Uniwersytet Mikołaja Kopernika
Wydział Matematyki i Informatyki

Tobiasz Pokorniecki
nr albumu: 291941

Praca inżynierska
na kierunku informatyk

Teoria i praktyczne zastosowanie uczenia maszynowego w grze typu Pac-Man

Opiekun pracy dyplomowej:
dr Piotr Przymus
Wydział Matematyki i Informatyki
Uniwersytet Mikołaja Kopernika

Toruń, 2021

Przymuję pracę i akceptuję

.....
data i podpis opiekuna pracy

Potwierdzam złożenie pracy dyplomowej

.....
data i podpis pracownika dziekanatu

Spis treści

1 Wprowadzenie

1.1 Wstęp

Sztuczna inteligencja, będąca częścią nieuniknionego postępu, zwykła nas fascynować jak i przerażać myślą, jak będzie ona wyglądać w przyszłości. Istnieje wiele tekstów popkultury opisujących to, jakoby sztuczna inteligencja miała przejąć kontrolę nad otaczającym nas światem, lub stać się zastępującą ludzkość super rasą. W tej pracy spróbuję przyczynić się do zanegowania tego scenariusza, przybliżając szczegółowo zagadnienie sztucznej inteligencji w oparciu o sieci neuronowe oraz tworząc przykładowy model, który nauczy się i zademonstruje swoje możliwości w środowisku opartym na grze w stylu popularnego Pac-Man'a.

Pierwsze zagadnienia nakreślające konwencję "sztucznej" myśli rodziły się w latach od 1945 do 1950 roku, kiedy swoje publikacje ukazywał między innymi Alan Turing, który jest uważany za jednego z pionierów dziedziny informatyki. Turing poruszał temat maszyn naśladujących schemat ludzkich zachowań, a nawet kształcących umiejętność rozumowania, która może być wykorzystywana przez maszynę do twórczej myśli podczas, chociażby, gry w szachy. Teoria ta zmusiła ludzkość do skrupulatnego zdefiniowania pojęcia myśli i narodziła nową ideę rózróżniającą czy rozumowanie w kontekście maszyn jest w ogóle możliwe. [1]

Konsekwencją rozmyślań dotyczących sztucznej inteligencji na przełomie XX i XXI wieku jest powstanie algorytmów uczenia maszynowego różnych typów. To, jaki typ algorytmu będzie najbardziej optymalny zależy od struktury postawionego zadania, oraz jaki jest wymóg poziomu dokładności wyników w rozsądnym, możliwie jak najmniejszym czasie. Równolegle rozwijającą się gałęzią tematyki sztucznej inteligencji, będącą wyróżnioną częścią algorytmów uczenia maszynowego, są sztuczne sieci neuronowe, będące uproszczonym modelem mózgu. Sieć neuronowa składa się z połączonych ze sobą równolegle neuronów zwanych perceptronami, które w procesie uczenia, wykorzystując jeden z algorytmów badających błąd (np. algorytm wstecznej propagacji), przetwarzają go aby wzmocnić lub osłabić połączenia między perceptronami warunkując przy tym jaki wynik otrzymamy na "wyjściu" sieci.

Wyniki opisywanego w poprzednim akapicie postępu technicznego są już powszechnie wykorzystywane. Algorytmy uczenia maszynowego, w tym sieci neuronowe, jesteśmy w stanie znaleźć u podstaw ówczesnych rozwiązań w najróżniejszych dziedzinach, rozpoczynając od rozpoznawania twarzy lub przedmiotów na zdjęciach, przez przewidywanie wahań na giełdach, wykrywanie oszustw w bankowości i kończąc na bardziej hobbystycznym wykorzystaniu, czyli analizie danych dostarczanych przez grę w celu jak najsprawniejszego jej przejścia, którego aspekty przybliżę w tej pracy.

1.2 Cel i zakres pracy

Głównym celem mojej pracy jest skonstruowanie sprawnie działającego modelu sieci neuronowej z klasy sieci splotowych, przechodzących stworzoną przeze mnie na potrzeby tej pracy grę arcade w stylu Pac-Man. Poruszę również tematykę algorytmów uczenia maszynowego i dokładniej sprecyzuję obszar uczenia przez wzmacnianie opisując zasady jego działania oraz związaną z nim terminologię. Przybliżę temat sieci neuronowych i algorytmu uczenia przez wzmacnianie - Q Learning - poddając oba rozwiązania próbie na stworzonym środowisku i porównam je pod względem zapotrzebowania na zasoby i szybkości osiągania szczytowej formy. Dodatkowo uwzględnię:

- Działanie i krótki opis implementacji środowiska Pac-Man stworzonego przy pomocy biblioteki pygame.
- Miejscowe różnice implementacyjne pomiędzy algorytmem Q-Learning a splotową siecią neuronową.
- Eksperymenty związane z optymalizacją oraz testowaniem potencjału obu podejść.

2 Uczenie przez wzmacnianie

2.1 Wstęp

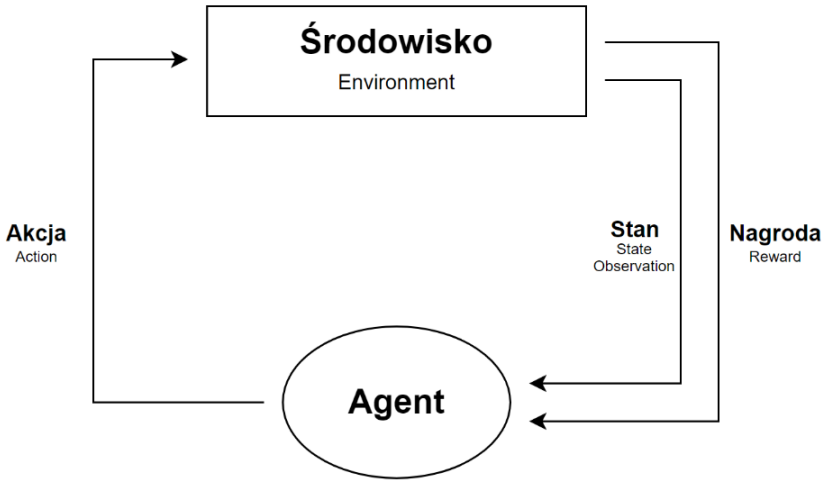
Największym atutem dziedziny uczenia maszynowego (inaczej ML) jest pełen wachlarz typów algorytmów dzięki którym przyczynia się do rozwiązywania problemów m.in. z kategorii klasyfikacji oraz decyzyjności. Problem określony jako klasyfikacyjny opiera się na trudności w stwierdzeniu do jakiej grupy przynależy brany pod uwagę obiekt; za przykład posłużyć może przeszukiwanie obrazu w celu przypisania go do konkretnej grupy obrazów w zależności, czy widnieje na nim pies, kot, czy inne zwierze. Najskuteczniejszymi w rozwiązywaniu problemów tego pokroju są algorytmy z gałęzi uczenia nadzorowanego i nienadzorowanego (chociaż w przypadku uczenia nienadzorowanego problemy tego typu określane są jako problemy klasteryzacji). Te pierwsze do poprawnego działania wymagają pierwotnego przeanalizowania we własnym zakresie licznych, skategoryzowanych już próbek, czyli swego rodzaju "nadzoru". Drugie natomiast, wykorzystując swój wewnętrzny potencjał, same znajdują prawidłowości we wprowadzanych danych i grupują je pod, być może, nieznanym jeszcze przez implementatora względem [2].

Problem występujący w sytuacjach, w których kluczowe jest podjęcie wyboru przez pewien podmiot zwany jest problemem decyzyjnym. Do poprawnego nakreślenia problemu decyzyjnego konieczne jest sprecyzowanie jego decydentów, ograniczników, puli dozwolonych wyborów, oraz zasad oceniania [3]. Dążąc do rozwiązania problemów typu "jaką drogę obrać w tym momencie aby odnieść jak największe korzyści?", który w sposób oczywisty wiąże się z definicją problemu decyzyjnego, z pewnością możemy polegać na rozwiązaniach uczenia przez wzmacnianie. Jest to dział uczenia maszynowego, który na podstawie stworzonych przez siebie struktur danych wielokrotnie analizuje problem, tym samym polepszając uzyskiwane wyniki w każdym kolejnym podejściu. Jest on więc idealnym narzędziem do komponowania ciągów przyczynowo-skutkowych prowadzących do celu. Warto wspomnieć, że wszystkie podane powyżej pojęcia związane z problemem decyzyjnym przekładają się niemalże jeden do jednego z pojeciami uczenia przez wzmacnianie. Z racji, że fundamentem każdej rozgrywki jest właśnie taki ciąg, nie będzie zaskoczeniem, że w obecnym czasie będzie to najlepsze podejście do zadeklarowanego przeze mnie celu pracy.

2.2 Opis i terminologia

Podstawą uczenia przez wzmacnianie (inaczej RL), odróżniającą ją od innych typów uczenia maszynowego jest system kar i nagród, który koryguje to, jakie akcje są pożądane w konkretnym czasie i konfiguracji środowiska [4]. Terminologia RL rozbija się o takie pojęcia jak:

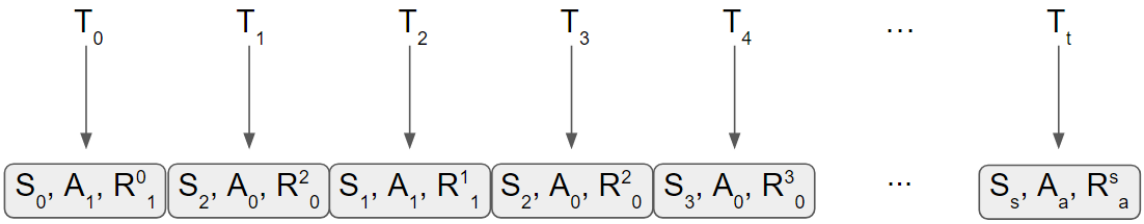
- Środowisko - przestrzeń posiadająca swój obecny stan oraz udostępniająca skończony zbiór akcji możliwych do wykonania na niej. W zależności od podjętej akcji w danym stanie karze lub nagradza agenta.
- Agent - (decydent) jednostka podejmująca decyzję jaką akcję wykonać mając do dyspozycji obecny stan środowiska. Jego celem jest zmaksymalizowanie sumarycznej wartości nagród zaczynając w konkretnym stanie środowiska.
- Akcja - krok wybrany ze skończonego zbioru czynności udostępnionych przez otoczenie, który podjął agent w oparciu o postawiony przed nim stan środowiska.
- Stan - cechy i konfiguracja środowiska w danym czasie.
- Nagroda - werdykt środowiska, czy akcja wykonana na obecnym stanie jest pożądana (liczba rzeczywista).



Rysunek 1: pojęcia i wizualizacja działania RL (źródło własne, wzorowane na wizualizacji z [4])

2.3 Zasady działania

Głównym filarem uczenia przez wzmacnianie jest środowisko, zawierające swój stan oraz zasady, którymi się rządzi. Uruchomienie środowiska, dalej zwane epizodem, jesteśmy w stanie podzielić na dyskretną sekwencję momentów decyzyjnych T_t. Każdy z takich momentów posiada obecny ww. stan S_s środowiska na którym agent podejmuje jedną akcję (na podstawie wybranego algorytmu) A_a ze skończonego zbioru możliwych akcji. Zadaniem środowiska jest ocenienie, na ile wybór podjęty przez agenta jest korzystny dla pomyślnego ukończenia środowiska. Wykorzystamy do tego liczbę rzeczywistą nazywaną później nagrodą lub zmienną nagrody R_a^s, która będzie przyjmowała różne wartości w zależności od tego, jaka akcja została podjęta na jakim stanie środowiska.



Rysunek 2: wizualizacja przykładowego dyskretnego ciągu decyzyjnego (źródło własne)

Celem agenta jest znalezienie oraz jak najczęstsze stosowanie jak najoptymalniejszej (najlepiej nagradzanej przez środowisko) sekwencji akcji. Staje on więc przed problemem jak eksperymentować, wynajdywać oraz trzymać się ścieżek w środowisku które będą dla niego najbardziej korzystne, czyli w skrócie jak nauczyć się rozwiązywać problem przedstawiony przez środowisko.

3 Środowisko Pac-Man

3.1 Wstęp

Gry i oparte na nich środowiska mają ogromny potencjał w uczeniu maszynowym, jednak większość ich implementacji nie jest odpowiednio na nie przygotowana. Jednym z kluczowych zagadnień przy wyborze lub konstrukcji środowiska jest m.in. średnia szybkość wykonywania jego epizodu. Oczywistym jest, że przy chęci stworzenia odpowiednio nauczonej struktury danych chcemy zrobić to jak najszybciej. Średnia szybkość wykonania jednego epizodu w środowiskach opartych na grach jest zależna praktycznie od wszystkiego, nie będzie więc zaskoczeniem, że odstęp pomiędzy momentami decyzyjnymi może być większy lub mniejszy, ich głównym celem jest zapewnienie graczowi rozrywki. Z racji, że nasza struktura uczy się tylko podczas dokonywania decyzji, to rozciągając odstęp pomiędzy decyzjami rozciągamy też czas potrzebny na nauczenie struktury. Kolejnym zagadnieniem jest to jak będziemy pobierać informacje

ze środowiska, formować je w stany i zaopatrywać w nie nasz algorytm. Tutaj plusem jest fakt sporego potencjału algorytmów uczenia maszynowego w wynajdowanie wzorów we wszelkich, nielosowych danych w byle jakim formacie, pod warunkiem, że zdecydujemy się na wykorzystanie akceptującego je algorytmu. Przykładem algorytmu sprawnie działającego na surowych danych z gry (wartości zmiennych w grze lub nawet wykorzystywane przez nią komórki pamięci) jest m.in. powyżej opisywany algorytm Q-learning, a jeśli zależy nam na analizie obrazu z gry (ze względu np. na złożoność stanów środowiska) pomoże nam klasa sieci neuronowych zwanych sieciami splotowymi (convolutional neural network), które opiszę i wykorzystam w podobnym celu później w tej pracy.

Stawiając przed sobą oba te problemy zdecydowałem się, że na potrzeby tego projektu samodzielnie zaprojektuję i zaimplementuję grę, która chociaż w pewnym stopniu będzie imitować grę Pac-Man, ale z pewnymi zmianami pozwalającymi na sprawniejsze pobieranie danych z gry (gra sama w sobie jest obudowana w algorytm z poziomu którego mogę łatwo zaciągać potrzebne mi zmienne) i umożliwiających uruchamianie gry bez interfejsu, co pozwala na dużo szybsze jej wykonywanie i przez to szybsze uczenie struktur. Pac-Man jest grą zręcznościową stworzoną przez Namco w 1980 roku. Rozgrywka opiera się na podróżowaniu tytułowym Pac-Man'em po zamkniętym labiryncie, w czym graczowi przeszkadzają przeciwnicy przedstawiani jako duchy. Spotkanie z jednym z takich duchów jest równoznaczne z porażką, natomiast warunkiem zwycięstwa jest zebranie wszystkich rozsianych równomiernie po labiryncie punktów. Powtarzane wszędzie w tej pracy słowo klucz "typu" podkreśla pewne ograniczenia w mojej implementacji względem oryginalnej wersji gry. Tymi ograniczeniami są między innymi inteligencja przeciwników, która w oryginale zależna jest od koloru duszka oraz jest ona bardziej złożona (W mojej implementacji przeciwnicy nie reagują na gracza i poruszają się w sposób losowy) i brak ikonicznych już "wisienek", które były swojego rodzaju wzmocnieniem Pac-Man'a i po zebraniu na moment pozwalały obrócić zasady rozgrywki i pokonywać przeciwników podczas spotkania z nimi [5, 6].

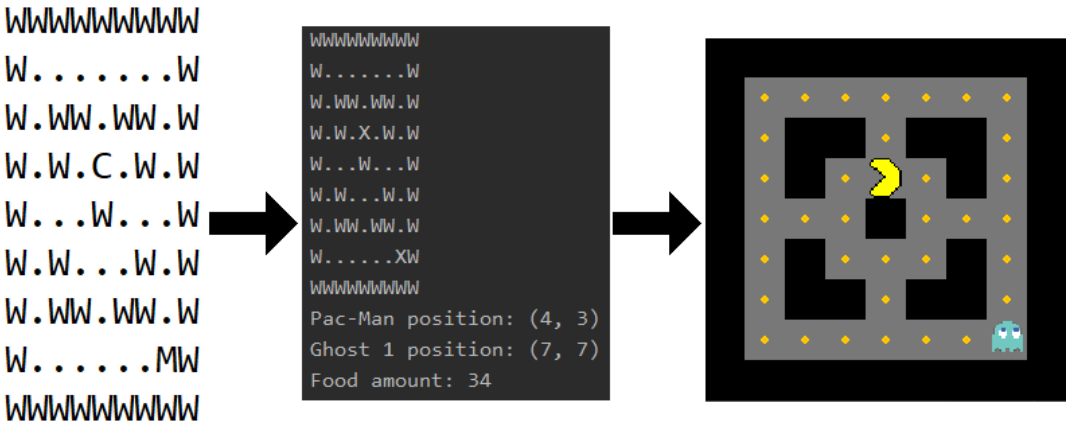
Oczywiście moja implementacja nie jest pierwszą tego typu. Warto również wspomnieć o istniejących już rozwiązaniach środowiskowych, które podobnie jak moja praca dostarczają pewne algorytmy uczenia maszynowego. `Gym` od OpenAI [7] to biblioteka do porównywania, wykorzystywania gotowych i samodzielnej implementacji algorytmów uczenia ze wzmocnieniem posiadająca szeroką bazę gotowych już środowisk oraz algorytmów. Jeden z projektów, zależny między innymi od biblioteki `Gym`, to propozycja użytkownika `github.com - NeymarL` skupiająca się na porównaniu wyników kilku algorytmów uczenia maszynowego w środowisku MS Pac-Man dostarczanym przez bibliotekę `Gym` [8]. Wyniki zaprezentowanie przez tego autora, niestety, są ciężkie do odniesienia się w celu porównania ich z moimi ze względu na różnice mechaniczne gry oraz zwracane przez środowisko statystyki (środowiska biblioteki `Gym` zwracają statystyki struktury ogólnikowej). Mój projekt nie opiera się chociażby na przedstawionej powyżej bibliotece ze względu na osobistą chęć wnikliwego poznania mechanik gry oraz wewnętrzną potrzebę pełnej kontroli środowiska. Mam tu na myśli również manipulację struktury planszy. Natomiast zwracane statystyki są spreparowane tak, by były jak najprostsze do zrozumienia i jak najlepiej ukazywały proces uczenia w zadeklarowanym przeze mnie środowisku.

