Sztuczna inteligencja

Uczenie ze wzmocnieniem:
Projekt i implementacja autonomicznego agenta gry Snake
wykorzystującego
Q-learning oraz sieci neuronowe

Autor:
Kacper Hyliński
kierunek Informatyka

Rzeszów, 2025

Wydział Elektrotechniki i Informatyki
Politechnika Rzeszowska im. Ignacego Łukasiewicza

Spis treści

1 Cel projektu

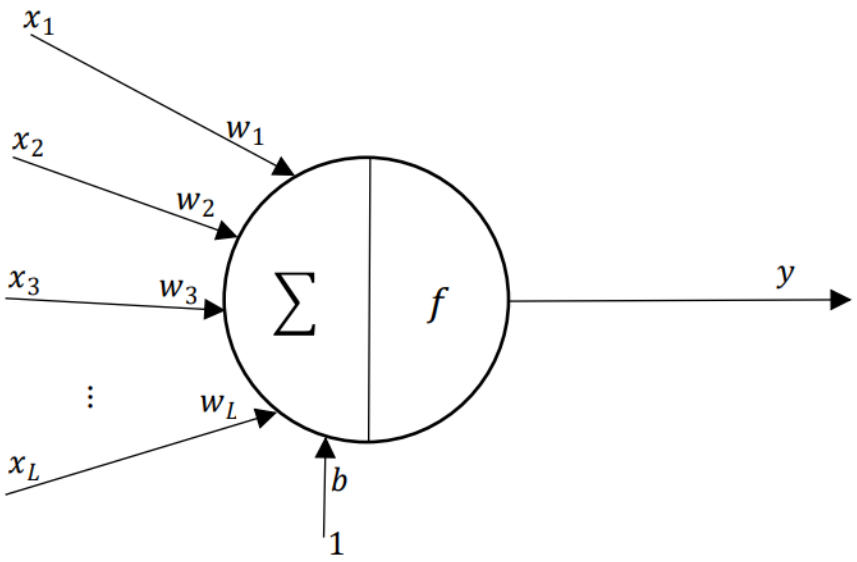
Celem niniejszego projektu jest opracowanie, implementacja oraz porównawcza analiza dwóch metod uczenia ze wzmocnieniem w zmodyfikowanym środowisku gry Snake, w którym agent ponosi karę za kolizję ze ścianami. W pierwszym podejściu zastosowano klasyczny algorytm Q-learning, natomiast w drugim – jego rozszerzenie o głęboką sieć neuronową (Deep Q-Network, DQN).

W ramach projektu zostanie utworzone środowisko symulacyjne, w którym agent będzie zobligowany do identyfikacji kluczowych elementów otoczenia, takich jak obszary zagrożenia oraz źródła pożywienia, a następnie do podejmowania decyzji na podstawie obserwowanych stanów. Ponadto zaimplementowany zostanie wariant Double DQN, a proces uczenia zostanie przyspieszony poprzez integrację technologii CUDA, z zapewnieniem możliwości alternatywnego przeprowadzenia treningu na procesorze centralnym w przypadku braku wsparcia GPU.

Dla optymalizacji przebiegu treningu opracowany zostanie również uproszczony interfejs gry, pozbawiony renderowania graficznego na ekranie, co pozwoli na znaczące zwiększenie efektywności procesu uczenia.

2 Część teoretyczna

2.1 Model neuronu



Rysunek 1: Model Neuronu [1].

$\boldsymbol{x_1 - x_L}$ - sygnał wejściowy,
$\boldsymbol{w_1 - w_L}$ - współczynnik wagowy,
$\boldsymbol{b}$ - bias,
$\sum$ - sumator,
$\boldsymbol{f}$ - funkcja aktywująca,
$\boldsymbol{y}$ - wartość wyjściowa

