Politechnika Wrocławska

Wydział Elektroniki, Fotoniki i Mikrosystemów

KIERUNEK: Automatyka i Robotyka (AIR)

PRACA DYPLOMOWA MAGISTERSKA

TYTUŁ PRACY:

Porównanie Q-learningu i Deep Q-learningu na przykładzie gry komputerowej „Wąż”

AUTOR:

Michał Kaniowski

PROMOTOR:

dr inż. Wojciech Domski

WROCŁAW 2023

Streszczenie

Celem niniejszej pracy było porównanie dwóch algorytmów uczenia przez wzmocnienie tj. Q-learningu i deep Q-learningu. Zostały one przetestowane w grze komputerowej „Wąż". Na początku pracy opisano środowisko gry oraz wektor stanu pobierany przez agentów. Cały projekt był pisany przy użyciu języka Python. Opisano działanie obu algorytmów. W algorytmie Q-learningu zbadano wpływ współczynników α i γ występujących w równaniu Bellmana na skuteczność działań agenta. W deep Q-learningu sprawdzono wpływ liczby warstw ukrytych oraz ich rozmiaru w głębokiej sieci neuronowej na działanie agenta. Dodatkowo przetestowano dwie metody poprawiania osiągnięć sieci neuronowych tj. batch normalization oraz dropout. Na końcu pracy porównano wyniki uzyskane przez wytworzonych agentów. Na ich podstawie stwierdzono, że najlepsze wyniki osiągnął agent bazujący na deep Q-learningu.

Summary

The purpose of this study was to compare two algorithms for reinforcement learning, i.e. Q-learning and deep Q-learning. They were tested in the computer game "Snake". At the beginning of the work, the game environment and the state vector used by the agents were described. The entire project was written using the Python language. The operation of both algorithms was described. In Q-learning, the effect of the α and γ coefficients present in the Bellman equation on the agent's effectiveness was examined. In deep Q-learning, the effect of the number and size of the hidden layer in a deep neural network on agent performance was tested. In addition, two methods of improving the performance of neural networks were tested, i.e. batch normalization and dropout. At the end of the thesis, the results obtained by all agents were compared. Based on them, it was found that the agent based on deep Q-learning achieved the best results.

Słowa kluczowe: uczenie maszynowe, uczenie przez wzmocnienie, Q-learning, deep Q-learning, gra komputerowa

Keywords: machine learning, reinforcement learning, Q-learning, deep Q-learning, computer game

Spis treści

Rozdział 1

Wstęp

W 1637 roku francuski matematyk i filozof René Descartes powiedział, że nadejdzie taki dzień, w którym maszyny zaczną same podejmować decyzje w sposób inteligentny. Stwierdzenie to okazało się nie być aż tak nierealne, patrząc na rozwój technologi w ostatnich dziesięcioleciach. Może nie posiadamy tak zaawansowanych maszyn jak pokazanych w filmach tj. *Star Wars* czy *I, robot*, ale niewątpliwie ludzkość jest na drodze do osiągnięcia takiego poziomu. Po ponad trzech wiekach, bo w 1956 roku amerykański informatyk John McCarthy jako pierwszy użył terminu sztuczna inteligencja (z ang. *Artificial Intelligence*). Uznaje się to za jej początek. Samo pojęcie posiada kilka definicji z czego jedną z lepszych jest ta, która mówi, że sztuczna inteligencja odnosi się do komputerów, które starają się naśladować, niektóre własności ludzkiego umysłu jak np. uczenie się na błędach [9]. Pierwsze 17 lat badań nad tą dziedziną często określa się jako lato sztucznej inteligencji. Rozpoczęto wtedy kilka projektów z czego jeden osiągnął największy sukces. Był to program ELIZA stworzony przez Josepha Weizenbauma z Massachusetts Institute of Technology (MIT). Był to algorytm, z którym komunikacja przeprowadzana była już nie przez tekst na ekranie, tylko w sposób naturalny posługując się mową. ELIZA jest prekursorem inteligentnych asystentów takich jak Siri czy Alexa. Niestety, następne 10 lat nie były już tak efektywne pod względem rozwoju, ponieważ naukowcy musieli rozwiązać problem związany z mocą obliczeniową komputerów. Finansowania powoli się kończyły, a wszystkie prace i badania zaczeły hamować. Dopiero w latach 90. zeszłego wieku sztuczna inteligencja znowu zaczęła się rozwijać za sprawą rozwoju komputerów oraz przez jedno z przełomowych badań firmy IBM, która zamiast stosowania sztywnych zasad zaproponowała wykorzystanie prawdopodobieństwa do maszynowego tłumaczenia języka francuskiego na angielski. W efekcie dzisiaj mamy m.in. dopracowane algorytmy uczenia maszynowego czy sieci neuronowe.

Zastosowania sztucznej inteligencji z biegiem czasu zaczęły być coraz szersze, zaczynając od zwykłych chatbotów, później przez sterowanie prostymi manipulatorami, aż po samochody autonomiczne i rozpoznawanie obrazów. Jednym z ciekawszych zastosowań było wykorzystanie jej do nauki gier. Na początku sztuczna inteligencja została wykorzystana do grania w gry koputerowe takie jak *Pong*, która polegała na przesuwaniu platformy w kierunku góra dół i odbijaniu piłki. Następnie zaczęto próbować nauczyć maszynę grać w bardziej złożone gry jak *Pac-Man* czy warcaby [11]. Dzisiaj mamy już silniki potrafiące grać w szachy czy *Go*. Kolejnym krokiem było wymyślenie pewnego rodzaju systemu oceny modeli sztucznej inteligencji. Dla SI, która miała werbalnie wchodzić w interakcję z ludzmi, w 1950 roku brytyjski matematyk Alan Turing opracował test. Polegał on na rozmowie. Z jednej strony siedzi sędzia i prowadzi rozmowę z dwiema stronami. Jedną z nich jest człowiek a drugą SI. Jeżeli po rozmowie sędzia nie będzie w stanie określić,

która strona jest maszyną, to w tym przypadku SI przechodzi test. Najlepszy wynik w tym teście uzyskał bot o nazwie *Eugene Gootsman*, który przekonał około 1/3 sędziów. Dla algorytmów zastosowanych w grach system oceniania jest prostszy, ponieważ bazuje on przede wszystkim na osiągniętym finalnym wyniku. Dla gier, w których gra się przeciwko innej osobie, ostatecznym testem okazywały się potyczki z najlepszymi graczami na świecie. Jedną z najbardziej znanych potyczek jest mecz szachowy między ówczesnym mistrzem świata Garrim Kasparovem, a sztuczną inteligenją o nazwie Deep Blue. Ostatecznie maszyna wygrała 3,5 do 2,5 i jest to uznawane za moment przewyższenia komputera nad człowiekiem w szachach. Po tym wydarzeniu nie organizowano już takich spotkań, ponieważ nie miało to już sensu w tej dziedzinie, gdyż algorytmy stawały się coraz mocniejsze. Do niedawna ostatnim bastionem w grach planszowych była gra *Go*. Jest to starochińska gra, która dzięki temu, że jest bardzo skomplikowana, była nieosiągalna dla maszyny. Wszystko do momentu opracowania programu AlphaGo. Jest to algorytm oparty o sieci neuronowe i przeszukiwanie bazujące na metodzie Monte Carlo. W 2015 roku pierwsza maszyna wygrała pojedynek z profesjonalnym graczem Go, Fan Hui. Mecz zakończył sie wynikiem 5:0.

Jedną z najbardziej obiecujących i rozwijanych gałęzi sztucznej inteligencji w dzisiejszych czasach jest uczenie maszynowe (z ang. *machine learning*). Główną cechą algorytmów uczenia maszynowego jest to, że wraz z czasem ich działania poprawiają swoje wyniki. Samo uczenie maszynowe można podzielić na kilka głównych rodzajów: uczenie nadzorowane, uczenie nienadzorowane, uczenie częściowo nadzorowane i uczenie przez wzmocnienie. Pierwsze trzy podkategorie opierają się na podstawie bazy danych wejściowych oraz klucza odpowiedzi albo na samych danych wejściowych. Algorytmy uczą się wtedy wzorców. Wykorzystywane jest to w systemach detekcji obiektów czy systemach do rozpoznawania twarzy. W ostatnim rodzaju uczenia maszynowego, czyli uczeniu ze wzmocnieniem, algorytm ma wiedzę na temat praw działania środowiska w jakim się znajduje oraz dostaje nagrody i kary za podjęte decyzje. Na tej podstawie maszyna z czasem zaczyna podejmować lepsze decyzje w danym momencie. Ten rodzaj uczenia można zastosować do tworzenia algorytmów uczących się grania w gry komputerowe [4].

Głównym celem tej pracy jest porównanie dwóch metod uczenia przez wzmocnienie w środowisku gry komputerowej. W celu osiągnięcia postawionego celu stworzono dwóch agentów. Jeden oparty o tradycyjny Q-learning, drugi natomiast oparty o deep Q-learning. Przygotowano również środowisko, którego rolą była symulacja gry „Wąż". Finalnie przeprowadzono testy oraz porównano otrzymane rezultaty.

1.1 Teza

Podczas trenowania modelu uczenia przez wzmocnienie deep Q-learning osiąga wyższe wyniki w krótszym czasie w porównaniu do tradycyjnej metody Q-learningu.

Rozdział 2

Środowisko programistyczne

2.1 Wykorzystane oprogramowanie

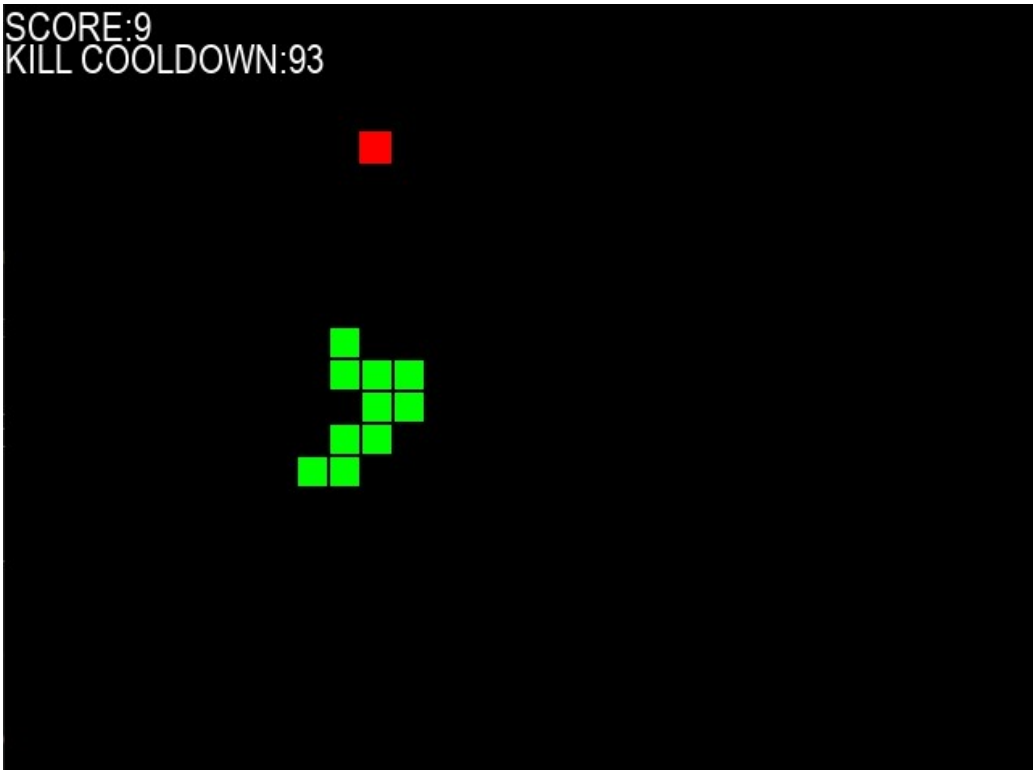
Wszystkie programy, czyli gra, agent bazujący na Q-learningu i agent bazujący na deep Q-learningu, zostały napisanę w języku Python. Dodatkowo posłużono się następującymi bibliotekami: numpy, json, random, pygame, Keras.

