Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий

Работа допущена к защите
Руководитель ОП
_____ В.Г. Пак
«_____» _____ 2020 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ ИССЛЕДОВАНИЕ ПРИМЕНЕНИЯ ТЕХНОЛОГИИ ОБУЧЕНИЯ С ПОДКРЕПЛЕНИЕМ В УПРАВЛЕНИИ МУЛЬТИАГЕНТНОЙ СИСТЕМОЙ В ИГРОВОЙ СРЕДЕ

по направлению подготовки 02.04.03 Математическое обеспечение и администрирование информационных систем
Направленность (профиль) 02.04.03_02 Проектирование и разработка информационных систем

Выполнил
студент гр. в3540203/80277 О.Ю. Григорьев

Руководитель
доцент ВШИСиСТ,
к.ф.-м.н. В.Г. Пак

Санкт-Петербург
2020

САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

Институт компьютерных наук и технологий

УТВЕРЖДАЮ
Руководитель ОП
_____ В.Г. Пак
«_____» _____ 2020г.

ЗАДАНИЕ
на выполнение выпускной квалификационной работы

студенту Григорьеву Олегу Юрьевичу гр. в3540203/80277

1. Тема работы: Исследование применения технологии обучения с подкреплением в управлении мультиагентной системой в игровой среде.
2. Срок сдачи студентом законченной работы
3. Исходные данные по работе
   3.1. Фреймворки для разработки алгоритмов обучения с подкреплением (pytorch и другие).
   3.2. Программные среды тестирования алгоритмов (OpenAI Gym и другие).
   3.3. Алгоритмы обучения с подкреплением.
   3.4. Теория управления MAC.
4. Содержание работы (перечень подлежащих разработке вопросов):
   4.1. Обзор технологии обучения с подкреплением.
   4.2. Обзор подходов к управлению мультиагентными системами (MAC).
   4.3. Обзор аналогичных исследований.
   4.4. Постановка задачи выпускной квалификационной работы.
   4.5. Разработка алгоритмов управления агентами в MAC. Их исследование.
   4.6. Внедрение разработанных алгоритмов в компьютерную игру, тестирование.
   4.7. Экспериментальное исследование алгоритмов.
   4.8. Заключение, выводы об эффективности исследованных алгоритмов.
5. Перечень графического материала (с указанием обязательных чертежей):
   5.1. Графики результатов экспериментов.

5.2. Блок-схемы алгоритмов.

5.3. Скриншоты

6. Консультанты по работе

6.1. Старший преподаватель ВШИСиСТ, Ю.Д. Заковряшин (нормоконтроль).

7. Дата выдачи задания

Руководитель ВКР _____В.Г. Пак

Задание принял к исполнению 03.02.2020

Студент _____О.Ю. Григорьев

РЕФЕРАТ

На 70 с., 32 рисунка, 6 таблиц, 3 приложения. Тема выпускной квалификационной работы: «Исследование применения технологии обучения с подкреплением в управлении мультиагентной системой в игровой среде»

В данной работе исследована технология обучения с подкреплением, её применимость к мультиагентным системам, а именно применимость алгоритма мультиагентной глубокой детерминированной политики градиента (MADDPG). Изучено общение агентов между собой. Были предложены, реализованы и протестированы различные способы оптимизации обучения. Эксперименты производились в различных сценариях в двухмерной среде multiagent-particle-envs [13] от компании OpenAI.

ABSTRACT

70 p., 32 figures, 6 tables, 3 appendices.

KEYWORDS: MACHINE LEARNING, REINFORCEMENT LEARNING, DEEP LEARNING, MULTI-AGENT SYSTEMS.

The subject of the graduate qualification work is «Investigation of the Application of Reinforcement Learning to Managing Multi-Agent systems in Gaming Environment».

In this thesis Reinforcement Learning and its applicability to Multi-Agent systems were examined. Specifically, Multi-Agent Deep Deterministic Policy Gradient was reviewed. Communication between agents was researched. Different optimizing methods were suggested, implemented, and tested. Experiments were set up on different scenarios in 2D-environment multiagent-particle-envs [13] by OpenAI.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В этой работе исследуется совместная работа нескольких агентов с использованием алгоритмов и методов глубокого обучения в 2D игровых средах. В данной работе это имеется в виду, когда изучаются и адаптируются современные алгоритмы и методы глубокого обучения с подкреплением к настройкам игры с несколькими агентами.

Глубокое обучение с подкреплением — это новая область исследований алгоритмов и методов, которая сочетает в себе обучение с подкреплением и глубокое обучение. Предыдущие работы в основном были направлены на адаптацию глубоких нейронных сетей для усиления алгоритмов обучения. Например, глубокая Q-сеть (Deep Q-network, DQN) [18] интегрирует глубокие нейронные сети в Q-обучение — классический табличный алгоритм обучения с подкреплением. Обученные сети могут играть в различные игры Atari 2600 [37] лучше человека. Это считается первой успешной попыткой обучить машину играть в видеоигры с прямым визуальным вводом большого размера. Алгоритм глубокого детерминированного градиента политики (deep deterministic policy gradient, DDPG) [6] — это ещё один пример использования глубоких нейронных сетей в контексте обучения с подкреплением и непрерывного пространства действий.

Большинство исследований посвящено обучению одного агента. Однако проблемы, связанные с мультиагентным сотрудничеством или конкуренцией, также очень распространены в социальной, экономической и инженерной областях.

В мире очень много задач, которые лучше всего решаются командой, несколькими агентами (см. Мультиагентные алгоритмы).

Было решено изучить мультиагентную составляющую обучения с подкреплением. А именно — общение. Без общения агентов между собой невозможно достичь хороших результатов. Как в любой командной игре, действия агентов также должны быть скоординированы.

При увеличении количества агентов, а также сложности среды, возрастает сложность обучения агентов, поэтому в данной работе также были изучены различные варианты оптимизации этого процесса.

Игры представляют собой упрощённые версии реальных задач, которые можно сделать идеальными тестовыми площадками для экспериментов. Поэтому в этой работе для изучения алгоритмов глубокого обучения с подкреплением для совместной работы нескольких агентов используются игровые сценарии.

Исходя из вышеизложенного, была поставлена следующая цель — разработка и исследование алгоритмов управления мультиагентными системами (МАС) с применением технологий глубокого обучения с подкреплением. Для достижения этой цели были поставлены следующие задачи:

A. Обзор технологии глубокого обучения с подкреплением и её применения в задачах управления;
B. Обзор методов, алгоритмов управления МАС, их реализаций, практического применения;
C. Разработка сценариев работы МАС, алгоритмов управления для них;
D. Реализация сценариев и алгоритмов в игровой среде, экспериментальное исследование алгоритмов, обучения и поведения агентов под влиянием алгоритмов;
E. Выводы об эффективности алгоритмов в исследованных сценариях, их применимости в прикладных задачах.

Объект исследования — это технология обучения с подкреплением.

Предмет исследования — это применимость обучения с подкреплением к мультиагентным системам.

В данной работе использовались следующие технологии и инструменты:

– IDE — PyCharm Community;
– язык программирования — Python;
– фреймворк — Tensorflow;
– среда для отработки мультиагентных алгоритмов multiagent-particle-envs [13] от компании OpenAI;
– базовая реализация алгоритма MADDPG [25] от компании OpenAI.

В этой работе сперва описываются вопросы исследования, а также сценарии, с которыми выполняются эксперименты.

Затем рассматриваются теория технологии обучения с подкреплением в целом и мультиагентных систем в частности. Описываются применяемые методы.

Далее описываются эксперименты, их результаты, и делаются выводы.

ГЛАВА 1. ВВОДНАЯ ГЛАВА

1.1. Вопросы исследования

В этой работе исследуются следующие вопросы:

1. Как несколько агентов могут научиться сотрудничать друг с другом во время обучения в определённых игровых сценариях?
2. Может ли после обучения появиться язык между агентами в определённых игровых сценариях?
3. Как можно оптимизировать и ускорить процесс обучения?

Сначала будут изучены современные алгоритмы и методы глубокого обучения с подкреплением. Затем будут проведены эксперименты игровых сценариев в модифицированной среде от компании OpenAI [27]. Наконец, будут рассмотрены и применены различные приёмы для оптимизации процесса обучения.

1.2. Сценарии

Чтобы исследовать вышеизложенные вопросы, были использованы некоторые из множества доступных сценариев, где агенты общаются друг с другом и физически перемещаются к определённым целям. Более подробно об этом написано в разделе Сетевая архитектура.

Некоторые из сценариев являются точными копиями экспериментов, проведённых в недавней работе [25], а другие представляют собой новые сценарии, которые расширяют исходные с целью дальнейшего изучения многоагентного сотрудничества и коммуникации.

1.2.1. Сценарий 1. Simple Speaker Listener

В этом сценарии есть три ориентира, представленные как красные, зелёные и синие ориентиры, как показано на рис.1.1. Два агента с разными функциями должны сотрудничать для достижения общей цели. Говорун (серый агент) не может двигаться, но видит цвет цели и может говорить с другим агентом. Слушатель (отображаемый таким же цветом, что и его цель) видит все ориентиры и их цвета (но не видит собственный цвет, т.е. не знает, какой из объектов является его целью), а также слышит говоруна и пытается перейти к правильному ориентиру. Более

подробные настройки среды этого сценария можно найти в разделе Эксперименты: Сценарий 1. Simple Speaker Listener.

Рис.1.1. Сценарий 1. Simple Speaker Listener. Говорун даёт команду, представляющую «зелёный», а слушатель слушает сообщение говоруна и направляется к цели

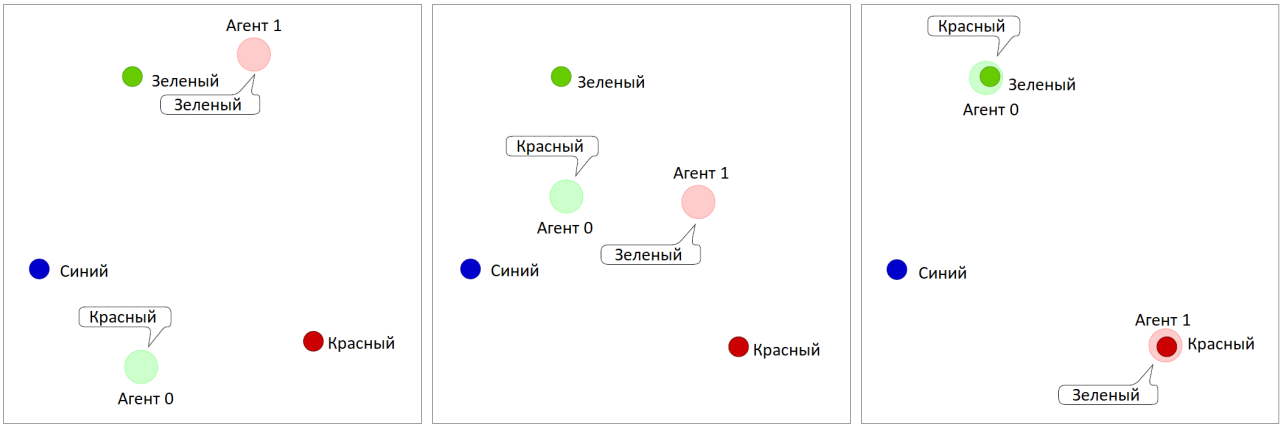

Рис.1.2. Сценарий 2: Simple Reference. Агент 0 (зелёный) выдаёт команду «красный», и слушает агента 1. А агент 1 (красный) слушает и командует «зелёный» для Агента 0

1.2.2. Сценарий 2. Simple Reference

Этот сценарий расширяет предыдущий, поскольку оба агента являются одновременно и говорунами и слушателями. Ориентиры остаются прежними, отображаются как три разноцветных объекта. В этом эпизоде агент 0 (отображаемый в том же цвете, что и его целевой ориентир) выдаёт коммуникационное действие, представляющее «красный», и слушает агента 1. А агент 1 (отображается в том же цвете, что и его целевой ориентир) слушает и издаёт «зелёный» для Агента 0. Скриншоты на рис.1.2 слева направо показывают, как ведут себя два агента в соответствии с оптимальными политиками, слушая друг друга и перемещаясь к целям.

Каждый из двух агентов пытается достичь своего целевого ориентира, который известен только другому агенту. Таким образом, он должен научиться сообщать другому агенту его цель и перемещаться к своей собственной. Что отличает сценарий от двух копий сценария Simple Speaker Listener, так это то, что единое общее вознаграждение, присуждаемое агентам, основано на общей производительности. Таким образом, агенты должны выяснить, что идёт хорошо, а что нет. Настройка среды подробно описана в разделе Эксперименты: Сценарий 2. Simple Reference.

1.2.3. Сценарий 3. Simple World Communication

В этом сценарии есть хорошие агенты (зелёные) и их противники – преследователи (красные). Также есть препятствия (чёрные), леса (зелёные) и еда (зелёные). Один из преследователей – лидер (тёмно-красный).

Хорошие агенты награждаются за то, что приближаются к еде и наказываются, когда касаются преследователей.

Преследователи награждаются за касание хороших агентов.

Лес скрывает агентов от преследователей. Лидер видит агентов даже в лесу (см. рис.1.3).

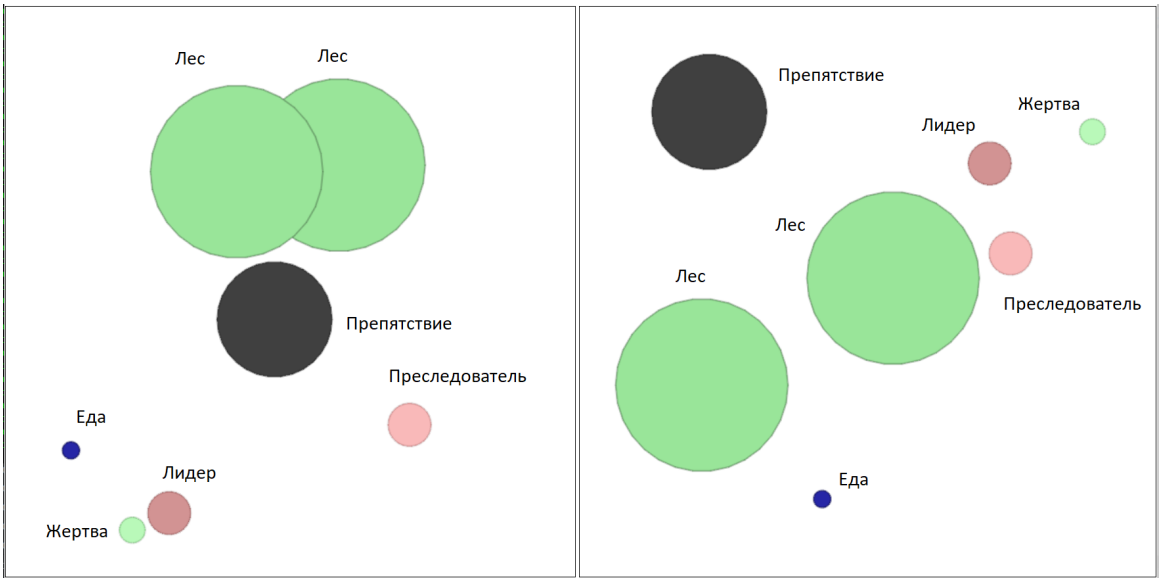

Рис.1.3. Сценарий Simple World Communication, жертва избегает преследователей, и, по возможности, приближается к еде, преследователи догоняют жертву

Более подробное описание и настройки можно найти в разделе Эксперименты: Сценарий 3. Simple World Communication.

1.2.4. Сценарий 4: Simple Tag

В этом сценарии есть хорошие агенты (зелёные) и их противники – преследователи (красные). Также есть препятствие (чёрное).

Хорошие агенты наказываются, в случае касания преследователей.

Преследователи награждаются за касание хороших агентов.

Преследователей больше, но у жертвы больше скорость. Теоретически, преследователи должны выработать политику, при которой они окружают жертву или другие подобные приёмы. Жертва должна не позволять загнать себя в угол.

Также данный сценарий был немного модифицирован — была добавлена граница по периметру поля, которая не даёт агентам покинуть видимую часть.

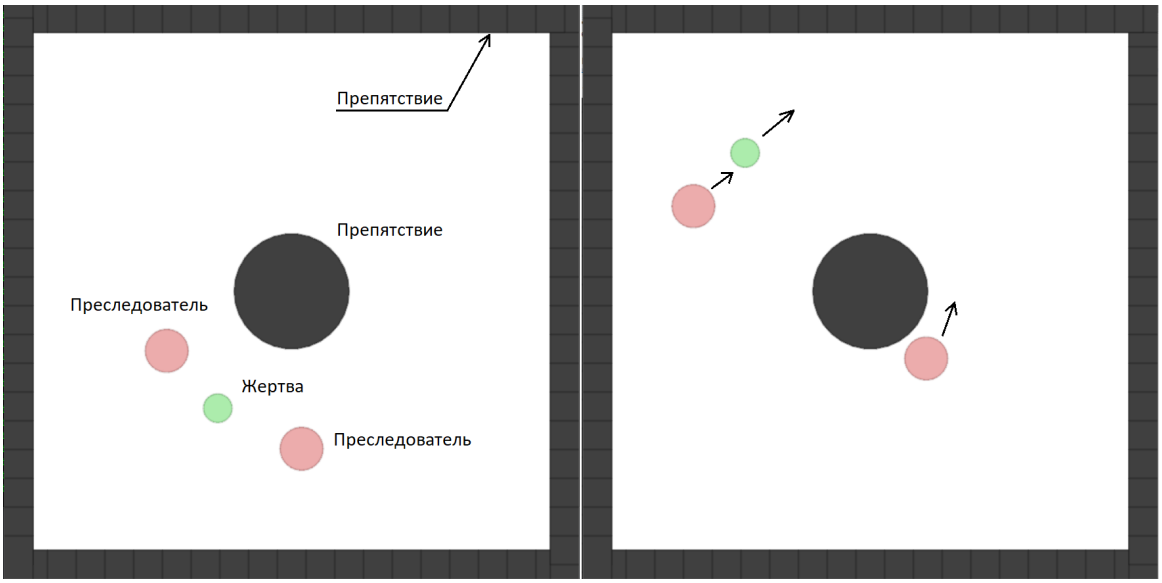

Рис.1.4. Сценарий Simple Tag. Жертва избегает преследователей. На правом скриншоте правый преследователь превентивно обходит препятствие справа и движется в ту же область, что и жертва

Более подробное описание и настройки можно найти в разделе Эксперименты: Сценарий 4. Simple Tag.

1.3. Практическая значимость

Проект сосредоточен на обучении нескольких агентов совместной работе с использованием глубокого обучения в игровой среде. Результаты исследования могут быть дополнительно разработаны и широко использованы во многих практических реальных приложениях в области экономики, управления, техники и т.д.

Например, он может применяться в робототехнике, к автономным транспортным средствам, производственным линиям, фондовым рынкам и т.д.

Перед применением в этих областях алгоритмы и методы, которые исследуются и разрабатываются в этой работе, должны быть хорошо протестированы и проверены, поскольку эти приложения непосредственно влияют на безопасность человека, социальную и финансовую безопасность. В долгосрочной перспективе применение мультиагентных алгоритмов приведёт к появлению всё большего числа автономных систем во многих областях.

Прежде чем приступить к исследовани, необходимо произвести обзор обучения с подкреплением.

ГЛАВА 2. ОБЗОР ТЕХНОЛОГИИ ОБУЧЕНИЯ С ПОДКРЕПЛЕНИЕМ

Алгоритмы и методы глубокого обучения с подкреплением (Deep Reinforcement Learning, DRL) – это методы, которые используются для исследования вопросов данной работы. DRL – это подраздел машинного обучения, который лежит на пересечении обучения с подкреплением (Reinforcement Learning) и глубокого обучения (Deep Learning).

Прежде чем говорить о DRL, нужно кратко рассмотреть понятия, связанные с искусственным интеллектом и машинным обучением.

На рис.2.1 представлен обзор концепций, о которых идёт речь в этой главе.

Сначала будут рассмотрены искусственный интеллект, машинное обучение и его три категории. Затем будет описано глубокое обучение. Наконец, будут рассмотрены основные алгоритмы глубокого обучения с подкреплением.

Таким образом можно будет понять, почему и как глубокое обучение и обучение с подкреплением объединяются в глубокое обучение с подкреплением, которое позволяет агентам взаимодействовать с более сложной средой и вести себя более разумно.

Рис.2.1. Диаграмма, показывающая концепты машинного обучения и искуственного интеллекта и их отношения

2.1. Искусственный интеллект

Искусственный интеллект [8] (ИИ), как следует из названия, является противопоставлением естественному интеллекту человека и животных в силу своего

искусственного происхождения. Именно такой тип интеллекта люди реализуют в машинах.

Конечной целью ИИ является создание таких автономных систем, которые способны учиться методом многочисленных проб и ошибок, чтобы найти оптимальное поведение для достижения максимально возможных целей в сложившемся окружении. [29]

С 21 века, благодаря нескольким прорывам, ИИ доминирует в викторинах и настольных играх, превосходя уровень игры людей [38] [1]. Наряду с увеличением вычислительной мощности, улучшением алгоритмов и доступностью больших наборов данных, ИИ развивался революционными темпами. В ближайшем будущем ИИ ещё сильнее повлияет на работу и повседневную жизнь человека.

2.2. Машинное обучение

Если ИИ – это более широкая концепция машинного интеллекта для решения задач, которые пока решают люди, то машинное обучение (МО) — это основной метод разработки ИИ, который не требует явного программирования [30]. МО позволяет компьютерам строить модели и применять алгоритмы при изучении больших объёмов данных. Модели МО обучаются с использованием методов статистики, чтобы понять структуру набора данных или последовательности экспериментов. Обученные модели могут распознавать паттерны и делать очень точные прогнозы, учитывая не очевидные данные, или обрабатывать определённые задачи в неочевидных сценариях [5].

Основные категории алгоритмов машинного обучения показаны на рисунке рис.2.2. А именно, обучение с учителем, обучение без учителя и обучение с подкреплением. Категории определяются тем, как алгоритмы и модели наполняются данными и как данные анализируются.

2.2.1. Обучение с учителем (Supervised Learning)

Обучение с учителем — один из способов машинного обучения, в ходе которого испытуемая система принудительно обучается с помощью примеров «стимул-реакция». С точки зрения кибернетики оно является одним из видов кибернетического эксперимента. Между входами и эталонными выходами (стимул–реакция) может существовать некоторая зависимость, но она неизвестна. Известна

Рис.2.2. Категории машинного обучения и соответстующие алгоритмы. Основано на [31]

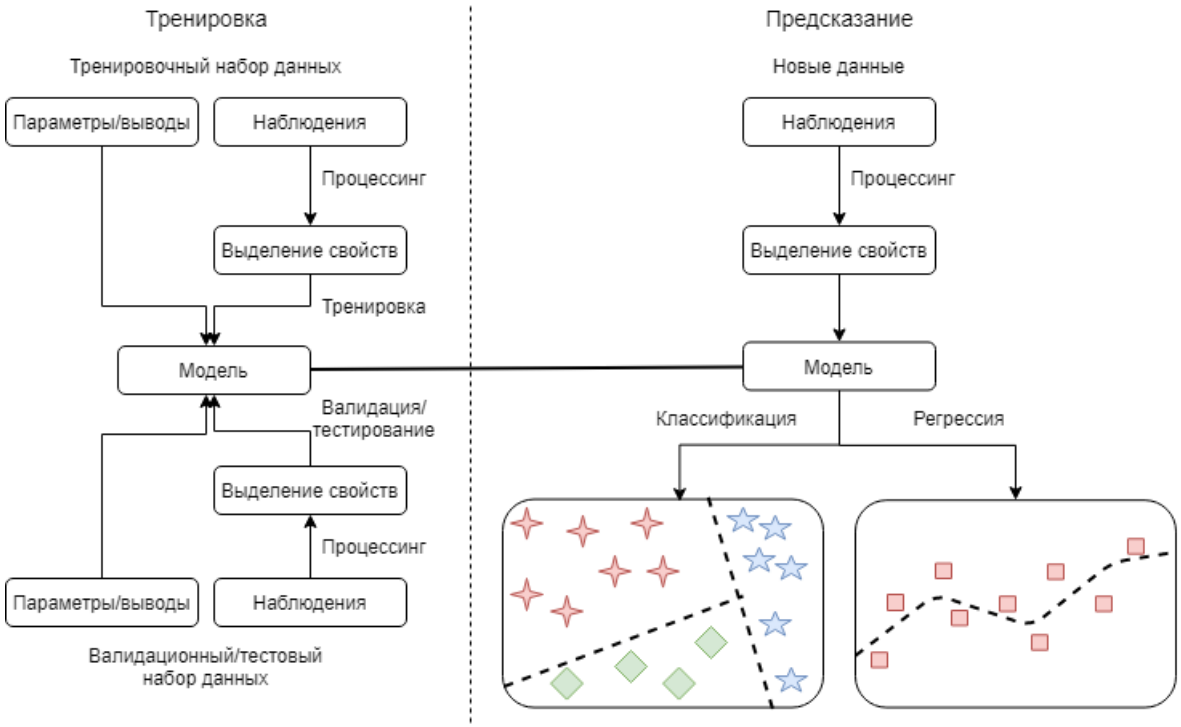

Рис.2.3. Функциональная схема обучения с учителем. Модель тренируется, валидируется и тестируется с использованием набора данных. После того как модель натренирована, она используется для предсказания по новым данным. Типичные задачи, решаемые обучением с учителем: классификация и регрессия

только конечная совокупность прецедентов — пар «стимул-реакция», называемая обучающей выборкой. На основе этих данных требуется восстановить зависимость (построить модель отношений стимул-реакция, пригодных для прогнозирования), то есть построить алгоритм, способный для любого объекта выдать достаточно точный ответ [2].

Обучение с учителем можно разделить на регрессию и классификацию в зависимости от того, являются ли выходные переменные количественными

или качественный. Количественные переменные принимают числовые значения, в то время как качественные переменные принимают значения в одном из K различных классов или категории [2]. Например, прогноз цены на жильё по данным параметрам, таким как местоположение дома, общая площадь и количество комнат и т.д. — это проблема регрессии. Диагноз рака — проблема классификации, поскольку выходной сигнал либо положительный, либо отрицательный.

В обучении с учителем набор данных делится на обучающий, набор для валидации и тестовый набор. Обучающий набор представляет собой пары ввода и вывода переменных, которые напрямую вводятся в модель для обучения. Валидационный набор используется для контроля за переобучением модели. Наконец, тестовый набор используется для подтверждения того, что обученная модель обобщена и точна. Функциональная схема обучения с учителем показана на рис.2.3.

Обучение с учителем — наиболее распространённая категория в машинном обучении, но оно требует больших наборов данных с правильными «ответами». Его применение может быть очень дорого, а в некоторых случаях не практично.

2.2.2. Обучение без учителя (Unsupervised Learning)

Это ещё одна важная категория в машинном обучении. В отличие от обучения с учителем алгоритмы этой категории используют наборы данных, не размеченные «ответами». Задача в этой категории — обнаружить скрытую структуру в данных и распределить их по группам.

