



POLITECHNIKA KRAKOWSKA im. T. Kościuszki
Wydział Inżynierii Elektrycznej i Komputerowej



Kierunek studiów: Informatyka w Inżynierii Komputerowej

STUDIA STACJONARNE

PRACA DYPLOMOWA

INŻYNIERSKA

Maciej Złotorowicz

Analiza algorytmów uczenia przez wzmacnianie z zaimplementowanym
algorytmem PPO (Proximal Policy Optimization)

Analysis of Reinforcement Learning Algorithms with
PPO implementation (Proximal Policy Optimization)

Opiekun pracy:
mgr inż. Grzegorz Nowakowski

Kraków, 2024

Spis treści

1 Wprowadzenie	3
1.1 Cele i zakres pracy	3
2 Przegląd literatury	4
3 Podstawy uczenia przez wzmacnianie	4
4 Toolchain	4
4.1 TensorFlow Graphs	5
4.2 Tensorboard	7
5 Ogólna architektura systemu	7
5.1 Struktura eksperymentu	9
6 Gębokie Q-Uczenie	12
6.1 Podstawowy Algorytm	12
6.1.1 Ulepszenia algorytmu	13
7 Proximal Policy Optimization	14
7.1 Algorytm gradientu polityki	14
7.2 Policy ratio	16
7.3 PPO	16
7.4 Replay Buffer	17
7.5 Off-policy vs On-policy	18
8 Curiosity-driven learning	18
8.1 Problem z ciekawością	18
8.2 Implementacja ciekawości	19
9 Selfplay	20
10 Hiperparametry i strojenie systemu	22
10.1 Hiperparametry w eksperymentach	22
10.2 Metryki	23
10.2.1 Średnie wartości	23
10.2.2 Nagrody jako metryka	23
10.2.3 Długość jako metryka	24
10.2.4 Strata krytyka	24
10.2.5 KL	25
10.3 Współczynniki pomocne przy strojeniu	25
11 Eksperymenty	28
11.1 Pacman	28
11.1.1 Sesje uczenia	31
11.1.2 DQN	32

11.1.3 PPO	38
11.1.4 Porównanie przebiegów uczących (v1,v1.3,v3,v4)	43
11.1.5 Curiosity (plain pixels)	44
11.1.6 Curiosity (autoencoder)	49
11.2 Connect4	54
12 Wyniki i dyskusja	57
13 Ograniczenia i dalszy rozwój	58
14 Podsumowanie	58

Streszczenie

W niniejszej pracy przedstawiono implementację algorytmu PPO (*ang. Proximal Policy Optimization*) oraz innych technik uczenia przez wzmacnianie, wykorzystując kompilator wbudowany w tensorflow. Analiza skupia się na porównaniu różnych metod i technik uczenia agentów w zróżnicowanych środowiskach, które obejmują zarówno potrzebę kontekstu, informacji temporalnych, jak i interakcji między wieloma agentami, zastosowanych w technikach self-play i eksploracji poprzez ciekawość (*ang. curiosity learning*).

Abstract

This paper presents implementation of the Proximal Policy Optimization algorithm and other reinforcement learning techniques using a tensorflow's graph compiler. The study focuses on comparing various methods and techniques for training agents in diverse environments, encompassing the need for temporal information, exploration through curiosity and interactions between multiple agents in self-play.

Uczenie przez wzmacnianie, Proximal Policy Optimization, Curiosity Learning, Arcade, Self-play, dqn, tensorflow, replay buffer, hiperparametry, implementacja, pacman

1 Wprowadzenie

W dzisiejszych czasach rozwój sztucznej inteligencji odgrywa kluczową rolę w osiąganiu zaawansowanych celów w niemal każdej dziedzinie życia. Obecnie modele generatywne osiągają rezultaty, które jeszcze kilka lat temu wydawały się nieosiągalne, co stanowi znaczący postęp w dziedzinie inżynierii. Algorytmy uczenia przez wzmacnianie stanowią podstawę dla tych modeli, charakteryzując się często alternatywnym podejściem do procesu trenowania. Pomijają one kwestie dokładnego rozwiązania, skupiając się głównie na osiągnięciu pożądanych efektów.

Dzięki nim, a szczególnie za sprawą stabilnego algorytmu PPO (*ang. Proximal Policy Optimization*), staje się możliwe precyzyjne dostosowywanie dużych modeli językowych do różnych zastosowań, jak na przykład zamiana ich w rozbudowane systemy czatu. Co więcej, sam algorytm, który doskonale sprawdza się w środowiskach wymagających interakcji i skomplikowanych strategii, może być doskonalony w celu osiągania wyników przewyższających zdolności ludzkie w popularnych grach, takich jak szachy, czy też w zaawansowanych grach komputerowych, jak Dota czy Starcraft.

1.1 Cele i zakres pracy

Celem niniejszej pracy jest przeprowadzenie szczegółowej analizy różnych metod i technik uczenia agentów w zróżnicowanych środowiskach. Skoncentrowano się tutaj na środowiskach

wymagających analizy danych graficznych oraz zrozumienia kontekstu stanu i interakcji między wieloma modelami w uczeniu w stylu *selfplay*, w którym agent gra sam ze sobą w celu opracowania efektywnej strategii w grze rywalizacyjnej.

W niniejszym opracowaniu skoncentrowano się na analizie algorytmu PPO oraz technik uczenia przez wzmacnianie. Oprócz tego zostały przedstawione techniki łączone z uczeniem przez wzmacnianie, w tym *curiosity learning*. Praca obejmuje także wyniki procesu uczenia oraz przykładowe działania agentów, które ilustrują skuteczność zastosowanych metod.

2 Przegląd literatury

Głównym źródłem inspiracji dla niniejszej pracy są artykuły naukowe poświęcone rozwojowi i analizie algorytmów uczenia przez wzmacnianie (*ang. reinforcement learning, RL*). Szczególnie istotne są publikacje takie jak '*Playing Atari with Deep Reinforcement Learning*' [1] oraz '*Dota 2 with Large Scale Deep Reinforcement Learning*' [2]. Algorytm PPO, używany w ramach niniejszego badania, został oryginalnie opisany w artykule '*Proximal Policy Optimization Algorithms*' [3]. Technika eksploracji przez ciekawość została oparta na artykule zatytułowanym '*Large-Scale Study of Curiosity-Driven Learning*' [4], który szczegółowo opisuje działanie tej metody i potencjalne problemy na jakie można natrafić w procesie jej implementacji. Techniki, pomysły oraz zalecenia dotyczące przebiegu uczenia się również zostały oparte na informacjach pochodzących z tych źródeł.

3 Podstawy uczenia przez wzmacnianie

Uczenie przez wzmacnianie to paradymat uczenia maszynowego, w którym *agent* (nazywany również modelem) jest uczony poprzez interakcję ze środowiskiem. Głównym celem *agenta* jest maksymalizacja nagród w określonych ramach czasowych, które otrzymuje za wykonywanie odpowiedniej sekwencji akcji. Uczenie przez wzmacnianie znajduje zastosowanie w sytuacjach, gdzie *agent* musi podejmować decyzje sekwencyjne, a rezultaty tych decyzji stają się widoczne dopiero w przyszłości [5].

To prowadzi do sytuacji, w której *agent* musi skorelować podjęte akcje z późniejszymi nagrodami. Problem ten jest powiązany z *procesami decyzyjnymi Markowa (MDP)*, w których optymalizacja nagród może być łatwo przybliżona za pomocą prostego algorytmu. Jednak w uczeniu przez wzmacnianie zazwyczaj nie dysponujemy pełnym modelem środowiska, a jedynie estymujemy go przy użyciu głębokiej sieci neuronowej. Oznacza to, że stan lub obserwacja zwracane przez środowisko to jedynie częściowa informacja.

4 Toolchain

Projekt został opracowany z użyciem biblioteki *TensorFlow* i *@tf.function*. Dodatkowo wykorzystano kilka otwartoźródłowych bibliotek do emulacji i konteneryzacji środowisk, takich jak *gym* i *pettingzoo*. W celu tworzenia interaktywnych *dashboardów* zastosowano narzędzie *TensorBoard*. Analiza danych, podsumowanie i generowanie wykresów zostały przeprowadzone przy użyciu bibliotek takich jak *Matplotlib*, *NumPy*, *Pandas* oraz notatników *Jupyter*.

4.1 TensorFlow Graphs

Podczas pracy z wysokopoziomowym językiem programowania, takim jak Python, w kontekście algorytmów przetwarzających duże ilości danych, istotnym wyzwaniem staje się kwestia wydajności. Szczególnie istotne jest wykorzystanie wbudowanych kompilatorów w bibliotece *TensorFlow*.

Podstawowym mechanizmem działania kompilatora *grappler* [6] jest interpretacja kodu źródłowego jako grafu przepływu informacji. Każda funkcja oznaczona jako `@tf.function` zostanie skompilowana do grafu, który może być efektywnie uruchomiony na sprzęcie przystosowanym do uczenia maszynowego, na przykład procesorze graficznym. Proces ten nazywany jest śledzeniem (*ang. tracing*) i wiąże się z kosztem, zwłaszcza podczas pierwszego wywołania funkcji. *Grappler* jest również w stanie optymalizować pętle i warunki, przedstawiając je w formie łatwej do wektorowania, a także minimalizować ilość alokacji na procesorze graficznym i automatycznie rozkładając obciążenie w dużych systemach.

Wadą tego podejścia jest konieczność dostosowania się do wymagań kompilatora, który z kolei potrzebuje poprawnie określonych typów zmiennych, precyzyjnie zdefiniowanych kształtów tensorów oraz korzystania z określonego zestawu narzędzi. Ponadto, wymaga to napisania "czystych funkcji", czyli funkcji, które nie generują żadnych efektów ubocznych. Warto zauważyć, że ogólnie rzecz biorąc, funkcje oznaczone adnotacją `@tf.function` stanowią pewien odrębny język i pisanie ich różni się znaczco od standardowego kodu w języku Python.

Te zasady wprowadzają dodatkowe wyzwania w pracy z systemem i wymagają dodatkowego wysiłku w zakresie programowania, aby unikać błędów, które mogą prowadzić do niezdefiniowanego zachowania, błędów komplikacji lub degradacji wydajności, szczególnie związaną z ponownym śledzeniem funkcji. Zazwyczaj wiąże się to z pomiarem wydajności, analizą wygenerowanego grafu lub dokładnym czytaniem komunikatów lub błędów. Przykładowo, poniższy kod jest nieprawidłowy, ponieważ nie jest to czysta funkcja.

```
1  i = 0
2  @tf.function
3  def sample_function():
4      global i
5      i += 1
```

Listening 1: Nieprawidłowy kod i wyzwania związane z czystymi funkcjami w TensorFlow [7]

Tracer umożliwia również przekształcanie pętli w operacje na tensorach. W związku z tym poniższy kod zostanie zrównoleglny, a pętla zostanie usunięta i zastąpiona odpowiednim wyrażeniem.

```

1  @tf.function
2  def sample_function():
3      buffer = tf.TensorArray(dtype=tf.float32, size=0, dynamic_size=True)
4
5      for i in tf.range(10):
6          buffer = buffer.write(i, i)
7
8  return buffer.stack()

```

Listening 2: Przykład użycia pętli i bufora w `@tf.function` [7]

`@tf.function` różnią się od języka Python na tyle, że można by je uznać za inny język, będący podzbiorem Pythona. Aby efektywnie współdziałać z kodem w standardowym Pythonie, konieczne jest korzystanie z dedykowanych funkcji umożliwiających zmianę kontekstu z trybu grafowego na tryb pythonowy. Przykład takiej funkcji przedstawiony jest poniżej, wymagając podania typów zwracanych i kształtu tensora. Taka struktura pozwala na minimalizację kosztów związanych z wykonywaniem operacji w Pythonie oraz umożliwia równoległe przetwarzanie kodu. Warto zaznaczyć, że taka reprezentacja znacznie ułatwia późniejsze różniczkowanie automatyczne, które wymaga śledzenia grafu wstecz.

```

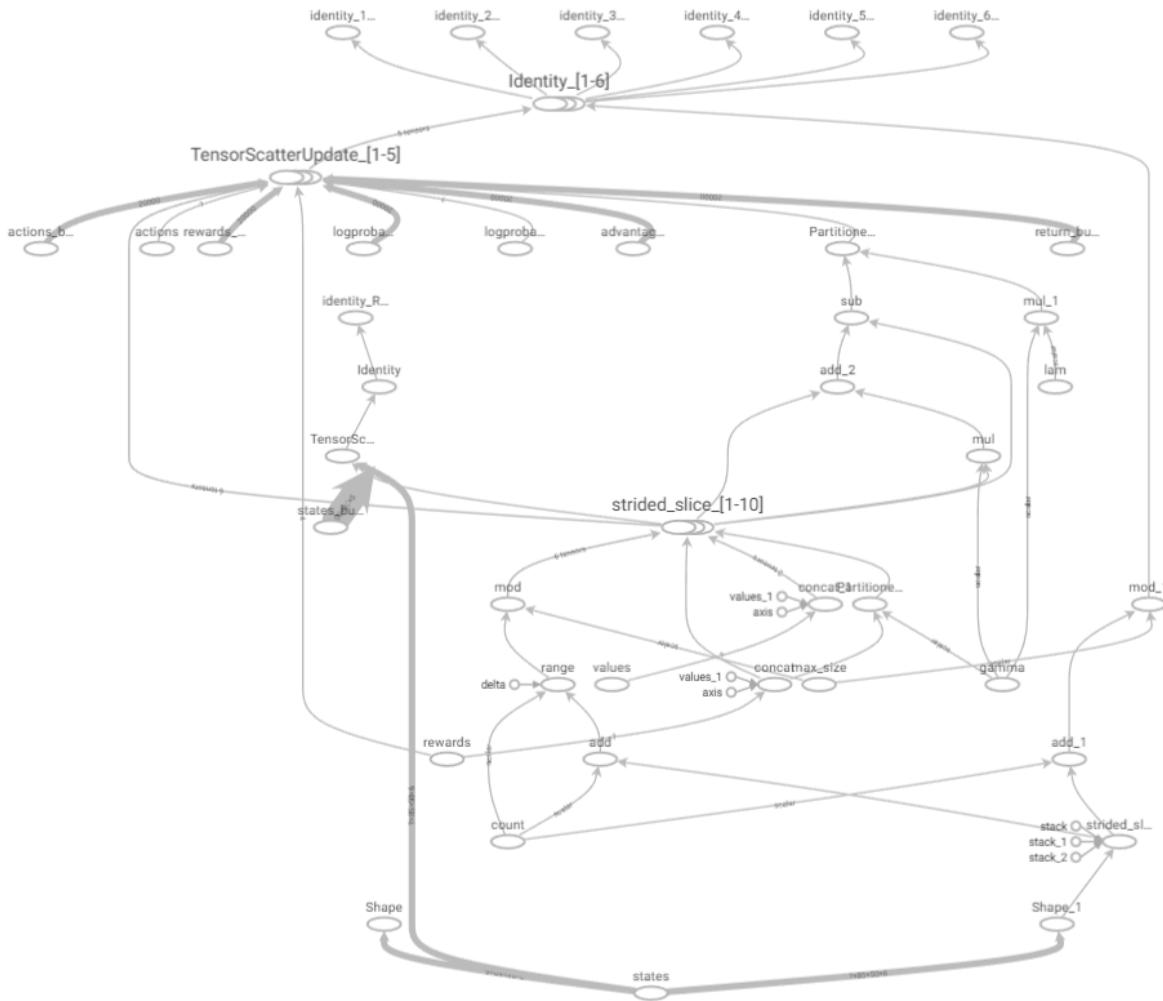
1  def step(action):
2      state, reward, done, _, _ = env.step(int(action))
3      state = observation_transformer(state)
4      return (np.array(state, np.float32), np.array(reward, np.float32), np.array(done, np.int32))
5
6  @tf.function(input_signature=[tf.TensorSpec(shape=(), dtype=tf.int32)])
7  def tf_env_step(action):
8      return tf.numpy_function(step, [action], (tf.float32, tf.float32, tf.int32))

```

Listening 3: Przykład korzystania z Pythona z wnętrza skompilowanych funkcji [7]

Zyski z procesu kompilacji stają się wyraźne zwłaszcza w przypadku kodu zawierającego liczne małe elementy, takie jak pętle czy warunki. W praktyce, im więcej takich elementów, tym większy potencjalny zysk. W trakcie testów kod zazwyczaj wykazuje wzrost wydajności od 4 do 10 razy. Warto jednak zaznaczyć, że korzyści te są szczególnie ograniczone, gdy funkcja już składa się z zoptymalizowanych operacji, co ma miejsce na przykład podczas ewaluacji sieci neuronowej. W tym przypadku sama sieć neuronowa stanowi graf wymagający intensywnych obliczeń, które nie korzystają z interfejsu Pythona.

Poniższy rysunek ilustruje graf funkcji *episode runnera*, której ogólne działanie zostało opisane w kolejnych sekcjach. Graf ten zawiera pętle oraz warunki i został przedstawiony w formie graficznej przy użyciu *TensorBoard*.



Rysunek 1: Przykładowy graf tensorflow [7]

4.2 Tensorboard

TensorBoard to narzędzie służące do prostego zapisywania danych z procesu uczenia. Pozwala na łatwą agregację i porównywanie danych skalarnych, histogramów, obrazów oraz danych tekstowych. Dodatkowo, *TensorBoard* oferuje kilka narzędzi do profilowania i pomiaru wydajności. Każde uruchomienie programu jest zapisywane jako odrębny przebieg, umożliwiając klarowne śledzenie postępów w różnych etapach uczenia. Większość wykresów generowana jest na podstawie danych zapisanych w *TensorBoard*. Do ich agregacji służą skrypty napisane w notatnikach *jupyter*.

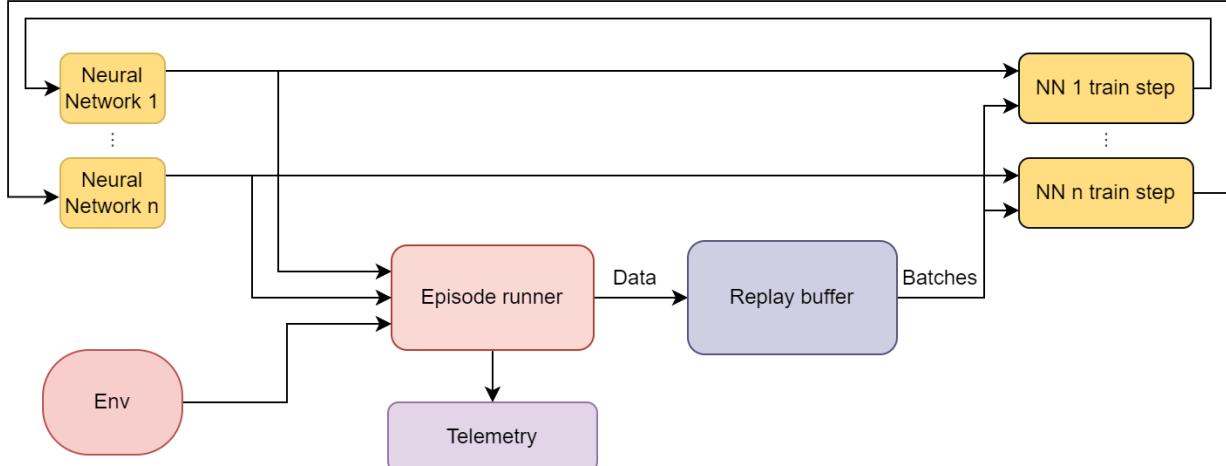
5 Ogólna architektura systemu

Każdy eksperyment posiada swój własny plik, gdzie zdefiniowane są hiperparametry, struktury sieci, pętle uczące oraz telemetryka. Funkcje i algorytmy wykorzystywane do procesu

uczenia są uogólnione i wydzielone do odrębnych plików. Kluczowymi elementami są definicje kroków uczenia, które znajdują się w `./src/algorithms`, kolektory doświadczeń w `./src/exp_collektors`, oraz bufory odtwarzania w `./src/memory.py`). Te komponenty są ogólne i wykorzystywane w różnych eksperymentach jako funkcje lub obiekty:

- `./src/algorithms` zawierają definicje kroków uczących, które najczęściej przyjmują model (lub modele), dane i optymalizator i wykonują zdefiniowany krok algorytmu spadku po gradiencie.
- `./src/exp_collektors` są to algorytmy, które definiują sposób eksploracji środowiska. Te algorytmy przyjmują środowisko oraz modele, a następnie wykonują jeden epizod w danym środowisku (do osiągnięcia limitu kroków lub wczesnego zakończenia definiowanego przez środowisko). Następnie zwracają zebrane informacje i pewne dane statystyczne.
- `./src/memory.py` zawierają bufory odtwarzania. Są to specjalne struktury przechowujące dane zebrane przez kolektory i generujące zbiory uczące dla każdego podzadania optymalizacyjnego.

Poniższy diagram przedstawia ogólną architekturę eksperymentów. W większości z nich zachodzi współpraca wielu sieci neuronowych jednocześnie, więc rysunek ogólnie przedstawia sieci i odpowiadające im funkcje optymalizacyjne.



