1. Opis rzeczywistego problemu

Cel: Stworzenie inteligentnego agenta AI, który będzie samodzielnie grać w grę Snake, optymalizując swoje decyzje w czasie rzeczywistym, aby osiągać jak najwyższe wyniki, tj. zjadać jak najwięcej owoców i unikać kolizji ze ścianami oraz własnym ciałem.

Motywacja: Praktyczne zastosowanie Reinforcement Learning (RL) pozwala na zrozumienie i demonstrację kluczowych koncepcji RL, takich jak agent, środowisko, stany, akcje, nagrody i kary. Gra Snake, mimo swojej prostoty, oferuje dynamiczne środowisko, które jest dobre do eksperymentowania z algorytmami RL.

2. State of the art

W sekcji drugiej należy zwięźle opisać znane koncepcje (minimum 3) rozwiązania tego lub podobnego problemu ze wskazaniem ich mocnych i słabych stron. Opisywane rozwiązania muszą być różnorodne, tj. korzystać z różnych metod sztucznej inteligencji.

Reinforcement Learning (RL) to gałąź uczenia maszynowego, w której agent uczy się optymalnego zachowania poprzez interakcję ze środowiskiem. Celem agenta jest maksymalizacja skumulowanej nagrody otrzymywanej ze środowiska. Proces uczenia polega na tym, że agent wykonuje akcje w środowisku, obserwuje wynikające z nich nowe stany i otrzymuje nagrody (lub kary).

Mocne strony:

- Zdolność do nauki złożonych strategii: RL może odkrywać i uczyć się strategii, które są zbyt skomplikowane do zaprogramowania ręcznie, nawet w dynamicznych i nieprzewidywalnych środowiskach.
- Adaptacja do dynamicznych środowisk: Agenci RL są w stanie dostosowywać swoje zachowanie do zmieniających się warunków środowiska bez potrzeby ponownego programowania.
- Brak potrzeby etykietowanych danych: W przeciwieństwie do uczenia nadzorowanego, RL nie wymaga wstępnie etykietowanych danych treningowych; agent uczy się poprzez eksperymentowanie.
- Uczenie się na błędach: Agent uczy się poprzez próby i błędy, co pozwala mu na ciągłe doskonalenie się i unikanie wcześniejszych pomyłek.

Słabe strony:

- Potrzeba wielu iteracji i danych: Proces uczenia RL często wymaga ogromnej liczby interakcji ze środowiskiem, co może być kosztowne obliczeniowo i czasochłonne, zwłaszcza w złożonych środowiskach.
- Problem eksploracji/eksploatacji: Agent musi balansować między eksplorowaniem nowych akcji (aby znaleźć potencjalnie lepsze strategie) a eksploatowaniem znanych, dobrych akcji. Niewłaściwa równowaga może prowadzić do nieoptymalnego uczenia.
- Wrażliwość na parametry i nagrody: Wydajność algorytmów RL jest często bardzo wrażliwa na dobór parametrów oraz na sposób projektowania funkcji nagrody. Niewłaściwe nagrody mogą prowadzić do niepożądanych zachowań agenta.

Algorytmy heurystyczne lub regułowe to podejście do tworzenia inteligentnych agentów, w którym decyzje są podejmowane na podstawie z góry zdefiniowanego zestawu reguł i heurystyk. Zamiast uczyć się optymalnego zachowania poprzez interakcję ze środowiskiem (jak w RL), agent po prostu wykonuje instrukcje zaprogramowane przez człowieka.

Mocne strony:

- Prostota i przewidywalność: Łatwe do zrozumienia, zaimplementowania i debugowania. Zachowanie agenta jest w pełni deterministyczne.
- Brak potrzeby uczenia: Agent działa od razu po zaprogramowaniu reguł, nie wymaga fazy treningowej ani dużej ilości danych.
- Niskie zasoby obliczeniowe: Nie wymaga dużej mocy obliczeniowej do działania w czasie rzeczywistym.

Słabe strony:

- Brak adaptacji: Agent nie potrafi uczyć się na błędach ani dostosowywać do nowych, nieprzewidzianych sytuacji. Jego zachowanie jest sztywne i niezmienne.
- Trudność w obsłudze złożonych scenariuszy: W miarę wzrostu złożoności gry (np. większa plansza, więcej przeszkód), liczba reguł potrzebnych do optymalnego działania może stać się zbyt duża i trudna do zarządzania.
- Łatwość „uwięzienia" agenta: Agent może łatwo wpaść w lokalne optima, czyli sytuacje, w których wykonuje optymalne ruchy w krótkim horyzoncie czasowym, ale długoterminowo prowadzi to do porażki (np. zamyka się w pułapce bez wyjścia, mimo że początkowo podążał za jedzeniem).
- Brak generalizacji: Reguły są specyficzne dla danego środowiska i nie można ich łatwo przenieść na inne problemy lub warianty gry.

