МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ФРАНКА

Факультет електроніки і комп’ютерних технологій

Кафедра радіоелектронних та комп’ютерних систем

Курсова робота

«Використання генетичних алгоритмів для керування рухомими об’єктами»

Виконавець:

студент групи ФеІ-14м

Речинський О.С.

Науковий керівник:

Доц. Соколовський Б.С.

Львів – 2023

АНОТАЦІЯ

В даній роботі розглядається використання генетичних алгоритмів та багатошарового персептрона для розв’язання задачі балансування жердини у середовищі OpenAI Gym. Дана задача є складним завданням через нестабільність та безперервний простір станів. Комбінація еволюційного та нейромережевого підходів дозволяють знайти оптимальні та надійні розв’язки для даної задачі. Результати роботи доводять ефективність даної комбінації для вирішення задач керування рухомими об’єктами у просторі.

ABSTRACT

This paper considers the use of genetic algorithms and a multilayer perceptron to solve the pole balancing problem in the OpenAI Gym environment. This problem is a challenging task due to its instability and continuous state space. The combination of evolutionary and neural network approaches allows us to find optimal and reliable solutions for this problem. The results of the work prove the effectiveness of this combination for solving problems of controlling moving objects in space.

ЗМІСТ

АНОТАЦІЯ 2

ABSTRACT 2

ЗМІСТ 3

ВСТУП 5

РОЗДІЛ 1. ГЕНЕТИЧНІ АЛГОРИТМИ ТА БАГАТОШАРОВИЙ ПЕРСЕПТРОН 7

1.1. Базові принципи генетичних алгоритмів 7

1.1.1. Загальний опис генетичних алгоритмів та репрезентації особин 7

1.1.2. Функція пристосованості 8

1.1.3. Оператор відбору особин 8

1.1.4. Оператор схрещування особин 9

1.1.5. Оператор мутації 9

1.1.6. Схема роботи генетичного алгоритму 10

1.2. Штучні нейронні мережі 11

1.2.1. Штучний нейрон 11

1.2.2. Багатошаровий персептрон 12

1.2.3. Функція активації 13

1.2.4. Процес прямого поширення 13

1.2.5. Процес навчання через зворотне поширення 14

1.2.6. Використання багатошарового персептрону для задачі балансування жердини 16

РОЗДІЛ 2. ПРОЦЕС СТВОРЕННЯ РІШЕННЯ НА ОСНОВІ ГЕНЕТИЧНИХ АЛГОРИТМІВ ТА БАГАТОШАРОВОГО ПЕРСЕПТРОНУ ... 17

2.1. Опис та простір подій задачі керування рухомими об’єктами 17

2.2. Створення контролера візочком із жердиною із використанням багатошарового персептрона 18
2.3. Представлення та оцінка рішення 19
2.4. Інтеграція генетичного алгоритму 20
2.5. Використане ПЗ 21
2.5.1. Python 21
2.5.2. OpenAI Gym 21
2.5.3. Scikit-learn 22
2.5.4. DEAP 22
РОЗДІЛ 3. АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ 23
ВИСНОВКИ 27
ВИКОРИСТАНІ ДЖЕРЕЛА ТА ЛІТЕРАТУРА 28
ЛІСТИНГ ПРОЕКТУ 29

ВСТУП

Останнім часом спостерігається стрімке зростання інтересу до галузі штучного інтелекту та машинного навчання, сфери застосування яких є дуже широкими, починаючи із розумних асистентів, мовних моделей, генеративних моделей для зображення і закінчуючи системами для керування рухомими об’єктами із використанням навчання з підкріпленням. Одним із прикладів керування рухомими об’єктами є задача балансування жердини на візочку.

Дана задача є популярним методом тестування різноманітних алгоритмів, мета яких – якомога довше утримати жердину від падіння. Незважаючи на те, що на перший погляд проблема здається простою, вона представляє значні труднощі через притаманну їй нестабільність та неперервний характер простору станів. Для вирішення даної задачі можна використати різні підходи, починаючи від традиційних підходів теорії управління і закінчуючи більш складними методами навчання із глибинним підкріпленням.

В даній роботі розглядається один із перспективних підходів вирішення цієї задача – використання генетичних алгоритмів і багатошарового персептрона. Генетичні алгоритми використовують принципи природної еволюції та імітують процеси відбору, мутації та схрещування для пошуку оптимальних рішень у просторі можливих розв’язків. Багатошаровий персептрон, в свою чергу, вміє добрим чином моделювати нелінійні взаємозв’язки.

Метою даної роботи є дослідження застосування генетичних алгоритмів та багатошарового персептрона для розв’язання задачі балансування жердини. Використовуючи здатність генетичних алгоритмів для пошуку оптимальних рішень та здатність багатошарового персептрону навчатися та генералізувати дані, ми прагнемо розробити інтелектуальний агент, який здатний досягати надійного та стабільного балансування жердини. Дана робота розкриває широкі можливості синергії між генетичними алгоритмами та нейронними мережами

для більш ефективних та дієвих підходів вирішення проблеми керування рухомими об’єктами.

РОЗДІЛ 1. ГЕНЕТИЧНІ АЛГОРИТМИ ТА БАГАТОШАРОВИЙ ПЕРСЕПТРОН

1.1. Базові принципи генетичних алгоритмів

1.1.1. Загальний опис генетичних алгоритмів та репрезентації особин

Генетичний алгоритм – сімейство пошукових алгоритмів, головна ідея яких базується на принципах еволюції в природі. Імітуючи процеси природного відбору, генетичні алгоритми вміють знаходити якісні рішення у задачах пошуку, оптимізації та навчання. Аналогія з природним відбором допомагає цим алгоритмам обходити складності, які виникають при використанні стандартних алгоритмів, такі як нестабільність та неперервний простір можливих станів.

В природі усі операції генетичних алгоритмів, такі як відбір, мутація та схрещування, виконується через генотип – набір генів, які згруповані в хромосоми. Під час схрещування двох особин утворюється потомок, кожна хромосома якого несе комбінацію генів батьків. Для прикладу, хромосому можна уявити у вигляді двійкової послідовності, кожен біт якої відповідає одному гену.

Рис. 1.1.1: представлення хромосоми

Задача генетичного алгоритму – створити якомога «сильнішу» популяцію особин, що представлена у вигляді найкращих потенційних розв’язків поставленої задачі. Оскільки кожна особина представлена хромосомою, то популяція розглядається як колекція хромосом. Популяція є завжди представленням поточного покоління та еволюціонує з часом, коли поточний стан замінюється на новий.

