Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«Уральский федеральный университет имени первого президента России Б.Н. Ельцина»

Институт естественных наук и математики
Департамент математики, механики и компьютерных наук
Кафедра алгебры и фундаментальной информатики

ДОПУСТИТЬ К ЗАЩИТЕ В ГЭК
Зав. кафедрой _____

_____ _____
(подпись) (Ф.И.О.)
«___»_____ 2019г.

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

КООПЕРАЦИЯ В ИГРАХ ПРИ ИСПОЛЬЗОВАНИИ ОБУЧЕНИЯ С ПОДКРЕПЛЕНИЕМ

Научный руководитель: д.ф.-м.н., ведущий научный сотрудник лаборатории комбинаторной алгебры Шур А.М. _____
(подпись)

Нормоконтролёр: Корабельникова С.В. _____
(подпись)

Студент группы МЕНМ-270213 Бороздин К.А. _____
(подпись)

Екатеринбург
2019

РЕФЕРАТ

Бороздин К.А., КООПЕРАЦИЯ В ИГРАХ ПРИ ИСПОЛЬЗОВАНИИ ОБУЧЕНИЯ С ПОДКРЕПЛЕНИЕМ, квалификационная работа: стр. 57.

Ключевые слова: ОБУЧЕНИЕ С ПОДКРЕПЛЕНИЕМ, Q-ОБУЧЕНИЕ, НЕЙРОННАЯ СЕТЬ, ТЕОРИЯ ИГР, СОЦИАЛЬНАЯ ДИЛЕММА.

Рассматривается задача построения оптимальной стратегии для марковских социальных дилемм с произвольным числом игроков. Вводятся основные определения из теории игр, а также формулируются критерии оптимальности стратегии. Даётся представление о марковских играх и алгоритме Q-обучения. Приводится полное описание алгоритма amTFT. Обобщаются избранные марковские игры, которые затем используются для тестирования алгоритма. Описывается библиотека, созданная для реализации алгоритма и проведения экспериментов. Описываются поставленные эксперименты и интерпретируются полученные результаты.

ABSTRACT

Kirill Borozdin, GAME COOPERATION WITH REINFORCEMENT LEARNING, qualifying thesis: 57 pages.

Keywords: REINFORCEMENT LEARNING, Q-LEARNING, NEURAL NETWORK, GAME THEORY, SOCIAL DILEMMA.

We consider the problem of constructing an optimal policy for Markov social dilemmas with an arbitrary number of players. First, the essential game theory definitions and the policy optimality conditions are given. After that, we introduce Markov games and the Q-learning algorithm. Then, the amTFT algorithm is described in details. In the next chapter, we generalize some Markov games to use them to test the algorithm. After that, we describe the programming framework designed for experimenting with Markov social dilemmas. Finally, the set up experiments are described and their results are discussed.

Содержание

1 Введение

Обучение с подкреплением — область машинного обучения, изучающая взаимодействие одного или нескольких агентов со средой. Задача агента состоит в том, чтобы выработать оптимальную стратегию поведения, соблюдая баланс между исследованием среды и использованием полученных о ней знаний [12].

В течение последних десятилетий обучение с подкреплением успешно применялось для решения различных игр. Однако значительный прорыв был достигнут благодаря использованию глубоких искусственных нейронных сетей [18]. Так, за последние годы алгоритмы превзошли человека в большинстве видеоигр для консоли Atari [13] и в классической игре го [17].

Обычно исследования в области обучения с подкреплением ограничиваются рассмотрением игр, в которых агенты стремятся максимизировать лишь собственную выгоду. Существует особый класс игр, называемых марковскими социальными дилеммами, в которых получение краткосрочной награды каждым игроком вступает в противоречие с возможностью получения долгосрочной выгоды всеми игроками. Самым известным примером такой игры является повторяющаяся дилемма заключённого [2]. Было показано, что стандартные алгоритмы обучения с подкреплением не находят оптимальную кооперативную стратегию даже в этой простой игре [16].

В последнее время было предложено несколько подходов к решению задач данного класса, среди них алгоритмы amTFT [11] и LOLA [7]. В данной диссертации изучается алгоритм amTFT, а также описывается его обобщение на три и более агента. Тестирование алгоритма производится с помощью самостоятельно написанной библиотеки [3].

Кратко опишем содержание диссертации. Во второй главе описывается дилемма заключённого, даются основные определения из теории игр, а также формулируются критерии оптимальности стратегии в повторяющейся социальной дилемме. В третьей главе описывается игра «Монетки» и включающий её класс марковских игр, а затем даётся представление о Q-обучении. В четвёртой главе описывается обобщённая версия алгоритма amTFT. В пятой главе описываются обобщения избранных марковских игр на произвольное число игроков. В шестой главе обосновывается необходимость написания собственной библиотеки для тестирования алгоритма, описывается её архитектура и приводятся примеры использования. В седьмой главе описываются проведённые эксперименты и интерпретируются полученные результаты.

2 Теоретико-игровая постановка задачи

В данной главе описывается дилемма заключённого, а также даются основные определения из теории игр. После этого формулируются критерии оптимальности стратегии в повторяющейся социальной дилемме.

Рассмотрим игру, которая называется дилемма заключённого. В неё играют два игрока, каждому нужно независимо выбрать одно из двух доступных действий: эгоистичное D (обвинить оппонента) или кооперативное C (взять вину на себя). Если оба игрока выбрали действие D, то они получают по -2 очка (попадают в тюрьму на два года). Если оба игрока выбрали действие C, то они получают по -1 очку. Наконец, если игроки выбрали разные действия, то выбравший действие C игрок получает -3 очка, а выбравший действие D игрок получает 0 очков. Цель игры — набрать максимальное количество очков. Отсутствие коммуникации приводит к тому, что оба игрока выбирают действие D, опасаясь выбора оппонента, и, тем самым, набирают меньше очков, чем могли бы. Чтобы формализовать данное наблюдение, введём необходимые определения.

Определение 1. *Игра в нормальной форме* Γ *— это кортеж* $\langle \mathcal{N}, A, u \rangle$*, где*

- $\mathcal{N}$ *— множество игроков* $\{1, \ldots, n\}$*;*
- $A = A_1 \times \cdots \times A_n$*, где* A_i *— конечное множество действий, доступное* i*-му игроку;*
- $u = (u_1, \ldots, u_n)$*, где* $u_i \colon A \mapsto \mathbb{R}$ *есть функция выплат* i*-го игрока.*

Игру для двух игроков можно представить в виде таблицы. Например, дилемма заключённого выглядит следующим образом:

	2-й игрок выбирает D	2-й игрок выбирает C
1-й игрок выбирает D	(-2, -2)	(0, -3)
1-й игрок выбирает C	(-3, 0)	(-1, -1)

Стратегия игрока — правило, согласно которому игрок выбирает действие. Перед тем, как строго определить это понятие, скажем, что $\Delta(X)$ обозначает множество всех вероятностных распределений над множеством X.

Определение 2. *Множеством стратегий для* i*-го игрока называется* $S_i = \Delta(A_i)$*.*

Теперь дадим несколько определений, которые позволяют понять, какие стратегии выбирают игроки, стремясь максимизировать свой счёт. Здесь и далее оператор $\mathrm{E}[\cdot]$ обозначает математическое ожидание.

Определение 3. *Набор стратегий s_{-i} содержит стратегии всех игроков, кроме i-ого, то есть $s_{-i} = s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n$.*

Определение 4. *Наилучшим ответом i-го игрока на набор стратегий s_{-i} называется стратегия $s_i^* \in S_i$, такая что $\forall s_i \in S_i \colon \mathrm{E}[u_i(s_i^*, s_{-i})] \geq \mathrm{E}[u_i(s_i, s_{-i})]$.*

Определение 5. *Набор стратегий $s = s_1, \ldots, s_n$ называется равновесием Нэша, если для всех i верно, что s_i — наилучший ответ на s_{-i}.*

Джон Нэш доказал, что в любой игре существует хотя бы одно равновесие Нэша [14]. Теперь определим вид равновесия, выгодный с точки зрения всех игроков.

Определение 6. *Набор стратегий $s = s_1, \ldots, s_n$ называется Парето-оптимальным, если не существует такого набора $s' = s'_1, \ldots, s'_n$, что $\forall i \colon \mathrm{E}[u_i(s')] \geq \mathrm{E}[u_i(s)]$ и $\exists i \colon \mathrm{E}[u_i(s')] > \mathrm{E}[u_i(s)]$.*

Можно заметить, что в дилемме заключённого равновесием Нэша является набор чистых стратегий DD, а Парето-оптимальными — наборы чистых стратегий CC, CD и DC. При этом набор стратегий CC строго лучше набора DD с точки зрения обоих игроков. Наличие Парето-оптимального набора стратегий, лучшего, чем все равновесия Нэша, является отличительной чертой социальных дилемм.

До этого момента рассматривались игры, состоящие из одного взаимодействия между игроками. Наибольший же интерес представляют повторяющиеся игры, так как они позволяют моделировать сложное поведение игроков. Например, повторяющаяся дилемма заключённого состоит из m ходов, каждый из которых эквивалентен обычной игре. Теорема, которую обычно называют фольклорной, утверждает, что при стремлении m к бесконечности и выполнении ряда условий кооперативный набор стратегий становится одним из равновесий Нэша [8].

Одна из известных стратегий в повторяющейся дилемме заключённого называется tit-for-tat [2]. На первом ходу игрок, придерживающийся данной стратегии, выполняет действие C. На последующих ходах игрок копирует предыдущее действие оппонента. Эмпирическая оптимальность tit-for-tat следует из выполнения следующих свойств:

- Не позволяет эгоистичным стратегиям себя эксплуатировать;

- Кооперируется со склонными к кооперации стратегиями (например, с собой);
- Позволяет вернуться к кооперации, даже если оппонент совершил ошибочное эгоистичное действие, а затем продолжил кооперативную игру;
- Мотивирует идеального оппонента (то есть оппонента, подстраивающегося под нашу стратегию) на кооперацию.

Таким образом, основная задача состоит в том, чтобы для произвольной повторяющейся социальной дилеммы найти стратегию, обладающую описанными выше свойствами.

3 Постановка задачи с точки зрения обучения с подкреплением

В данной главе описывается игра «Монетки», а также включающий её класс марковских игр. После этого даётся представление о Q-обучении, которое в дальнейшем будет использовано для построения алгоритма amTFT.

Повторяющаяся дилемма заключённого отличается от многих игр тем, что состояние среды не изменяется при совершении действий. Рассмотрим игру, которая называется «Монетки». В неё играют два игрока, действие происходит на квадратном поле $k \times k$. Игра состоит из m ходов, на каждом ходу игроки одновременно выбирают одно из четырёх доступных действий: сдвинуться на одну клетку вправо, вверх, влево или вниз. Каждому игроку соответствует свой цвет; с вероятностью p после каждого хода на случайной свободной клетке (если такие есть) появляется монетка случайного цвета. Если игрок оказывается в одной клетке с монеткой, он получает 1 очко, а монетка исчезает. Однако, если это была монетка не своего цвета, то оппонент также получает -2 очка. Цель игры — набрать максимальное количество очков. Данная игра, несмотря на значительно большую вариативность стратегий, напоминает повторяющуюся дилемму заключённого. Действительно, эгоистичная стратегия сбора всех монеток со стороны обоих игроков приводит к худшему результату, чем если бы каждый игрок действовал кооперативно, собирая только монетки своего цвета. Формализуем описание игры такого вида.

Определение 7. *Марковская игра* Γ *— это кортеж* $\langle S, \mathcal{N}, A, T, r \rangle$*, где*

- S *— конечное множество состояний;*

- $\mathcal{N}$ — *множество игроков* $\{1, \ldots, n\}$;
- $A = A_1 \times \cdots \times A_n$, *где* A_i — *конечное множество действий, доступное* i*-му игроку;*
- $T \colon S \times A \mapsto \Delta(S)$ — *функция переходов, зависящая от текущего состояния и набора действий;*
- $r = (r_1, \ldots, r_n)$, *где* $r_i \colon S \times A \mapsto \mathbb{R}$ *есть функция наград* i*-го игрока.*

В описанной выше игре состоянием будет набор позиций игроков, а также расположение и цвет всех монеток. Также заметим, что для любого конечного количества ходов повторяющиеся игры являются марковскими играми. Теперь рассмотрим важный подкласс марковских игр, традиционно изучаемый в области обучения с подкреплением.

Определение 8. *Марковская игра, в которой участвует один игрок (агент), называется марковским процессом принятия решений.*

Так как в марковских процессах принятия решений исход определяется исключительно действиями единственного агента, становится возможным строго определить оптимальность стратегии.

Определение 9. *Пусть* Γ — *марковский процесс принятия решений и* $\gamma \in (0; 1)$ — *заранее выбранный параметр. Стратегия* $\pi \colon S \mapsto \Delta(A)$ *называется оптимальной, если она максимизирует ожидаемую награду для всех состояний* $s \in S$*:*

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

Существует множество алгоритмов для решения марковских процессов принятия решений, то есть для нахождения оптимальной стратегии. Рассмотрим метод Q-обучения [19]. Для этого введём вспомогательную функцию:

$$Q(s, a) = \sum_{s' \in S} T(s, a, s') \left(r(s, a) + \gamma \max_{a'} Q(s', a') \right)$$

Заметим, что данная функция позволяет легко построить оптимальную стратегию. Здесь и далее оператор $[\cdot]$ равен единице, если условие верно, и нулю в обратном случае:

$$\pi(s, a) = \left[a = \operatorname*{argmax}_{a} Q(s, a) \right]$$

Будем поддерживать $\hat{Q}(s, a)$ в некоторой структуре данных (в случае малого числа состояний и действий можно использовать простую таблицу, иначе — любой аппроксиматор, например, нейронную сеть). Каждый раз, когда агент выполняет переход из состояния s в состояние s', совершив действие a и получив награду r, структура данных обновляется следующим образом (α — коэффициент скорости обучения):

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

Можно показать, что если каждая пара состояние-действие посещается бесконечное число раз, $\hat{Q}$ сходится к своему истинному значению [19].

Осталось определить стратегию агента во время обучения. Простейший способ сделать это называется ε-жадным исследованием (параметр ε убывает в ходе обучения):

$$\pi(s, a) = (1 - \varepsilon) \left[a = \operatorname*{argmax}_a \hat{Q}(s, a) \right] + \frac{\varepsilon}{|A|}$$

Вернёмся к рассмотрению марковских игр и постараемся применить Q-обучение. Для этого каждый агент может рассматривать остальных агентов как часть среды, сводя задачу к марковскому процессу принятия решений. К сожалению, данный подход приводит к непредсказуемым результатам, так как нарушается фундаментальное свойство марковского процесса — стационарность. Действия агентов влияют на функции переходов и наград и при этом зависят друг от друга. Для марковских социальных дилемм это означает, что стратегии агентов могут не сойтись или стать некооперативными. Как было продемонстрировано в статье [16], проблемы возникают даже при использовании Q-обучения для решения повторяющейся дилеммы заключённого. Соответственно, для решения более сложных игр следует искать другие подходы.

4 Алгоритм amTFT

В данной главе с теоретической и практической точек зрения описывается алгоритм amTFT [11]. При этом все определения и утверждения из оригинальной статьи обобщаются на три и более игрока.

Идея алгоритма вдохновлена описанной во второй главе стратегией tit-for-tat. Агент, использующий данную стратегию, стремится к кооперации, однако всегда готов переключиться на эгоистичную стратегию в случае некооперативного поведения

оппонента. При этом оппонент будет вынужден вернуться на путь кооперации, чтобы максимизировать свою ожидаемую награду. В марковских социальных дилеммах кооперативные и эгоистичные стратегии обычно описываются не единичным действием, а их последовательностью. Формализуем, что понимается под этими словами.

