Asynchroniczny algorytm Advantage Actor–Critic (A3C) dla gry *Pong*

Sprawozdanie z projektu – przedmiot „Algorytmy optymalizacji”

Adrian Galik Nr albumu: 268864

24 czerwca 2025

Spis treści

1 Wprowadzenie

Gry wideo z rodziny Atari stały się w ostatniej dekadzie *benchmarkiem* dla algorytmów uczenia ze wzmocnieniem (RL), ponieważ łączą dużą przestrzeń stanów (surowe piksele) z niewielką liczbą dyskretnych akcji oraz wyraźnie zdefiniowaną funkcją nagrody. Celem niniejszego projektu jest zbudowanie i przeanalizowanie **asynchronicznego algorytmu Advantage Actor–Critic (A3C)** oraz jego **synchronicznego odpowiednika A2C**, a następnie porównanie obu metod na przykładzie gry `PongNoFrameskip-v4`.

W szczególności skupiamy się na aspektach **optymalizacji**:

- *Równoległość danych.* A3C wykorzystuje wiele procesów, które równolegle symulują środowisko i asynchronicznie aktualizują wspólne parametry sieci, podczas gdy A2C sumuje gradienty *synchronicznie* po każdym kroku uczenia.
- *Efektywność obliczeń.* Analizujemy, jak liczba procesów i rozmiar mini-batcha wpływają na przepustowość danych (klatki/s) oraz szybkość zbieżności nagrody.
- *Stabilność uczenia.* Badamy wpływ entropii, klipu gradientu i strategii n-step na oscylacje funkcji wartości i polityki.

2 Podstawy teoretyczne

W niniejszym rozdziale streszczamy niezbędne podstawy teorii uczenia ze wzmocnieniem (RL) w ujęciu książki Lapana [**?**] oraz oryginalnego artykułu A3C [**?**]. Skupiamy się na elementach istotnych z punktu widzenia *optymalizacji asynchronicznej*.

2.1 Model formalny RL

Środowisko opisujemy procesem MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, P, r, \gamma \rangle$, gdzie $\mathcal{S}$ i $\mathcal{A}$ to odpowiednio przestrzeń stanów i akcji, $P(s'|s,a)$ – funkcja przejścia, a $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ – natychmiastowa nagroda. Agent, obserwując stan s_t, wybiera akcję $a_t \sim \pi_\theta(\cdot|s_t)$ (*polityka*) otrzymuje nagrodę r_t i przechodzi do stanu s_{t+1}. Celem jest maksymalizacja zdyskontowanej sumy nagród

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}, \qquad \gamma \in (0,1).$$

Wartość stanu i wartość akcji definiujemy następująco:

$$V^\pi(s) = \mathbb{E}_\pi[R_t \mid s_t = s], \qquad Q^\pi(s,a) = \mathbb{E}_\pi[R_t \mid s_t = s, a_t = a].$$

2.2 Gradient polityki i aktor–krytyk

Twierdzenie o gradiencie polityki pozwala zapisać pochodną funkcji celu $J(\theta) = \mathbb{E}_{s\sim d^{\pi_\theta}} \mathbb{E}_{a\sim\pi_\theta}[R_t]$ w postaci:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s,a\sim\pi_\theta}\Big[\nabla_\theta \log \pi_\theta(a|s)\, Q^{\pi_\theta}(s,a)\Big]. \tag{1}$$

Aktor–krytyk wprowadza aproksymację funkcji Q za pomocą krytyka $V_w(s)$ oraz *advantage* $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$, co zmniejsza wariancję estymatora gradientu.

2.3 n-krokowa aktualizacja

Zamiast pojedynczego kroku TD, A3C wykorzystuje n-krokowy zwrot :

$$R_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V_w(s_{t+n}),$$

na podstawie którego definiujemy

$$A_t^{(n)} = R_t^{(n)} - V_w(s_t).$$

Wspólny gradient dla parametrów aktora i krytyka wynosi

$$g_\theta = \nabla_\theta \log \pi_\theta(a_t|s_t)\, A_t^{(n)}, \tag{2}$$

$$g_w = \nabla_w \tfrac{1}{2}\Big(R_t^{(n)} - V_w(s_t)\Big)^2. \tag{3}$$

2.4 Synchroniczny A2C a asynchroniczny A3C

A2C (Advantage Actor–Critic). Kilka procesów zbiera dane równolegle, lecz po każdym mini-batchu *blokująco* agreguje gradienty i wykonuje wspólny krok optymalizatora („data parallel – synchronous").

