Zachodniopomorski Uniwersytet Technologiczny w Szczecinie

Wydział Informatyki

Jakub Młokosiewicz

numer albumu: 26875
kierunek studiów: Informatyka
specjalność: Systemy komputerowe i oprogramowanie
forma studiów: stacjonarne

ZASTOSOWANIE UCZENIA ZE WZMOCNIENIEM W DOBORZE STRATEGII REPREZENTOWANEJ PRZEZ SIEĆ NEURONOWĄ W GRZE PLANSZOWEJ CONNECT 4

APPLICATION OF THE REINFORCEMENT LEARNING IN SEARCHING OF THE POLICY REPRESENTED BY THE NEURAL NETWORK FOR CONNECT 4 BOARD GAME

praca dyplomowa inżynierska

napisana pod kierunkiem:

dr inż. Marcina Plucińskiego
Katedra Metod Sztucznej Inteligencji i Matematyki Stosowanej

Data wydania tematu pracy: 21.11.2014
Data złożenia pracy: 29.04.2016

Szczecin 2016

Application of the reinforcement learning in searching of the policy represented by the neural network for Connect 4 board game

Jakub Młokosiewicz
Supervisor: dr inż. Marcin Pluciński

Abstract

There are two basic ways of solving the problem of playing board games alghoritmically – searching the game tree, and learning by interaction with the environment. In the first case it is necessary to define a method of board state evaluation. Reinforcement learning is an example of the latter approach involving agent learning to make good decisions based on received (delayed) reinforcements.

The main goal of the thesis is to design and implement an alghoritm based on reinforcement learning which would allow learning a policy for Connect 4 board game granted that it is represented by an artificial neural network. The secondary objective is to develop application allowing the practical realisation of learning and testing its results.

OŚWIADCZENIE
AUTORA PRACY DYPLOMOWEJ

Oświadczam, że praca dyplomowa inżynierska pt.
Zastosowanie uczenia ze wzmocnieniem w doborze strategii reprezentowanej przez sieć neuronową w grze planszowej Connect 4
napisana pod kierunkiem:
dr inż. Marcina Plucińskiego
jest w całości moi samodzielnym autorskim opracowaniem sporządzonym przy wykorzystaniu wykazanej w pracy literatury przedmiotu i materiałów źródłowych.

Złożona w dziekanacie Wydziału Informatyki treść mojej pracy dyplomowej w formie elektronicznej jest zgodna z treścią w formie pisemnej.

Oświadczam ponadto, że złożona w dziekanacie praca dyplomowa ani jej fragmenty nie były wcześniej przedmiotem procedur procesu dyplomowania związanych z uzyskaniem tytułu zawodowego w uczeniach wyższych.

.....
podpis dyplomanta

Szczecin, dn. 29.04.2016

Spis treści

Spis tabel

Spis rysunków

Rozdział 1

Wprowadzenie

W nauce istnieją obecnie dwa zasadnicze podejścia do problemu "maszynowego" grania w gry planszowe – budowanie i przeszukiwanie drzewa gry oraz uczenie poprzez interakcje ze środowiskiem. W tym pierwszym przypadku konieczne jest zdefiniowanie funkcji oceny sytuacji na planszy. Przykładem drugiego podejścia jest uczenie ze wzmocnieniem, polegające na uczeniu się przez agenta podejmowania odpowiednich decyzji na podstawie otrzymywanych opóźnionych w czasie wzmocnień.

1.1 Cel pracy

Podstawowym celem pracy jest zaprojektowanie i implementacja algorytmu bazującego na uczeniu ze wzmocnieniem, umożliwiającego uczenie się strategii dla gry Connect 4, przy założeniu, że jest ona reprezentowana przez sztuczną sieć neuronową. Celem dodatkowym jest opracowanie aplikacji umożliwiającej praktyczną realizację uczenia oraz testowania jego wyników.

1.2 Struktura pracy

Tak sformułowanym celom pracy podporządkowano jej układ. Rozdział drugi poświęcono grze Connect 4, będącej przedmiotem uczenia się ze wzmocnieniem, do którego wprowadzenie zawarto w trzecim rozdziale pracy. Czwarty rozdział stanowi wprowadzenie do tematyki sieci neuronowych, zastosowanych w niniejszej pracy jako aproksymator funkcji w algorytmie uczenia ze wzmocnieniem. Opis zaprojektowanych i zaimplementowanych jako cel pracy algorytmów został zawarty w rozdziale piątym. Rozdział szósty zawiera opis badań wpływu poszczególnych współczynników na skuteczność uczenia się, przeprowadzonych z wykorzystaniem wcześniej opisanych narzędzi.

Rozdział 2

Gra Connect 4

Gra Connect 4, w Polsce znana także jako Czwórki, jest grą dla dwóch graczy, toczącą się na pionowo ustawionej planszy o 7 kolumnach i 6 wierszach. Gracze naprzemiennie wrzucają od góry swoje żetony, które opadają na dół, zajmując najniższe wolne miejsce w kolumnie. Celem gry jest ułożenie obok siebie – pionowo, poziomo lub na skos – czterech swoich żetonów, zanim uczyni to przeciwnik [1].

Rysunek 2.1: Przykładowa rozgrywka – wygrał gracz grający czerwonymi żetonami[1]

Connect 4 jest grą o pełnej informacji, co znaczy, że gracze grają naprzemiennie i mają informację o wszystkich ruchach, które miały miejsce, a także o wszyskich potencjalych ruchach, które można wykonać w danym stanie gry. Connect 4 jest także grą o sumie zerowej – korzyść jednego gracza oznacza stratę drugiego [1].

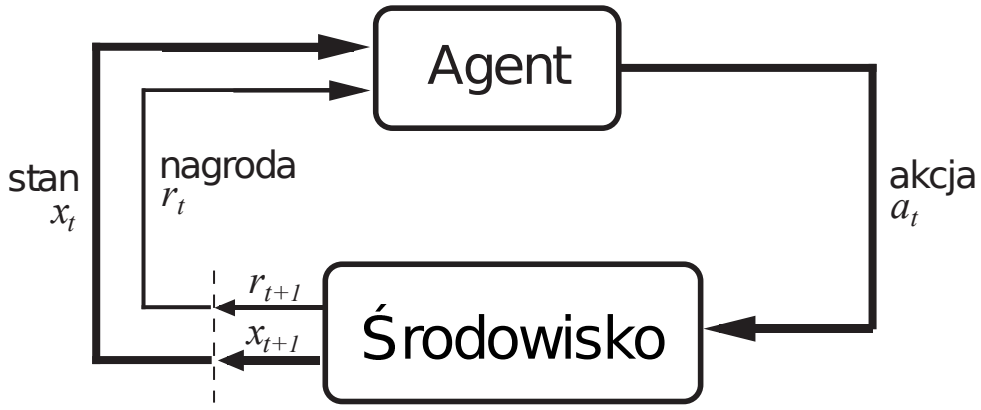
W klasycznym Connect 4 istnieje 4.531.985.219.092 [2] pozycji możliwych do wystąpienia podczas gry. Warto jednak zauważyć, że większość z tych stanów planszy występuje stosunkowo rzadko w porównaniu do innych, częstszych oraz że większość pozycji posiada do pary stan odbity symetrycznie względem środkowej kolumny, równoważny względem planowania strategii gry.

[1] Źródło: `https://commons.wikimedia.org/wiki/File:Connect_Four.jpg`

Rozdział 3

Uczenie ze wzmocnieniem

Uczenie ze wzmocieniem ma za zadanie zamodelować problem uczenia się na podstawie interakcji z otoczeniem, by ostatecznie osiągnąć zadany cel. Uczeń (zarazem jednostka podejmująca decyzje) nazywany jest *agentem*. Wszystko poza nim nazywane jest *środowiskiem*. Agent w ciągły sposób wchodzi w interakcje ze środowiskiem poprzez podejmowanie *akcji*. W odpowiedzi na nie, środowisko przedstawia agentowi swój nowy *stan*. Środowisko daje agentowi *wzmocnienia* o wartościach dodatnich lub ujemnych. Agent stara się maksymalizować ich sumę [3].



Rysunek 3.1: Interakcja agenta ze środowiskiem[1]

Agent oraz środowisko oddziałują na siebie w dyskretnych krokach czasu, $t = 0, 1, 2, 3, ...$[2]. W każdym kroku czasu t, agent obserwuje reprezentację stanu i na jego podstawie wybiera akcję (należącą do zbioru akcji dostępnych w tym stanie). W kolejnym punkcie czasu, jako konsekwencję wybranej akcji, agent otrzymuje numeryczną nagrodę r_t oraz przechodzi do nowego stanu [3].

3.1 Zadanie ucznia

Jeśli zadanie do wykonania przez agenta przedstawione jest tylko w postaci nagród, w najbardziej ogólnym przypadku można powiedzieć, że tak naprawdę polega ono na nauczeniu się strategii, która będzie prowadziła do maksymalizacji sumy

[1]Diagram zaczerpnięty z [3]. Tłumaczenie wykonane przez autora pracy.

[2]Autorzy [3] wspominają w tym miejscu, że czas można traktować również jako wartość ciągłą. Wybiega to jednak poza zakres niniejszej pracy.

otrzymywanych nagród (i pośrednio do zdefiniowanego kryterium jakości, np. wygrania gry) [4]. Jednym z przypadków jest ten, w którym uczeń ma maksymalizować długoterminową sumę nagród – w tym ujęciu dobra strategia może okazać się opłacalna dopiero w dłuższym horyzoncie czasowym. Biorąc pod uwagę ten typ uczenia można zdefiniować kryterium jakości, które system uczący powinien maksymalizować, w postaci oczekiwanej zdyskontowanej sumy otrzymanych nagród:

$$\mathbb{E}\left[\sum_{t=0}^{\infty}\gamma^t r_t\right]. \tag{3.1}$$

Współczynnik dyskontowania $\gamma \in [0,1]$ reguluje ważność krótko- i długoterminowych nagród. Im większa wartość tego współczynnika, tym większe znaczenie mają przyszłe nagrody. W szczególnym przypadku, gdy $\gamma = 0$, uczeń będzie maksymalizował tylko natychmiastowe nagrody; natomiast gry $\gamma = 1$, wzmocnienia ze wszystkich kroków czasu będą dla ucznia równie ważne i brane pod uwagę.

