

# Algorytmy uczenia ze wzmacnieniem: DQN, SG, A2C i PPO

Robert A. Kłopotek  
[r.klopotek@uksw.edu.pl](mailto:r.klopotek@uksw.edu.pl)

Wydział Matematyczno-Przyrodniczy. Szkoła Nauk Ścisłych, UKSW

25.11.2025

## Section 1

### Deep Q-Network

# Deep Q-Network

---

- Deep Q-Network(DQN), nazywany też Deep Q-Learning, to opracowana w 2013 roku, ulepszona wersja Q-Learning.
- Zamiast tabeli lub mapy, do reprezentacji wiedzy używa **sieci neuronowej**. Dzięki temu algorytm działa również w przypadku problemów, w których przestrzeń możliwych stanów jest bardzo duża.
- Natomiast nadal nie poradzi on sobie dobrze w wypadku, gdy będzie zbyt wiele możliwych do wykonania akcji. Jest tak dlatego, że algorytmy używające wartości Q muszą ją obliczyć dla każdego możliwego do wykonania ruchu.
- Jednak mimo to, DQN jest bardzo skuteczny i pozwala grać nawet w skomplikowane gry 3D.

## Funkcja straty (loss function)

---

- By aktualizować wagi sieci neuronowej potrzebujemy odpowiedniej funkcji straty(loss). Dla DQN przyjmuje ona postać:

$$\text{loss} = (Q_{\text{target}} - Q_{\text{current}})$$

- $Q_{\text{current}}$  to wartość Q zwracana przez nasz model dla stanu  $s$ .  $Q_{\text{target}}$  obliczamy za pomocą wzoru:

$$Q_{\text{target}} = \text{nagroda} + \gamma \max(Q(s_{\text{next}}))$$

- gdzie  $s_{\text{next}}$  to stan po wykonaniu akcji, natomiast  $\max(Q(s_{\text{next}}))$  to najlepsza wartość Q dla  $s_{\text{next}}$ .  $\max(Q(s_{\text{next}}))$  otrzymujemy z sieci neuronowej.

## Problemy z siecią neuronową

---

- **Zapominanie poprzednich doświadczeń** - wagi uzyskane podczas poprzednich gier są łatwo nadpisywane w wyniku nowszych doświadczeń. Z tego względu agent, który doszedł do końca gry, może sobie nie poradzić z pierwszymi poziomami, które opanował już wcześniej.
- **Korelacja między doświadczeniami** - zapamiętywanie sekwencji akcji zamiast analizowaniu obecnego stanu środowiska, np. bot zaczyna strzelać w miejsce, gdzie powinien być wróg a nie tam, gdzie rzeczywiście on się w danym momencie znajduje

## Rozwiązywanie problemów

---

- Rozwiązaniem obu przedstawionych wcześniej problemów jest **oddzielenie procesów eksploracji i nauki**. Podczas rozgrywania epizodu nie aktualizujemy żadnych wag. Zamiast tego do bufora zapisujemy przebieg rozgrywki (stany, akcje, nagrody). Najlepiej by bufor miał formę kolejki o pewnej maksymalnej wielkości.
- Dopiero po zakończeniu gry rozpoczęamy proces nauki. Nie przebiega on jednak liniowo tak jak rozgrywka. Zamiast tego elementy bufora, które posłużą do nauki wybieramy losowo.
- Zaprezentowane rozwiązanie pozwala na jednoczesne uniknięcie problemu korelacji (sieć nie może stworzyć sekwencji akcji, bo nie są one podejmowane kolejno) i zapominania doświadczeń (zdarzenia dawniejsze i późniejsze w procesie nauki, mogą występować koło siebie).

## Pseudokod DQN (1/2)

---

```
koniec_gry=False
inizjalizuj model m wspolczynnikiem uczenia i losowymi wagami
inizjalizuj ROZMIAR_PROBKI_DO_NAUKI w zaleznosci od problemu
inizjalizuj gamma, liczba_epizodow i rozmiar_max w zaleznosci od problemu
pamiec=kolejka(rozmiar_max)

for(epizod in liczba_epizodow)
{
    stan_srodowiska=srodowisko.reset()
    koniec_gry=False
    while(koniec_gry!=True)
    {
        akcja=max_index(model.przewiduj(stan_srodowiska))
        nowy_stan_srodowiska,koniec_gry,nagroda=srodowisko.krok(akcja)
        pamiec.dodaj_element((stan_srodowiska,nowy_stan_srodowiska,
        koniec_gry,nagroda,akcja))
        stan_srodowiska=nowy_stan_srodowiska
    }
    probka_do_nauki=losuj_probke(pamiec,ROZMIAR_PROBKI_DO_NAUKI)

    ...
}
```

