

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

Sprawozdanie z projektu – przedmiot „Algorytmy optymalizacji”

Adrian Galik Nr albumu: 268864

25 czerwca 2025

## Spis treści

<b>1</b>	<b>Wprowadzenie</b>	<b>2</b>
<b>2</b>	<b>Podstawy teoretyczne</b>	<b>2</b>
2.1	Model formalny RL . . . . .	2
2.2	Gradient polityki i aktor–krytyk . . . . .	2
2.3	$n$ -krokowa aktualizacja . . . . .	3
2.4	Synchroniczny A2C a asynchroniczny A3C . . . . .	3
2.5	Funkcja straty . . . . .	4
<b>3</b>	<b>Środowisko PongNoFrameskip-v4</b>	<b>4</b>
3.1	Surowa przestrzeń obserwacji i akcji . . . . .	4
3.2	Pipeline przetwarzania danych . . . . .	4
3.3	Metryki używane w eksperymentach . . . . .	5
<b>4</b>	<b>Algorytm A3C</b>	<b>5</b>
4.1	Architektura sieci . . . . .	5
4.2	$n$ -krokowy zwrot i advantage . . . . .	6
4.3	Mechanika asynchroniczna . . . . .	6
4.4	Konfiguracja CPU/GPU w implementacji . . . . .	6
<b>5</b>	<b>Konfiguracja eksperymentu</b>	<b>6</b>
<b>6</b>	<b>Wyniki</b>	<b>7</b>
6.1	Porównanie czasu treningu . . . . .	8
6.2	Wariancja nagrody . . . . .	8
6.3	Analiza efektywności . . . . .	9

# 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 [1] oraz oryginalnego artykułu A3C [2]. 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  $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$  – 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

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

Wartość stanu i wartość akcji definiujemy następująco:

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

### 2.2 Gradient polityki i aktor–krytyk

Twierdzenie o gradiencie polityki [3, 1] pozwala zapisać pochodną funkcji celu  $J(\theta) = \mathbb{E}_{s \sim d^{\pi_\theta}} \mathbb{E}_{a \sim \pi_\theta}[R_t]$  w postaci:

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

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 [1] :

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

na podstawie którego definiujemy

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

Wspólny gradient dla parametrów aktora i krytyka wynosi

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

$$g_w = \nabla_w \frac{1}{2} \left( R_t^{(n)} - V_w(s_t) \right)^2. \quad (3)$$

## 2.4 Synchroniczny A2C a asynchroniczny A3C

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

**A3C (Asynchronous Advantage Actor–Critic).** Każdy worker oblicza gradient zaraz po zebraniu własnego mini-batcha i *natychmiast* aktualizuje wspólne wagi w pamięci RAM [2]. Skutkuje to brakiem bariery synchronizacji i prawie liniowym wzrostem przepustowości przy zwiększaniu liczby rdzeni.

Istnieją dwa sposoby zrównoleglania metody aktor-krytyk: [1]

- **Zrównoleglenie na poziomie danych** - Każdy proces ma własne środowisko i kopię sieci neuronowej. Samodzielnie zbiera dane (przejścia), oblicza gradienty i lokalnie aktualizuje swoją sieć. Co jakiś czas asynchronicznie przesyła gradienty do wspólnej, głównej sieci, która jest aktualizowana.
- **Zrównoleglenie na poziomie gradientów** - Różne procesy obliczają gradienty równolegle na różnych danych. Gradienty są sumowane i używane do jednej wspólnej aktualizacji sieci neuronowej. Zaktualizowane wagi sieci neuronowej muszą być na bieżąco przekazywane do innych procesów.

**Zrównoleglenie 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 oraz ze względu na większe trudności w implementacji, a niższy zysk z prędkości w ramach optymalizacji.

## 2.5 Funkcja straty

Całkowita funkcja straty używana w eksperymencie, zgodnie z [2, 1], to

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

gdzie  $c_v$  to współczynnik części wartości (przyjmujemy 1), zaś  $\beta$  (=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 [2, 1]. 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  $\pm 21$ ).

### 3.2 Pipeline przetwarzania danych

Aby zmniejszyć wymiar wejścia i ustabilizować uczenie, stosujemy standardowy zestaw wrapperów zalecany w [2, 1]:

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  $84 \times 84$ px.

4. **ImageToPyTorch** – zmiana kolejności kanałów z HWC na CHW (wymóg PyTorch).
5. **FrameStack (k=4)** – konkatencja 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

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

### 3.3 Metryki używane w eksperymentach

- **Reward per game** – suma nagród od rozpoczęcia do końca meczu; wartość docelowa  $\geq 18$  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.).

