

Wykorzystanie systemu do głębokiego uczenia opartego o model actor-critic do poruszania się mobilnego agenta na torze wyścigowym w grze TORCS

Izabela Ryś-Salata

1 Wstęp

Niniejsza praca omawia wykorzystanie algorytmu DDPG (Deep Deterministic Policy Gradient) do poruszania się mobilnego agenta na torze wyścigowym w grze TORCS dostępnej na licencji open-source.

2 Środowisko

Całość kodu źródłowego jest napisana w języku Python3.

W projekcie wykorzystano:

- Tensorflow2,
- TORCS (The Open Racing Car Simulator) - <http://torcs.sourceforge.net/>,
- nakładkę do TORCS - https://github.com/ugo-nama-kun/gym_torcs,
- numpy, matplotlib.

Zarówno uczenie, jak i testowanie przeprowadzano w środowisku Windows10.

3 TORCS

Środowisko TORCS umożliwia łatwe szkolenie modeli sterujących agentem. Szczegółowy opis oprogramowania, zarówno od strony konfiguracji, jak i zarządzania agentem w czasie wyścigu znajduje się w pracy: <https://arxiv.org/pdf/1304.1672.pdf>.



Bardzo ważnym elementem projektu jest nakładka `gym_torcs`, umożliwiająca łatwą komunikację między uczonym modelem a środowiskiem TORCS.

W każdym tiku gry agent dostaje obserwację ze środowiska. Z obserwacji wybierane są elementy *znaczące*:

1. **angle**: kąt między kierunkiem pojazdu i kierunkiem osi toru,
2. **track**: wektor 19 sensorów zawierających informacje o odległości pojazdu od brzegu toru,
3. **trackPos**: odległość między pojazdem a osią toru, znormalizowana do $[-1, 1]$,
4. **speedX**: prędkość pojazdu na jego osi wzdłużnej,
5. **speedY**: prędkość pojazdu na jego osi poprzecznej.

Następnie agent decyduje o akcji, na którą składa się:

1. **speed**: wartość z przedziału $[-1, 1]$, gdzie

$$\begin{cases} \text{speed} \geq 0 & - \text{przyspieszanie} \\ \text{speed} < 0 & - \text{hamowanie} \end{cases}$$

2. **steer**: wartość z przedziału $[-1, 1]$, gdzie

$$\begin{cases} \text{steer} > 0 & - \text{skręt w prawo} \\ \text{steer} < 0 & - \text{skręt w lewo} \end{cases}$$

4 Uczenie maszynowe

Agent został nauczony z wykorzystaniem metody DDPG. Ta metoda jest połączeniem trzech algorytmów:

- Deterministic Policy Gradient,
- Actor-Critic,
- Deep Q-Network.

Algorytm został przedstawiony w pracy <https://arxiv.org/pdf/1509.02971.pdf>.

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ .
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
 Initialize a random process \mathcal{N} for action exploration
 Receive initial observation state s_1
 for $t = 1, T$ **do**
 Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
 Execute action a_t and observe reward r_t and observe new state s_{t+1}
 Store transition (s_t, a_t, r_t, s_{t+1}) in R
 Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
 Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'}))|\theta^{Q'}$
 Update critic by minimizing the loss: $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
 Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

Update the target networks:

$$\theta^{Q'} \leftarrow \tau \theta^Q + (1 - \tau) \theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau \theta^\mu + (1 - \tau) \theta^{\mu'}$$