Определение 10. *Набор стратегий $\pi^C = \pi_1^C, \ldots, \pi_n^C$ называется кооперативным, если он максимизирует суммарную ожидаемую награду $\sum_{i=1}^{n} V_i^{\pi^C}(s)$ для всех состояний $s \in S$. Набор стратегий $\pi^D = \pi_1^D, \ldots, \pi_n^D$ называется эгоистичным, если каждый агент максимизирует собственную ожидаемую награду $V_i^{\pi^D}(s)$ для всех состояний $s \in S$, считая остальных агентов частью среды.*

Будем считать, что любой кооперативный набор стратегий π^C приводит к одному и тому же распределению наград, иначе агенты должны будут некоторым образом договариваться о конкретном способе кооперации, что выходит за рамки данной работы.

Для поиска описанных выше стратегий можно воспользоваться алгоритмом Q-обучения. Чтобы найти кооперативные стратегии, следует положить награду каждого агента равной сумме наград всех агентов. При этом Q-обучение не будет испытывать проблем со сходимостью, так как агенты максимизируют одну и ту же величину, другими словами, существует воображаемый суперагент, работающий с марковским процессом принятия решений. Для отыскания эгоистичных стратегий так же воспользуемся Q-обучением. В этом случае, чтобы избежать проблем со сходимостью, потребуется тщательный подбор гиперпараметров, который, однако, возможен на практике.

Теперь сформулируем пару определений, описывающих важное свойство марковской социальной дилеммы: если все агенты придерживаются эгоистичной стратегии, то ни одному из них не выгодно переключаться на кооперативную стратегию.

Определение 11. *Составная стратегия π^{XkY} — стратегия, в которой первые k ходов совершаются согласно π^X, а все последующие ходы согласно π^Y.*

Определение 12. *Игра называется π^D-доминируемой, если для любого i, любого k, любого состояния $s \in S$ и любой стратегии π_i^X верно, что*

$$V_i^{\pi_1^{DkC},\ldots,\pi_i^{DkC},\ldots,\pi_n^{DkC}}(s) \geq V_i^{\pi_1^{DkC},\ldots,\pi_i^{XkC},\ldots,\pi_n^{DkC}}(s)$$

Опишем алгоритм amTFT. Предположим, что мы знаем наборы стратегий π^C и π^D

всех агентов, а также истинные Q-функции. В каждый момент времени агент пребывает в кооперативном или эгоистичном режиме, начиная игру в кооперативном режиме. При этом текущий режим определяет, какую стратегию использует агент: π_i^C или π_i^D. Также, в кооперативном режиме агент вычисляет потенциальную выгоду, которую извлекли оппоненты, совершив на текущем ходу действия $a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n$:

$$d = \max_{j \neq i} \left(Q_j^{\pi^C}(s, a_1, \ldots, a_n) - V_j^{\pi^C}(s) \right)$$

Если $d > 0$, то следующие k ходов агент проводит в эгоистичном режиме, где k — минимальное число, такое что

$$\max_{j \neq i} \left(V_j^{\pi^C}(s') - V_j^{\pi^{DkC}}(s') \right) > \alpha d$$

Здесь $\alpha > 1$ — множитель, определяющий, насколько часто агента сможет эксплуатировать эгоистичная стратегия. По прошествии k ходов агент возвращается в кооперативный режим.

Следующая теорема показывает, что алгоритм amTFT мотивирует оппонентов следовать кооперативной стратегии.

Теорема 1. *Пусть* Γ *—* π^D*-доминируемая марковская игра, где* π^D *и* π^C *— наборы стратегий, построенные описанным выше способом. Пусть*

$$d^* = \max_{s \in S, i \in \mathcal{I}, a_i \in A_i} \left(Q_i^{\pi^C}\left(s, \pi_1^C(s), \ldots, a_i, \ldots, \pi_n^C(s)\right) - V_i^{\pi^C}(s) \right)$$

Пусть для любого игрока i *и состояния* $s \in S$ *верно, что*

$$V_i^{\pi^C}(s) - V_i^{\pi^D}(s) > \alpha d^*$$

Пусть все игроки, кроме i*-го, играют согласно алгортиму amTFT, а* i*-й игрок максимизирует свою выгоду. Тогда он будет следовать стратегии* π_i^C*, пока остальные игроки находятся в кооперативном режиме, и стратегии* π_i^D*, пока остальные игроки находятся в эгоистичном режиме.*

Доказательство этого факта аналогично его доказательству для двух игроков и может быть найдено в оригинальной статье [11].

Разумеется, на практике мы не знаем истинные Q-функции. Первый подход к решению данной проблемы состоит в том, чтобы использовать аппроксиматор $\hat{Q}$,

построенный в ходе обучения. Второй подход заключается в использовании так называемых отыгрышей. Отыгрыш — симуляция игры на M ходов вперёд. Если произвести B отыгрышей и усреднить ожидаемую выгоду, то при больших значениях M и B мы получим несмещённую оценку Q-функции. Второй подход оказывается значительно более точным, хотя и более вычислительно затратным. Главное преимущество отыгрышей состоит в том, что они корректно оценивают стратегии, которые сильно отличаются от π^C и π^D, но эквивалентны им в распределении наград.

Чтобы избежать переходов в эгоистичный режим, вызванных погрешностью вычислений, введём параметр T — ограничение на суммарное уклонение оппонентов. Будем накапливать в кооперативном режиме вычисленные значения d, пока их сумма не превысит T, и лишь затем переходить в эгоистичный режим.

Для вычисления k, определённого выше, можно воспользоваться бинарным поиском по ответу. Нужно отметить, что параметр M является естественной верхней границей поиска.

5 Обобщения некоторых марковских игр

В данной главе описываются обобщения двух марковских игр, которые затем будут использованы для тестирования алгоритма amTFT. Среди них обобщение повторяющейся дилеммы заключённого на n игроков, которое ранее не встречалось в литературе.

Чтобы проверить корректность реализации алгоритма, а также качество его работы, необходимо выбрать несколько марковских социальных дилемм для тестирования. В связи с тем, что в данной диссертации рассматриваются игры на произвольное количество игроков, все выбранные среды должны быть параметризованы числом n.

В предыдущих главах рассматривалась повторяющаяся дилемма заключённого. Разумеется, данная игра должна легко решаться алгоритмом amTFT, так как он основан на стратегии tit-for-tat. С другой стороны, повторяющаяся дилемма заключённого позволяет убедиться в корректности алгоритма, но лишь относительно его версии для двух игроков. Опишем возможное обобщение игры на число игроков, большее двух, сохраняющее свойства социальной дилеммы.

Пусть $c_1, \ldots, c_n$ и $d_0, \ldots, d_{n-1}$ — два набора действительных чисел, задающих награды игроков. На каждом ходу игроки по-прежнему должны выбрать одно из двух действий: эгоистичное (D) или кооперативное (C). Пусть ровно k игроков выбрали

действие C. Тогда все выбравшие действие C игроки получают награду c_k, а все выбравшие действие D игроки получают награду d_k. Например, для двух игроков $c = (-3, -1)$ и $d = (-2, 0)$. Сформулируем требования к наборам c_i и d_i, достаточные, чтобы игра оставалась социальной дилеммой:

1. $\forall i \in [1; n-1]$: $d_i > d_{i-1}$, то есть с ростом количества кооперирующихся игроков награды возрастают;
2. $\forall i \in [2; n]$: $c_i > c_{i-1}$, аналогично предыдущему пункту;
3. $\forall i \in [1; n]$: $d_{i-1} > c_i$, то есть каждому отдельно взятому игроку выгоднее совершить действие D;
4. $\forall i \in [1; n]$: $ic_i + (n-i)d_i > (i-1)c_{i-1} + (n-i+1)d_{i-1}$, то есть с ростом количества кооперирующихся игроков суммарная награда возрастает.

Из пункта 3 следует, что равновесие Нэша достигается только тогда, когда все игроки совершают действие D. Из пункта 4 же следует, что такой исход не является Парето-оптимальным, зато исход, в котором все игроки совершают действие C, является. Можно убедиться, что следующая конструкция для $n > 2$ удовлетворяет всем описанным свойствам.

- $c = (-n, -(n-1), \ldots, -2, -1)$;
- $d = (-(n-1), -(n-2), \ldots, -1, 0)$.

Перейдём ко второй игре. В предыдущих главах рассматривалась игра «Монетки». Несмотря на то, что она формулируется для двух игроков, её обобщение не представляет особенных сложностей. Теперь на поле одновременно находятся n игроков, каждому игроку соответствует свой цвет. Если игрок собирает монетку не своего цвета, то штраф получает соответствующий цвету монетки оппонент.

6 Описание библиотеки для тестирования

В данной главе описываются требования к библиотеке, рассматриваются существующие решения и мотивируется необходимость написания собственного. Далее описывается архитектура новой библиотеки и даются примеры её использования.

Чтобы провести тестирование алгоритма amTFT, необходимо было реализовать сам алгоритм, так как авторы оригинальной статьи [11] не опубликовали исходный код.

Также потребовалась реализация Q-обучения; использующихся в нём нейронных сетей; марковских игр, описанных выше; инфраструктуры для взаимодействия агентов и среды.

Проще всего ситуация обстоит с нейронными сетями. Машинное обучение переживает период расцвета, поэтому в открытом доступе находятся сразу несколько библиотек для создания быстрых и конфигурируемых сетей, например Tensorflow [1] и Pytorch [15]. Так как автор данной диссертации имеет больший опыт взаимодействия с библиотекой Pytorch, именно она была использована при написании кода.

Поиск полноценной библиотеки для обучения с подкреплением оказался значительно сложнее. Первые эксперименты были выполнены с помощью библиотеки Tensorforce [10], но затем было принято решение отказаться от неё. Во-первых, данная библиотека предоставляет слишком высокий уровень абстрации, что усложняет отладку и изменение кода. Во-вторых, Tensorforce не поддерживает взаимодействие нескольких агентов со средой, то есть появилась бы необходимость модифицировать саму библиотеку. В-третьих, она содержит ощутимое число ошибок и недоработок. Так, автору данной диссертации пришлось сделать пулл-реквест (отправить исправление автору библиотеки), чтобы корректно провести несложный эксперимент. Другая библиотека, которая рассматривалась и затем была отвергнута, называется Dopamine [4]. В настоящее время она стремительно развивается, но на момент начала проведения экспериментов ещё не была в достаточной степени готова.

Таким образом, было принято решение реализовать собственную библиотеку [3]. Основным преимуществом стала лёгкость модификации кода. Итоговый размер библиотеки составил всего лишь около 700 строк. Используемый язык — Python 3, фактически, ставший стандартом в машинном обучении.

Рассмотрим архитектуру библиотеки. Поведение агента описывается с помощью классов `BaseAgent` и `BasePolicy`. Класс `BaseAgent` содержит два основных метода, которые переопределяются в наследниках:

- `get_policy(state, do_exploration)` — вернуть стратегию (наследника `BasePolicy`) в состоянии `state`, возможно, с учётом исследования среды;
- `observe(prev_state, action, reward, next_state)` — получить информацию о нескольких переходах; i-ый из них совершался из состояния `prev_state[i]` в состояние `next_state[i]` путём совершения набора действий `action[i]`, при этом агент получил награду `reward[i]`.

Класс `BasePolicy` содержит единственный метод, который переопределяется в наследниках:

- `sample(state)` — вернуть действие в состоянии `state`.

Наследники класса `BasePolicy` позволяют удобно работать с типичными стратегиями:

- `OneHotPolicy(action)` — возвращает действие `action` независимо от состояния;
- `ConstantPolicy(distribution)` — возвращает случайное действие согласно распределению `distribution` независимо от состояния;
- `CompoundPolicy(fst_policy, snd_policy, change_after)` — первые `change_after` ходов использует стратегию `fst_policy`, затем — `snd_policy`;
- `LambdaPolicy(func)` — возвращает действие согласно стратегии, полученной путём вызова `func(state)`.

Класс `LambdaPolicy` позволяет получить стратегию, имея агента. Обратное преобразование также возможно: наследник класса `BaseAgent` под названием `PolicyAgent` принимает стратегию `policy` и действует согласно ей, игнорируя вызовы `observe`.

Описанное выше разделение агента и стратегии не является обязательным, но позволяет подчеркнуть тот факт, что стратегия заведомо не является обучаемой. Более того, такая архитектура упрощает сопоставление теоретического описания алгоритма и его практической реализации.

Среда (или игра) описывается с помощью класса `BaseEnvironment`. Он содержит три основных метода, которые переопределяются в наследниках:

- `reset(state=None)` — сбросить состояние среды или установить его равным `state`;
- `get_state()` — получить текущее состояние среды;
- `execute(actions)` — применить действия агентов `actions`, соответствующим образом изменить состояние среды и вернуть полученные агентами награды.

Инфраструктура для взаимодействия агентов и среды реализована в виде класса `Runner`. Его конструктор принимает множество параметров и позволяет настроить инфраструктуру в зависимости от сценария использования. Приведём несколько примеров:

- `Runner(num_runs=100, num_steps=50)` — выполняет 100 запусков по 50 шагов; возвращает средние арифметические наград агентов. Используется для вычисления отыгрышей;

- `Runner(num_runs=1, num_steps=500, visualize=True, visualize_gameplay=True)` — выполняет единственный запуск из 500 ходов; отображает состояние среды в виде анимации (требуется запуск с помощью Jupyter Notebook [9]). Используется для визуальной отладки алгоритма;

- `Runner(num_runs=200, num_steps=10**4, batch_size=8, replay_size=10**3, replay_batch_size=16, coop_reward=True, agent_names=['Alice', 'Bob'], visualize=True, visualize_every=4)` — выполняет 200 запусков по 10000 шагов; переходы передаются агенту раз в 8 ходов; поддерживается история последних 1000 ходов в текущем запуске, и каждые 8 ходов агенту также передаются 16 переходов из истории; награда каждого агента равна сумме наград всех агентов; строятся графики наград на каждом четвёртом запуске; на графиках используются выбранные названия агентов. Используется для обучения агентов.

Для запуска экземпляра класса `Runner` используется метод `run(env, agents, rnd_seeds=None)`, который принимает среду, набор агентов и, возможно, инициализаторы для генератора случайных чисел на каждый запуск.

Осталось заметить, что алгоритмы Q-обучения и amTFT можно реализовать в виде наследников `BaseAgent`. Класс `DQNAgent` принимает параметры обучения, например, α, γ и настройки ε-жадного исследования, а также нейронную сеть, возвращающую значения Q-функции для данного состояния. Нейронная сеть представляет собой наследника класса `Module` из библиотеки Pytorch [15] и реализуется отдельно для каждой среды.

Класс `AMTFTAgent` принимает параметры алгоритма, например, α и T; экземпляр класса `Runner`, использующийся для вычисления отыгрышей; заранее вычисленные наборы стратегий π^C и π^D.

Полный исходный код можно найти в приложении к данной диссертации.

7 Эксперименты

В данной главе детально описываются проведённые эксперименты, затем приводятся результаты этих экспериментов и даётся их интерпретация.

Для тестирования алгоритма мы будем использовать две марковские социальные дилеммы, описанные в предыдущих главах: повторяющуюся дилемму заключённого и игру «Монетки». Каждая игра будет рассматриваться в версиях для двух и трёх игроков. Такой выбор обусловлен тем, что в оригинальной статье [11] алгоритм amTFT был сформулирован для двух игроков, то есть рассмотрение случая трёх игроков необходимо для проверки сделанного в данной диссертации обобщения. С другой стороны, при дальнейшем увеличении количества игроков не происходит качественного изменения алгоритма, но лишь увеличивается время его обучения и тестирования.

