МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
„КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. ІГОРЯ СІКОРСЬКОГО”
НАВЧАЛЬНО-НАУКОВИЙ КОМПЛЕКС
„ІНСТИТУТ ПРИКЛАДНОГО СИСТЕМНОГО АНАЛІЗУ”
КАФЕДРА МАТЕМАТИЧНИХ МЕТОДІВ СИСТЕМНОГО АНАЛІЗУ

КУРСОВА РОБОТА

на тему:

Трейдер на основі мережі DeepMind

Виконали:
Барзій І.І., гр. КА-41
Задохін Д.В., гр КА-43
Мороз А.Я., гр. КА-43

Прийняв:
доцент Тимошенко Ю. О.

Оцінка:_____

Підпис:_____

Київ 2016

РОБОТУ ВИКОНАЛИ:

Барзій Ілля Ігорович, студент групи КА-41:
розділи 1-2, додаток, висновки (файли trader_game.py, qlearn.py).
Задохін Дмитро Володимирович, студент групи КА-43:
розділи 1−2, додаток (файли CSVReader.java, linear.java, kyrs.py).
Мороз Андрій Ярославович, студент групи КА-43:
вступ, розділ 1, висновки, додаток (файл specgram_demo.py).

РЕФЕРАТ

Дана робота складається з 68 сторінок. Складається зі вступу, основної частини, 10 ілюстрацій, 1 таблиці, висновків, списку використаних джерел та додатку.

Об’єкт дослідження: можливості мережі DeepMind для створення бота для гри на валютній біржі.

Мета і задачі дослідження. Метою курсової роботи є реалізація програмного продукту, який буде спроможний приймати участь у торгах на валютній біржі у реальному часі.

Ступінь впровадження. Даний програмний продукт можна впроваджувати у торги на рельних валютних біржах.

Галузь застосування. Можливості даної роботи можна використати при грі на валютних біржах.

Ключові слова: ТРЕЙДЕР, БІРЖА, DEEPMIND, НАВЧАННЯ З ГЛИБИННИМ ПІДКРІПЛЕННЯМ, СПЕКТРОГРАМА.

ЗМІСТ

ВСТУП

Суть роботи трейдера на валютній біржі полягає у покупці валюти за однією ціною та перепродаж їх за іншою, більш вищою. За рахунок різниці трейдер збагачується. Сучасна торгівля на біржі в основному здійснюється шляхом торгівельних інтернет-платформ. Ці програми дозволяють отримувати всю необхідну інформацію про ринок, таку як котування валют, новини, виставлені ордери.

Для написання бота, який брав би участь у торгівлі на біржі було обрано мережу, розроблену компанією DeepMind Technologies Limited, яка використовує алгоритм навчання з глибинним підкріпленням. Технічно вона використовує глибинне навчання на згортковій нейронній мережі, з новітньою формою Q-навчання, різновидом безмодельного навчання з підкріпленням. Сама компанія тестувала цю систему на відеоіграх, з поміж яких варто виділити ранні аркади на кшталт Space Invaders чи Breakout. Без внесення змін у власний код штучний інтелект починає розуміти як грати у гру, та, після певного часу гри, у деяких іграх (найпомітніше у Breakout), робить це більш ефективно, ніж це колись робила людина.

Метою курсової роботи є реалізація програмного продукту, який був би спроможний на основі даних котування валют приймати зважені рішення для участі у торгах на валютній біржі.

Для цього необхідно було знайти спосіб представлення даних, з якими могла б працювати дана мережа, написати емулятор біржі для навчання бота і, власне, навчити трейдер грати на цій біржі.

В подальшому планується приєднати навченого бота до торгівлі на реальній біржі у реальному часі.

1. ТЕОРЕТИЧНА ЧАСТИНА

1.1. Перетворення ряду котувань валют у спектрограми

DeepMind Q Learner на вхід приймає зображення і попіксельно його обробляє, тому нам потрібно візуалізувати наші дані. Було вирішено для кожного значення будувати спектрограму, що отримується з певної кількості попередніх значень. Спектрограми дозволяють нам відслідкувати як частоти сигналу змінюються з часом.

Спектрограма зазвичай створюється одним з двох способів: апроксимується, як набір фільтрів, отриманих із серії смугових фільтрів (це був єдиний спосіб до появи сучасних методів цифрової обробки сигналів), або розраховується за сигналом часу, використовуючи перетворення Фур'є.

Метод смугових фільтрів зазвичай використовується в аналоговій обробці для поділу вхідного сигналу на частотні діапазони.

Створення спектрограми за допомогою віконного перетворення Фур'є зазвичай виконується методами цифрової обробки. Проводиться цифрова вибірка даних в тимчасовій області. Сигнал розбивається на частини, які, як правило, перекриваються, і потім проводиться перетворення Фур'є, щоб розрахувати величину частотного спектра для кожної частини. Кожна частина відповідає вертикальній лінії на зображенні - значення амплітуди в залежності від частоти в кожен момент часу.

У своїй роботі ми будували спектрограми другим способом.

Оскільки маємо справу з дискретним сигналом, використовуємо **дискретне перетворення Фур’є [6]**.

Дискретне перетворення Фур'є (в англомовній літературі DFT, Discrete Fourier Transform) - це одне з перетворень Фур'є, широко застосовуваних у алгоритмах цифрової обробки сигналів (його модифікації застосовуються в стисненні звуку в MP3, стисненні зображень в JPEG і ін.), А також в інших областях, пов'язаних з аналізом частот в дискретно (наприклад, оцифрованому аналоговому) сигналі. Дискретне перетворення Фур'є вимагає в якості входу

дискретну функцію. Такі функції часто створюються шляхом дискретизації (вибірки значень з безперервних функцій).

Формула перетворення:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \qquad k = 0, \ldots, N-1$$

Обернене перетворення:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{\frac{2\pi i}{N} kn} \qquad n = 0, \ldots, N-1.$$

N - кількість значень сигналу, виміряних за період, а також кількість компонент розкладання;

$x_n, \quad n = 0, \ldots, N-1$ - виміряні значення сигналу, що є вхідними данними для прямого перетворення і вихідними для оберненого.

$X_k, \quad k = 0, \ldots, N-1$ - N комплексних амплітуд синусоїдальних сигналів, що складають вихідний сигнал; є вихідними даними для прямого перетворення і вхідними для зворотного; оскільки амплітуди комплексні, то по ним можна обчислити одночасно і амплітуду, і фазу.

$\frac{|X_k|}{N}$ – звичайна, дійсна амплітуда k-го синусоідального сигналу.

$\arg(X_k)$ - фаза k-го синусоїдального сигналу (аргумент комплексного числа).

k - індекс частоти. Частота k-го сигналу дорівнює $\frac{K}{T}$, T – період часу, протягом якого брались вхідні данні.

З останнього видно, що перетворення розкладає сигнал на синусоїдальні складові (які називаються гармоніками) з частотами від N коливань за період до одного коливання за період. Оскільки частота дискретизації сама по собі дорівнює N відліків за період, то високочастотні складові не можуть бути коректно відображені - виникає муаровий ефект. Це призводить до того, що друга половина з N комплексних амплітуд, фактично, є дзеркальним відображенням першої і не несе додаткової інформації.

Оскільки об’єм вибірки достатньо великий, є сенс використовувати швидке перетворення Фур’є, алгоритм швидкого обчислення дискретного перетворення Фур'є (ДПФ). Тобто, алгоритм обчислення за кількість дій, менше ніж $O(N^2)$, необхідних для прямого (за формулою) обчислення ДПФ. Іноді під швидким перетворенням Фур’є розуміється один з швидких алгоритмів, званий алгоритмом проріджування за частотою / часу або алгоритмом за основою 2, що має швидкість $O(Nlog(N))$.

1.2. Алгоритм роботи мережі Deep Q-Network

Deep Q-Network - це алгоритм навчання розроблений Google DeepMind щоб грати в ігри Atari [3]. Було продемонстровано, як комп’ютер навчився грати у відео ігри Atrai 2600 тільки стежачи за зображенням на екрані та отримуючи винагороди, коли ігровий рахунок збільшувався. Результат показав, що алгоритм здатний грати у багато ігор. В деякі іграх комп’ютер грав краще за людину. Цей алгоритм був застосований до 49 ігор та перевершив людину в половині з них.

Розглянемо гру *Breakout*. У цій гри ви керуєте платформою знизу екрану та маєте відбивати м’ячик, щоб знищити всі кубики у верхній частині екрану. Кожного разу як м’ячик торкається кубику, той зникає, а рахунок гравця збільшується - отримуєте винагороду.

Припустимо, ви захотіли навчити нейронну мережу грати в цю гру. Вона буде отримувати зображення екрану, а віддавати три дії: йти вправо, йти вліво або стріляти (запустити м’ячик). Треба зрозуміти, що треба робити за кожного екрану. Для цього потрібні приклади, та багато. Можна записувати ігри професіоналів, проте ми хочемо самі робити дії на основі відгуків гри – винагород або штрафів.

Це і є задача навчання з підкріпленням (reinforcement learning). Такий вид навчання є середнім між навчанням з вчителем та навчанням без вчителя. У разі навчання з вчителем є лише кінцева сукупність прецендентів - пар "стимул-

реакція", що називається навчальною вибіркою. На основі таких даних треба відновити залежність, тобто побудувати алгоритм, що може для будь-якого об’єкта видати доволі точну відповідь. При навчанні без вчителя, використовується система спонтанного навчання, тобто навчання відбувається без втручання експерементатора. У разі навчання з прікріпленням пари правильних вводів/виводів не представляються, а недостатньо оптимальні дії явно не виправляються. Інтерактивний продуктивність включає знаходження балансу між дослідженням (exploration) та використанням (exploitation). Проте, видаються винагороди. На основі них агент має навчитися поводитись у середовищі.

Існує декілька труднощів. Наприклад, коли кубик зникає, це не пов’язано з діями безпосередньо перед цим. Правильний вібір зроблено, коли платформу було правильно встановлено та м’ячик відіб’ється. Це - задача про призначення, тобто які саме з попередніх дій призвели до отримання винагороди та до якої межі.

Коли знайшлася стратегія для отримання певної винагороди, чи треба її притримуватися, чи шукати нову, що матиме більшу винагороду? У грі Breakout при запуску м’ячик частіше летить вліво, аніж вправо. Так, лишаючись в лівій стороні можна легко заробити деяку малу кількість балів. Це і є дилема дослідження-використання, тобто чи треба використовувати наявну стратегію, чи достіджувати інші, можливо кращі стратегії.

Модель навчання з підкріпленням схожа на модель навчання людини та тварин. Хвала батьків, оцінки в школі, заробітна платня - приклади винагород. саме тому важливо достліджувати дану проблему, а ігри - чудовий інструмент моделювання для знахождення нових стратегій.

Марковський процес вирішення:

Як формалізувати проблему навчання з підкріпленням? Найчастіше використовується процесс вирішення Маркова.

Нехай, ви - агент, що знаходиться в середовищі (наприклад у грі Breakout). У середовища є певний стан (наприклад позиція платформи, положення та

напрямок руху м'яча, наявність певного кубика тощо). Агент може виконувати певний набір дій у середовищі (наприклад, рухати платформу вправо та вліво) Ці дії інколи призводять до винагороди (збільшення рахунку у грі). Дії змінюють середовище та призводять до нового стану, де агент може знову зробити дію і так ділі. Правила, за якими ви обираєте ці дії називаються політикою. Середовище, взагалі кажучи, є стохастичним, тобто наступний стан може бути частково випадковим (наприклад, при втраті м’яча, новий запускається у випадковому напрямі).