Biblioteka numpy jest narzędziem umożliwiającym sprawne i szybkie operacje na macierzach. W programach zostało wykorzystane m.in. do tworzenia i operacji na macierzy Q, tworzenia wektorów stanów, czy sprawdzania wektorów decyzji. Biblioteka json służy do sprawnego operowania na plikach typu JSON. W algorytmach posłużyła ona do zapisywania macierzy Q, wyników poszczególnych gier i ilości rozegranych gier. Moduł random jest narzędziem służącym do wybierania losowych liczb z dowolnych przedziałów i przy dowolnych rozkładach prawdopodobieństwa. Posłużył on do wprowadzania losowości podejmowanych decyzji. Biblioteka pygame jest to narzędzie do tworzenia gier/symulacji graficznych. Za jej pomocą wyświetlane jest okno gry zawierające planszę ze środowiskiem. Biblioteka Keras to jedna z bibliotek przeznaczonych do budowy sieci neuronowej. Posłużyła ona do zbudowania głębokiej sieci neuronowej. Na etapie programowania tego rozwiązania zastanawiano się również nad biblioteką PyTorch, jednak zważywszy na prostszą i czytelniejszą architekturę tworzonego modelu zdecydowano się na bibliotekę Keras. Jednakże biblioteka ta jest wolniejsza, ale w tym przypadku baza danych nie była na tyle rozbudowana, aby negatywnie wpływało to na przeprowadzone badania symulacyjne [2].

2.2 Środowisko gry

Zasady gry są takie same jak w oryginalnej wersji z 1976 roku. Na planszy podzielonej na pola porusza się wąż, może on przemieszczać się w 4 kierunkach: góra, dół, lewo i prawo. Nie może on natomiast wykonać ruchu w kierunku przeciwnym do tego, w którym aktualnie się porusza. Pierwszy, przedni element węża jest nazywany głową, a pozostałe elementy ciałem. Ciało podąża śladem głowy. Na mapie znajduje się jedno specjalne pole, na którym umieszczony jest dodatkowy punkt (w orginalnej wersji znajdował się tam owoc). Kiedy głowa węża znajdzie się na tym polu zdobywa punkt i jego ciało wydłuża się o jedno pole. Gra może się zakończyć na dwa sposoby, gdy głowa uderzy w ścianę planszy lub gdy głowa uderzy w ciało węża. Wprowadzono również modyfikację, która to przerywa grę, jeżeli wąż przez określoną ilość ruchów nie zdobędzie punktu. Po zdobyciu punktu licznik ruchów się zeruje. Zdecydowano się na takie rozwiązanie, ponieważ przyjęto, że agent nie będzie karany za zmianę pola bez zdobycia punktu lub kolizji.

Okno gry składa się z planszy podzielonej na kwadraty. Wymiary planszy to 32 pola w poziomie i 24 pola w pionie. Starano się utrzymać przejrzystość gry. Tło ekranu jest czarne. Punkt dodatkowy podświetlany jest kolorem czerwonym, a wszystkie elementy węża podświetlane są kolorem zielonym. W głównej mierze mechanika wyświetlania gry oparta jest o jedną listę, która zawiera współrzędne wszystkich elementów węża. Z każdym ruchem do listy dodawane są nowe współrzędne głowy oraz usuwane są ostatnie współrzędne na tej liście, z wyjątkiem sytuacji gdy agent trafi na dodatkowy punkt. Elementy węża reprezentowane są kolorem zielonym. W celu wizualnego zwiększenia czytelności ruchów, elementy węża rysowane są bez wąskiej obramówki. Dodatkowo w lewej górnej części ekranu widać aktualny wynik oraz liczbę dostępnych ruchów. Na rysunku 2.1 przedstawiono przykładowy widok gry. Na początku widoczna jest tylko głowa węża, znajdująca się na środku ekranu skierowana w prawą stronę oraz dodatkowy punkt znajdujący się w losowym miejscu na mapie.



Rysunek 2.1 Przykładowy widok z gry

Rozdział 3

Uczenie maszynowe przez wzmocnienie

Jak wspomniano wcześniej uczenie maszynowe można podzielić na 4 główne kategorie: uczenie nadzorowane, uczenie nienadzorowane, uczenie częściowo nadzorowane i uczenie przez wzmacnianie. Czwarta grupa znacząco różni się od pozostałych.

Sama idea uczenia przez wzmacnianie wzięła się od tresowania zwierząt, dlatego można spotkać się z określeniem, że jest to metoda kija i marchewki. Podczas nauki psa wykonywania sztyuczek treser wymawia komendę, po której zwierzę powinno wykonać określoną akcję. Za każdą poprawną podjętą decyzję czworonóg otrzymuje nagrodę, za wykonanie innej czynności niż podanej w komendzie człowieka pies otrzymuje karę. Wiadomo, że na początku zwierzę będzie podejmować losowe akcje, ale z czasem zacznie zapamiętywać, że po wykonaniu pewnej akcji po usłyszeniu pewnego zwrotu otrzymuje nagrodę i będzie wykonywał ją coraz częściej [12]. Taka sytuacja określana jest jako wzmocnienie zachowań.

Główną różnicą między uczeniem przez wzmocnienie, a pozostałymi gałęziami uczenia maszynowego jest to, że w tej metodzie potrzebna jest tylko znajomość zasad panujących w środowisku oraz system nagród i kar. Nie potrzeba bazy danych z wcześniej przygotowanymi odpowiedziami, algorytm sam po pewnym czasie nauki stworzy swój własny klucz odpowiedzi do danych sytuacji. Można to porównać do nauki gry w koszykówkę. Na początku znamy zasady gry. Wraz z czasem przeznaczonym na trening zaczynamy zauważać jakie akcje przynoszą nam najlepsze rezultaty.

Przed zaczęciem opisywania już konkretnych rozwiązań należałoby wyjaśnić kilka podstawowych pojęć. Środowisko jest to miejsce, w którym agent działa i podejmuje decyzje. Może to być środowisko fizyczne, np. labirynt czy zwykły pokój oraz może to być symulowane środowisko komputerowe. Jest ono dynamiczne i reaguje na podjęte decyzje przez agenta. Ze środowiska pobierane są takie informacje jak aktualny stan, następny stan czy nagroda. Stan jest sytuacją w jakiej znajduje się aktualnie agent. Reprezentuje on istotne informacje do podejmowania akcji przez agenta. Opisywany najczęściej jest jako wektor cech, które opisują różne aspekty sytuacji, w której w danej chwili znajduje się agent. W rzeczywistości może to być zestaw danych zebranych z czujnika robota, np. odległość od przeszkód, położenie czy prędkość. W przypadku gier mogą to być informacje na temat położenia agenta, położenia innych ważnych elementów gry na mapie czy punktacja gracza lub graczy. Agent, czyli algorytm podejmujący decyzje na podstawie otrzymanego stanu w jakim się obecnie znajduje i wyników akcji podjętych wcześniej. Jego zadaniem jest uczenie się optymalnej polityki, czyli podejmowania najlepszych decyzji prowadzących do maksymalizowania nagród. Dodatkowo agent posiada funkcję, która pobiera ze środowiska informację o stanie i zmienia go w wektor stanu.

Agenci z każdym ruchem otrzymują od gry informacje zwrotną na temat nagrody, wyniku gry, tego czy gra się skończyła oraz pobierają opis stanu, w którym się znajdują

na planszy. Zdecydowano się na następującą prezentacje stanu. Jest on przedstawiony jako wektor 11 zmiennych w postaci 0 i 1. W jego skład wchodzą 3 sekcje. Pierwsza określa kierunek poruszania się węża. Wśród tych czterech elementów tylko jeden może być opisany jako 1, reszta musi mieć wartość 0. Druga sekcja określa, czy wokół głowy znajdują się zagrożenia. Jako niebezpieczeństwo zdefiniowano pola zajęte przez ciało lub znajdujące się poza polem gry. W tym przypadku dozwolona jest każda możliwa kombinacja 0 i 1. Ostatnia sekcja opisuje położenie głowy względem nagrody. W tej grupie pierwsze dwa elementy określają zależność horyzontalną położenia, a kolejne dwie zależność wertykalną położenia. W obu przypadkach niemożliwe jest osiągnięcie stanu 1 1, co oznaczałoby, że nagroda znajduje się jednocześnie po dwóch przeciwnych stronach względem głowy. Można było zastosować prostszą wersję wektora stanu, w którym zamiast osobno sprawdzać dwie przeciwne strony, sprawdzanoby tylko jedną stronę, np. zamiast sprawdzać czy nagroda znajduje się po lewej stronie i później sprawdzać, czy nagroda znajduje się po prawej stronie, można byłoby sprawdzić, czy nagroda znajduje się tylko po lewej stronie bądź na równi z głową i w takim przpadku przypisywać 1, a gdy ten warunek się nie spełni przypisać 0, co byłoby równoznaczne z tym, że nagroda znajduje się po prawej stronie. Zmniejszyłoby to rozmiar wektora stanu o 2 pozycje. Nie zdecydowano się na to z jednego powodu. W przypadku z czterema elementami może wystąpić sytuacja, w której na pozycji 8 i 9 lub 10 i 11 wystąpi dwa razy 0, co będzie równoznaczne z tym, że nagroda znajduje się na odpowiedniej szerokości lub wysokości względem głowy węża. Na rysunku 3.1 przedstawiono przykładowy wektor stanu pobierany przez agenta ze środowiska.

Rysunek 3.1 Przykład wektora stanu

Dokładna kolejność i znaczenie poszczególnych składowych wektora stanu przedstawiono poniżej.

1. Wąż skierowany jest w lewo.
2. Wąż skierowany jest w prawo.
3. Wąż skierowany jest do góry.
4. Wąż skierowany jest do dołu.
5. Na przeciwko głowy znajduje się zagrożenie.
6. Po prawej stronie od głowy znajduje się zagrożenie.
7. Po lewej stronie od głowy znajduje się zagrożenie.
8. Nagroda znajduje się na lewo od głowy.

9. Nagroda znajduje się na prawo od głowy.

10. Nagroda znajduje się nad głową.

11. Nagroda znajduje się pod głową.

3.1 Q-learning

Pierwszym algorytmem poddanym badaniom będzie Q-learning. Jest to jeden z częściej stosowanych algorytmów uczenia ze wzmacnianiem. Używany jest on w środowiskach dynamicznych do podejmowania najlepszych decyzji w danej sytuacji [5]. Może być on stosowany do sterowania robotami w świecie rzeczywistym lub środowisku symulacyjnym. Głównym zadaniem w tym przypadku jest znajdowanie optymalnej strategii nawigacji po mapie, unikanie przeszkód czy ewentualna manipulacja obiektami znajdującymi się w otoczeniu. Kolejnym zastosowaniem może być podejmowanie decyzji dotyczących zarządzaniem zasobami. Agent uczy się optymalnej polityki dystrybucji w celu zmniejszenia kosztów i zwiększenia wydajności. Odnosi się to do m.in. transportu czy logistyki. Innym zastosowaniem może być zarządzanie ruchem lotniczym, kolejowym czy miejskim. Głównym celem jest znajdowanie optymalnych rozwiązań rozkładania ruchu i minimalizacji zatorów. Dodatkowo może być wykorzystywany do gier komputerowych i uczenia się optymalnych strategi gry. Może byc stosowany do różnych typów gier m.in. strategicznych czy planszowych.

