MCTSnet - Marta Oniszk

August 12, 2018

1 MCTSnet

ZADANIE

Wyjaśnić zasadę działania i cele metody optymalizacji drzew Monte Carlo opisanych w publikacji: https://arxiv.org/abs/1802.04697v1. Ocenić możliwość zastosowania tego podejścia do problemów optymalizacyjnych.

Omówienie zostało wykonane na podstawie ww. dokumentacji, ale też danych z Internetu, w celu lepszego zrozumienia algorytmu.

Celem publikacji jest przedstawienie działania algorytmu **MCTSnet**, która jest pochodną podstawowego algorytmu MCTS. Omawiany algorytm zawiera rozszerzenie, ewaluację i backup. Co więcej zastosowano tu metody pozwalające na optymalizację występującego *gradientu*. Co w efekcie pozwala na skuteczniejsze wykorzystanie algorytmu.

2 PODSTAWOWE INFORMACJE

w algorytmie zastosowana jest metodologia **RL**

"Reinforcement learning (RL) is an area of machine learning, inspired by behaviorist psychology, concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward." https://en.wikipedia.org/wiki/Reinforcement_learning

Zadaniem RL jest działania na zasadzie trial-and-error, czyli na zasadzie prób i błędów tworzymy kolejne scenariusze, ale naszym celem jest maksymalizacja efektu nagrody

Może być rozumiany jako algorytm szukający, który w sposób kontrolowany odzwierciedla symulację podróżowania. Może też być rozumiany jako sieć neuronowa w postaci wykresu obliczeniowego, który działa na zasadzie pośrednich obliczeń w celu wybrania jednej najlepszej ścieżki.

Inne tłumaczenie: agent (osoba) dokonuje akcji w nieznanym sobie środowisku. Za każdą wykonaną akcję dodane jest wzmocnienie (może to być nagroda albo kara, czyli pozytywne albo negatywne). Celem jest wyznaczenie optymalnej policy, która pozwoli na uzyskanie jak największe ilości nagród - metoda prób i błędów.

ZASADY DOTYCZĄCE DRZEWA:

1.ma jasno określone zasady, pozwalające na kontrolę kolejnych ruchów

2.wykorzystujemy tu *simulation policy*, która pozwala nam na wybranie odpowiedniej drogi (root), podążając kolejnymi punktami (liścmi - leafs) (liczba symulacji jest nieograniczona)

3.liście (leafs) oparte są na embedding network(osadzenie), czyli naszym punktem przejściowym jest liść na którym zapisywane są dane i który definiuje kolejny krok

4.przy każdym ruchu po liściach wykorzystujemy backup network, który zapisuje każdy ruch i analizuje go

To pozwala na wygenerowanie (trenowanie), przeanalizowanie wielu różnych ścieżek (iterative local compucations and structured memory), a potem wybranie tej najlepszej, która pozwoli na maksymalizację efektu nagrody

GDZIE MOŻNA WYKORZYSTAĆ ALGORYTM: - gry takie jak gra GO, PacMan, gra kółko-krzyżyk

ZALETY MCTS - *przygotowane na podstawie innych publikacji na temat tego algorytmu*

1.ten rodzaj drzewa nie potrzebuje żadnych danych na początek, na temat gry, co oznacza że może być stosowany przy wielu rodzajach gier przy lekkiej modyfikacji (jest to wyjątek, standardowo algorytm na początku potrzebuje danych)

2.jest możliwość budowania drzewa w sposób asymetryczny

3.łatwy do implementacji

4.dobry do stosowania przy dużych zbiorach danych

WADY MCTS

1.ze względu na to, że można przechodzić ścieżki i szukać danego algorytmu w nieskończoność, algorytm może np. działać na ślepo, nie uczyć się wystarczająco szybko. W efekcie możemy nie uzyskać zamierzonego efektu w przewidzianym czasie. Jeśli nie mamy podstaw, hintów, to może się okazać, że algorytm będzie potrzebował znacznie więcej czasu na naukę i wybór odpowiedniej ścieżki - *dzięki zastosowaniu metody małych kroków,omówionej w artykule minimalizujemy to zjawisko*

2.jeżeli zbiór danych jest duży wiąże się to z tym, że potrzebujemy dużo próbek, czyli dużo czasu na wygenerowanie tej dużej ilości próbek

W przypadku gdy start jest *spacious*, czyli przestronny, zaczynamy już z pozycji, gdzie od początku mamy wiele różnych wyników, czyli kumuluje nam się liczba ścieżek(symulacja jest droga i przynosi mało rezultatów).To w efekcie sprawia, że liście są trudne do ewaluacji (duży szum wokół).