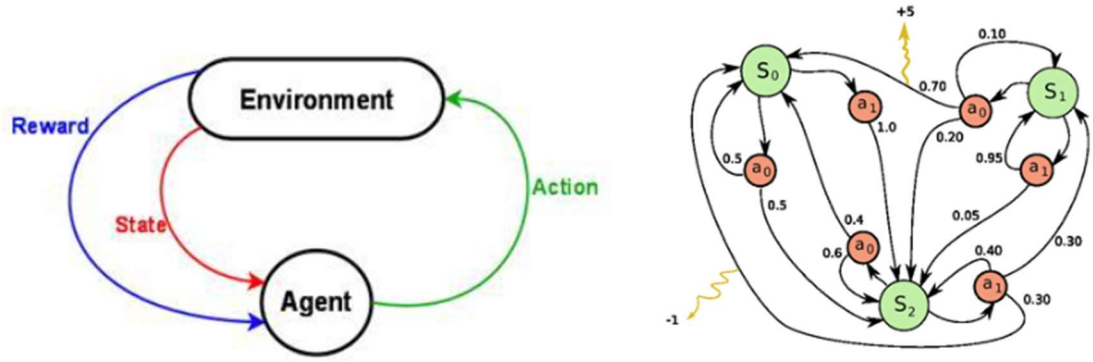

Рис. 1.1. Марковський процес вирішування

Набір станів та дій разом з правилами для переходу від одного стану до іншого створюють Марковський процес вирішування. Один епізод цього процесу (одна гра) формує нескінченну послідовність станів, дій та винагород.

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, \ldots, s_{n-1}, a_{n-1}, r_n, s_n$$

Si позначає стан, Ai - дію, а Ri+1 - винагороду після виконання дії. Епізод закінчується кінцевим (термінальним) станом Sn. Марковський процес вирішування спирається на припущенні Маркова, що ймовірність наступного стану Si+1 залежить тільки від поточнорго стану Si та дії Ai, але не від попередніх станів та дій.

Винагорода, затримана в часі:

Щоб довгострокові дії мали сенс, треба враховувати не тільки негайнівинагороди, а також майбуті, що отримаємо. Але як?

Маючи одне проходження Марковского процесу вирішування, ми можемо порахувати загальну винагороду за один епізод.

$$R = r_1 + r_2 + r_3 + \ldots + r_n$$

Маючи її, повна майбутня винагорода від моменту t та далі може бути виражена як.

$$R_t = r_t + r_{t+1} + r_{t+2} + \ldots + r_n$$

Але, оскільки середовище стохастичне, ми ніколи не можемо бути впевнені, що ми отримаємо ті ж самі винагороди, якщо будемо виконувати ті ж самі дії. Чим далі ми йдемо, тим більше че може різнитись. Через це зазвичай використовується винагорода, затримана в часі:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \ldots + \gamma^{n-t} r_n$$

Тут у -значення між 0 та 1 - чим далі в майбутньому винагорода, тим меньше вона враховується. Винагорода, затримана в часі у час t може бути виражена таким самим чином у час t+1.

$$R_t = r_t + \gamma(r_{t+1} + \gamma(r_{t+2} + \ldots)) = r_t + \gamma R_{t+1}$$

Якщо ми виставляємо значення у=0, наша стратегія буде короткозорою та ми будемо залежати тільки від негайних винагород. Якщо ми хочемо балансувати між гнегайними та майбутніми винагородами, беремо приблизно у=0.9. Якщо наше середовище детерміноване та однакові дії завжди призводять до однакових результатів, то можемо встановити у=1.

Гарною стратегією для агента буде завжди використовувати дію, що збільшує винагорорду, затриману в часі.

Q-learning:

В Q-learning ми визначаємо функцію Q(s,a), що представляє максимальну винагороду затриману в часі, коли ми виконуємо дію a у стані s.

$$Q(s_t, a_t) = max\ R_{t+1}$$

Q(s,a) можна вважати найкращім можливим рахунком в кінці гри після виконання дії a у стані s. Вона називається Q-function, бо вона представляє якість (Quality) певної дії у наданому стані.

Як ми можемо визначати рахунок в кінці гри, якщо знаємо тільки наданий стан та дію, але не дії, що слідують за нею? Ми не можемо. Але теоретично вважаємо коректність даної функції.

Як ми отримуємо функцію Q? Звернемо увагу на переході <s,a,r,s'>. Так само як винагорода, затримана в часі, можемо виразити значення Q у стані s та дії a через значення Q у наступонму стані s'.

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

Це називається рівнянням Белмана.

Основна ідея Q-learning полягає в тому, що ми можемо ітеративно апроксимувати функцію Q використовуючи рівняння Белмана. У найпростішому випадку функція Q виражається через таблицю, що має стани за рядки та дії за колонки.

```
initialize Q[num_states,num_actions] arbitrarily
observe initial state s
repeat
      select and carry out an action a
      observe reward r and new state s'
      Q[s,a] = Q[s,a] + α(r + γ max_a' Q[s',a'] - Q[s,a])
      s = s'
until terminated
```

Рис. 1.2. Основна ідея алгоритму Q-learning

Deep Q Network:

Стан середовища в грі Breakout може бути визначений через положення платформи, положенням та напрямом руху м’ячика та наявністю відповідного кубику. Однак це представлення залежить від гри. Чи можемо знайти щось більш універсальне, що підходило б під усі ігри? Очевидим вибором є пікселі на екрані, вони імпліментно мають всю актуальну інформацію про ситуацію у грі, окрім швидкості та напрямку руху м'яча. Два послідовних екрани матимуть також і цю інформацію.

Ми використовуємо однакові обробки ігрових екранів як у DeepMind - беремо чотири останні знімки екрану, перетворюємо її розмір у 84*84 та конвертуємо у чорно-білі зображення з 256 відтінками сірого. - тож маємо

$256^{84*84*4} \sim 10^{67970}$ можливих станів гри. Це означає 10^{67970} рядків в уявній Q-таблиці, більше за кількість атомів у відомому всесвіті. Навіть якщо вважати, що багато станів ніколи не з"являться через архітектуру гри, таблиця все одно дуже велика. До того ж, ми хотіли бі мати Q-значення у станах, що раніше не з’являлися.

Саме тут використовується deep learning. Нейронні мережі гарно знаходять рішення для високоструктурованих данних. Ми можемо представити нашу Q-функцію нейронною мережею, що приймає стан (чотири ігрових екрани) та дію на вхід, а виводить відповідне Q-значення. Або, ми можемо брати на вхід тільки ігрові екрани, а виводити Q-значення для кожної можливої дії. Цей підхід має перевагу, бо якщо хочемо змінити Q-значення або вибрати дію з найбільшим Q-значенням, то маємо лише раз пройтися по мережі та отримати усі Q-значення для усіх доступних дій негайно.

Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18

Табл 1.1. Архітектура нейронної мережі, що використовує DeepMind

Це класична згорткова нейронна мережа з трьома згортковими шарами.

Нейронна мережа приймає чотири 84*84 сірі ігрові екрани, а повертає Q-значення для будь-якої можливої дії (18 для Atari). Q-значення можуть бути будь-якими дійсними значеннями, що перетворює це на регресійну задачу, що може бути оптимізована звичайною квадратичною функцією втрати.

$$L = \frac{1}{2}[\underbrace{r + max_{a'}Q(s', a')}_{\text{target}} - \underbrace{Q(s, a)}_{\text{prediction}}]^2$$

Наданий перехід <s,a,r,s'>, правило на оновлення Q-таблиці має бути замінено з попередього алгоритму на:

1. Зробити прохід поточного стану s, щоб отримати передбачені Q-значення всіх векторів.

2. Зробити прохід наступного стану s' та порахувати максимальні виводи мережі max a' (s',a')

3. Встановити Q-значення для дії r+ y max a' Q(s',a') (Використовуючи максимум, вирахованиу на попередьому кроці, для усіх інших дій встановити Q-значення такими. що були отримані на першому кроці, встановлюючи для тих виводів помилку 0).

4. Оновити важелі методом зворотнього розповсюдження помилки.

Досвідчений програш:

Тепер ми знаємо, як оцінити майбутню винагороду у кожному стані використовуючи Q-learning та апроксимувати Q-функцію використовуючи згорткову нейронну мережу. Проте, виявляється, що апроксимація Q-значень використовуючи нелінійні функції не дуже стабільна. Є багато хитрощів, щоб згорнути таку мережу. Та це займає доволі багато часу.

Найважливіший трюк - досвідчений програш. Протягом гри всі переходи <s,a,r,s'> зберігаються в пам"яті програшу. Коли нейронна мережа тренується, випадкові міні-партії з пам"яті програшу використовуються замість останніх переходів. Це змінює однотипність тренувальних прикладів, що інакше приводили б мережу до локального мінімуму. Також, досвідченй програш перетворює завдання тренування більш схожим на звичайне навчання з вчителем, що спрощує пошук помилок та тестування алгоритму.

Дослідження-Використання:

Q-learning намагається розв'язати проблему розподілення довіри - воно розповсюджує винагороди назад в часі, аж поки не досягне критичного вибору, що був справжньою причиною отриманої винагороди. Але ми ще не розглядали дилему дослідження - використання.

Спочатку помітимо, що коли Q-таблиця або Q-мережа ініцалізовані випадковим чином, то її передбачення також випадкові. Якщо ми оберемо найбільше Q-значення , дія буде випадковою та агент виконує дослідження. По мірі того, як Q-функція згортається, вонаповертає все більш послідовні Q-значення та кількість досліджень зменшується. Так, можна казати, що Q-learning включає дослідження як частину алгоритму. Але це дослідження є жадібним, воно притримується першої ефективної стратегії, яку знайде.

Просте та ефективне виправлення для такої проблеми - епсілон-жадібне дослідження - з ймовірністю епсілон обирається випадкова дія, інакше використовується жадібна дія з найбільшим Q-значенням. У своїй системі DeepMind зменшує епсілон з часом - з 1 до 0.1 - на початку система виконує

абсолютно випадкові рухи, щоб максимально дослідити простір станів, а потім досліджує з фіксованим темпом.

Згорткові мережі – це декілька шарів згорток з нелінійними функціями, як ReLU або tanh, що застосовуються до результатів. У традиційній нейронній мережі ми зв'язуємо кожен нейрон вводу до кожного нейрону виводу у інший шар. Це також називається повністю зв'язаним шаром. У згорткових мережах, згортки застосовуються до вводного шару, щоб вирахувати вивід [8]. Це призводить до локальних зв'язків, де кожен регіон вводу пов”язаний з нейроном виводу. Кожен шар застосовує різні фільтри, зазвичай десятки тисяч та комбінує отримані результати. При тренуванні, згорткова нейронна мережа авто атично вивчає значення своїх фільтрів, в залежності від виконуваного завдання. Наприклад, при класифікації зображень, згорткова нейронна мережа може навчитися знаходити границі на зображенні з пікселів.

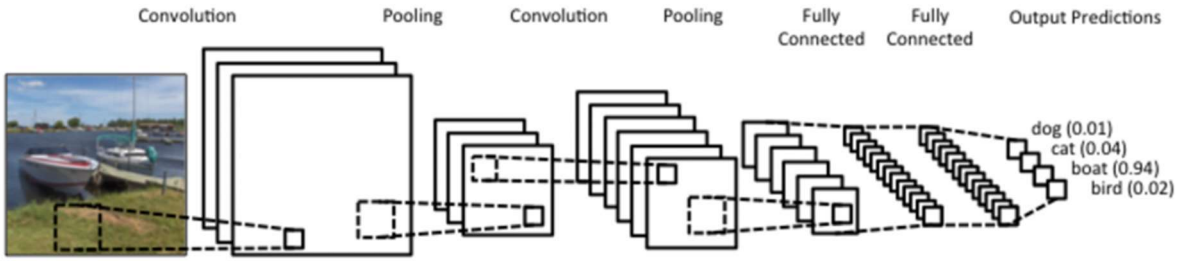

Рис. 1.3. Приклад згорткової нейронної мережі

Архітектура моделі:

Існує декілька шляхів параметризування Q, використовуючи нейронну мережу. Оскільки Q співставляє пари історія-дія скалярним наближенням їх Q-значень, історія та дії використовуються як ввід до нейронної мережі. Найголовніший недолік такого типу архітектури в тому, що необхідне окреме проходження щоб вирахувати Q-значення кожної дії, що прямо залежить від кількості дій. На відміну від цього, ми використовували архітектуру, де для кожної можливої дії є окремий вивід та вводом до нейронної мережі є тільки

представлення стану. Виводи відповідають передбаченим Q-значенням кожної індивідуальної дії для введеного стану. Найголовнішою перевагою такого типу архітектури є можливість вираховувати Q-значення для усіх можливих дій у заданому стані після лише одного проходу через енйронну мережу.