3.1.1 Macierz Q

Najważniejszym elementem tej metody jest macierz Q. Jest to macierz zawierająca spis wszystkich możliwych stanów oraz możliwych akcji, którym przypisana jest wartość Q. Q oznacza liczbową wartość podjętej decyzji w danej sytuacji. Poniżej przedstawiono przykładową macierz Q. Znajdują się w niej 5 różnych stanów i 3 akcje. Na początku wszystkie elementy są przyjmowane jako 0. Taka praktyka stosowana jest w większości przypadków, tak zrobiono też w tej pracy, ale istnieją inne sposoby wstępnego zapełniania macierzy Q, np. wartości losowe, *Artificial Potential Fields*, *Whale Optimization Algorithm*, *Flower Pollination Algorithm*. Z każdą podjętą decyzją odpowiednie komórki są aktualizowane. Im większa wartość tym dana akcja jest lepsza, analogicznie im mniejsza wartość tym dana akcja jest gorsza. Na podstawie przykładowej macierzy Q (3.1), w której kolumny reprezentują możlwe akcje, a wiersze oznaczają stany, w stanie 1 agent podjąłby akcję 2, ponieważ posiada ona największą wartość Q, bo aż 8.1. Analogicznie w stanie 2 podjąłby akcję 1, w stanie 3 podjąłby akcję 2 itd.

$$Q = \begin{bmatrix} 6 & 8.1 & -7.8 \\ 7 & 1 & 3.2 \\ -5 & 2.4 & 0.3 \\ -5.8 & 1.6 & 2.7 \\ -0.8 & 3.4 & 6 \end{bmatrix} \tag{3.1}$$

Z uwagi na 4 kierunki ruchu agenta, 8 możliwości przedstawienia zagrożeń wokół głowy agenta, 9 możliwości przedstawienia położenia punktu dodatkowego oraz 3 dopuszczalnych akcji, macierz Q będzie miała rozmiar 288 x 3, co daje 864 komórek.

3.1.2 Równanie Bellmana

Wartości w macierzy Q muszą w jakiś sposób być aktualizowane wraz z przebiegiem uczenia się algorytmu. Do tego służy równanie wymyślone przez Richarda Ernesta Bellmana w latach 50. ubiegłego wieku stosowane w programowaniu dynamicznym. Zostało ono przedstawione w (3.2):

$$Q_{k+1}(s_t, a_t) \leftarrow Q_k(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)), \tag{3.2}$$

gdzie Q_{k+1} oznacza nowa wartość Q, Q_k to stara wartość Q, s_t określa aktualny stan, a_t to aktualna akcja, r_t jest to nagroda z aktualnego stanu,natomiast α jest współczynnikiem uczenia, a γ współczynnikiem dyskontowania.

Wartości Q z (3.2) są pobierane z pamięci, czyli z macierzy Q. Nagroda natomiast jest zwracana przez środowisko. Współczynnik uczenia α jest parametrem kontrolującym wagę aktualizacji wartości Q. Przyjmuje on wartości z przedziału 0 - 1. Jego niższe wartości powodują, że Q zmienia się wolniej i agent bardziej opiera się na swoich wcześniejszych wyliczeniach. Z drugiej strony, gdy parametr ten jest bliski 1, to aktualne wydarzenia mają większą wagę i wartości Q zmieniają się szybko. Ostatnim parametrem równania Bellmana jest współczynnik dyskontowania. On również przyjmuje wartości od 0 do 1. Ma on wpływ na znaczenie przyszłych stanów i nagród w odniesieniu do bieżącej nagrody. Gdy jego wartość jest niska to większe znaczenie będą mieć aktualne nagrody, a gdy wartość tego parametru będzie bliższa 1, to przyszłe nagrody mają porównywalną wartość do tych aktualnych. Współczynniki α i γ ciężko określić na podstawie jakiejś zasady czy wzoru. W praktyce dobranie optymalnych wartości wymaga eksperymentów i zmieniania ich w trakcie uczenia agenta.

3.1.3 Eksploracja i eksploatacja

W kontekście Q-learningu początkowym etapem pracy agenta jest uczenie się, czyli zapełnianie i aktualizacja komórek pamięci. Na początku wartości w tych komórkach są równe zero bądź są wypełnione losowymi liczbami, więc nie można stwierdzić, aby na tym etapie agent podejmował optymalne decyzje. W tym celu wprowadzono balans pomiędzy eksploracją i eksploatacją. Eksploracja jest to etap, w którym agent podejmuje losowe akcje w celu poszukiwania i odkrywania nowych stanów. Podjęte decyzje niekoniecznie muszą być najlepsze, ale ich celem jest dotarcie do nieodkrytych przestrzeni stanów i akcji. Eksploracja jest szczególnie ważna na początku uczenia się algorytmu. Później z czasem zaczyna byc ważna eksploatacja. Jest to etap, w którym agent podejmuje decyzję na podstawie pamięci i wcześniejszych doświadczeń. W celu osiągnięcia najlepszych wyników agent powinien mieć odpowiednio dobrany balans między eksploracją i eksploatacją. W przypadku zbyt długiego eksplorowania agent może zmniejszać wyniki osiągane przez agenta. Z drugiej strony zbyt mała eksploracja może prowadzić do tego, że algorytm nie znajdzie najlepszej polityki i utknie w lokalnym optimum.

W tej pracy ten problem rozwiązano w następujący sposób. W pierwszej grze agent podejmuje decyzję w 100% losowo. Wraz z każdą kolejną grą ten procent spada, aż po osiągnięciu pewnej ilości rozegranych partii jego decyzje są w 100% opartę o pamięć agenta. Przykładowo jeżeli z każdą kolejną grą procent losowości decyzji spada o 1%, to po setnej grze agent działa tylko w oparciu o wcześniejsze doświadzenia.

3.1.4 Działanie algorytmu

Znając już podstawowe aspekty Q-learningu można przedstawić w pełni działanie tego algorytmu. Poniżej przedstawiono poszczególne kroki tej metody.

1. Inicjalizacja:
   - inicjalizacja macierzy Q, która przechowuje estymowane wartości Q dla każdego stanu-akcji, na początku wszystkie wartości przyjumują 0.
2. Główna pętla algorytmu. Dla każdej gry (epizodu) powtarzane są następujące kroki:
   - wybierz akcję na podstawie aktualnego stanu zgodnie z strategią eksploracji/eksploatacji (losowo lub w oparciu o macierz Q),
   - wykonaj wybraną akcję i obserwuj nagrodę oraz następny stan,
   - zaktualizuj wartość Q dla poprzedniego stanu i akcji na podstawie otrzymanej nagrody i najwyższej wartości Q dla następnego stanu,
   - jeżeli skończy się jedna gra lub epoka zmień strategię eksploracji/eksploatacji.
3. Powtarzaj krok 2 przez ustaloną liczbę gier czy epok lub do osiągnięcia pożądanego poziomu wydajności.

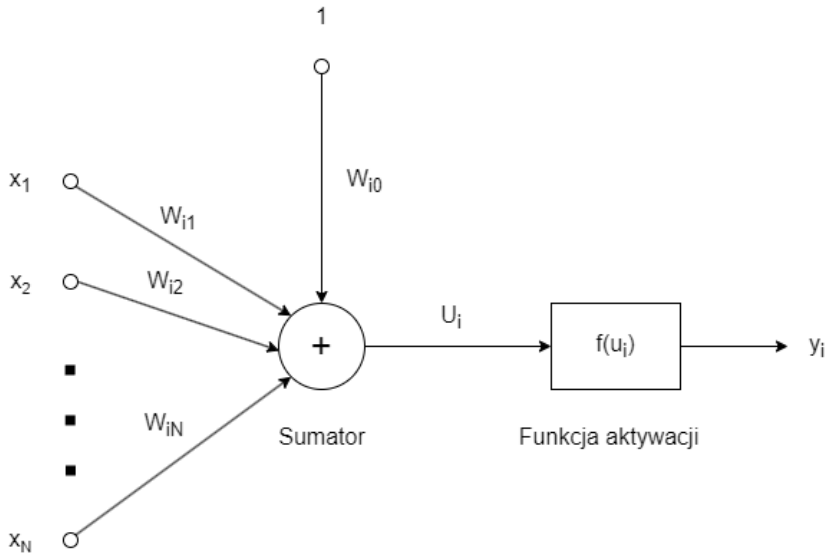
3.2 Deep Q-learning

Drugim testowanym algorytmem był deep Q-learning. Jest to pewnego rodzaju rozszerzenie do zwykłego Q-learningu, które do obliczenia wartości Q wykorzystuje głębokie sieci neuronowe. To podejście ma podobne zastosowania jak Q-learning, ale stosowane jest do większej przestrzeni stanów i akcji, ponieważ w pierwszym algorytmie ograniczeniem jest rozmiar macierzy Q. Wraz ze zwiększeniem przestrzeni stanów i akcji gwałtownie zwiększa się zapotrzebowanie na pamięć i moc obliczeniową [1]. W deep Q-learningu nie ma takiego problemu, ponieważ decyzje nie są podejmowane na podstawie pamięci, tylko na podstawie wyliczeń sieci neuronowej.

Algorytm działania deep Q-learningu jest podobny do zwykłego Q-learningu. Agent dostaje informacje o stanie, nagrodzie i karze od środowiska. Następnie sieć neuronowa na podstawie danego stanu estymuje wartości Q dla wszystkich możliwych ruchów w danym stanie.

Sieć z definicji jest to zbiór pewnych elementów, w tym przypadku sztucznych neuronów. Nie są to jednak elementy, które nawet w przybliżonym zakresie odpowiadają złożoności biologicznych neuronów, choć na pewnym poziomie abstrakcji posiadają one wspólne cechy. Pierwszy sztuczny neuron opracowali w 1943 roku dwaj amerykańscy naukowcy Warren Sturgis McCulloch i Walter Harry Pitts Jr [8]. Udało im się teoretycznie udowodnić, że każde obliczalne funkcję mogą być obliczane przez sieć połączonych ze sobą neuronów. Oczywiście na początku te koncepcje nie miały swojej interpretacji w postaci sprzętowej, na to trzeba było poczekać jeszcze kilkanaście lat. Model neuronu został przedstawiony na rysunku 3.2.

Na początku dane wejściowe (od X_1 do X_n) oddziaływują na neuron równocześnie, nie ma żadnych zależności czasowych miedzy stanami wcześniejszymi oraz poprzednie stany nie mają wpływu na stan aktualny. Wartości wejścia mnożone są przez odpowiadające



Rysunek 3.2 Schemat sztucznego neuronu [8]

im wagi i sumowane są w neuronie. Dodawana jest też waga w_0, nazywana progiem aktywacji. Następnie suma ważona jest przekazywana do funkcji aktywacji, która to określa ostateczną wartość liczbową jaka wyjdzie z neuronu.

