

Politechnika Poznańska  
Wydział Informatyki i Zarządzania  
Instytut Informatyki

Praca dyplomowa magisterska

## **IMITATION LEARNING**

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

<b>1</b>	<b>Wstęp teoretyczny</b>	<b>1</b>
1.1	Wstęp . . . . .	1
1.2	Uczenie ze wzmocnieniem . . . . .	1
1.3	Aproksymatory funkcji Q . . . . .	1
1.4	Głębokie sieci Q . . . . .	1
1.5	Uczenie z ekspertem . . . . .	1
1.6	VizDoom . . . . .	1
<b>2</b>	<b>Wstęp teoretyczny</b>	<b>2</b>
2.1	Uczenie przez demonstrację a uczenie nadzorowane . . . . .	2
2.2	Podążanie za ekspertem a przewyższanie eksperta . . . . .	3
2.3	Eksploracja . . . . .	3
<b>3</b>	<b>Niepewność</b>	<b>5</b>
3.1	Niepewność - uproszczony eksperyment . . . . .	5
3.1.1	Eksperyment . . . . .	5
	Rozpoznawanie nieznanych danych . . . . .	5
	Ocena stopnia pewności wyników . . . . .	5
3.1.2	Implementacja bazowa . . . . .	6
3.1.3	Miary jakości . . . . .	6
	Rozpoznawanie nieznanych danych . . . . .	6
	Ocena stopnia pewności wyników . . . . .	6
3.1.4	Dropout - konfiguracja . . . . .	6
3.1.5	Bootstrap - konfiguracja . . . . .	7
3.1.6	Niepewność - wyniki eksperymentu z nieznanymi danymi . . . . .	7
3.1.7	Niepewność - wyniki eksperymentu z oceną stopnia pewności . . . . .	10
3.1.8	Niepewność - wnioski . . . . .	11
<b>4</b>	<b>Zaimplementowane podejścia</b>	<b>12</b>
4.1	Scenariusze . . . . .	12
	Basic . . . . .	12
	Defend the center . . . . .	13
	Health gathering supreme . . . . .	13
4.2	Q-learning . . . . .	14
4.3	Kopiowanie zachowań . . . . .	14
4.3.1	Implementacja . . . . .	14
4.3.2	Implementacja techniczna . . . . .	15
4.3.3	Zachowanie . . . . .	16

4.3.4	Wnioski . . . . .	16
4.4	Q-learning z Ekspertem . . . . .	16
4.5	Dagger . . . . .	16
4.5.1	Implementacja . . . . .	16
4.5.2	Przekazywanie sterowania . . . . .	17
	Losowe przekazanie sterowania . . . . .	17
	Analiza niepewności sieci . . . . .	17
	Decyzja Eksperta . . . . .	18
4.5.3	Techniczna implementacja . . . . .	18
4.5.4	Zachowanie . . . . .	19
4.5.5	Wnioski . . . . .	20
4.6	Świadomie prezentujący Ekspert . . . . .	20
4.6.1	Techniczna implementacja . . . . .	20
4.6.2	Zachowanie . . . . .	20
4.6.3	Wnioski . . . . .	21
4.7	Wnioski . . . . .	21

# Rozdział 1

## Wstęp teoretyczny

1.1 Wstęp

1.2 Uczenie ze wzmocnieniem

1.3 Aproksymatory funkcji  $Q$

1.4 Głębokie sieci  $Q$

1.5 Uczenie z ekspertem

1.6 VizDoom

## Rozdział 2

# Wstęp teoretyczny

### 2.1 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 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 zadaniem 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 kolejnej 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.

## 2.2 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.

## 2.3 Eksploracja

Podstawowym i często używanym podejściem do eksploracji jest wspomniany wcześniej e-zachłanny algorytm, w którym agent z zadaniem 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.

Prostym, ale skutecznym i posiadającym teoretyczne gwarancje zbieżności algorytmem jest zaproponowany w [?] R-max, realizujący ideę optyimizmu 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 spodzie-

wana 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.

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 nieznan 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 stanami używane były do wytrenowania właściwego, prostszego aproksymatora, na podstawie błędów którego określano nowość stanu. Dla każdego przejścia między stanami przyznawano bonusową nagrodę zależną od nowości. 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.

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 bootstrappingu. 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.



## Rozdział 3

# Niepewność

### 3.1 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 kontekście klasyfikacji, jednak traktowanie wartości liczbowych cyfr jako etykiet (w przeciwieństwie do stosowanego w klasyfikacji one-hot encoding [REF]) w oczywisty sposób odpowiada problemowi regresji.

