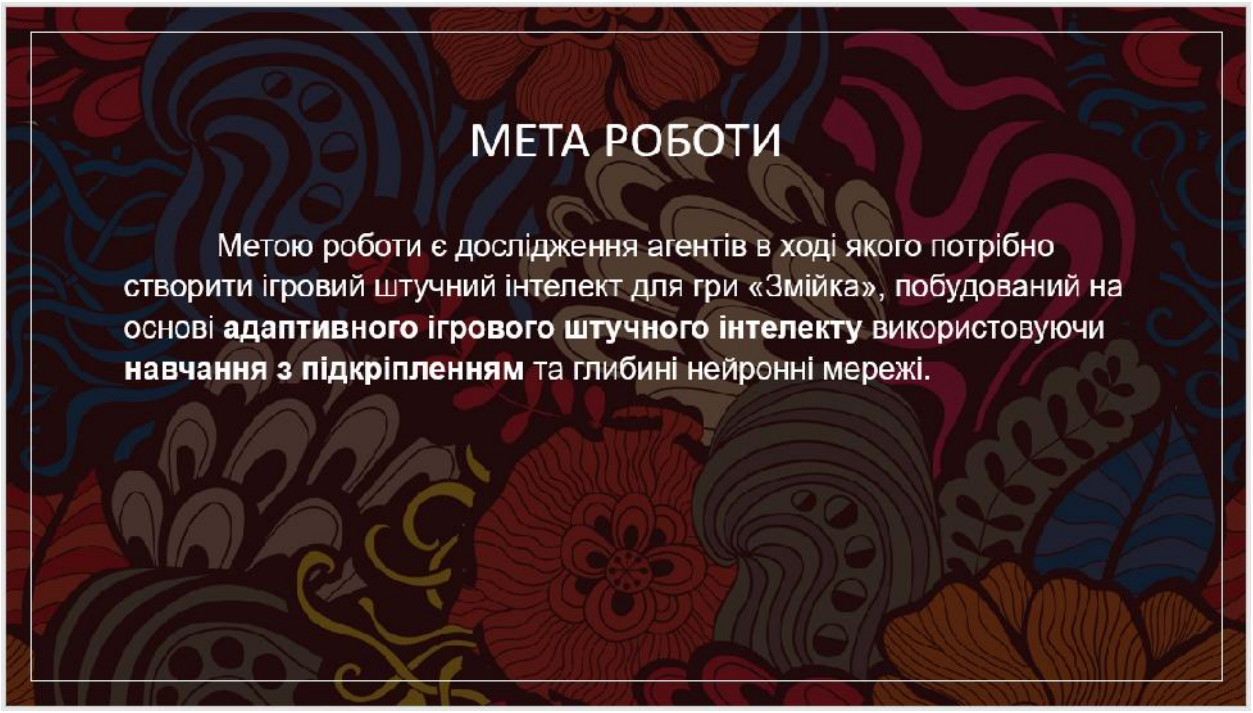
ДОДАТОК А

Графічний матеріал кваліфікаційної роботи


МЕТОДИ ЗАБЕЗПЕЧЕННЯ АДАПТИВНОСТІ ІГРОВОГО ШТУЧНОГО ІНТЕЛЕКТУ
Автор:
Супруненко
ст. гр. СПм-20-1
Керівник:
Іващенко Г.С.
доцент кафедри ЕСМ,
к.т.н


МЕТА РОБОТИ
Метою роботи є дослідження агентів в ході якого потрібно створити ігровий штучний інтелект для гри «Змійка», побудований на основі **адаптивного ігрового штучного інтелекту** використовуючи **навчання з підкріпленням** та глибині нейронні мережі.

АКТУАЛЬНІСТЬ РОБОТИ

Актуальність дослідження обумовлена популяризацією методів навчанням з підкріпленням для розробки ігрових штучних інтелектів.

Також актуальності сприяє поширення у користувачів спеціалізованих обчислювальних ресурсів, що дозволяє використовувати ШНМ для недетермінованої поведінки та враховувати попередній досвід.

