Komparatívna analýza vybraných *reinforcement learning algoritmov* a ich výkonu v závisosti od zvolených parametrov na jednoduchých *OpenAI Gym* hrách

Denisa Lampášová

3. februára 2019

Abstrakt

TODO

1 Úvod

Idea učenia algoritmu čisto na základe hodnotenia jeho správnania pozitívnym/negatívnym rewardom a skutočnosť, že sa tak naozaj dobre vedia naučiť napríklad hrať hry[1] alebo nájsť vhodnú stratégiu pre obchodovanie s akciami[2] mi prišla veľmi fascinujúca. Tieto algoritmy nám taktiež poskytujú nové možnosti, ako sa naučiť niečo sami o sebe, napríklad akú výhodu majú predchádzajúce vedomosti pri hraní videohier[3] alebo možnosť zistiť aké reward funkcie nás mohli viesť k naučeniu sa rôznych úkonov[4]. Bohužiaľ som nemala čas ani nápad na podobne dobrý výskum, ale chcela som sa aspoň poriadne naučiť základy reinforcement learningu a to spísaním teoretického úvodu a experimentovaním s rôznymi konceptami a algoritmami v tejto oblasti.

Konkrétne som sa rozhodla naimplementovať[5] *Value iteration, Policy iteration, Q-learning, SARSA*-u *a Deep Q-learning* na natrénovanie agentov majúcich dobrý performance pri hraní jednoduchých *OpenAI Gym* hier – *FrozenLake* a *CartPole* a pohrať sa s tým, ako tieto algoritmy závisia od rôznych parametrov. Použité algoritmy sú dostupné tu: TODO.

FrozenLake

Je krásny zimný deň a s kamarátmi si hádžete frisbee v parku. Pri jednom nešťastnom hode sa ti však podarilo hodiť disk do stredu jazera. Našťastie je jazero z väčšej časti zamrznuté, ale je tam niekoľko dier, do ktorých nechceš spadnúť. Tvojou úlohou je dostať sa úspešne k disku. Ľad je ale klzký, takže nie vždy sa pohneš v zamýšľanom smere.

[1] https://arxiv.org/pdf/1712.01815.pdf

[2] https://arxiv.org/abs/1811.07522

[3] https://www.youtube.com/watch?v=0l0-c9OE3VQ, prípadne samotný článok: https://arxiv.org/pdf/1802.10217.pdf.

[4] https://ai.stanford.edu/~ang/papers/icml00-irl.pdf

[5] Nie nutne from scratch. Teda aspoň pochopiť a použiť už existujúce implementácie algoritmov.

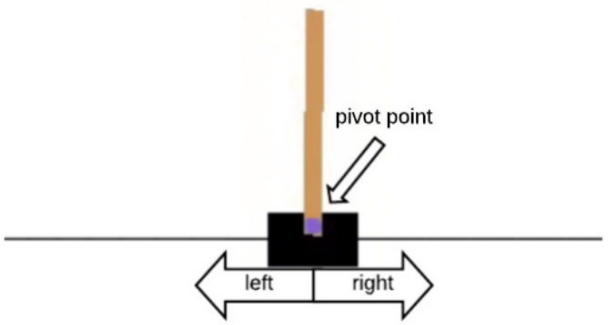
Hracia plocha je popísaná nasledujúcou mriežkou:

$SFFF$	(S : starting point, safe)
$FHFH$	(F : frozen surface, safe)
$FFFH$	(H : hole, fall to your doom)
$HFFG$	(G : goal, where the frisbee is located)

Epizóda končí keď dorazíš k frisbee alebo spadneš do diery. Keď dorazíš k frisbee, získavaš reward +1, vo všetkých ostatných prípadoch je tvoj reward 0.

CartPole (Inverted pendulum)