3.2 Specyfikacja mojego środowiska

Moja wersja środowiska na podstawie gry Pac-Man, zaimplementowana specjalnie na potrzeby wykorzystania we wszelkich algorytmach uczenia maszynowego, została napisania w języku Python przy wykorzystaniu biblioteki PyGame [9]. Ta bardzo prosta biblioteka, przygotowana typowo pod tworzenie gier, umożliwia stworzenie podstawowego okna oraz pełną kontrolę nad wyświetlaniem w nim kolejnych klatek gry z wykorzystaniem klas reprezentujących wszelkie kształty, kolory, tekst oraz obrazy wczytane z dysku. Właśnie w takim okienku przedstawiana jest wygenerowana z pliku tekstowego plansza poziomu, w której skład wchodzą ściany, punkty, puste pola, duchy i Pac-Man. Generowanie planszy z pliku tekstowego jest swojego rodzaju translacją kolejnych znaków z pliku, najpierw na reprezentację w programie (tj. tablicę przedstawiającą planszę oraz zmienne zawierające pozycje wszelkich kreatur), a następnie tę reprezentację na odpowiadające im obiekty w oknie. Każdy obiekt ma swój własny wygląd oraz w przypadku kreatur - animacje, wzorowane na oryginale. Słownik wygląda następująco:

- W - ściana
- . - punkt
- X - puste pole
- C - pole startowe Pac-Man'a
- M - pole startowe ducha

Dodatkowo, dużym atutem mojej implementacji jest możliwość uruchomienia środowiska bez translacji z reprezentacji w programie na okno PyGame, co jest w stanie bardzo przyspieszyć wykonywanie środowiska w kolejnych epizodach, tym samym przyspieszając znacznie proces uczenia.



Rysunek 3: translacja przykładowej planszy (źródło własne)

Opisując środowisko konieczne jest przedstawienie jakie stany posiada, jakie warunki muszą być spełnione do jego ukończenia oraz jakie akcje agent jest w stanie na nim podjąć i jak są one nagradzane. Tak jak było to opisane powyżej, z racji, że środowisko jest moim autorskim tworem, mam możliwość komponowania stanów środowiska w jakichkolwiek formach chcę przy wykorzystaniu drugiego, opisanego powyżej, kroku translacji. W samym kodzie projektu translacja z postaci pierwszej do drugiej odbywa się przy wykorzystaniu funkcji `generate_board` przyjmującej ścieżkę do pliku tekstowego. Pobiera ona znaki z pliku pod daną ścieżką i wylicza odpowiednie wartości zmiennych by ostatecznie zwrócić obiekt klasy `Board` zawierający wszelkie informacje o początkowej konfiguracji gry w której skład wchodzą: tablica, której kluczami są współrzędne na planszy, a wartościami obiekt występujący na tym polu (tutaj trzeba zauważyć, że pozycje kreatur podczas translacji są pobierane do osobnych zmiennych, także na ich miejsca wchodzą wartości X, czyli puste pola, patrz rysunek 3), rozmiar planszy zapisany w krotce postaci `(szerokość, wysokość)`, zmienna zawierająca liczbę punktów obecnych w tym momencie na planszy oraz kolejne krotki zawierające początkowe pozycje wszystkich kreatur na planszy postaci `(x, y)`. Obiekt klasy `Board` jest swego rodzaju bazą danych, którą będziemy inicjalizować środowisko i analizować dane na różne sposoby by zasilać nimi wykorzystywany algorytm uczenia maszynowego oraz wypełniać za jego pomocą okno zawierające graficzną reprezentację środowiska. Będzie on tworzony na samym początku uruchomienia środowiska przy pomocy wyżej opisanej translacji, a następnie będzie tworzona jego kopia, by dalej na kopii śledzić obecność punktów oraz ich liczbę. Pracowanie na kopii pozwala nam na szybsze odtworzenie początkowego stanu planszy (tj. początkowe umiejscownienie kreatur i punktów), bez konieczności ponowego wykorzystywania funkcji generującej w kolejnych epizodach.

Implementacja każdej kreatury na planszy, w tym agenta reprezentowanego przez Pac-Man'a, opiera się na klasie `Creature`. Koniecznymi argumentami podczas inicjalizacji obiektu tej klasy są:

- `species` - łancuch znaków (`String`) zawierający wartość z dziedziny "pacman" lub "ghost", konieczny ze względu na rozróżnienie mechanizmów obsługujących animacje reprezentacji graficznej tych gatunków,
- `board`, `board_size`, `position` - wartości podane z utworzonego uprzednio obiektu klasy `Board` pozwalające kreaturze odnaleźć się na planszy. Algorytmy wyliczające możliwość wykonania zadanej akcji opierają się właśnie na tych wartościach,

posiada ona również jeden argument opcjonalny

- `visages` - będący zbiorem obrazów składających się na animację kreatury.

Klasa ta posiada również wszelkie metody potrzebne do wykonywania przez kreaturę akcji na środowisku oraz odpowiednie rysowanie jej animacji w oknie PyGame. Akcje możliwe do wykonania przez kreaturę to ruchy w 4 kierunkach, w górę, w dół, w lewo i w prawo. Dodatkowo zaimplementowany został system kolejkujący ruch. System ten jest próbą odwzorowania doświadczenia użytkownika występującego podczas grania w oryginał, przykładowo, gdy gracz wykonał akcję w pozycji której nie jest ona możliwa

do wykonania (ruch w ścianę), to akcja ta jest kolejkowana i zostanie wykonana przy najbliższej możliwej okazji, w pozycji która pozwala na taki ruch. W kolejce tej mieści się tylko jedna akcja i każda następna nadpisuje poprzednią. System wykorzystywany podczas poruszania się delikatnie różni się pomiędzy typami kreatur (Pac-Man i duch), wspólną zasadą poruszania się jest oczywisty brak możliwości ruchu skierowanego w ścianę (wtedy następuje kolejkowanie ruchu), jednak w przypadku duchów poruszanie się ma pewne dodatkowe ograniczenia (wzorowane na oryginale) wyliczane przez metodę `get_possible_actions`, mianowicie nie pozwala ona duchom na zawracanie na skrzyżowaniach i zakrętach, duch jest w stanie zawrócić tylko wtedy, gdy jest to jego jedyna opcja. Wszystkie metody związane z poruszaniem się oraz jego ograniczeniami i odpowiednim wyświetlaniem kreatury (o ile wykorzystujemy krok translacji tworzący graficzną reprezentację) są wyliczane po wykonaniu metody `do_action` na obiekcie klasy `Creature`, argument który ona przyjmuje to indeks od 0 do 3 zmapowanej akcji, odpowiednio reprezentujący ruch: w górę, prawo, dół i lewo.

Każde ukończenie środowiska, zwane epizodem, jest w większości algorytmów uczenia maszynowego koniecznym elementem do poprawnego przeprowadzenia procesu uczenia. To jak nasz agent stosunkuje się do kolejnych stanów środowiska jest zależne od tego, jak jest nagradzany i co za tym idzie, z jakim wynikiem ukończył środowisko. W moim przypadku wartość nagrody jest ściśle związana z ruchem Pac-Man'a, a konkretniej, czy po wykonaniu ruchu Pac-Man zebrał punkt, czy był to ruch "pusty", oraz, co jest oczywiste, czy Pac-Man ukończył środowisko pomyślnie, zbierając wszystkie punkty (wartość zmiennej `food_amount` w obiekcie `Board` spadła do 0), czy odniósł porażkę dając schwytać się przez jednego z przeciwników (pozycja Pac-Man'a zgrywa się z pozycją jednego z przeciwników).

Samo uruchomienie mojego środowiska rozkłada się na kilka czynników. Najważniejszym czynnikiem jest klasa `Gameplay`. Klasa ta jest spoiwem pomiędzy wszystkimi aspektami wykorzystywanymi podczas uczenia i nie tylko. Jest ona między innymi odpowiedzialna za synchronizację akcji wykonywanych przez kreatury z reprezentacją graficzną i logiczną układu planszy. Na samej górze posiada zadeklarowaną konfigurację parametrów, różną w zależności od wykorzystywanego algorytmu uczenia maszynowego, jednak wspólnymi parametrami dla większości algorytmów tego typu są m.in.:

- `episodes` - ilość epizodów wykonanych na jedno uruchomienie środowiska,
- `render_per` - ilość epizodów pomiędzy którymi wyświetlana ma być graficzna reprezentacja środowiska,
- `death_penalty, win_reward, point_reward, move_penalty` - zmienne definiujące wartości nagrody, ich nazwa wskazuje na zdarzenie, którego dotyczą,
- `action_size` - stała zawierająca ilość akcji możliwych do wykonania, będzie ona niezmienna podczas działania całego programu oraz każdej jego konfiguracji, dlatego nie będę się do niej odnosił nigdzie w dalszej części pracy (zawsze będzie miała wartość 4).

Operacja inicjalizująca obiekt klasy `Gameplay` przygotowuje środowisko do uruchomienia odpowiednio wyliczając swoją konfigurację na podstawie podanych argumentów. Te argumenty to:

- `model` - zmienna zawierająca ścieżkę do pliku z zadeklarowaną już strukturą uczącą wykorzystywaną w algorytmie, za pomocą której chcemy zainicjalizować środowisko, jeśli None to wygenerowanie zostanie nowa struktura (domyślnie None),
- `board_file` - zmienna ze ścieżką do pliku tekstowego zawierającego tekstową reprezentację planszy (domyślnie "pacman_board.txt").
- `ALGORITHM_USED` - zmienna typu `String` zawierająca wartość z dziedziny {"Q", "DQN"} będąca wskazówką dla programu na jakim rodzaju algorytmu uczenia maszynowego ma się opierać. Pomiedzy tymi opcjami wykorzystywane są zupełnie inne techniki oraz inne struktury uczące, które będą opisane skrupulatnie w dalszej części pracy (wartości tego argumentu są precyzyjnie nakreślone, każda inna wartość lub nie podanie tego argumentu poskutkuje błędem uruchomieniowym środowiska).

Termin przygotowania środowiska do uruchomienia obejmuje takie czynniki jak odpowiednie zadeklarowanie zmiennych potrzebnych do wyświetlania reprezentacji graficznej (`display`, `clock` oraz `pacman_visages` i `ghost_visages` zawierające załadowane z dysku obrazy będące częściami animacji kreatur reprezentujących Pac-Man'a oraz ducha), utworzenie początkowego układu planszy przy pomocy opisywanej powyżej funkcji `generate_board`, skopiowanie zadanej w argumencie `model` struktury, bądź w przypadku jej braku, utworzenie jej.

Najważniejszą metodą klasy `Gameplay` jest metoda `playthrough`, która rozpoczyna, obsługuje, oraz zwraca wartości wynikowe pojedynczego epizodu środowiska (tzn. czy epizod został zakończony powodzeniem czy porażką, oraz z ilu momentów decyzyjnych się składał). Jej jedynym argumentem jest wartość boolowska definiująca czy dany epizod ma być wyświetlany (jego reprezentacja graficzna) czy nie. Samą metodę możemy podzielić na kilka kluczowych części. Pierwszą z nich jest przygotowanie zmiennej `board` zawierającej obecny układ planszy dokonując kopii specjalnie utworzonej na tę okazję zmiennej `starting_board` zawierającej układ początkowy. W skład tej części wchodzi również operacja inicjalizacji kreatur (obiektów klasy `Creature`) zaopatrująca każdą kreaturę w informacje na temat układu planszy, jej pozycji na niej oraz, pod warunkiem że obecne uruchomienie ma posiadać reprezentację graficzną, we wcześniej przygotowane klatki animacji podając je jako ostatni argument - `visages`. Na końcu tej części zerujemy ilość momentów decyzyjnych w epizodzie (zmienna `rounds`), deklarujemy zmienną `win` mówiącą o wyniku epizodu. Następnie zakładamy, że epizod nie został ukończony (`done = False`) i przechodzimy do kolejnej części metody.

Część tą możemy w pewnym stopniu opisać jako serce środowiska, gdyż to tutaj będziemy precyzować działanie wybranego algorytmu i wizualizować zwracane przez niego wyniki poprzez reprezentację graficzną. Część ta, będąca w całości pętlą zależną od wartości boolowskiej zmiennej `done`, odwzorowywuje zasadę działania uczenia przez wzmacnianie (patrz rysunek 2), gdzie każda iteracja odpowiada jednemu momentowi decyzyjnemu. Pojedyncza iteracja tej pętli składa się z następujących kolejno operacji.

1. Pobranie stanu środowiska (możliwie wartości pobrane prosto z obiektów klas `Board` i `Creature` lub w jakiś sposób przetworzone). Analiza tego, jaki wybór wartości daje najlepsze wyniki, będzie opisana w dalszych częściach pracy odwołujących się do konkretnych algorytmów.

2. Wybranie akcji Pac-Man'a. Tutaj sposób w jaki podejmiemy akcję jest w zupełności zależny od nas, możemy nawet dopisać kod obsługujący przyciski klawiatury i sterować Pac-Man'em samemu, lecz w tej pracy ograniczę się do podejmowania decyzji poprzez algorytmy uczenia maszynowego.

3. Pętla po wszystkich przeciwnikach, w tym początkowo sprawdzenie czy pozycja Pac-Man'a po wykonaniu akcji zgrywa się z którymś z duchów, w takim przypadku przypisujemy odpowiednią wartość nagrody i będziemy zbiegać w programie w kierunku końca epizodu (ustawiając wartość zmiennej `done` na `True`), następnie wyliczamy ruch ducha przy pomocy opisywanej wcześniej funkcji `get_possible_actions` i na końcu jeszcze raz sprawdzamy, czy pozycje po ruchu ducha nie zgrywają się z pozycją Pac-Man'a z tą samą możliwą konsekwencją.

4. Etap stworzony wyłącznie dla epizodów uruchamianych z reprezentacją graficzną (argument funkcji `playthrough - render` = `True`) na potrzeby wyświetlania obecnego układu planszy oraz animowania zmian dokonanych po wykonaniu akcji przez kreatury.

5. Obsługa zdarzeń związanych z zebraniem przez Pac-Man'a punktów. Jeśli Pac-Man po wykonaniu powyżej zadeklarowanej akcji znajduje się na polu z wartością "." (punkt) to wartość w tym polu układu planszy zmieniana jest na "X" (puste pole) i obliczana jest zmienna `food_amount` w obiekcie `board` i modyfikowana wartość nagrody (o ile wcześniej nie została ona ustalona przez np. zdarzenie kolizji Pac-Man'a z duchem). Sprawdzane jest również, czy zmienna `food_amount` osiągnęła wartość 0, co oznacza, że środowisko zostało pomyślnie ukończone.

6. Ostatnią częścią tej pętli są operacje związane z nauczaniem wybranego algorytmu (pobieranie nowego stanu i wyliczanie nowych wartości do struktury uczącej) i zamknięcie okna reprezentacji graficznej, jeśli takowa była wykorzystywana.

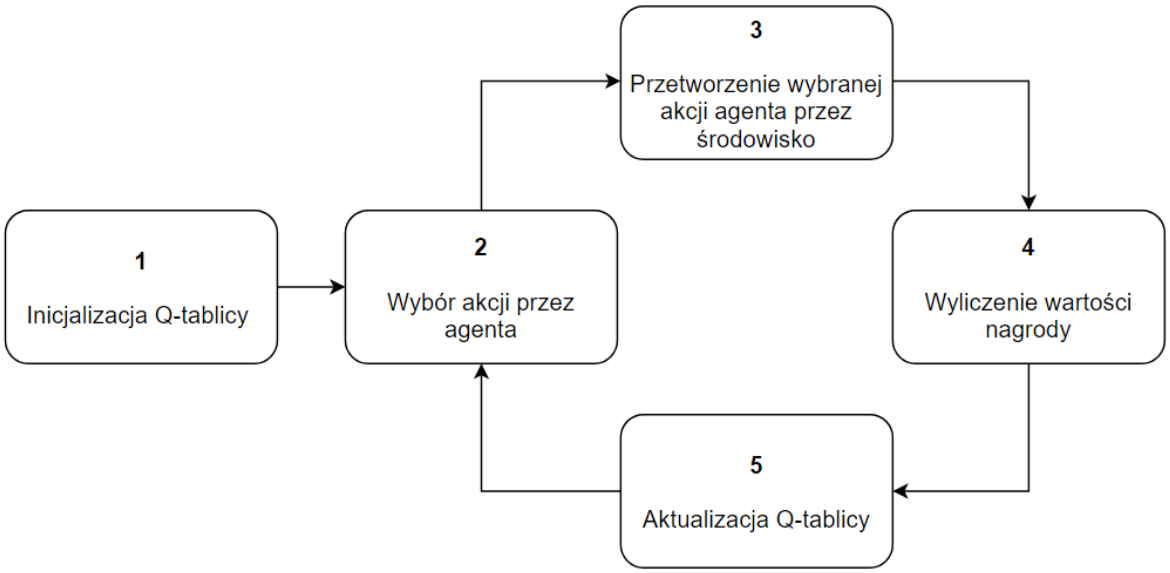
Ostatnią kluczową metodą klasy `Gameplay` jest metoda `start` będąca pomniejszym interfejsem pomiędzy użytkownikiem, a środowiskiem. Jest ona rozrusznikiem całego procesu uczenia, przeprowadza ten proces poprzez kolejne epizody środowiska, wyświetla średnią ocenę wyników algorytmu co ilość epizodów zadaną w parametrze klasy `Gameplay` - `render_per` oraz - o ile ma to sens (algorytm zwraca jakkolwiek porządane wyniki) - zapisuje strukturę uczącą w formacie zależnym od wykorzystywanego algorytmu. Przymuje jeden argument zwany `render`, który podczas działania przekazywany jest dalej do metody `playthrough` którą wywołuje w pętli oraz obsługuje zwracane przez nią wyniki.

4 Algorytm Q-learning

4.1 Teoria

Algorytm Q-learning opiera się na strukturze danych zwanej tablicą Q-wartości. Tablica Q-wartości jest co najmniej 2-wymiarową (wymiar zależy od ilości składowych stanu środowiska) tablicą liczb rzeczywistych. Będzie ona obrazować wszystkie możliwe kombinacje stanów środowiska i akcji możliwych do podjęcia oraz odpowiadającą tym kombinacjom liczbę rzeczywistą - Q-wartością. To właśnie ta struktura będzie swoistym mózgiem agenta który będziemy udoskonalać z każdym epizodem. Wyobrażeniem działania takiej tablicy może być zwierzęcy instynkt podpowiadający zwierzęciu co należy zrobić przy zadanych warunkach otoczenia.

Tak jak zaznaczałem podczas opisywania swojego środowiska, sercem algorytmów uczenia przez wzmacnianie jest pętla, której każda jedna iteracja odpowiada kolejnym momentom decyzyjnym w epizodzie. W przypadku Q-learningu pętla ta wygląda następująco (rysunek 4). Dalej opiszę każdą składową pętli [10].



Rysunek 4: pętla algorytmu Q-learning (źródło własne)

1. Inicjalizacja Q-tablicy jest wstępnym krokiem algorytmu. Odnosi się do wypełnienia owej tablicy liczbami rzeczywistymi. Wartości te mogą być losowane, najlepiej z zakresu komplementującego zbiór wartości kar i nagród lub wczytane z pamięci gdy posiadamy tablicę na której algorytym już działał.

2. Wybór akcji przez agenta odbywa się poprzez sprawdzenie która Q-wartość (odpowiadająca akcji) w tablicy w rzędzie obecnego stanu jest najwyższa i to właśnie ta akcja jest wyborem agenta.

3. Przetworzenie wybranej akcji agenta przez środowisko opiera się na sekwencji operacji w środowisku. Sekwencje te będą w oczywisty sposób różnić się w zależności od zadeklarowanego środowiska. Będzie to na przykład ruch agenta w którymś kierunku.

