UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ
W LUBLINIE
Wydział Matematyki, Fizyki i Informatyki

Kierunek: **Informatyka**
Specjalność: -

Wiktor Wajszczuk
nr albumu: 296534

Zastosowanie uczenia przez wzmacnianie w środowisku agentowym

Application of reinforcement learning in an agent-based environment

Praca licencjacka
napisana w Katedrze Oprogramowania Systemów Informatycznych
pod kierunkiem dr Marcina Denkowskiego

Lublin, rok 2022

Spis treści

Wstęp

Uczenie maszynowe (ang. *machine learning*) jest dziedziną zajmującą się tworzeniem metod, które potrafią się "uczyć" to znaczy, wykorzystywać dane aby poprawiać wydajność w rozwiązywaniu pewnych problemów[1]. Do głównych technik uczenia maszynowego zaliczają się:

- Uczenie nadzorowane gdzie komputer dostaje przykładowe dane wejściowe i pożądane dane wyjściowe, jego celem jest nauczyć się generalnej zasady przypisywania danych wejściowych do wyjściowych
- Uczenie nienadzorowane gdzie dane wejściowe nie są w żaden sposób oznaczone a algorytm uczenia musi sam znaleźć ukryte wzory w danych.
- **Uczenie przez wzmacnianie** (ang. *reinforcement learning*) W odróżnieniu od zarówno uczenia nadzorowanego i nienadzorowanego nie potrzebujemy w tym przypadku żadnych gotowych danych wejściowych i wyjściowych. Zamiast tego, algorytm pozyskuje dane na bieżąco ze środowiska, do którego jest zastosowany. Następnie wykorzystuje zaobserwowane dane, by osiągnąć określony cel.

Właśnie dzięki temu, że algorytmy uczenia przez wzmacnianie nie potrzebują do nauki gotowych zestawów danych możemy zastosować je do problemów takich jak gra na giełdzie [2], nauka grania w gry, tworzenie autonomicznych pojazdów, uczenie chodzenia robotów z nogami czy optymalne chłodzenie serwerowni[3].

Celem niniejszej pracy jest zaproponowanie, implementacja oraz analiza algorytmu uczenia przez wzmacnianie zdolnego do kierowania agentem w popularnej grze. W toku analizy został wybrany algorytm **Q-Learning**, który został zastosowany do wytrenowania agenta realizującego prostą grę "Flappy Bird". Implementacja została zrealizowana z użyciem języka python oraz modułu pygame.

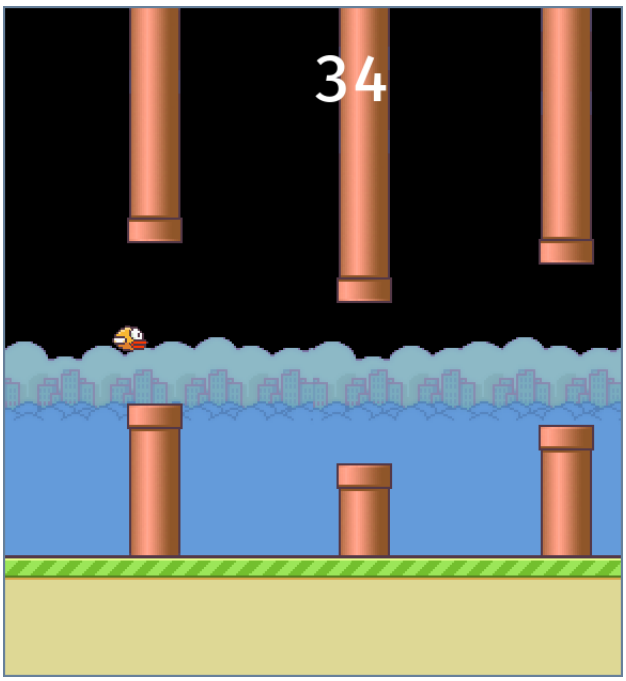
W rozdziale "Gra" przedstawię jaką grę wybrałem, opiszę jej zasady oraz zaprezentuję szczegóły jej implementacji. Następnie w rozdziale "Uczenie maszynowe" przybliżę zagadnienie uczenia przez wzmacnianie. Algorytm Q-learning oraz jego implementację opiszę w rozdziale "Algorytm Q-Learning". Na koniec w rozdziale "Testy i rezultaty" przedstawię testy po optymalizacjach algorytmu, które zastosowałem.

Rozdział 1

Gra

W celu realizacji założonych celów pracy zdecydowałem się na zaimplementowanie popularnej gry z 2013 roku “Flappy Bird”, autorstwa wietnamskiego developera Dong Nguyena[4]. Wybrałem tę grę ze względu na ograniczoną ilość akcji jaką może podjąć gracz.

1.1 Zasady gry



Rysunek 1.1: Zrzut ekranu z gry.

Gracz kontroluje obiekt reprezentowany w grze przez żółtego ptaka. Zadaniem gracza jest tak nim sterować by omijać obiekty reprezentowane przez czerwone rury oraz nie pozwolić na to by siła grawitacji doprowadziła do kolizji obiektu ptaka z obiektem ziemi

(na rysunku 1.1 zielona część na dole). Na obiekt ptaka działa jedynie siła grawitacji a obiekty rur przesuwane są w lewo w jego stronę ze stałą prędkością. Gracz może kontrolować jedynie to czy obiekt, którym steruje zwiększy swoją wysokość o stałą, zakodowaną ilość jednostek czy tego nie zrobi i obiekt reprezentowany przez ptaka zmniejszy swoją wysokość. Gra kończy się gdy dojdzie do kolizji obiektu ptaka z, którymkolwiek obiektem rur: dolną bądź górną albo z obiektem ziemi. Punkty przyznawane są jeżeli graczowi uda się ominąć nadchodzące przeszkody przelatując górną szczelinę między nimi. Za każde ominięcie przyznawany jest jeden punkt. Na górze rysunku 1.1 białymi cyframi wypisana jest ilość osiągniętych przez gracza punktów. W tym przypadku gracz ominął już trzydzieści cztery pary rur. Podsumowując – gra sprowadza się do kontrolowania wysokości ptaka tak by ten przelatywał między nadchodzącymi rurami.

1.2 Wykorzystane technologie

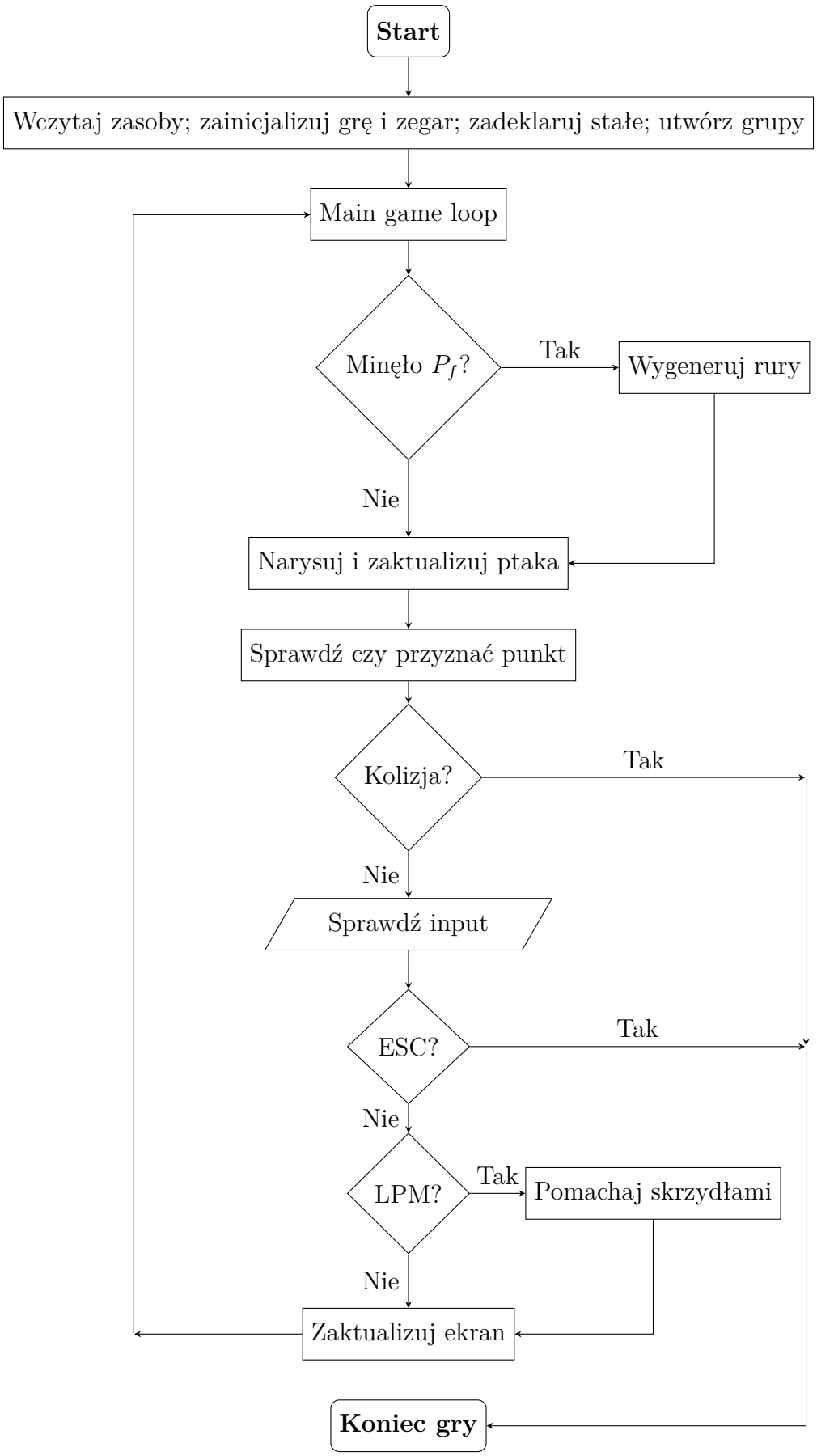
Do zaimplementowania tej gry zdecydowałem się na użycie wysoko poziomowego interpretowanego języka **python** w wersji 3.8. Python jest dostępny na większości współczesnych platform oraz jest to język open-source. Dodatkowo wykorzystałem moduł **pygame**.

Pygame używa funkcjonalności biblioteki SDL. Simple DirectMedia Layer(SDL) jest to wieloplatformowa biblioteka zapewniająca nisko poziomowy dostęp do audio, urządzeń peryferyjnych takich jak mysz, klawiatura, joystick oraz sprzętu graficznego (za pomocą OpenGL i Direct3D). Jest to technologia open source, która została zaimplementowana w C. Używana w grach firm takich jak *Valve Software* czy *SuperTuxCart*[5]. Pygame używa tej biblioteki do odczytywania wejścia z klawiatury/myszy oraz rysowania wyjścia na ekran.

Pygame pozwala tworzyć w pełni funkcjonalne gry oraz programy multimedialne w pythonie. Zaletami pygame są między innymi : podstawowe funkcje używają zoptymalizowanego kodu w C oraz Assemblerze, dostępny podobnie jak python na większości współczesnych platform. Ponadto jest modularny co pozwala programiście używać tylko tych komponentów, których naprawdę potrzebuje. Gier stworzonych z użyciem pygame na samej stronie projektu jest ponad 660 [6].