### 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 Mnih'a i wsp. [2] 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 [2, 1] używamy architektury widocznej na rys. ??:

- część konwolucyjna wspólna dla polityki i krytyka:  $\text{Conv}_{8,4,32} \rightarrow \text{Conv}_{4,2,64} \rightarrow \text{Conv}_{3,1,64} \rightarrow \text{Flatten} \rightarrow \text{FC}_{512}$ ,
- **głowa polityki** – warstwa w pełni połączona  $\text{FC}_{n_{\text{actions}}}$  (logity),
- **głowa wartości** –  $\text{FC}_1$ .

## 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  $g_\theta$ ,  $g_w$  zgodnie z (1) i formułą entropijną (??).

## 4.3 Mechanika asynchroniczna

Każdy proces-worker:

1. pobiera aktualne wagi  $\theta, w$  (w RAM),
2. zbiera  $n$  kroków trajektorii,
3. liczy gradient  $(g_\theta, g_w)$  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  $\approx \frac{\text{czas symulacji} + \text{backward}}{N_{\text{proc}}}$ , co przyspiesza uczenie prawie liniowo z liczbą rdzeni (jak pokazują wyniki w rozdz. ??).

## 4.4 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 [1] i pozwala efektywnie wykorzystać pojedynczą kartę RTX 2080, zachowując zalety asynchronicznej aktualizacji A3C.

## 5 Konfiguracja eksperymentu

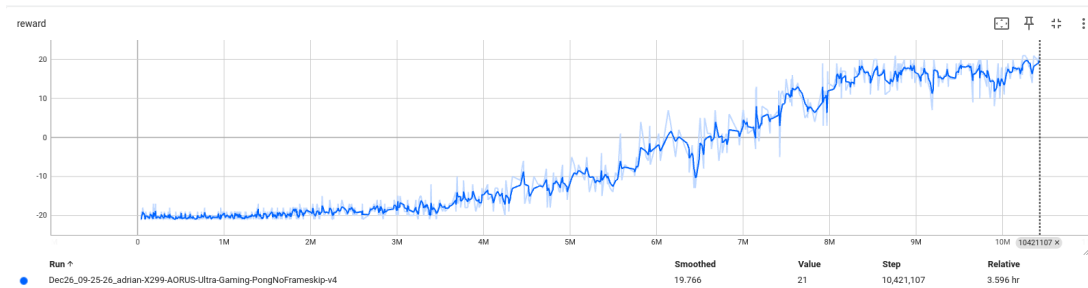
- Sprzęt: Intel i7-7820X (8C/16T), GPU RTX 2080, 16GB RAM
- Oprogramowanie: Python 3.12.7, PyTorch 2.5.1, Gymnasium 1.0.0
- Hiperparametry: tabela 1

Tabela 1: Kluczowe hiperparametry

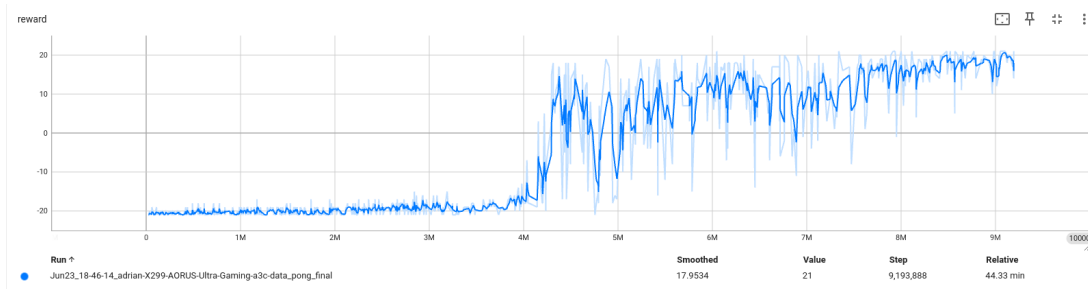
Parametr	Wartość
Współczynnik $\gamma$	0,99
Szybkość uczenia ( $\alpha$ )	$1 \cdot 10^{-3}$
Entropy $\beta$	0,01
Liczba procesów	8 (testy 8-16)
Liczba środowisk / proces	4
$n$ -krok	4
Mini-batch	128

