AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INFORMATYKI, ELEKTRONIKI, I TELEKOMUNIKACJI

INSTYTUT TELEKOMUNIKACJI

Praca inżynierska

Implementacja prostej gry zręcznościowej i algorytmu korzystającego z uczenia (ze wzmocnieniem) uczącego się grać w tę grę

Implementation and algorithm supporting a simple arcade game by a reinforcement learning for playing that game

Autor:	*Dawid Kwapisz*
Kierunek studiów:	*Teleinformatyka*
Opiekun pracy:	*dr inż. Jarosław Bułat*

Kraków, 2022

Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystycznego wykonania albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.): „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej «sądem koleżeńskim».", oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Serdecznie podziękowania mojemu promotorowi
dr Jarosławowi Bułatowi za zaangażowanie
i merytoryczną pomoc w przygotowaniu pracy.

Spis treści

Wstęp

Uczenie maszynowe jest w obecnych czasach dynamicznie rozwijającą się dziedziną i dającą olbrzymie możliwości. Jedną z odmian uczenia maszynowego jest uczenie przez wzmacnianie (ang. *Reinforcement Learning*, w skrócie RL), które znajduje zastosowanie w różnych dziedzinach, między innymi w grach. Ostatni rozwój algorytmów uczenia przez wzmacnianie jest silnie nakierowany na uogólnienie działania algorytmów - tak, aby jeden algorytm był w stanie dostosować się do wielu, różnych środowisk. Dziedzina ta, znajduje szerokie zastosowanie między innymi w grach, robotyce, prowadzeniu autonomicznych pojazdów [1], automatyzacji przemysłu [2] czy w opiece zdrowotnej [3]. Uczenie przez wzmacnianie bywa też wsparciem dla innych metod uczenia maszynowego, jak na przykład przy przetwarzaniu języka naturalnego (ang. *Natural Language Processing*, w skrócie NLP), co powoduje zwiększenie skuteczności algorytmów [4].

Celem pracy jest implementacja gry zręcznościowej, która opiera swoją problematykę na znajdywaniu ścieżki oraz algorytmu bazującego na uczeniu przez wzmacnianie, który będzie w stanie osiągnąć pewną biegłość w wygrywaniu losowo wygenerowanych poziomów. Wykorzystując uczenie przez wzmacnianie możliwe jest zaprojektowanie algorytmu, który zaadaptuje się do określonego środowiska i będzie reagował na wszystko to, co będzie się w nim dziać.

W pierwszym rozdziale pracy opisano zagadnienia teoretyczne dotyczące uczenia przez wzmacnianie, które są niezbędne, aby zrozumieć dalszą część pracy. W drugim rozdziale, opisane zostały przykładowe implementacje algorytmów uczenia przez wzmacnianie w grach. Kolejny, trzeci rozdział jest poświęcony implementacji gry, algorytmu sztucznej inteligencji oraz implementacji komunikacji w architekturze klient-serwer. W rozdziale czwartym opisano przebieg uczenia algorytmu, który został podzielony na cztery etapy. W ostatnim, piątym rozdziale porównano wytrenowany model uczenia przez wzmacnianie z algorytmem A* pod kątem znajdowania najkrótszych tras oraz czasów obliczeń.

1. Wprowadzenie do uczenia maszynowego

Uczenie maszynowe jest tematem, który w ostatnich latach został bardzo spopularyzowany. Jego geneza sięga jednak lat 60 XX wieku. Wtedy to po raz pierwszy został użyty termin "uczenie maszynowe" (ang. *Machine Learning*, w skrócie ML). Takiego terminu użył pewien pracownik firmy IBM, który jest przez wielu uważany za pioniera w dziedzinie sztucznej inteligencji - Arthur Samuel. Był on również autorem pierwszego modelu sztucznej inteligencji grającego w Warcaby [5].

Ostatnie lata przyniosły znaczny skok mocy obliczeniowej komputerów, co w praktyce przełożyło się na olbrzymi rozwój algorytmów uczenia maszynowego i sztucznej inteligencji. Wiele czołowych firm, takich jak Google, Apple, Facebook czy Amazon zaczęły inwestować duże pieniądze w rozwój tych technologii i implementować je w swoich aplikacjach czy urządzeniach. Powyższe firmy opracowały również wiele rozbudowanych i dostępnych publicznie narzędzi służących do tworzenia modeli uczenia maszynowego. Przykładami takich bibliotek są: TensorFlow (Google/Alphabet), PyTorch (Facebook/Meta), czy mniej popularne Caffe (Berkeley Vision and Learning Center). Podobnych narzędzi jest wiele, więc pojawia się pytanie, które z nich wybrać? Na to pytanie nie ma jednoznacznej odpowiedzi, każde narzędzie ma swoje wady i zalety. W Internecie można znaleźć wiele porównań i opinii doświadczonych osób jak wybrać odpowiednią bibliotekę do swojego problemu. Na ten temat zostało również wykonane wiele badań - dobrym przykładem jest artykuł pt. "A Comparative Measurement Study of Deep Learning as a Service Framework" [6], w którym można znaleźć wiele informacji na temat porównania kwestii od strony sprzętowej i programowej kilku najpopularniejszych bibliotek.

Rozwój uczenia maszynowego i algorytmów sztucznej inteligencji przyczynił się do powstania wielu firm, które zajmują się rozwijaniem tej dziedziny. Przykładem takiego przedsiębiorstwa jest założone w 2010r. DeepMind, które odniosło wiele sukcesów, szczególnie w dziedzinie gier komputerowych. DeepMind zostało przejęte w 2014r. przez Google [7]. Ich sztandarowym projektem jest algorytm sztucznej inteligencji "AlphaGo", który jako pierwszy był w stanie pokonać człowieka w grę "Go" [8]. Na ich stronie można znaleźć wiele badań oraz publikacji w temacie uczenia maszynowego [9]. Kolejnym przykładem firmy, która odnosi sukcesy w dziedzinie sztucznej inteligencji, jest OpenAI. Największym i najważniejszym projektem prowadzonym przez OpenAI jest model *Generative Pre-trained Transformer* (w skrócie GPT), aktualnie w wersji GPT-3, który pozwala na generowanie tekstu na podstawie otrzymanej frazy. Firma udostępnia do swojego rozwiązania API wraz z dokumentacją, dzięki czemu możliwe jest zastosowanie modelu we własnej aplikacji [10]. W ostatnim czasie bardzo popularnym tematem jest

też projekt systemu sztucznej inteligencji *DALL-E 2*, który służy do generowania obrazów na podstawie podanej przez użytkownika frazy [11].

1.1. Podział technik uczenia maszynowego

Uczenie maszynowe jest bardzo szerokim tematem, który można podzielić na trzy główne kategorie.

Uczenie nadzorowane

Uczenie nadzorowane (ang. *supervised learning*) to najczęściej używana dziedzina uczenia maszynowego. Dane dostarczone do modelu są etykietowane, czyli przygotowane i opisane wcześniej przez człowieka. Model uczenia nadzorowanego uczy się sposobu przypisywania etykiet do danych wejściowych. Dlatego też, główne zastosowania algorytmów uczenia nadzorowanego to klasyfikacja i regresja. Wiele współczesnych aplikacji implementujących algorytmy uczenia maszynowego korzysta z tej metody. Najpopularniejsze zastosowania uczenia nadzorowanego to: klasyfikatory obrazów, optyczne rozpoznawanie tekstu, rozpoznawanie mowy, przewidywanie trendu. Główną wadą uczenia nadzorowanego jest konieczność utworzenia etykiet do danych. Zbiory danych wykorzystywane w uczeniu maszynowym zazwyczaj są bardzo duże, przez co zebranie i etykietowanie danych jest czasochłonnym procesem.

Uczenie nienadzorowane

Jest to dziedzina uczenia maszynowego, która zajmuje się transformacją danych wejściowych bez korzystania z etykietowania danych. Algorytmy uczenia nienadzorowanego (ang. *unsupervised learning*) służą między innymi do: wizualizacji danych (poprzez redukcję liczby wymiarów), kompresji, analizy skupień (grupowanie danych) czy usuwania szumu. Bardzo często skorzystanie z algorytmów uczenia nienadzorowanego pomaga w zrozumieniu zbioru danych przed wykorzystaniem ich w uczeniu nadzorowanym. Zaletą tej dziedziny jest możliwość analizy danych w inny sposób - możliwe jest poznanie powiązań między danymi, które ciężko zaobserwować stosując standardowe podejście.

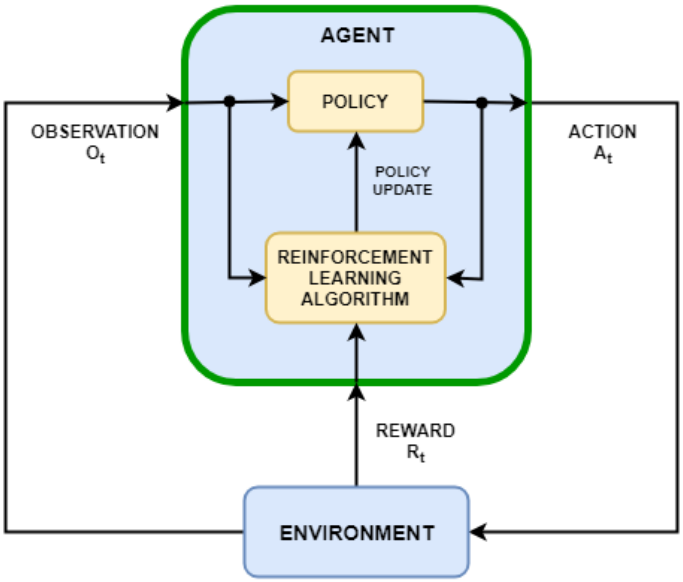
Uczenie przez wzmacnianie

Dziedzina, która przez długi czas była w cieniu uczenia nienadzorowanego i nadzorowanego. W przeciwieństwie do poprzednich kategorii, uczenie przez wzmacnianie (ang. *reinforcement learning*) nie wymaga zbierania i przygotowania danych uczących, co jest dużą zaletą. W tej dziedzinie, definiowane jest środowisko, w którym odbywa się badany problem. Elementem odpowiadającym za interakcję ze środowiskiem jest agent, a same interakcje nazywane są akcjami. Za wykonanie akcji, która jest zdefiniowana przez twórcę algorytmu jako pożądana, agent otrzymuje nagrodę, a za wykonanie złej akcji - karę. Na podstawie otrzymanych nagród i kar algorytm uczy się polityki - jakie akcje powinien wykonać w konkretnym stanie środowiska, aby zmaksymalizować nagrodę. Maksymalizowanie nagród jest jest podobną koncepcją co minimalizacja funkcji kosztu w uczeniu nadzorowanym, jednak w uczeniu przez

wzmacnianie dane nie mają zdefiniowanych etykiet. Można powiedzieć, że to sam model (agent) tworzy etykiety poprzez oddziaływanie ze środowiskiem i poznawaniem go. Sekwencja oddziaływań agenta ze środowiskiem w zdefiniowanych ramach nazywamy epizodem [12]. Pojedyncza sekwencja (wykonana akcja) nazywana jest krokiem lub iteracją. Żeby lepiej zobrazować elementy występujące w uczeniu ze wzmocnieniem dobrze jest przeanalizować konkretny przykład. Rozpatrując grę w szachy, zdefiniowane są następujące elementy:

- **środowisko**: plansza,
- **agent**: szachista,
- **nagroda**: wygranie partii, zbicie bierki
- **kara**: przegranie partii, utrata bierki
- **stan gry**: aktualna pozycja figur na szachownicy
- **akcje**: możliwe ruchy do wykonania w danym posunięciu
- **polityka**: lista akcji wraz z opisem, które posunięcie jest najlepsze (prowadzi do jak najwyższej nagrody: wygrania partii)
- **epizod**: partia
- **krok/iteracja**: pojedynczy ruch gracza

Ogólna zasada działania typowego algorytmu uczenia przez wzmacnianie została przedstawiona na rysunku 1.1.



Rys. 1.1. Sposób działania algorytmu uczenia przez wzmacnianie [13]

Algorytmy uczenia przez wzmacnianie znajdują dosyć szerokie zastosowanie w kilku dziedzinach, a najpopularniejszymi z nich są: gry, robotyka, autonomiczne pojazdy czy nawet chemia [14].

1.2. Wprowadzenie do uczenia przez wzmacnianie

Procesy decyzyjne Markowa

Rozpoczynając przygodę z uczeniem przez wzmacnianie należy w pierwszym kroku przyjrzeć się procesom decyzyjnym Markowa (ang. *Markov decision processes* - MDP). Oznaczając zmienne losowe: S_0 jako stan początkowy, S_t - stan w kroku t, A_t - akcja w kroku t oraz R_{t+1} - nagroda po wykonaniu akcji A_t w stanie S_t, możliwe jest określenie rozkładu prawdopodobieństwa warunkowego (1.1) zmiennych $S_{t+1} = s'$ i $R_{t+1} = r$ na podstawie poprzedniego stanu S_t i podjętej akcji A_t. Zmienne s - stan, s' - przyszły stan, r - nagroda oraz a - akcja oznaczają wartości przyjmowane przez odpowiadające im zmienne losowe.

$$p(s', r|s, a) = P(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a) \tag{1.1}$$

Równanie (1.1) definiuje tzw. dynamikę środowiska, na podstawie której możliwe jest wyliczenie wszystkich prawdopodobieństw zmian jakie zachodzą w środowisku. Metody, które wymagają znajomości dynamiki środowiska są nazywane metodami modelowymi (ang. *model-based*). W rzeczywistych problemach, dynamika środowiska najczęściej nie jest znana - korzysta się wtedy z metod bezmodelowych (ang. *model-free*) [12].

Problem oszacowania odległych nagród

W rzeczywistych środowiskach występuje problem szacowania nagród w czasie (ang. *future reward estimation*) - czy agent powinien dążyć do zbierania nagród natychmiastowych, czyli możliwych do zebrania jak najszybciej, czy zbierania przyszłych nagród? Może zajść sytuacja, gdy nagroda natychmiastowa będzie o bardzo niskiej wartości, a przyszłe nagrody będą znacznie przewyższały nagrody natychmiastowe. Problem ten jest również określany jako *sparse reward*. W pomocy określenia żądanej polityki zbierania nagród pomaga tzw. zwrot (ang. *return*). Przyjmując $R_{t+1} = r$ jako nagrodę natychmiastową (która jest otrzymywana po wykonaniu akcji A_t w czasie t), a R_{t+2}, R_{t+3} itd. jako kolejne nagrody, możliwe jest zdefiniowanie zwrotu G_t (1.2).

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0} \gamma^k R_{t+k+1} \tag{1.2}$$

Parametr γ jest nazywany współczynnikiem dyskontowania (ang. *discount factor*). Jego wartość mieści się w przedziale $[0, 1]$ i określa ona jak bardzo premiowane powinny być przyszłe nagrody w aktualnej chwili czasowej t. Mając do dyspozycji parametr γ można w prosty sposób utworzyć algorytm, który

będzie nastawiony na jak najszybszą nagrodę, nie biorąc pod uwagę przyszłych nagród. Dla $\gamma = 0$ jedynym czynnikiem w równaniu (1.2) zostanie R_{t+1} co oznacza, że algorytm będzie brał pod uwagę tylko nagrodę natychmiastową. W przypadku dobrania parametru $\gamma = 1$ uzyskany zostanie zwrot, który będzie sumą nagród w kolejnych krokach czasowych. W praktyce jednak, ustawienie współczynnika dyskontowania na 1 ma sens tylko w przypadkach środowisk w pełni deterministycznych - ta sama akcja zawsze daje w wyniku tę samą nagrodę. W przeciwnym przypadku, gdy środowisko nie jest deterministyczne, dobranie parametru $\gamma = 1$ jest błędne, ponieważ będziemy mieć do czynienia z nieskończoną sumą, która najprawdopodobniej nie będzie zbieżna. Gdy wartość parametru γ znajduje się w przedziale $(0, 1)$, wtedy widać następującą zależność - im dalej w przyszłości leży nagroda, tym mniejszą ma wagę [12].

Polityka

Kolejnym elementem uczenia przez wzmacnianie jest polityka (ang. *policy*), nazywana również strategią. Politykę definiuję się jako funkcję, która określa wybór kolejnych akcji. Może być ona w pełni deterministyczna, jednak w rzeczywistych środowiskach, najczęściej spotykana jest polityka stochastyczna, która wyznacza prawdopodobieństwo wybrania konkretnej akcji.

$$\pi(a|s) = P(A_t = a|S_t = s) \tag{1.3}$$

Strategia stochastyczna jest określana jako prawdopodobieństwo warunkowe (1.3) - jakie jest prawdopodobieństwo wyboru akcji A_t, pod warunkiem, że zachodzi stan S_t? Polityka może się zmieniać w trakcie uczenia modelu - agent zaczyna zdobywać doświadczenie i dowiaduje się, które akcje prowadzą do większej nagrody w określonym stanie. Często w algorytmach uczenia przez wzmocnienie, agent korzysta z polityki losowej, gdzie prawdopodobieństwo wyboru każdej z akcji jest takie same, a wraz z procesem uczenia, jest ona optymalizowana. Finalizacją procesu uczenia jest tzw. strategia optymalna $\pi_*(a|s)$, która prowadzi do zdobywania jak największych nagród przez agenta [12].

Równanie Bellmana

Wykorzystując zależności (1.2) i (1.3) możliwe jest zdefiniowanie tzw. funkcji wartości (ang. *value function*). Funkcja ta opisuje oczekiwany zwrot G_t w konkretnym stanie S_t (1.4). Warto zwrócić uwagę, że funkcja wartości jest zależna od aktualnej polityki π.

$$v_\pi(s) = E_\pi(G_t|S_t = s) \tag{1.4}$$

Kolejnym krokiem, jest opisanie funkcji akcja-wartość (ang. *action-value function*), która jest rozbudowaną wersją funkcji wartości. Definiuje ona oczekiwany zwrot G_t w danym stanie S_t, jeśli agent wybierze akcję a zgodnie z polityką π (1.5).

$$q_\pi(s, a) = E_\pi(G_t|S_t = s, A_t = a) \tag{1.5}$$

Takie zdefiniowanie funkcji wartości oraz funkcji akcja-wartość pozwala agentowi odpytywać w czasie rzeczywistym jaką powinien wykonać akcję w danym stanie. Dzięki temu, agent nie musi czekać na wynik długoterminowy. Funkcja wartości jest w stanie skondensować wszystkie możliwości poprzez uśrednienie nagród. Pozwala to na ocenę jakości różnych polityk.

Kluczową własnością funkcji wartości z punktu widzenia uczenia przez wzmocnienie jest możliwość jej rekurencyjnego zastosowania.

$$v_\pi(s) = E_\pi(R_t + \gamma G_{t+1}|S_t = s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)(r + \gamma v_\pi(s')) \tag{1.6}$$

Zależność (1.6) jest nazywana równaniem Bellmana, które jest podstawą działania algorytmów uczenia przez wzmocnienie.

Równanie to, opisuje dla każdej polityki π zależność jaka zachodzi między wartością stanu s, a wartością jej możliwych przyszłych stanów. Istnieje również zdefiniowane równanie Bellmana dla funkcji akcja-wartość.

$$q_\pi(s, a) = E_\pi(R_t + \gamma G_{t+1}|S_t = s, A_t = a) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)(r + \gamma \sum_a \pi(a|s) q_\pi(s', a')) \tag{1.7}$$

Struktura procesów decyzyjnych Markowa jest wykorzystywana w równaniu Bellmana, aby przekształcić nieskończoną sumę do układu równań liniowych. Rozwiązanie tych równań pozwala wyznaczyć dokładne wartości stanu. Głównym zadaniem uczenia przez wzmocnienie jest odnalezienie polityki, która w długim okresie czasu przyniesie jak największą nagrodę [15].

Eksploracja-eksploatacja

Algorytmy uczenia przez wzmacnianie często muszą przeciwstawić się tzw. problemowi "eksploracja vs eksploatacja" (ang. *exploration vs exploitation*). Wynika to z tego, że agent, który bierze udział w uczeniu, często wykonuje akcje, które wiążą się z niepewną nagrodą. Na początkowych etapach uczenia, agent nie posiada wiedzy o środowisku i nie jest w stanie określić, które akcje w danej chwili czasowej będą prowadzić do największej nagrody. Stąd też, pojawia się dylemat - czy agent powinien wykonywać najlepsze możliwe akcje i "eksploatować" aktualną politykę? Czy może lepszym wyborem będzie eksploracja środowiska i próba podjęcia nowych decyzji, mając nadzieję na jeszcze większe nagrody - ulepszenie polityki?

Jednym z najbardziej popularnych, a jednocześnie prostych rozwiązań, które zaradzają problemowi eksploracji i eksploatacji jest metoda ϵ-greedy [16]. Pozwala ona na zdefiniowanie prawdopodobieństwa (1.8), z którym będą wykonywane losowe i najlepsze możliwe (według aktualnej polityki) akcje.

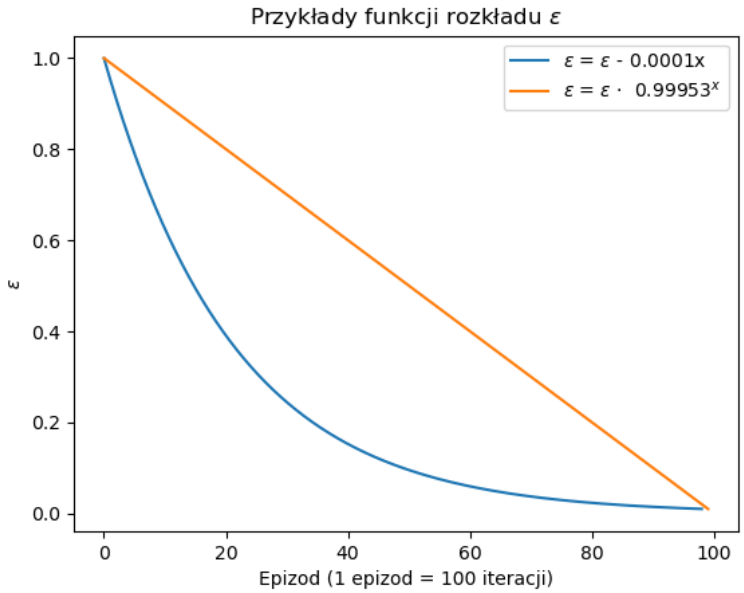
$$A = \begin{cases} \text{najlepsza możliwa akcja,} & \text{z prawdopodobieństwem } 1 - \epsilon, \\ \text{losowa akcja,} & \text{z prawdopodobieństwem } \epsilon \end{cases} \tag{1.8}$$

Dobranie odpowiedniej wartości parametru ϵ pozwala na uzyskanie równowagi pomiędzy eksploracją i eksploatacją w taki sposób, że przez większość czasu wybierana będzie akcja z najwyższą szacowaną nagrodą. Jednak element eksploracji pozwoli na podejmowanie nowych decyzji, czasem sprzecznych z aktualną polityką, umożliwiając tym samym, nauczenie się jeszcze lepszej polityki [17].

W rzeczywistych algorytmach, parametr ϵ nie powinien być wartością stałą, która nie zmienia się podczas trenowania algorytmu. Przykładowo, dla $\epsilon = 0.4$, 40% akcji będzie wybierane w sposób losowy, a 60% jako najlepsza możliwa akcja według aktualnej polityki. Bardzo często podczas tworzenia algorytmu, na samym początku procesu uczenia, agent kompletnie nie zna środowiska, którego próbuje się nauczyć. Wykonywanie przez niego 60% akcji jako najlepsze możliwe, wydaje się nie mieć sensu, ponieważ początkowo polityka nie jest optymalna. Podobna sytuacja zachodzi, gdy polityka zaczyna się klarować w kierunku polityki optymalnej - wtedy niepotrzebne wydaje się wybieranie aż 40% akcji w sposób losowy. Intuicyjnie wydaje się, że najlepszym podejściem będzie dobranie parametru ϵ tak, aby na początku procesu uczenia jego wartość była duża (np. bliska 1), gdy agent nie zna jeszcze środowiska. Wtedy początkowe decyzje będą wybierane w sposób losowy. Wraz z kolejnymi iteracjami, polityka jest ulepszana i jest coraz mniejsza potrzeba eksploracji środowiska - zwiększa się zapotrzebowanie na wykonywanie najlepszych akcji. Z tego powodu, implementując algorytm ϵ-greedy, ustala się parametr rozkładu ϵ_d oraz wartość minimalną ϵ_{min}. Z każdą kolejną iteracją wartość ϵ jest zmniejszana, aż do momentu osiągnięcia ϵ_{min}. Dzięki temu, na początku uczenia algorytmu, agent eksploruje środowisko, wykonując losowe akcje, a wraz z kolejnymi iteracjami, szala przesuwa się w kierunku eksploatacji - agent coraz częściej zaczyna wybierać najlepsze możliwe akcje. Pozostaje kwestia doboru wielkości parametru ϵ_d (jak szybko prawdopodobieństwo wybrania losowej akcji powinno zmaleć do wartości ϵ_{min}) oraz sposobu z jakim ϵ jest zmniejszany.

Istnieje wiele sposobów na implementację algorytmu ϵ-greedy i wyzwaniem jest wybranie odpowiedniego podejścia. Wartość parametru ϵ można redukować korzystając z wielu funkcji takich jak liniowa czy eksponencjalna. Każde środowisko jest inne, a odpowiednie wartości należy dobrać w sposób empiryczny. Do algorytmu można wprowadzić pewne modyfikacje, takie jak warunek, że parametr ϵ jest obniżany tylko wtedy, gdy agent odniesie sukces (otrzyma dużą nagrodę). Jeśli na samym początku uczenia, agent nie ma żadnej wiedzy o środowisku, wtedy warto rozważyć rozpoczęcie uczenia od $\epsilon = 1$, tak aby rozpocząć proces od całkowitej eksploracji.

Na rysunku 1.2 przedstawiono przykładowy rozkład parametru ϵ dla funkcji liniowej i eksponencjalnej. Podczas obliczeń założono, że 1 epizod składa się z dokładnie 100 iteracji algorytmu. Obliczenie liczby epizodów potrzebnych do osiągnięcia konkretnej wartości parametru $\epsilon = \epsilon_{min}$ (w tym przypadku $\epsilon_{min} = 0.01$) jest potrzebne, aby w odpowiedni sposób samą wartość parametru oraz współczynnika rozkładu, która będzie powiązana z ogólną liczbą epizodów podczas trenowania modelu.



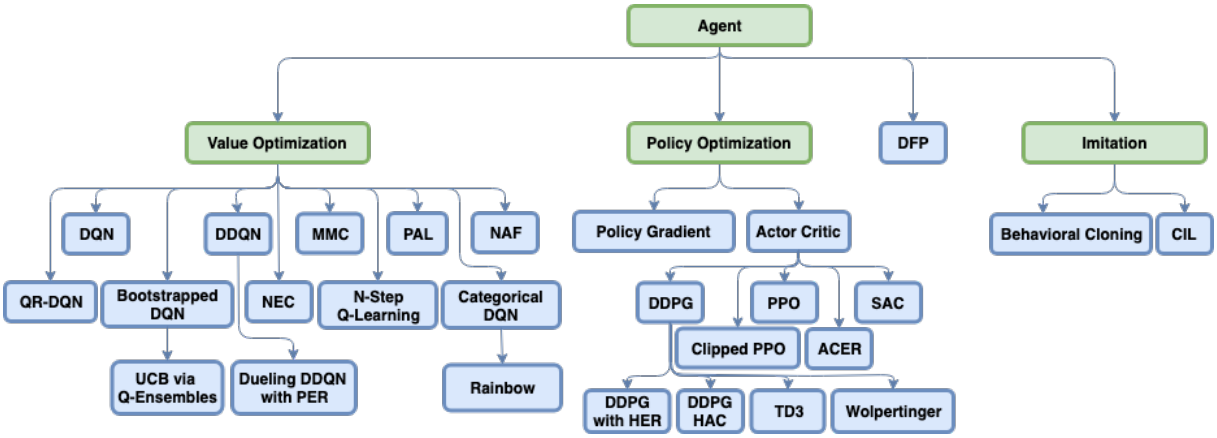
Rys. 1.2. Rozkład parametru epsilon - przykład funkcji liniowej i logarytmicznej

1.3. Wybrane metody uczenia przez wzmacnianie

Metod uczenia przez wzmocnienie jest wiele, a ich wybór jest mocno zależny od środowiska w jakim algorytm ma pracować. Poszczególne algorytmy można podzielić na trzy grupy: algorytmy optymalizujące wartości (ang. *Value Optimization*), algorytmy optymalizujące politykę (ang. *Policy Optimization*) oraz algorytmy imitacyjne (ang. *Imitation*). Podział większości z nich przedstawiono na rysunku 1.3.

Ważnym kryterium podczas wyboru algorytmu jest specyfika środowiska i przestrzeni akcji. Środowisko może dzielić zadania na epizody - każdy epizod zaczyna się w konkretnym czasie t i kończy się stanem końcowym S_t. Zadanie może być zdefiniowane bez określonego momentu definiującego zakończenie epizodu - nie istnieje wtedy stan końcowy S_t [12]. Przykładem środowiska z zadaniem skończonym jest gra w szachy - gracze rozpoczynają partię w określonym momencie - gdy wszystkie figury są na swoich miejscach, a kończą po poddaniu się zawodnika, wygraniu partii poprzez "Szach mat" lub po wypełnieniu warunku uprawniającego do remisu. Przykładem środowiska, które charakteryzuje się nieskończonym zadaniem jest tzw. "Cart-Pole" [18]. W tym przypadku nie ma możliwości zakończenia zadania w stanie końcowym S_t.