ЗАДАЧІ РОБОТИ

- Аналіз предметної області
- Створення ігрового середовища для ігри «Змійка»
- Дослідження методів забезпечення адаптивності ігрового штучного інтелекту та створення агенту
- Дослідження ефективності створених агентів
- Порівняння розробленого адаптивного агенту з існуючими реалізаціями ігрового штучного інтелекту у рамках змагання на платформі **Kaggle**

МЕТОДИ ЗАБЕЗПЕЧЕННЯ ІГРОВОГО ШТУЧНОГО ІНТЕЛЕКТУ

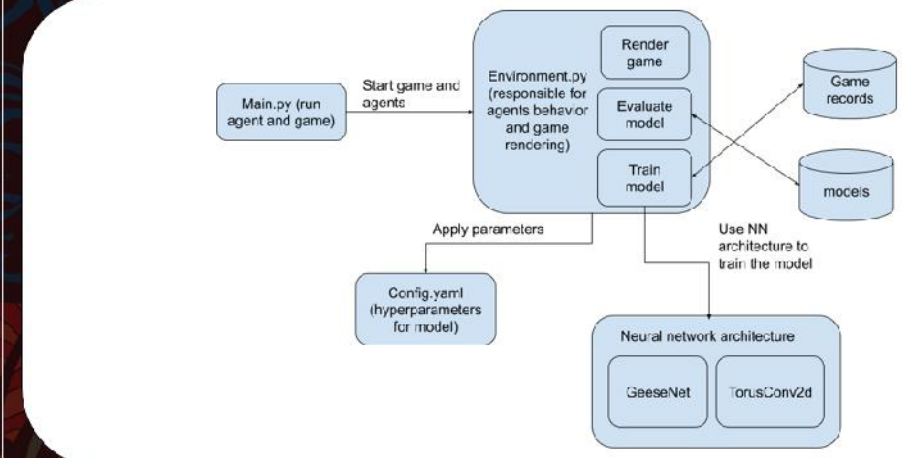
Методи забезпечення **статичного** ігрового інтелекту:

1. Створення загальних заготовлених правил поведінки
2. Використання евристик для модифікації поведінки агенту
3. Підготовка стратегії роботи агенту (наприклад: схильність на ризик)

Методи забезпечення **динамічного** ігрового інтелекту:

1. Побудова нейронної мережі
2. Тренування та оцінка агенту
3. Використання методів фільтрації малоефективних кроків алгоритму
4. Використання евристик для модифікації поведінки агенту
5. Проведення спільних ігрових партій з іншими агентам та отримання досвіду для агента

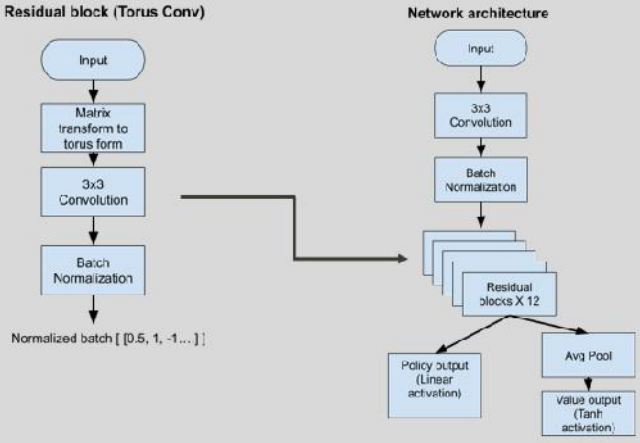
ПРОГРАМНА РЕАЛІЗАЦІЯ



СТВОРЕННЯ ІГРОВОГО АГЕНТУ

1. Побудова архітектури ШНМ, що використовується для забезпечення адаптивності ігрових агентів.
2. Попередня обробка даних перед поданням їх для навчання ШНМ.
3. Навчання ШНМ.
4. Реалізація можливості зберігання налаштувань моделі.
5. Налаштування Monte-Carlo Search Tree для фільтрації малоефективних кроків агенту.
6. Візуалізація ефективності роботи ігрових агентів.
7. Оцінка ефективності роботи застосунку.

