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

Kierunek: **Informatyka**
Specjalność: -

Mateusz Murawski
nr albumu: 303840

System automatycznego rozgrywania gry typu snake

Self-play system for a snake-type game

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

Lublin, rok 2023

Spis treści

Wstęp

W dzisiejszych czasach rozwój technologiczny niesie ze sobą wiele możliwości w dziedzinie informatyki. Jednym z obszarów, który wzbudza szczególne zainteresowanie, jest sztuczna inteligencja. Tworzeni są coraz bardziej inteligentni agenci zdolni do podejmowania decyzji w różnych dziedzinach, w tym w świecie gier komputerowych. Gry komputerowe stanowią idealne środowisko do badania możliwości sztucznej inteligencji, ponieważ pozwalają na symulację różnorodnych problemów występujących w rzeczywistym świecie, jednocześnie dając całkowitą kontrolę nad otoczeniem.

Celem niniejszej pracy licencjackiej jest stworzenie agenta, który będzie potrafił nauczyć się grać w popularną grę Wąż (ang. *Snake)* i zdobywać coraz większą liczbę punktów w miarę postępującego treningu. Gra Wąż, mimo swojej pozornej prostoty, stawia przed agentem wyzwania, które wymagają zdolności do skutecznego manewrowania w dynamicznym środowisku, zbierania punktów i unikania kolizji ze ścianami oraz ze sobą samym. Cały projekt wykonam w języku *Python*.

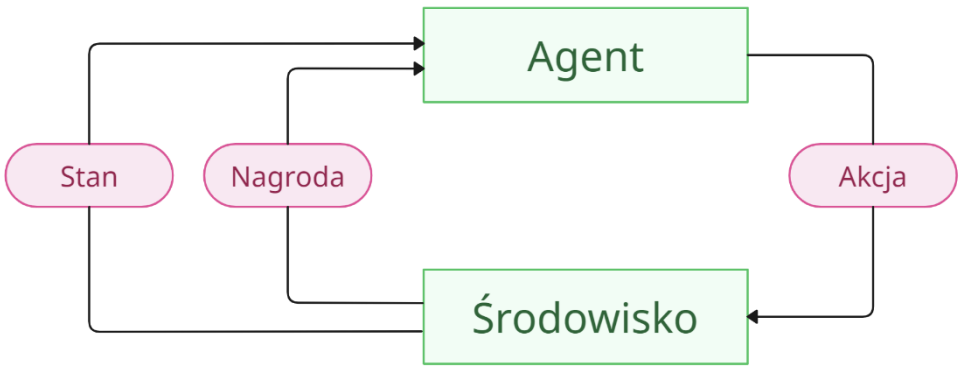
W pierwszym rozdziale omówię teorię związaną z uczeniem przez wzmacnianie. Rozpocznę od wprowadzenia, przedstawiając główne założenia i ideę działania tego podejścia. Następnie skoncentruję się na algorytmie *Q-Learning* [1] oraz na bardziej zaawansowanej metodzie, jaką jest *Deep Q-Learning* [2]. Omówię także konkretne implementacje *Deep Q-Learningu*, takie jak *Deep Q-Network* (*DQN*) [3] i jego różne warianty. Na zakończenie tego rozdziału przedstawię dotychczasowe osiągnięcia *DQN* w grze Wąż [4-5]. W drugim rozdziale omówię środowisko, które stworzyłem w celu dokładnej oceny działania agenta i umożliwienia testowania go w kontrolowanych warunkach. Dzięki temu będę mógł przeprowadzić szereg prób i zbadać zachowanie agenta. Środowisko to zostało stworzone w języku *Python*, przy wykorzystaniu głównie biblioteki *Pygame* [6], która umożliwia tworzenie gier w prosty sposób. W trzecim rozdziale przedstawię moją implementację agenta opartego na algorytmie *DQN*, wraz z różnymi technikami poprawiającymi jego skuteczność. Zaproponuję również własny model sieci konwolucyjnej (*CNN* [7]), który posłuży do reprezentacji stanów gry. Do implementacji agenta wykorzystam język programowania *Python* oraz bibliotekę *PyTorch* [8], która umożliwia tworzenie i trenowanie sieci neuronowych. Ponadto, zaimplementuję trzy różne warianty klasycznego *DQN*. W czwartym rozdziale omówię oraz przeprowadzę analizę wyników uzyskanych przez poszczególne warianty algorytmu *DQN*. Przeprowadzę porównanie wyników mojego agenta z innymi agentami z różnych prac naukowych [4-5].

Celem tego kroku jest zbadanie skuteczności i wydajności różnych wersji algorytmu w kontekście gry Wąż. Dokładnie przeanalizuję uzyskane wyniki, uwzględniając aspekty takie jak liczba zdobytych punktów, czas uczenia oraz stabilność działania agentów. Na podstawie tych analiz wyciągnę wnioski dotyczące efektywności poszczególnych wariantów algorytmu *DQN*.

Rozdział 1. Nauczanie przez wzmacnianie

1.1. Wstęp do nauczania przez wzmacnianie

Uczenie przez wzmacnianie (ang. *reinforcement learning*) jest gałęzią sztucznej inteligencji, która bada sposób, w jaki agent podejmuje decyzje w dynamicznym środowisku w celu maksymalizacji zdyskontowanej sumy nagród. Jest to metoda uczenia maszynowego, w której agent interaktywnie uczestniczy w środowisku, podejmuje akcje i na podstawie otrzymywanych nagród znajduje optymalne strategie. W uczeniu przez wzmacnianie agent jest umieszczany w środowisku, które zazwyczaj jest opisane jako proces decyzyjny Markowa (*MDP*). Środowisko definiuje zestaw stanów, w których agent może się znajdować, dostępne akcje, jakie może podjąć, prawdopodobieństwa przejść między stanami oraz funkcję nagrody, która ocenia jakość wykonanych akcji. Celem agenta jest nauczenie się strategii, która maksymalizuje oczekiwaną sumę nagród, jaką otrzyma w długim okresie czasu. Proces uczenia przez wzmacnianie składa się z kolejnych cykli. W każdym cyklu agent obserwuje stan środowiska, podejmuje akcję na podstawie aktualnej strategii, otrzymuje nagrodę i przechodzi do nowego stanu. Rysunek 1.1 przedstawia opisywany cykl. Agent aktualizuje swoją wiedzę o środowisku, korzystając z technik opartych na wartościach (ang. *value-based*) lub polityce (ang. *policy-based*), aby uaktualnić swoją strategię w celu osiągnięcia lepszych wyników. Agent może eksplorować różne akcje i obserwować ich skutki, aby zrozumieć, które działania prowadzą do pożądanych rezultatów. Przykładowymi algorytmami uczenia przez wzmacnianie są *Q-learning*, *SARSA*, *A2C* i *DQN*.



Rysunek 1.1: Schemat przedstawiający cykl w uczeniu przez wzmacnianie.

1.2. Algorytm *Q-Learning*

Q-learning jest popularnym algorytmem w dziedzinie uczenia ze wzmocnieniem, opartym na wartościach (ang. *value-based*), który umożliwia agentowi naukę optymalnych strategii w dynamicznym środowisku. Algorytm ten opiera się na programowaniu dynamicznym i polega na iteracyjnej optymalizacji funkcji wartości akcji, znanej jako funkcja *Q*.

Algorytm *Q-learningu* polega na aktualizowaniu funkcji wartości akcji (funkcji *Q*) w procesie iteracyjnym. Funkcja *Q* jest tablicą, która przypisuje wartość każdej parze stanu i akcji. Początkowo funkcja *Q* jest inicjalizowana losowo lub zerowo dla wszystkich par stanu i akcji. Ilość możliwych stanów i akcji jest zależna od konkretnego środowiska, na przykład w grze Wąż mamy zazwyczaj 4 dostępne akcje (ruch w górę, w dół, w prawo i w lewo) oraz dużą ilość stanów, zależną głównie od wielkości planszy gry. Funkcje *Q* oraz cały przebieg pojedynczego cyklu algorytmu przedstawia Rysunek 1.2. W każdym kroku iteracji agent obserwuje aktualny stan środowiska, wykonuje akcję na podstawie strategii eksploracji/eksploatacji i otrzymuje nagrodę oraz następny stan. Następnie aktualizuje wartość funkcji *Q* dla pary stanu i akcji, używając wzoru 1.1.

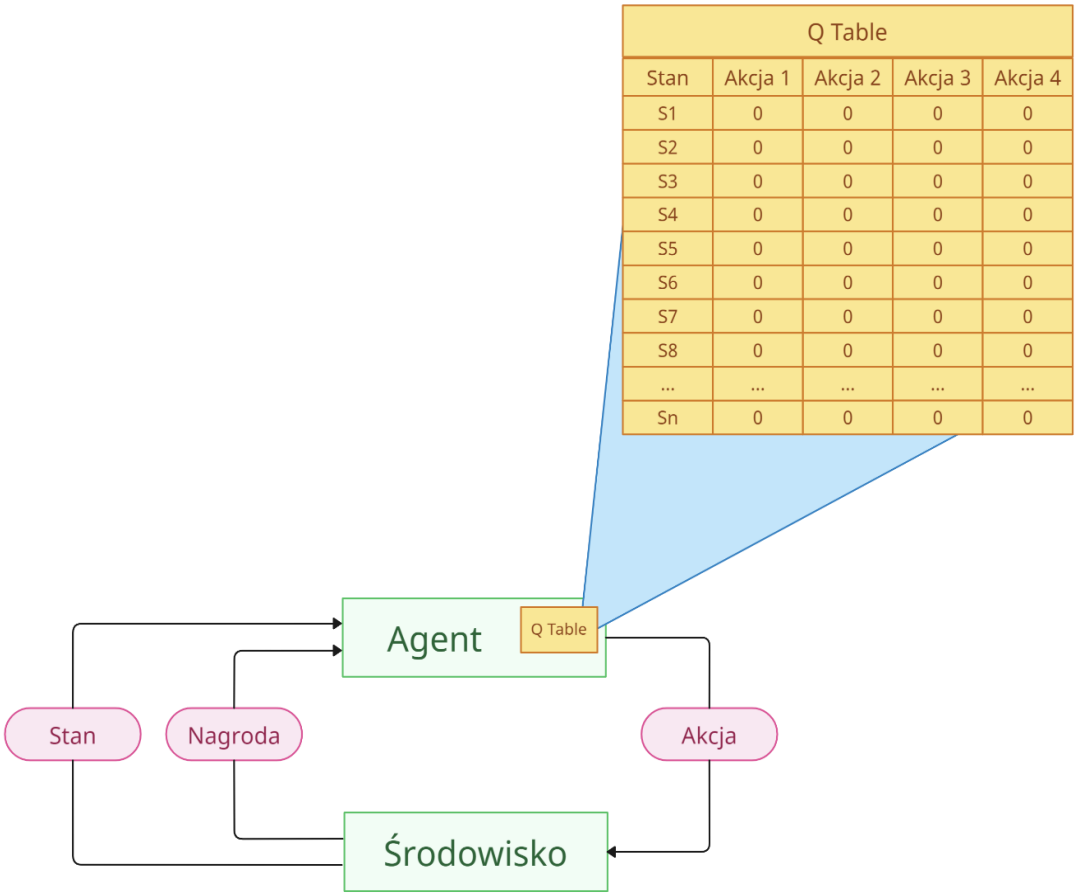
$$Q(s, a) = Q(s, a) + \alpha * (r + \gamma * max[Q(s', a')] - Q(s, a)) \quad (1.1)$$

Gdzie:

- *Q(s, a)* to wartość funkcji *Q* dla pary stanu *s* i akcji *a*,
- α to współczynnik uczenia (ang. *learning rate*), który kontroluje jak bardzo agent uwzględnia nowe informacje w porównaniu z tym, co już zna,
- *r* to nagroda otrzymana po wykonaniu akcji *a* ze stanu *s*,
- γ to współczynnik dyskontujący, który określa, jak bardzo agent kieruje się przyszłymi nagrodami w porównaniu do natychmiastowych nagród,
- *max[Q(s', a')]* to maksymalna wartość funkcji *Q* dla wszystkich dostępnych akcji ze stanu *s'*. Stan *s'* jest następnym stanem, do którego agent przechodzi po wykonaniu akcji *a*.

Proces iteracyjny trwa, dopóki agent nie nauczy się optymalnych strategii lub do osiągnięcia zadanego warunku stopu. Po wielu iteracjach wartości funkcji *Q* zbiegają do optymalnych wartości, które reprezentują optymalną strategię dla agenta w danym środowisku.