4. Wyliczenie wartości nagrody polega na ocenie jakości podjętej akcji. Odpowiednie wyznaczenie wartości nagrody podlega eksperymentom. Pytania które mogą pojawić się podczas doboru tych wartości to między innymi: Jak chcemy nakierowywać agenta na poprawne rozwiązania środowiska lub czy chcemy raczej skupić się na rozwiązaniach jak najbezpieczniejszych, bądź z jak najszybszym ukończeniem (np. karząc agenta ujemnymi wartościami nagrody przy wykonywaniu "pustych" ruchów).

5. Aktualizacja Q-tablicy nową wartością w polu podjętej akcji na danym stanie odbywa się przy pomocy wzoru

$$Q[a_t][s_t] = Q[a_t][s_t] + \alpha * (R_a^s + \gamma * maxQ(s_{t+1}) - Q[a_t][s_t]), \tag{1}$$

gdzie:

- Q - tablica Q-wartości. Odwołanie $Q[a_t][s_t]$ oznacza Q-wartość akcji o indeksie a_t na stanie o indeksie s_t (subindeks t oznacza numer momentu decyzyjnego),

- α - liczba rzeczywista z dziedziny od 0 do 1 zwana wskaźnikiem uczenia (learning rate). Definiuje do jakiego stopnia chcemy wykorzystywać nową informację. Wartość ta wykorzystywana jest w większości algorytmów uczenia maszynowego,

- R_a^s - wartość nagrody wyliczona przez środowisko, którą agent uzyskał za akcję o indeksie a na stanie o indeksie s,

- γ - liczba rzeczywista z dziedziny od 0 do 1 zwana wskaźnikiem dyskontowym (discount factor). Definiuje na ile cenimy przyszłe wybory agenta, które mogą doprowadzić do pozytywnej nagrody. W celu otrzymania długoterminowych benefitów należy ustawić ten parametr blisko maksymalnej wartości. Niższe sprawią, że agent będzie mniej ufny jeśli chodzi o nagrody i kary, a co za tym idzie bardziej ostrożny,

- $maxQ(s)$ - funkcja zwracająca maksymalną wartość z Q-tablicy w rzędzie/kolumnie odpowiadającej stanowi o indeksie s.

Wzór przedstawia zwiększenie Q-wartości w danym polu o pewną część wartości przyszłych akcji, które być może były już odpowiednio nagrodzone. Pozwoli nam to na zwiększenie częstotliwości podejmowania sekwencji akcji, których zsumowane wartości nagród dają satysfakcjonujący wynik, a co za tym idzie zwiększenie prawdopodobieństwa pomyślnego ukończenia środowiska w kolejnych epizodach, nawet przy pewnej losowości stanów.

4.2 Praktyka

Od strony użytkownika zmiana wykorzystywanego algorytmu w moim środowisku ogranicza się do zmiany argumentu `ALGORITHM_USED` podawanego podczas konstrukcji obiektu klasy `Gameplay`. W przypadku chęci wykorzystania algorytmu Q-learning parametr ten musi posiadać wartość "Q". W tym podrozdziale opiszę co tak naprawdę dzieje się w kodzie przy wyborze tego algorytmu.

Pierwsza różnica występuje już podczas inicjalizacji samego obiektu `gameplay`. Tak jak było opisywane we wcześniejszym podrozdziale, algorytm ten wykorzystuje strukturę uczącą zwaną tablicą Q-wartości, która jest tworzona oraz na samym początku wypełniana losowymi wartościami. Funkcja odpowiedzialna za tworzenie owej tablicy to metoda klasy `Gameplay` - `create_q_table` i jej ogólna, przykładowa struktura może wyglądać następująco:

```
1 def create_q_table(self):
2   size_x = self.starting_board.size[0]
3   size_y = self.starting_board.size[1]
4   q_table = {}
5   for x in range(0, size_x):
6     for y in range(0, size_y):
7       q_values = [np.random.uniform(-5, 0) for _ in range(self
            .action_size)]
8       q_table[((x, y))] = q_values
9   return q_table
```

Z definicji tablica ta powinna posiadać wszystkie kombinacje możliwych do osiągnięcia stanów oraz akcji. W tej implementacji tablica Q-wartości tak naprawdę nie jest tablicą, a Python'owym słownikiem (dict) zapisanym w zmiennej `q_table`. Zmienna `q_table` jest wypełniana kombinacjami poprzez zapętlenie operacji przypisania losowych Q-wartości (przypisanie występujące w 7 linijce wykorzystujące funkcję `np.random.uniform` oraz mechanizm odwzorowywania tablicy zwraca wartości postaci `[a, b, c, d]`, gdzie każdy parametr od `a` do `d` jest zmienną losową z rozkładu jednostajnego z zakresu od -5 do 0) do kolejnych kluczy będących identyfikatorami stanów. Ilość zapętleń oraz ich zakresy będą się zmieniać w przypadku wykorzystania różnych postaci pojedynczego stanu. W powyższej deklaracji funkcji `create_q_table` zapętlenia występują po dwóch zmiennch `x` i `y`, kolejno najpierw po współrzędnych

szerokości planszy, a potem po jej wysokości. Każdy wyprodukowany w tej sposób klucz jest wykorzystywany do przypisania do Q-tablicy losowych wartości ze zmiennej `q_values`. Tego typu Q-tablica, bedąca odwzorowaniem planszy, nie mówi wiele o obecnej sytuacji. Wnioskując po zakresach które obrazuje, najbardziej rozsądnie może być wykorzystany do przekazywania do algorytmu obecnej pozycji Pac-Man'a na planszy. Podążając jednak dalej za tym podstawowym przykładem, odwołując się do zmiennej `q_table` po przykładowym kluczu `(1, 2)` (w sposób `q_table[(1, 2)]`) otrzymujemy Q-wartości każdej akcji w stanie w którym pozycja Pac-Man'a w obecnym układzie planszy jest usytuowana w drugiej kolumnie i trzecim rzędzie planszy.

Bazując na moim dotychczasowym doświadczeniu, choć wynika to też z logiki, jestem w stanie stwierdzić, że zakodowanie w pojedynczym stanie samej wartości odzwierciedlającej pozycję Pac-Man'a na planszy nie jest dostatecznie dobrym zbiorem informacji do otrzymywania miarodajnych wyników z algorytmu. W powyższym przypadku agent tak na prawdę nie wie gdzie znajduje się jego przeciwnik, w wyniku czego ciężko będzie mu go unikać, nie potrafi też określić gdzie znajdują się jeszcze niezebrane punkty. Będą to więc następne informacje o które rozszerzymy nasz stan, a co za tym idzie, Q-tablicę. Kolejność deklarowania składowych nie ma znaczenia, ważne jest jednak by potem odnosić się do nich konsekwentnie, tak jak zostało to zadeklarowane. Chcąc rozwinąć stan o kolejne informacje należy skupić się na tym, jakiej są struktury oraz jakie wartości przyjmują. Chcąc wykorzystać w algorytmie składową stanu pokroju pozycji Pac-Man'a lub ducha, sprawa jest w miarę prosta. Z racji, że posiadamy dwuwymiarową planszę musimy po prostu zapętlić przypisanie Q-wartości po współrzędnych wszystkich pól na planszy. Zakresy pętli w takim przypadku będą równe od 0 do szerokości planszy po zmiennej x i od 0 do wysokosci planszy po zmiennej y. Pojedyncza składowa zawierająca pozycję na planszy będzie miała postać `(x, y)`. Składowa symbolizująca pozycję ducha będzie tworzona analogicznie, lecz nadal może być to zmienna odnosząca się do innej tematyki składowych, więc konieczne jest wykorzystanie innej nazwy dla zmiennej zapętlającej, w ukazanym poniżej kodzie będzie to `(xd, yd)`, litera 'd' oznacza tu odpowiednik zmiennej dla pozycji ducha. Kwestią wymagającą dłuższego zastanowienia jest informacja o pozycji punktów, gdyż nie jesteśmy w stanie podać pozycji każdego możliwego punktu bez tworzenia z Q-tablicy potężnego monolitu posiadającego 81^{38} kombinacji (już przy stosunkowo małej planszy 9 na 9 pól, zawierającej 36 punktów + pozycję Pac-Man'a i ducha). Moją propozycją jest podanie do algorytmu drogi do jednego, najbliższego Pac-Man'owi punktu. W tym celu zaimplementowałem funkcję `get_way_to_closest_food` wyszukującą najbliżej położony punkt przy pomocy twierdzenia Pitagorasa i zwracającą drogę do niego względem Pac-Man'a (przez drogę należy rozumieć różnicę współrzędnych x i y w położeniu Pac-Man'a i punktu, różnica ta zdefiniowana jest w metodzie `way_to(x, y)` klasy `Creature`). Zwracane wartości będą postaci `(xp, yp)` i każda z nich będzie zawierać się w przedziale podobnym do wartości określających pozycję, ale z rozszerzeniem o wartości ujemne (tę operację można określić jako rozszerzenie dziedziny tych zmiennych o ich lustrzane odbicie na osi), co koniecznie będzie trzeba uwzględnić przy definicji funkcji generującej Q-tablicę. Podsumowując, po przemyśleniu tego, jakiej struktury będzie nasz stan musimy odpowiednio zmodyfikować zapętlenia w funkcji `create_q_table`. Zapętlenia związane z budowaniem całej tablicy w oparciu o opisywaną powyżej strukturę stanu przedstawiają się tak jak następuje:

```
for x in range(0, size_x):
  for y in range(0, size_y):
    for xd in range(0, size_x):
      for yd in range(0, size_y):
        for xp in range(-size_x + 1, size_x):
          for yp in range(-size_y + 1, size_y):
            q_values = [np.random.uniform(-5, 0) for _ in
               range(self.action_size)]
            q_table[((x, y), (xd, yd), (xp, yp))] = q_values
```

Gdy posiadamy już odpowiednio określony stan, możemy przejść do przedstawienia tego, co dzieje się już w samej pętli algorytmu. W kodzie, tak jak było to sprecyzowane w dziale "Specyfikacja mojego środowiska", pętla ta zawiera się w metodzie `playthrough` klasy `Gameplay`. Przyjrzyjmy się dokładniej w jaki sposób zdeterminować w jakim stanie znajduje się gra oraz którą akcję, według algorytmu, Pac-Man powinien podjąć. Wpierw, by odnieść się do konkretnych Q-wartości stanu musimy stworzyć klucz stanu. Struktura klucza musi być ujęta konsekwentnie, tak jak uprzednio została zadeklarowana Q-tablica, by uniknąć wychodzenia poza jej zakresy, a co za tym idzie wywoływania wyjątków związanych ze strukturą Python'owego słownika (dict). Klucz budujemy więc tak, jak z góry został określony stan:

```
f = (self.pacman.x, self.pacman.y)
g = (self.ghosts[0].x, self.ghosts[0].y)
h = self.get_way_to_closest_food()
state = (f, g, h)
```

Zmienna `f` przyjmuje wartość pozycji Pac-Man'a. Analogicznie zmienna `g` przyjmuje wartość pozycji ducha (odwołanie `[0]` występuje z faktu możliwości obsługiwania przez środowisko więcej niż jednego ducha, w tym podejściu skupiamy się jednak na obecności jednego ducha). W linijce 3 zdefiniowana jest zmienna `h` przyjmująca wartość zwracaną przez metodę `get_way_to_closest_food`. Ostatecznie klucz zespalamy w całość tworząc zmienną `state`, przy pomocy której możemy dowiedzieć się co o danym stanie sądzi algorytm, a dokładniej Q-tablica. Odwołując się do niej poprzez `self.model[state]` otrzymujemy tablicę czterech Q-wartości [a, b, c, d] z których najpierw musimy wybrać tę największą, a następnie określić jej indeks w tej tablicy. Indeks ten będzie indeksem akcji "wybranej" przez algorytm, ktory następnie możemy bezpośrednio przekazać do agenta. Obie te operacje jesteśmy w stanie jednocześnie wykonać dzięki wykorzystaniu funkcji `argmax` z biblioteki `numpy`. Wybranie oraz wykonanie akcji przez agenta zawiera się więc w linijkach:

```
action = np.argmax(self.model[state])
self.pacman.do_action(action)
```

Ostatnim i najważniejszym wkładem algorytmu Q-learning w nasze środowisko jest zastosowanie jego operacji uczenia opierającej się na wykorzystaniu wzoru (1) opisanego w części teoretycznej tego działu. Odwołując się do tamtejszej fragmentaryzacji wzoru, przyporządkujmy sobie jego elementy na konkretne nazwy zmiennych w kodzie. Oprócz oczywistego odwołania do wartości z Q-tablicy na podstawie której podjęliśmy akcję w obecnym momencie decyzyjnym, posiadamy również odwołania do stałych α i γ, w kodzie są one polami klasy `Gameplay` o nazwach kolejno `learning_rate` i `discount_factor` oraz część R_a^s, w kodzie obrazowaną przez zmienną `reward` przypisywaną podczas wykonywania każdego pojedynczego momentu decyzyjnego (patrz zapętlenie `while not done`). Fragment we wzorze sprecyzowany jako funkcja `maxQ()` będzie wymagał szczególnej uwagi. Definicja tej funkcji mówi, że powinna ona zwrócić maksymalną wartość z Q-tablicy w rzędzie lub kolumnie (zależy od implementacji) odpowiadającej stanowi o danym indeksie, co oznacza, że potrzebujemy najwyższej Q-wartości w stanie, w którym znalazł się agent po wykonanej, wybranej przez algorytm akcji. Stwórzmy więc klucz do Q-wartości nowo zaistniałego stanu oraz wybierzmy z nich tę największą. Tym razem zamiast funkcji `argmax` zwracającej indeks najwyższej wartości pobierzemy samą wartość przy pomocy funkcji `max`.

```
1 current_q = self.model[state][action]
2 f = (self.pacman.x, self.pacman.y)
3 g = (self.ghosts[0].x, self.ghosts[0].y)
4 h = self.get_way_to_closest_food()
5 new_state = (f, g, h)
6 max_future_q = np.max(self.model[new_state])
```

Posiadając teraz zmienne `current_q` (Q-wartość z wykonanej akcji), `max_future_q` (Q-wartość obecnie najbardziej opłacalnej akcji) oraz resztę, możemy zmierzyć się z przedstawieniem wzoru Q-learning (1) w kodzie. Wyliczmy najpierw nową Q-wartość, a następie przypiszmy ją do miejsca kryjącego się za kluczem poprzedniego stanu.

```
1 new_q = current_q + self.learning_rate * (reward + (self.
    discount_factor * max_future_q) - current_q)
2 self.model[state][action] = new_q
```

4.3 Uruchomienie i ulepszenia

Przed przystąpieniem do uruchomienia należy mieć pewność co do zadeklarowanej konfiguracji w samej klasie `Gameplay`. Parametry wymagające szczególnej uwagi oraz wykorzystywane w nich wartości podczas testowego uruchomienia środowiska opierającego się na algorytmie Q-learning wyglądają następująco:

```
1  episodes = 10_000
2  render_per = 1_000
3
4  death_penalty = -300
5  win_reward = 600
6  point_reward = 20
7  move_penalty = 0
8
9  learning_rate = 0.1
10 discount_factor = 0.95
```

Taka konfiguracja oznacza, że będziemy uruchamiać 10000 epizodów, co każdy 1000 będzie zwracany uśredniony wynik dotychczasowej wydajności algorytmu w naszym środowisku oraz zależne od parametru `render` ukazanie przebiegu tego epizodu w formie reprezentacji graficznej. W linijkach 4-7 zadeklarowana jest "tabela" nagród za konkretne wydarzenia do których doprowadził agent. Dalej stałe wartości `learning_rate` oraz `discount_factor` których wkład w proces uczenia został opisany podczas przybliżenia teorii Q-learning przy okazji opisywania wzoru (1). Interpretowane dalej uruchomienie oraz zwracane przez nie wyniki opierają się na zadeklarowanej powyżej konfiguracji oraz Q-tablicy generowanej z zapętlenia (uzupełnienia funkcji `create_q_table`) pokazanego na stronie 16. Cała klasa `Gameplay` zaimplemetowana jest w taki sposób, by można było ją w łatwo uruchomić z poziomu własnego, Python'owego skryptu. Zobrazuję to przedstawiając mój skrypt, w projekcie inżynierskim nazwany `main.py`, który uruchamia moje środowisko w trybie algorytmu Q-learning.

```
1  from gameplay import Gameplay
2
3  ALGORITHM_USED = 'Q'
4  render = True
5  model = None
6
7  gameplay=Gameplay(model=model, ALGORITHM_USED=ALGORITHM_USED)
8
9  gameplay.start(render)
```

Przed wykonaniem naszego skryptu należy upewnić się co do poprawności ścieżki do pliku zawierającego tekstową wersję planszy. Błędna ścieżka może doprowadzić do wywołania wyjątku, a niepoprawne ustrukturyzowanie tekstu do błędów z pamięcią lub

wyświetlaniem reprezentacji graficznej. Dla tego wywołania, plik `pacman_board.txt` ma następującą strukturę:

```
1 W W W W W W W W W W
2 W . . . . . . . . W
3 W . W W . W W . . W
4 W . W . C . W . . W
5 W . . . W . . . . W
6 W . W . . . W . . W
7 W . W W . W W . . W
8 W . . . . . . M W W
9 W W W W W W W W W W
```

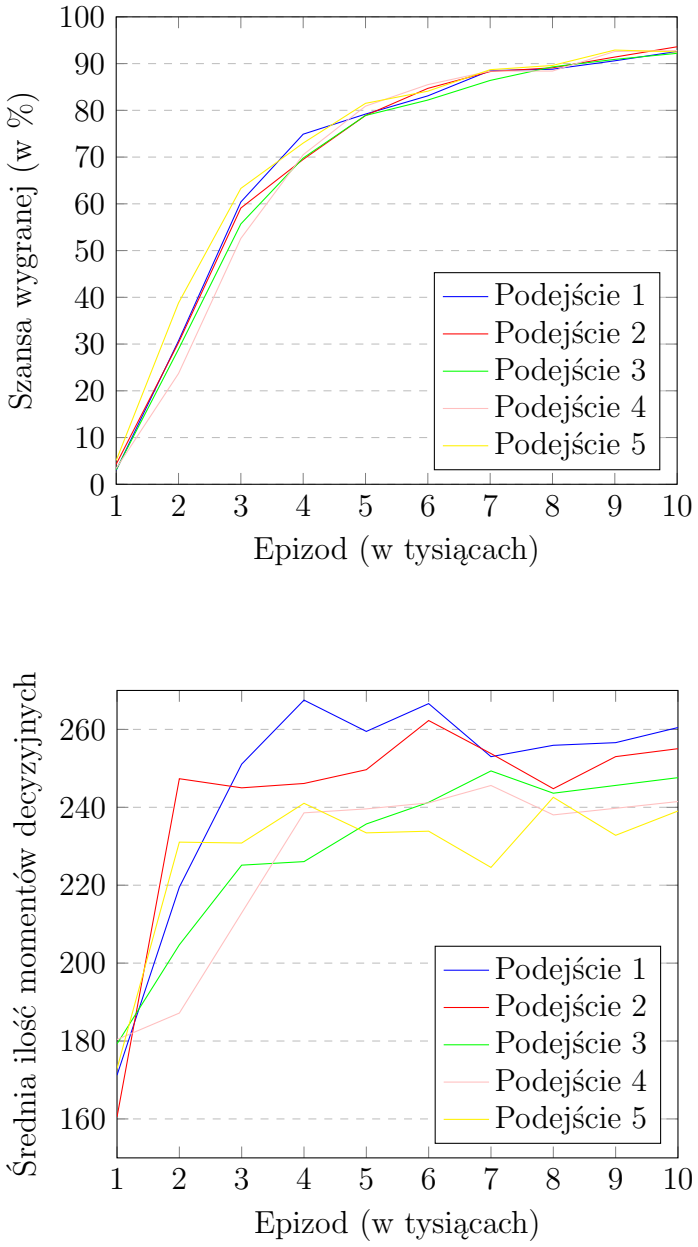
Przybliżę teraz kolejne linijki powyższego skryptu `main.py`. W pierwszej linijce skryptu zawiera się zaimportowanie głównej klasy mojego projektu - `Gameplay` (której importowanie wiąże się również z inicjalizacją środowiska PyGame, można to zaobserwować poprzez automatyczne wypisanie do terminala obecnej wersji biblioteki). Następnie wyszczególnione są zmienne konfiguracyjne związane kolejno z wykorzystywanym algorytmem, chęci wyświetlania reprezentacji graficznej w tej sesji i ścieżka do ewentualnego pliku z wygenerowanym już modelem. Ostatecznie inicjalizujemy obiekt klasy `Gameplay` (może to chwilę potrwać ze względu na alokowanie w pamięci generowanej Q-tablicy) oraz wywołujemy metodę `start` uruchamiającą środowisko.