Przy wyborze algorytmu należy też wziąć pod uwagę przestrzeń akcji (ang. *action space*). Przestrzeń ta, może być dyskretna lub ciągła. Przykładem środowiska, które charakteryzuje się dyskretną przestrzenią akcji jest wcześniej wspomniany "Cart-Pole". Jedyne akcje jakie mogą być wykonane w tym środowisku to: ruch w prawo oraz ruch w lewo. Obie akcje są wykonywane ze stałymi wartościami siły. Środowiskiem, w którym występuje ciągła przestrzeń akcji jest na przykład "Mountain Car" [19]. W tym przypadku, sterując samochodem należy podać siłę, która rozpędzi samochód w podjeżdżaniu pod górę. Podawana wartość siły nie jest stała i mieści się w ciągłym przedziale $[-1, 1]$.



Rys. 1.3. Podział algorytmów uczenia przez wzmacnianie [20]

Q-Learning

Q-Learning jest bezmodelową metodą uczenia przez wzmacnianie, która swoje początki miała już w latach 90 [21]. Ogólną zasadą działania jest przyrostowa aktualizacja wartości akcji w zależności od stanu. Do każdej możliwej pary akcja-stan przypisywana jest wartość liczbowa, która jest definiowana poprzez funkcję natychmiastowej nagrody za podjęcie akcji a oraz oczekiwanej nagrody w przyszłości w oparciu o następny stan s', który jest wynikiem podjęcia tej akcji a. Z takich par, tworzona jest macierz Q (nazywana często Q-tablicą, ang. *Q-table*), która najczęściej jako wiersze przyjmuje wszystkie możliwe stany, a jako kolumny - wszystkie możliwe akcje, które może podjąć agent. Każda komórka macierzy zawiera wartość liczbową, która ocenia wartość akcji a w stanie s. Podczas procesu uczenia, aktualizacja macierzy przebiega za pomocą równania (1.9), gdzie $Q(s, a)$ to wcześniej wspomniana macierz, r - nagroda, (s, a) - para stan-akcja w aktualnym kroku, (s', a') - para stan-akcja w przyszłym kroku, γ - współczynnik dyskontowania i α - szybkość uczenia (ang. *learning rate*). Parametr szybkości uczenia jest potrzebny, aby kontrolować zbieżność uczenia. Zbyt duża wartość parametru α bardzo często powoduje brak możliwości znalezienia optymalnej polityki, a zbyt mała sprawi, że uczenie będzie trwało dłużej. Zwykle szybkość uczenia jest dobierana z wartości rzędu 10^{-4}.

$$Q(s, a) = Q(s, a) + \alpha[r + \gamma max_{a'} Q(s', a') - Q(s, a)] \tag{1.9}$$

Należy zwrócić uwagę, że powyższe rozwiązanie ma znaczną wadę - ilość informacji, którą należy przechowywać jest bardzo duża. Wynika to z potrzeby przechowywania każdego możliwego stanu oraz każdej możliwej akcji, co daje rozmiar macierzy równy $S \cdot A$, gdzie S - liczba wszystkich stanów, A - liczba wszystkich akcji. Dla środowisk o ciągłej przestrzeni akcji i/lub ciągłej przestrzeni stanów - nie ma możliwości zaimplementowania tej metody. Wynika to z faktu, że ciągła przestrzeń zawiera nieskończoność wartości pośrednich - niemożliwe jest więc stworzenie tabeli Q, która te wszystkie wartości przechowa.

Proces uczenia algorytmu Q-Learning wygląda następująco:

Zainicjalizuj tablicę $Q(s, a)$ w dowolny sposób
Powtórz (wykonaj założoną liczbę epizodów):
    Zainicjalizuj stan s
    Powtórz (wykonaj kolejne kroki w epizodzie):
        Wybierz akcję a na podstawie stanu s na podstawie $Q(s, a)$
        Wykonaj akcję a, zaobserwuj nagrodę r oraz nowy stan s'
        Wykonaj aktualizację tablicy $Q(s, a)$ (równanie 1.9)
        Przypisz s' do stanu s

Q-Learning jest stosowany w środowiskach podlegających pod procesy decyzyjne Markowa. Jednak gdy przestrzeń stanów i akcji jest zbyt duża, wtedy algorytm nie jest w stanie ich wszystkich się nauczyć - powoduje to znaczne problemy z uogólnieniem działania modelu [22]. Kolejnym problemem, z jakim można się spotkać implementując algorytm Q-Learning jest zjawisko przeszacowania wartości (ang. *The Overestimation Phenomenon*) [23]. Wynika on z tego, że podczas uczenia może dojść do sytuacji, gdy wiele wartości w macierzy jest bardzo podobna - przykładowo, stan s_1 zawiera akcje (a_1, a_2, a_3, a_4), które mają przypisane wartości kolejno $(0.95, 0.93, 0.97, 0.34)$. Z samej idei działania algorytmu, powinno się dążyć do wykonywania akcji o największej wartości z tablicy Q. Jednak jak łatwo zauważyć - wartości akcji a_1, a_2, a_3 są bardzo podobne. Biorąc pod uwagę możliwość pojawienia się szumu [22], dochodzi do sytuacji, gdy nachodzą na siebie różne wartości akcji - są one bardzo podobne co prowadzi do błędów, ponieważ akcja o wartości 0.97 wcale nie musi być tą najlepszą - może być przeszacowana.

Deep Q-Network

Metoda Deep Q-Network (DQN) jest podobną metodą co Q-Learning, jednak rozwiązuje kilka jej problemów. Ogólne założenie jest następujące - zastąpić mało wydajną metodę przechowywania wartości akcja-stan w postaci tabeli/macierzy Q innym narzędziem, które dobrze sprawdza się w aproksymacji funkcji - siecią neuronową. Na wejście sieci neuronowej podawany jest aktualny stan środowiska, a na wyjściu otrzymywana jest lista akcji wraz z ich wartościami. Z założenia, im wyższa wartość, tym akcja jest lepsza, prowadzi do większej nagrody. Pierwszym zyskiem jaki z tego wynika jest możliwość zaimplementowania metody w środowiskach z ciągłą przestrzenią stanów - sieć neuronowa nie potrzebuje definicji całej przestrzeni stanów do budowy modelu. Na wejście do sieci można wprowadzić wartości ciągłe. Takie podejście powoduje również, że nie jest konieczne odwiedzenie każdego możliwego stanu przez agenta, co jest wymagane przez metodę Q-Learning, aby zbudować dobrze działającą Q-tablicę. Kolejną zaletą jest różnorodność oraz dowolność pod względem architektur sieci neuronowych. Możliwe jest zaimplementowanie modelu bazującego na perceptronie wielowarstwowym, ale jest kilka alternatyw. Jedną z nich są sieci splotowe, które doskonale sprawdzają się przy przetwarzaniu obrazów. Daje to np.

możliwość użycia ramki (zrzutu ekranu) jako stanu środowiska. W teorii, zrzuty ekranu jako stan można również zaimplementować w metodzie Q-Learning, jednak byłoby to wysoce nieefektywne.

Proces uczenia sieci DQN znacznie różni się od uczenia w przypadku Q-Learning. Ucząc model Deep Q-Network wskazane jest zaimplementowanie bufora powtórek (ang. *Expierence Replay* lub *Replay Buffer*). Służy on do przechowywania informacji z danego kroku algorytmu, takich jak:

- **s**: aktualny stan
- **a**: aktualnie wykonana akcja
- **s'**: stan po wykonaniu akcji a
- **r**: nagroda za wykonanie akcji a
- **t**: stan zakończenia epizodu (ang. *terminal*) -> jeśli w danym kroku epizod się zakończył, wpisana jest wartość 1, w przeciwnym przypadku 0 [24]

Zasada działania bufora jest prosta - z punktu widzenia implementacji, tworzone są odpowiednie listy/tablice o zadanej wielkości, w której przetrzymuje się wszystkie dane. Gdy miejsca w buforze zaczyna brakować (listy/tablice są w 100% zapełnione danymi), wtedy rozpoczyna się nadpisywanie danych, które zostały wpisane najwcześniej i cały proces zapisu zaczyna się od nowa. Jest to tzw. bufor kołowy. Implementacja takiej pamięci znacznie zwiększa wydajność próbkowania algorytmu, ponieważ umożliwia to wielokrotne wykorzystywanie tych samych danych do treningu (zamiast pozbywanie się ich natychmiast po zebraniu). Istnieje wiele metod pobierania danych z bufora, jednak w pracy skupiono się nad najbardziej podstawową - równomiernym próbkowaniu (ang. *uniform sampling*). Oznacza to, że dane zostają wyciągane z bufora w sposób losowy, z takim samym prawdopodobieństwem. Powoduje to poprawę stabilności sieci podczas szkolenia dzięki użyciu nieskorelowanych ze sobą partii danych [25].

Z tak utworzonego bufora, w celu uczenia wyciągana jest seria danych, tzw. *batch*, który następnie jest wykorzystywany do uczenia modelu. Sam proces trenowania został przedstawiony w równaniach (1.10-1.12) i polega na minimalizowaniu funkcji straty $L_i(\theta_i)$ (ang. *loss function*) w każdym kroku algorytmu. Zmienna θ opisuje parametry (wagi) sieci neuronowej w kroku i.

$$L_i(\theta_i) = (y - x)^2 \tag{1.10}$$

$$x = Q(s, a; \theta_i) \tag{1.11}$$

$$y = r + \gamma max_{a'} Q(s', a'; \theta_i) \tag{1.12}$$

Zmienne opisują kolejno: s - stan, a - akcja, r - nagroda po wykonaniu akcji a w stanie s, γ - współczynnik dyskontowania. Funkcja (1.10) opisuje stratę, jaka jest pomiędzy wartością docelową (1.12), a wartością, która została przewidziana przez sieć (1.11). Parametr y jest nazywany różnicą czasową (TD

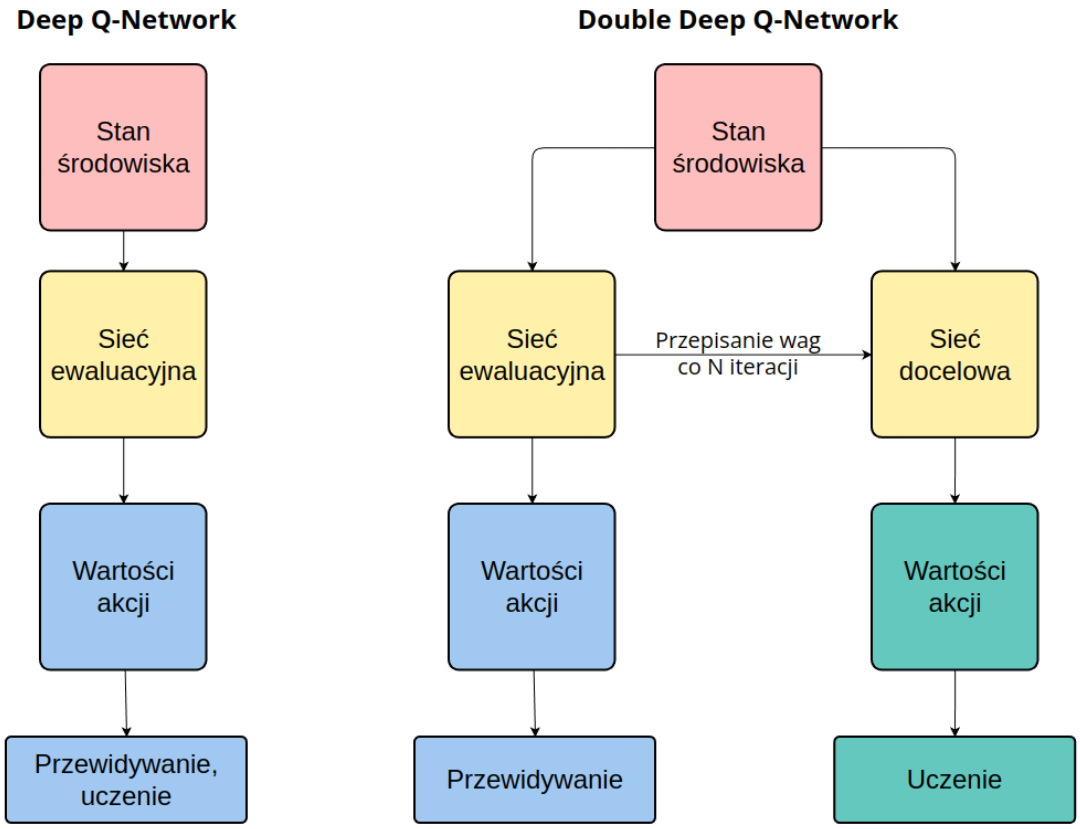
- ang. *temporal difference*), natomiast $y - x$ jest nazywane błędem różnicy czasowej (ang. *TD error*) [26]. Taka technika obliczania wartości ma swoją nazwę - *Temporal Difference Learning* i znajduje zastosowanie nie tylko w uczeniu przez wzmacnianie, ale ogólnie w uczeniu maszynowym oraz innych dziedzinach. Jest to metoda, która pozwala na przewidywanie danej wielkości, która jest zależna od przyszłych wartości sygnału. Podejście to, stara się przewidzieć jaka będzie natychmiastowa nagroda w chwili t oraz nagroda w kolejnej chwili $t + 1$. Gdy moment $t + 1$ nadejdzie, nowe przewidywania są porównywane z wartościami oczekiwanymi. Jeśli te przewidywania się od siebie różnią, algorytm oblicza różnicę, oraz stara się dostosować swoje przyszłe predykcję, tak aby poprawić swoje działanie [27]. W tym przypadku, to sieć neuronowa odpowiada za wszystkie przewidywania i w momencie gdy różnica wystąpi - następuje aktualizacja parametrów sieci w taki sposób, aby tę różnicę zminimalizować - tak, aby strata była jak najmniejsza.

Ta metoda ma jednak sporą wadę. Wykorzystanie tej samej sieci neuronowej zarówno do obliczania wartości docelowej (ang. *target*), jak i wartości przewidywanej (ang. *predict*) zwiększa prawdopodobieństwo wyboru zawyżonych wartości, co skutkuje zbyt optymistycznymi oszacowaniami. Wartości Q szacowane przez jedną sieć będą na ogół większe niż prawdziwe wartości Q, przez co agent będzie przeceniał oczekiwane przyszłe nagrody, co zaburzy uczenie algorytmu. [28].

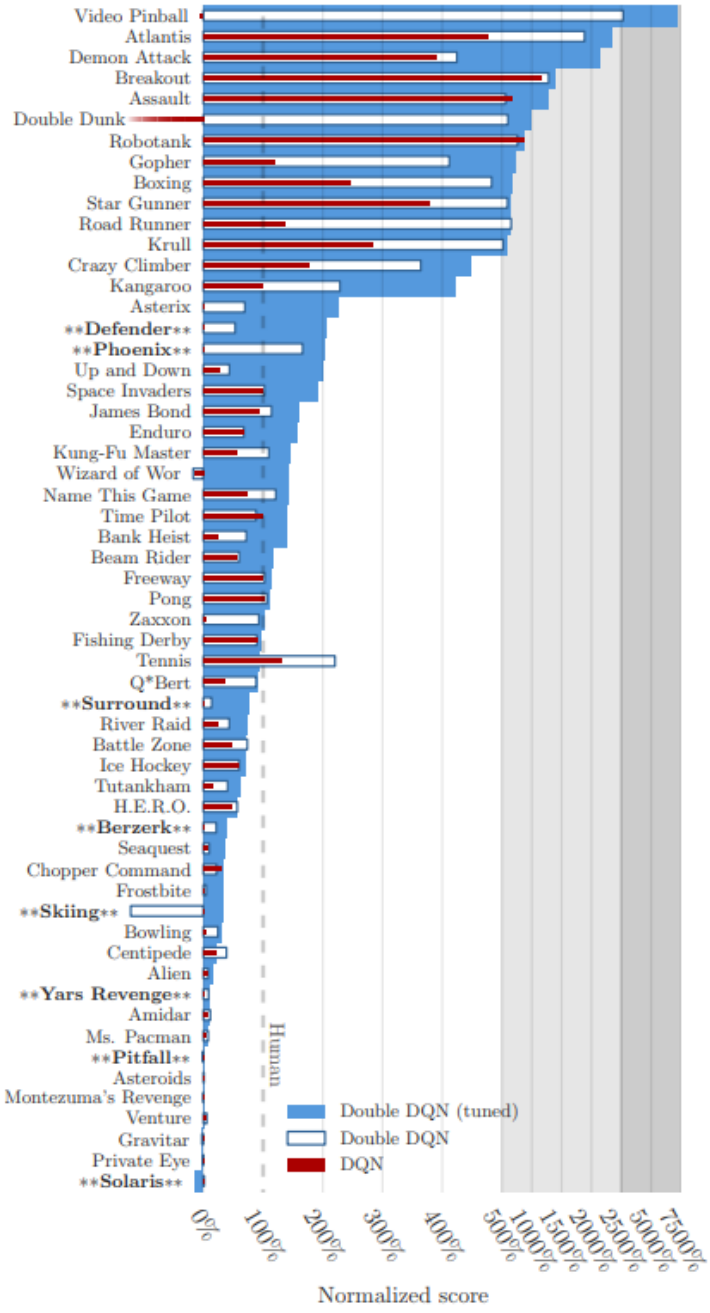
Double Deep Q-Network

W związku z problemem zbyt optymistycznych oszacowań, powstała metoda Double Deep Q-Networks (DDQN), nazywana również Double Q-Learning. Rozbudowuje ona metodę Deep Q-Network poprzez zastosowanie dwóch sieci neuronowych - docelową (ang. *target network*) oraz ewaluacyjną (ang. *evaluation network*). W metodzie DQN, jedna i ta sama polityka odpowiada za obliczanie akcji i jej ocenianie. W metodzie DDQN, sieć docelowa zajmuje się obliczaniem różnicy czasowej (1.12), natomiast sieć ewaluacyjnej - wartością przewidywaną (1.11). Zasada działania została przedstawiona na rysunku 1.4. Rozpoczynając proces uczenia, następuje inicjalizacja obu sieci w sposób niezależny. Co określoną liczbę iteracji, następuje przepisanie wag z sieci ewaluacyjnej do sieci docelowej. Liczbę iteracji, po której powinno nastąpić przepisanie wag należy dobrać w sposób empiryczny.

Wprowadzanie opóźnienia do polityki pozwala uzyskać lepsze rezultaty podczas uczenia, co zostało dowiedzione empirycznie w pracy pt. "Deep Reinforcement Learning with Double Q-Learning". W tej pracy potwierdzono, że zastosowanie dwóch osobnych sieci neuronowych do obliczania wartości przewidywanych i docelowych powoduje stabilniejszy proces uczenia i najczęściej prowadzi do uzyskania lepszych wyników [28]. Przykładowe zyski z zastosowania metody DDQN zostały przedstawione na rysunku 1.5.



Rys. 1.4. Zasada działania dwóch sieci w metodzie DDQN w porównaniu do metody DQN

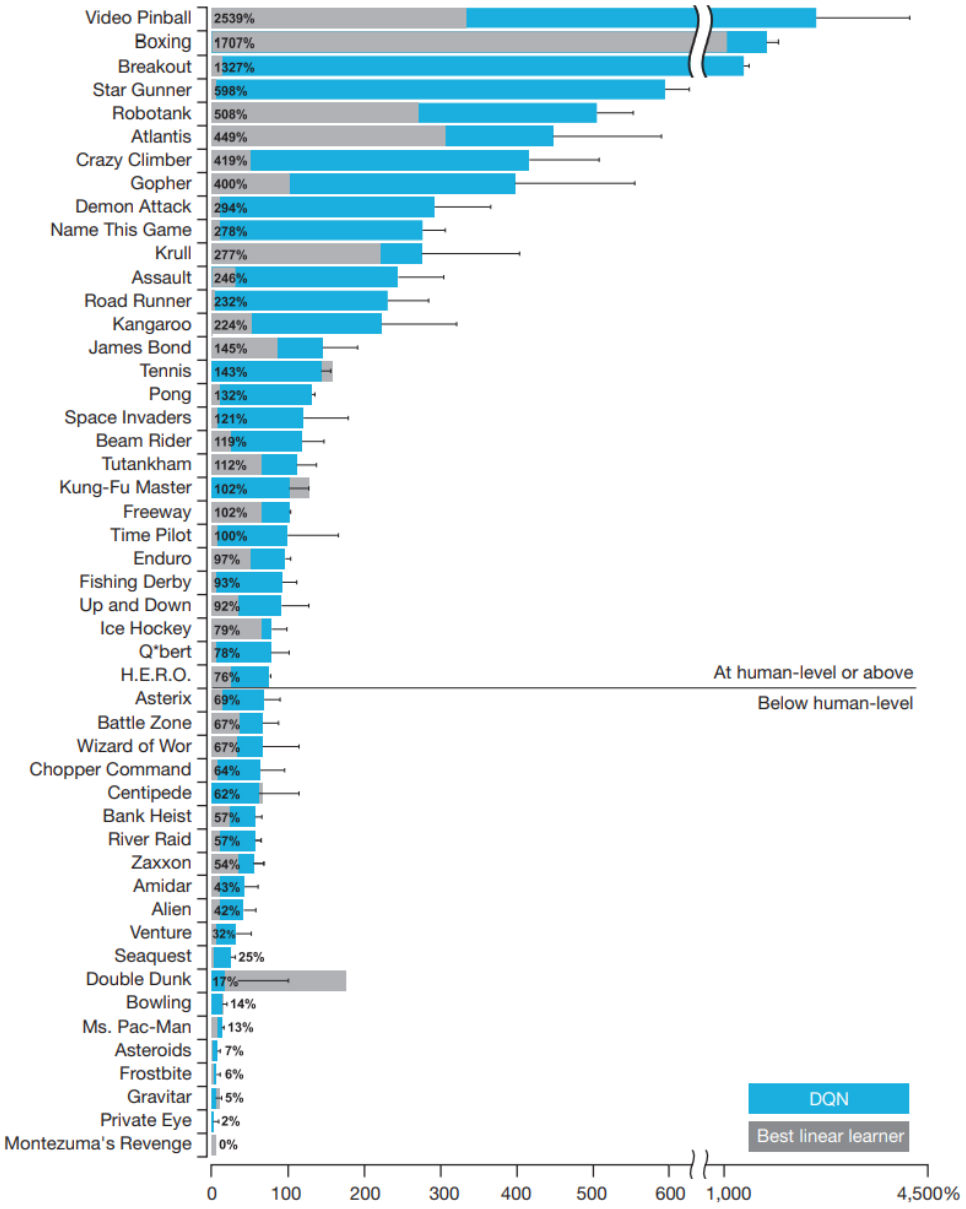


Rys. 1.5. Skuteczność metod DQN i DDQN w grach ATARI [28].

2. Przykłady uczenia przez wzmacnienie w grach

2.1. Gry ATARI

Uczenie przez wzmacnianie przez długi okres czasu pozostawało w cieniu innych metod uczenia maszynowego. Technika ta była bardzo rzadko wykorzystywana, głównie z powodu swoich ograniczeń. Początki tej metody opierały się głównie na metodzie *Q-Learning*, która była bardzo nieefektywna przy bardziej skomplikowanych środowiskach lub nawet niemożliwa do zaimplementowania. Sytuacja zmieniła się diametralnie w 2015, gdy grupa naukowców opublikowała swoją pracę w czasopiśmie naukowym "nature" pt. "Human-level control through deep reinforcement learning". Przedstawili oni wtedy nową metodę *Deep Q-Learning*, które swoje działanie opierała na sieciach neuronowych. Był to przełomowy krok, który pozwolił uogólnić działanie algorytmów uczenia przez wzmacnianie poprzez zastąpienie bardzo nieefektywnej metody tworzenia Q-tablicy poprzez aproksymację funkcji jakości Q przy użyciu sieci neuronowych. W celu przetestowania swojej nowej metody, naukowcy wykorzystali 49 gier Atari, porównując poziom, jaki osiągnął algorytm DQN z poziomem najlepszych, dotychczasowych metod uczenia przez wzmacnianie zaprojektowanych pod konkretne gry oraz zestawili to z poziomem człowieka. Algorytm, w ponad połowie gier osiągnął taki sam lub znacznie wykraczający poziom w porównaniu do człowieka (rys. 2.1). Główna przewaga w zastosowaniu sieci neuronowych w stosunku do Q-tabeli wynika z braku konieczności jawnego określania wszystkich możliwych stanów. Dobrym przykładem są tutaj wcześniej wspomniane gry Atari. Określenie wszystkich możliwych stanów środowiska jako klatka z pojedynczej iteracji gry, nawet w niewielkiej rozdzielczości 84x84x4, wydaje się nierealne i wymagało by ogromnych zasobów pamięciowych. Dlatego taką przewagę wprowadzają sieci neuronowe w uczeniu przez wzmacnianie - umożliwiają określenie rozbudowanego stanu środowiska, który pozwoli algorytmowi lepiej reagować na to, co się dzieje w środowisku, przy jednoczesnej dużej skuteczności w aproksymacji funkcji Q [29].

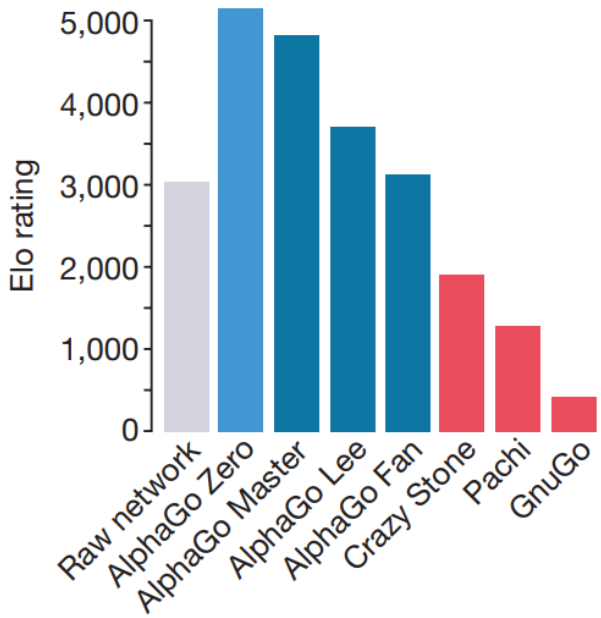


Rys. 2.1. Porównanie algorytmu DQN z najlepszymi dotychczas metodami uczenia przez wzmacnianie i skutecznością człowieka w grach Atari [29]

2.2. AlphaGo, AlphaZero i MuZero

Gra planszowa Go przez bardzo długi okres czasu była poza zasięgiem wszelakich algorytmów sztucznej inteligencji przez swój poziom skomplikowania [8]. Głównym problemem w implementacji takiego algorytmu jest ogromna liczba kombinacji, jaka może zajść podczas rozgrywki, szacowana na około 10^{360}. Liczba kombinacji jest znacznie większa niż w przypadku gry w szachy i wynika ona z zastosowania większej planszy rozgrywki (19x19 zamiast 8x8). W pojedynczej turze, gracz może wykonać 250 ruchów, a standardowa długość rozgrywki jest szacowana na około 150 posunięć obu graczy. Takie liczby sprawiają, że niemożliwe jest zastosowanie typowego podejścia siłowego, aby obliczyć wszystkie

możliwe kombinacje (na dużej głębokości) i wybrać tę, która będzie najlepsza. Mimo tak olbrzymiej ilości kombinacji, naukowcom z DeepMind udało się opracować algorytm bazujący na uczeniu przez wzmacnianie, który jako pierwszy algorytm sztucznej inteligencji był w stanie pokonać czołowych graczy w grę Go. W algorytmie została zaimplementowana sieć neuronowa, która została wytrenowana na 30 mln pozycjach ułożenia elementów na planszach ze 160 000 rzeczywistych gier pochodzących z bazy danych gier Go. Model na początku uczył się wykorzystując technikę uczenia nadzorowanego. Sieć neuronowa otrzymywała konkretną pozycję z gry i miała za zadanie przewidzieć jaki ruch powinna wykonać - porównując to na końcu z rzeczywistym ruchem, jaki w tej pozycji był wykonany przez człowieka. W zależności czy ruch był poprawny, czy też nie - model aktualizował swoje parametry tak, aby w następnych iteracjach przewidywać jeszcze lepiej. Takie podejście pozwoliło w pewien sposób nauczyć algorytm zasad gry i opanować pewną biegłość w przeprowadzaniu rozgrywek. Następnie, model został zestawiony w rozgrywce sam ze sobą, stosując uczenie przez wzmacnianie, aby jeszcze lepiej poprawić swoją grę [30]. Początkowe wersje algorytmu posiadały szacowany rankingu ELO na poziomie 3 000 punktów [31], jednak najlepsi gracze posiadają znacznie wyższy ranking - w granicach 3 600-3 900 punktów ELO [32]. DeepMind zdecydowało się rozwijać swój algorytm. Z algorytmu, który uczył się na rzeczywistych partiach, powstał algorytm *AlphaGo Zero*, który nie miał dostępu do bazy danych partii, tylko rozgrywał partię sam ze sobą, znając jedynie zasady rozgrywki. Takie podejście pozwoliło osiągnąć mu znacznie większą siłę gry, przebijając barierę 5 000 punktów ELO, co dla najlepszych graczy jest nieosiągalnym wynikiem. Porównanie siły gry algorytmów *AlphaGo* i *AlphaGo Zero* zostało przedstawione na rysunku 2.2. W zestawieniu, kolorem czerwonym oznaczone zostały inne algorytmy, stosowane w grze Go, przed powstaniem *AlphaGo*. Ich ranking punktów ELO sięgał zaledwie 2 000 punktów.



