Politechnika Poznańska
Wydział Informatyki i Zarządzania
Instytut Informatyki

Praca dyplomowa magisterska

UCZENIE PRZEZ DEMONSTRACJĘ NA PODSTAWIE INFORMACJI OBRAZOWEJ W ŚRODOWISKU 3D.

Wojciech Kopeć, 101675

Promotor
dr inż. Krzysztof Dembczyński

Poznań, 2017 r.

Tutaj przychodzi karta pracy dyplomowej;
oryginał wstawiamy do wersji dla archiwum PP, w pozostałych kopiach wstawiamy ksero.

Spis treści

Rozdział 1

Wstęp

Rozdział 2

Wstęp teoretyczny

2.1 Wstęp

2.2 Proces decyzyjny Markova

Środowisko, w którym porusza się agent jest matematycznie zamodelowane za pomocą Procesu decyzyjnego Markova *(ang. MDP - Markov decision process)* [REF?].

Intuicyjnie, Proces decyzyjny Markova opisuje środowisko, w którym porusza się agent i które w każdym momencie czasu ma jakiś określony stan i umożliwia wykonanie określonych akcji, za które agent może otrzymać nagrodę. Rezultat wykonania jednej z akcji, czyli stan, w którym znajdzie się agent po wykonaniu akcji, jest zależny tylko od aktualnego stanu agenta i wybranej akcji, a nie jest zależne od poprzednich stanów środowiska. Takie środowisko ma właściwość braku pamięci, albo *właśność Markova*. Im bardziej odległe w przyszłości nagrody, tym mniej są wartościowe (są dyskontowane). Celem agenta jest zgromadzenie nagród o jak największej sumie wartości.

Formalnie, Proces decyzyjny Markowa jest opisany krotką $(S, A, T(\cdot,\cdot), R(\cdot,\cdot), \gamma)$.

- S jest skończonym zbiorem możliwych stanów środowiska,
- A_s jest skończonym zbiorem możliwych wykonywalnych akcji w stanie s,
- $T_a(s, s')$, funkcja przejść, jest funkcją prawdopodobieństwa trafienia do stanu s' po wykonaniu akcji a w stanie s,
- $R_a(s, s')$, funkcja nagrody determinuje nagrodę (lub wartością oczekiwaną nagrody, obie mogą być negatywne) otrzymywaną po wykonaniu akcji a w stanie s i trafieniu na skutek tego do stanu s',
- $\gamma \in [0, 1]$ jest współczynnikiem dyskontowym, obniżającym wartość nagród uzyskanych w przyszłości.

Celem jest maksymalizacja

$$\sum_{t=0} \gamma^t R(s_t, s_{t+1})$$

gdzie kolejne t są kolejnymi momentami czasowymi. Ponadto:

- Polityką (strategią) π, realizowaną przez agenta, nazywamy funkcję $\pi : S \to A$, która określa, jak agent powinien się zachować w danym stanie w celu osiągnięcia maksymalnej możliwej nagrody.

- Funkcja użyteczności $U(s)$ lub wartości *(ang. Value)* $V(s)$ określa maksymalną oczekiwaną nagrodę, jaką agent może osiągnąć znajdując się stanie s i postępując dalej zgodnie z aktualną polityką. Poniższe równanie nazywane jest równaniem Bellmana.

$$U(s) = V(s) = (max_{a \in A(s)} \sum_{s'} T_a(s, s')(R_a(s, s') + \gamma U(s')))$$

- Funkcja Q $Q(s, a)$ określa maksymalną oczekiwaną nagrodę, jaką agent może osiągnąć wykonując w stanie s akcję a i postępując dalej zgodnie z aktualną polityką.

$$Q(s, a) = \sum_{s'} T_a(s, s')(R_a(s, s') + \gamma max_{a' \in A(s)} Q(s', a'))$$

W badanym problemie środowisko VizDoom jest *częściowo obserwowalnym procesem decyzyjnym Markova*, co oznacza, że stan obserwowany przez agenta nie zawiera pełnych informacji o środowisku. Jest też stochastyczne, co oznacza że skutki działań agenta nie są deterministyczne - wielokrotne wykonanie tej samej akcji w tym samym stanie może przynieść różne rezultaty.

Znając funkcje przejść możliwe jest iteracyjne określenie optymalnej polityki działania agenta.

2.3 Uczenie ze wzmocnieniem

W przypadku, kiedy funkcja przejść $T_a(s, s')$ albo funkcja nagród $R_a(s, s')$ nie jest znana (albo wielkość przestrzeni stanów S sprawia, że analityczne podejście nie jest możliwe), mamy do czynienia z *uczeniem ze wzmocnieniem (ang. Reinforcement learning, RL)*.

Intuicyjnie, uczenie ze wzmocnieniem opisuje sytuację, w której agent porusza się po nieznanym środowisku. Na podstawie obserwowanych rezultatów swoich działań buduje wiedzę o środowisku, pozwalającą określić strategię działania optymalną dla danej wizji środowiska. Postępując według tej strategii zdobywa dalsze doświadczenia, pozwalające uaktualniać wiedzę o środowisku i politykę agenta.

2.3.1 Uczenie ze wzmocnieniem a uczenie nadzorowane

W rozdziale 2.2 określono politykę π jako funkcję $\pi : S \to A$, określającą optymalną akcję do wykonania a dla każdego stanu s. Odpowiednikiem w uczenia nadzorowanego byłoby określenie π jako klasyfikatora $S \to A$. Do uczenia ze wzmocnieniem nie stosuje się jednak technik uczenia nadzorowanego, ponieważ, bez rozwiązania całego problemu, nigdy nie są znane "poprawneąkcje a dla danego stanu s. Mimo, że środowisko dostarcza czasem informacji zwrotnej w postaci nagród lub kar, to wykonanie danej akcji w danym stanie jest najczęściej konsekwencją całej poprzedniej sekwencji ruchów - określenie, który z ruchów w sekwencji był faktycznie kluczowy dla uzyskania określonego rezultatu jest nietrywialne.

Przykładowo, w partii szachów zwycięski ruch jest najczęściej konsekwencją konkretnych zagrań lub błędów popełnionych wiele ruchów wcześniej - algorytm uczący musi w jakiś sposób określić, które z ruchów były decydujące dla końcowego wyniku. Ruch, które doprowadzają do zwycięstwa mogą krótkodystansowo przynosić straty (np. poświęcenie figury) - algorytm uczący musi zrozumieć, że mimo negatywnej informacji zwrotnej dany ruch był porządany.

2.3.2 Zalety i zastosowanie

Uczenie ze wzmocnieniem jest narzędziem, które świetnie sprawdza się w sytuacjach, w których środowisko jest zbyt skomplikowane, żeby analitycznie znaleźć optymalną politykę działania. Dzięki temu, że model przejść i model nagród opisane są rozkładami prawdopodobieństwa uczenie ze wzmocnieniem radzi sobie bez problemu z modelowaniem zachowań w bardzo niepewnym świecie. Dzięki współczynnikowi dyskontowemu γ, możliwe jest balansowanie pomiędzy optymalizacją krótko i długoterminowych zysków.

Najważniejsza jest jednak możliwość działania bez żadnej wiedzy i silnych założeń na temat środowiska, a którym znajduje się agent. RL zakłada brak wiedzy o modelu świata, a wszystkie informacje czerpane są z doświadczeń na temat interakcji ze środowiskiem. Jest to kluczowa właściwość, ponieważ dla wielu praktycznych problemów, które adresuje RL stworzenie dokładnego modelu świata jest niemożliwe (np. stworzenie dokładnego matematycznego modelu aerodynamiki i zachowania śmigłowca), albo mimo znajomości modelu matematycznego przestrzeń możliwych stanów jest zbyt wielka, by analitycznie osiągnąć rozwiązanie (np. szachy, go).

Uczenie ze wzmocnieniem stosuje się z powodzeniem do sterowania robotami, samochodami i windami, grania w gry planszowe (szachy, backgammon, go, warcaby) i gry komputerowe.

2.4 Metody

Podejścia stosowane do uczenia ze wzmocnieniem możemy podzielić na trzy rodzaje, w zależności od typu informacji na której bazuje agent [**?**].

1. Agent z polityką - uczy się polityki $\pi : S \to A$. Przykłady:
   - algorytmy ewolucyjne (Nie uczenie ze wzmocnieniem).
   - uczenie z ekspertem (Nie uczenie ze wzmocnieniem).
2. Agent z funkcją użyteczności U - czy się funkcji U. Przykłady:
   - adaptatywne programowanie dynamiczne *(ang. adaptative dynamic programming, ADP)*.
   - metoda różnica czasowych *(ang. temporal difference learning, TDL)*.
3. Agent z funkcją użyteczności Q - czy się funkcji Q. Przykłady:
   - Q-learning (TDL)
   - SARSA *(ang. State, Action, Reward, State, Action)* (TDL)

Poniżej przedstawione zostaną najważniejsze metody.

2.4.1 Metoda różnic czasowych

Metoda różnic czasowych [REF Sutton, Bellman] opiera się na uaktualnianiu stanu wiedzy agenta na podstawie różnicy pomiędzy spodziewanym a zaobserwowanym wynikiem.

Agent trafiający do stanu s' po wykonaniu akcji a w stanie s może uaktualnić stan swojej wiedzy:

$$U(s) \leftarrow U(s) + \alpha(R(s, s') + \gamma U(s') - U(s))$$

lub

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s,s') + \gamma max_{a'} Q(s',a') - Q(s,a))$$

gdzie α jest współczynnikiem prędkości uczenia. Jeżeli α w odpowiedni sposób zmniejsza się w czasie, to TDL gwarantuje zbieżność do optimum globalnego [**?**].

2.4.2 Funkcja U, Q i SARSA

- Funkcja U (patrz: rozdział 2.2) opisuje użyteczność stanu,
- Funkcja Q (patrz: rozdział 2.2) opisuje użyteczność wykonania danej akcji w danym stanie,
- SARSA również posługuje się funkcją Q, ale w przeciwieństwie do Q-learningu aktualizuje wartości na podstawie przebytej przez agenta trajektorii, $s \rightarrow a \rightarrow s' \rightarrow a'$, a nie na podstawie wartości funkcji Q dla najlepszej akcji w stanie s'. Aktualizacja TD w SARSA-ie wygląda następująco:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s,s') + \gamma Q(s',a') - Q(s,a))$$

Mimo podobnych wzorów i definicji nauka funkcji Q ma jedną, diametralną przewagę na nauką funkcji U - funkcja Q nie wymaga znajomości modelu świata do wyboru najlepszej akcji do wykonania. Zbiór dostępnych akcji A jest znany agentowi. Przy wyborze najlepszej akcji a w stanie s:

- Agent z funkcją Q wybiera akcję $a = argmax_{a \in A} Q(s,a)$.
- Agent z funkcją U wybiera akcję, która maksymalizuje $U(s')$ - wartość stanu, do którego trafi agent: $a = argmax_{a \in A} sum_{s'} T_a(s,s') U(s')$. Obliczenie tego wyrażenia wymaga znajomości modelu przejść $T_a(\cdot,\cdot)$, czyli modelu świata. Można przyjąć, że dla trudniejszych i realnych problemów model świata nie jest dostępny.

Z tego powodu większość wiodących rozwiązań w dziedzinie uczenia ze wzmocnieniem oparta jest na Q-learningu. SARSA może zachowywać się nieznacznie lepiej, ale w większości przypadków będzie się uczyła wolniej bez wpływu na jakość agenta. Dalsza część pracy przyjmuje Q-learning jako obowiązującą metodę rozwiązywania problemu uczenia ze wzmocnieniem.

Niezależnie od metody, dla sensownej wielkości problemów uczenie ze wzmocnieniem jest wymagające obliczeniowo i czasowo. Mimo wspomagających agenta technik, nauka sprowadza się najczęściej do interaktowania ze środowiskiem metodą prób i błędów - potrzeba wiele prób i błędów, zanim agent zacznie pojmować zasady rządzące środowiskiem w którym się znajduje, a potem dużo dalszych zanim znajdzie dla danego środowiska satysfakcjonująco skuteczną politykę działania. [DOKOŃCZYĆ?]

