WYDZIAŁ MATEMATYKI
I INFORMATYKI

PRACA DYPLOMOWA INŻYNIERSKA

INFORMATYKA

Implementacja agenta do Visual Doom AI

Autorzy:

Tomasz Musiała

Adrian Witek

Promotor: dr Jakub Kowalski

Wrocław, wrzesień 2017

....

podpis promotora podpis autora

Streszczenie

Celem naszej pracy była implementacja agenta do Visual Doom AI. Jest to platforma pozwalająca napisanym kontrolerom na rozgrywkę w środowisku gry DOOM, ograniczając dane wejściowe do bufora graficznego. Chcieliśmy aby nasz bot odpowiednio reagował na bodźce otaczającego go świata 3D i efektywnie go eksplorował. W przeciwieństwie do zazwyczaj stosowanych sieci neuronowych, użyliśmy analitycznych technik przetwarzania obrazu. Powstały agent spełnia założone wymagania. Uzyskane wyniki pokazują, że zaimplementowane przez nas metody mogą konkurować z technikami wykorzystującymi sieci neuronowe.

Spis treści

Rozdział 1

Wstęp

1.1 Cel pracy

Tworząc agenta, który będzie się zachowywał efektywnie w środowisku ViZDoom mając za dane wejściowe jedynie obraz z bufora graficznego, zazwyczaj stosuje się techniki intensywnie wykorzystujące sieci neuronowe. My jednak chcieliśmy spróbować innego podejścia. Naszym celem było wykorzystanie różnych sposobów analizy obrazu do rozpoznawania elementów środowiska agenta, a także zastosowanie odpowiednich algorytmów, które umożliwią agentowi skuteczne poruszanie się po mapie, jej eksplorację oraz walkę. Postanowiliśmy przetestować jakie wyniki da czysto analityczne podejście.

Rozdział 2

ViZDoom

2.1 Opis platformy

ViZDoom jest platformą pozwalającą na prowadzenie prac badawczych dotyczących uczenia maszynowego z informacji wizualnych, której autorami są Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek oraz Wojciech Jaśkowski z Instytutu Informatyki na Politechnice Poznańskiej [7]. Środowisko oparte jest na znanej grze z gatunku strzelanek pierwszoosobowych (FPS, ang. first-person shooter) DOOM [6]. Agent (bot) w ViZDoom musi skutecznie postrzegać i interpretować świat 3D w celu jego eksploracji oraz podejmowania strategicznych, taktycznych decyzji związanych z walką i zbieraniem przedmiotów.

Platforma posiada interfejs (API, ang. Application Programming Interface) dla języków `Python`, `C++`, `Java` oraz `Lua`. Środowisko jest szeroko konfigurowalne, definiować można niestandardowe scenariusze różniące się mapami, elementami środowiska, celami i nagrodami.

Konfigurowalne są również inne parametry takie jak:

- rozdzielczość generowanego bufora ekranu,
- format obrazu (kolor, skala szarości),
- renderowanie broni,
- renderowanie zwłok,
- renderowanie HUD (ang. heads-up display),
- renderowanie efektów,

- odtwarzanie dźwięku,
- dostępne ruchy,
- dostępne zmienne gry

i inne dotyczące renderowania.

Ruchy, które może wykonywać opisany w pracy agent to:

- obrót w prawo/lewo,
- ruch do przodu/tyłu,
- ruch w prawo/lewo,
- atak.

Mamy również dostęp do następujących zmiennych:

- ilość amunicji,
- poziom życia,
- poziom opancerzenia,
- liczba fragów,
- gotowość do ataku.

```
from vizdoom import *
from random import choice
from time import sleep, time

game = DoomGame()
game.load_config("../config/basic.cfg")
game.init()

# Sample actions. Entries correspond to buttons:
# MOVE_LEFT, MOVE_RIGHT, ATTACK
actions = [[True, False, False],
     [False, True, False], [False, False, True]]
# Loop over 10 episodes.
for i in range(10):
  game.new_episode()
  while not game.is_episode_finished():
    # Get the screen buffer and and game variables
    s = game.get_state()
    img = s.image_buffer
    misc = s.game_variables
    # Perform a random action:
    action = choice(actions)
    reward = game.make_action(action)
    # Do something with the reward...

  print("total reward:", game.get_total_reward())
```

Rysunek 2.1: Przykładowy kod agenta ViZDoom

Rysunek 2.2: Widok pierwszoosobowy podczas rozgrywki

2.2 Zawody

Od dwóch lat (rok 2016 i 2017) odbywają się zawody Visual Doom AI Competition. Uczestnicy muszą napisać i wysłać kontroler, który gra w Dooma korzystając z API ViZDoom. Zwycięzca zostaje wybrany w turnieju śmierci (ang. deathmatch tournament).
Zawody składają się z dwóch etapów, pierwszy z nich to ograniczony pojedynek na znanej mapie. Agent ma dostęp do tylko jednej broni, z którą zaczyna mecz, i może zbierać wyłącznie apteczki oraz amunicję.
W drugim etapie agent ma dostęp do wielu różnych broni i przedmiotów, dostarczone są mapy na których można trenować agenta. Ostateczna ewaluacja odbywa się na kilku nieznanych uczestnikom mapach. Każda gra, podczas której wszystkie kontrolery walczą ze sobą nawzajem, trwa 10 minut. Pojedynki na każdej mapie powtarzane są co najmniej 10 razy.
Kontrolery są klasyfikowane według liczby fragów, gdzie frag jest zdefiniowany jako: `liczba zabitych przeciwników − liczba samobójstw`.
Każda z drużyn może zgłosić tylko jednego bota, a drużyny i boty nie mogą ze sobą współpracować.

