МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Львівський національний університет ім. Івана Франка
Факультет електроніки та комп’ютерних технологій
Кафедра системного проектування

КУРСОВА РОБОТА

на тему: **“НАВЧАННЯ З ПІДКРІПЛЕННЯМ. МЕТОД Q-НАВЧАННЯ. МЕТОД ТИМЧАСОВИХ РІЗНИЦЬ TD”**

Виконав:
студент групи ФеІм-12с
Головень Р. Т.
Науковий керівник:
доцент Ляшкевич В. Я.

Львів - 2023р.

ЗМІСТ

ВСТУП…… 4

АНАЛІЗ МАТЕРІАЛІВ ТА МЕТОДІВ…….…….8

ЕКСПЕРИМЕНТИ……13

ОБГОВОРЕННЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ……...…29

ВИСНОВКИ……...32

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ……..……...….33

ДОДАТОК А……...…...35

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ І СКОРОЧЕНЬ

AI – англ. Artificial Intelligence

TD – англ. Temporal difference

RL – англ. Reinforcement learning

MDP – англ. Markov decision process

SARSA – англ. State–action–reward–state–action

ВСТУП

Курсову роботу присвячено комплексному вивченню галузі машинного навчання – навчанню з підкріпленням, а також її підрозділам та методам, в особливості методам Q-навчання та тимчасових різниць (TD).

Актуальність:

Машинне навчання з кожним днем займає все більше місце в нашому житті з огляду на величезний спектр застосувань: починаючи від розумних автомобілів з автопілотом та закінчуючи ботами для ігор зі штучним інтелектом. Машинне навчання вважається розділом штучного інтелекту, основна ідея якого полягає в тому, щоб комп'ютер не просто використовував заздалегідь написаний алгоритм, а сам навчився вирішувати поставлене завдання.

Більшу частину завдань машинного навчання можна розділити на навчання з учителем (англ. supervised learning) і навчання без вчителя (англ. unsupervised learning). Коротко пояснити цих два типи можна наступним чином: при навчанні з учителем у нас є дані, на підставі яких потрібно щось передбачити, і деякі гіпотези, а при навчанні без учителя у нас є тільки дані, властивості яких ми і хочемо знайти.

Проте є ще й інший розділ машинного навчання - навчання з підкріпленням (англ. reinforcement learning) [5]. Його суть у тому, що агент повинен вчиняти дії в середовищі таким чином, щоби максимізувати деяке уявлення про довготермінову винагороду. Навчання з підкріпленням відрізняється від навчання з учителем тим, що пари правильних входів/виходів йому ніколи не представляються, і не зовсім оптимальні дії ніколи явно не виправляються.

Відповідно, у навчанні з підкріпленням можна виділити дві основних мети:

- Перша мета – мінімізувати помилки. Саме тому машина вчиться аналізувати інформацію перед кожним наступним ходом.

- Друга мета – отримати від виконання завдання максимальну вигоду, при тому сама вигода при цьому повинна бути запрограмована заздалегідь.

Навчання з підкріпленням застосовується у випадках коли потрібно вибрати кращий варіант серед багатьох або досягти складної мети за безліч ходів. Алгоритми підкріплення, які включають в себе глибоке навчання, можуть перемогти чемпіонів світу в грі Go, починаючи з базового розуміння правил гри і тренуючись від партії до партії. Таким чином, це вже штучний інтелект в дії: машина намагається вирішити задачу різними способами, помиляється, вчиться на своїх помилках, покращує показники. Цей метод використовують у першу чергу там, де потрібно навчити машину виживати в реальному середовищі.

Метою і завданням дослідження є комплексний аналіз машинного навчання з підкріпленням, оптимальний вибір методів, а також пошук оптимальних параметрів, що дозволить дослідити ефективність методу навчання з підкріпленням у штучному середовищі.

Об’єктом дослідження даної роботи є методи Q-навчання та тимчасових різниць.

Структура та обсяг роботи:

Робота складається зі вступу, трьох розділів (з висновками до них), (загальних) висновків, списку використаних джерел, та додатків. Загальний обсяг роботи — 40 сторінок (обсяг основного тексту – 30–35 сторінок)

Опис літературних джерел:

Дослідження вчених та розробників з відомої лабораторії досліджень штучного інтелекту OpenAI на практиці показують ефективність та можливості алгоритмів глибокого навчання, серед яких широко застосовується навчання з підкріпленням, Q-навчання, а також передавальне навчання та поєднання багато інших методів машинного навчання. Серед даних досліджень можна виділити наступні:

- OpenAI Five [9] – команда з п'яти OpenAI ботів, яких використовують у конкурентній відеогрі Dota 2 п'ять на п'ять, і які вчаться грати проти гравців-людей

на високому рівні майстерності через алгоритми спроб і помилок. Ця система [8] використовує різновид навчання з підкріпленням, оскільки боти вчаться з часом, граючи проти самих себе сотні разів на день протягом місяців, й отримують винагороду за такі дії як вбивство ворога та взяття цілей на мапі. Як показали результати [8-9], то штучний інтелект переміг чинних чемпіонів світу з цієї гри, таким чином доказавши свою ефективність.

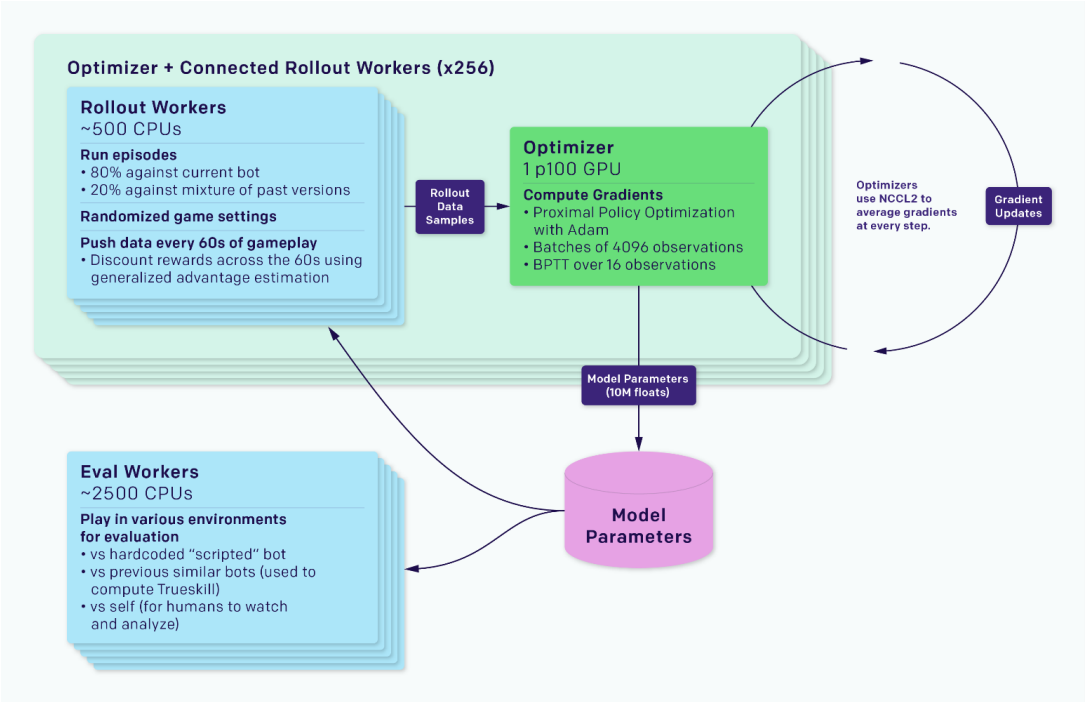


Рис. 1.1. Візуалізація комплексної системи навчання з підкріпленням “Rapid”

- Dactyl [10] – система, що використовує машинне навчання з підкріпленням, щоб тренувати роботизовану руку з нуля, використовуючи той самий алгоритм, що й OpenAI Five [9]. Результати досліджень [11] показали, що можливо навчати агентів у симуляції вирішувати реальні завдання без моделювання фізично-точного світу.

Рис. 1.2. Приклади маніпуляцій, які засвоїв Dactyl

- Hide and Seek AI [12] – дослідники з OpenAI навчали агентів в простій грі в хованки, проте вони зауважили, що протягом навчання агенти розвивали все більш складні стратегії. Результати були зафіксовані у детально описаній дослідницькій роботі [13]. Для навчання агентів вчені з OpenAI використали систему, яка вже зустрічалась в OpenAI Five [9] та Dactyl [10]. Розробники помітили, що при багатократних повтореннях підкріплень агенти навчились використовувати помилки фізичної симуляції середовища неберeдбачуваними шляхами. Дослідники вважають [12], що при більш відкритому середовищі поведінка агентів максимально наближається до людської та розвиває складніші, невідомі стратегії.

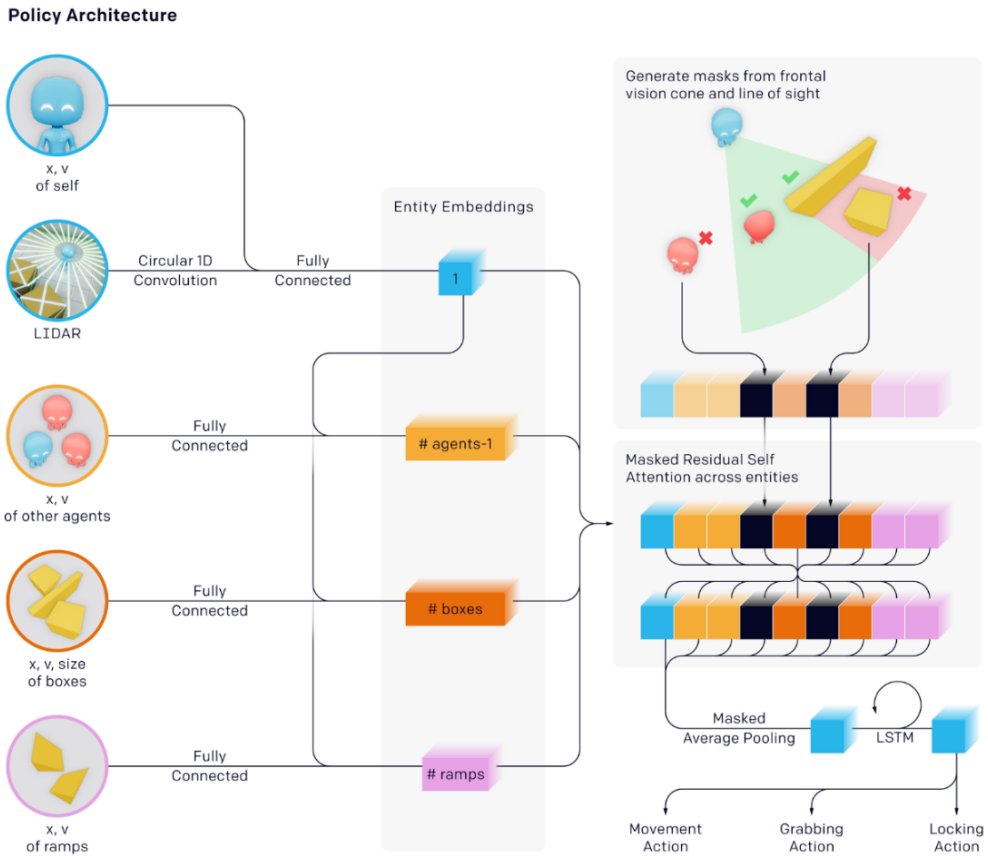


Рис. 1.3. Візуалізація системи навчання агентів в гру в хованки

На основі даних досліджень можна заявити, що область машинного навчання з підкріпленням, Q-навчання є достатньо складною, внаслідок чого не до кінця дослідженою та актуальною для аналізу та випробовувань.

