeniem jego skuteczności. Model nie był w stanie się nauczyć sterowania statkiem, przy ograniczonej liczbie neuronów. Najniższa testowana wartość dla Dropout wynosiła 0.2.

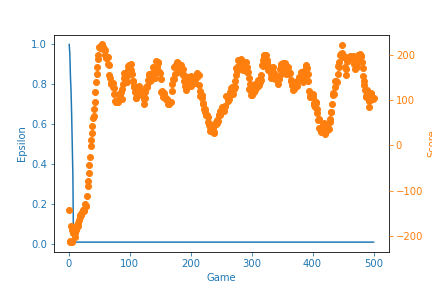
Proces uczenia dla modelu z dropout:



Zmniejszenie wymiarów warstw

Ograniczenie liczby neuronów w warstwach, skutkowało podobnymi obserwacjami jak przy Dropoucie. Model nie mógł się nauczyć skutecznie prowadzić statek przy mniejszej liczbie neuronów. W tym przypadku wyniki są gorsze od niezmodyfikowanego rozwiązania, ale nie są aż tak złe jak przy modelu z Dropoutem.

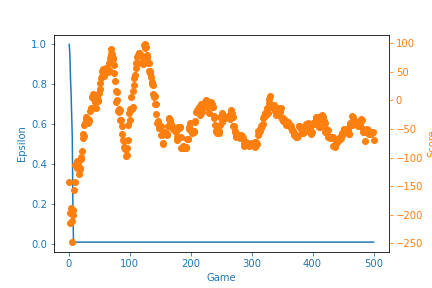
Proces uczenia dla modelu o mniejszej wymiarowości:



Zmiana funkcji straty

Przy zamianie funkcji straty z MSELoss() na L1Loss() model znacznie obniżył swój wynik. Obliczanie błędów jako błąd średniokwadratowy jest przy tym zadaniu skuteczniejsze niż obliczanie błędu bezwzględnego.

Proces uczenia modelu z funkcją straty L1Loss():

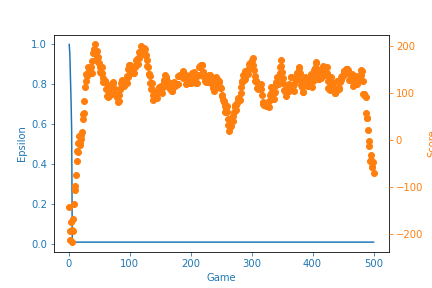


1. **Modyfikacja hiper parametrów agenta**

Agent ma współczynnik uczenia równy 0.001 i był uczony przez 500 epizodów.

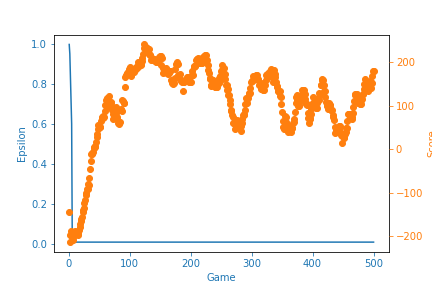
Po zwiększeniu współczynnika uczenia do wartości 0.002 proces uczenia się jest o wiele bardziej poszarpany. Różnice w wynikach z epizodu na epizod są znacznie większe, tym samym błędy są znacznie większe.

Proces uczenia dla zwiększonego współczynnika uczenia:



Pójście w drugą stronę czyli zmniejszenie współczynnika uczenia do wartości 0.0004 także przyniosło pogorszenie wyników. Model przez mniejszy współczynnik łatwiej „utykał” w minimach lokalnych i nie był już w stanie osiągnąć tak samo dobrych wyników.

Proces uczenia dla zmniejszonego współczynnika uczenia:



Symulacje dla podejścia losowego, „hard-coded” oraz wszystkich wytrenowanych modeli można włączyć za pomocą pliku *tests.py*.

1. **Technologia, w której realizowano projekt**

Projekt zrealizowano z użyciem języka Python w środowisku Box2D, będącego częścią Gym OpenAI. Platformą bazową został Windows 10, ale kod jest możliwy do uruchomienia na dowolnej platformie.

Korzystanie z wyżej wymienionego środowiska Gym jest utrudnione na platformie Windows, ale nie niemożliwe. W celu poprawnego działania środowiska na Windows 10 konieczne jest:

* zainstalowanie Microsoft Visual C++ Build Tools
* stworzenie wirtualnego środowiska z użyciem Anacondy oraz zainstalowanie następujących pakietów:

- z użyciem pip: gym, Box2D

- z użyciem Condy: swig

* zainstalowanie Xming oraz nieprzerwane działanie tego programu w trakcie działania środowiska