Эта категория алгоритмов получила своё название из-за отсутствия меток или выходных переменных. Кластеризация является типичным инструментом, который используется для того, чтобы понять связь между наблюдениями и распределить их в разные группы [15].

При обучении без учителя данные не делятся на обучающие, проверочные и тестовые. Набор данных подаётся в модель напрямую и распределяются по отдельныем группам, как показано на рис.2.4.

2.2.3. Обучение с подкреплением (Reinforcement Learning)

Последняя категория машинного обучения является междисциплинарной областью, которая сочетает в себе машинное обучение, неврологию, поведенческую психология, теорию управления и т.д. Цель обучения с подкреплением заключается

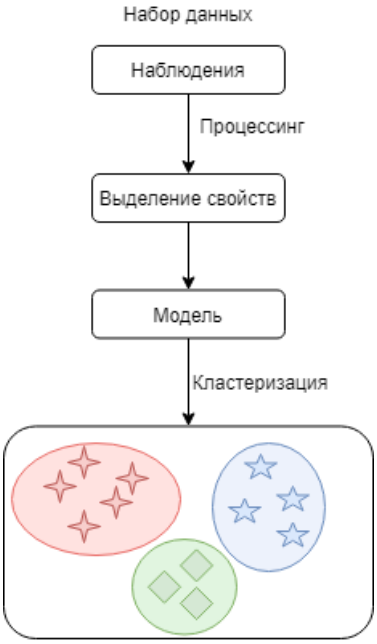

Рис.2.4. Функциональная схема обучения без учителя. Обрабатывается набор данных без соответствующих меток и ответов, из него извлекаются свойства и подаются на вход модели. Они распределяются по различным группам в соответствии со сходством скрытых свойств

в достижении целей без чётких инструкций, но с наградами или штрафами, получаемыми от взаимодействий с окружающей средой.

Агент в обучении с подкреплением изучает оптимальную политику, последовательность действий, которые максимизируют общую будущую награду (reward) [32].

В обучении с подкреплением агент наблюдает состояние s_t на этапе времени t, затем он взаимодействует с окружающей средой, выполняя действие a_t. Среда переходит в следующее состояние s_{t+1}, учитывая текущее состояние и выбранное действие, которое ведёт к получению агентом вознаграждения r_t. Целью агента является определение политики π, которая сопоставляет состояния с действиями, так что последовательность действий, выбранная агентом, максимизирует ожидаемое будущее вознаграждение. На каждом шаге взаимодействия со средой агент генерирует переход $\{s_t, a_t, s_{t+1}, r_t\}$, который даёт информацию, необходимую для улучшения политики, см. рис.2.5.

2.3. Глубокое обучение

Глубокое обучение (Deep Learning, DL) — подраздел машинного обучения, с применением которого в настоящее время достигаются выдающиеся результаты во многих областях, в том числе в распознавании изображений, компьютерном

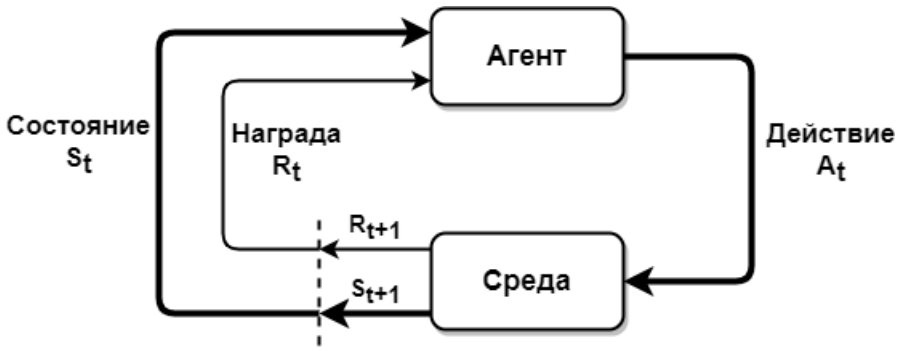

Рис.2.5. Агент взаимодействует с окружающей средой, сначала наблюдая состояние, затем совершая действие и, наконец, получая вознаграждения за выбранное действия. Через многочисленные попытки и ошибки агент учится формулировать оптимальную политику [33]

зрении, распознавании речи и обработке естественных языков. Глубокое обучение как бы имитирует биологический мозг, обрабатывает информацию с помощью искусственных нейронных сетей.

Идея симуляции работы нейронов мозга человека зародилась десятилетия назад. Тем не менее, прорыв произошёл в последние годы, когда стали доступны большие наборы данных и вычислительные мощности [12]. Теперь можно строить модели с большим количеством слоёв искусственных нейронов, чем когда-либо прежде. Сети достигают исключительной производительности в области распознавания изображений и речи при значительном увеличении глубины. Считается, что глубокое обучение является одним из наиболее перспективных подходов для решения текущих задач ИИ.

2.3.1. Искусственная нейронная сеть

Мозг человека и животных — чрезвычайно сложный орган, который до сих пор до конца не изучен. Тем не менее, некоторые аспекты его структуры и функции были расшифрованы. Фундаментальным рабочим элементом в мозге является нейрон. Многочисленные нейроны связаны между собой сложным образом, что даёт возможность запоминания, мышления и принятия решений.

Хотя нейроны сложны и функционируют разными способами, все они имеют некоторые базовые компоненты такие как: ядро, дендриты, аксон и синапсы. Дендриты действуют как входные каналы, через которые нейроны получают информацию от синапсов других нейронов. Затем ядро обрабатывает эти сигналы и превращает в вывод, который затем отправляется в другие нейроны. Связь этих компонентов играет роль линии передачи в нейронных сетях.

Хоть ИНС и не так сложны, как человеческий мозг, они имитируют базовую структуру, которая включает входные, выходные слои, а также, обычно, скрытые слои. В каждом слое есть искусственные нейроны. Нейроны в одном слое обычно связаны с каждым нейроном в следующем слое. Они передают числовые сигналы через соединения с другими нейронами, примерно, как это происходит в биологической нейронной сети [17].

Поскольку выходные сигналы нейронов ИНС представляются в виде вещественных чисел, выход обычно сравнивается с порогом. Только в том случае, если порог превышен, нейрон передаёт сигнал следующим подключённым нейронам.

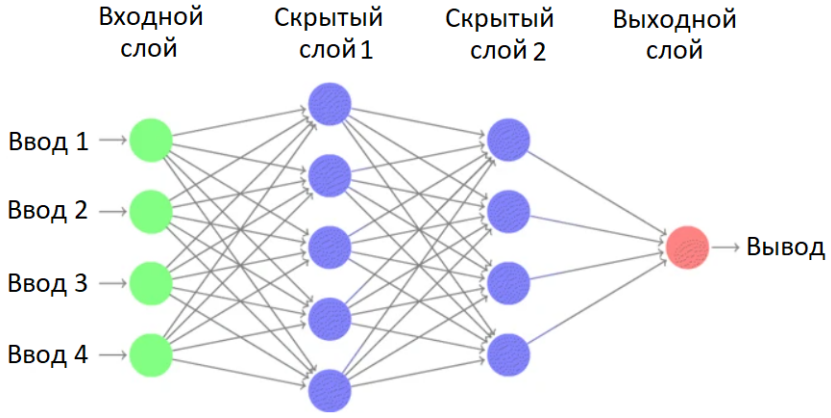

Рис.2.6. Базовая структура ИНС. ИНС обычно состоит из входного и выходного слоёв, а также пары скрытых слоёв. Основано на [21] [23]

Связи между нейронами параметризованы весами, которые обновляются во время обучения, чтобы отрегулировать мощность сигналов, проходящих через сеть [21]; рис.2.6 показывает базовую структуру ИНС.

2.3.2. Глубокая нейронная сеть

Как уже упоминалось выше, концепция нейронной сети не нова.

На ранних этапах из-за ограничений вычислительной мощности, нейронные сети имели очень малую глубину. Обычно они содержали только входной, выходной слои и пару скрытых слоёв. Кроме того, количество нейронов в каждом слое также было ограничено. Глубокие нейронные сети не находили практического применения до последних лет, когда стали доступны огромные вычислительные мощности и большие объёмы данных.

Глубокие нейронные сети с несколькими слоями хороши для извлечения скрытых свойств [20]. Каждый слой решает свою собственную задачу. Узлы в каждом слое учатся на конкретных наборах свойств, которые приходят из

предыдущего слоя. Теоретически, чем глубже нейронная сеть, тем более сложны и абстрактны особенности, которые она может распознать, поскольку каждый нейрон агрегирует выводы из нейронов предыдущего слоя [4]. Например, в задаче распознавания изображений вход представляет собой матрицу пикселей. Первый слой извлекает начальные объекты, такие как рёбра из пикселей, затем следующий слой кодирует расположение краёв и следующий слой распознаёт глаза, рот, уши, ноги, крылья, хвост и т.д. Наконец, последний слой распознаёт на изображении кошку, собаку или птицу. Таким образом, глубокие нейронные сети не нуждаются во вмешательстве людей, а сами изучают иерархию объектов.

С математической точки зрения нейронная сеть определяет функцию $y = f(x; \theta)$. Она описывает соответствие входных данных x выходным y, где y — это категория в задаче классификации или выходное значение в задачах регрессии. Тренировка модели с использованием набора данных ведёт к вычислению параметра θ.

После завершения обучения предполагается, что нейронная сеть аппроксимирует целевую функцию f^* [14]. Правильно обученная ИНС может лучше соответствовать набору данных, а также делает прогноз, учитывая неочевидные зависимости от данных.

Глубокие нейронные сети могут быть реализованы по-разному в зависимости от конкретных практических задач. Например, свёрточные нейронные сети специализируются на компьютерном зрении, а рекуррентные нейронных сети лучше подходят для обработки естественного языка.

Функцию отображения $f(x)$ можно рассматривать как цепочку из многих связанных функций в виде $f(x) = f^{(n)}(f^{(n-1)}(\ldots f^{(2)}(f^{(1)}(x))))$; n связанных функций соответствуют глубине ИНС. Функция стоимости определяется на основе сравнения вывода y из $f(x)$ с целевым значением t из тренировочного набора данных. Цель обучения нейронной сети состоит в том, чтобы приблизить $f(x)$ функцей, минимизирующей функцию стоимости. Минимизация функции стоимости является задачей оптимизации, и для этого часто используют алгоритм градиентного спуска. В обратном распространении (backpropagation) градиент используется для итеративного обновления нейронной сети во время обучения. Оптимизатор решает, какие параметры и как должны быть обновлены, а скорость обучения (learning rate) задаёт размер шага, на который параметр обновляется на каждой итерации.

2.4. Обучение с подкреплением

Хотя алгоритмы RL эффективно решают различные задачи, им не хватает масштабируемости и размерности.

С ростом глубоких нейронных сетей в последние годы обучение с подкреплением начинает использовать их функции приближения и представления свойств [16]. Это помогает преодолеть недостатки алгоритмов обучения с подкреплением.

Это устраняет необходимость описывать свойства вручную, позволяя обучать модели, способные непосредственно выводить оптимальные действия, на основе необработанного и высокоразмерного ввода с сенсоров. Это проиллюстрировано на рис.2.7.

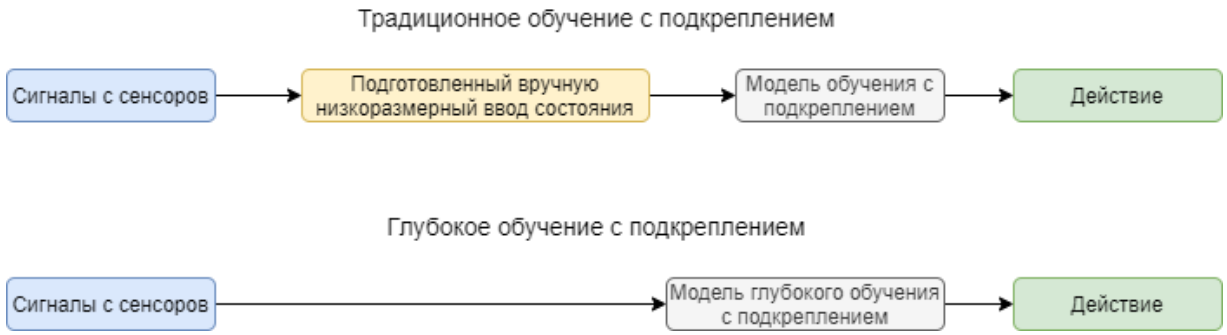

Рис.2.7. Сравнение DRL и традиционной RL, где необходимо явно указать, какие действия производить в каких состояниях. Использование глубокой нейронной сети в DRL позволяет обучаться и принимать решения на основе необработанного сенсорного ввода

Таким образом, использование ИНС в обучении с подкреплением создаёт новую область – глубокое обучение с подкреплением (Deep Reinforcement Learning, DRL).

2.4.1. Алгоритмы глубокого обучения с подкреплением

Два основных подхода к обучению с подкреплением: основанные на value–функции и на градиенте политики. Q-обучение — вероятностный value итерационный метод, цель которого — оценка Q-функции. Методы градиента политики же стараются оптимизировать политику напрямую.

Здесь нужно рассказать о марковском процессе принятия решений (Markov Decision Process, MDP).

2.4.1.1. Марковский процесс принятия решений

Свойство Маркова означает, что следующее состояние зависит только от текущего состояния, тем самым, при принятии решения, можно игнорировать все прошлые состояния и учитывать только текущее.

Процесс обучения с подкреплением является формой МППР. Он состоит из нескольких элементов:

- набор состояний окружающей среды S;
- набор действий A;
- динамика перехода $T(s_{(t+1)}|s_t, a_t)$, которая описывает распределение новых состояний s_{t+1}, в которые может попасть агент, совершив действие a_t в состоянии s_t;
- функция награды $R(s_t; a_t; s_{t+1})$;
- дискаунт фактор $\gamma \in [0, 1]$ для экспоненциального снижения будущих наград.

Политика π сопоставляет состояниям распределение вероятностей принятия действий: $\pi : S \rightarrow p(A = a|S)$.

Эпизод — это предопределённый период времени, когда среда, начиная со случайного состояния, порождает серию переходов. Переходы в эпизоде можно рассматривать как траекторию политики.

Сумма наград, собранных в траектории политики: $R = \sum_{t=0}^{T-1} \gamma^t r_{t+1}$.

Цель обучения с подкреплением состоит в том, чтобы получить оптимальную политику π^*, которая максимизирует ожидаемую награду из всех состояний: $\pi^* = argmax_x \mathbb{E}[R|\pi]$ [28].

2.4.2. Подход, основанный на функции состояния (Value Based подход)

Этот подход состоит в том, чтобы оптимизировать значение функции $V(s)$.

Value-функция —– это функция, которая сообщает нам максимальное ожидаемое будущее вознаграждение, которое агент получит в каждом состоянии. Значение каждого состояния — это общая сумма вознаграждений, на которую агент может рассчитывать в будущем, начиная с этого состояния.

2.4.2.1. Q-обучение

Функция состояния-значения (state-value function) $V^{\pi}(s) = \mathbb{E}[R|s, \pi]$ — ожидаемая награда с данными состоянием s и политикой π. В то же время в RL такой переход T не всегда возможен, поэтому обычно используется другая функция состояния-значения или функция качества $Q^{\pi}(s,a) = \mathbb{E}[R|s, a, \pi]$. Оптимальная политика получается через выбор на каждом шаге действия, которое максимизирует Q-функцию [34]. Q-обучение — это алгоритм без политики (off-policy), так как он жадно выбирает действие исходя из текущего состояния, вместо того, чтобы следовать политике.

Для рекурсивного вычисления Q^{π} применяется уравнение Беллмана:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{s_{t+1}}[r_{t+1} + \gamma Q^{\pi}(s_{t+1}, \pi(s_{t+1}))]. \tag{2.1}$$

Традиционное Q-обучение способно сформулировать оптимальную политику обучением состояние-действие-значение функции. Однако, оно предназначено для дискретных пространств действий с малым количеством измерений и не способно решать более-менее сложные задачи.

2.4.2.2. Глубокая Q-сеть (DQN)

Глубокая Q-сеть (DQN) — вариант Q-обучения, который использует глубокую сверточную нейронную сеть для вычисления Q-функции. Является прорывом в обучении с подкреплением. [11]

DQN была применена в игре Atari и достигла производительности, сопоставимой с человеческим уровнем. [18]

Для решения проблемы нестабильности и расхождения нелинейных функций аппроксиматоров, таких как нейронные сети, был применён метод Experience Replay [22]. Идея Experience Replay состоит в том, чтобы равномерно рандомизировать предыдущие переходы при обучении модели, что нарушает корреляцию последовательности наблюдений. Опыт агента $e_t = (s_t, a_t, r_t, s_{t+1)}$ на каждом шаге t сохраняется в буфер $D_t = (e_1, ..., e_t)$. Во время тренировки модели случайным образом извлекается небольшая часть опыта $(s; a; r; s') \sim U(D)$.

Еще одно новое приложение в DQN заключается в том, что создаются две Q-сети. Текущая Q-сеть $Q(s, a; \theta_i)$ обновляется итеративно во время обучения, а целевая Q-сеть $Q(s', a'; \theta_i)$ используется для получения целевого Q-значения и об-

новляется только периодически. Целевая Q-сеть уменьшает смещение, вызванное неточностями Q-сети в начале обучения.

На каждой итерации i Q-сеть обновляется на ошибку темпоральной разницы (temporal difference, TD):

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}[(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i))]. \quad (2.2)$$

Где γ —– это скорость затухания для будущих наград. θ_i^- параметры целевой Q-сети, и θ_i —текущей Q-сети на итерации. $[(r + \gamma \max_{a'} Q(s', a'; \theta_i^-)]$ — это цель Беллмана при данной оценке θ_i^-.

DQN для решения проблемы извлечения низкоразмерных свойства из высокомерного необработанного сенсорного сигнала, такого как пиксели изображения в играх, были применены глубокие нейронные сетеи. Тем не менее, он всё ещё ограничен его дискретным и низкоразмерным пространством действий.

2.4.3. Линия поведения (Policy Based)

Вместо получения оптимальной политики путём поддержания Q-функции, алгоритм напрямую ищет оптимальную политику путём максимизации ожидаемого значения $\mathbb{E}[R|\pi_\theta]$. Необходимо итеративно подгонять параметр θ сети так, чтобы максимизировать $\mathbb{E}[R|\pi_\theta]$.

В основном используется оптимизация, основанная на градиенте, так как это более эффективно при работе с большими сетями со множеством параметров. [10]

2.4.3.1. Градиент политики (Policy Gradients)

В методах, основанных на градиенте политики, параметризованную политику представляет нейронная сеть, которая обновляется при изучении сигналов. В обучении с подкреплением без модели для оценки градиента на примерах, сгенерированных в траектории политикой, используется правило REINFORCE или функция оценки. Предположим, что $f(x)$ — это функция оценки, где x — случайная величина для одного перехода. Градиент политики может быть рассчитан с использованием отношения правдоподобия:

$$\nabla_\theta \mathbb{E}_x[f(x)] = \mathbb{E}_x[f(x)\nabla_\theta \log px|\theta]. \quad (2.3)$$

Теперь рассмотрим траекторию τ с переходами (a_t, s_t, r_t, s_{t+1}) в соответствии с политикой, тогда градиент политики — это:

$$\nabla_\theta \mathbb{E}[R_\tau] = \mathbb{E}[\sum_\tau R_\tau \nabla_\theta \log \pi a_\tau | s_\tau; \theta]. \quad (2.4)$$

Недостатком policy gradient является низкая скорость работы —– требуется большое количество вычислений для подсчёта награды. Также policy gradient может «застрять» в локальном оптимуме, не найдя глобального.

2.4.3.2. Актор-критик

Так как value-функция может предоставить обучающие сигналы для прямого поиска оптимальной политики, естественным было объединить два подхода.

В DRL две нейронные сети, представляющие актора и критика соответственно, используются для приближения функции, где актор (политика) учится по Q-значениям, оценённым критиком (value-функция) [10]. На рис.2.8 показано, как актор и критик сети взаимодействуют с окружающей средой.

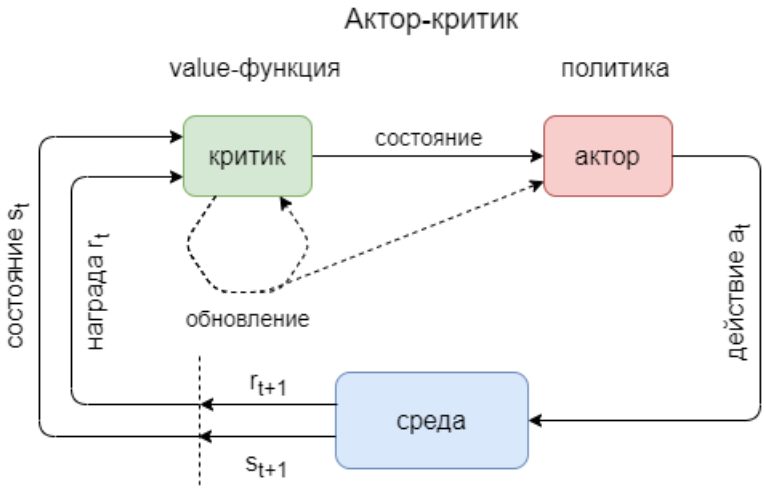

Рис.2.8. Актор получает состояние из окружающей среды и реагирует на действие, критик получает состояние и награду и рассчитывает TD ошибку для обновления себя и актора. Основано на [10]

2.4.3.3. Глубокий детерминированный градиент политики (DDPG)

DDPG — это актор-критик алгоритм, в котором нет модели и нет политики [6]. Он расширяет DPG использованием глубоких нейронных сетей. Также, он использует хорошо зарекомендовавший себя приём DQN с текущей и целевой Q-сетями (сети критиков) и experience replay, чтобы стабилизировать обучение. И, наконец, он использует детерминированную политику (сеть акторов) вместо политики стохастического поведения.

В отличие от стохастической политики, которая определяет вероятность распределения через действия в данном состоянии, детерминированность в DDPG подразумевает, что конкретное действие аппроксимируется в данном состоянии. Соответственно конкретное состояние для следующего шага, также детерминировано.

Следовательно, вместо использования рекурсивного программирования, как в уравнении Беллмана, используется детерминированная политика $\mu : S \leftarrow A$ [6]

$$Q^{\mu}(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1}\ E}[r(s_t, a_t) + \gamma Q^{\mu}(s_{t+1}, \mu(s_{t+1})], \tag{2.5}$$

что очень похоже на Q-обучение – алгоритм с жадной политикой. Сеть критика обновляется по функции потерь

$$L(\theta^{Q}) = \mathbb{E}[(Q(s_t, a_t|\theta^{Q}) - y_t)^2], \tag{2.6}$$

где

$$y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1}) j \theta^{Q}). \tag{2.7}$$

Сеть актора обновляется функцией потерь

$$\begin{aligned} \nabla_{\theta\mu} J \approx \mathbb{E}[\nabla_{\theta\mu} Q(s, a\theta^{Q})|_{s=s_t, a=\mu(s_t|\theta^{\mu})}] = \\ = \mathbb{E}[\nabla_a Q(s, a\theta^{Q})|_{s=s_t, a=\mu(s_t)} \nabla_{\theta\mu} \mu(s|\theta^{\mu})|s=s_t]. \end{aligned} \tag{2.8}$$

Как и в DQN, чтобы избежать расхождения, в DDPG применяется мягкое обновление (soft update) для целевых сетей критика и актора. Они обновляются только раз в указанное количество шагов.

2.5. Выводы

В этой главе были рассмотрены основные понятия, связанные с машинным обучением и обучением с подкреплением, в частности. Это позволяет перейти к рассмотрению мультиагентных систем и алгоритмов для работы с ними.

ГЛАВА 3. ОБУЧЕНИЕ МУЛЬТИАГЕНТНЫХ СИСТЕМ

В этой главе представлена соответствующая работа по теме «Мультиагентное глубокое обучение с подкреплением». Прежде всего, речь идет о современных алгоритмах, которые соответствуют мультиагентным сценариям, выбранным в данной работе.

3.1. Мультиагентные алгоритмы

Методы обучения с подкреплением, которые специализируются на решении задач с одним агентом, плохо адаптированы для многих реальных задач, таких как автономные транспортные средства, управление водоразделом, согласованный поиск объектов, управление городским траффиком, тороговля на фондовом рынке. Поэтому необходимо расширить эти алгоритмы или даже создать новые для более сложных сценариев с настройками для нескольких агентов.

Некоторые алгоритмы, которые вдохновили эту работу, иллюстрируются в следующих разделах. Этими алгоритмами являются Multi Agent Deep Deterministic Policy Gradient [25], Counterfactual baseline for multi-agent policy gradient [7] и emergent grounded compositional language [24].