Wiele praktycznych zadań ma charakter epizodyczny, to znaczy taki, że występuje *stan absorbujący* kończący próbę. Równanie zdyskontowanej sumy otrzymanych nagród można wówczas sprowadzić do skończonego przypadku:

$$\mathbb{E}\left[\sum_{t=0}^{n-1}\gamma^t r_t\right], \tag{3.2}$$

gdzie n oznacza liczbę kroków próby.

W uczeniu się ze wzmocnieniem mamy do czynienia z dwoma szczególnymi typami zadań strojących przed agentem. Cichosz nazywa je zadaniami do-sukcesu i do-porażki [4].

Zadania do-sukcesu

W zadaniach tego typu uczeń w każdej kolejnej próbie ma za zadanie osiągnięcie pewnego celu, np. konkretnego stanu środowiska. Osiągniecie tego celu oznacza zakończenie próby sukcesem, dobrze więc, by osiągnięcie go było możliwie najszybsze. Można to osiągnąć przyznając uczniowi wartość wzmocnienia r_1 we wszystkich krokach poprzedzających osiągnięcie celu i $r_2 >= r_1$ po jego osiągnięciu. Aby systemowi uczącemu opłacało się jak najszybsze osiągnięcie sukcesu należy także zapewnić, aby dla każdego $n > 0$ została spełniona poniższa nierówność:

$$\sum_{t=0}^{n-1}\gamma^t r_1 + \gamma^n r_2 > \sum_{t=0}^{n}\gamma^t r_1 + \gamma^{n+1} r_2. \tag{3.3}$$

Tego typu zadaniem jest gra w Connect 4 – agentowi powinno zależeć na możliwie szybkiej wygranej, zanim wygra przeciwnik.

Zadania do-porażki

W zadaniach tego typu zadaniem agenta jest jak najdłuższe unikanie pewnej niepożądanej sytuacji. Jako przykład Cichosz podaje zadanie balansowania odwróconego wahadła. W tym wariancie, odwrotnie niż w poprzednim przypadku, nagroda r_1 otrzymywana we wszystkich pośrednich krokach powinna być większa lub równa

niż r_2 uzyskiwana w wypadku porażki. Aby agentowi opłacało się jak najbardziej odwlekać porażkę, dla dowolnego $n > 0$ musi zostać spełniony warunek:

$$\sum_{t=0}^{n-1} \gamma^t r_1 + \gamma^n r_2 < \sum_{t=0}^{n} \gamma^t r_1 + \gamma^{n+1} r_2. \tag{3.4}$$

3.2 Proces decyzyjny Markowa

Matematycznym modelem dla zadania uczenia się ze wzmocnieniem jest problem decyzyjny Markowa. Polega on na znalezieniu optymalnej strategii decyzyjnej dla środowiska, którego modelem jest proces decyzyjny Markowa [4].

Proces decyzyjny Markowa jest zdefiniowany jako czwórka $\langle X, A, \varrho, \delta \rangle$, gdzie:

X – skończony zbiór stanów środowiska,

A – skończony zbiór akcji możliwych do wykonania przez agenta,

$\varrho(x, a)$ – funkcja wzmocnienia określająca mechanizm nagród, zdefiniowana jako zmienna losowa o wartościach rzeczywistych dla każdej pary $\langle x, a \rangle \in X \times A$, oznaczająca nagrodę otrzymywaną po wykonaniu akcji a w stanie x,

$\delta(x, a)$ – funkcja określająca mechanizm zmian stanów, zdefiniowana jako zmienna losowa o wartościach ze zbioru X dla każdej pary $\langle x, a \rangle \in X \times A$, oznaczająca kolejny stan po wykonaniu akcji a w stanie x.

Z przedstawionej powyżej definicji wynika *własność Markowa*, która ma zasadnicze znaczenie z punktu widzenia poszukiwania strategii dla procesów decyzyjnych Markowa. Własność ta jest nazywana także niezależnością od historii albo niezależnością od ścieżki (sposobu osiągnięcia aktualnego stanu). Zgodnie z własnością Markowa wartości funkcji wzmocnienia ϱ i przejść δ nie zależą od historii wcześniejszych decyzji. W każdym kroku nagroda i następny stan zależą jedynie od aktualnego stanu i akcji w nim podjętej.

3.3 Strategie i funkcje wartości

W każdym kroku interakcji ze środowiskiem agent przechowuje mapowanie stanów na wartość każdej możliwej akcji [3, 4]. Takie odniesienie nazywane jest *strategią* agenta, oznaczaną jako π. Strategią dla procesu decyzyjnego Markowa $\langle X, A, \varrho, \delta \rangle$ jest dowolna funkcja $\pi : X \mapsto A$. Mówimy, że system posługuje się strategią π, jeśli w każdym kroku t wykonuje on akcję $a_t = \pi(x_t)$. Jakość strategii ocenia *funkcja wartości stanu*, która dla procesu decyzyjnego Markowa $\langle X, A, \varrho, \delta \rangle$ ze względu na strategię π jest dla każdego stanu $x \in X$ określana następująco:

$$V^\pi(x) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid x_0 = x \right], \tag{3.5}$$

gdzie $\mathbb{E}_\pi$ oznacza wartość oczekiwaną przy założeniu posługiwania się strategią π. Funkcja przyporządkowuje każdemu stanowi x wartość zdyskontowanej sumy przyszłych nagród, które będą otrzymane przez agenta znajdującego się w tym stanie i posługującego się strategią π.

W ten sam sposób można zdefiniować funkcję oceniającą wartość oczekiwanych przyszłych nagród po wykonaniu przez agenta akcji a w stanie x, nazywaną *funkcją wartości akcji*:

$$Q^{\pi}(x,a) = \mathbb{E}_{\pi}\left[\varrho(x,a) + \sum_{t=1}^{\infty}\gamma^{t} r_t \mid x_0 = x, a_0 = a\right]. \tag{3.6}$$

Z punktu widzenia kryterium jakości będącego zdyskontowaną sumą oczekiwanych nagród, lepsza jest ta strategia, przy której stosowaniu wartość każdego stanu jest nie mniejsza niż przy stosowaniu porównywanej z nią gorszej strategii oraz dodatkowo dla przynajmniej jednego stanu wartość ta jest większa. *Strategią optymalną* jest taka strategia, dla której nie istnieje strategia od niej lepsza (jest możliwe wystąpienie wielu optymalnych strategii). *Strategia zachłanna* względem funkcji wartości stanu to taka strategia, która dla każdego stanu $x \in X$ zwraca akcję prowadzącą agenta do stanu o największej wartości funkcji V. Analogicznie strategią zachłanną względem funkcji wartości akcji będzie strategia zwracająca dla każdego stanu $x \in X$ akcję o największej wartości $Q(x,a)$.

Strategia agenta jest modyfikowana podczas interakcji z otoczeniem, zgodnie z wybraną metodą uczenia ze wzmocnieniem.

3.4 Algorytm Q-learning

Najczęściej stosowaną metodą uczenia ze wzmocnieniem jest algorytm Q-learning, iteracyjnie przybliżający optymalną funkcję wartości akcji Q [4]. W każdym kroku czasu agent wykonuje następujące czynności:

1. obserwuj aktualny stan x_t,
2. wybierz akcję do wykonania a_t na podstawie x_t i Q_t,
3. wykonaj akcję a_t,
4. obserwuj wzmocnienie r_t oraz kolejny stan x_{t+1},
5. uaktualnij przybliżenie Q według wzoru (3.7).

$$\begin{aligned} Q_{t+1}(x_t,a_t) = {} & \underbrace{Q_t(x_t,a_t)}_{\text{poprzednia wartość akcji}} \\ & + \underbrace{\alpha_t(x_t,a_t)}_{\substack{\text{współczynnik}\\ \text{uczenia się}}} \cdot \Bigg(\overbrace{\underbrace{r_{t+1}}_{\text{nagroda}} + \underbrace{\gamma}_{\substack{\text{współczynnik}\\ \text{dyskontowania}}} \cdot \underbrace{\max_a Q_t(x_{t+1},a)}_{\substack{\text{szacowane optymalne}\\ \text{przyszłe wartości}}}}^{\text{nauczona wartość akcji}} \\ & \qquad - \underbrace{Q_t(x_t,a_t)}_{\text{stara wartość akcji}} \Bigg), \end{aligned} \tag{3.7}$$

gdzie:

$Q_t(x_t,a_t)$ – funkcja wartości akcji przyporządkowująca każdemu stanowi x i akcji a wartość zdyskontowanych przyszłych nagród po wykonaniu przez agenta akcji a w stanie x w punkcie czasu t.

3.5 Wybór akcji – eksploracja kontra eksploatacja

Eksploracja to zdobywanie przez przez agenta wiedzy o środowisku, czyli dążenie do wykonania wszystkich możliwych akcji i dotarcia do wszystkich możliwych stanów. Eksploatacja polega na wybieraniu zawsze najlepszych możliwych akcji na podstawie wypracowanej strategii.

Niewystarczająca eksploracja może powodować przedwczesną zbieżność do strategii suboptymalnej (agent nigdy nie odkryje sekwencji akcji pozwalających uzyskać większą sumę nagród), natomiast zbyt duża eksploracja powoduje, że zbieżność jest powolna (szczególnie, gdy dostępnych jest wiele stanów i akcji) [4]. Rozwiązaniem wspomnianego problemu może być zastosowanie strategii probabilistycznej.