Funkcje aktywacji mogą być różne, najczęściej stosuje się funkcje progowe/schodkowe, funkcje liniowe, funkcje sigmoidalne lub funkcje Gaussa [13]. Zostały one przedstawione na rysunku 3.3. Funkcja progowa przypisuje dwie wartości 0 lub 1 (można się też spotkać z wartościami -1 i 1). Najczęściej granicą dla wartości sumy ważonej jest 0, wszystkie wyniki poniżej tej wartości przyjmują 0 lub -1, a powyżej 1. Można tę funkcję przedstawić wzorem ogólnym (3.3), a jej wykres został przedstawiony na rysunku 3.3(a).

$$f(x) = \begin{cases} 0 & \text{dla } x < \text{a} \\ 1 & \text{dla } x \geqslant \text{a} \end{cases} \tag{3.3}$$

Funkcja liniowa (3.4) również jest często stosowana, ale zazwyczaj parametr b jest równy 0, więc funkcja liniowa przechodzi przez środek układu współrzędnych i tylko za pomocą parametru a można zmieniać jej nachylenie. Została ona przedstawiona na rysunku 3.3(b).

$$f(x) = ax + b, \tag{3.4}$$

Dodatkowo można modyfikować tę funkcję przez odcinanie jednostronne lub dwustronne, czyli przyjmowanie stałych wartości dla sum ważonych mniejszych lub większych od wcześniej ustalonych. Jedną z takich funkcji postanowiono użyć w budowie sieci neuronowej w tym projekcie. Jest to funkcja ReLU (z ang. *Rectified Liniear Unit*) (3.5). Funkcja ta zwraca wartości wejściowe, jeśli są dodatnie, albo 0 w przypadku ujemnych. Jej wykres został przedstawiony na rysunku 3.3(c).

$$f(n) = \begin{cases} 0 & \text{dla } x < \text{a} \\ x & \text{dla } x \geqslant \text{a} \end{cases} \tag{3.5}$$

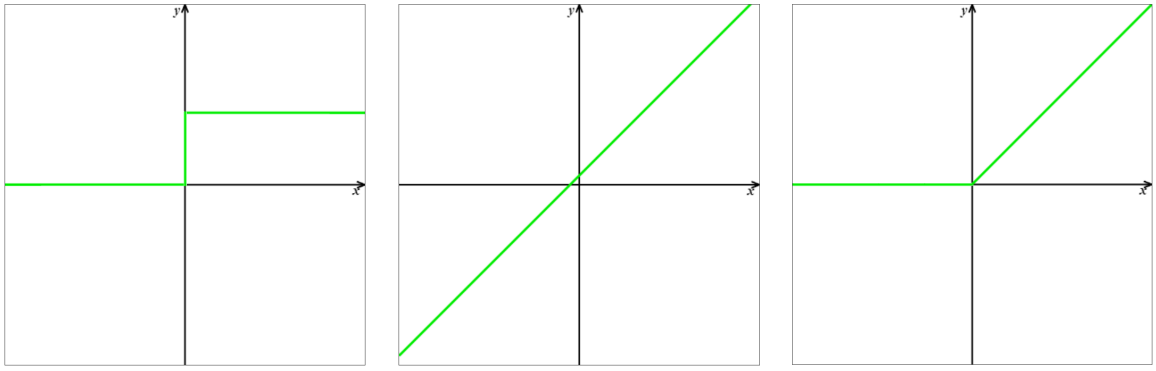
Funkcja sigmoidalna natomiast zamienia sumę ważoną na wartość z zakresu od 0 do 1. Ma ona postać (3.6). Jej wykres został przedstawiony na rysunku 3.3(d). Jest ona teraz

rzadziej używana ze względu na problemy związane ze znikającym gradientem (z ang. *vanishing gradient*).

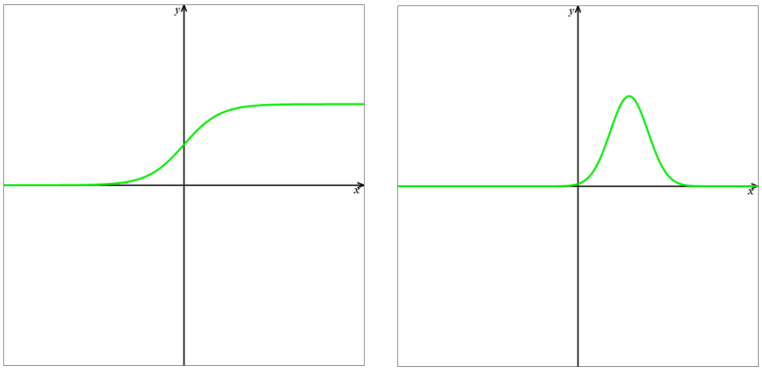
$$f(x) = \frac{1}{1 + e^{-x}}, \tag{3.6}$$

Funkcja Gaussa stosowana jest rzadko i w specyficznych sytuacjach, np. do aproksymacji funkcji, zwłaszcza w zadaniach regresji. Niestety może być ona kosztowna pod względem obliczeniowym. Ma ona postać (3.7) i jej wykres został przedstawiony na rysunku 3.3(e).

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), \tag{3.7}$$



(a) funkcja progowa/schodkowa (b) funkcja liniowa (c) funkcja ReLU



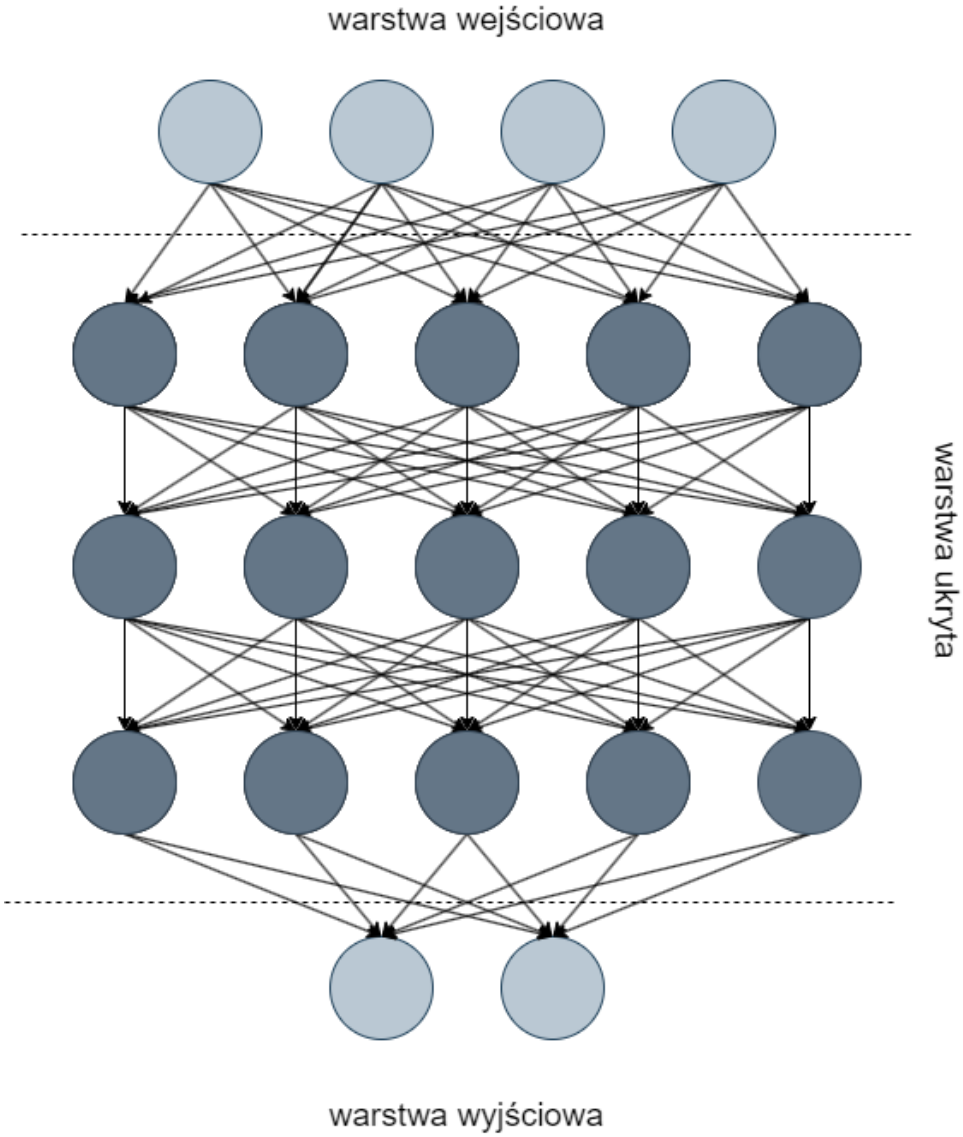
(d) funkcja sigmoidalna (e) funkcja Gaussa

Rysunek 3.3 Wykresy funkcji aktywacji

Jak sama nazwa wskazuje do deep Q-learningu należy użyć głębokiej sieci neuronowej. Zwykła sieć neuronowa posiada tylko 3 warstwy: wejściową, ukrytą i wyjściową. Głęboka sieć neuronowa musi posiadać wiele warstw ukrytych. Na początku badań nad głębokimi sieciami neuronowymi stosowano do kilkunastu warstw ukrytych. Dzisiaj występują już modele zawierające kilkadziesiąt a nawet kilkaset warstw. W danym przypadku warstwa wejściowa ma rozmiar 11 neuronów, a warstwa wyjściowa ma rozmiar 3 neuronów. Na rysunku 3.4 przedstawiono schemat przykładowej architektury głębokiej sieci neuronowej.

3.2.1 Równanie Bellmana

Równanie Bellmana jest ważnym narzędziem w deep Q-learningu. Jest używane do aktualizacji wartości akcji w procesie uczenia. Wzór ten jest uproszczoną wersją równania



Rysunek 3.4 Schemat architektury głębokiej sieci neuronowej

używanego w klasycznym Q-learningu. Został on przedstawiony w (3.8). W równaniu Bellmana wykorzystywanym w deep Q-learningu nie potrzebna jest wartość aktualnego stanu. Dodatkowo nie potrzebny jest współczynnik uczenia, ponieważ jest on wykorzystywany w optymalizatorze.

$$Q(s_t, a_t) = r_t + \gamma \cdot \max_a Q(s_{t+1}, a_{t+1}), \tag{3.8}$$

gdzie Q oznacza wartość funkcji Q dla stanu s i akcji a, s_t reprezentuje aktualny stan, a_t określa aktualną akcję, r_t jest nagrodą z aktualnego stanu, a γ jest współczynnik dyskontowania.

Równanie Bellmana mówi nam, że wartość funkcji Q dla pary stan-akcja jest równa sumie natychmiastowej nagrody i zdyskontowanej maksymalnej wartości funkcji Q dla następnego stanu. W procesie uczenia deep Q-learningu staramy się znaleźć optymalną funkcję Q, która spełnia to równanie dla wszystkich par stan-akcja. W praktyce, w tym algorytmie, używamy głębokiej sieci neuronowej do aproksymacji funkcji Q w danym stanie i stanie następnym. Następnie wybrana wartość dla danej akcji w danym stanie jest

wyliczana za pomocą wzoru (3.8). Na koniec aktualizowane są wagi modelu, aby minimalizować różnicę między wartością estymowaną przez głęboką sieć neuronową, a wartością wyliczoną na podstawie równania Bellmana, wykorzystując metody gradientowe, takie jak metoda spadku gradientowego.

3.2.2 Pamięć i rozmiar partii

W deep Q-learningu pamięć (z ang. *memory*) i rozmiar partii (z ang. *batch size*) są dwoma ważnymi elementami, które odgrywają istotną rolę w procesie uczenia. Pamięć, znana również jako pamięć doświadczenia lub bufor doświadczenia, jest strukturą danych, w której przechowuje się doświadczenia agenta. Głównym celem pamięci w deep Q-learningu jest przechowywanie i odtwarzanie uprzednio zebranych informacji, aby umożliwić agentowi skuteczniejsze uczenie się. Pamięć może być implementowana jako bufor FIFO (z ang. *First-In-First-Out*) lub jako bufor priorytetowy, w którym doświadczenia o większej wadze mają większe szanse na wybór. Podczas eksploracji środowiska agent wykonuje akcje i obserwuje stany oraz natychmiastowe nagrody. Te doświadczenia są zapisywane w pamięci w postaci osobnych list lub krotek. Ich zawartość to: stan, akcja, nagroda, następny stan. Przechowywanie doświadczeń w pamięci pozwala na ponowne wykorzystanie ich podczas procesu uczenia, poprawiając stabilność i efektywność uczenia się.

Rozmiar partii odnosi się do liczby doświadczeń, które są pobierane z pamięci i wykorzystywane do aktualizacji wag sieci neuronowej w jednym kroku uczenia. Zamiast aktualizować wagi na podstawie pojedynczego doświadczenia, wykorzystuje się mini-partię (z ang. *mini batch*) do aktualizacji, co prowadzi do bardziej stabilnego uczenia i zmniejsza wariancję gradientów. Przy wyborze rozmiaru partii istnieje pewien kompromis. Duży rozmiar partii może przyspieszyć obliczenia, ale może prowadzić do większej zmienności w aktualizacji wag. Z drugiej strony, mały rozmiar partii może spowolnić uczenie, ale ma tendencję do bardziej stabilnych aktualizacji.