A3C (Asynchronous A^{3}C). Każdy worker oblicza gradient zaraz po zebraniu własnego mini-batcha i *natychmiast* aktualizuje wspólne wagi w pamięci RAM Skutkuje to brakiem bariery synchronizacji i prawie liniowym wzrostem przepustowości przy zwiększaniu liczby rdzeni.

Równoległość na poziomie danych vs gradientów. W naszej implementacji stosujemy wyłącznie *data parallelism*: każdy proces:

1. symuluje środowisko na CPU,
2. liczy gradient na CPU,
3. wysyła gradient do głównego procesu (GPU) lub bezpośrednio modyfikuje współdzielone parametry.

Nie używamy równoległości gradientów (*model parallelism*), gdyż dysponujemy pojedynczą kartą RTX 2080; kopiowanie tensora między kilkoma GPU byłoby bardziej kosztowne niż zysk z równoległości.

2.5 Funkcja straty

Całkowita funkcja straty używana w eksperymencie, zgodnie z , to

$$\mathcal{L} = -\,\mathbb{E}\Big[\log \pi_\theta(a_t|s_t)\, A_t^{(n)}\Big] + c_v\, \mathbb{E}\Big[\Big(R_t^{(n)} - V_w(s_t)\Big)^2\Big] - \beta\, \mathcal{H}(\pi_\theta(\cdot|s_t))\,, \tag{4}$$

gdzie c_v to współczynnik części wartości (przyjmujemy 1), zaś β (=`ENTROPY_BETA`) kontroluje siłę regularizacji entropijnej $\mathcal{H}$.

Zestaw wzorów i założeń przedstawiony w tym rozdziale stanowi podstawę implementacji opisanej w rozdziale 4 oraz eksperymentów porównujących wersję synchroniczną (A2C) i asynchroniczną (A3C).

3 Środowisko `PongNoFrameskip-v4`

`PongNoFrameskip-v4` pochodzi z pakietu *Arcade Learning Environment* (ALE) i stanowi klasyczny benchmark dla algorytmów RL [**?**, **?**]. Jest to dwuwymiarowa gra ping-pong, w której agent steruje paletką po lewej stronie ekranu, a rywal (sterowany przez silnik gry) - po prawej. Agent otrzymuje nagrodę $+1$ po zdobyciu punktu, -1 po jego stracie, w przeciwnym razie 0.

3.1 Surowa przestrzeń obserwacji i akcji

- **Obserwacja** – pojedyncza klatka RGB o rozdzielczości $210 \times 160 \times 3$ px (typ `uint8`, zakres $[0, 255]$).
- **Akcje** – dyskretny zbiór $\mathcal{A} = \{0, 1, 2, 3, 4, 5\}$, gdzie zgodnie z ALE:

0	`NOOP`
1	`FIRE`
2	`UP`
3	`RIGHT`
4	`LEFT`
5	`DOWN`

- **Epilog gry** – mecz kończy się, gdy jedna ze stron zdobędzie 21 punktów (maks. zwrot ± 21).

3.2 Pipeline przetwarzania danych

Aby zmniejszyć wymiar wejścia i ustabilizować uczenie, stosujemy standardowy zestaw wrapperów zalecany w [**?**, **?**]:

1. **`MaxAndSkipEnv` (skip=4)** – wykonuje tę samą akcję przez cztery klatki i zwraca maksimum piksel-po-pikselu z dwóch ostatnich; zmniejsza migotanie i przyspiesza symulację $\approx 4\times$.
2. **`FireResetEnv`** – po resecie wysyła akcje `FIRE`, aby rozpocząć grę.
3. **`ProcessFrame84`** – konwersja do skali szarości, resize do 84×84px.
4. **`ImageToPyTorch`** – zmiana kolejności kanałów z HWC na CHW (wymóg PyTorch).
5. **`FrameStack` (k=4)** – konkatenacja czterech kolejnych klatek $\Rightarrow$ obserwacja $(4, 84, 84)$, pozwala sieci odtworzyć prędkość piłki.
6. `(opc.)` **`ScaledFloatFrame`** – dzieli piksele przez 255, zamienia `uint8` $\rightarrow$ `float32`.

Efektem jest końcowy tensor

$$s_t \in \mathbb{R}^{4\times 84\times 84}, \quad s_t \in [0, 1]. \tag{5}$$

3.3 Metryki używane w eksperymentach

- **Reward per game** – suma nagród od rozpoczęcia do końca meczu; wartość docelowa ≥ 18 punktów (`REWARD_BOUND` w kodzie).
- **Frames per second (FPS)** – przepustowość danych (kl./s) liczona jako liczba przetworzonych klatek/sekundę na CPU. Służy do porównania A2C (synchron.) i A3C (asynchron.).