2.3 Najczęściej stosowane algorytmy

Wielu uczestników Visual Doom AI Competition publikuje prace opisujące szczegóły rozwiązań zastosowanych przy implementacji ich bota. Na podstawie tych publikacji przyjrzyjmy się metodom, które zostały użyte:

Tytuł pracy	Autorzy	Zastosowana metoda
Autoencoder-augmented Neuroevolution for Visual Doom Playing[2]	Samuel Alvernaz, Julian Togelius	Sieci neuronowe (Autoenkoder)
Learning to Act by Predicting the Future[5]	Alexey Dosovitskiy, Vladlen Koltun	Sieci neuronowe
Training Agent for First-Person Shooter Game with Actor-Critic Curriculum Learning[11]	Yuxin Wu, Yuandong Tian	Sieci neuronowe
Arnold: An Autonomous Agent to play FPS Games[4]	Devendra Singh Chaplot, Guillaume Lample	Sieci neuronowe (Deep Q Learning)
Playing Doom with SLAM-Augmented Deep Reinforcement Learning[3]	Shehroze Bhatti, Alban Desmaison, Ondrej Miksik, Nantas Nardelli, N. Siddharth, Philip H. S. Torr	Sieci neuronowe (Deep Q Learning)
Playing FPS Games with Deep Reinforcement Learning[10]	Guillaume Lample, Devendra Singh Chaplot	Sieci neuronowe (Deep Reinforcement Learning)
Deep Successor Reinforcement Learning[9]	Tejas D. Kulkarni, Ardavan Saeedi, Simanta Gautam, Samuel J. Gershman	Sieci neuronowe (Deep Reinforcement Learning)

O ile poszczególne techniki różnią się podejściem do problemu, to mają jedną wyraźną cechę wspólną — wszystkie opierają się na sieciach neuronowych.

Rozdział 3

Użyte algorytmy i techniki

3.1 Wskaźnik podobieństwa strukturalnego (SSIM)

Informacje strukturalne obrazu są to te jego atrybuty, które reprezentują strukturę obiektów na scenie, niezależnie od średniej luminancji i kontrastu. Ponieważ luminancja i kontrast mogą zmieniać się w różnych miejscach sceny, używamy lokalnej luminancji i kontrastu dla naszej definicji.
Załóżmy, że x i y są dwoma nieujemnymi sygnałami obrazów (wektorami wartości pikseli na obrazie), które zostały przyrównane do siebie nawzajem. Jeśli założymy, że jeden z obrazów jest doskonałej jakości, wtedy miara podobieństwa może służyć jako wskaźnik jakości drugiego obrazu. Zadanie zmierzenia podobieństwa dzielimy na wykonanie trzech porównań:

- luminancji,
- kontrastu,
- struktury.

Chcemy również, aby miara podobieństwa spełniała trzy następujące warunki:

- Symetria: $S(x, y) = S(y, x)$
- Ograniczenie: $S(x, y) \leq 1$
- Unikalne maksimum: $S(x, y) = 1$ wtedy i tylko wtedy gdy $x = y$ (w reprezentacji dyskretnej $x_i = y_i$ dla każdego $i = 1, 2, \ldots, N$).

Po pierwsze porównuje się luminancję każdego sygnału. Przyjmując sygnały dyskretne, szacuje się ją jako średnią intensywność

$$\mu_x = \frac{1}{N}\sum_{i=1}^{N} x_i. \tag{3.1}$$

Funkcja porównania luminancji $l(x, y)$ jest wtedy funkcją μ_x i μ_y

$$l(x, y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1} \tag{3.2}$$

gdzie stała C_1 jest uwzględniona by uniknąć niestabilności gdy $\mu_x^2 + \mu_y^2$ jest bardzo bliskie zera. W szczególności wybieramy

$$C_1 = (K_1 L)^2 \tag{3.3}$$

Gdzie L to zakres wartości pikseli (255 dla 8-bitowego obrazu w skali szarości), a $K \ll 1$ to mała stała. Podobne rozważania dotyczą również porównywania kontrastu i struktury.

Po drugie wykorzystujemy odchylenie standardowe (pierwiastek kwadratowy wariancji) jako oszacowanie kontrastu sygnału

$$\sigma_x = \left(\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu_x)^2\right)^{\frac{1}{2}}. \tag{3.4}$$

Porównanie kontrastu $c(x, y)$ jest porównaniem σ_x i σ_y

$$c(x, y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2} \tag{3.5}$$

Po trzecie, porównanie struktury jest przeprowadzane na znormalizowanych sygnałach $(x - \mu_x)/\sigma_x$ i $(y - \mu_y)/\sigma_y$. Definiujemy funkcję porównania struktury w następujący sposób

$$s(x, y) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3} \tag{3.6}$$

W dyskretnej formie, σ_{xy} może zostać przybliżona przez

$$\sigma_{xy} = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu_x)(y_i - \mu_y). \tag{3.7}$$

Wreszcie łączymy trzy porównania (3.2), (3.5) oraz (3.6) i nazywamy powstałą miarę wskaźnikiem podobieństwa strukturalnego (SSIM, ang. Structural Similarity) pomiędzy sygnałem x i y

$$SSIM(x, y) = [l(x, y)]^\alpha \cdot [c(x, y)]^\beta \cdot [s(x, y)]^\gamma \tag{3.8}$$

gdzie $\alpha > 0$, $\beta > 0$ oraz $\gamma > 0$ są parametrami stosowanymi w celu dostosowania względnego znaczenia trzech składników. Łatwo sprawdzić że ta definicja spełnia trzy przedstawione powyżej warunki. W celu zmniejszenia złożoności wyrażenia ustalamy $\alpha = \beta = \gamma = 1$ oraz $C_3 = C_2/2$. Otrzymujemy specyficzną formę wskaźnika SSIM

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}. \tag{3.9}$$

Po bardziej szczegółowy opis algorytmu SSIM odsyłamy Czytelnika do [12].