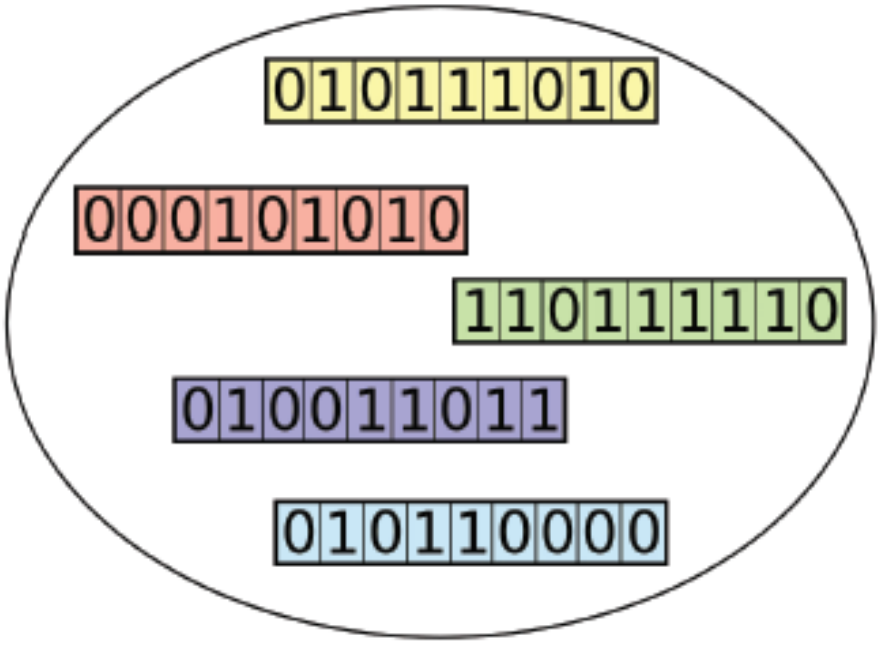

Рис. 2.1.2: популяція особин

1.1.2. Функція пристосованості

На кожній ітерації алгоритму, кожна особина оцінюється за допомогою функції пристосовуваності, оптимізацією якої ми і займаємось. Особини, для яких дана функція дає найкращу оцінку, і є найкращими розв’язками задачі та з більшою ймовірністю будуть обрані для процесу схрещування. З часом, якість розв’язків покращується, і як тільки буде досягнути критерію зупинки, найкращий результат буде збережений як кінцевий розв’язок. Роль функції пристосовуваності полягає у скеруванні генетичного алгоритму до кращих рішень, присвоюючи вищі значення пристосованості особинам, які демонструють бажані риси або поведінку.

1.1.3. Оператор відбору особин

Важливою операцією в генетичному алгоритмі є процес відбору, який визначає особин, що будуть обрані для схрещування. Відбір базується на результаті функцію пристосовуваності особини і є прямо пропорційним до ймовірності бути обраним для схрещування і передати свій генетичний матеріал наступному поколінню. Існують різноманітні варіанти процесу відбору, такі як турнірний відбір, відбір на основі колеса рулетки або ранговий відбір. Процес відбору гарантує, що особини з кращою пристосованістю мають

більший шанс бути відібраними для розмноження, таким чином зберігаючи сприятливі ознаки в популяції протягом багатьох поколінь.

1.1.4. Оператор схрещування особин

Для утворення пар нових особин використовується процес схрещування, суть якого полягає в обранні нащадків з поточного покоління та схрещуванні їх хромосом. Аналогічно із процесом відбору, існують різні процеси схрещування, кожен з яких має власні переваги та недоліки. Популярними варіантами схрещування є одно- та n-точкове схрещування, частково дзеркальне схрещування тощо.

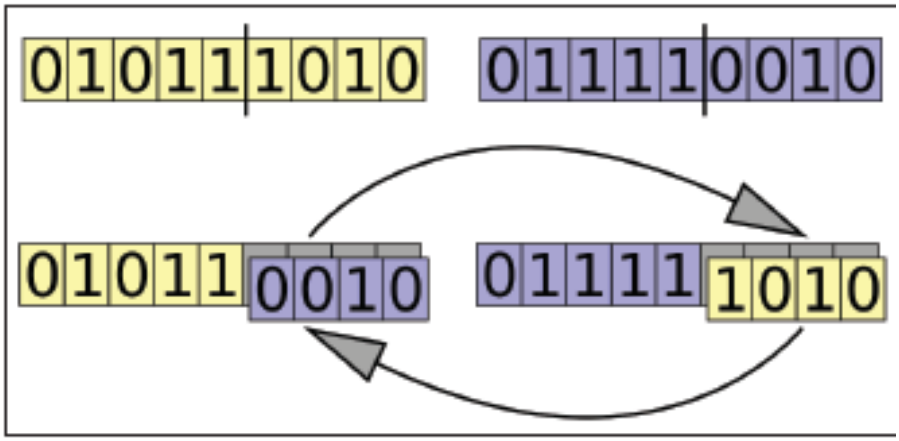

Рис. 3.1.3: процес схрещування

1.1.5. Оператор мутації

Також, із заданою ймовірністю у особини може відбутись мутація, що вносить нові комбінації генів в хромосому, тим самим сприяючи пошук розв’язку в нових, ще нерозвіданих напрямках і не дозволяє алгоритму застрягнути у локальному оптимумі.

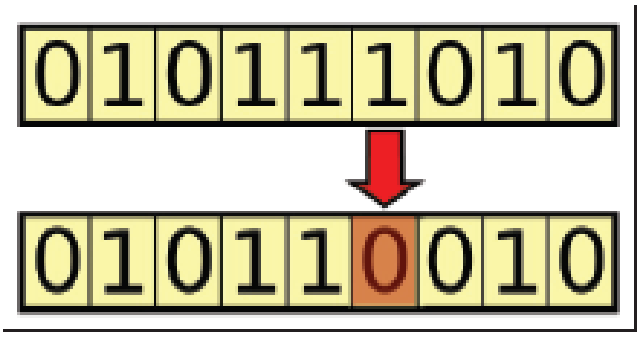

Рис. 4.1.4: процес мутації

1.1.6. Схема роботи генетичного алгоритму

Резюмуючи, принципову схему генетичного алгоритму можна зобразити наступним чином:

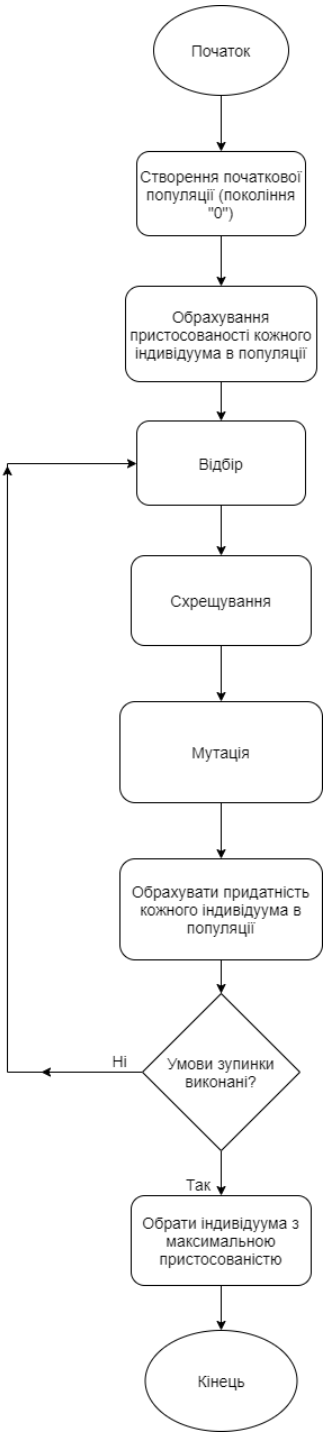

Рис. 5.1.5: принципова схема роботи генетичного алгоритму

Початкова популяція алгоритму складається із випадково обраних потенційних розв’язків (особин), формат хромосом яких має відповідати умовам та обмеженням задачі.

