

# MPUM project - flappy bird reinforcement learning

Bartosz Bromblik & Jacek Markiewicz

## 1 Wstęp

"*Flappy bird*" to prosta gra z długą historią irytowania wszystkich graczy. Mimo swojej prostoty, wcale nie jest prosta. A przynajmniej dla człowieka. A czy tak samo jest dla maszyny? Celem projektu jest sprawdzenie skuteczności domowej roboty implementacji kilku algorytmów uczenia przez wzmocnienie (ang. RL - Reinforcement Learning), tu:

- Q-learning,
- Sieć neuronowa,
- Algorytm genetyczny.

Do raportu załączone jest (chaotyczne) repozytorium z zaklepanymi algorytmami, ich wynikami oraz samą grą.

## 2 Opis gry

Jesteśmy ptakiem (żółtą kulką) i chcemy dolecieć jak najdalej. Ale na naszej drodze jest pełno rur (to zielone). Na szczęście są w nich dziury, przez które możemy spróbować przelecieć. I to tyle co można powiedzieć o celu gry.

Całość kontroli to klikanie *spacji*, które powoduje skok ptaka. Poza tym *Esc* wyłącza grę, *R* (niekoniecznie duże) restartuje rozgrywkę. Każdy inny przycisk działa jako pauza (lub tę pauzę cofa).

Gra jest mniej więcej w pełni konfigurowalna. Parametry można znaleźć w pliku *game\_config.py*. Ptak zawsze zaczyna na wysokości połowy mapy, z prędkością poziomą równą 0. Grawitacja jest stała, prędkość pozioma też. Skok ustawia prędkość pionową na tę samą wartość. Grubość (pozioma) rury jest stała. Odległości między rurami są brane z rozkładu normalnego. Środek dziury jest brany z rozkładu jednostajnego. Promień dziury jest brany z rozkładu normalnego i z każdą kolejną rurą maleje (tempo zależy od poziomu trudności).

Hiperparametr *HARD* określa poziom trudności gry (*False* to łatwy, *True* to trudny).

Silnik gry nie ma wbudowanego zegara. Dzięki temu można trenować modele szybciej niż w czasie rzeczywistym. A w trybach do grania jest oddzielny zegar, który pilnuje odpowiedniego framerate'u.

Moduły do wyświetlania gry jest napisany w pygame'ie.

### 3 Rozgrywka

Pliki *play\_\*.py* to ta grywalna część projektu. Każdy odpowiada któremuś z algorytmów (sposób implementacji może się różnić między plikami) poza plikiem *play\_yourself.py*, gdzie nie ma żadnego wspomagania. Tak, sterowanie działa wszędzie i algorytmom można przeszkadzać.

### 4 Ogólnie o uczeniu

Podczas gry, na górze okna wyświetlają się na czerwono 3 tuple liczb. Oznaczają one kolejno:

- obecną (uogólnioną) stratę, czyli odległość pozioma minus różnica poziomów środka ptaka i środka następnej dziury, używana do oceny modelu. Często zwana również **wynikiem**.
- stan, czyli (pozycję pionową, prędkość pionową, pozycję dolnego końca następnej dziury, pozycję górnego końca następnej dziury, odległość poziomą do następnej rury). Przy czym to ostatnie jest liczone od prawego końca ptaka do lewego końca rury. Dopiero gdy ptak w całości minie linię końca rury, zmieniana jest "następna rura".
- (ilość miniętych rur, status), gdzie to drugie to 1 gdy dalej żyjemy i 0 gdy już nie.

Stan świata jaki widzi model to dokładnie ta środkowa tupla.

### 5 Uczenie nie maszynowe

Jako kontekst powiem, że pomimo wielu prób, nie udało mi się zdobyć wyniku powyżej 7 (samej siódemki też zresztą nie).

### 6 Q-learning

Pierwszą z zastosowanych metod była metoda tzw. Q-learningu. Algorytm ten jest bardzo ogólny i robi niewiele założeń. Traktujemy nasz rozważany świat jako zbiór *stanów*, między którymi jesteśmy w stanie przechodzić przy użyciu *akcji* (skok / nie-skok).

Teraz będziemy starać nauczyć funkcji  $Q : \text{stan} \times \text{akcja} \rightarrow \mathbb{R}$  ("Q" od ang. *quality*) - która dla każdej potencjalnej akcji w danym stanie da nam "jakość" danego ruchu. Decyzję ostateczną podejmiemy zależnie od tego, która z akcji będzie miała dla danego stanu odpowiednio większą wartość Q.