2.5 Aproksymatory i głębokie sieci Q

Zdefiniowana w podrozdziale 2.4 przykładowa reguła aktualizacji wartości funkcji Q wygląda następująco:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s,s') + \gamma max_{a'} Q(s',a') - Q(s,a))$$

Wynika z niej, że wartość funkcji Q dla poprzedniego stanu aktualizujemy na podstawie otrzymanej nagrody i wartości funkcji Q dla stanu aktualnego. Oznacza to, że dla każdego stanu, który

analizujemy, konieczna jest znajomość jego wartości funkcji Q dla wszystkich możliwych akcji. Oznacza to, że dla dokładnego przedstawienia funkcji $Q(s, a)$ konieczne jest zapamiętanie $|S| \cdot |A|$ wartości. Co więcej, aby uzyskać sensowne wartości tej funkcji konieczne jest odwiedzenie każdego ze stanów wiele razu. Wiele z tych stanów jest też bardzo podobnych do siebie nawzajem, więc wiedza wyniesiona dla jednego stanu powinna się w pewien sposób generalizować na podobne stany.

Backgammon, jedna z gier planszowych służąca jako benchmark algorytmów uczenia ze wzmocnieniem, ma 10^{20} możliwych stanów, a szachy 10^{40}. Jeden obraz 90x60 pixeli w skali szarości, używany jako zapis stanu w większości poniższej pracy może przyjąć 256^{5400} różnych kombinacji. Wiele realnych problemów opisanych jest wartościami ciągłymi zamiast dyskretnymi, a praktyczna liczba ich możliwych stanów rośnie wykładniczo wraz ze wzrostem dokładności pomiaru.

2.5.1 Aproksymatory funkcji Q

Rozważanie i zapamiętanie każdego stanu z osobna dla bardziej skomplikowanych problemów jest niemożliwe i niepraktyczne ze względu na podobieństwo wielu stanów. Rozwiązaniem jest wykorzystanie *Aproksymatora funkcji Q* - niestablicowanej, parametrycznej funkcji pary (stan,akcja) $\hat{Q}_\theta(s, a)$, gdzie θ jest wektorem parametrów funkcji.

Aproksymator [**?**]:

- musi być łatwo obliczalny,
- kompresuje dużą przestrzeń stanów w znacznie mniejszą przestrzeń parametrów,
- uogólnia wiedzę na temat podobnych stanów.
- w większości przypadków przyspiesza uczenie w stosunku do wersji stablicowanej ze względu na uogólnianie wiedzy

Jako jedne z pierwszych i prostszych aproksymatorów zastsowano funkcje liniowe, opierające się na ręcznie zdefiniowanych cechach: $\hat{Q}_\theta(s, a) = \theta_0 + \theta_1 cecha1 + \theta_2 cecha2 + ... + \theta_n cechan$. Przykładem zastosowania może być gra w warcaby, opisana w [REF SAMUEL 1959]. Zaletami liniowego aproksymatora są prostota i łatwość interpretacji, a także szybkość obliczania i nauki. Dalszym krokiem było wykorzystanie sieci neuronowych jako aproksymatorów przy grze w Backgammona [REF TESAURO 1992]. W pierszej wersji algorytmu wykorzystano ręcznie zaprojektowane cechy, w kolejnych wykorzystano prawie surową informację o rozkładzie pionków na planszy. Sieć neuronowa jest bardziej skomplikowanym i trudniejszym do nauczenia aproksymatorem, ale jest w stanie zamodelować znacznie bardziej złożone funkcje.

2.5.2 Uczenie na podstawie surowych danych obrazowych - Atari 2600

Jednym z największych przełomów uczenia ze wzmocnieniem ostatnich lat była praca [REF MNIH 2015], w której autorzy wykorzystali głębokie sieci neuronowe do stworzenia agenta potrafiącego grać na ludzkim poziomie w klasyczne gry z Atari 2600 wykorzystując jako reprezentację stanu jedynie surowy zapis obrazu 2D. Dotychczas, jak w poprzednich przykładach, algorytmy uczenia ze wzmocnieniem opierały się na ekspercko wykreowanej reprezentacji stanów - [REF MNIH 2015] pokazali, że możliwe jest stworzenie rozwiązania, które samo będzie potrafiło ekstrahować wysokopoziomowe cechy z niskopoziomowych danych. Zaproponowana architektura, jak również pomysłowe usprawnienia zwiększające stabilność uczenia zaproponowane w tej pracy, a opisane

w rozdziale 2.6 stanowią obecnie podstawę dla większości dalszych badań na temat uczenia ze wzmocnieniem.

Jako aproksymator funkcji Q wykorzystano głęboką sięć neuronową. Z tego powodu opisywane podejście określa się często skrótem DQN *ang. Deep Q Network)*, czyli głęboka sieć Q. Zastosowana architektura wygląda następujaco:

[RYSUNEK SIECI]

Analogicznie jak w obenie stosowanych architekturach rozpoznawania obrazu, pierwsze warstwy sieci to warstwy konwolucyjne, które wykrywają kolejno nisko i wysokopoziomowe cechy obrazu. Dalsze warstwy w pełni połączone łączą informacje z warstw konwolucyjnych we wnioski na temat stanu świata, dla których następne warstwy mogą określić wartość funkcji Q.

2.6 Q-learning - usprawnienia

Skuteczność Q-learningu może zostać drastycznie polepszona dzięki zastosowaniu następujących technik.

2.6.1 Pamięć powtórek

Szkielet uczenia ze wzmocnieniem opiera się na zbieraniu doświadczeń i uaktualnianiu na ich podstawie stanu wiedzy agenta. W praktyce, doświadczenia zbierane bezpośrenio po sobie są silnie skorelowane - przykładowo agent uczący się na podstawie obrazu jazdy samochodem w kolejnych klatkach widzi niemal identyczne obrazy i wykonuje najczęsciej te same akcje. Oznacza to, że aktualizowanie wiedzy agenta na podstawie nowych doświadczeń, czy to pojedynczo czy w paczkach, będzie skutkować funkcją obciążoną w stosunku tych doświadczeń.

Aby temu zapobiec w [REF] zaproponowano metodę pamięci powtórek *(ang. Replay memory)*. Metoda opiera się na zapamiętywaniu znacznej ilości najnowszych doświadczeń. Po każdym kroku nowe doświadczenia dodawane są do pamięci (w przypadku braku miejsca zastępując nastarsze), a następnie z pamięci wybierana jest losowa próbka doświadczeń, na podstawie których aktualizowana jest wiedza agenta. Dzieki tej technice dane użyte do nauki przez agenta są nieskorelowane i niezależne. Dodatkowo, dzięki dostępowi do starszych danych agent jest mniej podatny na obniżanie jakości gry na skutek krótkotrwałych spadków wyników.

Dalsze rozszerzenia metody mają na celu np. priorytetyzowanie używania do nauki najważniejszych doświadczeń [REF].

2.6.2 Zamrażanie docelowej sieci

Podobnie jak Pamięć powtórek, zamrażanie docelowej sieci *(ang. Target network freezeing / fixed target network)* służy zmniejszenia skutków obciążenia agenta wynikającego ze sposobu zbierania próbek. Zamrażanie sieci zakłada utrzymywanie dwóch funkcji Q - starej i nowej. Agent działa na podstawie nowej funkcji, ale wartości Q "docelowychśtanów pobierane są ze starej funkcji. Co jakiś czas do starej funkcji przepisywana jest nowa funkcja.

Technika ma na celu zniwelowanie oscylacji i ustabilizowanie zachowań agenta. Dzięki wykorzystaniu źamrożonych"wartości do nauki funkcji Q zerwane jest sprzężenie zwrotne pomiędzy zebranymi danymi a wartościami docelowymi.

2.6.3 Kształtowanie

W wielu zadania stawianych przed uczeniem ze wzmocnieniem osiągnięcie celu jest bardzo trudne, a agent dostaje nagrody dopiero po osiągnięciu stanów terminalnych, albo przynajmniej na zaawansowanym etapie zadania. Agent uczący się na podstawie się na podstawie prób, błędów i losowych akcji nie jest najczęściej w stanie wykonać wystarczająco dużo zadania, żeby dostać informację zwrotną w postaci nagrody, a więc nie ma jak się uczyć lub uczenie następuje bardzo wolno.

Kształtowanie *(ang. Shaping)* zakłada sztuczne wprowadzenie do środowiska dodatkowych nagród, które agent będzie dostawał po wykonaniu etapów pośrednich zadania. Przykładowo, przy grze w szachy, w której agent dostaje nagrodę tylko za wygraną lub przegraną (1 lub -1) można byłoby wprowadzić nagrodę 0.1 za zbijanie figur przeciwnika.

Kształtowanie wymaga możliwości ingerencji w środowisko albo percepcję agenta (rozpoznawanie, kiedy agent powinien dostać sztuczną nagrodę i ingerowanie w odczyty nagrody dokonywane przez agenta). Co ważniejsze, wymaga wiedzy eksperckiej na temat zadania wykonywanego przez agenta (możliwość określenia sensownych etapów zadania, na których agent miałby dostać sztuczną nagrodę) i wiedzy na temat środowiska, w którym agent się porusza (wysokość sztucznej nagrody musi być dopasowana do prawdziwych nagród, które może dostawać agent). Dodatkowo, kroki określone przez eksperta mogą wymuszać nieoptymalną politykę działania i powstrzymać agenta przed odkryciem optymalnych strategii.

2.7 Eksploracja

W uczeniu ze wzmocnieniem agent posiada umiejętność uczenia się na podstawie odbytych doświadczeń. Na początku każdej nauki jest jednak zupełnie nieświadomy zasad świata w którym się znajduje i nie jest w stanie podejmować sensownych działań. Konieczna jest metoda pozwalająca na zdobywanie nowych doświadczeń przy jednoczesnej możliwości szlifowania i ulepszania opracowanych wcześniej przez agenta sposobów. Proces nakładania agenta do zbadania nieznanych jeszcze obszarów przestrzeni stanów nazywany jest eksploracją.

2.7.1 Algorytm e-zachłanny

Podstawowym i często używanym podejściem do eksploracji jest wspomniany wcześniej e-zachłanny algorytm, w którym agent z zadanym prawdopodobieństwem e zamiast akcji optymalnej względem aktualnej polityki wykonuje akcję losową. Takie zachowanie jest nieskuteczne, kiedy optymalne zachowanie agenta wymaga zaplanowania złożonych lub dalekosiężnych planów.

2.7.2 Algorytm R-max

Prostym, ale skutecznym i posiadającym teoretyczne gwarancje zbieżności algorytmem jest zaproponowany w [**?**] R-max, realizujący ideę optymizmu wobec niepewności. Podstawą R-maxa jest optymistyczna inicjalizacja – przed rozpoczęciem uczenia funkcja aproksymacyjna powinna zwracać maksymalną nagrodę dla wszystkich stanów i akcji. W ramach działania agent będzie uaktualniał (czyli obniżał) spodziewaną nagrodę w odwiedzonych stanach.

Największa spodziewana nagroda będzie zwracana dla zachowań, które agent odkrył już jako zyskowne i dla zachowań jeszcze nieodkrytych (dla których funkcja aproksymacyjna nie jest jeszcze poprawiona). Ten prosty zabieg powoduje, że algorytmy uczenia ze wzmocnieniem naturalnie balansują pomiędzy eksploracją i intensyfikacją przeszukiwania bez dodatkowych modyfikacji.

Od strony teoretycznej zaletą R-maxa jest duża ogólność zastosowania – algorytm wymaga spełnienia bardzo luźnych założeń, badany proces nie musi być nawet procesem decyzyjnym Markowa.