Ввід у мережу складається з зображення 257*13*3, де 257 – висота зображення в пікселях, 13 – довжина зображення в пікселях, 3 – розбиття зображення на кольори – червоний, зелений та синій. Перший шар згортає зображення 32 фільтрами з розміром 8*8 та кроком 4. Другий шар згортає зображення 64 фільтрами з розміром 4*4 та кроком 2. Треій шар згортає зображення 64 фільтрами з розміром 4*4 та кроком 1. Останній шар – повністю зв'язаний з одиничним виводом на кожну допустиму дію. В нашому випадку число допустимих дій – 3.

2. ПРАКТИЧНА ЧАСТИНА

2.1. Збір даних

Для навчання взято тікові значення відношення EUR/USD на проміжку з 2016.11.21 06:25:34 до 2016.11.21. 22:36:55.

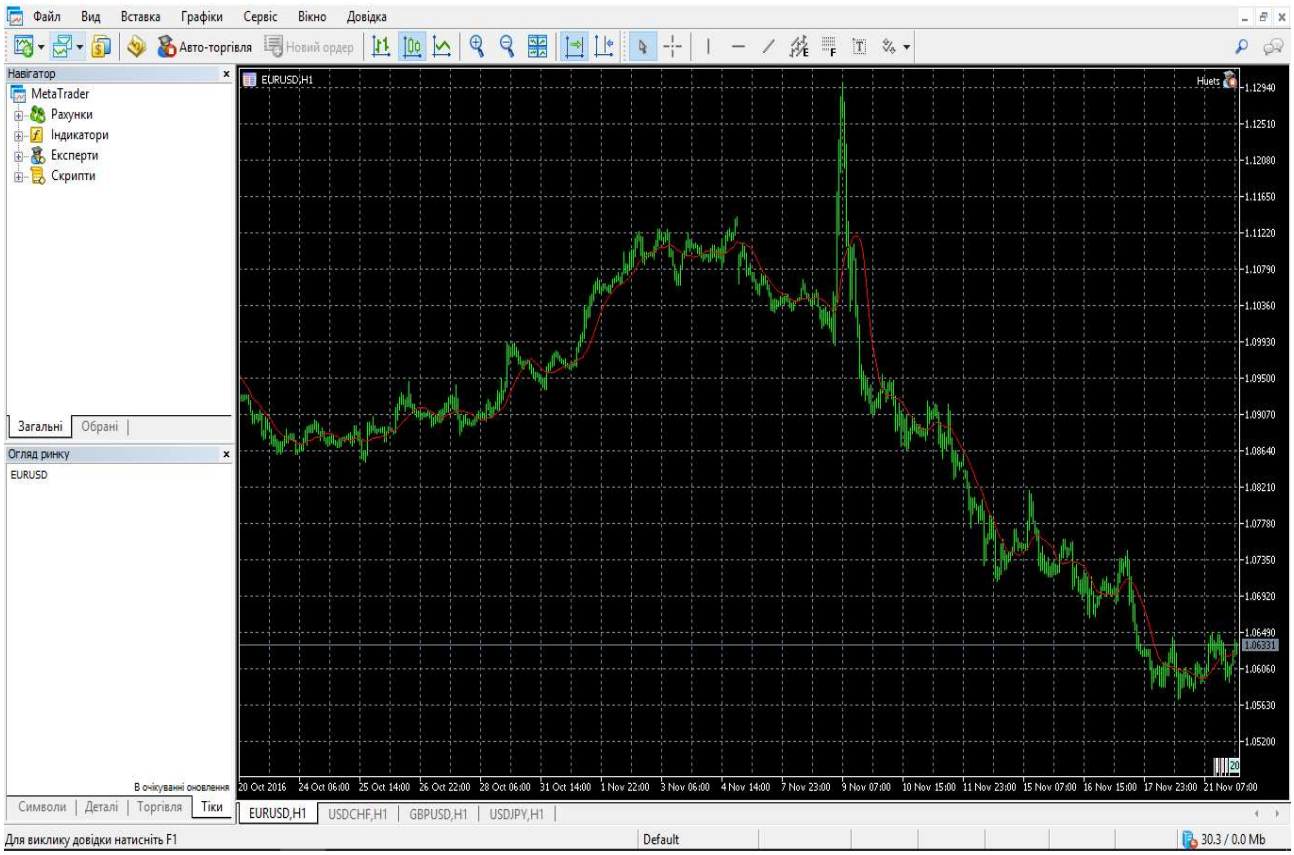

Рис. 2.1. Значення, отримані за допомогою сервісу MetaTrader5

Отриманий файл, EURUSD.csv, містив 100000 останніх тікових значень відношення валют.

Time,Bid,Ask,Last,Volume,Type
2016.11.21 06:25:34.282,1.05942,1.05952,1.05942,1000000,Bid
2016.11.21 06:25:34.382,1.05941,1.05952,1.05941,1735000,Bid
2016.11.21 06:25:34.531,1.05942,1.05952,1.05942,1000000,Bid
2016.11.21 06:25:34.636,1.05941,1.05952,1.05941,1735000,Bid
2016.11.21 06:25:34.745,1.05942,1.05952,1.05942,1000000,Bid
2016.11.21 06:25:34.836,1.05941,1.05952,1.05941,1735000,Bid
2016.11.21 06:25:35.047,1.05942,1.05952,1.05942,1000000,Bid
2016.11.21 06:25:35.155,1.05941,1.05952,1.05941,1735000,Bid
2016.11.21 06:25:35.244,1.05942,1.05952,1.05942,1000000,Bid
2016.11.21 06:25:35.356,1.05941,1.05952,1.05941,1735000,Bid

Рис. 2.2. Уривок файлу EURUSD.csv

Для того, щоб навчати нейронну мережу грати на біржі, потрібно щоб вона отримувала значення на рівномірних проміжках часу, чого не можна сказати про тікові значення. Тому виникла потреба інтерполювати функцію зміни відношення валют в часі і отримати ці значення з рівномірними часовими інтервалами. Для того, щоб отримати достатньо велику для ефективного навчання мережі кількість значень ми обрали секундний інтервал.

Щоб реалізувати дану інтерполяцію виникла потреба із файлу EURUSD витягнути не лише значення відношення валют, а й час, який відповідає кожному значенню.

Для того, щоб не зберігати два значення (купівля, продаж), було знайдено середнє значення, а на етапі використання, знаходити потрібну ціну додавши, чи віднявши значення спреда. Насправді, спред також змінюється у часі, але дуже в незначній мірі, тому було вирішено взяти стале значення.

Для зчитування файлу EURUSD та подальшої інтерполяції були використані засоби мови програмування Java. У ході реалізації виникла проблема з тим, що сервіс MetaTrader 5 при записі файлу через символ вставляв пустий символ, що унеможливило автоматичне перетворення рядків на числові значення. Тому було потрібно провести певні дії перед інтерполяцією (див. файл CSVReader.java у додатку).

Після роботи програми отримано два файли: з тіковими значеннями і кількістю мілісекунд, що пройшли після першої секунди, для якої маємо записи.

Vector.txt:

1.0594700000000001
1.0594649999999999
1.0594700000000001
1.0594649999999999
1.0594700000000001
1.0594649999999999
1.0594700000000001
1.0594649999999999
1.0594700000000001
1.0594649999999999
1.059475
1.0594700000000001
1.0594700000000001

…

Date.txt:

282
382
531
636
745
836
1047
1155
…

Для знаходження секундних значень було вирішено використати лінійну інтерполяцію, також виконано за допомогою мови Java (див. файл linear.java у додатку).

Отримано файл newvector.txt:

1.0594700000000001
1.0594688862559243
1.0594652536231883
1.0594658433734938
1.0594653198198196
1.0594698243243243
1.0594687142857144
1.059468686006826
1.059475076070901
…

Всього отримано 58 266 значень відношення EUR/USD з інтервалом в одну секунду.

2.2. Перетворення даних

Алгоритми перетворення Фур'є вже реалізовані в додаткових бібліотеках мови Python, ми скористались цими засобами, зокрема бібліотекою matplotlib (див. файл makespecgrams.py).

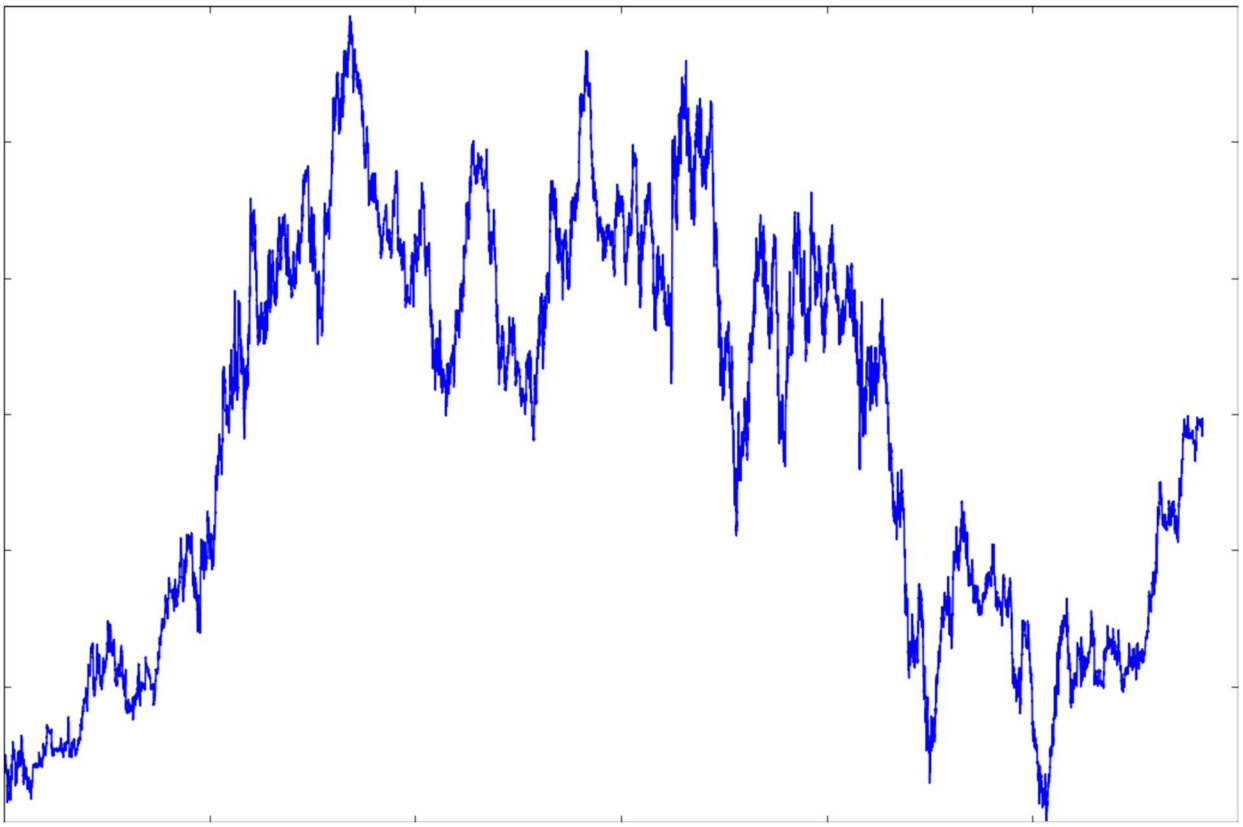

Рис. 2.3. Графік усього ряду

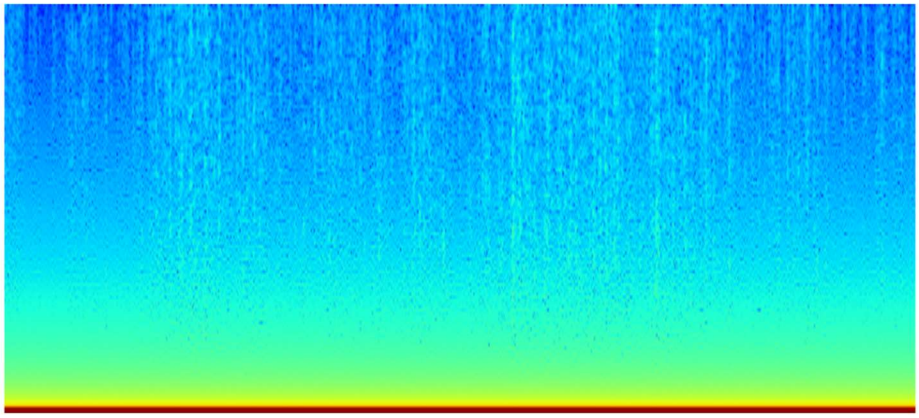

Рис. 2.4. Спектрограма усього ряду

Для наших цілей ми будували спектрограми для кожної секунди, що базувались на 1024 попередніх секундних значеннях.