3.2 Dopasowywanie wzorca

Wyszukiwanie wzorca na obrazie polega na przesuwaniu wzorca o szerokości w i wysokości h po obrazie o większej rozdzielczości i porównywaniu nakładających się części. Porównanie to wykonywane jest jedną z następujących metod [1]:

- Metoda dopasowania różnicy kwadratów

$$R_{sq\_diff} = \sum_{x',y'} [T(x', y') - I(x + x', y + y')]^2 \tag{3.10a}$$

- Znormalizowana metoda dopasowania różnicy kwadratów

$$R_{sq\_diff\_normed} = \frac{\sum_{x',y'} [T(x', y') - I(x + x', y + y')]^2}{\sqrt{\sum_{x',y'} T(x', y')^2 \cdot \sum_{x',y'} I(x + x', y + y')^2}} \tag{3.10b}$$

- Metoda dopasowywania korelacji

$$R_{ccorr} = \sum_{x',y'} T(x', y') - I(x + x', y + y') \tag{3.10c}$$

- Znormalizowana metoda dopasowywania korelacji krzyżowej

$$R_{ccorr\_normed} = \frac{\sum_{x',y'} T(x', y') \cdot I(x + x', y + y')}{\sqrt{\sum_{x',y'} T(x', y')^2 \cdot \sum_{x',y'} I(x + x', y + y')^2}} \tag{3.10d}$$

- Metoda dopasowywania współczynników korelacji

$$R_{ccoeff} = \sum_{x',y'} T'(x', y') \cdot I'(x + x', y + y') \tag{3.10e}$$

$$T'(x', y') = T(x', y') - \frac{\sum_{x'',y''} T(x'', y'')}{(w - h)} \tag{3.10f}$$

$$I'(x + x', y + y') = I(x + x', y + y') - \frac{\sum_{x'',y''} I(x'', y'')}{(w - h)} \tag{3.10g}$$

- Znormalizowana metoda dopasowywania współczynników korelacji

$$R_{ccoeff\_normed} = \frac{\sum_{x',y'} T'(x',y') \cdot I'(x+x', y+y')}{\sqrt{\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} I'(x+x', y+y')^2}} \tag{3.10h}$$

gdzie T' i I' zdefiniowane są tak jak w (3.10f) i (3.10g).

Wraz ze wzrostem złożoności metod porównywania wzrasta również jakość wykonywanych porównań. W naszej implementacji wykorzystana została znormalizowana metoda porównywania współczynników korelacji (3.10h) ze względu na to, że potrzebna była jak największa precyzja wykrywania.

3.3 Wybrane elementy algorytmu eksplorowania mapy

W przypadku gdy agent nie walczy ani nie szuka żadnego zasobu, zasadnym jest zastosowanie sensownego algorytmu do eksploracji mapy. Naszym zdaniem powinien on spełniać następujące warunki:

- nie wpadać na ściany,
- płynnie poruszać się po mapie.

3.3.1 Szczytowy stosunek sygnału do szumu (PSNR)

Zaimplementowany algorytm opiera się na rozróżnieniu w poszczególnych klatkach tekstury podłogi od reszty obrazu. Wyzwaniem było znalezienie odpowiedniego charakteryzatora, pozwalającego osiągnąć ten cel z satysfakcjonującą skutecznością jednocześnie działając w czasie rzeczywistym. Dosyć dobre wyniki okazała się dać nam metoda bazująca na tzw. szczytowym stosunku sygnału do szumu (ang. peak signal-to-noise ratio, najczęściej opisywane skrótem PSNR). PSNR wyznacza wartość dla dwóch obrazów wyrażającą się wzorem $\frac{MAX^2}{MSE}$, gdzie MAX jest maksymalną możliwą wartością piksela,

natomiast MSE został opisany wzorem (4.1). Szczytowy stosunek sygnału do szumu najczęściej jest stosowany przy porównaniu jakości obrazów np. po kompresji. Więcej szczegółów można znaleźć tu [8]. W naszym przypadku, o ile tekstury dostatecznie się różniły, metoda ta pozwoliła osiągnąć zadowalające efekty.

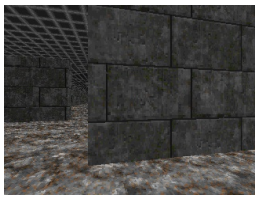
3.3.2 Wykrywanie podłogi

Wykrywając w którym miejscu obrazu znajduje się podłoga, wychodzimy z następujących założeń:

- podłoga, jeżeli występuje, to zawiera się w dolnym rzędzie klatki,
- podłoga nie przekracza dolnej połowy obrazu (ze względu na stałą perspektywę).