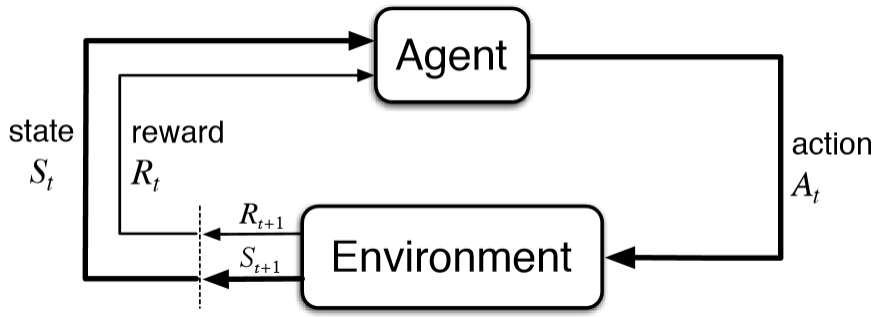
Máme vozidlo, na ktorom je pripevnené inverzné kyvadlo. Aplikovaním sily +1 alebo -1 (iná možná akcia nie je) ovládame pohyb tohoto vozidla po 1 rozmernej čiare (bez trenia). Začiatočný stav (poloha vozidla, rýchlosť vozidla, uhol vychálenia tyče, uhlová rýchlosť tyče) je náhodne inicializovaný medzi +/-0.05 a našim cieľom je zabrániť prevráteniu kyvadla. Za každý časový krok, kedy sa kyvadlo ešte neprevrátilo dostávame odmenu +1. Epizóda končí, keď je tyč vychýlená o viac než 15° (= keď sa kyvadlo prevrátilo) alebo keď sa vozidlo vzdiali viac než 2.4 jednotiek od stredu.



Obr. 1: Ilustrácia CartPole systému.

2 Reinforecement Learning - theoretical background

Markov Decision Process (MDP)



Obr. 2: Schéma MDP.

Agent (decision maker) interaguje s prostredím nasledovne (viď Obr. 2). V každom časovom kroku t agent obdrží nejakú reprezentáciu stavu prostredia s_t. Na základe tohto stavu sa agent rozhodne, ktorú akcu a_t v tomto kroku vykoná. Po vykonaní tejto akcie sa stav prostredia zmení na s_{t+1} a agent ako následok za jeho akciu obdrží nejaký reward r_{t+1}. Dostávame sa do ďalšieho časového kroku a celý proces sa opakuje.

Celkovo teda vytvárame istú postupnosť

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \cdots ,$$

v ktorej sa kumuluje celkový reward

$$R = r_1 + r_2 + r_3 + \cdots .$$

Cieľom agenta je tento reward maximalizovať. Nakoľko je ale často predpovedanie blízkeho rewardu presnejšie, ňež krokovo vzdialenejšieho rewardu a teda, napríklad, ak má agent možnosť vybrať si, či dostane odmenu (dolár) v najbližšom kroku (dnes) alebo až v tom ďalšom (zajtra), chceme aby sa vždy rozhodol pre najbližší krok. Dolár dnes je predsa cennejší než dolár zajtra. Preto zavádzame *discount faktor* γ a agent sa snaží maximalizovať *discounted cummulative reward*

$$\begin{aligned} G_t &= r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+1} \\ &= r_{t+1} + \gamma G_{t+1}. \end{aligned}$$

Ďalšou výhodou je, že ak má agent na výber dlhšiu cestu s väčším rewardom a kratšiu cestu s menším rewardom, vieme výberom γ ovplyvniť, ktorú trasu má zvoliť.

Definícia. *Markov Desicion Process je 5-tica* $(S, A, \{P_{sa}\}, \gamma, R)$*, kde:*

- S *je množina* ***stavov*** *(v prípade FrozenLake-u je to id políčka, na ktorom agent stojí.)*

- *A je množina* ***akcií*** *(Napríklad smery ktorými sa môže agent pohnúť - hore, dole, doľava a doprava.)*
- $p\big(s' \mid s, a\big)$ *are the* ***state transition probabilities*** *- pravdepodobnosť, že sa dostaneme do stavu s', ak v stave s vykonáme akciu a. Môžeme sa na to pozerať aj ako na pravdepodobnostnú distribúciu nad možnými stavmi s', do ktorých nás vie dostať akcia a zo stavu s. (Napríklad vo FrozenLake-u je plocha šmykľavá a preto je nejaká pravdepodobnosť, že my síce zvolíme akciu doprava, ale v skutočnosti sa pohneme napríklad na políčko nad nami.)*
- $\gamma \in [0, 1]$ *je discount faktor,*
- $R : S \to \mathbb{R}$ *je* ***reward funkcia*** *(všeobecnejšie sa definuje ako $R : S \times A \to \mathbb{R}$ - môžeme agenta odmeňovať aj za zvolenie niektorej akcie, nie nutne len za skončenie v niektorom stave).*

Ako ale určiť, ako dobrý je pre agenta nejaký stav alebo akcia? A ako presne sa agent v stave s rozhoduje, ktorú akciu zvolí? Zaveďme si na to teda terminológiu a zodpovedajme tieto otázky.