Algorytmy przeszukujące stany gry są kluczowym narzędziem w sztucznej inteligencji, szczególnie w kontekście gier, gdzie celem jest znalezienie optymalnej sekwencji ruchów. Ich działanie polega na analizie możliwych przyszłych stanów gry, tworząc wirtualne drzewo, w którym każdy węzeł reprezentuje potencjalny stan gry, a każda krawędź – możliwy ruch. Poprzez przeszukiwanie tego drzewa, agent jest w stanie "przewidzieć" konsekwencje swoich decyzji i wybrać ten ruch, który prowadzi do najlepszego możliwego wyniku, biorąc pod uwagę ruchy przeciwnika.

Mocne strony:

- Optymalność w ograniczonym horyzoncie: Algorytmy te są w stanie znaleźć optymalne ruchy w ramach zdefiniowanego horyzontu przeszukiwania, co oznacza, że mogą „przewidywać" przyszłość i podejmować strategiczne decyzje.
- Gwarancja optymalnego rozwiązania (w zakresie przeszukiwania): Dla gier deterministycznych i w ramach ustalonego horyzontu, algorytmy te gwarantują znalezienie najlepszego możliwego ruchu.

Słabe strony:

- Wysoka złożoność obliczeniowa: Przeszukiwanie wszystkich możliwych stanów gry staje się wykładniczo drogie wraz ze wzrostem głębokości horyzontu i liczby możliwych ruchów. W grach o dużej przestrzeni stanów (jak szachy czy Go), pełne przeszukiwanie jest niewykonalne.

- Brak adaptacji do nieprzewidywalnych elementów: Algorytmy te najlepiej sprawdzają się w grach deterministycznych, gdzie każdy ruch prowadzi do przewidywalnego stanu. W grach z elementami losowości (np. rzut kostką) ich efektywność maleje, ponieważ nie mogą dokładnie przewidzieć przyszłych stanów.
- Wymaga funkcji oceny stanu: Skuteczność algorytmu zależy od jakości heurystycznej funkcji oceny, która przypisuje wartość liczbową każdemu stanowi gry. Opracowanie dobrej funkcji oceny może być trudne i czasochłonne.

3. Opis wybranej koncepcji

Po szczegółowej analizie różnych podejść w sekcji State of the Art, Reinforcement Learning (RL), a w szczególności algorytm Q-learning, zostały wybrane jako fundamentalna koncepcja do opracowania inteligentnego agenta AI dla gry Snake. Wybór ten podyktowany jest unikalnymi mocnymi stronami RL, które idealnie pasują do dynamicznego i strategicznego charakteru gry.

Q-learning, jako algorytm uczenia wartości, pozwoli agentowi na naukę optymalnego zachowania poprzez interakcję ze środowiskiem gry. Agent będzie podejmował decyzje ruchowe (akcje) w różnych stanach gry (pozycja węża, położenie jedzenia, obecność przeszkód), a na podstawie otrzymanych nagród (za zjedzenie jedzenia) i kar (za kolizję), będzie aktualizował swoją wiedzę o tym, które akcje są najbardziej wartościowe w danych stanach.

Aby agent Q-learning mógł efektywnie uczyć się optymalnych strategii w grze Snake, kluczowe jest zdefiniowanie, jakie informacje ze środowiska gry będą służyć jako dane wejściowe, reprezentujące bieżący „stan" środowiska. Ten stan musi być na tyle zwięzły, aby przestrzeń stanów była zarządzalna, a jednocześnie wystarczająco informatywny, aby agent mógł rozróżniać między różnymi sytuacjami i podejmować świadome decyzje.

- Elementy stanu gry:
  Pozycja głowy węża (x, y): Absolutne współrzędne głowy węża na planszy. Jest to podstawowy punkt odniesienia dla wszystkich innych elementów.
- Pozycje segmentów ciała węża (lista współrzędnych x, y): Zestaw współrzędnych wszystkich segmentów tworzących ciało węża. Informacja ta jest niezbędna do unikania kolizji z własnym ciałem.
- Pozycja jedzenia (x, y): Absolutne współrzędne owocu na planszy, do którego wąż dąży.
- Aktualny kierunek ruchu węża: Informacja o tym, w którą stronę wąż poruszał się w poprzedniej turze (np. góra, dół, lewo, prawo). Jest to ważne, ponieważ wąż nie może natychmiastowo zawrócić o 180 stopni.

Ze względu na potencjalnie dużą liczbę stanów, zamiast używać bezpośrednich współrzędnych, które mogłyby prowadzić do ogromnej przestrzeni stanów, zastosujemy podejście oparte na relatywnych informacjach i kluczowych wskaźnikach bezpieczeństwa/kierunku:

- Kierunek do jedzenia (relatywny): Zamiast absolutnej pozycji jedzenia, agent otrzyma informację czy jedzenie znajduje się na północ, południe, wschód, czy zachód od głowy węża, lub czy jest w tej samej linii/kolumnie. Może to być reprezentowane przez 4 wartości binarne (np. [jedzenie_na_prawo, jedzenie_na_lewo, jedzenie_do_gory, jedzenie_na_dol]).
- Kierunek aktualny i zagrożenia (relatywne do głowy): Agent będzie „widział" w swoim bezpośrednim otoczeniu. Będzie miał dostęp do informacji o tym, czy w kierunkach: prosto, w

lewo (90 stopni od aktualnego kierunku) i w prawo (90 stopni od aktualnego kierunku) znajduje się przeszkoda (ściana lub ciało węża). Może to być 3 wartości binarne (np. [zagrozenie_prosto, zagrozenie_lewo, zagrozenie_prawo]).
- Kierunek obecnego ruchu: Informacja o tym, w którą stronę wąż się porusza, aby uniemożliwić ruch wsteczny. Może to być 4 wartości binarne, np. [ruch_w_gore, ruch_w_dol, ruch_w_lewo, ruch_w_prawo].