Q-learning jest algorytmem *off-policy*, co oznacza, że agent uczy się na podstawie swoich doświadczeń, niekoniecznie zgodnie ze strategią, którą aktualnie stosuje. Pozwala to na równoczesne eksplorowanie i eksploatowanie wiedzy.



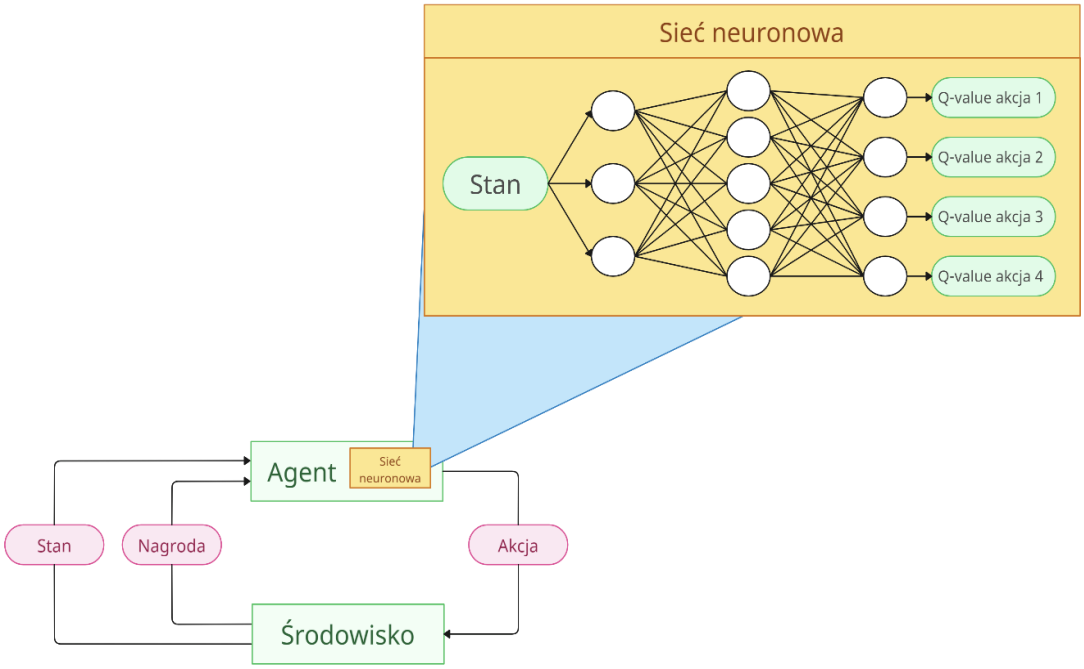
Rysunek 1.2: Schemat przedstawiający cykl oraz funkcje *Q* w algorytmie *Q-Learning*.

1.3. Algorytm *Deep Q-Learning*

Deep Q-Learning to zaawansowana metoda uczenia ze wzmocnieniem, która łączy techniki *Q-Learningu* z głębokimi sieciami neuronowymi (ang. *Deep Neural Networks*). Jej głównym celem jest nauka optymalnej strategii decyzyjnej dla agenta, który podejmuje akcje w dynamicznym środowisku w celu maksymalizacji zdyskontowanej sumy nagród.

W *Deep Q-Learningu*, podobnie jak w klasycznym *Q-Learningu*, agent działa w otoczeniu, w którym wykonuje akcje i obserwuje stan środowiska. Agent podejmuje decyzje

na podstawie swojej strategii decyzyjnej, która jest reprezentowana przez funkcję wartości akcji, oznaczaną jako *Q*. Funkcja *Q(s, a)* szacuje oczekiwaną sumę nagród, jaką agent otrzyma po wykonaniu akcji *a* w stanie *s*. Główna różnica między klasycznym *Q-Learningiem* a *Deep Q-Learningiem* polega na sposobie reprezentacji funkcji *Q*. Różnicę tą możemy zaobserwować na Rysunku 1.3. W *Deep Q-Learningu*, funkcję *Q* aproksymuje się za pomocą głębokiej sieci neuronowej, która jest zdolna do nauki skomplikowanych zależności między stanami a akcjami. Wykorzystanie głębokich sieci neuronowych umożliwia automatyczne wyodrębnienie istotnych cech i reprezentacji stanów, co pozwala na naukę bardziej skomplikowanych strategii decyzyjnych. Proces uczenia *Deep Q-Learningu* polega na iteracyjnym aktualizowaniu funkcji wartości *Q* na podstawie doświadczeń zgromadzonych przez agenta w trakcie interakcji ze środowiskiem. Agent eksploruje środowisko, podejmuje akcje, obserwuje nagrody oraz nowe stany, a następnie aktualizuje wartości *Q* w celu ulepszenia strategii decyzyjnej.



Rysunek 1.3: Schemat przedstawiający cykl oraz funkcje *Q* w algorytmie *Deep Q-Learning*.

1.4. Różne implementacje algorytmu *Deep Q-Learning*

1.4.1. Algorytm *DQN*

Deep Q-Network (*DQN*) to konkretna implementacja *Deep Q-Learningu*, która wykorzystuje głęboką sieć neuronową jako aproksymator funkcji wartości *Q*. *DQN* został zaprojektowany w celu skutecznego uczenia strategii decyzyjnych w przypadkach, gdzie przestrzenie stanów i akcji są duże. W przypadku gry Wąż, gdzie wejściem sieci jest obrazek o rozmiarze nxn pikseli, a wyjściem są cztery kierunki (góra, dół, lewo, prawo), architektura sieci dla *DQN* może wyglądać następująco (Rysunek 1.4):

- Warstwa wejściowa: Przyjmuje obrazek o rozmiarze nxn pikseli. Każdy piksel może reprezentować różne informacje, takie jak obecność ściany, jedzenia, ciała węża, itp.
- Opcjonalne warstwy konwolucyjne: sieć może zawierać jedną lub kilka warstw konwolucyjnych, które pomagają w wyodrębnianiu cech z obrazka wejściowego. Warstwy konwolucyjne są w stanie wykrywać wzorce i struktury w obrazku, które są istotne dla podejmowania decyzji.
- Warstwy w pełni połączone: po warstwach konwolucyjnych, obrazy są spłaszczane i przekazywane do jednej lub kilku warstw w pełni połączonych (ang. *fully connected*). Te warstwy są odpowiedzialne za uczenie się zależności między wyodrębnionymi cechami a oczekiwanymi *Q-wartościami* dla poszczególnych akcji.
- Warstwa wyjściowa: Na końcu sieci znajduje się warstwa wyjściowa, która generuje *Q-wartości* dla czterech dostępnych kierunków (góra, dół, lewo, prawo). Wyjście z tej warstwy jest używane do podejmowania decyzji, wybierając akcję z najwyższą *Q-wartością*.

W przypadku gry Wąż, gdy agent korzysta z wyuczonej sieci *DQN* podczas rozgrywki (czyli gdy się nie uczy), proces podejmowania decyzji wygląda następująco: agent otrzymuje obrazek planszy gry jako wejście i przekazuje go do sieci neuronowej/konwolucyjnej *DQN*. Sieć oblicza *Q-wartości* dla wszystkich czterech kierunków, a następnie agent wybiera akcję o najwyższej *Q-wartości* jako swoją decyzję. Po wykonaniu tej akcji, agent obserwuje nagrodę i nowy stan gry, które są przekazywane z powrotem do *DQN*.

Aby trenować sieć używaną w *DQN*, wykorzystuje się funkcję straty, która jest zwykle definiowana jako błąd średniokwadratowy (*MSE*) między oczekiwanymi *Q-wartościami* a

obliczonymi *Q-wartościami* dla danych przykładów. Oczekiwane *Q-wartości* są obliczane w oparciu o równanie Bellmana 1.2.

Aby skutecznie trenować sieć *DQN*, często wykorzystuje się partie danych (ang. *batch)*. W każdej iteracji uczącej losowo wybiera się próbkę przykładów z bufora pamięci, który przechowuje doświadczenia agenta. Umożliwia to efektywne wykorzystanie zbioru danych, a także redukcję korelacji między kolejnymi przykładami.

Bufor pamięci (ang. *replay buffer*) [9] jest wykorzystywany do przechowywania doświadczeń agenta w formie par (stan, akcja, nagroda, kolejny stan). Podczas treningu, doświadczenia są losowo pobierane z bufora pamięci, aby zbudować *batch* do aktualizacji sieci *DQN*. Dzięki temu, agent może uczyć się na podstawie wcześniejszych doświadczeń, co poprawia stabilność i skuteczność procesu uczenia.

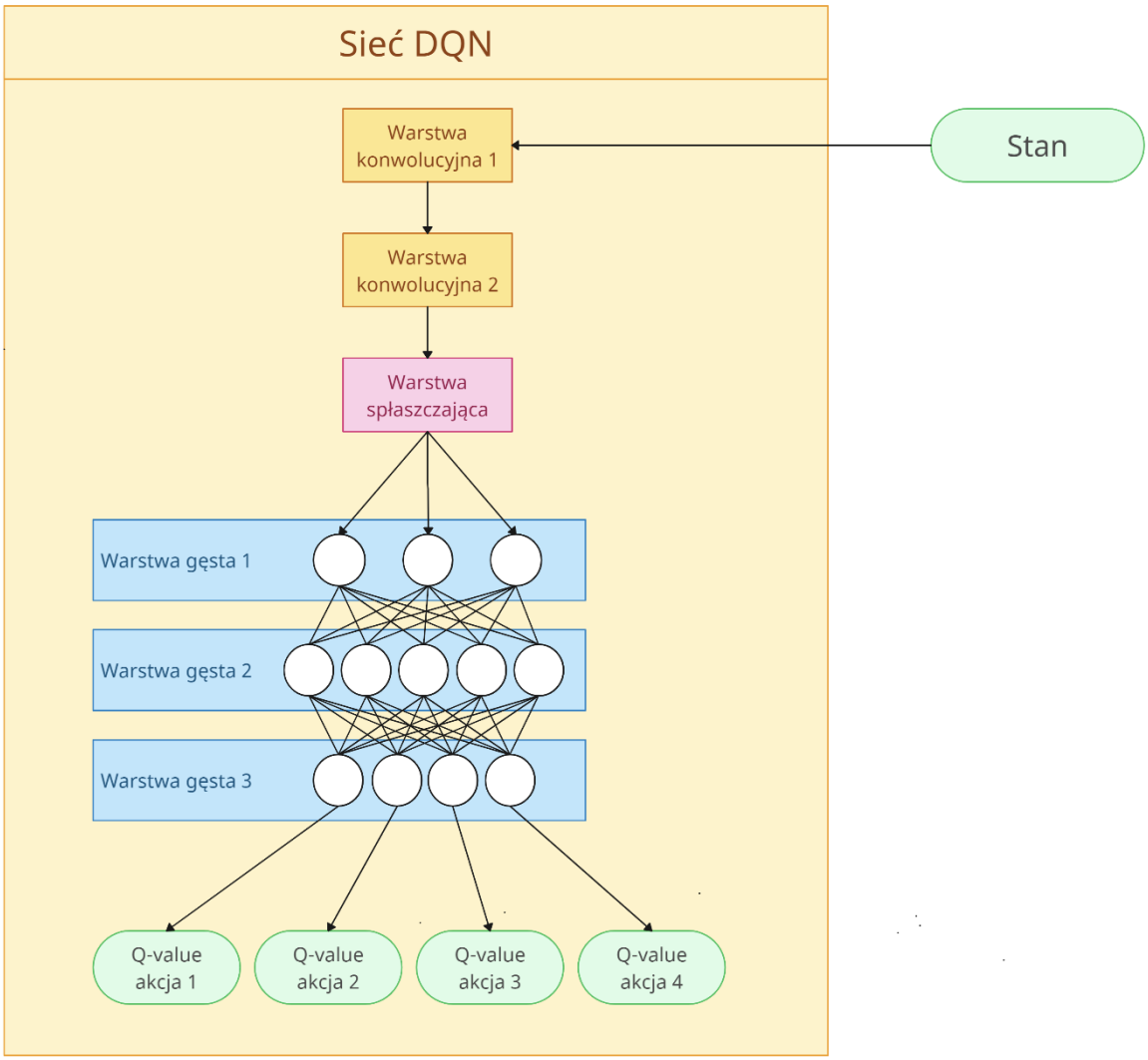
W skrócie, proces nauki *DQN* z wykorzystaniem bufora pamięci i *batchów* wygląda następująco: agent eksploruje środowisko, wykonuje akcje, obserwuje nagrodę i nowy stan, a następnie zapisuje doświadczenie do bufora pamięci. Następnie, losowo pobiera próbkę z bufora pamięci i aktualizuje sieć *DQN*, za pomocą funkcji straty, która jest zwykle definiowana wzorem 1.3. Cały ten proces obrazuje Rysunek 1.5. Ten cykl jest powtarzany wielokrotnie, aby agent stopniowo poprawiał swoją strategię decyzyjną.

