Adrian Kwaśnik

2 rok Informatyka (Profil Praktyczny)

Grupa 2

Indeks: 292587

DOKUMENTACJA

UNIWERSYTET GDAŃSKI

Wydział Matematyki, Fizyki i Informatyki

Instytut Informatyki

Inteligencja Obliczeniowa

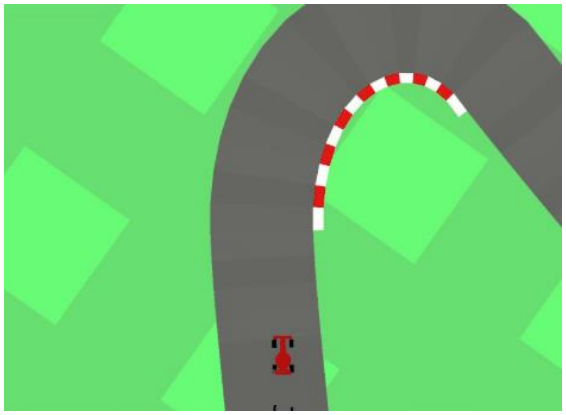
PROJEKT

Temat: Sterowanie Agentem w środowisku GYM car race.

1. Wprowadzenie.

Celem projektu było sprawdzenie jakie są możliwości wykorzystania inteligencji obliczeniowej, w celu stworzenia agenta sterującego autem w środowisku wirtualnym

OpenAI Gym – Car Racing.

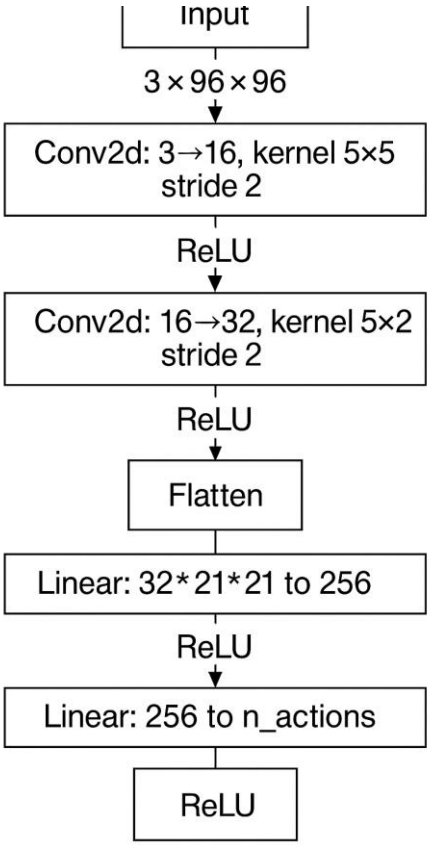


2. Wstępne rozeznanie.

Od początku chciałem skupić się na napisaniu odpowiedniego kodu, który wypisuję progres, czas szkolenia, tworzy krzywą uczenia, ewaluuję model i straty, zapisuje przykładowe powtórki oraz tworzy wykresy. W momencie kiedy stworzyłem taki kod z wykorzystaniem Deep Q Network, mogłem zacząć implementować inne modele tylko zmieniając kilka linijek kodu lub dokonywać lekkiej refaktoryzacji jeżeli chce wytestować inny sposób.

Pierwszy kod to po prostu implementacja Reinforcement Learningu przy pomocy Deep Q Network.

Model Summary



Ten model otrzymuje 96x96 pixeli RGB, czyli jedną klatkę ze środowiska Gym Car Racing, i do tej klatki wybiera akcję spośród zdefiniowanych akcji:

AKCJE

```
acts = [
    np.array([0.0, 0.0, 0.0], dtype=np.float32),
    np.array([0.0, 1.0, 0.0], dtype=np.float32),
    np.array([0.0, 0.0, 0.8], dtype=np.float32),
    np.array([-1.0, 0.0, 0.0], dtype=np.float32),
    np.array([1.0, 0.0, 0.0], dtype=np.float32),
    np.array([-1.0, 0.5, 0.0], dtype=np.float32),
    np.array([1.0, 0.5, 0.0], dtype=np.float32),
]
```