3.2.3 Funkcja strat

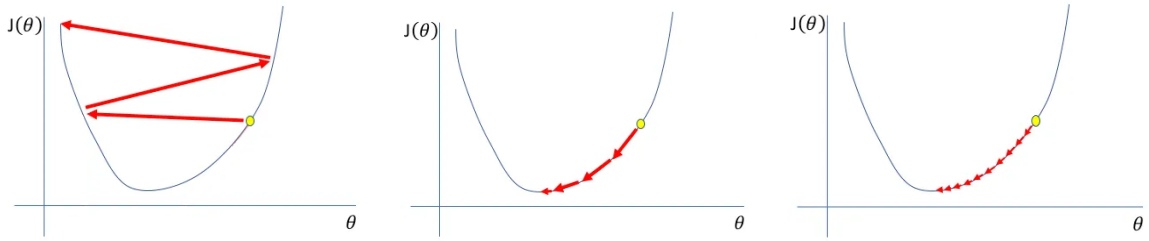
Głównym zadaniem sieci neuronowej podczas jej trenowania jest poprawianie swoich decyzji. Uczenie się w uczeniu maszynowym polega na minimalizacji funkcji strat (z ang. *loss function*) przez iteracyjne aktualizowanie wag sieci. Do obliczania tej funkcji potrzebne są dwie rzeczy: wartość wyliczona przez sieć neuronową oraz wartość, do której przy danym wektorze wejścia powinna dążyć sieć neuronowa (w uczeniu nadzorowanym są to etykiety nadane wcześniej przez człowieka, a w uczeniu przez wzmacnianie jest to wartość wyliczona przy użyciu równania Bellmana (3.8)). Jedną z częściej używanych funkcji strat jest średni błąd kwadratowy (z ang. *Mean Squared Error*), została ona wykorzystana przy budowie sieci neuronowej w tym projekci, a jej wzór został przedstawiony poniżej (3.9).

$$loss = (Q_{new} - Q)^2. \tag{3.9}$$

Oczywiście nie aktualizuje się wag w oparciu o jedną próbkę. W innych rodzajach uczenia maszynowego robi się to na podstawie całej dostępnej pamięci, natomiast w uczeniu ze wzmacnianiem wykorzystuje się do tego wcześniej określoną liczbę doświadczeń zapisanych w pamięci.

3.2.4 Optymalizator

Optymalizator jest metodą minimalizacji funkcji strat. Działa on w oparciu o spadek gradientowy. Spadek gradientowy to algorytm optymalizacyjny oparty na funkcji wypukłej, który iteracyjnie zmienia jej parametry, aby zminimalizować daną funkcję do jej lokalnego minimum, iteracyjnie zmniejsza funkcję straty, przesuwając się w kierunku przeciwnym do kierunku najbardziej stromego wzniesienia [7]. Jest to zależne od pochodnych funkcji straty w celu znalezienia minimów. Wykorzystuje dane całego zbioru uczącego do obliczenia gradientu funkcji kosztu do parametrów, co wymaga dużej ilości pamięci i spowalnia proces. Najważniejszym współczynnikiem przy zejściu gradientowym jest współczynnik uczenia (z ang. *learning rate*). Odzwierciedla on wielkość kroku, który prowadzi w kierunku minimum lokalnego. Na rysunku 3.5 przedstawiono zależność wielkości współczynnika uczenia się do funkcji kosztu. Jak widać zbyt mała wartość tego współczynnika wymaga więcej iteracji do osiągnięcia minimum. Z drugiej strony zbyt duża wartość może prowadzić do drastycznych zmian, co prowadzi do rozbieżnych zachowań.



(a) funkcja progowa/schodkowa (b) funkcja liniowa (c) funkcja ReLU

Rysunek 3.5 Wykresy zależności wielkości współczynnika uczenia się do osiągnięcia optymalnych wartości wag między neuronami [6]

W tym projekcie zastosowano jeden z częściej używanych optymalizatorów o nazwie ADAM (z ang. *Adaptive Moment Estimation*). Jest to algorytm łączący w sobie dwa inne rozwiązania: algorytm schodzenia gradientowego z pędem oraz algorytm RMSP. Optymalizator ADAM wykorzystuje dwie techniki bazujące na spadku gradientu. Pierwszą z nich jest pęd. Służy on do przyspieszania opadania gradientu poprzez uwzględnienie wykładniczej średniej ważonej gradientu. Zastosowanie średnich powoduje zwiększenie tempa zbliżania się algorytmu do minimów. Drugą techniką jest propagacja średniej kwadratowej. RMPS (z ang. *Root Mean Square Prop*) jest to zmodyfikowana wersja algorytmu AdaGrad, w którym zamiast stosować sumę kwadratów gradientów, stosuje się wykładniczą średnią ruchomą. W rezultacie optymalizator ADAM odznacza się niskimi kosztami trenowania i wysoką wydajnością.

3.2.5 Działanie algorytmu

Znając już podstawowe aspekty Q-learningu można przedstawić w pełni działanie tego algorytmu. Poniżej przedstawiono poszczególne kroki tej metody.

1. Inicjalizacja:
    - inicjalizuj sieć neuronową z losowymi wagami,
    - inicjalizuj pamięć doświadczenia na przechowywanie zebranych informacji przez agenta.

2. Główna pętla algorytmu. Dla każdej gry (epizodu) powtarzane są następujące kroki:
   - wybierz akcję na podstawie aktualnego stanu zgodnie ze strategią eksploracji/eksploatacji,
   - wykonaj wybraną akcję i obserwuj nagrodę oraz następny stan,
   - zapisz doświadczenie (stan, akcja, nagroda, następny stan) do pamięci,
   - jeżeli skończy się jedna gra lub epoka wykonaj krok uczenia, który został opisany w podpunkcie 3 oraz restartuj środowisko do stanu początkowego.
3. Uczenie się modelu:
   - z losowo wybranych próbek z pamięci (batch size) pobierz dane wejściowe: stany, akcje, nagrody i następne stany,
   - oblicz funkcję Q dla aktualnego stanu: $Q(s_t, a_t)$, używając sieci neuronowej,
   - oblicz funkcję Q dla następnego stanu:$Q(s_{t+1}, a_{t+1})$, używając sieci neuronowej,
   - oblicz docelowe wartości Q zgodnie z równaniem Bellmana,
   - oblicz błąd kwadratowy między obliczoną funkcją Q a docelową wartością Q i oblicz gradient tego błędu,
   - zaktualizuj wagi sieci Q, wykonując krok optymalizacji gradientowej (np. za pomocą algorytmu ADAM) w kierunku minimalizacji błędu.
4. Powtarzaj krok 2 i 3 przez ustaloną liczbę iteracji lub do osiągnięcia pożądanego poziomu wydajności.

3.2.6 Batch normalization

Trenowanie głębokich sieci neuronowych bywa trudne, ponieważ występują takie zjawiska jak nadmierne dopasowanie (z ang. *overfitting*), czy niedopasowanie (z ang. *underfitting*), dlatego powstały techniki pomagające z tymi problemami. Jedną z nich jest normalizacja wsadowa (z ang. *batch normalization*). Jest to metoda adaptacyjnej reparametryzacji motywowana trudnościami w uczeniu bardzo głębokich modeli. Gradient w sieciach neuronowych mówi o tym jak zaktualizować każdy parametr przy założeniu, że inne warstwy się nie zmieniają. W rzeczywistości wszystkie warstwy aktualizowane są jednocześnie. W rezultacie mogą wystąpić nieoczekiwane wyniki, ponieważ aktualizowane wartości wyliczane są dla złego założenia [3]. Problem ten narasta wraz ze wzrostem ilości warstw w sieci neuronowej. Można to przyrównać do celowania w ruchomy cel. Do przeliczania nowych wartości wag używana jest propagacja wsteczna (z ang. *backpropagation*), która sprawdza wszystkie wagi od końca do początku sieci neuronowej. W przypadku wielu warstw gradient powoli zaczyna zanikać. Normalizacja wsadowa normalizuje sygnał na poziomie warstwy, dzięki czemu sygnał nie traci na sile i może być odpowiednio przetwarzany przez kolejne warstwy.

Na początku na podstawie danych z partii (*batch*) wyliczana jest średnia (3.11) oraz odchylenie standardowe (3.12) dla każdego neuronu. Następnie dla każdego neuronu standaryzowane są wartości wejsciowe przez odejmowanie średniej i dzielenie przez odchylenie standardowe (3.10). Nowe wartości są przekształcone do zbioru, w którym średnia jest bliska 0, a odchylenie standardowe wynosi około 1. Na koniec wszystkie wartości są skalowane i przesuwane (3.13). Każdy neuron posiada unikalne parametry γ i β, które są wyliczane

automatycznie z każdym uczeniem się sieci neuronowej. Ten ostatni krok stosuje się w celu uwzględnienia nieliniowości.

$$H' = \frac{H - \mu}{\sigma}, \tag{3.10}$$

$$\mu = \frac{1}{m}\sum_i H_i, \tag{3.11}$$

$$\sigma = \sqrt{\delta + \frac{1}{m}\sum_i (H - \mu)_i^2}, \tag{3.12}$$

$$z^{(i)} = \gamma \bigotimes \widehat{x}^{(i)} + \beta. \tag{3.13}$$

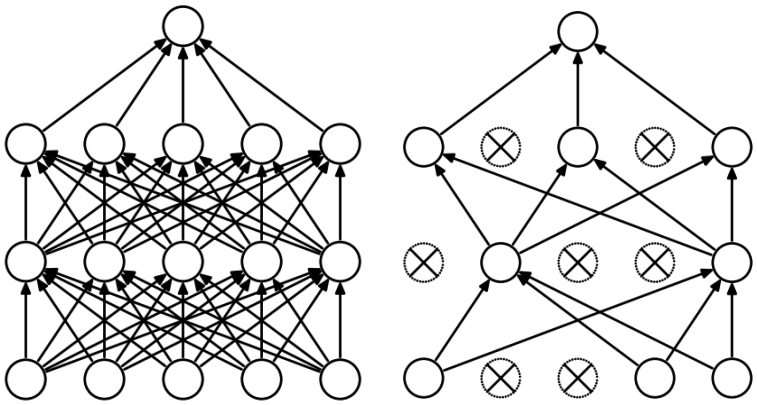
Wektor danych wejściowych w tym sposobie, również musi zostać znormalizowany. Oznacza to przeskalowanie wartości wejściowych do przedziału od 0 do 1, np. gdy wiemy, że dla danych wejściowych istnieje jakaś wartość maksymalna wystarczy wartości wejściowe podzielić przez tę wartość maksymalną. W ten sposób dane będą w odpowiednim przedziale oraz zostaną zachowane stosunki między danymi. Można to zauważyć np. przy klasyfikacji numerów pisanych ręcznie, w których wektor wejścia zawiera liczbowe przedstawienie jasności pikseli na danym zdjęciu. Im ciemniejszy piksel tym wartość jest bliższa 0, a im jaśniejszy piksel tym większa wartość, ale maksymalnie może osiągnąć wartość 255. W tym przypadku wystarczyłoby podzielenie wszystkich wartości przez 255. W tym projekcie wektor wejścia zawiera same zera i jedynki, przez co nie trzeba normalizować wartości wchodzących w skład wektora wejścia.

Używając pakietu Keras implementacja normalizacji wsadowej jest prosta. Wystarczy przy budowaniu modelu sieci neuronowej w odpowiednim miejscu wpisać funkcję *BatchNormalization()*.