Rysunek 2: Schemat przedstawiający architekturę skryptów trenujących modele [8]

Główna pętla iteruje po epizodach, gdzie każdy epizod reprezentuje jedno przejście przez *episode runnera*, odpowiadające jednej rozgrywce, zwanej również *trajektorią*.

Trajektoria to uporządkowany zbiór danych zebranych po przejściu przez środowisko jednokrotnie. Zazwyczaj zbierane dane obejmują obserwacje, nagrody i podjęte akcje. Jednak w zależności od algorytmu, zbierane są również inne wartości, takie jak prawdopodobieństwa wyprodukowane przez model, następny stan lub predykcje innych sieci.

Poniżej znajduje się pseudokod ogólnego algorytmu wykonywania jednego epizodu w środowisku. Proces ten polega na przechodzeniu przez środowisko, wywołując funkcję *make_step* z akcją wybraną na podstawie predykcji modelu, a następnie zbieraniu wszystkich danych do buforów (*TensorArray*):

```
1 def run_episode(initial_state: tf.Tensor, model: tf.keras.Model, max_steps: tf.Tensor, make_step: EnvStep):
2     """
3         Run a single episode of the environment using the given model.
4     """
5     states = tf.TensorArray(dtype=tf.float32)
6     actions = tf.TensorArray(dtype=tf.int32)
7     rewards = tf.TensorArray(dtype=tf.float32)
8     next_states = tf.TensorArray(dtype=tf.float32)
9     dones = tf.TensorArray(dtype=tf.float32)
10
11    state = initial_state
12
13    for t in tf.range(max_steps, dtype=tf.int32):
14        states = states.write(t, state)
15
16        action_logits_t = model(state)
17        action = sample_action(action_logits_t)
18
19        next_state, reward, done = make_step(action)
20
21        actions = actions.write(t, action)
22        rewards = rewards.write(t, reward)
23        next_states = next_states.write(t, next_state)
24        dones = dones.write(t, done)
25
26        state = next_state
27
28        if done:
29            break
30
31    return (states.stack(), actions.stack(), rewards.stack(), next_states.stack(), dones.stack())
```

Listening 4: Funkcja wykonująca jedną iterację w środowisku [7]

5.1 Struktura eksperymentu

Każdy eksperiment to plik w języku Python, którego uruchomienie rozpoczyna proces uczenia. Są one zaprojektowane w taki sposób, aby umożliwiać łatwe wznawianie procesu treningu.

Każda sesja jest oznaczona nazwą, która zawiera:

- Timestamp uruchomienia eksperymentu,
- Wersję eksperymentu (dowolny ciąg znaków, zazwyczaj *v1*, *v2* itp.),
- Przedrostek *dry_* dla eksperymentów, które zostały uruchomione w celu testowania lub debugowania.

Ustawienia każdej sesji są zapisywane w logach na serwerze *TensorBoard*, gdzie można śledzić przebieg procesu trenowania agenta. Informacje te są zapisywane w formie tekstowej, co jest kluczowe dla zapewnienia reprodukowalności wyników. Dodatkowo, dla istotnych zmian tworzone są nowe pliki eksperymentów. Poniżej przedstawiono jeden z takich zapisów w formie tekstowej na podstawie którego można później odtworzyć eksperiment lub sprawdzić jego wariację.

```
1  ### dry_20231116-175936
2  #### 2023-11-16 17:59:38
3  ---
4  ## params
5  - env_name: CartPole-v1
6  - version: v1.0
7  - DRY_RUN: True
8  - actor_lr: 0.0002
9  - critic_lr: 0.001
10 - action_space: 2
11 - observation_space_raw: (4,)
12 - observation_space: (4,)
13 - episodes: 100000
14 - max_steps_per_episode: 1000
15 - discount_rate: 0.99
16 - eps_decay_len: 1000
17 - eps_min: 0.1
18 - clip_ratio: 0.2
19 - lam: 0.97
20 - batch_size: 4000
21 - train_interval: 200
22 - iters: 80
23 - save_freq: 500
```

Listening 5: Przykład zapisu hiperparametrów z logów [7]

Każdy eksperiment zawiera również definicje modeli oraz główną pętlę uczącą, w której częstotliwości różnych akcji są określane przez hiperparametry. Informacje zapisane tutaj mogą być łatwo prześledzone przy użyciu dedykowanego skryptu.

Poniższy kod przedstawia uproszczoną strukturę eksperymentu w celu zachowania czytelności. Po inicjalizacji środowiska (*env*), wczytaniu argumentów (*parser*) i zdefiniowaniu parametrów użytych przy treningu (*params*) następuje zdefiniowanie lub wczytanie sieci neuronowych z dysku. Następnie uruchamiana jest główna pętla ucząca, która oprócz definicji funkcji do iteracji po środowisku, wygenerowanej przez *make_tensorflow_env_step*, określa również ogólną pętlę, w której uruchamia się *episode runner* i kroki uczące. Oryginalny kod jest dodatkowo wzbogacony o zapisy telemetryczne oraz inne działania mające na celu ułatwienie pracy z systemem.

```

1  parser = argparse.ArgumentParser()
2  parser.add_argument("--resume", type=str, default=None, help="resume from a model")
3
4  env = environments.get_packman_stack_frames()
5
6  params = argparse.Namespace()
7
8  params.env_name = env.spec.id
9  params.version = "v3.0"
10 params.DRY_RUN = False
11
12 params.actor_lr = 3e-8
13 params.critic_lr = 1e-5
14 # define other parameters
15
16 # init everything
17
18 def run():
19     memory = PPOReplayMemory(20_000, params.observation_space)
20
21     env_step = environments.make_tensorflow_env_step(env, lambda x: environments.pacman_transform_observation_stack_big(x))
22     env_reset = environments.make_tensorflow_env_reset(env, lambda x: environments.pacman_transform_observation_stack_big(x))
23
24     runner = get_ppo_runner(env_step)
25     runner = tf.function(runner)
26
27     t = tqdm.tqdm(range(params.episodes))
28     for episode in t:
29         initial_state = env_reset()
30
31         (states, actions, rewards, values, log_probs), total_rewards = runner(initial_state,
32                             actor, critic,
33                             max_steps_per_episode,
34                             action_space)
35
36         memory.add(states, actions, rewards, values, log_probs)
37
38         if len(memory) >= params.batch_size and int(episode) % params.train_interval == 0:
39             batch = memory.sample(batch_size)
40
41             training_step_ppo(batch, actor, action_space, clip_ratio, policy_optimizer, episode)
42
43             # other optimization steps
44
45         if episode % params.save_freq == 0 and episode > 0:
46             save_model(episode, actor, critic, params)
47
48 run()

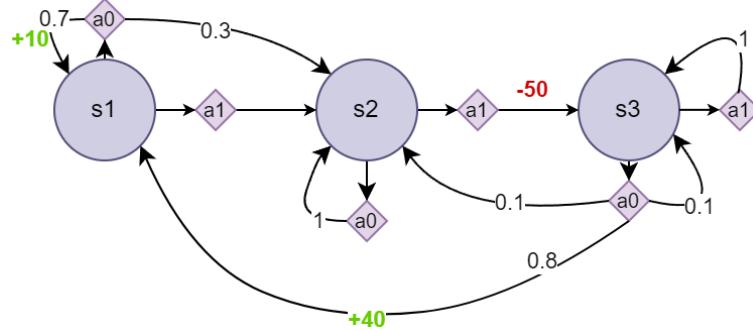
```

Listening 6: Przykładowa uproszczona struktura eksperymentu [7]

6 Gębokie Q-Uczenie

Q-Uczenie to algorytm, który na początku zakłada, że prawdopodobieństwa przejść między stanami oraz nagrody są nieznane. W celu eksploracji środowiska, algorytm stosuje strategię *epsilon-greedy* (epsilon-zachłanną), co oznacza wybieranie losowej akcji z prawdopodobieństwem epsilon, zamiast akcji o największym prawdopodobieństwie. Strategia ta, która różni się od tej używanej podczas standardowej rozgrywki w której wybierana jest zawsze najlepsza akcja, jest nazywana *off-policy*. Pozwala ona na losową eksplorację środowiska, zaczynając od epsilon równej 1 (co oznacza, że agent działa losowo) i stopniowo zmniejszając tę wartość w stronę 0. Jednakże wartość epsilon niekoniecznie musi osiągnąć zero, aby umożliwić ciągłą eksplorację środowiska [1].

Dzięki temu agent może oszacować prawdopodobieństwa przejść i kojarzyć je z nagrodami, budując wewnętrzny graf przejść [9], podobny do przedstawionego poniżej. W tym grafie akcje (a_0, a_1) mogą prowadzić do różnych stanów, a niektóre z tych przejść są skorelowane z przypisanymi im nagrodami.



Rysunek 3: Przykład procesu decyzyjnego Markowa [8]

Gębokie Q-uczenie używa w celu oszacowania tych Q-wartości sieci neuronowej.

6.1 Podstawowy Algorytm

Podstawowy algorytm jest stosunkowo prosty, ale niestety również wykazuje niską skuteczność. Jest on szczególnie wrażliwy na różne parametry systemu, nawet na rozmiar sieci — dodanie kilku neuronów może istotnie zdegradować jego wydajność. Wysoka niestabilność stanowi istotną wadę tego algorytmu, dlatego opracowano wiele technik mających na celu poprawę jego stabilności. Zastosowanie tych technik jest niezbędne, zwłaszcza w przypadku średnio skomplikowanych środowisk. Funkcja docelowych Q-wartości jest relatywnie prosta i przedstawiona jest na poniższym wzorze. Dla uzyskania funkcji straty można skorzystać z dowolnej funkcji obliczania błędu, na przykład średniego błędu kwadratowego (*MSE*) lub, w celu stabilizacji algorytmu, funkcji straty Hubera.

$$Q_{target}(s, a) = r + \gamma \cdot \max_{a'} Q_\theta(s', a')$$

Wzór ten polega na szacowaniu przyszłych nagród na podstawie aktualnej nagrody i szacowanej przyszłej nagrody zdyskontowanej o współczynnik *discount_ratio* (często w notacji stosuje się grecką literę gamma (γ)).

Podstawowa implementacja jest dość prosta. Należy pamiętać, że zbiór danych jest budowany przez aktora w czasie rzeczywistym i ciągle się zmienia. Po pobraniu partii danych (ang. *batch*) można oszacować aktualne Q-wartości w porównaniu do rzeczywistych i obliczyć docelowe Q-wartości. Następnie można je wykorzystać do wzmacnienia lub osłabienia danej akcji. W tym przykładzie użyto błędu średniokwadratowego.

```

1 def training_step_DQN(batch, model, num_of_actions, discount_rate, optimizer):
2     states, actions, rewards, next_states, dones = batch
3
4     # calculate target Q values
5     next_Q_values = model(next_states)
6     max_next_Q_values = tf.reduce_max(next_Q_values, axis=1)
7     # get target Q values, special case for last step (done)
8     target_Q_values = rewards + (1 - dones) * discount_rate * max_next_Q_values
9
10    mask = tf.one_hot(actions, num_of_actions)
11
12    with tf.GradientTape() as tape:
13        # get current Q values from model
14        all_Q_values = model(states)
15        # select only the Q values for the actions that were taken by agent
16        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
17        # get the loss
18        loss = mse(target_Q_values, Q_values)
19
20    grads = tape.gradient(loss, model.trainable_variables)
21    optimizer.apply_gradients(zip(grads, model.trainable_variables))

```

Listening 7: Algorytm optymalizacyjny w algorytmie DQN [7]

6.1.1 Ulepszenia algorytmu

Algorytm w tej formie jest praktycznie bezużyteczny i nie nadaje się do prawie żadnego zadania. Dlatego konieczne jest wprowadzenie kilku technik mających na celu jego ustabilizowanie.

Stosuje się tutaj techniki takie jak sieci docelowe (ang. *target network*) [9], podwójnie sieci DQN (ang. *double network*) [10] oraz algorytm Aktora-Krytyka ([11])

7 Proximal Policy Optimization

Proximal Policy Optimization (w skrócie PPO) to algorytm, który stanowi wariację gradientu polityki. Jego celem jest stabilizacja procesu uczenia poprzez minimalizację liczby wprowadzanych zmian w polityce w każdym epizodzie.

7.1 Algorytm gradientu polityki

Algorytm gradientu polityki polega na estymacji gradientów polityki i wykorzystaniu standardowego algorytmu spadku po gradiencie. W przypadku PPO używana jest wersja algorytmu przedstawiona poniżej. W tym wzorze $\pi_\theta(a_t|s_t)$ to polityka (sieć neuronowa) przyjmująca na wejście stan w chwili t i zwracająca prawdopodobieństwa akcji. W algorytmie korzysta się z logarytmicznego prawdopodobieństwa, co wynika z metody obliczania gradientów w tym wzorze. Dzięki zastosowaniu "triku logarytmicznego różniczkowego" (ang. *log-derivative trick*) można właściwie zastosować regułę łańcuchową. Ten trik pozwala na ustabilizowanie gradientów i minimalizację błędów numerycznych [3].

$$L^{PG}(\theta) = \hat{E}_t[\log \pi_\theta(a_t|s_t) \hat{A}_t]$$

Następnie, to prawdopodobieństwo mnoży się przez przewagę (*ang. advantage*), we wzorze oznaczony jako \hat{A}_t . Jest to różnica między przybliżeniem i rzeczywistą zdyskontowaną nagrodą w jednym epizodzie.

Wzór poniżej przedstawia jak można obliczyć *advantage*. Tutaj t to krok czasowy i T oznacza ostatni krok czasowy, dla którego często nie znamy nagrody, więc estymujemy ją przy użyciu krytyka (alternatywą jest przyjęcie jakieś stałej - na przykład zera).

$$\hat{A}_t = -V(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T)$$

Należy pamiętać że *advantage* to wektor i oblicza się go dla każdej wartości T w ciągu. W kodzie ważną funkcją jest zdyskontowana kumulatywna suma przedstawiona poniżej.

```
1 def discounted_cumulative_sums_tf(x, discount_rate) -> tf.Tensor:
2     size = tf.shape(x)[0]
3     x = tf.reverse(x, axis=[0])
4     buffer = tf.TensorArray(dtype=tf.float32, size=size)
5
6     discounted_sum = tf.constant(0.0, dtype=tf.float32)
7
8     for i in tf.range(size):
9         discounted_sum = x[i] + discount_rate * discounted_sum
10        buffer = buffer.write(i, discounted_sum)
11
12    return tf.reverse(buffer.stack(), axis=[0])
```

Listening 8: Algorytm obliczający zdyskontowane kumulatywne sumy [7]

Następnie można obliczyć *advantage* na podstawie wcześniej zapisanych estymacji krytyka (*values*) oraz rzeczywistych nagród (*rewards*). Warto zauważyć, że parametr *lam* pełni rolę dodatkowego hiperparametru, który umożliwia niezależne sterowanie zachowaniem krytyka i aktora. Obliczenia w kodzie mogą wydawać się trochę inne, ale wynika to z faktu, że obliczamy tutaj wartości dla wszystkich podciągów rozpoczynających się od pierwszej obserwacji.

```
1 # finish trajectory (assume last reward is 0 instead of V(s_T))
2 rewards = tf.concat([rewards, [0.0]], axis=0)
3 values = tf.concat([values, [0.0]], axis=0)
4
5 # calculate deltas between actual rewards and estimated values from critic
6 deltas = rewards[:-1] + gamma * values[1:] - values[:-1]
7
8 advantages = discounted_cumulative_sums_tf(
9     deltas, gamma * lam
10 )
```

Listening 9: Algorytm obliczania *advantage* [7]

W celu wytrenowania krytyka oblicza się w tym samym punkcie czasowym wartość *returns*, która stanowi zdyskontowaną sumę nagród. Jest to cel optymalizacji dla krytyka.

```
1 returns = discounted_cumulative_sums_tf(
2     rewards, gamma
3     )[:-1]
4
```

Listening 10: Algorytm obliczania *returns* [7]

Te wartości są niezbędne do obliczenia gradientu polityki. Niestety, ten algorytm posiada jedną istotną wadę - jest niestabilny i trudny w użyciu. W przypadku wielokrotnego wykorzystania tych samych próbek w procesie spadku po gradiencie dochodzi do ekstremalnie dużych aktualizacji polityki, co często skutkuje jej zniszczeniem. Pomimo tego wielokrotne wykorzystanie tych samych próbek jest bardzo pożąданie, ponieważ minimalizuje ilość potrzebnych iteracji w środowisku.

Rozwiązaniem jakie zaproponowano w algorytmie PPO jest zmiana celu optymalizacji.

7.2 Policy ratio

W PPO zamiast gradientu polityki oblicza się $r_t(\theta)$ czyli stosunek prawdopodobieństw ze starej polityki do prawdopodobieństw w nowej ([3], strona 3).

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

Wartości są większe od 1, gdy nowa polityka ma większe prawdopodobieństwo wybrania danej akcji w porównaniu do starej i mniejsze od 1, gdy to prawdopodobieństwo jest mniejsze. Implementacja w kodzie różni się trochę ze względu na optymalizację – można pominąć zapisywanie starej polityki po prostu zapisując te prawdopodobieństwa w trakcie zbierania danych.

Wartości w *logprobability_buffer* są pobierane z pamięci i reprezentują logarytmiczne prawdopodobieństwa akcji według starej polityki. Aktualną polityką jest *actor* a na podstawie niej ponownie obliczane są te prawdopodobieństwa. Stosunek (*ratio*) można obliczyć bazując na podstawowych właściwościach potęgowania.

```
1 # batch from memory
2 observation_buffer, action_buffer, logprobability_buffer, advantage_buffer = batch
3
4 with tf.GradientTape() as tape:
5     logits = actor(observation_buffer) # current policy
6
7     ratio = tf.exp(
8         logprobabilities(logits, action_buffer, num_of_actions)
9         - logprobability_buffer
10    )
```

Listening 11: Obliczanie ratio [7]

7.3 PPO

Proximal Policy Optimization korzysta z przycinania (*ang. clipping*) w celu ograniczenia tego, jak bardzo nowa polityka może się różnić od starej. Stosuje się cel optymalizacji przedstawiony na poniższym wzorze:

$$L_t(\theta) = \min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t)$$

Funkcja przycina wartości $r_t(\theta)$ między pewnym zakresem mieszczącym się między $1 - \epsilon$ a $1 + \epsilon$. Warto zaznaczyć, że epsilon to prawie zawsze wartość 0.2, co oznacza, że nowa polityka może się różnić od starej nie więcej niż o 20%. W efekcie tego mechanizmu możliwość zmiany polityki aktora na podstawie aktualnego stanu bufora, z którego się uczy, zostaje zablokowana w pewnym zakresie. Konsekwencją tego jest wymuszenie na aktorze powolnych zmian w zachowaniu, co może wydawać się nieintuicyjne. Jednak szybkość trenowania nie

ma znaczenia, jeżeli algorytm nie działa poprawnie. PPO skutecznie rozwiązuje problem z dużą niestabilnością oraz nieprzewidywalnym procesem trenowania modelu, co charakteryzuje algorytmy gradientu polityki i DQN. Cała funkcja optymalizacyjna w kodzie jest bardzo podobna i zawiera drobne przekształcenia wzoru w celu uproszczenia implementacji.

```
1 with tf.GradientTape() as tape:
2     logits = actor(observation_buffer)
3
4     ratio = tf.exp(
5         logprobabilities(logits, action_buffer, num_of_actions)
6         - logprobability_buffer
7     )
8
9     min_advantage = tf.where(
10        advantage_buffer > 0,
11        (1 + clip_ratio) * advantage_buffer,
12        (1 - clip_ratio) * advantage_buffer,
13    )
14
15     policy_loss = -tf.reduce_mean(
16         tf.minimum(ratio * advantage_buffer, min_advantage)
17     )
18     policy_grads = tape.gradient(policy_loss, actor.trainable_variables)
19     optimizer.apply_gradients(zip(policy_grads, actor.trainable_variables))
```

Listening 12: Algorytm PPO [7]

7.4 Replay Buffer

Implementacja pamięci odtwarzania (*ang. replay buffer*) przechowuje dane niezbędne do szkolenia wszystkich sieci neuronowych. Oprócz funkcji *add* i *reset* zawiera ona kilka wariantów funkcji *sample*, które służą do zbierania partii danych (*ang. batch*) dla każdego podproblemu optymalizacyjnego. Bufory przechowywane przez tę pamięć obejmują:

- *states_buffer* - bufor stanów, zawierający stan środowiska, który ma rozmiar obserwacji;
- *advantages_buffer* - przewagę obliczoną dla tej próbki uczącej;
- *actions_buffer* - bufor zawierający podjętą akcję dla tego stanu (*int*);
- *rewards_buffer* - nagrodę otrzymaną po podjęciu akcji;
- *return_buffer* - zdyskontowaną sumę nagród;
- *logprobability_buffer* - logarytmiczne prawdopodobieństwo akcji wygenerowane przez model;
- *next_states_buffer* - stan środowiska, w którym się ono znalazło po wykonaniu akcji.