Rysunek 1: Schemat przetwarzania klatki w pipeline'ie A3C.

Podsumowanie

Przedstawiony pipeline zmniejsza wymiar wejścia ponad 10-krotnie i eliminuje zbędne informacje kolorystyczne, pozwalając sieci konwolucyjnej skupić się na ruchu piłki i paletki. W dalszych rozdziałach wykorzystujemy identyczny preprocessing zarówno dla A2C, jak i A3C, aby porównanie było rzetelne.

4 Algorytm A3C

Algorytm **Asynchronous Advantage Actor–Critic** (A3C) zaproponowany przez Mniha i wsp. [**?**] łączy trzy idee:

1. gradient polityki z funkcją advantage (aktor–krytyk),
2. estymację n-krokową zwrotu,
3. równoległe, *asynchroniczne* aktualizacje wspólnych wag.

4.1 Architektura sieci

Zgodnie z [**?**, **?**] używamy architektury widocznej na rys. 2:

- część konwolucyjna wspólna dla polityki i krytyka: $\text{Conv}_{8,4,32} \to \text{Conv}_{4,2,64} \to \text{Conv}_{3,1,64} \to \text{Flatten} \to \text{FC}_{512}$,
- **głowa polityki** – warstwa w pełni połączona $\text{FC}_{n_{\text{actions}}}$ (logity),
- **głowa wartości** – FC_1.

Rysunek 2: Architektura sieci A3C: wspólny ekstraktor cech + dwie głowice (policy, value).

4.2 n-krokowy zwrot i advantage

Dla każdego stanu s_t i akcji a_t obliczamy

$$R_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V_w(s_{t+n}), \quad A_t^{(n)} = R_t^{(n)} - V_w(s_t).$$

Na tej podstawie wyznaczamy gradienty g_θ, g_w zgodnie z (1) i formułą entropijną (**??**).

4.3 Mechanika asynchroniczna

Każdy proces–worker:

1. pobiera aktualne wagi θ, w (w RAM),
2. zbiera n kroków trajektorii,
3. liczy gradient (g_θ, g_w) lokalnie na CPU,
4. *bez blokady* dodaje je do wspólnego wektora wag,
5. zeruje licznik i powtarza.

Brak bariery synchronizacyjnej sprawia, że złożoność czasowa jednej iteracji $\approx \frac{\text{czas symulacji+backward}}{N_{\text{proc}}}$, co przyśpiesza uczenie prawie liniowo z liczbą rdzeni (jak pokazują wyniki w rozdz. **??**).

4.4 Pseudokod

Algorithm 1 Asynchroniczny worker A3C (wersja CPU)

Require: wspólne wagi θ, w, krok optymalizatora *opt*

$t \leftarrow 0$
while nieosiągnięto *max_frames* **do**
  $s \leftarrow$ *env.reset()*
  repeat
    $a \sim \pi_\theta(\cdot|s)$
    $(s', r, done) \leftarrow$ *env.step*(a)
    bufor $\leftarrow$ bufor $\cup\, (s, a, r)$
    $t \leftarrow t + 1$
    if *done* **or** $|\text{bufor}| = n$ **then**
      if *done* **then**
        $R \leftarrow 0$
      else
        $R \leftarrow V_w(s')$
      for elementy bufora od końca **do**
        $R \leftarrow r_i + \gamma R$
        $g_\theta \mathrel{+}= \nabla_\theta \log \pi_\theta(a_i|s_i)\,(R - V_w(s_i))$
        $g_w \mathrel{+}= \nabla_w \frac{1}{2}(R - V_w(s_i))^2$
      atomically *opt.step*(g_θ, g_w)
      wyzeruj bufor
    $s \leftarrow s'$
  until *done*

4.5 Konfiguracja CPU/GPU w implementacji

W praktycznej implementacji:

- sieć w workerach działa na **CPU** (`torch.set_num_threads(1)`), redukując przełączanie kontekstu GPU,

- wagi współdzielone są w pamięci `share_memory()`,
- główny proces konsoliduje gradienty i kopiuje model na GPU wyłącznie do etapu aktualizacji (batch 128).

Takie podejście spełnia zalecenia Lapana [**?**] i pozwala efektywnie wykorzystać pojedynczą kartę RTX 2080, zachowując zalety asynchronicznej aktualizacji A3C.

