Wydział Elektroniki i Technik Informacyjnych
Politechnika Warszawska

Algorytmy Heurystyczne
Neuroewolucja - gra w gre

Kacper Kula, Krzysztof Pierczyk
Warszawa, czerwiec 2020

Spis treści

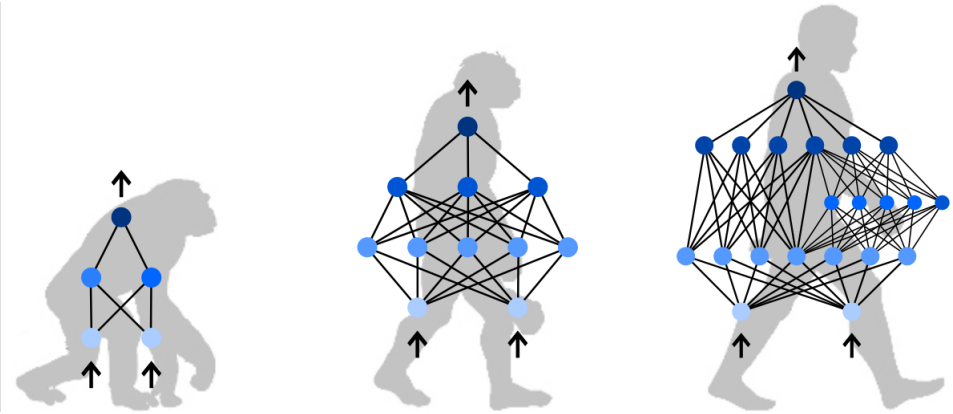
1. Wprowadzenie 2
Opis projektu 2
2. Podejście klasyczne 4
Uczenie ze wzmocnieniem 4
Deep Q Network 5
Implementacja 6
Bibliografia 7
Bibliografia 7

1. Wprowadzenie

Inspiracje biologiczne są obecne w inżynierii już od wielu lat. Możemy się na nie natknąć nie tylko w mechanice, awionice czy rbotyce. Zagrzały one miejsce również wśród szeroko rozumianych metod sztucznej inteligencji (ang. *Artificial Intelligence, AI*). Jedną z najciekawszych inspiracji na tym polu wydają sie być sztuczne sieci neuronowe (ang. *Artificial Neural Networks, ANN*). Ich koncepcja pojawiła się już w połowie XX wieku [4], jednak brak teoretycznych podstaw, które umożliwiałyby efektywne uczenie sieci sprawi, że nie zyskały one popularności w następnych latach. Dopiero zaproponowana w 1986 metoda wstecznej propagacji błędu [2] spowodowała powrót do koncepcji ANN szerszego grona badaczy. Dynamiczny wzrost mocy obliczeniowej komputerów oraz wypracowanie solidnych podstaw teoretycznych sprawiły, że sztuczne sieci neuronowe są dzisiaj jednym z głównych narzędzi sztucznej inteligencji, które wykorzystywane jest w badaniach naukowych, analizach rynku czy szeroko pojętym modelowaniu.

Opis projektu

Chociaż algorytmy oparte na wstecznej propagacji błędu stanowiły przez lata oś zainteresowania w kwestii uczenia ANN okazują się one nie być jedynymi dostępnymi. Celem naszego projektu było zaimplementowanie alternatywnych metod uczenia opartych o mechanizmy znane z algorytmów ewolucyjnych, sprawdzenie ich wydajności oraz porównanie z podejściem klasycznym na bazie gier znanych z konsoli Atari. Jako środowisko testowe posłużyła nam otwarta biblioteka *OpenAI Gym* dostępna w języku Python. Poprzez prosty interfejs udostępnia ona szerek ujednoliconych środowisk testowych umożliwiających testowanie algorytmów uczenia ze wzmocnieniem. Spośród nich wybraliśmy emulację gry Breakout (w wersji ze stanem reprezentowanym przez pamięć RAM gry). Klasyczne wykorzystanie sieci neuronowych kojarzy się raczej z uczeniem nadzorowanym, którego zadaniem jest budowa modelu potrafiącego przypisać danym wejsciowym jedną z dostępnych etykiet. Istnieją jednak metody uczenia, które pozwalają wykorzystać sieci równiez w zadaniach uczenia ze wzmocnieniem. Właśnie jedeną z takich metod - *Deep Q Learning* - wykorzystaliśmy jako punkt odniesienia dla podejścia ewolucyjnego.



