Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра математики

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему: **«Алгоритм знаходження оптимальної стратегії в задачах керування марковськими ланцюгами з обмеженим горизонтом»**

Виконав: студент 4-го року навчання
освітньої програми «Прикладна математика»,
спеціальності 113 Прикладна математика

Бабін Ігор Андрійович

Керівник: Чорней Р. К.,
кандидат фіз.-мат. наук, доцент

Рецензент_____
(прізвище та ініціали)

Кваліфікаційна робота захищена
з оцінкою _____

Секретар ЕК _____
«____» _____ 20____ р.

Київ – 2021

Зміст

Вступ

Задачі керування марковськими ланцюгами набули великої популярності за останні 10 років у зв'язку з бурним розвитком такої сфери машинного навчання як навчання з підкріпленням, яке базується на Марковському процесі прийняття рішення(Markov decision process, MDP)[1]. MDP надає досить гнучкий опис задачі, що дозволяє звести досить багато сучасних задач до формулювання MDP. Прикладами таких задач є керування роботами[3], гра Go[9], генерування нових сполук[4], безпілотні автомобілі[8] та інші.

MDP насьогодні не має широкого використання в задачах комбінаторної оптимізації, які можна сформулювати як MDP з обмеженим горизонтом [10]. Прикладом такої задачі може бути знаходження розв'язку для Кубіка Рубіка [11] який має велику кількість станів - $4.3{\cdot}10^{19}$, але всього один стан-розв'язок. Більше того, випадкова послідовність поворотів будь-якої довжини досить з великою ймовірністю не приведе до розв'язку, але будь-який стан можна вирішити за менш ніж 20 кроків

В даній роботі розглядаються знаходження оптимального розв'язку для Кубика Рубіка (далі КР) розміру 3x3x3, а саме загальні підходи до вирішення задач сформульованих як задача MDP, розглянуто алгоритм Глибинний куб та розроблено покращення цього алгоритму. Результати цього дослідження мають теоретичне та практичне спрямування і можуть надати теоричне обгрунтування для майбутніх досліджень.

Об'єктом дослідження є Марковські процеси прийняття рішення на прикладі задач комбінаторної оптимізації. Предметом дослідження є знаходження оптимального розв'язку задачі комбінаторної оптимізації, сформульованої в термінах Марковського процесу прийняття рішення. Метою роботи є дослідити й описати результати запропоноваго алгоритму навчання з підкріпленням Глибинний куб [2] та розробити рекомендації щодо покращення цього алгоритму.

Робота складається з 3 розділів. В першому розглянуто основні означення та підходи до вирішення задач MDP, у другому теоритично проаналізовано метод Глибинний куб, в останньому розділі описана теоретична ідея покращення методу Глибинний куб та наведена практична імплементація разом з екпериментами для порівняння двох методів.

Розділ 1

Необхідні визначення

В цьому розділі наведено огляд необхідної літератури для того, щоб перейти до аналізу Глибинного куба

1.1 Марковський процес прийняття рішень

Означення 1.1.1. *Марковський процес прийняття рішень[1](MDP) - це множина* $(S, A, P_a(\cdot,\cdot), R_a(\cdot,\cdot), \gamma)$*, де:*

- S *множина можливих станів(може бути нескінченною)*
- A *множина можливих дій(може бути скінченною, незліченною або залежити від стану)*
- $P_a(s, s') = Pr(s_{t+1}|s_t = s, a_t = a)$ *ймовірність того, що дія* a *в стані* s *в момент часу* t *призведе до стану* s' *момент часу* $t + 1$
- $R_a(s, s')$ *винагорода (безпосередня) за перехід до стану* s' *зі стану* s
- $\gamma \in [0, 1]$ *коефіцієнт дисконтування, який представляє відмінність важливості майбутніх та поточних винагород.*

Також $P_a(\cdot,\cdot)$ задовольняє Марківську властивість - ймовірність переходу в наступний стан залежить лише від поточного стану і не залежить від попередніх.

Означення 1.1.2. *Стратегія* π *- відображення* $\pi : S \times A \to [0; 1]$*, де* $\pi(s, a)$ *позначає ймовірність обрати дію* a*, коли середовище в стані* s

Основною задачею MDP є знаходження оптимальної стратегії π^*, яка максимізує функцію кумулятивного очікування майбутніх винагород:

$$\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1}) \qquad a_t = \pi(s_t)$$

1.2 Вирішення MDP за допомогою динамічного програмування

Ітерація за цінністю - базується на рівнянні Белмана [1]:

$$V_{i+1}(s) := \max_a(\sum_{s'} P_a(s,s')(R_a(s,s') + \gamma V_i(s'))),$$

де i є номером ітерації. V_i це таблиця значень для кожного стану з множини станів S. Після кожної ітерації значення оновлюється. Алгоритм закінчується, коли зміни не перевищують якесь задане ϵ.