Gdzie dana akcja na klatkę to:

[skręt, gaz, hamulec]

Skręt w lewo : -1

Skręt w prawo: 1

Oczywiście w zaawansowanym modelu akcje to

- **Kierownica** ∈ [–1.0, +1.0]
- **Gaz** ∈ [0.0, +1.0]
- **Hamulec** ∈ [0.0, +1.0]

Przykładowa akcja:

[1.0,1.0,0] - pełny skręt w prawo podczas pełnego gazu.

1. Sieć

- **Sieć główna (Q-network)** przyjmuje na wejściu obraz stanu (np. klatkę z gry) i przetwarza go przez kolejne warstwy konwolucyjne oraz w pełni połączone.
- Na wyjściu sieć generuje wektor wartości Q dla każdej dostępnej akcji. Wyższa wartość Q oznacza lepszą prognozę zysku z wykonania danej akcji.

2. Eksploracja i eksploatacja

- W pierwszych etapach treningu agent wybiera akcje losowo (eksploracja), aby poznać środowisko.
- Z biegiem treningu stopniowo maleje parametr ε (epsilon), przechodząc od czystej eksploracji do eksploatacji, czyli wyboru akcji o najwyższej wartości Q.

- Taki mechanizm (epsilon-greedy) balansuje między poszukiwaniem nowych strategii a wykorzystaniem już nauczonych.

3. Replay buffer (bufor doświadczeń)

- Każde doświadczenie agenta (stan, akcja, nagroda, następny stan, flaga zakończenia) jest zapisywane w buforze o ograniczonej pojemności.
- W trakcie treningu losowane są minibatche doświadczeń, co:
    - Zmniejsza zależność między kolejnymi próbkami (próby są od siebie niezależne).
    - Umożliwia wielokrotne uczenie się na tych samych danych, co znacznie poprawia stabilność i efektywność nauki.

4. Sieć docelowa (target network)

- Aby uniknąć niestabilności przy jednoczesnym aktualizowaniu sieci i wyliczaniu celów Bellmana, stosuje się drugą, rzadziej aktualizowaną kopię sieci (network target).
- Cele dla aktualizacji wyznaczane są na podstawie tej sieci docelowej, co stabilizuje proces optymalizacji.

5. Proces aktualizacji wag

1. **Obliczenie bieżącej wartości Q**: sieć główna przewiduje wartość Q dla podanych stanów i wykonanych akcji.
2. **Obliczenie wartości docelowej** (target): nagroda + czynnik dyskontujący (γ) pomnożony przez maksymalną wartość Q w stanie następnym, wyliczoną za pomocą sieci docelowej.
3. **Funkcja straty**: obliczana jest różnica (średni błąd kwadratowy) między przewidywanymi Q a wartością docelową.
4. **Optymalizacja**: za pomocą algorytmu Adam (lub innego optymalizatora) wagi sieci są stopniowo korygowane, aby minimalizować tę stratę.

6. Harmonogram treningu

- Trening odbywa się w odcinkach (epizodach). Każdy epizod to seria kroków od zresetowania środowiska do momentu zakończenia.
- Po każdym odcinku ε jest zmniejszane (z limitem dolnym), a co pewną liczbę kroków sieć docelowa jest synchronizowana z siecią główną.

3. Sprawdzanie innych modeli.

Od momentu utworzenia kodu DQN, zacząłem badać i trenować inne modele (zwiększenie liczby cech w sieciach konwolucyjnych lub dodanie kolejnej warstwy sieci konwolucyjnej) lub innych metod optymalizacji (SAC, PPO, NEAT).

4. Sprawdzanie innych sposobów.

Żeby dokonać dokładnego badania tworzenia agentów kontrolujących musiałem spróbować również innych sposobów tj.