АНАЛІЗ МАТЕРІАЛІВ ТА МЕТОДІВ

Формально навчання з підкріпленням моделюється, як марковський процес прийняття рішення. Ключовими поняттями навчання із підкріпленням є взаємодія агента та середовища. Період часу, за якого відбувається ця взаємодія зазвичай є скінченним. В кожен дискретний момент часу t агент отримує спостереження (observation) o_t, яке зазвичай включає винагороду (reward) r_t. Потім агент обирає дію (action) a_tіз множини його допустимих дій. Правило, на основі якого агент робить вибір дії називається стратегією (policy). Обрана дія впливає на середовище таким чином, що воно змінює свій стан (state) на s_{t+1} та визначає винагороду r_{t+1} за дію a_t агента. Взаємодія триває до настання фінального стану S_F середовища. Тоді основна задача машинного навчання із підкріпленням задається як пошук стратегій для максимізації отримуваного виграшу [3]. Функція, що за заданим станом середовища визначає найвигіднішу дію агента називається функцією цінності (quality function) Q^*.

Для оцінювання оптимальної стратегії використаємо функцію оптимальної цінності (англ. optimal value function) $V(s)$. Вона є очікуваною знеціненою сумою винагороди, яку агент отримає, якщо почне діяти в певному стані, застосовуючи оптимальну стратегію. Її можна визначити за допомогою ітераційного процесу, який називається ітерацією за цінністю (англ. value iteration) [14].

$$V(s) = max_a \left(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')\, V(s') \right), \qquad \forall s \in S$$

Обчислювальна складність алгоритму оцінювання оптимальної стратегії за ітераціями є квадратичною за кількістю станів і лінійної за кількістю дій [15].

Після того як оптимальна стратегія оцінена її потрібно поліпшити, для цього застосуємо процес ітерації за стратегією (англ. policy iteration), який є циклом між оцінкою стратегії та її поліпшенням. Тобто, як тільки дізнаємося цінність кожного стану в рамках поточної стратегії, то розглядаємо варіант підвищення цінності, змінюючи першу дію. Цей

процес гарантує підвищення ефективності стратегії. Якщо ніяких поліпшень не буде, тоді стратегія є оптимальною. З огляду на функцію оптимальної цінності визначаємо оптимальну стратегію як:

$$\pi(s) = argmax_a \left(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')\, V(s') \right)$$

Оскільки існує не більше $|A|^{|S|}$ різних стратегій, а їхня послідовність поліпшується на кожному кроці, то цей алгоритм описується експоненціальною кількістю ітерацій [14].

У роботі застосуємо алгоритм знаходження оптимальної стратегії [3]:

1. Ініціалізація масивів:
   $V(s) \in \mathbb{R}$; $\pi(s) \in A(s)$, довільно, наприклад $V(s) = 0$.
2. Оцінка стратегії:
   Повторювати доки $\Delta < 0$ (невелике позитивне число):
   $\Delta \leftarrow 0$
   Для кожного $s \in S$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow max_a(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')\, V(s'))$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
3. Покращення стратегії:
   $policy\ stable \leftarrow true$
   Для кожного $s \in S$:
   $a \leftarrow \pi(s)$
   $\pi(s) = argmax_a(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')\, V(s'))$
   Якщо $a \neq \pi(s)$, тоді $policy\ stable \leftarrow false$
   Якщо $policy\ stable \leftarrow true$, тоді зупинити та повернути V та π, інакше перейти на крок 2.

Зважаючи на описану математичну основу використаємо метод безмодельного навчання або Q-навчання (Q-learning), коли модель, яка складається із функцій переходу до стану $T(s, a, s')$ та винагороди $R(s, a)$, невідома заздалегідь [6].

Одна з переваг Q-навчання – це те, що воно в змозі порівняти очікувану корисність доступних дій, не формуючи моделі навколишнього середовища. Відповідно, як згадувалось раніше, воно застосовується для ситуацій, які можна представити у вигляді марківського процесу прийняття рішень.

Нехай $Q(s, a)$ – очікуване знецінене підкріплення дії a у стані s, $V(s)$ – це значення s, яке передабчає, що найкраща дія здійснюється спочатку. Тоді формулу запишемо у вигляді

$$V(s) = max_a(Q(s, a))$$

Отже $Q(s, a)$ представимо рекурсивно:

$$Q(s, a) = R(s, a) + \gamma \sum_{s' \in S} T(s, a, s')\, max_a(Q(s', a'))$$

Зауважимо також, оскільки $V(s) \leftarrow max_a(Q(s, a))$, то $\pi(s) = argmax_a\, Q(s, a)$ – оптимальна стратегія.

Оскільки Q-функція (англ. Q-function) робить дію явною, то оцінимо Q-значення (англ. Q-values), використовуючи метод $TD(0)$, а також визначимо стратегію (дія може бути вибрана простим відбором максимального Q-значення для поточного стану).

Табличний метод $TD(0)$ – один з найпростіших методів TD. Це окремий випадок більш загальних методів стохастичною апроксимації. Він оцінює функцію значення стану кінцевого марковского процесу прийняття рішень (MDP) в рамках політики.

Значення стратегії обчислюється із використанням методу $TD(0)$, який є екземпляром більш загального класу методів $TD(\lambda)$. $TD(\lambda)$ – це алгоритм навчання, винайдений Річардом С. Саттоном на основі більш ранньої роботи Артура Семюела по вивченню тимчасової різниці. Для прикладу, цей алгоритм був ефектвно застосований Джеральдом Тесауро для створення TD-Gammon, програми, яка навчилася грати в нарди на рівні досвідчених гравців.

Правило оновлення значень функції цінності запишемо таким чином:

$$V(s) = V(s) + \alpha\big(r + \gamma V(s') - V(s)\big)$$

Тобто кожного разу, коли настає стан s, його оцінка оновлюється, щоб буи ближчою до значення $r + \gamma V(s')$. $TD(0)$ проглядає простір тільки на один крок вперед при коригуванні оцінок цінностей. Якщо швидкість навчання коригується належним чином (вона повинна бути повільно зменшена) і стратегія незмінна, то $TD(0)$ гарантовано буде збігатися до функції оптимальної цінності, але для цього може знадобитися досить багато часу.

Загальне правило методу $TD(\lambda)$ подібне формулі (5), але воно застосовується до кожного стану згідно зі значенням $e(s)$, тобто

$$V(s) = V(s) + \alpha\big(r + \gamma V(s') - V(s)\big)\, e(s)$$

$$e(s) = \sum_{k=1}^{t} (\lambda\gamma)^{t-k}\, \delta_{s,s_k}, \qquad \delta_{s,s_k} = \begin{cases} 1, & s = s_k \\ 0, & s \neq s_k \end{cases}$$

Обчислювально складніше використовувати метод $TD(\lambda)$, хоча він часто працює значно швидше для великих λ.

Отже, врахувавши написане вище, сформулюємо правило для Q-навчання:

$$Q(s,a) = (1-\alpha)\, Q(s,a) + \alpha\big(r + \gamma max_{a'}(Q(s',a'))\big),$$

де $\{s,a,r,s'\}$ – набір досвіду; $0 < \alpha \leq 1$ – швидкість навчання; γ – коефіцієнт знецінення.

Якщо кожна дія виконується в кожному стані нескінченну кількість разів, то Q-значення будуть збігатися до оптимальних Q із імовірністю $P = 1$. Коли Q-значення майже сходяться до оптимальних значень, для агента доцільно діяти з «жадібністю» (англ. greedy policy), приймаючи в кожній ситуації дію з найвищим значенням Q.

Слід зазначити, що Q-навчання не залежить від «розвідки» (англ. exploration): це в свою чергу означає, що Q-значення будуть збігатись до оптимальних значень незалежно від того, як агент буде себе поводити під час отримання винагороди. З цих причин Q-навчання є одним з найефективніших та відповідно найпопулярніших методів безмодельного навчання. Однак, він може досить повільно наближатися до оптимальної стратегії, але це не буде критичним фактором для нашого експериментального дослідження.

ЕКСПЕРИМЕНТИ

Суть навчання з підкріпленням полягає в тому, як агент ітеративно оновлює оцінку стану шляхом випробовувань. У попередніх розділах ми вже розглянули про те, як у Q-навчанні агент оновлює своє Q-значення на основі цього методу. Насправді, окрім класичного методу оновлення, визначеного в Q-навчанні, існують ще й інші способи оновлення оцінок стану та дій. В експериментах ми дослідимо ще один метод – SARSA та порівняємо його з класичним Q-навчанням і побачимо, як різниця в методах впливає на поведінку агента.

Спочатку згадаємо про тимчасові різниці, які є основою методу оновлення. Ми знаємо, що під час кожної ітерації чи епізоду агент досліджує навколишнє середовище, дотримуючись певної політики (наприклад є-greedy), і на основі свого останнього спостереження він оновлює свої поточні оцінки. Саме ця різниця між значеннями останнього спостореження називається TD – тобто, тимчасовою різницею, саме з цієї тимчасової різниці агент навчається і покращує себе.

Визначення тимчасової різниці відрізняє методи один від одного. Розглянемо у чому вона полягає. У вищерозглянутому Q-навчанні тимчасова різниця визначається як:

$$[R + max_a(Q(S', a)) - Q(S, A)]$$

Таким чином при спостереженні Q-значення наступного стану пара дій може не впливати безпосередньо на оновлення поточного стану. Q-навчання завжди використовує максимальне значення наступного стану, тому дія, яка використовується для оновлення Q-значення може не узгоджуватись з поточною політикою. Цей метод називається off-policy.

На противагу цьому SARSA[19] використовує on-policy, оскільки процес оновлення відповідає поточній політиці. Та визначається наступним чином:

$$[R + max_a(Q(S', A')) - Q(S, A)]$$

Існує ще модифікація SARSA методу – Expected SARSA[20]. Замість того, щоб брати максимум для наступних пар стан-дія, він використовує очікуване значення, беручи до уваги те, наскільки ймовірно кожна дія знаходиться під поточною політикою.

$$\left[R_{t+1} + \sum_{a} \pi\,(a|S_{t+1})Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

Враховуючи наступний стан S_{t+1} цей алгоритм рухається в тому ж напрямку, що очікується від SARSA, цим і спричинена його назва. Хоча він більш обчислювально складний, але він усуває дисперсію через випадковий вибір A_{t+1}

Насправді, ця невелика різниця між алгоритмами може призвести до суттєво різної поведінки агента. Тому проведемо експерименти на практичній задачі, де побачимо, чи є ця різниця, на що вона впливає та як змінюватимуться результати.

В навчанні з підкріпленням існує класична проблема – це задача про прогулянку по скелі (англ. cliff walking problem) [16]. Це стандартне епізодичне завдання в середовищі у клітинку зі станами початку та цілі. Агент може рухатись вліво, вправо, вниз та вгору по клітинках. Нагороду дорівнює -1 за всі переходи, крім тих, що переходять в регіон позначений скелею. При попаданні за край скелі агент отримує нагороду -100 та повертається на початок гри.