$$y = f\Big(\sum_{j=1}^{L} w_j \cdot x_j + b\Big) \tag{1}$$

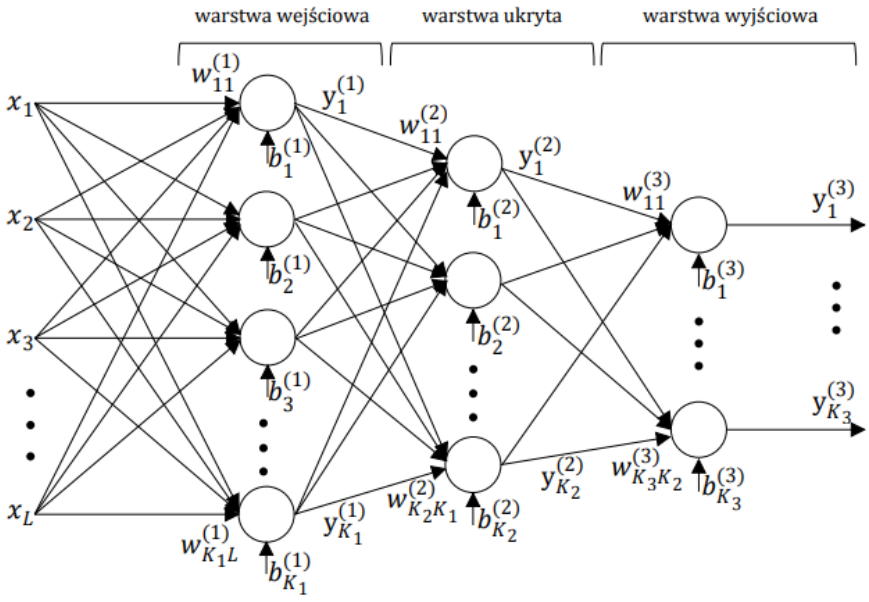
Symbol x_j oznacza j-ty sygnał wejściowy ($j = 1, 2, \ldots, L$), natomiast w_j odpowiada przypisanej mu wadze.

Sumę ważoną sygnałów wejściowych wraz z wartością przesunięcia (biasu) określa się mianem *łącznego pobudzenia neuronu*, które w dalszej części oznaczane będzie symbolem z:

$$z = \sum_{j=1}^{L} w_j \cdot x_j + b. \tag{2}$$

2.2 Sieć neuronowa wielowarstwowa (MLP)

Taką sieć nazywa się trójwarstwową. Występują tu połączenia pomiędzy warstwami neuronów typu każdy z każdym. Sygnały wejściowe podawane są do warstwy wejściowej neuronów, których wyjścia stanowią sygnały źródłowe dla kolejnej warstwy. Można wykazać, że sieć trójwarstwowa nieliniowa jest w stanie odwzorować praktycznie dowolne odwzorowanie nieliniowe.



Rysunek 2: Sieć jednokierunkowa wielowarstwowa [2].

Każda warstwa neuronów posiada swoją macierz wag $\mathbf{w}$, wektor przesunięć $\mathbf{b}$, funkcję aktywacji f i wektor sygnałów wyjściowych $\mathbf{y}$. Aby je rozróżniać w dalszych rozważaniach, do każdej z wielkości dodano numer warstwy, której dotyczą. Na przykład dla warstwy drugiej (ukrytej) macierz wag oznaczana będzie symbolem $\mathbf{w}^{(2)}$. Działanie każdej z warstw można rozpatrywać oddzielnie. I tak np. warstwa druga posiada: $L = K_1$ sygnałów wejściowych, $K = K_2$ neuronów i macierz wag $\mathbf{w} = \mathbf{w}^{(2)}$ o rozmiarach $K_2 \times K_1$. Wejściem warstwy drugiej jest wyjście warstwy pierwszej $\mathbf{x} = \mathbf{y}^{(1)}$, a wyjściem $\mathbf{y} = \mathbf{y}^{(2)}$. Działanie poszczególnych warstw dane jest przez

$$y^{(1)} = f^{(1)}\left(w^{(1)}x + b^{(1)}\right), \tag{3}$$

$$y^{(2)} = f^{(2)}\left(w^{(2)}y^{(1)} + b^{(2)}\right), \tag{4}$$

$$y^{(3)} = f^{(3)}\left(w^{(3)}y^{(2)} + b^{(3)}\right). \tag{5}$$

Działanie całej sieci można więc opisać jako

$$\mathbf{y}^{(3)} = f^{(3)}\left(\mathbf{w}^{(3)}f^{(2)}\left(\mathbf{w}^{(2)}f^{(1)}\left(\mathbf{w}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)}\right) + \mathbf{b}^{(3)}\right). \tag{6}$$