#### 3.1.1 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 nieznanymi 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 przetransformowane 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 nieznanymi danych

Zachowanie wobec nieznanymi danych zbadano ucząc sieć 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.

### 3.1.2 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.

### 3.1.3 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 tę 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 eksperymentach 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.

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

### 3.1.4 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 pojedynczym odpytaniu.

### 3.1.5 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  $p_{incl} \in \{0.25, 0.5, 0.75, 1\}$ .

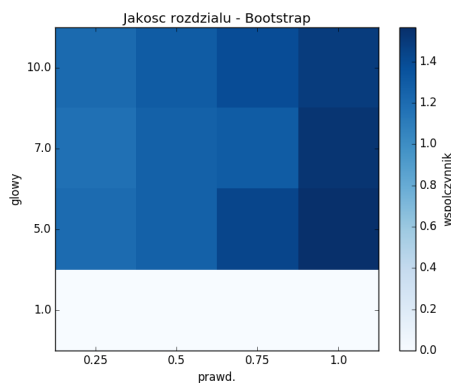
Zastosowana implementacja z wykorzystaniem Tensorflow decyduje o uwzględnianiu przez poszczególne głowy dla pełnych batchy danych, a nie dla pojedynczych krotek. Przy odpytywaniu kilku głów o ten sam przykład przeliczenie warstw konwolucyjnych następuje tylko jednokrotnie.

### 3.1.6 Niepewność - wyniki eksperymentu z nieznanymi danymi

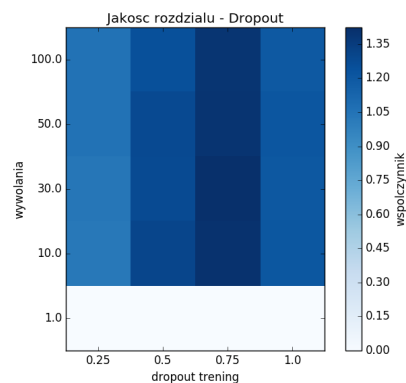
Na wykresach 3.1, 3.2, 3.3, 3.4, 3.5, 3.6 przedstawiono średnie i odchylenia standardowe miary jakości  $quality_{ND}$  uzyskane dla poszczególnych konfiguracji eksperymentów. Największa wartość  $quality_{ND}$  uzyskana za pomocą Bootstrapa (1.565 dla 5 głów i prawdopodobieństwa 1) jest wyższa niż największa wartość uzyskana za pomocą Dropoutu (1.424 dla 30 wywołań i prawdopodobieństwa 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.

Bootstrap 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 nieznanymi danych.

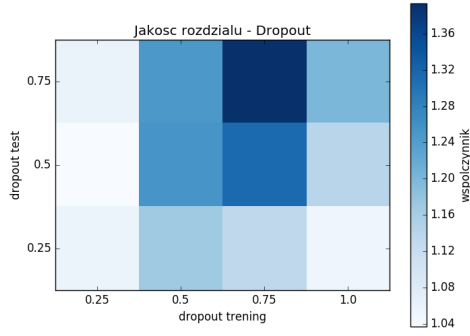
Dla Dropoutu zachodzi podobne zjawisko dla liczby wywołań jak dla głów w Bootstrapie - wbrew oczekiwaniom najlepsze wyniki osiągane są dla mniejszej liczby powtórzeń, przy zachowaniu niewielkich różnic między wartościami. Podobnie zgodnie z oczekiwaniami zachowuje się też drugi z parametrów - prawdopodobieństwo 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.



