Nauka przez wzmocnienia

Poznaliśmy dwa podstawowe typy technik uczenia maszynowego: uczenie nadzorowane i uczenie bez nadzoru. W przypadku nadzorowanego uczenia model jest szkolony na podstawie danych historycznych (obserwacji) w celu przewidywania wyników na podstawie nowych danych wejściowych. W przypadku uczenia się bez nadzoru model próbuje uzyskać wzorce w zestawach danych i zdefiniować granice grupowania logicznego w celu oddzielenia przestrzeni rozwiązania. Istnieje trzeci typ algorytmu uczenia maszynowego, który jest równie ważny dla ewolucji sztucznej inteligencji. Pamiętaj o procesie nauki jazdy na rowerze. Obserwujemy inną osobę, która jedzie na rowerze, tworzymy model mentalny, jak to zrobić i sami próbujemy. Nie jest możliwe uzyskać równowagę i ruch na rowerze już za pierwszym razem. My (aktor) próbujemy po raz pierwszy (akcja) na drodze (środowisko) i możemy upaść (nagroda). Staramy się w kółko z różną równowagą po lewej i prawej stronie, z różną prędkością i strategią pedałowania, a tym razem możemy pokonać większy dystans (wyższa nagroda) i ostatecznie uzyskać prawidłową jazdę rowerem (cel!). Proces ten, wielokrotnie powtarzany, wzmacnia właściwy zestaw działań w oparciu o warunki środowiskowe w danym momencie w aby zmaksymalizować nagrodę. Proces, który właśnie wizualizowaliśmy, nazywa się uczeniem się przez wzmocnienie. Jest to trzecia podstawowa kategoria algorytmów uczenia maszynowego, którą zamierzamy w tym zakresie zbadać. W tej części zrozumiemy:

* Koncepcję algorytmów uczenia się przez wzmocnienie

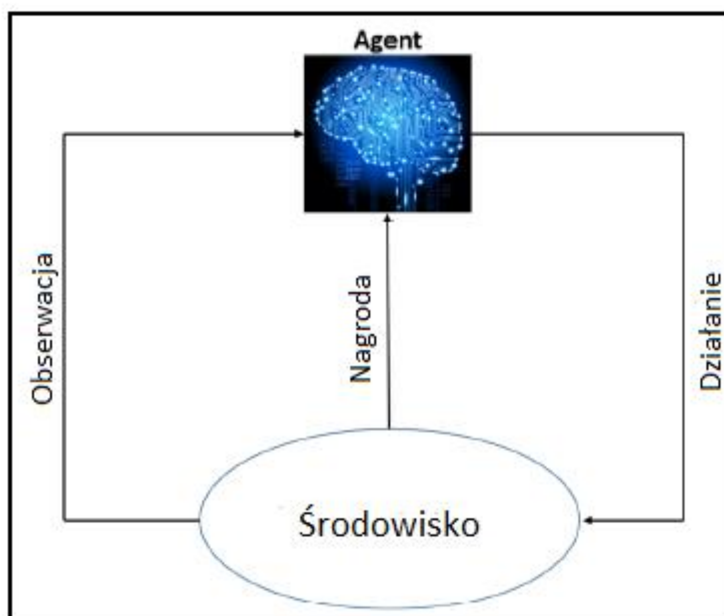
* Q-learning

* Naukę SARSA

* Naukę głębokiego wzmocnienia

Koncepcja algorytmów uczenia się przez wzmocnienie

Stwórzmy uproszczony model uczenia się przez wzmocnienie ze wstępem podstawowych terminologii:



Na każdym etapie i czasie (t) agent:

*Wykonuje akcję a_t

*Otrzymuje obserwację ot

*Odbiera nagrodę r_t

Na każdym etapie i czasie (t) środowisko:

*Otrzymuje działanie a_t

*Generuje obserwację ot + 1

*Generuje nagrodę skalarną r_{t+1}

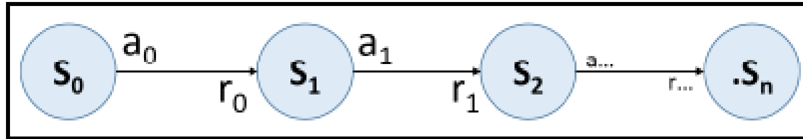
Środowisko jest uważane za niedeterministyczne (akcja a_t na podstawie o_t otrzyma nagrodę r_t, a ta sama akcja w tym samym stanie może skutkować różnymi nagrodami).

Agent (inteligentna maszyna) jest związany z kontekstem środowiskowym poprzez jego obserwację i działanie. Agent postrzega środowisko w sposób unikalny dla siebie i decyduje o działaniu w oparciu o niektóre popularne i ewoluujące techniki. Na każdym etapie agent odbiera sygnały reprezentujące stan środowiska. Agent odpowiada działaniem, które jest jedną z kilku możliwych opcji w tym momencie. Akcja generuje dane wyjściowe, które zmieniają stan środowiska. Pamiętasz piramidę wyników z pierwszego rozdziału? Jeśli agent potrzebuje lepszych wyników, musi podjąć właściwe działania w oparciu o środowisko i ogólny cel swojego istnienia. Zmiana stanu środowiska spowodowana działaniem agenta jest przesyłana z powrotem do agenta za pomocą sygnału wzmacniającego r. Ogólny wynik jest kombinacją dyskretnych działań, które agent musi wybrać, aby zmaksymalizować lub zwiększyć sumę nagrody (sygnał wzmocnienia). Uczymy się tego przez pewien czas na podstawie strategii prób i błędów obsługiwanych przez niektóre algorytmy ewolucyjne. Na tym tle wyraźnie widać, że istnieją dwa odrębne sposoby osiągnięcia uczenia wzmacniającego:

* Użyj algorytmu genetycznego i programowania: W tym podejściu agent szuka w obrębie możliwych ścieżek do optymalnego rozwiązania lub działania w oparciu o kontekst środowiskowy. Podczas gdy zastosowanie modelu algorytmu genetycznego ma tendencję do eliminowania zależności od brutalnej siły w osiąganiu ogólnego celu agenta, jakim jest maksymalizacja nagród, to podejście to w pewnym momencie przynosi suboptymalizowane działania dla agenta.

* Używaj technik statystycznych i modelu programowania dynamicznego: jest to podejście przyjęte przez nowoczesny paradygmat obliczeniowy obliczeń rozproszonych i przetwarzania równoległego w rozwoju agentów, które przewyższają ludzką inteligencję w niektórych trudnych zadaniach (gry takie jak Chess and Go).