3 ANALIZA ALGORYTMU MCTSnet

```
In [21]: from IPython.display import Image
         Image(filename='/home/marta/Desktop/mctsnet1.png')
```

```
Out[21]:
```

Algorithm 2: **MCTSnet**
For $m = 1 \ldots M$, do simulation:

1. Initialize simulation time $t = 0$ and current node $s_0 = s_A$.

2. Forward simulation from root state. Do until we reach a leaf node ($N(s_t) = 0$):

   (a) Sample action a_t based on *simulation policy*, $a_t \sim \pi(a|h_{s_t}; \theta_s)$,

   (b) Compute the reward $r_t = r(s_t, a_t)$ and next state $s_{t+1} = T(s_t, a_t)$.

   (c) Increment t.

3. Evaluate leaf node s_L found at depth L.

   (a) Initialize node statistics using the embedding network: $h_{s_L} \leftarrow \epsilon(s_L; \theta_e)$

4. Back-up phase from leaf node s_L, for each $t < L$

   (a) Using the backup network β, update the node statistic as a function of its previous statistics and the statistic of its child:

$$h_{s_t} \leftarrow \beta(h_{s_t}, h_{s_{t+1}}, r_t, a_t; \theta_b)$$

After M simulations, readout network outputs a (real) action distribution from the root memory, $\rho(h_{s_A}; \theta_r)$.

Jest to generalizacja podstawowego algorytmu, gdzie dodatkowo mamy takie aspekty jak algorytm śledzący zmiany, backup pozwalajacy na zapamiętywanie dokonanych ścieżek i politykę polegającą na rozszerzaniu naszej analizy oraz wyboru najlepszej drogi, pozwalającej na maksymalizację nagrody

```
In [22]: from IPython.display import Image
         Image(filename='/home/marta/Desktop/2.png')
```

Out[22]:

1. Initialize simulation time $t = 0$ and current node $s_0 = s_A$.

Inicjalizujemy symulację, gdzie startujemy z pozycji zerowej czyli t=0, a node, czyli liść to 1

```
In [23]: from IPython.display import Image
         Image(filename='/home/marta/Desktop/3.png')
```

```
Out[23]:
```

2. Forward simulation from root state. Do until we reach a leaf node ($N(s_t) = 0$):
   (a) Sample action a_t based on *simulation policy*, $a_t \sim \pi(a|h_{s_t}; \theta_s)$,
   (b) Compute the reward $r_t = r(s_t, a_t)$ and next state $s_{t+1} = T(s_t, a_t)$.
   (c) Increment t.

a) Kolejny krok to potwarzająca się symulacja, którą powtarzamy, aż uzyskamy N(node) na pozycji początkowej, czyli 0. Opieramy się na *policy*, która mówi nam jaki ruch najlepiej wykonać, dzięki temu przechodzimy ze *state do state (s)* przez wykonanie akcji (a). Ważny jest tu element ekspansji (expansion step), czyli mini symulacji, pozwalających na zbadanie wielu różnych opcji dróg,ale są to symulacje ukryte (pośrednie), pokazujące, jaki może być nasz kolejny ruch. Kolejny krok to wykorzystanie *random (rollup) policy*, która pozwala na określenie możliwych ścieżek i przeanalizowaniu, jaka ilość nagród czeka na nas po drodze - daje nam to *estimate Q value (s,a)*, czyli tworząc możliwe ścieżki, ale nie wykonując jej jeszcze, jesteśmy coraz bardziej pewni tego, co mamy wykonać. Wersja ostateczna Q składa się tylko z pewnych, przeanalizowanych ruchów, bo w trakcie wykonaliśmy random policy

b) obliczanie, która z dróg da nam najwięcej nagród i wybranie jej. Tu oznacza się to przez +1, czyli wiemy, że ta akcja da nam najlepszy wynik

c) podniesienie t, czyli liczba akcji jakie można wykonać na pewno, czyli na przykład dzięki rollup policy jesteśmy pewni, że możemy wykonać ruch w prawo 2 razy

```
In [24]: from IPython.display import Image
         Image(filename='/home/marta/Desktop/4.png')
```

```
Out[24]:
```