Policy

Policy je ľubovoľná funkcia $\pi : S \to A$. Úlohou agenta je rozhodnúť sa na základe stavu s, ktorú akciu a zvolí. Policy slúži presne na popis týchto rozhodnutí - hovoríme potom, že agent nasleduje/riadi sa policy π.

Value function

- **State-value function $\boldsymbol{v_\pi(s)}$** nám hovorí, ako dobrý je nejaký stav s pre agenta nasledujúceho policy π
$$v_\pi(s) = \mathrm{E}\big[G_t | s_t = s\big].$$
- **Action-value function** (alebo tiež **Q-function**) $\boldsymbol{q_\pi(s, a)}$ evaluuje, ako dobré je pre agenta v stave s nasledujúceho policy π zvoliť akciu a
$$q_\pi(s, a) = \mathrm{E}\big[G_t | s_t = s, a_t = a\big].$$

Theorem (Bellman equations)**.** *Majme MDP $M = (S, A, \{P_{sa}\}, \gamma, R)$ a policy $\pi : S \to A$. Potom pre všetky $s \in S$, $a \in A$, v_π a q_π spĺňajú*

$$\begin{aligned} v_\pi(s) &= \sum_{s'} p\big(s' \mid s, \pi(s)\big) \cdot \big(R(s') + \gamma \cdot v_\pi(s')\big) \\ &= \mathrm{E}\big[r | s, \pi(s)\big] + \gamma \sum_{s' \in S} p(s' | s, \pi(s)) \cdot v^\pi(s'), \\ q_\pi(s, a) &= \sum_{s'} p\big(s' \mid s, a\big) \cdot \big(R(s') + \gamma \cdot v_\pi(s')\big) \\ &= \mathrm{E}\big[r | s, a\big] + \gamma \sum_{s' \in S} p(s' | s, a) \cdot v^\pi(s'). \end{aligned}$$

Tieto vzťahy sa ukážu byť veľmi užitočné (napríklad vo value a policy iteration algoritmoch), nakoľko nám navrhujú spôsob ako iteratívne vypočítať hodnoty v_π a q_π.

Optimal policy

Cieľom reinforcement learningu je nájsť **optimálnu policy $\pi^\star$**. Hovoríme, že policy π je lepšia/optimálnejšia než policy π' ($\pi \geq \pi'$) práve vtedy, keď $v_\pi(s) \geq v_{\pi'}(s)$ pre všetky $s \in S$. Nuž a optimálna policy je lepšia než akákoľvek iná možná policy.

K optimálnej policy prislúcha aj **optimálna state-value funkcia $v_\star$**

$$v_\star(s) = \max_\pi v_\pi(s) \quad (\forall s \in S),$$

čo je vlastne najlepšia možná očakávaná hodnota discounted cummulative rewardu od toho momentu, ktorá vie byť dosiahnutá. Môžeme taktiež definovať **optimálna action-value funkcia $q_\star$** ako

$$q_\star(s,a) = \max_\pi q_\pi(s,a) \quad (\forall s \in S, \forall a \in A),$$

ktorá nám zas hovorí, aká je najlepšia možná očakávaná hodnota discounted cummulative rewardu potom, čo v stave s vykonáme akciu a.

Theorem (Bellman optimality)**.** *Majme MDP $M = (S, A, \{P_{sa}\}, \gamma, R)$ a policy $\pi : S \to A$. Potom π je optimálna policy práve vtedy, keď pre všetky $s \in S$*

$$\pi(s) \in \arg\max_{a \in A} q_\pi(s,a).$$

Celkovo je teda zrejmé, že

$$\begin{aligned}
v_\star(s) &= \max_a q_\star(s,a), \\
v_\star(s) &= \max_a \Big[\sum_{s'} p\big(s' \mid s,a\big) \cdot \big(R(s') + \gamma \cdot v_\star(s')\big) \Big], \\
q_\star(s,a) &= E\Big[r_{t+1} + \gamma \max_{a'} q_\star(s',a') \Big] = \sum_{s'} p\big(s' \mid s,a\big) \cdot \big(R(s') + \gamma \cdot v_\star(s')\big),
\end{aligned}$$

kde posledné dve rovnice sa nazývajú **Bellman optimality equations**. Teraz máme už slušný teoretický základ a môžeme sa pozrieť na základné algoritmy.