Умова зупинки роботи визначають, коли генетичний алгоритм повинен припинити ітерації та оголосити найкраще рішення. Найпоширенішими критеріями зупинки є досягнення максимальної кількості поколінь, досягнення бажаного порогу пристосованості або спостереження незначного поліпшення популяції протягом наступних ітерацій.

Генетичний алгоритм розвиває популяцію протягом декількох поколінь. Після проходження процесу відбору, схрещування та мутацій нове потомство замінює деяких, більш слабких, особин у поточній популяції. Розмір популяції є відносно постійним, таким чином забезпечуючи різноманітний набір потенційних рішень протягом усього процес оптимізації. Такий ітеративний підхід дозволяє генетичному алгоритму з кожним поколінням шукати якомога кращі рішення.

Використовуючи принципи відбору, схрещування та мутацій, генетичні алгоритми забезпечують ефективну та гнучку структуру оптимізації, що може бути застосована для широкого кола проблем. Їх здатність досліджувати складні пошукові простори і знаходити оптимальні рішення робить їх цінними інструментом для вирішення складних проблем, наприклад як балансування візочка із жердиною у просторі.

1.2. Штучні нейронні мережі

1.2.1. Штучний нейрон

Основною одиницею штучних нейронних мереж є нейрон (персептрон), задача якого полягає в обчисленні зваженої суму його входів та застосування нелінійної функції активації. Дана модель імітує поведінку людського нейрона, який збуджується (посилає серію імпульсів через свій вихідний канал), якщо зважена сума вхідних сигналів є більшою за певний поріг.

Модель персептрона можна легко використовувати для простих задач класифікації або регресії, якщо правильно налаштувати вагові коефіцієнти, але

зазвичай персептрони об’єднують в пошаровану та більш потужну структуру, яка називається багатошаровим персептроном.

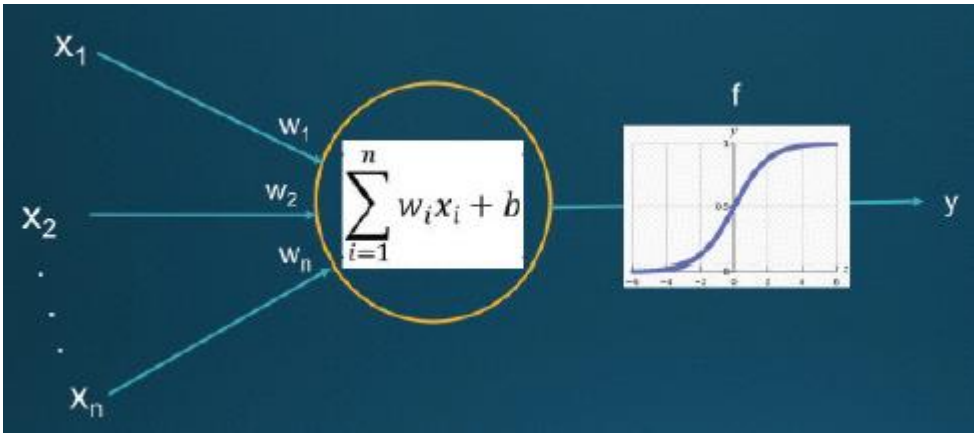

Рис. 1.2.1: схема штучного нейрона

1.2.2. Багатошаровий персептрон

Багатошаровий персептрон є узагальненням ідеї персептрону: об’єднання декількох блоків, кожен із яких реалізує персептрон. Блоки в такій структурі організовані в шари, кожен із яких з’єднаний із наступним. Базова структура багатошарового персептрона показана нижче.

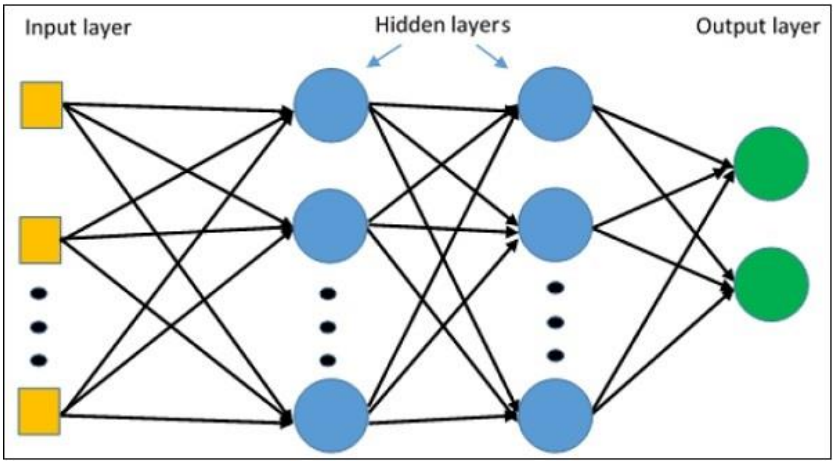

Рис. 1.2.2: структура багатошарового персептрона

Багатошаровий персептрон складається із декількох частин:

- Вхідний шар: отримує вхідні значення і з’єднує кожне із них з кожним нейроном наступного шару.
- Приховані шари: нейрони саме даних шарів виконують усі необхідні обчислення для отримання бажаних результатів. На рисунку вище

зображено тільки два прихованих шари, але їх кількість може бути значно більшою. Дані шари розташовані між вхідними та вихідними шарами і по мірі збільшення кількості прихованих шарів наша мережа стає більш глибинною і може виконувати більш складні нелінійні відображення вхідних даних.

- Вихідний шар: повертає результати обчислення. Якщо багатошаровий персептрон використовується у якості класифікатора, то кожен вихід представляє один клас, а якщо для задачі регресії, то вихідний блок тільки один і повертає неперервне значення.

1.2.3. Функція активації

Кожен нейрон застосовує функцію активації до своєї зваженої суми входів, визначаючи вихід або активацію нейрона. Найпоширенішими функціями активації є сигмоїдна функція, гіперболічна тангенціальна функція та випрямлена лінійна одиниця (ReLU). Функції активації потрібні для внесення нелінійності у мережу, що дає змогу моделювати нелінійні зв’язки між входами і виходами.

