Projekt USD - Space Invaders

Autorzy:
Adam Grabski
Jakub Dmochowski

Wprowadzenie

Celem naszego projektu było zaimplementowanie dwóch algorytmów sztucznej inteligencji, grających w grę Space Invaders oraz porównanie ich osiągnięć. Algorytmy które postanowiliśmy wykorzystać to Aktor-krytyk oraz Sarsa.

Założenia wstępne

W obu algorytmach wstępnie założyliśmy, że będziemy pracować na wersji Space Invaders, która nie wykorzystuje stanu w postaci zbioru pikseli wyświetlacza, tylko w postaci pamięci RAM wykorzystywanej do wyświetlenia gry.

Jeśli algorytmy poprawnie działałyby dla mniejszej ilości danych o stanie, to powinny również dobrze się sprawdzić dla większych ilości danych o stanie.

Tak więc raport został utworzony na podstawie zmagań ze stanem składającym się ze 128 zmiennych przyjmujących 256 wartości.

Do implementacji algorytmów użyliśmy bibliotek Tensorflow i Keras, oraz biblioteki wspomagającej NumPy.

Algorytmy

Środowisko

Stan $X = \{x_1, x_2, ..., x_t\}$, $x_t = [x_{t1}, x_{t2}, ..., x_{t128}]$, $x_{ti} \varepsilon \{0, 1, ..., 255\}$ dla $i \varepsilon Z \wedge i \leq 128$
Akcje $U = \{u_1, u_2, ..., u_6\}$

Aktor-krytyk(λ)

Dodatkowe założenia środowiskowe

Estymator wartości stanu (Krytyk) $\overline{V}(x_t; v)$,gdzie v to wektor parametrów Krytyka
Estymator prawdopodobieństw akcji dla stanu (Aktor) $\pi(u_t; x_t; \theta)$, gdzie θ to wektor Aktora

Pseudokod

0: Zainicjalizuj θ i v; przypisz $t \leftarrow 1$, $z \leftarrow 0$, $y \leftarrow 0$.
1: W chwili t wykonaj akcję $u_t \sim \pi(\cdot\,; x_t, \theta)$.
2: Obliczenie *różnicy czasowej*: Jeśli t jest ostatnią chwilą epizodu oblicz

$$d_t = r_t - \bar{V}(x_t; v)$$

w przeciwny razie

$$d_t = r_t + \gamma \bar{V}(x_{t+1}; v) - \bar{V}(x_t; v).$$

3: Poprawka polityki (uczenie Aktora)

$$z \leftarrow \lambda\gamma z + \frac{\partial}{\partial \theta^T} \ln \pi(u_t; x_t, \theta),$$
$$\theta \leftarrow \theta + \beta_t^\theta d_t z.$$

4: Poprawka aproksymatora $\bar{V}$ (uczenie Krytyka)

$$y \leftarrow \lambda\gamma y + \frac{\partial}{\partial v^T} \bar{V}(x_t; v),$$
$$v \leftarrow v + \beta_t^v d_t y.$$

5: Jeśli t jest ostatnią chwilą epizodu, wyzeruj z i y.
6: Przypisz $t \leftarrow t + 1$ i przejdź do Punktu 1.

[1]

Implementacja

Przyjęliśmy, że estymatory Aktor i Krytyk będą zaimplementowane w postaci sieci neuronowej z wykorzystaniem Tensorflow. Sieć neuronowa jako wejście dostaje zestaw 128 zmiennych będących reprezentacją stanu, ma jedną warstwę ukrytą składającą się z 64 neuronów, która jest przekazywana jako wejście do dwóch warstw wyjściowych, gdzie jedna warstwa zwraca wynik estymaty Aktora, a druga Krytyka.
Wynik Aktora to wektor 6-elementowy, reprezentujący prawdopodobieństwa podjęcia akcji w stanie który podamy sieci na wejściu. Tą sieć neuronową nazywamy modelem.
Uczenie modelu, odbywa się z użyciem obiektu GradientTape (tape). Zbiera on informacje o zmiennych w ramach zakresu bloku kodu źródłowego względem obserwowanych zmiennych, gdzie zmienne trenowalne modelu są automatycznie zmiennymi obserwowanymi.

