Maxim Lapan

Głębokie uczenie przez wzmacnianie

Praca z chatbotami oraz robotyka, optymalizacja dyskretna i automatyzacja sieciowa w praktyce

Wydanie II, zawiera analizę metod wieloagentowych i zaawansowanych technik eksploracyjnych

Helion

Packt›

Tytuł oryginału: Deep Reinforcement Learning Hands-On: Apply modern RL methods to practical problems of chatbots, robotics, discrete optimization, web automation, and more, 2nd Edition

Tłumaczenie: Jacek Janusz

ISBN: 978-83-283-8052-3

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/glucz2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

6

ROZDZIAŁ

Głębokie sieci Q

W rozdziale 5. zapoznaliśmy się z równaniem Bellmana i praktyczną metodą jego wykorzystania zwaną **iteracją wartości**. Dzięki temu rozwiązaniu mogliśmy znacznie poprawić prędkość działania kodu i jego konwergencję w przypadku środowiska FrozenLake. Czy da się jednak pójść jeszcze dalej? W tym rozdziale zastosujemy to samo podejście do problemów o znacznie większej złożoności — gier zręcznościowych z platformy Atari 2600, które są standardowym punktem odniesienia dla naukowców zajmujących się uczeniem przez wzmacnianie.

W tym rozdziale:

- Przeanalizuję problemy związane z metodą iteracji wartości i zaprezentuję jej odmianę zwaną **Q-uczeniem**.
- Zastosuję metodę Q-uczenia w tak zwanych środowiskach świata siatki. Metoda ta jest w takich przypadkach zwana **Q-uczeniem tabelarycznym**.
- Pokażę, w jaki sposób Q-uczenie może być używane razem z sieciami neuronowymi. Takie połączenie nosi nazwę **głębokich sieci Q** (**DQN**).

Pod koniec rozdziału ponownie zaimplementujemy algorytm DQN zaprezentowany w słynnym artykule *Playing Atari with Deep Reinforcement Learning* V. Mniha i in., który został opublikowany w 2013 roku i zapoczątkował nową erę w rozwoju uczenia przez wzmacnianie.

Rozwiązywanie realnego problemu z wykorzystaniem metody iteracji wartości

Wyniki, które otrzymaliśmy w środowisku FrozenLake po zamianie entropii krzyżowej na metodę iteracji wartości, są dość zachęcające. Można się więc zastanowić nad zastosowaniem tej metody w przypadku rozwiązywania trudniejszych problemów. Najpierw jednak przyjrzyjmy się jej założeniom i ograniczeniom.

Zacznijmy od szybkiego podsumowania. W każdym z kroków metoda iteracji wartości przetwarza w pętli wszystkie stany i aktualizuje ich wartości za pomocą równania Bellmana. Istnieje również wariant tej metody, który dotyczy wartości Q (wartości dla akcji). Działa prawie tak samo jak wersja oryginalna, ale wyznacza i przechowuje wartości dla każdego stanu i akcji. W czym więc leży problem?

Pierwszą rzeczą, jaką trzeba wziąć pod uwagę, jest liczba stanów środowiska i zdolność programu do ich przetwarzania. Metoda iteracji wartości zakłada, że znamy z wyprzedzeniem wszystkie stany środowiska, potrafimy je przetwarzać, a także przechowywać przybliżone wartości z nimi związane. Jest to, ogólnie rzecz ujmując, prawdą w przypadku tak prostego środowiska świata siatki, jakim jest FrozenLake. Jak jednak działa ta metoda w innych środowiskach?

Najpierw spróbujmy ustalić, jak bardzo skomplikowane mogą być problemy rozwiązywane za pomocą metody iteracji wartości. Innymi słowy, dowiedzmy się, ile stanów możemy przetworzyć w każdej pętli. Nawet komputer średniej wielkości może przechowywać w pamięci kilka miliardów liczb zmiennoprzecinkowych (8,5 miliarda w 32 GB pamięci RAM), więc nie jest to zbytnim ograniczeniem. Przetwarzanie miliardów stanów i akcji będzie co prawda wymagało użycia lepszego procesora (CPU), ale jest to również osiągalne.

Obecnie dostępne systemy wielordzeniowe są przez większość czasu bezczynne. Prawdziwym problemem jest liczba próbek wymaganych do uzyskania dobrych przybliżeń dynamiki przejść pomiędzy stanami. Załóżmy, że masz środowisko z miliardem stanów (co odpowiada mniej więcej światowi FrozenLake o wymiarach 31 600×31 600). Aby wyznaczyć z dużym przybliżeniem wartości dla każdego stanu tego środowiska, musielibyśmy przetworzyć setki miliardów przejść pomiędzy stanami, co nie ma zbytniego sensu.

Przykładem środowiska z jeszcze większą liczbą potencjalnych stanów jest konsola do gier Atari 2600. Była ona bardzo popularna w latach 80. XX wieku i zawierała wiele gier zręcznościowych. Jak na dzisiejsze standardy jest ona już archaiczna, ale jej gry stanowią doskonały zestaw zadań związanych z uczeniem przez wzmacnianie. Ludzie potrafią je dość szybko rozwiązywać, a z drugiej strony dla komputerów wciąż stanowią wyzwanie. Nic dziwnego, że ta platforma gier (w postaci emulatora) jest bardzo popularnym punktem odniesienia dla naukowców zajmujących się uczeniem przez wzmacnianie.

Obliczmy przestrzeń stanów dla platformy Atari. Rozdzielczość ekranu wynosi 210×160 pikseli, a każdy z nich może mieć jeden ze 128 kolorów. Tak więc każda klatka ma 210×160 = 33 600 pikseli, a liczba różnych ekranów wynosi 128^{33600}, czyli nieco więcej niż 10^{70802}. Gdybyśmy chcieli tylko raz wyliczyć wszystkie możliwe stany Atari, nawet najszybszemu superkomputerowi zajęłoby to miliardy miliardów lat. Ponadto 99,99% tej pracy byłoby stratą czasu, ponieważ większość kombinacji nigdy nie zostałaby wyświetlona podczas nawet długiej rozgrywki. Chociaż nigdy nie wykorzystywalibyśmy próbek niepotrzebnych stanów, metoda iteracji wartości i tak „na wszelki wypadek" chciałaby je przetwarzać.

Inny problem związany z podejściem wykorzystującym iterację wartości polega na tym, że jesteśmy ograniczeni do zajmowania się dyskretnymi przestrzeniami akcji. Zarówno w przypadku przybliżeń $Q(s, a)$, jak i $V(s)$ istnieje założenie, że akcje należą do zbioru wzajemnie wykluczających

się danych dyskretnych. Przybliżenia te nie mogą więc zostać zastosowane podczas rozwiązywania problemów dotyczących sterowania ciągłego, w których akcje dotyczą zmiennych z wartościami ciągłymi, takich jak kąt skrętu kierownicy, siła elementu napędzającego lub temperatura grzejnika. Sposoby rozwiązywania takich trudnych zagadnień omówię w ostatniej części książki, w rozdziałach poświęconych problemom związanym z ciągłymi przestrzeniami akcji. Na razie załóżmy, że mamy pewną liczbę akcji i nie jest ona zbyt duża (w granicach 10). Jak powinniśmy poradzić sobie z rozmiarem przestrzeni stanów?

Q-uczenie tabelaryczne

Zastanówmy się przede wszystkim, czy naprawdę musimy przetwarzać każdy stan w przestrzeni stanów. Mamy środowisko, które może być źródłem rzeczywistych próbek stanów. Jeśli jakiś stan w przestrzeni stanów nie jest udostępniany przez środowisko, to dlaczego mamy się interesować jego wartością? Za pomocą informacji uzyskanych ze środowiska możemy zaktualizować wartości stanów, co może zaoszczędzić wiele pracy.

Jak wspomniałem wcześniej, ta modyfikacja metody iteracji wartości jest znana jako Q-uczenie. W przypadku jawnego odwzorowywania stanu na wartość algorytm przedstawia się następująco:

1. Zacznij od pustej tabeli i odwzoruj stany na wartości akcji.
2. Dzięki interakcji ze środowiskiem uzyskaj krotkę s, a, r, s' (stan, akcja, nagroda i nowy stan). Na tym etapie musisz zdecydować, jaką akcję należałoby podjąć. Niestety, nie istnieje gotowe rozwiązanie, które można byłoby zastosować. Wspominałem już o tym problemie na początku książki, a także poświęcę mu dużo miejsca w tym rozdziale.
3. Zaktualizuj wartość $Q(s, a)$ za pomocą przybliżenia Bellmana:

$$Q(s,a) \leftarrow r + \gamma \max_{a' \in A} Q(s', a')$$

4. Wróć do kroku 2.

Podobnie jak w przypadku iteracji wartości, warunkiem końcowym może być przekroczenie pewnej wartości progowej. Możemy również wykonać epizody testowe, aby oszacować oczekiwaną nagrodę wynikającą z polityki.

Należy także zwrócić uwagę na sposób aktualizowania wartości Q. Próbki pobieramy ze środowiska, dlatego złym pomysłem jest podmienianie istniejących wartości nowymi, ponieważ proces trenowania może stać się niestabilny.

W praktyce aktualizację $Q(s, a)$ wykonuje się zazwyczaj przy użyciu techniki „łączenia", która polega na uśrednianiu starych i nowych wartości Q za pomocą współczynnika uczenia α z zakresu od 0 do 1:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{a' \in A} Q(s', a'))$$

Dzięki temu nawet w przypadku, gdy środowisko jest zakłócone, wartość Q może zostać poprawnie oszacowana. Oto ostateczna wersja algorytmu:

1. Rozpocznij od pustej tabeli wartości $Q(s, a)$.
2. Uzyskaj ze środowiska krotkę (s, a, r, s').
3. Wykonaj aktualizację Bellmana:
   $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{a' \in A} Q(s', a'))$.
4. Sprawdź warunek konwergencji. Jeśli nie jest spełniony, powtórz czynności od kroku 2.

Jak wspomniałem wcześniej, ta metoda jest zwana Q-uczeniem tabelarycznym, ponieważ stosujemy tabelę stanów z wartościami Q. Wypróbujmy ją w środowisku FrozenLake. Cały kod programu można znaleźć pod adresem *https://ftp.helion.pl/przyklady/glucz2.zip* (plik *r06/01_frozenlake_q_learning.py*).

```
import gym
import collections
from tensorboardX import SummaryWriter

ENV_NAME = "FrozenLake-v0"
GAMMA = 0.9
ALPHA = 0.2
TEST_EPISODES = 20

class Agent:
    def __init__(self):
        self.env = gym.make(ENV_NAME)
        self.state = self.env.reset()
        self.values = collections.defaultdict(float)
```

Na początku importujemy pakiety i definiujemy stałe. Nowością jest wprowadzenie wartości α, która będzie używana jako współczynnik uczenia podczas aktualizacji wartości. Inicjalizacja klasy `Agent` jest teraz prostsza, ponieważ nie musimy śledzić historii nagród i liczników przejść, a jedynie tabelę wartości. Zmniejszy to zużycie pamięci, co nie jest dużym problemem w świecie FrozenLake, ale może się okazać krytyczną kwestią w przypadku większych środowisk.

```
    def sample_env(self):
        action = self.env.action_space.sample()
        old_state = self.state
        new_state, reward, is_done, _ = self.env.step(action)
        self.state = self.env.reset() if is_done else new_state
        return old_state, action, reward, new_state
```

Powyższa metoda umożliwia wyznaczenie kolejnego przejścia w środowisku. Z przestrzeni akcji wybieramy losową akcję i zwracamy krotkę zawierającą poprzedni stan, wykonaną akcję, uzyskaną nagrodę i nowy stan. Krotka ta zostanie później użyta w pętli treningowej.

```
    def best_value_and_action(self, state):
        best_value, best_action = None, None
        for action in range(self.env.action_space.n):
            action_value = self.values[(state, action)]
```

```
            if best_value is None or best_value < action_value:
                best_value = action_value
                best_action = action
        return best_value, best_action
```