1.3 Implementacja

Głównym elementem implementacji jest nieskończona pętla, w której sprawdzam zdarzenia, które zaszły, sprawdzam czy mamy wejście od gracza, i aktualizuję obiekty gry oraz ekran. Na rysunku 1.2 opisana jako *Main game loop*.



Rysunek 1.2: Uproszczony diagram przepływu gry.

Program zaczyna od wczytania wczytania zasobów gry – obrazków reprezentujących tło, rurę, ziemię, ptaka. Każdemu z tych zasobów tworzy obiekt klasy *pygame.image*, wczytywane są za pomocą metody *load*. Do śledzenia czasu w grze wykorzystywany jest obiekt klasy *pygame.time.Clock*.

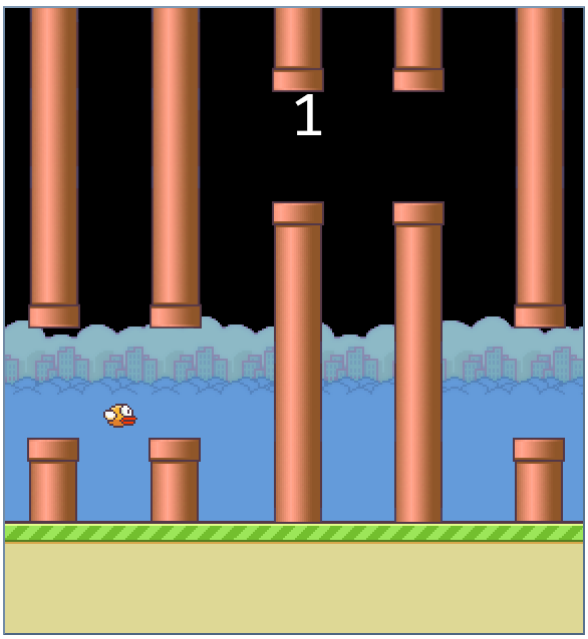
Stałe deklarowane na początku programu to na przykład: ilość jednostek o ile przesunąć w lewo obiekty rur przy każdym tyknięciu zegara, szerokość szczeliny między obiektami rur, czy czas po jakim generować nowe rury. W tym miejscu przypisujemy wartość stałej opisującej co jaki czas mamy generować nowe obiekty rur. Jest wyliczana ze wzoru:

$$P_f = 900 * (30 \div F) \tag{1.1}$$

Gdzie P_f to czas po jakim generowane są nowe obiekty rur (ang. *pipe frequency*), F to ilość klatek na sekundę (ang. *frames per second*). Dzięki takiemu podejściu gra może częściej generować obiekty rur przy większej ilości klatek na sekundę, przez co odległość między nimi zostaje taka sama bez względu na tempo gry.

Przy uruchamianiu programu użytkownik ma opcję wybrać wartość parametru klatek na sekundę : 30, 60 bądź 120. Domyślnie jest to 30, stąd we wzorze znajduje się ta liczba. W ten sposób przy 30 klatkach na sekundę nowe obiekty rur będą generowane co 900 milisekund, przy 60 co 450 milisekund i co 225 milisekund w przypadku 120 klatek na sekundę. Po kilku próbach stwierdziłem, że częstsze generowanie obiektów rur sprawiało, że gra była zbyt trudna dla gracza lub w najgorszym wypadku sprawiało, że gra była nie do przejścia jak na przykład w scenariuszu przedstawionym na rysunku 1.3 gdzie obiekt ptaka nie będzie miał wystarczająco czasu by wznieść się na odpowiednią wysokość a potem opaść by ominąć rury. Częstsze generowanie obiektów rur nie ma wpływu przy większej ilości klatek na sekundę, gdyż te przemieszczają się w lewo co tyknięcie zegara o stałą ilość jednostek, a tykanie zegara uzależnione jest od ilości klatek na sekundę – większa ilość klatek na sekundę oznacza częstsze tykanie zegara przez co rury przemieszczają się z większą prędkością. Rozgrywka staje się zdecydowanie dynamiczniejsza i przez to również trudniejsza, ale tylko dla ludzi. Q-learningowy agent dostaje informacje o grze co tyknięcie zegara i jeszcze w tym samym tyknięciu podejmuje decyzje o ruchu, ale więcej o tym w rozdziale Algorytm Q-Learning

Po określeniu wartości stałych tworzymy trzy instancje klasy *pygame.sprite.Group*, jedną dla obiektu ptaka, drugą dla obiektów rur dolnych oraz jedną dla obiektów rur górnych. Będą nam potrzebne później przy sprawdzaniu czy zaszła kolizja między obiektem ptaka a obiektami rur. Sam obiekt ptaka jest instancją klasy *Bird* dziedziczącej po *pygame.sprite.Sprite*. Wspomniana klasa jest przeznaczona dla obiektów, które mają być widoczne na ekranie, zawiera między innymi pole na obrazek reprezentujący obiekt oraz metodę *update*[7], którą będę wywoływał co każdy obrót głównej pętli programowej. W przypadku obiektu ptaka w przeładowanej metodzie *update* znajduje się logika “latania”



Rysunek 1.3: Często generowane obiekty rur.

– co wywołanie metody update inkrementuję zmienną która mówi o ile ma być przesunięty obiekt ptaka w dół do maksymalnej wartości ośmiu jednostek. Jeżeli został wciśnięty lewy przycisk myszy to wartość ta jest zmniejszana o dziesięć (minusowe wartości tej zmiennej powodują, że obiekt zostanie przesunięty w górę).

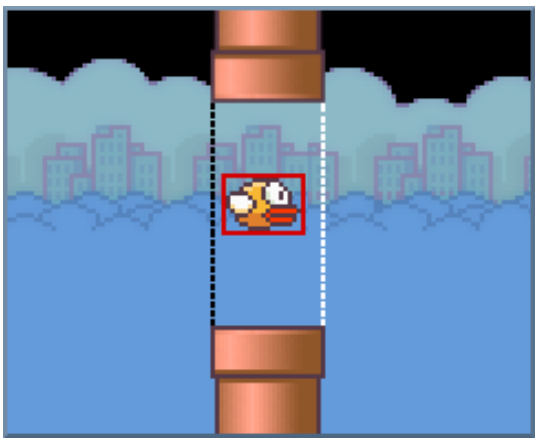
Po tym zaczyna się główna pętla programowa. Co P_f czasu generujemy nowe rury – polega to na stworzeniu dwóch instancji klasy *Pipe* dziedziczącej po *pygame.sprite.Sprite*. Generuję liczbę losową r z zakresu $(-100, 100)$, która będzie potrzebna w ustaleniu wysokości, na której wygeneruję obiekty rur. W konstruktorze nadaję obiektom odpowiedni obrazek. Po wczytaniu obrazku używam *get_rect* będącej metodą klasy *pygame.Surface*, której instancją jest nadany w konstruktorze obrazek. Zwrócony obiekt przypisuję do pola *rect*. Metoda ta mapuje obrazek na najmniejszy prostokąt, w którym zmieści się wczytany obrazek. Mamy dzięki temu dostęp do jego krawędzi, którym możemy nadawać współrzędne na ekranie, w których ma zostać narysowany. Wszystkie obiekty rysowane są w przestrzeni kartezjańskiej, gdzie punkt $(0, 0)$ jest lewym górnym rogiem tła. Górnej lewej krawędzi obiektu dolnej rury nadajemy współrzędne:

$$(W, H \div 2 + r + G \div 2)$$

Natomiast dolnej lewej krawędzi obiektu górnej rury nadajemy współrzędne:

$$(W, H \div 2 + r - G \div 2)$$

Gdzie W to szerokość tła ustalona na początku, H to wysokość tła, a G to stała oznaczająca odległość między dolną a górną rurą, przypisywana na początku programu. Tak



Rysunek 1.4: Granice *rect* obiektów *sprite*.

wygenerowane obiekty rur zostaną narysowane zaraz za ekranem z prawej strony, przesunięte od środka ekranu w pionie o r oraz połowę odległości ustalonej na początku działania programu. Tak wygenerowane obiekty dodaję do odpowiednich instancji *pygame.sprite.Group*. Jako, że klasa *Pipe* dziedziczy po *pygame.sprite.Sprite* (tak samo jak klasa *Bird*), to powinna przeładowywać metodę *update*[7]. W przypadku klasy reprezentującej rury jest tam tylko logika przesuwania obiektu w lewo co sprowadza się do zmniejszenia współrzędnej x zmiennej *rect* oraz ewentualnym usunięciu go jeżeli współrzędna ta po zmniejszeniu staje się ujemna (oznacza to, że jest po za lewą krawędzią ekranu przez co jest nie widoczna i nie potrzebna). Usuwanie polega na wywołaniu metody *kill*, co powoduje, że obiekt zostaje usunięty z każdej grupy, której był elementem.

W następnym kroku rysuję na ekranie obiekt ptaka za pomocą metody *draw* wywoływanej na grupie, do której ten obiekt należy. Metoda ta korzysta z nadanego w konstruktorze obrazka, oraz zmapowanego obiektu prostokąta (*sprite.Rect*) do określenia pozycji, na której ma zostać narysowany[8]. Rysowanie obiektów rur odbywa się w ten sam sposób, z tym, że wspomniana metoda jest wywoływana na odpowiednio grupie obiektów rur dolnych i górnych. Następnie na grupie, do której należy obiekt ptaka wywołuję metodę *update*, która obsługuje logikę "latania".

Najbliższym z prawej strony obiektem rury jest zawsze obiekt o indeksie zero z grupy dolnych bądź górnych rur, jest tak dzięki logice metody *update* klasy *Pipe*. Aby sprawdzić, czy przyznać graczowi punkt sprawdzamy najpierw, czy obiekt ptaka znalazł się w strefie między dwoma rurami, w tym celu wystarczy sprawdzić czy współrzędna x pola *rect.left* obiektu ptaka(na rysunku 1.4 zwizualizowana jako lewa krawędź czerwonego prostokąta wokół obiektu ptaka) jest większa tej samej współrzędnej obiektu dolnej rury. Na rysunku 1.4 jest to zaznaczone czarną przerywaną linią z lewej strony. Jeżeli tak jest to ustawiam logiczną flagę reprezentującą to zajście. W następnych obrotach głównej pętli programowej sprawdzane jest, czy ta flaga jest ustawiona i jeżeli tak jest to sprawdzam,

czy ptak opuścił strefę między dwoma rurami. Dzieje się to analogicznie do sprawdzania, czy wszedł z tą różnicą, że porównuję x pola *rect.right* obiektu dolnej rury (to znaczy współrzędną prawej krawędzi dolnej rury, na rysunku 1.4 oznaczona białą przerywaną linią z prawej strony) z współrzędną x pola *rect.left* obiektu ptaka. Jeżeli tak się stało to punkt zostaje przyznany a wcześniej wspomnianą flaga zresetowana. Powody takiego rozwiązania są dwa:

- Jeżeli sprawdzałbym tylko czy ptak ominął tylko lewą, albo tylko prawą cześć pola *rect* obiektu dolnej rury to punkty byłyby naliczane przy każdym obrocie głównej pętli programowej gdy obiekt rury najbliższej z prawej został ominięty a nie dotarł jeszcze za lewą granicę ekranu i nie został jeszcze usunięty.
- Jeżeli przyznawałbym punkt, gdy tylko obiekt ptaka znajdzie się w strefie między dwoma obiektami rurami to byłaby szansa, że gracz wpadnie na obiekt rury zaraz po przyznaniu punktu – byłoby to nieprawidłowe zachowanie

Aby sprawdzić, czy zaszła kolizja obiektu ptaka z ziemią wystarczy sprawdzić, czy współrzędna y dolnej części *rect* (czyli *rect.bottom*) będąca zmienną obiektu ptaka jest większa od H (przypomnę, że w *pygame* układ współrzędnych kartezjańskich ma swój początek w lewym górnym rogu ekranu a współrzędne z osi OY "rosną w dół"), gdzie H to wysokość tła przypisana na początku. Sprawdzenie kolizji między obiektem ptaka a obiektami rur dzięki *pygame* sprowadza się do wywołania metody **pygame.sprite.groupcollide**, która sprawdza, czy doszło do kolizji pomiędzy dwiema grupami podanymi jako argumenty i zwraca słownik, gdzie kluczami są obiekty z grupy podanej jako pierwszy argument a wartościami każdy obiekt z grupy podanej jako drugi argument, z którym obiekt będący kluczem wszedł w kolizję. Pod spodem metoda ta sprawdza tak naprawdę czy zmienne *rect* obiektu z każdej grupy nie nachodzą na siebie[9]. Jako, że grupa obiektów *Bird* ma tylko jeden element, to wystarczy, że wspomniana metoda zwróci cokolwiek by stwierdzić, że zaszła kolizja między obiektem ptaka a obiektami rur i zakończyć grę.