Istnieje zasadnicza różnica między uczeniem się przez wzmacnianie a modelami uczenia się nadzorowanego. W przypadku uczenia nadzorowanego mamy dostęp do danych historycznych, które mapują zmienne niezależne na zmienne wyjściowe. Te dane historyczne są wykorzystywane jako dane wejściowe do szkolenia nadzorowanego modelu uczenia się. Model jest następnie w stanie przewidzieć wartość wyjściową dla nowego zestawu wejściowych zestawów danych. W przypadku uczenia się przez wzmocnienie agent musi przeszukiwać dostępną przestrzeń rozwiązania i nie ma dostępu do historycznego zestawu działań, które przyniosły maksymalną nagrodę. Hybrydowe podejście, w którym punktem wyjścia dla agenta jest wyszkolony model, który eliminuje część przestrzeni wyszukiwania, a agent może osiągnąć cel (maksymalizacja nagrody) dla zestawu przejść środowiskowych w bardziej zoptymalizowany sposób, wydaje się być preferowanym inteligencja maszyn budowlanych. Przejście stanu dla środowiska na podstawie działań agenta można wizualizować w następujący sposób:



Ogólnym celem algorytmu uczenia się przez wzmocnienie jest ustalenie zasady P, aby zmaksymalizować sumę nagrody za wszystkie połączone działania:

$$\mathbb{P} = Max(\sum_{t=0}^{n} r(t))$$

Istnieją dwa podstawowe strategie, które algorytm musi zastosować do uczenia wzmacniającego. Wyobraź sobie naukę wspomagania jako nawigację w labiryncie, w którym po drodze dostajemy pozytywne i negatywne nagrody. Wyznaczamy zasady nawigacji z eksploracją i prześledzimy ścieżkę wstecz, jeśli nagrody zostaną zmniejszone w wyniku wielu akcji. Ta technika nazywa się eksploracją z naciskiem na nagrody. Jednak po prostu podążając ścieżką maksymalnej nagrody w ramach ograniczonej widoczności do labiryntu, nie możemy osiągnąć stanu końcowego znalezienia optymalnej ścieżki. Musimy losowo wykorzystywać nieznane terytoria, aby zapuszczać się w nowe kierunki. Jest to formalnie określane jako wykorzystanie przestrzeni wyszukiwania. Połączenie etapów eksploracji i eksploatacji prowadzi do ogólnego celu uczenia się wzmacniającego. Chociaż agent stosuje eksplorację i eksploatację, aby osiągnąć ogólny cel maksymalizacji nagród, należy wziąć pod uwagę optymalne zachowanie. Istnieją trzy różne tryby, w których agent może zoptymalizować wyszukiwanie w przestrzeni rozwiązań w widocznym środowisku:

* Model skończonego horyzontu: w dowolnym momencie agent nie może mieć widoczności całej przestrzeni wyszukiwania. Agent przerywa poszukiwanie maksymalnej nagrody za kolejne m kroków:

$$\mathbb{P} = Max(\sum_{t=0}^{m} r(t))$$

Agent nie martwi się o kolejne etapy w przyszłości. W tym podejściu agent ma niestacjonarne zasady, które mogą ulec zmianie w zależności od napotkanego środowiska. W tym momencie agent podejmuje m-step optymalną akcję, która jest najlepszą sekwencją akcji dla m kroków w zbrojeniu. W następnym kroku agent optymalizuje kroki m-1 i tak dalej do końca ograniczonej przestrzeni wyszukiwania.

* Model nieskończonego horyzontu: zauważalną różnicą w tym modelu jest to, że przestrzeń wyszukiwania i przejścia stanu są uważane za nieskończone. Model jest szkolony z myślą o maksymalizacji długoterminowej nagrody w całej przestrzeni wyszukiwania. Nagrody są dyskontowane w proporcji geometrycznej według współczynnika dyskontowego β z zakresem wartości od 0 do 1:

$$\mathbb{P} = Max(\sum_{t=0}^{\infty} \beta^t r(t))$$

Agent nie martwi się o kolejne etapy w przyszłości. W tym podejściu agent ma niestacjonarne zasady, które mogą ulec zmianie w zależności od napotkanego środowiska. W tym momencie agent podejmuje m-step optymalną akcję, która jest najlepszą sekwencją akcji dla m kroków w zbrojeniu. W następnym kroku agent optymalizuje kroki m-1 i tak dalej do końca ograniczonej przestrzeni wyszukiwania.

* Model nieskończonego horyzontu: zauważalną różnicą w tym modelu jest to, że przestrzeń wyszukiwania i przejścia stanu są uważane za nieskończone. Model jest szkolony z myślą o maksymalizacji długoterminowej nagrody w całej przestrzeni wyszukiwania. Nagrody są dyskontowane w proporcji geometrycznej według współczynnika dyskontowego β z zakresem wartości od 0 do 1:

* Model średniej nagrody: w tym przypadku agent podejmuje działania na podstawie optymalnej wartości średniej nagrody na poszczególnych etapach akcji. Jest to ograniczający przypadek modelu nieskończonego horyzontu i jest uważany za bardziej konserwatywny pod względem dygresji do niezoptymalizowanego rozwiązania w międzyczasie.

Gdy algorytm jest zgodny z jednym z modeli, wydajność jest mierzona za pomocą trzech podstawowych kryteriów:

* Powolna i ostateczna konwergencja do optymalnej: Agent, który powoli inicjuje naukę i ostatecznie osiąga optymalny stan z krokami działania zapewniającymi maksymalną nagrodę, jest mniej preferowany w porównaniu z tymi, które szybko osiągają 90% optymalnego zachowania.

* Miara prędkości konwergencji do optymalnej: Ponieważ stan optymalny jest niepewny, prędkość konwergencji musi być miarą względną i subiektywną oraz funkcją dopuszczalnej różnicy od globalnych optymów lub bliskiej optymalności. Możemy również zmierzyć poziom wydajności po określonym czasie lub czynności. Zazwyczaj jest czas, w którym błędy nie występują i stąd minimalny czas musi zostać starannie wybrany w kontekście środowiska, w którym agent działa. Czasami staje się nieodpowiednim środkiem, jeśli agent działa w środowisku przez dłuższy czas. Możliwe jest również, że agent płaci wysoką karę w całym okresie nauki. Za pomocą tej miary można wybrać model, który szybko zbliża się do progowej wydajności i dokładności.

Techniki uczenia się przez wzocnienie

Mając to doświadczenie w uczeniu się przez wzmacnianie, w kilku następnych sekcjach przyjrzymy się niektórym z formalnych technik eksploracji przestrzeni poszukiwań w celu maksymalizacji nagród w optymalny sposób.

Procesy decyzyjne Markowa

Aby zrozumieć procesy decyzyjne Markowa (MDP), zdefiniujmy dwa typy środowiska:

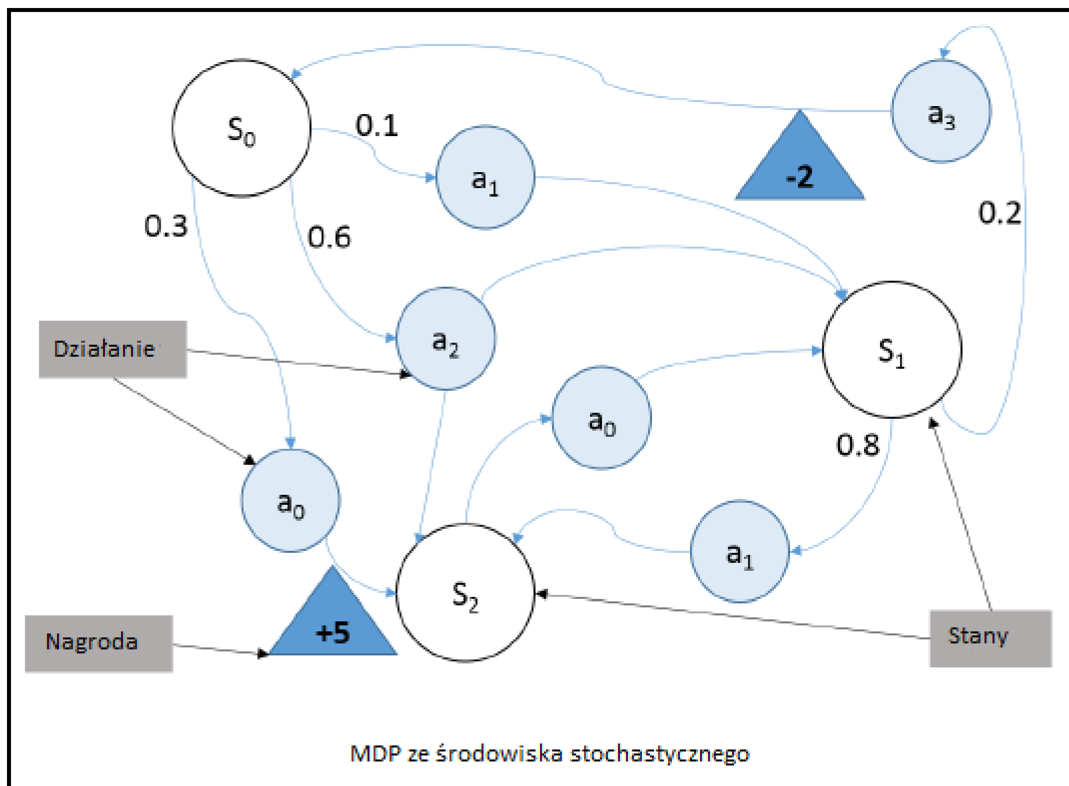
* Środowisko deterministyczne: w środowisku deterministycznym działanie podejmowane w określonym stanie środowiska determinuje określony wynik. Na przykład, w grze w szachy ze wszystkich możliwych ruchów na początku gry, kiedy przenosimy pionek z e4 na e5, natychmiastowy następny krok jest pewny i nie różni się w zależności od gry. Istnieje także poziom pewności nagrody w deterministycznym środowisku wraz z następnym możliwym stanem (stanami).