To co może rzucić nam się w oczy zaraz po uruchomieniu środowiska, to pierwsze wyniki algorytmu wypisane w terminalu, wyglądające mniej więcej w taki sposób:

<1001 | Win chance: 0.0%> <AVG Rounds: NO DATA>

(pomięto tutaj jedną ze statystyk związaną z algorytmem ϵ-zachłannym, który zostanie opisany w dalszej części tego działu). Wartości te są wyliczane oraz wyświetlane z poziomu funkcji `start` i oznaczają kolejno: którego z kolei epizodu dotyczą statystyki, ile procent ostatnich epizodów (ilość epizodów ze zmiennej `render_per`) zostało ukończonych pomyślnie oraz ile średnio momentów decyzyjnych to zajęło (Tutaj wartość NO DATA występuje w przypadku braku pomyślnego ukończenia środowiska w ostatnich epizodach). Statystyka "Win chance" obliczana jest ze wzoru wins / self.render_per * 100 z czego wins to zmienna zawierająca liczbę "wygranych" epizodów (zwiększa się ona za każdym razem, gdy metoda `playthrough` zwróci win = True), resetowana do 0 przy każdym progu zmiennej `render_per`. Podobnie zachowuje się statystyka "AVG Rounds". Tutaj liczby zwracane przez zwycięskie wywołania `playthrough` (zmienna `rounds_c`) są sumowane i dzielone przez ich ilość. Jeśli metoda `start` została uruchomiona z parametrem `True`, powinniśmy również ujrzeć okno PyGame z renderowaną w czasie rzeczywistym (docelowo 60 klatek na sekundę) reprezentacją graficzną obecnie rozgrywanego epizodu.

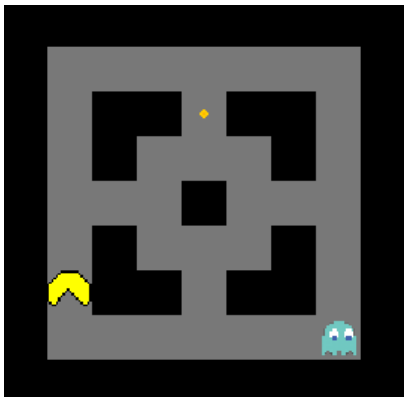
Przyjrzyjmy się wykresom przedstawiającym wyniki (statystyki z terminala) algorytmu Q-learning z pięciu podejść. Za każdym razem Q-tablica została wygenerowana losowo.



Rysunek 5: Wykresy przedstawiające statystki z 5 podejść dla algorytmu Q-learning (źródło własne)

Szansa wygranej przedstawiona na pierwszym wykresie jest swego rodzaju krzywą uczenia, unaocznia ona sam proces uczenia oraz jego efektywność. Drugi wykres przedstawia średnią ilość momentów decyzyjnych potrzebnych do pomyślnego ukończenia środowiska, ta zmienna wyrażana jest już w dużo większym zakresie wartości, nadal jednak dostrzegalny jest wzrost od średnio 172,86 do 248,73 momentów decyzyjnych

na epizod. Samo uczenie poskutkowało, Pac-Man jest w stanie przechodzić zadeklarowany poziom raz za razem z szansą na zwycięstwo na poziomie ponad 90% już po 10000 powtórzeń. Dodatkowo jest spora szansa na poprawienie tego wyniku lub nawet osiągnięcie 100%, gdyby przedłużyć działanie algorytmu. Niepokojący jest fakt powtarzającego się, wysokiego wyniku średniej ilość momentów dezycyjnych. Biorąc pod uwagę wielkość planszy oraz ilość pól możliwych do odwiedzenia ($(9*9)-44 = 37$), posiłkując się tylko i wyłącznie wynikami z terminala, można wydedukować dwa możliwe scenariusze. Spłaszczając lekko problem, podczas epizodu każde pole na planszy jest odwiedzane przez Pac-Man'a średnio 7 razy, tym samym nie zakańczając środowiska. Zważając jednak na strukturę problemu postawionego przed algorytmem oraz otrzymywane bardzo dobre wyniki, ten scenariusz jest mało prawdopodobny (agent dostaje duże nagrody za wydarzenia związane ze zbieraniem punktów i nie ma nic co mogłoby wskazywać na to, że ma tendencję do unikania ich). Dużo bardziej prawdopodobym scenariuszem jest "zacinanie" się Pac-Man'a na ścianach. Przyjrzyjmy się jednak temu, co dzieje się naprawdę przy pomocy reprezentacji graficznej w oknie pygame i jak się okazuje, problem jest nieco bardziej złożony i dodatkowo wystąpił on w każdym opisywanym powyżej podejściu. Pac-Man dochodzi do etapu, w którym pozostaje na planszy bardzo mało punktów do zebrania (1-3) i jego zachowanie wskazuje na to, jakby priorytetem dla niego stało się paniczne unikanie przeciwnika aniżeli szybsze zebranie ostatnich punktów i ukończenie środowiska. Zasugerowane przeze mnie blokowanie się Pac-Man'a na ścianach również występuje, ale nie przesądza o wielkości statystyki średniej ilości momentów decyzyjnych tak, jak jego "strach" przed ukończeniem środowiska.



Rysunek 6: migawka z reprezentacji graficznej ukazująca dotychczasowy problem (źródło własne)

Możnaby wysnuć teorię, że z czasem podczas uczenia ta tendencja będzie zanikać poprzez brak nagradzania agenta za puste ruchy, ale jest lepszy sposób, który pomoże agentowi uporać się z tym problem oraz uniknąć innych, które mogą się mu przydarzyć w przyszłości. Sposobem tym jest wykorzystanie algorytmu ϵ-zachłannego.

4.3.1 Algorytm ϵ-zachłanny

Algorytm ten jest na tyle przydatny, że postanowiłem zaimplementować go na stałę w moim środowisku. Jest on wykorzystywany w każdym podejściu przedstawianym w tej pracy. Dlatego też uruchamiając moje środowisko w takiej konfiguracji, jaka została opisana powyżej, można otrzymać z goła inne wyniki, w bazowej wersji programu algorytm ten jest wymuszany. Można jednak uruchomić środowisko bez tego algorytmu, zakomentowując w kodzie linijki odpowiedzialne za ten algorytm (co zrobiłem ja w poprzednich uruchomieniach, ukazując tym samym problem występujący podczas nie korzystania z niego), które będę opisywał w tym podrozdziale.

Główny filar algorytmu zawiera się w kilku linijkach metody `playthrough` na etapie wybierania przez strukturę akcji do wykonania przez agenta, a cały jego zamysł polega na pewnej losowości w ich podejmowaniu. Implementacja opiera się istnieniu procentowej szansy na zastąpienie wyboru akcji przez strukturę na wybór zupełnie losowy. Taki zabieg pozwoli agentowi na eksplorację przyspieszającą zaznajomienie się ze środowiskiem w czasie uczenia oraz wymusi na nim zmierzenie się ze stanami środowiska z którymi być może, patrząc na ogół procesu uczenia, nigdy by się nie spotkał. Szansa na wykonanie losowej akcji (eksploracji) jest zależna od programisty. W mojej implementacji zmniejsza się ona po każdym wykonanym epizodzie. Innym podejściem może być przyjęcie pewności eksploracji do pewnego progu w procesie uczenia, a następnie wyłączenie go w zupełności, wszystko zależne jest od przedstawionego problemu. Zmienna przechowująca wartość szansy przyjmuje wartości od 0 do 1, gdzie 0 oznacza stuprocentową szansę na wykonanie losowej akcji, a 1 to jej brak.

Przed przystąpieniem do losowania akcji, zapoznajmy się ze stałymi wykorzystywanymi w algorytmie zadeklarowanymi jako pola klasy `Gameplay`, a są nimi:

```
1 epsilon_episodes = 5_000
2 epsilon_decay = (1 - epsilon) / epsilon_episodes
3 epsilon_ceiling = 1
```

gdzie stała `epsilon_episodes` określa przez ile pierwszych epizodów ma występować pomniejszanie szans. W `epsilon_decay` zapisany jest krok pojedynczego pomniejszenia, a w `epsilon_ceiling` określona jest graniczna wartość szansy której algorytm nie może przekroczyć. Potrzebna jest również sama zmienna zawierająca procentową szansę wraz z jej wartością początkową (100% szansy na wykonanie losowej akcji):

```
1 epsilon = 0.0
```

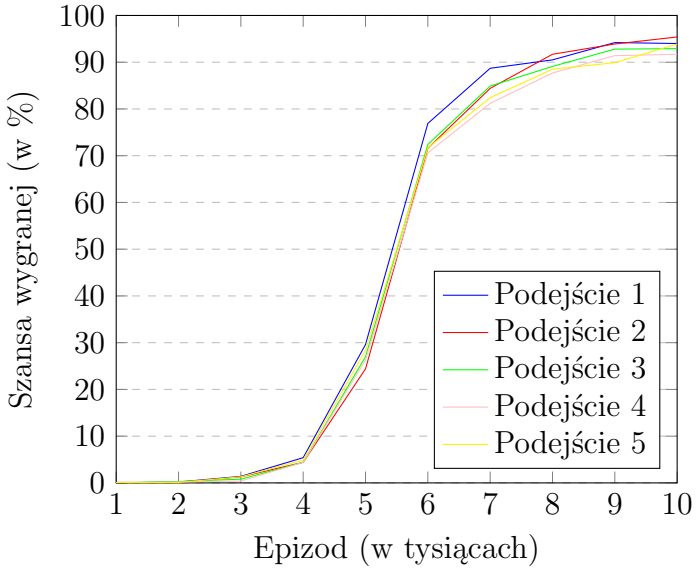
Wartość tej własnie zmiennej uwidaczniana jest jako statystyka `epsilon` w terminalu, którą uprzednio pominąłem przy tłumaczeniu. Przejdźmy do implementacji samego algorytmu w metodzie `playthrough`. Między krokiem algorytmu pierwszego pobrania stanu ze środowiska, a podjęciem decyzji o akcji włącznie, możemy dostrzec kod:

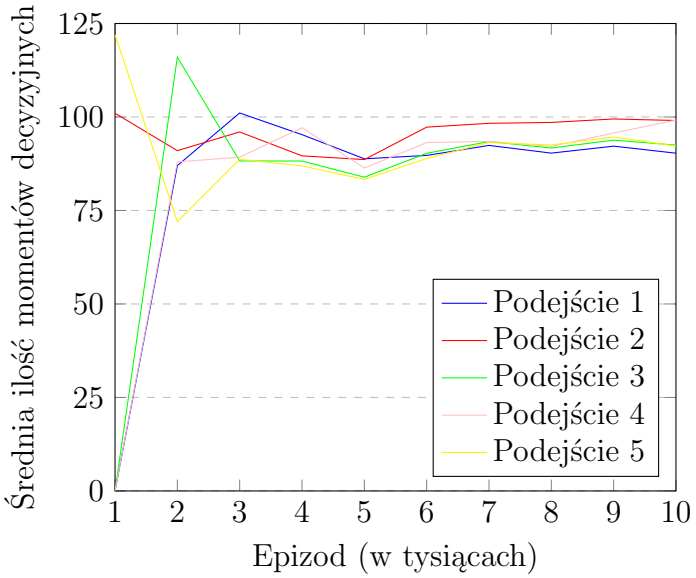
```
1  if self.epsilon >= self.epsilon_ceiling:
2    self.epsilon = self.epsilon_ceiling
3
4  action = None
5  if np.random.random() < self.epsilon:
6    # Tutaj występuje wybór akcji przez strukturę
7    # (część zależna od wybranego podejścia)
8  else:
9    action = np.random.randint(0, 4)
10
11 self.pacman.do_action(action)
```

Linijki 1-2 to upewnienie się co do zawierania się obecnej wartości zmiennej `epsilon` pod progiem wartości zmiennej `epsilon_ceiling`. Następnie w linijce 5 wykorzystywana jest funkcja `np.random.random()` losująca pseudo-losową wartość od 0 do 1, przyrównana do wartości szansy ze zmiennej `epsilon`. Samą akcję losuję w linijce 9. funkcją `np.random.randint(0, 4)`, zwracającą w tym przypadku liczbę naturalną z dziedziny {0, 1, 2, 3} odpowiadającej indeksowi akcji. Ostatnim dodatkiem do kodu jest zwiększenie zmiennej `epsilon` o obliczony krok `epsilon_decay` tuż przed zakończeniem epizodu (operacją return w metodzie `playthrough`).

```
1  self.epsilon += self.epsilon_decay
```

Sprawdźmy teraz jakie korzyści przynosi wykorzystywanie algorytmu ϵ-zachłannego podczas wyboru wykonywanej przez agenta akcji. Tak jak poprzednio, na wykresie przedstawiono 5 prób, z których każda rozpoczęta została na wylosowanej Q-tablicy.





Rysunek 7: Wykresy przedstawiające statystyki z 5 podejścia dla algorytmu Qlearning z uwzględnieniem algorytmu ϵ-zachłannego (źródło własne)

Pierwsze co rzuca się w oczy na wykresach to inny kształt krzywej uczenia (szansa wygranej), która wydaje się też bardziej stabilna. Z racji, że na początku losowość jest duża, szansa agenta na pomyślne ukończenie środowiska na początku jest nikła. Mogłoby się wydawać, że algorytm ϵ-zachłanny nie daje korzyści, jednak w miarę przybliżania się do epizodu granicznego (zatrzymania losowości w działaniach) szansa ta się zwiększa, a po przekroczeniu jej utrzymuje poziom porównywalny z tym bez wykorzystania algorytmu. Dużo bardziej istotnym faktem jest ogromny przeskok w średniej ilości wykonywanych momentów decyzyjnych. Zaznaczę tutaj, że średnia ilość przedstawiona na wykresie jako wartość 0 odnosi się do braku danych, ze względu na brak zwycięstw w ostatnim tysiącu epizodów. Biorąc pod uwagę tę statystykę pobraną po 10 tysiącach epizodów w przypadku algorytmu Q-learning bez eksploracji i z eksploracją mówimy tutaj o ponad dwukrotnym obniżeniu ze średnio 248,73 momentów decyzyjnych do 94,65 momentów decyzyjnych.

4.4 Zalety i wady

Podpierając się opisem oraz eksperymentami algorytmu Q-learning przedstawionymi w tym rozdziale przybliżę pozytywne i negatywne aspekty tego podejścia do rozwiązywania problemów decyzyjnych w moim środowisku. Algorytm ten jest bardzo prosty do przedstawienia. Delikatnie wyższym progiem wejścia charakteryzuje się samo pojęcie uczenia przez wzmacnianie, ale po jego zrozumieniu, algorytm Q-learning jest po prostu programistyczną implementacją tego pojęcia. Całość algorytmu jest modularna, posiada wiele parametrów którymi można manipulować, przez co pozwala na wszelkie próby, modyfikacje i eksperymenty dające gorsze lub lepsze wyniki (algorytm ϵ-zachłanny) jednak wszystko zależne jest od problemu przedstawionego. Sama implementacja to zastosowanie jednego wzoru w pętli, najcięższą rzeczą jest tu odpowiednie odwoływanie się do pól w Q-tablicy. Otrzymywanie tak dobrych wyników w tak krótkim czasie może utwierdzić w przekonaniu, że jest to jedno z lepszych podejść do tego typu problemów. Dużym minusem jest słaba skalowalność.

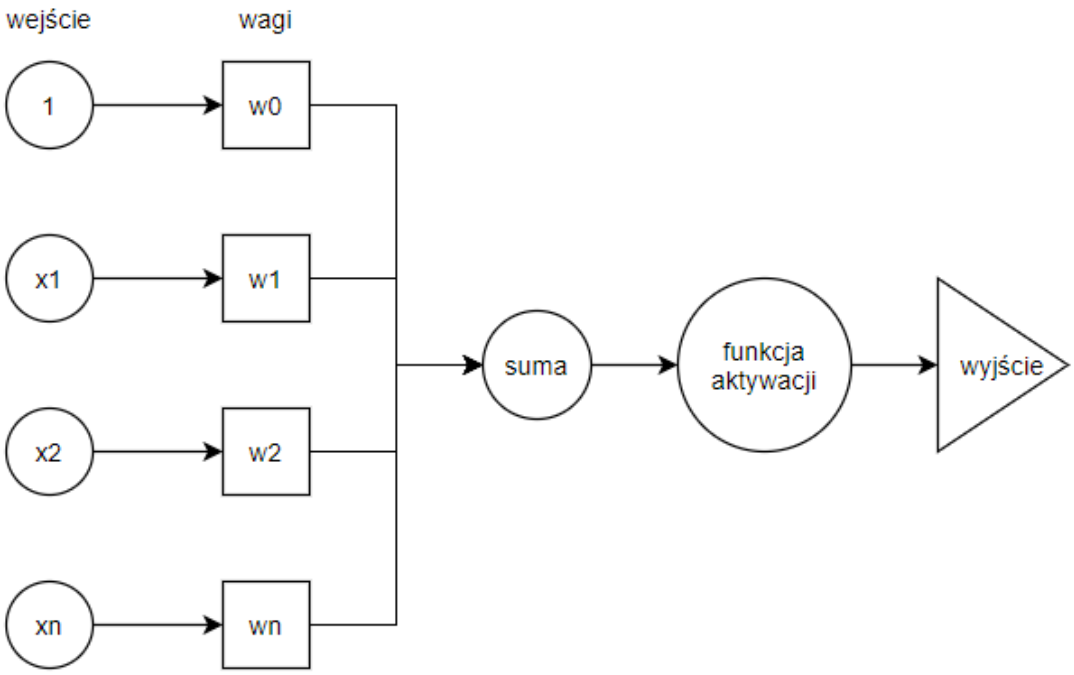
Wady tego podejścia mogą być niezauważalne na pierwszy rzut oka, zwłaszcza gdy podąża się z narracją tego rozdziału, dlatego podkreślę je tutaj. Środowisko zostało zbudowane tak, by można było w prosty sposób budować labirynty dla Pac-Man'a, więc ile użytkowników, tyle pomysłów na wyzwania dla agenta. Do tego implementacja algorytmu musi być na tyle elastyczna, by być gotowa na wszelkie zmiany związane z planszą. Tak jak skomplikowanie poziomu poprzez dodanie ślepych uliczek czy umieszczenie punktów w niebezpiecznych dla Pac-Man'a miejscach nie zaburzy działania algorytmu, tak powiększenie planszy, czy zwiększenie liczby przeciwników może być problemem. Każda składowa stanu środowiska oparta jest o rozmiar planszy, więc powiększenie jej wiąże się ze zwiększniem liczby kombinacji stanów w Q-tablicy. Jeszcze groźniejsze wydaje się zwiększenie liczby przeciwników, gdyż informacje o przeciwniku zawierają się w osobnej składowej, co potęguje zapotrzebowanie na pamięć. Algorytm zawiera więc potencjalne zagrożenie przepełnienia pamięci RAM przez Q-tablicę oraz sam zapis Q-tablicy zajmuje dużo miejsca. Dla przykładu, jeśli chcielibyśmy wygenerować Q-tablicę o takiej jak opisywana w tym rozdziale strukturze stanu, ale na planszy o rozmiarze 12 szerokości na 12 wysokości, w pamięci zajmowałaby ona już ponad 5 GB. Zapisywanie takiej Q-tablicy chociażby co 1000 epizodów bardzo szybko zapełniłoby niemałą część dysku, a i sam proces zapisywania opóźniałby wykonywanie algorytmu. W następnym rozdziale opiszę podejście, które wyjdzie poza ramy ograniczeń pamięciowych przeciętnego komputera, ale z wyższym zapotrzebowaniem czasowym.