3.2.7 Dropout

Kolejnym sposobem na poprawianie wyników sieci neuronowej jest metoda porzucania (z ang. *dropout*). Zasada działania tego sposobu jest prosta. Podczas uczenia się modelu dla każdej partii losowane są neurony przeznaczone do wyłączenia. Losowanie neuronów odbywa się na zasadzie rozkładu Bernoulliego, w którym każdy z neuron ma taką samą szansę na zostanie wylosowanym [10]. Następnie wylosowane neurony zostają tymczasowo wyłączone przez przemnożenie wartości aktywacji przez 0, co skutkuje ich wyzerowaniem. W rezultacie sygnał przechodzi tylko przez aktywne neurony. Reszta neuronów nie bierze udziału w propagacji sygnału, więc ich wpływ na wynik estymacji sieci neuronowej jest tymczasowo pomijany. Na rysunku 3.6 przedstawiono schemat działania metody porzucenia.

Technika ta ma kilka korzyści. Jedną z nich jest regularyzacja. Oznacza to, że przez losowe wyłączanie neuronów, model nie może polegać na pojedynczych neuronach, co minimalizuje efekt przeuczenia. Dodatkowo sieć neuronowa jest odporniejsza na pewnego rodzaju zmienność danych i szumy. Dzięki porzucaniu model lepiej generalizuje nowe dane. Dropout również redukuje współzależności między neuronami. Oznacza to, że sieć lepiej uczy się niezależnych cech.



(a) standardowa głęboka sieć neuronowa

(b) głęboka sieć neuronowa po zastosowaniu metody dropout

Rysunek 3.6 Model sieci neuronowej dropout [10]

Rozdział 4

Wyniki badań

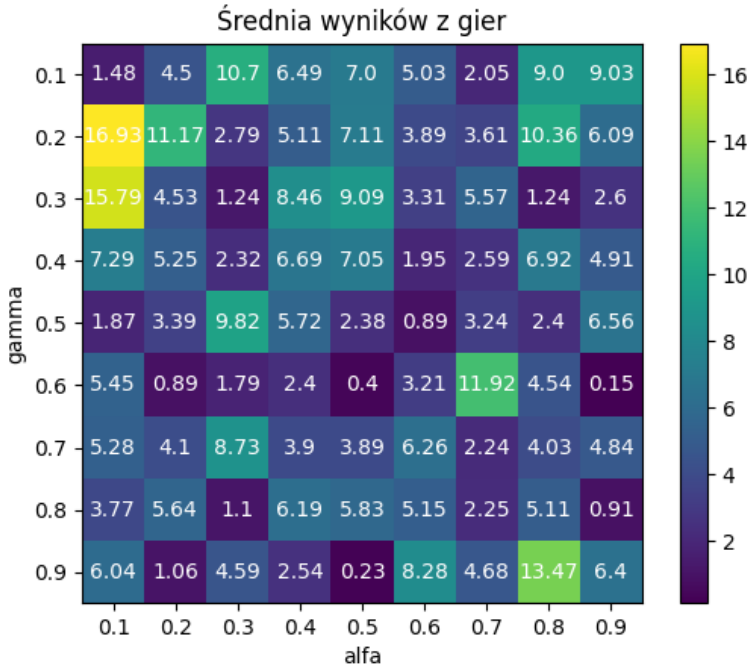
W niniejszym rozdziale przedstawiono wyniki przeprowadzonych badań oraz analizę zebranych danych. Celem tej pracy było porównanie dwóch metod uczenia przez wzmocnienie, czyli Q-learningu i deep Q-learningu. Dodatkowym celem było zgłębienie i zrozumienie zależności pomiędzy poszczególnymi czynnikami w obu metodach. Jako kryteria oceny przyjęto średnią z zagranych gier oraz najwyższy wynik z pojedynczej gry. Każdy agent był testowany w ten sam sposób. Do rozegrania miał 300 gier z czego pierwsze 150 było częścią treningową, zgodną z balansem eksploracji i eksploatacji. Na początku agent podejmuje decyzje w 100% losowo i z każdą kolejną grą ta szansa zmniejsza się, aż do gry numer 150, gdzie już w 100% podejmuje sam decyzje. Z tego powodu do analizy wyników brane są tylko rezultaty uzyskane w części eksploatacyjnej.

4.1 Agent oparty o Q-learning

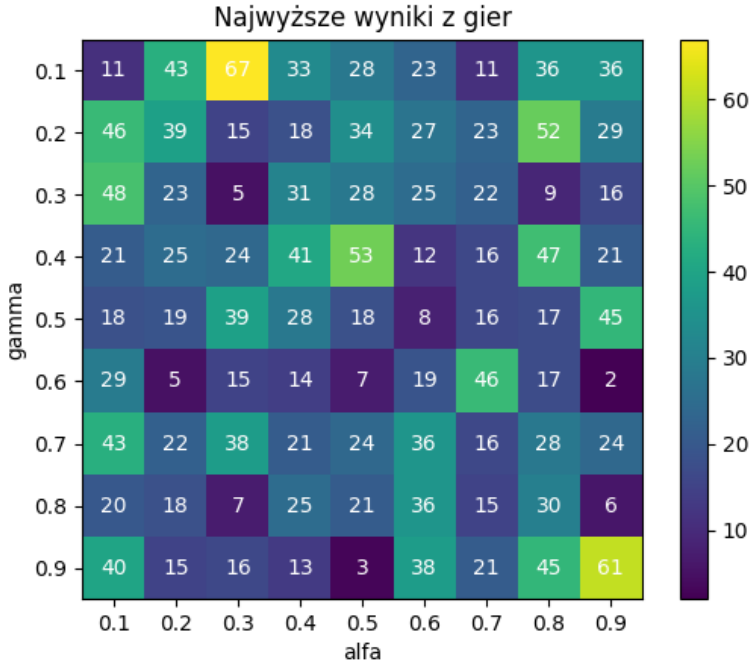
Jako pierwszy testowany był agent bazujący na Q-learningu. W tym przypadku zdecydowano się zbadać zależność dwóch parametrów występujących we wzorze Bellmana (3.2), czyli współczynnika uczenia się α oraz współczynnika dyskontowania γ. Oba te parametry występują w zakresie od 0 do 1. Zdecydowano się na przetestowanie kombinacji tych współczynników od wartości 0.1 do 0.9 ze skokiem 0.1 w obu przypadkach. Uzyskane wyniki, czyli średni wynik gier oraz najwyższy wynik z gier, zostały przedstawione na rysunkach 4.1 i 4.2.

Na podstawie tych wykresów można zauważyć, że najlepsze wartości dla obu parametrów znajdują się w dolnych wartościach, w zakresie do 0.3 dla obu wskaźników. Najlepszą konfiguracją okazało się połączenie $\alpha = 0.1$ i $\gamma = 0.2$, ponieważ osiągneła ona średnią wyników w przybliżeniu równą 16.93. Co dziwne w tym przypadku nie został zarejestrowany najwyższy wynik, ani nawet wynik z najlepszej piątki. Rekord tego przypadku zajmował dopiero 6 miejsce wśród pozostałych kombinacji. Barierę średniej wynoszącej 10 przekroczyło tylko 7 agentów. Co ciekawe drugi i trzeci najwyższy wynik w grze należą do agentów, którzy nie przekroczyli średniej 10, a nawet nie byli blisko, ponieważ jeden z nich osiągnął średnią 7.05, a drugi osiągnął średnią 6.4. Można również zauważyć, że niestety nie wszystkie kombinacje α i γ poradziły sobie z zadaniem. Średnie wyników 6 agentów nie przekroczyły nawet 1 punkta, a 13 nie przekroczyło 2 punktów. Tak niskie wyniki mogą być spowodowane tym, że gra jest losowa i podczas etapu eksploracji rzadko natrafiał na nagrody. Mogło to spowolnić jego proces nauki i prowadzić do nieoptymalnych taktyk. Jedną z nich była zbyt długa droga do nagrody, przez co gra się kończyła z uwagi na wyczerpanie limitu kroków. Innym problemem było, gdy przyjęta taktyka zakładała

pewien krok, którego w jakimś stanie algorytm nie mógł wykonać, przez co agent się zapętlał. Rozwiązaniem tego problemu mogłoby być zwiększenie ilości gier i wydłużenie okresu treningu, w którym występuje jeszcze eksploracja. Występowanie tych „luk" utrudniało przedstawienie planszy zależności α i γ jako przybliżonej płaszczyzny w 3D.



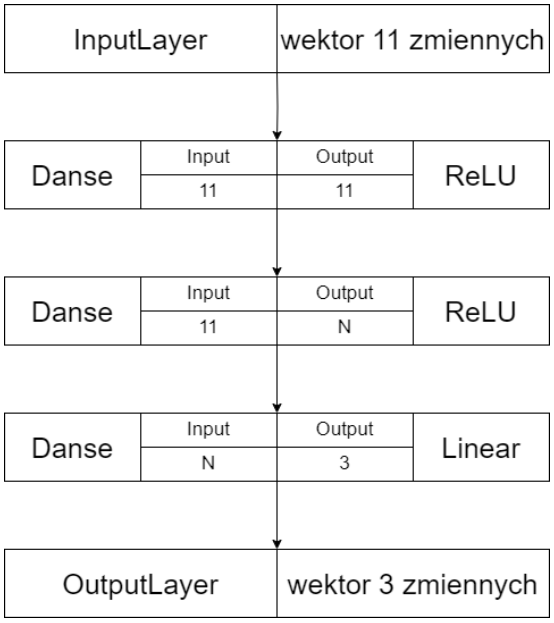
Rysunek 4.1 Rozkład średniej wyników w zależności od parametrów γ i α



Rysunek 4.2 Rozkład maksymalnych wyników w zależności od parametrów γ i α

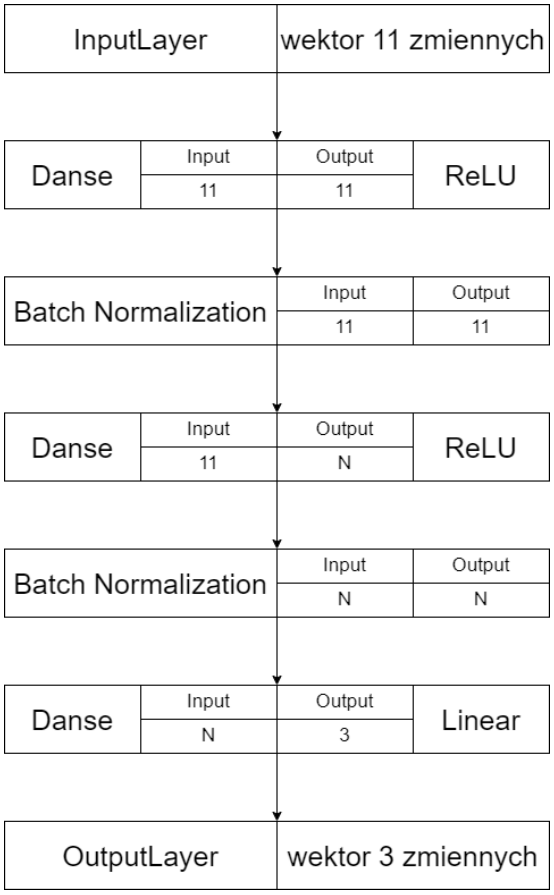
4.2 Agent oparty o deep Q-learning

Pierwotnie planowano zbadać wpływ wielkości jednej warstwy ukrytej w sieci neuronowej. Ostatecznie zdecydowano się na przetestowanie 6 różnych modeli głębokiej sieci neuronowej, w których zmieniano rozmiar warstw ukrytych. Pierwszym modelem był model z jedną warstwą ukrytą 4.3. Warstwa wejściowa *Danse* jest warstą gęstą, w której każdy neuron pobiera informacje ze wszystkich neuronów znajdujących się w warstwie poprzedniej. Ma ona rozmiar 11 neuronów aktywowanych funkcją $ReLU$, warstwa ukryta składa się z N neuronów aktywowanych funkcją $ReLU$, a w warstwie wyjściowej znajdują się 3 neurony aktywowane funkcją liniową.