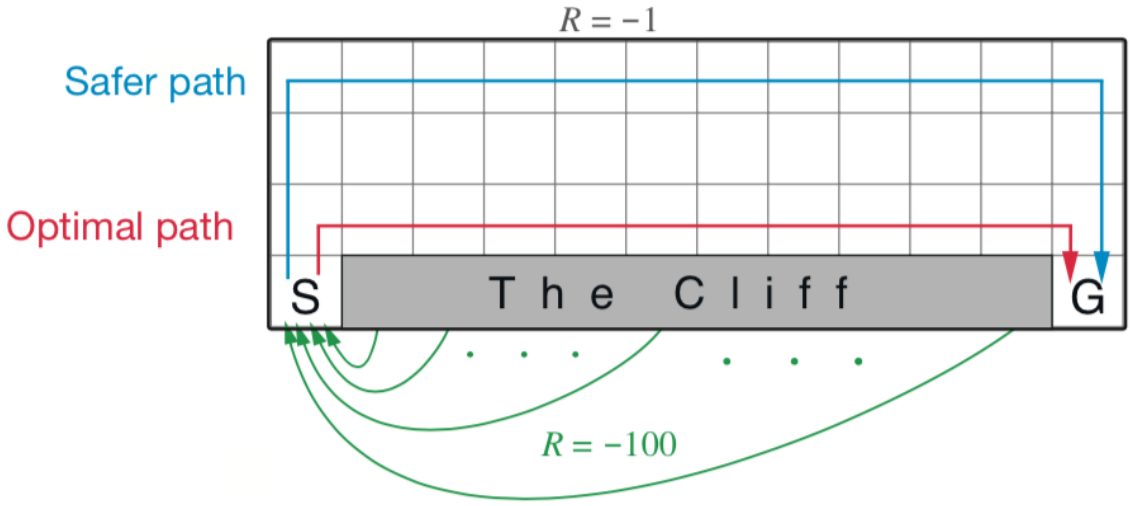


Рис. 3.1. Візуалізація cliff walking problem [17]

У нашому випадку агент рухатиметься в середовищі 4×12, де стани пронумеровані наступним чином:

```
[[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11],
 [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23],
 [24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35],
 [36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47]]
```

На початку кожного епізоду, стан 36 є початковим. Стан 47 це єдиний можливий стан виходу (тобто ціль), та скеля знаходиться на проміжку від 37 до 46.

Агент має 4 наступні можливі дії:

- UP = 0;
- RIGHT = 1;
- DOWN = 2;
- LEFT = 3.

Оптимальна функція стану-значення (англ. state-value function) візуалізована, як:

Рис. 3.2. Визначення оптимальної state-value function

Напишемо функцію для оновлення Q-значення:

```
def update_Q(Qsa, Qsa_next, reward, alpha, gamma):
    return Qsa + (alpha * (reward + (gamma * Qsa_next) - Qsa))
```

Та функцію для жадібної стратегії (англ. greedy policy):

```
def epsilon_greedy_probs(env, Q_s, i_episode, eps=None):
    epsilon = 1.0 / i_episode
    if eps is not None:
        epsilon = eps
    policy_s = np.ones(env.nA) * epsilon / env.nA
    policy_s[np.argmax(Q_s)] = 1 - epsilon + (epsilon / env.nA)
    return policy_s
```

Функції кожного алгоритму будуть використовувати дві попередні, а також приймати вхідні аргументи:

- **env**: середовище, яке ми будемо використовувати (OpenAI Gym)
- **num_episodes**: кількість епізодів, яку згенерувати для взаємодії середовища та агента.
- **alpha**: параметр, який визначає розмір кроку для оновлення, тобто фактично темп навчання (повинен бути між 0 та 1)
- **gamma**: коефіцієнт знецінення (повинен бути між 0 та 1)

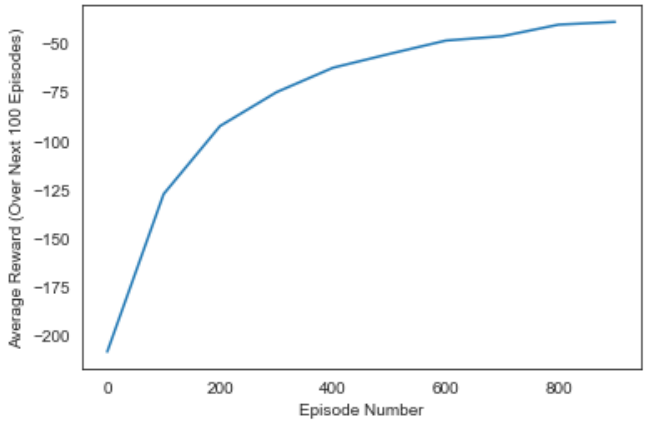
Та наша функція повертатиме масив значень Q, де $Q[s][a]$ – це оцінка дії, що відповідає стану s та дії a.

Проведемо декілька експериментів з різними параметрами та пізніше проведемо аналіз по кожному з них.

Експеримент №1 – класичний алгоритм Q-навчання

1. Спочатку запустимо середовище з 1000 епізодами та темпом навчання 0.01

```
Q_sarsamax = q_learning(env, 1000, .01)
Episode 1000/1000
```




```
Best Average Reward over 100 Episodes:  -38.92
```

Result is not passed

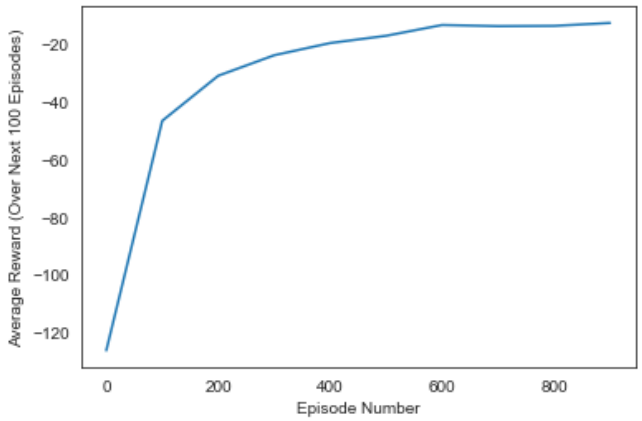
```
Estimated Optimal Policy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3, N/A = -1):
[[ 0  3  3  1  1  1  2  1  1  2  1  3]
 [ 0  3  1  1  1  0  0  3  1  1  1  2]
 [ 1  1  0  1  3  1  0  3  3  1  1  2]
 [ 3 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]]
```

Рис. 3.3. Експеримент з класичним алгоритмом Q-навчання для 1000 епізодів та темпу навчання 0,01

Можемо спостерігати на рис. 3.3., що 1000 епізодів та темпу навчання 0.01 недостатньо для досягнення прийнятного результату. Тому будемо змінювати один з параметрів. Спочатку спробуємо змінити темп навчання до 0.05

2. Середовище з 1000 епізодами та темпом навчання 0.05

```
Q_sarsamax = q_learning(env, 1000, .05)
Episode 1000/1000
```




```
Best Average Reward over 100 Episodes:  -13.06
```

Result passed

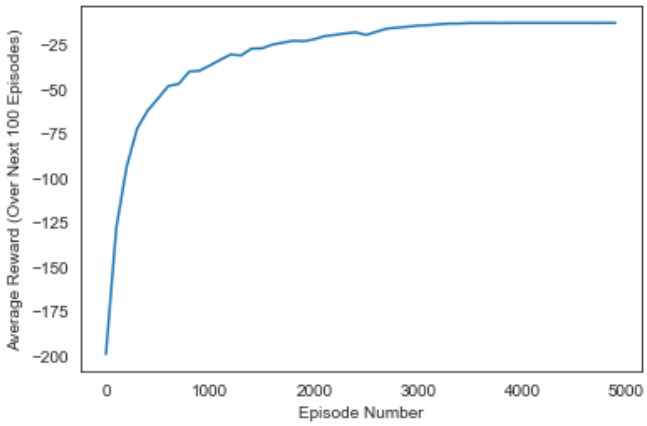
```
Estimated Optimal Policy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3, N/A = -1):
[[ 2  0  0  2  2  1  1  1  1  2  2  2]
 [ 1  1  0  1  1  1  0  1  1  2  1  2]
 [ 1  1  1  1  1  1  1  1  1  1  1  2]
 [ 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]]
```

Рис. 3.4. Експеримент з класичним алгоритмом Q-навчання для 1000 епізодів та темпу навчання 0,05

Як бачимо на рис 3.4. збільшення темпу навчання дало нам прийнятний результат вже за 1000 епізодів. Проте деколи збільшення цього параметра може призвести до того, що агент навчиться швидко, але погано. Спробуємо цього разу збільшити кількість епізодів.

3. Середовище з 5000 епізодами та темпом навчання 0.01

```
Q_sarsamax = q_learning(env, 5000, .01)
Episode 5000/5000
```




```
Best Average Reward over 100 Episodes:  -13.0
```

Result passed

```
Estimated Optimal Policy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3, N/A = -1):
[[ 1  1  1  2  1  0  0  1  0  1  2  1]
 [ 0  1  1  0  2  2  1  1  1  2  1  2]
 [ 1  1  1  1  1  1  1  1  1  1  1  2]
 [ 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]]
```

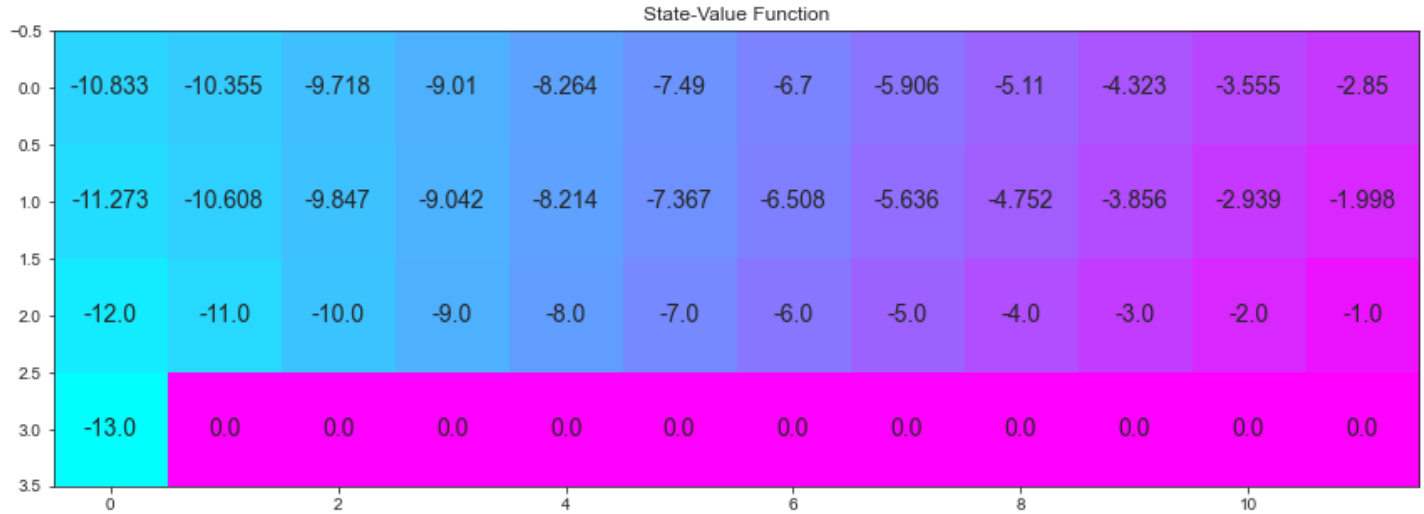


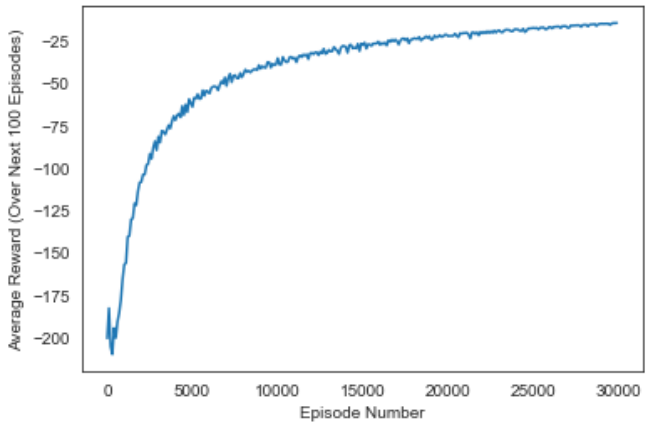
Рис. 3.5. Експеримент з класичним алгоритмом Q-навчання для 5000 епізодів та темпу навчання 0,01

Можемо бачити на рис 3.5., що при темпі навчання 0.01 достатньо близько 3000 епізодів, оскільки результат нагороди пізніше майже не змінюється, що видно по графіку.