Ітерація за стратегією - також базується на рівнянні Белмана. Відрізняється від ітерації за цінністю тим, що замість лише цінності стану s зберігається таке значення для кожної дії виконаної в цьому стані. Тобто "яка цінність виконати дію a_i в цьому стані".

1.3 Навчання з підкріпленням

Означення 1.3.1. *Якщо ймовірності переходу або нагороди невідомі, то така задача є задачею навчання з підкріпленням, тобто існує агент, який досліджує стани, отримує винагороди за переходи і, базуючись на цьому, намагається знайти оптимальну стратегію.*

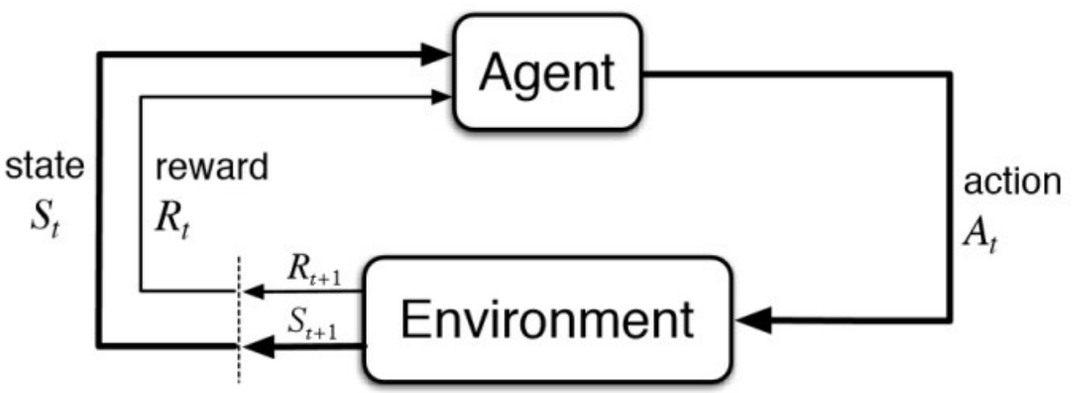

Рис. 1.1: Кроки навчання з підкріпленням

У звичайному машинному навчані ми маємо якийсь набір данних і маємо навчити нейромережу прогнозувати відповідь або знайти залежність між даними та відповідями. У навчанні з підкріпленням ми не знаємо відповіді на початку, ми маємо лише модель середовища і маємо взаємодіяти з середовищем за допомогою дій, щоб отримати відповіді, які після цього можна навчитися прогнозувати за допомогою нейромереж.

1.4 Глибинне Q-навчання

Цей алгоритм став першою спробою використаня нейромереж в задачах RL. Алгоритм глибинного Q-навчання вперше з'явився у статті пов'язаній зі знаходженням стратегії в Атарі іграх [6]. Найбільшим викликом стало те, що ці ігри мають досить велику кількість станів - зображення розміру 210 x 160 x 3 x 255^3 (RGB). Така кількість станів завелика, щоб використати класичне Q-

Рис. 1.2: Ігри Атарі

навчання. Тому науковці вирішили апроксимувати Q-функцію за допомогою нейромережі. На вхід мережа отримує стан (зображення) і має спрогнозувати одну з можливих дій. Зазвичай на вході мережі маємо скінченний набір дій, і на виході отримуємо ймовірнісний розподіл на множині дій. Тренування відбувається трохи незвичайним шляхом: спочатку агент робить якісь дії (мережа ініціалізована випадково), потім збирає досвід до фіксованого розміру буфера досвіду, потім береться якась кількість з цього буфера і робиться один крок навчання мережі.

Algorithm 1: Глибине Q-навчання

Ініціалізувати буфер досвіду D розміра N
Ініціалізувати мережу-апроксимацію Q функції випадковим чинном
for *епізод = 1, M* **do**
  for $t = 1, T$ **do**
    Обрати дію $a_i = max_a Q(s_t, a; \theta)$
    Виконати дію a_i
    Отримати нагороду r_t і новий стан (зображення) s_{t+1}
    Зберегти досвід (s_t, a_t, r_t, s_{t+1}) до буфера D
    Взяти якусь кількість $x_i = (s_t, a_t, r_t, s_{t+1})$ з буфера D
    $y_i = r_j + \gamma max_a Q(s_{t+1}, a'; \theta)$
    Виконати крок градієнтного спуска для $(y_j - Q(x_j, a_j; \theta))^2$
  end
end