3 Reinforcement learning - algoritmy

3.1 Value iteration

Value iteration počíta optimálnu state-value function tak, že najskôr zinicializuje $v(s)$ na nuly a potom iteratívne, pomocou Bellman optimality equation, vylepšuje hodnoty $v(s)$, kým nezkonvergujú. Keď už máme $v_\star$, vieme jednoducho (použitím Bellman otimality theorem) získať optimálnu policy $\pi^\star$. Pseudokód:

```
Initialize v(s) to arbitrary values (usually 0s);
repeat
    for all s ∈ S do
        for all a ∈ A do
            q(s, a) ← E[r|s, a] + γ · Σ_{s'∈S} p(s'|s, a) · v(s');
        end
        v(s) ← max_a q(s, a);
    end
until v(s) converges;
```

Algorithm 1: Pseudocode for value iteration algorithm.

Máme ale 2 možnosti, ako update-ovať $v(s)$ vo vnútornom *for*-cykle.

1. V každej iterácii (repeat till $v(s)$ converges), najskôr pre všetky $s \in S$ vypočítame $v(s)$ a až potom sunchrónne prepíšeme všetky hodnoty. Takýto spôsob voláme **synchronous** update.

2. Môžeme použiť aj **asynchronous** update, kedy pre každý stav $s \in S$, hneď ako zistíme novú hodnotu $v(s)$, ju priamo aj prepíšeme. Nečakáme kým, zistíme ostatné.

V oboch prípadoch sa dá ukázať, že v naozaj skonverguje do $v_\star$. Nevýhodou tohoto algoritmu je ale predpoklad, že vlastnosti prostredia už poznáme. Tieto vlastnosti nám ale nie sú vždy popredu známe (potom ich samozrejme je možné nejako odpozorovať, ale získame tým nejaké nepresnosti). Taktiež, priamočiaro vieme tento algoritmus použiť len pre prípady s diskrétnou množinou stavov S (ako je napríklad vo FrozenLake-u). Pre prípad so spojitou množinou stavov S (ako je napríklad v CartPole-e), je potrebné túto množinu diskretizovať, čo môže zapríčiniť, že policy, ktorú získame, už nebude optimálna.

3.2 Policy iteration

Value iteration algoritmus vylepšuje hodnoty $v(s)$ až kým neskonverguje. Ale všetko, čo agent potrebuje vedieť je len optimálna policy. Niekedy sa stáva, že optimálna policy zkonverguje skôr než value funkcia. Policy iteration algorithm teda miesto opakovaného vylepšovania hodnôt value funkcie, v každom kroku redefinuje policy. Na základe tejto novej policy potom iteratívne vypočíta príslušnú state-value funkciu v_π. Podľa vypočítanej value funkcie potom opäť predefinuje policy a proces sa opakuje, kým policy neskonverguje. Pseudokód:

Initialize a policy π' arbitrarily;
repeat
1. $\pi \leftarrow \pi'$;
2. Iteratively evaluate the value-function under policy (for all states, till v^π converges):
$v^\pi(s) = \mathrm{E}\big[r|s, \pi(s)\big] + \gamma \sum_{s' \in S} p(s'|s, \pi(s)) \cdot v^\pi(s')$;
3. Improve the policy at each state:
$\pi'(s) \leftarrow \arg\max_a(\mathrm{E}\big[r|s, a\big] + \gamma \sum_{s' \in S} p(s'|s, a) \cdot v^\pi(s'))$;

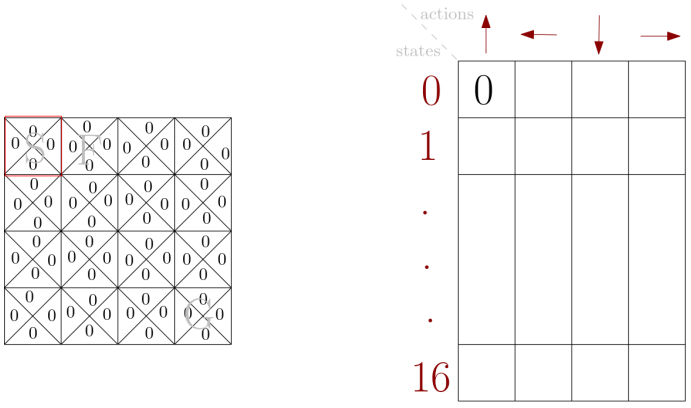
until $\pi = \pi'$;

Algorithm 2: Pseudocode for policy iteration algorithm.