Rys. 2.2. Porównanie implementacji algorytmów AlphaZero w zestawieniu z wcześniej istniejącymi algorytmami [31].

Dalszy rozwój algorytmu dążył w kierunku uogólnienia działania. Tym sposobem powstała wersja *AlphaZero* - jeden algorytm, który był w stanie osiągnąć mistrzowski poziom w trzech grach: Go, szachy oraz Shogi (japońska odmiana szachów). Takie podejście znalazło swoje zastosowanie w innych dziedzinach. *AlphaZero* został bowiem wykorzystany w takich obszarach jak chemia [33] czy fizyka kwantowa [34], jako algorytm, który potrafi dostosować swoje działanie w nieznajomym środowisku. DeepMind kontynuowało pracę nad algorytmem w kierunku uogólnienia jego działania. Konsekwencją było powstanie algorytmu *MuZero*, który, w porównaniu do poprzednich algorytmów, jako jedyny nie znał zasad panujących w środowisku. Algorytm rozszerzył również swoje zastosowanie i do listy gier: Go, szachy i Shogi, dodano gry Atari [35].

Na przykładzie algorytmów utworzonych przez DeepMind można wysnuć wniosek, że algorytmy uczenia przez wzmacnianie idą w kierunku uogólnienia swojego działania. Praktyka pokazuje, że takie algorytmy bardzo dobrze odnajdują się w środowiskach o nieznanej dynamice i są w stanie nauczyć się, jak postępować, żeby odnieść założony sukces.

3. Implementacja aplikacji

Głównym celem pracy jest implementacja algorytmu uczenia przez wzmacnianie do prostej gry zręcznościowej. Zamiast wykorzystywać istniejące gry lub korzystać ze środowisk typu OpenAI Gym [36], zdecydowano się zaimplementować własną grę, aby mieć kontrolę nad jak największą liczbą elementów. Własnoręczna implementacja oznacza pełną znajomość kodu źródłowego i zasad działania gry, co umożliwia jej dostosowanie do przeprowadzenia procesu uczenia. Dużą zaletą jest również możliwość jej modyfikacji poprzez upraszczanie lub utrudnianie środowiska, dzięki czemu, trening modelu można rozpocząć od prostszych przypadków, a w raz z nabieranym doświadczeniem można komplikować grę w dowolny sposób.

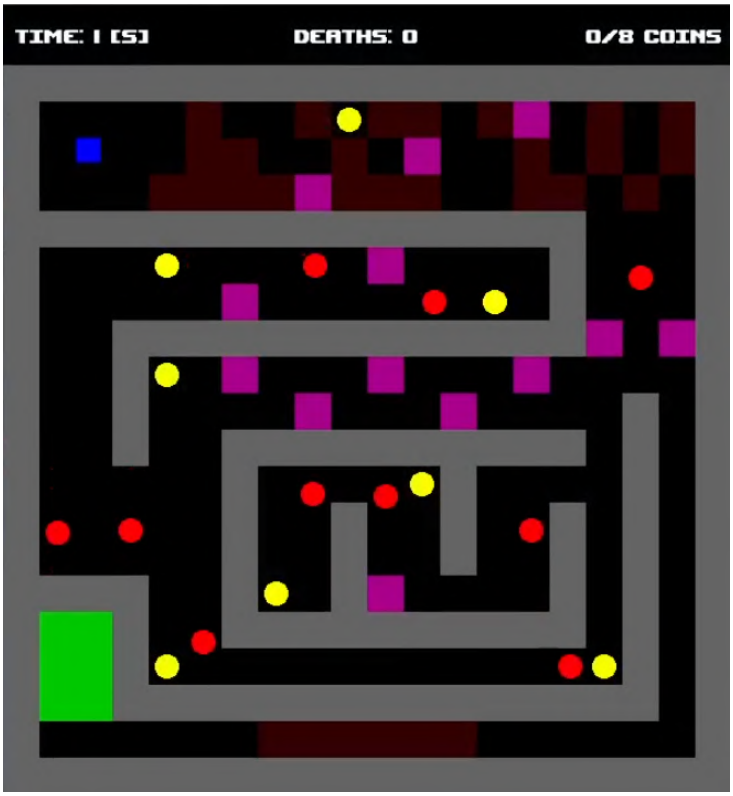
3.1. Gra SmartSquareGame

Opis i zasady gry

SmartSquareGame jest grą 2D, która została napisana przez autora tej pracy w języku C++. Jest ona inspirowana popularną grą przeglądarkową o pt. "The World's Hardest Game" [37]. Zrzut ekranu gry został zaprezentowany na rysunku 3.1.

Cel gry jest prosty - gracz (niebieski element) musi zebrać wszystkie monety występujące na planszy (żółte elementy), a następnie udać się do wyjścia (zielona strefa), aby ukończyć poziom. Zadanie wydaje się łatwe, ale w poziomie są w kilku miejscach są umieszczeni przeciwnicy (czerwone elementy), które poruszają się wertykalnie lub horyzontalnie oraz rozmieszczone są pułapki (różowe elementy). Po kontakcie z przeciwnikiem lub pułapką, poziom jest restartowany, a gracz musi zacząć od początku. Dodatkowym elementem utrudniającym są przeszkody (bordowe elementy), które blokują graczowi przejście. Aby utorować drogę, gracz musi zniszczyć przeszkody, strzelając do nich. Poruszanie gracza odbywa się przy użyciu klawiszy WASD, natomiast strzelanie w odpowiednich kierunkach, za pomocą strzałek. Szare elementy definiują ściany oraz granicę planszy. Nie mogą być one zniszczone, a ani gracz, ani przeciwnik nie może się po nich poruszać.

Gra została zaimplementowana przy wykorzystaniu biblioteki graficznej SFML (Simple and Fast Multimedia Library), która umożliwia stosunkowo łatwą implementacje elementów graficznych oraz umożliwia wykonywanie takich operacji na obiektach jak wykrywanie kolizji. Kod źródłowy gry jest dostępny publicznie na repozytorium GitHub autora pracy [38].



Rys. 3.1. Zrzut ekranu prezentujący grę SmartSquareGame

Struktura projektu

Podczas implementacji gry wykorzystano podejście obiektowe, które pozwala w wygodny sposób zaimplementować silnik gry oraz wszystkie elementy występujące w aplikacji. Na rysunku 3.2 przedstawiono uproszczony diagram klas. Na diagramie nie zawarto wszystkich atrybutów i metod, które zostały zaimplementowane w klasach, a jedynie główny zarys. Opis przedstawionych klas:

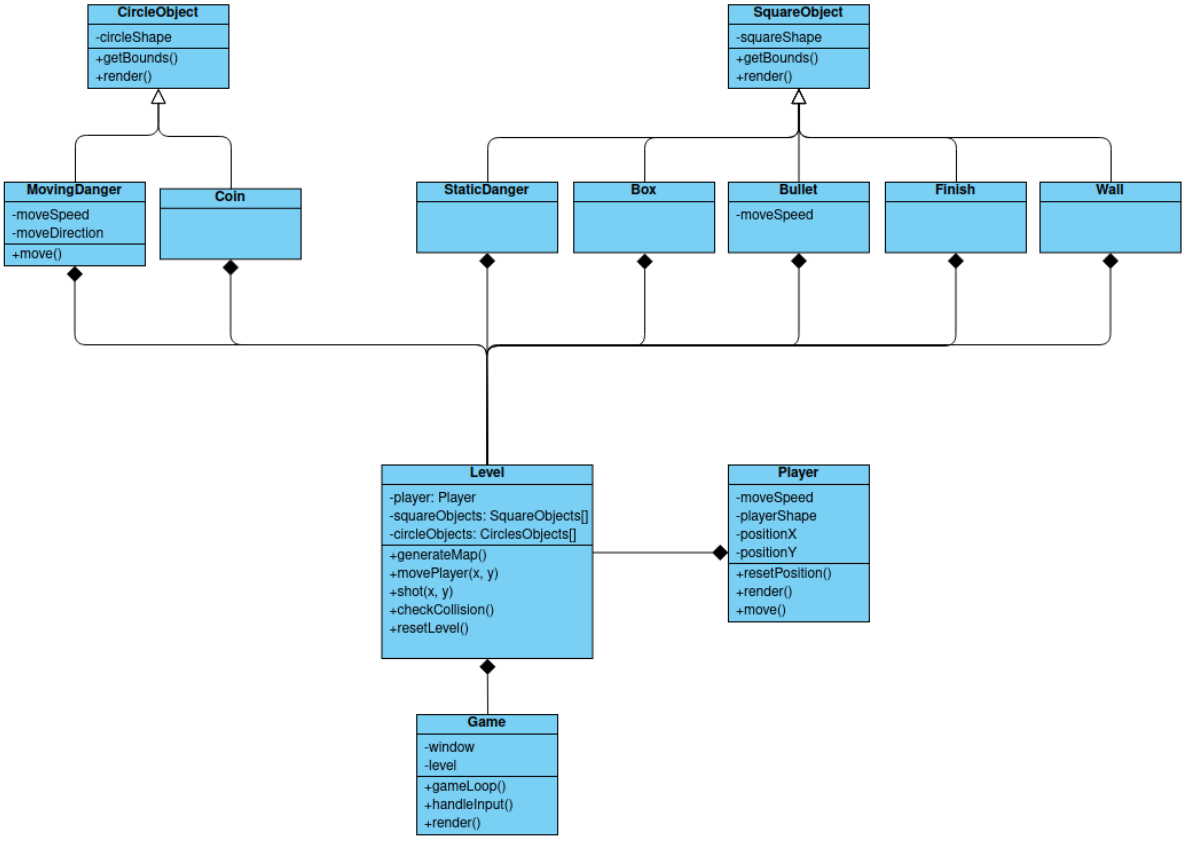
– **Game**: silnik gry odpowiadający za obsługę pętli gry (ang. *gameloop*), tworzenie i obsługę okna, renderowanie obiektów graficznych oraz przechwytywanie wejścia z klawiatury.

– **Level**: klasa odpowiadająca za logikę całego poziomu. Przeprowadzane są w niej obliczenia dotyczące obiektów na aktualnej planszy, takie jak generowanie/usuwanie/poruszanie się obiektów, sprawdzanie kolizji itp. Funkcja *movePlayer()* odpowiada za sprawdzanie kolizji, a jeśli ona nie wystąpiła, to wywołuje funkcję *move()* w klasie Player, która powoduje poruszanie się gracza w odpowiednim kierunku. Dodatkowo, funkcja *shot()* odpowiadająca za strzelanie gracza została umieszczona w tej klasie, a nie w klasie Player, ponieważ klasa Level obsługuje wszystkie obiekty, które znajdują się na planszy - dynamiczne (np. StaticDanger) i statyczne (np. Wall). Przechowywanie i obsługa wszystkich obiektów planszy w jednej klasie jest wygodne z punktu widzenia implementacji.

– **Player**: klasa implementująca gracza. Definiuje jego parametry, takie jak prędkość strzału oraz udostępnia metody umożliwiające np. poruszanie się.

– **CircleObject, SquareObject**: klasy szablonowe, łączące wspólne zachowania obiektów występujących na mapie. Używana do dziedziczenia cech, które powtarzają się we wszystkich tych obiektach (kształt, współrzędne itp.).

Reszta klas to obiekty występujące na mapie takie jak pułapki (*StaticDanger*), przeciwnicy (*MovingDanger*), monety (*Coin*), przeszkody (*Box*), pociski (*Bullet*), wyjście z poziomu (*Finish*), ściana (*Wall*). Wszystkie te klasy dziedziczą z *SquareObject* lub *CircleObject* w zależności od kształtu modelu graficznego.



Rys. 3.2. Uproszczony diagram klas gry SmartSquareGame

Pętla gry

```
void Game::gameLoop() {
    while (this -> window -> isOpen()) {
        if (!gameFinished) {
            this -> update();
            this -> render();
        } else {
            this -> window -> close();
            delete this;
            return;
        }
    }
}

void Game::update() {
    this -> updateWindowEvents();
    this -> updatePlayerInput();
    this -> updateLabels();
    this -> level -> updateBullets();
    this -> level -> updateDangerMovement();

    if (this -> level -> isLevelFinished()) {
        int lastLevelNum = this -> level -> getLevelNumber();
        int mapsCount = this -> level -> getMapsCount();

        if (lastLevelNum + 1 > mapsCount) {
            std::cout << "Player wins \n";
            gameFinished = true;
        } else {
            delete this -> level;
            this -> level = new Level( levelNumber: lastLevelNum + 1);
        }
    }
}

void Game::render() {
    this -> window -> clear();
    this -> level -> renderGameObjects( &: *this -> window);
    this -> renderLabels();
    this -> window -> display();
}
```

Rys. 3.3. Pętla odpowiadająca za działanie gry wraz z funkcjami pomocniczymi.

Na rysunku 3.3 przedstawiono implementację pętli gry. Pętla została rozdzielona na dwie funkcje pomocnicze - *update()* oraz *render()*. Funkcja *update()* odpowiada głównie za obliczenia elementów występujących na planszy, jak np. przechwytywanie zdarzeń związanych z oknem gry (*updateWindowEvents()*) wraz z odświeżaniem statystyk z rozgrywki (*updateLabels()*), przechwytywanie i obliczanie pozycji gracza (*updatePlayerInput()*), obliczanie pozycji pocisków/przeciwników oraz załadowywanie

nowej planszy w razie wygrania aktualnej. Funkcja *render()* odpowiada za wyświetlanie wszystkich obiektów występujących na mapie oraz wyświetlanie pomocniczych statystyk takich jak liczba zebranych monet czy czas gry.

Generowanie mapy

Silnik gry został tak zaprojektowany, aby wczytywać mapy o wymiarach 20x20 z pliku tekstowego, gdzie każdy obiekt występujący na mapie ma przypisany unikatowy numer. Tworzenie plików tekstowych z wartościami numerycznymi nie jest zbyt wygodną i przejrzystą metodą na tworzenie map. Aby uniknąć ręcznego tworzenia plików tekstowych, został utworzony skrypt, który mapuje plik graficzny o wymiarach 20x20 pikseli na plik tekstowy. Każdemu obiektowi w pliku tekstowym został przypisany identyfikator w postaci cyfry od 0 do 9, natomiast w pliku graficznym - kolor RGB. Mapowanie zostało przedstawione w tabeli 3.1. Dodatkowo, zostało zaznaczone rozróżnienie na przeciwników poruszających się horyzontalnie (*MovingDangerH*) oraz wertykalnie (*MovingDangerV*).

Tabela 3.1. Mapowania kolor - ID dla poszczególnych obiektów

Obiekt	**Kolor (R, G, B)**	**ID**
Floor	(255, 255, 255)	0
Wall	(0, 0, 0)	1
Box	(50, 0, 0)	2
StaticDanger	(170, 0, 150)	3
Coin	(255, 255, 0)	4
MovingDangerH	(255, 0, 0)	5
MovingDangerV	(120, 0, 0)	6
Finish	(80, 120, 0)	7
Player	(0, 255, 0)	8

Wygodnymi narzędziami do przygotowywania obrazów w małych wymiarach (20x20 pikseli) są aplikacje do tworzenia tzw. sztuki pikselowej (ang. *Pixel Art*, np. *Pixilart.com*). Można w nich utworzyć własną paletę kolorów z wartościami RGB takie jakie odpowiadają obiektom na mapach. Pozwoli to w wygodny sposób przygotować plansze do rozgrywki.

3.2. Algorytm DDQN

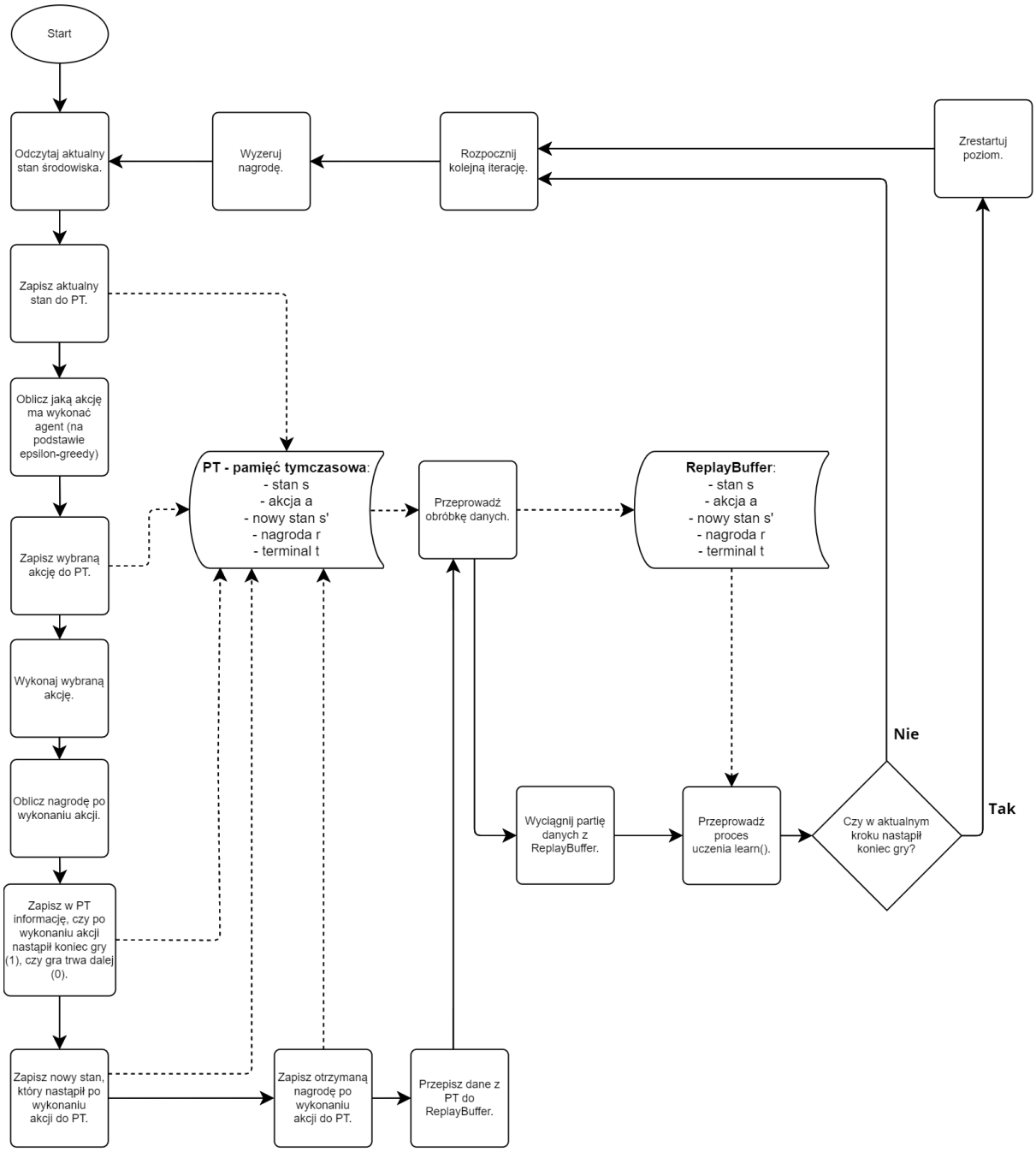
Implementacja algorytmu DDQN

W celu utworzenia sztucznej inteligencji bazującej na uczeniu przez wzmacnianie zaimplementowano algorytm Double Deep Q-Learning. Oznacza to, że zdefiniowane są dwie sieci neuronowe - jedna do przewidywania (sieć ewaluacyjna), a druga do obliczania wartości docelowych (sieć docelowa). Wagi z sieci ewaluacyjnej są kopiowane do sieci docelowej co określoną liczbę kroków algorytmu. Algorytm został zaimplementowany w języku Python przy wykorzystaniu biblioteki TensorFlow oraz interfejsu programistycznego Keras. Dodatkowo, z racji posiadania karty graficznej z rodziny NVIDIA GeForce wyposażone w rdzenie CUDA, możliwe było wykorzystanie biblioteki TensorFlowGPU, co zapewniło szybsze obliczenia. Podczas tworzenia algorytmu, inspirowano się gotowymi implementacjami metod DQN oraz DDQN [39].

Na rysunku 3.4 został przedstawiony proces uczenia podczas pojedynczej iteracji algorytmu. Na początku każdej iteracji, wszystkie dane są zapisywane do tzw. pamięci tymczasowej. Jest to spowodowane tym, że dane muszą przejść pewną obróbkę, zanim trafią do bufora powtórek. Sam bufor powtórek jest zaimplementowaniu przy użyciu tablic z biblioteki *numpy*. Strzałkami przerywanymi zaznaczono operacje polegające na wpisywaniu lub wyciąganiu informacji do/z pamięci (tymczasowej lub do ReplayBuffer).

Istnieją 4 warunki, które definiują kiedy następuje "koniec gry": przekroczenie limitu iteracji na epizod, dotknięcie przeciwnika/pułapki, trzykrotne powtórzenie pozycji lub wygranie planszy. Podczas procesu uczenia, oprócz zwykłej śmierci gracza podczas kontaktu z przeciwnikiem lub pułapką, wskazane jest zaimplementowanie limitu iteracji, jakie może wykorzystać agent w danym epizodzie. Częstym zjawiskiem podczas procesu uczenia jest możliwość zablokowania się gracza np. w momencie, gdy aktualna polityka błędnie wskazuje, że najlepszą akcją do wykonania jest poruszanie się w kierunku, gdzie występuje ściana - agent blokuje się na ścianie i nie wykonuje żadnej innej akcji. W środowisku SmartSquareGame częstym problemem był również problem z zapętleniem się agenta. Powodowało to, że w pewnych momentach agent powtarzał takie akcje jak prawo-lewo-prawo-lewo lub góra-dół-góra-dół. Z tego powodu, został dodany warunek, że jeśli gracz powtórzy trzykrotnie przeciwne akcje (przeciwne akcje = poruszy się w przeciwnych kierunkach), plansza zostaje zrestartowana.

Na rysunku 3.5 przedstawiono implementację metody odpowiedzialnej za uczenie się algorytmu. Przed rozpoczęciem procesu uczenia należy zdefiniować rozmiar partii danych (*BATCH_SIZE*), na których, w każdej iteracji, będzie się uczył algorytm. Metoda *learn()* jest wykonywana w każdej iteracji algorytmu, ale dopiero od momentu, gdy w buforze powtórek uzbiera się więcej danych niż wynosi liczba *BATCH_SIZE*. Po wywołaniu funkcji z bufora wyciągane są próbki stanów, akcji, nagród i statusu zakończenia poziomu (*done*). Próbki są wyciągane z losowego przedziału, który został już wypełniony danymi (niemożliwe jest wyciągnięcie z bufora przedziału, który jest pusty). Każdy przedział ma takie samo prawdopodobieństwo na bycie użytym podczas procesu uczenia. W następnym kroku, obliczany jest tzw. *action_indices*, za pomocą którego będzie możliwe odwołanie się do odpowiednich pozycji w



Rys. 3.4. Uproszczony proces uczenia algorytmu DDQN.

tablicy *state_eval_pred*. Kolejnym etapem jest obliczenie wartości przewidywanych przez sieć neuronową. Tworzone są trzy elementy - *new_state_target_pred*, *new_state_eval_pred* oraz *state_eval_pred*.

– **new_state_target_pred**: obliczane na podstawie stanu s' przy użyciu sieci docelowej (*neural_network_target*)

– **new_state_eval_pred**: obliczane na podstawie stanu s' przy użyciu sieci ewaluacyjnej (*neural_network_eval*)

– **state_eval_pred**: obliczane na podstawie stanu s (stanu, który był bezpośrednio przez stanem s') przy użyciu sieci ewaluacyjnej (*neural_network_eval*)

```
def learn(self):
    if self.memory.buffer_index > BATCH_SIZE:
        state, action, reward, new_state, done = self.memory.get_sample_batch()

        action_values = np.array(self.action_space, dtype=np.int8)
        action_indices = np.dot(action, action_values)

        new_state_target_pred = self.neural_network_target.predict(new_state)
        new_state_eval_pred = self.neural_network_eval.predict(new_state)
        state_eval_pred = self.neural_network_eval.predict(state)

        max_actions = np.argmax(new_state_eval_pred, axis=1)

        batch_index = np.arange(BATCH_SIZE, dtype=np.int32)
        state_eval_pred[batch_index, action_indices] = reward + self.gamma * new_state_target_pred[
            batch_index, max_actions.astype(int) * done]

        self.neural_network_eval.fit(state, state_eval_pred, verbose=0)

        if self.memory.buffer_index % self.replace_target == 0:  # replace every 100 iteration by default
            self.neural_network_target.set_weights(self.neural_network_eval.get_weights())
```

Rys. 3.5. Funkcja odpowiadająca za uczenie się algorytmu.

Na wejście sieci neuronowych są podawane partie danych (tzw. *batch*). W tym przypadku, pojedyncza partia danych składa się z N kolejnych stanów pochodzących z bufora powtórek i jest to tensor o wymiarach $N \times M \times M \times O$, gdzie $M \times M$ to wymiar planszy w danym stanie (standardowo jest to 20×20), a O jest to pojedynczy obiekt występujący na mapie opisany poprzez kodowanie "1 z n". Zmienne utworzone za pomocą funkcji *predict* na sieciach neuronowych to nic innego jak obliczone wartości akcji w podanych stanach. Przy wykorzystaniu N stanów na wejściu modelu, sieć neuronowa zwróci macierz o wymiarach $N \times A$, gdzie A to liczba możliwych do podjęcia akcji. Obliczone wartości oznaczają prawdopodobieństwa wszystkich możliwych akcji w kolejnych N stanach. Wybranie akcji o najwyższym prawdopodobieństwie (operacja *np.argmax()*) maksymalizuje szansę na wygranie gry. Po uzyskaniu najbardziej wartościowych akcji, możliwe jest przeprowadzenie procesu uczenia. W pierwszym kroku, następuje aktualizacja tablicy *state_eval_pred* na podstawie przewidywań z sieci ewaluacyjnej, sieci docelowej, otrzymanych nagród oraz parametru gamma. Następnie, przeprowadzany jest trening sieci neuronowej (*neural_network_eval*) przy użyciu funkcji *fit()*. Na zakończenie metody learn(), sprawdzany jest aktualny liczba wpisów do bufora i jeśli jest ona podzielna przez parametr *replace_target*, to wagi z sieci ewaluacyjnej są przepisywane do sieci docelowej. W tym przypadku parametr *replace_target* wynosi 100, co oznacza, że co 100 iteracji wagi zostaną przepisane.

Uproszczenie problemu

Środowisko SmartSquareGame jest dosyć rozbudowane i implementacja algorytmu uczenia przez wzmocnienie wydaje się trudnym i czasochłonnym zadaniem. Z tego powodu zrezygnowano z kilku funkcjonalności, które oferuje gra i skupiono się na implementacji algorytmu w uproszczonym środowisku. Z gry zostały usunięte obiekty takie jak: przeciwnicy, pułapki i pudła, przez co przestrzeń akcji zmniejszyła się z 8 do 4 (strzelanie przestało być potrzebne). Agent ma więc za zadanie tylko zebrać wszystkie monety poziomie i dojść do wyjścia.

Stany, akcje, nagrody

Stany

Odpowiednie dobranie stanu środowiska nie zawsze jest trywialnym zadaniem. Prawidłowa definicja stanu gry jest kluczowa, aby agent mógł w poprawny sposób nauczyć się jak reagować w danej sytuacji. Agent w środowisku SmartSquareGame musi nauczyć się dobierania odpowiedniej trasy z punkt A (początku planszy) do punktu B (wyjścia z poziomu), po drodze zbierając monety. Z tego powodu nasuwa się pomysł, aby stan gry przekazywać jako zrzut aktualnego rozmieszczenia elementów na planszy. Mapa składa się siatki pól 20x20, gdzie każdy pole mapy ma wymiary 30x30 pikseli. Gdyby jako stan środowiska przesyłać cały zrzut mapy jako zrzut ekranu, stan gry miałby wymiary 600x600x7, co jest dużą liczbą. Wykorzystując fakt, że każdy element na mapie ma stałe wymiary 30x30 pikseli, możliwe jest zmapowanie całej planszy i przedstawienie jako macierz 20x20. Ponieważ każdy obiekt, który występuje na mapie, ma przypisany identyfikator (ściana - 0, gracz - 8, moneta - 4 itd.), można to wykorzystać w definiowaniu aktualnego stanu środowiska. Przykładowy stan został przedstawiony na rysunku 3.6. W każdej iteracji gry, wystarczy odświeżać jedynie pozycję gracza i usuwać zebrane monety, a tak utworzony stan przesyłać do algorytmu.

Należy zwrócić uwagę, że stan środowiska jest reprezentowany przez zbiór danych kategorialnych. Przed użyciem tych danych w procesie uczenia algorytmu, konieczne jest wprowadzenie kodowania do formy "1 z n" (ang. *One-hot Encoding*) [40]. Metoda ta dobrze się sprawdza w przypadku kodowania małej liczby elementów. Dla środowiska SmartSquareGame, mapowanie stanu zostało przedstawione w tabeli 3.2.