2.7.3 Przewidywanie przejść za pomocą autoenkodera

W [?] autorzy zaproponowali rozwiązanie, które pozwala ocenić, w jakim stopniu odwiedzony stan jest dla agenta nowością. Opiera się ono na stworzeniu aproksymatora, którego zadaniem jest przewidywanie, jaki stan osiągnie agent po wykonaniu danej akcji w danym stanie. Predykcja porównywana jest z faktycznie osiągniętym stanem, a wielkość błędu jest wyznacznikiem nowości stanu – im większy błąd predykcji, tym bardziej nieznany stan, za co przyznawana jest większa nagroda eksploracyjna. Jak większość opisywanych publikacji, w [?] rozwiązywano problem uczenia agenta grania w gry zręcznościowe na podstawie surowego obrazu z wykorzystaniem Q-learningu i głębokich sieci neuronowych.

Pierwszą kwestią do rozwiązania przy implementacji pomysłu jest metryka pozwalająca określić podobieństwo stanów. Próby predykcji wartości konkretnych pikseli opisane przez autorów nie przyniosły efektów, generując tylko szum. Zamiast tego trenowano głęboką sieć neuronową do przewidywania następnego stanu i wykorzystano jedną z ukrytych warstw tej sieci o mniejszej liczbie jednostek jako enkoder stanu, który przenosi surowy obraz do przestrzeni o znacznie mniejszej liczbie parametrów. Za miarę podobieństwa między stanami przyjęto odległość kartezjańską parametrów uzyskanych z zakodowania dwóch stanów. Zakodowanymi stany używane były do wytrenowania właściwego, prostszego aproksymatora, na podstawie błędu którego określano nowość stanu. Dla każdego przejścia między stanami przyznawano sztuczną nagrodę zależną od nowości odwiedzonego stanu.

Potencjalnym problemem związanym z tym podejściem jest to, że Q-learning stara się nauczyć funkcji, która jest niestacjonarna Autorzy piszą, jednak, że w praktyce nie stanowiło to problemu.

2.7.4 Bootstrapowane DQN

Innym taktykę dywersyfikacji przeszukiwania przy wykorzystaniu głębokiej sieci neuronowej zaprezentowano w [?]. Podobnie jak w [?] uczono sieć funkcji Q, jednak zamiast pojedynczej funkcji Q trenowano jednocześnie K funkcji Q, przy czym każda trenowana była tylko na podzbiorze przykładów uzyskanym za pomocą techniki bootstrapingu. Każda funkcja Q reprezentowana była przez jedną K „głów" wspólnej wielopoziomowej sieci.

Dla każdego z epizodów wybierana losowo była jedna głowa – funkcja Q i przez cały epizod agent kierował się polityką optymalną dla tej funkcji Q.

Dzięki temu zabiegowi każda z sieci Q była nauczona na podstawie nieco różnych doświadczeń i prezentowała nieco inną politykę działania. Nowe informacje o porządanych zachowaniach były prędzej czy później propagowane do każdej zgłów, ale jednocześnie różnorodność zachowań była wystarczająca, żeby utrzymać eksplorację.

Autorzy raportują spowolnienie uczenia o zaledwie 20% w stosunku do normalnej sieci, ale przeprowadzone w ramach tej pracy eksperymenty dawały znacznie gorsze rezultaty.

2.8 Uczenie przez demonstację

2.8.1 Uczenie przez demonstrację a uczenie nadzorowane

Najprostszym podejściem do uczenia przez demonstrację jest traktowanie go jak każdego innego problemu uczenia nadzorowanego, przy czym w przeciwieństwie do minimalizowania kosztu działania agenta minimalizowana jest różnica pomiędzy pomiędzy polityką wyuczonego agenta a polityką eksperta. Najprostsze podejście zakłada jednak, że dane uczące i testowe są niezależne i mają jednakowy rozkład, podczas gdy przy uczeniu przez demonstrację nauczona polityka ma bezpośredni wpływ na osiągane później stany, na podstawie których dana polityka będzie sprawdzana. Jak dowiedziono w [**?**] wynikający z tego błąd rośnie kwadratowo w stosunku do czasu trwania epizodów – gdy klasyfikator popełni błąd w odwzorowywaniu polityki eksperta najprawdopodobniej trafi do stanu nieodwiedzonego przez eksperta, co z dużym prawdopodobieństwem oznacza popełnianie następnych błędów, ponieważ uczeń nie miał jak nauczyć się „podnoszenia się" po błędach.

Jednym ze sposobów radzenia sobie z tym problemem jest wprowadzanie małych zmian podczas iteracji polityki, dzięki czemu rozkład stanów dla nowej polityki jest bliski staremu. Idea polega na zaczynaniu od polityki całkowicie identycznej z polityką eksperta i stopniowym przechodzeniu na politykę wyuczoną. Aby to osiągnąć można wymagać, aby podczas uczenia uczeń mógł w każdej chwili zapytać eksperta, jakie akcje ekspert podjąłby w danym stanie. Dany układ wymaga większej interakcji, ale może być zrealizowany dla wielu z praktycznych przykładów wykorzystania uczenia przez demonstrację.

Pierwszym podejściem opisywanym przez [**?**] jest uczenie w przód. Podejście opiera się na przeprowadzeniu kilku powtórzeń uczenia, gdzie w każdym kroku następuje uczenie się jednej polityki w jednym, konkretnym, momencie. Jeżeli uczenie będzie przeprowadzone po kolei dla każdego kolejnego kroku w czasie, to próbka uzyskanych stanów, na których prowadzone jest dalsze uczenie odpowiada dystrybucji stanów testowych, a algorytm może odpytać eksperta o właściwe działanie w osiągniętych stanach, dzięki czemu ekspert ma okazję zaprezentować jak „podnosić się" po popełnieniu błędów przez klasyfikator. Powyższe podejście działa tylko dla zadań o skończonym horyzoncie czasowym, wymaga dużej interakcji z ekspertem i możliwości zrestartowania systemu i dokładnego odtworzenia uzyskanego wcześniej stanu, co w wielu przypadkach nie będzie możliwe do zrealizowania.

W celu wyeliminowania tych ograniczeń [**?**] proponują Iterowany Probabilistyczny Mieszający algorytm. Opierając się na algorytmie iterowania polityki algorytm w każdym kroku stosuje nową stochastyczną politykę wybierając z zadanym prawdopodobieństwem pomiędzy wykonywaniem polityki wyuczonej w poprzednim kroku i konstruowanej w danej iteracji nowej polityki, przy czym prawdopodobieństwo wyboru nowej polityki jest niewielkie. Algorytm zaczyna od dokładnego wykonywania akcji eksperta. W każdej kolejnej iteracji algorytmu prawdopodobieństwo odpytania eksperta jest coraz niższe i zbiega się do 0. Opisane rozwiązanie zostało z powodzeniem przetestowane na przykładzie grania w proste gry, gdzie danymi wejściowymi był obraz z ekranu. Autorzy zdecydowali się na klasyfikator wybierający konkretne akcje dla danego stanu, zamiast częściej używanego w uczeniu ze wzmocnieniem klasyfikatora odwzorowującego funkcję kosztu. Wadą tego podejścia jest brak odrzucania nieskutecznych polityk podczas iteracji, co może prowadzić do niestabilnych wyników.

Wykorzystanie analogicznego rozwiązania proponują [**?**]. Ich propozycja zakłada wybieranie z prawdopodobieństwem e polityki eksperta i z prawdopodobieństwem $1 - e$ polityki wyuczonej. Początkowa wartość e powinna wynosić 1, aby klasyfikator mógł nauczyć się odtwarzać politykę

eksperta. Wraz z postępem nauki e powinno stopniowo maleć do 0, aby klasyfikator miał szanse nauczyć się stanów nieodwiedzonych przez eksperta.

W kolejnej publikacji [**?**] prezentują nowe podejście, nazwane Agregacją Zbioru Danych. W uproszczeniu, podejście to jest następujące: W pierwszej iteracji algorytm zbiera dane testowe stosując politykę pokazaną przez eksperta, po czym trenuje klasyfikator odwzorowujący zachowanie eksperta na danym zbiorze danych. W każdej kolejne iteracji algorytm stosuje politykę wygenerowaną w poprzedniej iteracji i dodaje dane uzyskane podczas jej stosowania do zbioru danych, po czym trenuje klasyfikator by odwzorowywał zachowanie eksperta na całym zbiorze danych. Podobnie jak w poprzednim algorytmie, żeby przyspieszyć uczenie na pierwszych etapach algorytmu, dodano opcjonalną możliwość odpytania eksperta o jego wybór akcji. Uzyskane z pomocą tej metody wyniki są wyraźnie lepsze od wyników uzyskanych za pomocą metody opisanej w poprzednim paragrafie.

2.8.2 Podążanie za ekspertem a przewyższanie eksperta

Dla wielu praktycznych problemów polityka eksperta może nie być optymalna. Algorytm, który stara się tylko i wyłącznie odwzorować politykę eksperta będzie generował w takiej sytuacji nieoptymalne wyniki, które w wielu praktycznych sytuacjach mogą znacznie odbiegać od optimum. Prostym rozwiązaniem tego problemu przedstawionym w [**?**] jest stosowanie e-zachłannej strategii – w każdym ruchu algorytm może wybrać z małym prawdopodobieństwem e wykonanie losowej akcji zamiast akcji optymalnej według wyuczonej polityki. Dzięki temu algorytm może znaleźć lokalne optimum bliskie polityce eksperta. Warto zauważyć, że wymusza to posługiwanie się całościową nagrodą (kosztem) wykonania zadania jako celem optymalizacji, w przeciwieństwie do prostszego minimalizowania różnicy pomiędzy wynikami wyuczonej polityki a polityki eksperta.

2.9 Odwrócone uczenie ze wzmocnieniem *(ang. Inverse reinforcement learning, IRL)*

Użycie kopiowania zachowań i wywodzących sie z niego metod 2.8 pozwala na wytrenowanie agenta odruchowego, który dla danego stanu będzie potrafił określić optymalną akcję na jeden krok do przodu. Taki agent zna optymalną politykę, ale nie jest świadomy jej powodów. Odwrócone uczenie ze wzmocnieniem opiera się na założeniu, że w ogólności optymalna polityka agenta nie stanowi najlepszego i najbardziej zwięzłego opisu zadania podstawionego przed agentem - najbardziej precyzyjnym opisem, pozwalającym na większą dowolność i adaptację jest znajomość funkcji nagród $R_a(\cdot,\cdot)$.

Oczywiście, bez modelu środowiska funkcja $R_a(\cdot,\cdot)$ nie jest dostępna, dlatego zadaniem postawionym przed odwróconym uczeniem ze wzmocnieniem jest odtworzenie $R_a(\cdot,\cdot)$ na podstawie dostarczonych trajektorii eksperta.

Zadanie odwróconego uczenia ze wzmocnieniem jest znacznie trudniejsze od zwykłego uczenia ze wzmocnieniem. Przede wszystkim, IRL musi się zmierzyć z 2 problemami:

- Niejasność $R_a(\cdot,\cdot)$ - w większości przypadków, dla danych trajektorii eksperta istnieje nieskończenie wiele pasujących $R_a(\cdot,\cdot)$. Formalnie oznacza to, że IRL nie ma zdefiniowanego poprawnego rozwiązania.
- Złożoność obliczeniowa - samo uczenie ze wzmocnieniem jest bardzo wymagające obliczeniowo. W odwróconym uczeniu ze wzmocnieniem, sprawdzanie $R_a(\cdot,\cdot)$ uzyskanych w kolejnych krokach wymaga każdorazowego rozwiązywania problemu uczenia ze wzmocnieniem na

podstawie aktualnej funkcji $R_a(\cdot,\cdot)$, co oznacza, że IRL wymaga większego o rząd wielkości kosztu obliczeniowego niż RL.

Przykładowe zastosowania odwróconego uczenia ze wzmocnieniem to parkowanie samochodu [REF Abbeel, Dolgov, Ng and Thrun, IROS 2008], nawigacja na podstawie obrazów satelitarnych [REF Ratliff, Bagnell and Zinkevich, ICML 2006] albo wykonywanie ewolucji helipkopterem [REF Abbeel, Ng].