Policy iteration tiež zaručenie skonverguje k optimálnej policy a často jej zkonvergovanie trvá menej iterácií než value iteration algoritmu, každá z týchto iterácií je ale výpočtovo náročnejšia. Taktiež tu platia rovnaké problémy s diskretizáciou a nevedomosťou prechodových pravdepodobností a raward funkciou.

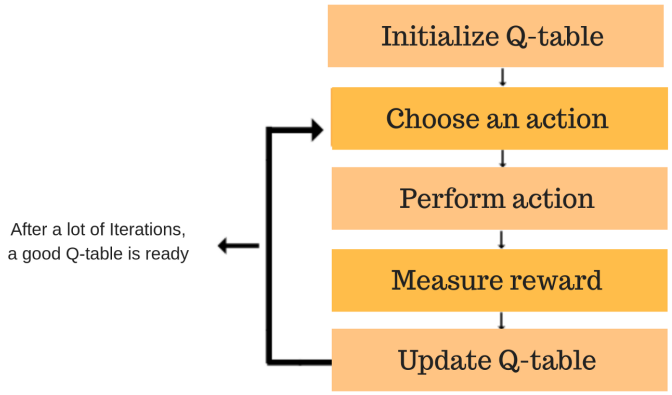
3.3 Q-learning

Prvou výhodou tohto algoritmu je, že nepredpokladá, že agent pozná efekty jeho akcií popredu. Agent pozná iba množinu možných stavov S a akcií A. Na ich základe si na začiatku vytvorí a inicializuje **q-table** na nuly. Následne chceme postupne update-ovať hodnoty (action-value funkcie) v tabuľke.



Obr. 3: Dve reprezentácie tej istej q-tabuľky pre FrozenLake - okienko predstavuje q(s,a).

Konkrétne, chceme aby agent volil akcie a na základe obdržaného rewardu update-oval hodnoty v tejto tabuľke:



Obr. 4: High-level schéma Q-learning algoritmu.

Akým spôsobom má ale agent voliť akcie? A ako sa má tento proces meniť s časom, keďže na začiatku agent o prostredí nevie nič ale postupne tieto vedomosti získava? Zaveďme teda koncept **exploration vs. exploitation**.

Explorácia je proces, kedy skúmame prostredie a získavame o ňom nejaké informácie. V našom prípade to konkrétne znamená, že akciu budeme voliť náhodne. *Exploitácia* je na druhú stranu proces, kedy využívame už zistenú informáciu o prostredí v snahe maximalizovať reward. Je teda zrejmé, že chceme začať exploráciou a postupne sa prepracovať k čistej exploitácii. Akým spôsobom to ale chceme urobiť?

Veľmi častá je tzv. **epilon greedy stratégia**. V nej sa zavádza **exploration rate** ε, ktorý predstavuje pravdepodobnosť, s ktorým budeme v najbližšom kroku explorovať. Začíname s $\varepsilon = 1$ a postupne ho po krokoch/epizódach znižujeme. Často si ho ale zastavíme na nejakej hranici, napríklad 0.1.

```
# Exploration-exploitation trade-off
exploration_rate_treshold = random.uniform(0,1)
if exploration_rate_treshold > exploration_rate:
    action = np.argmax(q_table[state,:]) #"najväčší reward" (exploitácia)
else:
    action = env.action_space.sample() #náhodná akcia (explorácia)
```

Už teda vieme ako voliť akcie. Akým spôsobom ale máme update-ovať hodnoty q-funkcie? Podľa vedomostí získaných v prvej časti, vieme, že týmto krokom získaná hodnota danej q-funkcie, by mala byť

$$q(s,a) = r_{t+1} + \gamma \max_{a'} q(s',a').$$

My sme už ale čo-to o tejto hodnote zistili. Preto hodnotu q-funkcie update-ujeme nasle-

dovne:

$$q^{new}(s,a) = (1-\alpha) \cdot \underbrace{q(s,a)}_{old\ value} + \alpha \cdot \overbrace{\left(r_{t+1} + \gamma \max_{a'} q(s',a')\right)}^{learned\ value}.$$

Čiže dávame nejakú váhu, konkrétne $1 - \alpha$, predchádzajúcim vedomostiam.

3.4 SARSA

3.5 Deep Q-learning network (DQN)

```
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam

# neural net to approximate Q-value function:
model = Sequential()
model.add( Dense(24, input_dim=state_size, activation='relu') )
model.add( Dense(24, activation='relu') )
model.add( Dense(action_size, activation='linear') )
model.compile(loss='mse', optimizer=Adam(lr=learning_rate))
```

4 Analýza

TODO

5 Záver

TODO