$$Oczekiwana\ wartość\ Q = r + y * maxQ(s', a') \tag{1.2}$$

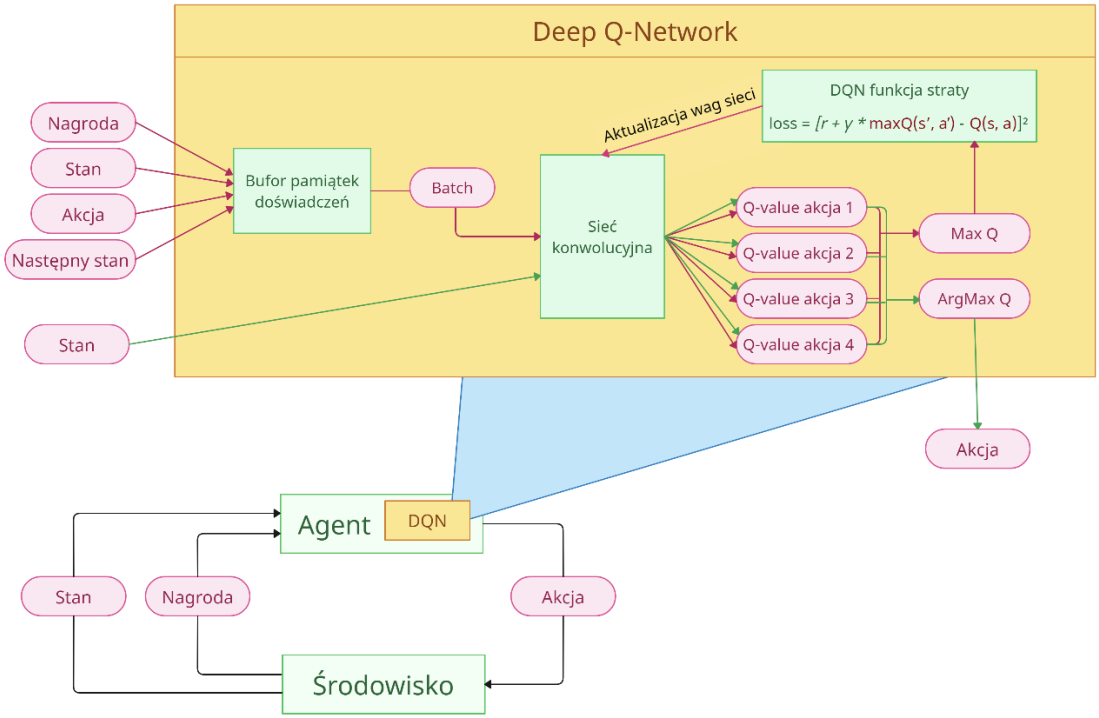
$$Funkcja\ straty = [r + y * maxQ(s', a') - Q(s, a)]^2 \tag{1.3}$$

gdzie:

- *a* to akcja,
- *s* to stan,
- *r* to nagroda za wykonanie akcji *a* w stanie *s*,
- *γ* to współczynnik dyskontowania,
- *s′* to nowy stan po wykonaniu akcji *a* w stanie *s*,
- *a′* to akcja, która maksymalizuje *Q-wartość* dla nowego stanu *s′*,
- *Q(s, a)* to estymowana wartość funkcji *Q* dla danego stanu *s* i akcji *a*. Oznacza to, jak dobrze wykonanie akcji *a* w stanie *s* jest oceniane przez model.
- *maxQ(s', a')* to maksymalna wartość funkcji *Q* dla wszystkich możliwych akcji w nowym stanie *s′*.



Rysunek 1.4: Schemat przedstawiający szczegółowo model sieci konwolucyjnej w algorytmie *DQN*.



Rysunek 1.5: Schemat przedstawiający szczegółowo cykl w algorytmie *Deep Q-Network*.

1.4.2. Algorytm *Dueling DQN*

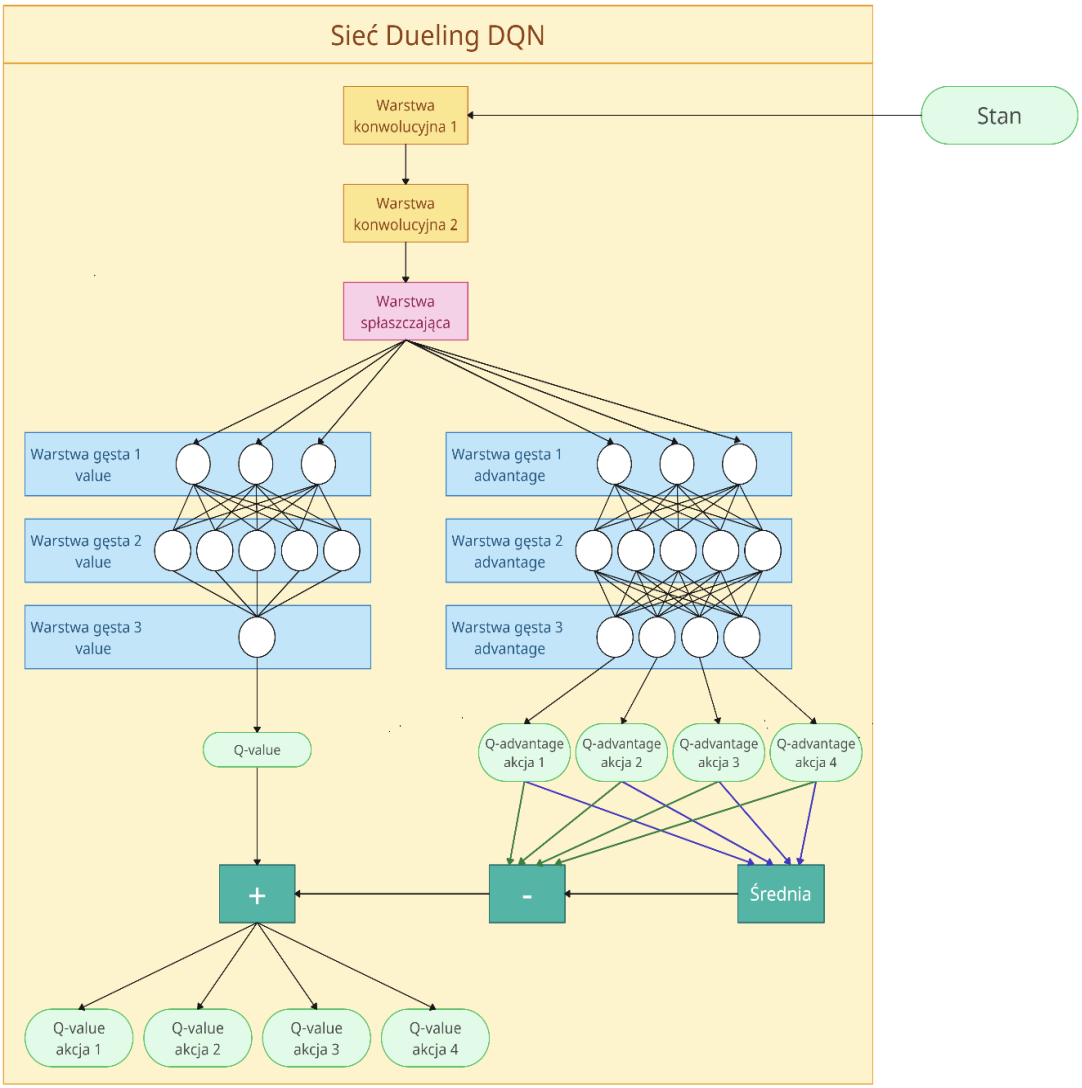
Dueling DQN [10] jest specjalną architekturą sieci neuronowej wykorzystywaną w *Deep Q-Learningu*. Koncepcja *Dueling DQN* polega na rozdzieleniu funkcji wartości *Q* na dwie składowe: wartość stanu (ang. *state value*) *V(s)* i zalety akcji (ang. *action advantages*) *A(a)*. Ta dekompozycja pozwala na efektywne uczenie się wartości stanu niezależnie od różnic między akcjami.

W tradycyjnym *DQN*, jedna sieć neuronowa jest używana do estymacji wartości *Q(s, a)*, gdzie *s* oznacza stan, a *a* oznacza akcję. W przypadku *Dueling DQN*, stosuje się inną strukturę sieci, która składa się z dwóch gałęzi, co zostało pokazane na Rysunku 1.6. Pierwsza gałąź estymuje wartość stanu *V(s)*, która reprezentuje, jak ważny jest dany stan. Druga gałąź estymuje zalety akcji *A(a)*, które reprezentują, jak korzystne są poszczególne akcje w danym stanie. Wyjścia z obu gałęzi są łączone, aby otrzymać estymację funkcji wartości *Q(s, a)* zgodnie z równaniem 1.4.

$$Q(s, a) = V(s) + A(a) - mean(A(a)) \quad (1.4)$$

Gdzie *V(s)* to wartość stanu, *A(a)* to zalety akcji dla akcji *a*, a *mean(A(a))* to średnia wartość zalety akcji dla wszystkich akcji w danym stanie. To równanie pozwala na oddzielenie informacji o wartości stanu od informacji o zaletach poszczególnych akcji.

Koncepcja *Dueling DQN* niesie ze sobą kilka korzyści. Po pierwsze, umożliwia efektywne uczenie się wartości stanu, niezależnie od różnic między akcjami. Po drugie, poprzez estymację zalety akcji, sieć jest w stanie skutecznie określić, które akcje są bardziej korzystne w danym stanie. To prowadzi do lepszej eksploatacji wiedzy o stanie i bardziej efektywnego podejmowania decyzji.



Rysunek 1.6: Schemat przedstawiający szczegółowo model sieci konwolucyjnej w algorytmie *Dueling DQN*.

1.4.3. Algorytm *Double DQN*

Double DQN [11] jest rozszerzeniem algorytmu *DQN*, które ma na celu rozwiązanie problemu przeszacowania wartości akcji, znanego jako nadmierna estymacja. Koncepcja *Double DQN* polega na wykorzystaniu dwóch oddzielnych sieci neuronowych do wyboru i oceny akcji w procesie uczenia.

W tradycyjnym *DQN*, jedna sieć neuronowa jest wykorzystywana do zarówno wyboru jak i oceny akcji w danym stanie. Jednak *Double DQN* wprowadza dodatkową sieć neuronową/konwolucyjną, nazywaną "siecią *target*". Sieć *target* jest używana do oceny wartości akcji, podczas gdy sieć wyboru jest odpowiedzialna za wybór najlepszej akcji na podstawie ocen sieci *target*. Proces uczenia w *Double DQN* przebiega w dwóch krokach. Najpierw, na podstawie stanu obliczane są wartości *Q* dla wszystkich akcji przy użyciu sieci wyboru. Następnie, wybierana jest akcja o najwyższej wartości *Q*. Jednak zamiast oceniać tę wartość *Q* przy użyciu tej samej sieci, wykorzystuje się *sieć target* do obliczenia wartości *Q* dla wybranej akcji. Ta wartość *Q* jest wykorzystywana do aktualizacji sieci wyboru w procesie uczenia za pomocą funkcji straty zdefiniowanej we wzorze 1.5. Sieć *target* nie jest aktualizowana za pomocą wartości *Q* ale co *N* epok są kopiowane wartości wag z sieci wyboru do sieci *target*. Cały proces został pokazany na Rysunku 1.7.