2.10 Środowisko VizDoom

Środowisko VizDoom, przedstawione w [**?**], jest narzędziem do testowania algorytmów sterowania na podstawie surowych danych o obrazie 3D. Środowisko bazujące na klasycznej grze Doom, w której gracz widzi trójwymiarowy świat z perspektywy pierwszoosobowej i strzela do piekielnych potworów. W stosunku do nauki w środowisku 2D, takim jak Atari 2600 [REF], nauka w środowisku 3D jest wielkim krokiem naprzód i stanowi znacznie lepsze przybliżenie nauki w realnym świecie.

VizDoom oferuje wygodny interfejs, który doskonale wpisuje się w standardowy szkielet metod uczenia ze wzmocnieniem. VizDoom potrzebuje mało zasobów, może działać bez środowiska graficznego, jest wydajny, pozwala na uruchamianie wielu instancji równolegle oraz na wygodne tworzenie nowych scenariuszy dopasowanych do potrzeb problemów badawczych. Vizdoom udostępnia interfejs programistyczny dla Pythona, Javy, C++ i Lua. Preferowany i najbardziej rozwijany jest Python.

Przykładowe minimalne użycie środowiska VizDoom przedstawione jest poniżej (prezentowany kod jest zmodyfikowanym przykładem *scenarios.py* dołączonym do środowiska):

```
from vizdoom import DoomGame

game = DoomGame()

game.load_config("../../scenarios/basic.cfg")

game.set_screen_resolution(ScreenResolution.RES_640X480)
game.set_window_visible(True)
game.init()

# Creates all possible actions depending on how many buttons there are.
actions = prepare_actions(game.get_available_buttons_size())

episodes = 10

for i in range(episodes):
    game.new_episode()
    while not game.is_episode_finished():
        # Gets the state and possibly to something with it
        state = game.get_state()
        # Makes a random action and save the reward.
        reward = game.make_action(choose(state, actions))
        new_state  = game.get_state()
        learn(state,game.get_last_action(), reward, new_state)
```

2.11 Używane terminy

W dalszej częsci pracy używane będą następujące terminy:

- *agent/gracz* - program sterujący agentem poruszającym się w badanym środowisku,

- *ekspert* - człowiek (*ludzki ekspert*) lub zewnętrzny program (*ekspert komputerowy*), który zna optymalną (lub bliską optymalnej) politykę działania i potrafi udzielić agentowi informacji na temat realizacji tej polityki i oceniać politykę agenta,
- *epizod/gra* - ciąg interakcji agenta lub eksperta z otoczeniem, od stanu początkowego do stanu terminalnego.
- *klatka/krotka/doświadczenie* - pojedyncza informacja s, a, s', r zebrana przez agenta lub eksperta w ramach jednego kroku interakcji ze środowiskiem.
- *trajektoria* - uporządkowana lista krotek pochodząca z jednego epizodu.

Rozdział 3

Zaimplementowane podejścia

W poniższym rozdziale przedstawione zostaną zaimplementowane i analizowane w pracy podejścia.

Każde z rozwiązań działa w ramach wspólnego szkieletu, bazującego na przykładowych rozwiązaniach towarzyszących środowisku VizDoom. Dzieki temu możliwe jest bezpośrednie porównanie zachowania różnych podejść przy zmianie tylko kluczowych algorytmów przy zachowaniu niezmienności pozostałych czynników.

3.1 Wykorzystane scenariusze

Eksperymenty przeprowadzono na następujących scenariuszach, reprezentujących łatwy, sredni i wysoki poziom trudności.

Basic

Sceneria składa się z prostokątnego pomieszczenia. Agent startuje w jednym końcu pomieszczenia, po środku ściany, a w losowym miejscu pod przeciwległą ścianą znajduje się pojedynczy, nieruchomy przeciwnik.

Agent może atakować i poruszać się bokiem w lewo i prawo. Agent ma ograniczoną amunicję i dostaje punkt za trafienie przeciwnika.

Strategia optymalna polega na przesunięciu się w kierunku przeciwnika i oddaniu do niego pojedynczego strzału.

Rysunek 3.1: Scenariusz Basic

Defend the center

Sceneria składa się z kolistej areny. Agent znajduje się na środku areny, a na jej krańcach losowo pojawiają się przeciwnicy, którzy poruszają się w stronę agenta, a po dotarciu do niego zadają mu obrażenia. Są dwa rodzaje przeciwników różniących się wyglądem i szybkością poruszania.

Agent może atakować i kręcić się wokół własnej osi w lewo i prawo. Agent ma ograniczoną amunicję i dostaje punkty za każde trafienie przeciwnika.

Strategia optymalna polega na kręceniu się w jedną strone w kółko, ignorowaniu odległych przeciwników i strzelaniu do bliskich, priorytetyzując szybszych przeciwników. Ignorowanie dalekich przeciwników jest konieczne, żeby w czasie strzelania do nich inni przeciwnicy nie zaszli agenta od tyłu - optymalna strategia wymaga zajmowania się najpierw najbliższym zagrożeniem.

Rysunek 3.2: Scenariusz Defend the center

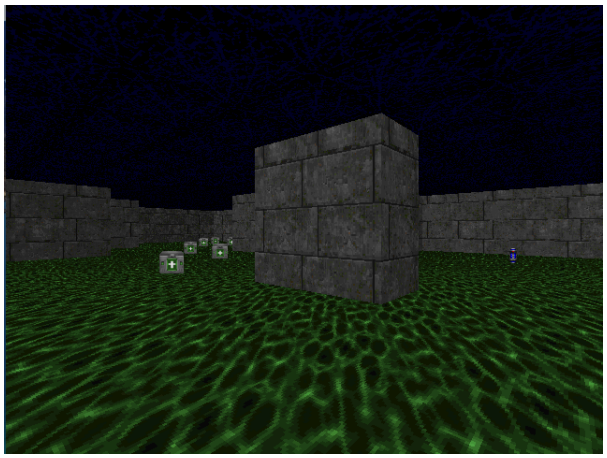
Health gathering supreme

Sceneria składa się z labiryntu, którego podłoga jest pokryta kwasem. Agent startuje w losowym miejscu labiryntu. W tym scenariuszu nie ma ruchomych przeciwników. Na podłodze labiryntu pojawiają się losowo apteczki, które dodają agentowi punkty życia i miny, które zabierają agentowi punkty życia. Kwas na podłodze nieustannie odbiera agentowi punkty życia.

Agent może iść prosto, iść na ukos i kręcić się w okół własnej osi w lewo i prawo. Agent dostaje punkty za pozostawanie przy życiu - im dłużej potrwa gra, tym większą sumę punktów zdobędzie agent.

Strategia optymalna polega na chodzeniu po labiryncie, niewchodzeniu na miny i zbieraniu apteczek, preferując kierowanie się do dużych skupisk apteczek. Wskazane jest unikanie zbierania pojedynczych, odizolowanych apteczek, gdyż liczba punktów życia uzyskana z takiej apteczki może być mniejsza niż liczba punktów życia straconych na dotarcie do apteczki. Wskazane jest niepozostawanie w jednym obszarze labiryntu, ponieważ nowe apteczki mogą nie pojawiać się wystarczająco szybko, żeby utrzymać agenta przy życiu.

Co istotne, że w tym scenariuszu bardzo często optymalna decyzja nie jest jasna - sensownie działający Ekspert i agent mogą często wybierać pomiędzy wieloma poprawnymi drogami i zachowaniami.



Rysunek 3.3: Health gathering supreme

3.2 Q-learning

Q-learning, opisany dokładniej w [ROZDZIAŁ O TEORII], jest najpopularniejszą metodą uczenia ze wzmocnieniem. Agent uczy się wartości funkcji Q odwiedzanych par stan → akcja. Wartość funkcji Q dla danego stanu odpowiada zdyskontowanej sumie nagród, jaką można uzyskać po trafieniu do danego stanu, przy założeniu wykonywania dalej optymalnych akcji. Agent rozpoczyna działając losowo, a potem uczy się łącząc działania losowe z działaniami optymalnymi według aktualnego stanu wiedzy i obserwując ich rezultaty.

Problemem Q-learningu jest konieczność długotrwałego uczenia agenta, a odpowiedzią na ten problem jest uczenie z Ekspertem. Wyniki Q-learningu będą stanowiły punkt odniesienia dla wyników innych metod.

3.2.1 Implementacja

Zastosowana implementacja opiera się na głębokiej sieci neuronowej jako aproksymatorze funkcji Q. Jest wyposażona w pamięć powtórek *(ang. replay memory)*[REF?], a do eksploracji używana jest metoda e-zachłanna *(ang. e-greedy)*[REF?]. Schemat sieci neuronowej jest następujący.

[SCHEMAT SIECI]

3.2.2 Zachowanie

Uczenie agenta trwa długo. Zastosowane proste rozwiązanie jest w stanie nauczyć się poprawnego działania dla scenariusza Basic. Dla Defend the center agent zatrzymuje się w optimum lokalnym, polegającym na częstym strzelaniu i kręceniu się w kółko bez przejmowania się konkretnymi przeciwnikami, a jego zachowanie nie wygląda sensownie. W Health gathering supreme agent nie potrafi sensownie poruszać się po labiryncie.

Zastosowanie bardziej wyrafinowanych metod uczenia ze wzmocnieniem, których implementacja wykracza poza zakres tej pracy, pozwala poprawić wyniki na trudniejszych scenariuszach. Dla Defend the center możliwe jest uzyskanie wyników bliskim optymalnym, ale do osiągnięcia sensownego zachowania w Health gathering supreme konieczna jest na przykład pomoc z zewnątrz programu w postaci techniki kształtowania *(ang. shaping)*[REF]. Mimo dobrych wyników, metody te ciągle wymagają przeanalizowania wielkich ilości klatek - wymagane liczby klatek są kilka rzędów wielkości większe niż liczba potrzebna do uczenia z Ekspertem.

3.2.3 Wnioski

Q-learning potrzebuje dużo czasu, żeby uzyskać przywoicie zachowującego się agenta, ma również problem z wychodzeniem z optimów lokalnych polityki zachowań (szczególnie przy zastosowanej prostej implementacji). Dobrze sprawdza się za to w ewolucyjnym, stopniowym poprawianiu zachowań agenta.

3.3 Kopiowanie zachowań

Kopiowanie zachowań (ang. *Behavioral Cloning*) [REF] stanowi najbardziej podstawowe podejście do uczenia z Ekspertem. Na podstawie zebranych trajektorii Eksperta uczony jest klasyfikator, który przyjmując na wejściu stan s ma za zadanie przewidzieć, jaką akcję a wykonałby w danej sytuacji Ekspert.

Mimo że dla wielu problemów Kopiowanie zachowań jest nieskuteczne (powody opisane są w [ROZDZIAŁ O TEORII]), na wyniki osiągane na badanych scenariuszach VizDoom są bardzo zadawalające.

Kopiowanie zachowań jest podstawą bardziej zaawansowanych technik opisanych w dalszych punktach.

Ważną różnicę w stosunku do Q-learningu stanowi fakt, że czas trwania metody ogranicza się w znaczącej częsci do czasu zbierania trajektorii prezentacyjnych przez Eksperta. Czas trenowania klasyfikatora na zebranych danych powinien być pomijalny w stosunku do czasu zbierania. Oznacza to, że uczenie agenta za pomocą Kopiowania zachowań trwa znacznie krócej niż za pomocą np. Q-learningu, w którym, dla wielu praktycznych problemów, agent musi grać przez przez wiele milionów klatek, by osiągnąć zadawalające wyniki.

3.3.1 Implementacja

Kopiowanie zachowań sprowadza się do klasycznego problemu uczenia nadzorowanego z wieloma etykietami (możliwymi akcjami), z których tylko jedna etykieta na raz jest poprawna. Dane wejściowe stanowią obrazy przedstawiające stan, wynikiem jest etykieta akcji, którą należy wykonać. Jako klasyfikatora użyto głębokiej sieci neuronowej, o artchitekturze bazującej na architekturze z 3.2.