АРХІТЕКТУРА НЕЙРОННОЇ МЕРЕЖІ

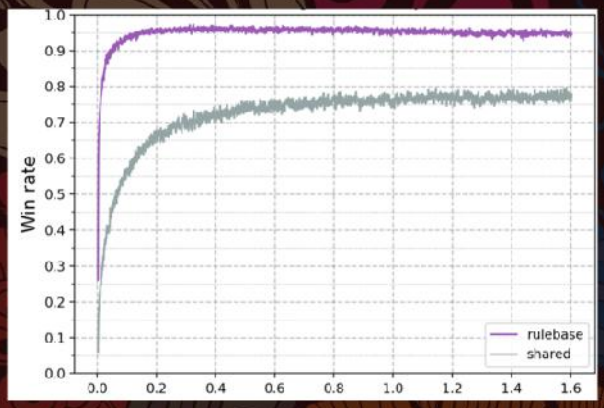


АДАПТАЦІЯ ІГРОВОГО АГЕНТУ

На графіку представлена залежність кількості перемог від кількості попередньо зіграних партій проти статичного та динамічного агентів, що демонструє врахування розробленим агентом попереднього досвіду.

Rulebase-агент представляє собою статичний ігровий ШІ, який націлений збір фруктів.

Shared модель представляє собою адаптивний ігровий агент, побудований з використанням генетичного алгоритму.



ПОРІВНЯННЯ ЗІ СТАТИЧНИМИ АГЕНТАМИ ДЛЯ ІГРИ «ЗМІЙКА»

Назва суперника	Опис	Ймовірність перемоги Smart Geese
simple_toward	Простий агент, який тільки націлений збір фруктів	96%
greedy	Жадібний агент	91%
risk_averse_greedy	Жадібний агент, який не схильний до ризику (позначає всі клітини, які можуть бути небезпечними на наступному кроці, як перешкоди)	94%
simple_bfs	Агент, що має великий список евристик для уникнення перешкод	95%
crazy_goose	Жадібний агент, який не схильний до ризику (позначає всі клітини, які можуть бути небезпечними на наступному кроці, як перешкоди) та має великий список евристик для уникнення перешкод	82%

ПОРІВНЯННЯ З АДАПТИВНИМИ АГЕНТАМИ ДЛЯ ІГРИ «ЗМІЙКА»

Назва суперника	Опис	Ймовірність перемоги Smart Geese
genetic_agent	Ігровий штучний інтелект на основі генетичних алгоритмів	67%
inclined_risk_agent	Ігровий штучний інтелект на основі жадібної політики зі схильністю використовувати ризик та нейронної мережі на основі згорткових шарів	56%
risk_averse_greedy	Ігровий штучний інтелект на основі жадної політики та нейронної мережі на основі згорткових шарів	60%