1.5 Апроксимована ітерація за стратегією

Апроксимація стратегії є іншим класом алгоритмів з використанням нейромереж для задач RL. Якщо у Q-навчанні була апроксимація цінності, то

тут маємо апроксимацію функції стратегії. Зазвичай клас таких методів називається політико-градієнтними методами. Базовим алгоритмом цього класу є алгоритм "Підкріплення"(REINFORCE [7])

Означення 1.5.1. *Траєкторія - послідовність трійок: стан, дія та нагорода. Немає обмеження на довжину такої послідовності. Останній стан може бути як довільним станом так і термінальним.*

$$\tau = (s_0, a_0, r_1, s_1, a_1, r_2, ..., a_H, r_{H+1}, s_{H+1}))$$

Означення 1.5.2. *Цінність траєкторії - значення майбутньої нагороди для кожного стану з траєкторії, відповідно до цієї траєкторії.*

$$R(\tau) = (G_0, G_1..G_H) \qquad G_k := \sum_{t=k+1}^{H+1} \gamma^{t-k-1} R_k$$

Алгоритм REINFORCE полягає в тому, що агент збирає траєкторію, використовуючи нейромережу як стратегію. Після цього обраховується очікувана цінність для траєкторії і робиться крок градієнтного підйому (так само як градієнтний спуск, але з протилежним знаком).

Algorithm 2: REINFORCE

Ініціалізувати мережу-апроксимацію π_θ випадковим чинном
while *Нейромережа оновлюється* **do**
  Використовуючи політику зібрати траєкторію
  $\tau = (s_0, a_0, r_1, s_1, a_1, r_2, ..., a_H, r_{H+1}, s_{H+1}))$
  Обрахувати $R(\tau) = (G_0, G_1..G_H)$
  $U = \sum_{t=0}^{H} \log(\pi_\theta(a_t|s_t))G_t$
  $\theta = \theta + \alpha U$
end

1.6 Дерево пошуку Монте-Карло MCTS

Цей алгоритм є широко використовуваним, зокрема, він використовувався в алгоритмі AlphaGo[9]. Є алгоритмом пошуку по дереву. Такі алгоритми як пошук в ширину (BFS) і пошук в глубину (DFS) є досить універсальними алгоритмуми пошуку, але коли кількість вершин велика, вони займають багато часу, щоб знайти потрібну вершину.

Хрестики-нулики є прикладом гри, де достатнью використати класичний метод пошуку, оскільки дерево можливих дій в цій грі складає всього 50 вершин. В той час як в грі Go на кожному кроці гравець має в середньому

250 можливих дій і на глибині дерева в 10 кроків кількість можливих станів буде $250^{10} \approx 9*10^{23}$, тому класичні методи пошуку просто не зможуть знайти потрібний стан.

Пошук Монте-Карло не потребує перебору по всіх нодах, він досліджує певні випадкові шляхи в дереві і вибирає найбільш привабливе для нього. Першим кроком алгоритму є вибір ноди на основі вже відомої інформації. На кожному рівні цінність ноди визначається за формулою $\hat{x}_i + c\sqrt{\frac{\log N}{n_i}}$, де $\hat{x}_i$ це середня цінність нод, що є нащадками цієї ноди, N кількість досліджених батьків, n_i - кількість досліджених нащадків.