3.1.1. Детерминированная политика для нескольких агентов

Multi Agent Deep Deterministic Policy Gradient (MADDPG) — это расширение DDPG, применяемое к настройкам нескольких агентов. Чтобы учесть все состояния среды и политики всех агентов RL в игровом сценарии, алгоритм учитывает совместные наблюдения и действия всех агентов при обучении сетей акторов и критиков. Когда дело доходит до принятия решения, сеть актора каждого агента учитывает только локальные наблюдения, см. рис.3.1. Эта структура централизованного обучения и децентрализованного исполнения позволяет каждому агенту вычислять оптимальную политику по консистентному градиентному сигналу. [25]

Совместные наблюдения всех агентов обозначаются через $\mathbf{x} = (o_1, ..., o_N)$, совместные действия — через $a = (a_1, ..., a_N)$. Они вместе с наградами r хранятся в буфере D в виде $(\mathbf{x}, a, r, \mathbf{x}')$. Случайная выборка из S выборок $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$ извлекается из D, а критик каждого агента обновляется путём минимизации потерь:

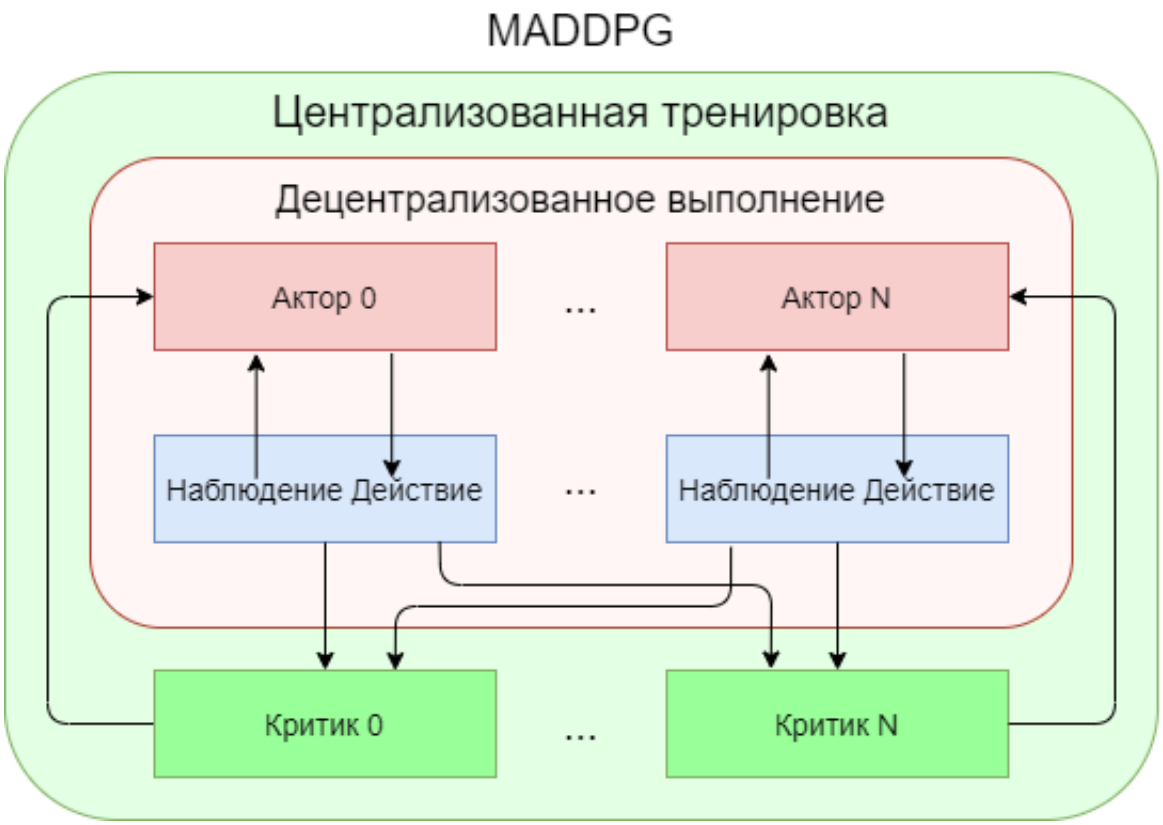

Рис.3.1. Структура сетей алгоритма MADDPG и информационный поток между централизованными критиками и децентрализованными акторами. Основано на [25]

$$L(\theta_i) = \frac{1}{S}\sum_j (y^j - Q_i^{\mu}(\mathbf{x}^j, a_1^j, ..., a_N^j))^2. \tag{3.1}$$

И актор обновляется семплированым градиентом политики:

$$\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i}\mu_i(o_i^j)\nabla_{a_i} Q_i^{\mu}(x^j, a_1^j, ..., a_i, ..., a_N^j)|_{a_i=\mu_i(o_i^j)}. \tag{3.2}$$

Подобно DDPG, целевые сети обновляются «мягко» на каждом шаге на $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$ с $\tau \ll 1$.

Алгоритм MADDPG хорошо подходит для сценариев с физической навигацией. Было решено продолжить изучение этого алгоритма с помощью многомерного пространства действий, Он хорошо подходит для сценариев, в которых агенты могут взаимодействовать при выполнении физических действий.

3.1.2. Контрафактный мультиагентный градиент политики (Counterfactual Multi-Agent Policy Gradient, COMA)

Контрфактический мультиагентный градиент политики (COMA) — это мультиагентный вариант метода актор-критик, который использует единого централизо-

ванного критика для приближения к Q-функции и отдельных децентрализованных акторов для оптимизации политики каждого агента $\pi(h)$. [7]

В кооперативных задачах с несколькими агентами сложность сотрудничества возрастает с увеличением количества агентов. Таким образом, было бы нецелесообразно и неэффективно иметь одну единственную оптимальную политику для всех агентов. Вместо этого децентрализованные политики для каждого агента формулируются в виде проблем с несколькими агентами. В COMA один централизованный критик и отдельные акторы тренируются на совместных действиях и совместных наблюдениях. Когда дело доходит до исполнения, каждый актор генерирует действия на основе собственной истории наблюдения h, см. рис.3.2.

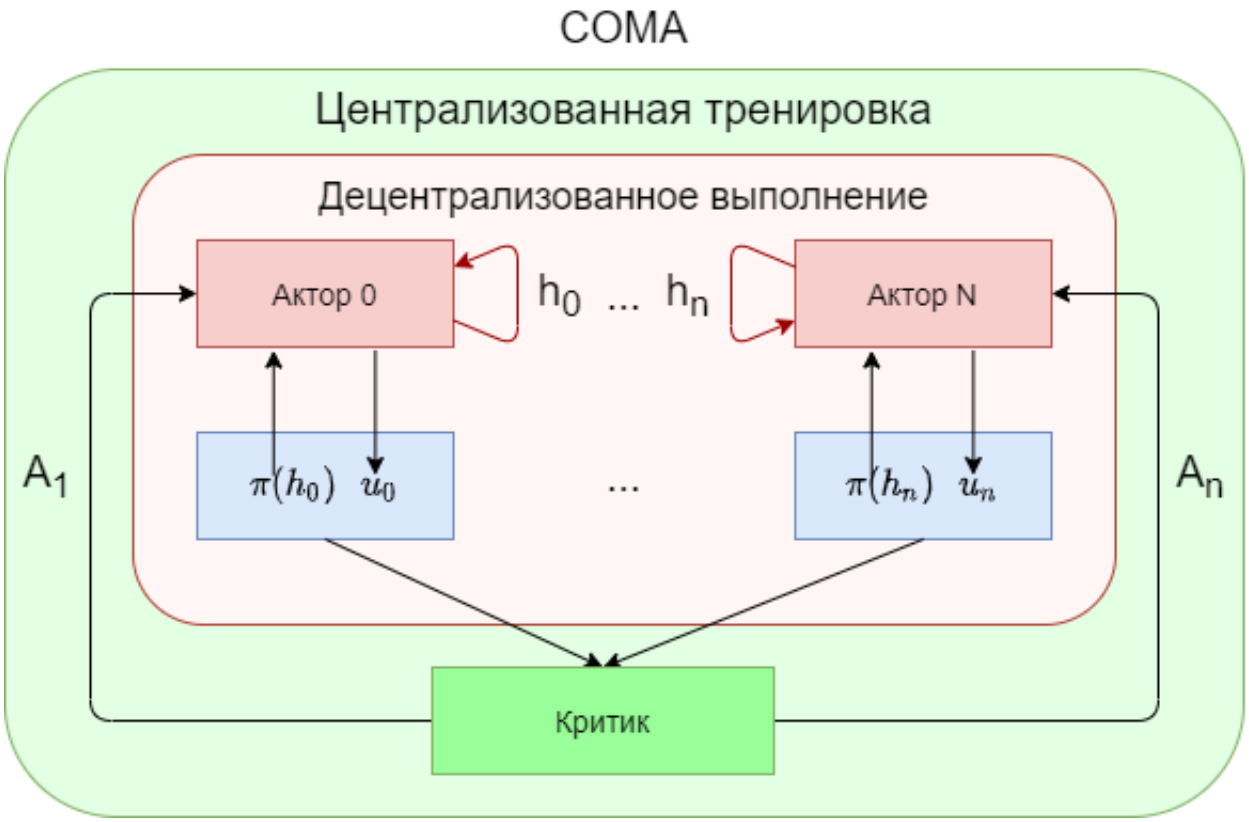

Рис.3.2. Структура сетей алгоритма COMA и информационный поток между централизованным критиком и децентрализованными акторами [7]

COMA предназначен для мультиагентных совместных сценариев с глобальной функцией награды для всех агентов. Однако агенты, обученные COMA, изучают отдельные политики без явного общения. [7]

3.1.3. Возникающий язык (Emergent Language)

Grounded compositional language обозначает простой язык, где агенты связывают конкретные символы с конкретными объектами, а затем объединяют эти

символы в значимые понятия [35]. Язык представлен в виде абстрактных дискретных символов, произнесённых агентами, которые не имеют заранее определённого значения, но возникли и сформировались в процессе обучения в соответствии со средой и целями [24]. В отличие от естественного языка, для работы с которым извлекаются языковые шаблоны из большого набора текстовых данных, этот язык, возникший во время обучения с подкреплением, понятен только агентам и используется ими для сотрудничества друг с другом в достижении общих целей.

3.2. Действия и награды

3.2.1. Архитектура ветвления действий (Action Branching)

Обычно довольно трудно исследовать проблемы с многомерными пространствами действий. Архитектура ветвления действий предназначена для решения таких проблем. Например, в среде с N-мерным пространством действий и d_n дискретных последующих действий для каждого измерения n необходимо рассмотреть в общей сложности $\Pi_{n=1}^{N} d_n$ возможных действий [36]. Правильно спроектированные разветвлённые архитектуры действий могут эффективно и результативно исследовать такое большое многомерное пространство действий [36].

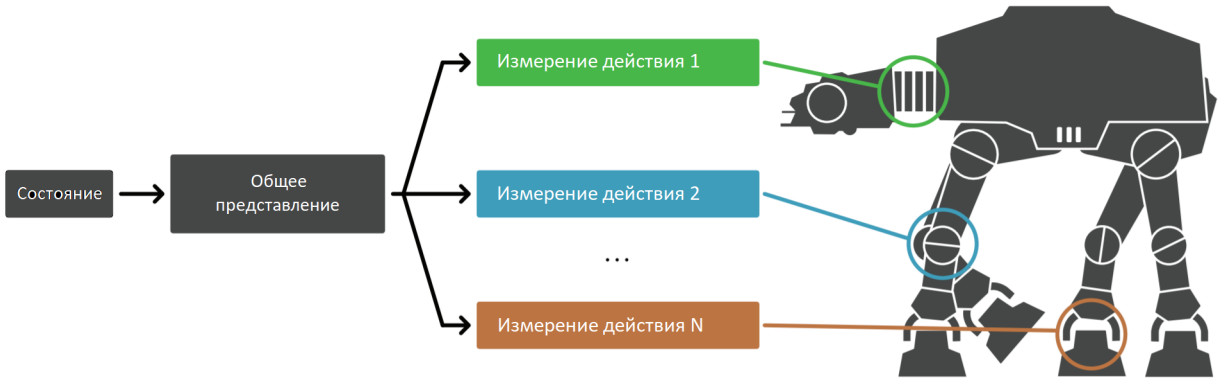

Рис.3.3. Архитектура сети для ветвления действий. Основано на [36]

Основной идеей архитектуры является модуль совместного принятия решений, который извлекает скрытое представление из входного наблюдения и создаёт отдельные выходные ветви для каждого измерения действия.

В архитектуре сети для ветвления действий, n измерений действий представляют собой n относительно независимых поддействий. Например, для робота, где сеть принимает состояние в качестве входных данных и разделяет скрытые слои для извлечения представлений признаков, а затем расходится по ветвям

для относительно независимых поддействий, включая, например, высказывание, действие рукой, действие ногой и т.д., как показано на рис.3.3.

В архитектурах ветвления действий значения Q рассчитываются для каждого измерения действия. Тем не менее, хотелось бы, чтобы многомерное действие оценивало одно единственное Q-значение, выводимое сетью критика.

3.2.2. Архитектура гибридного вознаграждения

Как показано в [19], жизненно важно иметь точную оптимальную value-функцию в обучении с подкреплением, поскольку она оценивает ожидаемую награду как сигнал для оптимизации политики. Как только оптимальная value-функция изучена, из неё можно получить оптимальную политику.

Рис.3.4. Глубокая нейронная сеть с одной «головой» аппроксимирует одну единственную Q-функцию с глобальной функцией награды (слева) и Q-сеть с гибридной архитектурой вознаграждения (справа). Сеть имеет n «голов», аппроксимирующих n Q-функций. Каждая Q-функция оценивает Q-значение с помощью соответствующей разложенной функции награды. Основано на [19]

Более того, две разные value-функции могут привести к одной и той же политике, когда агент действует «жадно» в соответствии с ними [19]. Следовательно, можно было бы изучить несколько более простых value-функций, если изучить сложную value-функцию сложно или даже невозможно. В таком случае глобальная функция награды может быть соответственно разложена на несколько различных функций награды:

$$R_{rev}(s, a, s') = \sum_{k=1}^{n} R_k(s, a, s'), \tag{3.3}$$

где R_{rev} — глобальная функция награды, которая разбита на n функций награды. Каждая разложенная функция награды зависит от подмножества состояний и имеет свою собственную функцию Q-значения. Глубокая Q-сеть с общими скрытыми слоями и n «головами» (под головой поразумевается выходной слой сети — для каждого действия свой) используется для аппроксимации n Q-значений, обуслов-

ливающих текущее состояние и действие с различными функциями разложения награды (рис.3.4):

$$Q_{HRA}(s, a; \theta) := \sum_{k=1}^{n} Q_k(s, a; \theta). \tag{3.4}$$

Q-сеть затем итеративно улучшается за счёт оптимизации функции потерь

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}[\sum_{k=1}^{n}(y_{k,i} - Q_k(s, a; \theta_i))^2], \tag{3.5}$$

где

$$y_{k,i} = R_k(s, a, s') + \gamma \max_{a'} Q_k(s', a'; \theta_{i-1}), \tag{3.6}$$

θ_i — это веса Q-сети в текущей итерации i, а θ_{i-1} — веса отдельной целевой сети в предыдущей итерации.

В гибридной архитектуре вознаграждений n Q-значений выводятся n «головами» из одной единственной Q-сети. Однако для того, чтобы иметь консистентный градиент для улучшения политики для каждого измерения действий, следует создавать отдельные Q-сети для оценки Q-значений для каждого вида действий.

3.3. Учебная программа

Обучение по учебной программе — это стратегия обучения, основанная на процессе обучения человека. В системе образования люди обучаются путём ознакомления с концепциями, которые строятся от простых до сложных уровней, от конкретных до более абстрактных уровней, от простых структур до сложных модулей.

Машинное обучение заимствует такой подход. Учебная программа усложняет обучение программы от небольших подзадач или простых аспектов к более тяжёлым и сложным задачам. Ожидается, что обученная модель достигнет высокой эффективности и результатов обучения [26].

В методе учебная программа простой аспект проблемы проливает свет на общую картину, а сложные факторы постепенно добавляются и раскрывают больше деталей целой картины. Поиск решения для сглаженной версии проблемы и затем переход к более детализированной может помочь обучению решить проблему постепенно [3].

Математически $C_\gamma(\theta)$ определяется как функции стоимости задачи, в которой γ - уровень сложности, а θ - параметры. Начальную сглаженную версию C_0 обычно легко оптимизировать, доведя θ до локального минимума. Локальный минимум затем используется в качестве основы для следующего уровня сложности. С увеличением γ C_γ становится менее сглаженной при дальнейшем поиске следующего локального минимума на основе предыдущего локального минимума. [9]

Учебный план также можно рассматривать как последовательное перевзвешивание, сначала на наборах данных из простых примеров, а затем на полном наборе данных. С увеличением уровня сложности в набор обучающих данных добавляются немного более сложные примеры, которые используются для повторного взвешивания распределения. В конце используется полный набор обучающих данных. [9]

3.4. Выводы

Произведённого обзора должно быть достаточно для того, чтобы определиться с применяемыми в данной работе методами. В следующей главе речь пойдёт о них.

ГЛАВА 4. ПРИМЕНЯЕМЫЕ МЕТОДЫ

Основным алгоритмом, применяемым в этой работе, является градиент глубокой детерминированной политики (Deep Deterministic Policy Gradient, DDPG), который использует архитектуру актор-критик и может работать в пространстве непрерывных действий. Поскольку сценарий игры предполагает совместную работу нескольких агентов, используется мультиагентная модификация алгоритма DDPG — мультиагентный градиент глубокой детерминированной политики (MADDPG). Основное его отличие, делающее его более подходящим для работы с несколькими агентами, состоит в том, что MADDPG имеет N наборов ИНС критиков-акторов, где N соответствует количеству агентов в сценарии, благодаря чему каждый агент имеет свой собственный механизм оптимизации политики. В оригинальном дизайне [25] это сделано для адаптации как к совместной работе, так и к конкуренции между агентами.

Поскольку эта работа фокусируется на совместной работе нескольких агентов, тестируются несколько вариантов MADDPG, подходящих для совместной работы.

4.1. Основные методы

4.1.1. Архитектура актор-критик

Как уже упоминалось выше, алгоритм MADDPG имеет набор сетей акторов-критиков для каждого агента в игровой среде. Для облегчения многоагентной совместной работы во время тренировки алгоритм учитывает наблюдения и действия всех агентов. Политики оптимизируются путём оценки качества, поведения всех агентов в разных состояниях среды. Таким образом, в условиях среды, требующей взаимодействия агентов, будет разработана политика, оптимальная для сотрудничества.

Глубокая нейронная сеть может рассматриваться как аппроксиматор нелинейных функций. В решении сложных задач глубокого обучения с подкреплением нейронная сеть нестабильна [6]. Эту проблему решает использование целевой сети, поскольку мягкое и разрежённое обновление целевой сети замедляет её приближение к исходной сети и уменьшает влияние ошибок. Таким образом, нарушается корреляция между текущим и целевым Q-значениями. Хотя это и

снижает скорость, но улучшает стабильность обучения. В алгоритме MADDPG и у актора, и у критика есть свои целевые сети.

4.1.2. Воспроизведение опыта (Experience Replay)

Воспроизведение опыта — это механизм, в котором во время тренировки в игровой среде буфер используется для сбора опыта, образцы которого из этого буфера затем случайным образом извлекаются для обучения модели.

Опыт генерируется последовательно во время взаимодействия агентов со средой в хронологическом порядке. Неизбежно то, что собранные последовательности опыта коррелируют друг с другом. Из-за этого сеть легко переобучается на имеющихся последовательностях опыта. В итоге, переобученная сеть не может обеспечить разнообразный опыт для последующего обучения.

Опыт случайно отбирается в небольшие блоки. Это нужно не только для эффективного использования аппаратного обеспечения и увеличения скорости обучения, но также нарушает корреляцию последовательности в буфере. Таким образом, сети имеют возможность формулировать более обобщённые политики по независимым друг от друга данным.

Для применения воспроизведения опыта в буфер складывается опыт в виде кортежей переходов, включающих наблюдения, действия, награды и последующие наблюдения, то есть $e_t = (\mathbf{s}, \mathbf{a}, \mathbf{r}, \mathbf{s}')$. Поскольку это многоагентная игровая среда, $\mathbf{s}$ — это совместные наблюдения, которые представляют собой совокупность наблюдений n агентов, $\mathbf{s} = [s_0, s_1, ..., s_n]$. То же самое относится к действиям, следующим наблюдениям и наградам. Буфер имеет ограниченную ёмкость, и при переполнении старый опыт вытесняется новым.

4.1.3. Тренировка ИНС

В то время как агенты исследуют игровую среду, опыт перехода на каждом шаге собирается в буфер памяти. Обучение происходит только тогда, когда количество кортежей в буфере превышает определённую величину. Когда начинается обучение, актор и критик каждого из n агентов обновляется на каждом шаге, в то время как целевой актор и целевой критик обновляются с задержкой (раз в определённое количество шагов).

На рисунке рис.4.1 показано, как обновляется набор сетей акторов-критиков. Текущее Q-значение $Q_i(\mathbf{s}, \mathbf{a}|\theta_i^Q)$ оценивается по сети критика, при этом на вход

подаются совместные текущие наблюдения **s** и совместные действия **a**. Целевое Q-значение y_i рассчитывается из вознаграждения и дисконтированного следующего Q-значения $Q'_i(\mathbf{s}', \mathbf{a}'|\theta^{Q'_i})$, аппроксимированного по целевому критику. Входные данные для сети целевого критика – это совместные последующие наблюдения **s**' из буфера и совместные последующие действия **a**', выбранные целевыми сетями акторов:

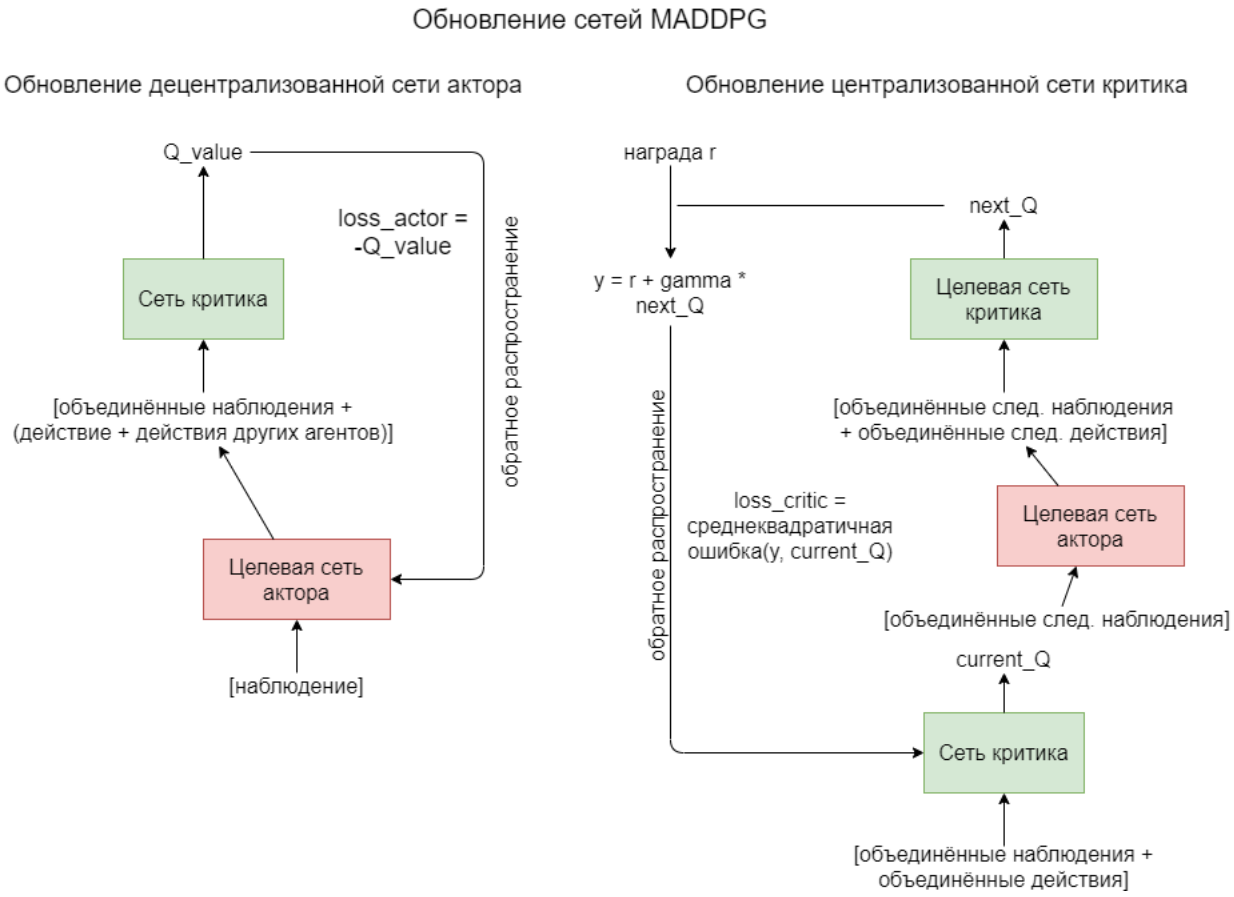

Рис.4.1. Обновление сети актора и критика для каждого агента в MADDPG

$$\begin{aligned}\mathbf{a}' &= [a'_0, a'_1, ..., a'_n] = \\ &= [\mu'_0(s'_0|\theta^{\mu'_0}), \mu'_1(s'_1|\theta^{\mu'_1}), ..., \mu'_n(s'_n|\theta^{\mu'_n})].\end{aligned} \tag{4.1}$$

Целевое Q-значение:

$$y_i = r_i + \gamma Q'_i(\mathbf{s}', \mathbf{a}'|\theta^{Q'_i}). \tag{4.2}$$

где r_i это награда – i-го агента, и γ — это коэффициент дисконтирования, или скорость затухания влияния будущих наград.

Наконец, сеть критика обновляется путём минимизации функции потерь:

$$L_i(\theta^{Q_i}) = \frac{1}{S}\sum(y_i - Q_i(\mathbf{s}, \mathbf{a}|\theta^{Q_i}))^2, \tag{4.3}$$

где S — это размер тренировочного набора.

Сеть актора i-го агента обновляется путём максимизации Q-значения сетью критика. Критик принимает совместные текущие наблюдения **s** из буфера и совместные текущие действия, в которых i-е действие подменяется последним действием, выбранным актором, то есть:

$$[a_0, a_1, a_i, ..., a_n] = [a_0, a_1, \mu_i(s_i|\theta^{\mu_i}), ..., a_n]. \tag{4.4}$$

Для оптимизации уместно преобразовать задачу максимизации в задачу минимизации. Следовательно, функция потерь для обновления сети актора:

$$L_i(\theta^{\mu_i}) = -Q_i(\mathbf{s}, [a_0, a_1, \mu_i(s_i|\theta^{\mu_i}), ..., a_n]|\theta^{Q_i}). \tag{4.5}$$

Сети целевых актора и критика обновляются не полным копированием сетей актора и критика, а с использованием мягкого обновления (soft update):

$$\begin{aligned} \theta^{\mu_i'} &= \tau\theta^{\mu_i} + (1-\tau)\theta^{\mu_i'}, \\ \theta^{Q_i'} &= \tau\theta^{Q_i} + (1-\tau)\theta^{Q_i'}, \end{aligned} \tag{4.6}$$

где τ обычно очень мала, например, 0,01 в этой работе.

4.2. Прикладные методы

4.2.1. Архитектура ветвления действий (Action Branching)

В сценариях, подразумевающих совместную работу нескольких агентов, наряду с физическими действиями важно иметь и коммуникационные действия. В некоторых случаях количество действий может быть большим, как в [36]. В игровых сценариях этой работы агент имеет два вида действия, которые представляют физическое движение и действие, выбранное для общения.

Полное действие в таких сценариях включает в себя два относительно независимых действия. Сеть акторов проектируется таким образом, что она выделяет скрытые представления из наблюдений и имеет две «головы» на выходе. Одна возвращает физическое действие, вторая – действие для общения.

В соответствии с теорией архитектуры ветвления действий [36], можно оптимизировать каждое измерение действия относительно независимо. Полное действие в итоге представляет собой объединение двух действий.

4.2.2. Исследовательский шум (Exploration Noise)

Исследование и эксплуатация (Exploration and exploitation) — это дилемма в обучении с подкреплением. Эксплуатация — это следование агента текущей политике, с целью совершения «жадных» действий, которые приносят наибольшую награду. Исследование берёт на себя риски, чтобы попробовать другие действия, которые потенциально могут принести лучшую награду в долгосрочной перспективе. Исследование необходимо агенту, ищущему оптимальную политику, хотя оно кажется неоптимальным в нынешней ситуации и даёт меньшее вознаграждение. В обучении с подкреплением агент обычно больше исследует окружающую среду в начале обучения. В ходе оптимизации политики с течением времени агент постепенно уменьшает и стабилизирует исследование до низкого уровня и в итоге больше придерживается получившейся оптимальной политики.