W tej architekturze zakłada się, że dodawane do pamięci wartości to trajektoria. W związku z tym *replay buffer* oblicza na ich podstawie *zwroty* i *przewagi* a następnie zapisuje je w odpowiednich buforach. Przewagi pobierane z bufora są zawsze znormalizowane, co udowodnione zostało jako korzystne dla efektywnego działania algorytmu PPO.

7.5 Off-policy vs On-policy

Algorytm PPO jest algorytmem *on-policy*, co oznacza, że może korzystać z tej samej polityki zarówno do uczenia, jak i do działania. Jest to różnica w porównaniu do algorytmu DQN, który wymaga heurystyki, aby skłonić go do eksploracji środowiska, na przykład *epsilon-greedy*.

8 Curiosity-driven learning

PPO czasami napotyka na problem utknięcia w lokalnych minimach. Aby przeciwdziałać temu zjawisku, często stosuje się technikę znaną jako eksploracja przez ciekawość (*ang. curiosity-driven learning*). Metoda ta polega na tym, że aktor jest zachęcany do eksploracji środowiska poprzez dodatkową nagrodę za odkrywanie nieznanych lub zaskakujących stanów [4].

Aby osiągnąć ten cel, wprowadza się kolejną sieć neuronową do procesu trenowania - sieć ciekawości, która przyjmuje na wejściu stan i akcje, a następnie próbuje przewidzieć następny stan środowiska.

$$s'_{t+1} = s_{t+1} \approx C_\theta(s_t, a_t)$$

Następnie błąd średniokwadratowy z wyniku tej estymacji jest wykorzystywany jako nagroda dla aktora.

$$\text{reward} = \text{reward} + c_{\text{coef}} \cdot \text{mse}(s'_{t+1}, s_{t+1})$$

Sieć ciekawości to problem uczenia nadzorowanego i wykonuje się go jako jeden z kroków pętli uczącej.

8.1 Problem z ciekawością

Niestety, mimo pewnych zalet, technika eksploracji przez ciekawość niesie ze sobą pewne wyzwania. Jednym z głównych problemów jest uzależnianie aktora od losowych lub chaotycznych elementów środowiska ([4], strona 3). W środowiskach, gdzie występuje szum, aspekty losowe lub chaotyczne, sieć ciekawości może napotykać trudności w dokładnej predykcji, co prowadzi do sytuacji, w której aktor otrzymuje duże nagrody za eksplorację obszarów środowiska, które mogą być losowe lub chaotyczne. To z kolei nie zawsze jest korzystne.

Niemniej jednak, badania wskazują, że preferowanie części środowiska o skomplikowanej dynamice może być skuteczną strategią w nauce, pomimo problemów, jakie ze sobą niesie ta technika, szczególnie po modyfikacji algorytmu [4].

Oryginalne badanie sugeruje, że modele osiągają jeszcze lepsze rezultaty, gdy nie polegają na predykcji opartej na pikselach lecz korzystają z pewnego rodzaju kodowania. Przykładem takiego kodowania może być wygenerowany przez *autoencoder* wektor reprezentujący

stan środowiska. W pierwotnej pracy proponowano użycie *autoenkodera wariacyjnego*, który potencjalnie osiąga jeszcze lepsze wyniki. Niemniej jednak, ze względu na złożoność tego rozwiązania, w tej implementacji skorzystano ze zwykłego *autoenkodera*. Trenowano go do kodowania stanu środowiska w N-wymiarowy wektor, który następnie był wykorzystywany w technice uczenia przez ciekawość.

Można to zapisać jako wzór:

$$reward = reward + c_{coef} \cdot mse(h_\theta(C_\theta(s_t, a_t)), h_\theta(s_{t+1}))$$

Gdzie h_θ to część kodująca z autoenkodera. Hiperparametr c_{coef} czyli współczynnik ciekawości to wartość określająca jaką wagę mają nagrody ciekawości. W kodzie często określa się go jako *curious_coef*.

8.2 Implementacja ciekawości

Implementacja ciekawości polega przede wszystkim na dodaniu dodatkowej nagrody, jednak wymaga to rozważenia kilku istotnych szczegółów. Po pierwsze, teraz nagrody są obciążone przez człon ciekawości, co oznacza, że aby uzyskać porównywalną miarę z standardową techniką lub po dostosowaniu współczynnika ciekawości, konieczne jest odjęcie tego elementu przy zapisywaniu w dziennikach lub logach.

Metoda ta wymaga zapisania następnego stanu w buforze odtwarzania wraz z bieżącym stanem, co prowadzi do znacznie większego zużycia pamięci. Jest to mniej uciążliwe w wariancie zawierającym *autoenkoder*, ponieważ przestrzeń kodowania *autoenkodera* zazwyczaj jest dość mała (na przykład wektor 100-wymiarowy).

Należy jednak zauważać, że jest to kwestia, która zwiększa zużycie pamięci, zwłaszcza gdy obserwacje są obszerne i trzeba je zapisywać podwójnie, na przykład w przypadku danych graficznych lub serii grafik.

Poniższy kod przedstawia sposób obliczania nagrody za ciekawość dla aktora w wersji bez *autoenkodera*. W przypadku *autoenkodera* stany muszą zostać zakodowane do wektora, a następnie z tego wektora obliczony zostaje błąd.

```

1 # Run the model and to get action probabilities and critic value
2 action_logits_t = actor(state)
3
4 # sample an action from the action probability distribution
5 action = tf.squeeze(tf.random.categorical(action_logits_t, 1), axis=1)
6 action = tf.cast(action, tf.int32)
7 action = tf.squeeze(action)
8
9 # make a step in the environment
10 next_state, reward, done = tf_env_step(action)
11
12 next_state.set_shape(initial_state_shape)
13
14 action_one_hot = tf.one_hot(action, env_actions)
15 action_input_processed = tf.expand_dims(action_one_hot, axis=0)
16
17 # calculate curiosity
18 encoded_state = encoder(state)
19 encoded_next_state = encoder(tf.expand_dims(next_state, 0))
20 encoded_predicted_state = curiosity([encoded_state, action_input_processed])
21 encoded_predicted_state = tf.squeeze(encoded_predicted_state, axis=0)
22
23 curiosity_reward = tf.reduce_sum(tf.square(encoded_next_state - encoded_predicted_state))
24
25 # add curiosity reward to the reward
26 reward = reward + curius_coef * curiosity_reward

```

Listening 13: Wycinek *episode runnera* korzystającego z ciekawości [7]

9 Selfplay

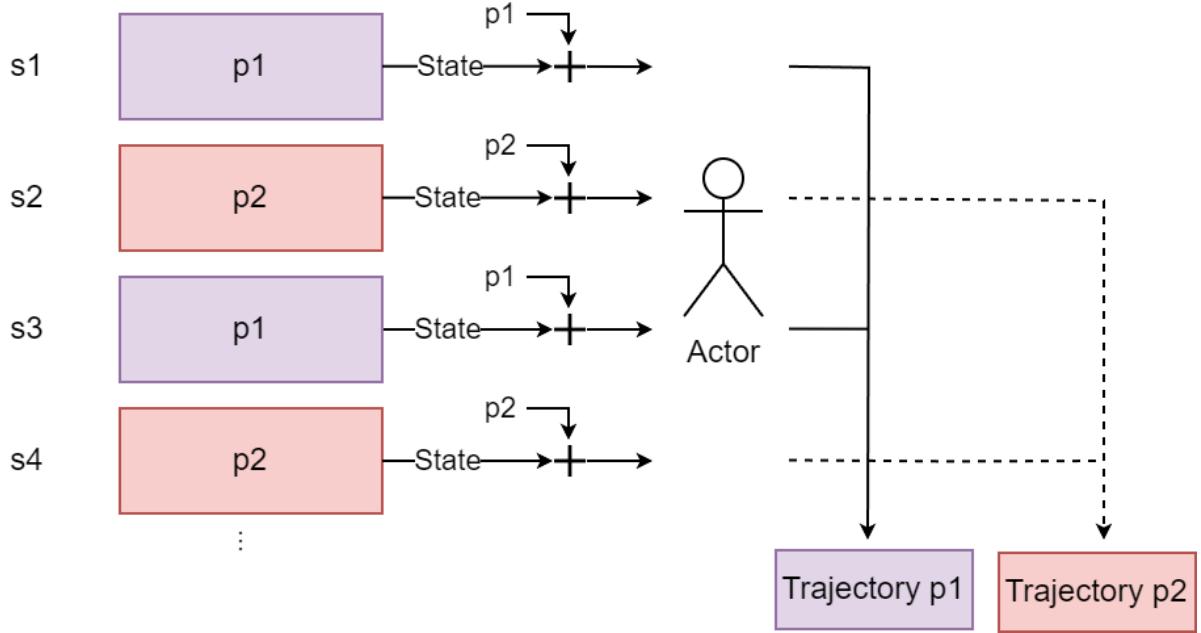
Technika *self-play* to wariant uczenia przez wzmacnianie (*ang. reinforcement learning, RL*), w którym wielu agentów dąży do współpracy lub rywalizacji w celu maksymalizacji nagród. Istnieje wiele metod i algorytmów w ramach tej techniki, z każdą różniącą się szczegółami, które wpływają na sposób lub podejście do problemu. Elementem wspólnym jest zawsze model, który optymalizuje swoje działania dla obu graczy, przyjmując na wejście stan i informacje o aktualnym graczu.

Środowiska wieloagentowe zazwyczaj przyjmują akcje kolejnych agentów sekwencyjnie, na przykład w grze dwuosobowej naprzemiennie: $p_1, p_2, p_1\dots$. Następnie zbierane są dwie trajektorie - po jednej dla każdego gracza, na podstawie których obliczane są przewagi i zwroty.

W systemie możemy rozważyć kilka wariantów trenowania:

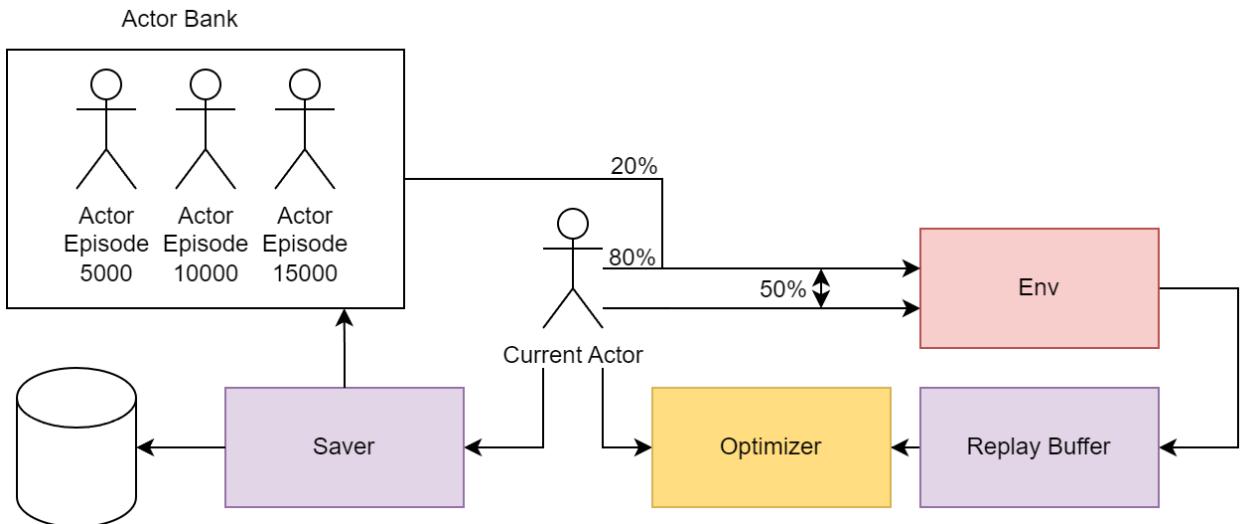
- uczymy dwa różne modele, które grają jedynie po jednej ze stron,
- uczymy jeden model, który uczy się grać po obu stronach,
- uczymy dwie kopie grające po obu stronach i wybieramy lepszy model co kilka epizodów.

Dwa całkowicie oddzielne modele uczące się różnych strategii jedynie dla jednego gracza to często nieskuteczna technika. Modele te mogą się rozbiec podczas procesu uczenia, a jeden z nich może utknąć w lokalnym minimum. W wyniku ciągłej optymalizacji obu stron może pojawić się problem z wyjściem z tej sytuacji.



Rysunek 4: Schemat zbierania danych w technice selfplay [8]

Znacznie lepszą strategią jest uczenie jednego modelu i przechowywanie kilku kopii starszych wersji, aby uodpornić model na nagłe zmiany we własnej strategii i degenerację poprzednio wygenerowanych taktyk. Niektóre prace sugerują, aby 80% gier rozgrywać z aktualną polityką i 20% ze starszymi wersjami ([2], strona 5). Rysunek 5 przedstawia typową architekturę self-play, w której wprowadzamy bank aktorów zawierający wcześniejsze wersje modeli. W wersji uproszczonej może to być jeden starszy aktor aktualizowany co relatywnie dużą ilość iteracji.



Rysunek 5: Koncepcja uczenia aktora w technice selfplay [8]

Innym problemem jest wprowadzenie jeszcze większej niestabilności w procesie trenowa-

nia. W środowisku zmienia się nie tylko strategia aktora, ale również strategia przeciwnika. Self-play jest szczególnie trudny do nauki, nawet jak na standardy uczenia przez wzmacnianie i wymaga bardzo ostrożnych aktualizacji modeli. Z eksperymentów wynika, że aktualizacje powinny być rzadkie i poparte dużą ilością rozgrywek.

10 Hiperparametry i strojenie systemu

Algorytmy optymalizacyjne zazwyczaj posiadają wiele parametrów, których strojenie znacząco wpływa na wyniki i skuteczność ich działania. Niestety, algorytmy uczenia przez wzmacnianie mają szczególnie dużo parametrów, które mają istotny wpływ na finalny model. Poniżej przedstawiono, jakie mają one znaczenie dla działania algorytmu oraz jak wygląda metodologia strojenia tych parametrów.

10.1 Hiperparametry w eksperymencie

Poza ogólnymi hiperparametrami charakterystycznymi dla algorytmów uczenia maszynowego takimi jak: *learning_rate* (*lr*) i *batch_size* *RL* posiada wiele specyficznych wartości. Poniżej znajduje się lista kilku podstawowych parametrów wraz z ich nazwami i krótkim opisem:

- *actor_lr* - współczynnik uczenia dla sieci aktora,
- *critic_lr* - współczynnik uczenia dla sieci krytyka,
- *episodes* - sumaryczna ilość iteracji po środowisku jaka zostanie wykonana w danym eksperymencie,
- *max_steps_per_episode* - maksymalna ilość kroków jaka może zostać wykonana w środowisku,
- *discount_rate* - stopa dyskontowa, hiperparametr w uczeniu przez wzmacnianie określający wagę przyszłych nagród w porównaniu do aktualnych,
- *epsilon* - wartość epsilon w polityce zachłannej określający szansę na wybranie losowej akcji zamiast akcji o najwyższym prawdopodobieństwie,
- *clip_ratio* - współczynnik przycinania w PPO,
- *lam* - współczynnik dzięki któremu można ustawić różne *discount_rate* przy obliczaniu advantage i returns,
- *curious_coef* - współczynnik ciekawości,
- *batch_size* - rozmiar podzbioru uczącego,
- *train_intervals* - okres, w którym zostaną uruchomione algorytmy optymalizacyjne,
- *iters* - ilość iteracji algorytmów optymalizacyjnych,
- *save_freq* - ilość iteracji co którą zostają zapisane modele,
- *replay_buffer_length* - długość bufora odtworzeń.

10.2 Metryki

Najważniejszym aspektem w procesie uczenia modeli jest ich efektywna ocena. Miary skuteczności zazwyczaj silnie zależą od charakterystyki środowiska, z przykładem popularnej metryki, jaką jest sumaryczna ilość nagród.

W profesjonalnych systemach, zwłaszcza opartych na *self-play*, powszechnie stosuje się pomiar Elo. Umożliwia on iteracyjne pozycjonowanie graczy przypisując im liczbową wartość, co pozwala określić ich umiejętności w relacji do innych graczy lub modeli. Niemniej jednak, ta technika wymaga ustalenia pewnego punktu odniesienia, czyli algorytmu lub gracza, który pozwoli na względne określenie jak dany model sobie radzi.

W niektórych środowiskach dobrą miarą skuteczności agenta może być ilość iteracji w środowisku, proporcjonalna do jego wydajności. To zależy od konkretnego środowiska, jednak wiele z nich określa warunki wczesnego zakończenia epizodu, co uniemożliwia dalsze zdobywanie nagród. Wartości funkcji straty, współczynnika KL (*ang. Kullback-Leibler*) a także oceny bezpośredniego zachowania modeli są również istotne dla oszacowania skuteczności.

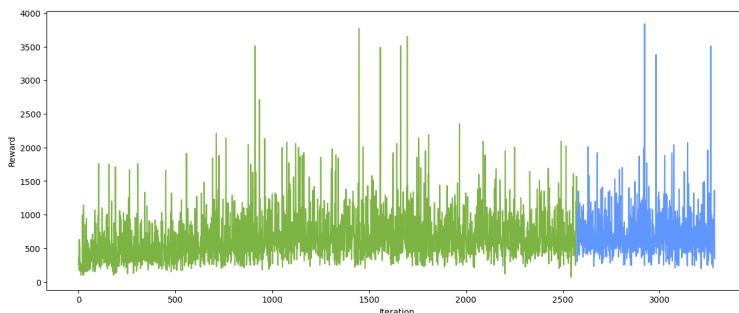
W trakcie trenowania modeli najczęściej obserwowane były metryki takie jak: nagrody, średnie nagrody, długość trwania rozgrywki oraz wartość funkcji straty.

10.2.1 Średnie wartości

Odszumianie przebiegów generowanych przez algorytmy często polega na wykorzystaniu średniej jako najprostszego sposobu. Wartości generowane przez algorytmy są często losowe, co sprawia, że bez kontekstu są trudne do interpretacji. Dlatego powszechnie metody przetwarzania wykresów obejmują stosowanie średniej ruchomej (o oknie 200) lub średniej wykładniczej (zmienny parametr). Często obie techniki są łączone w celu uzyskania silniejszego efektu wygładzenia wykresu.

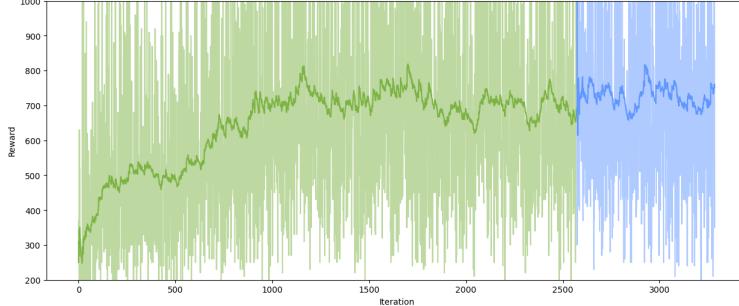
10.2.2 Nagrody jako metryka

Nagrody zazwyczaj stanowią bardzo zaszumioną metrykę. Bez uśredniania lub innego rodzaju wygładzania ilość szumu może znacząco utrudnić dokładne zrozumienie przebiegu lub zidentyfikowanie jakiejkolwiek zależności. Przebieg uczenia z nagrodami przedstawiony poniżej stanowi przykład. Na jego podstawie trudno jest oszacować, jak efektywnie radzi sobie aktor.



Wykres 1: Nagrody uzyskiwane w czasie uczenia (przykład) [12]

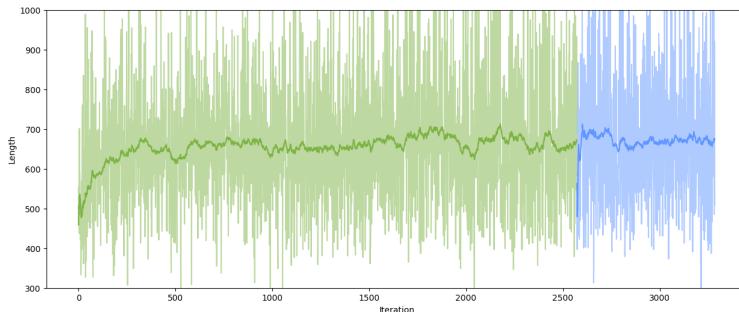
Wykres staje się znacznie bardziej czytelny po zastosowaniu wygładzania. Najczęściej stosowanymi metodami są średnia wykładnicza ze zmiennym współczynnikiem (wbudowana w *TensorBoard*) oraz średnia ruchoma. Na poniższym wykresie zastosowano wygładzanie ze współczynnikiem 0.98.