[2]

W projekcie Snake AI do aproksymacji funkcji wartości $Q(s, a)$ wykorzystujemy wielowarstwową sieć neuronową (MLP) o strukturze:

- Warstwa wejściowa: wymiar wektora stanu $d = 11$ cech,
- trzy warstwy ukryte:
  - pierwsza z $H_1 = 256$ neuronów,
  - druga z $H_2 = 256$ neuronów,

– trzecia z $H_3 = 128$ neuronów,

- Warstwa wyjściowa: liczba wyjść równa liczbie akcji $|A| = 3$.

2.3 MLP w projekcie Snake AI

W projekcie Snake AI do aproksymacji funkcji wartości Q(s,a) wykorzystujemy wielowarstwową sieć neuronową (MLP) składającą się z warstwy wejściowej, dwóch warstw ukrytych i warstwy wyjściowej.

$$\mathbf{y}^1 = f^1(\mathbf{z}^1) = f^1(\mathbf{w}^1\mathbf{x} + \mathbf{b}^1) \tag{7}$$

$$\mathbf{y}^2 = f^2(\mathbf{z}^2) = f^2(\mathbf{w}^2\mathbf{y}^1 + \mathbf{b}^2) \tag{8}$$

$$\mathbf{y}^3 = f^3(\mathbf{z}^3) = f^3(\mathbf{w}^3\mathbf{y}^2 + \mathbf{b}^3) \tag{9}$$

$$\mathbf{y}^4 = f^4(\mathbf{z}^4) = f^4(\mathbf{w}^4\mathbf{y}^3 + \mathbf{b}^4) \tag{10}$$

Gdzie f^1, f^2, f^3 to funkcje Leaky ReLU, a f^4 to funkcja liniowa.
wzór opisujący działanie całej sieci jako złożenie funkcji:

$$\mathbf{y}^4 = f^4\left(\mathbf{w}^4 f^3\left(\mathbf{w}^3 f^2\left(\mathbf{w}^2 f^1(\mathbf{w}^1\mathbf{x} + \mathbf{b}^1) + \mathbf{b}^2\right) + \mathbf{b}^3\right) + \mathbf{b}^4\right) \tag{11}$$

2.4 Funkcje aktywacji

W ramach niniejszego projektu, którego celem jest implementacja agenta Deep Q-Network (DQN) w środowisku gry Snake, zastosowano różne funkcje aktywacji. Ich dobór został przeprowadzony w sposób celowy i dostosowany do charakterystyki poszczególnych warstw sieci neuronowej.

Leaky ReLU – funkcja aktywacji, która wprowadza niewielki współczynnik nachylenia dla wartości ujemnych, co pozwala na uniknięcie problemu zanikania gradientu i „martwych neuronów". Funkcja ta jest zdefiniowana jako:

$$f(x) = \begin{cases} x & \text{jeśli } x > 0 \\ \alpha x & \text{jeśli } x \leq 0 \end{cases} \tag{12}$$

lub w bardziej zwartej postaci:

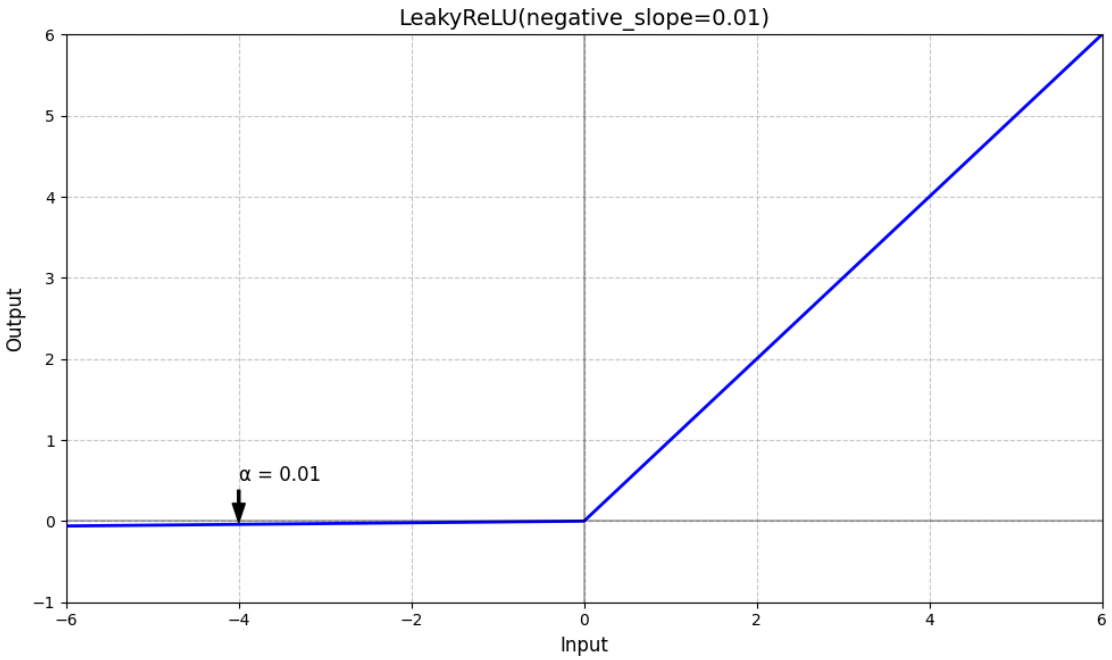
$$f(x) = \max(\alpha x, x),$$

gdzie α to mała wartość (w naszym projekcie 0,01), która kontroluje nachylenie dla wartości ujemnych. W sieci DQN używamy Leaky ReLU we wszystkich warstwach ukrytych.

Funkcja liniowa – funkcja aktywacji używana w warstwie wyjściowej sieci DQN, która nie wprowadza żadnej nieliniowości, co pozwala na nieograniczony zakres wartości wyjściowych. Jest zdefiniowana jako:

$$f(x) = x \tag{13}$$

Funkcja ta jest szczególnie przydatna w warstwie wyjściowej sieci aproksymującej funkcję Q, gdzie wyjścia mogą przyjmować dowolne wartości rzeczywiste.



Rysunek 3: Wykres LeakyRelu

Funkcja liniowa w warstwie wyjściowej sieci DQN została wybrana ze względu na następujące właściwości i zalety:

1. **Nieograniczony zakres wartości wyjściowych** – w kontekście aproksymacji funkcji wartości Q, wyjścia sieci mogą przyjmować dowolne wartości rzeczywiste. Funkcja liniowa nie wprowadza żadnych ograniczeń zakresu wartości wyjściowych, co jest kluczowe dla poprawnego przewidywania wartości funkcji Q [3].

2. **Zgodność z teorią DQN** – w oryginalnej publikacji opisującej algorytm Deep Q-Network autorstwa Mnih et al. (2015), również zastosowano liniową funkcję aktywacji w warstwie wyjściowej, co potwierdza zasadność wyboru tego rozwiązania [3].

3. **Rozwiązanie problemu "umierających neuronów":** Standardowa funkcja ReLU zwraca 0 dla wszystkich wartości ujemnych, co może prowadzić do "umierania neuronów", kiedy neuron zawsze daje wartość 0 na wyjściu. LeakyReLU rozwiązuje ten problem, pozwalając na przepływ małego gradientu dla wartości ujemnych.

2.5 Wprowadzenie do uczenia ze wzmocnieniem i MDP

Uczenie ze wzmocnieniem (ang. *Reinforcement Learning*, RL) to rodzaj uczenia maszynowego, w którym agent uczy się podejmować sekwencje decyzji poprzez interakcję ze środowiskiem. Po każdej akcji podjętej w stanie środowiska agent otrzymuje *nagrodę* – skalarny feedback – i przechodzi do nowego stanu. Celem agenta jest opracowanie *strategii (polityki)* wyboru akcji maksymalizującej skumulowaną nagrodę w długim horyzoncie czasowym.