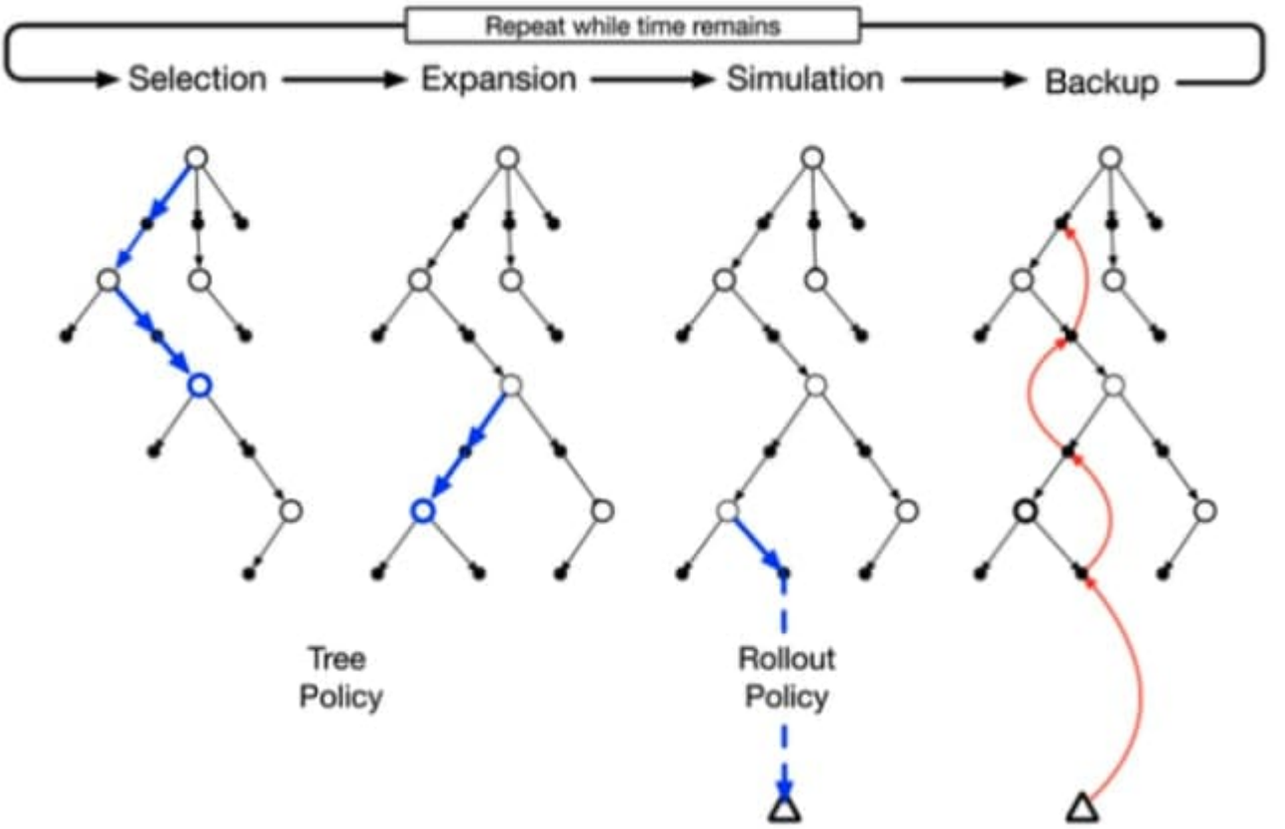

Рис. 1.3: Кроки алгоритму Монте-Карло

На другому кроці випадково обирається одна з листкових нод, яка буде далі досліджуватися. Далі іде rollout стратегія, за якою досліджується дерево, це може бути будь-яка функція, нейромережа або випадковість. Після досягнення термінальної ноди, йде останній крок, коли оновлюються значення нод на шляху до цієї термінальної ноди.

Algorithm 3: Дерево пошуку Монте-Карло

```
for епізод = 1, M do
    шлях = select(вершина)
    листочок = шлях[-1]
    листочок = expand(листочок)
    нагорода = 0
    for глибина = 1, N do
        нагорода += simulate(листочок)
    end
    backup(шлях, нагорода)
end
```

1.7 Евристичний пошук

Алгоритм знаходження найкоротшого шляху в зваженному графі, який базується на алгоритмі Дейкстри. Головною відмінністю алгоритму Дейкстри від BFS і DFS є те, що він обходить вершини в іншому порядку - в порядку збільшення ваг вершин до якоїсь вершини. Тобто тут ми маємо чергу з пріоритетом, в якій пріоритет - це вага ребра до цієї вершини.

Натомість, евристичний пошук враховує не тільки вагу ребра, а ще й якусь евристику. Наприклад, нехай маємо граф, який представляє собою карту міст, де вершина - це місто, а ребро - дорога. Тоді евристикою кожної вершини може бути відстань (фізична) до цільової вершини. Такий пошук працює значно швидше, коли ми маємо чітко визначену евристику.

Algorithm 4: Евристичний пошук

```
закриті вершини = {}
відкриті = черга з пріоритетом()
відкриті <- додати початкову вершину
while відкриті не порожня do
    вершина = відкриті.top()
    якщо вершина = цільова, повернути вершину
    якщо вершина в закриті, продовжити
    додати всі суміжні вершини до відкритих
    додати вершину до закритих
end
```

Розділ 2

Огляд алгоритму знаходження оптимального розв'язку MDP

В цьому розділі огляд алгоритму Глибинний куб [2]: самостійна ітерація (Autodidactic iteration) як метод, за допомогою якого можна натренувати нейромережу, яка буде оцінювати кількість кроків, щоб розв'язати КР, і як ця мережа використовується разом з MCTS.

2.1 Самостійна ітерація

Цей метод належить до класу методів політико-градієнтних, коли ми апроксимуємо функцію стратегії. Також цей алгоритм добре справляється, коли середовище має досить розріджені нагороди. Цей метод добре підходить для КР, тому що це середовище має лише один стан з нагородою - стан-розв'язок.