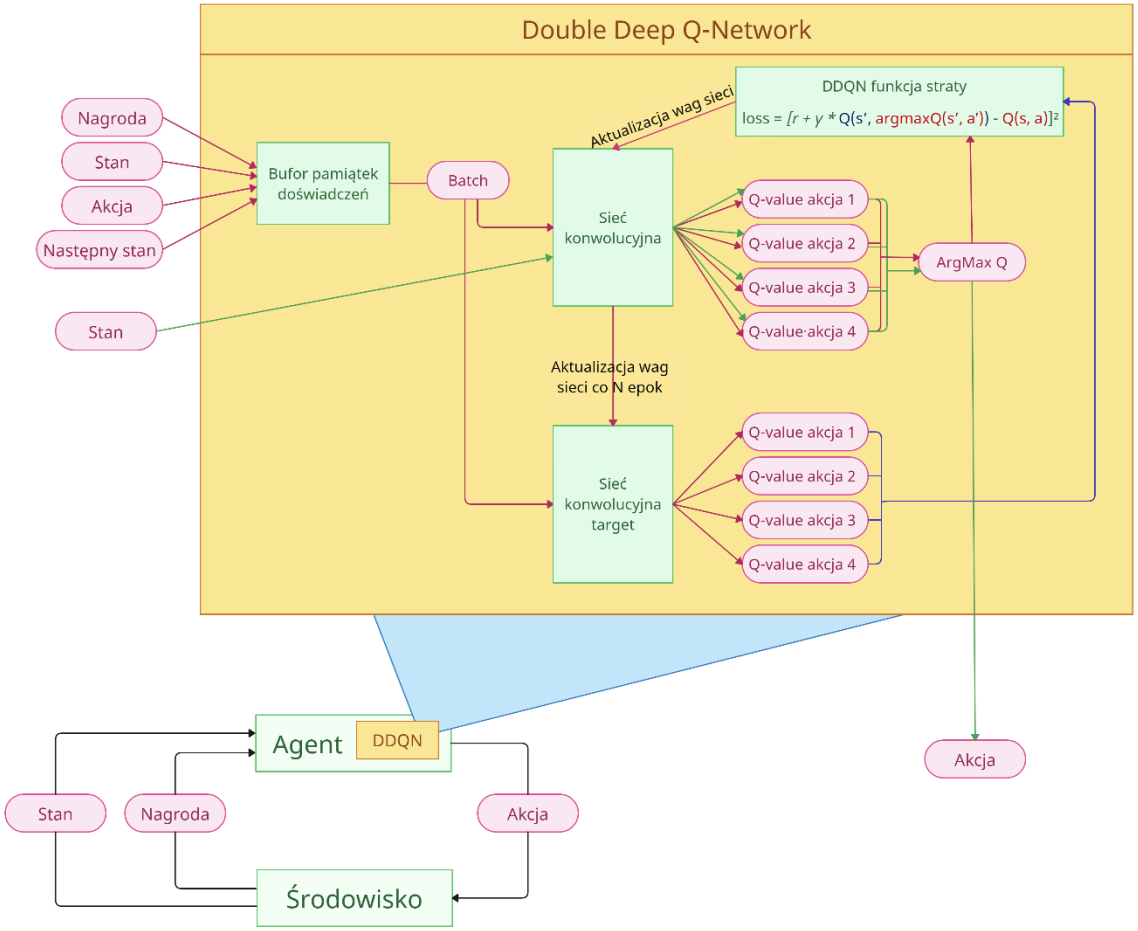
Korzyścią wynikającą z użycia *Double DQN* jest zmniejszenie przeszacowania wartości akcji. Ponieważ ocena wartości akcji jest dokonywana przez oddzielną sieć *target*, istnieje mniejsze ryzyko, że sieć wyboru przeszacuje wartość akcji. To pozwala na bardziej stabilne i precyzyjne uczenie się strategii przez agenta.

$$Funkcja\ straty = [r + y * Q(s', argmaxQ(s', a')) - Q(s, a)]^2 \quad (1.5)$$

gdzie:

- *a* to akcja,
- *s* to stan,
- *r* to nagroda za wykonanie akcji *a* w stanie *s*,
- *γ* to współczynnik dyskontowania,
- *s'* to nowy stan po wykonaniu akcji *a* w stanie *s*,
- *a'* to akcja, która maksymalizuje *Q-wartość* dla nowego stanu *s'*,

- *Q(s, a)* to estymowana wartość funkcji *Q* dla danego stanu *s* i akcji *a*. Oznacza to, jak dobrze wykonanie akcji *a* w stanie *s* jest oceniane przez model,
- *argmaxQ(s', a')* to akcja, która maksymalizuje wartość *Q* dla nowego stanu *s'*.



Rysunek 1.7: Schemat przedstawiający szczegółowo cykl w algorytmie *Double Deep Q-Network*.

1.4.4. Algorytm *Dueling Double DQN*

Dueling Double DQN [12] jest zaawansowaną techniką w *Deep Q-Learningu*, która łączy koncepcje *Dueling DQN* [10] i *Double DQN* [11]. To połączenie dwóch innowacyjnych rozwiązań ma na celu poprawę stabilności i efektywności uczenia się w przypadku problemów z dużą przestrzenią stanów i akcji. W przypadku gry Wąż, *Dueling Double DQN* może być zastosowane do efektywnego uczenia agenta. Wykorzystując dwie gałęzie sieci (*Dueling DQN)*, agent jest w stanie oddzielić wartość stanu planszy gry od zalet poszczególnych akcji. To znaczy, że jedna gałąź sieci ocenia jak dobry jest dany stan gry, niezależnie od podjętej akcji, a druga gałąź sieci, ocenia jakie akcje są korzystne w danym stanie gry. Wykorzystując dwie oddzielne sieci (*Double DQN)*, agent może dokonywać

lepszych wyborów akcji i skuteczniej uczyć się strategii, aby zdobywać jak najwięcej punktów. Jedna sieć jest używana do wyboru optymalnej akcji, podczas gdy druga sieć służy do oceny wartości tej akcji.

1.5. Dotychczasowe osiągnięcia *DQN* w grze Wąż

Wąż to gra, w której gracz kontroluje węża poruszającego się po planszy. Wąż ma za zadanie zjadać pożywienie, które pojawia się na planszy, aby zwiększać swoją długość. W miarę jak wąż rośnie, gra staje się trudniejsza, ponieważ trzeba unikać kolizji z własnym ciałem i ścianami planszy. Celem gry jest zdobycie jak największej liczby punktów poprzez zjadanie pożywienia. Przykład takiej gry przedstawiono na Rysunku 1.8.

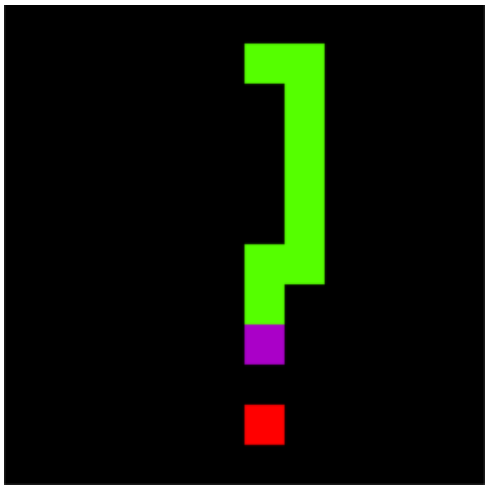
W niniejszych akapitach przedstawiam dotychczasowe osiągnięcia algorytmu *Deep Q-Network (DQN)* w grze Wąż, opierając się na dwóch artykułach naukowych [4-5]. Oba opracowania wykorzystują podobne środowisko gry węża, gdzie rozmiar planszy wynosi 240x240 pikseli, podzielonej na siatkę o wielkości 12x12 pól, co umożliwia porównanie obu rozwiązań. W obu badaniach pojedynczy stan w powtórce doświadczeń składa się z czterech kolejnych obrazów gry, co pozwala na uwzględnienie aspektu ruchu i dynamiki rozgrywki.

W pierwszym artykule [4] zastosowano algorytm *DQN* wraz z kilkoma technikami, które znacząco poprawiały osiągi procesu uczenia. Autorzy skorzystali z zaawansowanego systemu nagród, specjalnie opracowanej architektury sieci oraz podwójnej pamięci na powtórki doświadczeń. Dodatkowo, użyli techniki *ε-greedy*. Po przeprowadzeniu 140 000 epok uczenia, agent średnio osiągał około 9,04 punktów na 50 gier (na wyuczonej sieci bez włączonej dalszej nauki dla gier pomiarowych), a najlepszy wynik wynosił 17 punktów. Warto wspomnieć, że w badaniach porównawczych z tego artykułu człowiek osiągał średnio około 1,98 punkta na 50 gier w tym samym środowisku, a maksymalny wynik wynosił 15 punktów.

Drugi artykuł [5] opiera się na pierwszym opracowaniu, wprowadzając jednak pewne modyfikacje. Autorzy zaproponowali zmodyfikowany system nagród, bardziej zaawansowaną architekturę sieci oraz zastosowali znacznie mniejszą pojedynczą pamięć na powtórki doświadczeń. Ponadto, obrazy gry jako stany są optymalizowane i zapisywane w skali szarości. Warto zaznaczyć, że w drugim artykule zastosowano również *Double Deep Q-Network*, co stanowi jedną z wariacji klasycznego algorytmu *DQN*. Według wyników osiągniętych w tym badaniu, agent średnio osiągał 9,53 punktów na 50 gier (na wyuczonej

sieci bez włączonej dalszej nauki dla gier pomiarowych), a maksymalny wynik wynosił 20 punktów. Szczegółowe wyniki zostały przedstawione w Tabeli 1.1.

Oba artykuły wskazują na pozytywne rezultaty algorytmu *DQN* w grze Wąż. Mimo podobnego środowiska gry i podstawowego podejścia wykorzystującego głębokie uczenie ze wzmocnieniem, różnice w zastosowanych technikach i modyfikacjach prowadzą do nieznacznych różnic w osiągniętych wynikach. Zastosowanie zaawansowanych technik, takich jak zmodyfikowany system nagród czy bardziej zaawansowana architektura sieci, może wpływać na poprawę wydajności algorytmu *DQN*. Warto nadal eksplorować różne techniki i modyfikacje w celu dalszej poprawy osiąganych wyników oraz rozwijania autonomicznych agentów. W kolejnych rozdziałach przedstawię moją własną implementację agenta, która będzie opierać się głównie na wnioskach z powyższych dwóch artykułów naukowych [4-5]. Będę również porównywać moje własne osiągnięte wyniki z rezultatami opisanymi w tych artykułach. Wykorzystam wnioski i techniki z tych publikacji, ale dodatkowo wprowadzę także własne modyfikacje mające na celu dalszą poprawę uzyskiwanych rezultatów.



Rysunek 1.8: Wygląd gry.

Tabela 1.1 Wyniki różnych agentów w grze Wąż.

Agent	Średnia ilość punktów na 50 gier	Maksymalna ilość punktów na 50 gier
Człowiek [4]	1,98	15
DQN [4]	9,04	17
Double DQN [5]	9,53	20

Rozdział 2. Środowisko

2.1. Struktura projektu i opis pakietu *game*

Strukturę projektu zaprojektowałem w sposób modułowy, aby zapewnić przejrzystość kodu oraz łatwość rozszerzania i modyfikowania funkcjonalności gry oraz agenta. Projekt składa się z dwóch pakietów: *game* oraz *agent,* które to zawierają moduły służące do tworzenia gry oraz agenta.

W pakiecie *game* znajdują się następujące moduły:

- *agent* to moduł zawierający abstrakcyjną klasę do kontrolowania węża w grze,
- *agentRandom* to moduł zawierający reprezentację klasy *agent*. Ten agent dostarcza losowych ruchów dla węża,
- *game* to moduł zawierający klasę odpowiedzialną za mechanikę rozgrywki w grze Wąż,
- *gameInfo* to moduł zawierający klasę reprezentującą bieżący stan gry, używaną do dostarczania informacji o grze dla agenta,
- *plot* to moduł zawierający klasę serwera, który generuje wykres na stronie internetowej.

Moduł *agent* opisze w kolejnym rozdziale.

2.2. Opis gry i środowiska

Stworzyłem własne środowisko, aby móc porównywać wyniki z innymi pracami [4-5]. Rozmiar gry ustawiłem na 240x240 pikseli i plansze na 12x12 pól, co odpowiada specyfikacji opisanej w artykułach [4-5]. Dzięki temu rozmiarowi i podziałowi planszy, moje wyniki będą porównywalne z wcześniejszymi badaniami. W mojej implementacji wąż zawsze rozpoczyna grę o długości 3 pól i skierowany jest w prawo. Głowa węża ma kolor fioletowy, ciało jest zielone, a pożywienie jest czerwone. Tło gry jest czarne. Wygląd środowiska gry przedstawiony jest na Rysunku 2.1. Środowisko umożliwia między innymi łatwą personalizację rozmiaru gry, kolorów komponentów, prędkości węża oraz wyświetlania wyników w różnych konfiguracjach. Dodatkowo, środowisko posiada wbudowany serwer do generowania wykresów w czasie rzeczywisty, który jest dostępny poprzez stronę internetową.