Będziemy to robić następująco - Q początkowo jest dowolnie zinicjalizowane. Następnie przeprowadzamy rozgrywkę. W każdym momencie czasowym  $t$  naszej rozgrywki możemy ją opisać przez stan  $S_t$ . Podejmujemy w każdym momencie akcję  $A_t$  zależnie od Q. W ten sposób przechodzimy do stanu  $S_{t+1}$  oraz otrzymujemy pewną *nagrodę*  $R_{t+1}$ . Następnie aktualizujemy funkcję Q w oparciu o poniższy wzór:

$$Q(S_t, A_t) = (1 - \alpha) \cdot Q(S_t, A_t) + \alpha \cdot \left( R_{t+1} + \gamma \max_a Q(S_{t+1}, a) \right)$$

gdzie  $\alpha$  - szybkość uczenia się (*learning rate*) oraz  $\gamma$  - współczynnik dyskontowy (*discount factor*). Czyli aktualizujemy  $Q$  w oparciu o oczekiwaną nagrodę w przyszłości. W trakcie implementacji tego rozwiązania napotkaliśmy jednak niejedną komplikację.

Otóż algorytm ten ma jedną, dość istotną wadę. Implementowany najczęściej jest on, ze względu na brak lepszego sposobu, poprzez fizyczne trzymanie tablicy  $Q$  indeksowanej stanami i akcjami. To narzuca jednak to, że zbiory stanów i akcji muszą być skończone. O ile akcje nie są problemem u nas, o tyle stany już sprawiają pewnego rodzaju zagwozdkę. Stan gry, w postaci krotki (*wysokość ptaka, prędkość pionowa ptaka, odl. pozioma do następnej rury, odl. pionowa do następnej rury*) trzeba zkwantyzować. Ostatecznie, w wyniku prób i błędów, każda wartość z owej krotki dostała odpowiednio 16, 11, 25, 14 unikalnych, dyskretnych wartości.

Ważne w algorytmie jest również *eksploracja*. Otóż, jeśli nie będziemy nigdy widzieć / eksplorować pewnych stanów gry, to po ich napotkaniu nasz algorytm nie będzie działał dobrze - za mało razy byliśmy w danym stanie i wartość  $Q$  dla niego nie jest jeszcze pewna. Dlatego też na początku treningu staraliśmy się wykonywać naszym agentem losowe akcje. W dalszej fazie treningu proporcja losowych akcji systematycznie spada do bardzo małej - ale niezerowej - wartości.

Istotne było również dobranie dobrej funkcji kosztu. Początkowo nagradzała jedynie za przeżywanie, oraz bardzo karmiła za przegrywanie. Jednakże nauka szła opornie. Dlatego dodałem małą modyfikację do funkcji kosztu - nagradzałem dodatkowo za przebywanie na wysokości w pobliżu dziury w następnej rurze. Efekt był zadowalający - algorytm szybciej i sprawniej się uczył.

Model oparty o  $Q$ -learning wytrenowany na przestrzeni 20000 rozgrywek jest w stanie pokonać średnio **4.66** rur.

## 6.1 SARSA

Zaprezentowany sposób aktualizacji tablicy  $Q$  nie jest jedynym możliwym. Algorytm *SARSA* (*state-action-reward-state-action*) bazuje na poniższej metodzie aktualizacji:

$$Q(S_t, A_t) = (1 - \alpha) \cdot Q(S_t, A_t) + \alpha \cdot (R_{t+1} + \gamma Q(S_t, A_t))$$

Reszta algorytmu pozostaje niezmienną względem  $Q$ -learningu. Dzięki zastosowaniu ten prostej zmiany, (oraz podobnie jak wcześniej wytrenowniu na przestrzeni 20000 rozgrywek) jesteśmy osiągnąć rezultaty na poziomie **5.39** przeskoczonych rur.

Niestety, ciężko nam uzyskać idealne rezultaty - kwantyzacja stanów idealna nie jest co przekłada się na niedokładności w reprezentacji rzeczywistego stanu oraz co za tym idzie okazjonalną niespodziewaną śmiercią.

## 7 Sieć neuronowa

## 8 Algorytm ewolucyjny

Ogólny zamysł metody jest taki, żeby na początku wygenerować losowo działające modele, a następnie wybierać najlepsze, usuwać najgorsze, mutować to co zostało i ponawiać pętle. W obecnej implementacji wszystko opiera się o arbitralnie wybraną sieć neuronową o wymiarach  $[5, 5, 2]$ , czyli z tylko jedną ukrytą warstwą. Funkcja aktywacji to ReLU, a wynik sieci jest mielony przy użyciu softmax'a. Opisany kod znajduje się w pliku *evolutionary.py*.