Kolejna metoda odczytuje stan środowiska i wybiera najlepszą akcję do wykonania poprzez wyszukanie największej wartości, jaka istnieje w tabeli. Jeśli z parą stan/akcja nie jest związana żadna wartość, zakładamy, że jest równa zeru. Ta metoda zostanie użyta dwa razy: po pierwsze w metodzie testowej, która uruchamia epizod przy użyciu tabeli z bieżącymi wartościami (w celu oceny jakości polityki), a po drugie w metodzie aktualizującej wartości w celu uzyskania danych dotyczących następnego stanu.

```
    def value_update(self, s, a, r, next_s):
        best_v, _ = self.best_value_and_action(next_s)
        new_v = r + GAMMA * best_v
        old_v = self.values[(s, a)]
        self.values[(s, a)] = old_v * (1-ALPHA) + new_v * ALPHA
```

Wykonujemy krok w środowisku i aktualizujemy tabelę wartości. Dodajemy nagrodę natychmiastową do zdyskontowanej wartości następnego stanu i w ten sposób obliczamy równanie Bellmana dla stanu *s* oraz akcji *a*. Uzyskujemy wartości dla poprzedniego stanu i akcji, a następnie łączymy je z nowymi przy użyciu współczynnika uczenia. W wyniku otrzymujemy kolejne przybliżenie wartości stanu *s* i akcji *a*, które zapisujemy w tabeli.

```
    def play_episode(self, env):
        total_reward = 0.0
        state = env.reset()
        while True:
            _, action = self.best_value_and_action(state)
            new_state, reward, is_done, _ = env.step(action)
            total_reward += reward
            if is_done:
                break
            state = new_state
        return total_reward
```

Ostatnia metoda klasy `Agent` wykonuje pełny epizod, wykorzystując w tym celu dostarczone środowisko testowe. To, jaka akcja jest podejmowana w każdym z kroków, wynika z danych zawartych w tabeli wartości Q. Metoda pozwala ocenić bieżącą politykę w celu sprawdzenia postępów w nauce. Zauważ, że nie modyfikuje ona tabeli wartości, a jedynie używa jej do wyszukania najlepszej akcji do wykonania.

Reszta kodu to pętla treningowa, która jest bardzo podobna do przykładów z rozdziału 5. Najpierw tworzymy środowisko testowe, agenta i moduł przesyłający dane do narzędzia TensorBoard. Następnie w pętli wykonujemy krok w środowisku i aktualizujemy wartości na podstawie uzyskanych danych. Poprzez wykonanie kilku etapów testowych sprawdzamy, jaka jest bieżąca polityka. Jeśli uzyskamy odpowiednio dobrą nagrodę, kończymy trenowanie.

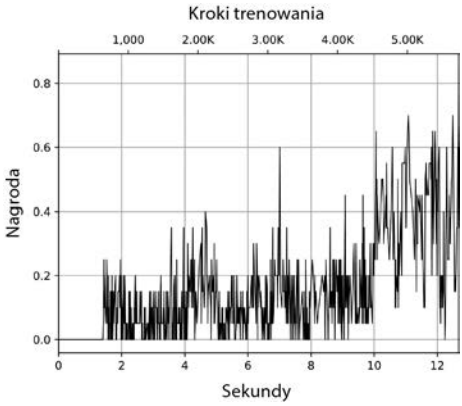
```
if __name__ == "__main__":
    test_env = gym.make(ENV_NAME)
    agent = Agent()
```

```
writer = SummaryWriter(comment="-q-learning")

iter_no = 0
best_reward = 0.0
while True:
    iter_no += 1
    s, a, r, next_s = agent.sample_env()
    agent.value_update(s, a, r, next_s)

    reward = 0.0
    for _ in range(TEST_EPISODES):
        reward += agent.play_episode(test_env)
    reward /= TEST_EPISODES
    writer.add_scalar("reward", reward, iter_no)
    if reward > best_reward:
        print("Nagroda uległa zmianie: %.3f -> %.3f" % (
            best_reward, reward))
        best_reward = reward
    if reward > 0.80:
        print("Rozwiązano w %d iteracjach!" % iter_no)
        break
writer.close()
```

Oto uzyskane wyniki:

```
r06$./01_frozenlake_q_learning.py
Nagroda uległa zmianie: 0.000 -> 0.150
Nagroda uległa zmianie: 0.150 -> 0.250
Nagroda uległa zmianie: 0.250 -> 0.300
Nagroda uległa zmianie: 0.300 -> 0.350
Nagroda uległa zmianie: 0.350 -> 0.400
Nagroda uległa zmianie: 0.400 -> 0.450
Nagroda uległa zmianie: 0.450 -> 0.550
Nagroda uległa zmianie: 0.550 -> 0.600
Nagroda uległa zmianie: 0.600 -> 0.650
Nagroda uległa zmianie: 0.650 -> 0.700
Nagroda uległa zmianie: 0.700 -> 0.750
Nagroda uległa zmianie: 0.750 -> 0.800
Nagroda uległa zmianie: 0.800 -> 0.850
Rozwiązano w 5738 iteracjach!
```

Jak widać, aby rozwiązać problem, program musiał działać dłużej w porównaniu z metodą iteracji wartości z poprzedniego rozdziału. Wynika to stąd, że nie korzystamy już z doświadczenia zdobytego podczas testów (w przykładzie *r05/02_frozenlake_q_iteration.py* okresowe testy aktualizowały statystyki tabeli Q, natomiast w przypadku obecnego kodu nie modyfikowaliśmy wartości Q, dzięki czemu trzeba było wykonać więcej iteracji). Ogólnie rzecz ujmując, liczba próbek pobieranych ze środowiska jest prawie taka sama jak poprzednio. Wykres wartości nagrody również potwierdza dobrą dynamikę trenowania, bardzo podobną do tej, którą otrzymaliśmy w metodzie iteracji wartości (rysunek 6.1).



Rysunek 6.1. Dynamika nagradzania w środowisku FrozenLake

Głębokie Q-uczenie

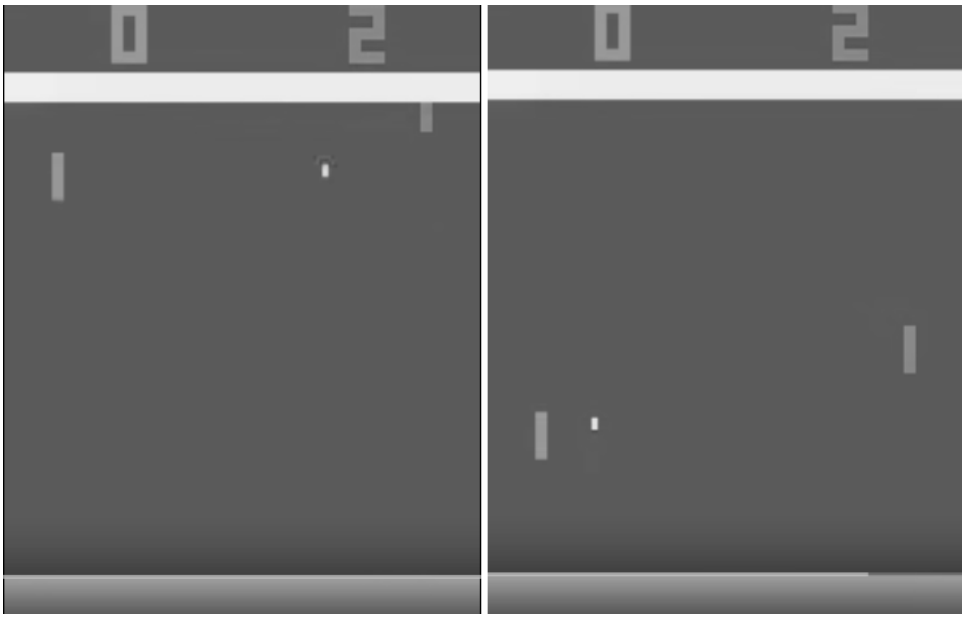
Metoda Q-uczenia, którą właśnie przeanalizowaliśmy, rozwiązuje problem przetwarzania całego zbioru stanów. Nadal jednak możemy się zmagać z sytuacjami, w których liczba obserwowalnych stanów jest bardzo duża. Na przykład gry Atari mogą mieć wiele różnych ekranów, więc jeśli zdecydujemy się użyć zwykłych pikseli do reprezentowania pojedynczych stanów, szybko się zorientujemy, że istnieje ich zbyt wiele, by można je było monitorować i wyznaczać ich wartości.

W niektórych środowiskach liczba obserwowalnych stanów może być prawie nieskończona. Na przykład w przypadku CartPole stan składa się z czterech liczb zmiennoprzecinkowych. Liczba kombinacji wartości jest co prawda skończona (są one przedstawiane jako bity), ale niezwykle duża. Moglibyśmy stworzyć kilka zakresów, aby skategoryzować te wartości, ale często takie podejście stwarza więcej problemów, niż ich rozwiązuje. Musielibyśmy na przykład zdecydować, które zakresy parametrów warto rozróżniać w postaci odmiennych stanów, a które grupować razem.

W przypadku Atari jedna zmiana piksela nie robi dużej różnicy, dlatego w podobnych przypadkach warto traktować oba obrazy jako jeden stan. Jednak wciąż musimy rozróżniać wiele stanów.

Na rysunku 6.2 przedstawiłem dwie różne sytuacje w grze Pong. Gramy przeciwko sztucznej inteligencji i sterujemy ruchem paletki (nasza jest po prawej stronie, a przeciwnika po lewej). Celem gry jest odbicie piłki i sprawienie, by znalazła się za paletką przeciwnika, a jednocześnie niedopuszczenie, by przedostała się za naszą paletkę. Możemy uznać dwie przedstawione sytuacje za zupełnie różne. Po prawej stronie piłka jest blisko przeciwnika, więc możemy się zrelaksować i jedynie obserwować, co się dzieje. Jednak sytuacja po lewej jest bardziej wymagająca — zakładając, że piłka porusza się od lewej do prawej, widzimy, że znajduje się ona blisko naszej pozycji, więc musimy szybko przesunąć paletkę, aby nie stracić punktu.

Sytuacje przedstawione na rysunku 6.2 to tylko dwie z 10^{70802} możliwych. Chcielibyśmy jednak, aby agent je rozpoznawał i odpowiednio się zachowywał.



Rysunek 6.2. Niejednoznaczność obserwacji w grze Pong. Na lewym ekranie piłka porusza się w prawo w kierunku naszej paletki, a na prawym w kierunku przeciwnym

Rozwiązaniem problemu może być użycie reprezentacji nieliniowej, która odwzorowuje na wartość zarówno stan, jak i akcję. Taki sposób jest zwany w uczeniu maszynowym analizą regresji. Co prawda poszczególne algorytmy mogą się od siebie różnić, ale jak można się domyślić na podstawie tytułu tego podrozdziału, użycie głębokich sieci neuronowych jest jednym z najczęściej spotykanych rozwiązań — szczególnie w przypadku obserwacji reprezentowanych w postaci ekranów. Dokonajmy więc odpowiedniej modyfikacji w algorytmie Q-uczenia:

1. Zainicjalizuj $Q(s, a)$ za pomocą początkowego przybliżenia.
2. Poprzez interakcję ze środowiskiem uzyskaj krotkę (s, a, r, s').
3. Jeśli epizod się zakończył, oblicz stratę, korzystając ze wzoru $\mathcal{L} = (Q(s,a) - r)^2$. W przeciwnym razie użyj wzoru $\mathcal{L} = \left(Q(s,a) - (r + \gamma \max_{a' \in A} Q_{s',a'})\right)^2$.
4. Zaktualizuj $Q(s, a)$ za pomocą algorytmu **stochastycznego spadku wzdłuż gradientu** (**SGD**), dzięki czemu zminimalizujesz stratę zależną od parametrów modelu.
5. Powtarzaj cały proces od kroku 2. aż do uzyskania stabilnych wyników.

Algorytm wydaje się być prosty, ale niestety nie działa zbyt dobrze. Zastanówmy się nad tym, jakie problemy mogą się pojawić.