Чтобы включить исследование, на действия накладываются шумы. Какой шум применять, определяется настройкой среды. Процесс Орнштейна-Уленбека используется в DDPG в [6] для получения коррелированных по времени исследований. Этот процесс считается, эффективным для проблем физического контроля с инерцией. Гауссовское распределение шума используется для физических движений в [24]. В этой работе для наложения шума на действия, генерируемые сетью акторов, выбирается стандартное гауссовское распределение:

$$\mu_i'(s_i) = \mu_i(s_i|\theta^{\mu_i}) + \mathcal{N}, \tag{4.7}$$

где μ' — политика исследования, а $\mathcal{N}$ — шум исследования, который можно выбрать в зависимости от настроек среды, $\mathcal{N}$ затухает на каждом шаге со скоростью ε, то есть $\mathcal{N} \leftarrow \varepsilon\mathcal{N}$.

4.3. Варианты алгоритмов

4.3.1. MADDPG с декомпозированным вознаграждением (Decomposed Reward)

Предполагается сценарий игры, в котором агенты выполняют относительно независимые поддействия, каждое поддействие может воздействовать на среду и приводить к вознаграждению, которое отделено от глобального вознаграждения. Идея декомпозирования награды MADDPG состоит в том, что для каждого под-

действия может быть сформулирована независимая политика. Каждый агент имеет *n* независимых наборов сетей актор-критиков, каждый из которых соответствует одному виду действия. Ожидается, что политики для отдельных видов действий могут быть оптимизированы путём обучения соответствующих групп критиков.

В этой работе планируется использовать этот вариант в сценарии Simple Reference.

4.3.2. MADDPG с общим мозгом

Другой вариант MADDPG вдохновлён [24]. В нём есть только один набор сетей акторов-критиков, который используется всеми агентами. Это подразумевает, что все агенты имеют одинаковую оптимальную политику. Этот подход использует предположение о том, что все агенты имеют одно и то же пространство действий и пространство наблюдений, и они также имеют общую глобальную награду. Этот подход особенно эффективен для коммуникационных действий, поскольку композиционный язык постоянно появляется среди всех участников игрового сценария. В варианте с общим набором сетей все агенты говорят на одном языке, в отличие от стандартного MADDPG, где каждый агент может интерпретировать один и тот же ориентир по-разному. Это особенно важно, когда сотрудничают более двух агентов, поскольку им нужно общаться на одном языке.

4.4. Выводы

В этой главе были рассмотрены методы, которые были выбраны для исследования. Именно эти методы, подходы и алгоритмы были задействованы в экспериментах, речь о которых пойдёт в следующей главе.

Сценарии, в которых производились эксперименты, уже упоминались в 1.2 Сценарии. Более подробно они так же будут рассмотрены в следующей главе.

ГЛАВА 5. СЦЕНАРИИ И ЭКСПЕРИМЕНТЫ

Эксперименты проводятся в мультиагентной среде multiagent-particle-envs [13] от компании OpenAI.

5.1. Эксперименты

Для тестирования был использован компьютер со следующими параметрами:

- Процессор: Intel Core i5-8365U 1.6 GHz;
- Количество физических ядер: 4;
- Количество логических ядер: 8;
- Объем оперативной памяти: 16 Гб.

5.1.1. Эксперимент на одном мозге

В алгоритме MADDPG каждый агент имеет набор сетей критиков, чтобы иметь собственный механизм оптимизации политики. Тем не менее, два агента в Simple Reference полностью симметричны в том смысле, что они оба имеют одинаковое пространство наблюдения, пространство действий и имеют общую глобальную награду. Тем самым агенты формулируют аналогичные оптимальные политики.

Таким образом, каждому агенту можно было бы не создавать свой собственный «мозг» из сетей критиков, а вместо этого тренировать один «мозг» для всех агентов. Кроме того, единый «мозг» MADDPG позволяет агентам общаться на одном «языке», что означает, что они произносят и понимают ориентиры в одних и тех же кодировках.

Также был разработан вариант этого алгоритма. Можно сказать, что в данном случае имеются два мозга: один для агентов из одной команды, второй – для агентов из другой команды. Этот вариант был применён к сценарию Simple Tag. Это возможно благодаря тому, что агенты из одной команды в этом сценарии имеют одинаковое пространство наблюдения, пространство действий и имеют общую глобальную награду, но это не так для агентов из разных команд. Поэтому и пришлось использовать отдельный набор сетей для каждой команды.

5.1.2. Эксперимент с декомпозированным вознаграждением

Чтобы обеспечить чёткие сигналы для двух независимых поддействий, мы попробовали реализовать отдельные наборы актор-критиков для физического движения и общения.

Рис.5.1. С левой стороны от пунктирной линии — действие агента, которое представляет собой совокупность физического движения u и коммуникационного действия c. С правой стороны показано, как награда рассчитывается и назначается для оценки действий. Два агента имеют одинаковую глобальную награду — среднее значение между r_0 и r_1

Из рис.5.1 видно, что физическое перемещение u_0 агента 0 оценивается r_0, а высказывание c_0 - r_1. Окончательное вознаграждение затем раскладывается как кортеж из r_0 и r_1, т.е. $[r_0; r_1]$, причём каждый элемент соответствует сигналу оценки физического движения и связи соответственно. Точно так же, последний набор вознаграждений для агента 1 равен $[r_1; r_0]$.

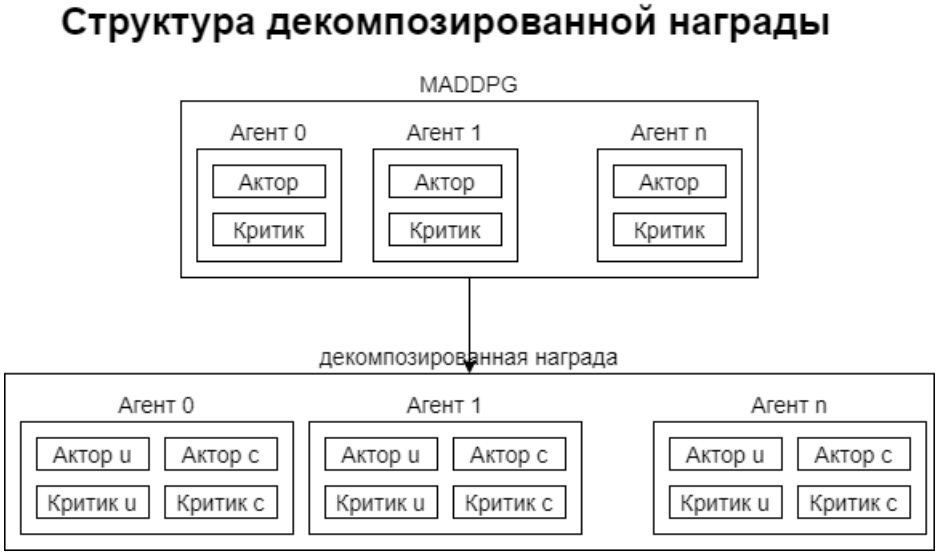

Рис.5.2. Сетевая структура декомпозированного вознаграждения по сравнению со стандартным MADDPG

Для физического движения и общения созданы два отдельных набора сетей акторов-критиков, чтобы обеспечить согласованные и точные градиенты для оптимизации двух поддействий, см. рис.5.2. Разложения вознаграждения можно рассматривать как параллельное обучение двух слушателей из Simple Speaker Listener. Слушатель 1 — это агент 0 с коммуникацией c_0 в качестве говорящего и агент 1 с физическим движением u_1 в качестве слушателя, а слушатель 2 — агент 1 с $c1$ в качестве говорящего и агент 0 с u_0 в качестве слушателя, как показано на рис.5.1.

MADDPG с разложенным вознаграждением может быть хорошо приспособлен только к Simple Reference, но не может быть обобщён для других игровых сценариев, поскольку, пространства наблюдений и действий у агентов в этом сценарии симметричны, а главное, каждый агент производит оба вида действия. В большинстве других сценариев это не так. Следовательно, необходимо проводить больше экспериментов с общей глобальной наградой.

5.1.3. Эксперимент с обучением по учебной программе

Обучение по учебной программе применено с целью решения проблемы расходящихся сетей, возникшей в ходе обучения. Примеры обучения организованы в таком порядке, что постепенно вводятся всё более сложные концепции.

Варианты учебной программы на примере сценария Simple Reference:

- Увеличивать количество ориентиров постепенно. На первом уровне сложности в этой учебной программе цель фиксирована для обоих агентов, скажем, обозначена красным ориентиром. На втором уровне добавляется ещё один ориентир, например, зелёный, и цель выбирается случайным образом из двух ориентиров. Последний уровень — это исходная настройка сценария, т. е. рандомизация цели из трёх ориентиров. Цель этой учебной программы состоит в том, чтобы агенты учились от простого сценария с меньшим количеством ориентиров к более сложному сценарию со всеми ориентирами.
- Обучить агента 0 в качестве говоруна и агента 1 в качестве слушателя на первом этапе, а на следующем этапе обучить двух агентов в качестве говорунов и слушателей в исходном игровом сценарии.
- Предоставить агентам заранее определённые коммуникационные действия для трёх цветов, скажем, $[1; 0; 0]$ для красного, $[0; 1; 0]$ для зелёного и

$[0; 0; 1]$ для синего. Таким образом, первая фаза состоит в том, чтобы изучить физическое движение, чтобы правильно двигаться. Затем, одному из агентов больше не предоставляются предопределённые коммуникационные действия, но взамен предоставляется связь с другим агентом (показываются его коммуникационные действия). Ожидается, что другому агенту будет легче научиться правильно общаться. Наконец, мы возвращаемся к исходным игровым настройкам для двух агентов, когда изучаются как физическое движение, так и общение.

5.2. Метрики измерения консистентности коммуникационных действий

Обучение модели можно ориентировочно оценивать на глаз при рендеринге физического движения, но можно и создать метрику и строить по ней графики. Выбранная метрика для коммуникационных действий заключается в том, что агент должен однозначно интерпретировать комунникационные действия и выбирать соответствующие им цвета цели.

Методика, используемая для измерения сходимости коммуникации, заключается в том, что создаётся матрица, которая сопоставляет интерпретацию агента и текущий цвет цели.

Действие общения — это вектор из трёх вещественных чисел. Каждой строке матрицы соответствует действие, причём максимальное значение заменено 1, а остальные — 0.

Тогда после нормализации матрицы, если обучение сходится, она должна выглядеть примерно так:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

или

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

и т. д.

Неважно, как агенты интерпретируют цвета, детерминант этой матрицы, в конце концов, стремится к 1 или -1. В экспериментах данной работы использовался именно этот способ.

Ещё один способ измерения заключается в том, чтобы смотреть, какое действие коммуникации выбрал говорун при наблюдении определённого цвета и проверять, совпадает ли оно с тем, что было выбрано в прошлый раз с тем же наблюдением. График с результатом должен приблизиться к 100% после схождения.

5.3. Сценарии

5.3.1. Сценарий 1. Simple Speaker Listener

Сценарий Simple Speaker Listener воспроизводится и тестируется с использованием алгоритма MADDPG [25].

Сценарий упоминается в разделе Вводная глава: Сценарий 1. Simple Speaker Listener. В этом сценарии два агента имеют разные пространства действий и наблюдений. Наблюдение говоруна o_s — это цвет цели, обозначенный 3-х канальным вектором $d \in \mathbb{R}^3$. Наблюдение слушателя o_l – это вектор конкатенации его собственной скорости $\upsilon \in \mathbb{R}^2$ его расстояния до трёх ориентиров $p = [p_1, p_2, p_3], p_i \in \mathbb{R}$ и сигнал, произнесённый говоруном на предыдущем временном шаге:

$$o_s = [d],$$
$$o_l = \begin{bmatrix} \upsilon \\ p \\ c \end{bmatrix}. \tag{5.1}$$

Коммуникационное действие говоруна обозначается «one-hot encoding» вектором $[1; 0; 0]$, или $[0; 1; 0]$, или $[0; 0; 1]$ для обозначения трёх ориентиров соответственно. Физическое действие u слушателя — это 5-канальный вектор, каждый из которых представляет одно направление движения (вверх, вниз, влево, вправо или без движения). Наблюдения и действия говоруна и слушателя и их взаимосвязь показаны на рисунке рис.5.3.

Два агента имеют общую награду r, которая является отрицательным евклидовым расстоянием между слушателем и его целью. Проблема, которую необходимо решить в этом сценарии, заключается в поиске оптимальных политик для говоруна

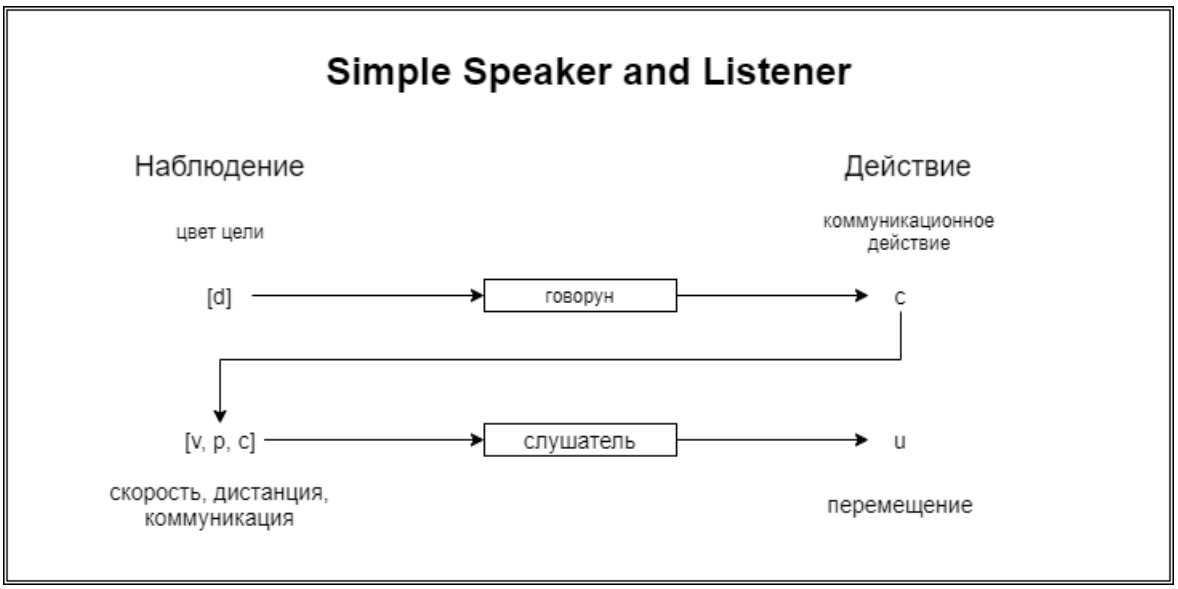

Рис.5.3. Говорун наблюдает цвет целевого ориентира d и издаёт коммуникационое действие, которое будет получено слушателем. Слушатель производит физическое движение

и слушателя, чтобы максимизировать ожидаемую награду, то есть найти $\max_\pi R(\pi)$, где

$$R(\pi) = \mathbb{E}[\sum_{t=0}^{T} r(s_t, a_t)]. \tag{5.2}$$

Во время обучения говорун учится различать три ориентира и передавать целевой ориентир слушателю. И слушатель должен изучить закодированные высказывания говоруна и перейти к правильной цели.

Архитектура актор-сети говоруна аналогична архитектуре слушателя, каждый из которых содержит два полносвязанных слоя с 64 нейронами и использует функцию активации relu. Однако их выходные слои различаются с точки зрения количества единиц. Сети критиков имеют структуру, аналогичную сетям акторов, за исключением того, что они выдают скалярное Q-значение.

На этом сценарии было произведено измерение консистентности коммуникационных действий для алгоритмов DDPG и MADDPG.

5.3.2. Сценарий 2. Simple Reference

Simple Reference — это более сложный сценарий, который расширяет Simple Speaker Listener.

Проблема, которую необходимо решить в этом сценарии, была изложена в разделе Вводная глава: Сценарий 2. Simple Reference. На рис.1.2 показан сценарий игры и поведение агентов при использовании оптимальных политик. В этом

сценарии оба агента одновременно выступают и слушателями, и говорунами, что означает, что они оба выполняют как физические, так и коммуникационные действия. Высказывание каждого агента на одном шаге наблюдается другим агентом на следующем шаге, как показано на рис.5.4.

Каждый агент наблюдает скорость, расстояние до ориентиров, цвет цели другого агента и высказывание другого агента. Действие каждого агента состоит из двух поддействий: физического движения u и коммуникационного действия c.

Наблюдение и действие:

$$o_i = \begin{bmatrix} \upsilon \\ p \\ d \\ c \end{bmatrix}, \qquad a_i = \begin{bmatrix} u \\ c \end{bmatrix}. \tag{5.3}$$

Это показано на рис.5.4.

Наградой каждого агента является среднее значение отрицательных евклидовых расстояний от агентов до их целей. Таким образом, оцениваются как физические, так и коммуникационные действия агентов. Предположим, что r_0 — это отрицательное евклидово расстояние от агента 0 до его цели, а r_1 — это расстояние от агента 1 до его цели. Окончательная награда для каждого агента:

$$r = \frac{r_0 + r_1}{2}. \tag{5.4}$$

Награда r_0 используется для оценки физического движения u_0 агента 0 и коммуникационного действия c_1 агента 1. То же относится и к r_1 - он используется для оценки физического движения u_1 агента 1 и высказывания c_0 агента 0. Это показано на рис.5.4.

Архитектура нейронных сетей актора-критика двух агентов в этом сценарии аналогична архитектуре в предыдущем сценарии.

В этом сценарии был поставлен эксперимент с декомпозированным вознаграждением, а также эксперимент с общим мозгом

Так же на этом сценарии производилось измерение консистентности коммуникационных действий.

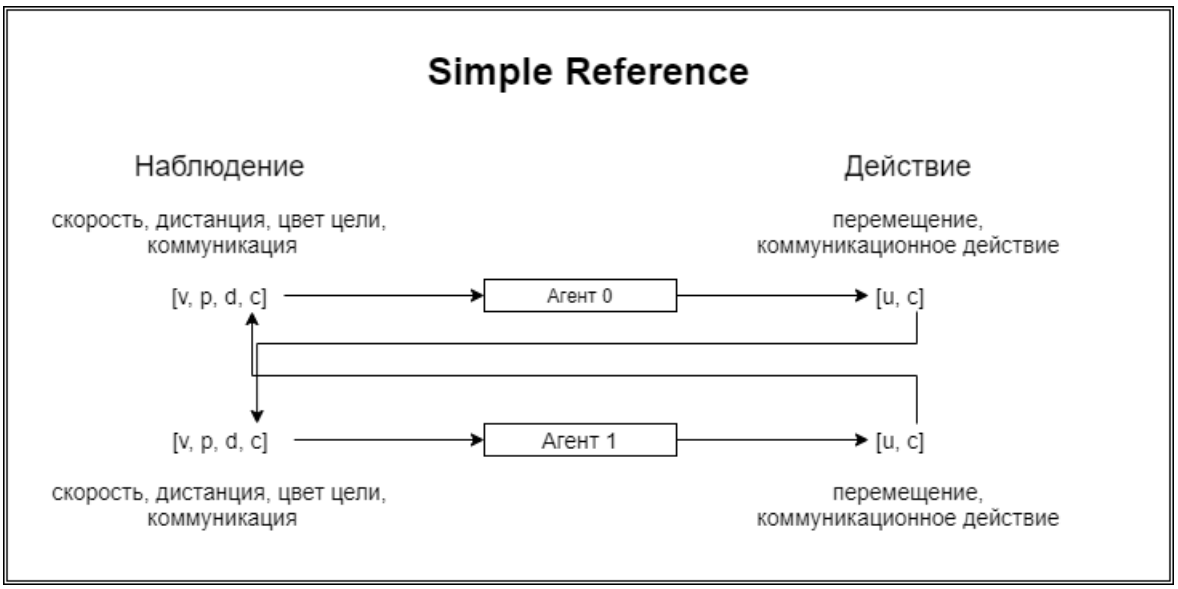

Рис.5.4. Каждый агент наблюдает целевой ориентир другого и производит коммуникационное действие, которое будет получено другим агентом на следующем шаге

5.3.3. Сценарий 3. Simple World Communication

Кратко этот сценарий был описан в разделе Вводная глава: Сценарий 3. Simple World Communication. На рис.1.3 показан сценарий игры и поведение агентов при использовании оптимальных политик. В этом сценарии лидер выступает говоруном, остальные преследователи – слушателями. И те, и другие перемещаются в погоне за хорошими агентами. Лидер выступает говоруном, а преследователи — слушателями. Высказывание лидера на одном шаге наблюдается другими преследователями на следующем шаге, как показано на рис.5.5.

Хорошие агенты просто наблюдают еду и преследователей и перемещаются.

Под едой подразумевается объект, при приближении к которому, хорошие агенты получают награду

Каждый преследователь наблюдает скорость $\upsilon \in \mathbb{R}^2$, расстояние до жертв $p = [p_1, p_2, p_3], p_i \in \mathbb{R}$, и высказывание лидера. Лидер наблюдает то же самое, кроме высказывания. Действие лидера состоит из двух поддействий: физического движения u и коммуникационного действия c. Действие преследователя – только из физического движения u. Наблюдение и действие лидера:

$$o_i = \begin{bmatrix} \upsilon \\ p \end{bmatrix}, \qquad a_i = \begin{bmatrix} u \\ c \end{bmatrix}. \tag{5.5}$$

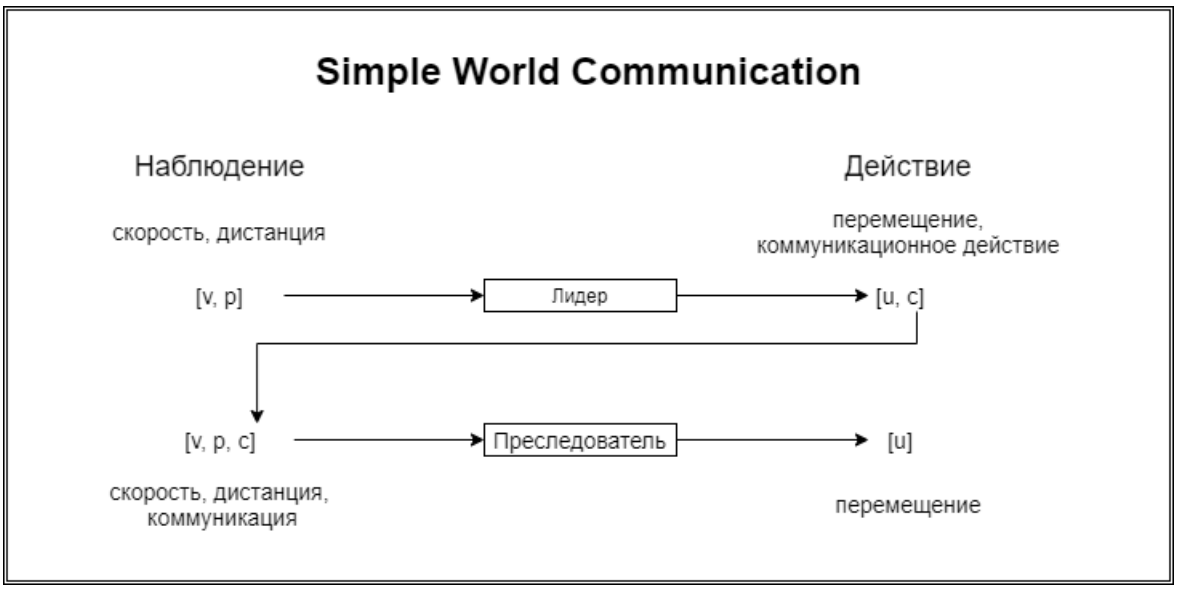

Рис.5.5. Лидер видит то, чего, возможно, не видят другие преследователи, и производит коммуникационное действие, которое будет получено преследователями на следующем шаге

Наблюдение и действие преследователя:

$$o_i = \begin{bmatrix} \upsilon \\ p \\ c \end{bmatrix}, \qquad a_i = \begin{bmatrix} u \end{bmatrix}. \tag{5.6}$$

Это показано на рисунке рис.5.5.

Наградой каждого преследователя является отрицательное евклидово расстояние до ближайшего агента.

Во время обучения лидер учится, как закодировать положение жертвы, чтобы передать его преследователю. И преследователь должен изучить закодированные высказывания лидера и двигаться к жертве.

Архитектура нейронных сетей в этом сценарии аналогична архитектуре в предыдущем сценарии.

На этом сценарии производилось измерение консистентности коммуникационных действий.

5.3.4. Сценарий 4: Simple Tag

Кратко этот сценарий был описан в разделе Вводная глава: Сценарий 4. Simple Tag. На рис.1.4 показан сценарий игры и поведение агентов при использовании оптимальных политик. В этом сценарии нет общения между агентами,

Таблица 5.1

Подходящие сетевые архитектуры для разных игровых сценариев

	MADDPG	MADDPG с одним мозгом	Декомпозированная награда
Speaker Listener	v	—	—
Simple Reference	v	v	v
World Communication	v	—	v
Simple Tag	v	v	—

но пространства наблюдения и действий всех преследователей совпадают, это позволяет применить к нему метод обучения с одним мозгом. В этом сценарии все агенты наблюдают друг друга и перемещаются по игровому полю.

Каждый преследователь наблюдает скорость $\upsilon \in \mathbb{R}^2$, расстояние до жертв $p = [p_1, p_2, p_3], p_i \in \mathbb{R}$.

Наблюдение и действие:

$$o_i = \begin{bmatrix} \upsilon \\ p \end{bmatrix}, \qquad a_i = \begin{bmatrix} u \end{bmatrix}. \tag{5.7}$$

Наградой каждого преследователя является отрицательное евклидово расстояние до ближайшего агента.

Во время обучения преследователи учатся догонять жертву, а жертва —- убегать.

Архитектура нейронных сетей в этом сценарии аналогична архитектуре в предыдущем сценарии.

На этом сценарии ставился эксперимент с одним мозгом.

5.4. Сетевая архитектура

В таблице табл.5.1 показана примененимость различных сетевых архитектур к различным игровым сценариям.

Такие сценарии как Simple Speaker Listener с двумя агентами, разделяющими глобальное вознаграждение, но обладающими разными наблюдениями и пространствами действий, могут использовать только стандартный MADDPG. Поскольку агенты функционируют по-разному, им необходимо искать различные оптимальные политики для своих ролей в игре.

Таблица 5.2

Наличие действий общения в сценариях

	Предполагает общение агентов
Speaker Listener	v
Simple Reference	v
World Communication	v
Simple Tag	—

Такие сценарии как Simple Reference, где два агента совместно получают глобальное вознаграждение, имеют одно и то же пространство действий и пространство наблюдений, могут работать со стандартным MADDPG, а также эти сценарии могут использовать общий мозг или декомпозированную награду.

Наиболее эффективной архитектурой являются использование сети с общим мозгом, поскольку при этом обучается меньшее количество сетей. Это значительно повышает эффективность обучения игровых сценариев, которые можно адаптировать к сетям с общим мозгом.