Tworzenie własnego agenta do gry, polega na napisaniu własnej klasy, która będzie dziedziczyć po klasie abstrakcyjnej *Agent* znajdującej się w pakiecie *game*. W nowo tworzonej klasie należy zaimplementować funkcje *getNewDirection(gameInfo)*, która jest

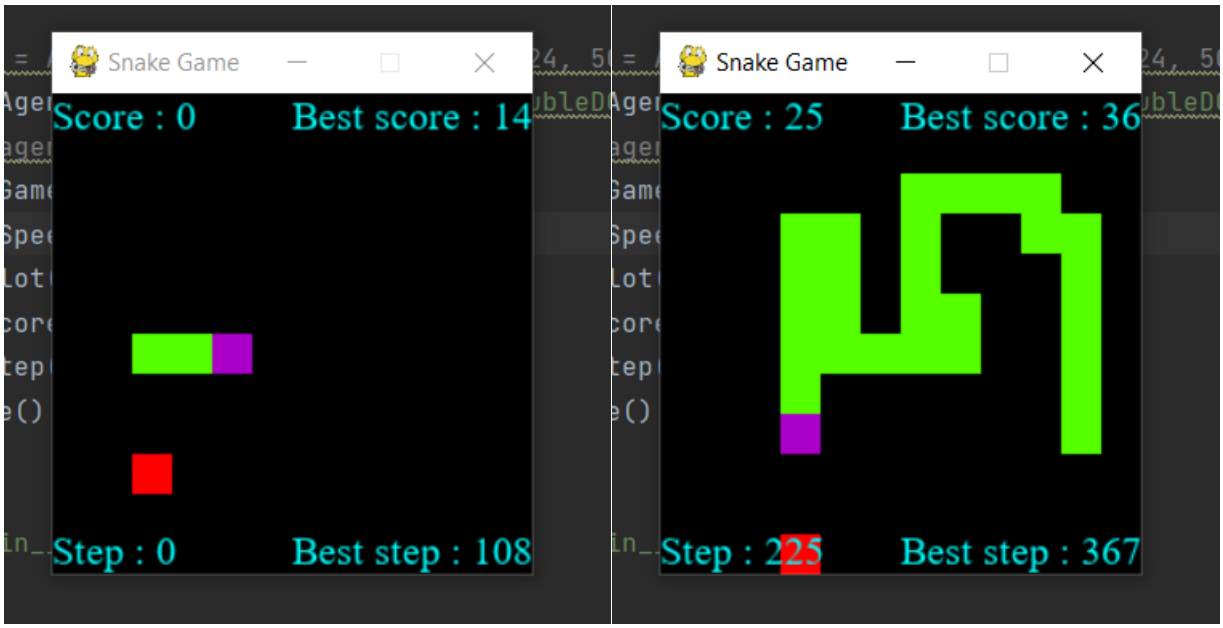
wywoływana przed każdym ruchem węża przez środowisko. Funkcja ta posiada argument *gameInfo*, który jest obiektem dostarczającym agentowi aktualnych danych o wężu i środowisku, takich jak: obraz z gry (z wynikami na ekranie i bez), ilość zdobytych punktów w obecnej grze, najwyższy wynik punktów ze wszystkich dotychczasowych gier, ilość wykonanych kroków w obecnej grze, najwyższa ilość wykonanych kroków we wszystkich dotychczasowych grach, numer aktualnie rozgrywanej gry, ostatnio użyty kierunek poruszania się węża, suma wykonanych kroków we wszystkich dotychczasowych grach, pozycja węża na planszy, pozycja pożywienia na planszy oraz pozycja każdego segmentu ciała węża na planszy. Funkcja powinna zwracać kierunek, w którym wąż powinien skręcić, reprezentowanego przez liczbę całkowitą:

0 – w górę,

1 – w prawo,

2 – w dół,

3 – w lewo.



Rysunek 2.1: Wygląd środowiska.

2.3. Tworzenia środowiska oraz agenta

Aby stworzyć środowisko gry oraz użyć w nim agenta dostarczonego wraz z pakietem *game,* który steruje wężem wykonując losowe ruchy, należy najpierw zaimportować moduły *Game* oraz *AgentRandom*. Następnie tworzyć nową instancję klasy *AgentRandom*. Potem tworzyć nową instancję klasy *Game,* która potrzebuje 3 następujących argumentów: obiekt agenta, który będzie sterował wężem w grze, ilość gier do rozegrania oraz flagę określającą, czy gra ma działać w tle czy ma być wyświetlana. W następnym kroku można skonfigurować środowisko o dodatkowe parametry, np. ustawić prędkość węża (maksymalna ilość klatek na sekundę), uruchomić serwer do tworzenia i wyświetlania wykresu punktów i kroków na stronie internetowej, wyświetlanie punktów i kroków na ekranie gry. Na koniec należy uruchomić skonfigurowane środowisko za pomocą funkcji *startGame()*. Gra zostanie uruchomiona z domyślnym rozmiarem planszy 240x240 pikseli, rozmiarem pojedynczego pola na planszy 20x20 pikseli, co daje planszę o rozmiarze 12x12 pól. Zostaną również zastosowane domyślne kolory (czarne tło, fioletowa głowa, zielone ciało, czerwone pożywienie). Tworzenie i uruchamianie środowiska gry zostało zaprezentowane na Listingu 2.1.

Listing 2.1. Tworzenie gry wraz z agentem.

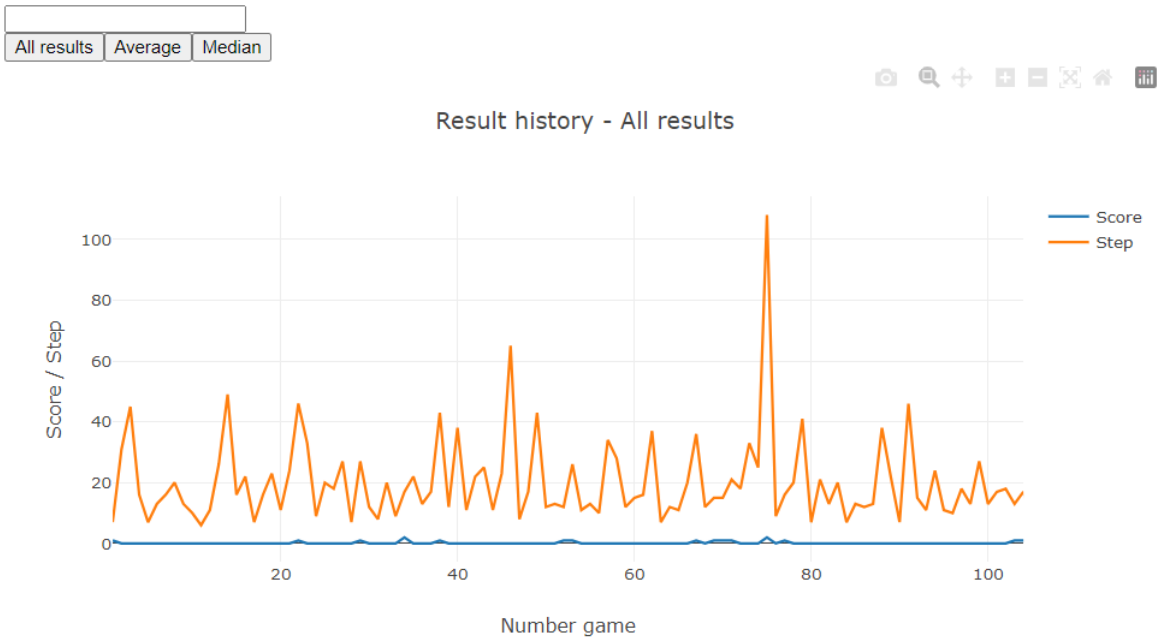
```
1   from game.game import Game
2   from game.agentRandom import AgentRandom
3
4   def main():
5           newAgent = AgentRandom()
6           newGame = Game(newAgent, 100, True)
7           newGame.setSnakeSpeed(1)
8           newGame.setShowPlot(True)
9           newGame.setShowScore(True)
10          newGame.setShowStep(True)
11          newGame.startGame()
12
13  if __name__ == '__main__':
14          main()
```

2.4. Serwer do tworzenia wykresów wbudowany w środowisko gry.

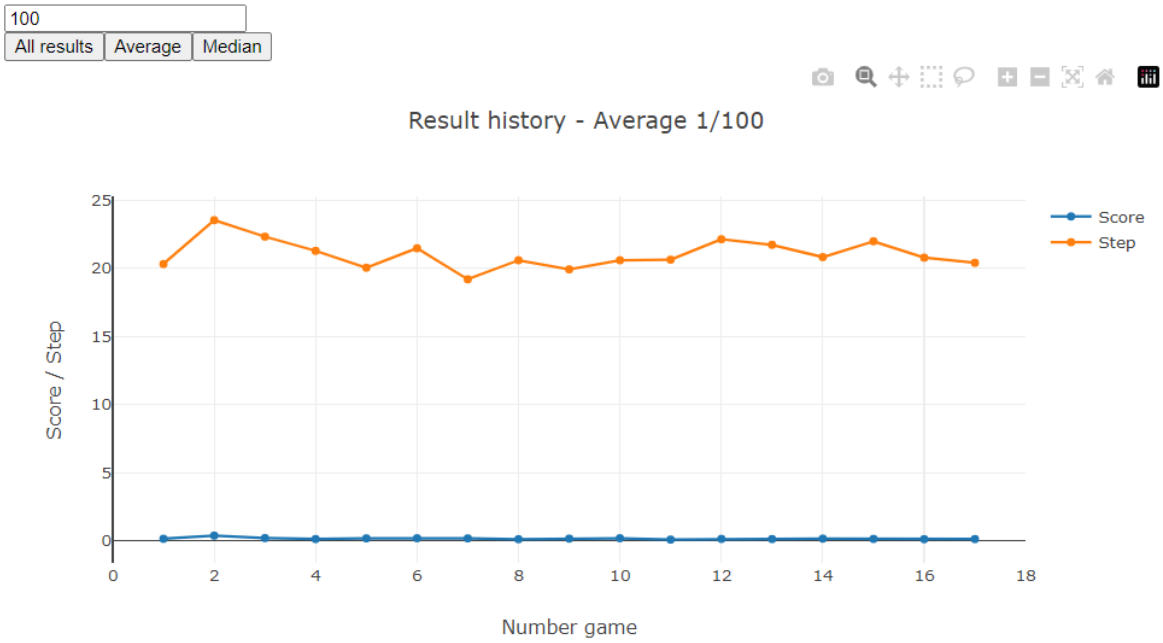
Aby umożliwić łatwą analizę wyników w środowisku gry, opracowałem serwer umożliwiający generowanie dynamicznych wykresów na podstawie rozegranych rozgrywek w czasie rzeczywistym. Serwer został zaimplementowany przy użyciu biblioteki *Dash* [13], która zapewnia prostą i efektywną metodę tworzenia serwera oraz interaktywnej strony, dostępnej poprzez ten serwer. Opracowana przeze mnie strona internetowa oferuje szereg przydatnych opcji, takich jak:

- wyświetlenie ilości zdobytych,
- wyświetlenie wykonanych kroków,
- wyświetlenie ilości zdobytych punktów i wykonanych kroków na jednym wykresie,
- agregacja danych (średnia oraz mediana dla dowolnego przedziału danych),
- powiększanie i pomniejszanie wykresu.

Na Rysunku 2.2 przedstawiłem przykład wyświetlania ilości zdobytych punktów i wykonanych kroków na jednym wykresie. Natomiast na Rysunku 2.3 przedstawiona jest agregacja danych, przedstawiająca średnią z każdych 100 rozegranych gier.



Rysunek 2.2: Wykres punktów i kroków we wszystkich grach.



Rysunek 2.3: Wykres punktów i kroków średnio z każdych 100 gier.

Rozdział 3. Implementacja *DQN*

3.1. Opis pakietu *agent*

Drugim pakietem w moim projekcie jest *agent*, który umożliwia użycie jednego z czterech zaimplementowanych algorytmów: *DQN*, *Dueling DQN*, *Double DQN*, *Dueling Double DQN*. Dzięki modułowości w łatwy sposób można modyfikować obecnych agentów oraz tworzyć nowych przy wykorzystaniu gotowych modułów, takich jak moduł odpowiadający za zarządzanie powtórkami doświadczeń.

W pakiecie *agent* znajdują się następujące moduły:

- *agentDQN* to moduł zawierający klasę do kontrolowania węża w grze, która uczy się na podstawie wybranego algorytmu (DQN, Double DQN, Dueling DQN, Dueling Double DQN),
- *agentLoadModel* to moduł zawierający klasę do kontrolowania węża w grze na podstawie wcześniej nauczonego modelu, wykorzystując klasę *AgentDQN*,
- *cnnDDQN* to moduł zawierający klasę definiującą model sieci *CNN* dla algorytmu *Dueling DQN*,
- *cnnDQN* to moduł zawierający klasę definiującą model sieci *CNN* dla algorytmu *DQN*,
- *ddqn* to moduł zawierający klasę, która zawiera funkcje szkolenia modelu sieci *CNN* dla algorytmu *Double DQN*,
- *dqn* to moduł zawierający klasę, która zawiera funkcje szkolenia modelu sieci *CNN* dla algorytmu *DQN*,
- *memory* to moduł zawierający klasę, która służy do przechowywania powtórek doświadczeń.