Pygame.event to moduł, który pozwala na interakcję z kolejką wydarzeń, które odbywają się w systemie np. wciśnięto lub zwolniono przycisk przycisk myszy bądź klawiatury, zmieniono rozdzielczość ekranu, wyłączono program. To ten moduł komunikuje się z biblioteką SDL. Jest zależny od modułu **pygame.display**[10], który zarządza wyświetlaniem okna programu na ekranie. Okno programu jest inicjalizowane na początku działania programu na podstawie wymiarów obrazków reprezentujących tło i ziemię w grze. Sprawdzanie wydarzeń polega na iteracji kolejki, którą *pygame* udostępnia za pomocą metody *pygame.event.get*. Jeżeli zajdzie się w niej wydarzenie o typie *pygame.QUIT* albo *pygame.KEYDOWN* (został wciśnięty przycisk klawiatury) oraz *pygame.event.key* to *pygame.K_ESCAPE* (wciśnięty klawisz to escape) to podobnie jak w przypadku wykrycia kolizji gra jest zakańczana.

Dodatkowo *pygame* umożliwia komunikację ze sprzętem bezpośrednio, nie przez

sprawdzanie kolejki wydarzeń. Mysz obsługiwana jest przez moduł **pygame.mouse**. Pozwala na sprawdzenie w każdym momencie, które przyciski na myszy są wciśnięte za pomocą metody *pygame.mouse.get_pressed*, która zwraca tablicę wartości logicznych reprezentujących przyciski na myszy. Sprawdzenie, czy gracz wcisnął lewy przycisk myszy jest sprawdzane w metodzie *update* obiektu ptaka i polega na sprawdzeniu dwóch warunków:

- Czy lewy przycisk myszy jest wciśnięty
- Czy lewy przycisk myszy był wciśnięty w ostatnim obrocie głównej pętli programowej

Jeżeli przycisk nie był wciśnięty to obiekt ptaka podlatuje i ustawiana jest flaga logiczna mówiąca o tym, że lewy przycisk myszy jest wciśnięty. Flaga ta jest resetowana, gdy przy następnym obrocie głównej pętli programowej przycisk nie będzie wciśnięty. Takie sprawdzanie jest konieczne, inaczej jedno wciśnięcie lewego przycisku myszy powodowałoby, że obiekt ptaka podlatywał by do góry o dużą ilość jednostek, dopóki przycisk nie zostałby zwolniony.

Na końcu aktualizowane jest okno gry dzięki użyciu metody **pygame.display.update**, która wysyła zaktualizowany obraz na ekran. Optymalizacja jaka jest zastosowana w tej metodzie, sprawia, że aktualizowane są tylko te części obrazu, które faktycznie się zmieniły a nie całość.

Gdy gra zostaje zakończona, na ekranie rysowany przycisk reset oraz ustawiana flaga logiczna zatrzymująca wykonywanie opisanych w tym rozdziale dyrektyw dopóki nie zostanie wciśnięty lewy przycisk myszy, wtedy gra jest resetowana, to znaczy wszystkie zmienne są ustawiane do stanu początkowego co powoduje rozpoczęcie gry na nowo.

Rozdział 2

Uczenie maszynowe

2.1 Czym jest sztuczna inteligencja

Sztuczna inteligencja (ang. *artificial intelligence*) to dziedzina nauki, która zajmuje się syntezowaniem oraz analizowaniem obliczeniowych agentów, którzy potrafią podejmować inteligentne decyzje.

Agent jest czymś, co potrafi podejmować decyzje w pewnym środowisku, robić coś. Agentami są na przykład: psy, termostaty, ludzie, korporacje, samoloty czy roboty.

Mówimy, że agent działa/podejmuje decyzje inteligentnie gdy:

- Uczy się na podstawie swojego doświadczenia
- Jest elastyczny na zmieniające się środowisko i zmieniające się cele
- Decyzje, które podejmuje są właściwe dla okoliczności, w których się znajduje oraz celów, które ma osiągnąć, biorące pod uwagę krótko i długo terminowe konsekwencje

Obliczeniowy agent to taki agent, którego podjęte decyzje mogą zostać wyjaśnione za pomocą prymitywnych operacji obliczeniowych czyli takich, które mogą zostać zaimplementowane na fizycznym urządzeniu (komputerze). Takie obliczenia mogą przyjmować wiele różnych form.

Wszyscy agenci mają swoje ograniczenia. Żaden agent nie jest wszechmocny czy wszechwiedzący. Agenci mogą jedynie obserwować swoje ograniczone środowisko w bardzo wyspecjalizowanych domenach. Agenci mają skończoną pamięć oraz określoną ilość czasu na podjęcie decyzji.

Jednym z naukowych celów sztucznej inteligencji jest zrozumienie podstawowych zasad inteligentnego zachowania, które są zastosowywane zarówno do naturalnych (zwierzęta) jak i sztucznych agentów[11]. Jest on osiągany przez:

- Analizę naturalnych i sztucznych agentów
- Formułowanie i sprawdzanie hipotez dotyczących czego potrzeba by skonstruować inteligentnych agentów

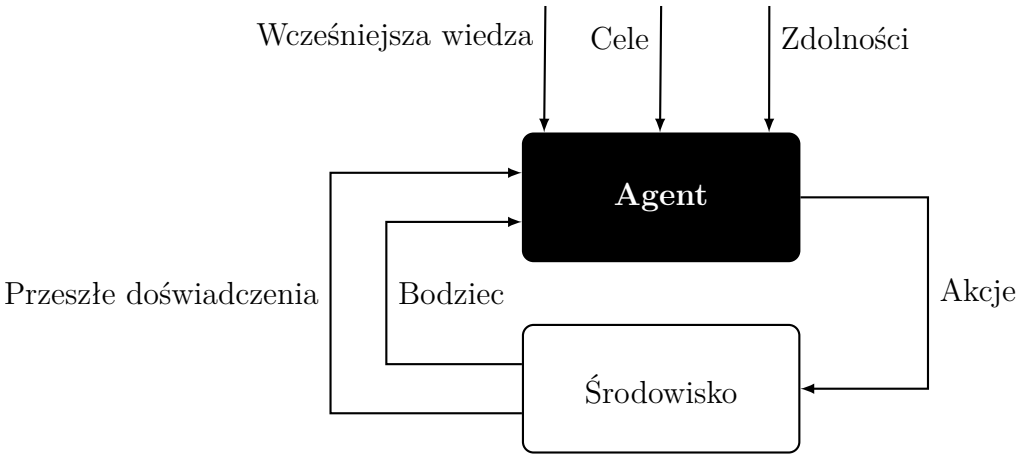
- Projektowanie, budowanie i eksperymentowanie z systemami obliczeniowymi, które wykonują zadania powszechnie uważane za wymagające inteligencji

Inżynieryjnym celem sztucznej inteligencji odpowiadającym wspomnianemu wyżej celu naukowemu jest projektowanie i syntezowanie użytecznych inteligentnych artefaktów. Chcemy tworzyć agentów, którzy będą działać inteligentnie. Tacy agenci są użyteczni w wielu problemach.

2.2 Interakcja agenta ze środowiskiem

Kwintesencją sztucznej inteligencji jest praktyczne rozumowanie, czyli rozumowanie w celu osiągnięcia jakiegoś celu. Agenta definiują: percepcja, rozumowanie oraz podejmowanie działań. Agent podejmuje działania w **środowisku**.

Agentem może być na przykład komputer z sensorami i siłownikami, innymi słowy robot, gdzie jego środowiskiem jest świat rzeczywisty. Może być to autonomiczny samochód lub jak w moim przypadku może być to program, który działa w czysto obliczeniowym środowisku to znaczy agent programowy (ang. *software agent*).



Rysunek 2.1: Schemat interakcji agenta ze środowiskiem.

Rysunek 2.1 pokazuje widok czarnej skrzynki agenta z wejściami i wyjściami. W dowolnym momencie agent jest zależny od:

- **Wcześniejszej wiedzy** o sobie samym i środowisku
- **Bodźców** otrzymanych od obecnego środowiska, które mogą zawierać obserwacje o środowisku, ale również akcje, które środowisko narzuca agentowi
- **Przeszłych doświadczeń** z poprzednich działań i bodźców lub innych danych, z których może się uczyć
- **Celów**, które musi starać się osiągnąć

- **Zdolności**, czyli prymitywnych działań, które może wykonać

W czarnej skrzynce, agent jest w pewnym stanie wiedzy, może być to wiedza o środowisku, o tym co próbuje osiągnąć i co zamierza zrobić. Agent aktualizuje ten stan bazując na bodźcach ze środowiska. Na podstawie stanu wiedzy decyduje o swoich działaniach[12].

2.3 Wiedza agenta

W najprostszym przypadku, agent w celu podjęcia decyzji pozyskuje ze środowiska (opartego o przestrzeń stanów) stan, ma określony do osiągnięcia cel i nie ma żadnych niepewności. Agent jest w stanie ustalić jak osiągnąć swój cel przeszukując przestrzeń stanów środowiska, tak by ze swojego obecnego stanu przejść do stanu, w którym cel jest osiągnięty. Mając kompletną przestrzeń stanów, próbuje znaleźć sekwencję działań, których podjęcie sprawi, że osiągnie cel, zanim podejmie jakiekolwiek działania.

Przestrzeń stanów (ang. *state space*) to jedno z ogólnych sformułowań inteligentnych działań. **Stan** zawiera wszystkie informacje potrzebne do przewidzenia efektów danego działania i ustalenia, czy tenże stan jest potrzebny do osiągnięcia celu. Przeszukiwanie przestrzeni stanów zakłada, że:

- Agent ma całkowitą wiedzę o przestrzeni stanów i zakłada, że istnieje stan, w którym będzie obserwował to w jakim stanie się znajduje – jest całkowita obserwowalność
- Agent ma zbiór akcji, które mają znane, deterministyczne efekty na środowisko
- Agent może ustalić czy stan spełnia cel

Rozwiązanie to sekwencja działań, które przeniosą agenta z jego obecnego stanu do stanu, który spełnia cel.