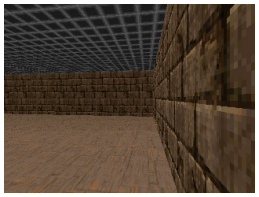
Rozwiązanie, które zaimplementowaliśmy w oparciu o powyższe warunki opiszę poniżej. Najpierw dzielimy obraz na siatkę kwadratów o zadanym boku. Docelowo chcemy otrzymać macierz o wymiarach takich, jak powstała siatka, której elementami będą 1 (podłoga) lub 0 (coś innego). Najpierw jednak używając PSNR porównujemy wszystkie pola otrzymanej siatki z jej lewym dolnym rogiem. Otrzymane w ten sposób wartości są z zakresu $0 - 50$. Im wyższa wartość, tym pola siatki były bardziej podobne. Zazwyczaj na testowych przykładach otrzymywaliśmy dość wyraźną grupę wysokich i niskich wartości. Grupujemy te wartości w dwa zbiory (za graniczną wartość przyjmujemy zwykłą średnią arytmetyczną wartości w macierzy), ten z wysokimi liczbami jest bardziej podobny do porównanego kwadratu z siatki. Niestety, jeden taki pomiar zazwyczaj pozostawiał sporo niedokładności i szumów. Aby wzmocnić skuteczność rozpoznania przeprowadzamy takie porównania dla każdego elementu z najniższego wiersza obrazu, który jest podobny do lewego dolnego rogu. Wyniki sumujemy i wyciągamy średnią arytmetyczną (dzielimy przez ilość przeprowadzonych porównań). Przykładowe wyniki można zobaczyć na rysunku 3.1 Po tych operacjach wysokie wartości zamieniamy na 1, a pozostałe na 0. Ponieważ chcielibyśmy, aby zbiory 1 i 0 były jak najbardziej „gładkie", stosujemy następujący algorytm na pozbycie się pojedynczych zer w obszarze jedynek:

- Krok 1: znajdź zbiór S zer sąsiadujących z co najmniej jedną jedynką,

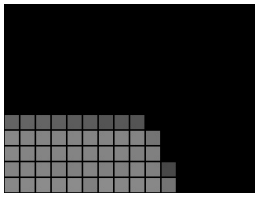


(a) Przykładowy obraz

(b) Wynik porównania PSNR



(c) Przykładowy obraz



(d) Wynik porównania PSNR

Rysunek 3.1: Przykładowe wyniki porównania PSNR

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1
1	1	1	1	1	1	1	1
1	0	0	1	1	1	1	0

Tablica początkowa

0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1

Tablica po kroku 2

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1

Tablica po kroku 4

Rysunek 3.2: Ilustracja wykonania algorytmu

- Krok 2: zamień wszystkie elementy S na 1,
- Krok 3: znajdź zbiór S' jedynek sąsiadujących z co najmniej jednym zerem,
- Krok 4: zamień wszystkie elementy S' na 0.

W naszym algorytmie „sąsiadujących” oznacza „mających jeden wspólny bok”. Na rysunku 3.2 przedstawione zostało działanie powyższego algorytmu. Następnym problemem jest określenie, które z wartości są podłogą. Porównując na początku lewy dolny element nie wiedzieliśmy czym on był. Tutaj wykorzystujemy fakt, że w górnym wierszu siatki nie powinna występować podłoga. Liczymy więc, których elementów jest tam więcej - jeżeli 0, to znaczy, że one są podłogą i wtedy zamieniamy 0 na 1 i odwrotnie. W przeciwnym razie nie musimy nic robić. W wyniku otrzymaliśmy macierz, gdzie jedynki odpowiadają wykrytej podłodze.

3.3.3 Wybór ruchu

Wybierając ruch, jaki ma wykonać nasz agent na podstawie macierzy z wykrytą wcześniej podłogą, staramy się dobrać kierunek, w którym widoczna jest jej największa część. W tym celu przeszukujemy wszystkie kolumny i zliczamy ile w każdej z nich jest następujących po sobie jedynek począwszy od dołu. Następnie wybieramy największą spójny fragment z maksymalną wartością zliczeń. Jego środkowa kolumna będzie szukanym kierunkiem. W przypadku, gdy agent jest zbyt blisko ściany (podłoga nie przekracza dwóch pierwszych rzędów siatki), wybierany jest ruch do tyły połączony z obrotem.

Rozdział 4

Opis implementacji

Agent został zaimplementowany zgodnie z paradygmatem programowania obiektowego. Złożony jest z oddzielnych modułów zajmujących się:

- detekcją obiektów,
- podejmowaniem decyzji o kolejnych akcjach,
- eksploracją mapy.

Każdy z modułów został szczegółowo opisany w kolejnych podrozdziałach.

4.1 Detekcja obiektów

Moduł ten zajmuje się wykrywaniem miejsc na obrazie, w których znajduje się podany wzorzec. Wzorce mogą występować w różnych rozmiarach, dlatego też pojedyncze uruchomienie algorytmu dopasowania (3.2) nie wystarczy. Na początku obraz jak i wzorzec przekształcane są do formatu w skali szarości, ale kopie kolorowych obrazów są zachowywane.

```
    self.graySprite = cv2.cvtColor(sprite["image"], cv2.COLOR_BGR2GRAY)
    self.sprite = sprite

    self.currentImg = img
    self.grayImg = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

Do obrazu dodawana jest czarna ramka po to, żeby algorytm wyszukiania wzorca mógł wykryć obiekty, które są cześciowo poza obrazem.

