Санкт–Петербургский государственный университет

Кондратов Иван Владимирович

Выпускная квалификационная работа

Методы обучения с подкреплением для класса задач теории расписания

Направление 01.04.02 «Прикладная математика и информатика»
Основная образовательная программа ВМ.5759.2019
«Цифровая экономика»

Научный руководитель:
доцент кафедры ММЭС
к.ф. - м.н. Петросян О. Л.

Рецензент:
Главный специалист - аналитик
отдела страховой статистики
АО «Совкомбанк страхование»
Бойко А. В.

Санкт-Петербург
2021 г.

Содержание

Введение

Проблема распределения связанных между собой задач между исполнителями присутствует во многих областях нашей жизни. В качестве примера можно рассмотреть две такие области: облачные вычисления и энергетика.

Потребность общества в вычислительных ресурсах растет из года в год весьма стремительно, а появление новых технологий только подстегивает взрывной рост. Как пример, рост популярности нейронных сетей привел к появлению запроса на высокопроизводительные вычисления с тензорами, что сильно увеличило спрос на видеокарты (GPU), которые ранее использовались в основном в узкоспециализированных задачах (компьютерная графика, симуляция физики). Нейронные сети внедряются повсеместно, что повлекло за собой внедрению специализированных вычислительных модулей в потребительской электронике, даже в мобильной технике.

Хоть рост вычислительной мощности индивидуальных устройств в целом следует знаменитому закону Мура, приближение к физическим ограничениям технического процесса производства процессоров и кремния как материала, становится очевидно [1], что локальные вычисления далеко не всегда смогут удовлетворять нужды пользователей.

Появление Apache Hadoop в 2006 году позволило популяризировать распределенные вычисления, показав преимущества горизонтальной интеграции и параллельных вычислений. Сейчас рынок облачных (распределенных) вычислений оценивается в 760 миллиардов долларов, и только продолжит расти [2].

При взаимодействии большого числа вычислительных узлов возникает проблема управления: как оптимально распределять связанные между собой вычислительные задачи? На эту проблему можно посмотреть с различных точек зрения: оптимизация по числу единовременно задействованных ресурсов, энергоэффективности, длительности выполнения задачи, затрачиваемым ресурсам, стоимости. Данную проблему в математике решает класс алгоритмов теории расписаний.

Алгоритмы теории расписаний долго применялись в данной области, но развитие нейронных сетей и алгоритмов обучения с подкрепление позволило превзойти классические эвристики [3].

Отдельно можно выделить проблему построения расписания на направленных ациклических графах. Для некоторых вычислительных задач уже заранее может быть известен набор подзадач, связи между ними и последовательность их выполнения, представляющий собой вычислительных граф. К таким задачам относится обучение нейронных сетей, декодирование генома.

Если же рассматривать энергетику, то рост так называемой зеленой экономики способствует к всё более широкому использованию возобновляемых источников энергии. Два наиболее популярных – солнечная и ветряная энергия имеют один существенный недостаток: непостоянство. Энергия ветра, преобразуемая в электричество ветряными электрогенераторами, может быть весьма непредсказуемой – в день с сильным ветром энергии может быть переизбыток, в то время как в штиль энергии может не быть совсем. Солнечная энергия же циклична, нам хорошо известен промежуток времени, когда её возможно генерировать. Но здесь возникает известная в энергетике проблема – что делать при резком скачке спроса на электричество? Освещение и отопление в большей степени требуется ночью, когда солнце уже не светит.

Хранение полученный из таких источников энергии может серьёзно повысить их эффективность и привлекательность для потребителя, даже на уровне отдельного домохозяйства [4]. Баланс между запасанием энергии из нескольких непостоянных источников, продажей её обратно в электросеть (популярно в Европе и США), и расходом сейчас может представлять из себя комплексную оптимизационную задачу.

Как и в случае с распределенными вычислениями, мы можем оптимизировать по уровню использования, энергоэффективности, цене. Возникает задача распределения нагрузки, получаемой из различных источников между потребителями и устройствами хранения. Если же уровень потребления электроэнергии известен заранее или может быть достаточно точно предсказан, можно выстроить граф подзадач потребления энергии и рас-

пределять каждую подзадачу на один из генераторов энергии.

Две приведенные выше области, на первый взгляд различные, имеют один и тот же набор оптимизационных задач – распределение связанных между собой задач между исполнителями. Для их решения возможно эффективно применить Reinforcement Learning, который превосходит классические алгоритмы теории расписаний, что и будет показано в данной работе.

Постановка задачи

Пусть задан направленный ациклических граф $G = (V, E)$, где $V = v_1, ..., v_n$ - множество вершин графа, представляющих собой задачи, E - множество ребер графа. Все вершины и ребра имеют веса: вес вершины W_{v_i} представляет собой трудозатраты на исполнение задачи, вес ребра $W_{i,j}$ в свою очередь представляет собой трудозатраты исполнителя при переключении на новую задачу. Каждое ребро $(i, j) \in E$ представляет собой отношение предшествования, задача не может начать своё выполнение, пока не были выполнены все предшествующие задачи.

Вершина без предшественников называется входной вершиной, без наследников – выходной. Если вершин без предшественников несколько, то все они становятся наследниками псевдо-входной вершины с нулевым весом как самой вершины, так и исходящих из неё рёбер. По аналогии, если выходных вершин несколько, то они все получат единого наследника с нулевым весом вершины и входящих в неё ребер. Данное требование не несет в себе практической цели для данной работы, но было введено с целью согласования с возможными требованиями алгоритмов для данного класса задач, по аналогии с работами

Введем понятие исполнителя – агента, который выполняет задачи. Пусть задано 3 типа исполнителя: $small, medium, large$. Каждый из исполнителей принадлежит к определенному типу, единственное их отличие будет заключаться в значении их мощности – параметра, характеризующего эффективность исполнителя при выполнении задачи. Пусть $P = \{P_{small}, P_{medium}, P_{large}\}$ - множество параметров мощности. Пусть задано множество исполнителей $M = \{m_{1_{P_*}}, ..., m_{k_{P_*}}\}$. Для исключения тривиальных решений будем полагать, что $k < n$.

Введем $pred(v_i), succ(v_i)$ - множества непосредственных предшественников и наследников соответственно. Пусть $comp(m_{j_{P_*}})$ - множество выполненных исполнителем $m_{j_{P_*}}$. Тогда длительность выполнения задачи v_i на

$m_{j_{P_*}}$ будет вычисляться как

$$CP(v_i, m_{j_{P_*}}) = \frac{W_{v_i}}{P_*} + \sum_{k \in ext(v_i, m_{j_{P_*}})} W_{(i,k)}$$

где $ext(v_i, m_{j_{P_*}}) = prec(v_i) \setminus (pred(v_i) \cap comp(m_{j_{P_*}}))$ – множество предшественников v_i , которые не были выполнены на машине $m_{j_{P_*}}$. Это означает что, если предшественник задачи был выполнен на данной машине, то затраты на переключение с предшественника на текущую задачу не будет.

Введем понятие относительной длины расписания [5]:

$$SLR = \frac{makespan(solution)}{\sum_{v_i \in CP_{MIN}} \min_{P_* \in P} W_{v_i}}$$

Знаменатель в SLR представляет собой нижнюю границу расписания, являясь минимумом вычислительных затрат задач, составляющего критический путь (наиболее длинный путь в смысле весов, т.е. “самый тяжелый” путь), и без учета веса ребер. В [5] показано, что это значение является нижней границей $makespan$, и, исходя из этого $SLR \geq 1$.

Целью работы является построение Reinforcement Learning модели, которая будет добиваться минимизация SLR при заданном графе G и наборе исполнителей P.

Обзор литературы

В первую очередь стоит обратить внимание на уже имеющиеся и зарекомендовавшие себя алгоритмы построения расписаний на графах. Сравнение производительности данных алгоритмов будет представлено в главе 3.

В статье [6] вводится алгоритм Heterogeneous-Earliest-Finish-Time, который является модификацией классического Earliest-Finish-Time алгоритма теории расписаний, применяется для гетерогенных систем. Его преимущество заключается в получении приемлемого $makespan$ при крайне низких вычислительных затратах. Состоит из двух фаз: приоритезация задачи и выбор исполнителя. Распределение приоритетов основано на ранжировании процессов, основываясь на вычислительной сложности задачи и средней стоимости пути от неё в сторону выходной вершины. Выбор исполнителя же основан на EFT – выбирается тот, время завершения на задачи на котором будет минимально. Вычислительная сложность алгоритма оценивается как $O(v^2 \times p)$, где v – число задач, p – число исполнителей.

Авторы [7] предлагают модификацию HEFT – алгоритм PEFT, его принципиальное отличие от прародителя заключается в использовании информации о наследниках задач, тем самым подразумевая использование направленного ациклического графа (DAG). PEFT берет во внимание суммарный вес наследников и исходящих из задачи ребер. Алгоритм, как и HEFT, двухфазовый, на фазе выбора исполнителя использует Lookahead схему из одноименного алгоритма [8]. Показывает результаты, превосходящие прародителя, но уступает ему в вычислительной скорости.

В [9] представлен Critical-Path-On-Processor (CPOP) алгоритм также основан на HEFT, но он распределяет приоритет задач, не только на наследниках, но и на предшественниках, учитывая затраты на переключение исполнителя. Основное же отличие состоит в фазе выбора исполнителя: выбирается critical path processor, который определяется как исполнитель, который минимизирует совокупное время выполнения задач на критическом пути (самом длинном пути от входного до выходного узла). Обобщенно – исполнитель, который выполнит задачи на критическом пути быстрее

всего. Сложность CPOP $O(v^2 \times p)$ – аналогична HEFT.

В работе [10] вводится Degree of Node First Task Scheduling (DONF)– алгоритм, который берет во внимание Weighted out Degree задачи, и с помощью этого ранжирует задачи. На фазе выбора процессора авторы модифицируют HEFT, введя новое понятие $EFT_{conflict}$ – самое раннее время завершения задачи на исполнителе, принимая во внимание возможные конфликты между исполнителями, принимая во внимания возможные задержи во времени начала выполнения задачи из-за предшествования. Подробнее данный алгоритм будет рассмотрен в главе 3.

Обучение с подкреплением (RL) – одна из парадигм машинного обучения, которая основывается на взаимодействии агентов со средой, совершая действия и получая награду в зависимости от полученных результатов. Deep RL – направление в RL, где также применяются сверточные нейросети.

Одна из фундаментальных работ по RL, [11], вводит Q-learning – алгоритм, в основе которого лежат марковский процесс принятия решений. На основе получаемой от среды отклика, алгоритм формирует таблицу полезности Q, что позволяет сформировать распределение вознаграждения в зависимости от действия агента, тем самым постепенно вырабатывая стратегию.

Advantage Actor-Critic [12] – алгоритм, который использует декомпозицию значения из таблицы Q-learning алгоритма на ценность состояние и ценность преимущества. Преимущество показывает, насколько совершенное агентом текущее действие лучше, чем друге действия при данном состоянии. Critic в этой системе учит значения преимущества, что позволяет модели судить о том, насколько одно действие может быть лучше других.

Google DeepMind, одна из ведущих лабораторий искусственного интеллекта, в [13] представляет алгоритм Deep Q Network, который является одним из наиболее популярных алгоритмов DRL, модифицированной версии Q-learning алгоритма. Авторы предложили новаторский подход для повышения стабильности и улучшения сходимости алгоритма путем ограничения выборки, на которой он обучается посредством Experience Replay – механизмом, вдохновленным биологическими процессами, который вы-

бирает некоторую случайную подвыборку из всех совершенных агентами действий, а не последними.

A3C [14] – модификация Advantage Actor-Critic алгоритма, новшеством которого является использование нескольких независимых агентов, которые действуют асинхронно, которые параллельно взаимодействуют с копиями оригинальной среды. После некоторого числа агенты сбрасывают свои параметры и берут параметры глобальной сети, критика. Так как момент сброса параметров не является фиксированным, возникает асинхронность, для коррекции которой авторы статьи разработали функцию временного сдвига, для коррекции расхождений.

Внедрение RL для составления расписаний находится на ранней стадии, но уже имеющиеся работы превосходят как классические алгоритмы теории расписаний, так и различные методы стохастической оптимизации.

В [15] авторы используют A3C алгоритм, параметризуя DAG для среды через набор эвристик, чтобы даль A3C больше информации о полученном наборе задач, тем самым существенно увеличивая качество построенных расписаний, превосходя как классические эвристики теории расписаний (First Come – First Serve, Shortest Job First), так и ML и RL методы без параметризации, используемые в решаемых авторами классе задач – управление распределенными вычислениями на высокопроизводительном кластере. Эвристики для параметризации включают в себя First Come – First Serve, Backfilling, Window-Sized BF, и прочие.

Работа [16] представляет другой подход – имея на вход множество DAG, авторы используют графовую нейронную сеть для параметризации DAG в виде набора векторов чисел, называемых embedding. Поддерживается несколько уровней embedding – для единичной вершины, для целого DAG, и параметризация включая всю информацию о среде. Далее embedding подается на вход RL алгоритму, чтобы он принял решение, какую из задач направить исполнителю. Авторы используют среду распределенных вычислений Apache Spark для тестирования алгоритма на производительность. Он также превосходит классические алгоритмы теории расписаний, применяемые в распределенных вычислениях: Shortest Job First, First In – First Out.