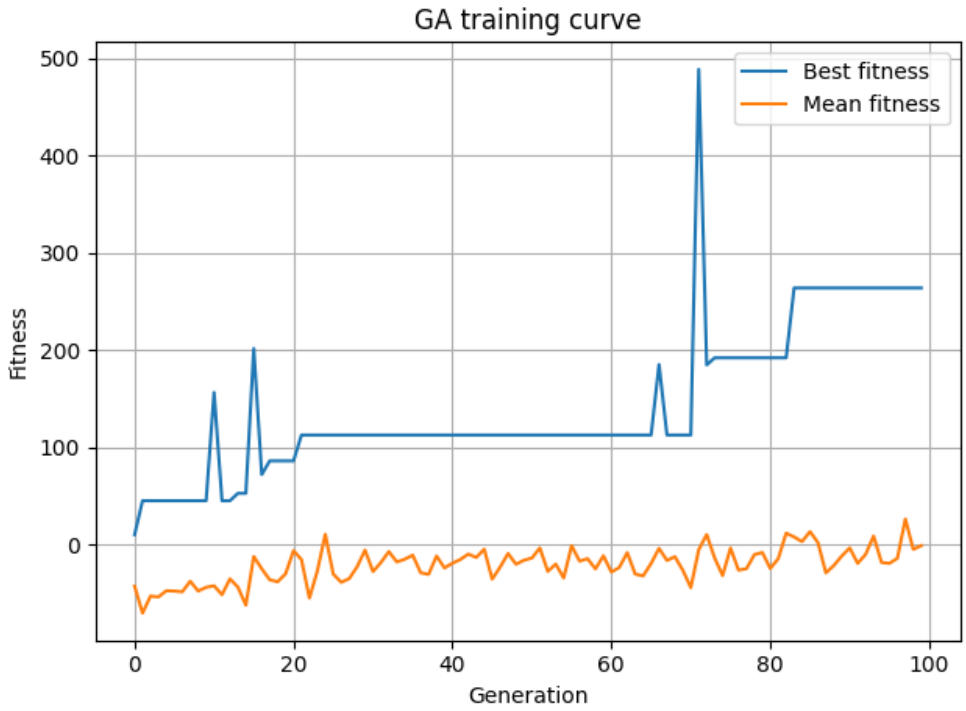
- Kontroler rozmyty
- Algorytm Genetyczny
- Inteligencja roju (PSO)

5. Wnioski.

Oczywiście najlepszym sposobem był reinforcement learning z wykorzystaniem sieci konwolucyjnych. Rozwiązania z rojem lub algorytmem genetycznym miałyby sens w momencie, kiedy auto zawsze jedzie po tym samym torze, wtedy algorytm genetyczny, którego gene_space to lista 1000 akcji, mógłby się uczyć i robić postępy, natomiast przez to, że auto co epizod jedzie po innym torze lub zaczyna w innym miejscu ten algorytm nie ma sensu.

Kontroler rozmyty był zbyt trudny do zaimplementowania może też za mało czasu poświęciłem na przebadanie tego sposobu, ale starałem się zdefiniować najlepsze zasady dla wykrywania zakrętów i nic z tego nie wyszło, następnie prosiłem chat o pomoc w tym zagadnieniu i niestety przez nieudane próby czułem, że jest to strata czasu.

NEAT czyli połączenie algorytmu genetycznego wraz z sieciami neuronowymi, ten sposób był całkiem ciekawy i robił postępy, poprawnie się uczył natomiast był bardzo czasochłonny w porównaniu z RL, nie rozumiałem dlaczego, ale już samo 100 generacji po 10 sieci trwało tyle samo co 1000 epizodów w sposobie z reinforcement learningiem. Dodatkowo najlepsze funkcje fintess zdobywały w ostatnich generacjach tylko po 300 punktów, gdzie np. DQN po takim czasie potrafiło uśrednić wyniki do 700 (reward score).