В сценарии Simple Tag агенты из одной команды также имеют одинаковые пространства наблюдений и действий. Здесь был применён отдельный набор акторов-критиков для преследователей и отдельный - для жертв. Общение в этом сценарии отсутствует, поэтому декомпозированная награда здесь не применима. Этот сценарий был выбран для сравнения стандартного MADDPG и MADDPG с одним мозгом.

В экспериментах со сценариями, в которых агенты общаются между собой (см. табл.5.2), производилось измерение консистентности коммуникационных действий.

Сценарии, в которых одни и те же агенты и перемещаются и говорят, подходят для применения декомпозированой награды. Simple Listener не подходит, т.к. один агент только перемещается, а другой только говорит, а в Simple Tag нет действий общения.

Таким образом, для ответа на вопросы исследования ставились эксперименты, показывающие общение агентов между собой, а также эксперименты, показывающие эффективность некоторых модификаций алгоритма, направленных на ускорение обучения. Исходя из этого и выбирались сценарии для экспериментов данной работы.

5.5. Выводы

В этой главе были подробно рассмотрены сценарии и эксперименты, которые над ними ставились. В следующей главе будут собраны результаты экспериментов, а также выводы.

ГЛАВА 6. РЕЗУЛЬТАТЫ И ОБСУЖДЕНИЯ

6.1. Ответы на вопросы исследования

Были получены ответы на вопросы исследования, поставленные в разделе Вопросы исследования.

1. Как несколько агентов могут научиться сотрудничать друг с другом во время обучения в определённых игровых сценариях?

Ответ: во время обучения для изучения оптимальной политики сотрудничества агенты должны принимать во внимание совместные наблюдения и совместные действия всех агентов.

2. Может ли после обучения появиться язык между агентами в определённых игровых сценариях?

Ответ: да, когда обучение сходится, композиционный язык у агентов возникает.

3. Как можно оптимизировать и ускорить процесс обучения?

Ответ: для ускорения обучения существуют разные методы и приёмы. В этой работе применялись некоторые из них, такие как: обучение с общим мозгом, декомпозированное вознаграждение, обучение по учебной программе.

6.2. Сценарии

Первые два сценария не предполагают конкуренции и не требуют большого количества шагов для достижения результата, поэтому их эпизоды ограничены 25 шагами.

В экспериментах со вторыми двумя сценариями было решено дать агентам возможность двигаться подольше, и длительность эпизодов была ограничена 150 шагами. Это замедлило обучение, но иначе невозможно обучить агентам сколько-нибудь сложным тактикам.

Во всех экспериментах проигрывалось 20000 эпизодов.

При анализе измерении времени работы алгоритмов использовалось распределение Стьюдента, поскольку оно больше подходит для небольших выборок. Эксперименты занимают довольно большое количество времени, поэтому удалось провести лишь по 10 экспериментов.

6.2.1. Сценарий 1. Simple Speaker Listener

Сходимость этого сценария становится возможной, когда говорун сообщает правильные целевые ориентиры и слушатель достигает их. В проведённых экспериментах агенты обучались с помощью алгоритмов MADDPG и DDPG.

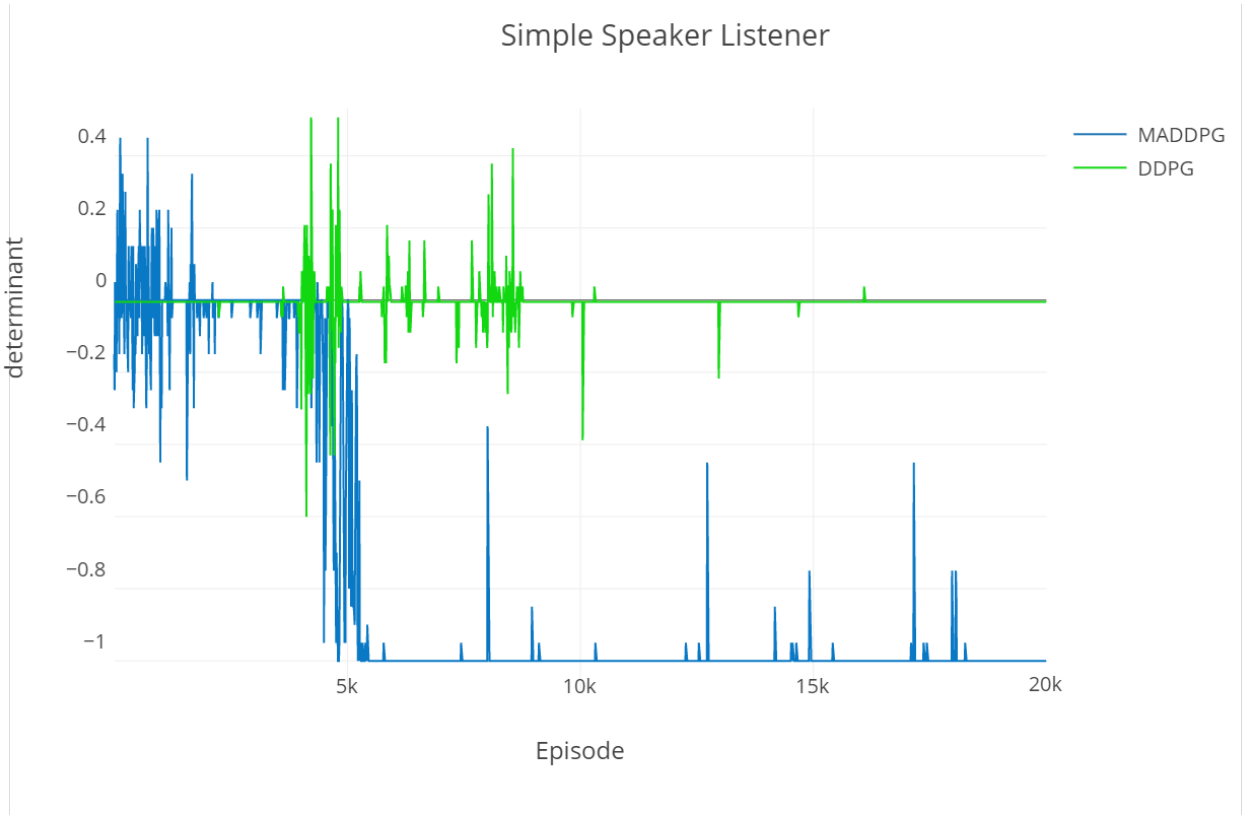

Рис.6.1. График согласованности коммуникаций говоруна в сценарии Simple Speaker Listener

Были построены графики согласованности коммуникаций и вознаграждений для двух агентов в сценарии Simple Speaker Listener, которые представлены на рис.6.1 и рис.6.2. На этих графиках синяя кривая — это результат обучения MADDPG, а зелёная — DDPG.

График на рис.6.1 показывает сходимость консистентности действий общения для говоруна, как описано в разделе Сценарии. Этот график показывает, что сходится он только для MADDPG. Это указывает на то, что говорун, обученный MADDPG, может постоянно интерпретировать ориентир в одной и той же кодировке, в то время как говорун, обученный DDPG, не может поддерживать консистентное общение.

График на рис.6.2 показывает среднее вознаграждение, которое агенты получили в конце каждого эпизода. Из этого графика видно, что вознаграждение, полученное агентами, обученными DDPG, ниже, чем агентами, обученными MADDPG.

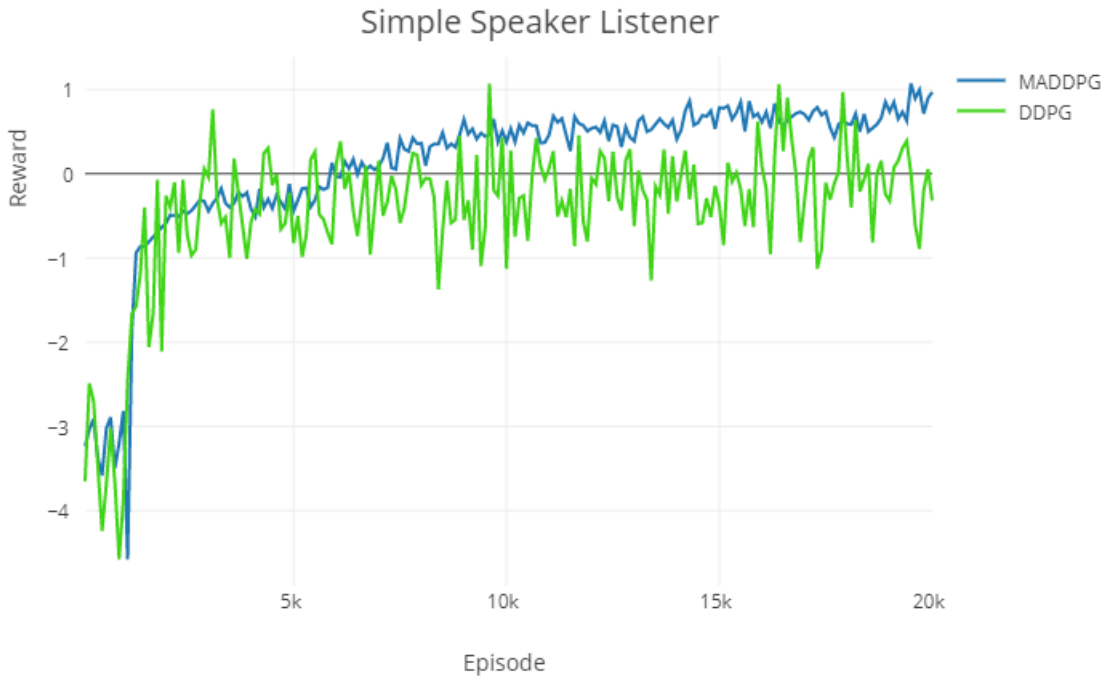

Рис.6.2. График среднего вознаграждения для двух агентов в сценарии Simple Speaker Listener. Результаты обучения по алгоритму MADDPG и DDPG

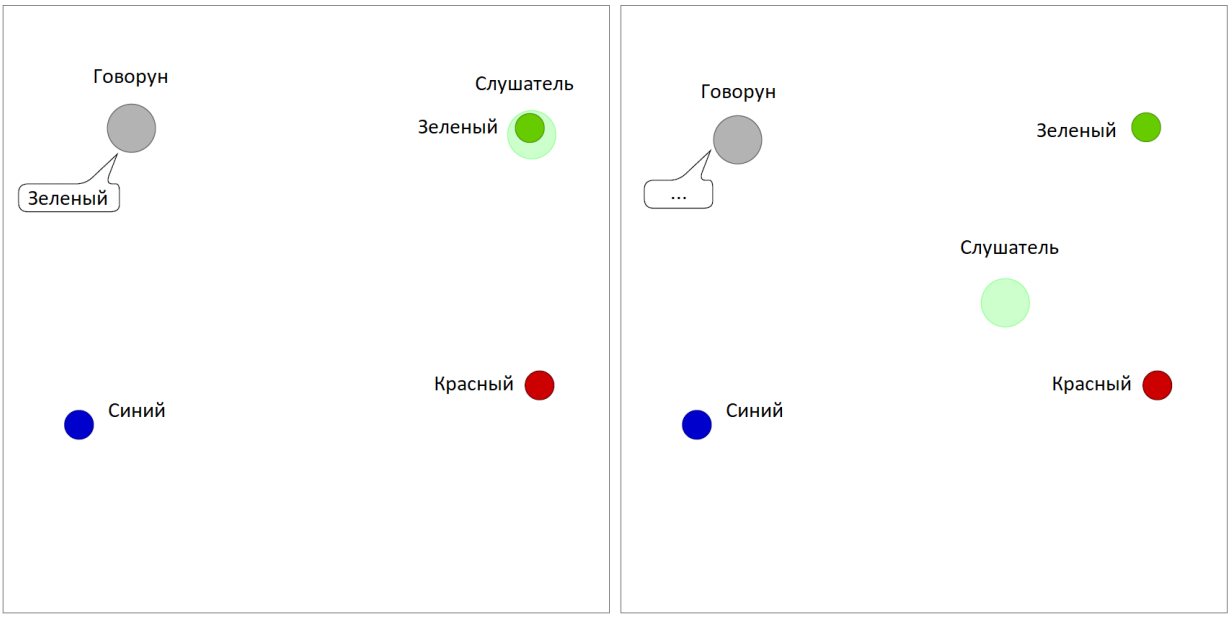

Рис.6.3. На левой стороне: говорун произносит корректное высказывание, слушатель перемещается к цели. Справа: коммуникационное действие отсутствует, и слушатель застревает между тремя ориентирами (видимо, в попытке минимизировать расстояние до каждого из них)

Агенты, обученные DDPG, предпринимают действия, не имея полного наблюдения за состоянием окружающей среды и политикой других агентов. Они не в состоянии вычислить политику оптимального взаимодействия друг с другом.

На рис.6.3 показан скриншот при сходимости и отсутствии сходимости Simple Speaker Listener.

6.2.2. Сценарий 2. Simple Reference

Сценарий Simple Reference сходится, когда агенты правильно перемещаются к своим собственным целевым ориентирам. На рис.1.2 показан скриншот поведения агентов после схождения сценария Simple Reference. В проведённых экспериментах агенты тренировались алгоритмами DDPG и MADDPG.

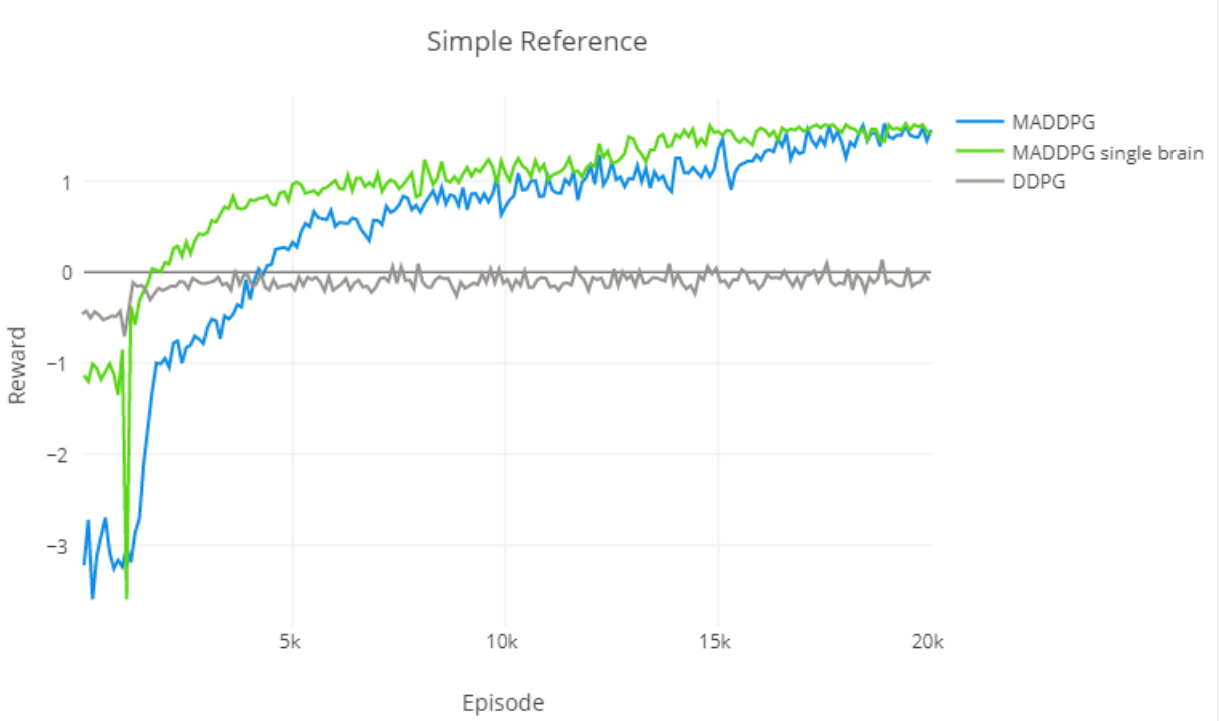

Рис.6.4. График среднего вознаграждения для двух агентов в сценарии Simple Reference. Результаты обучения по алгоритму MADDPG, MADDPG с одним мозгом и DDPG

Был построен график вознаграждения для двух агентов, который представлен на рис.6.4. На этом графике синяя кривая — результат, обученный MADDPG, зелёная — одним мозгом, а серая — DDPG.

Агенты, обученные с помощью DDPG, получают меньшее вознаграждение, чем агенты MADDPG и его варианты. Также из рендеринга игры видно, что агенты DDPG блуждают среди ориентиров, не зная, какой из них является правильной целью.

Графики на рис.6.5 показывают согласованность действий общения. Для MADDPG и его варианта средний детерминант стремится к 1 или -1. Это указывает на то, что агенты, обученные с помощью этих алгоритмов, могут общаться согласованно. Для агентов, обученных DDPG, значение стремится к 0, это иллюстрируют, что они не могут корректно сообщать цели.

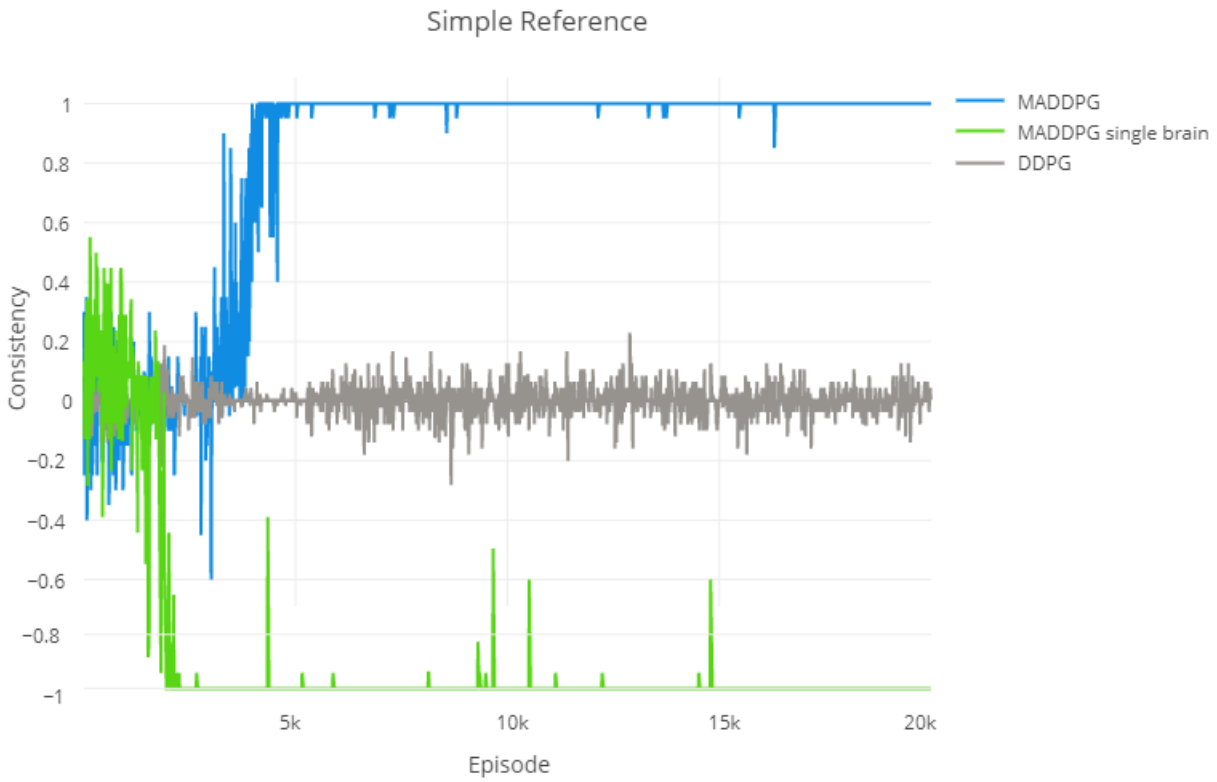

Рис.6.5. Графики согласованности взаимодействия для двух агентов в сценарии Simple Reference. Результаты обучения по алгоритму MADDPG, MADDPG с одним мозгом и DDPG

Таблица 6.1

Среднее время, потраченное на обучение с различными алгоритмами в 20000 эпизодах

	MADDPG	MADDPG с одним мозгом
Время, 20000 эпизодов	820 ± 98 с	635 ± 45 с

В процессе выполнения многочисленных экспериментов можно было наблюдать, что полная несходимость всегда сопровождается тем, что агенты, не способны дифференцировать и сообщать правильные ориентиры друг другу. То же справедливо и для частичной сходимости, когда агенты могут перемещаться только к тем ориентирам, которые правильно сообщены другими агентами, см. рис.6.6. Это также показывает, что агентам для достижения целей необходимо сотрудничать как в физических, так и в коммуникационных действиях.

Наконец, между агентами возникает язык. Проведённые эксперименты показали, что при каждой тренировке агенты по-разному интерпретируют ориентиры. Например, после одной тренировки красный ориентир может выглядеть как $[1; 0; 0]$, а после другой — как $[0; 1; 0]$ или $[0; 0; 1]$.

В табл.6.1 показано время, затраченное на обучение по сценарию Simple Reference с разными алгоритмами.

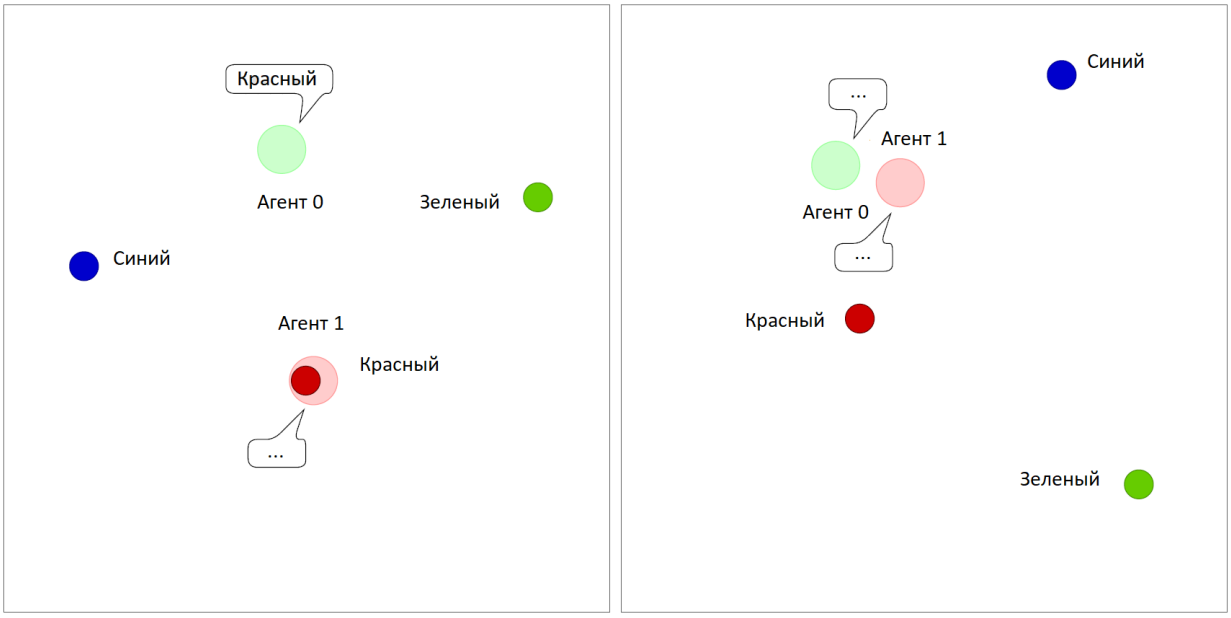

Рис.6.6. Частичная сходимость с левой стороны: один агент перемещается к цели, а другой ждёт между ориентирами. Несходимость на правой стороне заканчивается тем, что оба агента не знают, куда двигаться и ждут между ориентирами

6.2.3. Сценарий 3: Simple World Communication

Были построены графики вознаграждений и согласованности коммуникаций для двух преследователей и одной жертвы в сценарии Simple World Communication, которые представлены на рис.6.7 и рис.6.8. На этих графиках зелёная кривая — это результат обучения MADDPG, а серая — DDPG.