Глава 1. Обучение с подкреплением

1.1 Мотивация

Обучение с подкреплением стремительно набирает популярность, и в некоторых задачах модели уже превосходят способности человека [17]. За счет своей архитектуры RL модели невероятно гибкие: способность принимать решения и совершать действия основываясь на состоянии обозреваемой среды и посредством этого вырабатывать комплексные стратегии поведения позволяет внедрять их в практически любую предметную область, где среда может быть формализована, а пространство действий определено.

В работе [15] авторы производят сравнение производительности алгоритмов с разработанным ими фреймворком MARS

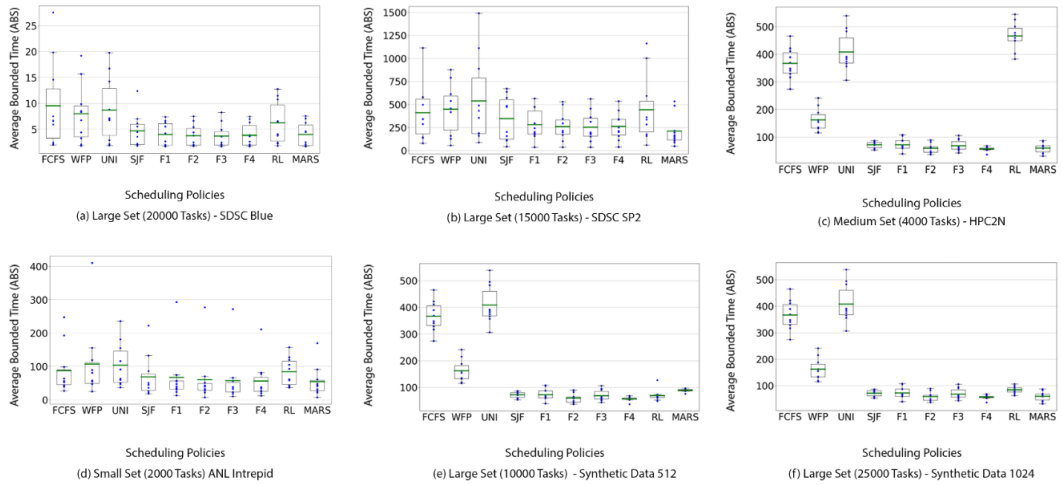


Рис. 1: Сравнение производительности MARS

Авторы сравнивают различные эвристики: классические FCFS, SJF, так и семейство нелинейных ML алгоритмов F1-F4 [18] в среде для высокопроизводительных вычислений на синтетических и реальных наборах данных - так называемых вычислительных графах (workflow traces), с учетом особенностей машин, на которых они были получены.

Name	CPU	Month(s)	Date
SDSC IBM-SP2	128	24	1998
SDSC IBM-Blue	1152	32	2000
High Performance Computing Center	240	42	2002
Argonne National Laboratory Intrepid	163840	8	2009
Synthetic_G001	256	12	2019
Synthetic_G002	1024	6	2019

Рис. 2: Наборы данных в MARS

MARS смог достичь улучшения производительности от 5% до 60% в терминах минимизации метрики Average Bounded Slowdown, определяемой как

$$ABS = -\max\left(\frac{T_\omega + T_r}{\max\{T_r, \tau\}}, 1\right)$$

где T_ω - время ожидания задачи, T_r - время её выполнения, τ - некоторая граница ожидания, задаваемая заранее.

Decima [16], используя отличный от MARS подход для параметризации среды, но тоже используя RL, также производит сравнение с FIFO в реализации фреймворка распределенных вычислений Apache Spark, модификацией Shortest-Job-First и Fair scheduler - простым алгоритмом который дает одинаковое время всем задачам для исполнения, и его оптимизированными для данной задачи вариациями.

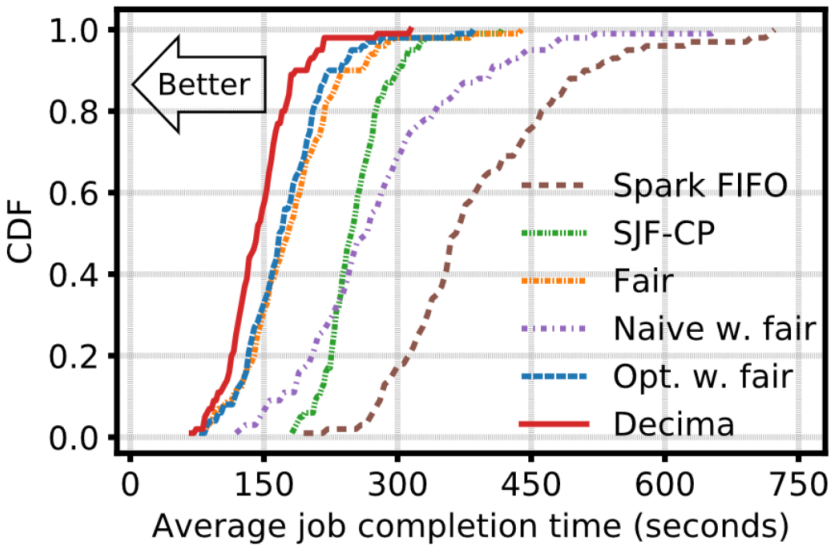
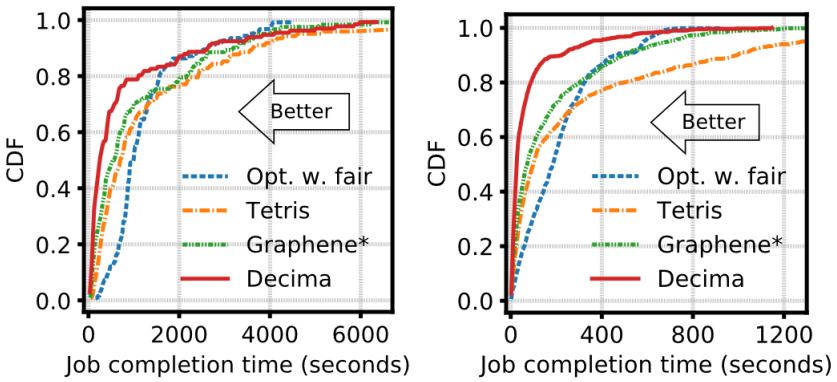


Рис. 3: Decima сравнение

Decima оказывает значительное улучшение результатов в смысле средней длительности исполнения работы по сравнению с применяемым в индустрии Apache Spark FIFO алгоритмом, как представлено на рисунке 3. На рисунке 4 представлено сравнение Decima с Graphene [19], фреймворке для составления расписаний, основанном на машинном обучении и Tetris [20].



(a) Industrial trace replay. **(b)** TPC-H workload.

Рис. 4: Decima сравнение

Приведенные выше примеры показывают, насколько эффективен может быть RL подход для решения класса задач теории расписаний в случае его комбинацией с правильной параметризацией среды для исполнения. В данной главе будет представлено описание базовой RL архитектуры и принципов работы алгоритмов, использованных при выполнении работы.

1.2 Концепция

Большинство методов машинного обучения, которые сейчас используются в индустрии, по своей сути являются предиктивными. Пытаясь найти комплексные зависимости, которые недоступны человеку, в данных, они пытаются предсказать будущие значения. Термин "обучение"в данном контексте означает, что чем больше данных будет подано в модель, тем качественнее модель будет выявлять зависимости, тем самым повышая качество предсказания. Существуют две категории машинного обучения: обучение с учителем и обучение без учителя. В первом случае модели требуется размеченный набор данных, т.е. исторические данные с известным для них исходом. Во втором же случае модель сама по себе ищет зависимости, не опираясь на исторически размеченные данные.

Обучение с подкреплением же, в свою очередь, отличается от этих двух видов алгоритмов машинного обучения. Модели с учителем и без, как правило, совершают предсказание для заданного набора данных единожды. RL напротив, был создан с целью имитировать более высокие когнитивные функции, в том смысле, что модели пытаются найти оптимальное действие (решение), которое будет выгодно в долгосрочной перспективе, даже если в краткосрочной перспективе приходится принимать нежеланные действия. Обучение с подкреплением не относится ни к обучению с учителем, ни к обучению без учителя - вместо этого в RL агент учится через взаимодействие со средой, записывая награды, которые он получает в процессе. Целью обучения является способность определять действия, которые при текущем состоянии среды будут вести к наибольшей совокупной награде. Так как алгоритмы обучения с подкреплением обучаются напрямую от среды, им не требуется огромный набор специализированных, заранее подготовленных и размеченных данных для обучения. Таким образом RL может служить общим алгоритмом обучения практически для любой среды.

Марковский процесс принятия решений формально описывает среду для обучения с подкреплением в случае, если среда целиком обозрима и текущее состояние целиком характеризует процесс.

1.2.1 Марковский процесс принятия решений

Состояние S_t обладает марковским свойством тогда и только тогда, когда

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, ..., S_t]$$

Т.е. состояние включает в себя всю релевантную информацию из истории. Для марковского состояния s и последующего состояния $s^{'}$ вероятность перехода между состояниями определяется как

$$\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s]$$

Матрица вероятностей переходов $\mathcal{P}$ определяет вероятности перехода из всех состояний s во все состояния $s^{'}$

$$\mathcal{P}_{ss'} = \begin{vmatrix} \mathcal{P}_{11} & ... & \mathcal{P}_{1n} \\ ... & & ... \\ \mathcal{P}_{n1} & ... & \mathcal{P}_{nn} \end{vmatrix}$$

где сумма элементов каждой строки из матрицы равна единице.
Можно определить марковский процесс как последовательность случайных состояний $\mathcal{S}_1, \mathcal{S}_2, ...$ обладающих марковским свойством. Более формально: марковский процесс это кортеж $\langle \mathcal{S}, \mathcal{P} \rangle$, где

- $\mathcal{S}$ - множество состояний
- $\mathcal{P}$ - матрица переходных вероятностей, для которой выполнено $\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s]$

По аналогии можно ввести марковский процесс с вознаграждением: это кортеж $\langle \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, где

- $\mathcal{S}$ - множество состояний
- $\mathcal{P}$ - матрица переходных вероятностей, для которой выполнено $\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s]$
- $\mathcal{R}$ - это функция награды, $\mathcal{R}_s = \mathbb{E}[R_{t+1}|S_t = s]$

- γ - коэффициент дисконтирования, $\gamma \in [0, 1]$

Тогда можно определить совокупную награду G_t как

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Коэффициент дисконтирования вводится чтобы избежать бесконечных совокупных наград в циклических марковских процессах.
Функция полезности состояния $v(s)$ определяется как математическое ожидание совокупных наград начиная с состояния s

$$v(s) = \mathbb{E}[G_t | S_t = s]$$

Функция полезности состояния может быть декомпозирована на непосредственную награду R_{t+1} и дисконтированную полезность последующего состояния $\gamma v(S_{t+1})$

$$v(s) = \mathbb{E}[G_t | S_t = s] = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... | S_t = s]$$

после некоторых преобразований становится возможно получить уравнение Беллмана

$$v(s) = \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) | S_t = s]$$

его можно переписать в виде

$$v(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v(s')$$

или в матричной форме $v = \mathcal{R} + \gamma \mathcal{P} v$, где

$$\begin{vmatrix} v(1) \\ ... \\ v(n) \end{vmatrix} = \begin{vmatrix} \mathcal{R}_1 \\ ... \\ \mathcal{R}_n \end{vmatrix} + \gamma \begin{vmatrix} \mathcal{P}_{11} & ... & \mathcal{P}_{1n} \\ ... & & ... \\ \mathcal{P}_{11} & ... & \mathcal{P}_{nn} \end{vmatrix} \begin{vmatrix} v(1) \\ ... \\ v(n) \end{vmatrix}$$

Уравнение Беллмана имеет прямое решение $v = (I - \gamma \mathcal{P})^{-1} \mathcal{R}$, но его приме-

нение возможно только для небольших марковских процессов с наградой. Есть несколько итеративных способов решения для больших марковских процессов, такие как:

- Динамическое программирование
- Метод Монте-Карло
- Обучение на временной разности

Теперь можно ввести марковский процесс принятия решений: кортеж $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, где

- $\mathcal{S}$ - конечное множество состояний
- $\mathcal{A}$ - конечное множество действий
- $\mathcal{P}$ - матрица переходных состояний, для каждого элемента которой верно $\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$
- $\mathcal{R}$ - функция наград, для которой верно $\mathcal{R}^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$
- γ - коэффициент дисконтирования, $\gamma \in [0, 1]$

Стратегией (политикой) π будем называть распределение действий при заданных состояниях

$$\pi(a|s) = \mathbb{P}[A_t = a | S_t = s]$$

Стратегия целиком описывает поведение агента, зависит только от текущего состояния и стационарна.

