WOJSKOWA AKADEMIA TECHNICZNA

im. Jarosława Dąbrowskiego

WYDZIAŁ CYBERNETYKI

Sprawozdanie

Zaawansowane metody uczenia maszynowego

Uczenie ze wzmocnieniem - Q-learning

Autor:

Karol Baranowski

Prowadzący:

mgr inż. Przemysław Czuba

Spis treści

Zadanie

1. Zapoznaj się z:
    a. Informacjami o RL: https://github.com/dennybritz/reinforcement-learning/tree/master/Introduction
    b. Środowiskiem OpenAI Gym - http://gym.openai.com/docs/
    c. Implementacją Q-learning'u w problemie Taxi – **self_driving_taxi.*py***
2. Zaimplementuj algorytm Q-learning dla wybranego środowiska OpenAI Gym (za wyłączeniem cartpole). Implementację oraz wyniki przedstaw w sprawozdaniu.

1. Zapoznanie się z RL

Zapoznano się z materiałami z załączonych linków o uczeniu ze wzmocnieniem, przejrzano dostępne środowiska na OpenAI Gym oraz zapoznano się z algorytmem Q-learning na podstawie załączonego przykładu.

2. Implementacja Q-learning

Implementacja załączona jest jako plik lab7.ipynb i szczegółowo opisana w komentarzach. Opis implementacji i działanie algorytmu Q-learning przedstawiono w poniższych podrozdziałach.

2.1. Wybór środowiska

Wybrano grę Frozen Lake. Celem tej gry jest przejście ze stanu początkowego (S - start) do stanu celu (G - goal), chodząc tylko po zamrożonych płytkach (F - frozen) i unikając dziur (H - hole).