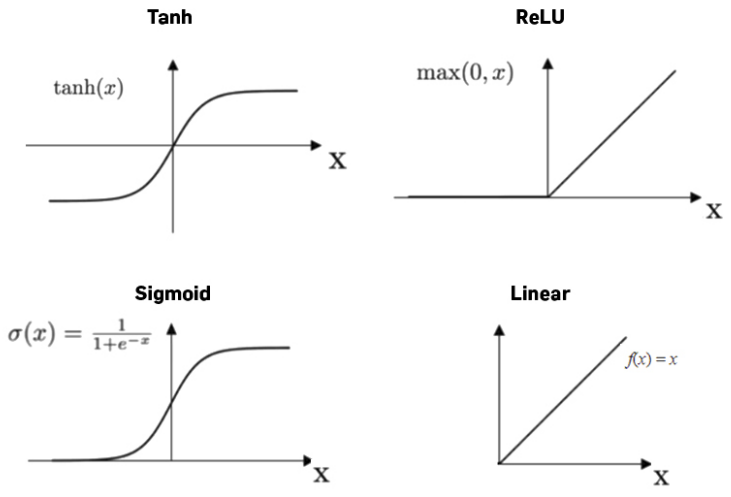

Рис. 1.2.3: популярні функції активації

1.2.4. Процес прямого поширення

Пряме поширення – це базовий процес в штучних нейронних мережах, суть якого полягає в проходженні вхідних даних через усі шари мережі та обчисленні вихідного значення.

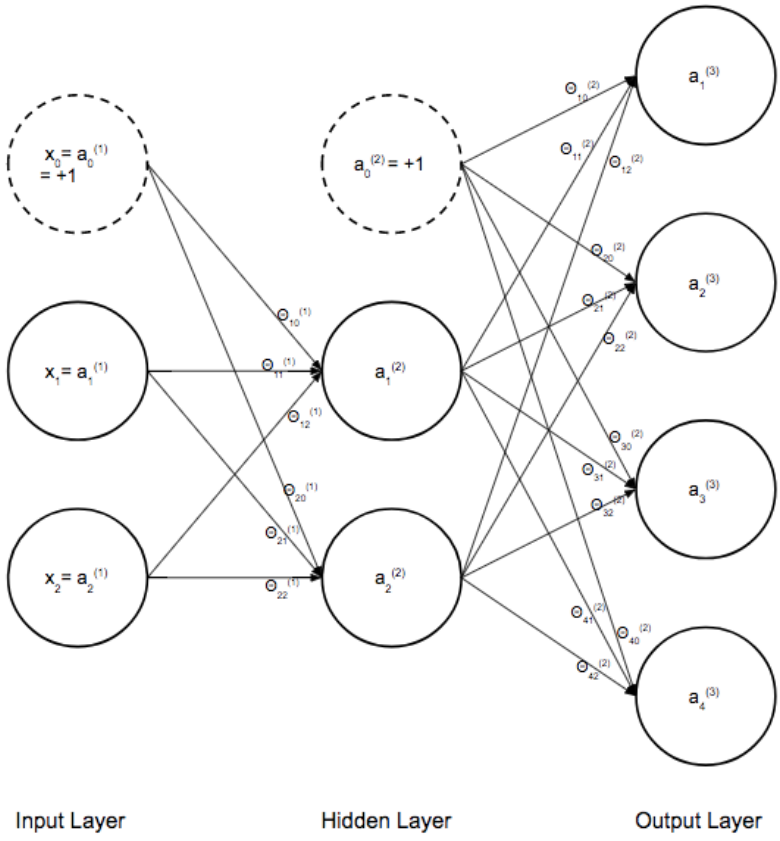

Рис. 1.2.4: схема прямого поширення

Під час даного етапу, кожен нейрон в шарі на вхід отримує вихідні значення з попереднього шару. Далі, обчислюється зважена сума і застосовується функція активації, яка і визначає вихідне значення нейрону.

Пряме поширення дає змогу трансформувати вхідні дані на змістовні вихідні представлення. Завдяки взаємопов'язаним шарам та функції активації, багатошаровий персептрон може вивчати та фіксувати складні взаємозв'язки між вхідними та вихідними даними.

1.2.5. Процес навчання через зворотне поширення

Навчання багатошарового персептрона передбачає коригування ваг зв'язків між нейронами таким чином, щоб мінімізувати розбіжність між передбаченими та «правильними» результатами. Даний підхід є стандартним для навчання штучних нейронних мереж на основі навчання із вчителем.

Процес зворотного поширення складається із двох кроків: прямий прохід та зворотній прохід. Під час прямого проходу, дані поступають на вхід до

мережі, і виконується процес прямого поширення. Отриманий результат порівнюється із бажаним, і похибка між ними оцінюється кількісно за допомогою відповідної функції втрат, наприклад середньоквадратичної або кросс-ентропійної похибки.

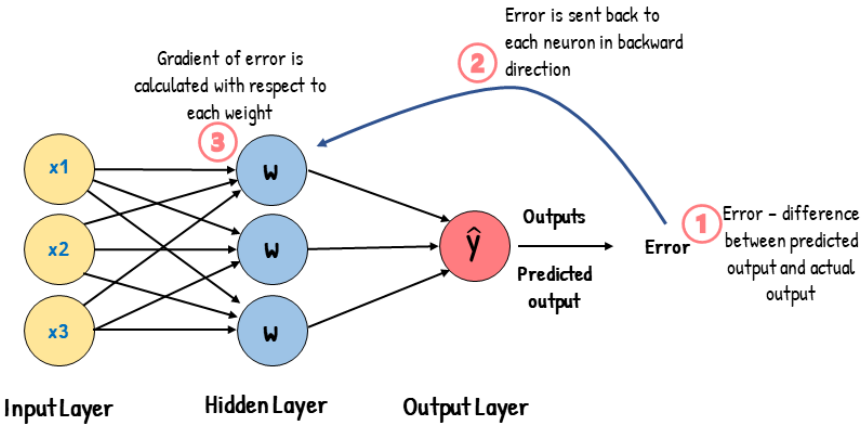

Рис. 1.2.5: схема зворотного поширення

Під час зворотного проходу, помилка поширюється назад через мережу від вихідного шару до прихованих шарів. Це відбувається шляхом обчислення градієнта помилки по відношенню до ваг кожного з’єднання. Для ефективного обчислення даних градієнтів шар за шаром використовується правило ланцюжка із математичного аналізу.

Використовуючи обчислені градієнти, ваги мережі оновлюються ітеративно за допомогою оптимізації градієнтного спуску. Ваги коригуються у напрямку, який мінімізує похибку, поступово покращуючи результати мережі. Даний процес прямого проходу, обчислення похибки, зворотного проходу та оновлення ваг повторюється протягом декількох ітерацій (епох), поки мережа не досягне бажаного рівня продуктивності.

Зворотне поширення дозволяє багатошаровому персептрону навчатися та коригувати свої ваги на основі похибки між прогнозованими та «правильними» результатами. Шляхом ітеративного коригування ваг, мережа має змогу поступово покращувати свою здатність генералізувати дані та робити точніші прогнози на нових даних.