Спробуємо збільшити кількість епізодів в 6 разів, при цьому зменшивши швидкість навчання в 10 разів.

4. Середовище з 30000 епізодами та кроком навчання 0.001

```
Q_sarsamax = q_learning(env, 30000, .001)
Episode 30000/30000
```




```
Best Average Reward over 100 Episodes:  -14.78
```

Result passed

```
Estimated Optimal Policy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3, N/A = -1):
[[ 2  0  1  1  0  0  1  3  1  0  3  2]
 [ 1  1  1  1  3  1  2  1  1  2  2  2]
 [ 1  1  1  1  1  1  1  1  1  1  1  2]
 [ 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]]
```

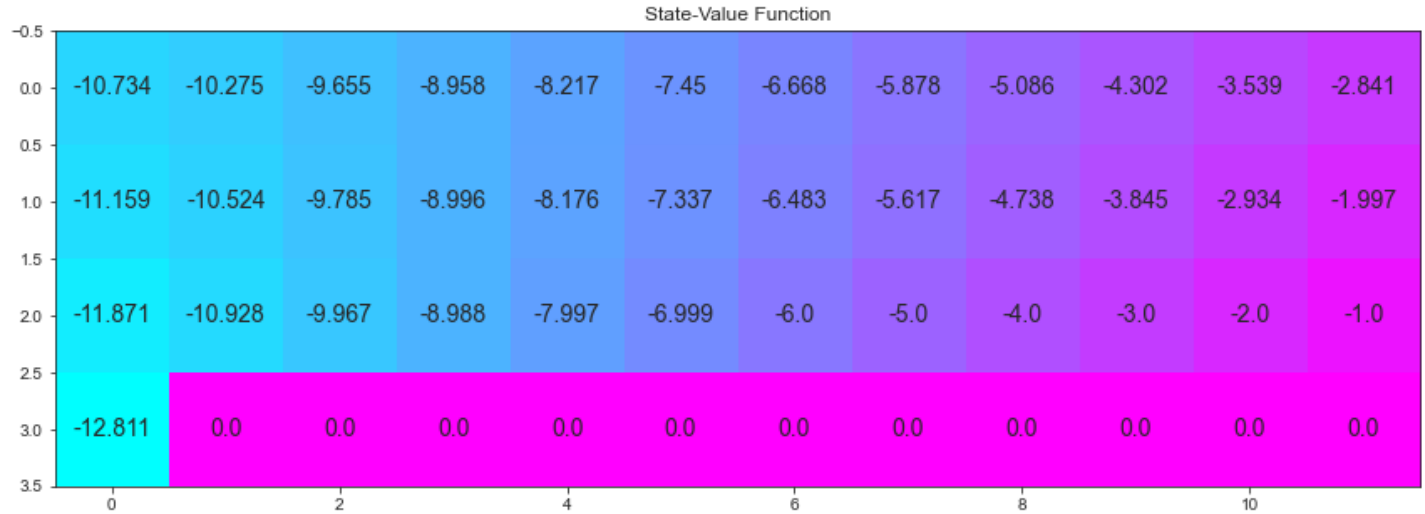


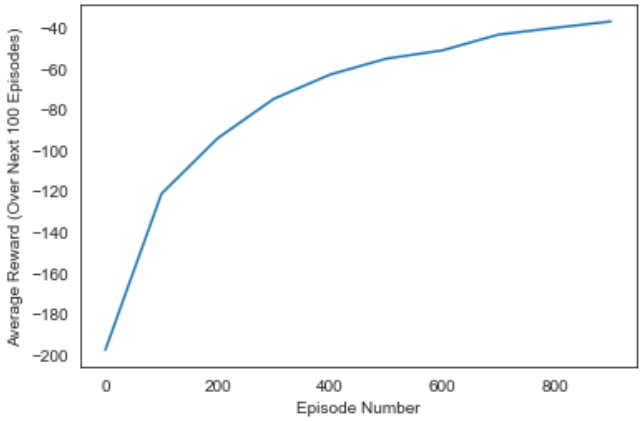
Рис. 3.6. Експеримент з класичним алгоритмом Q-навчання для 30000 епізодів та темпу навчання 0,001

Можемо спостерігати на рис. 3.6., що алгоритм досяг результату за 30000 епізодів при низькому темпі навчання. При цьому бачимо, що алгоритм має деяку просадку значень на початку навчання. Перевіримо поведінку інших алгоритмів.

Експеримент №2 – алгоритм SARSA.

1. Спочатку запустимо середовище з 1000 епізодами та темпом навчання 0.01

```
Q_sarsa = sarsa(env, 1000, 0.01)
Episode 1000/1000
```




```
Best Average Reward over 100 Episodes:  -37.3
```

Result is not passed

```
Estimated Optimal Policy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3, N/A = -1):
[[ 2  2  1  0  3  0  1  2  1  3  2  2]
 [ 1  3  3  3  1  3  1  3  0  0  0  2]
 [ 1  1  1  0  1  3  0  1  1  1  1  2]
 [ 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]]
```

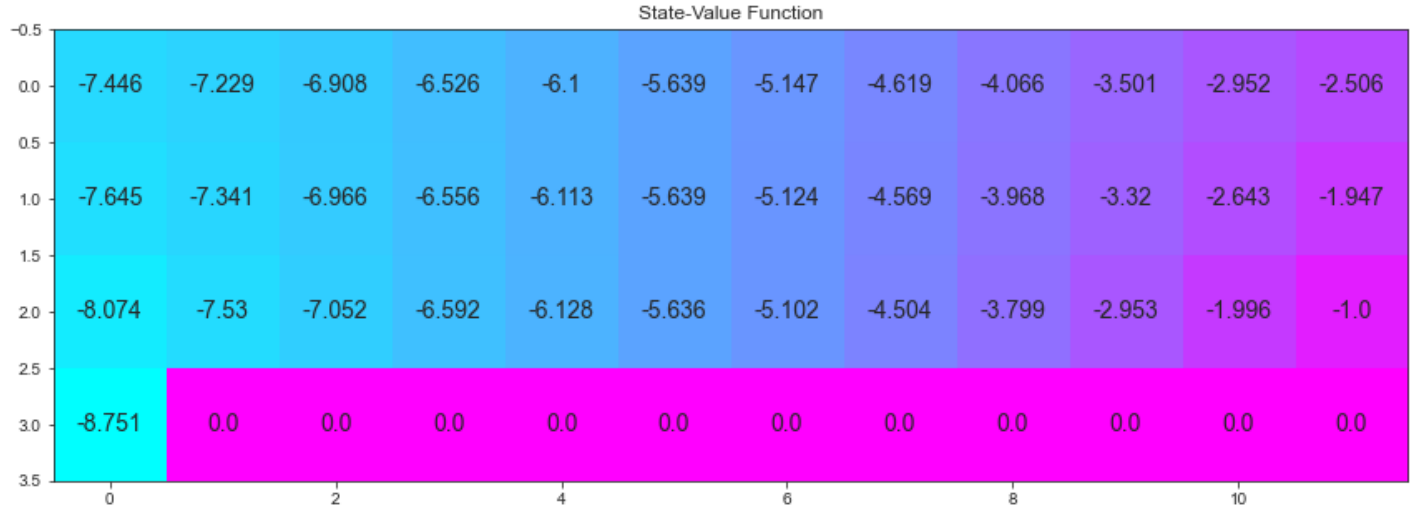


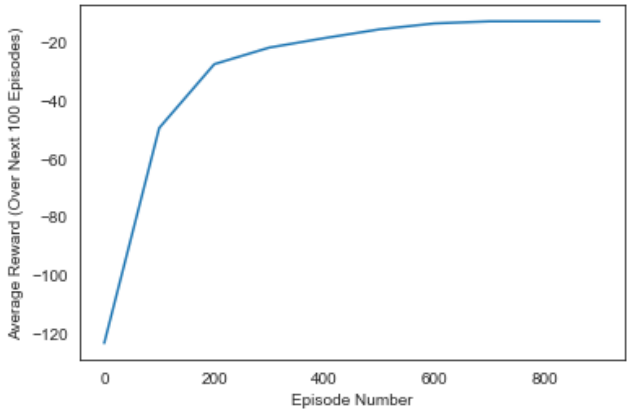
Рис. 3.7. Експеримент з алгоритмом SARSA для 1000 епізодів та темпу навчання 0,01

Як бачимо з рис. 3.7., при даних вхідних параметрах агент не встигає навчитись, тому досягає поганого результату нагороди -37.3.

Збільшимо швидкість навчання до 0.05 при тій же кількості епізодів.