Interakcja ze środowiskiem

Aby otrzymywać dane do trenowania, musimy wchodzić w interakcję ze środowiskiem. W prostych środowiskach, takich jak FrozenLake, możemy działać losowo, ale czy taka strategia jest optymalna? Rozważmy grę w Ponga. Jakie jest prawdopodobieństwo zdobycia jednego punktu poprzez losowe przesunięcie paletki? Nie jest zerowe, ale jednak bardzo małe, co po prostu oznacza, że taka sytuacja zdarza się bardzo rzadko. Akcje można też alternatywnie wybierać za pomocą przybliżenia funkcji Q (tak jak zrobiliśmy wcześniej w metodzie iteracji wartości, gdy zapamiętaliśmy doświadczenia z fazy testowej).

Jeśli reprezentacja wartości Q jest poprawna, zdobyte doświadczenie pozwoli na uzyskanie odpowiednich danych do trenowania. Gdy jednak przybliżenie nie będzie idealne (na przykład na początku procesu trenowania), pojawią się problemy. W takim przypadku agent może się zapętlić w niektórych stanach i wciąż wykonywać błędne akcje. Jest to dylemat *eksploracja kontra eksploatacja*, o którym krótko wspomniałem w rozdziale 1., a także na początku tego. Z jednej strony agent musi zbadać środowisko, aby utworzyć pełny obraz przejść i wyników uzyskanych po przeprowadzeniu akcji. Z drugiej strony należy efektywnie wykorzystywać interakcję ze środowiskiem — nie powinno się tracić czasu na przypadkowe testowanie akcji, które już zostały sprawdzone i powiązane z odpowiednimi wynikami.

Jak widać, przypadkowe działania są lepsze na początku procesu trenowania, gdy przybliżenie Q jest niedokładne, ponieważ dzięki temu uzyskujemy bardziej równomiernie rozłożone informacje o stanach środowiska. W miarę upływu czasu losowe zachowanie staje się jednak nieskuteczne. Chcemy więc powrócić do przybliżenia Q, aby zdecydować, jak należy postępować.

Algorytm, który łączy w sobie te dwa skrajne sposoby działań, jest znany jako **metoda epsilonu zachłannego**. Oznacza on po prostu wybór pomiędzy polityką losową a funkcją Q przy użyciu hiperparametru prawdopodobieństwa ε. Dzięki zmianie wartości ε możemy decydować, jak dużo powinno być losowych akcji. Zazwyczaj proces rozpoczyna się od $\varepsilon = 1{,}0$ (100% losowych akcji), a następnie powoli zmniejsza się jego wartość do przykładowo 5% lub 2% przypadkowych akcji. Metoda epsilonu zachłannego bada środowisko na początku trenowania, a pod jego koniec umożliwia stosowanie właściwej polityki. Istnieją też inne sposoby rozwiązywania problemu eksploracja kontra eksploatacja — niektóre z nich omówię w trzeciej części książki. W uczeniu przez wzmacnianie problem ten jest jedną z podstawowych kwestii spornych i nie został jeszcze wyjaśniony mimo badania przez wielu naukowców.

Optymalizacja za pomocą stochastycznego spadku wzdłuż gradientu (SGD)

Podstawa procedury Q-uczenia została zapożyczona z uczenia nadzorowanego. Próbujemy przybliżyć złożoną, nieliniową funkcję $Q(s, a)$ za pomocą sieci neuronowej. W tym celu musimy obliczyć cele dla tej funkcji za pomocą równania Bellmana, a następnie założyć, że mamy do rozwiązania problem dotyczący uczenia nadzorowanego. Mimo że takie podejście jest poprawne, należy pamiętać o jednym z podstawowych wymagań optymalizacji SGD — dane treningowe powinny być *niezależne i identycznie dystrybuowane*.

W tym przypadku dane, których będziemy używać do optymalizacji SGD, nie spełniają powyższych kryteriów:

1. Próbki nie są niezależne. Nawet jeśli zgromadzimy dużą partię próbek danych, wszystkie będą bardzo związane ze sobą, ponieważ zostaną uzyskane podczas tego samego epizodu.
2. Dystrybucja danych treningowych nie będzie taka sama jak w przypadku próbek udostępnianych przez optymalną politykę, której agent powinien się nauczyć. Dane, które mamy, będą wynikiem innej polityki (funkcji Q, polityki losowej lub obu z nich w przypadku metody epsilonu zachłannego). Nie chcielibyśmy jednak nauczyć agenta, jak powinien się losowo zachowywać — chcemy optymalnej polityki z najlepszą nagrodą.

Aby poradzić sobie z tym problemem, nie możemy korzystać z najnowszych doświadczeń. Zamiast tego dane treningowe powinniśmy pobierać z dużego **bufora** zawierającego informacje zebrane wcześniej przez agenta. Najprostszą implementacją jest bufor o stałym rozmiarze, w którym nowe dane są dodawane na końcu, przy jednoczesnym usuwaniu doświadczeń najstarszych.

Bufor pozwala trenować przy użyciu mniej lub bardziej niezależnych danych. Będą one jednak nadal wystarczająco świeże, aby mogły zostać wykorzystane do trenowania próbek wygenerowanych przez najnowszą politykę. W następnym rozdziale przeanalizuję bufor z priorytetami, który udostępnia bardziej zaawansowaną metodę próbkowania.

Korelacja pomiędzy krokami

Kolejna kwestia praktyczna dotycząca domyślnej procedury trenowania także jest związana z brakiem danych niezależnych i identycznie dystrybuowanych, jednak w tym przypadku chodzi o coś innego. Równanie Bellmana dzięki $Q(s', a')$ dostarcza wartości $Q(s, a)$ (proces ten nazywa się **bootstrappingiem**). Jak pamiętasz, stany s i s' zostały osiągnięte w kolejnych krokach. To sprawia, że są do siebie bardzo podobne, więc sieć neuronowa ma problem, by je rozróżnić. Gdy aktualizujemy parametry sieci neuronowej, aby przybliżyć wartość $Q(s, a)$ do pożądanego wyniku, możemy pośrednio także zmienić wartość generowaną przez $Q(s', a')$ i inne pobliskie stany. Może to spowodować bardzo niestabilne trenowanie i przyczynić się do tego, że model będzie się „kręcić w kółko". Po zaktualizowaniu funkcji Q dla stanu s okaże się, że wartość $Q(s', a')$ staje się coraz gorsza, a kolejne próby mogą jeszcze bardziej popsuć przybliżenia $Q(s, a)$.

Aby uczynić trening bardziej stabilnym, można zastosować sztuczkę zwaną **siecią docelową**. Umożliwia ona przechowywanie kopii sieci i używanie jej jako wartości $Q(s', a')$ w równaniu Bellmana. Ta sieć jest tylko okresowo synchronizowana z właściwą siecią, na przykład co N kroków (przy czym N jest zwykle hiperparametrem o dużej wartości, równym 1000 lub 10 000 iteracjom treningowym).

Własność Markowa

Zaprezentowane metody uczenia przez wzmacnianie wykorzystują procesy decyzyjne Markowa, które zakładają, że środowisko jest zgodne z własnością Markowa. Oznacza, to obserwacje ze środowiska są wszystkim, czego potrzebujemy, aby działać optymalnie (innymi słowy, obserwacje pozwalają rozróżniać stany).

Jak można było zauważyć na rysunku 6.2, jeden obraz z gry Atari nie wystarczy do uzyskania wszystkich ważnych informacji (zauważ, że nie mamy pojęcia o prędkości i kierunku ruchu obiektów takich jak piłka i paletka przeciwnika). To oczywiście nie spełnia własności Markowa i sprawia, że środowisko z pojedynczym zrzutem ekranu z gry Pong staje się **częściowo obserwowalnym procesem decyzyjnym Markowa**. Jest to w zasadzie zwykły proces decyzyjny Markowa pozbawiony własności Markowa. Ma on duże znaczenie w praktycznych zastosowaniach. Na przykład w przypadku większości gier karcianych, w których nie widać kart przeciwników, mamy do czynienia z częściowo obserwowalnymi procesami decyzyjnymi Markowa, ponieważ bieżące obserwacje (Twoje karty oraz karty leżące na stole) mogą odpowiadać różnym kartom w rękach przeciwników.

W tej książce nie będziemy szczegółowo analizować częściowo obserwowalnych procesów decyzyjnych Markowa. Użyjemy jednak pewnego sposobu, dzięki któremu środowisko znajdzie się z powrotem w domenie zwykłych procesów decyzyjnych Markowa. Rozwiązanie polega na zachowaniu kilku obserwacji z przeszłości i wykorzystaniu ich jako stanu. W przypadku gier Atari zazwyczaj gromadzimy k kolejnych klatek, a następnie używamy ich jako obserwacji w każdym ze stanów. Pozwala to agentowi na ustalenie dynamiki bieżącego stanu, na przykład w celu uzyskania informacji o prędkości piłki i kierunku. Liczba k dla Atari jest zazwyczaj równa 4. Oczywiście to tylko pewna sztuczka, ponieważ w środowisku mogą występować dłuższe zależności, ale dla większości gier działa ona dobrze.

Ostateczna wersja procedury trenowania dla głębokich sieci Q

Istnieje wiele innych sposobów, które odkryli naukowcy, aby uczynić proces trenowania głębokich sieci Q bardziej stabilnym i wydajnym. Najciekawsze zostaną przeanalizowane w następnym rozdziale. Metoda epsilonu zachłannego, bufor i sieć docelowa zostały wykorzystane przez firmę DeepMind zajmującą się sztuczną inteligencją do wytrenowania sieci DQN przy użyciu zestawu 49 gier Atari i zademonstrowania skuteczności tego rozwiązania w przypadku skomplikowanych środowisk.

W oryginalnym artykule naukowym (nieuwzględniającym sieci docelowej) wykorzystano siedem gier. Został on opublikowany pod koniec 2013 roku (Mnih i in., *Playing Atari with Deep Reinforcement Learning*, 1312.5602v1). Później, na początku 2015 roku, poprawiona wersja artykułu, wykorzystująca 49 różnych gier, została opublikowana w czasopiśmie naukowym „Nature" (Mnih i in., *Human-Level Control Through Deep Reinforcement Learning*, doi:10.1038/nature14236).

Algorytm trenowania dla głębokich sieci Q, zaprezentowany w powyższych artykułach, składa się z następujących kroków:

1. Zainicjalizuj parametry dla $Q(s,a)$ i $\hat{Q}(s,a)$ przy użyciu losowych wag ($\varepsilon \leftarrow 1{,}0$), a następnie opróżnij bufor.
2. Z prawdopodobieństwem równym ε wybierz losową akcję a. W przeciwnym razie użyj wzoru $a = \arg\max_a Q(s,a)$.
3. Wykonaj akcję a w emulatorze i obserwuj nagrodę r oraz następny stan s'.
4. Zapisz przejście (s,a,r,s') w buforze.
5. Pobierz losową minipaczkę przejść z bufora.
6. Jeśli epizod zakończył się w tym kroku, dla każdego z przejść oblicz wartość docelową $y = r$. W przeciwnym razie użyj wzoru $y = r + \gamma \max_{a' \in A} \hat{Q}(s',a')$.
7. Oblicz stratę: $\mathcal{L} = (Q(s,a) - y)^2$.
8. Zaktualizuj wartość $Q(s,a)$ za pomocą algorytmu SGD poprzez zminimalizowanie straty w odniesieniu do parametrów modelu.
9. Co N kroków kopiuj wagi z Q do $\hat{Q}$.
10. Powtarzaj proces od punktu 2. aż do osiągnięcia stabilnych wyników.

Zaimplementujmy teraz ten algorytm i spróbujmy wygrać w niektórych grach Atari.

Użycie głębokiej sieci Q w grze Pong

Przed prezentacją kodu konieczne jest pewne wprowadzenie. Przykłady będą coraz bardziej wymagające i złożone, co nie jest zaskakujące, ponieważ rośnie również złożoność problemów, które próbujemy rozwiązać. Mimo że kody przykładów są tak proste i zwięzłe, jak tylko możliwe, niektóre ich fragmenty mogą być na początku trudne do zrozumienia.

Kolejnym ważnym zagadnieniem jest wydajność. Poprzednie przykłady dotyczące światów FrozenLake i CartPole nie były wymagające z punktu widzenia wydajności, ponieważ złożoność środowisk oraz sieci neuronowych była niewielka. Skrócenie trwania iteracji pętli treningowej o kilka milisekund nie było więc koniecznie. Teraz jednak sytuacja uległa zmianie. Jedna obserwacja ze środowiska Atari to około 100 tysięcy różnych wartości, które należy przeskalować, przekonwertować na liczby zmiennoprzecinkowe i zapisać w buforze. Kopiowanie tablicy z takimi danymi może być kosztowne i sprawić, że nawet w przypadku dostępu do szybkiego procesora graficznego trenowanie nie będzie już liczone w minutach, ale w godzinach.