Różnica pomiędzy architekturami sporowadza się do uczenia i interpretacji wyników. Przy Q-learningu sieć musi przewidywać wartość funkcji Q dla wszystkich akcji, a przy Kopiowaniu zachowań wystarczy określenie najbardziej pasującej akcji. Wykonane i przewidywane akcje są zakodowane za pomocą *one-hot encoding*, a wynik uzyskiwany jest przez zastosowanie funkcji *softmax* na wartościach q z architektury Q-learningu. Schemat sieci wygląda następująco.

[SCHEMAT SIECI]

Co istotne, dla większości scenariuszy problem uczenia stan $\rightarrow$ akcja ma niezbalansowany zbiór danych. Akcje „strzelaj" występują znacznie rzadziej niż akcje ruchu. Oznacza to, że klasyfikator naiwnie nauczony na niezmienionym zbiorze danych, mimo dobrej teoretycznej trafności, będzie zupełnie nieskuteczny (np. nie wybierze nigdy akcji „strzelaj").

Problem ten został rozwiązany przez zbalansowanie zbioru danych przy użyciu metody *oversampling* [REF?], która polega na wielokrotnym uwzględnieniu w zbiorze uczącym przykładów mniej licznych klas w taki sposób, żeby liczność przykładów dla każdej z klas w uzyskanym zbiorze danych była podobna. W zastosowanej implementacji dla każdej akcji przetrzymywany jest oddzielny zbiór danych, a użyte do uczenia próbki składają się w równych proporcjach z przykładów zastosowania każdej akcji.

Warto zauważyć, że architektura Q-learningu wymaga, żeby każda możliwa akcja była zdefiniowana oddzielnie, łącznie z akcjami stanowiącymi złożenie innych, podstawowych akcji. Przykładowo akcje „lewo", „prosto" i „lewo i prosto" są dla modelu zupełnie niezwiązane, mimo że często można byłoby stosować je zamiennie. W przypadku Kopiowania zachowań możliwe byłoby stworzenie klasyfikatora stan → akcja, który jest jednocześnie klasyfikatorem binarnym dla każdej podstawowej akcji z osobna. Taki klasyfikator mógłby zamiast wybierać pomiędzy „lewo", „prosto" i „lewo i prosto" zdecydować „lewo" - tak i „prosto" - tak, uzyskując „lewo i prosto". Jednakże takie rozwiązanie nie zostało w tej pracy zbadane.

3.3.2 Techniczna implementacja

Zbieranie danych zostało zrealizowane za pomocą trybu SPECTATOR środowiska VizDoom, pozwalającemu agentowi obserować grę człowieka. Podczas gry Eksperta zapisywane są stany, akcje i nagrody dla każdej kolejnej klatki. Trajektoria Eksperta serializowana jest do pliku za pomocą narzędzia *pickle* dostępnego dla języka python.

Eksperta gra przy rozdzielczości 640x480 pikseli, i takiej wielkości obrazy zapisywane są do pliku z trajektorią. Konsekwencją są bardzi duże rozmiary plików (3GB dla 6 tysięcy klatek). Obrazy nie są zmniejszane przed zapisem, żeby umożliwić swobodne manipulowanie wielkością obrazów używanych do uczenia klasyfikatora, bez konieczności generowania nowych trajektorii Eksperta przy innych ustawieniach obrazu.

Tryb SPECTATOR ustawiany jest w następujący sposób.

```
game.set_window_visible(True)
game.set_mode(Mode.SPECTATOR)
```

Trajektoria Eksperta zbierana i zapisywana jest następująco.

```
    game.new_episode()
    while not game.is_episode_finished():
        state = game.get_state()
        game.advance_action()
        next_state = game.get_state()
        last_action = game.get_last_action()
        reward = game.get_last_reward()
        isterminal = game.is_episode_finished()

        print("State #" + str(state.number))
        print("Game variables: ", state.game_variables)
        print("Action:", last_action)
        print("Reward:", reward)
        print("====================")
        memory.append((state.screen_buffer,last_action, next_state.screen_buffer, reward,
     isterminal))
```

Zapis trajektorii do pliku wygląda następująco.

```
with open('recorder_episode.pkl', 'wb') as f:
    pickle.dump(memory, f, 2)
```

3.3.3 Zachowanie

Eksperymenty były prowadzone na scenariuszach Health gathering supreme i Defend the center. W obu przypadkach Kopiowanie zachowań nauczone na już na podstawie 3 trajektorii Eksperta (6 tysięcy klatek) osiągało wizualnie sensowne zachowanie agentów i zaskakująco dobre wyniki.

Dla Defend the center agentowi zdarzało się strzelać w nieodpowiednim momencie lub niestrzelać w odpowiednim. Częste było też nieoptymalne zachowanie w postaci strzelania do odległych przeciwników, podczas gdy inni przeciwnicy mogli podkraść się za plecy agenta zabijając go i kończąc grę.

W tym scenariuszu zwiększanie liczby trajektorii Eksperta użytych do uczenia zwiększało wyniki agenta, który w dużej częsci gier osiągał wyniki bliskie maksymalnym i tylko sporadycznie dawał się na początku gry zajść od tyłu, co skutkowało pojedynczymi niskimi wynikami.

Dla Health gathering supreme agentowi często zdarzało się blokować w rogach labiryntu, wpadając w nieskończoną pętlę akcji. Problem i rozwiązanie zostało opisane w 3.6. Po wyelimininowaniu problemu agent zachowywał się wizualnie sensownie i osiągał przyzwoite wyniki. Problemem jest tylko nauczenie agenta omijania min.

Na początku pracy ze scenariuszem ludzki Ekspert uznał miny za mniejsze apteczki i nie zauważył spadku życia po wejściu w nie. Wyniki uzyskiwane przez Eksperta wchodzącego czasami w miny były tylko nieznacznie lepsze od wyników agenta nauczonego na podstawie tych trajektorii.

W tym scenariuszu zwiększanie liczby trajektorii Eksperta użytych do uczenia nie polepszało wyników agenta.

3.3.4 Wnioski

W badanych scenariuszach Kopiowanie zachowań osiąga znacznie lepsze wyniki, niż sugerowałaby literatura i uzyskuje je w ciągu ułamka czasu potrzebnego klasycznym metodom uczenia ze wzmocnieniem. Uzyskani agenci w większości przypadków zachowują się sensownie, chociaż czasem popełniają systematyczne błędy. Kopiowanie zachowań wydaje się świetnym punktem startowym dla VizDooma i wydaje się wskazane, żeby inne metody rozszerzały to podejście, zamiast je zastępować.

3.4 Q-learning z Ekspertem

Q-learning jest potężną metodą, która jest zdolna osiągać wyniki znacznie prześcigające ludzkie. Jej wadą jest natomiast znaczna ilość czasu i klatek działania, których potrzeba zanim agent zacznie zachowywać się sensownie. Uczenie z Ekspertem, z drugiej strony, może na podstawie trajektorii Eksperta szybko uzyskać przyzwoicie zachowującego się agenta, którego osiągi są niższe lub równe osiągom użytego Eksperta.

Wobec tych dwóch przeciwstawnych podejść dobrym pomysłem wydaje się próba wykorzystania agenta uzyskanego na pomocą uczenia z Ekspertem jako stanu początkowego dla Q-learningu - dzięki temu pomijane jest czasochłonne początkowe rozpoznawanie podstaw mechaniki środowiska, a metoda rozwija tylko za pomocą uczenia ze wzmocnieniem przyzwoitego już wcześniej agenta.

Podstawowym problemem jest niekompatybilność wyjść obu metod - klasyfikator z 3.3 potrafi określić najlepszą akcję dla danego stanu, ale te informacja nie pozwala na pełne odtworzenie wartości Q poszczególnych stanów. Wartości Q dla stanów odwiedzonych przez Eksperta można odtworzyć na podstawie jego trajektorii (albo przybliżyć, jeżeli przejścia i nagrody nie są deterministyczne), ale niemożliwe będzie odtworzenie wartości Q dla akcji niewybranych przez Eksperta - należy za to zagwarantować, że ich Q wartości będą niższe niż Q wartości wybranych akcji.

[UZUPEŁNIĆ]

3.5 DAgger

Podejście Agregacji Zbioru Danych (ang. Dataset Aggregation) [**?**] zostało opisane we wcześniejszym rozdziale. Kluczowym założeniem metody jest odpytywanie Eksperta o właściwe działanie w stanach, które nie były wcześniej przez niego pokazane (i nie należą do „poprawnych" trajektorii), a które zostały odwiedzone przez agenta na skutek jego nieoptymalnego zachowania.

W rzeczywistości, dla bardziej skomplikowanych zadań, odpytywanie Eksperta o decyzję dla każdego odwiedzonego przez agenta stanu jest niepraktyczne. Ocenianie wielu kolejnych stanów może być drogie i nużące dla Eksperta, co może przekładać się na obniżoną jakość decyzji. Ocena dokonywana przez eksperta może też w praktyce różnić się w zależności od tego, czy Ekspert napotkał dany stan podczas normalnego działania, czy podczas oceny pojednyczych, wyrwanych z kontekstu stanów.

Aby zminimalizować ten problem, konieczne jest określenie mniejszego podzbioru stanów, dla których potrzebna jest ocena eksperta.

3.5.1 Implementacja

Zastosowana implementacja jest rozszerzeniem 3.3. Pierwszym krokiem jest załadowane przygotowanych wcześniej trajektorii Eksperta do pamięci agenta (zestawu danych).

Następnie agent rozpoczyna działanie, bazując na swoim aktualnym stanie wiedzy. Po wystąpieniu określonych warunków, definiujących potrzebę odpytania Eksperta, działanie programu zostaje wstrzymane, a sterowanie przekazane jest do Eksperta. Aby dostosować się do ograniczeń ludzkiego Eksperta, po przekazaniu sterowania program przechodzi w tryb synchroniczny - przed każdą kolejną klatką czeka na na reakcję Eksperta. Ubocznym skutkiem tej implementacji jest pomijanie akcji „nic nie rób", która jest wykonywana dopiero po wciśnięciu dedykowanego klawisza.

Po wystąpieniu określonych warunków, definiujących koniec potrzeby odpytywania Eksperta, wszystkie stany i akcje odwiedzone w trakcie danej demonstacji dodawane są do pamięci agenta (trajektoria może być dodana do pamięci z większą wagą niż początkowe prezentacje - w przeciwnym wypadku dodanie nowych danych mogło by nie być odczuwalne). Agent aktualizuje klasyfikator akcji na podstawie noworozszerzonego zestawu danych, po czym przejmuje sterowanie od Eksperta i wraca do normalnego działania bazując na uaktualnionym stanie wiedzy.

Po ponownym wystąpieniu określonych warunków, kontrola może ponownie zostać przekazana do Eksperta.

3.5.2 Przekazywanie sterowania

Jednym z najważniejszych problemów jest zdefiniowanie, kiedy przekazywać sterowanie pomiędzy agentem a Ekspertem. Wybór sposobu będzie decydował o tym, jak często Ekspert będzie odpytywany i na ile istotna będzie uzyskana wiedza. Sprawdzone zostały trzy następujące sposoby.

Losowe przekazanie sterowania

1. Przed wykonaniem każdej akcji agent z bardzo małym prawdopodobieństwem może zdecydować o przekazaniu sterowania Ekspertowi.

2. Po każdej akcji Eksperta program z większym prawdopodbieństwem może zdecydować o przekazaniu sterowania do agenta.

Losowe przekazywanie sterowania jest niepraktyczną metodą - dla analizowanych problemów agent nie potrzebuje pomocy Eksperta przez większość czasu, więc losowo wybrane momenty przekazania sterowania w ogromnej większości nie dostarczają istotnej informacji. Zaletą jest natomiast automatyczność decyzji - program podczas gry agenta może działać w przyspieszonym tempie.

Analiza niepewności sieci

1. Przed wykonaniem każdej akcji sprawdzana jest jej niepewność sieci dla danego stanu [REF]. W przypadku wystąpienia zadanej liczby kolejnych niepewnych akcji sterowanie przekazywane jest do Eksperta.