Należy zwrócić uwagę, że w powyższym mapowaniu zostało uwzględnione rozróżnienie na podłogę, która została odwiedzona przez gracza oraz tę, której gracz nie odwiedził. Zostało to zaimplementowane w celu zachęcenia agenta do odkrywania nowych fragmentów planszy, w których jeszcze nie odwiedził. Dalsze wyjaśnienie tej funkcjonalności zostało zawarte w podrozdziale o nagrodach.

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 1 0 0 0 4 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 1
1 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 1
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1
1 0 0 0 4 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 1 0 0 0 0 4 0 0 0 0 0 0 1
1 7 7 0 0 0 0 1 0 0 0 0 0 0 0 0 8 0 0 1
1 7 7 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Rys. 3.6. Stan środowiska

Tabela 3.2. Kodowanie stanów ID - "1 z n"

Obiekt	**ID**	**Kod "1 z n"**
Podłoga (nieodkryta)	0	[0, 0, 0, 0, 0, 1]
Ściana	1	[0, 0, 0, 0, 1, 0]
Moneta	4	[0, 0, 0, 1, 0, 0]
Wyjście z poziomu	7	[0, 0, 1, 0, 0, 0]
Gracz	8	[0, 1, 0, 0, 0, 0]
Podłoga (odkryta)	9	[1, 0, 0, 0, 0, 0]

Akcje

Akcje, które zostały zdefiniowane to poruszanie się w górę, dół, prawo i lewo. Akcje również są kodowane za pomocą kodu "1 z n" (tabela 3.3) w analogiczny sposób jak stan środowiska. Aktualna implementacja nie przewiduje wykonywania dwóch akcji w jednym kroku, co powoduje, że agent nie może poruszać się na skos.

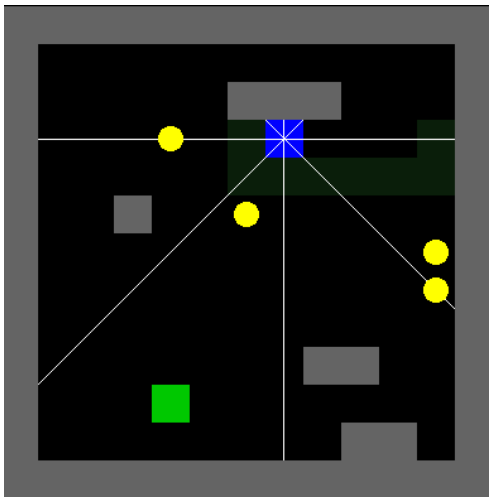
Nagrody

Przed zdefiniowaniem listy nagród, w środowisku zaimplementowano dwie funkcjonalności (rys. 3.7). Pierwszą z nich jest odkrywanie mapy. Każdy fragment podłogi posiada przypisaną zmienną typu

Tabela 3.3. Kodowanie akcji "1 z n"

Akcja	ID	Kod "1 z n"
Ruch w górę	0	[0, 0, 0, 1]
Ruch w prawo	1	[0, 0, 1, 0]
Ruch w dół	2	[0, 1, 0, 0]
Ruch w lewo	3	[1, 0, 0, 0]

prawda/fałsz, czy dany fragment podłogi został odkryty przez gracza. Rozróżnienie na odkryte/nieodkryte fragmenty mapy jest uwzględnione w przesyłanym stanie. Drugą funkcjonalnością jest pole widzenia (ang. *Field of View*, w skrócie FoV), które pozwala przyspieszyć nauczenie agenta zbierania monet poprzez przyznawanie niewielkich nagród, gdy agent zbliża się do monety w linii prostej. Pole widzenia jest zaimplementowane w taki sposób, że agent generuje zadaną liczbę wektorów dookoła swojej postaci. Początek wektorów znajduje się w środku agenta, natomiast kończy w momencie napotkania przeszkody (w tym przypadku ściany).



Rys. 3.7. Ciemnozielone fragmenty mapy opisują miejsca odwiedzone przez gracz. Białym kolorem zaznaczone są promienie pola widzenia.

W tabeli 3.4 przedstawiono wykaz nagród za poszczególne akcje. Nagroda w danej iteracji jest sumowana, co oznacza, że w jednym kroku algorytmu agent może otrzymać kilka cząstkowych nagród - np. jedną za dotknięcie ściany (-20), a drugą za oddalenie się od monety, gdy moneta była w zasięgu widzenia (-5), w konsekwencji czego, do bufora powtórek zostanie wpisana nagroda -25. Największe nagrody są za zbieranie monet i przejście poziomu, aby agent był jak najbardziej zachęcany do wykonywania tych akcji. Niewielkie nagroda jest za odkrywanie fragmentów map. Wynika to z tego, że agent powinien być zachęcany do eksplorowania mapy, ale nagroda za to powinna być znacznie mniejsza niż za zebranie monety czy przejście poziomu. Gdyby nagroda za odkrywanie planszy była większa, mogłoby dochodzić do sytuacji, gdy agent przed przejściem poziomu stara się najpierw odkryć całą mapę, zamiast tylko

zebrać monety. Nagrody związane ze zbliżaniem się i oddalaniem od monet powodują przyspieszenie procesu uczenia, ponieważ zachęcają algorytm do zbierania monet, gdy znajdą się na linii z monetą w płaszczyźnie poziomej lub pionowej (lub ewentualnie po skosie). Agent prędzej ustawi się na linii z jakąś monetą niż przypadkowo ją zbierze i dowie się, że za monetę jest nagroda +75. Zostało to sprawdzone eksperymentalnie i w przypadku zastosowania nagród za zbliżanie się do monety oraz kar za oddalanie się - agent szybciej nauczał się zbierać monety na planszy. Kara za dotknięcie ściany jest zdefiniowana, aby zniechęcić agenta do blokowania się na ścianach. Ujemna nagroda za przegranie poziomu jest zależna od liczby zebranych monet w danym epizodzie i jest przyznawana tylko w pojedynczej (ostatniej) iteracji epizodu, dzięki czemu nie powinna ona zniechęcać agenta do zbierania monet.

Tabela 3.4. Wykaz nagród dla algorytmu DDQN

Akcja	**Wartość nagrody**
Zebranie monety	+75
Przejście poziomu	+500
Odkrycie nowego fragmentu mapy	+1
Zbliżanie się do monety, gdy moneta jest w FoV	+1
Oddalanie się od monety, gdy moneta jest w FoV	-5
Zbliżanie się do wyjścia, gdy wyjście jest w FoV i wszystkie monety zebrano	+3
Oddalanie się od wyjścia, gdy wyjście jest w FoV i wszystkie monety zebrano	-10
Dotknięcie ściany	-20
Przegranie poziomu, gdy zebrano N monet	-300 + (N · (-125))
Przegranie poziomu, gdy zebrano wszystkie monety	-500

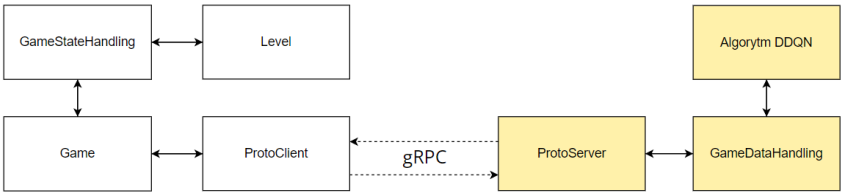
Wartości przedstawione w tabeli 3.4 zostały dobrane eksperymentalnie. Zostały przetestowane zarówno różne wartości nagród i kar jak i ilość akcji, za które przysługuje nagroda/kara. Wartości, które zostały przedstawione mogą nie być optymalne, jednak uzyskiwały one najlepsze rezultaty ze wszystkich testowanych scenariuszy.

3.3. Integracja gry z algorytmem AI

Architektura klient-serwer

Z powodu użycia dwóch różnych języków programowania (implementacja gry - C++, algorytm uczenia przez wzmacnianie - Python) konieczne było zapewnienie komunikacji między tymi dwoma aplikacjami - algorytmem i grą. W pracy zdecydowano się zaimplementować komunikację w architekturze klient-serwer przy użyciu biblioteki gRPC (Remote Procedure Call). Wybrano takie rozwiązanie, ponieważ biblioteka gRPC zapewnia komunikację wieloplatformową z bardzo niskim narzutem, co jest bardzo ważne w algorytmie, w którym jest wykonywane wiele operacji na sekundę. Szybkość jest kluczowa - sama komunikacja nie powinna stanowić "wąskiego gardła" w procesie uczenia.

W projekcie zastosowano implementację o nazwie *Unary* [41], co w uproszczeniu oznacza komunikację naprzemienną między klientem, a serwerem. Klient, po wysłaniu zapytania do serwera czeka na odpowiedź i nie może wysłać kolejnego zapytania, dopóki tej odpowiedzi nie dostanie. Klientem w tym przypadku jest gra SmartSquareGame, natomiast serwerem - algorytm uczenia przez wzmacnianie. Z obu aplikacji zostały wyizolowane części komunikacyjne (klasy *ProtoServer* oraz *ProtoClient*) oraz części odpowiedzialne za przetwarzanie danych używanych podczas treningu (klasy *GameStateHandling* oraz *GameDataHandling*). Diagram części komunikacyjnej został przedstawiony na rysunku 3.8.

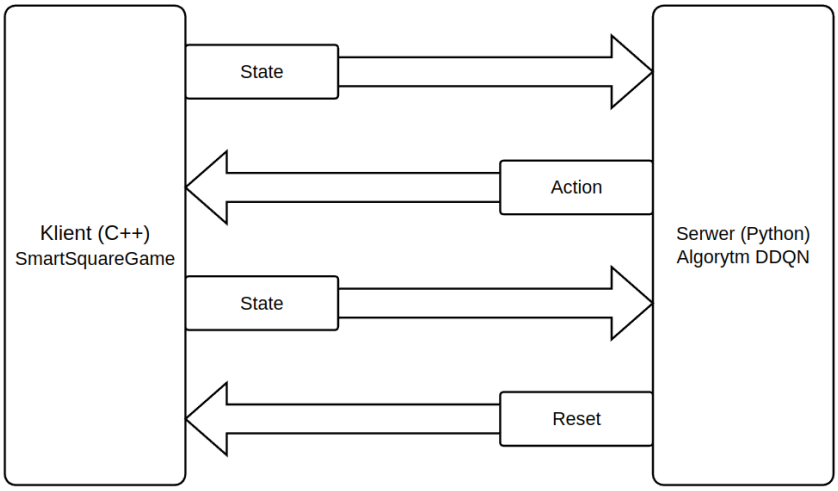


Rys. 3.8. Wydzielone części komunikacyjne oraz klasy odpowiedzialne za przetwarzanie danych. Kolorem białym oznaczono część zaimplementowaną w języku C++, natomiast żółtym - w języku Python.

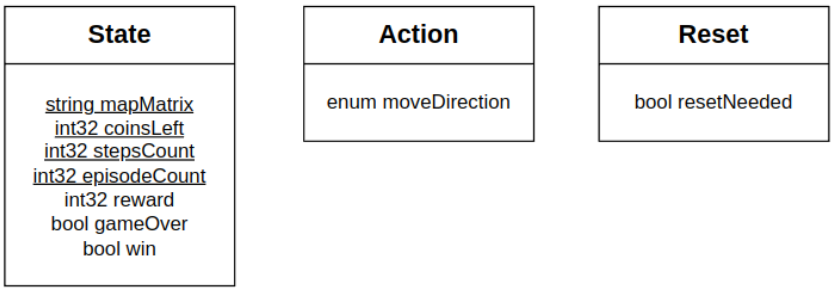
Struktura i przepływ wiadomości

gRPC domyślnie korzysta z *Protocol Buffers* zarówno do opisu interfejsu usług jak i definiowania struktury wiadomości [41]. W projekcie zdefiniowano dwie usługi - *StateAction* oraz *StateReset*. Nazwy usług zostały utworzone na podstawie nazw wiadomości jakie są wymieniane podczas wywoływania konkretnej (rys. 3.9). W momencie korzystania z usługi *StateAction*, jako zapytanie do serwera wysyłana jest wiadomość "State", a serwer odpowiada wiadomością "Action". W usłudze *StateReset*, zapytanie jest również wiadomością o strukturze "State", natomiast odpowiedź z serwera jest w strukturze "Reset". Zawartość struktur poszczególnych wiadomości przedstawiono na rysunku 3.10.

W usłudze *StateAction* w wiadomości "State" ustawiane są tylko pola, które zostały podkreślone na rysunku 3.10, natomiast w usłudze *StateReset*, przesyłane są już wszystkie informacje. Najważniejszym polem wiadomości "State" jest *mapMatrix*. Jest to stan środowiska, skonwertowany do postaci ciągu znaków, gdzie każdy wiersz macierzy zawierającej zrzut planszy jest odseparowany znakiem #. Tak otrzymany stan, przed wpisaniem do ReplayBuffer zostaje przekonwertowany do tablicy numpy. Przesyłanie wielowymiarowych tablic przy użyciu *Protocol Buffers* jest dosyć nieintuicyjne, dlatego zdecydowano się na parsowanie danych po stronie serwera. Kolejnym bardzo ważnym polem jest *reward*, czyli nagroda po wykonaniu akcji. Pola *stepsCount* oraz *episodeCount* oznaczają kolejno liczbę iteracji algorytmu oraz liczbę epizodów, natomiast *gameOver* oraz *win* informują serwer o zakończeniu danego epizodu (poprzez porażkę lub zwycięstwo). Ostatnie pole - *coinsLeft* wskazuje ile jeszcze zostało monet na planszy do zebrania. Logowanie informacji z procesu uczenia odbywa się po stronie serwera - dlatego potrzebne są dodatkowe informacje oprócz nagrody i stanu.



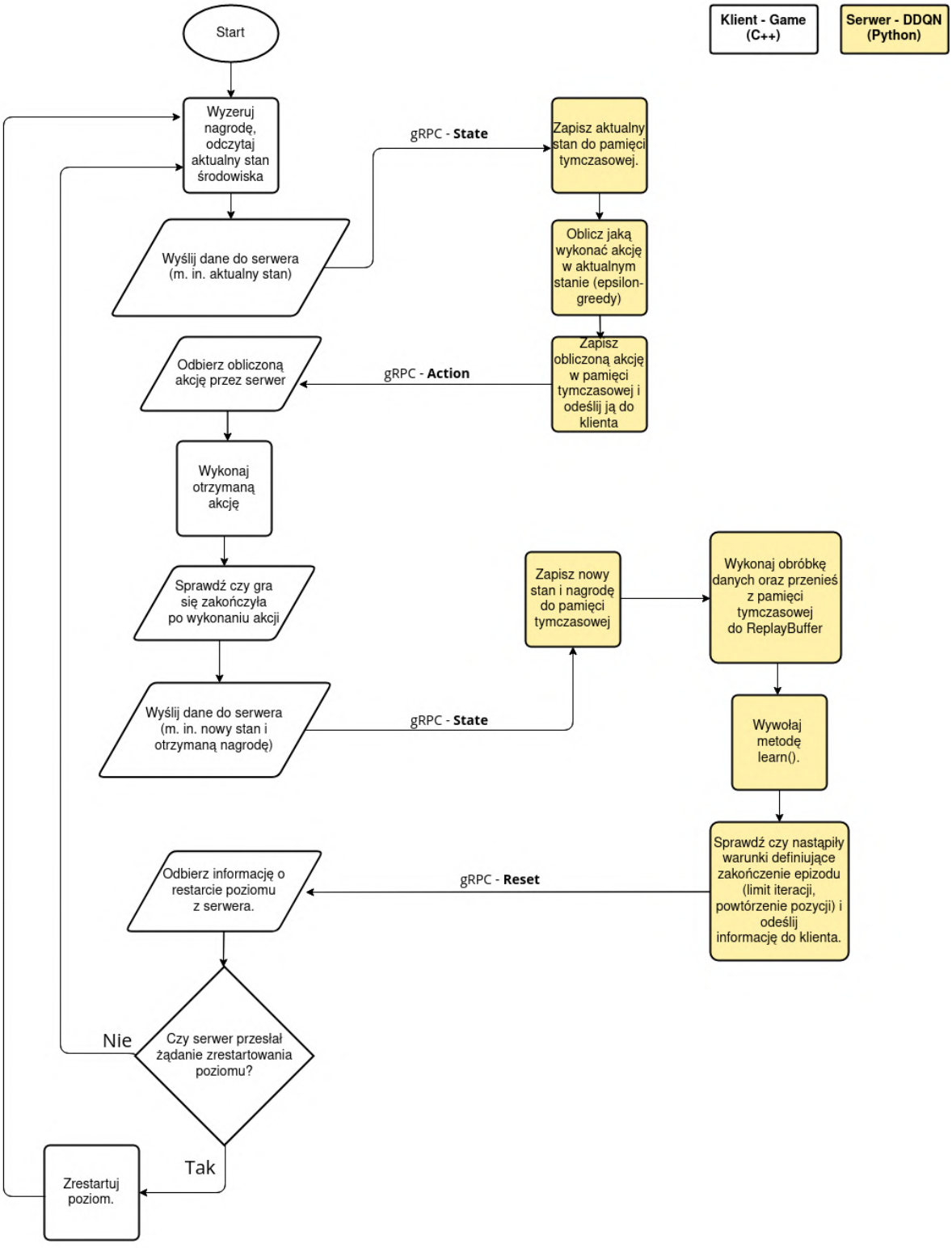
Rys. 3.9. Wymiana wiadomości za pomocą protokołu gRPC podczas pojedynczej iteracji algorytmu.



Rys. 3.10. Struktura poszczególnych wiadomości gRPC.

Po wysłaniu pierwszej wiadomości stanu (za pomocą usługi *StateAction*) serwer zwraca akcję, jaką ma wykonać agent. Akcja może być wybrana w sposób losowy lub obliczona na podstawie aktualnej polityki - wszystko zależy od aktualnych wartości algorytmu $\epsilon - greedy$. Po wykonaniu akcji, następuje kolejna komunikacja z serwerem (za pomocą usługi *StateReset*), gdzie znowu jest przekazywany stan (wraz z nagrodą) - tym razem już po wykonaniu akcji, a serwer zwraca informację, czy z punktu widzenia algorytmu uczenia jest potrzebny restart środowiska (planszy). Wiadomość Action składa się z jednego pola, zdefiniowanego jako wartość typu wyliczeniowego (UP - góra, DOWN - dół, RIGHT - prawo, LEFT - lewo).

Cała wymiana informacji wraz z uproszczonym procesem uczenia została przedstawiona na rysunku 3.11. Żółtym kolorem oznaczono operacje, które są wykonywane po stronie serwera, a białym - po stronie klienta.



Rys. 3.11. Proces uczenia algorytmu wraz z wymianą informacji w architekturze klient-serwer.

3.4. Uczenie równoległe

Proces uczenia algorytmu przez wzmacnianie jest długotrwały i wymaga wielu prób. Każde środowisko jest inne i wymaga doboru hiperparametrów w sposób empiryczny. Z tego powodu, w projekcie zaimplementowano mechanizm, który pozwala na uczenie wielu modeli jednocześnie. W tym celu został utworzony plik z rozszerzeniem json (*learning_parameters.json*), w którym należy zdefiniować hiperparametry modelu. W tym pliku, każdy z hiperparametrów posiada przypisaną listę. Jeśli do listy zostanie wpisana tylko jedna wartość - wtedy zostanie ona zastosowana do wszystkich modeli. Jeśli dany parametr ma zostać przetestowany w kilku wersjach - wtedy należy dodać ich tyle, ile modeli będzie testowane równolegle. Na rysunku 3.12 przedstawiono przykład konfiguracji pliku (*learning_parameters.json*) do testowania szybkości uczenia *learning_rate*.

```
{
  "iter_per_episode": [150],
  "target_episodes": [2000],
  "batch_size": [64],
  "mem_size":  [50000],
  "network_layers": [2],
  "neurons":  [[1800, 1500]],
  "huber_delta": [1.35],
  "learning_rate":  [0.01, 0.003, 0.001, 0.0003, 0.0001, 0.00003],
  "gamma": [0.85],
  "epsilon":  [1.0],
  "epsilon_decay": [0.999964],
  "epsilon_min":  [0.01],
  "replace_target": [100]
}
```

Rys. 3.12. Definiowanie parametrów testowych - plik *learning_params.json*

Uruchamiając 6 instancji aplikacji, każda z nich otrzyma inną wartość szybkości uczenia, a reszta hiperparametrów będzie taka, jaka została zdefiniowana w tym pliku w liczbie pojedynczej. Możliwe jest testowanie kilku parametrów na raz - wtedy analogicznie należy podać tyle wartości, ile instancji zostanie uruchomionych. Jest to jednak niezalecane, ponieważ wyniki takiego uczenia będą niejasne - nie będzie wiadomo, który z parametrów powoduje potencjalny problem z procesem uczenia.

Aplikacja do komunikacji korzysta z biblioteki gRPC, dla której domyślnym portem jest port 50051. W przypadku uczenia równoległego, każda z instancji aplikacji komunikuje się na innym porcie - w konsekwencji, konieczne jest zdefiniowanie portów, na których klient oraz serwer będą się komunikować. Dodatkowym parametrem, który musi zostać podany jest identyfikator instancji, oznaczony w serwerze jako *WORKER_ID*. Na podstawie tego numeru, serwer będzie wiedział, który parametr z pliku *learning_params.json* powinien użyć. Dodatkowo, na podstawie tego parametru tworzona jest nazwa pliku z logami oraz nazwy plików, do których zapisywane będą zrzuty sieci neuronowych. Należy zwrócić uwagę na fakt, że numerowanie instancji należy rozpocząć od wartości 0, ponieważ algorytm odwołuje się do konkretnych miejsc w tablicach w pliku *learning_params.json*, a te są numerowane od 0. Na

rysunkach 3.13 oraz 3.14 przedstawiono skrypty, które służą do uruchamiania pięciu instancji jednocześnie.

```
python ProtoServer.py --WORKER_ID 0 --PORT 50051 &
python ProtoServer.py --WORKER_ID 1 --PORT 50052 &
python ProtoServer.py --WORKER_ID 2 --PORT 50053 &
python ProtoServer.py --WORKER_ID 3 --PORT 50054 &
python ProtoServer.py --WORKER_ID 4 --PORT 50055
```

Rys. 3.13. Skrypt uruchamiający 5 instancji serwerów.

```
SmartSquareRL/cmake-build-release/SmartSquareRL 50051 &
SmartSquareRL/cmake-build-release/SmartSquareRL 50052 &
SmartSquareRL/cmake-build-release/SmartSquareRL 50053 &
SmartSquareRL/cmake-build-release/SmartSquareRL 50054 &
SmartSquareRL/cmake-build-release/SmartSquareRL 50055
```

Rys. 3.14. Skrypt uruchamiający 5 instancji klientów.

Każda z instancji loguje informacje z przebiegu uczenia do odpowiedniego pliku ze swoim numerem *WORKER_ID*. Plik z logami w każdej kolejnej linii zawiera informacje o poszczególnych epizodach takie jak - całkowita nagroda za epizod, numer epizodu, liczba monet, która nie została zebrana, wartość parametru ϵ, liczba kroków wykonanych w epizodzie oraz informacja, czy epizod został wygrany. Fragment pliku z logami został przedstawiony na rysunku 3.15. Dodatkowo, co określoną liczbę epizodów (domyślnie 1000), zapisywany jest stan sieci neuronowej, tak, aby w razie wystąpienia problemów, można było odtworzyć proces uczenia od konkretnego momentu.

```
DEBUG:root:Reward: -460, Ep: 3159, Coins left: 1, Epsilon: 0.2607014034718483, Steps in episode: 30, [LOSE]
DEBUG:root:Reward: -601, Ep: 3160, Coins left: 2, Epsilon: 0.2606662109203799, Steps in episode: 10, [LOSE]
DEBUG:root:Reward: 611, Ep: 3161, Coins left: 0, Epsilon: 0.2606134310005505, Steps in episode: 15, [WIN]
DEBUG:root:Reward: -380, Ep: 3162, Coins left: 1, Epsilon: 0.2605782503246444, Steps in episode: 10, [LOSE]
DEBUG:root:Reward: 622, Ep: 3163, Coins left: 0, Epsilon: 0.260543074397841, Steps in episode: 10, [WIN]
DEBUG:root:Reward: -355, Ep: 3164, Coins left: 1, Epsilon: 0.2605043863628058, Steps in episode: 11, [LOSE]
DEBUG:root:Reward: -625, Ep: 3165, Coins left: 2, Epsilon: 0.26043757510563686, Steps in episode: 19, [LOSE]
DEBUG:root:Reward: -548, Ep: 3166, Coins left: 2, Epsilon: 0.26041648037401166, Steps in episode: 6, [LOSE]
DEBUG:root:Reward: -572, Ep: 3167, Coins left: 2, Epsilon: 0.26038484148018487, Steps in episode: 9, [LOSE]
DEBUG:root:Reward: -393, Ep: 3168, Coins left: 0, Epsilon: 0.26036023610920045, Steps in episode: 7, [LOSE]
DEBUG:root:Reward: 604, Ep: 3169, Coins left: 0, Epsilon: 0.2602688650861231, Steps in episode: 26, [WIN]
DEBUG:root:Reward: -441, Ep: 3170, Coins left: 0, Epsilon: 0.26013889237408205, Steps in episode: 37, [LOSE]
DEBUG:root:Reward: 608, Ep: 3171, Coins left: 0, Epsilon: 0.260054620456904, Steps in episode: 24, [WIN]
DEBUG:root:Reward: -527, Ep: 3172, Coins left: 0, Epsilon: 0.2599352819708554, Steps in episode: 34, [LOSE]
DEBUG:root:Reward: -396, Ep: 3173, Coins left: 1, Epsilon: 0.25979846114259864, Steps in episode: 39, [LOSE]
DEBUG:root:Reward: 598, Ep: 3174, Coins left: 0, Epsilon: 0.2596722289153299, Steps in episode: 36, [WIN]
```

Rys. 3.15. Fragment pliku z logami zebranymi w czasie procesu uczenia.

Implementując mechanizm uczenia równoległego należy zwrócić uwagę na kwestię techniczną - rozmiar bufora (*ReplayBuffer*). Z jednej strony, powinien być on stosunkowo duży - aby pomieścić jak najwięcej iteracji, z których algorytm będzie mógł korzystać podczas procesu uczenia. Z drugiej strony,

gdy rozmiar bufora będzie zbyt duży, konsekwencją będzie całkowite zapełnienie się pamięci RAM. Ten problem może również wystąpić podczas uczenia jednej instancji - wszystko zależy jak dużą pamięcią dysponuje komputer/serwer, na którym proces uczenia jest przeprowadzany, jednak w przypadku jednej instancji, problem jest trudniejszy do zauważenia, ponieważ do zapełnienia całej pamięci potrzeba więcej iteracji algorytmu. W takim przypadku, można przeoczyć dobranie zbyt dużego rozmiaru bufora, ponieważ dla jednego środowiska bufor może nie zdążyć się jeszcze zapełnić do krytycznej wartości (proces uczenia skończy się przed zapełnieniem bufora), ale kilka instancji, może spowodować całkowite zapełnienie pamięci. W przypadku środowiska SmartSquareGame, zaimplementowany bufor składa się z 5 tablic numpy: dwa bufory stanów - przed i po wykonaniu danej akcji (każdy o wymiarach Nx20x20x6), bufor akcji (Nx4), bufor nagród (N) oraz bufor stanów końcowych (N), gdzie N - rozmiar bufora, czyli liczba iteracji, które może pomieścić. Analizując wykorzystaną ilość pamięci RAM podczas procesu uczenia zauważono, że każde 1000 iteracji zwiększa zużycie o około 4 MB. Dodatkowo, do całego zużycia pamięci należy doliczyć około 1200 MB działania pojedynczego serwera z uruchomionym algorytmem uczenia przez wzmacnianie (zużycie instancji klienta wynosi zaledwie 20 MB) oraz zużycie systemu operacyjnego (Linux) - około 1 GB. Na podstawie tych danych można wykonać przykładowe obliczenia - 5 instancji aplikacji, gdzie każda posiada bufor o rozmiarze 75 000 iteracji będzie generowała zużycie 8,6 GB pamięci RAM (ok. 1,5 GB na instancję). Podczas uczenia algorytmu użyto niewłaściwych typów danych, co prowadziło do nieoptymalnego zużycia pamięci, jednak nie miało to żadnego negatywnego wpływu na wyniki osiągane przez algorytm.

Drugą ważną kwestią techniczną jest pamięć karty graficznej. Jeśli podczas procesu uczenia używana jest biblioteka TensorFlowGPU, należy pamiętać o wyłączeniu opcji automatycznego przydzielania całej pamięci GPU (rys. 3.16). W przeciwnym wypadku, podczas uruchamiania kilku instancji, pierwsza z nich otrzyma pełną pamięć karty graficznej do dyspozycji, a na innych instancji wystąpi problem braku pamięci (ang. *Out of memory*).

```
gpu_devices = tf.config.experimental.list_physical_devices('GPU')
for device in gpu_devices:
    tf.config.experimental.set_memory_growth(device, True)
```

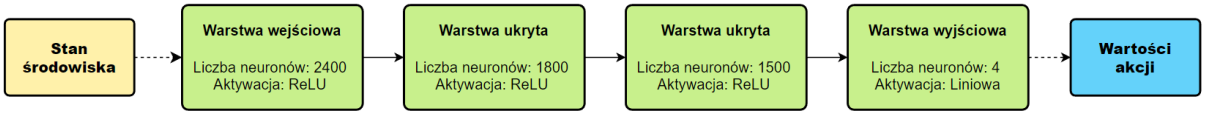
Rys. 3.16. Wyłączenie opcji automatycznego przydzielania całej pamięci GPU do instancji.