Wąskim gardłem może być również pętla treningowa sieci neuronowej. Oczywiście modele używane w uczeniu ze wzmacnianiem nie są tak ogromnymi konstrukcjami jak najnowocześniejsze sieci neuronowe ImageNet. Należy jednak pamiętać, że starszy model głębokiej sieci Q z 2015 roku zawiera ponad 1,5 mln parametrów, co nawet dla nowoczesnego procesora graficznego wciąż stanowi pewien problem. Krótko mówiąc, wydajność ma znaczenie, zwłaszcza gdy eksperymentujesz z hiperparametrami i musisz wytrenować dziesiątki modeli.

Biblioteka PyTorch jest dość przystępna, więc kod może wyglądać o wiele mniej tajemniczo niż zoptymalizowane wykresy TensorFlow. Nadal jednak można za jej pomocą napisać program, który będzie działać zbyt wolno lub, co gorsza, błędnie. Na przykład proste rozwiązanie, które wyznacza stratę w sieci DQN poprzez przetwarzanie każdej paczki w pętli, działa około dwa razy wolniej niż bardziej zaawansowana wersja współbieżna. Pamiętaj jednak, że pojedyncza instrukcja kopiowania paczki danych może spowolnić kod aż 13 razy, co staje się już dość istotnym problemem.

Kod obecnego przykładu został podzielony na trzy moduły ze względu na jego wielkość, strukturę logiczną i możliwość ponownego wykorzystania. Oto one:

- *r06/lib/wrappers.py* — kod opakowujący środowisko Atari, w większości zaczerpnięty z projektu *Baselines*, którego twórcą jest laboratorium badawcze OpenAI.
- *r06/lib/dqn_model.py* — warstwa sieci neuronowej o tej samej architekturze co głęboka sieć Q zaprezentowana w artykule „Nature".
- *r06/02_dqn_pong.py* — główny moduł zawierający pętlę treningową, wyznaczanie funkcji straty i definicję bufora.

Opakowania

Granie w gry Atari przy użyciu uczenia ze wzmacnianiem jest dość wymagające z punktu widzenia zasobów. Aby przyspieszyć działanie, zastosowano kilka transformacji, które zostały przedstawione we wspomnianym wcześniej artykule. Niektóre z nich mają wpływ tylko na wydajność, zaś inne korygują pewne cechy platformy Atari, które sprawiają, że trenowanie trwa długo i jest niestabilne. Transformacje są zwykle implementowane jako opakowania za pomocą biblioteki OpenAI Gym. Pełna ich lista jest dość długa, a poza tym istnieje kilka różnych implementacji tych samych opakowań. Moim ulubionym źródłem kodów jest repozytorium OpenAI Baselines, które zawiera zestaw metod i algorytmów uczenia przez wzmacnianie zaimplementowanych przy użyciu biblioteki TensorFlow. Zostały one zastosowane w popularnych narzędziach do testów porównawczych w celu ustalenia punktu odniesienia pozwalającego na porównywanie metod. Repozytorium jest dostępne pod adresem *https://github.com/openai/baselines*, a opakowania w pliku *https://github.com/openai/baselines/blob/master/baselines/common/atari_wrappers.py*.

Oto lista najpopularniejszych transformacji Atari używanych przez naukowców zajmujących się dziedziną uczenia przez wzmacnianie:

- *Przekształcanie wskrzeszeń w oddzielne epizody*. Ogólnie rzecz ujmując, epizod zawiera wszystkie kroki wykonane przez gracza od rozpoczęcia gry do pojawienia się ekranu informującego o jej zakończeniu. Taki epizod może się składać z tysięcy kroków (obserwacji i akcji). Zwykle w grach zręcznościowych gracz otrzymuje pewną liczbę wskrzeszeń, które pozwalają mu na wykonanie kilku prób. Transformacja dzieli cały epizod na mniejsze okresy, odpowiadające pojedynczym wskrzeszeniom gracza. Nie wszystkie gry udostępniają taką możliwość (nie istnieje ona na przykład w grze Pong), jednak w dostępnych środowiskach ta transformacja zwykle przyspiesza uzyskanie stabilnych wyników, ponieważ epizody stają się krótsze.

- *Wykonywanie losowej liczby (do 30) pustych akcji na początku gry*. Dzięki temu w niektórych grach Atari można pominąć wstępne ekrany, które nie są istotne dla rozgrywki.
- *Podejmowanie co* K *kroków decyzji o wykonaniu akcji, gdzie* K *wynosi zwykle 4 lub 3*. W klatkach pośrednich wybrana akcja jest po prostu powtarzana. Pozwala to na znaczne przyspieszenie trenowania, ponieważ przetwarzanie każdej klatki za pomocą sieci neuronowej jest dość wymagającą operacją, a różnice między kolejnymi klatkami są zwykle niewielkie.
- *Wyznaczanie wartości maksymalnych dla pikseli pochodzących z ostatnich dwóch klatek i wykorzystywanie tych wartości jako obserwacji*. Niektóre gry Atari powodują migotanie, co jest spowodowane ograniczeniami platformy (Atari ma ograniczoną liczbę sprajtów, które można wyświetlić w pojedynczej klatce). Dla ludzkiego oka takie szybkie zmiany nie są widoczne, ale mogą zdezorientować sieć neuronową.
- *Naciskanie przycisku* FIRE *na początku gry*. Niektóre gry (w tym Pong i Breakout) wymagają od użytkownika naciśnięcia przycisku *FIRE*, aby rozpocząć grę. Bez wykonania tej akcji środowisko staje się częściowo obserwowalnym procesem decyzyjnym Markowa, ponieważ na podstawie samej obserwacji agent nie może stwierdzić, czy przycisk *FIRE* został już wciśnięty.
- *Skalowanie w dół każdej klatki z trzema kanałami kolorów i o rozdzielczości 210×160 pikseli do obrazu z odcieniami szarości i rozdzielczości 84×84 pikseli*. Dostępne są różne rozwiązania. Na przykład w artykule opisano tę transformację jako pobranie kanału Y z przestrzeni kolorów YCbCr, a następnie przeskalowanie całego obrazu do rozdzielczości 84×84. Inni naukowcy usuwają kolory z obrazu, wycinają jego nieistotne części, a następnie go zmniejszają. W repozytorium Baselines (i w kodzie przykładu) zastosowano to drugie podejście.
- *Zestawianie kilku (zwykle czterech) kolejnych klatek, aby sieć mogła otrzymać informacje o dynamice obiektów gry*. Ta metoda została wcześniej przedstawiona jako rozwiązanie w przypadku braku informacji o dynamice gry w pojedynczej klatce.
- *Ograniczanie nagród do wartości –1, 0 i 1*. Uzyskany wynik może się znacznie różnić w zależności od gry. Na przykład w grze Pong otrzymujesz 1 punkt za każdą piłkę, której przeciwnik nie zdoła odbić. Jednak w niektórych grach, takich jak KungFuMaster, za każdego zabitego wroga otrzymasz nagrodę w wysokości 100 punktów. Ta rozpiętość nagród sprawia, że w każdej grze strata ma zupełnie inną skalę wartości, co utrudnia znajdowanie wspólnych hiperparametrów w przypadku całego zestawu gier. Aby rozwiązać ten problem, nagrodę po prostu ogranicza się do zakresu `[-1…1]`.
- *Konwersja wartości obserwacji będących bajtami o typie* `unsigned` *na typ* `float32`. Zrzut ekranu z emulatora jest kodowany jako tensor bajtów o wartościach od 0 do 255, co nie jest najlepszą reprezentacją w przypadku sieci neuronowej. Musimy więc przekonwertować obraz na liczby zmiennoprzecinkowe, a następnie przeskalować uzyskane wartości do zakresu `[0.0…1.0]`.

W przypadku gry Pong, którą za chwilę przeanalizujemy, nie potrzebujemy niektórych opakowań, takich jak przekształcanie wskrzeszeń w oddzielne epizody czy ograniczanie nagród, więc nie zostały one uwzględnione w kodzie. Na wypadek, gdybyś zdecydował się poeksperymentować

z innymi grami, powinieneś jednak być świadomy ich istnienia. Gdy głęboka sieć Q nie wykazuje zbieżnej tendencji, czasami przyczyną jest nieprawidłowo opakowane środowisko, a nie błędny kod. Musiałem poświęcić kilka dni, aby rozwiązać problem, który wynikał z braku naciśnięcia przycisku *FIRE* na początku gry!

Przyjrzyjmy się implementacji opakowań, których kod jest dostępny w pliku *r06/lib/wrappers.py*:

```
import cv2
import gym
import gym.spaces
import numpy as np
import collections

class FireResetEnv(gym.Wrapper):
    def __init__(self, env=None):
        super(FireResetEnv, self).__init__(env)
        assert env.unwrapped.get_action_meanings()[1] == 'FIRE'
        assert len(env.unwrapped.get_action_meanings()) >= 3

    def step(self, action):
        return self.env.step(action)

    def reset(self):
        self.env.reset()
        obs, _, done, _ = self.env.step(1)
        if done:
            self.env.reset()
        obs, _, done, _ = self.env.step(2)
        if done:
            self.env.reset()
        return obs
```

Powyżej przedstawiony kod powoduje naciśnięcie przycisku *FIRE* w środowiskach, które wymagają tego działania do rozpoczęcia gry. Oprócz tego sprawdza kilka skrajnych przypadków, które mogą się pojawić w niektórych grach.

```
class MaxAndSkipEnv(gym.Wrapper):
    def __init__(self, env=None, skip=4):
        super(MaxAndSkipEnv, self).__init__(env)
        self._obs_buffer = collections.deque(maxlen=2)
        self._skip = skip

    def step(self, action):
        total_reward = 0.0
        done = None
        for _ in range(self._skip):
            obs, reward, done, info = self.env.step(action)
            self._obs_buffer.append(obs)
            total_reward += reward
            if done:
                break
        max_frame = np.max(np.stack(self._obs_buffer), axis=0)
        return max_frame, total_reward, done, info
```

```
    def reset(self):
        self._obs_buffer.clear()
        obs = self.env.reset()
        self._obs_buffer.append(obs)
        return obs
```

To opakowanie powtarza te same akcje przez *K* klatek, a także wyznacza wartości maksymalne dla pikseli pochodzących z dwóch kolejnych klatek.

```
class ProcessFrame84(gym.ObservationWrapper):
    def __init__(self, env=None):
        super(ProcessFrame84, self).__init__(env)
        self.observation_space = gym.spaces.Box(
            low=0, high=255, shape=(84, 84, 1), dtype=np.uint8)

    def observation(self, obs):
        return ProcessFrame84.process(obs)

    @staticmethod
    def process(frame):
        if frame.size == 210 * 160 * 3:
            img = np.reshape(frame, [210, 160, 3]).astype(
                np.float32)
        elif frame.size == 250 * 160 * 3:
            img = np.reshape(frame, [250, 160, 3]).astype(
                np.float32)
        else:
            assert False, "Błędna rozdzielczość."
        img = img[:, :, 0] * 0.299 + img[:, :, 1] * 0.587 + \
              img[:, :, 2] * 0.114
        resized_screen = cv2.resize(
            img, (84, 110), interpolation=cv2.INTER_AREA)
        x_t = resized_screen[18:102, :]
        x_t = np.reshape(x_t, [84, 84, 1])
        return x_t.astype(np.uint8)
```