Wykres 2: Nagrody uzyskiwane w czasie uczenia z wygładzaniem średnią wykładniczą [12]

10.2.3 Długość jako metryka

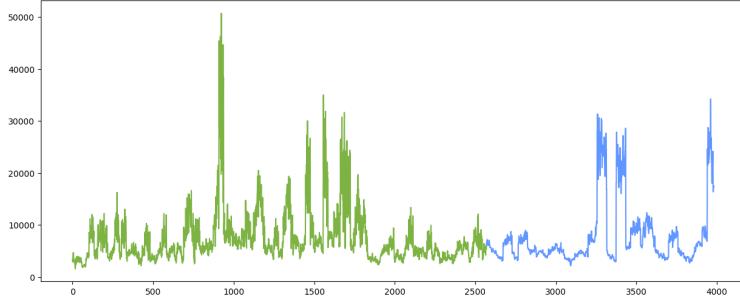
Długość to kolejna fundamentalna metryka, a często dłuższe przeżycie agenta w środowisku jest interpretowane jako lepsze wyniki. Wzrost długości często sygnalizuje, że aktor podejmuje decyzje, które pozwalają mu utrzymać się dłużej w środowisku. Jednym z podstawowych sposobów poprawy czytelności przebiegu tej metryki jest zastosowanie wygładzania przy użyciu średniej wykładniczej.



Wykres 3: Długość rozgrywki w czasie uczenia modelu [12]

10.2.4 Strata krytyka

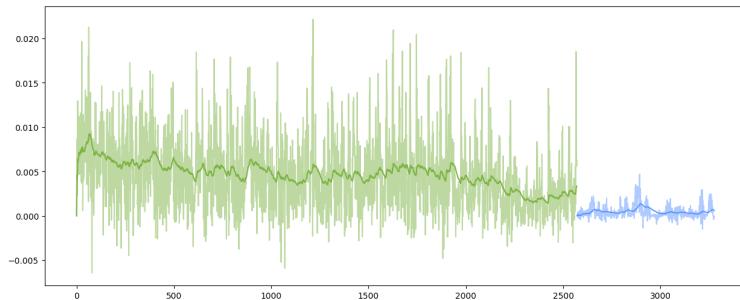
Funkcja straty krytyka również może być źródłem istotnych informacji. Ponieważ krytyk uczy się przewidywać opłacalność akcji, jego strata stanowi miarę nieprzewidywalności w środowisku. Na przykład, poniżej aktor zaczął zdobywać relatywnie wysokie nagrody, co spowodowało wzrost straty krytyka. Następnie, gdy krytyk dostosował się do nowej strategii funkcja straty zaczęła maleć. Taka dynamiczna reakcja w systemie jest objawem dobrze działającego krytyka.



Wykres 4: Nagrody uzyskiwane w czasie uczenia z wygładzaniem średnią wykładniczą [12]

10.2.5 KL

Różnica Kullbacka-Leiblера (*ang. Kullback–Leibler divergence, KL*) to miara niepodobieństwa między dwoma rozkładami prawdopodobieństwa. Jest bardzo łatwa do obliczenia na podstawie danych generowanych podczas uczenia, co pozwala śledzić siłę zmian w każdej iteracji uczenia. Małe zmiany wskazują, że polityka znalazła strategię i nie zachodzą już duże zmiany. Jest to rodzaj wskaźnika niepewności sieci w podejmowaniu decyzji.



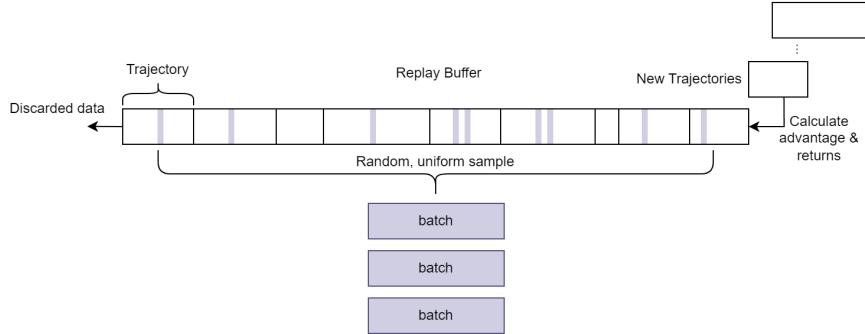
Wykres 5: Zmiana wartości współczynnika KL podczas uczenia [12]

10.3 Współczynniki pomocne przy strojeniu

Aby ułatwić zarządzanie hiperparametrami, warto zwrócić uwagę na kilka współczynników, które można obliczyć na podstawie danych zbieranych w trakcie uczenia lub parametrów modelu.

Bardzo ważnym współczynnikiem jest średnia ilość użycia jednej próbki w procesie uczenia. Wartość równa 1 oznacza, że próbka jest używana tylko raz, a następnie usuwana z pamięci. Wartości większe oznaczają, że ta sama próbka jest używana w kilku iteracjach uczenia. Optymalnie powinno się dążyć do wielokrotnego wykorzystywania tych samych próbek, a tę zasadę podzielają twórcy wielu algorytmów uczenia maszynowego ([3], strona 2).

Ilość użyć zależy od rozmiaru grupy (*ang. batch*), wielkości pamięci oraz ilości danych generowanych przez środowisko. Na przykład, jeżeli środowisko produkuje średnio 500 próbek na epizod, zapełnienie bufora o rozmiarze 10000 próbek zajmie 20 epizodów. Gdy uruchomimy iteracje uczenia z grupą o wielkości 2048, nasz model "zobaczy" 2048 próbek przy każdej iteracji. W zależności od ilości iteracji i okresu, co jaki trenujemy, można określić, ile średnio



Rysunek 6: Replay Buffer i przepływ informacji [7]

razy sieć zobaczy tą samą próbkę, co wpływa na szybkość uczenia i stabilność. Znaczące nadpróbkowanie może negatywnie wpływać na działanie algorytmu. Algorytmy starają się maksymalizować swoją skuteczność przy wysokim próbkowaniu - dlatego na przykład PPO jest dość odporne na wielokrotną iterację po tych samych danych.

Algorytmy uczenia przez wzmacnianie często korzystają z dużych minigrup ([2], strona 12). Rozmiar tych grup może zaczynać się od 2048, ale większość eksperymentów została przeprowadzona na grupach o wielkości od 4000 do 8000 próbek. W niektórych artykułach proponowano grupy o rozmiarach nawet kilkunastu tysięcy próbek. Głównym powodem takiego podejścia jest uzyskanie dokładniejszych gradientów poprzez zebranie większej ilości informacji.

Dobór współczynnika uczenia (*ang. learning rate, lr*) jest dość prosty. Ze względu na stosunkowo duże minigrupy, zazwyczaj zaczynamy od dużych wartości, na przykład od 0.01 do 0.001 i stopniowo je zmniejszamy, jeżeli szybkość trenowania jest zbyt niska. Często krytyk posiada oddzielny, większy o rzad wielkości *lr*. Należy jednak pamiętać, aby nie zmniejszać *lr* za szybko, ponieważ może to niepotrzebnie spowolnić proces trenowania. Uczenie przez wzmacnianie to dynamiczny proces, w którym cel optymalizacji może się ciągle zmieniać. Na przykład aktor może zacząć obserwować zupełnie nową część środowiska, której wcześniej nie widział i wtedy niski *lr* spowolni uczenie w tym obszarze w porównaniu do wcześniejszego. Często lepszym podejściem do przyspieszenia procesu uczenia jest zwiększenie ilości iteracji zamiast zwiększania *lr* ([2], strona 31).

Ilość iteracji i okres trenowania również zmieniają szybkość i charakter. Niektóre algorytmy optymalizacyjne posiadają wewnętrzne stany takie jak inercja, na przykład ADAM korzysta z gradientów jak z przyspieszenia. Wielokrotne stosowanie tych samych próbek może znacznie przyspieszyć działanie tego algorytmu. W przypadku PPO nie ma obaw o niestabilny proces uczenia, ponieważ ten algorytm jest do tego przystosowany. Często można więc wielokrotnie iterować na tym samym *batch*'u lub zestawie uczącym. Przykładowo, mając 2048 próbek w *batch*'u, można uruchomić 40 kroków uczących bez obaw o zniszczenie polityki. 10 iteracji na każde 512 próbek to dobry współczynnik dla problemów jakie testowano. Wartość ta może być zawsze dostosowywana, jeżeli istnieją obawy co do zbyt szybkiego lub niestabilnego uczenia.

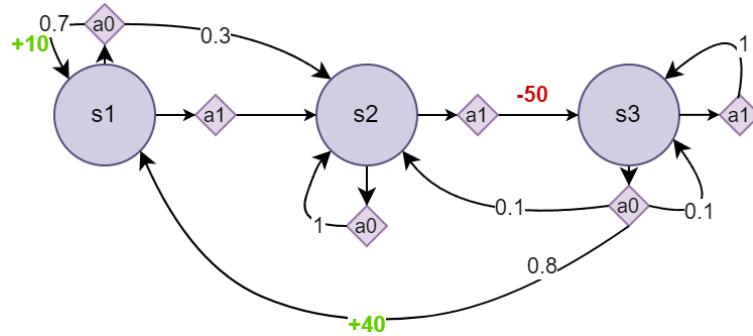
Dobór parametrów związanych z ciekawością stanowi również trudne zadanie. Jedynym sposobem na ich właściwe ustalenie jest iteracyjne i empiryczne dostosowanie do konkretnego problemu. Współczynnik ciekawości jest uzależniony od typowych błędów generowanych

przez sieć neuronową oraz od średnich nagród, jakie otrzymuje. W niektórych badaniach prezentowane jest trenowanie oparte jedynie na nagrodach z ciekawości, co sprawia, że *reinforcement learning (RL)* może przybierać formę procesu uczenia nienadzorowanego co ma zastosowania w niektórych kontekstach. Ostateczna wartość tego współczynnika zależy od stopnia, w jakim chcemy zachęcać aktora do eksploracji.

Wysokie wartości skłaniają aktora do podejmowania częściej suboptimalnych działań lub dążenia do chaosu, podczas gdy niższe wartości sprawiają, że bardziej skupia się na nagrodach pochodzących ze środowiska. W trakcie eksperymentów ustalano ten współczynnik tak, aby nagrody za ciekawość stanowiły od 10% do 50% ogólnych nagród w zależności od siły eksploracji jaką chce się uzyskać. Ponieważ sieć ciekawości i autoencoder są stale w procesie uczenia konieczne jest częste dostosowywanie tego współczynnika w oparciu o obserwacje nagród otrzymywanych przez aktora. Z uwagi na ciągłe modyfikacje wagi tych nagród zostały one usunięte z telemetryki, aby całkowite nagrody nie obejmowały zmiennych nagród za ciekawość.

Współczynnik stopy dyskontowej jest uzależniony od charakterystyki środowiska oraz celu optymalizacji. Istotnym parametrem jest okres półtrwania nagrody (*ang. reward half-life*), który określa ile iteracji jest potrzebnych, aby przyszła nagroda miała 50% wartości w porównaniu do bieżącej. Dla przykładu, przy *discount_rate* równym 0.99 nagrody, które aktor otrzyma za 69 epizodów będą miały wartość bliską 50% bieżącej nagrody. Zmiana tego parametru wpływa na okres, w którym aktor optymalizuje swoje działania ([2], strona 14).

Na przykład, jeżeli zdobycie potencjalnie wysokiej nagrody zajmie 69 epizodów a nagrodą będzie jeden punkt, ale jednocześnie wiązać się będzie z utratą 0.5 punktu to nasz aktor prawdopodobnie nie wybierze tej akcji. Rysunek poniżej przedstawia graf przejść w procesie decyzyjnym Markowa. Posiada on po dwie akcje (a_0 i a_1) wraz z prawdopodobieństwami przejścia w inne stany (s_1 , s_2 i s_3) oraz ewentualnymi nagrodami (+40, -50). Jakie są optymalne akcje dla każdego stanu? Odpowiedź zależy od *discount_rate*. Dla niskich wartości na przykład 0.9 optymalną akcją w s_1 jest a_0 , ale po przejściu do s_2 model preferuje w nim pozostać (wybierając akcję a_0). Dla dużych wartości na przykład 0.99, model preferuje już przejście do s_3 przez wybranie akcji a_1 , mimo że kara za wybór tej akcji jest wyższa od ewentualnej natychmiastowej nagrody (+40).



Rysunek 7: Przykładowy proces decyzyjny Marcova [7]

Wartość tą należy dobrać na podstawie środowiska; warto zacząć od standardowej wartości - 0.99.

11 Eksperymenty

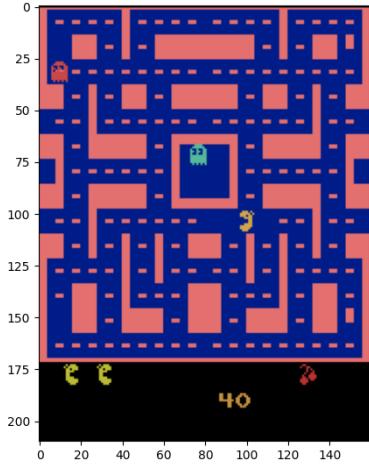
Poniżej przedstawiono przeprowadzone eksperymenty wraz z omówieniem innych prób rozwiązania problemu. Eksperymenty są dostępne w kodzie źródłowym [7] i można odtworzyć wyniki uruchamiając plik z konkretnym eksperymentem i ustawiając odpowiednie wartości hiperparametrów. Kilka wytrenowanych modeli, w tym modele przedstawione poniżej również znajdują się w kodzie źródłowym.

11.1 Pacman

Ms Pacman stanowi jedno z dostępnych środowisk w ramach projektu *OpenAI GYM, ALE* ([13]). Jest to klasyczna gra konsolowa *Atari* wydana w 1982 roku. Posiada ona cechy, które czynią ją idealnym środowiskiem do testowania i rozwijania algorytmów uczenia przez wzmacnianie. Proste obserwacje w postaci klatek z gry oraz niewielka liczba podstawowych akcji (9 możliwych) sprawiają, że jest atrakcyjna do eksperimentowania. Niemniej jednak skuteczne nawigowanie w grze i opracowywanie strategii wymagają zrozumienia kilku złożonych aspektów mechaniki gry a także umiejętności łączenia powiązań między różnymi elementami.

Gra *Ms Pacman* jest wymagająca dla algorytmów ze względu na konieczność wysokopoziomowego planowania i przetwarzania złożonych danych wizualnych. Przykładowo, należy umiejętnie poruszać się po alejkach zbierając punkty i planując trasę jednocześnie unikając przeciwników lub aktywnie atakując w przypadku zdobycia bonusu, bazując jedynie na klatkach z gry. To sprawia, że gra staje się nie tylko testem algorytmów, ale również wyzwaniem dla zdolności planowania i podejmowania skomplikowanych decyzji przez sztuczną inteligencję.

Na samym początku algorytmy były testowane na podstawie surowych klatek z gry, które są kolorowymi obrazkami o rozmiarze 210 na 160, tworząc tensora o wymiarach (210, 160, 3). Poniższy rysunek przedstawia obserwację bez żadnych modyfikacji. Teoretycznie algorytm mógłby operować jedynie na takich surowych danych, jednakże byłoby to znaczne marnotrawstwo zasobów i znaczco wydłużyłoby proces treningu.

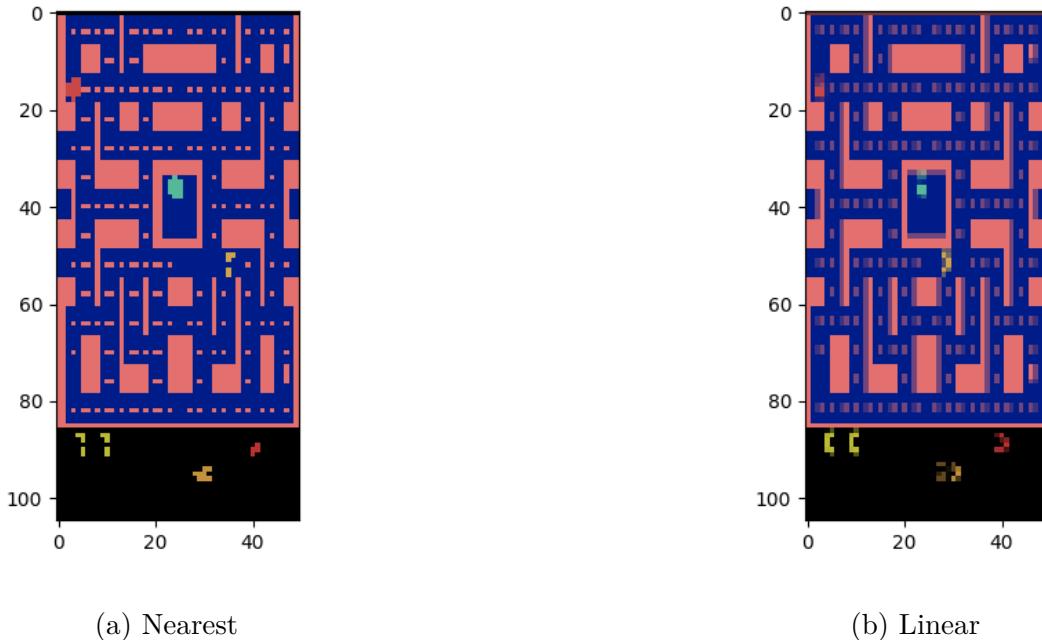


Rysunek 8: Niemodyfikowana obserwacja środowiska Pacman [7]

Z powodu nadmiarowej ilości informacji, które nie są istotne dla skutecznego podejmowania decyzji przez komputer, zdecydowano się szybko zmniejszyć rozdzielcość przy użyciu interpolacji, takiej jak *NEREAST* lub *LINEAR*, do wymiarów 50 na 105 pikseli. Poniżej zaprezentowano pobrane obserwacje z zastosowaniem interpolacji liniowej i bliskościowej. Ostatecznie wybór padł na interpolację bliskościową, głównie ze względu na lepszą estetykę ([1], strona 5).

Niestety, mimo dokonanej pierwszej redukcji ilości informacji okazało się, że to nadal nie jest wystarczająca ilość. Informacje dotyczące kolorów nie są istotne (z kilkoma wyjątkami) a ponadto istnieje potrzeba uwzględnienia kontekstu dla danej klatki z uwagi na niepełne informacje w grze Pacman. Bez uwzględnienia kontekstu niektóre aspekty stanu nie dały się oszacować, takie jak historia ruchu gracza (w niektórych przypadkach), ruch duszków czy nawet ich pozycja, ponieważ nie są one widoczne w każdej klatce.

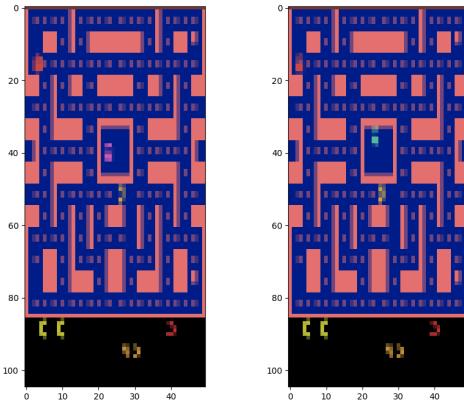
Jednym z zaawansowanych rozwiązań jest zastosowanie sieci neuronowych przystosowanych do pracy z seriami czasowymi, na przykład rekurencyjnych. Jednak prostszym a również efektywnym rozwiązaniem przyjemnym do trenowania jest po prostu modyfikacja obserwacji poprzez dodanie poprzednich klatek. Daje to agentowi kontekst na podstawie którego może lepiej dobierać optymalne akcje.



Rysunek 9: Interpolacja obserwacji [7]

W nowej wersji, wejście do sieci składa się z połączenia poprzedniej i bieżącej klatki. Dzięki takiemu podejściu model posiada kontekst, który pozwala mu zrozumieć, w którą stronę poruszają się obiekty co umożliwia odczytanie pewnych informacji temporalnych. Poniżej przedstawiono rysunek ilustrujący taką obserwację, składającą się z dwóch kolejnych klatek.

To rozwiązanie okazało się tak skuteczne, że kolejnym etapem było całkowite pozbycie się informacji o kolorze poprzez zamianę na skalę szarości. Dodatkowo zdecydowano się na połączenie aż 6 kolejnych klatek. W tym kontekście została również wycięta dolna część klatki, ponieważ zawiera ona niewiele przydatnych informacji (na pewno nieproporcjonalnie dużo jak na obszar o wymiarach 1000 pikseli).



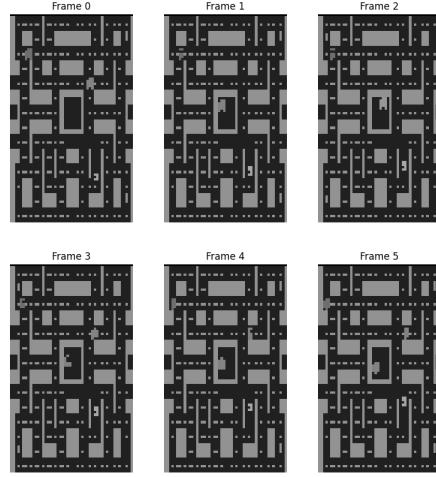
Rysunek 10: Obserwacja składająca się z poprzedniej i bieżącej klatki [7]

Niestety, takie przekształcenie powoduje, że obiekty stają się trudne do odróżnienia. Dlatego zamiast tego obserwacje transformowano do przestrzeni barw *HSV* (*ang. Hue Saturation Value*) a następnie pobrano kanał *Hue*. Dzięki temu eliminuje się 66% informacji jednocześnie zachowując większość istotnych danych.