3.5.1 Strategia ϵ-zachłanna

W strategii tej z prawdopodobieństwem $\epsilon > 0$ wybierana jest losowa akcja (według rozkładu jednostajnego). Oznacza to, że akcja zachłanna wybierana jest z prawdopodobieństwem $1 - \epsilon$.

Wadą tej strategii jest fakt, że decyzja o wykonaniu przez agenta losowej akcji nie zależy od etapu jego uczenia. Można więc zmniejszać wartość ϵ w miarę uczenia tak, że na początku większość decyzji agenta jest losowa, a w miarę uczenia agent bardziej opiera się na nauczonej strategii zachłannej.

3.5.2 Strategia oparta na rozkładzie Boltzmanna

Przy zastosowaniu tej strategii, szansa na wybór akcji niezachłannej jest tym mniejsza, im wartość funkcji wartości akcji $Q(x, a)$ o największej wartości jest większa od pozostałych.

$$\pi(x, a^*) = \frac{exp(Q((s, a^*)/T)}{\sum_a exp(Q(s, a)/T)} \tag{3.8}$$

gdzie:

$\pi(x, a^*)$ – funkcja przyporządkowująca każdemu stanowi x i akcji a^* prawdopodobieństwo wykonania akcji zachłannej a^* w stanie x przez agenta,

T – parametr większy od zera, nazywany temperaturą, regulujący stopień losowości; im wartość jest większa, tym większa losowość wyboru akcji.

3.6 Reprezentacja funkcji

Dla algorytmu Q-learning istnieje dowód zbieżności, jednak wymaga on założenia o tablicowej reprezentacji funkcji [4]. W praktycznych zastosowaniach taka reprezentacja najczęściej jest bardzo nieefektywna bądź w ogóle niemożliwa do realizacji. Przy dużej ilości stanów funkcja wartości, wartości akcji lub funkcji strategii dla wszystkich stanów może wymagać bardzo dużo pamięci oraz uczenie jej będzie powolne i niepraktyczne, z tej przyczyny, że potrzeba bardzo wielu kroków, by wypełnić tablicę wartościami. W celu rozwiązania tego problemu uczenie ze wzmocnieniem integruje się z aproksymatorami funkcji w taki sposób, że każde odwołanie

do wartości funkcji powoduje odtworzenie jej z aproksymatora. W niniejszej pracy w roli aproksymatora wykorzystano sieć neuronową.

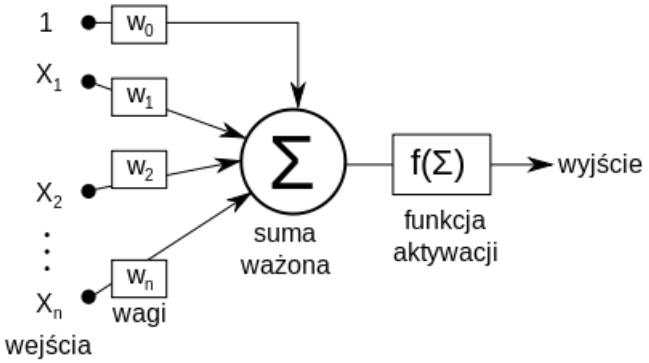
Rozdział 4

Sieci neuronowe

Podstawową cechą sieci neuronowych jest ich zdolność uogólniania, czyli generowania właściwego rozwiązania dla danych, które nie pojawiły się w zestawie danych uczących. W pracy sieć neuronowa została wykorzystana w roli aproksymatora funkcji wartości akcji w procesie uczenia ze wzmocnieniem.

4.1 Model sztucznego neuronu

Sztuczna sieć neuronowa jest bardzo uproszczonym modelem struktury mózgu. W 1943 roku Warren S. McCulloch oraz Walter Pitts opracowali model sztucznego neuronu, na którym opierają się także współcześnie używane modele sieci [5, 6].



Rysunek 4.1: Neuron McCullocha-Pittsa[1]

$X_0, X_1, ..., X_n$ – sygnały wejściowe (przy czym zakładamy, że $X_0 = 1$),

$W_0, W_1, ..., W_n$ – wagi wzmacniające lub osłabiające rolę sygnału z wejść,

f – funkcja aktywacji (może mieć różną postać) - w pierwotnym modelu funkcja typu skoku jednostkowego.

[1] Źródło: `https://pl.wikipedia.org/wiki/Plik:Neuron_McCullocha-Pittsa.svg`

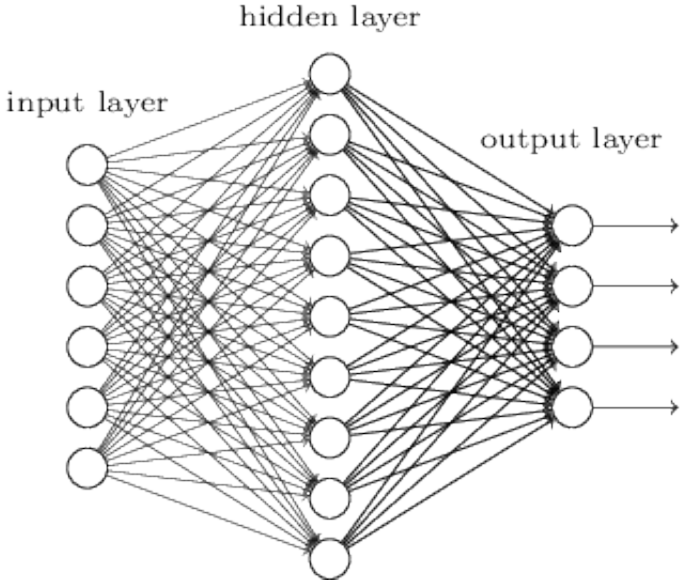
4.2 Wielowarstwowe sieci jednokierunkowe

W dalszej części rozważań przyjęto model neuronu podobny do przedstawionego w poprzednim podrozdziale. Składa się on z elementu sumacyjnego, do którego dochodzą sygnały wejściowe $x_1, x_2, ..., x_N$ tworzące wektor wejściowy $x = [x_1, x_2, ..., x_N]^T$ pomnożone przez przyporządkowane im wagi $W_{i1}, W_{i2}, ..., W_{iN}$ tworzące wektor wag i-tego neuronu $W_i = [W_{i1}, W_{i2}, ..., W_{iN}]^T$ oraz wartość W_{i0} zwana *progiem* (ang. *bias*). Sygnał wyjściowy sumatora oznaczono jako u_i:

$$u_i = \sum_{j=1}^{N} W_{ij} x_j + W_{i0}. \tag{4.1}$$

Sygnał ten jest podawany na blok realizujący funkcję aktywacji, skąd trafia na wejście kolejnego neuronu bądź na wyjście sieci [6].

Wielowarstwowa sieć neuronowa składa się z koncepcyjnych warstw. W podstawowym modelu sieci wszystkie neurony na warstwie są połączone ze wszystkimi neuronami warstwy sąsiedniej. Sygnał dociera do sieci przez *warstwę wejściową* (ang. *input layer*), skąd trafia na wejścia wszystkich neuronów na pierwszej warstwie, gdzie jest przetwarzany przez neuron według powyższego opisu. Następnie trafia na wejścia wszystkich neuronów kolejnej warstwy (jeśli istnieje) i proces się powtarza, bądź na wyjście sieci. Ostatnia warstwa sieci nazywa się *warstwą wyjściową* (ang. *output layer*). Wszystkie warstwy znajdujące się pomiędzy wejściową a wyjściową to *warstwy ukryte* (ang *hidden layers*) [5].



Rysunek 4.2: Przykładowa architektura sieci z 6 wejściami, 9 neuronami na warstwie ukrytej i 4 neuronami na warstwie wyjściowej[2].

Dobór liczby wejść sieci jest uwarunkowany wymiarem wektora danych X. Liczba neuronów na warstwie wyjściowej równa jest rozmiarowi wektora danych zadanych d. Problemem pozostaje dobór ilości warstw ukrytych i ilości neuronów, które się na nich znajdują [6].

[2]Źródło: `http://nb4799.neu.edu/wordpress/?p=246`

4.3 Uczenie sieci – wsteczna propagacja błędów

Sieć neuronowa pełni rolę układu aproksymującego dane uczące (x, d). W trakcie uczenia sieci, dobierane są współczynniki tej funkcji aproksymującej – wektory wag poszczególnych neuronów. Na etapie odtwarzania następuje obliczenie wartości funkcji aproksymującej dla danego wektora wejściowego przy ustalonych wartościach wag [6].

Osowski stwierdza, że najlepsze wyniki w uczeniu sieci można uzyskać poprzez odpowiedni dobór wstępnych wartości wag. Pożądany jest start z wartości wag zbliżonych do optymalnych. Pozwala to uniknąć utknięcia w niewłaściwym minimum lokalnym i jednocześnie przyspiesza proces uczenia. Niestety w ogólnym przypadku nie istnieje metoda doboru wag będących właściwym punktem startowym dla każdego problemu uczenia się. W większości zastosowań korzysta się zwykle z losowego doboru wag, przyjmując rozkład równomierny w określonym przedziale liczbowym [6].

Celem uczenia sieci jest określenie wartości wag neuronów wszystkich warstw sieci w taki sposób, aby przy zadanym wektorze wejściowym x uzyskać na wyjściu sieci sygnały wyjściowe y_i o wartościach równych, z dostateczną dokładnością, wartościom żądanym d_i, dla $i = 1, 2, ..., M$. Przy założeniu ciągłości funkcji celu najskuteczniejszymi metodami uczenia pozostają gradientowe metody optymalizacyjne [6], w których uaktualnianie wektora wag (uczenie) odbywa się zgodnie ze wzorem:

$$W(k+1) = W(k) + \Delta W, \tag{4.2}$$

w którym:

$$\Delta W = \eta p(W), \tag{4.3}$$

η jest współczynnikiem uczenia, a $p(W)$ – określa kierunek największego spadku wartości funkcji błędu w przestrzeni wielowymiarowej W [6].