1.2.6. Використання багатошарового персептрону для задачі балансування жердини

Багатошарові персептрони знайшли значне застосування в системах керування рухомими об’єктами, де їх можна використати для моделювання складної динаміки та стратегії керування. У контексті задачі балансування жердини на візочку, багатошаровий персептрон може навчитись оптимальній стратегії керуванні для якомога довшого балансування, зіставляючи простір станів із відповідними керуючими діями.

Гнучкість, можливість нелінійного моделювання та універсальні апроксимаційні властивості багатошарового персептрона роблять його цінним інструментом для вирішення подібних задач.

РОЗДІЛ 2. ПРОЦЕС СТВОРЕННЯ РІШЕННЯ НА ОСНОВІ ГЕНЕТИЧНИХ АЛГОРИТМІВ ТА БАГАТОШАРОВОГО ПЕРСЕПТРОНУ

2.1. Опис та простір подій задачі керування рухомими об’єктами

Наша задача полягає у балансуванні жердини, що закріплена шарнірно на візочку, який рухається вліво та вправо по доріжці. Для утримання жердини у вертикальному положенні потрібно прикласти силу, скеровану вліво чи вправо.

Жердина поводить себе як зворотній маятник, в початковий момент відхиляючись від вертикального положення на невеликий кут, як показано на рисунку нижче.

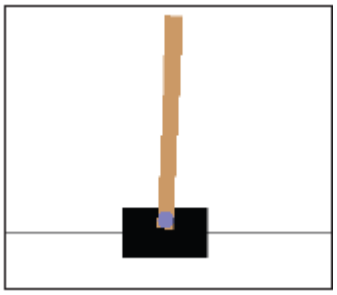

Рис. 2.1: початкове положення жердини на візочку

Мета даної задачі – не дати жердині впасти якомога довше, а точніше на протязі 500 часових кроків. За кожен крок, під час якого жердина не падає, нараховується винагорода, максимальне значення якої складає 500. Епізод закінчується достроково в одному із трьох випадків:

- Жердина відхилилась від вертикального положення на кут більший за 12°;
- Візочок від’їхав від центру більше ніж на 2.4 одиниці (центр візочка досягнув краю доріжки);
- Досягнута максимальна кількість часових кроків.

В даному середовищі можливі дві дії:

- 0: штовхнути візочок вліво;

- 1: штовхнути візочок вправо.

Також ми маємо простір спостережень, який містить наступну інформацію про візочок із жердиною:

- Положення візочка (від -4.8 до 4.8);
- Швидкість візочка (від мінус нескінченності до нескінченності);
- Кут нахилу жердини (від -24° до 24°);
- Швидкість кінчика жердини (від мінус нескінченності до нескінченності)

Для прикладу, ми можемо отримати наступне спостереження *[0.357218, 0.382424, -0.002333, -0.391618]*. Ці значення ми будемо використовувати як вхідні дані на кожному часовому кроці, на основі яких і визначається потрібна дія.

2.2. Створення контролера візочком із жердиною із використанням багатошарового персептрона

Для вирішення задачі балансування жердини ми маємо динамічним чином реагувати на зміни в навколишньому середовищі. Наприклад, якщо жердина відхиляється в одному напрямі, то, скоріше за все, нам треба штовхнути візочок в цьому напрямі, але перестати штовхати як тільки жердина почне вирівнюватись. Таким чином, на даному етапу цю задачу можна розглядати як навчання контролера балансування жердини шляхом відображення чотирьох вхідних змінних – положення візочка, швидкості візочка, кута нахилу жердину, швидкості жердини.

Для цього можна використати багатошаровий персептрон, який дозволяє реалізувати складні відображення між входами та виходами. Для цього ми маємо налаштувати внутрішні параметри мережі – вагові коефіцієнти та зсуви нейронів. Структура багатошарового персептрону буде мати один прихований шар, який складається із 4 нейронів. Вхідний шар також має 4 нейрони, по одному для кожного вхідного значення, а вихідний шар складається всього із 1

нейрона, оскільки ми маємо лише одне вихідне значення – дію, яку варто виконати.

Зазвичай, нейронні мережі навчаються за допомогою підходу навчання із вчителем, що передбачає наявність навчального набору даних, який містить входи та правильні виходи для них, а мережа навчалась таким чином, щоб вхід відображався на відповідний йому вихід. Але у нашому випадку таких навчальних даних немає, що означає що замість стандартних алгоритмів навчання ми маємо використати підхід, який дозволить знайти нам оптимальні параметри мережі на основі результатів епізодів взаємодії із середовищем. Саме для такого випадку і потрібні генетичні алгоритми, які дозволяють знайти оптимальний набір параметрів, що дають найкращий результат, за умови що ми правильно оцінюємо та порівнюємо результати.

2.3. Представлення та оцінка рішення

Оскільки ми будемо керувати візочком за допомогою багатошарового персептрону, то набір параметрів для оптимізації складається із вагових коефіцієнтів та зсувів мережі.

- Вхідний шар: даний шар не бере участі у відображенні входів на виході мережею. Його задача просто отримати вхідні дані та передати їх кожному нейрону наступного шару. Тому ніяких параметрів тут немає.
- Прихований шар: кожен нейрон даного шару пов’язаний із кожним входом мережі, тому нам треба чотири вагових коефіцієнти та один зсув.
- Вихідний шар: єдиний нейрон даного шару пов’язаний із кожним нейроном прихованого шару, тому нам треба чотири зсуви та один зсув.

Сумарно, нам треба знайти 20 вагових коефіцієнтів та 5 зсувів, тому потенційне рішення можна представити у вигляді списку із 25 чисел з плаваючою комою.

2.4. Інтеграція генетичного алгоритму

Оскільки для представлення рішення ми обрали список, що складається із вагових коефіцієнтів та зсував мережі, то ми можемо задати межі шуканих значень у діапазоні [-1.0; 1.0]:

```
BOUNDS_LOW, BOUNDS_HIGH = -1.0, 1.0
```

Тепер побудуємо правильним чином функцію пристосовуваності. Оскільки наша мета полягає у максимізації оцінки – часу, протягом якого ми можемо утримувати жердину в рівновазі, то ми маємо максимізувати нашу функцію пристосовуваності:

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

Також створюємо допоміжну функцію для генерації випадкових чисел із плаваючою комою, що будуть рівномірно розподілені в заданому діапазоні:

```
def random_float(low, up):
    return [random.uniform(l, u) for l, u in zip([low] * NUM_OF_PARAMS, [up] * NUM_OF_PARAMS)]
```

Створимо оператор, який буде повертати список випадкових чисел із заданого діапазону:

```
toolbox.register("attr_float", random_float, BOUNDS_LOW, BOUNDS_HIGH)
```

Визначимо оператор, який буде заповнювати особину за допомогою попереднього оператора:

```
toolbox.register("individualCreator",tools.initIterate, creator.Individual, toolbox.attr_float)
```

Тепер ми маємо визначити безпосередню реалізацію функцію пристосованості для особини:

```
def score(individual):
    return cartPole.get_score(individual),

toolbox.register("evaluate", score)
```

Тепер налаштуємо наші генетичні оператори.