[1] Paweł Wawrzyński, prezentacja z przedmiotu USD blok-7

Obiekt zwracany przez funkcje tape.gradient(arg, model.trainable_variables) traktujemy jako kierunek zmian pochodnych estymatorów po zmiennych trenowalnych modelu.
Według naszej interpretacji działania tej funkcji, przyjmuje ona parametr arg i oblicza kierunek zmiany zmiennych trenowalnych modelu względem tego parametru.
Argument powinien mieć strukturę odpowiadającą wyjściu modelu.
Wynik traktujemy go jako wektor kierunku zmian parametrów trenowalnych modelu.
W matematycznym zapisie, naszą interpretacją działania tej funkcji jest formuła

$$grads = \frac{\partial}{\partial \theta} arg$$

Gdzie θ to trenowalne parametry modelu.
Aby uzyskać formułę 3 i 4 kroku pseudokodu, obliczamy pochodne używając tej funkcji:
Grads = tape.gradient([log_prob, critic_value], model.trainable_variables)
A następnie mnożymy jej wyniki przez różnicę czasową (zgodnie z krokiem 2. pseudokodu)
Time_diff = reward - critic_value
Lub gdy t nie jest ostatnią chwilą epizodu
Time_diff = reward + gamma * next_critic_value - critic_value
Gdzie critic_value to wyjście Krytyka z modelu dla stanu t,
Next_critic_value to wyjście Krytyka z modelu dla stanu t+1 (po podjęciu akcji sprawdzony stan znowu przepuszczamy przez sieć neuronową i sprawdzamy wynik Krytyka)

Jeśli nasza interpretacji funkcji gradient jest poprawna, to matematycznie moglibyśmy zapisać wynik jako

$$grads = [\frac{\partial}{\partial \theta} ln(\pi(u_t; x_t; \theta)),\ \frac{\partial}{\partial \theta} \overline{V}(x_t; v)] \cdot d_t$$

Tak utworzony obiekt $grads$ aplikujemy do modelu.
W pseudokodzie w krokach 3 i 4 występuje tam również parametr kroku β_t^v, który w naszej implementacji jest zarządzany przez optymalizator, jest on przez niego dobierany z użyciem algorytmu Adam, dla którego parametr prędkości uczenia ustawiliśmy na 0,0001 (w trakcie jednego epizodu wykonujemy czasem kilkaset akcji, żeby model nie zmienił się zbyt drastycznie w trakcie jednego epizodu, musieliśmy ustawić parametr uczenia na tyle mały, by w trakcie działania pełnego epizodu parametry trenowalne nie zmieniły się bardziej niż o 10%)
Aplikacja wyliczonego gradientu, pomnożonego przez parametr kroku traktujemy jako równoważną z nadpisywaniem parametrów trenowalnych modelu θ i v z pseudokodu w krokach 3 i 4.
W implementacji aplikacja gradientu wykonywana jest przez następujący kod:
optimizer.apply_gradients(zip(grads, model.trainable_variables))

Dopasowanie parametrów:
γ -parametr gamma ustalamy na 0.99, ponieważ chcemy by nagrody uzyskane po dłuższym czasie (nawet 100 kroków), były również brane pod uwagę do zmiany parametrów modelu.
λ-parametr lambda ustaliliśmy na 0.99, ponieważ chcemy by okno przyszłych wartości nagród było szerokie
Wielkość wspólnej warstwy ukrytej w modelu - ustaliliśmy na 64, może być to mało biorąc pod uwagę 128 zmiennych wejściowych. Kierowaliśmy się doborem zgodnym z regułą kciuka głoszącą, że wielkość warstwy nie powinna być większa niż ta warstwy poprzedniej ani mniejsza niż warstwy następnej.
Learning rate optymalizatora - ustaliliśmy jako 0.0001 w taki sposób, by w trakcie jednego epizodu względnie model nie zmienił zbyt mocno swojego wytrenowania. Ponieważ