При заданном марковском процессе принятия решений $\langle S, \mathcal{A}, P, \mathcal{R}, \gamma, \rangle$ и политикой π, заданной последовательности состояний $S_1, S_2, ...$ являющейся марковским процессом $\langle S, P^\pi \rangle$, и заданной последовательности наград-состояний $S_1, R_1, S_2, R_2, ...$, являющейся марковским процессом с вознаграждением $\langle S, P^\pi, R^\pi, \gamma, \rangle$, можно определить вероятность перехода из состояния s в s' и награду как

$$P^\pi_{s,s'} = \sum_{a \in A} \pi(a|s) P^a_{ss'}$$

$$R_s^\pi = \sum_{a \in A} \pi(a|s) R_s^a$$

Функция полезности состояния при заданной политике π может быть декомпозирована как

$$v_\pi(s) = \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s]$$

функция полезности действия $q_\pi(s, a)$ марковского процесса принятия решения это математическое ожидание функции совокупной награды начиная с состояния s, выполняя действие a и следуя стратегии π:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

По аналогии с функцией полезности состояния, функция полезности действия может быть декомпозирована как

$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a]$$

По аналогии со случаем без стратегии, данный вариант декомпозиции является уравнением ожидания Беллмана

$$v_\pi = R^\pi + \gamma P^\pi v_\pi$$

и имеет прямое решение

$$v_\pi = (I - \gamma P^\pi)^{-1} R^\pi$$

Введем понятия оптимальной функции полезности состояния $v_*(s)$ - максимизирующая полезность относительно всех стратегий

$$v_*(s) = \max_\pi v_\pi(s)$$

Оптимальная функцией полезности действия $q_*(s, a)$ - максимизирующая

полезность действия относительно всех стратегий

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

Для сравнения стратегий вводится отношение частичного порядка:

$$\pi \geqslant \pi' \iff v_\pi(s) \geqslant v'_\pi(s), \forall s$$

в [21] доказана теорема, которая гласит

- Существует оптимальная стратегия π_*, для которой верно $\pi_* \geqslant \pi, \forall \pi$
- На всех оптимальных стратегиях достигается оптимальная функция полезности состояния, $v_{\pi_*}(s) = v_*(s)$
- На всех оптимальных стратегиях достигается оптимальная функция полезности действия, $q_{\pi_*}(s,a) = q_*(s,a)$

Уравнение оптимальности Беллмана нелинейно, и в общем случае может не иметь аналитического решения, поэтому используются различные итеративные методы для его решения:

- Policy iteration - итерация по стратегии
- Value iteration - итерация по полезности
- Q-learning

Первые два подхода будут рассмотрены в общем виде, и, в целом будут рассматриваться с точки зрения решения задачи оптимизации управления в смысле принятия наиболее выгодного решения при заданном состоянии, так как обычно неизвестна структура марковского процесса, который мог бы смоделировать управляемый процесс. Частично обозримые, эргодические и содержащие бесконечный горизонт планирования марковские процессы принятия решений выходят за рамки рассмотрения концепций, на которых основан RL, с ними можно ознакомиться в [21].

1.2.2 Policy iteration и Value iteration

Подход итерации по стратегии состоит из двух этапов: оценка стратегии как получение значения v_π при заданной стратегии π, и генерация стратегии $\pi' \geqslant \pi$, улучшающей показатель целевой функции (в силу отношения частичного порядка для стратегий)

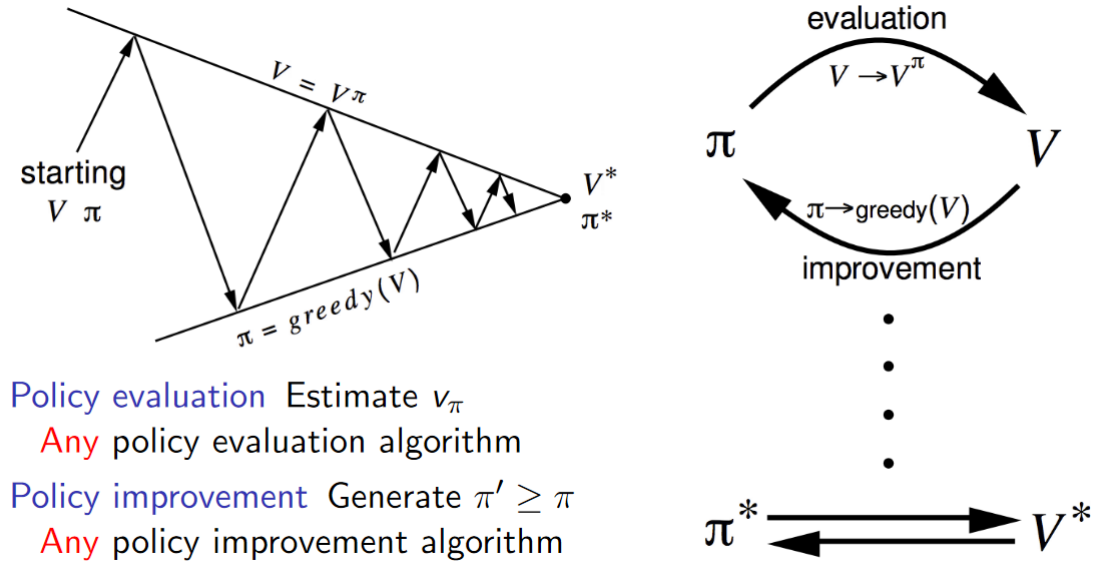


Рис. 5: Общая схема работы метода итерации по стратегии [21]

В [??] приведен принцип оптимальности, который гласит, что стратегия $\pi(a|s)$ достигает оптимальной функции полезности из состояния s, $v_\pi(s) = v_*(s) \iff$

- $\forall s'$ достижимо из s
- π достигает оптимального значения из состояния s', $v_\pi(s') = v_*(s')$

На рисунке, приведенном выше, представлена схема работы подхода итерации по стратегии: обычно стратегии инициируют случайным образом, далее происходит реализация стратегии, вычисляется полезность состояния; после этого происходит улучшение стратегии, часто используется жадный подход. Для детерминированной стратегии это выглядит следующим образом:

- задана детерминированная стратегия, т.е. $a = \pi(s)$

- улучшение стратегии жадным образом $\pi'(s) = \underset{a \in A}{argmax} q_\pi(s, a)$
- Происходит улучшение стратегии из любого состояния s на один шаг

$$q_\pi(s, \pi'(s)) = \max_{a \in A} q_\pi(s, a) \geqslant q_\pi(s, \pi(s))) = v_\pi(s)$$

- Происходит улучшение функции полезности состояния $v'_\pi(s) \geqslant v_\pi(s)$
- Если улучшение прекращается, то

$$q_\pi(s, \pi'(s)) = \max_{a \in A} q_\pi(s, a) = q_\pi(s, \pi(s)) = v_\pi(s)$$

Если улучшение прекратилось, то полученная функция полезности состояния удовлетворяет уравнению Беллмана [21] об оптимальности

$$v_\pi(s) = \max_{a \in A} q_\pi(s, a)$$

и, следовательно, $v_\pi(s) = v_*(s), \forall s \in S$. Это означает, что π - оптимальная стратегия.

Value iteration, в отличие от policy iteration, данный подход не требует оценки стратегии, тем самым снижая вычислительную сложность. Процесс проходит аналогично, но выбор действия происходит напрямую через получаемую награду.

$$v_{k+1}(s) = \max_{a \in A}(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s'))$$

$$v_{k+1}(s) = \max_{a \in A} R^a + \gamma P^a v_k$$

Рассмотренные итеративные подходы обычно решаются посредством динамического программирования для конечных марковских процессов принятия решений, т.е. содержащих конечное множество наград, состояний и действий, однако возможно их расширение на случаи с бесконечными множествами за счет применения различных подходов к аппроксимации .

1.2.3 Importance Sampling

Существует дилемма при решении задач управления с точки зрения марковских процессов принятия решения: необходимо построить распределение действий над состояниями достигающих оптимального результата, тем самым построив оптимальную стратегию, однако для её построение необходимо принимать не оптимальные действия с целью исследования среды чтобы, по-сути, найти оптимальную стратегию.

Для решения данной дилеммы используют два подхода: on-policy и off-policy learning. В on-policy ищутся значения для близкой к оптимальной стратегии, которая способна к исследованию среды. Более распространенный в RL метод для управления - off-policy, в котором исползьуется два типа стратегий - целевая (оптимальная) и поведенческая, которая используется для непосредственного получения действий. Off-policy методы достаточно разнообразны, и могут использовать в качестве поведенческой стратегии набор данных человеческого поведения или классического не обучающегося управляющего устройства.

Рассмотрим ситуацию, когда необходимо вычислить v_π или q_π используя поведенческую стратегию b, при этом $b \neq \pi$, где π - целевая стратегия. И целевая, и поведенческая стратегии в данном примере считаются определенными и известными. Почти все off-policy методы используют importance sampling, подход для вычисления условного математического ожидания значений из одного распределения при заданном другом распределении. Данный подход применяется для определения веса для суммарной награды, который характеризует условную вероятность реализации определенной последовательности действий и состояний (траектории), следуя целевой стратегии при заданной поведенческой стратегии.

Формально, пусть задано начальное состояние s_t, тогда вероятность реализации траектории $a_t, s_{t+1}, a_{t+1}, ..., s_T$ при стратегии π

$$Pr\{a_t, s_{t+1}, a_{t+1}, ..., s_T | s_t, a_{t:T-1}\ \pi\} = \prod_{k=t}^{T-1} \pi(a_k|s_k) p(s_{k+1}|s_k, a_k)$$

где p - вероятность перехода между состояниями. В таком случае, относительная вероятность реализации заданной траектории при целевой и поведенческой политике можно представить как

$$p_{t:T-1} = \frac{\prod_{k=t}^{T-1} \pi(a_k|s_k) p(s_{k+1}|s_k, a_k)}{\prod_{k=t}^{T-1} b(a_k|s_k) p(s_{k+1}|s_k, a_k)} = \prod_{k=t}^{t-1} \frac{\pi(a_k|s_k)}{b(a_k|s_k)}$$

таким образом вероятность реализации траектории не зависит от переходных вероятностей марковского процесса принятия решения, в силу чего вероятность реализации зависит только от стратегий и самой траектории.

Для решения оптимизационной задачи в марковском процессе принятия решений обычно необходимо вычислить суммарную награду G_t при начальном состоянии s, однако, следуя поведенческой стратегии возможно вычислить только

$$\mathbb{E}[G_t | s_t = s] = v_b(s)$$

которая не может быть просто усреднена [21] с целью получения v_π. Для этого используется importance sampling, относительная вероятность $p_{t:T-1}$ позволяет свести математическое ожидание поведенческой стратегии к целевой:

$$\mathbb{E}[p_{t:T-1} G_t | s_t = s] = v_\pi(s)$$

Off-policy подход в совокупности с importance sampling используется для решения уравнения оптимальности Беллмана, в совокупности с Temporal Difference Learning или методом Монте-Карло.

1.3 Temporal Difference Learning

Одной из особенностей методов Temporal Difference (TD) является то, что они могут обучатся напрямую на данных предыдущих действий, не затрагивая при этом переходные вероятности марковского процесса принятия решений. TD обновляет прогнозируемые (ожидаемые) значения не дожидаясь завершения траектории, используя bootstrap.

Обновление значения для заданного состояния $v(s_t)$ в TD для t происходит сразу на следующем шаге, используя награду r_{t+1} и оценку $v(s_{t+1})$ как

$$v(s_t) \leftarrow v(s_t) + \alpha[r_{t+1} + \gamma v(s_{t+1}) - v(s_t)]$$

Данный подход называется TD(0) - так как используются значения только следующего шага, однако возможно обобщение для более чем одного шага. В [21] показано, что

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|s_t = s] \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma G_{t+1}|s_t = s] \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma v_\pi(s_{t+1})|s_t = s] \end{aligned}$$

так как TD(0) использует уже частично известные значения, он считается bootstrap алгоритмом. Алгоритм использует для обновления значений ошибку временной разницы (TD error), которая показывает разницу между оценкой $v(s)$ с более качественной оценкой $r_{t+1} + \gamma v(s_{t+1}$, в литературе обозначается как

$$\delta_t = r_{t+1} + \gamma v(s_{t+1}) - v(s)$$

данная ошибка используется во множестве алгоритмов обучения с подкреплением.

TD(0) используется для оценки функции v_π при заданной стратегии π, для решения задачи управления в смысле поиска оптимальной стратегии используются алгоритмы SARSA и Q-learning.

1.3.1 SARSA

Для применения Temporal Difference Learning в задачах управления необходимо оценить $q_\pi(s, a)$ для текущей стратегии π для всех состояний s и действий a. Так как алгоритм принадлежит к семейству on-policy, используется только одна стратегия. По аналогии с TD(0):

$$q(s_t, a_t) \leftarrow q(s_t, a_t) + \alpha[r_{t+1} + \gamma q(s_{t+1}, a_{t+1}) - q(s_t, a_t)]$$

такое обновление происходит для каждого перехода из не конечного состояния s_t, в противном случае полагается $q(s_{t+1}, a_{t+1}) = 0$.

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Рис. 6: Псевдокод алгоритма SARSA [21]

SARSA, как и Q-learning используют ϵ-жадные (ϵ-greedy) стратегии: выбор действия на каждом из шагов происходит между действием, максимизирующим q(s,a) с вероятностью $1 - \epsilon$, и случайным действием с вероятностью ϵ. Данный подход позволяет соблюсти баланс между исследованием среды и совершением уже хорошо изученных действий. Сходимость алгоритма доказана при использовании как ϵ-greedy, так и ϵ-soft стратегий, что представлено в [21]

1.3.2 Q-learning

В отличие от SARSA, Q-learning не следует стратегии, а напрямую максимизирует $q(s, a)$. Благодаря данному упрощению сходимость алгоритма хорошо изучена [11]. Стоит отметить, что стратегия всё равно влияет на то, какие пары действий и состояний будут обновлены в ходе обучения. Основным отличием от SARSA является механизм обновления q:

$$q(s_t, a_t) \leftarrow q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - q(s_t, a_t)]$$

по аналогии с SARSA, для выбора действия используется ϵ-greedy стратегия, но в то же время обновление значения $q(s, a)$ происходит с максимизацией его напрямую. Псевдокод алгоритма представлен ниже.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Рис. 7: Псевдокод алгоритма Q-learning [21]

Оба метода, SARSA и Q-learning представляют собой так называемые табличные методы: происходит обновление таблицы, содержащей q значения для всех пар состояние-действие, до тех пор, пока не будет выполнен критерий останова, заполняя данную таблицу, из которая по своей сути и будет являться решением уравнения оптимальности Беллмана. Но в случаях, когда размерность пространства действий и состояний велика, не всегда возможно использовать таблицы. Для этого созданы Policy Gradient

алгоритмы и модификации Q-learning, такие как Deep-Q-Netwok (DQN).