3.2. Główne założenia implementacji agenta

Moja implementacja agenta zakłada, że rozmiar gry wynosi 240x240 pikseli, a pojedyncze pole ma rozmiar 20x20 pikseli, co daje planszę o rozmiarze 12x12 pól. W tego typach grach, gdzie agent sterowanie opiera tylko na obrazie gry, występuje problem określenia kierunku, w którym porusza się postać, w tym przypadku wąż. Prace [4-5] rozwiązują ten problem, tworząc stan gry z sekwencji 4 kolejnych klatek gry. Moja implementacja w przeciwieństwie do tych prac wykorzystuje pojedynczą klatkę gry jako stan gry, lecz koloruję głowę węża na inny kolor niż reszta ciała. Dzięki temu rozwiązaniu

zaoszczędzam bardzo dużo pamięci, ponieważ pojedynczy stan zajmuje znacznie mniej miejsca, a mimo to mój agent nadal potrafi określić kierunek, w którym porusza się wąż.

Moja agent pobiera od środowiska następujące informacje: obraz gry bez wyświetlanych informacji o punktach i krokach na ekranie, numer rozgrywanej gry, ilość zdobytych punktów w obecnej rozgrywanej grze oraz ilość wykonanych kroków przez węża we wszystkich rozegranych grach. Pobrany obraz gry jest przetwarzany w celu optymalizacji pamięci oraz mocy obliczeniowej. Następnie przetworzona klatka gry jest transformowana na pamiątkę doświadczenia, która się składa z poprzedniego stanu gry, akcji, nagrody oraz obecnego stanu gry. W tym momencie również zostaje obliczona nagroda potrzebna do zbudowania pamiątki doświadczenia. Gotowa pamiątka doświadczenia jest zapisywana do bufora. Następnie pobierany jest losowy *batch* z bufora i na podstawie tego *batcha* następuje trening agenta. Po przeprowadzeniu treningu, agent na podstawie obecnego stanu gry określa kierunek, w którym wąż powinien się poruszać, i zwraca tę wartość do środowiska.

Dzięki modułowości, mój agent może łatwo wykorzystać jeden z dostępnych w tym pakiecie algorytmów: *DQN*, *Dueling DQN*, *Double DQN* lub *Dueling Double DQN*. Wystarczy wybrać odpowiednią funkcje szkolenia (zmienna *trainingFunction*) oraz model sieci (zmienna *model*) w konstruktorze agenta. Możliwe ustawienia zmiennych *model* i *trainingFunction*:

- *model = CNNDQN (DQN*) lub *CNNDDQN* (*Dueling DQN*)
- *trainingFunction = DQN* (*DQN*) lub *DDQN* (*Double DQN*)

Na przykład, jeśli chcemy użyć algorytmu *Dueling Double DQN* w naszym agencie, musimy ustawić zmienną *model* na instancje klasy *CNNDDQN,* a zmienną *trainingFunction* na instancje klasy *DDQN,* podając następujące argumenty:

- *model* to obiekt reprezentujący model sieci neuronowej,
- *learningRate* to współczynnik uczenia używany przez optymalizator,
- *gamma* to współczynnik dyskontowania używany do obliczania przyszłych nagród,
- *clipByNorm to* wartość, do której zostanie przeskalowana wielkość gradientu. Ustawienie go na „None” powoduje wyłączenie ograniczania wielkości gradientu,
- *tau* to parametr interpolacji używany do aktualizacji sieci docelowej (tylko w przypadku korzystania z *Double DQN*).

Przykładowe ustawienie algorytmu *Dueling Double DQN* zostało zaprezentowane na Listingu 3.1.

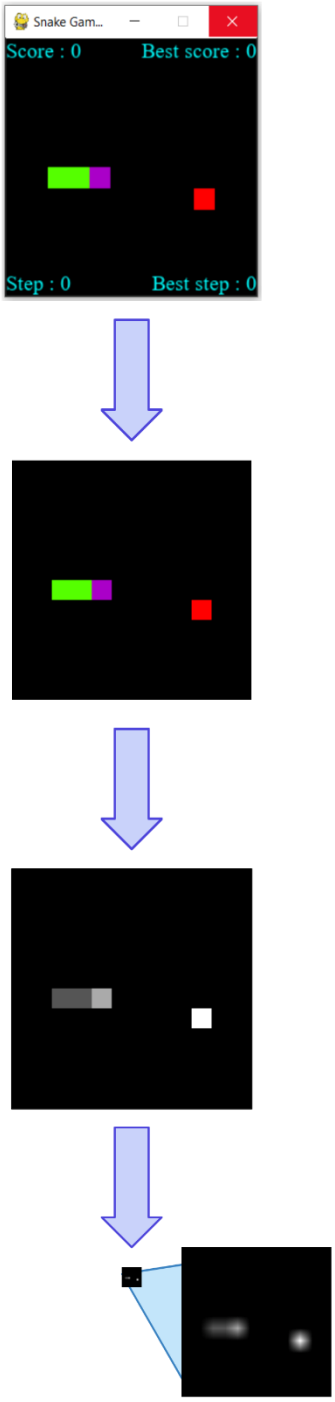
Listing 3.1. Ustawienie w agencie algorytmu *Dueling Double DQN*.

```
1  model = CNNDDQN()
2  trainingFunction = DDQN(model, learningRate, gamma, clipByNorm, tau)
```

3.3. Poszczególne zaimplementowane elementy w agencie

3.3.1. Wstępne przetwarzanie danych

Z uwagi na ograniczoną pamięć oraz moc obliczeniową potrzebną do treningu modelu sieci, przed przekazaniem obrazu z gry do agenta poddaję go wstępnemu przetworzeniu. Na początku agent otrzymuje bieżący stan gry (klatkę) ze środowiska, który nie zawiera żadnych informacji o zdobytych punktach czy wykonanych krokach. Następnie wyodrębniam tylko jeden kanał z obrazu *RGB* planszy gry i przekształcam go na skalę szarości. Kolory elementów w grze są tak dobrane, aby łatwo można było przedstawić zdjęcie planszy gry w skali szarości, jednocześnie zachowując czytelność. W kolejnym kroku zmniejszam rozmiar obrazu z 240x240 pikseli do 12x12 pikseli przy użyciu parametru *sizeResize*. Dzięki temu jeden piksel odpowiada jednemu polu na planszy gry. Proces przetwarzania obrazu jest również pokazany na Rysunku 3.1. Dzięki takiemu przetworzeniu oszczędzam znaczną ilość pamięci, a także skracamy czas przetwarzania przez sieć konwolucyjną. Dodatkowo, zastosowanie kolorowania głowy węża innym kolorem pozwala na reprezentację stanu gry jedynie za pomocą pojedynczej klatki, w przeciwieństwie do podejścia opisanego w pracach [4-5], co również zmniejsza zużycie pamięci. Na końcu, obraz jest normalizowany, aby wartości pikseli mieściły się w zakresie od 0 do 1, zamiast standardowego zakresu od 0 do 255. Robi się to, ponieważ sieć konwolucyjna korzysta lepiej z danych w zakresie od 0 do 1, co wpływa na efektywność treningu. Przetworzenie obrazu w ten sposób umożliwia oszczędne zarządzanie pamięcią (redukcja wagi pojedynczego stanu o 99,3%, co zostało przedstawione w Tabeli 3.1) oraz efektywne wykorzystanie zasobów obliczeniowych sieci neuronowej.



Rysunek 3.1: Proces wstępnego przetwarzania obrazu w agencie.

Tabela 3.1 Waga pojedynczego obrazu.

Obraz przed przetworzeniem	168,75kb
Obraz po przetworzeniu	1,25kb
Zaoszczędzona pamięć	99,3%

3.3.2. System nagród

System nagród w moim agencie jest bardzo podobny do tego opisanego w pracy naukowej [5], z niewielką różnicą. Podobnie jak w tej pracy, posiadam trzy rodzaje nagród. Za zdobycie punktu (zjedzenie owocu) agent otrzymuje 1 punkt, za przegranie (zderzenie się ze ścianą lub z samym sobą) otrzymuje -1 punkt, a za każdy krok, w którym nie zdobył punktu ani nie przegrał, otrzymuje -0.05 punktu, co zostało również przedstawione w Tabeli 3.2. Jedyna różnica między moim systemem nagród a tym opisanym w pracy [5] dotyczy ostatniej nagrody, która w pracy naukowej wynosiła -0.1 punktu. Z moich obserwacji wynika, że zmniejszenie tej kary pozytywnie wpływa na efektywność nauki agenta. Dzięki temu prostemu i czytelnemu systemowi nagród agent szybko uczy się odpowiednich strategii. Bardzo małe ujemne punkty za ruchy bez zdobycia punktu zachęcają agenta do zdobywania kolejnych punktów, nie zapominając o unikaniu przegranej. Zakres nagród od 1 do -1 również przyczynia się korzystnie do wyników nauki agenta.

Tabela 3.2 System nagród agenta.

1pkt	Zdobycie punktu (zjedzenie owocu)
-1pkt	Przegranie (zderzenie się ze ścianą lub z samym sobą)
-0.05pkt	Każdy inny krok (brak przegranej i zdobycia punktu)

3.3.3. Model sieci konwolucyjnej

W moim agencie zastosowałem swoją propozycję sieci konwolucyjnej, która przyjmuje obrazek o rozmiarze 12x12 pikseli w skali szarości. Sieć składa się z trzech warstw konwolucyjnych. Pierwsza z nich używa filtru o rozmiarze 5x5, a kolejne dwie mają filtry o rozmiarze 3x3. Wszystkie warstwy konwolucyjne mają krok o rozmiarze 1. Po drugiej warstwie konwolucyjnej występuje warstwa *Max Pooling*, która ma filtr o rozmiarze 2x2 i krok o rozmiarze 2. Po trzeciej warstwie konwolucyjnej następuje warstwa spłaszczająca, a następnie mamy cztery warstwy gęste (ang. *Fully Connected*). Pierwsze trzy warstwy gęste mają 128 neuronów, a ostatnia warstwa gęsta jest warstwą wyjściową o czterech neuronach. Wszystkie warstwy gęste i konwolucyjne wykorzystują funkcję aktywacji *ReLU*. Model sieci również został zaprezentowany w Tabeli 3.3.

Podczas uczenia sieci zastosowałem algorytm optymalizacji Adam, który efektywnie dostosowuje tempo uczenia w trakcie procesu optymalizacji. To przyczynia się do lepszej

zbieżności modelu i efektywniejszego uczenia. Parametr *learningRate* pozwala ustawić wartość współczynnika uczenia dla tego optymalizatora.

Tabela 3.3 Model sieci konwolucyjnej.

Warstwa	Filtr	Krok	Aktywacja	Rozmiar wyjścia
Wejście				12x12x1
Konwolucyjna 1	5x5	1	ReLU	8x8x16
Konwolucyjna 2	3x3	1	ReLU	6x6x32
Max Pooling	2x2	2		3x3x32
Konwolucyjna 3	3x3	1	ReLU	1x1x64
Spłaszczająca				64
Gęsta 1			ReLU	128
Gęsta 2			ReLU	128
Gęsta 3			ReLU	128
Wyjście				4 (liczba akcji)

3.3.4. Bufor pamiątek doświadczeń

Bufor pamiątek doświadczeń jest strukturą danych przechowującą doświadczenia agenta w postaci krotek (stan, akcja, nagroda, następny stan). Jako że nie znam następnego stanu będąc w obecnym stanie, rozwiązałem ten problem, uznając, że obecny stan w grze to poprzedni stan, a obecny stan to następny stan w pamięci doświadczeń. Bufor powtórek w mojej implementacji został zrealizowany za pomocą kolejki *deque* [14] z biblioteki *collections* w języku Python. Bufor ma określoną pojemność (parametr *memorySize*), która determinuje maksymalną ilość doświadczeń, jakie można w nim przechowywać. Gdy bufor osiąga maksymalną pojemność, najstarsze doświadczenia są usuwane, aby zrobić miejsce dla nowych.

Bufor powtórek odgrywa kluczową rolę w algorytmie *DQN*. Gromadzenie i ponowne wykorzystywanie doświadczeń agenta ma na celu złagodzenie problemu korelacji między kolejnymi obserwacjami, które mogą wystąpić podczas uczenia. Dzięki temu agent może lepiej generalizować zdobyte doświadczenia i efektywniej uczyć się strategii.