end for
end for

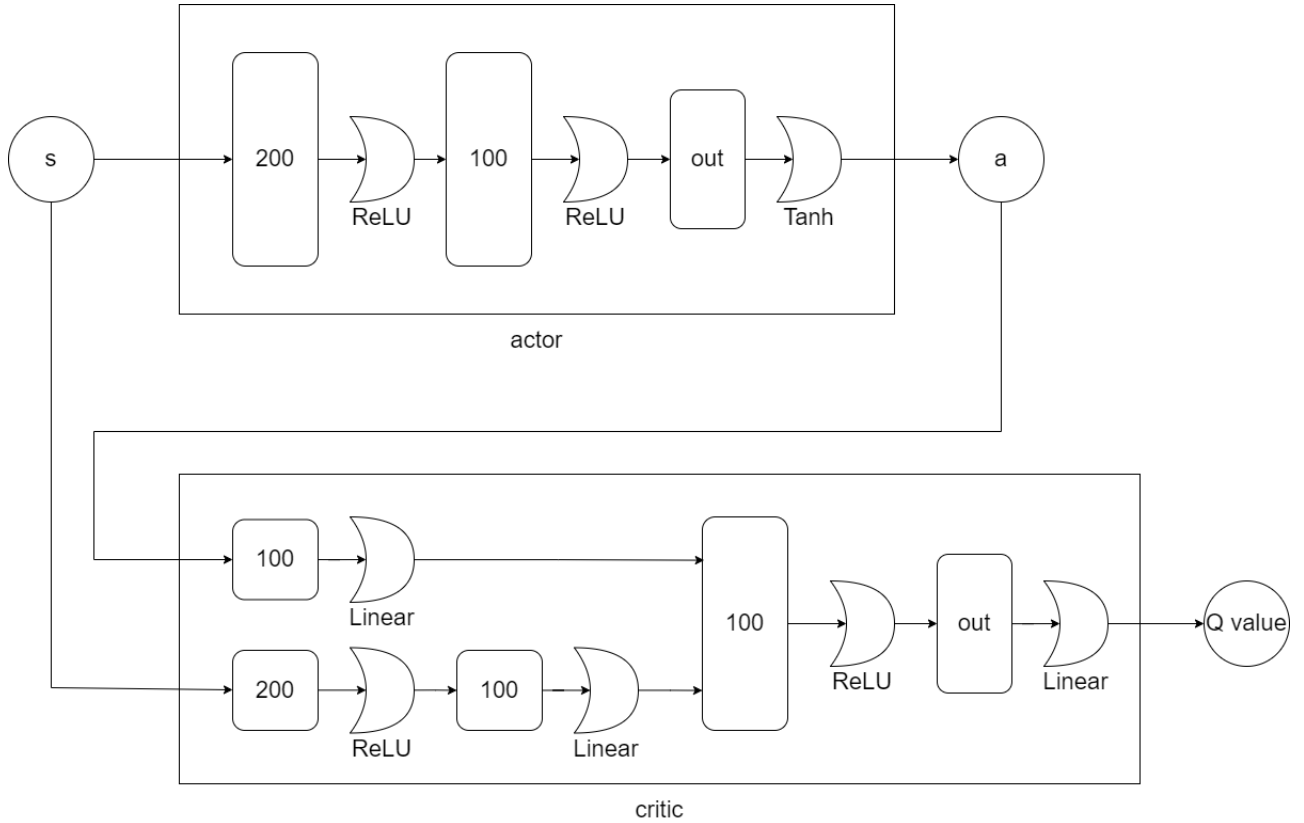
Ważnym elementem jest losowy proces wykorzystany w celu eksploracji środowiska. W projekcie skorzystano z procesu Ornstein-Uhlenbeck, wyrażającego się wzorem:

$$dx_t = \theta(\mu - x_t)dt + \sigma W_t$$

Parametry dobrano jako:

action	θ	μ	σ
speed	1.0	0.5	0.1
steer	0.6	0.0	0.3

Architektura modelu składa się z sieci: actor, target-actor, critic, target-critic, których schemat przedstawiony jest poniżej:



Zadanie poszczególnych sieci jest następujące:

- **actor:** podejmuje akcję (zmiana prędkości i sterowania) na podstawie stanu (obserwacji ze środowiska),
- **critic:** na podstawie stanu i akcji wybranej przez aktora ocenia wybór akcji szacując wartość Q (nagroda).

Metoda DDPG wprowadza dodatkowo sieci *target*, które sprawiają, że uczenie jest bardziej stabilne, ponieważ parametry tych sieci są uaktualniane powoli.

Ostatnim elementem architektury modelu jest bufor, przechowujący wydarzenia z przeszłości. Zamiast opierać się wyłącznie na poprzednich wydarzeniach, uczenie modelu korzysta z losowej próbki wybranej z wydarzeń umieszczonych w buforze, które są postaci:

$$(state, action, reward, next\ state)$$



Nagroda jest zdefiniowana następująco:

$$reward = S \cdot \cos(angle) - |S \cdot \sin(angle)| - S \cdot |trackPos|$$

gdzie S oznacza prędkość pojazdu.

5 Wyniki

Model nauczył się jeździć wystarczająco dobrze, aby pokonać łatwiejsze trasy w dobrym czasie:

nazwa trasy	mapa trasy	czas agenta
F - speedway		2:57:60
E - track 5		02:05:78

Niestety agentowi nie udało się pokonać trasy z ostrzejszymi zakrętami. Wynika to z faktu, że proces uczenia był mało wydajny- sieci składały się z niewystarczającej liczby neuronów, ponieważ ograniczenia sprzętowe nie pozwalały na trenowanie bardziej złożonego modelu.

6 Źródła

1. <http://torcs.sourceforge.net/>
2. https://github.com/ugo-nama-kun/gym_torcs
3. <https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html>
4. https://keras.io/examples/rl/ddpg_pendulum/
5. *Simulated Car Racing Championship Competition Software Manual*, 2013, Daniele Loiacono, Luigi Cardamone, Pier Luca Lanzi
6. *Continuous Control with Deep Reinforcement Learning*, 2016, Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, Daan Wierstra
7. *Temporal Exploration for Reinforcement Learning in Continuous Action Spaces*, 2016, Jeroen van den Heuvel, Marco A. Wiering, Walter A. Kusters