2. Середовище з 1000 епізодами та темпом навчання 0.05

```
Q_sarsa = sarsa(env, 1000, 0.05)
Episode 1000/1000
```




```
Best Average Reward over 100 Episodes:  -13.0
```

Result passed

```
Estimated Optimal Policy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3, N/A = -1):
[[ 3  1  1  1  1  2  1  2  2  1  3  0]
 [ 2  1  1  1  1  1  1  1  2  1  1  2]
 [ 1  1  1  1  1  1  1  1  1  1  1  2]
 [ 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]]
```

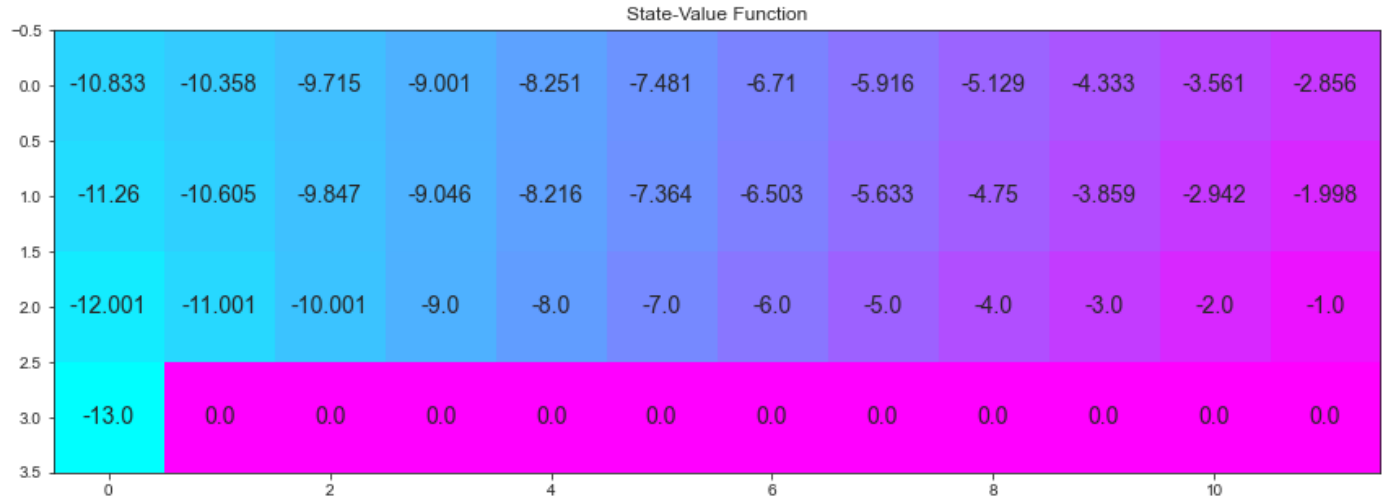


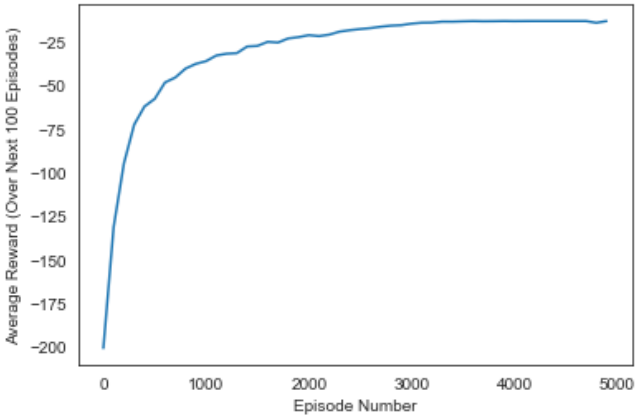
Рис. 3.8. Експеримент з алгоритмом SARSA для 1000 епізодів та темпу навчання 0,05

Як бачимо з рис. 3.8., з даними вхідними параметрами агент швидше навчається, тому вже за 1000 епізодів отримує найкращий результат винагороди -13.0.

Спробуємо зменшити темп навчання та збільшити кількість епізодів.

3. Середовище з 5000 епізодами та темпом навчання 0.01

```
Q_sarsa = sarsa(env, 5000, 0.1)
Episode 5000/5000
```




```
Best Average Reward over 100 Episodes:  -13.0
```

Result passed

```
Estimated Optimal Policy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3, N/A = -1):
[[ 1  1  0  1  1  1  1  0  1  2  2  2]
 [ 1  2  0  2  1  0  3  2  1  2  1  2]
 [ 1  1  1  1  1  1  1  1  1  1  1  2]
 [ 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]]
```

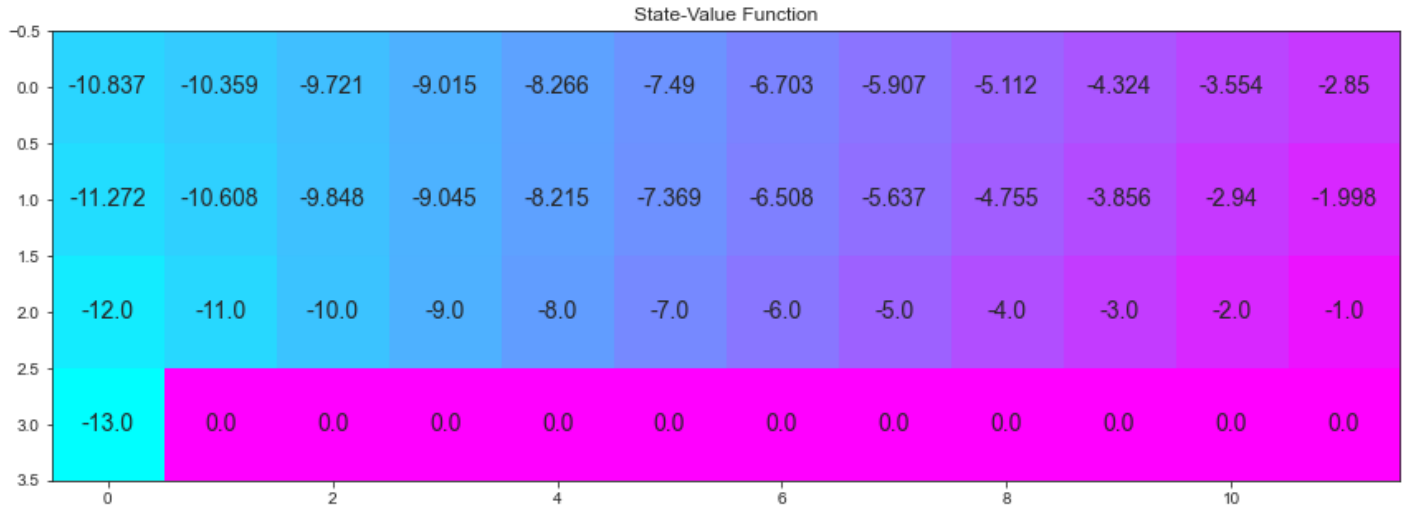
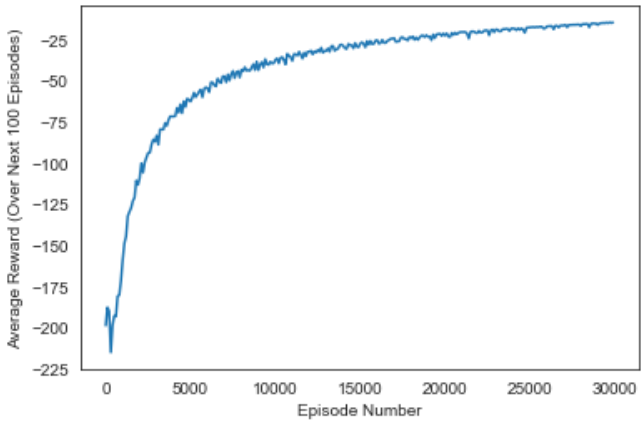


Рис. 3.9. Експеримент з алгоритмом SARSA для 5000 епізодів та темпу навчання 0,01

Як бачимо з рис. 3.9, при темпі навчання 0.01 агент досягає найкращого результату вже після близько 3000 епізодів.

4. Зменшимо швидкість навчання до 0.001 і збільшимо кількість епізодів до 30000.

```
Q_sarsa = sarsa(env, 30000, 0.001)
Episode 30000/30000
```




```
Best Average Reward over 100 Episodes:  -14.84
```

Result passed

```
Estimated Optimal Policy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3, N/A = -1):
[[ 1  1  2  1  1  1  1  2  1  1  1  2]
 [ 1  1  3  1  2  3  2  1  1  1  2  2]
 [ 1  1  1  1  1  1  1  1  1  1  1  2]
 [ 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]]
```

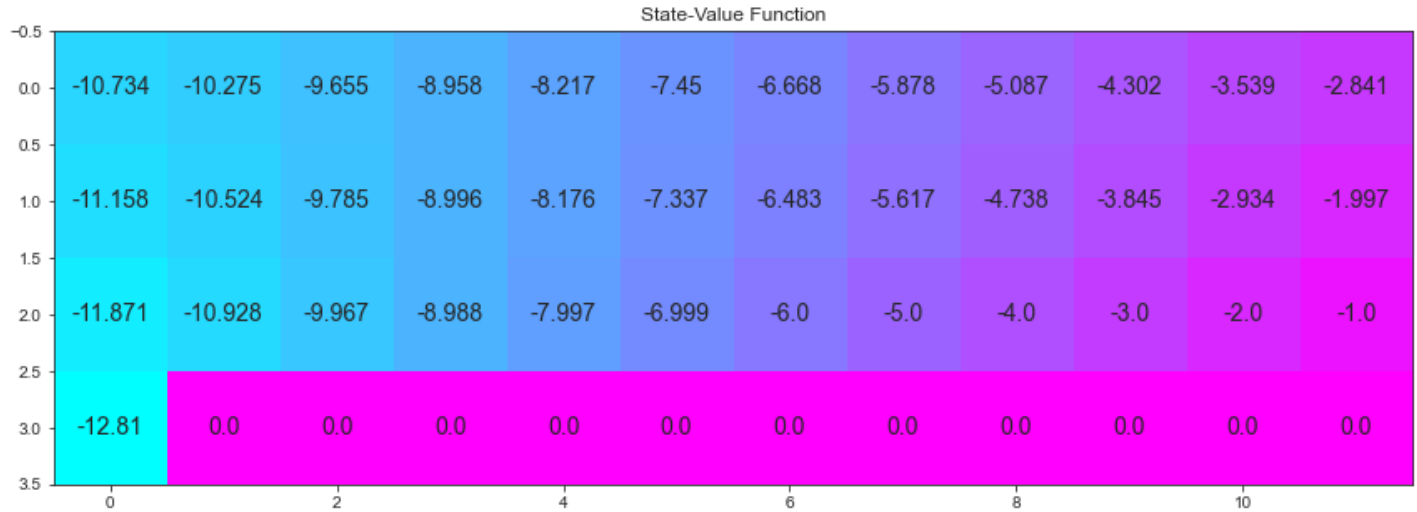


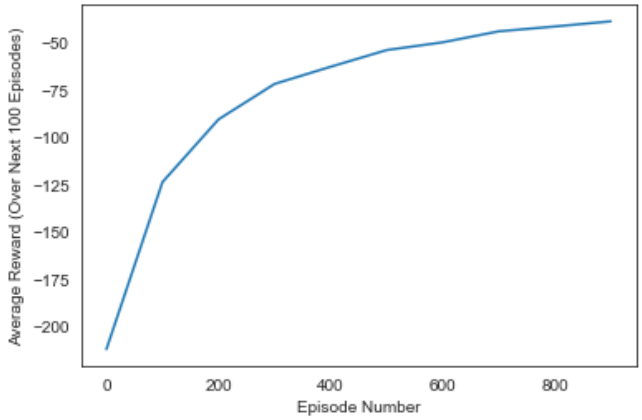
Рис. 3.10. Експеримент з алгоритмом SARSA для 30000 епізодів та темпу навчання 0,001

Алгоритм SARSA також отримує прохідний результат, але що цікаво, то він достатньо відрізняється від класичного Q-навчання. Проаналізуємо це детальніше в наступному розділі.

Експеримент №3 – алгоритм Expected SARSA.