1.4 Policy Gradient

До текущего момента рассматривались алгоритмы, которые строили распределение состояний-действий Q, и выбирали действия на основе построенной оценки. Существует другой подход, позволяющий отойти от состояний-действий и выбирать действия, не основываясь на оценке их полезности. Для этого вводится понятие параметрической стратегии, которая может выбирать действия не обращаясь к Q, используя полезность только при оценке параметров стратегии. Пусть $\theta \in \mathbb{R}^n$ - вектор параметров стратегии, тогда выражение

$$\pi(a|s,\theta) = Pr\{a_t = a|s_t = s, \theta_t = \theta\}$$

определяет вероятность выбора действия a в момент времени t, если среда находится в состоянии s с набором параметров θ. Если алгоритм использует полезность состояния, то функцию полезности можно задать как $\hat{v}(s,w)$, где w - вектор весов состояний, $w \in \mathbb{R}^{n'}$

В данном семействе алгоритмов поиск стратегии происходит основываясь на градиенте скалярной меры полезности $J(\theta)$ относительно параметров стратегии. Цель - максимизация полезности, и с этой целью используется градиентный подъем по $J(\theta)$:

$$\theta_{t+1} = \theta_t + \alpha\nabla\hat{J}(\theta_t)$$

где $\nabla\hat{J}(\theta_t) \in \mathbb{R}^n$ - стохастическая оценка, математическое ожидание которого аппроксимирует градиент меры полезности относительно θ. Все методы, использующие данный метод называются policy gradient методы, а алгоритмы, которые аппроксимирует помимо меры полезности функцию полезности v называются actor-critic методы, где actor относится к найденной стратегии, а critic относится к оцененной функции полезности состояния. В рамках данной работы будут рассмотрены только эпизодические случаи.

В policy gradient алгоритмах, стратегия может быть параметризована любым образом, с условием что $\pi(a|s,\theta)$ дифференцируема относительно своих параметров и $\nabla\pi(a|s,\theta)$ существует и конечна для $\forall s \in S, \forall a \in A$ и $\theta \in R^n$. На практике [21], чтобы пространство было исследовано, на стратегии накладывается ограничение об их недетерминированности, $\pi(a|s,\theta) \in (0,1)\forall s,a,\theta$.

Если пространство действий дискретно, одним из способов параметризации является введение численных предпочтений $h(s,a,\theta) \in R$ для каждой пары действие-состояние. Действия с более высоким предпочтением выбираются с большей вероятностью. Как пример, возможно использовать soft-max [21]:

$$\pi(a,|s,\theta) = \frac{e^{h(s,a,\theta)}}{\sum_b e^{h(s,b,\theta)}}$$

где знаменатель необходим для того, чтобы вероятности во всех состояниях в сумме равнялись единице. Сами численные предпочтения могут быть параметризованы отдельно, напирмер могут быть вчислены через сверточные нейронные сети, или в линейном виде как

$$h(s,a,\theta) = \theta^T x(s,a)$$

где $x(s,a) \in \mathbb{R}^n$ - параметризация через полиномы, радиальный базис и прочее [21]

Для случая оптимизации по эпизодам, авторы [21] доказывают теорему о policy gradient, которая связывает функцию ценности состояния для конкретного не случайного состояния s_0 следуя стратегии π с набором параметров θ с числовой мерой полезности как

$$J(\theta) = v_{\pi_\theta}(s_0)$$

в результате чего можно получить представление для $\nabla J(\theta)$ через градиент стратегии как

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta)$$

где символ $\propto$ в данном случае обозначает пропорциональность, и в эпизодическом случае значением пропорциональности является средняя длина эпизода (длина траектории), а в непрерывном случае данное выражение переходит в равенство. В данном случае μ - аналогичное on-policy методам распределение при π.

1.4.1 REINFORCE

Используя теорему о policy gradient, можно получить выражение для шага градиентного подъема. Отношение пропорциональности будет компенсировано засчет параметра α - шага обучения. В [22] показано, что

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta) = \mathbb{E}_\pi[\sum_a q_\pi(s_t,a) \nabla \pi(a|s_t,\theta)$$

в таком случае, выражение для стохастического градиентного подъема примет вид

$$\theta_{t+1} = \theta_t + \alpha \sum_a \hat{q}(s_t,a,w) \nabla \pi(a|s_t,\theta)$$

где $\hat{q}$ это аппроксимация q_π полученная в ходе обучения. Этот подход включает в себя обновление по всем действиям, однако, более эффективно свести выражение к зависимости от a_t - действия, предпринятого в момент времени t [22]:

$$\begin{aligned}
\nabla J(\theta) &\propto \mathbb{E}_\pi[\sum_a \pi(a,|s_t,\theta) q_\pi(s_t,a) \frac{\nabla \pi(a|s_t,\theta)}{\pi(a|s_t,\theta)}] \\
&= \mathbb{E}_\pi[q_\pi(s_t,a_t) \frac{\nabla \pi(a|s_t,\theta)}{\pi(a|s_t,\theta)}] \\
&= \mathbb{E}_\pi[G_t \frac{\nabla \pi(a|s_t,\theta)}{\pi(a|s_t,\theta)}]
\end{aligned}$$

где G_t - суммарная награда. В таком случае, шаг градиентного подъема примет вид

$$\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla \pi(a_t|s_t, \theta)}{\pi(a_t|s_t, \theta_t)}$$

Каждое обновление шага пропорционально произведению суммарных наград с вектором, представляющим собой градиент вероятности осуществления выбранного действия деленного на вероятность выбрать данного действия. Этот вектор указывает направление которое наиболее сильно увеличивает вероятность повторить действие a_t при s_t. Такой способ обновление параметров стратегии позволяет двигаться в направлении максимизации суммарной награды [22]. Стоит отметить, что в псевдокоде приводится $\nabla ln \pi(a_t|s_t, \theta_t)$ - классический прием упрощающий работу с вероятностями при вычислениях.

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\quad (G_t)$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Рис. 8: Псевдокод алгоритма REINFORCE [21]

1.4.2 Actor-Critic

Основное отличие Actor-Critic подхода от REINFORCE заключается в использовании параметризованного значения приближения функции полезности состояния $\hat{v}(s_t, w)$:

$$\theta_{t+1} = \theta_t + \alpha(G_{t:t+1} - \hat{v}(s_t, w))\frac{\nabla\pi(a_t|s_t,\theta_t)}{\pi(a_t|s_t,\theta_t)}$$
$$= \theta_t + \alpha(r_{t+1} - \gamma\hat{v}(s_{t+1}, w) - \hat{v}(s_t, w))\frac{\nabla\pi(a_t|s_t,\theta_t)}{\pi(a_t|s_t,\theta_t)}$$
$$= \theta_t + \alpha\delta_t\frac{\nabla\pi(a_t|s_t,\theta_t)}{\pi(a_t|s_t,\theta_t)}$$

по аналогии с TD(0) алгоритмом, в Actor-Critic используется TD error, которая, на самом деле, является разностью представления q и v через уравнение Беллмана [11], и, в свою очередь показывает ценность совершения конкретного действия при заданном состоянии, и, градиент в AC направлен в сторону максимизации данного значения

One-step Actor–Critic (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
  Initialize S (first state of episode)
  $I \leftarrow 1$
  Loop while S is not terminal (for each time step):
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\delta\nabla\hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I\delta\nabla\ln\pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

Рис. 9: Псевдокод алгоритма Actor-Critic [21]

Policy Gradient подход лег в основу многих алгоритмом обучения с подкреплением с применением нейронных сетей, которые были использованы при выполнении данной работы. Их рассмотрению посвящены следующие два раздела.

1.5 DQN

Deep Q Learning - алгоритм, представленный Google DeepMind [13], который позволяет преодолеть ограничения, накладываемые табличным

подходом - не всегда получится хранить таблицу значений Q, особенно при больших размерностях пространства действий и/или состояний. В основе алгоритма лежит идея аппроксимации $Q(s, a)$ посредством нейронной сети, чтобы отойти от идеи хранения значений для каждой пары действие-состояние.

Использование нейронных сетей, в решении задачи обучения с учителем для построения аппроксимации используются размеченные данные, и для них существует требование, что данные должны быть независимы и одинаково распределенные. Обучение обычно происходят на подмножествах данных (batch), и с этой позиции требования принимают вид:

- Подмножества составляются из данных случайным образом, и каждое из подмножеств должно иметь одинаковое распределение данных
- Данные в рамках одного подмножества не зависят друг от друга в рамках данного подмножества

Нарушение представленных выше условий в данных, используемых для обучения, скорее всего приведет к переобучению - потери моделью возможности к восприятию новых данных надлежащим образом. Дополнительно стоит отметить, что в классическом случае обучения с учителем разметка данных для каждого отдельного образца не изменяется со временем. В обучении с подкреплением же, наоборот, если рассматривать пары действие-состояние и значение Q как разметку исследуемой среды, изменения постоянны. Помимо этого, значение Q зависит само от своего значения на предыдущих итерациях, что делает целевой показатель нестационарным. Существует также проблема корреляции между траекториями (последовательностями действий-состояний, как описывалось в предыдущем разделе), во время итерации во время обновления параметры Q обновляются с целью приближения к истинным значениям, но если пространство состояний крайне велико и состояния схожи друг с другом, то обновленное значение Q будет выше чем предыдущее, что повлечет за собой увеличение целевого показателя не зависимо от полезности совершенного действия на предыдущем этапе [13]. В совокупности эти факторы делают процесс обучения

крайне нестабильным с точки зрения нейронных сетей.

Для решение данной проблемы используется два подхода: целевая нейросеть (target network) и expirience replay.

- Целевая нейросеть - используется две нейросети θ и θ', базовая и целевая соответственно. Модель получает значения Q для заданного состояния из целевой сети, в то время как базовая включает в себя все обновления Q во время обучения. После определенного числа итераций обучения сети синхронизируются (т.е происходит копия весов из базовой сети в целевую). Целью данного подхода является фиксация значений Q, чтобы временно зафиксировать цель

- Expirience replay - для повышения стабильности работы и снижения зависимости между данными, последние n переходов из состояния в состояние сохраняются, и из них случайным образом выбирается m (обычно 32) для обучения нейросети

Функция потерь представляет собой

$$L(\theta) = \mathbb{E}_{s,a,s',r\sim D}[(\underbrace{r_t + \gamma \max_{a'} Q(s', a'; \theta_i')}_{y_i} - Q(s, a; \theta_i))^2]$$

где y_i - значения, полученные из целевой сети θ'.

DQN относится к model-free, т.е. получает значения напрямую от среды, и off-policy - использует жадный подход к выбору действий $\max_a Q(s, a, \theta)$, обычно используется ϵ-жадная стратегия для обучения, чтобы обеспечить баланс между исследованием среды и выбором оптимальных действий.

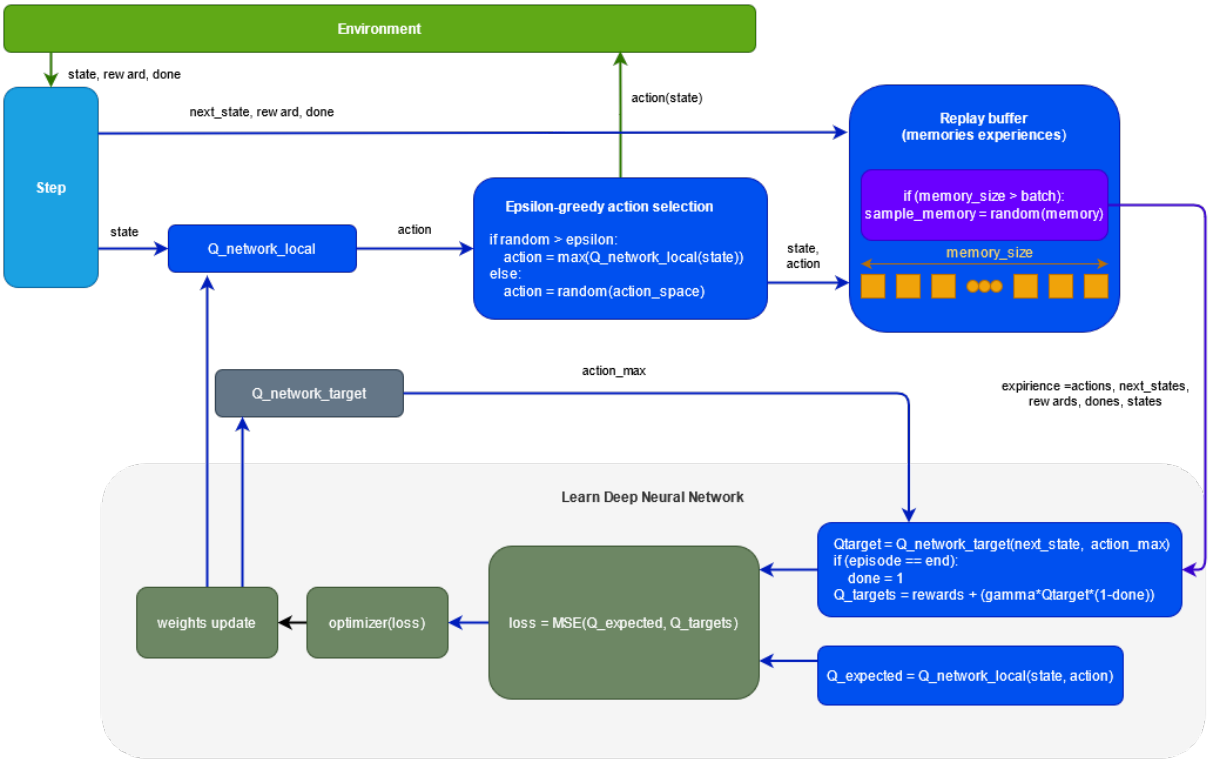


Рис. 10: Архитектура DQN

На рисунке выше представлена схема работы модели и её взаимодействия со средой. Replay buffer обозначает Expirience replay механизм, а обновление весов происходит не на каждой итерации для целевой сети. Параметр Done сигнализирует о переходе в терминальное состояние. Модель получает от среды состояние, награду, сигнал об окончании, далее совершается шаг - базовая сеть выбирает действие, которое осуществляется посредством ϵ-жадной стратегии. Далее состояние, награда, действие записывается в Expirience replay, как и следующее состояние. Из Expirience replay происходит выбор набора пар действие-состояние из уже выполненных действий, и происходит обучение - минимизация функции потерь от показателей целевой сети и образца из памяти, после чего происходит обновление весов в базовой сети. Псевдокод алгоритма представлен ниже.