Najlepszym sposobem optymalizacji okazał się model PPO (proximal policy optimization), to dzięki niemu udało się wytrenować model, który jeździ znakomicie, a trenowanie było o wiele prostsze i szybsze przez zastosowanie “tricków”, które oferowała od razu biblioteka stable_baselines3, z której importowaliśmy model.

```
# Setup vectorized environments
envs = DummyVecEnv([make_env for _ in range(8)])
envs = VecMonitor(envs)
envs = VecFrameStack(envs, n_stack=4)
os.makedirs(args.model_dir, exist_ok=True)
```

Dzięki tym narzędziom mogliśmy trenować model w 8 środowiskach na raz co znacznie przyspieszyło proces, dodatkowo do sieci konwolucyjnej jako stan nie wchodziła jedna klatka tylko 4 (VecFrameStack), czyli stanem dla modelu PPO były 4 klatki gry, te rzeczy znacznie usprawniły nauczanie, ale sam gym już zaoferował nam usprawnienie w postaci możliwości

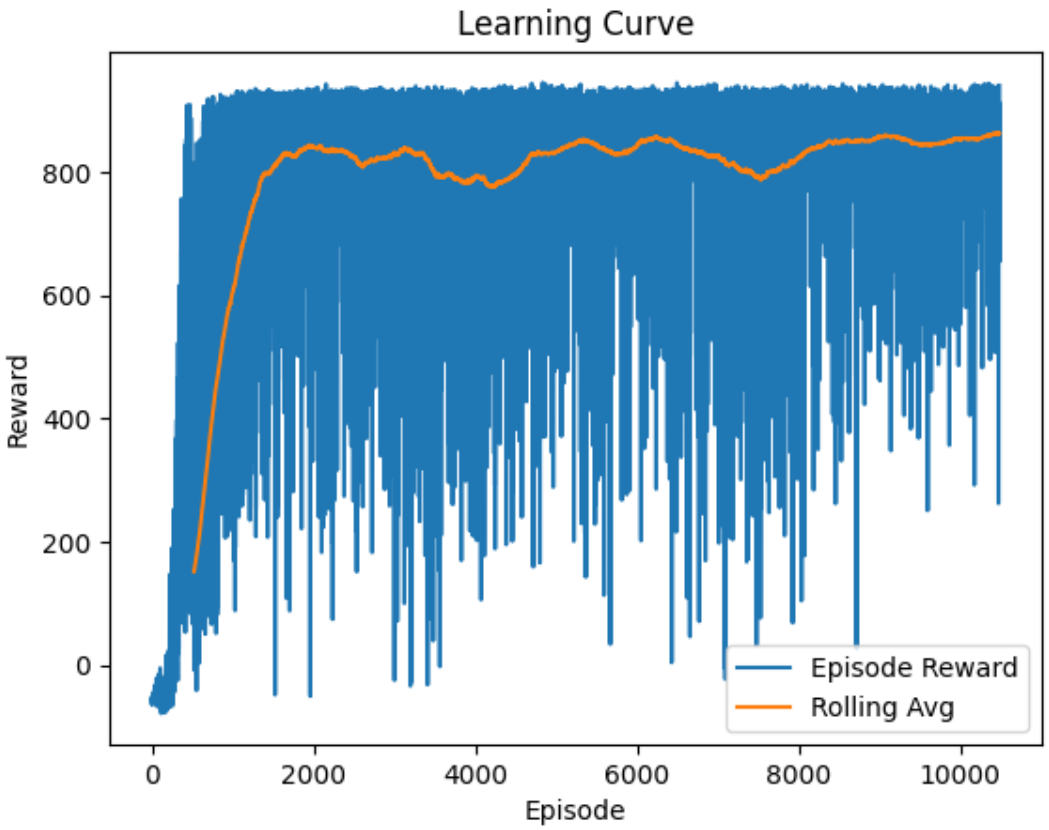
resizeowania środowiska z 96x96 pikseli na 84x84 i przejście z pikseli RGB na skalę szarości “GrayScaleObservation”.