## Pseudokod DQN (2/2)

---

```
for(epizod in liczba_epizodow)
{
    ...

    for(stan_srodowiska ,nowy_stan_srodowiska ,koniec_gry ,nagroda ,akcja
        in probka_do_nauki)
    {
        q_wejsciowe = model.przewiduj(stan_srodowiska)
        q_poprawne=q_wejsciowe.copy()
        q_poprawne[akcja]=nagroda + gamma *
        max(model.przewiduj(nowy_stan_srodowiska))
        if(koniec_gry==True)
        {
            q_poprawne[akcja]=nagroda
        }
        model.popraw_wagi(q_poprawne ,q_wejsciowe)
    }
}
```

## Section 2

Strategia Gradientowa - SG

## Strategia Gradientowa

---

- **Strategia Gradientowa (SG)** jest algorytmem ze strategią, reprezentującym podejście Monte Carlo.
- Jest to algorytm starszy ale jednocześnie skuteczniejszy niż bardziej znane DQN, co przyznają nawet autorzy tego drugiego w swojej pracy.
- W przeciwieństwie do DQN, gdzie krzywa uczenia może mocno oscylować, w SG zwykle obserwować można bardzo stabilny przyrost wiedzy. Dzieje się tak dzięki temu, że zamiast bazować na mało stabilnej wartości Q, przy aktualizacji wag używamy prostego gradientu. Sieć neuronowa wprost szuka najlepszej do wykonania akcji. Niestety ma to też swoją ciemną stronę, bo taki agent często wpada w maksima lokalne.
- Dodatkowo, co prawda proces nauki jest stabilny, ale jednocześnie znacznie wolniejszy od podejść opierających się o obliczanie wartości Q.

## Strategia Gradientowa - zalety względem DQN

---

- Dużą przewagą SG względem DQN jest to, że działa on dobrze zarówno gdy duża jest przestrzeń stanów jak i akcji. To czyni go znacznie lepszym wyborem do bardziej kompleksowych zadań, takich jak na przykład prowadzenie samochodu.
- Kolejną ważną zaletą strategii gradientowej jest to, że może ona użyć strategii stochastycznej. Polega ona na tym, że nasz model zamiast po prostu zwracać akcję jaką powinien wykonać agent, daje prawdopodobieństwo wykonania danego ruchu. Następnie spośród akcji losowana jest (uwzględniając wspomniane wcześniej prawdopodobieństwo) akcja, która ostatecznie zostanie wykonana.
- Jest to ważna cecha, ponieważ dzięki temu nie trzeba we własnym zakresie implementować strategii eksploracji. Implementacja tego elementu może być często dość skomplikowana i mocno zależeć od rodzaju problemu, jaki chcemy rozwiązać. Używając strategii gradientowej mamy gotowe rozwiązanie, które jest uniwersalne i w wielu wypadkach doskonalsze od strategii takich jak  $\epsilon$  – greedy.

## Obliczanie gradientu

---

- Wcześniej do aktualizacji wag używaliśmy funkcji loss i by dokonać najlepszej optymalizacji, musielibyśmy znaleźć jej minimum globalne. Takie podejście nazywa się **gradient descent**.
- W przypadku strategii gradientowej i jej funkcji oceny, stosujemy odwrotne podejście czyli **gradient ascent**. By uzyskać najlepsze wyniki, musimy wtedy znaleźć maksimum funkcji oceny.
- By zaktualizować wagi  $\theta$  sieci neuronowej agenta, posługujemy się następującym wzorem:

$$\theta_{new} = \theta_{old} + \alpha \nabla_{\theta} \log(\pi(\tau|\theta)) \cdot R(\tau)$$

- gdzie  $\tau$  to sekwencją zdarzeń, która uwzględnia stan, akcje i otrzymaną nagrodę:  $\tau = ((s_1, r_1, a_1), (s_2, r_2, a_2) \dots)$ .  $R(\tau)$  to średnia nagroda, jaką otrzymamy za wykonanie sekwencji  $\pi(\tau|\theta)$  to w tym wypadku prawdopodobieństwo wystąpienia serii zdarzeń  $\tau$ , gdy mamy wagi  $\theta$ .