W algorytmie *DQN* istnieje konieczność rozróżniania powtórek, które zakończyły się przegraną (ang. *terminal states*), od pozostałych doświadczeń. W tym celu korzystamy z

nagrody agenta. Gdy agent osiąga stan końcowy, nagroda przyjmuje wartość -1, co informuje o zakończeniu danego epizodu.

3.3.5. Wypełnianie bufora pamiątkami doświadczeń

Początkowa faza nauki agenta polega na tym, że agent wykonuje losowe ruchy w środowisku nie trenując sieci aby zbierać odpowiednią ilość pamiątek doświadczeń. Wypełnianie bufora poprzez losowe ruchy agenta bez trenowania sieci ma trzy główne zalety. Po pierwsze, pozwala na zebranie zróżnicowanego zestawu doświadczeń, co przyspiesza proces uczenia i umożliwia agentowi lepsze generalizowanie zdobytych informacji o środowisku. Po drugie, losowe ruchy pozwalają na eksplorację różnych obszarów i unikanie pułapek lokalnych, co umożliwia znalezienie bardziej optymalnych strategii. Po trzecie, wypełnienie bufora losowymi ruchami przygotowuje go do efektywnego trenowania sieci, zapewniając bazę danych, na której agent może się oprzeć.

W mojej implementacji wartością parametru *stepWithoutLearn* wskazujemy, ile losowych ruchów agent ma wykonać przed rozpoczęciem procesu trenowania sieci. Określa to, ile doświadczeń zostanie zebranych przez agenta przed rozpoczęciem efektywnego uczenia.

3.3.6. Technika *Epsilon-Greedy*

Technika *Epsilon-Greedy* jest popularną strategią eksploracji-wykorzystania stosowaną w uczeniu ze wzmocnieniem. W przypadku algorytmów *DQN*, *Epsilon-Greedy* jest używane do podejmowania decyzji dotyczących eksploracji (wybierania losowych akcji) lub wykorzystania (wybierania najlepszej znanej akcji) w zależności od wartości parametru epsilon. Główną korzyścią z zastosowania tej techniki jest osiągnięcie równowagi pomiędzy eksploracją a wykorzystaniem w procesie uczenia ze wzmocnieniem. Na początku treningu agent potrzebuje eksplorować środowisko, aby odkryć nowe akcje i zdobyć jak najwięcej doświadczeń. W miarę postępu treningu, agent nabiera większej pewności swoich umiejętności i zaczyna wykorzystywać zdobytą wiedzę. Stopniowe zmniejszanie wartości epsilon w czasie pozwala stopniowo skupiać agenta na wykorzystywaniu zdobytej wiedzy.

W mojej implementacji *Epsilon-Greedy* sterowanie odbywa się za pomocą trzech wartości: startEpsilon (wartość, od której wartość epsilon zaczyna maleć), stopEpsilon (wartość, do której wartość epsilon ma spaść), reductionEpsilon (wartość, o jaką wartość epsilon jest zmniejszana w każdym kroku gry). Praktycznie oznacza to, że gdy te parametry

są ustawione na wartości kolejno: 1.0, 0.0, 0.1, agent będzie zawsze wybierał losową akcję z prawdopodobieństwem 100% na początku treningu, a po dziesięciu krokach gra będzie polegała w 100% na wykorzystaniu nauczonej strategii.

3.3.7. Przycinanie gradientów w sieci

Przycinanie gradientów (ang. *gradient clipping*) w sieci neuronowej jest techniką regularyzacji, która polega na ograniczaniu wartości gradientów podczas procesu wstecznej propagacji. Ma to na celu zapobieganie "wybuchającemu gradientowi", czyli sytuacji, gdy wartości gradientów są bardzo duże i mogą prowadzić do niestabilności lub trudności w procesie uczenia. W przypadku algorytmu *DQN*, przycinanie gradientów ma na celu kontrolowanie skali gradientów podczas aktualizacji wag sieci neuronowej. W procesie uczenia *DQN*, gradienty są obliczane na podstawie błędu średniokwadratowego (*MSE*) pomiędzy przewidywanymi *Q-wartościami*, a docelowymi *Q-wartościami*. Przycinanie gradientów można zastosować do tych gradientów, aby ograniczyć ich wartości do ustalonego zakresu.

Przycinania gradientów, takie jak *clip by value* czy *clip by norm*, mogą być stosowane w *DQN* w zależności od potrzeb i charakterystyki problemu. *Clip by value* polega na ograniczaniu wartości gradientów do określonego zakresu wartości. Natomiast *clip by norm* polega na skalowaniu gradientów, aby zachować ich normę w określonym zakresie.

W mojej implementacji używam przycinania gradientów typu *clip by norm* w zakresie od 0.0 do 1.0. Oznacza to, że norma (długość) gradientów będzie skalowana w taki sposób, aby pozostała w zakresie od 0.0 do 1.0. Jeśli norma gradientu przekracza ten zakres, zostanie ona przeskalowana tak, aby spełnić to ograniczenie.

3.3.8. Problem z pamiątkami doświadczeń po zdobyciu punktu

W algorytmie *DQN* wyróżniamy dwa rodzaje powtórek doświadczeń: te, które kończą się przegraną (ang. *terminal states*) i te, które nie kończą się przegraną. W przypadku powtórek kończących się przegraną, *Q-wartość* oczekiwana jest równa otrzymanej nagrodzie. Natomiast dla pozostałych powtórek, *Q-wartość* oczekiwana jest obliczana z użyciem wzoru Bellmana, zaprezentowanym w poprzednim rozdziale jako wzór 1.2. Jednak podczas gry w Węża i analizy obrazu z gry pojawia się problem z powtórkami doświadczeń, które są tworzone po zdobyciu punktu przez agenta. W takiej sytuacji zmienia się położenie nowego owocu na mapie, co powoduje niekompatybilność między obecnym a następnym

stanem. W mojej implementacji traktuję takie powtórki doświadczeń tak samo jak powtórki z przegraną, czyli *Q-wartość* oczekiwana jest równa nagrodzie za dany stan. Zarówno powtórki z przegraną, jak i powtórki po zdobyciu punktu, są rozróżniane na podstawie nagrody, która wynosi -1.0 dla przegranych i 1.0 dla zdobycia punktu. Ta technika dobrze działa i umożliwia agentowi efektywne uczenie się strategii w grze Wąż.

3.3.9. *Dueling DQN*, *Double DQN* oraz *Soft Update*

W poprzednich podpunktach omówiłem elementy zaimplementowanego agenta opartego na algorytmie *DQN*. Teraz przedstawię implementację rozszerzenia tego algorytmu, takie jak *Dueling DQN* i *Double DQN*, które zostały omówione w pierwszym rozdziale.

Algorytm *Dueling DQN* różni się od klasycznego *DQN* poprzez zastosowanie zmodyfikowanego modelu sieci, w którym warstwy gęste są podzielone na dwie gałęzie, jak przedstawiono w Tabeli 3.4. W mojej implementacji zastosowałem takie same warstwy gęste w obu gałęziach, z wyjątkiem ostatniej warstwy wyjściowej, gdzie jedna gałąź ma jeden neuron, a druga gałąź cztery neurony. Aby użyć tego algorytmu, w konstruktorze agenta należy przypisać instancję klasy *CNNDDQN* do zmiennej *model*.

Algorytm *Double DQN* różni się od klasycznego *DQN* poprzez zastosowanie zmodyfikowanej funkcji uczenia, w której wykorzystuje się dwie sieci konwolucyjne. Oba modele mają identyczną strukturę i początkowo mają takie same wagi. Jedna sieć jest aktualizowana na bieżąco, podczas gdy druga jest okresowo aktualizowana przez kopiowanie wag z pierwszej sieci. Aby użyć tego algorytmu, w konstruktorze agenta należy przypisać instancję klasy *DDQN* do zmiennej *trainingFunction*.

Moja implementacja *Double DQN* dodatkowo wykorzystuje technikę *Soft Update*, która polega na tym, że zamiast okresowego kopiowania wag sieci, losowo wybierane wagi pierwszej sieci są stopniowo kopiowane do drugiej sieci w każdym kroku. Kontrolę nad tym, jak duża część wag jest kopiowana, zapewnia parametr *tau*. Przykładowo, ustawienie wartości tej zmiennej na 0.1 oznacza, że co krok 10% losowo wybranych wag jest kopiowanych.

Tabela 3.4 Model sieci konwolucyjnej dla algorytmu *Dueling DQN*.

Warstwa	Filtr	Krok	Aktywacja	Rozmiar wyjścia
Wejście				12x12x1
Konwolucyjna 1	5x5	1	ReLU	8x8x16
Konwolucyjna 2	3x3	1	ReLU	6x6x32
Max Pooling	2x2	2		3x3x32
Konwolucyjna 3	3x3	1	ReLU	1x1x64
Spłaszczająca				64
Gęsta 1 – gałąź 1			ReLU	128
Gęsta 2 – gałąź 1			ReLU	128
Gęsta 3 – gałąź 1			ReLU	128
Wyjście – gałąź 1				4
Gęsta 1 – gałąź 2			ReLU	128
Gęsta 2 – gałąź 2			ReLU	128
Gęsta 2 – gałąź 2			ReLU	128
Wyjście – gałąź 2			ReLU	1

Rozdział 4. Porównanie wyników oraz ich analiza

W celu przetestowania mojej implementacji agenta używając różnych wariantów algorytmu *DQN*, zastosowałem hiperparametry podane w Tabeli 4.1. Przez wielokrotne eksperymenty i metodę prób i błędów znalazłem optymalne ustawienia tych parametrów.

Najpierw przyjrzyjmy się wynikom osiągniętym podczas treningu agenta, który wykorzystuje różne warianty algorytmu *DQN*. Na wykresie przedstawionym na Rysunku 4.1 można zauważyć, że w początkowej fazie treningu wyniki są bliskie zeru i przez pewien krótki czas nie rosną. Jest to spowodowane techniką wypełniania bufora pamiątkami doświadczenia, w której agent wykonuje wszystkie ruchy losowo i nie trenuje sieci. Następnie przechodzimy do fazy z zastosowaniem techniki *epsilon-greedy*, w której agent rozpoczyna trenowanie sieci. Na początku tej fazy pierwsze ruchy są wykonywane losowo, ale z czasem coraz więcej ruchów jest wykonywanych zgodnie z wyuczoną strategią agenta. Już w tej fazie można zaobserwować znaczący wzrost zdobywanych punktów przez agenta. Widoczna jest również przewaga zaawansowanych wariantów algorytmu *DQN* nad jego klasyczną wersją, ponieważ rozszerzone algorytmy szybciej uczą się odpowiedniej strategii. Następnie następuje klasyczna faza treningu, w której funkcja zdobywanych punktów powoli się wypłaszcza dla każdego z algorytmów. Algorytm *DQN* oraz *Dueling DQN* osiągają najwyższy wynik zdobywanych punktów (oba algorytmy średnio około 16,5 punktu) po około 125 tysiącach rozegranych gier, a następnie następuje delikatne przeuczenie modelu i wyniki spadają powoli wraz z dalszym treningiem. Można zauważyć, że algorytm *Dueling DQN* ma przewagę nad zwykłym *DQN*, ponieważ szybciej uczy się optymalnej strategii, ale ostatecznie oba algorytmy osiągają podobne wyniki. W przypadku algorytmów *Double DQN* oraz *Dueling Double DQN* szybkość nauki w środkowej fazie treningu jest porównywalna do *Dueling DQN*, ale są w stanie nauczyć się ostatecznie lepszej strategii, zdobywając więcej punktów (*Double DQN* - średnio około 22,5 punktu, *Dueling Double DQN* - średnio około 20,5 punktu). Podczas ostatniej fazy funkcja zdobywanych punktów wypłaszcza się znacznie wolniej niż w przypadku *DQN* i *Dueling DQN*. Ponadto w ostatniej fazie algorytm *Double DQN* uczy się szybciej niż *Dueling Double DQN*, osiągając lepszą strategię.

Teraz przeanalizujemy wyniki osiągnięte przez mojego agenta w porównaniu z agentami z innych prac naukowych [4-5]. Pomiar został przeprowadzony poprzez rozegranie 50 gier z wyuczoną wcześniej wersją agenta na podstawie 140 tysięcy rozegranych gier.