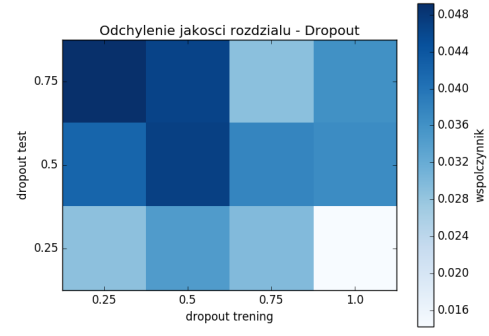
RYSUNEK 3.1: Bootstrap



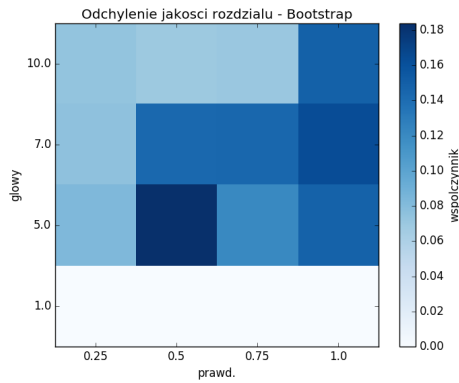
RYSUNEK 3.2: Dropout



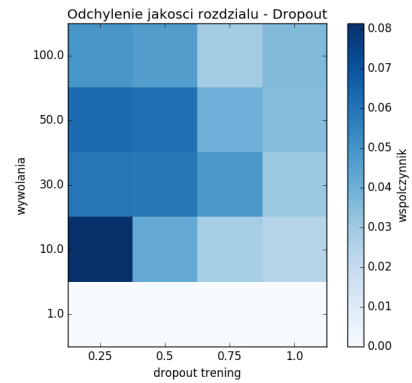
RYSUNEK 3.3: Dropout



RYSUNEK 3.4: Dropout

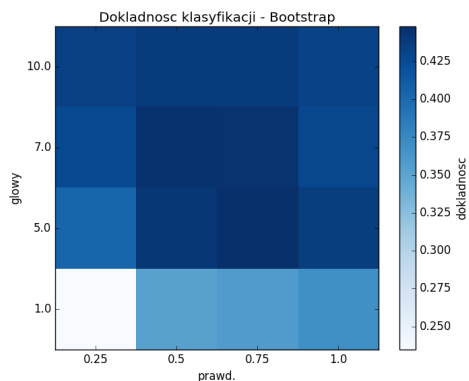


RYSUNEK 3.5: Bootstrap

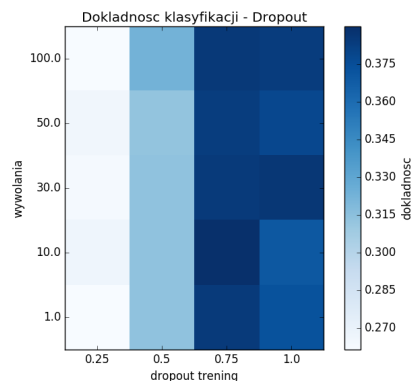


RYSUNEK 3.6: Dropout

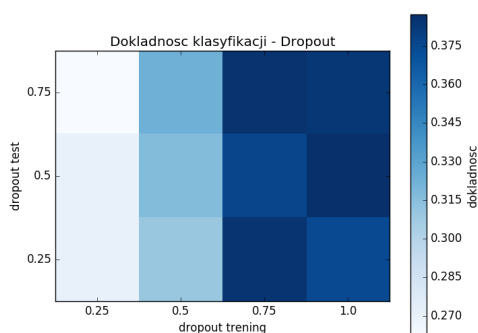
Na wykresach 3.7, 3.8, 3.9, 3.10, 3.12, 3.11 przedstawiono średnie i odchylenia standardowe dokładności klasyfikacji uzyskane dla poszczególnych konfiguracji eksperymentów. Wyniki przedstawiają się podobnie jak dla miary  $quality_{ND}$ . Lepsze wyniki osiąga Bootstrap (44.81% dla 5 głów i prawdopodobieństwa 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 Bootstrap wariancji (2%). Warto zauważyć, że dla obu metod wyniki są bardzo podobne 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 regularizatory).