3. Evaluate leaf node s_L found at depth L.
   (a) Initialize node statistics using the embedding network: $h_{s_L} \leftarrow \epsilon(s_L; \theta_e)$

Ewaluacja liści, wykonanych ruchów, przez stworzenie node, który jest osadzony, czyli jesteśmy pewni, że to jest dobry ruch. L to jest przykładowy punkt zatrzymania, przykładowy state.

```
In [25]: from IPython.display import Image
         Image(filename='/home/marta/Desktop/5.png')
```

Out[25]:

4. Back-up phase from leaf node s_L, for each $t < L$

   (a) Using the backup network β, update the node statistic as a function of its previous statistics and the statistic of its child:

$$h_{s_t} \leftarrow \beta(h_{s_t}, h_{s_{t+1}}, r_t, a_t; \theta_b)$$

After M simulations, readout network outputs a (real) action distribution from the root memory, $\rho(h_{s_A}; \theta_r)$.

Każdy z node ma swoją własną statystykę dzięki backup.

W artykule jest mowa o technice pozwalającej na zminimalizowanie straty dla każdego ze state (cały czas korzystamy z *reinforcement learning*), które pozwala na manipulowanie *bias-variance trade-off*. Na czym polega bias-variance trade-off? Mamy do czynienia z 2 różnymi rodzajami błędów związanych z przewidywaniem *bias* - kiedy algorytm ma małą elastyczność do nauki sygnałów z bazy, czyli jesteśmy daleko od celu oraz *variance*, kiedy nasze modele są przetrenowane i przez to rejestrują szum zamiast prawdziwych danych - celem jest znalezienie balancu między bias i variance

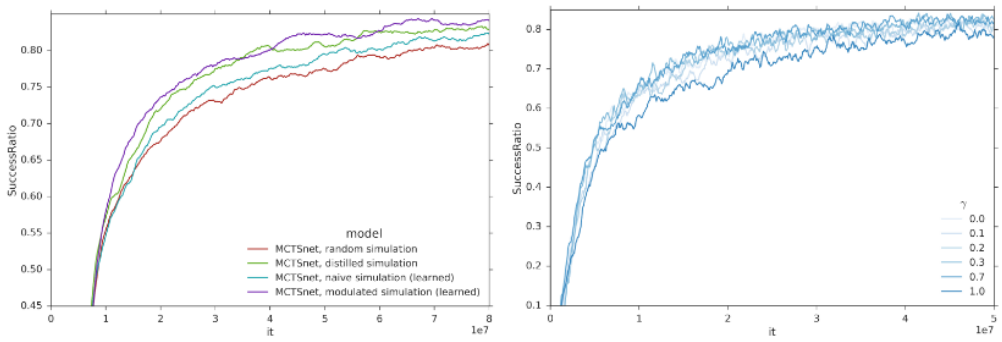
Powyższe działanie wiąże się z innym aspektem. W algorytmie wykorzystujemy różnicę między *terminal loss* czyli określonym błędem, a pozycją startową. W kolejnym kroku robimy trade off dla bias-variance, przez γ, czyli gamma, czyli prawdopodobienstwo. W efekcie nagradzane są małe kroki, ale przynoszące większy skutek (nagrodę), bo produkuje się w ten sposób mniejszy noise, czyli mniejszy error, czyli nasze działanie jest efektywniejsze, a algorytm szybciej i skuteczniej się uczy

Jak można zoptymalizować algorytm? Znając specyfikę gry można wprowadzić udogodnienia pozwalające na przyspieszenie odpowiedniego działania, czyli odpowiednio zmodyfikowac nasz algorytm. Dla gry Go będzie to inny rodzaj dopasowania, niż dla gry w kółko i krzyżyk (na przykład w grze Go mamy system liczenia chiński i japoński, który kamień rozpoczyna czarny czy biały, inne odmiany Go jak Rengo (parami) itd.) - indywidualizacja algorytmu pod określone cele