4. Faza uczenia algorytmu

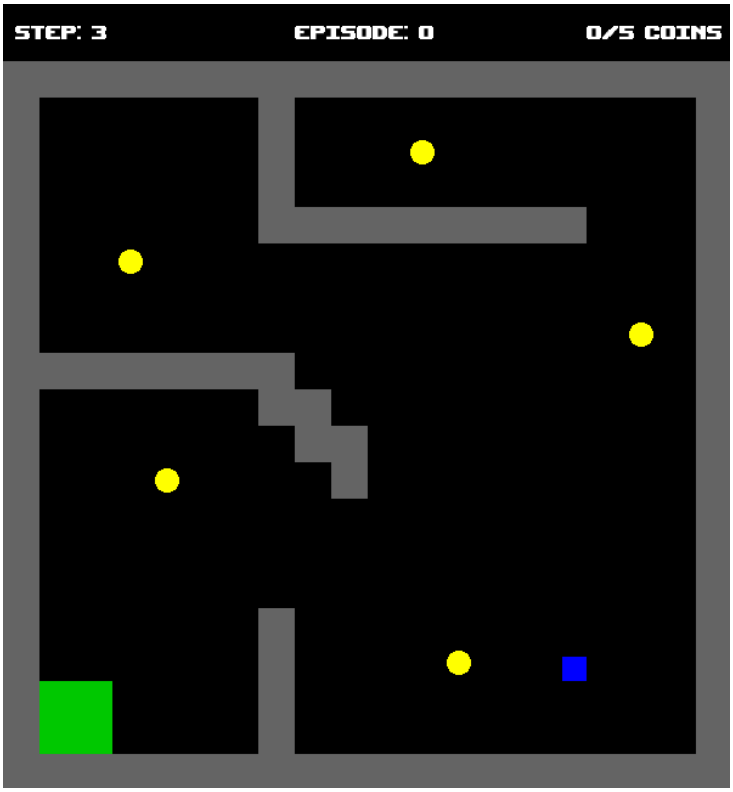
4.1. Etap 1 - pojedyncza plansza 20x20

Zaprojektowanie modelu, który skutecznie będzie wygrywał złożone poziomy o dużych wymiarach planszy (20x20) jest trudnym zadaniem. W pierwszej kolejności podjęto próbę wytrenowania modelu tylko na jednej planszy. Celem było sprawdzenie czy model jest w stanie poradzić sobie z tak złożonym problemem. Oczywiście spodziewano się, że wystąpi przeuczenie modelu (ang. *overfitting*), ale generalizacja nie była celem na tym etapie. Parametry takiego modelu wydają się być dobrym punktem startowym do utworzenia algorytmu, który będzie generalizował problem i skutecznie działał niezależnie od struktury planszy. Wszystkie obliczenia były wykonywane na komputerze dysponującym procesorem Intel i5-6500, kartą graficzną NVidia GeForce GTX 1060 6 GB oraz pamięci RAM 28 GB DDR4 2133 MHZ. Implementacja całej aplikacji wraz z danymi zebranymi podczas uczenia jest dostępna na stronie GitHub w publicznym repozytorium autora tej pracy [42].

Wszystkie próby zawarte w tym etapie zostały przeprowadzone na jednej, tej samej planszy. Mapa testowa została przedstawiona na rysunku 4.2. W początkowych etapach wykorzystano sekwencyjny model sieci neuronowej typu MLP. Topologia sieci została przedstawiona na rysunku 4.1. Podana architektura sieci była wykorzystywana we wszystkich próbach korzystających z modeli typu MLP. Zdecydowano się zastosować aktywację *ReLU* na warstwie wejściowej i warstwach ukrytych, natomiast na warstwie wyjściowej pozostała aktywacja liniowa. Jest to częsty zabieg stosowany w sieciach neuronowych, które są wykorzystywane w uczeniu przez wzmacnianie [43].



Rys. 4.1. Topologia sieci neuronowej typu MLP.



Rys. 4.2. Zrzut ekranu przedstawiający mapę wykorzystaną przy tworzeniu algorytmu.

Dobór szybkości uczenia

W pierwszej próbie, badaniu poddano różne wartości szybkości uczenia α. Parametry uczenia przedstawiono w tabeli 4.1. Zdecydowano się wykorzystać bufor powtórek, który przechowuje 50 000 iteracji, gdzie maksymalna liczba iteracji, która może zostać osiągnięta podczas uczenia to 300 000. Rozmiar bufora jest kwestią dyskusyjną. Z jednej strony, bufor powinien być duży, ponieważ wykorzystanie nieskorelowanych danych podczas uczenia przynosi pozytywne rezultaty. Z drugiej strony, im większy bufor, tym większe zużycie pamięci i algorytm może korzystać z bardzo starych danych, które zostały zebrane z początkowych, nieoptymalnych polityk, co może zaburzać uczenie [25]. Dobrana wartość jest wartością eksperymentalną. Wartość tę modyfikowano w kolejnych próbach w zależności od całkowitej liczby iteracji i osiągniętego wyniku.

Podczas uczenia wykorzystano 5 różnych wartości szybkości uczenia. Na podstawie zebranych danych utworzono wykres, na którym przedstawiono wysokość uzyskanej nagrody przez agenta w kolejnych epizodach.

Na podstawie wykresu 4.3, zdecydowano się wybrać szybkość uczenia równą $10^{(-4)}$, która jednocześnie jest wartością domyślną, ponieważ model z tą wartością jako jedyny był w stanie zebrać 4 monety na planszy. W tabeli 4.2 przedstawiono całkowitą liczbę zebranych monet przez poszczególnych agentów oraz średnią monet na epizod. Teoretycznie, agent z szybkością uczenia równą 0,00003 zebrał

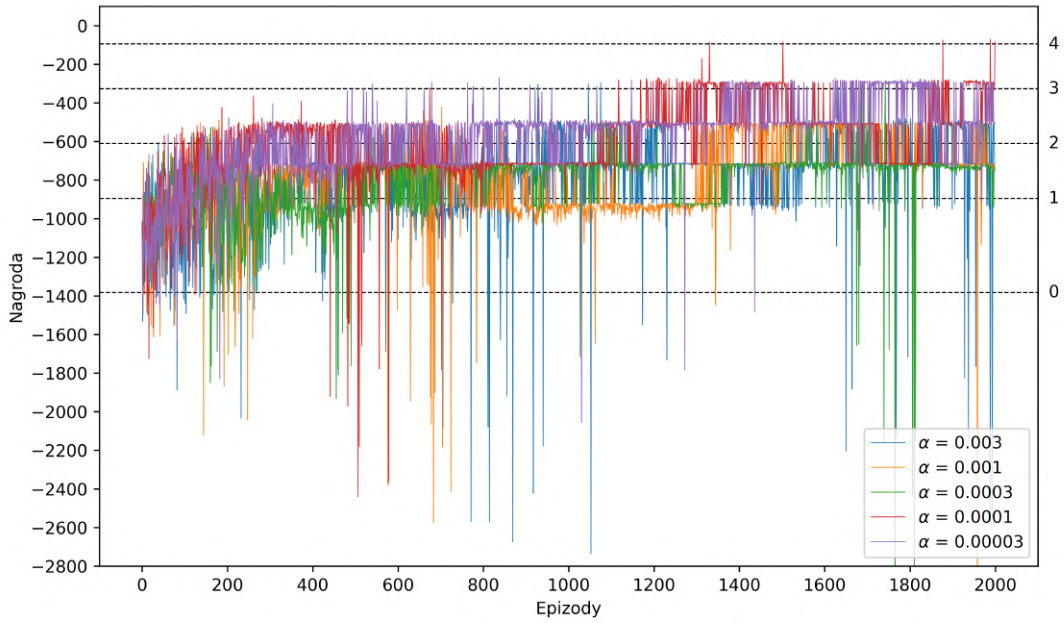
Tabela 4.1. Parametry wykorzystane podczas testowania szybkości uczenia.

Parametr	**Wartość**
Limit iteracji na epizod	150
Docelowa liczba epizodów	2000
Batch size	64
Rozmiar bufora	50 000
Współczynnik dyskontowania γ	0,99
ϵ	1,0
ϵ_d	0,999964
ϵ_{min}	0,01
Liczba neuronów warstwy wejściowej	2400
Liczba neuronów warstwy ukrytej (1)	1800
Liczba neuronów warstwy ukrytej (2)	1500
Liczba neuronów warstwy wyjściowej	4
Liczba parametrów podlegających trenowaniu	7 029 304
Szacowany czas uczenia pojedynczego modelu	17 godzin

Tabela 4.2. Liczba zebranych monet podczas uczenia.

α	**Całk. liczba zebranych monet**	**Śr. liczba zebranych monet na epizod**
0,003	2020	1,01
0,001	1921	0,96
0,0003	1716	0,86
0,0001	3365	1,68
0,00003	3686	1,84

ich więcej, jednak różnica jest niewielka i może ona wynikać z większego szczęścia podczas wyboru losowych akcji lub bardziej przychylnej początkowej inicjalizacji sieci neuronowej, a większa szybkość uczenia pozwoli szybciej uzyskać wyniki w przyszłych treningach. Na wykresie można zaobserwować charakterystyczne przeskoki między różnymi poziomami nagrody. Wynika to z tego w jaki sposób przyznawana jest nagroda za zbieranie monet oraz kara na koniec epizodu za pozostawione monety. Za zebranie monety przysługuje nagroda +75, natomiast za pozostawienie N monet, przysługuje kara -300 + (N * (-125)). Nagrody, które są przedstawione na wykresie są zsumowane za cały epizod. Dla przykładu, jeśli agent zbierze 2 monety, a pozostawi 3, to kara za cały epizod wyniesie około -525, natomiast przy zebraniu 3 monet i zostawieniu 2, kara wyniesie -400 (pomijając pomniejsze nagrody/kary za takie akcje jak dotknięcie ściany czy odkrywanie fragmentów mapy). Dla większej przejrzystości, na wykresach



Rys. 4.3. Testowanie różnych wartości parametru szybkości uczenia.

zaznaczono liniami przerywanymi, szacowaną wysokość nagrody w zależności od liczby zebranych monet. Poziomy szacowanych nagród obliczono na podstawie średniej nagrody wyliczonej ze wszystkich epizodów dla każdej liczby zebranych monet.

Dobór współczynnika dyskontowania

W następnym kroku badano wpływ różnych wartości współczynnika dyskontowania na ilość zebranych monet w kolejnych epizodach. Próba z szybkością uczenia pokazała, że agent przez 2000 epizodów potrafi zebrać maksymalnie 2-3 monety, z pojedynczymi przypadkami zebrania 4 monet. Wynik jest daleki od zamierzonego celu - przejścia poziomu. Żadnemu z dotychczasowych modeli nie udało się wygrać poziomu. Maksymalna nagroda, jaką agent może osiągnąć za perfekcyjne przejście poziomu oscyluje w granicach 900-1000. Parametry uczenia kolejnej próby przedstawiono w tabeli 4.1.

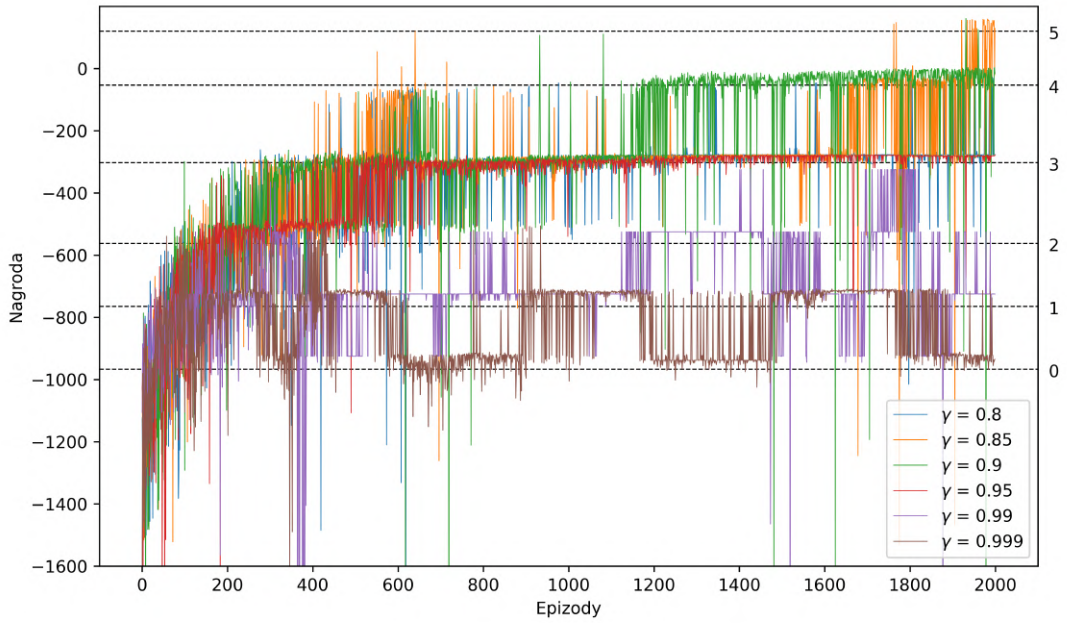
Wyniki z próby przedstawiono w tabeli 4.4. Można zauważyć, że najwyższa średnia zebranych monety przypada dla współczynnika dyskontowania równego 0,9. Wcześniej wykorzystany parametr γ = 0,99 był błędem, ponieważ daje znacznie gorsze rezultaty w porównaniu do niższych wartości. W dalszych próbach uczenia algorytmu wykorzystano jednak parametr $\gamma = 0{,}85$, mimo, że średnia i całkowita liczbę zebranych monet, wypada gorzej niż $\gamma = 0{,}9$. Pomijając wyniki z tabeli 4.4, można zauważyć na wykresie (rys. 4.4), że nagroda dla $\gamma = 0{,}9$ szybciej osiąga wartość nagrody bliską 0, natomiast dla γ = 0,85 pod sam koniec uczenia pojawia się przeskok nagrody do wartości powyżej 0, co dla $\gamma = 0{,}9$ jest niezauważalne. Są to momenty, gdy agent zebrał wszystkie 5 monet i był prawdopodobnie bliski wygrania poziomu. Z tego powodu, w przyszłych próbach zdecydowano się użyć współczynnika dyskontowania

Tabela 4.3. Parametry wykorzystane podczas testowania współczynnika dyskontowania.

Parametr	**Wartość**
Limit iteracji na epizod	150
Docelowa liczba epizodów	2000
Batch size	64
Rozmiar bufora	50 000
Szybkość uczenia α	0,0001
ϵ	1,0
ϵ_d	0,999964
ϵ_{min}	0,01
Liczba neuronów warstwy wejściowej	2400
Liczba neuronów warstwy ukrytej (1)	1800
Liczba neuronów warstwy ukrytej (2)	1500
Liczba neuronów warstwy wyjściowej	4
Liczba parametrów podlegających trenowaniu	7 029 304
Szacowany czas uczenia pojedynczego modelu	17 godzin

równemu 0,85 i zobaczyć jakie da rezultaty. Wykres pokazuje również, że uczenie trwało zbyt krótko. Przełomowy wzrost nagrody wynikający z zebrania wszystkich monet pojawia się dopiero w ostatnich 50-100 epizodach, co daje nadzieję na jeszcze lepsze rezultaty przy większej liczbie epizodów.

Proces uczenia dla $\gamma = 0{,}85$ został powtórzony, aby potwierdzić, że taka wartość współczynnika dyskontowania daje pozytywne rezultaty. Dodatkowo, zbadano, czy limit iteracji na epizod ma duży wpływ na proces uczenia algorytmu. Przed uruchomieniem treningu obliczono, że przy perfekcyjnym rozegraniu planszy przedstawionej na rys. 4.2 agent będzie potrzebował około 100 iteracji, aby przejść poziom. Zdecydowano się jednak przetestować większe wartości, tak, aby agent miał pewne pole do popełnienia błędów. Limit iteracji nie powinien być natomiast zbyt duży, ponieważ każde jego podniesienie znacznie zwiększa czas uczenia, niekoniecznie przynosząc pozytywne rezultaty. Reszta hiperparametrów modelu została bez zmian (tabela 4.3).

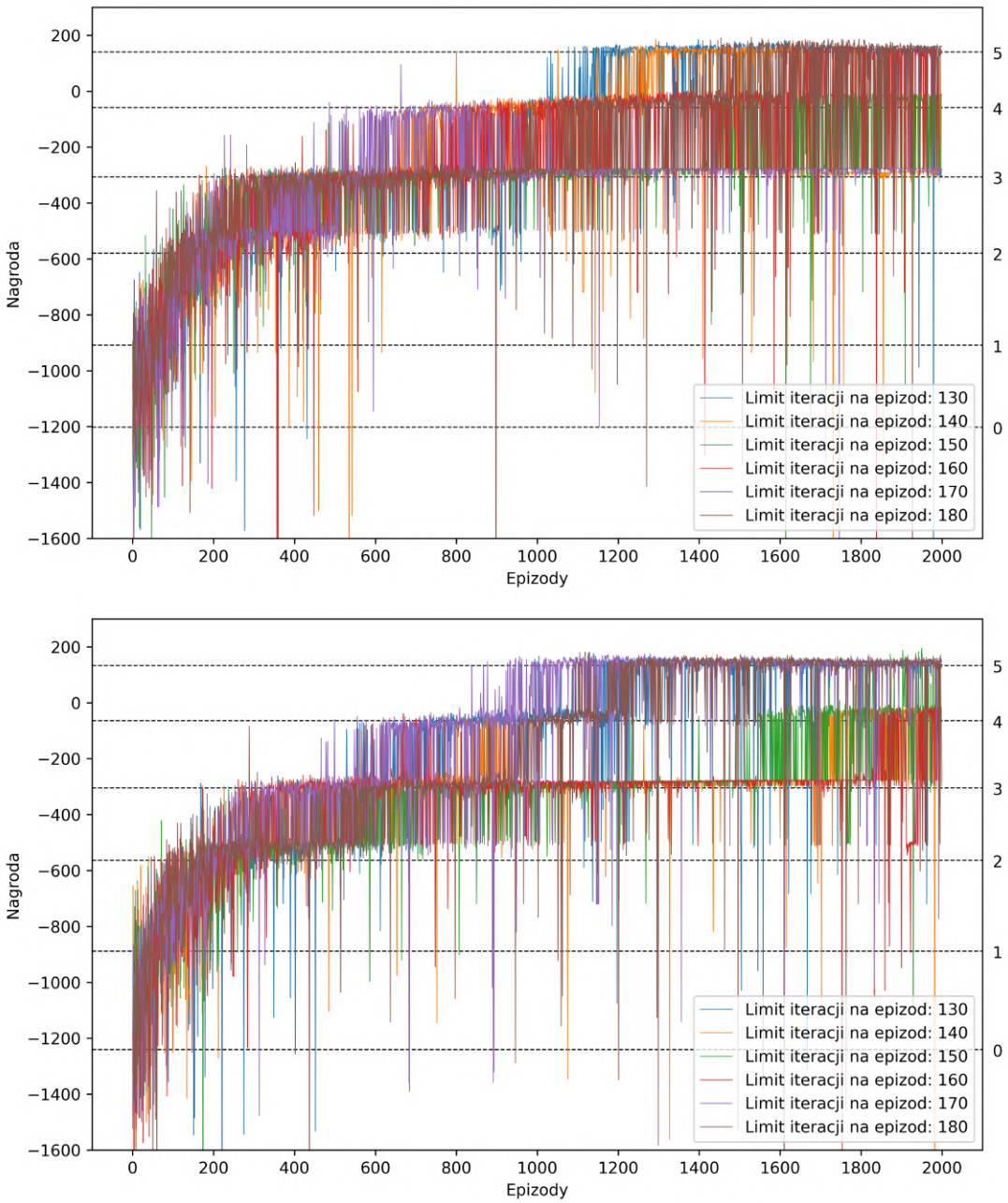


Rys. 4.4. Testowanie różnych wartości parametru współczynnika dyskontowania.

Tabela 4.4. Liczba zebranych monet podczas uczenia.

γ	**Całk. liczba zebranych monet**	**Śr. liczba zebranych monet na epizod**
0,8	5482	2,74
0,85	5888	2,94
0,9	6314	3,16
0,95	5358	2,68
0,99	2475	1,24
0,999	1238	0,62

Na rysunku 4.5 przedstawiono dwie niezależne próby uczenia modelu dla różnych limitów iteracji na epizod. Podczas tej próby, limit epizodów na cały okres uczenia wynosił tyle samo (2000), co w przypadku próby doboru współczynnika dyskontowania. Tutaj można zauważyć, że wiele modeli znacznie szybciej doszło do stanu, w którym agent potrafi zebrać wszystkie 5 monet w poziomie, niż w przypadku poprzedniej próby (rys. 4.4), co oznacza, że wybór parametru $\gamma = 0{,}85$ jest dobrym wyborem.



Rys. 4.5. Testowanie różnych limitów iteracji na epizod - dwie próby.

Wygranie pojedynczego poziomu

Podczas prób przedstawionych na rysunku 4.5 zauważono, że algorytm wielokrotnie potrafi zebrać wszystkie monety w poziomie, jednak ani razu nie udało mu się wygrać poziomu. Sumując wszystkie podejścia wszystkich agentów, na podstawie danych zebranych podczas uczenia obliczono, że na 24 000 epizodów (2 próby, każda po 6 instancji, a instancja po 2000 epizodów) aż 4352 razy zebrano wszystkie monety w poziomie, jednak mimo tylu prób, poziom nie został wygrany. Na podstawie wykresów 4.5 można zauważyć, że od około 1000-1200 epizodu, agent potrafi zebrać wszystkie monety -

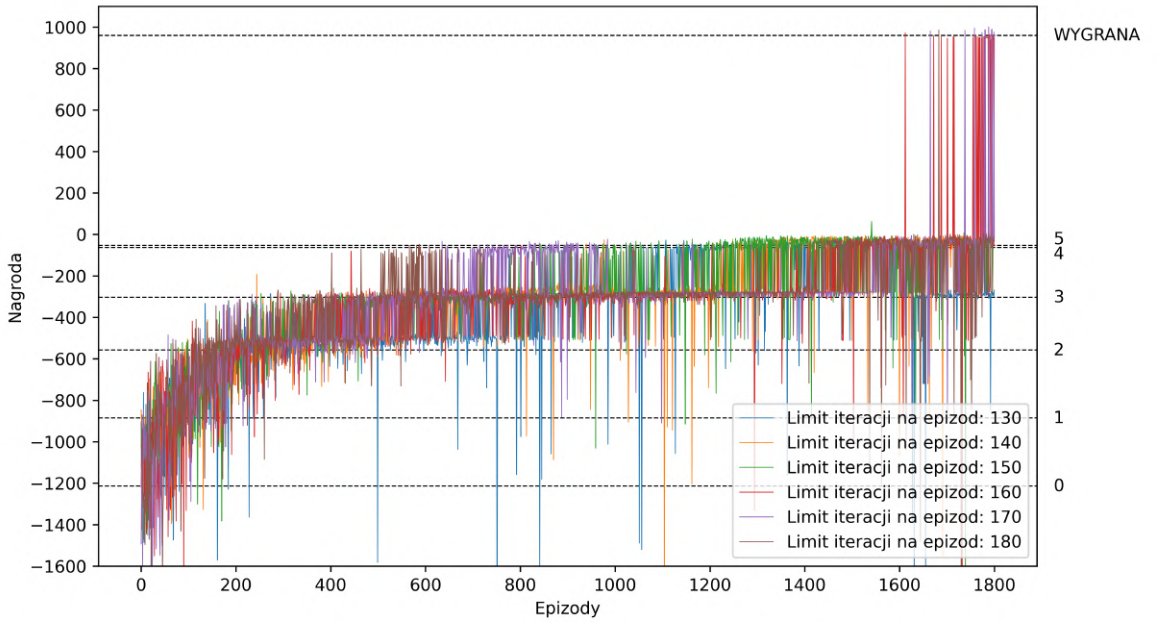
jednak przyglądając się danym z uczenia, zauważono, że wartość parametru ϵ spadła do wartości ϵ_{min} = 0,01, co oznacza, że agent wykonuje 99% akcji według aktualnej polityki, a tylko 1% akcji w sposób losowy. Można z tego wywnioskować, że agent przy takim ϵ nie jest w stanie już nauczyć się, jak dojść do wyjścia z poziomu, które przecież jest definiowane jako osobne pole na planszy. Agent nie wie, że należy do niego dojść i przy tak małej wartości ϵ jest bardzo małe prawdopodobieństwo, że się tego nauczy. Jednym z rozwiązań tego problemu jest zwiększenie parametru ϵ w momencie, gdy agent zbierze wszystkie monety - tak aby ponownie wprowadzić agenta w fazę eksploracji. To rozwiązanie jednak jest czasochłonne - trudno przewidzieć, kiedy agentowi uda się dojść do wyjścia podczas wykonywania losowych akcji. Takie podejście może również popsuć aktualną politykę. Z tego powodu, zdecydowano się w pewien sposób oszukać algorytm, poprzez dynamiczną podmianę stanu. Do tej pory, fragment mapy oznaczający wyjście z poziomu był identyfikowany jako osobna wartość przy użyciu kodowania "1 z n". Skoro agent potrafi zebrać wszystkie monety w poziomie, ale nie potrafi nauczyć się, że należy po zebraniu monet dojść do wyjścia - co gdyby zasymulować, że wyjście z poziomu to tak naprawdę moneta? Z perspektywy gry, nadal po wejściu na pole "Finish", poziom byłby wygrywany, jednak z perspektywy algorytmu uczenia przez wzmocnienie - pole reprezentujące wyjście z poziomu byłoby widoczne jako moneta. Podmiana pola powinna się jednak odbyć dopiero po zebraniu wszystkich monet w poziomie - przed ich zebraniem, wyjście z poziomu jest widoczne jako zwykły, nieodkryty fragment podłogi, natomiast po zebraniu wszystkich monet - wyjście również zamienia się w monetę. To rozwiązanie ma również dodatkową zaletę - redukuje się liczba danych, która jest wprowadzana na wejście sieci neuronowych, dzięki czemu uczenie trwa szybciej i jest mniej skomplikowane dla samego algorytmu. Redukcja rozmiaru danych wynika z zastosowania kodowania "1 z n" - zmniejsza się wymiar wektora reprezentującego dany obiekt. Nowe kodowanie stanów przedstawiono w tabeli 4.5.

Tabela 4.5. Nowe kodowanie stanów ID - "1 z n"

Obiekt	**ID**	**Kod "1 z n"**
Podłoga (nieodkryta)	0	[0, 0, 0, 0, 1]
Ściana	1	[0, 0, 0, 1, 0]
Moneta	4	[0, 0, 1, 0, 0]
Gracz	8	[0, 1, 0, 0, 0]
Podłoga (odkryta)	9	[1, 0, 0, 0, 0]

Po wprowadzeniu powyższej zmiany w logice stanów, ponownie przeprowadzono uczenie dla różnych limitów iteracji na epizod. Przy uczeniu wykorzystano parametry z poprzedniej próby (tabela 4.3), a współczynnik dyskontowania nadal wynosił 0,85.

Na wykresie (rys. 4.6) można zauważyć, że po zmianach z logiką stanu, algorytm pierwszy raz zaczął przechodzić poziom. Do pracy dodano link do filmu, w którym można obejrzeć jak AI sobie z tym radzi [44]. Na podstawie zebranych danych podczas uczenia, najlepsza instancja (z limitem iteracji na epizod wynoszącym 160) na 1800 epizodów, wygrała 35 razy. Należy jednak zwrócić uwagę, że uczenie było



Rys. 4.6. Próba dla różnych limitów iteracji na epizod, po wprowadzeniu zmian w stanie środowiska.

przeprowadzone na jednej i tej samej mapie, co spowodowało kompletne przeuczenie modelu. Agent nauczył się przechodzić plansze na pamięć, co potwierdzają próby z innymi planszami - przetestowano zapisaną sieć neuronową na 10 innych mapach. Po rozpoczęciu rozgrywki na innej mapie, agent albo się zapętlał (wykonywał ruchy typu prawo-lewo-prawo-lewo przez cały czas) albo wykonywał przez cały czas jedną akcję - np. ruch w prawo, przez co blokował się na pierwszej napotkanej ścianie. Jeśli na jego drodze była moneta, to był w stanie ją zebrać, jednak nic to nie zmieniło - i tak agent się blokował.