Сигнал із цих 1024 значень разбивався на сегменти по 256 значень, для кожного з яких виконувалось швидке перетворення Фур'є. Далі, сегмент зміщувався на 64 значення. Тобто, всього створювалось 13 вертикальних послідовностей пікселів, що відповідали цим перетворенням. Для наочності

виведемо графіки сигналу на деяких проміжках, а також спектрограми, побудовані на них (в збільшеному вигляді).

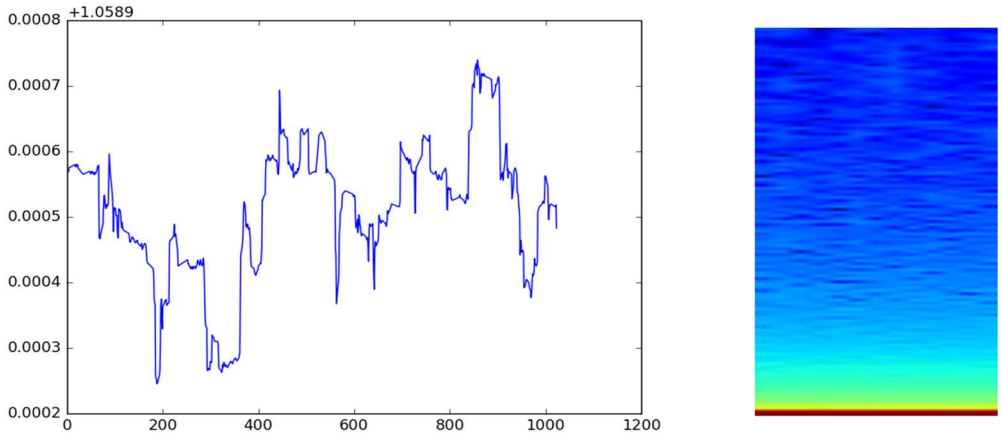

Рис. 2.5. Приклад ділянки графіку сигналу та відповідна спектрограма №1

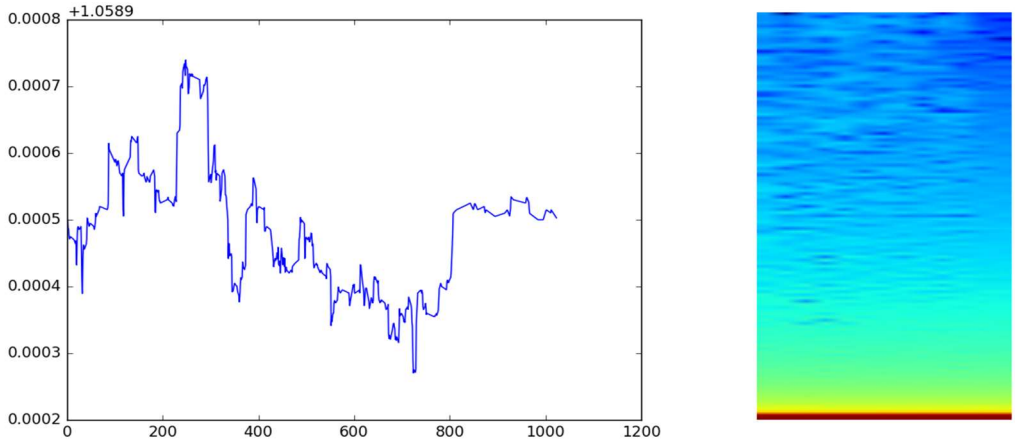

Рис. 2.6. Приклад ділянки графіку сигналу та відповідна спектрограма №2

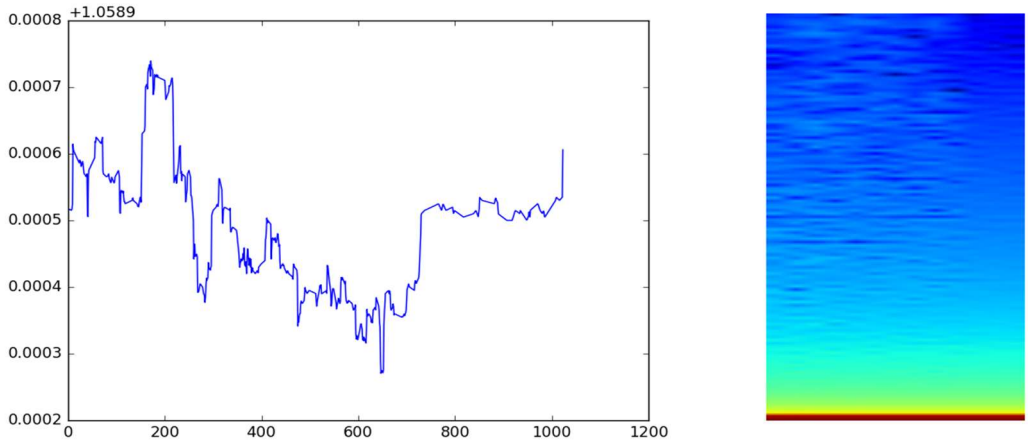

Рис. 2.7. Приклад ділянки графіку сигналу та відповідна спектрограма №3

2.3. Описання встановлення бібліотек і програм

Обчислення були виконані у системі Linux Ubuntu 16.0, Nvidia GTX 960m 4 gb, 8gb ram, i7-6700hq.

sudo apt-get install git-all

python get-pip.py

python -m pip install --upgrade pip

д

Для відстеження версій програми емулятора та програми запуску тренування нейронної мережі була встановлена система конторлю версій git.

sudo apt-get install python-pip python-dev build-essential

був встановлений менеджер пакетів для Python під назвою pip

pip install --user numpy scipy matplotlib ipython jupyter pandas sympy nose

За допомогою pip були встановлені такі бібліотеки як numpy, scipy, matplotlib, ipython, jupyter, pandas, sympy та nose. PyYAML була завантажена з офіційного ресурсу [7].

cd PyYAML-3.12/

python setup.py install

Далі, потрібно було встановити платформу для паралельного обчислення nvidia CUDA

g++ -V

dpkg --list | grep compiler

get http://www.netlib.org/blas/blas.tgz

sudo apt install gfortran

f77 -c -O3 *.f

ar rv libblas.a *.o

lspci | grep -i nvidia

sudo apt-get install linux-headers-$(uname -r)

sudo dpkg -i cuda-repo-ubuntu1604_8.0.44-1_amd64.deb

sudo apt-get update

sudo apt-get install cuda

Ці команди необхідні для перевірки того, що система задовольняє вимогам для встановлення nvidia CUDA, а також встанолена бібліотека blas для обчислення дій з матрицями. Після, nvidia CUDA була завантажена і встановлена.

export PATH=/usr/local/cuda-8.0/bin${PATH:+:${PATH}}

sudo apt-get install nvidia-367

cuda-install-samples-8.0.sh /home/illya/CUDA

cd NVIDIA_CUDA-8.0_Samples/

make

cd 1_Utilities/deviceQuery

./deviceQuery

Далі, були зроблені деякі налаштування та зібрані приклади для тестування коректності роботи nvidia CUDA. Тестування були пройдені успішно.

sudo pip install Theano

Був встановлений пакет Theano для підрахунків на графічному ядрі, а не на центральному процесорі.

THEANO_FLAGS=floatX=float32,device=gpu python /usr/lib/python2.*/site-packages/theano/misc/check_blas.py

Здійснено тестування коректності встановленного пакету BLAS

THEANO_FLAGS='mode=FAST_RUN,device=gpu,floatX=float32,optimizer_including=cudnn' python gpu_test.py

Так здійснилося тестування коректності підрахунку на графічному ядрі за допомогою пакету Theano

git clone https://github.com/fchollet/keras.git

sudo python setup.py install

sudo apt-get install python-pygame

sudo pip install scikit-image

Так був встановлений пакет keras для побудови нейронної мережі, pygame для написання емулятору біржі, scikit-image для обробки зображень.

Як виявилося, не вистачало декількох бібліотек, що й були встановлені

sudo apt-get install python-tk

sudo pip install tensorflow

sudo pip install h5py

sudo apt-get install build-essential libgl1-mesa-dev

Тепер, всі бібліотеки та програми були встановлені.

2.4. Написання емулятора біржі

Емулятор написаний на мові програмування Python з використанням бібліотек numpy, pygame, sys, random, os.

У цьому емуляторі гравцю виводяться спектрограми, що побудовані на основі минулих 1024 секунд. При виводі нової спектрограми є три можливих дії: “купити”, “продати”, або “нічого не робити”. Гравець не може мати відкритим декілька ордерів одночасно. Закриттю наявного ордера відповідає відкриття протилежного. Тобто дія “ закрити ордер купівлі” еквівалентна дії “ відкрити ордер продажу”. На вхід приймається тільки одна дія на спектрограму. Некоректні дії не виконуються.

На початку, визначаємо необхідні далі константи:

FPS =120

SCREENWIDTH = 13

SCREENHEIGHT = 257

Тут FPS визначає, з якою швидкістю зображення будуть передаватися до нейронної мережі. SCREENWIDTH та SCREENHEIGHT ширину та висоту ігрового екрану відповідно.

Такі розміри взяті через те, що отримані екундні спектрограми мають нестиснутий розмір 256*13. Ще один ряд пікселів використовується для виводу гравцю стану, в якому він зараз знаходиться. Зелений, якщо зараз відкритий ордер на продаж, червоний, якщо на данний момент ордерів не відкрито та синій, якщо зараз відкритий ордер на купівлю.

Було обрано використовувати саме кольори, адже нейронна мережа зможе їх сприймати та обробляти.

```
pygame.init()
FPSCLOCK = pygame.time.Clock()
SCREEN = pygame.display.set_mode((SCREENWIDTH, SCREENHEIGHT))
pygame.display.set_caption('Trader')
```

Наступні рядки створюють вікно, в яке будуть транслюватися спектрограмма та індикатор. Крім того, виставляється швидкість подачі на екран нових спектрограм.

```
class GameState:
  def __init__(self):
  self.money = 1000 # actual money
  self.multiplyer = 100 # 1:100 (Not used in lite version)
  self.orderprice = 0.1 #fixed price for order
  self.order = 0 # -1 :currently sell order 0 : currently no orders 1: currently buy order
  self.order_price = 0 #price at which order was opened
  self.spread = 0.000001 #obviously, spread value
  self.kick_price = 800 # At this balance your game ends
  path, dirs, files = os.walk("/home/illya/Trader/src/images/specgrams").next()
  self.max_frames = len(files)
  self.frame = random.randint(1024, self.max_frames-10000)
  with open("/home/illya/Trader/src/newvector.txt") as f:
```

```
self.prices = map(float, f)
```

Далі визначається клас GameState та його конструктор. При кожній новій грі клас буде створюватись заново.

При створенні визначаються змінні:

money – кількість коштів на рахунку гравця з початку;

multiplayer – кредитне плече гравця. При створенні більш досконалої версії емулятора, ця змінна визначатиме, якою частиною від грошей, що пішли на відкриття ордерів ще можна користуватися (відкривати нові ордери)

orderprice – визначає розмір ордеру на купівлю/продаж. Оскільки торгуємо на валютній парі, розмір одного лоту становить 1000 у.о. Тоді, розмір ордеру становитиме відповідно 100 у.о.

order – визначає, який ордер відкритий на данний момент, якщо відкритий. -1 означає, що відкритий ордер на продаж, 0 – немає відкритого ордеру та 1 – відкритий ордер на купівлю. На початку гри відкритого ордеру немає.

order_price – визначає, ціну, за якою данний ордер був відкритий, якщо відкритий.

Spread – значення спреду, тобто різниці в ціні, яку доводиться платити на біржі за здійснення транзакції. Спред являє собою різницю між ціною, за якою ми можемо відкрити ордер та закрити ордер.

Kick_price – значення рахунку, за якого гра вважається закінченою. Ми взяли значення у 800 у.о. Це становить 20% рахунку. Слід сказати, що верхньої межі рахунку немає.

max_frames визначає число спектрограм, що маємо. Ця змінна введена для того, щоб коректно обробляти ситуацію закінчення спекторграм. Значення цієї змінної обчислюється перерахунком усіх спектрограм, що містяться у відповідній папці.