Celem tego opakowania jest konwersja obserwacji wejściowych z ekranu emulatora, który zwykle ma rozdzielczość 210×160 pikseli i zawiera kanały kolorów RGB. Docelowy obraz ma odcienie szarości i rozdzielczość 84×84 pikseli. Wykonywana jest konwersja kolorymetryczna na skalę szarości (metoda ta jest bliższa sposobowi postrzegania kolorów przez człowieka niż zwykłe uśrednianie kanałów kolorów), a także zostają zmienione rozmiary obrazu oraz przycięta jego górna i dolna część.

```
class BufferWrapper(gym.ObservationWrapper):
    def __init__(self, env, n_steps, dtype=np.float32):
        super(BufferWrapper, self).__init__(env)
        self.dtype = dtype
        old_space = env.observation_space
        self.observation_space = gym.spaces.Box(
            old_space.low.repeat(n_steps, axis=0),
            old_space.high.repeat(n_steps, axis=0), dtype=dtype)
```

```
    def reset(self):
        self.buffer = np.zeros_like(
            self.observation_space.low, dtype=self.dtype)
        return self.observation(self.env.reset())

    def observation(self, observation):
        self.buffer[:-1] = self.buffer[1:]
        self.buffer[-1] = observation
        return self.buffer
```

Powyższy kod tworzy stos złożony z kolejnych klatek i zwraca go jako obserwację. Dzięki temu sieć może otrzymać informacje o dynamice obiektów, takie jak prędkość i kierunek piłki w grze Pong lub sposób poruszania się wrogów. To bardzo ważne informacje, których nie da się uzyskać z pojedynczego zrzutu ekranu.

```
class ImageToPyTorch(gym.ObservationWrapper):
    def __init__(self, env):
        super(ImageToPyTorch, self).__init__(env)
        old_shape = self.observation_space.shape
        new_shape = (old_shape[-1], old_shape[0], old_shape[1])
        self.observation_space = gym.spaces.Box(
            low=0.0, high=1.0, shape=new_shape, dtype=np.float32)

    def observation(self, observation):
        return np.moveaxis(observation, 2, 0)
```

To proste opakowanie zmienia format kształtu obserwacji z HWC (wysokość, szerokość, kanał) na CHW (kanał, wysokość, szerokość) wymagany przez bibliotekę PyTorch. W tensorze wejściowym kanał koloru występuje jako ostatni wymiar, ale warstwy konwolucyjne biblioteki PyTorch wymagają, by był pierwszy.

```
class ScaledFloatFrame(gym.ObservationWrapper):
    def observation(self, obs):
        return np.array(obs).astype(np.float32) / 255.0
```

Ostatnie opakowanie, jakie występuje w bibliotece, zamienia dane obserwacyjne (bajty) na liczby zmiennoprzecinkowe, a następnie skaluje wartość każdego piksela do zakresu `[0.0…1.0]`.

```
def make_env(env_name):
    env = gym.make(env_name)
    env = MaxAndSkipEnv(env
    env = FireResetEnv(env)
    env = ProcessFrame84(env)
    env = ImageToPyTorch(env)
    env = BufferWrapper(env, 4)
    return ScaledFloatFrame(env)
```

Na końcu pliku znajduje się prosta funkcja, która tworzy środowisko na podstawie przekazanej nazwy i stosuje w nim wszystkie niezbędne opakowania. A teraz przyjrzyjmy się modelowi.

Model głębokiej sieci Q

Model opublikowany w czasopiśmie „Nature" ma trzy warstwy konwolucyjne, po których występują dwie warstwy w pełni połączone. Pomiędzy warstwami umieszczono nieliniowe funkcje rektyfikatora ReLU. Danymi wyjściowymi modelu są wartości Q odpowiadające akcjom dostępnym w środowisku. Wartości nie są przetwarzane przez funkcję nieliniową, ponieważ mogą być dowolne. Metoda polegająca na obliczeniu wszystkich wartości Q za pomocą jednorazowej operacji umożliwia znaczne zwiększenie wydajności w porównaniu z traktowaniem funkcji $Q(s, a)$ dosłownie i przekazywaniem obserwacji oraz akcji do sieci.

Kod modelu znajduje się w pliku *r06/lib/dqn_model.py*:

```
import torch
import torch.nn as nn
import numpy as np

class DQN(nn.Module):
    def __init__(self, input_shape, n_actions):
        super(DQN, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU()
        )

        conv_out_size = self._get_conv_out(input_shape)
        self.fc = nn.Sequential(
            nn.Linear(conv_out_size, 512),
            nn.ReLU(),
            nn.Linear(512, n_actions)
        )
```

Ponieważ chciałem zaimplementować sieć w sposób generyczny, podzieliłem ją na dwie części: *konwolucyjną* i *sekwencyjną*. Niestety, biblioteka PyTorch nie udostępnia warstwy „spłaszczającej", która mogłaby przekształcić tensor trójwymiarowy w jednowymiarowy wektor liczb, a następnie przesłać go do warstwy w pełni połączonej. Ten problem został rozwiązany za pomocą funkcji `forward()`, dzięki której można przekształcić paczkę tensorów trójwymiarowych w paczkę wektorów jednowymiarowych.

Pewnym problemem jest także to, że nie znamy dokładnej liczby wartości, które pojawiają się na wyjściu warstwy konwolucyjnej i są generowane na podstawie danych wejściowych o określonym kształcie. Taką liczbę musimy przekazać do konstruktora pierwszej, w pełni połączonej warstwy. Jednym z możliwych rozwiązań byłoby zaszycie tej liczby w programie, ponieważ jest ona funkcją kształtu wejściowego (w przypadku danych wejściowych 84×84 wynik warstwy konwolucyjnej będzie się składał z 3136 wartości). Nie jest to jednak najlepszy sposób, ponieważ w takim przypadku kod będzie mniej odporny na zmiany kształtu danych

wejściowych. Lepszym rozwiązaniem jest użycie prostej funkcji `_get_conv_out()`, która akceptuje kształt wejściowy w postaci parametru, a następnie używa warstwy konwolucyjnej z fikcyjnym tensorem o takim właśnie kształcie. Wynik funkcji jest równy liczbie zwróconych parametrów. Taka metoda jest szybka, ponieważ funkcja może zostać wywołana jedynie raz przy tworzeniu modelu. Jej zaletą jest także możliwość wykorzystania kodu generycznego.

```
def _get_conv_out(self, shape):
    o = self.conv(torch.zeros(1, *shape))
    return int(np.prod(o.size()))

def forward(self, x):
    conv_out = self.conv(x).view(x.size()[0], -1)
    return self.fc(conv_out)
```

Ostatnim elementem modelu jest funkcja `forward()`, która akceptuje tensor o czterech wymiarach (pierwszy wymiar to rozmiar paczki, drugi jest kanałem koloru, czyli stosem kolejnych klatek, natomiast trzeci i czwarty to rozmiary obrazu).

Zastosowanie transformacji odbywa się w dwóch krokach: najpierw stosujemy warstwę konwolucyjną z danymi wejściowymi, dzięki czemu na wyjściu otrzymujemy czterowymiarowy tensor. Wynik zostaje spłaszczony do dwóch wymiarów: rozmiaru paczki oraz wektora liczbowego zawierającego parametry zwracane przez konwolucję. Ta operacja jest realizowana w funkcji `view()`, w której jeden z wymiarów będących argumentami może mieć wartość `-1`, a przez to służyć jako **symbol wieloznaczny**. Jeśli przykładowo mamy trójwymiarowy tensor `T` o kształcie `(2, 3, 4)`, składający się z 24 elementów, możemy za pomocą funkcji `T.view(6, 4)` przekształcić go w dwuwymiarowy tensor o sześciu wierszach i czterech kolumnach. Ta operacja nie utworzy nowego obiektu w pamięci ani nie przeniesie danych w inne miejsce — po prostu zmieni wysokopoziomowy kształt tensora. Ten sam wynik można uzyskać za pomocą wywołania `T.view(-1, 4)` lub `T.view(6, -1)`. Takie rozwiązanie jest bardzo wygodne, gdy pierwszym wymiarem tensora jest rozmiar paczki. Wreszcie przekazujemy spłaszczony tensor dwuwymiarowy do warstw w pełni połączonych, aby uzyskać wartości Q dla każdej danej wejściowej.

Trenowanie

Trzeci moduł zawiera definicje bufora doświadczeń, agenta, funkcji straty, a także pętlę treningową. Zanim przejdę do kodu, chciałbym wspomnieć o hiperparametrach wykorzystywanych podczas trenowania. Artykuł w czasopiśmie „Nature" zawierał tabelę hiperparametrów użytych we wszystkich 49 grach Atari do trenowania modelu. W każdej z nich wykorzystano te same parametry (przy czym model był zawsze trenowany od podstaw), ponieważ intencją badaczy było udowodnienie, że metoda wykorzystująca jeden model jest wystarczająco solidna, aby można ją było stosować w wielu grach o różnej złożoności, przestrzeni akcji czy strukturze nagród. Jednak nasz cel jest znacznie skromniejszy — chcemy po prostu skutecznie grać jedynie w Ponga.

Gra Pong jest dość prosta w porównaniu z innymi, które należą do zestawu testowego Atari, więc w tym przypadku wartości hiperparametrów zaprezentowane w artykule są trochę przesadzone. Aby uzyskać najlepsze wyniki we wszystkich 49 grach, użyto na przykład bufora pozwalającego przechowywać milion obserwacji, który wymagał około 20 GB pamięci RAM.

Zastosowany sposób wygaszania wartości epsilon również nie jest najlepszy w przypadku gry Pong. W oryginalnym rozwiązaniu podczas trenowania przy użyciu pierwszego miliona klatek wartość epsilon liniowo się zmniejszała od 1,0 do 0,1. Jednak przeprowadzone przeze mnie eksperymenty wykazały, że w przypadku gry Pong należy zmniejszać wartość epsilon jedynie przez pierwsze 150 tysięcy klatek. Bufor również może być znacznie mniejszy — wystarczy przechowywać w nim 10 tysięcy przejść.

W poniższym przykładzie użyłem własnych parametrów. Różnią się one od parametrów zaprezentowanych w artykule, ale dzięki temu pozwalają znaleźć rozwiązanie dla gry Pong w czasie około 10 razy krótszym. Zmodyfikowany przeze mnie kod, wykorzystujący kartę graficzną GeForce GTX 1080 Ti, pozwolił na osiągnięcie średniego wyniku 19,0 w ciągu od jednej do dwóch godzin. Jeśli zostałyby zastosowane oryginalne hiperparametry, na trenowanie należałoby poświęcić co najmniej jeden dzień.

To przyspieszenie wynika z dopasowania modelu do określonego środowiska i może spowodować pojawienie się problemów w innych grach. Model ze zmienionymi opcjami możesz oczywiście przetestować w innych grach z zestawu Atari.

```
from lib import wrappers
from lib import dqn_model

import argparse
import time
import numpy as np
import collections

import torch
import torch.nn as nn
import torch.optim as optim

from tensorboardX import SummaryWriter
```

Najpierw importujemy niezbędne moduły i definiujemy hiperparametry.

```
DEFAULT_ENV_NAME = "PongNoFrameskip-v4"
MEAN_REWARD_BOUND = 19.0
```

Dwie stałe określają domyślną nazwę środowiska oraz wartość progową nagrody dla ostatnich 100 epizodów, której przekroczenie pozwoli zakończyć proces trenowania. Jeśli chcesz, możesz zmienić nazwę środowiska za pomocą opcji wiersza poleceń.

```
GAMMA = 0.99
BATCH_SIZE = 32
REPLAY_SIZE = 10000
REPLAY_START_SIZE = 10000
LEARNING_RATE = 1e-4
SYNC_TARGET_FRAMES = 1000
```

Powyższe parametry definiują:

- Wartość gamma stosowaną w przybliżeniu Bellmana (`GAMMA`).
- Rozmiar paczki danych pobieranych z bufora (`BATCH_SIZE`).

- Maksymalną pojemność bufora (REPLAY_SIZE).
- Liczbę ramek, które gromadzimy przed rozpoczęciem trenowania, aby zapełnić bufor (REPLAY_START_SIZE).
- Współczynnik uczenia używany w optymalizatorze Adam (LEARNING_RATE).
- Częstość synchronizowania wag modelu treningowego z modelem docelowym, który jest używany w przybliżeniu Bellmana do uzyskania wartości następnego stanu (SYNC_TARGET_FRAMES).

```
EPSILON_DECAY_LAST_FRAME = 150000
EPSILON_START = 1.0
EPSILON_FINAL = 0.01
```

Ostatni zestaw hiperparametrów definiuje sposób zanikania wartości epsilon. W celu osiągnięcia właściwego poziomu eksploracji na początku trenowania wartość epsilon wynosi 1,0, co powoduje, że wszystkie akcje są wybierane losowo. Następnie, podczas pierwszych 150 tysięcy klatek, wartość epsilon zmniejsza się liniowo do 0,01, co odpowiada losowej akcji wykonywanej w 1% kroków. Podobny schemat został przedstawiony w oryginalnym artykule, jednak czas zanikania był prawie 10 razy dłuższy (więc wartość epsilon równa 0,01 została osiągnięta po milionie klatek).