## 6 Wyniki

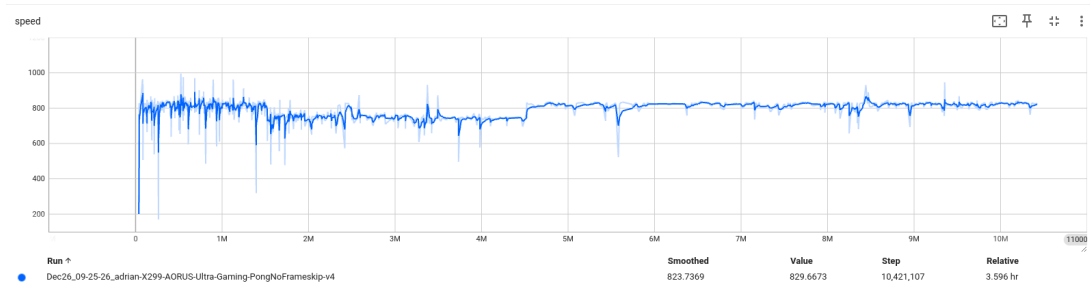
Poniższe wykresy prezentują przebieg uczenia dla **A2C (synchronicznego)** i **A3C (asynchronicznego)**. Oba eksperymenty uzyskały podobną liczbę kroków treningowych ( $\approx 10,4$  M dla A2C vs  $\approx 9,2$  M dla A3C); główna różnica dotyczy *tempa* generowania danych i szybkości zbieżności.



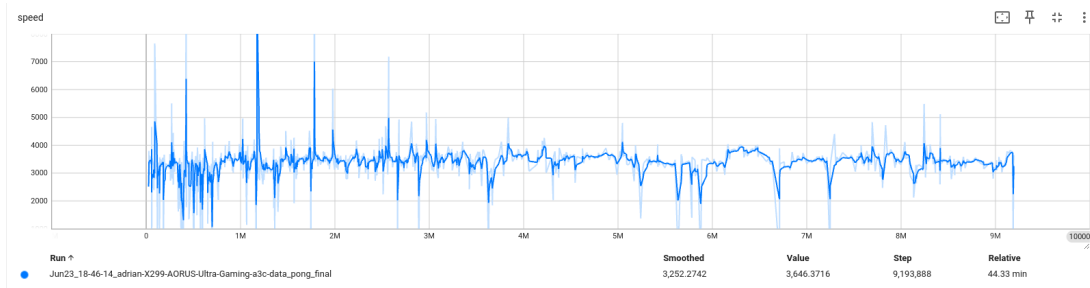
Rysunek 1: A2C – przebieg nagrody w czasie uczenia.



Rysunek 2: A3C – przebieg nagrody w czasie uczenia. Duża wariancja wynika z asynchronicznych, częstszych aktualizacji.



Rysunek 3: A2C – szybkość symulacji (kl./s)



Rysunek 4: A3C – szybkość symulacji (kl./s)

## 6.1 Porównanie czasu treningu

Tabela 2: Czas rzeczywisty potrzebny do osiągnięcia progu  $\text{REWARD\_BOUND} = 18$ .

	Czas [hh:mm]	FPS (średnie)
A2C	03:36	$\approx 800$
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. 2) 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 ( $\text{MICRO\_BATCH\_SIZE}$ ), 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 odchyień, średnia krocząca A3C zbiega szybciej i stabilizuje się w obszarze  $\approx 19\text{--}21$  punktów (nagroda maksymalna dla *Ponga*).

### 6.3 Analiza efektywności

- **Wydażność CPU.** Dzięki `PROCESSES_COUNT=8` i `NUM_ENVS=4` wykorzystujemy 32 lekkie środowiska, co saturażuje wszystkie 8 rdzeni fizycznych (HT: 16 wątków) i przekłada się na  $\sim 3,5\text{k kl./s}$ .
- **Koszt synchronizacji.** A2C traci czas na barierę zbierania gradientów: FPS ustala się na  $\sim 800 \text{ kl./s}$  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 zrównoleżenie 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  $\alpha$ , aby uniknąć zbyt gwałtownych oscylacji wartości krytyka.

Tabela 3: Wpływ pojedynczych zmian hiperparametrów na czas uczenia oraz końcową nagrodę.

Zmodyfikowany hiperparametr	Czas uczenia	Reward
-	44,33 min	21
<code>NUM_ENVS = 8</code>	39,9 min	20
<code>NUM_ENVS = 2</code>	1 h 02 min	21
<code>PROCESSES_COUNT = 10</code>	47,8 min	20
<code>PROCESSES_COUNT = 12, BATCH_SIZE = 256</code>	49,2 min	20
<code>LEARNING_RATE = 0.002</code>	18,5 min	19
<code>LEARNING_RATE = 0.003</code>	33,69 min	19

## Literatura

- [1] Maxim Lapan. *Deep Reinforcement Learning Hands-On*. Packt Publishing, Birmingham, UK, wydanie 1, 2018.
- [2] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *arXiv preprint arXiv:1602.01783*, 2016.

- [3] Richard S. Sutton, David McAllester, Satinder Singh, Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in Neural Information Processing Systems 12*, strony 1057–1063, 2000.