# Ćwiczenie 7

# Uczenie ze wzmocnieniem

Adam Nowakowski (300428)

## 1. Zadanie

Zadaniem była implementacja *algorytmu Q-learning, która ma działać na dowolnym środowisku OpenAI gym o dyskretnych przestrzeniach akcji oraz stanów. Należało także przetestować działanie algorytmu na środowisku FrozenLake8x8-v0.*

## 2. Wprowadzenie teoretyczne

Problem uczenia ze wzmocnieniem można opisać stosując model procesu decyzyjnego Markowa (PDM), który zawiera:

- $x \in X$ – przestrzeń stanów
- $u \in U$ – przestrzeń akcji
- $P_x(x_{t+1}|x_t, u_t)$ – stały rozkład przejścia stanów
- $r_t = r(u_t, x_{t+1})$ – nagrody
- $X^*$ - stany terminalne epizodów
- $P_0$ – rozkład stanów początkowych epizodu
- $P_x, r, X^*, P_0$ – środowisko

Aby lepiej zobrazować sobie, jak PDM działa, model można rozłożyć na 3 oddzielne byty:

- **Środowisko** w którym postępuje jakiś proces dynamiczny.
- **Agent akcjami** oddziałuje na środowisko (zmienia jego stan). Dobiera je za podstawie **stanu**, który zaobserwował wcześniej oraz tego co się zdążył nauczyć.
- **Arbiter** przygląda się całemu oddziaływaniu pomiędzy agentem i środowiskiem i przyznaje agentowi nagrody (im lepiej agent zareagował na dany stan środowiska, tym lepsza nagroda).

Sposób w jaki agent dobiera akcję na podstawie zaobserwowanego stanu nazywa się **polityką** $\pi \in \Pi$. Naszym celem jest dobranie tak polityki, aby zmaksymalizować wypłacaną w przyszłości nagrodę. Narzędziami do analizy polityki są:

- Stany oraz akcje
- Środowisko $P_x, r, X^*, P_0$ oraz polityka $\pi$
- Funkcja wartości $V^\pi(x) = \varepsilon\left(\sum_{i=0}^{\infty} \gamma^i r_{t+1} \middle| x_t = x; \pi\right)$, która przypisuje stanowi sumę zdyskontowanych nagród, których agent może się spodziewać w

przyszłości, startując ze stanu, który jest argumentem tej funkcji oraz posługując się daną polityką.

- Funkcja wartości-akcji $Q^{\pi}(x,u) = \varepsilon\left(\sum_{i=0}^{\infty} \gamma^{i} r_{t+1} \middle| x_t = x, u_t = u; \pi\right)$ jest podobna do funkcji wartości z dokładnością do faktu, że Q w argumencie przyjmuje też akcję, którą wykonuje na początku, a dopiero później posługuje się daną polityką.

Za pomocą $V^{\pi}(x)$ oraz $Q^{\pi}(x,u)$ można w łatwy sposób indukować politykę, która jest nie gorsza od poprzedniej, czyli zbiega do optymalnej. Opisuje to algorytm iteracji polityki. Z niego wynika bezpośrednio algorytm **Q-learning**. Jego ideą jest utrzymywanie funkcji $Q$, która ma zbiegać do $Q^{*}$. Przedstawiony jest on poniżej:

1. Należy zainicjować $t = 0$ oraz $Q$
2. Wylosować akcję $u_t$ oraz zarejestrować $x_{t+1}$ i $r_t$
3. Poprawić $Q$ w następujący sposób:
$$Q(x_t, u_t) = Q(x_t, u_t) + \beta(r_t + \gamma \max_{u} Q(x_{t+1}, u) - Q(x_t, u_t))$$
4. $t = t + 1$, wrócić do pkt 2.

Tak wyznaczona funkcja $Q$ będzie w przyszłości służyć do indukcji polityki – w danym stanie będzie wybierana akcja, której wybranie daje największą sumę zdyskontowanych nagród.

# 3. Implementacja

W katalogu `/implementacja` znajdują się dwa skrypty w kodzie Pythona:

- *`q_learning.py`* – zawiera klasę, w której zaimplementowano algorytm Q-learning oraz klasy, które pomogły w jego implementacji.
- *`test.py`* – zawiera przykładowe użycie algorytmu

Klasa `Agent` implementuje reprezentację agenta. W swoich polach zawiera informacje nt. nagrody jaką zebrał, stanu w którym się znajduje i funkcji $Q$, która będzie trenowana. Za pomocą swoich metod może zaobserwować aktualny stan i nagrodę, wykonać akcję – w przypadku gdy funkcja została już nauczona (najlepsza akcja), lub gdy dopiero się uczy (wykorzystywana jest strategia Bolzmanowaska). Jest też metoda, która resetuje stan agenta.

Klasa `QFunction` implementuje reprezentację funkcji $Q$, którą będziemy trenować. Zawiera pole – tabelę, która reprezentuje sumy zdyskontowanych nagród w danym stanie i przy wybraniu danej funkcji. Na obiekcie tej klasy można wykonać metody: która zwraca najlepszy ruch i przypisaną do niej sumę nagród oraz taką, która zwraca sumę nagród dla danego stanu i danego ruchu

Funkcje `lake_arbiter` oraz `basic_arbiter` na podstawie nagrody, która jest obserwowana ze środowiska, daje agentowi nagrody, które są zmodyfikowane, by mógł się on lepiej nauczyć.

Funkcja `q_learning` implementuje algorytm Q-learning przedstawiony w paragrafie 2.

## 4. Rozwiązanie zadania

Środowisko, w którym jest nasz agent ma mocno niedeterministyczną naturę. Jeśli agent chce wykonać ruch np. w prawo, to ma pewność, że nie pójdzie on w przeciwną stronę. Ma jednak $1/3$ szans, że pójdzie w którymś z pozostałych kierunków. Badając parametry, najlepszym średnim wynikiem, jaki udało mi się uzyskać, to było ok. 65% poprawności. Nie jest to dużo, lecz zważając na naturę środowiska, można powiedzieć, że to dobry wynik.