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode = 1, M **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For t = 1,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

  End For

End For

Рис. 11: Псевдокод алгоритма DQN [13]

DQN изначально был представлен как алгоритм, решавший задачу управления в компьютерных играх, где без дополнительных модификаций в 6 играх превзошел все представленные до него подходы в RL, а также в трех из них превзошел эксперта. Как можно видеть, DQN является расширением классического Q-learning посредством нейросетей, что позволило алгоритму превзойти ограничения табличного подхода.

1.6 A3C

Asynchronous Advantage Actor-Critic (A3C) - модификация Advantage Actor-Critic (A2C), который в свою очередь основан на AC, с целью улучшения сходимости алгоритма, за счет использования нескольких параллельных агентов и сред, чьи результаты потом усредняются в глобальной сети.

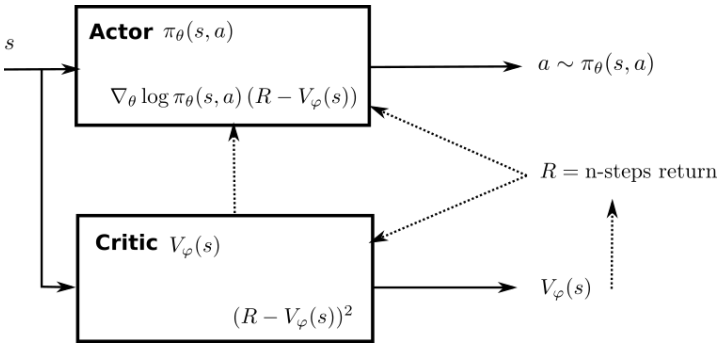


Рис. 12: Схема работы A2C

A2C отличается от AC только тем, что оценивает функцию ценности состояния на несколько шагов вперед: actor вычисляет $\pi_\theta(s,a)$, s, $critic$v$_\phi(s)$ состояния s. Схема работы A2C:

- Получить множество (batch) переходов (s,a,r,s') используя текущую стратегию π_θ
- Для каждого из полученных состояний actor подсчитывает сумму следующих n наград, a critic вычисляет функцию полезности состояния для n полученных состояний

$$R_t = \sum_{k=0}^{n-1} \gamma^k r_{t+k+1} + \gamma^n v_\varphi(s_{t+n+1})$$

- Обновить стратегию actor аналогично AC

$$\nabla_\theta J(\theta) = \sum_t \nabla_\theta \log \pi_\theta(s_t, a_t)(R_t - v_\phi(s_t))$$

- Обновить critic для минимизации TD error

$$L(\varphi) = \sum_t (R_t - v_\varphi(s_t))^2$$

Для actor градиент будет обновляться как:

$$\theta \leftarrow \theta + \nabla_\theta \log \pi_\theta(s_k, a_k)(R - v_\varphi(s_k))$$

в то время как для critic:

$$\varphi \leftarrow \varphi + \nabla_\varphi (R - v_\varphi(s_k))^2$$

В целом A3C представляет собой много параллельно работающих A2C моделей, которые через определенное время передают параметры в глобальную сеть, которая асинхронно обновляется, также как и в обычном A2C. Для существует несколько подходов к работе с асинхронными сетями, обычно используется HogWild! [23].

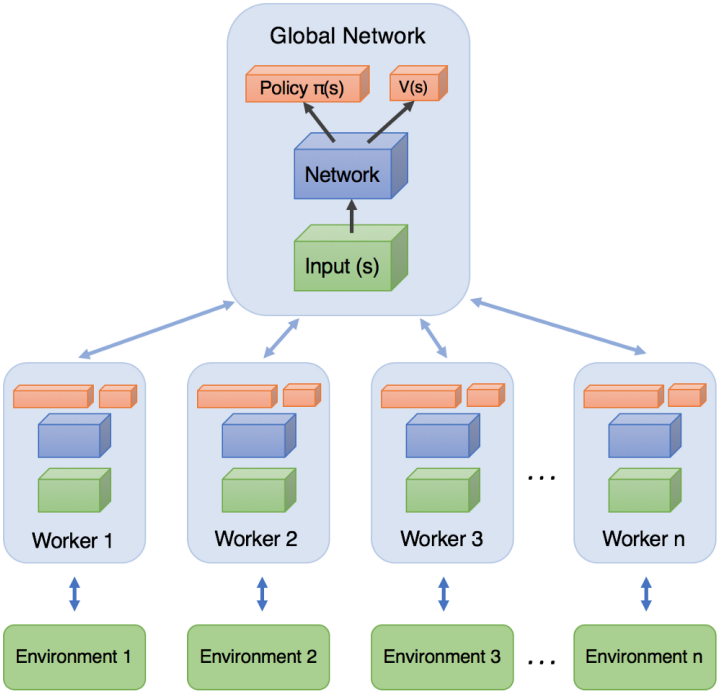


Рис. 13: Архитектура A3C

Существует несколько существенных различий между DQN и A3C:

- DQN использует experience replay для решения проблемы зависимости входных данных, в то время как A3C использует параллельные среды
- A3C относится к on-policy алгоритмам: выученная стратегия должна быть использована для исследования среды, в то время как DQN использует жадный подход
- DQN использует целевую сеть для борьбы с проблемой нестационарных Q, в то время как A3C использует полезность состояния и преимущество, которые куда более стабильны [14]
- A3C может быть использован для непрерывных сред за счет параметризации стратегий, в отличие от DQN

A3C превышает DQN по эффективности, однако куда более требователен к вычислительным ресурсам. Оба алгоритма были успешно применены в рамках выполнения данной работы.

Глава 2. Данные и распределенные вычисления

В данной главе пойдет речь о получении данных для обучения модели и проведения экспериментов, рассматривается алгоритм, выбранный для сравнительных испытаний а также рассматриваются программные продукты, используемые для современного распределенного обучения с подкреплением.

2.1 Генерация данных

В статье [5], выбранной для сравнения результатов построенного расписаний, используется программа Daggen [24] – генератор случайных графов, который также использовался авторами других алгоритмов, таких как HEFT [6], PEFT [7], HSIP [25], популяризовавшие подход к построению расписаний на графах используя топологические особенности графов.

Daggen – ПО для генерации случайных синтетических ациклических направленных графов (DAG) задач с весами, с целью тестирования графовых алгоритмов теории расписаний. Вершины в графах генерируемом Daggen могут быть нескольких видов:

- Корневая – уникальная искусственная вершина, с нулевой стоимостью, без предшественников, а её наследниками являются входные вершины DAG
- Вычислительная – псевдовычислительная задача, её вес отражает затраты в терафлопс на её исполнение
- Конечная – уникальная искусственная вершина с нулевой стоимостью, без наследников, её предшественниками являются выходные вершины в DAG

Возможно использование параметров:

- n – количество вычислительных вершин в графе
- mindata – минимальная вычислительная сложность

- maxdata – максимальная вычислительная сложность
- fat – ширина DAG в смысле количества вычислительных задач, которые могут быть выполнены одновременно. Низкое значение ведет к генерации цепи, которое будет отображать низкую возможность к параллелизму у моделируемого процесса, когда высокое – наоборот
- density – параметр, определяющий количество зависимостей между вычислительными вершинами
- ccr – ценность коммуникации – определяет соотношение веса на ребрах к весам на вершинах графа, с целью определять насколько задача требовательна к передаче информации или к вычислительной мощности
- regular – распределение количества задач (вершин) по различным уровням DAG
- jump – количество уровней DAG, между задачами которых могут быть ребра

Генерация происходит следующим образом:

1. Генерируются задачи
    - Определяется предпочтительное число задач на одном уровне p (DAG генерируемый Daggen можно воспринимать в виде дерева) как $e * fat * \log(n)$
    - Определяется число задач на уровень с использованием параметра regular: выбирается случайное число из промежутка $[regular * perfect; (1 - regular) * perfect]$
    - Каждой из вершин присваивается вес, случайным образом завися от mindata, maxdata, ccr
2. Генерируются зависимости на основе jump и density, и им присваиваются веса с учетом ccr

Daggen реализован на C, может генерировать графы с большим числом нод достаточно быстро. Для выполнения работы были сгенерированы графы со следующими параметрами, соответствуя описанному в [5]:

- $n \in 1000, 4000, 16000, 64000, 128000$
- $fat \in 0.2, 0.5$
- $density \in 0.1, 0.4, 0.8$
- $regularity \in 0.2, 0, 8$
- $jump \in 2, 4$
- $ccr \in 0.2, 0.8, 4, 8$

В работе были использованы только размером 1000 вершин из-за высокой вычислительной сложности у RL моделей.

2.2 Алгоритм для сравнения

Был имплементирован алгоритм DONF[5], на языке python с использованием библиотеки networkx, который осуществляет построения расписания в два этапа: определение приоритета у задач из множества готовых задач (те, у которых предшественники уже выполнены) и выбор машины, на которой задачи будут исполнены.

Пусть задан DAG $G = (V, E)$, $V = v_1, ..., v_n$ - множество вершин E - множество ребер. Ребрам и вершинам задан вес. Задано множество исполнителей $P = p_1, ..., p_k$. Поддерживается множество выполненных задач и множество задач, готовых к выполнению (задачи, предшественники которых находятся в множестве выполненных задач). Работа алгоритма состоит из двух этапов:

1. Выбор задачи из списка готовых задач с наибольшим WOD, определяемым как
$$WOD = \sum_{v_j \in succ(v_i)} \frac{1}{ID(v_j)}$$

где $ID(v_j)$ - входная степень вершины v_j, количество входящих в неё ребер; $succ(v_i)$ - множество наследников v_i

2. Выбирается исполнитель минимизирующий $EFT_{conflict}$:

$$EFT_{conflict}(v_i, p_j) = EST_{conflict}(v_i, p_j) + w_{ij}$$

$$EST_{conflict} = \max[T_{ava}(p_j) \max_{v_m \in pred(v_i)} [\max[AFT(v_m), T_{ava}(p_m, p_j)] + c_{m,i}]]$$

где $AFT(v_m)$ - время, когда v_m перешел в множество завершенных процессов, $T_{ava}(p_m, p_j)$ - время, когда соединение между p_m и p_j будет свободно, а p_m в данном случае исполнитель, ответственный за выполнение v_m; $c_{m,i}$ - стоимость коммуникации между вершинами.

Задача алгоритма состоит в минимизации SLR - scheduling lenght ratio, относительной длины расписания. Введем понятие makespan:

$$makespan = \max AFT(v_{exit})$$

makespan - время завершения последней задачи (вершины в графе без наследников, если таковых несколько - берется максимум по времени их завершения). Тогда можно определить SLR как

$$SLR = \frac{makespan(solution)}{\sum_{v_i \in CP_{MIN}} \min_{p_j \in P}(w_{(i,j)})}$$

где знаменатель представляет собой нижнюю границу длительности расписания [5]. Это сумма минимального времени выполнения задач v_i из критического пути CP_{MIN} исполнителем p_j, обозначаемого $w_{(i,j)}$. Критический путь в данном случае обозначает наиболее длинный путь в графе в смысле весов вершин, т.е. самый "тяжелый"путь от входной до выходной вершины. Таким образом, SLR показывает отличие полученного расписания от нижней границы, поэтому $SLR \geqslant 1$, и чем ближе он к 1, тем лучше полученное расписание.