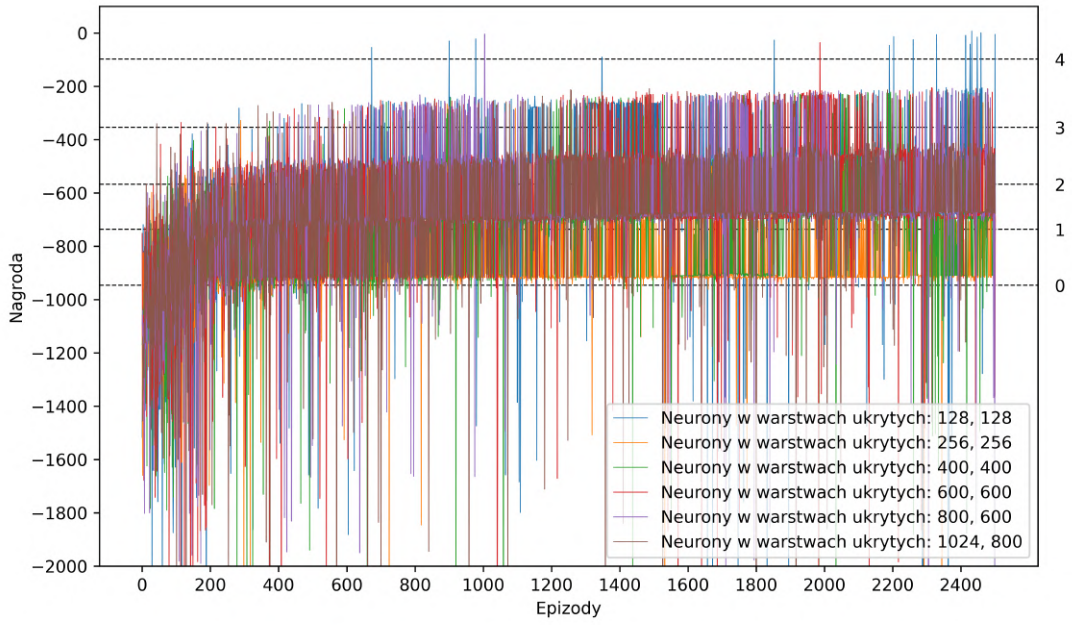
Próby generalizacji modelu na mapach 20x20

Próby generalizacji modelu rozpoczęto od zredukowania liczby parametrów sieci neuronowej poprzez zmniejszenie liczby neuronów. Duża liczba parametrów sprzyja przeuczeniu modelu, a mniejsza - pozwala na lepszą generalizacje problemu. Dotychczasowa struktura sieci została przedstawiona w tabeli 4.6. W związku ze zmianą logiki stanu w poprzedniej próbie, warstwa wejściowa zmieniła swój rozmiar z 2400 neuronów na 2000.

Tabela 4.6. Dotychczasowa sieć neuronowa.

Parametr	Wartość
Liczba neuronów warstwy wejściowej	2400 (2000)
Liczba neuronów warstwy ukrytej (1)	1800
Liczba neuronów warstwy ukrytej (2)	1500
Liczba neuronów warstwy wyjściowej	4

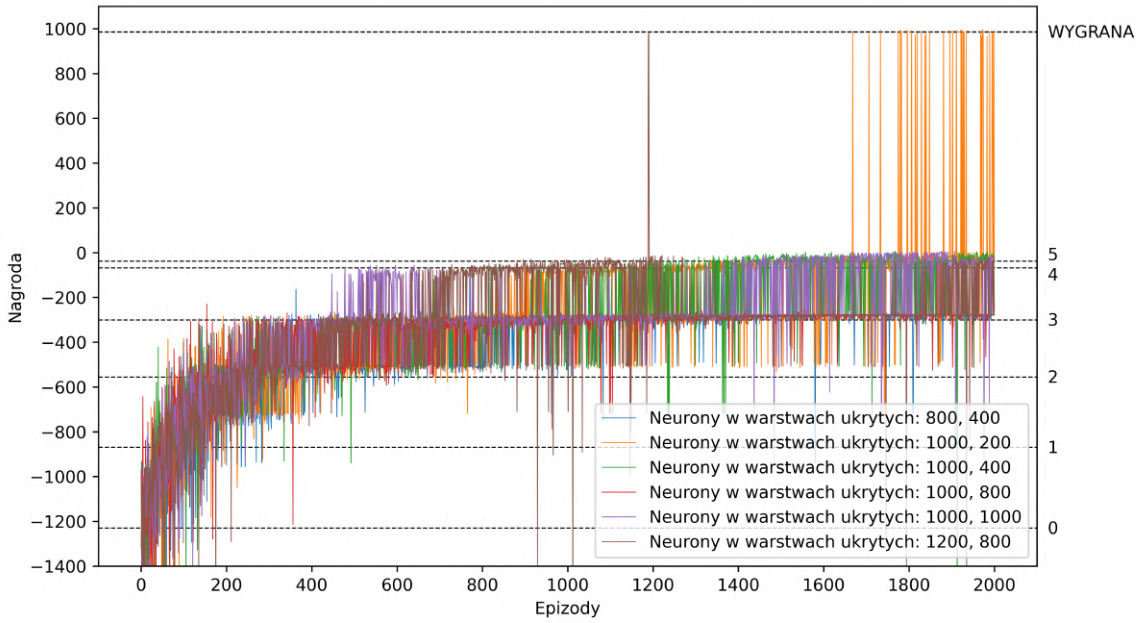
Celem tej próby jest zmniejszenie liczby neuronów w warstwach ukrytych do momentu, aż algorytm będzie w stanie wygrać poziom. Plansza wykorzystana do uczenia pozostała taka sama jak w poprzednich próbach. Wykres z procesu uczenia przedstawiono na rysunku 4.7.



Rys. 4.7. Pierwsza próba dla różnej liczby neuronów w warstwach ukrytych.

Wykres 4.7 przedstawiający pierwszą próbę generalizacji charakteryzuje się bardzo dużym chaosem. Algorytm nie jest w stanie w przeciągu 2500 epizodów zebrać 5 monet i daleko mu do przejścia poziomu.

W drugiej próbie (rys. 4.8) zauważono, że sieć, która posiada kolejno 1000 i 200 neuronów w warstwach ukrytych, była w stanie z powodzeniem przejść poziom. Na 2000 epizodów, udało się to 33 razy ze średnią zebranych monet na poziomie 3,33, co wydaje się być dobrym wynikiem. Z takim rozmiarem sieci neuronowej, wykonano dwie próby generalizacji modelu na mapach o wymiarze 20x20. Do tego celu użyto losowo wygenerowane mapy o parametrach podanych w tabeli 4.7. Przykładowe, wygenerowane mapy zostały przedstawione na rysunku 4.9.

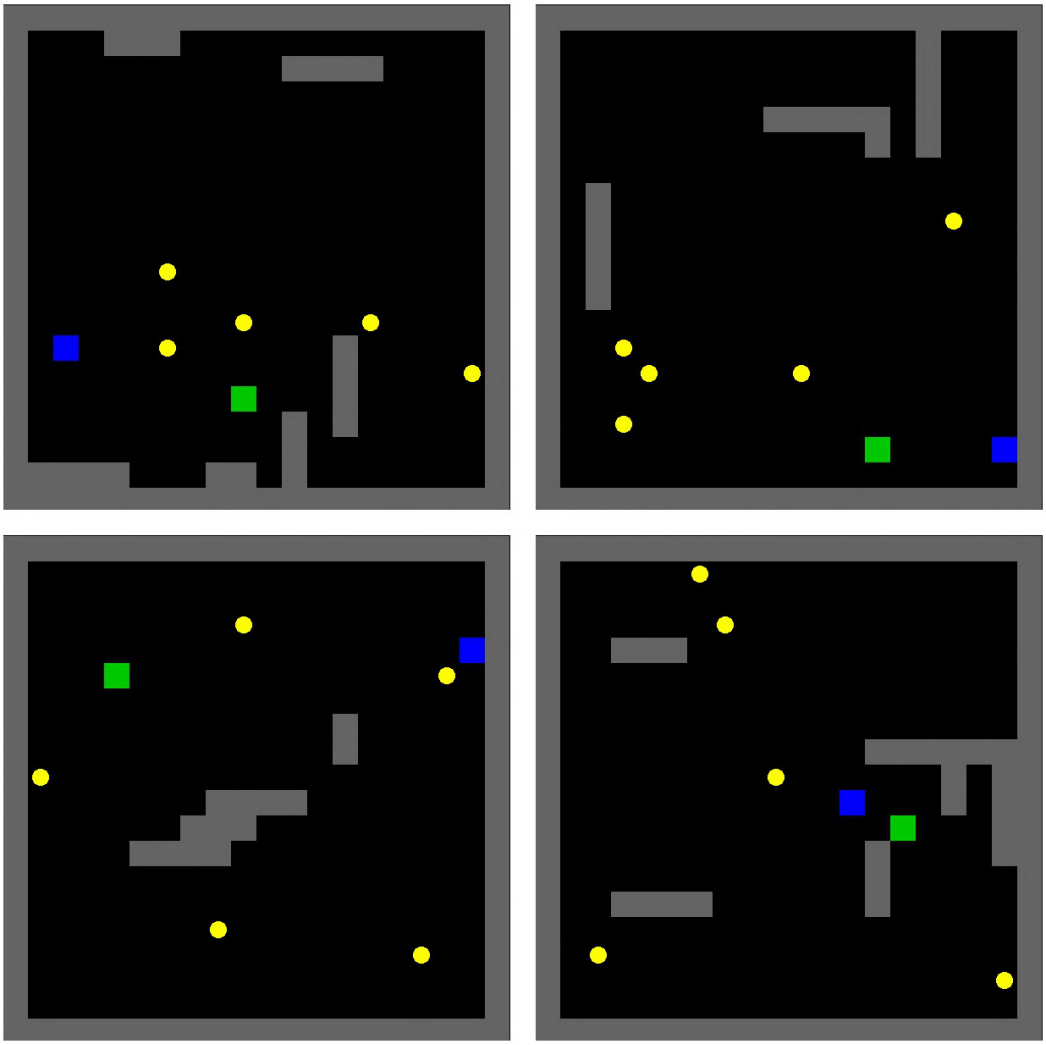


Rys. 4.8. Druga próba dla różnej liczby neuronów w warstwach ukrytych.

Tabela 4.7. Parametry losowych map o wymiarach 20x20.

Liczba monet	5
Min. liczba ścian	2
Maks. liczba ścian	8
Min. wymiar ściany	2x1 / 1x2 (prawd. 50% zmiany orientacji)
Maks. wymiar ściany	6x1 / 1x6 (prawd. 50% zmiany orientacji)
Wymiar wyjścia z poziomu	1x1

Na podstawie tych parametrów wygenerowano 1000 losowych map. Wszystkie plansze zostały sprawdzone pod kątem występowania duplikatów za pomocą skryptu, który porównuje zawartość plików tekstowych definiujących mapy, czy występują dwie (lub więcej) identyczne mapy. Podczas przeprowadzanych treningów algorytmu, nie wykorzystano augmentacji danych - wydaje się ona niepotrzebna, ponieważ zawsze istnieje możliwość wygenerowania większej liczby map, które będą unikalne. Wygenerowane mapy dzielono na dwa zbiory - treningowy oraz testowy. Parametry map nie różniły się pomiędzy zbiorami. Zbiór treningowy był wykorzystywany podczas uczenia algorytmu i najczęściej zawierał 5-10 razy więcej map, niż zbiór testowy. Zbiór testowy służył do ostatecznej walidacji modelu - podczas testowania, wartość parametru ϵ była równa 0, więc wszystkie akcje były wybierane na podstawie polityki uzyskanej podczas trenowania modelu. Uzyskane modele nie były sprawdzane pod kątem przeuczenia - zazwyczaj liczba wygenerowanych map była na tyle duża, że podczas uczenia powtarzały się one niewielką liczbę razy, przez co model nie powinien zdążyć się przeuczyć.



Rys. 4.9. Przykładowe, losowo wygenerowane plansze o wymiarze 20x20.

Algorytm tworzenia map nie jest idealny i pojawia się możliwość wygenerowania planszy, która jest niemożliwa do przejścia (brak dostępu do wyjścia/monety, zablokowany gracz), jednak prawdopodobieństwo utworzenia takiej mapy jest bardzo niskie. Sprawdzono około 1000 wygenerowanych, unikalnych map i zaledwie 2 z nich były wygenerowane błędnie - ich przejście było niemożliwe. Po utworzeniu losowych map przeprowadzono dwie próby generalizacji, każda po 5 instancji o tych samych parametrach (w obrębie próby). Parametry uczenia zostały przedstawione w tabeli 4.8.

Tabela 4.8. Parametry wykorzystane podczas dwóch prób generalizacji modelu na planszach 20x20.

Parametr	**Próba 1**	**Próba 2**
Limit iteracji na epizod	200	
Docelowa liczba epizodów	5000	10000
Batch size	64	
Rozmiar bufora	70 000	
Współczynnik dyskontowania γ	0,85	
ϵ	1,0	
ϵ_d	0,999983	0,999993
ϵ_{min}	0,01	
L. neuronów warstwy wejściowej	2000	
L. neuronów warstwy ukrytej (1)	1000	1400
L. neuronów warstwy ukrytej (2)	200	800
L. neuronów warstwy wyjściowej	4	

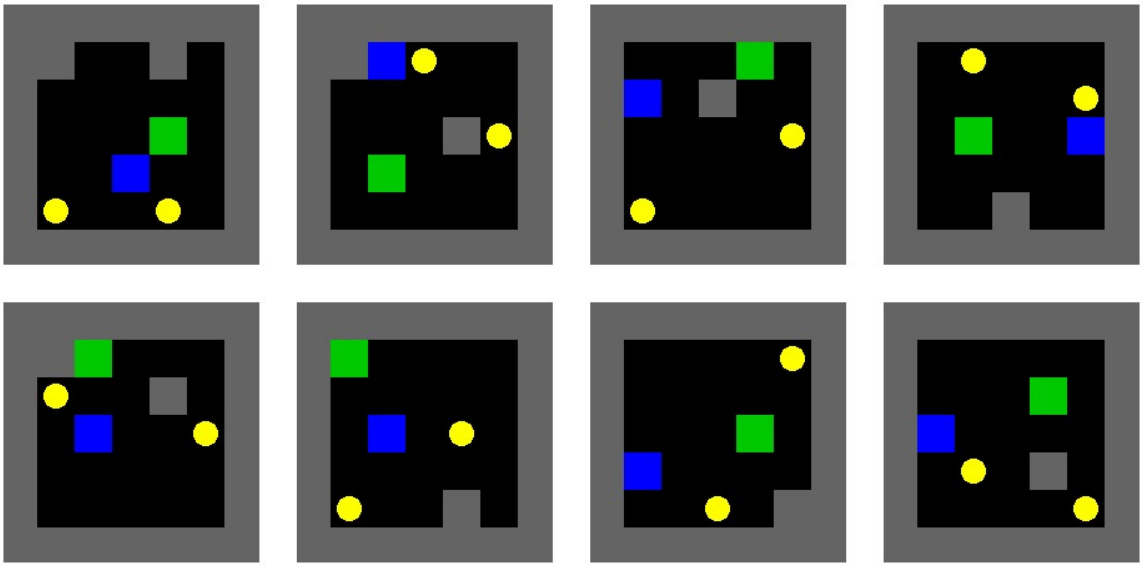
Obie próby zakończyły się niepowodzeniem. W pierwszej próbie, średnia liczba zebranych monet na epizod wyniosła ok. 0,88, a algorytm w przeciągu 25 000 epizodów zaledwie 4 razy zebrał wszystkie monety (nie wygrywając ani razu poziomu). Druga próba przebiegła niewiele lepiej. Średnio, algorytm zbierał ok. 1,58 monet epizod, a podczas 50 000 epizodów wygrał poziom tylko 19 razy. Zapisane modele sieci neuronowej nie zostały przetestowane na zbiorze testowym, ponieważ z powodu niewielkiej średniej zebranych monet oraz bardzo niskiej liczbie wygranych poziomów, nie przyniosłoby to pozytywnego rezultatu. Skoro algorytm w drugiej próbie osiągnął współczynnik zwycięstw na poziomie 0,038%, w momencie gdy plansze się powtarzały (1000 plansz, 10 000 epizodów - plansze były używane kilkukrotnie), to z pewnością nie poradziłby sobie z kompletnie nowymi mapami ze zbioru testowego.

4.2. Etap 2 - generalizacja modelu na planszach 7x7

Generalizacja modelu na mapach o wymiarach 20x20 nie zakończyła się powodzeniem. Nauczenie algorytmu przechodzenia dużych map nie jest trywialnym zadaniem i zajmuje dużo czasu. Przykładowo, próba 2, której parametry zostały przedstawione w tabeli 4.8 trwała około 60 godzin. Możliwe jest, że przy znacznie większej liczbie epizodów oraz większym zbiorze treningowym, algorytm z powodzeniem nauczyłby się przechodzić mapy o rozmiarze 20x20, jednak pochłonęłoby to ogromną ilość czasu i zasobów obliczeniowych. Zgrubne szacunki pokazują, że podczas realizacji całego projektu zużyto około 200-300 kWh. Zdecydowano się rozpocząć generalizacje modelu od prostszego przypadku - mniejszych map. Rozmiar mapy został zmniejszony z 20x20 do 7x7. Parametry generowania losowych map przedstawiono w tabeli 4.9. Przykładowe, wygenerowane mapy zostały przedstawione na rysunku 4.10.

Tabela 4.9. Parametry losowych map o wymiarach 7x7.

Liczba monet	2
Min. liczba ścian	1
Maks. liczba ścian	2
Min. wymiar ściany	1x1
Maks. wymiar ściany	1x1
Wymiar wyjścia z poziomu	1x1



Rys. 4.10. Przykładowe, losowo wygenerowane plansze o wymiarze 7x7.

Przeprowadzono 5 prób, w których głównie testowano wpływ liczby neuronów w warstwach ukrytych na skuteczność modelu. Eksperymentowano również z wielkością zbioru treningowego, parametrem ϵ_d i docelową liczbą epizodów. Parametry algorytmu przedstawiono w tabeli 4.10, natomiast testowane liczby neuronów warstw ukrytych zostały zawarte w tabeli 4.11. Wartości te, oznaczają ile neuronów posiadała sieć w kolejnych warstwach ukrytych.

Tabela 4.10. Parametry wykorzystane podczas 5 prób generalizacji modelu na planszach 7x7.

Parametr	**Próba 1**	**Próba 2**	**Próba 3**	**Próba 4**	**Próba 5**
Rozmiar zbioru treningowego	100	200	800	1600	2400
Rozmiar zbioru testowego	N/A	N/A	200	400	600
Liczba testowanych instancji	10	10	7	7	5
Limit iteracji na epizod	30	30	40	40	40
Docelowa liczba epizodów	1000	2000	10 000	10 000	10 000
Batch size	64				
Rozmiar bufora	30 000	30 000	60 000	60 000	60 000
Szybkość uczenia	0,0001				
Współczynnik dyskontowania γ	0,85				
ϵ	1,0				
ϵ_d	0,9993	0,9996	0.9999865	0.9999864	0.9999864
ϵ_{min}	0,01				
L. neuronów warstwy wejściowej	245				
L. neuronów warstwy wyjściowej	4				

Tabela 4.11. Testowane liczby neuronów w kolejnych warstwach ukrytych na planszach 7x7

Nr instancji	**Próba 1**	**Próba 2**	**Próba 3**	**Próba 4**	**Próba 5**
Instancja 1	64, 16	128, 16	128, 16	128, 32	512, 64
Instancja 2	32, 32	128, 32	128, 32	256, 32	512, 256
Instancja 3	64, 64	128, 64	256, 32	128, 16, 16	512, 128, 32
Instancja 4	128, 64	100, 20, 10	256, 20, 10	256, 32, 16	256, 128, 64, 32
Instancja 5	16, 16, 16	128, 16, 16	128, 16, 16	256, 64, 32	512, 256, 128, 64
Instancja 6	32, 16, 16	64, 16, 16	64, 64, 16	128, 64, 64, 32	N/A
Instancja 7	32, 32, 16	64, 32, 16	256, 32, 16	256, 64, 32, 16	N/A
Instancja 8	32, 16, 8, 8	64, 32, 16, 8	N/A	N/A	N/A
Instancja 9	16, 8, 8, 8	128, 32, 8, 8	N/A	N/A	N/A
Instancja 10	32, 8, 4, 4, 4	128, 32, 16, 8, 4	N/A	N/A	N/A

Pierwsze dwie próby obejmowały tylko i wyłącznie proces uczenia, bez testowania algorytmu na odrębnym zbiorze. Docelowa liczba epizodów również była znacznie mniejsza. Takie testowanie miało na celu szybkie przybliżenie jak będzie zachowywał się algorytm przy różnych liczbach neuronów w warstwach ukrytych - czy w ogóle odniesie sukces. Z powodu zmiany rozmiaru mapy, zmienił się również rozmiar danych wejściowych do sieci neuronowej (z 2000 na 245), więc dotychczasowa liczba neuronów warstw ukrytych sieci powinna zostać dobrana na nowo. W tabeli 4.12 przedstawiono procent zwycięstw osiągnięty przez każdy z modeli. Procent zwycięstw dla prób 1 i 2 został obliczony na podstawie zwycięstw podczas procesu uczenia (na zbiorze treningowym), natomiast dla prób 3-5 wykorzystano również zbiory testowe, aby sprawdzić działanie algorytmu.

Tabela 4.12. Procent zwycięstw dla badanych modeli na mapach 7x7. Dla modeli 1-2 przedstawiono tylko wyniki ze zbioru treningowego, natomiast dla modeli 3-5 wyniki ze zbiorów treningowych/testowych.

Nr instancji	Próba 1	Próba 2	Próba 3	Próba 4	Próba 5
Instancja 1	11,3% / -	10,75% / -	44,21% / 67,5%	66,13% / 85%	82,28% / 93,33%
Instancja 2	10,2% / -	13,25% / -	55,25% / 81,5%	79,4 % / **91,5%**	85,12% / 94,33%
Instancja 3	10,3% / -	11,8% / -	69,34% / 88,5%	67,28% / 87,25%	84,05% / **95,33%**
Instancja 4	9,9% / -	5,15% / -	64,5% / 87,5%	77,47% / 90,75%	72,94% / 91,5%
Instancja 5	8,1% / -	11,1% / -	59,87% / 88,5%	78,1% / 90,75%	84,75% / 91,17%
Instancja 6	10,1% / -	5,95% / -	41,63% / 58,5%	67,3% / 86%	N/A
Instancja 7	12% / -	6,75% / -	66,5% / **89%**	76,88% / 91%	N/A
Instancja 8	4,1% / -	8% / -	N/A	N/A	N/A
Instancja 9	6,5% / -	5,45% / -	N/A	N/A	N/A
Instancja 10	3,8% / -	1,45% / -	N/A	N/A	N/A

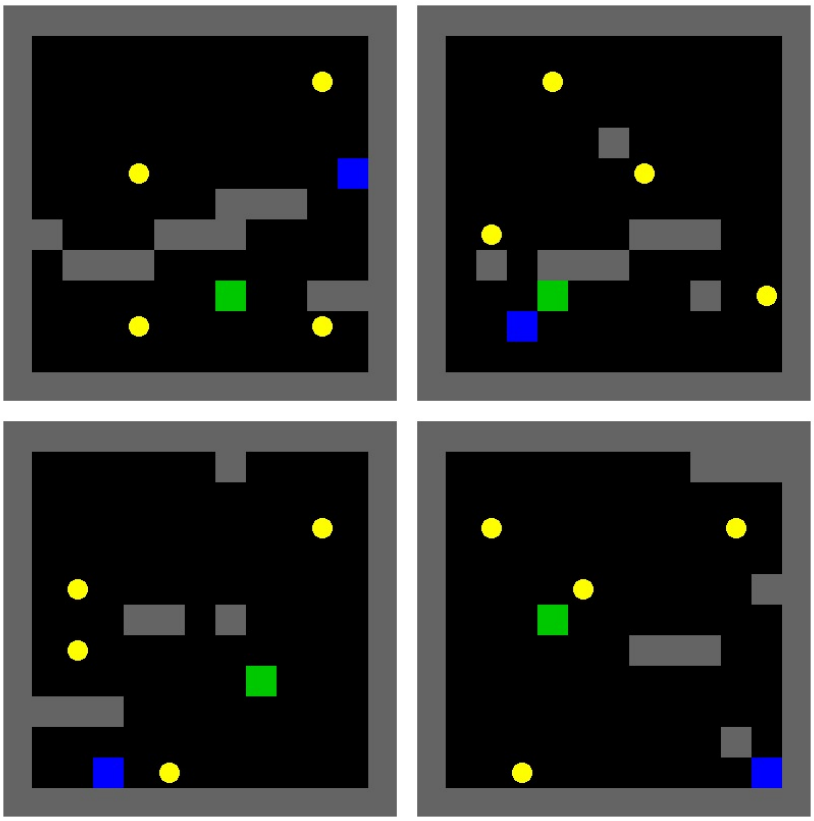
Próby 1 i 2 nie przyniosły rezultatu w postaci skutecznego algorytmu, który będzie w stanie z powodzeniem przechodzić losowe poziomy. Najwyższy współczynnik zwycięstw odnotowała instancja 2 w próbie 2 i wynosił on zaledwie 13,25% w zbiorze treningowym. Oznacza to, że agent wygrał poziom 265 razy na 2000 epizodów, jednak należy zwrócić uwagę, że zbiór treningowy zawierał zaledwie 200 różnych plansz, co powoduje, że w trakcie uczenia plansze powtarzały się nawet 10-krotnie. Niski poziom zwycięstw mógł być spowodowany zbyt krótkim uczeniem - docelowa liczba epizodów w próbach 1, 2 wynosiła kolejno 1000 i 2000 epizodów. Dodatkowo, próby te pokazały, że stosowanie zbyt głębokiej architektury sieci (np. 5 warstw ukrytych) nie jest dobrym rozwiązaniem. W kolejnych procesach uczenia (próby 3-5) zwiększono docelową liczbę epizodów do 10 000 oraz sukcesywnie podnoszono liczbę neuronów w warstwach ukrytych. Ponieważ liczba parametrów sieci rosła wraz ze zwiększaniem liczby neuronów, rozszerzano również zbiory treningowe. Rozbieżności wyników pomiędzy zbiorami treningowymi i testowymi wynikają z zastosowania algorytmu $\epsilon - greedy$ - na początku uczenia, agent wykonuje większość akcji w sposób losowy, co często uniemożliwia mu przejście poziomu. Ostateczny

wynik, jaki algorytm był w stanie osiągnąć to 95,33% stosunku zwycięstw do wszystkich rozegranych epizodów. Próba testowa została przeprowadzona na 600 unikalnych, losowo wygenerowanych mapach, a algorytm był w stanie wygrać aż 572 z nich. Zarówno zbiory testowe, jak i zbiory treningowe zostały dokładnie sprawdzone pod kątem duplikatów. Dalsze zwiększanie liczby neuronów oraz zbiorów treningowych prawdopodobnie przyniosłoby jeszcze lepsze rezultaty, jednak po osiągnięciu wyniku 95,33% zdecydowano się zakończyć etap. Do pracy został dołączony odnośnik z filmem, w którym przedstawiono działanie najlepszego modelu na zbiorze testowym [45].

4.3. Etap 3 - generalizacja modelu na planszach 13x13

Sekwencyjne modele typu MLP

Kolejny etap rozpoczęto od zmiany parametrów algorytmu generującego losowe mapy (tabela 4.13). Przykładowe plansze przedstawiono na rysunku 4.11. Ponownie, każdy wygenerowany zbiór został sprawdzony pod kątem duplikatów, w szczególności, czy duplikaty nie występują między zbiorem treningowym i testowym. W pierwszym kroku, przeprowadzono dwie próby podobne do tych, które były przeprowadzane dla map 7x7.



Rys. 4.11. Przykładowe, losowo wygenerowane plansze o wymiarze 13x13.

Tabela 4.13. Parametry losowych map o wymiarach 13x13.

Liczba monet	4
Min. liczba ścian	2
Maks. liczba ścian	5
Min. wymiar ściany	1x1
Maks. wymiar ściany	1x3, 3x1 (prawd. 50% zmiany orientacji)
Wymiar wyjścia z poziomu	1x1

Tabela 4.14. Parametry wykorzystane podczas 2 prób generalizacji modelu na planszach 13x13.

Parametr	**Próba 1**	**Próba 2**
Rozmiar zbioru treningowego	3000	
Rozmiar zbioru testowego	N/A	N/A
Liczba testowanych instancji	7	5
Limit iteracji na epizod	70	80
Docelowa liczba epizodów	10000	14000
Batch size	64	
Rozmiar bufora	60 000	
Szybkość uczenia	0,0001	
Współczynnik dyskontowania γ	0,85	
ϵ	1,0	
ϵ_d	0.9999861	0.9999986
ϵ_{min}	0,01	
L. neuronów warstwy wejściowej	845	
L. neuronów warstwy wyjściowej	4	

Niestety, przeprowadzone próby nie przyniosły pozytywnego rezultatu. W 1 próbie, najlepsza była instancja 3, która na 10 000 epizodów, wygrała zaledwie 11 razy. W 2 próbie, wyniki były nieznacznie lepsze, ponieważ najwięcej wygranych osiągnęła instancja nr 1 - 40 zwycięstw na 14 000 epizodów. Z powodu tak słabych wyników, zrezygnowano z testowania algorytmu na zbiorze testowym.