Метою цього метода є оптимізація нейромережі, яка представляє собою апроксимацію функції-стратегії. У випадку з КР, політика буде функцією, яка оцінює стан і ця оцінка - кількість кроків до стану розв'язку. Оскільки станів досить багато у КР, то ми не можемо вчити нейромережу, даючи їй всі можливі стани як вхід. Саме тому автори метода запропонували наступний підхід до навчання мережі: на кожному кроці ми генеруємо випадковим чином дії, які виконуємо з термінального стану і збираємо всі стани, які ми отримали під час виконання цих дій. Також оскільки ми почали з термінального стану, то ми знаємо, скільки кроків потрібно в кожному зі станів, щоб дійти до нього. Тому ми маємо дані, які подаємо на вхід до мережі. Крім того, генерація станів проходить не зовсім випадково, а таким чином, щоб ми не повернулися до якогось зі станів з цієї послідовності.

Algorithm 5: Самостійна ітерація

Ініціалізувати нейромережу f_θ
for *епізод = 1, M* **do**
  A = Послідовність з 20 дій
  X = Множина станів, які отримали при виконанні дій з A, починаючи з термінального стану
  for x_i *in* X **do**
    $Y_i = f_\theta(x_i)$
  end
  $\theta = train(f_\theta, X, Y)$
end

2.2 Використання дерева пошуку Монте-Карло

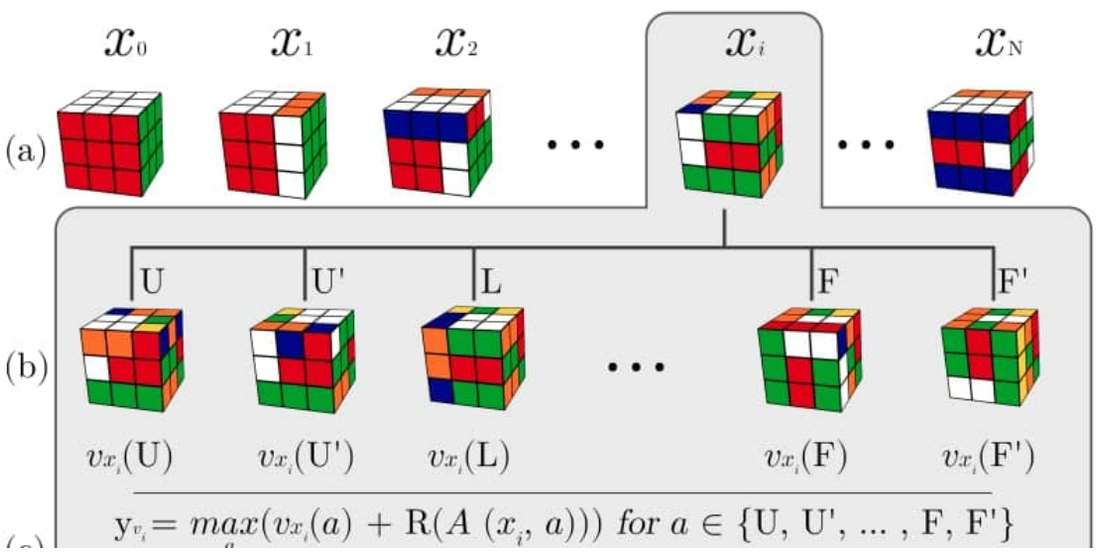

Рис. 2.1: Дерепо пошуку Монте-Карло на прикладі КР

Основною адаптацією дерева пошуку стало те, що автори використовують мережу, натреновану за допомогою самостійної ітерації як rollout-функції. Крім того, щоб метод знаходив більш короткий розв'язок, автори додали ще глибину, тобто кількість кроків від початкового стану.

Проте таке використання MCTS призводить до субоптимальних розв'язків, оскільки MCTS досліджує не всі можливі шляхи, а певну кількість, про що і свядчять результати - в середньому розв'язки 30 кроків.

Розділ 3

Розробка власного алгоритму

В цьому розділі опис запропонованого алгоритму, його практична імплементація та експерименти у рівних умовах з Глибинним кубом

3.1 Основна ідея алгоритму

Дерево пошуку Монте-Карло є досить ефективним для пошуку шляху до термінального стану, особливо у мінімакс іграх як Go, але ми не маємо гарантій, що цей шлях буде оптимальним. Це можна помітити по результатах Глибинного куба - він знаходить розв'язки до 30 кроків, в той час як оптимальний - не більше 20.

Саме тому запропонованний метод використовує інший тип пошуку - евристичний пошук, який може гарантувати оптимальний розв'язок за умови, що ми маємо досить гарну евристику. За евристику можна залишити нейромережу з алгоритму Глибинного куба, тому що кількість кроків до розв'язку є досить гарною евристикою для пошуку, навіть якщо ці оцінки не є точними. Тому пошук має давати оптимальне рішення.