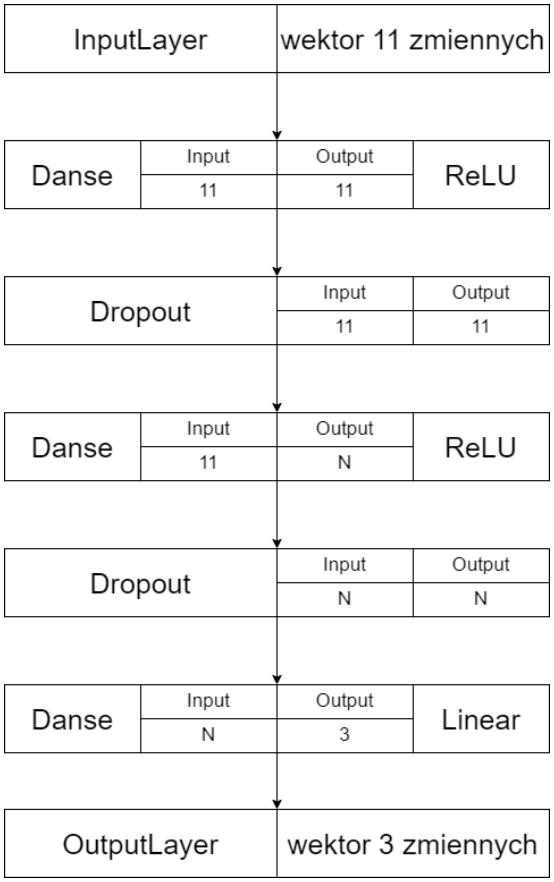
Rysunek 4.3 Architektura modelu głębokiej sieci neuronowej z jedną warstwą ukrytą

Drugim modelem była głęboka sieć neuronowa z jedną warstwą ukrytą wraz z metodą normalizacji wsadowej między wszystkimi warstwami, przedstawiono to na rysunku 4.4. Jest to modyfikacja modelu 4.3 poprzez zastosowanie między warstwą wejściową a ukrytą oraz między ukrytą a wyjściową warstwy *Batch Normalization*, mającej na celu znormalizowanie danych wyliczonych przez warstwę poprzedzającą.



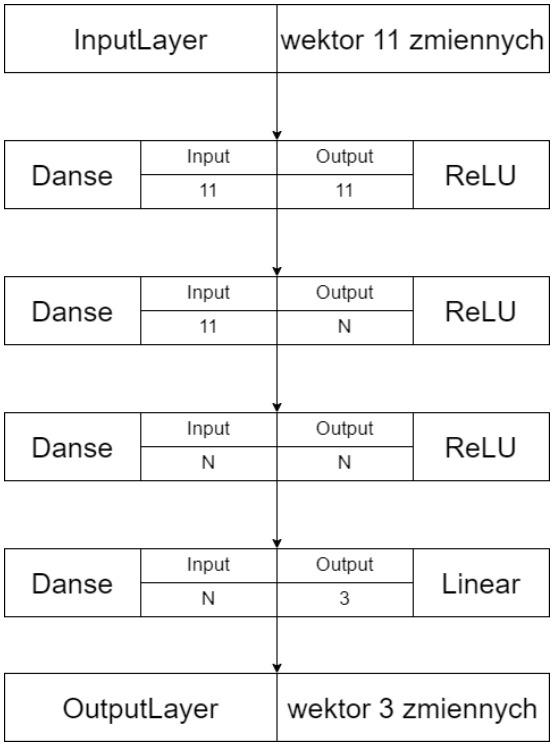
Rysunek 4.4 Architektura modelu głębokiej sieci neuronowej z jedną warstwą ukrytą oraz normalizacją wsadową

Trzecim modelem była głęboka sieć neuronowa z jedną warstwą ukrytą i zastosowaną metodą dropout między wszystkimi warstwami jak na rysunku 4.5. Przypomina on model 4.4 z tą różnicą, że zamiast normalizacji wsadowej zastosowano metodę *dropout* o współczynniku usuwanych neuronów równym 0.2.



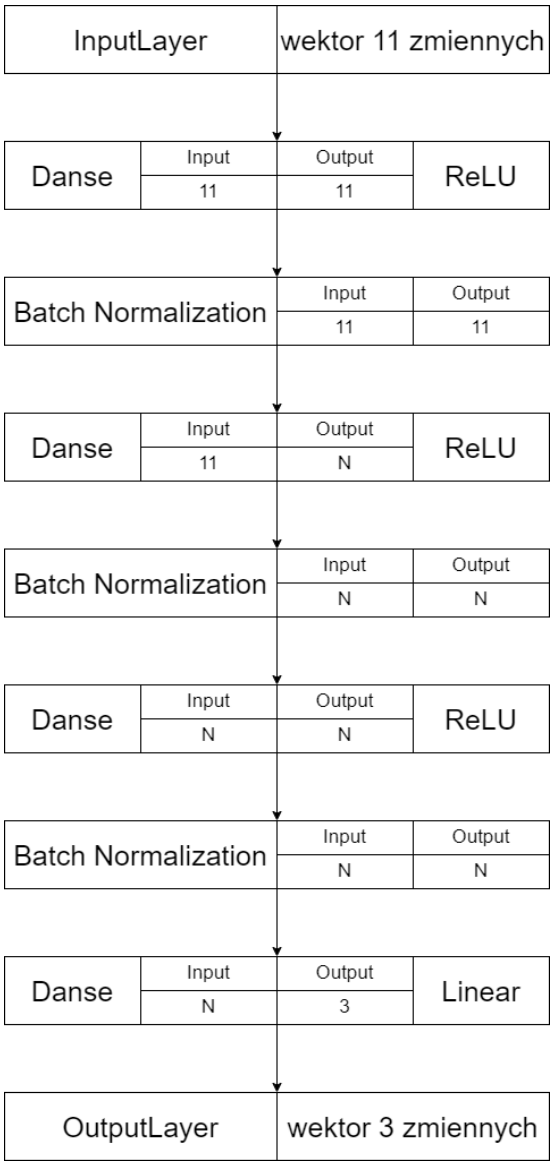
Rysunek 4.5 Architektura modelu głębokiej sieci neuronowej z jedną warstwą ukrytą oraz metodą porzucania

Kolejnym modelem była głęboka sieć neuronowa o dwóch warstwach ukrytych, przedstawiono ją na rysunku 4.6. Zawiera on 4 warstwy gęste *Danse*, z których pierwsza ma rozmiar 11, następne dwie mają rozmiar N, a ostatnia rozmiar 3. Pierwsze 3 warstwy posiadają funkcję aktywacji *ReLU*, a ostatnia warstwa aktywowana jest funkcją liniową.



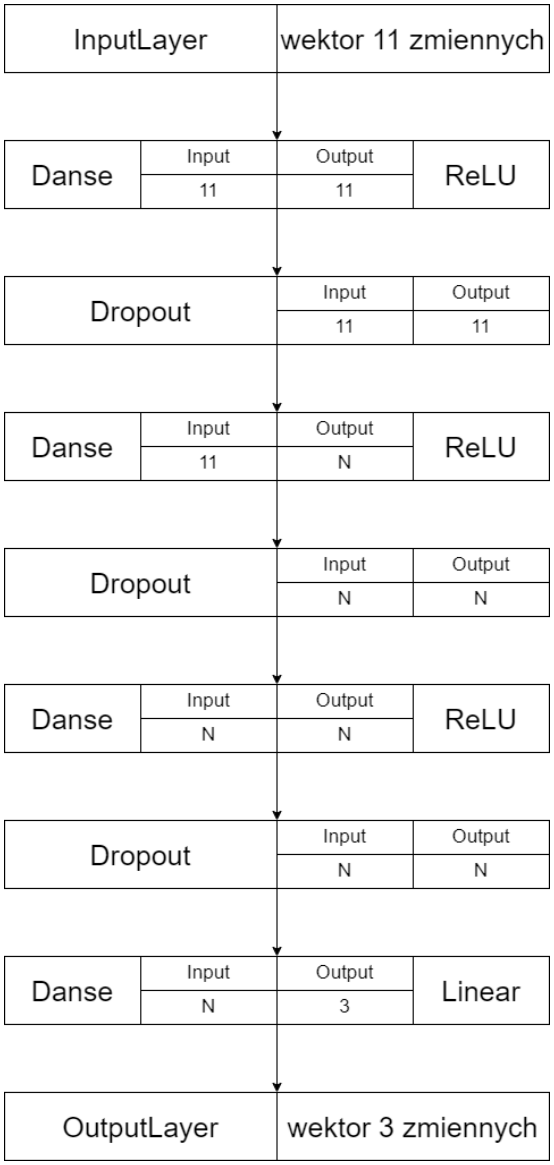
Rysunek 4.6 Architektura modelu głębokiej sieci neuronowej z dwiema warstwami ukrytymi

Piątym testowanym modelem była głęboka sieć neuronowa o dwóch warstwach ukrytych z zastosowaną metodą normalizacji wsadowej jak na rysunku 4.7. Jest to modyfikacja modelu 4.6 poprzez zastosowanie metody *Batch Normalization* między wszystkimi warstwami.



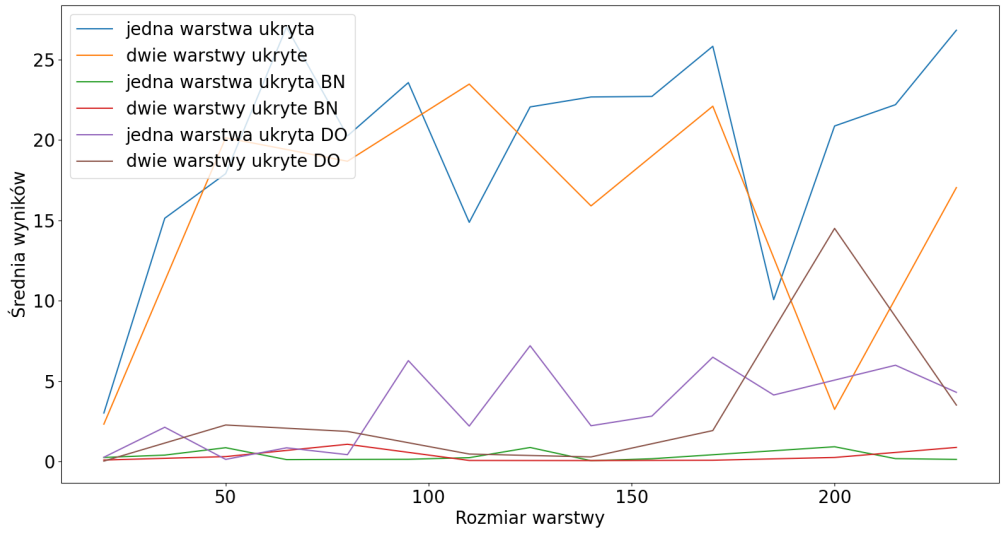
Rysunek 4.7 Architektura modelu głębokiej sieci neuronowej z dwiema warstwami ukrytymi oraz normalizacją wsadową

Ostatnim modelem była głeboka sieć neuronowa o dwóch warstwach ukrytych z zastosowaną metodą *dropout* jak na rysunku 4.8. Przypomina on model 4.7 z tą różnicą, że zamiast normalizacji wsadowej zastosowano metodę *dropout* o współczynniku usuwanych neuronów równym 0.2.

