Wstęp do sztucznej inteligencji

Lab_06 – Q-Learning

Bartosz Latosek

310 790

Wstęp

Celem ćwiczenia jest implementacja algorytmu Q-learning pozwalającego agentowi na znajdywanie najkrótszej ścieżki do zadanego celu w labiryncie.

Użyte biblioteki:

> Pygame

> Argparse

> MatPlotLib

1.1 Opis środowiska i agenta

Agentem w zadanym ćwiczeniu jest "Q-Uber" poruszający się analogicznie do szachowej wieży. Na planszy rozmieszczone są przeszkody, których napotkanie przez agenta jest karane ujemnymi punktami. Na planszy znajduje się też jedno pole z dodatnią wartością - cel agenta.

Stan agenta to jego położenie na planszy. Jest określany przez numer rzędu i kolumny indeksując od 0. **Akcją** jest decyzja, w którą stronę pójść z danego stanu (góra, dół, lewo, prawo). **Nagrodą** jest stan, w którym agent znajduje się w celu. **Polityka** określana jest prostą zasadą - agent powinien jak najszybciej dotrzeć do celu, nie wbiegając przy tym w pola z ujemną wartością.

Wartości pól planszy to odpowiednio –1 dla pustej ścieżki, którą agent może się przemieszczać, -100 dla przeszkody na planszy i 100 dla pola oznaczającego cel.

1.2 Opis implementacji

Wizualna część programu znajduje się w pliku main.py. Klasa *Board* jest odpowiedzialna za generowanie planszy o podanych parametrach oraz jej wizualizację za pomocą biblioteki *pygame*. Zbiera też ona interaktywnie dane wejściowe od gracza i kontroluje działanie algorytmu Q-learning.

Sam algorytm Q-learning znajduje się w pliku Q_Learning.py a jego działanie ilustruje poniższy pseudokod.

Dane: *episodes, board, discount_factor, learning_rate, epsilon*

start:
q_table = initializeQTable()
for *e* **in** *episodes:*
  current_pos *= getStartingLocation()*
  ***w*hile !** *IsTerminalState(**current_pos**):*
    action *= getNextAction(**current_pos, epsilon**)*
    prev_pos *=* ***current_pos***
    current_pos *= getNextLocation(**current_pos, action**)*

***reward** = board(**current_pos**)*
old_q_val = q_table(prev_pos, action)
temporal_difference** = **reward** + (**discount_factor * max(q_table(current_pos))**) - **old_q_val

q_values(prev_pos, action) += learning_rate * temporal_differnce

episodes – liczba epok, przez które trenowany będzie algorytm

board – plansza z wagami odpowiadającymi wartościom stanów

discount_factor – parametr określający wagę korzyści przyszłościowych po podjęciu danej akcji

learning_rate – parametr określający szybkość uczenia się algorytmu

epsilon – parametr określający prawdopodobieństwo wybrania losowej akcji w danym stanie (eksploracja)

1.3 Przykład działania

Board-size – 25

Starting_pos - (0, 0)

Goal - (24, 24)