Zastosowanie metod gradientowych do uczenia sieci wielowarstwowej wymaga do wyznaczenia kierunku $p(W)$ obliczenia wektora gradientu względem wag wszystkich warstw sieci. Jedynie w przypadku wag warstwy wyjściowej zadanie to jest określone w sposób natychmiastowy. Aby określić wspomniany wektor gradnientu dla wektorów pozostałych warstw należy wykorzystać *algorytm propagacji wstecznej* [6].

W każdym cyklu uczącym wyróżnia się następujące etapy uczenia [6]:

1. Analiza sieci neuronowej o zwykłym kierunku przepływu sygnałów przy założeniu sygnałów wejściowych sieci równych elementom aktualnego wektora x. W wyniku analizy otrzymuje się wartości sygnałów wyjściowych neuronów warstw ukrytych oraz warstwy wyjściowej, a także odpowiednie pochodne funkcji aktywacji w poszczególnych warstwach.

2. Realizacja propagacji wstecznej przez odwrócenie kierunków przepływu sygnałów, zastąpienie funkcji aktywacji przez ich pochodne, a także podanie do byłego wyjścia (obecnie wejścia) sieci wymuszenia w postaci odpowiedniej różnicy między wartością aktualną i żądaną. Dla tak utworzonej sieci należy obliczyć wartości odpowiednich różnic wstecznych.

3. Adaptacja wag (uczenie sieci) odbywa się na podstawie wyników uzyskanych w punktach 1 i 2 dla sieci zwykłej i sieci o propagacji wstecznej według odpowiednich wzorów.

Proces opisany w punktach 1, 2, 3 należy powtórzyć dla wszystkich wzorców uczących, kontynuując go do chwili spełnienia warunku zatrzymania algorytmu. Działanie algorytmu kończy się w momencie, gdy norma gradnientu spadnie poniżej pewnej wartości ϵ określającej dokładność procesu uczenia. Dokładniejszy opis metody wstecznej propagacji błędów można znaleźć w [5] oraz [6].

Algorytm wstecznej propagacji błędów jest bardzo podobny koncepcyjnie do wcześniej przedstawionego algorytmu Q-learning (patrz wzór 3.7). Wykorzystując sieć neuronową jako aproksymator funkcji wartości akcji, w celu uaktualnienia przybliżenia wartości Q wystarczy nauczyć sieć próbką uczącą (x, d), gdzie x jest reprezentacją pary stan-akcja, a d nauczoną wartością akcji, ze współczynnikiem uczenia sieci $\eta = \alpha$.

Rozdział 5

Implementacja

Część implementacyjna pracy została przygotowana głównie w języku Python 3 wraz z bibliotekami NumPy i PyBrain (wykorzystanych w implementacji algorytmów uczenia się) oraz Pillow i PyYAML (wykorzystanych w implementacji interfejsu graficznego gry).

5.1 Konwencje stosowane w implementacji

Podczas tworzenia implementacji, m.in. ze względu na specyfikę języka Python (np. dynamiczne typowanie), zdecydowano się na zastosowanie pewnych konwencji. W wypadku ich złamania program nie będzie działał prawidłowo (najprawdopodobniej zakończy działanie z błędem). Poniżej wymieniono najważniejsze z nich, wykorzystywane w strukturach opisywanych w dalszej części pracy.

Akcja, w implementacji najczęściej przechowywana pod nazwą `action` (także `col`), w kontekście gry oznacza numer kolumny planszy, licząc kolejno od lewej, rozpoczynając od zera. Na standardowej pustej planszy do Connect 4 można wykonać następujące akcje: 0, 1, 2, 3, 4, 5, 6.

Identyfikator (numer) gracza (`player_id`) może przyjmować wartość 1 lub 2, kolejno dla gracza pierwszego (rozpoczynającego grę) i drugiego.

5.2 Plansza oraz stan gry

Kod odpowiedzialny za logikę gry opisany w tym podrozdziale znajduje się w pliku `gamestate.py` umieszczonym w folderze `common` implementacji.

5.2.1 Utworzenie planszy oraz wykonywanie ruchów

Planszę do gry opisuje obiekt `BoardState`. Nową pustą planszę można utworzyć za pomocą konstruktora tego obiektu, podając jako argumenty wysokość (ilość wierszy) i szerokość (ilość kolumn) planszy – dla standardowej gry Connect 4 będzie to 6 wierszy i 7 kolumn. Aby wykonać ruch, należy skorzystać z metody `make_move`, przyjmującej jako argumenty kolejno numer kolumny (od lewej; licząc od zera), w której gracz umieszcza żeton oraz identyfikator tego gracza.

Metody operujące na obiektach typu `BoardState`, w tym `make_move`, nie modyfikują stanu obiektu, a zwracają nowy obiekt zawierający porządane zmiany (obiekty klasy `BoardState` są więc w tym sensie *niemutowalne*, niezmienne).

Układ żetonów na planszy jest przechowywany w obiektach `BoardState` w polu `board`, w postaci listy kolejnych rzędów od góry do dołu planszy, które z kolei są listami kolejnych pól od lewej do prawej strony. Pole może mieć wartość `None` – być puste lub 1 albo 2, zgodnie z numerem gracza, który je zajął.

Listing 5.1 Operacje na obiektach `BoardState` – sesja interaktywna interpretera

```
>>> from common.gamestate import BoardState
>>> from pprint import pprint
>>> board_state = BoardState(6,7)
>>> pprint(board_state.board)
[[None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None]]
>>> new_board_state = board_state.make_move(3,1)
>>> pprint(new_board_state.board)
[[None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, 1, None, None, None]]
>>> pprint(board_state.board)
[[None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None]]
>>> new_board_state = new_board_state.make_move(3,2)
>>> pprint(new_board_state.board)
[[None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, 2, None, None, None],
 [None, None, None, 1, None, None, None]]
```

5.2.2 Sprawdzenie rezultatu gry

Funkcję `game_result`, umożliwiającą sprawdzenie stanu gry (czy nastąpiła wygrana, remis albo czy rozgrywka wciąż trwa), umieszczono w definicji obiektu `BoardState`.

Metoda ta w pierszym rzędzie sprawdza, czy na planszy pozostały jeszcze jakieś wolne pola - jeśli nie, zwraca od razu informację o remisie. Następnie, w pętli dla

każdego z dwóch graczy, sprawdza, czy jego bierki nie zostały ułożone w łańcuchu o długości przynajmniej `needed_to_win`, czyli wymaganej do wygranej (w przypadku klasycznej rozgrywki liczba ta wynosi 4; mechanizm został napisany z myślą o możliwej przyszłej modyfikacji). Algorytm polega na próbie wyszukania odpowiednio wielu bierek jednego gracza w jednej linii kolejno w poszczególnych wierszach, kolumnach oraz po skosie.

Metoda zwraca słownik (`dict`) zawierający klucze:

- `result` - zawierający ciąg informujący o rezultacie gry - wygranej (`'won'`) albo remisie (`'tied'`),
- `player_id` - zawierający numer gracza (1 lub 2), który wygrał rozgrywkę (obecny tylko wtedy, gdy rezultatem gry jest wygrana któregoś z graczy)

lub `None`, gdy gra jest jeszcze nierozstrzygnięta.

5.2.3 Przekształcenia obiektów klasy `BoardState`

`board_state.switch_players()` – zwraca obiekt stanu planszy analogiczny do `board_state`, z tym, że żetony pierwszego gracza zostają zamienione z żetonami drugiego gracza.

`board_state.mirror()` – zwraca obiekt stanu planszy analogiczny do `board_state`, z tym że pozycje żetonów są odbite symetrycznie względem środkowej kolumny.

Listing 5.2 Przekształcenia obiektów klasy `BoardState` – sesja interaktywna interpretera

```
>>> pprint(board_state.board)
[[None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, 2, None, None, None],
 [None, None, None, 1, 1, None, None]]
>>> board_state=board_state.make_move(2,2)
>>> pprint(board_state.board)
[[None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, 2, None, None, None],
 [None, None, 2, 1, 1, None, None]]
>>> pprint(board_state.board)
[[None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, 2, None, None, None],
```

```
 [None, None, 2, 1, 1, None, None]]
>>> board_state_switched_players=board_state.switch_players
    ()
>>> pprint(board_state_switched_players.board)
[[None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, 1, None, None, None],
 [None, None, 1, 2, 2, None, None]]
>>> board_state_mirrored=board_state.mirror()
>>> pprint(board_state_mirrored.board)
[[None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None],
 [None, None, None, 2, None, None, None],
 [None, None, 1, 1, 2, None, None]]
```

5.2.4 Stan gry

Do przechowywania informacji o aktualnym stanie gry służy obiekt klasy `GameState`. Kontruktor tego typu przyjmuje kolejno liczbę wierszy (`rows`) i kolumn (`cols`) planszy oraz liczbę żetonów w linii konieczną do wygranej (`needed_to_win`).

5.3 Sterownik gracza

W projekcie przyjęto koncepcję sterownika gracza – obiektu odpowiadającego za wybór poszczególnych ruchów w toku gry. Klasa opisująca sterownik gracza musi definiować co najmniej: stałą `IS_GUI_CONTROLLED` oraz dwie metody – `configure_new_game` i `make_move`.