Tabela 4.15. Testowane liczby neuronów oraz współczynnik zwycięstw w kolejnych warstwach ukrytych na planszach 13x13

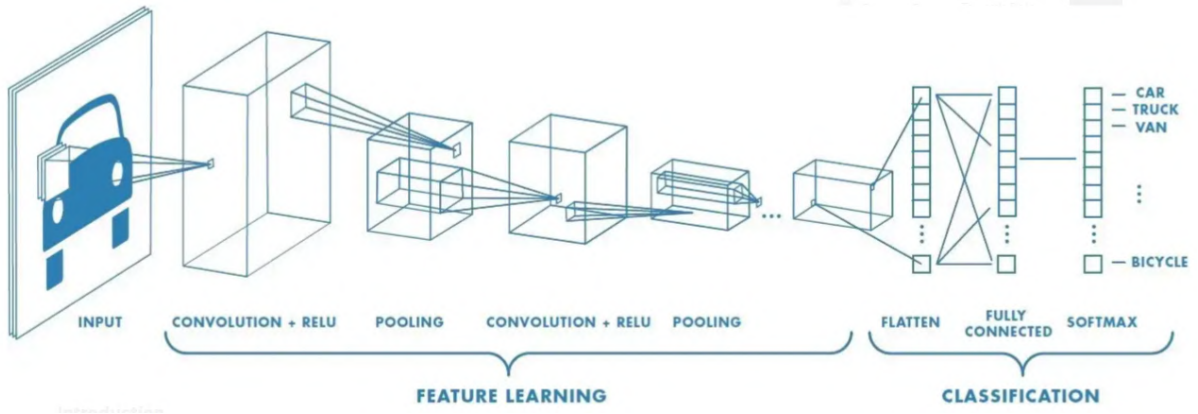
Nr instancji	Próba 1	Próba 2
Instancja 1	512, 256	1400, 200
Instancja 2	1024, 128	1800, 400
Instancja 3	1024, 512	1600, 400, 100
Instancja 4	768, 256, 32	2000, 1000, 100
Instancja 5	512, 256, 128	1600, 800, 400, 200
Instancja 6	1024, 256, 128, 32	N/A
Instancja 7	768, 512, 256, 64	N/A

Sieci splotowe

Z powodu bardzo słabej skuteczności w poprzednich dwóch próbach na mapach o wymiarze 13x13 zdecydowano się zmienić podejście do problemu. W teorii, zwykłe gęste sieci neuronowe bazujące na perceptronie, przy odpowiednich parametrach mogłyby skutecznie nauczyć się generalizować problem przechodzenia większych map, jednak proces uczenia jest bardzo czasochłonny (uczenie modelu z 14 000 epizodami trwało około 5-6 dni), a mała pomyłka w parametrach jest kosztowna. Gęste sieci neuronowe charakteryzują się dużą liczbą parametrów, co sprzyja przeuczeniu algorytmu. Aby temu zaradzić, zdecydowano się wykorzystać inną architekturę sieci neuronowych, konkretnie sieci splotowe. Sieci te, ze względu na swoje właściwości (wykonywanie operacji splotu, zastosowanie filtrów itd.) bardzo dobrze sprawdzają się przy przetwarzaniu obrazów, szczególnie w uczeniu nadzorowanym (np. klasyfikacja obrazów). Zastosowanie takiej architektury wydaje się dobrym pomysłem w przypadku tego problemu - stan gry przypomina obraz, zmapowany do rozmiaru 13x13 pikseli - zastosowanie operacji splotu może być tu bardzo pomocne. Dodatkowo, sieci splotowe charakteryzują się znacznie mniejszą liczbą parametrów, które podlegają trenowaniu, dzięki czemu są one dużo bardziej odporne na przeuczenie niż zwykłe gęste sieci neuronowe.

Standardowa sieć splotowa składa się z warstw splotowych (*Convolutional*) oraz warstw odpowiadających za skalowanie (*MaxPooling*). Następnie, dołączona jest warstwa, która spłaszcza przetworzone dane (*Flatten*), a na końcu znajduje się zwykła sieć neuronowa typu MLP (*Dense*), która działa jako klasyfikator. Bardzo znanym przykładem sieci splotowej jest sieć VGG16, która służy do klasyfikacji obrazów [46]. Schemat obrazujący architekturę sieci splotowej został przedstawiony na rysunku 4.12.

W pierwszym kroku, zdecydowano zastosować się prostą architekturę sieci splotowej i przetestować wpływ liczby neuronów na wyjściu sieci (po spłaszczeniu) na skuteczność modelu. Testowana sieć została przedstawiona na rysunku 4.13. Zdecydowano się na początku zrezygnować z warstw *MaxPooling*, aby przetestować mniej skomplikowany model i sprawdzić czy zmiana architektury sieci pomoże w rozwiązaniu problemu. Liczba filtrów, która została zastosowana jest wartością dobraną testowo - wykorzystano niewielkie rozmiary filtrów, ponieważ dane wejściowe nie są skomplikowane z punktu widzenia



Rys. 4.12. Przykładowy schemat sieci splotowej [47].

przetwarzania obrazów (niewielka rozdzielczość, mała liczba różnych obiektów). Zastosowano również niewielki rozmiar jądra - 3x3 (ang. *kernel size*), ponieważ rozmiar danych wejściowych jest prawdopodobnie zbyt mały, aby dobranie większego rozmiaru było skuteczne. Ponadto, wykorzystano parametr *padding* w trybie krawędziowym (ang. *same padding*), aby zachować wymiary przestrzenne danych - wymiar danych wejściowych i wyjściowych się nie zmienia. Natomiast jeśli chodzi o parametr strides - zdecydowano się wybrać wartość (1, 1), aby wyciągnąć jak najwięcej szczegółów z obrazu. Parametry wykorzystane przy uczeniu zostały przedstawione w tabeli 4.16, natomiast wyniki wraz z testowanymi wartościami liczb neuronów przedstawiono w tabeli 4.17.

```
model = Sequential()
model.add(Conv2D(16, (3, 3), strides=(1, 1), padding='same', input_shape=(13, 13, 5), activation='relu'))
model.add(Conv2D(32, (3, 3), strides=(1, 1), padding='same', activation='relu'))
model.add(Conv2D(64, (3, 3), strides=(1, 1), padding='same', activation='relu'))
model.add(Flatten())
model.add(Dense(x1, activation='relu'))
model.add(Dense(x2, activation='relu'))
model.add(Dense(4))
```

Rys. 4.13. Testowa sieć splotowa wykorzystana do określenia liczby neuronów na wyjściu dla map 13x13.

Na podstawie otrzymanych wyników (tabela 4.17) wywnioskowano, że zastosowanie innej architektury w postaci sieci splotowej było dobrym wyborem w przypadku tego problemu. Mimo, że jest to dopiero pierwsza próba, praktycznie każda testowana sieć osiągnęła znacznie lepsze wyniki niż w próbie z architekturą MLP. W poprzednich próbach, sieć osiągnęła 0.3% zwycięstw na zbiorze treningowym, gdzie plansze się powtarzały, więc algorytm miał większe szanse się ich nauczyć. W przypadku sieci splotowych, osiągnięty wynik to ponad 35% skuteczności na zbiorze testowym, czyli na planszach, których algorytm wcześniej nie widział. Na podstawie wyników z przeprowadzonego uczenia, zdecydowano się użyć klasyfikatora, który uzyskał największy współczynnik zwycięstw - 42,7%. Wynik jest daleki od ideału, niecałe 43% skuteczności nie jest wysokim wynikiem patrząc pod kątem klasyfikatora, jednak algorytm uczenia przez wzmocnienie musi wykonać wiele sekwencyjnych akcji, aby dojść do sukcesu,

Tabela 4.16. Wykorzystane parametry podczas określania liczby neuronów dla map 13x13.

Parametr	**Wartość**
Rozmiar zbioru treningowego	4000
Rozmiar zbioru testowego	1000
Liczba testowanych instancji	5
Limit iteracji na epizod	80
Docelowa liczba epizodów	16 000
Batch size	64
Rozmiar bufora	75 000
Szybkość uczenia	0,0001
Współczynnik dyskontowania γ	0,85
ϵ	1,0
ϵ_d	0,999991
ϵ_{min}	0,01

Tabela 4.17. Współczynnik zwycięstw dla poszczególnych instancji dla map 13x13. Liczba neuronów (x_1, x_2) odnosi się do fragmentu kodu z rysunku 4.13.

Liczba neuronów (x_1, x_2)	**% zwycięstw na zbiorze testowym**
(32, 16)	35,1%
(64, 16)	37,9%
(128, 16)	42,7%
(128, 32)	40,6%
(256, 32)	37,2%

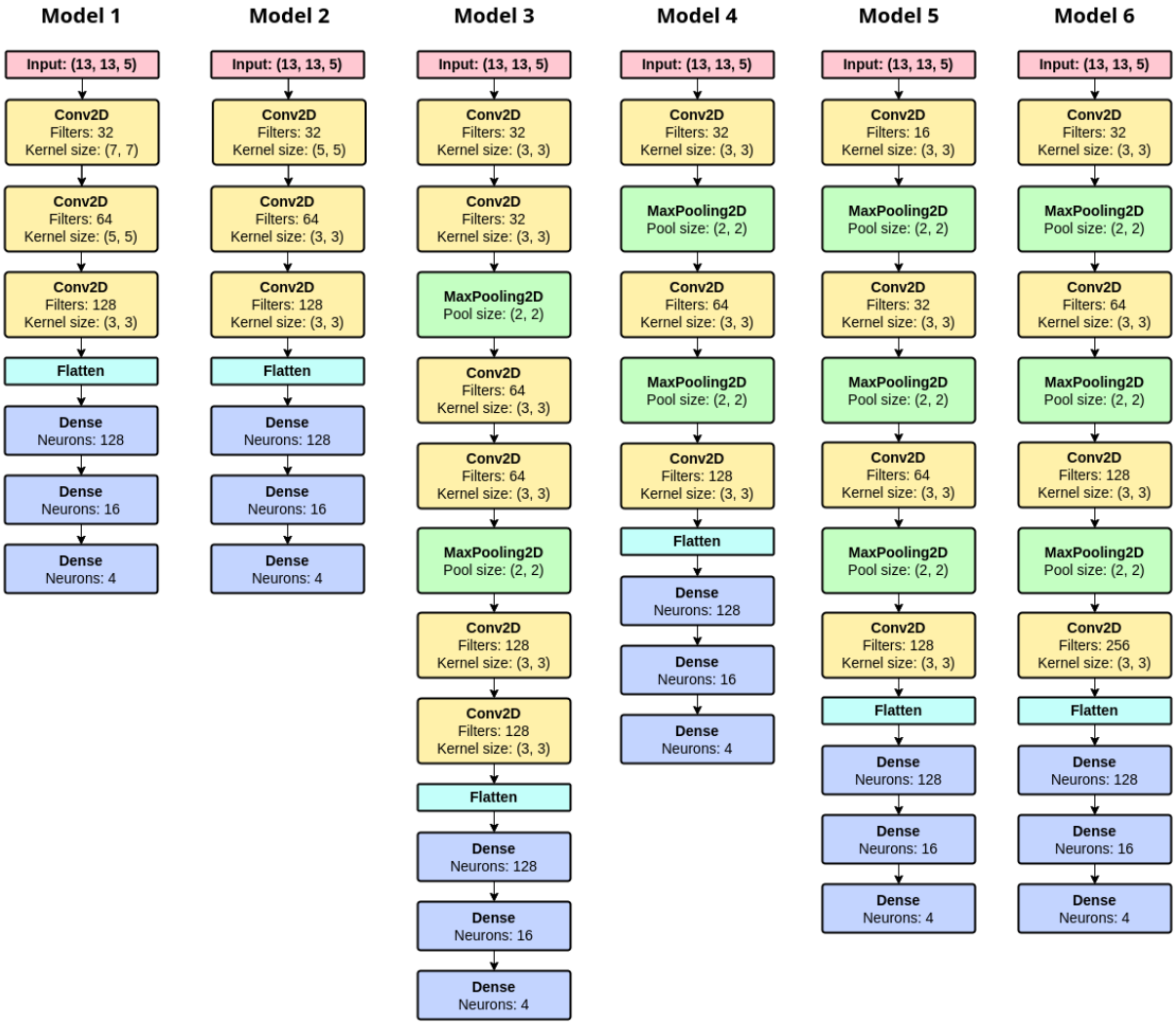
co jest dużo bardziej skomplikowane od zwykłej klasyfikacji. Mimo wszystko, ogromny przeskok skuteczności względem zwykłych sieci neuronowych typu MLP daje nadzieję na lepsze wyniki.

W kolejnym kroku, zdecydowano się przetestować różne warianty warstw splotowych. Zmienione parametry względem poprzedniej próby (tabela 4.16) przedstawiono w tabeli 4.18.

Przetestowane architektury zostały przedstawione na rysunku 4.14. W większości przypadków, zdecydowano się zastosować warstwy *MaxPooling2D*, które są w stanie zredukować liczbę cech z każdym kolejnym użyciem. Każda warstwa *MaxPooling2D* pozwala przeprowadzić ekstrakcję cech, zmniejszając wymiar danych czterokrotnie (zastosowano okno ekstrakcji o wymiarach 2x2). W modelach 1 i 2 przedstawionych na rysunku 4.14 nie zastosowano warstw skalujących, dzięki czemu, po spłaszczeniu sieć posiada ogromną ilość parametrów. Obliczono, że modele 1 i 2 posiadają około 2,9 mln parametrów. Są to duże liczby, które bardzo sprzyjają przeuczeniu się algorytmu. Zdecydowano się jednak przetestować takie architektury, aby porównać je z innymi. W kolejnych modelach testowano różne liczby warstw

Tabela 4.18. Zmienione parametry względem tabeli 4.14.

Parametr	Wartość
Rozmiar zbioru treningowego	10 000
Liczba testowanych instancji	7
Limit iteracji na epizod	90
Docelowa liczba epizodów	10 000
Rozmiar bufora	70 000



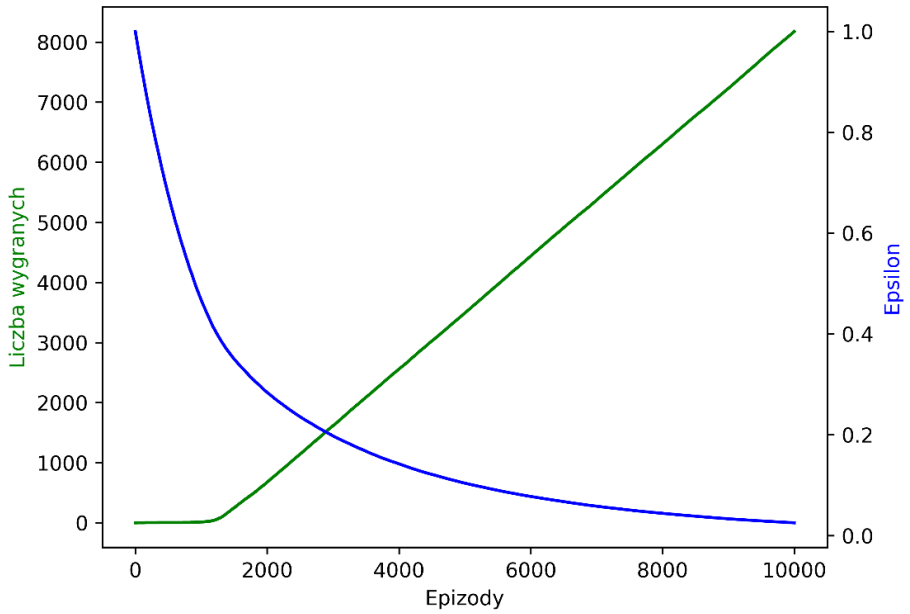
Rys. 4.14. Architektury testowanych modeli sieci splotowych dla plansz 13x13.

splotowych, różną ich głębokość oraz rozmiary filtrów. We wszystkich modelach zostały wykorzystane parametry: *strides* = (1, 1), *padding* = "same" oraz aktywacja "ReLU" na warstwach Conv2D oraz Dense. Wszystkie modele zostały przetestowane na zbiorze testowym obejmującym 1000 losowych map. Wyniki z powyższej próby zostały przedstawione w tabeli 4.19.

Tabela 4.19. Współczynnik zwycięstw dla testowanych architektur sieci splotowych dla map 13x13.

Nr modelu	% zwycięstw na zbiorze treningowym	% zwycięstw na zbiorze testowym
Model 1	39,86%	43,4%
Model 2	32,61%	42,2%
Model 3	72,38%	83,4%
Model 4	81,76%	**91%**
Model 5	78,02%	90,2%
Model 6	78,57%	90,2%

Zastosowanie warstw *MaxPooling* pozwoliło na znaczne poprawienie wyników algorytmu. Warto zwrócić uwagę na wyniki przedstawione w rubryce "% zwycięstw w zbiorze treningowym" w tabeli 4.17. Uczenie obejmowało 10 000 epizodów, tyle co liczebność zbioru treningowego - każdy epizod był trenowany na unikalnej mapie. Algorytm dobierania map do poziomu został zaprojektowany tak, aby żadna z map nie mogła się powtórzyć podczas uczenia. Modele 3-6 osiągnęły ponad 70% skuteczności na zbiorze treningowym, co oznacza, że wygrały ponad 7 000 losowych plansz. Parametr ϵ_d został dobrany w taki sposób, że w okolicy 2000 epizodu, wartość ϵ wynosiła około 0,25, co oznacza, że wtedy 25% akcji podejmowanych przez gracza było wybierane w sposób losowy, a 75% według najlepszej polityki. Mimo, że co 4 akcja była losowa, algorytm zaczął wtedy wygrywać znaczną większość poziomów. Podczas fazy uczenia, stan sieci neuronowej był zapisywany co 500 epizodów. Dla najlepszego modelu (Model 4) przetestowano poszczególne zrzuty sieci neuronowej i już model, który był zapisany w 2 000 epizodzie, był w stanie osiągnąć aż 85% skuteczności. Oznacza to, że zastosowanie takiej architektury znacznie przyspieszyło proces uczenia. Kolejne 7 500 epizodów pomogło podnieść skuteczność zaledwie o kilka procent. Rozbieżności pomiędzy zbiorami treningowymi i testowymi wynikają z parametru ϵ. Na początku trenowania algorytmu, wartość ϵ jest bliska 1, przez co znaczna większość akcji jest podejmowana w sposób losowy. Początkowe, losowe zachowanie agenta znacznie utrudnia przejście poziomu w początkowych fazach uczenia. Zostało to zobrazowane na wykresie (rys. 4.15), gdzie można zauważyć, że już przed 2000 epizodem, mimo, że ϵ znajdował się w przedziale 0,2 - 0,4, to algorytm był w stanie już wygrywać poziomy. W pracy umieszczono link do filmu, w którym przedstawiono działanie najlepszego modelu na zbiorze testowym [48].



Rys. 4.15. Wykres przedstawiający jak algorytm wygrywał poziomy wraz z wartością parametru ϵ w danych epizodach.

4.4. Etap 4 - generalizacja modelu na planszach 20x20

Ostatnim etapem uczenia algorytmu jest generalizacja modelu na mapach 20x20. W tym etapie również zdecydowano się wykorzystać sieci splotowe, jako, że uzyskały one znacznie lepsze rezultaty w ostatnich próbach na mapach 13x13. Większość parametrów wykorzystanych podczas uczenia była taka sama, jak przy ostatniej próbie dla plansz o wymiarach 13x13 (tabela 4.18). Z racji powiększenia mapy, musiał zwiększyć się limit iteracji na epizod, ponieważ agent potrzebuję wykonać więcej ruchów, aby przejść poziom. Limit ten wzrósł z 90 do 200. Uczenie zostało przeprowadzone w dwóch seriach - po 4 instancje, więc zwiększył się również rozmiar bufora - z 70 000 do 80 000. Mimo uruchomienia prawie dwukrotnie mniejszej ilości środowisk jednocześnie (4 zamiast 7), zmiana rozmiaru bufora jest niewielka, lecz należy pamiętać, że zwiększenie się rozmiaru mapy oznacza, że pojedynczy epizod zajmuje w buforze więcej - z powodu większego wymiaru tensora opisującego stan środowiska (z 13x13x5 do 20x20x5). Z racji zwiększenia rozmiaru map, zmieniły się dane, na podstawie których plansze są generowane. Parametry te przedstawiono w tabeli 4.20.

W tym etapie, zdecydowano się przetestować zarówno takie architektury sieci splotowych, które były skuteczne w poprzednim etapie, dla map 13x13, jak i nowe kombinacje. Model 5, przedstawiony na rysunku 4.16 posiada taką samą architekturę co model o najwyższej skuteczności dla plansz 13x13, a model 6 jest jego kopią ze zmienioną liczbą neuronów w warstwach końcowych. Dla modelu 6 zdecydowano się przetestować większą liczbę parametrów sieci, ponieważ rozmiar danych wejściowych zwiększył się ponad dwukrotnie (z 845 do 2000). Zdecydowano się również przetestować sieci o większej liczbie

Tabela 4.20. Parametry losowych map o wymiarach 20x20 (sieci splotowe).

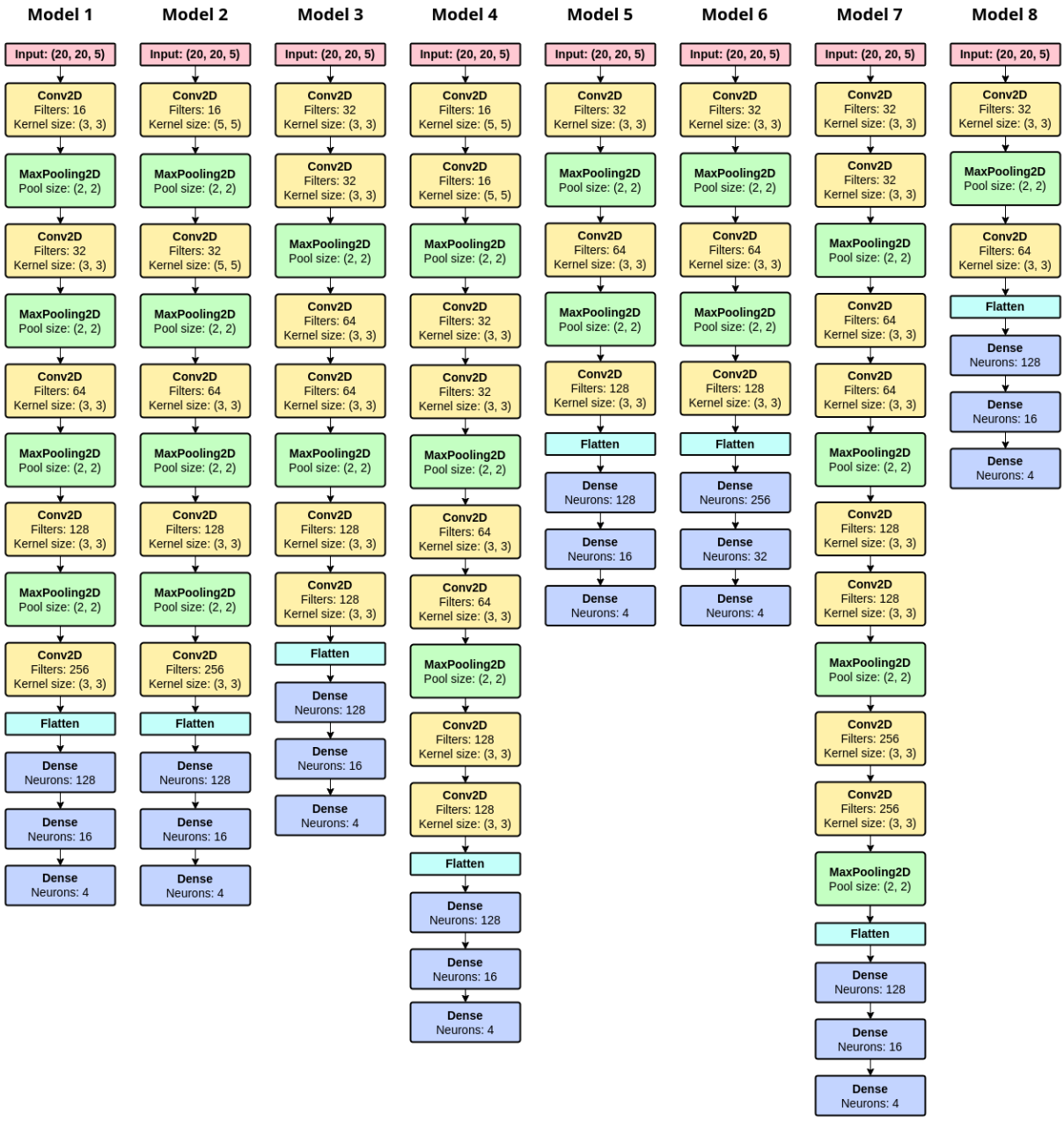
Liczba monet	5
Min. liczba ścian	1
Maks. liczba ścian	7
Min. wymiar ściany	2x1 / 1x2 (prawd. 50% zmiany orientacji)
Maks. wymiar ściany	5x1 / 1x5 (prawd. 50% zmiany orientacji)
Wymiar wyjścia z poziomu	1x1

warstw, z podwójnymi warstwami splotowymi (modele 4, 7-8), które swoją strukturą przypominają modele VGG. Reszta hiperparametrów, które nie zostały przedstawione na rysunku (np. aktywacja warstw, padding itd.) pozostały takie same jak w ostatniej próbie w poprzednim etapie uczenia.

Tabela 4.21. Współczynnik zwycięstw dla testowanych architektur sieci splotowych dla map 20x20.

Nr modelu	**% zwycięstw na zbiorze treningowym**	**% zwycięstw na zbiorze testowym**
Model 1	24,56%	46,9%
Model 2	0,15%	N/A
Model 3	32,95%	53,7%
Model 4	0%	N/A
Model 5	42,64%	64,4%
Model 6	41,68%	58,2%
Model 7	48,45%	**67,9%**
Model 8	20,32%	42,1%

Wyniki przedstawione w tabeli 4.21 prezentują się znacznie gorzej niż w przypadku plansz o wymiarach 13x13. Należy jednak pamiętać, że wraz ze wzrostem wymiaru map, rośnie ich poziom skomplikowania, ponieważ na mapie znajduje się więcej monet i więcej przeszkód. Podczas fazy uczenia i testowania zaobserwowano, że agent najgorzej radzi sobie w momencie, gdy na mapie pojawiają się wąskie przejścia (np. szerokości jednego bloku). Wtedy zazwyczaj się blokuje lub zapętla i nie potrafi wybrnąć, aby przejść poziom. Modele 2 i 4 jako jedyne, na pierwszych warstwach splotowych, posiadały rozmiar jądra (na rysunku 4.16 oznaczono jako *kernel size*) ustawiony na większą wartość niż (3, 3). W tym przypadku, prawdopodobnie rozmiar danych wejściowych był zbyt mały i zwiększenie parametru *kernel size* spowodowało praktycznie zerową skuteczność modelu na zbiorze treningowym. Z tego powodu, zrezygnowania z testowania modelu na mapach testowych. Model 5 posiada taką samą architekturę jak najlepszy model z poprzedniego etapu, który wtedy osiągnął ok. 90% skuteczności, jednak w przypadku większych map zaliczył spadek skuteczności o około 30 punktów procentowych. Dużo głębszy Model 7 z podwójnymi warstwami splotowymi spisał się lepiej, osiągając 67,9% skuteczności, co było najlepszym wynikiem. Liczba neuronów warstw ukrytych po wykonaniu "spłaszczenia" wydaje



Rys. 4.16. Architektury testowanych modeli sieci splotowych dla plansz 20x20.

się dobrze dobrana, ponieważ jej zwiększenie spowodowało spadek skuteczności modelu o około 6 pp. - mowa tutaj o Modelu 5 i 6.