5 Splotowa sieć neuronowa

Splotowe sieci neuronowe to klasa głębokich sieci neuronowych, która specjalizuje się w przetwarzaniu tabelarycznych danych, takich jak obraz [11], dlatego w tym podrozdziale najpierw postaram się wejść w szczegóły sieci neuronowych, które są wymagane do zrozumienia ich głębokiej i splotowej klasy. Następnie przedstawię zagadnienia potrzebne do wykorzystywania splotowych sieci neuronowych w zadeklarowanym przeze mnie środowisku.

5.1 Teoria sieci neuronowych

Tak jak przedstawione to zostało we wstępie tej pracy, sieć neuronowa w pewnym stopniu może być interpretowana jako sztuczny mózg. Każda sieć neuronowa to inaczej sieć perceptronów, czyli sztucznych odpowiedników biologicznych neuronów. O inspiracji i podobieństwach perceptronów do biologicznych neuronów będących w mózgu przeczytać można w [12]. Ogólnym zamysłem perceptronu jest możliwość wyliczenia pewnej wartości bazując na przynajmniej jednej danej wejściowej, w programowaniu ten efekt można skojarzyć z najzwyklejszą funkcją (podstawowe implementacje perceptronów mogą opierać się właśnie na pojedynczej funkcji). Kluczowym jednak w zrozumieniu perceptronu jest przedstawienie z czego się składa oraz co tak na prawdę dzieje się podczas jego działania.



Rysunek 8: Struktura perceptronu (źródło własne, wzorowane na [13])

Składowe oraz działanie perceptronu określę analizując rysunek 8, opierając swoją wiedzę na istniejącym już opracowaniu, do którego odniesienie znajduje się w bibliografii [13]. Zaczynając od lewej strony mamy zatem kolejno:

1. wejście - n liczb rzeczywistych będących ustandaryzowanymi danymi na których opiera się problem oraz jedna liczba naturalna 1, która w połączeniu ze swoją wagą ($w0$) daje wartość przesunięcia (bias). Jest ona konieczna w przypadku niektórych problemów, lecz ze względu na swoją przydatność wykorzystywana jest bazowo we wszelkich implementacjach.

2. wagi - $n + 1$ liczb rzeczywistych wykorzystywanych podczas wyliczania wartości wyjściowej perceptronu. Są to liczby które ulegają zmianom w procesie uczenia, to w nich zawarty jest potencjał całej sieci. Ich wartości są swego rodzaju kluczem do rozwiązań przedstawionego problemu tak, jak w przypadku algorytmu Q-learning kluczem była Q-tablica.

3. suma - wartość sumy ważonej wyliczana ze wzoru

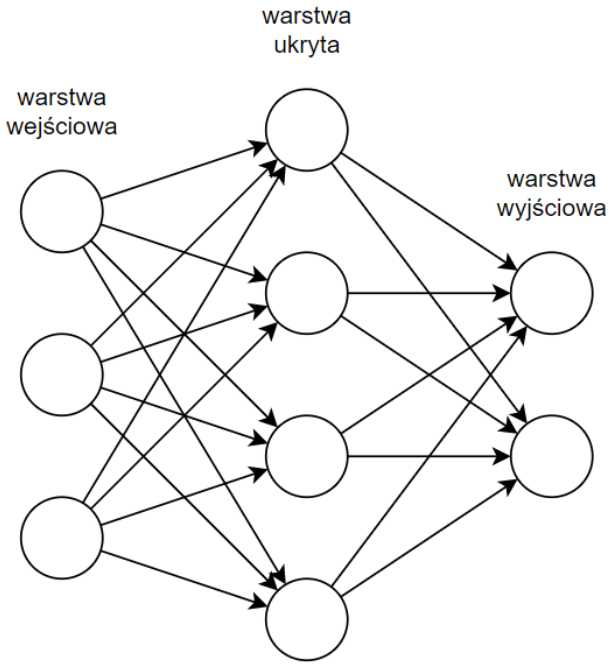
$$1 * w0 + (\sum_{i=1}^{n} xi * wi), \tag{2}$$

   obecność pierwszego iloczynu we wzorze wynika z wykorzystania wartości przesunięcia - bias.

4. funkcja aktywacji - funkcja przez którą należy "przepuścić" wynik sumy ważonej w celu utworzenia nieliniowości w zwracanych wynikach, co też może być kluczowe przy niektórych problemach. Nie jest powiedziane jednak, że sama funkcja nie może być liniowa. Funkcje aktywacji wykorzystywane przy moim problemie przedstawię w dalszej części pracy.

5. wyjście - wartość końcowa wyliczana przez cały proces działania perceptronu, od policzenia sumy ważonej na podstawie wejścia oraz swoich wag, wywołania funkcji aktywacji oraz pobrania jej wyniku.

Topologia sieci neuronowej ma strukturę warstwową (warstwy perceptronów) z czego każda warstwa może być innego typu (wykonywać obliczenia innego rodzaju) oraz posiadać różne liczby zawartych w nich perceptronów. Podstawowym typem jest warstwa gęsta (dense layer) przedstawiająca wyniki zwracane przez każdy perceptron z tej warstwy jako dane wejściowe do perceptronów z następnej. Dodatkowo, warstwa przetwarzająca "czyste" dane podawane prosto do sieci jest zwana warstwą wejściową, a warstwa przetwarzająca je jako ostatnia - warstwą wyjściową. Wyniki z warstwy wyjściowej interpretowane są jako wyniki całej sieci neuronowej. Każda warstwa pomiędzy warstwą wejściową a wyjściową nazywana jest warstwą ukrytą (hidden layer). Istnieje pewne rozróżnienie sieci neuronowych zawierających więcej niż jedną warstwę ukrytą, będąca klasą głębokich sieci neuronowych.



Rysunek 9: Topologia sieci neuronowej z trzema warstwami gęstymi (źródło własne)

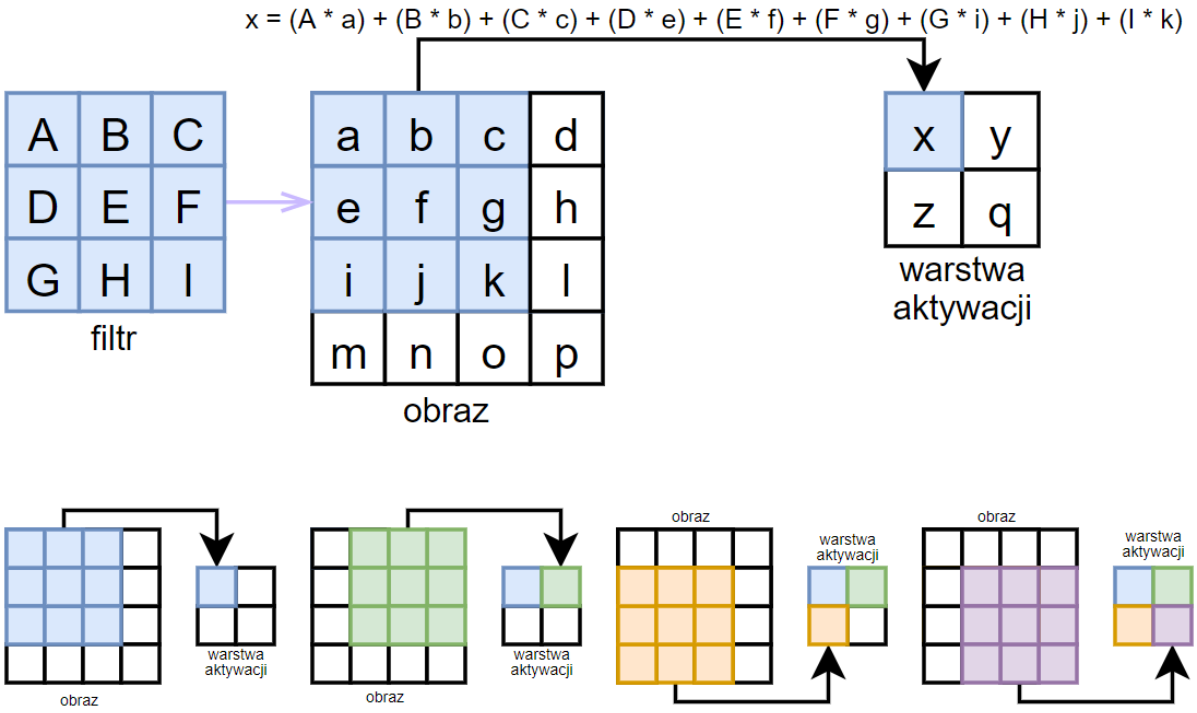
Warstwowa struktura sieci neuronowych może przysporzyć nie lada problemu jeśli chodzi o jej uczenie. Tak jak nauka pojedynczego perceptronu jest bardzo prostym procesem opierającym się na zastosowaniu jednego wzoru modyfikującego jego wagi [14] na podstawie wcześniej przygotowanych danych uczących (SPLA), tak zastosowanie tego schematu na całej sieci nie wchodzi w grę. Należy przejść o krok dalej i nie uczyć każdego perceptronu z osobna, a całą warstwę naraz. Z pomocą przychodzi nam algorytm wstecznej propagacji błędu, który podobnie jak SPLA swoje działanie opiera na przygotowanych wcześniej danych uczących. W dużym skrócie algorytm ten najpierw zapisuje wyniki zwracane przez każdy perceptron w sieci podczas wyliczeń na konkretnych danych wejścowych, liczy różnicę pomiędzy wynikami z całej sieci, a wynikami oczekiwanymi (dla podanych danych wejściowych), a następnie śledzi całą sieć warstwami wstecz (słowo klucz "wsteczna") korygując wagi wszystkich perceptronów, a zwłaszcza tych, które najbardziej przyczyniają się do zaistniałego błędu [14]. Korekcja odbywa się podobnie jak w przypadku algorytmu Q-learning z wykorzystaniem stałej współczynnika uczenia (learning rate).

5.2 Teoria splotowych sieci neuronowych

Splotowe sieci neuronowe są kolejną próbą odwzorowania funkcji biologicznego mózgu na podejście algorytmiczne. Tym razem to nie pojedyncza "myśl" opierająca się na możliwie nie do końca sprecyzowanych bodźcach, a przetworzenie i zrozumienie obrazu tak, jak robi to kora wzrokowa mózgu. Sieci te zawierają w swojej strukturze specjalną warstwę nazywaną splotową (convolutional). Specjalizuje się ona w przetwarzaniu danych zawartych w strukturze tabelarycznej, w domyśle głównie będą to obrazy [11].

Przewagą splotowych sieci neuronowych nad ich rodzimą klasą, którą jakby nie patrzeć również możemy przetwarzać obrazy (ówcześnie spłaszczając je w wektor wartości) jest zachowanie informacji przestrzennych (który piksel na obrazie znajduje się obok którego). Są one w stanie wykrywać cechy pokroju linii, zakrzywień, kształtów i kolorów, by potem głębiej w sieci łączyć te cechy w bardziej skomplikowane, pozwalające na wykrywanie lub klasyfikację całych przedmiotów lub twarzy. Można powiedzieć, że wykorzystywanie sieci splotowych umożliwia komputerom widzenie [11].

Nazwa splotowych sieci neuronowych pochodzi od wykorzystywanego w nich operatora splotu. W matematyce operator splotu zastosowany między dwoma funkcjami mierzy całkę z ich iloczynu punktowego, jednak w kontekście przetwarzania obrazów, splot stosowany jest pomiędzy dwoma tablicami, pierwsza z tablic to przetwarzany obraz, a druga to jądro konwolucji (kernel), inaczej zwana też filtrem. Wynikiem konwolucji w zastosowaniu pomiędzy tablicami jest również tablica (obraz) nazywana warstwą aktywacji [15].



Rysunek 10: Zobrazowanie operacji splotu na tablicy dwuwymiarowej z filtrem 3x3 (źródło własne)

Potocznie pojedynczą składową operacji konwolucji można tłumaczyć jako "przykładanie" filtra do kolejnych pikseli obrazu i wymnażając ich wartości przez odpowiadające im wartości z filtra, na końcu zsumowanie i składowanie ich w warstwie aktywacji. W tej warstwie, to nie wagi perceptronów są wartością uczoną, a wartości zawarte w każdym pojedynczym filtrze. Główna różnica pomiędzy perceptronem, a filtrem jest w nazewnictwie oraz w tym, w jaki sposób układają one swoje wagi (filtr układa je tabelarycznie, tym samym nie gubiąc informacji przestrzennych). Za to jak będzie

wyglądać operacja konwolucji w naszej warstwie odpowiada kilka czynników:

- Liczba filtrów - można interpretować ją jako odpowiednik liczby perceptronów w warstwie.
- Rozmiar filtra - odpowiada szerokości i wysokości filtra (liczby naturalne), który może przysłużyć się do kojarzenia informacji zawartych na większym obszarze. Rozmiar filtra poniekąd definiuje też jakiej wielkości będzie wynikowa warstwa aktywacyjna.
- Krok - co ile pól (pikseli) filtr ma przesuwać się po powierzchni obrazu, najczęściej wykorzystywany jest krok jeden piksel w osi horyzontalnej, a potem jeden piksel w osi wertykalnej. Razem z rozmiarem filtra ma wkład w rozmiar warstwy aktywacyjnej.
- Dopełnienie - problem rozwiązywany przez warstwę splotową może wymagać utrzymania rozmiaru przetwarzanej tablicy. Takie podejście dopełnienia jest nazywane "same padding" i w tym przypadku pola, nie poddawane konwolucji wypełniane są zerami. Podejście dopełnienia z redukcją tablicy nazywane jest "valid padding".

Podobnie jak w przypadku warstw gęstych, uczenie może odbywać się przy pomocy algorytmu wstecznej propagacji błędu. Błąd dla poszczególnych pól w filtrze rozdzielany jest wtedy poprzez odwrotną operację konwolucji. Po więcej informacji na temat algorytmu wstecznej propagacji w sieciach splotowych oraz samych sieci zapraszam na stronę [16] oraz do pracy [17].

5.3 Przygotowanie do części praktycznej

Operacje wykonywane podczas działania sieci neuronowych wymagają bardzo dużego potencjału obliczeniowego. Obliczenia są proste, ale jest ich dużo. Większość implementacji wykorzystuje tę zależność w celu wykonywania ich na kartach graficznych, co znacznie przyspiesza działanie. Dlatego, by nie wynajdywać koła od początku, wykorzystam w projektowaniu oraz uruchamianiu modelu sieci neuronowych bibliotekę TensorFlow [18] opierającą się o prosty do zrozumienia interfejs, który będzie tłumaczony na bieżąco w trakcie korzystania z niego w dalszej części pracy.

W odróżnieniu od algorytmu Q-learning, który w domyśle daje konkretne wyniki, sieć neuronowa może zwracać najróżniejsze, bardziej nieprzewidywalne wyniki, które niekoniecznie dobrze działają z problemami pokroju środowiska gry. Ze względu na to, jej parametry muszą być staranniej skonfigurowane oraz trzeba wspomóc sieć w niektórych aspektach. Najważniejszym z nich jest dostęp sieci do danych uczących. Nasze środowisko nie oferuje żadnej bazy danych z odpowiedziami który ruch w danym momencie decyzyjnym jest najbardziej opłacalny. Wygenerowanie takich scenariuszy zajęłoby bardzo dużo czasu, gdyż by robić to sumiennie i zapełnić całą ich dziedzinę, musiałoby to być robione ręcznie lub w najlepszym przypadku jakimś algorytmem symulującym odpowiedź na każdy stan środowiska, na przykład innym algorytmem

uczenia maszynowego, co na tym etapie przestaje mieć sens. Sprytniejszym podejściem jest zainicjalizowanie obok sieci jej własnej bazy danych, która będzie wypełniana wspomnieniami z poprzednich momentów decyzyjnych. Podejście które tutaj opisuje nie jest typowym dla sieci neuronowych. Tak jak zaznaczyłem wcześniej, sieci neuronowe nie są przystosowane do zwracania konkretnych, jednogłośnych wyników, jednak dobrze radzą sobie ze skojarzaniem danych wejściowych, co wydaje się też bardzo dobrą cechą przy chociażby ponownym wykorzystywaniu sieci (na przykład przy wykorzystaniu sieci w środowisku z inną liczbą przeciwników, a nauczonej na innej). Jest pewna metoda spalająca aspekty algorytmu Q-learning oraz sieci neuronowych nazywana głęboką Q-siecią (Deep Q-Network, DQN). To, co jest największa wadą algorytmu Q-learning, czyli zasobożerność Q-tablicy, jest w tym podejściu podmieniona własnie na dużo bardziej optymalną sieć neuronową [19].

5.3.1 Deep Q-Network

Przejście z sieci neuronowej na algorytm DQN wiąże się z pewnymi wymogami. Najważniejszym jest przygotowanie wcześniej wspominanej już bazy danych uczących zwanej pamięcią. Pamięć ta ma za zadanie przetrzymywać skończoną (ustaloną jako parametr) liczbę wspomnień o strukturze zawierającej wszystkie kluczowe informacje z pojedynczego momentu decyzyjnego, to jest:

- stan,
- wykonana akcja,
- wartość nagrody za wykonaną akcję na danym stanie,
- stan zaistniały po wykonanej akcji,
- czy zaistniały stan jest stanem wynikowym w epizodzie przedstawiający porażkę lub zwycięstwo (czy to ostatni stan w epizodzie).