- Відбір: використаємо турнірний відбір із розміром 2;
- Схрещування: використаємо бінарне схрещування із ймовірністю 0.9;

- Мутації: використаємо поліноміальну мутацію із ймовірністю 0.5.

В результати, отримуємо наступну реалізацію:

```
toolbox.register("select", tools.selTournament, tournsize=2)

toolbox.register("mate", tools.cxSimulatedBinaryBounded, low=BOUNDS_LOW, up=BOUNDS_HIGH, eta=CROWDING_FACTOR)

toolbox.register("mutate", tools.mutPolynomialBounded, low=BOUNDS_LOW,up=BOUNDS_HIGH, eta=CROWDING_FACTOR, indpb=1.0/NUM_OF_PARAMS)
```

Що стосується параметрів самого алгоритму, то ми будемо використовувати елліністичний підхід, тобто будемо без змін копіювати кращих на даний момент особин із залу слави в наступне покоління.

- Кількість поколінь: 10;
- Розмір покоління: 20;
- Розмір залу слави: 3.

Запуск процесу генетичного алгоритму відбувається таким чином:

```
population, logbook = elitism.eaSimpleWithElitism(population, toolbox, cxpb=P_CROSSOVER, mutpb=P_MUTATION, ngen=MAX_GENERATIONS, stats=stats, halloffame=hof, verbose=True)
```

2.5. Використане ПЗ

2.5.1. Python

Python – це популярна мова програмування, стандартні можливості якої дозволяють із легкістю закривати потреби розробників та науковців. Завдяки широкий популярності, дана мова має велику кількість різноманітних додаткових пакетів для різних сфер роботи, включаючи науку роботи із даними, машинне навчання та еволюційні алгоритми.

2.5.2. OpenAI Gym

OpenAI Gym – бібліотека із відкритим сирцевим кодом, яка пропонує стандартизований набір задач навчання із підкріпленням, а також комплект інструментів для розробки та порівняння алгоритмів їх вирішення. Дана

бібліотека містить великий набір середовищ із спільним інтерфейсом, який дозволяє тестувати свої підходи до вирішення задач.

2.5.3. Scikit-learn

Scikit-learn – бібліотека із відкритим сирцевим кодом, яка надає широкий вибір інструментів для аналізу даних та комфортної роботи із ними. Вона містить у собі різноманітні модулі для попередньої обробки даних. Побудови моделей машинного навчання, оптимізації параметрів тощо.

2.5.4. DEAP

DEAP – бібліотека, яка призначена для роботи із еволюційними алгоритмами, пропонуючи гнучкий та модульний дизайн для імплементації власних рішень на основі еволюційного навчання. Бібліотека надає широкий спектр інструментів та функцій для створення та керування роботою алгоритму.

РОЗДІЛ 3. АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

Для зручності, усі необхідні та обрані параметри алгоритму та простори подій та спостережень подані у таблицях нижче.

Таблиця 3.1.

Простір можливих дій для візочка із жердиною

Значення	Дія
0	Штовхнути візочок вліво
1	Штовхнути візочок вправо

Таблиця 3.2.

Простір спостережень для візочка із жердиною

Спостереження	Мінімальне значення	Максимальне значення
Положення візочка	-4.8	4.8
Швидкість візочка	-inf	inf
Кут нахилу жердини	-24°	24°
Швидкість кінчика жердини	-inf	inf

Таблиця 3.3.

Параметри багатошарового персептрона

Параметр	Значення
Кількість прихованих шарів	1
Кількість нейронів у вхідному шарі	4
Кількість нейронів у прихованому шарі	4
Кількість нейронів у вихідному шарі	1

Таблиця 3.4.

Параметри генетичного алгоритму

Параметр	Значення
Діапазон шуканих значень	[-1.0; 1.0]
Вид відбору	Турнірний відбір із розміром 2
Вид схрещування	Бінарне схрещування
Ймовірність схрещування	0.9
Вид мутації	Поліноміальна мутація
Ймовірність мутації	0.5
Кількість поколінь	10
Розмір покоління	20

Розмір залу слави	3

Перевіримо якість нашого підходу, спробувавши вирішити нашу задачу із заданими вище параметрами і отримуємо наступні результати.

```
gen     nevals  max     avg
0       20      58      15.75
1       15      58      18.25
2       16      58      23.55
3       16      137     42.55
4       17      151     46.5
5       16      219     63
6       16      334     89.85
7       16      500     135.35
8       15      500     155.3
9       16      500     214.85
10      14      500     294.55

Best Solution =  [-0.6026096180517646, -0.8061532496011057, -0.13789763518724496, -0.15284275396015845, -0.06595066392665005, -0.4472736578610963, 0
.3361958592370396, 0.945527405068349, -0.7831307016301179, -0.1465479154632064, 0.40103067929670916, 0.716581094539382, -0.5168240536636348, -0.6195
82183118377, -0.2527326862806113, 0.6369839621535089, -0.4452275174449332, 0.3888302998571858, -0.719325681961743, 0.8630817580210459, 0.72825339890
11432, 0.44874825797859236, -0.9007919901835112, 0.5317978144704925, -0.28220288364087043]
Best Score =  500.0
```

Рис. 3.1: отримані результати

Із даних результатів видно, що наш алгоритм успішно справився із поставленою задачею, оскільки є особини, що досягнули максимального значення нагороди. Помітно, що вже після шостого покоління найкраща особина мала максимальну оцінку, тобто жердина була в стані рівноваги весь епізод. Також за середнім значенням функції пристосованості можна помітити, наскільки сильно змінюється якість особин у поколінні.

Змінивши параметри, для прикладу, ймовірність схрещування = 0.7 та ймовірність мутації = 0.4, можна отримати цікаві результати.

```
gen     nevals  max     avg
0       20      58      15.75
1       14      58      18.25
2       15      100     32.35
3       13      150     52.25
4       13      164     79.5
5       16      500     131.15
6       10      500     162.75
7       12      500     209
8       16      500     247.9
9       16      500     242.35
10      14      500     347.85

Best Solution =  [0.6968069485215568, 0.9365914325389612, -0.12379983217099189, 0.03516567313712485, -0.8105944635405489, -0.4640776383867059, 0.436
4466665480038, -0.42077911956476877, -0.539770534806846, -0.23382545422112214, -0.40901023809799264, -0.3878067993088588, -0.5471052671058632, 0.634
3856162535868, 0.9436612052644194, -0.9487243263437184, -0.508514993554844, 0.32978935318609015, -0.9701880730762769, -0.7571362588888054, 0.7148160
494778151, 0.14208618665056894, -0.1017587267345562, 0.5110238206566107, 0.6757136915260618]
Best Score =  500.0
```

Рис. 3.2: отримані результати при змінених параметрах

Як видно із середнього значення функція пристосованості для покоління, генетичний алгоритм створював більш «сильні» популяції раніше, і особина із максимальним значенням винагороди з’явилась вже у 5 поколінні. Нюанс

полягає у тому, що через це процес може бути менш стабільним, що видно по результатах 9 покоління, де середнє значення винагороди впало. Для даної задачі це не є критичним, але за більш складної проблеми і більшого розміру пошукового простору ми могли би не отримати оптимальний розв’язок, тому завжди потрібно знаходити баланс.