1. Запустимо середовище з таким ж параметрами, як в попередніх експериментах

```
Q_expsarsa = expected_sarsa(env, 1000, 0.01)
Episode 1000/1000
```




```
Best Average Reward over 100 Episodes:  -38.68
```

Result is not passed

```
Estimated Optimal Policy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3, N/A = -1):
[[ 0  1  2  1  2  0  3  3  0  2  2  2]
 [ 1  2  0  2  1  1  3  2  2  2  1  2]
 [ 1  1  3  1  0  0  0  1  1  1  1  2]
 [ 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]]
```

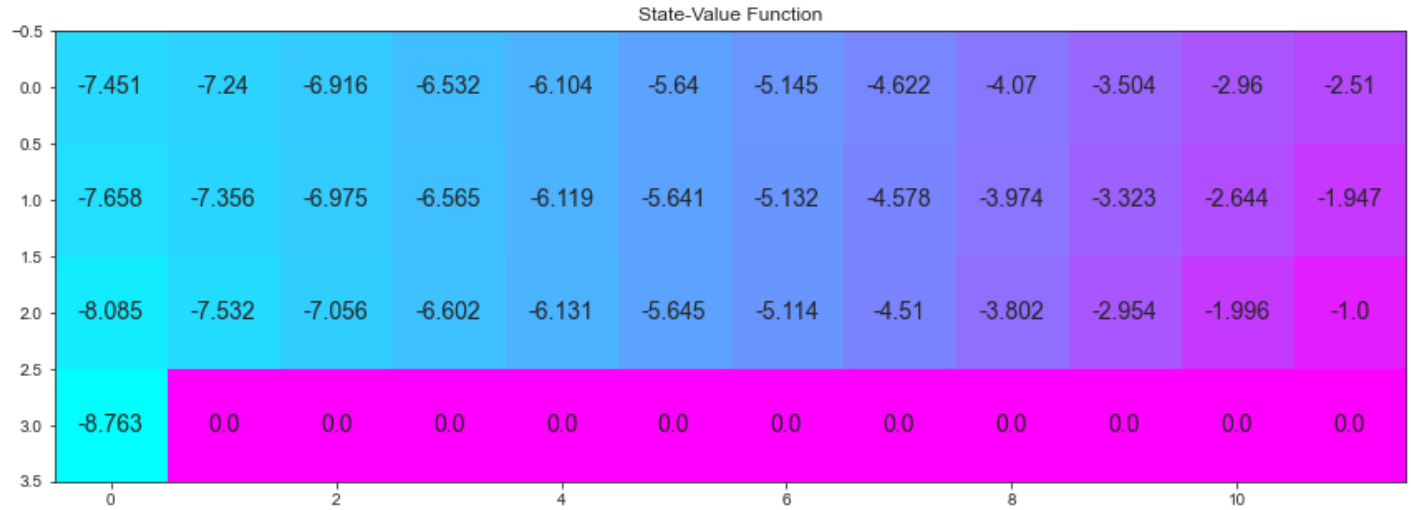
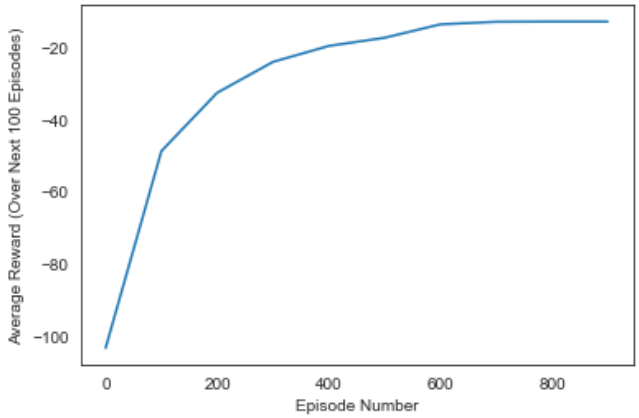


Рис. 3.11. Експеримент з алгоритмом Expected SARSA для 1000 епізодів та темпу навчання 0,01

Даний алгоритм також не справляється з даними вхідними параметрами навчання, тому будемо їх змінювати.

2. Середовище з темпом навчання 0.05 та 1000 епізодами.

```
Q_expsarsa = expected_sarsa(env, 1000, 0.05)
Episode 1000/1000
```




```
Best Average Reward over 100 Episodes:  -13.0
```

Result passed

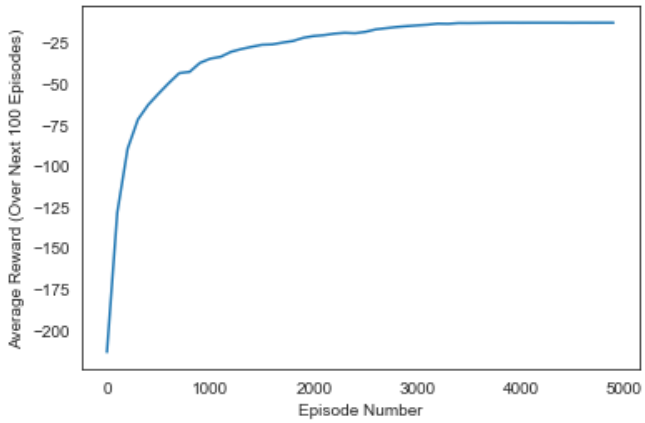
```
Estimated Optimal Policy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3, N/A = -1):
[[ 0  3  1  2  1  0  2  2  1  3  0  2]
 [ 3  1  1  1  1  1  2  1  1  2  2  2]
 [ 1  1  1  1  1  1  1  1  1  1  1  2]
 [ 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]]
```

Рис. 3.12. Експеримент з алгоритмом Expected SARSA для 1000 епізодів та темпу навчання 0,05

Алгоритм справляється так само добре, як і його аналоги. Тепер зменшимо темп навчання, при цьому збільшуючи кількість епізодів для навчання.

3. Середовище з кількістю епізодів 5000 та швидкістю навчання 0.01

```
Q_expsarsa = expected_sarsa(env, 5000, 0.01)
Episode 5000/5000
```




```
Best Average Reward over 100 Episodes:  -13.0
```

Result passed

```
Estimated Optimal Policy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3, N/A = -1):
[[ 1  0  1  1  1  1  1  2  1  1  1  2]
 [ 2  1  1  0  1  1  1  1  1  1  1  2]
 [ 1  1  1  1  1  1  1  1  1  1  1  2]
 [ 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]]
```

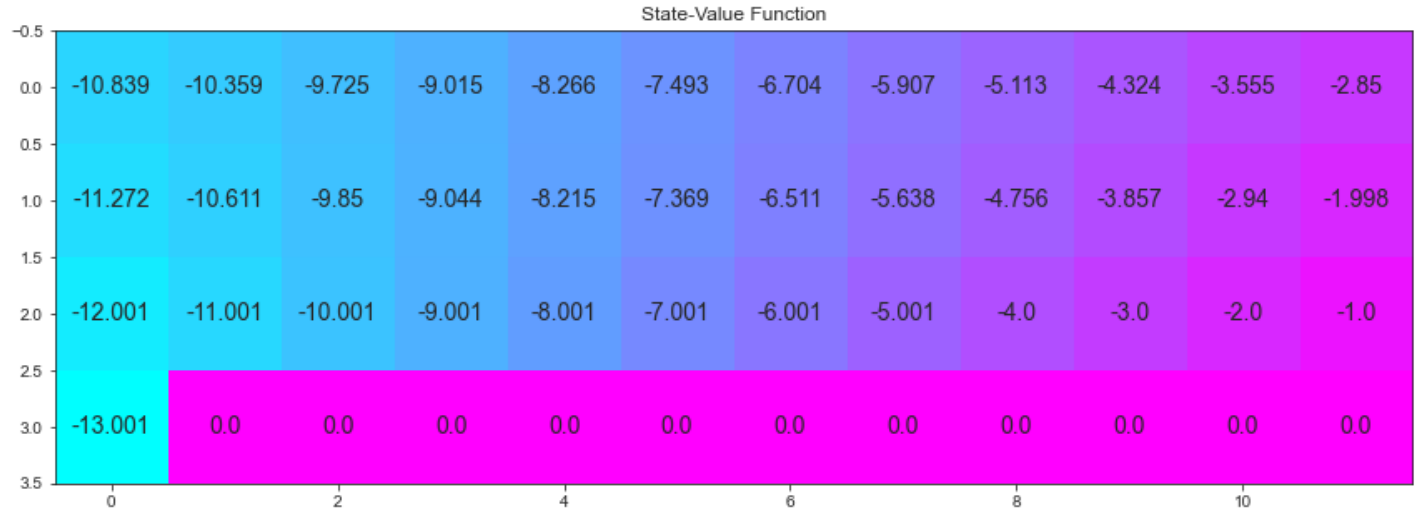
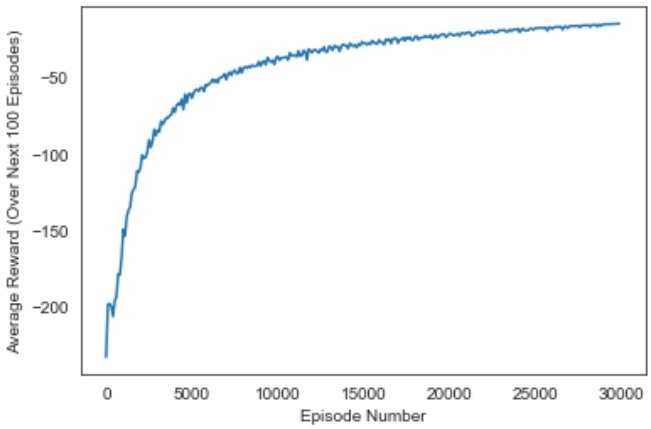


Рис. 3.13. Експеримент з алгоритмом Expected SARSA для 5000 епізодів та темпу навчання 0,01

Навіть при малому параметрі темпу навчання ми отримуємо хороший результат. При цьому бачимо, що оцінена оптимальна політика дещо відрізняється від інших методів.

4. Середовище з 30000 епізодами та темпом навчання 0.001

```
Q_expsarsa = expected_sarsa(env, 30000, 0.001)
Episode 30000/30000
```




```
Best Average Reward over 100 Episodes:  -14.75
```

Result passed

```
Estimated Optimal Policy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3, N/A = -1):
[[ 1  3  1  1  1  1  1  2  1  2  0  2]
 [ 0  3  2  1  1  1  1  1  1  1  1  2]
 [ 1  1  1  1  1  1  1  1  1  1  1  2]
 [ 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]]
```

Рис. 3.14. Експеримент з алгоритмом Expected SARSA для 30000 епізодів та темпу навчання 0,001

Можемо бачити, що алгоритм Expected SARSA не має подібної просадки в початкових епізодах, як інші алгоритми. А також можемо бачити, що оцінка оптимальної політики також відрізняється від попередніх результатів.

ОБГОВОРЕННЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

Для об’єктивного порівняння спочатку переглянемо графіки залежності середньої винагороди від кількості епізодів для усіх розглянутих методів.