Formalnie środowisko modeluje się jako *proces decyzyjny Markowa* (MDP), zdefiniowany jako krotka

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R),$$

gdzie:

- $\mathcal{S}$ – zbiór stanów,
- $\mathcal{A}$ – zbiór akcji,
- $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1], \quad P(s' \mid s, a) = \Pr(s_{t+1} = s' \mid s_t = s,\, a_t = a)$ – funkcja przejścia definiująca rozkład prawdopodobieństwa kolejnego stanu,
- $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}, \quad R(s, a) = \mathbb{E}[r_{t+1} \mid s_t = s,\, a_t = a]$ – funkcja nagród.

Jeśli w chwili t agent znajduje się w stanie s_t i wybierze akcję a_t, to z prawdopodobieństwem $P(s_{t+1} = s' \mid s_t, a_t)$ przejdzie do stanu $s_{t+1} = s'$ i otrzyma nagrodę $r_{t+1} = R(s_t, a_t)$. Proces ten powtarza się iteracyjnie, tworząc ścieżkę stanów, akcji i nagród. Sumaryczna zdyskontowana nagroda od chwili t jest zwykle definiowana jako

$$R_t \;=\; \sum_{k=0}^{\infty} \gamma^k \, r_{t+k+1},$$

gdzie $\gamma \in [0, 1]$ to współczynnik dyskontowania.

gdzie $0 \leq \gamma < 1$ to współczynnik dyskontowania określający ważność przyszłych nagród.[1] Formalnym celem uczenia ze wzmocnieniem jest znalezienie strategii π^*, która maksymalizuje wartość oczekiwaną powyższej sumy nagród.

Funkcja wartości

Kluczową koncepcją w uczeniu ze wzmocznieniem jest *funkcja wartości*. Dla danej strategii π definiujemy wartość stanu

$$V^\pi(s) = \mathbb{E}\Big[\sum_{k=0}^{\infty} \gamma^k \, r_{t+k+1} \;\Big|\; s_t = s\Big].$$

Analogicznie *funkcję wartości akcji* określa się jako

$$Q^\pi(s, a) = \mathbb{E}\Big[r_{t+1} + \gamma \, Q^\pi\Big(s_{t+1}, \pi(s_{t+1})\Big) \;\Big|\; s_t = s,\, a_t = a\Big].$$

Równanie to (zwane *równaniem Bellmana* dla Q^π) wyraża zależność wartości akcji od natychmiastowej nagrody oraz wartości przyszłego stanu.

[1] Niższa wartość γ sprawia, że agent przywiązuje większą wagę do nagród natychmiastowych niż odległych w czasie.

Dla strategii optymalnej π^*, maksymalizującej zdyskontowaną nagrodę, zachodzi *równanie optymalności Bellmana*:

$$Q^*(s,a) = r(s,a) + \gamma \sum_{s'} P(s' \mid s,a) \max_{a'} Q^*(s',a').$$

gdzie $Q^*(s,a)$ oznacza optymalną funkcję wartości akcji. Innymi słowy, zakładając znajomość pełnego modelu środowiska (P,R), Q^* spełnia układ równań, z którego (przynajmniej teoretycznie) da się wyznaczyć optymalne wartości i strategię:

$$\pi^*(s) \;=\; \arg\max_a Q^*(s,a).$$

W praktyce jednak model środowiska często nie jest znany lub zbyt złożony, a przestrzeń stanów ogromna, dlatego stosujemy metody iteracyjne i przybliżone, aby oszacować Q^*.

2.6 Klasyczny algorytm Q-learning

Q-learning jest jedną z najpopularniejszych metod uczenia ze wzmocnieniem typu *off-policy* wykorzystujących różnice czasowe. Algorytm ten potrafi uczyć się bezpośrednio na podstawie doświadczeń z interakcji, nie znając apriori funkcji przejścia ani nagród. Zbieżność Q-learningu do optymalnych wartości Q^* została udowodniona w przypadku dyskretnych przestrzeni stanów i akcji przy odpowiednich założeniach (m.in. że każda para stan–akcja jest odwiedzana nieskończenie często, a współczynnik uczenia maleje w czasie).