Каждая из четырёх групп экспериментов выглядит следующим образом. Сначала производится обучение агентов для получения наборов стратегий π^C и π^D. Каждый из дальнейших экспериментов — это пять запусков игры на фиксированное число ходов. Результатом каждого эксперимента является усреднённый график зависимости суммарных наград игроков от номера хода. Опишем эти эксперименты подробнее:

1. Участвуют $n-1$ кооперативных стратегий и одна эгоистичная, эксперимент проверяет, насколько сильно эгоистичная стратегия может эксплуатировать кооперативные;

2. Участвуют $n-1$ кооперативных стратегий и одна вручную запрограммированная эгоистичная стратегия, цель эксперимента аналогична предыдущей;

3. Участвуют n агентов amTFT, эксперимент проверяет, могут ли агенты amTFT кооперироваться друг с другом;

4. Участвуют $n-1$ агентов amTFT и одна эгоистичная стратегия, эксперимент проверяет, устойчивы ли агенты amTFT к эксплуатации эгоистичной стратегией;

5. Участвуют $n-1$ агентов amTFT и одна вручную запрограммированная эгоистичная стратегия, эксперимент проверяет, устойчивы ли агенты amTFT к эксплуатации эгоистичной стратегией, не совпадающей с π_i^D;

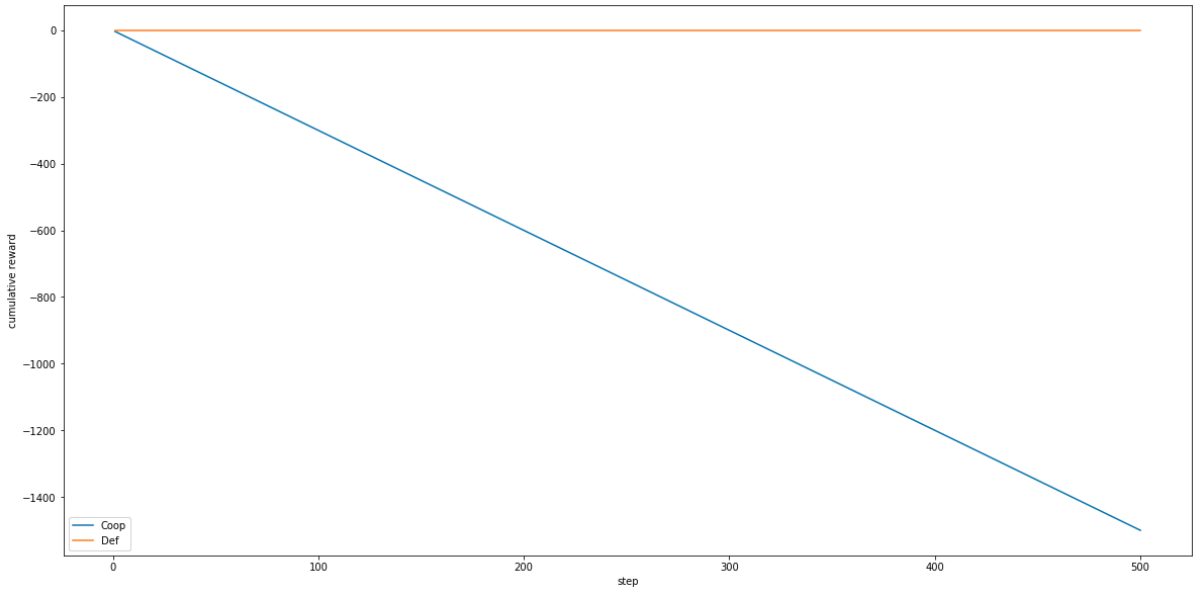
6. Участвуют $n-1$ агентов amTFT и одна вручную запрограммированная кооперативная стратегия, эксперимент проверяет, могут ли агенты amTFT кооперироваться с кооперативной стратегией, не совпадающей с π_i^C.

Теперь мы можем описать особенности каждой из рассматриваемых игр и обсудить полученные результаты.

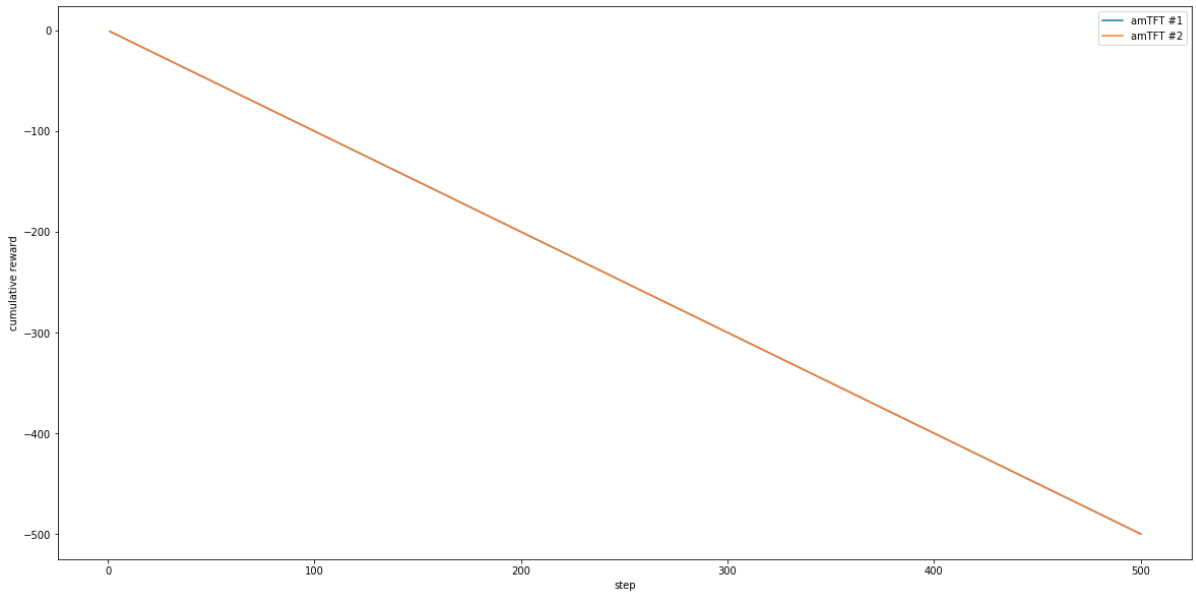
7.1 Повторяющаяся дилемма заключённого

В качестве состояния игры будем рассматривать действия игроков на предыдущем ходу (отдельное состояние соответствует первому ходу игры). В силу малой мощности множества состояний можно выбирать практически любые параметры обучения и конфигурацию нейронной сети. Было принято решение использовать следующую (избыточную для данной задачи) нейронную сеть: эмбеддинг (вложение) из множества состояний в $\mathbb{R}^{16}$, за которым следует скрытый линейный слой из 16 нейронов, за которым следует активационная функция ELU [5], за которой следует выходной линейный слой из двух нейронов. В качестве вручную запрограммированной эгоистичной стратегии будем использовать стратегию, всегда совершающую действие D, так как для повторяющейся дилеммы заключённого невозможно придумать нечто более сложное, но при этом эффективное. В качестве вручную запрограммированной кооперативной стратегии будем использовать стратегию tit-for-tat [2], обобщённую для n игроков. А именно, игрок выбирает действие C, если и только если все оппоненты на предыдущем ходу выбрали действие C (либо это первый ход игры).

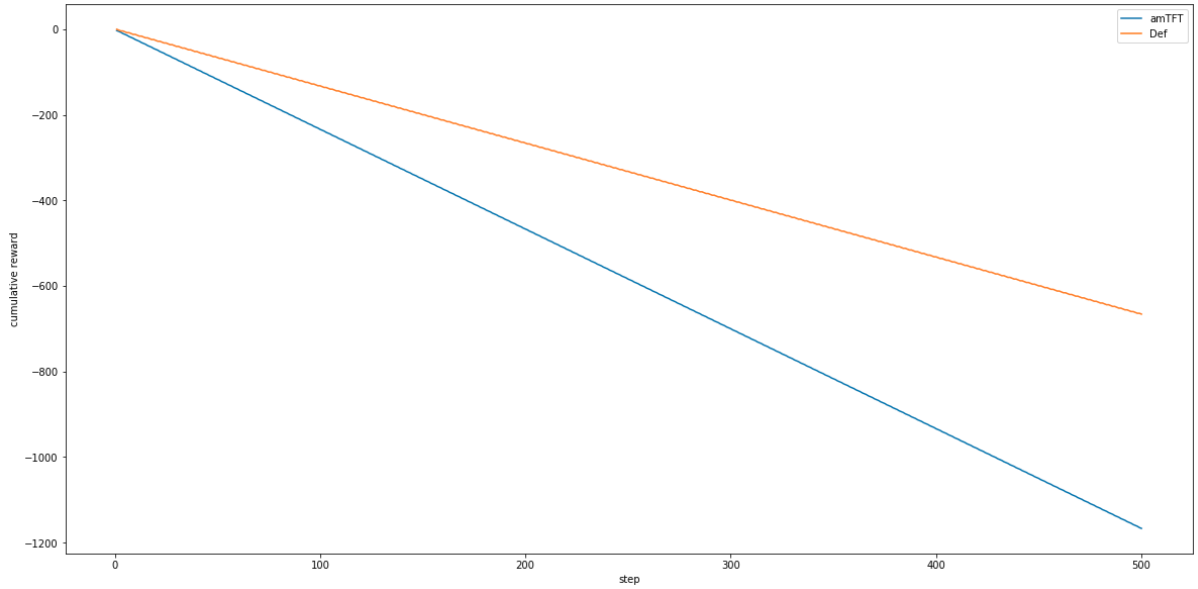
Рассмотрим версию игры для двух игроков. После непродолжительного обучения кооперативный агент начинает всегда совершать действие C, а эгоистичный агент — действие D. То есть обученная эгоистичная стратегия совпала с вручную запрограммированной. Посмотрим на результат первых двух экспериментов, чтобы убедиться в том, что эгоистичная стратегия эксплуатирует кооперативную:



На графике видно, что кооперативный агент (Coop) получает -3 очка на каждом ходу, а эгоистичный агент (Def) — 0 очков на каждом ходу. Так как игра происходит детерминированно, нет необходимости тщательно подбирать гиперпараметры алгоритма amTFT, чтобы продемонстрировать улучшение результатов. Будем запускать 30 отыгрышей по четыре хода, а также положим $T = 0.5$ и $\alpha = 1$. Результаты третьего и шестого экспериментов совпадают и демонстрируют идеальную кооперацию:

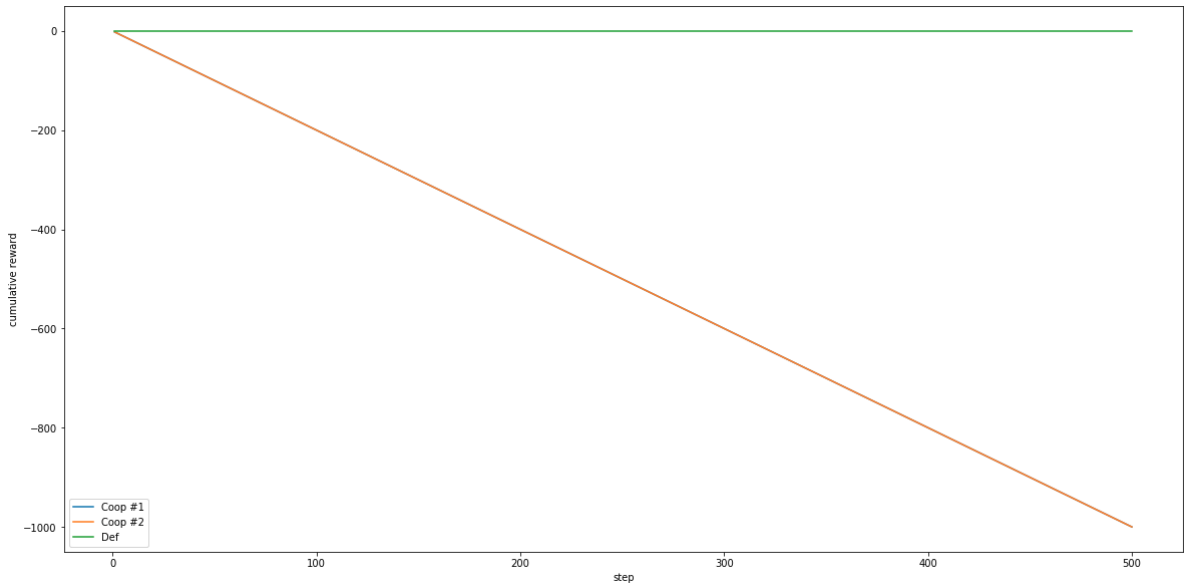


Результаты четвёртого и пятого экспериментов также совпадают и демонстрируют устойчивость к эксплуатации:

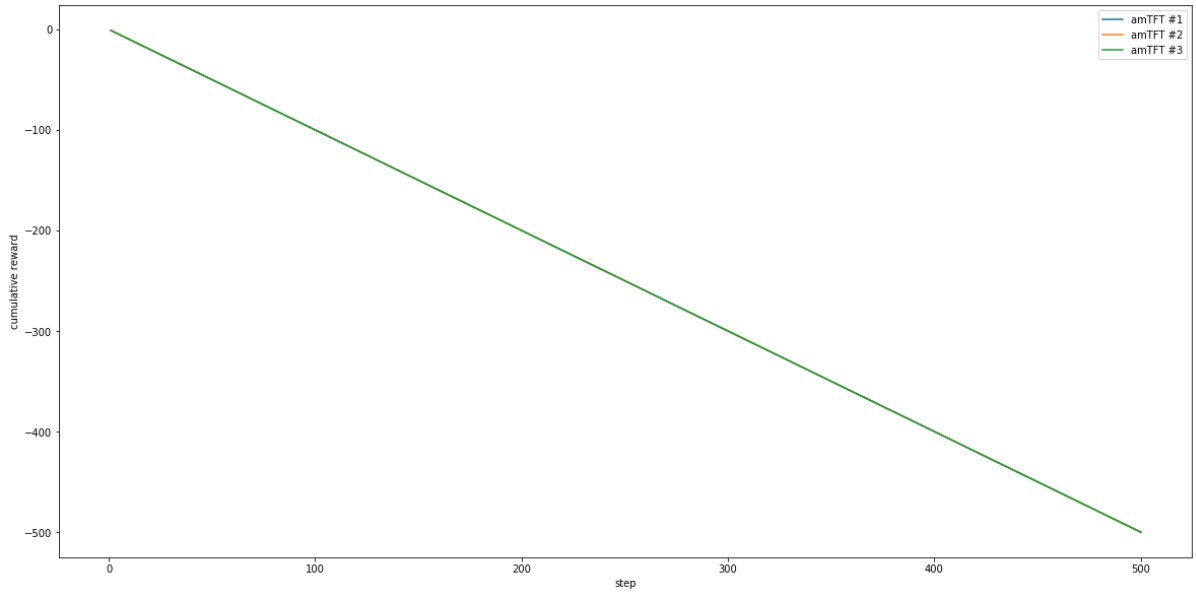


Нужно отметить, что параметр α контролирует итоговую разницу в количестве набранных очков, то есть путём его увеличения можно добиться меньшего преимущества эгоистичной стратегии. Другой способ сделать это — каждый раз домножать α на параметр $\beta > 1$.