Dodatkowymi parametrami wykorzystywanymi podczas obsługi wspomnień są: rozmiar partii wspomnień na jakich przeprowadzane jest szkolenie oraz co jaką ilość momentów decyzyjnych ma być przeprowadzane szkolenie. Bazowo, sieci neuronowe (a przynajmniej implementacja TensorFlow) nie są przystosowane do przetwarzania lub nawet brania pod uwagę takich wartości jak nagroda i współczynnik dyskontowy co jest na porządku dziennym w algorytmie Q-learning, dlatego to podlega własnej implementacji. Sam proces uczenia opiera się też na lekko zmodyfikowanym wzorze (1) znanym z algorytmu Q-learning. Modyfikacja polega na opuszczeniu wartości współczynnika uczenia, ze względu na już istniejącą jego obsługę w samej sieci. Wzór sprowadza się więc do postaci

$$Q = R_a^s + \gamma * maxQ(s_{t+1}), \tag{3}$$

wartość Q przypisywana będzie do tablicy wynikowej sieci w polu wykonanej przez agenta akcji, a następnie przy pomocy algorytmu wstecznej propagacji błędu cała sieć będzie trenowana w oparciu o wyprodukowaną tablicę. Stosowanie tego wzoru

pozwala zachować umiejętność kojarzenia ciągów prowadzących do pożądanego celu, znaną z algorytmu Q-learning [19, 20]. Tak rozwiązany początkowy problem braku danych uczących sprawia, że w trakcie procesu uczenia do pamięci powinno trafiać coraz więcej dobrze nagradzanych akcji, a co z tego wynika, wystąpi większa szansa na podjęcie prawidłowych decyzji.

Ważne w tym podejściu jest również zastosowanie nie jednej, a dwóch równoległych sieci neuronowych. Z założenia, będą one miały te same wagi przez większość działania programu (przy generowaniu sieci kopiuje się wagi z jednej do drugiej). Rozróżnione one są na zwykły model, który będzie wykorzystywany do pobierania wyników oraz model dopasowujący (fit model), który będzie uczony przy pomocy wspomnień z partii, a na koniec sesji jego wagi będą kopiowane do zwykłego modelu. Takie rozwiązanie pomoże nam w jakimś stopniu uniknąć przeuczenia sieci oraz robienia zbyt dużych skoków decyzyjnych sieci pomiędzy przetwarzaniami kolejnych wspomnień z partii [20].

5.4 Praktyka

Tak jak w przypadku algorytmu Q-learning, zmiana algorytmu odbywa się poprzez zmienną `ALGORITHM_USED`. Chcąc wybrać podejście Deep Q-Network z wykorzystaniem splotowych sieci neuronowych należy ustawić ten parametr na wartość "DQN". Najpierw jednak przyjrzyjmy się stałym zadeklarowanym jako pola klasy `Gameplay`, które będą kluczowe podczas uruchamiania tego podejścia. Oprócz znanych już stałych odpowiedzialnych za system nagród, obsługę uruchomienia oraz algorytm ϵ-zachłanny znajdziemy jeszcze takie parametry jak:

- `memory_size` będący liczbą naturalną deklarujacą wielkość pamięci wspomnień agenta,
- `batch_size` to liczba naturalna ustalająca z jak dużą partią wspomnień z pamięci ma być uruchamiana pojedyncza sesja uczenia sieci,
- `FIT_EVERY` i `FIT_COUNTER` to zmienne odpowiadające temu, co ile momentów decyzyjnych ma dojść do sesji uczenia sieci oraz druga zmienna będąca licznikiem podjętych momentów decyzyjnych na której podstawie określane jest, czy sesja uczenia powinna się odbyć po obecnym momencie decyzyjnym, wartość tej zmiennej z założenia musi na początku posiadać wartość 0.

Uruchomienie programu z parametrem "DQN" zainicjalizuje podczas konstrukcji obiektu klasy `Gameplay` wymagane struktury, są to:

- `memory` zmienna przechowująca strukturę z biblioteki `collections` zwaną `deque`, która działa jak zwykła tablica jednowymiarowa z lekko zmodyfikowaną operacją dodawania nowego obiektu, która najpierw sprawdza, czy osiągnięto limit obiektów w `deque` zadeklarowany jako parametr `maxlen`, jeśli tak, to dodając nadmiarowy obiekt struktura automatycznie robi dla niego miejsce usuwając najstarszy element,

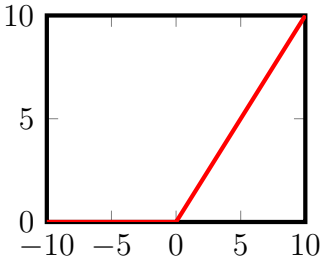
- `model` i `fit_model` to zmienne przechowujące odnośniki do dwóch osobnych, lecz początkowo identycznych sieci (ze względu na początkowe skopiowanie wag z sieci `model` do sieci `fit_model` przy pomocy set'era i get'era wag obiektu klasy `Sequential` z biblioteki `TensorFlow` oznaczający sekwencyjny model sieci) zainicjalizowanych przy pomocy funkcji `create_cnn_model`, przedstawiającej się następująco:

```
1  def create_cnn_model(self, file=None):
2    board_width = self.starting_board.size[0]
3    board_height = self.starting_board.size[1]
4    model = Sequential()
5    model.add(Conv2D(64, (1, 1),
6        activation="relu",
7        input_shape=(board_width, board_height, 3)))
8    model.add(Conv2D(128, (2, 2), activation="relu"))
9    model.add(Conv2D(256, (3, 3), activation="relu"))
10   model.add(Flatten())
11   model.add(Dense(self.action_size, activation="linear"))
12   model.compile(loss="mse", optimizer=Adam(lr=self.
       learning_rate))
13
14   if file:
15     model.load_weights(file)
16
17   model.summary()
18
19   return model
```

Funkcja `create_cnn_model` całościowo opiera się na rozwiązaniach zaproponowanych przez bibliotekę `TensorFlow`, dlatego po bardziej szczegółowe informacje odsyłam na oficjalną stronę biblioteki - `tensorflow.org`. Funkcja przyjmuje opcjonalny argument `file` będący ścieżką do zapisanych wag, bez tego argumentu sieć zostanie zainicjalizowana wagami o losowych wartościach. Główny obiekt inicjalizowany w linijce 4 to obiekt klasy `Sequential` będący modelem sieci neuronowej o najzwyklejszym stosie warstw ułożonych sekwencyjnie, które są kolejno dodawane przy pomocy metody `add`. Wartości przy warstwach ustawione są w sposób dający jak najbardziej optymalne wyniki dla domyślnej planszy przedstawionej na rysunku 3, wykorzystywanej dla porównania również w algorytmie Q-learning. Warstwa splotowa operująca na dwuwymiarowych tablicach w `TensorFlow` odwzorowywana jest poprzez klasę `Conv2D` i jako parametry w konstruktorze przyjmuje między innymi liczbę filtrów, ich rozmiar, funkcję aktywacji oraz w przypadku warstwy wejściowej "kształt" danych wejściowych, tutaj zadeklarowanych jako migawka z gry o wymiarach szerokość na wysokość na 3. Ostatni wymiar (3) oznacza wartości każdego kanału koloru RGB pola. Kolejna warstwa jest warstwą typu `Flatten`, która sama w sobie nie posiada żadnych wag, a jej zadaniem jest, jak sama nazwa wskazuje, przekształcenie danych przechodzących przez nią w płaski wektor (przypominam, że wynikiem warstwy splotowej są obrazy) który

następnie można wprowadzić do warstwy gęstej, w `TensorFlow` reprezentowanej przez klasę `Dense`, której w powyższym kodzie liczba perceptronów (pierwszy argument) odpowiada liczbie akcji możliwych do wykonania w zadeklarowanym środowisku, a drugi argument - zastosowanej funkcji aktywacji. Jest to konieczna w omawianym problemie wyjściowa warstwa sieci.

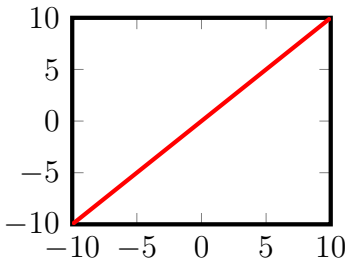
Pora przyjrzeć się jak wyglądają omawiane przy teorii sieci neuronowych funkcje aktywacji. Każda wartość wyliczona przez sumę ważoną w przypadku perceptronów, a w filtrach poprzez operator splotu, jest przepuszczana przez funkcję aktywacji zadaną dla odpowiadającej jej warstwy. W zadeklarowanej w powyższym kodzie sieci wykorzystywane funkcje oznaczane są jako "relu" dla warstw splotowych oraz "linear" dla warstwy gęstej, po więcej przykładów funkcji aktywacji oraz ich opisów odsyłam do [21]. W celu optymalnego uczenia głębokiej sieci neuronowej przy użyciu algorytmu wstecznej propagacji ważnym jest wykorzystanie funkcji aktywacji, która jak najbardziej przypomina funkcję liniową (ze względu na prostotę wyliczenia wartości pochodnych wykorzystywaną w algorytmie wstecznej propagacji), ale tak naprawdę zapewnia nieliniowość pozwalającą na wykrywanie bardziej skomplikowanych związków pomiędzy danymi [22]. Właśnie taką funkcją jest funkcja ReLU (Rectified Linear Unit). Wykres jej wartości przebiega następująco:



Rectified Linear Unit

$$ReLU(x) = \begin{cases} x \leqslant 0, & 0 \\ x > 0, & x \end{cases} = max(0, x)$$

Funkcja zakodowana pod nazwą "linear" odpowiada najzwyklejszej funkcji liniowej. Wykorzystywana jest w wyjściowej warstwie gęstej ze względu na chęć czystego przetworzenia oraz podania jako wyjście informacji z poprzedzających warstw splotowych. Wzór jest trywialny:



Linear

$$Linear(x) = x$$

Dalej w kodzie (linijka 12) występuje wywołanie metody `compile` z argumentami: `loss` odpowiadającym temu, jakiej funkcji obliczającej błąd (stratę) należy użyć. Wartość "mse" oznacza funkcję straty Mean Square Error, średni błąd kwadratowy. Po więcej informacji o tej i innych funkcjach straty odsyłam do [23]. Drugim argumentem jest argument `optimizer` z wartością będącą obiektem klasy `Adam` z parametrem współczynnika uczenia. Adam (skrót od Adaptive Moment Estimation) jest jednym z wielu optymalizatorów [24]. Algorytmy optymalizatorów odpowiedzialne są za podejście do zmniejszania błędu wyliczanego przez funkcję straty. Ich działanie można wyobrazić sobie jako zdefiniowany sposób poruszania się piłki po płaszczyźnie nakreślonej przez funkcję błędu dla różnych danych wejściowych, gdzie piłka w naturalny sposób wyszukuje minimum tej funkcji. Współczynnik uczenia odpowiada za szybkość poruszania się piłki na płaszczyźnie.

Ostatnim fragmentem tej metody jest załadowanie wag z pliku do którego ścieżka podana jest w argumencie funkcji (o ile taki podano) oraz wyświetlenie sformatowanego podsumowania modelu i zwrócenie go.

Następną częścią rozróżniającą podejścia jest pobieranie stanu środowiska. W przypadku sieci splotowej wejściem jest obraz (migawka) z gry, należy ją więc stworzyć i odpowiednio przygotować. Tutaj wykorzystałem fakt, że całe środowisko jest moim autorskim projektem i zamiast do sieci podawać obraz z gry widoczny przez użytkownika (ten render'owany w oknie pygame), który jest bardziej złożony, podaję uproszczony obraz będący kolorowym odwzorowaniem struktur przechowujących stan na planszy ze zmiennej `board` oraz zmiennych z klasy `Creature`. Pobranie migawki zawarte jest w metodzie `get_board_image`, która przedstawia się następująco:

```
1  def get_board_image(self):
2    data = np.zeros((1, self.board.size[0], self.board.size[1],
       3), dtype=np.uint8)
3    for i in range(self.board.size[1]):
4      for j in range(self.board.size[0]):
5        if self.board.board[(j, i)] == "W":
6          data[0][j][i] = (0, 0, 0)
7        elif self.board.board[(j, i)] == "X":
8          data[0][j][i] = (255, 255, 255)
9        elif self.board.board[(j, i)] == ".":
10         data[0][j][i] = (0, 255, 0)
11   data[0][self.pacman.x][self.pacman.y] = (0, 0, 255)
12   for ghost in self.ghosts:
13     data[0][ghost.x][ghost.y] = (255, 0, 0)
14   return data/255
```

Na początku w funkcji deklaruję tablicę typu `numpy` o rozmiarach odpowiadających wielkości planszy z klasy `Board` i dodatkowym wymiarem przechowującym wartość RGB koloru pola (3), początkowo wypełnioną zerami (cała czarna). Pierwszy zadeklarowany wymiar (1) uwzględniony jest na potrzeby wymogu metody `predict` z bibliote-

ki `TensorFlow`, która dla sieci splotowych domyślnie operuje na kilku obrazach naraz. Ze względu na zaimplementowane operacje pośrednie związane z DQN, wymuszamy operowanie na pojedynczym obrazie, dlatego też dalej wymagane jest dodatkowe odwołanie "`[0]`" do zmiennej `data`. Następnie w dwuwymiarowej pętli po współrzędnych planszy kolorowany jest odpowiednio obraz w zależności od występującego w tym polu obiektu, poza pętlą znajduje się jeszcze kolorowanie pola zajętego przez Pac-Man'a oraz jego przeciwników. To jakie kolory zostaną wybrane nie ma większego znaczenia, lecz może się przydarzyć, że bardziej kontrastujące kolory będą działać odrobinę lepiej, ważnym jest jedynie, by być w tym konsekwentnym. Zastosowanie innego koloru w obrazie jako wejście dla sieci nauczonej na innym może skutkować utratą potencjału sieci. Z funkcji zwracana jest ustandaryzowana (podzielenie przez 255 zawęża dziedzinę tych wartości na przedział od 0 do 1) struktura przechowywana w zmiennej `data`.

Pobranie stanu do zmiennej oraz wybranie akcji do wykonania, którego sposób nadal jest warunkowany przez algorytm ϵ-zachłanny, w przypadku sieci neuronowych odbywa się następująco:

```
1  state = self.get_board_image()
2
3  if self.epsilon >= self.epsilon_ceiling:
4    self.epsilon = self.epsilon_ceiling
5
6  action = None
7  if np.random.random() < self.epsilon:
8    action = np.argmax(self.model.predict(state)[0])
9  else:
10   action = np.random.randint(0, 4)
11
12 self.pacman.do_action(action)
```

Kluczowa różnica do algorytmu Q-learning znajduje się w linijce 8 w której pobierana jest akcja do wykonania. W tym przypadku najpierw wywoływana jest metoda `predict` zmiennej `model` zwracająca dwuwymiarową tablicę z czterema wartościami, odwołanie "[0]" wycina niepotrzebny wymiar zostawiając tylko potrzebne wartości będące wynikami z czterech perceptronów z ostatniej warstwy sieci. Wybór indeksu akcji odbywa się poprzez sprawdzenie jaki jest indeks największej wartości w tym wektorze (funkcja `argmax` z biblioteki `numpy`).

Po wykonanej akcji przechodzimy do pobrania nowego stanu oraz dodajemy zestaw informacji z momentu decyzyjnego do pamięci - struktury `deque`.

```
1 new_state = self.get_board_image()
2 self.memory.append((state, action, reward, new_state, done))
```

Zaraz po dodaniu nowego wspomnienia do pamięci, sprawdzany jest warunek konieczny do rozpoczęcia sesji uczenia. Warunkiem tym jest zapełnienie pamięci przynajmniej tyloma wspomnieniami, z ilu ma składać się sesja uczenia (wartość zadeklarowana w zmiennej `batch_size`) oraz zmienna licząca momenty decyzyjne `FIT_COUNTER` jest większa od parametru `FIT_EVERY`. Po pierwszym uruchomieniu sesji uczenia, następne sesje będą wywoływane właśnie co liczbę momentów decyzyjnych zadeklarowaną w `FIT_EVERY`.

```
1  if len(self.memory) > self.batch_size and self.FIT_COUNTER >=
       self.FIT_EVERY:
2    self.FIT_COUNTER = 0
3    batch = np.random.sample(self.memory, self.batch_size)
4    for state, action, reward, new_state, done in batch:
5      q_value = reward
6      if not done:
7        q_value = (reward + self.discount_factor * np.max(self.
             model.predict(new_state)[0]))
8      q_vector = self.model.predict(state)
9      q_vector[0][action] = q_value
10     self.fit_model.fit(state, q_vector, epochs=1, verbose=0)
11   self.model.set_weights(self.fit_model.get_weights())
```

Pierwsza rzecz wykonywana w każdej sesji uczenia to wyzerowanie licznika oraz pobranie losowej próbki wspomnień o wielkości `batch_size` z pamięci przy pomocy funkcji `sample` z biblioteki `random` oraz przypisanie jej do zmiennej `batch`. Następnie wykorzystywana jest składnia rozpakowująca pojedynczy element tablicy i iterująca po niej, w ten sposób każda informacja zapisana we wspomnieniu przypisana jest do swojej oddzielnej zmiennej. Kolejność nazywania zmiennych jest konsekwentna z kolejnością zapisywania ich. W pętli po wspomnieniach odbywa się faktyczny proces uczenia z rozwinięciami związanymi z implementacją DQN. Linijki 5 i 6 porównać można do ustalenia Q-wartości w algorytmie Q-learning. Do zmiennej `q_value` przypisywana jest wartość ze wspomnienia `reward` odpowiedzialna za wartość nagrody, a jeśli moment ten nie był ostatnim w epizodzie, to wartość `q_value` wyliczana jest przy pomocy wzoru 3. Wartość $maxQ(s_{t+1})$ ze wzoru ustalana jest poprzez odpytanie sieci o wynik na podstawie danych wejściowych z `new_state` (ze względu na indeks $t+1$) oraz wybranie wartości maksymalnej z czterech zwróconych (`np.max()`). Z racji, że algorytm wstecznej propagacji wymaga danych w takim samym formacie jak zwracane wyniki, przypisanie Q-wartości odbywać się będzie poprzez uprzednie pobranie tablicy wynikowej z sieci na podstawie stanu `state` do zmiennej `q_vector`, a następnie podmienienie Q-wartości w polu odpowiedzialnym za wykonaną w tym momencie decyzyjnym akcji. Tak spreparowaną tablicę wykorzystamy teraz w uczeniu sieci (algorytmie wstecznej propagacji). Przypominam i zaznaczam, że pomocnym w podejściu DQN jest zastosowanie dwóch sieci neuronowych, dlatego każde wywołanie metody `predict` odbywa się na obiekcie `model`, a uczenie metodą `fit` na obiekcie `fit_model`. Metoda `fit` przyjmuje kilka argumentów, pierwszym z nich są dane wejściowe, a drugim tablica, na podstawie której wyliczany jest błąd pomiędzy wynikami z sieci, a zadaną tablicą.