* Środowisko stochastyczne: w przypadku środowiska stochastycznego zawsze występuje poziom losowości i niepewności w odniesieniu do następnego stanu środowiska w oparciu o działanie agenta w poprzednim stanie.

Jak można wyczuć, większość rzeczywistych środowisk, w których agenci będą uczestniczyć, gdy buduje się inteligentne systemy, będzie miała charakter stochastyczny. MDP zapewniają ramy, które ułatwiają

podejmowanie decyzji w środowisku stochastycznym, a ogólnym celem agenta jest znalezienie polityki pozwalającej osiągnąć ostateczny zamierzony stan w oparciu o szereg działań w tym kontekście. MDP odbiegają od prostego planowania w tym sensie, że działania są ustalane i dostosowywane na podstawie warunków środowiskowych. MDP zapewniają formalny model kwantyfikacji dla procesu decyzyjnego dla agenta w środowisku stochastycznym. Agent wykonuje krok (działanie a) ze zbioru wszystkich dostępnych działań w czasie t w swoim bieżącym stanie s. Środowisko przenosi się do nowego stanu s 'w drodze, dając nagrodę agentowi Ra (s, s). Ze względu na stochastyczny charakter środowiska nie można z pewnością zagwarantować przejścia ze stanu do stanu określonego. To przejście jest możliwe przy wartości prawdopodobieństwa Pa (s, s '). Każdy krok akcji w obrębie stanu s jest niezależny od poprzednich stanów i akcji i spełnia własność Markowa. Proces stochastyczny ma właściwość Markowa, jeśli warunkowy rozkład prawdopodobieństwa przyszłych stanów procesu (zależny zarówno od stanów przeszłych, jak i obecnych) zależy tylko od stanu obecnego, a nie od sekwencji zdarzeń, które go poprzedzały. Proces z tą właściwością nazywa się procesem Markowa.

Stochastyczną naturę środowiska z przejściami stanu wynikającymi z szeregu działań można przedstawić w następujący sposób:



MDP ze środowiska stochastycznego

MDP składa się z pięciu podstawowych elementów:

* S: Zestaw wszystkich możliwych stanów środowiska.

* O: Zestaw wszystkich możliwych działań agenta. As reprezentuje zestaw możliwych działań w stanie s.

* $P_a(s, s')$: Prawdopodobieństwo, że akcja w stanie s prowadzi do stanu s'. Na poprzednim schemacie istnieje prawdopodobieństwo 0,6, że działanie a_2 w stanie s_1 spowoduje przejście stanu środowiska do s_1.

* $R_a(s, s')$: Reprezentuje nagrodę w wyniku działania a, gdy środowisko przechodzi ze stanu s do s w wyniku działania a. Na poprzednim schemacie agent otrzymuje nagrodę -2 za działanie a3 w stanie s_1 i przejście do stanu s0x`.

* ϒ ∈ [0,1]: Jest to współczynnik dyskontowy, który stanowi różnicę między przyszłymi nagrodami a bieżącą nagrodą za zmianę stanu na podstawie określonego działania.

MDP próbuje znaleźć politykę, która maksymalizuje skumulowaną nagrodę za wszystkie działania w skończonym zbiorze stanów. Cel można osiągnąć za pomocą dynamicznych ram programistycznych.

Dynamiczne programowanie i uczenie się przez wzmocnienie

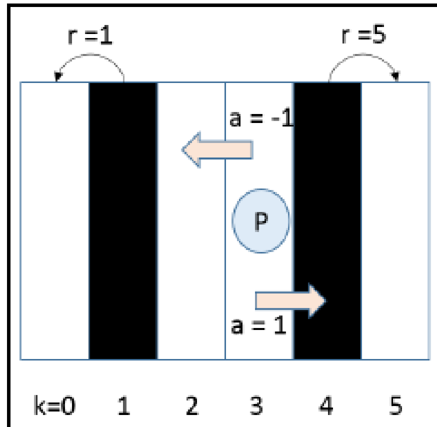
W kontekście uczenia się przez wzmacnianie podejście dynamiczne zajmuje się interakcjami między kontrolerem lub agentem, które muszą podjąć działania, a procesem w środowisku. Ta interakcja odbywa się za pomocą trzech rodzajów wyraźnych sygnałów:

* Sygnał stanu: opisuje stan procesu

* Sygnał akcji: Dzięki temu agent (kontroler) wpływa na proces

* Sygnał nagrody: Dostarcza informację zwrotną do kontrolera na podstawie jego ostatniej akcji

Agent porusza się po przestrzeni rozwiązania z powtarzalnymi iteracjami cyklu stan-akcja-stan-stan. Polityka określa ogólne zachowanie agenta. Polityka może być dynamicznie dostosowywana w zależności od charakteru środowiska (deterministycznego lub stochastycznego). W przypadku programowania dynamicznego ogólnym celem agenta jest znalezienie optymalnej polityki, która maksymalizuje skumulowaną nagrodę (zwrot) w trakcie istnienia agenta. Rozważymy powrót ponad horyzont nieskończony, co prowadzi do stacjonarnej optymalnej polityki, w której dla danego stanu wybór optymalnych działań będzie zawsze taki sam. Podczas gdy DP i RL mają ten sam cel w nieskończonym horyzoncie, istnieją między nimi pewne różnice pod względem zastosowań i algorytmów. DP i RL stosują wspólne strategie iteracyjne, takie jak iteracja wartości, iteracja polityki, a także polityki wyszukiwania w celu osiągnięcia celu optymalizacji. Rozważmy najpierw algorytmy DP i RL w kontekście deterministycznego otoczenia. W tym środowisku, gdy agent podejmuje działanie a_t w stanie s_t w kroku czasu t, stan zmienia się na $s_t + 1$ zgodnie z funkcją przejścia $f: S \times A \rightarrow S$, tak że $s_{t+1} = f(s_t, w)$. W tym czasie agent odbiera skalarny sygnał nagrody r_{t+1} zgodnie z funkcją nagrody $\rho: S \times A \rightarrow \mathbb{R}$ tak, że $r_{t+1} = \rho(s_t, a_t)$. Agent wybiera dalsze akcje zgodnie z polityką $\pi: S \rightarrow A$, używając at = $\pi(s_t)$. Gdy znana jest funkcja przejścia f, funkcja nagrody ρ, aktualny stan st i bieżące działanie w, można określić następny stan s_{t+1} i następną nagrodę r_{t+1}. Uczenie się w środowisku deterministycznym z iteracją polityki Pozwól nam zrozumieć proces uczenia się agenta oparty na modelu programowania dynamicznego w środowisku deterministycznym przedstawionym na poniższym diagramie. Wyobraźmy sobie agenta, który uczy się odtwarzać muzykę na prostej klawiaturze:



Na tym schemacie P oznacza agenta grającego na klawiaturze, a K {0,1,2,3,4,5} oznacza klawisze ponumerowane od 0 do 5. W tej prostej konfiguracji agent może poruszać się do przodu i do tyłu reprezentowany przez A { -1,1}. Przejście na prawą stronę oznacza a = 1, a lewa strona a = -1. Załóżmy, że agent otrzymuje nagrodę za zagranie określonej nuty i w tym przypadku, gdy przechodzi z klawisza nr 4 do 5, nagrodą jest 5, a gdy przechodzi z klawisza 1 do klawisza 0, nagrodą jest 1. Dla wszystkich inne przejścia, nagroda wynosi 0. Załóżmy dla uproszczenia, że klucze 0 i 5 to stany końcowe dla nuty dźwiękowej, a gdy agent tam dotrze, nie może odejść. W takim przypadku funkcję przejścia można przedstawić następująco:

$$f(k,a) = k + a \quad \text{jeśli } 1 \leq k < 4$$

$$f(k,a) = k \quad \text{jeśli } k = 0 \text{ lub } k = 5$$

Funkcja nagrody jest reprezentowana w następujący sposób:

$$\rho(s,a) = \begin{cases} 5 & \text{Jeśli } s = 4 \text{ i } a = 1 \\ 1 & \text{Jeśli } s = 1 \text{ i } a = -1 \\ 0 & \text{Dla innych stanów-przejścia} \end{cases}$$

W tym kontekście celem agenta jest uzyskanie najwyższej skumulowanej nagrody w oparciu o przejścia na klawiaturze w oparciu o dowolną pozycję początkową k_x. Nagroda za nieskończony horyzont jest sformułowana w następujący sposób:

$$R(k_x) = \sum_{k=0}^{\infty} \gamma^k r_{(}k+1_{)} = \sum_{k=0}^{\infty} \gamma^k \rho(s_k, f(x_k))$$

W takim przypadku $\gamma \in [0, 1]$ współczynnik dyskontowy reprezentuje opóźnione przyjęcie wdzięczności dla agenta w odniesieniu do nagród. Dzięki temu łączna nagroda jest ograniczona, jeśli nagrody za poszczególne działania są ograniczone. Agent wykorzystuje tylko informacje zwrotne z każdego kroku akcji w celu maksymalizacji ogólnej skumulowanej nagrody. Bieżący krok akcji w tym przypadku nie zapewnia żadnej wskazówki na temat ogólnej nagrody dla agenta. Konieczne jest wybranie odpowiedniej wartości dla γ która określa kompromis między jakością rozwiązania w maksymalizacji nagrody a współczynnikiem konwergencji. Aby uzyskać optymalną politykę dla agenta, używane są funkcje wartości. Istnieją dwa typy funkcji wartości oznaczone jako funkcje Q i funkcje V. Funkcje Q są funkcjami wartości akcji stanu, a funkcje V są funkcjami wartości stanu.

Funkcja Q Q^p: S x A → ℝ zasady ℙ daje zwrot, rozpoczynając od danego stanu i podanej akcji a i zgodnie z zasadą ℙ. W rezultacie $Q^p(s,a) = \bar{\rho}(s,a) + \gamma\bar{\mathbb{R}}^p(f(s,a))$. Oto $\bar{\mathbb{R}}^p(f(s,a)$ powrót z następnego kroku f(s,a). Funkcja Q może być również reprezentowana jako zdyskontowana suma nagród poprzez wzięcie ws, a następnie przestrzeganie zasady ℙ:

$$Q^p(s,a) = \sum_{k=0}^{\infty} \gamma^k \rho(s_k, a_k)$$

Gdy $(s_0, a_0) = (s, a)$, $s_{k+1} = f(s_k, a_k)$dla k = 0 i $a_k = \mathbb{P}(s_k)$dla k ≥ 1, pierwszy człon można oddzielić od funkcji wartości skumulowanej.

$$Q^p(s,a) = \mathbb{P}(s,a) + \sum_{k=1}^{\infty} \gamma^k \mathbb{P}(s_k, a_k)$$

$$= \mathbb{P}(s,a) + \gamma \sum_{k=1}^{\infty} \gamma^{k-1} \mathbb{P}(s_k, \mathbb{P}(s_k))$$

$$= \mathbb{P}(s,a) + \gamma \mathbb{R}^p(f(s,a))$$

Optymalna funkcja Q to taka, która daje maksymalną wartość Q przy różnych przejściach agenta do przestrzeni wyszukiwania

$$Q^*(s,a) = max_p Q^p(s,a)$$

Funkcję V V^p: S → ℝ zasady p uzyskuje się, zaczynając od określonego klucza i wykonując p. Tę funkcję V można wyprowadzić z funkcji Q polityki p:

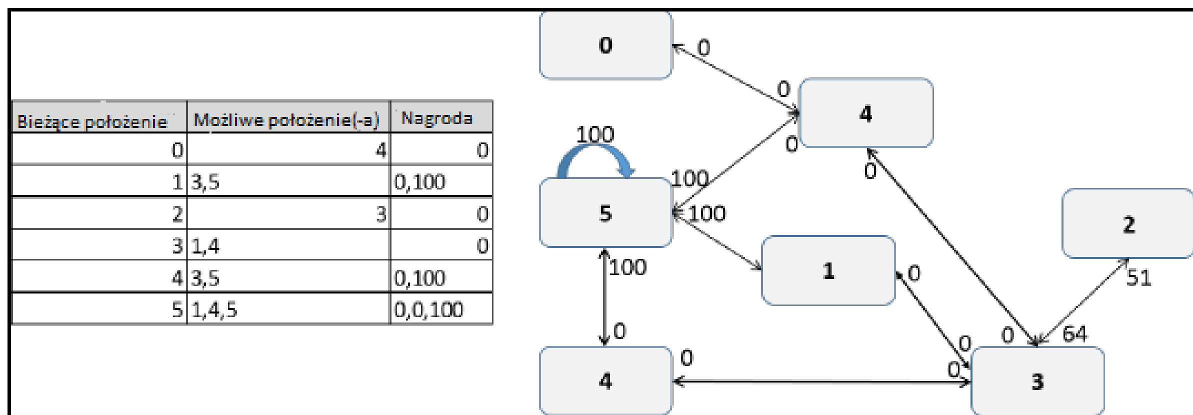
$$V^p(s) = \bar{R}^p(s) = Q^p(s, \rho(s))$$

Ponownie, optymalna funkcja V to taka, która daje maksymalną wartość V dla różnych przejść i może być obliczona na podstawie optymalnej funkcji Q. Ucząc się w środowisku stochastycznym, agent nie może przejść do stanu s + 1 z pewnością, gdy podejmie działanie a + 1. W takim przypadku wartość Q i wartość V uzyskuje się jako a prawdopodobieństwo przejścia wyuczonego przez agenta w wielu

iteracjach w przestrzeni wyszukiwania. W następnej części zajmiemy się jednym z popularnych algorytmów Q-uczenia się bez modeli.