W kolejnym fragmencie kodu zostaje zdefiniowany bufor, który przechowuje przejścia uzyskane ze środowiska (krotki zawierające obserwację, akcję, nagrodę, flagę done i kolejny stan). Gdy w środowisku wykonujemy nowy krok, w buforze zostaje zapisane przejście. Liczba kroków jest stała (w tym przypadku równa 10 tysięcy). Na potrzeby trenowania losowo pobieramy paczkę przejść z bufora, dzięki czemu nie uwzględniamy związku między kolejnymi krokami w środowisku.

```
Experience = collections.namedtuple(
    'Experience', field_names=['state', 'action', 'reward',
                               'done', 'new_state'])

class ExperienceBuffer:
    def __init__(self, capacity):
        self.buffer = collections.deque(maxlen=capacity)

    def __len__(self):
        return len(self.buffer)

    def append(self, experience):
        self.buffer.append(experience)

    def sample(self, batch_size):
        indices = np.random.choice(len(self.buffer), batch_size,
                                   replace=False)
        states, actions, rewards, dones, next_states = \
            zip(*[self.buffer[idx] for idx in indices])
        return np.array(states), np.array(actions), \
               np.array(rewards, dtype=np.float32), \
               np.array(dones, dtype=np.uint8), \
               np.array(next_states)
```

Kod definiujący bufor jest dość prosty — wykorzystuje zdolność klasy `deque` do przechowywania określonej liczby wpisów. W metodzie `sample()` tworzymy listę losowych indeksów, a następnie zamieniamy pobierane dane na tablice NumPy w celu wygodniejszego obliczania funkcji straty.

Niezbędna jest również deklaracja klasy `Agent`, która współdziała ze środowiskiem i zapisuje wyniki w buforze doświadczeń:

```
class Agent:
    def __init__(self, env, exp_buffer):
        self.env = env
        self.exp_buffer = exp_buffer
        self._reset()

    def _reset(self):
        self.state = env.reset()
        self.total_reward = 0.0
```

Podczas inicjalizowania agenta musimy zapamiętać referencje do środowiska i bufora, dzięki czemu będzie mógł on mieć dostęp do bieżących obserwacji i nagrody sumarycznej:

```
    @torch.no_grad()
    def play_step(self, net, epsilon=0.0, device="cpu"):
        done_reward = None

        if np.random.random() < epsilon:
            action = env.action_space.sample()
        else:
            state_a = np.array([self.state], copy=False)
            state_v = torch.tensor(state_a).to(device)
            q_vals_v = net(state_v)
            _, act_v = torch.max(q_vals_v, dim=1)
            action = int(act_v.item())
```

Główna metoda klasy `Agent` wykonuje krok w środowisku i zapisuje uzyskany wynik w buforze. Najpierw należy jednak wybrać akcję. Z prawdopodobieństwem epsilon (przekazanym jako argument) wykonujemy akcję losową. W przeciwnym razie używamy poprzedniego modelu, aby uzyskać wartości Q dla wszystkich możliwych akcji i wybrać tę, która jest najlepsza.

```
        new_state, reward, is_done, _ = self.env.step(action)
        self.total_reward += reward

        exp = Experience(self.state, action, reward,
                         is_done, new_state)
        self.exp_buffer.append(exp)
        self.state = new_state
        if is_done:
            done_reward = self.total_reward
            self._reset()
        return done_reward
```

Po wybraniu akcji przekazujemy ją do środowiska, uzyskujemy kolejną obserwację i nagrodę, zapisujemy dane w buforze, a wreszcie sprawdzamy warunek zakończenia epizodu. Jeśli epizod się zakończył, w wyniku funkcji otrzymujemy nagrodę sumaryczną. W przeciwnym razie zwracana jest wartość `None`.

Teraz nadszedł czas na przedstawienie ostatniej funkcji w module treningowym, która oblicza wartość straty dla wybranej paczki. Ponieważ wykorzystuje ona współbieżność procesów karty graficznej, przetwarza wszystkie paczki za pomocą operacji wektorowych, co sprawia, że trudniej jest ją zrozumieć niż zwykłą pętlę. Jednak ta optymalizacja się opłaca — wersja równoległa jest ponad dwa razy szybsza niż przetwarzanie paczek w pętli.

Oto funkcja straty, jaką musimy obliczyć:

- w przypadku kroków, które nie występują pod koniec epizodu:
  $$\mathcal{L} = \left(Q(s,a) - (r + \gamma \max_{a' \in A} \hat{Q}(s',a'))\right)^2;$$
- w przypadku kroków końcowych: $\mathcal{L} = (Q(s,a) - r)^2$.

```
def calc_loss(batch, net, tgt_net, device="cpu"):
    states, actions, rewards, dones, next_states = batch
```

Do metody przekazujemy paczkę jako krotkę tablic (umieszczonych w buforze za pomocą metody `sample()`), sieć, którą trenujemy, a także sieć docelową, która będzie okresowo synchronizowana z treningową.

Pierwszy model (przekazany jako argument `net`) jest używany do wyznaczania gradientów. Drugi, przekazany w argumencie `tgt_net`, jest stosowany do obliczania wartości dla kolejnych stanów. Ponieważ obliczenia te nie powinny wpływać na gradienty, używamy tensorowej funkcji `detach()` z biblioteki PyTorch, aby zapobiec ich propagacji w sieci docelowej. Funkcja ta została opisana w rozdziale 3.

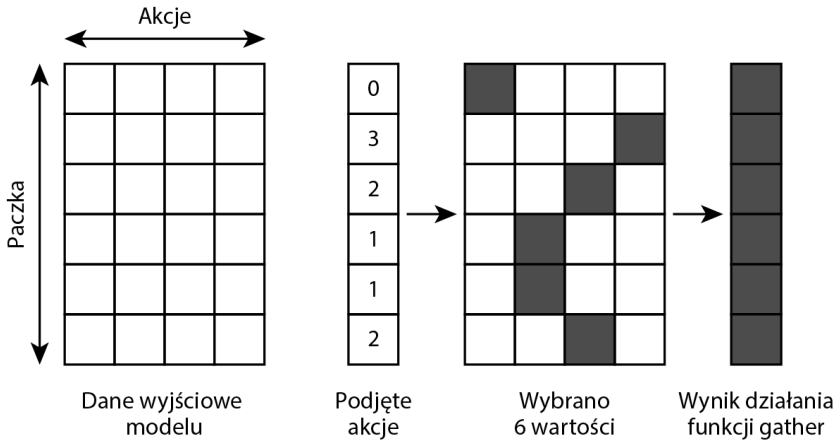
```
    states_v = torch.tensor(np.array(
        states, copy=False)).to(device)
    next_states_v = torch.tensor(np.array(
        next_states, copy=False)).to(device)
    actions_v = torch.tensor(actions).to(device)
    rewards_v = torch.tensor(rewards).to(device)
    done_mask = torch.BoolTensor(dones).to(device)
```

Powyższy kod jest prosty — tablice NumPy razem z paczkami zamieniamy na tensory biblioteki PyTorch, a potem kopiujemy je do karty graficznej, jeśli w argumencie określono, że należy wykorzystać architekturę CUDA.

```
    state_action_values = net(states_v).gather(
        1, actions_v.unsqueeze(-1)).squeeze(-1)
```

Przekazujemy obserwacje do pierwszego modelu, a przy użyciu operacji tensorowej `gather()` wyodrębniamy określone wartości Q dla podjętych akcji. Pierwszy argument funkcji `gather()` jest indeksem wymiaru, dla którego chcemy wykonać operację (w tym przypadku jest on równy 1, co odpowiada akcjom).

Drugi argument to tensor z indeksami wybranych elementów. Dodatkowe wywołania `unsqueeze()` i `squeeze()` są wymagane do wyznaczenia poprawnego argumentu indeksu dla funkcji `gather` i pozbycia się dodatkowych wymiarów (indeks powinien mieć taką samą liczbę wymiarów jak dane, które przetwarzamy). Na rysunku 6.3 przedstawiłem operację, którą wykonuje funkcja `gather()` w przypadku sześciu elementów i czterech akcji:



Rysunek 6.3. Transformacja tensorów podczas wyznaczania funkcji straty w głębokiej sieci Q

Pamiętaj, że wynik funkcji `gather()` użytej z tensorami jest operacją różniczkowalną, która spowoduje, że wszystkie gradienty pozostaną związane z końcową wartością straty.

```
next_state_values = tgt_net(next_states_v).max(1)[0]
```

W powyższym wierszu wykorzystujemy sieć docelową z obserwacjami następnego stanu i obliczamy maksymalną wartość Q dla 1. wymiaru, czyli *akcji*. Funkcja `max()` zwraca zarówno wartości maksymalne, jak i ich indeksy (więc wyznacza funkcję max, jak i argmax), co jest bardzo wygodne. Jednak w tym przypadku interesują nas tylko wartości, więc używamy pierwszego elementu wyniku.

```
next_state_values[done_mask] = 0.0
```

Wykonujemy proste, ale bardzo ważne działanie — jeśli przejście w paczce pochodzi z ostatniego kroku w epizodzie, wartość akcji nie uwzględnia zdyskontowanej nagrody za następny stan, ponieważ po prostu go już nie ma. Taka korekta może się wydawać drobna, ale w gruncie rzeczy jest bardzo znacząca — bez niej trenowanie nie wykaże zbieżnej tendencji.

```
next_state_values = next_state_values.detach()
```

Oddzielamy wartość od wykresu obliczeniowego, aby zapobiec przepływowi gradientów do sieci neuronowej używanej w celu wyznaczenia przybliżenia Q dla następnych stanów.

Jest to ważna operacja, ponieważ bez niej propagacja wsteczna straty zacznie wpływać zarówno na prognozy dotyczące stanu bieżącego, jak i następnego. Nie chcemy zmieniać prognoz dla stanu następnego, ponieważ są one używane w równaniu Bellmana do obliczania referencyjnych

wartości Q. Aby nie dopuścić, by gradienty przepływały do danej gałęzi wykresu, używamy metody `detach()`, która zwraca tensor, nie uwzględniwszy historii jego obliczeń.

```
        expected_state_action_values = next_state_values * GAMMA + \
                                       rewards_v
        return nn.MSELoss()(state_action_values,
                            expected_state_action_values)
```

Wreszcie obliczamy wartość przybliżenia Bellmana i stratę w postaci średniego błędu kwadratowego. Reszta kodu to pętla treningowa.

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--cuda", default=False,
                        action="store_true", help="Użyj technologii CUDA")
    parser.add_argument("--env", default=DEFAULT_ENV_NAME,
                        help="Nazwa środowiska. Wartość domyślna=" +
                             DEFAULT_ENV_NAME)
    args = parser.parse_args()
    device = torch.device("cuda" if args.cuda else "cpu")
```

Na początek tworzymy parser argumentów wiersza poleceń. Możemy zastosować technologię CUDA, a także trenować w środowiskach, które różnią się od domyślnego.

```
    env = wrappers.make_env(args.env)
    net = dqn_model.DQN(env.observation_space.shape,
                        env.action_space.n).to(device)
    tgt_net = dqn_model.DQN(env.observation_space.shape,
                            env.action_space.n).to(device)
```

Tworzymy środowisko ze wszystkimi wymaganymi opakowaniami, sieć neuronową, którą zamierzamy wytrenować, a także sieć docelową o tej samej architekturze. Na początku zostaną one zainicjalizowane losowymi wagami, ale nie ma to większego znaczenia, ponieważ i tak będziemy je synchronizować co 1000 klatek, co w przybliżeniu odpowiada jednemu epizodowi gry w Ponga.