# Pseudokod SG (1/2)

---

```
koniec_gry=False
inizjalizuj model m wspolczynnikiem uczenia i losowymi wagami
inizjalizuj liczba_epizodow
inicjalizuj gamma
for(epizod in liczba_epizodow)
{
    nagrody=lista()
    stany=lista()
    akcje=lista()
    stan_srodowiska=srodowisko.reset()
    koniec_gry=False
    while(koniec_gry!=True)
    {
        akcja=losuj_akcje(model.przewiduj(stan_srodowiska))
        nowy_stan_srodowiska,koniec_gry,nagroda=srodowisko.krok(akcja)
        nagrody.dodaj_element(nagroda)
        stany.dodaj_element(stan_srodowiska)
        akcje.dodaj_element(akcja)
    }
    ...
}
```

## Pseudokod SG (2/2)

---

```
for(epizod in liczba_epizodow)
{
    ...

    #standaryzacja nagrod
    srednia_nagrod=oblicz_srednia(nagrody)
    std=oblicz_odchylenie_standardowe(nagrody)
    for(nagroda in nagrody)
    {
        nagroda=(nagroda-srednia_nagrod)/std
    }
    #nauka
    log_prawdopodobienstwo=model.cross_entropy_with_logits(stany,akcje)
    model.popraw_wagi(log_prawdopodobienstwo*nagrody)
}
```

## Section 3

Advantage Actor Critic - A2C

## Advantage Actor Critic

---

- **Advantage Actor Critic (A2C)** łączy w sobie cechy algorytmów z i bez strategii. Dzięki takiemu podejściu jest to rozwiązanie, które jest zarówno szybkie, jak i bardzo uniwersalne.
- A2C stanowi hybrydę dwóch niezależnych sieci neuronowych:
  - **Krytyk:** określa jak dobra była wykonana akcja,
  - **Aktor:** Bezpośrednio kontroluje jak aktor się zachowuje.
- Krytyk reprezentuje tu podejście bez strategii, natomiast aktor ze strategią.

## A2C

---

- Stosujemy takie rozwiązanie po to, by nie premiować złych ruchów z sekwencji, która zakończyła się sukcesem.
- W przeciwieństwie do algorytmów ze strategią, A2C dyskontuje nagrodę przy każdym ruchu, dzięki czemu unikamy powyższego problemu.
- By dokonać oceny akcji potrzebna jest nagroda, jaka została za nią przyznana. Niestety, w większości gier za dobrze wykonany ruch zwykle dostajemy nagrodę, dopiero po jakimś czasie.
- Dlatego w A2C używamy dodatkowej sieci neuronowej - krytyka - która ma przewidzieć jaką nagrodę uzyskamy za dany ruch w przyszłości

## Aktualizacja strategii - Krytyk

---

- Aktualizacja wag przebiega nieco inaczej niż w DQN. Różnica polega na wykorzystaniu funkcji  $V$  zamiast  $Q$ . Funkcja  $V(s)$  (reprezentowana przez sieć neuronową), zamiast oceniać wartość pojedynczej pary stan - akcja wystawia ocenę dla całego stanu  $s$ .
- Używanie wartości  $V$  zamiast  $Q$  poprawia stabilność uczenia. Nasz agent unika stanów które są jako całość mniej bezpieczne. Aktualizacja wag krytyka przebiega w następujący sposób:

$$\Delta w = \beta(r_t + \gamma V(s_{t+1}) - V(s_t))\nabla_w V_w(s_t)$$

- gdzie  $\beta$  jest tu współczynnikiem uczenia się krytyka (aktor i krytyk mają różne współczynniki uczenia)

## Aktualizacja strategii - Aktor (1/2)

---

- Do aktualizacji wag aktora nie będziemy używać bezpośrednio wartości zwracanych przez krytyka (czyli funkcji  $V$ ). Byłoby to rozwiązanie poprawne, niestety jednocześnie charakteryzujące się dość wysoką niestabilnością.
- Zamiast tego użyjemy funkcji  $A$ . Jej wzór ma postać:

$$A(s_t, s_{t+1}) = r_t + \gamma V(s_{t+1}) - V(s_t)$$

- Jak łatwo zauważyc jest to po prostu funkcja loss krytyka.
- Ostatecznie aktualizacja wag aktora przebiega natomiast zgodnie z poniższym wzorem:

$$\Delta\theta = \alpha[\nabla(\log \pi(s, a, \theta)) \cdot A(s_t, s_{t+1}) + c \nabla S^\pi(s_t)]$$

- gdzie:  $\pi$  strategia(aktor),  $\alpha$  współczynnik uczenia,  $A_w(s_t, s_{t+1})$  krytyk (pełniący rolę funkcji nagrody),  $c$  to współczynnik sterujący, jak duży wpływ na naukę ma entropia.