2.2. Przygotowanie środowiska

```
[1]  1 import random
     2 import numpy as np
     3 import gym
     4
     5 # załadowanie środowiska
     6 env = gym.make("FrozenLake-v0", is_slippery=False).env
     7 # reset środowiska do losowego stanu
     8 env.reset()
     9
    10 # tworzenie QTable
    11 action_size = env.action_space.n # ilość akcji (kolumny)
    12 state_size = env.observation_space.n # ilość stanów (wiersze)
    13 qtable = np.zeros((state_size, action_size)) # inicjalizacja qtable zerami
    14 print(qtable)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

W pierwszym kroku wczytano środowisko FrozenLake-v0 z opcją is_slippery=False, tzn. na lodzie nie można się poślizgnąć i przejść na niezamierzone pole lub dziurę co czyni grę nielosową. Następnie zainicjalizowano Quality table (dalej nazywaną qtable), czyli tabelę jakości akcji dla danego stanu, zerami. Liczba wierszy (stany) i kolumn (akcji) ustalona jest według parametrów danego środowiska. Jak widać w tabeli stanów jest 16 (liczba pozycji na mapie), a akcji cztery (przejście w dół, górę, lewo, prawo)

2.3. Hiperparametry

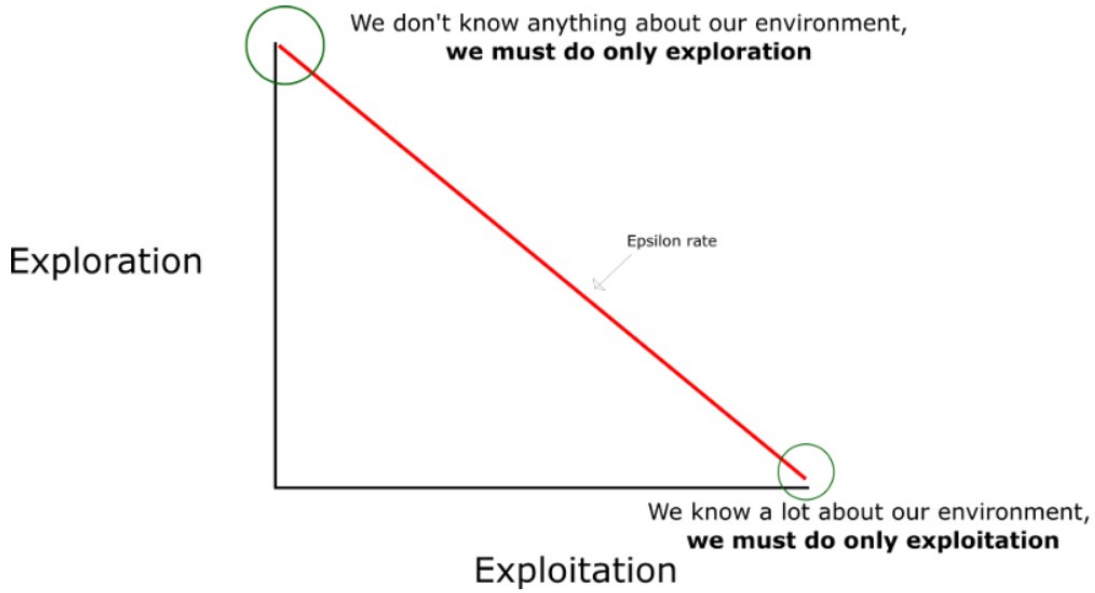
```
[2]   1 # Parametry
      2 episodes = 10000
      3 lr = 0.8
      4 steps = 99 # maksymalna liczba kroków na epokę
      5 gamma = 0.95 # współczynnik obniżenia
      6
      7 # Parametry eksploracji
      8 eps = 1.0 # współczynnik eksploracji
      9 max_eps_prob = 1.0 # prawdopodobieństwo eksploracji na starcie
     10 min_eps_prob = 0.01 # najmniejsze prawdopodobieństwo eksploracji
     11 decay_rate = 0.005 # szybkość zaniku wykładniczego
```

Końcowym zamierzeniem algorytmu Q-learning jest posiadanie dobrze wyliczonej qtable tzn. takiej, na podstawie której agent będzie umiał poprawnie przejść grę. Trenowanie qtable polega na podejmowaniu akcji w danym stanie i aktualizowaniu w pętli wartości w qtable, aby zmaksymalizować nagrodę wynikającą z podjęcia odpowiedniej akcji. W celu zaktualizowania wartości jakości w tabeli Q(s,a), używa się równania Bellmana, które wyjaśniono bardziej szczegółowo w następnym podrozdziale. Aby jednak uczenie było jak najlepsze trzeba ustawić pewne parametry, które dobrano jak na zrzucie powyżej metodą eksperymentalną (dla tych parametrów, agent bezbłędnie przechodzi grę):

1. liczbę epizodów – ile razy zostanie wykonana pętla trenująca
2. learning rate – współczynnik uczenia
3. maksymalną ilość kroków, którą można zrobić w epoce, co eliminuje możliwość wybrania nieskończonej drogi np. chodzenie cały czas w prawo
4. parametry eksploracji:

   Określamy współczynnik eksploracji „epsilon”, który na początku ustawiamy na 1. To jest stosunek kroków, które wykonamy losowo do wszystkich wykonanych kroków. Na początku współczynnik ten musi być najwyższy, ponieważ nie wiemy nic o wartościach w qtable. Oznacza to, że musimy przeprowadzić wiele eksploracji, wybierając losowo nasze działania. Na początku szkolenia w funkcji Q (Bellmana) potrzebny jest duży epsilon na początku. Następnie zmniejszamy go

stopniowo (wykorzystując decay_rate), gdy agent będzie bardziej pewny szacowania wartości Q, aż do 0.01. Poniżej przedstawiono zilustrowanie procesu



2.4. Uczenie z wykorzystaniem algorytmu Q-learning

Pseudokod algorytmu wygląda następująco:

1. Inicjalizacja wartości Q(s,a) zerami dla wszystkich par stan – akcje
2. Dopóki nie stracisz życia, przejdziesz grę lub nie przekroczysz maksymalnej liczby kroków:
    1. Wybierz akcję a w danym stanie s, na podstawie najwyżej wartości Q(s, lista akcji)
    2. Przeprowadź tą akcję i zapisz stan wynikowy s’ oraz nagrodę r, wynikającą z przejścia na ten stan
    3. Zaktualizuj Q(s,a) według równiania Bellmana:

$$NewQ(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma max Q'(s',a') - Q(s,a)]$$

New Q value for that state and that action

Current Q value

Learning Rate

Reward for taking that action at that state

Discount rate

Maximum expected future reward **given the new s' and all possible actions at that new state**

Powyższe równanie można zapisać jako:

```
Nowa wartość Q = aktualna wartość Q + współczynnik uczenia * [nagroda +
decay_rate * (najwyższa wartość Q między możliwymi akcjami z nowego stanu
s) - aktualna wartość Q]
```

W poniższym zrzucie implementacja algorytmu wraz komentarzami objaśniającymi poszczególne linie:

```
# Lista nagród
rewards = []
# Pętla ucząca
for episode in range(episodes):
  state = env.reset() # reset gry - początkowe ustawienie
  step = 0
  done = False
  total_rewards = 0
  for step in range(steps):
    # Wybranie akcji w obecnym stanie świata (s)
    exp_exp_tradeoff = random.uniform(0,1) # Losowa wartość
    # Jeżeli wylosowana wartość > eps, to wykonuj eksploitację
    # (wzięcie największej wartości Q dla danego stanu)
    if exp_exp_tradeoff > eps:
      action = np.argmax(qtable[state,:]) # wybranie najwyżej wartości z wiersza
    else:
      action = env.action_space.sample() # eksploracja - wybierz losową akcję
    # Dla wybranej akcji policz stan i nagrodę
    new_state, reward, done, info = env.step(action)
    # Równanie Bellmana - aktualizacja wartości Q w qtable:
    qtable[state, action] = qtable[state, action] \
     + lr * (reward + gamma * np.max(qtable[new_state, :]) - qtable[state, action])
    # licznik nagród
    total_rewards += reward
    # aktualizacja stanu
    state = new_state
    # Przedwczesny koniec, porażka w grze:
    if done == True:
      break
  # Redukcja epsilonu, aby z każdą sukcesywną epoką mieć mniej eksploracji,
  # a więcej eksploitacji
  eps = min_eps_prob + (max_eps_prob - min_eps_prob) * np.exp(-decay_rate
                                                              * episode)
  rewards.append(total_rewards)
print("Training finished.\n Average rewards per episode: " + str(sum(rewards)/episodes))
print(qtable)
```

Pętla ucząca przechodzi przez liczbę ustawionych w parametrach epizodów. W każdym epizodzie stan gry jest resetowany, kroki i nagroda zerowana, a flaga done oznaczająca koniec ustawiana jako False. Następnie sterowanie wchodzi do wewnętrznej pętli iterującej przez ilość kroków wykonanych w epizodzie. Ustawiono maksymalną liczbę kroków na 99, chociaż co widać kodzie, jeżeli

stracimy grze życie, done przestawi się na True i wykona się break z pętli wewnętrznej.

Na początku pętli wewnętrznej generowana jest liczba losowa. Jeśli będzie ona większa niż epsilon, wówczas dokonamy eksploitacji (oznacza to, że wykorzystujemy to, co już wiemy, aby wybrać najlepszą akcję na każdym etapie). W przeciwnym razie wybieramy się eksplorację.

W dalszej części kodu liczona jest wartość nowego stanu i nagroda, a wartość Q(s,a) aktualizowana jest wedle omówionego równania Bellmana. W *Frozen Lake* możliwe są 3 nagrody o wartościach:

- 0 – przejście na neutralne pole
- -1 – wpadnięcie do dziury, gra jest wtedy zakończona
- 1 – dojście do mety

Następnie aktualizowany jest epsilon określający stosunek eksploracji do eksploitacji i liczone statystyki. Poniżej tabela z wynikiem uczenia po 10000 epizodach:

```
Training finished.
 Average rewards per episode: 0.9661
[[0.73509189 0.77378094 0.6983373  0.73509189]
 [0.73509189 0.         0.66183271 0.69752098]
 [0.69833177 0.49165312 0.40141551 0.45412746]
 [0.63584666 0.         0.         0.        ]
 [0.77378094 0.81450625 0.         0.73509189]
 [0.         0.         0.         0.        ]
 [0.         0.9025     0.         0.63667347]
 [0.         0.         0.         0.        ]
 [0.81450625 0.         0.857375   0.77378094]
 [0.81450625 0.9025     0.9025     0.        ]
 [0.857375   0.95       0.         0.857375  ]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.         0.9025     0.95       0.857375  ]
 [0.9025     0.95       1.         0.9025    ]
 [0.         0.         0.         0.        ]]
```

Średnio na epizod zdobyto 0.97 nagrody na maksymalnie 1 co znaczy, że otrzymano wysoko wytrenowaną tabelę. W tabeli po wytrenowaniu widać, że niektóre kroki są prawie zawsze pewne (bliskie 1), niektóre nigdy nie podejmowane

0, a czasem różnica jest mała między paroma wartościami, jednak dzięki aktualizacji epsilona, agent będzie skupiał się na eksploitacji.

2.5. Ewaluacja agenta po uczeniu

po 10000 epizodach trenowania, qtable będzie wykorzystywana jako „cheatsheet” przez agenta jakie akcje wykonywać w danym stanie. Poniższy fragment kodu prezentuje przejście gry przez agenta 100 razy według wartości z wytrenowanej qtable:

```
# Agent grający w Frozen Lake
env.reset()
episodes, total_epochs = 100, 0
rewards = []
for episode in range(episodes):
  state = env.reset()
  step, total_rewards = 0, 0
  done = False

  for step in range(steps):
    # Wybierz akcję (indeks qtable) z najwyższą wartość nagrody dla tego stanu
    action = np.argmax(qtable[state,:])
    new_state, reward, done, info = env.step(action)
    total_rewards += reward
    if done:
      # ostatni stan - agent dotarł do celu lub wpadł do dziury
      total_epochs += step
      break
    state = new_state
  rewards.append(total_rewards)

print(f"Results after {episodes} episodes:")
print(f"Average timesteps per episode: {total_epochs/episodes}")
print(f"Average rewards per episode: {sum(rewards)/episodes}")
```

```
Results after 100 episodes:
Average timesteps per episode: 5.0
Average rewards per episode: 1.0
```

Agent w każdym epizodzie w każdym kroku wybiera maksymalną wartość dla danego stanu tzn. akcję ruchu w jedną ze stron. W trakcie przechodzenia liczone są również statystyki i po 100 epizodach agent średnio przechodzi grę w 5 krokach i zdobywa 1 punkt tzn. wszystkie 100 epizodów udało mu się przejść, więc używając tak wytrenowanej tabeli agent przechodzi grę bezbłędnie. Poniżej przykładowe przejście gry prezentujące wszystkie stany po kolei:

```
  2   Wizualizacja
  3 '''
  4 state = env.reset()
  5 done = False
  6 while not done:
  7   env.render()
  8   action = np.argmax(qtable[state])
  9   state, reward, done, info = env.step(action)
 10 env.render()
```

```
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
```

Agent za każdym razem wybrał ta samą optymalną drogę, unikając dziur stosując się najwyższej wartości w qtable.

Wnioski

1. Cel zadania został osiągnięty. Wszystkie polecenia udało się wykonać i zaimplementować algorytm uczenia ze wzmocnieniem – Q-learning.
2. Algorytm szacuje, które działanie należy podjąć na podstawie funkcji określającej wartość akcji, która określa wartość bycia w określonym stanie i podjęcia określonego działania w tym stanie.
3. Celem jest maksymalizacja funkcji wartości Q (oczekiwana przyszła nagroda przy danym stanie i działaniu).
4. Dzięki qtable, agent może znaleźć najlepszą akcję dla każdego stanu.
5. Algorytm Q-learning sprawdza się dla prostych gier i środowisk z OpenAI gym, ponieważ analizuje on wszystkie stany występujące w danym środowisku, a gry te mają stosunkowo niewielką liczbę stanów. Złożone środowiska jak gra Doom, sterowanie sztuczną dłonią czy autonomicznym samochodem posiada zbyt dużą liczbą wszystkich możliwych stanów, aby konstruować statyczną qtable. Dlatego stosuje się algorytmy deep Q-learning, które używają sieci neuronowych do podejmowania decyzji w stanach, które się wydarzą.