2.4 Proces decyzyjny

W poprzednim podrozdziale założyłem, że rozwiązanie to skończona sekwencja działań. Co jednak jeżeli problem, który agent musi rozważyć to trwający proces lub nie wiadomo ile działań agent będzie musiał jeszcze podjąć by osiągnąć cel. Takie problemy nazywane są **problemami nieskończonego horyzontu** (ang. *infinite horizon problems*), gdzie proces może trwać wiecznie, jak na przykład problem grania we "Flappy Bird". Albo też **problemami nieokreślonego horyzontu** (ang. *indefinite horizon problems*), w których agent w końcu zakończy podejmowanie działań, ale nie wie kiedy to nastąpi.

Dla trwających procesów, może nie mieć sensu rozważanie użyteczności na końcu procesu ponieważ agent może nigdy nie dojść do końca. Zamiast tego agent może otrzymać

sekwencję **nagród** (ang. *rewards*). Negatywne nagrody nazywane są **karami** (ang. *punishments*). Problemy niezdefiniowanego horyzontu mogą być wymodelowane używając stanu stop. **Stan stop** bądź **stan absorbujący** to stan, w którym wszystkie działania nie mają żadnego efektu. To znaczy jeżeli agent znajdzie się w tym stanie to wszystkie jego działania dostają zerową nagrodę ze środowiska.

2.4.1 Łańcuchy Markova

Mówimy, że zmienna X jest **warunkowo niezależna** od losowej zmiennej Y zakładając, że mamy zbiór losowych zmiennych Z_s jeżeli

$$P(X|Y, Z_s) = P(X|Z_s)$$

gdzie prawdopodobieństwa są dobrze zdefiniowane (żadne nie jest równe 0). Zapis $P(X|Y, Z_s)$ oznacza prawdopodobieństwo zajścia X pod warunkiem, że zajdzie Y oraz Z_s. To oznacza, że dla każdego $x \in$ dziedzina(X), dla każdego $y \in$ dziedzina(Y) oraz dla każdego $z \in$ dziedzina(Z_s) jeżeli $P(Y = y \wedge Z_s = z) > 0$,

$$P(X = x|Y = y \wedge Z_s = z) = P(X = x|Z_s = z).$$

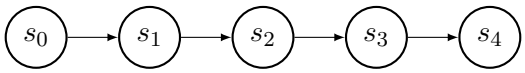
To znaczy, mając wartość każdej zmiennej Z_s, wiedza o prawdopodobieństwie Y nie ma wpływu na przekonanie o X.

Pojęcie warunkowej niezależności jest użyte by nadać zwięzłą reprezentację wielu domenom. Chodzi o to, że jeżeli mamy daną losową zmienną X, to może istnieć zbiór zmiennych W, które bezpośrednio wpływają na wartość X. W pewnym sensie X jest warunkowo niezależne od pozostałych zmiennych biorąc pod uwagę zmienne W. Zbiór lokalnie wpływających na X zmiennych jest nazwany **ogrodzeniem Markova** (ang. *Markov blanket*). Ta lokalność jest wykorzystywana w sieciach przekonań[13].

Sieć przekonań (ang. *belief network*), również nazywana **siecią Bayesa** to acykliczny graf skierowany, gdzie wierzchołki są losowymi zmiennymi. Ponadto istnieje ścieżka od każdego elementu ze zbioru rodzice(X_i) do X_i. Z siecią przekonań powiązany jest zbiór rozkładu prawdopodobieństw warunkowych, które określają warunkowe prawdopodobieństwo między każdą zmienną a jej rodzicami (co włącza wcześniejsze prawdopodobieństwa zmiennych bez rodziców). A więc sieć przekonań składa się z:

- Grafu acyklicznego skierowanego, gdzie każdy wierzchołek jest oznaczony losową zmienną
- Dziedziny dla każdej losowej zmiennej
- Zbioru rozkładu prawdopodobieństw warunkowych przypisującego $P(X|\text{rodzice}(X))$ dla każdej zmiennej X

Łańcuch Markova to sieć przekonań z losowymi zmiennymi ułożonymi w sekwencję, gdzie każda zmienna bezpośrednio zależy od swojego poprzednika w sekwencji. Łańcuchy Markova są używane do reprezentowania sekwencji wartości jak na przykład sekwencja stanów w dynamicznym systemie lub sekwencja słów w zdaniu. Każdy punkt w sekwencji jest nazywany **etapem** (ang. *stage*). Rysunek 2.2 pokazuje ogólny przypadek łańcucha



Rysunek 2.2: Łańcuch Markova jako sieć przekonań.

Markova jako sieć przekonań. Sieć ma pięć etapów, ale nie musi zatrzymywać się na s_4, może rozrastać się w nieskończoność. Sieci przekonań opierają się na założeniu niezależności:

$$P(S_{i+1}|S_0, \ldots, S_i) = P(S_{i+1}|S_i),$$

które jest nazywane **założeniem Markova**

Często sekwencje są rozłożone w czasie, wtedy S_t reprezentuje stan w momencie t. Intuicyjnie S_t przekazuje całą informację o historii, która mogłaby wpłynąć na przyszłe stany. Niezależność przypuszczenia w łańcuchu Markova może być rozumiana jako "biorąc pod uwagę teraźniejszość – przyszłość jest warunkowo niezależna od przeszłości"

Mówimy, że łańcuch Markova jest **modelem stacjonarnym** (ang. *stationary model*) oraz **modelem czasowo homogenicznym** (ang. *time-homogenous model*) jeżeli wszystkie zmienne mają tę samą dziedzinę oraz wszystkie prawdopodobieństwa przejść są takie same dla każdego etapu, to znaczy:

$$\forall i \geq 0, P(S_{i+1}|S_i) = P(S_1|S_0)$$

By określić stacjonarny łańcuch Markova, podano dwa prawdopodobieństwa warunkowe:

- $P(S_0)$ specyfikuje wszystkie początkowe warunki
- $P(S_{i+1}|S_i)$ specyfikuje **dynamikę**, która jest taka sama dla każdego $i \geq 0$

Stacjonarne łańcuchy Markova interesują nas z kilku względów:

- Zapewniają prosty model, który jest łatwy do określenia
- Założenie stacjonarności jest często naturalnym modelem, ponieważ dynamika środowiska zazwyczaj nie zmienia się w czasie.
- Sieć może się rozrastać w nieskończoność. Określenie małej liczby parametrów daje nam nieskończoną sieć[14].

2.4.2 Sieć decyzyjna

Zazwyczaj agent nie podejmuje decyzji bez wcześniejszych obserwacji środowiska, nie podejmuje też tylko jednej decyzji. Bardziej typowy scenariusz wygląda następująco – agent obserwuje swoje środowisko, decyduje jaką akcję podjąć, wykonuje tę akcję, obserwuje zmienione po jego akcji środowisko, i tak dalej. Następujące po sobie (sekwencyjne) akcje agenta mogą zależeć od tego co agent zaobserwuje, a to co zaobserwuje może zależeć od poprzednich podjętych przez niego akcji. W takim wypadku, często jedynym powodem dla wykonania jakiejś akcji jest to by dostarczyć kontekst dla przyszłych akcji. Akcje wykonywane tylko po to by nabyć informacje nazywane są **akcjami poszukującymi informacji**. Formalnie nie muszę odróżniać akcji poszukujących informacji od innych akcji. Zazwyczaj podjęte akcje będą prowadziły zarówno do pozyskania nowych informacji jak i wywarciu jakiejś zmiany na środowisku.

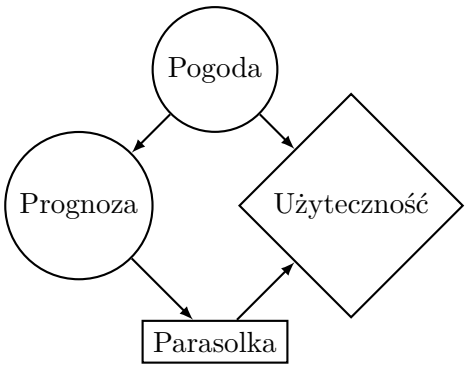
Sekwencyjny model decyzyjny specyfikuje:

- Jakie akcje są dostępne dla agenta w danym etapie
- Jaka informacja jest, albo będzie dostępna dla agenta, kiedy będzie musiał podjąć akcję
- Efekty akcji na środowisko
- To jak pożądane te efekty będą z perspektywy osiągnięcia celu

Sieć decyzyjna zwana również **diagramem wpływu** to graficzna reprezentacja skończonego problemu decyzji sekwencyjnych[15]. Sieć decyzyjna rozszerza sieć przekonań by zawierać zmienne decyzji (akcji) i użyteczności.

W szczególności, sieć decyzyjna to skierowany graf acykliczny, z następującymi wierzchołkami:

- **Wierzchołek decyzji**, rysowany jako prostokąt reprezentuje zmienne decyzji. Agent może wybrać wartość każdej ze zmiennych decyzji.
- **Wierzchołek szansy**, rysowany jako okrąg reprezentuje zmienne losowe. To są te same wierzchołki co w sieciach przekonań. Każdy wierzchołek szansy ma powiązaną ze sobą dziedzinę i warunkowe prawdopodobieństwo biorące pod uwagę rodziców wierzchołka. Tak jak w sieciach przekonań, rodzice wierzchołka reprezentują warunkową zależność: zmienna jest niezależna od innych zmiennych nie będących jej potomkami. W sieci decyzyjnej zarówno wierzchołki decyzji i wierzchołki szansy mogą być rodzicami wierzchołka szansy.
- **Wierzchołek użyteczności**, rysowany jako romb reprezentuje użyteczność. Zarówno wierzchołki szansy jak i wierzchołki decyzji mogą być rodzicami wierzchołka użyteczności. Mówimy tutaj o użyteczności w odniesieniu do wyników akcji agenta – tego jaki wpływ na jego cel mają wyniki jego akcji.



Rysunek 2.3: Sieć decyzyjna reprezentująca decyzję, czy wziąć parasolkę.

Krawędzie wchodzące do wierzchołków decyzyjnych reprezentują informację, która będzie dostępna, jeżeli decyzja zostanie podjęta. Krawędzie wchodzące do wierzchołków szansy reprezentują zależność probabilistyczną. Krawędzie wchodzące do wierzchołków użyteczności reprezentują od czego ta użyteczność zależy.

Mówimy, że **agent nie zapomina**, jeżeli decyzje, które podjął są uporządkowane w czasie, i agent pamięta swoje poprzednie decyzje i każdą informację, która była dostępna przy poprzedniej decyzji.

Mówimy, że **sieć decyzyjna nie zapomina**, jeżeli jej wierzchołki decyzji są całkowicie uporządkowane, to znaczy, jeżeli wierzchołek decyzyjny D_i jest przed D_j to znaczy, że D_i jest rodzicem D_j i każdy rodzic D_i jest również rodzicem D_j. A to oznacza, że każda informacja dostępna dla D_i jest dostępna dla każdej późniejszej decyzji w sekwencji i akcja wybrana dla decyzji D_i jest częścią informacji dostępnej dla każdej późniejszej decyzji.