Q-Learning

Q-learning jest algorytmem uczenia się bez modelu, który jest użyteczny w sytuacjach, gdy agent zna wszystkie możliwe stany i działania, które prowadzą do tych stanów w przestrzeni wyszukiwania. Q-learning ma możliwość wyboru między nagrodą natychmiastową a nagrodą długoterminową, co umożliwia optymalizację w celu osiągnięcia celu maksymalizacji nagród zgromadzonych w ramach zestawu działań. Wyjaśnijmy to na prostym przykładzie. Zastanów się nad labiryntem z sześcioma lokalizacjami (L∈ {0,1,2,3,4,5}), a gdy agent dojdzie do lokalizacji numer 5, znajdzie skarb (stan drugi lub cel agenta). Labirynt ma następującą strukturę. Dwukierunkowe strzałki wskazują możliwe przejścia stanu, a liczby oznaczają nagrodę:



Bieżące położenie	Możliwe położenie(-a)	Nagroda
0	4	0
1	3,5	0,100
2	3	0
3	1,4	0
4	3,5	0,100
5	1,4,5	0,0,100

Przejścia stanu są reprezentowane w znormalizowany sposób w Q-learningu jako macierz, w której wiersze wskazują stan, a kolumny wskazują działania. -1 oznacza, że działanie a nie jest możliwe lub zablokowane w określonym stanie, 100 oznacza nagrodę 100 punktów za przejście stanu. Za wszystkie pozostałe przejścia nagrodą jest 0.

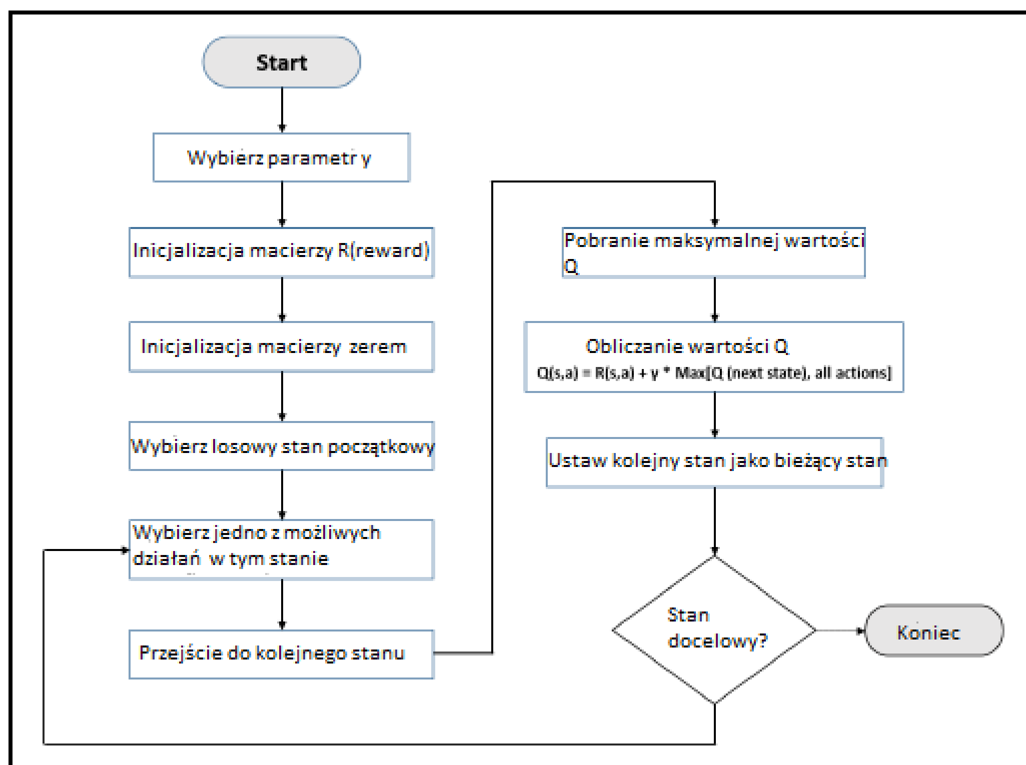
$$
\begin{array}{c}
\text{Działanie} \\
\text{Stan} \quad
\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array}
\end{array}
$$

$$
R = \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}
\begin{bmatrix}
-1 & -1 & -1 & -1 & 0 & -1 \\
-1 & -1 & -1 & 0 & -1 & 100 \\
-1 & -1 & -1 & 0 & -1 & -1 \\
-1 & 0 & 0 & -1 & 0 & -1 \\
0 & -1 & -1 & 0 & -1 & 100 \\
-1 & 0 & -1 & -1 & 0 & 100
\end{bmatrix}
$$

Agent musi teraz zbudować macierz Q, która przechowuje całą naukę, jaką wykonuje agent z serią akcji i odpowiadającymi im przejściami stanu. W macierzy Q wiersze reprezentują bieżący stan, a kolumny przedstawiają możliwe działania, które prowadzą do następnego etapu. Początkowy stan macierzy Q występuje wtedy, gdy agent nie wie nic o środowisku, a zatem macierz zawiera wszystkie wartości zerowe. W naszym przykładzie załóżmy, że agent ma świadomość, że istnieje sześć możliwych stanów środowiska. Jednak w prawdziwych scenariuszach agent nie będzie miał wiedzy o wszystkich stanach i musi zbadać przestrzeń wyszukiwania. W takim przypadku algorytm uczenia Q dodaje kolumny do macierzy Q, gdy pojawi się nowy stan. Reguła przejścia dla uczenia się Q jest reprezentowana jako

$$Q(s,a) = R(s,a) + \gamma * Max[Q(s+1, a_{0,n})]$$

Wartość przypisana w macierzy Q reprezentuje sumę odpowiednich wartości w macierzy R i parametru uczenia pomnożoną przez maksymalną wartość Q dla wszystkich możliwych działań w następnym stanie. Gdy agent przechodzi od pozycji początkowej do stanu celu, aktualizuje macierz Q, a przejście to nazywa się jednym odcinkiem. W tym kontekście algorytm uczenia Q jest reprezentowany w następujący sposób:



Dzięki temu algorytmowi pamięć agenta jest wzbogacana z każdym odcinkiem i przechowuje więcej informacji o nagrodach za przejście do stanu. Po przeszkoleniu w rozsądnej liczbie odcinków agent może szybko uzyskać optymalną ścieżkę w przestrzeni wyszukiwania. Zakres parametru γ wynosi od 0 do 1. Gdy parametr γ n jest bliższy zeru, agent nadaje pierwszeństwo nagrodom podczas początkowych odcinków. Gdy wartość jest bliższa 1, agent rozważa przyszłe nagrody o większej wadze, skłonne opóźnić nagrodę w interesie skumulowanego zysku. Użyjmy algorytmu dla kilku odcinków opartych na przykładzie labiryntu, który widzieliśmy wcześniej. Oto stan początkowy macierzy nagród R i macierzy Q Q:

Działanie

$$R = \begin{array}{c|} \text{Stan} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array} \qquad Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Stan Początkowy dla macierzy Reward i Q

Rozważmy, że stanem początkowym agenta jest 1 i używamy arbitralnej wartości γ równej 0,8. Jak wiemy, ze stanu 1 możliwe stany, do których agent może przejść, to 3 i 5, i w tym momencie rozważmy, że agent losowo przechodzi do stanu 5. Na etapie 5 agent ma trzy możliwe opcje wyboru stanu: 1 , 4 i 5. Zastosujmy równanie uczenia się Q:

$$Q(s,a) = R(s,a) + \gamma * Max[Q(s+1, a_{0,n})]$$

$$Q(1,5) = R(1,5) + 0.8 * Max[Q(5,1), Q(5,4), Q(5,5)]$$

$$= 100 + 0.8 * Max[0,0,0]$$

$$= 100 + 0.8 * 0 \text{ (Remember the Q matrix is initialized to a zero)}$$

$$= 100$$

Ponieważ 5 jest stanem docelowym, zakończyliśmy jeden odcinek nową wersją macierzy Q w następujący sposób:

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Macierz Q po wydarzeniu 1

W kolejnym odcinku agent przechodzi do stanu początkowego 3. Odwołaj się do macierzy R, na etapie 3 istnieją trzy możliwe działania: 1, 2 i 4. Agent decyduje się na działanie 1, które powoduje, że ląduje w stan 1. Teraz wyobraź sobie, że agent jest w stanie 1. W tym momencie agent może przejść do stanów 3 i 5. Obliczmy wartość Q dla tej trasy:

$$Q(s,a) = R(s,a) + \gamma * Max[Q(s+1, a_{0,n})]$$

$$Q(3,1) = R(3,1) + 0.8 * Max[Q(1,3), Q(1,5)]$$

$$= 0 + 0.8 * Max[0,100]$$

$$= 0 + 0.8 * 100$$

$$= 80$$

W tym momencie agent znajduje się w stanie 1, który nie jest stanem końcowym ani docelowym, a zatem pętla przechodzi do stanu docelowego (w tym przypadku 5). Załóżmy, że agent losowo przechodzi do stanu 5 ze stanu 1, który jest stanem celu, a zatem epizod 2 zostaje zakończony. Oto macierz Q na końcu odcinka 2:

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 80 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \Rightarrow Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 400 & 0 \\ 0 & 0 & 0 & 320 & 0 & 500 \\ 0 & 0 & 0 & 320 & 0 & 0 \\ 0 & 400 & 256 & 0 & 400 & 0 \\ 320 & 0 & 0 & 320 & 0 & 500 \\ 0 & 400 & 0 & 0 & 400 & 500 \end{bmatrix} \end{array}$$

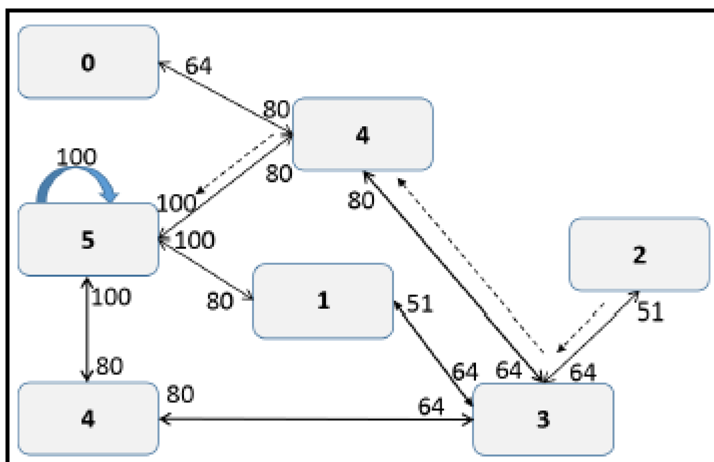
Macierz Q po wydarzeniu 1 | Macierz Q po kowergencji

Macierz można skalować, dzieląc wszystkie niezerowe liczby przez liczbę maksymalną i mnożąc przez 100. W przypadku normalizacji końcowa zbieżna macierz Q wygląda następująco:

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 80 & 0 \\ 0 & 0 & 0 & 64 & 0 & 100 \\ 0 & 0 & 0 & 64 & 0 & 0 \\ 0 & 80 & 51 & 0 & 80 & 0 \\ 64 & 0 & 0 & 64 & 0 & 100 \\ 0 & 80 & 0 & 0 & 80 & 100 \end{bmatrix} \end{array}$$

Znormalizowana macierz Q

Po uzyskaniu zbieżnej i znormalizowanej macierzy Q agent zapamiętał i nauczył się optymalnych działań dla przejść stanu w celu osiągnięcia stanu celu (w tym przypadku 5). Schemat przejścia stanu z wartościami macierzy Q wygląda następująco:



Po zdefiniowaniu tej macierzy przejścia agent może optymalnie nawigować w przestrzeni wyszukiwania, wybierając akcję na każdym kroku z maksymalną wartością Q, jak wskazano kropkowaną strzałką na poprzednim diagramie. Oto fragment kodu, który implementuje Algorytm Q-learning z tym samym przykładem, który widzieliśmy wcześniej:

```
package com.aibd.rl;

import java.util.Random;

public class QLearner {
```

```
private static final int STATE_COUNT = 6;
private static final double GAMMA = 0.8;
private static final int MAX_ITERATIONS = 10;
private static final int INITIAL_STATES[] = new int[] {1, 3, 5, 2,
4, 0};
// initialize the R matrix with the state transition combinations
private static final int R[][] =
new int[][] {{-1, -1, -1, -1, 0, -1},
{-1, -1, -1, 0, -1, 100},
{-1, -1, -1, 0, -1, -1},
{-1, 0, 0, -1, 0, -1},
{0, -1, -1, 0, -1, 100},
{-1, 0, -1, -1, 0, 100}};
private static int q[][] = new int[STATE_COUNT][STATE_COUNT];
private static int currentState = 0;
public static void main(String[] args) {
train();
test();
return;
}
private static void train() {
// initialize the Q matrix to zero values
initialize();
// Perform training, starting at all initial states.
for(int j = 0; j < MAX_ITERATIONS; j++){
for(int i = 0; i < STATE_COUNT; i++) {
episode(INITIAL_STATES[i]);
}
}
System.out.println("Q Matrix:");
for(int i = 0; i < STATE_COUNT; i++) {
```

```
for(int j = 0; j < STATE_COUNT; j++){
System.out.print(q[i][j] + ",\t");
}
System.out.print("\n");
}
System.out.print("\n");
return;
}
private static void test() {
// Perform tests, starting at all initial states.
System.out.println("Shortest routes from initial states:");
for(int i = 0; i < STATE_COUNT; i++) {
currentState = INITIAL_STATES[i];
int newState = 0;
do {
newState = maximum(currentState, true);
System.out.print(currentState + " --> ");
currentState = newState;
}while(currentState < 5);
System.out.print("5\n");
}
return;
}
private static void episode(final int initialState) {
currentState = initialState;
do {
chooseAnAction();
}while(currentState == 5);
for(int i = 0; i < STATE_COUNT; i++){
chooseAnAction();
}
```

```
return;
}
private static void chooseAnAction() {
int possibleAction = 0;
// Randomly choose a possible action connected to the current
state.
possibleAction = getRandomAction(STATE_COUNT);
if(R[currentState][possibleAction] >= 0){
q[currentState][possibleAction] = reward(possibleAction);
currentState = possibleAction;
}
return;
}
private static int getRandomAction(final int upperBound) {
int action = 0;
boolean choiceIsValid = false;
// Randomly choose a possible action connected to the current
state.
while(choiceIsValid == false) {
// Get a random value between 0(inclusive) and 6(exclusive).
action = new Random().nextInt(upperBound);
if(R[currentState][action] > -1){
choiceIsValid = true;
}
}
return action;
}
private static void initialize() {
for(int i = 0; i < STATE_COUNT; i++)
{
for(int j = 0; j < STATE_COUNT; j++)
```

```
{
q[i][j] = 0;
} // j
} // i
return;
}
private static int maximum(final int State, final boolean
ReturnIndexOnly) {
// If ReturnIndexOnly = True, the Q matrix index is returned.
// If ReturnIndexOnly = False, the Q matrix value is returned.
int winner = 0;
boolean foundNewWinner = false;
boolean done = false;
while(!done) {
foundNewWinner = false;
for(int i = 0; i < STATE_COUNT; i++)
{
if(i != winner){ // Avoid self-comparison.
if(q[State][i] > q[State][winner]){
winner = i;
foundNewWinner = true;
}
}
}
if(foundNewWinner == false){
done = true;
}
}
if(ReturnIndexOnly == true){
return winner;
}else{
```

```
return q[State][winner];
}
}
private static int reward(final int Action) {
return (int)(R[currentState][Action] + (GAMMA * maximum(Action,
false)));
}
```