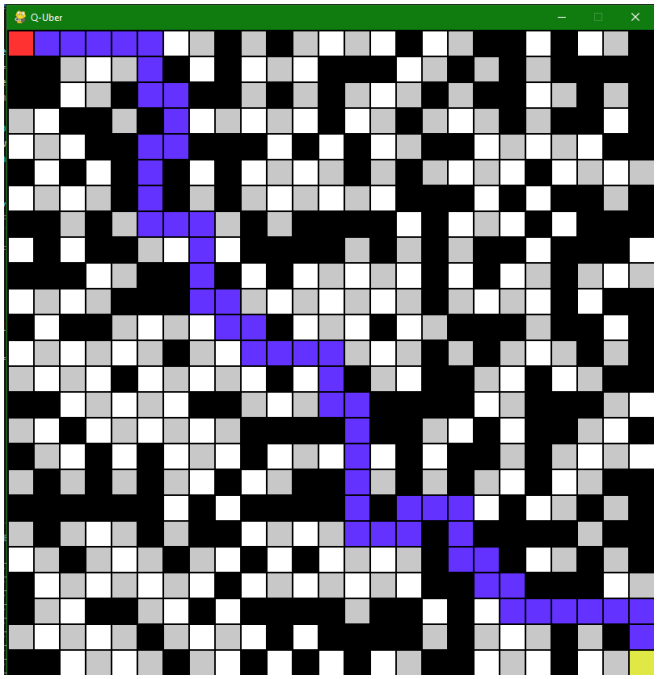
Obstacles – 250

Episodes – 10 000

Discount_factor – 0.9

Learning_rate – 0.5

Epsilon – 0.9



Rys. 1 Przykład działania algorytmu

Analiza działania

Analiza działania algorytmu zostanie przeprowadzona na podstawie obserwowania zmian zachowania się agenta w zależności od parametrów algorytmu Q-learning. Nie decydowałem się na analizę zmian parametrów generowania planszy, gdyż są efekty są proste do przewidzenia. Wraz ze zwiększeniem rozmiaru planszy, algorytm będzie potrzebować więcej epok w celu poprawnego wytrenowania, a ze względu na fakt, że w każdym epizodzie trenowania, pozycja startowa agenta wybierana jest losowo, rozmieszczenie pola startowego i celu na planszy nie ma najmniejszego znaczenia. Pozycja startowa jest odpowiedzialna jedynie za poprawne generowanie labiryntu (aby był on poprawny musi istnieć ścieżka pomiędzy nim a celem)

W związku z powyższym, analizę działania algorytmu można podzielić na 4 aspekty.

Analiza zostanie wykonana przy rozmiarze tablicy 25 x 25 i punkcie startowym i końcowym odpowiednio w lewym górnym i prawym dolnym rogu planszy. Jako miarodajność jakości algorytmu posłuży średnia liczba kroków z losowo wybranego punktu planszy do celu w 500 losowaniach. (Istnieje droga bezpośrednio z wylosowanego pola do celu)

I. Episodes

(discount_factor = 0.9, learning_rate = 0.5, epsilon = 0.9)