### 8.1 Proces uczenia

Na początku parametry wszystkich modeli są inicjalizowane rozkładem normalnym  $\mathcal{N}(0, 10^{-1})$ . Następnie w pętli *#epok*-krotnie:

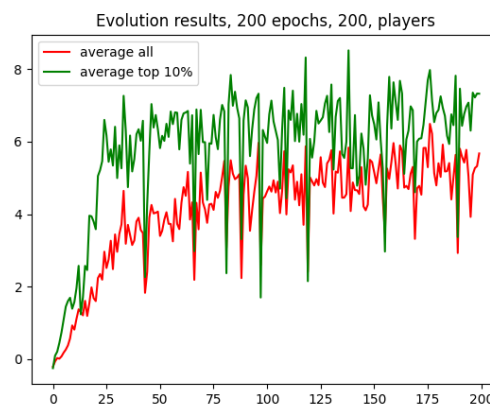
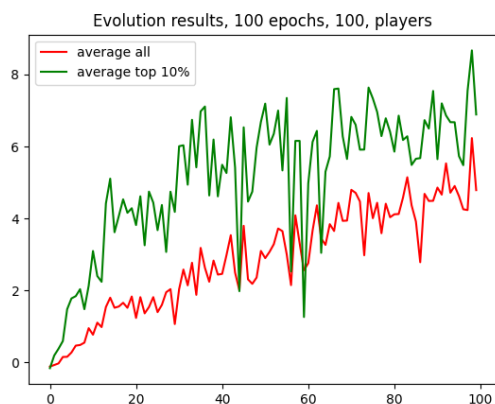
- generowany jest układ rur (mapa), na którym oceniane będą modele,
- każdy model jest oceniany,
- 10% najlepszych modeli pozostaje w populacji nieulegając zmianie,
- 30% najlepszych modeli (w czym powyższe) trafia do populacji w trzech kopiach, odrobinę zmienionych. Tzn. do ich parametrów dodawane są wartości brane z  $\mathcal{N}(0, 10^{-4})$ ,
- zapisywane są wyniki (nie wpływa na działanie algorytmu).

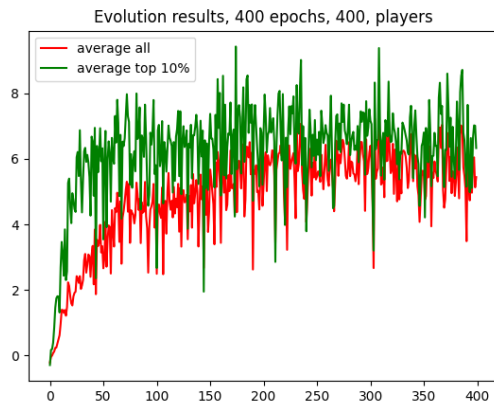
Należy zauważyć, że rozmiar populacji nie ulega zmianie.

Na koniec każdy pozostały w populacji model jest oceniany ja 10 konfiguracjach mapy (każdy model na każdej z tych map) i wybierany jest ten działający średnio najlepiej.

### 8.2 Rezultaty

Pomimo relatywnie prostej implementacji, bez krzyżowania modeli, przekazywania genów itd., wyniki są całkiem niezłe. Zapisane są trzy wyniki, dla (*#epok*, *rozmiar populacji*) kolejno: (100, 100), (200, 200), (400, 400), o czasach trenowania rzędu: minuta, pare minut, parnaście minut.





Zdecydowanie warto zauważyć, że wyniki mocno zależą od mapy. Uśrednione wyniki oscylują w okolicy 6 z hakiem, choć trafiają się również wysmienite układy rur pozwalające na wyniki około 9.

Na trzecim rysunku widać, że wyniki całości populacji niewiele odbiegają od wyników najlepszych 10%, co może być spowodowane fizycznymi ograniczeniami mapy.

### 8.3 Dokładne rezultaty

Modele będące wynikami przebiegów, które wygenerowały powyższe rysunki, dostępne są w pliku *models\_evolutionary.py*. Plik ten można odpalić by dokładniej ocenić osiągnięte sieci. Po takim właśnie przetestowaniu, na  $10^4$  mapach, wyniki prezentują się następująco:

- Próba mała, (100, 100): 7.0758.
- Próba średnia, (200, 200): 7.4886.
- Próba duża, (400, 400): 7.0753.

Wszystkie wyniki są całkiem dobre, zdecydowanie lepsze niż nawet rekordy co poniektórych autorów. Należy pamiętać, że każde z powyższych to średni wynik z wielu przebiegów **JEDNEGO** modelu, co niewiele jest w stanie powiedzieć o rzeczywistej przewadze tych czy innych hiperparametrów.

Ten środkowy model jest dołączony również w pliku *play\_evo.py*, gdzie można pooglądać, jak sobie gra.

Jego taktyka nie wydaje się zbyt skomplikowana. Model utrzymuje ptaka na wysokości środka kolejnej dziury i zdaje się nie myśleć zbyt wiele więcej, choć nie są to bardzo uważne obserwacje.

## 9 Podsumowanie