Algorytm utrzymuje estymację funkcji wartości akcji $Q(s,a)$ w postaci tabelarycznej (tablicy wartości dla każdej pary stan–akcja). Początkowo przypisuje się jej pewne wartości (np. losowe lub zerowe). Następnie agent rozgrywa serię epizodów interakcji ze środowiskiem. W każdym kroku epizodu ze stanu s_t wybierana jest akcja a_t zgodnie z aktualną strategią eksploracyjno–eksploatacyjną. Najczęściej stosuje się strategię ε-greedy, w której z prawdopodobieństwem $1-\varepsilon$ wybierana jest akcja optymalna według bieżących wartości Q, a z prawdopodobieństwem ε agent eksploruje, wybierając akcję losową. Pozwala to uniknąć „uwięzienia” w polityce zachłannej opartej na niedokładnych wartościach i zapewnia odwiedzanie różnych stanów. Parametr ε (np. początkowo 1) często zmniejsza się w miarę uczenia, aby eksploracja malała z czasem na rzecz eksploatacji zdobytej wiedzy.

Po podjęciu akcji a_t w stanie s_t i otrzymaniu nagrody r_t oraz obserwacji nowego stanu s_{t+1} następuje aktualizacja oceny $Q(s_t,a_t)$. Klasyczny Q-learning wykorzystuje w tym celu błąd przewidywania, tzw. błąd TD (temporal difference), definiowany jako:

$$\delta = r + \gamma \max_{a'} Q(s',a') \;-\; Q(s,a).$$

Błąd TD wyraża różnicę między aktualną estymacją wartości $Q(s,a)$ a wartością docelową

$$y = r + \gamma \max_{a'} Q(s',a'),$$

opartą na bieżących ocenach stanu następnego. Następnie wartość $Q(s,a)$ jest przesuwana w kierunku wartości docelowej o krok proporcjonalny do współczynnika uczenia α (zwykle $\alpha \in [0,1]$):

$$Q(s,a) \;\leftarrow\; Q(s,a) + \alpha\Big[r + \gamma \max_{a'} Q(s',a') \;-\; Q(s,a)\Big].$$

Powyższa reguła aktualizacji jest implementacją iteracji Bellmana na podstawie pojedynczych doświadczeń. Intuicyjnie, jeśli zaobserwowana natychmiastowa nagroda r wraz z najlepszą przewidywaną wartością przyszłą $\max_{a'} Q(s', a')$ przewyższa dotychczasową ocenę $Q(s, a)$, to $Q(s, a)$ zostanie zwiększone (i odwrotnie – przeszacowane wartości są zmniejszane). Współczynnik α reguluje szybkość uczenia: małe α sprawia, że wartości zmieniają się wolniej (co uśrednia wiele doświadczeń), natomiast $\alpha = 1$ oznacza całkowite zastąpienie starej wartości nowym jednorazowym oszacowaniem. Dzięki niezależności aktualizacji od konkretnej strategii wyboru akcji (algorytm off-policy), Q-learning w teorii konwerguje do Q^* nawet jeśli podczas uczenia agent nie zawsze podąża strategią zachłanną.

Algorytm Q-learning stopniowo poprawia przybliżenia wartości akcji. Z czasem, gdy $Q(s, a)$ zbliża się do wartości optymalnych, strategia zachłanna względem tych wartości staje się strategią optymalną π^*. W praktyce tablicowy Q-learning jest stosowany tylko dla stosunkowo niewielkich i dyskretnych przestrzeni stanów, ponieważ przechowywanie i aktualizacja tablicy $Q(s, a)$ w większych problemach (np. gdy stanem jest obraz Atari) jest niewykonalne. Rozwiązaniem problemu skalowalności jest zastąpienie tablicy Q funkcją przybliżającą – tu z pomocą przychodzi sieć neuronowa*.