Frame – визначає, яку саме спекторграму зараз використовувати. При створенні класу це значення береться довільним з наведеного інтервалу для того,

щоб мережа не перенавчилася – тобто не навчилася шрати тільки на даній фіксованій послідовності спектрограм.

Prices – масив, що містить значення курсу для кожної спектрограми.

```
    def frame_step(self, input_actions):
        pygame.event.pump()
        reward = 0
        terminal = False

        if sum(input_actions) != 1:
            raise ValueError('Multiple input actions!')
```

Далі, визначається функція frame_step , що приймає на вхід дію у вигляді масиву з трьох значень, одне з яких має бути 1, а інші 0.

reward – змінна, що визначає винагороду мережі за дану дію. Вона необхідна, адже використовуємо начання з підкріпленням. За замовчанняю, винагорода 0.

Та, робимо перевірку на коректність вводу дії.

```
        if input_actions[0] == 1:
            if self.order == 0: #Making sell order
            self.order = -1
            self.order_price = self.prices[self.frame]
      if self.order == 1: #Closing buy order
            buf = 1000 * self.orderprice * self.multiplyer *(self.prices[self.frame] - self.order_price -  self.spread) # Income/ Loss
            self.money += buf
            self.order = 0
            self.order_price = 0
            if buf > 0:
```

```
            reward = 0.5 # Got money
        if buf < 0:
            reward = -0.5# Lost money
```

Відповідно обробляємо інформацію про дію. Якщо у масиві перший елемент має значення 1 , то дія означає створення ордера на продаж або закриття ордеру на купівлю, другий елемент – 1, дія не потрібна, третій елемент 1 – створення ордеру на купівлю або закриття відкритого ордеру на продаж.

Точна дія визначається змінною order – якщо зараз не відкритий ордер, то він, відповідно створюється, у змінну order_price записується значення з prices, за якого ордер був відкритий.

Якщо ж зараз відкритий ордер на купівлю, то він закривається наступним чином: змінній buf присвоється зачення прибутку за ордер. Рахується як різниця між ціною на час закриття та відкриття, далі віднімається спред, отримане значення помножується на об’єм ордера, що закривається. До значення змінної money додається buf, значення order та order_price обнуляються. В залежності від прибутковості закритого ордеру визначається винагорода.

Якщо зараз відкритий ордер на продаж, то дія не обробляється, так як вважається некоректною.

Абсолютно аналогічно обробляється дія створення ордеру на купівлю або закриття ордеру на продаж.

```
buf = 0
if self.order == 1:
    buf = 1000 * self.order_price* self.multiplyer * ( self.prices[self.frame] - self.order_price - self.spread)
if self.order == -1:
    buf = 1000 * self.order_price* self.multiplyer * ( self.order_price - self.prices[self.frame] - self.spread)
```

```
buff = self.money + buf
    isCrash= (buff < self.kick_price)
```

Тепер, визначається, чи закінчена гра. Тобто, якщо на рахунку менше грошей, ніж потрібно для закриття відкритого ордера з урахуванням поточної ціни, або грошей менше ніж мінімальне значення рахунку, гра вважається закінченою та значення isCrash відбуває відповідно значення true або false.

```
    if isCrash:
        terminal = True
        self.__init__()
reward = -1
```

Так обробляється випадок закінчення гри. Змінна terminal приймає значення true, коли гра закінчена. Тоді, заново викликається конструктор класу. Винагорода за програну гру стає відповідною.

```
isOutOfFrames = (self.frame + 2 > self.max_frames)
if isOutOfFrames:
    self.frame = random.randint(1024, self.max_frames-10000)
```

Наступний код обробляє випадок закінчення спектрограм. Якщо зараз маємо передостанню спектрограму, змінна isOutOfFrames набуває значення true та наступна спектрограма обирається випадковим чином за доступних.

```
IMAGE_PATH = '/home/illya/Trader/src/images/newspecgrams/img' + str(self.frame) + '.png'
    current_screen = pygame.image.load(IMAGE_PATH).convert()
        SCREEN.blit(current_screen, (0,0))
```

Далі, завантажується наступна спектрограма та відображається у ігровому вікні.

Спектрограма знаходиться за відповідною змінною frame.

showState(self)

Це функція, що виводить відповідний індикатор про стан. Індикатор заємає останній рядок за 13 пікселів. Описується наступним чином:

```
def showState(self):
   if self.order == -1: #green indicator
  pygame.draw.line(SCREEN, GREEN, [0, 256], [12, 256], 1)
   if self.order == 0: # red indicator
  pygame.draw.line(SCREEN, RED, [0, 256], [12, 256], 1)
   if self.order == 1: # blue indicator
  pygame.draw.line(SCREEN, BLUE, [0, 256], [12, 256], 1)
image_data = pygame.surfarray.array3d(pygame.display.get_surface())
```

Змінній image_data надається теперешній ігровий екран у вигляді матриці з пікселів, що реперезентовані масивом з трьох значень – RGB спектр.

```
self.frame += 1
  ac = self.money
       return image_data, reward, terminal, ac
```

І, нарешті, збільшуємо номер спектрограми на 1.

Функція повертає ігровий екран, винагороду, прапорець про закінчення гри та поточний рахунок, щоб відстежувати прогресс навчання.

2.5. Використання Keras та Deep Q-Network для гри на біржі

Необхідні пакети та програми:

- python 2.7
- Keras 1.0
- pygame
- scikit-image
- Cuda with cuDNN
- Theano для GPU або TensorFlow для CPU
- та деякі інші

Запуск:

CPU:

git clone https://github.com/PanPip/Trader.git

cd Trader

python qlearn.py -m "Run"

GPU:

git clone https://github.com/PanPip/Trader.git

cd Trader

KERAS_BACKEND=theano

THEANO_FLAGS='floatX=float32,device=gpu0,lib.cnmem=0.2' python qlearn.py -m "Run"

lib.cnmem=0.2 означає, що буде використовуватись 20% відеопамяті для програми.

Якщо ви хочете тренувати мережу з самого початку, видаліть "model.h5" та замініть "Run" на "Train"

Алгоритм Deep Q-learning:

Розглянемо qlearn.py

```
initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
repeat
      select an action a
            with probability ε select a random action
            otherwise select a = argmax_a' Q(s,a')
      carry out action a
      observe reward r and new state s'
      store experience <s, a, r, s'> in replay memory D

      sample random transitions <ss, aa, rr, ss'> from replay memory D
      calculate target for each minibatch transition
            if ss' is terminal state then tt = rr
            otherwise tt = rr + γmax_a' Q(ss', aa')
      train the Q network using (tt - Q(ss, aa))^2 as loss

      s = s'
until terminated
```

Цей код виконує наступні дії:

1. Отримує Ігровий екран у формі масиву пікселів.

2. Обробляє зображення.

3. Оброблене зображення передається до нейронної мережі (Convolution Neural Network), а мережа тоді обирає найкращу дію (купити, продати або нічого не робити).

4. Мережа тренується багато разів через алгоритм, що називається Q-learning, щоб збільшити майбутню винагороду.

1 - Отримання ігрового екрану

Емулятор біржі trader_game вже написаний на Python через pygame, так що дістаємо код trader_game API.

import trader_game as game

x_t1_colored, r_t, terminal = game_state.frame_step(a_t)

Наступним рядком даємо грі на вхід дію a_t (-1 - продати; 0 - нічого не робити; 1 - купити), API видасть наступне зображення ігрового екрану x_t1_colored, винагороду (-0.1 - якщо угода збиткова; 0.1 - якщо угода прибуткова; -1 - якщо на рахунку лишилося замало коштів) та terminal - флаг, що визначає чи гра закінчена. Ми скористалися порадою DeepMind про значення винагород у проміжку від -1 до 1 щоб покращити стійкість системи. Поки що тестування з іншими винагородами не проводилося.

Винагороди можуть бути змінені у game/trader_game.py", у функції **def frame_step(self, input_actions)

2 - Обробка зображення

Щоб зробити тренування швидшим треба дещо обробити зображення ігрового екрану. Зроблено насупні дії:

1. Зображення зроблені чорно-білими

2. Зображення урізаються до розмірів 80x80

3. Зображення групуються по 4 перед тим, як передаються у нейронну мережу

```
x_t1 = skimage.color.rgb2gray(x_t1_colored)
x_t1 = skimage.transform.resize(x_t1,(80,80))
x_t1 = skimage.exposure.rescale_intensity(x_t1, out_range=(0, 255))

x_t1 = x_t1.reshape(1, 1, x_t1.shape[0], x_t1.shape[1])
s_t1 = np.append(x_t1, s_t[:, :3, :, :], axis=1)
```

x_t1 - одне зображення з розміром 1x1x80x80

s_t1 - згрупокані зображення з розміром 1x4x80x80

3 - Convolution Neural Network (Згорткова нейронна мережа)

Тепер можемо передати оброблене зображення до мережі, що є згортковою нейронною мережею:

```
def buildmodel():
    print("Now we build the model")
    model = Sequential()
    model.add(Convolution2D(32, 8, 8, subsample=(4,4),init=lambda shape, name:normal(shape,scale=0.01,name=name),border_mode='same',input_shape=(img_channels,img_rows,img_cols)))
    model.add(Activation('relu'))
    model.add(Convolution2D(64, 4, 4, subsample=(2,2),init=lambda shape, name: normal(shape, scale=0.01, name=name), border_mode='same'))
    model.add(Activation('relu'))
    model.add(Convolution2D(64, 3, 3, subsample=(1,1),init=lambda shape, name: normal(shape, scale=0.01, name=name), border_mode='same'))
    model.add(Activation('relu'))
    model.add(Flatten())
```

```
        model.add(Dense(512, init=lambda shape, name: normal(shape, scale=0.01, name=name)))
        model.add(Activation('relu'))
        model.add(Dense(2,init=lambda shape, name: normal(shape, scale=0.01, name=name)))

        adam = Adam(lr=1e-6)
        model.compile(loss='mse',optimizer=adam)
        print("We finish building the model")
        return model
```

Побудова наступна:

Передаються 4*80*80 зображення. Перший шар згортає 32 фільтри 8*8 з кроком 4 та застосовує функцію активації ReLU. Другий шар згортає 64 фільтри 4*4 з кроком 2 та застосовує функцію активації ReLU. Третій шар згортає 65 фільтри 3*3 з кроком 1 та застосовує функцію активації ReLU. Останній шар - повністю з’єднаний та складається з 512 випрямлювачів. Вихідний шар - повніст'ю з'єднаний лінійний шар для будь-якоі допустимої дії.

4 - DQN

```
if t > OBSERVE:
    #sample a minibatch to train on
    minibatch = random.sample(D, BATCH)

    inputs = np.zeros((BATCH, s_t.shape[1], s_t.shape[2], s_t.shape[3]))   #32, 80, 80, 4
    targets = np.zeros((inputs.shape[0], ACTIONS))                         #32, 2

    #Now we do the experience replay
    for i in range(0, len(minibatch)):
```

```
        state_t = minibatch[i][0]
        action_t = minibatch[i][1]   #This is action index
        reward_t = minibatch[i][2]
        state_t1 = minibatch[i][3]
        terminal = minibatch[i][4]
        # if terminated, only equals reward

        inputs[i:i + 1] = state_t    #I saved down s_t

        targets[i] = model.predict(state_t)  # Hitting each buttom probability
        Q_sa = model.predict(state_t1)

        if terminal:
            targets[i, action_t] = reward_t
        else:
            targets[i, action_t] = reward_t + GAMMA * np.max(Q_sa)

        loss += model.train_on_batch(inputs, targets)

    s_t = s_t1
```

5 - Дослідження - Використання

```
if random.random() <= epsilon:
    print("----------Random Action----------")
    action_index = random.randrange(ACTIONS)
    a_t[action_index] = 1
        else:
    q = model.predict(s_t)       #input a stack of 4 images, get the prediction
    max_Q = np.argmax(q)
    action_index = max_Q
```

a_t[max_Q] = 1

Мережу треба тренувати принаймні на 1 000 000 екранах, щоб вона працювала .