Dalszymi argumentami są `epochs` ustawiający liczbę epok pojedynczej sesji uczenia sieci `TensorFlow`. Przez to, że wykonujemy operacje pośrednie związane z DQN, wymuszamy pojedyncze epoki, dalej argument `verbose` odpowiada temu, w jaki sposób mają być wypisywane statystyki związane z tą sesją uczenia, wartość 0 odpowiada brakowi takich wypisań ze względu na zbyt dużą ich ilość oraz temu, że potrzebne statystyki są prowadzone samodzielnie przez środowisko. Po wyjściu z pętli w linijce 11 następuje przepisanie wag z sieci uczącej, do właściwiej, a po przejściu każdego momentu decyzyjnego, na końcu zwiększamy licznik momentów `FIT_COUNTER` o jeden.

```
1 self.FIT_COUNTER += 1
```

5.5 Uruchomienie

Podobnie jak przy poprzednim uruchomieniu trzeba zadbać o konfigurację struktury i środowiska z poziomu pól klasy `Gameplay`, której konfiguracja w uruchomieniach na których będą opierane badania wygląda następująco:

```
1  episodes = 15_000
2  render_per = 1_000
3
4  death_penalty = -5
5  win_reward = 1
6  point_reward = 0.25
7  move_penalty = -0.001
8
9  epsilon = 0
10 epsilon_episodes = 10_000
11 epsilon_decay = (1 - epsilon) / epsilon_episodes
12 epsilon_ceiling = 1
13
14 learning_rate = 0.0001
15 discount_factor = 0.95
16
17 memory_size = 500
18 batch_size = 64
19 FIT_EVERY = 100
```

Największą różnicą względem konfiguracji z algorytmu Q-learning są wartości z systemu nagród oraz wartość współczynnika uczenia, są one o wiele mniejsze, ze względu na dane na których operuje sieć oraz na delikatność całej struktury. Tak jak pisałem na początku przechodząc do tego podejścia, jest ono o wiele bardziej czasochłonne i nie chodzi tu tylko o skomplikowaność obliczeń ale i ilość wymaganych epizodów która

z 5 tysięcy skoczyła do 15 tysięcy. Tak samo liczba epizodów w czasie których następuje stopniowe powiększanie epsilonu.

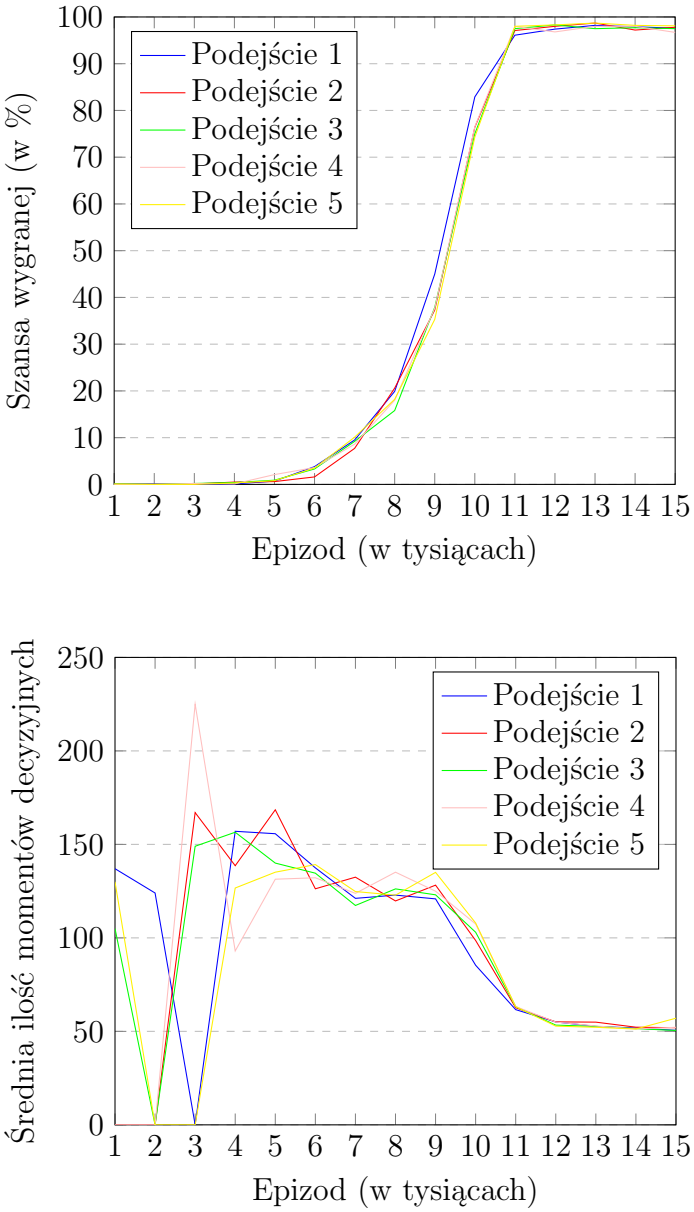
Oczywistym jest również, że chcąc wykorzystać podejście DQN, należy zmienić wartość zmiennej `ALGORITHM_USED` na "DQN" w moim przypadku w skrypcie uruchamiającym środowisko - `main.py`.

```
ALGORITHM_USED = 'DQN'
```

W przypadku uruchomienia środowiska opartego na sieciach neuronowych, należy uzbroić się w cierpliwość. Nawet jeśli maszyna na której wykonywane są obliczenia wykorzystuje do nich kartę graficzną, to ich ilość w pojedynczym uczeniu jest zatrważająca, a mnoży to jeszcze fakt, że nie wykorzystujemy pełni potencjału oferowanego przez bibliotekę `TensorFlow` z racji modyfikacji na które nie jest przygotowana (rozwiązania DQN). Metoda `summary` wywoływana podczas tworzenia sieci pokazuje w terminalu sformatowane informacje o strukturze samej sieci oraz że łączna ilość parametrów, które będą ulegać procesowi uczenia to 365188.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 9, 9, 64)          256
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 8, 8, 128)         32896
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 6, 6, 256)         295168
_________________________________________________________________
flatten (Flatten)            (None, 9216)              0
_________________________________________________________________
dense (Dense)                (None, 4)                 36868
=================================================================
Total params: 365,188
Trainable params: 365,188
Non-trainable params: 0
```

Wyniki tego podejścia prezentują sie następująco:



Rysunek 11: Wykresy przedstawiające statystki z 5 podejść dla algorytmu DQN (źródło własne)

Szansa wygranej, tak jak możnaby się spodziewać, rośnie bardzo szybko w okolicach osiągania przez zmienną epsilon wartości maksymalnej. Po 15 tysiącach epizodów średnia szansa na wygraną wynosi 97.5%. Wynik ten jest delikatnie lepszy, lecz nadal porównywalny z podejściem algorytmu Q-learning (okolice 94%). Co jest bardziej zauważalne to dużo mniejsza średnia ilość momentów decyzyjnych. Wartość tej statystyki spadła niemalże dwukrotnie z 94,65 do 52,44 momentów, co nie powinno być zaskoczeniem. Splotowa sieć neuronowa, jakby nie patrzeć, operuje na dużo większej ilości informacji pokroju samego układu planszy. Agent ma zatem świadomość dróg jakie podejmuje oraz jakie drogi są podejmowane przez jego przeciwników.

5.6 Zalety i wady

Najważniejszą zaletą podejścia DQN, przyrównując go do algorytmu Q-learning, jest stosunek zapotrzebowania sprzętowego do potencjału samego algorytmu. Pozwala to na dużo większą swobodę w wykorzystywaniu i eksperymentowaniu. Dla porównania, poniżej przedstawiam tabelę ukazującą przybliżone rozmiary plików zapisu struktur uczących oraz ich zapotrzebowanie na pamięć RAM (pomiar wielkości struktury w pamięci RAM został wykonany przy pomocy Python'owej funkcji `sys.getsizeof()`) dla obu podejść, z konfiguracjami z odpowiadających im działów tej pracy. W przypadku algorytmu Q-learning pamięć zajmowana jest przez Q-tablicę, a w DQN przez tablicę wag sieci. Widocze są diametralne różnice.

Podejście	Q-learning	Deep Q-Networks
Rozmiar na dysku (w KB)	116 915	1 480
Rozmiar w pamięci RAM (w KB)	81 920	0.14

Problemy z którymi się spotkałem i postanowiłem uwzględnić je jako wady tego podejścia to między innymi bardzo wysoki próg wejścia dla osoby będącej początkującą w dziedzinie uczenia maszynowego, a nawet czasami, przez ilość wszelkich parametrów i poziomu skomplikowania niektórych operacji, może wydać się problematyczna dla osoby zaznajomionej z tematem. Pomijając jednak ten fakt, większą wadą, będącą wręcz oczywistą w stwierdzeniu, jest czas wymagany podczas uczenia struktury rozwiązującej problem. Posiadając dostatecznie dużo czasu jesteśmy w stanie bardzo dobrze nauczyć sieć, ale mogą zdarzyć się przypadki, że z początku proces wygląda bardzo obiecująco, a po czasie mogą wyjść na jaw problemy z którymi, dla przykładu, sieć o niewystarczająco dużej ilości perceptronów lub warstw nie jest sobie w stanie poradzić, tym samym tracąc czas poświęcony już dla danego podejścia. By zobrazować pewien punkt odniesienia, próby analizowane w poprzednim podrozdziale składające się z 15 tysięcy epizodów średnio wymagały 14 godzin i 15 minut.

6 Eksperymenty

Mając już pewien punkt zaczepienia w postaci danych statystycznych z wielu sesji uczących w obu podejściach, postaram się w tym dziale pokazać wszelkie dokonane przeze mnie eksperymenty na środowisku. Celem będzie pokazanie zakresu możliwości optymalizacyjnych zaproponowanych już rozwiązań oraz testowanie potencjału zastosowanego algorytmu.

6.1 Algorytm Q-learning

Ze względu na znane już spore zapotrzebowanie sprzętowe w algorytmie Q-learning, ten eksperyment skupiał się będzie na optymalizacji stanu środowiska stosowanego w algorytmie, z możliwie jak najmniejszą stratą wydajności struktury. Pierwszy aspekt który przeanalizujemy, to pytanie jakiej jakości jest struktura obecnie zaimplemetowanego stanu. Stawiając się na miejscu agenta jesteśmy w stanie wydedukować na ile dana informacja jest dla niego przydatna. O ile jego własna pozycja (Pac-Man'a) na planszy wydaje się absolutnie kluczową składową stanu, tak trzeba zastanowić się, czy koniecznym jest zastosowanie aż tak precyzyjnych wartości pozycji dla wszystkich jego przeciwników. Wiadomo, że do pewnego stopnia konkretne pozycje przeciwników bardzo przydają się agentowi do znajdowania bardziej skomplikowanych ścieżek unikania ich, sprawdźmy jednak jak wpłynie na jego wydajność zastosowanie dużo mniej obciążającego stan podejścia. Plan jest taki, by przekazywać agentowi informacje o przeciwniku tylko wtedy, gdy jest dostatecznie blisko, tak by Pac-Man mógł na czas sobie z nim poradzić. W programie zaimplementowałem funkcję `enemy_aura` zawierającą w sobie dwa parametry: `aura_density`, który odpowiada temu, poniżej jakiej odległości od Pac-Man'a należy zwracać obecność przeciwnika oraz parametr `basic`, który jest wartością składowej stanu, jeśli wokół Pac-Man'a na obszarze mniejszym niż zadeklarowany nie ma przeciwnika (tak zaimplementowałem omawiany w poprzednim zdaniu brak konieczności podawania informacji). Ustawiając parametr gęstości aury na wartość 2 stan związany z lokalizacją przeciwnika zmniejszamy z 81 (9x9 pól) do 25 (5x5 pól) możliwości (dla omawianej na przestrzeni całej tej pracy planszy), tak też zróbmy, odpowiednio modyfikując funkcję generującą Q-tablicę, gdzie teraz zamiast generowania stanów po wszystkich polach na planszy, wygenerujemy tylko zakres liczbowy od -2 do 2 (tak wskazuje zmienna `aura_density`). Kod z metody `create_q_table` zapętlający wartości ze składowej (xd, yd):

```
1 for xd in range(0, size_x):
2   for yd in range(0, size_y):
```

teraz będzie wyglądał:

```
1 for xd in range(-2, 3):
2   for yd in range(-2, 3):
```

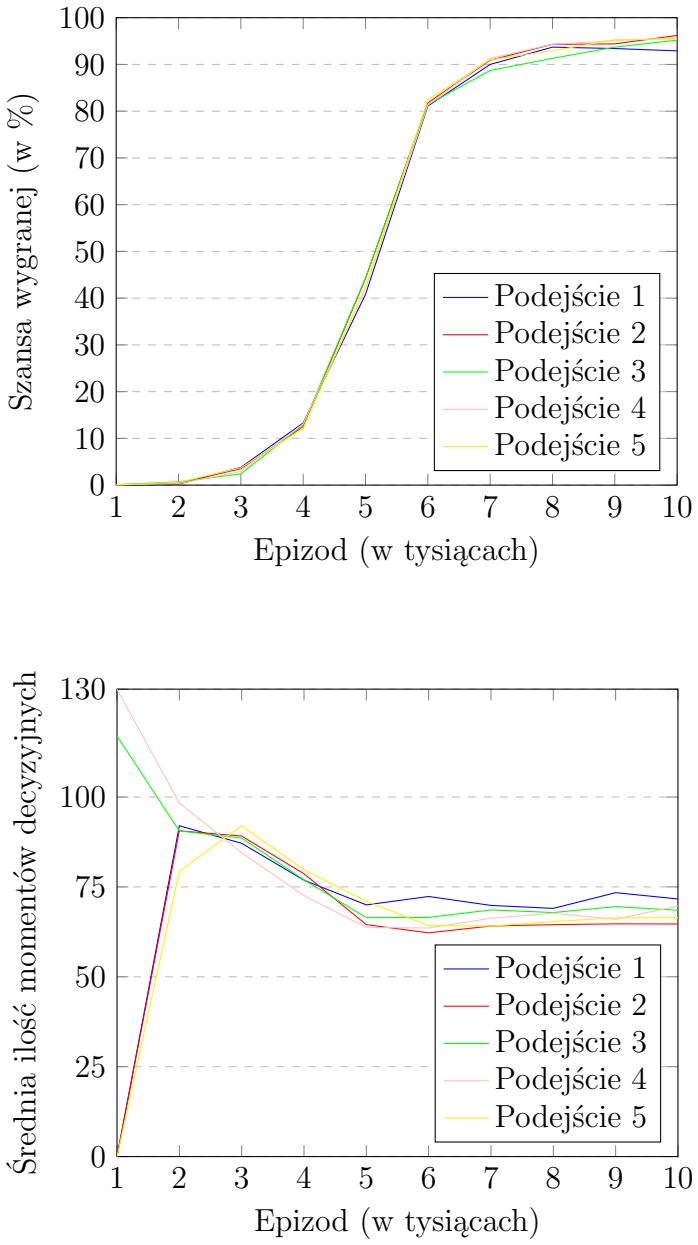
A każde pobranie składowej `g` ze skonkretyzowaną informacją:

```
1 g = (self.ghosts[0].x, self.ghosts[0].y)
```

teraz wykorzystuje metodę `enemy_aura`:

```
1 g = self.enemy_aura()
```

W celu otrzymania jak najwierniejszego porównania nowo zastosowanej metody, konfiguracja z jaką uruchomiony zostanie następujący proces uczenia pozostanie niezmieniona (identyczna jak w przypadku przedstawionego podejścia z działu o Q-learning). Wyniki z pięciu następujących po sobie procesów uczenia wyglądają następująco:



Rysunek 12: Wykresy przedstawiające statystki z 5 podejść dla algorytmu Q-learning z zoptymalizowaną składową stanu związaną z położeniem przeciwnika (źródło własne)

Pierwsze co widzimy na wykresach to równomierne rozłożenie wartości. Każdy z procesów zwraca niemalże identyczne wyniki pod względem procentowej szansy na wygranie oraz, co zaskakujące, średnia ilość momentów decyzyjnych potrzebnych do zwycięstwa

spadła przy wykorzystaniu metody `enemy_aura` w stosunku do stanu zawierającego konkretne położenie przeciwnika. Tym samym zoptymalizowano każdy pojedyczny stan (zapotrzebowanie sprzętowe) oraz polepszono wydajność samego algorytmu.

Pójdźmy o krok dalej i spróbujmy zastosować podobne pomniejszenie stanu, tym razem tego odpowiedzialnego za pozycję najbliższego punktu, który, jeśli następny krok się powiedzie, zostanie zmniejszony z 324 możliwości do tylko 9. Propozycja nowego stanu opierać się będzie nie na drodze do najbliższego punktu jak dotychczas, a po prostu na kierunku do niego. W tym celu, zmodyfikujemy metodę `get_way_to_closest_food` dodając zaraz przed operacją zwrócenia kod "ucinający" nadmiarowe wartości ponad 1 oraz poniżej -1 oraz zwrócenie tylko tych wartości.