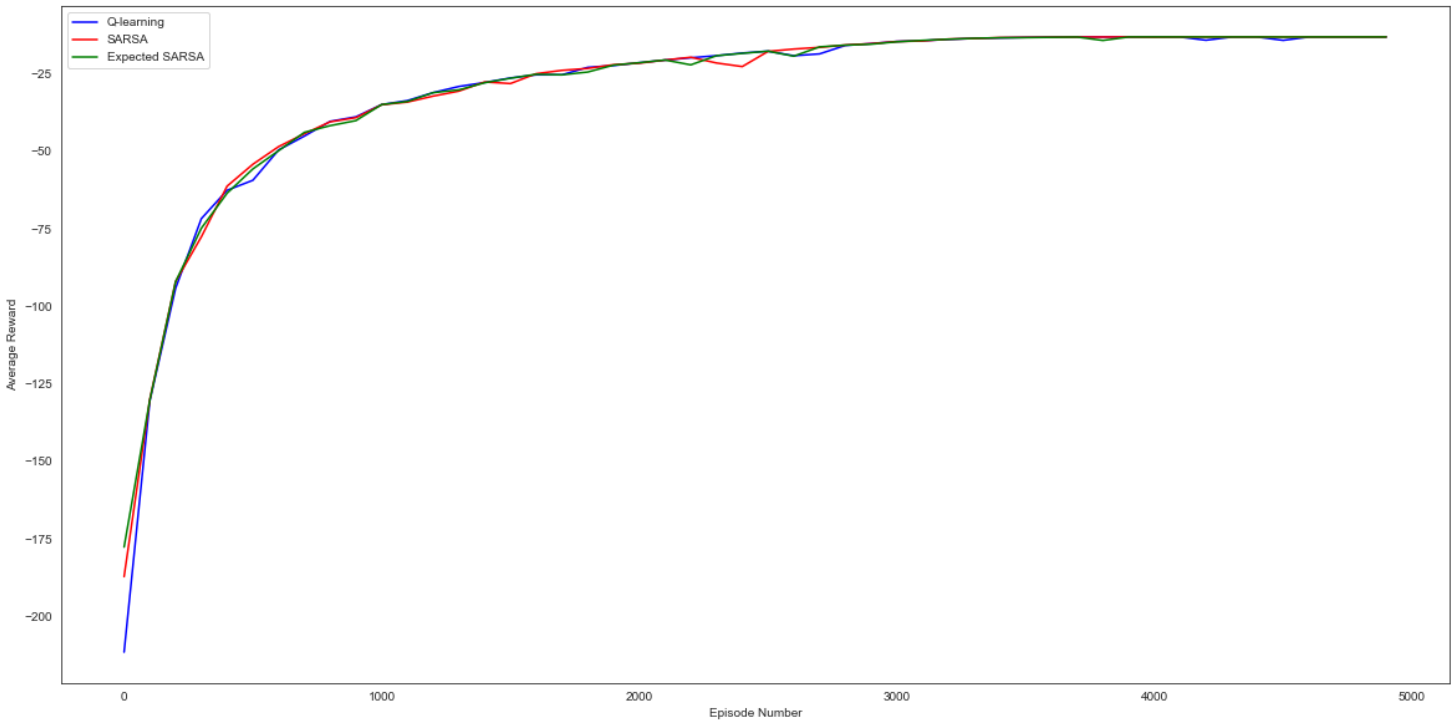


Рис. 4.1. Залежність розміру винагороди від кількості епізодів при темпі навчання 0.01 для 5000 епізодів.

З графіку можемо бачити, що алгоритми показують себе практично однаково з невеликими винятками. Проте Expected SARSA виглядає лідером, оскільки не містить подібних явних викидів, тому графік є досить плавним, без аномальних значень навіть на початку тренування.

Проаналізуємо навчання з іншими параметрами кількості епізодів та темпу навчання для наглядного порівняння.

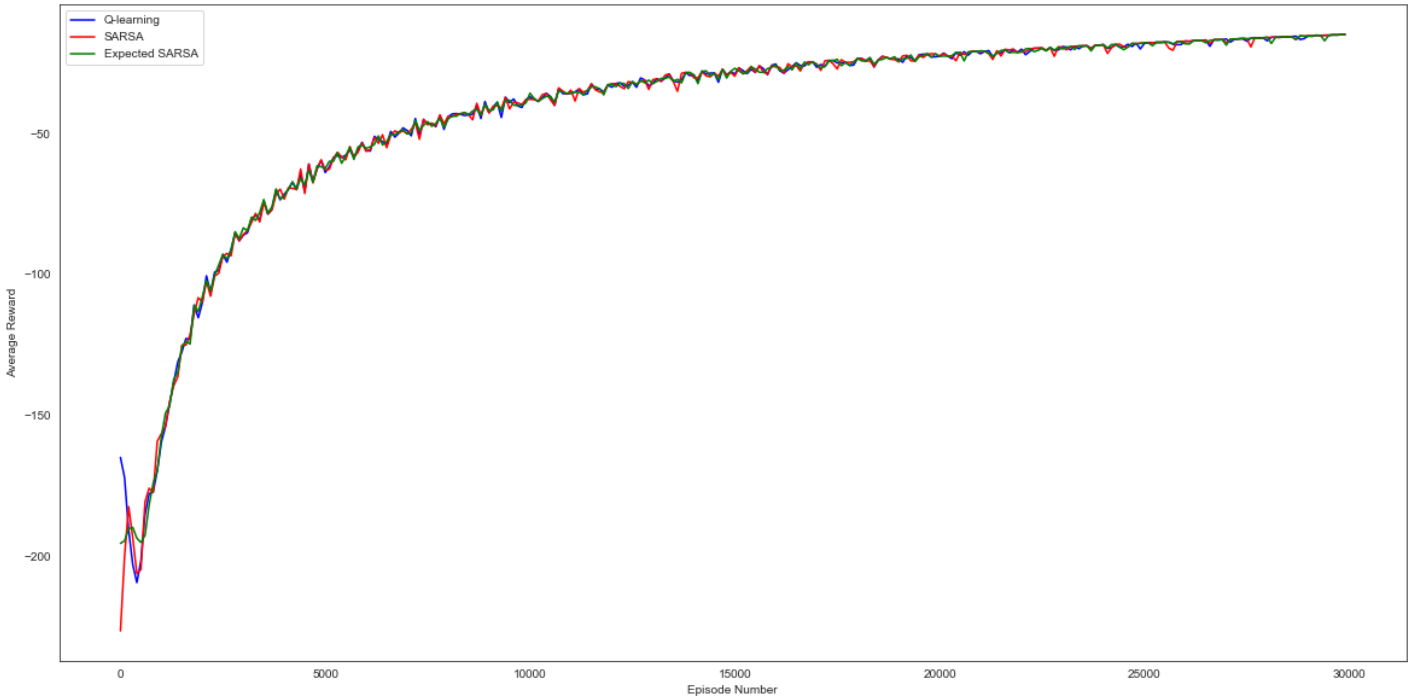


Рис. 4. 2. Залежність розміру винагороди від кількості епізодів при темпі навчання 0.001 для 30000 епізодів.

За даних параметрів добре видно різницю між методами. Цього разу найгірше проявляє себе SARSA, оскільки на початку тренування отримує найгірші значення, а також містить велику кількість викидів. Кращим є Q-навчання, проте тут зворотня проблеми до алгоритму SARSA, але цього разу к-сть викидів не така велика. Найкраще знову себе проявляє Expected SARSA.

Що стосується аналізу оптимального вибору шляху, то Q-навчання засвоює значення для оптимальної політики, що рухається прямо по краю скелі. Проте, на жаль, це призводить до того, що час від часу воно падає зі скелі через вибір жадібної стратегії. З іншого боку, SARSA враховує вибір дій, та вивчає довший, але більш безпечний шлях через верхню частину сітки. Таким чином невелика зміна в оновленні тимчасових різниць дає досить великі розбіжності в результатах. Щодо Expected SARSA, то алгоритм враховує недоліки SARSA, тому гарантує зменшення очікуваної помилки TD, цим і пояснюється стабільність графіку.

Таким чином може здатись, що Expected SARSA – найкращий алгоритм зі 3 представлених, проте в деяких випадках за інших параметрів Q-навчання та SARSA нічим не поступались у своїх результатах. Тому можна зробити висновок, що результат прямо залежить від правильного підбору параметрів, а також задачі, де алгоритм використовується.

ВИСНОВКИ

На даній курсовій роботі було розглянуто галузь машинного навчання, а саме навчання з підкріпленням. Досліджено методи Q-навчання, а саме алгоритми класичного Q-навчання, SARSA та Expected SARSA – on-policy та off-policy методи. Дані методи використовують різні стратегії оновлення Q-значення на основі тимчасових різниць TD.

Для проведення досліджень було обрано задачу Q-навчання в середовищі cliff walking. Дана проблема наглядно показує різницю між досліджуваними методами, що дозволяє зробити їхню порівняльну характеристику. Проте не завжди ці результати можуть бути показовими та зразковими для інших задач навчання з підкріпленням.

В результаті проведених досліджень встановлено, що від вибору правильних методів та алгоритмів прямо залежить результат навчання агентів. А від вибору коректних вхідних параметрів залежить не тільки темп навчання, а й його якість.

On-policy методи є корисними, коли потрібно оптимізувати значення для агента, який досліджує. Off-policy методи можуть бути корисні та ефективні в реальних сценаріях навчання з підкріпленням. Характеристика агента для дослідження та пошуку нових шляхів для задоволення функції винагороди робить його придатним для гнучких операцій. Для прикладу, роботизована рука, якій доручено малювати щось інше, ніж те, на чому вона навчена. Тобто мета в таких випадках – вчитись на ходу, для чого добре підходять дані алгоритми.

Проте off-policy методи також не позбавлені недоліків. Оцінка стає складною при великій кількості досліджень. Тому ці алгоритми можуть припускати, що метод оцінки, який не відповідає стратегії є точним при оцінці ефективності. Але агенти на основі минулого досвіду, можуть діяти зовсім інакше, ніж нові навчені агенти, що буде ускладнювати отримання хороших оцінок ефективності.

На основі даних експериментів можна заявити, що область машинного навчання з підкріпленням є відкритою та актуальною для подальших досліджень.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Asynchronous Methods for Deep Reinforcement Learning / [V. Mnih, M. Mirza, A. Graves та ін.]. // Nature. – 2016. – №7.
2. OpenAI Gym Docs [Електронний ресурс] – Режим доступу до ресурсу: https://gym.openai.com/docs.
3. Reinforcement learning [Електронний ресурс] – Режим доступу до ресурсу: https://www.cs.ox.ac.uk/people/nando.defreitas/machinelearning/lecture12.pdf.
4. Reinforcement learning [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Reinforcement_learning.
5. Learning from Delayed Rewards : дис. докт. техн. наук / . – Cambridge, 1989. – 241 с.
6. Microsoft Azure Service Fabric [Електронний ресурс]. – 2016. – Режим доступу до ресурсу: https://docs.microsoft.com/ru-ru/archive/msdn-magazine/2016/february/microsoft-azure-azure-service-fabric-q-learning-and-tic-tac-toe.
7. Lucas Vazquez. Reinforcement Learning (RL for the intimates) [Електронний ресурс] / Lucas Vazquez // Medium. – 2018. – Режим доступу до ресурсу: https://medium.com/@lgvaz/reinforcement-learning-rl-for-the-intimates-eef924cd5ee4.
8. Competitive AI system Dota 2 [Електронний ресурс] // OpenAI. – 2017. – Режим доступу до ресурсу: https://openai.com/blog/dota-2/.
9. OpenAI Five [Електронний ресурс] // OpenAI. – 2018. – Режим доступу до ресурсу: https://openai.com/blog/openai-five/.
10. Learning Dexterity [Електронний ресурс] // OpenAI. – 2018. – Режим доступу до ресурсу: https://openai.com/blog/learning-dexterity/.
11. Learning Dexterous In-Hand Manipulation [Електронний ресурс] // OpenAI. – 2019. – Режим доступу до ресурсу: https://arxiv.org/pdf/1808.00177.pdf.
12. Emergent Tool Use from Multi-Agent Interaction [Електронний ресурс] // OpenAI. – 2019. – Режим доступу до ресурсу: https://openai.com/blog/emergent-tool-use/.