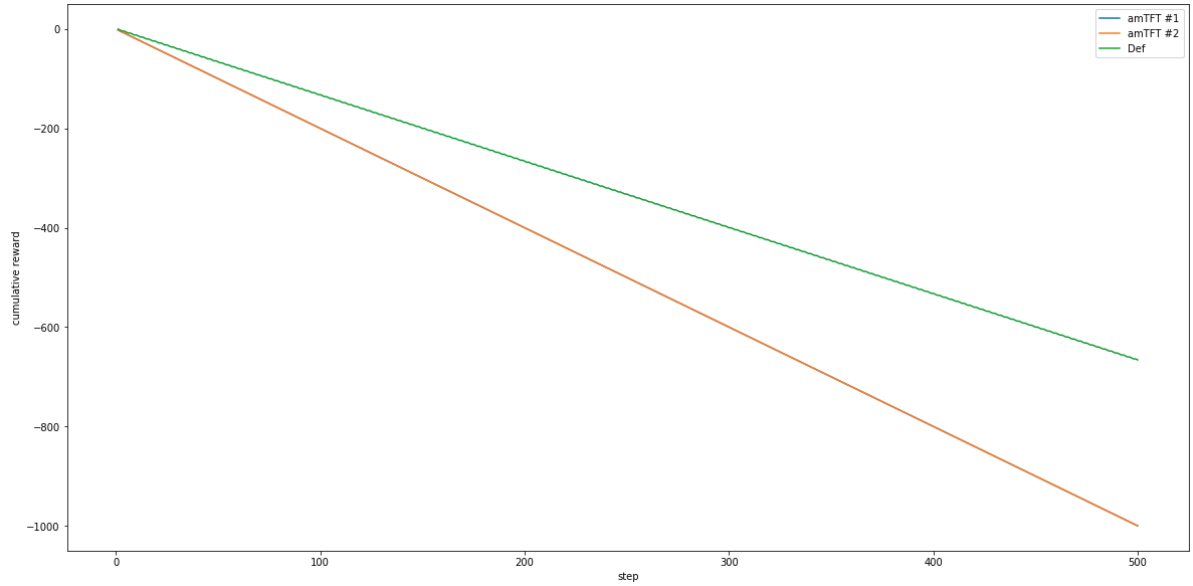
Рассмотрим версию игры для трёх игроков. Результаты всех экспериментов подобны вышеприведённым, поэтому графики будут даны без дополнительных пояснений. Первый и второй эксперименты (графики кооперативных агентов совпадают):



Третий и шестой эксперименты (графики агентов amTFT совпадают):



Четвёртый и пятый эксперименты (графики агентов amTFT вновь совпадают):



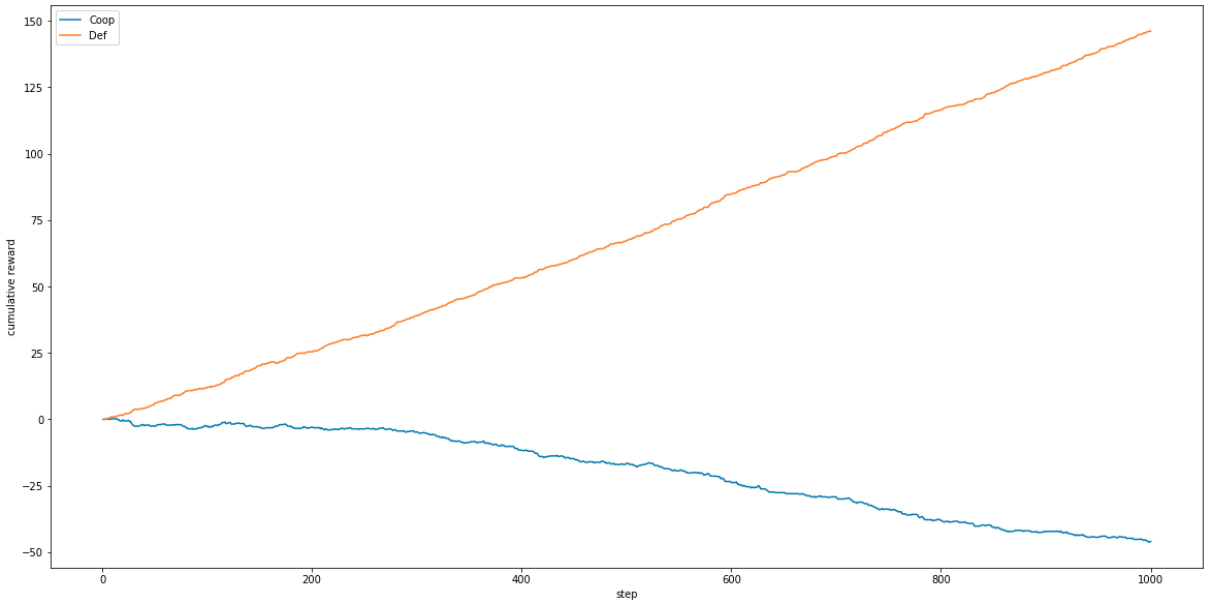
Можно заключить, что реализация алгоритма amTFT работает корректно. Перейдём к рассмотрению более сложной игры.

7.2 «Монетки»

В экспериментах мы будем рассматривать игру на поле 5×5, вероятность появления монетки положим равной 0.2. В качестве состояния игры возьмём $2n$ бинарных матриц 5×5, где n — количество игроков. Первые n матриц при этом задают позиции игроков, i-ая из них содержит единицу в клетке, соответствующей позиции i-ого игрока. Следующие n матриц задают позиции монеток, i-ая из них содержит единицы

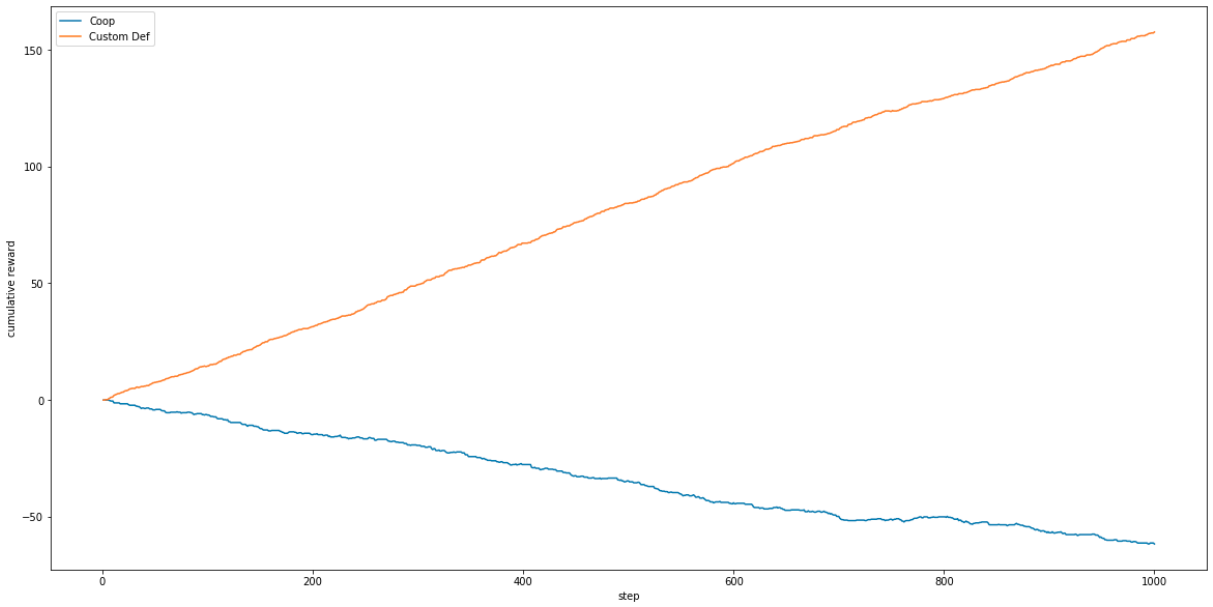
в клетках, соответствующих позициям монеток i-ого цвета. Теперь опишем конфигурацию используемой нейронной сети. Сначала следуют четыре свёрточных слоя. Размер свёртки для каждого из них равен трём, а отступ между свёртками (stride) равен двум для всех слоёв, кроме первого, для которого отступ равен единице. После применения каждого слоя количество каналов увеличивается в два раза. За каждым свёрточным слоем следует активационная функция ELU [5]. Далее следует скрытый линейный слой из 32 нейронов, за которым вновь следует ELU, за которым следует выходной линейный слой из четырёх нейронов. Будем использовать следующие параметры для Q-обучения: коэффициент скорости обучения α равен 10^{-4}; дисконтирующий коэффициент γ равен 0.95; коэффициент исследования ε уменьшается от 0.9 до 0.05 с экспоненциальным шагом $1 - 10^{-6}$. Обучение состоит из 1000-2000 запусков по 2000-5000 ходов. В качестве вручную запрограммированной эгоистичной стратегии будем использовать агента, который каждый раз идёт в направлении ближайшей монетки (или к центру поля, если монеток нет). В качестве вручную запрограммированной кооперативной стратегии будем использовать агента, который каждый раз идёт в направлении монетки своего цвета, при этом стараясь обходить чужие монетки. Для вычисления маршрута используется алгоритм Дейкстры [6], детали реализации можно найти в классе `CoinsGreedyAgent`.

Рассмотрим версию игры для двух игроков и выполним обучение агентов. Путём непосредственного изучения процесса игры можно убедиться, что эгоистичный агент старается собирать все монетки, а кооперативный — только монетки своего цвета. Результат первого эксперимента выглядит следующим образом:

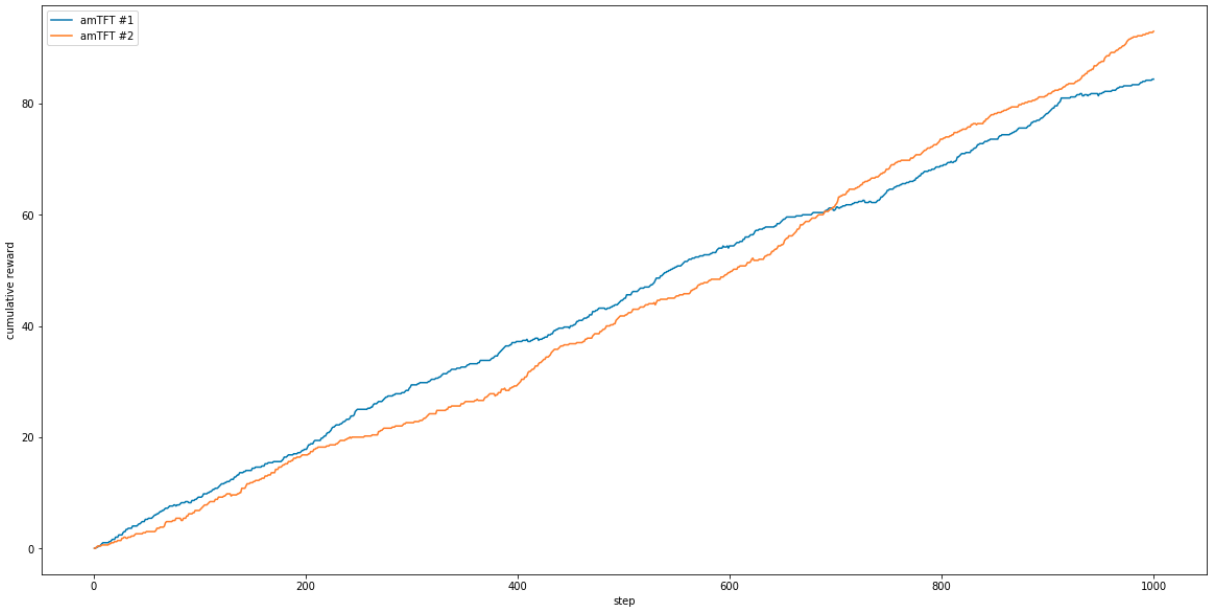


На графике видно, что эгоистичный агент (Def) набирает значительно больше очков,

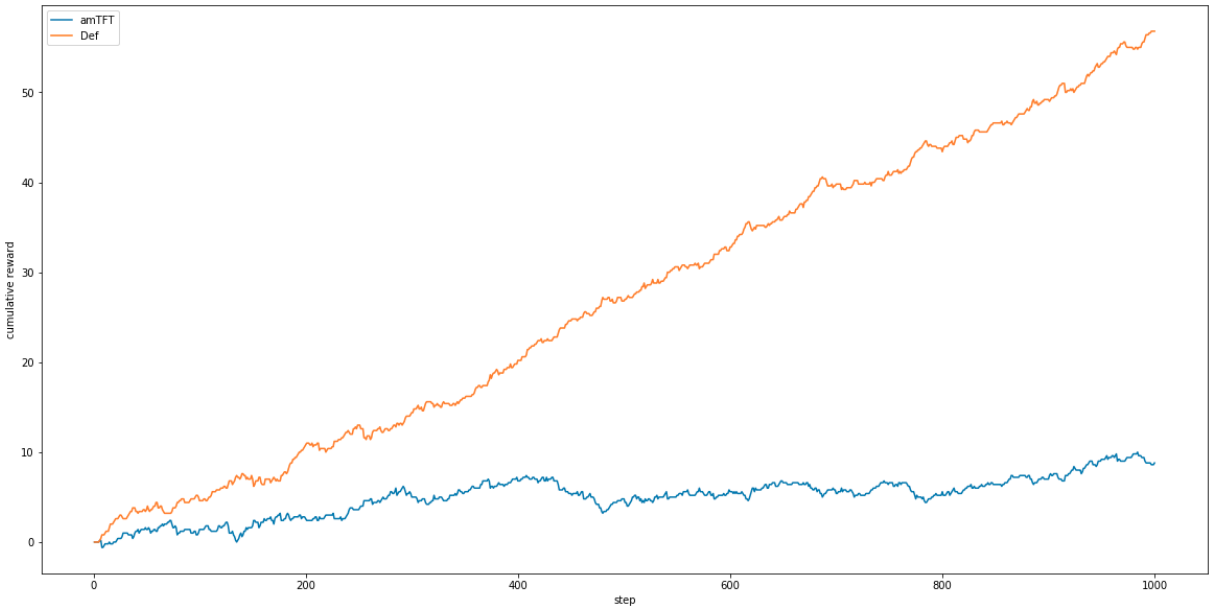
чем кооперативный агент (Coop). Обученная эгоистичная стратегия действует неидеально, поэтому, как показывает второй эксперимент, вручную запрограммированная эгоистичная стратегия достигает ещё большей разницы в счёте:



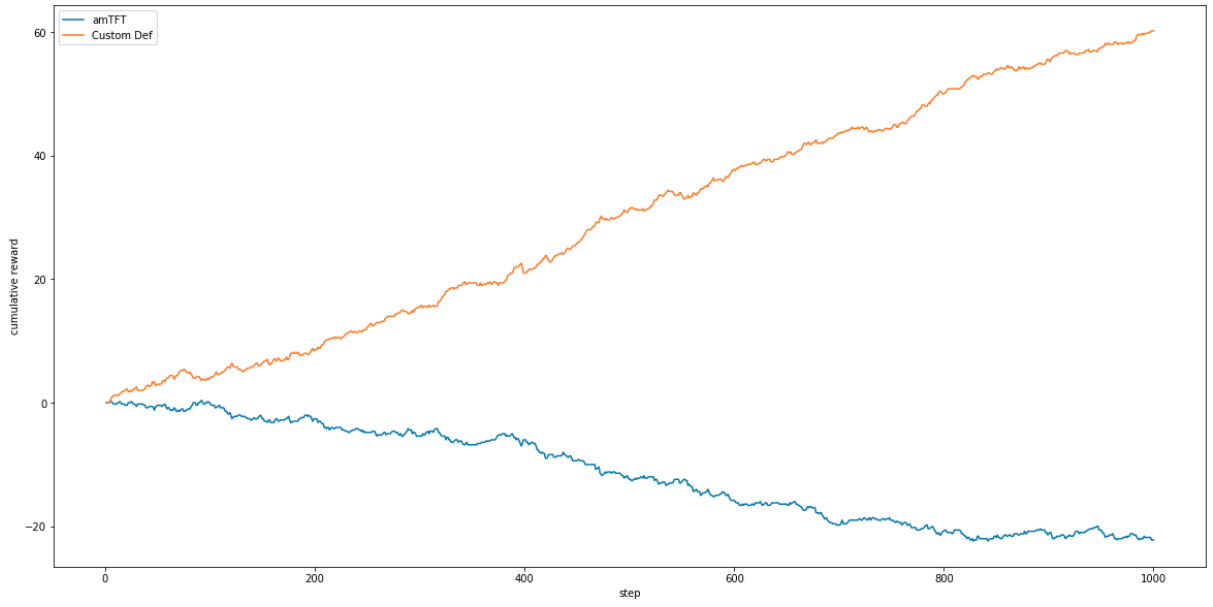
Теперь рассмотрим эксперименты с участием агентов amTFT. Положим $T = 1$ и $\alpha = 3$. Будем запускать 30 отыгрышей по 50 ходов, чтобы позволить агенту уходить в эгоистичный режим на длительное время. Результат третьего эксперимента выглядит следующим образом:



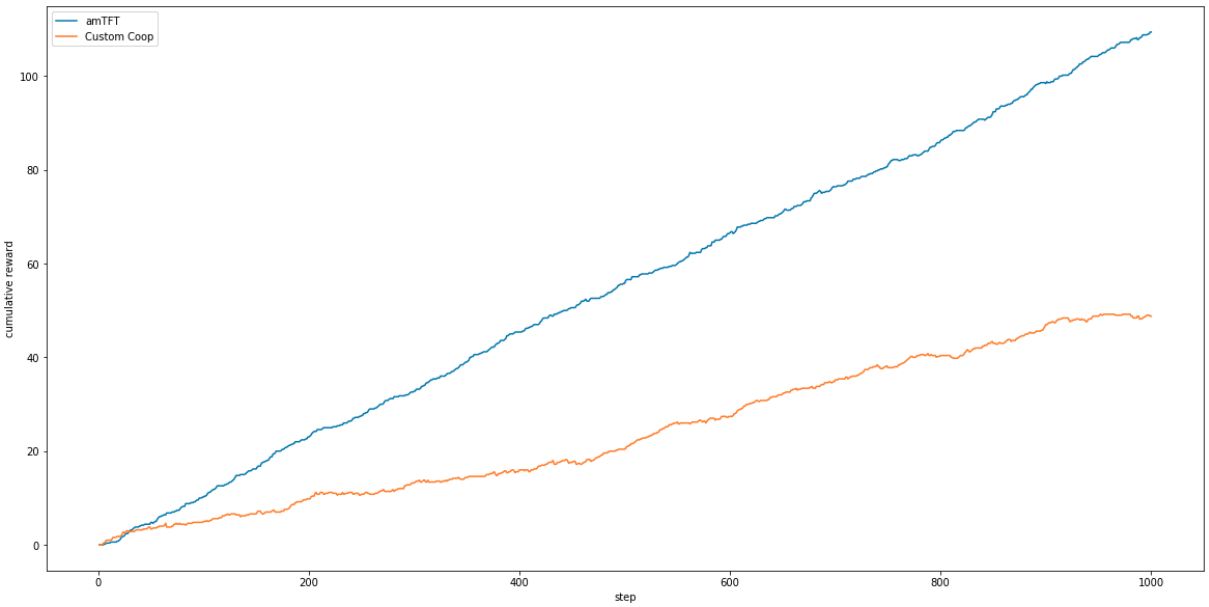
Видно, что агенты amTFT успешно кооперируются, играя в кооперативном режиме. Четвёртый эксперимент проверяет их способность противостоять эксплуатации:



Можно заметить, что эгоистичный агент получает меньшее преимущество, играя с агентом amTFT. То же самое можно наблюдать в пятом эксперименте:

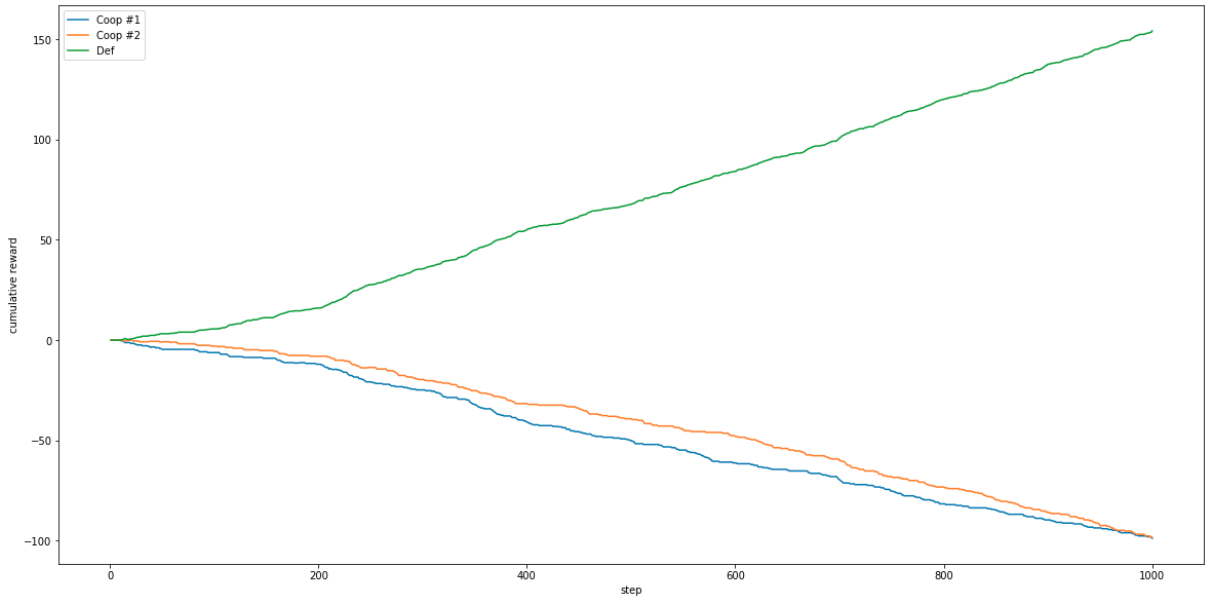


Посмотрим, сможет ли агент amTFT кооперироваться с неизвестной стратегией. Так выглядит результат шестого эксперимента:

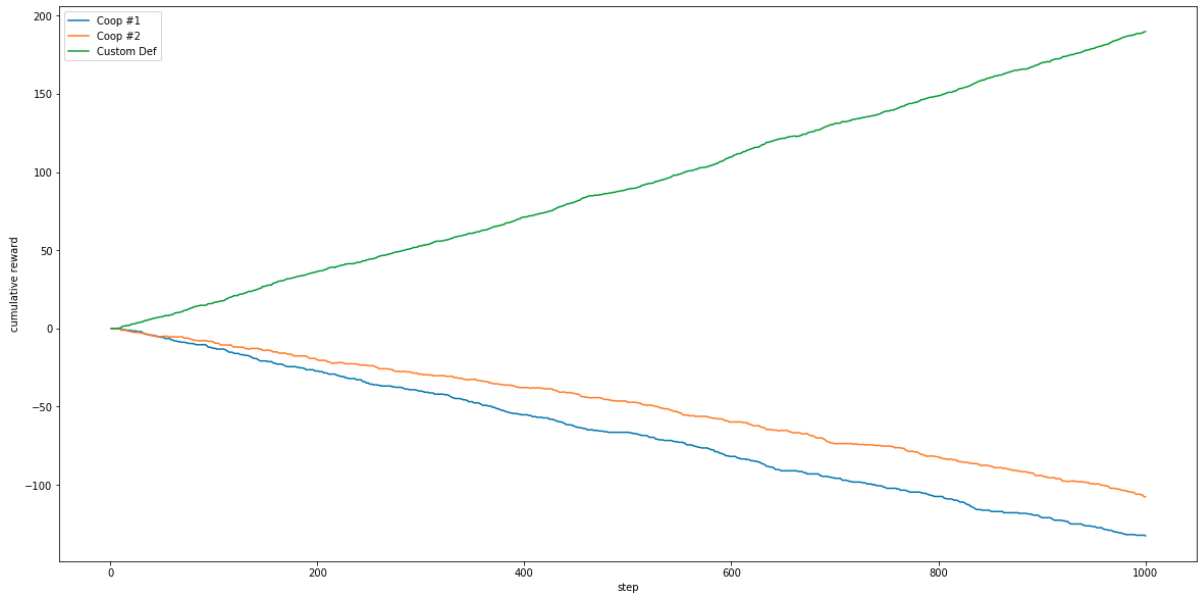


Ошибки аппроксимации приводят к тому, что агент amTFT время от времени переходит в эгоистичный режим. Но большую часть ходов можно наблюдать кооперацию агентов.

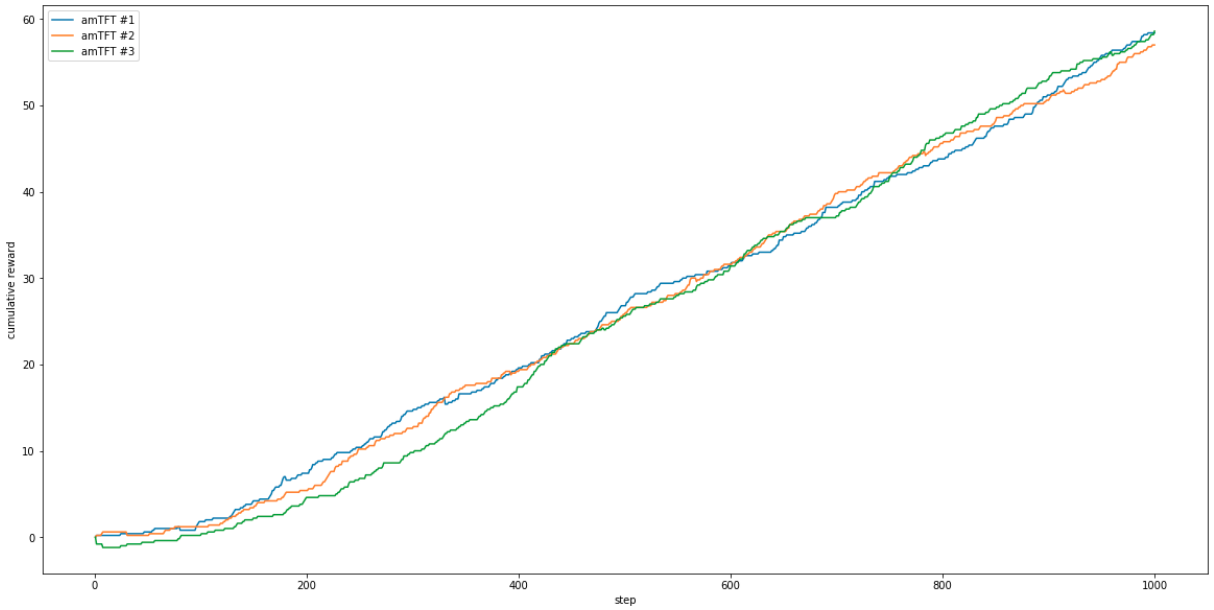
Рассмотрим версию игры для трёх игроков. Результаты большинства экспериментов подобны вышеприведённым, поэтому некоторые графики будут даны без пояснений. Первый эксперимент:



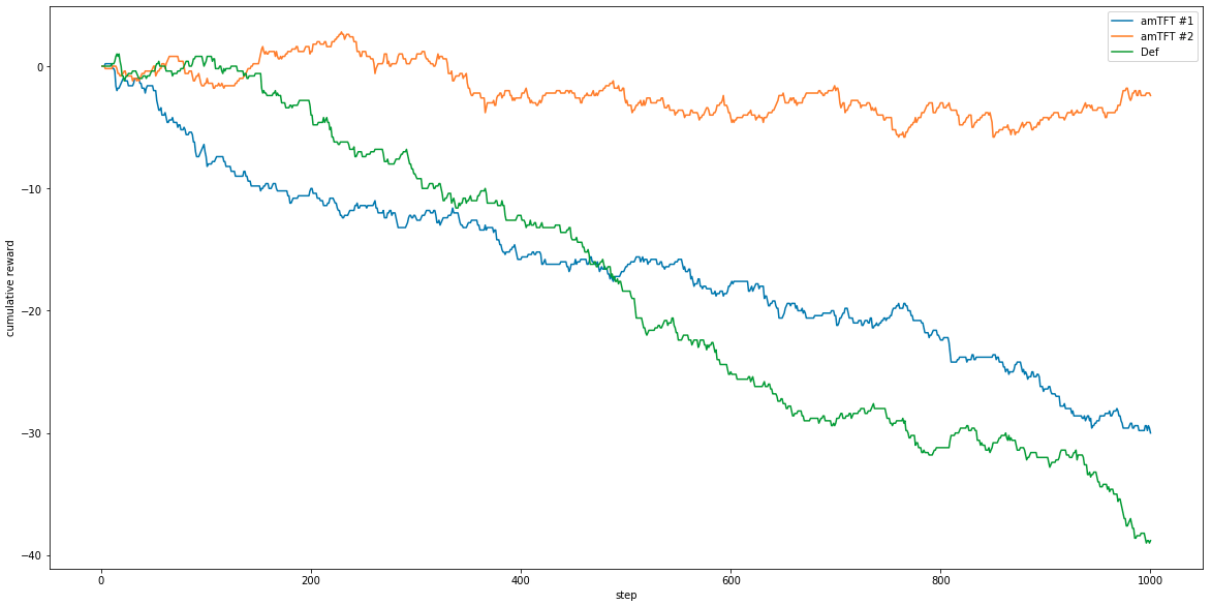
Второй эксперимент:



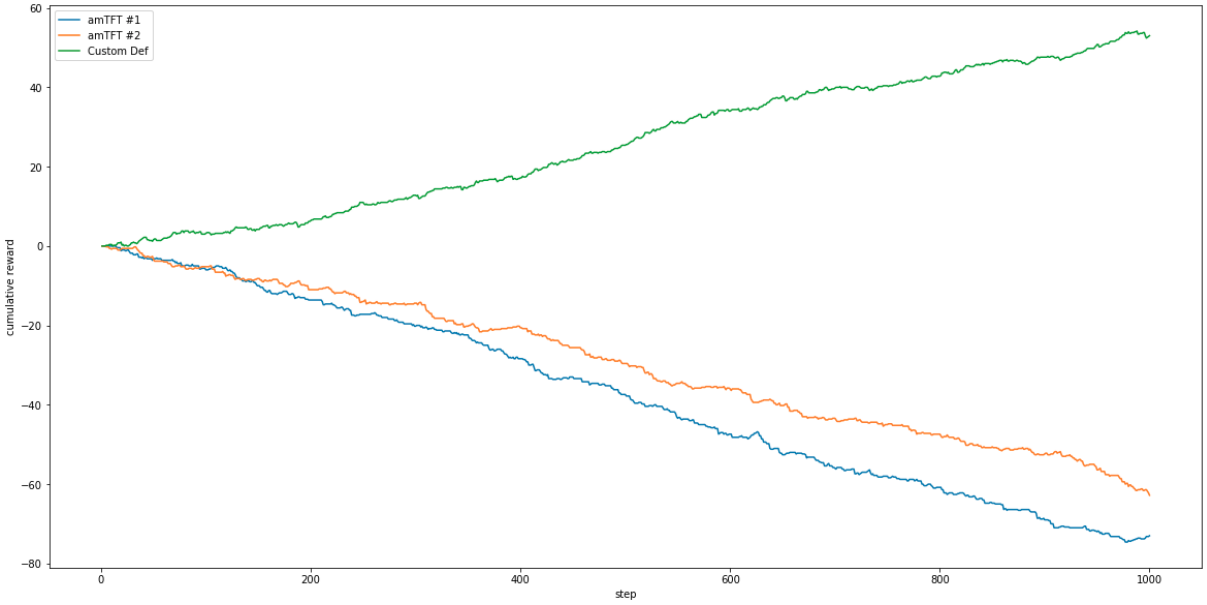
Можно заметить, что разница между обученной и вручную запрограммированной эгоистичными стратегиями стала больше, так как сложность игры растёт вместе с количеством игроков. Перейдём к третьему эксперименту:



Результаты четвёртого эксперимента выглядят неожиданно:



Дело в том, что стратегии плохо адаптировались к смене стратегии оппонента после обучения. Пятый эксперимент показывает ожидаемый с теоретической точки зрения результат:



Шестой эксперимент приводит к неожиданному исходу:

Видно, что агенты кооперируются на некоторых ходах, но ошибки аппроксимации не дают количеству очков значительно вырасти. Возможное решение состоит в том, чтобы повысить её точность.

8 Заключение

В данной работе была рассмотрена задача о построении оптимальной стратегии в марковских социальных дилеммах для произвольного числа игроков. В ходе работы была реализована библиотека для проведения экспериментов с играми этого класса. Теоретические рассуждения и результаты поставленных экспериментов показывают, что алгоритм amTFT позволяет успешно обобщить стратегию tit-for-tat [2] на более сложные игры.

Существуют и другие подходы к решению поставленной задачи (например, [7]). Дальнейшие исследования могут быть направлены как на их изучение, так и на разработку новых алгоритмов.

9 Список литературы

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens,

Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Robert Axelrod et al. The evolution of strategies in the iterated prisoner's dilemma. *The dynamics of norms*, pages 1–16, 1987.