2. Po każdej akcji Eksperta sprawdzana jest akcja, którą wykonałby agent. Jeżeli przez zadaną liczbę kolejnych kroków agent postąpiłby identycznie jak Ekspert, to sterowanie wraca do agenta.

Analiza niepewności sieci jest skuteczniejsza niż losowe przekazywanie sterowania. Wybrane tym sposobem okna działania Eksperta częściej pokrywają się z oknami niepoprawnego działania agenta. W dalszym ciągu skuteczność metody nie jest zadowalająca - przyjęta miara niepewności powoduje, że agent może przekazać sterowanie do Eskperta w obliczu sytuacji, dla której więcej niż jedna akcja jest sensowna. Porównywanie akcji agenta i Eksperta przez zadaną liczbę kroków jest skuteczne dla problemów z niewielką liczbą akcji, ale nieskuteczne w sytuacji, w której podobny efekt można uzyskać za pomocą różnych sekwencji kroków (przykładowo dojście do danego punktu za pomocą permutacji akcji łewo", "prosto"i łewo i prosto"). Podobnie jak przy losowym podejściu, dzięki automatycznemu działaniu możliwe jest działanie programu w przyspieszonym tempie poczas gry agenta.

Decyzja Eksperta

1. Ekspert obserwuje działanie agenta. Ekspert przejmuje sterowanie kiedy uzna, że agent trafił do niepożądanego stanu.

2. Kiedy Ekspert uzna, że agent nie jest już w niepożądanym stanie może oddać sterowanie agentowi.

Decyzja Eksperta jest najskuteczniejszą metodą i jest używana w dalszych eksperymentach. Ekspert może sam stwierdzić, kiedy działanie agenta jest niezgodne z pożądanym, maksymalizując skuteczność odpytywania Eksperta. Oczywiście, Ekspert musi spędzić więcej czasu obserwując działanie agenta, ale obserwacja jest dużo mniej uciążliwa (a zatem tańsza), niż prezentowanie. Problemem w niektórych sytacjach jest możliwość rozróżnienia, kiedy agent zachowa się niepożądanie i należałoby przejąć sterowanie - w wielu sytuacjach Ekspert reaguje zbyt późno, żeby demonstracja była skuteczna.

3.5.3 Techniczna implementacja

Architektura sieci neuronowej jest identyczna z architekturą zastosowaną w 3.3.

Implementacja przekazywania sterowania do Eksperta w środowisku VizDoom byłaby wymagająca i czasochłonna, dlatego zastosowano znacznie prostsze, chociaż mniej eleganckie rozwiązanie.

Przy odpytywaniu Eksperta o akcje program oczekuje na następny znak, który pojawi się na standardowym strumieniu wejścia programu (następny znak wpisany w konsoli). Wybrane znaki

są przypisane do indeksów wybranych akcji, wpisanie nieznanego znaku powoduje wybranie akcji o indeksie 0, czyli „nic nie rób”.

```
1
2     def get_expert_action(self):
3         fd = sys.stdin.fileno()
4         old_settings = termios.tcgetattr(fd)
5         try:
6             tty.setraw(sys.stdin.fileno())
7             move = sys.stdin.read(1)
8         finally:
9             termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
10        if move == 'j':
11            return 4
12        if move == 'l':
13            return 2
14        if move == 'a':
15            return 1
16
17        if move == 'i':
18            return 1
19        if move == 'u':
20            return 5
21        if move == 'o':
22            return 3
23        return 0
```

Przy metodzie 3.5.2 konieczne jest asynchroniczne przetwarzanie działania Eksperta. Program nie może oczekiwać na działanie Eksperta, ale kiedy Ekspert zarząda przekazania sterowania nastepna akcja powinna być już wykonywana przez niego.

W tym celu wykorzystano bibliotekę PyKeyboardEvent, która umożliwia reagowanie na systemowe informacje o wciśnięciu bądź puszczeniu klawiszy klawiatury. Poniższa klasa wywołuje zadaną funkcję po wciśnięciu lub puszczeniu zadanych klawiszy.

```
1  from __future__ import print_function
2  from pykeyboard import PyKeyboardEventwojciech_kopec_101675.pdf
3
4
5  class KeyMonitor(PyKeyboardEvent):
6      def __init__(self, keys, keypress_handler):
7          PyKeyboardEvent.__init__(self)
8          self.keypress_handler = keypress_handler
9          self.keys = set(keys)
10
11     def tap(self, keycode, character, press):
12         if character in self.keys:
13             self.keypress_handler(character, press)
```

Wywoływana funkcja znajduje się poniżej. Klawisz ’p’ przekazuje sterowanie pomiędzy Ekspertem i agentem. Klawisze ’,’ i ’.’ zwalniają i przyspieszają działanie programu podczas gry agenta.

```
1      def __toggle_user_input(self, character):
2          if character == 'p':
3              if self.expert_mode:
4                  self.learn_all()
5              self.expert_mode = not self.expert_mode
6              print ("Expert toggled: " + str(self.expert_mode))
7          elif character == '.':
8              self.framerate+=5
9              print ("Framerate: " + str(self.framerate))
10         elif character == ',':
11             self.framerate -= 5
12             print ("Framerate: " + str(self.framerate))
13         return True
```

3.5.4 Zachowanie

Eksperymenty były prowadzone na przede wszystkim na scenariuszu Health gathering supreme. Początkowe trajektorie Eksperta były wygenerowane zgodnie z 3.6.

Dla każdego z badanych scenariuszy uwzględnianie fragmentów trajektorii zaprezentowanych przez Eksperta w trakcie gry obniża początkowo wyniki. Na skutek nauczenia się niespójnych zachowań Eksperta agent zachowuje się mniej płynnie i częściej wpada w nieskończone pętle ruchów (przykładowo obracanie się naprzemian w lewo i w prawo w rogu labiryntu), co prowadzi do osiągania niższych wyników.

W scenariuszu Health gathering supreme głównym problemem agenta 3.3 jest nieomijanie min i celem zastosowania DAggera jest wyeliminowanie tego problemu. Za każdym razem, kiedy agent zbliża się do min Ekspert przejmuje kontrolę i omija miny bądź wybiera inną ścieżkę. Pary stan → akcja uzsykane w ten sposób są dodawane do pamięci dziesięciokrotnie.

Podczas pierwszych epizodów nauki wyniki uzyskiwane przez agenta zauważalnie się obniżają, a problem wchodzenia na miny nie jest wyeliminowany.

Następne epizody nauki powoli poprawiają wyniki agenta, przywracając je do poziomu wyjściowego lub nieznacznie go przewyższającego. Agent rzadziej wchodzi w miny, ale problem w dalszym ciągu pozostaje obecny.

Kolejne epizody nauki doprowadzają do przeuczenia - wyniki obniżają się, a agent regularnie wpada w nieskończone pętle ruchów. Wchodzenie w miny nie zostaje wyelimininowane. Pogorszenie zachowania agenta może wynikać ze znużenia Eksperta, a co za tym idzie zmiany jego zachowań i pogorszenia jego decyzji.

3.5.5 Wnioski

Dla wypróbowanych problemów DAgger nie wydaje się być skuteczny. W VizDoomie decyzje podejmowane przez ludzkiego Eksperta są bardziej skomplikowane niż w Mario Cart, przedstawianym w publikacji, co, na skutek niespójności przedstawianych przez Eksperta zachowań, zamiast do podwyższenia wyników agenta prowadzi do obniżania jego skuteczności. Zastosowane głębokie sieci neuronowe mogą też znacznie skuteczniej uogólniać wiedzę zdobytą podczas pierwszej prezentacji Eksperta niż prostsze klasyfikatory SVM, a co za tym idzie nawet bez użycia DAggera agent potrafi znaleźć sensowne wyjście z większości sytuacji. Uzyskiwanie oceny Eksperta jest uciążliwe i kosztowne.

3.6 Świadomie prezentujący Ekspert

W sekcji 3.3 opisano agenta budującego klasyfikator (stan → akcja) na podstawie trajektorii zebranych podczas gry Eksperta. Uzyskany agent zachowywał się sensownie, ale problem stanowiło między innymi blokowanie się w rogach labiryntu i wchodzenie na miny. Główną praktyczną wadą metody 3.5, która miała na celu zaradzenie temu, jest niespójność zachowań Eksperta podczas pierwszej (ciągłej) prezentacji i zachowań podczas krótkich prezentacji podczas gry agenta oraz uciążliwość obserwacji i przejmowania sterowania od agenta w trakcie gry.

Problem wchodzenia w ściany, dla przykładu, jest łatwo zauważalny podczas obserwacji działania agenta. Oczywistym jest też powód jego występowania - Ekspert, w przeciwnieństwie do agenta, pamięta jak dotarł do danego stanu i znajdując się w rogu pamięta, w którą stronę powinien z niego wychodzić. Badani agenci mogą pamiętać tylko kilka ostatnich odwiedzonych klatek i nie pamiętają swoich trajektorii. Dlatego klasyfikator nauczony na trajektoriach Eksperta nie ma

wystarczających informacji żeby rozróżnić konieczność wychodzenia z rogu obracając się w prawo bądź w lewo.

Rozwiązaniem jest powtórne zebranie trajektorii Eksperta, kładąc przy prezentacji nacisk na zachowywanie się w sposób spójny i ułatwiający klasyfikatorowi skuteczną naukę. Możliwe jest też pokazywanie rozwiązań sytuacji, które wcześniej sprawiały klasyfikatorowi problem, w celu pokazania poprawnego zachowania w danej sytuacji.

Oczywiście, takie zachowanie Eksperta skutkuje uzyskiwaniem przez niego nieoptymalnych wyników, a co za tym idzie wyniki możliwe do osiągnięcia przez idealnie odwzorowującego agenta też są niższe. W praktyce różnica pomiędzy wynikami Eksperta i agenta powinna się zmniejszyć dzięki świadomej prezentacji, skutkując wyższymi wynikami osiąganymi przez agenta.

3.6.1 Techniczna implementacja

Architektura sieci neuronowej jest identyczna z architekturą zastosowaną w 3.3.

3.6.2 Zachowanie

Eksperymenty były prowadzone na scenariuszach Health gathering supreme i Defend the center.

W scenariuszu Defend the center Ekspert podczas świadomej prezentacji powstrzymywał się od strzelania do odległych przeciwników i świadomie preferował strzelanie do szybszych przeciwników. Świadoma prezentacja zmniejszyłą liczbę niepotrzebnych strzałów nauczonego agenta.

W scenariuszu Health gathering supreme Ekspert podczas świadomej prezentacji zawsze wychodził z rogów obracając się w tę samą stronę i w miarę możliwości odwracał się od tras z minami. Będąc otoczony przez miny wybierał trasę jak najbardziej odległą od nich. Świadoma prezentacja prawie całkowicie wyeliminowała wpadanie w nieskończone pętle ruchów w rogach. W niektórych sytuacjach zdarzało się, że agent zawracał za to w ciasnych, ale możliwych do przejścia korytarzach - było to zachowanie wyraźnie nieoptymalne, ale bez zauważalnego wpływu na osiągane wyniki. Niestety, świadoma prezentacja nie wyeliminowała wchodzenia w miny. Wynik punktowy agenta zwiększył się istotnie po zastosowaniu świadomej prezentacji.

To, jak ważna jest świadoma prezentacja widoczne było przy zwiększaniu wielkości trajektorii Eksperta użytych do nauki klasyfikatora. Dla Defend the center, który jest prostszym scenariuszem i dla którego zysk ze świadomej prezentacji był mniej zauważalny, zwiększanie liczby trajektorii uczących prowadziło do wyższych wyników. Dla bardziej skomplikowanego Health gathering supreme agent nauczony na podstawie małej liczby trajektorii świadomego Eksperta przewyższał agenta nauczonego na większej liczbie trajektorii nieświadomego Eksperta i agenta nauczonego na mieszance trajektorii.