13. EMERGENT TOOL USE FROM MULTI-AGENT AUTOCURRICULA [Електронний ресурс] // OpenAI. – 2019. – Режим доступу до ресурсу: https://d4mucfpksywv.cloudfront.net/emergent-tool-use/paper/Multi_Agent_Emergence_2019.pdf.
14. R. S. Sutton, and A. G. Barto, Reinforcement learning: an introduction. Cambridge: The MIT Press, 2015
15. C. Jin, Z. Allen-Zhu, S. Bubeck, and M. Jordan, "Is Q-learning Provably Efficient?", arXiv.org, 2018
16. Richard S. Sutton. Reinforcement Learning: An Introduction [Електронний ресурс] / Richard S. Sutton, Andrew G. Barto // Second edition, in progress – Режим доступу до ресурсу: https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf.
17. Environment for Cliff Walking [Електронний ресурс] // OpenAI Gym – Режим доступу до ресурсу: https://github.com/openai/gym/blob/a5a6ae6bc0a5cfc0ff1ce9be723d59593c165022/gym/envs/toy_text/cliffwalking.py.
18. Understanding Q-Learning, the Cliff Walking problem [Електронний ресурс] // Medium. – 2018. – Режим доступу до ресурсу: https://medium.com/@lgvaz/understanding-q-learning-the-cliff-walking-problem-80198921abbc.
19. CSE UNSW [Електронний ресурс] – Режим доступу до ресурсу: https://www.cse.unsw.edu.au/~cs9417ml/RL1/algorithms.html.
20. Expected SARSA in Reinforcement Learning [Електронний ресурс] // GeeksForGeeks – Режим доступу до ресурсу: https://www.geeksforgeeks.org/expected-sarsa-in-reinforcement-learning/.

Додаток А

Код створеного програмного продукту.

```
import sys
import gym
import numpy as np
from collections import defaultdict, deque
import matplotlib.pyplot as plt

import check_test
from plot_utils import plot_values

env = gym.make('CliffWalking-v0')

V_opt = np.zeros((4,12))
V_opt[0:13][0] = -np.arange(3, 15)[::-1]
V_opt[0:13][1] = -np.arange(3, 15)[::-1] + 1
V_opt[0:13][2] = -np.arange(3, 15)[::-1] + 2
V_opt[3][0] = -13

plot_values(V_opt)



def update_Q(Qsa, Qsa_next, reward, alpha, gamma):
    """ updates the action-value function estimate using the most recent
time step """
    return Qsa + (alpha * (reward + (gamma * Qsa_next) - Qsa))



def epsilon_greedy_probs(env, Q_s, i_episode, eps=None):
    """ obtains the action probabilities corresponding to epsilon-greedy
policy """
    epsilon = 1.0 / i_episode
    if eps is not None:
        epsilon = eps
    policy_s = np.ones(env.nA) * epsilon / env.nA
    policy_s[np.argmax(Q_s)] = 1 - epsilon + (epsilon / env.nA)
    return policy_s



def sarsa(env, num_episodes, alpha, gamma=1.0):
    # initialize action-value function (empty dictionary of arrays)
    Q = defaultdict(lambda: np.zeros(env.nA))
    # initialize performance monitor
    plot_every = 100
    tmp_scores = deque(maxlen=plot_every)
```

```
    scores = deque(maxlen=num_episodes)
    # loop over episodes
    for i_episode in range(1, num_episodes+1):
        # monitor progress
        if i_episode % 100 == 0:
            print("\rEpisode {}/{}".format(i_episode, num_episodes),
end="")
            sys.stdout.flush()
        # initialize score
        score = 0
        # begin an episode, observe S
        state = env.reset()
        # get epsilon-greedy action probabilities
        policy_s = epsilon_greedy_probs(env, Q[state], i_episode)
        # pick action A
        action = np.random.choice(np.arange(env.nA), p=policy_s)
        # limit number of time steps per episode
        for t_step in np.arange(300):
            # take action A, observe R, S'
            next_state, reward, done, info = env.step(action)
            # add reward to score
            score += reward
            if not done:
                # get epsilon-greedy action probabilities
                policy_s = epsilon_greedy_probs(env, Q[next_state],
i_episode)
                # pick next action A'
                next_action = np.random.choice(np.arange(env.nA),
p=policy_s)
                # update TD estimate of Q
                Q[state][action] = update_Q(Q[state][action],
Q[next_state][next_action],
                                            reward, alpha, gamma)
                # S <- S'
                state = next_state
                # A <- A'
                action = next_action
            if done:
                # update TD estimate of Q
                Q[state][action] = update_Q(Q[state][action], 0, reward,
alpha, gamma)
                # append score
                tmp_scores.append(score)
                break
        if (i_episode % plot_every == 0):
            scores.append(np.mean(tmp_scores))
    # plot performance

plt.plot(np.linspace(0,num_episodes,len(scores),endpoint=False),np.asarray
(scores), color="red")
```

```
    plt.xlabel('Episode Number')
    plt.ylabel('Average Reward (Over Next %d Episodes)' % plot_every)
    #plt.show()
    # print best 100-episode performance
    print(('Best Average Reward over %d Episodes: ' % plot_every),
np.max(scores))
    return Q


print("Q_sarsa = sarsa(env, 30000, 0.001)")
Q_sarsa = sarsa(env, 30000, 0.001)

policy_sarsa = np.array([np.argmax(Q_sarsa[key]) if key in Q_sarsa else -1
for key in np.arange(48)]).reshape(4,12)
check_test.run_check('td_control_check', policy_sarsa)
print("\nEstimated Optimal Policy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3,
N/A = -1):")
print(policy_sarsa)

V_sarsa = ([np.max(Q_sarsa[key]) if key in Q_sarsa else 0 for key in
np.arange(48)])



def q_learning(env, num_episodes, alpha, gamma=1.0):
    # initialize action-value function (empty dictionary of arrays)
    Q = defaultdict(lambda: np.zeros(env.nA))
    # initialize performance monitor
    plot_every = 100
    tmp_scores = deque(maxlen=plot_every)
    scores = deque(maxlen=num_episodes)
    # loop over episodes
    for i_episode in range(1, num_episodes+1):
        # monitor progress
        if i_episode % 100 == 0:
            print("\rEpisode {}/{}".format(i_episode, num_episodes),
end="")
            sys.stdout.flush()
        # initialize score
        score = 0
        # begin an episode, observe S
        state = env.reset()
        # limit number of time steps per episode
        for t_step in np.arange(300):
            # get epsilon-greedy action probabilities
            policy_s = epsilon_greedy_probs(env, Q[state], i_episode)
            # pick action A
            action = np.random.choice(np.arange(env.nA), p=policy_s)
            # observe R, S'
            next_state, reward, done, info = env.step(action)
```

```
            # add reward to score
            score += reward
            if not done:
                # update TD estimate of Q
                Q[state][action] = update_Q(Q[state][action],
np.max(Q[next_state]),
                                            reward, alpha, gamma)
                # S <- S'
                state = next_state
            if done:
                # update TD estimate of Q
                Q[state][action] = update_Q(Q[state][action], 0, reward,
alpha, gamma)
                # append score
                tmp_scores.append(score)
                break
        if (i_episode % plot_every == 0):
            scores.append(np.mean(tmp_scores))
    # plot performance

plt.plot(np.linspace(0,num_episodes,len(scores),endpoint=False),np.asarray
(scores), color="yellow")
    plt.xlabel('Episode Number')
    plt.ylabel('Average Reward (Over Next %d Episodes)' % plot_every)
    # plt.show()
    # print best 100-episode performance
    print(('Best Average Reward over %d Episodes: ' % plot_every),
np.max(scores))
    return Q


print("Q_sarsamax = q_learning(env, 30000, .001)")
Q_sarsamax = q_learning(env, 30000, .001)

policy_sarsamax = np.array([np.argmax(Q_sarsamax[key]) if key in
Q_sarsamax else -1 for key in np.arange(48)]).reshape((4,12))
check_test.run_check('td_control_check', policy_sarsamax)
print("\nEstimated Optimal Policy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3,
N/A = -1):")
print(policy_sarsamax)

plot_values([np.max(Q_sarsamax[key]) if key in Q_sarsamax else 0 for key
in np.arange(48)])



def expected_sarsa(env, num_episodes, alpha, gamma=1.0):
    # initialize action-value function (empty dictionary of arrays)
    Q = defaultdict(lambda: np.zeros(env.nA))
    # initialize performance monitor
```

```
    plot_every = 100
    tmp_scores = deque(maxlen=plot_every)
    scores = deque(maxlen=num_episodes)
    # loop over episodes
    for i_episode in range(1, num_episodes+1):
        # monitor progress
        if i_episode % 100 == 0:
            print("\rEpisode {}/{}".format(i_episode, num_episodes),
end="")
            sys.stdout.flush()
        # initialize score
        score = 0
        # begin an episode, observe S
        state = env.reset()
        # limit number of time steps per episode
        for t_step in np.arange(300):
            # get epsilon-greedy action probabilities
            policy_s = epsilon_greedy_probs(env, Q[state], i_episode)
            # pick action A
            action = np.random.choice(np.arange(env.nA), p=policy_s)
            # observe R, S'
            next_state, reward, done, info = env.step(action)
            # add reward to score
            score += reward
            if not done:
                # update TD estimate of Q
                policy_next_s = epsilon_greedy_probs(env, Q[next_state],
i_episode)
                Q[state][action] = update_Q(Q[state][action],
np.sum(np.multiply(policy_next_s, Q[next_state])),
                                            reward, alpha, gamma)
                # S <- S'
                state = next_state
            if done:
                # update TD estimate of Q
                Q[state][action] = update_Q(Q[state][action], 0, reward,
alpha, gamma)
                # append score
                tmp_scores.append(score)
                break
        if (i_episode % plot_every == 0):
            scores.append(np.mean(tmp_scores))
    # plot performance

plt.plot(np.linspace(0,num_episodes,len(scores),endpoint=False),np.asarray
(scores), color="green")
    plt.xlabel('Episode Number')
    plt.ylabel('Average Reward (Over Next %d Episodes)' % plot_every)
    # plt.show()
    # print best 100-episode performance
```

```
    print(('Best Average Reward over %d Episodes: ' % plot_every),
np.max(scores))
    return Q


print("Q_expsarsa = expected_sarsa(env, 30000, 0.001)")
Q_expsarsa = expected_sarsa(env, 30000, 0.001)

policy_expsarsa = np.array([np.argmax(Q_expsarsa[key]) if key in
Q_expsarsa else -1 for key in np.arange(48)]).reshape(4,12)
check_test.run_check('td_control_check', policy_expsarsa)
print("\nEstimated Optimal Policy (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3,
N/A = -1):")
print(policy_expsarsa)

plot_values([np.max(Q_expsarsa[key]) if key in Q_expsarsa else 0 for key
in np.arange(48)])
```