2.4.3 Proces decyzyjny Markova

Proces decyzyjny Markova, może być rozumiany jako łańcuch Markova powiększony o działania agenta oraz nagrodę dla agenta. W każdym etapie agent decyduje jakie działanie podjąć, nagroda i nowy stan zależą od podjętego przez agenta działania oraz poprzedniego stanu.

Rozważę jedynie stacjonarne modele łańcuchu Markova, gdzie przejścia między stanami oraz nagrody nie zależą od czasu.

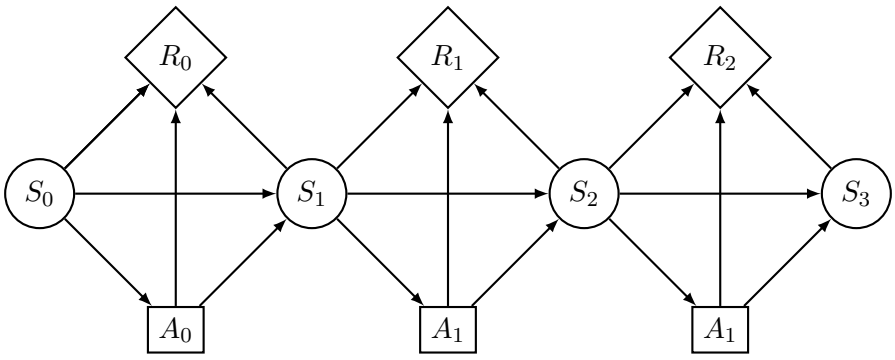
Proces decyzyjny Markova (ang. *Markov decision process (MDP)*) składa się z:

- S – zbiór stanów środowiska
- A – zbiór decyzji (akcji możliwych do podjęcia przez agenta)
- $P : S \times S \times A \to [0, 1]$ – funkcja, która określa **dynamikę**. Zapis $P(s'|s, a)$ oznacza prawdopodobieństwo, że agent przejdzie do stanu s' pod warunkiem, że był w stanie

s i podjął akcję a, czyli:

$$\forall s \in S \; \forall a \in A \sum_{s' \in S} P(s'|s,a) = 1$$

- $R : S \times S \times A \times S \to \Re$, gdzie $R(s,a,s')$ to **funkcja nagrody**, dająca oczekiwaną natychmiastową nagrodę za wykonanie działania a i przejścia do stanu s' ze stanu s. Czasem wygodnie jest użyć zapisu $R(s,a)$ – oczekiwana wartość wykonania a w stanie s, która równa jest $R(s,a) = \sum_{s'} R(s,a,s') * P(s'|s,a)$.



Rysunek 2.4: Sieć decyzyjna reprezentująca skończoną część MDP.

By zdecydować jaką akcję podjąć agent porównuje różne sekwencje nagród. Najczęstszym sposobem jest konwersja sekwencji nagród do jednej liczby zwanej **nagrodą skumulowaną** (ang. *cumulative reward*). Aby to zrobić agent łączy obecną nagrodę z innymi nagrodami w przyszłości. Załóżmy, że agent otrzyma następującą sekwencję nagród: $r_1, r_2, r_3, r_4, \ldots$

Trzema najczęstszymi sposobami połączenia sekwencji nagród w nagrodę skumulowaną V są

Nagroda całkowita (ang. *total reward*) $V = \sum_{i=1}^{\infty} r_i$. W tym przypadku nagroda skumulowana to suma wszystkich nagród. Metoda ta działa kiedy możemy zagwarantować, że suma jest skończona. Jeżeli nie możemy tego założyć to nie możemy określić, która sekwencja nagród jest preferowana. Na przykład: Sekwencja nagród po 1zł sumuje się do tej samej V co sekwencja nagród 100zł (obie są nieskończone). Nagroda jest skończona na przykład w przypadku, gdzie występują stany stopu a agent ma zawsze nie zerowe prawdopodobieństwo wejścia w taki stan.

Średnia nagród (ang. *average reward*) $V = \lim_{n\to\infty} \frac{(r_1+\cdots+r_n)}{n}$. W tym przypadku, skumulowana nagroda agenta to wartość średnia otrzymanych przez niego nagród, uśredniona dla każdego odcinka czasu. Dopóki nagrody są skończone dopóty V też będzie wartością skończoną.

Dyskontowa nagroda (ang. *discount reward*) $V = r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots + \gamma^{i-1} r_i + \ldots$, gdzie γ to stopa dyskontowa będąca liczbą w zakresie $0 \leq \gamma < 1$. Przy tych założeniach

przyszłe nagrody są warte mniej niż obecna nagroda. Jeżeli γ byłaby równa 1, to byłby to przypadek całkowitej nagrody. Jeżeli $\gamma = 0$ to znaczy, że agent ignoruje wszystkie przyszłe nagrody. To, że $0 \leq \gamma < 1$ gwarantuje nam, że jeżeli nagrody są wartościami skończonymi to skumulowana nagroda również będzie skończona. Nagrodę dyskontową można zapisać jako:

$$\begin{aligned} V &= \sum_{i=1}^{\infty} \gamma^{i-1} r_i \\ &= r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots + \gamma^{i-1} r_i + \ldots \\ &= r_1 + \gamma(r_2 + \gamma(r_3 + \ldots)). \end{aligned} \tag{2.1}$$

Załóżmy, że V_k to nagroda skumulowana od czasu k:

$$\begin{aligned} V_k &= r_k + \gamma(r_{k+1} + \gamma(r_{k+2} + \ldots)) \\ &= r_k + \gamma V_{k+1} \end{aligned} \tag{2.2}$$

W dalszej części pracy będę rozważał nagrodę skumulowaną jako nagrodę dyskontową.

2.4.4 Funkcja polityki

Polityka (ang. *policy*) precyzuje co agent powinien robić w stanie, w którym obecnie się znajduje[16]. **Stacjonarna polityka** to funkcja $\pi : S \to A$ (gdzie S to zbiór stanów środowiska a A to zbiór dostępnych dla agenta akcji).

Biorąc pod uwagę kryterium przyznania nagrody polityka zawiera spodziewaną nagrodę skumulowaną dla każdego stanu. Niech $V^{\pi}(s)$ będzie spodziewaną nagrodą skumulowaną użycia polityki π w stanie s. To znaczy, jaką nagrodę skumulowaną agent oczekuje, że otrzyma po zastosowaniu tej polityki w tym stanie. Mówimy, że π jest **polityką optymalną**, jeżeli nie istnieje polityka π' dla dowolnego stanu s taka, że $V^{\pi'}(s) > V^{\pi}(s)$. To znaczy, taka polityka, która ma większą spodziewaną nagrodę skumulowaną dla każdego stanu, niż jakakolwiek polityka optymalna.

Dla problemów nieskończonego horyzontu, stacjonarny proces decyzyjny Markova zawsze ma optymalną stacjonarną politykę.[17]

Jak zatem obliczyć nagrodę skumulowaną optymalnej polityki? Niech $Q^*(s, a)$, gdzie s to stan, a to akcja będzie oczekiwaną nagrodą skumulowaną wykonania a w stanie s i potem stosowania optymalnej polityki. Niech $V^*(s)$ będzie oczekiwaną nagrodą skumulowaną używania optymalnej polityki począwszy od stanu s.

$$\begin{aligned} Q^*(s, a) &= \sum_{s'} P(s'|s, a)(R(s, a, s') + \gamma V^*(s')) \\ &= R(s, a) + \gamma \sum_{s'} P(s'|s, a)\gamma V^*(s'). \end{aligned} \tag{2.3}$$

gdzie $R(s,a) = \sum_{s'} P(s'|s,a)R(s,a,s')$ a s' to stan, do którego przeszedł agent ze stanu s po wykonaniu akcji a. $V^*(s)$ otrzymywane jest przez wykonywanie akcji, która daje najwyższą nagrodę skumulowaną w każdym stanie:

$$V^*(s) = \max_a Q^*(s,a). \tag{2.4}$$

Optymalna polityka π^* to jedna z polityk, która daje najlepszą wartość w każdym stanie:

$$\pi^*(s) = \arg\max_a Q^*(s,a). \tag{2.5}$$

Gdzie $\arg\max_a Q^*(s,a)$ to funkcja od stanu s a jej wartością jest akcja a, w wyniku wykonania której otrzymamy największą wartość $Q^*(s,a)$.

2.5 Uczenie przez wzmacnianie

Agent uczony przez wzmacnianie działa w środowisku, obserwuje jego stan oraz otrzymuje nagrody. Na podstawie tych informacji decyduje co ma zrobić. Agenta charakteryzują następujące cechy:

- Agent ma dane możliwe stany, w których może się znaleźć oraz zbiór akcji, które może wykonać.
- Za każdym razem, po zaobserwowaniu stanu środowiska wykonuje akcję
- Celem agenta jest zmaksymalizowanie nagrody dyskontowej, dla jakiejś stopy dyskontowej γ

Uczenie przez wzmacnianie może być sformalizowane pod kątem procesu Decyzyjnego Markova, w którym agent początkowo zna tylko zbiór możliwych stanów oraz zbiór możliwych akcji. Dynamika $P(s'|a,s)$ oraz funkcja nagrody $R(s,a)$, nie są agentowi znane. Tak jak w procesie decyzyjnym Markova po każdej akcji agent obserwuje stan, w którym się znajduje i otrzymuje nagrodę.

2.6 Różnice czasowe

Załóżmy, że mamy sekwencję wartości numerycznych $v_1, v_2, v_3, \ldots$ i celem jest przewidzenie następnej wartości biorąc pod uwagę poprzednie wartości. Jednym ze sposobów jest obliczenie bieżącego przybliżenia oczekiwanej wartości v_i. Może to być zaimplementowane poprzez utrzymywanie bieżącej średniej:

Niech A_k będzie oszacowaną oczekiwaną wartością bazującą na pierwszych k punktach danych $v_1, \ldots, v_k$. Przybliżeniem może być po prostu średnia:

$$A_k = \frac{v_1 + \cdots + v_k}{k}$$

. Po przemnożeniu przez k:

$$\begin{aligned} k * A_k &= v_1 + \cdots + v_{k-1} + v_k \\ &= (k-1)A_{k-1} + v_k \end{aligned} \tag{2.6}$$

Dzieląc przez k otrzymujemy:

$$A_k = \left(1 - \frac{1}{k}\right) * A_{k-1} + \frac{v_k}{k} \tag{2.7}$$

Niech $\alpha_k = \frac{1}{k}$, wtedy:

$$\begin{aligned} A_k &= (1 - \alpha_k) * A_{k-1} + \alpha_k * v_k \\ &= A_{k-1} + \alpha_k * (v_k - A_{k-1}) \end{aligned} \tag{2.8}$$

Różnicę $v_k - A_{k-1}$ nazywamy **błędem różnicy czasowej**. Określa jak bardzo różni się nowa wartość v_k, od starej prognozy A_{k-1}. Stare przybliżenie A_{k-1} jest zaktualizowane o α_k razy błąd różnicy czasowej by otrzymać nowe przybliżenie A_k. Interpretacja wzoru na różnicę czasową jest następująca: jeżeli nowa wartość jest większa niż poprzednia, zwiększ przewidywaną wartość; jeżeli nowa wartość jest mniejsza niż poprzednia zmniejsz przewidywaną wartość. Zmiana jest proporcjonalna do różnicy między nową wartością a poprzednią przewidywaną wartością.