```
    writer = SummaryWriter(comment="-" + args.env)
    print(net)

    buffer = ExperienceBuffer(REPLAY_SIZE)
    agent = Agent(env, buffer)
    epsilon = EPSILON_START
```

Następnie tworzymy bufor o wymaganym rozmiarze i przekazujemy go agentowi. Parametr epsilon został na początku zainicjalizowany wartością `1.0`, a po każdej iteracji będzie zmniejszany.

```
    optimizer = optim.Adam(net.parameters(), lr=LEARNING_RATE)
    total_rewards = []
    frame_idx = 0
    ts_frame = 0
    ts = time.time()
    best_m_reward = None
```

Zanim rozpoczniemy pętlę treningową, definiujemy jeszcze optymalizator, bufor nagród za pełne epizody, licznik ramek i kilka zmiennych do monitorowania prędkości działania programu i wartości najlepszej nagrody średniej. Za każdym razem, gdy wartość średniej nagrody pobije rekord, zapiszemy model w pliku.

```
while True:
    frame_idx += 1
    epsilon = max(EPSILON_FINAL, EPSILON_START -
                  frame_idx / EPSILON_DECAY_LAST_FRAME)
```

Na początku pętli treningowej zwiększamy liczbę wykonanych iteracji i zmniejszamy wartość epsilon zgodnie z przyjętą metodą. Będzie się ona liniowo zmniejszać przez określoną liczbę ramek (`EPSILON_DECAY_LAST_FRAME=150k`), a następnie zostanie utrzymana na tym samym poziomie `EPSILON_FINAL=0.01`.

```
reward = agent.play_step(net, epsilon, device=device)
if reward is not None:
    total_rewards.append(reward)
    speed = (frame_idx - ts_frame) / (time.time() - ts)
    ts_frame = frame_idx
    ts = time.time()
    m_reward = np.mean(total_rewards[-100:])
    print("%d: gry - %d, nagroda %.3f, "
          "eps %.2f, %.2f fps" % (
        frame_idx, len(total_rewards), m_reward, epsilon,
        speed
    ))
    writer.add_scalar("epsilon", epsilon, frame_idx)
    writer.add_scalar("speed", speed, frame_idx)
    writer.add_scalar("reward_100", m_reward, frame_idx)
    writer.add_scalar("reward", reward, frame_idx)
```

W powyższym fragmencie kodu agent wykonuje pojedynczy krok w środowisku (używając bieżącej sieci i wartości epsilon). Funkcja `play_step` zwraca wynik różny od `None` tylko wtedy, gdy dany krok jest ostatnim krokiem w epizodzie. W takim przypadku informujemy o postępach. W szczególności obliczamy i wyświetlamy (zarówno w konsoli, jak i narzędziu TensorBoard) następujące wartości:

- szybkość jako liczbę klatek przetwarzanych na sekundę,
- liczbę wykonanych epizodów,
- średnią nagrodę za ostatnie 100 epizodów,
- bieżącą wartość epsilon.

```
if best_m_reward is None or best_m_reward < m_reward:
    torch.save(net.state_dict(), args.env +
               "-best_%.0f.dat" % m_reward)
    if best_m_reward is not None:
        print("Nagroda uległa zmianie: %.3f -> %.3f" % (
            best_m_reward, m_reward))
    best_m_reward = m_reward
if m_reward > MEAN_REWARD_BOUND:
    print("Rozwiązano po %d klatkach!" % frame_idx)
    break
```

Za każdym razem, gdy średnia nagroda za ostatnie 100 epizodów osiąga maksimum, wyświetlamy komunikat i zapisujemy parametry modelu. Jeśli średnia nagroda przekroczy określoną wartość graniczną, przestajemy trenować. W przypadku Ponga wartość ta wynosi 19,0, co oznacza wygraną w więcej niż 19 z 21 możliwych gier.

```
if len(buffer) < REPLAY_START_SIZE:
    continue

if frame_idx % SYNC_TARGET_FRAMES == 0:
    tgt_net.load_state_dict(net.state_dict())
```

Sprawdzamy, czy bufor jest wystarczająco duży dla procesu trenowania. Na początku powinniśmy poczekać, aż zostanie zebrana wystarczająca ilość danych — w tym przypadku oznacza to 10 tysięcy przejść. W następnej instrukcji synchronizujemy parametry sieci treningowej z siecią docelową co `SYNC_TARGET_FRAMES`, czyli 1000 klatek.

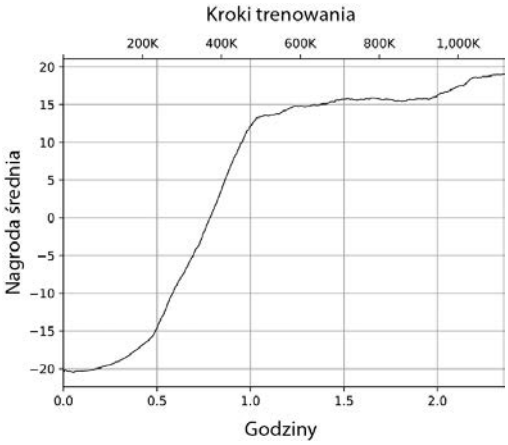
```
optimizer.zero_grad()
batch = buffer.sample(BATCH_SIZE)
loss_t = calc_loss(batch, net, tgt_net, device=device)
loss_t.backward()
optimizer.step()
```

Ostatni fragment pętli treningowej jest bardzo prosty, ale jego wykonanie zajmuje najwięcej czasu — zerujemy gradienty, pobieramy paczkę danych z bufora, obliczamy stratę i wykonujemy optymalizację, aby ją zminimalizować.

Uruchomienie programu i sprawdzenie jego wydajności

Ten program jest wymagający, jeśli chodzi o zasoby. Należy przetworzyć około 400 tysięcy klatek, aby osiągnąć średnią nagrodę równą 17 (co oznacza wygranie ponad 80% gier). Podobna liczba klatek będzie potrzebna, aby uzyskać nagrodę o wartości 19. Wynika to stąd, że postępy w nauce będą coraz mniejsze i modelowi będzie trudno poprawić wynik. Aby w pełni wytrenować model, wymagany jest mniej więcej milion ramek. W przypadku karty graficznej GTX 1080 Ti osiągnąłem prędkość około 120 klatek na sekundę, co oznacza w przybliżeniu dwie godziny trenowania. Jeśli użyjesz zwykłego procesora, uzyskasz znacznie mniejszą prędkość — około dziewięć klatek na sekundę, czyli trenowanie trwające półtora dnia. Pamiętaj, że dotyczy to gry Pong, która jest stosunkowo łatwa do wygrania. Inne gry wymagają setek milionów klatek i 100-krotnie większego bufora.

W rozdziale 8. przeanalizujemy różne metody opracowane przez naukowców od 2015 roku, które będą mogły pomóc zarówno zwiększyć szybkość trenowania, jak i przetwarzania danych. Rozdział 9. będzie poświęcony technicznym sztuczkom umożliwiającym poprawę wydajności metod uczenia przez wzmacnianie. Bez względu na to, w przypadku gier Atari zawsze będziesz potrzebować odpowiednich zasobów i cierpliwości. Na rysunku 6.4 zaprezentowałem wykresy dynamiki trenowania.



Rysunek 6.4. Zmiany nagrody średniej przez ostatnich 100 epizodów

Oto wyniki uzyskane na początku trenowania:

```
r06$ ./02_dqn_pong.py --cuda
  (conv): Sequential(
    (0): Conv2d(4, 32, kernel_size=(8, 8), stride=(4, 4))
    (1): ReLU()
    (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
    (3): ReLU()
    (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
    (5): ReLU()
  )
  (fc): Sequential(
    (0): Linear(in_features=3136, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=6, bias=True)
  )
)
971: gry - 1, nagroda -21.000, eps 0.99, 890.81 fps
1733: gry - 2, nagroda -21.000, eps 0.98, 984.59 fps
2649: gry - 3, nagroda -20.667, eps 0.97, 987.53 fps
Nagroda uległa zmianie: -21.000 -> -20.667
3662: gry - 4, nagroda -20.500, eps 0.96, 921.47 fps
Nagroda uległa zmianie: -20.667 -> -20.500
4619: gry - 5, nagroda -20.600, eps 0.95, 965.86 fps
5696: gry - 6, nagroda -20.500, eps 0.94, 963.99 fps
6671: gry - 7, nagroda -20.429, eps 0.93, 967.28 fps
Nagroda uległa zmianie: -20.500 -> -20.429
7648: gry - 8, nagroda -20.375, eps 0.92, 948.15 fps
Nagroda uległa zmianie: -20.429 -> -20.375
8528: gry - 9, nagroda -20.444, eps 0.91, 954.72 fps
9485: gry - 10, nagroda -20.400, eps 0.91, 936.82 fps
10394: gry - 11, nagroda -20.455, eps 0.90, 256.84 fps
11292: gry - 12, nagroda -20.417, eps 0.89, 127.90 fps
12132: gry - 13, nagroda -20.385, eps 0.88, 130.18 fps
```

Podczas pierwszych 10 tysięcy kroków prędkość jest bardzo duża, ponieważ nie wykonujemy trenowania, które jest najbardziej wymagającą operacją w kodzie. Następnie zaczynamy pobierać paczki treningowe i wydajność znacznie spada.

Po zakończeniu setek epizodów głęboka sieć Q powinna już zacząć wygrywać jedną lub dwie gry na 21 możliwych. Ponieważ wartość epsilon zmalała, spadła też prędkość — model wykorzystujemy nie tylko do trenowania, ale także do wykonywania kroków w środowisku:

```
94101: gry - 86, nagroda -19.512, eps 0.06, 120.67 fps
Nagroda uległa zmianie: -19.541 -> -19.512
96279: gry - 87, nagroda -19.460, eps 0.04, 120.21 fps
Nagroda uległa zmianie: -19.512 -> -19.460
98140: gry - 88, nagroda -19.455, eps 0.02, 119.10 fps
Nagroda uległa zmianie: -19.460 -> -19.455
99884: gry - 89, nagroda -19.416, eps 0.02, 123.34 fps
Nagroda uległa zmianie: -19.455 -> -19.416
101451: gry - 90, nagroda -19.411, eps 0.02, 120.53 fps
Nagroda uległa zmianie: -19.416 -> -19.411
103812: gry - 91, nagroda -19.330, eps 0.02, 122.41 fps
Nagroda uległa zmianie: -19.411 -> -19.330
105908: gry - 92, nagroda -19.283, eps 0.02, 119.85 fps
Nagroda uległa zmianie: -19.330 -> -19.283
108259: gry - 93, nagroda -19.172, eps 0.02, 122.09 fps
Nagroda uległa zmianie: -19.283 -> -19.172
```

Wreszcie po wielu kolejnych grach głęboka sieć Q ostatecznie zaczyna pokonywać (niezbyt wyrafinowanego) przeciwnika z wbudowaną sztuczną inteligencją:

```
1097050: gry - 522, nagroda 18.800, eps 0.01, 132.71 fps
Nagroda uległa zmianie: 18.770 -> 18.800
1098741: gry - 523, nagroda 18.820, eps 0.01, 134.58 fps
Nagroda uległa zmianie: 18.800 -> 18.820
1100507: gry - 524, nagroda 18.890, eps 0.01, 132.11 fps
Nagroda uległa zmianie: 18.820 -> 18.890
1102198: gry - 525, nagroda 18.920, eps 0.01, 133.68 fps
Nagroda uległa zmianie: 18.890 -> 18.920
1103947: gry - 526, nagroda 18.920, eps 0.01, 130.07 fps
1105745: gry - 527, nagroda 18.920, eps 0.01, 130.27 fps
1107423: gry - 528, nagroda 18.960, eps 0.01, 130.08 fps
Nagroda uległa zmianie: 18.920 -> 18.960
1109286: gry - 529, nagroda 18.940, eps 0.01, 129.04 fps
1111058: gry - 530, nagroda 18.940, eps 0.01, 128.59 fps
1112836: gry - 531, nagroda 18.930, eps 0.01, 130.84 fps
1114622: gry - 532, nagroda 18.980, eps 0.01, 130.34 fps
Nagroda uległa zmianie: 18.960 -> 18.980
1116437: gry - 533, nagroda 19.080, eps 0.01, 130.09 fps
Nagroda uległa zmianie: 18.980 -> 19.080
Rozwiązano po 1116437 klatkach!
```

Ze względu na losowość w procesie trenowania rzeczywista dynamika może różnić się od tej, którą zaprezentowałem w książce. W niektórych, rzadkich przypadkach (około 1 na 10) trenowanie w ogóle nie przynosi rezultatów — przez długi czas otrzymuje się nagrody równe –21.

Jeśli w ciągu pierwszych 100 – 200 tysięcy iteracji nie zauważysz poprawy wyników, cały proces należy zrestartować.

Użycie modelu

Przeprowadzenie procesu trenowania to nie wszystko. Ostatecznym celem nie jest samo wytrenowanie modelu — chcielibyśmy również, aby uzyskiwał on dobre wyniki. Gdy podczas trenowania aktualizujemy średnią nagrodę dla ostatnich 100 gier, zapisujemy model do pliku *PongNoFrameskip-v4-best.dat*. W archiwum dostępnym pod adresem *https://ftp.helion.pl/przyklady/glucz2.zip* znajduje się plik *r06/03_dqn_play.py*, który zawiera program umożliwiający wczytanie modelu i wykonanie jednego epizodu, wyświetlając przy okazji zmiany dynamiki.

Kod jest bardzo prosty, ale obserwowanie, jak kilka macierzy z milionami parametrów może grać w Ponga z nadludzką sprawnością, jest niesamowitym doświadczeniem.