ДЕМОНСТРАЦІЯ ПОВЕДІНКИ АГЕНТІВ

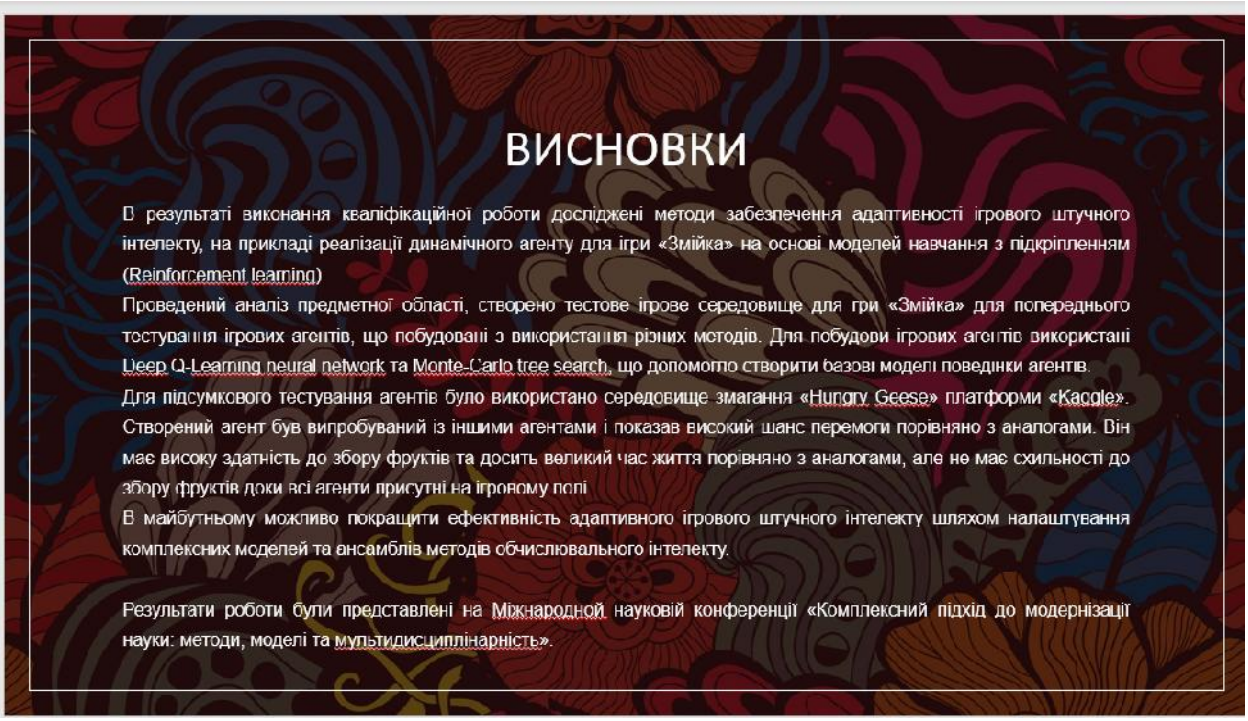
Представлені агенти побудовані на основі таких підходів:

Зелений – жадібна політика та ШНМ на основі згорткових шарів.

Синій – жадібна політика зі схильністю використовувати ризик та ШНМ на основі згорткових шарів.

Білий – генетичний алгоритм.

Червоний – агент створений в ході кваліфікаційної роботи


ВИСНОВКИ
В результаті виконання кваліфікаційної роботи досліджені методи забезпечення адаптивності ігрового штучного інтелекту, на прикладі реалізації динамічного агенту для ігри «Змійка» на основі моделей навчання з підкріпленням (Reinforcement learning)
Проведений аналіз предметної області, створено тестове ігрове середовище для гри «Змійка» для попереднього тестування ігрових агентів, що побудовані з використанням різних методів. Для побудови ігрових агентів використані Deep Q-Learning neural network та Monte-Carlo tree search, що допомогло створити базові моделі поведінки агентів.
Для підсумкового тестування агентів було використано середовище змагання «Hungry Geese» платформи «Kaggle». Створений агент був випробуваний із іншими агентами і показав високий шанс перемоги порівняно з аналогами. Він має високу здатність до збору фруктів та досить великий час життя порівняно з аналогами, але не має схильності до збору фруктів доки всі агенти присутні на ігровому полі
В майбутньому можливо покращити ефективність адаптивного ігрового штучного інтелекту шляхом налаштування комплексних моделей та ансамблів методів обчислювального інтелекту.
Результати роботи були представлені на Міжнародній науковій конференції «Комплексний підхід до модернізації науки: методи, моделі та мультидисциплінарність».