Rys. 1.1. Poglądowe spojrzenie na neuroewolucję

Jednym z największych problemów stojących przed projektantem sztucznej sieci neuronowej jest odpowiedni dobór hiperparametrów takich jak ilość wartw czy ilość neuronów w poszczególnych warstwach. Dzięki wykorzystaniu mechanizmów ewolucyjnych możliwe jest stworzenie algorytmów, które poza parametrami sieci dostosowują również w dynamiczny sposób jej topologię. W ramach projektu zdecydowaliśmy się zbadać zarówno metody o stałej jak i o zmiennej topologii co dobprowadziło do wyróżnienia czterech rónych wariantów uczenia:

- uczenie sieci o stałej topologii z wykorzystaniem wstecznej propagacji błędu
- uczenie sieci o stałej topologii z wykorzystaniem algorytmu ewolucyjnego
- uczenie sieci o zmiennej topologii dostosowywanej przez algorytm ewolucyjny; dobór wag metodą wstecznej propagacji błędu
- uczenie sieci o zmiennej topologii dostosowywanej przez algorytm ewolucyjny; dobór wag metodą ewolucyjną

2. Podejście klasyczne

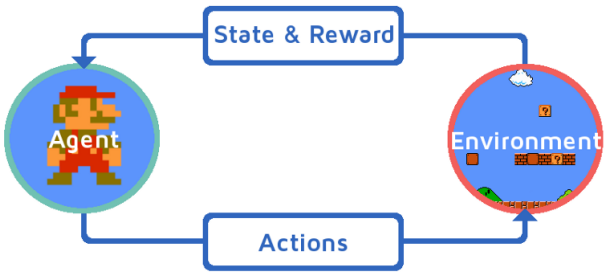
Aby wyznaczyć punkt odniesienia dla badań związanych z neuroewolucją rozpoczęliśmy od uczenia naszej sieci metodą wstecznej propagacji błędu. Pomysł na wykorzystanie jej w uczeniu ze wzmocnieniem nie przychodzi w tak oczywisty sposób jak w przypadku uczeni nadzorowanego. Z tego wzlędu postanowiliśmy przyjrzeć się głębiej naturze samego problemu.

Uczenie ze wzmocnieniem

Uczenie ze wzmocnieniem (ang. *Reinforcement Learning, RL*)jest to dziedzina uczenia maszynowego zajmująca się metodami wyznaczania optymalnej strategii zachowania agenta w nieznanym mu środowisku. Celem agenta jest maksymalizacja nagrody otrzymywanej za interakcję ze środowiskiem. środowisko modeluje się najczęciej jako proces decyzyjny Markova (ang. *Markov decission process, MDP*), który można przedstawić jako krotkę podstaci

$$< S, A, P_a(s, s'), R_a(s, s'), O > \tag{2.1}$$

gdzie S - zbiór wszystkich stanów agenta i środowiska, A - zbiór akcji możlwiwych do podjęcia przez agenta, $P_a(s, s')$ - prawdopodobieństwo przejścia ze stanu s do stanu s' pod wplywem akcji a, $R_a(s, s')$ - natychmiastowa nagroda przy przejściu ze stanu s do stanu s' pod wpływem akcji a, O - zasady opisujące obserwacje agenta. Agent oddziałuje z otoczeniem w dyskretnych chwilach. W każdej chwili t otrzymuje on obserację o_t i nagrodę r_t (zazwyczaj w postaci wartości skalarnej) oraz decyduje się na podjęcie akcji a_t. Proces ten został poglądowo przedstawiony na Rys.2.1



Rys. 2.1. Typowy schemat interakcji agenta ze środowiskiem

Model podejmowania decyzji przez agenta nazywa się polityką (ang. *policy*) i można go przedstawić w podstaci 2.2. Polityka określa prawdopodobieństwo podjęcia przez agenta akcji a w stanie s.

$$\begin{cases} \pi : A \times S \to [0, 1] \\ \pi(a, s) = P\{a_t = a | s_t = s\} \end{cases} \tag{2.2}$$

Aby ocenić jakość stanu agent dysponuje funkcją wartości stanu (ang. *state-value function*) postaci widocznej na 2.3, gdzie γ jest skalarem z przedziału $[0, 1]$. Zwraca ona wartość oczekiwaną sumy przyszłych nagród wziętych z odpowiednimi współczynnikami oznaczaną przez R.

$$V_\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s\right] = E\left[R | s = s_0\right] \tag{2.3}$$

Funkcja wartości polityki(ang. *value function*) jest z kolei metodą określenia jakości całego mechanizmu decyzyjnego. Analogicznie do funkcji wartości stanu jest ona zdefiniowana jako wartość oczekiwana ważonej sumy przyszłych nagród dla stanu s przy wykorzystaniu polityki π. Ukazano ją na 2.4.

$$V^\pi(s) = E\left[R | s, \pi\right] \tag{2.4}$$

Definicja ta pozwala nam zdefiniować optymalną politykę jako tę, która maksymalizuje swoją wartość V^π niezależnie od wybranego stanu s. Chociaż definicja ta jest wystarczająca, często definiuje się dodatkowy element nazywany funkcją wartości akcji (ang. *action-values function*). Jest ona postaci 2.5 i określa oczekiwaną wartość ważonej sumy przyszłych nagród w sytuacji, w której agent, znajdując sie w stanie s wykonał akcję a, a następnie działał zgodnie z politykąπ.