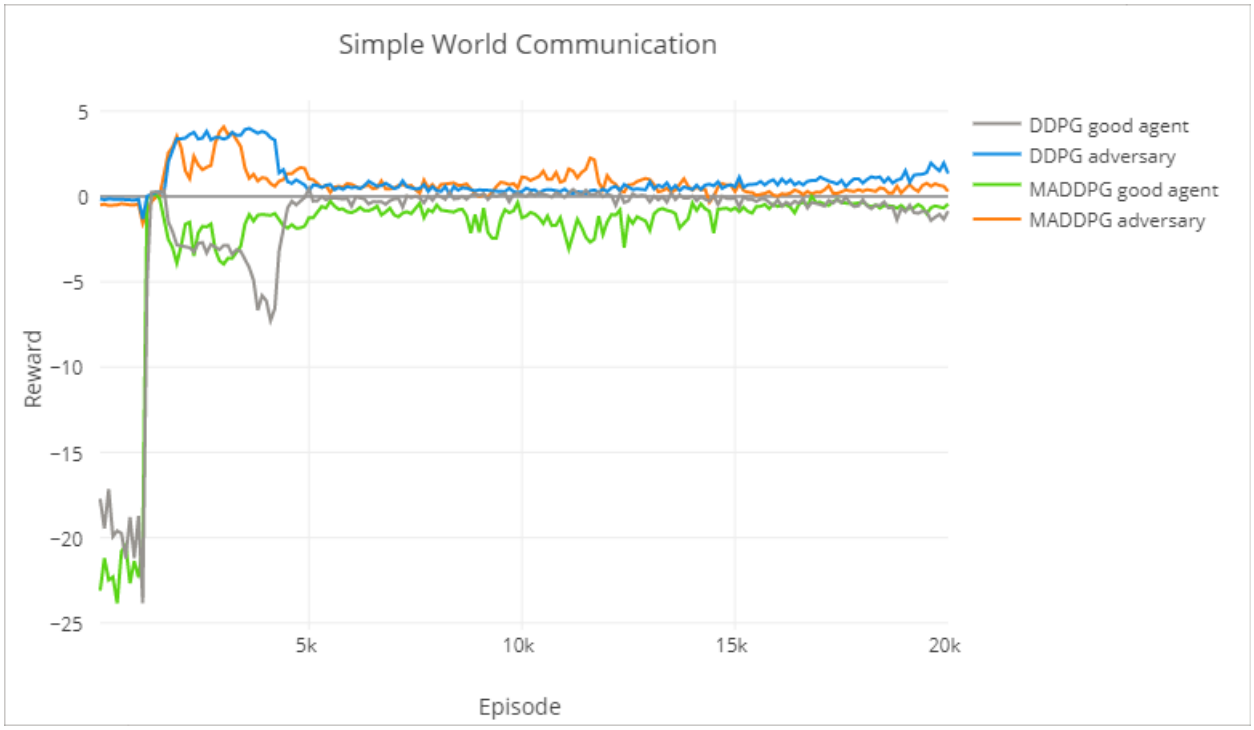

Рис.6.7. Среднее вознаграждение преследователей и жертвы с алгоритмами DDPG и MADDPG

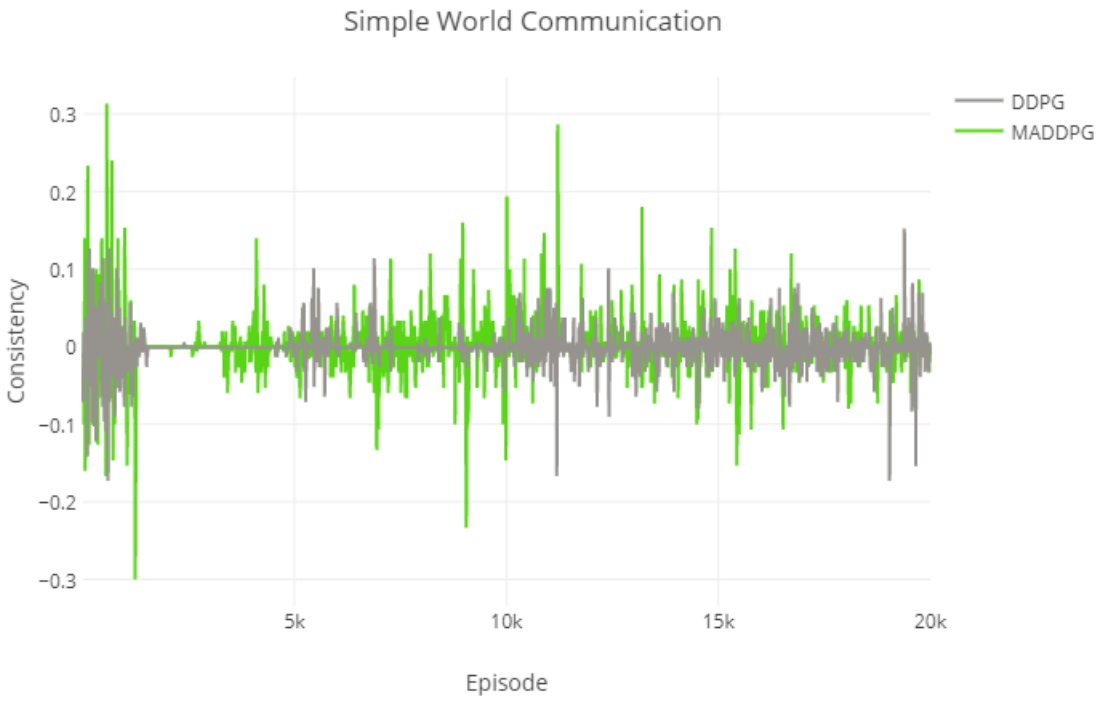

Рис.6.8. График консистентности действий коммуникации для алгоритмов DDPG и MADDPG

К сожалению, из-за ограниченности времени и вычислительных ресурсов нам не удалось добиться сходимости консистентности действий общения в этом сценарии, это видно на рис.6.8. Из графика видно, что отклонения от нуля больше в случае MADDPG, но этого оказалось недостаточно для того, чтобы «закрепить результат», научиться преследователю использовать получаемые сигналы лидера так, чтобы улучшить общее вознаграждение.

Вследствие чего результаты алгоритма MADDPG ничем не выделяются по сравнению с DDPG, это можно видеть на рис.6.7.

Также в этом сценарии трудно установить «конец игры», когда цель достигнута, так как во время обучения агенты обеих команд постепенно действуют всё более адекватно, однако это трудно увидеть на графике среднего вознаграждения. То преследователи изобретают более-менее эффективную тактику, и получают большую награду, то жертва обучается на своих ошибках и начинает учитывать это. Однако, на рендеринге можно видеть, что и преследователи, и жертва ведут себя вполне адекватно. Преследователи догоняют жертву, жертва убегает и, по возможности, стремится к еде.

Скорее всего, если обучать модель дольше, можно было бы увидеть более сложные и согласованные действия преследователей.

Таблица 6.2

Среднее время, потраченное на обучение с различными алгоритмами в 20000 эпизодах: 2 преследователя, 1 жертва

	DDPG	MADDPG	MADDPG с одним мозгом
Время, 20000 эпизодов	27960 ± 2236 с	37430 ± 2994 с	13320 ± 1065 с

6.2.4. Сценарий 4: Simple Tag

Как уже упоминалось выше, этот сценарий не подразумевает коммуникационных действий. Но он был выбран нами, так как он достаточно сложный, и при этом агенты из одной команды имеют одинаковые пространства наблюдения и действий, что позволяет применить вариант алгоритма MADDPG с общим мозгом.

6.2.4.1. Двое против одного

Был поставлен эксперимент с двумя преследователями и одной жертвой.

На рис.6.9a и рис.6.9b изображены графики средней награды преследователей и жертв. Видно, что графики симметричны. Когда преследователи выучиваются двигаться в нужном направлении и понимают, что нужно преследовать жертву, они получают большую награду, а жертва — меньшую. После чего жертва выучивается убегать от преследователей — тогда её награда растёт, а награда преследователей снижается. Так же, как и в предыдущем сценарии.

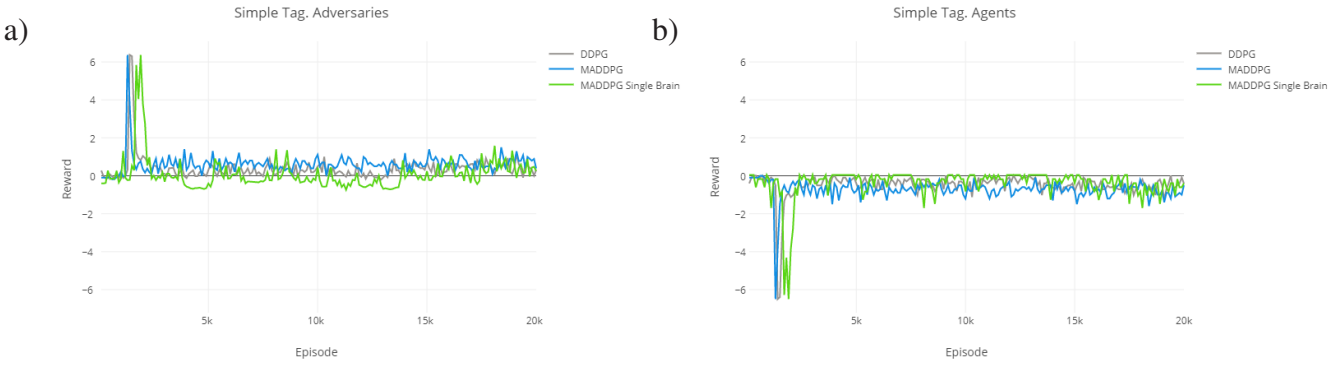

Рис.6.9. Награда с применением алгоритма DDPG, MADDPG, MADDPG с общим мозгом: *a* — преследователей; *b* — жертв

В табл.6.2 указано время, затраченное на обучение в данной конфигурации для разных алгоритмов.

Таблица 6.3

Среднее время, потраченное на обучение с различными алгоритмами в 20000 эпизодах: 4 преследователя, 2 жертвы

	DDPG	MADDPG	MADDPG с одним мозгом
Время, 20000 эпизодов	40980 ± 3278 с	60480 ± 4838 с	31100 ± 2488 с

6.2.4.2. Четверо против двоих.

С этим сценарием также были поставлены эксперименты с более сложными настройками — 4 преследователя и 2 жертвы.

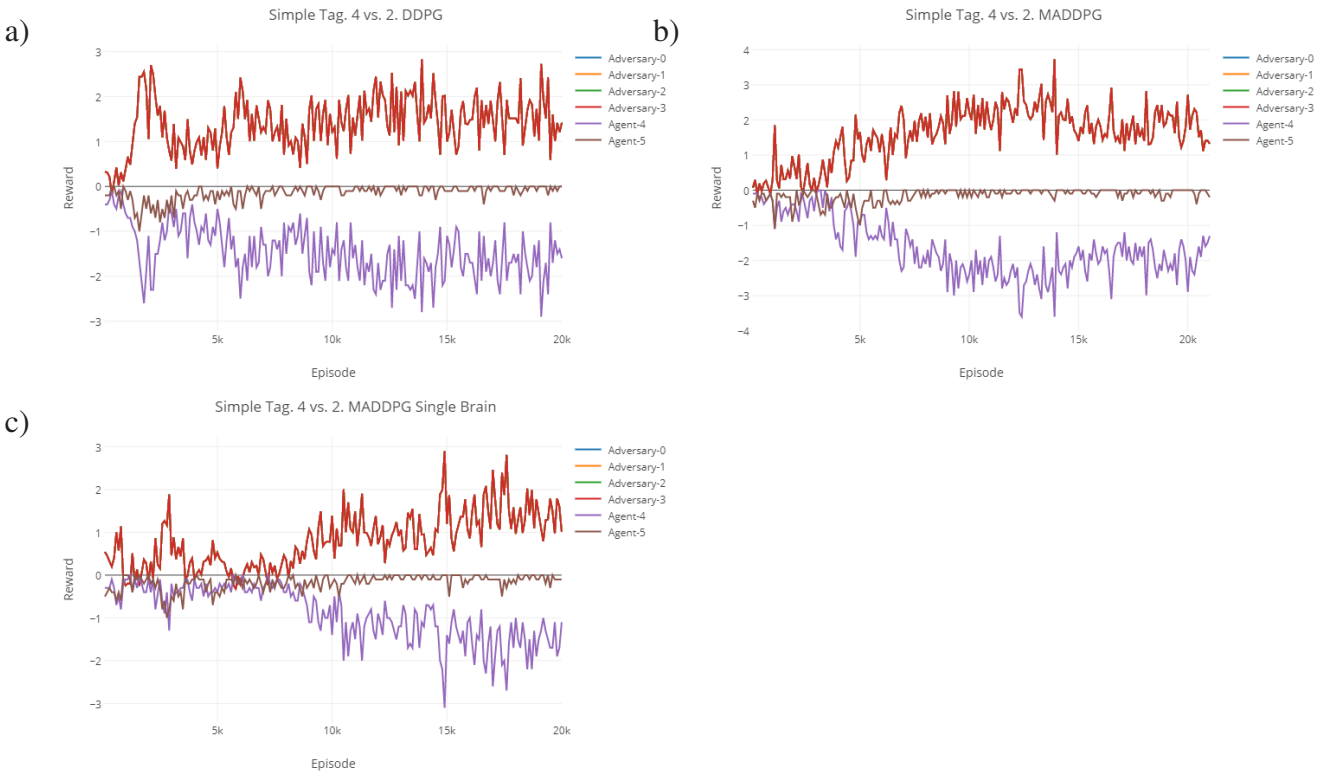

Рис.6.10. Награда каждого из 6 агентов с применением алгоритма: *a* — DDPG; *b* — MADDPG; *c* — MADDPG с общим мозгом

На рис.6.10a, рис.6.10b и рис.6.10c изображены графики средней награды преследователей и жертв.

В табл.6.3 указано время, затраченное на обучение в данной конфигурации для разных алгоритмов.

6.2.4.3. Вывод из результатов эксперимента.

Как уже упоминалось выше, награды преследователей и жертв симметрично-противоположны. Это вытекает из того, что: во-первых — сам сценарий имеет конкурентную природу, а награда одним агентам даётся за то же, за что у

других отнимается (это описано в разделе Вводная глава: Сценарий 4. Simple Tag); во-вторых — во время тренировки обучаются как преследователи, так и жертвы.

Если взять обученную модель агента одной из команд и начать тренировать против него агентов другой команды, но при этом не обучать первого агента, можно было бы увидеть некоторой прогресс, увидеть на графике, как награда первого агента со временем снижается, а его противников — растёт.

Приведённые выше графики наград отчётливо это показывают. В ходе обучения они ни к чему не сходятся.

Однако, это не значит, что агенты не обучаются. При запуске игры на обученных моделях видно, что агенты из обеих команд ведут себя вполне адекватно — преследователи гоняются за жертвами, жертвы избегают преследователей. Как на правом скриншоте на рис.1.4.

И это не помешало нам измерить затраченное время на выполнение каждого эксперимента.

Из табл.6.2 и табл.6.3 видно, что алгоритм DDPG показал лучшие результаты, чем MADDPG. Это объясняется тем, что этот алгоритм проще и требует меньших вычислительных мощностей, в то же время, он не даёт тех возможностей коммуникации, которые могут быть необходимы во многих задачах.

Зато алгоритм MADDPG с общим мозгом показал значительное преимущество. Это объясняется тем, что в этом случае не происходит обучения одному и тому же разных наборов сетей актора-критика. Обучается один «мозг» для каждой команды агентов.

6.3. Выводы и замечания

Алгоритм MADDPG считается довольно капризным и требует тонкой настройки. Во время проведения экспериментов алгоритм сходился далеко не всегда, а некоторые эксперименты так и не увенчались успехом.

Обучение сходится с алгоритмом MADDPG, а также с вариантом с одним мозгом. К сожалению, при выполнении эксперимента с учебной программой не удаётся применить модель, обученную на простых настройках к более сложным. Модель, хорошо выступающая на простом уровне, расходится после повышения уровня сложности. Причиной является то, что при усложнении настроек среды растёт и размерность вектора наблюдений. А также, возможно, то, что обучение модели происходит не сразу, а после наполнения буфера.

Эксперимент с декомпозированной наградой тоже не увенчался успехом, вероятно, потому что награда поступает из среды за общее действие, а не за отдельные поддействия. Возможно, если обучать модель значительно дольше, можно было бы увидеть более хорошие результаты.

ЗАКЛЮЧЕНИЕ

Как это было сказано во введении, целью данной работы является разработка и исследование алгоритмов управления MAC с применением технологий глубокого обучения с подкреплением.

Во время работы над этой выпускной работой были решены поставленные задачи: был произведён обзор технологии глубокого обучения с подкреплением, а также методов и алгоритмов управления MAC; были поставлены вопросы исследования, а затем выбраны и доработаны алгоритмы и игровые сценарии, которые позволяют дать ответы на эти вопросы; поставлены эксперименты и сделаны выводы.

В ходе исследования был сделан вывод, что лучше всего для управления в мультиагентной среде подходит алгоритм MADDPG. Сам алгоритм хорошо описан [25]. В этой же работе было решено исследовать, могут ли агенты сотрудничать, а также общение агентов между собой. Как упоминалось в разделе 3.2 Мультиагентные алгоритмы, многие задачи из реального мира лучше решаются многоагентными алгоритмами; для достижения оптимального результата действия агентов должны быть скоординированы.

С другой стороны, чем больше агентов и чем сложнее среда, тем больше ресурсов требуется на обучение. Поэтому следующий вопрос был посвящён методам оптимизации обучения.

Данная работа способствует дальнейшему пониманию совместной работы множества агентов в игровой среде. Проведённые эксперименты доказывают, что агенты должны вырабатывать оптимальную политику сотрудничества с учётом полного наблюдения за состоянием окружающей среды и политиками других агентов. Очевидно, что в результате обучения возникает композиционный язык, который улучшает сотрудничество. В данной работе алгоритм действует в определённых сценариях, но к сожалению, не удалось реализовать все задуманные приёмы, и не все реализованные варианты алгоритма удалось заставить работать, вероятно, из-за нехватки времени и ресурсов.

По сравнению с предыдущими работами эта дипломная работа расширяет MADDPG до более сложного сценария, когда агентам необходимо сотрудничать в многомерных пространствах действий. Вкратце, агенты одновременно выполняют физическое движение и коммуникационное высказывание.

Некоторые варианты MADDPG исследуются для разных игровых сценариев.

MADDPG с разложенным вознаграждением может быть адаптирован к сценариям со сложным глобальным вознаграждением.

MADDPG с одним мозгом может применяться, если агенты в сценарии симметричны в пространстве наблюдения и действий и имеют одинаковое глобальное вознаграждение. Кроме того, агенты, обученные MADDPG с одним мозгом, говорят на одном языке, что важно для сценариев с более чем двумя агентами, которые должны общаться между собой для достижения общей цели.

Как уже упоминалось выше, из-за конкурентной природы сценариев Simple World Communication и Simple Tag нам не удалось добиться того, чтобы награда агентов из разных команд стремилась к какому-то значению или постоянно росла. Эти сценарии были выбраны для того, чтобы исследовать вопросы, поставленные нами в начале данной работы.

Зато эксперименты с этими сценариями показали эффективность алгоритма MADDPG с общим мозгом.

Опишем возможную дальнейшую работу в продолжение данного исследования.

Алгоритм MADDPG считается довольно капризным и требующим тонкой настройки. Возможно поэтому, а также из-за недостатка времени и вычислительных ресурсов, не увенчались успехом эксперименты по некоторым методам оптимизации обучения: декомпозированное вознаграждение и обучение по учебной программе.

Следующие шаги в данном исследовании могут быть связаны с тем, чтобы всё-таки реализовать эти методы и поставить соответствующие эксперименты.

Затем следует исследовать сотрудничество большего количества агентов. Они могут обладать более многомерным пространством действий. Агенты должны говорить на одном языке, если необходимо общение между более чем двумя агентами. Следовательно, алгоритмы и сети необходимо модифицировать и адаптировать с учётом новых ситуаций. Приведённые выше выводы в этой дипломной работе могут способствовать дальнейшим исследованиям.

Следующее усложнение — это использование алгоритма детерминированного градиента политики для задач с несколькими агентами в сценариях трёхмерных игр. Основной проблемой для алгоритма в таких сценариях может быть высокоразмерный ввод пикселей. Дальнейшая работа должна быть сделана соответственно над структурой нейронных сетей, например, сверточных слоёв в акторе и настройке гиперпараметров.

Все эти этапы — это шаги с постепенным усложнением задач перед попыткой применения уже отработанных алгоритмов к управлению роботами и их совместной командной работе.

Основные сложности применения алгоритмов обучения с подкреплением к управлению роботами заключаются в следующем:

- очень высокая размерность наблюдения;
- высокая степень неопределённости (никогда нельзя быть до конца уверенным, как поменяется среда после очередного действия);
- дороговизна и длительность тренировки модели в реальной среде (невозможно сократить время тренировки за счёт вычислительных мощностей), а зачастую невозможность такой тренировки (в процессе обучения неизбежны ошибки; например автомобиль, управляемый программой, может представлять опасность).

Мы считаем, что из всего вышесказанного следует, что первые этапы изучения следует производить в симуляторах естественной среды.

И наконец, следующим шагом была бы апробация применения алгоритм к обучению нескольких роботов для достижения общей цели в реальной среде. Например, сбор мусора, различные задачи на преследование и т.д.

Напоследок, хотелось бы выразить бладодарность за помощь в написании данной работы моему научному руководителю, доценту ВШИСиСТ Паку Вадиму Геннадьевичу, а также научному сотруднику СПИИРАН Клеверову Денису Анатольевичу за помощь в технических вопросах и за рецензию. Без этих людей этой работы не было бы.

СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ

МАС Мультиагентная система.

МО Машинное обучение (Machine Learning).

ИИ Искусственный интеллект (Artificial Intelligence, AI).

DL Глубокое обучение (Deep Learning).

RL Обучение с подкреплением (Reinforcement Learning).

DRL Глубокое обучение с подкреплением (Deep Reinforcement Learning).

DPG Детерминированный градиент политики (Deterministic Policy Gradient).

DDPG Глубокий детерминированный градиент политики (Deep Deterministic Policy Gradient).

MADDPG Мультиагентный глубокий детерминированный градиент политики (Multiagent Deep Deterministic Policy Gradient).

COMA Контрафактный мультиагентный градиент политики (Counterfactual Multi-Agent Policy Gradient, COMA).

ИНС Искусственная нейронная сеть (Artificial Neural Network).

МППР Марковский процесс принятия решений (Markov Decision Process, MDP).

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. AlphaGo. — URL: https://en.wikipedia.org/wiki/AlphaGo (дата обращения: 01.08.2020).

2. An Introduction to Statistical Learning: with Applications in R / G. James [и др.]. — Springer New York, 2014. — (Сер.: Springer Texts in Statistics). — URL: https://books.google.se/books?id=at1bmAEACAAJ.

3. Automated Curriculum Learning for Neural Networks / A. Graves [и др.]. — 2017. — arXiv: 1704.03003 [cs.NE].

4. *Bengio Y.*, *Courville A.*, *Vincent P.* Representation Learning: A Review and New Perspectives. — 2012. — arXiv: 1206.5538 [cs.LG].

5. *Bishop C.* Pattern Recognition and Machine Learning. — Springer, 2006. — 738 с. — URL: http://research.microsoft.com/en-us/um/people/cmbishop/prml.

6. Continuous control with deep reinforcement learning / T. P. Lillicrap [и др.]. — 2019. — arXiv: 1509.02971 [cs.LG].

7. Counterfactual Multi-Agent Policy Gradients / J. Foerster [и др.]. — 2017. — arXiv: 1705.08926 [cs.AI].

8. *Crevier D.* AI: The Tumultuous History of the Search for Artificial Intelligence. — 1993. — с. 1—386. — URL: https://www.researchgate.net/publication/233820788_AI_The_Tumultuous_History_of_the_Search_for_Artificial_Intelligence.

9. Curriculum Learning / Y. Bengio [и др.] // Proceedings of the 26th Annual International Conference on Machine Learning. — Montreal, Quebec, Canada: Association for Computing Machinery, 2009. — с. 41—48. — (Сер.: ICML '09). — DOI 10.1145/1553374.1553380. — URL: https://doi.org/10.1145/1553374.1553380.

10. Deep Reinforcement Learning: A Brief Survey / K. Arulkumaran [и др.] // IEEE Signal Processing Magazine. — 2017. — т. 34, № 6. — с. 26—38. — DOI 10.1109/msp.2017.2743240. — URL: http://dx.doi.org/10.1109/MSP.2017.2743240.

11. *Dimitri P. Bertsekas J. N. T.* Neuro-dynamic Programming. — Athena Scientific, 1996. — (Сер.: Anthropological Field Studies). — URL: https://books.google.ru/books?id=WxCCQgAACAAJ.

12. *Edwards C.* Growing Pains for Deep Learning // Commun. ACM. — New York, NY, USA, 2015. — т. 58, № 7. — с. 14—16. — DOI 10.1145/2771283. — URL: https://doi.org/10.1145/2771283.

13. github: openai - multiagent-particle-envs. — URL: https://github.com/openai/multiagent-particle-envs (дата обращения: 01.08.2020).

14. *Goodfellow I.*, *Bengio Y.*, *Courville A.* Deep Learning. — MIT Press, 2016. — (http://www.deeplearningbook.org).

15. *Hastie T.*, *Tibshirani R.*, *Friedman J.* The Elements of Statistical Learning: Data Mining, Inference, and Prediction. — Springer, 2001. — (Сер.: Springer series in statistics). — URL: https://books.google.ru/books?id=VRzITwgNV2UC.

16. *Hornik K.* Approximation capabilities of multilayer feedforward networks // Neural Networks. — 1991. — т. 4, № 2. — с. 251—257. — DOI https://doi.org/10.1016/0893-6080(91)90009-T. — URL: http://www.sciencedirect.com/science/article/pii/089360809190009T.

17. *Huang S. H.*, *Hong-Chao Zhang*. Artificial neural networks in manufacturing: concepts, applications, and perspectives // IEEE Transactions on Components, Packaging, and Manufacturing Technology: Part A. — 1994. — т. 17, № 2. — с. 212—228.

18. Human-level control through deep reinforcement learning / V. Mnih [и др.] // Nature. — 2015. — т. 518. — с. 529—33. — DOI 10.1038/nature14236. — (дата обращения: 01.08.2020).

19. Hybrid Reward Architecture for Reinforcement Learning / H. van Seijen [и др.]. — 2017. — arXiv: 1706.04208 [cs.LG].

20. *Kanter J. M.*, *Veeramachaneni K.* Deep feature synthesis: Towards automating data science endeavors // 2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA). — 2015. — с. 1—10.

21. *Khajanchi A.* Artificial Neural Networks : The next intelligence //. — 2003. — URL: https://www.semanticscholar.org/paper/Artificial-Neural-Networks-%3A-The-next-intelligence-Khajanchi/312a65e33ebba4cbff154a79f57e2a0ff386e6f6.

22. *Lin L.* Reinforcement learning for robots using neural networks //. — 1992.

23. *Mitchell T.* Machine Learning. — McGraw-Hill, 1997. — (Сер.: McGraw-Hill International Editions). — URL: https://books.google.ru/books?id=EoYBngEACAAJ.

24. *Mordatch I.*, *Abbeel P.* Emergence of Grounded Compositional Language in Multi-Agent Populations. — 2019. — arXiv: 1703.04908 [cs.AI].

25. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments / R. Lowe [и др.]. — 2020. — arXiv: 1706.02275 [cs.LG].

26. *Narvekar S.* Curriculum Learning in Reinforcement Learning // Proceedings of the 26th International Joint Conference on Artificial Intelligence. — Melbourne, Australia: AAAI Press, 2017. — с. 5195—5196. — (Сер.: IJCAI'17).

27. OpenAI Gym. — URL: https://github.com/openai/gym (дата обращения: 01.08.2020).

28. *Otterlo M. van*, *Wiering M. A.* Reinforcement Learning and Markov Decision Processes // Reinforcement Learning. — 2012.

29. *Russell S.*, *Norvig P.* Artificial Intelligence: A Modern Approach. — 3rd. — USA: Prentice Hall Press, 2009. — с. 1—5.

30. *Samuel A. L.* Some Studies in Machine Learning Using the Game of Checkers // IBM Journal of Research and Development. — 1959. — т. 3, № 3. — с. 210—229. — URL: http://dx.doi.org/10.1147/rd.33.0210.

31. *Sultan K.*, *Ali H.*, *Zhang Z.* Big Data Perspective and Challenges in Next Generation Networks // Future Internet. — 2018. — т. 10, № 7. — с. 56. — DOI 10.3390/fi10070056. — URL: http://dx.doi.org/10.3390/fi10070056.

32. *Sutton R. S.*, *Barto A. G.* Reinforcement Learning: An Introduction. — 2-е изд. — The MIT Press, 2008. — с. 2. — URL: https://books.google.ru/books?id=VRzITwgNV2UC.

33. *Sutton R. S.*, *Barto A. G.* Reinforcement Learning: An Introduction. — 2-е изд. — The MIT Press, 2008. — с. 50. — URL: https://books.google.ru/books?id=VRzITwgNV2UC.

34. *Sutton R. S.*, *Barto A. G.* Reinforcement Learning: An Introduction. — 2-е изд. — The MIT Press, 2008. — с. 107—108. — URL: https://books.google.ru/books?id=VRzITwgNV2UC.

35. *Szabó Z. G.* Compositionality // Stanford Encyclopedia of Philosophy. — 2008.

36. *Tavakoli A.*, *Pardo F.*, *Kormushev P.* Action Branching Architectures for Deep Reinforcement Learning. — 2019. — arXiv: 1711.08946 [cs.LG].