Taka modyfikacja obserwacji znacznie przyczyniła się do osiągnięcia znacznie lepszych wyników w porównaniu do standardowej jednoklatkowej obserwacji. Dodatkowo przyspieszyła proces uczenia poprzez znaczną redukcję rozmiaru obserwacji, co było głównym celem optymalizacji. Istotnym aspektem tego było także zmniejszenie obciążenia bufora odtwarzania, który musi przechowywać dwie takie obserwacje na każdą próbę uczącą.

11.1.1 Sesje uczenia

Pacman był najintensywniej testowanym środowiskiem. Eksperymenty można podzielić na trzy grupy: *DQN* (v1, v2), *PPO* (v3, v4) oraz *PPO* z dodatkiem ciekawości (*ang. curiosity*) (v5, v6). Poniżej przedstawione są wyniki eksperymentów wraz z omówieniem przebiegu procesu uczenia.



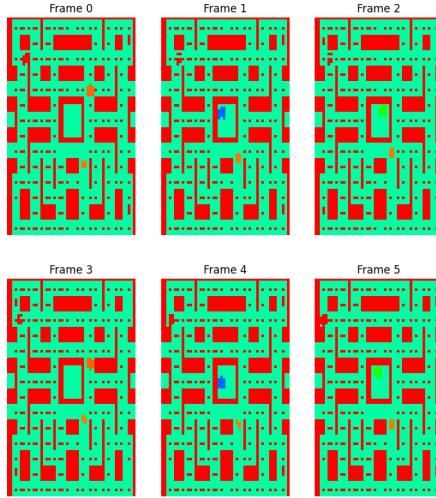
Rysunek 11: Obserwacja składająca się z sześciu poprzednich klatek, skala szarości [7]

11.1.2 DQN

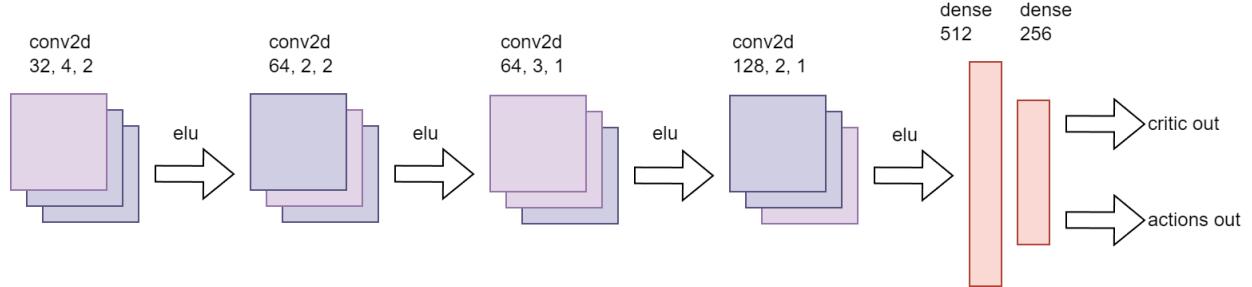
Trenowanie było podzielone na sesje, a wyniki każdej sesji dostępne są w logach. Pierwsza zakończona sesja została oznaczona numerem *20231110-191738v1.0*. Algorytm w tej wersji był najbardziej niedostrojony a proces trenowania był dość nieudany. Poniżej przedstawiono istotne parametry uczenia z tej sesji:

- **lr:** 0.003
- **observation_space:** (84, 84, 3) [jedna klatka, kolor]
- **discount_rate:** 0.99
- **eps_decay_len:** 100
- **batch_size:** 128
- **train_interval:** 4
- **target_update_freq:** 100

W ramach tego eksperymentu użyto jednej sieci neuronowej, która pełniła zarówno rolę aktora jak i krytyka. Sieć przyjmowała na wejście obserwacje, które były przetwarzane przez serię warstw konwolucyjnych a następnie przechodziły przez kilka warstw w pełni połączonych.



Rysunek 12: Obserwacja składająca się z sześciu poprzednich klatek, informacja o barwie [7]

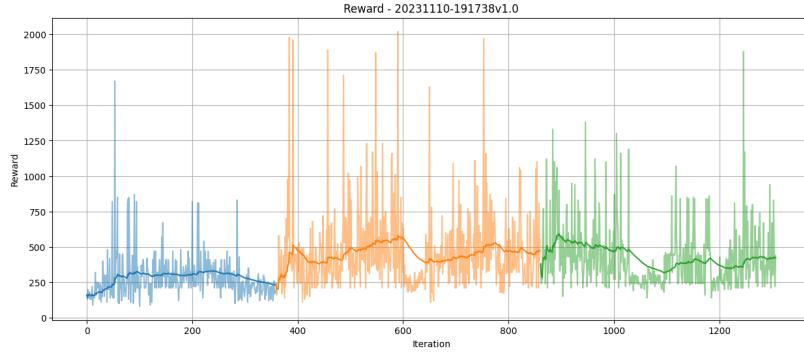


Sieć Neuronowa 1: Sieć neuronowa dla aktora i krytyka, dla eksperymentów w wersji v1 (DQN). Opis architektury warstw przedstawiono w formie (*filters, kernel_size, strides*) lub ilość neuronów dla warstw gęstych. [7]

Algorytm z zastosowanymi ustawieniami nie osiągnął wyników znacznie lepszych od losowego, co odzwierciedla się w przebiegu procesu uczenia przedstawionym na wykresie 6. Problemem wydaje się być zbyt mała skala dla tego konkretnego środowiska. Aby algorytm działał skutecznie konieczne byłoby stopniowe uczenie na dużej ilości danych.

W kolejnym eksperymencie oznaczonym jako *20231111-204750v1.3* zauważalne jest istotnie wolniejsze tempo procesu uczenia. Dokonano zwiększenia parametru *batch_size* oraz wydłużono czas agresywnej losowej eksploracji, co umożliwiło agentowi dłuższy okres wstępnej nauki. Poniżej przedstawiono istotne parametry uczenia z tej sesji:

- **lr:** 0.0002 do 0.0001
- **observation_space:** (84, 84, 3) [jedna klatka, kolor]
- **max_steps_per_episode:** 1000
- **eps_decay_len:** 5000



Wykres 6: Nagrody dla algorytmu w wersji v1 [12]



Animacja 1: Przykład działania modelu v1, link: v1.0_20231110-191738

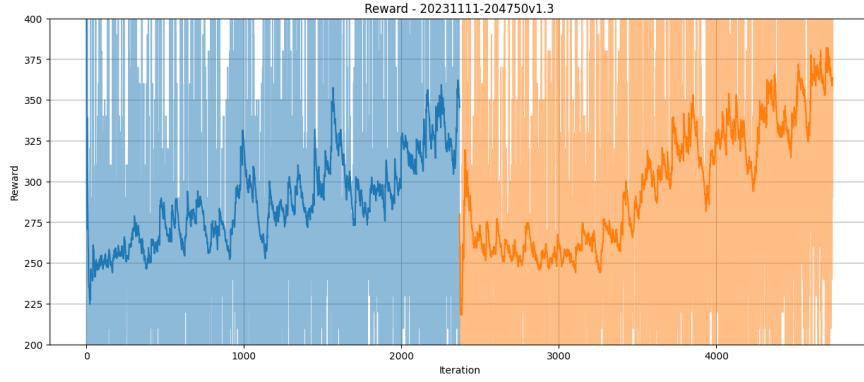
- **batch_size:** 2048
- **iters_per_episode:** 20
- **mini_batch_size:** 128
- **train_interval:** 2
- **target_update_freq:** 100

Architektura sieci jest taka sama jak w pierwszym eksperymentie.

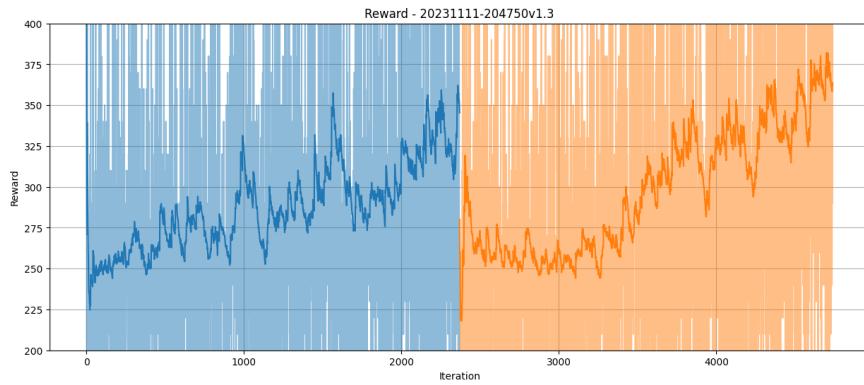
Sesja (Wykres 7) nie trwała wystarczająco długo, aby agent osiągnął minimalny epsilon, jednak pojawił się liniowy wzrost. To dowodzi, że algorytm działa i jest w stanie optymalizować swoje akcje w celu maksymalizacji zysków. Różne kolory reprezentują różne sesje trenowania tego samego agenta. Epsilon spada relatywnie do początku sesji, dlatego można zauważać spadek między dwoma sesjami a następnie liniowy wzrost. W tej wersji algorytm osiągnął średnio nawet 375 punktów. Wykres 8 (poniżej) przedstawia inną skalę, ukazującą maksymalne nagrody jakie agent zdobywał.

Wersja v1.4 to etap dłuższego uczenia przy zachowaniu podobnych ustawień. Nastąpiły jednak pewne modyfikacje w zakresie obserwacji – wprowadzono drugą klatkę do obserwacji w celu dostarczenia agentowi dodatkowego kontekstu czasowego. Poniżej przedstawiono istotne parametry uczenia z tej sesji:

- **observation_space:** (50, 50, 6) [dwie kolorowe klatki]



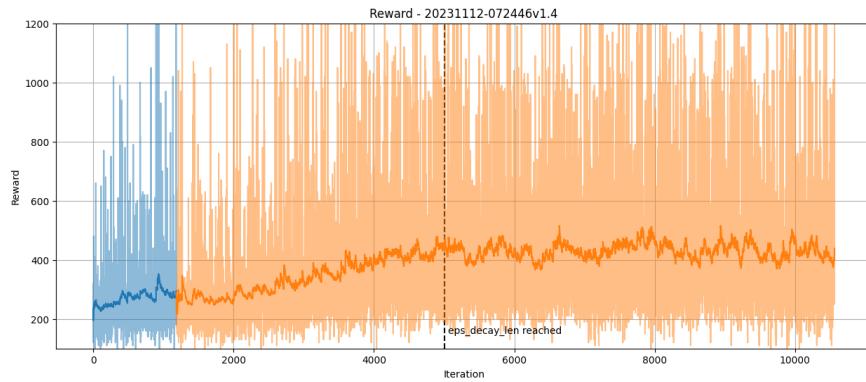
Wykres 7: Nagrody dla algorytmu w wersji v1.3 [12]



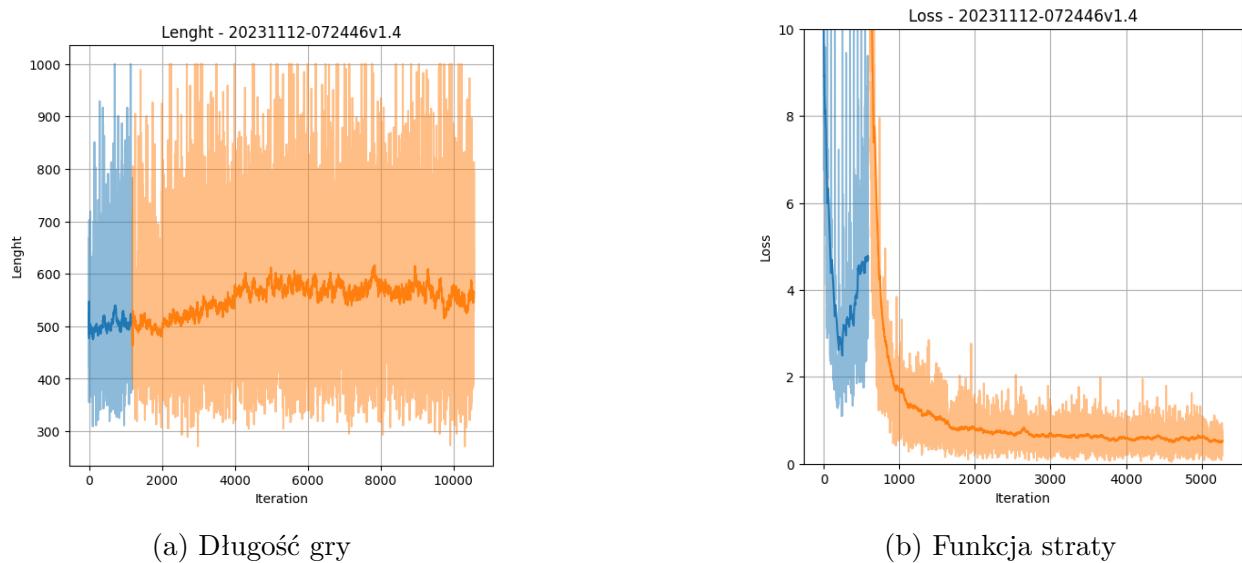
Wykres 8: Nagrody dla algorytmu w wersji v1.3 [12]

- **discount_rate:** 0.8
- **eps_decay_len:** 4000
- **target_update_freq:** 150
- **lr:** 0.0002

Architektura sieci nie uległa zmianie. Tym razem proces trenowania został uruchomiony na ponad 10 000 iteracji a aktor osiągnął średni wynik trochę powyżej 400, co można zaobserwować na Wykresie 9. Dodatkowo, na Wykresach 10 przedstawiono odpowiednie przebiegi wartości funkcji straty i długości gry. Zauważalny jest niewielki wzrost, co wskazuje na lepsze umiejętności omijania "duszków".



Wykres 9: Nagrody dla algorytmu w wersji v1.4 [12]



(a) Długość gry

(b) Funkcja straty

Wykres 10: Inne metryki [12]

Po dostosowaniu hiperparametrów algorytm jest teraz zdolny do wypracowania nieco bardziej efektywnej strategii przeszukiwania mapy. Dodatkowo, zaobserwowano częste zachowanie polegające na dążeniu do jednego z rogów w celu zdobycia bonusu a następnie oczekiwania, aż duszek zbliży się umożliwiając aktorowi zdobycie 1000 punktów. Ta strategia jest często obserwowana i może znaczaco spowolnić postęp aktora.



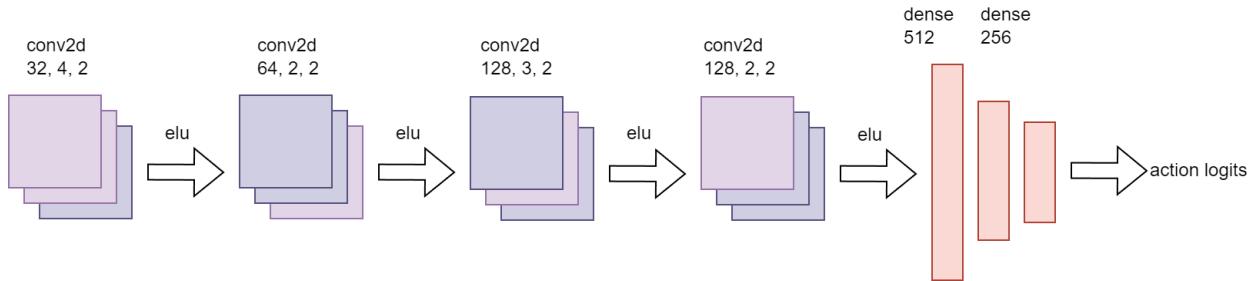
Animacja 2: Rozgrywka modelu v1.4, link do animacji: [1.4_20231112-072446](https://1drv.ms/v/s!AqBzJLcOOGQy4HfjPmIuXWzvCzU)

11.1.3 PPO

Pierwsze próby z zastosowaniem algorytmu PPO przyniosły znaczco lepsze wyniki w porównaniu do DQN. Ten algorytm wymaga znacznie mniej dostosowań a jego parametry mają mniejszy wpływ na stabilność procesu uczenia. Poniżej przedstawiono parametry użyte w ramach pierwszego eksperymentu (v3):

- **actor_lr:** 0.0001
- **critic_lr:** 0.0003
- **observation_space:** (85, 50, 6) [dwie klatki, kolorowe]
- **max_steps_per_episode:** 1000
- **discount_rate:** 0.99
- **clip_ratio:** 0.2
- **lam:** 0.95
- **batch_size:** 1024
- **train_interval:** 1
- **iters:** 1

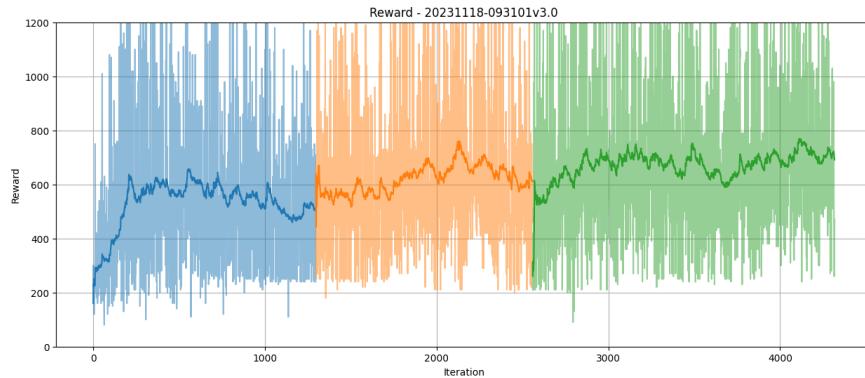
Model sieci dla algorytmu PPO został poddany zmianom. Nowa architektura wyróżnia się przede wszystkim lepiej rozłożonymi wagami w porównaniu do poprzedniej a także charakteryzuje się mniejszą liczbą parametrów. Dodatkowo, dokonano separacji sieci krytyka i aktora. Oba modele posiadają tę samą strukturę, jednak wyjście sieci krytyka zostało zastąpione pojedynczą wartością (przewidywaniem zwrotu). Nowa sieć krytyka zawiera około 1 miliona parametrów, co sprawia, że nie jest nadmiernie rozbudowana.



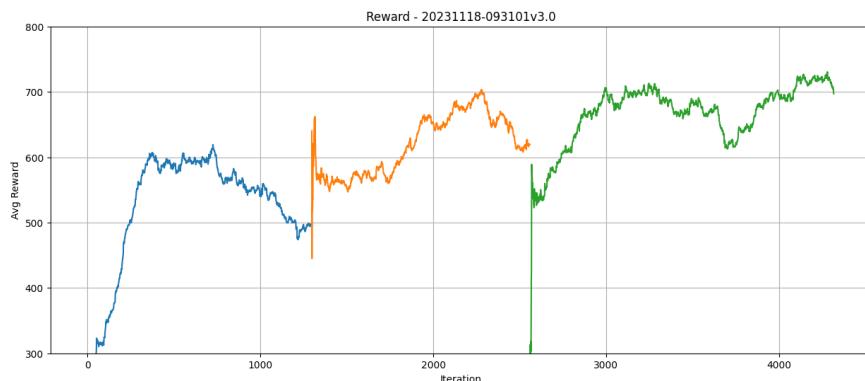
Sieć Neuronowa 2: PPO (v3, v5 i v6) [8]

Model był trenowany podczas kilku sesji a podane wartości odzwierciedlają jego początkowe osiągnięcia. W trakcie kolejnych etapów eksperymentu zdecydowano się zmniejszyć wartość lr na $7e-05$, jednocześnie modyfikując $batch_size$ do wartości 2048 i 4096. Wybór niskiego lr wynikał z konieczności przeprowadzenia znacznej liczby przebiegów uczących w trakcie tego eksperymentu (uczenie po każdej iteracji).

Te parametry przyczyniły się do osiągnięcia dobrych wyników. Średnie nagrody wykazują tendencję wzrostową (Wykres 11) co jest jeszcze wyraźniej widoczne na średniej ruchomej z oknem 200 (Wykres 12).