[3] Kirill Borozdin. Cooperative RL framework, amTFT implementation and experiments. `https://github.com/kborozdin/cooperative_rl_amtft`, 2019.

[4] Pablo Samuel Castro, Subhodeep Moitra, Carles Gelada, Saurabh Kumar, and Marc G. Bellemare. Dopamine: A Research Framework for Deep Reinforcement Learning. 2018.

[5] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[6] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[7] Jakob Foerster, Richard Y Chen, Maruan Al-Shedivat, Shimon Whiteson, Pieter Abbeel, and Igor Mordatch. Learning with opponent-learning awareness. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 122–130. International Foundation for Autonomous Agents and Multiagent Systems, 2018.

[8] James W Friedman. A non-cooperative equilibrium for supergames. *The Review of Economic Studies*, 38(1):1–12, 1971.

[9] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87 – 90. IOS Press, 2016.

[10] Alexander Kuhnle, Michael Schaarschmidt, and Kai Fricke. Tensorforce: a tensorflow library for applied reinforcement learning. `https://github.com/tensorforce/tensorforce`, 2017.

[11] Adam Lerer and Alexander Peysakhovich. Maintaining cooperation in complex social dilemmas using deep reinforcement learning. *arXiv preprint arXiv:1707.01068*, 2017.

[12] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[14] John Nash. Non-cooperative games. *Annals of mathematics*, pages 286–295, 1951.

[15] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[16] Tuomas W Sandholm and Robert H Crites. Multiagent reinforcement learning in the iterated prisoner's dilemma. *Biosystems*, 37(1-2):147–166, 1996.

[17] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[19] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

10 Приложение

Данное приложение содержит исходный код библиотеки и экспериментов [3].

1 Library

In [ ]:
```
from collections import namedtuple, defaultdict, OrderedDict
from copy import deepcopy
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from time import sleep
from IPython.display import clear_output
from pathos.pools import ParallelPool, ProcessPool
from queue import PriorityQueue


device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
cpu_device = torch.device('cpu')


def np_locmax(a):
    return tuple(np.array(np.unravel_index(a.argmax(), a.shape)))


def np_max_except_idx(a, idx):
    if idx == 0:
        return a[1:].max()
    if idx == len(a) - 1:
        return a[:-1].max()
    return max(a[:idx].max(), a[idx+1:].max())


def np_random_generate_seeds(size):
    return np.random.randint(2**32, size=size)
```

In [ ]:
```
class BaseEnvironment:
    def get_num_agents(self):
        raise NotImplementedError

    def get_num_actions(self):
        raise NotImplementedError

    def get_state(self):
        raise NotImplementedError

    def reset(self, state=None):
        pass
```

```
        def set_seed(self, seed=None):
            pass

        def execute(self, actions):
            pass
```

In [ ]:
```
class BasePolicy:
    def sample(self, state):
        raise NotImplementedError


class OneHotPolicy(BasePolicy):
    def __init__(self, action):
        self.action = action

    def sample(self, state):
        return self.action


class ConstantPolicy(BasePolicy):
    def __init__(self, distribution):
        self.distribution = distribution / distribution.sum()

    def sample(self, state):
        return np.random.choice(np.arange(self.distribution.size),
                                p=self.distribution)


class LambdaPolicy(BasePolicy):
    def __init__(self, func):
        self.func = func

    def sample(self, state):
        return self.func(state).sample(state)


class CompoundPolicy(BasePolicy):
    def __init__(self, fst_policy, snd_policy, change_after):
        self.fst_policy = fst_policy
        self.snd_policy = snd_policy
        self.change_after = change_after
        self.turns = 0

    def sample(self, state):
        if self.turns < self.change_after:
            policy = self.fst_policy
        else:
            policy = self.snd_policy
```

```
                self.turns += 1
                return policy.sample(state)
```

In [ ]:
```
class BaseAgent:
    def get_policy(self, state, do_exploration=True):
        raise NotImplementedError

    def act(self, state):
        return self.get_policy(state).sample(state)

    def on_run_start(self):
        pass

    def observe(self, prev_state, action, reward, next_state):
        pass

    def get_observe_all_actions(self):
        return False

    def get_run_loss(self):
        return 0

    def cross_mutate(self, other):
        raise NotImplementedError


class PolicyAgent(BaseAgent):
    def __init__(self, policy):
        self.policy = policy
        self.current_policy = None

    def get_policy(self, state, do_exploration=True):
        return self.current_policy

    def on_run_start(self):
        self.current_policy = deepcopy(self.policy)


def get_pure_policy(agent, do_exploration=False):
    return LambdaPolicy(lambda s:
                        agent.get_policy(s, do_exploration=do_exploration))
```

In [ ]:
```
class ReplayBuffer:
    def __init__(self, max_size):
        self.storage = []
        self.max_size = max_size
        self.pointer = 0
```

```
            def __len__(self):
                return len(self.storage)

            def add(self, prev_state, actions, rewards, next_state):
                data = (prev_state, actions, rewards, next_state)
                if len(self.storage) < self.max_size:
                    self.storage.append(data)
                else:
                    self.storage[self.pointer] = data
                    self.pointer = (self.pointer + 1) % self.max_size

            def sample(self, batch_size):
                idxes = np.random.choice(len(self.storage),
                                         size=batch_size, replace=False)
                batch = []
                for idx in idxes:
                    batch.append(self.storage[idx])
                return tuple(zip(*batch))
```

```
In [ ]: def split_equally(length, num_parts):
            parts = []
            part_len = length // num_parts
            num_bigger_parts = length % num_parts
            start_idx = 0
            while start_idx < length:
                end_idx = start_idx + part_len
                if start_idx // part_len < num_bigger_parts:
                    end_idx += 1
                parts.append((start_idx, end_idx))
                start_idx = end_idx
            return parts


        class Runner:
            def __init__(self, num_runs, num_steps, batch_size=1, replay_size=0,
                         replay_batch_size=0, coop_reward=False, agent_names=None,
                         visualize=False, visualize_every=1, visualize_gameplay=False,
                         visualize_gameplay_delay=0.5, vis_summary_only=False,
                         num_workers=1):
                assert not visualize or num_workers == 1, 'not supported'
                self.num_runs = num_runs
                self.num_steps = num_steps
                self.batch_size = batch_size
                self.replay_size = replay_size
                self.replay_batch_size = replay_batch_size or batch_size
                self.coop_reward = coop_reward
                self.agent_names = agent_names
                self.visualize = visualize
```

```
        self.visualize_every = visualize_every
        self.visualize_gameplay = visualize_gameplay
        self.visualize_gameplay_delay = visualize_gameplay_delay
        self.vis_summary_only = vis_summary_only
        self.num_workers = num_workers

    def get_num_runs(self):
        return self.num_runs

    def get_num_steps(self):
        return self.num_steps

    def run(self, env, agents, rnd_seeds=None):
        if rnd_seeds is None:
            rnd_seeds = np_random_generate_seeds(self.num_runs)
        assert len(rnd_seeds) == self.num_runs
        assert self.agent_names is None or len(self.agent_names) == len(agents)
        if self.visualize:
            if self.vis_summary_only:
                plt.figure(figsize=[20, 10])
            else:
                num_plots = self.num_runs // self.visualize_every
                if num_plots > 0:
                    plt.figure(figsize=[25, 3 * num_plots])

        def single_run_wrapper(indices):
            all_rewards = np.zeros([len(agents), self.num_steps])
            for run_idx in range(indices[0] + 1, indices[1] + 1):
                if self.should_visualize_on(run_idx):
                    print('Run #{}...'.format(run_idx))
                    if not self.vis_summary_only:
                        plt.subplot(num_plots, 1,
                                    run_idx // self.visualize_every)
                env_copy = deepcopy(env)
                if rnd_seeds is not None:
                    env_copy.set_seed(rnd_seeds[run_idx - 1])
                all_rewards += self.single_run(run_idx, env_copy, agents)
            return all_rewards

        index_pairs = split_equally(self.num_runs, self.num_workers)
        if self.num_workers > 1:
            with ProcessPool(nodes=self.num_workers) as pool:
                all_rewards_list = pool.map(single_run_wrapper, index_pairs)
        else:
            all_rewards_list = list(map(single_run_wrapper, index_pairs))

        all_rewards = np.zeros([len(agents), self.num_steps])
        for all_rewards_elem in all_rewards_list:
```

```
            all_rewards += all_rewards_elem
        all_rewards /= self.num_runs
        if self.visualize and self.vis_summary_only:
            agent_names = self.get_agent_names(agents)
            for agent_name, agent_rewards in zip(agent_names, all_rewards):
                plt.plot(np.arange(1, self.num_steps + 1),
                         np.cumsum(agent_rewards), label=agent_name)
            plt.xlabel('step')
            plt.ylabel('cumulative reward')
            plt.legend()
        return all_rewards.sum(axis=1)

    def single_run(self, run_idx, env, agents):
        for agent in agents:
            agent.on_run_start()

        all_rewards = [[] for _ in range(len(agents))]
        batch_buffer = ReplayBuffer(self.batch_size)
        if self.replay_size:
            replay_buffer = ReplayBuffer(self.replay_size)

        if self.should_visualize_on(run_idx) and self.visualize_gameplay:
            clear_output(True)
            print(env)

        for step_idx in range(1, self.num_steps + 1):
            prev_state = env.get_state()
            actions = [a.act(prev_state) for a in agents]
            rewards = env.execute(actions)
            next_state = env.get_state()

            for agent_idx in range(len(agents)):
                all_rewards[agent_idx].append(rewards[agent_idx])
                batch_buffer.add(prev_state, actions, rewards, next_state)
                if self.replay_size:
                    replay_buffer.add(prev_state, actions, rewards, next_state)

            if step_idx % self.batch_size == 0:
                self.sample_and_observe(agents, batch_buffer, self.batch_size,
                                        self.coop_reward)
                if self.replay_size:
                    self.sample_and_observe(agents, replay_buffer,
                                            self.replay_batch_size,
                                            self.coop_reward)

            if self.should_visualize_on(run_idx) and self.visualize_gameplay:
                sleep(self.visualize_gameplay_delay)
                clear_output(True)
```

```
                print(env)

        all_rewards = np.array(all_rewards)
        if self.should_visualize_on(run_idx):
            print('Total rewards:', all_rewards.sum(axis=1),
                  '| Total losses:', [a.get_run_loss() for a in agents])
            if not self.vis_summary_only:
                agent_names = self.get_agent_names(agents)
                for agent_idx in range(len(agents)):
                    plt.plot(np.arange(1, self.num_steps + 1),
                             all_rewards[agent_idx], label=agent_names[agent_idx])
                plt.xlabel('step')
                plt.ylabel('reward')
                plt.legend()
        return all_rewards

    @staticmethod
    def sample_and_observe(agents, buffer, size, coop_reward):
        size = min(size, len(buffer))
        prev_states, actions, rewards, next_states = buffer.sample(size)
        actions, rewards = np.array(actions), np.array(rewards)
        for agent_idx in range(len(agents)):
            if agents[agent_idx].get_observe_all_actions():
                action_arg = actions
            else:
                action_arg = actions[:, agent_idx]
            if coop_reward:
                reward_arg = rewards.sum(axis=1)
            else:
                reward_arg = rewards[:, agent_idx]
            agents[agent_idx].observe(np.stack(prev_states), action_arg,
                                      reward_arg, np.stack(next_states))

    def get_agent_names(self, agents):
        return self.agent_names or [str(idx) for idx in range(1, len(agents) + 1)]

    def should_visualize_on(self, run_idx):
        return self.visualize and run_idx % self.visualize_every == 0
```

In [ ]:
```
def merge_state_dicts(first, second):
    result = OrderedDict()
    for (name1, val1), (name2, val2) in zip(first.items(), second.items()):
        assert name1 == name2
        result[name1] = (val1 + val2) / 2
    return result


class DQNAgent(BaseAgent):
```

```
def __init__(self, num_actions, network, double_dqn_freq=None, lr=1e-3,
             gamma=0.95, use_eps_greedy=True, eps_init=0.9,
             eps_mult=0.99, eps_min=0.01, eps_reset=False,
             device=device):
    self.network = network
    self.frozen_network = None
    self.double_dqn_freq = double_dqn_freq
    self.double_dqn_freq_counter = None
    self.optimizer = optim.Adam(self.network.parameters(), lr=lr)
    self.gamma = gamma
    self.use_eps_greedy = use_eps_greedy
    self.eps_init = eps_init
    self.eps_mult = eps_mult
    self.eps_min = eps_min
    self.eps_reset = eps_reset
    self.eps = None
    self.run_loss = None
    self.uniform_distribution = np.ones(num_actions) / num_actions
    self.device = device

def get_policy(self, state, do_exploration=True):
    with torch.no_grad():
        state = torch.tensor(state, device=self.device).unsqueeze(0)
        q_values = self.network(state).squeeze(0).cpu().data.numpy()
    if self.use_eps_greedy or not do_exploration:
        if do_exploration:
            distribution = self.uniform_distribution * self.eps
        else:
            distribution = np.zeros_like(self.uniform_distribution)
        action = q_values.argmax()
        distribution[action] += 1 - self.eps if do_exploration else 1
    else:
        distribution = np.exp(q_values)
    return ConstantPolicy(distribution)

def on_run_start(self):
    if self.eps_reset or self.eps is None:
        self.eps = self.eps_init
    self.run_loss = 0

def uses_double_dqn(self):
    return self.double_dqn_freq is not None

def observe(self, prev_state, action, reward, next_state):
    if self.uses_double_dqn():
        if not self.double_dqn_freq_counter:  # Zero or None
            self.frozen_network = None
        if self.frozen_network is None:
```

```
                    self.frozen_network = deepcopy(self.network)
                    self.double_dqn_freq_counter = self.double_dqn_freq
                self.double_dqn_freq_counter -= 1

            prev_state = torch.tensor(prev_state, device=self.device)
            action = torch.tensor(action, device=self.device)
            next_state = torch.tensor(next_state, device=self.device)
            reward = torch.tensor(reward, dtype=torch.float, device=self.device)

            old_q_value = self.network(
                prev_state).gather(1, action.unsqueeze(dim=1)).squeeze(1)
            target_network = self.frozen_network if self.uses_double_dqn()\
                             else self.network
            with torch.no_grad():
                new_q_value = reward + self.gamma * target_network(
                    next_state).max(1)[0]
            loss = F.smooth_l1_loss(old_q_value, new_q_value)
            self.run_loss += loss.to(cpu_device).data.numpy()

            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()

            self.eps *= self.eps_mult ** len(prev_state)
            self.eps = max(self.eps, self.eps_min)

        def get_run_loss(self):
            return self.run_loss

        def cross_mutate(self, other):
            assert isinstance(other, DQNAgent)
            state_dict = merge_state_dicts(self.network.state_dict(),
                                           other.network.state_dict())
            merged = deepcopy(self)
            merged.network.load_state_dict(state_dict)
            return merged
```