3.6.3 Wnioski

Dla bardziej skomplikowanych scenariuszy świadoma prezentacja Eksperta jest prostym i bardzo skutecznym sposobem eliminowania części oczywistych błędów popełnianych przez agenta. Dla niektórych problemów i sytuacji może wypełniać zadanie postawione przed 3.5 w wygodniejszy i bardziej naturalny sposób. Świadoma prezentacja nie jest formalną metodą, a raczej wytyczną. Dzięki temu można ją z powodzeniem stosować w połączeniu z innymi technikami uczenia z Ekspertem.

3.7 Wnioski z porównania metod

Rozdział 4

Niepewność

4.1 Niepewność - uproszczony eksperyment

W celu porównania skuteczności Bootstrapa i Dropoutu w ocenianiu niepewności wyników sieci przeprowadzono prostszy eksperyment na lepiej znanych i kontrolowanych danych. Do eksperymentu wykorzystano zbiór MNIST [REF] zawierający odręcznie pisane cyfry. MNIST jest często używany jako przykładowy zbiór danych, służący do przystępnej prezentacji i porównań metod uczenia maszynowego. Najczęściej wykorzystywany jest w kontekscie klasyfikacji, jednak traktowanie wartości liczbowych cyfr jako etykiet (w przeciwieństwie do stosowanego w klasyfikacji one-hot encoding [REF]) w oczywisty sposób odpowiada problemowi regresji.

4.1.1 Eksperyment

W ramach eksperymentów uczono sieć neuronową na niezbalansowanych zbiorach danych bazujących na MNIST. Dobrane dystrybucje przykładów o konkretnych etykietach w danych uczących mają pokazać, jak poszczególne techniki zachowują się wobec zupełnie nieznanych danych i mało znanych danych.

Zbiór danych MNIST składa się z 60 tysięcy przykładów uczących i 10 tysięcy przykładów treningowych. Każdy obrazek przedstawia jedną czarno-białą cyfrę i ma rozmiar 28x28 pikseli. W eksperymencie etykiety zostały przetrasformowane liniowo z przedziału $[0, 9]$ do $[-0.5, 0.5]$.

Pierwszy, prostszy, eksperyment posłużył również do znalezienia optymalnych parametrów obu metod. Drugi eksperyment został przeprowadzony z wykorzystaniem znalezionych wcześniej parametrów. Każdy z eksperymentów został powtórzony 10 razy.

Rozpoznawanie nieznanych danych

Zachowanie wobec nieznanych danych zbadano ucząc sięć neuronową wyłącznie na przykładach parzystych cyfr. W wynikowej klasyfikacji stopień niepewności zwracanych wyników powinien być znacznie wyższy dla cyfr nieparzystych niż parzystych.

Ocena stopnia pewności wyników

Zachowanie wobec mało znanych danych zbadano ucząc sieć neuronową na niezbalansowanym zbiorze danych. Przykłady trafiały do zbioru uczącego z prawdopodobieństwem proporcjonalnym do przedstawianej cyfry, przykładowo 0 z prawdopodobieństwem $p = 0$, 5 z prawdopodobieństwem $p = 0.5$ i 9 z prawdopodobieństwem $p = 0.9$. W wynikowej klasyfikacji stopień niepewności zwracanych wyników powinien być zależny od prawdopodobieństwa trafienia do zbioru danej etykiety.

4.1.2 Implementacja bazowa

Bazą dla implementacji użytych w eksperymencie był przykład klasyfikacji zbioru MNIST za pomocą konwolucyjnych sieci neuronowych udostępniony z biblioteką Tensorflow. Użyta w przykładzie sieć składa się z dwóch warstw konwolucyjnych i dwóch warstw w pełni połączonych.

W ramach obu eksperymentów uczenie trwa 1 milion iteracji, a wielkość batcha wynosi 128. Prędkość uczenia wynosi 0.005, przy czym dla Bootstrapu ta wartość jest normalizowana przez średnią liczbę użyć każdego przykładu.

4.1.3 Miary jakości

Kluczowe dla określenia jakości obu metod jest zdefiniowanie miary jakości wyników. Docelowo przyjęta miara powinna dobrze oddawać przydatność do oceny stanu przez agenta DQN.

Za miarę niepewności przyjęto rozstęp międzykwartylowy próbek uzyskanych z sieci. O wyborze rozstępu międzykwartylowego zdecydowała większa od odchylenia standardowego odporność na skrajne wartości. Oprócz rozstępu międzykwartylowego sprawdzono eksperymentalnie również wariancję (która dawała nieznacznie gorsze wyniki) i różnicę między skrajnymi wartościami (zależność od skrajnych wartości uczyniła tą miarę bardzo niestabilną i mało skuteczną).

$$unc = |q(75) - q(25)|$$

Rozpoznawanie nieznanych danych

Liczba znanych i nieznanych etykiet w zbiorze tekstowym jest równa, dlatego za miarę jakości rozdziału danych znanych i nieznanych przyjęto stosunek sumy średnich niepewności dla kolejnych nieznanych etykiet do sumy niepewności dla kolejnych znanych etykiet. Wartości bliskie 1 oznaczają brak rozdziału danych. W eskperymentach wynikowe miary cząstkowe nie przekraczały wartości 2.

$$quality_{ND} = \frac{\sum_{l \in \{unknown\}} \overline{unc_l}}{\sum_{l \in \{known\}} \overline{unc_l}}$$

Ocena stopnia pewności wyników

Niepewność powinna być odwrotnie proporcjonalna do trafności klasyfikacji, dlatego za miarę jakości oceny niepewności wyników przyjęto wartość absolutną współczynnika korelacji pomiędzy średnią niepewnością a średnią trafnością klasyfikacji dla kolejnych etykiet. Użyty w eksperymencie współczynnik regresji jest liczony dla etykiet od 1 do 9, ponieważ zerowa trafność dla etykiety 0 jest wspólna dla obu metod i zaburza wyraźnie liniową zależność dla reszty etykiet.

$$quality_{OP} = |r_{unc\ acc}|$$

4.1.4 Dropout - konfiguracja

Dropout został dodany pomiędzy ostatnią warstwą konwolucyjną sieci a pierwszą w pełni połączoną oraz pomiędzy obiema w pełni połączonymi warstwami. Parametry modelu to prawdopodobieństwo zachowania neuronu w czasie treningu p_{train}, prawdopodobieństwo zachowania neuronu w czasie testu p_{test} i liczbę odpytań sieci n przy określaniu niepewności. W eksperymentach sprawdzano wartości $p_{train} \in \{0.25, 0.5, 0.75, 1\}$, $p_{test} \in \{0.25, 0.5, 0.75\}$ i $n \in \{10, 30, 50, 100\}$.

Zastosowana implementacja z wykorzystaniem Tensorflow wymaga każdorazowego przeliczenia wszystkich wartości przy każdym pojedyńczym odpytaniu.

4.1.5 Bootstrap - konfiguracja

Bootstrapowana sieć ma wspólne warstwy konwolucyjne. Z warstw konwolucyjnych wychodzi n niezależnych "głów", składających się z dwóch warstwy w pełni połączonych wykorzystanych w przykładzie bazowym. Parametry modelu to liczba "głów" n i prawdopodobieństwo uwzględnienia krotki danych przez głowę p_{incl}. W eksperymentach sprawdzano wartości $n \in \{5, 7, 10\}$ i $p_{incl} \in \{0.25, 0.5, 0.75, 1\}$.

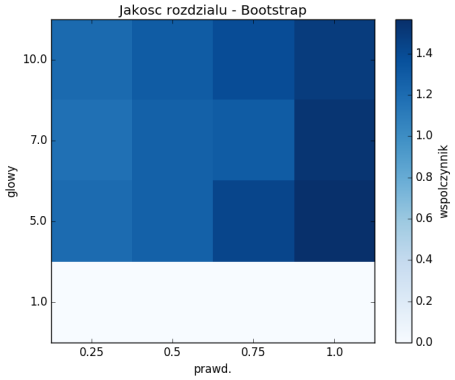
Zastosowana implementacja z wykorzystaniem Tensorflow decyduje o uwzględnianiu przez poszczególne głowy dla pełnych batchy danych, a nie dla pojedynczych krotek. Przy odpytywaniu kilku głów o ten sam przykład przeliczenie warstw konwolucyjnych następuje tylko jednokrotnie.

4.1.6 Niepewność - wyniki eksperymentu z nieznanymi danymi

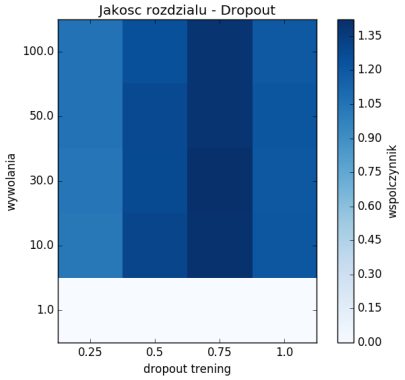
Na wykresach 4.1, 4.2, 4.3, 4.4, 4.5, 4.6 przedstawiono średnie i odchylenia standardowe miary jakości $quality_{ND}$ uzyskane dla poszczególnych konfiguracji eksperymentów. Największa wartość $quality_{ND}$ uzyskana za pomocą Bootstrapa (1.565 dla 5 głów i prawdopodobieństwa 1) jest wyższa niż największa wartość uzyskana za pomocą Dropoutu (1.424 dla 30 wywołań i prawdopodobieństw dropoutu = 0.75). Wyniki bootstrapa dla tych parametrów cechują się trzykrotnie większą wariancją (0.148 a 0.049), ale mimo to sumarycznie wypadają korzystniej od Dropoutu.

Boostrap najlepiej wypada dla małej (5) liczby głów - jest to zaskakujące zachowanie, które może być artefaktem zbyt małej liczby powtórzeń eksperymentu. Podobne wyniki dla różnej liczby głów utrzymują jednak stałą przewagę nad Dropoutem. Zgodna z oczekiwaniami jest przewaga konfiguracji z prawdopodobieństwem uwzględnienia przez głowę próbki równym 1. Dzięki temu poszczególne głowy są lepiej dopasowane do znanych przykładów powiększając różnicę w stosunku do nieznanych danych.

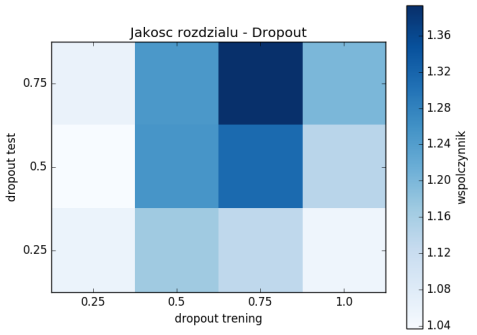
Dla Dropoutu zachodzi podobne zjawisko dla liczby wywołań jak dla głów w Bootstrapie - wbrew oczekiwaniom najlepsze wyniki osiągane są dla mniejszej liczby powtórzeń, przy zachowaniu niewielkich różnic między wartościami. Podobnie zgodnie z oczekiwaniami zachowuje się też drugi z parametrów - prawdopodobieństwo zachowania neuronu w czasie treningu. Wysokie prawdopodobieństwo pozwala "poznać" lepiej dane, a prawdopodobieństwo równe 1 uniemożliwia poprawne działanie dropoutu, ponieważ sieć nie jest przyzwyczajona do dropoutu pojawiającego się dopiero w teście.