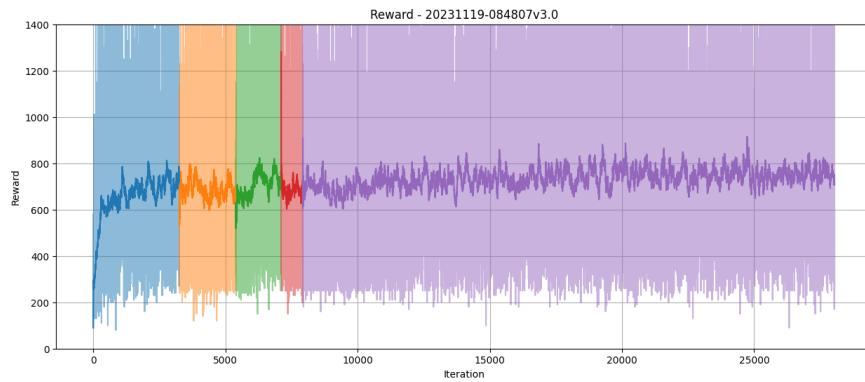
Wykres 11: PPO (v3) - Nagrody w przebiegu uczenia [12]



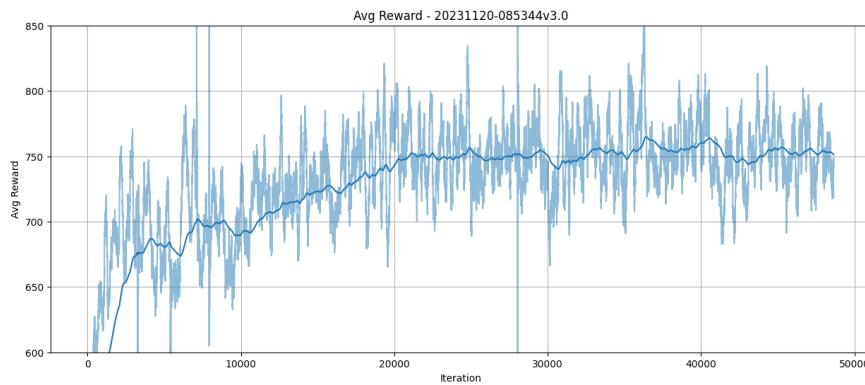
Wykres 12: PPO (v3) - Średnie nagrody w przebiegu uczenia [12]

W związku z powyższym, w kolejnym etapie eksperymentu wznowiono proces uczenia i kontynuowano go przez kilkadziesiąt tysięcy kolejnych epizodów. Ze względu na możliwość występowania szumu, trend może być trudny do zauważenia na wykresie 13. Dlatego też na wykresie 14 zastosowano intensywniejsze wygładzanie. Poniżej przedstawiono istotne parametry uczenia z tego etapu sesji:

- **actor_lr:** 3e-06
- **critic_lr:** 0.001
- **discount_rate:** 0.995
- **lam:** 0.97
- **batch_size:** 1024
- **train_interval:** 1
- **iters:** 1



Wykres 13: PPO (v3) Kontynuacja uczenia [12]

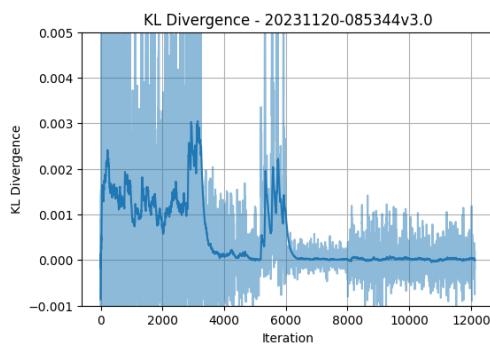


Wykres 14: PPO (v3) Kontynuacja uczenia, silne wygładzanie [12]

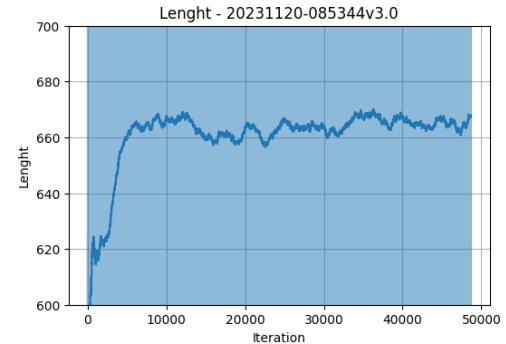
Algorytm PPO niemal osiągnął średnią wynoszącą 800 punktów. Możliwe, że zwiększenie wartości *batch_size* mogłoby jeszcze bardziej usprawnić proces uczenia w tej iteracji.

Niemniej jednak, osiągnięcie średniej na poziomie 800 punktów to rezultat znaczaco przewyższający zarówno heurystykę losową, jak i agenta trenowanego przez DQN.

Poniższe wykresy przedstawiają inne metryki związane z procesem uczenia. Długość rozgrywki ustabilizowała się po około 5000 epizodach utrzymując się na lekkim poziomie oscylacji. To zjawisko wynika z ustabilizowania polityki sieci, gdzie model osiągnął równowagę pomiędzy eksploracją korytarzy, unikaniem duszków a ich schwytaniem. Współczynnik KL (ang. *Kullback-Leibler*) wskazuje, że po 6000 iteracjach trenowania można by było zakończyć proces uczenia, choć warto zauważyc, że współczynnik ten zaczął lekko wzrastać po przekroczeniu 8000 iteracji.



(a) Współczynnik KL, który osiąga bardzo niskie wartości pod koniec przebiegu co świadczy o pewności w podejmowaniu akcji



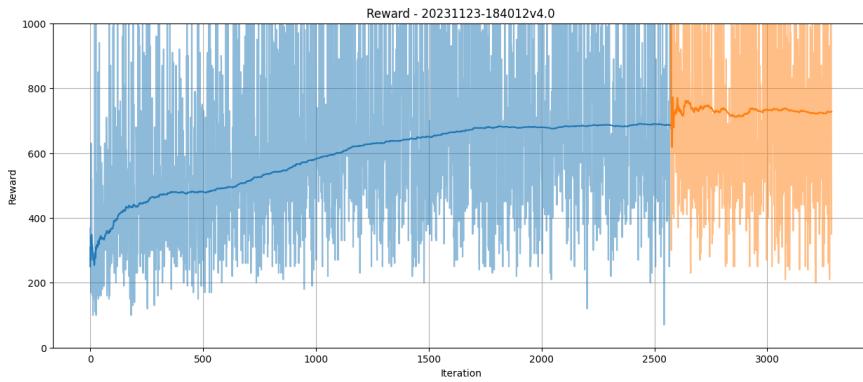
(b) Funkcja straty

Wykres 15: Inne metryki w sesji v3 [12]

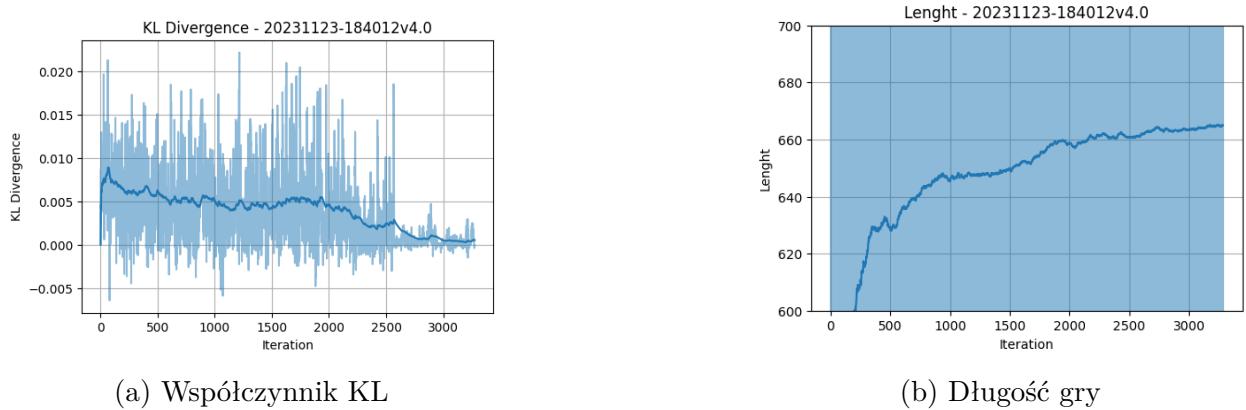


Animacja 3: Rozgrywka modelu v3.0, link: v3.0_20231120-085344

W wersji 4 dokonano zamiany obserwacji z dwóch klatek na obserwację sześciu w których każda zawiera jedynie informacje o kolorze. Dodatkowo, dzięki właściwemu dobraniu hiperparametrów aktor osiągnął wyniki zbliżone do wersji v3 ale znacznie szybciej. Pierwsze etapy trenowania przeprowadzono przy początkowych wartościach lr wynoszących $1e-06$ (dla aktora) i $3e-05$ (dla krytyka), które później zostały zmniejszone. Wartość *batch_size* była modyfikowana w zakresie od 1024 do 4096 . Proces trenowania przebiegł sprawnie i stabilnie, choć sesja była zbyt krótka, aby pełni miarodajnie porównać ją do eksperymentu v3. Uzyskana średnia liczba punktów jest zbliżona do rezultatów uzyskanych w poprzednim eksperymencie.



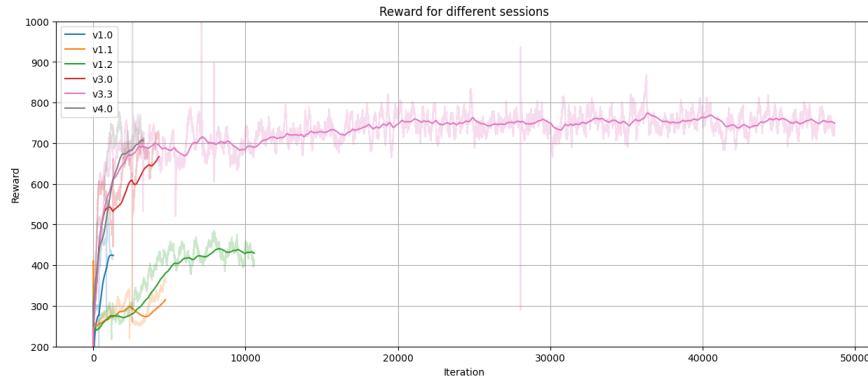
Wykres 16: PPO (v4) [12]



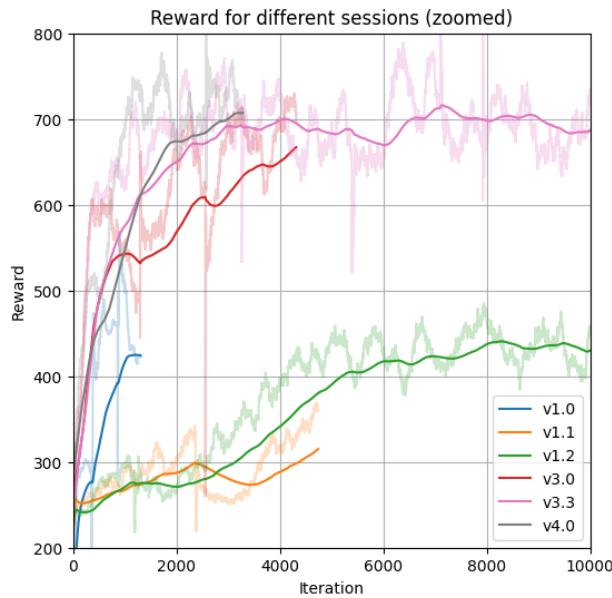
Wykres 17: Inne metryki w sesji v4 [12]

11.1.4 Porównanie przebiegów uczących (v1,v1.3,v3,v4)

Poniższe wykresy prezentują porównanie dotyczących omawianych modeli. Algorytmy PPO wykazują wyraźnie lepsze wyniki, utrzymując się w zakresie 700-800 punktów, podczas gdy DQN osiągnął maksymalnie 450. Istnieje prawdopodobieństwo, że po optymalnym dostrojeniu parametrów i dopracowaniu algorytmów, oba mogłyby uzyskać znacznie lepsze wyniki. Jednak do efektywnego działania DQN wymaga precyzyjnego dobrania hiperparametrów i powolnego, stabilnego uczenia opartego na bardzo dużej ilości próbek i trajektorii.



Wykres 18: PPO (v4) [12]



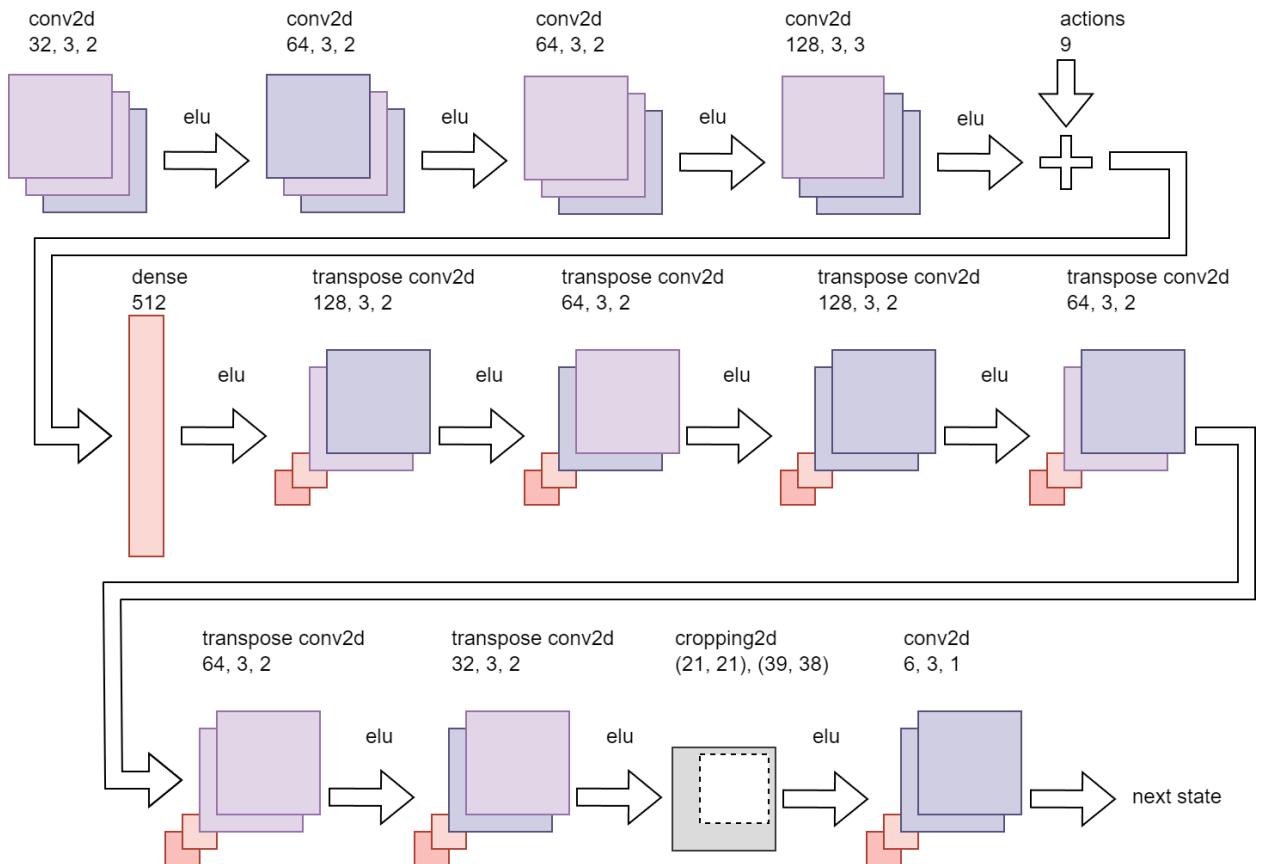
Wykres 19: PPO (v4) [12]

11.1.5 Curiosity (plain pixels)

W eksperymencie v5 algorytm ciekawości został zrealizowany poprzez próbę maksymalizacji błędu sieci przewidującą następny stan na podstawie poprzedniego. W oryginalnej pracy sugerowano, że rozwiązanie przewidujące wartości pikseli może być nieoptymalne a zalecano zastosowanie autoenkodera w celu eliminacji potencjalnych problemów. W tym badaniu jednakże zdecydowano się na użycie pojedynczej sieci ze względu na prostotę takiego podejścia. Kolejny eksperyment obejmuje implementację, która korzysta z autoenkodera.

Architektura sieci ciekawości różni się od struktury poprzednich modeli. Przypomina autoenkoder, ale w warstwach gęstych dodano wektor akcji. Liczba parametrów tej architektury jest nieco większa niż w przypadku sieci krytyka i aktora osiągając około 1.1 miliona.

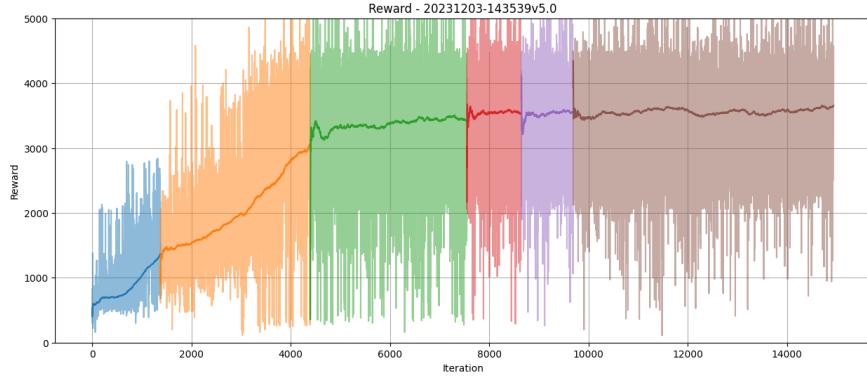
Składa się ona z warstw konwolucyjnych, których wyniki są przekazywane przez warstwy w pełni połączone, dodane w celu ułatwienia integracji z wektorem akcji. Następnie taki wektor jest przetwarzany przez warstwy konwolucyjne transponujące stopniowo zwiększące rozdzielczość do rozmiaru nieznacznie przekraczającego obserwacje. Warstwa wycinająca fragment (*ang. cropping*) została również zastosowana w celu dostosowania kształtu do wymagań obserwacji.



Sieć Neuronowa 3: Sieć ciekawości (conv→dense → deconv) liczby nad warstwami określają parametry dla warstw (conv - (filters, kernel, stride), dense (units), crop - (top, bottom), (left, right)) [8]

Trening przeprowadzono w kilku sesjach, które łącznie trwały trzy dni (1x RTX 3080). Dokładne wartości hiperparametrów znajdują się w kodzie źródłowym, lecz ogólnie *batch_size* wynosił 4096 , *lr* dla aktora oscylował między $1e-05$ a $1e-06$, a *lr_krytyka* było nieco większe, mieściło się w zakresie od $1e-03$ do $1e-04$. Model intensywnie trenowano, rozpoczynając od 20 iteracji co 5 epizodów. W czwartej sesji zwiększoną liczbę iteracji na 40 , a następnie na 80 co 10 epizodów. Ostatnia sesja obejmowała 100 iteracji w każdym epizodzie. Parametr *curious_coef* był dynamicznie modyfikowany, aby nagrody za ciekawość stanowiły od 10% do 50% całkowitych nagród. Początkowo ustalono go na wartość 0.013 , później zwiększano, gdy sieć ciekawości dostosowywała się do środowiska (0.04), a na końcu osiągnął 0.08 . Wartości te były dostosowywane empirycznie, jednak automatyzacja na podstawie docelowej wartości nagród za ciekawość mogłyby być korzystnym usprawnieniem.

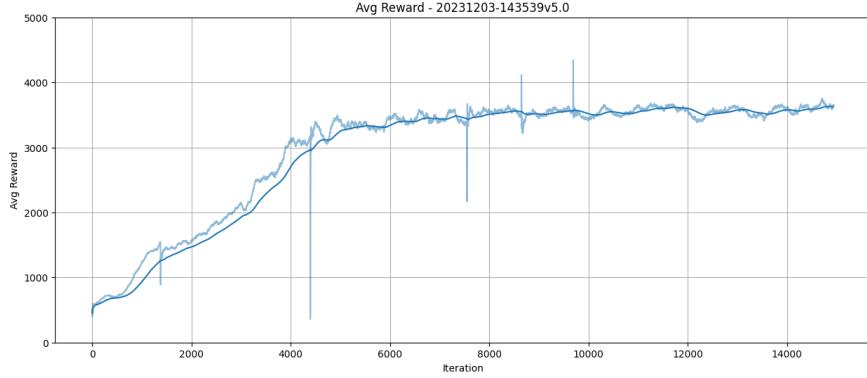
Przebieg uczenia został przedstawiony na *Wykresie 20*. Należy zauważać, że skala tego modelu uzyskała wyjątkowo wysoki wynik średnio osiągając 3500 punktów, co stanowi rezultat znacznie wyższy niż ten uzyskany przez standardowy algorytm PPO. Tak wysoki wynik aktor osiąga poprzez przyjęcie specyficznej strategii, którą znalazł dzięki ciekawości.



Wykres 20: PPO + ciekawość, nagrody uzyskiwane w czasie uczenia (tylko ze środowiska) [12]

Ciekawość zmienia cel optymalizacji umożliwiając aktoriowi większą eksplorację środowiska i zachęcając go do podejmowania normalnie suboptimalnych akcji. To z kolei potencjalnie prowadzi do sytuacji, w których aktor mógłby się nie znaleźć, gdyż droga do nich jest nieoptimalna i trudna do znalezienia przy obecnych nagrodach. Dodatkowo, taka architektura sprawia, że aktor nie jest nagradzany za podejmowanie tych samych akcji eliminując problem zacinania się aktora w niektórych obszarach środowiska.