Ten program generuje następujące dane wyjściowe:

```
Q Matrix:
0, 0, 0, 0, 396, 0,
0, 0, 0, 316, 0, 496,
0, 0, 0, 316, 0, 0,
0, 396, 252, 0, 396, 0,
316, 0, 0, 316, 0, 496,
0, 396, 0, 0, 396, 496,
Najkrótsze trasy ze stanów początkowych:
1 -> 5
3 -> 1 -> 5
5 -> 5
2 -> 3 -> 1 -> 5
4 -> 5
0 -> 4 -> 5
```

Jak widzieliśmy, Q-learning jest metodą optymalizacji nagród zdyskontowanych, generalnie czyniąc przyszłe nagrody mniej priorytetowymi w porównaniu do nagród krótkoterminowych. W następnym rozdziale przyjrzymy się odmianom algorytmów Q-learning zwanych uczeniem SARSA.

Nauka SARSA

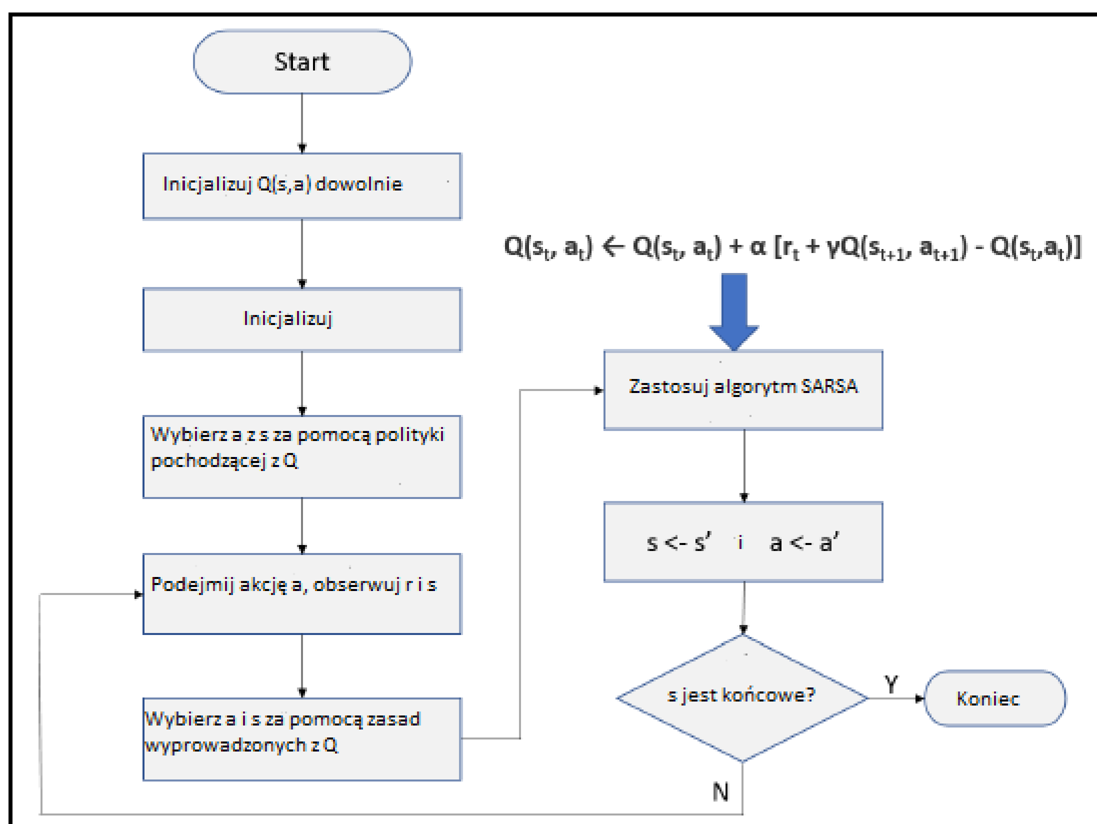
State-Action-Reward-State-Action (SARSA) to algorytm dotyczący polityki, w którym ta sama polityka, która wygenerowała poprzednie działania, może wygenerować następne działanie. Jest to inaczej niż w przypadku Qlearning, w którym algorytm jest niezgodny z polityką i uwzględnia jedynie bieżący stan i nagrody wraz z dostępnymi następnymi działaniami bez uwzględnienia bieżącej polityki.

Na każdym etapie SARSA działanie agenta jest oceniane i ulepszane przez poprawę oszacowań Qfunkcji. Wartość Q jest aktualizowana w wyniku błędu i korygowana o współczynnik uczenia się określany jako. W tym przypadku wartości Q reprezentują potencjalną nagrodę z następnej zmiany stanu w wyniku działania przy + 1 w stanie s plus zdyskontowaną (γ) przyszłą nagrodę otrzymaną z

następnej obserwacji stanu-działania. Algorytm można przedstawić matematycznie w następujący sposób:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \, [r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

Pierwsze odchylenie od uczenia się Q polega na tym, że w przypadku uczenia się przez SARSA agent uczy się funkcji wartości akcji, a nie funkcji wartości stanu. Agent musi oszacować Q^{ρ} (s, a) dla bieżącego dopasowania do polityki ℙ dla wszystkich stanów i działań a. Agent musi rozważyć przejście z jednej pary stan-akcja do innej pary stan-akcja i poznać wartość par para-akcja. Aktualizacje są wykonywane po przejściu ze st, gdzie jest to stan nieterminalny. Jeśli s_{t+1} jest stanem końcowym, to Q (s_{t+1}, a_{t+1}) jest zdefiniowane jako 0. SARSA wykorzystuje wszystkie elementy przy podejmowaniu decyzji (st, at, rt + 1, st + 1, przy at+ 1) na przejściu z jednego stanu do drugiego. Podobnie jak Q-learning, SARSA jest również algorytmem iteracyjnym, który można przedstawić w następujący sposób:



Inną metodą, która jest mniej popularnie stosowana w uczeniu się przez wzmacnianie, jest R-learning, który próbuje zoptymalizować średnie wynagrodzenie dla agenta. Przy podejmowaniu optymalnej polityki uwzględnia równorzędnie przyszłe i krótkoterminowe korzyści.

Nauka głębokiego wzmacniania

Aby algorytm uczenia wzmacniającego mógł zostać wdrożony w rzeczywistych przypadkach użycia i scenariuszach, musimy wykorzystać moc głębokich sieci neuronowych, które mogą wywnioskować informacje z otoczenia w sposób podobny do ludzkiego. Jednym z celów sztucznej inteligencji jest zwiększanie ludzkich możliwości, tworząc autonomiczne czynniki, które wchodzą w interakcje ze środowiskiem, w którym działają, uczą się optymalnych zachowań, które z czasem się poprawiają, i uczą się na błędach. Na przykład sygnały z kamery wideo można interpretować za pomocą głębokiej