```
grayWithBorder = cv2.copyMakeBorder(self.grayImg, borderSize[0],
                                    borderSize[0], borderSize[1],
                                    borderSize[1], cv2.BORDER_CONSTANT,
                                    value=[0, 0, 0])
```

Wzorzec jest skalowany i dla każdego jego rozmiaru, od największego (obiekty blisko agenta) do najmniejszego (obiekty daleko od agenta), uruchamiany jest algorytm wykrywania wzorca.

```
graySpriteResized = cv2.resize(self.graySprite, None, fx=ratio, fy=ratio,
                                    interpolation=interp)
res = cv2.matchTemplate(grayWithBorder, graySpriteResized, cv2.
                                    TM_CCOEFF_NORMED)
```

Po tym etapie otrzymujemy współrzędne wystąpień wzorca, lecz nie są one pewne, dlatego musimy zrewidować każdy z nich. Potwierdzone wystąpienia trzymane są w tablicy, a kolejne przetwarzane wystąpienie jest od razu odrzucane jeśli występuje w małej odległości od nich - jest to duplikat już potwierdzonego wystąpienia.
Następnie wykonywana jest weryfikacja wykrycia:

- porównanie jasności
- porównanie kolorów
- obliczenie wskaźnika podobieństwa strukturalnego

```
if not isDetected(actualPt)
        and checkBrightness(actualPt, actW, actH)
        and checkColors(actualPt, actW, actH)
        and checkSsimMatch(pt):
  print("DETECTION", [actualPt, (w, h)])
  self.matches.append([actualPt, (w, h)])
```

Wskaźnik jasności obliczany jest używając wzoru na błąd średniokwadratowy (MSE, ang. Mean Squared Error)

$$MSE = \frac{1}{N} \sum_{x,y} I[x,y]^2 \tag{4.1}$$

gdzie I jest obrazem w skali szarości.

```
def checkBrightness(pt, actW, actH):
  toCheck = self.grayImg[pt[1]:pt[1] + actH, pt[0]:pt[0] + actW]
  im = Image.fromarray(toCheck)
  br1 = ImageStat.Stat(im).rms[0]
  br2 = self.sprite["grayBrightness"]
  return not (br1 < br2*0.3 or br1 > br2 + br2*0.3)
```

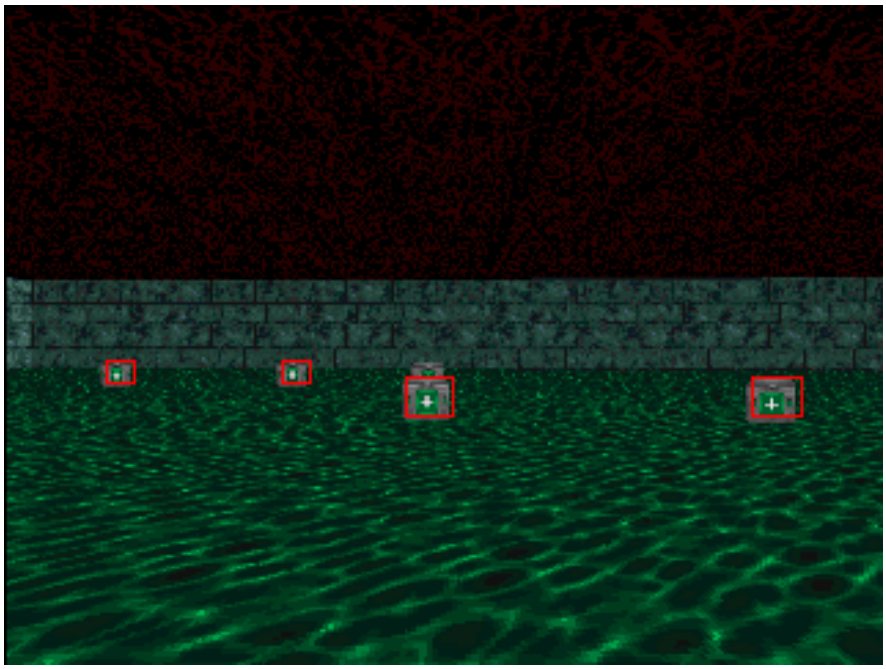
Kolejnym etapem jest sprawdzenie zgodności kolorów. Obliczane są średnie wartości trzech kolorów (czerwony, zielony, niebieski) dla wzorca oraz miejsca wykrycia i porównywane są stosunki kolorów czerwony/zielony oraz czerwony/niebieski.

```
def checkColors(pt, actW, actH):
  toCheck = self.currentImg[pt[1]:pt[1] + actH, pt[0]:pt[0] + actW]
  im = Image.fromarray(toCheck)
  r1, g1, b1 = ImageStat.Stat(im).mean
  r2, g2, b2 = self.sprite["meanColors"]
  return abs(r1/g1 - r2/g2) < self.colorthr and abs(r1/b1 - r2/b2) < self.
                                    colorthr
```

Ostatni etap to obliczenie wskaźnika podobieństwa strukturalnego (3.1).