Используются исполнители трех типов:

- Small - CPU 10 Gflops, Network Port 1562.2 mb/s

- Medium - CPU 100 Gflops, Network Port 3125 mb/s
- Large - CPU 1000 Gflops, Network Port 3125 mb/s

Тестирование имплементации DONF проводилось в конфигурации [DONF]: Small x 4 + Medium x 2 + Large x 4. Результат сравнения имплементации с данными из статьи по выборке из 48 графов с 1000 вершин:

Таблица 1: SLR оригинал и имплементация

Количество вершин	DONF статья	DONF имплементация
1000	21.6	19.06

К сожалению, авторы работы не приводят механизм расчета времени выполнения и времени передачи данных в зависимости от параметров исполнителя, поэтому для расчета использовался следующий подход: вес вершины делился на CPU показатель, вес ребра делился на Network Port показатель с целью определения длительности выполнения и длительности передачи данных между задачами соответственно - что скорее всего и послужило причиной для различия результатов между полученным SLR и приведенным в статье. Для графов с более 1000 вершин тестирование не проводилось в силу проблем с networkx (переполнение памяти), и в силу того что обучение на графах с более чем 1000 вершин занимает слишком много времени.

Algorithm 1. DONF Scheduling Algorithm

Input: DAG, set of processors P
Output: scheduling result
Init: put ready tasks into ready queue
repeat
  Get or compute WOD for ready tasks using (3) or (4) or (5)
  Pick the task n_i with maximum WOD value
  for $p_j \in P$ **do**
    Compute $EFT_{\text{conflict}}(n_i, p_j)$ using (6)
  end
  Assign task n_i to processor p_j that minimizes EFT_{conflict} of task n_i
  Update ready queue
until ready queue is empty;

Рис. 14: Псевдокод DONF[5]

2.3 Ray и RLlib

Ray - библиотека для распределенных вычислений, предоставляющая API и абстракции, с которыми возможно работать из различных языков программирования, таких как Python и Java. Ray поддерживает параллелизм, распределенное управление памятью, автоматическое масштабирование вычислений. Помимо этого, Ray позволяет построенным на его базе приложениям получать низкоуровневый контроль над ресурсами вычислительного устройства. Общая схема устройства Ray представлена на рисунке ниже.

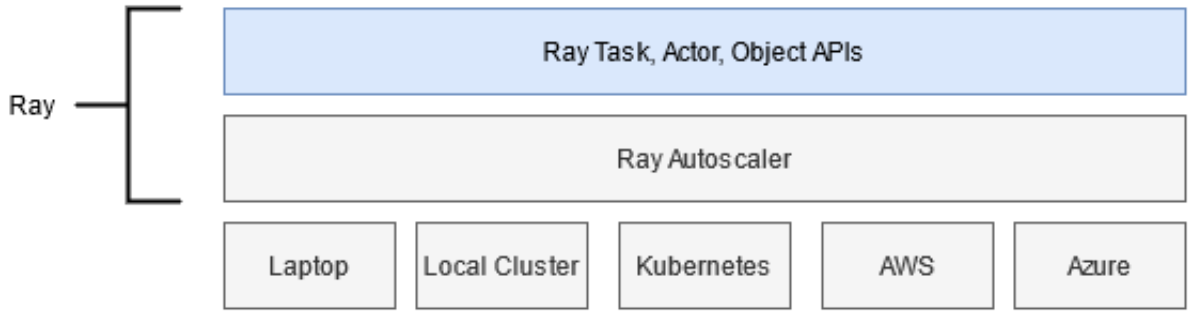


Рис. 15: Общая схема Ray [26]

Для выполнения работы использовалась библиотека RLlib - расширение Ray для обучения с подкреплением, в комбинации с OpenAI Gym - унифицированного интерфейса для создания среды для обучения с подкреплением.

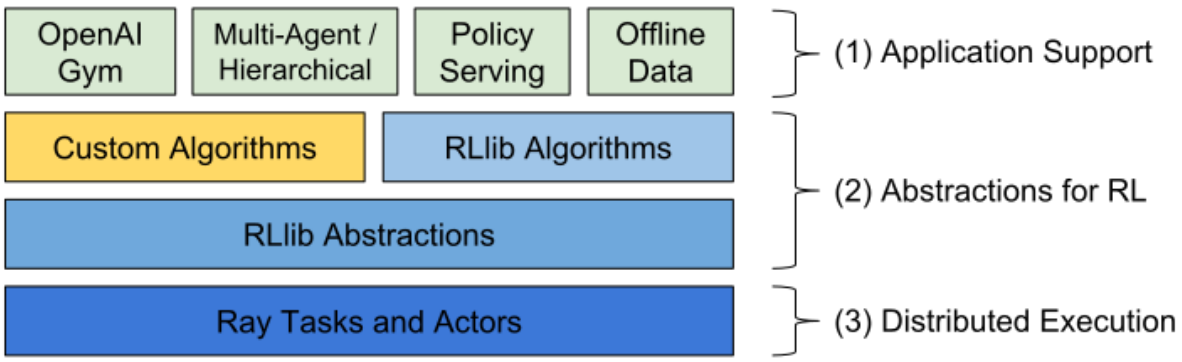


Рис. 16: Общая схема RLlib [27]

Суть данной системы заключается в разделении среды, алгоритма и используемого в нем агентов (как, например, в A3C) и абстракций для распределенных вычислений. Благодаря унифицированному интерфейсу Gym возможно приведение вывода симулирующей среду модели к виду, когда любой из реализованных в рамках RLlib Algorithms алгоритмов, будь то A3C или DQN, может быть исполнен на множестве устройств параллельно или распределенно, в зависимости от поставленной задачи, тем самым открывая возможности для масштабирования проводимых экспериментов. RLlib Algorithms содержит эталонные реализации классических алгоритмов обучения с подкреплением для двух наиболее популярных библиотек машинного обучения: TensorFlow и Torch. Реализации алгоритмов оптимизированы, и способны без дополнительных изменений быть выполнены на распределенных системах, поддерживающих также GPU и Multi-GPU. Это позволяет значительно снизить время, затрачиваемое на обучение моделей.

OpenAI Gym представляет собой интерфейс между симуляцией среды и агентом. Во время обучения, действия от агента переедаются в среду через метод step и получают отклик от неё (награду) в качестве результата. Действия и состояния среды реализованы через gym.spaces и имеют стандартизированное численное представление. Метод reset позволяет вернуть среду к изначальному состоянию, а метод close используется для завершения работы среды. Опциональный метод render служит для реализации возможности отрисовки среды (как, например, в случае, когда среда является видеоигрой).

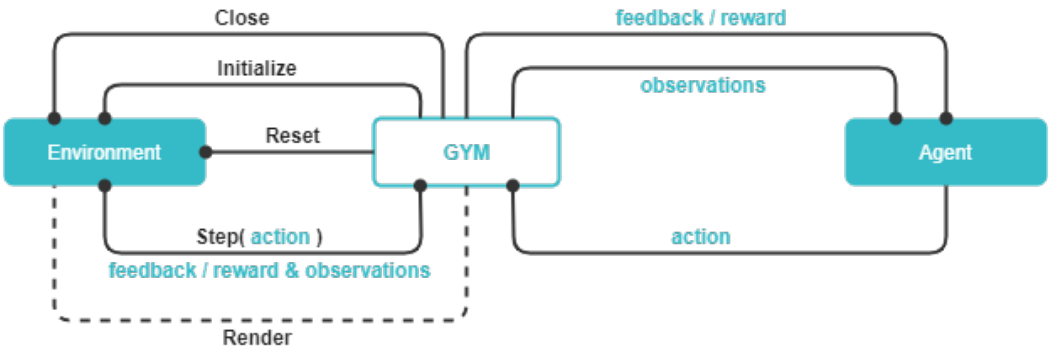


Рис. 17: Общая схема Gym [28]

Преимущество Gym в его стандартизации - появляется возможность реализовать среду, подходящую для любого алгоритма обучения с подкреплением, так как Gym это по сути и есть реализация парадигмы обучения с подкреплением, где агент исполняет роль RL алгоритма. В рамках данной работы среда была реализована через граф netwrokx, который передает свои параметры в Gym, а в качестве агента выступает алгоритм из RLlib Algorithms.

Глава 3. Реализация

3.1 Архитектура

Реализованная модель обучения с подкреплением включает в себя 4 основных компонента: state space (среда), action space (действия), reward (награда). В реализованной модели это:

- State space - матрица $N \times 3$, где каждый столбец соответствует задаче из множества готовых к выполнению задач, строки же представляют собой характеристики задач как вершин графа
  - взвешенная выходная вершина графа

$$WOD(v_i) = \sum_{v_j \in succ(v_i)} \frac{1}{InDegree(v_j)}$$

  - $OutDegree(v_i)$ - выходная степень вершины графа
  - $InDegree(v_i)$ - входная степень вершины графа

  State space строится следующим образом: сортируется множество готовых к выполнению задач по гиперпараметру (WOD в текущей реализации, сортировка по убыванию) и выбирается N первых элементов

- Action space - множество $\{1, 2, \ldots, N\}$, i-е действие означает, что следующим на исполнение будет подана работа, стоящая на i-м месте в State Space

- Reward - награда за текущий эпизод определяется как

$$R_t = SLR^{t-1}_{ready} - SLR^{t}_{ready}$$

$$SLR^{t}_{ready} = \frac{makespan^{t}_{complete}}{CP^{t}_{MIN_{complete}}}$$

  где $makespan^{t}_{complete}$ - длительность расписания, состоящего из уже выполненных задач на момент времени t; $CP^{t}_{MIN_{complete}}$ - CP_{MIN}, описанный в лаве 2, вычисленный на уже готовых задачах

Модель составляет расписания в два этапа:

- Выбор задачи, которая будет направлена исполнителю на выполнение посредством модели обучения с подкреплением
- Выбор исполнителя - выбирается тот, который минимизирует EFT, происходит перебором

$$EFT(v_i, P_j^{M_*} = EST(v_i, P_j^{M_*} + PD((v_i, P_j^{M_*})$$

$$EST(v_i, P_j^{M_*}) = \max\{T_{ava}(P_j^{M_*}), \max_{v_m \in pred(v_i)}\{AFT(v_m)\}\}$$

где $AFT(v_m)$ - время, когда задача перешла в множество готовых задач, $T_{ava}(P_j^{M_*})$ - время, когда исполнитель $P_j^{M_*}$ станет свободен

Процесс составления расписание останавливается, когда не остается готовых к выполнению задач. В качестве алгоритмов обучения с подкреплением использовались DQN и A3C в реализации RLlib Algorithms, для построение среды - OpenAI Gym.

3.2 Результаты

Эксперименты проводились на идентичном с реализованной версией алгоритма DONF наборе исполнителей. Из полученных 48 случайных графов в процессе генерации данных случайным образом было выбрано 16 графов с целью уменьшения времени, затрачиваемого на обучение моделей. Обучение происходило на облачном сервисе Google Colab, обладающего характеристиками:

- CPU: Intel Xenon 2.3 GHz
- RAM: 12 GB
- GPU: Nvidia K80/T4 12GB/16GB

Два вида GPU, как и не указанное количество ядер у CPU является следствием процесса динамического распределения ресурсов у Google Colab,

который зависит от текущей загруженности сервиса, доступного оборудования в регионе и прочих факторов.

В качестве метрики для сравнения используется средний SLR - среднее по SLR расписаний для каждого из графов в выборке. По результатам процесса обучения были получены следующие результаты:

Таблица 2: Результат работы модели

Алгоритм	Средний SLR
DONF	8.8201
A3C	6.6719
DQN	6.7518

Наилучший результат был получен при использовании A3C алгоритма. A3C показала результаты на $\approx 24\%$ превосходящие результат имплементацию DONF, описанную во второй главе.

В обучении с подкреплением основной показатель работоспособности модели - средняя награда за эпизод. Графики для DQN и A3C представлены ниже.