37. The Arcade Learning Environment: An Evaluation Platform for General Agents / M. G. Bellemare [и др.] // Journal of Artificial Intelligence Research. — 2013. — т. 47. — с. 253—279. — DOI 10.1613/jair.3912.

38. Watson (computer). — URL: https://en.wikipedia.org/wiki/Watson_(computer) (дата обращения: 01.08.2020).

Приложение 1

Псевдокод алгоритма DDPG

Случайным образом инициализируется сеть критика $Q(s, a|\theta^{\mu})$ с весами θ^{Q} и θ^{μ}
Инициализируется целевая сеть Q' и μ' весами $\theta^{Q'} \leftarrow \theta^{Q}$, $\theta^{\mu`'} \leftarrow \theta^{\mu}$
Инициализируется реплей-буффер R
for episode = 1, M **do**
Инициализируется рандомный процесс $\mathcal{N}$ для исследования
Получаются начальные наблюдаемые состояния s_l
for t = 1, T **do**
Выбирается действие $a_t = \mu(s_t|\theta^{\mu}) + \mathcal{N}_t$ исходя из текущей политики и исследовательского шума
Выполняется действие a_t и наблюдается награда r_t и новое состояние s_{t+1}
Сохраняется переход (s_t, a_t, r_t, s_{t+1}) в R
Семплируется мини-набор из N переходов (s_i, a_i, r_i, s_{i+1}) из R
Задаётся $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
Обновляется сеть критика минимизацией функции потерь: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^{Q}))^2$
Обновляется политика актора с использованием семплированного градиента политики:

$$\nabla_{\theta^{\mu}} \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^{Q})|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^{\mu}} \mu(s|\theta^{\mu})|_{s_i} \qquad (П1.1)$$

Обновляются целевые сети:

$$\begin{aligned} \theta^{Q'} &\leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'} \\ \theta^{\mu'} &\leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'} \end{aligned} \qquad (П1.2)$$

end for
end for

Приложение 2

Псевдокод алгоритма MADDPG

for episode = 1, M **do**

Инициализируется рандомный процесс $\mathcal{N}$ для исследования

Получаются начальные наблюдаемые состояния x

for t = 1, максимальная длина эпизода **do**

Для каждого агента i выбирается действие $a_t = \mu_{\theta_i}(o_i) + \mathcal{N}_t$ исходя из текущей политики и исследовательского шума

Выполняются действия $a = (a_1, ..., a_N)$ и наблюдаются награды r и новое состояние x'

Сохраняется переход (x, a, r, x') в реплей-буффер D

$x \leftarrow x'$

for episode = 1, N **do**

Семплируется мини-набор из S переходов (x^j, a^j, r^j, x'^j) из D

Задаётся $y^i = r_i^j + \gamma Q_i^{\mu'}(x'^j, a'_1, ..., a'_N)|_{a'_k = \mu'_k(o_k^j)\theta^{Q'})}$

Обновляется сеть критика минимизацией функции потерь: $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j (y^j - Q_i^{\mu}(x^j, a_1^j, ..., a_N^j|\theta^Q))^2$

Обновляется сеть актора с использованием семплированного градиента политики:

$$\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i}\mu_i(o_i^j)\nabla_{a_i} Q_i^{\mu}(x^j, a_1^j, ..., a_i, ..., a_N^j)|_{a_i=\mu_i(o_i^j)} \qquad (П2.1)$$

end for

Обновляются параметры целевых сетей для каждого агента i:

$$\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i \qquad (П2.2)$$

end for

end for

Приложение 3

Реализация

Листинг 3.1

experiments/comm_checker.py

```
import numpy as np


class CommChecker(object):
    def __init__(self, num_agents, comm_check_rate=10,
       comm_dim=3):
        self.num_agents = num_agents
        self.communications_matches = np.zeros(num_agents)
        self.communications_matches_count = 0
        self.communications_matches_matrix = np.zeros([
           num_agents, comm_dim, comm_dim])  # env.world.dim_c
            == 3
        self.comm_check_rate = comm_check_rate
        self.comm_dim = comm_dim

    def check(self, observations, agents, episode_step):
        if episode_step % self.comm_check_rate != 0:
            return
        matches_results = np.zeros(self.num_agents)
        for i, obs, agent, matrix, matches_result \
                in zip(range(self.num_agents), observations,
                   agents, self.communications_matches_matrix,
                       matches_results):
            if agent.silent:
                continue
            obs = obs[:self.comm_dim]
            obs_max_index = obs.argmax()
            comm_action = agent.action.c
            max_comm_action = np.zeros(len(comm_action))
            max_comm_action[comm_action.argmax()] = 1.
            # determinant
            matrix[obs_max_index] = max_comm_action
            matches_results[i] = np.linalg.det(matrix)

        self.communications_matches_count += 1
        self.communications_matches = self.
           communications_matches + matches_results
```

```
    def get_result(self):
        result = self.communications_matches / self.
            communications_matches_count

        self.communications_matches = np.zeros(self.num_agents
            )
        self.communications_matches_count = 0

        return result
```

Листинг 3.2

experiments/parse_args.py

```
import argparse

def parse_args():
    parser = argparse.ArgumentParser("Reinforcement Learning
        experiments for multiagent environments")
    # Environment
    parser.add_argument("--scenario", type=str, default="
        simple_tag", help="name of the scenario script")
    parser.add_argument("--max-episode-len", type=int, default
        =150, help="maximum episode length")
    parser.add_argument("--num-episodes", type=int, default
        =50000, help="number of episodes")
    parser.add_argument("--exp-name", type=str, default="st-
        borders-50k-0", help="name of the experiment")
    parser.add_argument("--num-adversaries", type=int, default
        =0, help="number of adversaries")
    parser.add_argument("--good-policy", type=str, default="
        maddpg", help="policy for good agents")
    parser.add_argument("--adv-policy", type=str, default="
        maddpg", help="policy of adversaries")
    # Core training parameters
    parser.add_argument("--lr", type=float, default=1e-2, help
        ="learning rate for Adam optimizer")
    parser.add_argument("--gamma", type=float, default=0.98,
        help="discount factor")
    parser.add_argument("--batch-size", type=int, default
        =1024, help="number of episodes to optimize at the same
         time")
    parser.add_argument("--num-units", type=int, default=64,
        help="number of units in the mlp")
    # Checkpointing
```

```
    parser.add_argument("--save-dir", type=str, default="./out
       /", help="directory in which training state and model
       should be saved")
    parser.add_argument("--save-rate", type=int, default=1000,
        help="save model once every time this many episodes
       are completed")
    parser.add_argument("--load-dir", type=str, default="",
       help="directory in which training state and model are
       loaded")
    # Evaluation
    parser.add_argument("--display", action="store_true",
       default=False)
    parser.add_argument("--plots-dir", type=str, default="./
       out/learning_curves/", help="directory where plot data
       is saved")
    return parser.parse_args()
```

Листинг 3.3

experiments/plotter.py

```
import numpy as np


class Plotter(object):
    def __init__(self, vis, frequency=100, title=None, ylabel=
       None, xlabel=None, legend=None):
        self.vis = vis
        self.win = None
        self.frequency = frequency
        self.title = title
        self.ylabel = ylabel
        self.xlabel = xlabel
        self.legend = legend
        self.values_array = []
        self.counter = 0
        self.reset()

    def reset(self):
        self.values_array.clear()
        self.counter = 0

    def log(self, episode, values):
        self.counter += 1
        self.values_array.append(values)
        if self.counter == self.frequency:
```

```
            self.plot(episode, np.mean(np.array(self.
               values_array), 0))
            self.reset()

    def plot(self, episode, values):
        n_lines = len(values)
        if self.win is None:
            self.win = self.vis.line(X=np.arange(episode,
               episode + 1),
                                     Y=np.array([np.array(
                                        values)]),
                                     opts=dict(
                                         ylabel=self.ylabel,
                                         xlabel=self.xlabel,
                                         title=self.title,
                                         legend=self.legend))
        else:
            self.vis.line(X=np.array(
                [np.array(episode).repeat(n_lines)]),
                Y=np.array([np.array(values)]),
                win=self.win,
                update='append')
```

Листинг 3.4

experiments/train.py

```
import pickle
import time
from pathlib import Path

import multiagent.scenarios as scenarios
import numpy as np
import tensorflow as tf
import tensorflow.contrib.layers as layers
import visdom

import maddpg.common.tf_util as U
from experiments.parse_args import parse_args
from maddpg.trainer.maddpg import MADDPGAgentTrainer


def mlp_model(input, num_outputs, scope, reuse=False,
   num_units=64, rnn_cell=None):
    # This model takes as input an observation and returns
       values of all actions
```

```
    with tf.variable_scope(scope, reuse=reuse):
        out = input
        out = layers.fully_connected(out, num_outputs=
           num_units, activation_fn=tf.nn.relu)
        out = layers.fully_connected(out, num_outputs=
           num_units, activation_fn=tf.nn.relu)
        out = layers.fully_connected(out, num_outputs=
           num_outputs, activation_fn=None)
        return out


def make_env(scenario_name, benchmark=False):
    # load scenario from script
    scenario = scenarios.load(scenario_name + ".py").Scenario
       ()
    # create world
    world = scenario.make_world()
    # create multiagent environment
    env = scenario.get_env(world, scenario.reset_world,
       scenario.reward, scenario.observation,
                           done_callback=scenario.done)
    return env


def get_trainers(env, num_adversaries, obs_shape_n, arglist):
    trainers = []
    model = mlp_model
    trainer = MADDPGAgentTrainer
    for i in range(num_adversaries):
        trainers.append(trainer(
            "adversary_%d" % i, model, obs_shape_n, env.
               action_space, i, arglist,
            local_q_func=(arglist.adv_policy == 'ddpg')))
    for i in range(num_adversaries, env.n):
        trainers.append(trainer(
            "agent_%d" % i, model, obs_shape_n, env.
               action_space, i, arglist,
            local_q_func=(arglist.good_policy == 'ddpg')))
    return trainers


def save_state(directory, saver):
    Path(directory).mkdir(parents=True, exist_ok=True)
    U.save_state(directory, saver=saver)
    print("Model weights saved to folder " + directory)
```

```
def load_state(load_dir):
    try:
        U.load_state(load_dir)
        print('Previous state loaded')
    except:
        print("can not load")


def process(arglist):
    if arglist.exp_name == None:
        arglist.exp_name = arglist.scenario
    save_dir = arglist.save_dir + arglist.exp_name + '/'
    Path(save_dir).mkdir(parents=True, exist_ok=True)
    # Load previous results, if necessary
    if arglist.load_dir == "":
        arglist.load_dir = save_dir

    if arglist.display:
        play(arglist)
    else:
        train(arglist)


def train(arglist):
    with U.single_threaded_session():
        # Create environment
        env = make_env(arglist.scenario)
        # Create agent trainers
        obs_shape_n = [env.observation_space[i].shape for i in
            range(env.n)]
        num_adversaries = get_num_adversaries(env)
        trainers = get_trainers(env, num_adversaries,
           obs_shape_n, arglist)
        print('Using good policy {} and adv policy {}'.format(
           arglist.good_policy, arglist.adv_policy))

        from experiments.plotter import Plotter
        vis = visdom.Visdom(port=8097)
        title = arglist.scenario + " " + arglist.exp_name
        episode = 'Episode'
        reward_plotter = Plotter(vis,
                                 title=title,
                                 ylabel='Reward',
```

```
                                xlabel=episode,
                                legend=['Agent-%d' % i for i
                                    in range(env.n)])
        comm_plotter = Plotter(vis,
                               title=title,
                               ylabel='Consistency',
                               xlabel=episode,
                               legend=['Agent-%d' % i for i in
                                   range(env.n)],
                               frequency=10)
        time_plotter = Plotter(vis,
                               title=title,
                               ylabel='Time, sec',
                               xlabel=episode,
                               legend=['Time'], frequency=10)
        overall_time_plotter = Plotter(vis,
                                       title=title,
                                       ylabel='Time from the
                                           beginning',
                                       xlabel=episode,
                                       legend=['Time'],
                                           frequency=1)

        U.initialize()

        load_state(arglist.load_dir)

        episode_rewards = [0.0]  # sum of rewards for all
            agents
        agent_rewards = [[0.0] for _ in range(env.n)]  #
            individual agent reward
        final_ep_rewards = []  # sum of rewards for training
            curve
        final_ep_ag_rewards = []  # agent rewards for training
             curve
        agent_info = [[[]]]  # placeholder for benchmarking
            info
        saver = tf.train.Saver()
        obs_n = env.reset()
        episode_step = 0
        train_step = 0
        episodes_count = 0
        t_start = time.time()
        t_start_p = time.time()
        t_start_overall = time.time()
```

```
135
        from experiments.comm_checker import CommChecker
        comm_checker = CommChecker(env.n, comm_dim=env.world.
           dim_c)

        print('Starting iterations...')
140     while True:
            # get action
            action_n = [agent.action(obs) for agent, obs in
               zip(trainers, obs_n)]
            # environment step
            new_obs_n, rew_n, done_n, info_n = env.step(
               action_n)
145         episode_step += 1
            comm_checker.check(obs_n, env.world.agents,
               episode_step)
            done = all(done_n)
            terminal = (episode_step >= arglist.
               max_episode_len)
            # collect experience
150         for i, agent in enumerate(trainers):
                agent.experience(obs_n[i], action_n[i], rew_n[
                   i], new_obs_n[i], done_n[i], terminal)
            obs_n = new_obs_n

            for i, rew in enumerate(rew_n):
155             episode_rewards[-1] += rew
                agent_rewards[i][-1] += rew

            if done or terminal:
                episodes_count += 1
160             reward_plotter.log(episodes_count, rew_n)
                if episodes_count % 10 == 0:
                    time_plotter.log(episodes_count, [time.
                       time() - t_start_p])
                    t_start_p = time.time()
                if episodes_count % 100 == 0:
165                 overall_time_plotter.log(episodes_count, [
                       time.time() - t_start_overall])
                comm_plotter.log(episodes_count, comm_checker.
                   get_result())

                obs_n = env.reset()
                episode_step = 0
170             episode_rewards.append(0)
```

```
                for a in agent_rewards:
                    a.append(0)
                agent_info.append([[]])

            # increment global step counter
            train_step += 1

            # update all trainers, if not in display mode
            loss = None
            for agent in trainers:
                agent.preupdate()
            for agent in trainers:
                loss = agent.update(trainers, train_step)

            # save model, display training output
            if terminal and (episodes_count % (arglist.
               save_rate * 2) == 0):
                save_state(arglist.load_dir + "ep" + str(
                   episodes_count) + "/", saver)
            if terminal and (episodes_count % arglist.
               save_rate == 0):
                save_state(arglist.load_dir, saver)
                mean_episode_reward = np.mean(episode_rewards)
                episode_rewards = [0.0]
                # print statement depends on whether or not
                   there are adversaries
                if num_adversaries == 0:
                    print("steps: {}, episodes: {}, mean
                       episode reward: {}, time: {}".format(
                        train_step, episodes_count,
                           mean_episode_reward, round(time.
                           time() - t_start, 3)))
                else:
                    print("steps: {}, episodes: {}, mean
                       episode reward: {}, agent episode
                       reward: {}, time: {}".format(
                        train_step, episodes_count,
                           mean_episode_reward,
                        [np.mean(rew[-arglist.save_rate:]) for
                            rew in agent_rewards], round(time.
                           time() - t_start, 3)))
                t_start = time.time()
                # Keep track of final episode reward
                final_ep_rewards.append(mean_episode_reward)
                for rew in agent_rewards:
```

```
                    final_ep_ag_rewards.append(np.mean(rew[-
                        arglist.save_rate:]))
205
            # saves final episode reward for plotting training
                curve later
            if episodes_count > arglist.num_episodes:
                Path(arglist.plots_dir).mkdir(parents=True,
                    exist_ok=True)
                rew_file_name = arglist.plots_dir + arglist.
                    exp_name + '_rewards.pkl'
210             with open(rew_file_name, 'wb') as fp:
                    pickle.dump(final_ep_rewards, fp)
                agrew_file_name = arglist.plots_dir + arglist.
                    exp_name + '_agrewards.pkl'
                with open(agrew_file_name, 'wb') as fp:
                    pickle.dump(final_ep_ag_rewards, fp)
215             print('...Finished total of {} episodes.'.
                    format(len(episode_rewards)))
                break


def play(arglist):
220     with U.single_threaded_session():
        # Create environment
        env = make_env(arglist.scenario)
        # Create agent trainers
        obs_shape_n = [env.observation_space[i].shape for i in
            range(env.n)]
225     num_adversaries = get_num_adversaries(env)
        trainers = get_trainers(env, num_adversaries,
            obs_shape_n, arglist)
        print('Using good policy {} and adv policy {}'.format(
            arglist.good_policy, arglist.adv_policy))

        # Initialize
230     U.initialize()

        load_state(arglist.load_dir)

        episode_rewards = [0.0]  # sum of rewards for all
            agents
235     agent_rewards = [[0.0] for _ in range(env.n)]  #
            individual agent reward
        agent_info = [[[]]]  # placeholder for benchmarking
            info
```

```
        obs_n = env.reset()
        episode_step = 0
        train_step = 0
        t_start = time.time()

        print('Starting iterations...')
        while True:
            # get action
            action_n = [agent.action(obs) for agent, obs in
               zip(trainers, obs_n)]
            # environment step
            new_obs_n, rew_n, done_n, info_n = env.step(
               action_n)
            episode_step += 1
            done = all(done_n)
            terminal = (episode_step >= arglist.
               max_episode_len)
            # # collect experience
            # for i, agent in enumerate(trainers):
            #     agent.experience(obs_n[i], action_n[i],
               rew_n[i], new_obs_n[i], done_n[i], terminal)
            obs_n = new_obs_n

            for i, rew in enumerate(rew_n):
                episode_rewards[-1] += rew
                agent_rewards[i][-1] += rew

            if done or terminal:
                print("train step: {}, episode reward: {},
                   time: {}".format(
                    train_step, np.mean(episode_rewards[-1:]),
                        round(time.time() - t_start, 3)))
                obs_n = env.reset()
                episode_step = 0
                episode_rewards.append(0)
                for a in agent_rewards:
                    a.append(0)
                agent_info.append([[]])

            # increment global step counter
            train_step += 1

            # for displaying learned policies
            time.sleep(0.1)
            env.render()
```

```
def get_num_adversaries(env):
    return np.sum([a.adversary if hasattr(a, 'adversary') else
        False for a in env.agents])


if __name__ == '__main__':
    arglist = parse_args()
    process(arglist)
```

Листинг 3.5

maddpg/common/distributions.py

```
import numpy as np
import tensorflow as tf
from multiagent.multi_discrete import MultiDiscrete
from tensorflow.python.ops import math_ops

import maddpg.common.tf_util as U


class Pd(object):
    """
    A particular probability distribution
    """

    def flatparam(self):
        raise NotImplementedError

    def mode(self):
        raise NotImplementedError

    def logp(self, x):
        raise NotImplementedError

    def kl(self, other):
        raise NotImplementedError

    def entropy(self):
        raise NotImplementedError

    def sample(self):
        raise NotImplementedError
```

```
class PdType(object):
    """
    Parametrized family of probability distributions
    """

    def pdclass(self):
        raise NotImplementedError

    def pdfromflat(self, flat):
        return self.pdclass()(flat)

    def param_shape(self):
        raise NotImplementedError

    def sample_shape(self):
        raise NotImplementedError

    def sample_dtype(self):
        raise NotImplementedError

    def param_placeholder(self, prepend_shape, name=None):
        return tf.placeholder(dtype=tf.float32, shape=
           prepend_shape + self.param_shape(), name=name)

    def sample_placeholder(self, prepend_shape, name=None):
        return tf.placeholder(dtype=self.sample_dtype(), shape
           =prepend_shape + self.sample_shape(), name=name)


class CategoricalPdType(PdType):
    def __init__(self, ncat):
        self.ncat = ncat

    def pdclass(self):
        return CategoricalPd

    def param_shape(self):
        return [self.ncat]

    def sample_shape(self):
        return []

    def sample_dtype(self):
        return tf.int32
```

```
class SoftCategoricalPdType(PdType):
    def __init__(self, ncat):
        self.ncat = ncat

    def pdclass(self):
        return SoftCategoricalPd

    def param_shape(self):
        return [self.ncat]

    def sample_shape(self):
        return [self.ncat]

    def sample_dtype(self):
        return tf.float32


class MultiCategoricalPdType(PdType):
    def __init__(self, low, high):
        self.low = low
        self.high = high
        self.ncats = high - low + 1

    def pdclass(self):
        return MultiCategoricalPd

    def pdfromflat(self, flat):
        return MultiCategoricalPd(self.low, self.high, flat)

    def param_shape(self):
        return [sum(self.ncats)]

    def sample_shape(self):
        return [len(self.ncats)]

    def sample_dtype(self):
        return tf.int32


class SoftMultiCategoricalPdType(PdType):
    def __init__(self, low, high):
        self.low = low
        self.high = high
```

```
        self.ncats = high - low + 1

    def pdclass(self):
        return SoftMultiCategoricalPd

    def pdfromflat(self, flat):
        return SoftMultiCategoricalPd(self.low, self.high,
           flat)

    def param_shape(self):
        return [sum(self.ncats)]

    def sample_shape(self):
        return [sum(self.ncats)]

    def sample_dtype(self):
        return tf.float32


class DiagGaussianPdType(PdType):
    def __init__(self, size):
        self.size = size

    def pdclass(self):
        return DiagGaussianPd

    def param_shape(self):
        return [2 * self.size]

    def sample_shape(self):
        return [self.size]

    def sample_dtype(self):
        return tf.float32


class BernoulliPdType(PdType):
    def __init__(self, size):
        self.size = size

    def pdclass(self):
        return BernoulliPd

    def param_shape(self):
        return [self.size]
```

```
165
    def sample_shape(self):
        return [self.size]

    def sample_dtype(self):
170         return tf.int32


# WRONG SECOND DERIVATIVES
# class CategoricalPd(Pd):
175 #     def __init__(self, logits):
#         self.logits = logits
#         self.ps = tf.nn.softmax(logits)
#     @classmethod
#     def fromflat(cls, flat):
180 #         return cls(flat)
#     def flatparam(self):
#         return self.logits
#     def mode(self):
#         return U.argmax(self.logits, axis=1)
185 #     def logp(self, x):
#         return -tf.nn.
   sparse_softmax_cross_entropy_with_logits(self.logits, x)
#     def kl(self, other):
#         return tf.nn.softmax_cross_entropy_with_logits(other
   .logits, self.ps) \
#                 - tf.nn.softmax_cross_entropy_with_logits(
   self.logits, self.ps)
190 #     def entropy(self):
#         return tf.nn.softmax_cross_entropy_with_logits(self.
   logits, self.ps)
#     def sample(self):
#         u = tf.random_uniform(tf.shape(self.logits))
#         return U.argmax(self.logits - tf.log(-tf.log(u)),
   axis=1)
195

class CategoricalPd(Pd):
    def __init__(self, logits):
        self.logits = logits

200     def flatparam(self):
        return self.logits

    def mode(self):
        return U.argmax(self.logits, axis=1)
```

```
205
    def logp(self, x):
        return -tf.nn.sparse_softmax_cross_entropy_with_logits
           (logits=self.logits, labels=x)

    def kl(self, other):
210         a0 = self.logits - U.max(self.logits, axis=1, keepdims
               =True)
            a1 = other.logits - U.max(other.logits, axis=1,
               keepdims=True)
            ea0 = tf.exp(a0)
            ea1 = tf.exp(a1)
            z0 = U.sum(ea0, axis=1, keepdims=True)
215         z1 = U.sum(ea1, axis=1, keepdims=True)
            p0 = ea0 / z0
            return U.sum(p0 * (a0 - tf.log(z0) - a1 + tf.log(z1)),
                axis=1)

    def entropy(self):
220         a0 = self.logits - U.max(self.logits, axis=1, keepdims
               =True)
            ea0 = tf.exp(a0)
            z0 = U.sum(ea0, axis=1, keepdims=True)
            p0 = ea0 / z0
            return U.sum(p0 * (tf.log(z0) - a0), axis=1)
225
    def sample(self):
        u = tf.random_uniform(tf.shape(self.logits))
        return U.argmax(self.logits - tf.log(-tf.log(u)), axis
           =1)

230 @classmethod
    def fromflat(cls, flat):
        return cls(flat)


235 class SoftCategoricalPd(Pd):
    def __init__(self, logits):
        self.logits = logits

    def flatparam(self):
240         return self.logits

    def mode(self):
        return U.softmax(self.logits, axis=-1)
```

```
    def logp(self, x):
        return -tf.nn.softmax_cross_entropy_with_logits(logits
           =self.logits, labels=x)

    def kl(self, other):
        a0 = self.logits - U.max(self.logits, axis=1, keepdims
           =True)
        a1 = other.logits - U.max(other.logits, axis=1,
           keepdims=True)
        ea0 = tf.exp(a0)
        ea1 = tf.exp(a1)
        z0 = U.sum(ea0, axis=1, keepdims=True)
        z1 = U.sum(ea1, axis=1, keepdims=True)
        p0 = ea0 / z0
        return U.sum(p0 * (a0 - tf.log(z0) - a1 + tf.log(z1)),
            axis=1)

    def entropy(self):
        a0 = self.logits - U.max(self.logits, axis=1, keepdims
           =True)
        ea0 = tf.exp(a0)
        z0 = U.sum(ea0, axis=1, keepdims=True)
        p0 = ea0 / z0
        return U.sum(p0 * (tf.log(z0) - a0), axis=1)

    def sample(self):
        u = tf.random_uniform(tf.shape(self.logits))
        return U.softmax(self.logits - tf.log(-tf.log(u)),
           axis=-1)

    @classmethod
    def fromflat(cls, flat):
        return cls(flat)


class MultiCategoricalPd(Pd):
    def __init__(self, low, high, flat):
        self.flat = flat
        self.low = tf.constant(low, dtype=tf.int32)
        self.categoricals = list(map(CategoricalPd, tf.split(
           flat, high - low + 1, axis=len(flat.get_shape()) -
           1)))

    def flatparam(self):
```

```
        return self.flat

    def mode(self):
        return self.low + tf.cast(tf.stack([p.mode() for p in
            self.categoricals], axis=-1), tf.int32)

    def logp(self, x):
        return tf.add_n(
            [p.logp(px) for p, px in zip(self.categoricals, tf
                .unstack(x - self.low, axis=len(x.get_shape())
                - 1))])

    def kl(self, other):
        return tf.add_n([
            p.kl(q) for p, q in zip(self.categoricals, other.
                categoricals)
        ])

    def entropy(self):
        return tf.add_n([p.entropy() for p in self.
            categoricals])

    def sample(self):
        return self.low + tf.cast(tf.stack([p.sample() for p
            in self.categoricals], axis=-1), tf.int32)

    @classmethod
    def fromflat(cls, flat):
        return cls(flat)


class SoftMultiCategoricalPd(Pd):  # doesn't work yet
    def __init__(self, low, high, flat):
        self.flat = flat
        self.low = tf.constant(low, dtype=tf.float32)
        self.categoricals = list(map(SoftCategoricalPd, tf.
            split(flat, high - low + 1, axis=len(flat.get_shape
            ()) - 1)))

    def flatparam(self):
        return self.flat

    def mode(self):
        x = []
        for i in range(len(self.categoricals)):
```

```
            x.append(self.low[i] + self.categoricals[i].mode()
                )
        return tf.concat(x, axis=-1)

    def logp(self, x):
        return tf.add_n(
            [p.logp(px) for p, px in zip(self.categoricals, tf
                .unstack(x - self.low, axis=len(x.get_shape())
                - 1))])

    def kl(self, other):
        return tf.add_n([
            p.kl(q) for p, q in zip(self.categoricals, other.
                categoricals)
        ])

    def entropy(self):
        return tf.add_n([p.entropy() for p in self.
            categoricals])

    def sample(self):
        x = []
        for i in range(len(self.categoricals)):
            x.append(self.low[i] + self.categoricals[i].sample
                ())
        return tf.concat(x, axis=-1)

    @classmethod
    def fromflat(cls, flat):
        return cls(flat)


class DiagGaussianPd(Pd):
    def __init__(self, flat):
        self.flat = flat
        mean, logstd = tf.split(axis=1, num_or_size_splits=2,
            value=flat)
        self.mean = mean
        self.logstd = logstd
        self.std = tf.exp(logstd)

    def flatparam(self):
        return self.flat

    def mode(self):
```

```
        return self.mean

    def logp(self, x):
        return - 0.5 * U.sum(tf.square((x - self.mean) / self.
           std), axis=1) \
360             - 0.5 * np.log(2.0 * np.pi) * tf.to_float(tf.
                   shape(x)[1]) \
                - U.sum(self.logstd, axis=1)

    def kl(self, other):
        assert isinstance(other, DiagGaussianPd)
365     return U.sum(other.logstd - self.logstd + (tf.square(
           self.std) + tf.square(self.mean - other.mean)) / (
                     2.0 * tf.square(other.std)) - 0.5, axis=1)

    def entropy(self):
        return U.sum(self.logstd + .5 * np.log(2.0 * np.pi *
           np.e), 1)
370
    def sample(self):
        return self.mean + self.std * tf.random_normal(tf.
           shape(self.mean))

    @classmethod
375 def fromflat(cls, flat):
        return cls(flat)


class BernoulliPd(Pd):
380 def __init__(self, logits):
        self.logits = logits
        self.ps = tf.sigmoid(logits)

    def flatparam(self):
385     return self.logits

    def mode(self):
        return tf.round(self.ps)

390 def logp(self, x):
        return - U.sum(tf.nn.sigmoid_cross_entropy_with_logits
           (logits=self.logits, labels=tf.to_float(x)), axis
           =1)

    def kl(self, other):
```

```
        return U.sum(tf.nn.sigmoid_cross_entropy_with_logits(
            logits=other.logits, labels=self.ps), axis=1) - U.
            sum(
395             tf.nn.sigmoid_cross_entropy_with_logits(logits=
                    self.logits, labels=self.ps), axis=1)

    def entropy(self):
        return U.sum(tf.nn.sigmoid_cross_entropy_with_logits(
            logits=self.logits, labels=self.ps), axis=1)

400 def sample(self):
        p = tf.sigmoid(self.logits)
        u = tf.random_uniform(tf.shape(p))
        return tf.to_float(math_ops.less(u, p))

405 @classmethod
    def fromflat(cls, flat):
        return cls(flat)


410 def make_pdtype(ac_space):
    from gym import spaces
    if isinstance(ac_space, spaces.Box):
        assert len(ac_space.shape) == 1
        return DiagGaussianPdType(ac_space.shape[0])
415 elif isinstance(ac_space, spaces.Discrete):
        # return CategoricalPdType(ac_space.n)
        return SoftCategoricalPdType(ac_space.n)
    elif isinstance(ac_space, MultiDiscrete):
        # return MultiCategoricalPdType(ac_space.low, ac_space
            .high)
420         return SoftMultiCategoricalPdType(ac_space.low,
            ac_space.high)
    elif isinstance(ac_space, spaces.MultiBinary):
        return BernoulliPdType(ac_space.n)
    else:
        raise NotImplementedError
425


def shape_el(v, i):
    maybe = v.get_shape()[i]
    if maybe is not None:
430         return maybe
    else:
        return tf.shape(v)[i]
```