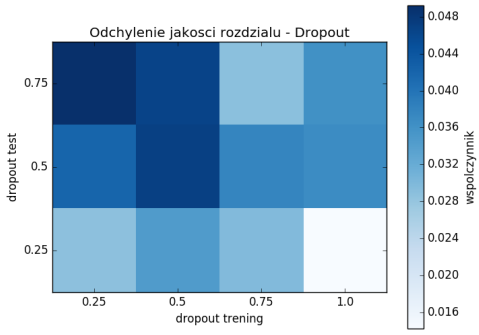
Rysunek 4.1: Bootstrap



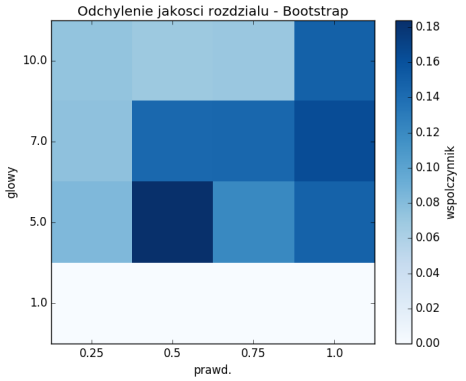
Rysunek 4.2: Dropout



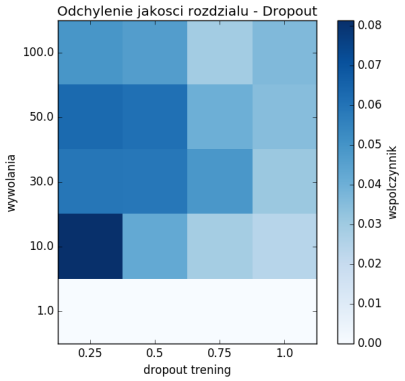
Rysunek 4.3: Dropout



Rysunek 4.4: Dropout

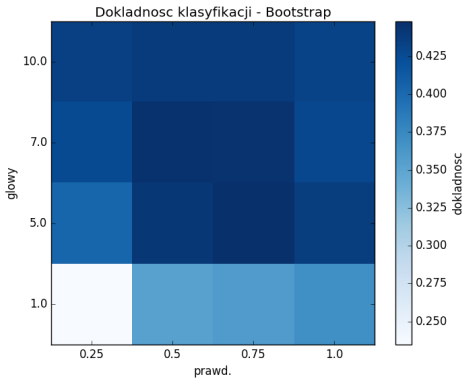


Rysunek 4.5: Bootstrap

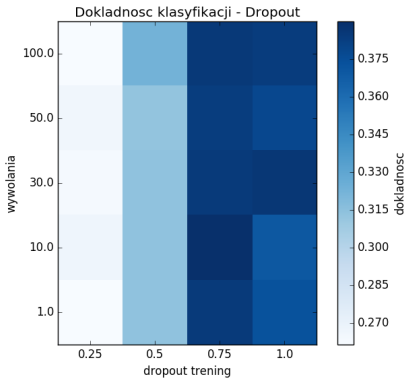


Rysunek 4.6: Dropout

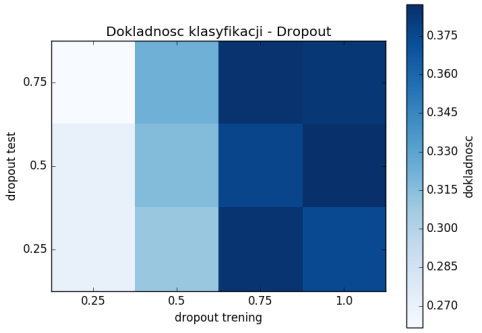
Na wykresach 4.7, 4.8, 4.9, 4.10, 4.12, 4.11 przedstawiono średnie i odchylenia standardowe dokładności klasyfikacji uzyskane dla poszczególnych konfiguracji eksperymentów. Wyniki przedstawiają się podobnie jak dla miary $quality_{ND}$. Lepsze wyniki osiąga Bootstrap (44.81% dla 5 głów i prawdopodobieństwa 0.75, 43.64% dla 5 głów i prawdopodobieństwa 1), a jego wyniki są podobne dla wszystkich sensownych parametrów. Wariancja jest minimalna. Wyniki Dropoutu oscylują dookoła 39% dla wszystkich sensownych parametrów, przy znacznie większej niż Boostrstrap wariancji (2%). Warto zauważyć, że dla obu metod wyniki są bardzo dodobne dla szerokich zestawów parametrów, i wyraźnie większe niż w przypadku braku bazowego rozwiązania (zaimplementowane w eksperymencie jako Bootstrap z jedną głową, osiągające 37% dokładności przy 5% wariancji. Gorszy wynik bazowej wersji wynika z braku odporności na przeuczenie - Bootstrap i Dropout działają jak regularyzatory).



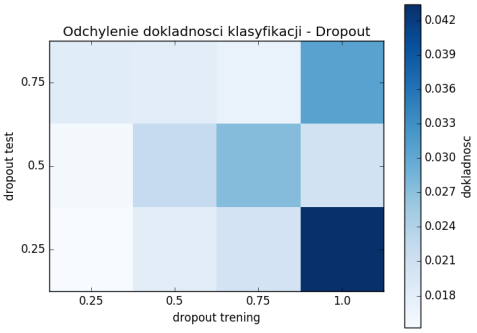
Rysunek 4.7: Bootstrap



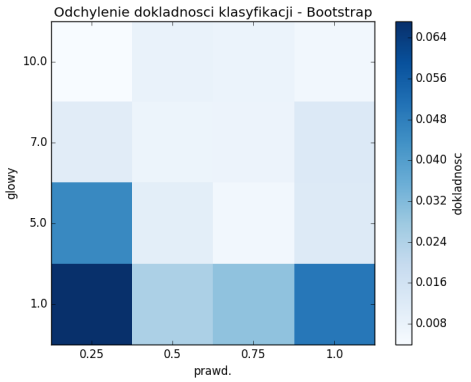
Rysunek 4.8: Dropout



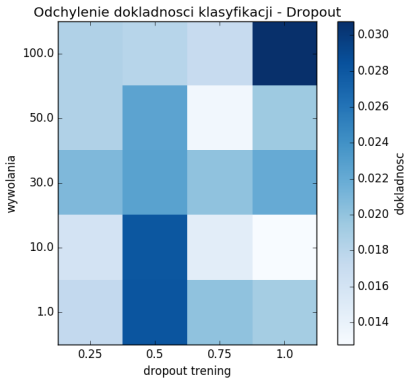
Rysunek 4.9: Dropout



Rysunek 4.10: Dropout



Rysunek 4.11: Bootstrap



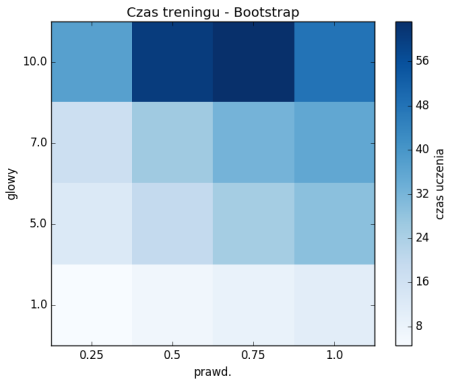
Rysunek 4.12: Dropout

Na wykresach 4.13, 4.14, 4.15, 4.16 przedstawiono czasy treningu i testowania. Na etapie czasu testowania Boostrap wypada znacznie gorzej niż Dropout. 25 sekund dla 5 głów i prawd=0.75 trwa dwa razy dłużej niż 12 sekund osiąganych przez Dropout na wszystkich parametrach, co jest znacznie większym narzutem niż 20% deklarowane przez autorów metody.

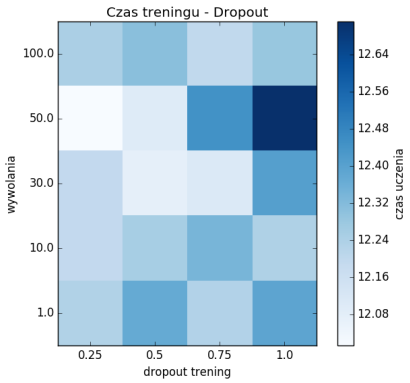
Czas testowania ponownie korzystniejszy jest dla Bootstrapa (0.157s), ponad 5 razy mniej niż Dropout (0.84s). Czas testowania jest bardzo istotny w kontekscie Q-learningu, gdzie dla każdej klatki konieczna jest ocena jakości każdego z możliwych ruchów.

Czas treningu Bootstrapa jest w przybliżeniu liniowo zależny od liczby głów pomnożonych przez prawdopodobieństwo uwzględnienia krotki: $t_{train} \sim n * p_{incl}$, natomiast czas testu jest w przybliżeniu stały. Czas treningu Dropoutu jest w przybliżeniu stały, natomiast czas testu jest w przybliżeniu liniowo zależny od liczby wywołań $t_{test} \sim n$.

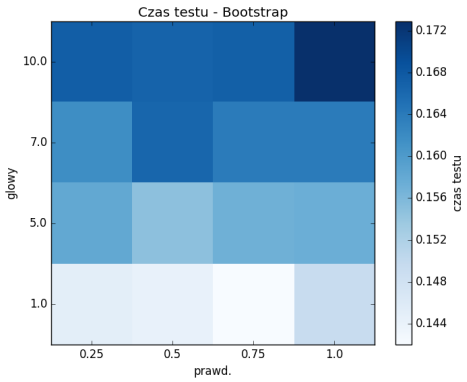
Powody znacznych różnic czasowych mogą tkwić w szczegółach implementacji obu metod. Warto zwrócić uwagę, że dla niektórych zastosowań koszt związany z dodatkowymi obliczeniami może być nieakceptowalny. Natomiast dla zastosowań, dla których koszt działania agenta (czyli koszt zbierania próbek danych) przewyższa koszt działania GPU dłuższy czas działania Bootstrapa może być nieistotny, albo zniwelowany przy wykorzystaniu mocniejszych lub większych zasobów.



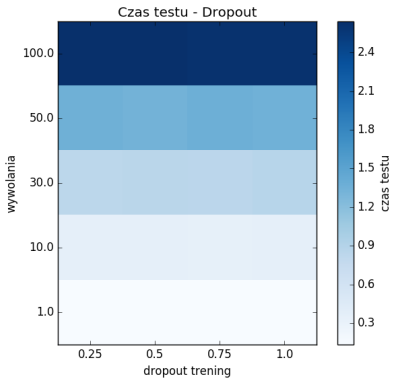
Rysunek 4.13: Dropout



Rysunek 4.14: Dropout



Rysunek 4.15: Dropout



Rysunek 4.16: Dropout

Na podstawie przeprowadzonego eksperymentu za najlepszą konfigurację Bootstrapa przyjęto 5 głów i prawdopodobieństwo 0.75, a dla Dropoutu 30 wywołań i oba prawdopodobieństwa równe 0.75

4.1.7 Niepewność - wyniki eksperymentu z oceną stopnia pewności

Dla liczby głów $n = 5$ w Bootstrapie i dla liczby wywołań $n = 30$ i prawdopodobieństwach dropoutu $p = 0.75$ metod współczynnik $quality_{OP}$ dla Bootstrapa z $p = 0.75$ wynosi 0.833 przy wariancji 0.11, dla Bootstrapa z $p = 1$ wynosi 0.823 przy wariancji 0.10, a dla Dropoutu wynosi 0.925 przy wariancji 0.04. Trafności klasyfikacji dla Bootstrapa z $p = 0.75$ wynosi 0.638 a przy wariancji 0.038, dla Bootstrapa z $p = 1$ wynosi 0.623 a przy wariancji 0.036 dla Dropoutu wynosi 0.527 przy wariancji 0.032.

4.1.8 Niepewność - wnioski

Bootstrap ma wyraźną przewagę na większości czynników. W drugim eksperymencie jego wyniki są nieznacznie gorsze, ale jako że obie metody osiągają bardzo wysoki współczynnik, ten współczynnik jest pomijalny. Niestety, długi czas treningu Bootstrapa sprawia, że Dropout nie może być kategorycznie odrzucony. Niewykluczone, że w docelowym rozwiązaniu gorsze wyniki Dropoutu będą niezauważalne, natomiast zwiększony czas obliczeń będzie niakceptowalny.

Najważniejszym wnioskiem jest natomiast obserwacja, że obie metody bardzo skutecznie estymują stopień niepewności.

Rozdział 5

Eksperymenty

Rozdział 6

Wnioski i perspektywy rozwoju

Instytut Informatyki, Wydział Informatyki
Politechnika Poznańska

Skład przy użyciu systemu LaTeX.

BibTeX:

```
@mastersthesis{ key,
    author = "Wojciech Kopeć ",
    title = "{Uczenie przez demonstrację na podstawie informacji obrazowej w środowisku 3D. }",
    school = "Poznan University of Technology",
    address = "Pozna{\'n}, Poland",
    year = "2017",
}
```