3.2 Опис середовища КР

Кожен стан КР у найпростішому випадку може бути представленим за допомогою 6 чисел (кольорів) для кожного з 9 значень кожної з 6 граней, тобто 54 числа доситатньо для зберігання стану КР. В імлементації кожен елемент має своє число від 0 до 54.

```
std::vector<uint8_t> states;
states.reserve(54);

for(uint8_t i = 0; i < 54; i++){
  states.push_back(i);
}
```

Маючи такий стан, перевірка чи зібраний КР у цьому стані виглядає дуже просто, треба лише перевірити кожен елемент

```
    bool isSolved() const {
  for(int i = 0; i < 54; i++)
  if (state[i] != i)
    return false;

  return true;
}
```

З будь-якого стану ми маємо 6 можливих дій за годинниковою стрілкою, 6 проти годиникової та 6 подвійних дій за годинниковою стрілкою, тому всього 18 можливих дій. Таке розбиття на дії є загальноприйнятим і 20 кроків для вирішення будь-якого КР пораховано у термінах таких дій. Опис алгоритму перестановки можна знайти тут: [2]

```
Cube getNextState(int action) const {
  state new_state(st);

  for(int i = 0; i < 24; i++){
    auto oldIdx = rotateIdxs_old[action][i];
    auto newIdx = rotateIdxs_new[action][i];
    new_state[newIdx] = st[oldIdx];
  }

  return Cube(new_state);
}
```

Також для евристичного алгоритму пошуку потрібно вміти отримати всі стани, які можна досягти за допомогою однієї дії з поточного стану:

```
std::vector<Cube*> getNextStates() const {
  std::vector<Cube*> states;

  for(int i = 0; i < numActions; i++){
    states.push_back(new Cube(getNextState(i)));
  }

  return states;
}
```

3.3 Імплементація та тренування мережі

В оригінальному методі використовується ResNet50 архітектура, до якої приєднано один шар нейронної мережі з одним нейроном в кінці, оскільки маємо задачу регресії.

```
nn.Sequential([
  nn.ResNet50()
  nn.Linear(1)
])
```