W uczeniu przez wzmacnianie, wartości (skumulowane nagrody) są efektami akcji. Nowsze wartości są bardziej dokładne od poprzednich ponieważ agent się uczy i przez to nowsze wartości powinny mieć większą wagę.

Rozdział 3

Algorytm Q-Learning

Autorem algorytmu Q-learning jest Christ Watkins, który zaprezentował go w swojej pracy doktorskiej zatytułowanej : "Learning from Delayed Rewards"[18]. W Q-learningu agent próbuje nauczyć się optymalnej polityki z historii interakcji ze środowiskiem. **Historia** agenta to sekwencja stanu-akcji-nagrody:

$$\langle s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, a_3, r_4, \dots \rangle$$

co oznacza, że agent był w stanie s_0, wykonał akcję a_0 co spowodowało przyznaniem mu nagrody r_1 i przejścia do stanu s_1, następnie wykonał akcję a_1, otrzymał nagrodę r_2 i przeszedł do stanu s_2 i tak dalej.

Historię interakcji traktujemy jako sekwencję doświadczeń, gdzie **doświadczenie** to krotka:

$$\langle s, a, r, s' \rangle$$

co oznacza, że agent był w stanie s, wykonał akcję a za co otrzymał nagrodę r i przeszedł do stanu s'. Te doświadczenia będą danymi, na podstawie których agent będzie się uczył co robić. Tak jak w planowaniu decyzji celem jest by agent zmaksymalizował swoją nagrodę skumulowaną, która zazwyczaj jest nagrodą dyskontową.

Q-learning wykorzystuje różnice czasowe do obliczenia przybliżonej wartości $Q^*(s, a)$. W Q-learningu agent utrzymuje tablicę $Q[S, A]$, gdzie S to zbiór możliwych dla danego środowiska stanów, a A to zbiór akcji, które agent może podjąć. $Q[s, a]$ reprezentuje obecne przybliżenie $Q^*(s, a)$

Doświadczenie $\langle s, a, r, s' \rangle$ zapewnia jeden punkt danych dla wartości $Q(s, a)$. Zawiera informację o tym, że agent otrzymał przyszłą wartość $r + \gamma V(s')$, gdzie $V(s') = \max_{a'} Q(s', a')$ (tutaj a' to akcja podjęta dla stanu s' czyli taka akcja, która daje najlepszą wartość Q). To jest aktualną rzeczywistą nagrodę plus dyskontowa oszacowana przyszła nagroda skumulowana. Agent może użyć równania na różnicę czasową 2.8 by zaktualizować swoje oszacowanie dla $Q(s, a)$:

$$Q[s, a] := Q[s, a] + \alpha * \Big(r + \gamma \max_{a'} Q[s', a'] - Q[s, a]\Big) \tag{3.1}$$

albo, równoważnie

$$Q[s,a] := (1-\alpha) * Q[s,a] + \alpha * \left(r + \gamma \max_{a'} Q[s',a']\right) \tag{3.2}$$

Algorithm 1 Algorytm Q-Learning[19]

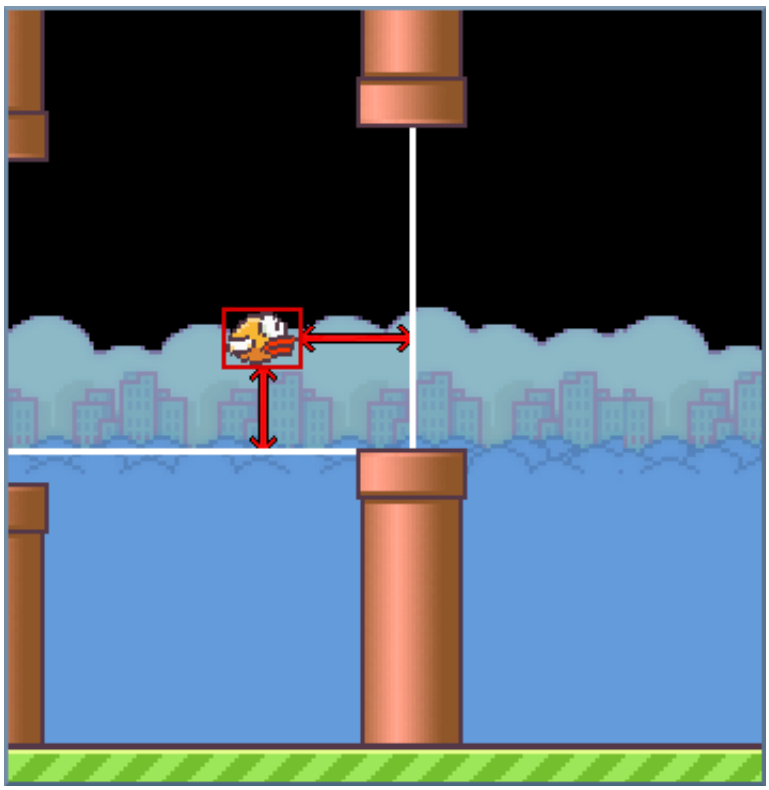
```
1: procedure Q-Learning(S, A, γ, α)
2:     Zmienne wejściowe
3:         S jest zbiorem stanów
4:         A jest zbiorem akcji
5:         γ jest stopą dyskontową
6:         α jest rozmiarem jednego kroku
7:     Zmienne lokalne
8:         tablica liczb rzeczywistych Q[S, A]
9:         stany s, s'
10:        akcja a
11:    zainicjalizuj Q[S, A] zerami
12:    zaobserwuj obecny stan s
13:    powtarzaj
14:        wybierz akcję a
15:        wykonaj(a)
16:        otrzymaj nagrodę r i zaobserwuj stan s'
17:        Q[s, a] := Q[s, a] + α * (r + γ * max_a' Q[s', a'] − Q[s, a])
18:        s := s'
19:    do zakończenia
20: end procedure
```

Q-Learningowy algorytm uczy się (przybliżenia) optymalnej Q-funkcji tak długo jak agent wystarczająco eksploruje i nie ma ograniczenia co do tego ile razy może spróbować wykonania akcji w dowolnym stanie.

3.1 Implementacja Q-Learningu

W mojej implementacji Q-Learningu, **przestrzeń stanów** zdyskretyzowałem (to znaczy ograniczyłem do zmiennych o skończonej ilości) nad następującymi parametrami:

- Odległość w pionie obiektu ptaka od obiektu najbliższej od prawej dolnej rury
- Odległość w poziomie obiektu ptaka od obiektu najbliższej od prawej dolnej rury
- Prędkość spadania obiektu ptaka.



Rysunek 3.1: Wizja agenta.

Te informacje są zwracane bezpośrednio agentowi od silnika gry, sam **agent nie ma zaimplementowanych** algorytmów wizji komputerowej.

Jak widać na rysunku 3.1 agent musi rozpatrywać tylko obiekty rur, które są po jego prawej stronie, te po lewej zostały ominięte co oznacza, że agent osiągnął swój cel w odniesieniu do nich. Aby stwierdzić, który obiekt rury jest najbliższym z prawej nie mogę jak poprzednio (gdzie wystarczyło ustalić, który obiekt rury jest najbliżej, nie ważne czy z lewej czy z prawej) wziąć obiektu o indeksie zero z grupy dolnych obiektów rur. Przypomnę, że grupa obiektów rur działa jak tablica, gdzie obiekty są ponumerowane indeksami od zera w odniesieniu ich pozycji na ekranie – obiekt najbardziej z lewej ma indeks zero. Istnieje scenariusz, w którym agent dopiero co ominął obiekt rury, ale powinien zacząć już rozpatrywać następne obiekty z prawej strony. Rozpatrywanie obiektu o indeksie zero z grupy obiektów rur byłoby tutaj błędne. Wystarczy rozważyć różnicę między współrzędną x obiektu ptaka a obiektu rury o indeksie zero z grupy obiektów rur dolnych. Jeżeli ta jest większa niż -30 jednostek to oznacza, że obiekt został przez agenta ominięty i najbliższą z prawej rurą będzie ta o indeksie jeden z grupy obiektów rur. W przeciwnym wypadku najbliższa będzie ta o indeksie zero. Skąd liczba 30? Jest to lekko ponad połowa szerokości obiektu rur w pixelach.

Przestrzeń stanów jest inicjalizowana w następujący sposób:

Listing 3.1: Inicjalizacja Q-Values

```
qval = {}
for x in list(range(-80, 510, 10)):
    for y in list(range(-300, 800, 10)):
        for v in range(-10, 9):
            qval[str(x) + "_" + str(y) + "_" + str(v)] = [0, 0]
```

Gdzie **qval** to słownik, którego kluczami są napisy w formacie X_Y_V, gdzie X to różnica w poziomie, Y to różnica w pionie, a V to prędkość spadania. Jest to odpowiednik tablicy $Q[S, A]$ z algorytmu 1. Wartości X są generowane co dziesięć w zakresie $[-80, 510)$, wartości Y co dziesięć w zakresie $[-300, 800)$, a wartość V w zakresie $[-10, 9)$. O tym dlaczego wartości X i Y są generowane co dziesięć piszę w następnym podrozdziale. Wartość V ma taki zakres, gdyż jest tak ograniczona przez sam program gry, gdzie wartość 8 oznacza, że obiekt ptaka spada najszybciej jak może, a wartość -10 to, że podlatuje do góry.

Agent ma możliwość podjąć jedynie **dwie akcje** – kliknąć lewy przycisk myszy (to znaczy podlecieć), albo nie robić nic i pozwolić obiektowi ptaka zmniejszyć wysokość.

Takim kluczom jest przypisywana tablica z dwoma zerami, gdzie indeks zerowy jest Q jeżeli dla tego stanu nie została podjęta akcja, a indeks jeden jest Q dla tego stanu, jeżeli agent zdecydował się podjąć akcję. Te wyzerowane wartości Q stanowią jedynie symbol zastępczy, potem będą wyliczane ze wzoru 3.1.

Funkcja nagrody R przyznaje 1 punkt za każdym razem, kiedy obiekt ptaka sterowany przez agenta żyje oraz -1000 punktów jeżeli nie żyje.

Jedyną zasadniczą modyfikacją jakiej dokonałem względem algorytmu 1, jest to, że zamiast aktualizować Q (krok z linii 17 w algorytmie) co każde zaobserwowane doświadczenie, obliczam je od tyłu po każdej rozegranej przez agenta grze – tak, że Q jest obliczane od ostatniego doświadczenia do pierwszego. Pomyślałem, że to pomoże w szybszym propagowaniu informacji o "złym stanie" to znaczy takim, gdzie agent wybrał akcję, która doprowadziła do śmierci obiektu ptaka.