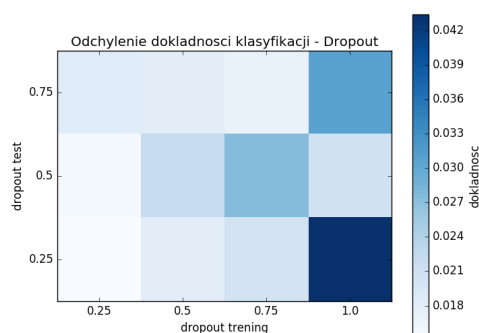
RYSUNEK 3.7: Bootstrap



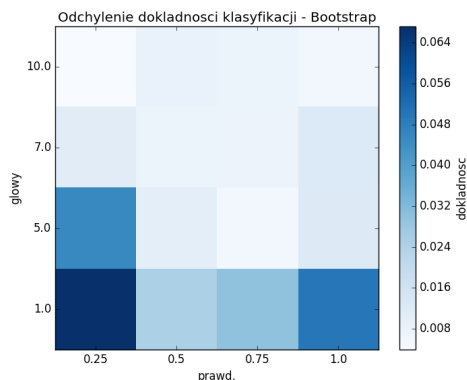
RYSUNEK 3.8: Dropout



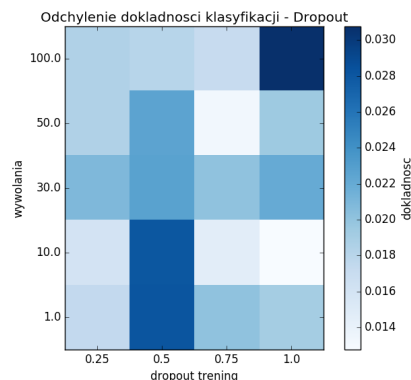
RYSUNEK 3.9: Dropout



RYSUNEK 3.10: Dropout



RYSUNEK 3.11: Bootstrap



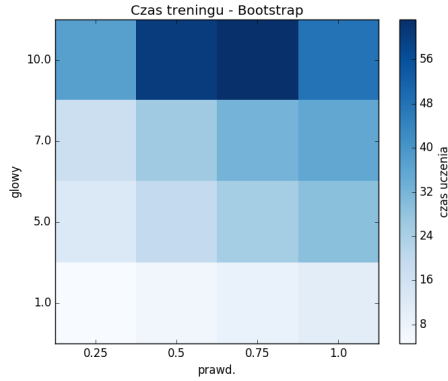
RYSUNEK 3.12: Dropout

Na wykresach 3.13, 3.14, 3.15, 3.16 przedstawiono czasy treningu i testowania. Na etapie czasu testowania Bootstrap wypada znacznie gorzej niż Dropout. 25 sekund dla 5 głów i  $\text{prawd}=0.75$  trwa dwa razy dłużej niż 12 sekund osiągniętych przez Dropout na wszystkich parametrach, co jest znacznie większym narzutem niż 20% deklarowane przez autorów metody.

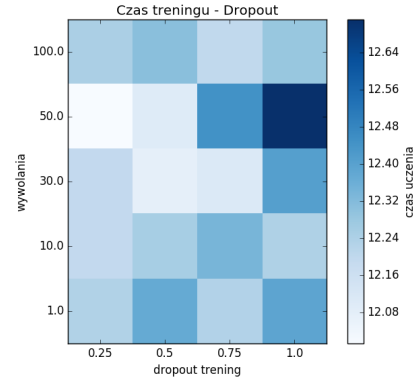
Czas testowania ponownie korzystniejszy jest dla Bootstrapa (0.157s), ponad 5 razy mniej niż Dropout (0.84s). Czas testowania jest bardzo istotny w kontekście Q-learningu, gdzie dla każdej klatki konieczna jest ocena jakości każdego z możliwych ruchów.

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  $t_{test} \sim n$ .

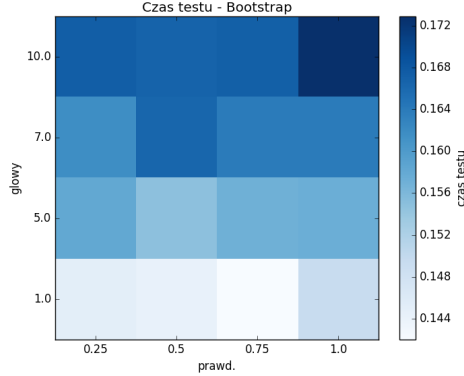
Powody znacznych różnic czasowych mogą tkwić w szczegółach implementacji obu metod. Warto zwrócić uwagę, że dla niektórych zastosowań koszt związany z dodatkowymi obliczeniami może być nieakceptowalny. Natomiast dla zastosowań, dla których koszt działania agenta (czyli koszt zbierania próbek danych)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



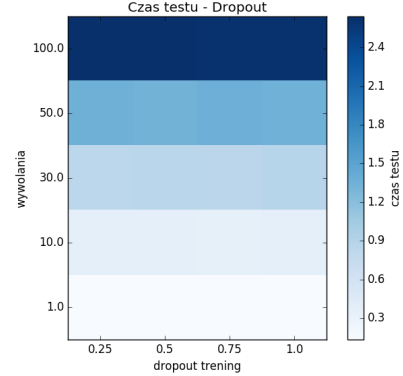
RYSUNEK 3.13: Dropout



RYSUNEK 3.14: Dropout



RYSUNEK 3.15: Dropout



RYSUNEK 3.16: 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

### 3.1.7 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.

### 3.1.8 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 4

# 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. Dzięki temu możliwe jest bezpośrednie porównanie zachowania różnych podejść przy zmianie tylko kluczowych algorytmów.