$$Q^\pi(s, a) = E[R | s, a, \pi] \tag{2.5}$$

Algorytmy uczenia ze wzmocnieniem są budowane na dwa sposoby. Pierwszy to bezpośrednia próba odnalezienia polityka $\pi(s)$, czyli modelu, który widząc na wejściu stan s zwraca optymalną akcję a. Drugie podejście modeluje nie samą politykę, a funkcję $Q^\pi(s, a)$. Z definicji tej funkcji wynika, że jeżeli π^* jest polityką optymalną, to agent moze postępować optymalnie poprzez wybieranie tej akcji ze zbioru $Q^{\pi^*}(s, a) : a \in A$, której wartość jest największa - nie wymaga od nas znajomości samej polityki π^*. To na pozór trywialne spostrzeżenie, zwane *równaniem Bellmana*[1] pozwala na budowę efektywnych algorytmów uczenia, których przykładem jest wybrany przez nas *Deep Q Network* (DQN).

Deep Q Network

Q-learning jest algorytmem uczenia ze wzmocnieniem bazującym na estymacji funkcji $Q(s, a)$ (stąd nazwa). Estymowana funkcja pozwala agentowi podejmować suboptymalne decyzje poprzez wybór tej z nich, która maksymalizuje jej wartość. Wartości Q są aktualizowane wraz z kolejnymi obserwacjami. Wzór na aktualizację bazuje na równaniu Bellmana i ma postać 2.6

$$Q^{t+1}(s_t, a_t) = Q^t(s_t, a_t) + \alpha \times \left[r_t + \gamma \times \max_a Q^t(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right] \tag{2.6}$$

Sieć aproksymująca posiada liczbę wejść równą liczbie zmiennych opisujących stan układu, natomiast liczb wyjść jest równa liczbie możliwych do podjęcia przez agenta akcji. Każde z wyjść opisuje zatem wartość jednej z tych akcji w stanie podanym na wejściu. Stosując równanie 2.6 do takiej sieci uzyskujemy funkcję błędu postaci 2.7

$$c(s) = E\left[\|(r_t + \gamma \times \max_{a_{t+1}} Q^t(s_{t+1}, a_{t+1})) - Q^t(s, a)\|\right] \tag{2.7}$$

Implementacja

Implementacja DQN została w dużej części zaczerpnięta z [3]. Jak już wcześniej wspomniano, poligonem testowym była dla nas gra Breakout, jednak sam program został stworzony tak, aby umożliwić wykorzystanie dowolnego środowiska udostepnianego przez *OpenAI Gym*. Agent jest inicjalizowany losowymi wagami. W kolejnych iteracjach wykonuje on akcje, dla których przewidywana wartość skumulowanej nagrody jest największa. Korelacja między kolejnymi obserwacjami może sprawić, że sieć będzie niestabilna. Aby temu zapobiec krotki postaci 2.8

$$< s_t, a_t, r_t, s_{t+1} > \tag{2.8}$$

są zapisywane w dedykowanej tablicy. Po zakończeniu podejścia do gry zostaną one wykorzystane w procesie uczenia. Zmniejszenie korelacji następuje poprzez losowy wybór próbek z zapisanego zbioru. Ilość próbek wykorzystana w pojedynczej iteracji STG (ang. *Stochastic Gradient Descend*) jest jednym z parametrów algorytmu. Obszar pamięci, w którym zapisywane są dane uczące stanowi kolejkę o ograniczonej pojemności. Jeżeli się ona przepeni, to nowe dane wstawiane są na początku kolejki, natomiast najstarsze dane zostją usunięte. Aby kontrolować tendencje algorytmu do eksplorowania przestrzeni wprowadzony został również parametr $\epsilon \in [0, 1]$. Gdy ma on niezerową wartość istnieje szansa, że agent wykona losową akcję zamiast tej przewidzianej przez sieć. Parametr ten jest inicjalizowany niezerową wartością i zmniejszany wraz przebiegiem uczenia.

Bibliografia

[1] R. Bellman. *On the Theory of Dynamic Programming*. 1952.

[2] Ronald J. Williams David E. Rumelhart, Geoffrey E. Hinton. *Learning representations by back-propagating errors*. 1986.

[3] John Krohn. Cartpole dqn. `https://github.com/the-deep-learners/TensorFlow-LiveLessons/blob/master/notebooks/cartpole_dqn.ipynb`.

[4] F. Rosenblatt. *The Perceptron: A Probabilistic Model For Information Storage And Organization In The Brain*. 1958.