ДОДАТОК Б

Вихідний код розроблених програмних засобів

```
import random
import itertools
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
# You need to install kaggle_environments, requests
from kaggle_environments import make
from ...environment import BaseEnvironment

class TorusConv2d(nn.Module):
    def __init__(self, input_dim, output_dim, kernel_size, bn):
        super().__init__()
        self.edge_size = (kernel_size[0] // 2, kernel_size[1] // 2)
        self.conv       =       nn.Conv2d(input_dim,       output_dim,
kernel_size=kernel_size)
        self.bn = nn.BatchNorm2d(output_dim) if bn else None
    def forward(self, x):
        h      =      torch.cat([x[:,:,:,-self.edge_size[1]:],      x,
x[:,:,:,:self.edge_size[1]]], dim=3)
        h       =       torch.cat([h[:,:,-self.edge_size[0]:],       h,
h[:,:,:self.edge_size[0]]], dim=2)
        h = self.conv(h)
        h = self.bn(h) if self.bn is not None else h
        return h

class GeeseNet(nn.Module):
    def __init__(self):
        super().__init__()
        layers, filters = 12, 32
        self.conv0 = TorusConv2d(17, filters, (3, 3), True)
        self.blocks = nn.ModuleList([TorusConv2d(filters, filters, (3,
3), True) for _ in range(layers)])
        self.head_p = nn.Linear(filters, 4, bias=False)
        self.head_v = nn.Linear(filters * 2, 1, bias=False)
    def forward(self, x, _=None):
        h = F.relu_(self.conv0(x))
        for block in self.blocks:
            h = F.relu_(h + block(h))
        h_head = (h * x[:,:1]).view(h.size(0), h.size(1), -1).sum(-1)
        h_avg = h.view(h.size(0), h.size(1), -1).mean(-1)
        p = self.head_p(h_head)
        v = torch.tanh(self.head_v(torch.cat([h_head, h_avg], 1)))
        return {'policy': p, 'value': v}

class Environment(BaseEnvironment):
    ACTION = ['NORTH', 'SOUTH', 'WEST', 'EAST']
    DIRECTION = [[-1, 0], [1, 0], [0, -1], [0, 1]]
    NUM_AGENTS = 4

    def __init__(self, args={}):
        super().__init__()
        self.env = make("hungry_geese")
        self.reset()
    def reset(self, args={}):
```

```
            obs = self.env.reset(num_agents=self.NUM_AGENTS)
            self.update((obs, {}), True)
        def update(self, info, reset):
            obs, last_actions = info
            if reset:
                self.obs_list = []
            self.obs_list.append(obs)
            self.last_actions = last_actions
        def action2str(self, a, player=None):
            return self.ACTION[a]
        def str2action(self, s, player=None):
            return self.ACTION.index(s)
        def direction(self, pos_from, pos_to):
            if pos_from is None or pos_to is None:
                return None
            x, y = pos_from // 11, pos_from % 11
            for i, d in enumerate(self.DIRECTION):
                nx, ny = (x + d[0]) % 7, (y + d[1]) % 11
                if nx * 11 + ny == pos_to:
                    return i
            return None

        def __str__(self):
            # output state
            obs = self.obs_list[-1][0]['observation']
            colors = ['\033[33m', '\033[34m', '\033[32m', '\033[31m']
            color_end = '\033[0m'

            def check_cell(pos):
                for i, geese in enumerate(obs['geese']):
                    if pos in geese:
                        if pos == geese[0]:
                            return i, 'h'
                        if pos == geese[-1]:
                            return i, 't'
                        index = geese.index(pos)
                        pos_prev = geese[index - 1] if index > 0 else None
                        pos_next = geese[index + 1] if index < len(geese) -
1 else None
                        directions    =    [self.direction(pos,    pos_prev),
self.direction(pos, pos_next)]
                        return i, directions
                if pos in obs['food']:
                    return 'f'
                return None

            def cell_string(cell):
                if cell is None:
                    return '.'
                elif cell == 'f':
                    return 'f'
                else:
                    index, directions = cell
                    if directions == 'h':
                        return colors[index] + '@' + color_end
                    elif directions == 't':
                        return colors[index] + '*' + color_end
                    elif max(directions) < 2:
                        return colors[index] + '|' + color_end
                    elif min(directions) >= 2:
                        return colors[index] + '-' + color_end
                    else:
                        return colors[index] + '+' + color_end
```

```
            cell_status = [check_cell(pos) for pos in range(7 * 11)]

            s = 'turn %d\n' % len(self.obs_list)
            for x in range(7):
                for y in range(11):
                    pos = x * 11 + y
                    s += cell_string(cell_status[pos])
                s += '\n'
            for i, geese in enumerate(obs['geese']):
                s += colors[i] + str(len(geese) or '-') + color_end + ' '
            return s

        def step(self, actions):
            # state transition
            obs = self.env.step([self.action2str(actions.get(p, None) or 0)
for p in self.players()])
            self.update((obs, actions), False)

        def diff_info(self, _):
            return self.obs_list[-1], self.last_actions

        def turns(self):
            # players to move
            return [p for p in self.players() if self.obs_list[-
1][p]['status'] == 'ACTIVE']

        def terminal(self):
            # check whether terminal state or not
            for obs in self.obs_list[-1]:
                if obs['status'] == 'ACTIVE':
                    return False
            return True

        def outcome(self):
            # return terminal outcomes
            # 1st: 1.0 2nd: 0.33 3rd: -0.33 4th: -1.00
            rewards = {o['observation']['index']: o['reward'] for o in
self.obs_list[-1]}
            outcomes = {p: 0 for p in self.players()}
            for p, r in rewards.items():
                for pp, rr in rewards.items():
                    if p != pp:
                        if r > rr:
                            outcomes[p] += 1 / (self.NUM_AGENTS - 1)
                        elif r < rr:
                            outcomes[p] -= 1 / (self.NUM_AGENTS - 1)
            return outcomes

        def legal_actions(self, player):
            # return legal action list
            return list(range(len(self.ACTION)))

        def action_length(self):
            # maximum action label (it determines output size of policy
function)
            return len(self.ACTION)

        def players(self):
            return list(range(self.NUM_AGENTS))

        def rule_based_action(self, player):
            from kaggle_environments.envs.hungry_geese.hungry_geese import
Observation, Configuration, Action, GreedyAgent
            action_map = {'N': Action.NORTH, 'S': Action.SOUTH, 'W':
```

```
Action.WEST, 'E': Action.EAST}
            agent = GreedyAgent(Configuration({'rows': 7, 'columns': 11}))
            agent.last_action                                                =
action_map[self.ACTION[self.last_actions[player]][0]]    if    player    in
self.last_actions else None
            obs = {**self.obs_list[-1][0]['observation'], **self.obs_list[-
1][player]['observation']}
            action = agent(Observation(obs))
            return self.ACTION.index(action)
        def net(self):
            return GeeseNet
        def observation(self, player=None):
            if player is None:
                player = 0
            b  =  np.zeros((self.NUM_AGENTS  *  4  +  1,  7  *  11),
dtype=np.float32)
            obs = self.obs_list[-1][0]['observation']

            for p, geese in enumerate(obs['geese']):
                # head position
                for pos in geese[:1]:
                    b[0 + (p - player) % self.NUM_AGENTS, pos] = 1
                # tip position
                for pos in geese[-1:]:
                    b[4 + (p - player) % self.NUM_AGENTS, pos] = 1
                # whole position
                for pos in geese:
                    b[8 + (p - player) % self.NUM_AGENTS, pos] = 1

            # previous head position
            if len(self.obs_list) > 1:
                obs_prev = self.obs_list[-2][0]['observation']
                for p, geese in enumerate(obs_prev['geese']):
                    for pos in geese[:1]:
                        b[12 + (p - player) % self.NUM_AGENTS, pos] = 1
            # food
            for pos in obs['food']:
                b[16, pos] = 1

            return b.reshape(-1, 7, 11)

    if __name__ == '__main__':
        e = Environment()
        for _ in range(100):
            e.reset()
            while not e.terminal():
                print(e)
                actions = {p: e.legal_actions(p) for p in e.turns()}
                print([[e.action2str(a, p) for a in alist] for p, alist in
actions.items()])
                e.step({p:  random.choice(alist)  for  p,  alist  in
actions.items()})
            print(e)
            print(e.outcome())
```