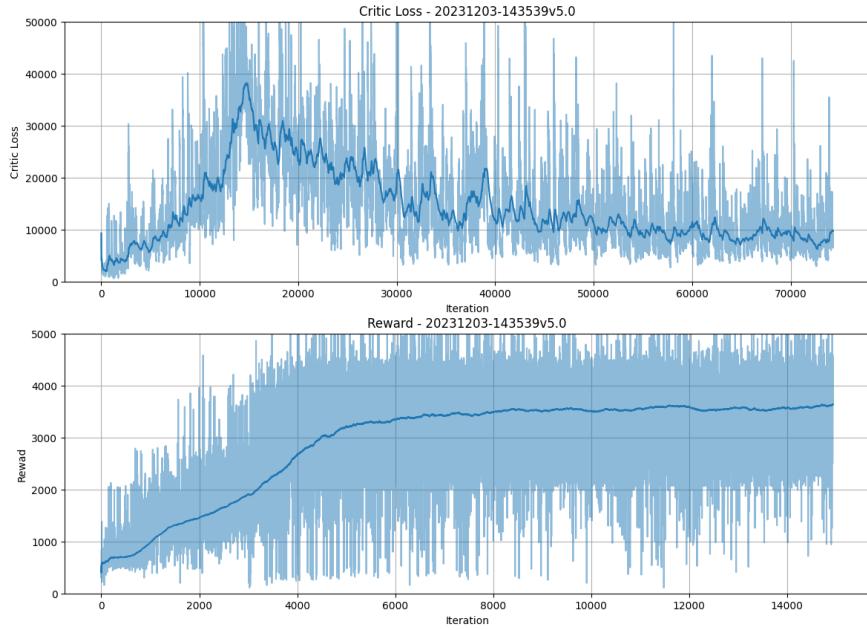
Wzrost nagród jest ciągły, nawet po ustabilizowaniu zbieranych nagród, co sprawia, że aktor kontynuuje naukę, eksplorując nowe taktyki.



Wykres 21: PPO + ciekawość, średnie nagrody uzyskiwane w czasie uczenia (tylko ze środowiska) [12]

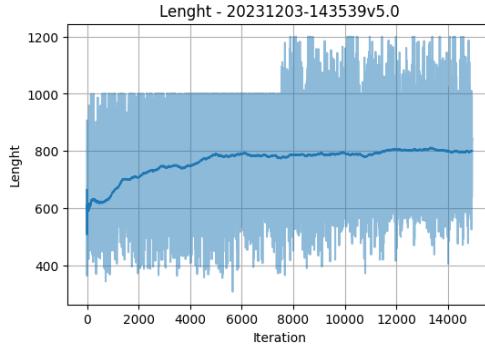
Szybka adaptacja aktora przy użyciu tego algorytmu skutecznie oddaje złożoną dynamikę działania sieci krytyka. W miarę gwałtownego wzrostu nagród, krytyk napotykał znaczne trudności w dokładnej estymacji wartości returns, co jest widoczne w funkcji straty aktora. Dopiero po pewnym czasie, gdy aktor przestał znaczco ulepszać swoją strategię a krytyk nabył nową taktykę, wartości funkcji straty zaczęły spadać.

Prawdopodobnie wydłużenie fazy nauki i dokładniejsze dostosowanie hiperparametrów mogłyby dodatkowo poprawić ten wynik.



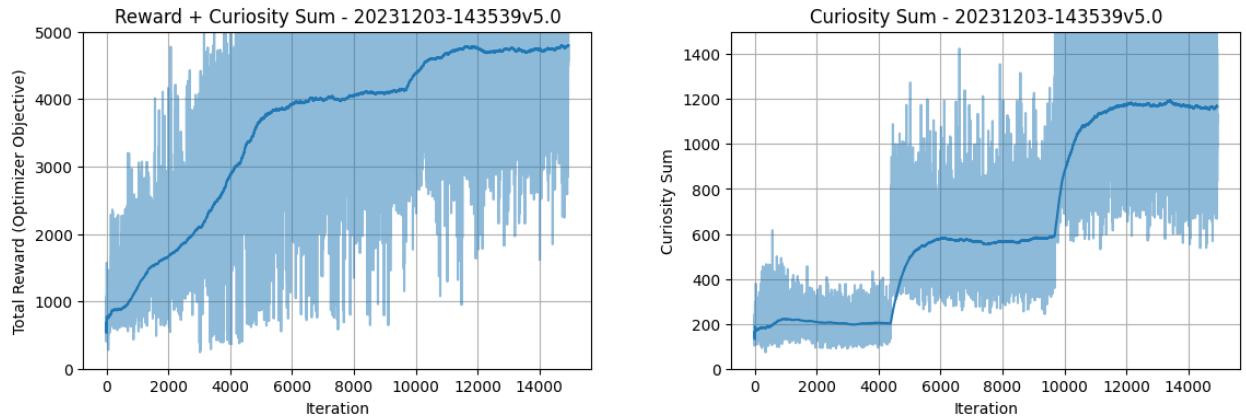
Wykres 22: Uzyskiwane nagrody i funkcja straty krytyka, skale x są różne ponieważ loss krytyka jest obliczany podczas trenowania a nagrody w każdym epizodzie. Skala na x na obu wykresach reprezentuje ten sam czas [12].

Aktor radził sobie tak dobrze, że w połowie procesu uczenia konieczne było zwiększenie wcześniejszej ustalonego limitu 1000 epizodów, ponieważ aktor zbyt często osiągał ten pułap.



Wykres 23: Długość rozgrywki. Nagły skok ponad 1000 to moment zmiany maksymalnej długości epizodu [12]

Celem optymalizacji aktora nie są jedynie nagrody uzyskane ze środowiska, ale również nagrody za ciekawość. Wartości nagród za ciekawość zostały oczywiście usunięte z wcześniejszych przebiegów, ponieważ sprawiłyby to, że byłyby one nieporównywalne do innych eksperymentów. Rzeczywisty cel optymalizacji, czyli suma nagród i ciekawości, został przedstawiony na Wykresie 24a.



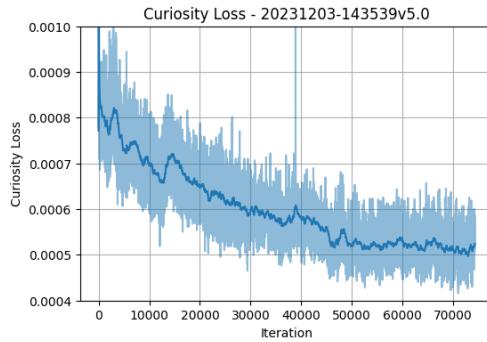
(a) Cel optymalizacji algorytmu (nagrody + ciekawość). Wartości ciekawości są przedstawione na wykresie obok (b)

(b) Długość rozgrywki. Nagły skok ponad 1000 to moment zmiany maksymalnej długości epizodu

Wykres 24: Nagrody za ciekawość [12]

Przebieg ten jest mniej informacyjny, ponieważ wartości są zależne od współczynnika ciekawości, który jest zmienny. Wartości samego współczynnika ciekawości są przedstawione na Wykresie 24b. Nagłe zmiany wartości odpowiadają początkom sesji, w których zmieniano wartość współczynnika c_{coef} . Na początku średnie nagrody miały wartość 200, następnie przez nagłe zwiększenie uzyskiwanych nagród aby aktor nadal utrzymywał swoją ciekawość c_{coef} co odbiło się na wartościach nagród uzyskiwanych przez aktora.

W trakcie procesu uczenia sieć ciekawości ciągle aktualizuje swoje wagi - minimalizując średni błąd. Te średnie wartości są istotne podczas strojenia współczynnika ciekawości. W pierwszych kilku epizodach wartości błędu mogą być bardzo duże, co prowadzi do nie-naturalnie wysokich nagród za ciekawość. Te nagrody maleją, gdy sieć nauczy się podstawa środowiska. Jest to kolejna wada podejścia opartego na przewidywaniu pikseli w kontekście ciekawości.



Wykres 25: Wartości funkcji straty sieci ciekawości [12]

Model osiągnął bardzo dobre wyniki i opracował dość nietypową strategię, w której na początku gry stara się szybko złapać wszystkie duszki. Takie zachowanie nie było obserwowane w przypadku PPO bez ciekawości, a tutaj aktor notorycznie dochodził do podobnego rozwiązania na początku gry (wersja v6, korzystająca z innej wersji ciekawości, również opracowała podobną taktykę).

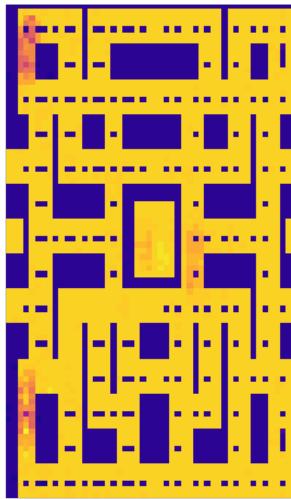
Prawdopodobnie wynika to z lepszych nagród, jakie aktor otrzymuje za ciekawość. Reszta rozgrywki przebiega zazwyczaj normalnie, gdzie model eksploruje mapę i zbiera punkty.



Animacja 4: Rozgrywka modelu v5.0 trenowanego przy użyciu PPO i ciekawości, link: v5.0_20231203-143539

Ze względu na sieć ciekawości, która operuje na obserwacjach, można przeglądać jej predykcje. Poniższa animacja przedstawia klatki generowane przez sieć ciekawości. Można zauważać, że sieć nie jest pewna pozycji dynamicznych obiektów na mapie, co objawia się rozmazaniem. Wszystkie statyczne elementy są wyraźne, ponieważ są nieruchome i łatwe

do przewidzenia. Nauka tych statycznych elementów skutkuje dużym błędem w sieci na początku, który następnie ulega normalizacji.



Animacja 5: Rozgrywka modelu v5.0 trenowanego przy użyciu PPO i ciekawości, link: Animacja predykcji sieci ciekawości

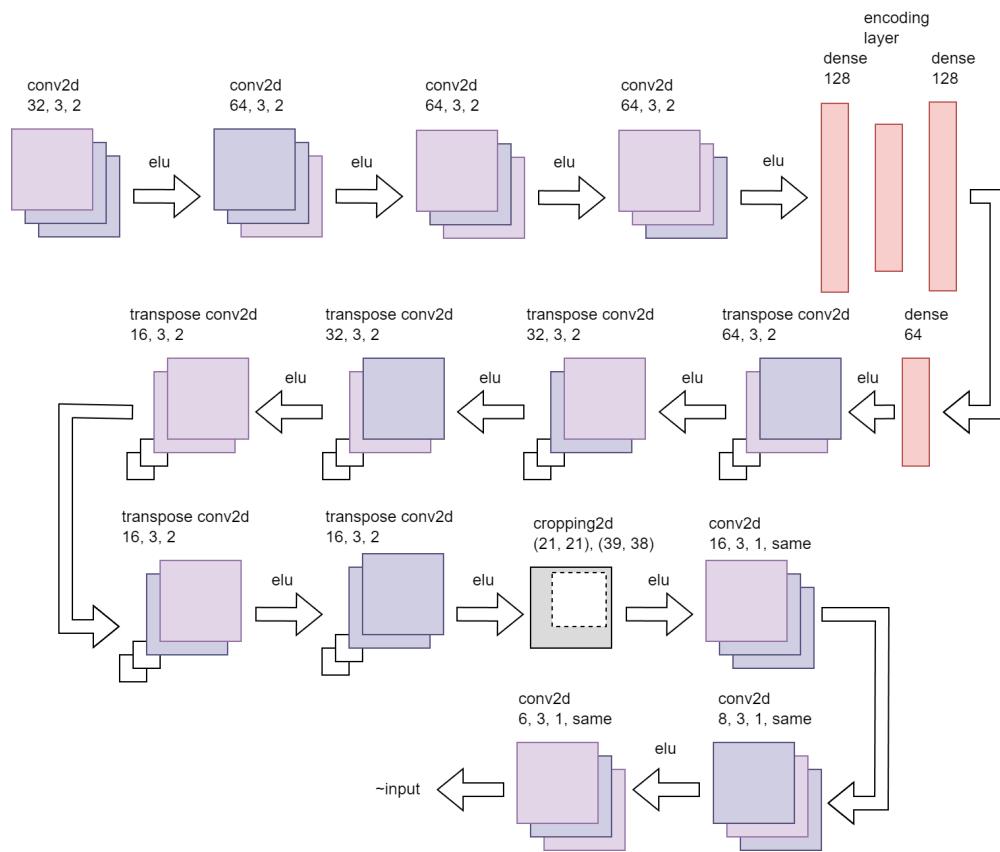
11.1.6 Curiosity (autoencoder)

Wersja v6 korzysta z ulepszonej wersji ciekawości, gdzie sieć ciekawości została rozdzielona na dwie mniejsze sieci - sieć autoenkodera, odpowiedzialną za kodowanie stanu środowiska do wektora oraz sieć ciekawości, która przewiduje następny zakodowany stan środowiska na podstawie aktualnego stanu i wybranej akcji. Dodatkowo, w ramach tego eksperymentu, celem wymuszenia odmiennego zachowania aktora, nagroda za złapanie ducha została zmniejszona z 1000 do 100, co oznacza, że strategia łapania wszystkich duszków na początku gry nie jest już tak opłacalna. Decyzja ta wynika częściowo z chęci wymuszenia taktyki bardziej przypominającej grę człowieka, który skupia się zarówno na eksploracji mapy, jak i zbieraniu punktów, równie mocno jak na atakowaniu duszków. Aktor, ze względu na wysoką nagrodę za atak i niewielką karę preferuje taktyki oparte na agresywnym zbieraniu duszków.

Architektura sieci aktora i krytyka jest identyczna jak w innych eksperymentach. Sieć autoenkodera jest oznaczona numerem 4 i obejmuje zarówno część kodującą, jak i dekodującą. Rozmiar warstwy kodowania to hiperparametr, który można regulować. W pierwotnej pracy proponowano stosowanie autoenkodera wariancyjnego. Tutaj użyto prostego autoenkodera opartego na warstwach konwolucyjnych a następnie dekonwolucyjnych. Część od wejścia do warstwy kodowania to sieć kodująca używana do kodowania stanów środowiska. Posiada ona funkcję aktywacji *tanh* w celu ograniczenia wartości wyjściowych z kodera do zakresu $(-1, 1)$. Trenowanie autoenkodera to problem uczenia nadzorowanego, w którym model aproksymuje swoje wejście.

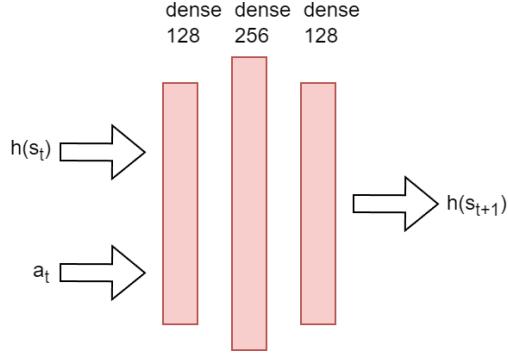
Trenowanie autoenkodera w tej sytuacji niesie ze sobą same korzyści:

- Sprawia, że ciekawość nie jest już tak bardzo zależna od chaotycznych aspektów środowiska, ponieważ istnieje duże prawdopodobieństwo, że sieć skupi się najpierw na elementach prostszych do zakodowania.
- Zakodowany stan jest wektorem, który ma ograniczony zakres wartości, co ułatwia przewidywanie maksymalnego błędu i ustalenie wartości parametru ciekawości.
- Nie ma potrzeby przechowywania całego następnego stanu w celu uczenia sieci ciekawości, ponieważ wystarczy zakodowany stan, który jest znacznie mniejszym wektorem. Ponadto autoenkoder wymaga jedynie bieżącej obserwacji do nauki.



Sieć Neuronowa 4: Architektura autoenkodera [8] [12]

Sama sieć ciekawości posiada bardzo prostą architekturę, która przyjmuje zakodowany stan i wybraną akcję, zwracając zakodowany stan następnej akcji. W eksperymencie v6 stosowano kodowanie o rozmiarze 128.



Sieć Neuronowa 5: Architektura sieci ciekawości bazującej na enkodowanym stanie środowiska [8]

Taka architektura sprawia, że nagrody za ciekawość są bardziej miarodajne i skuteczniej prowadzą aktora. Wiele prac wskazuje, że gęsto rozłożone nagrody pozwalają na skuteczniejszą naukę aktora. Curiosity, w takiej formie, pozwoliłoby nawet na uczenie nienadzorowane, w którym aktor uczyłby się grać w grę bazując jedynie na nagrodach za ciekawość.

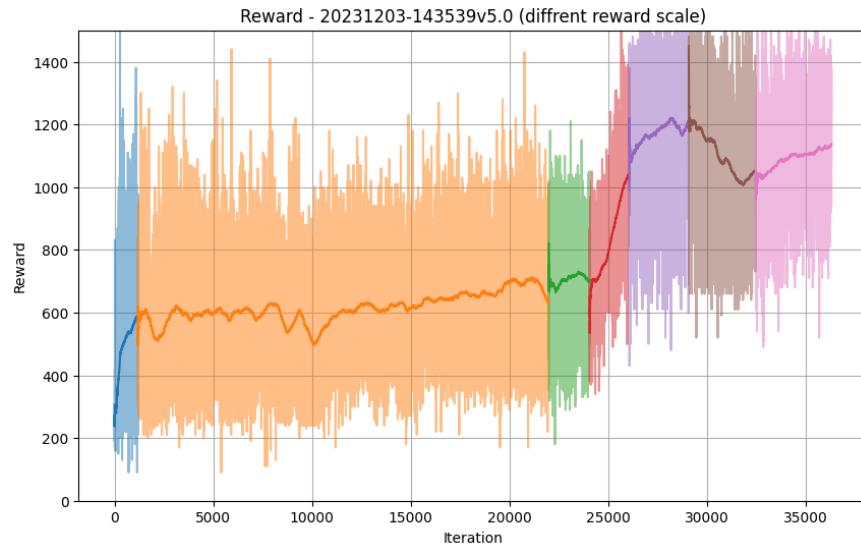
Trenowanie zostało rozpoczęte z poniższymi parametrami:

- *actor_lr*: 0.0001
- *critic_lr*: 0.0003
- *observation_space*: (85, 50, 6) [6 kalatek, barwa]
- *encoding_size*: 128
- *max_steps_per_episode*: 1200
- *discount_rate*: 0.99
- *clip_ratio*: 0.2
- *lam*: 0.98
- *curius_coef*: 2.0
- *batch_size*: 4096
- *batch_size_curius*: 300
- *train_interval*: 1
- *iters*: 1
- *iters_curious*: 10

Należy zwrócić uwagę na kilka istotnych faktów dotyczących tych hiperparametrów: *batch_size* wynosił 4096 i nie zmieniał się przez cały proces uczenia. *Curiosity* posiada mniejszy *batch*, ponieważ ta sieć wymaga dużej ilości pamięci, a duże *batche* znaczaco zwiększały jej użycie. Jako rekompensata wykonuje proporcjonalnie więcej iteracji uczenia w każdym eposodzie. Parametr *actor_lr* jest również relatywnie niski. W trakcie uczenia był kilkukrotnie zmniejszany: $1e-05 \rightarrow 3e-06 \rightarrow 1e-06$. Sytuacja wyglądała podobnie z parametrem *critic_lr*

- spadł maksymalnie do $1e-05$. Parametr *curius_coef* był ustalany na bieżąco tak, aby utrzymać średnie nagrody za ciekawość na poziomie podobnym jak w poprzednim eksperymencie (10%-50% powinny stanowić nagrody za ciekawość). Jego wartość z początkowej 2.0 szybko wzrosła do 4.0 , ponieważ sieć ciekawości ustabilizowała się i wartości straty spadły. Następnie ciekawość była zmniejszana: $1.0 \rightarrow 0.6 \rightarrow 0.8 \rightarrow 0.2$. Zmieniała się również intensywność trenowania (ilość iteracji na epizod). Na początku wynosiła 1 ale potem, aby przyspieszyć uczenie, została zwiększa do 10 . Nie miało to negatywnego wpływu na stabilność uczenia.

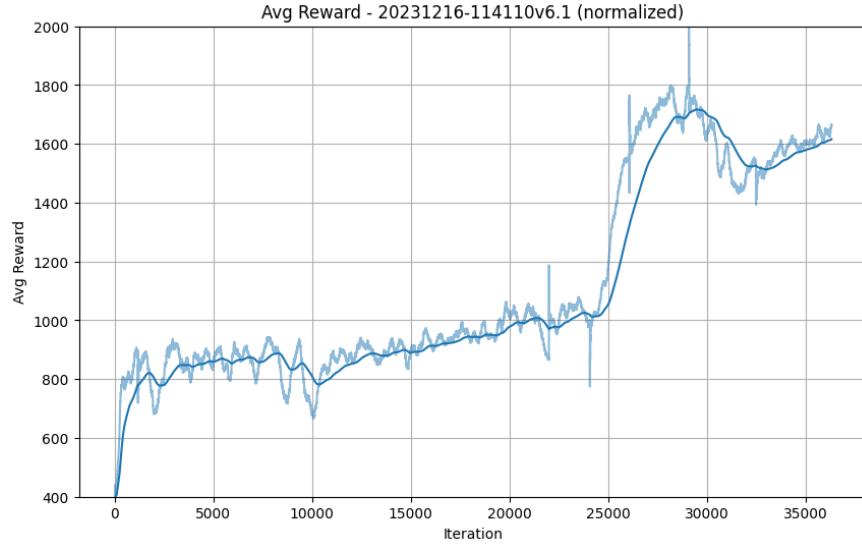
Poniższy wykres przedstawia przebieg trenowania z zaznaczonymi sesjami. Od sesji 4 ilość iteracji została zwiększa do 10 . Nagły spadek nagród (*kolor brązowy*) jest rezultatem gwałtownego wzrostu ciekawości; w tym momencie aktor zaczął priorytetować eksplorację nad zdobywaniem nagród ze środowiska, co skutkowało spadkiem średniej. Dużym błędem było także początkowe powolne tempo uczenia. Wykonywanie większej ilości iteracji nie jest tak ryzykowne jak w przypadku innych algorytmów, ponieważ PPO automatycznie dostosowuje się, co eliminuje obawy dotyczące niestabilności podczas szybkiego uczenia. Warto utrzymywać duży rozmiar partii (*batch_size*), aby aktor i krytyk otrzymywali dokładne średnie trajektorii, co przekłada się na lepsze gradienty i skuteczniejszą politykę. Należy zaznaczyć, że skala na tym wykresie nie odpowiada poprzednim, ponieważ nagrody zostały zmodyfikowane.