sieci neuronowej. Po zinterpretowaniu tego sygnału obiekty i wzorce obserwowane przez kamerę można analizować za pomocą głębokiej sieci neuronowej, jak widzieliśmy w rozdziałach na temat sztucznych sieci neuronowych (ANN). Te głębokie sieci neuronowe mogą być następnie wykorzystane do zastosowania algorytmów uczenia wzmacniającego do stworzenia systemu nawigacyjnego, który uczy się przez pewien czas na podstawie danych treningowych. Zasadniczo połączenie głębokich sieci neuronowych i algorytmów uczenia wzmacniającego pozwala osiągnąć wydajność zbliżoną do ludzkiej z wykrywaniem obiektów, samojezdnymi samochodami, grami wideo, przetwarzaniem języka naturalnego i tak dalej. W tej sekcji dokonamy przeglądu różnych podejść i technik dla DRL. Jak wiemy, głębokie sieci neuronowe (DNN) mogą wywnioskować niskopłaszczyznowe reprezentacje wysokowymiarowych zestawów danych, takich jak sygnały audio / wideo. Z drugiej strony model uczenia wzmacniającego zmniejsza zależność od danych treningowych i opiera się na paradygmacie nagroda / kara dla agenta, aby poruszać się po stochastycznych środowiskach i doskonalić się przez pewien czas. Głębokie uczenie się umożliwia uczenie się przez wzmacnianie na nowy poziom w kierunku wydajności zbliżonej do ludzkiej, a w przypadku niektórych działań wymagających brutalnej siły autonomiczny agent jest w stanie przewyższyć ludzkie możliwości. Podczas gdy wiele pionierskich prac jest wykonywanych przy użyciu DRL, początkowy przełom został osiągnięty poprzez szkolenie algorytmu do opanowania gier wideo Atari 2600 i osiągnięcia nadludzkiego poziomu wiedzy po prostu poprzez podawanie danych pikselowych. Agent został przeszkolony wyłącznie na podstawie sygnału nagrody w połączeniu z mapą pikseli, która reprezentowała środowisko stochastyczne. Kolejnym znaczącym sukcesem był inteligentny agent AlphaGo, który pokonał mistrza świata Go w oparciu o wykorzystanie sieci neuronowych, które były szkolone przy użyciu kombinacji uczenia nadzorowanego i wzmacniającego oraz algorytmu samouczącego się. DRL okazały się przydatne w dziedzinie robotyki, w której sygnał wideo jest interpretowany za pomocą ANN, która aktywuje kontrolery wykonujące zadania o kluczowym znaczeniu, takie jak obsługa maszyn CNC, a także próby operacji. Nacisk polega na tworzeniu agentów, którzy potrafią meta-uczenia się, co oznacza naukę uczenia się. Jest to również możliwe w przypadku DRL. Konieczne jest, aby agenci DRL ewoluowali, aby w pełni uzupełnić ludzkie możliwości w najbliższej przyszłości. DRL odniosły sukces dzięki możliwości ekstrapolacji technik uczenia się w małych wymiarach na wielowymiarowe, nieustrukturyzowane zestawy danych. Sieci neuronowe są dobre w aproksymacji i uczeniu się w oparciu o dane wielowymiarowe. Dzięki temu DRL mogą poradzić sobie z przekleństwem wymiaru i treningiem modelu dla różnych środowisk stochastycznych w przestrzeni o dużych wymiarach. Splotowe sieci neuronowe (CNN) mogą być wykorzystywane jako elementy składowe DRL, co umożliwia uczenie się bezpośrednio z rzeczywistych zasobów surowych danych, które mają charakter wielkoformatowy. DRL umożliwiają szkolenie głębokiej sieci neuronowej w celu uzyskania optymalnej polityki poprzez przejścia stanu wraz z funkcjami optymalnej wartości, V, Q i A. Podczas gdy możliwości kombinacji sieci neuronowych i uczenia wzmacniającego są ogromne, ocenimy aplikację głębokich sieci neuronowych jako aproksymatory funkcji w wyszukiwaniu polityk metod w DRL. Jednym z najpopularniejszych algorytmów jest głęboka sieć Q (DQN).

Często Zadawane Pytania

P: Jaka jest różnica między uczeniem nadzorowanym a uczeniem się wzmacniającym?

O: W przypadku nadzorowanych algorytmów uczenia model jest szkolony w oparciu o dane historyczne, które historycznie opisują trend danych i ustanawiają korelację między danymi zdarzenia a wynikowym wynikiem. W takim przypadku nadzorowany model uczenia się jest ćwiczeniem polegającym na dopasowywaniu krzywych, które odwzorowuje punkty danych (zmienne niezależne) na zbiór zmiennych wyjściowych (zmienne zależne). Dostępność danych historycznych jest niezbędna do nadzorowanego uczenia się. W przypadku uczenia się przez wzmocnienie agent jest modelowany

na podstawie nagród, które otrzymuje i odbiera na podstawie działań podejmowanych w kontekście środowiska, w którym działa. Agent nie ma dostępnych danych historycznych do samodzielnego szkolenia. Jednak podejście hybrydowe często działa świetnie, gdy agent jest świadomy trendów historycznych, a także stosuje strategie eksploracji i eksploracji, aby zmaksymalizować nagrodę, gdy przechodzi przez przestrzeń poszukiwań do celu. P: Jakie są podstawowe elementy uczenia się przez zbrojenie?

O : Nauka wzmacniania odbywa się w kontekście środowiska. Środowisko definiuje wszystkie czynniki zewnętrzne wpływające na wydajność agenta, który jest drugim składnikiem RL. Agent przechodzi przez przestrzeń rozwiązania w wielu stanach, aby osiągnąć stan końcowy i zmaksymalizować nagrodę po drodze. Każda czynność, którą wykonuje agent, generuje nagrodę lub karę i powoduje odpowiednią zmianę stanu środowiska.

P: Jakie rodzaje środowiska napotyka inteligentny agent?

O: Inteligentny agent napotyka deterministyczne lub stochastyczne środowisko. Środowisko deterministyczne ma poziom pewności oparty na standardzie środowiska i najnowszym działaniu agenta. W tego typu środowisku działanie w czasie t, gdy środowisko znajduje się w stanie st, powoduje stan deterministyczny i nagrodę. Jednak w przypadku środowiska stochastycznego istnieje pewien poziom niepewności pod względem stanu środowiska, a także nagrody za to samo działanie w tym samym stanie środowiska.

Podsumowanie

Omówiliśmy jedną z najważniejszych technik uczenia maszynowego, RL. Rozumiemy różnicę między RL a nauką nadzorowaną. Uczenie się agenta w oparciu o wzmocnienie behawioralne ma ogromne znaczenie w modelowaniu inteligentnych maszyn, które wypełnią lukę między ludzkimi możliwościami a inteligentnymi maszynami. Widzieliśmy podstawowe koncepcje algorytmu RL wraz z uczestniczącymi komponentami. Próbowaliśmy także ustanowić równania matematyczne dla ogólnego algorytmu RL, w których ogólnym celem jest maksymalizacja skumulowanych korzyści dla agenta, ponieważ przechodzi przez różne stany przy każdym działaniu. Próbowaliśmy krótko zrozumieć MDP w deterministycznym i stochastycznym środowisku. Omówiliśmy również w skrócie koncepcje programowania dynamicznego wraz z algorytmami uczenia Qlearning i SARSA. Na koniec omówiliśmy pokrótce DRL uczenia się głębokiego wzmacniania jako połączenie głębokich sieci neuronowych i paradygmatu uczenia wzmacniającego. Przypadki użycia, które można wyprowadzić, są ogromne w tej części stworzyliśmy podstawy do eksploracji naszej kreatywności. W następnym rozdziale zajmiemy się jednym z najważniejszych aspektów zarządzania danymi, bezpieczeństwem. Cyberbezpieczeństwo jest niezwykle ważne przy rosnącej ilości danych. Przeanalizujemy podstawowe koncepcje ochrony infrastruktury wraz z niektórymi strukturami dostępnymi do przetwarzania strumieniowego i wykrywania zagrożeń w czasie rzeczywistym.