Розбір самої програми qlearn.py:

Дана програма використовує такі пакети, як argparse, skimage, sys, random, numpy, collections та keras

GAME = 'trader' # the name of the game being played for log files

CONFIG = 'nothreshold'

ACTIONS = 3 # number of valid actions

GAMMA = 0.99 # decay rate of past observations

OBSERVATION = 3200. # timesteps to observe before training

EXPLORE = 1000000. # frames over which to anneal epsilon

FINAL_EPSILON = 0.0001 # final value of epsilon

INITIAL_EPSILON = 0.1 # starting value of epsilon

REPLAY_MEMORY = 50000 # number of previous transitions to remember

BATCH = 32 # size of minibatch

FRAME_PER_ACTION = 1

img_rows , img_cols = 257, 13

img_channels = 3

Вводиться ряд констант, що використовуються далі.

ACTIONS визначає кількість доступних дій.

GAMMA визначає значимість минулих спостережень.

OBSERVATION визначає кількість кроків у фазі обстеження. У цьому режимі виконуються випадкові дії та, фактично мережі вказується, що вона може робити та результати цього.

EXPLORE визначає кількість кроків у фазі дослідження. Тут мережа починає обробляти данні з ігрового екрану та вчитися

FINAL_EPSILON визначає ймовірність випадкової дії в кінці навчання.

INITIAL_EPSILON визначає ймовірність випадковоїдії на початку навчання.

REPLAY_MEMORY визначає, скільки минулих дій зберігається в пам'яті.

BATCH визначає розмір вибірки

FRAME_PER_ACTION визначає кількістть дій на один ігровий екран

img_rows та img_cols визначають розмір зображення, що приймається від гри.

img_channels визначає, по скільки зображень групуємо.

Наступний код будує нейронну мережу:

```
def buildmodel():
    print("Now we build the model")
    model = Sequential() # A linear stack of layers
    model.add(Convolution2D(32, 8, 8, subsample=(4,4),init=lambda shape, name:normal(shape,scale=0.01,name=name),border_mode='same',input_shape=(img_channels,img_rows,img_cols)))
    model.add(Activation('relu'))
    model.add(Convolution2D(64, 4, 4, subsample=(2,2),init=lambda shape, name: normal(shape, scale=0.01, name=name), border_mode='same'))
    model.add(Activation('relu'))
    model.add(Convolution2D(64, 3, 3, subsample=(1,1),init=lambda shape, name: normal(shape, scale=0.01, name=name), border_mode='same'))
    model.add(Activation('relu'))
    model.add(Flatten())
    model.add(Dense(512, init=lambda shape, name: normal(shape, scale=0.01, name=name)))
    model.add(Activation('relu'))
```

```
    model.add(Dense(3,init=lambda shape, name: normal(shape, scale=0.01, name=name)))
    adam = Adam(lr=1e-6)
    model.compile(loss='mse',optimizer=adam)
    print("We finish building the model")
    return model
```

Її опис надано раніше.

```
def trainNetwork(model,args):
    game_state = game.GameState()
    D = deque()
    do_nothing = np.zeros(ACTIONS)
    do_nothing[1] = 1 #Because in our game this action is doing nothing
    x_t, r_0, terminal, ac= game_state.frame_step(do_nothing)

    #x_t = skimage.color.rgb2gray(x_t)
    #x_t = skimage.transform.resize(x_t,(80,80)) Not transforming for now
    #x_t = skimage.exposure.rescale_intensity(x_t,out_range=(0,255))

    x_g = x_t[:,:,1]
    x_r = x_t[:,:,0]
    x_b = x_t[:,:,2]
    s_t = np.stack((x_r, x_g, x_b), axis=0)
    s_t = s_t.reshape(1, s_t.shape[0], s_t.shape[2], s_t.shape[1])
```

Тепер, визначимо функцію trainNetwork, що приймає на вхід модель та деякі аргументи, що визначимо пізніше.

Спочатку робимо game_state об’єктом класу GameState.

Зберігаємо у D попередні спостереження з пам'яті.

Далі, передаємо грі дію “нічого не робити” та отримуємо її вивід – екран, винагороду, прапорець про закінчення гри та поточний рахунок.

x_g, x_r та x_b отримуємо з ігрового екрану x_t як відповідне значення зеленого, червоного та синього на кожному пікселі зі спектру RGB.

s_t зберігає зв’язані x_g, x_r та x_b.

```
if args['mode'] == 'Run':
    OBSERVE = 999999999    #We keep observe, never train
    epsilon = FINAL_EPSILON
    print ("Now we load weight")
    model.load_weights("model.h5")
    adam = Adam(lr=1e-6)
    model.compile(loss='mse',optimizer=adam)
    print ("Weight load successfully")
else:                       #We go to training mode
    OBSERVE = OBSERVATION
    epsilon = INITIAL_EPSILON
```

Цей код визначає, дії при тренуванні мережі чи її тестуванні. У разі, коли мережа тестується, OBSERVE надається велике значення. Таким чином, мережа не вчиться. Epsilon одразу виставляється кінцевим. Натренована мережа завантажується з файлу model.h5, що лежить в тій самій директорії.

У випадку тренування, все лишається без змін.

```
t = 0
while (True):
    loss = 0
    Q_sa = 0
    action_index = 0
    r_t = 0
    a_t = np.zeros([ACTIONS])
    #choose an action epsilon greedy
    if t % FRAME_PER_ACTION == 0:
```

```
            if random.random() <= epsilon:
                print("----------Random Action----------")
                action_index = random.randrange(ACTIONS)
                a_t[action_index] = 1
            else:
                q = model.predict(s_t)       #input a stack of 3 images, get the prediction
                max_Q = np.argmax(q)
                action_index = max_Q
                a_t[max_Q] = 1
        #We reduced the epsilon gradually
        if epsilon > FINAL_EPSILON and t > OBSERVE:
            epsilon -= (INITIAL_EPSILON - FINAL_EPSILON) / EXPLORE
        #run the selected action and observed next state and reward
        x_t1, r_t, terminal ,ac = game_state.frame_step(a_t)
        x_g1 = x_t1[:,:,1]
        x_r1 = x_t1[:,:,0]
        x_b1 = x_t1[:,:,2]
        s_t1 = np.stack((x_r1, x_g1, x_b1), axis=0)
    s_t1 = s_t1.reshape(1, s_t1.shape[0], s_t1.shape[2], s_t1.shape[1])
```

Далі, t визначає, який, за загальним рахунком, екран розглядається. Потім створюється нескінченний цикл.

Якщо настав час обирати дію, то з ймовірністю epsilon вона буде випадковою, інакше дія визначатиметься по передбаченню мережі. На кожному наступному кроці, epsilon зменшується, аж пока не досягне значення FINAL_EPSILON.

Наступними кроками, дія передається грі, що видає екран, винагороду, прапорець та рахунок. Екран поділяється на три по кольорах. Екрани склеюються та перегруповуються для подальшого використання.

```
D.append((s_t, action_index, r_t, s_t1, terminal))
```

```
            if len(D) > REPLAY_MEMORY:
                D.popleft()
            #only train if done observing
            if t > OBSERVE:
                #sample a minibatch to train on
                minibatch = random.sample(D, BATCH)

                inputs = np.zeros((BATCH, s_t.shape[1], s_t.shape[2], s_t.shape[3]))
#32, 80, 80, 4
                targets = np.zeros((inputs.shape[0],ACTIONS ))                         #32, 2
                 #Now we do the experience replay
                for i in range(0, len(minibatch)):
                    state_t = minibatch[i][0]   #big array gagin
                    action_t = minibatch[i][1]   #This is action index 0 or 1 or 2
                    reward_t = minibatch[i][2] #0 or 1 or -1
                    state_t1 = minibatch[i][3] #big array
                    terminal = minibatch[i][4] #Fale/true
                    # if terminated, only equals reward
                    inputs[i:i + 1] = state_t    #I saved down s_t
                    targets[i] = model.predict(state_t)  # Hitting each buttom probability
                    Q_sa = model.predict(state_t1)

                    if terminal:
                        targets[i, action_t] = reward_t
                    else:
                        targets[i, action_t] = reward_t + GAMMA * np.max(Q_sa)
                # targets2 = normalize(targets)
                loss += model.train_on_batch(inputs, targets)

            s_t = s_t1
```

```
        t = t + 1
```

Тепер, пам’ять минулих дій та станів D поповнюється.

Якщо закінчили стадію вивчення, то будується Q-функція

```
        if t % 100 == 0:
            print("Now we save model")
            model.save_weights("model.h5", overwrite=True)
            with open("model.json", "w") as outfile:
                json.dump(model.to_json(), outfile)
```

Кожні 100 кроків зберігаємо отриману модель у файл model.h5

```
        state = ""
        if t <= OBSERVE:
            state = "observe"
        elif t > OBSERVE and t <= OBSERVE + EXPLORE:
            state = "explore"
        else:
            state = "train"
        print("TIMESTEP", t, "/ STATE", state, \
            "/ EPSILON", epsilon, "/ ACTION", action_index, "/ REWARD", r_t, \
            "/ Q_MAX " , np.max(Q_sa), "/ Loss ", loss, "/ Cash: ", ac)
    print("Episode finished!")
    print("************************")
```

Та виводимо інформацію про поточний стан керування.

```
def playGame(args):
    model = buildmodel()
    trainNetwork(model,args)
```

Визначаємо функцію playGame, що спочатку будує модель, а потім вчить нейронну мережу.

```
def main():
    parser = argparse.ArgumentParser(description='Description of your program')
    parser.add_argument('-m','--mode', help='Train / Run', required=True)
    args = vars(parser.parse_args())
    playGame(args)
```

Цей код, власне, запускає програму. Спочатку він зчитує параметри, що передаються при виклику програми та викликає за нифи функцію playGame [4].

3. РЕЗУЛЬТАТИ

1) Результати мережі, що вчилася при спреді 0.000001. На 50000 зображеннях

Sell: Gain: 8.040;

Buy: Gain: 8.532;

Sell: Gain: -1.260;

Buy: Gain: 5.290;

Sell: Gain: 3.467;

Buy: Gain: -4.082;

Sell: Gain: 1.351;

Sell: Gain: -10.752;

Sell: Gain: -0.066;

Buy: Gain: 3.422;

Sell: Gain: 6.912;

Buy: Gain: 8.532;

Sell: Gain: -1.260;

Buy: Gain: 5.290;

Sell: Gain: 3.467;

Buy: Gain: -4.082;

Sell: Gain: 1.351;

Buy: Gain: 10.206;

Sell: Gain: -11.824;

Sell: Gain: -8.890;

Buy: Gain: 1.765;

Sell: Gain: -10.823;

Sell: Gain: 2.530;

Buy: Gain: 4.635;

Sell: Gain: 4.956;

Buy: Gain: 1.952;

Sell: Gain: -0.248;

Buy: Gain: 2.667;

Вдалі ордери: 18. Невдалі: 9 Дельта добутку: 32.159

2)Та ж сама мережа, але вчилася при спреді 0.00001. На 50000 зображеннях

Buy: Gain: 9.254;

Sell: Gain: -1.350;

Buy: Gain: 5.200;

Sell: Gain: 3.377;

Buy: Gain: -4.172;

Sell: Gain: 1.261;

Buy: Gain: -7.626;

Sell: Gain: -5.969;

Buy: Gain: 4.545;

Sell: Gain: 4.866;

Buy: Gain: 1.862;

Sell: Gain: -0.338;

Buy: Gain: 2.577;

Sell: Gain: 10.949;

Buy: Gain: -1.995;

Sell: Gain: 6.822;

Buy: Gain: 8.442;

Sell: Gain: -8.104;

Buy: Gain: 2.643;

Sell: Gain: 3.377;

Buy: Gain: -4.172;

Sell: Gain: 1.261;

Sell: Gain: -8.304;

Buy: Gain: 1.675;

Sell: Gain: -10.913;

Sell: Gain: -9.199;

Buy: Gain: 1.675;

Sell: Gain: -10.913;

Вдалі ордери: 17. Невдалі: 12 Дельта добутку: -3.224

3)Та ж сама мережа, але вчилася при спреді 0.0001. На 50000 зображеннях

Sell: Gain: -12.714;

Sell: Gain: -12.630;

Buy: Gain: 5.052;

Sell: Gain: 3.966;

Buy: Gain: 0.962;

Sell: Gain: -1.238;

Buy: Gain: 1.677;

Sell: Gain: 10.049;

Buy: Gain: -2.895;

Sell: Gain: 5.922;

Buy: Gain: 7.542;

Sell: Gain: -2.250;

Buy: Gain: 4.300;

Sell: Gain: 2.477;

Buy: Gain: -5.072;

Sell: Gain: 0.361;

Buy: Gain: 14.576;

Buy: Gain: 4.137;

Buy: Gain: 7.669;

Sell: Gain: -12.457;

Вдалі ордери: 13. Невдалі: 7 Дельта добутку: 19.434

4) Схема, що вчилася на 0.0001, на спреді 0.0001. 50000 зображень

Buy: Gain: 10.584;

Buy: Gain: -3.697;

Buy: Gain: 8.127;

Вдалі ордери: 2. Невдалі: 1 Дельта добутку: 15.014

Також, ефективність нейронної мережі була виміряна на нових даних, відмінних від тих, на яких вона вчилася.