```
def checkSsimMatch(pt):
  toCheck = grayWithBorder[pt[1]:pt[1] + h, pt[0]:pt[0] + w]
  win_size = (min([w, h]) / 2) * 2 - 1
  win_size = win_size if win_size < 11 else None
  s = ssim(graySpriteResized, toCheck, win_size=win_size)
  return s >= self.ssimthr
```

Po przejściu weryfikacji, lista z potwierdzonymi wystąpieniami jest powiększana o aktualnie przetwarzany element.



Rysunek 4.1: Wynik wyszukiwania apteczek przez moduł detekcji

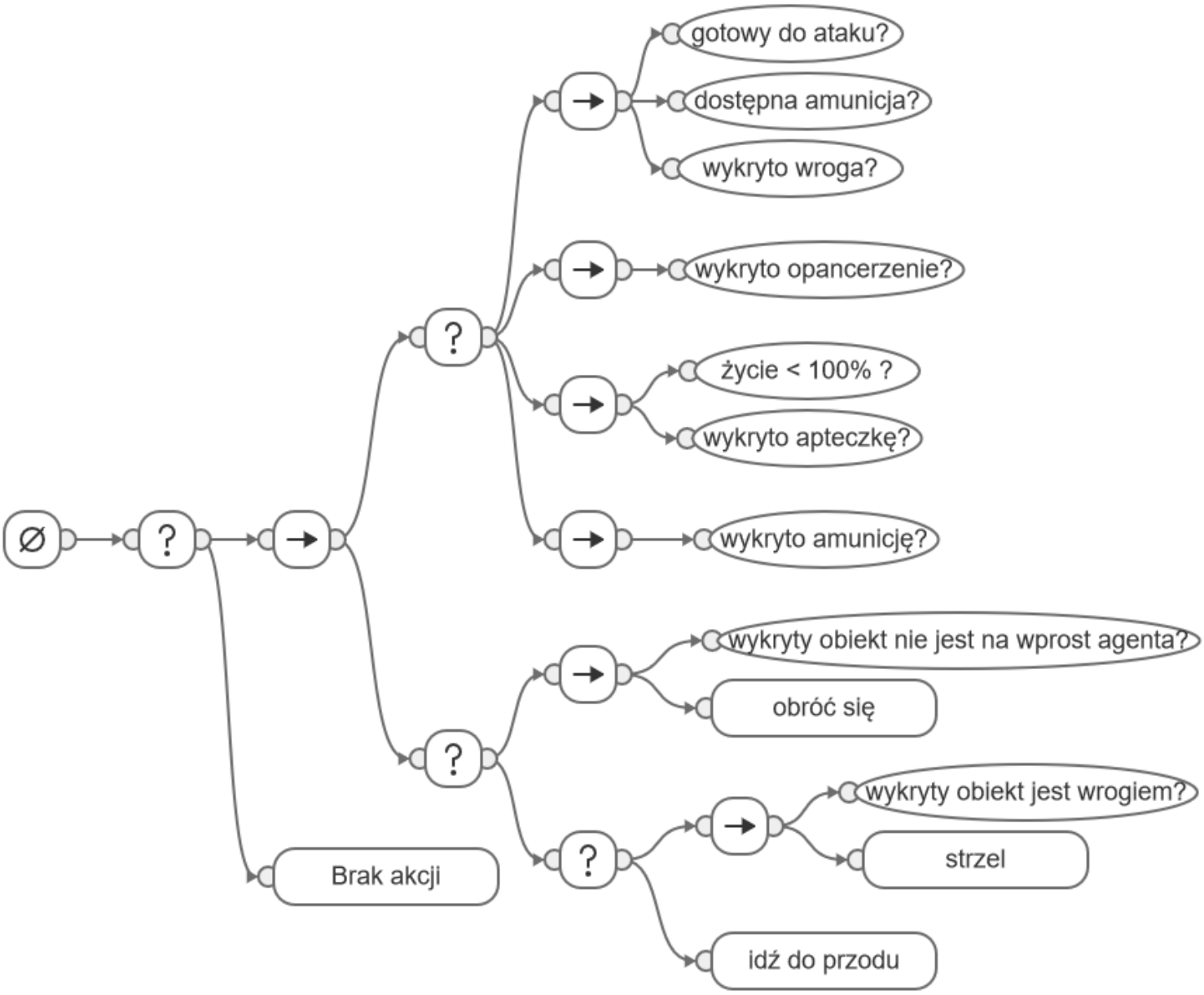
4.2 Podejmowanie decyzji

Moduł ten zajmuje się podejmowaniem decyzji o kolejnych ruchach bazując na pozycjach wykrytych obiektów. Zaimplementowane zostało drzewo behawioralne widoczne na rysunku 4.2. Po wykryciu obiektu przez moduł detekcji, agent obraca się w stronę obiektu i zależnie od jego typu wykonuje odpowiednią akcję przez określoną ilość cykli.

Jeśli wykryty obiekt nie jest na wprost agenta musi on wykonać obrót. Obliczany jest kąt α (rysunek 4.3) pomiędzy obiektem a linią przebiegającą przez środek obrazu, a następnie wykonywany jest obrót w odpowiednią stronę. Ilość cykli D przez które ma trwać obrót obliczana jest wzorem

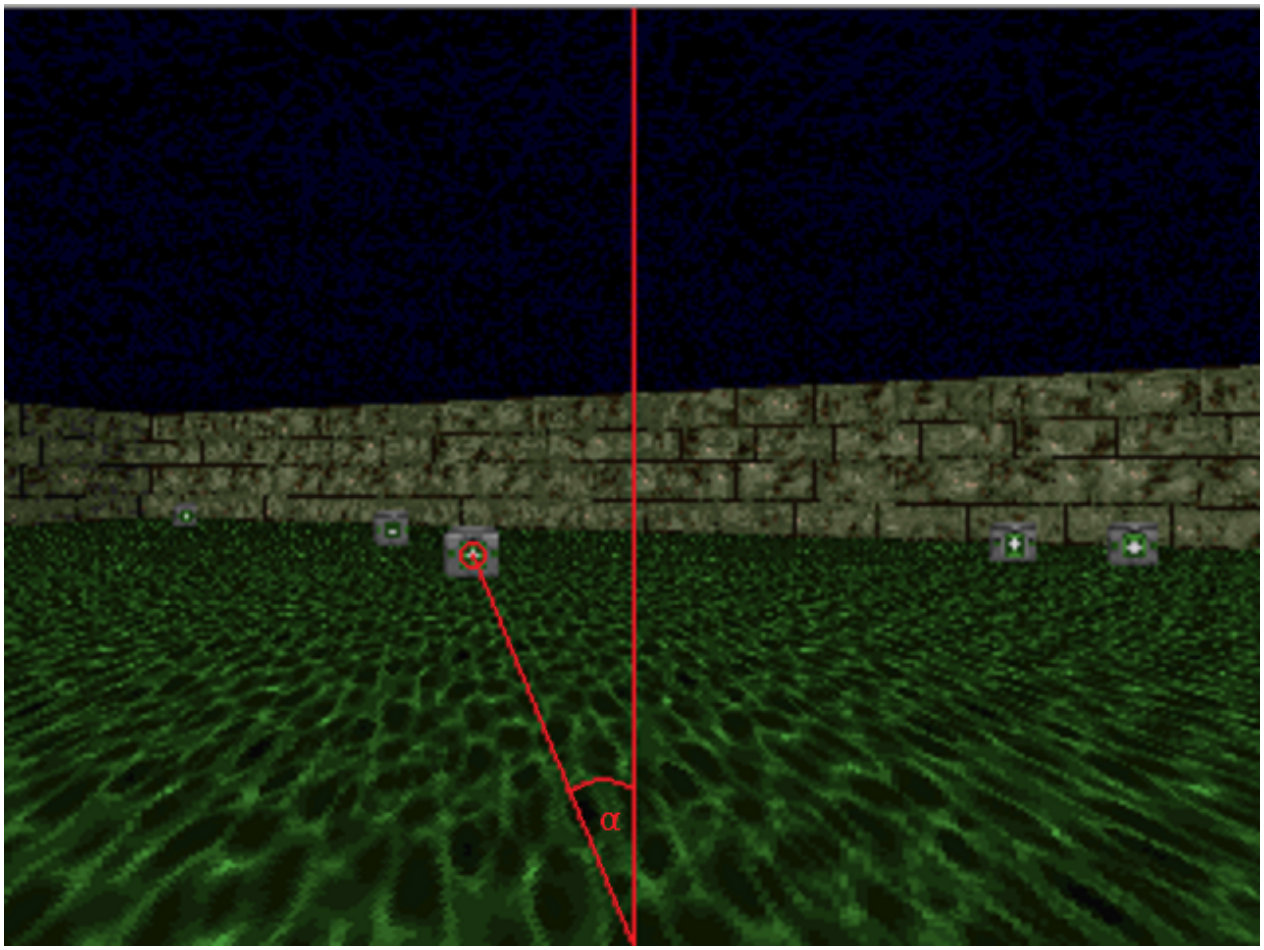
$$D = \frac{\alpha}{K}, \tag{4.2}$$

gdzie $K = 5.346$ jest wielkością kąta o który obraca się agent wykonując komendę obrotu przez jeden cykl.



Rysunek 4.2: Drzewo behawioralne części walcząco-zbierającej agenta.

Jeśli żadna z akcji nie zostanie wybrana, sterowanie przekazywane jest do modułu eksploracji.



Rysunek 4.3: Kąt obrotu potrzebny do wycentrowania obiektu.

4.3 Eksploracja