Також є сенс протестувати знайдене рішення окремо, щоб переконатись що ми отримуємо бажані результати.

```
500 : --------------------------
action =  1
observation =  [-2.1041386  -0.36272293 -0.01237118 -0.31632528]
reward =  1.0
totalReward =  500.0
done =  False
truncated =  True
```

Рис. 3.3: тестування отриманого розв’язку

Видно, що знайдений розв’язок чудово працює і досягає максимального значення винагороди, що підтверджує доцільність використання такого підходу для вирішення задачі керування рухомими об’єктами у просторі.

ВИСНОВКИ

В даній роботі було досліджено застосування генетичних алгоритмів та багатошарових персептронів для розв’язання задачі керування рухомими об’єктами у просторі, а саме балансуванні жердини на візочку. Поєднавши можливості пошуку генетичного алгоритму та можливості апроксимації багатошарового персептрону, було створення рішення, яке чудовим чином розв’язує поставлену задачу.

Завдяки інтеграції генетичних алгоритмів, ми змогли застосувати еволюційних підхід до пошуку простору розв’язків для оптимальних стратегій керування. Схема кодування, функція пристосованості та генетичні оператори дозволили еволюціонувати індивідуумам з дедалі кращими показниками протягом кількох поколінь.

Використання багатошарового персептрону дозволили змоделювати та апроксимувати оптимальну політику керування об’єктом. Процес прямого поширення перетворив спостереження у керуючі дії, тоді як зворотне поширення дозволили більш точним чином налаштувати ваги алгоритму для покращення його ефективності.

Поєднавши дані алгоритми, вдалось створити потужний механізм, який є перспективним рішенням для вирішення більш складних задач керування.

Загалом, дана робота продемонструвала потенціал еволюційних алгоритмів у вирішенні проблем керування об’єктами у просторі. Знання та ідеї, отримані в результаті цієї роботи, можуть бути поширені на інші області та проблеми, які потребують інтелектуального керування та правильної стратегії прийняття рішень.

ВИКОРИСТАНІ ДЖЕРЕЛА ТА ЛІТЕРАТУРА

1) Документація пакету DEAP [онлайн-джерело]. – Режим доступу: https://deap.readthedocs.io/en/master/
2) Документація пакету Scikit-learn [онлайн-джерело]. – Режим доступу: https://scikit-learn.org/stable/
3) Документація пакету OpenAI Gym [онлайн-джерело]. – Режим доступу: https://www.gymlibrary.dev/
4) Frances Buontempo. “Genetic Algorithms and Machine Learning for Programmers”. The Pragmatic Programmers, Jan 2019, 209p.
5) Agoston E. Eiben, J.E. Smith. “Introduction to Evolutionary Computing”, 2010.
6) Стаття “Introduction to Genetic Algorithm” [онлайн-джерело]. – Режим доступу: https://medium.com/geekculture/introduction-to-genetic-algorithm-d417119040b7
7) Jason Brownlee. Crash Course on MLP NN [онлайн-джерело]. – Режим доступу: https://machinelearningmastery.com/neural-networks-crash-course/

ЛІСТИНГ ПРОЕКТУ

Головний файл pole_stabilization.py:

```
from deap import base
from deap import creator
from deap import tools

import random
import numpy

import cart_pole
import elitism


# Genetic Algorithm constants
POPULATION_SIZE = 20
P_CROSSOVER = 0.9
P_MUTATION = 0.5
MAX_GENERATIONS = 10
HALL_OF_FAME_SIZE = 3
CROWDING_FACTOR = 10.0

# Random seed:
RANDOM_SEED = 42
random.seed(RANDOM_SEED)

# Create the cart pole object
cartPole = cart_pole.CartPole(RANDOM_SEED)
NUM_OF_PARAMS = len(cartPole)

# Weight and bias values of MLP are bound between -1 and 1:
BOUNDS_LOW, BOUNDS_HIGH = -1.0, 1.0

toolbox = base.Toolbox()

# Define a maximizing fitness strategy:
creator.create("FitnessMax", base.Fitness, weights=(1.0,))

# Create the Individual class
creator.create("Individual", list, fitness=creator.FitnessMax)

# Helper function for creating random float numbers uniformly distributed in range
def random_float(low, up):
    return [random.uniform(l, u) for l, u in zip([low] * NUM_OF_PARAMS, [up] *
NUM_OF_PARAMS)]

# Define an operator that randomly returns a float in the desired range:
toolbox.register("attr_float", random_float, BOUNDS_LOW, BOUNDS_HIGH)

# Create an operator that fills up an Individual instance:
```

```python
toolbox.register("individualCreator",
                 tools.initIterate,
                 creator.Individual,
                 toolbox.attr_float)

# Create an operator that generates a list of individuals:
toolbox.register("populationCreator",
                 tools.initRepeat,
                 list,
                 toolbox.individualCreator)


# fitness calculation using the CrtPole class:
def score(individual):
    return cartPole.get_score(individual),


toolbox.register("evaluate", score)


# genetic operators:
toolbox.register("select", tools.selTournament, tournsize=2)

toolbox.register("mate",
                 tools.cxSimulatedBinaryBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR)

toolbox.register("mutate",
                 tools.mutPolynomialBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR,
                 indpb=1.0/NUM_OF_PARAMS)


# Genetic Algorithm flow:
def main():

    # create initial population (generation 0):
    population = toolbox.populationCreator(n=POPULATION_SIZE)

    # prepare the statistics object:
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("max", numpy.max)
    stats.register("avg", numpy.mean)

    # define the hall-of-fame object:
    hof = tools.HallOfFame(HALL_OF_FAME_SIZE)
```

```
    # perform the Genetic Algorithm flow with hof feature added:
    population, logbook = elitism.eaSimpleWithElitism(population,
                                                      toolbox,
                                                      cxpb=P_CROSSOVER,
                                                      mutpb=P_MUTATION,
                                                      ngen=MAX_GENERATIONS,
                                                      stats=stats,
                                                      halloffame=hof,
                                                      verbose=True)

    # print best solution found:
    best = hof.items[0]
    print()
    print("Best Solution = ", best)
    print("Best Score = ", best.fitness.values[0])
    print()

    # save best solution for a replay:
    cartPole.save_params(best)

    # find average score of 100 episodes using the best solution found:
    print("Running 20 episodes using the best solution...")
    scores = []
    for test in range(20):
        scores.append(cart_pole.CartPole().get_score(best))
    print("scores = ", scores)
    print("Avg. score = ", sum(scores) / len(scores))


if __name__ == "__main__":
    main()
```