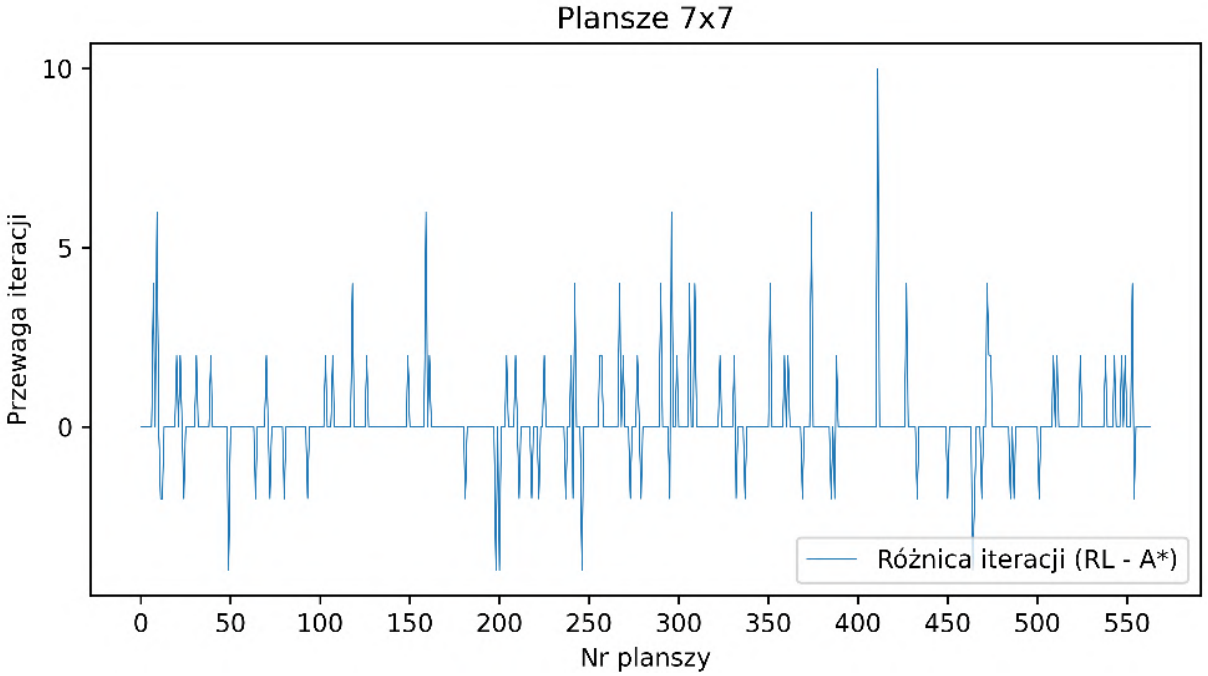
5. Testy gotowego algorytmu

Spośród wszystkich prób wykonanych na planszach o wymiarach 7x7, 13x13 oraz 20x20 wybrano najlepsze modele, które charakteryzowały się największym współczynnikiem zwycięstw. Modele te, porównano z algorytmem znajdywania najkrótszej ścieżki - **A*** [49]. Problem z zaimplementowaniem algorytmu sztucznej inteligencji do gry SmartSquareGame po części pokrywa się z problemem Komiwojażera (ang. *travelling salesman problem*), który polega na znalezieniu w grafie cyklu Hamiltona o minimalnym koszcie. Każde pole na planszy odpowiada wierzchołkowi grafu, który jest rozpatrywany w problemie. Przekładając to na grę, pojawia się problem - jak algorytm sztucznej inteligencji powinien ustalić trasę tak, żeby była ona jak najkrótsza, obejmowała wszystkie monety i kończyła się na wyjściu z poziomu. Algorytm A* jest dobrym punktem odniesienia, ponieważ jest bardzo często stosowany w grach dzięki swojej szybkości działania, która jest spowodowana wykorzystaniem heurystyki. Na pewno przewagą algorytmu A* nad modelem uczenia przez wzmocnienie jest fakt, że zawsze znajdzie trasę do celu - o ile ona istnieje, natomiast niekoniecznie będzie to trasa najkrótsza. Monety oraz wyjście z poziomu oznaczają wierzchołki docelowe, które algorytm musi odwiedzić. Istotną modyfikacją względem problemu Komiwojażera jest fakt, że algorytm nie musi odwiedzać wszystkich wierzchołków grafu (wszystkich pól na planszy), a jedynie te wierzchołki, które odpowiadają monetom oraz wyjściu z poziomu. Kolejność monet, z którą algorytm A* będzie je zbierał będzie wyznaczana na podstawie bezpośredniego dystansu między graczem, a monetą - algorytm zawsze będzie kierował się w pierwszej kolejności do tej monety, która jest bezwzględnie najbliżej. Takie podejście sprawia, że może wystąpić sytuacja, w której trasa będzie nieoptymalna - dla przykładu, najbliższa moneta może znajdować się 3 kratki od gracza, natomiast być za ścianą i być w rzeczywistości najdalej, przez co gracz będzie musiał się cofać na planszy, co wydłuży trasę ponad wartość optymalną. Sama trasa do pojedynczej monety, ze względu na charakterystykę algorytmu A*, będzie najkrótsza, jednak całkowita trasa - niekoniecznie. W testach zdecydowano się wykorzystać gotową implementację algorytmu A* [50].

5.1. Plansze 7x7

Na rysunku 5.1 przedstawiono wykres opisujący jaka była różnica w liczbie iteracji, które algorytm potrzebował do przejścia poziomu. Plansze, które zostały użyte do porównania są dobrane tak, żeby oba algorytmy były w stanie je wygrać. Różnica jest obliczana odejmując liczbę wykorzystanych iteracji przez algorytm A* od liczby iteracji wykorzystanych przez algorytm uczenia przez wzmocnienie (RL) na

tej samej mapie. Z tego powodu, wszystkie wartości dodatnie znajdujące się na wykresie oznaczają o ile iteracji szybciej trasę znalazł algorytm A*, natomiast ujemne wartości stanowią o przewadze algorytmu uczenia przez wzmacnianie. Dokładniejsze wartości zostały podane w tabeli 5.1.



Rys. 5.1. Różnica iteracji potrzebnych do przejścia poziomów przez algorytm uczenia przez wzmocnienie (RL), a algorytm A*.

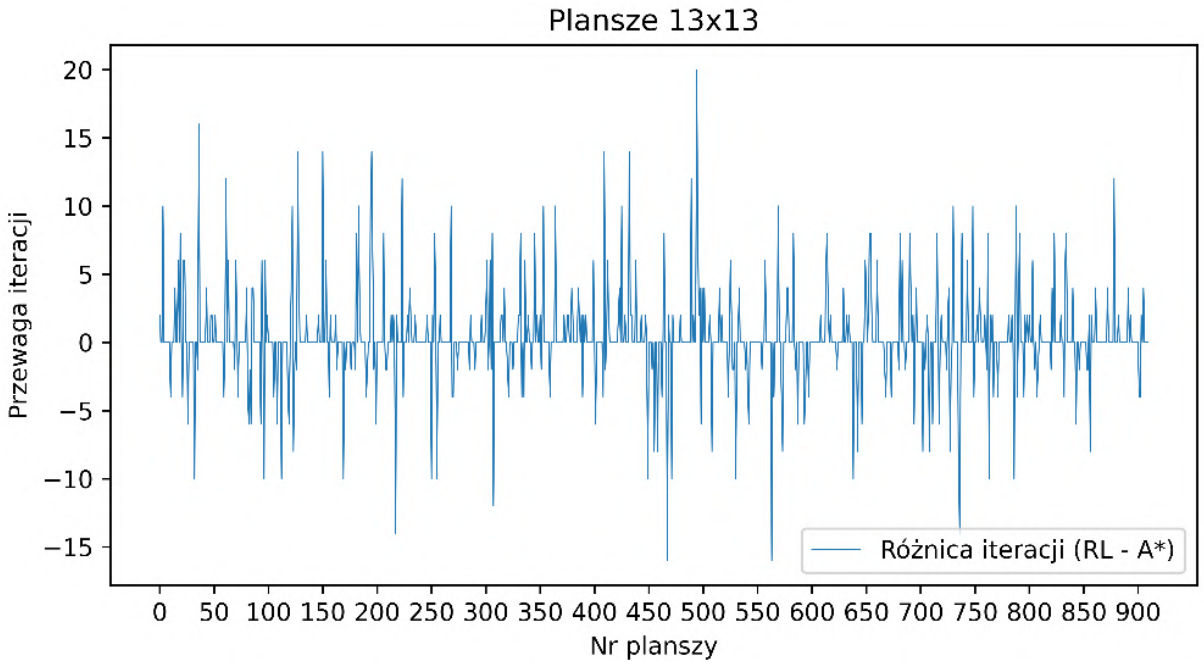
Tabela 5.1. Wyniki porównania A* i RL dla map 7x7.

Cecha	**Wartość**
Liczba testowanych poziomów	564
Liczba poziomów, w których RL był lepszy	34
Liczba poziomów, w których A* był lepszy	49
Liczba poziomów, w których nastąpił remis	481
Łączna liczba wykorzystanych iteracji (RL)	4660
Łączna liczba wykorzystanych iteracji (A*)	4594

W przypadku map 7x7 różnica w obu algorytmach jest niewielka. Algorytm uczenia przez wzmacnianie oraz algorytm A* uzyskały taką samą liczbę iteracji dla około 85% map, a łączna liczba wykorzystanych iteracji jest zaledwie o około 1,5% większa dla. Taka mała różnica wynika z małego rozmiaru map i niewielkiej liczby monet potrzebnych do zebrania. Analizując dane uzyskane podczas testowania zauważono, że znaczną część plansz, oba algorytmy były w stanie przejść w mniej niż 10 iteracji. Mimo wszystko, z niewielką przewagą pod kątem znajdowania najkrótszych tras wyszedł algorytm A*.

5.2. Plansze 13x13

W następnym kroku, przetestowano najlepszy model uzyskany dla map 13x13, który uzyskał 91% skuteczności. Ponownie obliczono różnicę wykorzystanych iteracji na poziom odejmując iteracje wykorzystane przez A* od iteracji wykorzystanych przez model uczenia przez wzmocnienie (RL). Wykres przewagi przedstawiono na rysunku 5.2.



Rys. 5.2. Różnica iteracji potrzebnych do przejścia poziomów przez algorytm uczenia przez wzmocnienie (RL), a algorytm A*.

Tabela 5.2. Wyniki porównania A* i RL dla map 13x13.

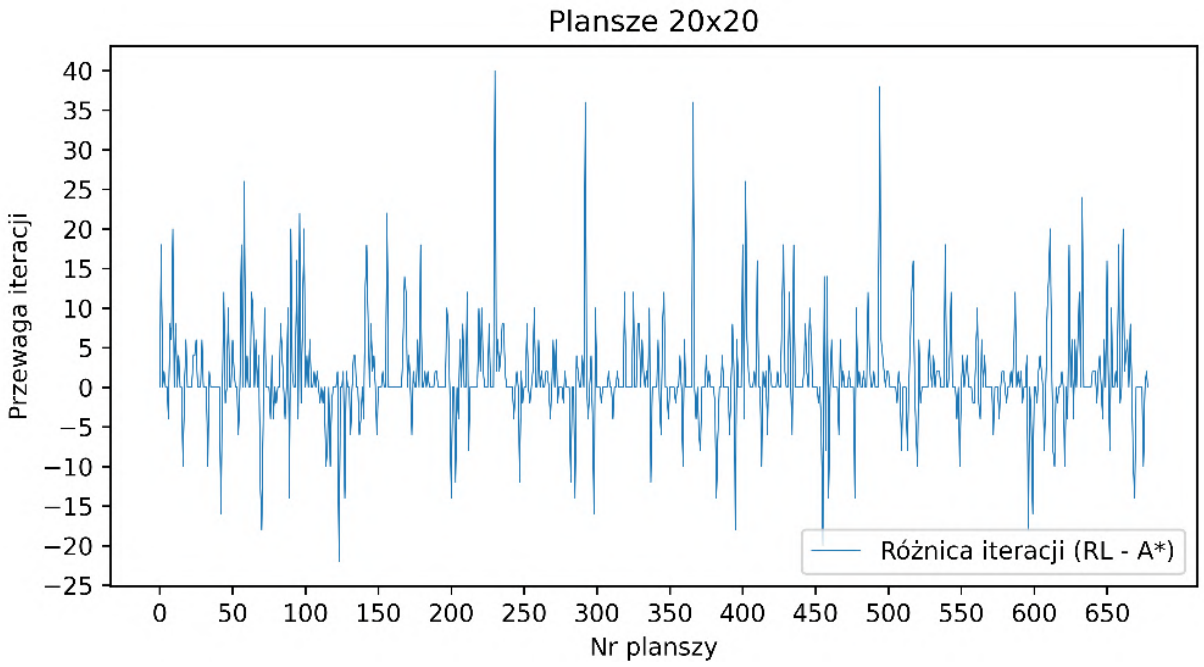
Cecha	Wartość
Liczba testowanych poziomów	910
Liczba poziomów, w których RL był lepszy	124
Liczba poziomów, w których A* był lepszy	185
Liczba poziomów, w których nastąpił remis	601
Łączna liczba wykorzystanych iteracji (RL)	25664
Łączna liczba wykorzystanych iteracji (A*)	25396

W tabeli 5.2 przedstawiono dokładniejsze wyniki, które zostały osiągnięte przez oba algorytmy. Tutaj sytuacja jest już bardziej rozbieżna, niż w przypadku mniejszych plansz 7x7. Łączna liczba wykorzystanych iteracji różni się o zaledwie 1%, natomiast znacznie częściej następywała sytuacja, w której jeden z algorytmów był lepszy od drugiego. Analizując wykres przedstawiony na rysunku 5.2 zauważono, że

najczęściej, przewaga iteracji wynosiła <10, jednak zdarzały się przypadki, w których jeden z algorytmów znalazł znacznie krótszą trasę od drugiego. Finalnie, 66% plansz zostało rozegranych przez oba algorytmy wykorzystując tę samą liczbę iteracji, natomiast w 20% przypadków, algorytm A* był lepszy. Model uczenia przez wzmocnienie uzyskał przewagę w 14% poziomów, co daje przewagę algorytmowi A* o ok. 6 punktów procentowych.

5.3. Plansze 20x20

W kolejnym etapie przetestowano działanie algorytmów na planszach o wymiarach 20x20. W tym przypadku, poziom skomplikowania map był największy, co przełożyło się szczególnie na skuteczność modelu uczenia przez wzmocnienie, który uzyskał zaledwie 67,9% skuteczności na 1000 poziomów. Rozbieżność w liczbie wykorzystanych iteracji również była największa, co zostało przedstawione na rysunku 5.3.



Rys. 5.3. Różnica iteracji potrzebnych do przejścia poziomów przez algorytm uczenia przez wzmocnienie (RL), a algorytm A*.

W tabeli 5.3 ponownie przedstawiono wyniki poszczególnych algorytmów i liczbę wykorzystanych iteracji. Model uczenia przez wzmocnienie wykorzystał średnio około 51,96 iteracji na poziom i był lepszy w ok. 15,7% poziomów, natomiast algorytm A* wykorzystywał średnio 50,67 iteracji na poziom i jego przewaga następowała w ok. 32.3% plansz. Przewaga jednego lub drugiego algorytmu bardzo często była niewielka, co widać na rysunku 5.3, jednak zdarzały się pojedyncze przypadki, gdzie jeden z algorytmów znajdował trasę, która wymagała ponad 15 iteracji mniej. W ogólnym rozrachunku i tak zwyciężył algorytm A*.

Tabela 5.3. Wyniki porównania A* i RL dla map 20x20.

Cecha	Wartość
Liczba testowanych poziomów	679
Liczba poziomów, w których RL był lepszy	107
Liczba poziomów, w których A* był lepszy	219
Liczba poziomów, w których nastąpił remis	353
Łączna liczba wykorzystanych iteracji (RL)	35278
Łączna liczba wykorzystanych iteracji (A*)	34408

5.4. Czasy obliczeń

W ostatnim kroku, porównano czasy potrzebne na wykonanie określonej liczby iteracji. Obliczenia zostały przeprowadzone komputerze wyposażonym w procesor Intel i5-6500 oraz kartę graficzną NVidia GTX 1060 6 GB. Wykorzystanie karty graficznej było kluczowe z punktu widzenia algorytmu uczenia maszynowego, ponieważ wykorzystana została biblioteka TensorFlowGPU, która w połączeniu z kartą graficzną wyposażoną w rdzenia CUDA, była w stanie znacznie przyspieszyć obliczenia. Wyniki zostały przedstawione w tabeli 5.4. Czas został zmierzony od momentu, gdy algorytm wykonał pierwszy ruch, tak aby czas ładowania modeli sieci neuronowych nie był brany pod uwagę.

Tabela 5.4. Czasy 1000 iteracji dla map o różnych rozmiarach.

	Czas 1000 iteracji dla RL [s]	Czas 1000 iteracji dla A* [s]
Plansze 7x7	28,32	2,13
Plansze 13x13	28,4	3,17
Plansze 20x20	28,54	7,67

W tabeli 5.4 można zauważyć, że czas potrzebny do wykonania 1000 iteracji znacznie zwiększa się wraz ze wzrostem rozmiaru planszy w przypadku algorytmu A*, natomiast dla algorytmu uczenia przez wzmocnienie, wzrost ten jest bardzo niewielki. Należy jednak zwrócić uwagę, że modele wykorzystane w algorytmie uczenia przez wzmocnienie dla wszystkich tych rozmiarów cechowały się podobnym rzędem wielkości jeśli chodzi o liczbę parametrów sieci. Liczba "trenowalnych" parametrów sieci dla map 7x7 wynosi 298 692, natomiast dla map 13x13 i 20x20 (struktura sieci jest identyczna dla obu rozmiarów plansz) wynosi 505 684. Modele dla map 13x13 i 20x20 wykorzystywały warstwy splotowe i posiadały ok. 200 000 więcej parametrów podlegających trenowaniu, jednak czasy obliczeń różniły się bardzo niewiele. Mimo tego, algorytm A* wykonywał obliczenia znacznie szybciej niż algorytm uczenia przez wzmocnienie.

Podsumowanie

W ramach pracy zaimplementowano grę SmartSquareGame oraz algorytm uczenia przez wzmacnianie bazujący na metodzie Double Deep Q-Learning. Utworzone modele charakteryzowały się bardzo dobrą skutecznością wygrywania losowo wygenerowanych plansz o małych (7x7) i średnich (13x13) rozmiarach - ponad 90%. Przy większym rozmiarze map (20x20), skuteczność algorytmu znacznie spadła, jednak wynik jest ponadprzeciętny - prawie 68%. Ogólny wynik jest bardzo dobry - model uczenia przez wzmacnianie skutecznie nauczył się wygrywać poziomy w grze SmartSquareGame.

Podczas trenowania algorytmu wykorzystano uproszczoną wersję gry. Wykorzystanie pełnego potencjału gry ze wszystkimi elementami mogłoby być dobrym punktem startowym do kontynuowania pracy, jak również udoskonalanie modeli, tak aby uzyskały jeszcze lepszą skuteczność, w szczególności na planszach o wymiarach 20x20. Praca pokazała, że architektura sieci neuronowych ma duży wpływ na skuteczność algorytmu. Na mniejszych mapach, sieci neuronowe bazujące na perceptronie (*fully-connected*) doskonale poradziły sobie z problemem, osiągając ok. 95% skuteczności, natomiast zwiększając rozmiar map, dotychczasowa architektura była niewystarczająca i lepiej sprawdziły się sieci splotowe, które są mniej podatne na przeuczenie. Dodatkowym punktem, który wymaga dalszej pracy jest zbadanie działania modeli na bardziej skomplikowanych środowiskach - takich, które posiadają znacznie więcej przeszkód i monet. Aktualne badania wykazały, że algorytm bardzo często gubi się, gdy w poziomie występuję ciasne korytarze i przejścia szerokości 1-2 pól, które trzeba przejść, aby dojść do celu. Skomplikowanie środowiska może się odbyć również poprzez dodanie dynamicznych elementów - poruszających się przeciwników, których trzeba omijać i przeszkody, które trzeba zniszczyć. Będzie to wymagać dodatkowych interakcji w środowisku oraz rozszerzenia przestrzeni stanów, co znacznie skomplikuje cały problem.

W trakcie trenowania modeli do gry SmartSquareGame wielokrotnie czas trwania treningu przekraczał 4-5 dni nieustannej pracy komputera. Przy bardziej skomplikowanych środowiskach, gdy plansza będzie zawierać więcej różnych obiektów, takich jak przeciwnicy, przeszkody, które trzeba zniszczyć itp. polityka będzie jeszcze trudniejsza do obliczenia, co będzie wymagało znacznie bardziej rozbudowanego modelu, większego zbioru treningowego i spowoduje znaczne wydłużenie oraz skomplikowanie treningu.

W pracy dokonano również porównania modelu uczenia przez wzmacnianie z algorytmem znajdowania trasy A*. Algorytm A* zyskiwał przewagę, częściej znajdując krótsze trasy. Należy jednak zaznaczyć, że model uczenia przez wzmocnienie charakteryzuje się znacznie większa uniwersalnością.

Zastosowanie algorytmu A* jest ograniczone praktycznie tylko do znajdywania tras, natomiast algorytmy uczenia przez wzmacnianie są w stanie poradzić sobie ze znacznie większą liczbą środowisk, które charakteryzują się różną dynamiką i problemem.

Przyszłość algorytmów uczenia przez wzmacnianie zdaje się zbiegać w kierunku modeli bazujących na architekturze Transformer, które odnoszą znaczne sukcesy szczególnie w przetwarzaniu języka naturalnego. Badania pokazują, że implementacja tzw. transformerów decyzyjnych, które przekształcają problematykę uczenia przez wzmacnianie do modelowania sekwencji warunkowych pozwala osiągnąć taką samą lub znacznie lepszą skuteczność w środowiskach typu ATARI, OpenAI Gym czy Key-To-Door niż najlepsze dotychczasowe, klasyczne modele uczenia przez wzmacnianie [51].

Bibliografia

[1] Bangalore Ravi Kiran i in. „Deep Reinforcement Learning for Autonomous Driving: A Survey". W: *CoRR* abs/2002.00444 (2020). arXiv: *2002.00444*.

[2] *Safety-first AI for autonomous data centre cooling and industrial control*. *https://www.deepmind.com/blog/safety-first-ai-for-autonomous-data-centre-cooling-and-industrial-control*. [Online; dostęp 13-12-2022].

[3] Chao Yu, Jiming Liu i Shamim Nemati. „Reinforcement Learning in Healthcare: A Survey". W: *CoRR* abs/1908.08796 (2019). arXiv: *1908.08796*.

[4] Romain Paulus, Caiming Xiong i Richard Socher. „A Deep Reinforced Model for Abstractive Summarization". W: *CoRR* abs/1705.04304 (2017). arXiv: *1705.04304*.

[5] Aishwarya Srinivasan. *The first of its kind AI Model- Samuel's Checkers Playing Program*. *https://medium.com/ibm-data-ai/the-first-of-its-kind-ai-model-samuels-checkers-playing-program-1b712fa4ab96*. [Online; dostęp 31-10-2022]. 2020.

[6] Yanzhao Wu i in. „A Comparative Measurement Study of Deep Learning as a Service Framework". W: *IEEE Transactions on Services Computing* 15.1 (2022), s. 551–566. DOI: *10.1109/TSC.2019.2928551*.

[7] *Timeline of DeepMind*. *https://timelines.issarice.com/wiki/Timeline_of_DeepMind*. [Online; dostęp 06-12-2022].

[8] DeepMind. *AlphaGo*. *https://www.deepmind.com/research/highlighted-research/alphago*. [Online; dostęp 06-11-2022].

[9] DeepMind. *Research*. *https://www.deepmind.com/research*. [Online; dostęp 06-11-2022].

[10] OpenAI. *Customizing GPT-3 for Your Application*. *https://openai.com/blog/customized-gpt-3/*. [Online; dostęp 11-12-2022].

[11] OpenAI. *Dall-E 2*. *https://openai.com/dall-e-2/*. [Online; dostęp 06-11-2022].

[12] Vahid Mirjalili Sebastian Raschka. *Python. Machine learning i deep learning. Wydanie III*. Wydawnictwo Helion, 2021.

[13] *Reinforcement Learning Agents - Matlab documentation. https://nl.mathworks.com/help/reinforcement-learning/ug/create-agents-for-reinforcement-learning.html.* [Online; dostęp 31-10-2022].

[14] Zhenpeng Zhou, Xiaocheng Li i Richard N. Zare. „Optimizing Chemical Reactions with Deep Reinforcement Learning". W: *ACS Central Science* 3.12 (2017). PMID: 29296675, s. 1337–1344. DOI: *10.1021/acscentsci.7b00492*.

[15] Sebastian Dittert. *Reinforcement Learning: Value Function and Policy. https://medium.com/analytics-vidhya/reinforcement-learning-value-function-and-policy-c22f5bd1d1b0.* [Online; dostęp 04-11-2022]. 2020.

[16] Eugen Lindwurm. *Intuition: Exploration vs Exploitation. https://towardsdatascience.com/intuition-exploration-vs-exploitation-c645a1d37c7a.* [Online; dostęp 04-11-2022]. 2019.

[17] baeldung. *Epsilon-Greedy Q-learning. https://www.baeldung.com/cs/epsilon-greedy-q-learning.* [Online; dostęp 04-11-2022]. 2021.

[18] *Gym Documentation - Cart Pole. https://www.gymlibrary.dev/environments/classic_control/cart_pole/.* [Online; dostęp 01-11-2022].

[19] *Gym Documentation - Mountain Car Continuous. https://www.gymlibrary.dev/environments/classic_control/mountain_car_continuous/.* [Online; dostęp 01-11-2022].

[20] *Reinforcement Learning Coach. https://intellabs.github.io/coach/selecting_an_algorithm.html.* [Online; dostęp 01-11-2022].

[21] Christopher J. C. H. Watkins i Peter Dayan. „Q-learning". W: *Machine Learning* 8.3 (1992), s. 279–292. ISSN: 1573-0565. DOI: *10.1007/BF00992698*.

[22] Deepshikha Pandey i Punit Pandey. „Approximate Q-Learning: An Introduction". W: *2010 Second International Conference on Machine Learning and Computing*. 2010, s. 317–320. DOI: *10.1109/ICMLC.2010.38*.

[23] Sebastian Thrun i A. Schwartz. „Issues in Using Function Approximation for Reinforcement Learning". W: *Proceedings of 4th Connectionist Models Summer School*. Red. D. Touretzky J. Elman M. Mozer P. Smolensky i A. Weigend. Erlbaum Associates, 1993.

[24] Volodymyr Mnih i in. *Playing Atari with Deep Reinforcement Learning*. 2013. DOI: *10.48550/ARXIV.1312.5602*.

[25] William Fedus i in. *Revisiting Fundamentals of Experience Replay*. 2020. DOI: *10.48550/ARXIV.2007.06700*.

[26] TF-Agents. *Introduction to RL and Deep Q Networks. https://www.tensorflow.org/agents/tutorials/0_intro_rl*. [Online; dostęp 10-11-2022].

[27] *Temporal difference learning (TD Learning). https://www.engati.com/glossary/temporal-difference-learning*. [Online; dostęp 12-12-2022].

[28] Hado van Hasselt, Arthur Guez i David Silver. *Deep Reinforcement Learning with Double Q-learning*. 2015. DOI: *10.48550/ARXIV.1509.06461*.

[29] Volodymyr Mnih i in. „Human-level control through deep reinforcement learning". W: *Nature* 518.7540 (2015), s. 529–533. ISSN: 1476-4687. DOI: *10.1038/nature14236*.

[30] Christof Koch. *How the Computer Beat the Go Master. https://www.scientificamerican.com/article/how-the-computer-beat-the-go-master/*. [Online; dostęp 24-11-2022].

[31] David Silver i in. „Mastering the game of Go without human knowledge". W: *Nature* 550.7676 (2017), s. 354–359. ISSN: 1476-4687. DOI: *10.1038/nature24270*.

[32] *Go Ratings. https://www.goratings.org/en/*. [Online; dostęp 24-11-2022].

[33] Marwin H. S. Segler, Mike Preuss i Mark P. Waller. „Planning chemical syntheses with deep neural networks and symbolic AI". W: *Nature* 555.7698 (2018), s. 604–610. ISSN: 1476-4687. DOI: *10.1038/nature25978*.

[34] Mogens Dalgaard i in. „Global optimization of quantum dynamics with AlphaZero deep exploration". W: *npj Quantum Information* 6.1 (2020), s. 6. ISSN: 2056-6387. DOI: *10.1038/s41534-019-0241-0*.

[35] DeepMind. *MuZero: Mastering Go, chess, shogi and Atari without rules. https://www.deepmind.com/blog/muzero-mastering-go-chess-shogi-and-atari-without-rules*. [Online; dostęp 11-12-2022].

[36] *OpenAI Gym - documentation. https://www.gymlibrary.dev/*. [Online; dostęp 09-12-2022].

[37] Gigazine. *"The World's Hardest Game" which is the most difficult game in the world. https://gigazine.net/gsc_news/en/20080430_the_worlds_hardest_game*. [Online; dostęp 11-11-2022]. 2008.

[38] *SmartSquareGame - repozytorium GitHub. https://github.com/dkwapisz/SmartSquareGame*. [Online; dostęp 09-12-2022].

[39] *Reinforcement Learning Implementations - GitHub. https://github.com/philtabor/Youtube-Code-Repository/tree/master/ReinforcementLearning/DeepQLearning*. [Online; dostęp 10-12-2022].

[40] Jason Brownlee. *Why One-Hot Encode Data in Machine Learning? https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/*. [Online; dostęp 15-11-2022].

[41] gRPC documentation. *Core concepts, architecture and lifecycle. https://grpc.io/docs/what-is-grpc/core-concepts*. [Online; dostęp 13-11-2022].

[42] *SmartSquareRL - repozytorium GitHub. https://github.com/dkwapisz/SmartSquareRL*. [Online; dostęp 13-12-2022].

[43] Kumar Chandrakant. *Reinforcement Learning with Neural Network. https://www.baeldung.com/cs/reinforcement-learning-neural-network*. [Online; dostęp 11-12-2022].

[44] Dawid Kwapisz. *AI solving level. https://www.youtube.com/shorts/34nVu4QA4Cc*. [Online; dostęp 11-12-2022].

[45] Dawid Kwapisz. *AI generalization - Map 7x7. https://youtu.be/te7rVuNhfH0*. [Online; dostęp 11-12-2022].

[46] Karen Simonyan i Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2014. DOI: *10.48550/ARXIV.1409.1556*.

[47] Sumit Saha. *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53*. [Online; dostęp 11-12-2022].

[48] Dawid Kwapisz. *AI generalization - Map 13x13. https://youtu.be/pOqKntf8jZQ*. [Online; dostęp 11-12-2022].

[49] *Introduction to A* Pathfinding. https://www.kodeco.com/3016-introduction-to-a-pathfinding*. [Online; dostęp 06-12-2022].

[50] *Applying the A* Path Finding Algorithm in Python (Part 1: 2D square grid). https://www.analytics-link.com/post/2018/09/14/applying-the-a-path-finding-algorithm-in-python-part-1-2d-square-grid*. [Online; dostęp 06-12-2022].

[51] Lili Chen i in. „Decision Transformer: Reinforcement Learning via Sequence Modeling". W: *CoRR* abs/2106.01345 (2021). arXiv: *2106.01345*.