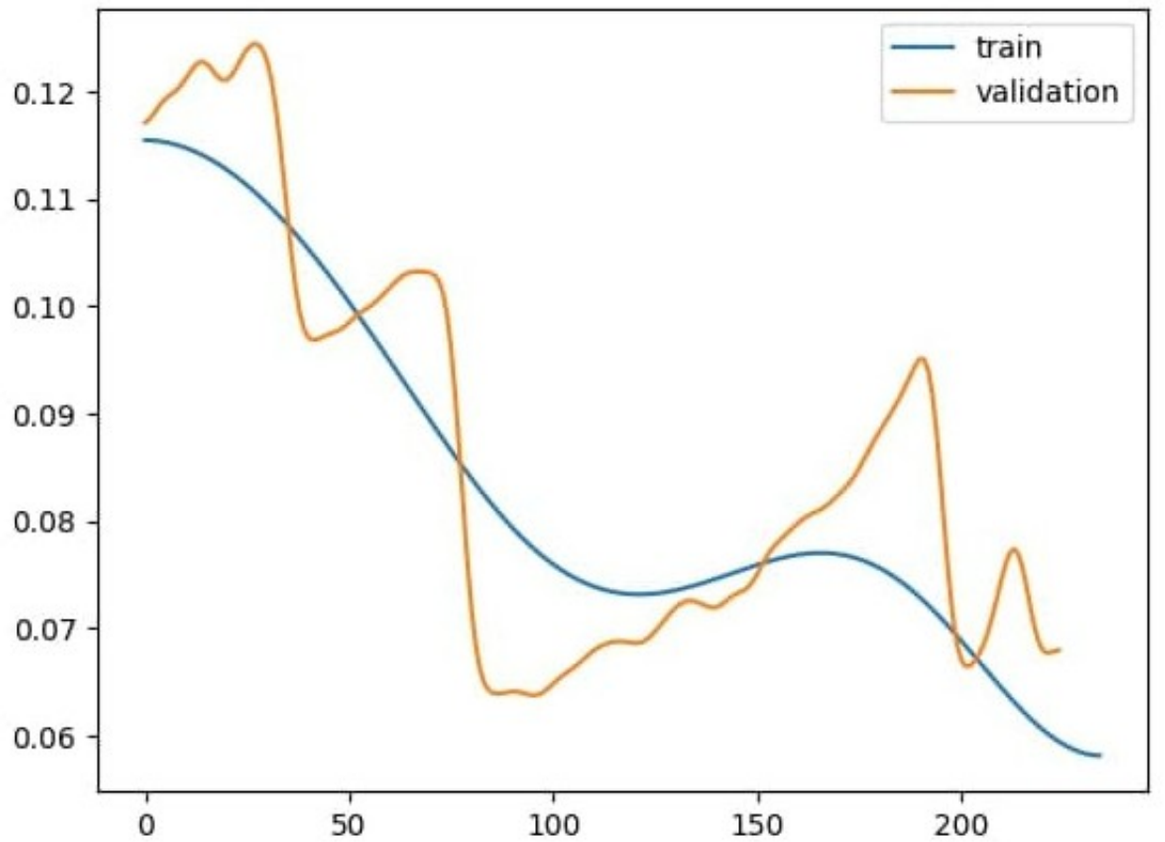

Рис. 3.1: Графік функції втрат

Генерація даних для тренування мережі відбувається згідно з алгоритмом Самостійної ітерації з тим урахування, що береться не випадкова дія, а якась дія, щоб не повернутися до попередніх станів

```
std::vector<pair<int, int>> actions;
for(int counter = 0; counter < 20; counter++){
  do {
  action = distrib(gen); // number in range 24
  } while (!isValidNextMove(actions, action));
  actions.emplace_back({action, counter});
}
```

Оскільки маємо задачу регресії, то як лосс-функція використано L2 відстань. Нижче наведено графік під час навчання та валідації.

3.4 Імплементація пошуку

Для алгоритму пошуку нам потрібно мати чергу з пріоритетом, в яку будуть додаватися сусідні стани. Також потрібно мати множину станів, в яких ми вже були, щоб рекурсивно не повторювати пошук. На початку черга ініціалізується початковою вершиною

```
struct Node {
  const Cube *env;
  int depth;
  int parentMove;
  float cost;
  float heuristic;
  Node *parent;
};
std::priority_queue<Node*> open;
std::unordered_set<Node*> closed;

open.push(new Node{&cube});
closed.insert(new Node{&cube});
```

Далі, поки ми маємо вершини в черзі, ми беремо першу вершину з черги, перевіряємо, чи не є це розв'язком, якщо є, то просто повертаємо. Якщо ні, то ми беремо всі сусідні стани до початкового та перевіряємо, чи не були вже в цьому стані.

```
Node *node = open.top();
open.pop();
if (node->env->isSolved()) {
  return node;
}
std::vector < Node * > children(cube.getNumActions());
std::vector<Cube*> children_env = node->env->getNextStates();
int depth = popped[i]->depth + 1;
for (size_t j = 0; j < children_env.size(); ++j) {
  Node *node = new Node{children_env[j], depth, (int) j, cost,
    heuristic, popped[i]};

  children[i * cube.getNumActions() + j] = node;
}

for (size_t i = 0; i < children.size(); ++i) {

  Node *node = children[i];
  auto found = closed.find(node);


  if (found == closed.end()) {
    closed.insert(node);
  } else if ((*found)->depth > node->depth) {
    open.push_back(node);
  } else {
    delete node;
  }
}
```

Далі для нод які залишилися, формується один батч, який передається до нейромережі, яка для кожного передбачає евристику, після цього вершини додаються до черги відкритих вершин і цикл починається знову.

```
std::vector<float> costs(nodesToAdd.size());
std::vector<float> values(nodesToAdd.size());

#pragma omp parallel for
for (size_t i = 0; i < nodesToAdd.size(); ++i) {
  values[i] = predict(nodesToAdd[i]->env->getState(), sockfd);
```

```
  float cost = values[i] * (!nodesToAdd[i]->env->isSolved()) +
    depthPenalty * ((float) nodesToAdd[i]->depth);
  costs[i] = cost;
}

for (size_t i = 0; i < nodesToAdd.size(); ++i) {
  Node *nodeToAdd = nodesToAdd[i];

  nodeToAdd->cost = costs[i];
  nodeToAdd->heuristic = values[i];

  open.push(nodeToAdd);
}
```

3.5 Експерименти

Автори алгоритму Глибинного куба тестували його таким чином [2]: брали початковий стан і виконували 1000 випадкових дій, після цього розв'язували отриманний стан. Таким чином вони розв'язали 640 станів і порівняли з класичними методами для ров'язання.