aktualizowany model jest w trakcie każdego kroku algorytmu, a kroków w algorytmie może być kilkaset, to łączna zmiana w trakcie epizodu nie powinna przekraczać wartości granicznej. Dla tak wybranego parametru, wartość graniczna jest na poziomie kilku punktów procentowych w najgorszym wypadku.

SARSA

Zaimplementowaliśmy algorytm Sarsa dla dyskretnego stanu. Ponieważ przestrzeń stanów gry Space Invaders jest bardzo duża i niekoniecznie ciągła (niektóre elementy pamięci ram są stałe) do przechowywania funkcji wartości akcji użyliśmy słownika zamiast bardziej typowej tablicy.

Stany, w celu indeksowania funkcji wartości akcji są przetwarzane na duże liczby całkowite. Odbywa się to przez konkatenację wszystkich bajtów wektora wejściowego. W trakcie implementacji tej funkcjonalności napotkaliśmy się na nietypowe zachowanie pythona.

Ten język, na poziomie implementacyjnym posiada dwa typy liczb całkowitych: int oraz long. Int ma stały, zależny od platformy rozmiar (dla procesorów x64 wynosi on 64 bity) natomiast long jest arbitralnie duży. Środowisko przełącza się między używaniem tych dwóch typów w zależności od potrzeby. Jeśli na przykład na liczbie 1 (int) dokonamy przesunięcia bitowego o 65 bitów to dojdzie do “promocji” do typu long. Natomiast, jeśli dokonamy przesunięcia o 32 bity a potem o 33 to dojdzie do integer overflow. Wczesne wersje naszej implementacji algorytmu SARSA przez ten błąd operowały na przestrzeni zaledwie około 40 stanów.

Po naprawieniu tego błędu, przestrzeń odwiedzonych stanów w czasie 6 tysięcy gier wzrosła do około 150 tysięcy, a w czasie około 20 tysięcy gier przekroczyła milion. Wykres nagrody za epizod z zaznaczoną linią trendu, pokazuje że wyniki agenta sterowanego tą polityką nie zależą od czasu treningu (wykres 1).

Drugim przyjętym podejściem było zastosowanie sieci neuronowej jako aproksymator funkcji wartości akcji. Użyliśmy tego mechanizmu, ponieważ ze względu na bardzo abstrakcyjną naturę prezentowanych stanów, nie mieliśmy możliwości stworzenia własnej funkcji aproksymującej.

W ramach eksperymentów z różnymi wartościami parametrów algorytmu, testowaliśmy początkową wartość $\varepsilon = 1$. Motywacją dla tej decyzji było zapewnienie agentowi okresu poświęconego całkowicie na eksplorację aby nie musiał operować na modelu nieposiadającym żadnych doświadczeń. To podejście okazało się jednak błędne, ponieważ całkowicie losowe akcje utrzymywały go w rejonie pozycji początkowej, przez co nie eksperymentował z poruszaniem się po całej mapie. Po tym teście, ustawiliśmy wartość początkową $\varepsilon = 0.7$, zmniejszoną w każdym epizodzie o 0.05 do wartości minimalnej $\varepsilon = 0.01$.

W porównaniu z poprzednim podejściem, trening z aproksymatorem w formie sieci neuronowej trwał znacznie dłużej niż używając tablicy wartości akcji. Jest to konsekwencja całkowicie zrozumiała, ponieważ aplikacja gradientu do zmiennych sieci neuronowej jest znacznie trudniejszą operacją niż przypisanie wartości w dużej tablicy.