Listing 5.3 Implementacja sterownika gracza losowego (game_agents/random.py)

```
 6 class RandomDriver(object):
 7     IS_GUI_CONTROLLED = False
 8
 9     def configure_new_game(self, player_id, rows, cols,
       needed_to_win):
10         self.cols = cols
11
12     def make_move(self, board_state):
13         while True:
14             column = randint(0, self.cols - 1)
15             if board_state.board[0][column] is None:
16                 break
17         return column
```

5.4 Agent uczący się ze wzmocnieniem

Model agenta uczącego się ze wzmocnieniem został zdefiniowany w klasie `ReinforcementPlayer`, znajdującej się w pliku `game_agents/reinforcement_backend/reinforcementplayer.py`. Jest on wykorzystywany w sterowniku gracza uczącego się ze wzmocnieniem znajdującym się w pliku `game_agents/reinforcement.py`

Konstruktor klasy `ReinforcementPlayer` jako argumenty przyjmuje kolejno obiekt odpowiedzialny za przechowywanie danych (`data_storage`) i konfigurację (`config`).

Konfiguracja agenta powinna mieć postać słownika o następującej strukturze:

Listing 5.4 Przykładowa konfiguracja agenta

```
{
    "board": {
        "rows": 6,
        "cols": 7
    },
    "needed_to_win": 4,
    "games_to_play": 5000,
    "saving_rate": 1000,
    "learning_rate": 0.05,
    "discount_factor": 1,
    "estimated_optimal_future_value": "best_known",
    "boltzmann_temperature": 1,
    "rewards": {
        "win": 2,
        "loss": -2,
        "tie": 0,
        "each_move": -0.01
    },
    "neural_storage": {
        "hidden_neurons": 50
    }
}
```

board – opis parametrów planszy, na której agent ma nauczyć się grać: "rows" oznacza liczbę wierszy a "cols" liczbę kolumn; w badaniach wykorzystywano standardową planszę z 6 wierszami i 7 kolumnami.

needed_to_win – liczba bierek w linii konieczna do wygranej; w badaniach wykorzystywano standardowe zasady gry.

games_to_play – ilość gier, które agent ma rozegrać (sam ze sobą) w ramach nauki.

saving_rate – częstotliwość zapisywania danych (umożliwia kontynuowanie procesu uczenia po jego wcześniejszym przerwaniu).

learining_rate – współczynnik szybkości uczenia się – α, η.

discount_factor – współczynnik dyskontowania – γ.

estimated_optimal_future_value – metoda szacowania zdyskontowanej wartości przyszłych akcji $Q_t(s_{t+1}, a_t)$ – "mean" lub "best_known" (patrz podrozdział 5.4.1).

boltzmann_temperature – parametr wykorzystywany w algorytmie wyboru akcji (patrz podrozdział 3.5.2).

rewards – wartości nagród przyznawanych agentowi w stanie wygrywającym ("win"), przegrywającym ("loss"), remisu ("tie") lub w każdym innym ("each_move").

Pozostałe klucze, jeśli istnieją, mogą być wykorzystywane przez obiekt odpowiedzialny za przechowywanie danych.

5.4.1 Implementacja uczenia się ze wzmocnieniem

Agent reprezentowany przez obiekt klasy `ReinforcementPlayer` uczy się grać w Connect 4 podczas treningu typu "sam ze sobą" – gra za obu graczy jednocześnie. Grę treningową można uruchomić wywołując metodę `self_play_game` tego obiektu.

Listing 5.5 Metoda odpowiedziana za trening agenta

```
111     def self_play_game(self):
112         board_state = BoardState(self.config['board']['rows'
                ], self.config['board']['cols'])
113         while True:
114             for player_id in (1, 2):
115                 selected_action = self.
                        select_action_boltzmann(player_id,
                        board_state)
116                 new_state = board_state.make_move(
                        selected_action, player_id)
117                 self.update_action_value(player_id,
                        board_state, selected_action, new_state)
118                 board_state = new_state
119                 if self.is_terminal(board_state):
120                     return board_state
```

Jak pokazano na listingu 5.5, funkcja najpierw tworzy pustą planszę do gry o wymiarach zgodnych z konfiguracją przekazaną wcześniej do konstruktora obiektu. Następnie, aż do zakończenia gry, zmienna `player_id` przyjmuje wartość `1` lub `2`. Kolejno następuje wybór akcji zgodnie ze strategią probabilistyczną opartą na rozkładzie Boltzmanna (patrz podrozdział 3.5.2) oraz jej wykonanie (poprzez wywołanie metody `board_state.make_move`), uzyskując tym samym od środowiska nowy stan, który jest następnie wykorzystywany do aktualizacji funkcji wartości akcji (wywołanie `update_action_value`). Następnie nowy stan jest zapisywany jako aktualny i jeśli gra jeszcze się nie skończyła, nadchodzi kolej drugiego gracza, według tego samego schematu.

Warto dokładnie przyjrzeć się wspomnianym metodom.

Listing 5.6 Metoda wybierająca akcję na podstawie strategii probabilistycznej

```
71      def select_action_boltzmann(self, player_id, state):
72          possible_actions = self.get_possible_actions(state)
73          actions_with_values = [(action, self.data_storage.
                get_value_of_action(player_id, state, action))
74                                 for action in
                                       possible_actions]
75          actions, values = zip(*actions_with_values)
76          numerators = np.exp(np.array(values) / self.
                boltzmann_temperature)
77          denumerator = sum(numerators)
78          probabilities = numerators / denumerator
79          return np.random.choice(actions, p=probabilities)
```

Metoda z listingu 5.6 pobiera akcje dostępne w stanie przekazanym jako parametr (numery kolumn, w których są jeszcze wolne pola), następnie pobiera ich wartości (patrz podrozdział 5.5) przewidywanych zdyskontowanych nagród z perspektywy gracza `player_id` i wylicza prawdopodobieństwo wyboru każdej z nich zgodnie z rozkładem Boltzmanna oraz konfiguracją przekazaną do konstruktora obiektu, a następnie zwraca wybraną akcję (została wykorzystana metoda z biblioteki NumPy zaimportowanej pod aliasem `np`).

Listing 5.7 Metoda aktualizująca przybliżenie funkcji wartości akcji

```
106     def update_action_value(self, player_id, old_state,
            action, new_state):
107         reward = self.reward_in_state(player_id, new_state)
108         learned_value = reward + self.config['
                discount_factor'] * self.
                estimated_optimal_future_value(player_id,
                new_state)
109         self.data_storage.update_value_of_action(player_id,
                old_state, action, learned_value)
```

Metoda przedstawiona na listingu 5.7 pobiera odpowiednią wartość wzmocnienia otrzymywanej w stanie `state` (na podstawie konfiguracji), następnie wylicza przewidywaną zdyskontowaną sumę przyszłych nagród przy wybraniu akcji `action` w stanie `old_state` i przekazuje ją do zapisu (patrz podrozdział 5.5).

Warto zauważyć, że w implementacji zawarto dwa warianty funkcji `estimated_optimal_function_value` (odpowiednia funkcja jest przypisywana pod ten alias w konstruktorze obiektu `ReinforcementPlayer` na podstawie konfiguracji). Obie funkcje przedstawiono na listingu 5.8.

Listing 5.8 Dwa warianty funkcji zwracającej przewidywaną zdyskontowaną sumę nagród

```
81      def estimated_optimal_future_value_mean(self, player_id,
            state):
82          possible_actions = self.get_possible_actions(state)
83          if len(possible_actions) == 0:
```

```
84              return 0
85          rewards = 0
86          for action in possible_actions:
87              possible_state = state.make_move(action,
                    other_player(player_id))
88              if not self.is_terminal(possible_state):
89                  best_next_action = self.best_possible_action
                        (player_id, possible_state)
90                  rewards += self.data_storage.
                        get_value_of_action(player_id,
                        possible_state, best_next_action)
91          return rewards / len(possible_actions)
92
93      def estimated_optimal_future_value_best_known(self,
            player_id, state):
94          if not self.is_terminal(state):
95              best_opponents_action = self.
                    best_possible_action(other_player(player_id),
                     state)
96              worst_possible_state = state.make_move(
                    best_opponents_action, other_player(player_id
                    ))
97              if not self.is_terminal(worst_possible_state):
98                  _, best_next_action_value = self.
                        best_possible_action_with_value(player_id
                        , worst_possible_state)
99                  return best_next_action_value
100         return 0
```

W przypadku, gdy `state` jest stanem absorbującym, obie funkcje zwracają wartość `0`.

Pierwsza metoda (`estimated_optimal_future_value_mean`) pobiera wszystkie możliwe akcje do wykonania w stanie `state` i "wykonuje je jako przeciwnik" (nie na oficjalnej planszy). Dla wszystkich tak osiągniętych stanów, które nie są terminalami, pobiera wartość najlepszej do wykonania akcji i zwraca średnią ich wartość.

Druga metoda (`estimated_optimal_future_value_best_known`) działa bardzo podobnie, tyle że nie liczy średniej, a zwraca wartość najlepszej akcji `player_id` po najlepszym ruchu gracza przeciwnego.

Wspomnając o wyborze najlepszych akcji (ruchów) autor ma na myśli najlepsze względem aktualnego przybliżenia funkcji wartości akcji Q.

5.5 Obiekt przechowujący dane

W poprzednim podrozdziale wspomniano, że agent deleguje przechowywanie danych do zewnętrznego obiektu. Taki obiekt musi posiadać następujące metody:

`get_value_of_action(player_id, state, action)` – umożliwiającą pobranie wartości akcji `action` w stanie `state` z perspektywy gracza `player_id`,

`update_value_of_action(player_id, state, action, learned_value)` – umożliwiającą zmianę wartości akcji `action` w stanie `state` z perspektywy gracza `player_id` na `learned_value`,

`save()` – zapisującą dane (np. w pliku na dysku).