2.7 Deep Q-Network (DQN)

Deep Q-Network (DQN) to algorytm, który łączy klasyczny Q-learning z możliwościami głębokich sieci neuronowych do aproksymacji funkcji wartości Q. Fundamentalna koncepcja polega na zastąpieniu tablicy $Q(s, a)$ przez sieć neuronową $Q(s, a; \theta)$ parametryzowaną wektorem θ. Sieć przyjmuje na wejściu reprezentację stanu s i generuje na wyjściu estymowane wartości $Q(s, a)$ dla zbioru dostępnych akcji. Dzięki temu podejściu algorytm uzyskuje zdolność skalowania do rozległych (w tym ciągłych) przestrzeni stanów, realizując generalizację – tj. aproksymując podobne wartości dla stanów wykazujących wspólne cechy, co pozostaje poza zasięgiem tablicowego Q-learningu. W implementacji DQN przyjęto strukturę czterowarstwowej sieci neuronowej typu MLP (Multi-Layer Perceptron).

Metodologia treningu

Proces uczenia sieci $Q(s, a; \theta)$ opiera się na minimalizacji różnicy między predykcjami a wartościami docelowymi obliczanymi na podstawie ewaluacji stanu środowiska (analogicznie do procedury w tablicowym Q-learningu). W każdej iteracji procesu uczenia analizowana jest próbka doświadczenia $(s, a, r, s', \mathit{done})$, na podstawie której wyznaczana jest wartość docelowa:

$$y = r + \gamma \max_{a'} Q(s', a'; \theta^-) \cdot (1 - \mathit{done}),$$

gdzie θ^- oznacza parametry sieci docelowej, a *done* stanowi indykator stanu terminalnego (tj. zakończenia epizodu).

W opisywanej implementacji zastosowano modyfikację algorytmu DQN określaną jako Double DQN, która adresuje problem systematycznego przeszacowania wartości Q. W standardowym DQN ta sama sieć wykorzystywana jest zarówno do

selekcji, jak i ewaluacji akcji, co może skutkować zawyżaniem wartości Q. W paradygmacie Double DQN sieć główna dokonuje selekcji akcji, natomiast sieć docelowa przeprowadza jej ewaluację:

$$y = r + \gamma \cdot Q(s', \arg\max_{a'} Q(s', a'; \theta); \theta^-) \cdot (1 - done)$$

Następnie definiowana jest funkcja straty. W implementacji wykorzystano funkcję Hubera (operacyjnie zaimplementowaną jako SmoothL1Loss w bibliotece PyTorch), która charakteryzuje się zwiększoną odpornością na wartości odstające w porównaniu do błędu średniokwadratowego:

$$\mathcal{L}_\delta(\theta) = \begin{cases} \frac{1}{2}\Big(y - Q(s,a;\theta)\Big)^2, & \text{jeśli } |y - Q(s,a;\theta)| \leq \delta, \\ \delta\Big(|y - Q(s,a;\theta)| - \frac{1}{2}\delta\Big), & \text{jeśli } |y - Q(s,a;\theta)| > \delta. \end{cases}$$

gdzie δ stanowi parametr delimitujący przejście między kwadratową a liniową charakterystyką funkcji straty (przyjęto $\delta = 1$). Funkcja Hubera zapewnia, że dla znacznych odchyleń model nie generuje nieograniczonych wartości gradientu (jak ma to miejsce w funkcji MSE), co zabezpiecza przed niestabilnością uczenia wywołaną zjawiskiem eksplozji gradientów.

W implementacji wykorzystano optymalizator Adam (w przypadku obliczeń na CPU) lub AdamW (w przypadku obliczeń na GPU) ze współczynnikiem uczenia $\alpha = 0.0003$. AdamW wprowadza regularyzację typu Weight Decay, co przyczynia się do zwiększenia stabilności procesu uczenia na architekturach GPU. Dodatkowo zastosowano technikę przycinania gradientów (gradient clipping) do normy jednostkowej, co stanowi kolejny mechanizm stabilizujący proces treningu.