W omawianym artykule skupiono się na γ, gdzie zmieniano jej wartość,w celu zredukowania szumu, a co za tym idzie błędu, co w efekcie ma nam dać skuteczny algorytm.

```
In [26]: from IPython.display import Image
         Image(filename='/home/marta/Desktop/9.png')
```

```
Out[26]:
```



(a) (b)

Figure 4. a) Comparison of different simulation policy strategies in MCTSnet with $M = 25$ simulations. b) Effect of different values of γ in our approximate credit assignment scheme.

b) Trenowano γ pod względem różnych wartości. Jak widać najlepszy rezultat uzyskano przy 0.3 i 0.7

a) algorytm byl trenowany pod względem różnej policy. Najlepszy rezultat odniósł dla MCTSnet, modulated simulation (learned)

```
In [27]: from IPython.display import Image
         Image(filename='/home/marta/Desktop/10.png')
```

```
Out[27]:
```

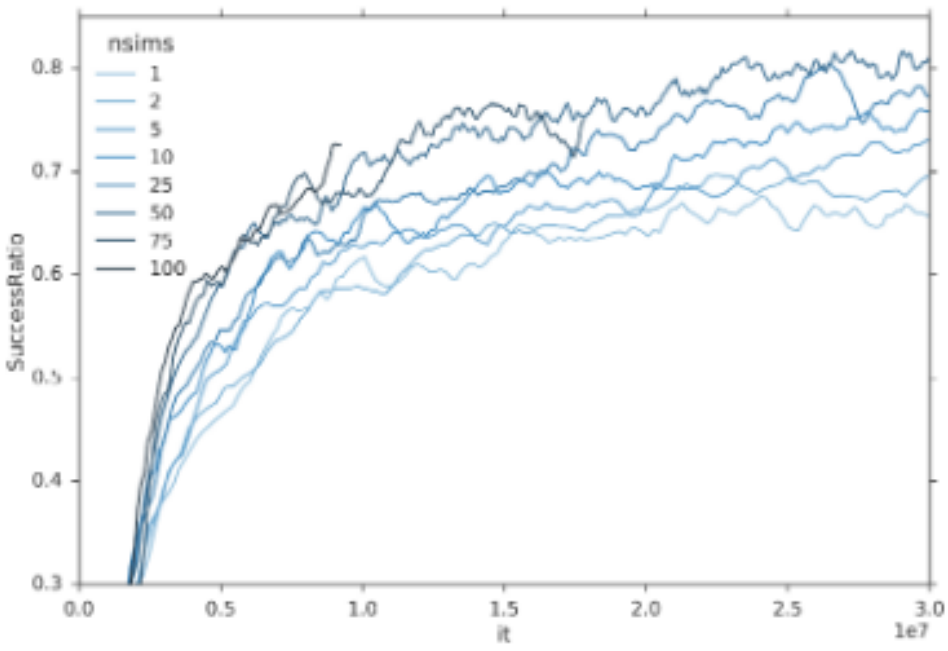


Figure 5. Performance of MCTSnets trained with different number of simulations M (nsims). Generally, larger searches achieve better results with less training steps.

Patrząc na liczbę symulacji, im większa tym osiągamy lepsze wyniki. Przekłada się też to na to, że przy wykorzystaniu takiego rozwiązania jesteśmy w stanie pracować z dużymi bazami danych (w artykule jest mowa o tym, że tu badania był wykonywane na mniejszych zbiorach, ale to nie znaczy, że ten algorytm nie zadziała na dużej bazie także) - jest to bardzo dobra wiadomość. Dzięki temu mamy możliwość zapanowania nad dużą wadą podstawowego algorytmu MCTS, gdzie symulacja może być nieskuteczna i długotrwała, a przy dyżych bazach danych, może dostarczyć "przekłamanych" wyników.

Źródła: https://arxiv.org/pdf/1802.04697v1.pdf

https://dloranc.github.io/blog/2017/04/16/reinforcement-learning-co-to-jest/

https://elitedatascience.com/bias-variance-tradeoff

http://mcts.ai/about/

https://www.quora.com/What-are-the-pros-and-cons-comparing-Monte-Carlo-tree-search-and-approximate-dynamic-programming

Dziękuję!