## Aktualizacja strategii - Aktor (2/2)

---

- **UWAGA:** Wzór aktualizacji jest niemal identyczny jak w wypadku strategii gradientowej.
- Różnica polega po prostu na tym, że zamiast faktycznie uzyskanej przez nas w epizodzie nagrody  $R(\tau)$  wykorzystujemy przewidywaną przyszłą nagrodę zwracaną przez krytyka  $A_w(s_t, s_{t+1})$ .

## Realizacja funkcji replay (1/2)

---

- Ponieważ w A2C używamy krytyka, który bazuje na DQN, również w tym algorytmie musimy zmierzyć się z podobnymi problemami
- Mimo, że problemy są podobne w A2C, stosuje nieco inne rozwiązanie. Zamiast zapisywać przebieg gry, a następnie przeprowadzać ją w losowej kolejności, używamy równoległego uczenia na kilku środowiskach.
- Robi się to w następujący krokach:
  - ① Agent wykonuje po jednym ruchu w każdym środowisku.
  - ② Dla każdego ruchu, w każdym środowisku obliczamy gradient.
  - ③ Z uzyskanych gradientów obliczamy gradient średni.
  - ④ Za pomocą gradientu średniego dokonujemy aktualizacji strategii.

## Realizacja funkcji replay (2/2)

---

- Innym sposobem jest aktualizacja strategii za pomocą dodatkowej sieci neuronowej.
- Jest ona odpowiedzialna za obliczanie nowych wag dla modelu bezpośrednio wykonującego akcje.
- Używamy jej, ponieważ może ona przetworzyć dane z kilku środowisk jednocześnie.
- Tego drugiego sposobu używa się w wypadku gdy obliczenia przeprowadzane są na GPU, ponieważ znaczco zwiększa to wtedy efektywność.

## A2C i A3C

---

- Istnieje inny bliźniaczy do **Advantage Actor Critic** algorytm: **Asynchronous Advantage Actor Critic(A3C)**.
- Różnią się one liczbą kopii agenta, których używamy do nauki jednocześnie. W wypadku A2C mamy zwyczajnie jednego agenta, który synchronicznie (równolegle) wykonuje ruchy we wszystkich środowiskach.
- A3C natomiast wykorzystuje dla każdego środowiska inną kopię agenta (pracowników). Dzięki temu gra w każdym środowiska może się rozgrywać w nieco innym tempie.
- Każdy z pracowników ma nieco inny zestaw wag w sieci neuronowej. Wagi różnią się tylko nieco, ponieważ po każdym ruchu aktualizuję swój zestaw wag, w oparciu o globalną strategię. Można tu zastosować analogię do serwisu github, w którym mamy centralne repozytorium aktualizowane co jakiś czas przez różnych developerów.

## Section 4

Proximal Policy Optimization - PPO

## Proximal Policy Optimization

---

- **Proximal Policy Optimization (PPO)** stanowi tzw. state of the art, czyli innymi słowy jest obecnie szczytowym osiągnięciem w zakresie uczenia ze wzmocnieniem
- To właśnie agent oparty o ten algorytm dokonał, w 2018 roku, ważnego przełomu, pokonując najlepszych zawodników w grę Dota 2.
- Algorytm ma architekturę podobną do tej znanej z A2C. Nadal mamy więc aktora, który wykonuje akcje, i krytyka, który ocenia jak była ona dobra.
- Głównym założeniem PPO jest unikanie zbyt dużych aktualizacji aktora. Dzięki temu proces nauki staje się bardziej stabilny.