Mechanizm Experience Replay

Kluczowym komponentem architektury DQN jest mechanizm Experience Replay, który gromadzi doświadczenia agenta w buforze pamięci i realizuje losowe próbkowanie podczas procesu uczenia. W prezentowanej implementacji zastosowano bufor doświadczeń o pojemności $5 \cdot 10^4$ przejść. W każdej iteracji uczenia losowany jest mini-batch o liczności 256 (w przypadku obliczeń na GPU) lub 64 (w przypadku obliczeń na CPU) próbek, co redukuje korelację temporalną między danymi treningowymi i stabilizuje proces uczenia.

Dodatkowo zaimplementowano optymalizację polegającą na asynchronicznym, wyprzedzającym pobieraniu batcha w odrębnym wątku obliczeniowym, co zwiększa efektywność czasową treningu. Bufor pamięci został zrealizowany jako struktura deque z ograniczoną długością maksymalną, co zapewnia automatyczną eliminację najstarszych doświadczeń po osiągnięciu założonej pojemności.

Sieć docelowa

W celu dalszego zwiększenia stabilności procesu uczenia, architektura DQN wykorzystuje dedykowaną sieć docelową (ang. Target Network) z parametrami θ^-, która podlega periodycznej aktualizacji poprzez transfer parametrów z sieci głównej w określonych interwałach czasowych (w implementacji co 10^3 kroków). Takie podejście zapewnia, że wartości docelowe wykorzystywane w procesie uczenia charakteryzują się odpowiednią stałością temporalną, co istotnie stabilizuje proces uczenia.

Techniki optymalizacji obliczeniowej

W prezentowanej implementacji algorytmu DQN wprowadzono szereg technik optymalizacji obliczeniowej celem akceleracji procesu treningu:

- Hybrydowy paradygmat obliczeń CPU/GPU, gdzie symulacja środowiska realizowana jest na CPU (z możliwością równoległej symulacji wielu instancji środowiska), natomiast operacje uczenia wykonywane są na GPU
- Adaptacyjny dobór precyzji reprezentacji liczb zmiennoprzecinkowych (float16/float32) w zależności od specyfikacji dostępnej architektury GPU
- Dynamiczna parametryzacja rozmiaru batcha w dostosowaniu do dostępnych zasobów obliczeniowych
- Paralelizacja procesu akwizycji doświadczeń z wielu współbieżnych instancji środowiska

Zastosowane techniki optymalizacji umożliwiają znaczącą akcelerację procesu uczenia, w szczególności na systemach wyposażonych w architekturę GPU.

3 Analiza Danych

4 Skrypt programu

5 Eksperymenty

6 Podsumowanie i wnioski

Literatura

[1] Zajdel R., *Sztuczna inteligencja / Metody sztucznej inteligencji w sterowaniu, Laboratorium, Ćwiczenie 4: Model Neuronu*, https://materialy.prz-rzeszow.pl/pracownik/pliki/34/%C4%86WICZENIE%2061.pdf.

[2] Zajdel R., *Sztuczna inteligencja / Metody sztucznej inteligencji w sterowaniu, Laboratorium, Ćwiczenie 7: Sieć neuronowa jednokierunkowa wielowarstwowa*, https://materialy.prz-rzeszow.pl/pracownik/pliki/34/ĄĘWICZENIE%209.pdf.

[3] Mnih V., Kavukcuoglu K., Silver D., Graves A., Antonoglou I., Wierstra D., Riedmiller M.,
Human-level control through deep reinforcement learning,
Nature. https://www.nature.com/articles/nature14236.

[4] R. Sutton, A. Barto, Reinforcement Learning: An Introduction, 2nd ed., 2018. https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf

[5] Wydział Elektroniki, Telekomunikacji i Informatyki, Politechnika Gdańska, *Uczenie ze wzmocnieniem – materiały wykładowe do kursu „Metody sztucznej inteligencji w sterowaniu”*, 2017, https://eti.pg.edu.pl/documents/176468/53881396/UW_MSU_2017.pdf.

[6] Leaky ReLU, *Dokumentacja*, https://docs.pytorch.org/docs/stable/generated/torch.nn.LeakyReLU.html

[7] Pytorch, *Dokumentacja*, https://pytorch.org/docs/stable/index.html