Wyjście algorytmu Q-learning to akcja, którą agent ma podjąć w danym stanie środowiska, aby zmaksymalizować swoją nagrodę. Algorytm, po przetworzeniu aktualnego stanu gry, wybiera akcję, która według niego jest najbardziej optymalna. Celem jest przekształcenie obecnego stanu w nowy, korzystniejszy stan.

Akcje, które może podjąć agent:

- Skręć w lewo: Zmienia kierunek ruchu węża o 90 stopni w lewo względem jego obecnego kierunku.
- Skręć w prawo: Zmienia kierunek ruchu węża o 90 stopni w prawo względem jego obecnego kierunku.
- Jedź prosto: Kontynuuje ruch w obecnym kierunku.

Wpływ akcji na środowisko gry:

- Zmiana pozycji węża: Głowa węża przesuwa się na nową komórkę na planszy zgodnie z wybranym kierunkiem. Całe ciało węża podąża za głową, a ostatni segment ciała jest usuwany (chyba że wąż zjadł jedzenie).
- Zjedzenie jedzenia: Jeśli głowa węża znajdzie się na tej samej pozycji co jedzenie, wąż zjada owoc. Skutkuje to zwiększeniem długości węża (ostatni segment ciała nie jest usuwany), naliczeniem pozytywnej nagrody i pojawieniem się nowego jedzenia w losowej, wolnej komórce na planszy.
- Kolizja ze ścianą: Jeśli głowa węża spróbuje wyjść poza granice planszy, następuje kolizja. Gra kończy się, a agent otrzymuje dużą negatywną nagrodę.
- Kolizja z własnym ciałem: Jeśli głowa węża wejdzie na komórkę zajmowaną przez jego własne ciało, następuje kolizja. Gra również się kończy, a agent otrzymuje dużą negatywną nagrodę.
- Kontynuacja gry: Jeśli żadna z powyższych sytuacji nie ma miejsca, gra trwa dalej, a agent przechodzi do następnego stanu, z potencjalnie małą negatywną nagrodą za każdy ruch (zachęcającą do szybszego zakończenia gry poprzez zjedzenie jedzenia).

Q-Learning to algorytm uczenia przez wzmacnianie, który trenuje funkcję Q (Q-Function) — funkcję wartości akcji — zakodowaną wewnętrznie w tabeli Q (Q-table), która zawiera wartości wszystkich par stan–akcja.

Dla danego stanu i akcji nasza funkcja Q przeszukuje tabelę Q, aby znaleźć odpowiadającą jej wartość.

Gdy trening się zakończy, otrzymujemy optymalną funkcję Q, a tym samym optymalną tabelę Q. A jeśli mamy optymalną funkcję Q, to mamy też optymalną politykę, ponieważ wiemy, dla każdego stanu, którą akcję najlepiej wybrać.

Jednak na początku nasza tabela Q jest bezużyteczna, ponieważ przypisuje arbitralne wartości każdej parze stan–akcja (najczęściej inicjujemy ją wartościami 0). Ale w miarę jak eksplorujemy środowisko i aktualizujemy tabelę Q, będzie ona dostarczać coraz lepszych przybliżeń.

4. Proof of concept

https://github.com/Jasko596/Snake_WdSI.git

Agent nie osiąga optymalnego wyniku, ale radzi sobie z podstawową grą. Nieosiąganie przez agenta wysokich wyników związane jest z tym, że wąż nie jest w stanie „przewidzieć" kilku ruchów do przodu, przez co trafia do sytuacji bez wyjścia. Obecna reprezentacja skupia się na bezpośrednim otoczeniu węża oraz relatywnej pozycji jedzenia, nie uwzględnia ona długoterminowych konsekwencji ruchów agenta (dany ruch może wydawać się bezpieczny, ale za kilka ruchów doprowadzi do pułapki). Agent nie wie, ile jest wolnej przestrzeni, w kierunku, w którym zmierza.

Aby poprawić wynik osiągany przez agenta można go zachęcić do planowania bezpieczniejszej ścieżki poprzez modyfikacje funkcji nagrody. Możemy tak zaprojektować funkcję nagrody, aby karała agenta za wpadanie w sytuacje, które prowadzą do przyszłych pułapek, lub nagradzała za zachowania, które zwiększają jego „przeżywalność". Agent mógłby otrzymywać nagrodę za utrzymywanie otwartej przestrzeni oraz karę za zbliżanie się do niebezpiecznych obszarów.