n = 100

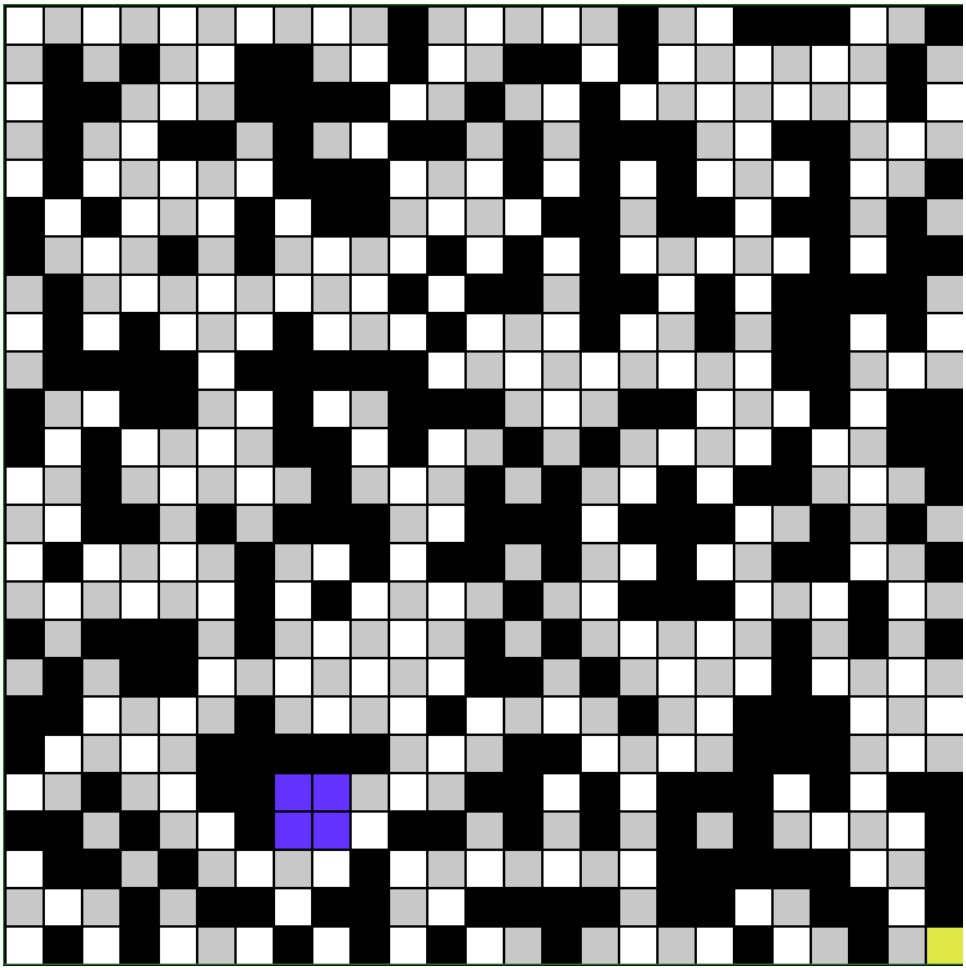
Random Car	Q-uber
4002.332	**317,726**

Tabela 1. Porównanie RandomCar i Q-uber na podstawie episodes

==========

Uwaga

W tym miejscu należałoby opisać zauważone zjawisko. Trenowanie Q-learning'u następuje poprzez losowe wybranie miejsca startowego a następnie eksplorację środowiska dzięki dotychczasowo stworzonej polityce oraz elementom losowym. W związku z czym, przy rozmiarze planszy 25 x 25 mamy 625 różnych dostępnych pól startowych. Przy założeniu, że przy 100 losowaniach pozycji startowej co najmniej 30% z nich w 2. ruchu wyląduje na polu z ujemną wartością (co od razu zakończy uczenie epoki). Pozostaje nam ok. 70 przejść (kończących się po średnio 5 ruchach). Jest to za mało, żeby jakkolwiek sensownie wytrenować agenta przez co w większości przypadków najkrótsza znaleziona przez niego trasa wygląda tak:



Rys 2. Problem w trenowaniu przy małej ilości epok

Agent po prostu zaczyna kręcić się w kółko aż jego trasa nie osiągnie z góry ustalonej długości, która przerwie działanie algorytmu.

Na podstawie powyższej uwagi, dalsza analiza działania przy zmianach parametru Epsilon traci sens, gdyż agent będzie niedouczony albo wytrenowany optymalnie, przy czym dalsze trenowanie go w 2. przypadku nie przynosi większych korzyści.

Porównanie dla liczby epok = 10 000

Random Car	Q-uber
3929.276	**16.282**

Tabela 2. Porównanie RandomCar i Q-uber na podstawie discount_factor

Jak widać na przykładzie, różnica w działaniu jest znacząca.

II. Discount Factor

(Episodes = 10 000, learning_rate = 0.5, epsilon = 0.9)

discount_factor	Random Car	Q-uber
0.01	**3888.754**	**365.352**
0.05	**4210.578**	**277.898**
0.1	**3779.118**	**428.506**
0.5	**3819.67**	**67.61**
0.99	**3996.006**	**25.554**

Tabela 3. Porównanie RandomCar i Q-uber na podstawie discount_factor

W pierwszych 3 przypadkach można zauważyć sytuację analogiczną do uprzednio zaobserwowanej. Przy zbyt małej wartości parametru *discount_factor*, agent nie jest wystarczająco trenowany, przez co zaczyna kręcić się w kółko.

W dwóch ostatnich wierszach tabeli możemy zaobserwować, że im większy współczynnik *discount_factor,* tym mniejsza liczba kroków agenta do celu.

III. Learning Rate

(Episodes = 10 000, discount_factor = 0.9, epsilon = 0.9)

learning_rate	Random Car	Q-uber
0.01	**4024.46**	**403.562**
0.05	**4056.988**	**152.836**
0.1	**3970.076**	**44.304**
0.5	**3952.2**	**39.342**
0.99	**3996.006**	**30.508**

Tabela 3. Porównanie RandomCar i Q-uber na podstawie learning_rate

Na podstawie powyższej tabeli również łatwo jest zaobserwować, że wraz ze wzrostem współczynnika *learning_rate*, maleje średnia liczba kroków Q-ubera.

IV. Epsilon

(Episodes = 10 000, discount_factor = 0.9, learning_rate = 0.5)

Epsilon jest parametrem określającym jak często podczas trenowania modelu agent powinien wybierać losową akcję (eksplorować środowisko), zamiast wybrać najkorzystniejszą z obecnie znanych.

epsilon	Random Car	Q-uber
0.01	**4045.578**	**196.598**
0.05	**3734.374**	**276.04583**
0.1	**4005.69**	**261.853**
0.5	**3952.2**	**39.342**
0.99	**3918.246**	**35.9190**

Tabela 3. Porównanie RandomCar i Q-uber na podstawie epsilon

Pierwsze 3 wiersze tabeli znowu wykazują złe wytrenowanie modelu. Agent przykłada małą wagę do eksploracji środowiska, przez co jego polityka ograniczona jest do podejmowania obecnie najlepszej akcji. Po odpowiednim wytrenowaniu modelu widzimy, że podobnie jak pozostałe współczynniki, po osiągnięciu wartości optymalnej, dalsza modyfikacja *epsilona* nie niesie za sobą szczególnej poprawy działania algorytmu.