In [ ]:
```
class AMTFTAgent(BaseAgent):
    def __init__(self, env_copy, runner, agent_idx, coop_policies, def_policies,
                 def_limit, init_def_penalty, def_penalty_mult=1.0,
                 length_bs_tol=2, optimize_defect_est=True, debug=0):
        self.env = env_copy
        self.runner = runner
        self.agent_idx = agent_idx
        self.coop_policies = coop_policies
        self.def_policies = def_policies
        self.def_limit = def_limit
        self.init_def_penalty = init_def_penalty
```

```
        self.def_penalty_mult = def_penalty_mult
        self.length_bs_tol = length_bs_tol
        self.optimize_defect_est = optimize_defect_est
        self.debug = debug
        self.def_penalty = None
        self.balance = None
        self.def_accum = None

    def get_policy(self, state, do_exploration=True):
        if self.balance == 0:
            return self.coop_policies[self.agent_idx]
        return self.def_policies[self.agent_idx]

    def on_run_start(self):
        self.def_penalty = self.init_def_penalty
        self.balance = 0
        self.def_accum = 0

    def observe(self, prev_state, action, reward, next_state):
        [prev_state] = prev_state
        [actions] = action
        [reward] = reward
        [next_state] = next_state

        if self.balance > 0:
            self.balance -= 1
        else:
            self.def_accum += self.estimate_defect(prev_state, actions)
            if self.def_accum > self.def_limit:
                self.balance = self.estimate_max_balance(
                    next_state, self.def_penalty * self.def_limit)
                self.def_accum = 0
                self.def_penalty *= self.def_penalty_mult

        if self.debug >= 2:
            print('agent_idx is', self.agent_idx)
            print('balance is', self.balance, 'def_accum is', self.def_accum)

    def get_observe_all_actions(self):
        return True

    def estimate_defect(self, state, actions):
        expected_actions = [p.sample(state) for p in self.coop_policies]
        if self.optimize_defect_est and (actions == expected_actions).all():
            return 0
        seeds = np_random_generate_seeds(self.runner.get_num_runs())
        return max(0, np_max_except_idx(
            self.estimate_q(state, self.coop_policies, actions, seeds) -
```

```
                self.estimate_v(state, self.coop_policies, seeds),
                self.agent_idx))

        def estimate_max_balance(self, state, min_diff):
            if self.debug >= 1:
                print('estimate balance: agent_idx is',
                      self.agent_idx, 'min_diff is', min_diff)
            seeds = np_random_generate_seeds(self.runner.get_num_runs())
            left, right = 0, (self.runner.get_num_steps() +
                              self.length_bs_tol - 1) // self.length_bs_tol
            while right - left > 1:
                mid = (left + right) // 2
                length = mid * self.length_bs_tol
                compound_policies = [CompoundPolicy(d, c, length)
                                     for c, d in zip(self.coop_policies,
                                                     self.def_policies)]
                actual_diff = np_max_except_idx(
                    self.estimate_v(state, self.coop_policies, seeds) -
                    self.estimate_v(state, compound_policies, seeds),
                    self.agent_idx)
                if self.debug >= 1:
                    print('try: length is', length, 'actual_diff is', actual_diff)
                if actual_diff >= min_diff:
                    right = mid
                else:
                    left = mid
            length = right * self.length_bs_tol
            if self.debug >= 1:
                print('estimated balance is', length)
            return length

        def estimate_q(self, state, policies, actions, seeds):
            one_hot_policies = [OneHotPolicy(a) for a in actions]
            return self.estimate_v(
                state, [CompoundPolicy(oh, p, 1)
                        for oh, p in zip(one_hot_policies, policies)], seeds)

        def estimate_v(self, state, policies, seeds):
            self.env.reset(state)
            agents = [PolicyAgent(p) for p in policies]
            return self.runner.run(self.env, agents, rnd_seeds=seeds)
```

In [ ]:
```
# Unused
class EvolutionaryTournament:
    AgentWithLifetime = namedtuple('AgentWithLifetime', ['agent', 'lifetime'])

    def __init__(self, env, agent_factory, num_places, num_high_deaths,
                 num_low_deaths, num_gens, num_steps, batch_size,
```

```
                 replay_size=0, replay_batch_size=0, visualize=False,
                 table_cell_width=15):
        assert num_places - num_high_deaths - num_low_deaths >= 2
        self.env = env
        self.agent_factory = agent_factory
        self.num_places = num_places
        self.num_high_deaths = num_high_deaths
        self.num_low_deaths = num_low_deaths
        self.num_gens = num_gens
        self.batch_size = batch_size
        self.replay_size = replay_size
        self.replay_batch_size = replay_batch_size
        self.visualize = visualize
        self.table_cell_width = table_cell_width
        self.runner = Runner(num_runs=1, num_steps=num_steps)

    def run(self):
        agents = [self.AgentWithLifetime(self.agent_factory(), 0)
                  for _ in range(self.num_places)]

        for gen_idx in range(1, self.num_gens + 1):
            scores = [0] * self.num_places
            scoreboard = [[0] * self.num_places for _ in range(self.num_places)]
            for first_idx in range(self.num_places):
                for second_idx in range(first_idx + 1, self.num_places):
                    totals = self.runner.run(
                        env=env,
                        agents=[agents[first_idx].agent,
                                agents[second_idx].agent],
                        batch_size=self.batch_size,
                        replay_size=self.replay_size,
                        replay_batch_size=self.replay_batch_size)
                    scoreboard[first_idx][second_idx] = totals
                    scores[first_idx] += totals[0]
                    scores[second_idx] += totals[1]

            ranklist = sorted([(scores[i], i, p) for i, p in enumerate(agents)],
                              reverse=True)
            top = [x[2] for x in ranklist[self.num_high_deaths:
                                          self.num_places-self.num_low_deaths]]
            agents = [self.AgentWithLifetime(p.agent, p.lifetime + 1)
                      for p in top] +\
                     [self.AgentWithLifetime(self.cross_mutate(top), 0)
                      for _ in
                      range(self.num_high_deaths + self.num_low_deaths)]

            if self.visualize:
                if gen_idx > 0:
```

```
                print()
            print('Gen #{}'.format(gen_idx))
            TEMPLATE = '{{:{}}}'.format(self.table_cell_width) *\
                       (self.num_places + 1)
            print(TEMPLATE.format('scoreboard',
                                  *map(str, range(1, self.num_places + 1))))
            for first_idx in range(self.num_places):
                elems = [str(first_idx + 1)]
                for second_idx in range(self.num_places):
                    if first_idx == second_idx:
                        elem = ['###']
                    elif first_idx < second_idx:
                        elem = scoreboard[first_idx][second_idx]
                    else:
                        elem = list(reversed(
                            scoreboard[second_idx][first_idx]))
                    elems.append(':'.join(map(str, elem)))
                print(TEMPLATE.format(*elems))
            for idx, row in enumerate(ranklist, 1):
                print('{}) id = {}, score = {}, lifetime = {}'.format(
                    idx, row[1] + 1, row[0], row[2].lifetime))

        return agents

    def cross_mutate(self, agents):
        idx0, idx1 = np.random.choice(len(agents), size=2, replace=False)
        return agents[idx0].agent.cross_mutate(agents[idx1].agent)
```

2 IPD

```
In [ ]: class IPDEnvironment(BaseEnvironment):
            DEFECT = 0
            COOPERATE = 1
            NUM_ACTIONS = 2

            def __init__(self, num_agents, memory_size):
                assert num_agents >= 2
                self.num_agents = num_agents
                self.memory_size = memory_size
                self.memory_cell_size = self.get_num_actions() ** num_agents + 1
                self.memory_cell_unk = self.memory_cell_size - 1
                if num_agents == 2:
                    self.rewards = [
                        [-2, 0, 0],
                        [0, -3, -1]
                    ]
                else:
```

```
                self.rewards = [
                    list(range(-num_agents + 1, 1)) + [0],
                    [0] + list(range(-num_agents, 0))
                ]
            self.reset()

        def get_num_agents(self):
            return self.num_agents

        def get_num_actions(self):
            return self.NUM_ACTIONS

        def get_state(self):
            return self.state.copy()

        def reset(self, state=None):
            if state is None:
                self.state = np.full(self.memory_size, self.memory_cell_unk)
            else:
                self.state = state.copy()

        def execute(self, actions):
            memory_cell = self.actions_to_memory_cell(actions)
            self.state = np.append(self.state[1:], memory_cell)
            return self.actions_to_rewards(actions)

        def actions_to_memory_cell(self, actions):
            cell = 0
            for action in actions:
                cell = cell * self.get_num_actions() + action
            return cell

        def actions_to_rewards(self, actions):
            num_cooperators = sum(actions)
            return [self.rewards[a][num_cooperators] for a in actions]

        def get_memory_desc(self):
            return (self.memory_size, self.memory_cell_size)
```

In [ ]:
```
class IPDNetwork(nn.Module):
    def __init__(self, ipd_env, emb_size=16, lstm_size=None, hid_size=16):
        super().__init__()
        self.emb = nn.Embedding(num_embeddings=ipd_env.get_memory_desc()[1],
                                embedding_dim=emb_size)
        cur_size = emb_size
        self.lstm = None
        if lstm_size is not None:
            self.lstm = nn.LSTM(input_size=cur_size, hidden_size=lstm_size,
```

```
                                    batch_first=True)
                cur_size = lstm_size
            if hid_size is not None:
                self.lin1 = nn.Linear(in_features=cur_size, out_features=hid_size)
                cur_size = hid_size
            self.lin2 = nn.Linear(in_features=cur_size,
                                  out_features=ipd_env.get_num_actions())

        def forward(self, x):
            x = self.emb(x)
            if self.lstm:
                x = self.lstm(x)[0][:, -1, :]
            else:
                x = x[:, -1]
            x.view(x.size(0), -1)
            if self.lin1:
                x = F.elu(self.lin1(x))
            return self.lin2(x)
```

```
In [ ]: class IPDTFTAgent(BaseAgent):
        def __init__(self, agent_idx):
            self.agent_idx = agent_idx
            self.last_action = None

        def on_run_start(self):
            self.last_action = IPDEnvironment.COOPERATE

        def get_policy(self, state, do_exploration=True):
            return OneHotPolicy(self.last_action)

        def observe(self, prev_state, action, reward, next_state):
            assert len(action) == 1
            self.last_action = IPDEnvironment.COOPERATE
            for idx, a in enumerate(action[0]):
                if idx == self.agent_idx:
                    continue
                if a == IPDEnvironment.DEFECT:
                    self.last_action = IPDEnvironment.DEFECT

        def get_observe_all_actions(self):
            return True
```

3 Coins

```
In [ ]: class CoinsEnvironment(BaseEnvironment):
        RIGHT = 0
        UP = 1
```

```
LEFT = 2
DOWN = 3
NUM_ACTIONS = 4

DIRECTIONS = np.array([
    [0, +1],
    [-1, 0],
    [0, -1],
    [+1, 0]
])

PICK_REWARD = +1
STEAL_REWARD = -2

def __init__(self, num_agents, size, toroidal=False, wall_hit_reward=0.0,
             coin_gen_prob=0.2):
    assert size >= 2
    self.num_agents = num_agents
    self.size = size
    self.toroidal = toroidal
    self.wall_hit_reward = wall_hit_reward
    self.coin_gen_prob = coin_gen_prob
    self.reset()
    self.set_seed()

def get_num_agents(self):
    return self.num_agents

def get_num_actions(self):
    return self.NUM_ACTIONS

def get_state(self):
    state = np.zeros([2 * self.num_agents, self.size, self.size])
    for idx in range(self.num_agents):
        state[idx][self.agents[idx]] = 1
    for coin_pos, coin_col in self.coins.items():
        state[self.num_agents + coin_col][coin_pos] = 1
    return state

def reset(self, state=None):
    self.coins = {}
    if state is None:
        self.agents = [(0, 0)] * self.num_agents
    else:
        for idx in range(self.num_agents):
            agent_state = state[idx]
            self.agents[idx] = np_locmax(agent_state)
        for idx in range(self.num_agents):
```

```
            coin_state = state[self.num_agents + idx]
            for row in range(self.size):
                for col in range(self.size):
                    if coin_state[row, col] > 0:
                        self.coins[(row, col)] = idx

    def set_seed(self, seed=None):
        if seed is None:
            [seed] = np_random_generate_seeds(1)
        self.rnd = np.random.RandomState(seed)

    def execute(self, actions):
        rewards = [0] * self.num_agents
        picked_coins = set()
        for idx in range(self.num_agents):
            next_pos = np.array(self.agents[idx]) +\
                       self.DIRECTIONS[actions[idx]]
            if self.toroidal:
                next_pos = (next_pos + self.size) % self.size
            if 0 <= next_pos.min() and next_pos.max() < self.size:
                self.agents[idx] = tuple(next_pos)
            else:
                rewards[idx] += self.wall_hit_reward
            if self.agents[idx] in self.coins:
                coin_col = self.coins[self.agents[idx]]
                picked_coins.add(self.agents[idx])
                rewards[idx] += self.PICK_REWARD
                if idx != coin_col:
                    rewards[coin_col] += self.STEAL_REWARD

        for coin_pos in picked_coins:
            del self.coins[coin_pos]
        if self.rnd.rand() < self.coin_gen_prob:
            coin_col = self.rnd.randint(self.num_agents)
            occupied_pos = set(self.agents) | set(self.coins)
            num_free_pos = self.size**2 - len(occupied_pos)
            if num_free_pos > 0:
                coin_pos_idx = self.rnd.randint(num_free_pos)
                found = False
                for row in range(self.size):
                    for col in range(self.size):
                        if (row, col) in occupied_pos:
                            continue
                        if coin_pos_idx == 0:
                            self.coins[(row, col)] = coin_col
                            found = True
                            break
                        coin_pos_idx -= 1
```

```
                        if found:
                            break
                    assert found

        return rewards

    def get_size(self):
        return self.size

    def __str__(self):
        board = [['.'] * self.size for _ in range(self.size)]
        for idx in range(self.num_agents):
            x, y = self.agents[idx]
            if board[x][y] == '.':
                board[x][y] = chr(ord('A') + idx)
            else:
                board[x][y] = '*'
        for coin_pos, coin_col in self.coins.items():
            board[coin_pos[0]][coin_pos[1]] = chr(ord('a') + coin_col)
        return '\n'.join(''.join(row) for row in board)
```

In [ ]:
```
class CoinsConvNetwork(nn.Module):
    def __init__(self, coins_env, kernel_size=3, stride=2, lin_hid_size=32,
                 pad_value=0.0, drop_coin_col=False):
        super().__init__()
        self.kernel_size = kernel_size
        self.stride = stride
        self.convs = nn.ModuleList()
        self.pad = nn.ConstantPad2d(padding=(kernel_size // 2),
                                    value=pad_value)
        self.drop_coin_col = drop_coin_col
        channels = coins_env.get_num_agents()
        if drop_coin_col:
            channels += 1
        else:
            channels *= 2
        size = coins_env.get_size()
        actions = coins_env.get_num_actions()

        while True:
            cur_stride = stride if self.convs else 1
            new_size = (size - kernel_size % 2) // cur_stride + 1
            if self.convs and size == new_size:
                break
            self.convs.append(nn.Conv2d(in_channels=channels,
                                        out_channels=(2 * channels),
                                        kernel_size=kernel_size,
                                        stride=cur_stride))
```

```
                channels *= 2
                size = new_size

            feats = channels * size**2
            if lin_hid_size is not None:
                self.lin1 = nn.Linear(in_features=feats, out_features=lin_hid_size)
                self.lin2 = nn.Linear(in_features=lin_hid_size, out_features=actions)
            else:
                self.lin1 = None
                self.lin2 = nn.Linear(in_features=feats, out_features=actions)

        def forward(self, x):
            if self.drop_coin_col:
                half = x.size(1) // 2
                x = torch.cat([x[:, :half, ...], x[:, half:, ...].sum(
                    dim=1).unsqueeze(dim=1)], dim=1)
            x = x.float()
            for conv in self.convs:
                x = F.elu(conv(self.pad(x)))
            x = x.view(x.size(0), -1)
            if self.lin1 is not None:
                x = F.elu(self.lin1(x))
            return self.lin2(x)
```