Листинг 3.6

maddpg/common/tf_util.py

```
import collections
import os

import numpy as np
import tensorflow as tf


def sum(x, axis=None, keepdims=False):
    return tf.reduce_sum(x, axis=None if axis is None else [
        axis], keep_dims=keepdims)


def mean(x, axis=None, keepdims=False):
    return tf.reduce_mean(x, axis=None if axis is None else [
        axis], keep_dims=keepdims)


def var(x, axis=None, keepdims=False):
    meanx = mean(x, axis=axis, keepdims=keepdims)
    return mean(tf.square(x - meanx), axis=axis, keepdims=
        keepdims)


def std(x, axis=None, keepdims=False):
    return tf.sqrt(var(x, axis=axis, keepdims=keepdims))


def max(x, axis=None, keepdims=False):
    return tf.reduce_max(x, axis=None if axis is None else [
        axis], keep_dims=keepdims)


def min(x, axis=None, keepdims=False):
    return tf.reduce_min(x, axis=None if axis is None else [
        axis], keep_dims=keepdims)


def concatenate(arrs, axis=0):
    return tf.concat(axis=axis, values=arrs)


def argmax(x, axis=None):
```

```
    return tf.argmax(x, axis=axis)


def softmax(x, axis=None):
    return tf.nn.softmax(x, axis=axis)


#
   ================================================================

# Misc
#
   ================================================================



def is_placeholder(x):
    return type(x) is tf.Tensor and len(x.op.inputs) == 0


#
   ================================================================

# Inputs
#
   ================================================================



class TfInput(object):
    def __init__(self, name="(unnamed)"):
        """Generalized Tensorflow placeholder. The main
           differences are:
            - possibly uses multiple placeholders internally
               and returns multiple values
            - can apply light postprocessing to the value feed
                to placeholder.
        """
        self.name = name

    def get(self):
        """Return the tf variable(s) representing the possibly
            postprocessed value
        of placeholder(s).
        """
```

```
        raise NotImplemented()

    def make_feed_dict(data):
        """Given data input it to the placeholder(s)."""
        raise NotImplemented()


class PlacholderTfInput(TfInput):
    def __init__(self, placeholder):
        """Wrapper for regular tensorflow placeholder."""
        super().__init__(placeholder.name)
        self._placeholder = placeholder

    def get(self):
        return self._placeholder

    def make_feed_dict(self, data):
        return {self._placeholder: data}


class BatchInput(PlacholderTfInput):
    def __init__(self, shape, dtype=tf.float32, name=None):
        """Creates a placeholder for a batch of tensors of a
           given shape and dtype

        Parameters
        ----------
        shape: [int]
            shape of a single elemenet of the batch
        dtype: tf.dtype
            number representation used for tensor contents
        name: str
            name of the underlying placeholder
        """
        super().__init__(tf.placeholder(dtype, [None] + list(
           shape), name=name))


class Uint8Input(PlacholderTfInput):
    def __init__(self, shape, name=None):
        """Takes input in uint8 format which is cast to
           float32 and divided by 255
        before passing it to the model.

        On GPU this ensures lower data transfer times.
```

```
        Parameters
        ----------
        shape: [int]
            shape of the tensor.
        name: str
            name of the underlying placeholder
        """

        super().__init__(tf.placeholder(tf.uint8, [None] +
           list(shape), name=name))
        self._shape = shape
        self._output = tf.cast(super().get(), tf.float32) /
           255.0

    def get(self):
        return self._output


def ensure_tf_input(thing):
    """Takes either tf.placeholder of TfInput and outputs
       equivalent TfInput"""
    if isinstance(thing, TfInput):
        return thing
    elif is_placeholder(thing):
        return PlacholderTfInput(thing)
    else:
        raise ValueError("Must be a placeholder or TfInput")


#
   ================================================================

# Mathematical utils
#
   ================================================================



def huber_loss(x, delta=1.0):
    """Reference: https://en.wikipedia.org/wiki/Huber_loss"""
    return tf.where(
        tf.abs(x) < delta,
        tf.square(x) * 0.5,
        delta * (tf.abs(x) - 0.5 * delta)
```

```
    )


#
    ====================================================================

# Optimizer utils
#
    ====================================================================



def minimize_and_clip(optimizer, objective, var_list, clip_val
    =10):
    """Minimized `objective` using `optimizer` w.r.t.
       variables in
    `var_list` while ensure the norm of the gradients for each
    variable is clipped to `clip_val`
    """
    if clip_val is None:
        return optimizer.minimize(objective, var_list=var_list
           )
    else:
        gradients = optimizer.compute_gradients(objective,
           var_list=var_list)
        for i, (grad, var) in enumerate(gradients):
            if grad is not None:
                gradients[i] = (tf.clip_by_norm(grad, clip_val
                   ), var)
        return optimizer.apply_gradients(gradients)


#
    ====================================================================

# Global session
#
    ====================================================================


def get_session():
    """Returns recently made Tensorflow session"""
    return tf.get_default_session()
```

```
def make_session(num_cpu):
    """Returns a session that will use <num_cpu> CPU's only"""
    tf_config = tf.ConfigProto(
        inter_op_parallelism_threads=num_cpu,
        intra_op_parallelism_threads=num_cpu)
    return tf.Session(config=tf_config)


def single_threaded_session():
    """Returns a session which will only use a single CPU"""
    return make_session(1)


ALREADY_INITIALIZED = set()


def initialize():
    """Initialize all the uninitialized variables in the
       global scope."""
    new_variables = set(tf.global_variables()) -
       ALREADY_INITIALIZED
    get_session().run(tf.variables_initializer(new_variables))
    ALREADY_INITIALIZED.update(new_variables)


#
   ============================================================


# Scopes
#
   ============================================================



def scope_vars(scope, trainable_only=False):
    """
    Get variables inside a scope
    The scope can be specified as a string

    Parameters
    ----------
    scope: str or VariableScope
        scope in which the variables reside.
    trainable_only: bool
```

```
        whether or not to return only the variables that were
            marked as trainable.

    Returns
    -------
    vars: [tf.Variable]
        list of variables in `scope`.
    """
    return tf.get_collection(
        tf.GraphKeys.TRAINABLE_VARIABLES if trainable_only
            else tf.GraphKeys.GLOBAL_VARIABLES,
        scope=scope if isinstance(scope, str) else scope.name
    )


def scope_name():
    """Returns the name of current scope as a string, e.g.
        deepq/q_func"""
    return tf.get_variable_scope().name


def absolute_scope_name(relative_scope_name):
    """Appends parent scope name to `relative_scope_name`"""
    return scope_name() + "/" + relative_scope_name


#
    ================================================================

# Saving variables
#
    ================================================================



def load_state(fname, saver=None):
    """Load all the variables to the current session from the
        location <fname>"""
    if saver is None:
        saver = tf.train.Saver()
    saver.restore(get_session(), fname)
    return saver


def save_state(fname, saver=None):
```

```
    """Save all the variables in the current session to the
       location <fname>"""
    os.makedirs(os.path.dirname(fname), exist_ok=True)
    if saver is None:
        saver = tf.train.Saver()
    saver.save(get_session(), fname)
    return saver


#
   ================================================================

# Theano-like Function
#
   ================================================================


def function(inputs, outputs, updates=None, givens=None):
    """Just like Theano function. Take a bunch of tensorflow
       placeholders and expersions
    computed based on those placeholders and produces f(inputs
       ) -> outputs. Function f takes
    values to be feed to the inputs placeholders and produces
       the values of the experessions
    in outputs.

    Input values can be passed in the same order as inputs or
       can be provided as kwargs based
    on placeholder name (passed to constructor or accessible
       via placeholder.op.name).

    Example:
        x = tf.placeholder(tf.int32, (), name="x")
        y = tf.placeholder(tf.int32, (), name="y")
        z = 3 * x + 2 * y
        lin = function([x, y], z, givens={y: 0})

        with single_threaded_session():
            initialize()

            assert lin(2) == 6
            assert lin(x=3) == 9
            assert lin(2, 2) == 10
            assert lin(x=2, y=3) == 12
```

```
    Parameters
    ----------
    inputs: [tf.placeholder or TfInput]
        list of input arguments
    outputs: [tf.Variable] or tf.Variable
        list of outputs or a single output to be returned from
            function. Returned
        value will also have the same shape.
    """
    if isinstance(outputs, list):
        return _Function(inputs, outputs, updates, givens=
            givens)
    elif isinstance(outputs, (dict, collections.OrderedDict)):
        f = _Function(inputs, outputs.values(), updates,
            givens=givens)
        return lambda *args, **kwargs: type(outputs)(zip(
            outputs.keys(), f(*args, **kwargs)))
    else:
        f = _Function(inputs, [outputs], updates, givens=
            givens)
        return lambda *args, **kwargs: f(*args, **kwargs)[0]


class _Function(object):
    def __init__(self, inputs, outputs, updates, givens,
        check_nan=False):
        for inpt in inputs:
            if not issubclass(type(inpt), TfInput):
                assert len(inpt.op.inputs) == 0, "inputs
                    should all be placeholders of rl_algs.
                    common.TfInput"
        self.inputs = inputs
        updates = updates or []
        self.update_group = tf.group(*updates)
        self.outputs_update = list(outputs) + [self.
            update_group]
        self.givens = {} if givens is None else givens
        self.check_nan = check_nan

    def _feed_input(self, feed_dict, inpt, value):
        if issubclass(type(inpt), TfInput):
            feed_dict.update(inpt.make_feed_dict(value))
        elif is_placeholder(inpt):
            feed_dict[inpt] = value
```

```
330
    def __call__(self, *args, **kwargs):
        assert len(args) <= len(self.inputs), "Too many
           arguments provided"
        feed_dict = {}
        # Update the args
335     for inpt, value in zip(self.inputs, args):
            self._feed_input(feed_dict, inpt, value)
        # Update the kwargs
        kwargs_passed_inpt_names = set()
        for inpt in self.inputs[len(args):]:
340         inpt_name = inpt.name.split(':')[0]
            inpt_name = inpt_name.split('/')[-1]
            assert inpt_name not in kwargs_passed_inpt_names,
               \
                "this function has two arguments with the same
                    name \"{}\", so kwargs cannot be used.".
                   format(inpt_name)
            if inpt_name in kwargs:
345             kwargs_passed_inpt_names.add(inpt_name)
                self._feed_input(feed_dict, inpt, kwargs.pop(
                   inpt_name))
            else:
                assert inpt in self.givens, "Missing argument
                   " + inpt_name
        assert len(kwargs) == 0, "Function got extra arguments
            " + str(list(kwargs.keys()))
350     # Update feed dict with givens.
        for inpt in self.givens:
            feed_dict[inpt] = feed_dict.get(inpt, self.givens[
               inpt])
        results = get_session().run(self.outputs_update,
           feed_dict=feed_dict)[:-1]
        if self.check_nan:
355         if any(np.isnan(r).any() for r in results):
                raise RuntimeError("Nan detected")
        return results
```

Листинг 3.7

maddpg/trainer/replay_buffer.py

```
import random

import numpy as np
5
```

```
class ReplayBuffer(object):
    def __init__(self, size):
        """Create Prioritized Replay buffer.

        Parameters
        ----------
        size: int
            Max number of transitions to store in the buffer.
               When the buffer
            overflows the old memories are dropped.
        """
        self._storage = []
        self._maxsize = int(size)
        self._next_idx = 0

    def __len__(self):
        return len(self._storage)

    def clear(self):
        self._storage = []
        self._next_idx = 0

    def add(self, obs_t, action, reward, obs_tp1, done):
        data = (obs_t, action, reward, obs_tp1, done)

        if self._next_idx >= len(self._storage):
            self._storage.append(data)
        else:
            self._storage[self._next_idx] = data
        self._next_idx = (self._next_idx + 1) % self._maxsize

    def _encode_sample(self, idxes):
        obses_t, actions, rewards, obses_tp1, dones = [], [],
           [], [], []
        for i in idxes:
            data = self._storage[i]
            obs_t, action, reward, obs_tp1, done = data
            obses_t.append(np.array(obs_t, copy=False))
            actions.append(np.array(action, copy=False))
            rewards.append(reward)
            obses_tp1.append(np.array(obs_tp1, copy=False))
            dones.append(done)
        return np.array(obses_t), np.array(actions), np.array(
           rewards), np.array(obses_tp1), np.array(dones)
```

```
    def make_index(self, batch_size):
        return [random.randint(0, len(self._storage) - 1) for
            _ in range(batch_size)]

    def make_latest_index(self, batch_size):
        idx = [(self._next_idx - 1 - i) % self._maxsize for i
            in range(batch_size)]
        np.random.shuffle(idx)
        return idx

    def sample_index(self, idxes):
        return self._encode_sample(idxes)

    def sample(self, batch_size):
        """Sample a batch of experiences.

        Parameters
        ----------
        batch_size: int
            How many transitions to sample.

        Returns
        -------
        obs_batch: np.array
            batch of observations
        act_batch: np.array
            batch of actions executed given obs_batch
        rew_batch: np.array
            rewards received as results of executing act_batch
        next_obs_batch: np.array
            next set of observations seen after executing
                act_batch
        done_mask: np.array
            done_mask[i] = 1 if executing act_batch[i]
                resulted in
            the end of an episode and 0 otherwise.
        """
        if batch_size > 0:
            idxes = self.make_index(batch_size)
        else:
            idxes = range(0, len(self._storage))
        return self._encode_sample(idxes)

    def collect(self):
```

```
        return self.sample(-1)
```

Листинг 3.8

maddpg/trainer/maddpg.py

```
import numpy as np
import tensorflow as tf

import maddpg.common.tf_util as U
from maddpg import AgentTrainer
from maddpg.common.distributions import make_pdtype
from maddpg.trainer.replay_buffer import ReplayBuffer


def discount_with_dones(rewards, dones, gamma):
    discounted = []
    r = 0
    for reward, done in zip(rewards[::-1], dones[::-1]):
        r = reward + gamma * r
        r = r * (1. - done)
        discounted.append(r)
    return discounted[::-1]


def make_update_exp(vals, target_vals):
    polyak = 1.0 - 1e-2
    expression = []
    for var, var_target in zip(sorted(vals, key=lambda v: v.
       name), sorted(target_vals, key=lambda v: v.name)):
        expression.append(var_target.assign(polyak *
           var_target + (1.0 - polyak) * var))
    expression = tf.group(*expression)
    return U.function([], [], updates=[expression])


def p_train(make_obs_ph_n, act_space_n, p_index, p_func,
   q_func, optimizer, grad_norm_clipping=None, local_q_func=
   False,
            num_units=64, scope="trainer", reuse=None):
    with tf.variable_scope(scope, reuse=reuse):
        # create distribtuions
        act_pdtype_n = [make_pdtype(act_space) for act_space
           in act_space_n]

        # set up placeholders
```

```
        obs_ph_n = make_obs_ph_n
        act_ph_n = [act_pdtype_n[i].sample_placeholder([None],
            name="action" + str(i)) for i in range(len(
           act_space_n))]

        p_input = obs_ph_n[p_index]

        p = p_func(p_input, int(act_pdtype_n[p_index].
           param_shape()[0]), scope="p_func", num_units=
           num_units)
        p_func_vars = U.scope_vars(U.absolute_scope_name("
           p_func"))

        # wrap parameters in distribution
        act_pd = act_pdtype_n[p_index].pdfromflat(p)

        act_sample = act_pd.sample()
        p_reg = tf.reduce_mean(tf.square(act_pd.flatparam()))

        act_input_n = act_ph_n + []
        act_input_n[p_index] = act_pd.sample()
        q_input = tf.concat(obs_ph_n + act_input_n, 1)
        if local_q_func:
            q_input = tf.concat([obs_ph_n[p_index],
               act_input_n[p_index]], 1)
        q = q_func(q_input, 1, scope="q_func", reuse=True,
           num_units=num_units)[:, 0]
        pg_loss = -tf.reduce_mean(q)

        loss = pg_loss + p_reg * 1e-3

        optimize_expr = U.minimize_and_clip(optimizer, loss,
           p_func_vars, grad_norm_clipping)

        # Create callable functions
        train = U.function(inputs=obs_ph_n + act_ph_n, outputs
           =loss, updates=[optimize_expr])
        act = U.function(inputs=[obs_ph_n[p_index]], outputs=
           act_sample)
        p_values = U.function([obs_ph_n[p_index]], p)

        # target network
        target_p = p_func(p_input, int(act_pdtype_n[p_index].
           param_shape()[0]), scope="target_p_func",
                          num_units=num_units)
```

```
        target_p_func_vars = U.scope_vars(U.
            absolute_scope_name("target_p_func"))
        update_target_p = make_update_exp(p_func_vars,
            target_p_func_vars)

        target_act_sample = act_pdtype_n[p_index].pdfromflat(
            target_p).sample()
        target_act = U.function(inputs=[obs_ph_n[p_index]],
            outputs=target_act_sample)

        return act, train, update_target_p, {'p_values':
            p_values, 'target_act': target_act}


def q_train(make_obs_ph_n, act_space_n, q_index, q_func,
    optimizer, grad_norm_clipping=None, local_q_func=False,
            scope="trainer", reuse=None, num_units=64):
    with tf.variable_scope(scope, reuse=reuse):
        # create distribtuions
        act_pdtype_n = [make_pdtype(act_space) for act_space
            in act_space_n]

        # set up placeholders
        obs_ph_n = make_obs_ph_n
        act_ph_n = [act_pdtype_n[i].sample_placeholder([None],
            name="action" + str(i)) for i in range(len(
            act_space_n))]
        target_ph = tf.placeholder(tf.float32, [None], name="
            target")

        q_input = tf.concat(obs_ph_n + act_ph_n, 1)
        if local_q_func:
            q_input = tf.concat([obs_ph_n[q_index], act_ph_n[
                q_index]], 1)
        q = q_func(q_input, 1, scope="q_func", num_units=
            num_units)[:, 0]
        q_func_vars = U.scope_vars(U.absolute_scope_name("
            q_func"))

        q_loss = tf.reduce_mean(tf.square(q - target_ph))

        # viscosity solution to Bellman differential equation
            in place of an initial condition
        q_reg = tf.reduce_mean(tf.square(q))
        loss = q_loss  # + 1e-3 * q_reg
```

```
        optimize_expr = U.minimize_and_clip(optimizer, loss,
            q_func_vars, grad_norm_clipping)

        # Create callable functions
        train = U.function(inputs=obs_ph_n + act_ph_n + [
            target_ph], outputs=loss, updates=[optimize_expr])
        q_values = U.function(obs_ph_n + act_ph_n, q)

        # target network
        target_q = q_func(q_input, 1, scope="target_q_func",
            num_units=num_units)[:, 0]
        target_q_func_vars = U.scope_vars(U.
            absolute_scope_name("target_q_func"))
        update_target_q = make_update_exp(q_func_vars,
            target_q_func_vars)

        target_q_values = U.function(obs_ph_n + act_ph_n,
            target_q)

        return train, update_target_q, {'q_values': q_values,
            'target_q_values': target_q_values}


class MADDPGAgentTrainer(AgentTrainer):
    def __init__(self, name, model, obs_shape_n, act_space_n,
        agent_index, args, local_q_func=False):
        self.name = name
        self.n = len(obs_shape_n)
        self.agent_index = agent_index
        self.args = args
        obs_ph_n = []
        for i in range(self.n):
            obs_ph_n.append(U.BatchInput(obs_shape_n[i], name=
                "observation" + str(i)).get())

        # Create all the functions necessary to train the
            model
        self.q_train, self.q_update, self.q_debug = q_train(
            scope=self.name,
            make_obs_ph_n=obs_ph_n,
            act_space_n=act_space_n,
            q_index=agent_index,
            q_func=model,
```

```
            optimizer=tf.train.AdamOptimizer(learning_rate=
                args.lr),
            grad_norm_clipping=0.5,
            local_q_func=local_q_func,
            num_units=args.num_units
        )
        self.act, self.p_train, self.p_update, self.p_debug =
            p_train(
            scope=self.name,
            make_obs_ph_n=obs_ph_n,
            act_space_n=act_space_n,
            p_index=agent_index,
            p_func=model,
            q_func=model,
            optimizer=tf.train.AdamOptimizer(learning_rate=
                args.lr),
            grad_norm_clipping=0.5,
            local_q_func=local_q_func,
            num_units=args.num_units
        )
        # Create experience buffer
        self.replay_buffer = ReplayBuffer(1e6)
        self.max_replay_buffer_len = args.batch_size * args.
            max_episode_len
        self.replay_sample_index = None

    def action(self, obs):
        return self.act(obs[None])[0]

    def experience(self, obs, act, rew, new_obs, done,
        terminal):
        # Store transition in the replay buffer.
        self.replay_buffer.add(obs, act, rew, new_obs, float(
            done))

    def preupdate(self):
        self.replay_sample_index = None

    def update(self, agents, t):
        if len(self.replay_buffer) < self.
            max_replay_buffer_len:  # replay buffer is not
            large enough
            return
        if not t % 100 == 0:  # only update every 100 steps
            return
```

```
        self.replay_sample_index = self.replay_buffer.
            make_index(self.args.batch_size)
        # collect replay sample from all agents
        obs_n = []
        obs_next_n = []
        act_n = []
        index = self.replay_sample_index
        for i in range(self.n):
            obs, act, rew, obs_next, done = agents[i].
                replay_buffer.sample_index(index)
            obs_n.append(obs)
            obs_next_n.append(obs_next)
            act_n.append(act)
        obs, act, rew, obs_next, done = self.replay_buffer.
            sample_index(index)

        # train q network
        num_sample = 1
        target_q = 0.0
        for i in range(num_sample):
            target_act_next_n = [agents[i].p_debug['target_act
                '](obs_next_n[i]) for i in range(self.n)]
            target_q_next = self.q_debug['target_q_values'](*(
                obs_next_n + target_act_next_n))
            target_q += rew + self.args.gamma * (1.0 - done) *
                target_q_next
        target_q /= num_sample
        q_loss = self.q_train(*(obs_n + act_n + [target_q]))

        # train p network
        p_loss = self.p_train(*(obs_n + act_n))

        self.p_update()
        self.q_update()

        return [q_loss, p_loss, np.mean(target_q), np.mean(rew
            ), np.mean(target_q_next), np.std(target_q)]
```

Листинг 3.9

maddpg/__init__.py

```
class AgentTrainer(object):
    def __init__(self, name, model, obs_shape, act_space, args
        ):
```

```
        raise NotImplemented()

    def action(self, obs):
        raise NotImplemented()

    def process_experience(self, obs, act, rew, new_obs, done,
        terminal):
        raise NotImplemented()

    def preupdate(self):
        raise NotImplemented()

    def update(self, agents):
        raise NotImplemented()
```