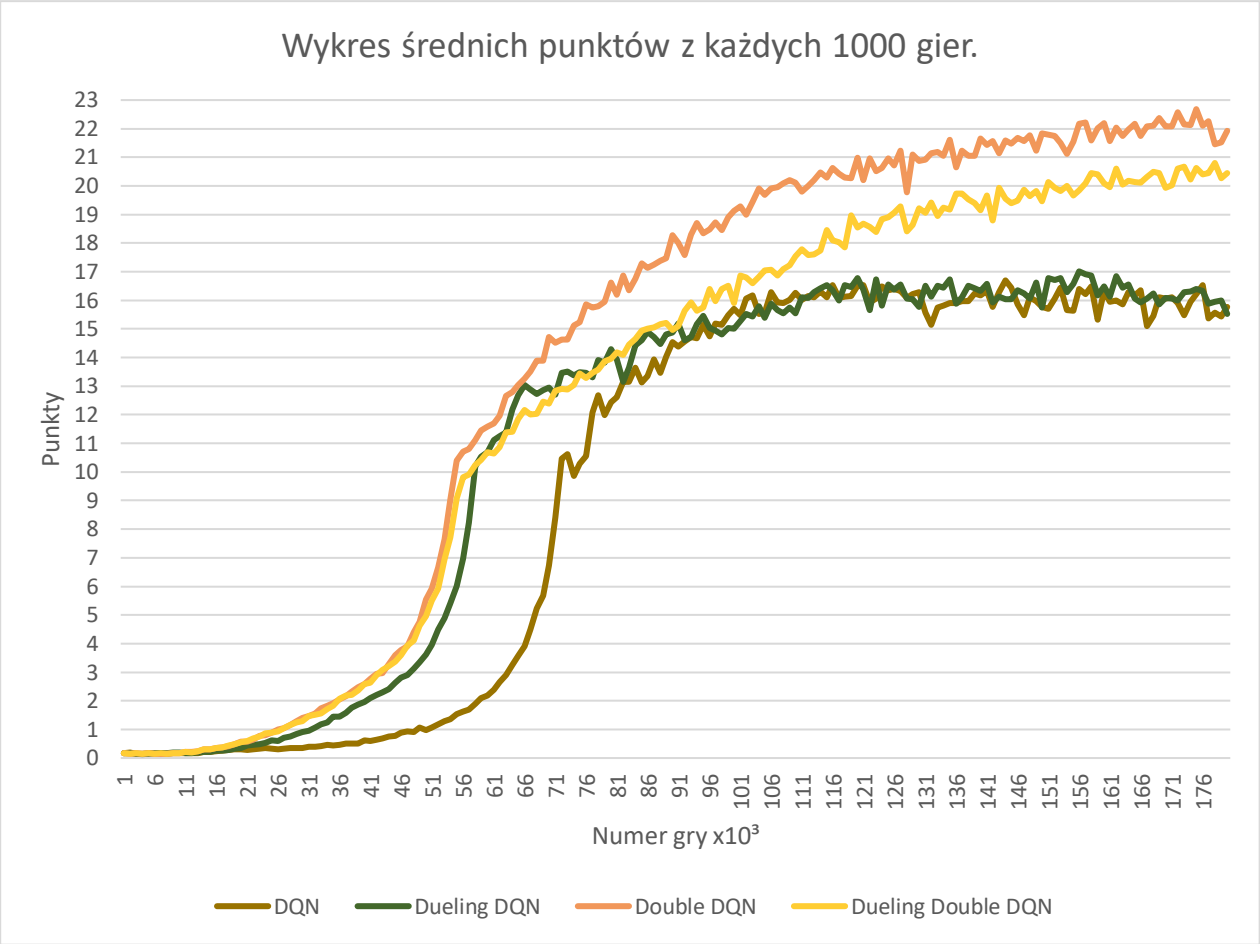
Następnie obliczono średnią ilość zdobytych punktów oraz wybrano najlepsze wyniki spośród tych gier.

Już na pierwszy rzut oka, analizując Tabele 4.2 i 4.3, można zauważyć, że mój agent osiągnął lepsze wyniki niż agenci przedstawieni w pracach [4-5]. W pracy z 2018 roku [4] agent wykorzystujący klasyczny algorytm *DQN* osiągnął średni wynik 9,04. Porównując to do mojego agenta z najprostszym, a jednocześnie najgorszym, dostępnym w moim pakiecie algorytmem, wynik jest o około 80% lepszy, osiągając średnio 16,24 punktu. Porównując wynik z tej pracy z najlepszym zastosowanym algorytmem w moim agencie (*Double DQN*), różnica jest jeszcze większa, ponieważ mój agent okazuje się lepszy o około 150%, osiągając średnio 23,36 punktu. Jeśli chodzi o pracę z 2023 roku [5], zauważamy, że osiągnięte wyniki są bardzo zbliżone do wyników z pracy z 2018 roku [4], przy czym są nieznacznie wyższe. W porównaniu do moich wyników, udało mi się znacząco przewyższyć wydajność agenta z obu prac. Analizując Tabelę 4.3, można również stwierdzić, że udało mi się przewyższyć maksymalny wynik osiągnięty przez człowieka [4] (15 punktów) w porównaniu do mojej implementacji agenta z algorytmem *Double DQN* (41 punktów) o aż około 173%.

Po analizie powyższych wyników można wyciągnąć kilka istotnych wniosków. Po pierwsze, wyniki osiągnięte przez mój agent z wykorzystaniem różnych wariantów algorytmu *DQN* są znacząco lepsze niż wyniki agentów z innych prac naukowych [4-5]. Moja implementacja agenta osiągnęła wyższe średnie wyniki oraz najlepsze wyniki w porównaniu do tych prac, co wskazuje na skuteczność zastosowanych wariantów algorytmu *DQN*. Po drugie, rozszerzone wersje algorytmu *DQN*, takie jak *Double DQN* i *Dueling Double DQN*, osiągnęły lepsze wyniki niż klasyczna wersja algorytmu *DQN*. To potwierdza, że wprowadzenie dodatkowych usprawnień i modyfikacji do algorytmu może przyczynić się do poprawy wydajności agenta i osiągnięcia lepszych rezultatów. Po trzecie, wyniki osiągnięte przez mój agent znacznie przewyższają wyniki człowieka w tej grze. Zarówno średnia ilość zdobytych punktów, jak i najlepsze wyniki mojego agenta są znacząco wyższe niż wynik osiągnięty przez człowieka. To dowodzi, że algorytmy oparte na uczeniu maszynowym są w stanie przewyższyć ludzką wydajność w niektórych zadaniach i osiągnąć lepsze rezultaty.

Tabela 4.1 Hiperparametry agenta.

Hiperparametr (nazwa parametru w konstruktorze)	**Wartość**	**Opis**
Współczynnik uczenia (*learningRate*)	0.0001	Wartość współczynnika uczenia dla optymalizatora.
Współczynnik dyskontowy (*gamma*)	0.9	Wartość współczynnika dyskontowego dla przyszłych nagród w funkcji maksymalizującej Q.
Kroki bez uczenia (*stepWithoutLearn*)	150000	Liczba kroków przed rozpoczęciem uczenia przez agenta.
Rozmiar partii (*batchSize*)	24	Rozmiar partii używany podczas uczenia.
Rozmiar pamięci (*memorySize*)	500000	Maksymalny rozmiar bufora pamięci.
Początkowy epsilon (*startEpsilon*)	1.0	Początkowa wartość epsilon dla eksploracji.
Końcowy epsilon (*stopEpsilon*)	0.001	Minimalna wartość epsilon dla eksploracji.
Redukcja epsilonu (*reductionEpsilon*)	0.00000055	Wskaźnik redukcji wartości epsilon.
Rozmiar do zmniejszenia zdjęcia (*sizeResize*)	12	Rozmiar, do którego obrazy ekranu gry są zmniejszane (domyślnie 12).
Nazwa pliku (*fileName*)	"model"	Nazwa pliku do zapisania wag modelu (domyślnie „model”).
Współczynnik ograniczenia (clipByNorm)	1.0	Maksymalna norma gradientów podczas aktualizacji wag sieci.
Zapis modelu po liczbie gier (*saveModelAfterGamesNumber*)	1000	Liczba gier po której ma nastąpić zapis modelu (domyślnie 1000).
Współczynnik interpolacji (*tau*)	0.0006	Parametr interpolacji używany do aktualizacji sieci docelowej. Wymagane tylko przy użyciu *Double DQN*.



Rysunek 4.1: Wykres wyników podczas treningu dla różnych algorytmów.

Tabela 4.2 Średnie wyniki z 50 gier.

Człowiek [4]	1.98
Wei [4]	9.04
Tushar [5]	9.53
DQN	16.24
Dueling DQN	17.16
Double DQN	23.36
Dueling Double DQN	21.06

Tabela 4.3 Najlepsze wyniki z 50 gier.

Człowiek [4]	15
Wei [4]	17
Tushar [5]	20
DQN	29
Dueling DQN	33
Double DQN	41
Dueling Double DQN	39

Podsumowanie

W ramach tej pracy licencjackiej stworzyłem agenta do gry w Węża, wykorzystującego różne warianty algorytmu *DQN*. Agent korzysta z sieci konwolucyjnej do analizy obrazu i określania stanu gry. Dodatkowo, stworzyłem dedykowane środowisko gry, które umożliwiło przetestowanie agenta oraz porównanie wyników z innymi agentami z prac naukowych [4-5].

Udało mi się osiągnąć cel postawiony przed projektem, ponieważ agent nauczył się skutecznych strategii i przewyższył wyniki agentów z innych prac naukowych, a nawet osiągnął wyniki lepsze od możliwości człowieka. Jednakże, nie byłem w stanie udowodnić, że algorytm *Dueling Double DQN* może osiągnąć lepsze wyniki niż *Double DQN* w przypadku gry w Węża. Istnieje możliwość, że ten problem był specyficzny dla tej gry, lub nie udało mi się znaleźć optymalnych parametrów dla algorytmu *Dueling Double DQN*.

Niestety, ze względu na ograniczenia czasowe i mocy obliczeniowej, nie byłem w stanie przetestować bardziej rozbudowanego modelu sieci konwolucyjnej i jego wpływu na zdobywane punkty. Jednakże, można przypuszczać, że dla większej sieci neuronowej i dłuższego czasu treningu, zastosowanie tych technik mogłoby prowadzić do osiągnięcia najwyższego możliwego wyniku w grze.

W rezultacie, ta praca wnosi istotny wkład w badanie algorytmów uczenia ze wzmocnieniem w kontekście gry Wąż. Wyniki potwierdzają skuteczność zastosowanych wariantów algorytmu *DQN* oraz pokazują potencjał dalszych badań i eksperymentów w celu dalszej optymalizacji wyników.

Bibliografia

[1] Christopher JCH Watkins and Peter Dayan, (1992), Q-learning, Machine learning, Kluwer Academic Publishers, 8, 279–292.

[2] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, Anil Anthony Bharath, (2017), A Brief Survey of Deep Reinforcement Learning, IEEE, Signal Processing Magazine.

[3] M. Roderick, J. MacGlashan, and S. Tellex, (2017), Implementing the deep q-network, ArXiv e-prints.

[4] Z. Wei, D. Wang, M. Zhang, A.-H. Tan, C. Miao, and Y. Zhou, (2018), Autonomous agents in snake game via deep reinforcement learning, IEEE International Conference on Agents (ICA).

[5] Md. Rafat Rahman Tushar, Shahnewaz Siddique, (2023), A Memory Efficient Deep Reinforcement Learning Approach For Snake Game Autonomous Agents, Journal of Artificial Intelligence Research.

[6] Pygame. (2023). Pygame Documentation, https://www.pygame.org/docs (dostęp: 06.2023).

[7] Keiron O'Shea, Ryan Nash, (2015), An Introduction to Convolutional Neural Networks, ArXiv e-prints.

[8] PyTorch. (2023). PyTorch Documentation, https://pytorch.org/docs/stable (dostęp: 06.2023).

[9] Shangtong Zhang, Richard S. Sutton, (2018), A Deeper Look at Experience Replay, ArXiv e-prints.

[10] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, Nando Freitas, (2016), Dueling Network Architectures for Deep Reinforcement Learning, Proceedings of The 33rd International Conference on Machine Learning.

[11] Hado van Hasselt, Arthur Guez, David Silver, (2015), Deep reinforcement learning with double Q-learning, arXiv preprint, ArXiv e-prints.

[12] Baiyu Peng, Qi Sun, Shengbo Eben Li, Dongsuk Kum, Yuming Yin, Junqing Wei, Tianyu Gu, (2021), End-to-End Autonomous Driving Through Dueling Double Deep Q-Network, Automotive Innovation volume, 4.

[13] Dash. (2023). Dash Documentation, https://dash.plotly.com (dostęp: 06.2023).

[14] Deque. (2023). Python Software Foundation Documentation, https://docs.python.org/3/library/collections.html#collections.deque (dostęp: 06.2023).