W niniejszej pracy zaprojektowano klasę `NeuralStorage`, znajdującą się w pliku `storages/neuralstorage.py`. Definiuje ona wspomniane wyżej metody. Obiekt typu `NeuralStorage` korzysta z następujących pól konfiguracji agenta:

board – wielkość planszy jest wykorzysywana przy dobieraniu rozmiaru wejścia sieci,

learning_rate – określa szybkość uczenia się sieci neuronowej,

neural_storage – "hidden_neurons" określa liczbę neuronów na warstwie ukrytej sieci.

5.5.1 Implementacja sieci neuronowej

Do zaimplementowania sieci neuronowej wykorzystano bibliotekę PyBrain. Zastosowano sieć z `liczba_wierszy * liczba_kolumn + 1` wejściami. Kolejne wejścia oznaczają wartości kolejnych pól na planszy w danym stanie, od góry do dołu oraz od lewej do prawej strony – `0` dla pola pustego, `1` dla pola zajętego przez pierwszego gracza oraz `2` oznaczająca pole zajęte przez drugiego gracza. Ostatnie wejście przyjmuje wartość od `0` do `liczba_kolumn-1` i oznacza numer akcji (kolumny). W sieci umieszczono jedną warstwę ukrytą o rozmiarze ustawianym poprzez opcję w konfiguracji, zawierającą neurony o sigmoidalnej funkcji aktywacji. Na warstwie wyjściowej znajduje się jeden neuron o liniowej funkcji aktywacji. Wyjście z sieci oznacza wartość akcji w stanie przekazanym na wejściu. Początkowe wagi są losowane z zakresu $[-1, 1]$.

5.5.2 Zabiegi ułatwiające uczenie

Podczas zapisu i odczytu danych w sieci neuronowej stosowane są pewne zabiegi, które mają na celu usprawnienie procesu uczenia.

Odbijanie planszy względem środkowej kolumny

Jak wspomniano we wstępie, w Connect 4 występują stany identyczne z punktu widzenia gry, jednak inaczej reprezentowane na planszy – są to stany symetryczne względem środkowej kolumny. Skorzystano więc z tego faktu i przy każdym zapisie wartości akcji proces uczenia sieci dokonywany jest także dla tej samej wartości dla symetrycznych względem środkowej kolumny stanu i akcji (nie dzieje się tak, gdy stan zapisywany jest tożsamy ze swoim odbiciem).

Zapis z punktu widzenia pierwszego gracza

Jako że na wejściu sieć nie dostaje informacji, z punktu widzenia którego gracza ma być obliczona wartość Q, przy uczeniu i odczycie danych z perspektywy drugiego gracza zastosowano konwencję zamiany wszystkich pól z wartości `1` na `2` i odwrotnie.

5.6 Narzędzia

Na potrzeby badań przygotowano zbiór skryptów w językach Python oraz bash.

5.6.1 Tworzenie i uczenie agentów

Do utworzenia oraz wytrenowania nowego agenta potrzebny jest odpowiedni plik w formacie JSON[1] (o rozszerzeniu `.config`) z opisem zmiennych wykorzystywanych w procesie uczenia (zawartość pliku powinna być analogiczna do konfiguracji agenta przedstawionej na listingu 5.4).

Za utworzenie i wytrenowane agenta na podstawie pliku konfiguracyjnego odpowiada skrypt `reinforcement_train.py` (umieszczony w głównym katalogu implementacji), będący w istocie "aliasem" do modułu `game_agents.reinforcement_backend.train`.

Skrypt można uruchomić z różnymi kombinacjami parametrów. W najprostszym przypadku skrypt przyjmuje jedynie ścieżkę do pliku konfiguracyjnego, na podstawie którego ma zostać wytrenowany nowy agent. Skrypt wyświetli identyfikator nowoutworzonego agenta (w schemacie nazwa_konfiguracji-kolejny numer, np. zakładając nazwę pliku konfiguracyjnego test.config nazwą agenta mógłby być ciąg test-1). Proces treningu można przerwać (np. korzystając z kombinacji klawiszy Ctrl+C) i wznowić w późniejszym czasie – służy do tego opcja `-resume` (w skrócie `-r`) z identyfikatorem treningu (wyświetlonym przez skrypt po uruchomieniu procesu uczenia).

Skrypty w języku bash, automatyzujące trenowanie agentów na poszczególnych etapach badań znajdują się w folderze `research/batch_training_scripts` pod nazwami `run_stage_X.sh`, gdzie `X` jest słownym zapisem kolejnych liczb w języku angielskim oznaczającymi etap badań.

5.6.2 Porównywanie agentów

W celu oceny jakości dobranych wzpółczynników konfiguracji agenta stworzono zestaw narzędzi umożliwiających zestawianie ze sobą agentów w celu wzajemnej gry.

Najprostszym sposobem na zestawienie dwóch agentów w grze jest skorzystanie ze skryptu `agents_duel.py` (umieszczonego w głównym katalogu implementacji). Opcja `--list-drivers` (lub krótsza `-d`) powoduje wypisanie wszystkich dostępnych konfiguracji sterowników gracza (nazwa budowana jest w postaci *nazwa_sterownika-nazwa_konfiguracji-kolejny_numer_treningu*). Uruchomienie pojedynku sprowadza się do wywołania skryptu z trzema parametrami – kolejno dwoma nazwami sterowników graczy i liczbą gier, które mają zostać rozegrane. Skrypt zwróci trzy liczby, kolejno: liczbę wygranych pierwszego gracza, liczbę remisów, liczbę wygranych gracza drugiego.

Listing 5.9 Przykładowe wywołanie skryptu `agents_duel.py`

```
$ agents_duel.py --list-drivers
random
reinforcement-stage_one_best_known_5k-1
```

[1] Oficjalny standard definiujący format JSON można znaleźć pod adresem `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`

```
reinforcement-stage_one_best_known_5k-2
reinforcement-stage_one_best_known_5k-3
...
$ ./agents_duel.py random random 1000
557 0 443
```

Użycie wspomnianego skryptu może być jednak kłopotliwe, gdy występuje potrzeba porównania wielu konfiguracji agentów z pozostałymi. Z tego względu postanowiono na podstawie części modułu `game_agents.duel` utworzyć zestaw użytecznych funkcji, umożliwiających przygotowanie w prosty i przejrzysty sposób skryptów porównujących wiele konfiguracji agentów. Poniższe funkcje zdefiniowano w pliku `research/stage.py`:

`run_duels(pairs, games_per_duel)` – przyjmuje (uporządkowane) pary (listy lub krotki) nazw sterowników graczy oraz liczbę pojedynków, które mają być przeprowadzone w każdej z tych par; wyświetla na ekran ranking konfiguracji na podstawie średniej liczby gier wygranych przez sterowniki przez nie wytrenowanych

`all_play_all(drivers_names, games_per_duel=1)` – przyjmuje listę sterowników graczy, a następnie każdy z nich dobiera w parę ze wszystkimi pozostałymi (zarówno jako gracza pierwszego, jak i drugiego), za wyjątkiem tych wytrenowanych za pomocą tej samej konfiguracji; następnie wywołuje `run_duels`

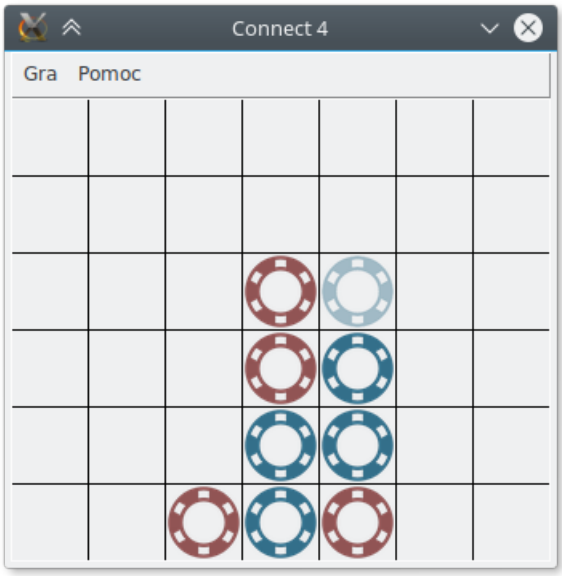
`all_with_random(drivers_names, games_per_duel)` – przyjmuje listę sterowników graczy, a następnie każdy z nich dobiera w parę z graczem losowym (zarówno jako gracza pierwszy, jak i drugiego); następnie wywołuje `run_duels`

Warto wspomnieć, że listę wszystkich dostępnych sterowników graczy (agentów wytrenowanych według konfiguracji) można uzyskać wywołując funkcję `get_available_drivers` z modułu `game_agents.duel`.

Z wykorzystaniem powyższych funkcji utworzono skrypty rankingujące dla każdego etapu badań. Znajdują się one w folderze `research` pod nazwami `stage_X_all_play_all.py` i `stage_X_all_with_random.py`, gdzie `X` jest słownym zapisem kolejnych liczb w języku angielskim oznaczającymi etap badań.

5.7 Graficzny interfejs gry

Interfejs graficzny umożliwia w prosty sposób rozegranie gry w Connect 4 pomiędzy dwoma osobami, osobą i agentem komputerowym oraz dwoma agentami. Aby uruchomić aplikację należy skorzystać ze skryptu `run_gui.py`. Ze względu na ograniczone miejsce została ona skonfigurowana w taki sposób, aby w menu wyboru sterowników graczy pokazywać tylko te sterowniki, które zostały przetestowane za jej pomocą w części badawczej. Aby zagrać z innymi sterownikami graczy należałoby zmodyfikować funkcję `should_be_visible` we wspomnianym wcześniej skrypcie.



Rysunek 5.1: Przykładowa rozgrywka w aplikacji okienkowej

Rozdział 6

Badania

Badania polegały na porównywaniu między sobą różnych konfiguracji parametrów uczenia się agenta ze względu na jego skuteczność w grze z agentami wytrenowanymi z użyciem innych konfiguracji oraz z graczem podejmującym decyzje losowo. Na podstawie każdej wspominanej konfiguracji zostało wytrenowanych pięciu agentów, a wyniki umieszczone w tabelach są wynikiem wyciągnięcia średniej z wyników wszystkich gier rozegranych przez agentów wytrenowanych według danej konfiguracji.