Wykres 26: Wartości nagród uzyskiwane przez aktora v6 (skala nagród nie odpowiada poprzednim przebiegom) [12]

Poniższy wykres przedstawia znormalizowane nagrody aktora. Normalizacja została przeprowadzona poprzez ustalenie średnich nagród uzyskanych przez aktora według standardewowych zasad (bez modyfikacji) a następnie przeskalowania wszystkich wartości według nowej normy.

Widać, że aktor osiągnął wysoki wynik, jednak niższy niż poprzedni agent. Wynika to z innego celu optymalizacji, w którym polowanie na duszki nie jest tak opłacalną taktyką.



Wykres 27: Wartości nagród, znormalizowane według zmierzonej wydajności aktora w środowisku. [12]

Zachowanie aktora jest bardziej naturalne. Ten aktor również opracował taktykę łapania dwóch duszków na początku każdej gry. Jego eksploracja mapy jest znacznie skuteczniejsza a gry trwają dłużej niż w przypadku starszych eksperymentów. Wynika to głównie ze zmiany nagród w środowisku, które wymuszają takie zachowanie. Aktor preferuje również alejki z większą ilością nagród.



Animacja 6: Rozgrywka modelu v6.0, link: v6.1_20231214-094337

11.2 Connect4

Connect4 to popularna dwuosobowa gra, której celem jest ułożenie czterech żetonów w linii - pionowej, poziomej lub diagonalnej na planszy o rozmiarze 6x7. Jest bardziej złożona niż kółko i krzyżyk, ponieważ ilość dostępnych strategii i możliwości jest znacznie większa. Żetony można wrzucać jedynie od góry i gry trwają od 8 do 40 epizodów. Wybór tego środowiska wynika z jego prostoty w porównaniu do gier takich jak szachy czy warcaby, co jednak nadal stanowi wyzwanie dla uczenia przez wzmacnianie z użyciem self-play.

Self-play jest niestabilny ze względu na ciągłe zmiany nie tylko aktora, ale także przeciwnika. To stanowi wyzwanie nawet dla zaawansowanego algorytmu PPO, dlatego zdecydowano się zrezygnować z treningu DQN w tym środowisku.

W środowisku Connect4 przeprowadzono dwa eksperymenty. Pierwszy z nich obejmował testowanie różnych wariantów algorytmu, ustawień systemu i metody oceny postępu uczenia. Niestety, w self-play metryka nagród okazała się praktycznie bezużyteczna, ponieważ aktor miał niemal zawsze współczynnik zwycięstw przeciwko sobie równy 50%.

Po kilku iteracjach i eksplorowaniu różnych metod oraz technik, takich jak:

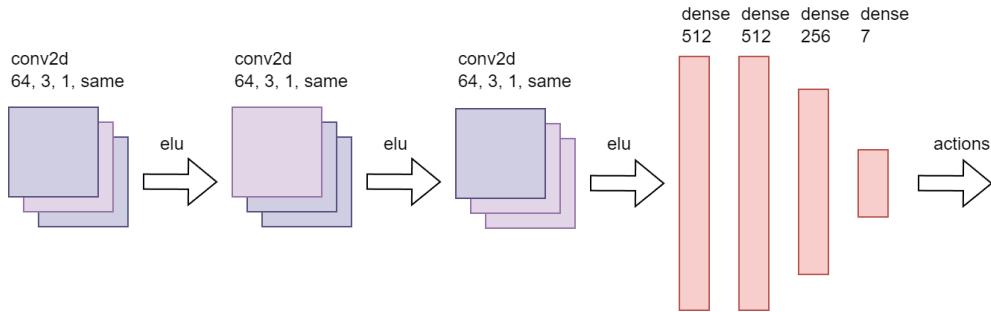
- Uczenie dwóch aktorów i kopowanie lepszego.
- 80% gier przeciwko sobie, 20% przeciwko przeszłej wersji.
- 25% rozgrywek przeciwko losowemu modelowi.
- Różne metody zapisu obserwacji (oddzielne wejście określające stronę dla której aktor powinien wykonać ruch), dodatkowy wymiar składający się z jedynek lub zer w zależności od strony.
- Modyfikacja nagród (szczególnie kara za każdą rozegrana turę).
- Różne metody ewaluacji sieci (winratio przeciwko sobie, winratio w konkretnym stanie środowiska, winratio przeciwko losowemu agentowi, winratio przeciwko agentowi wykonującemu ten sam ruch, itp.)

Ostatecznie system był trenowany w dwóch eksperymentach (v4 i v5). Wersja v4 trenowała jeden model grający po obu stronach i trenowany na doświadczeniu z obu stron. W wersji v5 aktor otrzymywał wielką karę za przedłużanie gier (-0.025 za każdą rozegrana turę, co przy średniej długości gry wynoszącej 20 tur daje sumarycznie karę -0.5 plus -1 lub 1 za wygraną lub przegraną).

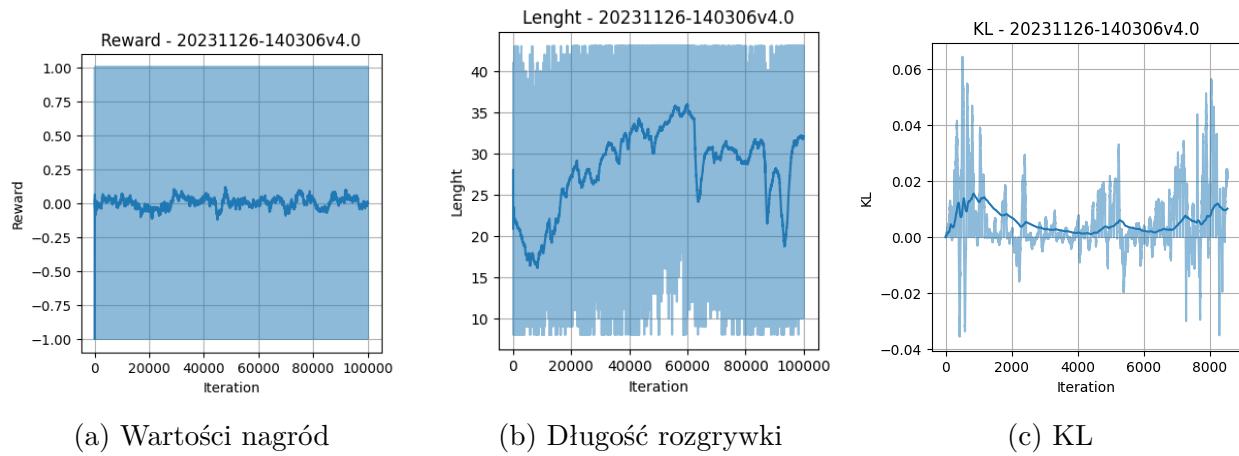
Ogólnie rzecz biorąc, średnie nagrody nie niosą ze sobą żadnej przydatnej informacji, zwłaszcza przy takich nagrodach. Algorytm będzie zawsze oscylował w okolicy 0, ponieważ taka jest średnia wartość nagród. Drobne wahania wynikają z procesu uczenia, w którym aktor zaczyna wybierać bardziej optymalne strategie.

Znacznie lepszą metryką jest długość gry. Dłuższe gry świadczą o tym, że model rzadziej popełnia błędy, które wcześniej prowadziły do zakończenia gry.

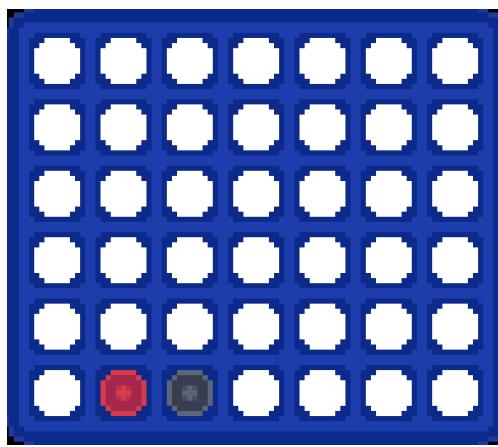
Sieci neuronowe użyte w tym zadaniu są dość małe. Zarówno sieć krytyka, jak i aktora, mają podobną architekturę różniącą się jedynie wyjściem. Składają się z kilku warstw konwolucyjnych i warstw w pełni połączonych. Poniżej znajduje się rysunek przedstawiający architekturę sieci neuronowej dla aktora. Sieć krytyka różni się jedynie wyjściem.



Sieć Neuronowa 6: Architektura sieci aktora dla selfplay [8]



Model opracował pewną strategię, niestety jednak wykazuje tendencję do przedłużania gier. Ponadto, proces treningu trwa długo a sam model nie jest zbyt wymagającym przeciwnikiem.



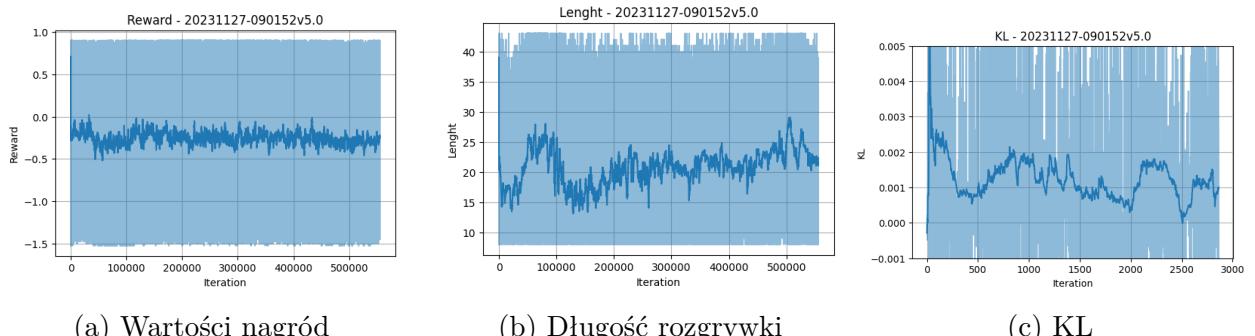
Animacja 7: Rozgrywka modelu v4.0 (Selfplay), link: v4.0_20231126-140306

Wersja algorytmu, oznaczona jako v5 i charakteryzująca się lekko zmienioną architekturą została zoptymalizowana w celu skutecznego przeciwdziałania problemowi przedłuża-

nia gier. Mechanizm ujemnej nagrody na początku skłania graczy do szybkiego zakończenia rozgrywki. W miarę postępu procesu uczenia gracze optymalizują swoje strategie dając do maksymalnej nagrody za wygraną lub przegraną.

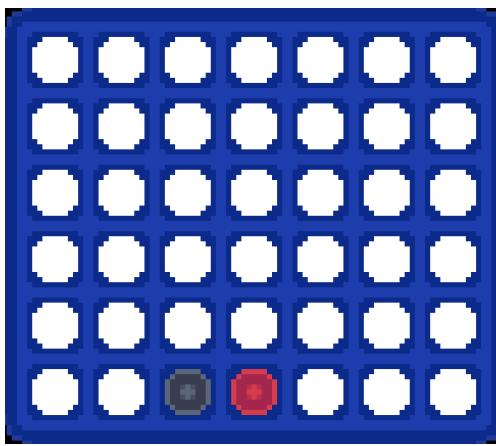
Następnie, w procesie ewolucji, przeciwnik, ucząc się kontratakować inicjuje cykl, który może trwać w nieskończoność tworząc samodoskonalący się system.

Istotnym aspektem jest przesunięcie nagrody w dół, w okolice -0.25 . Wynika to z kary za każdą turę otrzymywana przez każdego gracza. Efektem tego jest próba skrócenia gry na początku (dwukrotny spadek do 15 tur). Następnie, po ustabilizowaniu się sieci, następuje stopniowa zmiana strategii, w której sieć reaguje na zmiany w zachowaniu przeciwnika (środowiska) wydłużając grę pomimo ujemnej nagrody. Dodatkowo obserwuje się powolny spadek współczynnika KL , co świadczy o stabilizacji modelu na pewnej taktyce.



Wykres 29: Metryki w selfplay, model v4 [12]

Model po modyfikacjach znacznie rzadziej podejmuje decyzje o późniejszym zakończeniu gry. Nadal zdarzają się okazjonalne nieoptimalne zagrania a prawdopodobnie dłuższy trening mógłby rozwiązać ten problem.



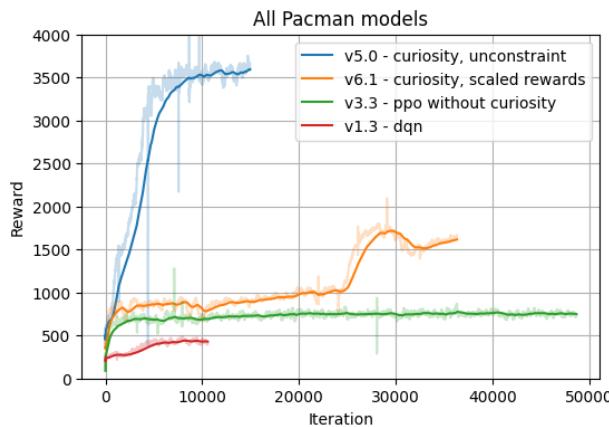
Animacja 8: Rozgrywka modelu v5.0 (Selfplay), link: v5.0_20231127-090152

12 Wyniki i dyskusja

Większość przeprowadzonych eksperymentów zakończyła się sukcesem. Algorytmy radzą sobie dobrze, nawet w rozwiązywaniu trudnych zadań, takich jak gra Pacman. PPO wykazuje znacznie większą stabilność i łatwość uczenia w porównaniu do DQN. Dodatkowo jego implementacja jest relatywnie łatwa a same eksperymenty nie wymagają intensywnego strojenia hiperparametrów. Kluczowe jest zrozumienie ogólnych wartości jakie powinny przyjąć hiperparametry oraz ich optymalizacja na podstawie działania modelu.

Sam algorytm nie osiąga znaczących rezultatów bez zastosowania ulepszeń, takich jak ciekawość (*ang. curiosity*). Ta technika pozwala na zageszczenie nagród i znalezienie równowagi między inteligentną eksploracją a maksymalizacją nagród ze środowiska, co o stanowi kluczowy element w rozwiązywaniu skomplikowanych środowisk. Dodatkowo, dzięki solidnej stabilności PPO, implementacja ciekawości nie wywołuje żadnych efektów ubocznych poza znanymi efektami stosowania tej techniki.

Wyniki algorytmów (wykres 30) kształtują się odpowiednio na poziomie 400 punktów (DQN), 800 punktów (PPO) i od 1600 do 3500 punktów (PPO + ciekawość). Wynik 700-800 punktów jest typowy dla prostych algorytmów RL w tym środowisku, jednak osiągnięcie średniej wyższej niż 1000 wymaga zaawansowanych technik i dobrze skalibrowanego środowiska. Osiągnięcie wyników w zakresie od 1600 do 3500 punktów jest godne uwagi. Wyniki algorytmów bez *curiosity* prawe zawsze osiągały jakieś maksimum, którego przebiecie było bardzo trudne, ale po zastosowaniu *curiosity* nawet długie przebiegi potrafiły powoli polepszać swój wynik.



Wykres 30: Zestawienie wszystkich modeli [12]

Algorytm DQN potrafi również osiągnąć bardzo wysokie wyniki w tych środowiskach, ale wymaga bardzo długiego trenowania i powolnego zmniejszania parametru epsilon ([1], strona 6), co jest niepraktyczne w przypadku komputerów nieprzeznaczonych specjalnie do uczenia maszynowego.

Przy dłuższym trenowaniu lub użyciu bardziej potężnego komputera eksperymenty te prawdopodobnie osiągnęłyby jeszcze lepsze rezultaty. Uczenie przez wzmacnianie jest dynamiczne i chaotyczne, a nawet po całkowitej stabilizacji wszystkich metryk, algorytm może

nagle znaleźć nowe rozwiązania lub strategie poprawiając wyniki po kilku milionach iteracji bez większych zmian.

Self-play pozostaje niestabilny i wymaga zastosowania algorytmu PPO. Dynamiczne zmiany w środowisku nakładają konieczność gromadzenia dużej ilości danych w celu stabilizacji gradientów. Proces uczenia modeli wymaga znacznej liczby iteracji, co pozwoli uzyskać satysfakcjonujące wyniki.

13 Ograniczenia i dalszy rozwój

Algorytmy opracowane w ramach tej pracy mają potencjał do dalszego rozwoju. Niskopoziomowa implementacja w czystym języku Python może spełnić potrzeby specyficznych implementacji lub modyfikacji algorytmów, które są trudne do osiągnięcia przy użyciu wysokopoziomowych API dostarczanych przez biblioteki takie jak *TensorFlow* czy *PyTorch*.

Niemniej jednak algorytmy te posiadają kilka ograniczeń. Przede wszystkim architektura nie obsługuje wyboru wielu akcji lub akcji w przestrzeni ciągłej. Jest to techniczne ograniczenie, które znacznie utrudniłoby implementację a jednocześnie nie miałoby zastosowania w żadnym z testowanych środowisk.

Warto byłoby przetestować implementację na różnych środowiskach, co mogłoby pomóc w identyfikacji innych potencjalnych problemów lub lepszej generalizacji algorytmu.

Niektóre eksperymenty nie udały się ze względu na niestabilność numeryczną, gdzie po kilkuset epizodach pojawiły się wartości *NaN*. To zjawisko może być spowodowane błędem w środowisku lub implementacji eksperymentu.

Algorytm *curiosity learning* mógłby być ulepszony poprzez dodanie sieci autoenkodera opartej na architekturze VAE (*ang. Variational Autoencoder*). Implementacja takiej sieci jest stosunkowo prosta i stanowiłaby skuteczną metodę poprawy wydajności algorytmu. Alternatywą mógłby być zastosowanie nowocześniejszych technik, takich jak destylacja neuronowa (*ang. neural network distillation*), co mogłoby pozwolić na uzyskanie jeszcze lepszych wyników.

14 Podsumowanie

Działający algorytm PPO wykazuje wysoką stabilność i osiąga lepsze wyniki niż algorytm PPO w podobnym czasie. Dodatkowo, *curiosity learning* pozwala na znacznie szybszą i inteligentną eksplorację, umożliwiając szybkie osiąganie bardzo wysokich wyników w wielu środowiskach.

Implementacja algorytmu PPO jest przygotowana do wykorzystania w innych środowiskach. W warunkach wymagających, zdołała udowodnić swoją efektywność, generując wyniki znacznie przewyższające możliwości bardziej podstawowych algorytmów. Istnieje potencjał do dalszego rozwijania przeprowadzanych eksperymentów a same algorytmy mogą być wzbgacane o nowe, zaawansowane techniki uczenia maszynowego, co z pewnością przyczyniłoby się do poprawy ich skuteczności.

Literatura

- [1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” 2013.
- [2] OpenAI, :, C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pauchocki, M. Petrov, H. P. d. O. Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, “Dota 2 with large scale deep reinforcement learning,” 2019.
- [3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” 2017.
- [4] Y. Burda, H. Edwards, D. Pathak, A. Storkey, T. Darrell, and A. A. Efros, “Large-scale study of curiosity-driven learning,” 2018.
- [5] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, “Deep reinforcement learning: A brief survey,” *IEEE Signal Processing Magazine*, vol. 34, p. 26–38, Nov. 2017.
- [6] R. M. Larsen and T. Shpeisman, “Tensorflow graph optimizations,” 2019.
- [7] M. Złotorowicz. <https://github.com/Lord225/ppo>, 2023. (online), dostęp 04-01-2024.
- [8] M. Złotorowicz, “draw.io: rysunki dołączone do pracy inżynierskiej,” 2023.
- [9] J. Fan, Z. Wang, Y. Xie, and Z. Yang, “A theoretical analysis of deep q-learning,” 2020.
- [10] H. van Hasselt, A. Guez, and D. Silver, “Deep reinforcement learning with double q-learning,” 2015.
- [11] T. Haarnoja, H. Tang, P. Abbeel, and S. Levine, “Reinforcement learning with deep energy-based policies,” 2017.
- [12] M. Złotorowicz, “Notatnik generujący wykresy do pracy inżynierskiej.” https://github.com/Lord225/ppo/blob/master/src/analyse/tb_edit.ipynb, 2023. (online), dostęp 04-01-2024.
- [13] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, “The arcade learning environment: An evaluation platform for general agents,” *Journal of Artificial Intelligence Research*, vol. 47, p. 253–279, June 2013.