### 4.1 Scenariusze

Eksperymenty przeprowadzono na następujących scenariuszach.

#### Basic

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

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

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



RYSUNEK 4.1: Scenariusz Basic



### Defend the center

Sceneria składa się z kolistej areny. Agent jest 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 w okół 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 stronę 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 stratego zakłada zajmowanie się najpierw najbliższym zagrożeniem.



RYSUNEK 4.2: Scenariusz Defend the center

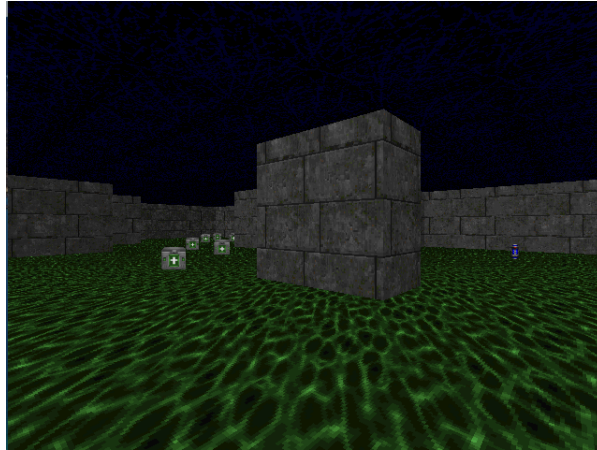
### Health gathering supreme

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

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 żyje, tym więcej punktów.

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.

Warto zauważyć, 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 4.3: Health gathering supreme

## 4.2 Q-learning

### 4.3 Kopiowanie zachowań

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

Dla wielu problemów Kopiowanie zachowań jest nieskuteczne (powody opisane są w [ROZDZIAŁ O TEORII]), ale na badanych scenariuszach VizDoom okazało się działać zaskakująco dobrze.

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 prawie wyłącznie do czasu zbierania trajektorii 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 agent musi przejść przez często przez wiele milionów klatek, by osiągnąć zadawalające wyniki.

#### 4.3.1 Implementacja

Kopiowanie zachowań sprowadza się do klasycznego problemu uczenia nadzorowanego z wieloma etykietami (możliwe akcje), z których tylko jedna jest poprawna. Dane wejściowe stanowią obrazy przedstawiające stan, wynikiem jest pasująca etykieta akcji. Jako klasyfikatora użyto głębokiej sieci neuronowej, o architekturze bazującej na architekturze z 4.2.

Różnica pomiędzy architekturami sporadza się do uczenia i interpretacji wyników. Przy Q-learningu sieć musi przewidywać wartość funkcji  $Q$  dla wszystkich akcji, a przy Kopiowaniu zachowań wystarczy znalezienie 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]

Warto zwrócić uwagę, że 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 niezmiennym zbiorze danych mimo dobrej teoretycznej trafności będzie zupełnie nieskuteczny (np. nie wybierając nigdy akcji „strzelaj”).

Problem ten został rozwiązany przez zbalansowanie zbioru danych przy użyciu metody *oversampling* [REF?] - 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 zdefiniowana była oddzielnie, łączenie z akcjami stanowiącymi złożenie innych 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 było by stworzenie klasyfikatora stan  $\rightarrow$  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 sprawdzone.

### 4.3.2 Implementacja techniczna

Zbieranie danych zostało zrealizowane za pomocą trybu SPECTATOR środowiska VizDoom, pozwalającemu agentowi obserwować 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.

Podczas gry Eksperta ustawiona jest rozdzielczość 640x480 pikseli, i takiej wielkości obrazy zapisywane są do pliku z trajektorią. Konsekwencją są bardzo 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.

```
1 game.set_window_visible(True)
2 game.set_mode(Mode.SPECTATOR)
3 game.set_screen_format(ScreenFormat.GRAY8)
```

Trajektoria Eksperta zbierana i zapisywana jest następująco.

```
1 game.new_episode()
2 while not game.is_episode_finished():
3     state = game.get_state()
4     game.advance_action()
5     next_state = game.get_state()
6     last_action = game.get_last_action()
7     reward = game.get_last_reward()
8     isterminal = game.is_episode_finished()
9
10    print("State #" + str(state.number))
11    print("Game variables: ", state.game_variables)
12    print("Action:", last_action)
13    print("Reward:", reward)
14    print("=====")
15    memory.append((state.screen_buffer, last_action, next_state.screen_buffer, reward,
16                  isterminal))
```

Zapis trajektorii do pliku wygląda następująco.