Porównywanie konfiguracji między sobą odbywa się na takiej zasadzie, że każdy agent jest dobierany w parę ze wszystkimi agentami, którzy nie byli wytrenowani na podstawie jego konfiguracji. Każda para rozgrywa dwie gry – każdy agent gra raz jako gracz pierwszy i raz jako gracz drugi. Nie istnieje potrzeba rozgrywania większej liczby gier, gdyż agenci nie wykazują losowości – odtwarzają jedynie wiedzę, której nauczyli się podczas gier treningowych.

Drugim sposobem porównywania konfiguracji jest zestawianie agentów z graczem losowym. W tym wypadku każdy wytrenowany agent rozgrywa grę z graczem losowym 1000 razy jako gracz pierwszy i 1000 razy jako gracz drugi. Większa liczba rozgrywek jest konieczna ze względu na dużą zmienność wyników gier z graczem losowym, którą można wyeliminować uśredniając je.

Badania rozpoczęto od następującej, wstępnie dobranej eksperymentalnie konfiguracji, pokazanej na listingu 6.1. Na poszczególnych etapach badań zmieniane są wspominane elementy konfiguracji. W kolejnych etapach wykorzystywano najlepiej ocenione parametry z poprzedniego etapu.

Celem eksperymentów jest wyłonienie konfiguracji, według której dałoby się wytrenować agenta mogącego wygrać z człowiekiem.

Listing 6.1 Wstępna konfiguracja agenta

```
{
    "board": {
        "rows": 6,
        "cols": 7
    },
    "needed_to_win": 4,
    "games_to_play": 5000,
    "saving_rate": 1000,
    "learning_rate": 0.05,
```

```
    "discount_factor": 1,
    "estimated_optimal_future_value": "best_known",
    "boltzmann_temperature": 1,
    "rewards": {
        "win": 2,
        "loss": -2,
        "tie": 0,
        "each_move": -0.01
    },
    "neural_storage": {
        "hidden_neurons": 50
    }
}
```

6.1 Etap 1. Metody szacowania wartości przyszłych nagród oraz liczba gier do rozegrania

W pierwszym etapie badań porównano dwie metody wyznaczania szacowanej wartości przyszłych nagród (`estimated_optimal_future_value`; patrz podrozdział 5.4) – `best_known` lub `mean` – oraz wpływ ilości rozegranych gier treningowych (`games_to_play`) – 5000, 10.000, 50.000, 100.000 lub 150.000 – na skuteczność agenta.

Tabela 6.1: Etap 1. Wzajemne rozgrywki między konfiguracjami

Nazwa konfiguracji	**Procent wygranych gier**		
	średni	**max**	**min**
stage_one_best_known_50k	58.44%	67.78%	52.22%
stage_one_best_known_100k	58.00%	64.44%	50.00%
stage_one_mean_150k	52.00%	60.00%	43.33%
stage_one_best_known_150k	51.11%	57.78%	42.22%
stage_one_best_known_5k	50.89%	61.11%	37.78%
stage_one_mean_50k	49.56%	58.89%	41.11%
stage_one_mean_5k	48.00%	58.89%	35.56%
stage_one_mean_100k	46.89%	62.22%	38.89%
stage_one_mean_10k	42.67%	48.89%	30.00%
stage_one_best_known_10k	39.78%	47.78%	27.78%

Tabela 6.2: Etap 1. Rozgrywki z graczem losowym

Nazwa konfiguracji	**Procent wygranych gier**		
	średni	**max**	**min**
stage_one_mean_150k	79.68%	81.60%	77.70%
stage_one_mean_100k	75.69%	78.80%	68.65%
stage_one_best_known_50k	71.63%	76.45%	63.50%
stage_one_best_known_100k	71.12%	78.05%	62.45%

stage_one_mean_50k	70.50%	74.80%	66.80%
stage_one_best_known_10k	65.66%	72.95%	59.95%
stage_one_best_known_150k	64.81%	74.40%	60.65%
stage_one_best_known_5k	62.60%	69.35%	58.50%
stage_one_mean_5k	62.10%	68.60%	56.00%
stage_one_mean_10k	61.16%	64.95%	57.25%

Przyglądając się danym zaprezentowanym w tabelach 6.1 oraz 6.2 można zaobserwować pewne zasadnicze różnice – w rankingu gier agentów pomiędzy sobą, skuteczniejsza okazała się metoda szacowania funkcji wartości akcji na podstawie najbardziej sprzyjającego przeciwnikowi ruchu i najbardziej opłacalnej późniejszej akcji agenta. W rozgrywce z graczem losowym skuteczniejsza okazała się strategia uśredniająca.

Na tym etapie za najlepszą konfigurację uznano `stage_one_best_known_50k`. Przemówiło za tym kilka względów. W obu zestawieniach ta konfiguracja zajęła wysokie miejsca – w tym pierwsze w grach z pozostałymi. Czas trenowania rośnie wraz z liczbą gier do rozegrania, a metoda wyznaczania przewidywanej wartości akcji metodą uśredniania zajmuje około dwa razy więcej czasu niż poleganie najlepszym znanym ruchu przeciwnika.

6.2 Etap 2. Liczba neuronów na warstwie ukrytej sieci

W drugim etapie porównano konfiguracje z różną liczbą neuronów (`hidden_neurons`) na warstwie ukrytej sieci, aproksymującej funkcję wartości: 25, 50, 75, 100 lub 125.

Tabela 6.3: Etap 2. Wzajemne rozgrywki między konfiguracjami

Nazwa konfiguracji	**Procent wygranych gier**		
	średni	**max**	**min**
stage_two_100	54.50%	65.00%	37.50%
stage_two_50	53.00%	62.50%	42.50%
stage_two_75	52.00%	60.00%	45.00%
stage_two_125	45.50%	67.50%	32.50%
stage_two_25	43.50%	57.50%	30.00%

Tabela 6.4: Etap 2. Rozgrywki z graczem losowym

Nazwa konfiguracji	**Procent wygranych gier**		
	średni	**max**	**min**
stage_two_75	75.81%	79.40%	73.65%
stage_two_50	75.02%	83.80%	63.80%
stage_two_25	73.75%	77.50%	70.80%
stage_two_100	72.37%	83.00%	61.00%
stage_two_125	66.54%	74.75%	55.65%

Dobry model powinien być na tyle prosty, by nie być podatny na zbytnie dopasowanie się do danych uczących. Wyraźnie można zaobserwować, że od pewnego momentu im większa liczba neuronów na warstwie ukrytej sieci, tym gorszy wynik. Dlatego zdecydowano się na wybranie konfiguracji `stage_two_50` jako najlepszej w tym etapie – utrzymuje stałe drugie miejsce w obu rankingach oraz używa stosunkowo małej liczby neuronów.

6.3 Etap 3. Temperatura w rozkładzie Boltzmanna

W trzecim etapie porównano różne wartości współczynnika temperatury w rozkładzie Boltzmanna wykorzystanym do randomizowania wyboru akcji przez agenta w taki sposób, by zapewnić odpowiednią eksplorację stanów środowiska. Przetestowano wartości `0.5`, `1`, `1.5` oraz `2`.

Tabela 6.5: Etap 3. Wzajemne rozgrywki między konfiguracjami

Nazwa konfiguracji	**Procent wygranych gier**		
	średni	**max**	**min**
stage_three_1.5	56.00%	73.33%	33.33%
stage_three_1.0	51.33%	63.33%	40.00%
stage_three_2.0	48.67%	73.33%	33.33%
stage_three_0.5	42.67%	63.33%	30.00%

Tabela 6.6: Etap 3. Rozgrywki z graczem losowym

Nazwa konfiguracji	**Procent wygranych gier**		
	średni	**max**	**min**
stage_three_2.0	75.04%	77.90%	70.45%
stage_three_1.5	74.50%	80.35%	70.25%
stage_three_1.0	73.60%	79.55%	59.15%
stage_three_0.5	70.38%	79.25%	61.60%

Obserwując zestawienie jako najlepszą konfigurację wybrano `stage_three_1.5`, która w pierwszym rankingu uzyskała pierwsze miejsce, a w drugim drugie, z bardzo małą różnicą względem pierwszego. Można więc skonkludować, że dość duża losowość wyboru akcji wspomaga proces uczenia, jednak nie powinna ona być zbyt duża.

6.4 Etap 4. Współczynnik dyskontowania

W kolejnym etapie zbadano współczynnik dyskontowania, oznaczający jak ważne są dla agenta przyszłe wzmocnienia. W konfiguracjach zawarto następujące wartości: `0` (tylko natychmiastowe nagrody są brane pod uwagę), `0.25`, `0.5`, `0.75` oraz `1` (liczą się wszystkie szacowane przyszłe wzmocnienia).

Tabela 6.7: Etap 4. Wzajemne rozgrywki między konfiguracjami

Nazwa konfiguracji	Procent wygranych gier		
	średni	max	min
stage_four_1.00	58.00%	72.50%	32.50%
stage_four_0.75	54.00%	62.50%	42.50%
stage_four_0.50	52.50%	92.50%	30.00%
stage_four_0.00	48.50%	60.00%	30.00%
stage_four_0.25	36.50%	47.50%	22.50%

Tabela 6.8: Etap 4. Rozgrywki z graczem losowym

Nazwa konfiguracji	Procent wygranych gier		
	średni	max	min
stage_four_0.00	72.41%	81.35%	65.60%
stage_four_1.00	72.35%	78.15%	68.45%
stage_four_0.50	71.77%	77.75%	64.80%
stage_four_0.75	70.28%	75.10%	63.70%
stage_four_0.25	65.38%	70.20%	62.50%

Do dalszych badań wybrano wartość `1` współczynnika dyskontowania. W obu przypadkach konfiguracja z tą wartością współczynnika zajęła wysokie (odpowiednio pierwsze i drugie) miejsca.