1)Результати нейронної мережі, що тренувалася на спреді 0.0001:

Buy: Gain: 15.083;

Buy: Gain: 1.276;

Вдалі ордери: 2. Невдалі: 0 Дельта добутку: 16.359

Або

Buy: Gain: 6.676;

Buy: Gain: -5.171;

Вдалі ордери: 1. Невдалі: 1 Дельта добутку: 1.575

Або

Buy: Gain: 6.088;

Вдалі ордери: 1. Невдалі: 0 Дельта добутку: 6.088

2)Результати мережі, що тренувалася на спреді 0.000001:

Sell: Gain: 1.459;

Buy: Gain: 2.633;

Sell: Gain: 3.390;

Buy: Gain: 5.709;

Sell: Gain: -6.464;

Buy: Gain: -1.587;

Sell: Gain: -1.097;

Buy: Gain: 0.138;

Sell: Gain: 6.178;

Buy: Gain: -1.017;

Sell: Gain: -6.375;

Buy: Gain: -0.636;

Sell: Gain: 0.150;

Buy: Gain: -5.354;

Sell: Gain: 2.451;

Buy: Gain: -14.244;

Sell: Gain: -2.527;

Вдалі ордери: 8. Невдалі: 9 Дельта добутку: -17.193

або

Sell: Gain: 1.459;

Buy: Gain: 2.095;

Sell: Gain: 2.891;

Buy: Gain: 5.709;

Sell: Gain: -6.464;

Buy: Gain: -1.587;

Sell: Gain: -1.097;

Buy: Gain: 0.138;

Sell: Gain: 6.178;

Buy: Gain: -1.017;

Sell: Gain: -6.375;

Buy: Gain: -0.636;

Sell: Gain: 0.150;

Buy: Gain: -5.354;

Sell: Gain: 2.451;

Buy: Gain: -14.244;

Sell: Gain: -2.527;

Вдалі ордери: 8. Невдалі: 9 Дельта добутку: -18.23

ВИСНОВКИ

Протягом виконання даної курсової роботи було натреновано чотири нейронних мережі за різними параметрами.

З виконаних дослідів можемо встановити, що для конкретної архітектури нейронної мережі та для данної реалізації емулятора, зміна нагород за вдалу або невдалу торгівлю не змінила результату навчання. Також не змінило результату навчання значення мінімального рахунку – значення, при досягненні якого гра закінчувалася.

Можемо бачити, що нейронна мережа, що вчилася на меншому спреді (0.000001) робить більше торгових угод та отримує більше прибутку, якщо тестування проводиться на тій вибірці, на якій вона вчилася. Проте, якщо використовувати нову вибірку, то мережа не робить прибутку.

З іншої сторони, мережа, що вчилася на більшому спреді (0.0001, цей спред вважається нормальним та часто використовується в реальній торгівлі) робить менше торгових угод та отримує дещо менше прибутку. Але, вона також має додатньє сальдо при торгівлі на новій вибірці.

В середньому, мережа, що вчилася торгувати на спреді 0.0001 здатна вигравати приблизно 6 у.о. за 36000 секунд, що становить 10 годин. При умові, що на рахунку 1000 у.о. та ордер відкривається з розміром у 100 у.о. При збільшенні об’єму ордеру прибуотк може бути значно більший.

Дана тема, безумовно, потребує більше досліджень. Є сенс розглянути нейронні мережі, що побудовані не на архітектурі, запропонованій DeepMind а також на емуляторі біржі більш складного типу, з можливостями відкривати ордери різного об'єму тощо.

В результаті курсової роботи, були досліджені архітектури нейронних мереж, принцип роботи торгової біржі, був побудоавний її емулятор на мові програмування Python, була створена нейронна мережа, що здатна торгувати в режимі реального часу, реальних умов торгівлі та приносити прибуток.

СПИСОК ВИКОРИСТАННОЇ ЛІТЕРАТУРИ

1. Dive into Python [Електронний ресурс] //.-Режим доступу: http://www.diveintopython.net/toc/index.html

2. Mastering the Game of Go with Deep Neural Networks and Tree Search [Електронний ресурс] //.-Режим доступу: http://airesearch.com/wp-content/uploads/2016/01/deepmind-mastering-go.pdf

3. Human-level control through deep reinforcement learning [Електронний ресурс] //.-Режим доступу: https://storage.googleapis.com/deepmind-data/assets/papers/DeepMindNature14236Paper.pdf

4. Using Keras and Deep Q-Network to Play FlappyBird [Електронний ресурс] //.-Режим доступу: https://yanpanlau.github.io/2016/07/10/FlappyBird-Keras.html

5. Фондовый рынок: Как устроены биржи и зачем они нужны? [Електронний ресурс] //.-Режим доступ: https://habrahabr.ru/company/itinvest/blog/210570/

6. Understanding the FFT Algorithm [Електронний ресурс] //.-Режим доступу: https://jakevdp.github.io/blog/2013/08/28/understanding-the-fft/

7. PyYAML Documentation [Електронний ресурс] //.-Режим доступу: http://pyyaml.org/wiki/PyYAMLDocumentation

8. Что такое сверточная нейронная сеть [Електронний ресурс] //.-Режим доступу: https://habrahabr.ru/post/309508/

ДОДАТОК

Файл CSVReader.java

```
import java.io.*;
import java.text.ParsePosition;
import java.text.SimpleDateFormat;
import java.util.Date;

public class CSVReader {
   static double myParcer(String input){
      double result = 0;
      String s;
     // char[] bytes = input.toCharArray();

     // s = "" + bytes[1] + ".";
      //int j = 1;
      //for(int i = 0; j <= 5; i+=2) {
       //   s += bytes[i+5];
        //  j++;
      //}
      return Double.parseDouble(input);
   }
   static long myDateParcer(String input){ //Parcing date in our csv file, return the number of miliseconds from the first second in
     /* char[] bytes = input.toCharArray(); //our data range
      String normal = "";
      for(int i = 1; i < bytes.length; i+=2) {
         normal += bytes[i];
      }*/
      SimpleDateFormat df = new SimpleDateFormat("YYYY.MM.dd HH:mm:ss.S");
      Date ex = df.parse("2016.12.19 00:00:00.115", new ParsePosition(0));
      Date d = df.parse(input, new ParsePosition(0));
      return d.getTime() - ex.getTime();
   }

   public static void main(String[] args) {

      String csvFile = "E:\\EURUSD_tick_ghf.csv";
```

```
BufferedReader br = null;
String line = "";
String cvsSplitBy = ",";


try {
    BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(new
FileOutputStream("E:\\vector.txt")));
    BufferedWriter writerdate = new BufferedWriter(new OutputStreamWriter(new
FileOutputStream("E:\\date.txt")));

    br = new BufferedReader(new FileReader(csvFile));
    int j = 0;
    while ((line = br.readLine()) != null) {
        double a, b;
        long ms;
        j++;
        /*if (j % 2 == 0 || j == 1)
            continue;*/
        // use comma as separator
        String[] tick = line.split(cvsSplitBy);
        a = myParcer(tick[1]);
        b = myParcer(tick[2]);
        ms = myDateParcer(tick[0]);
        writer.write((a+b)/2+"");
        System.out.println((a+b)/2);
        writer.newLine();
        writerdate.write(ms+"");
        System.out.println(ms);
        writerdate.newLine();
        writer.flush();
        writerdate.flush();

    }

} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
```

```
        } finally {
            if (br != null) {
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }

    }

}
```

Файл linear.java

```
import java.io.*;
public class Linear {
    BufferedReader values;
    BufferedReader date;
    BufferedWriter output;
    public Linear()throws FileNotFoundException{
        values = new BufferedReader(new FileReader("E:\\vector.txt"));
        date = new BufferedReader(new FileReader("E:\\date.txt"));
        output = new BufferedWriter(new OutputStreamWriter(new FileOutputStream("E:\\newvector.txt")));
    }
    public void getSeconds() throws IOException{
        double last = Double.parseDouble(values.readLine()), curr;
        long lastTime = Long.parseLong(date.readLine()), currTime;
        output.write(last+"");//first second value is copied first tick value
        output.newLine();
        for(int i = 1; i < 162633; i++){
            curr = Double.parseDouble(values.readLine());
            currTime = Long.parseLong(date.readLine());
            if(currTime - lastTime > 1000){  //if one second or more missed
                for (long temp = (lastTime / 1000 + 1) * 1000; temp < currTime - 1000; temp += 1000) {
                    double toWrite = last + (curr - last) * (temp - lastTime) / (currTime - lastTime);
                    output.write(toWrite + "");
                    output.newLine();
```

```
                output.flush();
            }
        }
        if(currTime/1000!=lastTime/1000){
            double toWrite = last + (curr - last) * (currTime / 1000 * 1000 - lastTime) / (currTime - lastTime);
            output.write(toWrite + "");
            output.newLine();
            output.flush();
        }
        last = curr;
        lastTime = currTime;
      }
   }

   public static void main(String[] args) throws Exception {
      Linear lin  = new Linear();
      lin.getSeconds();
   }
}
```

Файл kyrs.py

```
from numpy.fft import fft
from PIL import Image
import matplotlib.pyplot as plt
from numpy.fft import fft
from PIL import Image
import matplotlib.pyplot as plt
with open('E:\\newvector.txt') as f:
   lines = f.readlines()
x = [float(elem) for elem in lines]

i = 0
while i + 1024 < len(x):
   plt.axis('off')
   Pxx, freqs, bins, im = plt.specgram(x[i:1024+i], mode = 'magnitude', NFFT = 256, noverlap = 192)
   plt.axis('off')
   fig = plt.gcf()
   fig.set_frameon(True)
```

```
    fig.set_size_inches(15,320)
    fig.savefig('E:\\newspecgrams\\img'+str(1024+i)+'.png', dpi = 1)
    im = Image.open('E:\\newspecgrams\\img'+str(1024+i)+'.png')
    w, h = im.size
    im = im.crop((2, 32, w, h - 32)).save('E:\\newspecgrams\\img'+str(1024+i)+'.png')
    plt.close()
    i+=1
```

Файл specgram_demo.py

```
import matplotlib.pyplot as plt
import numpy as np

dt = 0.0005
t = np.arange(0.0, 20.0, dt)
s1 = np.sin(2*np.pi*100*t)
s2 = 2*np.sin(2*np.pi*400*t)

# create a transient "chirp"
mask = np.where(np.logical_and(t > 10, t < 12), 1.0, 0.0)
s2 = s2 * mask

# add some noise into the mix
nse = 0.01*np.random.random(size=len(t))

x = s1 + s2 + nse  # the signal
NFFT = 1024       # the length of the windowing segments
Fs = int(1.0/dt)  # the sampling frequency

# Pxx is the segments x freqs array of instantaneous power, freqs is
# the frequency vector, bins are the centers of the time bins in which
# the power is computed, and im is the matplotlib.image.AxesImage
# instance

ax1 = plt.subplot(211)
plt.plot(t, x)
plt.subplot(212, sharex=ax1)
Pxx, freqs, bins, im = plt.specgram(x, NFFT=NFFT, Fs=Fs, noverlap=900,
                                    cmap=plt.cm.gist_heat)
```

```
plt.show()
```