5 Konfiguracja eksperymentu

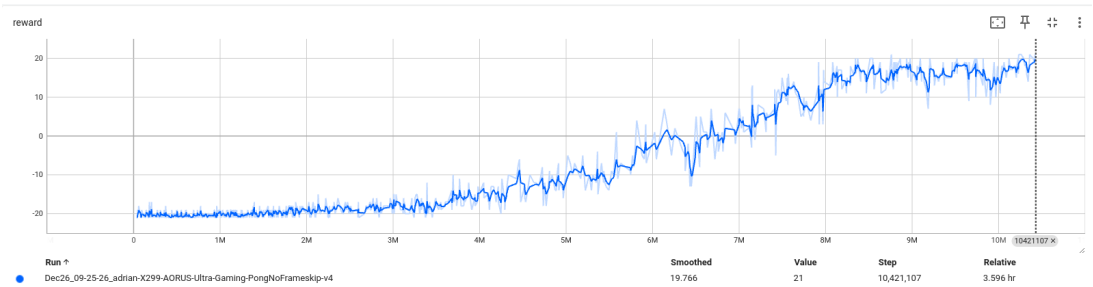
- Sprzęt: Intel i7-7820X (8C/16T), GPU RTX 2080, 16GB RAM
- Oprogramowanie: Python 3.12.7, PyTorch 2.5.1, Gymnasium 1.0.0
- Hiperparametry: tabela 1

Tabela 1: Kluczowe hiperparametry

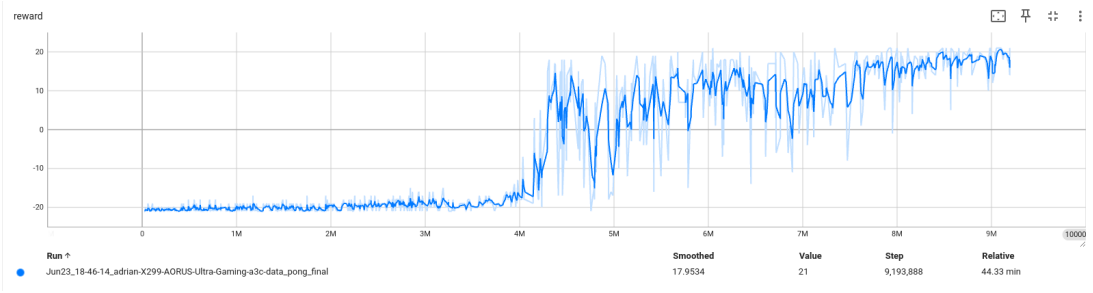
Parametr	Wartość
Współczynnik γ	0,99
Szybkość uczenia (α)	$1 \cdot 10^{-3}$
Entropy β	0,01
Liczba procesów	8 (testy 8-16)
Liczba środowisk / proces	4
n-krok	4
Mini-batch	128

6 Wyniki

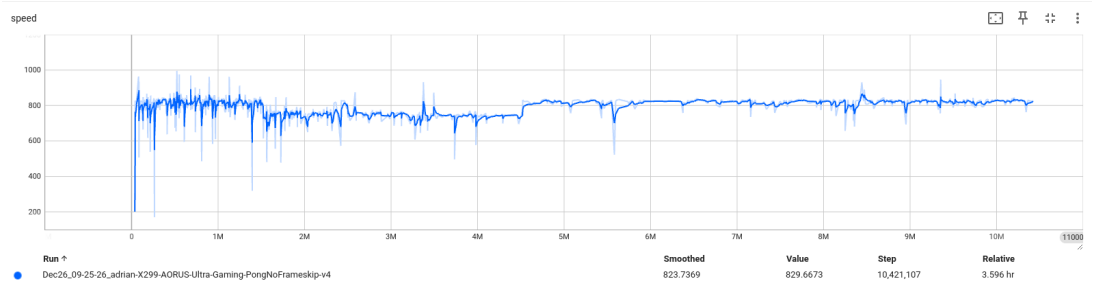
Poniższe wykresy prezentują przebieg uczenia dla **A2C (synchronicznego)** i **A3C (asynchronicznego)**. Oba eksperymenty uzyskały podobną liczbę kroków treningowych ($\approx$ 10,4 M dla A2C vs $\approx$ 9,2 M dla A3C); główna różnica dotyczy *tempa* generowania danych i szybkości zbieżności.