```
way = self.pacman.way_to(food[0], food[1])
directon_x, directon_y = way
if directon_y >= 1:
  directon_y = 1
elif directon_y <= -1:
  directon_y = -1

if directon_x >= 1:
  directon_x = 1
elif directon_x <= -1:
  directon_x = -1

return directon_x, directon_y
```

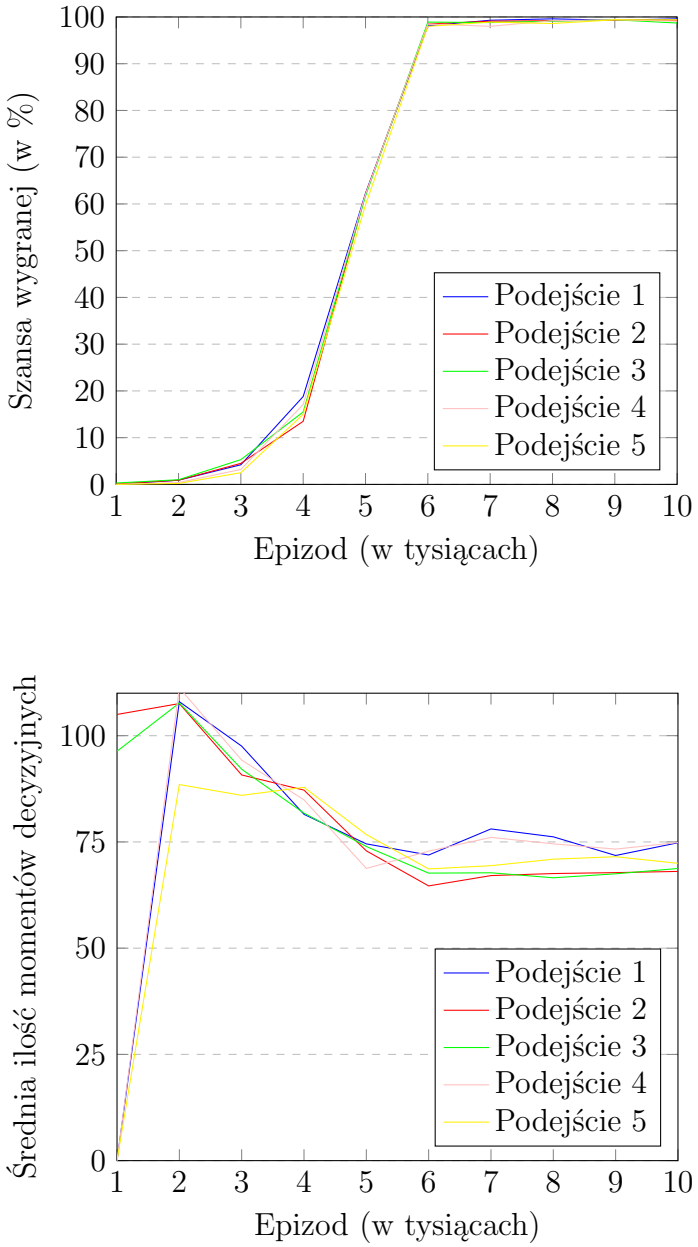
Inicjalizacja Q-tablicy pozbywa się w tym momencie swojej ogromnej części:

```
for xp in range(-size_x + 1, size_x):
  for yp in range(-size_y + 1, size_y):
```

Zastąpionej przez bardzo skromną składową stanu:

```
for xp in range(-1, 2):
  for yp in range(-1, 2):
```

Dalsze zmiany nie są wymagane, zastosowano tę samą konfigurację, przejdźmy od razu do wyników.



Rysunek 13: Wykresy przedstawiające statystki z 5 podejść dla algorytmu Q-learning z zoptymalizowaną składową stanu związaną z położeniem przeciwnika oraz punktu(źródło własne)

Wygląda na to, że ta zmiana również wyszła na dobre. Nie dość, że odchudziliśmy stan, który teraz posiada miejsce na dalszy rozwój, być może pod względem ilości przeciwników, to na wykresie z szansą wygranej widzimy znaczny skok wydajnościowy. Agent zaraz po upuszczeniu losowych akcji wybieranych przez algorytm ϵ-zachłanny wskakuje na wyżyny swoich możliwości osiągając od razu wartości około 99% szansy na wygraną, kosztem delikatnego podwyższenia w średniej ilości wykonywanych momentów decyzyjnych. Osobiście uznaję to za spory sukces.

6.2 Deep Q-Networks

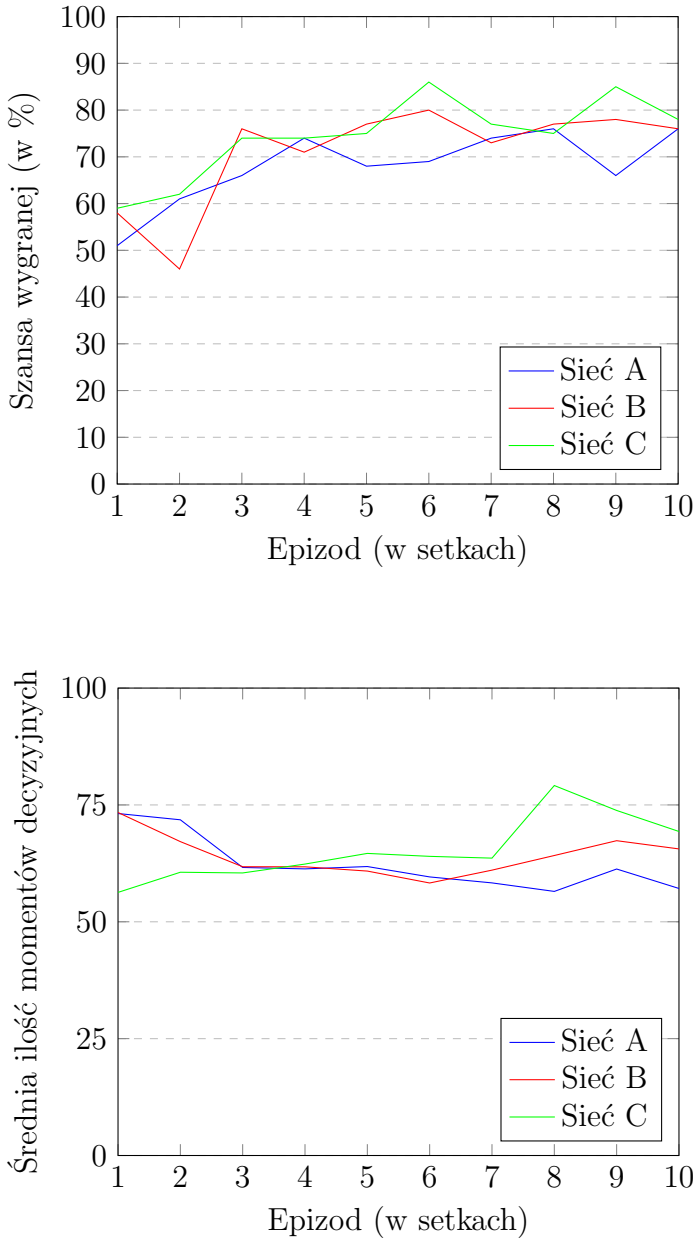
Swoje eksperymenty związane z DQN zawrę w serii testów trzech nauczonych sieci splotowych, każda z nich osiąga wyniki na poziomie około 97.5% szansy na zwycięstwo na klasycznej dla tej pracy planszy. Pierwszy test sprawdzi jakie wyniki osiągną sieci, które są nauczone grać przeciw jednemu przeciwnikowi, uruchomione w środowisku z dwoma przeciwnikami oraz jak szybko (w jaką liczbę epizodów) sieć jest w stanie się dostosować do nowej sytuacji oraz jaki osiągnie rekordowy wynik. Dodanie kolejnego przeciwnika wymaga tylko modyfikacji pliku zawierającego tekstową reprezentację planszy, a dodany on zostanie w dolnym lewym rogu planszy. Najpierw uruchomię środowisko dla tysiąca epizodów, z początkową wartością epsilon równą 1 i nieosiągalnym parametrem `FIT_EVERY` o wartości 999999999999 (celem wyłączenia procesu uczenia), by jak najbardziej rzetelnie zweryfikować wyniki. Przedstawiam tabelę dla trzech sieci, w której wyniki są postaci szansa wygranej/średnia liczba momentów decyzyjnych:

Nazwa sieci	A	B	C
Wyniki bazowe	97.6%/50.56	97.5%/50.47	97.8%/51.95
Wyniki dla dwóch przeciwników	75.3%/57.22	53.8%/52.66	68.4%/55.52

Jak widać, wyniki są w miarę satysfakcjonujące. Wprowadzane dane różnią się diametralnie pod względem wymagań dotyczących mechanik unikania przeciwników. W przypadku jednego, wystarczy po prostu wykonywać akcje oddalające agenta od przeciwnika (uciekać od niego), w przypadku dwóch wymagane jest przyszłościowe planowanie i bardzo dobra znajomość planszy, by nie dać się zapędzić w sytuacje bez odwrotu. Ze sposobu gry agenta, widocznej w reprezentacji graficznej, można wywnioskować, że podlega on pewnemu wyuczonemu mechanizmowi priorytetyzowania przeciwników w zależności od tego, który bardziej mu zagraża. Brany pod uwagę jest duch, z którym Pac-Man najprawdopodobniej skrzyżuje lub może skrzyżować swoje drogi, gdyż sieć również bierze pod uwagę losowość w wykonywanych przez przeciwników akcjach i stara się jak najbardziej zminimalizować możliwość zakleszczenia siebie pomiędzy nimi. Takie zachowania świadczą o bardzo dobrym zrozumieniu środowiska przez sieć. Co ciekawe, wyniki procentowe przedstawione w powyższej tabeli nie są kompletnie losowymi wartościami, jakby mogłoby się wydawać, ze względu na niemałe rozbieżności pomiędzy sieciami. Wyniki te zawierają się w granicach +- 5% szansy w obrębie pojedynczej sieci, lecz nadal są ściśle związane ze średnią ilością momentów decyzyjnych. Mniejsza osiągana ilość momentów decyzyjnych oznacza niestety mniejszą szansę na odniesienie zwycięstwa.

Po takim teście, uruchomię teraz z powrotem uczenie i sprawdzę jak długo zajmie sieciom ponowne osiągnięcie wyników na satysfakcjonującym poziomie. Słowo klucz "satysfakcjonyjącym" użyte jest ze względu na bardzo małą szansę powrócenia do wydajności z przed zmiany ilości przeciwników, ich obecność związana jest również z istnienem sytuacji bez wyjścia, których - wnioskując ze struktury planszy - nie będzie mało. Ponowne uczenie uruchomione zostało na konfiguracji z działu o Splotowej sieci neuronowej z jedną różnicą, początkowa wartość epsilonu od razu ustawiana jest na wartość maksymalną (1), z racji braku zapotrzebowania algorytmu na eksplorację,

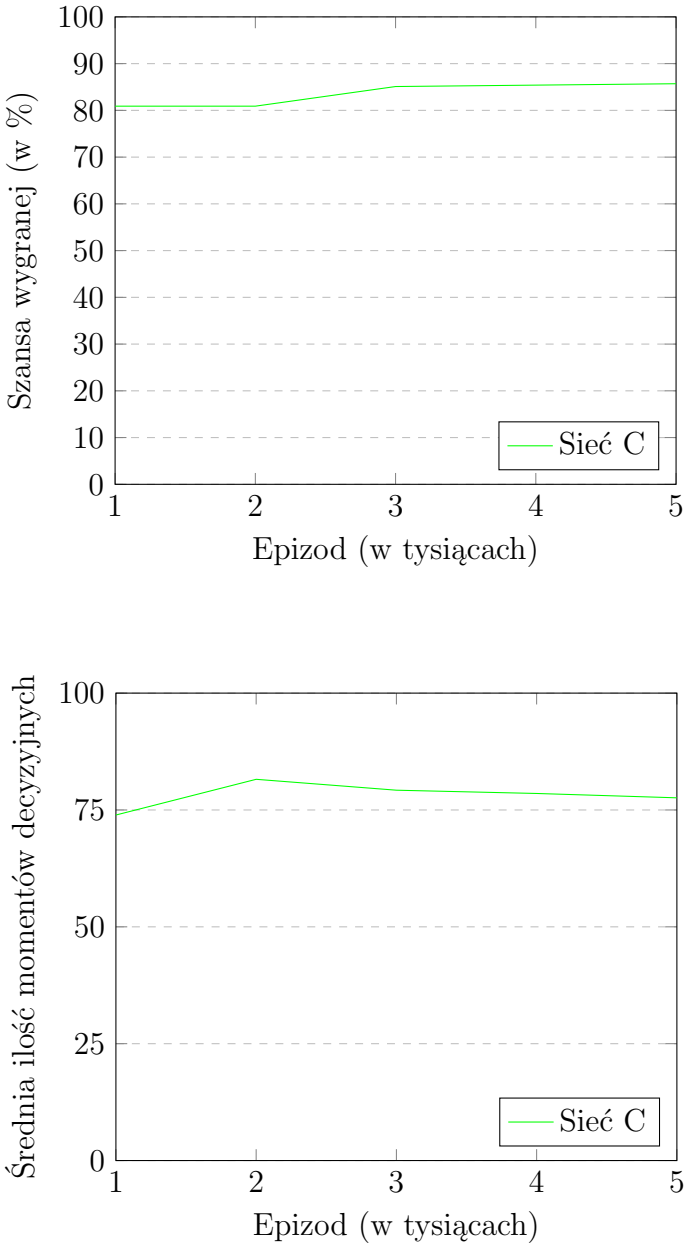
gdyż większość środowiska jest sieci już bardzo dobrze znana. Wydajność procesu ponownego uczenia przedstawiam na wykresach:



Rysunek 14: Wykresy przedstawiające statystki ponownego uczenia trzech splotowych sieci neuronowych pracy na dwóch przeciwnikach, początkowo nauczonych na jednym (źródło własne)

Uruchomienie ponownego uczenia łączy się ze spodziewanym efektem początkowego spadku w wydajności. Już po 100 epizodach uczących sieć widząc, że gdzieś zaszła zmiana stara się skorygować swoje decyzje do nowych sytuacji, co sprawia, że nieraz podejmie gorszą decyzję, niż w przypadku sieci "nieprawidłowej" (nie przystosowanej

do dwóch przeciwników). W ciągu kolejnych epizodów widoczna jest poprawa. Sieć B, która bez uczenia osiągała wyniki oscylujące koło 55% prawie natychmiastowo zaczęła nadganiać wydajnościowo pozostałe sieci. Osobiście myślę, że sieć C obecnie jest na najlepszej drodze do szybkiego osiągnęcia wysokiej szansy na wygraną. Wydaje mi się, że poświęcenie kilku momentów decyzyjnych na rzecz wyników na poziomie 85%, co jak widać po wykresie obecnie ta sieć uskutecznia, jest bardzo dobrym posunięciem, dlatego też, spróbuję przedłużyć uczenie sieci C do 5 tysięcy epizodów i sprawdzić, jak wtedy sobie poradzi. Wyniki kontynuacji uczenia sieci C wyglądają tak:



Rysunek 15: Wykres przedstawiający statystyki z wydłużonego procesu uczenia splotowej sieci neuronowej o kolejne 5 tysięcy epizodów (źródło własne)

Zależność związana z korelacją średnich momentów decyzyjnych do szansy wygranej wypunktowana przeze mnie zdanie temu jest widoczna gołym okiem. Wraz ze wzrostem momentów decyzyjnych w dwutysięcznym epizodzie rośnie powoli szansa na zwycięstwo. Jak widać po krzywej uczenia rysowanej na pierwszym wykresie agentowi ciężko osiągnąć coś więcej w tej materii. Szansa na wygraną zbiega do 86% i uznam ten wynik za okolicę rekordowego, możliwego do osiągnięcia.

7 Podsumowanie

Wybór algorytmu rozwiązujący problemy decyzyjne jest ściśle związany z tym co tak naprawdę chcemy osiągnąć oraz jakimi narzędziami dysponujemy. Stawiając przed sobą problem mniej złożony, posiadając dobry sprzęt lecz będąc ograniczonym czasowo zdecydowanie lepszą próbą będzie wykorzystanie algorytmu Q-learning. Nieuniknione zapewne będą eksperymenty związane z oceną jakości składowych stanu, by zrównoważyć zapotrzebowanie sprzętowe algorytmu do jego osiągów, jednak proces uczenia jest na tyle szybki, że popełnianie pomyłek w szacowaniach nie jest aż tak uciążliwe.

Natomiast, jeśli czas nie wchodzi w grę lub wymagania sprzętowe algorytmu Q-learning przerastają możliwości urządzenia na którym mamy zamiar operować, podejście DQN jest bardziej rozsądne. Opierając swoją implementację o DQN łączymy korzyści pochodzące z algorytmu Q-learning (kojarzenie do jakich stanów środowiska mogą prowadzić podejmowane obecnie akcje) wraz z korzyściami sieci neuronowych [19]. Wartości wag potrafią generalizować problem pozwalając wykorzystywać je nawet w nieco zmienionym środowisku (patrz eksperyment związany z DQN). Wykorzystywana w tej pracy biblioteka `TensorFlow` jest projektem bardzo dobrze wspieranym. Zawiera wiele implementacji pozwalających na uruchamianie nauczonych już modeli na innych urządzeniach, na przykład na smartfonach. Biblioteka nie jest ograniczona tylko do języka Python, swoją implementację posiada również język JavaScript [18]. Dlatego też wyniki tego podejścia nie są zamknięte w moim programie. Można bardzo łatwo (między innymi z racji małego rozmiaru struktury) wyeksportować model i wykorzystać go do podobnych środowisk na innych urządzeniach z lepszym lub gorszym skutkiem.

7.1 Dalsza praca

To czego nie udało mi się zaimplementować, a wydaje mi się na tyle ciekawym tematem by o nim wspomnieć, to próba wierniejszego odwzorowania mechaniki poruszania się przeciwników. W oryginale mechanika ta była oparta o wyszukiwanie drogi posiadającej największe prawdopodobieństwo złapania Pac-Man'a. W dodatku, jeśli poziom zawierał więcej niż jednego przeciwnika, ich algorytmy były zaimplementowane w taki sposób by jak najlepiej ze sobą współpracowały oraz dopełniały swoje wady, co sprawiłoby agentowi nieco większe problemy ale być może zwracałby on też dużo ciekawsze wyniki. Następnym krokiem mogłoby być wykorzystanie kolejnych sieci neuronowych uczonych podejmować decyzje za każdego z przeciwników Pac-Man'a, zwiększając po czasie poziom trudności gry jeszcze bardziej i sprawdzając czy przeciwnicy zaczną współpracować i w konsekwencji możliwie znajdą taki sposób, by uniemożliwić Pac-Man'owi jakiekolwiek zwycięstwo.

7.2 Oprogramowanie / Biblioteki

Poniżej wypisałem całe oprogramowanie oraz biblioteki wymagane do uruchomienia środowiska wraz z wersjami na których osobiście pracowałem.

- **Python**: 3.8.6
  - math
  - random
  - copy
  - time
  - pickle
  - collections
- **numpy**: 1.19.4
- **pygame**: 2.0.0
- **tensorflow**: 2.4.0
  - **keras**: 2.3.1

8 Bibliografia

Literatura

[1] A. M. Turing. I.-COMPUTING MACHINERY AND INTELLIGENCE. LIX(236):433–460.

[2] Yagang Zhang. *New Advances in Machine Learning*.

[3] Problem decyzyjny, *Wikipedia*. Page Version ID: 39671027.

[4] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning, second edition: An Introduction*. MIT Press.

[5] Pac-man, *Wikipedia*. Page Version ID: 987466660.

[6] Pac-man, https://pacman.com.

[7] OpenAI, http://gym.openai.com/.

[8] NeymarL. github.com/neymarl/pacman-rl.

[9] PyGame, https://www.pygame.org.

[10] Christopher I.C.H. Watkins. Technical note: Q-learning.

[11] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks.

[12] George Cevora. The relationship between biological and artificial intelligence.

[13] Jan Jantzen. Introduction to perceptron networks.

[14] Maja Czokòw, Jarosław Piersa, and Tomasz Schreiber. Wstęp do sieci neuronowych.

[15] A. Kwasigroch and M. Grochowski. Rozpoznawanie obiektòw przez głębokie sieci neuronowe. Nr 60.

[16] Andrzej Rutkowski. Szczegòły implementacyjne splotowych sieci neuronowych.

[17] Tianyi Liu, Shuangsang Fang, Yuehui Zhao, Peng Wang, and Jun Zhang. Implementation of training convolutional neural networks. page 10.

[18] TensorFlow, https://www.tensorflow.org.

[19] Harrison Kinsley. Deep q learning and deep q networks (DQN) intro and agent.

[20] Jianqing Fan, Zhaoran Wang, Yuchen Xie, and Zhuoran Yang. A theoretical analysis of deep q-learning.

[21] Sagar Sharma. Activation functions in neural networks.

[22] Jason Brownlee. A gentle introduction to the rectified linear unit (ReLU).

[23] Prince Grover. 5 regression loss functions all machine learners should know.

[24] Dami Choi, Christopher J. Shallue, Zachary Nado, Jaehoon Lee, Chris J. Maddison, and George E. Dahl. On empirical comparisons of optimizers for deep learning.