Implementacja

0. Dane: sekwencja $\{\beta_t, t = 1, 2, \dots\}$ spełniająca (2.15).
   Zainicjalizuj: $t := 1$, υ. Wylosuj $u_1 \in \mathcal{U}^+$.
1. Wykonaj u_t, zarejestruj x_{t+1} i r_t.
2. Wylosuj $u_{t+1} \in \mathcal{U}^+$ na podstawie $\bar{Q}$ i x_{t+1}.
3. Przypisz

$$\upsilon := \upsilon + \beta_t \frac{\partial \bar{Q}(x_t, u_t; \upsilon)}{\partial \upsilon^T} \left(r_t + \gamma \bar{Q}(x_{t+1}, u_{t+1}; \upsilon) - \bar{Q}(x_t, u_t; \upsilon) \right).$$

4. Przypisz $t := t + 1$ i wróć do Punktu 1.

W roli aproksymatora użyliśmy sieci neuronowej z pakietu keras o jednej warstwie ukrytej, wielkości 64 oraz warstwy normalizacyjnej. Podobnie jak w wypadku algorytmu Aktor-krytyk(λ), do aplikacji gradientu użyliśmy obiektu GradientTape z pakietu tensorflow.

Zgodnie z podanym wyżej algorytmem, wartość przekazywana do wyliczania gradientu wynosi:

$$reward + \gamma * currentStateValueEstimate - previousStateValueEstimate$$

Używając gradientu policzonego przez GradientTape, po każdym kroku aktualizujemy sieć neuronową używając optymalizatora z paczki Keras.

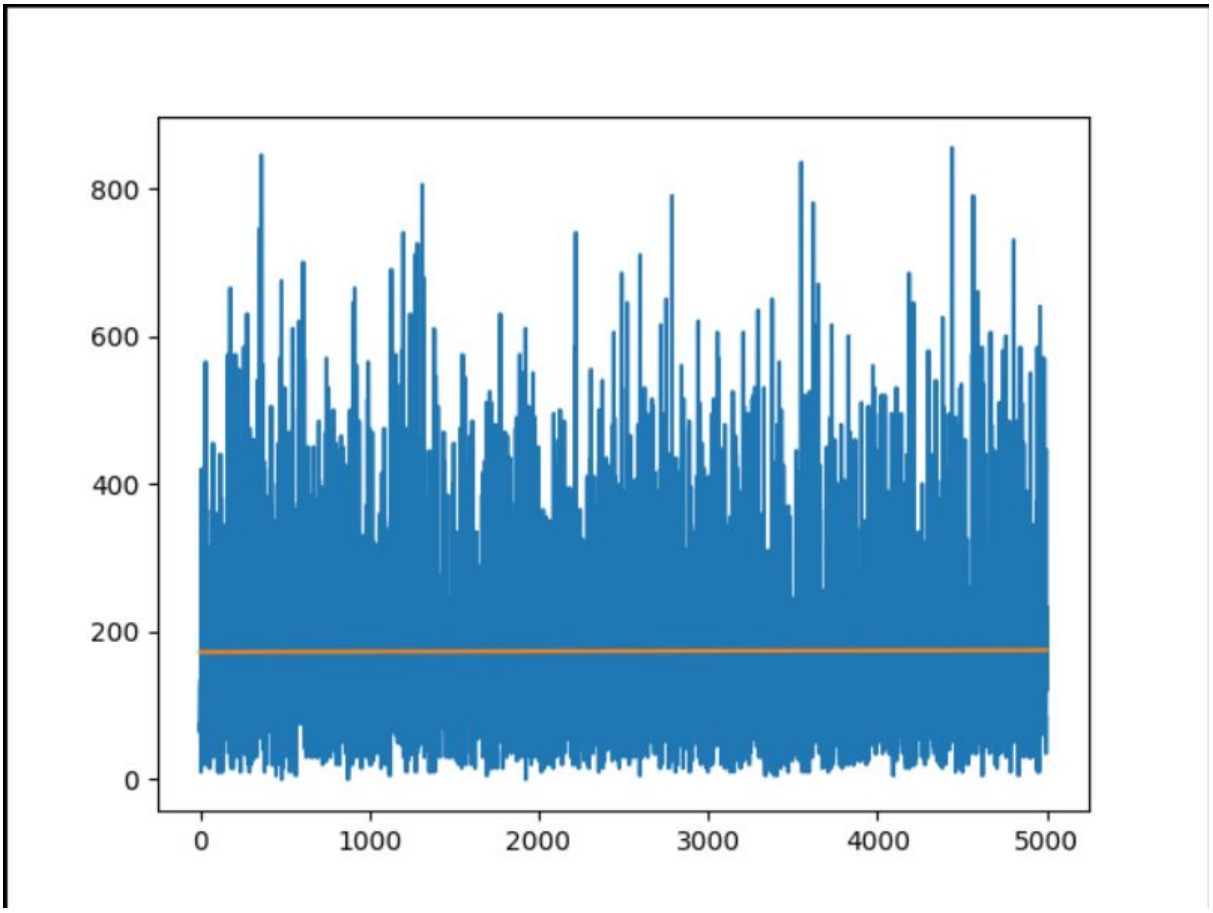
W ostatecznej wersji użyliśmy prostszego wyrażenia wyliczającego tą wartość:

$$reward - estimate$$

Ponieważ wersja pierwsza doprowadzała z niewiadomych przyczyn do sytuacji w której aktor nie podejmował żadnych akcji.

Wyniki

SARSA



Wykres 1. Nagroda za epizod algorytmu sarsa na dyskretnym stanie i z dyskretnymi akcjami (niebieski) i linią trendu (pomarańczowy) po 5 tysiącach epizodów.

Wersja algorytmu wykonana na tablicy, nie poprawiała swoich wyników w czasie.
Wersja używająca sieci neuronowej po krótkim szkoleniu (40 gier) była w stanie schować się za przeszkodami, unikać niektórych strzałów i okazjonalnie dojść do końcowej fazy gry.

Algorytm aktor-krytyk

Po dostosowaniu parametrów i próbie uruchomienia algorytmu, przeszliśmy przez 1300 epizodów, gdzie co 5 epizod zapisywaliśmy stan modelu. Początkowo zmiany w prawdopodobieństwach akcji były niewielkie i algorytm wydawał się podejmować względnie losowe akcje. Im więcej epizodów minęło, tym częściej algorytm zwracał dobre wyniki, ucząc się powoli.
Po około 750 epizodach, udało się mu przejść całą grę i uzyskać wynik >1200, gdzie średnio uzyskiwał po 400-500 punktów.
Dalej trenowana sieć wykazywała oznaki przetrenowania - pierwsze akcje podejmowane w trakcie epizodu miały prawdopodobieństwa bliskie 1. Uznaliśmy że stan sieci po około 900 epizodach miał już gorszą jakość niż ten po 750-800 epizodach. Przy sprawdzeniu sposobu

rozgrywki sieci po 1200 epizodach, widzieliśmy zatrzymanie rozwoju i wykonywanie jednej polityki decyzyjnej z prawdopodobieństwami bliskimi 1.
Niestety algorytm bardzo powoli lecz stabilnie zwiększał wartości krytyka.

Wnioski

- Na podstawie wyników algorytmu SARSA z dyskretnym stanem, można zauważyć że w praktycznych aplikacjach dyskretny stan o bardzo dużym rozmiarze, lepiej traktować jako stan ciągły.
- Sieć neuronowa o nawet prostej strukturze (jedna warstwa ukryta z 64 neuronami) jest w stanie dobrze aproksymować funkcję wartości akcji gry space invaders
- Algorytm aktora-krytyka z lambdą uczył się bardzo powoli. Ciężko jest w nim określić stopień przetrenowania. Trenował on akcje w dobrym kierunku, jednak nie znalazł momentu w którym powinien zawrócić i nie zwiększać prawdopodobieństw dobrych akcji.