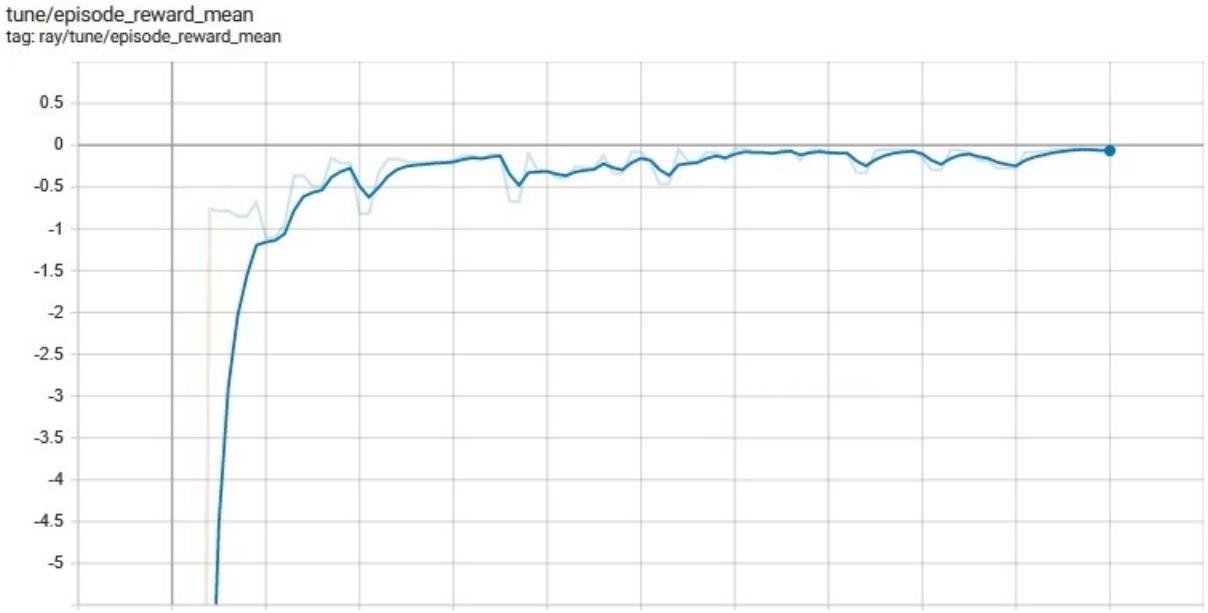


Рис. 18: График средней награды за эпизод DQN

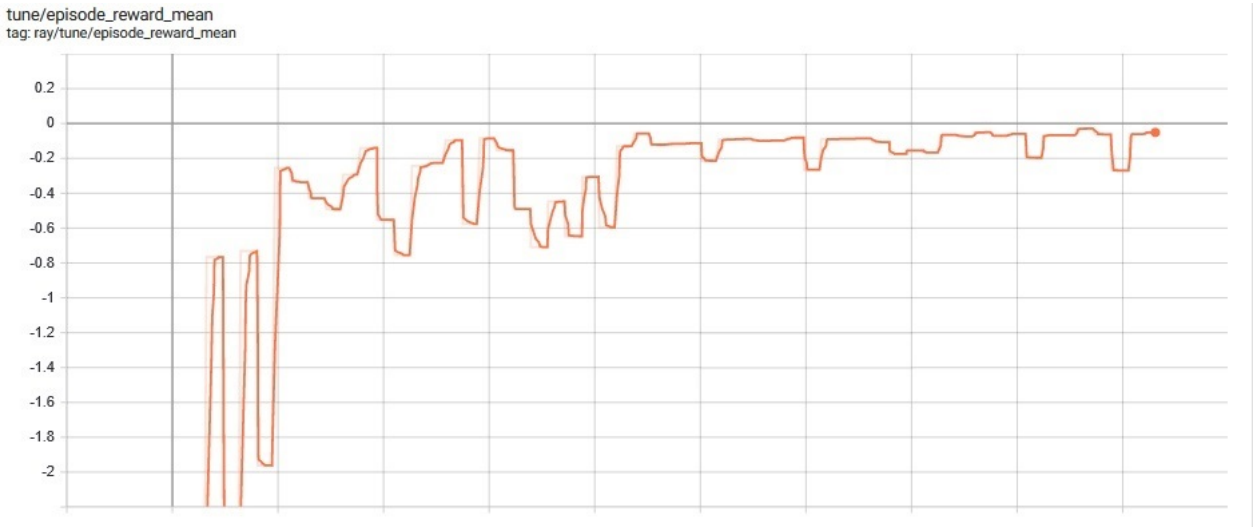


Рис. 19: График средней награды за эпизод АЗС

Существует проблема используемого механизма наград - чем больше размер графа, тем меньше награда. Такое затухание награды уменьшает способность к оптимизации управления на более поздних стадиях работы модели, когда уже большая часть задач выполнена. Однако даже при существующих недостатках, результаты работы модели превосходит классические эвристики, но остается пространство для улучшения модели.

Выводы

Многие, позволяющие эффективно отвечать на бросаемые современным миром вызовы, содержат в себе проблему распределения связанных между собой задач между исполнителями. В облачных вычислениях и зеленой энергетике такие проблемы стоят особенно остро.

Благодаря развитию обучения с подкреплением, подобную задачу становится возможно решать более эффективно, чем это было возможно до недавнего времени. Возможность RL моделей принимать решения в режиме реального времени при новых показателях среды позволяет им эффективно адаптироваться и использовать закономерности, которые могли быть не замечены создателями обычных алгоритмов.

Заключение

Поставленная задача выполнена полностью. Были рассмотрены подходы к решению проблемы построения расписания на графах, на основе рассмотренных подходов сделан выбор в пользу методов обучения с подкреплением. Была реализована модель обучения с подкреплением, решающая задачу минимизации относительной длительности расписания на графах. Был выбран алгоритм для сравнения из классических эвристик для построения расписания на графах, сгенерированы данные для тестирования. Проведенное сравнение показало, что построенная в рамках работы модель превосходит классические эвристики. Модель, основанная на обучении с подкреплением выполняет свою задачу и обладает потенциалом к масштабированию.

Список литературы

[1] We're not prepared for the end of Moore's Law // 2020
URL:https://www.technologyreview.com/2020/02/24/905789/were-not-prepared-for-the-end-of-moores-law/ (дата обращения: 14.02.2021)

[2] Cloud Computing Market Worth // 2020
URL:https://www.globenewswire.com/news-release/2020/10/01/2102033/0/en/Cloud-Computing-Market-Worth-760-98-Billion-at-18-6-CAGR-Tech-Giants-Such-as-IBM-and-Microsoft-to-Focus-on-Introducing-Advanced-Cloud-Solutions-Fortune-Business-Insights.html
(дата обращения: 14.02.2021)

[3] Shyalika, C., Silva, T. Karunananda, A. Reinforcement Learning in Dynamic Task Scheduling: A Review // SN COMPUT. SCI. 1, 2020, pp. 306.

[4] Fares, R., Webber, M. The impacts of storing solar energy in the home to reduce reliance on the utility // Nat Energy 2, 17001, 2017.

[5] Lin, H., Li, MF., Jia, CF. et al. Degree-of-Node Task Scheduling of Fine-Grained Parallel Programs on Heterogeneous Systems // J. Comput. Sci. Technol. 34, 2019, pp. 1096–1108.

[6] Topcuouglu H, Hariri S, Wu M Y. Performance-effective and low-complexity task scheduling for heterogeneous computing // IEEE Trans. Parallel Distrib. Syst., 13(3), 2002, pp. 260-274.

[7] Arabnejad H, Barbosa J G. List scheduling algorithm for heterogeneous systems by an optimistic cost table // IEEE Trans. Parallel and Distributed Systems, 25(3), 2014, pp. 682-694.

[8] Bittencourt L F, Sakellariou R, Madeira E R M. DAG scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm // In Proc. the 18th Euromicro Int. Conf. Parallel, Distributed and Network-Based Processing, 2010, pp.27-34.

[9] M. Divyaprabha, V. Priyadharshni and V. Kalpana, Modified Heft Algorithm for Workflow Scheduling in Cloud Computing Environment // 2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT), 2018, pp. 812-815.

[10] Fei, Yin, D. Xiaoli, Jiang Chang-jun and Deng Rong., Directed Acyclic Task Graph Scheduling for Heterogeneous Computing Systems by Dynamic Critical Path Duplication Algorithm // Journal of Algorithms and Computational Technology 3, 2009, pp. 247-270.

[11] Bozinovski, S. A self learning system using secondary reinforcement // Cybernetics and Systems Research: Proceedings of the Sixth European Meeting on Cybernetics and Systems Research, 1982, pp. 397–402.

[12] Degris, T., Martha White and R. Sutton. "Off-Policy Actor-Critic." ArXiv, 2012, abs/1205.4839.

[13] Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning // Nature 518, 2015, pp. 529–533.

[14] Mnih, V., Adrià Puigdomènech Badia, Mehdi Mirza, A. Graves, T. Lillicrap, Tim Harley, D. Silver and K. Kavukcuoglu. // Asynchronous Methods for Deep Reinforcement Learning, ArXiv, 2016, abs/1602.01783.

[15] Baheri, Betis and Qiang Guan, MARS: Multi-Scalable Actor-Critic Reinforcement Learning Scheduler // ArXiv, 2020, abs/2005.01584.

[16] Mao, Hongzi, M. Schwarzkopf, S. Venkatakrishnan, Zili Meng and M. Alizadeh, Learning scheduling algorithms for data processing clusters // Proceedings of the ACM Special Interest Group on Data Communication, 2019.

[17] Silver, D., Huang, A., Maddison, C. et al. Mastering the game of Go with deep neural networks and tree search // Nature 529, 2016, pp. 484–489.

[18] Carastan-Santos, Danilo and R. D. Camargo // Obtaining dynamic scheduling policies with simulation and machine learning, Proceedings of

the International Conference for High Performance Computing, Networking, Storage and Analysis, 2017.

[19] Grandl, Robert, Srikanth Kandula, S. Rao, Aditya Akella and Janardhan Kulkarni // GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters, OSDI, 2016.

[20] Thiam P., Kessler V., Schwenker F. // A Reinforcement Learning Algorithm to Train a Tetris Playing Agent, Lecture Notes in Computer Science, vol 8774. Springer, Cham, 2014.

[21] Sutton R.S., Barto A.G. // Reinforcement Learning: An Introduction, MIT Press, Cambridge, MA, 2018.

[22] Williams, R. J. // Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning, Machine Learning 8, 1992, pp. 229-256.

[23] Recht, B., Christopher Ré, Stephen J. Wright and Feng Niu. // Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent, NIPS, 2011.

[24] Daggen // 2013
URL:https://github.com/frs69wq/daggen/ (дата обращения: 09.03.2021)

[25] Guan Wang, Yuxin Wang, Hui Liu, He Guo // HSIP: A Novel Task Scheduling Algorithm for Heterogeneous Computing Scientific Programming, vol. 2016, Article ID 3676149, 2016, pp. 5-8.

[26] Ray // 2021
URL:https://docs.ray.io/en/master/index.html (дата обращения: 24.02.2021)

[27] Liang, Eric, Richard Liaw, P. Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, J. Gonzalez, Michael I. Jordan and I. Stoica // RLlib: A Framework for Distributed Reinforcement Learning, ArXiv, 2017, abs/1712.09381.

[28] Brockman, Greg, Vicki Cheung, Ludwig Pettersson, J. Schneider, John Schulman, Jie Tang and W. Zaremba // OpenAI Gym, ArXiv, 2016, abs/1606.01540.

Приложение

/donf.py

```
import networkx


class Donf:
    def __init__(self, scheduling_model, workflow,
      ↪ depth_level=1, alpha=0.5):
        self.scheduling_model = scheduling_model
        self.workflow = workflow
        self.depth_level = depth_level
        self.alpha = alpha
        self.parameter_dict = {}
        self.est = None
        self.wod_dict = {}
        for processor in scheduling_model.processors.
          ↪ keys():
            self.parameter_dict[processor] = {task: {}
              ↪  for task in self.workflow.nodes}
        self.calculate_wod()

    def algorithm_body(self):

        max_wod_list = [(task, self.wod_dict[task])
          ↪ for task in self.scheduling_model.
          ↪ ready_queue]
        task = max(max_wod_list, key=lambda x: x[-1])
          ↪ [0]

        processor_eft_list = [
            (processor, *self.calculate_eft_conflict(
              ↪ task, processor)) for processor in
```

```
            ↪ self.scheduling_model.processors
        ]

        processor, eft_conflict, est = min(
            ↪ processor_eft_list, key=lambda x: x[1])

        return task, processor, est

    def wod_for_env(self, task, depth_level):
        immediate_successors = set(self.workflow.
            ↪ successors(task))
        descendants = set(networkx.
            ↪ single_source_dijkstra_path_length(
            self.workflow,
            source=task,
            cutoff=depth_level
        ))
        descendants = descendants.difference(
            ↪ immediate_successors)
        descendants.remove(task)
        return sum([1 / self.workflow.in_degree(
            ↪ successors) for successors in
            ↪ immediate_successors]) + \
            self.alpha * sum([1 / self.workflow.
                ↪ in_degree(descendant) for descendant
                ↪ in descendants])



    def calculate_wod(self):
        for task in self.workflow.nodes:
            immediate_successors = set(self.workflow.
                ↪ successors(task))
```

```
            descendants = set(networkx.
              ↪ single_source_dijkstra_path_length(
                self.workflow,
                source=task,
                cutoff=self.depth_level
            ))
            descendants = descendants.difference(
              ↪ immediate_successors)
            descendants.remove(task)
            self.wod_dict[task] = \
                sum([1/self.workflow.in_degree(
                  ↪ successors) for successors in
                  ↪ immediate_successors]) + \
                self.alpha * sum([1/self.workflow.
                  ↪ in_degree(descendant) for
                  ↪ descendant in descendants])

    def calculate_eft_conflict(self, task, processor):
        eft_conflict = self.calculate_est_conflict(
          ↪ task, processor)
        network_finish_time = processor.
          ↪ calculate_network_finish_time(task)
        cpu_finish_time = processor.
          ↪ calculate_network_finish_time(task)
        return eft_conflict + network_finish_time +
          ↪ cpu_finish_time, self.
          ↪ calculate_est_conflict(task, processor)

    def calculate_est_conflict(self, task, processor):
        predecessors = self.scheduling_model.
          ↪ predecessors[task]
        if len(predecessors) > 0:
            predecessors_aft = [
```

```
                (
                    predecessor,
                    self.get_processor_of_task(
                      ↪ predecessor).
                      ↪ task_finish_time_dict[
                      ↪ predecessor]
                ) for predecessor in predecessors
            ]
            max_aft_cmi = max(predecessors_aft, key=
              ↪ lambda x: x[-1])
            if processor.aft[-1] > max_aft_cmi[-1]:
                return processor.aft[-1]
            else:
                return max_aft_cmi[-1]
        else:
            return processor.aft[-1]

    def get_processor_of_task(self, task):
        processor = self.scheduling_model.
          ↪ task_processor_pair[task]
        return processor

    def processors_aval_time(self):
        return 0
```