Moduł ten pozwala na wykrycie tekstury podłogi na obrazie oraz obliczenie współrzędnej x, dla której widoczny jest największy jej fragment. Klasa `FloorDetector` zawiera konstruktor, który powinniśmy zainicjalizować podając rozmiar pojedynczego pola siatki, na której będziemy pracować. Dostępnych mamy kilka metod, które pokrótce opiszę poniżej:

- `getPsnrMatch(image)` — zwraca opisaną w rozdziale 3.3.2 macierz o rozmiarze siatki z wartościami porównań PSNR — wykorzystujemy tu funkcję modułu `skimage compare_psnr`,
- `detectFloor(array)` — zwraca macierz wypełnioną 1 w miejscu podłogi i 0 w

pozostałych polach,

- `clearZerosNoise(array)` — zwraca macierz, w której usunięto szumy spowodowane przez pojedyncze 0 — szczegóły w rozdziale 3.3.2,
- `getXDirection(array)` — zwraca parę (x, wartość), gdzie x jest kolumną siatki, w której znajduje się największy fragment podłogi, a wartość jest ilością pól siatki do niej należący,
- `getBestDirection(array)` — metoda, która korzystając z powyżej opisanych funkcji zwraca odpowiednią kolumnę, bądź w przypadku, gdy rozpoznanie podłogi nie spełnia naszych założeń, -1.

Rozdział 5

Instalacja i uruchomienie

Poniższy opis instalacji dotyczy systemu Unix\Linux. Agent napisany jest w języku Python 2.7.12, do jego uruchomienia potrzebna jest wersja 2.7+. Program korzysta z następujących zewnętrznych modułów:

- `vizdoom v1.1.0`[1]
- `opencv-python v3.2.0.6`[2]
- `Pillow v4.0.0`[3]
- `numpy v1.12.0`[4]
- `scikit-image v0.12.3`[5]
- `scipy v0.19.0`[6]

Moduły te są wymagane do działania programu, wersja każdego z nich musi być równa lub wyższa od podanej powyżej. Opis instalacji platformy ViZDoom znajduje się pod adresem `https://github.com/mwydmuch/ViZDoom`. Pozostałe biblioteki można zainstalować za pomocą menedżera pakietów `pip`:

```
>$ sudo apt-get install python-pip
```

a następnie:

[1] `https://github.com/mwydmuch/ViZDoom`
[2] `https://pypi.python.org/pypi/opencv-python`
[3] `http://pillow.readthedocs.io/en/3.2.x/`
[4] `https://pypi.python.org/pypi/numpy`
[5] `http://scikit-image.org/docs/dev/install.html`
[6] `https://pypi.python.org/pypi/scipy`

```
>$ pip install <packagename>==<version>
```

gdzie `<packagename>` oraz `<version>` to kolejne nazwy pakietów wraz z ich wersjami. Aby uruchomić agenta, w głównym folderze projektu wykonujemy komendę:

```
>$ python main.py
```

Rozdział 6

Wyniki i wnioski

Do przetestowania modułu detekcji wraz z modułem podejmowania decyzji wykorzystany został scenariusz, w którym agent znajdujący się w pomieszczeniu co kilka cykli traci część punktów życia. Aby przeżyć, musi on wykrywać i zbierać kolejne apteczki pojawiające się wokół niego. Wykonano 1000 uruchomień scenariusza, każde z nich trwało do momentu śmierci agenta lub do wykonania 1000 cykli. W 963 przypadkach bot zdołał zbierać apteczki i utrzymać się przy życiu, a tylko 37 z nich (3,7%) zakończyło się śmiercią agenta.

Aby przetestować skuteczność wykrywania podłogi na obrazach przeprowadziliśmy test polegający na uruchomieniu bota dla trzech scenariuszy (czas ustalony został na 1000 cykli). Policzyliśmy stosunek poprawnie rozpoznanych klatek do sumy wszystkich, które zostały interpretowane. Otrzymane wyniki zostały przedstawione w tabeli 6.1. Mapa health_gathering.wad miała teksturę podłogi bardzo zbliżoną do otoczenia, co tłumaczy niski wynik.

Mapa	Wynik
basic.wad	96%
health_gathering_supreme.wad	79%
health_gathering.wad	51%

Tabela 6.1: Skuteczność wykrywania podłogi na poszczególnych mapach.

Powyższe rezultaty pozwalają stwierdzić, że założenia dotyczące zachowania agenta zostały spełnione. Ma on wysoką skuteczność w interpretacji środowiska, sprawnie wykrywa obiekty, które są w polu jego widzenia, podejmuje decyzje adekwatne do sytuacji w jakich się znajduje oraz efektywnie eksploruje mapę. Pokazaliśmy więc, że wyniki uzyskane przy pomocy opisanych w niniejszej pracy technik i algorytmów są porównywalne z rezultatami osiągniętymi przez agentów opartych na sieciach neuronowych.

Bibliografia

[1] Gary Bradski Adrian Kaehler. *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library*. O'Reilly Media, Inc., 2016.

[2] Samuel Alvernaz and Julian Togelius. Autoencoder-augmented neuroevolution for visual doom playing. *CoRR*, abs/1707.03902, 2017.

[3] Shehroze Bhatti, Alban Desmaison, Ondrej Miksik, Nantas Nardelli, N. Siddharth, and Philip H. S. Torr. Playing doom with slam-augmented deep reinforcement learning. *CoRR*, abs/1612.00380, 2016.

[4] Devendra Singh Chaplot* and Guillaume Lample*. Arnold: An autonomous agent to play fps games. In *31st AAAI Conference on Artificial Intelligence (AAAI-17), San Francisco, USA.*, 2017.

[5] Alexey Dosovitskiy and Vladlen Koltun. Learning to act by predicting the future. *CoRR*, abs/1611.01779, 2016.

[6] Software Id. DOOM, 1993.

[7] Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. ViZDoom: A Doom-based AI research platform for visual reinforcement learning. In *IEEE Conference on Computational Intelligence and Games*, pages 341–348, Santorini, Greece, Sep 2016. IEEE. The best paper award.

[8] Zoran Kotevski and Pece Mitrevski. *Experimental Comparison of PSNR and SSIM Metrics for Video Quality Estimation*, pages 357–366. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[9] T. D. Kulkarni, A. Saeedi, S. Gautam, and S. J. Gershman. Deep Successor Reinforcement Learning. *ArXiv e-prints*, June 2016.

[10] Guillaume Lample and Devendra Singh Chaplot. Playing FPS games with deep reinforcement learning. *CoRR*, abs/1609.05521, 2016.

[11] Yuxin Wu and Yuandong Tian. Training agent for first-person shooter game with actor-critic curriculum learning. 2017.

[12] IEEE Alan C. Bovik Fellow IEEE Hamid R. Sheikh Student Member IEEE Zhou Wang, Member and IEEE Eero P. Simoncelli, Senior Member. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing*, 13(4), 2004.