Listing 3.2: Zaimplementowany algorytm Q-Learning

```
def update_scores():
    history = list(reversed(moves))

    t = 1
    for experience in history:
        state = experience[0]
        act = experience[1]
        res_state = experience[2]

        if t == 1 or t == 2:
            cur_reward = reward[1]
        else:
            cur_reward = reward[0]
```

```
14
15          qvalues[state][act] = (1 - lr) * (qvalues[state][act]) + lr * \
16        (cur_reward + discount * max(qvalues[res_state]))
17
18          t += 1
19      moves = []
```

Na listingu 3.2 *moves* to tablica doświadczeń (s, a, s'), gdzie s to poprzedni stan, a to poprzednia akcja jaką wykonał agent a s' to obecny stan. Tablica ta jest uzupełniana o nowe doświadczenie, za każdym razem, gdy program gry wywołuję metodę agenta proszącą go o wybór akcji. *reward* zwraca wartość 1 dla indeksu 0, oraz wartość -1000 dla indeksu 1. To oznacza, że dla ostatnich dwóch stanów, jest przyznawana nagroda -1000. Dzieje się tak ponieważ to one, skoro były ostatnie (a metoda jest wywoływana w momencie końca gry) to doprowadziły do porażki. Pozostałym stanom przyznawana jest nagroda 1, ponieważ akcje podjęte dla tych stanów sprawiały, że agent omijał obiekty rur. *discount* to liczba określająca stopę dyskontową i jest to stała wynosząca 0.7 Oczywiście dla każdego doświadczenia aktualizowana jest wartość Q zgodnie ze wzorem 3.1 i wypisywana odpowiednio do słownika *qvalues* dla odpowiedniego stanu i akcji.

Listing 3.3: Metoda wybierająca akcję agenta

```
def act(xdif, ydif, vel):
    state = map_state(xdif, ydif, vel)
    moves.append((last_state, last_action, state))
    last_state = state

    if qvalues[state][0] >= qvalues[state][1]:
        last_action = 0
        return 0
    else:
        last_action = 1
        return 1
```

Program gry przystosowany dla agenta, różni się niewiele od tego zaprezentowanego na diagramie 1.2. Różnice są następujące:

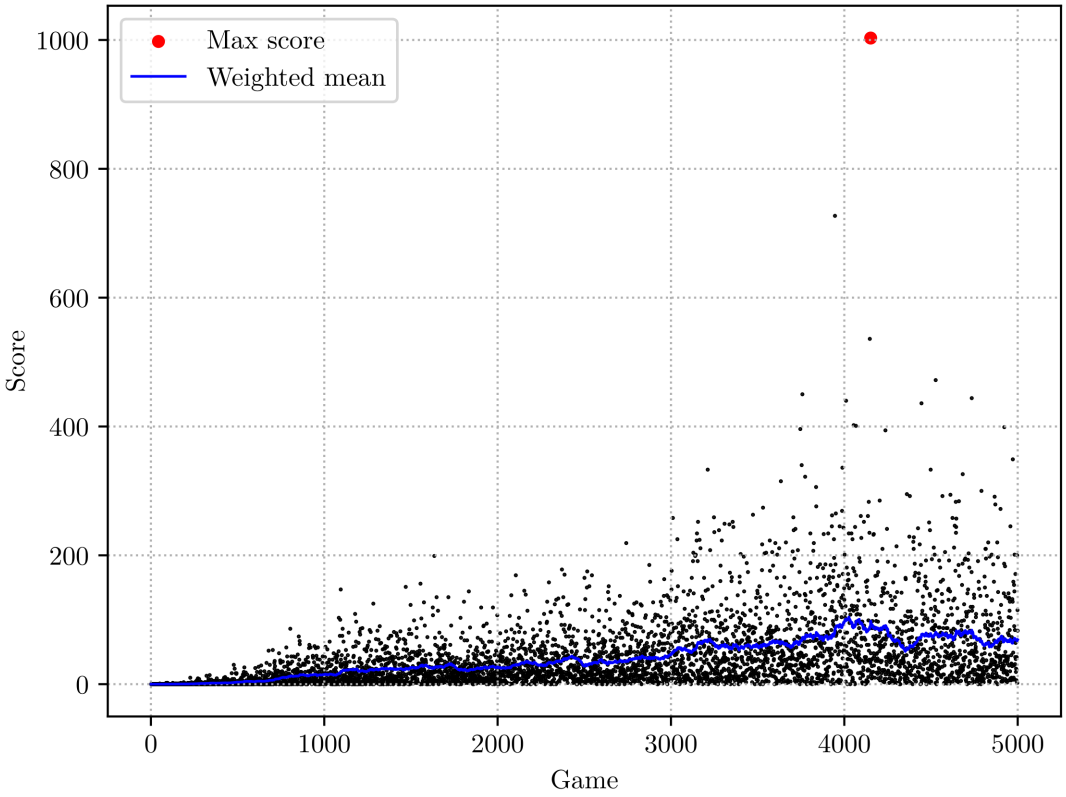
- Nie jest sprawdzane, żadne wejście
- Co obrót głównej pętli programowej agent jest pytany o swoją akcję, do podjęcia której program daje agentowi wcześniej wspomniane wartości, które stanowią "obserwację" agenta. Sam wybór akcji polega na zmapowaniu otrzymanych ze środowiska wartości do tych, które zostały zainicjalizowane tak jak jest to pokazane na listingu 3.1. Czyli w tym przypadku zaokrąglenie wartości X i Y do części dziesiątych. A następnie wybieranie akcji, jak pokazuje to listing 3.3 – agent wybiera akcję 1 (kliknij lewy przycisk myszy) albo 0 (nie rób nic) w zależności od tego dla której akcji wartość Q jest większa.

- W przypadku końca gry wywoływana jest metoda pokazana na listingu 3.2, po czym zaczyna się nowa gra. Gra jest resetowana tyle razy ile poda się przy uruchamianiu programu.

Rozdział 4

Testy i rezultaty

4.1 Optymalizacja pod względem rozmiaru przestrzeni stanów



Rysunek 4.1: Wykres gry od wyniku.

Przestrzeń stanów to zbiór S, którego elementy wyglądają następująco:

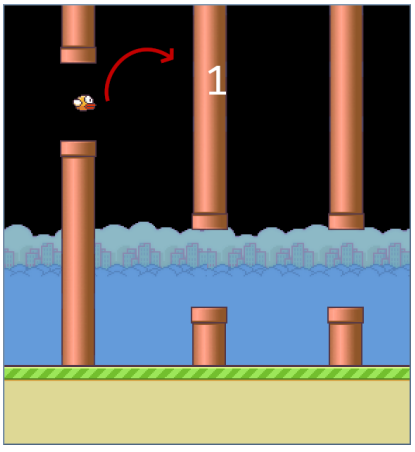
$$x \in [-80, 509] \cap \mathbb{Z}\ y \in [-300, 799] \cap \mathbb{Z}\ v \in [-10, 8] \cap \mathbb{Z}\ \ \{x, y, v\}$$

Co daje nam $(509+80+1)*(300+799+1)*(10+8+1) = 12331000$ stanów. Wyuczenie agenta – to znaczy wypełnienie wartości Q dla większości stanów zajęłoby bardzo dużo czasu. Postanowiłem poświęcić precyzję z jaką agent wykonuje akcje na rzecz ograniczenia ilości stanów. Dlatego jak w wcześniej wspominałem stany są inicjalizowane z wartościami x i y generowanymi co 10. W ten sposób przestrzeń stanów ma $(50 + 8 + 1) * (30 + 79 + 1) * (10 + 8 + 1) = 123310$ stanów, co jak można się spodziewać jest prawie stukrotnym pomniejszeniem ilości elementów w przestrzeni stanów. Środowisko dalej zwraca agentowi "pełne liczby", jednak ten zanim podejrzy wartość Q mapuje te wartości zaokrąglając je w dół do części dziesiątych. Tak zaimplementowany algorytm i grę uruchomiłem po czym otrzymałem wyniki widoczne na wykresie 4.1.

Wykres 4.1 pokazuje jaką ilość punktów agent otrzymywał w danej grze – te dane są zaznaczone na wykresie czarnymi punktami. Agent rozegrał 5000 gier. Maksymalny wynik jaki udało mu się osiągnąć to 1003 w grze o numerze 4152. Jest to jednak bardzo odizolowany przypadek. Niebieską linia to wykres średniej kroczącej. Gdzie średnia ta jest liczona w każdym punkcie dla ostatnich 100 gier (z wyjątkiem 100 pierwszych gier). Po 5000 gier średnia wynosiła $69,09$.

4.2 Optymalizacja algorytmu

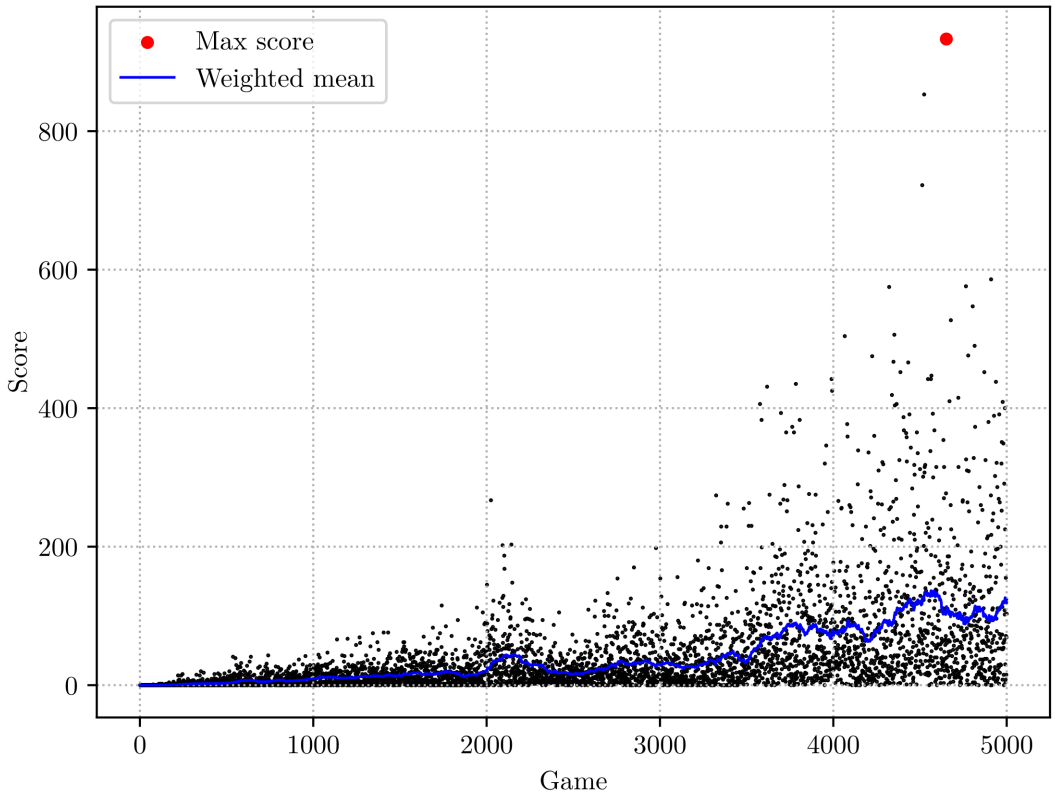
Zauważyłem, że podczas uczenia często miał miejsce scenariusz przedstawiony na rysunku 4.2. Gdy wygenerowana szczelina znajdowała się wysoko, a następna za nią zde-



Rysunek 4.2: Agent wybiera złą akcję.

cydowanie niżej, agent bardzo często wybierał by podskoczyć zamiast dać czas obiektowi ptaka na zmniejszenie swojej wysokości co oczywiście sprawiało, że przegrywał tę grę. Nie

mogłem znaleźć przyczyny takiego zachowania dlatego postanowiłem lekko zmodyfikować algorytm przedstawiony w listingu 3.2. Dodałem flagę, która jest włączana, gdy różnica w pionie między obiektem ptaka a dolną rurą była wystarczająco wysoka. Wtedy, jeżeli ostatnią akcją agenta było podskoczenie ($last\_action = 1$) i flaga była włączona, dla tego stanu przyznawałem nagrodę -1000. Po takim zmodyfikowaniu algorytmu otrzymałem następujące wyniki:

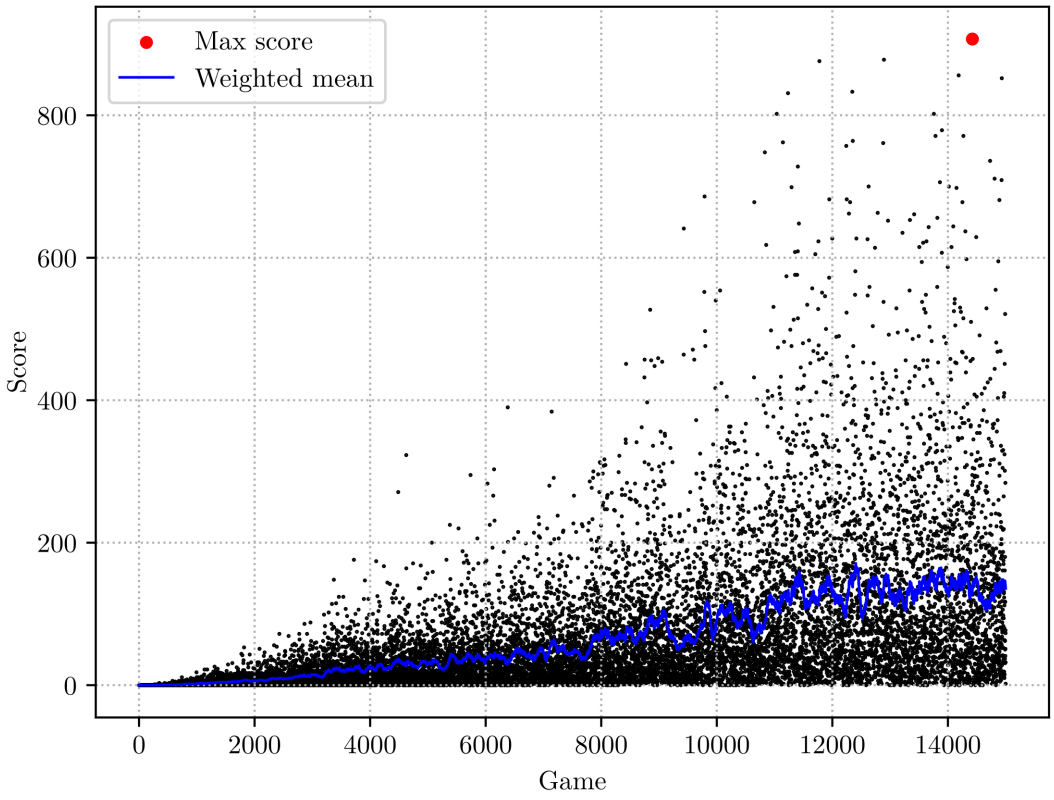


Rysunek 4.3: Wykres gry od wyniku po zmianie algorytmu.

Agent uczył się tutaj od początku (wszystkie wartości Q zostały wyzerowane dla każdego stanu przed uruchomieniem programu). Jak widać na wykresie 4.3 chociaż maksymalny wynik jaki udało mu się osiągnąć jest mniejszy niż w poprzedniej grze – jest to 933 uzyskanych punktów w grze 4652 czyli aż o 70 punktów mniej niż poprzednio to średni wynik po 5000 rozegranych gier wynosi już $119,36$, czyli o 72.7% więcej. Ponadto w porównaniu do poprzedniej nauki agent po około 3500 grach zdecydowanie częściej osiąga wyniki powyżej 200 punktów, w porównaniu do poprzedniego podejścia, gdzie nawet po tylu grach zdarzało się, że agent zdobywał niskie wyniki, właśnie ze względu na zachowanie przedstawione na rysunku 4.2.

4.3 Redukcja optymalizacji w przestrzeni stanów

Aby jeszcze zwiększyć średni wynik nauczonego agenta postanowiłem zmniejszyć skalę z jaką zoptymalizowałem rozmiar przestrzeni stanów. Co prawda taki zabieg wydłuży czas uczenia (więcej stanów oznacza, że agent będzie musiał zagrać więcej gier, aby obliczyć wartości Q dla każdego z nich), ale zwiększy precyzje z jaką agent podejmuje decyzje.



Rysunek 4.4: Wykres gry od wyniku po zwiększeniu przestrzeni stanów.

Postanowiłem zwiększyć przestrzeń stanów czterokrotnie – w tym podejściu stany generuję z dokładnością do wielokrotności liczby 5 to znaczy wartości X i Y przedstawione na listingu 3.1 generowane są co 5 a nie co 10. Analogicznie wartości obserwowane przez agenta ze środowiska są pomniejszane o resztę z dzielenia przez 5. W tym przypadku zbiór stanów ma 493240 elementów, co dalej stanowi $\frac{1}{25}$ ilości elementów zbioru stanów, gdzie żadnej optymalizacji nie zastosowano. W związku z takim powiększeniem przestrzeni stanów zwiększyłem ilość gier, które rozegrał agent (ponownie, agent uczył się od nowa, po wyzerowaniu wartości Q). Początkowo chciałem by było to 4 razy więcej gier niż poprzednio, jednak okazało się, że agent po około 8000 gier bardzo często zdobywał wyniki powyżej 200 co sprawiało, że rozgrywane gry trwały zdecydowanie dłużej. W tym

układzie rozegranie 15000 gier zajęło około 36 godzin przy najszybszym trybie gry czyli 120 klatkach na sekundę. Ponownie agentowi nie udało się pobić największego wyniku z pierwszego podejścia, uzyskał on maksymalnie 907 punktów, w grze 14424, co mimo wszystko jest lepszym wynikiem od poprzedniego podejścia. Średnia w tym podejściu pod koniec była najlepsza i wynosiła 135.6, czyli o 13.6% więcej niż poprzednio.

Podsumowanie

Przedstawiona w niniejszej pracy problematyka nie wyczerpuje w całości zagadnienia uczenia przez wzmacnianie, jednak szczegółowo opisuje działanie algorytmu Q-Learning. Samodzielne zaimplementowanie gry "Flappy Bird" pomogło mi w zrozumieniu i opisaniu jak agent działa w środowisku. Uważam, że wybrane przeze mnie technologie były trafionym pomysłem, były łatwe w użyciu i pozwoliły mi na całkowite zrealizowanie mojej pracy.

Największą komplikacją okazał się sam czas uczenia, mimo, że specjalnie dla agenta wprowadziłem możliwość uruchomienia szybszej rozgrywki, to przejście przez agenta 20000 gier okazało się zabierać zdecydowanie za dużo czasu. Osiągane przez agenta wyniki mimo moich optymalizacji nie są też zbyt wysokie w porównaniu do innych rozwiązań, jak na przykład Tonego Xu, któremu udało się tak zoptymalizować algorytm, by agent w ogóle nie mógł przegrać[20]. Myślę, że mógłbym bardziej optymalizować swój algorytm. Potrzebny byłby nakład pracy przyszłej w formie modyfikacji programu by ten nie przedstawiał graficznej wizualizacji gry (wszak agent potrzebuje tylko informacji o położeniu najbliższego obiektu rury, nie potrzeba wizualizacji). Wtedy mógłbym zaimplementować tak główną pętlę programową by ta nie wykonywała się raz na tyknięcie symulowanego zegara, a tak szybko jak pozwalałby na to procesor, na którym program został uruchomiony.

Niniejsza praca pokazuje jak skomplikowanym i fascynującym zagadnieniem może być uczenie przez wzmacnianie. Od samego wymyślenia algorytmu do jego optymalizacji wymaga nie tylko ogromnej wiedzy z zakresu prawdopodobieństwa, ale i kreatywnego myślenia.

Podsumowując, chociaż osiągane przez agenta wyniki nie są najlepsze (w porównaniu do innych tego typu rozwiązań, nie do graczy będących ludźmi) mimo wszystko stosując algorytm uczenia przez wzmacnianie udało mi się wytrenować agenta do grania w grę, nie mniej jednak zalecany byłby większy nakład pracy by osiągać naprawdę dobre wyniki.

Bibliografia

[1] Mitchell T 1997 *Machine Learning* (McGraw Hill) ISBN 0070428077

[2] Dabérius K, Granat E and Karlsson P 2019 (dostęp 06.2022) URL http://dx.doi.org/10.2139/ssrn.3374766

[3] Yuxi L 2019 (dostęp 06.2022) URL https://doi.org/10.48550/arXiv.1908.06973

[4] Strona internetowa dotgears (dostęp 06.2022) URL https://dotgears.com/games/flappybird

[5] Strona internetowa sdl (dostęp 06.2022) URL https://www.libsdl.org/

[6] Strona about projektu pygame (dostęp 06.2022) URL https://www.pygame.org/wiki/about

[7] *Dokumentacja klasy pygame.sprite.Sprite* (dostęp 06.2022) URL https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.Sprite

[8] *Dokumentacja metody pygame.sprite.Group.Draw* (dostęp 06.2022) URL https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.Group.draw

[9] *Dokumentacja metody pygame.groupcollide* (dostęp 06.2022) URL https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.groupcollide

[10] *Dokumentacja modułu pygame.event* (dostęp 06.2022) URL https://www.pygame.org/docs/ref/event.html

[11] Poole D and Mackworth A 2017 *Artificial Intelligence Foundations of Computational Agents* (Cambridge University Press) chap What is Artificial Intelligence? 2nd ed ISBN 9781107195394

[12] Poole D and Mackworth A 2017 *Artificial Intelligence Foundations of Computational Agents* (Cambridge University Press) chap Agents Situated in Environments 2nd ed ISBN 9781107195394

[13] Poole D and Mackworth A 2017 *Artificial Intelligence Foundations of Computational Agents* (Cambridge University Press) chap Belief Networks 2nd ed ISBN 9781107195394

[14] Poole D and Mackworth A 2017 *Artificial Intelligence Foundations of Computational Agents* (Cambridge University Press) chap Markov Chains 2nd ed ISBN 9781107195394

[15] Poole D and Mackworth A 2017 *Artificial Intelligence Foundations of Computational Agents* (Cambridge University Press) chap Decision Networks 2nd ed ISBN 9781107195394

[16] Sawka K 2020 *Uczenie maszynowe z użyciem Scikit-Learn i TensorFlow* 2nd ed (Helion) ISBN 9788328360020

[17] Poole D and Mackworth A 2017 *Artificial Intelligence Foundations of Computational Agents* (Cambridge University Press) chap Policies 2nd ed ISBN 9781107195394

[18] Watkins C J C H 1989 *Learning from Delayed Rewards* Ph.D. thesis University of Cambridge

[19] Poole D and Mackworth A 2017 *Artificial Intelligence Foundations of Computational Agents* (Cambridge University Press) chap Q-Learning 2nd ed ISBN 9781107195394

[20] Xu T 2020 *Towards Data Science* (dostęp 06.2022) URL https://towardsdatascience.com/use-reinforcement-learning-to-train-a-flappy-bird-never-to-die-35b9625aaecc/