```
import gym
import time
import argparse
import numpy as np
import torch

from lib import wrappers
from lib import dqn_model

import collections

DEFAULT_ENV_NAME = "PongNoFrameskip-v4"
FPS = 25
```

Na początku importujemy potrzebne moduły bibliotek PyTorch i Gym. Parametr `FPS` (liczba klatek na sekundę) określa przybliżoną częstotliwość wyświetlanych klatek.

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("-m", "--model", required=True,
                        help="Plik modelu")
    parser.add_argument("-e", "--env", default=DEFAULT_ENV_NAME,
                        help="Nazwa środowiska. Wartość domyślna=" +
                             DEFAULT_ENV_NAME)
    parser.add_argument("-r", "--record", help="Nazwa katalogu do przechowywania
pliku wideo")
    parser.add_argument("--no-vis", default=True, dest='vis',
                        help="Wyłączenie wizualizacji",
                        action='store_false')
    args = parser.parse_args()
```

Do skryptu możesz przekazać nazwę pliku zapisanego modelu, a także nazwę środowiska Gym (oczywiście model i środowisko muszą być zgodne ze sobą). Dodatkowo w opcji `-r` możesz podać nazwę nieistniejącego katalogu, który zostanie wykorzystany do zapisania pliku wideo z rozgrywki (przy użyciu opakowania `Monitor`). Domyślnie skrypt wyświetla tylko klatki, ale taka opcja może się przydać, jeśli chciałbyś na przykład przesłać nagrany filmik na YouTube'a.

```
env = wrappers.make_env(args.env)
if args.record:
    env = gym.wrappers.Monitor(env, args.record)
net = dqn_model.DQN(env.observation_space.shape,
                    env.action_space.n)
state = torch.load(args.model, map_location=lambda stg,_: stg)
net.load_state_dict(state)

state = env.reset()
total_reward = 0.0
c = collections.Counter()
```

Powyższy kod jest całkiem prosty — tworzymy środowisko i model, a następnie wczytujemy wagi z pliku, którego nazwę przekazaliśmy w parametrze wiersza poleceń. Argument `map_location`, przekazany do funkcji `torch.load()`, jest niezbędny do odwzorowania tensora z karty graficznej na standardowy procesor. Domyślnie klasa `torch` próbuje wczytać tensory do tego samego urządzenia, z którego zostały odczytane. Jeśli jednak do komputera bez odpowiedniej karty graficznej skopiujesz model, który otrzymałeś podczas trenowania w maszynie wyposażonej w GPU, lokalizacje muszą zostać ponownie odwzorowane. W tym przykładzie w ogóle nie stosujemy karty graficznej, ponieważ proces wnioskowania jest wystarczająco szybki z wykorzystaniem zwykłego procesora.

```
while True:
    start_ts = time.time()
    if args.vis:
        env.render()
    state_v = torch.tensor(np.array([state], copy=False))
    q_vals = net(state_v).data.numpy()[0]
    action = np.argmax(q_vals)
    c[action] += 1
```

Jest to prawie dokładna kopia funkcji `play_step()` należącej do klasy `Agent` z kodu treningowego. Różnica polega na tym, że nie używamy metody epsilonu zachłannego. Po prostu przekazujemy obserwację agentowi i wybieramy akcję o maksymalnej wartości. Jedyną nowością jest metoda `render()` należąca do środowiska, która jest używana w bibliotece Gym do wyświetlania bieżącej obserwacji (w tym celu musisz wykorzystywać graficzny interfejs użytkownika — GUI).

```
    state, reward, done, _ = env.step(action)
    total_reward += reward
    if done:
        break
    if args.vis:
        delta = 1/FPS - (time.time() - start_ts)
        if delta > 0:
            time.sleep(delta)
print("Nagroda sumaryczna: %.2f" % total_reward)
print("Liczba akcji:", c)
if args.record:
    env.env.close()
```

Reszta kodu jest również prosta. Przekazujemy akcję do środowiska, wyznaczamy nagrodę sumaryczną i zatrzymujemy pętlę, gdy epizod dobiegnie końca. Następnie wyświetlamy wartość nagrody sumarycznej i liczbę akcji wykonanych przez agenta.

Nagrania rozgrywek z różnych etapów trenowania możesz znaleźć na tej liście YouTube'a: *https://www.youtube.com/playlist?list=PLMVwuZENsfJklt4vCltrWq0KV9aEZ3ylu*.

Rzeczy do przetestowania

Jeśli chciałbyś samodzielnie poeksperymentować z kodem z tego rozdziału, możesz uwzględnić poniższą listę ze wskazówkami. Pamiętaj jednak, że wykonanie wszystkich testów może zająć sporo czasu, a oprócz tego nie zawsze wszystko będzie działać zgodnie z oczekiwaniami. Jednak dzięki takim eksperymentom dogłębnie zrozumiesz materiał i będziesz go mógł zastosować w praktyce:

- Wypróbuj inne gry Atari, takie jak Breakout, Atlantis czy River Raid (moja ulubiona gra z dzieciństwa). Może to jednak wymagać dostrojenia hiperparametrów.
- Zamiast FrozenLake użyj innego środowiska wykorzystującego świat siatki. Mam na myśli grę Taxi, w której taksówkarz musi odebrać pasażerów i zawieźć ich do miejsca docelowego.
- Spróbuj zmodyfikować hiperparametry dla gry Pong. Czy dałoby się trenować szybciej? Na stronie OpenAI stwierdzono, że za pomocą asynchronicznej metody aktor-krytyk (którą przeanalizujemy w trzeciej części książki) można rozwiązać problem Ponga w 30 minut. Być może taki czas można byłoby uzyskać również za pomocą głębokiej sieci Q.
- Spróbuj przyspieszyć działanie kodu treningowego głębokiej sieci Q. W projekcie z repozytorium OpenAI Baselines uzyskano prędkość 350 fps przy użyciu biblioteki TensorFlow z kartą graficzną GTX 1080 Ti. Wygląda więc na to, że kod PyTorch można zoptymalizować. Ten temat omówię w rozdziale 8., jednak w międzyczasie możesz również przeprowadzić własne eksperymenty.
- Na podstawie nagrań wideo możesz wywnioskować, że modele z wynikiem zbliżonym do zera grają całkiem dobrze. Faktycznie, również odniosłem wrażenie, że grają one lepiej niż modele ze średnim wynikiem 10 – 19. Być może tak się dzieje z powodu nadmiernego dopasowania do określonych sytuacji w grze. Czy mógłbyś spróbować naprawić ten problem? Może udałoby się zastosować generatywną sieć przeciwstawną, aby zmusić jeden model do współdziałania z innym?
- Postaraj się wygenerować model *Największego Zwycięzcy w Ponga*, który osiąga średni wynik równy 21. Nie powinno to być trudne — użyj metody polegającej na stopniowym zmniejszaniu wartości współczynnika uczenia.

Podsumowanie

W tym rozdziale przeanalizowałem wiele dość skomplikowanych zagadnień. Dowiedziałeś się, jakie trudności napotyka zastosowanie metody iteracji wartości w złożonych środowiskach, które charakteryzują się dużymi przestrzeniami obserwacji. Poznałeś skuteczne rozwiązanie stosowane w takich przypadkach, czyli Q-uczenie. Odpowiedni algorytm przetestowaliśmy w środowisku FrozenLake, a następnie omówiłem przybliżanie wartości Q za pomocą sieci neuronowej oraz problemy, jakie to powoduje.

Poznałeś również kilka sztuczek związanych z głębokimi sieciami Q, które poprawiają stabilność ich trenowania i poziom konwergencji — chodziło o bufor doświadczeń, sieci docelowe i składanie klatek. Zastosowaliśmy je w implementacji głębokiej sieci Q, która mogła skutecznie grać w Ponga z pakietu gier Atari.

W następnym rozdziale zapoznasz się z całym zestawem sztuczek, które naukowcy wymyślili od 2015 roku w celu poprawy konwergencji i jakości głębokich sieci Q. Dzięki ich zastosowaniu można uzyskać wyjątkowe wyniki w większości z 54 (dodano nowe) gier Atari. Ten zestaw został opublikowany w 2017 roku. Przeanalizujemy i zaimplementujemy wszystkie zaprezentowane sztuczki.

Witaj, świecie prawdziwej sztucznej inteligencji!

Głębokie uczenie przez wzmacnianie rozwija się bardzo dynamicznie. Dziedzinę tę charakteryzuje niewyczerpany potencjał rozwiązywania trudnych problemów. Zajmuje się tym co najmniej kilka grup badawczych, koncentrujących się na wdrażaniu głębokiego uczenia przez wzmacnianie w różnych branżach. Niestety, opisy najnowszych osiągnięć są trudne do zrozumienia i zbyt abstrakcyjne, aby można było je łatwo zastosować w praktycznych implementacjach, a przecież poprawne działanie aplikacji jest uwarunkowane gruntownym zrozumieniem problemu przez projektanta.

To zaktualizowane i rozszerzone wydanie bestsellerowego przewodnika po najnowszych narzędziach i metodach związanych z uczeniem przez wzmacnianie. Zawiera wprowadzenie do teorii uczenia przez wzmacnianie, a także wyjaśnia praktyczne sposoby kodowania samouczących się agentów w celu rozwiązywania praktycznych zadań. W tym wydaniu dodano sześć nowych rozdziałów poświęconych takim osiągnięciom technologii jak dyskretna optymalizacja, metody wieloagentowe, środowisko Microsoft TextWorld czy zaawansowane techniki eksploracji. Opisano również inne zagadnienia, między innymi głębokie sieci Q, gradienty polityk, sterowanie ciągłe i wysoce skalowalne metody bezgradientowe. Poszczególne kwestie zostały zilustrowane kodem wraz z opisem szczegółów implementacji.

W książce między innymi:

- związki między uczeniem przez wzmacnianie a głębokim uczeniem
- różne metody uczenia przez wzmacnianie, w tym entropia krzyżowa, sieć DQN, a także algorytmy: aktor-krytyk, TRPO, PPO, DDPG, D4PG i inne
- praktyczne zastosowanie dyskretnej optymalizacji w celu rozwiązania problemu kostki Rubika
- trenowanie agentów przy użyciu oprogramowania AlphaGo Zero
- chatboty oparte na sztucznej inteligencji
- zaawansowane techniki eksploracyjne, w tym metody destylacji sieci

Maxim Lapan jest niezależnym badaczem z wieloletnim doświadczeniem zawodowym w dziedzinie programowania i architektury systemów. Gruntownie poznał takie zagadnienia jak duże zbiory danych, uczenie maszynowe i rozproszone systemy obliczeniowe o wysokiej wydajności. Obecnie zajmuje się zastosowaniami uczenia głębokiego, w tym głębokim przetwarzaniem języka naturalnego i głębokim uczeniem przez wzmacnianie.

ISBN 978-83-283-8052-3