## Funkcja zmiany

---

- By mierzyć, jak bardzo zmieniła się nasza strategia używamy funkcji  $r_t(\theta)$ , która ma następującą formę:

$$r_t(\theta) = \frac{\pi_\theta(s_t | a_t)}{\pi_{\theta old}(s_t | a_t)}$$

- $\pi_\theta(s_t | a_t)$  to w tym wypadku znana ze strategii gradientowej funkcja, zwracająca prawdopodobieństwo wykonania akcji  $a$  w stanie  $s$  (w programie reprezentuje ją sieć neuronową)
- $r_t(\theta)$  mówi więc po prostu jak bardzo zmieni się działanie sieci po dokonaniu aktualizacji.
- Celem PPO jest utrzymać  $r_t(\theta)$  pomiędzy  $1 - \epsilon$  i  $1 + \epsilon$  ( $\epsilon$  jest wartością z przedziału  $(0,1)$  i zależy od rodzaju problemu). Służy nam do tego funkcja clip.

## Funkcja clip

---

- Celem PPO jest utrzymać  $r_t(\theta)$  pomiędzy  $1 - \epsilon$  i  $1 + \epsilon$  ( $\epsilon$  jest wartością z przedziału  $(0,1)$  i zależy od rodzaju problemu). Służy nam do tego funkcja clip.
- Ma ona następującą postać:

$$clip(r_t(\theta), \epsilon) = \begin{cases} 1 - \epsilon & \text{gdy } r_t(\theta) < 1 - \epsilon \\ r_t(\theta) & \text{gdy } r_t(\theta) \in (1 - \epsilon, 1 + \epsilon) \\ 1 + \epsilon & \text{gdy } r_t(\theta) > 1 + \epsilon \end{cases}$$

## Funkcja loss

---

- Funkcja loss dla aktora PPO składa się z trzech części:
  - ➊ loss  $\pi$
  - ➋ loss krytyka
  - ➌ entropia

## Funkcja loss $\pi$ i loss krytyka

---

- **Loss  $\pi$  ma następującą postać:**

$$L^{clip} = E_t[\min r_t(\theta) \cdot A_t(s_t, s_{t+1}), clip(r_t(\theta), \epsilon) \cdot A_t(s_t, s_{t+1})]$$

- Wartość powyższej funkcji jest bardzo mocno zależna od  $A_t(s_t, s_{t+1})$
- Jeśli wykonany ruch był gorszy, od zakładanego potencjału stanu  $s$  to  $A_t(s_t, s_{t+1})$  jest ujemne. W przeciwnym wypadku przyjmuje ona wartość dodatnią.
- Gdy funkcja  $A_t(s_t, s_{t+1}) > 0$  oznacza to, że wykonana akcja była lepsza niż przewidywana. Niemożliwe jest więc wtedy, by nasza nowa strategia zmniejszała prawdopodobieństwo wykonania tejże akcji w przyszłości. Analogiczna sytuacja jest gdy  $A_t(s_t, s_{t+1}) < 0$
- **Loss krytyka ma postać analogiczną jak w A2C**, będzie więc to w praktyce wartość funkcji  $A_t(s_t, s_{t+1})$ , czyli:

$$L^V = r_t + \gamma V(s_{t+1}) - V(s_t)$$

## Funkcja loss entropii i loss całkowity

---

- **Ostatnim elementem funkcji loss jest entropia.** Mierzy ona jak bardzo nieprzewidywalny jest wynik naszych działań, gdy znajdziemy się w stanie  $s$ .
- Używa się jej by na początku nauki zwiększyć nieprzewidywalność zachowania agenta. Entropia mierzy jak bardzo nieprzewidywalne jest otrzymanie danego wyniku. Dlatego też premiowanie ruchów o wyższej entropii wprowadza element losowości.
- Pewna nieprzewidywalność na początku nauki jest pożądana w celu omijania nieskończonych pętli akcji. Entropię oznaczamy w następujący sposób:  $S[\pi](s_t)$
- **Ostateczny wzór na funkcję loss** będzie miał więc postać:

$$Loss = E_t[L^V(\theta) - c_1 \cdot L^{clip}(\theta) + c_2 \cdot S[\pi](s_t)]$$

- gdzie  $c_1$  i  $c_2$  to współczynniki, którymi można manipulować.

## PPO joint

---

- **PPO joint** jest lekko zmodyfikowaną wersją PPO.
- **Joint** oznacza, że algorytm wykorzystuje rozwiążanie opisane na slajdzie z funkcją replay dla A2C, czyli by zaktualizować strategię, w każdym kroku obliczamy gradient średni, uzyskany z kilku różnych rozgrywek (mogą się one odbywać równocześnie tak jak w A3C lub kolejno tak jak w A2C).

Dziękuję!