6.5 Etap 5. Współczynnik szybkości uczenia się

W piątym etapie badań zbadano różne wartości współczynnika szybkości uczenia się: `0.001`, `0.01`, `0.05` oraz `0.1`.

Tabela 6.9: Etap 5. Wzajemne rozgrywki między konfiguracjami

Nazwa konfiguracji	Procent wygranych gier		
	średni	max	min
stage_five_0.05	62.00%	73.33%	46.67%
stage_five_0.1	49.33%	63.33%	43.33%
stage_five_0.001	46.00%	63.33%	26.67%
stage_five_0.01	42.00%	56.67%	36.67%

Tabela 6.10: Etap 5. Rozgrywki z graczem losowym

Nazwa konfiguracji	Procent wygranych gier		
	średni	max	min
stage_five_0.05	72.04%	81.10%	63.45%
stage_five_0.1	67.59%	77.95%	59.05%
stage_five_0.001	60.79%	63.30%	58.00%
stage_five_0.01	58.52%	68.65%	53.05%

W przypadku obu zestawień najskuteczniejszy okazał się współczynnik `0.05`, dlatego też został on wybrany do dalszych etapów badań.

6.6 Etap 6. Nagrody

W szóstym etapie badań porównano różne kombinacje wartości nagród. Przygotowano następujące zestawy wzmocnień:

Tabela 6.11: Etap 6. Porównywane zestawy wzmocnień

Numer zestawu	**Wartość wzmocnienia**			
	za wygraną	**za przegraną**	**za remis**	**za każdy ruch**
1	2	-2	0	-0.01
2	2	-2	0	0
3	2	0	0	0
4	0	-2	0	0
5	5	-2	0	0
6	2	-5	0	0

Tabela 6.12: Etap 6. Wzajemne rozgrywki między konfiguracjami

Nazwa konfiguracji	**Procent wygranych gier**		
	średni	**max**	**min**
stage_six_6	58.80%	76.00%	48.00%
stage_six_3	53.60%	64.00%	28.00%
stage_six_2	50.40%	70.00%	34.00%
stage_six_1	49.20%	58.00%	36.00%
stage_six_5	45.60%	58.00%	36.00%
stage_six_4	40.80%	44.00%	36.00%

Tabela 6.13: Etap 6. Rozgrywki z graczem losowym

Nazwa konfiguracji	**Procent wygranych gier**		
	średni	**max**	**min**
stage_six_1	75.11%	80.85%	69.35%
stage_six_2	71.81%	80.05%	59.60%
stage_six_6	69.26%	75.50%	59.15%
stage_six_3	66.76%	73.30%	58.25%
stage_six_4	62.74%	69.30%	59.70%
stage_six_5	61.33%	66.90%	58.40%

Na podstawie wyników porównań możemy stwierdzić, że podobnie nieskuteczni zarówno w grze z pozostałymi agentami, jak i z graczem losowym, byli agenci wytrenowani zgodnie z konfiguracjami nr 5 oraz nr 6. W grze z pozostałymi agentami skuteczni okazali się agenci dużo bardziej karani za porażkę niż nagradzani za wygraną (zestaw 6). W grze z graczem losowym przewagę uzyskali agenci, którzy podczas treningu byli nagradzani i karani w podobny sposób oraz dodatkowo

delikatnie karani za każdy wykonany ruch.

6.7 Rozgrywka z wykorzystaniem interfejsu graficznego

Na zakończenie postanowiono porównać konfiguracje `stage_six_6` oraz `stage_six_1`: najlepsze z obu rankingów z poprzedniego porównania, za pomocą gry poprzez interfejs graficzny. Autor rozegrał po jednej grze z każdym agentem wytrenowanym na podstawie wspomnianych konfiguracji, zarówno jako gracz pierwszy, jak i drugi, w rezultacie stwierdzając, że agenci nie są skuteczni w grze z człowiekiem. Każda gra skończyła się przegraną agenta.

Rozdział 7

Podsumowanie

Pracę rozpoczęto od przedstawienia celu i struktury pracy, którym podporządkowano dalszą część wywodu. Po zaprezentowaniu zasad gry Connect 4, skupiono się na koncepcji uczenia ze wzmocnieniem. Ukazano jego teoretyczną podstawę – procesy decyzyjne Markowa, najpopularniejszy algorytm tego typu – Q-learning oraz techniki mające zapewnić równowagę pomiędzy eksploracją i eksploatacją środowiska. Następnie dokonano krótkiego wprowadzenia do tematyki sieci neuronowych.

Dla realizacji celu pracy zaprojektowano algorytm bazujący na uczeniu ze wzmocnieniem, umożliwiający uczenie się strategii dla gry Connect4, przy założeniu, że jest ona reprezentowana przez sieć neuronową. Wykonano jego implementację w języku Python wykorzystując biblioteki do obliczeń numerycznych oraz implementację sieci neuronowej (PyBrain). Opracowano także skrypty umożliwiające tworzenie i uczenie agentów oraz testowanie jego wyników poprzez rozgrywanie gier pomiędzy agentami lub pomiędzy nimi a graczem losowym.

Na poszczególnych etapach badań sprawdzano wpływ zmian wartości wybieranych kolejno parametrów uczenia na skuteczność gry agenta. Efekty uczenia się agentów okazały się niesatysfakcjonujące: w kontekście ich gry pomiędzy sobą najlepsza wśród przeprowadzonych prób konfiguracja osiągnęła średnią wartość 62% wygranych gier, natomiast najlepsza w kontekście gry z graczem losowym – 79,68%. W grze z człowiekiem agenci popełniali podstawowe błędy (z perspektywy logicznego myślenia).

Uzyskane rezultaty potwierdzają, że uczenie ze wzmocnieniem jest techniką wrażliwą na wartości współczynników. Dla uzyskania lepszej skuteczności konieczne byłyby dalsze badania oraz modyfikacje. Poprawie wyników uczenia mogłaby służyć na przykład modyfikacja reprezentacji stanu przekazywanej na wejście sieci neuronowej aproksymującej funkcję wartości akcji.

Bibliografia

[1] Connect four. https://en.wikipedia.org/w/index.php?title=Connect_Four&oldid=713871973. [dostęp: 2016-04-06 20:14].

[2] S. Edelkamp, P. Kissmann. Symbolic classification of general two-player games. http://www.tzi.de/~edelkamp/publications/conf/ki/EdelkampK08-1.pdf.

[3] R. S. Sutton, A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.

[4] P. Cichosz. *Systemy uczące się.* Wydawnictwa Naukowo-Techniczne, Warszawa, 2000.

[5] M. Flasiński. *Wstęp do sztucznej inteligencji.* Wydawnictwo Naukowe PWN, Warszawa, 2011.

[6] S. Osowski. *Sieci neuronowe w ujęciu algorytmicznym.* Wydawnictwa Naukowo-Techniczne, Warszawa, 1996.

Dodatek A

Instalacja potrzebnego oprogramowania

Do uruchomienia programów stanowiących część implementacyjną niniejszej pracy konieczny jest interpreter języka Python 3 oraz zainstalowane dodatkowych bibliotek NumPy, SciPy, PyBrain (wykorzystanych w implementacji algorytmów uczenia się) oraz Pillow i PyYAML (wykorzystanych w implementacji interfejsu graficznego gry).

A.1 Instalacja interpretera języka Python 3

A.1.1 Linux

W celu instalacji interpretera języka najlepiej skorzystać z menedżera pakietów, np. apt-get (Debian), pacman (Arch Linux). Pakiet powinien nazywać się, w zależności od dystrybucji, `python3` lub `python`.

A.1.2 Windows

Do instalacji Pythona 3 w systemie Windows najlepiej wykorzystać gotową dystrybucję wraz z pakietem bibliotek, np. WinPython[1], gdyż instalacja niektórych z nich (NumPy, SciPy) mogłaby wymagać osobnego doinstalowania narzędzi niezbędnych do ich kompilacji.

A.2 Instalacja potrzebnych bibliotek

A.2.1 Linux

Biblioteki NumPy oraz SciPy najlepiej zainstalować korzystając z menedżera pakietów, co pozwala na uniknięcie procesu kompilacji oraz instalowania potrzebnych do niego narzędzi. Pakiety powinny nazywać się, podobnie jak w poprzednim podrozdziale – w zależności od dystrybucji, `python3-numpy` i `python3-scipy` albo `python-numpy` i `python-scipy`.

[1] Oficjalna strona projektu znajduje się pod adresem `https://winpython.github.io`

Pozostałe potrzebne biblioteki można doinstalować np. poleceniem `pip3` (lub `pip` – patrz wcześniejsze uwagi), przechodząc wcześniej w konsoli do lokalizacji, do której skopiowano pliki dołączone do niniejszej pracy:

```
pip3 install -r requirements.txt
```

A.2.2 Windows z WinPython

Aby uruchomić konsolę systemową wraz z dostępem do Pythona należy uruchomić program `WinPython Command Prompt.exe` z lokalizacji, w której zainstalowany został WinPython.

Pozostałe potrzebne biblioteki można doinstalować np. poleceniem `pip`, przechodząc wcześniej w konsoli do lokalizacji, do której skopiowano pliki dołączone do niniejszej pracy:

```
pip install -r requirements.txt
```

Dodatek B

Opis zawartości płyty CD

Na płycie dołączonej do pracy znajdują się następujące foldery i pliki:

jmlokosiewicz_inz2016.pdf – praca inżynierska w formacie PDF,

latex – folder ze źródłami pracy w systemie składu LaTeX,

connect4 – folder ze źródłami implementacji opisywanej w rozdziale 5.