Файл trader_game.py

```
import numpy as np
import sys
import random
import pygame
#import flappy_bird_utils
import pygame.surfarray as surfarray
import os, os.path # To count images in directory Too complicatd?
from pygame.locals import *


#Trading on second graph
FPS =60 # Or, may be faster?
SCREENWIDTH  = 13
SCREENHEIGHT = 257
BLUE =  (  0,   0, 255) # for line indicators drawing
GREEN = (  0, 255,   0)
RED =   (255,   0,   0)

pygame.init()
FPSCLOCK = pygame.time.Clock()
SCREEN = pygame.display.set_mode((SCREENWIDTH, SCREENHEIGHT))
pygame.display.set_caption('Trader')


#This is a lite version of a program. Only one open order is allowed at a time, fixed size.
class GameState:
    def __init__(self):
    self.money = 1000 # actual money
    #self.money_i = 0 # how much order is worth (for kicking a trader)
    self.multiplyer = 1000 # 1:100 (Not used in lite version)
    self.orderprice = 0.1 #fixed price for order
    self.order = 0 # -1 :currently sell order 0 : currently no orders 1: currently buy order
    self.order_price = 0 #price at which order was opened
```

```
self.spread = 0.0001 #obviously, spread value
   #self.score =  0 #Not shure if needed yet
self.kick_price = 900 # At this balance your game ends
#path, dirs, files = os.walk("/home/illya/Trader/src/images/specgrams").next()
#self.max_frames = len(files)
self.max_frames = 20000 #Just for now
self.frame = random.randint(1024, self.max_frames-10000)
with open("/home/illya/Trader/src/newvector.txt") as f:
        self.prices = map(float, f)

def frame_step(self, input_actions):
   pygame.event.pump()

   reward = 0
   terminal = False

   if sum(input_actions) != 1:
      raise ValueError('Multiple input actions!')

   # input_actions[0] == 1: sell
   # input_actions[1] == 1: do nothing
# input_actions[2] == 1: buy
   if input_actions[0] == 1:
      if self.order == 0:
   self.money -= self.orderprice
   self.order = -1
   self.order_price = self.prices[self.frame]
   if self.order == 1:
   buf = self.order_price * ( self.prices[self.frame] - self.orderprice - self.spread) # Income/ Loss
   self.money += buf
   self.order = 0
   self.order_price = 0
   if buf > 0:
      reward = 0.1 # Got money
   if buf < 0:
      reward = -0.1# Lost money
   #No other cases - Can't have two sell orders

   if input_actions[1] == 1:
```

```
        if self.order == 0:
    self.money -= self.orderprice
    self.order = 1
    self.order_price = self.prices[self.frame]
    if self.order == -1:
    buf = self.order_price * ( self.orderprice - self.prices[self.frame] - self.spread) # Income/ Loss
    self.money += buf
    # self.money_i not here
    self.order = 0
    self.order_price = 0
    if buf > 0:
        reward = 0.1 # Got money
    if buf < 0:
        reward = -0.1# Lost money
    #No other cases - Can't have two sell orders


    # check if game ended  - out of money ! or pictures ended
buf = 0 # In case no order is held
if self.order == 1:
    buf = self.order_price * ( self.prices[self.frame] - self.orderprice - self.spread)
if self.order == -1:
    buf = self.order_price * ( self.orderprice - self.prices[self.frame] - self.spread)

#Need an exception when have not enough values!!!(self.prices)
#Leaving game if no money or no more frames
    isCrash= ((self.money - buf) < self.kick_price)

isOutOfFrames = (self.frame + 2 > self.max_frames)

    if isCrash:
        terminal = True
        self.__init__()
    reward = -1
    #If we've ran out of frames, it's not networks fault

if isOutOfFrames:
    self.frame = random.randint(1024, self.max_frames-10000)
```

```
    # draw images
  IMAGE_PATH = '/home/illya/Trader/src/images/newspecgrams/img' + str(self.frame) + '.png'
  current_screen = pygame.image.load(IMAGE_PATH).convert()
    SCREEN.blit(current_screen, (0,0)) # draws one image over another

    showState(self)

    image_data = pygame.surfarray.array3d(pygame.display.get_surface())

  #Updating
    pygame.display.update() # Do we need it?
    #print ("FPS" , FPSCLOCK.get_fps())
  FPSCLOCK.tick(FPS)
  self.frame += 1
  ac = self.money
    return image_data, reward, terminal, ac # did this to know, what's the progress



#Need to be calibrated - so doesn't overlay needed information
#Now this function also tells player the state he is ight now by colour line in low pixel row
#And doesn't tell money anymore - at this size it won't be seen either way
def showState(self):

  if self.order == -1: #green indicator
  pygame.draw.line(SCREEN, GREEN, [0, 256], [12, 256], 1) #Numeration from 0, so...
  if self.order == 0: # red indicator
  pygame.draw.line(SCREEN, RED, [0, 256], [12, 256], 1)
  if self.order == 1: # blue indicator
  pygame.draw.line(SCREEN, BLUE, [0, 256], [12, 256], 1)
```

Файл qlearn.py

```
#!/usr/bin/env python
from __future__ import print_function
```

```
import argparse
import skimage as skimage
from skimage import transform, color, exposure
from skimage.transform import rotate
from skimage.viewer import ImageViewer
import sys
sys.path.append("game/")
import trader_game as game
import random
import numpy as np
from collections import deque

import json
from keras import initializations
from keras.initializations import normal, identity
from keras.models import model_from_json
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation, Flatten
from keras.layers.convolutional import Convolution2D, MaxPooling2D
from keras.optimizers import SGD , Adam

#Initializing constants

GAME = 'trader' # the name of the game being played for log files
CONFIG = 'nothreshold'
ACTIONS = 3 # number of valid actions
GAMMA = 0.99 # decay rate of past observations
OBSERVATION = 3200. # timesteps to observe before training
EXPLORE = 1000000. # frames over which to anneal epsilon
FINAL_EPSILON = 0.0001 # final value of epsilon
INITIAL_EPSILON = 0.1 # starting value of epsilon
REPLAY_MEMORY = 50000 # number of previous transitions to remember
BATCH = 32 # size of minibatch
FRAME_PER_ACTION = 1

img_rows , img_cols = 257, 13
#Convert image into Red Green Blue
img_channels = 3 #We stack three images. Each representig a colour
```

```
#Building out neural network. It is the example of DeepMind neural network

def buildmodel():
    print("Now we build the model")
    model = Sequential() # A linear stack of layers
    model.add(Convolution2D(32, 8, 8, subsample=(4,4),init=lambda shape, name: normal(shape, scale=0.01,
name=name), border_mode='same',input_shape=(img_channels,img_rows,img_cols)))
    model.add(Activation('relu'))
    model.add(Convolution2D(64, 4, 4, subsample=(2,2),init=lambda shape, name: normal(shape, scale=0.01,
name=name), border_mode='same'))
    model.add(Activation('relu'))
    model.add(Convolution2D(64, 3, 3, subsample=(1,1),init=lambda shape, name: normal(shape, scale=0.01,
name=name), border_mode='same'))
    model.add(Activation('relu'))
    model.add(Flatten())
    model.add(Dense(512, init=lambda shape, name: normal(shape, scale=0.01, name=name)))
    model.add(Activation('relu'))
    model.add(Dense(3,init=lambda shape, name: normal(shape, scale=0.01, name=name)))

    adam = Adam(lr=1e-6)
    model.compile(loss='mse',optimizer=adam)
    print("We finish building the model")
    return model


def trainNetwork(model,args):
    # open up a game state to communicate with emulator
    game_state = game.GameState()

    # store the previous observations in replay memory
    D = deque()

    # get the first state by doing nothing and preprocess the image to 80x80x4
    do_nothing = np.zeros(ACTIONS)
    do_nothing[1] = 1 #Because in our game this action is doing nothing
    x_t, r_0, terminal, ac= game_state.frame_step(do_nothing)

    #x_t = skimage.color.rgb2gray(x_t)
```

```
#x_t = skimage.transform.resize(x_t,(80,80)) Not transforming for now
#x_t = skimage.exposure.rescale_intensity(x_t,out_range=(0,255))

x_g = x_t[:,:,1]
x_r = x_t[:,:,0]
x_b = x_t[:,:,2]

s_t = np.stack((x_r, x_g, x_b), axis=0)

#In Keras, need to reshape
s_t = s_t.reshape(1, s_t.shape[0], s_t.shape[2], s_t.shape[1])

if args['mode'] == 'Run':
    OBSERVE = 999999999    #We keep observe, never train
    epsilon = FINAL_EPSILON
    print ("Now we load weight")
    model.load_weights("model.h5")
    adam = Adam(lr=1e-6)
    model.compile(loss='mse',optimizer=adam)
    print ("Weight load successfully")
else:                       #We go to training mode
    OBSERVE = OBSERVATION
    epsilon = INITIAL_EPSILON

t = 0
while (True):
    loss = 0
    Q_sa = 0
    action_index = 0
    r_t = 0
    a_t = np.zeros([ACTIONS])
    #choose an action epsilon greedy
    if t % FRAME_PER_ACTION == 0:
        if random.random() <= epsilon:
            print("----------Random Action----------")
            action_index = random.randrange(ACTIONS)
            a_t[action_index] = 1
        else:
            q = model.predict(s_t)       #input a stack of 3 images, get the prediction
```

```
            max_Q = np.argmax(q)
            action_index = max_Q
            a_t[max_Q] = 1

    #We reduced the epsilon gradually
    if epsilon > FINAL_EPSILON and t > OBSERVE:
        epsilon -= (INITIAL_EPSILON - FINAL_EPSILON) / EXPLORE

    #run the selected action and observed next state and reward
    x_t1, r_t, terminal ,ac = game_state.frame_step(a_t)

    x_g1 = x_t1[:,:,1]
    x_r1 = x_t1[:,:,0]
    x_b1 = x_t1[:,:,2]

    s_t1 = np.stack((x_r1, x_g1, x_b1), axis=0)

s_t1 = s_t1.reshape(1, s_t1.shape[0], s_t1.shape[2], s_t1.shape[1])

#x_t1 = skimage.color.rgb2gray(x_t1_colored)
    #x_t1 = skimage.transform.resize(x_t1,(80,80))
    #x_t1 = skimage.exposure.rescale_intensity(x_t1, out_range=(0, 255))

    # store the transition in D
    D.append((s_t, action_index, r_t, s_t1, terminal))
    if len(D) > REPLAY_MEMORY:
        D.popleft()

    #only train if done observing
    if t > OBSERVE:
        #sample a minibatch to train on
        minibatch = random.sample(D, BATCH)


        inputs = np.zeros((BATCH, s_t.shape[1], s_t.shape[2], s_t.shape[3]))   #32, 80, 80, 4
        targets = np.zeros((inputs.shape[0],ACTIONS ))                         #32, 2
    #How it works and why it is ACTIONS not just 2?
        #Now we do the experience replay
        for i in range(0, len(minibatch)):
```

```
        state_t = minibatch[i][0]   #big array gagin
        action_t = minibatch[i][1]   #This is action index 0 or 1 or 2
        reward_t = minibatch[i][2] #0 or 1 or -1
        state_t1 = minibatch[i][3] #big array
        terminal = minibatch[i][4] #Fale/true
        # if terminated, only equals reward

        inputs[i:i + 1] = state_t    #I saved down s_t

        targets[i] = model.predict(state_t)  # Hitting each buttom probability
        Q_sa = model.predict(state_t1)

        if terminal:
            targets[i, action_t] = reward_t
        else:
            targets[i, action_t] = reward_t + GAMMA * np.max(Q_sa)

    # targets2 = normalize(targets)
    loss += model.train_on_batch(inputs, targets)

s_t = s_t1
t = t + 1

# save progress every 10000 iterations
if t % 100 == 0:
    print("Now we save model")
    model.save_weights("model.h5", overwrite=True)
    with open("model.json", "w") as outfile:
        json.dump(model.to_json(), outfile)

# print info
state = ""
if t <= OBSERVE:
    state = "observe"
elif t > OBSERVE and t <= OBSERVE + EXPLORE:
    state = "explore"
else:
    state = "train"
```

```
        print("TIMESTEP", t, "/ STATE", state, \
            "/ EPSILON", epsilon, "/ ACTION", action_index, "/ REWARD", r_t, \
            "/ Q_MAX " , np.max(Q_sa), "/ Loss ", loss, "/ Cash: ", ac)

    print("Episode finished!")
    print("************************")

def playGame(args):
    model = buildmodel()
    trainNetwork(model,args)

def main():
    parser = argparse.ArgumentParser(description='Description of your program')
    parser.add_argument('-m','--mode', help='Train / Run', required=True)
    args = vars(parser.parse_args())
    playGame(args)

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