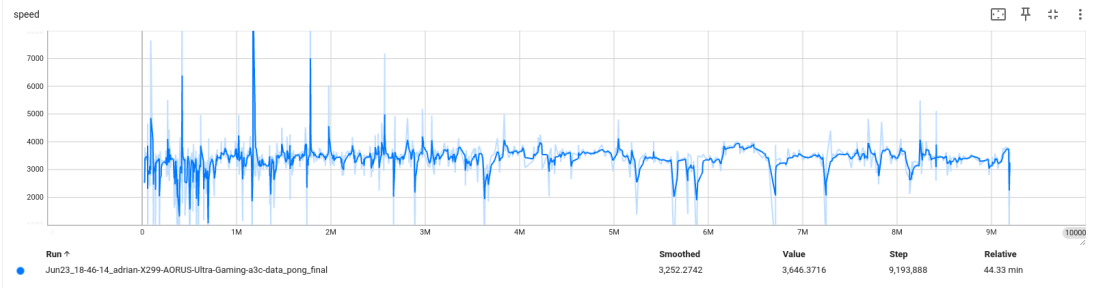
Rysunek 3: A2C – przebieg nagrody w czasie uczenia.



Rysunek 4: A3C – przebieg nagrody w czasie uczenia. Duża wariancja wynika z asynchronicznych, częstszych aktualizacji.



Rysunek 5: A2C – szybkość symulacji (kl./s)



Rysunek 6: A3C – szybkość symulacji (kl./s)

6.1 Porównanie czasu treningu

Tabela 2: Czas rzeczywisty potrzebny do osiągnięcia progu `REWARD_BOUND` = 18.

	Czas [hh:mm]	FPS (średnie)
A2C	03:36	≈ 800
A3C	00:45	$\approx 3{,}500$

Mimo podobnej liczby klatek, A3C osiąga granicę nagrody **czterokrotnie szybciej** w czasie rzeczywistym, dzięki wyższej przepustowości CPU. Jest to bezpośredni efekt większej przepustowości symulacji oraz częstszych, choć bardziej hałaśliwych aktualizacji wag.

6.2 Wariancja nagrody

Krzywa nagrody A3C (Rys. 4) wykazuje znacznie wyższą wariancję niż A2C. Źródła zjawiska:

1. **Asynchroniczne opóźnienie gradientu**. W momencie gdy jeden worker modyfikuje wspólne wagi, pozostałe procesy wciąż mogą liczyć gradient względem *starej* wersji sieci, co wprowadza stochastyczny „szum aktualizacji".

2. **Mniejszy n-batch lokalny**. Każdy worker A3C propaguje gradient co 32 próbki (`MICRO_BATCH_SIZE`), podczas gdy A2C kumuluje pełne batche 128 elementów przed jednym, synchronicznym krokiem.

3. **Różnica w n-kroku**. $n = 4$ w A3C oznacza dłuższy horyzont bootstrapu, a więc wyższą wariancję celu $R^{(n)}$ w porównaniu z $n = 3$ w A2C.

Mimo większych odchyleń, średnia krocząca A3C zbiega szybciej i stabilizuje się w obszarze $\approx$ 19–21 punktów (nagroda maksymalna dla *Ponga*).

6.3 Analiza efektywności

- **Wydajność CPU**. Dzięki `PROCESSES_COUNT`=8 i `NUM_ENVS`=4 wykorzystujemy 32 lekkie środowiska, co saturuje wszystkie 8 rdzeni fizycznych (HT: 16 wątków) i przekłada się na $\sim$ 3,5k kl./s.

- **Koszt synchronizacji**. A2C traci czas na barierę zbierania gradientów: FPS ustala się na $\sim$ 800 $kl./s$ mimo 50 środowisk (`NUM_ENVS`=50) — koszt kopiowania dużego batcha na GPU co krok.

- **Zużycie GPU**. Podczas trenowania algorytmu A2C zużycie GPU było na poziomie $\sim$ 40%, natomiast dla A3C dzięki wykorzystaniu synchroniczności osiągnęło $\sim$ 80%.

Wnioski z eksperymentu

1. A3C znacząco redukuje czas treningu na sprzęcie CPU + jedna GPU, wykorzystując proste zrównoleglenie na poziomie danych i brak bariery synchronizacji.

2. Wyższa wariancja nagrody to efekt „starych" gradientów i mniejszych batchy, lecz nie pogarsza końcowej wydajności agenta.

3. Przy zwiększaniu liczby procesów warto jednocześnie podnieść rozmiar globalnego batcha lub obniżyć α, aby uniknąć zbyt gwałtownych oscylacji wartości krytyka.