Файл із реалізацією генетичного алгоритму elitism.py:

```
from deap import tools
from deap import algorithms

def eaSimpleWithElitism(population, toolbox, cxpb, mutpb, ngen, stats=None,
             halloffame=None, verbose=__debug__):
    """This algorithm is similar to DEAP eaSimple() algorithm, with the
modification that
    halloffame is used to implement an elitism mechanism. The individuals contained
in the
    halloffame are directly injected into the next generation and are not subject
to the
    genetic operators of selection, crossover and mutation.
    """
    logbook = tools.Logbook()
    logbook.header = ['gen', 'nevals'] + (stats.fields if stats else [])
```

```python
    # Evaluate the individuals with an invalid fitness
    invalid_ind = [ind for ind in population if not ind.fitness.valid]
    fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
    for ind, fit in zip(invalid_ind, fitnesses):
        ind.fitness.values = fit

    if halloffame is None:
        raise ValueError("halloffame parameter must not be empty!")

    halloffame.update(population)
    hof_size = len(halloffame.items) if halloffame.items else 0

    record = stats.compile(population) if stats else {}
    logbook.record(gen=0, nevals=len(invalid_ind), **record)
    if verbose:
        print(logbook.stream)

    # Begin the generational process
    for gen in range(1, ngen + 1):

        # Select the next generation individuals
        offspring = toolbox.select(population, len(population) - hof_size)

        # Vary the pool of individuals
        offspring = algorithms.varAnd(offspring, toolbox, cxpb, mutpb)

        # Evaluate the individuals with an invalid fitness
        invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
        fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit

        # add the best back to population:
        offspring.extend(halloffame.items)

        # Update the hall of fame with the generated individuals
        halloffame.update(offspring)

        # Replace the current population by the offspring
        population[:] = offspring

        # Append the current generation statistics to the logbook
        record = stats.compile(population) if stats else {}
        logbook.record(gen=gen, nevals=len(invalid_ind), **record)
        if verbose:
            print(logbook.stream)

    return population, logbook
```

Файл із реалізацією середовища тестування та багатошарового персептрону cart_pole.py:

```
import gym
import time

import numpy as np
import pickle

from sklearn.neural_network import MLPRegressor

from sklearn.exceptions import ConvergenceWarning
from sklearn.utils._testing import ignore_warnings

INPUTS = 4
HIDDEN_LAYER = 4
OUTPUTS = 1

class CartPole:

    def __init__(self, randomSeed=None):

        self.env = gym.make('CartPole-v1', render_mode="human")

        if randomSeed is not None:
            self.env.reset(seed=randomSeed)

    def __len__(self):
        return INPUTS * HIDDEN_LAYER + HIDDEN_LAYER * OUTPUTS + HIDDEN_LAYER +
OUTPUTS

    @ignore_warnings(category=ConvergenceWarning)
    def init_mlp(self, netParams):
        """
        initializes a MultiLayer Perceptron (MLP) Regressor with the desired
network architecture (layers)
        and network parameters (weights and biases).
        :param netParams: a list of floats representing the network parameters
(weights and biases) of the MLP
        :return: initialized MLP Regressor
        """

        # create the initial MLP:
        mlp = MLPRegressor(hidden_layer_sizes=(HIDDEN_LAYER,), max_iter=1)

        # This will initialize input and output layers, and nodes weights and
biases:
        # we are not otherwise interested in training the MLP here, hence the
settings max_iter=1 above
```

```
        mlp.fit(np.random.uniform(low=-1, high=1, size=INPUTS).reshape(1, -1),
np.ones(OUTPUTS))

        # weights are represented as a list of 2 ndarrays:
        # - hidden layer weights: INPUTS x HIDDEN_LAYER
        # - output layer weights: HIDDEN_LAYER x OUTPUTS
        numWeights = INPUTS * HIDDEN_LAYER + HIDDEN_LAYER * OUTPUTS
        weights = np.array(netParams[:numWeights])
        mlp.coefs_ = [
            weights[0:INPUTS * HIDDEN_LAYER].reshape((INPUTS, HIDDEN_LAYER)),
            weights[INPUTS * HIDDEN_LAYER:].reshape((HIDDEN_LAYER, OUTPUTS))
        ]

        # biases are represented as a list of 2 ndarrays:
        # - hidden layer biases: HIDDEN_LAYER x 1
        # - output layer biases: OUTPUTS x 1
        biases = np.array(netParams[numWeights:])
        mlp.intercepts_ = [biases[:HIDDEN_LAYER], biases[HIDDEN_LAYER:]]

        return mlp

    def get_score(self, netParams):
        """
        calculates the score of a given solution, represented by the list of float-
valued network parameters,
        by creating a corresponding MLP Regressor, initiating an episode of the
Cart-Pole environment and
        running it with the MLP controlling the actions, while using the
observations as inputs.
        Higher score is better.
        :param netParams: a list of floats representing the network parameters
(weights and biases) of the MLP
        :return: the calculated score value
        """

        mlp = self.init_mlp(netParams)

        self.env.reset()

        actionCounter = 0
        totalReward = 0
        observation = self.env.reset()
        observation = observation[0]
        action = int(mlp.predict(observation.reshape(1, -1)) > 0)

        while True:
            actionCounter += 1
            observation, reward, done, truncated, info = self.env.step(action)
            totalReward += reward

            if done or truncated:
```

```
                break
            else:
                action = int(mlp.predict(observation.reshape(1, -1)) > 0)
                #print(action)

        return totalReward

    def save_params(self, netParams):
        """
        serializes and saves a list of network parameters using pickle
        :param netParams: a list of floats representing the network parameters
(weights and biases) of the MLP
        """
        savedParams = []
        for param in netParams:
            savedParams.append(param)

        pickle.dump(savedParams, open("cart-pole-data.pickle", "wb"))

    def replay_with_saved_params(self):
        """
        deserializes a saved list of network parameters and uses it to replay an
episode
        """
        savedParams = pickle.load(open("cart-pole-data.pickle", "rb"))
        self.replay(savedParams)

    def replay(self, netParams):
        """
        renders the environment and uses the given network parameters to replay an
episode, to visualize a given solution
        :param netParams: a list of floats representing the network parameters
(weights and biases) of the MLP
        """
        mlp = self.init_mlp(netParams)



        actionCounter = 0
        totalReward = 0
        observation = self.env.reset()
        observation = observation[0]
        action = int(mlp.predict(observation.reshape(1, -1)) > 0)
        self.env.render()

        while True:
            actionCounter += 1
            self.env.render()
            observation, reward, done, truncated, info = self.env.step(action)
            totalReward += reward
```

```
            print(actionCounter, ": -------------------------")
            print("action = ", action)
            print("observation = ", observation)
            print("reward = ", reward)
            print("totalReward = ", totalReward)
            print("done = ", done)
            print("truncated = ", truncated)
            print()

            if done or truncated:
                break
            else:
                time.sleep(0.03)
                action = int(mlp.predict(observation.reshape(1, -1)) > 0)

        self.env.close()


def main():
    cart = CartPole()
    cart.replay_with_saved_params()

    exit()


if __name__ == '__main__':
    main()
```