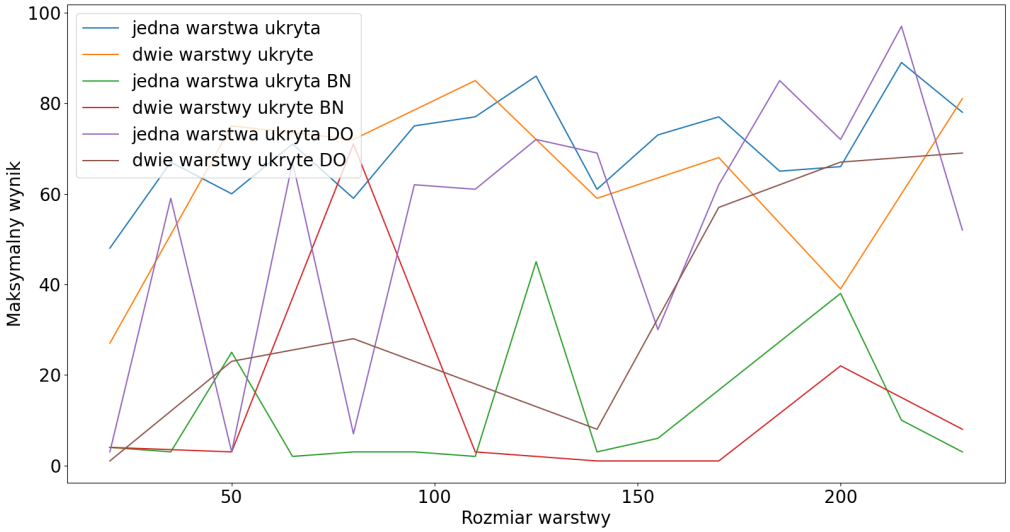


Rysunek 4.8 Architektura modelu głębokiej sieci neuronowej z dwiema warstwami ukrytymi oraz metodą porzucania

Rozmiary warstw ukrytych miały rozmiary N, gdzie N przyjmowało wartość od 20 do 230 ze skokiem 15 dla modeli z jedną warstwą ukrytą oraz ze skokiem 30 dla modeli z dwoma warstwami ukrytymi. Zdecydowano się na zwiększenie skoku z uwagi na skrócenie czasu testowania. Wyniki tych badań przedstawiono na rysunkach 4.9 oraz 4.10. Skróty BN oraz DO użyte w legendzie obu rysunków oznaczają wykorzystanie metody odpowiednio *batch normalization* oraz *dropout*.



Rysunek 4.9 Wykres średnich wyników osiągniętych przez agentów deep Q-learningu



Rysunek 4.10 Wykres maksymalnych wyników osiągniętych przez agentów deep Q-learningu

Na podstawie tych wykresów można zauważyć, że model z jedną warstwą ukrytą radził sobie lepiej od modelu z dwoma warstwami ukrytymi. Najwyższą średnią jaką osiągnął agent z jedną warstwą ukrytą wyniosła około 26.8. Dodatkowo zastosowanie modyfikacji w postaci normalizacji wsadowej czy porzucania pogarszały wyniki agentów. Jednakże model z jedną warstwą ukrytą i dropoutem osiągnął najlepszy wynik pojedynczej gry równy 97. Interesujący okazał się agent z dwoma warstwami ukrytymi i normalizacją wsadową, ponieważ dla całego badanego przekroju rozmiarów warstw (N), średnia nie przekraczała 3, ale w jednym przypadku, gdy N = 80 agent rozegrał 2 gry na conajmniej 66 punktów, a jedną na 71 punktów, co było prawie najwyższym wynikiem w danym rozmiarze warstwy. Można również stwierdzić, że zwiększenie rozmiaru warstwy (N) polepsza wyniki agentów, ale po osiągnięciu rozmiaru około 100, rezultaty nie zwiększają się już z taką prędkością jak przed osiągnięciem tej granicy. Z drugiej strony należy pamiętać o tym, że wraz ze wzrostem rozmiaru warstw zwiększa się liczba wag, przez co zwiększa się zapotrzebowanie na moc obliczeniową. Większa liczba wag do dostrojenia może prowadzić do wolniejszego uczenia się sieci neuronowej. Zwiększanie rozmiaru warstwy ukrytej, w przypadkach złożonych wpływa pozytywnie na działanie modelu, ale w przypadku prostszych problemów sieć zaczyna zbyt dokładnie dopasowywać się do danych, co prowadzi do efektu przeuczenia (z ang. *overfitting*) i utraty zdolności generalizowania przez sieć neuronową.

4.3 Porównanie wyników

Algorytm	Średni wynik	Najlepszy wynik
QL	16.93	67
DQL 1WU	27.03	89
DQL 1WU BN	0.91	45
DQL 1WU DO	7.2	97
DQL 2WU	23.48	85
DQL 2WU BN	1.07	71
DQL 2WU DO	3.51	69

Tabela 4.1 Wyniki testowanych algorytmów

W tabeli 4.1 przedstawiono osiagniecia dla poszczególnych algorytmów i modeli. W tabeli skróty algorytmów oznaczają: Q-learning (QL), deep Q-learning (DQL), warstwa ukryta (WU), batch normalization (BN), dropout (DO). Na podstawie tej tabeli można zauważyć, że najlepszym okazał się deep Q-learning z jedną warstwą ukrytą, ponieważ osiągnął on najwyższą średnią z rozegranych gier. Wynik średnie agenta opartego o Q-learning był o około 10 punktów mniejszy.

Rozdział 5

Podsumowanie

W zadaniu uczenia maszynowego najważniejszym zadaniem jest wybranie odpowiedniego algorytmu oraz dobranie jego parametrów dla danego problemu. W tej pracy postanowiono porównać dwa algorytmy uczenia ze wzmocnieniem, czyli Q-learning oraz deep Q-learning. Pierwszy z nich polega na ciągłym przeliczaniu aktualnych stanów, drugi natomiast wykorzystuje głębokie sieci neuronowe do estymowania podejmowanych akcji. Postanowiono przetestować je na grze komputerowej „Wąż". Głównym celem gry było zdobycie jak największej ilości punktów. Głównym zadaniem agentów podczas rozgrywki było wyliczanie wartości dla wszystkich możliwych akcji w danym stanie, a następnie podejmowanie najlepszej dla nich decyzji. Każdego agenta testowano na takiej samej partii gier. Na początku przetestowano metodę Q-learning. W tym przypadku sprawdzano wpływ dwóch parametrów na działanie algorytmów: α i γ. Oba mogą przyjmować wartości od 0 do 1. W wyniku przeprowadzonych testów, zauważono problem, z którym kilku agentów Q-learningu sobie nie poradziło. Była to losowość gry. Jednym z rozwiązań mogłaby być zwiększenie okresu trenowania agenta. Drugim sposobem ominięcia tego problemu mogłoby być zmiana systemu nagród na taki, który byłby zależny od odległości głowy węża od nagrody, a nie przyjmowania nagrody równej 0 w stanach nieterminalnych.

W przypadku deep Q-learningu zdecydowano się zbadać 6 modeli głębokiej sieci neuronowej. Na początku postanowiono przetestować 2 modele, jeden z jedną warstwą ukrytą i jeden z dwiema warstwami ukrytymi. Z uwagi na prostotę gry i relatywnie niski wymiar wektora wejścia nie zdecydowano się na zastosowanie sieci neuronowej o większej liczbie warstw ukrytych, ponieważ mogłoby to spowalniać uczenie się sieci, zwiększyć ryzyko przeuczenia się sieci czy zwiększyć złożoność obliczeniową pozyskanej sieci neuronowej. Następnie do obu tych modeli zastosowano metodę normalizacji wsadowej i warstwy porzucające (z ang. *dropout*). Na podstawie przeprowadzonych badań można stwierdzić kilka rzeczy. Po pierwsze zwiększenie rozmiaru warstwy ukrytej polepsza skuteczność modelu. Jednakże po osiągnięciu pewnego poziomu skuteczność ta nie zwiększa się już w tak szybkim tempie, za to zwiększa się złożoność obliczeniowa modelu. Po drugie zastosowanie drugiej warstwy ukrytej nie przyniosło lepszych efektów, a około dwukrotnie zwiększyła się ilość obliczanych parametrów względem modelu z jedną warstwą ukrytą. Po trzecie zastosowanie obu metod optymalizacji głębokich sieci neuronowych okazało się niekorzystne. W przypadku normalizacji wsadowej agent w każdej grze nie zdobywał punktów, z wyjątkiem pojedynczych gier. Może to być spowodowane tym, że średnia i wariancja są obliczane na podstawie partii danych (*batch*), a nie całej pamięci agenta. Może to wpływać na wyliczanie parametrów γ i β (4.9), dlatego bardzo istotne jest jakie komórki pamięci zastaną wybrane. Im więcej komórek z ważnymi informacjami tym normalizacja będzie lepsza. Niestety podczas treningu i zbierania doświadczenia agent najczęściej znajdował

się w sytuacjach, które nie dawały żadnej nagrody. Kiedy do aktualizacji parametrów γ i β wylosowano więcej wartościowych doświadczeń model był lepszy, ale prawdopodobieństwo na to było niewielkie. Przez to nieliczne gry agenta były dobre. Możliwe, że zmiana systemu nagród poprawiłaby wyniki tej metody. W przypadku wykorzystania warstwy z porzucaniem okazało się, że jest ona lepsza niż normalizacja wsadowa. Niestety nie przyniosła ona oczekiwanych efektów względem średniej wyników, ale w pojedynczych grach model z tą metodą potrafił osiągnąć więcej niż model bez niej.

Ostatecznie zgodnie z tezą można stwierdzić, że algorytm deep Q-learningu radzi sobie lepiej od algorytmu Q-learningu. Mimo to Q-learning dobrze sobie poradził z prostą grą komputerową. Wybór między tymi dwoma metodami zależy od rozmiaru potencjalnej macierzy Q oraz od dostępnej mocy obliczeniowej komputera. Warto również zauważyć, że zaproponowane algorytmy radziły sobie dobrze z grą typu „Wąż", jednakże należałoby w dalszych badaniach sprawdzić jak owe algorytmy zachowują się w przypadku innych gier.

Dodatek

Do pracy dołączono płytę CD zawierającą:

`/Praca_magisterska.pdf` — wersja cyfrowa pracy,

`/Kod_zrodlowy` — kod źródłowy oprogramowania wykorzystanego do symulacji środowiska gry, agenta oparty o Q-learning, agenta oparty o deep Q-learning, generowania wykresów. Dodatkowo dołączono pliki zawierające historię wyników osiąganych przez algorytmy.

Literatura

[1] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, J. Pineau. *An Introduction to Deep Reinforcement Learning.* NOW the essence of knowledge, Boston, 2018.

[2] GeeksforGeeks. Keras vs PyTorch. `https://www.geeksforgeeks.org/keras-vs-pytorch/`.

[3] I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning.* MIT Press, 2016.

[4] A. Géron. *Uczenie maszynowe z użyciem Scikit-Learn i TensoeFlow.* Helion, Gliwice, 2018.

[5] B. Jang, M. Kim, G. Harerimana, J. W. Kim. *Q-Learning Algorithms: A Comprehensive Classification and Applications.* IEEE Access, 2019.

[6] J. Jordan. Setting the learning rate of your neural network. `https://www.jeremyjordan.me/nn-learning-rate/`.

[7] B. Kröse, P. van der Smagt. *An introduction to Neural Networks.* Akademicka Oficyna Wydawnicza, Amsterdam, 1996.

[8] S. Osowski. *Sieci neuronowew ujęciu algorytmicznym.* WNT, Warszawa, 1997.

[9] K. Różanowski. Sztuczna inteligencja rozwój, szanse i zagrożenia. *Zeszyty Naukowe Warszawskiej Wyższej Szkoły Informatyki*, 2, 2007.

[10] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 2014.

[11] A. Surdyk, J. Szeja. *Kulturotwórcza funkcja gier Cywilizacja zabawy czy zabawy cywilizacji? Rola gier we współczesnośći.* Zakład Teorii i Filozofii Komunikacji, Poznań, 2008.

[12] R. S. Sutton, A. G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, Cambridge, 2014.

[13] R. Tadeusiewicz. *Sieci Neuronowe.* Akademicka Oficyna Wydawnicza, Warszawa, 1993.