Wnioski najlepszych modeli:

Model PPO stosował politykę, czyli można by powiedzieć coraz to nowsze lepsze zasady nauczania, metoda jest tu podobna jak w generacji obrazów GAN, dla danego stanu i akcji jest aktor, który podejmuje akcje i krytyk, który ocenia czy wykonanie akcji A w stanie S, było lepsze czy gorsze niż przeciętnie oczekiwana nagroda dla stanu S.

Model SAC, podobnie jak DQN korzysta z replay buffer do uczenia się, a nie matematyczną funkcję polityki tak jak PPO. Ogólnie to SAC też był jak najbardziej dobrym modelem, który dobrze się uczył aczkolwiek po prostu trochę wolniej.

Trenowanie modelu PPO



(trosze za często badałem Episode Reward)

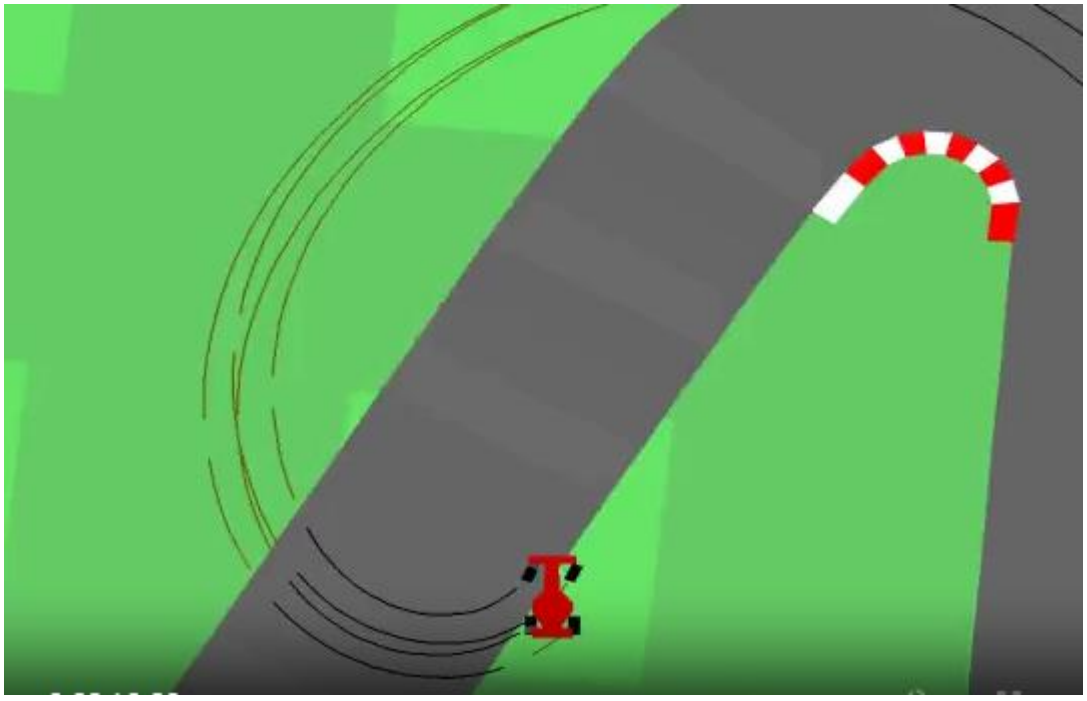
Jak widać model dobrze się uczy, a średni Episode Reward jest coraz wyższy (słupki na dole, nie opadają aż tak do dołu).