Автори мали великі обсчилювальні можливості, тому було обрано інший підхід до тестування - виконувати i випадкових дій 20 разів, для $i = \overline{1, 20}$

```

for (int steps = 1; steps <= 20; steps++) {

  for (int i = 0; i < 20; i++) {

  auto cube = randomScramble(steps);


  search(cube, model);

  }

}
```

3.6 Результати

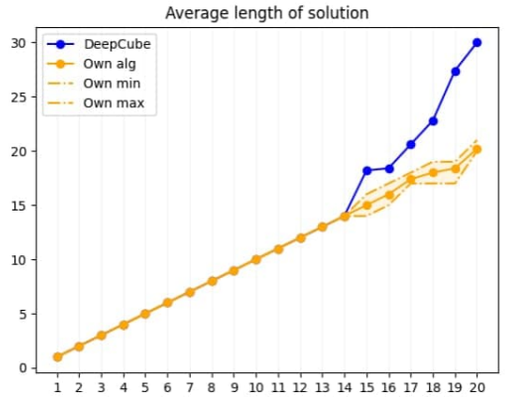

Рис. 3.2: Середня довжина розв'язку

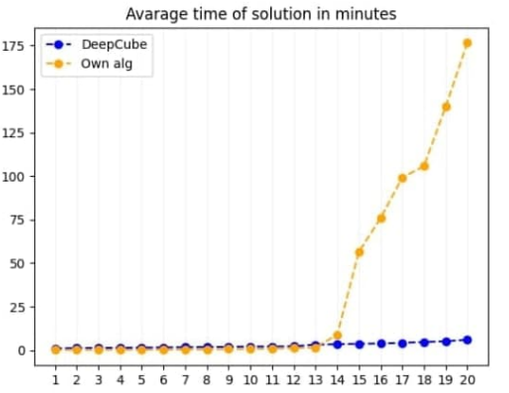

Рис. 3.3: Середній час пошуку розв'язку

На першому графіку вказано середню кількість кроків для вирішення станів отриманих за певною кількістю кроків. Як і очікувалось, модифікований алгоритм знаходить оптимальні рішення на відмінну від оригінального, але інколи рішення можуть бути на 1-2 крок більші або менші. Це пов'язано з тим, що евристика не є досить точною: для тренування використовувались випадкові дії, але отриманий стан, наприклад, за 18 кроків, інколи міг бути скороченний на декілька кроків, тому мережа інколи отримувала неправильні дані, що призвело до неточної оцінки. Також було проведено тест Мана-Уітні[12] для результатів у кожній групі і було отримано, що медіана покращенного алгоритма менша за оригінального алгоритма.

На другому графіку середній час розв'язання стану. Тут можна побачити основний недолік модифікації - досить велики час роботи: для 20 кроків в середньому потрібно 175 хвилин, в той час як оригінальний потребує в середньому 6 хвилин.

Висновки

В цій роботі було розроблено модифікацію алгоритму Глубинного куба, яка використовує інший тип пошуку. Також було практично реалізовано алгоритм Глубинного куба та його модифікацію. Експериментально було перевірено результати авторів алгоритму Глубинного куба та перевірено більшу оптимальність модифікованного алгоритму у кількості кроків для розв'язання

Найбільшим недоліком алгоритму є те, що він є досить повільний через те, що мережа отримує неправильні дані. Як можливе покращення, потрібно спробувати змінити генерацію даних для тренування або ж використати існуючу мережу для того, щоб перевіряти оптимальність згенерованного стану.

Як зовсім інший напрямок до рішення цієї задачі можна розглянути алгоритм сімества апроксимації дії - тобто мережа буде передбачувати не цінність стану, а яку дію краще обрати. Найбільшою практичною проблемою в цьому випадку буде потреба у великих обрахункових потужностях для тренування мережі.

Запропонована модифікація вирішила 100% знегерованих станів різної довжини для генерації, майже у всіх випадках було отримано оптимальний розв'язок, але у декількох було отримано на 1-3 кроки коротший/довший розв'язок

Література

[1] Bellman, R. (1957). A Markovian Decision Process. Journal of Mathematics and Mechanics

[2] Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. Solving the Rubik's cube with deep reinforcement learning and search. Nature Machine Intelligence, 1(8):356–363, 2019.

[3] Mahmood et al. Benchmarking Reinforcement Learning Algorithms on Real-World Robots

[4] Alphafold Google DeepMind

[5] Peter E.Hart, Nils J.Nilsson, and Bertram Raphael. A formal Basis for the Heuristic Determination of Minimum Cost Paths. SRI International, 1968.

[6] Mnih, V. et al. Playing Atari with Deep Reinforcement Learning

[7] Williams Ronald et al. Simple Statistical Gradient-Following Algorithms for Connnectionist Reinforcement Learning

[8] Kiran1 et al. Deep Reinforcement Learning for Autonomous Driving

[9] Silver et al. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm

[10] Combinatorial optimization and Markov decision process for planning MRI examinations Na Geng

[11] Deep Reinforcement Learning Hands-On - Second Edition

[12] Mann H. B., Whitney D. R. On a test of whether one of two random variables is stochastically larger than the other. // Annals of Mathematical Statistics. — 1947. — № 18. — P. 50—60.