In [ ]:
```
class CoinsGreedyAgent(BaseAgent):
    def __init__(self, coins_env, agent_idx, own_coins_only=False):
        self.num_agents = coins_env.get_num_agents()
        self.size = coins_env.get_size()
        self.agent_idx = agent_idx
        self.own_coins_only = own_coins_only

    def get_policy(self, state, do_exploration=True):
        start = np_locmax(state[self.agent_idx])
        NOT_A_COIN = -1
        coins = np.full([self.size, self.size], NOT_A_COIN)
        for row in range(self.size):
            for col in range(self.size):
                color = np_locmax(state[self.num_agents:, row, col])[0]
                if state[self.num_agents + color, row, col] > 0:
                    coins[row, col] = color
        INFINITY = self.size**2
        # Numpy hack
        filler = np.empty((), dtype=object)
        filler[()] = (INFINITY, INFINITY)
        distance = np.full_like(coins, filler, dtype=np.object)
        distance[start] = (0, 0)
        last_action = np.zeros_like(distance)
        queue = PriorityQueue()
```

```
                queue.put(distance[start] + start)

                while not queue.empty():
                    elem = queue.get()
                    pos = elem[2:]
                    if elem[:2] != distance[pos]:
                        continue
                    coin_length = 0 if coins[pos] == NOT_A_COIN else 1
                    new_distance = tuple(np.array(distance[pos]) +
                                         np.array([coin_length, 1]))
                    for action, vector in enumerate(CoinsEnvironment.DIRECTIONS):
                        new_pos = pos + vector
                        if (new_pos < 0).any() or (new_pos >= self.size).any():
                            continue
                        new_pos = tuple(new_pos)
                        if distance[new_pos] > new_distance:
                            distance[new_pos] = new_distance
                            last_action[new_pos] = action
                            queue.put(new_distance + new_pos)

                min_distance = (INFINITY, INFINITY)
                finish = None
                for row in range(self.size):
                    for col in range(self.size):
                        if coins[row, col] == NOT_A_COIN:
                            continue
                        if self.own_coins_only and coins[row, col] != self.agent_idx:
                            continue
                        if min_distance > distance[row, col]:
                            min_distance = distance[row, col]
                            finish = (row, col)
                if finish is None:
                    finish = (self.size // 2, self.size // 2)

                action = 0
                while finish != start:
                    action = last_action[finish]
                    finish = tuple(np.array(finish) -
                                   CoinsEnvironment.DIRECTIONS[action])
                return OneHotPolicy(action)
```

4 Experiment: 2-player IPD

```
In [ ]: env = IPDEnvironment(num_agents=2, memory_size=1)
        agent_factory = lambda: DQNAgent(num_actions=env.get_num_actions(),
                                         network=IPDNetwork(env), lr=1e-4,
                                         gamma=0.95, eps_init=0.9,
```

```
                                    eps_mult=0.999, eps_min=0.01)
        coop_agents = [agent_factory() for _ in range(env.get_num_agents())]
        runner = Runner(num_runs=50, num_steps=1000, batch_size=4, replay_size=1000,
                        replay_batch_size=4, coop_reward=True,
                        agent_names=['Coop DQN #1', 'Coop DQN #2'],
                        visualize=True, visualize_every=2)
        runner.run(env=env, agents=coop_agents)
```

```
In [ ]: agent_factory = lambda: DQNAgent(num_actions=env.get_num_actions(),
                                    network=IPDNetwork(env), lr=1e-4,
                                    gamma=0.95, eps_init=0.9,
                                    eps_mult=0.999, eps_min=0.01)
        def_agents = [agent_factory() for _ in range(env.get_num_agents())]
        runner = Runner(num_runs=100, num_steps=1000, batch_size=4, replay_size=1000,
                        replay_batch_size=4, coop_reward=False,
                        agent_names=['Def DQN #1', 'Def DQN #2'],
                        visualize=True, visualize_every=2)
        runner.run(env=env, agents=def_agents)
```

```
In [ ]: coop_policies = [get_pure_policy(a, do_exploration=False) for a in coop_agents]
        def_policies = [get_pure_policy(a, do_exploration=False) for a in def_agents]
        agents = [PolicyAgent(coop_policies[0]), PolicyAgent(def_policies[1])]
        runner = Runner(num_runs=5, num_steps=500, agent_names=['Coop', 'Def'],
                        visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)
```

```
In [ ]: agents = [PolicyAgent(coop_policies[0]),
                   PolicyAgent(OneHotPolicy(IPDEnvironment.DEFECT))]
        runner = Runner(num_runs=5, num_steps=500, agent_names=['Coop', 'Custom Def'],
                        visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)
```

```
In [ ]: rollouts_runner = Runner(num_runs=30, num_steps=4)
        agent_factory = lambda idx: AMTFTAgent(env_copy=deepcopy(env),
                                               runner=rollouts_runner,
                                               agent_idx=idx,
                                               coop_policies=coop_policies,
                                               def_policies=def_policies,
                                               def_limit=0.5, init_def_penalty=1.0)
        agents = [agent_factory(idx) for idx in range(env.get_num_agents())]
        runner = Runner(num_runs=5, num_steps=500, agent_names=['amTFT #1', 'amTFT #2'],
                        visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)
```

```
In [ ]: agents = [agent_factory(0), PolicyAgent(def_policies[1])]
        runner = Runner(num_runs=5, num_steps=500, agent_names=['amTFT', 'Def'],
                        visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)
```

```
In [ ]: agents = [agent_factory(0), PolicyAgent(OneHotPolicy(IPDEnvironment.DEFECT))]
        runner = Runner(num_runs=5, num_steps=500, agent_names=['amTFT', 'Custom Def'],
                        visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)

In [ ]: agents = [agent_factory(0), IPDTFTAgent(1)]
        runner = Runner(num_runs=5, num_steps=500, agent_names=['amTFT', 'Custom Coop'],
                        visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)
```

5 Experiment: 3-player IPD

```
In [ ]: env = IPDEnvironment(num_agents=3, memory_size=1)
        agent_factory = lambda: DQNAgent(num_actions=env.get_num_actions(),
                                         network=IPDNetwork(env), lr=1e-4,
                                         gamma=0.95, eps_init=0.9,
                                         eps_mult=0.999, eps_min=0.01)
        coop_agents = [agent_factory() for _ in range(env.get_num_agents())]
        runner = Runner(num_runs=100, num_steps=1000, batch_size=4, replay_size=1000,
                        replay_batch_size=4, coop_reward=True,
                        agent_names=['Coop DQN #1', 'Coop DQN #2', 'Coop DQN #3'],
                        visualize=True, visualize_every=4)
        runner.run(env=env, agents=coop_agents)

In [ ]: agent_factory = lambda: DQNAgent(num_actions=env.get_num_actions(),
                                         network=IPDNetwork(env), lr=1e-4,
                                         gamma=0.95, eps_init=0.9,
                                         eps_mult=0.999, eps_min=0.01)
        def_agents = [agent_factory() for _ in range(env.get_num_agents())]
        runner = Runner(num_runs=100, num_steps=1000, batch_size=4, replay_size=1000,
                        replay_batch_size=4, coop_reward=False,
                        agent_names=['Def DQN #1', 'Def DQN #2', 'Def DQN #3'],
                        visualize=True, visualize_every=4)
        runner.run(env=env, agents=def_agents)

In [ ]: coop_policies = [get_pure_policy(a, do_exploration=False) for a in coop_agents]
        def_policies = [get_pure_policy(a, do_exploration=False) for a in def_agents]
        agents = [PolicyAgent(coop_policies[0]), PolicyAgent(coop_policies[1]),
                  PolicyAgent(def_policies[2])]
        runner = Runner(num_runs=5, num_steps=500,
                        agent_names=['Coop #1', 'Coop #2', 'Def'],
                        visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)

In [ ]: agents = [PolicyAgent(coop_policies[0]), PolicyAgent(coop_policies[1]),
                  PolicyAgent(OneHotPolicy(IPDEnvironment.DEFECT))]
        runner = Runner(num_runs=5, num_steps=500,
                        agent_names=['Coop #1', 'Coop #2', 'Custom Def'],
```

```
                    visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)
```

```
In [ ]: rollouts_runner = Runner(num_runs=30, num_steps=4)
        agent_factory = lambda idx: AMTFTAgent(env_copy=deepcopy(env),
                                               runner=rollouts_runner, agent_idx=idx,
                                               coop_policies=coop_policies,
                                               def_policies=def_policies,
                                               def_limit=0.5, init_def_penalty=1.0)
        agents = [agent_factory(idx) for idx in range(env.get_num_agents())]
        runner = Runner(num_runs=5, num_steps=500,
                        agent_names=['amTFT #1', 'amTFT #2', 'amTFT #3'],
                        visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)
```

```
In [ ]: agents = [agent_factory(0), agent_factory(1), PolicyAgent(def_policies[2])]
        runner = Runner(num_runs=5, num_steps=500,
                        agent_names=['amTFT #1', 'amTFT #2', 'Def'],
                        visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)
```

```
In [ ]: agents = [agent_factory(0), agent_factory(1),
                   PolicyAgent(OneHotPolicy(IPDEnvironment.DEFECT))]
        runner = Runner(num_runs=5, num_steps=500,
                        agent_names=['amTFT #1', 'amTFT #2', 'Custom Def'],
                        visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)
```

```
In [ ]: agents = [agent_factory(0), agent_factory(1), IPDTFTAgent(2)]
        runner = Runner(num_runs=5, num_steps=500,
                        agent_names=['amTFT #1', 'amTFT #2', 'Custom Coop'],
                        visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)
```

6 Experiment: 2-player Coins

```
In [ ]: env = CoinsEnvironment(num_agents=2, size=5)
        agent_factory = lambda: DQNAgent(num_actions=env.get_num_actions(),
                                         network=CoinsConvNetwork(env).to(device),
                                         lr=1e-4, gamma=0.95, eps_init=0.9,
                                         eps_mult=0.999999, eps_min=0.05)
        coop_agents = [agent_factory() for _ in range(env.get_num_agents())]
        runner = Runner(num_runs=250, num_steps=2*10**4, batch_size=16,
                        replay_size=10**4, replay_batch_size=16, coop_reward=True,
                        agent_names=['Coop DQN #1', 'Coop DQN #2'],
                        visualize=True, visualize_every=4)
        runner.run(env=env, agents=coop_agents)
```

```
In [ ]: agent_factory = lambda: DQNAgent(num_actions=env.get_num_actions(),
                                        network=CoinsConvNetwork(env).to(device),
                                        lr=1e-4, gamma=0.95, eps_init=0.9,
                                        eps_mult=0.999999, eps_min=0.05)
        def_agents = [agent_factory() for _ in range(env.get_num_agents())]
        runner = Runner(num_runs=300, num_steps=2*10**4, batch_size=16,
                        replay_size=10**4, replay_batch_size=16, coop_reward=False,
                        agent_names=['Def DQN #1', 'Def DQN #2'],
                        visualize=True, visualize_every=4)
        runner.run(env=env, agents=def_agents)

In [ ]: coop_policies = [get_pure_policy(a, do_exploration=False)
                         for a in coop_agents]
        def_policies = [get_pure_policy(a, do_exploration=False)
                        for a in def_agents]
        agents = [PolicyAgent(coop_policies[0]), PolicyAgent(def_policies[1])]
        runner = Runner(num_runs=5, num_steps=1000,
                        agent_names=['Coop', 'Def'],
                        visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)

In [ ]: agents = [PolicyAgent(coop_policies[0]),
                  CoinsGreedyAgent(coins_env=env, agent_idx=1, own_coins_only=False)]
        runner = Runner(num_runs=5, num_steps=1000,
                        agent_names=['Coop', 'Custom Def'],
                        visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)

In [ ]: rollouts_runner = Runner(num_runs=30, num_steps=50)
        agent_factory = lambda idx: AMTFTAgent(env_copy=deepcopy(env),
                                               runner=rollouts_runner, agent_idx=idx,
                                               coop_policies=coop_policies,
                                               def_policies=def_policies,
                                               def_limit=1.0, init_def_penalty=3.0)
        agents = [agent_factory(idx) for idx in range(env.get_num_agents())]
        runner = Runner(num_runs=5, num_steps=1000,
                        agent_names=['amTFT #1', 'amTFT #2'],
                        visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)

In [ ]: agents = [agent_factory(0), PolicyAgent(def_policies[1])]
        runner = Runner(num_runs=5, num_steps=1000,
                        agent_names=['amTFT', 'Def'],
                        visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)

In [ ]: agents = [agent_factory(0),
                  CoinsGreedyAgent(coins_env=env, agent_idx=1, own_coins_only=False)]
        runner = Runner(num_runs=5, num_steps=1000,
```

```
                        agent_names=['amTFT', 'Custom Def'],
                        visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)
```

```
In [ ]: agents = [agent_factory(0),
                  CoinsGreedyAgent(coins_env=env, agent_idx=1, own_coins_only=True)]
        runner = Runner(num_runs=5, num_steps=1000,
                        agent_names=['amTFT', 'Custom Coop'],
                        visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)
```

7 Experiment: 3-player Coins

```
In [ ]: env = CoinsEnvironment(num_agents=3, size=5)
        agent_factory = lambda: DQNAgent(num_actions=env.get_num_actions(),
                                         network=CoinsConvNetwork(env).to(device),
                                         lr=1e-4, gamma=0.95, eps_init=0.9,
                                         eps_mult=0.999995, eps_min=0.05)
        coop_agents = [agent_factory() for _ in range(env.get_num_agents())]
        runner = Runner(num_runs=1000, num_steps=5000, batch_size=32,
                        replay_size=5000, coop_reward=True,
                        agent_names=['Coop DQN #1', 'Coop DQN #2', 'Coop DQN #3'],
                        visualize=True, visualize_every=50)
        runner.run(env=env, agents=coop_agents)
```

```
In [ ]: runner.run(env=env, agents=coop_agents)
```

```
In [ ]: agent_factory = lambda: DQNAgent(num_actions=env.get_num_actions(),
                                         network=CoinsConvNetwork(
                                             env, drop_coin_col=True).to(device),
                                         lr=1e-4, gamma=0.95, eps_init=0.9,
                                         eps_mult=0.999995, eps_min=0.05)
        def_agents = [agent_factory() for _ in range(env.get_num_agents())]
        runner = Runner(num_runs=2000, num_steps=2000, batch_size=32,
                        replay_size=0, coop_reward=False,
                        agent_names=['Def DQN #1', 'Def DQN #2', 'Def DQN #3'],
                        visualize=True, visualize_every=50)
        runner.run(env=env, agents=def_agents)
```

```
In [ ]: coop_policies = [get_pure_policy(a, do_exploration=False)
                          for a in coop_agents]
        def_policies = [get_pure_policy(a, do_exploration=False)
                        for a in def_agents]
        agents = [PolicyAgent(coop_policies[0]), PolicyAgent(coop_policies[1]),
                  PolicyAgent(def_policies[2])]
        runner = Runner(num_runs=5, num_steps=1000,
                        agent_names=['Coop #1', 'Coop #2', 'Def'],
                        visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)
```

```
In [ ]: agents = [PolicyAgent(coop_policies[0]), PolicyAgent(coop_policies[1]),
                 CoinsGreedyAgent(coins_env=env, agent_idx=2, own_coins_only=False)]
        runner = Runner(num_runs=5, num_steps=1000,
                        agent_names=['Coop #1', 'Coop #2', 'Custom Def'],
                        visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)

In [ ]: rollouts_runner = Runner(num_runs=20, num_steps=50)
        agent_factory = lambda idx: AMTFTAgent(env_copy=deepcopy(env),
                                               runner=rollouts_runner, agent_idx=idx,
                                               coop_policies=coop_policies,
                                               def_policies=def_policies,
                                               def_limit=1.0, init_def_penalty=3.0)
        agents = [agent_factory(idx) for idx in range(env.get_num_agents())]
        runner = Runner(num_runs=5, num_steps=1000,
                        agent_names=['amTFT #1', 'amTFT #2', 'amTFT #3'],
                        visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)

In [ ]: agents = [agent_factory(0), agent_factory(1), PolicyAgent(def_policies[2])]
        runner = Runner(num_runs=5, num_steps=1000,
                        agent_names=['amTFT #1', 'amTFT #2', 'Def'],
                        visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)

In [ ]: agents = [agent_factory(0), agent_factory(1),
                 CoinsGreedyAgent(coins_env=env, agent_idx=2, own_coins_only=False)]
        runner = Runner(num_runs=5, num_steps=1000,
                        agent_names=['amTFT #1', 'amTFT #2', 'Custom Def'],
                        visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)

In [ ]: agents = [agent_factory(0), agent_factory(1),
                 CoinsGreedyAgent(coins_env=env, agent_idx=2, own_coins_only=True)]
        runner = Runner(num_runs=5, num_steps=1000,
                        agent_names=['amTFT #1', 'amTFT #2', 'Custom Coop'],
                        visualize=True, vis_summary_only=True)
        runner.run(env=env, agents=agents)
```