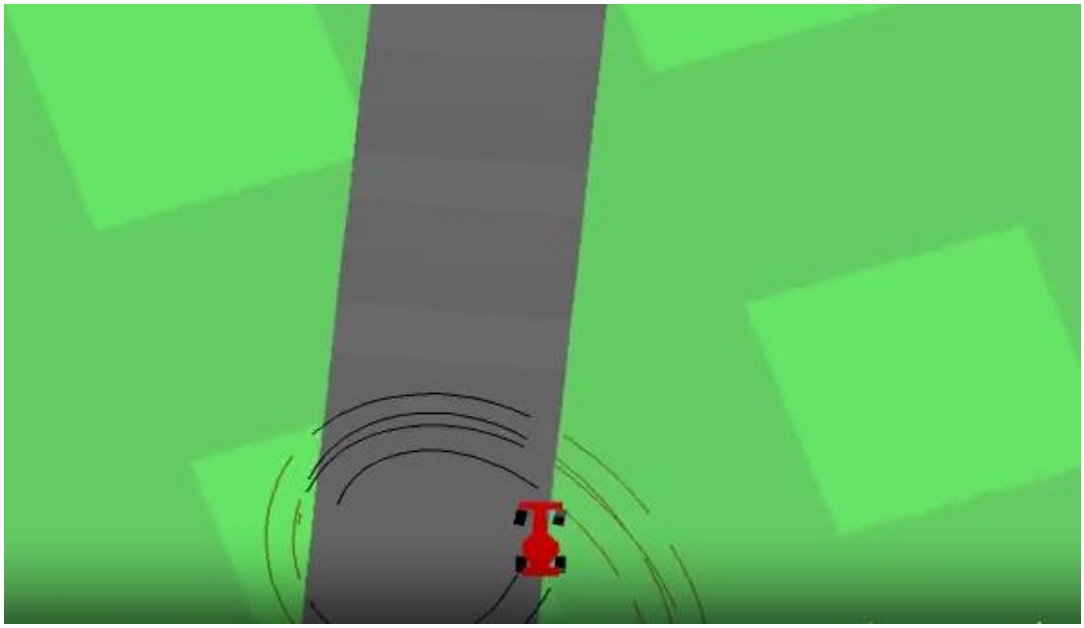
6. Przemyślenia.

Łatwo zauważyłem, że modele aktor/krytyk w sposobie z Reinforcement Learningiem były najbardziej skuteczne więc głównie skupiłem się na nich. Fajnym pomysłem byłoby dodanie mnożnika punktów w trakcie driftowania auta, model wtedy mogły nauczyć się bardziej optymalizować nie samą jazdę, ale też poślizg do zdobywania punktów.

Jednym z ważnych aspektów było dodanie pełnej ilości akcji, to znacznie usprawniło eksplorację w trakcie nauczania, w finalnym kodzie aplikacji Box([-1. 0. 0.], [1. 1. 1.], (3,), float32), czyli Box(low,high,shape, dtype). Aktor mógł wybierać dowolną akcję i optymalizować politykę dla dowolnej akcji A w stanie S, a mimo to i tak uczył się szybciej. Wygląda na to, że model im więcej ma możliwości tym szybciej osiągał wyższe nagrody.

Samo utrzymanie auta w prostej linii to nie było po prostu trzymanie gazu, dla modelu było to robienie jedno-klatkowych (szybkich) skrętów na torze lewo-prawo-lewo-prawo..., żeby utrzymać się w prostej linii. Bardzo fajnym zjawiskiem było jak model, żeby utrzymać prędkość, a zbierać punkty robił jedną pętlę poślizgiem po czym wracał na tor.





Tak to mniej więcej wyglądała sytuacja z pętlą.

Tracił kilka punktów na ostrym zakręcie by zyskać czas.

7. Podsumowanie.

Projekt, na pewno dałoby się rozwinąć, zwłaszcza sposób nagradzania modelu. Dzięki poświeceniu czasu na projekt zagłębiłem się w tematy związane z Reinforcement Learningiem, metodami optymalizacji sieci neuronowych, ale też i danych wejściowych. Zwłaszcza to środowisko GYM od OpenAI trenowanie na tym środowisku przystosowanym do tego jest troszkę prostsze niż już środowisko prawdziwej gry. Stworzenie Agenta do takiego przystosowanego środowiska to dobre wprowadzenie do tworzenia agentów w trudniejszych środowiskach oraz jak to mogłoby wyglądać.

Podsumowując jestem zadowolony, że udało mi się wytrenować dobry model i osiągać nie do pobicia przez człowieka wyniki.