/schedule.py

```
import networkx
import donf
import glob
import json
from tqdm import tqdm
import numpy as np


class Processor:
    def __init__(self, schedule, processor_index,
      ↪ processor_type: str = 'small'):
        self.processor_type = processor_type
        if self.processor_type == 'small':
            self.computing_speed = 10  # GFlops
            self.ram_size = 1  # GB
            self.memory_bandwidth = 1085
            self.network_port = 1562.5
        elif self.processor_type == 'medium':
            self.computing_speed = 100
            self.ram_size = 1
            self.memory_bandwidth = 1310
            self.network_port = 3125
        elif self.processor_type == 'large':
            self.computing_speed = 1000
            self.ram_size = 2
            self.memory_bandwidth = 1310
            self.network_port = 3125

        self.machine_queue = []
        self.aft = [0]
        self.earliest_ready_time = 0
```

```
        self.schedule = schedule
        self.processor_index = processor_index
        self.task_finish_time_dict = {}

    def __repr__(self):
        return '{}_{}'.format(self.processor_type,
          ↪ self.processor_index)

    def task_assignment(self, task: object, est=0):
        self.machine_queue.append(task)
        if est > self.aft[-1]:
            self.aft.append(est)
        else:
            self.aft.append(
                self.aft[-1] + self.
                  ↪ calculate_network_finish_time(
                  ↪ task) + self.
                  ↪ calculate_cpu_finish_time(task)
            )

        self.earliest_ready_time = self.aft[-1]

        self.task_finish_time_dict[task] = self.aft
          ↪ [-1]

    def calculate_network_finish_time(self, task):
        # get all predecessors (edges from graph)
        # calculate mem bandwidth and network port
          ↪ time to
        # transfer all data
        network_cost = 0
        for predecessor in self.schedule.predecessors[
          ↪ task]:
```

```
                network_cost += float(self.schedule.
                    ↪ workflow.get_edge_data(predecessor,
                    ↪ task)[0]['size'].strip('"'))
            if network_cost > 0:
                return network_cost / self.
                    ↪ memory_bandwidth
            else:
                return 0

        def calculate_cpu_finish_time(self, task) -> float
            ↪ :
            return float(self.schedule.workflow._node[task
                ↪ ]['size'].strip('"')) / self.
                ↪ computing_speed


    class Schedule:
        def __init__(self, workflow: object, scheduler:
            ↪ callable(object), processor_stack: dict,
            ↪ processor_class: object):
            self.workflow_file = workflow
            self.workflow = networkx.drawing.nx_pydot.
                ↪ read_dot(self.workflow_file)
            self.graph_normalization()
            self.processor_stack = processor_stack
            self.ready_queue = set()
            self.complete_queue = set()
            self.processors = {}
            self.dag_params = {}
            self.predecessors = {}
            self.task_processor_pair = {}
            self.processor_class = processor_class
            self.initial_tasks()
```

```
        self.create_processors()
        self.get_dag_params()
        self.fill_in_predecessors_dict()
        self.scheduler = scheduler(self, self.workflow
          ↪ , 2, 0.5)

    def fill_in_predecessors_dict(self):
        self.predecessors = {node: set(self.workflow.
          ↪ predecessors(node)) for node in self.
          ↪ workflow.nodes}

    def check_readiness(self, task) -> bool:
        # check if predecessors of a task are
          ↪ completed
        if self.predecessors[task].issubset(self.
          ↪ complete_queue):
            return True
        else:
            return False

    def update_ready_queue(self):
        for complete_task in self.complete_queue:
            for successors in self.workflow.successors
              ↪ (complete_task):
                if self.check_readiness(successors)
                  ↪ and successors not in self.
                  ↪ complete_queue:
                    self.ready_queue.add(successors)

    def schedule_task(self) -> None:
        task, processor, est = self.scheduler.
          ↪ algorithm_body()
        processor.task_assignment(task, est)
```

```
        self.processors[processor].add(task)
        self.ready_queue.remove(task)
        self.complete_queue.add(task)
        self.task_processor_pair[task] = processor
        self.update_ready_queue()

    def initial_tasks(self) -> None:
        # getting nodes with 0 in-degree to fill in
          ↪ ready queue
        for node in self.workflow.nodes:
            if self.workflow.in_degree(node) == 0:
                self.ready_queue.add(node)

    def create_processors(self) -> None:
        # initializing cluster with predefined number
          ↪ of processors
        for key, value in self.processor_stack.items()
          ↪ :
            for i in range(value):
                self.processors[self.processor_class(
                  ↪ self, i, key)] = set()

    def get_predecessors(self, task) -> set:
        return self.predecessors[task]

    def get_size(self, task) -> tuple:
        return task, float(self.workflow._node[task]['
          ↪ size'].strip('"'))

    def calculate_minimal_processing_time(self, task)
      ↪ -> float:
        return min([processor.
          ↪ calculate_cpu_finish_time(task) for
```

```
        ↪ processor in self.processors.keys()])

    def slr(self) -> float:
        makespan = self.get_makespan()
        critical_path = networkx.dag_longest_path(self
            ↪ .workflow, weight='size')
        cp_min = sum([self.
            ↪ calculate_minimal_processing_time(task)
            ↪ for task in critical_path])
        self.dag_params[self.workflow_file]['slr'] =
            ↪ makespan / cp_min
        return makespan / cp_min

    def get_makespan(self) -> float:
        aft = [(processor, processor.aft[-1]) for
            ↪ processor in self.processors.keys()]
        return max(aft, key=lambda x: x[-1])[-1]

    def single_source_longest_dag_path_length(self):
        s = [task for task in self.workflow.nodes if
            ↪ len(self.get_predecessors(task)) == 0][1]
        dist = dict.fromkeys(self.workflow.nodes, -
            ↪ float('inf'))
        dist[s] = 0
        topological_order = networkx.topological_sort(
            ↪ self.workflow)
        for n in topological_order:
            for sc in self.workflow.successors(n):
                if dist[sc] < dist[n] + self.get_size(
                    ↪ sc)[-1]:
                    dist[sc] = dist[n] + self.get_size
                        ↪ (sc)[-1]
        return dist
```

```
    def get_dag_params(self):
        with open(self.workflow_file) as f:
            data = f.readlines()
        params = list(filter(None, data[1].split('-'))
          ↪ )[1:-2]
        self.dag_params = {self.workflow_file: {}}
        for parameter in params:
            key, value = parameter.split(' ')[:-1]
            self.dag_params[self.workflow_file][key] =
              ↪  value

    def graph_normalization(self):
        for node in self.workflow.nodes:
            sz = self.workflow._node[node]['size'].
              ↪ strip('"')
            if sz == 0:
                self.workflow._node[node]['size'] = '
                  ↪ "{}"'.format(1)
            else:
                self.workflow._node[node]['size'] = '
                  ↪ "{}"'.format(float(sz) * 10 ** (-
                  ↪ len(sz) + 1))
        for _ in self.workflow.edges():
            eg = self.workflow[_[0]][_[1]][0]['size'].
              ↪ strip('"')
            self.workflow[_[0]][_[1]][0]['size'] = '
              ↪ "{}"'.format(float(eg) * 10 ** (-len(
              ↪ eg) + 1))


def get_avg_slr(file_name):
    with open(file_name) as json_file:
```

```
            data = json.load(json_file)
        s, count = 0, 0
        for machine in data.keys():
            for graph in data[machine].keys():
                s += data[machine][graph]['slr']
                count += 1
        return s / count


def test(case=3):
    p_s = {'small': 2, 'medium': 2}
    p_s1 = {'small': 4, 'medium': 2, 'large': 2}
    p_s2 = {'small': 8, 'medium': 4, 'large': 4}
    machines = [(p_s, 'p_s'), (p_s1, 'p_s1'), (p_s2, '
      ↪ p_s2')]
    dict1000 = {'p_s': {}, 'p_s1': {}, 'p_s2': {}}

    ready_queue_max_size = 0
    if case == 1:
        for machine in machines:
            for file in tqdm(glob.glob('1000/*')
              ↪ [::10]):
                sc = Schedule(file, donf.Donf, machine
                  ↪ [0], Processor)
                for _ in tqdm(range(len(list(sc.
                  ↪ workflow.nodes)))):
                    sc.schedule_task()
                    if len(sc.ready_queue) >
                      ↪ ready_queue_max_size:
                        ready_queue_max_size = len(sc.
                          ↪ ready_queue)
                sc.slr()
                dict1000[machine[-1]].update(sc.
```

```
                    ↪ dag_params)
    elif case == 2:
        for machine in machines:
            sc = Schedule('1000/100058228.dot', donf.
              ↪ Donf, machine[0], Processor)
            for _ in tqdm(range(len(list(sc.workflow.
              ↪ nodes)))):
                sc.schedule_task()
            for _ in sc.processors:
                print(_, _.machine_queue)
            print(sc.slr())
    else:
        slr = []
        for file in tqdm(glob.glob('1000_for_dqn/*')):
            sc = Schedule(file, donf.Donf, p_s2,
              ↪ Processor)
            while len(sc.ready_queue) > 0:
                sc.schedule_task()
            slr.append(sc.slr())

    print(np.mean(slr))

    # with open('1000_reimplemented.json', 'a') as
      ↪ outfile:
            #     json.dump(dict1000, outfile)
```

/train.py

```
import donf
import gym
import ray
import donf
from gym import spaces
from ray import tune
```

```
from ray.tune import run_experiments
from ray.tune.registry import register_env
from ray.rllib.agents.callbacks import
  ↪ DefaultCallbacks
from ray.rllib.env import BaseEnv
from ray.rllib.evaluation import MultiAgentEpisode,
  ↪ RolloutWorker
from ray.rllib.policy import Policy
from ray.rllib.policy.sample_batch import SampleBatch
from schedule import Schedule, Processor

class MyEnv(gym.Env):

    def __init__(self, scheduling_model, scheduler,
      ↪ graph, processor, state_space_size = 15):
        self.scheduling_model = scheduling_model
        self.scheduler_into_model = scheduler
        self.state_space_size = state_space_size
        self.processor = processor
        self.graph = graph
        self.action_space = spaces.Discrete(15)
        self.observation_space  = spaces.Box(low=-1,
          ↪ high=1000, shape=(self.state_space_size
          ↪ ,1), dtype=np.int32)
        self.backlog = set()
        self.action_taken_list = [1]
        self.trial = 0
        self.info = {'slr':[], 'reward':[], '
          ↪ finish_slr':[]}
        self.custom_metric = 0
        self.step_number = 0
        self.last_step_slr = 1
        self.scheduler = self.scheduling_model(self.
```

```
        ↪ graph, self.scheduler_into_model,{'small'
        ↪ : 2, 'medium': 2}, self.processor)


def step(self, action):
    done = False
    obs = self.observe(str(action))

    if len(self.scheduler.ready_queue) == 0:
        done = True
        self.info['finish_slr'].append(self.
          ↪ scheduler.slr())


    for task in obs:
        if task[0] != -1:
            self.scheduler.schedule_task(self.
              ↪ scheduler.workflow.vertex(str(
              ↪ task[0])))
        else:
            break
    info = {}
    reward = self.get_score()
    self.step_number += 1
    self.info['slr'].append(self.last_step_slr)
    self.info['reward'].append(reward)
    self.kek += 1
    self.info.update(info)
    return obs, reward, done, info

def get_score(self):
    makespan = self.scheduler.get_makespan()
    cp_min = self.scheduler.cp_min_processing_time
```

```
        ↪ (partial = True)

    slr = 0.0

    if cp_min != 0:
        slr = makespan/cp_min

    rw = self.last_step_slr - slr

    self.last_step_slr = slr

    return rw


def reset(self):
    with open('res1.json', 'w') as f:
        json.dump(self.info, f)

    self.scheduler = self.scheduling_model(self.
        ↪ graph, self.scheduler_into_model, {'small
        ↪ ': 2, 'medium': 2}, self.processor)
    return self.observe(0)

def observe(self, action):
    wod = [(task, self.scheduler.scheduler.
        ↪ wod_method(task, 5)) for task in self.
        ↪ scheduler.ready_queue]
    wod = sorted(wod, key=lambda x: x[-1], reverse
        ↪ =True)
    if len(wod)<15:
        while len(wod) < 15:
            wod.append((-1,0))
    return np.asarray([task[0] for task in wod
```

```
        ↪ ][:15]).astype(np.int32).reshape(15,1)


class MyCallbacks(DefaultCallbacks):
    def on_episode_start(self, *, worker:
      ↪ RolloutWorker, base_env: BaseEnv, policies:
      ↪ Dict[str, Policy],episode:MultiAgentEpisode,
      ↪ env_index: int, **kwargs): print("episode␣{}␣
      ↪ (env-idx={})␣started.".format(episode.
      ↪ episode_id, env_index))
    episode.user_data["slr"] = []

    def on_episode_step(self, *, worker: RolloutWorker
      ↪ , base_env: BaseEnv, episode:
      ↪ MultiAgentEpisode, env_index: int, **kwargs):
        slr = base_env.get_unwrapped()[0].
          ↪ last_step_slr
        episode.user_data["slr"].append(slr)

    def on_episode_end(self, *, worker: RolloutWorker,
      ↪  base_env: BaseEnv,
                       policies: Dict[str, Policy],
                         ↪ episode: MultiAgentEpisode
                         ↪ ,
                       env_index: int, **kwargs):
        episode.custom_metrics["final␣slr"] = episode.
          ↪ user_data["slr"][-1]
        print(episode.user_data["slr"])

def env_creator(p):
    return gym_env.MyEnv(Schedule, donf.Donf, '/
      ↪ content/test.graphml', Processor)
```

```
register_env("my_env1", env_creator)

drm_config = {
        "env": "my_env1",
        "num_workers": 1,
        "num_gpus": 1,
        "callbacks": MyCallbacks
    }


rs1 = tune.run(
    "A3C",
    stop={"training_iteration": 100},
    config = drm_config
).trials
```