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

### 4.3.3 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 potrzebnym. Często 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 części gier osiągał wyniki bliskie maksymalnym i tylko sporadycznie dawał się na początku zająć od tyłu, co skutkowało pojedynczymi niskimi wynikami.

Dla Defend the center agentowi często zdarzało się blokować w rogach labiryntu, wpadając w nieskończoną pętlę akcji. Problem i rozwiązanie zostało opisane w 4.6. Po wyeliminowaniu problemu agent zachowuje się wizualnie sensownie i osiąga 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 zwiększało wyników agenta.

### 4.3.4 Wnioski

W badanych scenariuszach Kopiowanie zachowań osiąga znacznie lepsze wyniki, niż sugerowałyby literatura i uzyskuje jest 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.

## 4.4 Q-learning z Ekspertem

### 4.5 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 pojedynczych, wyrwanych z kontekstu, stanów.

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

### 4.5.1 Implementacja

Zastosowana implementacja jest rozszerzeniem 4.3. Pierwszym krokiem jest załadowanie 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 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 demonstracji 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.

### 4.5.2 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 prawdopodobieństwem może zdecydować o przekazaniu sterowania do agenta.

Losowe przekazywanie sterowania okazało się 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 [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 Eksperta 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 LEWO", "PROSTO" i LEWO i PROSTO"). Podobnie jak przy losowym podejściu, dzięki automatycznemu działaniu możliwe jest działanie programu w przyspieszonym tempie podczas 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 działanie. Problemem w niektórych sytuacjach jest możliwość rozróżnienia, kiedy agent zachowa się niepożądanie i należałoby przejąć sterowanie - w wielu sytuacjach Ekspert reaguje zbyt późno, żeby demonstracja była skuteczna.

#### 4.5.3 Techniczna implementacja

Architektura sieci neuronowej jest identyczna z architekturą zastosowaną w 4.3.

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 (następny znak wpisany w konsoli). Wybrane znaki są przypisane do indeksów wybranych akcji, wpisanie nieznanego znaku powoduje wybranie akcji i 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 4.5.2 konieczne jest asynchroniczne przetwarzanie działania Eksperta. Program nie może oczekiwać na działanie Eksperta, ale kiedy Ekspert zarządzi przekazanie sterowania następna 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

```

#### 4.5.4 Zachowanie

Eksperymenty były prowadzone na przede wszystkim na scenariuszu Health gathering supreme. Początkowe trajektorie Eksperta były wygenerowane zgodnie z 4.6.

Dla każdego z badanych scenariuszy uwzględnianie fragmentów trajektorii zaprezentowanych przez Eksperta w trakcie gry obniża początkowo wyniki. Na skutek nauczania 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).

W scenariuszu Health gathering supreme głównym problemem agenta 4.3 jest nieomijanie min i celem 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) uzyskane 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 jest obecny.

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

#### 4.5.5 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 zamiast do podwyższenia wyników agenta prowadzi do obniżania jego skuteczności na skutek niespójności prezentowanych danych. 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.

### 4.6 Świadomie prezentujący Ekspert

W sekcji 4.3 opisano agenta tworzącego klasyfikator (stan  $\rightarrow$  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 4.5, 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.

Przykładowo, problem wchodzenia w ściany jest łatwo zauważalny podczas obserwacji działania agenta. Oczywistym jest też powód jego występowania - Ekspert, w przeciwieństwie 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 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 podczas 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.

#### 4.6.1 Techniczna implementacja

Architektura sieci neuronowej jest identyczna z architekturą zastosowaną w 4.3.

#### 4.6.2 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 zmniejszyła liczbę niesłusznych 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 osiągnięte 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.

#### 4.6.3 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 4.5 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.

### 4.7 Ogólne wnioski



© 2017 Wojciech Kopeć,

Instytut Informatyki, Wydział Informatyki  
Politechnika Poznańska

Skład przy użyciu systemu L<sup>A</sup>T<sub>E</sub>X.

BibT<sub>E</sub>X:

```
@mastersthesis{ key,  
  author = "Wojciech Kopeć \and ",  
  title = "{Imitation Learning}",  
  school = "Poznan University of Technology",  
  address = "Pozna{\n}, Poland",  
  year = "2017",  
}
```